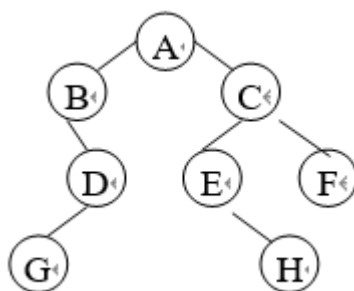


实验三 二叉树及其应用

实验要求

1. 以三叉链表存储二叉树
2. 依次输入二叉树的**中序**和**先序遍历**的结果来创建二叉树
 - 中序: B, G, D, A, E, H, C, F
 - 先序: A, B, D, G, C, E, H, F



3. 二叉树的遍历: 对所建的二叉树进行验证
 - 按**中序**遍历方法遍历该二叉树, 看遍历的结果是否与初始输入一致
4. 二叉树的应用: 线索二叉树的创建
 - 基于二叉树遍历思想的其它问题的求解: 扩展二叉树的存储结构, 增加表示直接**后继**线索的链域, 给出创建给定二叉树的后序线索化二叉树的程序
5. 线索二叉树的遍历: 编写在**后序**线索化树上的遍历算法
6. 二叉树创建的特例——表达式树
 - 实现输入为合法的**波兰式**来创建表达式树
7. 二叉树遍历的特例——表达式树
 - 针对用6创建的表达式树, 用**中序**遍历该树, 比较它与实际的中缀式之间的区别
8. 二叉树的应用——表达式转换
 - 输出表达式**逆波兰式**。

实验内容

1. 以三叉链表存储二叉树

每个二叉树的结点包含以下内容:

```

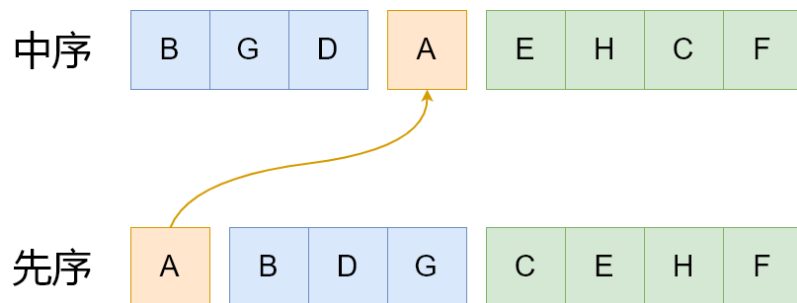
1 typedef struct BNode{
2     char key;           //关键字的值
3     struct BNode* left; //左子树
4     struct BNode* right; //右子树
5     struct BNode* parent; //父节点
6     int ltag;           //左孩子类型 (node/thread)
7     int rtag;           //右孩子类型 (node/thread)
8 }BNode, *Btree;

```

2.用中序和先序遍历结果创建二叉树

思路描述：

先序遍历第一个位置的元素是根结点。利用先序遍历第一个元素将中序遍历序列分成两个子序列，左边的是是根结点左子树的中序遍历，右边是根结点右子树的中序遍历。再根据两个子树的中序遍历序列的个数将先序遍历分割，得到两个子树的先序遍历结果，递归创建得到完整二叉树。



关键代码：

```

1 void MakeTree(char* in, char* pre, Btree* T, BNode* parent){
2     /*T is root,in is inorder,pre is preorder,parent is T's parent*/
3     /*make node for pre[0]*/
4     /*put new in-order and pre-order sequence of subtrees into in_new_left,
pre_new_left, in_new_right_pre_new_right*/
5     if(strlen(in_new_left)) MakeTree(in_new_left, pre_new_left, &((*T)->left), *T);
6     if(strlen(in_new_right)) MakeTree(in_new_right, pre_new_right, &((*T)->right), *T);
7 }

```

3.二叉树的遍历：对所建的二叉树进行验证

中序遍历：

```

1 void PrintInorder(Btree T){
2     /*judge whether '(' and ')' should be printed*/
3     if(leftflag) printf("(");
4     if(T->left) PrintInorder(T->left);
5     if(leftflag) printf(")");
6     printf("%c", T->key);
7     if(rightflag) printf("(");
8     if(T->right) PrintInorder(T->right);
9     if(rightflag) printf(")");
10    return;
11 }

```

先中序遍历左子树，再输出当前结点的key，再中序遍历右子树。

4.二叉树的应用：后继线索二叉树的创建

建立后继线索二叉树需要分很多种情况讨论。

总的来说：

前驱：

- 若右子树存在，前驱为右子树的根结点 (①)
- 若右子树不存在，且左子树存在，前驱为左子树的根结点（此时不用将此节点线索化，因为此节点左子树是node） (②)
- 若左右子树均不存在
 - 从此结点开始向上寻找直到找到一个结点，这个结点为右子树，且兄弟存在，前驱为此结点的兄弟。若找到了根，那么前驱为NULL (③)

```

1  if((*T)->left == NULL){//左子树为空，需要线索化
2      (*T)->ltag = thread;
3      BNode* temp = (*T)->parent;
4      if((*T)->right) (*T)->left = (*T)->right;//有右孩子，前驱为右孩子 (①)
5      else{
6          if(temp){//有父节点
7              temp = (*T);
8              //向上寻找 (③)
9              while(temp->parent){
10                 if(temp == temp->parent->right && temp->parent->ltag == node && temp->parent->left) break;
11                 temp = temp->parent;
12             }
13             if(temp->parent) (*T)->left = temp->parent->left;
14             else (*T)->left = temp->parent;
15         }
16         //无父节点
17         else (*T)->left = NULL;//根结点前驱为NULL
18     }
19 }

```

后继：

- 若此结点为右子树，后继为其父节点 (①)
- 若此结点为左子树且兄弟存在，后继为以其兄弟为根的子树的第一个结点 (②)
- 若此结点为左子树且兄弟不存在，后继为其父节点 (③)
- 若此结点为根，后继为NULL (④)

```
1  if((*T)->right == NULL){
2      (*T)->rtag = thread;
3      BNode* temp = (*T)->parent;
4      if(temp){
5          if(!flag){//T是左孩子
6              if(temp->right){//(②)
7                  temp = temp->right;
8                  (*T)->right = FIRST(temp);
9              }
10             //(③)
11             else (*T)->right = temp;
12         }
13         else{//T是右孩子 (①)
14             (*T)->right = temp;
15         }
16     }
17     else (*T)->right = NULL; //(④)
18 }
```

5.线索二叉树的遍历：编写在后序线索化树上的遍历算法

从根结点的第一个结点开始，按照Next(t)依次向后找后继

```
1  void PrintThread(Btree t){
2      if(t == NULL) return;
3      printf("%c", t->key);
4      PrintThread(Next(t));
5      return;
6  }
```

分情况讨论：

- 此节点rtag是thread，返回t->right (①)
- 此节点为根结点，返回NULL (②)
- 自己是右子树，返回父节点 (③)
- 自己是左子树，且兄弟存在，返回兄弟的第一个结点 (④)
- 自己是左子树，且兄弟不存在，返回父节点 (⑤)

```

1 BNode* Next(BNode* t){
2     if(t->rtag == thread) return t->right;//(①)
3     else {
4         if(t->parent == t) return NULL;//根结点 (②)
5         else{
6             if(t->parent && t->parent->rtag == node && t->parent->right == t) return
t->parent;//(③)
7             else if(t->parent && t->parent->rtag == node && t->parent->left == t)
return FIRST(t->parent->right);//(④)
8             else if(t->parent && t->parent->rtag == thread) return t->parent; //(⑤)
9             else return NULL;
10        }
11    }
12 }
13 }

```

6.二叉树创建的特例——表达式树

利用栈和波兰式创建表达式树。

大致思路：

将波兰式从后往前扫描，每扫描到一个就创建一个结点。若当前的值是非算符，将结点压栈，否则弹出两个结点，以当前结点为根，弹出的两个结点分别是两个子树连接起来，再将当前结点压栈，扫描完毕，栈中应存放表达式树的根结点。

```

1 for(i=len-1;i>=0;i--){
2     temp = MakeNode(exp[i]);
3     if(exp[i] != '+' && exp[i] != '-' && exp[i] != '*' && exp[i] != '/'){
4         Push(&S, temp);//找到非算符，压栈
5     }
6     else { //找到算符
7         BNode* t1 = Pop(&S);//弹出两个结点
8         BNode* t2 = Pop(&S);
9         temp->left = t1;//建成小数
10        temp->right = t2;
11        temp->ltag = node;
12        temp->rtag = node;
13        t1->parent = temp;
14        t2->parent = temp;
15        Push(&S, temp);//再压回去
16    }
17 }

```

7.二叉树遍历的特例——表达式树

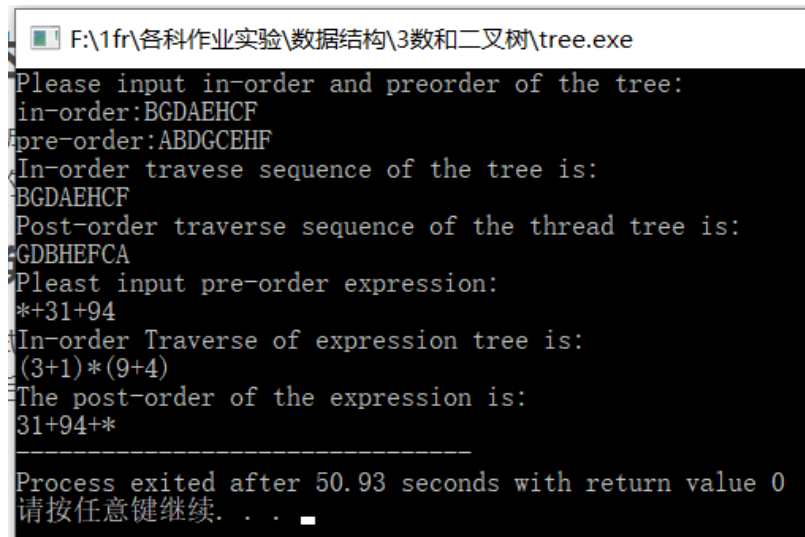
利用已经写好的PrintInorder遍历表达式树。遇到*和/会输出括号。

8.二叉树的应用——表达式转换

先将表达式树线索化，再利用线索二叉树输出后续遍历表达式树，即逆波兰式。

```
1 Thread(T_expression);
2 PrintThread(FIRST(*T_expression));
```

实验结果及分析



```
F:\1fr\各科作业实验\数据结构\3数和二叉树\tree.exe
Please input in-order and preorder of the tree:
in-order:BGDAEHCF
pre-order:ABDGCEHF
In-order traverse sequence of the tree is:
BGDAEHCF
Post-order traverse sequence of the thread tree is:
GDBHEFCA
Please input pre-order expression:
*+31+94
In-order Traverse of expression tree is:
(3+1)*(9+4)
The post-order of the expression is:
31+94+*
-----
Process exited after 50.93 seconds with return value 0
请按任意键继续. . .
```

第五行：正确输出中序序列

第七行：正确输出二叉树的后序遍历序列

第十一行：正确输出中缀表达式

第十三行：正确输出逆波兰式

实验小结

在本次实验中，我锻炼了将二叉树后续线索化的技能。我最大的感受是，在写代码前一定要想好写什么，怎样写，绝不能边写边想，这样反而会降低效率，越写越晕。后序线索化二叉树重点在于把所有情况都想到，把指针玩好。这并非易事，消耗了我一定时间