

# Stacks, Queues, and Dequeues

Data Structures and Algorithms with Python

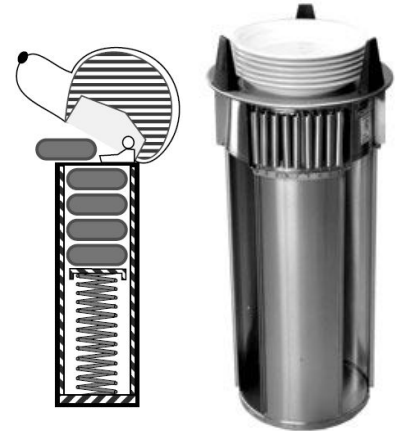
Lecture 6

# Overview

- Stacks
- Queues
- Double-Ended Queues

# Stacks

- A **stack** is a collection of objects that are inserted and removed according to the ***last-in, first-out (LIFO)*** principle.
- A user may insert objects into a stack at any time, but may only access or remove the most recently inserted object that remains (at the so-called “top” of the stack).
- Two operations are defined on a stack:
  - Push - Inserting information into the stack
  - Pop - retrieving information from the stack.
- Examples:
  - Internet Browsers store the addresses of recently visited sites in a stack. Each time a users visits a new site, the address is pushed onto the stack. The browser allows the user to “pop” back to previously visited sites using the “back” button.
  - Text Editors usually provide an “undo” mechanism that cancels recent editing operations and reverts to former states of a document. This undo operation can be accomplished by keeping text changes in a stack



# The Stack Abstract Data Type

- Formally, a stack is an abstract data type (ADT) such that an instance `S` supports the following two methods:
  - `S.push(e)` : Add element `e` to the top of stack `S`.
  - `S.pop()` : Remove and return the top element from the stack `S`; an error occurs if the stack is empty.
- Additionally, let us define the following accessor methods for convenience:
  - `S.top()` : Return a reference to the top element of stack `S`, without removing it; an error occurs if the stack is empty.
  - `S.is_empty()` : Return `True` if stack `S` does not contain any elements.
  - `len(S)` : Return the number of elements in stack `S`; in Python, we implement this with the special method `__len__`.
- By convention, we assume that a newly created stack is empty, and that there is no a priori bound on the capacity of the stack. Elements added to the stack can have arbitrary type.

Operation	Return Value	Stack Contents
<code>S.push(5)</code>	—	[5]
<code>S.push(3)</code>	—	[5, 3]
<code>len(S)</code>	2	[5, 3]
<code>S.pop()</code>	3	[5]
<code>S.is_empty()</code>	False	[5]
<code>S.pop()</code>	5	[]
<code>S.is_empty()</code>	True	[]
<code>S.pop()</code>	“error”	[]
<code>S.push(7)</code>	—	[7]
<code>S.push(9)</code>	—	[7, 9]
<code>S.top()</code>	9	[7, 9]
<code>S.push(4)</code>	—	[7, 9, 4]
<code>len(S)</code>	3	[7, 9, 4]
<code>S.pop()</code>	4	[7, 9]
<code>S.push(6)</code>	—	[7, 9, 6]
<code>S.push(8)</code>	—	[7, 9, 6, 8]
<code>S.pop()</code>	8	[7, 9, 6]

A series of stack operations and their effect on an initially empty stack `S`.

# Simple Array-based Stack Implementation

We will use an **adapter** design pattern to create stack ADT using a list.

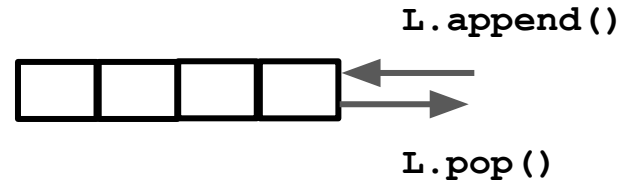
The **adapter** design pattern applies to any context where we effectively want to modify an existing class so that its methods match those of a related, but different, class or interface.

As an example, we will implement a stack using Python List.

We will define a new exception “Empty” which is more appropriate for an empty stack than “IndexError” raised for an empty list.

<i>Stack Method</i>	<i>Realization with Python list</i>
S.push(e)	L.append(e)
S.pop()	L.pop()
S.top()	L[-1]
S.is_empty()	len(L) == 0
len(S)	len(L)

Realization of a stack S as an adaptation of a Python list L.



```

1 # define a new type of exception for stack ADT
2 class Empty(Exception):
3     ''' Error attempting to access an element from an empty container.'''
4     pass
5
6 class ArrayStack:
7     ''' LIFO stack implementation using a Python List as underlying storage'''
8
9     def __init__(self):
10         ''' create an empty stack'''
11         self._data = [] # nonpublic list instance

```

```

12
13     def __len__(self):
14         ''' return the number of elements
15         return len(self._data)
16
17     def is_empty(self):
18         ''' Return True if the stack is empty
19         return len(self._data) == 0
20
21     def push(self, e):
22         ''' Add element e to the top of the stack
23         self._data.append(e) # new item s
24

```

```

25
26     def top(self):
27         '''
28         Return the element at the top of the stack
29         Raise Empty Exception if the stack is empty
30         '''
31         if self.is_empty():
32             raise Empty('Stack is Empty')
33         return self._data[-1] # the last item in the list
34
35     def pop(self):
36         '''
37         Remove and return the element from the top of the stack
38         Raise Empty exception if the stack is empty
39         '''
40         if self.is_empty():
41             raise Empty('Stack is Empty')
42         return self._data.pop()
43
44     def __str__(self):
45         '''
46         A string representation of the stack
47         An arrow shows the top of the stack
48         '''
49         return ''.join(str(self._data)) + '>'

```

```

50 #####
51 S = ArrayStack()
52 S.push(5)
53 S.push(3)
54 print('Stack Length: ', len(S))
55 print('S: ', S)
56 print('Pop ', S.pop())
57 print('Is stack Empty? ', S.is_empty())
58 print('Pop ', S.pop())
59 print('Is stack Empty? ', S.is_empty())
60 print('S:', S)
61 S.push(7)
62 S.push(9)
63 print('Top Element in Stack: ', S.top())
64 S.push(4)
65 S.push(6)
66 print('S: ', S)

```

```

Stack Length: 2
S: [5, 3]>
Pop 3
Is stack Empty? False
Pop 5
Is stack Empty? True
S: []>
Top Element in Stack: 9
S: [7, 9, 4, 6]>

```

## Analyzing the array-based stack implementation

- The implementations for `top()`, `is_empty()` and `len()` functions use constant time in worst case.
- The  $O(1)$  time for `push()` and `pop()` are amortized bounds but can be  $O(n)$  in the worst case when an operation causes the list to resize its internal array.
- The space usage for a stack is  $O(n)$ .
- It is more efficient in practice to construct a list with initial length  $n$  than it is to start with an empty list and append  $n$  items (even though both approaches run in  $O(n)$  time).

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

\*amortized

Performance of our array-based stack implementation.

# Reversing Data using a Stack

- As a consequence of the LIFO protocol, a stack can be used as a general tool to reverse a data sequence.
- Examples
  - printing lines in a file in reverse order.
  - Reverse the elements of a list using a stack.
  - Reversing the order in which elements are stored in a stack.

```
1 # reversing data using a stack
2 def reverse_file(filename):
3     ''' Overwrite given file with its content line-by-line reversed'''
4
5     S = ArrayStack()
6     original = open(filename)
7     for line in original:
8         S.push(line.rstrip('\n')) # we will re-insert newlines when writing
9     original.close()
10
11     # Now we overwrite with contents in LIFO order
12     output = open(filename, 'w') # reopening file overwrites original
13     while not S.is_empty():
14         output.write(S.pop() + '\n') # re-insert newline characters
15     output.close()
16
17 #####
18 file = open("initial.txt", 'w')
19 file.write("I am going home.\n")
20 file.write("Today is a holiday.")
21 file.close()
22
23 !cat initial.txt
24 print('\n\n')
25 reverse_file("initial.txt")
26 !cat initial.txt
```

```
I am going home.
Today is a holiday.
```

```
Today is a holiday.
I am going home.
```



# Matching Parenthesis

- We perform a left-to-right scan of the original sequence, using a stack S to facilitate the matching of grouping symbols.
- Each time we encounter an opening symbol, we push that symbol onto S.
- Each time we encounter a closing symbol, we pop a symbol from the stack S (assuming S is not empty), and check that this forms a valid pair with the corresponding opening symbol.
- If we reach the end of the expression and the stack is empty, then the original expression was properly matched.
- Run-time complexity is  $O(n)$ .
  - Expression having n characters will make n calls to push and n calls to pop. These calls run in  $O(n)$  time.
  - Selection of possible delimiters has fixed size providing constant time for commands: c in lefty, righty.index(c)

```
1 def is_matched(expr):
2     ''' Return True if all delimiters are properly matched; False otherwise'''
3
4     lefty = '([{'          # opening delimiters
5     righty = ')]}'         # respective closing delimiters
6     S = ArrayStack()
7     for c in expr:
8         if c in lefty:      # push left delimiter on stack
9             S.push(c)
10        elif c in righty:
11            if S.is_empty():
12                return False # Nothing to match
13            if righty.index(c) != lefty.index(S.pop()):
14                return False # mismatch
15        return S.is_empty() # were all symbols matched
16
17
18 #####
19
20
21 expr1 = '[(5+x)-(y+z)]'
22 print(is_matched(expr1))
23 expr2 = '[(5+x)-(y+z)]'
24 print(is_matched(expr2))
```

```
False
True
```

# Matching Tags in a Markup Language

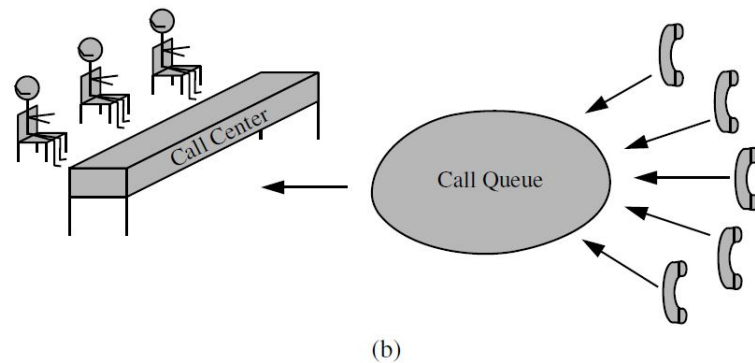
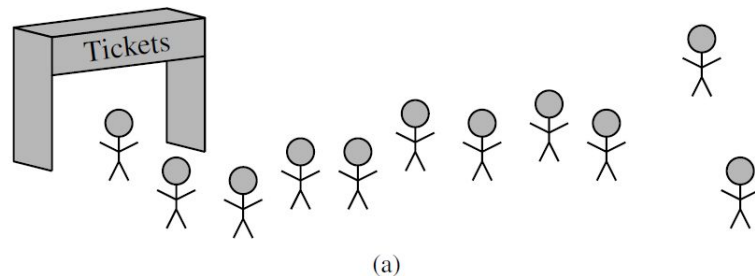
Adjoining code demonstrates the use of stacks in checking for matching tags in a HTML document.

```
1 def is_matched_html(raw):
2     ''' return True if all HTML tags are properly match; False otherwise'''
3     S = ArrayStack()
4     j = raw.find('<')           # find first '<' character (if any)
5     while j != -1:
6         k = raw.find('>', j+1)   # find next '>' character
7         if k == -1:
8             print('Invalid Tag')
9             return False        # invalid tag
10        tag = raw[j+1:k]         # strip away < >
11        if not tag.startswith('/'): # this is opening tag
12            S.push(tag)
13        else:                    # this is closing tag
14            if S.is_empty():
15                print('Stack is empty. Nothing to match with')
16                return False      # nothing to match with
17            if tag[1:] != S.pop():
18                print('Tag Mismatch:', tag)
19                return False      # mismatched delimiter
20            j = raw.find('<', k+1)  # find next '<' character (if any)
21        return S.is_empty()
22
23 #####
24
25 is_matched_html('<<body>
26 <center>
27 <h1> The Little Boat </h1>
28 </center>
29 <p> The storm tossed the little boat like a cheap sneaker in an
30 old washing machine. The three drunken fishermen were used to
31 such treatment, of course, but not the tree salesman, who even as
32 a stowaway now felt that he had overpaid for the voyage. </p>
33 <ol>
34 <li> Will the salesman die? </li>
35 <li> What color is the boat? </li>
36 <li> And what about Naomi? </li>
37 </ol>
38 </body>''')
```

True

# Queues

- It is a close “cousin” of the stack.
- A **queue** is a collection of objects that are inserted and removed according to the **first-in, first-out (FIFO)** principle.
- The elements can be inserted at any time, but only the element that has been in the queue the longest can be next removed.
- Examples:
  - People waiting to get on an amusement park ride.
  - Phone calls being routed to a customer service center.



# The Queue Abstract Data Type

- Formally, the queue abstract data type defines a collection that keeps objects in a sequence, where *element access and deletion are restricted* to the **first** element in the queue, and *element insertion is restricted* to the **back** of the sequence.
- In other words, FIFO rule is enforced on the sequence.
- The queue ADT supports two fundamental methods for a queue Q:
  - **Q.enqueue(e)** : Add element e to the back of queue Q.
  - **Q.dequeue()** : Remove and return the first element from queue Q; an error occurs if the queue is empty.
- Additional accessor methods:
  - **Q.first()** : Return a reference to the element at the front of queue Q, without removing it; an error occurs if the queue is empty.
  - **Q.is\_empty()** : Return True if queue Q does not contain any elements.
  - **len(Q)** : Return the number of elements in queue Q; in Python, we implement this with the special method `__len__` .

- By convention, we assume that a newly created queue is empty.
- There is no a priori bound on the capacity of the queue.
- Elements added to the queue can have arbitrary type.

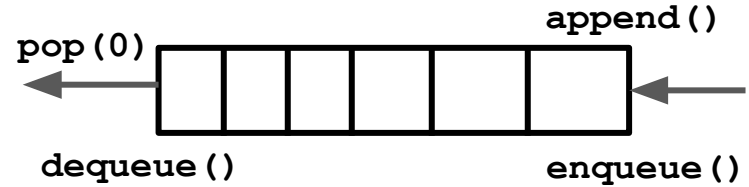
Operation	Return Value	first $\leftarrow$ Q $\leftarrow$ last
Q.enqueue(5)	–	[5]
Q.enqueue(3)	–	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue()	5	[3]
Q.is_empty()	False	[3]
Q.dequeue()	3	[ ]
Q.is_empty()	True	[ ]
Q.dequeue()	“error”	[ ]
Q.enqueue(7)	–	[7]
Q.enqueue(9)	–	[7, 9]
Q.first()	7	[7, 9]
Q.enqueue(4)	–	[7,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

Series of queue operations and their effect on an initially empty queue Q.

# Array-based Queue Implementation

## Implementation #1

- Use **append(e)** to add elements at the end of the list.
- Use **pop(0)** to remove the first element from the list when dequeuing.
  - Inefficient implementation with  $\Theta(n)$  time.



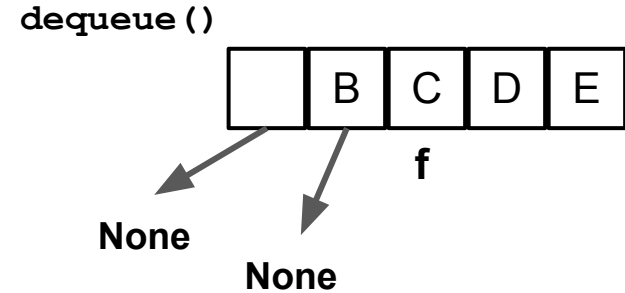
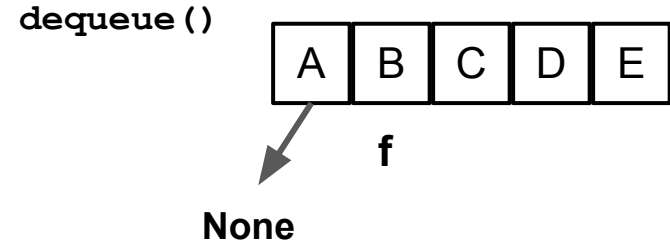
## Implementation #2



- Same method for adding elements
- Replacement of pop(0) implementation:
  - Replace the dequeued entry in the array with a reference to None.
  - Maintain an explicit variable  $f$  to store the index of the element that is currently at the front of the queue.
  - **$O(1)$  run-time.**
  - But  **$O(m)$  space requirement for  $n$  elements** where  $m > n$ , where  $m$  is the number of enqueue operations and  $n$  is the actual number of elements in the queue.
  - In other words, a queue with a fewer elements will be stored in a large array.

This design would have detrimental consequences in applications in which queues have relatively modest size, but which are used for long periods of time.

For example, the wait-list for a restaurant might never have more than 30 entries at one time, but over the course of a day (or a week), the overall number of entries would be significantly larger.





## Using an Array Circularly

- We allow the front of the queue to drift rightward and allow the contents of the queue to wrap around the end of an underlying array.
- We assume that our underlying array has fixed length  $N$  that is greater than the actual number of elements in the queue.
- New elements are enqueued toward the “end” of the current queue, progressing from the front to index  $N - 1$  and continuing at index 0, then 1.
- This provides  $O(1)$  memory requirement for the implementation.



$$f = (f+1) \% N$$

# A Python Queue Implementation

- Internally, the queue class maintains the following three instance variables:
  - `_data`: is a reference to a list instance with a fixed capacity.
  - `_size`: is an integer representing the current number of elements stored in the queue (as opposed to the length of the data list).
  - `_front`: is an integer that represents the index within data of the first element of the queue (assuming the queue is not empty).
- We provide the following accessor functions:
  - `len()` : Current Queue length
  - `is_empty()` : Check if the queue is empty
  - `first()` : return the element at the front of queue
- Mutator functions:
  - `enqueue()` : To add element into the queue
  - `dequeue()` : To remove element from the queue

```
1 class ArrayQueue:
2     '''
3     FIFO Queue implementation using a Python List as underlying storage
4     '''
5     DEFAULT_CAPACITY = 5          # moderate capacity for all new queues
6
7     def __init__(self):
8         ''' Create an empty queue '''
9         self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
10        self._size = 0
11        self._front = 0
12
13
14    def __len__(self):
15        ''' return the number of elements in the queue'''
16        return self._size
17
18    def is_empty(self):
19        ''' Return True if the queue is empty'''
20        return self._size == 0
21
22    def first(self):
23        '''
24        Return (but do not remove) the element at the front of the queue
25        Raise Empty Exception if the queue is empty.
26        '''
27        if self.is_empty():
28            raise Empty('Queue is Empty')
29        return self._data[self._front]
30
```

```

32 def dequeue(self):
33     '''
34     remove and return the first element of the queue.
35     raise Empty exception if the queue is empty.
36     '''
37     if self.is_empty():
38         raise Empty('Queue is Empty')
39     answer = self._data[self._front]
40     self._data[self._front] = None # help garbage collection
41     self._front = (self._front + 1) % len(self._data) # circular indexing
42     self._size -= 1 # reduce the queue size
43     return answer
44
45 def enqueue(self, e):
46     '''Add an element to the back of queue'''
47     if self._size == len(self._data):
48         self._resize(2*len(self._data)) # double the array size
49     avail = (self._front + self._size) % len(self._data)
50     self._data[avail] = e
51     self._size += 1
52
53
54 def _resize(self, cap):
55     '''resize to a new list of capacity >= len(self)'''
56     old = self._data
57     self._data = [None] * cap
58     walk = self._front
59     for k in range(self._size): # only consider existing elements
60         self._data[k] = old[walk] # intentionally shift indices
61         walk = (1+walk) % len(old) # use old size as modulus
62     self._front = 0 # front has been realigned.
63
64 def __str__(self):
65     '''string representation of the queue'''
66     return '<'+''.join(str(self._data))+'<'
67

```

```

68
69 #####
70
71 Q = ArrayQueue()
72 Q.enqueue(5)
73 Q.enqueue(7)
74 Q.enqueue(9)
75 Q.enqueue(2)
76 Q.enqueue(6)
77 Q.enqueue(4)
78 Q.enqueue(1)
79 Q.enqueue(0)
80 |
81 print('Q: ', Q)
82 print('Queue Length:', len(Q))
83 print('Remove last item: ', Q.dequeue())
84 print('Remove last item: ', Q.dequeue())
85 print('Q: ', Q)
86 print('Queue Length:', len(Q))
87
88
89
90 Q: <[5, 7, 9, 2, 6, 4, 1, 0, None, None]<
91 Queue Length: 8
92 Remove last item: 5
93 Remove last item: 7
94 Q: <[None, None, 9, 2, 6, 4, 1, 0, None, None]<
95 Queue Length: 6

```

## Adding and removing elements

- Index where the next element is added to the queue:

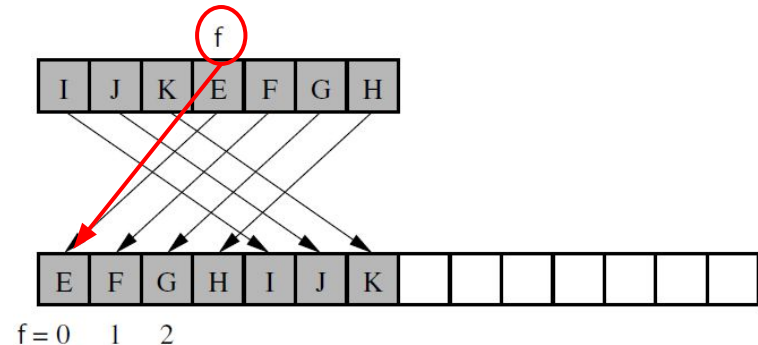
`avail = (self._front + self._size) % len(self._data)`

- During dequeuing, the value at index `self._front` is returned and removed. The `_front` index is updated to next position in the circular queue.

`self._front = (self._front + 1) % len(self._data)`

## Resizing the Queue

- When queue size equals the size of the underlying array, the storage capacity of the underlying list is doubled.
- The references from the old list is copied into the new list.
- While transferring the contents, we intentionally realign the front of the queue with index 0 in the new array.



# Shrinking the underlying array

```
def dequeue(self):
    """
    remove and return the first element of the queue.
    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) # circular indexing
    self._size -= 1 # reduce the queue size

    if 0 < self._size < len(self._data) // 4: # shrink the array size by half
        self._resize(len(self._data)//2) # when queue size 1/4 of the
    return answer # total array capacity
```

- A robust approach is to reduce the array to half of its current size, whenever the number of elements stored in it falls below one fourth of its capacity.
- Space usage:  $\sim \Theta(n)$

```
75 Q = ArrayQueue()
76 Q.enqueue(5)
77 Q.enqueue(7)
78 Q.enqueue(9)
79 Q.enqueue(2)
80 Q.enqueue(6)
81 Q.enqueue(4)
82 Q.enqueue(1)
83 Q.enqueue(0)
84
85 print('Q: ', Q)
86 print('Queue Length:', len(Q))
87 print('Remove last item: ', Q.dequeue())
88 print('Remove last item: ', Q.dequeue())
89 print('Q: ', Q)
90 print('Queue Length:', len(Q))
91 print('Remove last item: ', Q.dequeue())
92 print('Remove last item: ', Q.dequeue())
93 print('Remove last item: ', Q.dequeue())
94 print('Remove last item: ', Q.dequeue())
95 print('Q: ', Q)
96 print('Queue Length:', len(Q))
97 print('Remove last item: ', Q.dequeue())
98 print('Q: ', Q)
99 print('Queue Length:', len(Q))
100
101
102
```

```
Q:  <[5, 7, 9, 2, 6, 4, 1, 0, None, None]>
Queue Length: 8
Remove last item: 5
Remove last item: 7
Q:  <[None, None, 9, 2, 6, 4, 1, 0, None, None]>
Queue Length: 6
Remove last item: 9
Remove last item: 2
Remove last item: 6
Remove last item: 4
Q:  <[None, None, None, None, None, None, 1, 0, None, None]>
Queue Length: 2
Remove last item: 1
Q:  <[0, None, None, None, None]>
Queue Length: 1
```

## Analyzing the array-based Queue Implementation

Operation	Running Time
Q.enqueue(e)	$O(1)^*$
Q.dequeue()	$O(1)^*$
Q.first()	$O(1)$
Q.is_empty()	$O(1)$
len(Q)	$O(1)$

\*amortized

## Double-Ended Queues

- A **deque** (*pronounced as “deck”*) or a **double-ended queue** is a queue-like data structure that supports insertion and deletion at both the front and the back of the queue.
- The deque abstract data type is more general than both the stack and the queue ADTs.
- Example: a restaurant using a queue to maintain a waitlist
  - Occasionally, the first person might be removed from the queue only to find that a table was not available; typically, the restaurant will re-insert the person at the first position in the queue.
  - It may also be that a customer at the end of the queue may grow impatient and leave the restaurant.

# The Deque Abstract Data Type

- The deque ADT is defined so that deque *D* supports the following methods:
  - ***D.add\_first(e)*** : Add element *e* to the front of deque *D*.
  - ***D.add\_last(e)*** : Add element *e* to the back of deque *D*.
  - ***D.delete\_first()*** : Remove and return the first element from deque *D*; an error occurs if the deque is empty.
  - ***D.delete\_last()*** : Remove and return the last element from deque *D*; an error occurs if the deque is empty.
- Additionally, the deque ADT will include the following accessors:
  - ***D.first()*** : Return (but do not remove) the first element of deque *D*; an error occurs if the deque is empty.
  - ***D.last()*** : Return (but do not remove) the last element of deque *D*; an error occurs if the deque is empty.
  - ***D.is\_empty()*** : Return True if deque *D* does not contain any elements.
  - ***len(D)*** : Return the number of elements in deque *D*; in Python, we implement this with the special method `len`.



## Implementing Deque with a circular array

- Maintain same three instance variables: **`_data`**, **`_size`** and **`_front`**.

- Back of the queue:

`back = (self._front + self._size - 1) % len(self._data)`

- **`add_first()`** requires circularly decrementing index:

`self._front = (self._front - 1) % len(self._data)`

Operation	Return Value	Deque
D.add_last(5)	—	[5]
D.add_first(3)	—	[3, 5]
D.add_first(7)	—	[7, 3, 5]
D.first()	7	[7, 3, 5]
D.delete_last()	5	[7, 3]
len(D)	2	[7, 3]
D.delete_last()	3	[7]
D.delete_last()	7	[ ]
D.add_first(6)	—	[6]
D.last()	6	[6]
D.add_first(8)	—	[8, 6]
D.is_empty()	False	[8, 6]
D.last()	6	[8, 6]

A series of operations and their effects  
on an initially empty deque D of integers

# Dequeues in Python Collections Module

Our Deque ADT	collections.deque	Description
len(D)	len(D)	number of elements
D.add_first()	D.appendleft()	add to beginning
D.add_last()	D.append()	add to end
D.delete_first()	D.popleft()	remove from beginning
D.delete_last()	D.pop()	remove from end
D.first()	D[0]	access first element
D.last()	D[-1]	access last element
	D[j]	access arbitrary entry by index
	D[j] = val	modify arbitrary entry by index
	D.clear()	clear all contents
	D.rotate(k)	circularly shift rightward k steps
	D.remove(e)	remove first matching element
	D.count(e)	count number of matches for e

## Performance:

- The deque class is formally documented to guarantee  $O(1)$ -time operations at either end, but  $O(n)$ -time worst-case operations when using index notation near the middle of the deque.

```
1 import collections
2 D = collections.deque()
3 D.appendleft(5)
4 D.appendleft(6)
5 D.append(10)
6 D.append(2)
7 D.appendleft(3)
8 D.appendleft(7)
9 print('Deque D: ', D)
10 print('Length: ', len(D))
11 D.rotate(5) #circularly shift rightward k steps
12 print('Deque D: ', D)
13 D.popleft()
14 D.pop()
15 print('Deque D: ', D)
16 print('Length: ', len(D))
```

```
Deque D:  deque([7, 3, 6, 5, 10, 2])
Length:  6
Deque D:  deque([3, 6, 5, 10, 2, 7])
Deque D:  deque([6, 5, 10, 2])
Length:  4
```

# Summary

- We studied about the following three ADTs
  - Stacks
  - Queues
  - Deques
- Implementing these ADTs using python lists?
- Analyzing the performance of these implementations.