



南昌大学

实 验 报 告

实验课程： 数据结构与算法实验

实验题目： 二叉树线索化相关操作

任课老师： 吴之旭老师

学生姓名： 殷骢睿

学 号： 5509121041

专业班级： 2021 级人工智能实验班

2022 年 11 月 29 日



南昌大学实验报告

学生姓名: 殷懿睿 学号: 5509121041 专业班级: 人工智能 211 班
实验类型: ☐ 验证 ☐ 综合 ☒ 设计 ☐ 创新 实验日期: 2022.11.29 实验成绩: _____

一、实验目的

- (1) 掌握二叉树线索化的建立
- (2) 掌握二叉树线索化的遍历
- (3) 掌握二叉树线索化的销毁

二、实验内容

具体请完成如下功能:

- (1) 设计算法实现对二叉树的链式存储
- (2) 中序遍历二叉树
- (3) 销毁二叉树

三、实验要求

- (1) 程序要添加适当的注释,程序的书写要采用**缩进格式**。
- (2) 程序要具有一定的**健壮性**,即当输入数据非法时,程序也能适当地做出反应,如查找的内容不存在时,要给予提示等等。
- (3) 程序要做到**界面友好**,在程序运行时用户可以根据相应的提示信息进行操作。
- (4) 根据实验报告模板详细书写实验报告,在实验报告中给出**流程图**。
- (5) **源程序和实验报告打包**。二叉树的头文件夹包含文件有: BinTreeThread.h; 二叉树的源程序文件夹包含文件有: BinTreeTread.c, main.c 等。头文件夹、源程序文件夹和实验报告压缩为一个文件,按以下方式命名: 学号姓名.rar, 如 0814101 王五.rar。

四、设计实现二叉树线索化程序主要实验步骤及程序分析

4.1 思路设计

遍历二叉树是以一定规则将二叉树中结点排列成一个线性序列,得到二叉树中节点的先序序列或中序序列或后序序列,这是一个对非线性结构进行线性化操作,使得除第一个和最后一个节点外的每一个结点在这些线性序列中有且仅有一个直接前驱和直接后继,

除每个节点在这些线性序列中有且只有一个直接前驱和直接后继，一个简单的办法是在每个节点中增加两个指针域，分别指示结点在按照任意次序遍历时得到的前驱和后继信息。在有 n 个结点的二叉链表中必定存在 $n+1$ 个空链域，可以利用这些空链域来大大降低存储的信息，中序线索树图例如下：

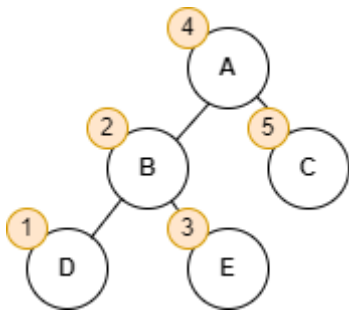


图 1 中序遍历按照 DBEAC 的顺序进行图

lchild	ltag	data	rtag	rchild
--------	------	------	------	--------

图 2 中序遍历线索化标志域图

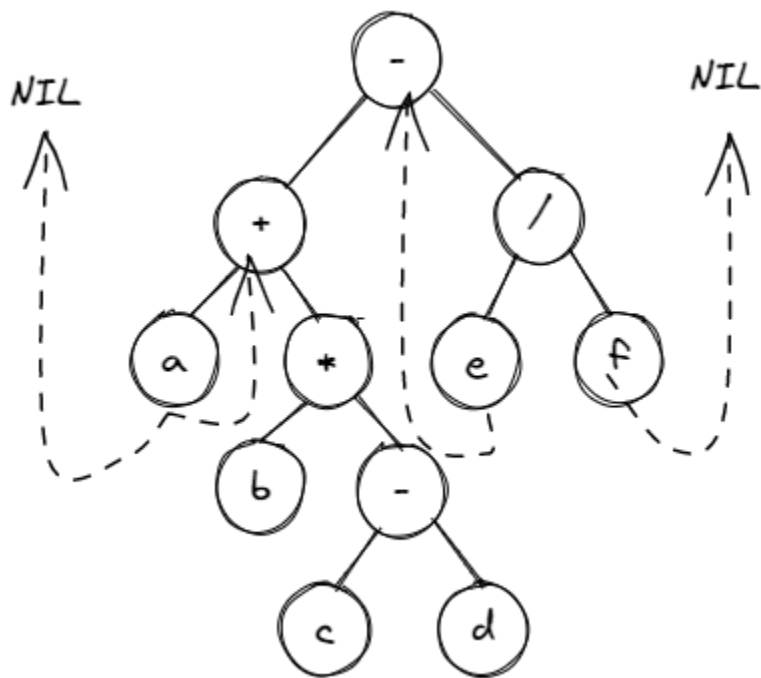


图 3 中序线索二叉树存储结构图

树中所有非终端结点的右链均为指针，则无法由此得到后继的信息，根据中序遍历的规律可知，结点的后继是遍历其右子树时访问的第一个结点。

在中序线索二叉树上遍历二叉树，时间复杂度为 $O(n)$ ，常数因子比非线索二叉树小，且

不需要设栈。因此，若在某程序中所用二叉树需经常遍历或查找结点在遍历所得线性序列中的前驱和后继，则应采用线索链表作存储结构。

4.2 主要具体步骤（完整代码见附录）

4.2.1 二叉线索存储表示和线索化二叉树

```
1.  /*中序遍历二叉树操作*/
2.  void InOrderNR(BiTree T) {
3.      BiTree stack[MAXLEN]; //定义顺序栈
4.      int top = 0;          //设置栈顶指针，空栈
5.      BiTree p = T;         //指向二叉树的根
6.
7.      while (p || top) {    //当前结点指针不为空或栈不为空时
8.          while (p) {       //顺着左链找最左下角结点
9.              stack[top++] = p; //不为空，入栈
10.             p = p->lchild;    //顺着左链找
11.         }
12.         if (top) {         //栈不为空
13.             p = stack[--top]; //出栈
14.             cout << p->data << " "; //访问
15.         }
16.         p = p->rchild; //处理右子树
17.     }
18. }
```

4.2.2 定义基础头文件

头文件设置如下：

```
1. #define TRUE 1
2. #define FALSE 0
3. #define OK 1
4. #define ERROR 0
5. #define INFEASIBLE -1
6. #define OVERFLOW -2
7. typedef int Status;
```

4.2.3 构建中序遍历二叉树

首先定义顺序栈，设置好栈顶指针，并且进行空栈操作，而后设置一个指针指向二叉树的根，进行相应的循环判断，若当前的结点指针不为空或栈不为空时，顺着左链开始找左下角的结点，利用队列的思想进行高效的循环遍历，详细相关步骤如源码注释所示：

```
1. void InOrderNR(BiTree T) {
2.     BiTree stack[MAXLEN]; //定义顺序栈
3.     int top = 0;          //设置栈顶指针，空栈
4.     BiTree p = T;         //指向二叉树的根
```

```

5.
6.     while (p || top) {      //当前结点指针不为空或栈不为空时
7.         while (p) {        //顺着左链找最左下角结点
8.             stack[top++] = p;    //不为空，入栈
9.             p = p->lchild;      //顺着左链找
10.        }
11.        if (top) {          //栈不为空
12.            p = stack[--top];      //出栈
13.            cout << p->data << " "; //访问
14.        }
15.        p = p->rchild; //处理右子树
16.    }
17. }
18.
19. void LevelOrder(BiTree T) {
20.     BiTree queue[MAXLEN] ; //定义循环队列
21.     int front, rear ;      //定义队列的队头和队尾指针
22.     BiTree p ;            //定义指向二叉树当前结点的指针
23.
24.     front = rear = 0 ;     //队列初始化，即空队列
25.     p = T ;
26.     rear = (rear + 1) % MAXLEN ;
27.     queue[rear] = p ;      //二叉树当前结点（指针）入队列
28.     while (front != rear) { //当队列非空
29.         front = (front + 1) % MAXLEN ;
30.         p = queue[front] ; //删除队头
31.         cout << p->data << " " ; //访问（即输出）
32.         if (p->lchild) {    //左孩子非空，入队列
33.             rear = (rear + 1) % MAXLEN ;
34.             queue[rear] = p->lchild ;
35.         }
36.         if (p->rchild) {    //右孩子非空，入队列
37.             rear = (rear + 1) % MAXLEN ;
38.             queue[rear] = p->rchild ;
39.         }
40.     }
41. }

```

4.2.4 销毁二叉树模块

先遍历清空左子树，再遍历清空右子树，而后释放根节点，即可销毁中序遍历二叉树。

```

1. void Clear(BiTree &T) {
2.     if (T) {
3.         Clear(T->lchild); //清空左子树
4.         Clear(T->rchild); //清空右子树

```

```

5.         delete T;           //释放根结点
6.         T = NULL;           //置空
7.     }
8. }
9. //销毁二叉树
10. void Destroy(BiTree &T) {
11.     Clear(T);
12. }

```

4.2.5 序列化二叉树和求二叉树叶子结点个数、深度模块

序列化二叉树和求二叉树叶子结点个数和深度算法如下，综合运用先序遍历、中序遍历和后续遍历，能够较为方便地得到二叉树的叶子结点个数和深度。

```

1. //序列化二叉树
2. void tostring(BiTNode *root, string &res) {
3.     if (root) {
4.         // res.push_back(' (');
5.         res += "(";
6.         res += root->data;
7.         tostring(root->lchild, res);
8.         tostring(root->rchild, res);
9.         // res.push_back(' ) ');
10.
11.         res += ")";
12.     }
13. }
14. string toSequence(BiTNode *root) {
15.     string res;
16.     tostring(root, res);
17.     return res;
18. }
19.
20. //调用函数时，参数 count 初始值为 0
21. void LeafCount(BiTree T, int &count) {
22.     if (T) {
23.         if (!T->lchild && !T->rchild)
24.             count++; //是叶子结点，计数器增 1
25.         LeafCount(T->lchild, count); //先序遍历左子树，计算叶子结点个数
26.         LeafCount(T->rchild, count); //先序遍历右子树，计算叶子结点个数
27.     }
28. }
29.

```

```

30. //求二叉树叶子结点个数
31. //利用后序遍历思想
32. int LeafCount(BiTree T) {
33.     int lnum, rnum ;
34.     if (!T)
35.         return 0 ;    //二叉树为空，则叶子结点 0 个
36.     else if (!T->lchild && !T->rchild)
37.         return 1 ; //二叉树只有根结点，则叶子结点 1 个
38.     else {
39.         lnum = LeafCount(T->lchild) ;    //后序遍历左子树，返回左子树叶子
        结点个数
40.         rnum = LeafCount(T->rchild) ;    //后序遍历右子树，返回右子树叶子
        结点个数
41.         return lnum + rnum ;            //访问根结点，计算二叉树结点个
        数
42.     }
43. }
44.
45. //求二叉树的深度
46. //利用后序遍历思想
47. int Depth(BiTree T) {
48.     int ldepth, rdepth ;
49.     if (!T)
50.         return 0 ;    //二叉树为空，则深度为 0
51.     else {
52.         ldepth = Depth(T->lchild) ;    //后序遍历左子树，计算左子树深度
53.         rdepth = Depth(T->rchild) ;    //后序遍历右子树，计算右子树深度
54.         //访问根结点，二叉树深度为左右子树大者加 1
55.         return (ldepth > rdepth ? ldepth : rdepth) + 1 ;
56.     }
57. }
58.
59. //求二叉树的深度
60. //调用之前 level 的初值为 1
61. //dval 的初值为 0
62. void Depth(BiTree T, int level, int &dval) {
63.     if ( T ) {
64.         //如果结点的当前层次大于深度，修改深度为当前层次值
65.         if (level > dval)
66.             dval = level;
67.         //先序遍历左子树，层次加 1
68.         Depth( T->lchild, level + 1, dval );
69.         //先序遍历右子树，层次加 1
70.         Depth( T->rchild, level + 1, dval );

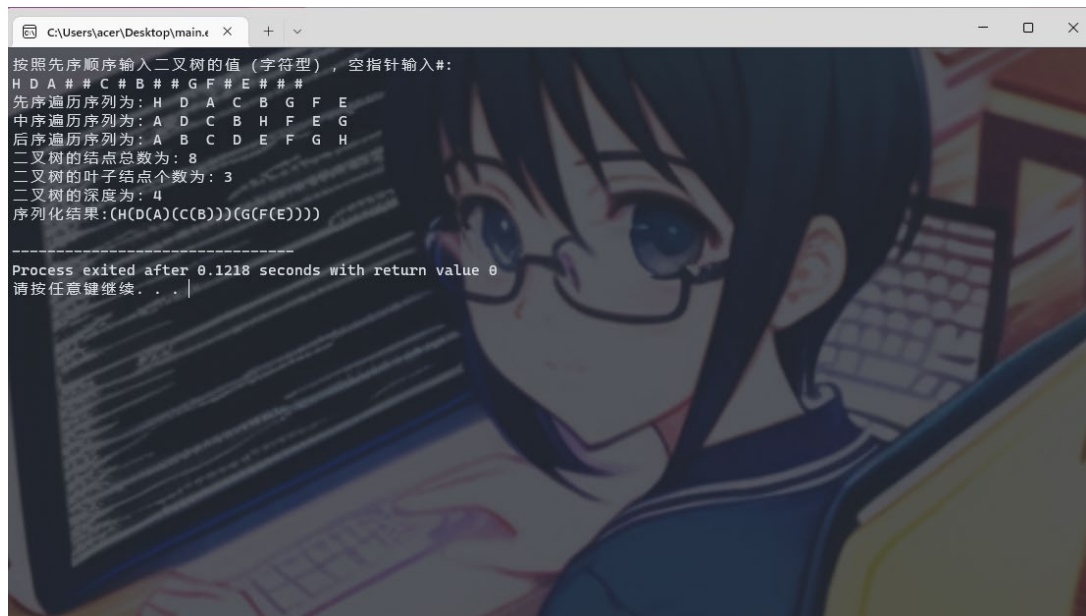
```

```
71.     }  
72. }
```

4.3 程序结果

输入：按照先序顺序输入二叉树的值（#作为空指针的提示标志符）。

输出：线索化先序遍历、中序遍历、后续遍历序列的结果、二叉树的结点总数、二叉树的叶子结点个数、二叉树的深度、序列化的结果。

The screenshot shows a Windows command prompt window with a dark background. The title bar reads 'C:\Users\acer\Desktop\main.cpp'. The output text is as follows:

```
按照先序顺序输入二叉树的值（字符型），空指针输入#：  
H D A # # C # B # # G F # E # # #  
先序遍历序列为：H D A C B G F E  
中序遍历序列为：A D C B H F E G  
后序遍历序列为：A B C D E F G H  
二叉树的结点总数为：8  
二叉树的叶子结点数为：3  
二叉树的深度为：4  
序列化结果：(H(D(A)(C(B))))(G(F(E)))  
-----  
Process exited after 0.1218 seconds with return value 0  
请按任意键继续. . .
```

图 4 程序结果图

五、编写代码时遇到的问题及实验心得体会

遇到的问题：

(1) 对指针理解不够，如在编写给每个结点设置树高时，之前的代码为：

```
1. void btHigh(node* root)  
2. {  
3.     if (root == NULL){  
4.         root->height = 0;  
5.         return ;//一定不要少了这个边界条件  
6.     }  
7.     btHigh(root->lchild);  
8.     btHigh(root->rchild);  
9.     //二叉树的高度为左子树和右子树中高的那个高度加 1  
10.    int ret1 = root->lchild->height;  
11.    int ret2 = root->rchild->height;
```



```
12.     root->height = ret1 > ret2 ? ret1 + 1 : ret2 + 1;
13. }
```

在此处有一个小错误，不能在 `root=NULL` 的时候给 `root->height` 赋值。

```
1.  if (root == NULL){
2.     root->height = 0;
3.     return ;//一定不要少了这个边界条件
4. }
```

(2) 开始对于遍历顺序的理论基础掌握不牢固，很容易犯错误，在这方面，编程时犯了类似的好几个错误，影响了编程的效率，但是好在写了代码之后完全掌握了相应的知识（果然理论和实际结合就是妙啊）。

实验心得体会：

(1)学习到了模块化思想解决树相关数据结构的问题的宝贵经验，发觉编写的程序要给人一种思路清晰的感觉，能够与用户进行交流（在此次课程设计中，与用户交流明显不足，这值得进一步改进），与此同时，还要注重程序的可读性、健壮性。

(2) 在编程时候应该很仔细，对于一般的语法错误应该尽量避免。同时对于一些模棱两可的想法应该尽快查阅书本，不能混过去，这样很容易导致在后期的调试过程中出现大量不可预期的错误。

(3) 这次编程坚决没有上网搜索现成的方案，锻炼了自己的意志力和克服困难的能力，很感谢老师的这次考验，精通数据结构不是一朝一夕所能完成的事情，需要锲而不舍的努力。我相信我有信心与毅力将这些知识理解透彻，提高自己的综合能力。

