# Stack&Queue

相同点

栈和队列都属于线性结构，即除了首尾元素之外，其他元素都只有一个直接前趋和直接后继，用数学表示为：{a1,a2,a3,…,an}。

栈和队列通常采用顺序存储结构和链式存储结构

不同点

栈：先进后出，不清楚的可以用子弹夹来理解，先进去的子弹被压在下面，但是开枪的时候在上面的子弹先被发射；队列：先进先出，跟我们平常排队一样，先到的先服务。

栈只能在线性结构的一端进行操作 ，并且弹栈和压栈都在栈尾操作；但是队列在队头出队，在队尾入队。同时，在使用队列的时候，最好使用循环队列，因为在平常的队列中，进行入队和出队操作的时候，只是队列的前指针跟后指针（或理解成脚标）在移动，对于之前被移动走的元素不可以复用，到后面就会出现前指针紧挨后指针，这时候再要插入的话，就会抛出溢出的异常。

假溢出

由于队列拥有固定的长度，并且每当元素出队都是从front中移出，并且fornt的脚标向后移，这时我们看到前面的空位置理应可以插入元素，但是当我们执行插入操作的时候却被抛出异常，队列溢出，这种溢出被称为假溢出。

循环队列

但是可以通过循环队列实现：通过把队列的首尾连接起来，并且在插入元素的时候先通过判断当前尾脚标（rear）的位置，或者在出队的时候判断前脚标（front)的位置是否在（队列的长度-1）上，因为这里从0开始。如果要插入元素，如果前脚标在队列最后面的时候，即队列最大长度-1的位置因为从0开始，把前脚标front赋值为0，这样相当于把前脚标放到队头。如果元素出队的时候判断当前脚标（front）是否在队尾，即最大队列长度-1的位置，如果是，把前脚标对应的元素移出之后，再赋值为队头，即front=0，但是这种方法判断队列是否满就要再定义一个变量来记录当前队列有多少个元素，这样就需要每次进队/出队的时候都要进行自增（++/自减（–）操作。

另外一种大众的方法：取模。元素入队的时候，让尾脚标（rear）+1再跟队列的长度取余，得到的数就是要插入元素将存储的位置。在元素出队的时候，用前脚标（front）+当前队列元素个数再跟队列长度取余。在java的集合框架中，也有关于queue（jdk 1.5更新）跟stack的实现，其中queue是接口，但是stack是类。

**代码：  
int EnQueue(Queue q,ElemType e)**

**{**

**If((q.rear+1)%MAX\_SIZE==p.front) //队满**

**Return false;**

**q.data[q.rear]=e;**

**q.rear=(q.rear+1)%MAX\_SIZE;**

**return ok;**

**}**

**int Dequeue(Queue e,int&e)**

**{**

**if((q.front)%MAX\_SIZE==q.rear) //队空**

**return false;**

**q.data[q.front]=e;**

**q.front=(q.front+1)%MAX\_SIZE;**

**return ok;**

**}**

**两个重要的判断条件：**

**(q.rear+1)%MAX\_SIZE==p.front 循环队列满 rear指针在front前面一个slot**

**(q.front)%MAX\_SIZE==q.rear 循环队列空 rear指针和front重合**

**一般队列的判空与判满**

一般队列的判空

front == rear

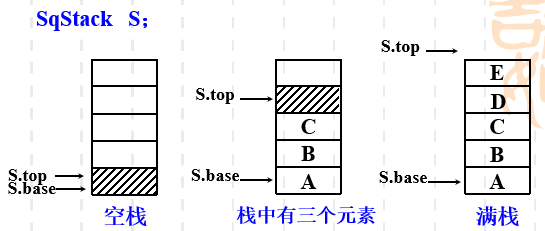
一般队列的判满

为 rear -front ==MAX\_QUEUE\_SIZE

判断栈不存在的条件为：S.base=NULL;

     空栈：S.base=S.top;

     满栈：S.top-S.base=S.stacksize;



这种情况是先赋值再移动指针 起初指针在0的位置，可以没有头节点

还有一种是先移动再赋值，删除的时候先删除再移动，这个是有头节点的 base起初再-1的位置

1. Status pop(SqStack &S) {
2. ElemType e;
3. if (isStackEmpty(S))
4. {
5. cout << "栈为空！" << endl;
6. return false;
7. }
8. return \*(--S.top);
9. }

Status push(SqStack &S, ElemType e) {

if (S.top - S.base >= S.stacksize) //栈满，追加栈的空间

{

S.base = (ElemType \*)realloc(S.base, (S.stacksize + STACKINCREMENT) \* sizeof(ElemType));

if (!S.base)

{

exit(OVERFLOW);

}

S.top = S.base + S.stacksize; //基地址+当前内存空间

S.stacksize += STACKINCREMENT;

}

\*S.top = e;

S.top++;

return true;

}