《数据结构与算法实验》第 6 次实验

学院： 专业： 年级：
姓名： 学号： 日期： 2022 年 4 月 10 日

第一部分 验证和设计实验

一、 实验目的

1. 验证静态链表的设计和实现。

2. 掌握栈的顺序存储结构，验证顺序栈及其操作的实现，验证栈的操作特性。

二、 实验内容

1. **静态链表的定义和测试：**建立一个静态链表。

   • 实现加入、删除、输出等操作。

   • 给定测试数据，验证抛出异常机制。

2. **顺序栈的实现（不带头节点）：** 建立一个空栈。

   • 对已建立的栈进行插入、删除、取栈顶元素等基本操作。

3. **进制转换：**设计一个算法，把十进制整数转换为二至九进制之间的任一进制输出。

   • 如课本 p77,5(3)，实验书 p52。

三、 设计与编码

1. 本实验用到的理论知识

• **进制转换：**进制转换的原理如图所示。可以看到， 98 是我们输入的 10 进制数，除以 2，余数 0 保留在栈中，得到的商 49 接着与 2 整除运算，直到 1 除以 2 等于 0，最后把栈中数据取出即可，正好用到了栈的规则，即先进后出的特性。

2. 算法设计

静态链表的设计和顺序栈的实现与之前的实验（链表、顺序表）类似，但是顺序栈只

能在栈顶进行插入和删除操作。

3. 编码

(1)静态链表的定义和测试-StaticList.h

```
#ifndef StaticList_H
#define StaticList_H
const int Maxsize=100;
template <class T>
struct Node
{
    T data;
    int link;
};
template<class T>
class StaticList
{
public:
    void InitList();  //初始化
    int Length();    //求链表长度
    int Search(T x);  //寻找元素 x 所在结点位置
    int Locate(int i); //寻找第 i 个结点的元素
    bool Append(T x);   //在表尾追加一个新结点
    bool Insert(int i,T x); //在第 i 个结点插入值为 x 的元素
    bool Remove(int i);   //删除第 i 个结点
    bool isEmpty();  //判断链表空否
    void PrintList();  //输出链表中所有元素
private:
    Node<T> elem[Maxsize];
    int avail;   //当前可支配空间首地址
};
#endif
```

(1)静态链表的定义和测试-StaticList.cpp

```
#include<iostream>
using namespace std;
#include"StaticList.h"
template <class T>
void StaticList<T>::InitList()
{
    elem[0].link=-1;
    avail=1;
```

```
    for(int i=1;i<Maxsize;i++)
    {
        elem[i].link=i+1;
        elem[Maxsize-1].link=-1;
    }
}
template <class T>
int StaticList<T>::Length()
{
    int p=elem[0].link;
    int count=0;
    while(p!=-1)
    {
       p=elem[p].link;
       count++;
    }
    return count;
}
template <class T>
int StaticList<T>::Search(T x)
{
    int p=elem[0].link;
    while(p!=-1)
    {
        if(elem[p].data==x)
        break;
        else p=elem[p].link;
    }
    return p;
}
template <class T>
int StaticList<T>::Locate(int i)
{
    if(i<0) return -1;
    if(i==0)  return 0;
    int j=1;
    int p=elem[0].link;
    while(p!=-1&&j<i)
    {
        p=elem[p].link;
        j++;
    }
    return elem[p].data;
}
```

```
template <class T>
bool StaticList<T>::Append(T x)
{
    if(avail==-1)  return false;
    int q=avail;
    avail=elem[avail].link;
    elem[q].data=x;
    elem[q].link=-1;
    int p=0;
    while(elem[p].link!=-1)
    p=elem[p].link;
    elem[p].link=q;
    return true;
}
template <class T>
bool StaticList<T>::Insert(int i,T x)
{
    if(avail==-1)  return false;
    if(i<1||i>Maxsize)  throw"位置";
    else
    {
        int j=1;
        int p=elem[0].link;
        while(p!=-1&&j<i-1)
        {
          p=elem[p].link;
          j++;
        }
        if(p==-1)  throw"位置";
        else
        {
        int s=avail;
        avail=elem[avail].link;
        elem[s].data=x;
        elem[s].link=elem[p].link;
        elem[p].link=s;
        }
    }
    return true;

}
template <class T>
bool StaticList<T>::Remove(int i)
{
```

```
    if(avail==-1)  return false;
    if(i<1||i>Maxsize)  throw"位置";
    else
    {
        int j=1;
        int p=elem[0].link;
        while(p!=-1&&j<i-1)
        {
          p=elem[p].link;
          j++;
        }
        if(p==-1||elem[p].link==-1)  throw"位置";
        else
        {
        int q=elem[p].link;
        elem[p].link=elem[q].link;
        elem[q].link=avail;
        avail=q;
        }
    }
    return true;
}
template <class T>
bool StaticList<T>::isEmpty()
{
    if(elem[0].link==-1)  return true;
    else return false;
}
template <class T>
void StaticList<T>::PrintList()
{
    int p=elem[0].link;
    while(p!=-1)
    {
        cout<<elem[p].data<<" ";
        p=elem[p].link;
    }
    cout<<endl;
}
```

(1)静态链表的定义和测试-StaticList_main.cpp

```
#include<iostream>
using namespace std;
```

```
#include "StaticList.cpp"
int main()
{
    StaticList<int>L;
    L.InitList();
    cout<<"判断链表是否为空："<<endl;
    if(L.isEmpty()==1)
    cout<<"链表为空"<<endl;
    else cout<<"链表不为空"<<endl;
    L.Append(1);
    L.Append(2);
    L.Append(3);
    L.Append(5);
    L.Append(6);
    cout<<"插入上述元素后判断链表是否为空："<<endl;
    if(L.isEmpty()==1)
    cout<<"链表为空"<<endl;
    else cout<<"链表不为空"<<endl;
    cout<<"寻找元素 3 所在结点位置:"<<endl;
    cout<<L.Search(3)<<endl;
    cout<<"第 2 个结点的元素的值为："<<endl;
    cout<<L.Locate(2)<<endl;
    cout<<"执行操作后链表输出为："<<endl;
    L.PrintList();
    L.Insert(4,4);
    cout<<"执行操作后链表输出为："<<endl;
    L.PrintList();
    cout<<"链表的长度为："<<" "<<L.Length()<<endl;
    L.Remove(6);
    cout<<"执行操作后链表输出为："<<endl;
    L.PrintList();
    return 0;
}
```

(2)顺序栈的实现-SeqStack.h

```
#ifndef SeqStack_H
#define SeqStack_H
const int StackSize = 10;
template<class DataType>
class SeqStack
{
    private:
        DataType data[StackSize];
```

```
        int top;
    public:
        SeqStack();
        ~SeqStack(){}
        void Push(DataType x);
        DataType Pop();
        DataType GetTop();
        int Empty();
};
#endif
```

(2) 顺序栈的实现-SeqStack.cpp

```
#include "SeqStack.h"

template<class DataType>
SeqStack<DataType>::SeqStack()
{
    top = -1;
}

template<class DataType>
void SeqStack<DataType>::Push(DataType x)
{
    if (top == StackSize-1) throw "ÉÏÒç";
    top++;
    data[top] = x;
}

template<class DataType>
DataType SeqStack<DataType>::Pop()
{
    DataType x;
    if (top == -1) throw "ÏÂÒç";
    x == data[top--];
    return x;
}

template<class DataType>
DataType SeqStack<DataType>::GetTop()
{
    if (top != -1)
    return data[top];
}
```

```
template<class DataType>
int SeqStack<DataType>::Empty()
{
    if (top == -1) return 1;
    else return 0;
}
```

(2) 顺序栈的实现-SeqStack_main.cpp

```
#include <iostream>
using namespace std;
#include "SeqStack.cpp"

int main()
{
    SeqStack<int> S;
    if (S.Empty())
    {
        cout<<"栈为空"<<endl;
    }
    else cout<<"栈非空"<<endl;
    cout<<"对 15 和 10 执行入栈操作"<<endl;
    S.Push(15);
    S.Push(10);
    cout<<"栈顶元素为: "<<S.GetTop()<<endl;
    cout<<"执行一次出栈操作"<<endl;
    S.Pop();
    cout<<"栈顶元素为: "<<S.GetTop()<<endl;
    return 0;
}
```

(3)链栈的实现-LinkStack.h

```
#ifndef SeqStack_H
#define SeqStack_H
//const int StackSize = 10;
template<class DataType>
struct Node
{
    DataType data;
    Node<DataType> *next;
};
```

```
template<class DataType>
class LinkStack
{
    public:
        LinkStack();
        ~LinkStack();
        void Push(DataType x);
        DataType Pop();
        DataType GetTop();
        int Empty();
    private:
        Node<DataType> *top;
};
#endif
```

(3)链栈的实现-LinkStack.cpp

```
#include <iostream>
using namespace std;
#include "LinkStack.h"

template <class DataType>
LinkStack<DataType>::LinkStack( )
{
    top=NULL;//无头结点
}

template <class DataType>
LinkStack<DataType> :: ~LinkStack()
{
    Node<DataType> *q;
    while (top != NULL)          //释放单链表的每一个结点的存储空间
    {
        q = top;                 //暂存被释放结点
        top = top -> next;       //first 指向被释放结点的下一个结点
        delete q;
    }
}

template <class DataType>
void LinkStack<DataType>::Push(DataType x)
{
    Node<DataType> *s;
    s = new Node<DataType>;     //直接自己抛出异常
```

```
    if(s == NULL) throw "上溢";
    s -> data = x;              //申请一个数据域为 x 的结点
    s -> next = top;
    top = s;                    //将结点放到栈顶
}

template <class DataType>
DataType LinkStack<DataType>::Pop( )
{
    DataType x;
    Node<DataType> *p;
    if (top == NULL) throw "下溢";
    x = top -> data;
    p = top;
    top = top -> next;          //top 指针移动
    delete p;
    return x;
}

template <class DataType>
DataType LinkStack<DataType>::GetTop( )
{
    if (top != NULL)
        return top -> data;
    else
        throw "空";
}

template <class DataType>
int LinkStack<DataType>::Empty( )
{
    if(top == NULL) return 1;
    else return 0;
}
```

(3)链栈的实现-LinkStack_main.cpp

```
#include <iostream>         //引用输入输出流
using namespace std;
#include "LinkStack.cpp"       //引入成员函数文件

int main( )
{
    LinkStack<int> S;      //创建模板类的实例
```

```
    try
    {
        if (S.Empty()==1)
            cout<<"栈为空"<<endl;
        else
            cout<<"栈非空"<<endl;
        cout<<"对 15、10、5、1 执行入栈操作"<<endl;
        S.Push(15);
        S.Push(10);
        S.Push(5);
        S.Push(1);
        cout<<"栈顶元素为:"<<S.GetTop( )<<endl;
        cout<<"执行一次出栈操作"<<endl;
        S.Pop( );           //执行出栈操作
        cout<<"栈顶元素为:"<<S.GetTop( )<<endl;
        cout<<"执行一次出栈操作"<<endl;
        S.Pop( );           //执行出栈操作
        cout<<"栈顶元素为:"<<S.GetTop( )<<endl;
        cout<<"执行一次出栈操作"<<endl;
        S.Pop( );           //执行出栈操作
        cout<<"栈顶元素为:"<<S.GetTop( )<<endl;
        cout<<"执行一次出栈操作"<<endl;
        S.Pop( );           //执行出栈操作
        cout<<"执行一次出栈操作"<<endl;
        S.Pop( );
    }
    catch (const char *s)
    {
        cout<<s<<"\n";
    }
    return 0;
}
```

(4)进制转换-LinkStack.h

```
#ifndef SeqStack_H
#define SeqStack_H
//const int StackSize = 10;
template<class DataType>
struct Node
{
    DataType data;
    Node<DataType> *next;
};
```

```
template<class DataType>
class LinkStack
{
    public:
        LinkStack();
        LinkStack(int a,int k);
        ~LinkStack();
        void Push(DataType x);
        DataType Pop();
        DataType GetTop();
        int Empty();
    private:
        Node<DataType> *top;
};
#endif
```

(4)进制转换-LinkStack.cpp

```
#include <iostream>
using namespace std;
#include "LinkStack.h"

template <class DataType>
LinkStack<DataType>::LinkStack( )
{
    top=NULL;//无头结点
}

template <class DataType>
LinkStack<DataType> :: ~LinkStack()
{
    Node<DataType> *q;
    while (top != NULL)          //释放单链表的每一个结点的存储空间
    {
        q = top;                 //暂存被释放结点
        top = top -> next;       //first 指向被释放结点的下一个结点
        delete q;
    }
}

template <class DataType>
void LinkStack<DataType>::Push(DataType x)
{
```

```
    Node<DataType> *s;
    s = new Node<DataType>;     //直接自己抛出异常
    if(s == NULL) throw "上溢";
    s -> data = x;              //申请一个数据域为 x 的结点
    s -> next = top;
    top = s;                    //将结点放到栈顶
}

template <class DataType>
LinkStack<DataType>::LinkStack(int a,int k)
{
    top = NULL;
    do
    {
        Node<DataType> *s;
        s = new Node<DataType>;
        s->data = a%k;        //申请一个数据域为 x 的结点
        s->next=top;
        top=s;
        a=a/k;
    }while(a!=0);
}

template <class DataType>
DataType LinkStack<DataType>::Pop( )
{
    DataType x;
    Node<DataType> *p;
    if (top == NULL) throw "下溢";
    x = top -> data;
    p = top;
    top = top -> next;          //top 指针移动
    delete p;
    return x;
}

template <class DataType>
DataType LinkStack<DataType>::GetTop( )
{
    if (top != NULL)
        return top -> data;
    else
        throw "空";
}
```

```
template <class DataType>
int LinkStack<DataType>::Empty( )
{
    if(top == NULL) return 1;
    else return 0;
}
```

(4)进制转换-LinkStack_main.cpp

```
#include <iostream>
#include <cstdio>
using namespace std;
#include "LinkStack.cpp"

int main()
{
    int a,k;
    cout<<"输入十进制数：  "<<endl;
    cin>>a;
    cout<<"输入待转换的进制（数字）：  "<<endl;
    cin>>k;
    LinkStack<int> S(a,k);
    cout<<"转换结果为：  "<<endl;
    while(S.Empty() != 1)
    {
        cout<<S.Pop();
    }
    return 0;
}
```

四、 运行与测试

1. 静态链表的定义和测试

```
判断链表是否为空：
链表为空
插入上述元素后判断链表是否为空：
链表不为空
寻找元素3所在结点位置:
3
第2个结点的元素的值为：
2
执行操作后链表输出为：
1 2 3 5 6
执行操作后链表输出为：
1 2 3 4 5 6
链表的长度为： 6
执行操作后链表输出为：
1 2 3 4 5
请按任意键继续. . .
```

2. 顺序栈的实现

```
栈为空
对15和10执行入栈操作
栈顶元素为：10
执行一次出栈操作
栈顶元素为：15
请按任意键继续. . .
```

3. 链栈的实现

```
栈为空
对15、10、5、1执行入栈操作
栈顶元素为:1
执行一次出栈操作
栈顶元素为:5
执行一次出栈操作
栈顶元素为:10
执行一次出栈操作
栈顶元素为:15
执行一次出栈操作
执行一次出栈操作
下溢
请按任意键继续. . .
```

4. 进制转换（以二进制和八进制为例）

- 二进制

```
输入十进制数：
92
输入待转换的进制（数字）：
2
转换结果为：
1011100请按任意键继续. . .
```

- 八进制

```
输入十进制数：
2346
输入待转换的进制（数字）：
8
转换结果为：
4452请按任意键继续. . .
```

五、 总结与心得

通过这次实验，我学习到了静态链表、顺序栈和链栈的建立和应用，顺序栈和链栈的实现其实是基于线性表实现的，只是限制了线性表的一些操作。因此只需要基于线性表的基础在其程序上进行一些调整即可。

在十进制转换的短除法中，因为最终结果是余数的逆序，故适合用有着先入后出的特点的栈来实现，可以说进制的转换是栈的具体应用的一个最佳契合的例子。

第二部分 设计实验：表达式求值

一、问题描述

对一个合法的中辍表达式求值。简单起见，假设表达式只包含+, -, ×, ÷等 4 个双目运算符，且运算符本身不具有二义性，故限制操作数均为一位整数。可以先使用中缀表达式直接求值，同时将中缀表达式转为后缀表达式输出，再利用后缀表达式求值，最后将二者进行比较。

二、基本要求

- 由用户从键盘输入中缀表达式或后缀表达式；

- 可实现一位数加减乘除取余的运算，符合四则运算规律；
- 输出最后的计算结果。

三、算法设计

1. 数据结构的设计

引入两个栈，一个符号栈(OPTR)用于储存运算符，一个数据栈(OPND)用于储存数字。

2. 算法设计

• 中缀表达式求值，需要先考虑各操作符的优先级，因此建立 isp（站内优先数）和 icp（栈外优先数）两个函数用于读取运算符并判断优先级，操作符优先数相等的情况只出现在括号配对或栈底#号与输入流中的#号配对时，代码如下。

```
int icp(const char a)//栈外优先级
{
    switch(a)
    {
        case '#':return 0;
        case '(':return 6;
        case '+':return 2;
        case '-':return 2;
        case '*':return 4;
        case '/':return 4;
        case '%':return 4;
        case ')':return 1;
        default:throw "ERROR!";
    }
}
```

```
int isp(const char a)//栈内优先级
{
    switch(a)
    {
        case '#':return 0;
        case '(':return 1;
        case '+':return 3;
        case '-':return 3;
        case '*':return 5;
        case '/':return 5;
        case '%':return 5;
        case ')':return 6;
        default:throw "ERROR!";
    }
}
```

(1) 中缀表达式求值算法步骤

逐字扫描输入字符串，分为以下几种情况：

① 扫描到数字：将扫描到的数压入操作数栈，扫描输入表达式的下一字符

② 扫描到操作符，且优先级大于操作符栈顶优先级：将扫描到的操作符压入操作符栈顶，扫描输入表达式的下一字符

③ 扫描到操作符，且优先级小于操作符栈顶优先级：弹出操作符栈顶字符，弹出操作数栈前两个元素，做运算后将运算结果压入操作数栈

④ 扫描到操作符，且优先级等于操作符栈顶优先级(只有可能是扫描到“)”，且

栈顶为“(”：弹出操作符栈栈顶的“(”，扫描输入表达式的下一字符。

(2) 中缀表达式转为后缀表达式算法步骤

先将操作符栈初始化，将结束符‘ ;’进栈。然后读入中缀表达式字符流的首字符 ch。重复执行以下步骤，直到 ch=‘ ;’，同时栈顶的操作符也是‘ ;’，停止循环：若 ch 是操作数直接输出，读入下一个字符 ch；若 ch 是操作符，判断 ch 的优先级 icp 和位于栈顶的操作符 op 的优先级 isp：

① 若 icp(ch)＞isp(op)，令 ch 进栈，读入下一个字符 ch。

② 若 icp(ch)＜isp(op)，退栈并输出。

③ 若 icp(ch)==isp(op)，退栈但不输出，若退出的是“(”号读入下一个字符 ch。

(3) 后缀表达式求值算法步骤

从左到右依次扫描表达式的每一个字符，执行下述操作：若当前字符是运算对象，则入栈 S，处理下一个字符；若当前字符是运算符，则从栈 S 出栈两个运算对象，执行运算并将结果入栈 S，处理下一个字符。输出栈 S 的栈顶元素，即表达式的运算结果。

四、代码实现

1. 中缀求值-LinkStack.h

```
#ifndef LinkStack_H
#define LinkStack_H
//const int StackSize = 10;
struct Node
{
    char data;
    Node *next;
};
class LinkStack
{
    public:
        LinkStack( );
        ~LinkStack( );
        void Push(char x);
```

```
        char Pop( );
        char GetTop( );
        int Empty( );
    private:
        Node *top;
};
bool InOP(char c);
int getpriority(char a);
char Preceded(char a,char b);
int Operate(char a,char thera,char b);
char com();
#endif
```

1. 中缀求值-LinkStack.cpp

```
#include<iostream>
using namespace std;
#include "LinkStack.h"

LinkStack::LinkStack()
{
    top = NULL;
}

LinkStack::~LinkStack()
{
    Node *q;
    while (top != NULL)          //释放单链表的每一个结点的存储空间
    {
        q = top;                 //暂存被释放结点
        top = top -> next;       //first 指向被释放结点的下一个结点
        delete q;
    }
}

void LinkStack::Push(char x)
{
    Node *s;
    s = new Node;     //直接自己抛出异常
    if(s == NULL) throw "上溢";
    s -> data = x;               //申请一个数据域为 x 的结点
    s -> next = top;
    top = s;                     //将结点放到栈顶
}
```

```
char LinkStack::Pop()
{
    Node *p;
    if (top == NULL) throw "下溢";
    char x = top -> data;
    p = top;
    top = top -> next;          //top 指针移动
    delete p;
    return x;
}

char LinkStack::GetTop()
{
    if (top != NULL)
        return top -> data;
    else
        throw "空";
}

int LinkStack::Empty()
{
    if (top == NULL) return 1;
    else return 0;
}

bool InOP(char c)
{
    bool x=0;
    if(c=='+'||c=='-'||c=='*'||c=='/'||c=='('||c==')') x=1;
    return x;
}

int getpriority(char a)
{
    int p=0;
    switch(a){
        case '+':p=1;break;
        case '-':p=1;break;
        case '*':p=3;break;
        case '/':p=3;break;
    }
    return p;
}
```

```
char Preceded(char a,char b)
{
    if(b=='(') return '<';
    if(b==')')
    {
        if(a=='(') return '=';
        else return '>';
    }
    int m,n;
    m=getpriority(a);
    n=getpriority(b);
    if(m>n) return '>';
    else if(m<n) return '<';
    else return '=';
}

int Operate(char a,char thera,char b)
{
    int c;
    int m,n;
    m=a-48;
    n=b-48;
    switch(thera)
    {
        case '+':c=m+n;break;
        case '-':c=m-n;break;
        case '*':c=(a-48)*(b-48);break;
        case '/':c=(a-48)/(b-48);break;
    }
    return c;
}

char com()//算术表达式求值的算符优先算法。
{//设 OPTR 和 OPND 分别为运算符栈和运算数栈。OP 为运算符集合。
    LinkStack(OPTR);
    OPTR.Push('#');
    LinkStack(OPND);
    char c=getchar();
    while (c != '#')
    {
        if('1'<=c && c<='9' ) //不是运算符则进栈
        {
            OPND.Push(c);
```

```
            c=getchar();
        }
        else if(c=='(')
        {
            OPTR.Push(c);
            c=getchar();
        }
        else if(c==')')
        {
            while( OPTR.GetTop() !='(')
            {
                char thera=OPTR.Pop();
                char b=OPND.Pop();
                char a=OPND.Pop();
                char v=('0'+ Operate(a,thera,b));
                OPND.Push(v);
            }
            OPTR.Pop();
            c=getchar();
        }
        else
        {
            switch (Preceded(OPTR.GetTop(),c))
            {
                case '<'://栈顶元素优先权低,运算符入栈
                    OPTR.Push(c);
                    c=getchar();
                    break;
                case '=':
                    //cout<<222<<endl;//脱括号并接收下一字符
                    OPTR.Push(c);
                    c=getchar();
                    break;
                case '>':  //退栈并将运算结果入栈
                    char thera=OPTR.Pop();
                    char b=OPND.Pop();
                    char a=OPND.Pop();
                    char v=Operate(a,thera,b);
                    OPND.Push(v);
                    break;
            } //switch
        }
    }//while
    while(OPTR.GetTop()!='#')
```

```
    {
        char thera=OPTR.Pop();
        char b=OPND.Pop();
        char a=OPND.Pop();
        char v=('0'+ Operate(a,thera,b));
        OPND.Push(v);
    }
    return OPND.GetTop();
}
```

1. 中缀求值-LinkStack_main.cpp

```
#include<iostream>
using namespace std;
#include "LinkStack.cpp"

int main()
{
    cout<<"输入中缀表达式为: "<<endl;
    char s;
    s = com();
    cout<<"求得值为: "<<endl;
    cout<<(s)<<endl;
}
```

2. 中缀转后缀-LinkList.h

```
#ifndef LinkStack2_H
#define LinkStack2_H
//const int StackSize = 10;
template <typename T>
struct Node
{
    T data;
    Node<T> *next;
};

template <typename T>
class LinkStack
{
    public:
        LinkStack( );
        ~LinkStack( );
```

```
        void Push(T x);
        T Pop( );
        T GetTop( );
        int Empty( );
        void makeEmpty();
    private:
        Node<T> *top;
};
int icp(const char a);
int isp(const char a);
void Run();
#endif
```

2. 中缀转后缀-LinkList.cpp

```
#include <iostream>
using namespace std;
#include "LinkStack2.h"
const int MaxSize = 100;

template <typename T>
LinkStack<T>::LinkStack()
{
    top = NULL;
}

template <typename T>
LinkStack<T>::~LinkStack()
{
    Node<T> *p;
    while (top != NULL)
    {
        p = top; //暂存栈顶地址
        top = top->next;
        delete p;
    }
}

template <typename T>
void LinkStack<T>::Push(T x)
{
    Node<T> *s = new Node<T>;
    s->data = x;
    s->next = top;
```

```
    top = s;
}

template <typename T>
T LinkStack<T>::Pop()
{
    if (top == NULL) throw"栈空";
    T temp = top->data;
    Node<T> *p = top; //暂存栈顶
    top = top->next;
    delete p;
    return temp;
}

template <typename T>
T LinkStack<T>::GetTop()
{
    if (top == NULL) throw"栈空";
    return top->data;
}

template <typename T>
int LinkStack<T>::Empty()
{
    if (top == NULL)return 1;
    else return 0;
}

template <typename T>
void LinkStack<T>::makeEmpty()
{
    top = -1;
}

int icp(const char a)//栈外优先级
{
    switch(a)
    {
        case '#':return 0;
        case '(':return 6;
        case '+':return 2;
        case '-':return 2;
        case '*':return 4;
        case '/':return 4;
```

```
        case '%':return 4;
        case ')':return 1;
        default:throw "ERROR!";
    }
}

int isp(const char a)//栈内优先级
{
    switch(a)
    {
        case '#':return 0;
        case '(':return 1;
        case '+':return 3;
        case '-':return 3;
        case '*':return 5;
        case '/':return 5;
        case '%':return 5;
        case ')':return 6;
        default:throw "ERROR!";
    }
}

void Run()
{
    char Inf[MaxSize],ch;
    LinkStack<char> OPTR;
    OPTR.Push('#');
    cout<<"请输入需要转换的中缀表达式："<<endl;
    cin>>Inf;
    int count = 0;
    ch = Inf[0];
    try
    {
        while(ch != '#' || OPTR.GetTop() != '#')
        {
            if(ch >= '0' && ch <= '9')
            {
                cout<<ch;
                ch = Inf[++count];
                if(ch=='\0') ch='#';
            }
            else if(icp(ch)>isp(OPTR.GetTop()))
            {
                OPTR.Push(ch);
```

```
                ch=Inf[++count];
                if(ch=='\0') ch='#';
            }
            else if (icp(ch) < isp(OPTR.GetTop()))
            {
                cout << OPTR.Pop();
            }
            else if (ch == ')')
            {
                OPTR.Pop();
                ch=Inf[++count];
                if(ch=='\0') ch='#';
            }
        }
        cout<<endl;
    }
    catch (const char *s)
    {
        cout<<s<<endl;
    }
}
```

2. 中缀转后缀-LinkList_main.cpp

```
#include<iostream>
using namespace std;
#include "LinkStack2.cpp"

int main()
{
    try
    {
        while (1) Run();
    }
    catch (const char *s)
    {
        cout << s << endl;
    }
    return 0;
}
```

3. 后缀求值-LinkStack.h

```
#ifndef LinkStack_H
#define LinkStack_H

struct Node
{
        char data;
        Node *next;
};

class LinkStack
{
    public:
        LinkStack();
        ~LinkStack();
        void Push(char x);
        char Pop();
        char GetTop();
    private:
        Node *top;
};
int Operate(char a,char thera,char b);
char Sufix();
#endif
```

3. 后缀求值-LinkStack.cpp

```
#include <iostream>
using namespace std;
#include "LinkStack3.h"

LinkStack::LinkStack()  {top=NULL;}

LinkStack::~LinkStack()
{
    Node *q;
    while (top != NULL)
    {
        q = top;
        top = top->next;
        delete q;
    }
}

void LinkStack::Push(char x)
```

```
{
    Node *s=new Node;s->data=x;
    s->next=top;top=s;
}

char LinkStack::Pop()
{
    char x;
    Node *p;
    if(top==NULL) cout<<"下溢"<<endl;
    x=top->data ;p=top;
    top=top->next ;
    delete p;
    return x;
}

char LinkStack::GetTop()
{
    if(top!=NULL) return top->data ;
}

int Operate(char a,char thera,char b)
{
    int c;
    int m,n;
    m=a-48;n=b-48;
    switch(thera){
    case '+':c=m+n;break;
    case '-':c=m-n;break;
    case '*':c=m*n;break;
    case '/':c=m/n;break;
    }
    return (c+48);
}

char Sufix()//算术表达式求值的算符优先算法。
{//设 OPTR 和 OPND 分别为运算符栈和运算数栈。OP 为运算符集合。
   LinkStack S;
   S.Push('#');
   char c=getchar();
   while (c !='#')
   {
        if('1'<=c && c<='9' ) //运算数
        {
```

```
            S.Push(c);
            c=getchar();
        }
        else
        {
            char b=S.Pop();
            char a=S.Pop();
            char v=Operate(a,c,b);
            S.Push(v);
            c=getchar();
        }
    }//while
    return S.GetTop();
} //Sufix
```

3. 后缀求值-LinkStack_main.cpp

```
#include <iostream>
using namespace std;
#include "LinkStack3.h"

int main()
{
    cout<<"请输入要转换的表达式（以#结束）："<<endl;
    char value;
    value=Sufix();
    cout<<"后缀表达式的值："<<(value-48)<<endl;
    return 0;
}
```

五、运行测试

测试结果如图。

1. 中缀求值

```
输入中缀表达式为:
2+3*2#
求得值为:
8

--------------------------------
Process exited after 12.77 seconds with return value 0
请按任意键继续. . .
```

2. 中缀转后缀

```
请输入需要转换的中缀表达式：
3*(4+2)/2-5#
342+*2/5-
请输入需要转换的中缀表达式：
q
ERROR!
请输入需要转换的中缀表达式：
```

3. 后缀求值

```
请输入要转换的表达式：
342+*2/5-#
后缀表达式的值：4

--------------------------------
Process exited after 11.82 seconds with return value 0
请按任意键继续. . .
```

六、总结与心得

在学会了中缀、前缀、后缀的表达方式和它们之间相互转化的原理的基础上，根据原理，先在纸上实现了中缀求值、中缀转后传、后缀求值等不同的功能，然后根据算法进行编程。对于我们而言中缀表达式更符合我们计算的处理逻辑，但是对于计算机而言则是后缀表达式更加容易处理，只需从左到右读取运算即可。这一算法非常地切合栈这种数据类型，有利于提高我们对栈这种数据类型的认识。

程序可提升的地方在于，由于人们并不经常使用后缀表达式，如果能够在键入的时候都使用中缀，在程序内部自动转换成后缀表达再进行后缀求值，程序将会更好些。但是在实践的过程中遭遇到了困难，最终还是将后两个实验分开来。