第6章 树

主要内容

- 6.1 树的定义和基本术语
- 6.2 树的链式存储结构
- 6.3 树的顺序存储结构
- 6.4 K叉树
- 6.5 树知识点总结

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　树的周游**

6.1.1 树和森林

- 树(tree)是包括n个结点的有限集合

  T（n ≥ 1），使得：

  - 有且仅有一个特定的称为根(root)的结点。
  - 除根以外的其它结点被分成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。其中树T_1，T_2，…，T_m称作这个根的子树(subtree)。

6.1.1 树和森林

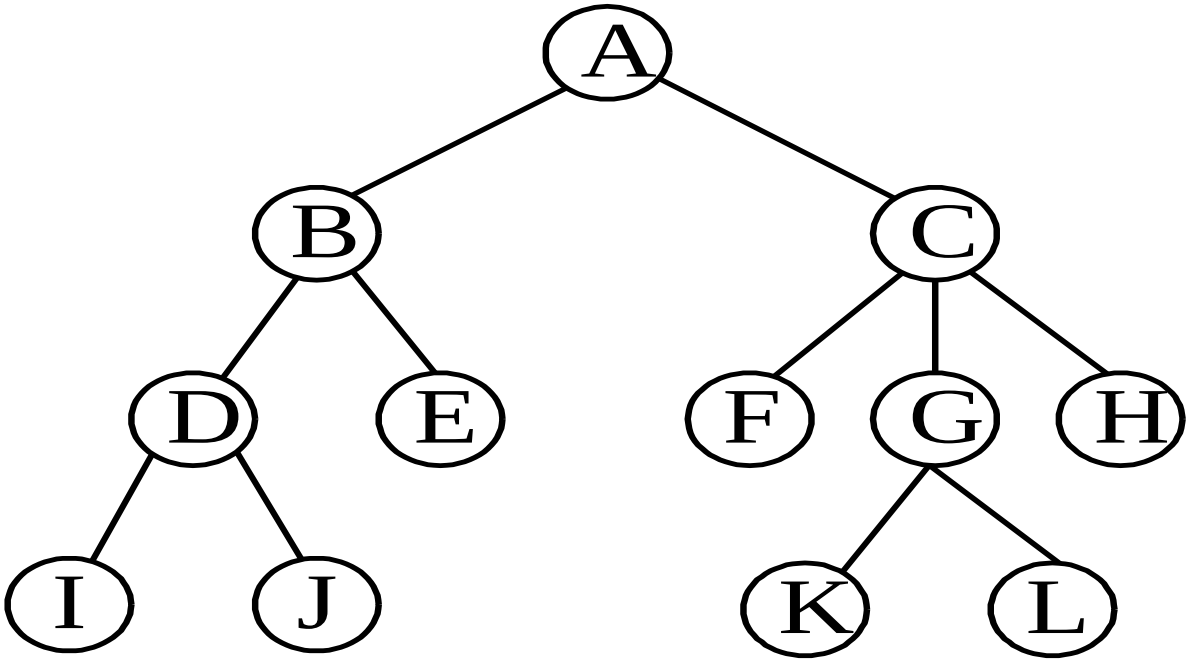


图6.1 树形表示法

6.1.1 树和森林

- 树的逻辑结构可以这样描述：树是包含n个结点的有穷集合K(n > 0)，且在K上定义了一个满足以下条件的二元关系R = {r}：
  - 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。
  - 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<ki-1，ki> $\in R$（$1 \le i \le s$）。这样的结点序列称为从根k_0到结点k的一条路径。

树形结构的各种表示法

树的逻辑结构是：

结点集合K={A，B，C，D，E，F，G，H，I，J}

K上的关系N={<A，B>，<A，C>，<B，D>，

<B，E>，<B，F>，<C，G>，

<C，H>，<E，I>，<E，J>}

树结构中的概念

- 在一棵树中，若存在结点k指向结点k’的连线，则称k是k’的父结点，而k’则是k的子结点，有向连线<k, k’>称作边。
- 同一个父结点的子结点之间互称兄弟。树中没有父结点的结点称为根。没有子结点的结点称为树叶。
- 结点的子树数目称为结点的度，树的度是树中各结点度的最大值，二叉树的度是2。

树结构中的概念

- 若树中存在结点序列k_0，k_1，…，k_s，使得<k_0，k_1>，<k_1，k_2>，…，<k_{s-1}，k_s>都是树中的边，则称从结点k_0到结点k_s存在一条路径。
- 若有一条由 k到达ks的路径，则称k是ks的祖先，ks是k的子孙。
- 结点的层数同样由树根开始定义的，根结点为第0层，非根结点的层数是其父结点的层数加1。

树结构中的概念

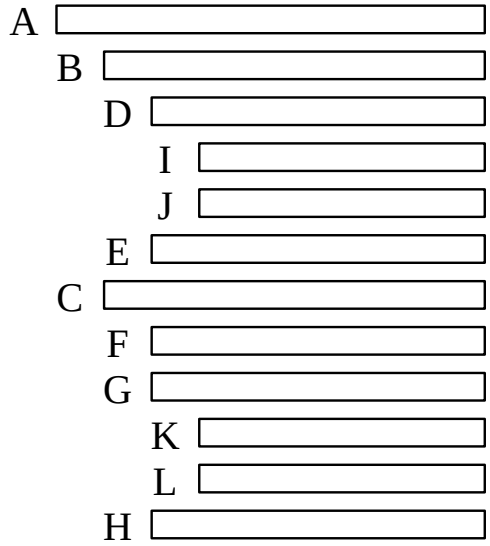
- 有序树：计算机的存储是有序的，为方便计算机处理，往往把子结点按从左到右的次序顺序编号，即把树作为有序树(ordered tree)看待。
- 度为2的有序树并不是二叉树，因为在第一子结点被删除后，第二子结点自然顶替成为第一子结点。因此，度为2并且严格区分左右两个子结点的有序树才是二叉树。

森林与树

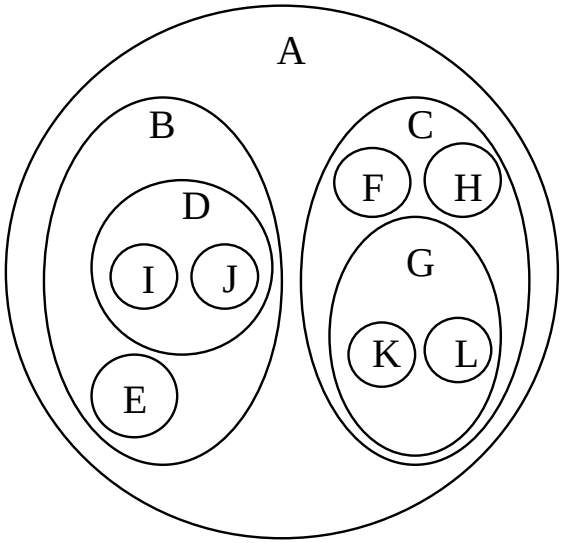
- 森林(forest)是零棵或多棵不相交的树的集合（通常是有序集合）。
- 对于树中的每个结点，其子树组成的集合就是森林；而加入一个结点作为根，森林就可以转化成一棵树了。

树形结构的各种表示法

- 树形结构在客观世界中是大量存在的，有多种逻辑表示方法：如树形表示法、凹入表示法、文氏图表示法、嵌套括号表示法等。



（a）凹入表表示法



（b）文氏图表示法

(A(B(D(I，J)，E)，C(F，G(K，L)，H)))

（c）嵌套括号表示法

图6.2 树形结构的各种表示法

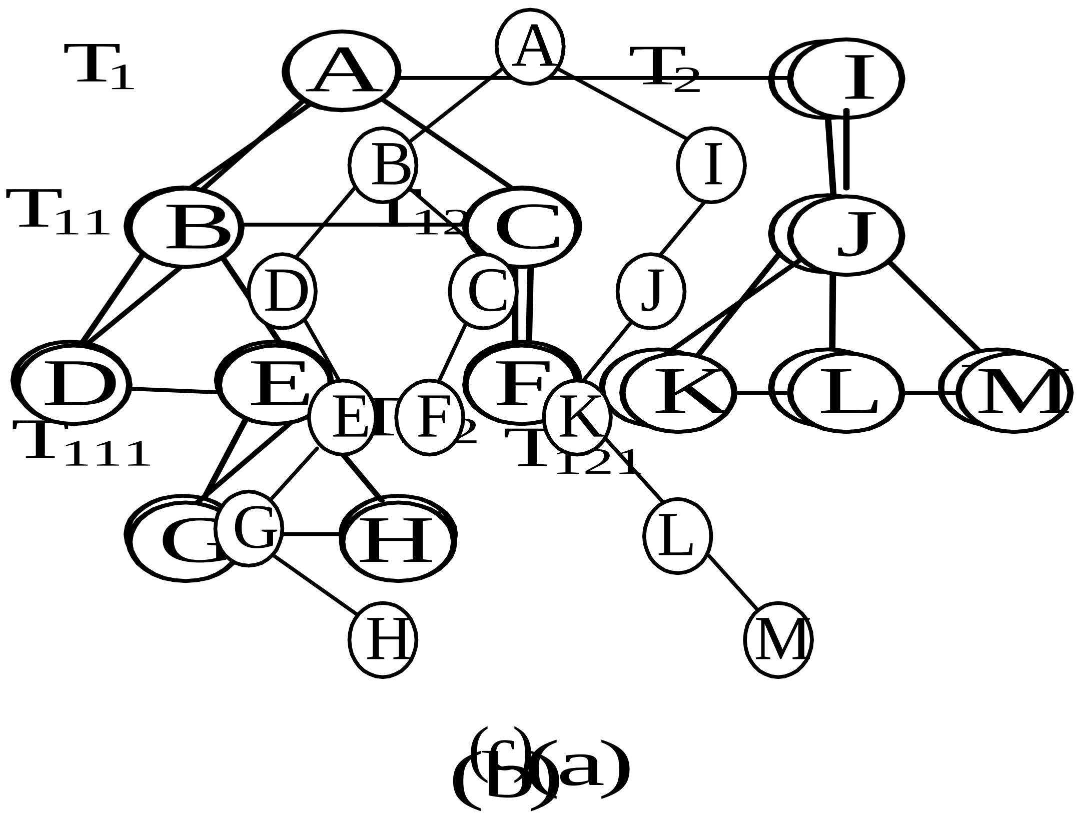
6.1.2 森林与二叉树的等价转换

- 树与二叉树、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。
- 树和森林转化成二叉树后，那么森林或树的相关操作都可以转换成对二叉树的操作。

6.1.2 森林与二叉树的等价转换

- 树和森林到二叉树的转换过程可用连线、切线、旋转“三步曲”来说明：
  - 连线：将兄弟结点用线连接起来。
  - 切线：保留父结点与其第一个子结点的连线，将父结点到其它子结点的连线切掉。
  - 旋转：以根为轴，平面向下顺时针方向旋转一定的角度。旋转只是为了调整画面，使得转化后的二叉树看起来比较规整。

6.1.2 森林与二叉树的等价转换



旋转操作

T1,T2两棵树构成的森林

经过连线、切线处理

图 6.3 森林转换为二叉树

“十一五”国家级规划教材。张铭，王腾蛟，赵海燕，《数据结构与算法》，高教社，2008. 6。

6.1.2 森林与二叉树的等价转换

- 森林或树转化成二叉树的形式定义如下：

设有序集合$F = \{T_1, T_2, \ldots, T_n\}$表示由树$T_1$，$T_2$，…，$T_n$组成的森林，则森林F可以按如下规则递归转换成二叉树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T1的根W1，B(F)的左子树是树T_1中根结点W1的子树森林$F = \{T_{11}, T_{12}, \ldots, T_{1m}\}$转换成的二叉树$B(T_{11}, T_{12}, \ldots, T_{1m})$；B(F)的右子树是从森林$F' = \{T_2, \ldots, T_n\}$转换而成的二叉树。

6.1.2 森林与二叉树的等价转换

- 二叉树转换为树或森林的操作：
  - 旋转：以根为轴，平面逆时针方向旋转。
  - 补线：若结点x是父结点y的左子结点，则把x的右子结点，右子结点的右孩子，依此类推，直到最右孩子，用连线与y连起来。
  - 删线：去掉所有父结点到右孩子的连线。

6.1.2 森林与二叉树的等价转换

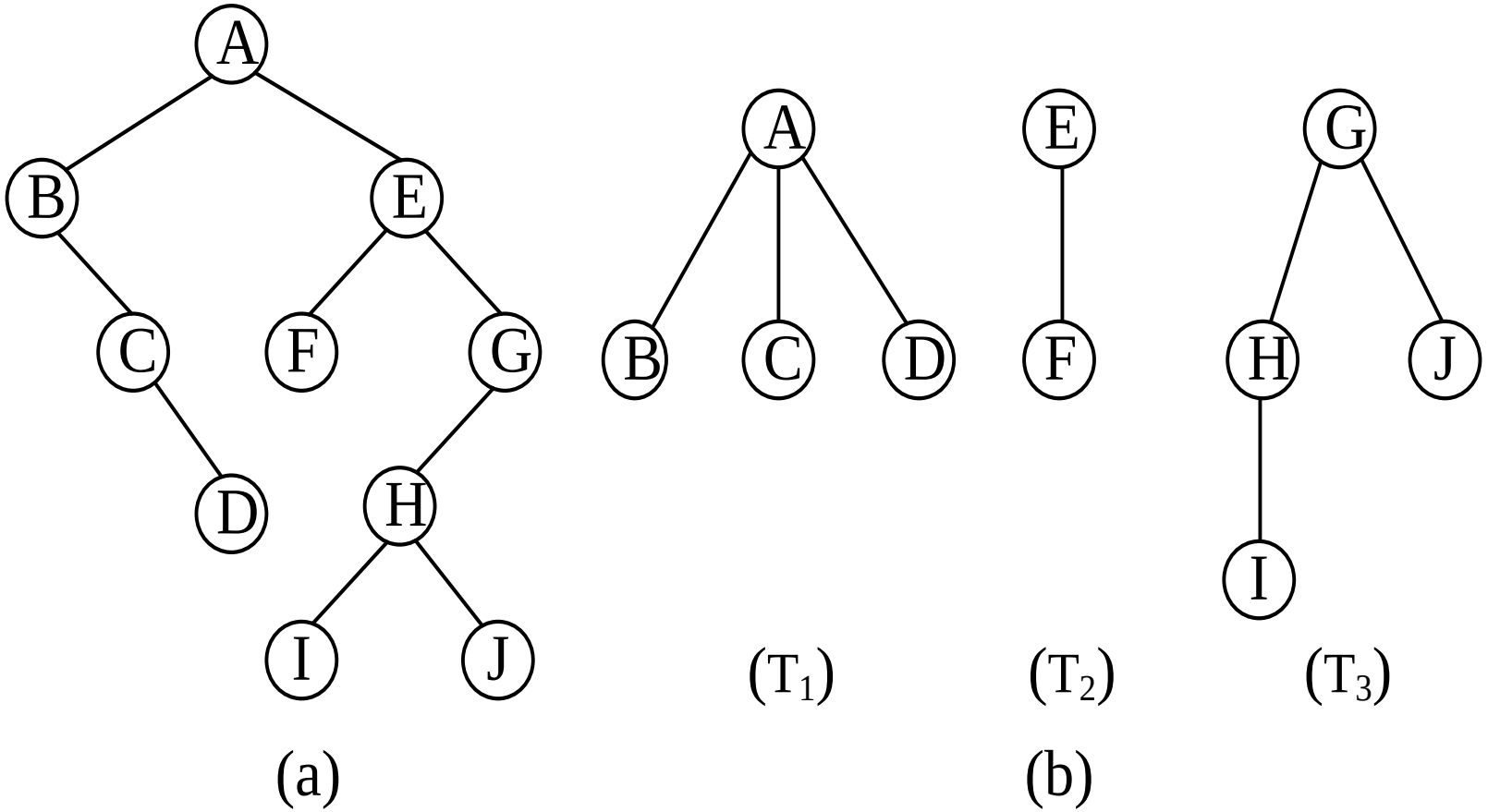


图6.4 二叉树转换为森林

6.1.2 森林与二叉树的等价转换

- 二叉树转化成森林或树的形式定义如下：

  设B是一棵二叉树，root是B的根，BL是root的左子树，BR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T1加上森林F(BR)，其中树T1的根为root，root的子树为F(BL)**

6.1.3 树的抽象数据类型

【代码6.1】树结点的抽象数据类型

```
template<class T>
class TreeNode  {
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
};
```

6.1.3 树的抽象数据类型

【代码6.2】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
```

6.1.3 树的抽象数据类型

```
        TreeNode<T>* Parent(TreeNode<T> *current);
// 返回当前结点的父结点
        TreeNode<T>* PrevSibling(TreeNode<T> *current);
//返回当前结点的前一个兄弟
void DeleteSubTree(TreeNode<T> *subroot);
// 删除以subroot为根的子树
        void RootFirstTraverse(TreeNode<T> *root);
// 先根深度优先周游树
        void RootLastTraverse(TreeNode<T> *root);
// 后根深度优先周游树
        void WidthTraverse(TreeNode<T> *root);
// 广度优先周游树
};
```

6.1.4 树的周游

- 按深度的方向周游
  - 先根次序：

    a)访问头一棵树的根

    b)在先根次序下周游头一棵树树根的子树

    c)在先根次序下周游其他的树
  - 后根次序：

    a)在后根次序下周游头一棵树树根的子树

    b)访问头一棵树的根

    c)在后根次序下周游其他的树

6.1.4 树的周游

- 图6.5 (a)所示的森林，按先根次序周游得到的结点序列是

  A B C E F D G H J I

  - 按先根次序周游森林的序列正好等同于其对应二叉树的前序序列。
- 按后根次序周游得到的结点序列是

  B E F C D A J H I G

  - 后根次序周游得到的排列正好是该森林对应的二叉树在中序次序周游下的结点序列。

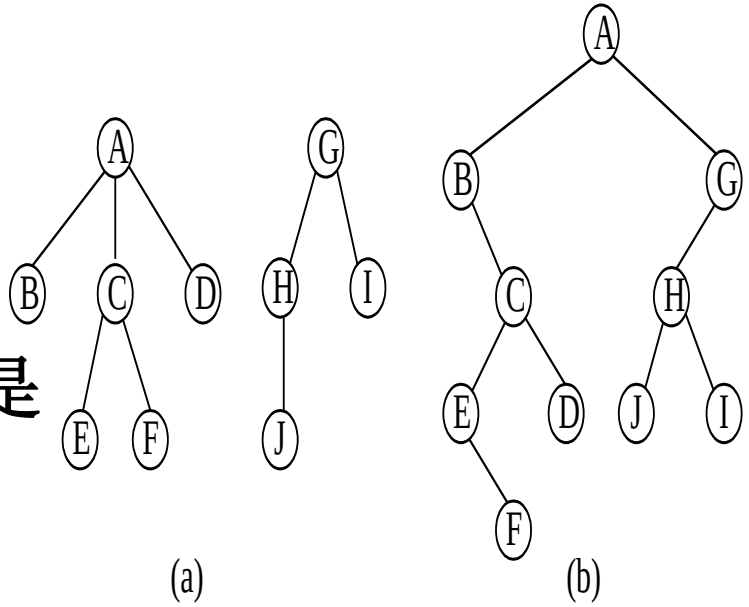


图6.5 森林和对应的二叉树

先根深度优先周游树或森林

【算法6.3】先根深度优先周游树或森林

```
template<class T>
void Tree<T>::RootFirstTraverse(TreeNode<T> * root)  {
while (root != NULL) {
                    Visit(root->Value());
                    // 访问当前结点
                    RootFirstTraverse(root->LeftMostChild());       // 周游第一棵树树根的子树
                    root = root->RightSibling();
          // 周游其它的树
          }
}
```

后根深度优先周游树或森林

【算法6.4】后根深度优先周游树或森林

```
template<class T>
void Tree<T>::RootLastTraverse(TreeNode<T> * root)  {
while (root != NULL) {
                RootLastTraverse(root->LeftMostChild());
        // 周游第一棵树树根的子树
                Visit(root->Value());
                // 访问当前结点
                root = root->RightSibling();
                // 周游其它的树
        }
}
```

广度优先周游森林

广度优先(breadth-first)周游也称“宽度优先周游”，或者“层次周游”：从树的第0层（根结点）开始，自上至下逐层周游；在同一层中，则按照从左到右的顺序对结点逐一访问。

按广度的方向周游图6.5 (a)的森林，得到的结点序列是：

A G B C D H I E F J

按广度优先周游图6.5 (a)中的第一棵树，得到的结点序列是：

A B C D E F

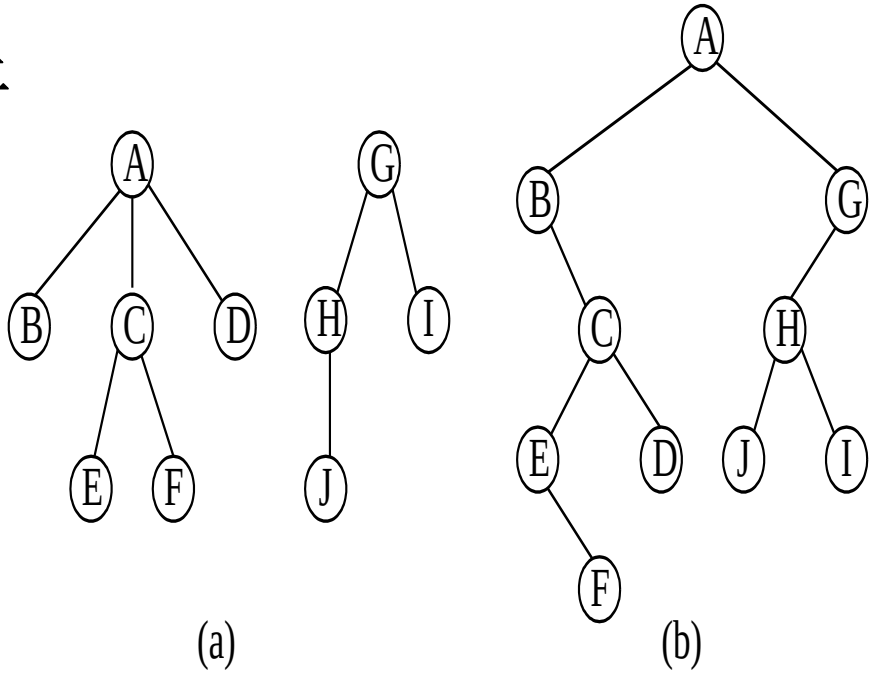


图6.5 森林和对应的二叉树

广度优先周游树或森林

【算法6.5】广度优先周游树或森林

```
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
    向当前结点的右兄弟
    }
```

广度优先周游树或森林

```
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();
      }
   }
}
```

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

6.2.1 “子结点表”表示方法

“子结点表”（list of children）表示方法，就是指每个分支结点的子结点按照从左至右的顺序形成一个链表存储在该分支结点中。其主体是一个存储了树中各结点信息的数组。数组中的每个元素包括三个域，分别用来存放结点信息的值，其父结点指针，以及指向其子结点表的指针。

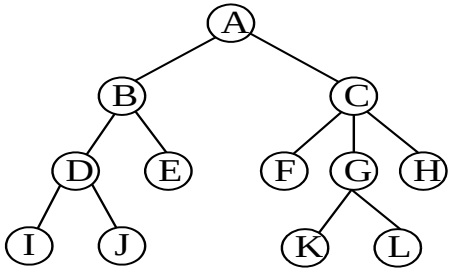


图6.1 树形表示法

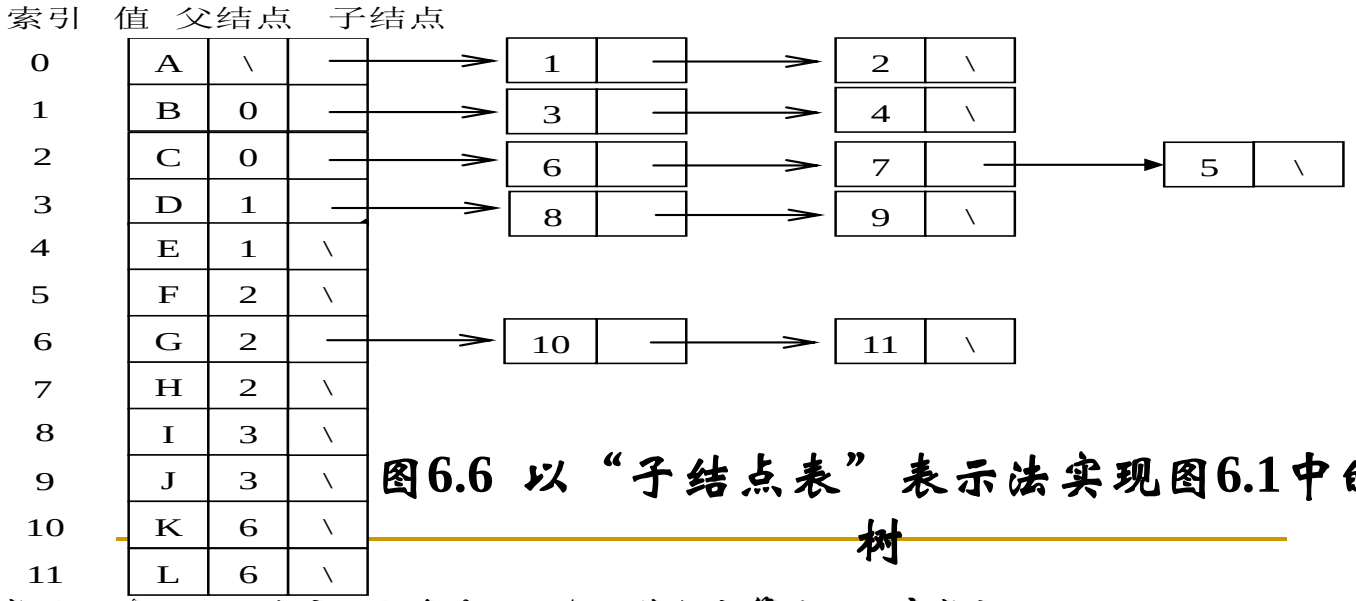


图6.6 以“子结点表”表示法实现图6.1中的树

6.2.2 静态“左孩子/右兄弟”表示法

- 方法仍然使用数组存储树中的各结点
- 每个结点元素包括四个域，分别用于存储结点的值、指向其父结点以及指向最左子结点和右侧兄弟结点的指针
- 这种表示法比“子结点表”表示法的空间效率更高，而且每个结点的存储空间大小固定

6.2.2 静态“左孩子/右兄弟”表示法

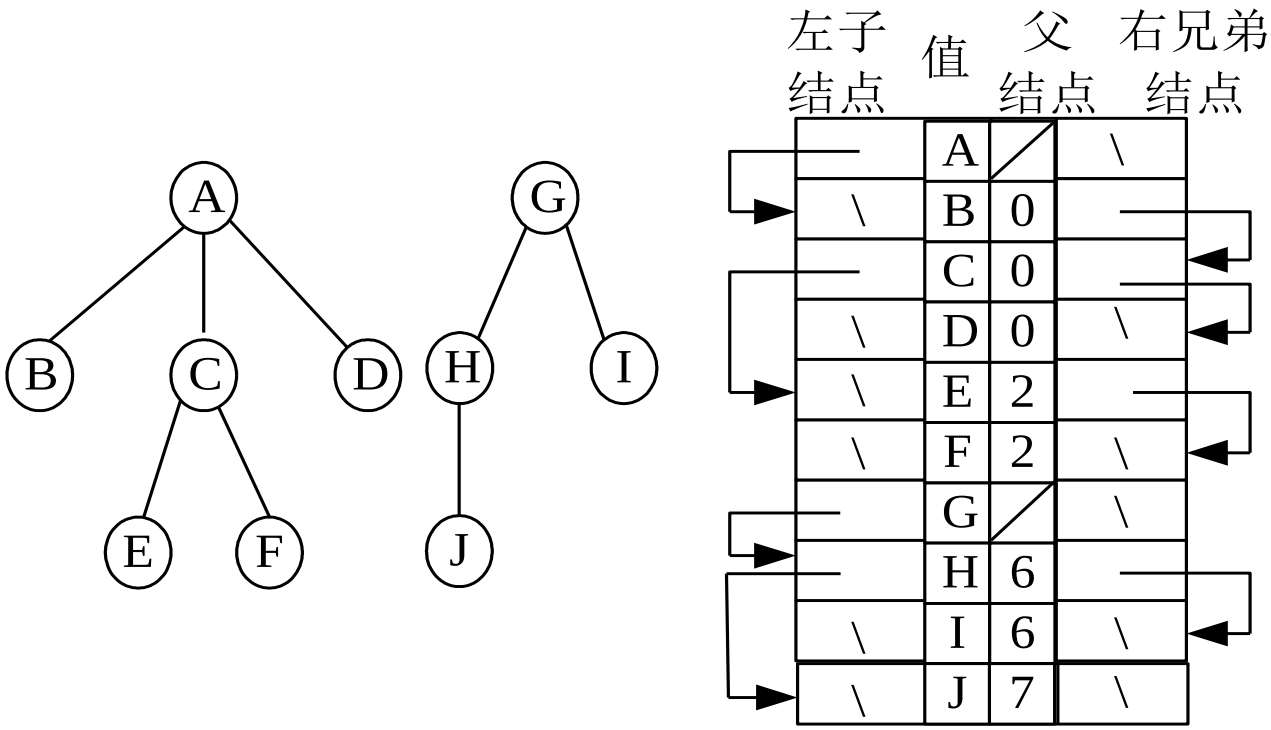
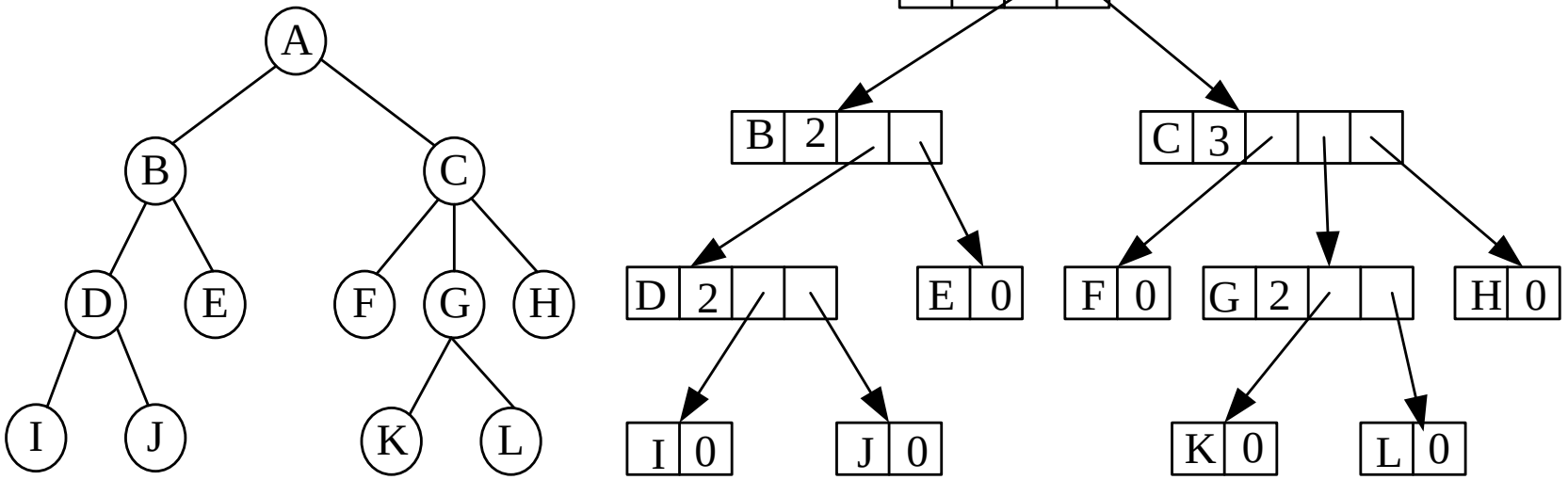


图6.7“左孩子/右兄弟”表示两棵树

6.2.3 动态表示法

- 是为每个结点分配可变的存储空间
  - 每一个结点都存储一个子结点指针表，其子结点的数目也存储在该结点中
  - 即使子结点数目发生变化，那么也只需给该结点重新分配一个大小合适的存储空间就可以了

6.2.3 动态表示法



(a) 树　　(b) 树的实现

6.2.4 动态“左孩子/右兄弟”二叉链表表示法

- 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

6.2.4 动态“左孩子/右兄弟”二叉链表表示法

【代码6.6】树结点的抽象数据类型中的关键实现细节

```
// 用二叉链表实现树要在代码6.1的TreeNode类中增加以下私有数据成员
private:
        T m_Value;                          // 树结点的值
        TreeNode<T> *pChild;                // 左孩子指针
        TreeNode<T> *pSibling;              // 右兄弟指针
// 部分成员函数的具体实现
template<class T>
bool TreeNode<T>::isLeaf()  {
        // 判断当前结点是否为叶结点
        if (pChild == NULL)
                // 如果结点是叶结点返回true
                return true;
        return false;
}
```

6.2.4 动态“左孩子/右兄弟”二叉链表表示法

```
template<class T>
void TreeNode<T>::setValue(const T& value) {        // 设置结点的值
        m_Value = value;
}
template<class T>
void TreeNode<T>::InsertFirst(TreeNode<T>* node) {
    // 以第一个孩子的身份插入结点
        if (pChild == NULL)                         // 如果没有子结点
                pChild = node;                      // 将node置为子结点
        else {
                                // 如果有子结点
                node->setSibling(pChild);
        // 将原来子结点置为node的右兄弟
                pChild = node;
                // 将node置为子结点
        }
}
```

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
- 用数组存储树的所有结点，同时在每个结点中附设一个指针指示其父结点的位置。

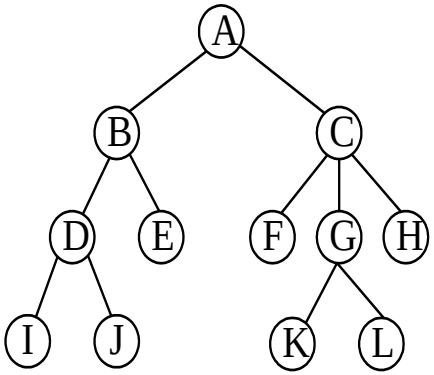


图6.1 树形表示法

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

图6.10 父指针表示法

6.2.5 父指针表示法及在并查集中的应用

- 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地表示任何一棵树。
- 在这种表示方法下，寻找一个结点的父结点只需要O(1)时间。
- 在树中可以从一个结点出发找出一条向上延伸到达其祖先的路径，即从一个结点到其父亲结点，再到其祖父结点等等。
- 求祖先路径所需要的时间正比于路径上的结点个数，因此父指针表示法对于求树根结点的运算非常方便。

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**：判断两个结点是否在同一个集合中
- **Union**：归并两个集合
- 像栈、队列一样，并查集也是一种重要的抽象数据类型，可以用于求解等价类问题

等价类(1)

等价关系和等价类的概念：

假定一个具有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）

等价类(2)

- 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，用数学语言表示为[x]R = {y| y∈S ∧ xRy}。R可将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(3)

- 假定集合S有n个元素，k个形如(x，y)的等价偶对就确定了一个等价关系R。那么应该如何划分这个等价类呢？
- 下面给出划分等价类的过程：
  - 初始情况S中的每个元素都属于一个独立的等价类S1，S2，…，Sn。
  - 依次读入k个偶对，对每个读入的偶对(x，y)，判定x和y所属的子集。假设x∈Si，y∈Sj，如果Si≠Sj，则将集合Si置入集合Sj并将集合Si置为空（或者将Sj置入Si并将Sj置为空）。
  - 当k个偶对都被处理完后，S1，S2，…，Sn中所有的非空子集就是S的R等价类。

等价类(4)

划分等价类需要对集合进行三种操作：

- 构造只含有一个元素的集合
- 判定某个元素所在的子集，目的是为了确定两个元素是否在同一个集合之中。即搜索包含该元素的等价类，对于同一个集合中的元素返回相同的结果，否则返回不同的结果
- 归并两个不相交的集合为一个集合

等价类(5)

用父指针表示的树形结构实现的并查集可以很容易地解决等价类问题:

- 约定森林F={T1，T2，…，Tr}表示集合S
- 森林中的每一棵树Ti表示集合S的一个子集
- 树中的结点表示集合S中的一个元素
- 树中的每一个非根结点都指向其父结点，用根结点作为集合的标识符

等价类(6)

- 由于用根结点表示子集的类别，那么“查找”某一个元素所属的集合，只要从该结点出发，沿父链域找到树的根结点即可。
- 实现集合的“并”操作只要将一棵子树的根指向另一棵子树的根即可

示例

- 图两棵树分别表示子集S1 = {1，3，5，7}和S2 = {2，4，6，8}
- 图6.11中的(c)就实现了S3 = S1∪S2

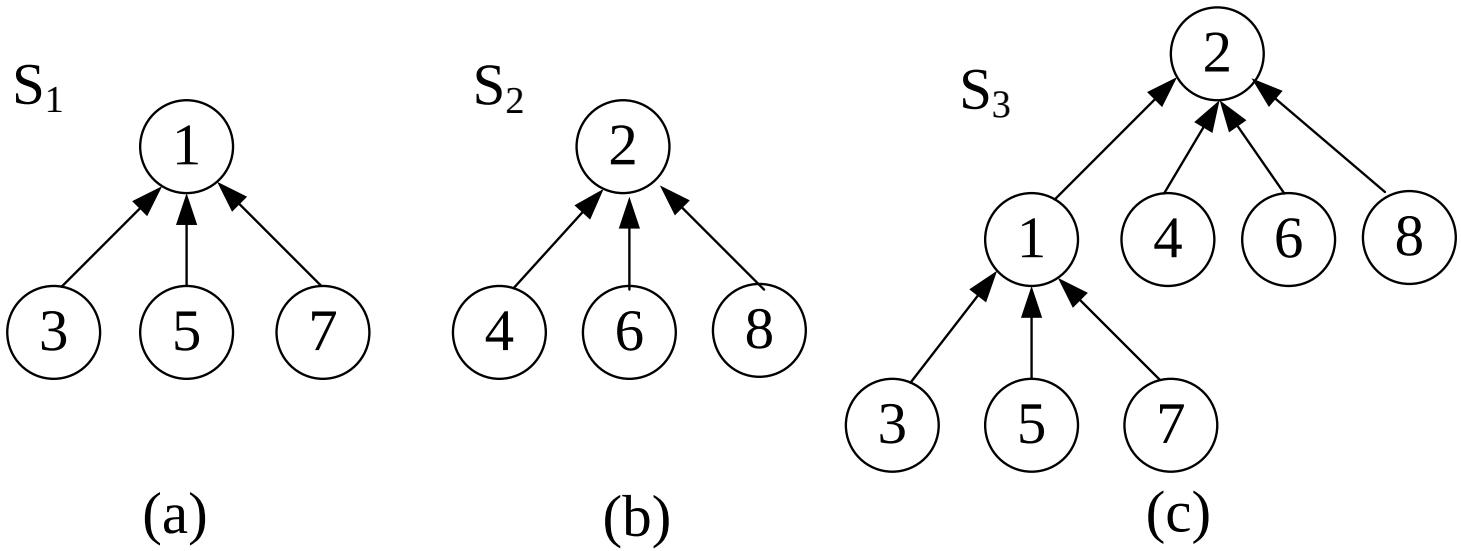


图6.11集合的表示方法

重量权衡合并规则

- 每次合并前都需要进行两次查找，查找所需要的时间由树的高度决定，合并所需的时间为**O(1)**
- 容易看出，在最坏情况下合并可能使**n**个结点的树退化成一条链
- 为了防止树退化为单链，应该让每个结点到其相应根结点的距离尽可能小，可以做如下两种改进
- 第一种改进的方法是在做“合并”操作之前先判别子集中所含成员的数目，然后令含成员少的子集的树根指向含成员多的子集的根，称作“重量权衡合并规则”（**weighted union rule**）。把小树合并到大树中去，可以把树的整体深度限制在O**(logn)**，每次**Find**操作只需要**O(logn)**时间

树的父指针表示与Union/Find算法实现

【代码6.8】 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;//以此结点为根的子树的总结点个数
public:
    ParTreeNode();                          //构造函数
    virtual ~ParTreeNode(){};    //析构函数
```

树的父指针表示与Union/Find算法实现

```
    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);
};
```

树的父指针表示与Union/Find算法实现

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

树的父指针表示与Union/Find算法实现

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

树的父指针表示与Union/Find算法实现

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

树的父指针表示与Union/Find算法实现

```
template<class T>
class ParTree
{   //树定义
public:
    ParTreeNode<T>* array;//存储树结点的数组
    int   Size;                              //数组大小
    //查找node结点的根结点
    ParTreeNode<T>*
    Find(ParTreeNode<T>* node) const;
```

树的父指针表示与Union/Find算法实现

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

树的父指针表示与Union/Find算法实现

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

树的父指针表示与Union/Find算法实现

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

树的父指针表示与Union/Find算法实现

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

树的父指针表示与Union/Find算法实现

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

树的父指针表示与Union/Find算法实现

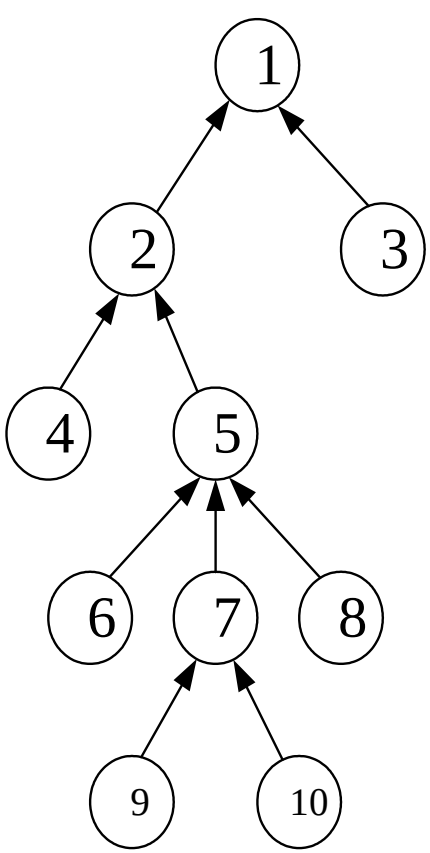
```
            pointeri->setCount(pointeri->
                        getCount()+pointerj->getCount());
            }
            else
            {
                        pointeri->setParent(pointerj);
                        pointerj->setCount(pointeri->
                                    getCount()+pointerj->getCount());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

路径压缩

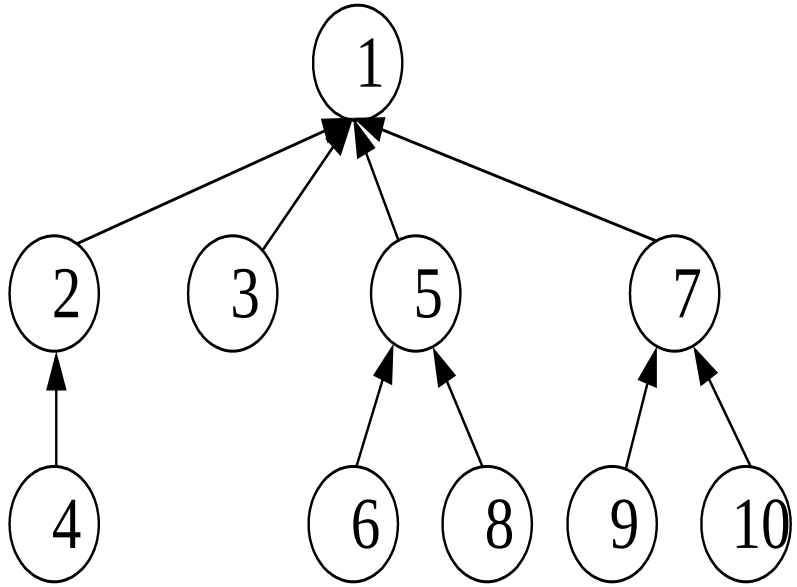
- 路径压缩加速Find运算
- 在执行Union时总是将小树并到大树上，而且

  在执行Find时实行路径压缩

路径压缩示例



(a) 路径压缩之前



(b) 路径压缩之后

图6.13

路径压缩示例

带路径压缩的FindPC算法

提供了一种实现路径压缩的递归算法。函数Find不仅返回当前结点的根结点，而且把当前结点所有祖先结点的父指针都指向根结点

【算法6.9】带路径压缩的FindPC算法

```
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();
}
```

The End