第5章 树和二叉树

CHAPTER

树与二叉树均属于树形结构，树形结构中的数据元素具有层次特性，适合表示数据元素之间具有一对多关系的数据对象。树与二叉树的定义是递归的，所以树与二叉树的许多问题可用递归方法解决。

5.1 树

树是 $n(n\geqslant 0)$ 个数据元素的有限集合，集合中的数据元素之间存在一对多的关系。$n=0$ 时，为空树。树结构具有以下特性。

(1) 非空树中唯一一个没有前驱的结点，为根结点。

(2) 非空树可以看成由根和根的互不相交的子树组成。

(3) 树中没有后继的结点，为叶结点。

(4) 树中元素具有层次结构。

5.2 二 叉 树

5.2.1 二叉树的存储定义

二叉树是度不超过 2 的有序树。二叉树的子树有左、右之分。满二叉树和完全二叉树适合顺序存储。常用的二叉树存储方式有二叉链表或三叉链表。二叉树的二叉链表结点结构如图 1.5.1 所示，存储定义如下：

```
template <class DT>
struct BTNode                    //结点定义
{
    DT data;                     //数据域
    BTNode * lchild;             //左孩子指针
    BTNode * rchild;             //右孩子指针
};
```

一个结点指针类型的变量，如 BTNode<DT> * bt，如果 bt 指向二叉

树的根，则可以标识该二叉树。图 1.5.1 为二叉树二叉链表存储示意图。

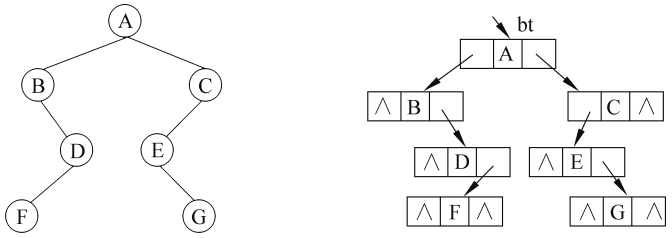

图 1.5.1　二叉树二叉链表存储示意图

5.2.2　二叉树操作实现原理

以下二叉树的遍历只考虑从左往右的顺序。

1. 先序递归遍历　void PreOrderBiTree(BTNode * bt)

先序遍历指按根(D)、左子树(L)、右子树(R)遍历二叉树和子树。先序遍历的递归算法步骤如下。

若二叉树为空，则返回；否则：a.访问根结点；b.先序遍历左子树；c.先序遍历右子树。

2. 中序递归遍历　void InOrderBiTree(BTNode * bt)

中序遍历指按左子树(L)、根(D)、右子树(R)遍历二叉树和子树。中序遍历的递归算法步骤如下。

若二叉树为空，则返回；否则：a. 中序遍历左子树；b. 访问根结点；c. 中序遍历右子树。

3. 后序递归遍历　void PostOrderBiTree(BTNode * bt)

后序遍历指按左子树(L)、右子树(R)、根(D)遍历二叉树和子树。后序遍历的递归算法步骤如下。

若二叉树为空，则返回；否则：a. 后序遍历左子树；b. 后序遍历右子树；c. 访问根结点。

4. 先序非递归遍历　void PreOrderBiTree_N(BTNode * bt)

先序遍历的顺序是根(D)、左子树(L)、右子树(R)。遍历从根开始，然后转向左子树，左子树遍历完遍历右子树。所以，在先序的非递归遍历中，需将根入栈，以便遍历完左子树后通过回溯找到右子树。先序遍历的非递归算法步骤如下。

Step 1　p 指向树根。

Step 2　p 非空或栈非空，重复下列操作：

2.1　访问 p 并将 p 入栈。

2.2　如果 p 有左孩子，转向 p 的左孩子(即 p=p->lchild)。

2.3　如果 p 没有左孩子，出栈至 p，执行下列操作：

2.3.1　如果出栈结点 p 有右孩子，转向 p 的右孩子。

2.3.2　否则，继续出栈。

当 p 和栈均为空时，遍历结束。

5. 中序非递归遍历　void InOrderBiTree_N(BTNode * bt)

中序遍历的顺序是左子树(L)、根(D)、右子树(R),一个结点被访问的前提是其左子树已遍历。所以,中序非递归遍历中需保存根,以便左子树遍历结束后通过回溯找到根以及之后转向根的右子树。中序遍历的非递归算法步骤如下。

Step 1　p 指向树根。

Step 2　只要结点 p 非空或栈不空,重复执行下列操作:

2.1　只要结点 p 有左孩子:

2.1.1　p 进栈。

2.1.2　左行,即 p=p->lchild。

2.2　栈非空,执行下列操作:

2.2.1　出栈至 p。

2.2.2　访问 p。

2.2.3　右行,即 p=p->rchild。

当 p 和栈均为空,遍历结束。

6. 后序非递归遍历　void PostOrderBiTree_N(BTNode * bt)

后序遍历的顺序是左子树(L)、右子树(R)、根(D),一个结点被访问的前提是其左、右子树均已被遍历。所以,后序非递归遍历中,需保存各结点,以便左子树访问完后通过回溯找到双亲的右子树和双亲。设置指针 r 指向刚被访问的结点,当栈顶点结点 p 的右孩子等于 r 时,表示 p 的左右子树均已遍历,p 可出栈访问。后序遍历的非递归算法步骤如下:

Step 1　p 指向根。

Step 2　重复下列操作,直至栈空:

2.1　p 非空,重复下列操作:

2.1.1　p 入栈。

2.1.2　左行,即 p=p->lchild。

2.2　处理栈顶元素的初始化:

2.2.1　r 为空,即 r=NULL。

2.2.2　设置标识 flag 为 1,表示可以连续处理从右子树上回溯的结点。

2.3　如果栈不空且访问标志 flag=1,重复下列操作:

2.3.1　获取栈顶元素 p。

2.3.2　如果 p->rchild==r,p 出栈,访问 p 结点,r 指向 p。

2.3.3　如果 p->rchild≠r,转向 p 的右孩子,访问标志 flag=0。

7. 层序遍历　void LevelOrderBiTree(BTNode * bt)

层序遍历指从上往下、从左往右依次访问各结点。操作步骤如下。

Step 1　p 指向根结点。

Step 2　p 非空,入队。

Step 3　队列非空,重复下列操作:

3.1　出队至 p。

3.2 访问 p->data。

3.3 如果 p 有左孩子,将 p 的左孩子(p->lchild)入队。

3.4 如果 p 有右孩子,将 p 的右孩子(p->rchild)入队。

退出 Step 3 循环时,队空,遍历结束。

8. 创建二叉树 void CreateBiTree(BTNode * &bt)

二叉树中的结点除根之外必须有唯一双亲,创建二叉树的结点,必须先创建其双亲结点(根除外)。所以,创建二叉树按先序遍历进行。操作步骤如下。

Step 1 输入数据元素值 e。

Step 2 如果值为空,根指针为空。

Step 3 如果值非空,创建一个结点 s,执行下列操作:

3.1 给 s 的值域赋值,即 s->data=e。

3.2 递归创建 s 的左子树。

3.3 递归创建 s 的右子树。

9. 销毁二叉树 void DestroyBiTree(BTNode * &bt)

二叉树的结点被销毁前应该先销毁其孩子结点,所以销毁二叉树按后序遍历进行。从根结点开始,操作步骤如下。

p 指向树根,如果 p 非空。

Step 1 递归销毁 p 的左子树。

Step 2 递归销毁 p 的右子树。

Step 3 销毁 p 结点。

10. 结点查询 BTNode* Search(BTNode * bt, DT e)

如果遍历中对元素的访问操作为结点元素值与查找值的比较,就可以实现结点的查询。先序、中序、后序或层序遍历均可以。以先序遍历为例,操作步骤如下。

Step 1 根结点 bt 空,返回空指针,表示未找到。

Step 2 如果 bt->data 等于查找值,返回 bt 值,表示结点所在位置。

Step 3 如果 bt->data 不等于查找值,到 bt 的左子树上递归查找:

3.1 若找到,返回结点位置。

3.2 若未找到,到 bt 右子树上递归查找,若找到,返回结点位置。

11. 计算二叉树深度 int Depth(BTNode * bt)

二叉树的深度为左、右子树深度的较大者加 1,其递归定义如下。

$$\text{depth(BTNode * bt)}=\begin{cases}0 & \text{bt=NULL}\\ \text{Max\{depth(bt->lchild),depth(bt->rvhild)\}}+1 & \text{其他}\end{cases}$$

因为要计算完左、右子树的高度,才能计算树的高度,所以,基于后序遍历思想求树的深度。操作步骤如下。

Step 1 空树,返回 0。

Step 2 非空树,进行下列操作:

2.1 递归求左子树深度 hl。

2.2 递归求右子树深度 hr。

2.3　树深为 max(hl,hr)+1。

12. 结点计数　int NodeCount(BTNode * bt)

二叉树由根、左子树和右子树构成，所以从递归角度看，二叉树的结点个数等于左子树结点个数+右子树结点个数+1(根结点)。结点计数的递归定义如下：

$$\begin{cases} 0 & bt=NULL \\ NodeCount(bt->lchild)+NodeCount(bt->rchild)+1 & 其他 \end{cases}$$

基于先序遍历的结点计数操作步骤如下。

Step 1　空树，返回 0。

Step 2　非空树，进行下列操作：

2.1　递归求左子树结点数。

2.2　递归求右子树结点数。

2.3　返回左、右子树结点数的和+1。

5.3　线索二叉树

利用二叉链表的空指针存储遍历序列的前驱或后继信息的二叉树，称为**线索二叉树**，被存储的前驱和后继信息，称为**线索**。

5.3.1　线索二叉树的存储定义

在线索二叉树中，用无左孩子结点的左孩子指针指向遍历的前驱结点，形成前驱线索；用无右孩子结点的右孩子指针指向遍历的后继结点，形成后继线索。为了区分左或右孩子指针是指向孩子结点还是遍历的前驱或后继，在线索二叉树的存储中另设了两个标志域来区分孩子信息和线索信息。线索二叉树存储定义如下：

```
template <class DT>
struct BiThrNode
{
  DT data;       //数据域
  int lflag;     //lfag=0,lchild 指向左孩子结点;lflag=1,lchild 指向遍历的前驱结点
  int rflag;     //rfag=0,rchild 指向右孩子结点;rflag=1,rchild 指向遍历的后继结点
  BiThrNode * lchild;        //左指针域
  BiThrNode * rchild;        //右指针域
};
```

中序线索二叉树存储示意图如图 1.5.2 所示。

5.3.2　线索二叉树操作实现原理

1. 线索化

给二叉树设置线索的过程称为**线索化二叉树**。进行二叉树的线索化时，设置指针 p 指向当前遍历结点，pre 为 p 的前驱，线索化的主要工作如下。

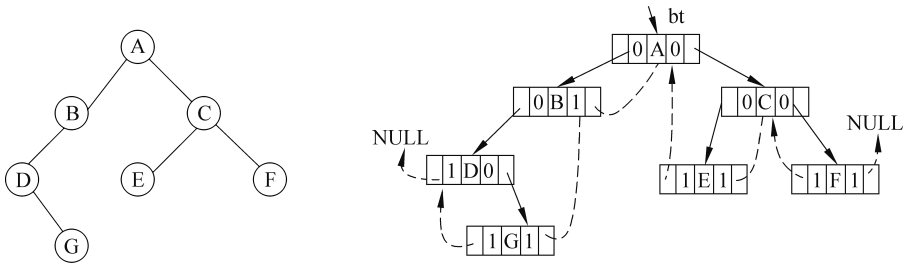

图 1.5.2　中序线索二叉树存储示意图

(1) 如果 p 没有左孩子,设置前驱线索标志(p->lflag=1),并把 p->lchild 指向 p 的前驱 pre;

(2) 如果 pre 没有右孩子,设置后继线索标志(p->rflag=1),并把 pre->rchild 指向 pre 的后继 p。

以中序遍历线索化二叉树为例,操作步骤如下。

Step 1　p 指向树根,pre=NULL。

Step 2　结点 p 非空,进行下列操作:

2.1　递归线索化 p 的左子树。

2.2　通过下列操作进行线索化:

2.2.1　如果 p 无左孩子,p->lflag =1; pre->lchild=pre。

2.2.2　如果 pre 无右孩子,pre->rflag=1,pre=p。

2.3　递归线索化 p 的右子树。

2. 先序遍历先序线索二叉树

先序线索二叉树上的后继线索给先序线索二叉树的先序遍历带来方便,整个过程不需要回溯。对于任一结点 p:

(1) 如果 p->rflag==1,p 的后继为线索所指结点(p->rchild);

(2) 否则,如果 p 有左孩子(即 p->lflag==0),p 的后继为 p 的左孩子(p->lchild);如果 p 无左孩子,有右孩子(即 p->rflag==0),p 的后继为 p 的右孩子(p->rchild)。

归纳起来,先序遍历先序线索二叉树的操作步骤如下。

Step 1　p 指向树根。

Step 2　p 非空,重复下列操作:

2.1　访问 p 结点。

2.2　如果 p->lflag==0,p=p->lchild;否则,p=p->rchild。

3. 中序遍历中序线索二叉树

中序线索二叉树上的后继线索给中序线索二叉树的中序遍历带来方便,整个过程不需要回溯。对于任一结点 p:

(1) 如果 p->rflag==1,则 p 的后继为后继线索所指结点(p->rchild);

(2) 否则(即 p 有右孩子),p 的后继为其右子树上中序遍历的第一点(即右子树上最左下的点)。

据此，中序遍历中序线索二叉树的操作步骤如下。

Step 1　p 指向树根。

Step 2　p 非空，重复下列操作：

2.1　只要 p->lflag==0，p=p->lchild。

2.2　访问 p 结点。

2.3　只要 p->rflag==1 且 p->rchild 非空，重复下列操作：

2.3.1　p=p->rchild。

2.3.2　访问 p 结点。

2.4　p=p->rchild。

5.4　最优二叉树

5.4.1　最优二叉树的存储定义和特性

最优二叉树指 *n* 个叶结点的二叉树中树的带权路径最小的二叉树。哈夫曼给出了最优二叉树的构造方法，最优二叉树也称为**哈夫曼树**。n 个叶结点的最优二叉树具有 $2n-1$ 个结点，可知的结点个数及构造的需要，最优二叉树采用顺序存储，存储结构定义如下：

```
struct HTNode              //结点结构
{
    int weight;            //权值域
    int parent;            //双亲结点在数组中的下标
    int lchild;            //左孩子结点在数组中的下标
    int rchild;            //右孩子结点在数组中的下标
};
```

最优二叉树具有以下特性：

(1) n 个叶结点的最优二叉树共有 $2n-1$ 个结点。

(2) 最优二叉树中没有度为 1 的结点。

5.4.2　最优二叉树的构建

构建最优二叉树的步骤如下。

Step 1　初始化。

1.1　所有结点的 parent、lchild、rchild 为 −1。

1.2　前 n 个结点的权值为叶结点权值，其余结点权值为 0。

Step 2　重复下列操作，求 $n-1$ 个中间结点。

2.1　从双亲为 −1 的前 k 个结点中，选择权值最小的两个结点，设下标为 $i1$，$i2$。

2.2　设生成的中间点序号为 k，则第 k 个结点的权值为第 $i1$，$i2$ 个结点权值的和。

2.3 下标 $i1$,$i2$ 分别为第 k 个结点的左、右孩子。

2.4 k 为第 $i1$,$i2$ 结点的双亲。

5.4.3 哈夫曼编码的构建

由最优二叉树生成的前缀编码为哈夫曼编码。每一个字符的编码如果用一个二进制串表示,可以设一个字符数组指针 char * HC[n],指向各编码字符串。

求解思路是依次以叶结点为出发点,向上回溯至根结点为止。回溯时走左分支则生成代码 0,走右分支则生成代码 1。按此方法求得的是低位到高位的各位编码。

因为各字符编码长度不一样,不适合预申请存储空间,计算中用一字符数组 char cd[n] 暂存计算结果;设指针 start 初始指向表尾,最先求得编码存在最低位,即 cd[--start] 中;当一个回溯结束时,将 start 开始的编码串复制到 HC[]中。

在哈夫曼树上求哈夫曼编码的操作步骤如下。

Step 1 初始化。

1.1 创建工作数组,即 cd=new char[n]。

1.2 添加字符串结束符,即 cd[n-1]='\0'。

Step 2 求每个叶结点的哈夫曼编码。

2.1 start 指向编码结束位置。

2.2 由第 i 个叶结点向上回溯,直到树根:

2.2.1 结点是双亲的左孩子,即 cd[--start]='0'。

2.2.2 结点是双亲的右孩子,即 cd[--start]='1'。

2.2.3 继续向上回溯。

2.3 为第 i 个字符编码分配空间,即 HC[i]=new char[n-start]。

2.4 把求得的编码复制到 HC[i],即 strcpy(HC[i],&cd[start])。

Step 3 释放工作数组空间,即 delete cd。