Queue ADT

IT5003: Data Structures and Algorithms
(**AY2019/20 Semester 1**)

Lecture Overview

- Queue
  - Introduction
  - Specification
  - Implementations
    - Linked List Based
    - Array Based
  - Application
    - Palindrome checking

What is a Queue

- Real life example:
  - A queue for movie tickets, Airline reservation queue, etc
- First item added will be the first item to be removed
  - Has the First In First Out (**FIFO**) property
- Major Operations:
  - **Enqueue:** Items are added to the **back of the queue**
  - **Dequeue:** Items are removed from the **front of the queue**
  - **Get Front:** Take a look at the first item

Queue: Illustration

A **queue** of 3 persons

Enqueue a new person to the **back of the queue**

Dequeue a person from the **front of the queue**

Queue ADT: Python Specification

```
class QueueBase(ABC):
    @abstractmethod
    def getFront(self):
        pass

    @abstractmethod
    def enqueue(self, newItem):
        pass

    @abstractmethod
    def dequeue(self):
        pass

    @abstractmethod
    def size(self):
        pass

    @abstractmethod
    def isEmpty(self):
        pass
```

Major Operations for Queue

Design Considerations

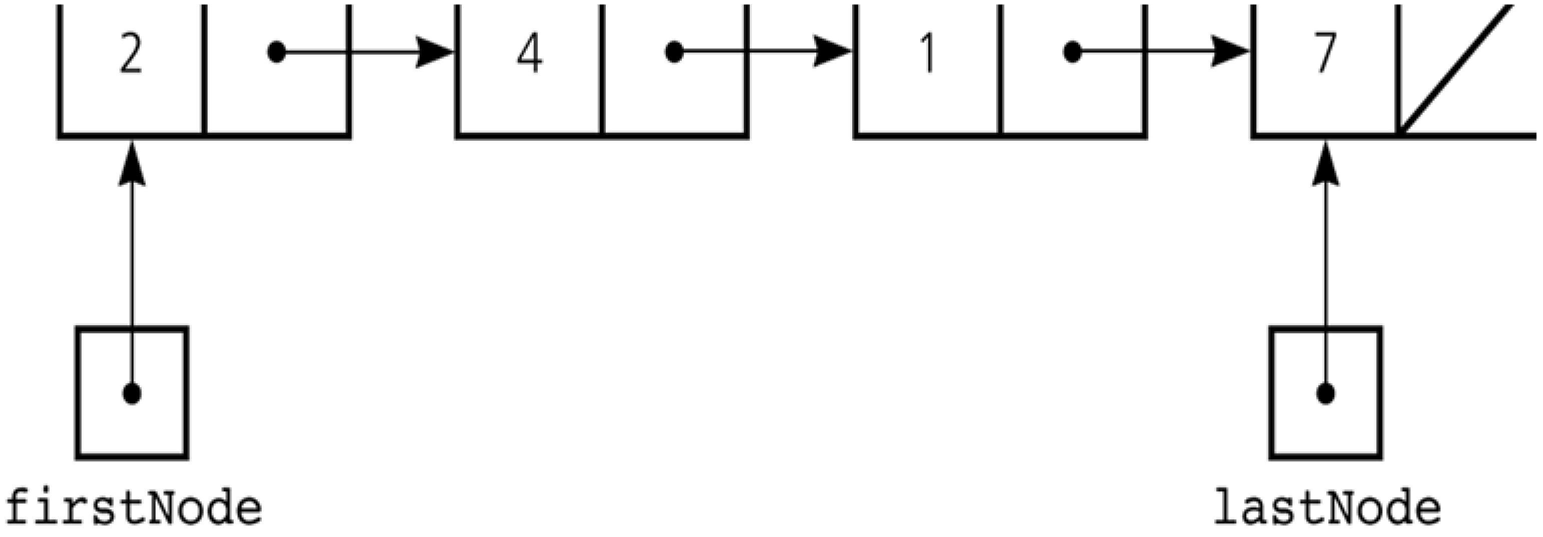
- How about the common coding choices?
  - Efficiency of **singly linked list implementation**:
    - 👍 Removing item at the head is the best case
    - 👎 Adding item at the back is the worst case
  - Efficiency of **array based implementation**:
    - 👎 Removing item at the head is the worst case
    - 👍 Adding item at the back is the best case
- Is it possible to have both efficient **`enqueue`**`()` and **`dequeue`**`()` operations?

QUEUE ADT
USING LINKED LIST VARIANT

Improving the Singly Linked List

- Singly linked list performs badly for `enqueue()`
  - Need to traverse all the way to the last node
  - Takes longer time as the queue grows
- How to avoid the traversal to the last node?
  - Just need to “know” where is the last node all the time……
- Solutions:
  - Keep an additional reference to the last node, OR
  - Circular linked list with a tail reference

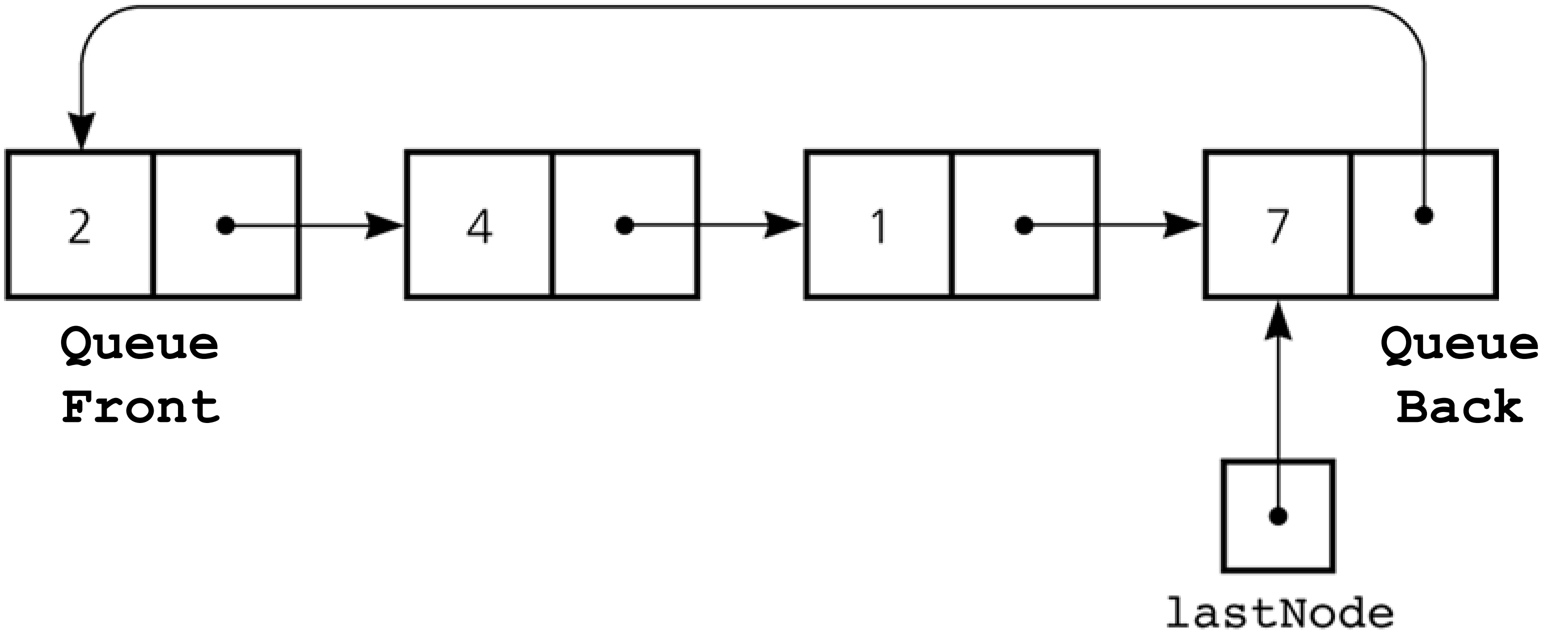
Linked List : head and tail



Queue Front

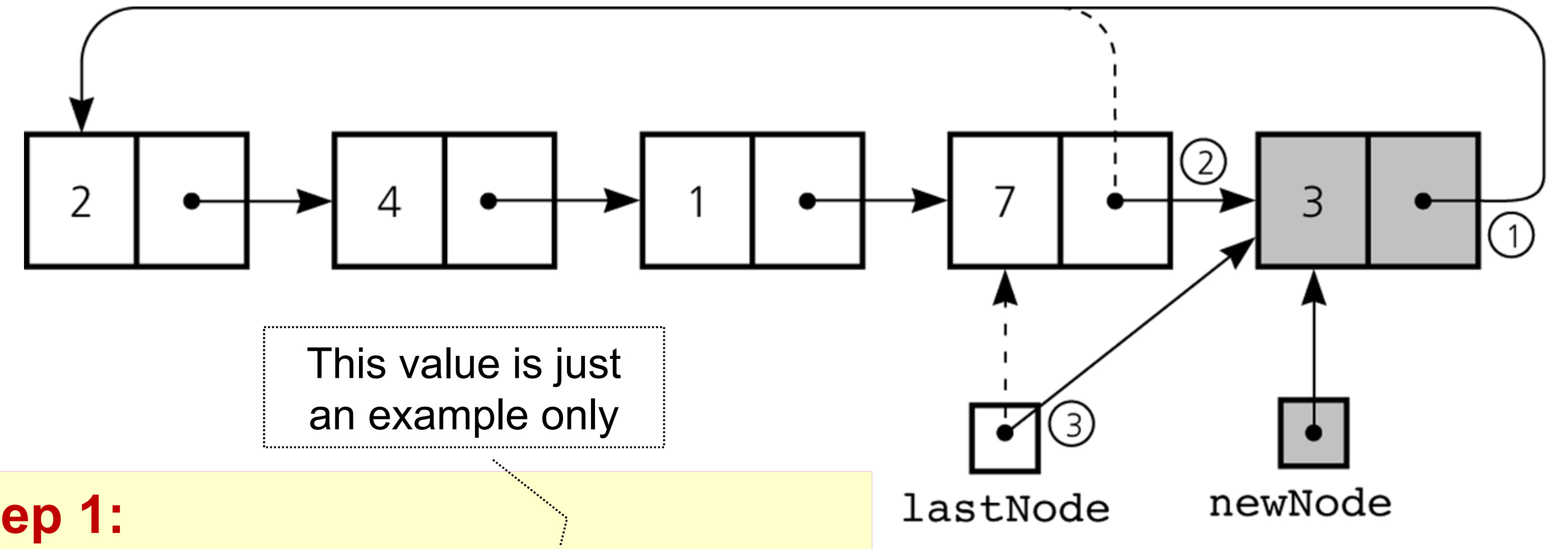
Queue Back

Circular Linked List



- Only keep tracks of **`lastNode`** reference
  - **`firstNode`** reference can be set when needed:
    - **`firstNode = lastNode.next`**
- We use circular linked list in our implementation

Enqueue: Non-Empty Queue



This value is just an example only

Step 1:

```
newNode = SinglyNode(3)
newNode.next = _lastNode.next
```

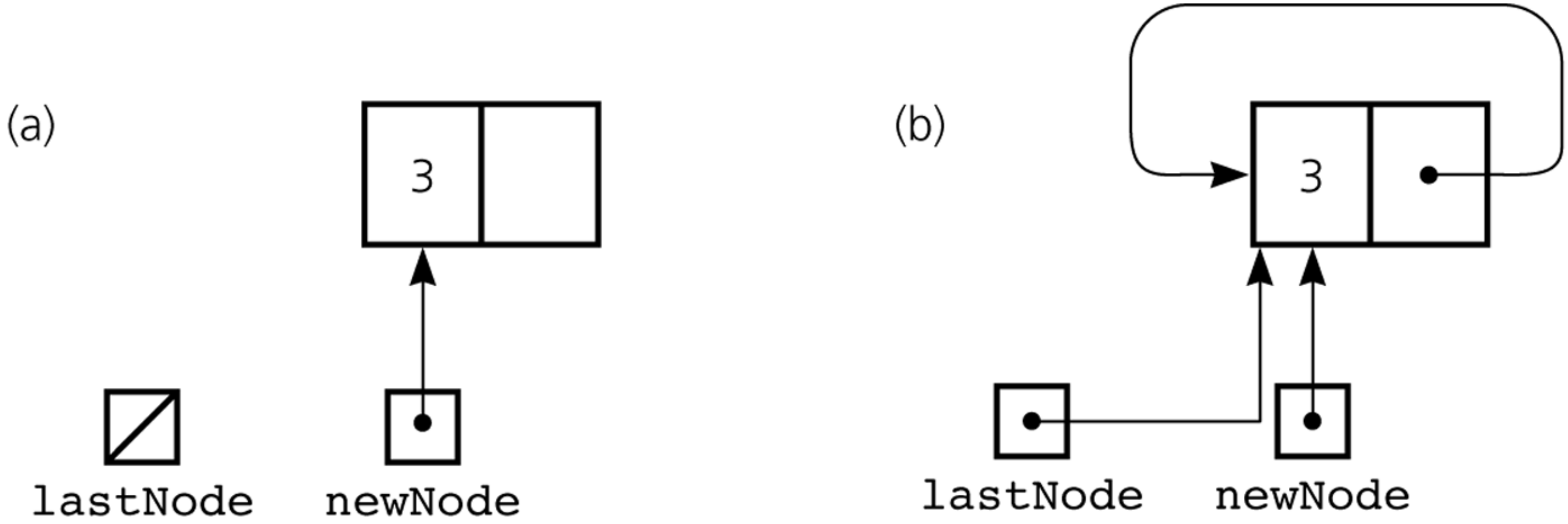
Step 2:

```
self._lastNode.next = newNode
```

Step 3:

```
_lastNode = newNode
```

Enqueue: Empty Queue



Step (a):

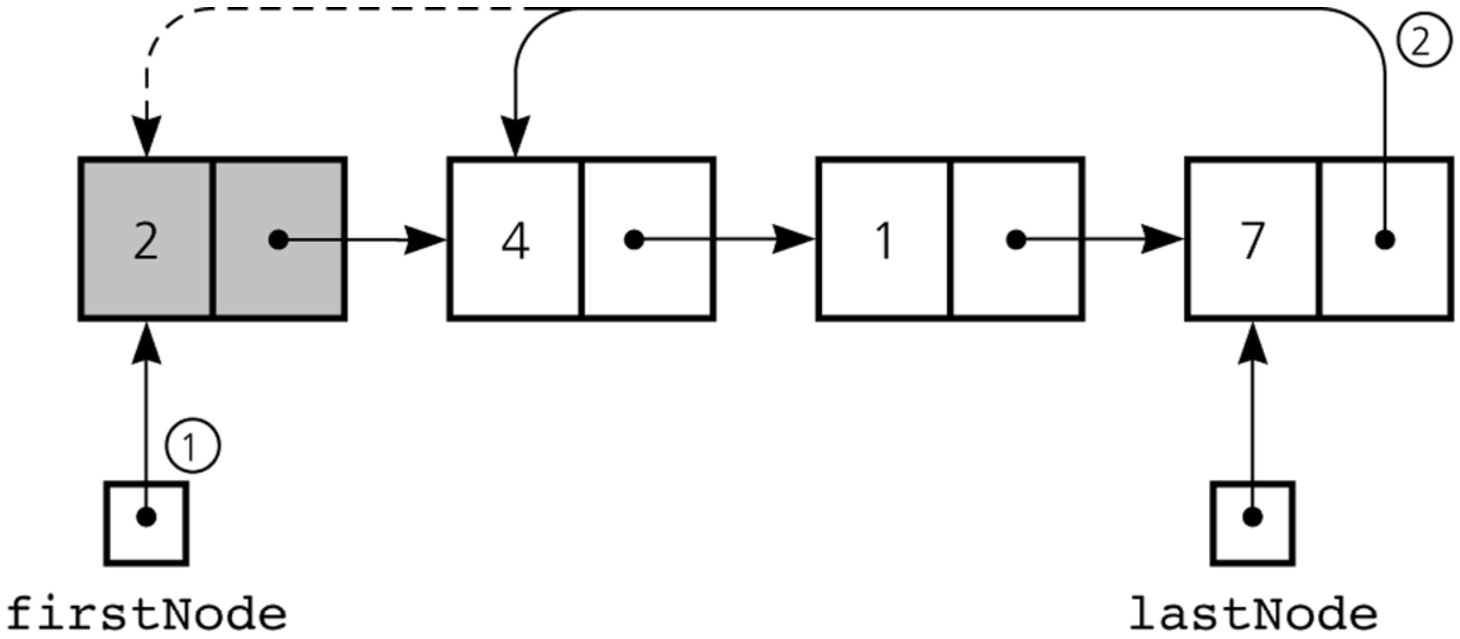
```
newNode = SinglyNode(3)
```

Step (b):

```
newNode.next = newNode
self._lastNode = newNode
```

Set up the “loop”

Dequeue: Queue with > 1 item



Step 1:

```
firstNode = lastNode.next
```

Step 2:

```
lastNode.next = firstNode.next
```

QueueLinkedList: Implementation

```python
class QueueLinkedList(QueueBase):

   def __init__(self):
      self._lastNode = None
      self._size = 0


   def getFront(self):
      if not self.isEmpty():
         return self._lastNode.next.item
      else:
         return None
```

QueueLinkedList.py

QueueLinkedList: Implementation

```
def enqueue(self, newItem):
    newNode = SinglyNode(newItem)

    if self._lastNode == None:
        newNode.next = newNode
        self._lastNode = newNode

    else:
        newNode.next = self._lastNode.next
        self._lastNode.next = newNode
        self._lastNode = newNode

    self._size += 1
    return True
```

Refer to the algorithm discussion slides

QueueLinkedList: Implementation

```
    def dequeue(self):
        if not self.isEmpty():
            firstNode = self._lastNode.next
            self._lastNode.next = firstNode.next

            if self._size == 1:
                self._lastNode = None

            self._size -= 1
            return True
        else:
            return False

    def size(self):
        return self._size

    def isEmpty(self):
        return self._size == 0
```

Refer to the algorithm discussion slides

Take note of the 1 node special case

What?! Array can be circular?

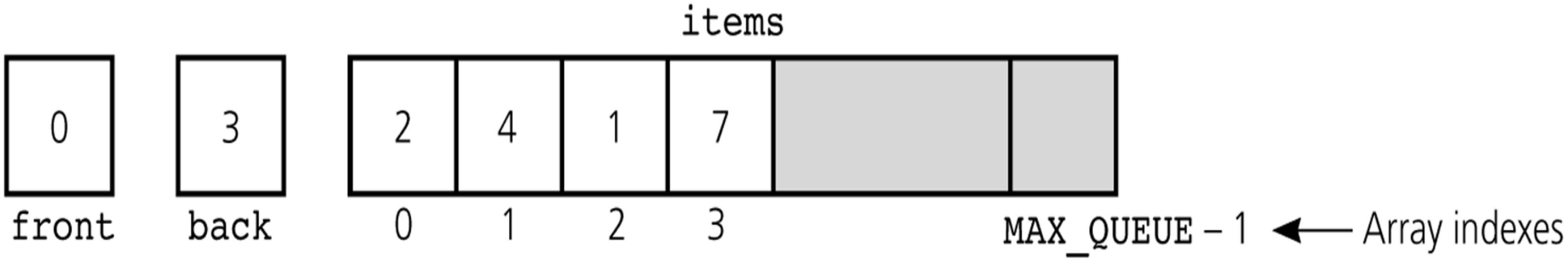
QUEUE ADT
USING CIRCULAR ARRAY

Array Implementation Issues

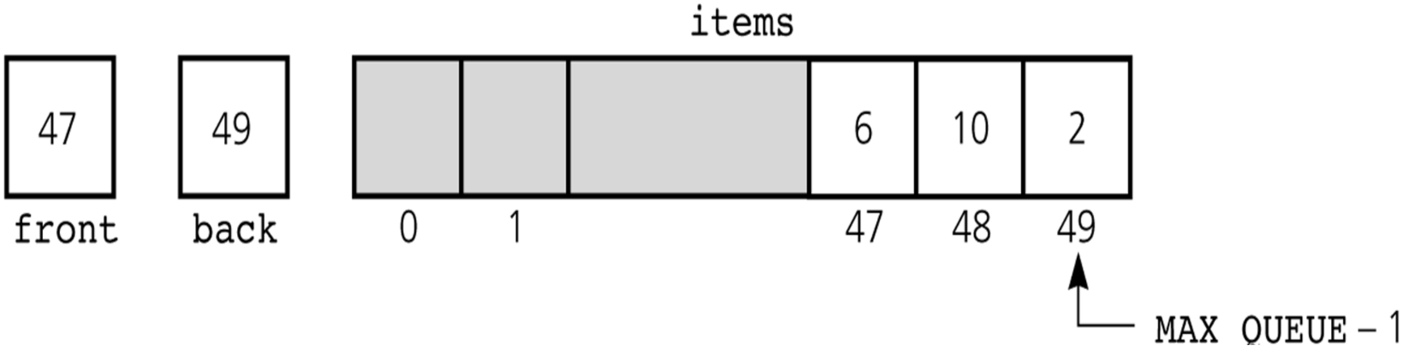
- Removing item from the front is inefficient
  - **Shifting items is too expensive**
- Basic Idea:
  - The reason for shifting is:
    - Queue's **front** is assumed to be at **index 0**
  - Instead of shifting items:
    - Why don’t we just ***shift* the front index**
- So, we have two indices:
  - **Front:** index of the queue front
  - **Back:** index of the queue back

Incorrect Implementation

- At the beginning, with 4 items queued

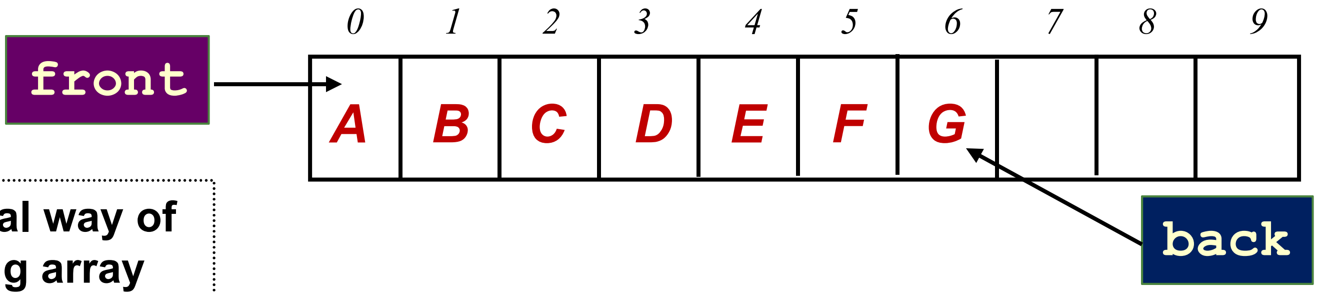


- After many queue operations



- The front index will drift to the right:
  - Most array locations empty and unusable

Circular Array: Illustration

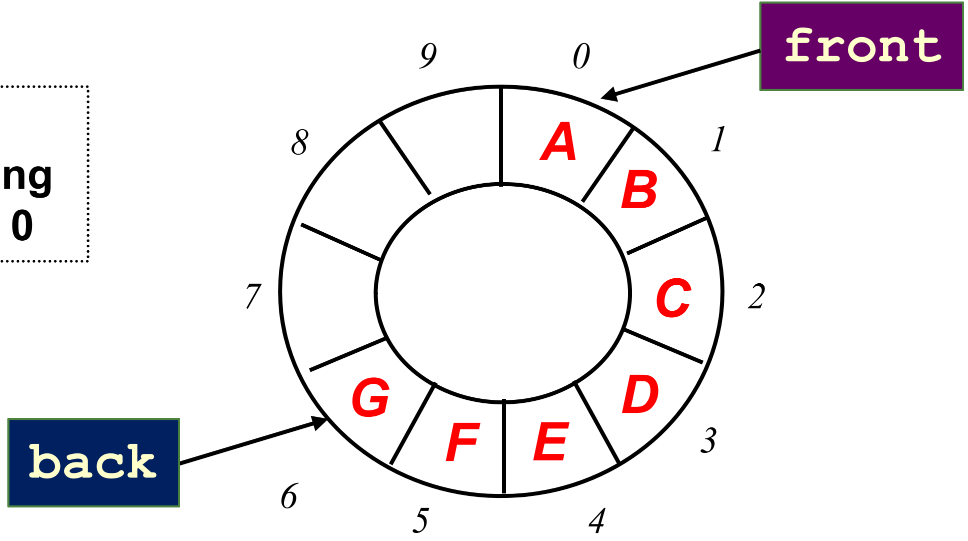


Normal way of using array

The "magic" ingredients

```
front = (front+1) % maxsize
back  = (back+1)  % maxsize
```

Make the array "circular" by wrapping the indices back to 0



QueueCircularArray: Implementation

```
class QueueCircularArray(QueueBase):
    MAXSIZE = 50 #arbitrarily chosen as example


    def __init__(self):
        self._items \
        = (QueueCircularArray.MAXSIZE * ctypes.py_object)()

        self._size = 0

        self._front = 0
        self._back = QueueCircularArray.MAXSIZE -1
```

MAXSIZE is a *class attribute*

Use static array to reduce "syntax burden".
Can be extended to dynamic array

`_front` : array index of the queue front

`_back` : array index of the queue back

`QueueCircularArray.py`

QueueCircularArray: Implementation

```
    def enqueue(self, newItem):
        if self._size == QueueCircularArray.MAXSIZE:
            return False
        else:
            self._back \
            = (self._back + 1 ) % QueueCircularArray.MAXSIZE
        self._items[self._back] = newItem
        self._size += 1

        return True

    def dequeue(self):
        if not self.isEmpty():
            self._front \
            = ( self._front+1 ) % QueueCircularArray.MAXSIZE
            self._size -= 1
            return True
        else:
            return False
```

Focus on the update of the _front and _back indices

Only selected methods implementation are shown. As you should be able to code the rest easily by now ☺.

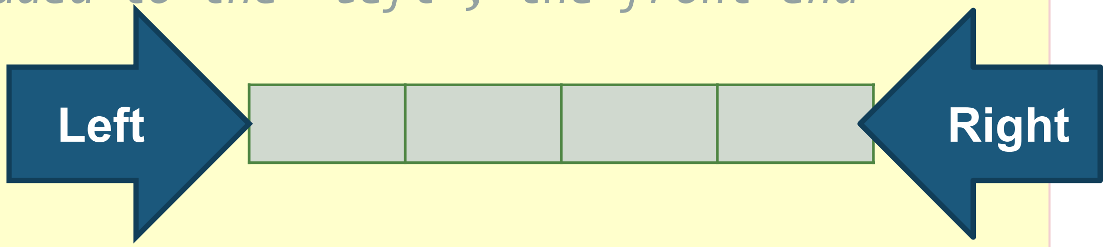
Queue has a standard implementation

PYTHON BUILT-IN QUEUE

Python Collections: **deque**

```
from collections import deque

def main():
    dq = deque()

    #Items can be added both ends
    dq.append(111) #added to the "right", the tail end
    dq.append(122)

    dq.appendleft(999) #added to the "left", the front end
    dq.appendleft(988)

    print(dq)

    #Items can be removed from both ends

    dq.pop() #removed from the right end
    dq.popleft() #removed from the left end
    print(dq)
```

Essentially a container that can be accessed from both ends: Left and Right. So, can be used as stack or queue!



Python-Deque-Demo.py

"Queue" Application

PALINDROME CHECKING

Palindrome : **Problem Description**

- **Palindrome** is a string which reads the same either *left to right*, or *right to left*
  - **Palindromes:** “radar” and “deed”
  - **Counter Example:** “data”

- Many solutions! We chose to:
  a. Use *stack* to reverse the input
  b. Use *queue* to preserve the input
  - The two sequence should be the same for palindrome
  - Also demonstrate the LIFO and FIFO property

Palindrome: Implementation

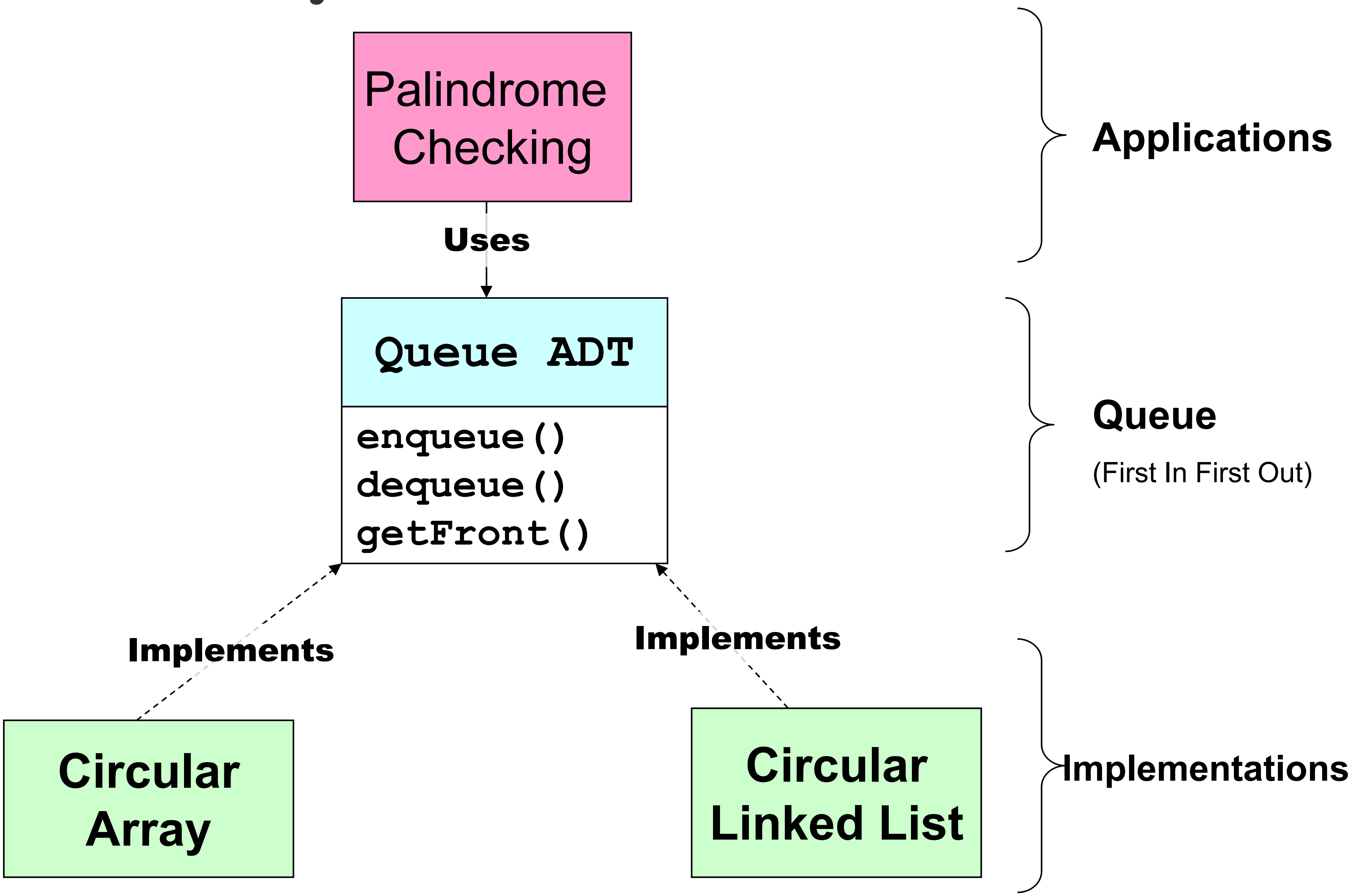
```
def isPalindrome( input ):
    s = StackList()
    q = QueueLinkedList()

    for ch in input:
        s.push(ch)
        q.enqueue(ch)

    while not s.isEmpty():
        if s.getTop() != q.getFront():
            return False

        s.pop()
        q.dequeue()

    return True
```

Any Stack / Queue is fine!

Push the same character into both queue and stack

Queue has the original sequence, Stack has the reversed. Compare to make sure they are the same

Summary



Palindrome Checking

Uses

Queue ADT

```
enqueue()
dequeue()
getFront()
```

Implements

Implements

Circular Array

Circular Linked List

Applications

Queue

(First In First Out)

Implementations

END