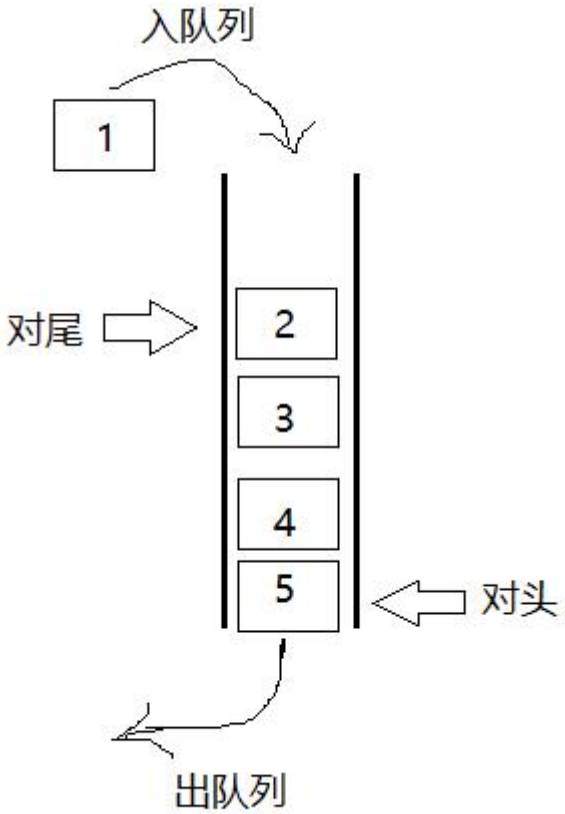
队列(FIFO)

概论

基本概念

只在线性表的一端进行添加元素，在线性表的另一端进行删除元素的数据结构称为队列。这种规则称为**FIFO（first input first output）**原则。 进入队列的一端称为“队尾”，出队列的一端称为“队头”。



应用场景

根据队列的存储特性，只需要使用数据结构实现一种顺序执行 / 访问的场景可以使用队列。

分类

根据实现方式可以分为：**链式队列** 和 **顺序队列。**顺序队列的一种非常常见的优化结构称为 **循环队列。**

方法

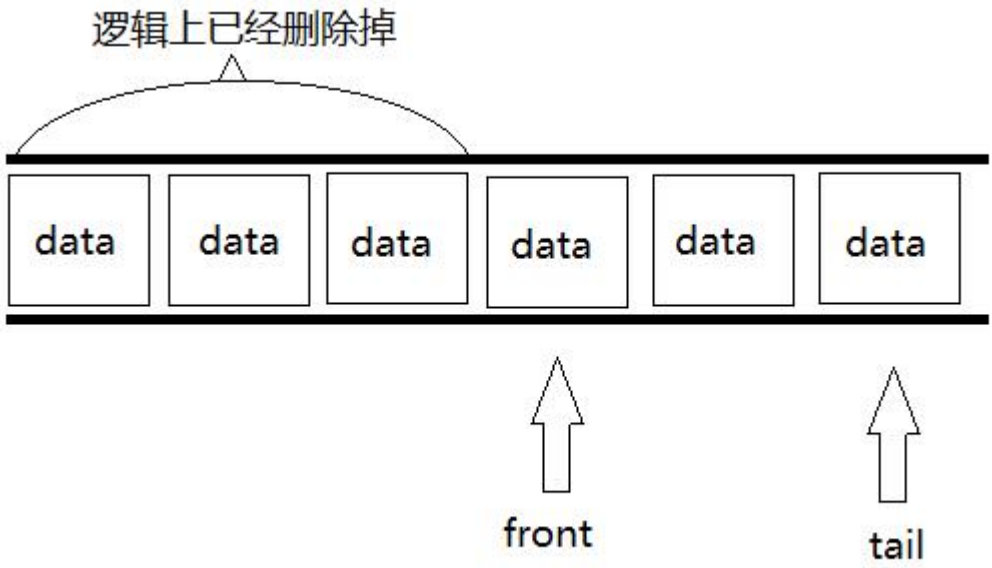
> 实现循环队列

普通的顺序队列和循环队列比较

普通的顺序队列只能往顺序表的尾部添加元素，只能从顺序表的开头删除元素。而删除元素就是让队头下标+1.

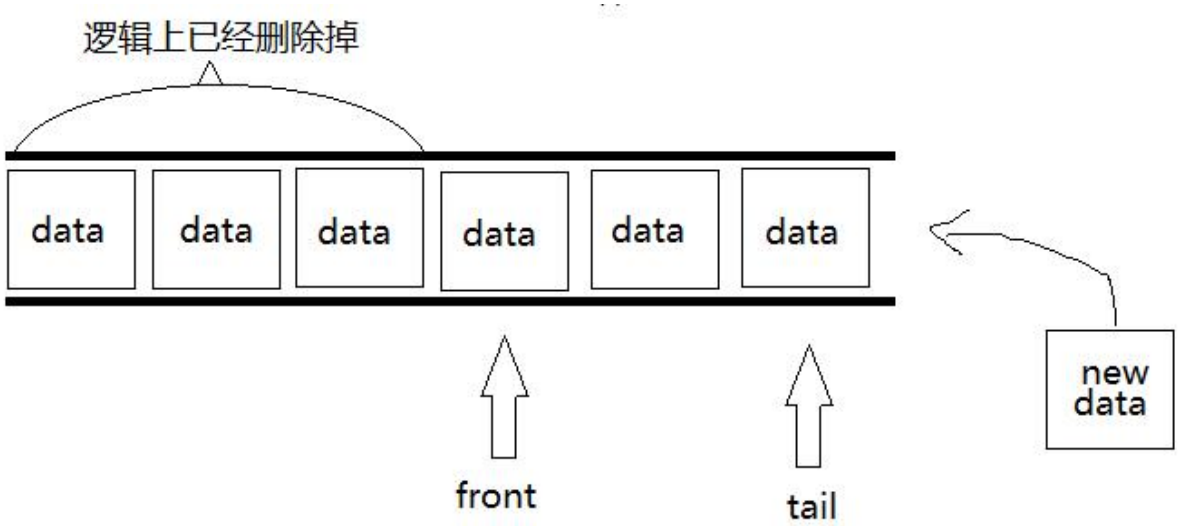
这样就造成了一个结果：队列已经满了，但是可用元素却没有把队列所有空间利用完。

就像这样：

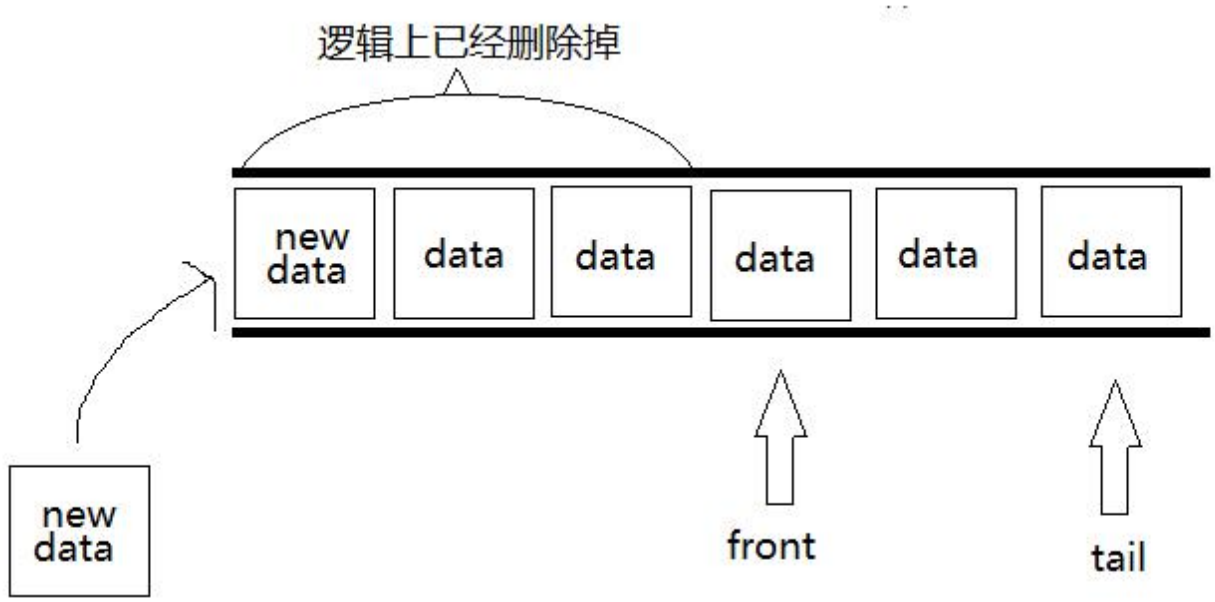


逻辑上已经删除的元素还依然占用着队列底层的空间，这就产生了大大的浪费。为了解决这个问题，就出现了 **循环队列。**

当队列已经“满”了的时候，需要添加新元素，如果是普通的顺序队列，逻辑上就是这样子的：



循环队列在面对这个问题的时候，这样子优化，就起到了充分利用空间的作用：



当队列已经"满"了的时候，它将新元素从队首开始添加，覆盖掉已经删除掉的元素，就起到了一个重复利用队列底层空间的效果。

定义节点

节点元素构成：

- 顺序表（即指定大小的数组）
- 对头下标
- 对尾下标

```
#define MAXSIZE 10
typedef int DataType;
typedef struct Node{
    DataType _data[MAXSIZE];
    int _front;
    int _tail;
} Queue;
```

初始化

初始化队列的时候将对头和对位的下标都置为0。

```
void InitQueue(Queue* queue){
    assert(queue);
    queue->_front = queue->_tail = 0;
}
```

入队

```
void inQueue(DataType* queue, DataType elem){
    if ((queue->_tail + 1) % MAXSIZE == queue->_front) {
        printf("空间已满");
        return;
    }
    queue->_data[queue->_tail % MAXSIZE] = elem;
    queue->_tail++;
}
```

出队

```
void outQueue(DataType* queue){
    if(queue->_front == queue->_tail){
        return;
    }
    queue->_front = (queue->_front + 1) % MAXSIZE;
}
```

获取对头

直接返回对头元素即可。

```
DataType FrontQueue(DataType* queue){
    return queue->_data[queue->_front];
}
```

> 实现链式队列

定义节点

首先需要定义队列的节点 `QNode`，看看它的构成：

- 数值域：需要存储的数据
- 指针域：下一个节点的地址

然后我们还需要管理队列的对头指针和对尾指针，所以还需要再封装一个数据类型 `Queue`，它的构成也很简单：

- 对头指针
- 对尾指针

```
typedef int DataType;
//def queue node
typedef struct QNode{
    DataType _elem;
    struct QNode* _next;
} QNode;

//def queue type
typedef struct Queue{
    QNode* _front;
    QNode* _tail;
} Queue;
```

初始化

队列为空，将对头指针和对尾指针都指向NULL，避免以后出现野指针。

```
void InitQueue(Queue* q){
    q->_tail = q->_front = NULL;
}
```

入队

入队逻辑：

- 如果对头指针为NULL，说明当前队列为空，则给对头指针分配空间并赋值即可，

  由于队列当前只有一个元素，所以让对尾指针和对头指针都指向这块新分配的空间
- 如果头指针不为NULL，由于对尾指针指向队尾元素，所以要给对尾指针的指针域分配空间

  然后赋值并且让对尾指针指向新的对尾元素

```
void inQueue(Queue* queue, DataType elem){
    if(NULL == queue->_front){
        queue->_front = (QNode*)malloc(sizeof(QNode));
        queue->_front->_elem = elem;
        queue->_front->_next = NULL;
        queue->_tail = queue->_front;
    } else {
        queue->_tail->_next = (QNode*)malloc(sizeof(QNode));
        queue->_tail->_next->_elem = elem;
        queue->_tail->_next->_next = NULL;
        queue->_tail = queue->_tail->_next;
    }
}
```

出队

出队逻辑：

- 判断当前队列是否为空，如果是，直接返回
- 否则，定义一个指针指向当前的对头元素，更新对头指针指向旧的对头的下一个节点
- 删除掉旧的对头元素

```
void outQueue(Queue* queue){
    if(NULL == queue->_front){
        return NULL;
    }
    QNode* tmp = queue->_front;
    queue->_front = queue->_front->_next;
    free(tmp);
}
```

获取对头

返回对头指针即可。

```
QNode* FrontQueue(Queue* queue){
    return queue->_front;
}
```