第 5 章　树和二叉树

树是一种非常重要的非线性数据结构。树由 $n(n\geqslant 0)$个数据元素组成，数据元素之间具有明显的层次结构。图 5-1 是树的树形图表示，由于它很像自然界中倒长的树，因此，被命名为“树”。树的树形图表示法规定在用直线连接起来的两端结点中，处在上端的结点是前驱，处在下端的结点是后继，如 A 是 B 的前驱，B 是 A 的后继。图 5-1 中树的逻辑结构可表示为 $T=(D, R)$，其中，数据元素集合 $D=\{A, B, C, D, E, F, G, H, I, J, K, L\}$，各数据元素之间的前后关系 $R=\{<A,B>, <A,C>, <A,D>, <B,E>, <B,F>, <C,G>, <D,H>, <F,I>, <H,J>, <H,K>, <H,L>\}$。

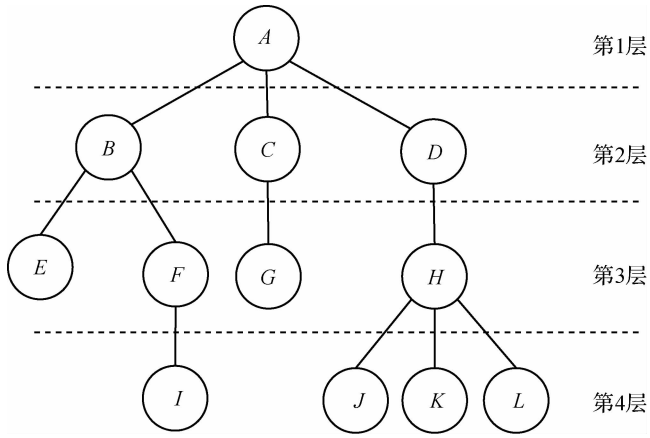


图 5-1　树的树形图表示

从图 5-1 可以很直观地看出树具有如下结构特性：

(1)只有最顶层的结点没有前驱，其余结点都有且只有一个前驱。

(2)一个结点可以没有后继，也可以有一个或多个后继。

在客观世界中，很多事物都可以用树这种数据结构来表示，图 5-2 是一个学校组织结构图。在计算机领域，树也有着非常广泛的应用，如用树形结构表示实体之间联系的层次模型

是最早用于商业数据库管理系统的数据模型;在操作系统中用树来表示文件目录结构等。

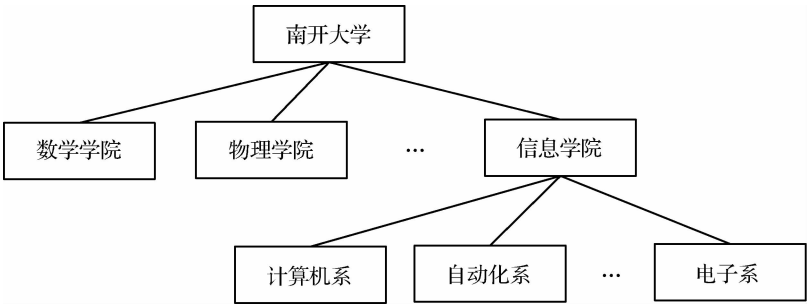


图 5-2　学校组织结构图

5.1　树的基本概念

前面通过图示给出了树形结构的直观说明,下面将以递归方式给出树的定义,介绍树的表示形式和树的基本术语。

5.1.1　树的定义

树是由 $n(n\geqslant 0)$个结点组成的有限集 T。当 $n=0$ 时,称为空树;当 $n>0$ 时,集合 T 须满足如下条件:

(1)有且仅有一个没有前驱的结点,该结点称为树的根结点。

(2)将根结点去除后,其余结点可分为 $m(m\geqslant 0)$个互不相交的子集 $T_1,T_2,\cdots,T_m$,其中,每个子集 $T_i(i=1,2,\cdots,m)$又是一棵树,并称其为根的子树。

可以看到,树的定义采用的是递归定义方式,即一棵非空的树由根结点和去除根结点后得到的若干棵规模更小的子树构成,而每一棵子树又是由该子树的根结点和去除该子树根结点后得到的若干棵更小的子树构成。例如,图 5-1 中的树可以看做是由根结点 A 和 3 棵子树 T_1,T_2,T_3(结点集合分别为 $D_1=\{B,E,F,I\}$、$D_2=\{C,G\}$和 $D_3=\{D,H,J,K,L\}$)构成;子树 T_1 又可以看做是由根结点 B 和两棵子树 T_{11},T_{12}(结点集合分别为 $D_{11}=\{E\}$,$D_{12}=\{F,I\}$)构成。

5.1.2　树的表示形式

树形图表示法因其直观性强而成为树的最常用的表示形式。除树形图表示法之外,树还有 3 种常用的表示形式,下面一一介绍。

1)嵌套集合表示法

嵌套集合表示法是通过集合包含的形式体现结点之间的关系的,后继结点集合包含在前

驱结点集合中。例如，采用嵌套集合表示法可将图 5-1 所示的树表示为图 5-3 所示的形式。

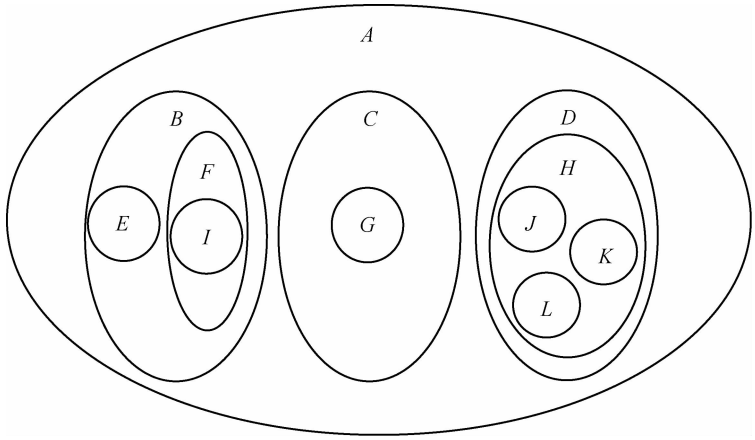


图 5-3　树的嵌套集合表示

2)凹入表表示法

凹入表表示法是利用书的目录形式表示结点之间的关系，后继结点位于前驱结点的下一层目录中。例如，采用凹入表表示法可将图 5-1 所示的树表示为图 5-4 所示的形式。

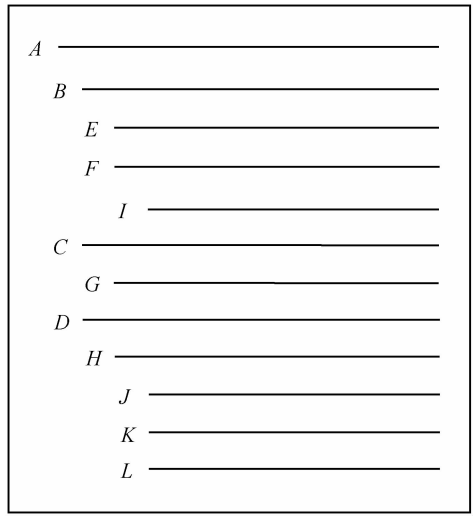


图 5-4　树的凹入表表示

3)广义表表示法

广义表表示法是利用广义表(广义表的相关知识请参考其他书籍)的多层次结构来表示树，后继结点位于前驱结点的下一层次。例如，采用广义表表示法可将图 5-1 所示的树表示为图 5-5 所示的形式。

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

图 5-5　树的广义表表示

5.1.3 树的基本术语

下面介绍树的基本术语。

1)度和树的度

一个结点的后继的数目称为该结点的度,树中各结点度的最大值称为树的度。例如,在图 5-1 中,结点 A 和 H 的度都为 3,结点 B 的度为 2,结点 C,D,F 的度都为 1,其余结点的度都为 0。树中所有结点度的最大值为结点 A 和结点 H 所具有的度 3,因此,该树的度为 3。

2)结点的层和树的深度

树的根结点所在的层为第一层,其余结点的层等于其前驱结点的层加 1,树中各结点的层的最大值称为树的深度。例如,在图 5-1 中标出了各结点的层,该树的深度为 4。

3)分支、路径、路径长度和树的路径长度

从一个结点到其后继结点之间的连线称为一个分支。从一个结点 X 到另一个结点 Y 所经历的所有分支构成结点 X 到结点 Y 的路径。一条路径上的分支数目称为路径长度,从树的根结点到其他各个结点的路径长度之和称为树的路径长度。例如,在图 5-1 中,结点 A 到结点 H 的路径为 $A\rightarrow D\rightarrow H$,路径长度为 2;结点 C 到结点 H 的路径不存在;树的路径长度为 23。

4)叶子结点、分支结点和内部结点

树中度为 0 的结点称为叶子结点(或终端结点),度不为 0 的结点称为分支结点(或非终端结点),除根结点以外的分支结点也称为内部结点。例如,在图 5-1 中,结点 E,G,I,J,K,L 是叶子结点,结点 A,B,C,D,F,H 是分支结点,结点 B,C,D,F,H 是内部结点。

5)结点间的关系

在树中,一个结点的后继结点称为该结点的孩子,相应地,一个结点的前驱结点称为该结点的双亲,即一个结点是其孩子结点的双亲、其双亲结点的孩子。例如,在图 5-1 中,结点 A 是结点 B,C,D 的双亲,结点 B,C,D 是结点 A 的孩子。

同一双亲的孩子结点之间互称为兄弟,不同双亲但在同一层的结点之间互称为堂兄弟。例如,在图 5-1 中,结点 B,C,D 互为兄弟,结点 E,F 也互为兄弟,结点 F,G,H 互为堂兄弟,结点 E,G,H 也互为堂兄弟。

从树的根结点到某一个结点 X 的路径上经历的所有结点(包括根结点但不包括结点 X)称为结点 X 的祖先,以某一结点 X 为根的子树上的所有非根结点(除结点 X 外)称为结点 X 的子孙。例如,在图 5-1 中,结点 B 是结点 E,F,I 的祖先,结点 E 是结点 A,B 的子孙。

6)有序树和无序树

对于树中的任一结点,如果其各棵子树的相对次序被用来表示数据之间的关系,即交换子树位置会改变树所示的内容,则称该树为有序树;否则称为无序树。例如,对于图 5-6 所示的两棵树,如果将其作为有序树,则它们会因根结点 A 的子树顺序的不同而表示不同的内容,

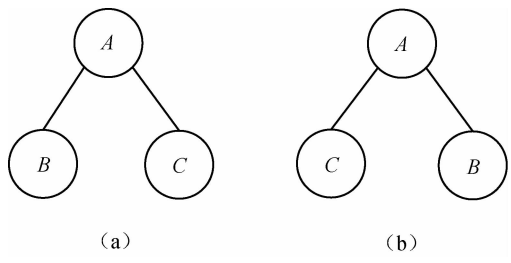


图 5-6 有序树和无序树举例

因此它们是两棵不同的树；如果将其作为无序树，则根结点 A 的子树顺序的不同不会影响树所表示的内容，因此它们是两棵相同的树。

7）森林

$m(m\geqslant 0)$棵互不相交的树的集合就构成了森林。显然，将一棵树的根结点删除就可以得到由根结点的子树组成的森林；将森林中的树用一个根结点连起来就可以得到一棵树。例如，在图 5-1 中，将根结点 A 删除，则可以得到由根结点的 3 棵子树组成的森林；将森林中的这 3 棵树再用一个根结点重新连起来，则森林又变成了一棵树。

5.2　二叉树及其基本性质

二叉树是一种特殊的树形结构，在实际应用中有着十分重要的意义。例如，在通信、数据压缩等领域有着广泛应用的哈夫曼树就是采用二叉树的结构，在数据库中可以选择使用二叉树结构管理数据等。这里主要介绍二叉树的定义和一些基本性质。

5.2.1　二叉树的定义

这里先给出一般二叉树的定义，再介绍两种特殊形式的二叉树：满二叉树和完全二叉树。

1）二叉树

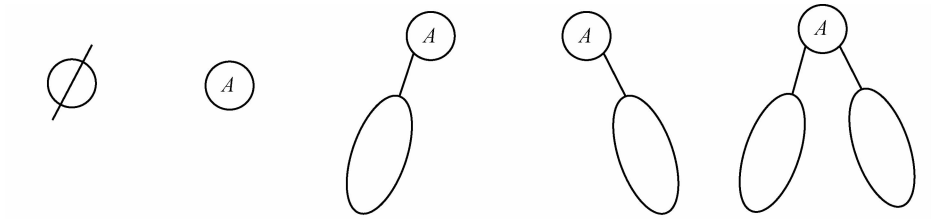
与树一样，二叉树的定义也采用递归定义方式。

二叉树是由 $n(n\geqslant 0)$个结点组成的有限集 T。当 $n=0$ 时，称为空二叉树；当 $n>0$ 时，集合 T 须满足如下条件：

(1)有且仅有一个没有前驱的结点，该结点称为二叉树的根结点。

(2)将根结点去除后，其余结点可分为两个互不相交的子集 T_1，T_2，其中每个子集 $T_i(i=1, 2)$又是一棵二叉树，并分别称为根结点的左子树和右子树。

可见，二叉树就是每个结点的度小于等于 2 的有序树。因此，二叉树共有 5 种基本形态，如图 5-7 所示。



(a) 空二叉树　(b) 只有根结点　(c) 右子树为空　(d) 左子树为空　(e) 左、右子树非空

图 5-7　二叉树的基本形态

对二叉树中的结点可以按照“自上而下、自左至右”的顺序进行连续编号(称为顺序编号法),即将其根结点编号为1;除第一层外其余各层第一个结点的编号等于上一层最后一个结点的编号加1。图5-8所示就是3棵深度为3、采用顺序编号法表示的二叉树。

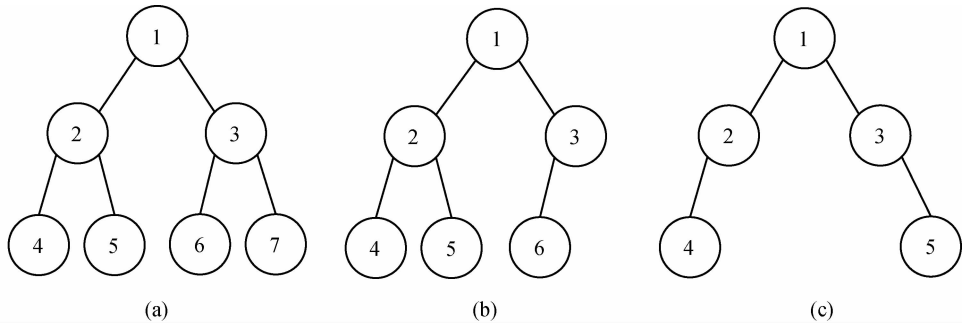


图5-8　采用顺序编号法表示的树

2)满二叉树和完全二叉树

满二叉树和完全二叉树是两种特殊形态的二叉树。

满二叉树是指除了最后一层的结点为叶子结点外,其他结点都有左、右两棵子树的二叉树。例如,图5-8(a)表示一棵满二叉树,而图5-8(b)中的结点3缺少右子树,图5-8(c)中的结点2缺少右子树、结点3缺少左子树,因此它们都不是满二叉树。

完全二叉树是指其结点与相同深度的满二叉树中的结点编号完全一致的二叉树。对于深度为k的完全二叉树,其前$k-1$层与深度为k的满二叉树的前$k-1$层完全一样,只是在第k层上有可能缺少右边若干个结点。显然,满二叉树必然是完全二叉树,而完全二叉树不一定是满二叉树。例如,图5-8(a)既是满二叉树也是完全二叉树,图5-8(b)与图5-8(a)中的满二叉树相比只是在最后一层上缺少右边的一个结点,因此是一棵完全二叉树;而图5-8(c)则不是完全二叉树。

5.2.2　二叉树的基本性质

二叉树具有以下几个基本性质。

性质1:在二叉树的第i层上至多有2^{i-1}个结点($i\geqslant 1$)。

证明:用数学归纳法证明。

(1)归纳基础:当$i=1$时,第1层上至多有$2^{i-1}=1$个结点,即根结点,命题成立。

(2)归纳假设:假设“在第$j(1\leqslant j<i)$层上至多有2^{i-1}个结点”成立。

(3)归纳步骤:根据归纳假设,第$i-1$层至多有$2^{(i-1)-1}$个结点。由于二叉树中每个结点至多有两个孩子结点,因此,第i层上的结点数至多是第$i-1$层上最大结点数的两倍,即$2^{(i-1)-1}*2=2^{i-1}$个。证毕。

性质2:深度为k的二叉树至多有2^k-1个结点。

证明:二叉树的最大结点数即为每一层最大结点数之和。

由性质1可知,深度为k的二叉树的最大结点数为$2^0+2^1+\cdots+2^{k-1}=2^k-1$。证毕。

显然,若深度为k的二叉树具有2^k-1个结点,即每一层的结点数都达到最大,则该二

叉树就是一棵满二叉树。

性质 3：在二叉树中，若度为 0 的结点（即叶子结点）数为 n_0，度为 2 的结点数为 n_2，则 $n_0=n_2+1$。

证明：设二叉树的结点总数为 n，度为 1 的结点数为 n_1。

二叉树可以看做由三类具有不同度的结点组成，即 $n=n_2+n_1+n_0$。

另一方面，除了根结点外每一个结点都是某一结点的孩子结点，因此二叉树又可以看做是由孩子结点和根结点组成，每一个度为 $i(i=0,1,2)$ 的结点产生 i 个孩子结点，即 $n=(2n_2+1n_1+0n_0)+1=2n_2+n_1+1$。

因此有 $n_2+n_1+n_0=2n_2+n_1+1$，即 $n_0=n_2+1$。证毕。

性质 4：具有 n 个结点的完全二叉树其深度为 $\lfloor\log_2 n\rfloor+1$（其中 $\lfloor\log_2 n\rfloor$ 表示不大于 $\log_2 n$ 的最大整数）。

证明：设完全二叉树的深度为 k，则根据完全二叉树的定义可知其结点数 n 大于深度为 $k-1$ 的满二叉树的结点数、小于等于深度为 k 的满二叉树的结点数。

再由性质 2 可得：$2^{k-1}-1<n\leqslant 2^k-1$，即 $2^{k-1}\leqslant n<2^k$。

两边同时取以 2 为底的对数，可得：$k-1\leqslant\log_2 n<k$，即 $k-1=\lfloor\log_2 n\rfloor$ 或 $k=\lfloor\log_2 n\rfloor+1$。证毕。

性质 5：采用顺序编号的完全二叉树具有如下性质：

(1)若一个分支结点的编号为 i，则其左子树的根结点（即左孩子结点）编号为 $2*i$，右子树的根结点（即右孩子结点）编号为 $2*i+1$；

(2)反之，若一个非根结点的编号为 i，则其双亲结点的编号为 $\lfloor i/2\rfloor$（其中 $\lfloor i/2\rfloor$ 表示不大于 $i/2$ 的最大整数）。

证明：先证明(1)。

设结点 i 位于第 k 层上，由性质 2 可知前 $k-1$ 层上的结点总数为 $2^{k-1}-1$，因此第 k 层上位于结点 i 前的结点数为 $i-1-(2^{k-1}-1)$。

若结点 i 有孩子结点，则结点 $j(1\leqslant j<i)$ 有两个孩子结点，因此，在第 $k+1$ 层上位于结点 i 的左孩子结点前的结点数为 $2*[i-1-(2^{k-1}-1)]=2*(i-2^{k-1})$，位于其右孩子结点前的结点数为 $2*[i-1-(2^{k-1}-1)]+1=2*(i-2^{k-1})+1$。

结点 i 的孩子结点位于第 $k+1$ 层上，由性质 2 可知前 k 层上的结点总数为 2^k-1。因此，结点 i 的左孩子结点编号为 $2^k-1+2*[i-1-(2^{k-1}-1)]+1=2*i$，右孩子结点编号为 $2^k-1+2*[i-1-(2^{k-1}-1)]+2=2*i+1$。

再由(1)证明(2)。

设非根结点 i 的双亲结点编号为 j。

若结点 i 为结点 j 的左孩子结点，则由(1)有 $i=2*j$，即 $j=i/2=\lfloor i/2\rfloor$。

若结点 i 为结点 j 的右孩子结点，则由(1)有 $i=2*j+1$，即 $j=(i-1)/2=\lfloor i/2\rfloor$。

证毕。

注意：符号 $\lfloor x\rfloor$ 表示不大于 x 的最大整数，反之，$\lceil x\rceil$ 表示不小于 x 的最小整数。

5.3 二叉树的抽象数据类型和表示方式

同线性表一样,二叉树中的每个结点既可以存储单值元素,又可以存储记录型元素。二叉树一般需要进行下面的基本操作:

(1)创建一棵空二叉树。
(2)删除一棵二叉树。
(3)先序遍历二叉树。
(4)中序遍历二叉树。
(5)后序遍历二叉树。
(6)逐层遍历二叉树。
(7)判断二叉树是否为空。
(8)清空二叉树。
(9)以指定元素值创建根结点。
(10)将一个结点作为指定结点的左孩子插入。
(11)将一个结点作为指定结点的右孩子插入。
(12)删除以指定结点为根的子树。
(13)按关键字查找结点。
(14)修改指定结点的元素值。
(15)获取指定结点的双亲结点。
(16)计算二叉树的深度。
(17)计算二叉树的叶子结点数。

下面是二叉树的抽象数据类型描述:

```
ADT BinTree
{
    Data:
        //具有二叉树形结构的0个或多个相同类型数据元素的集合
    Operations:
        BinTree();                  //创建空二叉树
        ~BinTree();                 //删除二叉树
        PreOrderTraverse();         //先序遍历
        InOrderTraverse();          //中序遍历
        PostOrderTraverse();        //后序遍历
        LevelOrderTraverse();       //逐层遍历
        IsEmpty();                  //判断二叉树是否为空
        CreateRoot();               //以指定元素值创建根结点
        Clear();                    //清空二叉树
```

```
    InsertLeftChild();          //将一个结点作为指定结点的左孩子插入
    InsertRightChild();         //将一个结点作为指定结点的右孩子插入
    DeleteSubTree();            //删除以指定结点为根的子树
    SearchByKey();              //按关键字查找结点
    ModifyNodeValue();          //修改指定结点的元素值
    GetParent();                //获取指定结点的双亲结点
} ADT BinTree
```

在计算机中存储二叉树的方法主要有两种:顺序表示法和链式表示法。下面分别介绍这两种表示方法并给出每种表示方法的具体实现。

5.3.1　二叉树的顺序表示及实现

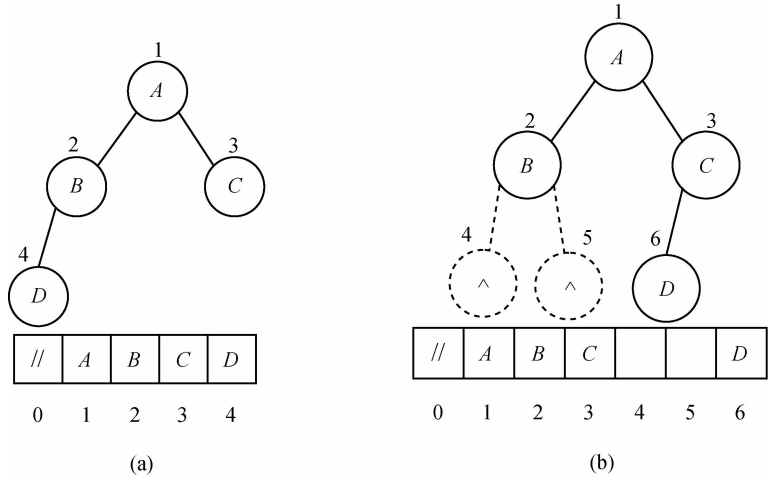
二叉树的顺序表示法操作方便,但缺点是容易造成存储空间的浪费。

1)二叉树的顺序表示

把二叉树的结点按完全二叉树的编号规则(5.2 节中的性质 5)自上而下、从左至右依次存放在一组地址连续的存储单元里就构成了二叉树的顺序存储,树中结点的编号就可以唯一地反映出结点之间的逻辑关系。通常通过定义一个一维数组来表示二叉树的顺序存储空间,为了使数组元素的下标值与其对应的结点编号一致,将下标为 0 的空间空闲不用或者用做其他用途。

例如,图 5-9 中给出了 3 种不同形式的二叉树,在其顺序表示中,根结点编号为 1,其余结点编号可以根据 5.2 节中的性质 5 计算得到。可以看出,在采用顺序表示时,除下标为 0 的空间外,图 5-9(a)中的完全二叉树没有其他空闲空间,空间利用率很高;图 5-9(b)和图 5-9(c)中的非完全二叉树,由于需要增添一些并不存在的空结点,所以有不同程度的空间浪费;尤其是图 5-9(c)中的右单分支二叉树(即二叉树中每个分支结点只有右孩子没有左孩子)空间利用率最低,可以证明深度为 k 且只有 k 个结点的右单分支二叉树需要 2^k-1 个结点的存储空间,空间利用率仅为 $k/(2^k-1)$。

因此,二叉树顺序表示适用于完全二叉树而不适用于非完全二叉树。



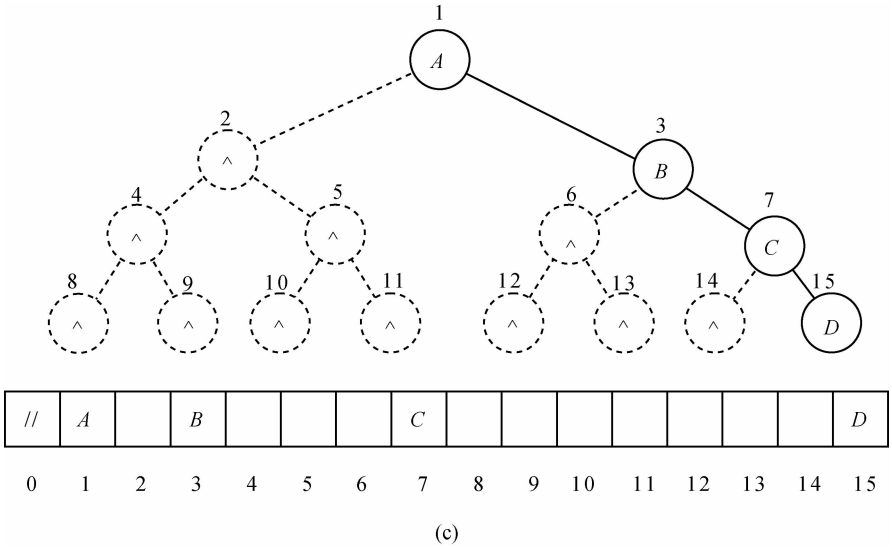


图 5-9　3 种不同形式的二叉树及其顺序表示

2)二叉树顺序表示的实现

由于顺序表示非完全二叉树时空间利用率较低,因此,二叉树的顺序表示在实际问题中应用不多。下面主要围绕二叉树的创建和删除这两个最基本的操作介绍二叉树顺序表示的实现。

【描述 5-1】　二叉树顺序表示的类模板 ArrayBinTree 的描述。

分析:在类模板中,需设置几个成员变量:一维数组 m_pElement 用于存储二叉树中各结点的值,数组类型由二叉树结点的值的类型决定;整型变量 m_nMaxSize 用于存储二叉树的最大结点数;一个 bool 型一维数组 m_pbUsed 用于表示结点状态,若某个结点不存在(见图 5-9b 中的结点 4 和结点 5),则 m_pbUsed 中相应元素的值为 false,否则相应元素的值为 true。

参考程序如下:

```
#include<iostream>
#include<assert.h>
using namespace std;
template<class T>
class ArrayBinTree
{
public:
    ArrayBinTree(int nDepth);          //创建空二叉树
    ~ArrayBinTree();                   //删除二叉树
    void CreateRoot(const T &x);       //以指定元素值创建根结点
    void Clear();                      //清空二叉树
    //将一个结点作为指定结点的左孩子插入
```

```
    bool InsertLeftChild(int nIndex,const T &x);
    //将一个结点作为指定结点的右孩子插入
    bool InsertRightChild(int nIndex,const T &x);
    void DeleteSubTree(int nIndex);   //删除以指定结点为根的子树
    void LevelOrderTraverse();        //逐层遍历
private:
    bool *m_pbUsed;                   //一维动态数组,保存每个结点中是否有值
    int m_nMaxSize;                   //树的最大结点数
    T *m_pElement;                    //一维动态数组,保存每个结点的值
};
```

下面对【描述 5-1】类模板中的成员函数给出具体实现。

【描述 5-2】 【描述 5-1】类模板中成员函数的实现。

```
//实现构造函数
template<class T>
ArrayBinTree <T>:: ArrayBinTree(int nDepth)
{
    int nI;
    assert(nDepth>0);  //树的深度必须大于 0
    //根据树的深度计算最大结点数
    m_nMaxSize=1;
    for(nI=0;nI<nDepth;nI++)
        m_nMaxSize *= 2;
    //根据最大结点数分配内存空间
    m_pElement=new T[m_nMaxSize];
    assert(m_pElement);
    m_pbUsed=new bool[m_nMaxSize];
    assert(m_pbUsed);
    //初始时所有结点中均没有值,即为一棵空树
    for(nI=0;nI<m_nMaxSize;nI++)
        m_pbUsed[nI]=false;
}
//实现析构函数
template<class T>
ArrayBinTree<T>::~ArrayBinTree()
{
    //释放内存
    if(m_pElement)  delete []m_pElement;
    if(m_pbUsed)  delete []m_pbUsed;
}
```

```
//实现以指定元素值创建根结点
template<class T>
void ArrayBinTree<T>::CreateRoot(const T &x)
{
    m_pElement[1]= x;
    m_pbUsed[1]=true;
}
//实现清空二叉树
template<class T>
void ArrayBinTree<T>::Clear()
{
    int nI;
    //将所有结点设置为没有值的状态
    for(nI=1;nI<m_nMaxSize;nI++)
        m_pbUsed[nI]=false;
}
//实现将一个结点作为指定结点的左孩子插入
template<class T>
bool ArrayBinTree<T>::InsertLeftChild(int nIndex, const T &x)
{
    int nChildIndex=2*nIndex;     //计算左孩子结点在数组中的位置
    if(nChildIndex >=m_nMaxSize) //左孩子结点所在位置不得超过最大结点数
        return false;
    //插入左孩子结点
    m_pElement[nChildIndex]=x;
    m_pbUsed[nChildIndex]=true;
  return true;
}
//实现将一个结点作为指定结点的右孩子插入
template<class T>
bool ArrayBinTree<T>::InsertRightChild(int nIndex, const T &x)
{
    int nChildIndex=2*nIndex+1; //计算右孩子结点在数组中的位置
    if(nChildIndex >=m_nMaxSize) //右孩子结点所在位置不得超过最大结点数
        return false;
    //插入右孩子结点
    m_pElement[nChildIndex]=x;
    m_pbUsed[nChildIndex]=true;
    return true;
```

```
}
//实现删除以指定结点为根的子树
template<class T>
void ArrayBinTree<T>::DeleteSubTree(int nIndex)
{
    int nLeftChildIndex=2*nIndex;          //获取左孩子结点在数组中的位置
    int nRightChildIndex=nLeftChildIndex+1;  //获取右孩子结点在数组中的位置
    assert(nIndex>0&&nIndex<m_nMaxSize);
                //待删除子树根结点必须在有效位置上
    m_pbUsed[nIndex]=false;                //将根结点置为没有值的状态
    if(nLeftChildIndex< m_nMaxSize)        //递归删除左子树
       DeleteSubTree(nLeftChildIndex);
    if(nRightChildIndex< m_nMaxSize)       //递归删除右子树
       DeleteSubTree(nRightChildIndex);
}
//实现逐层遍历(即从根结点开始,按照自上而下、从左到右的顺序访问结点)
template<class T>
void ArrayBinTree<T>::LevelOrderTraverse()
{
    int nI,nNodeNum=0;
    //按照自上而下、从左到右的顺序输出各结点的值
    for(nI=1;nI<m_nMaxSize;nI++)
  {
        if(m_pbUsed[nI])
        {
            cout<<nI<<": "<<m_pElement[nI]<<endl;
            nNodeNum++;
        }
    }
    //若二叉树中没有结点,则输出"空二叉树"
    if(nNodeNum==0)
        cout<<"空二叉树"<<endl;
}
```

提示:将【描述 5-1】中关于二叉树顺序表示类模板的声明代码和【描述 5-2】中关于二叉树顺序表示类模板的实现代码一起存储在 ArrayBinTree.h 文件中,以后就可以基于该类模板快速完成顺序二叉树相关应用问题的求解。

3)二叉树顺序表示代码复用示例

【例 5-1】 基于二叉树顺序表示的 C++ 实现代码,完成如下操作:

(1)创建如图 5-9(b)所示的二叉树,其中每一个结点保存一个字符信息,并通过逐层遍

历输出各结点的值。

(2)将以结点 C 为根结点的子树删除,并通过逐层遍历输出各结点的值。

(3)清空二叉树中的元素。

分析:对于由简单数据元素构成的二叉树,一般可以直接使用 C++语言提供的基本数据类型来描述数据元素,本例中可直接使用 char 类型。

参考程序如下:

```
/* Chap5-1.cpp:二叉树顺序存储结构的应用问题*/
#include "ArrayBinTree.h"
int main()
{
    ArrayBinTree<char> btree(3);  //声明以 char 为数据元素的 3 层二叉树
    btree.CreateRoot('A');
    btree.InsertLeftChild(1, 'B');
    btree.InsertRightChild(1, 'C');
    btree.InsertLeftChild(3, 'D');
    cout<<"当前二叉树的元素为:\n";
    btree.LevelOrderTraverse();
    btree.DeleteSubTree(3);
    cout<<"当前二叉树的元素为:\n";
    btree.LevelOrderTraverse();
    btree.Clear();
    cout<<"当前二叉树的元素为:\n";
    btree.LevelOrderTraverse();
    return 0;
}
```

程序的运行结果如下:

```
当前二叉树的元素为:
1:A
2:B
3:C
6:D
当前二叉树的元素为:
1:A
2:B
当前二叉树的元素为:
空二叉树
```

5.3.2　二叉树的链式表示及实现

与顺序表示相比，链式表示通常具有更高的空间利用率，因此在实际应用中一般会使用链式表示来存储二叉树。下面介绍二叉树的链式表示及其具体实现。

1)二叉树的链式表示

在二叉树的链式表示中，结点之间的关系通过指针来体现。根据一个结点中指针域数量的不同，二叉树的链式表示又可以分为二叉链表表示和三叉链表表示。

图 5-10 是一个用二叉链表表示法存储二叉树的例子。在二叉链表中，双亲结点有指向其孩子结点的指针，而孩子结点不包含指向其双亲结点的指针。由于二叉树中每个结点最多有两个孩子，因此在一个结点中设置两个指针域 leftchild 和 rightchild，分别指向其左孩子和右孩子，数据域 data 用于存放每个结点中数据元素的值。如果一个结点没有左孩子，则其 leftchild 指针为空(用 NULL 或 0 表示)；如果一个结点没有右孩子，则其 rightchild 指针为空(用 NULL 或 0 表示)。

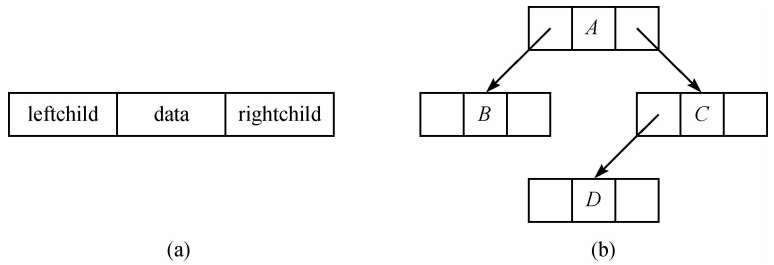


图 5-10　二叉树的二叉链表存储结构

图 5-11 是一个用三叉链表表示法存储二叉树的例子。在三叉链表中，双亲结点有指向其孩子结点的指针，而孩子结点也包含指向其双亲结点的指针。因此，在用三叉链表表示的二叉树的每个结点中，除了具有二叉链表中的两个指向孩子结点的指针域 leftchild 和 rightchild 外，还有一个指向双亲结点的指针域 parent。根结点没有双亲，所以它的 parent 指针为空(用 NULL 或 0 表示)。

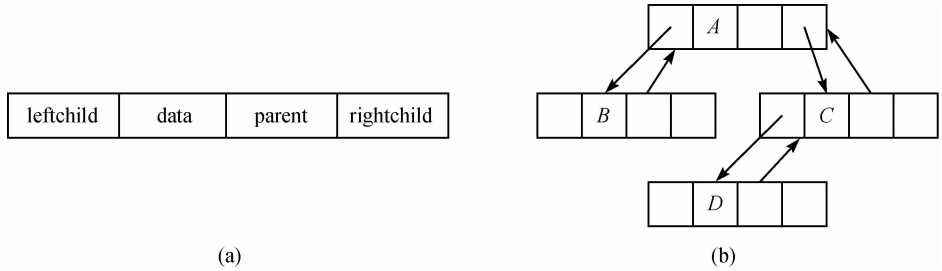


图 5-11　二叉树的三叉链表存储结构

与顺序表示相比，链式表示需要在每个结点中增加额外的指针域来表示结点间的关系，空间利用率似乎更低。但在实际应用中，一个结点数据域所占据的空间一般要远大于指针域所占据的空间，所以即便有指针域的额外开销，链式表示通常也比顺序表示具有更高的空

间利用率。

2)二叉树链式表示的实现

二叉链表表示是二叉树最常用的存储结构,这里以二叉链表为例,围绕二叉树的创建介绍二叉树链式表示的实现,其他常用操作的实现将在后面给出。

下面将二叉树二叉链表表示的类模板命名为 LinkedBinTree。

【描述 5-3】 二叉树二叉链表表示的类模板描述。

分析:与线性表的链式表示相同,在二叉树的链式表示中,应先定义结点类模板,该结点类模板中除了要存放结点的值,还要存放指向其他结点的指针。在二叉链表表示中,每个结点中都有从双亲结点指向其孩子结点的指针,因此,在二叉树类模板中只要设置一个指向根结点的指针 m_pRoot,即可从根结点出发,访问到其他结点。

参考程序如下:

```
#include<iostream>
#include "LinkQueue.h"          //3.4.2 中实现的链接队列类模板
#include "LinkStack.h"          //3.2.3 中实现的链接栈类模板
#include<assert.h>
using namespace std;
//结点类模板
template<class T>
class LinkedNode
{
    template<class T>
    friend class LinkedBinTree;
public:
    LinkedNode()  //构造函数
    {
        m_pLeftChild=m_pRightChild=NULL;
    }
    LinkedNode(const T &x)  //构造函数
    {
        m_pLeftChild=m_pRightChild=NULL;
        m_data = x;
    }
private:
    T m_data;
    LinkedNode<T> *m_pLeftChild, *m_pRightChild;
};
//二叉树的二叉链表表示类模板
template<class T>
class LinkedBinTree
```

```
{
public:
    LinkedBinTree();          //创建空二叉树
    ~LinkedBinTree();         //删除二叉树
    bool IsEmpty();  //判断二叉树是否为空
    LinkedNode<T> * CreateRoot(const T &x);  //以指定元素值创建根结点
    void Clear();  //清空二叉树
    LinkedNode<T> * GetRoot();  //获取根结点
    //将一个结点作为指定结点的左孩子插入
    LinkedNode<T> * InsertLeftChild(LinkedNode<T> * pNode,const T &x);
    //将一个结点作为指定结点的右孩子插入
    LinkedNode<T> * InsertRightChild(LinkedNode<T> * pNode,const T &x);
    //修改指定结点的元素值
    bool ModifyNodeValue(LinkedNode<T> * pNode,const T &x);
    //获取指定结点的元素值
    bool GetNodeValue(LinkedNode<T> * pNode,T &x);
    //获取指定结点的左孩子结点
    LinkedNode<T> * GetLeftChild(LinkedNode<T> * pNode);
    //获取指定结点的右孩子结点
    LinkedNode<T> * GetRightChild(LinkedNode<T> * pNode);
    void PreOrderTraverse(LinkedNode<T> * pNode);  //按递归方式先序遍历
    void InOrderTraverse(LinkedNode<T> * pNode);  //按递归方式中序遍历
    void PostOrderTraverse(LinkedNode<T> * pNode);  //按递归方式后序遍历
    void PreOrderTraverse();          //按非递归方式先序遍历
    void InOrderTraverse();           //按非递归方式中序遍历
    void PostOrderTraverse();         //按非递归方式后序遍历
    void LevelOrderTraverse();        //按非递归方式逐层遍历
    //按非递归方式获取指定结点的双亲结点
    LinkedNode<T> * GetParent(LinkedNode<T> * pNode);
    //删除以指定结点为根的子树
    void DeleteSubTree(LinkedNode<T> * pNode);
    //由 DeleteSubTree 函数调用按非递归方式删除以指定结点为根的子树
    void DeleteSubTreeNode(LinkedNode<T> * pNode);
    //按非递归方式根据关键字查找结点
    LinkedNode<T> * SearchByKey(const T &x);
private:
    LinkedNode<T> * m_pRoot;   //指向根结点的指针
};
```

提示：由于本程序中同时使用栈和队列，为了区分栈和队列的结点类，将栈和队列的结

点类分别重新命名为 StackNode 和 QueueNode。

下面对【描述 5-3】类模板中的部分成员函数给出具体实现。

【描述 5-4】 【描述 5-3】类模板中部分成员函数的实现。

```
//实现创建空二叉树
template<class T>
LinkedBinTree<T>::LinkedBinTree()
{
    m_pRoot=NULL;  //将指向根结点的指针置为空
}
//实现以指定元素值创建根结点
template<class T>
LinkedNode<T> * LinkedBinTree<T>::CreateRoot(const T &x)
{
    //如果二叉树中原来存在结点,则将其清空
    if(m_pRoot!= NULL)  //若原先存在根结点,则直接将根结点的值置为 x
    m_pRoot->m_data=x;
    else                    //否则,创建一个新结点作为根结点
        m_pRoot=new LinkedNode<T>(x);
    return m_pRoot;
}
//将一个结点作为指定结点的左孩子插入
template<class T>
LinkedNode<T> * LinkedBinTree<T>::InsertLeftChild(LinkedNode<T> *
pNode,const T &x)
{
    LinkedNode<T> *pNewNode;
    //对传入参数进行有效性判断
    if (pNode==NULL)
        return NULL;
    //创建一个新结点
    pNewNode = new LinkedNode<T>(x);
    if (pNewNode==NULL)  //若分配内存失败
        return NULL;
    //将新结点作为 pNode 的左孩子(即将结点中的左孩子指针指向新结点)
    pNode->m_pLeftChild=pNewNode;
    return pNewNode;
}
//将一个结点作为指定结点的右孩子插入
template<class T>
```

```
LinkedNode<T> * LinkedBinTree<T>::InsertRightChild(LinkedNode<T> *
pNode,const T &x)
{
    LinkedNode<T> *pNewNode;
    if (pNode==NULL)  //对传入参数进行有效性判断
        return NULL;
    //创建一个新结点
    pNewNode=new LinkedNode<T>(x);
    if(pNewNode==NULL)  //若分配内存失败
        return NULL;
    //将新结点作为 pNode 的右孩子(即将结点中的右孩子指针指向新结点)
    pNode->m_pRightChild=pNewNode;
    return pNewNode;
}
```

提示:将【描述 5-3】中的声明代码和【描述 5-4】中的实现代码(【描述 5-4】中只给出了部分成员函数的实现,应将那些没有给出实现的函数声明注释以避免编译报错)一起存储在 LinkedBinTree.h 文件中。

3)二叉树链式表示代码复用示例

【例 5-2】 基于二叉树二叉链表表示的 C++实现代码,创建如图 5-12 所示的二叉树,其中每一个结点保存一个字符信息。

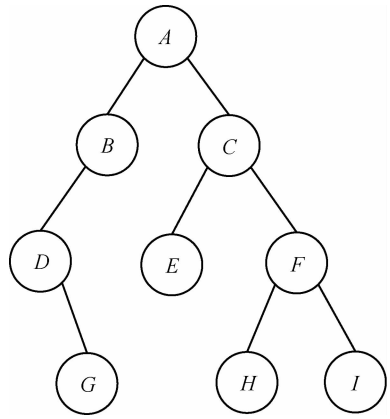
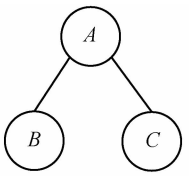


图 5-12　二叉树

分析:创建二叉树的方式有多种,这里采用基于队列的层次创建方式。其创建步骤如图 5-13 所示。

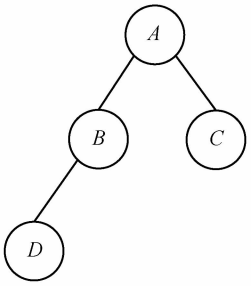
操作：创建根结点A，并将其入队。
队列元素：结点A。

(a)



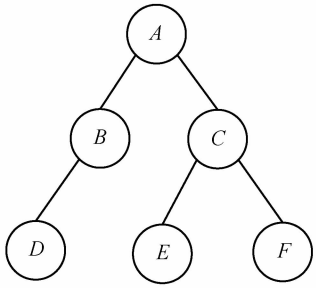
操作：将队头元素结点A 出队，创建其孩子结点B 和C 并入队。
队列元素：结点B，C。

(b)



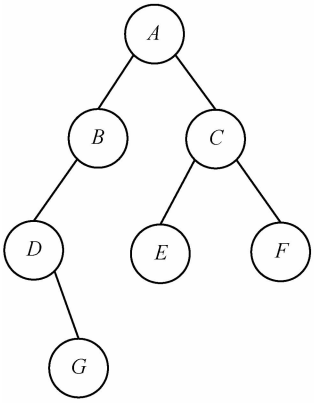
操作：将队头元素B出队，创建其左孩子结点D并入队。
队列元素：结点C，D。

(c)



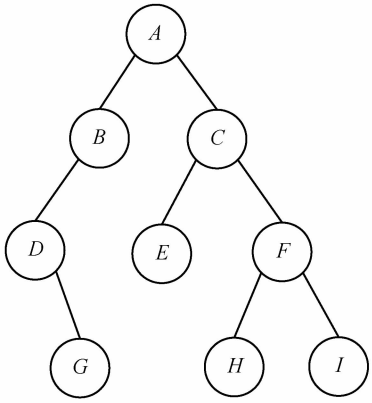
操作：将队头元素C出队，创建其孩子结点E和F并入队。
队列元素：结点D，E，F。

(d)



操作：将队头元素D出队，创建其右孩子结点G并入队；将队头元素E出队，其无孩子结点。
队列元素：结点F，G。

(e)



操作：将队头元素F出队，创建其孩子结点H和I并入队；依次将G，H，I出队，它们均无孩子结点。
队列元素：空。

(f)

图 5-13　按层次创建二叉树

参考程序如下：

```
/* Chap5-2.cpp:创建链式存储结构的二叉树 */
#include "LinkedBinTree.h"
int main()
{
```

```
LinkedBinTree<char> btree;     //声明以 char 为数据元素的二叉树
LinkedNode<char> *pNode=NULL, *pNewNode=NULL;
LinkQueue<LinkedNode<char>*> q;
//按层次创建二叉树
//创建根结点,并将根结点入队
pNewNode=btree.CreateRoot('A');
q.Insert(pNewNode);
//将队头元素 A 出队,创建 A 的孩子结点 B 和 C,并将创建的孩子结点入队
q.Delete(pNode);
pNewNode=btree.InsertLeftChild(pNode,'B');
q.Insert(pNewNode);
pNewNode=btree.InsertRightChild(pNode,'C');
q.Insert(pNewNode);
//将队头元素 B 出队,创建 B 的左孩子结点 D,并将创建的孩子结点入队
q.Delete(pNode);
pNewNode=btree.InsertLeftChild(pNode,'D');
q.Insert(pNewNode);
//将队头元素 C 出队,创建 C 的孩子结点 E 和 F,并将创建的孩子结点入队
q.Delete(pNode);
pNewNode=btree.InsertLeftChild(pNode,'E');
q.Insert(pNewNode);
pNewNode=btree.InsertRightChild(pNode,'F');
q.Insert(pNewNode);
//将队头元素 D 出队,创建 D 的右孩子结点 G,并将创建的孩子结点入队
q.Delete(pNode);
pNewNode=btree.InsertRightChild(pNode,'G');
q.Insert(pNewNode);
//将队头元素 E 出队,因 E 无孩子,所以不创建孩子结点
q.Delete(pNode);
//将队头元素 F 出队,创建 F 的孩子结点 H 和 I,并将创建的孩子结点入队
q.Delete(pNode);
pNewNode=btree.InsertLeftChild(pNode,'H');
q.Insert(pNewNode);
pNewNode=btree.InsertRightChild(pNode,'I');
q.Insert(pNewNode);
//依次将 G,H 和 I 出队,因它们无孩子,所以不创建孩子结点
q.Delete(pNode);
q.Delete(pNode);
```

```
    q.Delete(pNode);
    return 0;
}
```

提示:读者可通过调试方式查看所创建的二叉树中各结点的值是否正确。

5.4 二叉树的遍历及常用操作

在二叉树的一些应用中,常常要求在树中查找具有某种特征的结点或者对树中全部结点逐一进行某种处理,这时就涉及二叉树的遍历问题。

5.4.1 二叉树的遍历及其实现

二叉树的遍历就是按照某种规则依次访问二叉树中的每个结点,且每个结点仅被访问一次。本书根据结点访问顺序上的不同,介绍4种常用的遍历方式:先序遍历、中序遍历、后序遍历和逐层遍历。

1)先序遍历

先序遍历又称先根遍历,其访问方式递归定义如下:

(1)对于一棵二叉树,先访问其根结点,再访问根结点的左、右子树。

(2)对于左、右子树中的结点仍然是按照先序遍历方式访问,即先访问根结点,再访问根结点的左、右子树。

提示:在先序遍历中,只规定了根结点和其子树的访问顺序,而没有规定左、右子树的访问顺序。本书中规定,在先序、中序和后序遍历时均是先访问左子树后访问右子树。

例如,对于图5-12所示的二叉树,其先序遍历的结果为 A,B,D,G,C,E,F,H,I。

先序遍历的实现可以采用递归方式和非递归方式,下面先给出递归实现的描述。

【描述5-5】【描述5-3】类模板中递归先序遍历成员函数的实现。

```
//按递归方式先序遍历
template<class T>
void LinkedBinTree<T>::PreOrderTraverse(LinkedNode<T> * pNode)
{
    if(pNode==NULL)
        return;
    //先访问 pNode
    cout<<pNode->m_data<<" ";
    //再以先序遍历方式访问 pNode 的左子树
    PreOrderTraverse(pNode->m_pLeftChild);
    //最后以先序遍历方式访问 pNode 的右子树
    PreOrderTraverse(pNode->m_pRightChild);
}
```

可见，按递归方式实现的先序遍历算法具有代码简洁、可读性好等优点，但大量递归调用会产生额外的时间和空间开销，因此，为了提升程序性能，在实际编程中应尽量使用非递归方式实现算法。下面给出先序遍历的非递归实现描述。

【描述 5-6】【描述 5-3】类模板中非递归先序遍历成员函数的实现。

分析：非递归先序遍历需要利用栈实现，栈顶元素即是下一棵要访问的子树的根结点。具体步骤为：

(1)将二叉树的根结点入栈。

(2)将栈顶元素出栈并访问(即先访问根结点)，若栈顶元素存在右子树则将右子树根结点入栈(即根结点右孩子先于左孩子入栈，后于左孩子出栈，所以后访问)，若栈顶元素存在左子树则将左子树根结点入栈(即根结点左孩子后于右孩子入栈，先于右孩子出栈，所以先访问)。

(3)重复步骤(2)直至栈为空。

对于【例 5-2】中构建的二叉树(见图 5-12)，其非递归先序遍历过程如表 5-1 所示。

表 5-1　图 5-12 所示二叉树的非递归先序遍历过程

操　　作	栈中元素 (最左边为栈底元素，最右边为栈顶元素)
根结点 A 入栈	A
栈顶元素 A 出栈并访问，将 A 的右孩子 C 和左孩子 B 依次入栈	C,B
栈顶元素 B 出栈并访问，将 B 的左孩子 D 入栈	C,D
栈顶元素 D 出栈并访问，将 D 的右孩子 G 入栈	C,G
栈顶元素 G 出栈并访问	C
栈顶元素 C 出栈并访问，将 C 的右孩子 F 和左孩子 E 依次入栈	F,E
栈顶元素 E 出栈并访问	F
栈顶元素 F 出栈并访问，将 F 的右孩子 I 和左孩子 H 依次入栈	I,H
栈顶元素 H 出栈并访问	I
栈顶元素 I 出栈并访问	栈空
栈为空，二叉树先序遍历结束	栈空

参考程序如下：

```
//按非递归方式先序遍历
template<class T>
void LinkedBinTree<T>::PreOrderTraverse()
{
    LinkStack<LinkedNode<T>*> s;
    LinkedNode<T> *pNode=NULL;
```

```
    if(m_pRoot==NULL)
        return;
    //将根结点入栈
    s.Push(m_pRoot);
    //栈不为空时循环
    while(!s.IsEmpty())
    {
        //栈顶元素出栈并被访问
        s.Pop(pNode);
        cout<<pNode->m_data<<" ";
        //若结点存在右子树,则将右子树根结点入栈
        if(pNode->m_pRightChild)
            s.Push(pNode->m_pRightChild);
        //若结点存在左子树,则将左子树根结点入栈
        if(pNode->m_pLeftChild)
            s.Push(pNode->m_pLeftChild);
    }
}
```

2)中序遍历

中序遍历又称中根遍历,其访问方式递归定义如下:

(1)对于一棵二叉树,先访问根结点左子树,再访问根结点,最后访问右子树。

(2)对于左、右子树中的结点仍然是按照中序遍历方式访问。

例如,对于图 5-12 所示的二叉树,其中序遍历的结果为:D,G,B,A,E,C,H,F,I。

中序遍历的实现也可以采用递归方式和非递归方式,下面先给出递归实现的描述。

【描述 5-7】 【描述 5-3】类模板中递归中序遍历成员函数的实现。

```
//按递归方式中序遍历
template<class T>
void LinkedBinTree<T>::InOrderTraverse(LinkedNode<T> * pNode)
{
    if(pNode==NULL)
        return;
    //先以中序遍历方式访问 pNode 的左子树
    InOrderTraverse(pNode->m_pLeftChild);
    //再访问 pNode
    cout<<pNode->m_data<<" ";
    //最后以中序遍历方式访问 pNode 的右子树
    InOrderTraverse(pNode->m_pRightChild);
}
```

下面给出中序遍历的非递归实现的描述。

【描述 5-8】【描述 5-3】类模板中非递归中序遍历成员函数的实现。

分析：非递归中序遍历需要利用栈实现，栈顶元素即是下一个要访问的结点。具体步骤为：

(1)从整棵二叉树的根结点开始，将每棵子树的根结点进行入栈操作。

(2)判断刚入栈的根结点是否有左子树，若有则对左子树重复步骤(1)，直至刚入栈的根结点没有左子树(这样对二叉树或任一子树的根结点来说，都是先于其左子树的根结点入栈，也就是说其左子树的根结点会先于其出栈，从而保证在访问根结点之前先访问其左子树)。

(3)判断栈是否为空，若不为空则将栈顶元素出栈并访问，再判断栈顶元素是否有右子树，若有则回到步骤(1)访问栈顶元素的右子树，否则重复步骤(3)直至栈为空。

读者可按表 5-1 的形式自己列出图 5-12 所示二叉树的非递归中序遍历过程。

参考程序如下：

```
//按非递归方式中序遍历
template<class T>
void LinkedBinTree<T>::InOrderTraverse()
{
    LinkStack<LinkedNode<T>*> s;
    LinkedNode<T> *pNode=m_pRoot;
    //pNode 不为空时循环
    while(pNode)
    {
        //当 pNode 不为空时，将其入栈，并令 pNode 指向其左孩子
        while(pNode)
        {
            s.Push(pNode);
            pNode=pNode->m_pLeftChild;
        }
        //栈不为空，则栈顶结点出栈并被访问，令 pNode 指向取出栈顶结点的右孩子
        while(!s.IsEmpty())
        {
            s.Pop(pNode);
            cout<<pNode->m_data<<" ";
            pNode=pNode->m_pRightChild;
            if(pNode)  //若栈顶结点有右子树，则访问其右子树
                break;
        }
    }
}
```

3)后序遍历

后序遍历又称后根遍历，其访问方式递归定义如下：

(1)对于一棵二叉树,先访问根结点的左子树,后访问右子树,最后访问根结点。

(2)对于左、右子树中的结点仍然是按照后序遍历方式访问。

例如,对于图 5-12 所示的二叉树,其后序遍历的结果为 G,D,B,E,H,I,F,C,A。

后序遍历的实现也可以采用递归方式和非递归方式,下面先给出递归实现的描述。

【描述 5-9】 【描述 5-3】类模板中递归后序遍历成员函数的实现。

```
//按递归方式后序遍历
template<class T>
void LinkedBinTree<T>::PostOrderTraverse(LinkedNode<T> * pNode)
{
    if(pNode==NULL)
        return;
    //先以后序遍历方式访问 pNode 的左子树
    PostOrderTraverse(pNode->m_pLeftChild);
    //再以后序遍历方式访问 pNode 的右子树
    PostOrderTraverse(pNode->m_pRightChild);
    //最后访问 pNode
    cout<<pNode->m_data<<" ";
}
```

下面给出后序遍历的非递归实现的描述。

【描述 5-10】 【描述 5-3】类模板中非递归后序遍历成员函数的实现。

分析:非递归后序遍历需要利用栈实现,栈顶元素即是当前正在访问子树的根结点。具体步骤为:

(1)从整棵二叉树的根结点开始,将每棵子树的根结点进行入栈操作。

(2)判断刚入栈的根结点是否有左子树,若有则对左子树重复步骤(1),直至刚入栈的根结点没有左子树(这样对二叉树或任一子树的根结点来说,都是先于其左子树的根结点入栈,也就是说其左子树的根结点会先于其出栈,从而保证在访问根结点之前先访问其左子树)。

(3)判断栈是否为空,若不为空则取出栈顶元素(注意这里先不出栈),判断其右孩子是否为空或已被访问(通过比较栈顶元素右孩子与前一个访问的结点可知前一个访问的结点是否是栈顶元素的右孩子),若右孩子不为空且没被访问,则回到步骤(1)访问栈顶元素的右子树(这样可以保证在访问根结点之前先访问其右子树),否则访问当前栈顶元素,将栈顶元素出栈,并设置栈顶元素为前一个访问的结点,重复步骤(3)直至栈为空。

读者可按表 5-1 的形式自己列出图 5-12 所示二叉树的非递归后序遍历过程。

参考程序如下:

```
//按非递归方式后序遍历
template<class T>
void LinkedBinTree<T>::PostOrderTraverse()
{
    LinkStack<LinkedNode<T> * > s;
    LinkedNode<T> * pNode=m_pRoot, * pPreVisitNode = NULL;
```

```
        //pNode 不为空时循环
        while(pNode)
        {
            //当 pNode 不为空时,将其入栈,并令 pNode 指向其左孩子
            while (pNode)
            {
                s.Push(pNode);
                pNode=pNode->m_pLeftChild;
            }
            while(!s.IsEmpty())
            {
                //当栈不为空时,取出栈顶元素
                s.Top(pNode);
                //若栈顶元素的右孩子为空或已被访问,则访问当前栈顶元素,并将
                //栈顶元素出栈
                if(pNode->m_pRightChild==NULL
                    || pNode->m_pRightChild==pPreVisitNode)
                {
                    cout<<pNode->m_data<<" ";
                    s.Pop(pNode);
                    pPreVisitNode = pNode;  //设置 pNode 为前一个访问的结点
                    //设置 pNode 为空,表示 pNode 及其左右子树均已访问完毕,访问下
                    //一个栈中元素
                        pNode = NULL;
                }
                //否则,应先访问栈顶元素的右孩子
                else
                {
                    pNode=pNode->m_pRightChild;
                    break;
                }
            }
        }
    }
```

4)逐层遍历

逐层遍历是指从第一层开始依次对每层中的结点按照从左至右的顺序进行访问。例如,对于图 5-12 所示的二叉树,其逐层遍历的结果为 A,B,C,D,E,F,G,H,I。

逐层遍历的实现只能采用非递归方式,下面给出具体实现方法。

【描述 5-11】【描述 5-3】类模板中非递归逐层遍历成员函数的实现。

分析:非递归逐层遍历需要利用队列实现,队头元素即是下一个要访问的结点。具体步骤为:

(1)将二叉树的根结点入队。

(2)将队头元素出队并访问,若队头元素有左子树则将左子树根结点入队,若队头元素有右子树则将右子树根结点入队。

(3)重复步骤(2)直至队列为空。

读者可按表 5-1 的形式自己列出图 5-12 所示二叉树的非递归逐层遍历过程。

参考程序如下:

```
//按非递归方式逐层遍历
template<class T>
void LinkedBinTree<T>::LevelOrderTraverse()
{
    LinkQueue<LinkedNode<T>*> q;
    LinkedNode<T> *pNode=NULL;
    if(m_pRoot==NULL)  return;
    //将根结点入队
    q.Insert(m_pRoot);
    //当队列不为空时循环
    while(!q.IsEmpty())
    {
        //将队头元素出队并访问
        q.Delete(pNode);
        cout<<pNode->m_data<<" ";
        //若结点存在左子树,则将左子树根结点入队
        if(pNode->m_pLeftChild)
            q.Insert(pNode->m_pLeftChild);
        //若结点存在右子树,则将右子树根结点入队
        if(pNode->m_pRightChild)
            q.Insert(pNode->m_pRightChild);
    }
}
```

5.4.2 二叉树常用操作的实现

这里给出二叉树常用操作的实现方法。

1)获取指定结点的双亲结点

在二叉树的二叉链表表示中,结点中没有指向其双亲结点的指针,要获取双亲结点需要从根结点开始遍历二叉树直至找到指定结点的双亲结点。因此,可以参照前面给出的二叉

树遍历算法，编写获取双亲结点的程序。下面只给出按非递归方式实现的算法。

【描述 5-12】【描述 5-3】类模板中按非递归方式获取指定结点双亲结点的成员函数实现。

```
//按非递归方式获取指定结点的双亲结点
template<class T>
LinkedNode<T> * LinkedBinTree<T>::GetParent(LinkedNode<T> *pNode)
{
    LinkQueue<LinkedNode<T> *> q;
    LinkedNode<T> *pCurNode=NULL;
    //若指定结点 pNode 为根结点，则返回空
    if(pNode==m_pRoot)
        return NULL;
    //若二叉树是空树，则返回空
    if(m_pRoot==NULL)
        return NULL;
    //按非递归逐层遍历的方式搜索双亲结点
    //将根结点入队
    q.Insert(m_pRoot);
    //当队列不为空时循环
    while(!q.IsEmpty())
    {
        //将队头元素出队
        q.Delete(pCurNode);
        //如果 pNode 是队头元素的孩子，则返回队头元素
        if(pCurNode->m_pLeftChild==pNode
            || pCurNode->m_pRightChild==pNode)
            return pCurNode;
        //若结点存在左子树，则将左子树根结点入队
        if(pCurNode->m_pLeftChild)
            q.Insert(pCurNode->m_pLeftChild);
        //若结点存在右子树，则将右子树根结点入队
        if(pCurNode->m_pRightChild)
            q.Insert(pCurNode->m_pRightChild);
        }
        return NULL;
}
```

2）删除以指定结点为根的子树

删除以指定结点为根的子树，一方面要将子树从二叉树中删除，另一方面要将子树中的结点释放。将子树从二叉树中删除是通过将指定结点的双亲结点的指针域置空来实现的

(若删除的是整棵二叉树,则应将根结点指针域置空),然后将子树中的结点释放,采用类似于遍历子树中所有结点的方式将各结点占据的内存释放。因此,同遍历算法一样,删除子树也可以采用递归方式和非递归方式。下面只给出按非递归方式实现的算法。

【描述 5-13】【描述 5-3】类模板中按非递归方式删除以指定结点为根的子树的成员函数实现。

```
//删除以指定结点为根的子树
template<class T>
void LinkedBinTree<T>::DeleteSubTree(LinkedNode<T> *pNode)
{
    LinkedNode<T> *pParentNode=NULL;
    //若指定结点为空,则返回
    if(pNode==NULL)
        return;
    //若将整棵二叉树删除,则令根结点为空
    if(m_pRoot==pNode)
        m_pRoot=NULL;
    //否则,若指定结点存在双亲结点,则将双亲结点的左右指针域置空
    else if((pParentNode=GetParent(pNode))!=NULL)
    {
        if(pParentNode->m_pLeftChild==pNode)
            pParentNode->m_pLeftChild=NULL;
        else
            pParentNode->m_pRightChild=NULL;
    }
    //否则,指定结点不是二叉树中的结点,直接返回
    else
        return;
    //调用 DeleteSubTreeNode 函数删除以 pNode 为根的子树
    DeleteSubTreeNode(pNode);
}
//由 DeleteSubTree 函数调用按非递归方式删除以指定结点为根的子树
template<class T>
void LinkedBinTree<T>::DeleteSubTreeNode(LinkedNode<T> *pNode)
{
    LinkQueue<LinkedNode<T>*> q;
    LinkedNode<T> *pCurNode=NULL;
    if(pNode==NULL)
        return;
    //按非递归层次遍历的方式删除子树
```

```
    q.Insert(pNode);
    while(!q.IsEmpty())
    {
        q.Delete(pCurNode);
        if(pCurNode->m_pLeftChild)
            q.Insert(pCurNode->m_pLeftChild);
        if(pCurNode->m_pRightChild)
            q.Insert(pCurNode->m_pRightChild);
        delete pCurNode;
    }
}
```

3)根据关键字查找结点

根据关键字查找结点，实质上就是按照某种规则依次访问二叉树中的每一结点，直至找到与关键字匹配的结点。因此，同遍历算法一样，根据关键字查找结点也可以采用递归方式和非递归方式。下面只给出按非递归方式实现的算法。

【描述 5-14】【描述 5-3】类模板中按非递归方式根据关键字查找结点的成员函数实现。

```
//按非递归方式根据关键字查找结点
template<class T>
LinkedNode<T> * LinkedBinTree<T>::SearchByKey(const T &x)
{
    LinkQueue<LinkedNode<T> * > q;
    LinkedNode<T>  * pMatchNode=NULL;
    if(m_pRoot==NULL)  return NULL;
    //按非递归层次遍历的方式查找结点
    q.Insert(m_pRoot);
    while (!q.IsEmpty())
    {
        q.Delete(pMatchNode);
        if(pMatchNode->m_data==x)
            return pMatchNode;
        if(pMatchNode->m_pLeftChild)
            q.Insert(pMatchNode->m_pLeftChild);
        if(pMatchNode->m_pRightChild)
            q.Insert(pMatchNode->m_pRightChild);
        }
        return NULL;
}
```

4)其他常用操作

下面给出其他常用操作的实现的描述。

【描述 5-15】【描述 5-3】类模板中其他成员函数实现。

```
//实现删除二叉树
template<class T>
LinkedBinTree<T>::~LinkedBinTree()
{
    Clear();  //清空二叉树中的结点
}
//实现清空二叉树
template<class T>
void LinkedBinTree<T>::Clear()
{
    DeleteSubTree(m_pRoot);
}
//判断二叉树是否为空
template<class T>
bool LinkedBinTree<T>::IsEmpty()
{
    if(m_pRoot==NULL)
        return true;
    return false;
}
//获取根结点
template<class T>
LinkedNode<T> * LinkedBinTree<T>::GetRoot()
{
    return m_pRoot;
}
//修改指定结点的元素值
template<class T>
bool LinkedBinTree<T>::ModifyNodeValue(LinkedNode<T> *pNode,const T &x)
{
    if(pNode==NULL)
        return false;
    pNode->m_data=x;
    return true;
}
//获取指定结点的元素值
template<class T>
bool LinkedBinTree<T>::GetNodeValue(LinkedNode<T> *pNode,T &x)
```

```
{
    if(pNode==NULL)
        return false;
    x=pNode->m_data;
    return true;
}
//获取指定结点的左孩子结点
template<class T>
LinkedNode<T> * LinkedBinTree<T>::GetLeftChild(LinkedNode<T> *pNode)
{
    if(pNode==NULL)
        return NULL;
    return pNode->m_pLeftChild;
}
//获取指定结点的右孩子结点
template<class T>
LinkedNode<T> * LinkedBinTree<T>::GetRightChild(LinkedNode<T> *pNode)
{
    if(pNode==NULL)
        return NULL;
    return pNode->m_pRightChild;
}
```

提示:将【描述 5-3】中的声明代码和【描述 5-4】至【描述 5-15】中的实现代码一起存储在 LinkedBinTree.h 文件中,以后就可以基于该类模板快速完成链式二叉树相关应用问题的求解。

【例 5-3】 基于二叉树二叉链表表示的 C++实现代码和【例 5-2】,完成如下操作:

(1)将以结点 F 为根结点的子树删除,并分别通过先序、中序、后序和逐层遍历输出各结点的值。

(2)将结点 G 的值改为"F",并分别通过先序、中序、后序和逐层遍历输出各结点的值。

(3)清空二叉树中的元素。

参考程序如下:

```
/* Chap5-3.cpp:二叉树链式存储结构的应用问题*/
#include "LinkedBinTree.h"
int main()
{
    …  //【例 5-2】中创建二叉树的代码
    cout<<"当前二叉树的元素为:\n";
    cout<<"先序遍历结果为:";
    btree.PreOrderTraverse();
```

```
cout<<endl;
cout<<"中序遍历结果为:";
btree.InOrderTraverse();
cout<<endl;
cout<<"后序遍历结果为:";
btree.PostOrderTraverse();
cout<<endl;
cout<<"逐层遍历结果为:";
btree.LevelOrderTraverse();
cout<<endl;
//将以结点 F 为根的子树删除
pNode=btree.SearchByKey('F');
btree.DeleteSubTree(pNode);
cout<<"将以结点 F 为根的子树删除后";
cout<<"当前二叉树的元素为:\n";
cout<<"先序遍历结果为:";
btree.PreOrderTraverse();
cout<<endl;
cout<<"中序遍历结果为:";
btree.InOrderTraverse();
cout<<endl;
cout<<"后序遍历结果为:";
btree.PostOrderTraverse();
cout<<endl;
cout<<"逐层遍历结果为:";
btree.LevelOrderTraverse();
cout<<endl;
//将结点 G 的值改为“F”
pNode=btree.SearchByKey('G');
btree.ModifyNodeValue(pNode,'F');
cout<<"将结点 G 的值改为“F”后";
cout<<"当前二叉树的元素为:\n";
cout<<"先序遍历结果为:";
btree.PreOrderTraverse();
cout<<endl;
cout<<"中序遍历结果为:";
btree.InOrderTraverse();
cout<<endl;
cout<<"后序遍历结果为:";
```

```
    btree.PostOrderTraverse();
    cout<<endl;
    cout<<"逐层遍历结果为:";
    btree.LevelOrderTraverse();
    cout<<endl;
    //将二叉树清空
    btree.Clear();
    cout<<"将二叉树清空后";
    cout<<"当前二叉树的元素为:\n";
    cout<<"先序遍历结果为:";
    btree.PreOrderTraverse();
    cout<<endl;
    cout<<"中序遍历结果为:";
    btree.InOrderTraverse();
    cout<<endl;
    cout<<"后序遍历结果为:";
    btree.PostOrderTraverse();
    cout<<endl;
    cout<<"逐层遍历结果为:";
    btree.LevelOrderTraverse();
    cout<<endl;
    return 0;
}
```

程序的运行结果如下：

```
当前二叉树的元素为:
先序遍历结果为:A B D G C E F H I
中序遍历结果为:D G B A E C H F I
后序遍历结果为:G D B E H I F C A
逐层遍历结果为:A B C D E F G H I
将以结点 F 为根的子树删除后当前二叉树的元素为:
先序遍历结果为:A B D G C E
中序遍历结果为:D G B A E C
后序遍历结果为:G D B E C A
逐层遍历结果为:A B C D E G
将结点 G 的值改为“F”后当前二叉树的元素为:
先序遍历结果为:A B D F C E
中序遍历结果为:D F B A E C
后序遍历结果为:F D B E C A
逐层遍历结果为:A B C D E F
```

将二叉树清空后当前二叉树的元素为：
先序遍历结果为：
中序遍历结果为：
后序遍历结果为：
逐层遍历结果为：

5.5 哈夫曼树和哈夫曼码

哈夫曼树又称最优二叉树，是指在一类有着相同叶子结点的树中具有最短带权路径长度的二叉树。哈夫曼树在实际中有着广泛的应用。

5.5.1 基本术语

在介绍哈夫曼树之前，先介绍一些基本术语。

1)结点的权和带权路径长度

在实际应用中，往往给树中的结点赋予一个具有某种意义的实数，该实数就称为结点的权。结点的带权路径长度是指从树根到该结点的路径长度与结点的权的乘积。

2)树的带权路径长度

树的带权路径长度是指树中所有叶子结点的带权路径长度之和，记为 $WPL=\sum_{i=1}^{n}W_iL_i$（其中，n 为叶子结点的数目，W_i 为第 i 个叶子结点的权，L_i 为根结点到第 i 个叶子结点的路径长度，可知 W_iL_i 为第 i 个叶子结点的带权路径长度）。

例如，图 5-14 中的两棵二叉树，其带权路径长度分别为：

$$WPL(a)=2*(9+7+5)+3*(2+3)=57$$

$$WPL(b)=1*3+3*(2+7+5+9)=72$$

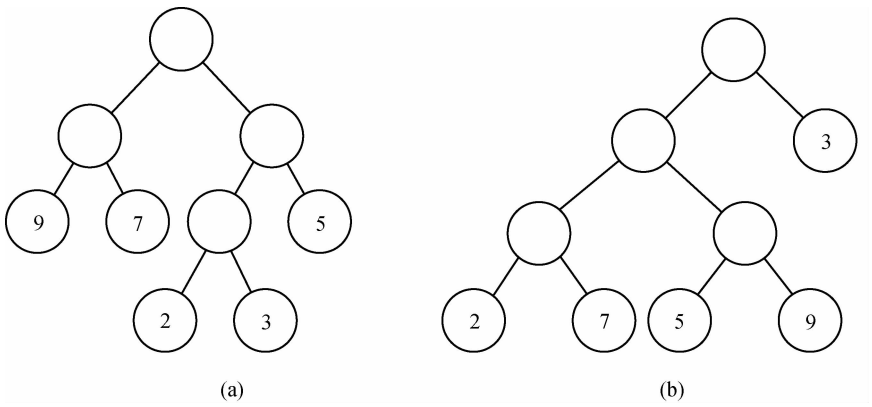


图 5-14　具有不同带权路径长度的二叉树

5.5.2　哈夫曼树及其构造方法

在由 n 个叶子结点构成的一类二叉树中，具有最短带权路径长度的二叉树称为哈夫曼树。其构造方法如下：

(1)已知 n 个权值为 $W_i(i=1, 2, \cdots, n)$的结点，将每个结点作为根结点生成 n 棵只有根结点的二叉树 T_i，形成森林 $F=\{T_1, T_2, \cdots, T_n\}$。

(2)从森林 F 中选出根结点权值最小的两棵二叉树 T_p 和 T_q，并通过添加新的根结点将它们合并为一棵新二叉树，新二叉树中 T_p 和 T_q 分别作为根结点的左子树和右子树，且根结点的权值等于 T_p 和 T_q 两棵二叉树的根结点权值之和。以合并后生成的新二叉树替代森林 F 中的原有二叉树 T_p 和 T_q。重复该步骤直至森林 F 中只存在一棵二叉树。

图 5-15 所示是哈夫曼树的构造过程示例。初始有根结点权值分别为 2,3,5,7,9 的 5 棵二叉树组成的森林，如图 5-15(a)所示。从中选取根结点权值最小的两棵二叉树进行合并，得到图 5-15(b)所示的森林……重复该合并操作，直至森林中只剩下一棵二叉树，这棵二叉树就是哈夫曼树，如图 5-15(e)所示。

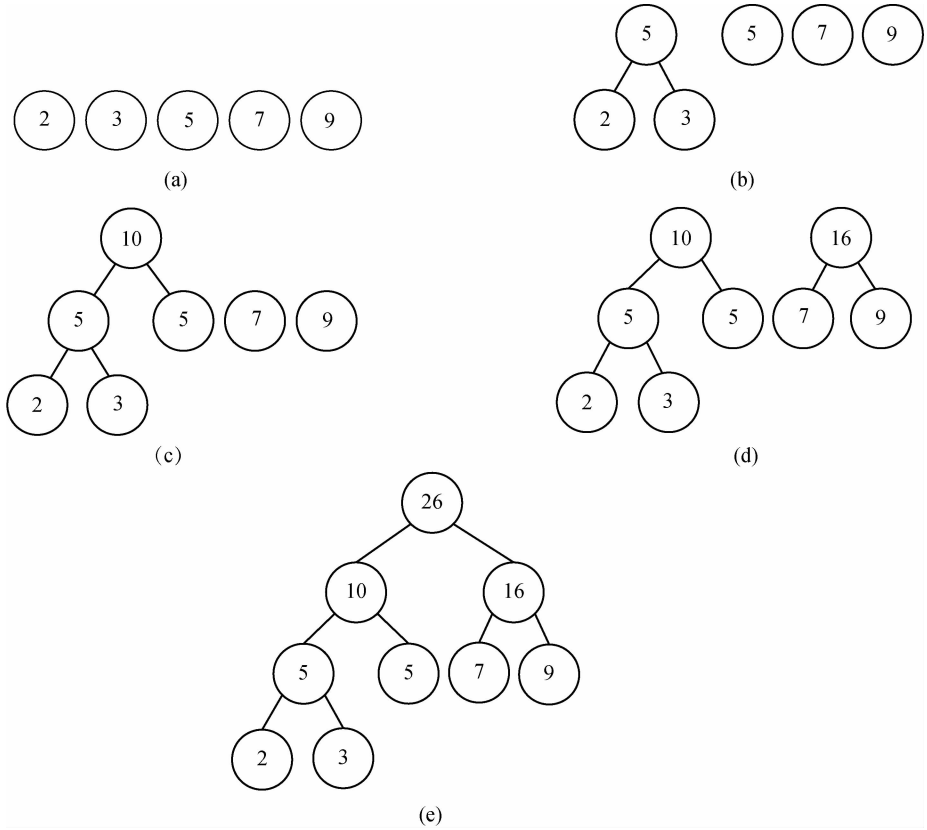


图 5-15　哈夫曼树构造过程示例

5.5.3　哈夫曼码及其编解码方法

哈夫曼码是利用哈夫曼树得到的一种不定长的二进制编码，它在数据压缩领域有着广

泛的应用。利用哈夫曼码进行数据压缩的基本原理是:对于出现频率较高的字符,使用较短的编码;而对于出现频率较低的字符,则使用较长的编码,从而使得用于表示字符序列的编码总长度最短,节省存储空间。

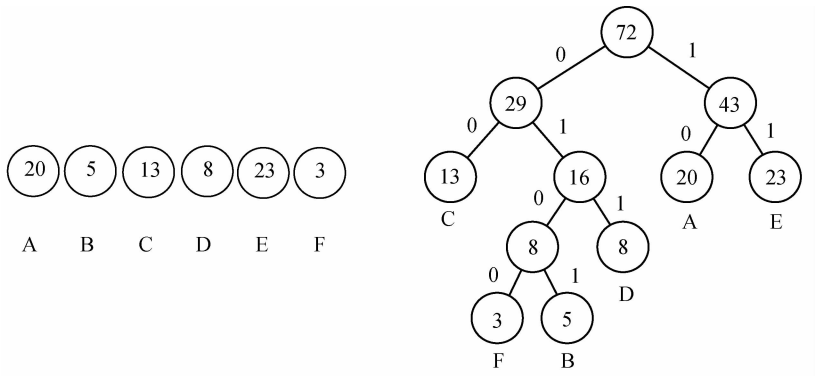
1)哈夫曼编码

哈夫曼编码是指将用其他编码法表示的字符序列转成用哈夫曼码表示,以减少存储空间,其具体方法为:

(1)以要编码的字符集 $C=\{c_1, c_2, \cdots, c_n\}$ 作为叶子结点,以字符出现的频度或次数 $W=\{w_1, w_2, \cdots, w_n\}$ 作为结点的权,构造哈夫曼树。

(2)规定哈夫曼树中,从根结点开始,双亲结点到左孩子结点的分支标为 0,双亲结点到右孩子结点的分支标为 1。从根结点到某一叶子结点经过的分支形成的编码即是该叶子结点所对应字符的哈夫曼码。

例如,假设要编码的字符集为{A, B, C, D, E, F},各字符的出现次数为{20, 5, 13, 8, 23, 3},则其哈夫曼码的编码过程如图 5-16 所示。首先根据图 5-16(a)所示的字符集和各字符出现次数构造图 5-16(b)所示的哈夫曼树,再利用哈夫曼树对各字符进行哈夫曼编码,编码结果如图 5-16(c)所示。可见,在最后得到的哈夫曼码编码表中,对于出现频率较高的字符采用较短的编码形式,如出现频率最高的 3 个字符 E、A、C 都是采用两位编码;对于出现频率较低的字符则采用较长的编码形式,如出现频率最低的两个字符 F、B 都是采用 4 位编码,这使得用于表示字符序列的编码总长度最短、节省了存储空间。另外,每个字符的哈夫曼码都不是另一个字符哈夫曼码的前缀,因此,在使用哈夫曼码存储数据时,各字符编码之间不需加空格等分隔符。



(a) 初始森林　　(b) 哈夫曼树

字符	A	B	C	D	E	F
出现次数	20	5	13	8	23	3
哈夫曼码	10	0101	00	011	11	0100

(c) 哈夫曼编码表

图 5-16　哈夫曼码编码过程示例

2)哈夫曼解码

哈夫曼解码是指将用哈夫曼码表示的字符序列转成其他编码法表示,以让计算机正确

显示字符内容，其具体方法为：

(1)将用于表示字符序列的哈夫曼码逐位取出并送入哈夫曼树中。

(2)从哈夫曼树的根结点开始，对于每一个结点，遇到位 0 则经左分支到其左孩子，遇到位 1 则经右分支到其右孩子。重复该过程，直至到达某一个叶子结点，该叶子结点所对应的字符即是解码结果。解码一个字符后回到哈夫曼树的根结点开始解码下一个字符。

例如，假设要解码的哈夫曼码是 0100100011，则根据图 5-16(b)所示的哈夫曼树可得到解码结果为 FACE。

提示：哈夫曼编码是选择结点建立二叉树的过程，哈夫曼解码则是根据编码中的 0 或 1 访问二叉树的左或右子树直到叶子结点的过程。读者可基于 LinkedBinTree.h 完成哈夫曼编码和解码问题的求解。关于哈夫曼树的问题将作为上机实习供读者练习。

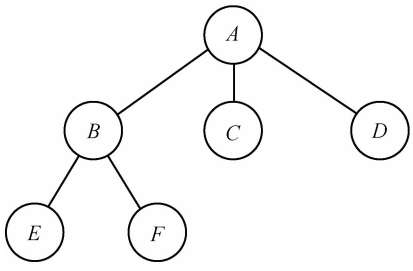
5.6　树的表示法

这里介绍树的 4 种常用表示方法，即双亲表示法、孩子表示法、孩子双亲表示法和孩子兄弟表示法。

5.6.1　双亲表示法

在树结构中，每个结点可以有零个或多个孩子结点，但至多只有一个双亲结点，因此，通过在孩子结点中设置一个指针域记录其双亲结点的存储位置就可以唯一地表示一棵树。这种表示方法称为双亲表示法。

双亲表示法通常采用顺序存储方式，从根结点开始编号为 0，其他结点按照自上而下、从左至右的顺序递增编号；每个结点除了有一个数据域存储数据外，还有一个指针域存储其双亲结点的位置；若一个结点没有双亲结点，则其指针域赋值为－1。例如，对于图 5-17(a)所示的树，其双亲表示法如图 5-17(b)所示。



(a)

	数据域	指针域
0	A	−1
1	B	0
2	C	0
3	D	0
4	E	1
5	F	1

(b)

图 5-17　双亲表示法示例

在双亲表示法中,通过一个结点的指针域可以直接获取到该结点的双亲结点,但要获取一个结点的孩子结点,则可能需要遍历整棵树。

5.6.2 孩子表示法

与双亲表示法相反,孩子表示法是通过在双亲结点中设置指向孩子结点的指针域来表示一棵树。一个结点可以有零个或多个孩子,根据结点中指向孩子结点的指针域数目是否固定可以分为定长孩子表示法和不定长孩子表示法。

1)定长孩子表示法

在定长孩子表示法中,每个结点中的指针域数目是固定的。若树的度为 n,则结点中指针域数目也为 n。定长孩子表示法中的结点结构如图 5-18 所示。

数据域	指针域 1	指针域 2	…	指针域 n

图 5-18 定长孩子表示法中的结点结构

对于图 5-17(a)所示的树,其定长孩子表示法如图 5-19 所示。

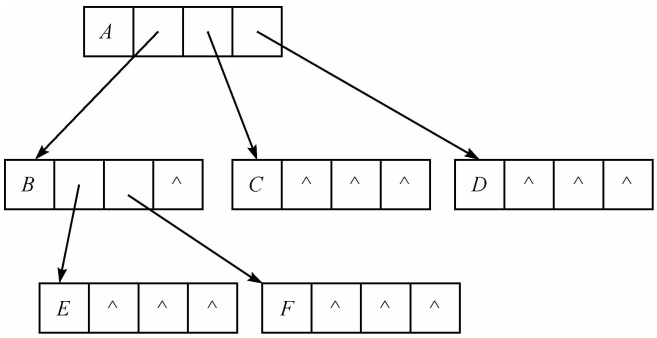


图 5-19 定长孩子表示法示例

2)不定长孩子表示法

在不定长孩子表示法中,每个结点中的指针域数目是不固定的。若结点的度为 d,则结点中指针域数目也为 d。不定长孩子表示法中的结点结构如图 5-20 所示。

数据域	结点的度 d	指针域 1	指针域 2	…	指针域 d

图 5-20 不定长孩子表示法中的结点结构

对于图 5-17(a)所示的树,其不定长孩子表示法如图 5-21 所示。

可见,定长孩子表示法需要预先确定树的度,而且空指针域数量较多,因此空间浪费较大,但优点是操作方便,不需人工管理内存;不定长孩子表示法则相对灵活,可以根据结点的度动态分配指针域,没有空间浪费,但需人工管理内存,初学者操作起来容易出错。

与双亲表示法相反,在孩子表示法中,通过一个结点的指针域可以直接获取到该结点的孩子结点,但要获取一个结点的双亲结点,则可能需要遍历整棵树。

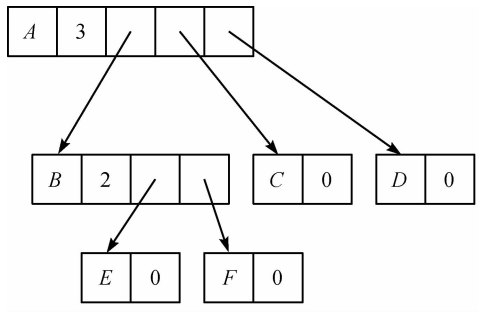


图 5-21　不定长孩子表示法示例

5.6.3　孩子双亲表示法

孩子双亲表示法综合了孩子表示法和双亲表示法的特点，既在孩子结点中设置记录双亲结点位置的指针域，又在双亲结点中设置记录孩子结点位置的指针域。孩子双亲表示法中的结点结构有多种形式可供选择，这里只给出一种示例形式，如图 5-22 所示。其中，指针域 0 是用于记录双亲结点位置的指针域，指针域 1～d 是用于记录孩子结点位置的指针域。

数据域	指针域0	结点的度d	指针域1	指针域2	…	指针域d

图 5-22　孩子双亲表示法中的结点结构示例

对于图 5-17(a)所示的树，其孩子双亲表示法如图 5-23 所示。

	数据域	指针域0	结点的度	指针域1	指针域2	指针域3
0	*A*	–1	3	1	2	3
1	*B*	0	2	4	5	
2	*C*	0	0			
3	*D*	0	0			
4	*E*	1	0			
5	*F*	1	0			

图 5-23　孩子双亲表示法示例

孩子双亲表示法兼具了孩子表示法和双亲表示法的优点，通过一个结点的指针域可以直接获取到该结点的孩子结点和双亲结点，但在添加或删除结点时操作比较复杂。

5.6.4　孩子兄弟表示法

孩子兄弟表示法又称二叉链表表示法，与二叉树的二叉链表表示法存储结构完全相同，只是结点中指针域的含义有所不同。在这种表示法中，结点的结构如图 5-24(a)所示。对于

图 5-17(a)所示的树,其孩子兄弟表示法如图 5-24(b)所示。

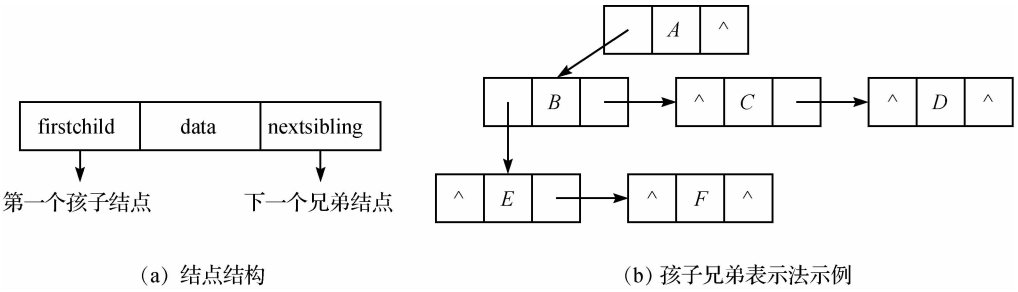


图 5-24　树的孩子兄弟表示法

采用孩子兄弟表示法的树与采用二叉链表表示法的二叉树具有相同的存储结构,可直接利用前面定义的各种二叉树算法来实现对树的操作,因此,孩子兄弟表示法是一种在实际中比较常用的树的表示法。

5.7　树、森林与二叉树的转换

5.7.1　树、森林转换为二叉树

将树和森林转换为二叉树,就可以使用二叉树的各种算法对树和森林进行操作。

1)树转换为二叉树

将树转换成二叉树的步骤为:

(1)将树用二叉链表表示法表示,即在相邻兄弟结点之间加上连线,并将除双亲结点到第一个孩子结点的连线保留外,双亲结点到其他孩子结点的连线均删除。对图 5-25(a)所示的树用二叉链表表示的结果如图 5-25(b)所示。

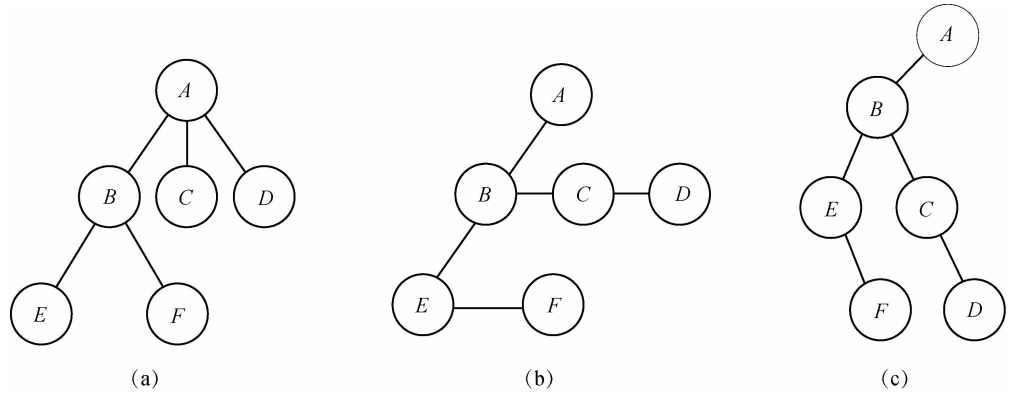


图 5-25　树转换为二叉树的过程示例

(2)将结点到第一个孩子结点的连线作为左子树的边,结点到兄弟结点的连线作为右子树的边。转换后的二叉树如图 5-25(c)所示。

2)森林转换为二叉树

将森林转换为二叉树的步骤为:

(1)将森林中的每棵树都用二叉链表表示法表示,并将各棵二叉树的根结点看做是兄弟结点,在它们之间加上连线。对图 5-26(a)所示的森林转换后的结果如图 5-26(b)所示。

(2)将结点到第一个孩子结点的连线作为左子树的边,结点到兄弟结点的连线作为右子树的边。转换后的二叉树如图 5-26(c)所示。

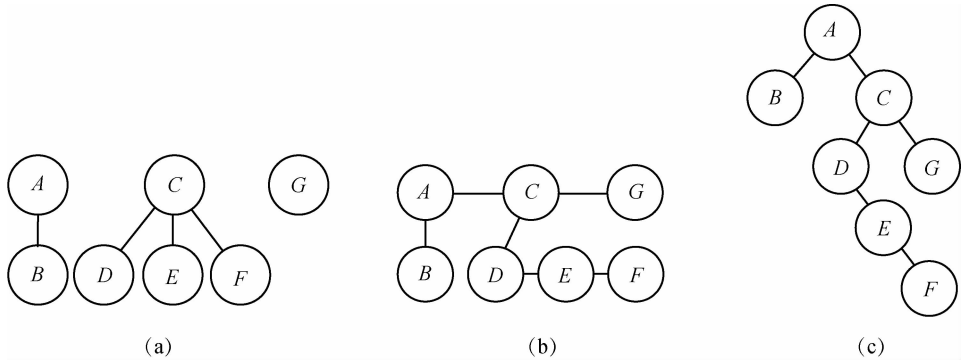


图 5-26　森林转换为二叉树的过程示例

5.7.2　二叉树转换为树、森林

从树转换为二叉树的过程可知,任何一棵树对应的二叉树,其根结点必然没有右子树。也就是说,并不是所有二叉树都能转为树,只有那些根结点右子树为空的二叉树可以转为树,而那些根结点右子树不为空的二叉树只能转为由若干棵树组成的森林。二叉树转化为树或森林的方法相同,具体步骤为:

(1)将一个结点左子树的边作为该结点指向第一个孩子结点的连线,右子树的边作为该结点到兄弟结点的连线。对图 5-25(c)所示的二叉树转换后的结果如图 5-25(b)所示;对图 5-26(c)所示的二叉树转换后的结果如图 5-26(b)所示。

(2)在双亲结点和它的各孩子结点之间加上连线,并删除兄弟结点之间的连线,得到一棵树(见图 5-25a)或一个包含若干棵树的森林(见图 5-26a)。

提示:感兴趣的读者可以自己实现树及其相关操作,在此不再赘述。

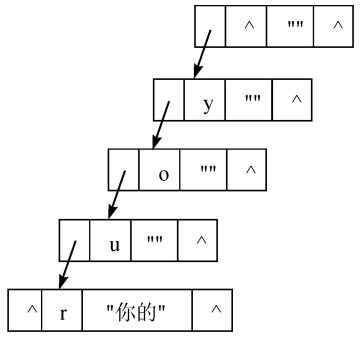
5.8　应 用 实 例

编写电子词典应用程序,要求如下:

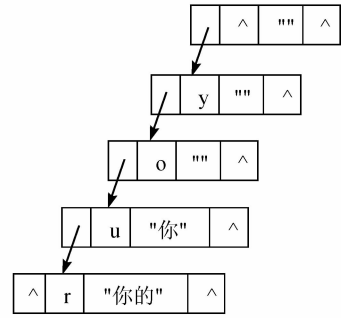
(1)可以向词典中添加英文单词及其中文解释。

(2)可以根据用户输入的英文单词查找其对应的中文解释。

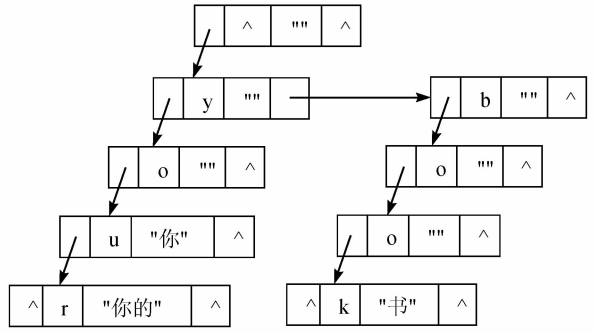
求解思路:先定义 Word 类,该类包含两个数据成员,char 型的 m_cEngLetter 和 string 型的 m_strChinese,分别用于存储英文字符及英文单词对应的中文解释,并根据问题需要为 Word 类定义若干成员函数。在主程序中,先基于【描述 5-4】至【描述 5-15】中的 LinkedBinTree.h 定义的二叉树的二叉链表类模板,根据用户输入的单词列表创建词典树(这里实际上是用前面定义的二叉链表来表示树,每个结点中的 m_pLeftChild 指针用于指向第一个孩子结点,m_pRightChild 指针用于指向下一个兄弟结点),再根据用户输入的单词在词典树中进行单词查询。创建的词典树的结构示例如图 5-27 所示。



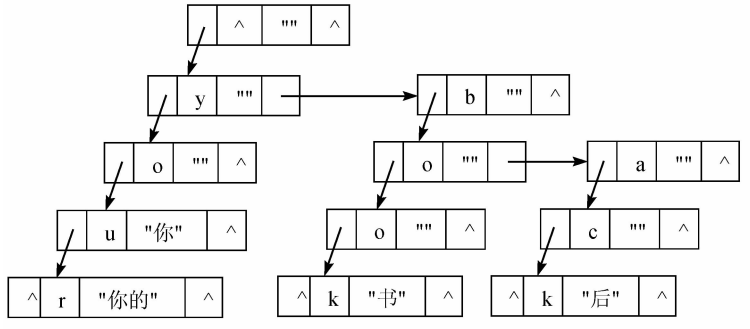
(a) 添加单词 your 及其中文解释“你的”之后的词典树



(b) 添加单词 you 及其中文解释“你”之后的词典树



(c) 添加单词book及其中文解释“书”之后的词典树



(d) 添加单词 back 及其中文解释“后”之后的词典树

图 5-27　词典树结构示例

将 Word 类的定义存储在 Word.h 头文件中，创建词典树和在词典树中查找单词的代码存储在 Chap5-4.cpp 源文件中。

参考程序如下：

```
//Word.h
#ifndef  _WORD_H
#define  _WORD_H
#include <string>
using namespace std;
class Word
{
public:
    Word(char cEngLetter='', string strChinese="")
    {
        m_cEngLetter=cEngLetter;
        m_strChinese=strChinese;
    }
     friend void AddWordToDictionary(LinkedBinTree<Word> * pTree, string
     strEnglish,string strChinese);
     friend string SearchWordFromDictionary(LinkedBinTree<Word> * pTree,
     string strEnglish);
private:
    char m_cEngLetter;
    string m_strChinese;
};
#endif  //end of _WORD_H
/* Chap5-4.cpp:电子词典应用程序*/
#include "LinkedBinTree.h"
#include "Word.h"
#include<iostream>
using namespace std;
//将英文单词及其中文解释添加到词典中
void AddWordToDictionary(LinkedBinTree<Word> * pTree, string strEnglish,
string strChinese)
{
    LinkedNode<Word> * pNode=pTree->GetRoot(), * pPreLevelNode=NULL,
     * pPreSiblingNode=NULL;
    int nI,nJ;
    bool bMatch;
    Word word;
```

```
if (pNode==NULL)
    return;
pPreLevelNode=pNode;
pNode=pTree->GetLeftChild(pNode);
for (nI=0;nI<(int)strEnglish.length();nI++)
{
    pPreSiblingNode=NULL;
    bMatch=false;
    //对于单词中的每个字符,从第一个孩子结点开始查找,直到找到匹配
    //结点或所有孩子都比较完毕
    while (pNode)
    {
        pTree->GetNodeValue(pNode, word);
        if (word.m_cEngLetter==strEnglish[nI])
        {
        pPreLevelNode=pNode;
        pNode=pTree->GetLeftChild(pNode);
        bMatch=true;
        break;
        }
    pPreSiblingNode=pNode;
    pNode=pTree->GetRightChild(pNode);
    }
//如果当前字符没有找到匹配的结点,则将当前字符及后继字符逐个添加到树中
if(!bMatch)
{
    word.m_strChinese="";
    for(nJ=nI; nJ<(int)strEnglish.length()-1; nJ++)
    {
        word.m_cEngLetter=strEnglish[nJ];
        if(pPreSiblingNode)
        {
            pTree->InsertRightChild(pPreSiblingNode, word);
            pPreLevelNode=pTree->GetRightChild(pPreSiblingNode);
            pPreSiblingNode=NULL;
        }
        else
        {
            pTree->InsertLeftChild(pPreLevelNode, word);
```

```
                pPreLevelNode=pTree->GetLeftChild(pPreLevelNode);
            }
        }
        //最后一个字符所对应的结点存储该单词的中文解释
        word.m_cEngLetter=strEnglish[nJ];
        word.m_strChinese=strChinese;
        if(pPreSiblingNode)
            pTree->InsertRightChild(pPreSiblingNode,word);
        else
            pTree->InsertLeftChild(pPreLevelNode,word);
        break;
    }
}
//如果都匹配上,表明单词中的所有字符都能找到匹配的结点,此时在与最后一个
//字符匹配的结点中存储该单词的中文解释
    if(nI==(int)strEnglish.length())
    {
    word.m_strChinese=strChinese;
    pTree->ModifyNodeValue(pPreLevelNode,word);
    }
}
//根据用户输入的英文单词查找其对应的中文解释
string SearchWordFromDictionary(LinkedBinTree<Word> *pTree,string strEnglish)
{
    string strChinese="";
    LinkedNode<Word> *pNode=pTree->GetRoot();
    int nI;
    Word word;
    //将用户输入的单词逐个字符与结点中存储的字符比较
    for(nI=0;nI<(int)strEnglish.length();nI++)
    {
        //如果在上一次比较中没有找到匹配的结点,则要查找的单词不存在
        if(pNode==NULL)
            break;
        //获取第一个孩子结点
        pNode=pTree->GetLeftChild(pNode);
        //从第一个孩子结点开始与 strEnglish[nI]比较,直到找到匹配的结点或者
        //所有孩子结点都比较完毕
        while(pNode)
```

```
            {
                pTree->GetNodeValue(pNode,word);
                if(word.m_cEngLetter==strEnglish[nI])
                    break;
                pNode=pTree->GetRightChild(pNode);
            }
        }
        //如果单词中的所有字符都能找到匹配的结点,则查找成功,将最后匹配结点中存
        //储的中文解释赋给 strChinese 作为返回值
        if(pNode!=NULL)
            strChinese=word.m_strChinese;
        return strChinese;
    }
    int main()
    {
        LinkedBinTree<Word> btree;
        char cAnswer;
        string strEnglish,strChinese;
        Word word;
        //创建根结点,根结点中不存储任何数据
        btree.CreateRoot(word);
        while(1)
        {
            cout<<"是否添加新单词到词典中? (y/n)";
            cin>>cAnswer;
            if(cAnswer=='n')
                break;
            cout<<"请输入英文单词:";
            cin>>strEnglish;
            cout<<"请输入中文解释:";
            cin>>strChinese;
            AddWordToDictionary(&btree,strEnglish,strChinese);
        }
        while(1)
        {
            cout<<"是否要查询单词? (y/n)";
            cin>>cAnswer;
            if (cAnswer=='n')
                break;
```

```
        cout<<"请输入要查询的英文单词:";
        cin>>strEnglish;
        strChinese=SearchWordFromDictionary(&btree,strEnglish);
        if(strChinese ! = "")
            cout<<"英文单词"<<strEnglish<<"的中文解释为:"<<strChi-
            nese<<endl;
        else
            cout<<"英文单词"<<strEnglish<<"不存在!"<<endl;
    }
    return 0;
}
```

习　　题

(1)简述树、二叉树、满二叉树和完全二叉树的结构特性。

(2)解释结点的度、树的度、结点的层、树的深度、分支、路径、路径长度、树的路径长度、叶子结点、分支结点、内部结点、孩子、双亲、兄弟、堂兄弟、祖先、子孙、有序树、无序树和森林等基本术语的含义。

(3)简述二叉树的 5 条基本性质。

(4)简述二叉树的常用操作及各操作的含义。

(5)简述顺序表示的二叉树中各结点的编号规则。

(6)简述二叉链表表示和三叉链表表示的二叉树中结点的结构。

(7)简述二叉树的 4 种遍历方式及每一种遍历方式中结点的访问顺序。

(8)已知一棵二叉树的先序遍历结果为 A,B,D,G,C,E,F,H,I,中序遍历结果为 D,G,B,A,E,C,H,F,I,请给出该二叉树的后序遍历结果。

(9)已知一棵二叉树的中序遍历结果为 D,G,B,A,E,C,H,F,I,后序遍历结果为 G,D,B,E,H,I,F,C,A,请给出该二叉树的先序遍历结果。

(10)已知一棵二叉树的先序遍历结果为 A,B,D,G,C,E,F,H,I,后序遍历结果为 G,D,B,E,H,I,F,C,A,请给出该二叉树的中序遍历结果。

(11)简述哈夫曼树的结构特性。

(12)简述结点的权、结点的带权路径长度、树的带权路径长度等基本术语的含义。

(13)简述哈夫曼树的构造方法。

(14)简述哈夫曼码的作用及其编码方法。

(15)简述树的 4 种常用表示方式。

(16)简述森林转换为二叉树的具体步骤。

(17)简述二叉树转换为树或森林的具体步骤。

上机实习1　二叉树的操作

1)实习目的

(1)熟悉并掌握二叉树的结构特点。

(2)熟悉并掌握二叉树的基本操作。

(3)能够使用二叉树解决实际应用问题。

2)实习内容

(1)构建一棵链式表示的二叉树,其中每一结点保存一个整数,且任一结点中的整数值大于其左子树各结点中的整数值、小于其右子树各结点中的整数值。假设将值为43,56,37,28,17,39,22,70的各结点依次插入二叉树中,插入完毕后采用中序遍历方式输出二叉树中每一结点的整数值。

(2)构建一棵链式表示的二叉树,其中每一结点保存一名学生的信息(包括学号、姓名和成绩),且任一结点中学生的学号大于其左子树各结点中学生的学号,小于其右子树各结点中学生的学号。假设将以下6名学生信息依次插入二叉树中:("1102030","李刚",65)、("1102035","王涛",92)、("1102041","吴明",73)、("1102023","马洪",85)、("1102033","赵冰",90)、("1102045","陈立",88),插入完毕后分别在二叉树中查找学号为1102033和1102037的结点,若查找成功则将结点中保存的学生信息输出,否则输出"查找失败!"。

3)实习指导

问题1:插入整数X时,从根结点开始,若X小于当前结点的值,则X应插入当前结点的左子树中;否则X应插入当前结点的右子树中。重复该步骤直至应插入的子树为空,此时将X作为该子树的根结点插入二叉树中。

问题2:可定义一个Student类,仿照问题1的方法构建二叉树。当根据给定学号K查找结点时,从根结点开始,若K小于当前结点的学号,则应到当前结点的左子树中继续查找;若K大于当前结点的学号,则应到当前结点的右子树中继续查找。重复该步骤,直至K等于当前结点的学号,查找成功应将匹配结点的学生信息输出;若待查找的子树为空,则查找失败。

上机实习2　哈夫曼树和哈夫曼码的操作

1)实习目的

(1)熟悉并掌握哈夫曼树的构造方法。

(2)熟悉并掌握哈夫曼码的编码方法和解码方法。

2)实习内容

(1)假设要编码的字符集为{A，B，C，D，E，F}，各字符的出现次数为{20，5，13，8，23，3}，构造一棵哈夫曼树。

(2)利用第(1)题中构造的哈夫曼树，得到字符串 FACE 的哈夫曼编码，再将编码结果输入哈夫曼树中，得到解码结果 FACE。

3)实习指导

问题 1：按照 5.5.2 中介绍的哈夫曼树的构造方法编写程序。

问题 2：将一个字符串中每个字符的哈夫曼码按从左至右的顺序组合在一起就形成了一个字符串的哈夫曼码；解码时按照 5.5.3 中介绍的哈夫曼码的解码方法编写程序。