파이썬으로 배우는 데이터 구조

한동대학교
전산전자공학부
김영섭 교수

학습 목표

Circular Queue로 Queue의 한계점을
어떻게 보완하는지 이해하고 구현할 수 있다

Data Structures in Python
Chapter 3 - 2

- Queue
- Deque
- Deque Profiling
- **Circular Queue**

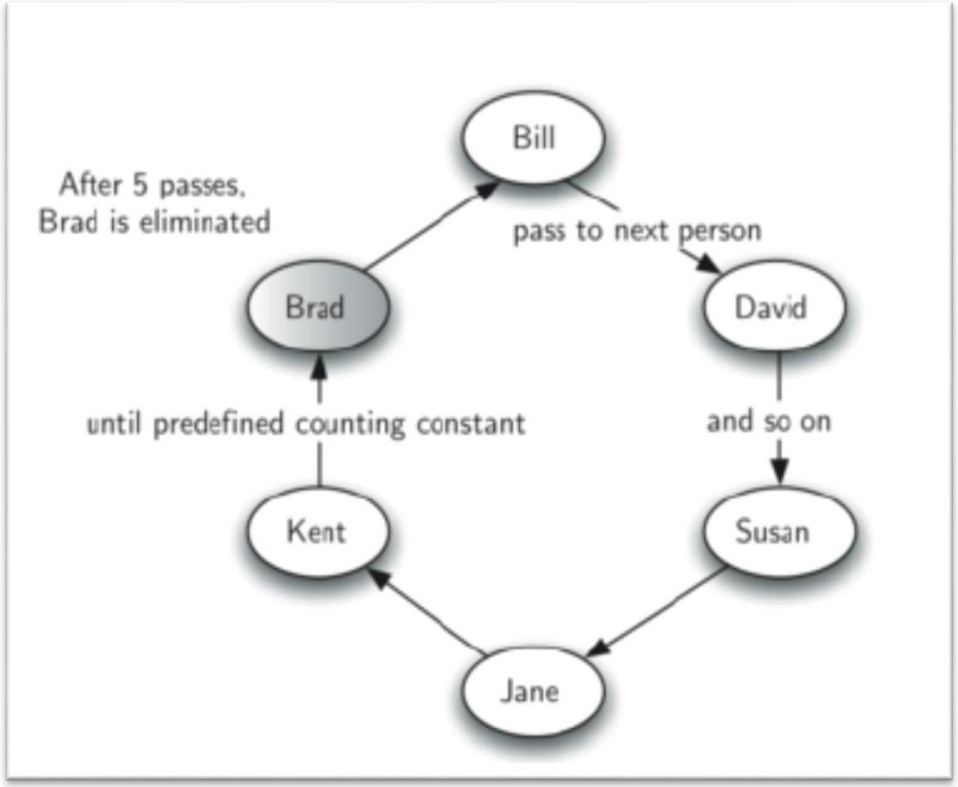
Agenda & Readings

- Agenda
  - Using the Queue ADT to solve problems
  - The Deque Abstract Data Type
  - **A Circular Queue**

Can you think of other examples of queues?

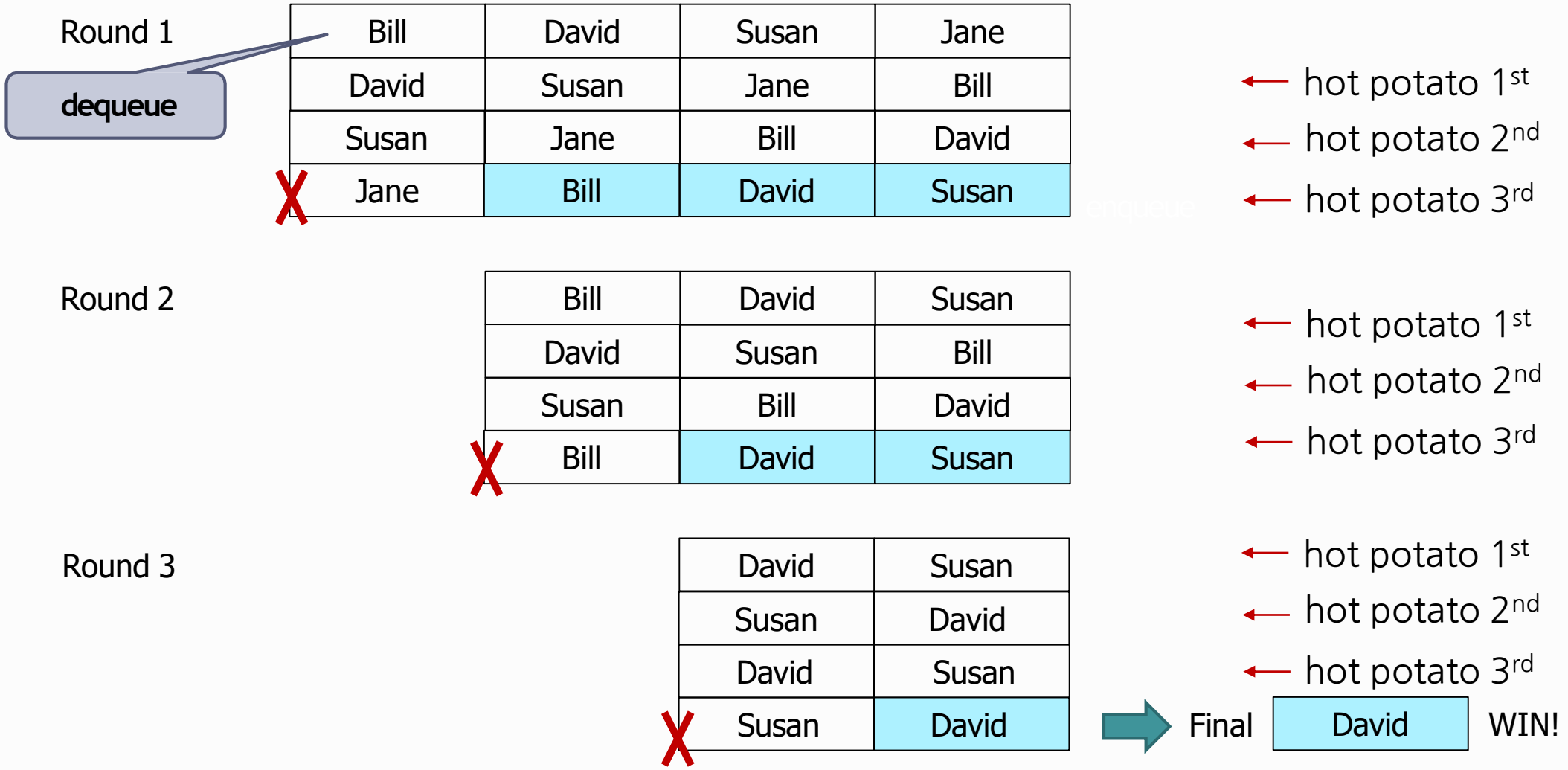
Queue Simulation: Hot Potato

- Example (six persons game):
  - Children form a circle and pass an item from neighbor to neighbor 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.



Queue Simulation: Hot Potato

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



Exercise: Simulation for Hot Potato

- This code is supposed to be a simulation of Hot Potato game, but some indents are removed purposely, and one line has a bug (missing a code). Debug the code.

```
#%%writefile hotPotato.py
from queue import Queue

def hotPotato(namelist, num):
    que = Queue()
    for name in namelist:
    que.put(name)
    assert len(namelist) == que.qsize()

    while que.qsize() > 1:
        for i in range(num):
        que.put()
        que.get()
    assert que.qsize() == 1
    return que.get()

if __name__ == "__main__":
    namelist = [ 'Bill', 'David', 'Susan', 'Jane' ]
    print("The winner is", hotPotato(namelist, 3))
```

Exercise: Simulation for Hot Potato

- This code is supposed to be a simulation of Hot Potato game, but some indents are removed purposely, and one line has a bug (missing a code). Debug the code.

```
#%%writefile hotPotato.py
from queue import Queue

def hotPotato(namelist, num):
    que = Queue()
    for name in namelist:
    que.put(name)
    assert len(namelist) == que.qsize()

    while que.qsize() > 1:
    for i in range(num):
    que.put()
    que.get()
    assert que.qsize() == 1
    return que.get()

    if __name__ == "__main__":
    namelist = [ 'Bill', 'David', 'Susan', 'Jane' ]
    print("The winner is", hotPotato(namelist, 3))
```

```
PS C:\GitHub\DSpyx\jupyter> python hotPotato.py
The winner is David
PS C:\GitHub\DSpyx\jupyter>
```

Since Python does not use conventional terminology in its Queue implementation, you may read its reference manual.

Exercise: Simulation for Hot Potato Hints

- This code is supposed to be a simulation of Hot Potato game, but some indents are removed purposely, and one line has a bug (missing a code). Debug the code.

```
#%%writefile hotPotato.py
from queue import Queue

def hotPotato(namelist, num):
    que = Queue()
    for name in namelist:
    que.put(name)
    assert len(namelist) == que.qsize()

    while que.qsize() > 1:
    for i in range(num):
    que.put()
    que.get()
    assert que.qsize() == 1
    return que.get()

    if __name__ == "__main__":
    namelist = [ 'Bill', 'David', 'Susan', 'Jane' ]
    print("The winner is", hotPotato(namelist, 3))
```

```
PS C:\GitHub\DSpyx\jupyter> python hotPotato.py
The winner is David
PS C:\GitHub\DSpyx\jupyter>
```

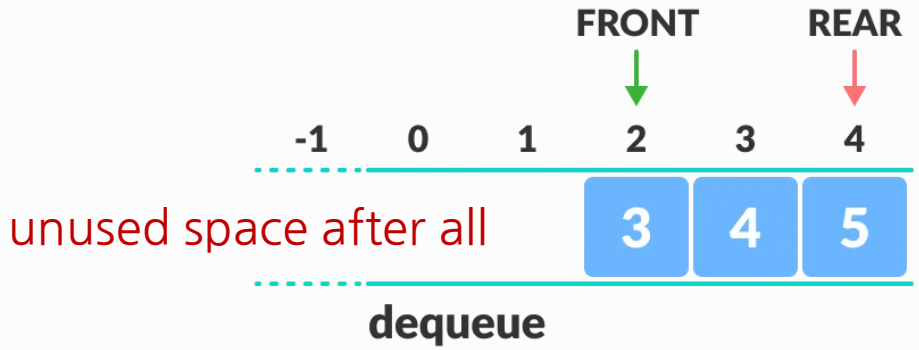
Move 'num' elements from the front of the queue to the end

Remove one name after num=3 enqueues/hot-potato

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

Circular Queue

- In a normal queue, after a bit of insertion and deletion, there will be non-usable empty space.
- The circular queue solves the major limitation of the normal queue.



Limitation of the regular Queue

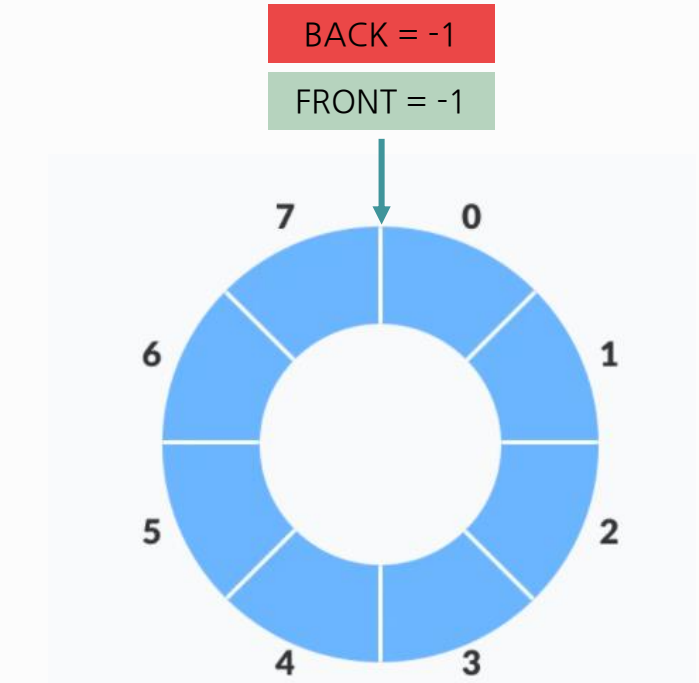
Circular Queue

- 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

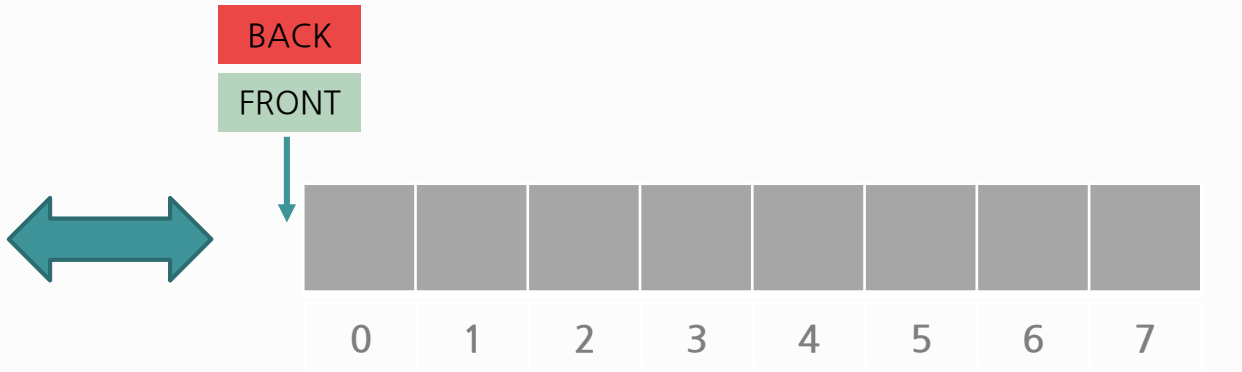
We must shift all list elements by one position to make room for the new item.

Circular Queue - How it works

- Circular Queue works by the process of circular increment i.e., when we try to increment the pointer and we reach the end of the queue, we start from the beginning of the queue.



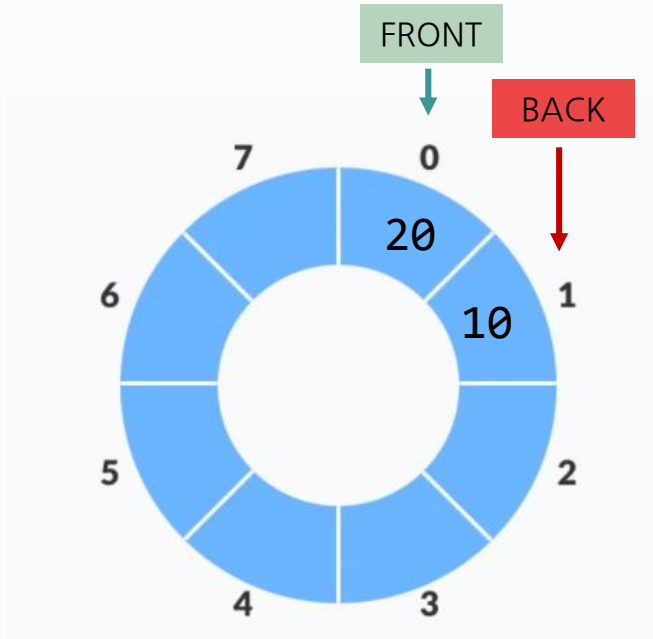
Viewed as a circle instead of line



Avoid the “shifting” of list elements by one position to make room for the new item

Circular Queue - How it works

- Circular Queue works by the process of circular increment i.e., when we try to increment the pointer and we reach the end of the queue, we start from the beginning of the queue.



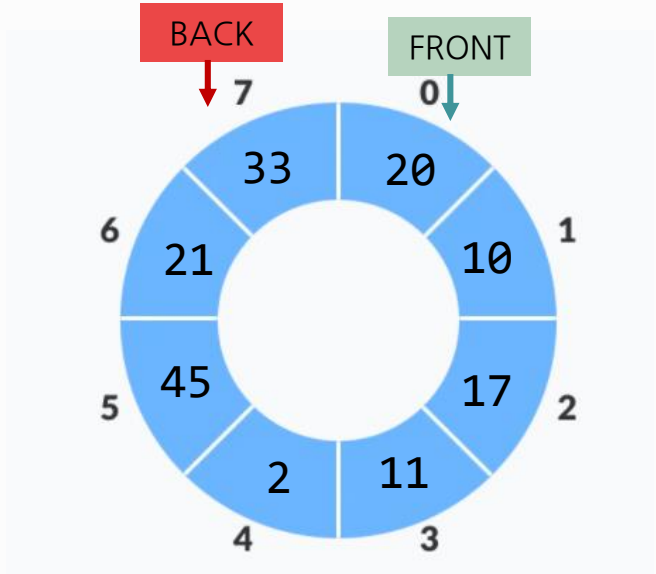
Viewed as a circle instead of line

Avoid the “shifting” of list elements by one position to make room for the new item

Circular Queue - How it works

- Circular Queue works by the process of circular increment i.e., when we try to increment the pointer and we reach the end of the queue, we start from the beginning of the queue.
- The circular increment is performed by modulo division with the queue size. That is,

```
if back + 1 == 8,                (overflow, when back = 7)
   back = (back + 1) % 8 = 0 (then, start of queue)
```



Viewed as a circle instead of line

Avoid the "shifting" of list elements by one position to make room for the new item

Circular Queue Operations

- The circular queue work as follows:
  - Let us have two pointers FRONT and BACK.
  - **FRONT** tracks the first element of the queue.
  - **BACK** tracks the last elements of the queue
  - initially, set value of FRONT and BACK to -1.

Circular Queue Operations - Enqueue

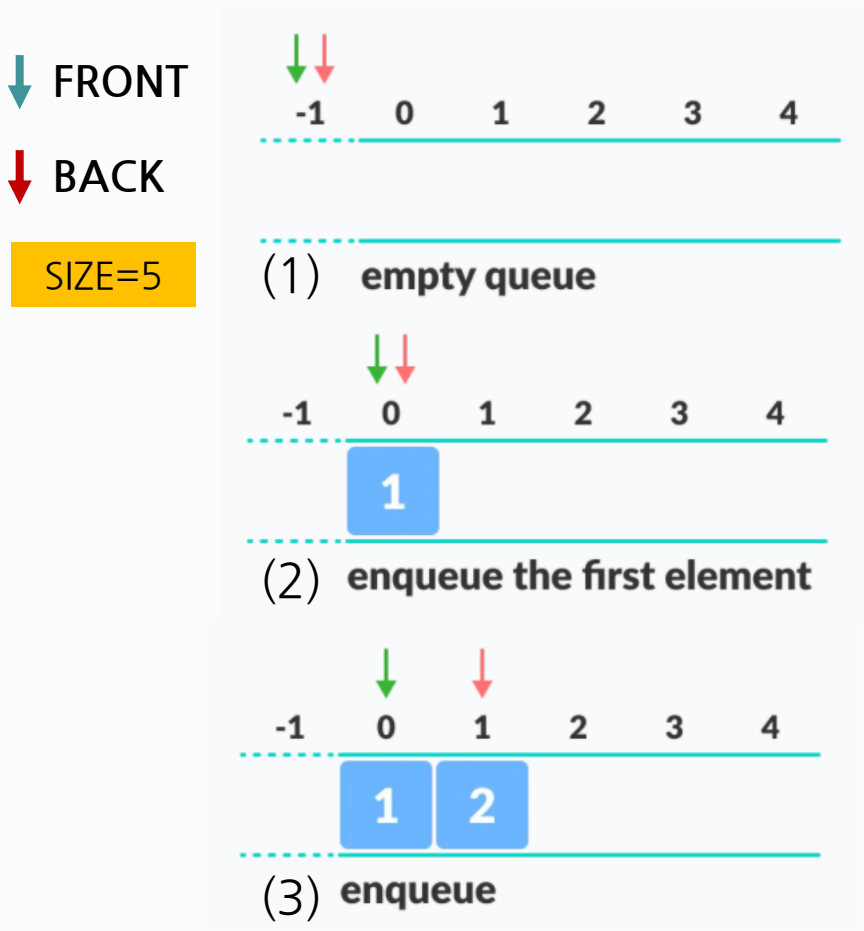
- Check if the queue is full.
- For the first element, set value of FRONT to 0.
- Circularly increase the BACK index by 1.
  (i.e., if the BACK reaches the end, next it would be at the start of the queue)
- Add the new element in the position pointed to by BACK.

Circular Queue Operations - Dequeue

- Check if the queue is empty.
- Return the value pointed by FRONT.
- Circularly increase the FRONT index by 1.
- For the last element, reset the values of FRONT and BACK to -1.

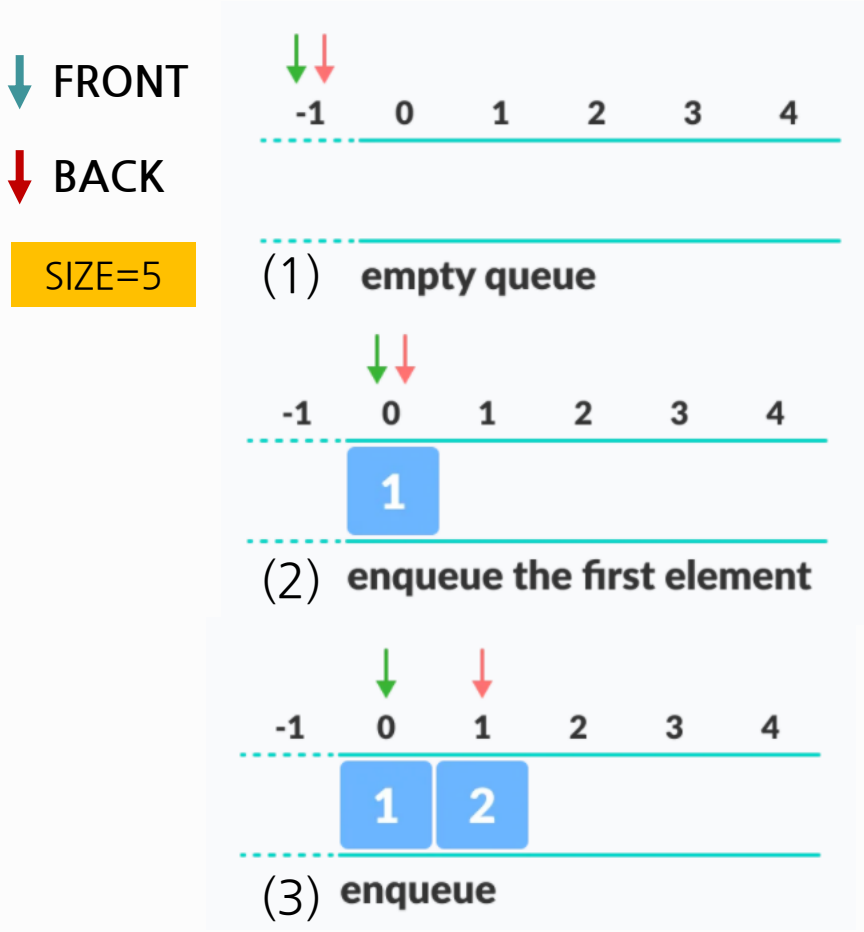
Circular Queue Operations - Full queue special cases

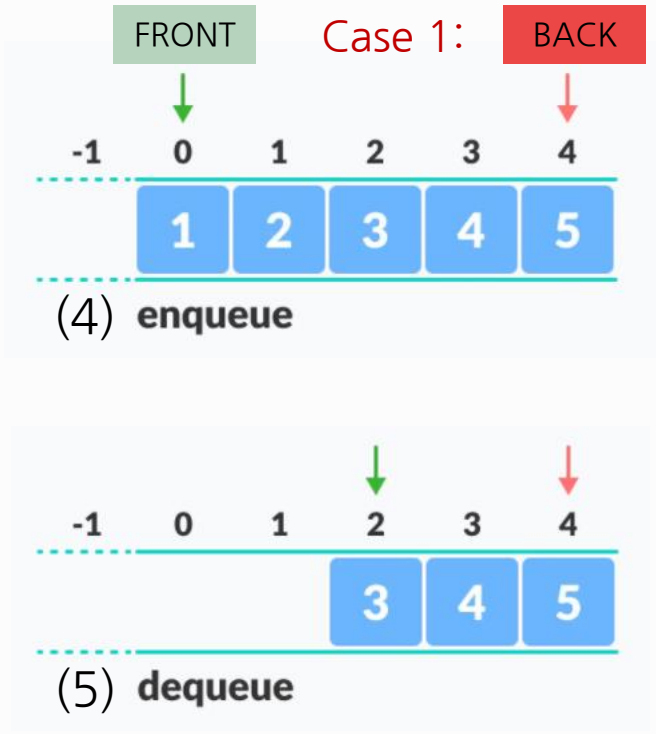
- Case 1: FRONT = 0 && BACK == SIZE - 1
- Case 2: FRONT = BACK + 1
  - The second case happens when BACK starts from 0 due to circular increment and when its value is just 1 less than FRONT, the queue is full.



Circular Queue Operations - Full queue special cases

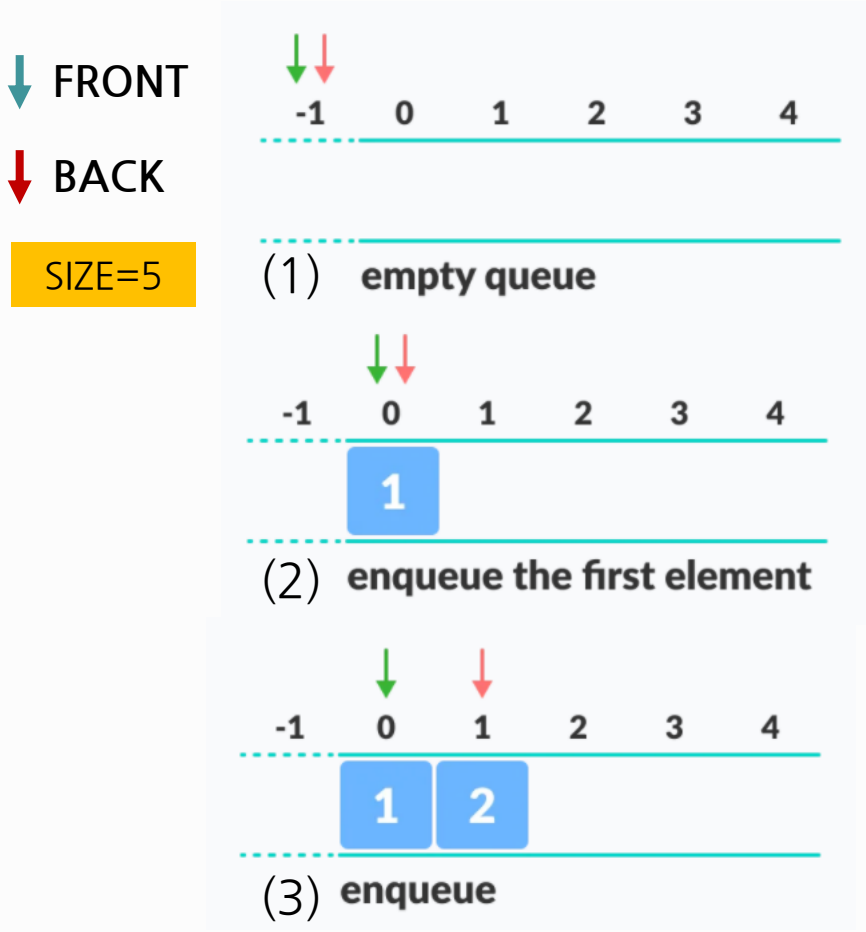
- Case 1: FRONT = 0 && BACK == SIZE - 1
- Case 2: FRONT = BACK + 1
  - The second case happens when BACK starts from 0 due to circular increment and when its value is just 1 less than FRONT, the queue is full.

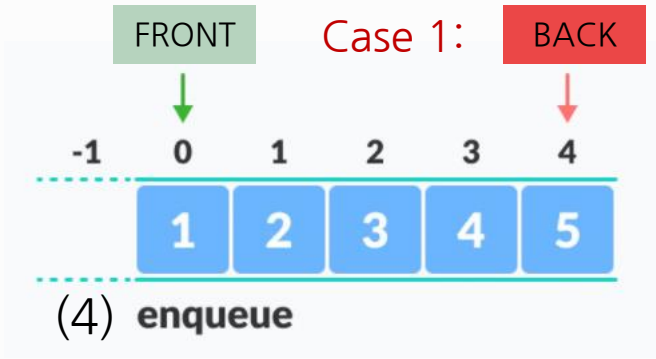


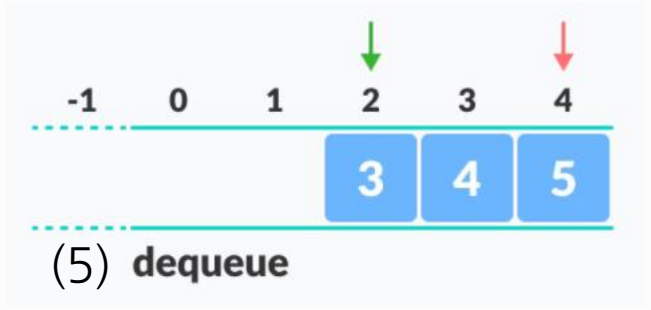


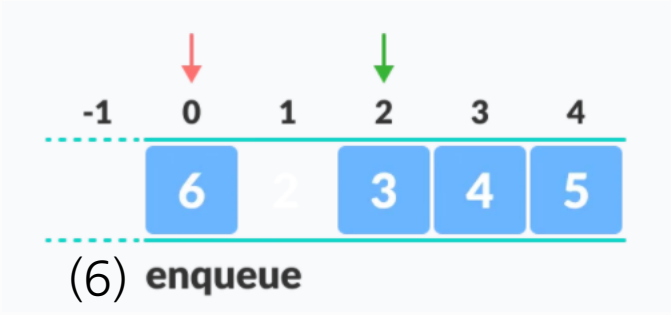
Circular Queue Operations - Full queue special cases

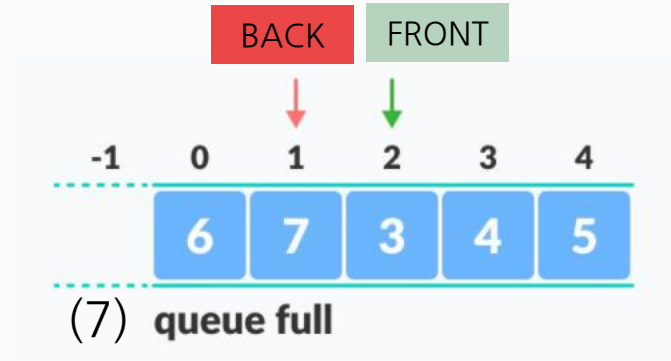
- Case 1: FRONT = 0 && BACK == SIZE - 1
- Case 2: FRONT = BACK + 1
  - The second case happens when BACK starts from 0 due to circular increment and when its value is just 1 less than FRONT, the queue is full.







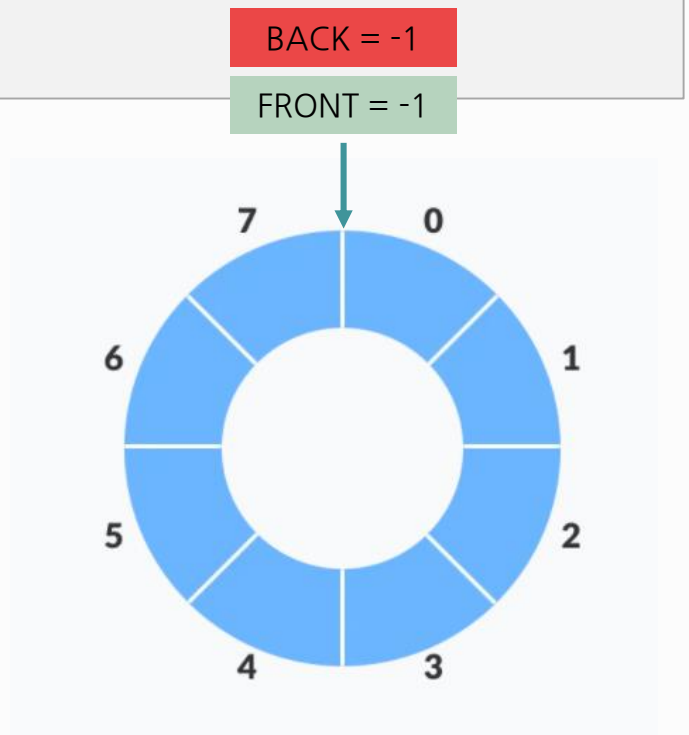




Circular Queue - Full & Empty

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

```
q = QueueCircular(8)
```

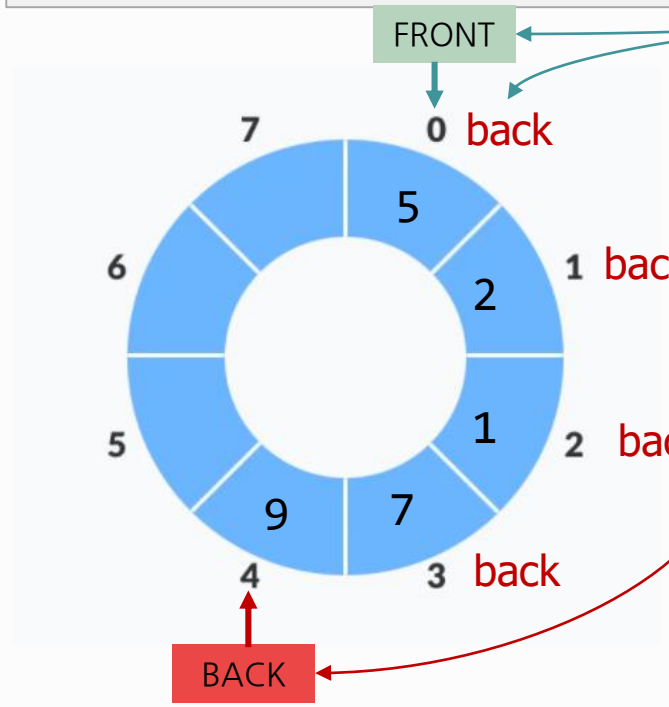



```
def __init__(self, size):
    self.items = [None] * size
    self.MAX = size
    self.front = -1
    self.back = -1
```

Circular Queue

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

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



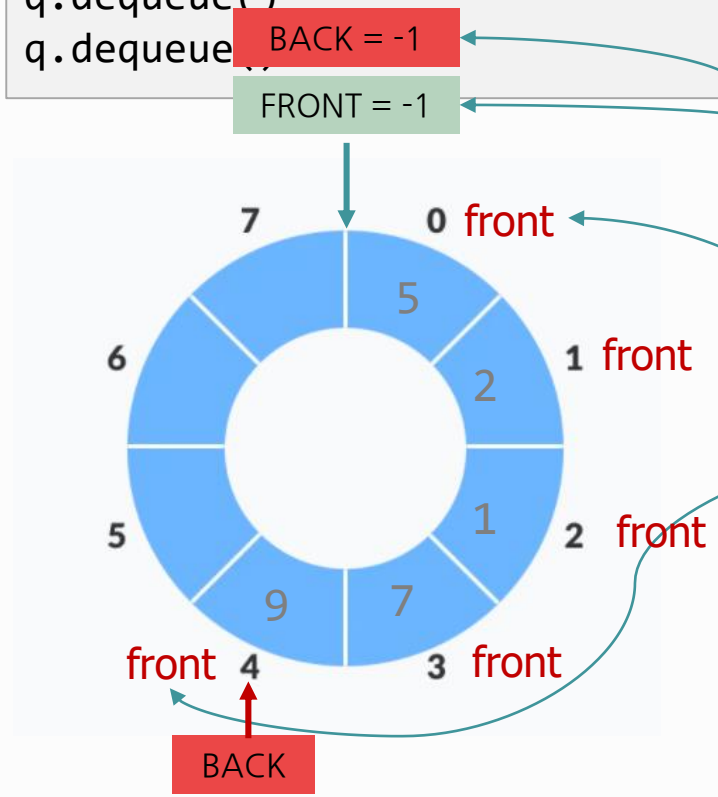

```
def enqueue(self, item): # if not full
    if (self.back + 1) % self.MAX == self.front:
        print("The circular queue is full")
    elif self.front == -1:
        self.front = 0
        self.back = 0
        self.items[self.back] = item
    else:
        self.back = (self.back + 1) % self.MAX
        self.items[self.back] = item
```

Either `enqueue()` or `dequeue()` may contain a bug. It is left as an exercise.

Circular Queue

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

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




```
def dequeue(self): # if not empty
  if self.front == -1:
      print("The circular queue is empty")
  elif self.front == self.back:
      item = self.items[self.front]
      self.front = -1
      self.back = -1
      self.items[self.back] = item
  else:
      item = self.items[self.front]
      self.front = (self.front + 1) % self.MAX
      return item
```

Either `enqueue()` or `dequeue()` may contain a bug. It is left as an exercise.

Circular Queue - Exercise 1

- What are the values of “front” and “back” and their contents after executing the following code snippet?

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

Circular Queue - Exercise 1 solution

- What are the values of “front” and “back” and their contents after executing the following code snippet?

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

- Front: 2 points (25)
- Back: 4 points (30)

Circular Queue - Exercise 2

- Implement QueueCircluar class such that it outputs as shown below?

```
if __name__ == '__main__':
    q = QueueCircular(4)
    print(q)
    q.enqueue(12)
    q.enqueue(17)
    q.enqueue(25)
    q.enqueue(11)
    q.enqueue(30)
    print(q)
    q.dequeue()
    q.dequeue()
    print(q)
```

```
PS C:\GitHub\DSpyx\jupyter> python queueCircular.py
QueueCircular([])
The circular queue is full
QueueCircular([12, 17, 25, 11])
QueueCircular([25, 11])
PS C:\GitHub\DSpyx\jupyter>
```

Circular Queue - Exercise 2 Hint

- Implement QueueCircluar class such that it outputs as shown below?

```
if __name__ == '__main__':
    q = QueueCircular(4)
    print(q)
    q.enqueue(12)
    q.enqueue(17)
    q.enqueue(25)
    q.enqueue(11)
    q.enqueue(30)
    print(q)
    q.dequeue()
    q.dequeue()
    print(q)
```

- Override `__repr__(self)` method.
- Notice that elements are a comma separated.

```
PS C:\GitHub\DSpyx\jupyter> python queueCircular.py
QueueCircular([])
The circular queue is full
QueueCircular([12, 17, 25, 11])
QueueCircular([25, 11])
PS C:\GitHub\DSpyx\jupyter>
```

Circular Queue - Exercise 2 Hint

- Implement QueueCircluar class such that it outputs as shown below?

```
if __name__ == '__main__':
    q = QueueCircular(5)
    print(q)
    for i in range(5, 10):
        q.enqueue(i)
    print(q)
    q.dequeue()
    q.dequeue()
    print(q)
    for i in range(3):
        q.enqueue(i)
    print(q)
```

- notice this additional test case

```
QueueCircular([])
QueueCircular([5, 6, 7, 8, 9])
QueueCircular([7, 8, 9])
The circular queue is full
QueueCircular([7, 8, 9, 0, 1])
```

Circular Queue - Using deque in Python

- If you supply a value to maxlen in Python deque, then your deque will only store up to maxlen items. In this case, you have a bounded deque which works like a circular queue.
  - Once a bounded deque is full with the specified number of items, adding a new item at either end automatically removes and discards the item at the opposite end:

```
from collections import deque

fourOnly = deque([0, 1, 2, 3, 4], maxlen=4)     # Discard 0
print(fourOnly)                                 # deque([1, 2, 3, 4], maxlen=4)
fourOnly.append(5)                              # Automatically remove 1
print(fourOnly)                                 # deque([2, 3, 4, 5], maxlen=4)
fourOnly.append(6)                              # Automatically remove 2
print(fourOnly)                                 # deque([3, 4, 5, 6], maxlen=4)
fourOnly.appendleft(2)                          # Automatically remove 6
print(fourOnly)                                 # deque([2, 3, 4, 5], maxlen=4)
fourOnly.appendleft(1)                          # Automatically remove 5
print(fourOnly)                                 # deque([1, 2, 3, 4], maxlen=4)
fourOnly.maxlen                                 # 4
```

Summary

- Applications of Circular Queue
  - CPU scheduling, Memory management, Traffic Management
  - Models of real-world systems often use queues.

A bottle-capping machine looks like using a sort of circular queue.

학습 정리

1) Queue를 Deque로 구현하면 필요 이상의 메모리를 사용한다

2) Circular queue에서는 FRONT와 BACK pointer를 사용하여 메모리 낭비 문제를 해결할 수 있다

3) Circular queue는 CPU 스케줄링, 메모리 관리, 교통 관리 등 실세계에 다양하게 활용된다

파이썬으로 배우는 데이터 구조

수고했습니다
곧 다음 시간에
다시 뵙겠습니다