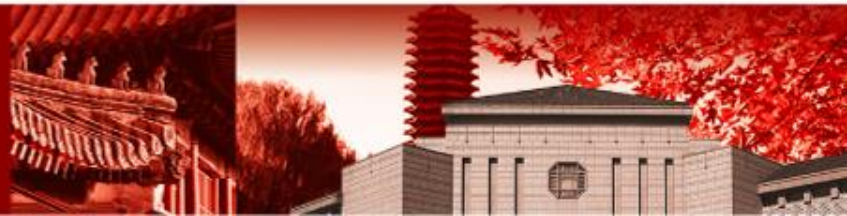
数据结构与算法B

05-队列

本章内容

- 队列
  - 定义和操作
  - 例题讲解



队列Queue：什么是队列？

- 队列是一种有次序的数据集合，其特征是，新数据项的添加总发生在一端（通常称为“尾rear”端），而现存数据项的移除总发生在另一端（通常称为“首front”端）
  - 当数据项加入队列，首先出现在队尾，随着队首数据项的移除，它逐渐接近队首。
- 新加入的数据项必须在数据集末尾等待，而等待时间最长的数据项则是队首。这种次序安排的原则称为（FIFO:First-in first-out）先进先出，或者叫“先到先服务first-come first-served”
- 队列的例子出现在我们日常生活的方方面面：排队
- 队列仅有一个入口和一个出口，不允许数据项直接插入队中，也不允许从中间移除数据项

队列Queue：什么是队列？

- 在计算机科学中有很多队列的例子
  - “打印队列”：当一台打印机面向多个用户提供服务时，由于打印速度比打印请求提交的速度要慢得多，所以有任务正在打印时，后来的打印请求就要排成队列，以FIFO的形式等待被处理。
  - “进程调度”：操作系统核心采用多个队列来对系统中同时运行的进程进行调度，由于CPU核心数总少于正在运行的进程数，将哪个进程放到CPU的哪个核心去运行多长一段时间，是进程调度需要决定的事情，而调度的原则是综合了“先到先服务”及“资源充分利用”两个出发点。
  - “键盘缓冲”：有时候键盘敲击并不马上显示在屏幕上，需要有个队列性质的缓冲区，将尚未显示的敲击字符暂存其中，队列的先进先出性质则保证了字符的输入和显示次序一致性。

队列

- 允许进行删除的这一端叫队列的头。
- 允许进行插入的这一端叫队列的尾。
- 当队列中没有任何元素时，称为空队列。
- 队列的插入操作通常称为进队列或入队列。
- 队列的删除操作通常称为退队列或出队列。

抽象数据类型Queue

- 抽象数据类型Queue是一个有次序的数据集合，数据项仅添加到“尾rear”端，而且仅从“首front”端移除，Queue具有FIFO的操作次序
- 抽象数据类型Queue由如下操作定义：
  - Queue()：创建一个空队列对象，返回值为Queue对象；
  - enqueue(item)：将数据项item添加到队尾，无返回值；
  - dequeue()：从队首移除数据项，返回值为队首数据项，队列被修改；
  - is_empty()：测试是否空队列，返回值为布尔值；
  - size()：返回队列中数据项的个数。

抽象数据类型Queue

Queue Operation	Queue Contents	Return Value
q=Queue()	[]	Queue object
q.is_empty()	[]	True
q.enqueue(4)	[4]	
q.enqueue('dog')	['dog',4]	
q.enqueue(True)	[True,'dog',4]	
q.size()	[True,'dog',4]	3
q.is_empty()	[True,'dog',4]	False
q.enqueue(8.4)	[8.4,True,'dog',4]	
q.dequeue()	[8.4,True,'dog']	4
q.dequeue()	[8.4,True]	'dog'
q.size()	[8.4,True]	2

注：队列的头部位于右端

ADT Queue的物理实现

- 顺序队列
  - –顺序表实现
  - –出队入队操作，其中一个时间复杂度为O(1)，另一个为O(n)
- 链式队列
  - –单链表实现
  - –出队入队操作，时间复杂度都为O(1)

顺序存储实现ADT Queue

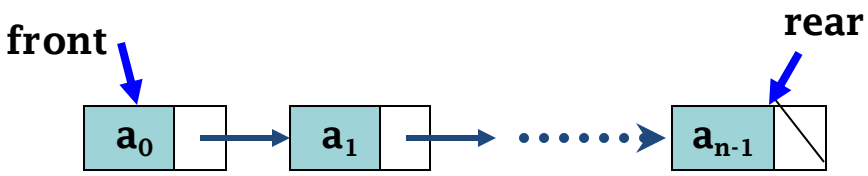
- 采用Python List来容纳Queue的数据项
- 将List的首端作为队列尾端
- List的末端作为队列首端
  - 倒过来也没问题
- enqueue复杂度为O(n)
- dequeue复杂度为O(1)
  - 倒过来的实现，复杂度也倒过来

```
class Queue:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0,item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
```

链式存储实现ADT Queue

- 采用单链表实现队列
  - –链接指针的方向是从队头到队尾
  - –队头在链头，队尾在链尾
  - –队空的条件：front == rear == None

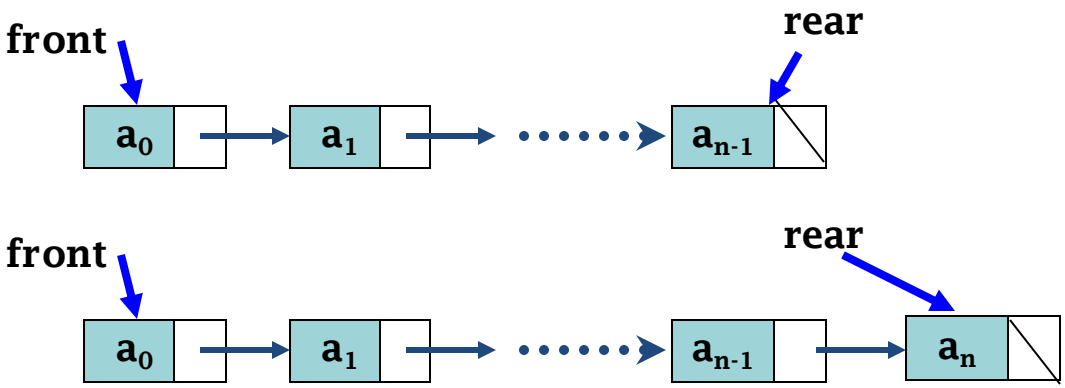
```
class Queue:
    def __init__(self):
        self.front = None
        self.rear = None
```



链式存储实现ADT Queue

- 入队操作enqueue：
  - 相当于在链表尾部插入元素
  - 队列为空时特殊处理
  - 时间复杂度仅O(1)

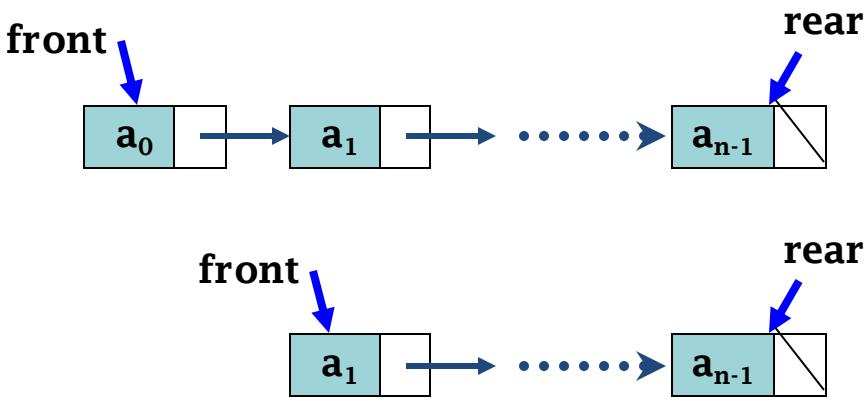
```
def enqueue(self, value):
    new_node = Node(value)
    if self.rear is None:
        self.front = self.rear = new_node
    else:
        self.rear.next = new_node
        self.rear = new_node
```



链式存储实现ADT Queue

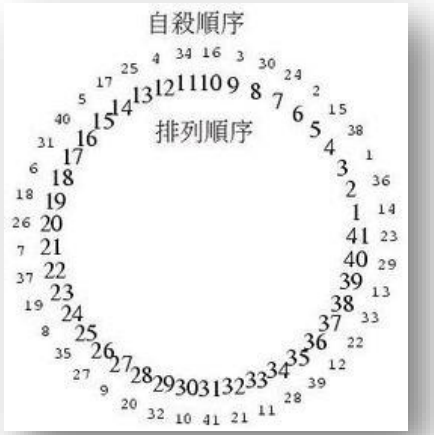
- 出队操作dequeue：
  - 相当于在链表头部删除元素
  - 队列仅有一个元素时特殊处理
  - 时间复杂度仅O(1)

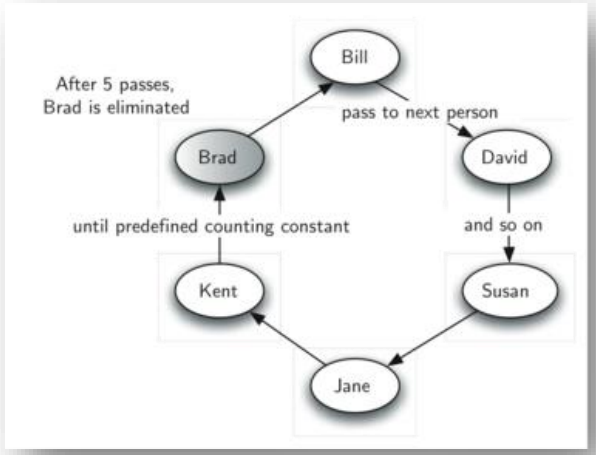
```
def dequeue(self):
    if self.front is None:
        raise IndexError("Queue is empty")
    removed_value = self.front.value
    self.front = self.front.next
    if self.front is None:
        self.rear = None
    return removed_value
```



模拟算法：热土豆问题（约瑟夫问题）

- “击鼓传花”的西方版本，传烫手热土豆，鼓声停的时候，手里有土豆的小孩就要出列。
- 如果去掉鼓，改为传过固定人数，就成了“现代版”的约瑟夫问题
  - 约瑟夫问题是传说犹太人反叛罗马人，落到困境，约瑟夫和39人决定自杀，坐成一圈儿，报数1～7，报到7的人出列自杀，结果约瑟夫给自己安排了个位置，最后活下来，投降了罗马……故事有很多版本





回顾：顺序表方式实现

- 实现Josephus算法的步骤：
  - 1. 建立顺序表
  - 2. 维护“报数”的变量，来模拟问题流程
    - 问题中，所有人围成一个圈
    - 变量应该对全体人数取模，来模拟环结构
  - 3. 迭代删除表中元素，直至只剩余一个人

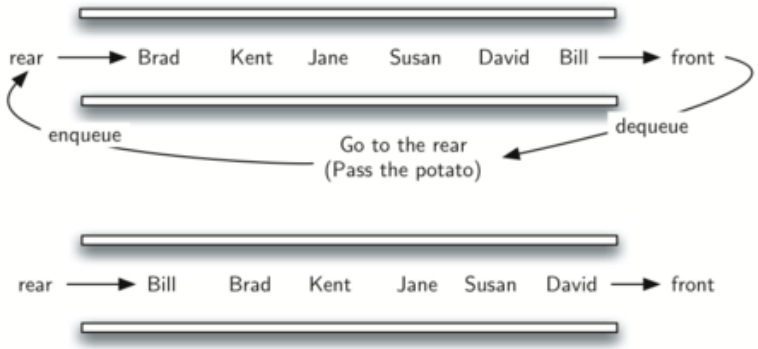
- 算法的时间复杂度分析：
  - 出列元素的删除（移动实现）为基本运算
  - 每次最多i-1个元素移动，需要n-1次
  - (n-1)+(n-2)+……+1 = n(n-1)/2 => $O(n^2)$

回顾：循环链表方式实现

- 实现Josephus算法的步骤：
  - 1. 建立循环链表；
  - 2. 出列算法；
    - 利用一个引用来维护当前的报数位置，初始为第s个结点
    - 迭代操作，直至剩余最后一人：
      - 沿着循环链表后移，模拟一次报数过程
      - 删除结点，并将当前报数位置后移
- 时间复杂度分析：
  - 创建链表，求第s个结点，求n个第m个应出列的元素
  - O(n) + O(s) + O(mn) = O(mn)

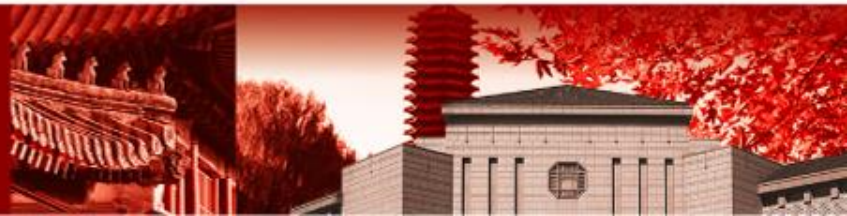
热土豆问题：算法

- 用队列来实现热土豆问题的算法，参加游戏的人名列表，以及传土豆次数num，算法返回最后剩下的人名
- 模拟程序采用队列来存放所有参加游戏的人名，按照传递土豆的方向从队首排到队尾，游戏开始时持有土豆的人在队首
- 模拟游戏开始，只需要将队首的人出队，随机再到队尾入队，算是土豆的一次传递，这时土豆就在队首的人手里
- 传递了num次后，将队首的人移除，不再入队
- 如此反复，直到队列中剩余1人
- 思考：与之前相比有什么好处？



热土豆问题：代码

```
def hot_potato(namelist, num):
    simqueue = Queue()
    for name in namelist:
        simqueue.enqueue(name)

    while simqueue.size() > 1:
        for i in range(num):
            simqueue.enqueue(simqueue.dequeue())

        simqueue.dequeue()

    return simqueue.dequeue()

print(hot_potato(["Bill","David","Susan","Jane","Kent","Brad"],7))
```

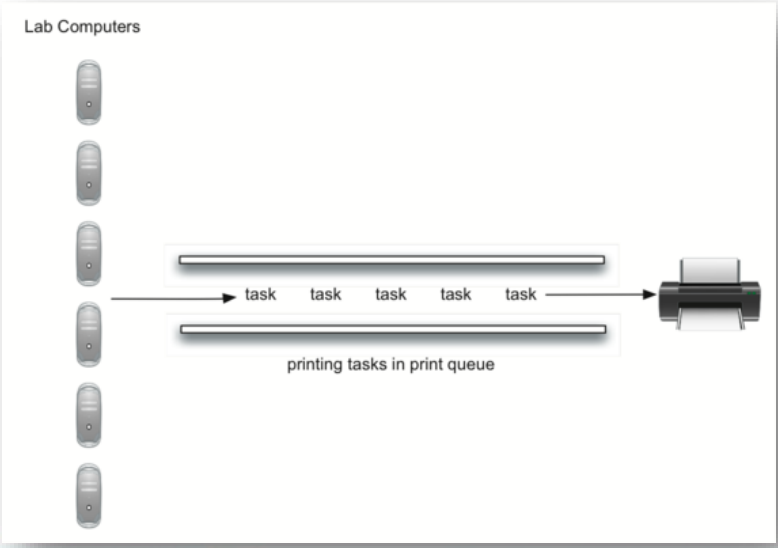



模拟算法：打印任务

- 多人共享一台打印机，采取“先到先服务”的队列策略来执行打印任务，在这种设定下，一个首要的问题就是，这种打印作业系统的容量有多大？在能够接受的等待时间内，系统能容纳多少用户以多高频率提交多少打印任务？
- 一个具体的实例配置如下：一个实验室，在任意的一个小时内，大约有10名学生在场，这一小时中，每人会发起2次左右的打印，每次1～20页，打印机的性能是：以草稿模式打印的话，每分钟10页，以正常模式打印的话，打印质量好，但速度下降为每分钟5页
- 问题是：怎么设定打印机的模式，让大家都不会等太久的前提下尽量提高打印质量？
- 我们要用一段程序来模拟这种打印任务场景，然后对程序运行结果进行分析，以支持对打印机模式设定的决策。

如何对问题建模？

- 首先对问题进行抽象，确定相关的对象和过程
  - 抛弃那些对问题实质没有关系的学生性别、年龄、打印机型号、打印内容、纸张大小等众多细节
- 对象：打印任务、打印队列、打印机
  - 打印任务的属性：提交时间、打印页数
  - 打印队列的属性：具有FIFO性质的打印任务队列
  - 打印机的属性：打印速度、是否忙



如何对问题进行建模？

- 过程：生成和提交打印任务
  - 确定生成概率：实例为每小时会有10个学生提交的20个作业，这样，概率是每180秒会有1个作业生成并提交，概率为每秒1/180。

$$\frac{20\ tasks}{1\ hour} \times \frac{1\ hour}{60\ minutes} \times \frac{1\ minute}{60\ seconds} = \frac{1\ task}{180\ seconds}$$

  - 确定打印页数：实例是1～20页，那么就是1～20页之间概率相同。
- 过程：实施打印
  - 当前的打印作业：正在打印的作业
  - 打印结束倒计时：新作业开始打印时开始倒计时，回0表示打印完毕，可以处理下一个作业
- 模拟时间：
  - 统一的时间框架：以最小单位均匀流逝的时间，设定结束时间
  - 同步所有过程：在一个时间单位里，对生成打印任务和实施打印两个过程各处理一次

打印任务问题：模拟流程

- 创建打印队列对象
- 时间按照秒的单位流逝
  - 按照既定概率1/180产生打印任务，如果有任务产生，则记录任务时间戳，加入打印队列。
  - 如果打印机空闲，打印队列中还有打印任务，则：
    - 从打印队列中移除队首打印任务，交给打印机
    - 将打印任务的生成时间戳与当前时间对比，得到等待时间
    - 记录这个任务的等待时间
    - 根据打印任务的页数，决定需要的打印时间
  - 如果打印机忙，就进行1秒的打印
  - 如果打印机中的任务打印完成，打印机就进入空闲状态
- 时间用尽，开始统计平均等待时间

打印任务问题：Python代码1

```python
class Printer:
    def __init__(self,ppm):
        self.pagerate = ppm
        self.current_task = None
        self.time_remaining = 0

    def tick(self):
        if self.current_task != None:
            self.time_remaining = self.time_remaining - 1
            if self.time_remaining <= 0:
                self.current_task = None

    def busy(self):
        if self.current_task != None:
            return True
        else:
            return False

    def start_next(self,new_task):
        self.current_task = new_task
        self.time_remaining = new_task.get_pages() * 60/self.pagerate
```

打印速度

打印任务

任务倒计时

打印1秒

打印忙？

打印新作业

```python
class Task:
    def __init__(self,time):
        self.timestamp = time
        self.pages = random.randrange(1,21)

    def get_stamp(self):
        return self.timestamp

    def get_pages(self):
        return self.pages

    def wait_time(self,current_time):
        return current_time - self.timestamp
```

生成时间戳

打印页数

等待时间

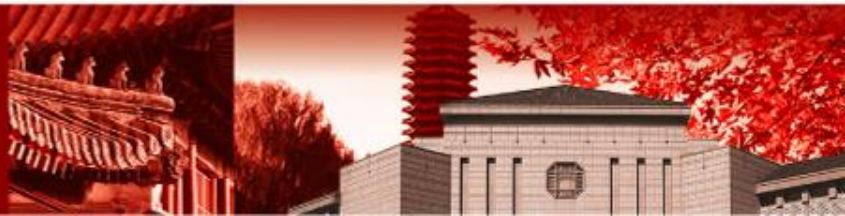
打印任务问题：Python代码2

模拟

时间流逝

```
def simulation(num_seconds,pages_per_minute):

    lab_printer = Printer(pages_per_minute)
    print_queue = Queue()
    waiting_times = []

    for current_second in range(num_seconds):

        if new_print_task():
            task = Task(current_second)
            print_queue.enqueue(task)

        if (not lab_printer.busy()) and (not print_queue.is_empty()):
            next_task = print_queue.dequeue()
            waiting_times.append(next_task.wait_time(current_second))
            lab_printer.start_next(next_task)

        lab_printer.tick()

    average_wait = sum(waiting_times)/len(waiting_times)
    print("Average Wait %6.2f secs %3d tasks remaining."%(average_wait,print_queue.size()))
```

```
def new_print_task():
    num = random.randrange(1,181)
    if num == 180:
        return True
    else:
        return False
```

打印任务问题：运行和分析1

- 按照5PPM、1小时的设定，模拟运行10次，结果如图
- 总平均等待时间93.1秒，最长的平均等待164秒，最短的平均等待26秒
- 有3次模拟，还有作业没开始打印

```
>>> for i in range(10):
    simulation(3600,5)


Average Wait  67.00 secs   0 tasks remaining.
Average Wait  26.00 secs   0 tasks remaining.
Average Wait  46.00 secs   2 tasks remaining.
Average Wait 115.00 secs   0 tasks remaining.
Average Wait  53.00 secs   0 tasks remaining.
Average Wait 121.00 secs   0 tasks remaining.
Average Wait 164.00 secs   1 tasks remaining.
Average Wait 136.00 secs   0 tasks remaining.
Average Wait 122.00 secs   2 tasks remaining.
Average Wait  81.00 secs   0 tasks remaining.
```

打印任务问题：运行和分析2

- 提升打印速度到10PPM、1小时的设定，模拟运行10次，结果如图
- 总平均等待时间12秒，最长的平均等待35秒，最短的平均等待0秒，也就是提交的时候就立即打印了
- 而且，所有作业都打印了

```
>>> for i in range(10):
    simulation(3600,10)


Average Wait  35.00 secs   0 tasks remaining.
Average Wait   8.00 secs   0 tasks remaining.
Average Wait  29.00 secs   0 tasks remaining.
Average Wait   0.00 secs   0 tasks remaining.
Average Wait  13.00 secs   0 tasks remaining.
Average Wait   5.00 secs   0 tasks remaining.
Average Wait   0.00 secs   0 tasks remaining.
Average Wait   8.00 secs   0 tasks remaining.
Average Wait  17.00 secs   0 tasks remaining.
Average Wait   5.00 secs   0 tasks remaining.
```

打印任务问题：讨论

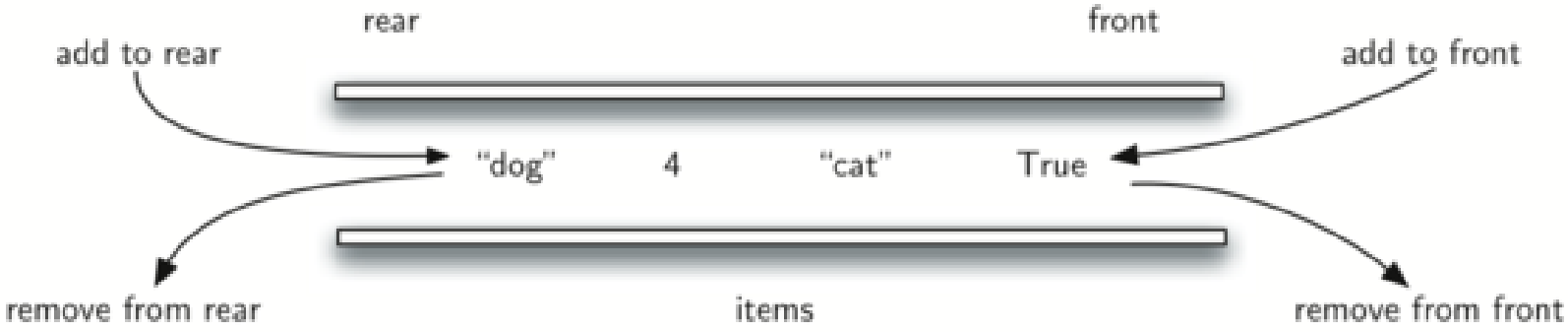
- 为了对打印机打印模式设置进行决策，我们写了一个模拟程序来评估在一定概率下的打印情况及任务等待时间
- 通过两种情况模拟仿真结果的分析，我们认识到，如果有那么多学生要拿着打印好的程序源代码赶去上课的话，那么，必须得牺牲打印质量，提高打印速度。
- 模拟系统通过对现实的仿真，在不耗费现实资源的情况下（有时候真实的实验是无法进行的），可以以不同的设定，反复多次模拟，来帮助我们进行决策。

打印任务问题：讨论

- 打印任务模拟程序还可以加进不同设定，来进行更丰富的模拟
  - 学生数量加倍了会怎么样？
  - 如果在周末，学生不需要赶去上课，能接受更长等待时间，会怎么样？
  - 如果改用Python编程，源代码大大减少，打印的页数减少了，会怎么样？
- 更真实的模拟，来源于对问题的更精细建模，以及以真实数据进行设定和运行

双端队列Deque：什么是Deque？

- 双端队列Deque是一种有次序的数据集，跟队列相似，其两端可以称作“首”“尾”端，但deque中数据项既可以从队首加入，也可以从队尾加入；数据项也可以从两端移除。
  - 某种意义上说，双端队列集成了栈和队列的能力。
- 但双端队列并不具有内在的LIFO或者FIFO特性，如果用双端队列来模拟栈或队列，需要由使用者自行维护操作的一致性



抽象数据类型Deque

- 抽象数据类型Deque是一个有次序的数据集，数据项可以从两端加入或者移除
- deque定义的操作如下：
  - Deque()：创建一个空双端队列
  - add_front(item)：将item加入队首
  - add_rear(item)：将item加入队尾
  - remove_front()：从队首移除数据项，返回值为移除的数据项
  - remove_rear()：从队尾移除数据项，返回值为移除的数据项
  - is_empty()：返回deque是否为空
  - size()：返回deque中包含数据项的个数

Python实现ADT Deque

- 采用Python List保存数据项
- List首端作为deque的尾
- List末端作为deque的首
- add_front/remove_front O(1)
- add_rear/remove_rear O(n)

```
class Deque:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def add_front(self,item):
        self.items.append(item)

    def add_rear(self,item):
        self.items.insert(0,item)

    def remove_front(self):
        return self.items.pop()

    def remove_rear(self):
        return self.items.pop(0)

    def size(self):
        return len(self.items)
```

```
from collections import deque
```

Python官方实现Deque

- 在 Python 中，collections.deque 是一个线程安全的双向队列，其底层逻辑类似于双向链表。deque 支持从队列的两端高效地添加和移除元素，时间复杂度均为 O(1)。
- 创建Deque对象：

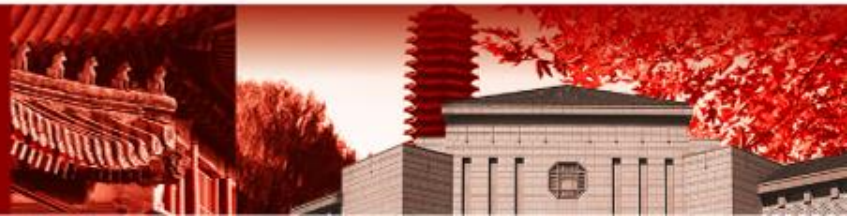
```
# 创建一个空的 deque
empty_deque = deque()
print(empty_deque)  # 输出: deque([])

# 创建一个包含初始元素的 deque
numbers_deque = deque([1, 2, 3, 4, 5])
print(numbers_deque)  # 输出: deque([1, 2, 3, 4, 5])

# 创建一个指定最大长度的 deque
maxlen_deque = deque([1, 2, 3], maxlen=3)
print(maxlen_deque)  # 输出: deque([1, 2, 3], maxlen=3)
```

Python官方实现Deque

- 从两端添加元素：
- append(x):在队列的右侧添加一个元素x。
- appendleft(x):在队列的左侧添加一个元素x。
- 从两端移除元素：
- pop():移除并返回队列右侧的元素。
- popleft():移除并返回队列左侧的元素。

```
d = deque([1, 2, 3])
d.append(4)
print(d)  # 输出: deque([1, 2, 3, 4])

d.appendleft(0)
print(d)  # 输出: deque([0, 1, 2, 3, 4])
```

```
d = deque([0, 1, 2, 3, 4])
right_element = d.pop()
print(right_element)  # 输出: 4
print(d)  # 输出: deque([0, 1, 2, 3])

left_element = d.popleft()
print(left_element)  # 输出: 0
print(d)  # 输出: deque([1, 2, 3])
```

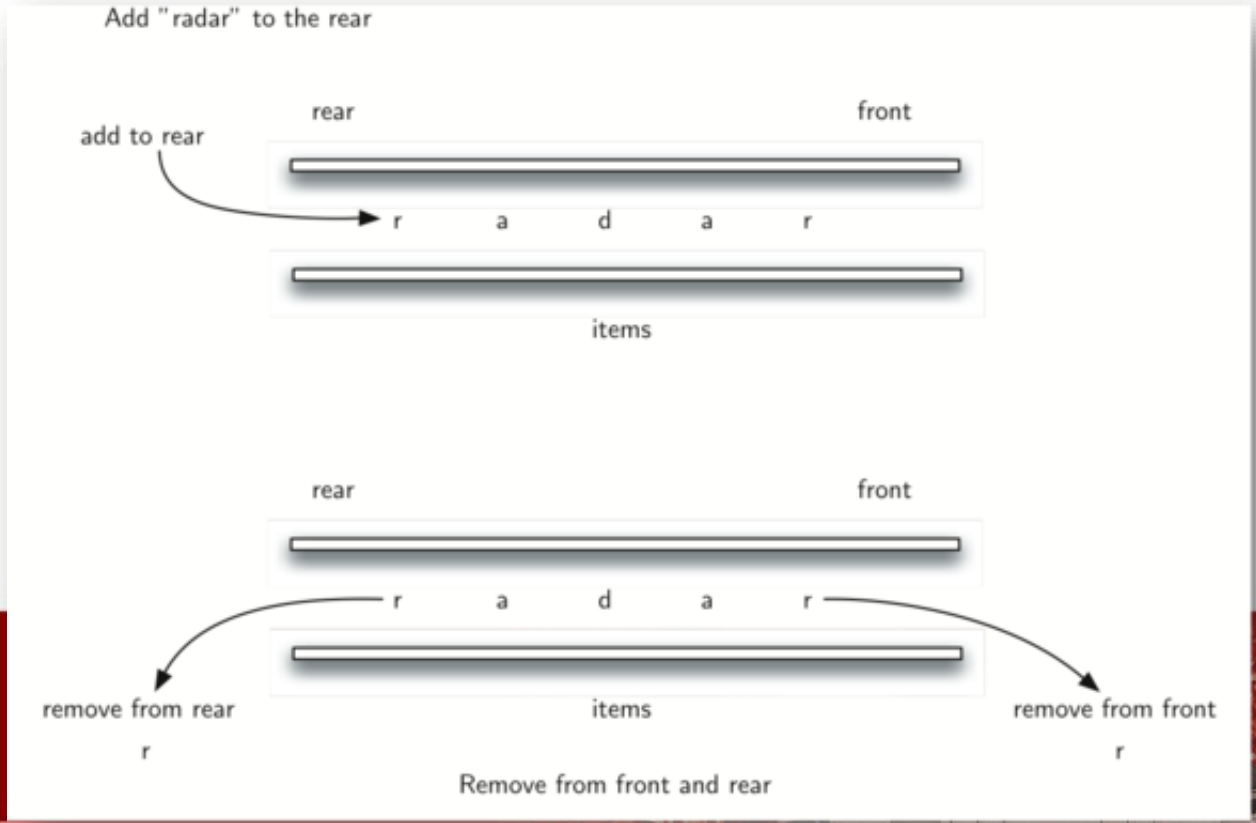
Python官方实现Deque

- 其他常用方法：
- extend(iterable)：在队列的右侧扩展一个可迭代对象的元素。
- extendleft(iterable)：在队列的左侧扩展一个可迭代对象的元素，注意元素的顺序会反转。
- rotate(n)：将队列向右旋转 n 步，如果 n 为负数，则向左旋转。

```
d = deque([1, 2, 3])
d.extend([4, 5])
print(d)  # 输出: deque([1, 2, 3, 4, 5])

d.extendleft([0, -1])
print(d)  # 输出: deque([-1, 0, 1, 2, 3, 4, 5])

d.rotate(2)
print(d)  # 输出: deque([4, 5, -1, 0, 1, 2, 3])

d.rotate(-1)
print(d)  # 输出: deque([5, -1, 0, 1, 2, 3, 4])
```

“回文词”判定

- “回文词”指正读和反读都一样的词，如radar、madam、toot等
  - “上海自来水来自海上”
- 用双端队列很容易解决“回文词”的判定问题
  - 先将需要判定的词从队尾加入deque
  - 再从两端同时移除字符判定是否相同，直到deque中剩下0个或1个字符



“回文词”判定

```
from collections import deque

def par_checker(a_string):
    char_deque = deque()
    for ch in a_string:
        char_deque.append(ch)
    while len(char_deque) > 1:
        first = char_deque.popleft()
        last = char_deque.pop()
        if first != last:
            return False
    return True

print(par_checker("lsdkjfskf"))
print(par_checker("radar"))
```

案例：滑动窗口

- 有一个长度为n的数的序列。序列上有一个大小为k的滑动窗口从序列最左端移动到最右端。窗口每次向右滑动一个数。请输出窗口在每个位置时窗口内的最大值和最小值。
- 输入：序列长度，窗口大小，序列。
- 示例：
- 8 3
- 1 3 -1 -3 5 3 6 7

窗口位置	最小值	最大值
[1 3 -1] -3 5 3 6 7	-1	3
1 [3 -1 -3] 5 3 6 7	-3	3
1 3 [-1 -3 5] 3 6 7	-3	5
1 3 -1 [-3 5 3] 6 7	-3	5
1 3 -1 -3 [5 3 6] 7	3	6
1 3 -1 -3 5 [3 6 7]	3	7

案例：滑动窗口

- 如果窗口每移动一次，就扫描整个窗口内的数找最大、最小值，相当于每次都要计算窗口中的最大值和最小值，时间复杂度较高，可能超时。
- 我们可以运用队列。用队列保存可能是答案的或者需要关注的项目。只有需要关注的才会出现在队列中，如果队列中的元素不需要，则将其移出队列。
- 维护两个双向单调序列，一个是最大值队列，其中的元素是递减的，另一个是最小值队列，其中的元素是递增的。最大值队列中存放的是可能成为最大值的数，最小值队列存放的是可能成为最小值的数。

案例：滑动窗口

- 关于最大值队列的操作：对于一个新的数number加入，由于其前面的数将会在number之前从窗口中被移出，因此若其前面的数的大小小于number，则无法成为最大值，因此可从队列中移除。可以从队列的尾端开始遍历，如果小于number直接出队列，然后将number入队列。
- 队列的头部永远是窗口内的最大值，如果队列头部元素在窗口移动的过程中正好被移出，则将队头出队列。（如何检测？可以在队列中存放元素的下标）
- 由于每个元素只会入队列一次，
- 最小值队列的操作同理。

滑动窗口移动	维持的队列 红色表示索引值 括号(x)里为对应数值
[1 3 -1] -3 5 3 6 7	[1(3) 2(-1)]
1 [3 -1 -3] 5 3 6 7	[1(3) 2(-1) 3(-3)]
1 3 [-1 -3 5] 3 6 7	[4(5)]
1 3 -1 [-3 5 3] 6 7	[4(5) 5(3)]
1 3 -1 -3 [5 3 6] 7	[6(6)]
1 3 -1 -3 5 [3 6 7]	[7(7)]

案例：滑动窗口

	滑动窗口移动	维持的队列 红色表示索引值 括号(x)里为对应数值
最大值队列	[1 3 -1] -3 5 3 6 7	[**1**(3) **2**(-1)]
	1 [3 -1 -3] 5 3 6 7	[**1**(3) **2**(-1) **3**(-3)]
	1 3 [-1 -3 5] 3 6 7	[**4**(5)]
	1 3 -1 [-3 5 3] 6 7	[**4**(5) **5**(3)]
	1 3 -1 -3 [5 3 6] 7	[**6**(6)]
	1 3 -1 -3 5 [3 6 7]	[**7**(7)]
最小值队列	[1 3 -1] -3 5 3 6 7	[**2**(-1)]
	1 [3 -1 -3] 5 3 6 7	[**3**(-3)]
	1 3 [-1 -3 5] 3 6 7	[**3**(-3) **4**(5)]
	1 3 -1 [-3 5 3] 6 7	[**3**(-3) **5**(3)]
	1 3 -1 -3 [5 3 6] 7	[**5**(3) **6**(6)]
	1 3 -1 -3 5 [3 6 7]	[**5**(3) **6**(6) **7**(7)]

案例：滑动窗口-代码

```
from collections import deque
n , k = map(int, input().split())
a = list(map(int, input().split()))
dqMax = deque()
dqMin = deque()
minA, maxA = [], []
for i in range(n):
    while dqMax and a[dqMax[-1]] < a[i]:
        dqMax.pop()
    dqMax.append(i)
    while dqMin and a[dqMin[-1]] > a[i]:
        dqMin.pop()
    dqMin.append(i)
    while dqMax[0] < i - k + 1:
        dqMax.popleft()
    while dqMin[0] < i - k + 1:
        dqMin.popleft()
    if i >= k - 1:
        maxA.append(a[dqMax[0]])
        minA.append(a[dqMin[0]])
for e in minA:
    print(e, end=' ')
print()
for e in maxA:
    print(e, end=' ')
print()
```

从队列的尾端让之后不再会用到的值出队列

如果队列头部已经在窗口之外，则队头出队列

此时滑动窗口：$[a_{i-k+1}, \ldots\ldots, a_i]$