第三讲 树（上）

3.1 树与树的表示

1.查找

查找是指根据某个给定关键字 K，从集合 R 中找出关键字与 K 相同的记录。它分为以下两类：

（1）静态查找：集合中记录是固定的，没有插入和删除操作。

（2）动态查找：集合中记录是动态变化的，除了查找，还可能发生插入和删除。

首先举一个**顺序查找**的例子。此例需要注意，其设置了一个哨兵，因此可以减少判断中的一个条件。这个例子要求是在 Element[1]~Element[n]中查找关键字为 K 的数据元素，其结构体如下：

```
typedef struct LNode *List;
struct LNode{
    ElementType Element[MAXSIZE];
    int Length;
};
```

则查找算法代码如下：

```
int SequentialSearch(List Tb1, ElementType K)
{
    int i;
    Tb1->Element[0]=K;/*建立哨兵*/
    for(i-Tb1->Length;Tb1->Element[i]!=K;i--);
    return i;/*查找成功返回下标，不成功返回 0*/
}
```

时间复杂性 **O(n)。**

再举一个**二分查找**的例子。二分查找遵循以下要求：假设 n 个数据元素的关键字满足有序，并且是连续存放。

例如，假设有 13 个数据元素，按照关键字由小到大顺序存放，二分查找关键字为 444 的数据。

5	16	39	45	51	98	100	202	226	321	368	444	501

理论步骤是如表 1.1 所示。

表 1.1 二分法步骤

left	right	mid	value	comparsion
1	13	7	100	100<444，右侧
8	13	10	321	321<444，右侧
11	13	12	444	444=444，结束

程序代码如下：

```
int BinarySearch(List Tbl, ElementType K)
{
    int left, right, mid, NoFount = -1;
```

```
    left = 1;/*初始左侧边界*/
    right = Tbl->Length;/*初始右侧边界*/
    while (left <= right)
    {
        mid = (left + right) / 2;
        if (K < Tbl->Element[mid])
            right = mid - 1;
        else if (K > Tbl->Element[mid])
            left = mid + 1;
        else return mid;
    }
    return NotFound;
}
```

时间复杂度是 **O(logN)**。

2.树的定义和术语

树（Tree）：n（n≥0）个结点构成的有限集合。当 n=0 时，称为空树。

对于任一棵非空树（n>0），它具备以下性质：

（1）树种有一个称为根的特殊结点，用 r 表示；

（2）其余结点可分为 m（m＞0）个互不相交的有限集 T1，T2，…，Tm，其中每个集合本身又是一棵树，称为原来树的“子树”。

（3）子树是不相交的；

（4）除了根结点外，每个结点有且只有一个父节点；

（5）一棵 N 个结点的树有 N-1 条边。

树的一些基本术语：

（1）结点的度：结点子树个数。

（2）树的度：树的所有结点中最大的度数。

（3）叶结点：度为 0 的结点。

（4）父结点：有子树的结点是其子树的根结点的父结点。

（5）子结点：若 A 结点是 B 结点的父结点，则称 B 结点是 A 结点的子结点。

（6）兄弟结点：具有同一父结点的各结点彼此是兄弟结点。

（7）路径和路径长度：从结点 n1 到 nk 的路径为一个结点序列 n1，n2，…nk，其中 ni 是 ni+1 的父结点。路径所包含边的个数为路径的长度。

（8）祖先结点：沿树根到某一结点路径上的所有结点都是这个结点的祖先结点。

（9）子孙节点：某一结点的子树中的所有结点是这个结点的子孙。

（10）结点的层次：规定根结点在 1 层，其它任一结点的层数是其父结点的层数加 1。

（11）树的深度：树种所有结点总的最大层次是这棵树的深度。

3.树的表示

数组、链表都是很难表示树的，因为树的结构是多变的，无法统一规定。因此在这里用“**儿子-兄弟表示法**”，结构如下面所示。

<table><tr><td colspan="2">Element</td></tr><tr><td>FirstChild</td><td>NextSibling</td></tr></table>

如图 1 左所示的树结构可以表示为图 1 右所示的表示法。

图 1

把这个树往右转向 45°，可以看出，这是一个度为 2 的树。这种方法也被称为——**二叉树**。

3.2 二叉树

1.二叉树的定义

二叉树 T：一个有穷的结点集合。这个集合可以为空。若不为空，则它是由根结点和称为其左子树 TL 和右子树 TR 的两个不相交的二叉树组成。

二叉树的五种基本形态如图 2 所示，几种特殊二叉树如图 3 所示。

图 2

图 3

对完全二叉树左一个说明。完全二叉树可以是满二叉树缺少最低一层最右侧几个，只要前面都对的起来即可，但是**绝不能在中间缺几个（如图 3 右下角）**。

2.二叉树的几个重要性质

（1）一个二叉树第 i 层的最大结点数为：2^{i-1}，i≥1。

（2）深度为 k 的二叉树最大节点数是 2^{k}-1，k≥1。

（3）对任何非空二叉树 T，若 n0 表示叶结点的个数，n2 是度为 2 的非叶结点个数，那么两者满足 n0=n2+1。

3.二叉树的抽象数据类型定义

类型名称：二叉树

数据对象集：一个有穷的结点集合。若不为空，则由根结点和其左、右二叉子树组成。

操作集：BT∈BinTree，Item∈ElementType，重要的操作有：

（1）Boolean IsEmpty(BinTree BT)：判断 BT 是否为空；

（2）void Traversal(BinTree BT)：遍历，按某顺序访问每个结点；

（3）BinTree CreatBinTree()：创建一个二叉树。

二叉树常用的便利方法有：

（1）void PreOrderTraversal(BinTree BT)：先序——根、左子树、右子树；

（2）void InOrderTraversal(BinTree BT)：中序——左子树、根、右子树；

（3）void PostOrderTraversal(BinTree BT)：后序：左子树、右子树、根；

（4）void LevelOrderTraversal(BinTree BT)：层次遍历：从上到下、从左到右。

4.二叉树的存储结构

（1）顺序存储结构

完全二叉树：按从上至下、从左到右顺序存储，n 个结点的完全二叉树的结点父子关系，可以使用顺序存储结果。如图 4 所示。

图 4

这个完全二叉树很有规律，很容易找到其父结点和子结点。我们可以发现：

①非根节点（序号 i>1）的父结点的序号是[i/2]（在这里[]表示取整）；

②结点（序号为 i）的左孩子结点序号为 2i（若 2i>=n，就没有左孩子）；

③结点（序号为 i）的右孩子结点序号为 2i+1（若 2i+1>=n，就没有右孩子）。

推广来说，一般二叉树也可以采用这种结构，方法就是通过补空位使其变成完全二叉树。但这样的结果是会造成空间浪费。

（2）链表存储

一般二叉树的结点可以用下面的方式表示：

Left	Data	Right

其代码表示如下：

```
typedef struct TreeNode *BinTree;
typedef BinTree Position:
struct TreeNode{
	ElementType Data;
	BinTree Left;
	BinTree Right;
};
```

5.二叉树的递归遍历

（1）先序遍历

遍历过程为：

①访问根结点；②先序遍历其左子树，先序遍历其右子树。

程序如下所示：

```
void PreOrderTraversal(BinTree BT)
{
	if(BT){
		printf("%d",BT->Data);
		PreOrderTraversal(BT->Left);
		PreOrderTraversal(BT->Right);
	}
}
```

（2）中序遍历

遍历过程为：

①中序遍历其左子树；②访问根结点；③中序遍历其右子树。

程序如下所示：

```
void InOrderTraversal(BinTree BT)
{
	if(BT){
		PreOrderTraversal(BT->Left);
		printf("%d",BT->Data);
		PreOrderTraversal(BT->Right);
	}
}
```

（3）后序遍历

便利过程：

①后续遍历其左子树；②后续遍历其右子树；③访问根结点。

代码如下：

```
void PostOrderTraversal(BinTree BT)
{
	if(BT){
		PreOrderTraversal(BT->Left);
```

```
        PreOrderTraversal(BT->Right)
        printf("%d",BT->Data);;
    }
}
```

6.二叉树的非递归遍历（以中序为例）

基本思路：使用堆栈。

如图 5 所示的二叉树，我们以此未来来介绍利用堆栈进行处理的方法，输出顺序为：DBEFAGHCI。处理的方法见表 3.1 所示。

图 5

表 3.1 使用堆栈算法的表格示意

序号	当前结点	堆栈（从左到右相当于从底到顶）	步骤说明
1	A	[A]	首先把 A 压入堆栈
2	B	[AB]	B 为 A 的左孩子，压入堆栈
3	D	[ABD]	D 为 B 的左孩子，压入堆栈
4		[AB]	D 无孩子，将 D 推出堆栈
5		[A]	**将 B 推出堆栈**
6	F	[AF]	由于 F 为 B 右孩子，在 B 推出时将 F 压入堆栈
7	E	[AFE]	E 为 F 左孩子，压入堆栈
8		[AF]	E 无孩子，将 **E 推出堆栈**
9		[A]	**将 F 推出堆栈**
10		[]	**将 A 推出堆栈**
11	C	[C]	将 G 压入堆栈
12	G	[CG]	G 为 C 的左孩子，压入堆栈
13		[C]	G 没有左孩子，**G 推出堆栈**
14	H	[CH]	将 G 的右孩子压入堆栈
15		[C]	H 无孩子，**H 推出堆栈**
16		[]	**将 C 推出堆栈**
17	I	[I]	将 I 压入堆栈
18		[]	**将 I 推出堆栈**

具体执行代码如下：

```
void InOrderTraversal(BinTree BT)
{
    BinTree T=BT;
    Stack S=CreatStack(MaxSize);/*创建并初始化堆栈 S*/
    while(T||!IsEmpty(S)){
        while(T)
        {/*一致向左将沿途结点压入堆栈*/
            Push(S,T);
            T=T->Left;
        }
        if(!IsEmpty(S))
        {
```

```
            T= Pop(S);/*结点弹出堆栈*/
            printf("%5d",,T->Data);/*打印结点*/
            T=T->Right;/*转向右子树*/
        }
    }
}
```

同理可得，中序遍历为：

```
void PreOrderTraversal(BinTree BT)
{
    BinTree T=BT;
    Stack S=CreatStack(MaxSize);/*创建并初始化堆栈 S*/
    while(T||!IsEmpty(S)){
        while(T)
        {/*一致向左将沿途结点压入堆栈*/
            printf("%5d",,T->Data);/*打印结点*/
            Push(S,T);
            T=T->Left;
        }
        if(!IsEmpty(S))
        {
            T= Pop(S);/*结点弹出堆栈*/
            T=T->Right;/*转向右子树*/
        }
    }
}
```

7.层序遍历

层次遍历是利用队列的方式进行。遍历从根结点开始，首先将根结点指针入队，然后开始执行下面三个操作：

（1）从队列中取出一个元素；

（2）访问该元素所指结点；

（3）若钙元素所指结点的左、右孩子结点费控，则将其左、右孩子的指针顺序入队。

不断执行这三步操作，直到队列为空，再无元素可取，二叉树的层序遍历就完成了。其代码如下所示：

```
void LevelOrderTraversal(BinTree BT)
{
    BinTree T;
    Queue Q;
    if(!BT) return;/*空树直接返回*/
    Q = CreatQueue(MaxSize);
    AddQ(Q,BT);
    while(!IsEmptyQ(Q)){
        T = DeleteQ(Q);
        printf("%d\n",T->Data); /*访问取出队列的结点*/
        if(T->Left)
```

```
                AddQ(Q,T->Left);
            if(T->Right)
                AddQ(Q,T->Right);
        }
}
```

8.遍历应用的例子

【例 1】遍历二叉树的应用：输出二叉树中的叶子结点。

```
void PreOrderTraversal(BinTree BT)
{
        if(BT){
            if(!BT->Left &&!BT->Right)
                printf("%d",BT->Data);
            PreOrderTraversal(BT->Left);
            PreOrderTraversal(BT->Right);
        }
}
```

【例 2】求二叉树的高度。

```
int PostOrderGetHeight(BinTree BT)
{
        int HL,HR,MaxH;
        if(BT){
            HL = PostOrderGetHeight(BT->Left);
            HR = PostOrderGetHeight(BT->Right);
            MaxH = (HL>HR)?HL:HR;
            return MaxH+1;
        }
        else return 0;
}
```

【例 3】二元运算表达式树及其遍历

如图 6 所示。

三种遍历可以得到三种不同的访问结果：

（1）中序遍历得到中缀表达式：a + b * c + d * e + f * g

（2）先序遍历得到前缀表达式：+ + a * b c * + * d e f g

（3）后序遍历得到后缀表达式：a b c * + d e * f + g * +

由此直接将表达式存入链表就可以了。

【例 4】由两种遍历序列确定二叉树：已知三种遍历中的**任意两种**遍历序列，能否**唯一确定**一棵二叉树呢？

只需要确保其中一个是**中序遍历**，就可以唯一确定。如果只有前序遍历、后序遍历，这就不可能找到唯一的。

下面以先序和中序遍历序列来确定一棵二叉树。

先序遍历序列的第一个结点就是根结点。这个根结点能够在中序遍历序列中将其余结点分割成两个子序列，根结点前面部分是左子树上的结点，而根结点后面的部分是右子树上的结点。

根据这两个子序列，在先序序列中找到对应的左子序列和右子序列，它们分别对应左子

树和右子树。

然后对左子树和右子树分别递归使用相同的方法继续分解。

9.二叉树的创建

由于树是非线性结构，创建一棵二叉树必须首先确定树种结点的输入顺序，常用的方法是**先序创建**和**层序创建**两种。