3.2.1

> 若希望循环队列中的元素都能得到利用，则需设置一个标志域 tag，并以 tag 的值为 0或 1来区分队头指针 $front$和队尾指针 $rear$相同时的队列状态是“空”还是“满”。试编写与此结构相应的入队和出队算法.

进队时置 tag为 1,出队时置 tag为 0,队列初始化时置为 0

- $rear$指向队尾元素的下一个位置

```
//参考循环队列1_3.cpp
void InitQueue(SqQueue &Q)//初始化
{
    Q.front=Q.rear=0;
    Q.tag=0;//表示此时无法删除(即初始化为空队列)
}

bool QueueEmpty(SqQueue Q)//判断队空
{
    if(Q.rear==Q.front&&Q.tag==0)
        return true;
    else return false;
}

bool QueueFull(SqQueue Q)//判断队满
{
    if(Q.rear==Q.front&&Q.tag==1)
        return true;
    else return false;
}

bool EnQueue(SqQueue &Q,ElemType x)//入队
{
    if(QueueFull(Q)==true)
        return false;
    Q.data[Q.rear]=x;
    Q.rear=(Q.rear+1)%MaxSize;

    Q.tag=1;//入队操作更新为1
    return true;
}

bool DeQueue(SqQueue &Q,ElemType &x)//出队
{
    if(QueueEmpty(Q)==true)
        return false;

    x=Q.data[Q.front];
    Q.front=(Q.front+1)%MaxSize;

    Q.tag=0;//出队操作更新为0
```

```
    return true;
}

```

- $rear$指向队尾元素

```
//参考循环队列2_3.cpp
void InitQueue(SqQueue &Q)//初始化
{
    Q.front=0;
    Q.rear=MaxSize-1;
    Q.tag=0;//表示此时无法删除(即初始化为空队列)
}

bool QueueEmpty(SqQueue Q)//判断队空
{
    if((Q.rear+1)%MaxSize==Q.front&&Q.tag==0)
        return true;
    else return false;
}

bool QueueFull(SqQueue Q)//判断队满
{
    if((Q.rear+1)%MaxSize==Q.front&&Q.tag==1)
        return true;
    else return false;
}

bool EnQueue(SqQueue &Q,ElemType x)//入队
{
    if(QueueFull(Q)==true)
        return false;
    Q.rear=(Q.rear+1)%MaxSize;//注意入队先+1,再赋值
    Q.data[Q.rear]=x;


    Q.tag=1;//入队操作更新为1
    return true;
}

bool DeQueue(SqQueue &Q,ElemType &x)//出队
{
    if(QueueEmpty(Q)==true)
        return false;

    x=Q.data[Q.front];
    Q.front=(Q.front+1)%MaxSize;

    Q.tag=0;//出队操作更新为0
    return true;
}
```

3.2.2

> Q是一个队列， S是一个空栈，实现将队列中的元素逆置的算法。

让队列中的元素逐个地出队列,入栈;全部入栈后再逐个出栈,入队列.

```
void Reverse_Queue(SqQueue &Q,SqStack &S)
{
    ElemType num;
    while(QueueEmpty(Q)==false)
    {
        DeQueue(Q, num);//出队列
        Push(S, num);//入栈
    }

    while(StackEmpty(S)==false)
    {
        Pop(S, num);//出栈
        EnQueue(Q, num);//入队列
    }
}
```

3.2.3

> 利用两个栈 S_1, S_2来模拟一个队列，已知栈的 4个运算定义如下:

```
Push(S,x);//元素x入栈S
Pop(S,x);//S出栈并将出栈的值赋给x
StackEmpty(S);//判断栈是否为空
stackOverflow(S);//判断栈是否满
```

> 如何利用栈的运算来实现该队列的 3个运算（形参由读者根据要求自己设计）？

```
EnQueue;//将元素x入队
DeQueue;//出队,并将出队元素存储在x中
QueueEmpty;//判读队列是否为空
```

- 入队操作

两个栈 S_1， S_2都为空，执行入队操作，将元素直接插入 S_1中。

栈 S_1为满，栈 S_2不为空，则队列为满，无法执行入队操作。

栈 S_1为满，栈 S_2为空，执行入队操作，先将栈 S_1中的元素逐一出栈，再逐一入栈 S_2,最后当 S_1为空时，将元素插入 S_1中，实现入队操作。

- 出队操作

栈 S_2不为空，则队首元素位于 S_2的栈顶（栈 S_1为满， S_2不为空），执行出队操作，将 S_2中的栈顶元素出栈，实现出队操作。

栈 S_2为空，栈 S_1为空，则队列为空，无法实现出队操作。

栈 S_1不空，栈 S_2为空，执行出队操作时，先将 S_1中元素逐一从 S_1出栈，再逐一入栈 S_2，最后将 S_2中的栈顶元素出栈，实现出队操作。

- 判空操作

两个栈 S_1和 S_2都为空时，队列为空。

```
bool EnQueue(SqStack &S1,SqStack &S2,ElemType x)//入队
{
    if(StackOverflow(S1)==false)//S1未满,放入S1中
    {
        Push(S1,x);
        return true;
    }
    if(StackOverflow(S1)==true&&StackEmpty(S2)==false)
    {
        //S1满了,且S2有元素
        cout << "队列满" << endl;
        return 0;
    }
    if(StackOverflow(S1)==true&&StackEmpty(S2)==true)
    {
        //S1满了,S2空
        ElemType num;
        while(StackEmpty(S1)==false)
        {
            Pop(S1,num);
            Push(S2,num);
        }
    }

    Push(S1,x);
    return true;
}

bool DeQueue(SqStack &S1,SqStack &S2,ElemType &x)//出队列
{
    if(StackEmpty(S2)==false)//S2不空
    {
        Pop(S2,x);
        return true;
    }
    else if(StackEmpty(S1)==true)//S2为空,S1为空
    {
        cout << "队列空" << endl;
        return 0;
    }
    else//S2为空,S1不为空
    {
        ElemType num;
        while(StackEmpty(S1)==false)
        {
```

```
            Pop(S1, num);
            Push(S2,num);
        }
        Pop(S2,x);
        return true;
    }
}

bool QueueEmpty(SqStack S1,SqStack S2)//判断队空
{
    if(StackEmpty(S1)==true&&StackEmpty(S2)==true)
        return true;
    else return false;
}
```

3.2.4

> **2019统考真题**：请设计一个队列，要求满足:
>
> ①初始时队列为空;
>
> ②入队时，允许增加队列占用空间;
>
> ③出队后，出队元素所占用的空间可重复使用，即整个队列所占用的空间只增不减;
>
> ④入队操作和出队操作的时间复杂度始终保持为 $O(1)$。
>
> 请回答下列问题:
>
> 1)该队列是应选择链式存储结构,还是应选择顺序存储结构?
>
> 2)画出队列的初始状态,并给出判断队空和队满的条件。
>
> 3)画出第一个元素入队后的队列状态。
>
> 4)给出入队操作和出队操作的基本过程。

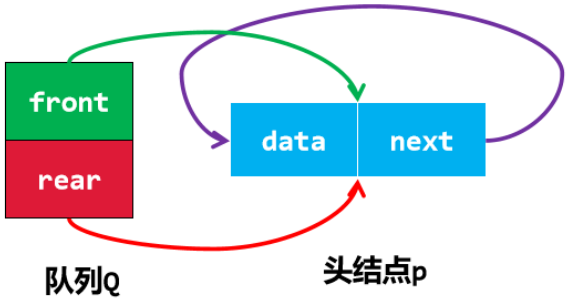
1)该队列是应选择链式存储结构,还是应选择顺序存储结构?

- 对于要求①，无论是链式存储还是顺序存储很容易满足，因为初始化队列都是空的;
- 对于要求②，可以使用**链式存储**，每次入队新元素时便创建新结点并分配空间，这样也增加了队列空间。而顺序存储空间是一开始就分配好的，队列的占用空间无法随着入队操作而增加，所以需要选择链式存储结构。
- 对于要求③，要求出队元素的空间可以重复使用，即即使链队结点出队后，也不要释放空间，以前的链表删除操作都会释放掉被删结点的空间。我们可以这样操作：对出队后的结点不真正释放（即不调用 `free` 函数释放空间），而是用队头指针指向新的队头结点，原队头结点仍然保留在链式队列中，但队头指针却不指向它了；而有新元素入队时，有空余结点则无需创建新结点开辟新空间，直接赋值到队尾后的第一个空结点即可，然后用队尾指针指向新的队尾结点。这就要求设计成一个首尾相接的循环单链表，即链式循环队列。
- 对于要求④，由于有队头指针和队尾指针，所以链式循环队列的入队操作和出队操作的时间复杂度都是 $O(1)$。

2)画出队列的初始状态,并给出判断队空和队满的条件。

该链式循环队列的实现可以参考**顺序循环队列**，不同之处在于链式循环队列可以动态地增加空间，出队的结点也可以循环利用，入队时空间不够也可以动态增加。同样，链式循环队列，也需要区分队满和队空的情况，这里参考顺序循环队列牺牲一个单元来判断，即在链式循环队列中有一个结点是不存储任何数据的，仅用来判断链式循环队列是队满还是队空。初始化时，创建一个只有空闲结点（初始时就必须有这个空闲结点，用来判断队空和队满的空闲结点）循环单链表，队头指针 $front$ 和队尾指针 $rear$ 均指向空闲结点。

初始化



- p->next=p;
- Q->front=Q->rear=p;

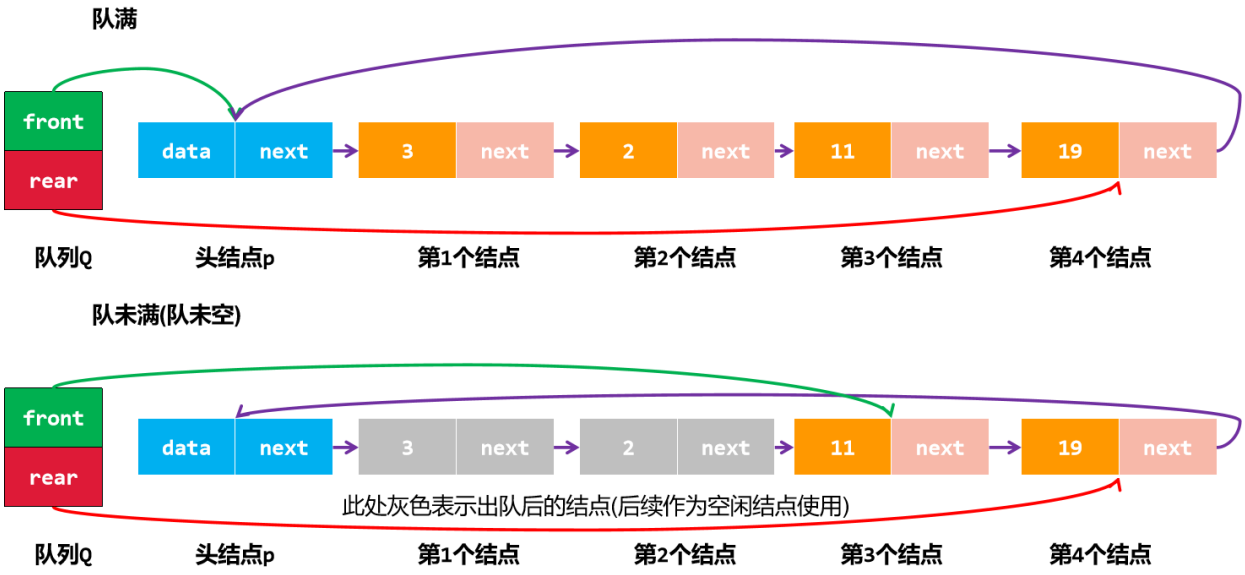
- **队空**: `Q->front==Q->rear`
- **队满**: `Q->front=Q->rear->next`

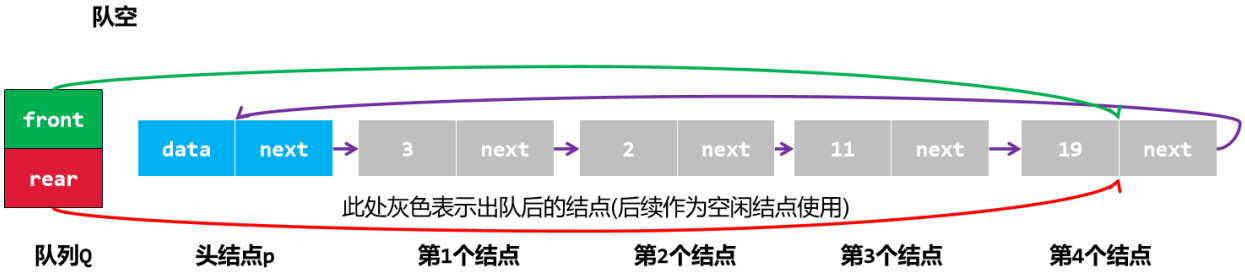
注:

1.初始时，无论是队空的判定还是队满的判定条件都能通过，所以初始时既是队空也是队满。

2.链式循环队列**不存在队满**的。由于在出队后，结点空间不会释放掉，所以会一直存在，那么队满的判定条件就会返回 0，表示可以直接利用空余结点来存储新元素。如果再入队，那么就会把空余结点也给用完，那时候就存在队满了，即没有空余结点就是队满状态。如果只是一直入队那么一直都是队满状态。

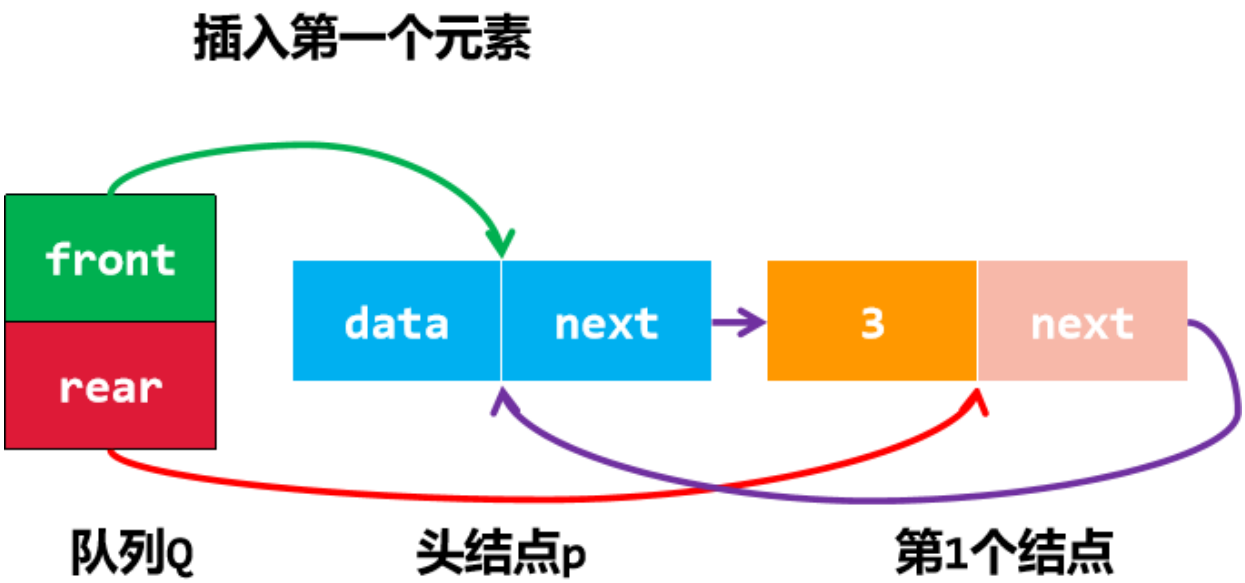
下面是**队满**、**队未满(队未空)**、**队空**的图示:





3)画出第一个元素入队后的队列状态。

插入第一个元素后的状态如图所示：



4)给出入队操作和出队操作的基本过程。

- **入队操作**

当队满情况下,即创建新节点并插入；当队未满情况下,直接使用已有的空闲结点空间

```
bool EnQueue(LinkQueue &Q,ElemType x)//入队
{
    if(QueueFull(Q)==true)//队满
    {
        LinkNode *p=(LinkNode *)malloc(sizeof(LinkNode));

        if(p==NULL)
            return false;

        p->data=x;
        p->next=NULL;

        p->next=Q.rear->next;//构成循环队列
        Q.rear->next=p;
        Q.rear=p;
    }
    else//队未满时利用空闲结点
    {
        Q.rear=Q.rear->next;
```

```
        Q.rear->data=x;
    }
    return true;
}
```

- **出队操作**

当队不为空情况下,将 $front$后移

```
bool DeQueue(LinkQueue &Q,ElemType &x)//出队
{
    if(QueueEmpty(Q)==true)//空队列
        return false;


    Q.front=Q.front->next;//后移
    x=Q.front->data;//由于带头结点(此处获取时要先后移)
    return true;
}
```