第5章 二 叉 树

树(tree)是一种非线性的数据结构,在计算机科学、电子工程、生物信息学等领域都有广泛的应用,例如计算机文件系统通常使用树结构来组织文件和目录的层次关系,网络中通常使用树结构来组织路由表,编程语言中的解析器通常使用树结构来解析程序的语法结构,数据库系统中的B树和B+树等被广泛用于实现高效的数据索引和查询,机器学习中决策树常用于分类等。本章的实验内容主要是二叉树的实现及其实际应用。

5.1 知 识 简 介


视频讲解

5.1.1 二叉树的定义和基本性质

二叉树(binary tree)是 $n(n\geqslant 0)$ 个结点所构成的集合,它或为空树($n=0$);或为非空树,对于非空树 T:

(1) 有且仅有一个称之为根的结点;

(2) 除根结点以外的其余结点分为两个互不相交的子集 T_1 和 T_2,分别称为 T 的左子树和右子树,且 T_1 和 T_2 本身均为二叉树。

简单讲,二叉树就是度不超过2的有序树。由于二叉树的结构简单,规律性强,且所有树都能转换为唯一对应的二叉树,为不失一般性,通常将普通树(多叉树)转换为二叉树来实现。

二叉树具有以下基本性质:

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

性质2:深度为 k 的二叉树至多有 2^k-1 个结点($k\geqslant 1$)。

性质3:对任意一颗二叉树 T,如果其终端结点(叶子结点)数为 n_0,度为2的结点数为 n_2,则 $n_0=n_2+1$。

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

性质5:如果将一棵有 n 个结点的完全二叉树(其深度为 $\lfloor\log_2 n\rfloor+1$)的结点按层序编号(从第1层到第 $\lfloor\log_2 n\rfloor+1$,每层从左往右),则对任一结点 $i(1\leqslant i\leqslant n)$,以下结论成立。

(1) 如果 $i=1$,则结点 i 是二叉树的根,无双亲;如果 $i>1$,则其双亲 Parent(i)是结点 $\lfloor i/2\rfloor$。

(2) 如果 $2i>n$,则结点 i 无左孩子(结点 i 为叶子结点);否则其左孩子 LChild(i)是结点 $2i$。

(3) 如果 $2i+1>n$,则结点 i 无右孩子;否则其右孩子 RChild(i)是 $2i+1$。

二叉树可以采用顺序存储和链式存储两种方式。


视频讲解

5.1.2 顺序存储

二叉树的顺序存储使用一组地址连续的存储单元来存储二叉树中的数据元素。将二叉

树的各个结点按完全二叉树的结点层次编号，依次存放二叉树中的数据元素。利用编号之间的关系来描述双亲与左孩子、右孩子之间的关系（即二叉树的性质 5）。如编号为 i 的结点，如果 i 不等于 1，则该结点不是根结点，有双亲，双亲的编号为 $i/2$。如果编号为 $2i$ 的结点存在，则编号为 $2i$ 的结点为编号为 i 的结点的左孩子。如果编号为 $2i+1$ 的结点存在，则编号为 $2i+1$ 的结点为编号为 i 的结点的右孩子。

二叉树的顺序存储定义如下：

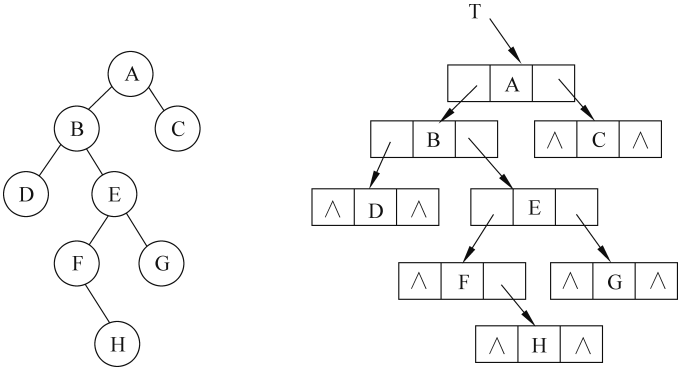
```
#define MAXTSZIE 100          //最大结点数
typedef TElemType SqBiTree[MAXTSIZE];
SqBiTree bt;
```

在顺序存储中，结点间关系蕴含在其存储位置中，寻找结点的双亲和孩子结点都非常方便。但对于普通的二叉树而言，顺序存储浪费空间，且不利于结点的插入和删除。顺序存储适用于存储完全二叉树。

5.1.3 链式存储

二叉树的链式存储又分为二叉链表和三叉链表两种方式。

二叉链表存储方式是每个结点不仅要存储数据元素的值，还需存储该结点左右孩子的地址。因此，每个结点包括三个部分：数据域（data）、指向左孩子指针（lchild）、指向右孩子指针（rchild）。链表的头指针指向二叉树的根结点。图 5-1(a)为一棵二叉树，图 5-1(b)为二叉树的二叉链表存储方式。



(a) 二叉树　　(b) 二叉链表存储方式

图 5-1　二叉树的二叉链表存储方式

二叉树的二叉链表存储结构描述如下：

```
typedef struct BiTNode
{
    TElemType data;                          //数据域
    struct BiTNode   * lchild,   * rchild;   //分别指向左右孩子的指针
}BiTNode, * BiTree;
```

三叉链表是在二叉链表的基础上加一个指向双亲的指针（parent）。

链式存储相较于顺序存储节省存储空间，插入删除结点时只需修改指针。但寻找指定

结点时很不方便。普通的二叉树一般用链式存储结构存储。

5.1.4 二叉树的遍历方式

二叉树的遍历方式主要有四种，包括先序遍历、中序遍历、后序遍历和层序遍历。

先序遍历(pre-order traversal)：先访问根结点，然后遍历左子树，最后遍历右子树。在遍历左、右子树时，仍然先访问根结点，然后遍历左子树，最后遍历右子树。图 5-1(a)表示的二叉树先序遍历的结果为：A B D E F H G C。

中序遍历(in-order traversal)：先遍历左子树，然后访问根结点，最后遍历右子树。在遍历左、右子树时，仍然先遍历左子树，然后访问根结点，最后遍历右子树。图 5-1(a)表示的二叉树中序遍历的结果为：D B F H E G A C。

后序遍历(post-order traversal)：先遍历左子树，然后遍历右子树，最后访问根结点。在遍历左、右子树时，仍然先遍历左子树，然后遍历右子树，最后访问根结点。图 5-1(a)表示的二叉树序后遍历的结果为：D H F G E B C A。

层序遍历(level-order traversal)：从上到下逐层遍历，在同一层中按照从左到右的顺序遍历。图 5-1(a)表示的二叉树序层次遍历的结果为：A B C D E F G H。

5.2 实 验 目 的

通过本章的实验，加深对二叉树的定义、性质、二叉链表存储方式以及遍历操作等知识的理解，培养学生用树结构解决实际问题的能力，同时锻炼学生实际编程和算法设计的能力。

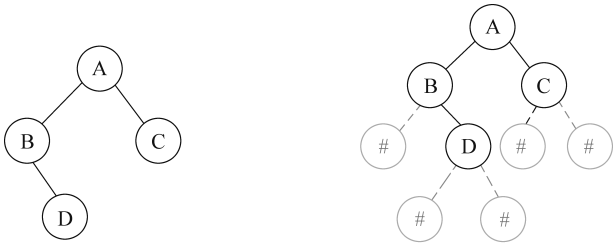
5.3 实 验 范 例

范例 1　采用二叉链表存储结构建立一棵二叉树


视频讲解

要求：假设二叉树的数据元素是字符，根据输入的一棵二叉树的扩展先序遍历序列建立一棵以二叉链表表示的二叉树。

所谓二叉树的扩展是指用一个特殊的符号(如"＃")表示空树。图 5-2(a)所示的二叉树对应的扩展二叉树如图 5-2(b)所示。扩展二叉树的先序遍历序列为：A B ＃ D ＃ ＃ C ＃ ＃。



(a) 二叉树　　(b) 扩展二叉树

图 5-2　二叉树扩展二叉树

1. 问题分析

要求利用扩展二叉树的先序遍历序列建立二叉树。首先分析序列 A B # D # # C # #，因为它是先序遍历的结果，所以该序列是按根、左子树、右子树的顺序得到的。因此，按该序列创建二叉树时也可以先创建根、然后创建左子树，再创建右子树。由图 5-2(b)可以看出，标 # 号的结点类似于指向 NULL 的指针，因此 # 表示的是一棵空树。

根据以上分析，创建二叉树的算法步骤如下：

(1) 输入的字符 ch。

(2) 如果 ch 为"#"，则返回空。

(3) 如果 ch 不为"#"，则生成结点，将字符存入结点的数据域，然后递归建立左子树，再递归建立右子树。

2. 算法描述

二叉表结构定义如下：

```
typedef char TElemType;
typedef struct BiTNode
{
    TElemType data;                                    //结点数据
    struct BiTNode * lchild;                           //左孩子
    struct BiTNode * rchild;                           //右孩子
} BiTNode, * BiTree;
```

创建二叉树后，应将根结点的地址返回给主调函数。因此，设计一个返回根结点指针的函数 CreateBiTree()，函数可定义如下：

```
BiTree CreateBiTree()
{
    char ch;                                           //存储结点的值
    BiTree root;
    scanf("%c",&ch);
    if(ch == "#")                                      //输入的是"#"
        root = NULL;
    else
    {
        root = (BiTNode *)malloc(sizeof(BiTNode));//生成新的结点
        root ->data = ch;                              //将输入的字符存入结点
        root ->lchild = CreateBiTree();                //递归建立左子树
        root ->rchild = CreateBiTree();                //递归建立右子树
    }
    return root;
}
```

3. 算法分析

创建二叉树的时间复杂度为 $O(n)$，其中 n 为输入序列的长度。

范例 2　实现二叉树的先序、中序、后序遍历递归算法

对二叉树进行先序、中序和后序遍历操作，并输出遍历序列，观察输出的序列是否与逻

辑上的序列一致。

1. 问题分析

可以用递归的方法对二叉树进行遍历，先序、中序、后序三种遍历方式如下。

(1) 先序遍历：若二叉树为空树，则空操作；否则先访问根结点，再先序遍历左子树，最后先序遍历右子树。

(2) 中序遍历：若二叉树为空树，则空操作；否则先中序遍历左子树，再访问根结点，最后中序遍历右子树。

(3) 后序遍历：若二叉树为空树，则空操作；否则先后序遍历左子树，再后序遍历右子树，最后访问根结点。

2. 算法描述

采用递归的方式先序遍历二叉树 T，设计函数如下：

```
void PreOrderTraverse(BiTree T)
{
    if(T == NULL)
        return;
    else
    {
        printf("%c ",T->data);            //访问根结点
        PreOrderTraverse(T->lchild);      //递归先序遍历左子树
        PreOrderTraverse(T->rchild);      //递归先序遍历右子树
    }
}
```

观察以上函数的内部结构可知，如果 T 等于 NULL 就返回，否则就执行 else 后的 3 条语句。这个逻辑可以换一种说法：如果 T 不等于 NULL 就执行原 else 后的 3 条语句，否则就返回。因此，先序遍历函数可以修改如下：

```
void PreOrderTraverse(BiTree T)
{
    if(T != NULL)
    {
        printf("%c ",T->data);            //访问根结点
        PreOrderTraverse(T->lchild);      //递归先序遍历左子树
        PreOrderTraverse(T->rchild);      //递归先序遍历右子树
    }
}
```

中序遍历二叉树 T，递归函数如下：

```
void InOrderTraverse(BiTree T)
{
    if(T!=NULL)
    {
        InOrderTraverse(T->lchild);       //递归中序遍历左子树
        printf("%c ",T->data);            //访问根结点
```

```
        InOrderTraverse(T->rchild);          //递归中序遍历右子树
    }
}
```

后序遍历二叉树 T，递归函数如下：

```
void PostOrderTraverse(BiTree T)
{
    if(T!=NULL)
    {
        PostOrderTraverse(T->lchild);        //递归后序遍历左子树
        PostOrderTraverse(T->rchild);        //递归后序遍历右子树
        printf("%c ",T->data);               //访问根结点
    }
}
```

在主函数 main()中，先创建一棵二叉树，然后进行先序、中序、后序三种方式的遍历。

```
int main()              //主函数
{
    BiTree T;
    printf("输入扩展后的二叉树的先序遍历序列: ");
    T=CreateBiTree();
    printf("先序遍历序列为:");
    PreOrderTraverse(T);
    printf("\n");
    printf("中序遍历序列为:");
    InOrderTraverse(T);
    printf("\n");
    printf("后序遍历序列为:");
    PostOrderTraverse(T);
    printf("\n");
    return 0;
}
```

3. 算法分析

二叉树的三种遍历算法的时间复杂度为 $O(n)$。

范例 3　实现二叉树先序遍历的非递归操作

要求：设计一个函数，用非递归的方法实现二叉树的先序遍历。

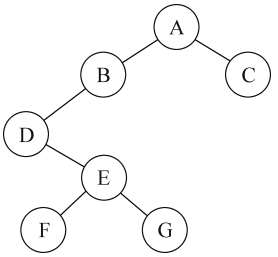


图 5-3　一棵二叉树

1. 问题分析

二叉树先序遍历的非递归过程需要用栈来辅助实现。如图 5-3所示的二叉树，先序遍历时应该从根结点 A 开始，然后访问它的左子树，再访问它的右子树。

因此，先序遍历该二叉树的过程为：首先访问根结点 A，并将 A 入栈，接着访问 A 的左子树。在访问 A 的左子树时，首先访问左子树的根结点 B，并将 B 入栈，然后再访问 B 的左子树。先访问 B 的左子树的根结点 D，并将结点 D 结点入栈。接着应

访问 D 的左子树，由于 D 的左子树为空，此时栈中的数据如图 5-4 所示。栈顶为结点 D，D 不需要再访问，应该接着访问 D 的右子树。此时将栈顶的结点 D 出栈，找到 D 的右子树。D 的右子树的根为 E，访问结点 E，并将其入栈。接着访问 E 的左子树的根结点 F，并将 F 入栈，由于 F 没有左子树，将 F 出栈，由于 F 没有右子树，以 F 为根结点的二叉树访问完成。将栈顶结点 E 出栈，找到 E 的右子树，右子树根结点为 G，访问 G，同时入栈。由于 G 没有左子树，将 G 出栈，找到 G 的右子树，G 的右子树为空，以 G 为根结点的二叉树访问完毕。接着将栈顶元素 B 出栈，B 的右子树为空，则以 B 为根结点的二叉树访问完。接着将栈顶元素 A 出栈，找到 A 的右子树。A 的右子树根结点为 C，访问 C，并将 C 入栈。C 没有左子树，将 C 出栈，然后访问 C 的右子树，C 的右子树也为空。最后栈为空，整个遍历过程结束。

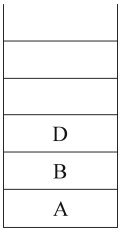


图 5-4　访问 D 结点之后栈中的数据

对上述案例的分析总结如下：

找到某个结点，访问该结点并将其入栈。当找某个结点的左子树或右子树为空，说明以某个结点为根结点的二叉树访问完毕，也就是栈顶结点的左子树访问完毕。接着将栈顶结点出栈，找到其右子树，如果此时栈为空，说明二叉树所有结点访问完毕。

2. 算法描述

根据上述分析，先序遍历的非递归过程描述如下：

(1) 初始化一个空栈 S，指针 p 指向根结点；

(2) 当 p 非空或者栈 S 非空时，循环执行以下操作：

① 如果 p 非空，访问 p 所指元素，并将 p 进栈，p 指向该结点的左孩子；

② 如果 p 为空，则栈顶元素出栈，将 p 指向出栈结点的右孩子。

具体代码描述如下：

```
void PreOrderTraverse_NoRecur(BiTree T)
{   //先序遍历二叉树 T 的非递归算法
    InitStack(S);
    p = T;
    while(p || !StackEmpty(S))
    {
        if(p){                                  //p 非空
              cout <<p->data;                   //访问根结点
              Push(S, p);                       //根指针进栈
              p = p->lchild;                    //遍历左子树
        }
        else {                                  //p 为空
              Pop(S, q);                        //栈顶出栈
              p = q->rchild;                    //遍历右子树
        }
    }
}
```

可以在范例 2 主函数的基础上，调用该函数对二叉树 T 进行非递归遍历，并比较递归遍历的序列和非递归遍历的序列是否一致。

3. 算法分析

由于每个结点只会被访问一次，该算法的时间复杂度是 $O(n)$，其中 n 是树中结点的数量。

5.4 实 验 任 务

任务 1：根据二叉树的先序遍历序列和中序遍历序列建立二叉树。

要求：设计一个函数，根据两种遍历序列建立二叉树。例如，已知先序遍历序列 ABCDEFGH 和中序遍历序列 BDCEAFHG，建立该二叉树。

任务 2：实现二叉树中序遍历的非递归操作。

要求：设计一个函数，用非递归方法实现对任务 1 建立的二叉树进行中序遍历。

任务 3：给定两棵二叉树，判断两棵二叉树是否相等。

假设 $T1$ 和 $T2$ 是两棵二叉树，如果两棵二叉树都是空树；或者两棵树的根结点的值相等，并且根结点的左、右子树也分别相等，则称二叉树 $T1$ 与 $T2$ 是相等的。要求设计一个函数判断二叉树 $T1$ 和 $T2$ 是否相等。

任务 4(选做)：编程求从二叉树根结点到指定结点 p 之间的路径。

要求：根据给出的一棵二叉树的根结点和任意给定的结点 p，输出从根结点到该结点 p 的路径。

5.5 任 务 提 示

1. 任务 1 提示

先序遍历是先访问根结点，然后先序遍历左子树，再先序遍历右子树。根结点在最前面，根据先序遍历序列可以找到根结点。中序遍历是先中序遍历左子树，再访问根结点，最后中序遍历右子树。利用先序遍历序列找到根结点后，再利用中序遍历序列可以分出该二叉树的左子树和右子树的结点。左右子树再用同样的方法确定根结点以及根结点的左子树和右子树。

例如已知先序遍历序列 ABCDEFGH 和中序遍历序列 BDCEAFHG，建立该二叉树。

由先序遍历序列可知 A 是根结点，再由中序遍历序列可知 A 的左子树包括四个结点，且左子树的中序遍历序列为 BDCE，先序遍历序列为 BCDE，A 的右子树包括 3 个结点，且右子树的中序遍历序列为 FHG，先序遍历序列为 FGH，如图 5-5 所示。

接着用同样的方法建立 A 的左右子树。

A 的左子树中序遍历序列 BDCE，先序遍历序列 BCDE。先序遍历序列中 B 在最前面，即 B 是根结点，是 A 的左孩子。中序遍历序列中 B 是第一个，前面没有元素，说明 B 没有左子树，右子树由 3 个结点组成，分别是 D、C、E，中序遍历序列为 DCE，先序遍历序列为 CDE，如图 5-6 所示。

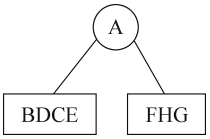


图 5-5　根据中序和先序建立二叉树(一)

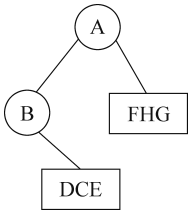


图 5-6　根据中序和先序建立二叉树(二)

接着建立 B 的右子树,B 的右子树中序遍历序列为 DCE,先序遍历序列为 CDE。由先序遍历序列可知,根结点为 C,由中序遍历序列可知,C 有左子树,左子树一个结点 D,即为 C 的左孩子。C 有右子树,右子树一个结点 E,即为 C 的右孩子。此时 A 的左子树建立完毕,如图 5-7 所示。接着建立 A 的右子树。

A 的右子树中序遍历序列是 FHG,先序遍历序列是 FGH。由先序遍历序列可知 F 是根结点,由中序遍历序列可知 F 没有左子树,右子树有两个结点,右子树的中序遍历序列为 HG,先序遍历序列为 GH,如图 5-8 所示。

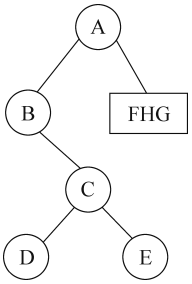


图 5-7　根据中序和先序建立二叉树(三)

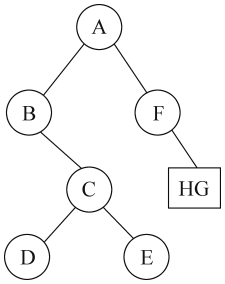


图 5-8　根据中序和先序建立二叉树(四)

F 的右子树中序遍历序列为 HG,先序遍历序列为 GH。由先序遍历序列可知 G 是根结点,即 G 是 F 的右孩子。由中序遍历序列可知,G 有左子树,左子树只有结点 H,H 就是 G 左孩子,G 没有右孩子。二叉树建立完毕,如图 5-9 所示。

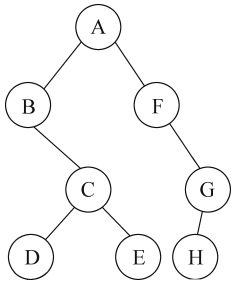


图 5-9　根据中序和先序建立二叉树(五)

建立过程中由先序遍历序列可知根结点,再由中序遍历序列可知根结点的左子树结点和右子树的结点。左子树和右子树可以用相同的方法建立,用递归来实现。

已知中序遍历序列 in[$i1..i1+len-1$]和先序遍历序列 pre[$i2..i2+len-1$],有 len 个结点,递归建立二叉树 T 的过程如下:

① 如果序列长度 len 为 0，T=NULL，否则执行以下操作；

② 生成根结点 T，将 pre 的第一个元素存入 T 的数据域；

③ 找到 pre 第一个元素在 in 中的位置 m；

④ 计算左子树个数 leftLen 和右子树的个数 rightLen；

⑤ 递归建立 T 的左子树，中序遍历序列为 in[$i1..m-1$]，先序遍历序列为 pre[$i2+1..i2+$leftLen]，左子树的结点个数为 leftLen；

⑥ 递归建立 T 的右子树，中序遍历序列为 in[$m+1..i1+$len-1]，先序遍历序列为 pre[$i2+$leftLen$+1..i2+$len-1]，右子树的结点个数为 rightLen。

伪代码描述如下：

```
typedef char TElemType;
typedef struct BiTNode
{
    TElemType data;                              //数据域
    struct BiTNode * lchild, * rchild;           //分别指向左右孩子的指针
}BiTNode, * BiTree;
void CreateTree(BiTree &T, char * in, int i1, char * pre, int i2, int len)
{
    if(len <= 0) T = NULL;                       //长度小于 1,递归结束
    else
    {
        T = (BiTNode * )malloc(sizeof(BiTNode));//生成根结点
        T->data = pre[i2];                       //将先序序列中的第一个元素放入数据域
        //找到先序序列第一个元素在中序序列中的位置 m
        for(m = i1;m <i1+len; m++)
            if(pre[i2] == in[m]) break;
        //计算左子树的个数
        leftLen = m - i1;
        //计算右子树的个数
        rightLen = len - (leftLen + 1);
        //递归建立左子树
        CreateTree(T->lchild, in, i1,pre,i2+1,leftLen);
        //递归建立右子树
        CreateTree(T->rchild, in, m+1, pre, i2+leftLen+1, rigthLen);
    }
}
```

2. 任务 2 提示

中序遍历的非递归过程和先序遍历的非递归类似，先序遍历过程中找到一个结点，就访问该结点，并将该结点入栈。中序遍历过程中，找到一个结点，由于需先访问其左子树，该结点先不能访问，直接将其入栈。当该结点的左子树访问完毕，回到该结点时再访问该结点，并找到该结点的右子树。所以在中序遍历的非递归过程中，当 p 不为空时，将结点入栈，当 p 为空，将栈顶元素出栈时再访问该出栈的结点。

中序遍历的非递归过程如下：