数据结构与算法

第五章　　树

任课教员：张　铭

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

mzhang@db.pku.edu.cn

北京大学信息科学与技术学院

网络与信息系统研究所

主要内容

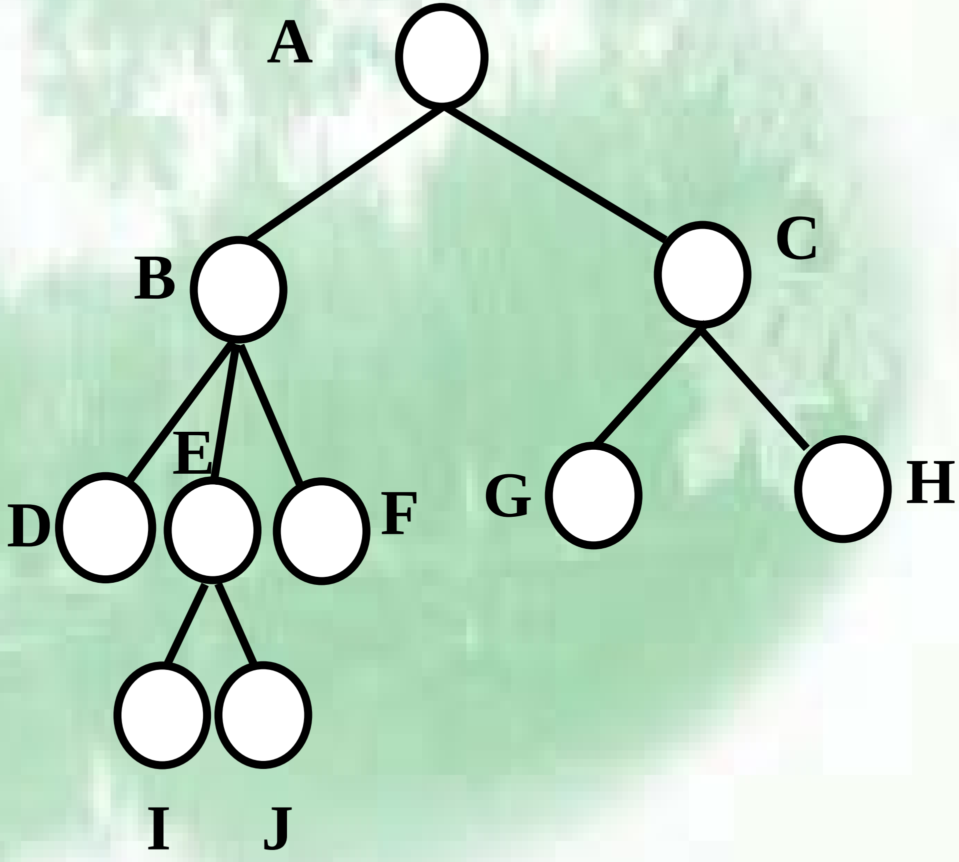
- 5.1 树的概念
- 5.2 树的链式存储
- 5.3 树的顺序存储
- 5.4 K叉树
- 补充 树计数

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

5.1.1 树和森林

- 树的逻辑结构
- 树形结构的各种表示法
- 树的定义和概念
- 森林的定义



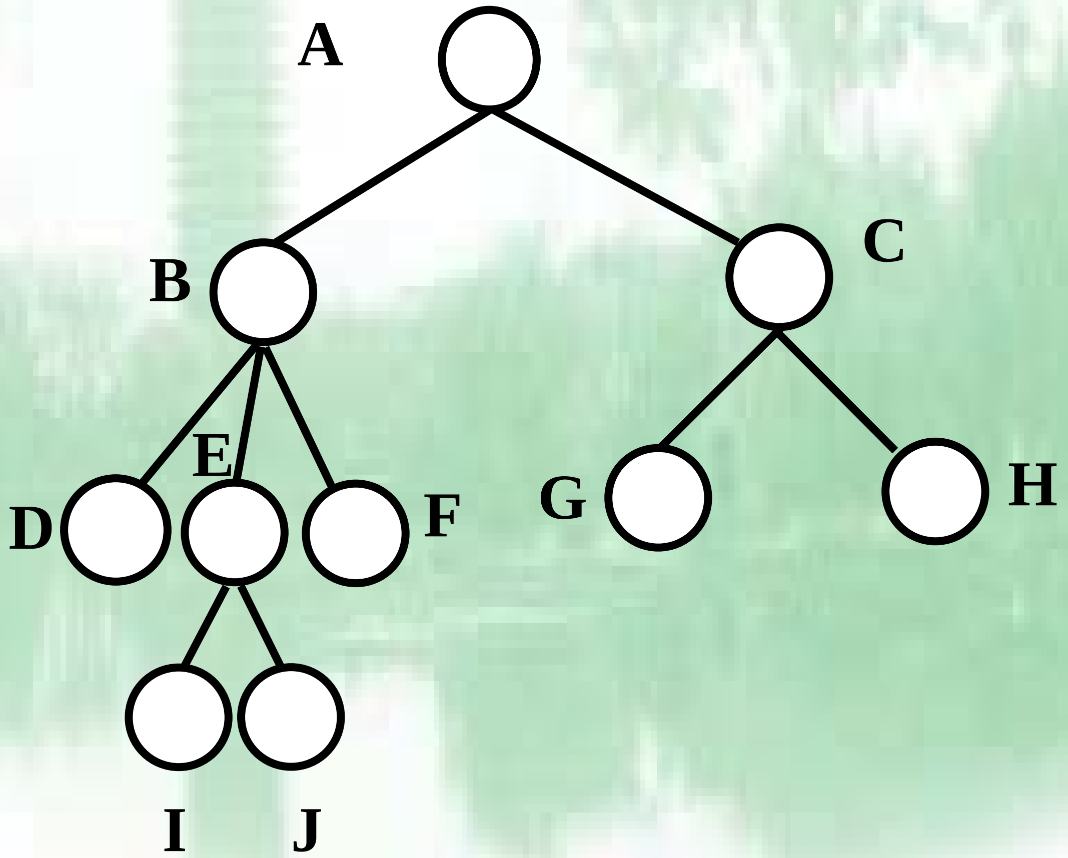
树的逻辑结构

- 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，…，k_s，使得k_0就是树根，且$k_s = k$，其中有序对$<k_{i-1}, k_i> \in N (1 \leqslant i \leqslant s)$。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

- 树形表示法
- 形式语言表示法
- 文氏图表示法
- 凹入表表示法
- 嵌套括号表示法

树形表示法



（a）树形表示法

形式语言表示法

树的逻辑结构是：

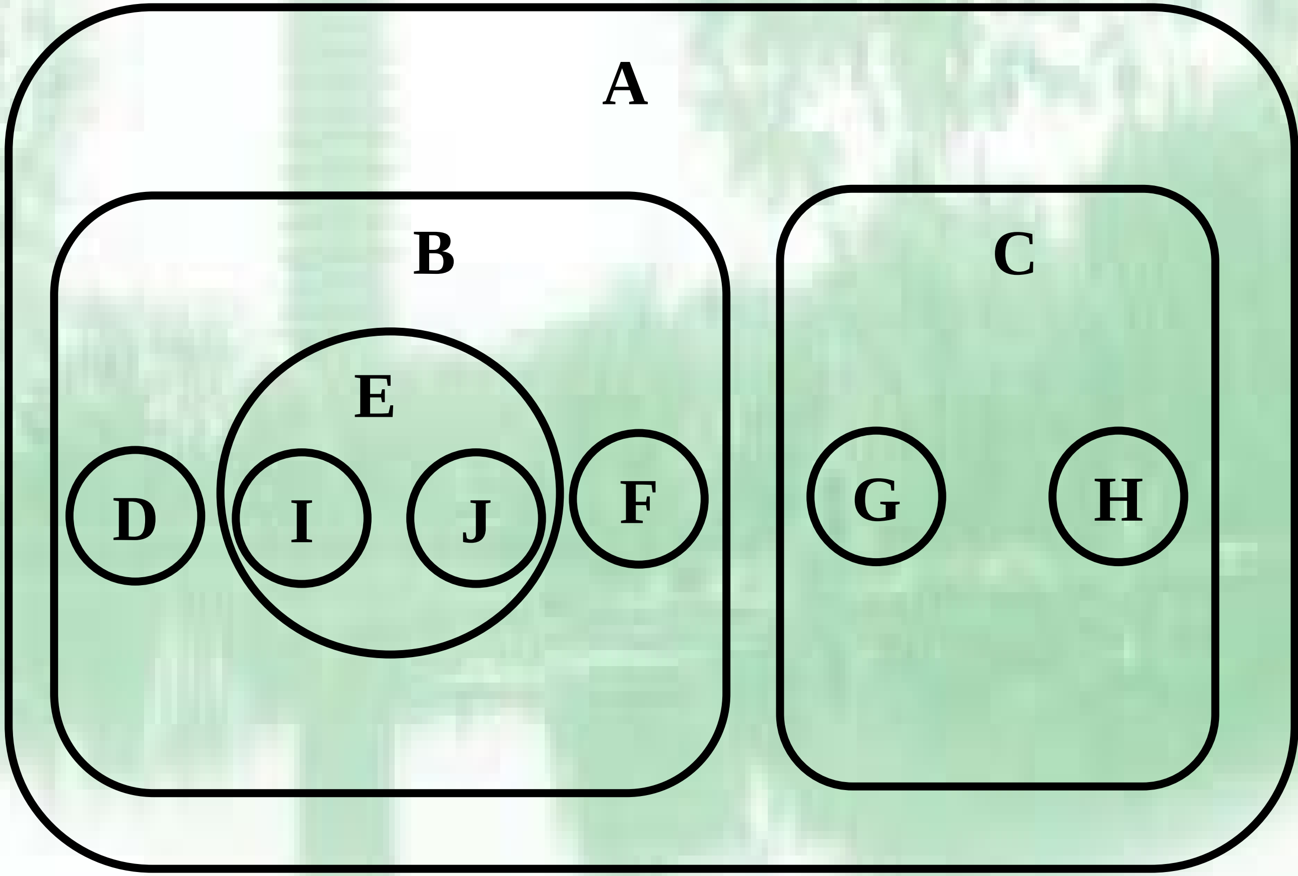
结点集合K={A，B，C，D，E，F，G，H，I，J}

K上的关系N={<A，B>，<A，C>，<B，D>，

<B，E>，<B，F>，<C，G>，

<C，H>，<E，I>，<E，J>}

文氏图表示法



（b）文氏图表示法

凹入表表示法

（c）凹入表表示法

5 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

5.2 树的链式存储

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 父指针表示法及等价类的并查算法

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

5.4 K叉树

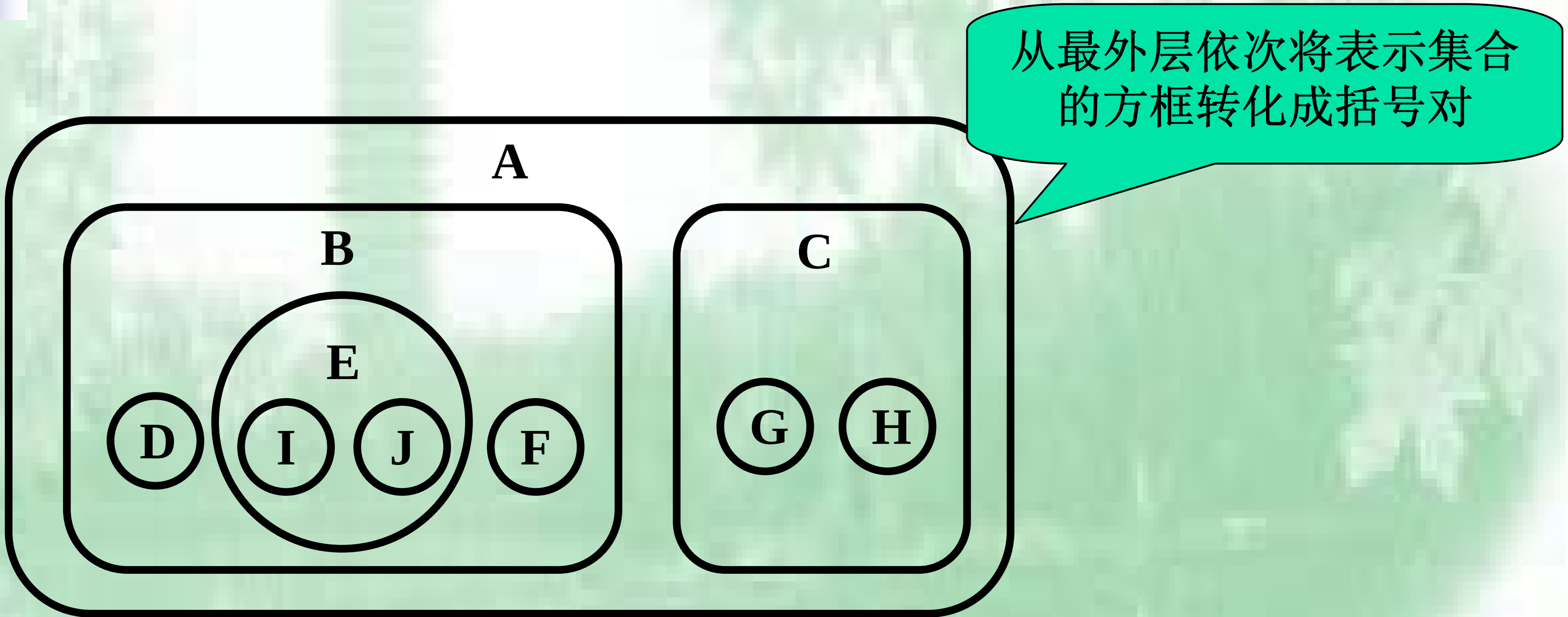
- 图书目录，杜威表示法

嵌套括号表示法

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

（**d**）嵌套括号表示法

文氏图到嵌套括号表示的转化



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

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，…，T_m，而且这些集合的每一个又都是树。树T_1，T_2，…，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(1)

- 若<k，k′ >∈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<k，k′ >及<k，k″ >∈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(2)

- 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(3)

- 有序树　在树T中如果子树T_1，T_2，…，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

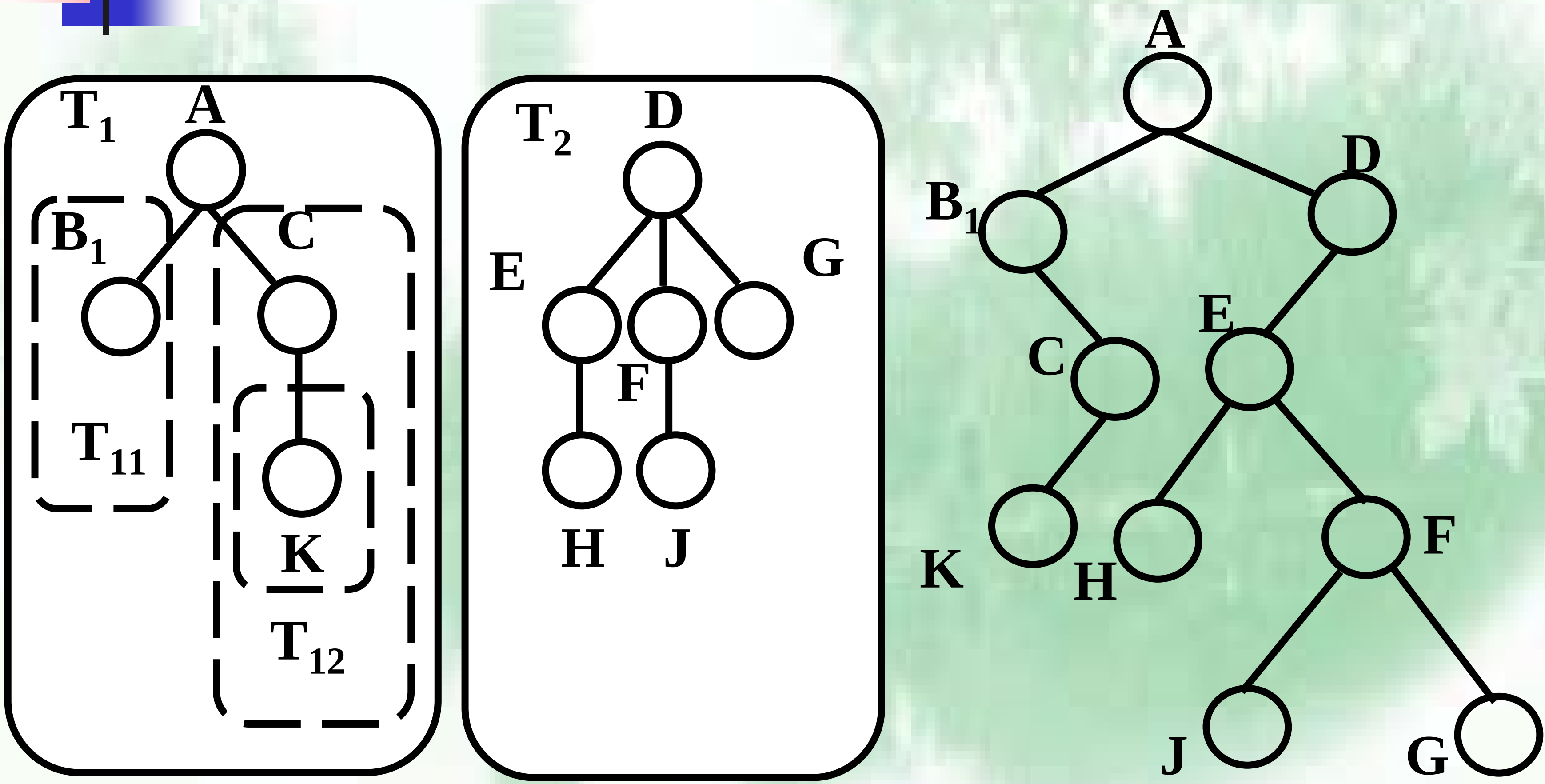
5.1.2 森林与二叉树的等价转换

- 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)。
  - 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(续)

- 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

图示：森林与二叉树



形式化：森林到二叉树

- 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W_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$W_1$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

形式化：二叉树到森林

- 设B是一棵二叉树，rt是B的根，L是rt的左子树，R是rt的右子树，则对应于B的森林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rt，rt的子树为F(L)

5.1.3 树/森林的结点ADT(1)

```
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/森林的结点ADT(2)

```
 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ADT(1)

```
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();
```

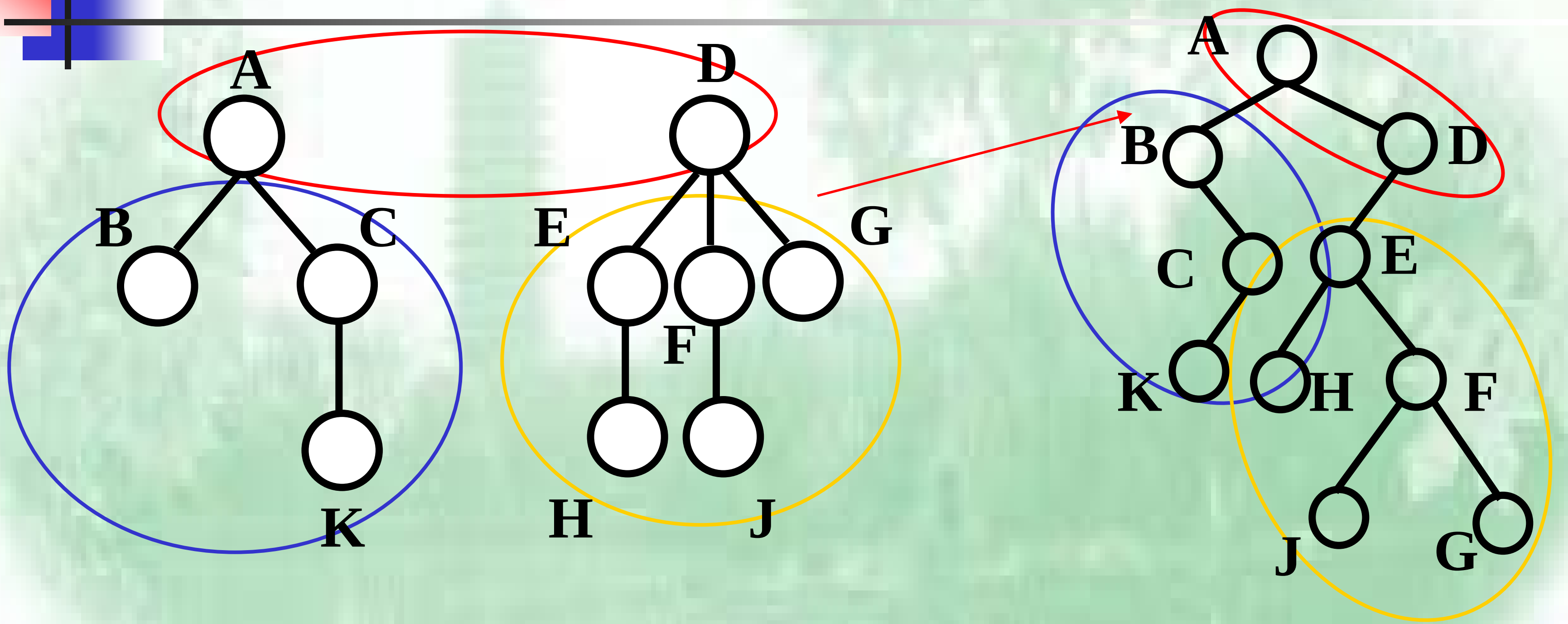
树/森林的ADT(2)

```
//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);
//广度优先周游树
void WidthTraverse(TreeNode<T>* root);
};
```


5.1.4 森林的周游

- 按深度方向周游
  - 先根次序
  - 后根次序
- 按广度方向周游
  - 宽度优先周游
  - 层次周游

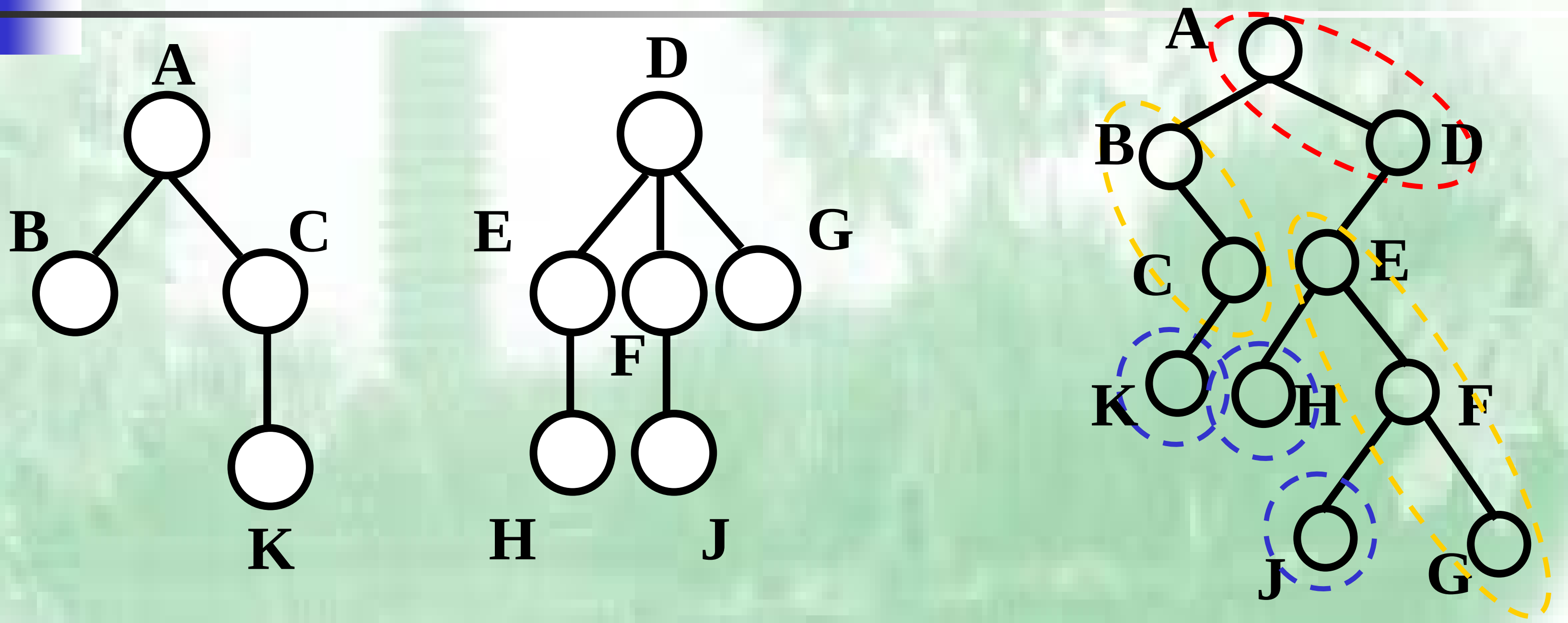
森林的周游



周游森林vs周游二叉树

- 先根次序周游森林
  - 前序法周游二叉树
- 后根次序周游森林
  - 按中序法周游对应的二叉树
- 中根周游？
  - 无法明确规定根在哪两个子结点之间

补充：一种广度优先周游森林



- **PrevSibling()**函数采用本框架

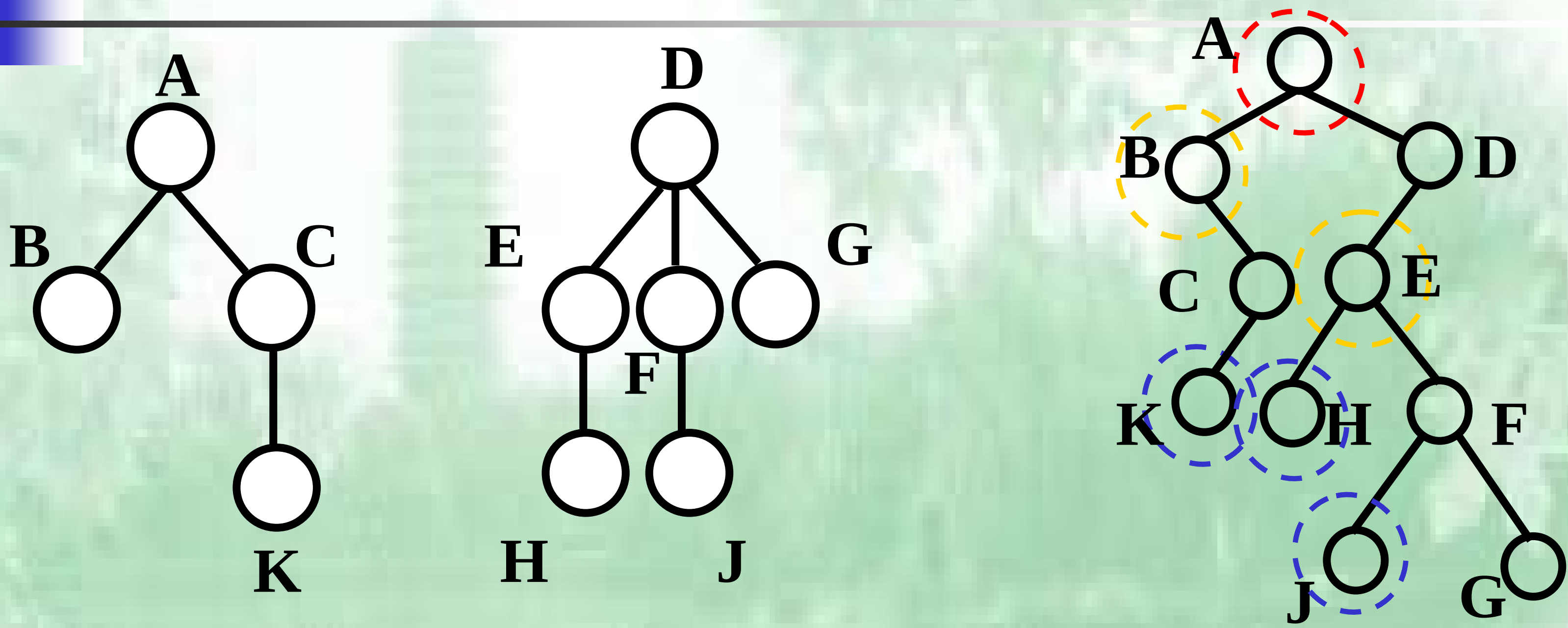
广度优先周游森林

```
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;
   while (pointer) {
       aQueue.push(pointer);
       pointer = pointer-> RightSibling() ;
   }
```

广度优先周游森林(续)

```
while(!aQueue.empty()){
    pointer=aQueue.front();            //取队首
    aQueue.pop();                      //出队列
    Visit(pointer->Value());           //访问
    pointer = pointer->LeftMostChild();
    while (pointer) {
      aQueue.push(pointer);
      pointer = pointer-> RightSibling() ;
    }
}
```

教材广度优先周游图示



广度优先周游森林（教材）

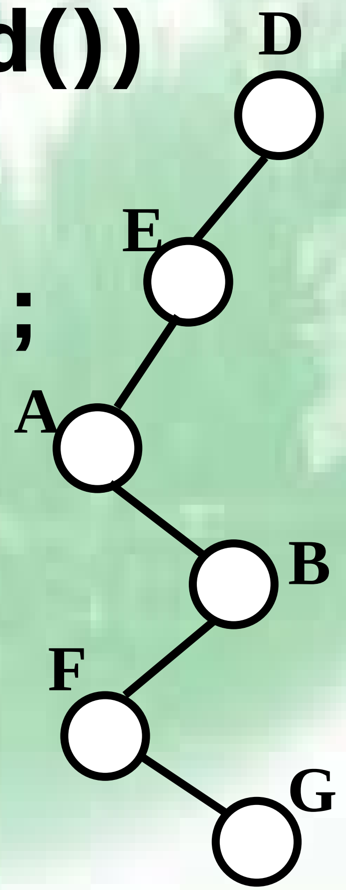
```
template <class T>
void Tree<T>::WidthTraverse1(TreeNode<T>*
   root){
   using std::queue;          //使用STL队列
   queue<TreeNode<T>*> aQueue;
   TreeNode<T>* pointer=root;
   if (!pointer)  return;
   aQueue.push(pointer);
   while(!aQueue.empty()) {
       pointer=aQueue.front();//取队列首结点指针
```


广度优先周游森林

```
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
}
```

广度优先周游森林

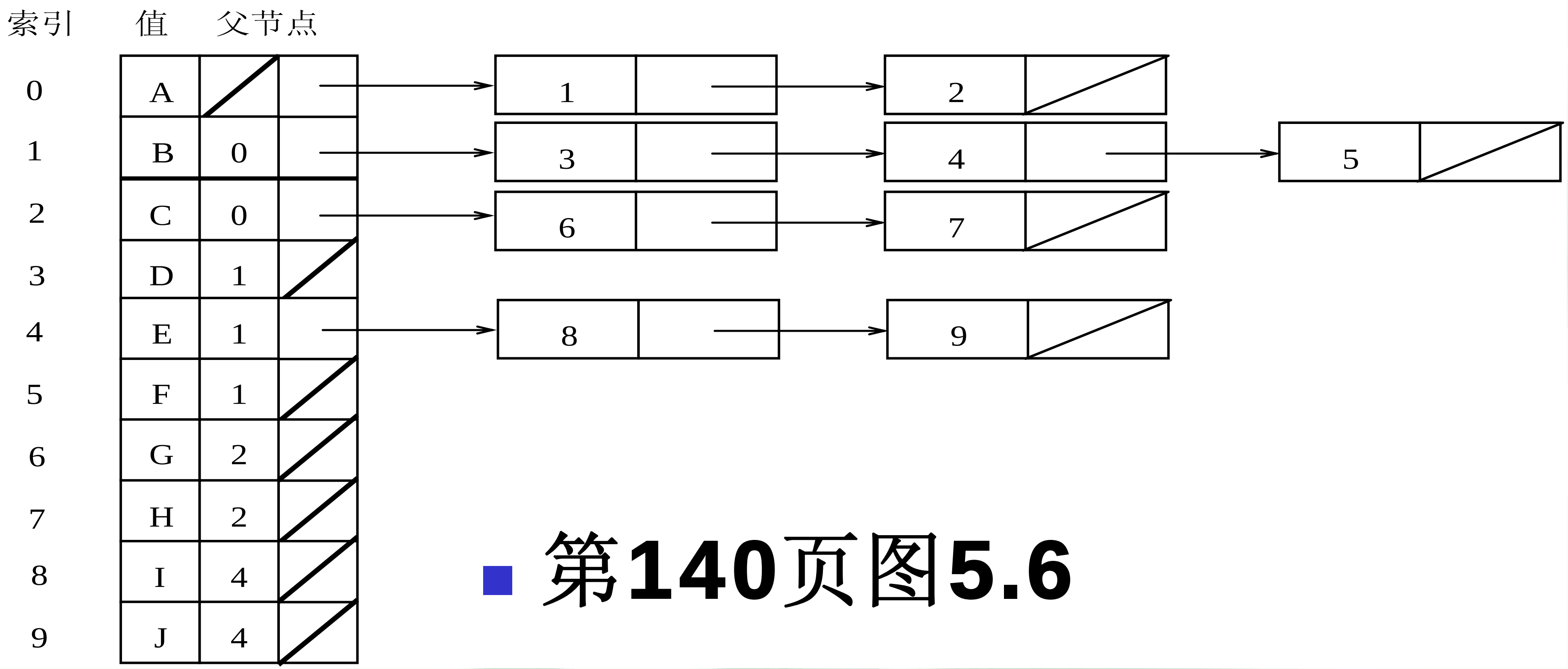
```
if (pointer->LeftMostChild())
    aQueue.push(pointer
        ->LeftMostChild());
aQueue.pop();  //出队列
}//end while
}
```



5.2 森林的链式存储

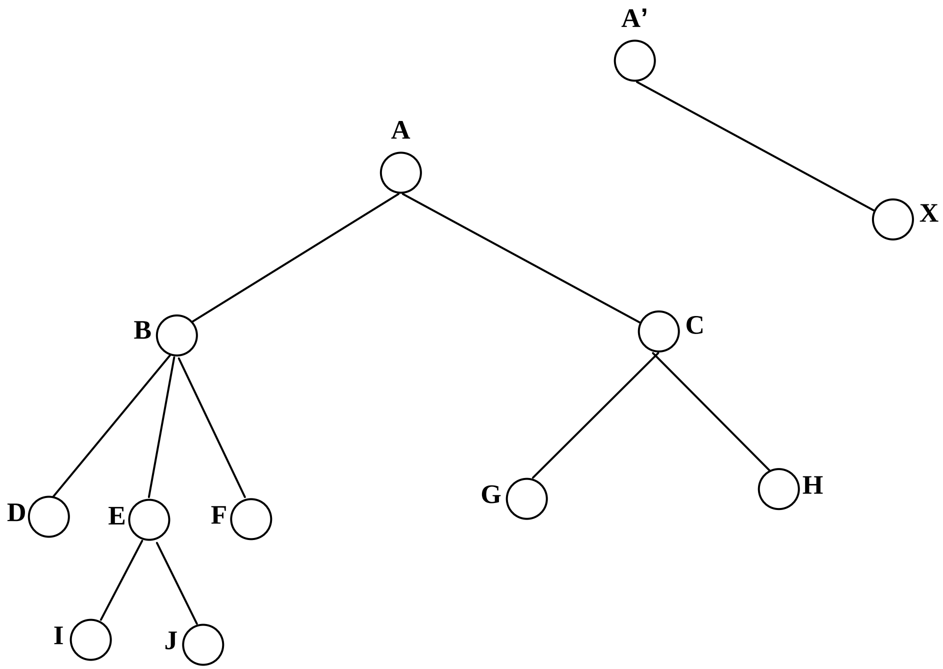
- 5.2.1 子结点表表示法
- 5.2.2 左子结点/右兄弟结点表示法
- 5.2.3 动态结点表示法
- 5.2.4 动态“左子/右兄弟”二叉链表表示法
- 5.2.5 父指针表示法

5.2.1 子结点表表示法



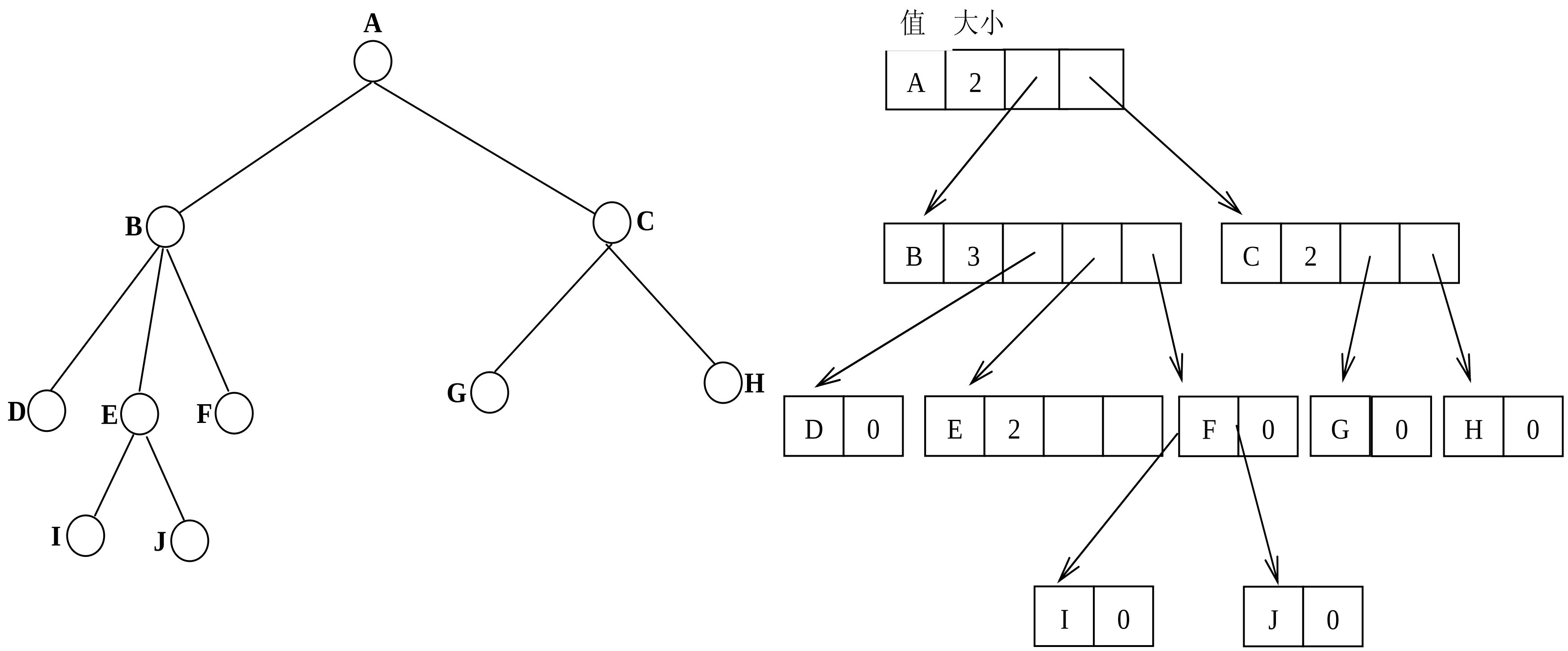
- 第140页图5.6

5.2.2 左子结点/右兄弟结点表示法
第140页图5.7



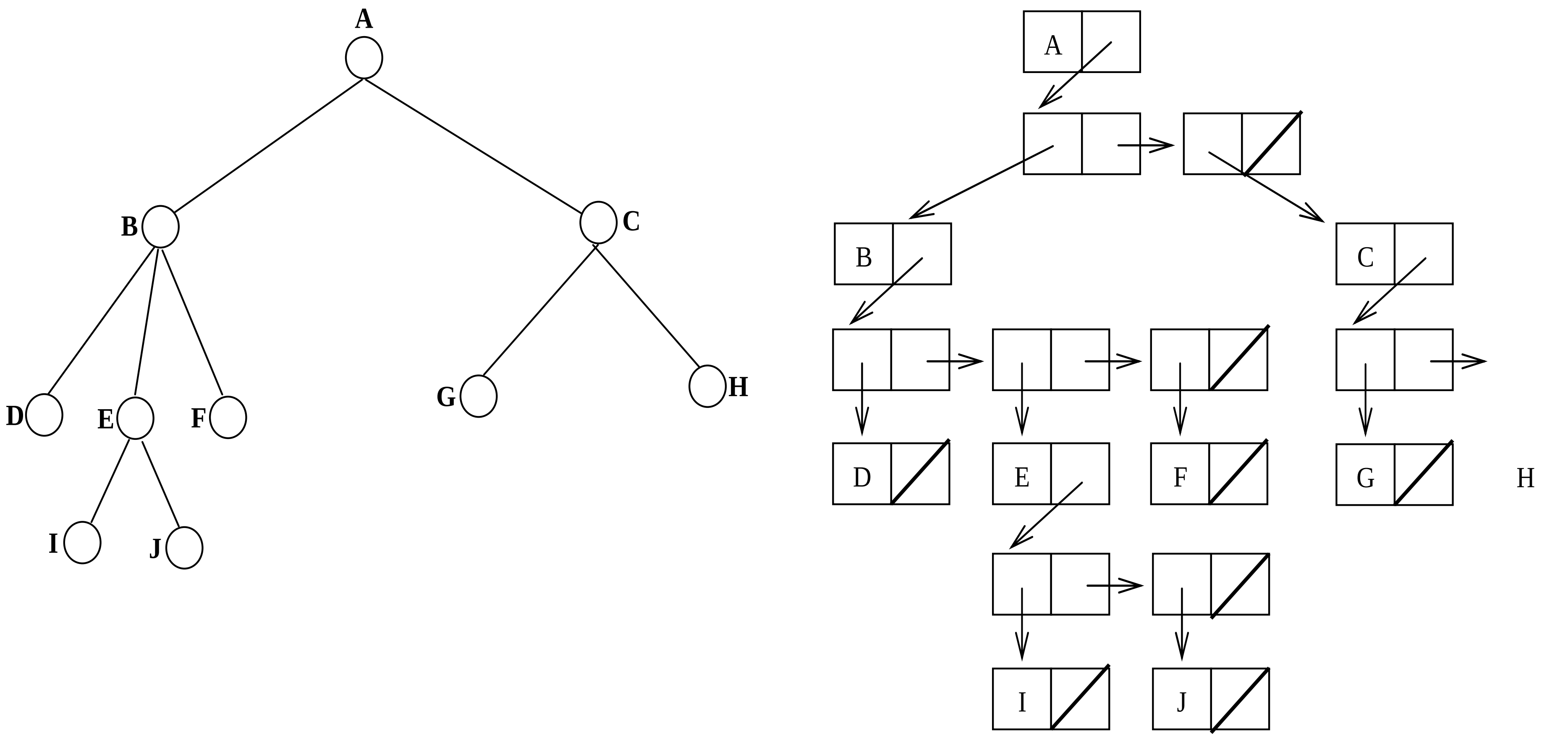
左子结点	值	父	右兄弟
1	A		
3	B	0	2
6	C	0	
	D	1	4
8	E	1	5
	F	1	
	G	2	7
	H	2	
	I	4	9
	J	4	
11	A'		
	X	10	

5.2.3 动态结点表示法
第142页图5.9



（a）树　　（b）树的实现

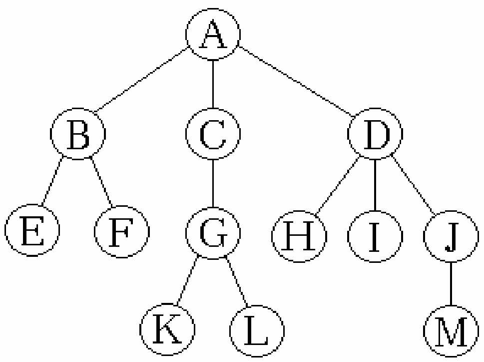
第142页图5.10
动态表示法



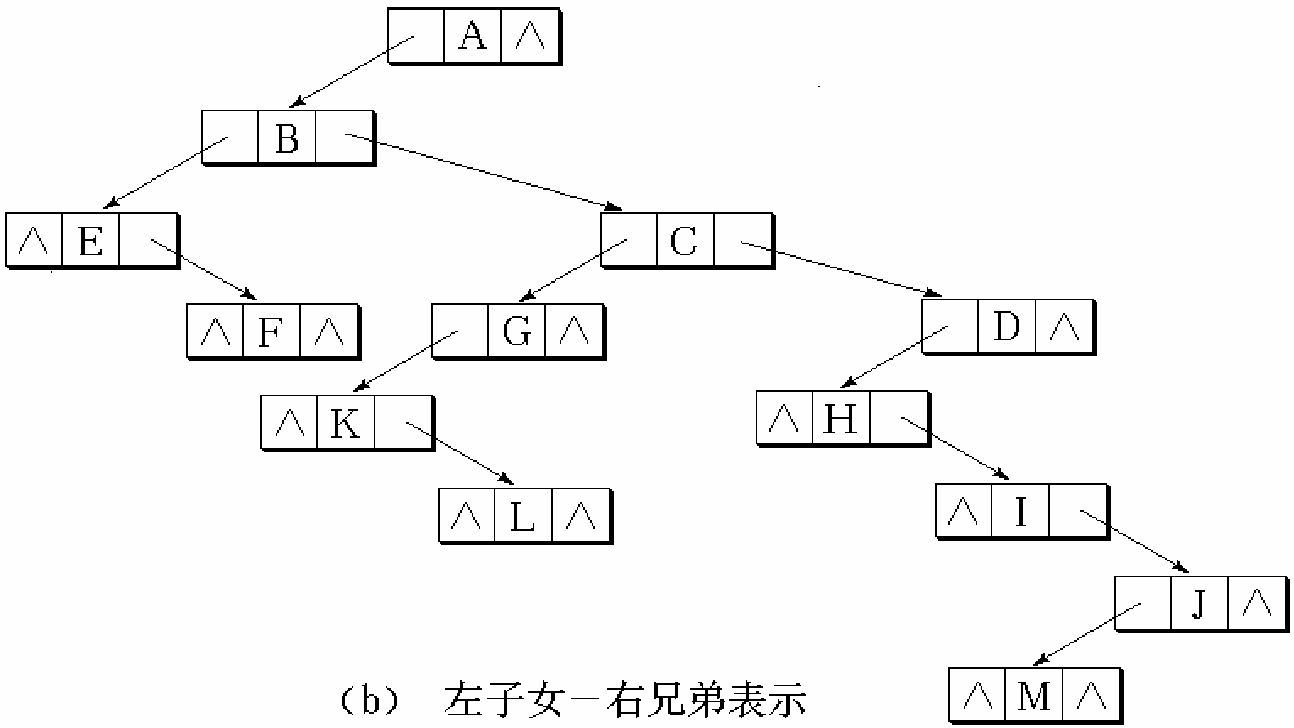
（a）树　　　　（b）树的实现

5.2.4 动态“左子/右兄弟”二叉链表

- 左结点/右兄弟：最常用



（a）树



（b） 左子女－右兄弟表示

Parent()函数的算法效率？

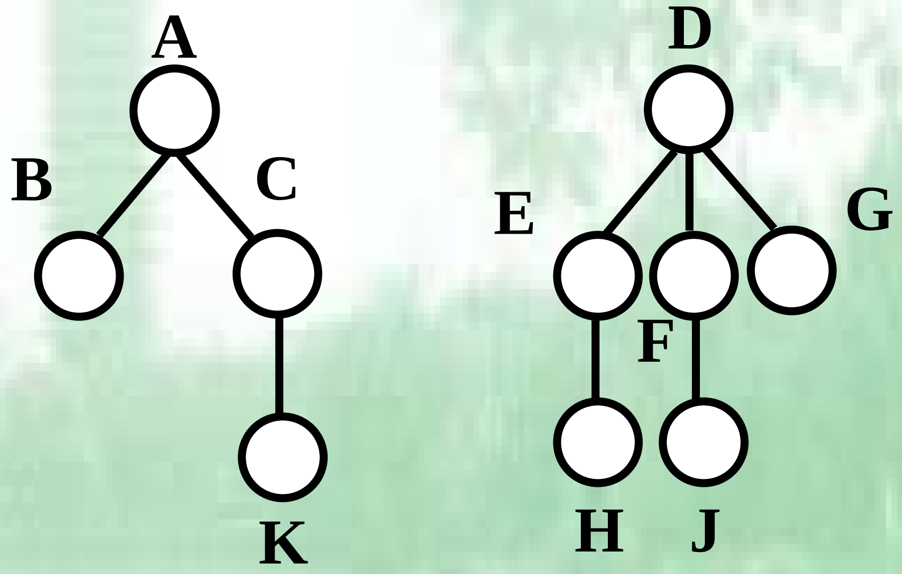
```
template <class T>
TreeNode<T>*
  Tree<T>::Parent(TreeNode<T>* current)
{//返回current结点的父结点
  TreeNode<T>* pointer=current;
   if(pointer!=NULL) return NULL;
  TreeNode<T>* leftmostChild=NULL;
```

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

5.2.5 父指针表示法

- 父指针(parent pointer)表示法
  - 每个结点只保存一个指针域指向其父结点
  - 最简单
  - 但是……
- 判断两个结点是否在同一棵树
  - 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



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

父指针表示法构建树专题

- 等价类及其并查算法
  - 算法示例
- 树的父指针表示与UNION/FIND算法实现
  - 路径压缩和优化

等价类(equivalence classes)

- 一个具有n个元素的集合S，另有一个定义在集合S上的r个关系的关系集合R。x，y，z表示集合中的元素
- 若关系R是一个等价关系，当且仅当如下条件为真时成立：
  - （a）对于所有的x，有(x，x)∈R（即关系是自反的）；
  - （b）当且仅当(x，y)∈R时(y，x)∈R（即关系是对称的）；
  - （c）若(x，y)∈R且(y，z)∈R，则有(x，z)∈R（即关系是传递的）。
- 如果(x，y)∈R，则元素x和y是等价的
- 等价类是指相互等价的元素所组成的最大集合。所谓最大，就是指不存在类以外的元素，与类内部的元素等价
- 由x∈S生成的一个R等价类
  - $[x]_R = \{y \mid y \in S \wedge xRy\}$
  - R将S划分成为r个不相交的划分S1，S2，…Sr，这些集合的并为S

等价类的并查（UNION/FIND）算法

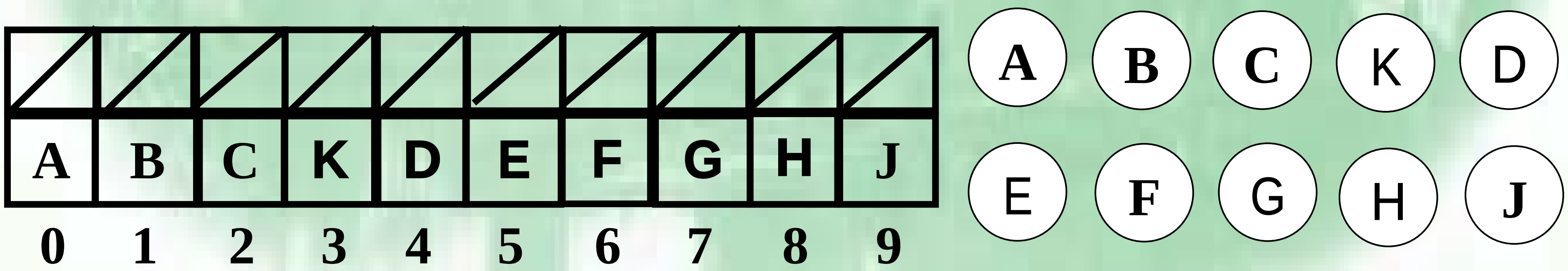
- **FIND**
  - 判断两个结点是否在同一个集合中
  - 查找一个给定结点的根结点
- **UNION**
  - 如果一个等价的两个元素不在同一棵树中
  - 归并两个集合，这个归并过程常常被称为

用树来表示等价类的并查

- "UNION/FIND"算法用一棵树代表一个集合
  - 集合用父结点代替
  - 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
  - 结点中仅需保存父指针信息
  - 存储为一个以其结点为元素的数组——静态指针数组

UNION/FIND算法示例(1)

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)



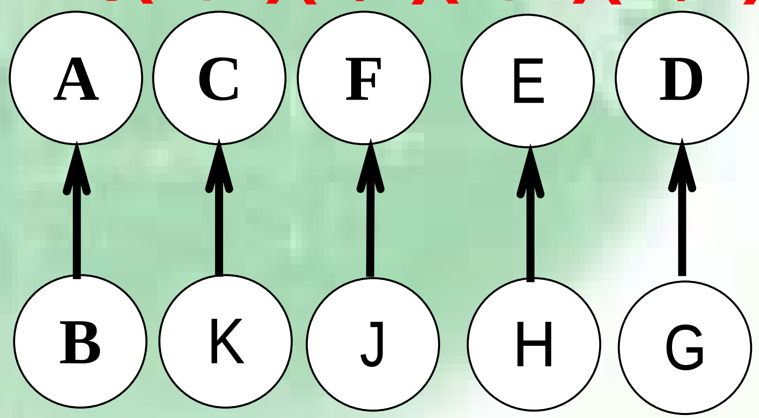
UNION/FIND算法示例(2)

首先对这5个等价对进行处理 (A,B)、(C,K) 、(J,F) 、(H,E) 、(D,G)

后四个等价对处理同（A，B）

(A,B)(C,K)(J,F)(E,H)(D,G)

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

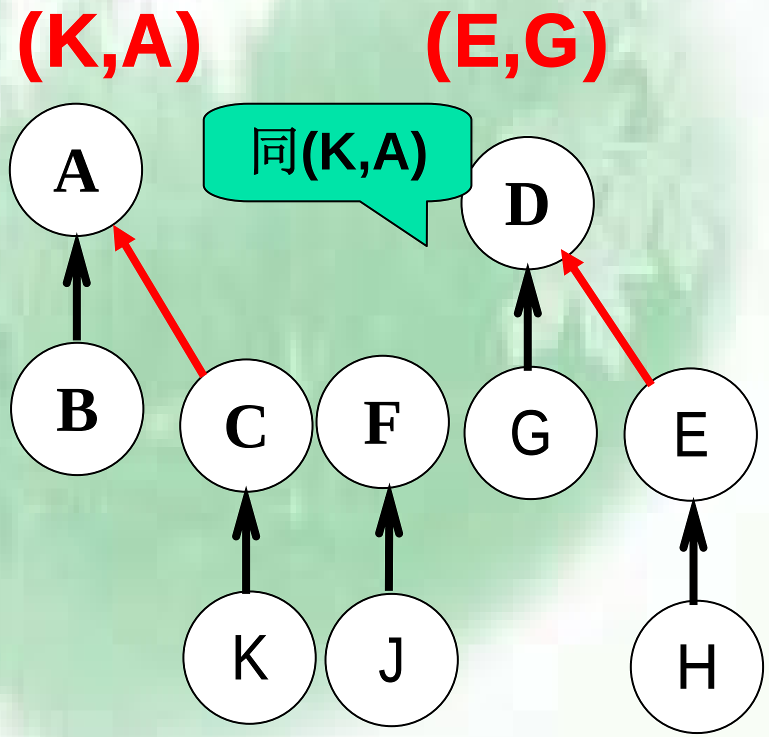


UNION/FIND算法示例(2)

然后对两个等价对（K，A）和（E，G）进行处理

K所在树的根为C，A自己是根节点，A！=C，所以两颗树要合并

/	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



父指针表示法的树结点定义

```
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
    //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);
};
```

父指针表示法的树定义

```
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
   //查找node结点的根结点
   ParTreeNode<T>*
   Find(ParTreeNode<T>* node) const;
```

```
    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);
};
```

FIND算法

```
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;
}
```

Different算法

```
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;
}
```

UNION算法

```
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)
   {    // 应用重量权衡合并规则
        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
}
```


路径压缩

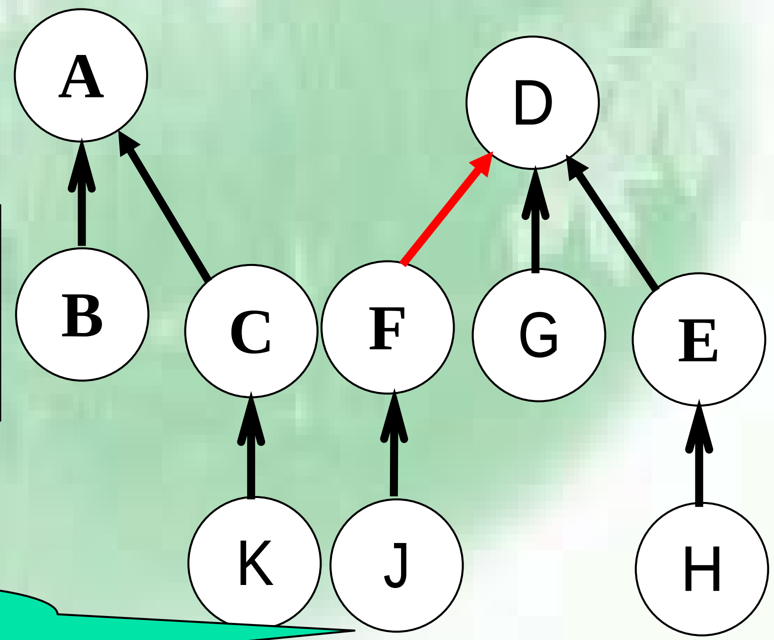
- 查找X
  - 设X最终到达根R
  - 顺着由X到R的路径把每个结点的父指针域均设置为直接指向R
- 产生极浅树

UNION/FIND算法示例(3)

最后使用标准重量权衡合并规则处理等价对（H, J）

(H,J)

/	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

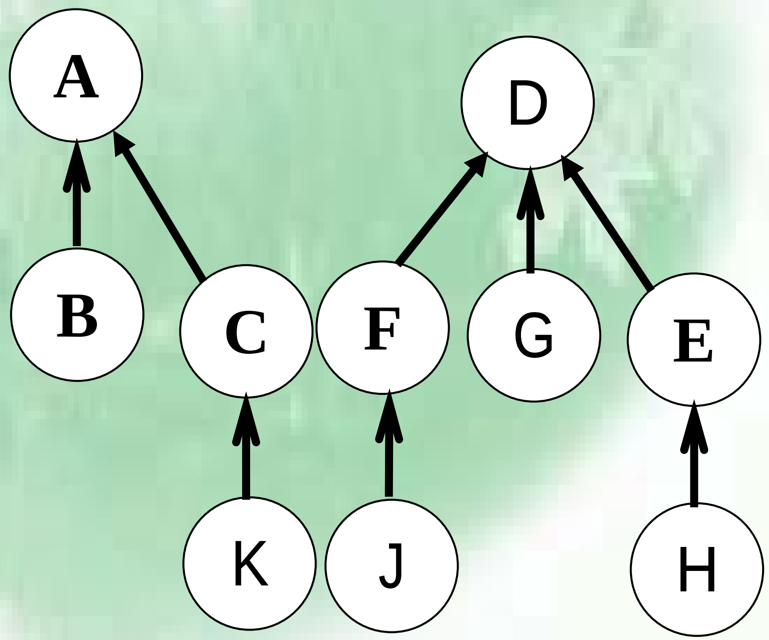


依据重量权衡合并规则，以F为根的树节点个数少，故将F指向E

UNION/FIND算法示例(4)

使用路径压缩规则处理等价对（H,E）

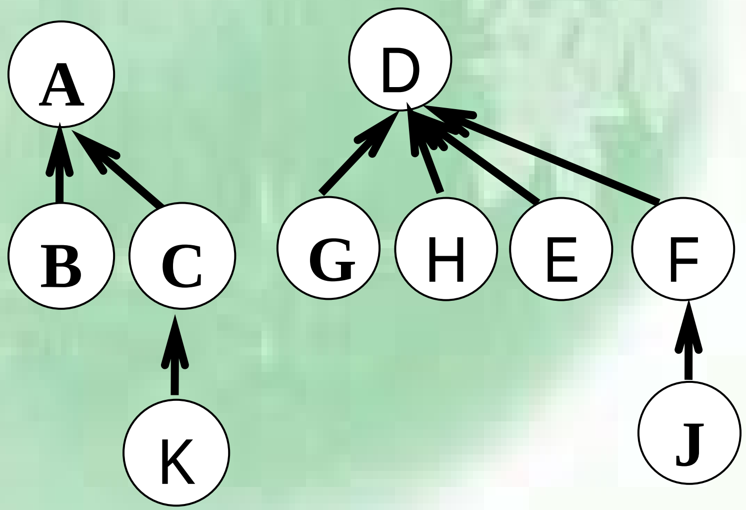
	0	0	2		4	4	4	5	6
A	B	C	K	D	E	F	G	H	J
0	1	2	3	4	5	6	7	8	9



UNION/FIND算法示例(4)

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();
}
```

路径压缩使Find开销接近于常数

- 对n个结点进行n次Find操作的路径压缩开销（用重量权衡合并规则来归并集合）为$\Theta(n\log^* n)$。
  - $\log^* n$是在$n \leqslant 1$之前要进行的对n取对数操作的次数。
  - $\log^* 65536 = 4$（4次log操作）。
- 一系列n个Find操作的开销非常接近$\Theta(n)$

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

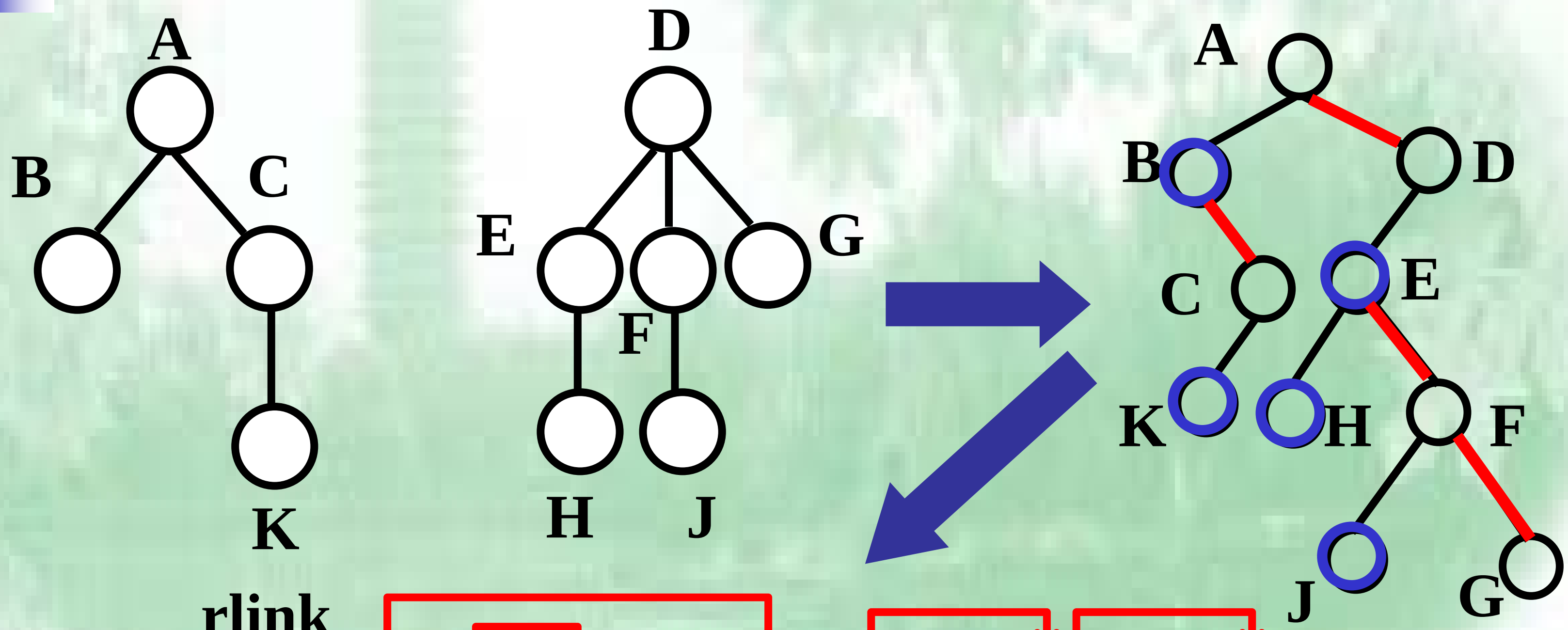
5.3.1 带右链的先根次序表示法

- 结点按先根次序顺序连续存储

ltag	info	rlink

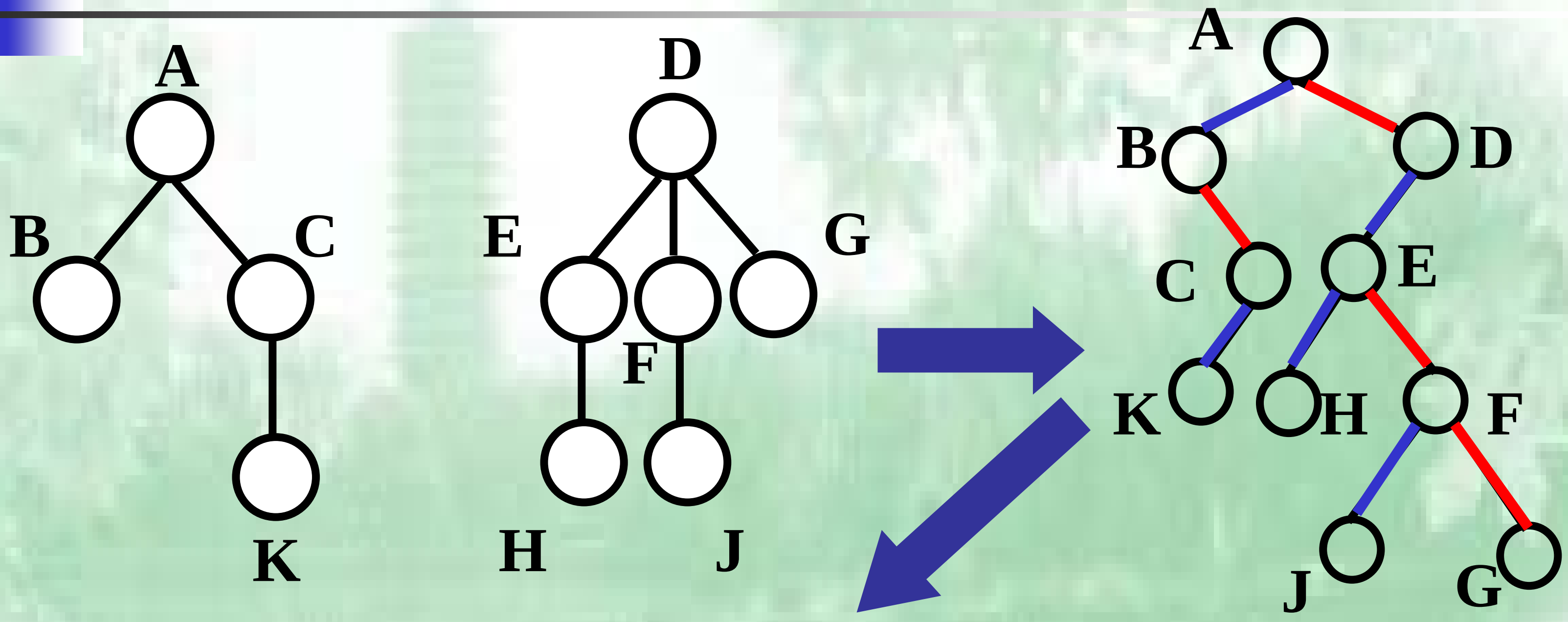
  - **info**：结点的数据
  - **rlink**：右指针
    - 指向结点的下一个兄弟、即对应的二叉树中结点的右子结点
  - **ltag**：标记
    - 树结点没有子结点，即二叉树结点没有左子结点，**ltag**为 **1**
    - 否则为**0**

带右链的先根次序表示法



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

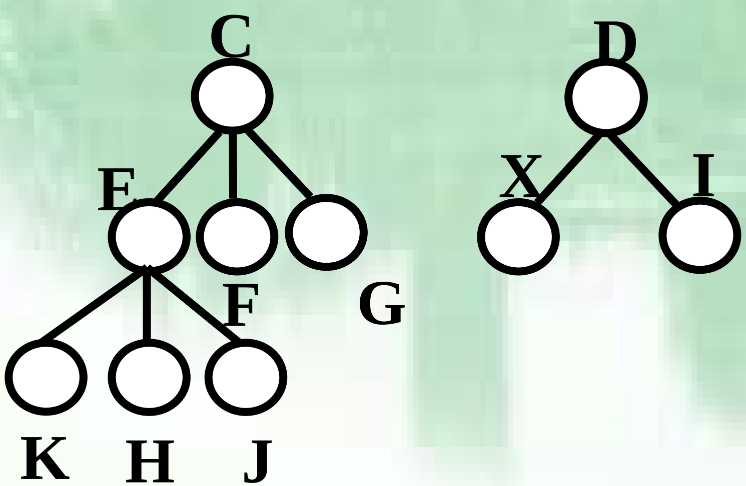
带双标记位的先根次序表示法

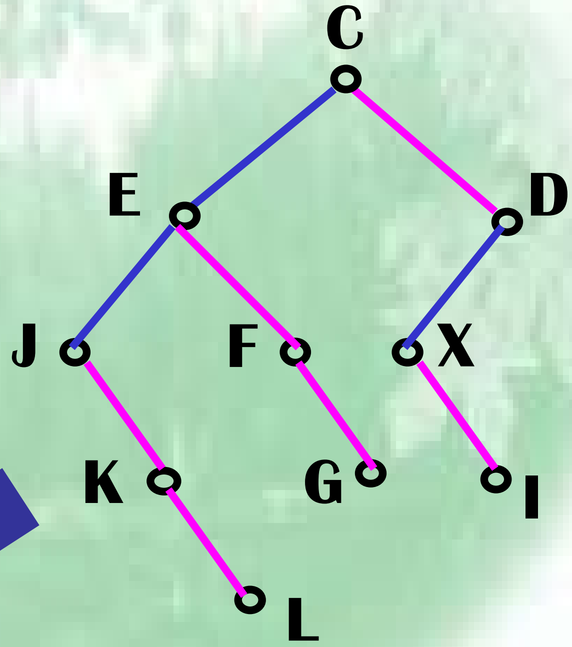


rtag	0	0	1	1	1	0	1	0	1	1
info	A	B	C	K	D	E	H	F	J	G
ltag	0	1	0	1	0	0	1	0	1	1

从先根rlink-ltag到树

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

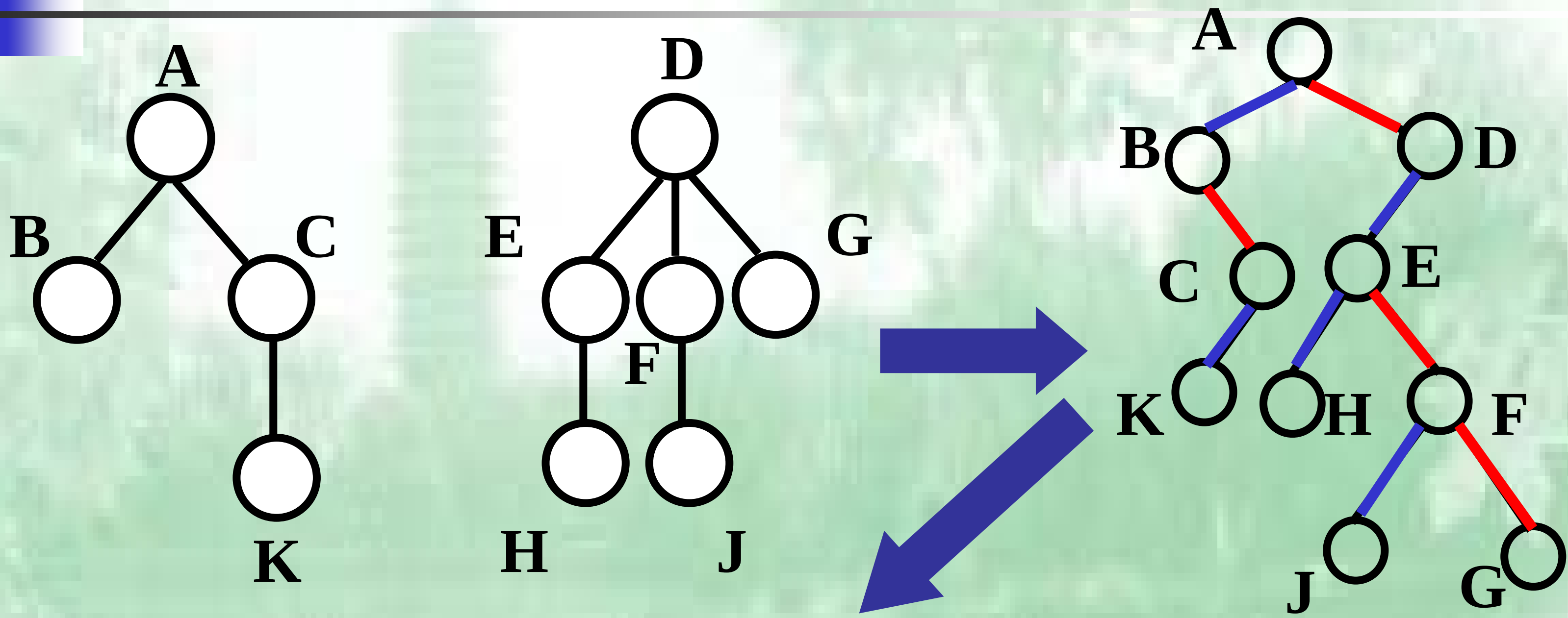




5.3.2 带双标记位的先根次序表示法

- 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？

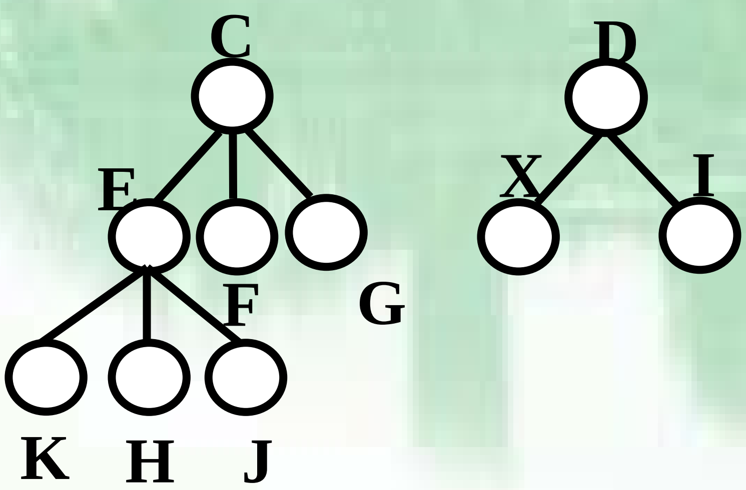
带双标记位的先根次序表示法

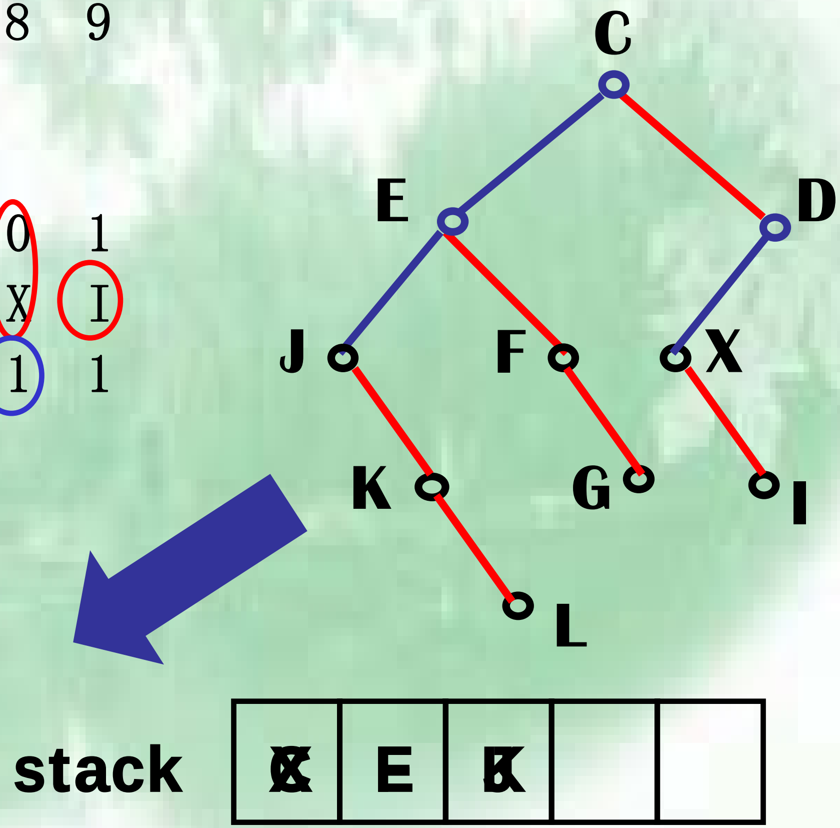


rtag	0	0	1	1	1	0	1	0	1	1
info	A	B	C	K	D	E	H	F	J	G
ltag	0	1	0	1	0	0	1	0	1	1

从rtag-ltag先根序列到树

下标	0	1	2	3	4	5	6	7	8	9
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





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:
        T       info;           //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>
Tree<T>::Tree(DualTagTreeNode<T>* 
  nodeArray,
                    int  count)
{
  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);
}
```

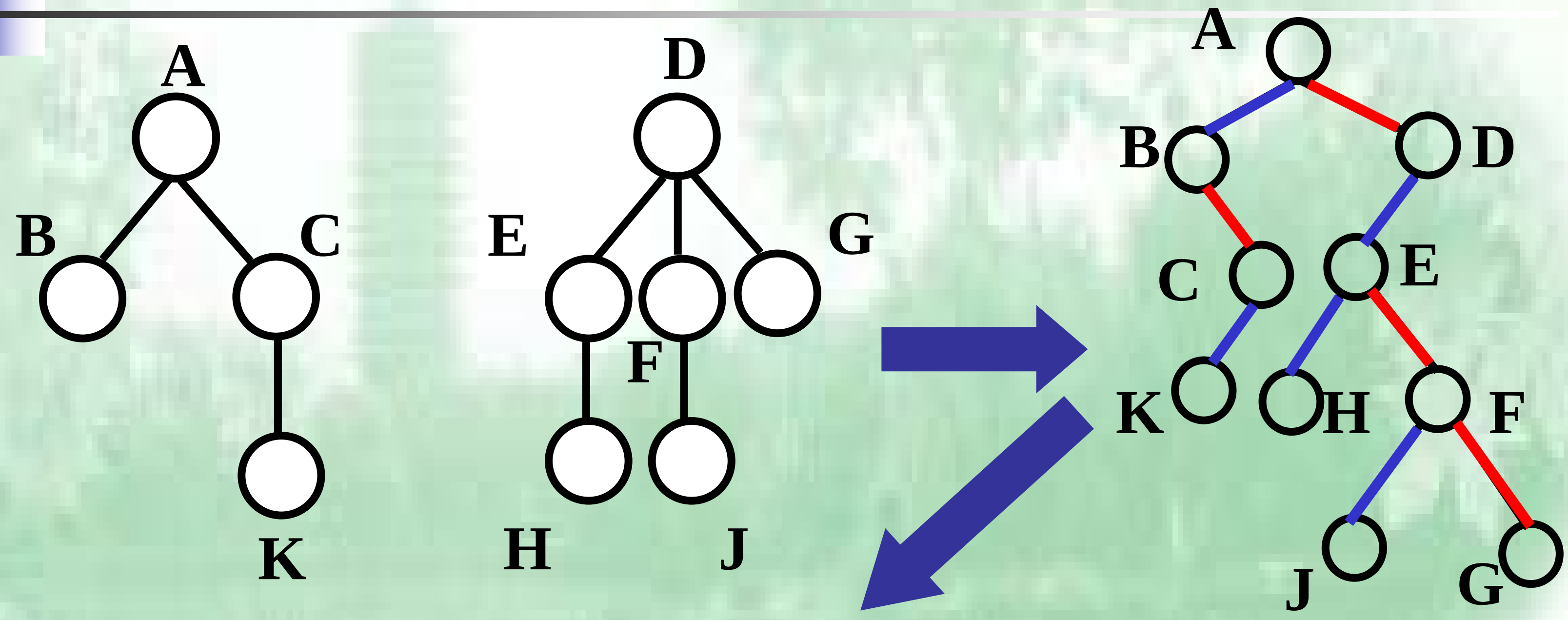
5.3.3 带左链的层次次序表示法

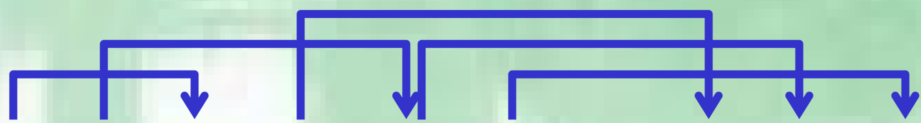
- 结点按层次次序顺序存储在连续存储单元

llink	info	rtag

  - **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

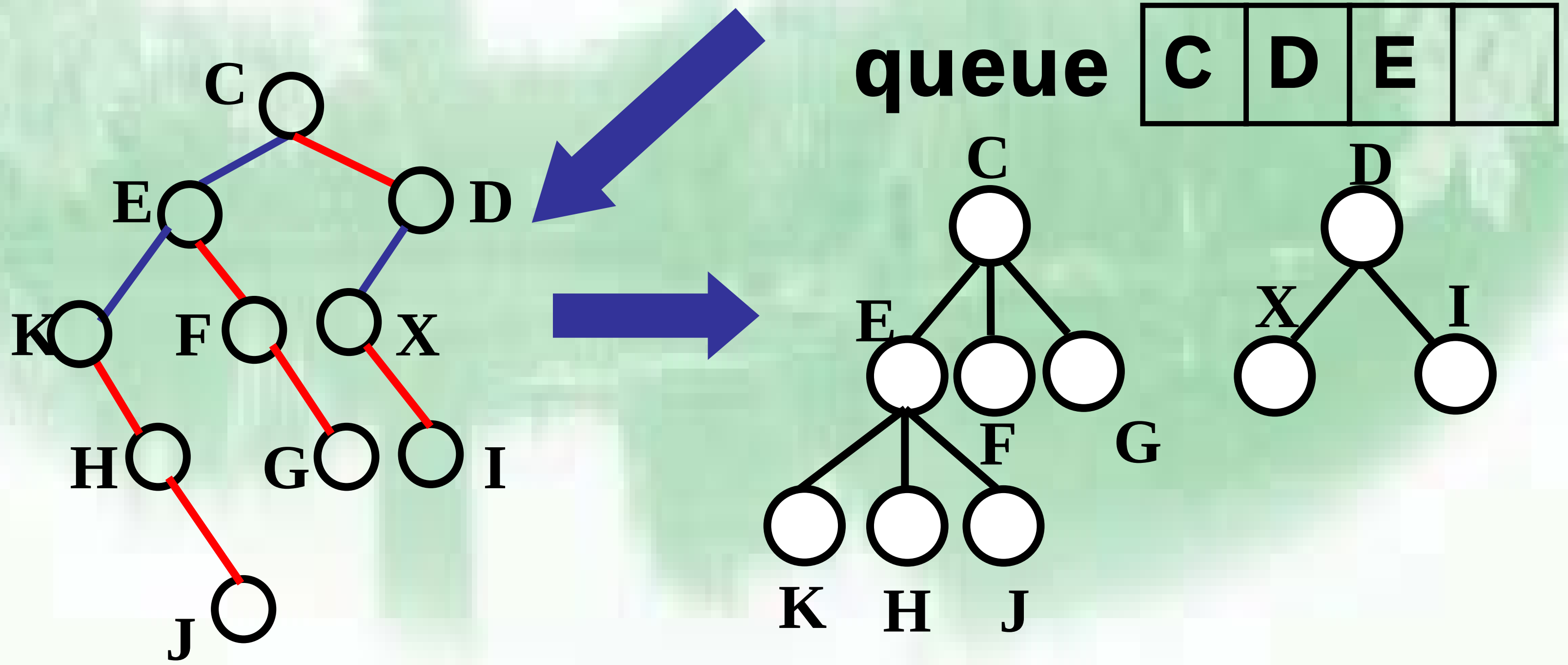




llink										
info	A	D	B	C	E	F	G	K	H	J
rtag	0	1	0	1	0	0	1	1	1	1

带左链的层次次序变成树

llink	2	5	7	-1	-1	-1	-1	-1	-1	-1
info	C	D	E	F	G	X	I	K	H	J
rtag	0	1	0	0	1	0	1	0	0	1



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:
  T   info;                          //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>
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
}
```

双标记层次次序树构造算法

```
template <class T>
Tree<T>::Tree(DualTagWidthTreeNode<T>*
    nodeArray, int count)  {
// 由带双标记位的层次次序表示构造左孩子右兄弟方式
    表示的树
using std::queue;                    // 使用STL队列
queue<TreeNode<T>*> aQueue;
    // 准备建立根结点
    TreeNode<T>* pointer = new TreeNode<T>;
    root = pointer;
```


双标记层次次序树构造算法

```
// 处理一个结点
  for (int i = 0;i<count-1;i++)  {
      pointer->setValue(nodeArray[i].info);
      if (nodeArray[i].ltag == 0)
          aQueue.push(pointer);    // 入队
      else  pointer->setChild(NULL);
      // 左孩子设为空
      TreeNode<T>* temppointer = new
  TreeNode<T>;
```

双标记层次次序树构造算法

```
    if (nodeArray[i].rtag == 0)
        pointer->setSibling(temppointer);
    else  {
        pointer->setSibling(NULL);     // 右设为空
        pointer = aQueue.front();     // 取队列首结点
        aQueue.pop();                  // 出队
        pointer->setChild(temppointer);
    }
    pointer = temppointer;
}
```

双标记层次次序树构造算法

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

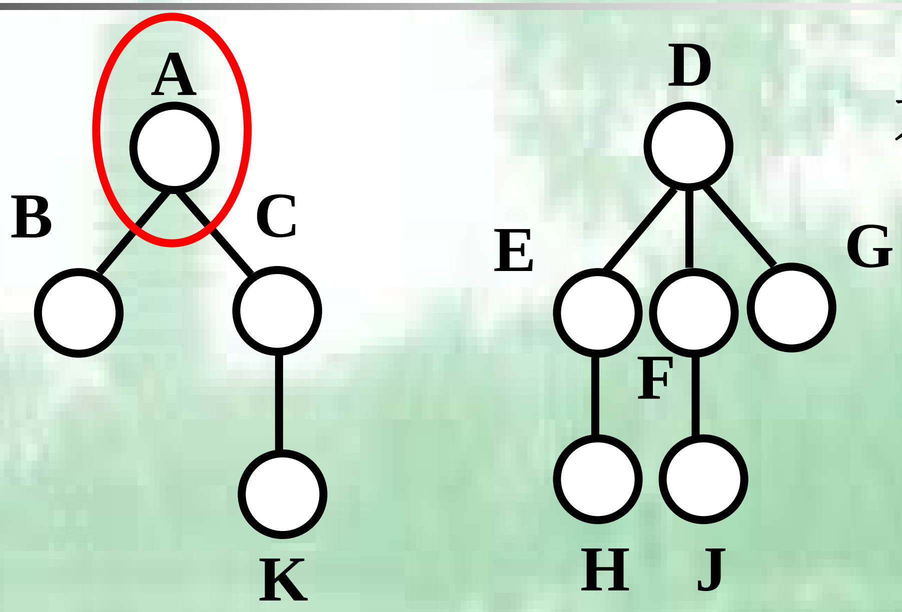

5.3.4 带度数的后根次序表示法

- 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

info	degree

- 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



不能用队列方法

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

- 冗余问题
- 树的其他顺序存储
  - 带度数的先根次序？
  - 带度数的层次次序？
- 二叉树的顺序存储？
  - 二叉树与森林对应，但语义不同
    - 带右链的二叉树前序
    - 带左链的二叉树层次次序
  - 带空指针信息的前序
  - 带标记的满二叉树

其他顺序表示法

- 二叉树与树的类似
  - 注意**left**、**right**含义
- 带空结点的前序（先根）
- 带标记的满二叉树
- 伪满二叉树

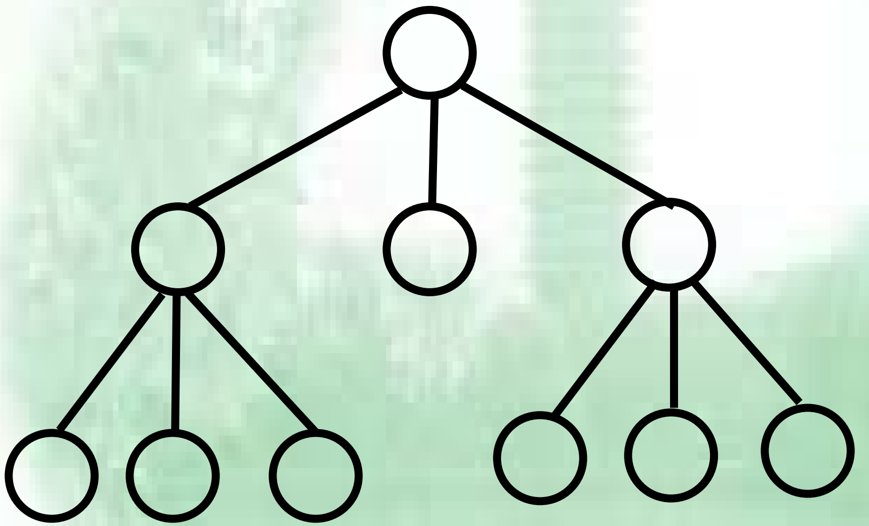
其他前序序列

- 前序扩充二叉树
- 带标记的满二叉树
- 带标记的伪满二叉树
- 嵌套括号的前序序列

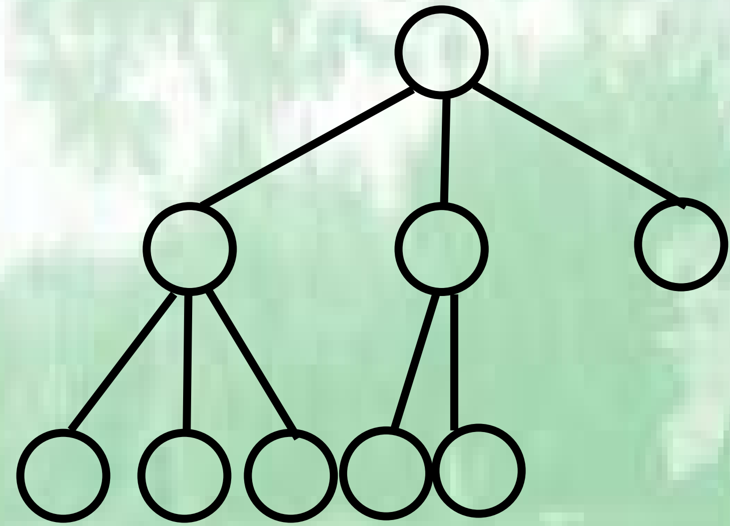
5.4 K叉树

- K叉树(K-ary Tree)的结点有K个子结点
- 二叉树的许多性质可以推广到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
  - 也可以把完全K叉树存储在一个数组中

满k叉树和完全k叉树



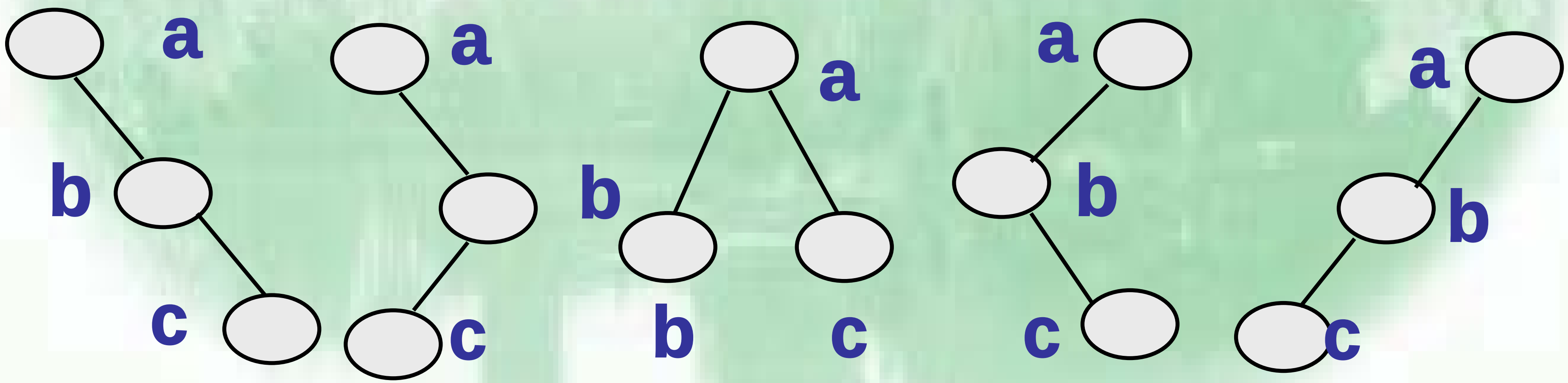
满3叉树



完全3叉树

补充：树计数

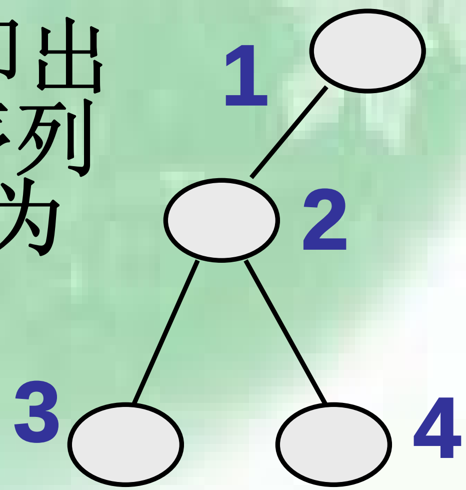
三个结点的二叉树



出栈序列与二叉树

- 对于数值为**1,2,3,4,...n**的进栈序列，构造二叉树。
- 例：**S1S2S3X3X2S4X4X1** 得到二叉树

这棵二叉树的中序遍历结果即出栈序列。显然所有的合法序列数就是**n**个结点，前序遍历为**12345...n**的二叉树个数。



一．二叉树计数

- 1． n个结点的不同形态二叉树的数目
  - 前序、中序可以决定一颗二叉树。
  - 令前序为1，2，…，n，合法的中序所构成的二叉树，中序周游的过程实际上是以1，2，…，n为顺序进栈、出栈的过程。
  - 1，2，…，n顺序进栈、出栈所得序列数目为*Catalan*函数：

$$b_n = c_{2n}^{n} - c_{2n}^{n-1} = \frac{1}{n+1} c_{2n}^{n}$$

方法一：二叉树递推方程

- 从另一个角度看，一棵具有**n(n≥1)**个结点的二叉树可以看为由一个根结点，一个**有i个结点的左子树**和一棵有**n-i-1个结点的右子树**组成（**0≤i≤n-1**）。而左子树和右子树的不同组合决定整棵树的形态，设**n**个结点的二叉树的个数为**b(n)**，则递推公式如下：

$$b_0 = 1$$

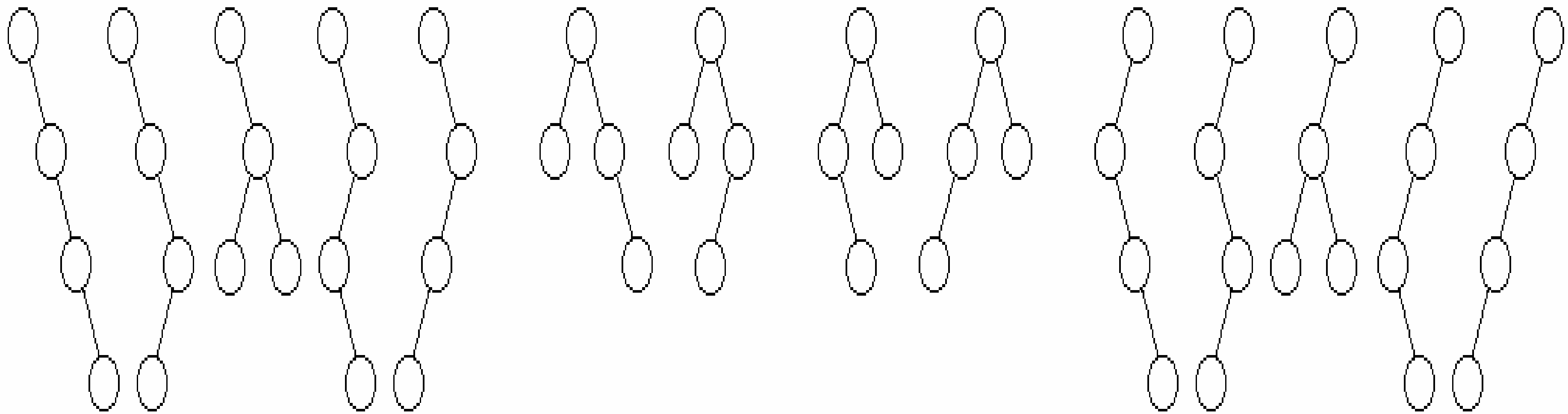
$$b_n = \sum_{i=0}^{n-1} b_i b_{n-i-1} \qquad \textbf{(n≥1)}$$

- 解此递推方程，得

$$b_n = \frac{1}{n+1} C_{2n}^{n}$$

递推公式的物理图景

- 例如，具有4个结点的不同二叉树



(a) $i=0$　(b) $i=1$　(c) $i=2$　(d) $i=3$

方法二：等概率匹配

- n个S，n个X的序列共有 (2n,n)种可能；
- 从左侧开始扫描这个序列，将S和X配对，其方法是每扫描到一个S或X，从它的右侧开始选取第一个与之匹配的X或S，删除这对SX或XS后再从最左侧开始下一个匹配。
  - 例如：SXSX和SSXX经扫描后都得到两个匹配：SX，SX。
- 对于一个长度为2n的序列，得到n对匹配。其中SX的个数可以为0，1，…，n共n+1种情况。根据栈的LIFO特性，只有SX个数为n的那些序列才是合法的。由于等概分布，故合法序列个数为(2n,n)/(n+1)。

方法三：Knuth方法(1)

- **n**个**S**、**n**个**X**的序列公有 **(2n,n)**种可能；
- 对于这样的序列从左到右进行扫描，遇到第一个**X**个数大于**S**个数的位置**j**，把从**1**到**j**这个**j**个位置的所有。这个序列就是一个**n+1**个**S**、**n-1**个**X**的**X换成S、S换成X**序列
- **S X S X X S**

  ↑ ↑ ↑ ↑ ↑

  X S X S S S

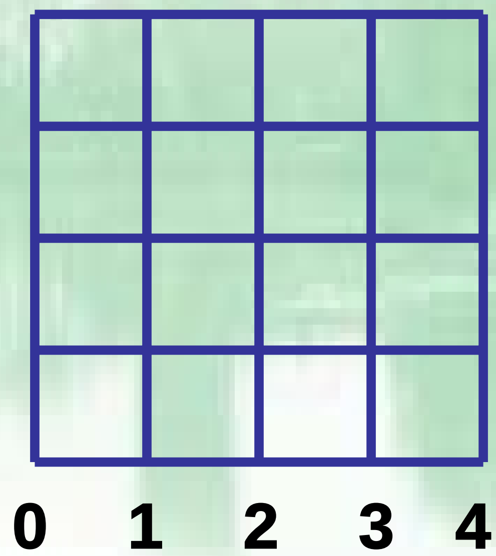
Knuth方法(2)

- 这样的由一个n+1个S、n-1个X组成的序列，可以与n个S、n个X所组成的不合法的序列一一对应
  - 这样的序列共有(2n,n+1)种可能
- 故合法序列个数为

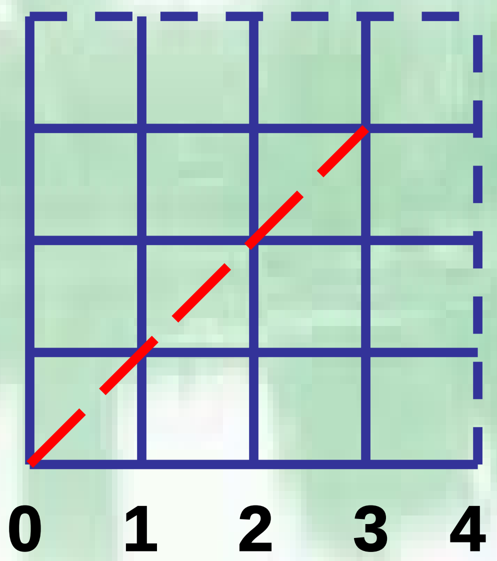
  (2n,n) -(2n,n+1)=(2n,n)/(n+1)。

方法四：非降路径问题（经典组合数学问题）

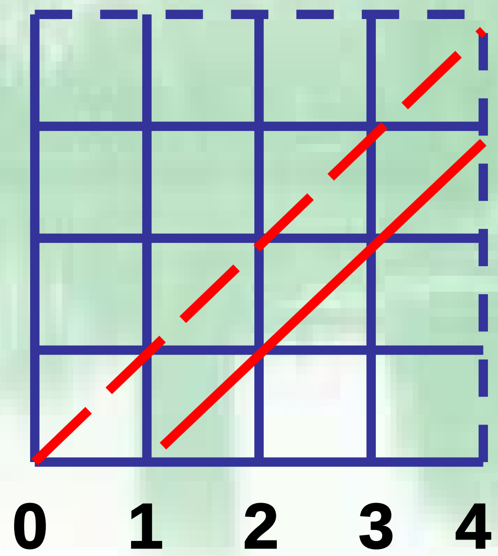
- 合法序列个数相当于从（**0**，**0**）点到（**n**，**n**）点不穿过直线**y=x**且在该直线下侧的非降路径数。
  - 其中沿**X**轴走一步表示压栈，沿**Y**轴走一步表示弹栈



- 接触到直线**y=x**表示该时刻栈为空。可以证明这样的对应是双射。
- 由非降路径问题可知，从（**0**，**0**）点到（**n**，**n**）点不接触直线**y=x**且在该直线下侧的非降路径数为**(2n,n)/2(2n-1)**。



- 从点(0，0)到点(n＋1,n＋1)的除端点外不接触直线y＝x的且在直线y=x下面的非降路径数
  - 等于从点(1，0)到点(n＋1,n)的可以接触不可穿过y=x－1且在直线y=x－1下面的非降路径数
  - 等于从点(0，0)到点(n,n)的可以接触不可穿过y=x且在直线y=x下面的非降路径数。



- 故合法序列个数为 (2(n+1),n+1)/2(2n+1) =(2n,n)/(n+1)

$$= \frac{1}{n+1}C_{2n}^{n}$$

二叉树计数

- 2. n个结点的不同形态二叉搜索树的数目为：

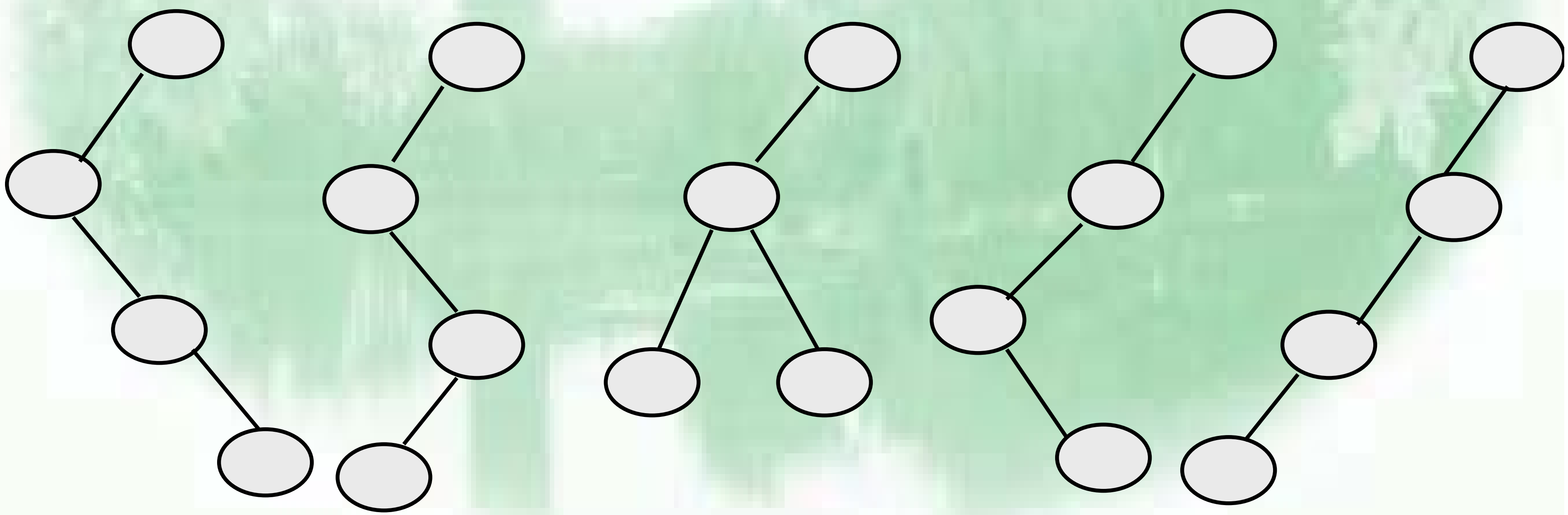
$$b_n = c_{2n}^{n} - c_{2n}^{n-1} = \frac{1}{n+1} c_{2n}^{n}$$

- 3. n个结点的不同形态标号二叉树的数目为：

$$\frac{n!}{n+1} c_{2n}^{n}$$

二. 树计数

- 例如，具有4个结点的，根的右子树为空的不同二叉树



树计数结论(1)

- 1. n个结点的不同形态有序树的数目
  - 由于一颗二叉树可以转换成一颗没有右子树的二叉树，反之亦然。
  - 因此，具有n个结点互不相似的树的数目t_n与具有n-1个结点的互不相似的二叉树的数目b_{n-1}相同。

$$t_n = b_{n-1} = \frac{1}{n} C_{2(n-1)}^{n-1}$$

树计数结论(2)

- 2. n个结点的不同形态标号有序树的数目

$$\frac{n!}{n}c_{2(n-1)}^{n-1}=(n-1)!c_{2(n-1)}^{n-1}$$

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

Thank you!

Q&A