

《数据结构与算法分析》

实验报告

实验名称	基于二叉树的表达式求值算法
姓名	曾庆文
学号	23060218
院系	自动化学院
专业	人工智能
班级	23061011
实验时间	2024.5

一、实验目的

1. 掌握二叉树的二叉链表存储表示和二叉树的遍历等基本算法。
2. 掌握根据中缀表达式创建表达式树的算法。
3. 掌握基于表达式树的表达式求值算法

二、实验设备与环境

编辑器：Visual Studio Code

编译器：gcc

三、实验内容与结果

基本操作的输入输出结果如下：

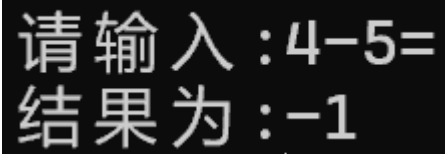
1. 测试样例：

```
请输入:2*(2+5)=  
结果为:14  
请输入:1+2=  
结果为:3  
请输入:=
```

```
请按任意键继续. . . |
```

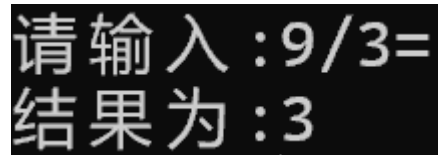
2. 补充测试:

2.1 结果为负的减法:



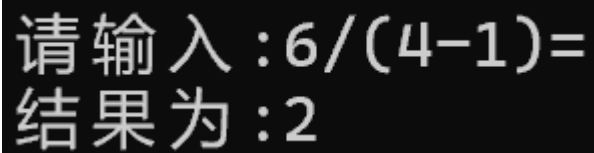
```
请输入:4-5=  
结果为:-1
```

2.2 除法:



```
请输入:9/3=  
结果为:3
```

2.3 有括号的表达式:



```
请输入:6/(4-1)=  
结果为:2
```

实验全部代码如下：

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #define maxsize 100
4.
5. // 二叉树的链式存储
6. typedef struct BiTNode
7. {
8.     char data;
9.     struct BiTNode *lchild, *rchild;
10. }BiTNode, *BiTree;
11.
12. // 栈的顺序存储
13. typedef struct
14. {
15.     BiTree *base;
16.     BiTree *top;
17.     int stacksize;
18. }SqStackN; // 结点栈结构体类型
19. typedef struct
20. {
21.     char *base;
22.     char *top;
23.     int stacksize;
24. }SqStackC; // 操作符栈结构体类型
25.
26.
27. /*-----树函数-----*/
28. void InitExpTree(BiTree *T);
29. void CreateExpTree(BiTree *T, BiTree a, BiTree b, char theta,
    a);
30. int EvaluateExpTree(BiTree T);
31. /*-----操作函数-----*/
32. int GetValue(char data, int lvalue, int rvalue);
33. char Precede(char theta1, char theta2);
34. int In(char ch);
35. /*-----栈函数-----*/
36. void InitStackN(SqStackN *s);
37. void PushN(SqStackN *s, BiTree e);
38. void PopN(SqStackN *s, BiTree *e);
39. void InitStackC(SqStackC *s);
40. void PushC(SqStackC *s, char e);
41. void PopC(SqStackC *s, char *e);
42. char GetTopC(SqStackC s);
```

```

43.
44.
45. // 程序主函数
46. int main()
47. {
48.     BiTree T = NULL;
49.     InitExpTree(&T);
50. }
51.
52. // 表达式树的创建算法
53. void InitExpTree(BiTree *T)
54. {
55.     SqStackN EXPT; // 结
                        点栈
56.     SqStackC OPTR; // 符
                        号栈
57.     InitStackN(&EXPT);
58.     InitStackC(&OPTR);
59.     PushC(&OPTR, '='); // 压
                        入 '=' 进符号栈
60.     char ch;
61.     char theta;
62.     BiTree a, b;
63.     a = (BiTree)malloc(sizeof(BiTreeNode));
64.     b = (BiTree)malloc(sizeof(BiTreeNode));
65.     printf("请输入:");
66.     scanf("%c", &ch);
67.     while (ch != '=')
68.     {
69.         while (ch != '=' || GetTopC(OPTR) != '=')
70.         {
71.             if (!In(ch)) //
                        ch 不是运算符, 是操作数
72.             {
73.                 CreateExpTree(T, NULL, NULL, ch); //
                        以 ch 为根创建一颗只有根结点的二叉树
74.                 PushN(&EXPT, *T); //
                        将创建的树压入结点栈
75.                 scanf("%c", &ch);
76.             }
77.             else // c
                        h 是运算符
78.             {
79.                 switch (Precede(GetTopC(OPTR), ch))

```

```

80.         {
81.             case '<':
82.                 PushC(&OPTR, ch);
83.                 scanf("%c", &ch);
84.                 break;
85.             case '>':
86.                 PopC(&OPTR, &theta);
87.                 PopN(&EXPT, &b); //
            根据栈的特性, 先弹出 b 再弹出 a
88.                 PopN(&EXPT, &a);
89.                 CreateExpTree(T, a, b, theta); //
            以 theta 为根结点, a, b 为左右子树
90.                 PushN(&EXPT, *T); //
            将新树的根节点入栈
91.             break;
92.             case '=':
93.                 PopC(&OPTR, &theta); // 弹
            出 '='
94.                 scanf("%c", &ch);
95.                 break;
96.         }
97.     }
98. }
99. printf("结果
    为:%d\n", EvaluateExpTree(*T)); // 计算结果并输出
100.    getchar(); //
    处理缓冲区
101.    printf("请输入:");
102.    scanf("%c", &ch);
103. }
104. }
105.
106. // 创建表达式树
107. void CreateExpTree(BiTree *T, BiTree a, BiTree b, char t
    heta)
108. {
109.     *T = (BiTree)malloc(sizeof(BiTreeNode));
110.     (*T)->data = theta; /
    / 以 theta 为根
111.     (*T)->lchild = a; /
    / a 为左子树
112.     (*T)->rchild = b; /
    / b 为右子树
113. }

```

```

114.
115. //遍历表达式树进行表达式求值
116. int EvaluateExpTree(BiTree T)
117. {
118.     int lvalue = 0;
119.     int rvalue = 0;
120.     if(T->lchild == NULL && T->rchild == NULL)      /
        / 结点为操作数，则返回值
121.     {
122.         return T->data - '0';
123.     }
124.     else
125.     {
126.         lvalue = EvaluateExpTree(T->lchild);          /
        / 递归求左子树的值
127.         rvalue = EvaluateExpTree(T->rchild);          /
        / 递归求右子树的值
128.         return GetValue(T->data, lvalue, rvalue);
129.     }
130. }
131.
132. //根据当前结点的运算符类型求值
133. int GetValue(char data, int lvalue, int rvalue)
134. {
135.     switch (data)
136.     {
137.         case '+':
138.             return lvalue + rvalue;
139.         case '-':
140.             return lvalue - rvalue;
141.         case '*':
142.             return lvalue * rvalue;
143.         case '/':
144.             return lvalue / rvalue;
145.     }
146.     return 0;
147. }
148.
149. //判定操作符栈的栈顶元素与读入的操作符之间优先关系的函数
150. char Precede(char theta1, char theta2)
151. {
152.     if((theta1=='(' && theta2==')') || (theta1=='=' && t
        heta2=='='))
153.         return '=';

```

```

154.         else if(theta1=='(' || theta1=='=' || theta2=='(' ||
            ((theta1=='+' || theta1=='-'
              ') && (theta2=='*' || theta2=='/'))))
155.             return '<';
156.         else
157.             return '>';
158.     }
159.
160.     //判断是不是操作符
161.     int In(char ch)
162.     {
163.         if(ch >= '0' && ch <= '9')    // 不是运算符
164.             return 0;
165.         else                            // 是运算符
166.             return 1;
167.     }
168.
169.     //结点栈初始化函数
170.     void InitStackN(SqStackN *s)
171.     {
172.         s->base=(BiTree *)malloc(sizeof(BiTree));
173.         if(!s->base){
174.             exit(0);
175.         }
176.         s->top=s->base;
177.         s->stacksize=maxsize;
178.     }
179.
180.     //结点栈入栈函数
181.     void PushN(SqStackN *s,BiTree e)
182.     {
183.         if(s->top - s->base == s->stacksize){
184.             exit(0);
185.         }
186.         *(s->top) = e;
187.         s->top++;
188.     }
189.
190.     //结点栈出栈函数
191.     void PopN(SqStackN *s,BiTree *e)
192.     {
193.         if(s->top == s->base){
194.             exit(0);
195.         }

```



```
196.         s->top--;
197.         *e = *(s->top);
198.     }
199.
200.
201.     //操作符栈初始化函数
202.     void InitStackC(SqStackC *s)
203.     {
204.         s->base=(char *)malloc(sizeof(char));
205.         if(!s->base){
206.             exit(0);
207.         }
208.         s->top=s->base;
209.         s->stacksize=maxsize;
210.     }
211.
212.     //操作符栈入栈函数
213.     void PushC(SqStackC *s,char e)
214.     {
215.         if(s->top-s->base==s->stacksize){
216.             exit(0);
217.         }
218.         *(s->top)=e;
219.         s->top++;
220.     }
221.
222.     //操作符栈出栈函数
223.     void PopC(SqStackC *s,char *e)
224.     {
225.         if(s->top==s->base){
226.             exit(0);
227.         }
228.         s->top--;
229.         *e=*(s->top);
230.     }
231.
232.     //取操作符栈栈顶元素函数
233.     char GetTopC(SqStackC s)
234.     {
235.         if(s.top!=s.base){
236.             return *(s.top-1);
237.         }
238.     }
```

四、实验总结及体会

本次实验对我个人来说有以下几个难点：

1. 创建表达式树：在生成表达式树时，需要将根结点与左右两颗子树连接。但是一开始我的代码给左右子树分配了两次空间，导致原先得到的子树被重置，多次 **debug** 后我才发现了这个问题并改了过来。
2. 栈的处理：本程序有两个不同数据类型的栈，一个是存储输入的操作符的符号栈，一个是存储创建好的树结点的结点栈。不过一开始我的代码错误的将结点栈写成存储 **int** 型的变量，这就导致代码不论怎么调试，怎么书写，总是有逻辑问题。在重新思考课本给出的伪码后，我才意识到其中一个栈存储的是结点而非整型变量。
3. 代码的分块书写与 **debug**：因为本次实验的函数比较多，所以我在每写完一个函数，就单独检测其是否能正确运行并得到正确结果。这个习惯大大减少了我 **debug** 的时间。