数据结构课程设计



班级： 1618403

学号： 161840227

姓名： 韦 鑫

指导教师：孙 涵

目录

1.采用的数据结构 ………………………………………… 3

2.算法设计思想 …………………………………………… 3

3.关键代码 ………………………………………………… 4

4.测试数据和结果 ………………………………………… 13

5.算法的时间复杂度及其改进方法 ……………………… 15

6.结束语 …………………………………………………… 15

一、采用的数据结构

栈、队列

//栈

typedef struct

{

node data[maxn];

int top;

}Stack;

//队列

typedef struct

{

node data[maxn];

int head, tail;

}Queue;

二、算法设计思想

①将算术表达式转化为逆波兰式：扫描输入字符串，若为操作数，则通过检测后边是否仍未操作数及小数点并记录位数，实现操作数的合并后直接入队；若为操作符且优先级大于栈顶操作符优先级，则操作符入栈，否则将栈顶元素弹出入队，直到栈顶操作符优先级小于输入操作符，（括号不入栈）

②计算逆波兰式：若队首为操作数，直接入栈；若为操作符，弹出栈中两操作数进行运算后将运算结果入栈，直到队空，输出栈顶元素值即为最后结果。

③表达式的错误检查分别在转化前、转换后、计算中进行：转换前，可对括号的匹配、异常字符的输入进行判断；转换后，通过统计操作数及操作符的数量，判断算术表达式是否正确；计算中若有除数为0，可认定输入算术表达式出错。

三、关键代码

//运行环境：Visual Studio 2017

#include<cstdio>

#include<iostream>

#include<cmath>

#include<cctype>

#include<cstring>

#pragma warning(disable:4996)

using namespace std;

const int maxn = 10000;

const double error = -(1 << 30 - 1);

//元素结构

struct node

{

double num; //操作数

char op; //操作符

bool flag; //1表示该元素为操作数，0表示为操作符

};

//栈

typedef struct

{

node data[maxn];

int top;

}Stack;

//栈的基本操作

void InitStack(Stack & s){s.top = -1;}

void push(Stack & s,node a){s.data[++s.top] = a;}

void pop(Stack & s){if (s.top >= 0)s.top--;}

node gettop(Stack s){if (s.top >= 0)return s.data[s.top];}

bool sempty(Stack s)

{

if (s.top >= 0)return false;

else return true;

}

//队列

typedef struct

{

node data[maxn];

int head, tail;

}Queue;

//队列的基本操作

void InitQueue(Queue & q){q.head = q.tail = 0;}

void enque(Queue & q,node a){q.data[q.tail++] = a;}

void deque(Queue & q){if(q.head < q.tail)q.head++;}

node gethead(Queue q){if (q.head < q.tail)return q.data[q.head];}

bool qempty(Queue q)

{

if (q.head < q.tail)return false;

else return true;

}

//运算符优先级

//0,1,2,3,4,5 == +,-,\*,/,(,)

int OP[6] = { 1,1,2,2,0,0 };

//判断运算符下标

int index(char c)

{

switch (c)

{

case '+':return 0;

case '-':return 1;

case '\*':return 2;

case '/':return 3;

case '(':return 4;

case ')':return 5;

}

}

//表达式转换为后缀表达式

void Change(char str[],Queue & q)

{

Stack s;

node temp;

InitStack(s);//操作符栈

InitQueue(q);//队列存储后缀表达式

bool IsPoint = 0; //记录是否为小数点

int t = 0; //绝对值表示小数点后几位

printf("中缀表达式转后缀表达式的变化过程：\n");

for (int i = 0; str[i] != '\0';)

{

//操作数

if (str[i] >= '0' && str[i] <= '9')

{

temp.flag = 1;

temp.num = str[i++] - '0';

//合并多位数

while (str[i] != '\0' && ((str[i] >= '0' && str[i] <= '9') || str[i] == '.'))

{

//若是小数点

if (str[i] == '.')

{

IsPoint = 1;

t = -1;

}

//合并小数点后的数

else if (IsPoint)

{

temp.num = temp.num + (str[i] - '0') \* pow(10, t);

t--;

}

else

{

temp.num = temp.num \* 10 + (str[i] - '0');

}

i++;

}

enque(q,temp);

}

//操作符

else

{

IsPoint = t = 0;//将记录小数点的两个值初始化

temp.flag = 0;

//若操作符为左括号,压入

if (str[i] == '(')

{

temp.op = '(';

push(s, temp);

i++;

}

//若为右括号,弹栈直到把左括号弹出

else if(str[i] == ')')

{

//弹出括号内的操作符至后缀表达式队列中

while (!sempty(s) && gettop(s).op != '(')

{

enque(q, gettop(s));

pop(s);

}

//弹出左括号

if (gettop(s).op == '(')pop(s);

i++;

}

//操作符为+,-,\*,/

//栈顶操作符优先级 < 操作符,将操作符压入栈

//栈顶操作符优先级 >= 操作符,弹栈至栈顶操作符优先级 < 操作符,再将操作符压入栈

else

{

while (!sempty(s) && OP[index(gettop(s).op)] >= OP[index(str[i])])

{

enque(q, gettop(s));

pop(s);

}

temp.op = str[i];

push(s, temp);

i++;

}

}

}

while (!sempty(s))

{

enque(q, gettop(s));

pop(s);

}

}

//用后缀表达式求值

double Calculate(Queue & q)

{

Stack s;

node temp,a,b,c;

int step = 1;//栈以及队列变化步骤

InitStack(s);

//显示队列初始情况

printf("%-4d", step++);

for (int i = q.head; i < q.tail; i++)

{

if (q.data[i].flag)printf("%-8.2lf ", q.data[i].num);

else printf("%c ", q.data[i].op);

}

printf("\n");

while (!qempty(q))

{

temp = gethead(q);

deque(q);

//操作数,直接入栈

if (temp.flag)push(s, temp);

//操作符

else

{

c.flag = 1; //存储计算结果c = a op b

b = gettop(s);

pop(s);

a = gettop(s);

pop(s);

switch (temp.op)

{

case '+':c.num = a.num + b.num; break;

case '-':c.num = a.num - b.num; break;

case '\*':c.num = a.num \* b.num; break;

case '/':

if (b.num == 0)return error;

else c.num = a.num / b.num;

break;

}

push(s, c);

//显示栈以及队列变化的变化过程

printf("%-4d", step++);

for (int i = 0; i <= s.top; i++)

{

printf("%-8.2lf ", s.data[i].num);

}

for (int i = q.head; i < q.tail; i++)

{

if (q.data[i].flag)printf("%-8.2lf ", q.data[i].num);

else printf("%c ", q.data[i].op);

}

printf("\n");

}

}

return gettop(s).num;

}

//后缀表达式输出函数

void RPNprint(char str[])

{

Queue q;

Change(str, q);

while (!qempty(q))

{

node temp;

temp = gethead(q);

if (temp.flag)

{

printf("%.2lf ", temp.num);

}

else printf("%c ", temp.op);

deque(q);

}

printf("\n");

}

//在转换成后缀表达式前的检查,存在异常返回0

bool Before\_Check(char str[])

{

Stack s;

node temp;

InitStack(s);

temp.flag = 0; //0表示操作符，1表示操作数

bool flag = 0; //0表示无问题,1表示有问题

//括号匹配 及 异常符号检查

for (int i = 0; str[i] != '\0';i++)

{

temp.op = str[i];

//情况1: 左括号 直接入栈

if (str[i] == '(')push(s, temp);

//情况2: 右括号 若栈顶元素是"("左括号则无异常，否则括号不匹配

else if (str[i] == ')')

{

if(gettop(s).op == '(')pop(s);

else

{

flag = 1;

break;

}

}

//非异常符号规定为 +,-,\*,/,.,(,),数字0-9

else if (!(str[i] == '+' || str[i] == '-' || str[i] == '\*' || str[i] == '/' || str[i] == '.' || (str[i] >= '0' && str[i] <= '9')))

{

printf("表达式存在异常符号！\n");

return 0;

}

}

//最后栈不空说明括号也不匹配

if (!sempty(s))flag = 1;

if (flag == 1)

{

printf("表达式括号匹配出错！\n");

return 0;

}

//检查操作数和操作符的位置关系

for (int i = 0; str[i] != '\0'; i++)

{

if (i >= 1 && str[i + 1] != '\0')

{

if (str[i] == '+' || str[i] == '-' || str[i] == '\*' || str[i] == '/')

{

if (!(str[i - 1] >= '0' && str[i - 1] <= '9' && str[i + 1] >= '0' && str[i + 1] <= '9' || str[i - 1] == '(' || str[i - 1] == ')' || str[i + 1] == '(' || str[i + 1] == ')'))

{

printf("表达式中操作数与操作符位置关系出错！\n");

return 0;

}

}

}

}

//else printf("无异常！\n");

return 1;

}

//转换为后缀表达式后的检查，存在异常返回0

bool After\_Check(Queue q)

{

int numcnt = 0,opcnt = 0;//记录操作数和操作符的个数

int p = 0;

while (p < q.tail)

{

//操作数

if (q.data[p].flag)numcnt++;

else opcnt++;

p++;

}

if (opcnt < numcnt - 1)

{

printf("表达式中操作符过少！\n");

return 0;

}

else if (opcnt > numcnt - 1)

{

printf("表达式中存在多余操作符！\n");

return 0;

};

//else printf("无异常！\n");

return 1;

}

int main()

{

while (1)

{

system("cls");

printf

(

"算术表达式求值\n"

"规定:\n"

"1.操作数限定为正实数\n"

"2.操作符仅含+,-,\*,/,(,)\n"

"3.注意括号的匹配\n"

"4.注意除数不能为0\n"

"5.注意半角输入符号\n"

"6.注意各符号间无空格\n"

"7.输入表达式形如：6+15\*(21-8/4)\n\n"

);

Queue q;//存储后缀表达式

char str[maxn];

bool flag = 1;//1表示表达式无异常

printf("请输入正确的算术表达式：\n");

scanf("%s", str);

getchar();

//括号匹配、异常符号、操作数操作符位置关系测试

if (!Before\_Check(str))flag = 0;

//中缀表达式转换为后缀表达式

if(flag)Change(str, q);

//操作数与操作符数量测试

if (flag)

{

if (!After\_Check(q))flag = 0;

}

//flag == 1无异常才继续进行计算,若返回error说明除数为0

double res = 0;

if (flag)res = Calculate(q);

if (res == error)printf("存在除数为0！\n");

if (res != error && flag != 0)

{

printf("%s = ", str);

printf("%.2lf\n", res);

}

//询问是否继续运算

char ch;

printf("是否继续计算？（y/n）：");

scanf("%c", &ch);

if (toupper(ch) == 'Y')continue;

else break;

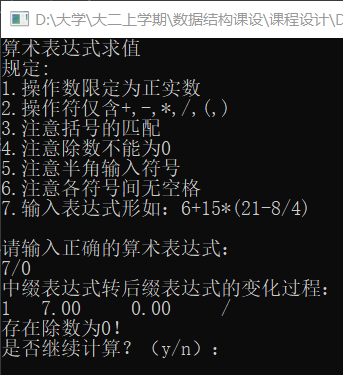
}

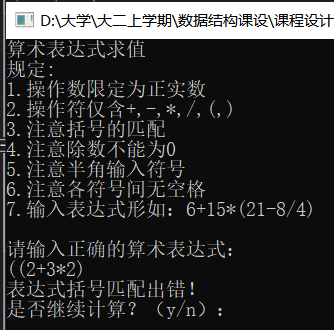
system("pause");

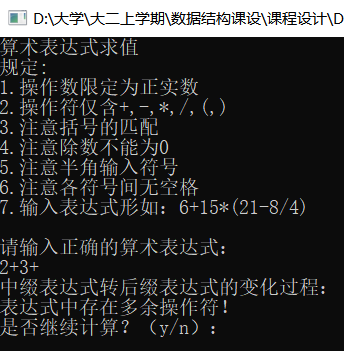
return 0;

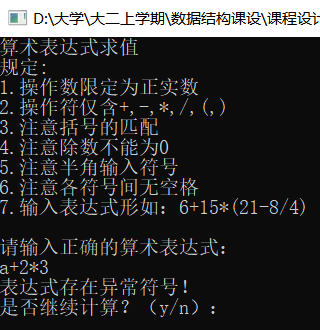
}

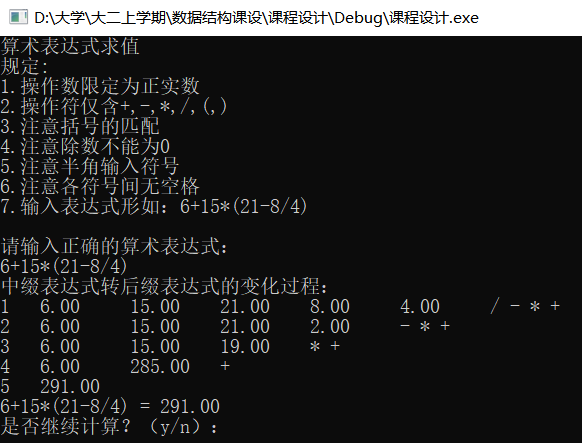
四、测试数据和结果











五、算法的时间复杂度即改进方法

算术表达式转换为逆波兰式的复杂度：O(n)

计算逆波兰式复杂度：O(n)

六、结束语

代码共395行

在该实验中，我第一次尝试着用逆波兰式计算算术表达式，而不是之前所用的中缀表达式，两者相比较，我认为逆波兰式虽然在转换时比较费劲，但是在计算时相当方便。

在书写代码之前，我在考虑如何使用两个不同的且手写的栈用于分别存放操作数和操作符，但是实际上这是不方便且会造成代码冗余的，因为两个栈的数据类型不同，需要写出两个不同的栈结构及函数，当然使用C++中的模板可以解决此问题。但在此我使用一个flag标记该元素是否为操作符便可轻而易举地解决该问题。