06
Stacks and Queues
Chapter 6

Abstract Data Types (ADTs)

An abstract data type (ADT) is an abstraction of a data structure

An ADT specifies:

- Data stored
- Operations on the data
- Error conditions associated with operations

It is not implementation specific, defines basic structure and actions that any implementation has to include.

Stack ADT, Linked List ADT, Queue ADT, etc.

Stack ADT

The Stack ADT stores arbitrary objects.

Insertions and deletions follow the last-in first-out (LIFO) scheme.

The last item placed on the stack will be the first item removed.

Operations:

```
S.push(e)
S.pop()
S.top()
S.is_empty()
len(S)
```

Stack ADT

Examples:

Text editors' undo operation stack

Visited web sites history stack

Arithmetic expression parsing 3 + 1 * 2 ---> + 3 * 1 2 (prefix exp.)

Most of the running code call stack

Html tag, source code paranthesis parsing <table><tr><td>

Stack ADT

Example Operation

Operation	Return Value	Stack Contents
S.push(5)	–	[5]
S.push(3)	–	[5, 3]
len(S)	2	[5, 3]
S.pop()	3	[5]
S.is_empty()	False	[5]
S.pop()	5	[ ]
S.is_empty()	True	[ ]
S.pop()	"error"	[ ]
S.push(7)	–	[7]
S.push(9)	–	[7, 9]
S.top()	9	[7, 9]
S.push(4)	–	[7, 9, 4]
len(S)	3	[7, 9, 4]
S.pop()	4	[7, 9]
S.push(6)	–	[7, 9, 6]
S.push(8)	–	[7, 9, 6, 8]
S.pop()	8	[7, 9, 6]

Implementation (with `list`)

Internally we will use a `list` object, and wrap it with functions that support stack ADT.

Stack Method	*Realization with Python list*
S.push(e)	L.append(e)
S.pop()	L.pop()
S.top()	L[−1]
S.is_empty()	len(L) == 0
len(S)	len(L)

Implementation (with `list`)

```
class ArrayStack:

    def __init__(self):
        self._data = [ ] # nonpublic list instance

    def __len__(self):
        return len(self._data)

    def is_empty(self):
        return len(self._data) == 0

    def push(self, e):
        self._data.append(e) # new item stored at end of list
```

Stack Method	*Realization with Python list*
S.push(e)	L.append(e)
S.pop()	L.pop()
S.top()	L[−1]
S.is_empty()	len(L) == 0
len(S)	len(L)

Implementation (with `list`)

Stack Method	*Realization with Python list*
S.push(e)	L.append(e)
S.pop()	L.pop()
S.top()	L[−1]
S.is_empty()	len(L) == 0
len(S)	len(L)

```python
def top(self):
    if self.is_empty():
        raise Empty('Stack is empty')
    return self._data[-1] # the last item in the list

def pop(self):
    if self.is_empty():
        raise Empty('Stack is empty')
    return self._data.pop( )
```

Implementation (with `list`)

We will also need a special exception type that is going to be thrown when the stack is empty.

```
class Empty(Exception):
    pass
```

Implementation (with `list`)

```
In [53]: S = ArrayStack( ) # contents: [ ]

In [54]: S.push(5) # contents: [5]

In [55]: S.push(3) # contents: [5, 3]

In [56]: print(len(S)) # contents: [5, 3]; outputs 2
2

In [57]: print(S.pop()) # contents: [5]; outputs 3
3

In [58]: print(S.is_empty()) # contents: [5]; outputs False
False

In [59]: print(S.pop()) # contents: [ ]; outputs 5
5

In [60]: print(S.is_empty()) # contents: [ ]; outputs True
True

In [61]: %paste
S.push(7) # contents: [7]
S.push(9) # contents: [7, 9]
print(S.top()) # contents: [7, 9]; outputs 9

## -- End pasted text --
9

In [62]: %paste
S.push(4) # contents: [7, 9, 4]
print(len(S)) # contents: [7, 9, 4]; outputs 3

## -- End pasted text --
3

In [63]: print(S.pop()) # contents: [7, 9]; outputs 4
4

In [64]: S.push(6) # contents: [7, 9, 6]

In [65]: S
Out[65]: <__main__.ArrayStack at 0x6547340>

In [67]: S._data
Out[67]: [7, 9, 6]
```

Complexity

Operation	Running Time
S.push(e)	$O(1)^*$
S.pop()	$O(1)^*$
S.top()	$O(1)$
S.is_empty()	$O(1)$
len(S)	$O(1)$

*amortized

Why amortized?
Recall that append (push) and pop operations on lists can cause the reallocation (extension or shrinking) of the underlying array. As long as the reallocation is performed for a certain fraction of the original array (newsize = oldsize*2, newsize = oldsize*1.25, newsize = oldsize*0.75, etc.), it can be shown that append and pop operations take constant time.

Example-1 Data Reversing

```
In [89]: f = open('deneme.txt', 'w')

In [90]: f.writelines(['shortcut\n','is\n','the\n','longest\n','distance'])

In [91]: f.close()
```

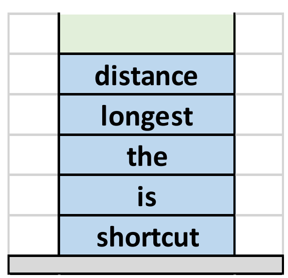
```
shortcut
is
the
longest
distance
```

```
In [92]: reverse_file('deneme.txt')
```

```
distance
longest
the
is
shortcut

```

```
shortcut
is
the
longest
distance
```

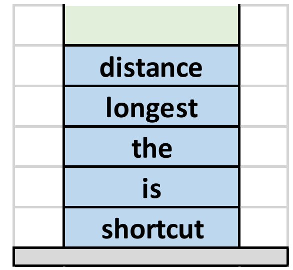



```
distance
longest
the
is
shortcut

```

Example-1 Data Reversing

```
shortcut
is
the
longest
distance
```




```
distance
longest
the
is
shortcut

```

```python
def reverse_file(filename):
    S = ArrayStack()
    original = open(filename)
    for line in original:
        S.push(line.rstrip('\n'))
    original.close()

    output = open(filename,'w')
    while not S.is_empty():
        output.write(S.pop( ) + '\n')
    output.close()
```

Example-2 Algebraic Expressions

Algebraic expressions can be formed in a variety of ways.

infix: operator in between operands (e.g., a*b)

postfix: operator after operands (e.g., ab*)

prefix: operator before operands (e.g., *ab)

Infix Expression	Postfix Expression	Prefix Expression
5 + 2 * 3	5 2 3 * +	+ 5 * 2 3
5 * 2 + 3	5 2 * 3 +	+ * 5 2 3
5 * (2 + 3) - 4	5 2 3 + * 4 -	- * 5 + 2 3 4

Example-2 Algebraic Expressions

Infix notation is easy to read for humans, pre-/postfix notation is easier to parse for a machine.

With pre-/post-fix notation, operator precedence rules is out of concern: No need to have additional knowledge to restore the original expression.

Consider the infix expression 1 # 2 $ 3. Precedence of # and $ is un known, so there are two possible corresponding postfix expressions: 1 2 # 3 $ and 1 2 3 $ #.

Without the precedence knowledge, the infix expression is useless.

Example-2 Algebraic Expressions

Calculating Postfix Expressions

When an operand is entered, the calculator

- Pushes it onto a stack

When an operator is entered, the calculator

- Applies it to the top two operands of the stack
  - Pops the operands from the stack and calculate the result
- Pushes the result of the operation onto the stack

Example-2 Algebraic Expressions

Example: 2 3 4 + *

Key entered	Calculator action		After stack operation: Stack (bottom to top)
2	push 2		2
3	push 3		2 3
4	push 4		2 3 4
+	operand2 = pop stack	(4)	2 3
	operand1 = pop stack	(3)	2
	result = operand1 + operand2	(7)	2
	push result		2 7
*	operand2 = pop stack	(7)	2
	operand1 = pop stack	(2)	
	result = operand1 * operand2	(14)	
	push result		14

Example-2 Algebraic Expressions

Converting Infix to Postfix

Initially `postfix_exp` string is empty. The stack is empty.

For each of the character **`ch`** in infix expression:

- If `ch` is operand, it is appended to `postfix_exp`.
- If `ch` is '(', it is pushed onto the stack.
- If `ch` is ')', all the items in the stack are popped and appended to `postfix_exp` until a '(' is met, and then '(' is removed.
- If `ch` is an operator, it is pushed onto stack.

While stack is not empty,

- Items in the stack are popped and appended to `postfix_exp`.

Example-2 Algebraic Expressions

Converting Infix to Postfix

Example:

ch	Stack (bottom to top)	postfixExp	
a		a	
–	–	a	
(	– (	a	
b	– (	ab	
+	– (+	ab	
c	– (+	abc	
*	– (+ *	abc	
d	– (+ *	abcd	
)	– (+	abcd*	Move operators
	– (	abcd*+	from stack to
	–	abcd*+	postfixExp until " ("
/	– /	abcd*+	
e	– /	abcd*+e	Copy operators from
		abcd*+e/–	stack to postfixExp

For each of the character **ch** in infix expression:

- If ch is operand, it is appended to postfix_exp.
- If ch is '(', it is pushed onto the stack.
- If ch is ')', all the items in the stack are popped and appended to postfix_exp until a '(' is met, and then '(' is removed.
- If ch is an operator, it is pushed onto stack.

While stack is not empty,

- Items in the stack are popped and appended to postfix_exp.

a – (b + c * d) / e ➔ a b c d * + e / –

Example-3 Matching Parantheses

Stacks can be used to verify whether a given text includes matching parantheses.

Matching Parantheses: 3 + [2 * (4 + 5)]

Non-matching Parantheses: 3 + 2 * (4 + 5)]

Example-3 Matching Parantheses

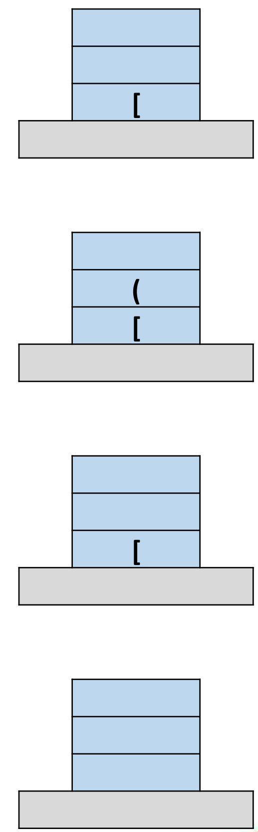
Algorithm (Is matched?):

For each paranthesis **pr** in a given string

- If pr is opening, push it onto stack.
- If pr is closing
  - If stack is empty, return False.
  - Pop pt from stack. If pt is not opening or is not the same type of pr, return False.

If stack is empty, return True.

1 + [3 + 2 * (4 + 5)]



Example-4 Recognizing Strings of a Language

A language L is a set of strings that are formed with a finite set of symbols.

L = {w$w' : w is a (possible empty) string of characters other than $, where w' = reverse(w) }

Which of these strings are in language L?

abccba, aa, $, abc$abc, ab, a

Example-4 Recognizing Strings of a Language

L = {w$w' : w is a (possible empty) string of characters other than $, where w’ = reverse(w) }

abccba, aa, $ are in language L.

abcabc, ab, a$ are not in language L.

Problem:

Deciding whether a given string **str** is in the language L or not.

Solution:

Example-4 Recognizing Strings of a Language

Deciding whether a given string **str** is in the language L or not.

```
sep_seen = false (Initially the separator symbol is not seen.)
For each symbol s in str,
        if not sep_seen:
            if s is $, sep_seen = true
            else push s onto stack
        else
            pop t from stack
            if t != s return false
If stack is empty return True otherwise return False.
```

Example-4 Recognizing Strings of a Language

Problem:

Deciding whether a given string **str** is in the language L or not.

```
def isinL(input_str):
    S = ArrayStack()
    sep_seen = False
    for symbol in input_str:
        if not sep_seen:
            if symbol == '$':
                sep_seen = True
            else:
                S.push(symbol)
        else:
            if S.pop() != symbol:
                return False

    if S.is_empty():
        return True
    else:
        return False
```

Example-5 Program Callstack

At runtime, each function call is pushed onto a **stack** as an activation record.

Each function (subroutine) call from inside of another function causes the stack to grow.

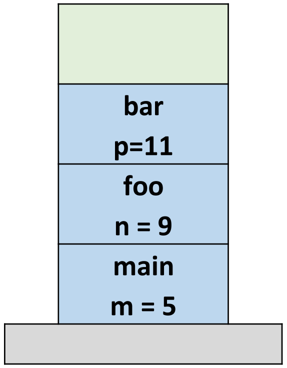
Each function return causes a **pop** operation on the stack.

Too many subroutine calls (e.g., uncontrolled recursive calls) may cause **stack overflow** (i.e., expand beyond a predefined memory area) error.

```
def _main():
    m = 5
    return foo(m + 2)

def foo(_m):
    n = _m + 2
    return bar(n)

def bar(_n):
    p = _n + 2 # <---
    return p
```



Queue ADT

A queue is a list from which items are deleted from one end (front) and into which items are inserted at the other end (rear, or back)

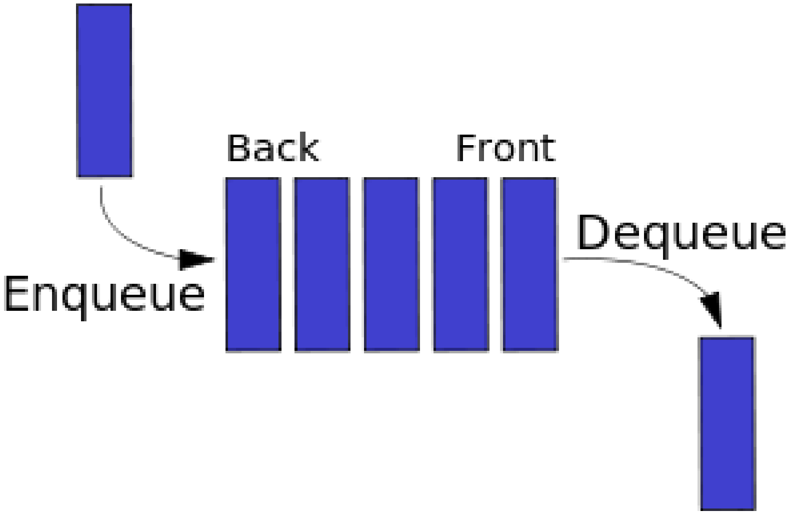
It is like line of people waiting to purchase tickets:

Queue is referred to as a first-in-first-out (FIFO) data structure.

The first item inserted into a queue is the first item to leave

Operations:

```
Q.enqueue(e)
Q.dequeue()
Q.first()
Q.is_empty()
len(Q)
```



Queue ADT

Examples:

Any application where a group of items is waiting to use a shared resource will use a queue.

- Jobs of the same priority in a single processor computer
- Print spooling
- Information segments/packets/frames in computer networks
- Web server responding to requests

Queue ADT

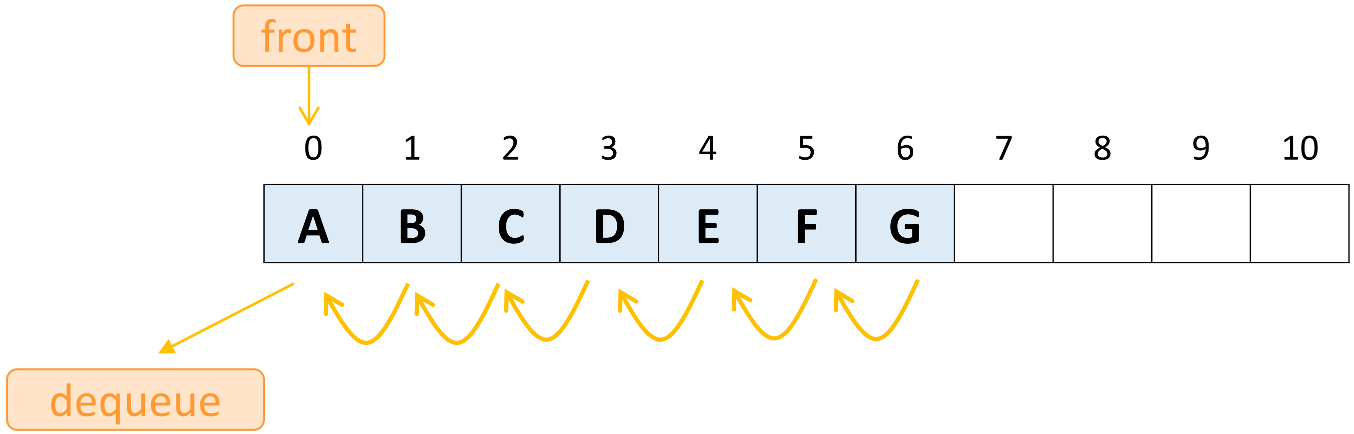
Example Operation

Operation	Return Value	first $\leftarrow Q \leftarrow$ last
Q.enqueue(5)	–	[5]
Q.enqueue(3)	–	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue( )	5	[3]
Q.is_empty( )	False	[3]
Q.dequeue( )	3	[ ]
Q.is_empty( )	True	[ ]
Q.dequeue( )	"error"	[ ]
Q.enqueue(7)	–	[7]
Q.enqueue(9)	–	[7, 9]
Q.first( )	7	[7, 9]
Q.enqueue(4)	–	[7, 9, 4]
len(Q)	3	[7, 9, 4]
Q.dequeue( )	7	[9, 4]

Queue ADT - Array-based Implementation

Python's list class has been quite handy for stack implementation (Approach 1).

- append(e) to add to the end (enqueue)
- pop(0) to get and remove the first element (dequeue)
- pop(0) would cause shifting all (n-1) items to the left the queue

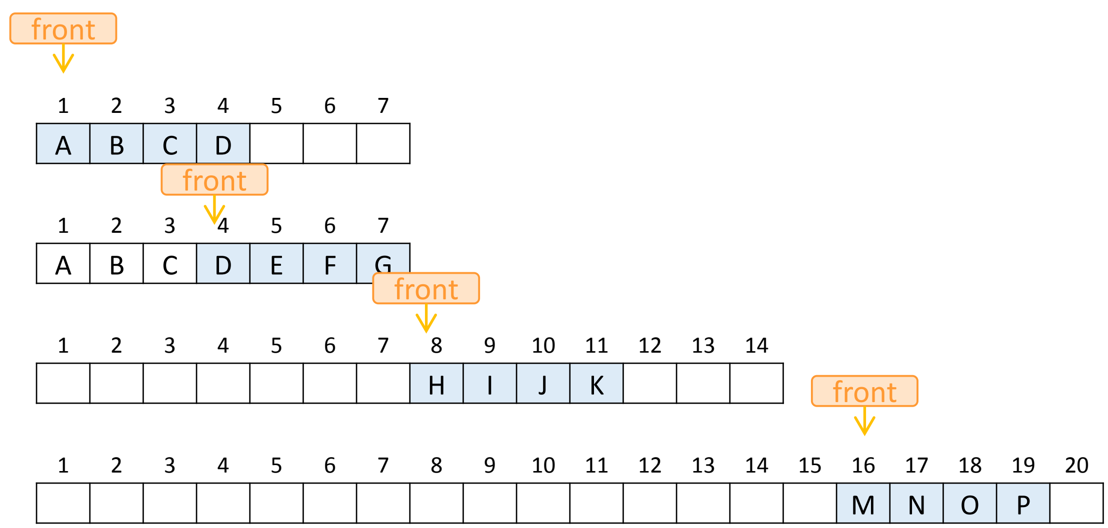


Queue ADT - Array-based Implementation

Python's list class has been quite handy for stack implementation (Approach 2).

- append(e) to add to the end (**enqueue**)
- Use a pointer (_front) to denote the beginning of the queue.

- pop(0) to get and remove the first element (**dequeue**) and increment _front (so no shifting required)
- **Problem:** Underlying array would be ever growing!

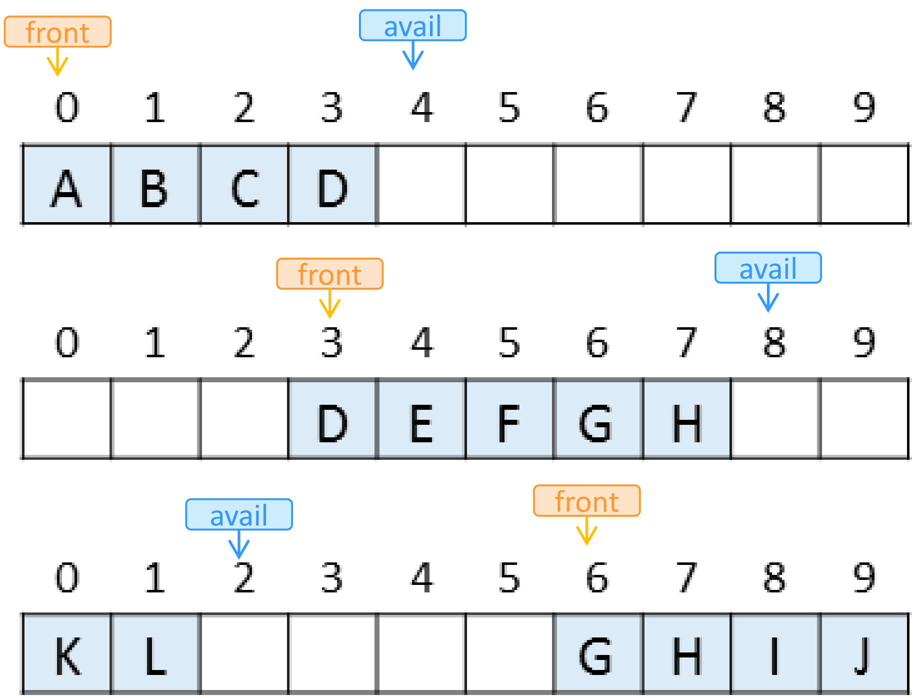


Queue ADT - Circular Array Implementation

size = 4
N = 10

size = 5
N = 10

size = 6
N = 10



dequeue
`front = (front + 1) % N`

enqueue
`avail = (front + size) % N`

resize
when `size == N`

first
returns the item at `first`

Queue ADT - Circular Array Implementation

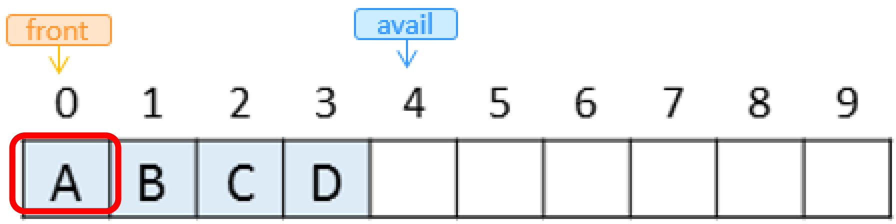
```
class ArrayQueue:
  """FIFO queue implementation using a Python list as underlying storage."""
  DEFAULT_CAPACITY = 10          # moderate capacity for all new queues

  def __init__(self):
    """Create an empty queue."""
    self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
    self._size = 0
    self._front = 0

  def __len__(self):
    """Return the number of elements in the queue."""
    return self._size

  def is_empty(self):
    """Return True if the queue is empty."""
    return self._size == 0
```

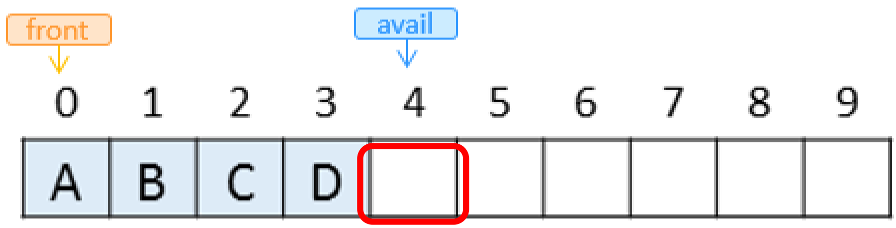
Queue ADT - Circular Array Implementation




```python
def first(self):
    """Return (but do not remove) the element at the front of the queue.

    Raise Empty exception if the queue is empty.
    """
    if self.is_empty():
        raise Empty('Queue is empty')
    return self._data[self._front]
```

Queue ADT - Circular Array Implementation




```
def enqueue(self, e):
  """Add an element to the back of queue."""
  if self._size == len(self._data):
    self._resize(2 * len(self.data))          # double the array size
  avail = (self._front + self._size) % len(self._data)
  self._data[avail] = e
  self._size += 1
```

Queue ADT - Circular Array Implementation

```
def _resize(self, cap):                  # we assume cap >= len(self)
  """Resize to a new list of capacity >= len(self)."""
  old = self._data                       # keep track of existing list
  self._data = [None] * cap              # allocate list with new capacity
  walk = self._front
  for k in range(self._size):            # only consider existing elements
    self._data[k] = old[walk]            # intentionally shift indices
    walk = (1 + walk) % len(old)         # use old size as modulus
  self._front = 0                        # front has been realigned
```

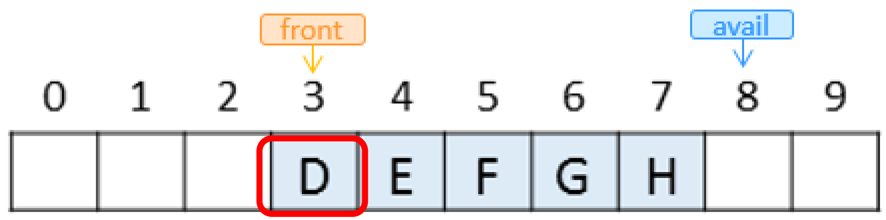
old

0	1	2	3	4	5	6	7	8	9
K	L					G	H	I	J

new

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
G	H	I	J	K	L														

Queue ADT - Circular Array Implementation




```
def dequeue(self):
  """Remove and return the first element of the queue (i.e., FIFO).

  Raise Empty exception if the queue is empty.
  """
  if self.is_empty():
    raise Empty('Queue is empty')
  answer = self._data[self._front]
  self._data[self._front] = None                    # help garbage collection
  self._front = (self._front + 1) % len(self._data)
  self._size -= 1
  return answer
```

Queue ADT - Circular Array Operations

Operation	Complexity
`Q.enqueue(e)`	
`Q.dequeue()`	
`Q.first()`	
`Q.is_empty()`	
`len(Q)`	