实 验 报 告

课程名称：数据结构 (C)

实验项目：栈或队列的基本操作与应用

实验仪器：计算机

项目	报告格式	写作质量	逻辑、注释质量、思想描述	复杂度分析	合计
百分比 (%)	15	25	40	20	100
得分	**9**	**25**	**25**	**10**	**100**

系　　别：计算机系

专　　业：数据科学与大数据技术

班级姓名：大数据 1701　张丹颖

日　　期：2018-10-26

成　　绩：

同组成员：无

指导教师：丁濛

一、实验目的

1. 熟练掌握栈和队列的基本操作原则；
2. 掌握栈和队列的顺序存储和链式存储实现方式；
3. 实现一个循环队列；(验证)
4. 会用栈解决实际问题。（设计、综合）

二、实验内容

2.1 项目一

实现一个链式存储的栈结构并分析每个功能的复杂度，要求具有栈的基本功能。

1. 构造，析构；
2. Pop，Top，Push，size

2.2 项目二

实现顺序存储的循环队列，并分析每个功能的复杂度，要求具有队列的基本功能。

1. 构造，析构；
2. EnQueue，DeQueue，Size

2.3 项目三

利用栈，完成一个后缀四则表达式的计算，并返回结果。

2.4 项目四

完成上机平台上实验二的题目。

三、实验过程

3.1 项目 1

大致思想：面向对象，stack 类继承 LinkList 类的部分方法，比如 stack 类中的 push 从父类 append 继承,pop 继承父类 deleteItem(size-1)。

(1) 实验步骤

1. CreateList(int num) 函数，直接在尾指针后 new 一个新结点接上，复杂度 O(1);
2. 构造函数 Stack() 继承构造函数 LinkList()，为头指针开辟空间，复杂度 O(1);
3. 析构 Stack(), 遍历整个链表 delete n 个结点, 复杂度 O(n)；

4. pop 函数，即所写 deleteItem(size-1) 函数，由于这里将 ptail 处理为存有意义数值结点的下一个结点，所以还得遍历到 ptail 前面一个结点，复杂度 O(n)，为结点个数。另一种做法：赋 ptail 有意义数值，直接 pop ptail，再遍历到新的最后一个结点设置为 ptail，复杂度还是 O(n);（若用双向链表，复杂度 O(1)）

5. push 即代码所写 append 函数, 直接赋值 ptail,new 出新 ptail, 复杂度 O(1);

6. top 函数，本次的操作为 ptail 处理为存有意义数值的后一个结点，所以 ptial 不代表最后一个有意义数值结点，遍历复杂度 O(n)（n 为结点个数）；另一种做法，赋 ptail 有意义数值, 复杂度 O（1）;

7. size 函数: 为了体现链表的线性结构, 和结点联系紧密，采取遍历到一个结点计数 size 就 +1 的操作，复杂度 O(n)（n 为结点个数）, 另一种做法：若直接采用 len 长度做 size 函数的返回值，复杂度 O(1), 虽然复杂度低，但和链表的线性结构没什么关系。

(2)　必要代码

```
/*
  stack 类继承LinkList类的部分方法，其中Append 为push,DeleteIltem(a.size()-1)为pop
  只有getTop（）和clear_stack 是在stack 类新加的，其他依靠继承
*/
#include<iostream>
using namespace std;
struct Node{
    int data;
    Node * next = NULL;
  };

class LinkList{
protected:
  int len = 0;
  Node *phead = NULL;
  Node *ptail = NULL;
public:
  LinkList (){
    phead = new Node;
    ptail = new Node;
    ptail = phead;
  }

  void
      Destrory(){//相当于显式调用LinkList的析构函数:~LinkList()，程序结束后delete每个new出来的结点
     Node *pdelete,*ptemp;
     pdelete = phead->next;
     while(pdelete!=NULL){
       ptemp = phead->next->next;//总是让头指针指向要删结点的下一个结点
       phead->next = ptemp;//ptemp存储下一个要删的结点
```

```
      delete pdelete;//删除本次要删的结点
      pdelete = ptemp;
    }

    //tip: 用 DeleteIltem and isEmpty()
        两个函数结合来destroy更简洁，就像下述stack类中~stack（）析构函数

    delete phead;//头结点也要释放
      ptail = phead = NULL;

    this->len = 0;

  }

//销毁栈之后，头结点也被销毁，需要重新构造
Node *buildPhead(){
  phead = new Node;
  return phead;
}

//创建链表的新结点：直接在ptail后new一个新结点接上
void Create_List(int num){
  ptail->data = num;
  ptail->next = new Node;
  ptail = ptail->next;
  ptail->next = NULL;
  len++;
}

//找
int GetItem(int idx){
  Node* p = phead;
  if(idx>=len){//输入下标不合法
    return -1;
  }

  for(int i=0;i<idx;i++){//遍历到目标位置
    p = p->next;
  }
  return p->data;
}

//增（push）
 void Append(int num){
  //栈被销毁就不能push：需要重新构造栈再push
  if(phead==NULL){
      cout<<"stack is non_exist,now build it!"<<endl;
      buildPhead();
      Create_List(num);
      cout<<"push successfully!"<<endl;
```

```
      return;
    }
    ptail->data = num;//直接在尾结点ptail后面追加
    ptail->next = new Node;
    ptail = ptail->next;
    len++;
    cout<<"push successfully!"<<endl;
}

//定位
int LocateItem(int num){
  Node* p = phead;
  int i=0;
  while(p->next!=NULL){
    if(p->data == num){//遍历，找到目标值为止，返回位置
      return i;
    }
    p = p->next;
    i++;
  }

  return -1;//遍历完都没找到，返回负数

}

void InsertBefore(int idx,int num){
  Node* p = phead;

  if(idx<0||idx>len){//输入下标不合法的情况
    cout<<"insert illegal!"<<endl;
    return;
  }
  else if(idx==0) {//插入头结点
    Node* newphead =new Node;
    newphead->next = phead;
    newphead->data = num;
    phead = newphead;//更换头结点
    len++;
  }
  else if(idx==len){//插入尾结点
    ptail->data = num;
    ptail->next = new Node;
    ptail = ptail->next;
    ptail->next = NULL;
    len++;
  }
  else //插入中间结点需要遍历
  {
    for(int i=0;i<idx-1;++i){
      p = p->next;
```

```
    }
    //遍历到idx-1,找到要插入位置的前一个元素，插入
    Node *newnode = new Node;
    newnode->next =NULL;
    newnode->data = num;

    newnode->next = p->next;
    p->next = newnode;
    len++;

  }
  cout<<"insert successfully!"<<endl;

}
  int DeleteIltem(int k){
   int datatemp;
   Node* p= phead;

   if(k<0||k>=len){//删除不合格
     return -1;
   }
   else if(k==0){//删除头结点
     Node* ptemp = phead;
     int datatemp = phead->data;
     phead = phead->next;
     delete ptemp;
     ptemp = NULL;
     len--;
     return datatemp;
   }
   else if(k==len-1){//删除尾结点（即pop）
     Node *p = phead;
     for(int i =0;i<k-1;++i){
         p = p->next;
     }

     Node * ptemp = p->next;
     datatemp = ptemp->data;
     p->next = ptail; //越过最后一个元素，指向尾指针（这里的ptail并没有存值）
     delete ptemp;
     ptemp = NULL;
     len--;
     return datatemp;
   }
   else{              //删除一般结点，需要遍历
      for(int i =0;i<k-1;++i){
        p = p->next;
      }
     Node* ptemp;
     ptemp = p->next;
```

```
    datatemp = ptemp->data;
    p->next = ptemp->next;
    delete ptemp;
    ptemp=NULL;
    len--;
    return datatemp;
  }



}

Node* getPtail(){//得到真正存有效数字的尾部，遍历
  Node*p = phead;
  while(p->next->next!=NULL){
    p = p->next;
  }
  return p;
}


bool isEmpty(){
    if(phead==NULL||phead->next==NULL){
      return true;
    }
    return false;
}

int size(){
   int n = 0;
   Node *p = phead;
   while(p->next!=NULL){ //依赖指针的空指针域判断size，不单单是len=0
    n++;
    p = p->next;
   }
   return n;
}

void printAll() {
    Node *p= phead;
    while(p->next!= NULL){
    cout<<p->data<<" ";
    p = p->next;
  }
  cout<<endl;
}
};
```

```
class Stack :public LinkList{
  public:
    Stack(){//构造函数
    }
    ~Stack(){//析构
      int length = size();
      while(!isEmpty()){
      DeleteIltem(length-1);
        length--;
      }
      delete phead;
        phead = NULL;
      len = 0;
    }

    int getTop(){
      if(isEmpty()){
        return -1;
      }
      return getPtail()->data;//直接调用父类获得有意义数值“尾”指针
    }

    void ClearStack(){//与destroy 区别：clear 不回收头指针
      while(phead->next!=NULL){//始终删除头结点后面的结点
        Node *pdelete = phead->next;
        phead->next = pdelete->next;
        delete pdelete;
      }
      ptail = phead;
      this->len = 0;
      cout<<"clear successfully!"<<endl;
    }



};


int main(){
  char ch;
  int geshu,in;
  Stack a;
  cout<<"input the stack height:";
  cin>>geshu;
  cout<<"please input numbers"<<endl;
  for(int i=0;i<geshu;i++){
    cin>>in;
    a.Create_List(in);
  }
  bool flag = 1;
```

```
while(flag){
  cout<<"a(push),b(pop),c(top),d(size),e(clear),f(print),g(quit)"<<endl;
  cin>>ch;
    switch(ch){
    case 'a':{
      int ina;
      cout<<"input the num you wanna push:";
      cin>>ina;
      a.Append(ina);
      break;
    }

    case 'b':{
      int inb = a.DeleteIltem(a.size()-1);
      if(inb==-1){
        cout<<"delete illegal!"<<endl;
        break;
      }
      else{
        cout<<"pop " << inb <<" successfully!"<<endl;
      }
      break;
    }

    case 'c':{
      int ans = a.getTop();
      if(ans==-1){
        cout<<"stack is empty,can't get top!"<<endl;
        break;
      }
      cout<<"the top element is:"<<ans<<endl;
      break;
    }

    case 'd':{
      cout<<"the height of stack is:"<<a.size()<<endl;
      break;
    }
    case 'e':{
      a.ClearStack();
      break;
    }
    case 'f':{
      a.printAll();
      break;
    }
    case 'g':{
      flag = 0;
      break;
    }
```

```
    default:{
      cout<<"input illegal.please input again!"<<endl;
      break;
    }

  }
 }


 return 0;
}
```

(3) 实验结果

见图 1

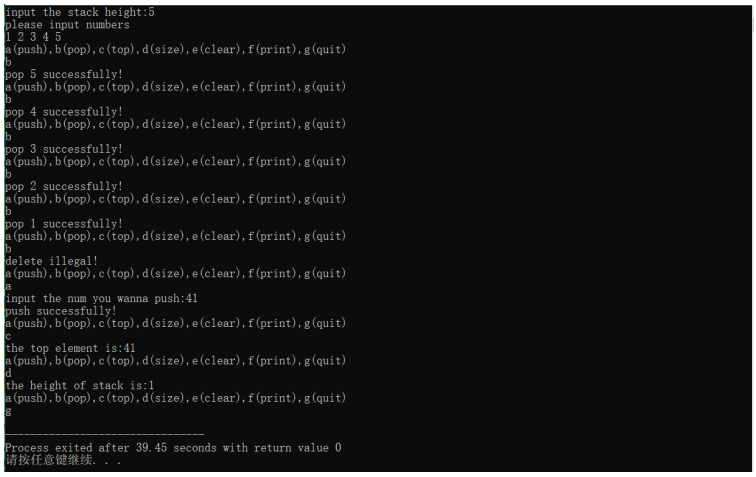

图 1: 栈的构造, 析构,pop,push,top 功能实现

3.2 项目 2

大致思想：顺序存储，利用一个数组存储队列中的数，标记队首 (front) 和队尾 (rear) 的位置。push back: rear++，pop front: front++; 节省空间。Q.base[Q.rear]=d; Q.rear = (Q.rear+1)%MAXSIZE; 对 MAXSIZE 取余，保证了 rear 的最大值永远不会超过容量。注意 front=rear 不能区分队列 empty 还是 full，所以少用一个空间存储数字，即当 front = rear+1 is full;

(1) 实验步骤

1. 构造 initeQueue() 函数, 只需要为数组开辟一片空间，初始化 rear 和 front, 复杂度 O(1);

2. 析构为了和项目 1 隐式析构区别，现写显式调用的析构函数 DestroyQueue，只需释放数组空间，初始化 rear front, 复杂度 O(1);

3. queueSize, 利用 (Q.rear-Q.front+MAXSIZE)%MAXSIZE 直接计算，复杂度 O(1);
4. EnQueue, 新元素只需要在队尾入队列，若队满调整 MAXSIZE, 复杂度 O(1);
5. DeQueue，调整队首，front++, 再对 MAXSIZE 取余，保证值始终在数组下标内范围，复杂度 O（1）;
6. isEmpty 函数由于判 full 有 (size+1=MAXSIZE)，不会与判 empty(rear=front) 重复，直接根据第一个 if 判断, 复杂度 O（1）;
7. print 函数，需要遍历一遍输出，复杂度 O(n)（n 为队列元素个数）;

(2)　必要代码

```
/*
  一般队列:
  操作和栈类似（也采用继承linklist and array
      的方法），只是pop底，push头（数组的最后）------------这些都是单链表的插入和删除的特殊操作
  双端队列:
  限定插入和删除在两端进行(输出输入等设限），限定某端插入只能从该端删除=两个栈底相连的栈
  **循环队列 （以下实现） : array and Queue
   tip1:  空队列rear=front = 0
      push_back: rear++
      pop_front: front++
         to save space, 循环队列
   tip2:  front=rear can't distinguish blank and full
         way1: 记住队列元素的个数
      way2:  少用一个空间，front = rear+1 is full!

                        注：添加头结点，空队列即头尾指针都指向头结点

*/


#include<iostream>
#define MAXSIZE 5
using namespace std;

struct queue{
  int *base = NULL;
  int front= -100000;
  int rear = -200;
  //int count   ways1记录队列中元素总数，  判断empty ,下述采用ways2
};


class QueueList{
  public:
    queue Q;
```

```
QueueList(){

}
void initeQueue(){//构造一个空队列
    Q.base = new int(MAXSIZE);//顺序存储，创建一个大小为MAXSIZE的数组
  Q.front = Q.rear= 0;//头尾标记指向一个地方 ，初始化
  cout<<"build queue successfully!"<<endl;
}

void DestroyQueue(){
  delete [] Q.base;
  Q.base=NULL;
  Q.front = Q.rear = -10000; //将front 和rear
      置为负数，利于下面isEmpty()判断列表不存在的情况
  cout<<"destroy queue successfully!"<<endl;
}

int queueSize(){
  return (Q.rear-Q.front+MAXSIZE)%MAXSIZE;//rear 和front相对大小为size
}

void EnQueue(int d){
  if((Q.rear+1)%MAXSIZE==Q.front){//队列满（要留出一个空位）,实际容量比MAXSIZE小1
    cout <<"queue is full,can't push,please modify MAXSIZE"<<endl;
    return;
  }
  else if(Q.base == NULL){
    cout<<"queue is non_exist，please inite first!"<<endl;
  }
  Q.base[Q.rear]=d;
  Q.rear = (Q.rear+1) %MAXSIZE;//rear 向后挪，对MAXSIZE 取余，保证了 rear
      的最大值永远不会超过容量
}

void DeQueue(){
  if(isEmpty()){
    return;
  }
  if(Q.base==NULL){
    cout<<"queue is non_exist，please inite first!"<<endl;
    return ;
  }
  cout<<"the element out of queue is
      "<<Q.base[Q.front]<<endl;//出队列的永远为front标记所在的元素
  Q.front = (Q.front+1) % MAXSIZE; //front向后挪
}

bool isEmpty(){
  if(Q.front==Q.rear&&Q.front>0&&Q.base!=NULL){//只有front= rear 不够,不能判断 是empty
      or full
```

```
      cout<<"queue is empty"<<endl;//empty(front=rear,才开始，元素还没入队列)
      return true;              //full(size+1=MAXSIZE)
    }
    else if(Q.front<0){   //队列还没被初始化，不存在
      cout<<"queue is non_exist ,please inite first!"<<endl;
      return true;
    }
    cout<<"queue is not empty"<<endl;
    return false;
  }

  bool isNULL(){
    return Q.base==NULL;
  }
  void getHead(){
    if(queueSize()==0){
      cout<<"queue is empty,can't get head!'"<<endl;
      return ;
    }
    cout<<"the head is :"<<Q.base[Q.front]<<endl;
  }

  void Print(){
    if(queueSize()==0){
      cout<<"the queue is empty! can't print!";
    }
    int co =
        Q.front;//临时存储front的原来位置，因为队列中留出一个空位，循环一圈后不会回到原来位置
    int length = this->queueSize();
    while(Q.base!=NULL&&length>0){//跟随front++遍历一遍
      cout<<Q.base[Q.front++]<<" ";
       length--;
    }
    Q.front = co;//还原front标记的位置
    cout<<endl;
  }

};

int main(){
  char ch;
  int flag = 1;

  QueueList IQ;

  while(flag){
    cout<<"a(initie),b(destroy),c(size),d(in),e(out),f(empty)?,g(gethead),h(print),i(quit)"<<endl;
    cin>>ch;
    switch(ch){
      case 'a':{
```

```
      IQ.initeQueue();
      break;
    }
    case 'b':{
      IQ.DestroyQueue();
      break;
    }
    case 'c':{
      cout<<"queue already has elements of "<<IQ.queueSize()<<endl;
      break;
    }
    case 'd':{
      if(IQ.queueSize()==MAXSIZE-1){//队列满的情况，这样不用rear=front混淆empty 和full
        cout<<"queue is full,please modify MAXSIZE!"<<endl;
        break;
      }
      if(!IQ.isNULL()){
        int in;
        cout<<"input :";
        cin>>in;
        IQ.EnQueue(in);
        break;
      }
      cout<<"queue is not exist,please inite first!"<<endl;
      break;
    }
    case 'e':{
      IQ.DeQueue();
      break;
    }
    case 'f':{
      IQ.isEmpty();
      break;
    }
    case 'g':{
      IQ.getHead();
      break;
    }
    case 'h':{
      IQ.Print();
      break;
    }
    case 'i':{
      flag = 0;
      break;
    }
    default:{
      cout<<"input error!"<<endl;
      break;
    }
```

```
    }

  }

  return 0;
}
```

(3)　实验结果

见下图 2

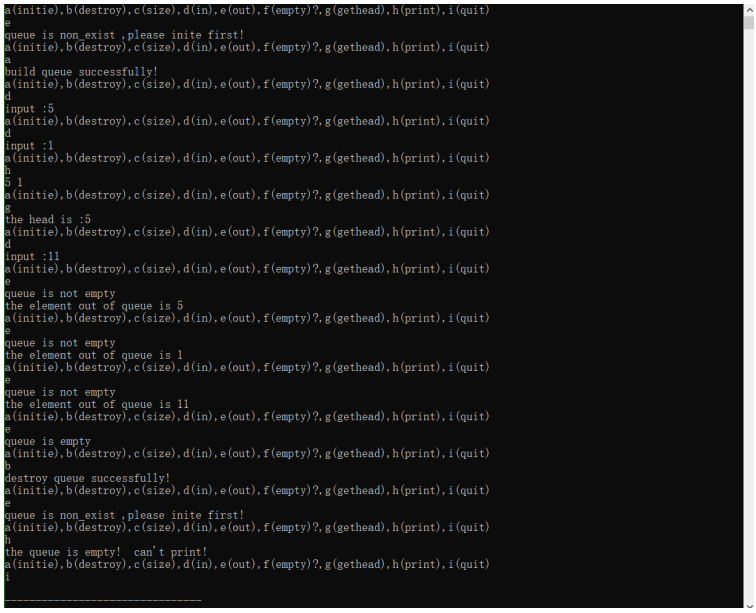

图 2: 队列的析构，构造，出入队，size,isEmpty () 等基本操作

3.3　项目 3

大致思想：利用栈结构特点存储计算后缀表达式，逐个读入字符，遇到数字就入栈，遇到计算符就 pop 出俩栈顶元素，并用这个运算符计算，再将结果 push 入栈，读入下一个字符。读完字符串后，取出栈顶元素，即最后结果。

(1)　实验步骤

1. compute 函数，判定运算符进行相应的计算，复杂度 O(1);
2. TransCharToNum 函数，将数字字符转换成真正的数字，复杂度 O（1）;
3. main 函数，逐个遍历读入字符串，计算 sum, 复杂度 O（n）（n 为字符串长度）;

(2)　必要代码

```
//平台第二题
#include<iostream>
#include<stack>
using namespace std;

double conpute(double a,double b,char c){//对从栈中弹出来的两个数字和一个符号做识别计算处理
  if(c=='+'){
    return a+b;
  }
  else if(c=='-'){
    return a-b;
  }
  else if(c=='*'){
    return a*b;
  }
  else {
    return a/b;
  }
}

double TransCharToNum(char a){//将符号转化成可以计算的数字
  return a-'0';
}



int main(){
  string str;
  stack <double>s; //栈专门存储数字
  double sum = 0;
  cin>>str;        //输入字符串
  for(int i = 0;i<str.size();i++){
    if(str[i] >= '0' && str[i] <= '9'){//遇到数字字符，就转化成真正的数字后push入栈
      s.push(TransCharToNum(str[i]));
    }
    else if(!s.empty()){
        //否则遇到运算符，开始计算,这个运算符肯定是计算刚刚push进去的栈顶的两个元素
      double FirstTop = s.top();
      s.pop();
      double SecondTop = s.top();
      s.pop();
      sum = conpute(SecondTop,FirstTop,str[i]);//取出栈顶两个元素，识别这个运算符并计算
      s.push(sum);              //将运算结果push入栈顶，参与下一个运算符号的运算
    }
  }
  printf("%f\n",s.top());//最后运算结果肯定是栈顶元素
  s.pop();           //清空栈

  return 0;
```

```
}
```

(3)　实验结果

见图 3

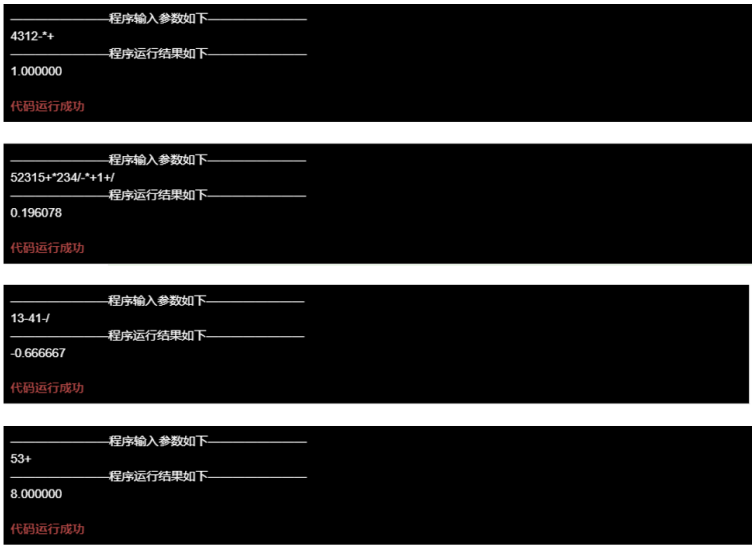

图 3: 利用栈的结构计算后缀表达式

3.4　项目 4.1

大致思想：利用栈结构特点存储自定义优先级的运算符，逐个读入字符串，遇到数字就打印，遇到非数字判断优先级，决定出入栈，注意实验步骤中描述的涉及到的用户输入容错处理。

(1)　实验步骤

1. 开头遇到正号无实际意义，直接跳过，遇到负号代表接下来的数字是个负数，直接打印，不归入运算符内；
2. 后缀表达式中间夹杂空格，跳过
3. 若遇到 +-,–,*-,/-,(-, 代表当前 a[i]= ‘-’ 后的数字为负数，直接输出符号，不是运算符
4. 若遇到)+, 空格 + 等一系列非数字后面接 + 号的情况，正号冗余，跳过;
5. main 函数内，真正处理运算符时，依照上述优先级判断入栈还是出栈，遍历一遍循环 n 次，循环内部判断优先级，pop 直到优先级相对大小变化时，最坏循环 n 次，综上，复杂度 O(n^2)（n 为字符串长度）

(2)　必要代码

```
#include<iostream>
#include<stack> //导入栈头文件，引用栈的结构存储运算符
#include<map> //键值对容器
using namespace std;

int main(){
 /*优先级比较原则：数字大的优先级大，左括号优先级最小，当前元素优先级小于等于栈顶元素，栈顶元素就要弹出，、
   当前元素在和新的栈顶元素比较，直到新的栈顶元素是左括号或者优先级小于当前运算符，直接push当前运算符
 */
 map<char,int> priority;//定义键值对char and int,规定运算符入栈出栈优先级
 priority['+'] = 1;
 priority['-'] = 1;
 priority['*'] = 2;
 priority['/'] = 2;

 int i = 0;
 string a;
 stack<char> s;
 getline(cin,a);//接收字符串a，依题目要求，遇到空格也要当作字符接收而不是标志结束输入

 //单独处理字符串最前面正负号的情形，负号代表紧接着的数字为负数，就直接输出这个负数
 if(a[0]=='-'){
   cout<<"-";
   i++;
 }
 //正号不输出，跳过
 else if(a[0]=='+'){
   i++;
 }

 while(i<a.size()){
   if(a[i]==' ') {//中间遇到空格，依题意跳过
     i++;
     continue;
   }

   if(a[i]<='9'&&a[i]>='0'){//数字直接输出
      cout<<a[i];
   }

   //若遇到+-,--,*-,/-,(-,代表当前a[i]='-'后的数字为负数，直接输出符号，不是运算符
   else if(a[i]=='-'&&(a[i-1]=='+'||a[i-1]=='-'||a[i-1]=='*'||a[i-1]=='/'||a[i-1]=='(')){
     cout<<'-';
   }
   //若遇)+,空格+等一系列非数字后面接+号的情况，这个正号跳过
   else if(a[i]=='+'&&a[i-1]!=')'&&a[i-1]!=' '&&!(a[i-1]<='9'&&a[i-1]>='0')) {
     i++;
```

```
  continue;
}
//遇到左括号就入栈
else if(a[i]=='('){
  s.push(a[i]);
}
//遇到右括号就pop出栈，直到遇到右括号为止
else if(a[i]==')'){
  char temp = s.top();
  while(temp!='('){
    cout<<temp;
    s.pop();
    temp = s.top();
  }
  s.pop();//pop出右括号
}
//如果遇到运算符
else if(a[i]=='+'||a[i]=='-'||a[i]=='*'||a[i]=='/'){
  char cur = a[i];

  if(s.empty()){//当前栈为空，不需要比较当前运算符与栈顶元素的优先级
    s.push(cur);
  }
  else{        //栈不为空，就要比较栈顶元素和当前运算符的优先级
    char top = s.top();
    if(top=='('){ //栈顶元素是左括号，直接push入栈
      s.push(cur);
      top = cur;
      i++;
      continue;
    }
    while(priority[cur]<=priority[top]&&!s.empty()){
      cout<<top;//当前元素优先级小于等于栈顶元素，pop栈顶，直到遇到新的栈顶元素是(或者当前元素优先级大于新栈顶元
      s.pop();
      if(!s.empty()){
        top = s.top();//判断在栈不空的情形下，获得新的栈顶元素
      }

      if(top=='('){
        s.push(cur);
          top = cur;
        break;
      }
    }
    if(priority[cur]>priority[top]||s.empty()){//当前元素优先级大于新栈顶元素，入栈
      s.push(cur);
    }


  }
```

```
    }
    i++;
    if(i==a.size()){//中缀表达式读取完毕
      while(!s.empty()){
      cout<<s.top();//pop出栈内剩余运算符
      s.pop();
      }
    }
  }
  while(!s.empty()){//彻底清空栈
    cout<<s.top();
    s.pop();
  }
  return 0;
}
```

(3)　实验结果

见图 4

图 4: 中缀表达式转换成后缀表达式

3.5　项目 4.2

即项目三

3.6　项目 4.3

大致思想：括号匹配属于经典栈的应用。逐个遍历字符串，但凡是左括号，为等待右括号匹配的待定项，入栈；若为右括号，与栈顶元素匹配，不匹配，本次检验失败，匹配，左括号出栈，进行下一轮读取。若栈空，没能匹配得到，匹配失败。遍历结束，栈空，说明都匹配成功。否则是匹配失败

(1)　实验步骤

1. main 函数，只需逐个遍历一次字符串，判断匹不匹配, 复杂度 O（n）（n 为字符串长度）;

(2)　必要代码

```
//括号匹配（栈的应用）
```

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

int main(){
  string str;
  stack <char> s;
  cin>>str;
  for(int i=0;i<str.size();i++){
    if(str[i]=='('){//但凡是左括号，为等待右括号匹配的待定项，入栈
      s.push('(');
    }
    else {      //若为右括号
      if(!s.empty()){
        if(s.top()=='('){//运气好，栈顶元素刚好与之匹配，左括号出栈，跳出本次，进行下一次循环
          s.pop();
          continue;
        }
        else{
          cout<<"NO"<<endl;//当前右括号str[i]没有与之匹配的，本次匹配失败，输出No,程序结束
          return 0;
        }
      }
      else{           //直到栈空，没能匹配当前str[i];输出No
        cout<<"NO"<<endl;
        return 0;
      }

    }
  }
  if(s.empty()){//最后结束，栈空，说明每个都匹配完毕
    cout<<"YES"<<endl;
  }
  else if(!s.empty()){//否则，还有未成对的，匹配失败
    cout<<"NO"<<endl;
  }
  return 0;
}
```

(3)　实验结果

见图 5

3.7　项目 4.4

大致思想：利用栈结构后进先出的特点存储转换过来的 n 进制数的每一位数，再逐个打印出栈顶元素和 pop，出来的顺序即进制转换的结果。

图 5: 栈的应用之括号匹配

(1)　实验步骤

1. conversion 函数，对十进制数累除，复杂度 O（$log_n num$），其中 n 为进制，num 为十进制数;

(2)　必要代码

```
//进制转换（栈的经典应用）
#include<iostream>
#include<stack>
using namespace std;

void conversion(int num,int n){
  stack <int> s;//进制转换每次取余，获得的个位数正好按照后进先出的排列顺序----栈结构
  while(num){
    s.push(num%n);//将每次取得的余数push入栈
    num/=n;//num除以n取整，更新num
  }
  while(!s.empty()){//逐个打印出n进制数
    cout<<s.top();
    s.pop();
  }
}

int main(){
  int num,n;
  cin>>num>>n;
  if(num==0){//单独考虑0的任何进制都是0
    cout<<"0"<<endl;
    return 0;
  }
  conversion(num,n);
  return 0;
}
```

(3) 实验结果

见图 6

图 6: 栈的应用之进制转换

四、实验总结

1. 进步：掌握栈和队列的基本操作（特别地，巩固了析构函数和循环队列的知识），掌握栈的三种应用：计算后缀表达式，进制转换，括号匹配。

2. 不足：程序设计考虑不周全, 不能一次性考虑周全所有情况；细节（尤其是链表操作头尾指针的单独处理操作）缺乏把控！