COMPSCI 105 S1 2017
Principles of Computer Science

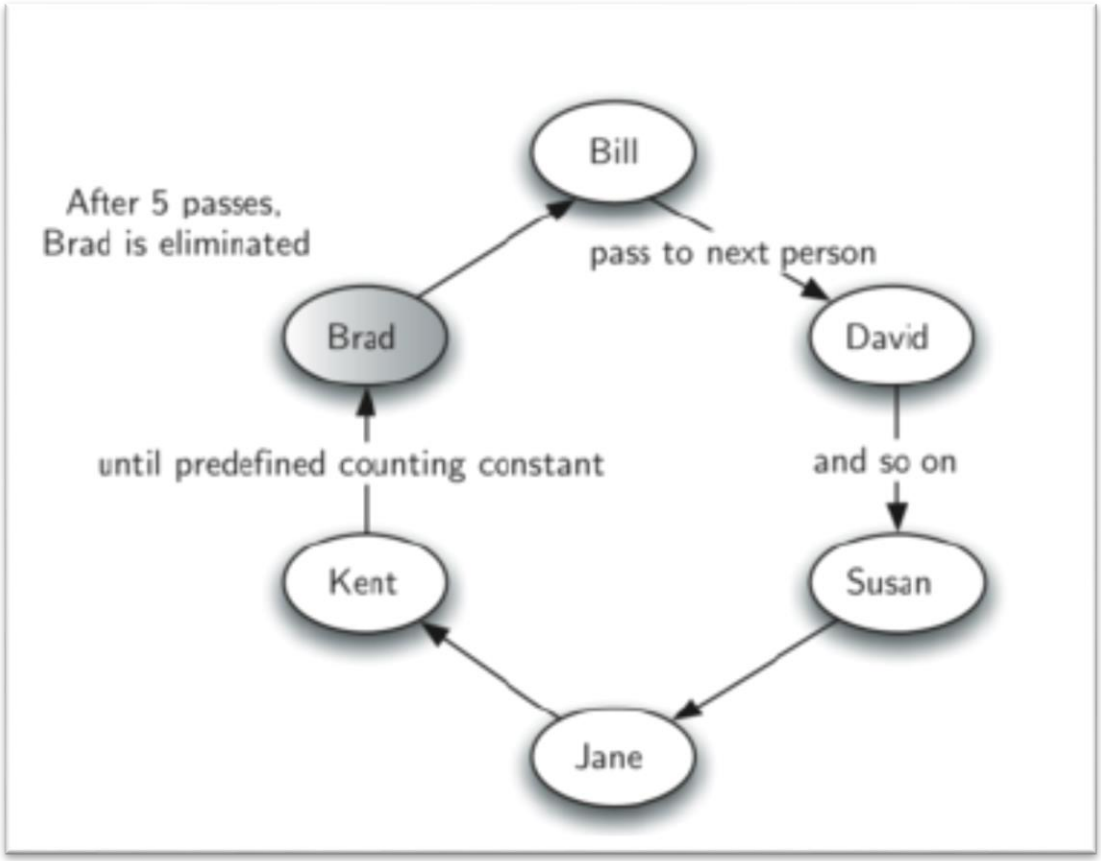
16 Queue(2)

Agenda & Readings

- Agenda
  - Using the Queue ADT to solve problems
  - A Circular Queue
  - The Deque Abstract Data Type
- Reference:
  - Textbook: Problem Solving with Algorithms and Data Structures
    - Chapter 3: Basic Data Structures

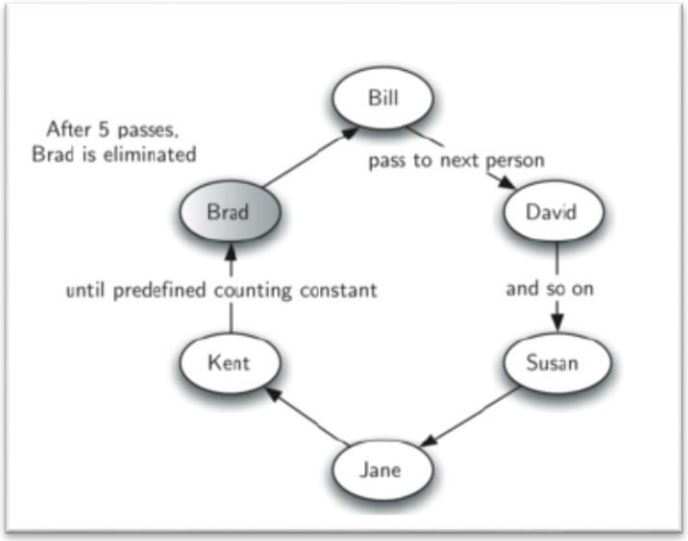
Simulation: Hot Potato

- Example (six persons game):



Simulation: Hot Potato

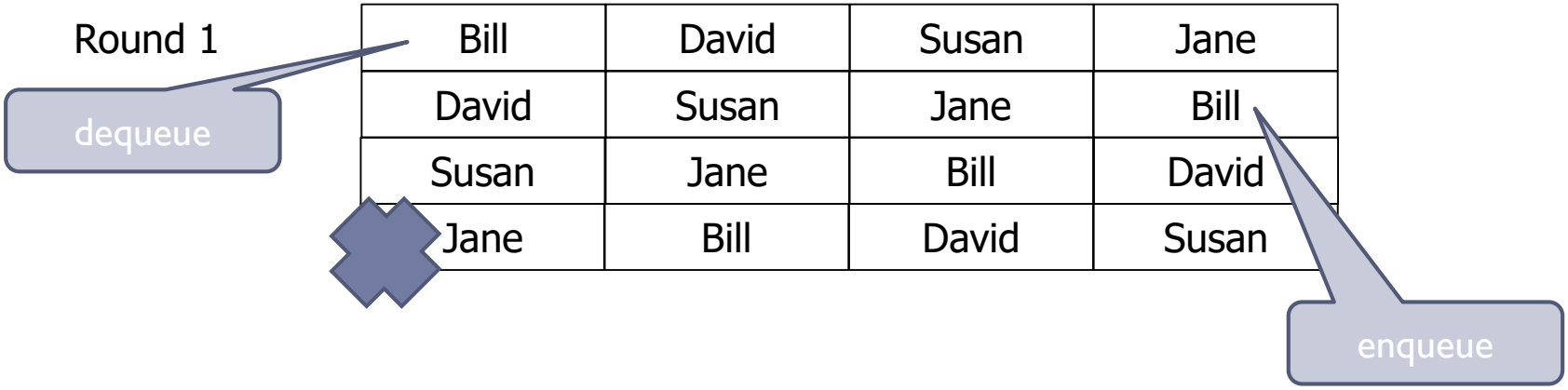
- Example (six persons game):
  - Children form a circle and pass an item from neighbour to neighbour as fast as they can
  - At a certain point in the game, the action is stopped and the child who has the item (the potato) is removed from the circle
  - Play continues until only one child is left



Simulation: Hot Potato

- Example (hotPotato([Bill, David, Susan, Jane], 3)):

Round 1



Bill	David	Susan	Jane
David	Susan	Jane	Bill
Susan	Jane	Bill	David
Jane	Bill	David	Susan

Round 2

Bill	David	Susan
David	Susan	Bill
Susan	Bill	David
Bill	David	Susan

Simulation: Hot Potato

- Example (hotPotato([Bill, David, Susan, Jane], 3)):

Round 3

David	Susan
Susan	David
David	Susan
Susan	David

Final

David

WIN!

Simulation: Hot Potato

- Code:

```
def hotPotato(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()
```

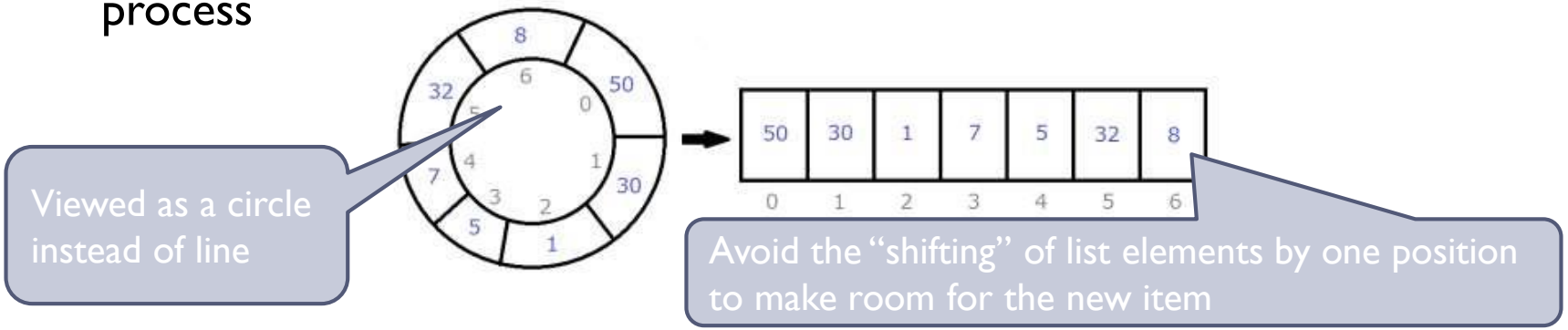
Move element from the front of the queue to the end

Return the name when there is only ONE name remains in the queue

Circular Queue

- What is the Big-O performance of enqueue and dequeue of the implementation using Python List?
  - enqueue(…): O(n)
    - Shifting array elements to the right after each addition – too Expensive!
  - dequeue() : O(1)
- Another Implementation: Circular Queue
  - enqueue & dequeue : O(1)
    - Items can be added/removed without shifting the other items in the process

We have to shift all list elements by one position to make room for the new item



Circular Queue - Set up

- Uses a Python list data structure to store the items in the queue
- There are three critical variables:
  - front: indicates the location of the item at the front
  - back: indicates the location of the item at the back
  - count: indicates the number of items in the queue
- The list has an initial capacity (all elements None)

Circular Queue - Set up

- Keeps an index of the current front of the queue and of the current back of the queue
  - set front to 0
  - set back to MAX_QUEUE – 1
  - set count to 0

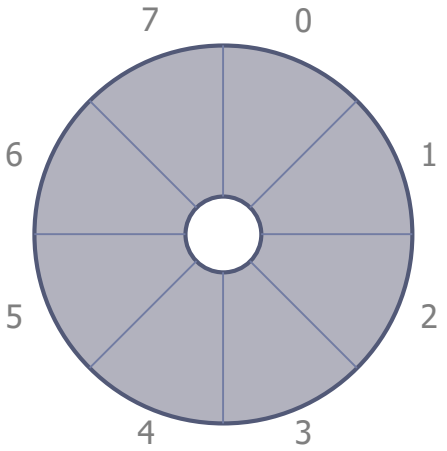
To initialize the queue

- New items are **enqueued** at the back index position
- Items are **dequeued** at the front index position.
- A **counting** of the queue items to detect queue-full and queue-empty conditions

Circular Queue - How To Advance

- Queue-empty:
  - front is one slot ahead of back
- When either front or back advances past MAX_QUEUE - 1, it wraps around to 0
  - The wrap-around effect: by using Modulus (%) arithmetic operator

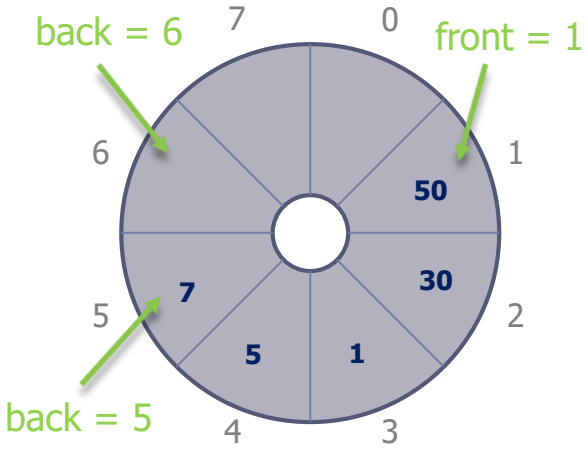
```
def enqueue(self, item): # if not full
    self.back = (self.back + 1) % self.MAX_QUEUE
    self.items[self.back] = item
    self.count += 1
def dequeue(self): # if not empty
    item = self.items[self.front]
    self.front = (self.front + 1) % self.MAX_QUEUE
    self.count -= 1
    return item
```



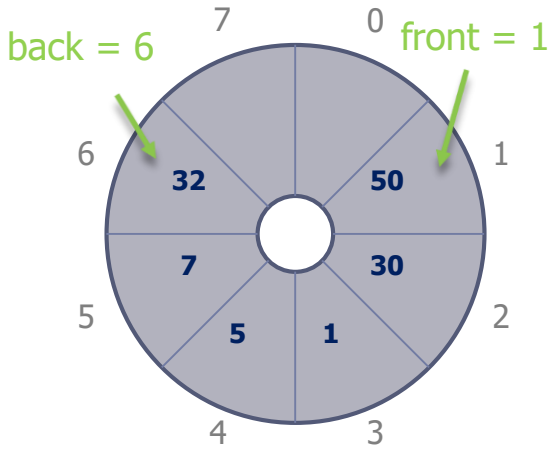
Enqueue

```
def enqueue(self, item): # if not full
    self.back = (self.back + 1) % self.MAX_QUEUE
    self.items[self.back] = item
    self.count += 1
```

- Example:
  - q.enqueue(32)
    - back is advanced by one position
    - New item is inserted at the position of back
    - count is incremented by 1



size = 8; count = 5

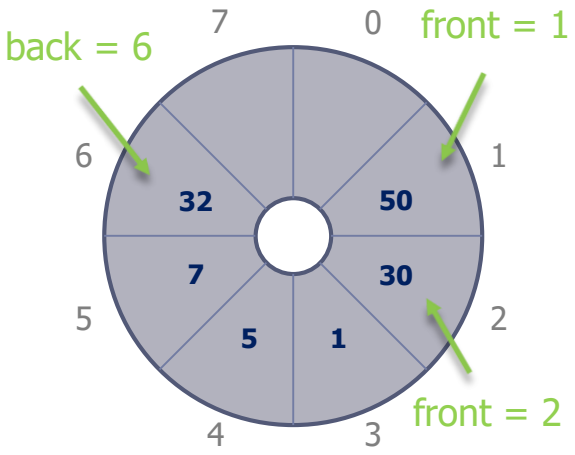


size = 8; count = 6

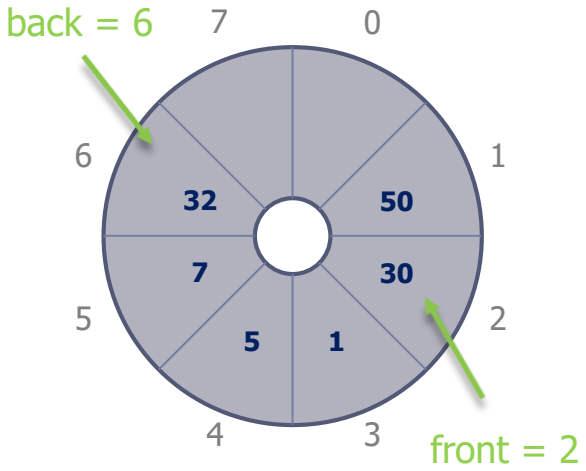
Dequeue

```
def dequeue(self): # if not empty
    item = self.items[self.front]
    self.front = (self.front + 1) % self.MAX_QUEUE
    self.count -= 1
    return item
```

- Example:
  - q.dequeue()
    - Value in front position is returned
    - front is advanced by 1
    - count is decremented by 1



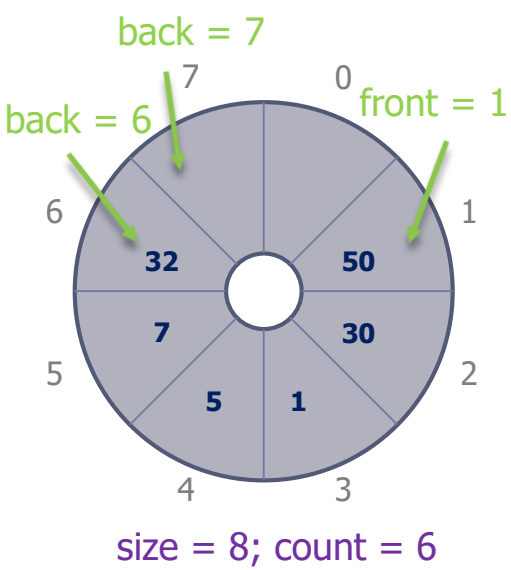
size = 8; count = 6



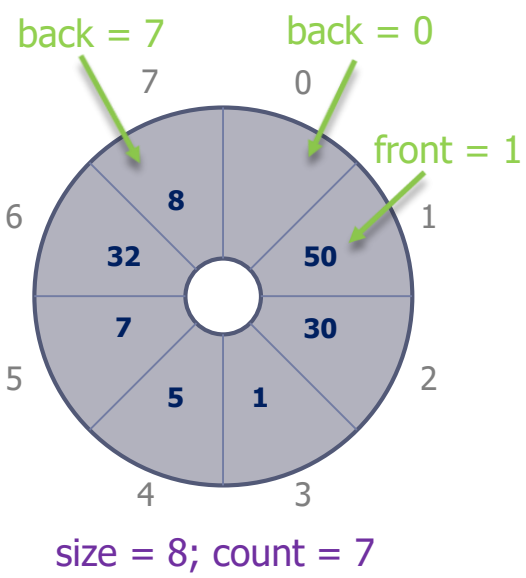
size = 8; count = 5

Enqueue

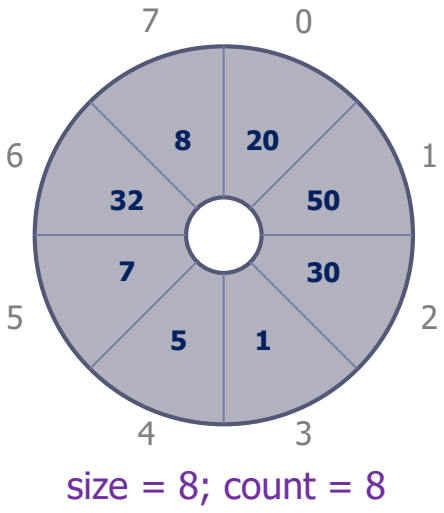
- q.enqueue(8)
  - After running the first enqueue, back = 7
- q.enqueue(20)
  - After running the second enqueue, back = 0 as the "back" is wrapped around the list



size = 8; count = 6



size = 8; count = 7



size = 8; count = 8

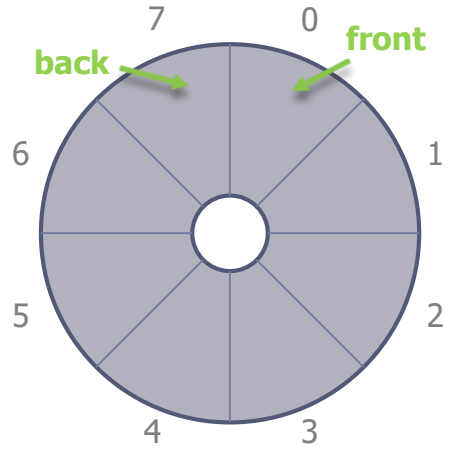
Full & Empty

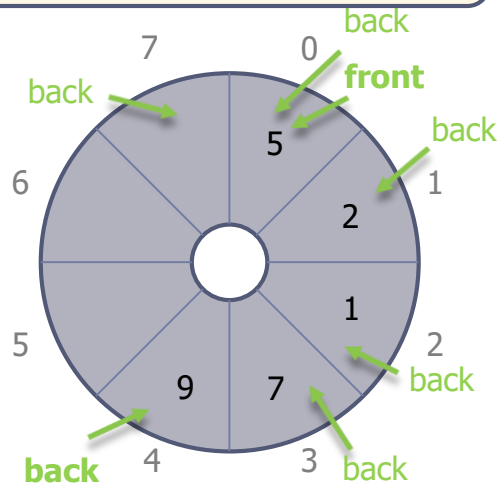
- front and back cannot be used to distinguish between queue-full and queue-empty conditions for a circular array

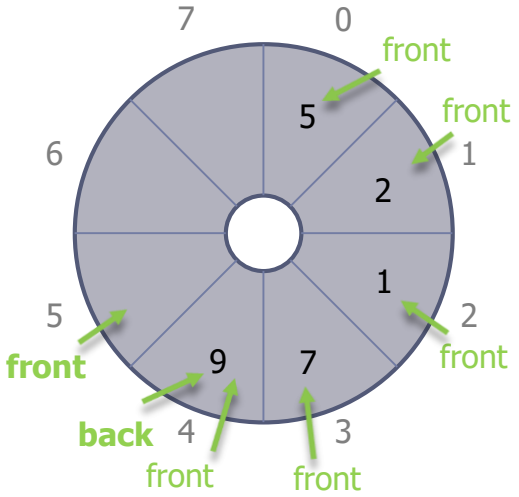
```
q = Queuecircular(8)
```

```
q.enqueue(5)
q.enqueue(2)
q.enqueue(1)
q.enqueue(7)
q.enqueue(9)
```

```
q.dequeue()
q.dequeue()
q.dequeue()
q.dequeue()
q.dequeue()
```





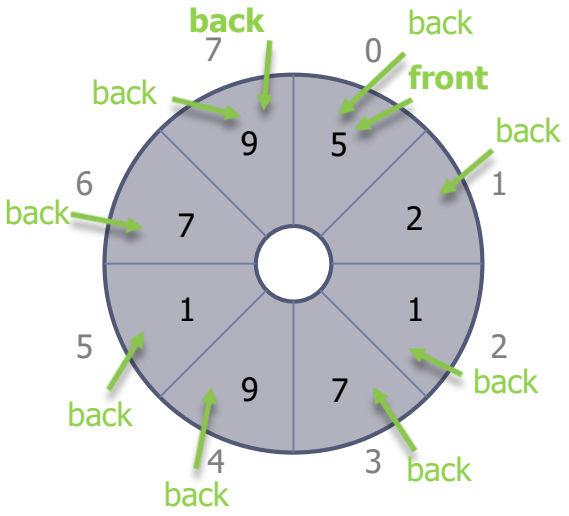


back = front - 1

Full & Empty

- front and back cannot be used to distinguish between queue-full and queue-empty conditions for a circular array

```
q = Queuecircular(8)
q.enqueue(5)
q.enqueue(2)
q.enqueue(1)
q.enqueue(7)
q.enqueue(9)
q.enqueue(1)
q.enqueue(7)
q.enqueue(9)
```




```
def is_empty():
    return self.count == 0
```

```
def is_full():
    return self.MAX_QUEUE <= self.count
```

Exercise 1

- What are the values of "front" and "back" after executing the following code fragment?

```
q = Queuecircular(10)
q.enqueue(12)
q.enqueue(17)
q.enqueue(25)
q.enqueue(11)
q.dequeue()
q.dequeue()
q.enqueue(30)
```

Deque Abstract Data Type

- Deque - Double Ended Queue
  - A deque is an ordered collection of items where items are added and removed from either end, either front or back
- The newest item is at one of the ends

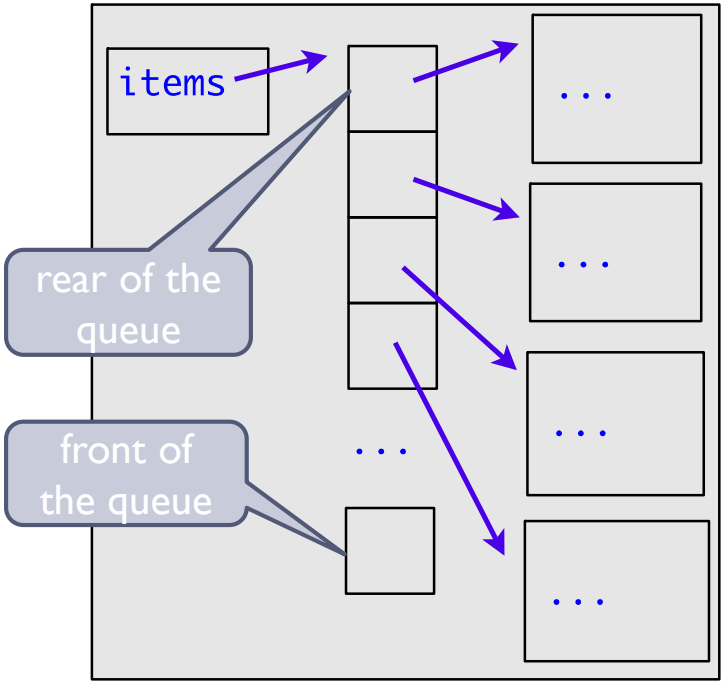
Deque Abstract Data Type

- What are the operations which can be used with a Deque Abstract Data?
- Create an empty deque:
- Determine whether a deque is empty:
- Add a new item to the deque:
  - add_front()
  - add_rear()
- Remove from the deque the item that was added earliest:
  - remove_front()
  - remove_rear()

Code Example

- We use a python List data structure to implement the deque

```
class Deque:
    def __init__(self):
        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)
```



Big-O?
add_front()/remove_front(): :O(1)
add_rear()/remove_rear(): O(n)

Code Example

- Code:

```
d.is_empty()
d.add_rear(4)
d.add_rear('dog')
d.add_front('cat')
d.add_front(True)
d.size()
d.is_empty()
d.add_rear(8.4)
d.remove_rear()
d.remove_front()
```

```
d = []
d = [4]
d = ['dog', 4]
d = ['dog', 4, 'cat']
d = ['dog', 4, 'cat', True]
d = ['dog', 4, 'cat', True]
d = ['dog', 4, 'cat', True]
d = [8.4, 'dog', 4, 'cat', True]
d = ['dog', 4, 'cat', True]
d = ['dog', 4, 'cat']
```

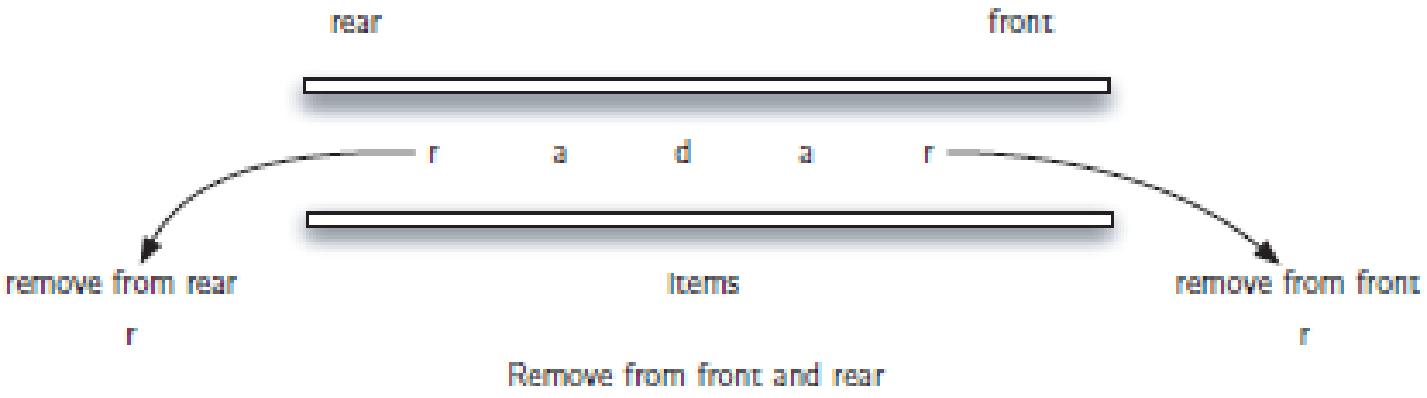
```
True


4
False

8.4
True
```

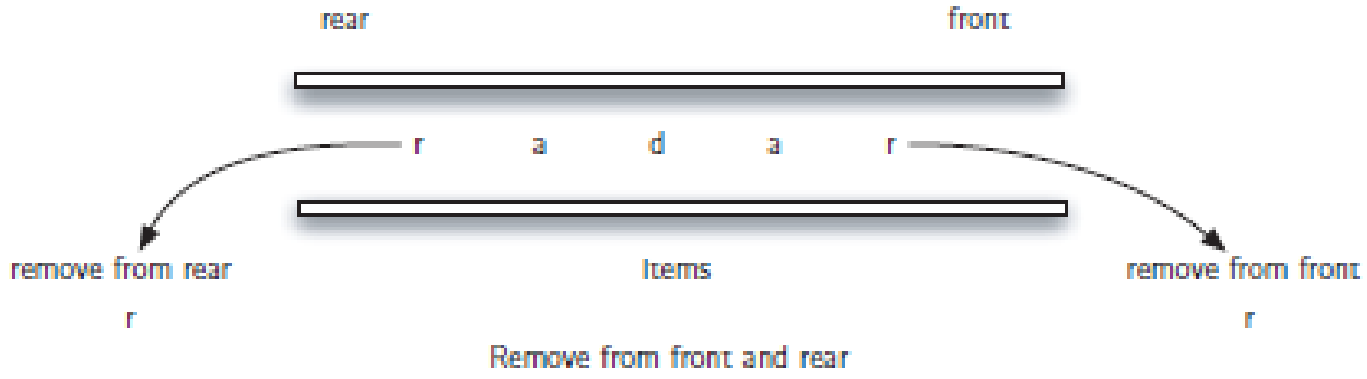
Application: Palindrome Checker

- A string which reads the same either left to right, or right to left is known as a palindrome
  - Radar
  - deed
  - A dog, a plan, a canal: pagoda



Palindrome Checker - Algorithm

- Create a deque to store the characters of the string
  - The front of the deque will hold the **first** character of the string and the rear of the deque will hold the **last** character
- Remove both of them directly, we can compare them and continue only if they match
  - If we can keep matching first and the last items, we will eventually either **run out of characters** or be left with **a deque of size 1**
    - In either case, the string must be a palindrome



Palindrome Checker - Examples

- print(pal_checker("lsdkjfskf"))
  - Queue: f, k, s, f, j, k, d, s, l
  - 1st round: compare f and l => FALSE, STOP
- print(pal_checker("radar"))
  - Queue: r, a, d, a, r
  - 1st round: compare r (front) and r (back)
  - 2nd round: compare a (front) and a (back)
  - 3rd round: size() = 1, STOP, return TRUE

Palindrome Checker - Codes

- Check:
  - The front of the deque (the first character of the string)
  - The rear of the deque (the last character of the string)

```
still_equal = True
    while char_deque.size() > 1 and still_equal:
        first = char_deque.remove_front()
        last = char_deque.remove_rear()
        if first != last:
            still_equal = False
    return still_equal
```

Summary

- To distinguish between the queue-full and queue-empty conditions in a queue implementation that uses a circular array
  - By counting the number of items in the queue
- Models of real-world systems often use queues