Stacks, Queues, and Deques

Data Structures and Algorithms with Python

Lecture 6

Overview

- Stacks
- Queues
- Double-Ended Queues

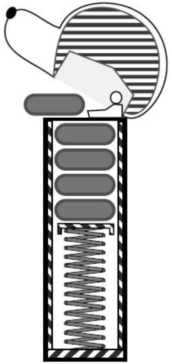
Stacks

- 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 stac



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
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]

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.

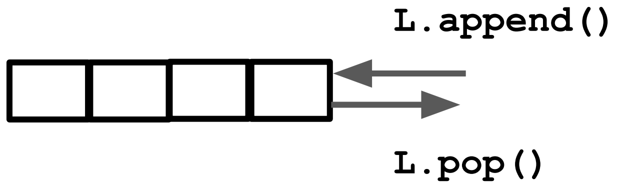
The ***adapter*** design pattern applies to any context where we effectively want to modify an existing class so that its methods match those of a related, but different, class or interface.

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.

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)

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



```python
# define a new type of exception for stack ADT
class Empty(Exception):
  ''' Error attempting to access an element from an empty container.'''
  pass

class ArrayStack:
  ''' LIFO stack implementation using a Python List as underlying storage'''

  def __init__(self):
    ''' create an empty stack'''
    self._data = []    # nonpublic list instance

  def __len__(self):
    ''' return the number of elements
    return len(self._data)

  def is_empty(self):
    ''' Return True if the stack is em
    return len(self._data) == 0

  def push(self, e):
    ''' Add element e to the top of th
    self._data.append(e)  # new item s
```

```python
  def top(self):
    '''
    Return the element at the top of the stack
    Raise Empty Exception if the stack is empty
    '''
    if self.is_empty():
      raise Empty('Stack is Empty')
    return self._data[-1]            # the last item in the li

  def pop(self):
    '''
    Remove and return the element from the top of the stack
    Raise Empty excepion if the stack is empty
    '''
    if self.is_empty():
      raise Empty('Stack is Empty')
    return self._data.pop()

  def __str__(self):
    '''
    A string representation of the stack
    An arrow shows the top of the stack
    '''
    return ''.join(str(self._data)) +'>'
```

```python
################
S = ArrayStack()
S.push(5)
S.push(3)
print('Stack Length: ', len(S))
print('S: ', S)
print('Pop ', S.pop())
print('Is stack Empty? ', S.is_empty())
print('Pop ', S.pop())
print('Is stack Empty? ', S.is_empty())
print('S:', S)
S.push(7)
S.push(9)
print('Top Element in Stack: ', S.top())
S.push(4)
S.push(6)
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
S.push(e)	$O(1)^*$
S.pop()	$O(1)^*$
S.top()	$O(1)$
S.is_empty()	$O(1)$
len(S)	$O(1)$

*amortized

Average time -> not all constant

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.

```
# reversing data using a stack
def reverse_file(filename):
  ''' Overwrite given file with its conent line-by-line reversed'''

  S = ArrayStack()
  original = open(filename)
  for line in original:
    S.push(line.rstrip('\n')) # we will re-insert newlines when writing
  original.close()

  # Now we overwrite with contents in LIFO order
  output = open(filename, 'w') # reopening file overwrites original
  while not S.is_empty():
    output.write(S.pop() + '\n') # re-insert newline characters
  output.close()

############
file = open("initial.txt", 'w')
file.write("I am going home.\n")
file.write("Today is a holiday.")
file.close()

!cat initial.txt
print('\n\n')
reverse_file("initial.txt")
!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)

```python
def is_matched(expr):
  ''' Return True if all delimiters are properly matched; False otherwise'''

  lefty = '({['              # opening delimiters
  righty = ')}]'             # respective closing delimiters
  S = ArrayStack()
  for c in expr:
    if c in lefty:           # push left delimiter on stack
      S.push(c)
    elif c in righty:
      if S.is_empty():
        return False         # Nothing to match
      if righty.index(c) != lefty.index(S.pop()):
        return False         # mismatch
  return S.is_empty()        # were all symbols matched


##########


expr1 = '[(5+x)-(y+z)}'
print(is_matched(expr))
expr2 = '[(5+x)-(y+z)]'
print(is_matched(expr2))
```

```
False
True
```

Matching Tags in a Markup Language

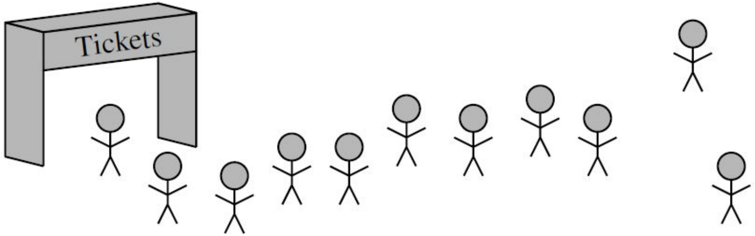
Adjoining code demonstrates the use of stacks in checking for matching tags in a HTML document.

```
def is_matched_html(raw):
  ''' return True if all HTML tags are properly match; False otherwise'''
  S = ArrayStack()
  j = raw.find('<')                    # find first '<' character (if any)
  while j != -1:
    k = raw.find('>', j+1)             # find next '>' character
    if k == -1:
      print('Invalid Tag')
      return False                     # invalid tag
    tag = raw[j+1:k]                   # strip away < >
    if not tag.startswith('/'):        # this is opening tag
      S.push(tag)
    else:                              # this is closing tag
      if S.is_empty():
        print('Stack is empty. Nothing to match with')
        return False                   # nothing to match with
      if tag[1:] != S.pop():
        print('Tag Mismatch:', tag)
        return False                   # mismatched delimiter
    j = raw.find('<', k+1)             # find next '<' character (if any)
  return S.is_empty()

################

is_matched_html('''<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little boat like a cheap sneaker in an
old washing machine. The three drunken fishermen were used to
such treatment, of course, but not the tree salesman, who even as
a stowaway now felt that he had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>''')
```

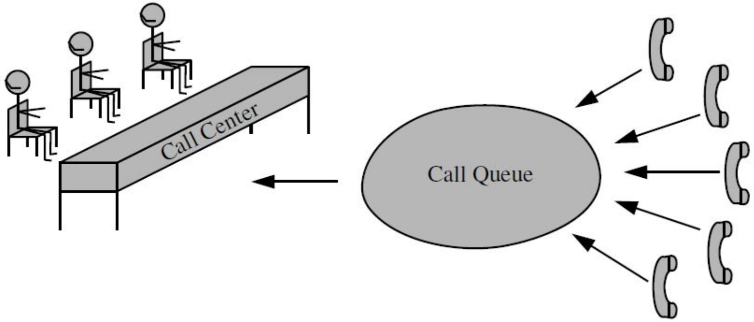
```
True
```

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.



(a)



(b)

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.

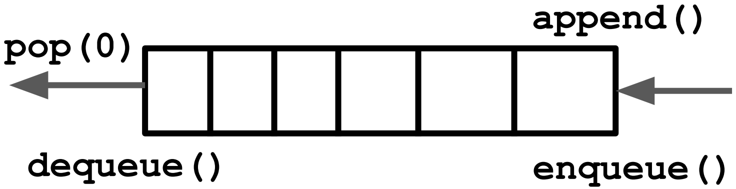
Operation	Return Value	first ← Q ← last
Q.enqueue(5)	–	[5]
Q.enqueue(3)	–	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue( )	5	[3]
Q.is_empty( )	False	[3]
Q.dequeue( )	3	[ ]
Q.is_empty( )	True	[ ]
Q.dequeue( )	"error"	[ ]
Q.enqueue(7)	–	[7]
Q.enqueue(9)	–	[7, 9]
Q.first( )	7	[7, 9]
Q.enqueue(4)	–	[7, 9, 4]
len(Q)	3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

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

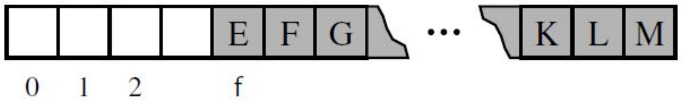
Array-based Queue Implementation

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.



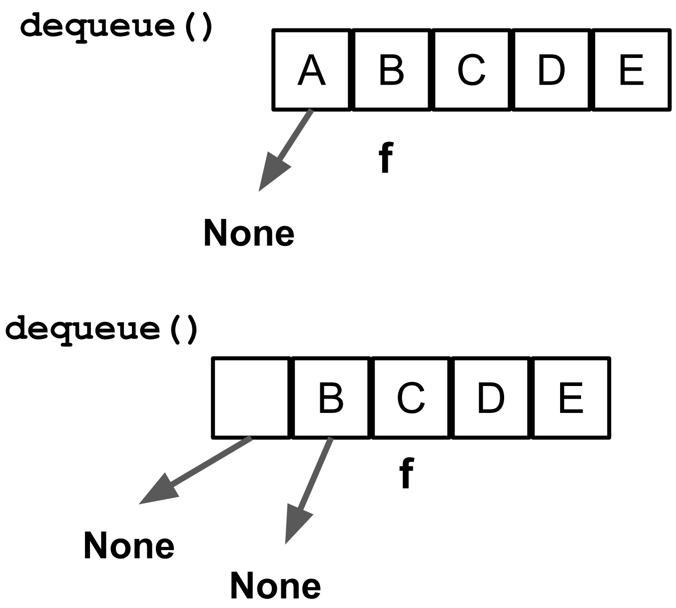
Implementation #2



- 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.



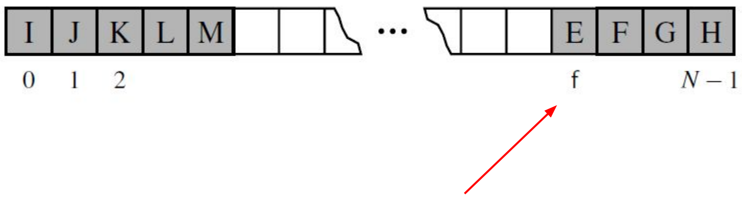
Using an Array Circularly

- 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 <u>O(1) memory requirement</u> 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

```python
class ArrayQueue:
  '''
  FIFO Queue implementation using a Python List as underlying storage
  '''
  DEFAULT_CAPACITY = 5      # 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

  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]
```

```python
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
  return answer

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


def _resize(self, cap):
  ''' resize to a new list of capacity >= len(self)'''
  old = self._data
  self._data = [None] * cap
  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.

def __str__(self):
  ''' string representation of the queue'''
  return '<'+''.join(str(self._data)) +'<'
```

```python
#################

Q = ArrayQueue()
Q.enqueue(5)
Q.enqueue(7)
Q.enqueue(9)
Q.enqueue(2)
Q.enqueue(6)
Q.enqueue(4)
Q.enqueue(1)
Q.enqueue(0)

print('Q: ', Q)
print('Queue Length:', len(Q))
print('Remove last item: ', Q.dequeue())
print('Remove last item: ', Q.dequeue())
print('Q: ', Q)
print('Queue Length:', len(Q))
```

```
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
```

Adding and removing elements

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

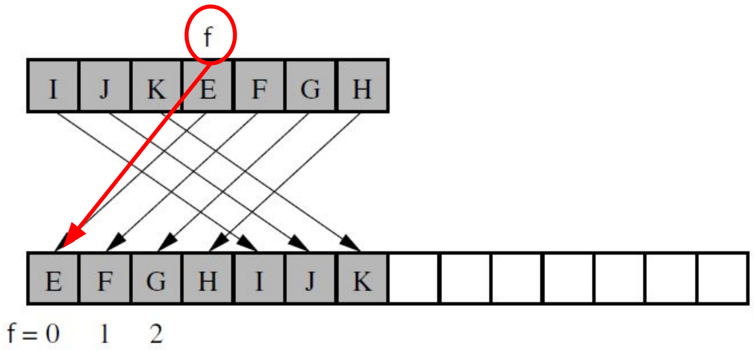
$$\text{avail} = (\text{self.\_front} + \text{self.\_size}) \ \% \ \text{len}(\text{self.\_data})$$

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

$$\textbf{self}.\text{\_front} = (\textbf{self}.\text{\_front} + 1) \ \% \ \text{len}(\textbf{self}.\text{\_data})$$

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

```python
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: $\sim \Theta(n)$

```python
Q = ArrayQueue()
Q.enqueue(5)
Q.enqueue(7)
Q.enqueue(9)
Q.enqueue(2)
Q.enqueue(6)
Q.enqueue(4)
Q.enqueue(1)
Q.enqueue(0)

print('Q: ', Q)
print('Queue Length:', len(Q))
print('Remove last item: ', Q.dequeue())
print('Remove last item: ', Q.dequeue())
print('Q: ', Q)
print('Queue Length:', len(Q))
print('Remove last item: ', Q.dequeue())
print('Remove last item: ', Q.dequeue())
print('Remove last item: ', Q.dequeue())
print('Remove last item: ', Q.dequeue())
print('Q: ', Q)
print('Queue Length:', len(Q))
print('Remove last item: ', Q.dequeue())
print('Q: ', Q)
print('Queue Length:', len(Q))
```

```
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:

$$\text{back} = (\text{self.\_front} + \text{self.\_size} - 1) \ \% \ \text{len(self.\_data)}$$

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

$$\text{self.\_front} = (\text{self.\_front} - 1) \ \% \ \text{len(self.\_data)}$$

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

Deques 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

```python
import collections
D = collections.deque()
D.appendleft(5)
D.appendleft(6)
D.append(10)
D.append(2)
D.appendleft(3)
D.appendleft(7)
print('Deque D: ', D)
print('Length: ', len(D))
D.rotate(5) #circularly shift rightward k steps
print('Deque D: ', D)
D.popleft()
D.pop()
print('Deque D: ', D)
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
```

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.

Summary

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