3 Queue（队列）

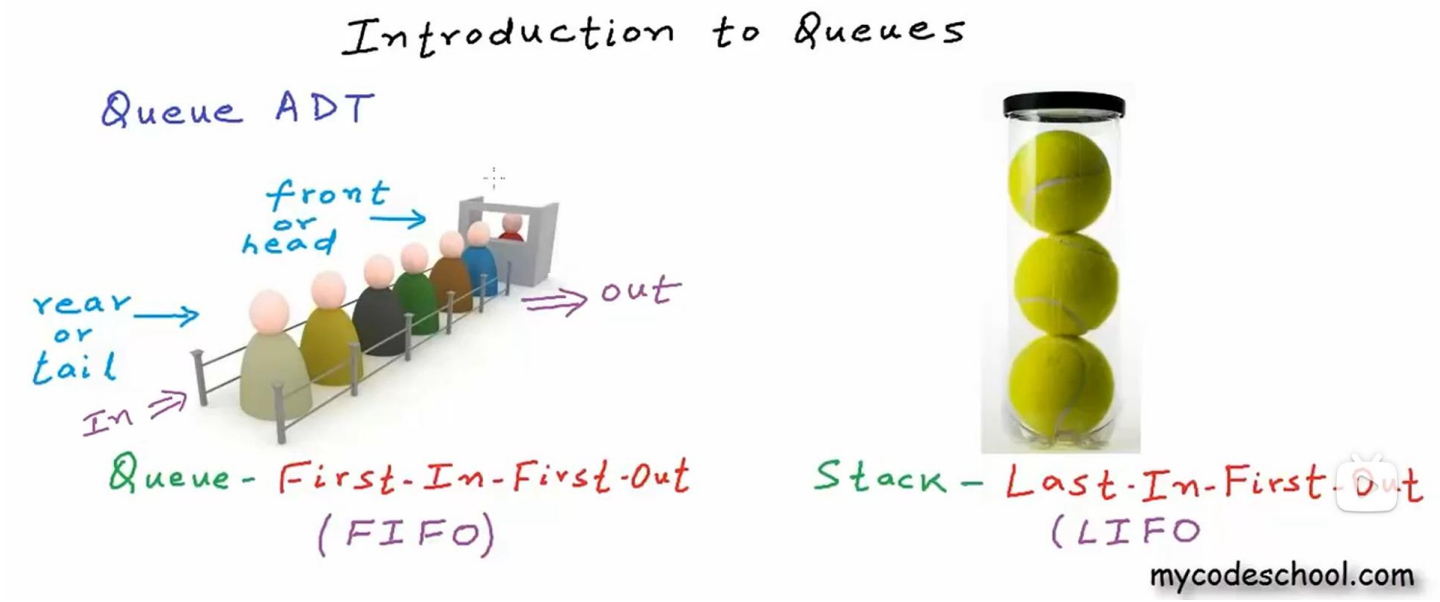
3.1 队列：基本介绍

3.2 使用数组实现一个队列

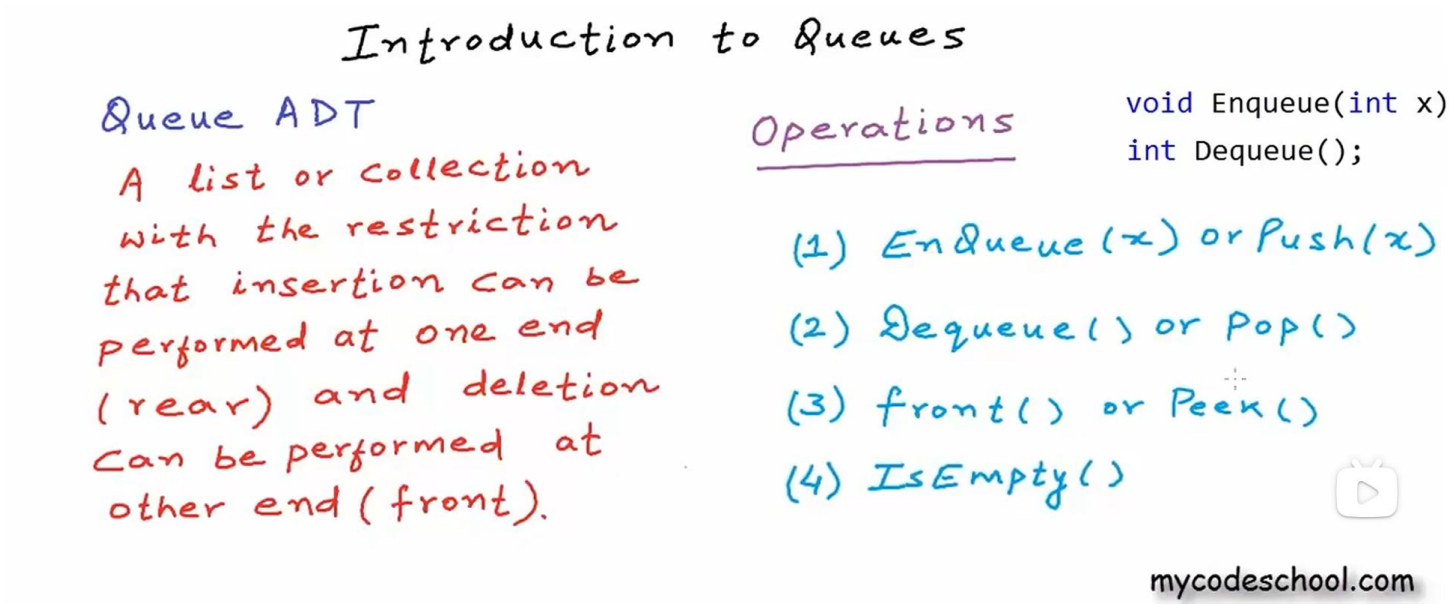
3.3 使用链表实现一个队列

总结：

3.1 队列：基本介绍



栈是这样一种集合，它的插入和删除都是从同一端进行，我们称之为**栈顶**。
但是在**队列**里面，插入必须从**后部**或者说**队尾**进行，而删除必须从另一端进行，我们称之为**前部**或者**队头**。



Push 和 Pop 这两个名字对于栈来讲更加知名一些，Enqueue 和 Dequeue 这两个名字对于队列来讲更加知名一些。在具体实现的时候，我们可以为接口选择合十的名字。

一般来讲，出队操作同时还会返回队列头部那个即将被删除的元素。
一般来讲，还会保留第三个操作，就是 front 或者 peek，用来查看队列头部的元素。这个操作应该仅仅是返回队列头部的元素而不是删除它。当然，我们还可以有其它操作。

总结：

Introduction to Queues

Queue ADT

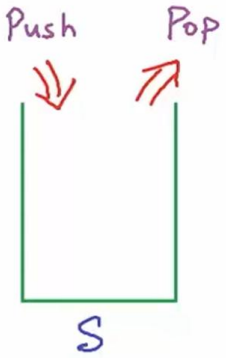
Operations

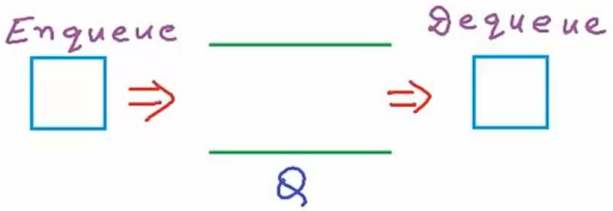
(1) EnQueue(x)
(2) Dequeue()
(3) front()
(4) IsEmpty()

Constant time or O(1)

Introduction to Queues

Queue ADT



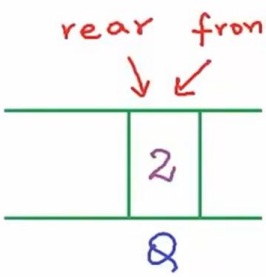


Operations

(1) EnQueue(x)
(2) Dequeue()
(3) front()
(4) IsEmpty()

Introduction to Queues

Queue ADT

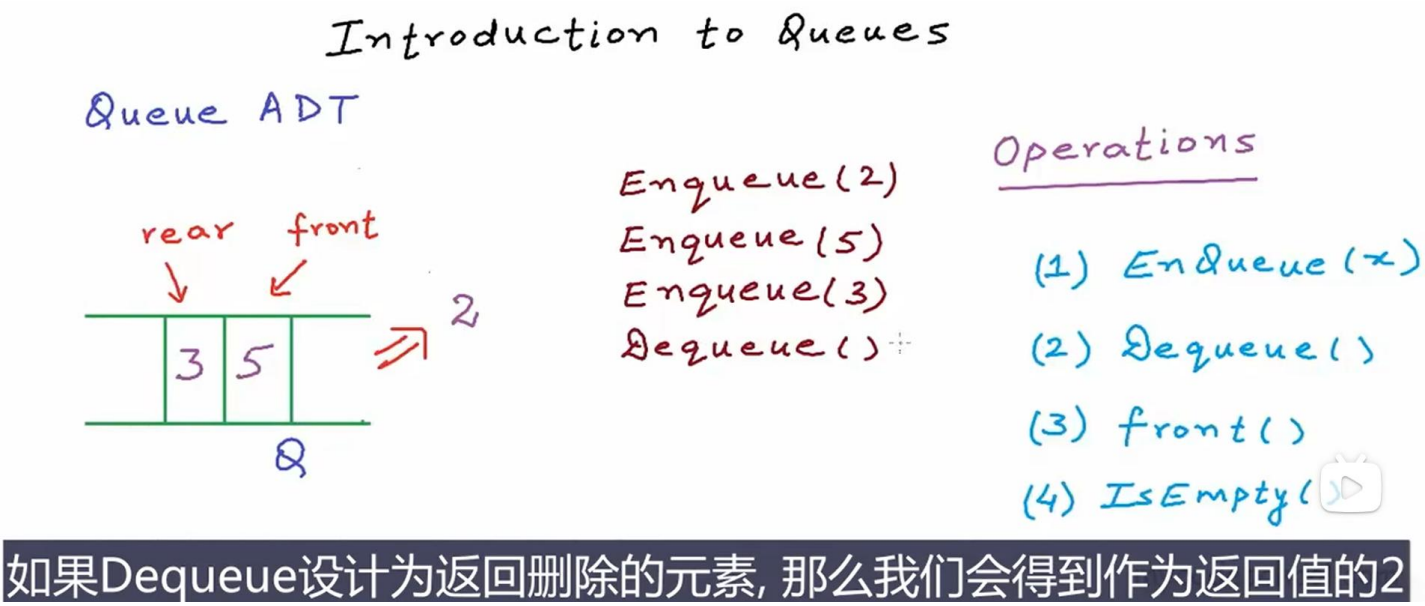


Enqueue(2)

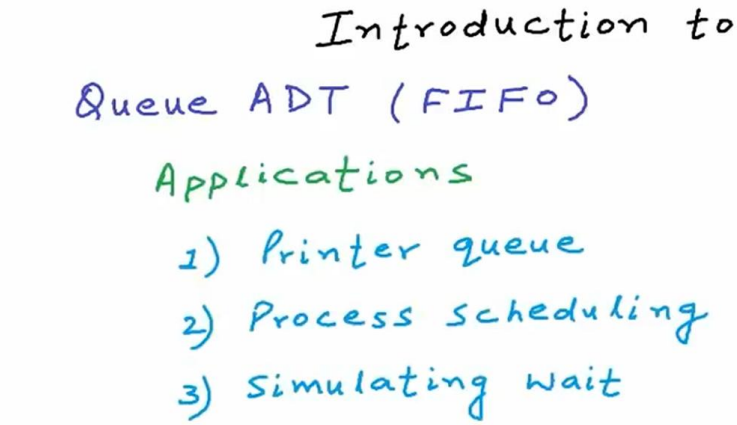
Operations

(1) EnQueue(x)
(2) Dequeue()
(3) front()
(4) IsEmpty()

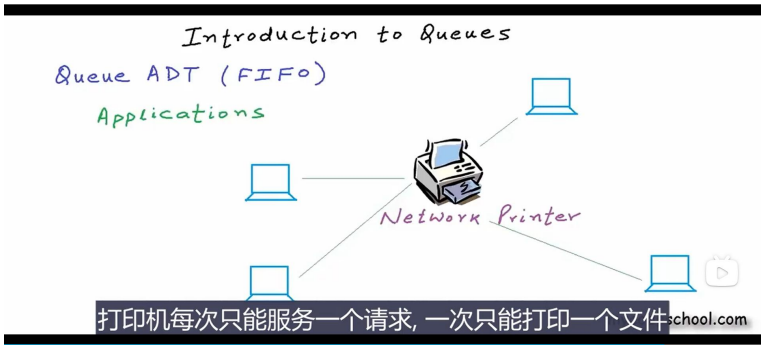
总结:

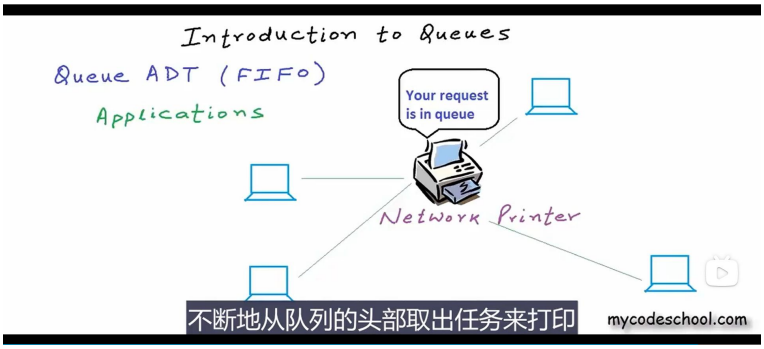


如果Dequeue设计为返回删除的元素，那么我们会得到作为返回值的2



总的来说，队列可以被用来模拟一系列需要等待的场景school.com





真实的场景中什么时候会使用队列呢？一个典型的场景，为了服务请求我们有一些共享的资源，但是这里的资源一次只能处理一个请求，每次只能服务一个请求，在这种情景下，比较合理的办法就是让请求排队，先来的请求先得到服务。

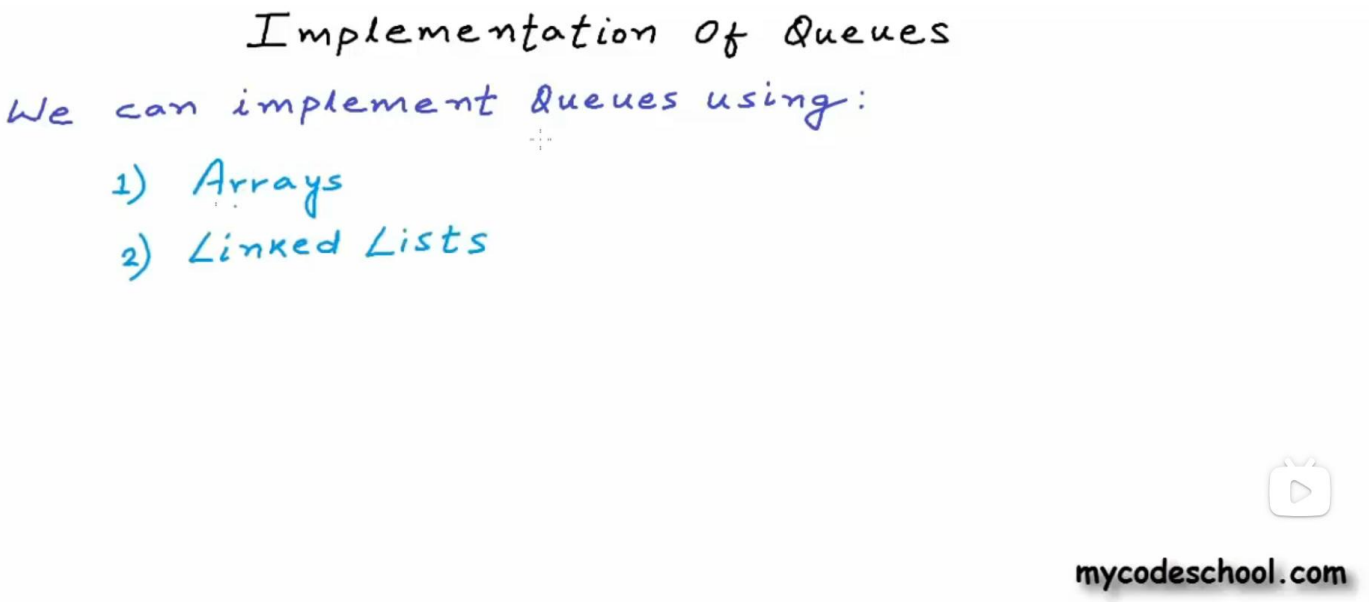
计算机的处理器也是一个共享的资源，很多程序或者说进程需要处理器的时间片来执行。处理器一次只能为一个进程服务。处理器用来执行所有的指令，执行所有的算术和逻辑操作，因此，这些进程会被放入一个队列。

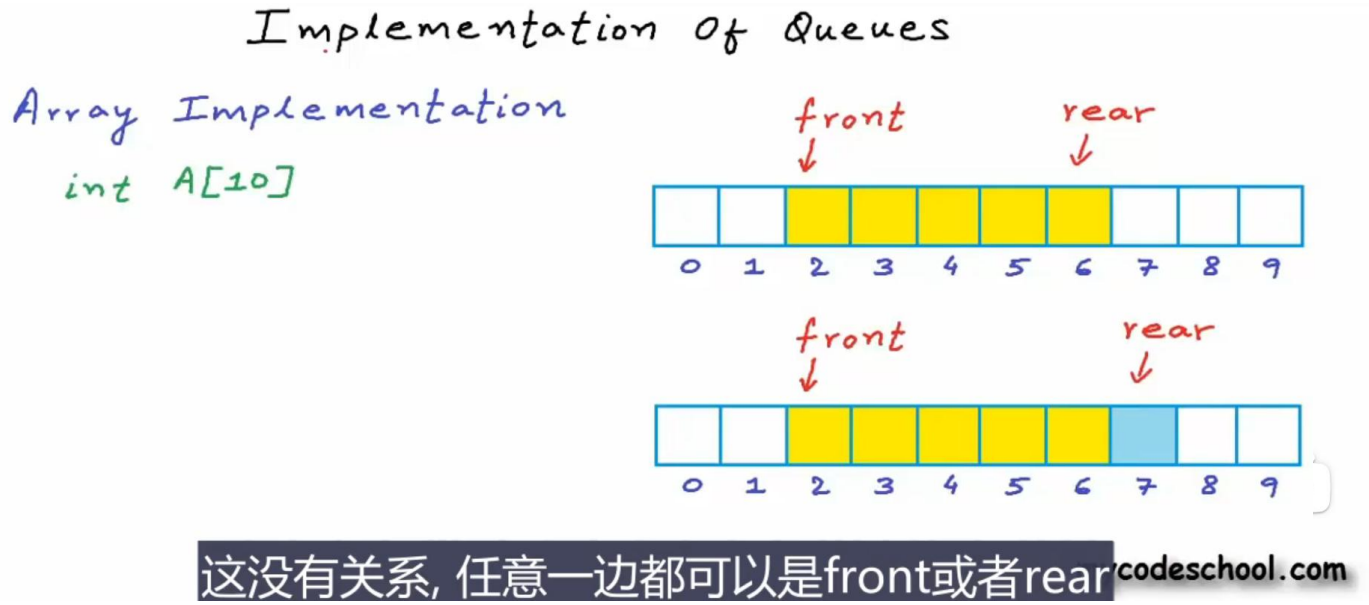
总的来说，队列可以被用来模拟一系列需要等待的场景，

总结：

3.2 使用数组实现一个队列

代码见





数组的任意一边都可以是 front 或者 rear，只不过对于一个元素来说，必须从 rear 这一端插入，必须从 front 这一端移出，因此，任意时刻，数组中的一段，从 front 索引位置到 rear 索引位置是我们的队列。数组中剩下的位置是空闲空间，可以用来扩张这个队列。插入一个元素（入队）时，我们可以增加 rear，因此我们将会往 rear 的方向增加一个单元。

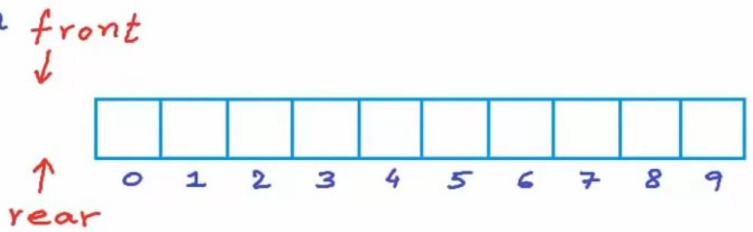
我么可以简单地增加 front 来进行出队操作，因为任何时候，只有从 front 开始 一直到 rear 才是我们的队列。增加 front 相当于把索引2这个位置从队列中丢弃了。我们并不在意不属于队列的那个单元中的值，因此针对出队操作只是增加 front 就够了。

总结:

Implementation of Queues

Array Implementation

```
int A[10]
front ← -1
rear ← -1
```



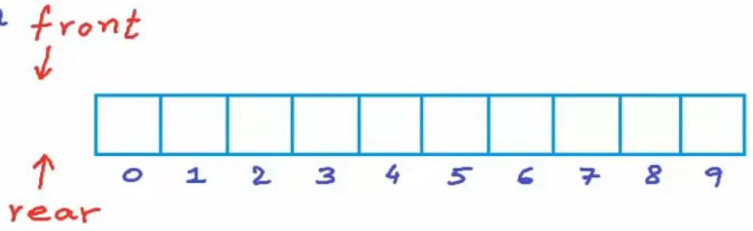
为了检查一个队列是否为空

mycodeschool.com

Implementation of Queues

Array Implementation

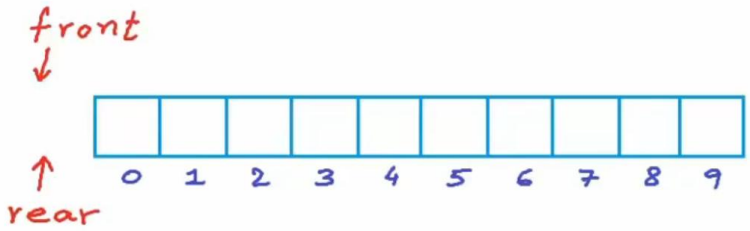
```
int A[10]
front ← -1
rear ← -1
IsEmpty()
{ if front==-1 && rear==-1
      return true
  else
      return false
}
```

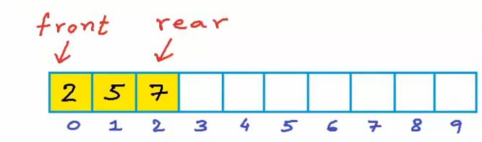


对于一个空队列来说没有front和rear

mycodeschool.com

```
Enqueue(x)
{ if IsFull()
      return
  else if IsEmpty()
  {
    front ← rear ← 0
  }
  else
  {
    rear ← rear+1
  }
  A[rear] ← x
```



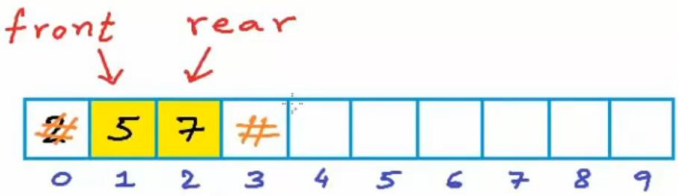


Enqueue(2)
Enqueue(5)
Enqueue(7)

mycodeschool.com

总结:

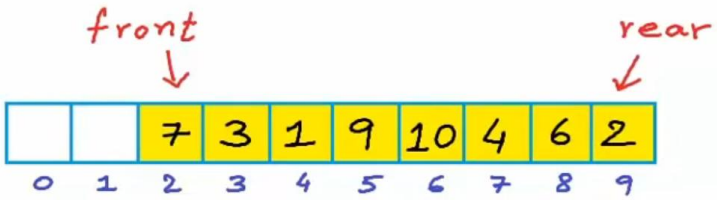
```
Dequeue()
{
  if IsEmpty()
      return
  else if front == rear
      front ← rear ← -1
  else
      front ← front + 1
}
```



Enqueue(2)
Enqueue(5)
Enqueue(7)
Dequeue()

mycodeschool.com

```
Dequeue()
{
  if IsEmpty()
      return
  else if front == rear
      front ← rear ← -1
  else
      front ← front + 1
}
```



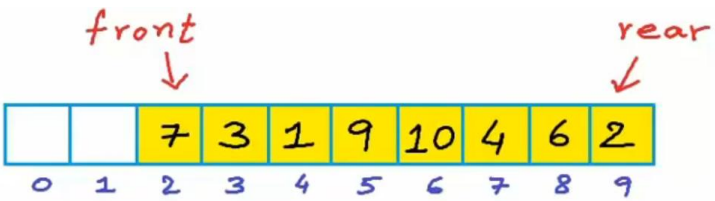
Enqueue(2)
Enqueue(5)
Enqueue(7)
Dequeue()
Enqueue(3)
Enqueue(1)

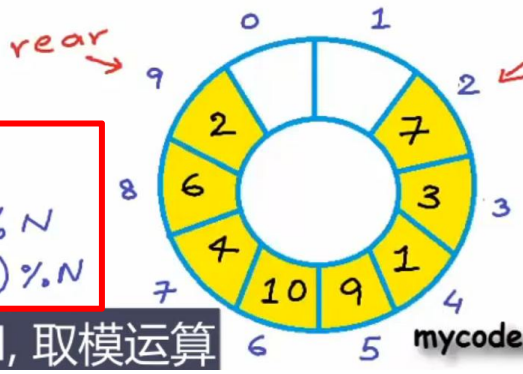
Enqueue(9)
Enqueue(10)
Enqueue(4)
Enqueue(6)
Dequeue()
Enqueue(2)

这个时候我们不能再执行Enqueue了，因为不能增加rear了

这个时候我们不能再执行 Enqueue 了，因为不能增加 rear 了， Enqueue 操作会失败。现在我们有两个未使用的单元，但是根据我们写的逻辑，我们没有办法使用左边剩下的这两个单元。实际上这是一个现世的问题，所有 front 左边的单元都不会再被使用，它们被浪费掉了。
我们可以做点什么来使用这些单元吗？事实上我们可以使用循环数组的概念。

```
Dequeue()
{
  if IsEmpty()
      return
  else if front == rear
      front ← rear ← -1
  else
      front ← front + 1
}
```





Current position = i
Next position = (i+1) % N
Previous position = (i+N-1) % N

N是数组的长度 (i + 1) % N, 取模运算

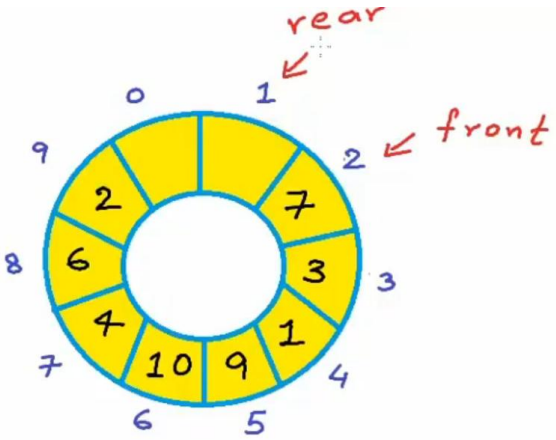
如果 i 不等于 N - 1, 取模操作并不会有什么影响

mycodeschool.com

总结：

通过使用循环数组，我们可以在执行 Enqueue 操作的时候增加 rear 了，只要数组中还有未被使用的单元，所以接下来我们要修改一下前面的伪代码。IsEmpty() 还是保持不变，我们需要修改 Enqueue(x) 的代码。

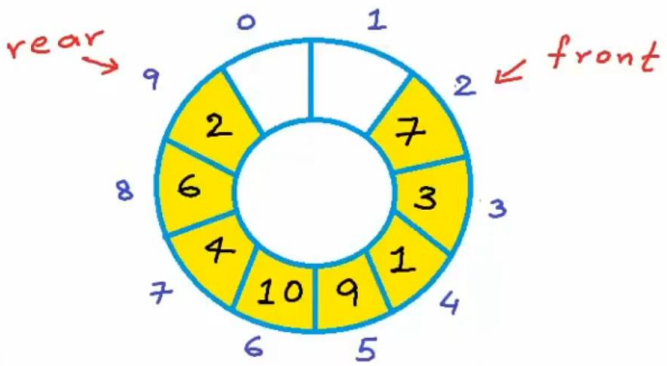
```
Enqueue(x)
{
  if (rear+1)%N == front
      return
  else if IsEmpty()
  {
    front ← rear ← 0
  }
  else
  {
    rear ← rear+1
  }
  A[rear] ← x
}
```



整个数组已经被用尽，这种情况不用做改变 mycodeschool.com

```
Enqueue(x)
{
  if (rear+1)%N == front
      return
  else if IsEmpty()
  {
    front ← rear ← 0
  }
  else
  {
⇒   rear ← (rear+1) % N
  }
  A[rear] ← x
}
```

(9+1) % 10 = 0

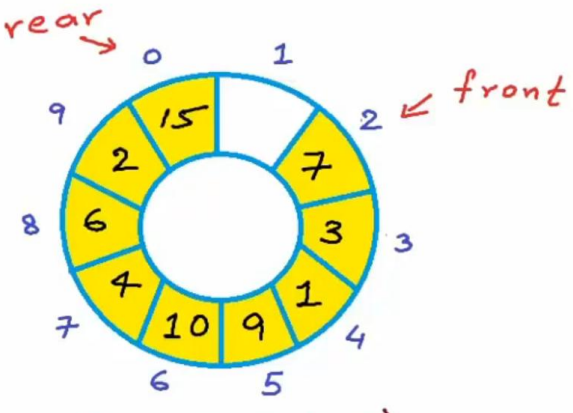


Enqueue(15)

这个求值结果是0，现在我的rear值是0 mycodeschool.com

```
Enqueue(x)
{
  if (rear+1)%N == front
      return
  else if IsEmpty()
  {
    front ← rear ← 0
  }
  else
  {
⇒   rear ← (rear+1) % N
  }
  A[rear] ← x
}
```

(9+1) % 10 = 0

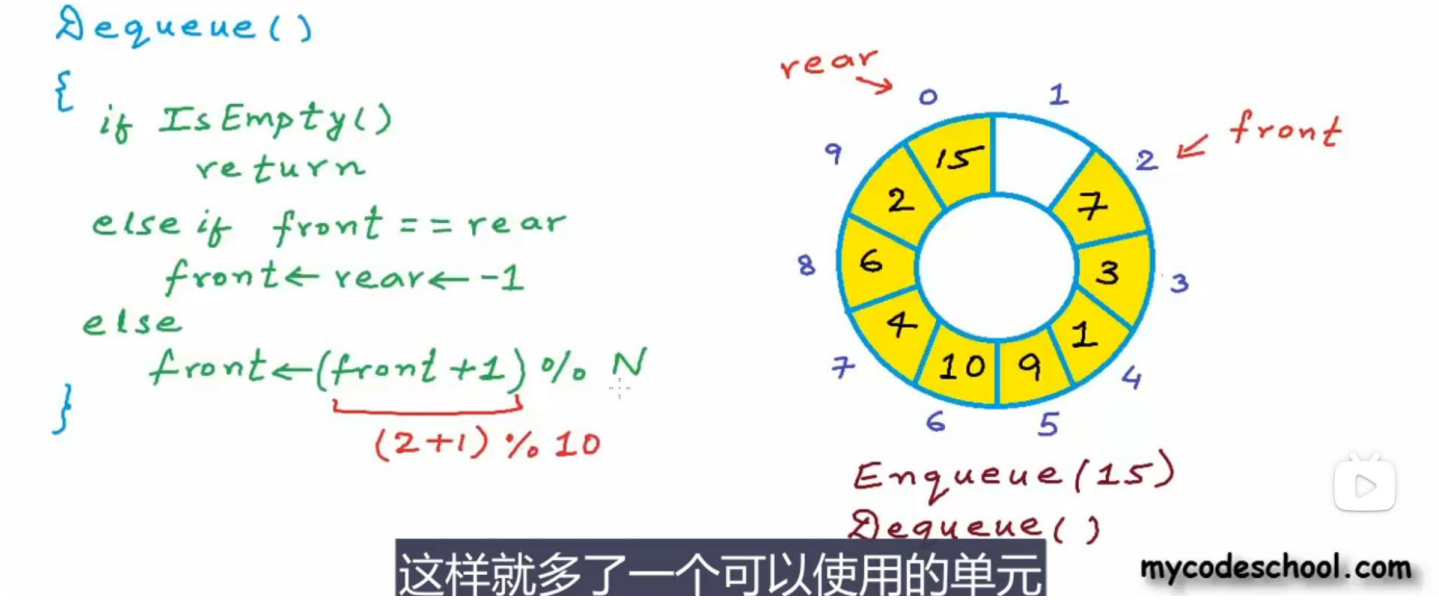


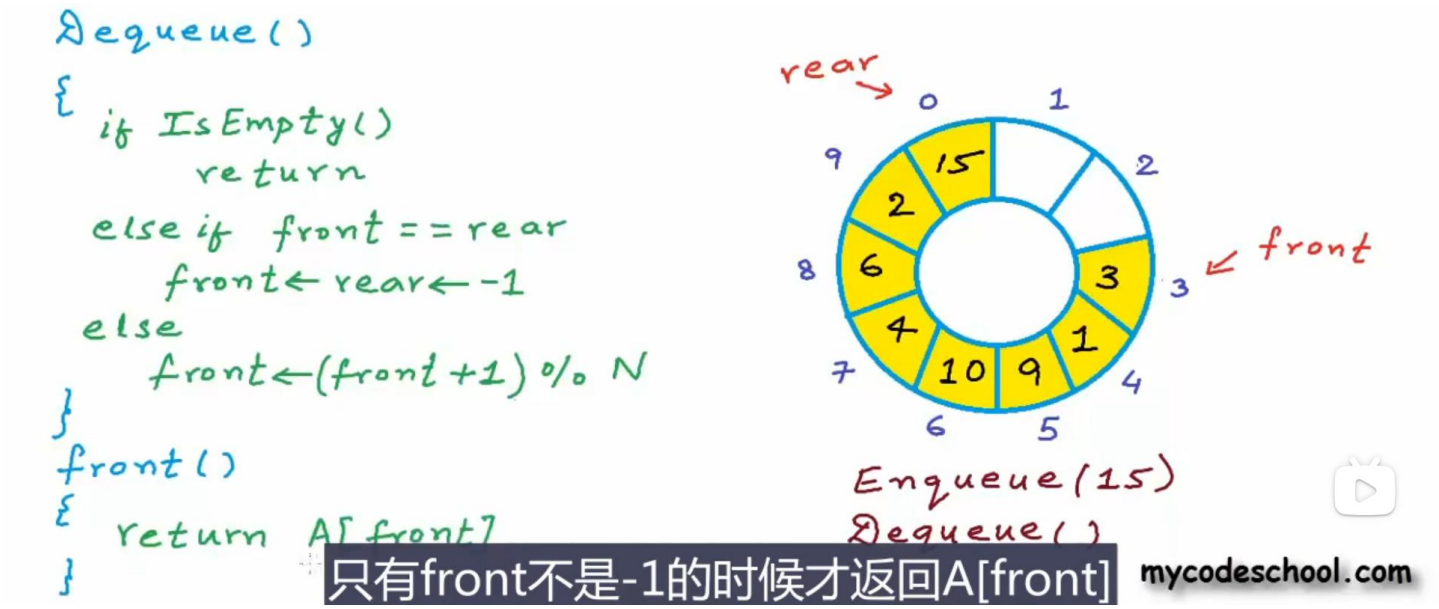
Enqueue(15)

mycodeschool.com

总结：

接下来再看看 Dequeue() 函数需要做什么改变。前面两个条件无需改变。



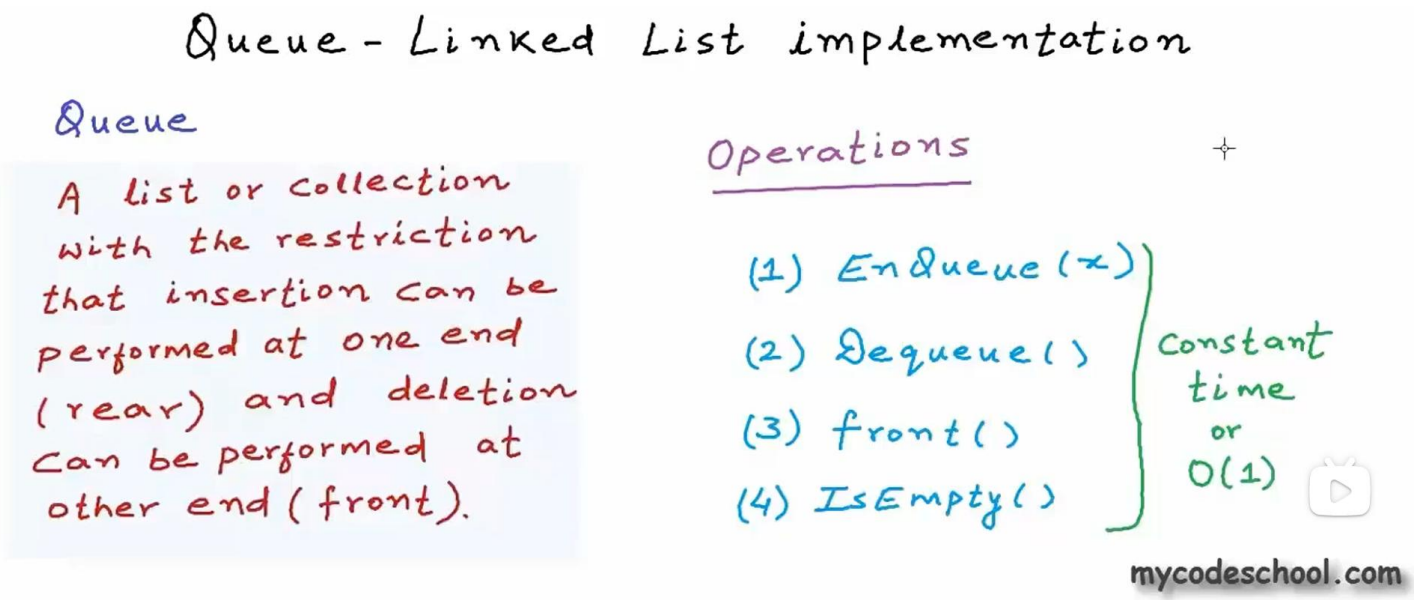


本小节写的所有函数都是常数时间的，它们的时间复杂度将是 O(1)，我们在函数中所执行的都是简单的算术和赋值操作。没有出现耗时的操作比如执行循环，因此时间不依赖于队列的大小或者其它的变量。

总结:

3.3 使用链表实现一个队列

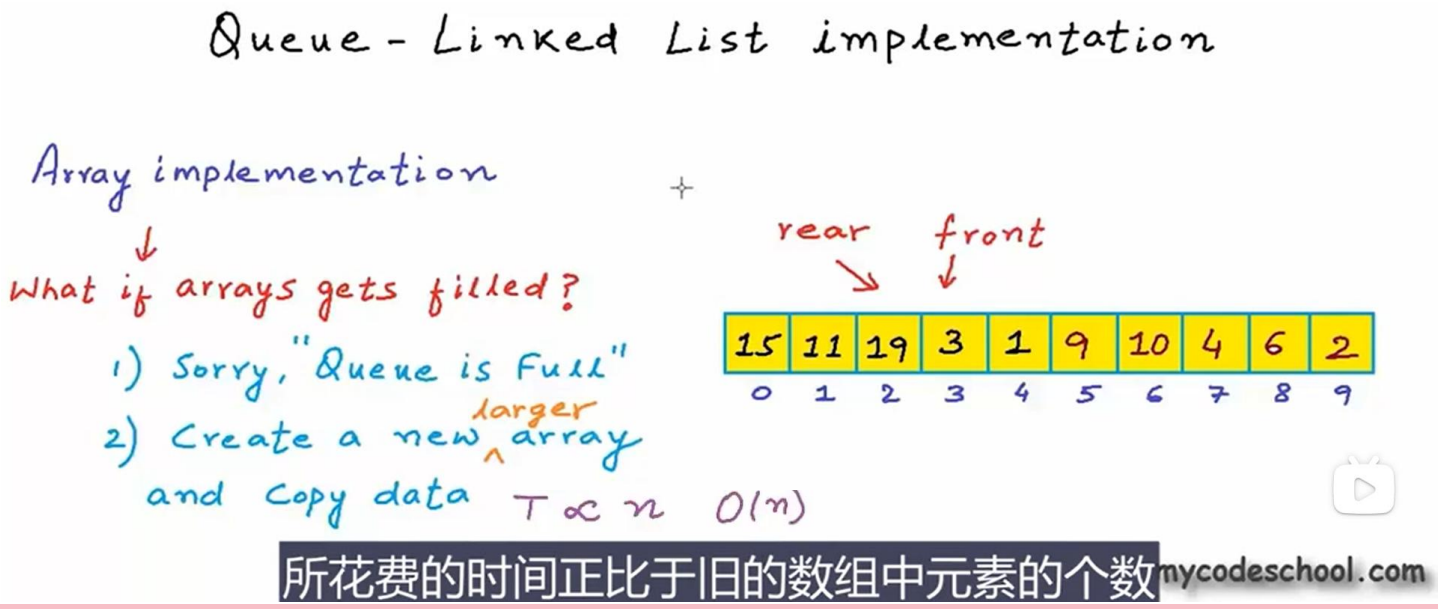
代码见



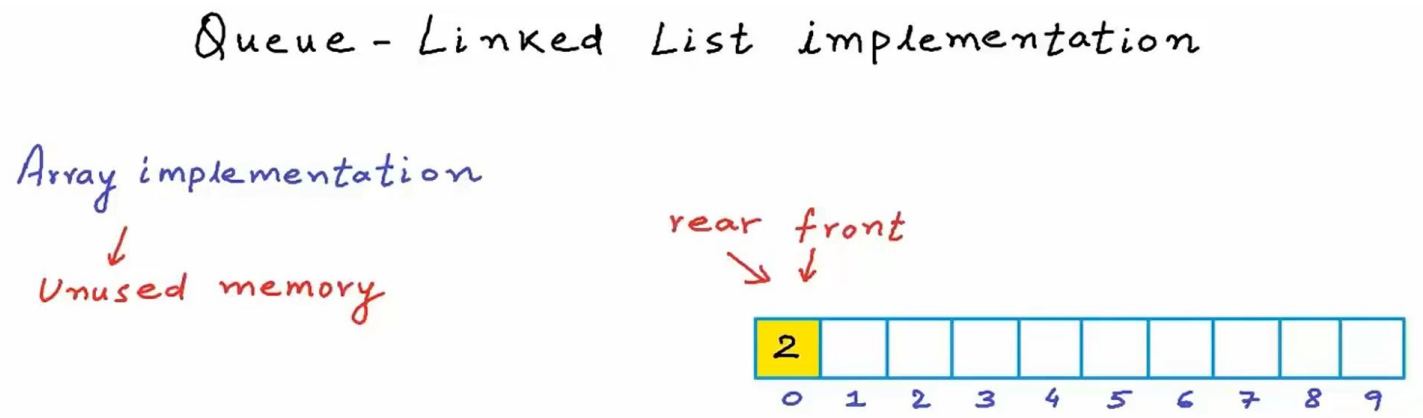
一旦数组耗尽，我们有两种选择。

- 我们可以拒绝插入，提示队列已满不能再插入任何元素了。
- 或者我们可以创建一个新的更大的数组，然后把之前数组中的元素拷贝到新的更大的数组中去。

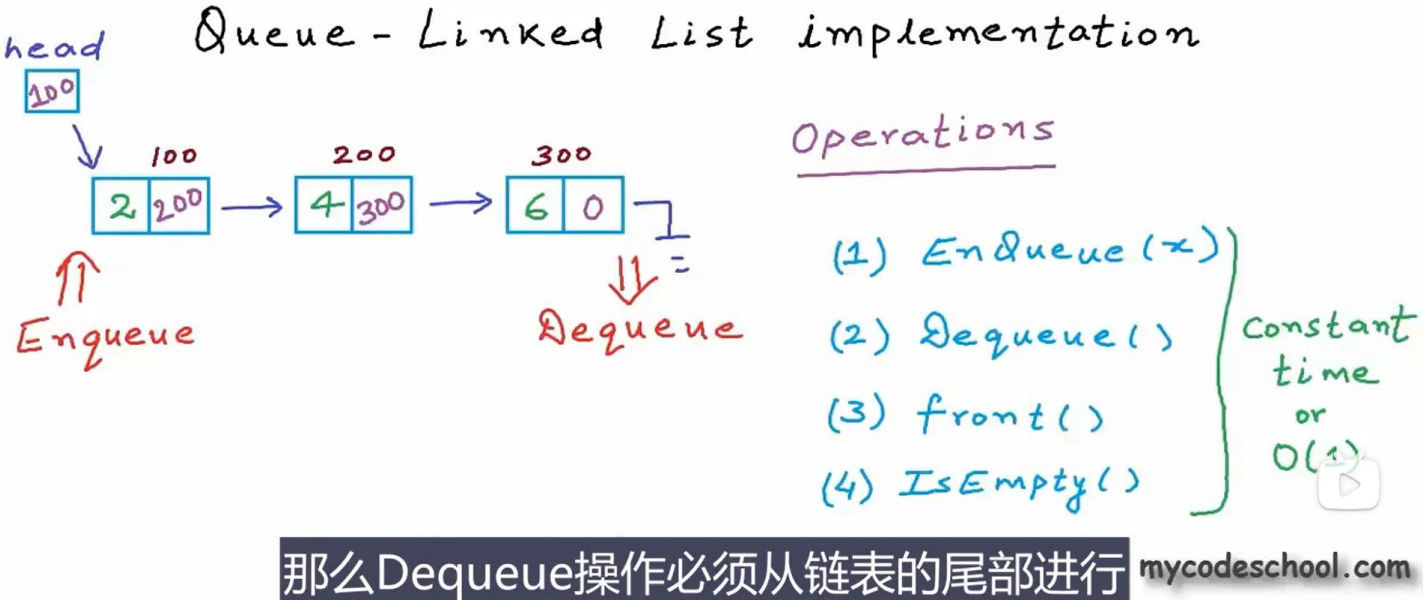
这两种方式（时间和空间）消耗很大，我们可以通过使用链表实现队列来避免这个问题。



总结：

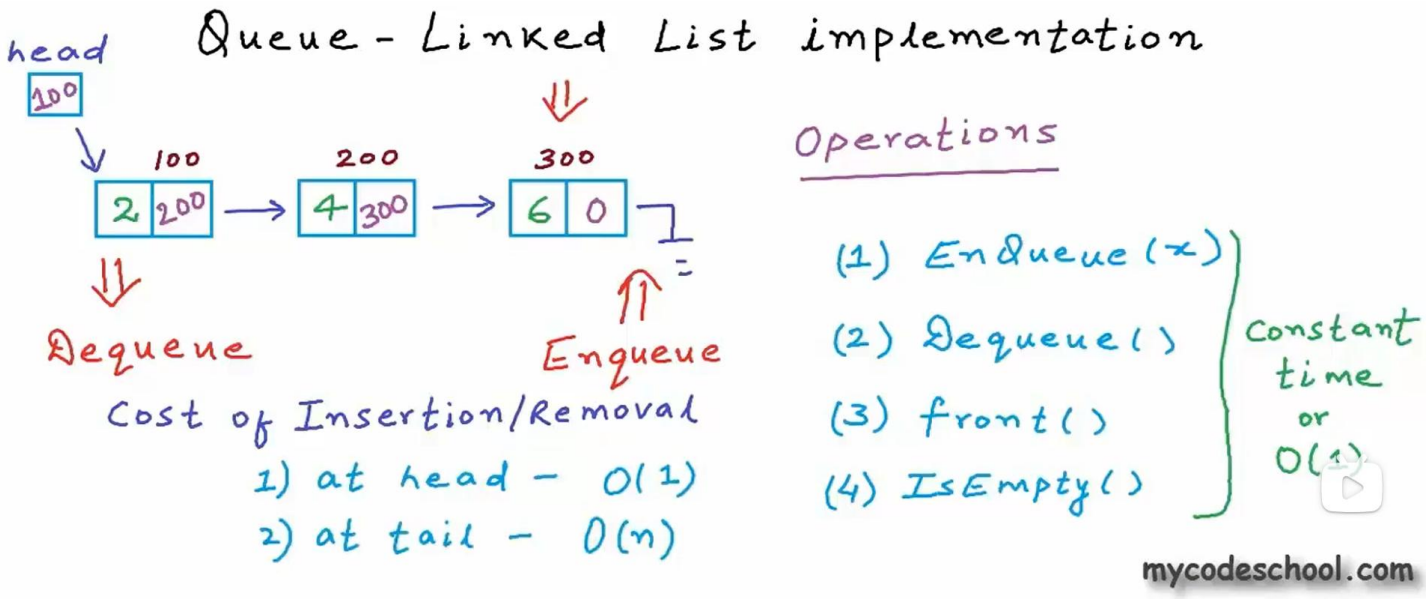


用数组来实现还有另外一个问题，我们有一个很大的数组，但是队列可能仅使用了其中的很小一部分，就像这里，数组的90%都没有被使用。内存是重要的资源，我们应该想办法避免不必要的内存占用。这并不是说在一个现代系统中多占用那一点点没有被使用的内存真的会出现实际的问题，而是说我们在设计解决方案和算法的时候，我们应该分析和理解这些可能的影响。我们现在来看看用链表来实现会有什么样的好处。

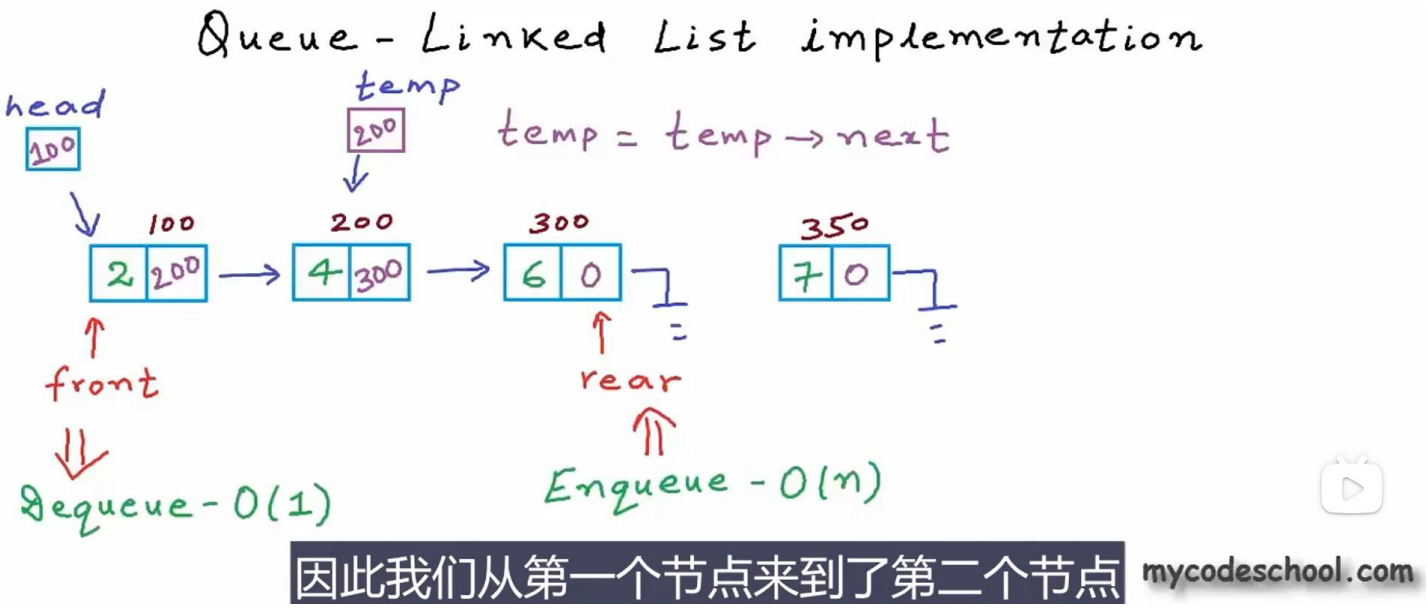


无论从哪一端，无论是什么操作，我们需要考虑一个需求，就是这些操作必须是时间常数的，它们的时间复杂度是 O(1)。

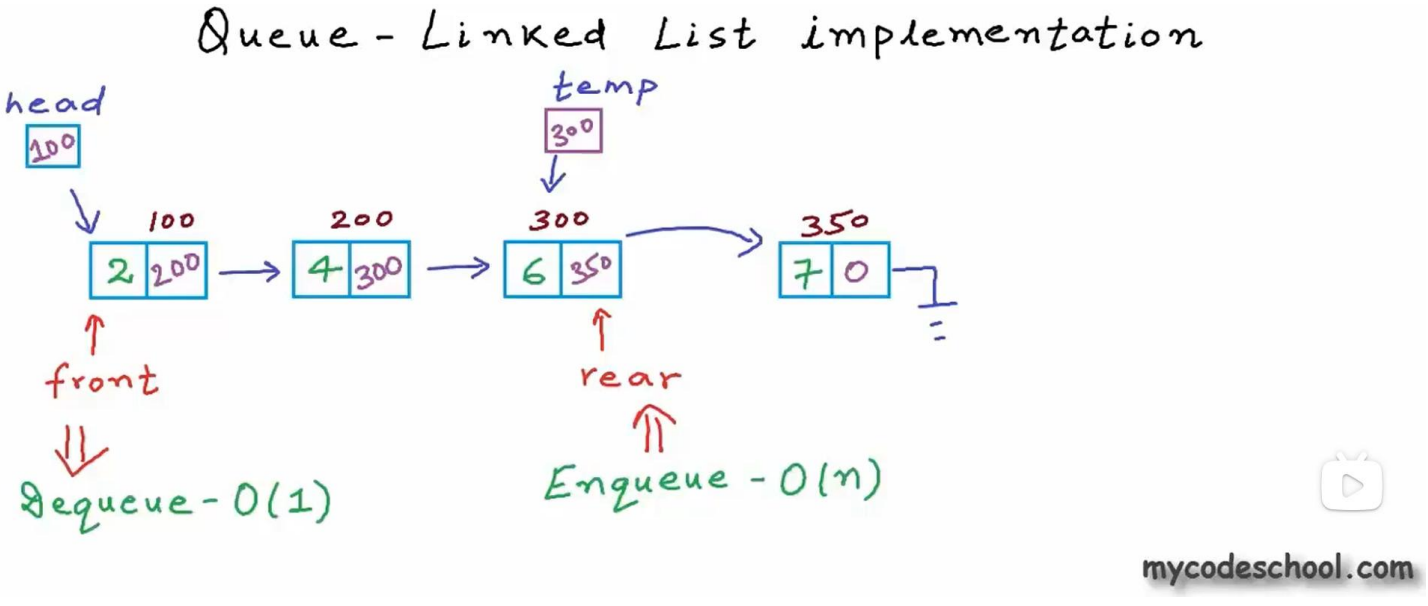
总结：



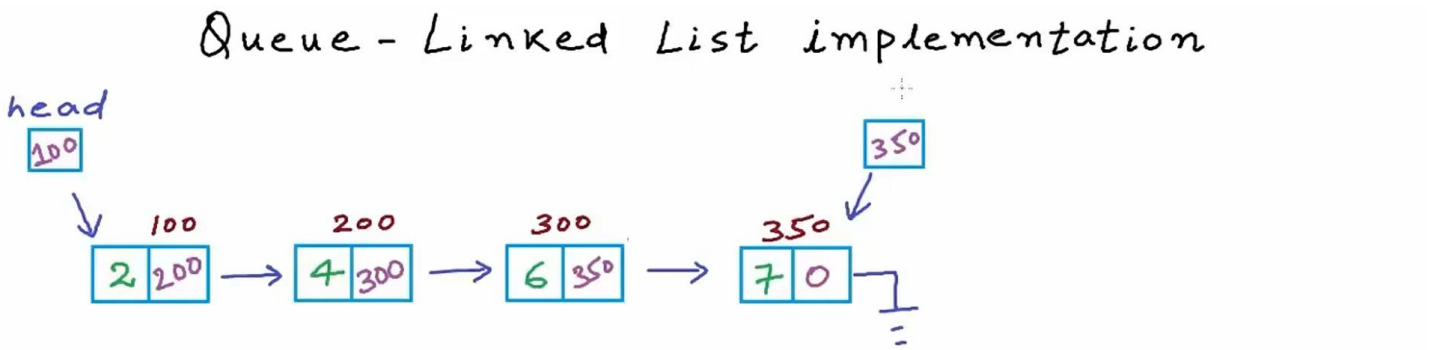
但是我们知道从链表尾部插入或者删除一个节点的时间复杂度是O(n)。所以在一个普通的链表实现中，如果从一端插入从另一端删除，那么 Enqueue 或者 Dequeue 的其中一个（取决于我们选择哪一端），它的时间复杂度会是 O(n)。但是现在的需求是所有这些操作的时间复杂度是常数。为此，我们肯定需要做些什么，来确保这些操作都是常数时间。



因此我们从第一个节点来到了第二个节点



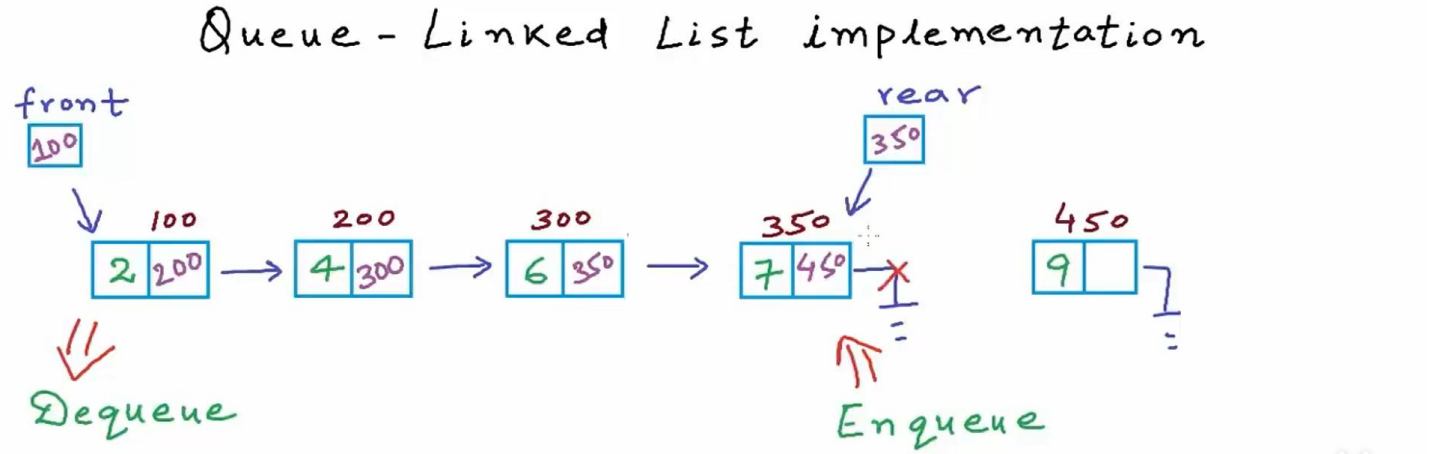
总结：



我可以把这个变量叫做tail或者rear

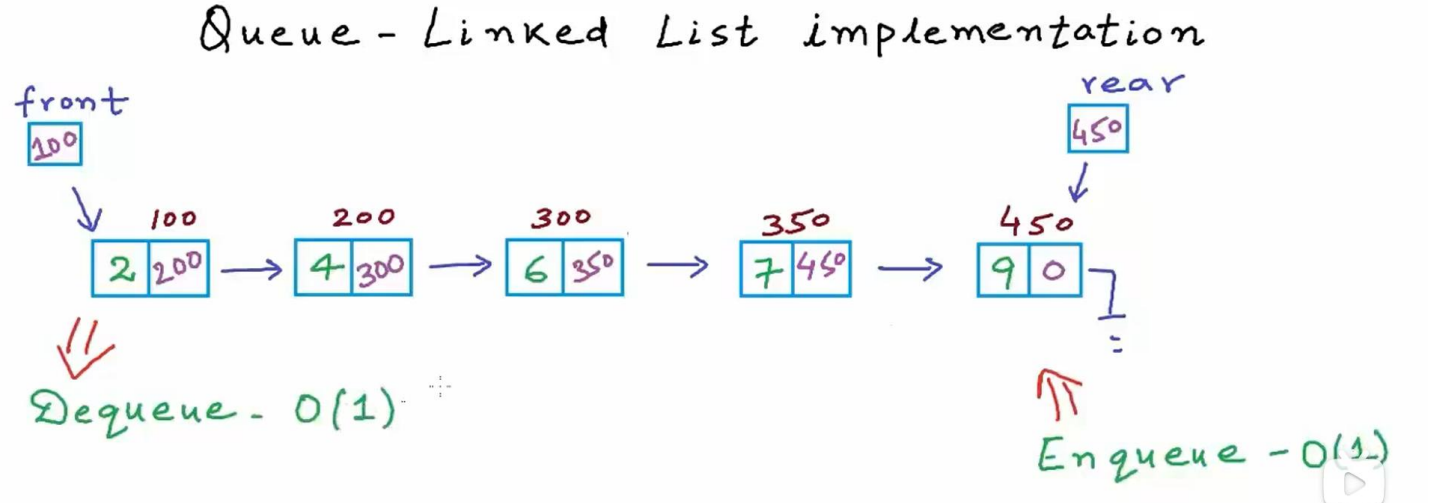
mycodeschool.com

我们能做的是，我们可以避免整个遍历，就是像链表头一样，用多一个变量来存储尾结点的地址。任何的插入或者删除，我们需要同时更新 front 或者 rear。



更新这里的地址域，由此建立这个链接

mycodeschool.com

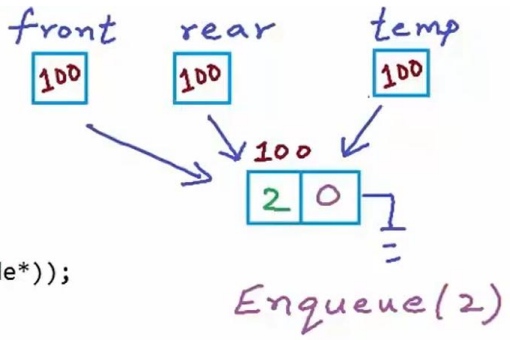


时间复杂度都是O(1)

mycodeschool.com

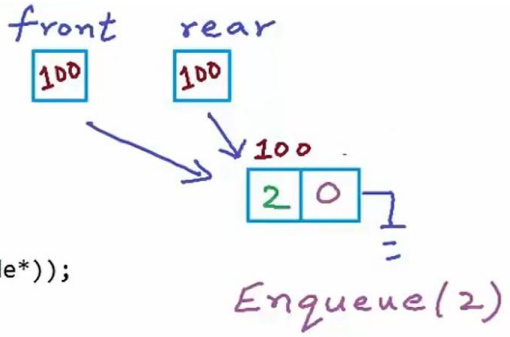
总结：

```
struct Node {
    int data;
    struct Node* next;
};
struct Node* front = NULL;
struct Node* rear = NULL;
void Enqueue(int x) {
    struct Node* temp =
        (struct Node*)malloc(sizeof(struct Node*));
    temp->data =x;
    temp->next = NULL;
    if(front == NULL && rear == NULL){
        front = rear = temp;
        return;
    }
    rear->next = temp;
    rear = temp;
}
```



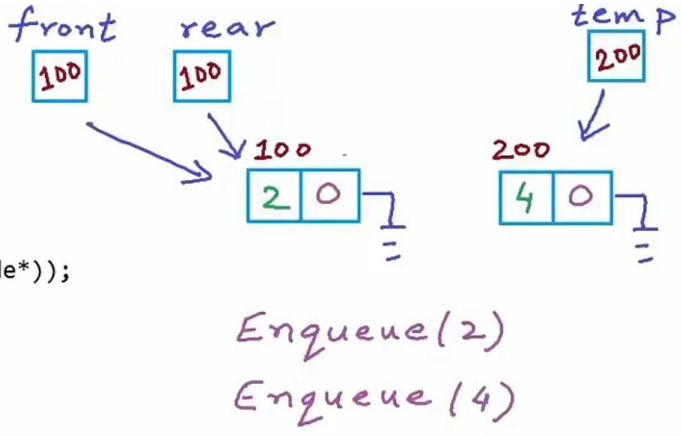
temp作为一个临时变量将被(从栈上)清除 mycodeschool.com

```
struct Node {
    int data;
    struct Node* next;
};
struct Node* front = NULL;
struct Node* rear = NULL;
void Enqueue(int x) {
    struct Node* temp =
        (struct Node*)malloc(sizeof(struct Node*));
    temp->data =x;
    temp->next = NULL;
    if(front == NULL && rear == NULL){
        front = rear = temp;
        return;
    }
    rear->next = temp;
    rear = temp;
}
```



我们返回 mycodeschool.com

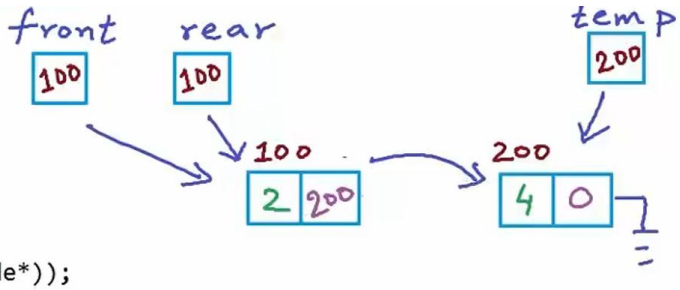
```
struct Node {
    int data;
    struct Node* next;
};
struct Node* front = NULL;
struct Node* rear = NULL;
void Enqueue(int x) {
    struct Node* temp =
        (struct Node*)malloc(sizeof(struct Node*));
    temp->data =x;
    temp->next = NULL;
    if(front == NULL && rear == NULL){
        front = rear = temp;
        return;
    }
    rear->next = temp;
    rear = temp;
}
```



rear->next=temp, 这会把100这个节点的next字段 eschool.com

总结:

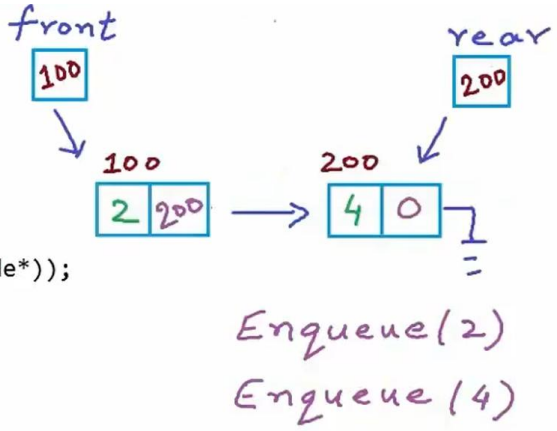
```
struct Node {
    int data;
    struct Node* next;
};
struct Node* front = NULL;
struct Node* rear = NULL;
void Enqueue(int x) {
    struct Node* temp =
        (struct Node*)malloc(sizeof(struct Node*));
    temp->data =x;
    temp->next = NULL;
    if(front == NULL && rear == NULL){
        front = rear = temp;
        return;
    }
    rear->next = temp;
    rear = temp;
}
```



现在我们保存新rear节点的地址

mycodeschool.com

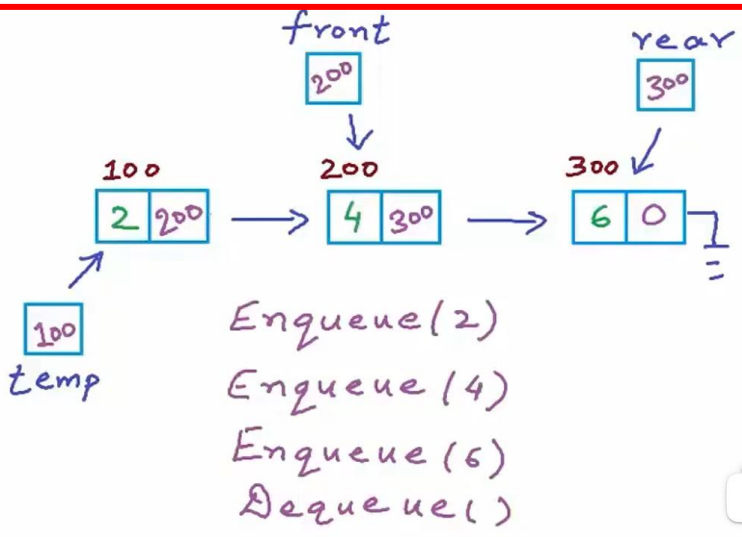
```
struct Node {
    int data;
    struct Node* next;
};
struct Node* front = NULL;
struct Node* rear = NULL;
void Enqueue(int x) {
    struct Node* temp =
        (struct Node*)malloc(sizeof(struct Node*));
    temp->data =x;
    temp->next = NULL;
    if(front == NULL && rear == NULL){
        front = rear = temp;
        return;
    }
    rear->next = temp;
    rear = temp;
}
```



这就是经过第二次Enqueue调用之后

mycodeschool.com

```
void Dequeue() {
    struct Node* temp = front;
    if(front == NULL) return;
    if(front == rear) {
        front = rear = NULL;
    }
    else {
        front = front->next;
    }
    free(temp);
}
```



而使用free函数需要给出节点的地址

mycodeschool.com

出队，Dequeue 也有若干种情形。

- 队列可能是空的，这种情况下我们可以打印一条错误信息然后返回。空队列的时候，变量 front 和 rear 将会都是 NULL，我们检查其中一个就够了。
- 当 front 和 rear 都相等的时候，我们把这两者都设置为 NULL。
- 在其它情形下，我们直接让 front 指向下一个节点。此时这里的代码为什么还要创建一个临时的节点指针变量 temp 呢？因为虽然出队了之后 front 虽然已经指向了下一个节点，但是上一个节点还在内存中，**动态内存需要显式来释放**，而使用 free() 函数需要给出节点的地址。

总结：