数据结构与算法（A）-W06/树

北京大学 陈斌

2024.10.11

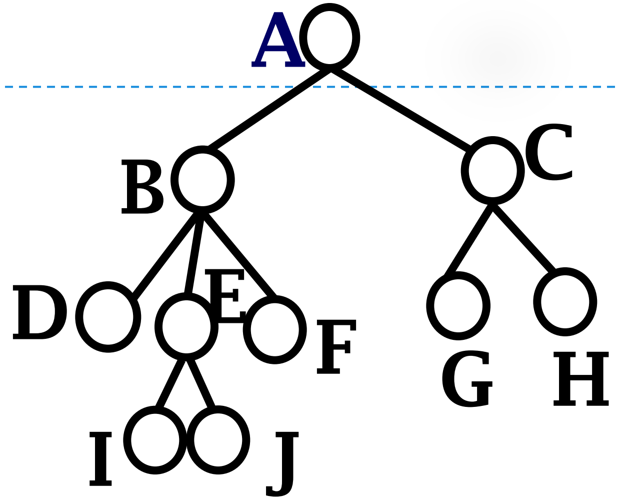
第六章 树

王腾蛟 主讲

采用教材：《数据结构与算法》，张铭，王腾蛟，赵海燕 编写
高等教育出版社，2008. 6 （“十二五”国家级规划教材）

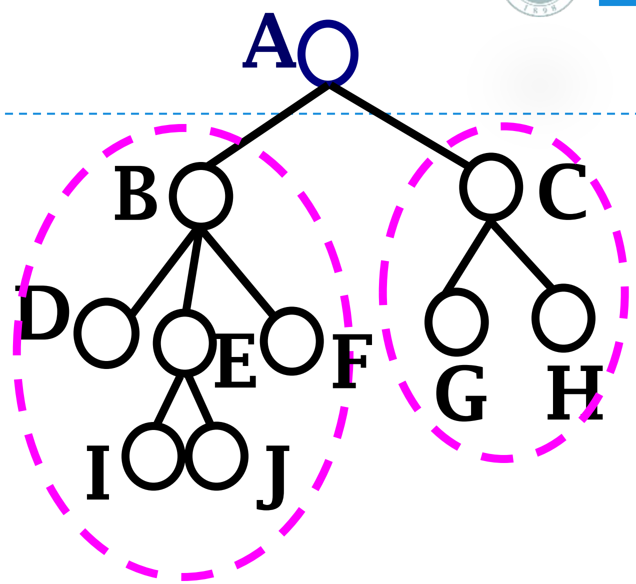
http://jpk.pku.edu.cn/course/sjjg/
https://www.icourse163.org/course/PKU-1002534001

第6章　树



- 树的定义和基本术语
  - 树和森林
  - 森林与二叉树的等价转换
  - 树的抽象数据类型
  - 树的遍历
- 树的链式存储结构
- 树的顺序存储结构
- K叉树

6.1 树的定义和基本术语

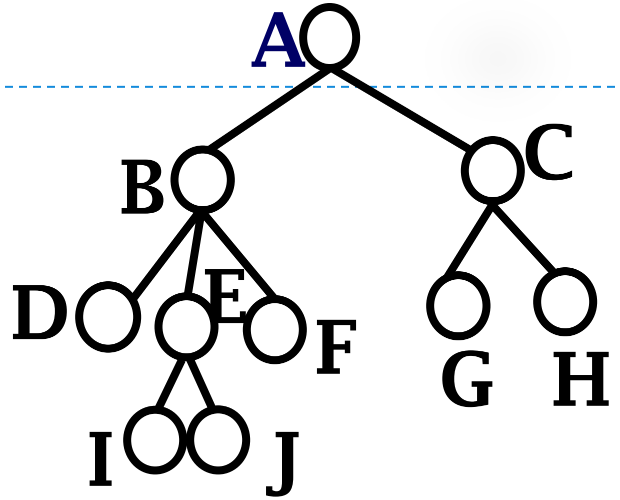


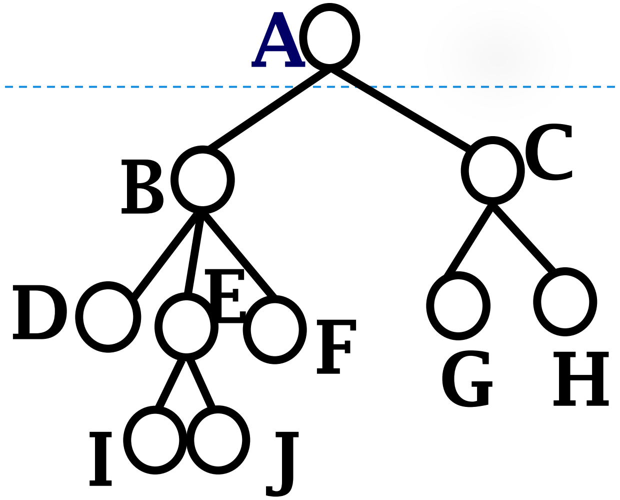
树和森林

- 树 (tree) 是包括 n 个结点的有限集合 T （n ≥ 1）：
  - 有且仅有一个特定的结点，称为 **根 (root)**
  - 除根以外的其他结点被分成 m 个 (m ≥ 0) 不相交的有限集合 T_1，T_2，…，T_m，而每一个集合又都是树，称为 T 的**子树 (subtree)**
  - 有向有序树：子树的相对次序是重要的
- 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一

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 r，关系 r 满足以下条件：
  - 有且仅有一个结点 $k_0 \in K$，它对于关系 r 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 r 来说都有且仅有 一个前驱
- 例如，
  - 结点集合 K={ A，B，C，D，E，F，G，H，I，J }
  - K 上的关系 r = { <A，B>，<A，C>，<B，D>，<B，E>，<B，F>，<C，G>，<C，H>，<E，I>，<E，J> }

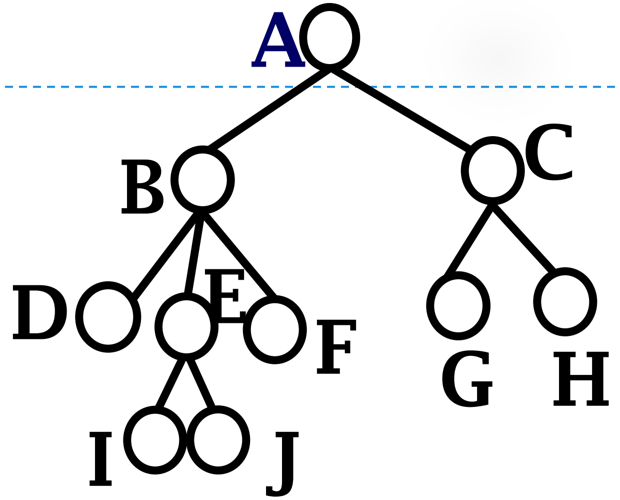




树的相关术语

- **结点**
  - **子结点、父结点、最左子结点**
    - 若 <k，k′> ∈ r，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”)
  - **兄弟结点前兄弟、后兄弟**
    - 若有序对 <k，k′>及 <k，k″>∈ r，则称 k′和 k″互为兄弟
  - **分支结点、叶结点**
    - 没有子树的结点称作 叶结点（或树叶、终端结点）
    - 非终端结点称为 分支结点

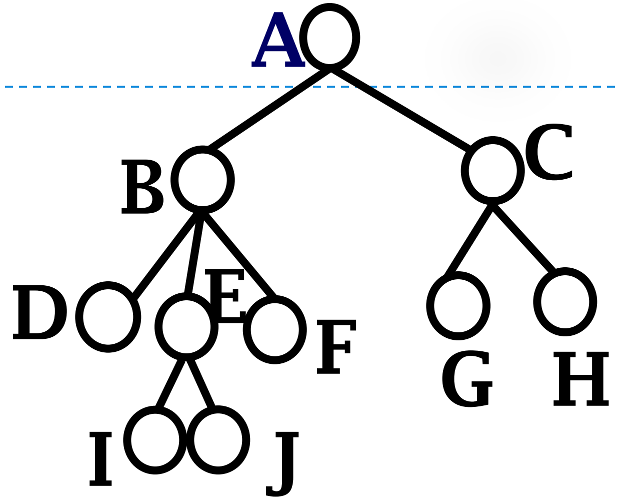
树的相关术语



- **边**
  - 两个结点的有序对，称作 边
- **路径、路径长度**
  - **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 $<k_{i-1}, k_i> \in r$（$1 \le i \le s$）。该序列称为从根到结点 k 的一条路径，其路径长度为 s (包含的边数)**
- **祖先、后代**
  - 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

树的相关术语

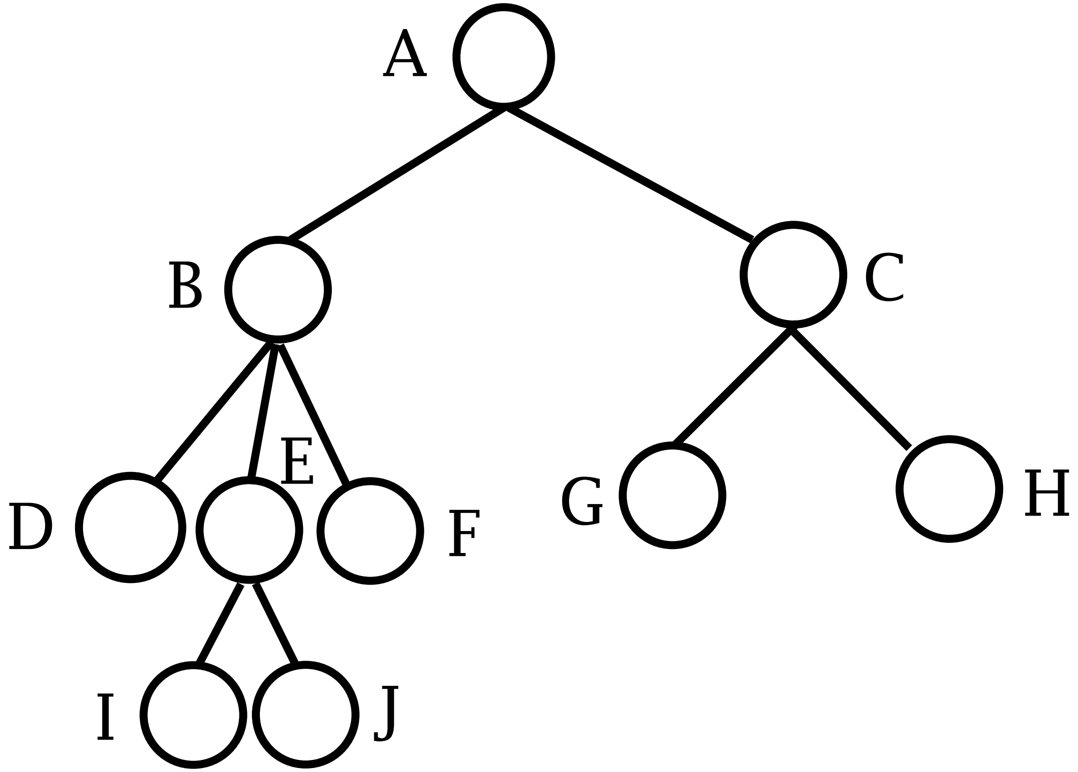
- **度数**：一个结点的子树的个数
- **层数**：根为第 0 层
  - 其他结点的层数等于其父结点的层数加 1
- **深度**：层数最大的叶结点的层数
- **高度**：层数最大的叶结点的层数加 1

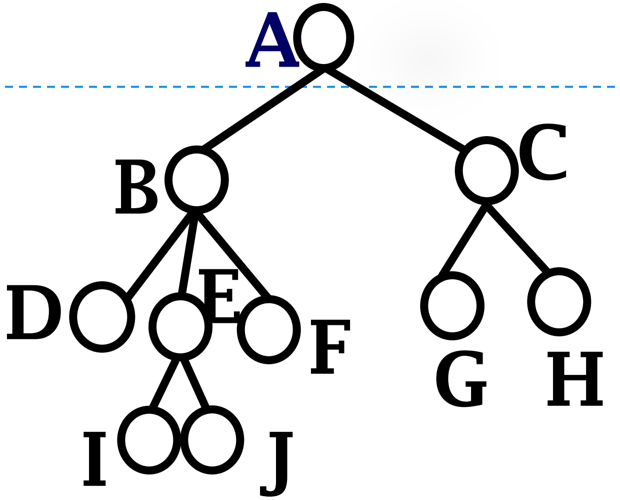


树形结构的各种表示法

- 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





形式语言表示法

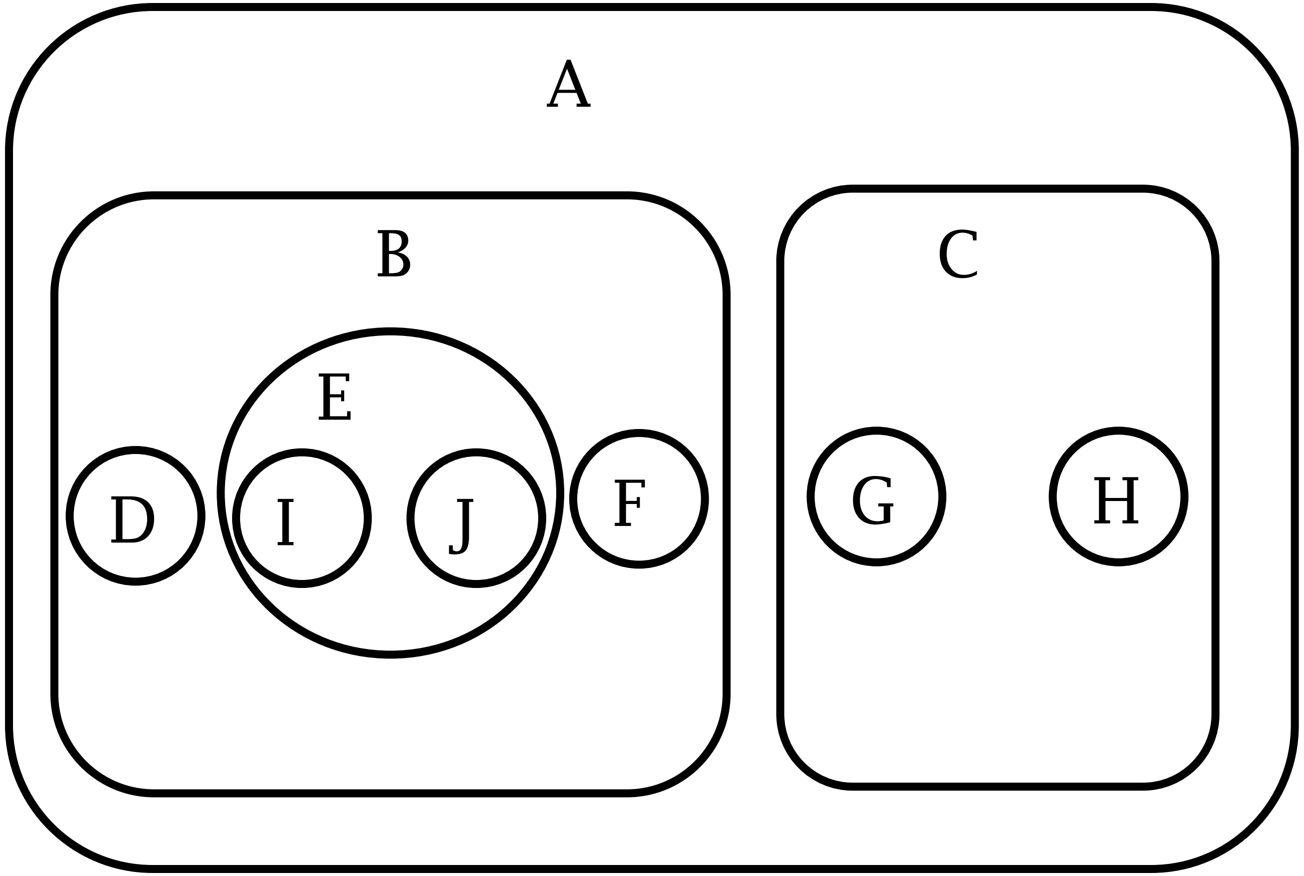
树的逻辑结构是：

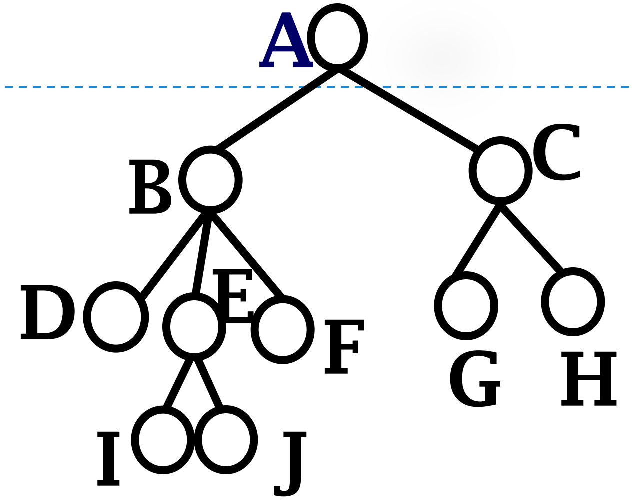
结点集合K={A，B，C，D，E，F，G，H，I，J}

K上的关系N={<A，B>，<A，C>，<B，D>，
<B，E>，<B，F>，<C，G>，
<C，H>，<E，I>，<E，J>}

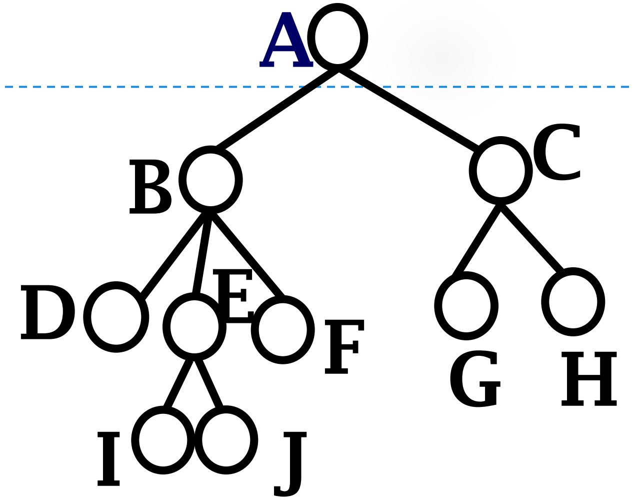
6.1 树的定义和基本术语

文氏图表示法



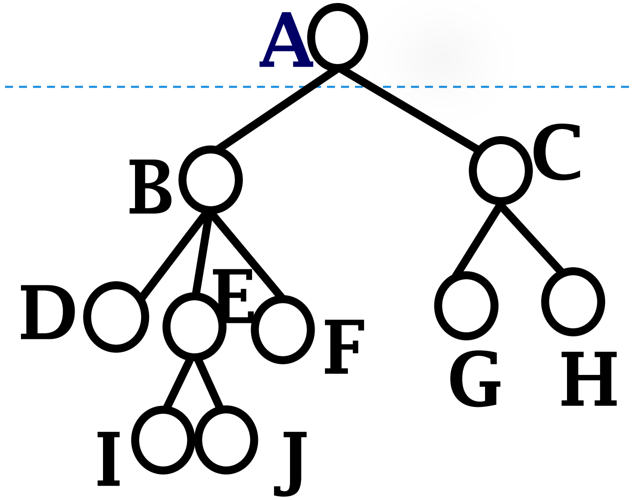


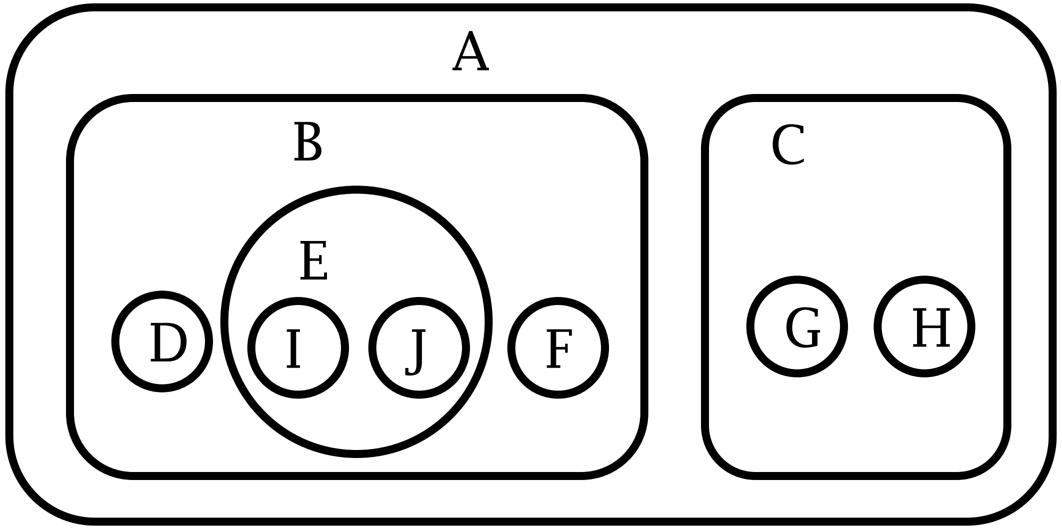
嵌套括号表示法



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

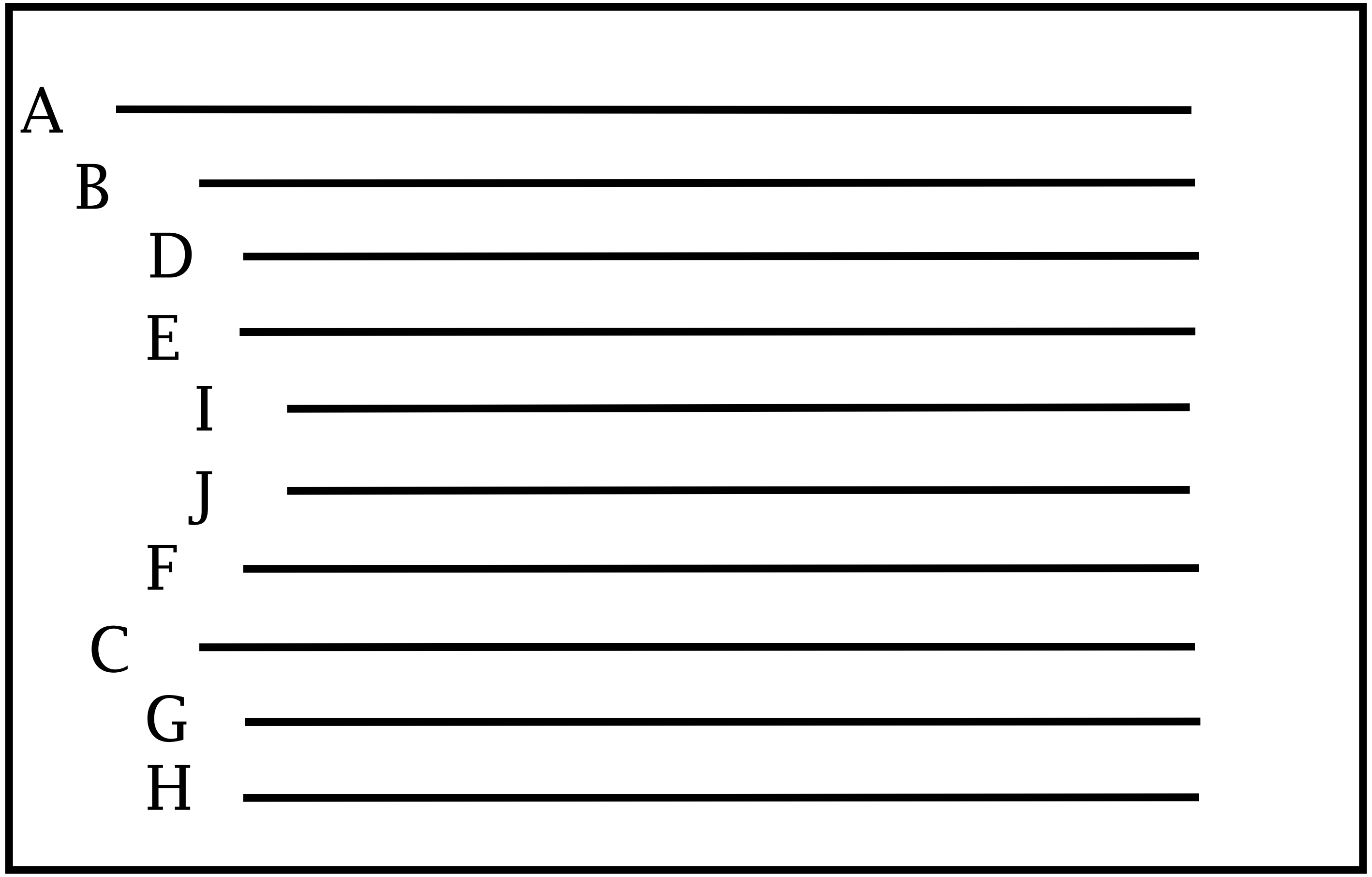
文氏图到嵌套括号表示的转化

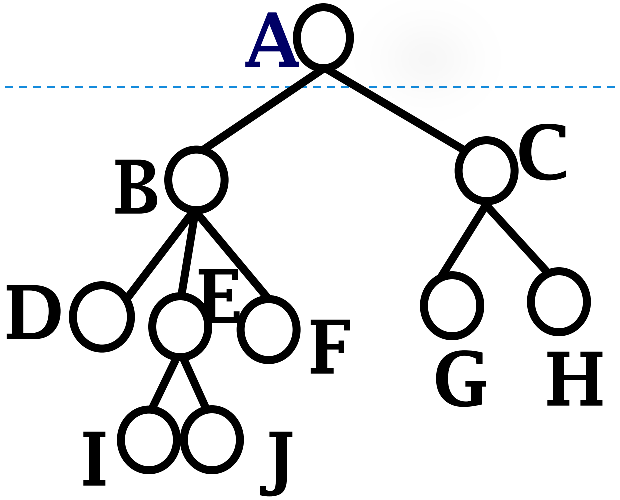




(A(B(D)(E(I)(J))(F))(C(G)(H)))

凹入表表示法

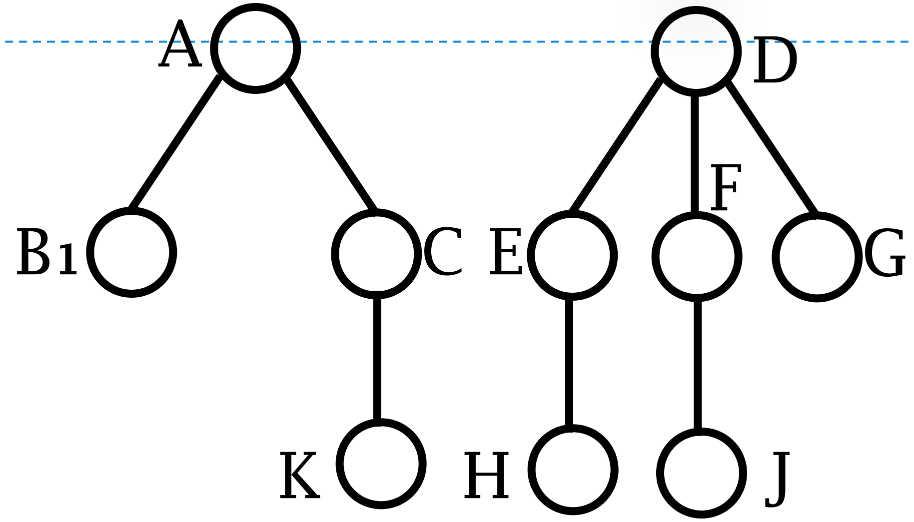




图书目录，杜威表示法

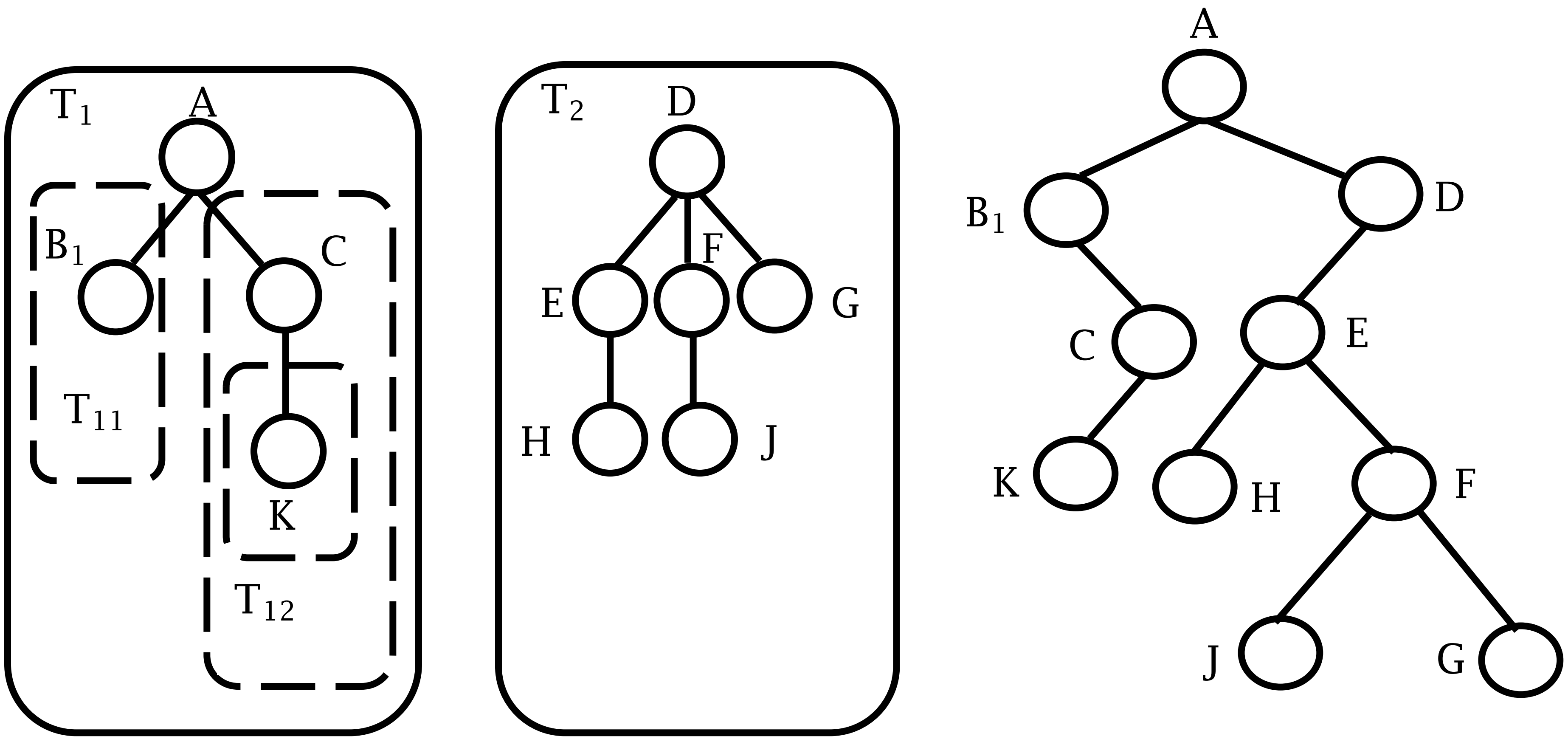
6 树
6.1 树的定义和基本术语
6.1.1 树和森林
6.1.2 森林与二叉树的等价转换
6.1.3 树的抽象数据类型
6.1.4 树的遍历
6.2 树的链式存储结构
6.2.1 “子结点表”表示方法
6.2.2 静态“左孩子/右兄弟”表示法
6.2.3 动态表示法
6.2.4 动态“左孩子/右兄弟”二叉链表表示法
6.2.5 父指针表示法及在并查集中的应用
6.3 树的顺序存储结构
6.3.1 带右链的先根次序表示
6.3.2 带双标记的先根次序表示
6.3.3 带度数的后根次序表示
6.3.4 带双标记的层次次序表示
6.4 K叉树
6.5 树知识点总结

森林与二叉树的等价转换



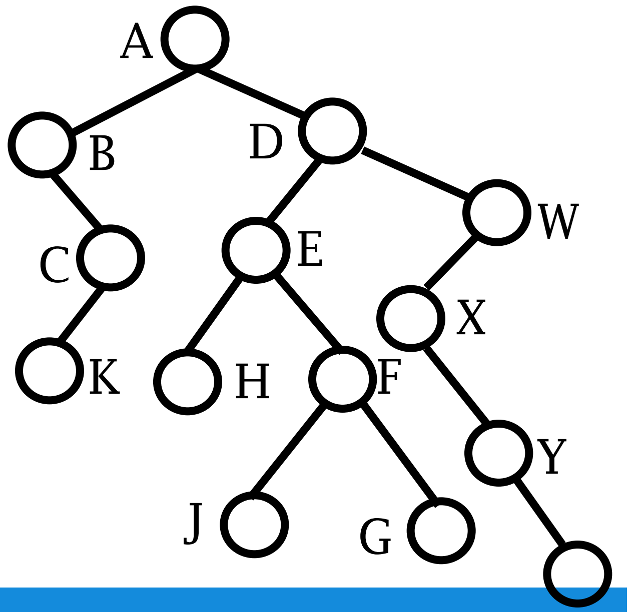
- **森林(forest)**：零棵或多棵 不相交 的树的集合（通常是有序）
- 树与森林的对应
  - 一棵树，删除树根，其子树就组成了森林
  - 加入一个结点作为根，森林就转化成了一棵树
- 森林与二叉树之间可以相互转化，而且这种转换是一一对应的
  - 森林的相关操作都可以转换成对二叉树的操作

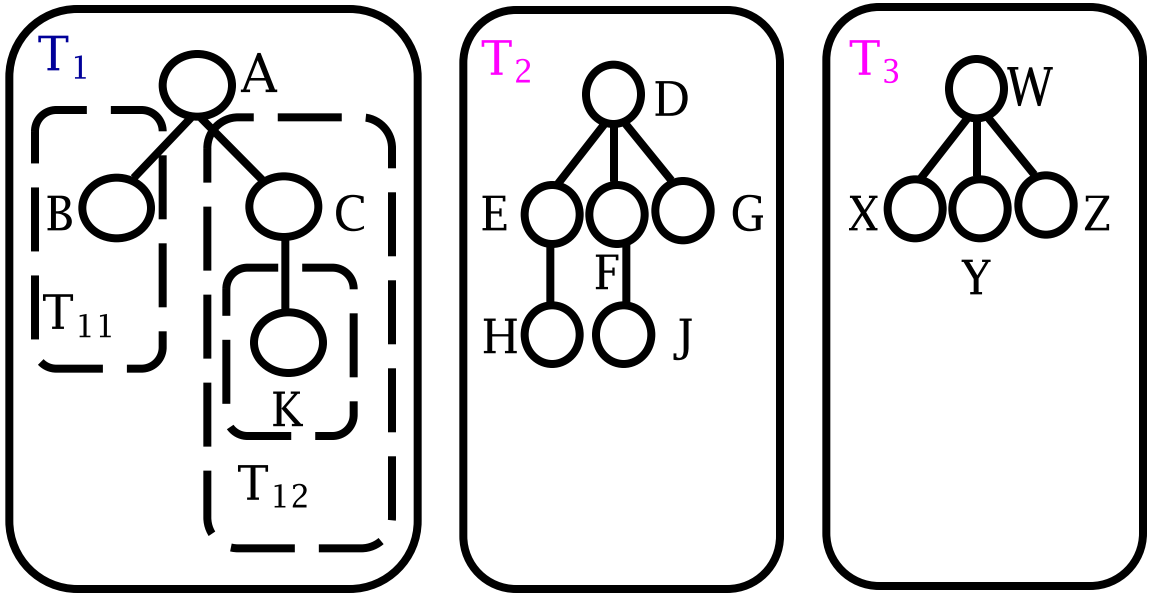
森林与二叉树如何对应?



森林转化成二叉树的形式定义

- 有序集合 $F = \{T_1, T_2, \ldots, T_n\}$ 是树 $T_1, T_2, \ldots, T_n$ 组成的森林，递归转换成二叉树B(F)：
  - 若 F 为空，即 n = 0，则 B(F) 为空。
  - 若 F 非空，即 n > 0，则 B(F) 的根是森林中第一棵树 T_1 的根 W_1，
    B(F) 的左子树是树 T_1 中根结点 W_1的子树森林 $F = \{T_{11}, \ldots, T_{1m}\}$ 转换成的二叉树$B(T_{11}, \ldots, T_{1m})$；
    B(F)的右子树是从森林 $F' = \{T_2, \ldots, T_n\}$ 转换而成的二叉树





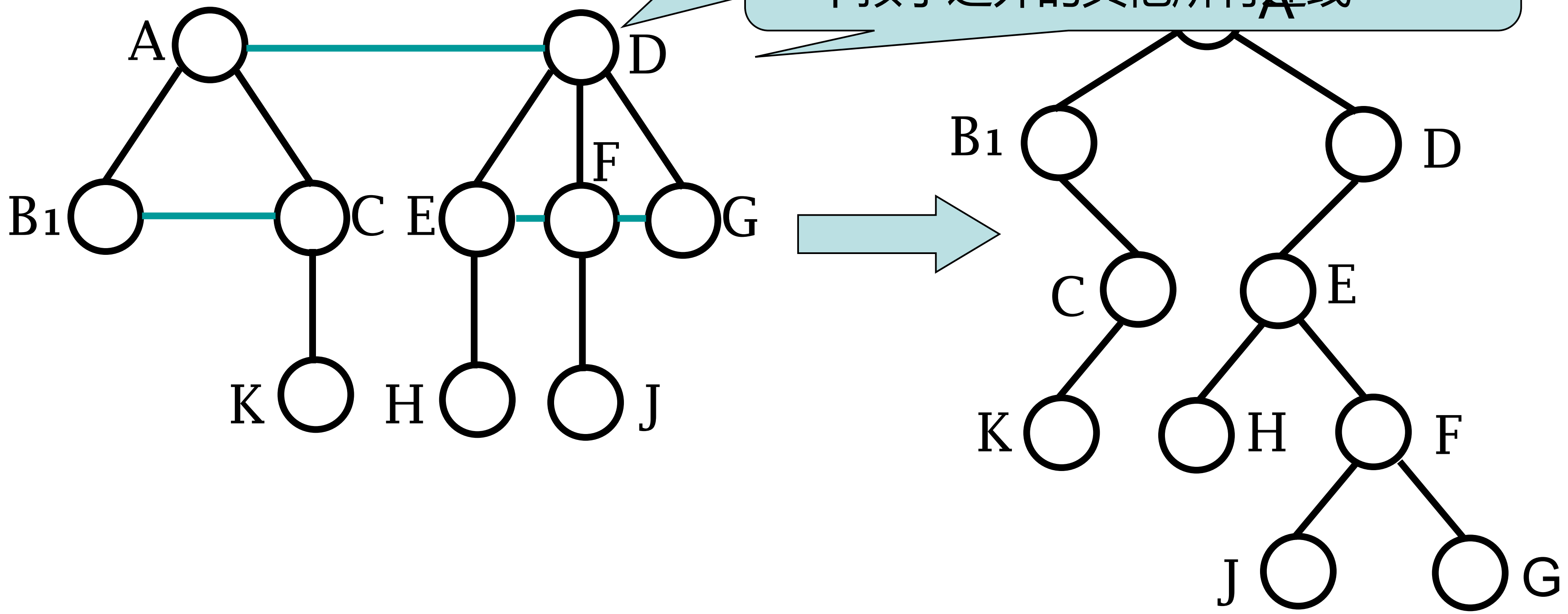
森林转化为二叉树

的所有兄弟结点之

点，去掉除了与第

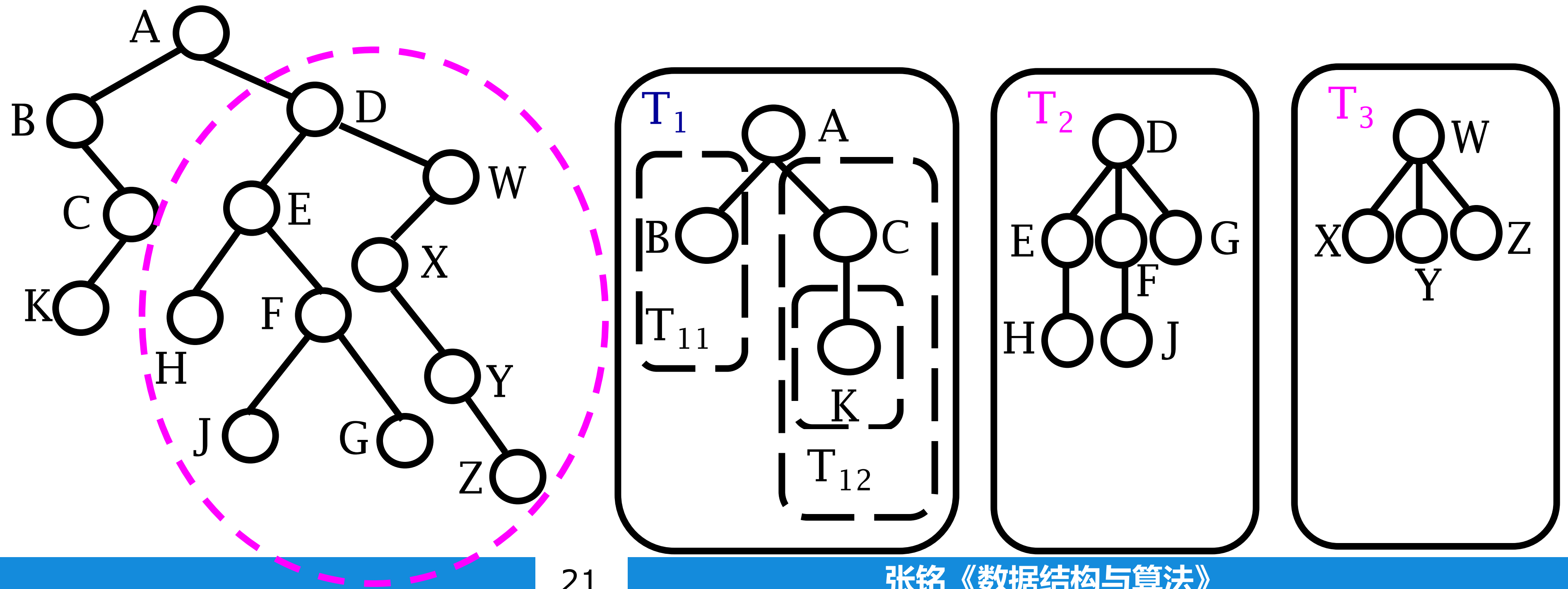
他所有连线

第三步：调整位置



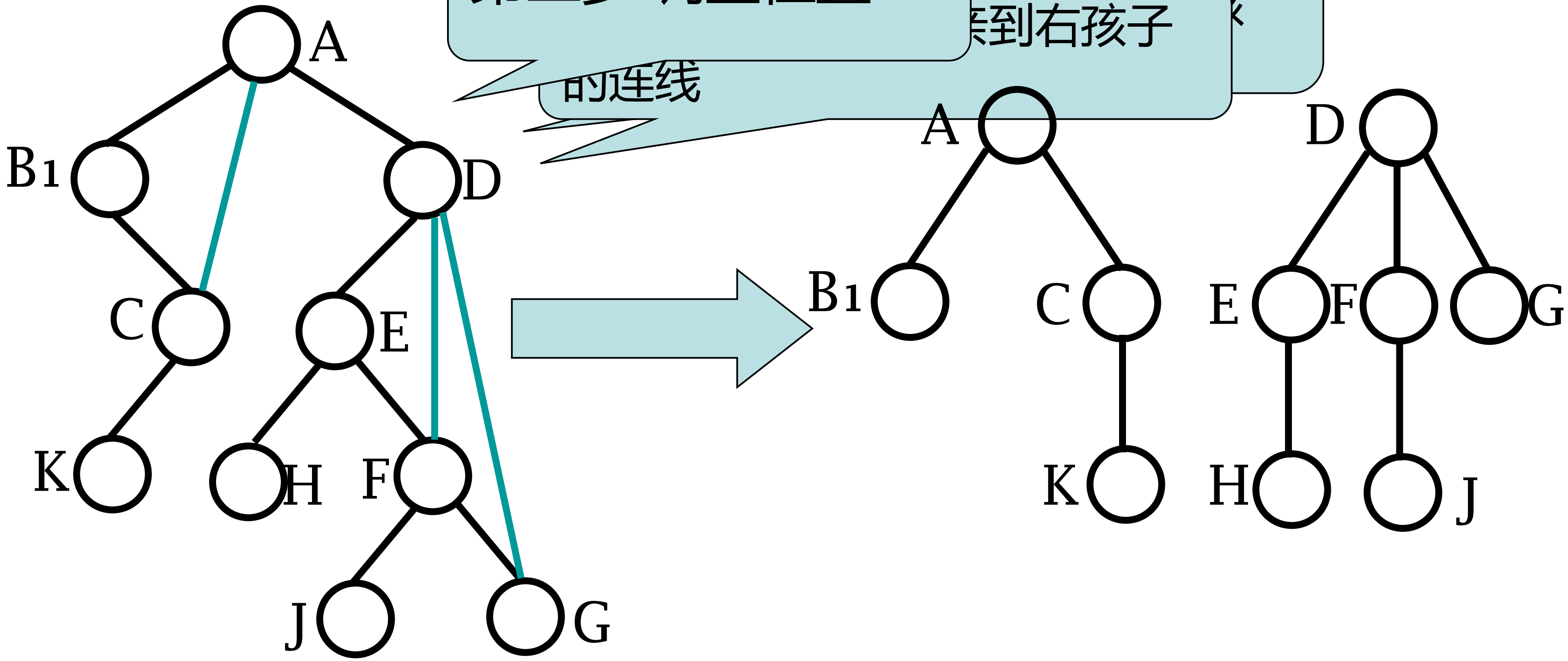
二叉树转化成森林或树的形式定义

- 设B是一棵二叉树，root是 B 的根，B_L 是 root 的左子树，B_R 是 root 的右子树，则对应于二叉树B的森林或树 F(B) 的形式定义是：
  - 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 $F(B_R)$，其中树 T_1 的根为 root，root 的子树为 $F(B_L)$



二叉树转换为森林

第三步:调整位置

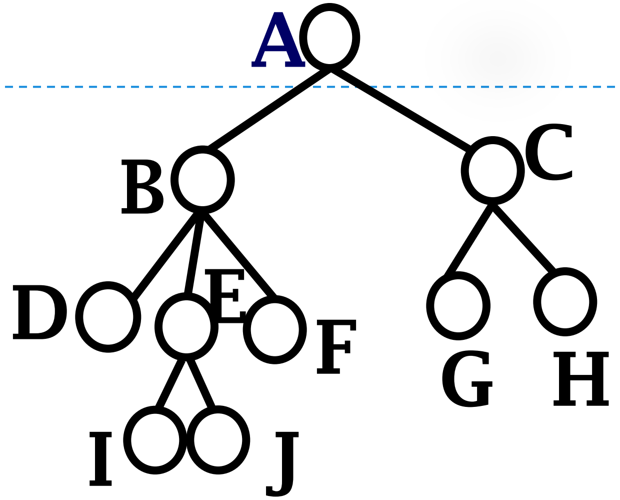


思考

- 1. 树也是森林吗？

- 2. 为什么要建立二叉树与森林的对应关系？

第6章　树



- 树的定义和基本术语
  - 树和森林
  - 森林与二叉树的等价转换
  - 树的抽象数据类型
  - 树的遍历
- 树的链式存储结构
- 树的顺序存储结构
- K叉树

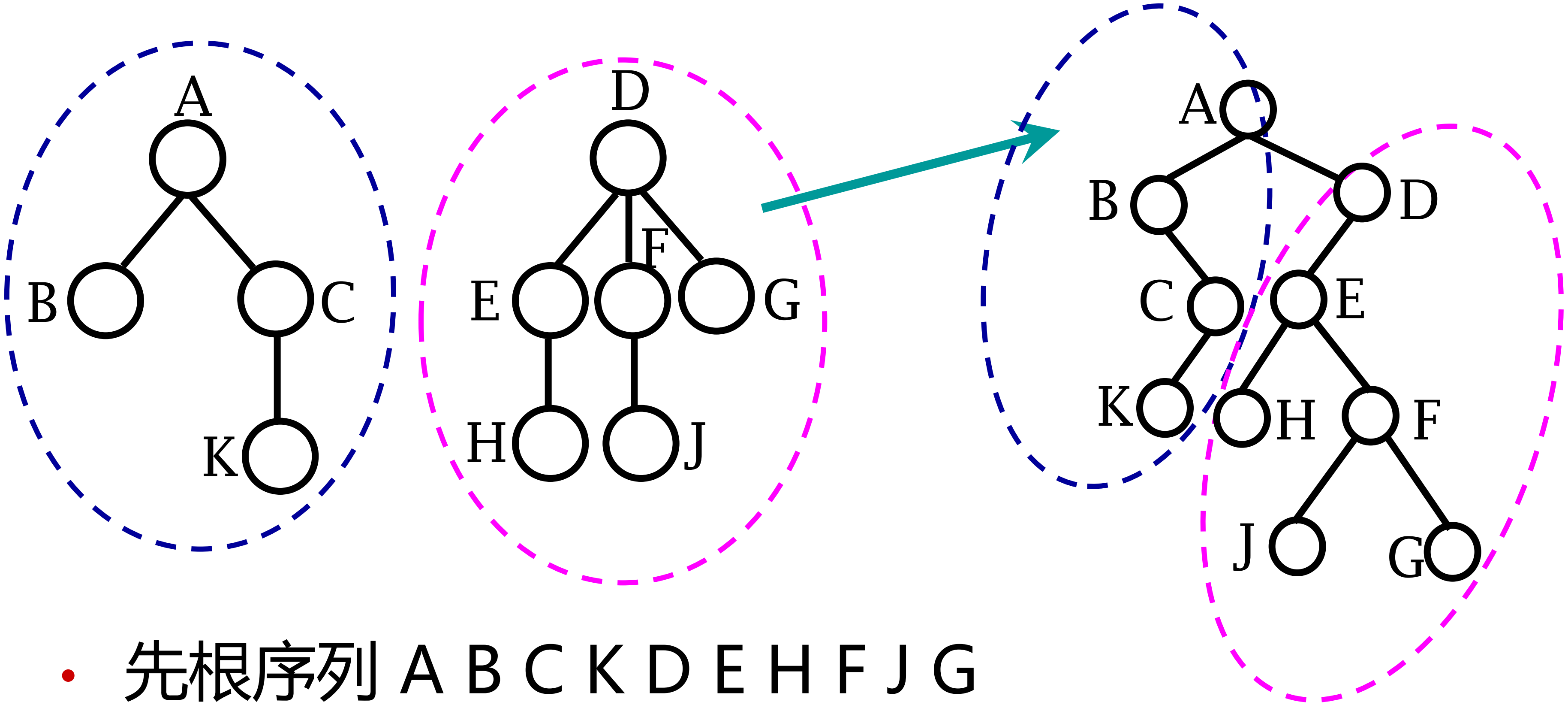
树的抽象数据类型

```
template<class T>
class TreeNode  {                                  // 树结点的ADT
public:
  TreeNode(const T& value);                        // 拷贝构造函数
  virtual ~TreeNode() {};                          // 析构函数
  bool isLeaf();                                   // 判断当前结点是否为叶结点
  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(const T& value);                   // 设置当前结点的值
  void setChild(TreeNode<T> *pointer);             // 设置左孩子
  void setSibling(TreeNode<T> *pointer);           // 设置右兄弟
  void InsertFirst(TreeNode<T> *node);             // 以第一个左孩子身份插入结点
  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
    void CreateRoot(const T& rootValue);                       // 创建值为rootValue的根结点
    bool isEmpty();                                            // 判断是否为空树
    TreeNode<T>* Parent(TreeNode<T> *current);                 // 返回父结点
    TreeNode<T>* PrevSibling(TreeNode<T> *current);            //返回前一个兄弟
    void DeleteSubTree(TreeNode<T> *subroot);                  // 删除以subroot子树
    void RootFirstTraverse(TreeNode<T> *root);                 // 先根深度优先遍历树
    void RootLastTraverse(TreeNode<T> *root);                  // 后根深度优先遍历树
    void WidthTraverse(TreeNode<T> *root);                     // 广度优先遍历树
};
```

森林的遍历

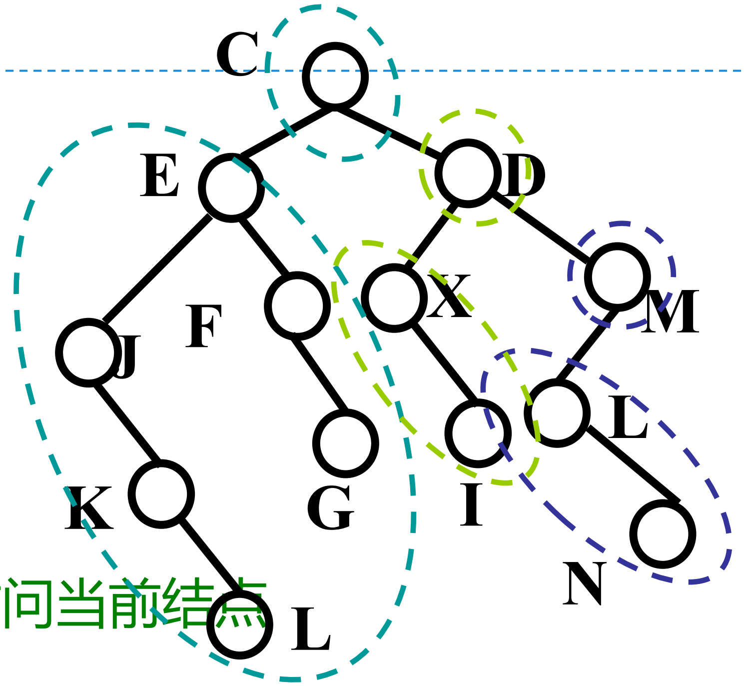


- 先根序列 A B C K D E H F J G
- 后根序列 B K C A H E J F G D

遍历森林vs遍历二叉树

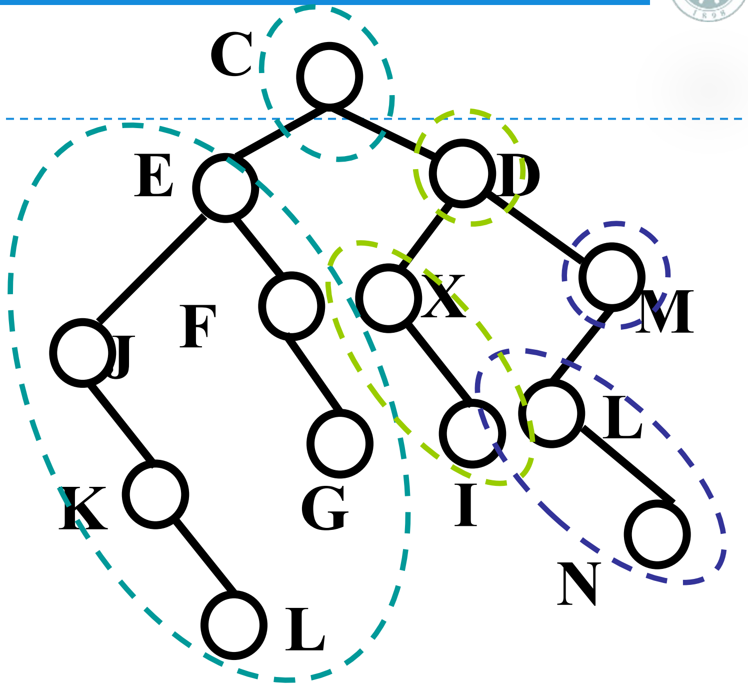
- 先根次序遍历森林
  - 前序法遍历二叉树
- 后根次序遍历森林
  - 按中序法遍历对应的二叉树
- 中根遍历?
  - 无法明确规定根在哪两个子结点之间

先根深度优先遍历森林




```
template<class T>
void Tree<T>::RootFirstTraverse(
        TreeNode<T> * root)  {
    while (root != NULL) {
        Visit(root->Value());                    // 访问当前结点
        // 遍历第1棵树根的子树森林(树根除外)
        RootFirstTraverse(root->LeftMostChild());
        root = root->RightSibling();             // 遍历其他树
    }
}
```

后根深度优先遍历森林



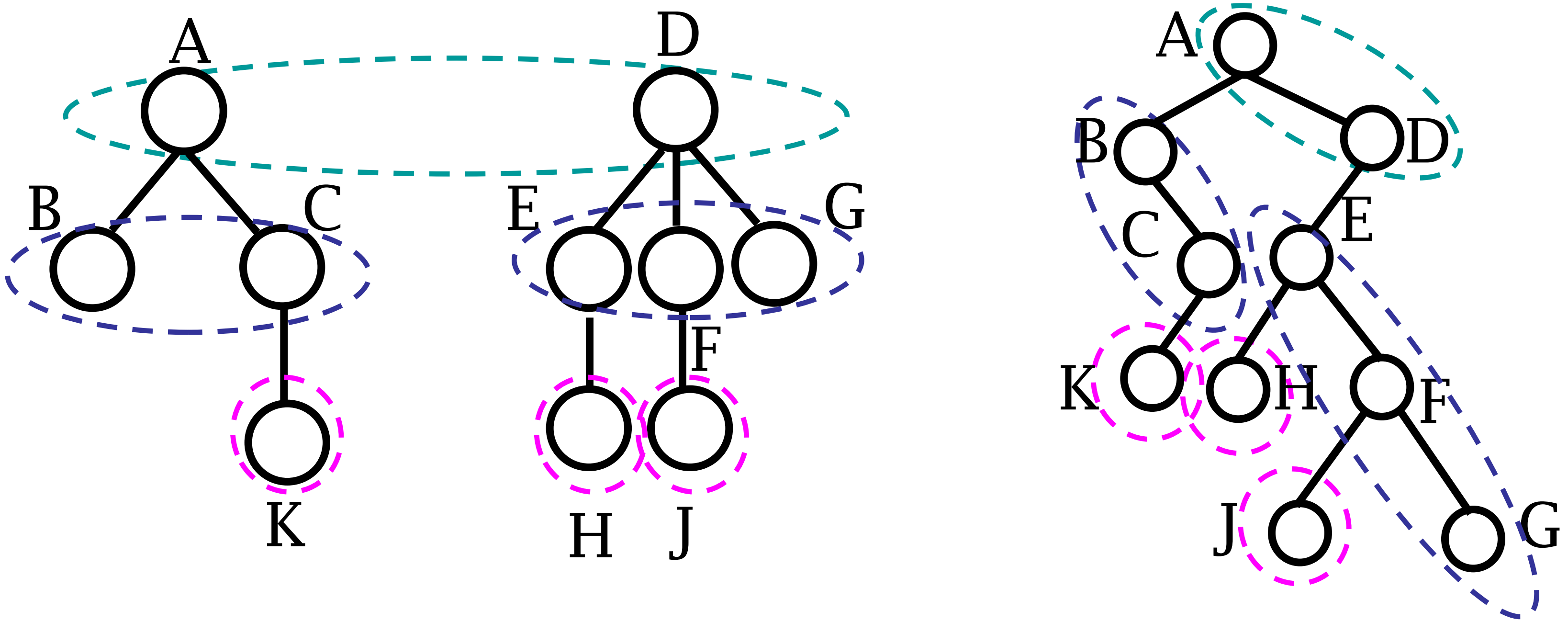

```
template<class T>
void Tree<T>::RootLastTraverse(
        TreeNode<T> * root)  {
    while (root != NULL) {
      // 遍历第一棵树根的子树森林
      RootLastTraverse(root->LeftMostChild());
      Visit(root->Value());           // 访问当前结点
      root = root->RightSibling();  // 遍历其他树
    }
}
```

宽度优先遍历森林

- 宽度优先遍历
  - 也称广度优先遍历
  - 或称层次遍历

- a) 首先依次访问层数为0的结点
- b) 然后依次访问层数为1的结点
- c) 直到访问完最下一层的所有结点

广度优先遍历森林



- 森林广度优先：A D B C E F G K H J
- **看二叉链存储结构的右斜线**

广度优先遍历森林

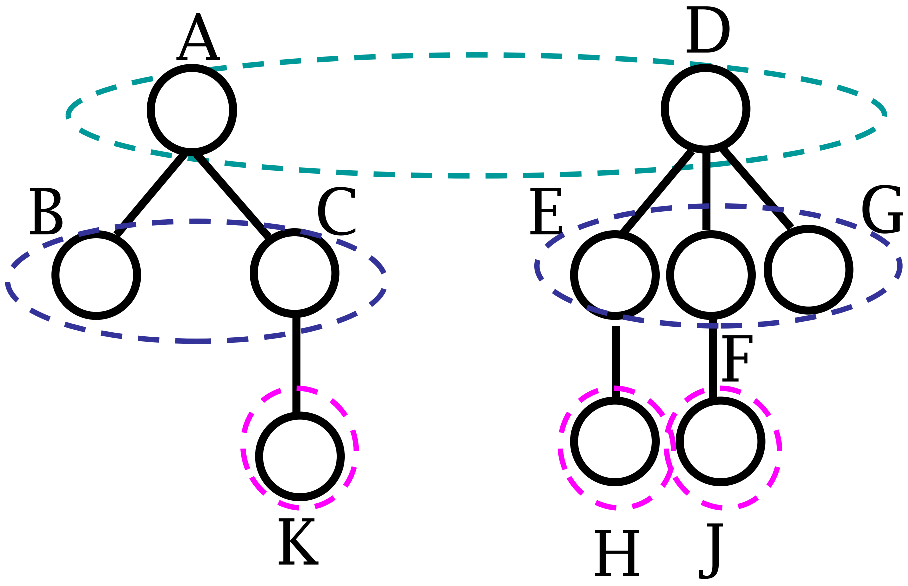
```
template<class T>
void Tree<T>::WidthTraverse(TreeNode<T> * root)  {
   using std::queue;                              // 使用STL队列
   queue<TreeNode<T>*> aQueue;
   TreeNode<T> * pointer = root;
   while (pointer != NULL) {
      aQueue.push(pointer);                       // 当前结点进入队列
      pointer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

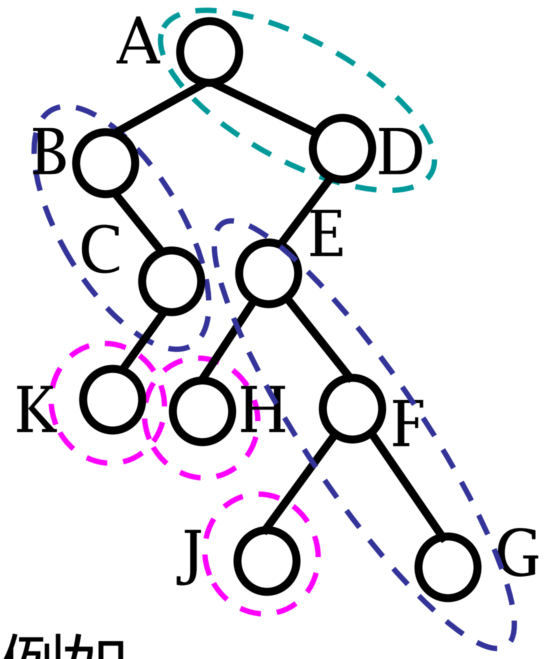
思考

- 1. 能否直接用二叉树前序遍历框架来编写森林的先根遍历?
- 2. 能否直接用二叉树中序遍历框架来编写森林的后根遍历?
- 3. 森林的非递归深搜框架?

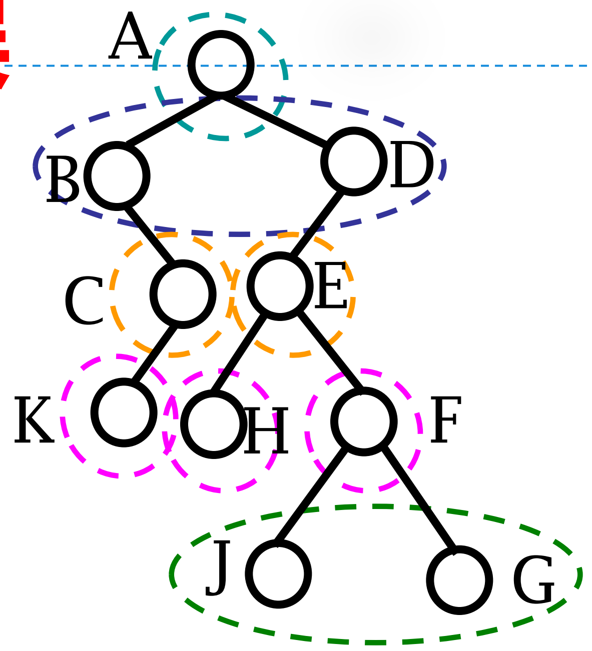
6.1 树的定义和基本术语

思考：宽搜的各种观点



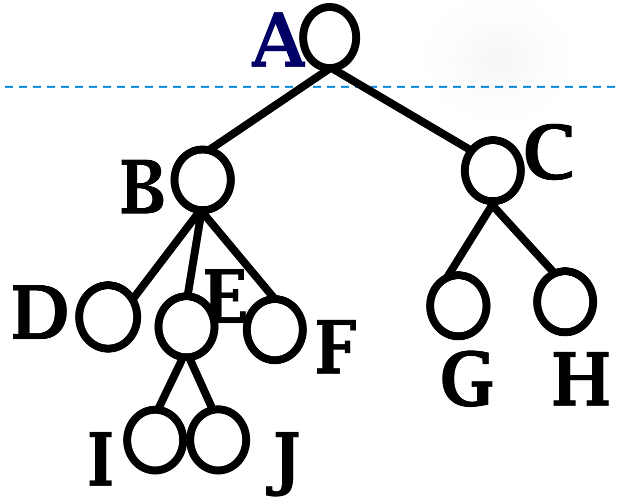


错误



- 不能用二叉树的广度遍历模板。例如，
  - 上左图，森林广度优先：A D B C E F G K H J
    - **看二叉树的右斜线**
  - 上右图，二叉树广度：　A B D C E K H F J G
    - **看平行横线**

第6章　树

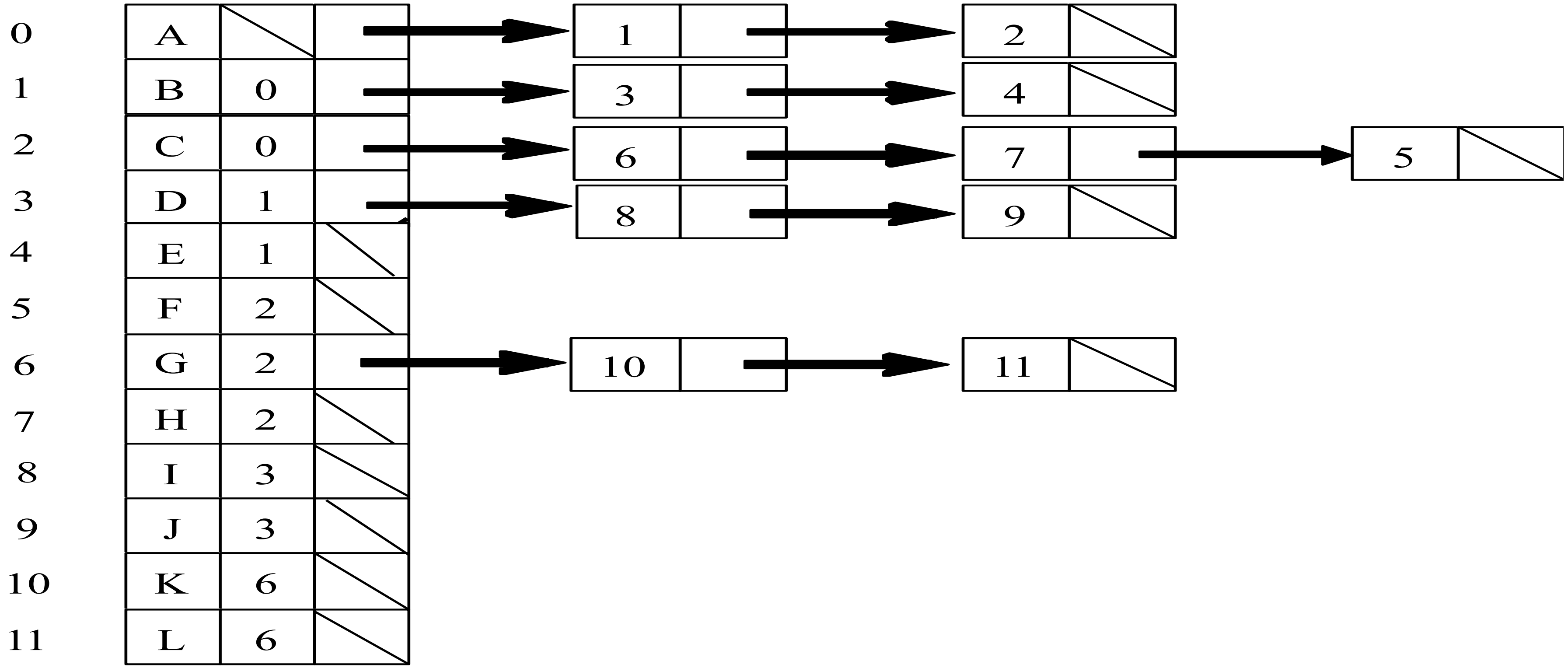


- 树的定义和基本术语
- 树的链式存储结构
  - “子结点表”表示方法
  - 静态“左孩子/右兄弟”表示法
  - 动态表示法
  - 动态“左孩子/右兄弟”表示法
  - 父指针表示法及其在并查集中的应用
- 树的顺序存储结构
- K叉树

"子结点表"表示方法

list of children，就是图的邻接表。

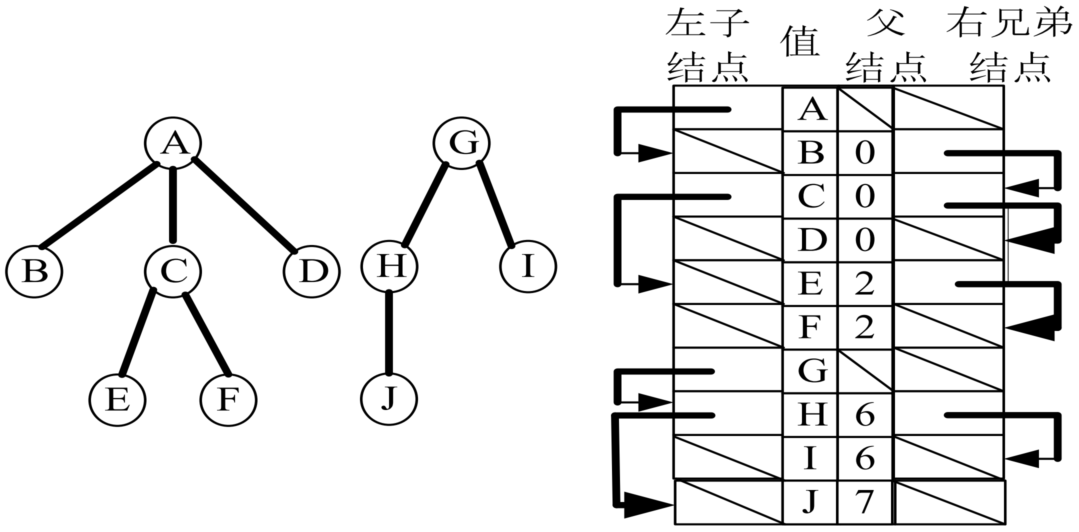
[地址，值，父结点，子结点]



6.2 树的链式存储结构

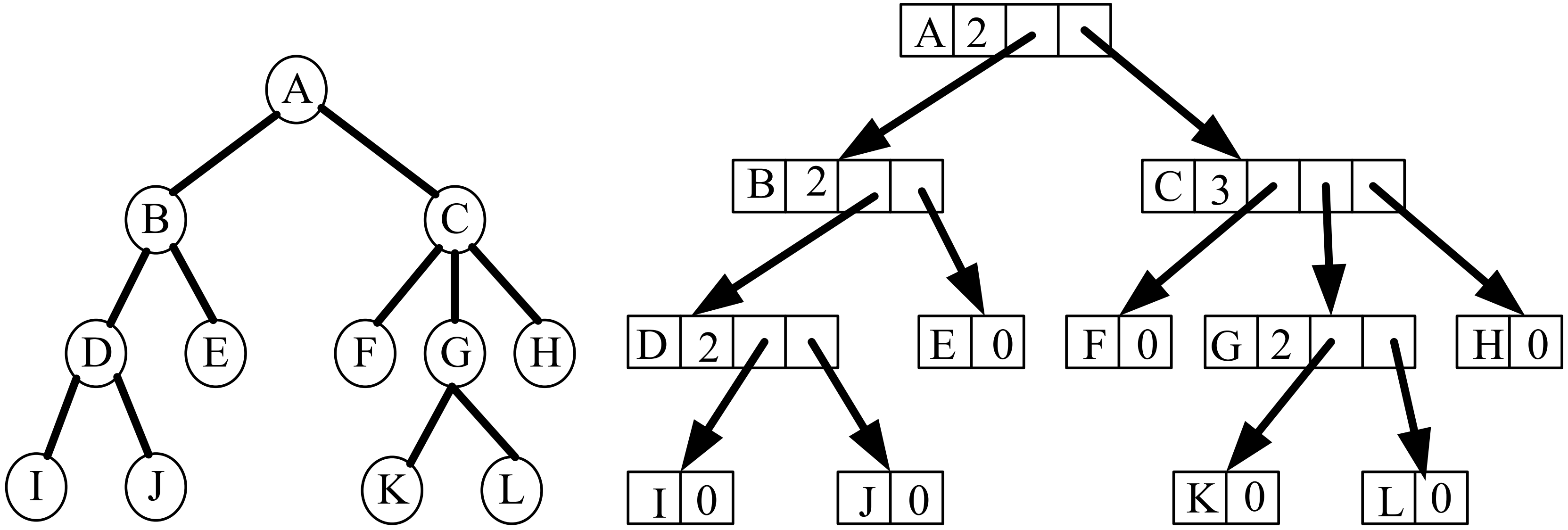
静态“左孩子/右兄弟”表示法

- 在数组中存储的“子结点表”



动态表示法

- 每个结点分配可变的存储空间
  - 子结点数目发生变化，需要重新分配存储空间

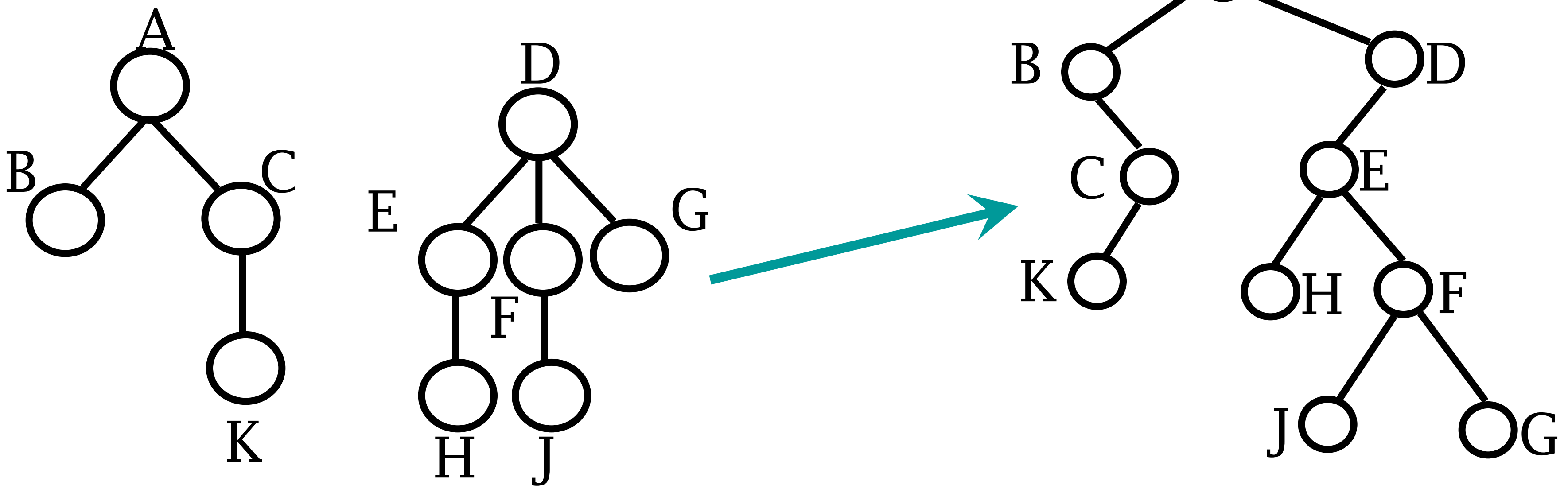


(a) 树　(b) 树的实现

动态“左子/右兄”二叉链表示法

- 左孩子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
- 根的右链就是森林中每棵树的根结点

pChild	info	pSibling



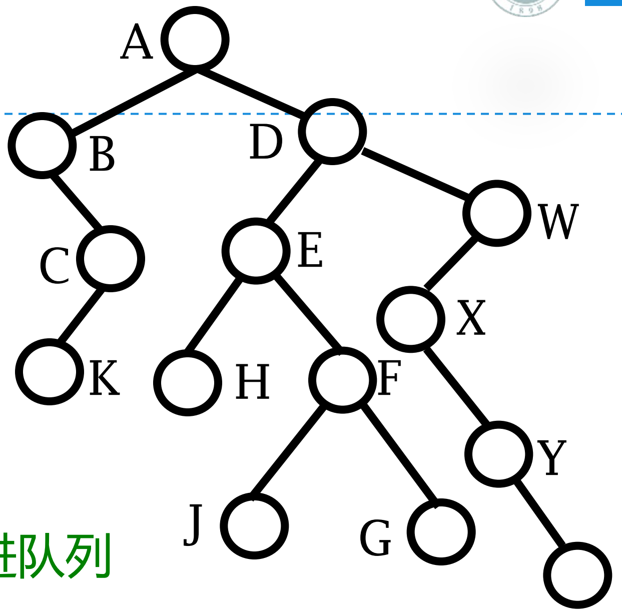
动态二叉链表树的关键实现细节

```
// 在TreeNode的抽象类中增加以下私有数据成员
private:
T m_Value;                    // 树结点的值
TreeNode<T> *pChild;          // 第一个左孩子指针
TreeNode<T> *pSibling;        // 右兄弟指针
```

寻找当前结点的父结点

```
template<class T>
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> *father = upperlevelpointer  = NULL;           // 记录父结点
if (current != NULL && pointer != current)  {
while (pointer != NULL) {                                  // 森林中所有根结点进队列
      if (current == pointer)                              // 森林中所有第一层根的父为空
break;
      aQueue.push(pointer);                                // 当前结点进队列
pointer=pointer-> RightSibling();                          // 指针指向右
}
```

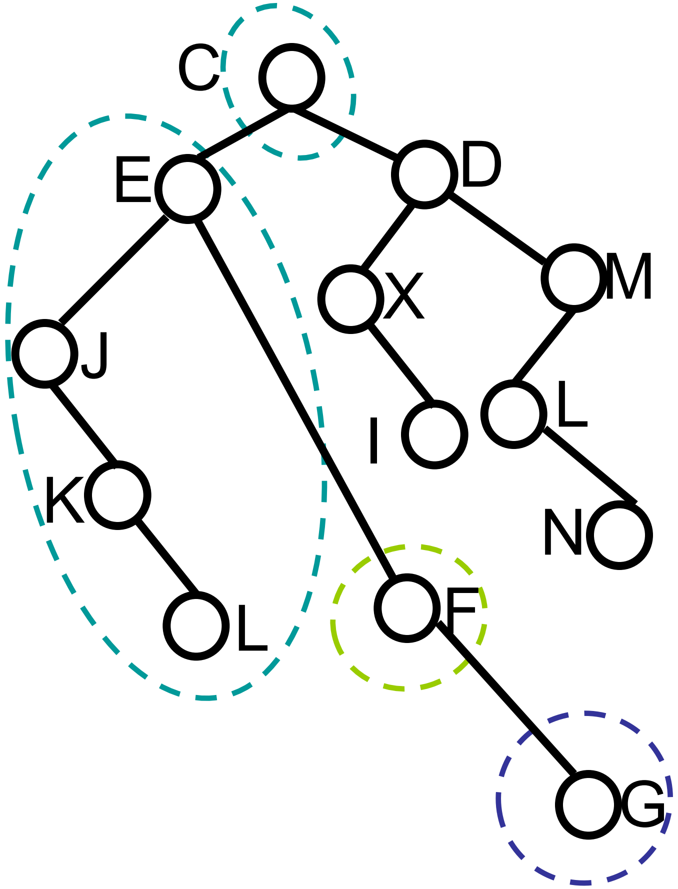
寻找当前结点的父结点




```
while (!aQueue.empty()) {
    pointer = aQueue.front();                       // 取队列首结点指针
    aQueue.pop();                                   // 当前元素出队列
    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) {
           father = upperlevelpointer;              // 返回父
           break;}
        else {
           aQueue.push(pointer);
           pointer = pointer->RightSibling();}
      }
    }
}
aQueue. clear( );                                   // 清空队列，也可以不写（局部变量）
return father;
}
```

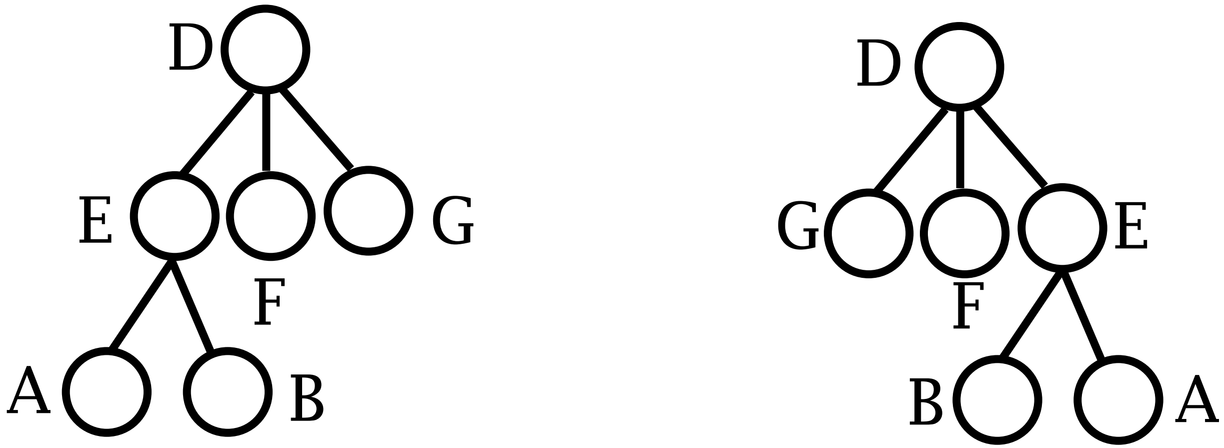
思考：下面的算法是否能遍历森林?

```
template <class T>
void Traverse(TreeNode <T> * rt) {
    if (rt==NULL)  return;
    Visit(rt);
    TreeNode * temp = rt-> LeftMostChild();
    while (temp != NULL) {
        Traverse(temp);
        temp = temp->RightSibling();
    }
}
```

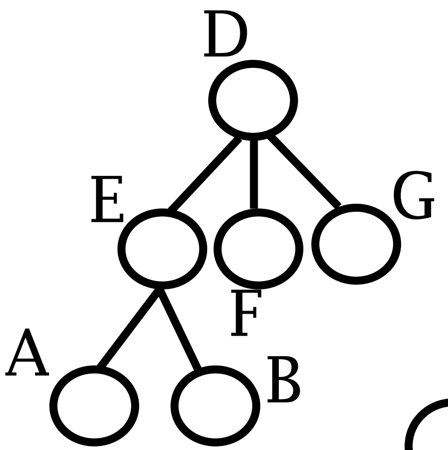


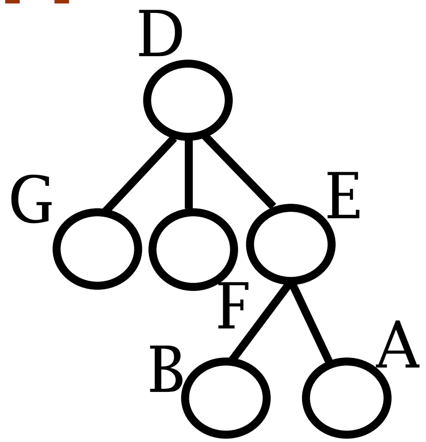
思考：灵活应用遍历框架

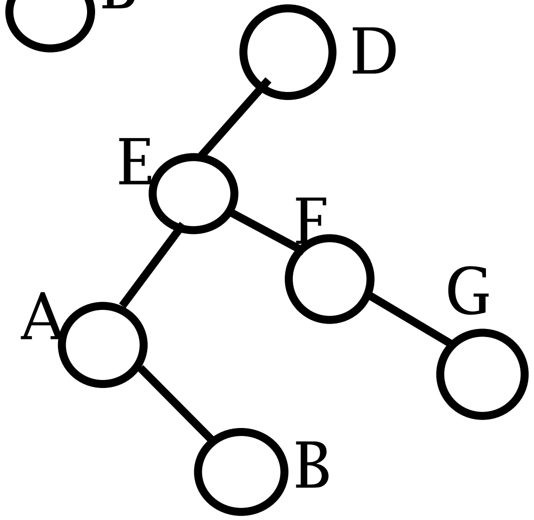
例：森林镜面映射

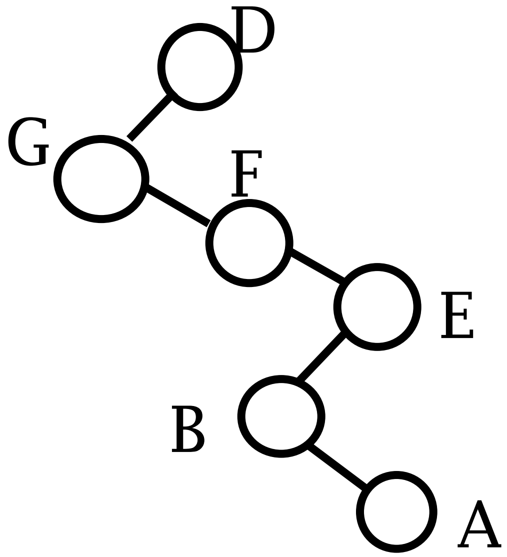


映射后









删除以root为代表的森林的所有结点

```
template <class T>
void Tree<T>::DestroyNodes(TreeNode<T>* root)  {
    if (root) {
        DestroyNodes(root->LeftMostChild());//递归删除第一子树
        DestroyNodes(root->RightSibling());   //递归删除其他子树
        delete root; //删除根结点
    }
}
```

思考：删除以subroot为根的子树

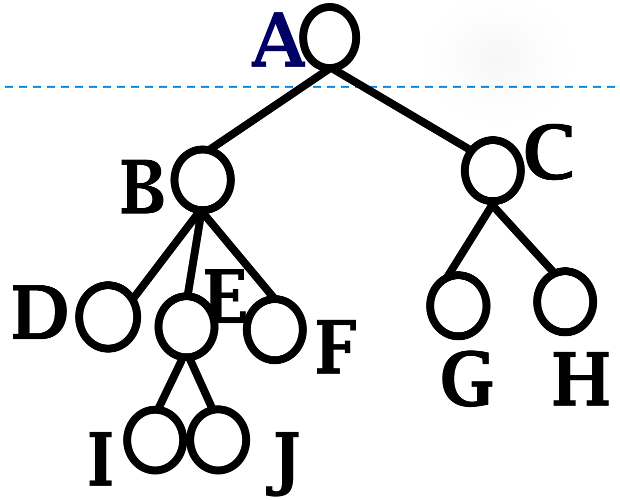
请注意待删除的子树是否为空、subroot有无父指针等情况的判断。

考虑删除以后各项相关链接的修改顺序。

删除以subroot为根的子树

```
template<class T>
void Tree<T>::DeleteSubTree(TreeNode<T> *subroot)  {
    if (subroot == NULL) return;// 若待删除的子树为空则返回
    TreeNode<T> *pointer = Parent (subroot); // 找subroot的父结点
    if (pointer == NULL) {// subroot没有父，则是某个树根
      pointer =  root;
      while (pointer->RightSibling() != subroot)// 顺右链找左邻树根
        pointer = pointer->RightSibling();
      pointer->setSibling(subroot->RightSibling()); // 前后挂接，脱链
    }
    else if (pointer->LeftMostChild() == subroot) // subroot为最左子
      pointer->setChild(subroot->RightSibling()); // 挂新的最左
    else {// subroot有左兄弟的情况
      pointer = pointer->LeftMostChild();   // 下降到最左兄弟
      while (pointer->RightSibling() != subroot)// 顺右链找左邻兄弟
        pointer = pointer->RightSibling();
      pointer->setSibling(subroot->RightSibling()); // 前后挂接，脱链
    }
    subroot->setSibling(NULL);// 非常重要，丢了会出错
    DestroyNodes(subroot);  // 删除以subroot代表的子森林的所有结点
}
```

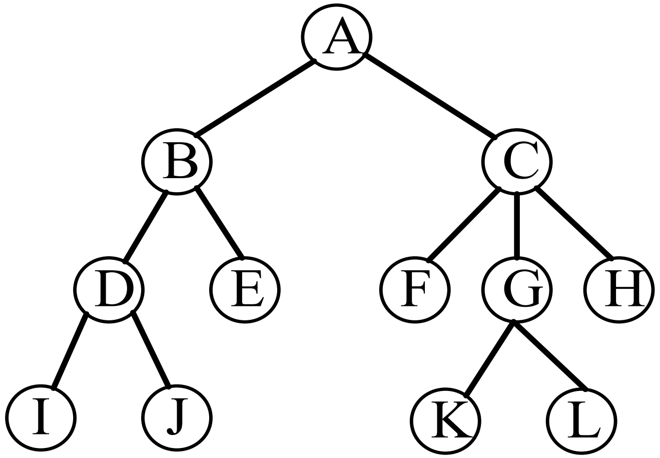
第6章　树



- 树的定义和基本术语
- **树的链式存储结构**
  - “子结点表”表示方法
  - 静态“左孩子/右兄弟”表示法
  - 动态表示法
  - 动态“左孩子/右兄弟”表示法
  - **父指针表示法及其在并查集中的应用**
- 树的顺序存储结构
- K叉树

父指针表示法

- 只需要知道父结点的应用
- 只需要保存一个指向其父结点的指针域，称为 父指针(parent pointer)表示法
- 用数组存储树结点，同时在每个结点中附设一个指针指示其父结点的位置



结点索引	0	1	2	3	4	5	6	7	8	9	10	11
值	A	B	C	D	E	F	G	H	I	J	K	L
父结点索引		0	0	1	1	2	2	2	3	3	6	6

父指针表示法：算法

- 查询结点的根
  - 从一个结点出发找出一条向上延伸到达根的祖先路径
    - O(k)，k为树高
- 判断两个结点是否在同一棵树
  - 两个结点根结点相同，它们一定在同一棵树中
  - 如果其根结点不同，那么两个结点就不在同一棵树中

并查集

并查集 是一种特殊的集合，由一些不相交子集构成，合并查集的基本操作是：

- Find：　查询结点所在集合
- Union：归并两个集合

- 并查集是重要的抽象数据类型
  - 应用于求解等价类等等问题

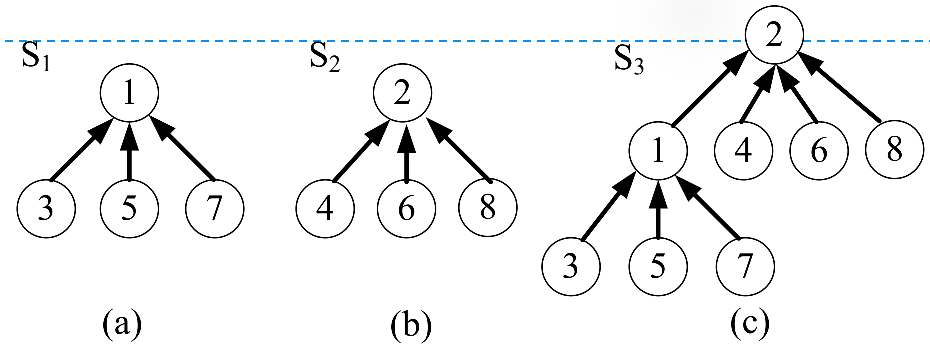
等价关系

- 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(equivalence classes)

- 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S_1，S_2，…S_r，这些集合的并为S

用树来表示等价类的并查



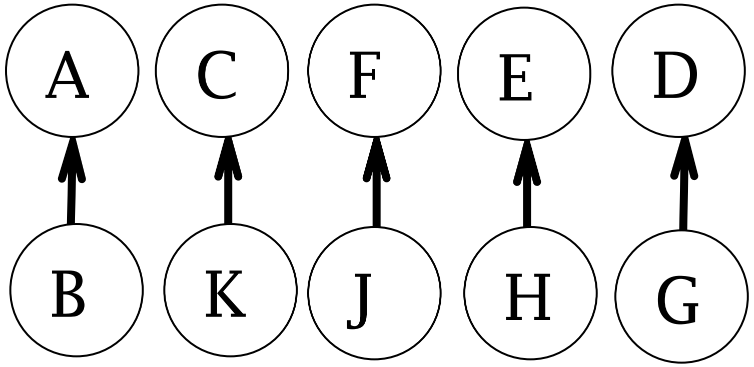
- 用一棵树代表一个集合
  - 集合用父结点代替
  - 若两个结点在同一棵树中，则它们处于同一个集合
- 树的实现
  - 存储在静态指针数组中
  - 结点中仅需保存父指针信息

UNION/FIND算法示例(1)

对这5个等价对进行处理 (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

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

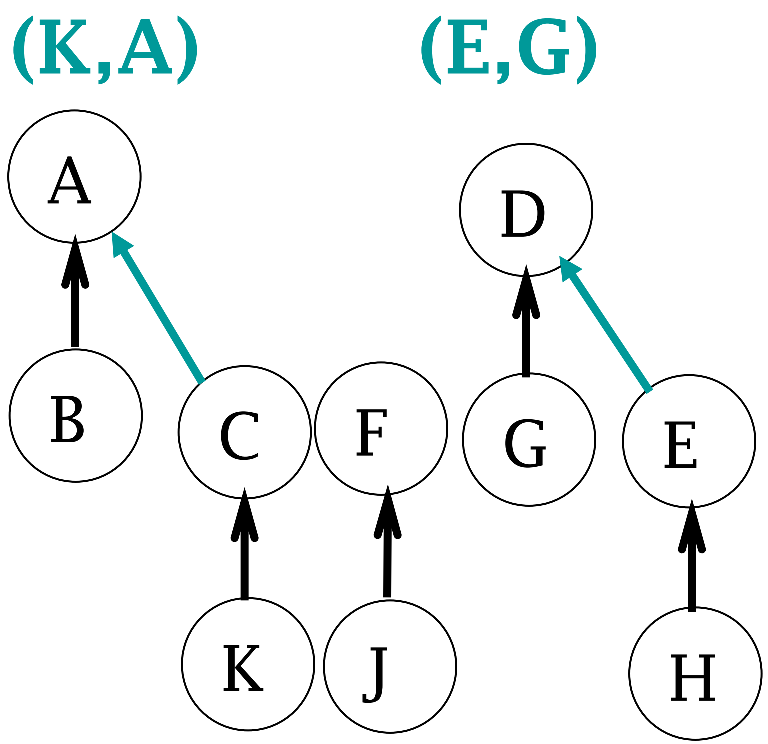


UNION/FIND算法示例(1)

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自己是根结点，A！=C,所以两棵树要合并

	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



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:
  T value;                                       //结点的值
  ParTreeNode<T>* parent;                        //父结点指针
  int nCount;                                    //集合中总结点个数
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);           //设置父指针
  int getCount();                                //返回结点数目
  void setCount(const int count);                //设置结点数目
};
```

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
  int Size;                                   // 数组大小
  ParTreeNode<T>*
  Find(ParTreeNode<T>* node) const;           // 查找node结点的根结点
  ParTree(const int size);                    // 构造函数
  virtual ~ParTree();                         // 析构函数
  void Union(int i,int j);                    // 把下标为i，j的结点合并成一棵子树
  bool Different(int i,int j);                // 判定下标为i，j的结点是否在一棵树中
};
```

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

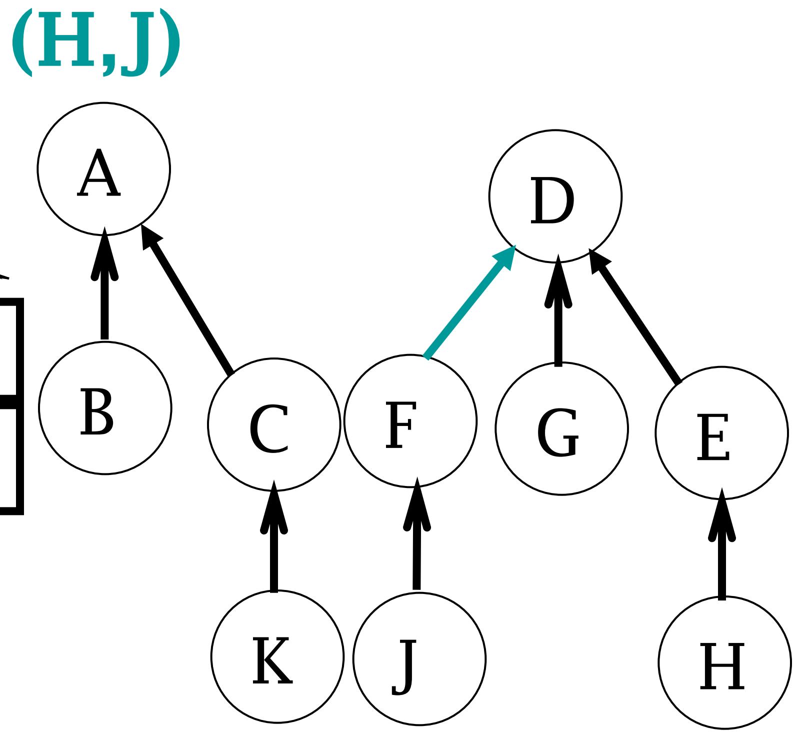
```
template<class T>
void ParTree<T>::Union(int i,int j) {
ParTreeNode<T>* pointeri = Find(&array[i]);     //找到结点i的根
ParTreeNode<T>* pointerj = Find(&array[j]);     //找到结点j的根
if (pointeri != pointerj) {
  if(pointeri->getCount() >= pointerj->getCount()) {
        pointerj->setParent(pointeri);
    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

UNION/FIND算法示例(2)

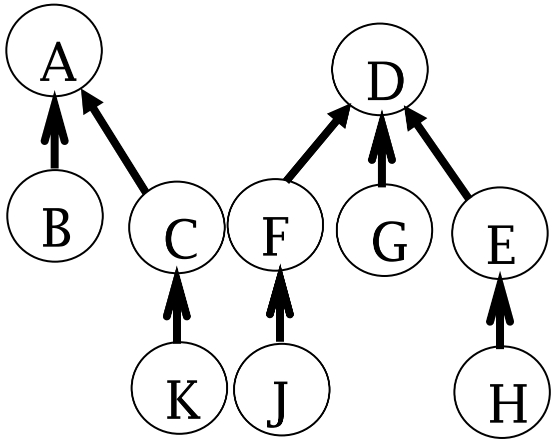
最后使用加权合并规则处理等价对（H,J）

依据加权合并规则，以F为根的树结点个数少，故将F指向D

	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



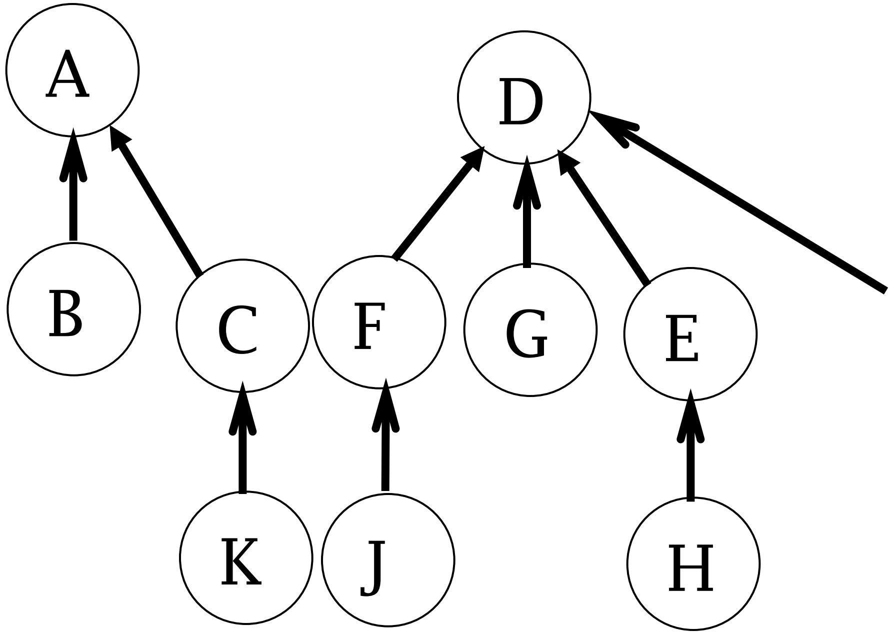
路径压缩



- 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)

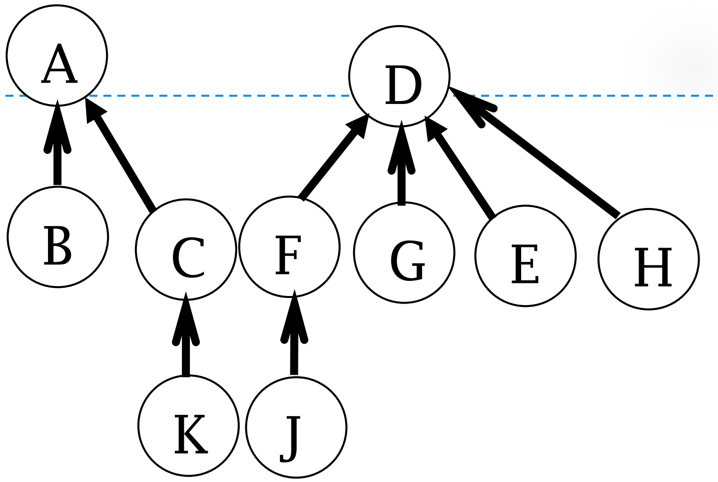
使用路径压缩规则处理Find(H)



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

6.2 树的链式存储结构

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();
}
```

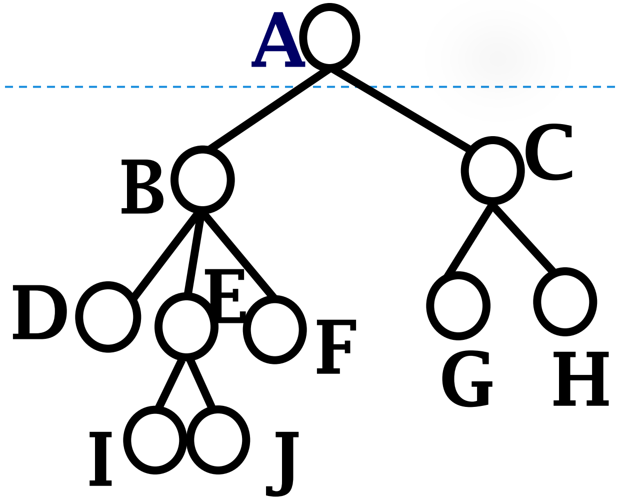
路径压缩使Find开销接近于常数

- 权重 + 路径压缩
- 对n个结点进行n次Find操作的开销为$O(n\alpha(n))$，约为$\Theta(n\log^{*}n)$
  - $\alpha(n)$是单变量Ackermann函数的逆，它是一个增长速度比logn慢得多但又不是常数的函数
  - $\log^{*}n$ 是在 $n = \log n \le 1$ 之前要进行的对 n 取对数操作的次数
  - $\log^{*}65536 = 4$（4次log操作）
- Find至多需要一系列n个Find操作的开销非常接近于$\Theta(n)$
  - 在实际应用中，$\alpha(n)$往往小于4

思考

- 可否使用动态指针方式实现父指针表示法?
- 查阅各种并查集权重和路径压缩优化方法，并讨论各种方法的异同和优劣

第6章　树



- 树的定义和基本术语
- 树的链式存储结构
- **树的顺序存储结构**
- K叉树

树的顺序存储结构

- 带右链的先根次序表示
- 带双标记的先根次序表示
- 带双标记的层次次序表示
- 带度数的后根次序表示

带右链的先根次序表示

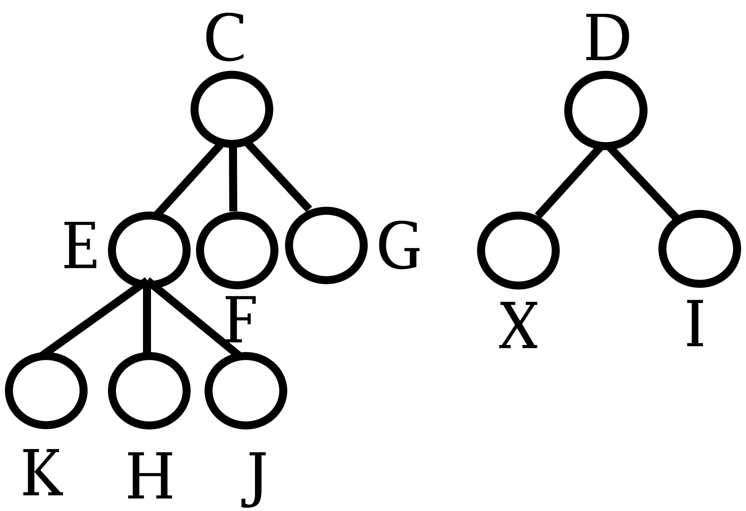
- 结点按先根次序顺序连续存储

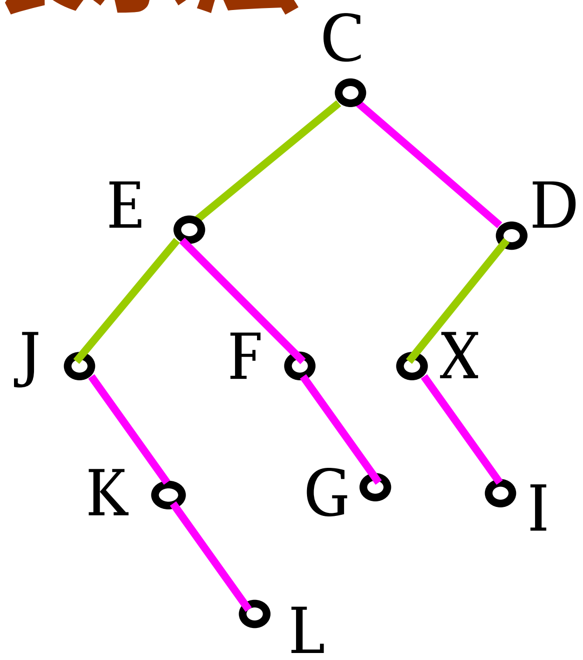
ltag	info	rlink

- 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

6.3 树的顺序存储结构

带右链的先根次序表示法

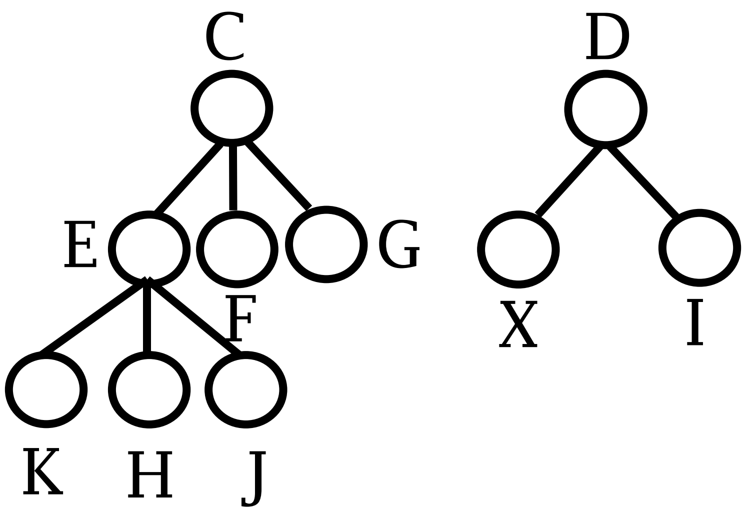


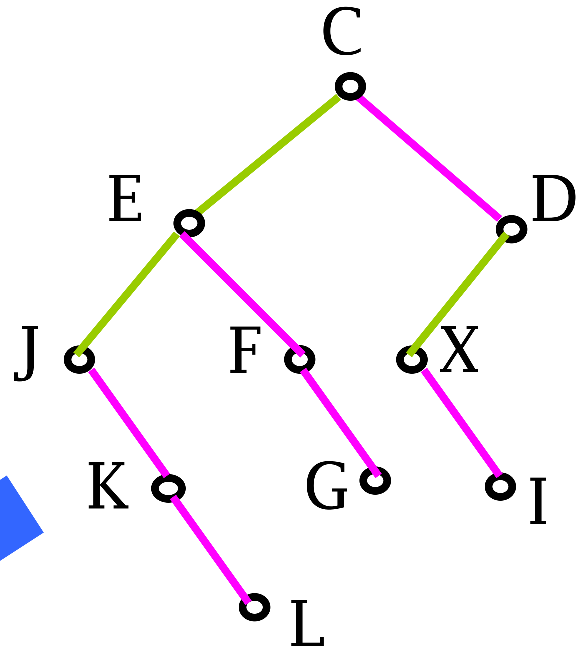


下标	0	1	2	3	4	5	6	7	8	9
rlink	7	5	3	4	-1	6	-1	-1	9	-1
info	C	E	J	K	L	F	G	D	X	I
ltag	0	0	1	1	1	1	1	0	1	1

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





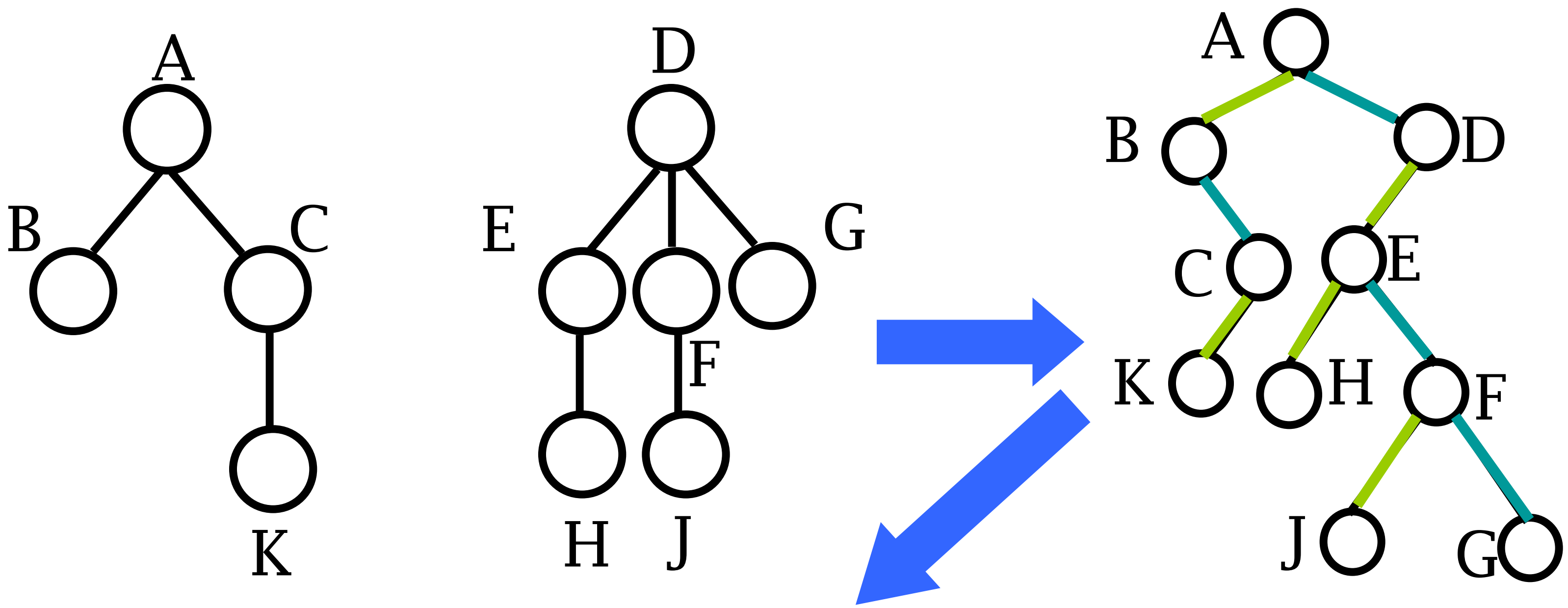
带双标记的先根次序表示

- 带右链的先根次序表示”中rlink也有冗余，可以把rlink指针替换为一个标志位rtag，成为“带双标记的先根次序表示”。其中，每个结点包括结点本身数据，以及两个标志位ltag和rtag，其结点的形式为：

ltag	info	rtag

由结点的先根次序以及ltag、rtag两个标志位，就可以确定树“左孩子/右兄弟”链表中结点的llink和rlink值。其中llink的确定与带右链的先根次序表示法相同。

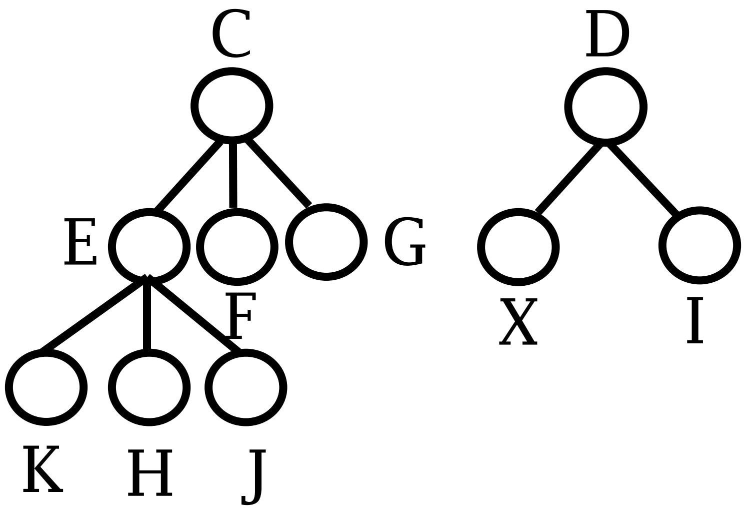
带双标记位的先根次序表示法

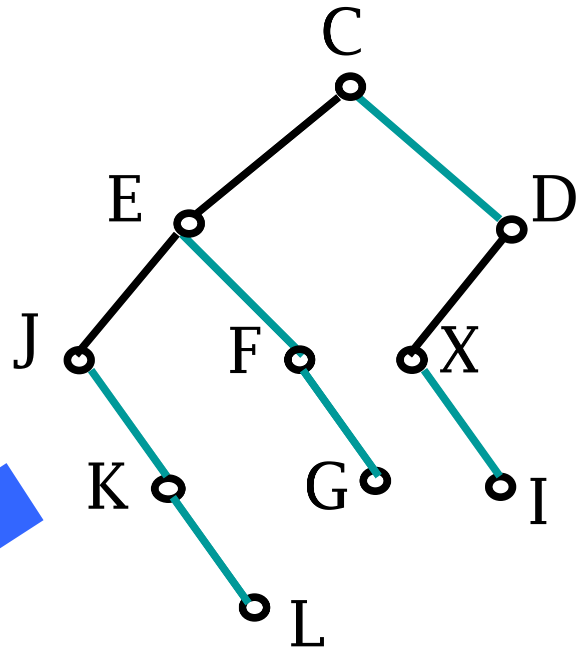


rtag	0	0	1	1	1	0	1	0	1	1
info	A	B	C	K	D	E	H	F	J	G
ltag	0	1	0	1	0	0	1	0	1	1

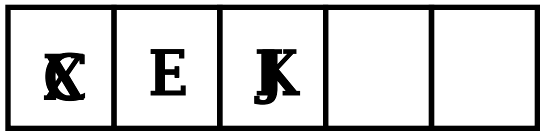
从rtag-ltag先根序列到树

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





stack



从双标记的先根次序恢复树

```
template<class T>
class DualTagTreeNode  {                    // 双标记位先根次序树结点类
public:
    T info;                                 // 结点数据信息
    int ltag, rtag;                         // 左、右标记
    DualTagTreeNode();                      // 构造函数
    virtual ~DualTagTreeNode(); };

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构造左孩子右兄弟表示的树
    using std::stack;                               // 使用STL中的栈
    stack<TreeNode<T>* > aStack;
    TreeNode<T> *pointer = new TreeNode<T>;   // 准备建立根结点
    root = pointer;
```

6.3 树的顺序存储结构

```
for (int i = 0; i < count-1; i++) {                // 处理一个结点
  pointer->setValue(nodeArray[i].info);            // 结点赋值
  if (nodeArray[i].rtag == 0)                      // 若右标记为0则将结点压栈
     aStack.push(pointer);
  else pointer->setSibling(NULL);                  // 右标记为1，则右兄弟指针为空
  TreeNode<T> *temppointer = new TreeNode<T>;      // 预先准备下一个
  if (nodeArray[i].ltag == 0)                      // 左标记为0，则设置孩子结点
     pointer->setChild(temppointer);
  else {                                           // 若左标记为1
     pointer->setChild(NULL);                      // 孩子指针设为空
     pointer = aStack.top();                       // 取栈顶元素
     aStack.pop();
     pointer->setSibling(temppointer); }           // 为栈顶设置一个兄弟结点
  pointer = temppointer; }
pointer->setValue(nodeArray[count-1].info); // 处理最后一个结点
pointer->setChild(NULL);  pointer->setSibling(NULL); }
```

带双标记的层次次序表示法

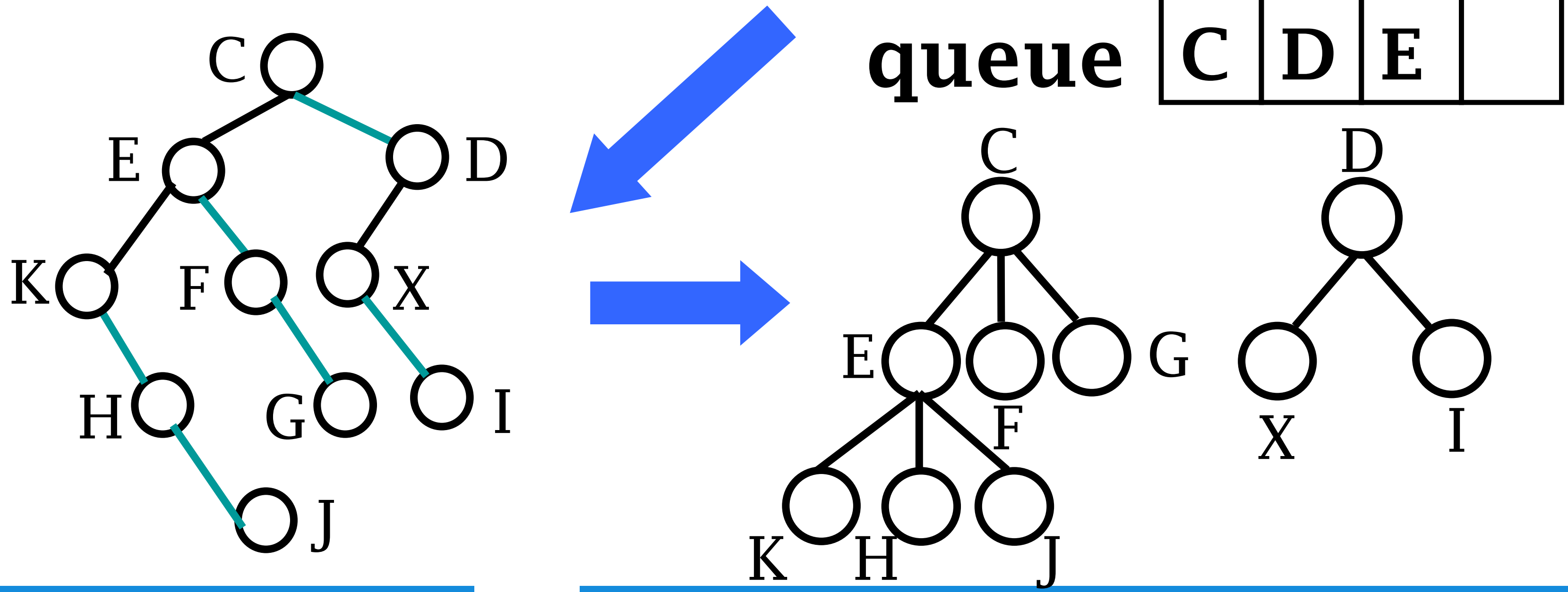
- 结点按 层次次序顺序 存储在连续存储单元

ltag	info	rtag

  - info是结点的数据
  - ltag是一个一位的左标记，当结点没有子节点，即对应的二叉树中结点没有左子结点时， ltag为1，否则为0
  - 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

带双标记的层次次序转换为树

ltag	0	0	0	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



带双标记位的层次次序构造

```
template <class T>
Tree<T>::Tree(DualTagWidthTreeNode<T>* nodeArray, int
count)  {
  using std::queue;                          // 使用STL队列
  queue<TreeNode<T>*> aQueue;
  TreeNode<T>* pointer=new TreeNode<T>; // 建立根
  root=pointer;
  for(int i=0;i<count-1;i++) {              // 处理每个结点
    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>;
```

```
    if(nodeArray[i].rtag == 0)
      pointer->setSibling(temppointer);
    else {
        pointer->setSibling(NULL);     // 右兄弟设为空
        pointer=aQueue.front();        // 取队列首结点指针
        aQueue.pop();                  // 队首元素出队列
        pointer->setChild(temppointer);
    }
    pointer=temppointer;
  }
  pointer->setValue(nodeArray[count-1].info); // 最后一个结点
  pointer->setChild(NULL);  pointer->setSibling(NULL);
}
```

带度数的后根次序表示

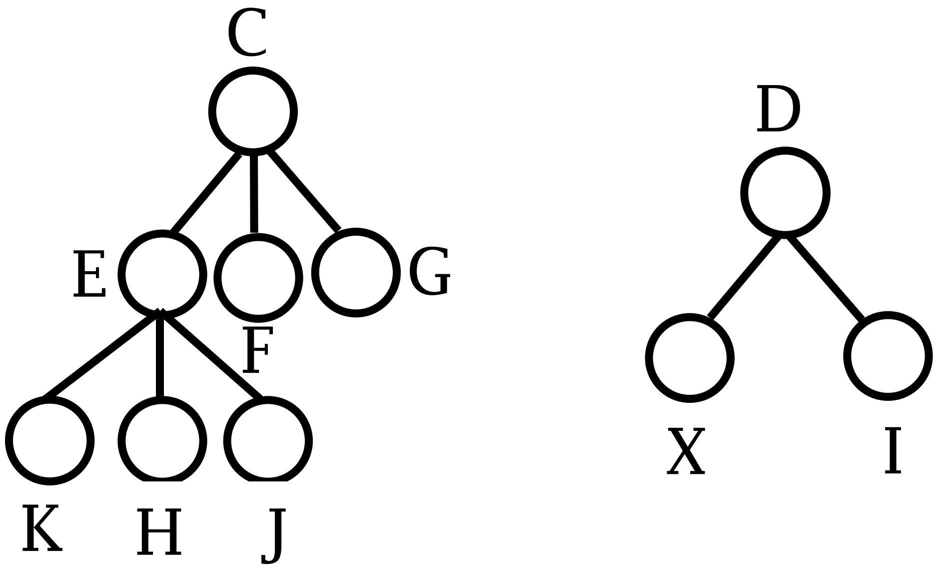
- 在带度数的后根次序表示中，结点按后根次序顺序存储在一片连续的存储单元中，结点的形式为

info	degree

- 其中info是结点的数据，degree是结点的度数

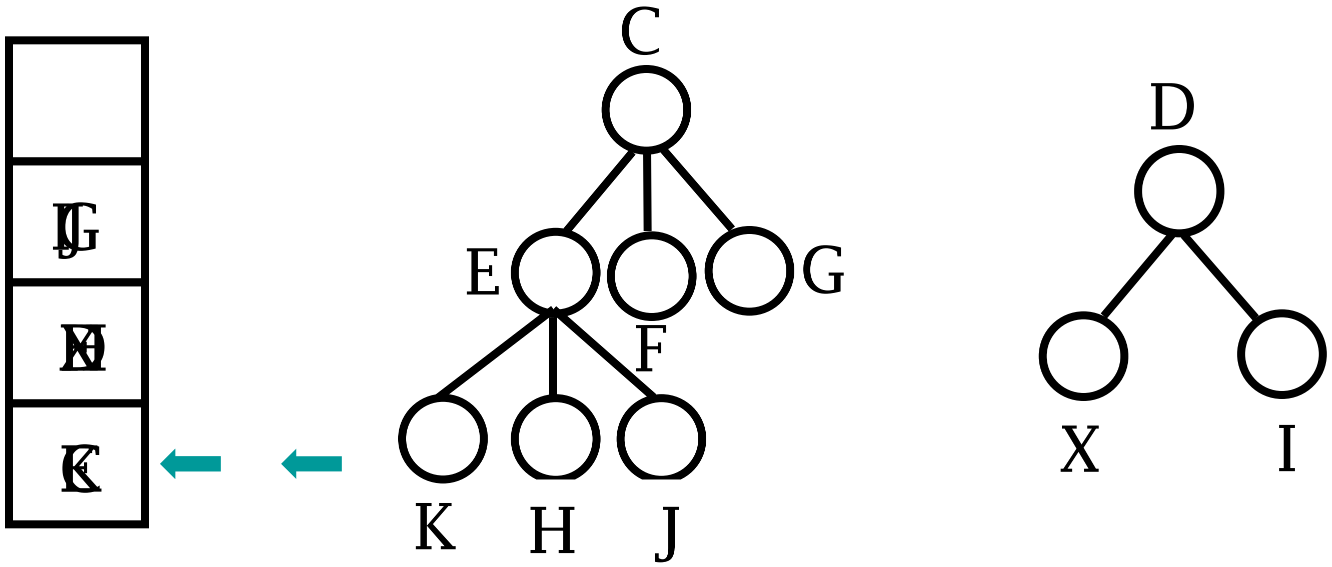
带度数的后根次序表示法

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



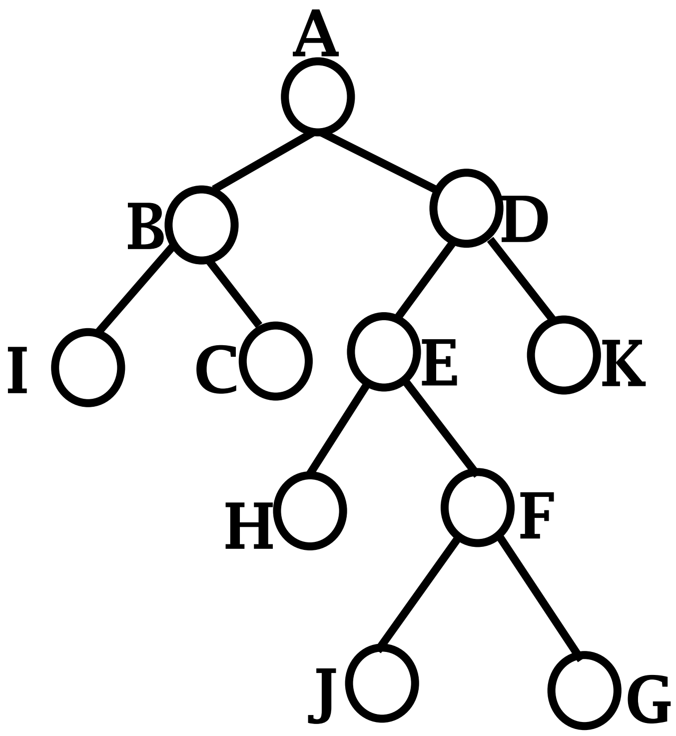
带度数的后根次序变成树

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

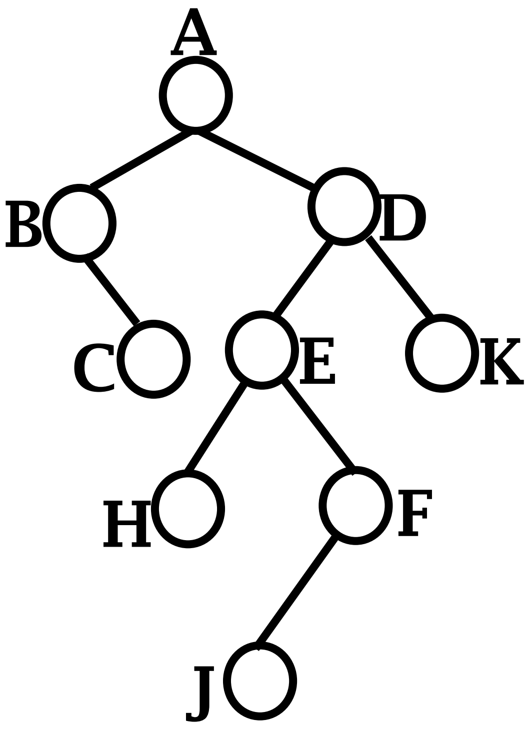


6.3 树的顺序存储结构

- 带标记的满二叉树前序序列
  A’ B’ I C D’ E’ H F’ J G K



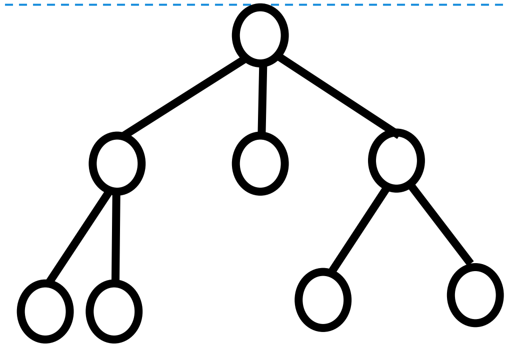
- 带标记的伪满二叉树前序序列
  A’ B’ / C D’ E’ H F’ J / K



思考：森林的顺序存储

- 信息冗余问题
- 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

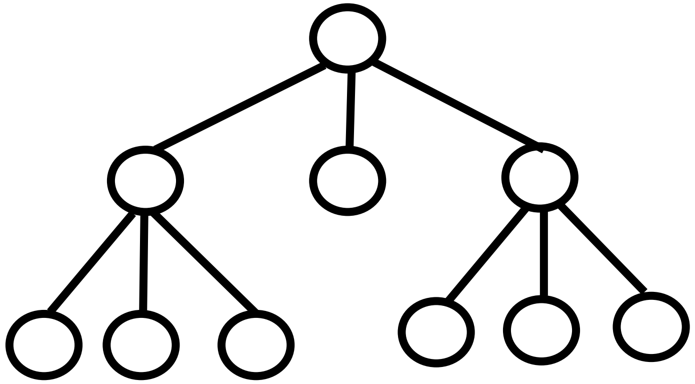
K 叉树定义



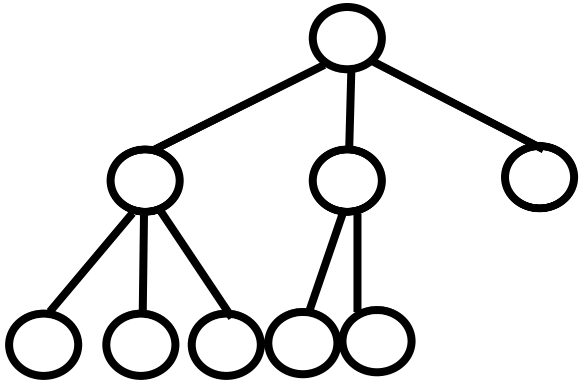
- K 叉树 T 是具有下列性质的有限结点集：
  - (a) 集合可以为空；
  - (b) 非空集合是由一个根结点 root 及 K 棵互不相交的 K 叉树构成。
- 其余结点被划分成 T_0，T_1，…，T_{K-1}（$K \geq 1$）个子集，每个子集都是 K 叉树，使得 $T = \{R, T_0, T_1, \ldots, T_{K-1}\}$。
- K 叉树的各分支结点都有 K 个子结点

满 K 叉树和完全 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3叉树



完全3叉树

数据结构与算法

感谢倾听

国家精品课“数据结构与算法”
http://jpk.pku.edu.cn/course/sjjg/
https://www.icourse163.org/course/PKU-1002534001

张铭，王腾蛟，赵海燕
高等教育出版社，2008. 6。“十二五”国家级规划教材