# 26栈和队列4

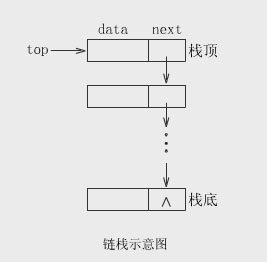
## 栈的链式存储结构

讲完了栈的顺序存储结构，也给大家结合了一些例题演练，相信大家对栈再也不陌生了吧？

现在我们来看下栈的链式存储结构，简称栈链。（通常我们用的都是栈的顺序存储结构存储，链式存储我们作为一个知识点，大家知道就好！）

栈因为只是栈顶来做插入和删除操作，所以比较好的方法就是将栈顶放在单链表的头部，栈顶指针和单链表的头指针合二为一。

No pic you say a J8……



teypedef struct StackNode

{

ElemType data; // 存放栈的数据

struct StackNode \*next;

} StackNode, \*LinkStackPtr;

teypedef struct LinkStack

{

LinkStackPrt top; // top指针

int count; // 栈元素计数器

}

## 进栈操作

* 对于栈链的Push操作，假设元素值为e的新结点是s，top为栈顶指针，我们得到如下代码：

Status Push(LinkStack \*s, ElemType e)

{

LinkStackPtr p = (LinkStackPtr) malloc (sizeof(StackNode));

p->data = e;

p->next = s->top;

s->top = p;

s->count++;

return OK;

}

## 出栈操作

* 至于链栈的出战Pop操作，假设变量p用来存储要删除的栈顶结点，将栈顶指针下移一位，最后释放p即可。

Status Pop(LinkStack \*s, ElemType \*e)

{

LinkStackPtr p;

if( StackEmpty(\*s) ) // 判断是否为空栈

return ERROR;

\*e = s->top->data;

p = s->top;

s->top = s->top->next;

free(p);

s->count--;

return OK;

}

## 终极实践

在讲解这道例题的时候，请允许小甲鱼花一点点的时间对小学时候的数学老师进行感谢，嗯，谢谢您，让我学会如何计算以下这道表达式，并且认为它十分简单：(1-2)\*(4+5)

人类早就熟悉这种中缀表达式的计算方式，随便拉一个小朋友过来，给他一颗糖，他会马上告诉你这个结果应该是等于-9，因为括号里边的要先进行计算。

但是计算机不喜欢了，因为我们有小括号中括号大括号，还允许一个嵌套一个，这样子计算机就要进行很多次if判断才行决定哪里先计算。

## 逆波兰表达式

后来，在20世纪三十年代，波兰逻辑学家Jan.Lukasiewicz不知道是像牛顿一样被苹果砸到脑袋而想到万有引力原理，或者还是像阿基米德泡在浴缸里突发奇想给皇冠是否纯金做验证，总之他也是灵感闪现了，然后发明了一种不需要括号的后缀表达式，我们通常把它称为逆波兰表达式(RPN) 。

很多鱼油好奇为什么他发明的东西是以他的国籍而不是以他的名字命名的呢？这也告诉我们，想要流芳百世，名字还得起得朗朗上口才行。

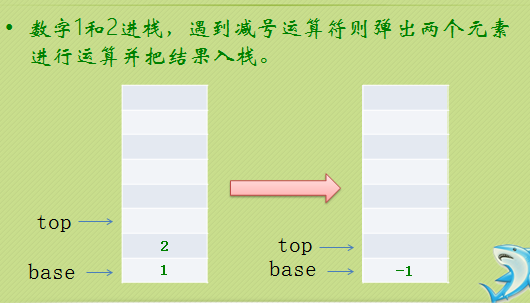
我们先来看看，对于(1-2)\*(4+5)，如果用逆波兰表示法，应该是这样：1 2 – 4 5 + \*

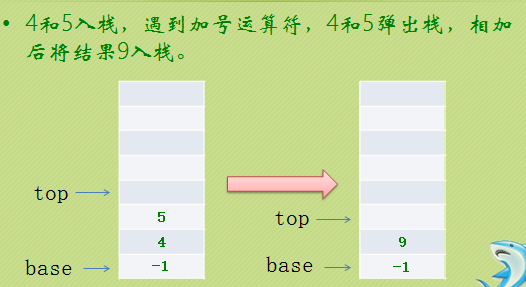
这种方式敢情我们人类是不大好接受的了，不过对于计算机来说，那可是喜爱至极。

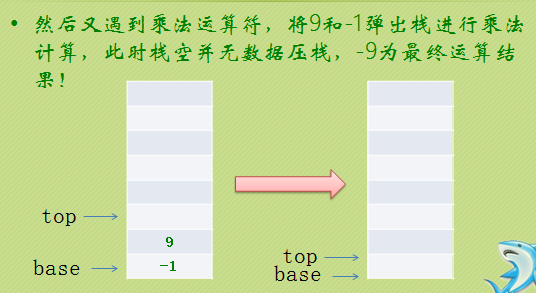
因为只需要利用栈的特点，就可以将这种后缀表达式的性能发挥到极致。

解析来就让小甲鱼图文并茂的解释一下吧！

No pic you say a J8……







RPN.c

#include <stdio.h>

#include <ctype.h>

#include <stdlib.h>

#define STACK\_INIT\_SIZE 20

#define STACKINCREMENT 10

#define MAXBUFFER 10

typedef double ElemType;

typedef struct

{

ElemType \*base;

ElemType \*top;

int stackSize;

}sqStack;

InitStack(sqStack \*s)

{

s->base = (ElemType \*)malloc(STACK\_INIT\_SIZE \* sizeof(ElemType));

if( !s->base )

exit(0);

s->top = s->base;

s->stackSize = STACK\_INIT\_SIZE;

}

Push(sqStack \*s, ElemType e)

{

// 栈满，追加空间，鱼油必须懂！

if( s->top - s->base >= s->stackSize )

{

s->base = (ElemType \*)realloc(s->base, (s->stackSize + STACKINCREMENT) \* sizeof(ElemType));

if( !s->base )

exit(0);

s->top = s->base + s->stackSize;

s->stackSize = s->stackSize + STACKINCREMENT;

}

\*(s->top) = e; // 存放数据

s->top++;

}

Pop(sqStack \*s, ElemType \*e)

{

if( s->top == s->base )

return;

\*e = \*--(s->top); // 将栈顶元素弹出并修改栈顶指针

}

int StackLen(sqStack s)

{

return (s.top - s.base);

}

int main()

{

// 亲爱的鱼油，请从此处开始写代码！

return 0;

}

# 27逆波兰计算器

## 逆波兰计算器

这节课小甲鱼带大家一起写一个逆波兰计算器，需要实现以下要求：

实现对逆波兰输入的表达式进行计算。

支持带小数点的数据。

正常的表达式 ---> 逆波兰表达式

a+b ---> a b +

a+(b-c) ---> a b c - +

a+(b-c)\*d ---> a b c - d \* +

a+d\*(b-c)---> a d b c - \* +

RPN.c

#include <stdio.h>

#include <ctype.h>

#include <stdlib.h>

#define STACK\_INIT\_SIZE 20

#define STACKINCREMENT 10

#define MAXBUFFER 10

typedef double ElemType;

typedef struct

{

ElemType \*base;

ElemType \*top;

int stackSize;

}sqStack;

InitStack(sqStack \*s)

{

s->base = (ElemType \*)malloc(STACK\_INIT\_SIZE \* sizeof(ElemType));

if( !s->base )

exit(0);

s->top = s->base;

s->stackSize = STACK\_INIT\_SIZE;

}

Push(sqStack \*s, ElemType e)

{

// 栈满，追加空间，鱼油必须懂！

if( s->top - s->base >= s->stackSize )

{

s->base = (ElemType \*)realloc(s->base, (s->stackSize + STACKINCREMENT) \* sizeof(ElemType));

if( !s->base )

exit(0);

s->top = s->base + s->stackSize;

s->stackSize = s->stackSize + STACKINCREMENT;

}

\*(s->top) = e; // 存放数据

s->top++;

}

Pop(sqStack \*s, ElemType \*e)

{

if( s->top == s->base )

return;

\*e = \*--(s->top); // 将栈顶元素弹出并修改栈顶指针

}

int StackLen(sqStack s)

{

return (s.top - s.base);

}

int main()

{

sqStack s;

char c;

double d, e;

char str[MAXBUFFER];

int i = 0;

InitStack( &s );

printf("请按逆波兰表达式输入待计算数据，数据与运算符之间用空格隔开，以#作为结束标志: \n");

scanf("%c", &c);

while( c != '#' )

{

while( isdigit(c) || c=='.' ) // 用于过滤数字

{

str[i++] = c;

str[i] = '\0';

if( i >= 10 )

{

printf("出错：输入的单个数据过大！\n");

return -1;

}

scanf("%c", &c);

if( c == ' ' )

{

d = atof(str);

Push(&s, d);

i = 0;

break;

}

}

switch( c )

{

case '+':

Pop(&s, &e);

Pop(&s, &d);

Push(&s, d+e);

break;

case '-':

Pop(&s, &e);

Pop(&s, &d);

Push(&s, d-e);

break;

case '\*':

Pop(&s, &e);

Pop(&s, &d);

Push(&s, d\*e);

break;

case '/':

Pop(&s, &e);

Pop(&s, &d);

if( e != 0 )

{

Push(&s, d/e);

}

else

{

printf("\n出错：除数为零！\n");

return -1;

}

break;

}

scanf("%c", &c);

}

Pop(&s, &d);

printf("\n最终的计算结果为：%f\n", d);

return 0;

}

// 5 - (6 + 7) \* 8 + 9 / 4

// 5 - 13 \* 8 + 9 / 4

// 5 - 104 + 2.25

// -99 + 2.25

// 5 6 7 + 8 \* - 9 4 / +

# 28中缀表达式转换为后缀表达式1

## 课前谈一谈

今天我们课前谈一谈，要说点什么好呢？

最近小甲鱼发现，很多鱼油在学习数据结构和算法的时候积极性已经开始有点下降了。甚至很多朋友怀疑数据结构和算法到底有没有用？

实话说，在大厦的防震设计、消除疾病、防止水源枯竭这些实际问题中，很遗憾，数据结构和算法几乎起不到任何直接作用。。。。。。

那为什么我们要学呢？

很简单，它可以锻炼我们的“高级”思维！

何为“高级”思维？

这所谓的“高级”也是小甲鱼自己发明的，算法的重要性不用说大家都知道，一个程序，特别是大型程序，优秀的算法和架构跟一般的算法和架构效率差别是千万倍！这就可以解释为什么国产的几大应用都在前几天相继投入血本进行重构。

这就跟建高楼大厦要打好根基是一个道理，很多人喜欢当“暴发户”，根基没打好就开始盖房，但盖到四五层的时候发现根基不稳，拆掉重盖！

## 中缀表达式转换为后缀表达式

小甲鱼上节课带大家编写了一个逆波兰计算器。

但是，我们人类确实是喜欢这样的表达式：

(1-2)\*(4+5)

而不是这样的：

1 2 – 4 5 + \*

所以，我们这节课的任务就是编写一个程序，将用户输入的中缀表达式转换为后缀表达式，而作为课后作业的延生，要求大家动手写一个中缀表达式计算器！

那么如何将“(1-2)\*(4+5)”转化为“1 2 – 4 5 + \*”呢？

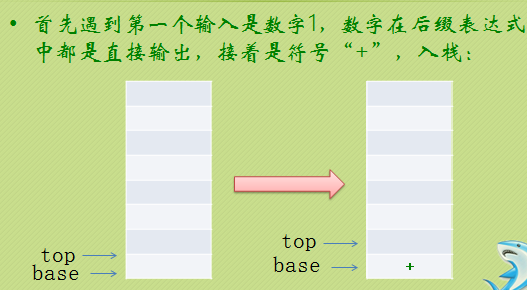
其实很简单，利用栈的“记忆”吧，符号都推入栈即可。

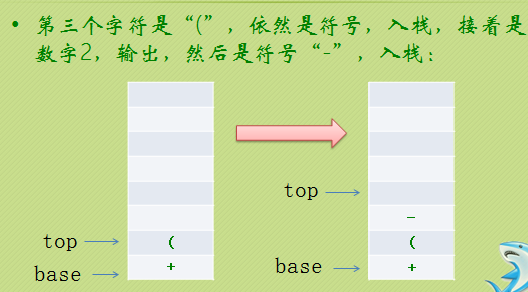
我们大家很清楚我们将要进入看图识字环节了！

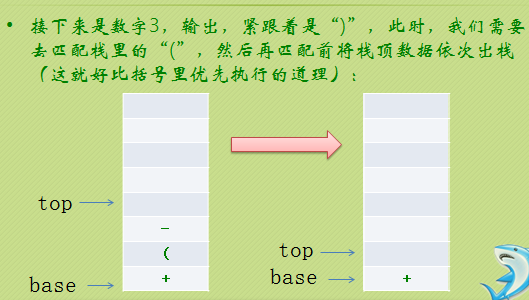
为了使得问题变得更加复杂，我们把假想敌设为：1+(2-3)\*4+10/5

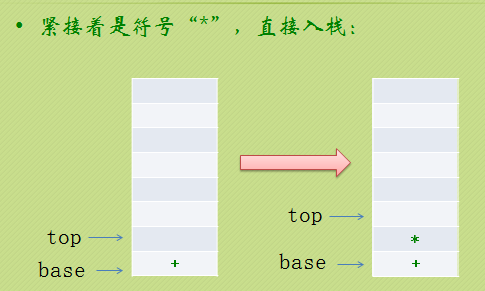
No pic you say a J8……

提示：备纸和笔随时记下输出内容！

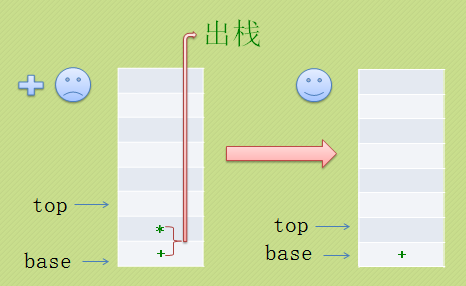


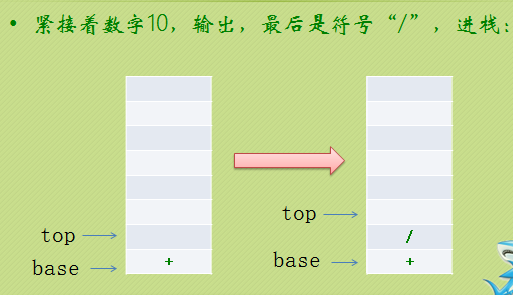






* 遇到数字4，输出，之后是符号“+”，此时栈顶元素是符号“\*”，按照先乘除后加减原理，此时栈顶的乘号优先级比即将入栈的加好要大，所以出栈。
* 栈中第二个元素是加好，按理来说大家平起平坐，但是按照先到先来后到吃屎的原则，栈里的加好呆得太久了，也要出栈透透气。（同理如果栈里还有其他操作符，也是出栈）
* 最后把刚刚吃屎的那个加好入栈，操作如下图：





* 最后一个数字5，输出，所有的输入处理完毕，但是栈中仍然有数据，所以将栈中符号依次出栈。
* 有些东西看上去很难，但事实上做起来会更加的麻烦，就像这道题，小甲鱼要求大家务必经过自己的思考，再跟着小甲鱼来打代码，一定会有更大的收获！
* 总结规则：从左到右遍历中缀表达式的每个数字和符号，若是数字则直接输出，若是符号，则判断其与栈顶符号的优先级，是右括号或者优先级低于栈顶符号，则栈顶元素依次出栈并输出，直到遇到左括号或栈空才将吃屎的那个符号入栈。

#include <stdio.h>

#include <stdlib.h>

#define STACK\_INIT\_SIZE 20

#define STACKINCREMENT 10

typedef char ElemType;

typedef struct

{

ElemType \*base;

ElemType \*top;

int stackSize;

}sqStack;

InitStack(sqStack \*s)

{

s->base = (ElemType \*)malloc(STACK\_INIT\_SIZE \* sizeof(ElemType));

if( !s->base )

exit(0);

s->top = s->base;

s->stackSize = STACK\_INIT\_SIZE;

}

Push(sqStack \*s, ElemType e)

{

// 栈满，追加空间，鱼油必须懂！

if( s->top - s->base >= s->stackSize )

{

s->base = (ElemType \*)realloc(s->base, (s->stackSize + STACKINCREMENT) \* sizeof(ElemType));

if( !s->base )

exit(0);

s->top = s->base + s->stackSize;

s->stackSize = s->stackSize + STACKINCREMENT;

}

\*(s->top) = e; // 存放数据

s->top++;

}

Pop(sqStack \*s, ElemType \*e)

{

if( s->top == s->base )

return;

\*e = \*--(s->top); // 将栈顶元素弹出并修改栈顶指针

}

int StackLen(sqStack s)

{

return (s.top - s.base);

}

int main()

{

sqStack s;

char c, e;

InitStack( &s );

printf("请输入中缀表达式，以#作为结束标志：");

scanf("%c", &c);

while( c != '#' )

{

while( c>='0' && c<='9' )

{

printf("%c", c);

scanf("%c", &c);

if( c<'0' || c>'9' )

{

printf(" ");

}

}

if( ')' == c )

{

Pop(&s, &e);

while( '(' != e )

{

printf("%c ", e);

Pop(&s, &e);

}

}

else if( '+'==c || '-'==c )

{

if( !StackLen(s) )

{

Push(&s, c);

}

else

{

do

{

Pop(&s, &e);

if( '(' == e )

{

Push(&s, e);

}

else

{

printf("%c ", e);

}

}while( StackLen(s) && '('!=e );

Push(&s, c);

}

}

else if( '\*'==c || '/'==c || '('==c )

{

Push(&s, c);

}

else if( '#'== c )

{

break;

}

else

{

printf("\n出错：输入格式错误！\n");

return -1;

}

scanf("%c", &c);

}

while( StackLen(s) )

{

Pop(&s, &e);

printf("%c ", e);

}

return 0;

}

# 29栈和队列7

## 神马是队列

这就是队列



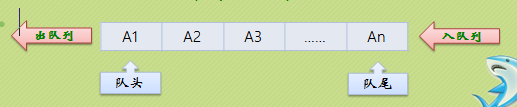
* 这不是队列





## 队列的定义

* 队列（queue）是只允许在一端进行插入操作，而在另一端进行删除操作的线性表。
* 与栈相反，队列是一种先进先出（First In First Out, FIFO）的线性表。
* 与栈相同的是，队列也是一种重要的线性结构，实现一个队列同样需要顺序表或链表作为基础。



此前我们用浏览器的历史记录作为栈的例子让大家了解她的应用广泛，队列在现实中也是应用十分广泛。

我们的输入缓冲区接受键盘的输入就是按队列的形式输入和输出的，不然的话就很容闹出问题。

例如有一天你突然心血来潮，用企鹅发了一句“You are my god”给你女朋友，表示她就是你的全部，但是输入缓冲区在输入god这个单词的时候不用队列这个结构而用栈的结构，就变成了“You are my dog”发出去了。。。。。。

## 队列的链式存储结构

队列既可以用链表实现，也可以用顺序表实现。跟栈相反的是，栈一般我们用顺序表来实现，而队列我们常用链表来实现(节省空间)，简称为链队列。

typedef struct QNode {

ElemType data;

struct QNode \*next;

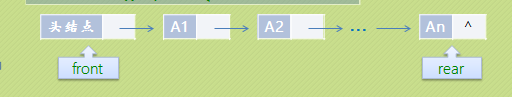
} QNode, \*QueuePrt;

typedef struct {

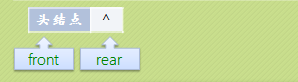
QueuePrt front, rear; // 队头、尾指针

} LinkQueue;

我们将队头指针指向链队列的头结点，而队尾指针指向终端结点。（注：头结点不是必要的，但为了方便操作，我们加上了。）



空队列时，front和rear都指向头结点。



## 创建一个队列

创建一个队列要完成两个任务：一是在内存中创建一个头结点，二是将队列的头指针和尾指针都指向这个生成的头结点，因为此时是空队列。

initQueue(LinkQueue \*q)

{

q->front=q->rear=(QueuePtr)malloc(sizeof(QNode));

if( !q->front )

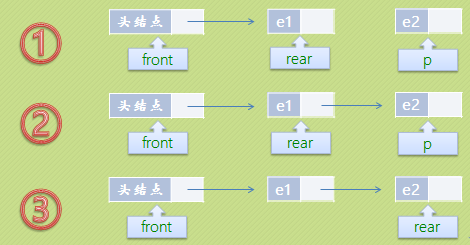
exit(0);

q->front->next = NULL;

}

## 入队列操作

入队列的操作过程如下：



InsertQueue(LinkQueue \*q, ElemType e)

{

QueuePtr p;

p = (QueuePtr)malloc(sizeof(QNode));

if( p == NULL )

exit(0);

p->data = e;

p->next = NULL;

q->rear->next = p;

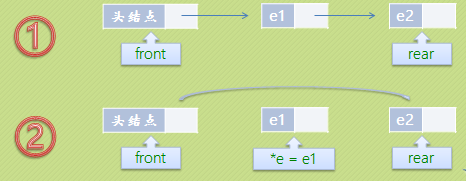
q->rear = p;

}

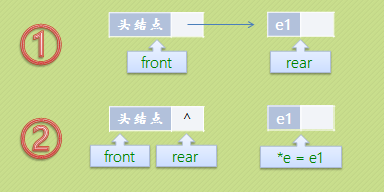
## 出队列操作

出队列操作是将队列中的第一个元素移出，队头指针不发生改变，改变头结点的next指针即可。

出队列的操作过程如下：



如果原队列只有一个元素，那么我们就应该处理一下队尾指针。



DeleteQueue(LinkQueue \*q, ELemType \*e)

{

QueuePtr p;

if( q->front == q->rear )

return;

p = q->front->next;

\*e = p->data;

q->front->next = p->next;

if( q->rear == p )

q->rear = q->front;

free(p);

}

## 销毁一个队列

由于链队列建立在内存的动态区，因此当一个队列不再有用时应当把它及时销毁掉，以免过多地占用内存空间。

DestroyQueue(LinkQueue \*q)

{

while( q->front ) {

q->rear = q->front->next;

free( q->front );

q->front = q->rear;

}

}

课后作业

* 编写一个链队列，任意输入一串字符，以#作为结束标志，然后将队列中的元素显示到屏幕上。
* 请自行完成后再参考小甲鱼提供的源代码。

# 30栈和队列8

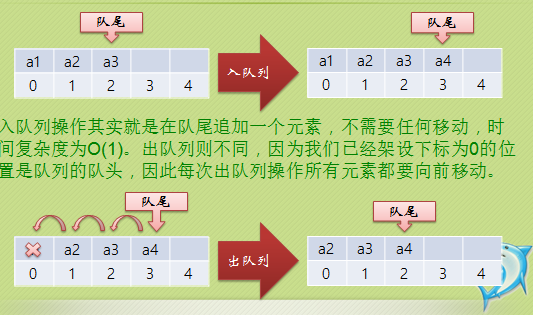
## 队列的顺序存储结构

为什么小甲鱼上节课说队列的实现上我们更愿意用链式存储结构来存储？

我们先按照应有的思路来考虑下如何构造队列的顺序存储结构，然后发掘都遇到了什么麻烦。

我们假设一个队列有n个元素，则顺序存储的队列需建立一个大于n的存储单元，并把队列的所有元素存储在数组的前n个单元，数组下标为0的一端则是队头。

No Pic You Say a J8



在现实中也是如此，一群人在排队买火车票，前边的人买好了离开，后面的人就要全部向前一步补上空位。

可是我们研究数据结构和算法的一个根本目的就是要想方设法提高我们的程序的效率，按刚才的方式，出队列的时间复杂度是O(n)，效率大打折扣！

如果我们不去限制队头一定要在下标为0的位置，那么出队列的操作就不需要移动全体元素。



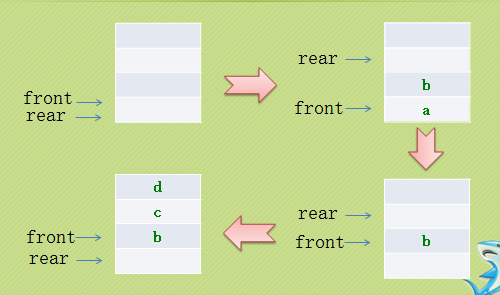
## 循环队列定义

我们再想想，要解决假溢出的办法就是如果后面满了，就再从头开始，也就是头尾相接的循环。

循环队列它的容量是固定的，并且它的队头和队尾指针都可以随着元素入出队列而发生改变，这样循环队列逻辑上就好像是一个环形存储空间。

但要注意的是，在实际的内存当中，不可能有真正的环形存储区，我们只是用顺序表模拟出来的逻辑上的循环。

我们通过一段动画片来加深印象吧！



于是我们发觉了，似乎循环队列的实现只需要灵活改变front和rear指针即可。

也就是让front或rear指针不断加1，即使超出了地址范围，也会自动从头开始。我们可以采取取余运算处理：

(rear+1) % QueueSize

(front+1) % QueueSize

取模就是取余数的意思，他取到的值永远不会大于除数，大家结合实例拿张纸算一算就知道啦~

代码清单

定义一个循环队列

#define MAXSIZE 100

typedef struct

{

ElemType \*base; // 用于存放内存分配基地址

// 这里你也可以用数组存放

int front;

int rear;

}

* 初始化一个循环队列
* initQueue(cycleQueue \*q)
* {
* q->base = (ElemType \*) malloc (MAXSIZE \* sizeof(ElemType));
* if( !q->base )
* exit(0);
* q->front = q->rear = 0;
* }
* 入队列操作
* InsertQueue(cycleQueue \*q, ElemType e)
* {
* if( (q->rear+1)%MAXSIZE == q->front )
* return; // 队列已满
* q->base[q->rear] = e;
* q->rear = (q->rear+1) % MAXSIZE;
* }
* 出队列操作
  + DeleteQueue(cycleQueue \*q, ElemType \*e)
  + {
    - if( q->front == q->rear )
      * return ; // 队列为空
    - \*e = q->base[q->front];
    - q->front = (q->front+1) % MAXSIZE;
  + }

人生如栈

人生，就像是栈的演变。

在父亲忙碌的入栈、出栈操作中，你，诞生了！

人生，仿佛是栈的重现。

每天你奔波于事业与家庭之间，做着似乎总是重复的事情，为的只是一餐温饱。

你总说，在哪里跌倒就在哪里爬起来，但是，你发现似乎总是在同一个地方跌倒无数次！

有一那么一次，你弹栈找不到返回地址，你忧郁了，迷茫了。。。。。。

很幸运，你看到了小甲鱼的视频^\_^

他会告诉你，在一个个栈的外边，其实隐藏着一个队列在调用你的每一个栈，只是你还年轻，没办法看得太清楚。

但是，你抬头仰望星空，在孤独中多思考，在彷徨中多回顾，就有希望，不断进取，就能成功！

人生，又是一个大队列的实现，春夏秋冬年复一年，改变的是时间，不变的是你对未来执着的信念！

地球近似于圆形，但并不完美。

选错了方向，你可能要多走一些路，

但，只要你肯坚持到底，你都可以到达终点！

