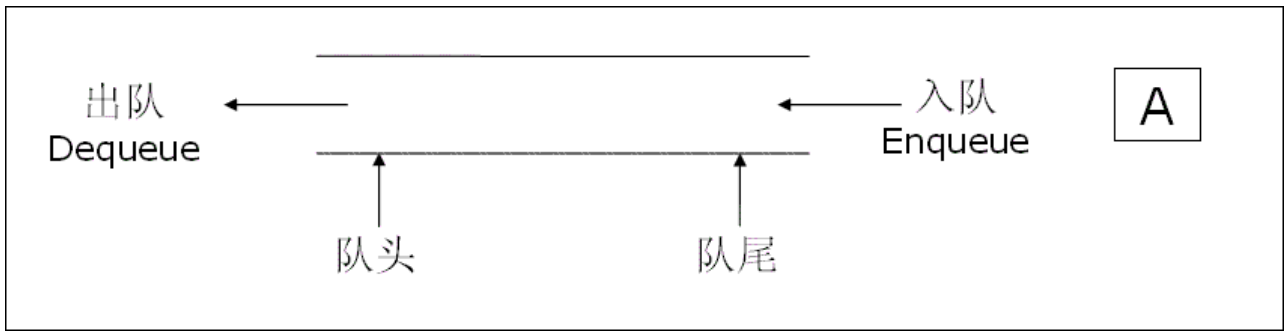
队列的表示和实现

队列的概念及结构

> 队列：只允许在一端进行插入数据操作，在另一端进行删除数据操作的特殊线性表，队列具有先进先出FIFO(First In First Out) 入队列：进行插入操作的一端称为队尾 出队列：进行删除操作的一端称为队头



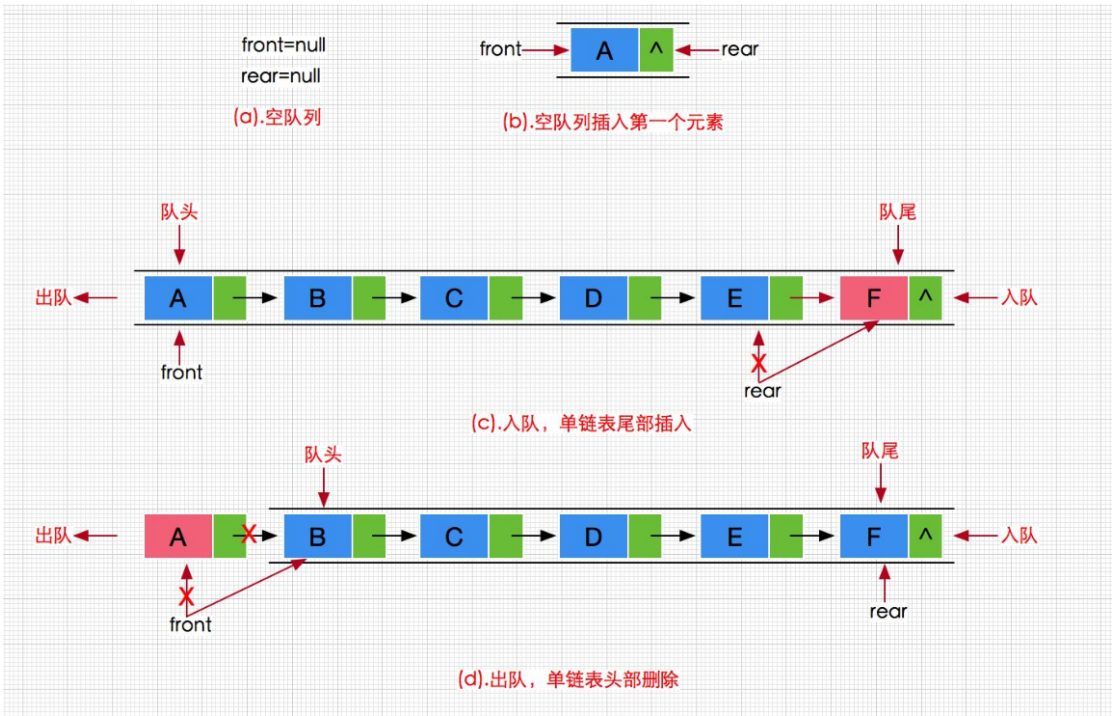
队列的实现

> 队列也可以数组和链表的结构实现，使用链表的结构实现更优一些，因为如果使用数组的结构，出队列在数组头上出数据，效率会比较低。

数组实现队列

1、头作队头，尾作队尾 头插入队尾删出队，头插为O(N)，尾删为O(1)，不适合

2、头做队尾，尾作队头 尾插入队头删出队，尾插为O(1)，头删为O(N)，不适合



队尾入队，队头出队

```
1  // 链式结构：表示队列
```

```
typedef struct QListNode
{
 struct QListNode* _pNext;
 QDataType _data;
}QNode;
// 队列的结构
typedef struct Queue
{
 QNode* _front;
 QNode* _rear;
}Queue;
// 初始化队列
void QueueInit(Queue* q);
// 队尾入队列
void QueuePush(Queue* q, QDataType data);
// 队头出队列
void QueuePop(Queue* q);
// 获取队列头部元素
QDataType QueueFront(Queue* q);
// 获取队列队尾元素
QDataType QueueBack(Queue* q);
// 获取队列中有效元素个数
int QueueSize(Queue* q);
// 检测队列是否为空，如果为空返回非零结果，如果非空返回0
int QueueEmpty(Queue* q);
// 销毁队列
void QueueDestroy(Queue* q);
```

初始化队列

```
void QueueInit(Queue* pq)
{
    assert(pq);
    pq->head = NULL;
    pq->tail = NULL;
}
```

销毁队列

队列元素被清空后，要考虑避免野指针的问题

```
void QueueDestroy(Queue* pq)//销毁
{
    assert(pq);
    QueueNode* cur = pq->head;
    while (cur != NULL)
    {
        QueueNode* next = cur->next;
        free(cur);
        cur = next;
    }
    pq->head = pq->tail = NULL;
}
```

判空

```
bool QueueEmpty(Queue* pq)//判空
{
    assert(pq);
    return pq->head == NULL;
}
```

入队

如果队列为空，头和尾都指向新结点

```
void QueuePush(Queue* pq, QDataType x)//进队
{
    assert(pq);
    QueueNode* newnode = (QueueNode*)malloc(sizeof(QueueNode));
    if (newnode == NULL)
    {
        printf("Realloc fail!\n");
        exit(-1);
    }
    newnode->data = x;
```

```
    newnode->next = NULL;
    if (pq->head == NULL)
    {
        pq->head = pq->tail = newnode;
    }
    else
    {
        pq->tail->next = newnode;
        pq->tail = newnode;
    }
}
```

出队

```
void QueuePop(Queue* pq)//出队
{
    assert(pq);
    /*if (pq->head)
        return;*/
    assert(!QueueEmpty(pq));//队空就不能继续删除了
    QueueNode* next = pq->head->next;
    free(pq->head);
    pq->head = next;
    if (pq->head==NULL)//空间被释放，但指针还在，避免出现野指针
    {
        pq->tail = NULL;
    }
}
```

取队头

```
QDataType QueueFront(Queue* pq)//取队头
{
    assert(!QueueEmpty(pq));
    assert(pq);
    return pq->head->data;
}
```

取队尾

```
QDataType QueueBack(Queue* pq)//取队尾
{
    assert(!QueueEmpty(pq));
    assert(pq);
    return pq->tail->data;
}
```

求队长

```
int QueueSize(Queue* pq)//求队长
{
    assert(pq);
    int n = 0;
    QueueNode* cur = pq->head;
    while (cur)
    {
        n++;
        cur = cur->next;
    }
    return n;
}
```

OJ题

用队列实现栈

1.入数据：往不为空的队列入，保持另一个队列为空
2.出数据：依此出队头的数据，转移另一个队列保存，只剩最后一个时，pop掉

```
typedef struct
{
    Queue q1;
    Queue q2;
}MyStack;

```

```c
MyStack* myStackCreate()
{
    MyStack* st =(MyStack*) malloc(sizeof(MyStack));
    if (st == NULL)
    {
        printf("Malloc fail!\n");
    }
    QueueInit(&(st->q1));
    QueueInit(&st->q2);
    return st;
}

void myStackPush(MyStack* obj, int x)
{
    if (!QueueEmpty(&obj->q1))
        QueuePush(&obj->q1, x);
    else
        QueuePush(&obj->q2, x);
}
int myStackPop(MyStack* obj)
{
    Queue* emptyQ = &obj->q1;
    Queue* nonemptyQ = &obj->q2;
    //默认q1为空，q2不为空
    if (!QueueEmpty(&obj->q1))
    {
        emptyQ = &obj->q2;
        nonemptyQ = &obj->q1;
    }
    while (QueueSize(nonemptyQ) > 1)//只剩最后一个元素时，删掉
    {
        //取出放到空队列，并把非空那边删掉
        QueuePush(emptyQ, QueueFront(nonemptyQ));
        QueuePop(nonemptyQ);
    }
        int top = QueueFront(nonemptyQ);
        QueuePop(nonemptyQ);
        return top;
}
```

```
46
47 int myStackTop(MyStack* obj)
48 {
49     if (!QueueEmpty(&obj->q1))
50         return QueueBack(&obj->q1);
51     else
52         return QueueBack(&obj->q2);
53 }
54
55 bool myStackEmpty(MyStack* obj)
56 {
57     return QueueEmpty(&obj->q1) && QueueEmpty(&obj->q2);
58 }
59
60 void myStakFree(MyStack* obj)
61 {
62     QueueDestroy(&obj->q1);
63     QueueDestroy(&obj->q2);
64     free(obj);
65 }
```

用栈实现队列

```
1  typedef struct
2  {
3      ST pushST;
4      ST popST;
5  }MyQueue;
6
7  MyQueue* myQueueCreate()
8  {
9      MyQueue* q = (MyQueue*)malloc(sizeof(MyQueue));
10     StackInit(&q->pushST);
11     StackInit(&q->popST);
12     return q;
```

```c
}

void myQueuePush(MyQueue* obj, int x)
{
    StackPush(&obj->pushST, x);
}

int myQueuePop(MyQueue* obj)
{
    if (StackEmpty(&obj->popST))//pop栈没有数据，就要从push栈里倒过来
    {
        while (!StackEmpty(&obj->pushST))
        {
            StackPush(&obj->popST, StackTop(&obj->pushST));
            StackPop(&obj->pushST);
        }
    }
    int front = StackTop(&obj->popST);
    StackPop(&obj->popST);
    return front;
}

int myQueuePeek(MyQueue* obj)
{
    if (StackEmpty(&obj->popST))//pop栈没有数据，就要从push栈里倒过来
    {
        while (!StackEmpty(&obj->pushST))
        {
            StackPush(&obj->popST, StackTop(&obj->pushST));
            StackPop(&obj->pushST);
        }
    }
    return StackTop(&obj->popST);
}

bool myQueueEmpty(MyQueue* obj)
{
    return StackEmpty(&obj->popST) && StackEmpty(&obj->pushST);
}

```

```c
void myQueueFree(MyQueue* obj)
{
    QueueDestroy(&obj->popST);
    QueueDestroy(&obj->pushST);
    free(obj);
}
```

循环队列

重点：循环队列，无论使用数组还是链表实现，都要多开一个空间，也就意味着，要是一个存k个数据的循环队列，要开k+1个空间，否则无法实现判空和判满。

数组实现：

链表实现：

```c
typedef struct
{
    int* a;//数组结构实现
    int front;
    int tail;
    int k;
}MyCircularQueue;

MyCircularQueue* myCircularQueueCreate(int k)
{
    MyCircularQueue* cq = (MyCircularQueue*)malloc(sizeof(MyCircularQueue));
    cq->a = (int*)malloc(sizeof(int) * (k + 1));
    cq->front = cq->tail = 0;
    cq->k = k;
    return cq;
}

//入队
bool myCircularQueueEnQueue(MyCircularQueue * obj, int value)
```

```
{
    if (myCircularQueueIsFUll(obj))
        return false;
    obj->a[obj->tail] = value;
    ++obj->tail;
    obj->tail %= (obj->k + 1);
    return true;
}

//出队
bool myCircularQueueDeQueue(MyCircularQueue* obj, int value)
{
    if (myCircularQueueIsEmpty(obj))
        return false;
    ++obj->front;
    obj->front %= (obj->k + 1);
    return true;
}

//取队头
int myCircularQueueFront(MyCircularQueue* obj)
{
    if (myCircularQueueIsEmpty(obj))
        return -1;
    return obj->a[obj->front];
}

//取队尾
int myCircularQueueRear(MyCircularQueue* obj)
{
    if (myCircularQueueIsEmpty(obj))
        return -1;
    if (obj->tail == 0)
    {
        return obj->a[obj->k];
    }
    else
    {
        return obj->a[obj->tail - 1];
    }
```

```c
    /*int i = (obj->tail + obj->k) % (obj->k + 1);
    return obj->a[i];*/

}


bool myCircularQueueIsEmpty(MyCircularQueue* obj)//判空
{
    return obj->front == obj->tail;
}

bool myCircularQueueIsFUll(MyCircularQueue* obj)//判满
{
    return (obj->tail + 1) % (obj->k + 1) == obj->front;
}
void myCircularQueueIsFree(MyCircularQueue* obj)
{
    free(obj->a);
    free(obj);
}
```