栈

栈的定义

栈是限定仅在表尾进行插入或者删除的线性表。对于栈来说，表尾端称为栈顶，表头端称为栈底。不含元素的空表称为空栈。因为栈限定在表尾进行插入或者删除，所以栈又被称为后进先出的线性表。在栈顶插入元素称为入栈，在栈顶删除元素称为出栈。

对栈中的元素进行操作只能从栈顶入手，但读取栈中的元素没有限制。·

顺序栈

栈的初始化

利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素。

通常定义时先为栈分配一个基本容量，使用过程中若空间不够再追加存储空间。

```
typedef struct S{
    int *base;//栈底指针
    int *top;//栈顶指针
    int StackSize;//当前已经分配的存储空间，以元素个数为单位。
}Stack;
```

按照设定的初始分配量进行第一次存储分配。base栈底指针，指向栈底。top栈顶指针，初值指向栈底。每当插入一个元素top++，删除一个元素top--。非空栈的栈顶指针指向栈顶元素的下一个位置。若base==top说明是空栈。

```
int InitStack(Stack *S){
    S->base=(int *)malloc(num*sizeof(int));//num是初始定义的元素个数
    if(S->base==NULL){
        return 0;
    }
    S->top=S->base;
    S->StackSize=num;
    return 1;
}
```

判断是否为空栈

```
int Judge_Null(Stack *S){
    if(S->top==S->base){
        return 1;//空栈
    }
    else{
        return 0;
    }
}
```

判断是否为满栈

```
int Judge_Full(Stack *S){
    if((S->top-S->base)==StackSize){
        return 1;//满栈
    }
    else{
        return 0;
    }
}
```

入栈

首先判断是否为满栈，若为满栈则追加存储空间。用realloc函数更改由malloc分配的内存。

函数原型：指针名=（数据类型*）realloc（要改变内存大小的指针名，新的大小）。

```
int Push(Stack *S,int Data){
    if((S->top-S->base)==S->StackSize){
        int *p=(int *)realloc(S->base,(S->StackSize+NUM)*sizeof(int));
        if(p==NULL){
            return 0;
        }
        S->base=p;
        S->top=S->base+S->StackSize;
        S->StackSize+=NUM;
    }//先插入元素，再移动栈顶指针。
    *(S->top)=Data;
    S->top++;
    return 1;
}
```

出栈

首先判断是否会下溢，若已为空栈则返回空指针。非空栈则top--。用整型指针传出被删除的数据。

```
int* Pop(Stack *S,int *Data){
    if(S->top==S->base){
        return NULL;
    }
    else{
        S->top--;
        Data=*(S->top);
        return Data;
    }
}
```

从栈底输出元素

```
void PrintStack(Stack *S){
    int *p=S->base;
    while(p!=S->top){
        printf("%d ",*p);
        p++;
    }
    printf("\n");
}
```

链栈

结构与单链表相同，使用带头结点的单链表实现，栈顶指针是链表的头指针。

```
typedef struct LS{
    LS *pNext;
    int Data;
}LinkedStack;
```

链栈的初始化

```
LinkedStack Init_LinkedStack(){
    LinkedStack *top=(LinkedStack *)malloc(sizeof(LinkedStack));
    if(top!=NULL){
        top->pNext=NULL;
    }
    return top;//链表的头指针。
}
```

判断是否为空栈

```
int Judge_Null(LinkedStack *top){
    if(top->pNext==NULL){//只有头结点。
        return 1;
    }
    else{
        return 0;
    }
}
```

入栈

令头结点的指针域指向新插入的栈顶元素，新插入元素的指针域指向原栈顶元素。

```
int Push(LinkedStack *top,int InData){
    LinkedStack *NodeToInsert=(LinkedStack *)malloc(sizeof(LinkedStack));
    if(NodeToInsert==NULL){
        return 0;
    }
    else{
        NodeToInsert->Data=InData;
        NodeToInsert->pNext=top->pNext;
        top->pNext=NodeToInsert;
        return 1;
    }
}
```

出栈

令头结点的指针域指向第二个结点，释放首元结点。可以用整型指针传出被删除的数据。

```
int* Pop(LinkedStack *top,int *OutData){
    if(top->pNext==NULL){
        return NULL;//已是空栈
    }
    else{
        LinkedStack *CurrentNode=top->pNext;
        *OutData=CurrentNode->Data;
        top->pNext=CurrentNode->pNext;
        free(CurrentNode);
        return OutData;
    }
}
```

数组模拟栈

定义top表示栈顶，初始值为-1。每当有一个数据入栈时，top+1,stack[top]=data。出栈时top-1。

```
typedef struct S{
    int StackSize;
    int top;
    int *Array;
}Stack;

int Judge_Null(Stack *S){
    if(S->top==-1){
        return 1;
    }
    else{
        return 0;
    }
}//若top为-1说明栈为空。

int Judge_Full(Stack *S){
    if(S->top==S->StackSize-1){
        return 1;
    }
    else{
        return 0;
    }
}//若top为StackSize-1说明栈以满。存储范围是0至StackSize-1。

Stack* Create_Stack(int Amount){//Amount为数组的最大容量。
    Stack S=(stack *)malloc(sizeof(Stack));
    if(S==NULL){
        return NULL;
    }
    S->Array=(Stack *)malloc(Amount*sizeof(Stack));
    if(S->Array==NULL){
        return NULL;
    }
    S->StackSize=Amount;
    S->top=-1;
    return S;
```

```
}

int Push(Stack *S,int Data){
    if(Judge_Full(S)){
        return 0;
    }
    S->Array[++S->top]=Data;
    return 1;
}

int Pop(Stack *S){
    if(Judge_Null(S)){
        return 0;
    }
    S->top--;
    return 1;
}
```

短除法进制转换

以十进制转化为二进制为例：十进制中*11*转化为二进制。*11<2^4=16*，所以二进制数为*4*位，设为*abcd*，且*abcd*只能为*0*或*1*。

*a*2^3+b*2^2+c*2^1+d=11*，提取前三项的公因数*2*，可以化为*2*(a*2^2+b*2^1+c)+d=11*。

又因为*d*只能是*0*或*1*，所以：*11/2=(a*2^2+b*2^1+c)*余*d*。

*a*2^2+b*2^1+c=5*，*d=1*。

与上述同理，*5/2=(a*2^1+b)*余*c*。*a*2^1+b=2*，*c=1*。

*2/2=a*余*b*，*b=0*，*a=1*。

二进制数*1011*为十进制数除以*2*取余数后逆序排列。

我们可以用这样的方式来表示一个十进制数: 将每个阿拉伯数字乘以一个以该数字所处位置为指数,以 10 为底数的幂之和的形式。与之相似的，对二进制数来说，也可表示成每个二进制数码乘以一个以该数字所处位置为指数,以 2 为底数的幂之和的形式。

一般说来，任何一个正整数 R 或一个负整数 -R 都可以被选来作为一个数制系统的基数。如果是以 R 或 -R为基数,则需要用到的数码为 0,1,....R-1。

例如当 R=7时,所需用到的数码是 0,1,2,3,4,5,6，这与其是 R 或 -R无关。如果作为基数的数绝对值超过 10,则为了表示这些数码，通常使用英文字母来表示那些大于 9 的数码。例如对 16进制数来说,用 A 表示 10,用 B 表示 11，用 C表示 12，以此类推。

在负进制数中是用 -R作为基数，例如 -15（十进制）相当于 110001（-2进制），并且它可以被表示为 2 的幂级数的和数：

110001=1×(−2)^5+1×(−2)^4+0×(−2)^3+0×(−2)^2+0×(−2)^1+1×(−2)^0。

设计一个程序,读入一个十进制数和一个负进制数的基数, 并将此十进制数转换为此负进制下的数。

输入的每行有两个输入数据。第一个是十进制数 n。 第二个是负进制数的基数 -R。对于 100% 的数据，−20≤R≤−2，|n|≤37336。

```
#include <stdio.h>
#include <stdlib.h>

typedef struct S{
```

```
    int *base;
    int *top;
    int StackSize;
}Stack;

int InitStack(Stack *S);
int Push(Stack *S,int Data);
void PrintStack(Stack *S);

int main(){
    int n,R;
    scanf("%d%d",&n,&R);
    printf("%d=",n);
    Stack *S=(Stack*)malloc(sizeof(Stack));
    InitStack(S);
    while(n!=0){
        int q,r;//商，余数
        q=n/R;
        r=n-q*R;
        while(r<0){
            r=r-R;
            q++;
        } //余数应大于等于0，若小于0则+除数的绝对值，也就是-除数。余数-除数则商+1。
        Push(S,r);//将每次的余数入栈，之后从栈顶到栈底输出。
        n=q;
    }
    PrintStack(S);
    printf("(base%d)",R);
    return 0;
}

int InitStack(Stack *S){
    S->base=(int *)malloc(20*sizeof(int));
    if(S->base==NULL){
        return 0;
    }
    S->top=S->base;
    S->StackSize=20;
    return 1;
}

int Push(Stack *S,int Data){
    if((S->top-S->base)==S->StackSize){
        int *p=(int *)realloc(S->base,(S->StackSize+10)*sizeof(int));
        if(p==NULL){
            return 0;
        }
        S->base=p;
        S->top=S->base+S->StackSize;
        S->StackSize+=10;
    }
    *(S->top)=Data;
    S->top++;
    return 1;
}

void PrintStack(Stack *S){
    int *p=(S->top)-1;//第一个数从top的下一个位置开始。
```

```
    while(p!=(S->base)-1){//最后一个数在base的位置。
        if(*p<10){
            printf("%d",*p);
        }
        else{
            printf("%c",*p-10+'A');//10以上进行转换。
        }
        p--;
    }
}
```

后缀表达式

式中不再引用括号，运算符号放在两个运算对象之后，所有计算按运算符号出现的顺序，严格地由左而右进行，不用考虑运算符的优先级。如：3*(5-2)+7对应的后缀表达式为：3．5．2．-*7．+@。’@’为表达式的结束符号。‘.’为操作数的结束符号。

输入后缀表达式（长度1000以内），输出表达式的值。

思路：先读入后缀表达式，再将数字按照读取顺序存入栈中。读取数字的方法：先按位读取，读到 . 表示数字结束。遇到运算符号时，将栈顶的两个数取出来运算，再存入栈中。最后输出栈中剩余的最后一个数字即可。

```
#include <stdio.h>
#include <stdlib.h>

typedef struct S{
    int *base;
    int *top;
    int StackSize;
}Stack;

int InitStack(Stack *S);
int Push(Stack *S,int Data);

int main(){
    Stack *S=(Stack*)malloc(sizeof(Stack));
    InitStack(S);
    int Current=0;//当前读取的数字
    char ch=getchar();
    while(ch!='@'){
        if(ch>='0'&&ch<='9'){
            Current*=10;
            Current+=ch-48;
        }
        else if(ch=='.'){
            Push(S,Current);
            Current=0;
        }
        else{
            int Result;
            switch(ch){
                case '+':
                    Result=(*(S->top-1))+(*(S->top-2));
                    break;
                case '-':
                    Result=(*(S->top-2))-(*(S->top-1));
```

```
                    break;
                case '*':
                    Result=(*(S->top-1))*(*(S->top-2));
                    break;
                case '/':
                    Result=(*(S->top-2))/(*(S->top-1));
                    break;
            }
            (S->top)-=2;
            Push(S,Result);
        }
        ch=getchar();
    }
    printf("%d",*S->base);
    return 0;
}

int InitStack(Stack *S){
    S->base=(int *)malloc(20*sizeof(int));
    if(S->base==NULL){
        return 0;
    }
    S->top=S->base;
    S->StackSize=20;
    return 1;
}

int Push(Stack *S,int Data){
    if((S->top-S->base)==S->StackSize){
        int *p=(int *)realloc(S->base,(S->StackSize+10)*sizeof(int));
        if(p==NULL){
            return 0;
        }
        S->base=p;
        S->top=S->base+S->StackSize;
        S->StackSize+=10;
    }
    *(S->top)=Data;
    S->top++;
    return 1;
}
```

表达式的转换

平常我们书写的表达式称为中缀表达式，因为它将运算符放在两个操作数中间，许多情况下为了确定运算顺序，括号是不可少的，而中缀表达式就不必用括号了。

后缀标记法：书写表达式时采用运算紧跟在两个操作数之后，从而实现了无括号处理和优先级处理，使计算机的处理规则简化为：从左到右顺序完成计算，并用结果取而代之。

例如：`8-(3+2*6)/5+4` 可以写为：`8 3 2 6*+5/-4+`

其计算步骤为：

```
8 3 2 6 * + 5 / - 4 +
8 3 12 + 5 / - 4 +
8 15 5 / - 4 +
8 3 - 4 +
5 4 +
9
```

编写一个程序，完成这个转换，要求输出的每一个数据间都留一个空格。

输入是一个中缀表达式。输入的符号中只有这些基本符号`0123456789+-*/^()`，并且不会出现形如`2*-3`的格式。表达式中的基本数字也都是一位的，不会出现形如`12`形式的数字。

输出若干个后缀表达式，第`I+1`行比第`I`行少一个运算符和一个操作数，最后一行只有一个数字，表示运算结果。

思路：先将中缀表达式转化为后缀表达式。后缀表达式相当于先计算了中缀表达式中括号里的部分，按运算优先级依次计算。所以我们把运算符号存入栈中。如果遇到'('就直接存入，遇到')'时，将栈中符号弹出并存入字符串数组，直到遇到第一个'('为止。对于每个运算符号，入栈之前先比较自己与栈顶运算符号的优先级，若自己更优先就入栈，否则将栈中符号依次弹出并存入字符串数组，直到遇到比当前符号优先级低的，将当前符号存入栈中。若遇到数字直接存入字符串数组即可。读取完中缀表达式后，若栈中还有运算符号则依次弹出并存入字符串数组中。所得字符串数组即为后缀表达式。

转化为后缀表达式后，按照与上一道题中类似的方法计算结果就可以了。注意每进行一次运算都要输出一次当前结果，要先输出数字栈中的数据，再输出字符串中的数据。直到算出数为止。第一次运算前也要输出一次。

```c
//太长了，别看了
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

typedef struct S{
    char *base;
    char *top;
    int StackSize;
}Stack;

typedef struct SS{
    int *base;
    int *top;
    int StackSize;
}Stack_B;

char str[1005];//后缀表达式
int InitStack(Stack *S);
int Push(Stack *S,char Data);
int InitStack_B(Stack_B *S);
int Push_B(Stack_B *S,int Data);
void PrintStack(Stack_B *S);

int main(){
    Stack *S=(Stack*)malloc(sizeof(Stack));
    InitStack(S);
    char ch;
    int i=0;
    while(scanf("%c",&ch)!=EOF){
```

```
        if(ch>='0'&&ch<='9'){
            str[i]=ch;
            i++;
        }
        else{
            switch(ch){
                case '(':
                    Push(S,ch);
                    break;
                case ')':
                    while((*(S->top-1))!='('){
                        str[i]=*(S->top-1);
                        i++;
                        S->top--;
                    }
                    S->top--;
                    break;
                case '^':
                    Push(S,ch);
                    break;
                case '*':
                case '/':
                    while(((*(S->top-1))!='+')&&((*(S->top-1))!='-')&&((*(S-
>top-1))!='(')&&(S->top!=S->base)){//入栈条件为：栈为空栈；(后第一个符号；+-之后。
                        str[i]=*(S->top-1);
                        i++;
                        S->top--;
                    }
                    Push(S,ch);
                    break;
                case '+':
                case '-':
                    while((S->top!=S->base)&&((*(S->top-1))!='(')){//入栈条件为：栈
为空栈；(后第一个符号。
                        str[i]=*(S->top-1);
                        i++;
                        S->top--;
                    }
                    Push(S,ch);
                    break;
            }
        }
    }
    while(S->top!=S->base){
        str[i]=*(S->top-1);
        i++;
        S->top--;
    }
    //后缀表达式转换完毕
    int L=strlen(str);
    for(i=0;i<L;i++){
        printf("%c ",str[i]);
    }
    printf("\n");
    //边计算边输出
    Stack_B *S2=(Stack_B*)malloc(sizeof(Stack_B));
    InitStack_B(S2);
    for(i=0;i<L;i++){
```

```
        if(str[i]>='0'&&str[i]<='9'){
            int Current=str[i]-48;
            Push_B(S2,Current);
        }
        else{
            int Result;
            switch(str[i]){
                case '+':
                    Result=(*(S2->top-1))+(*(S2->top-2));
                    break;
                case '-':
                    Result=(*(S2->top-2))-(*(S2->top-1));
                    break;
                case '*':
                    Result=(*(S2->top-1))*(*(S2->top-2));
                    break;
                case '/':
                    Result=(*(S2->top-2))/(*(S2->top-1));
                    break;
                case '^':
                    Result=pow((*(S2->top-2)),(*(S2->top-1)));
                    break;
            }
            (S2->top)-=2;
            Push_B(S2,Result);
            PrintStack(S2);
            int j=i+1;
            for(j=i+1;j<L;j++){
                printf("%c ",str[j]);
            }
            printf("\n");
        }
    }
    return 0;
}

int InitStack(Stack *S){
    S->base=(char *)malloc(20*sizeof(char));
    if(S->base==NULL){
        return 0;
    }
    S->top=S->base;
    S->StackSize=20;
    return 1;
}

int Push(Stack *S,char Data){
    if((S->top-S->base)==S->StackSize){
        char *p=(char *)realloc(S->base,(S->StackSize+10)*sizeof(char));
        if(p==NULL){
            return 0;
        }
        S->base=p;
        S->top=S->base+S->StackSize;
        S->StackSize+=10;
    }
    *(S->top)=Data;
    S->top++;
```

```
    return 1;
}

int InitStack_B(Stack_B *S){
    S->base=(int *)malloc(20*sizeof(int));
    if(S->base==NULL){
        return 0;
    }
    S->top=S->base;
    S->StackSize=20;
    return 1;
}

int Push_B(Stack_B *S,int Data){
    if((S->top-S->base)==S->StackSize){
        int *p=(int *)realloc(S->base,(S->StackSize+10)*sizeof(int));
        if(p==NULL){
            return 0;
        }
        S->base=p;
        S->top=S->base+S->StackSize;
        S->StackSize+=10;
    }
    *(S->top)=Data;
    S->top++;
    return 1;
}

void PrintStack(Stack_B *S){
    int *p=S->base;
    while(p!=S->top){
        printf("%d ",*p);
        p++;
    }
}
```

也可以用数组模拟栈，思路是一样的。

判断入栈条件时也可以先规定符号的优先级：

1.从左到右运算，越靠左的符号优先级越高。因为左边的符号先被读取，先入栈，所以新符号入栈时要先弹出已入栈的同级符号，直到遇到低一级的符号。

2.+-的入栈条件为：栈为空栈或(后第一个符号。*/的入栈条件为：栈为空栈或(后第一个符号或+-之后。^不需要判断直接入栈。

3、遇到)则弹出所有符号直至遇到(。

定义一个函数判断+-*/^(的优先级，返回优先级的数值，越大越优先：

```
int Level(char ch){
    if(ch=='*'||ch=='/'){
        return 2;
    }
    else if(ch=='('){
        return 0;
    }
    else if(ch=='^'){
```

```
        return 3;
    }
    else{//+-的情况
        return 1;
    }
}
```

之后定义两个数组分别表示符号栈和后缀栈。读取数字入后缀栈，符号入符号栈，从符号栈弹出进入后缀栈。之后定义数组表示数字栈，计算后缀表达式。

完整代码如下：

```
#include <stdio.h>
#include <math.h>

int Level(char ch){
    if(ch=='*'||ch=='/') return 2;
    else if(ch=='(') return 0;
    else if(ch=='^') return 3;
    else return 1;
}

int Record=-1,Top=-1;
//用于计数。
int NumStack[100];//数字栈。
char Postfix[200],Operator[100];
//分别为：后缀栈，符号栈。

int main(){
    char Current;
    while(scanf("%c",&Current)!=EOF){
        if(Current>='0'&&Current<='9'){
            Postfix[++Record]=Current;//数字直接入后缀栈
        }
        else if(Current=='('){
            Operator[++Top]=Current;//(直接入符号栈
        }
        else if(Current==')'){
            while(Operator[Top]!='('){
                Postfix[++Record]=Operator[Top];//符号栈弹出符号，入后缀栈
                Top--;
            }
            Top--;//弹出(
        }
        else{//读取符号
            while(Level(Operator[Top])>=Level(Current)&&Top>=0){//不符合入栈条件时
                Postfix[++Record]=Operator[Top];//符号栈弹出符号，入后缀栈
                Top--;
            }
            Operator[++Top]=Current;//新符号入栈
        }
    }
    //读取完毕后若符号栈中还有符号，都弹出
    while(Top>=0){
        Postfix[++Record]=Operator[Top];//符号栈弹出符号，入后缀栈
        Top--;
    }
```

```
    Postfix[++Record]='\0';//字符串数组的结尾，后缀表达式转换完成
    int i=0;//用于输出计数
    while(Postfix[i]!='\0'){
        printf("%c ",Postfix[i]);
        i++;
    }
    printf("\n");
    //之后计算后缀表达式的值，每计算一步输出一次当前值
    int Scan=0;//遍历后缀栈
    Top=-1;//数字栈的栈顶
    while(Postfix[Scan]!='\0'){
        if(Postfix[Scan]>='0'&&Postfix[Scan]<='9'){
            NumStack[++Top]=Postfix[Scan]-48;
        }
        else{
            switch(Postfix[Scan]){
                case '+':
                    NumStack[Top-1]=NumStack[Top-1]+NumStack[Top];
                    break;
                case '-':
                    NumStack[Top-1]=NumStack[Top-1]-NumStack[Top];
                    break;
                case '*':
                    NumStack[Top-1]=NumStack[Top-1]*NumStack[Top];
                    break;
                case '/':
                    NumStack[Top-1]=NumStack[Top-1]/NumStack[Top];
                    break;
                case '^':
                    NumStack[Top-1]=pow(NumStack[Top-1],NumStack[Top]);
                    break;
            }
            Top--;
            for(i=0;i<=Top;i++){
                printf("%d ",NumStack[i]);
            }
            i=Scan+1;
            while(Postfix[i]!='\0'){
                printf("%c ",Postfix[i]);
                i++;
            }
            printf("\n");
        }
        Scan++;
    }
    return 0;
}
```