阅读：25,304　　作者：解学武

线索二叉树的创建及遍历(C语言实现)

通过前面对二叉树的学习，了解到二叉树本身是一种非线性结构，采用任何一种遍历二叉树的方法，都可以得到树中所有结点的一个线性序列。在这个序列中，除第一个结点外，每个结点都有自己的直接前趋；除最后一个结点外，每个结点都有一个直接后继。

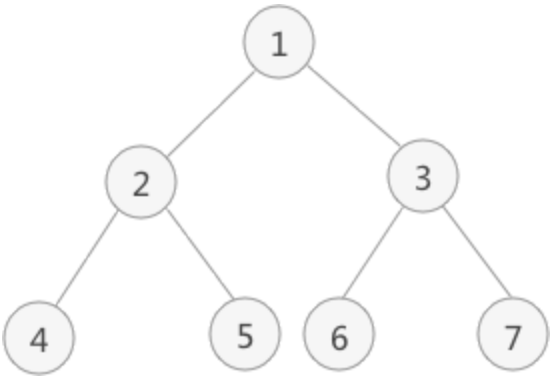

图1 满二叉树

例如，图 1 采用先序遍历的方法得到的结点序列为：`1 2 4 5 3 6 7`，在这个序列中，结点 2 的直接前趋结点为 1，直接后继结点为 4。

什么是线索二叉树

如果算法中多次涉及到对二叉树的遍历，普通的二叉树就需要使用栈结构做重复性的操作。

线索二叉树不需要如此，在遍历的同时，使用二叉树中空闲的内存空间记录某些结点的前趋和后继元素的位置（不是全部）。这样在算法后期需要遍历二叉树时，就可以利用保存的结点信息，提高了遍历的效率。使用这种方法构建的二叉树，即为“线索二叉树”。

线索二叉树的结点结构

如果在二叉树中想保存每个结点前趋和后继所在的位置信息，最直接的想法就是改变结点的结构，即添加两个指针域，分别指向该结点的前趋和后继。

但是这种方式会降低树存储结构的存储密度。而对于二叉树来讲，其本身还有很多未利用的空间。

> 存储密度指的是数据本身所占的存储空间和整个结点结构所占的存储量之比。

每一棵二叉树上，很多结点都含有未使用的指向NULL的指针域。除了度为2的结点，度为 1 的结点，有一个空的指针域；叶子结点两个指针域都为NULL。

> 规律：在有 n 个结点的二叉链表中必定存在 n+1 个空指针域。

线索二叉树实际上就是使用这些空指针域来存储结点之间前趋和后继关系的一种特殊的二叉树。

线索二叉树中，如果结点有左子树，则 lchild 指针域指向左孩子，否则 lchild 指针域指向该结点的直接前趋；同样，如果结点有右子树，则 rchild 指针域指向右孩子，否则 rchild 指针域指向该结点的直接后继。

为了避免指针域指向的结点的意义混淆，需要改变结点本身的结构，增加两个标志域，如图 2 所示。

图2 线索二叉树中的结点结构

LTag 和 RTag 为标志域。实际上就是两个布尔类型的变量：

- LTag 值为 0 时，表示 lchild 指针域指向的是该结点的左孩子；为 1 时，表示指向的是该结点的直接前趋结点；
- RTag 值为 0 时，表示 rchild 指针域指向的是该结点的右孩子；为 1 时，表示指向的是该结点的直接后继结点。

结点结构代码实现：

```
#define TElemType int//宏定义，结点中数据域的类型
//枚举，Link为0，Thread为1
typedef enum PointerTag{
    Link,
    Thread
}PointerTag;
//结点结构构造
typedef struct BiThrNode{
    TElemType data;//数据域
    struct BiThrNode* lchild,*rchild;//左孩子，右孩子指针域

    PointerTag Ltag,Rtag;//标志域，枚举类型
}BiThrNode,*BiThrTree;
```

表示二叉树时，使用图 2 所示的结点结构构成的二叉链表，被称为线索链表；构建的二叉树称为线索二叉树。

> 线索链表中的“线索”，指的是链表中指向结点前趋和后继的指针。二叉树经过某种遍历方法转化为线索二叉树的过程称为线索化。

对二叉树进行线索化

将二叉树转化为线索二叉树，实质上是在遍历二叉树的过程中，将二叉链表中的空指针改为指向直接前趋或者直接后继的线索。

> 线索化的过程即为在遍历的过程中修改空指针的过程。

在遍历过程中，如果当前结点没有左孩子，需要将该结点的 lchild 指针指向遍历过程中的前一个结点，所以在遍历过程中，设置一个指针（名为 pre ），时刻指向当前访问结点的前一个结点。

代码实现（拿中序遍历为例）：

```
01.  //中序对二叉树进行线索化
02.  void InThreading(BiThrTree p){
03.      //如果当前结点存在
04.      if (p) {
05.          InThreading(p->lchild);//递归当前结点的左子树，进行线索化
06.          //如果当前结点没有左孩子，左标志位设为1，左指针域指向上一结点 pre
07.          if (!p->lchild) {
08.              p->Ltag=Thread;
09.              p->lchild=pre;
10.          }
11.          //如果 pre 没有右孩子，右标志位设为 1，右指针域指向当前结点。
12.          if (!pre->rchild) {
13.              pre->Rtag=Thread;
14.              pre->rchild=p;
15.          }
16.          pre=p;//线索化完左子树后，让pre指针指向当前结点
17.          InThreading(p->rchild);//递归右子树进行线索化
18.      }
19.  }
```

> 注意：中序对二叉树进行线索化的过程中，在两个递归函数中间的运行程序，和之前介绍的中序遍历二叉树的输出函数的作用是相同的。

将中间函数移动到两个递归函数之前，就变成了前序对二叉树进行线索化的过程；后序线索化同样如此。

使用线索二叉树遍历

图 3 中是一个按照中序遍历建立的线索二叉树。其中，实线表示指针，指向的是左孩子或者右孩子。虚线表示线索，指向的是该结点的直接前趋或者直接后继。

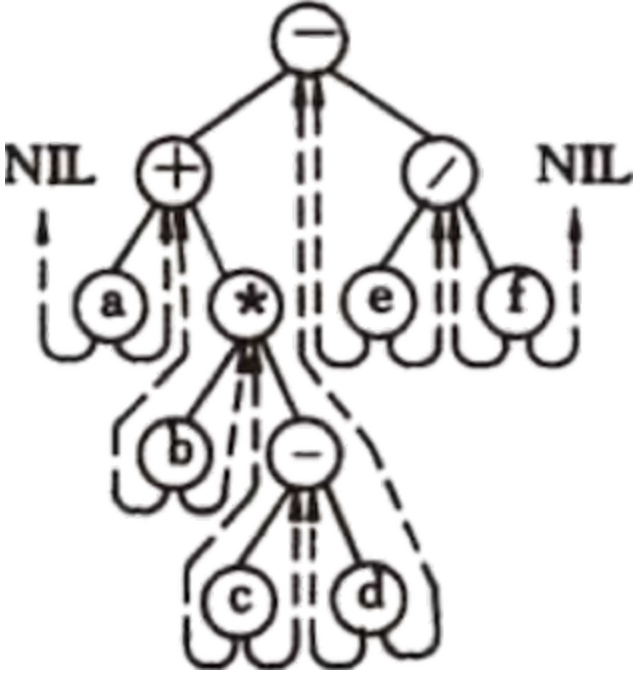

图 3 线索二叉树

使用线索二叉树时，会经常遇到一个问题，如图 3 中，结点 b 的直接后继直接通过指针域获得，为结点 * ；而由于结点 * 的度为 2 ，无法利用指针域指向后继结点，整个链表断掉了。当在遍历过程，遇到这种问题是解决的办法就是：寻找先序、中序、后序遍历的规律，找到下一个结点。

在先序遍历过程中，如果结点因为有右孩子导致无法找到其后继结点，如果结点有左孩子，则后继结点是其左孩子；否则，就一定是右孩子。拿图 3 举例，结点 + 的后继结点是其左孩子结点 a ，如果结点 a 不存在的话，就是结点 * 。

在中序遍历过程中，结点的后继是遍历其右子树时访问的第一个结点，也就是右子树中位于最左下的结点。例如图 3 中结点 * ，后继结点为结点 c ，是其右子树中位于最左边的结点。反之，结点的前趋是左子树最后访问的那个结点。

后序遍历中找后继结点需要分为 3 种情况：

1. 如果该结点是二叉树的根，后继结点为空；
2. 如果该结点是父结点的右孩子（或者是左孩子，但是父结点没有右孩子），后继结点是父结点；
3. 如果该结点是父结点的左孩子，且父结点有右子树，后继结点为父结点的右子树在后序遍历列出的第一个结点。

> 使用后序遍历建立的线索二叉树，在真正使用过程中遇到链表的断点时，需要访问父结点，所以在初步建立二叉树时，宜采用三叉链表做存储结构。

遍历线索二叉树非递归代码实现：

```
01.  //中序遍历线索二叉树
02.  void InOrderThraverse_Thr(BiThrTree p)
03.  {
04.      while(p)
```

```
    {
        //一直找左孩子，最后一个为中序序列中排第一的
        while(p->Ltag == Link){
            p = p->lchild;
        }
        printf("%c ", p->data);  //操作结点数据
        //当结点右标志位为1时，直接找到其后继结点
        while(p->Rtag == Thread && p->rchild !=NULL){
            p = p->rchild;
            printf("%c ", p->data);
        }
        //否则，按照中序遍历的规律，找其右子树中最左下的结点，也就是继续循环遍历
        p = p->rchild;
    }
}
```

整节完整代码（可运行）

```
#include <stdio.h>
#include <stdlib.h>
#define TElemType char//宏定义，结点中数据域的类型
//枚举，Link为0，Thread为1
typedef enum {
    Link,
    Thread
}PointerTag;
//结点结构构造
typedef struct BiThrNode{
    TElemType data;//数据域
    struct BiThrNode* lchild,*rchild;//左孩子，右孩子指针域
    PointerTag Ltag,Rtag;//标志域，枚举类型
}BiThrNode,*BiThrTree;

BiThrTree pre=NULL;

//采用前序初始化二叉树
//中序和后序只需改变赋值语句的位置即可
void CreateTree(BiThrTree * tree){
    char data;
    scanf("%c",&data);
    if (data!='#'){
        if (!((*tree)=(BiThrNode*)malloc(sizeof(BiThrNode)))){
            printf("申请结点空间失败");
            return;
        }else{
            (*tree)->data=data;//采用前序遍历方式初始化二叉树
```

```
            CreateTree(&((*tree)->lchild));//初始化左子树
            CreateTree(&((*tree)->rchild));//初始化右子树
        }
    }else{
        *tree=NULL;
    }
}
//中序对二叉树进行线索化
void InThreading(BiThrTree p){
    //如果当前结点存在
    if (p) {
        InThreading(p->lchild);//递归当前结点的左子树，进行线索化
        //如果当前结点没有左孩子，左标志位设为1，左指针域指向上一结点 pre
        if (!p->lchild) {
            p->Ltag=Thread;
            p->lchild=pre;
        }
        //如果 pre 没有右孩子，右标志位设为 1，右指针域指向当前结点。
        if (pre&&!pre->rchild) {
            pre->Rtag=Thread;
            pre->rchild=p;
        }
        pre=p;//pre指向当前结点
        InThreading(p->rchild);//递归右子树进行线索化
    }
}
//中序遍历线索二叉树
void InOrderThraverse_Thr(BiThrTree p)
{
    while(p)
    {
        //一直找左孩子，最后一个为中序序列中排第一的
        while(p->Ltag == Link){
            p = p->lchild;
        }
        printf("%c ", p->data);  //操作结点数据
        //当结点右标志位为1时，直接找到其后继结点
        while(p->Rtag == Thread && p->rchild !=NULL)
        {
            p = p->rchild;
            printf("%c ", p->data);
        }
        //否则，按照中序遍历的规律，找其右子树中最左下的结点，也就是继续循环遍历
        p = p->rchild;
    }
}

```

```
76.  int main() {
77.      BiThrTree t;
78.      printf("输入前序二叉树:\n");
79.      CreateTree(&t);
80.      InThreading(t);
81.      printf("输出中序序列:\n");
82.      InOrderThraverse_Thr(t);
83.      return 0;
84.  }
```

运行结果

```
输入前序二叉树:
124###35##6##
输出中序序列:
4 2 1 5 3 6
```