第8章　二叉树和其他树

在一片丛林中有各种各样的树、植物和动物。在数据结构的世界中也有许多“树”，不过本书不可能全部介绍。在本章中将学习两种基本的树：一般树 (简单树)和二叉树。第9、10和11章中对其他树有更详细的介绍。

在本章的应用部分给出了树的两个应用。第一个应用是关于在一个树形分布的网络中设置信号调节器。第二个应用是3.8.3节中所介绍的在线等价类问题。在线等价类问题在本章中又被称为合并/搜索问题。利用树来解决等价类问题要比3.8.3节中的链表解决方案高效得多。

另外，本章中还覆盖了以下内容：

- 树和二叉树的术语，如高度、深度、层、根、叶子、子节点、父节点和兄弟节点。
- 二叉树的公式化描述和链表描述。
- 4种常用的二叉树遍历方法：前序遍历，中序遍历，后序遍历和按层遍历。

8.1　树

到目前为止，我们已经介绍了线性数据结构和表数据结构。这些数据结构一般不适合于描述具有层次结构的数据。在层次化的数据之间可能有祖先-后代、上级-下属、整体-部分以及其他类似的关系。

例8-1　[Joe的后代] 图8-1给出了Joe的后代，并按层次方式组织，其中Joe在最顶层。Joe的孩子（Ann，Mary和John）列在下一层，在父母和孩子间有一条边。在层次表示中，非常容易地找到Ann的兄弟姐妹，Joe的后代，Chris的祖先等。

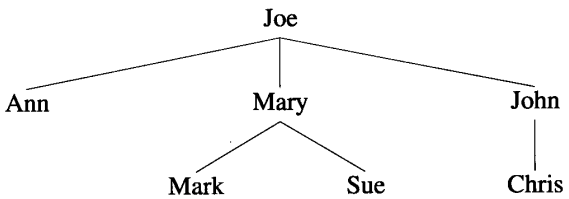


图8-1　Joe的后代

例8-2　[合作机构] 作为层次数据的一个例子，考虑图8-2的合作管理机构。在层次中地位最高的人（此处为总裁）在图中位置最高。在层次中地位次之的（即副总裁）在图中位于总裁之下等等。副总裁为总裁的下属，总裁是他们的上级。每个副总裁都有他自己的下属，而其下属又

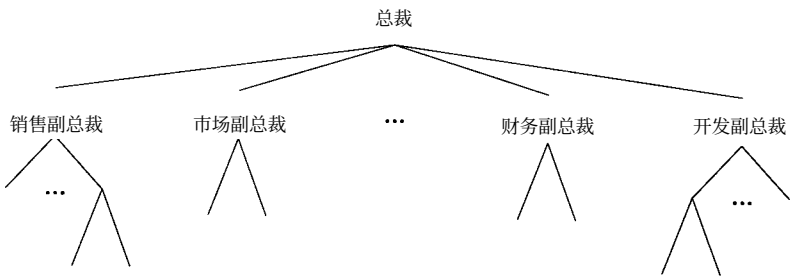


图8-2　合作管理结构

有他们自己的下属。在图中，在每个人与其直接下属或上级之间都有一条边互连。

例8-3 [政府机构] 图8-3是联邦政府各分支机构的层次图。在最顶层是整个联邦政府。层次结构的下一级，是其主要的隶属单位(例如不同的部)。每个部可进一步细分。这些分支在层次结构的下一级画出。例如，国防部分成陆军、海军、空军和海军陆战队。在每个机构及其分支机构间都有一条边。图8-3的数据即为整体-部分关系的例子。

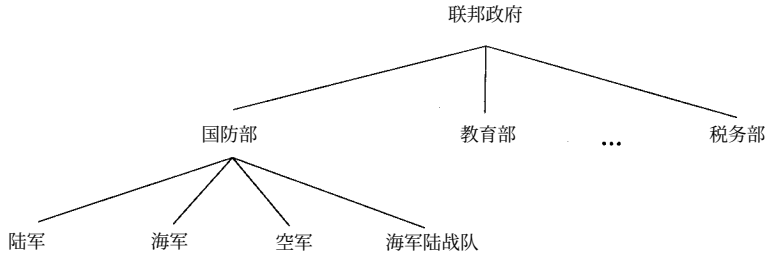


图8-3 联邦政府模型

例8-4 [软件工程] 考察另一种层次数据——软件工程中的模块化技术。通过模块化，可以把大的复杂的任务分成一组小的不太复杂的任务。模块化的目标是把软件系统分成许多功能不相关的部分或模块以便于进行相对独立的开发。由于解决几个小问题比解决大问题更容易一些，因此模块化方法可以缩短整个软件的开发时间。另外，不同的程序员可以同时开发不同的模块。如果有必要，每个模块可以再细分，从而得到如图 8-4所示的用树形表示的模块层次结构。该树给出了某文字处理器的一种可行的模块分解图。

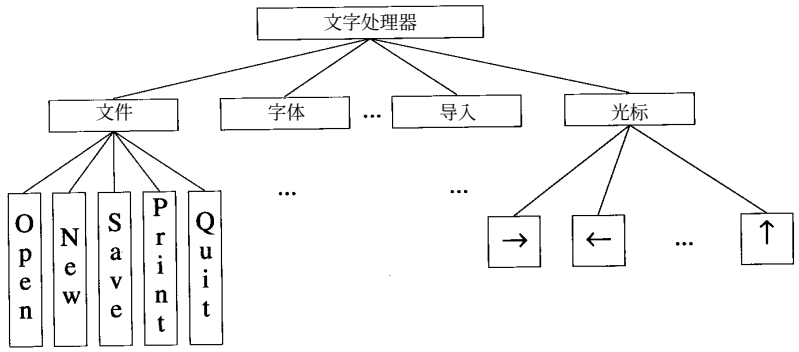


图8-4 文字处理器的模块层次结构

文字处理器的最顶层模块被划分为下一层的几个模块，在图 8-4中只给出4个。文件模块完成与文本文件有关的操作，如打开一个已存在文件（Open），打开一个新文件（New），保存文件（Save），打印文件（Print），从文字处理器中退出（Quit）。在层次结构中下一层的每一个模块分别代表一个函数。字体模块处理与字体有关的所有功能。这些功能包括改变字体、大小、颜色等。若把具有这些功能的模块在图中画出的话，那么它们一定出现在字体模块之下。导入模块用于处理图形、表格以及其他格式的文本文件。光标模块处理屏幕上光标的移动。在接口完全设计好后，程序员可以以相对独立的方式分析、设计和开发每个模块。

当一个软件系统以模块化方式划分好之后，可以很自然地以模块为单位来开发该系统。在最终完成的软件系统中，所含有的模块数与模块层次结构中节点数一样多。模块化可以促进对

欲解决问题的智能化管理。通过把一个大问题系统地分解成小而相对独立的小问题，可以使大问题的解决更省力。可以将独立的问题分配给不同的人同时解决。而在一个单一模块上进行分工是非常困难的。开发模块化软件的另一好处是，分开测试一些小而独立的模块比测试一个大的模块要容易得多。层次结构清晰地给出了模块间的关系。

定义 **[树]** 树（tree）t 是一个非空的有限元素的集合，其中一个元素为根（root），余下的元素（如果有的话）组成t 的子树（subtree）。

现在看一下定义与层次数据例子之间的关系。层次中最高层的元素为根。其下一级的元素是余下元素所构成的子树的根。

例8-5 在Joe的后代例子中(例8-1)，数据集合是{Joe，Ann，Mary，Mark，Sue，John，Chris}，因此n=7，树的根为Joe。余下的元素被分成三个不相交的集合 {Ann}，{Mary，Mark，Sue}和{ John，Chris }。{Ann}是只有一个元素的树，其根为Ann。{ Mary，Mark，Sue } 的根为Mary，而{ John，Chris }的根为John。集合{ Mary，Mark，Sue}余下的元素分成不相交的集合{ Mark }和{ Sue }，二者均为单元素的子树，集合{ John，Chris } 余下的元素也为单元素子树。

在画一棵树时，每个元素都代表一个节点。树根在上面，其子树画在下面。在树根与其子树的根（如果有子树）之间有一条边。同样的，每一棵子树也是根在上，其子树在下。在一棵树中，边连结一个元素及其子节点。在图 8-1中，Ann，Mary，John是Joe的孩子（children），Joe是他们的父母（parent）。有相同父母的孩子为兄弟（sibling）。Ann，Mary，John在图8-1的树中为兄弟，而Mark和Chris则不是。此外还有其他术语：孙子（grandchild），祖父（grandparent），祖先（ancestor），后代（descendent）等。树中没有孩子的元素称为叶子（leaf）。因此在图8-1中，Ann，Mark，Sue 和Chris 是树的叶子。树根是树中唯一一个没有父节点的元素。

例8-6 在合作机构的例子中（例8-2），公司雇员是树中的元素。总裁是树的根，余下的分成不相交的集合，代表公司的不同分支，每个分支有一个副总裁，为该分支子树的根。分支中余下的元素分成不相交的集合，代表不同的部。部长是子树的根。余下元素同样可分成不同的科等。

副总裁是总裁的子节点，部长是副总裁的子节点。总裁是副总裁的父节点，每个副总裁是其分支中元素的父节点。

在图8-3中，根为联邦政府。其子树的根为国防部，教育部，…，税务部等。联邦政府是其子节点的父节点。国防部的子节点为陆军、海军、空军、海军陆战队。国防部的子节点之间为兄弟关系，同时它们也是叶节点。

树的另一常用术语为级（level）。指定树根的级为1，其孩子（如果有）的级为2，孩子的孩子为3，等等。在图8-3中，联邦政府在第1级，国防部、教育部、税务部在第2级，陆军、海军、空军和海军陆战队在第3级。

元素的度（degree of an element）是指其孩子的个数。叶节点的度为0，在图8-4中文件模块的度为5。树的度（degree of a tree）是其元素度的最大值。

练习

1. 给出本书中主要元素的树形表示(整本书、章、节、小节)。

1) 树中共有多少个元素?

2) 标出叶节点。

3) 标出第3级元素。

4) 给出每个元素的度。

2. 访问http://www.cise.ufl.edu，即佛罗里达大学计算机、信息科学和工程系主页。通过连接到更下一级的网页，绘出网页间的层次结构。用节点表示网页，用线连接网页。

1) 此结构一定是一棵树吗? 为什么？

2) 若此结构是一棵树，指出其根和叶子。

8.2 二叉树

定义 [**二叉树**] 二叉树（binary tree）t 是有限个元素的集合（可以为空）。当二叉树非空时，其中有一个称为根的元素，余下的元素（如果有的话）被组成 2个二叉树，分别称为t的左子树和右子树。

二叉树和树的根本区别是：

• 二叉树可以为空，但树不能为空。

• 二叉树中每个元素都恰好有两棵子树（其中一个或两个可能为空）。而树中每个元素可有若干子树。

• 在二叉树中每个元素的子树都是有序的，也就是说，可以用左、右子树来区别。而树的子树间是无序的。

像树一样，二叉树也是根节点在顶部。二叉树左 (右)子树中的元素画在根的左 (右)下方。在每个元素和其子节点间有一条边。

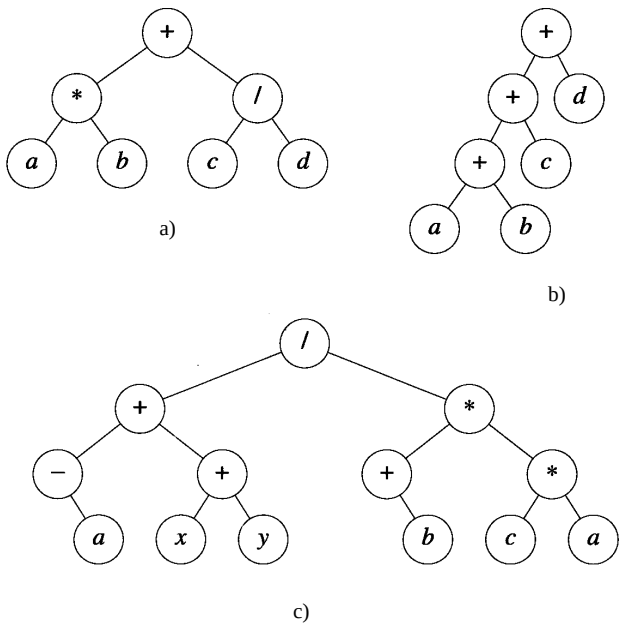


图8-5 数学表达式树

a) $(a * b) + (c / d)$ b) $((a + b) + c) + d)$ c) $((-a) + (x + y)) / ((+b) * (c * a))$

图8-5给出了表示数学表达式的二叉树。每个操作符（十，一，*，/）可以有一个或两个操作数。左操作数是操作符的左子树，右操作数是操作符的右子树。树中的叶节点为常量或变量。注意在数学表达式树中没有括号。

数学表达式二叉树的一个应用是产生一个表达式的优化代码。我们并不研究怎样从数学表达式树中产生最优代码的算法，只是用它来说明许多操作可以用二叉树来解决。

练习

3. 1) 标出图8-5中二叉树的叶子。
2) 标出图8-5b 中第3级的所有节点。
3) 在图8-5c 中第4级有多少个节点？
4. 给出如下各表达式的二叉树：
1) (*a* + *b*)/ (*c* - *d* * *e*) + *e* + *g* * *h* /*a*。
2) -*x*-*y* * *z* + (*a* + *b* + *c* / *d* * *e*)。
3) ((*a* + *b*) > (*c* - *e*)) | | *a* < *f* && (*x* < *y* | | *y* > *z*)。

8.3 二叉树的特性

特性1 包含*n* (*n*>0)个元素的二叉树边数为*n* - 1。
证明 二叉树中每个元素 (除了根节点) 有且只有一个父节点。在子节点与父节点间有且只有一条边，因此边数为*n* - 1。

二叉树的高度（height）或深度（depth）是指该二叉树的层数。图8-5a 中二叉树的高度为3，而图8-5b 和c 的高度为4。

特性2 若二叉树的高度为*h*，*h*≥0，则该二叉树最少有*h*个元素，最多有2^h-1个元素。
证明 因为每一层最少要有1个元素，因此元素数最少为*h*。每元素最多有2个子节点，则第*i* 层节点元素最多为2^i-1个，*i*>0。*h*=0时，元素的总数为0，也就是2^0-1。当*h*>0时，元素的总数不会超过 $\sum_{i=1}^{h}2^{i-1}=2^h-1$。

特性3 包含*n* 个元素的二叉树的高度最大为*n*，最小为⌈$\log_2(n+1)$⌉。
证明 因为每层至少有一个元素，因此高度不会超过*n*。由特性2，可以得知高度为*h* 的二叉树最多有2^h - 1个元素。因为*n*≤2^h - 1，因此*h*≥$\log_2(n+1)$。由于*h* 是整数，所以*h*≥⌈$\log_2(n+1)$⌉。

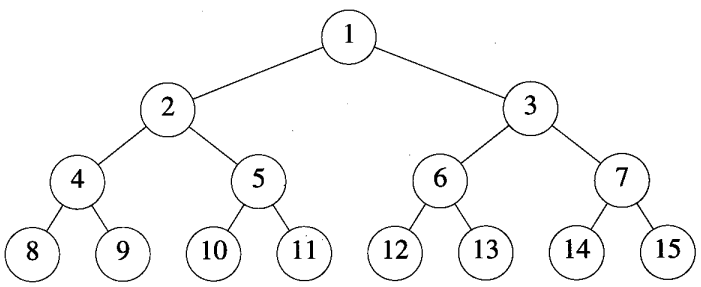


图8-6 高度为4的满二叉树

当高度为h的二叉树恰好有2^h-1个元素时，称其为满二叉树（full binary tree）。图8-5a是一个高度为3的满二叉树。图8-5b和c的二叉树不是满二叉树。图8-6给出高度为4的满二叉树。

假设对高度为h的满二叉树中的元素按从第上到下，从左到右的顺序从1到2^h-1进行编号（如图8-6所示）。假设从满二叉树中删除k个元素，其编号为2^h-i, $1\le i\le k$，所得到的二叉树被称为完全二叉树（complete binary tree）。图8-7给出三棵完全二叉树。注意满二叉树是完全二叉树的一个特例，并且，注意有n个元素的完全二叉树的深度为$\lceil \log_2(n+1) \rceil$。

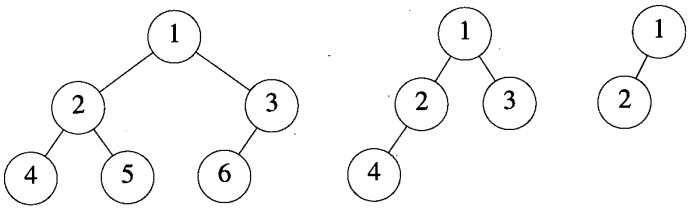


图8-7 完全二叉树

在完全二叉树中，一个元素与其孩子的编号有非常好的对应关系。其关系在特性4中给出。

特性4 设完全二叉树中一元素的序号为i, $1\le i\le n$。则有以下关系成立：

1) 当$i=1$时，该元素为二叉树的根。若$i>1$，则该元素父节点的编号为$\lfloor i/2 \rfloor$。

2) 当$2i>n$时，该元素无左孩子。否则，其左孩子的编号为$2i$。

3) 若$2i+1>n$，该元素无右孩子。否则，其右孩子编号为$2i+1$。

证明 通过对i进行归纳即可得证。

练习

5. 证明特性4。

6. 在k叉树中，每个节点最多有k个孩子。其子节点分别称为该节点的第一个，第二个…，第k个孩子。

1) 模拟特性1给出k叉树的性质。

2) 模拟特性2给出k叉树的性质。

3) 模拟特性3给出k叉树的性质。

4) 模拟特性4给出k叉树的性质。

7. 有m个叶子的二叉树最多有多少个节点？

8.4 二叉树描述

8.4.1 公式化描述

二叉树的公式化描述利用了特性4。二叉树可以作为缺少了部分元素的完全二叉树。图8-8给出二叉树的两个样例。第一棵二叉树有三个元素(A、B和C)，第二棵二叉树有五个元素(A、B、C、D和E)。没有涂阴影的圈表示缺少的元素。所有的元素(包括缺少的元素)按前面介绍的方法编号。

在公式化描述方法中，按照二叉树对元素的编号方法，将二叉树的元素存储在数组中。图

8-8同时给出了二叉树的公式化描述。缺少的元素由白圈和方格描述。当缺少很多元素时，这种描述方法非常浪费空间。实际上，一个有 n 个元素的二叉树可能最多需要 2^n-1 个空间来存储。当每个节点都是其他节点的右孩子时，存储空间达到最大。图 8-9给出这种情况下一棵有四个元素的二叉树，这种类型的二叉树称为右斜(right-skewed)二叉树。当缺少的元素数目比较少时，这种描述方法很有效。

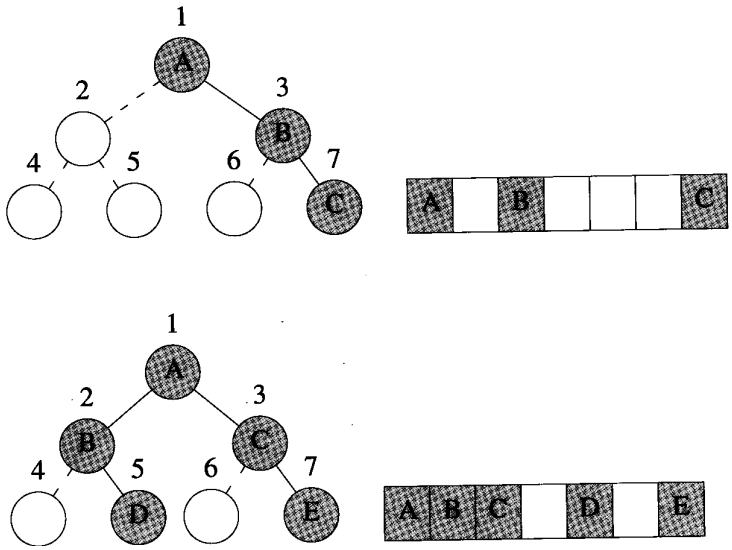


图8-8 不完全二叉树

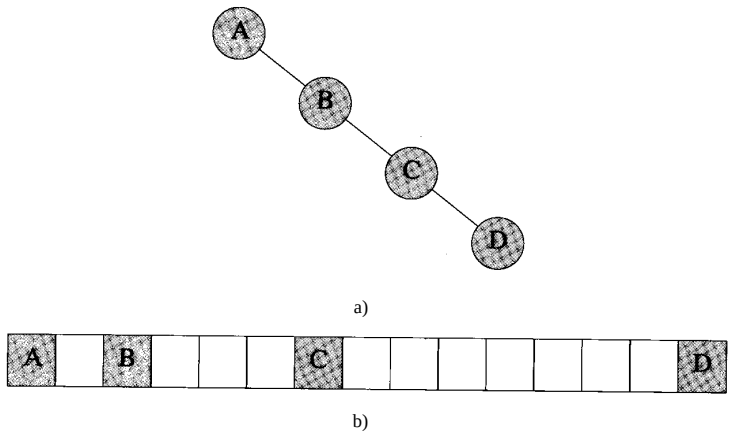


图8-9 右斜二叉树

a) 右斜树 b) 数组表示

8.4.2 链表描述

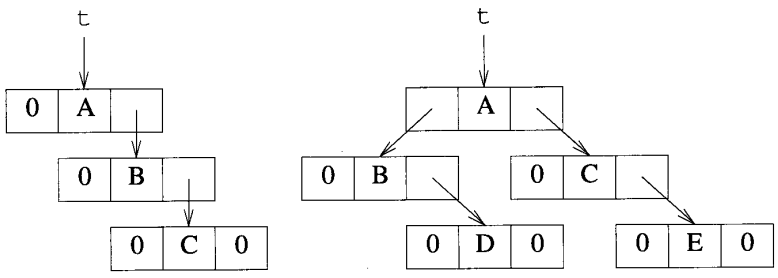
二叉树最常用的描述方法是用链表或指针。每个元素都用一个有两个指针域的节点表示，这两个域为LeftChild和RightChid。除此两个指针域外，每个节点还有一个 data域。可以像程序8-1那样把节点结构定义为 C++类模板。这种定义为二叉树节点提供了三种构造函数。第一种

无参数，初始化时节点的左右孩子域被置为 0（即NULL）；第二种有一个参数，可用此参数来初始化数据域，孩子域被置为0；第三种有3个参数，可用来初始化节点的3个域。

二叉树的边可用一个从父节点到子节点的指针来描述。指针放在父节点的指针域中。因为包括n 个元素的二叉树恰有$n-1$条边，因此将有$2n-(n-1)=n+1$个指针域没有值，这些域被置为0。图8-10给出图8-8中二叉树的链表描述。

程序8-1　链表二叉树的节点类

```
template <class T>
class BinaryTreeNode {
  friend void Visit(BinaryTreeNode<T> *);
  friend void InOrder(BinaryTreeNode<T> *);
  friend void PreOrder(BinaryTreeNode<T> *);
  friend void PostOrder(BinaryTreeNode<T> *);
  friend void LevelOrder(BinaryTreeNode<T> *);
  friend void main(void);
  public:
    BinaryTreeNode() {LeftChild = RightChild = 0;}
    BinaryTreeNode(const T& e)
        {data = e;
         LeftChild = RightChild = 0;}
    BinaryTreeNode(const T& e, BinaryTreeNode *l,
                   BinaryTreeNode *r)
        {data = e;
         LeftChild = l;
         RightChild = r;}
  private:
    T data;
    BinaryTreeNode<T> *LeftChild,  //左子树
                      *RightChild;  // 右子树
};
```



a)　　b)

图8-10　链表描述

用一个变量(在图8-10中为t) 来保存二叉树的根，用该变量的名称来指称根节点或整个二叉树，因此可以说根节点t 或二叉树t。从根节点开始，沿着LeftChild 和RightChild 指针域逐层进行搜索，可以访问二叉树t 中的所有节点。在二叉树中不设置指向父节点的指针一般不会有什么问题，因为在二叉树的大部分函数中并不需要此指针。若某些应用需要此指针，可在每个节

点增加一个指针域。

8.5 二叉树常用操作

二叉树的常用操作为：

- 确定其高度。
- 确定其元素数目。
- 复制。
- 在屏幕或纸上显示二叉树。
- 确定两棵二叉树是否一样。
- 删除整棵树。
- 若为数学表达式树，计算该数学表达式。
- 若为数学表达式树，给出对应的带括号的表达式。

以上所有操作可以通过对二叉树进行遍历来完成。在二叉树的遍历中，每个元素仅被访问一次。在访问时执行对该元素的所有操作。这些操作包括：把该元素显示在屏幕或纸上；计算以该元素为根的子树所表示的数学表达式的值；对二叉树中元素的个数加 1；删除表示该元素的节点等。

8.6 二叉树遍历

有四种遍历二叉树的方法：

- 前序遍历。
- 中序遍历。
- 后序遍历。
- 逐层遍历。

前三种遍历方法在程序8-2，8-3，8-4中给出。假设要遍历的二叉树采用前一节所介绍的链表的方法来描述，并且BinanyTreeNode被定义为类或模板结构。

程序8-2 前序遍历

```
template <class T>
void PreOrder(BinaryTreeNode<T> *t)
{// 对*t进行前序遍历
   if (t) {
     Visit(t);                    // 访问根节点
     PreOrder(t->LeftChild);      // 前序遍历左子树
     PreOrder(t->RightChild);     // 前序遍历右子树
     }
}
```

程序8-3 中序遍历

```
template <class T>
void InOrder(BinaryTreeNode<T> *t)
{// 对*t进行中序遍历
   if (t) {
```

```
    InOrder(t->LeftChild);        // 中序遍历左子树
    Visit(t);                     // 访问根节点
    InOrder(t->RightChild);       // 中序遍历右子树
    }
}
```

程序8-4　后序遍历

```
template <class T>
void PostOrder(BinaryTreeNode<T> *t)
{// 对*t进行后序遍历
  if (t) {
    PostOrder(t->LeftChild);      // 后序遍历左子树
    PostOrder(t->RightChild);     // 后序遍历右子树
    Visit(t);                     // 访问根节点
    }
}
```

在前三种方法中，每个节点的左子树在其右子树之前遍历。这三种遍历的区别在于对同一个节点在不同时刻进行访问。在进行前序遍历时，每个节点是在其左右子树被访问之前进行访问的；在中序遍历时，首先访问左子树，然后访问子树的根节点，最后访问右子树。在后序遍历时，当左右子树均访问完之后才访问子树的根节点。

图8-11给出程序8-2，8-3和8-4产生的结果。其中Visit(t)由cout << t->data代替。所输入的二叉树为图8-5所示的二叉树。

	a)	b)	c)
前序	*+*ab/cd*	*+++abcd*	*/+-a+xy*+b*ca*
中序	*a*b+c/d*	*a+b+c+d*	*−a+x+y/+b*c*a*
后序	*ab*cd/+*	*ab+c+d+*	*a−xy++b+ca**/*

图8-11　二叉树按前序，中序，后序遍历的结果

当对一棵数学表达式树进行中序，前序和后序遍历时，就分别得到表达式的中缀、前缀和后缀形式。中缀（infix）形式即平时所书写的数学表达式形式，在这种形式中，每个二元操作符（也就是有两个操作数的操作符）出现在左操作数之后，右操作数之前。在使用中缀形式时，可能会产生一些歧义。例如，$x+y\times z$ 可以理解为$(x+y)\times z$ 或$x+(y\times z)$。为了避免这种歧义，可对操作符赋于优先级并采用优先级规则来分析中缀表达式。在完全括号化的中缀表达式中，每个操作符和相应的操作数都用一对括号括起来。更甚者把操作符的每个操作数也都用一对括号括起来。如((*x*)+(*y*))，((*x*)+((*y*)*(*z*)))和(((*x*)+(*y*))*((*y*)+(*z*)))*(*w*)。这种表达形式可通过修改程序8-5的中序遍历算法得到。

程序8-5　输出完全括号化的中缀表达式

```
template <class T>
void Infix(BinaryTreeNode<T> *t)
{// 输出表达式的中缀形式
  if (t) {cout << '(';
      Infix(t->LeftChild);        // 左操作数
```

```
        cout << t->data;            // 操作符
        Infix(t->RightChild);       // 右操作数
        cout << ')';}
}
```

在后缀（postfix）表达式中，每个操作符跟在操作数之后，操作数按从左到右的顺序出现。在前缀（prefix）表达式中，操作符位于操作数之前。在前缀和后缀表达式中不会存在歧义。因此，在前缀和后缀表达式中都不必采用括号或优先级。从左到右或从右到左扫描表达式并采用操作数栈，可以很容易确定操作数和操作符的关系。若在扫描中遇到一个操作数，把它压入堆栈，若遇到一个操作符，则将其与栈顶的操作数相匹配。把这些操作数推出栈，由操作符执行相应的计算，并将所得结果作为操作数压入堆栈。

在逐层遍历过程中，按从顶层到底层的次序访问树中元素，在同一层中，从左到右进行访问。由于遍历中所使用的数据结构是一个队列而不是栈，因此写一个按层遍历的递归程序很困难。程序 8-6用来对二叉树进行逐层遍历，它采用了队列数据结构 (见6.3节中定义的类LinkedQueue)。队列中的元素指向二叉树节点。当然，也可以采用公式化队列。

程序8-6 逐层遍历

```
template <class T>
void LevelOrder(BinaryTreeNode<T> *t)
{// 对*t逐层遍历
  LinkedQueue<BinaryTreeNode<T>*> Q;
  while (t) {
    Visit(t); // 访问 t

    // 将t的右孩子放入队列
    if (t->LeftChild) Q.Add(t->LeftChild);
    if (t->RightChild) Q.Add(t->RightChild);

    // 访问下一个节点
    try {Q.Delete(t);}
    catch (OutOfBounds) {return;}
    }
}
```

程序8-6中，仅当树非空时，才进入while循环。首先访问根节点，然后把其子节点加到队列中。当队列添加操作失败时，由Add引发NoMem异常，由于没有捕获该异常，因此当异常发生时LevelOrder函数将退出。在把t的子节点加入队列后，要从队列中删除t元素。若队列为空，则由Delete引发OutOfBounds异常，这由catch语句来处理。因为一个空队列意味着遍历的结束，所以执行return语句。若队列非空，则Delete把所删除的元素返回至变量t。被删除的元素指向下一个欲访问的节点。

设二叉树中元素数目为 n。这四种遍历算法的空间复杂性均为 $O(n)$，时间复杂性为 $\Theta(n)$。当t 的高度为 n 时(也就是图8-9给出的右斜二叉树的情况)，通过观察其前序、中序和后序遍历时所使用的递归栈空间即可得到上述结论。当 t为满二叉树时，逐层遍历所需要的队列空间为 $\Theta(n)$。每个遍历算法花在树中每个节点上的时间为 $\Theta(1)$ (假设访问一个节点的时间为 $\Theta(1)$)。

练习

8. 编写公式化描述的二叉树的前序遍历程序。假设二叉树的元素存储在数组 a 中，其中Last 用于保存树中最后一个元素的位置。当位置i 中没有元素时，a[i]=0。给出该程序的时间复杂性。

9. 按中序遍历方法完成练习1。

10. 按后序遍历方法完成练习1。

11. 按逐层遍历方法完成练习1。

12. 编写一个C++函数，用于复制一个公式化描述的二叉树。

13. 编写两个C++函数，复制用BinaryTreeNode模板结构描述的二叉树t。第一个函数按前序遍历整棵树，第二个按后序遍历。两个函数所需要到的递归栈空间有什么不同？

14. 编写一个函数，计算用模板结构 BinanyTreeNode描述的表达式树的值。假设每个节点有一个value域，常量和变量的value域内有其相应的值。

15. 编写一个函数删除二叉树t (提示：采用后序遍历方法)。假设t 是一链表树，调用C++函数delete释放节点空间。

16. 写一交互式进程按中序方法来遍历链表二叉树，在程序中可采用公式化堆栈。尽量使进程做得比较完美。遍历需多少栈空间？给出栈空间与节点数n 间的关系。

17. 按前序遍历方法完成练习16。

18. 接后序遍历方法完成练习16。

*19. 设t 是数据域类型为 int 的二叉树，每个节点的数据都不相同。数据域的前序和中序排列是否可唯一地确定这棵二叉树？如果能，给出一函数来构造此二叉树。指出函数的时间复杂性。

20. 按前序和后序遍历方法完成练习19。

*21. 按中序和后序遍历方法完成练习19。

22. 编写一个C++函数，输入后缀表达式，构造其二叉树表示。设每个操作符有一个或两个操作数。

*23. 用前缀表达式完成练习22。

24. 编写一个C++函数，把后缀表达式转换为完全括号化的中缀表达式。

*25. 用前缀表达式完成练习24。

*26. 把一个表达式的中缀形式（不一定是完全括号化的）转换成后缀形式。在该练习中假定允许的操作符为二元操作符 +、一、×、/，允许的分界符为 (和)。因为操作数的顺序在中缀、前缀、后缀中都是一样的，所以在从中缀向后缀、前缀转换时，仅从左到右扫描中缀表达式，看到操作数时，则直接输出，而把操作符保留在栈中，直到合适的时机输出（具体的输出时机由操作符和分界符(的优先级来确定)。假定＋和－的优先级为1，×和 / 的优先级为2。栈外的（的优先级为3，栈内的（的优先级为0。

*27. 完成练习26，要求产生前缀表达式。

*28. 完成练习26，要求输出二叉树形式。

29. 编写一个函数，计算以后缀表达式的值。假设表达式以数组方式描述。

8.7 抽象数据类型BinaryTree

现在我们已明白什么是二叉树，可以用抽象数据类型来描述二叉树 (见ADT8-1)。注意：

抽象数据类型的描述应独立于具体的实现形式。尽管我们希望在二叉树上进行的操作非常多，但这里只列出了几个常用的操作。在8-9节将对ADT 8-1进行扩充。

ADT8-1 二叉树的抽象数据类型描述

```
抽象数据类型 BinaryTree{
实例
    元素集合；如果不空，则被划分为根节点、左子树和右子树；
        每个子树仍是一个二叉树
操作
    Create ()：创建一个空的二叉树；
    IsEmpty：如果二叉树为空，则返回true ，否则返回false
    Root (x)：取x为根节点；如果操作失败，则返回 false，否则返回true
    MakeTree (root，left，right)：创建一个二叉树，root作为根节点，left作为左子树，right作为右子树
    BreakTree (root，left，right)：拆分二叉树
    PreOrder：前序遍历
    InOrder：中序遍历
    PostOrder：后序遍历
    LevelOrder：逐层遍历
}
```

8.8 类BinaryTree

程序8-7给出了二叉树抽象数据类型的C++定义。此定义中采用了链接描述的二叉树，对应的C++类为BinaryTree。函数Visit作为遍历函数的参数，以利于不同操作的实现。

程序8-7 二叉树类定义

```
template<class T>
class BinaryTree {
  public :
    BinaryTree( ) {root = 0; };
    ~BinaryTree( ) { };
    bool IsEmpty( ) const
      {reurn ((root) ? false : true);}
    bool Root(T& x) const;
    void MakeTree(const T& element，BinaryTree<T>& left，BinaryTree<T>& right);
    void BreakTree(T& element，BinaryTree<T>& left ，BinaryTree<T>& right);
    void PreOrder(void(*Visit) (BinaryTreeNode<T> *u))
        {PreOrder(Visit，root);}
    void Inorder(void (*Visit) (BinaryTreeNode<T> *u))
        {Inorder(Visit，root);}
    void PostOrder(void(*Visit )  (BinaryTreeNode<T> *u));
        {Postorder (Visit, root); }
    void LevelOrder(void(*Visit) (BinaryTreeNode<T> *u));
  private:
    BinaryTreeNode<T>  *root; // 根节点指针
```

```
    void PreOrder(void (*Visit) (BinaryTreeNode<T> *u)，BinaryTreeNode<T> *t);
    void  InOrder(void(*Visit) (BinaryTreeNode<T> *u)，BinaryTreeNode<T> *t);
    void  PostOrder(void(*Visit) (BinaryTreeNode<T> *u)，BinaryTreeNode<T> *t);
};
```

程序8-8给出了共享成员函数Root，MakeTree，BreakTree的代码，而程序8-9和8-10给出私有遍历函数的代码。函数MakeTree和BreakTree要求参与操作的三棵树应该互不相同。若它们相同的话，则程序有可能得出错误结果。例如调用Y.MakeTree(e,X,X)得到一棵二叉树，其左右子树共享同样的节点。这种共享只有在X为空二叉树时才正确。X.MakeTree(e,X,Y)将在返回之前把X.root(left.root)置为0。因此不管X，Y的初始值是什么，在调用MakeTree之后，X均为空二叉树。在练习30中，要求给出能弥补以上缺陷的MakeTree和BreakTree版本。

程序8-8 共享成员函数的实现

```
template<class T>
bool BinaryTree<T>::Root(T& x) const
{// 取根节点的data域，放入x
 // 如果没有根节点，则返回false
  if (root) {x = root->data;
             return true;}
  else return false;  // 没有根节点
}

template<class T>
void BinaryTree<T>::MakeTree(const T& element, BinaryTree<T>& left, BinaryTree<T>& right)
{// 将left, right和element 合并成一棵新树
 // left, right和this必须是不同的树
  // 创建新树
  root = new BinaryTreeNode<T>
          (element, left.root, right.root);

  // 阻止访问left和right
  left.root = right.root = 0;
}

template<class T>
void BinaryTree<T>::BreakTree(T& element, BinaryTree<T>& left, BinaryTree<T>& right)
{// left, right和this必须是不同的树
  // 检查树是否为空
  if (!root) throw BadInput(); // 空树

  // 分解树
  element = root->data;
  left.root = root->LeftChild;
  right.root = root->RightChild;

  delete root;
```

```
  root = 0;
}
```

程序8-9 前序，中序和后序遍历

```
template<class T>
void BinaryTree<T>::PreOrder(void(*Visit)(BinaryTreeNode<T> *u), BinaryTreeNode<T> *t)
{// 前序遍历
  if (t) {Visit(t);
          PreOrder(Visit, t->LeftChild);
          PreOrder(Visit, t->RightChild);
          }
}

template <class T>
void BinaryTree<int>::InOrder(void(*Visit)(BinaryTreeNode<T> *u), BinaryTreeNode<T> *t)
{// 中序遍历
  if (t) {InOrder(Visit, t->LeftChild);
          Visit(t);
          InOrder(Visit, t->RightChild);
          }
}

template <class T>
void BinaryTree<T>::PostOrder(void(*Visit)(BinaryTreeNode<T> *u), BinaryTreeNode<T> *t)
{// 后序遍历
  if (t) {PostOrder(Visit, t->LeftChild);
          PostOrder(Visit, t->RightChild);
          Visit(t);
          }
}
```

程序8-10 逐层遍历

```
template<class T>
void BinaryTree<T>::LevelOrder(void(*Visit)(BinaryTreeNode<T> *u))
{// 逐层遍历
  LinkedQueue<BinaryTreeNode<T>*> Q;
  BinaryTreeNode<T> *t;
  t = root;
  while (t) {
    Visit(t);
    if (t->LeftChild) Q.Add(t->LeftChild);
    if (t->RightChild) Q.Add(t->RightChild);
    try {Q.Delete(t);}
    catch (OutOfBounds) {return;}
```

```
    }
}
```

程序8-11中使用了BinaryTree类。程序中构造了一个四节点的二叉树，并进行了前序遍历以确定树中的节点数目。

程序8-11　类BinaryTree的应用

```
#include <iostream.h>
#include "binary.h"

int count = 0;
BinaryTree<int> a,x,y,z;

template<class T>
void ct(BinaryTreeNode<T> *t) {count++;}

void main(void)
{
  y.MakeTree(1,a,a);
  z.MakeTree(2,a,a);
  x.MakeTree(3,y,z);
  y.MakeTree(4,x,a);
  y.PreOrder(ct);
  cout <<  count << endl;
}
```

8.9　抽象数据类型及类的扩充

本节将ADT8-1中的抽象数据类型进行扩充，增加如下二叉树操作：

- *PreOutput* ( )：按前序方式输出数据域。
- *InOutput* ( )：按中序方式输出数据域。
- *PostOutput* ( )：按后序方式输出数据域。
- *LevelOutput* ( )：逐层输出数据域。
- *Delete* ( )：删除一棵二叉树，释放其节点。
- *Height* ( )：返回树的高度。
- *Size* ( )：返回树中节点数。

8.9.1　输出

四个输出函数(前序输出、中序输出、后序输出、逐层输出)可以通过定义一个私有静态成员函数Output来实现，此静态成员函数的形式如下：

```
static void Output(BinaryTreeNode<T> *t)
        { cout << t->data << ' ' ;}
```

四个共享输出函数的形式如下：

```
void PreOutput( )
```

```
    { PreOrder(Output, root); cout << end1;}
void InOutput( )
    { InOrder(Output, root); cout << end1;}
void PostOutput( )
    {PostOrder(Output, root); cout << end1;}
void LevelOutput( )
    { LevelOrder(Output); cout << end1;}
```

因为Visit操作的时间复杂性为$\Theta(1)$，对包括n个节点的二叉树来说，每种遍历方法所花费时间均为$\Theta(n)$ (假设遍历成功)，因此每种输出方法的时间复杂性为$\Theta(n)$。

8.9.2 删除

要删除一棵二叉树，需要删除其所有节点。可以通过后序遍历在访问一个节点时，把其删除。也就是说先删除左子树，然后右子树，最后删除根。共享成员函数 Delete 的形式如下：

```
void Delete( ) { PostOrder (Free, root); root = 0;}
```

其中Free为私有成员函数：

```
static void Free(BinaryTreeNode<T> *t) {delete t;}
```

当要删除的二叉树中有n个节点时，Delete 函数的时间复杂性为$\Theta(n)$。

8.9.3 计算高度

通过进行后序遍历，可以得到二叉树的高度。首先得到左子树的高度 hl，然后得到右子树的高度hr。此时，树的高度为：

```
max{hl, hr} + 1
```

不幸的是，不能用程序8-9中给出的后序遍历代码，因为在进行遍历时要有返回值 (也就是子树的高度)。为了实现共享成员函数Height，在程序8-7的public部分增加如下语句：

```
int Height( ) const {return Height (root);}
```

在private部分增加：

```
int Height (BinaryTreeNode<T> *t) const;
```

私有成员函数Height在程序8-12中给出。其时间复杂性为$\Theta(n)$，其中n是二叉树中的节点数。

程序8-12 计算二叉树的高度

```
template <class T>
int BinaryTree<T>::Height(BinaryTreeNode<T> *t) const
{// 返回树*t的高度
  if (!t) return 0;                    // 空树
  int hl = Height(t->LeftChild);       // 左子树的高度
  int hr = Height(t->RightChild);      // 右子树的高度
  if (hl > hr) return ++hl;
  else return ++hr;
}
```

8.9.4 统计节点数

可以用上述四种遍历方法中的任何一种来获取二叉树中的节点数。因为在每种遍历方法中对每个节点都仅访问一次，只要在访问每个节点时将一个全局计数器增加 1即可。在程序8-11中，使用用户自定义函数 ct来获得到二叉树的节点数。通过把如下代码加入到程序 8-7中的public部分，可以定义一个等价类成员函数Size。

```
int Size ( )
 { _count = 0;
 PreOrder(Add1，root);
 return _count;}
```

_count是在类定义之外定义的一个整型变量，其定义如下：

```
int _count;
```

私有成员函数Addl的原型为：

```
static void Add1(BinaryTreeNode<T> *t) { _count++;}
```

Size的时间复杂性为$\Theta(n)$，其中n为二叉树中节点数。

练习

30. 编写二叉树成员函数MakeTree和BreakTree的新版本，考察每个操作所用到的三棵树是否相同。如果有相同者，确定应采取什么措施并给出函数代码。

31. 1) 扩充抽象数据类型*BinaryTree*，增加*Compare*(*X*)操作，用来比较该二叉树与二叉树*X*。若两棵二叉树相同返回true，否则返回false。

2) 扩充C++类BinaryTree，增加用于二叉树比较的共享成员函数，并测试代码。

32. 1) 扩充抽象数据类型*BinaryTree*，增加复制二叉树的操作Copy( )。若此操作失败，引发一个合适的异常。

2) 扩充C++类BinaryTree，增加共享成员函数Copy。测试代码的正确性。

*33. 从类BinaryTree派生子类Expression，该类中包括以下操作

1) 输出完全括号化的中缀表达式。

2) 输出前缀和后缀表达式。

3) 从前缀形式转换成表达式树。

4) 从后缀形式转换成表达式树。

5) 从中缀形式转换成表达式树。

6) 计算表达式树的值。

用适当数据测试代码的正确性。

8.10 应用

8.10.1 设置信号放大器

在一个分布网络中，资源被从生产地送往其他地方。例如，汽油或天然气经过管道网络从汽油/天然气生产基地输送到消耗地。同样的，电力也是通过电网从发电厂输送到各消耗点。可以用术语信号（signal）来指称所输送的资源(汽油、天然气、电力等)。当信号在网络中传输时，其性能的某一个或几个方面可能会有所损失或衰减。例如在传输过程中，天然气的气压会减少，电的电压会降低。另一方面，当信号在网络中传输时，噪声会增加。在信号从信号源到

消耗点传输过程中，仅能容忍一定范围内的信号衰减。为了保证信号衰减不超过容忍值(tolerance)，应在网络中合适的位置放置信号放大器（signal booster)。信号放大器可以增加信号的压强或电压使其与源端的相同；可以增强信号，使信号与噪声之比与源端的相同。在本节中，将设计一个算法以确定把信号放大器放在何处。目标是要使所用的放大器数目最少并且保证信号衰减(与源端信号相关)不超过给定的容忍值。

为简化问题，设分布网络是一树形结构，源是树的根。树中的每一节点 (除了根)表示一个可以用来放置放大器的子节点，其中某些节点同时表示消耗点。信号从一个节点流向其子节点。图8-12给出一树形分布网络。每条边上标出从父节点到子节点的信号衰减量。信号衰减的单位可认为是附加剂。在图8-12中信号从节点p 流到节点v 的衰减量为5。从节点q 到节点x 的衰减量为3。

设$d(i)$ 表示节点i 与其父节点间的衰减量。因此，在图 8-12中，$d(w)=2$，$d(p)=0$，$d(r)=3$。因为信号放大器只能放在树的节点上，若节点i 的$d(i)>$容忍值，则不可能通过放置放大器来使信号的衰减不超过容忍值。例如，若容忍值为 1，则在图8-12中是不可能在p 和r 间通过放置放大器使衰减量小于等于1的。

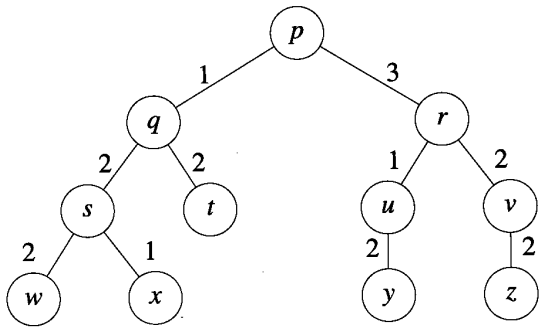


图8-12 树形分布网络

对任一节点i，设$D(i)$ 为从节点i 到以i 为根节点的子树的任一叶子的衰减量的最大值。若i 为叶节点，则$D(i)=0$。图8-12中，当$i\in\{w, x, t, y, z\}$ 时，$D(i)=0$。对于其他节点，$D(i)$ 可以用下式来表示：

$$D(i)=\max_{j\text{是}i\text{的一个孩子}}\{D(j)+d(j)\}$$

因此$D(s)=2$。在此公式中，要计算节点的D值，必须先计算其子节点的D值。因而必须遍历整棵树，先访问子节点然后访问父节点。这样当访问一个节点时，就可同时计算其 D值。这种遍历方法是高度大于2的树的后序遍历的一种自然扩充。

假设按照上面方法，在计算D的过程中，遇到一节点i，且其有一子节点j 满足$D(j)+d(j)>$容忍值。若不在j 处放置放大器，则从i 节点到叶节点的信号衰减量将会超过容忍值，即使在i 处放置放大器也是如此。例如在图8-12中，当计算$D(q)$ 时，有$D(s)+d(s)=4$。若容忍值为3，则在q 点或其祖先的任意一点放置放大器，并不能减少 q 与其后代间的衰减量。必需在s 点放一个放大器或在其子节点放一个或多个放大器。若在节点s 处放一放大器，则$D(q)=2$。

计算D并放置放大器s 的伪代码为：

```
D(i)=0;
for (i 的每个孩子 j)
    if ((D(j)+d(j))>tolerance) 在 j 放置放大器;
    else D(i)=max{D(i), D(j)+d(j)};
```

在图8-12中应用此计算方法，可知应在节点r、s 和v 处放置放大器（如图8-13所示)。每个节点的D值在节点内给出。

定理8-1 以上进程所需放大器数目最少。

证明 可通过对树的节点数n 进行归纳来证明。当$n=1$时，定理显然成立。设$n\leq m$ 时，定理成立，其中m 为任意的自然数。设t 为有$n+1$个节点的树。令X 为由上述进程所确定的放置放大器

的节点的集合，W为满足容忍值限制的拥有最少放大器的节点集合，因此只需证$|X| = |W|$。

若$|X| = 0$，则$|X| = |W|$。若$|X|>0$，则设z为由上述进程给出的放置第一个放大器的节点，t_z为树t中以z为根的子树。因为$D(z)+d(z)$>容忍值，W至少需包含t_z中的某个节点u。若W中还包含了u以外的元素，则W一定不是最好的方案，因为通过在$W-\{$所有的象u这样的节点$\} + \{z\}$集合中放置放大器也能满足容忍值。因此W中只包含节点u。设$W'=W-\{u\}$，t'为从树t中除去子树t_z（但保留z）而得到的树，则对于t'而言，W'为满足容忍值的放置放大器数目最少的方案。而且，$X'=X-\{z\}$在树t'也满足容忍值。因t'中的节点数小于$m+1$，$|X'| = |W'|$，因此$|X|=|X'|+1 = |W'|+1=|W|$。

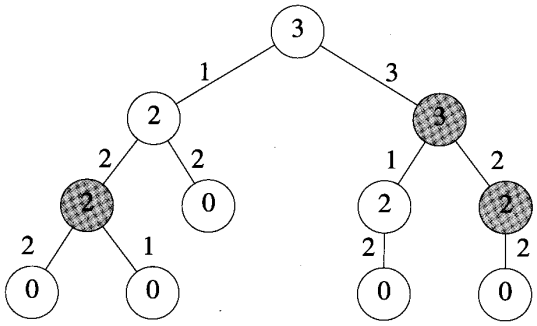


图8-13 放置信号放大器的分布网络

当分布树中每个节点的孩子数都少于2时，可利用程序8-7给出的类BinaryTree 和程序8-13给出的类Booster 把该树描述成二叉树。域boost 用来区分节点是否放置了放大器。二叉树的数据域类型为Booster。因为在8.9节中对程序8-7进行扩充时，定义了一个静态函数Output 用来输出节点的数据域，因此必须重载输出操作符 <<。

程序8-13 类Booster

```
class Booster {
   friend void main(void);
   friend void PlaceBoosters(BinaryTreeNode<Booster> *);
  public:
    void Output(ostream& out) const
    {out << boost << ' ' << D << ' ' << d << ' ';}
  private:
    int D,          // 叶节点的衰减量
        d;          // 父节点的衰减量
    bool boost;  //当且仅当本处设置放大器，则boost为true
};

// 重载操作符 <<
ostream& operator<<(ostream& out, Booster x)
  {x.Output(out); return out;}
```

可通过在二叉树中进行后序遍历来计算 D值及确定放置放大器的节点集合。程序 8-14用于访问每个节点。PlaceBoosters为Booster的一个友元，tolerance为一个全局变量。

程序8-14 在二叉树中放置放大器并计算D值

```
void PlaceBoosters(BinaryTreeNode<Booster> *x)
{// 计算 *x.处的衰减量，如果衰减量超出了容忍值，则设置放大器
```

```
    BinaryTreeNode<Booster> *y = x->LeftChild;
    int degradation;
    x->data.D = 0;  // 初始化 x的衰减量
    if (y) {// 从左孩子来计算
           degradation = y->data.D + y->data.d;
           if (degradation > tolerance)
             {y->data.boost = true;
              return;}
           else x->data.D = degradation;
          }
    y = x->RightChild;
    if (y) {// 从右孩子来计算
           degradation = y->data.D + y->data.d;
           if (degradation > tolerance)
             {y->data.boost = true;
           else if (x->data.D < degradation)
             x->data.D = degradation;
          }
}
```

若X为类BinaryTree<Booster>的成员，且d中的值为衰减值， boost域为0，则调用X.PostOrder(PlaceBoosters) 可重新为D和boost 赋值。PlaceBoosters 的结果可通过X.PostOutput输出。由于PlaceBoosters 的时间复杂性为$\Theta(1)$，所以调用X.PostOrder(PlaceBoosters)所花费的时间为$\Theta(n)$，其中n 为树中的节点数。

树的二叉树描述

当分布树t 含有超过两个孩子的节点时，同样可以用二叉树来描述该树。此时，对于树 t 的每个节点x，可用其孩子节点的RightChild 域把x 的所有孩子链成一条链。x 节点的LeftChild指针指向该链的第一个节点。x 节点的RightChild 域用来指向x的兄弟。

图8-14给出树及其二叉树描述。实线表示指向左孩子的指针，虚线表示指向其右孩子的指针。

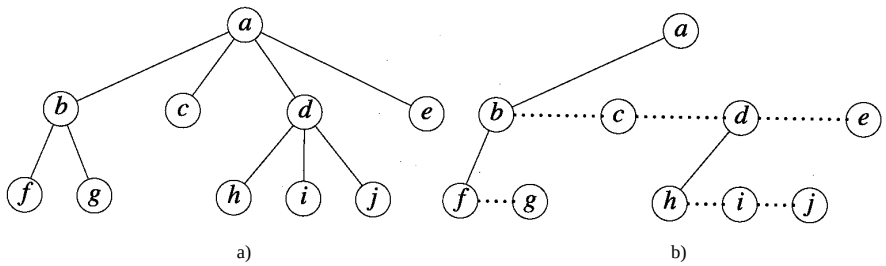


图8-14 树及其二叉树描述

a) 一棵树 b) 转换成二叉树

当一棵树用二叉树描述时，调用 X.PostOrder(PlaceBoosters)不会产生预期的结果。在练习37中要求设计新的计算D和boost的函数。

8.10.2 在线等价类

在3.8.3节中讨论了在线等价类问题。有 n个元素从1到n编号，最开始，每一个元素在其自

己的类中，然后执行一系列 Find和Combine操作。操作Find(e)返回元素e 所在类的唯一特征，而Combine(a,b)用来合并包含 *a*和*b*的类。在 3.8.3节中， Combine(a,b)是通过使用合并操作Union(i,j)完成的。其中i = Find(a), j = Find(b)。3.8.3节中的解决方案使用了链表，其复杂性为O(*n*+*u*log*u*+*f*)，其中*u* 是合并操作的次数，*f* 是查找操作的次数。在线等价类问题也称作离散集合合并/搜索问题（disjoint set union-find problem）。注意等价类可以看成元素的离散集合。

在这本节中，将采用另一种方案来解决在线等价类问题，其中把每个集合 (类)描述为一棵树。图8-15给出一些描述成树的集合。而树中每个非根节点都指向其父节点。用根元素作为集合标识符。因此可以说元素 1、2、20、30等在以0为根的集合中；元素 11、16、25、28在以16为根的集合中；元素 15在以15为根的集合中；元素 26和32在以26为根的集合中（或简称为集合26）。

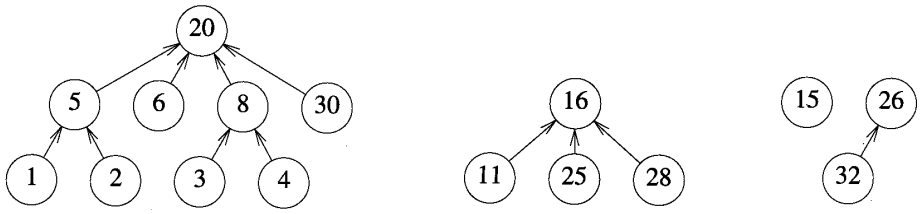


图8-15 离散集合的树形描述

1. 树形描述

合并/搜索问题的解决方案是使用模拟指针一个极好的例子。这里采用链表方法来描述树。每个节点必需有一个parent 域，但不必有children 域。要求能直接访问每个元素。为找到含有元素10的集合，需确定哪个节点表示元素 10，然后由其parent 域一直搜索至根。若节点索引号为1到n 且节点*e* 表示元素*e*，则很容易实现直接访问。每个 parent 域给出父节点的索引，因此parent 域为整数类型。图8-16就是采用这种方法来描述图8-15的。节点内的数字是其parent域的值，节点外的数字为其索引。索引同时也就是该节点所表示的元素。根节点的parent域被置为0。因为没有节点的索引会为0，因此parent=0 表示不指向任何节点(即为空链)。

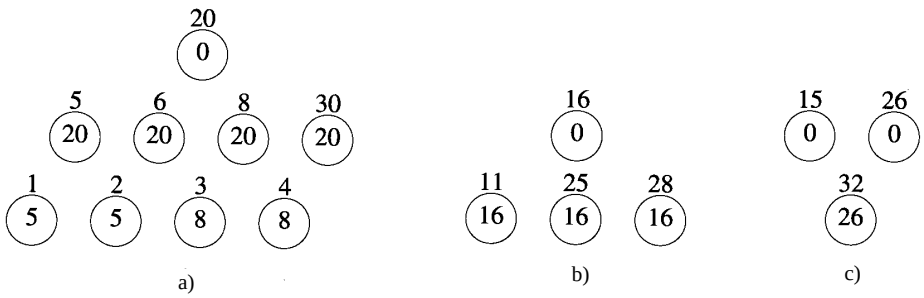


图8-16 图8-15中树的描述

因为每个节点只有一个域，只需如下定义：

```
int *parent；
```

2. 操作

开始时每个元素在仅包含它自己的一个集合中。为了创建初始结构，需分配数组 parent并置parent[1: *n*]为0。这由程序8-15中Initialize函数来完成。Initialize的复杂性为$\Theta(n)$。

程序8-15 基于树结构的合并/搜索问题解决方案

```
void Initialize(int n)
{// 初始化，每个类/树有一个元素
  parent = new int[n+1];
  for (int e = 1; e <= n; e++)
    parent[e] = 0;
}

int Find(int e)
{//返回包含 e的树的根节点
  while (parent[e])
    e = parent[e];  // 上移一层
  return e;
}

void Union(int i, int j)
{// 将根为 i 和 j的两棵树进行合并
  parent[j] = i;
}
```

为找到包含元素e 的集合，从节点e 出发，由parent 链搜索至根节点。若e=4，集合的状态如图8-15a所示，从4开始。由parent链到达节点8。然后由节点8的parent链到达节点20，而节点20的parent为0，则20是根即该集合的标识符，程序 8-15中函数Find用来完成此功能。程序假设1≤e≤n(也就说e为有效值)。Find的复杂性为O (*h*)，其中*h* 为包含元素e 的树的高度。

根为i 和j (i≠j)的集合的合并是通过把i作为j的子树，或把j 作为i 的子树来实现的。例如，取i=16且j=26(如图8-15所示)，若把i 作为j 的子树则结果为图8-17a，而当j 为i 的子树时，结果为图8-17b。程序8-15中的函数Union 可用来执行合并操作，假设在调用Union之前已检查了i≠j。通常把j 作为i 的子树来处理。Union 的时间复杂性为$\Theta(1)$。

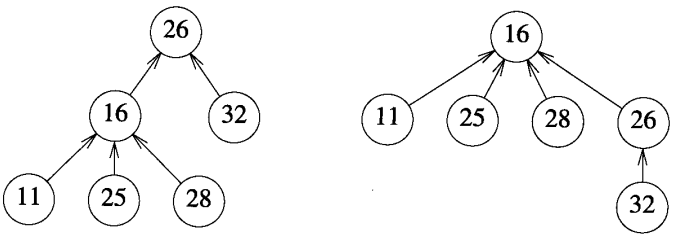


图8-17 合并操作

3. 性能评价

假设要执行u 次合并和f 次查找。因为每次合并前都必须执行两次查找，因此可假设 $f>u$。每次合并所需时间为$\Theta(1)$。而每次查找所需时间由树的高度决定。在最坏情况下，有 m个元素的树的高度为m。当执行以下操作序列时，即可导致最坏情况出现：

Union(2,1)，*Union*(3,2)，*Union*(4,3)，*Union*(5,4)，…

因此每一次查找需花费$\Theta(q)$时间，其中q是在执行查找之前所进行的合并操作的次数。

4. 性能改进

在对树i 和树j 进行合并操作时，可以通过使用重量规则或高度规则来提高合并 /搜索算法的性能。

定义 [**重量规则**] 若树i 节点数少于树j 节点数，则将j 作为i 的父节点。否则，将i 作为j 的父节点。

定义 [**高度规则**] 若树i 的高度小于树j 的高度，则将j 作为i 的父节点，否则将i 作为j 的父节点。

若对图8-15a 和b 中两树进行合并，则无论是采用重量规则还是高度规则，根为 16的树都将成为根为20的树的子树。若对图8-18a 和b 的树进行合并操作，若按照重量规则，根为 16的树为根20树的子树。然而，若按照高度规则，根为20的树成为根为16的树的子树。

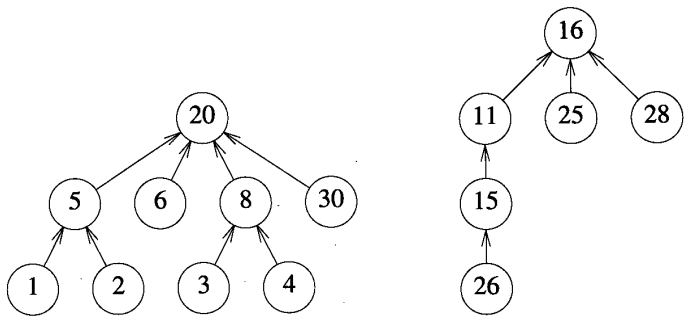


图8-18 两棵树

为了把重量规则应用到合并算法中，在每个节点上增加一个布尔域 root。若当前节点为根节点则其root 域为true。每个根节点的parent 域用来保存该树中节点的总数。对于图8-15的树，当且仅当i=20、16、15或26时，root[i]=true。并且当i=20、16、15和26时，parent[i]分别为9、4、1和2。余下节点的parent域不变。

初始化、查找和合并的代码见程序8-16，root 的定义如下：

```
bool *root;
```

练习44要求用一个数组来重新编写这三个函数，每个数组元素包含两个域： parent和root。

虽然执行合并操作的时间有所增加，但仍为一个常数，即 $\Theta(1)$。定理8-1给出进行查找操作所需时间的最大值。

程序8-16 用重量规则进合并

```
void Initialize(int n)
{// 初始化，每个类/树有一个元素
  root = new bool[n+1];
  parent = new int[n+1];
  for (int e = 1; e <= n; e++) {
     parent[e] = 1;
```

```
    root[e] = true;}
}

int Find(int e)
{// 返回包含 e的树的根节点
  while (!root[e])
    e = parent[e];  // 上移一层
  return e;
}

void Union(int i, int j)
{// 将根为 i 和 j的两棵树进行合并
 // 利用重量规则
  if (parent[i] < parent[j]) {
    // i 成为j的子树
    parent[j] += parent[i];
    root[i] = false;
    parent[i] = j; }
  else {// j 成为 i 的子树
    parent[i] += parent[j];
    root[j] = false;
    parent[j] = i;}
}
```

定理8-2 [**重量规则定理**] 假设从单元素集合出发，用重量规则进行合并操作(如程序8-16)。若按此方法构建有p个节点的树t，则t的高度最多为$\lfloor \log_2 p \rfloor + 1$。

证明 当p=1时定理显然成立。假设当$i \le p$-1时，对所有具有i个节点的树，定理均成立。下面将证明$i=p$时定理也成立。设由程序8-16产生的树t有p个节点。最后一次合并操作为*Union* (k, j)，设树j的节点数为m，树k的节点数为p-m。不失一般性可假设$1 \le m \le p/2$。则树t的高度与k的高度要么相同，要么比j的高度大1。若为前者，则t的高度$\le \lfloor \log_2 (p-m) \rfloor + 1 \le \lfloor \log_2 p \rfloor + 1$。若后者为真则$t$的高度$\le \lfloor \log_2 m \rfloor + 2 \le \lfloor \log_2 p/2 \rfloor + 2 \le \lfloor \log_2 p \rfloor + 1$。

若从单元素集合出发，混合执行u次合并和f次查找序列，则每个集合不会超过u+1个元素。由定理8-1知，若使用重量规则，合并和查找序列的代价(不包括初始化时间)为$O(u + f \log u)$。若采用高度规则而非重量规则时，定理8-1的高度限制仍然适用。练习40，41和42给出了高度规则的具体应用。

在最坏情况下的性能可通过修改程序8-16的查找过程来提高，方法是缩短从元素e到根的查找路径。路径的缩短可以通过称为路径压缩（path compression）的过程实现，其实现最少有3种不同方法。第一种方法，称为紧凑路径法（path compaction），可改变从节点e (要查找的节点)到根的路径上所有节点的指针，使这些指针直接指向根节点。考察图 8-19，若执行Find(10)操作，节点10、15和3位于从10到根的路径上，将其parent域改为2，就得到图8-20的树。(因为节点3本来已指向2，其parent域可不必修改。但在写程序时，为简化起见仍对其进行修改。)

虽然路径紧凑增加了单个查找操作的时间，但它减少了此后查找操作的时间。例如在图 8-20的紧凑路径中查找元素10和15会更快。程序8-17给出紧凑规则的实现。

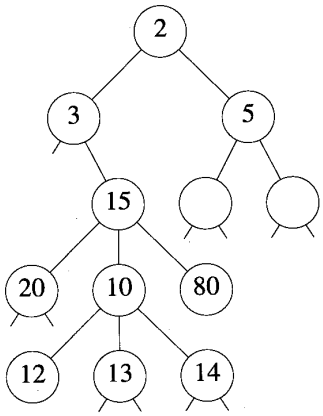


图8-19 样树

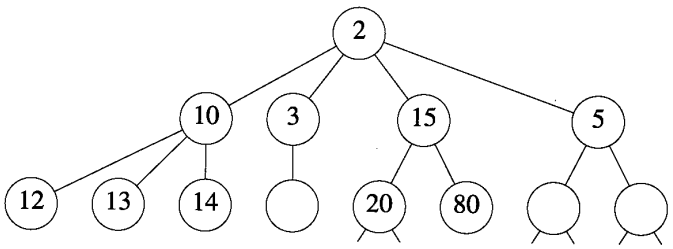


图8-20 路径紧凑

程序8-17 路径紧凑的实现代码

```
int Find(int e)
{// 返回包含 e的树的根节点
 // 并对从e到根节点的路径进行紧凑
   int j = e;
   // 搜索根节点
   while (!root[j])
     j = parent[j];

   // 路径紧凑
   int f = e;  // 从 e 开始
   while (f != j) {// f 不是根节点
     int pf = parent[f];
     parent[f] = j;  // 上移至2 层
     f = pf;         // f 移至原父节点
     }

   return j;
}
```

余下的两种路径压缩方法称为路径分割（path splitting）和路径对折（path halving）。在路径分割中，改变从e 到根节点路径上每个节点（除了根和其子节点）的 parent指针，使其指向各自的祖父节点。在图8-19中，路径分割从节点13开始，结果得到图8-21中的树。在路径分割时，只考虑从e 到根节点的一条路径就足够了。

在路径对折中，改变从e 到根节点路径上每隔一个节点（除了根和其子节点）的parent 域，使其指向各自的祖父节点。在路径对折中指针改变数仅为路径分割中的一半。同样，在路径对折中只考虑从e 到根节点的一条路径就够了。图 8-22中给出图8-19从节点13开始进行路径对折的结果。

采用了合并和查找的性能改进算法，执行一串交错的合并和查找操作所需的时间几乎与合并和查找的次数成线性关系。为了更精确地计算时间复杂性，首先要定义 Ackermann函数$A(i, j)$和其倒数$\alpha(p, q)$：

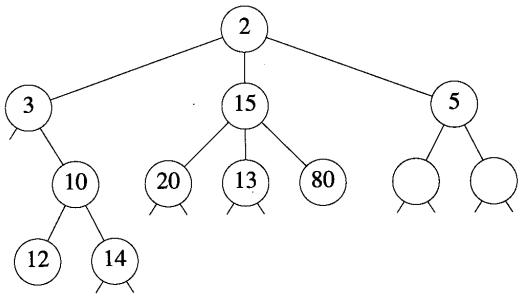


图8-21 路径分割

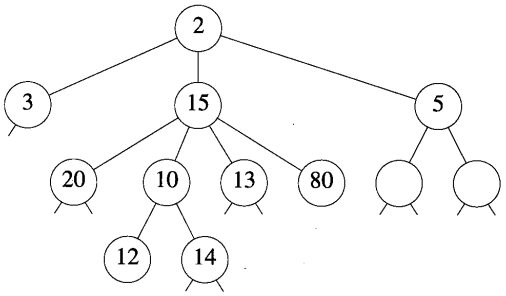


图8-22 路径对折

$A(l, j)=2j,$ $j\geq 1$

$A(i, l)=A(i\text{-}1, 2)$ $i\geq 2$

$A(i, j)=A(i\text{-}1, A(i, j\text{-}1))$ $i, j\geq 2$

$\alpha(p, q)=\min\{z \geq 1 \mid a(z, \lfloor p/q \rfloor) > \log_2 q\}$, $p\geq q\geq 1$

函数$A(i,j)$增长很快。相应地，α随着p和q的增长而缓慢地增长。实际上，对于$q<2^{16}=65\,536$且$p\geq q$，有$A(3,1)=16$，$\alpha(p,q)\leq 3$。但$A(4,1)$将是一个很大很大的数。在实际应用中，q是集合中元素的个数，p为$n+f$（f为查找的个数），总是有$\alpha(p,q)\leq 4$。

定理8-3 [Tarjan和Van Leeuwen] 设$T(f,u)$是交错的f次查找和u次合并操作所需的最大时间。假设$u\geq n/2$，则

$$k_1(n+f\alpha(f+n,n))\leq T(f,u)\leq k_2(n+f\alpha(f+n,n))$$

其中k_1和k_2为正常数。此定理适用于从单元素集合出发，采用重量或高度规则进行合并，同时按三种路径压缩方法的任意一种进行查找操作的情况。

定理8-3中，要求$u\geq n/2$并不是非常重要。因为当$u<n/2$时，某些元素在合并操作中并未涉及到。在合并和查找操作中，留在单元素集合中的元素，并不需要考虑，因为与这些元素有关的每一次操作可在O(1)时间内完成。虽然函数$\alpha(f, u)$增长很慢，但是合并/搜索的复杂性并不是与合并和查找的次数成线性关系。至于空间要求，每个元素需要一个节点即可。

练习

34. 画出图8-15a和b，图8-17a和b和图8-18a和b的二叉树描述。

35. 画出图8-12的二叉树描述。(注意这类树的二叉树描述方法与用左孩子指针指向一个节点，右孩子指针指向另一节点的描述方法不同)。

36. 森林（forest）是一棵或多棵树的集合。在树的二叉树描述中，根没有右孩子。由此可以用二叉树来描述有m棵树的森林。首先得到树林中每棵树的二叉树描述，然后，第i棵作为第i-1棵树的右子树，画出图8-15中有4棵树的森林的二叉树描述，同时还有图8-17和图8-18的二棵树的森林。

37. 设t为类BinanyTree的对象。假定t为分布树(如图8-14所示)的二叉树描述。给出计算t中每个节点的D和boost值的程序。程序应调用t.PostPrint( )输出得到的结果，用适当的分布树来检查程序的正确性。

38. 设有n个集合，每个集合中元素各不相同

1) 证明若执行u次合并操作，则所有集合中的元素数都小于等于u+1。

2) 证明在集合数目变成1之前，最多执行了n-1次合并操作。

3) 证明若执行的合并次数小于$[n/2]$，则至少有一个集合仅有一个元素。

4) 证明若执行了n次合并操作，则最少有$\max\{n-2u,0\}$个单元素集合。

39. 给出一个例子，由单元素集合出发执行一系列合并操作，使生成树的高度与定理 8-2给出的上限相等。假设每次合并均遵循重量规则。

40. 给出Union函数(见程序8-16)的另一个版本，采用高度规则而不是重量规则。

41. 当采用高度规则而非重量规则时证明定理8-2。

42. 给出一个例子，由单元素集合出发执行一系列合并操作，使生成树的高度等于定理 8-2给出的上限。假设每次合并均遵循高度规则。

43. 比较程序8-15和8-16的平均性能(用程序8-17的find函数代替程序8-16的find函数)。取n的不同值时进行比较。对于每个n值，产生一随机序偶(i, j)。用两个查找操作替换这个序偶(其中一个查找i，另一个查找j)。若这2个元素在不同集合中，则执行一次合并操作。使用多个不同的随机序偶重复进行实验。测试所有操作所需的总时间。自己设计实验的详细细节，设计一个有意义的实验来比较两组程序的平均性能。写出实验过程和结果报告，其中包括程序、平均时间表和图。

44. 重写程序8-16和8-17，采用类型为Node 的节点数组。Node 可定义为类，类中的每一实例均含有私有成员parent 和root。若数组为E[0:n+1]，则当且仅当e 为根节点时E[e].parent 是元素e 的父节点且E[e].root 为true。

45. 编写find 函数，采用路径分割而不是路径紧凑方法（程序8-17）。

46. 编写find 函数，采用路径对折而不是路径紧凑方法（程序8-17）。

8.11 参考及推荐读物

放置放大器的问题在下面论文中有深入研究： D.Paik, S.Reddy, S.Sahni. Deleting Vertices in Dags to Bound Path Lengths. *IEEE Transactions on Computers*, 43, 9, 1994, 1091~1096。还有D.Paik, S.Reddy. Heuristics for the Placement of Flip-Flops in Partial Scan Designs and for the Placement of Signal Boosters in Lossy Circuits. *Sixth International Conference On VLSI Design*, 1993, 45~50。

在线等价类问题的树形描述在下面论文中有详细的介绍： R.Tarjan, J.Leeuwen. Worst Case Analysis of Set Union Algorithms. *Journal of the ACM*, 31, 2, 1984, 245~281。