# 数据结构和算法
## （Python描述）

**郭 炜**

**微信公众号**

微博：http://weibo.com/guoweiofpku

**学会程序和算法，走遍天下都不怕!**

讲义照片均为郭炜拍摄

# 队列

# 队列的概念和实现

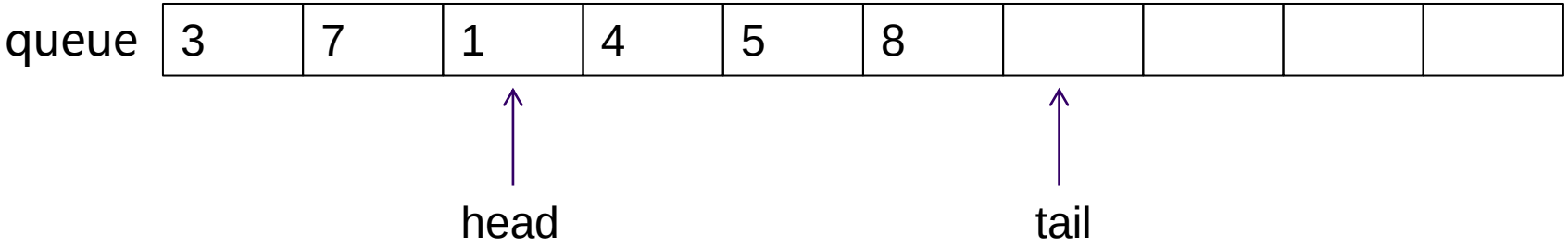
美国黄石公园

# 队列的概念

- 即排队的队列。只能一头进(push)，另一头出(pop)。先进先出
- 要求进出的复杂度都是O(1)
- 如果用列表的append进，pop(0)出，则出的复杂度为O(n)

# 队列的实现方法一

用足够大的列表实现，维护一个队头指针和队尾指针，初始：head=tail = 0




- head指向队头元素，tail指向队尾元素的后面
- push(x)的实现：

```
queue[tail] = x
tail+=1
```

- pop()的实现：

```
head += 1
```

- 判断队列是否为空：

```
head == tail
```

# 队列的实现方法二 ★

如果不想浪费空间开足够大的列表，而是想根据实际情况分配空间，则可以用列表+头尾循环法实现队列

1) 预先开设一个capacity个空元素的列表queue，head = tail = 0
2) 列表没有装满的情况下：

- push(x)的实现：

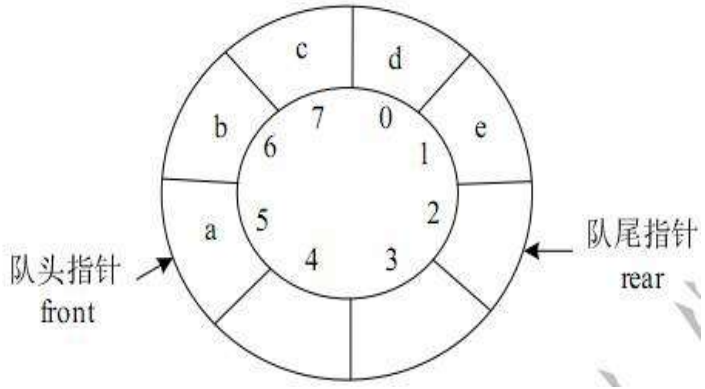
  queue[tail] = x

  tail = (tail+1) % capacity

- pop()的实现：

  head = (head+1) % capacity

capacity可以是4,8,16.....



# 队列的实现方法二

3) 如何判断队列是否为空：

方法1：维护一个元素总数size, size == 0即为空

方法2：不维护size,浪费queue中一个单元的存储空间

head == tail 即为空

4) 如何判断队列是否为满:

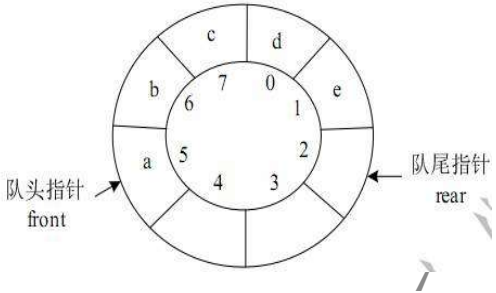
方法1：维护一个元素总数size, size == capacity即为满

方法2：不维护size, 浪费queue中一个单元的存储空间，

(tail + 1) % capacity == head 即为满

如果不浪费，就无法区分head == tail

是队列空导致，还是队列满导致



# 队列的实现方法二

4) 若一个push操作后导致列表满：

1. 建一个大小是原列表k倍大的新列表(k>1,可以取1.5,2.....)
2. 将原列表内容全部拷贝到新列表，作为新队列
3. 重新设置新列表的head和tail
4. 原列表空间自动被Python解释器回收

- 导致队列满的push的时间复杂度是O(n)。平均push操作是O(1)
- Python列表append做到O(1)的实现也是这种原理,且k取1.125，空间换时间
- 若每次增加空间只增加固定数量，比如20个单元，则push平均复杂度还是O(n)

# 队列的实现方法二

```
class Queue:
    _initC = 8      #存放队列的列表的初始容量
    _expandFactor = 1.5   #扩充容量时容量增加的倍数
    def __init__(self):
        self._q = [None for i in range(Queue._initC)]
        self._size = 0                    #队列元素个数
        self._capacity = Queue._initC #队列最大容量
        self._head = self._rear = 0
    def isEmpty(self):
        return self._size == 0
    def front(self):   #看队头元素。空队列导致re
        if self._size == 0:
            raise Exception("Queue is empty")
        return self._q[self._head]
```

# 队列的实现方法二

```
def back(self):    #看队尾元素，空队列导致re
    if self._size == 0:
        raise Exception("Queue is empty")
    if self._rear > 0:
        return self._q[self._rear - 1]
    else:
        return self._q[-1]
```

# 队列的实现方法二

```
def push(self,x):
    if self._size == self._capacity:
        tmp = [None for i in range(
                int(self._capacity*Queue._expandFactor))]
        k = 0
        while k < self._size:
            tmp[k] = self._q[self._head]
            self._head = (self._head + 1) % self._capacity
            k += 1
        self._q = tmp  #原来self._q的空间会被Python自动释放
        self._q[k] = x
        self._head,self._rear = 0,k+1
        self._capacity = int(
                self._capacity*Queue._expandFactor)
    else:
        self._q[self._rear] = x
        self._rear = (self._rear + 1) % self._capacity
    self._size += 1
```

# 队列的实现方法二

```
    def pop(self):
        if self._size == 0:
            raise Exception("Queue is empty")
        self._size -= 1
        self._head = (self._head + 1) % len(self._q)

q = Queue()
for i in range(1,314):
    q.push(i)
    print(q.back(),end=",")
print()
while not q.isEmpty():
    print(q.front(),end=",")
    q.pop()
```

# 用两个栈实现一个队列

执行push(x)操作时，将x压入栈inStack，执行pop()或front()操作时，看另一个栈outStack是否为空，若不为空，弹出栈顶元素或访问栈顶元素即可；若为空，则先将inStack中的全部元素弹出并依次压入outStack，然后再弹出或访问outStack的栈顶元素。

由于每个元素最多出入inStack各一次，出入outStack各一次，所以pop和front操作的平均复杂度是O(1)的。

# Python中的队列

collections库中的deque是双向队列，可以像普通列表一样访问，且在两端进出，复杂度都是O(1)

```
import collections
dq = collections.deque()
dq.append('a') #右边入队
dq.appendleft(2) #左边入队
dq.extend([100,200]) #右边加入100,200
dq.extendleft(['c','d']) #左边依次加入 'c','d'
print(dq.pop()) #>>200 右边出队
print(dq.popleft()) #>>d 左边出队
print(dq.count('a')) #>>1
dq.remove('c')
print(dq)       #>>deque([2, 'a', 100])
dq.reverse()
print(dq)       #>>deque([100, 'a', 2])
print(dq[0],dq[-1],dq[1]) #>>100 2 a
print(len(dq)) #>>3
```

# Python中的队列

collections库中的deque是双向队列，可以像普通列表一样访问，且在两端进出，复杂度都是O(1)

```
dq.clear()                                  #清空队列
print(len(dq),dq)                       #>0 deque([])
```