Stack

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). The name “stack” is derived from the metaphor of a stack of plates in a spring-loaded, cafeteria plate dispenser. In this case, the fundamental operations involve the “pushing” and “popping” of plates on the stack. When we need a new plate from the dispenser, we “pop” the top plate off the stack, and when we add a plate, we “push” it down on the stack to become the new top plate. Perhaps an even more amusing example is a PEZ® candy dispenser, which stores mint candies in a spring-loaded container that “pops” out the topmost candy in the stack when the top of the dispenser is lifted (see Figure 6.1). Stacks are a fundamental data structure. They are used in many applications, including the following.

- Internt Web Browsers - history function
- Text editors - Undo mechanism

The Stack Abstract Data Type (ADT)

Stacks are the simplest of all data structures, yet they are also among the most important. They are used in a host of different applications, and as a tool for many more sophisticated data structures and algorithms. Formally, a stack is an abstract data type (ADT) such that an instance S supports the following two methods:

Mehod	Function
`S.push(e)`	Add element e to the top of stack **S**.
`e = 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:

Mehod	Function
`e = 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.

```
class ArrayStack: # Space used: O(n)
    def __init__(self):
        self._data = []    # O(1)

    def __len__(self):
        return len(self._data) # O(1)


    def is_empty(self):
        return len(self._data) == 0   # O(1)

    def push(self, e):
        self._data.append(e)    # O(1)*

    def top(self):
        if self.is_empty():
            raise EmptyException('Stack is empty')
        return self._data[-1] # O(1)

    def pop(self):
        if self.is_empty():
            raise EmptyException('Stack is empty')
        return self._data.pop()   # O(1)*, same to self._data.pop(-1)
```

In [3]:

```
from dsLecture3 import ArrayStack, ArrayQueue
a = ArrayStack()
```

In [4]:

```
a.push('a')
a.display()
a.push('b')
a.display()
a.push('c')
a.display()
```

```
STACK: B|a|T
STACK: B|ab|T
STACK: B|abc|T
```

In [5]:

```
print(a.pop())
a.display()
print(a.top())
a.display()
print(a.pop())
print(a.pop())
a.display()
```

```
c
STACK: B|ab|T
b
STACK: B|ab|T
b
a
STACK: B||T
```

In [6]:

```
a.pop()
```

```
---------------------------------------------------------------------------
EmptyException                            Traceback (most recent call last)
<ipython-input-6-9c070c907602> in <module>
----> 1 a.pop()

~/Documents/Materials/Lecture/2020-Spring DGIST Data Structure/Lecture/Week
4/notebooks/dsLecture3.py in pop(self)
     24     def pop(self):
     25         if self.is_empty():
---> 26             raise EmptyException('Stack is empty')
     27         return self._data.pop()  # O(1)*, same to self._data.pop(-1)
     28

EmptyException: Stack is empty
```

Queue

Another fundamental data structure is the ***queue***. It is a close “cousin” of the stack, as a queue is a collection of objects that are inserted and removed according to the ***first-in, first-out (FIFO)*** principle. That is, elements can be inserted at any time, but only the element that has been in the queue the longest can be next removed.

We usually say that elements enter a queue at the back and are removed from the front. A metaphor for this terminology is a line of people waiting to get on an amusement park ride. People waiting for such a ride enter at the back of the line and get on the ride from the front of the line. There are many other applications of queues (see Figure 6.4). Stores, theaters, reservation centers, and other similar services typically process customer requests according to the FIFO principle. A queue would therefore be a logical choice for a data structure to handle calls to a customer service center, or a wait-list at a restaurant. FIFO queues are also used by many computing devices, such as a networked printer, or a Web server responding to requests.

The Queue Abstract Data Type (ADT)

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. This restriction enforces the rule that items are inserted and deleted in a queue accord- ing to the first-in, first-out (FIFO) principle. The queue abstract data type (ADT) supports the following two fundamental methods for a queue Q:

Mehod	Function
`Q.euqueue(e)`	Add element e to the back of queue **Q**.
`e = Q.dequeue()`	Remove and return the first element from queue **Q**; an error occurs if the queue is empty.

The queue ADT also includes the following supporting methods (with first being analogous to the stack's top method):

Mehod	Function
`e = 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(S)`	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, and that there is no a priori bound on the capacity of the queue. Elements added to the queue can have arbitrary type.

```
class ArrayQueue: # Space used: O(n)
    DEFAULT_CAPACITY = 10

    def __init__(self, capacity=DEFAULT_CAPACITY):
        self._data = [None] * capacity
        self._size = 0
        self._front = 0

    def __len__(self): # O(1)
        return self._size

    def is_empty(self): # O(1)
        return self._size == 0

    def first(self): # O(1)
        if self.is_empty():
            raise EmptyException('Queue is empty')
        return self._data[self._front]

    def dequeue(self):    # O(1)*
        if self.is_empty():
            raise EmptyException('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

    def enqueue(self, e): # O(1)*
        if self._size == len(self._data):
            self._resize(2*len(self._data))
        avail = (self._front + self._size) % len(self._data)
        self._data[avail] = e
        self._size += 1

    def _resize(self, cap): # O(n)
        old = self._data
        self._data = [None] * cap
        walk = self._front
```

```
        for k in range(self._size):
            self._data[k] = old[walk]
            walk = (1+walk) % len(old)
        self._front = 0
```

In [7]:

```
b = ArrayQueue(2)
```

In [8]:

```
b.enqueue('a')
b.display()
b.enqueue('b')
b.display()
```

```
QUEUE: |a.|
        ^
QUEUE: |ab|
        ^
```

In [9]:

```
print(b.dequeue())
b.display()
print(b.dequeue())
b.display()
```

```
a
QUEUE: |.b|
         ^
b
QUEUE: |..|
        ^
```

In [10]:

```
b.enqueue('a')
b.display()
b.enqueue('b')
b.display()
b.enqueue('c')
b.display()
b.enqueue('d')
b.display()
b.enqueue('e')
b.display()
print(b.dequeue())
b.display()
print(b.dequeue())
b.display()
print(b.dequeue())
b.display()
print(b.dequeue())
b.display()
print(b.dequeue())
b.display()
```

```
QUEUE: |a.|
        ^
QUEUE: |ab|
        ^
QUEUE: |abc.|
        ^
QUEUE: |abcd|
        ^
QUEUE: |abcde...|
        ^
```

```
a
QUEUE: |.bcde...|
         ^
b
QUEUE: |..cde...|
          ^
c
QUEUE: |...de...|
           ^
d
QUEUE: |....e...|
            ^
e
QUEUE: |........|
             ^
```

In [11]:

```
b.enqueue('a')
b.display()
b.enqueue('b')
b.display()
b.enqueue('c')
b.display()
b.enqueue('d')
b.display()
b.enqueue('e')
b.display()
print(b.dequeue())
b.display()
print(b.dequeue())
b.display()
print(b.dequeue())
b.display()
print(b.dequeue())
b.display()
print(b.dequeue())
b.display()
```

```
QUEUE: |.....a..|
              ^
QUEUE: |.....ab.|
              ^
QUEUE: |.....abc|
              ^
QUEUE: |d....abc|
              ^
QUEUE: |de...abc|
              ^
a
QUEUE: |de....bc|
               ^
b
QUEUE: |de.....c|
                ^
c
QUEUE: |de......|
        ^
d
QUEUE: |.e......|
          ^
e
QUEUE: |........|
           ^
```

In [12]:

```
b.enqueue('a')
b.display()
b.enqueue('b')
b.display()
b.enqueue('c')
b.display()
b.enqueue('d')
b.display()
```

```
b.enqueue('e')
b.display()
print(b.dequeue())
b.display()
print(b.dequeue())
b.display()
print(b.dequeue())
b.display()
print(b.dequeue())
b.display()
print(b.dequeue())
b.display()
```

```
QUEUE: |..a.....|
          ^
QUEUE: |..ab....|
          ^
QUEUE: |..abc...|
          ^
QUEUE: |..abcd..|
          ^
QUEUE: |..abcde.|
          ^
a
QUEUE: |...bcde.|
           ^
b
QUEUE: |....cde.|
            ^
c
QUEUE: |.....de.|
             ^
d
QUEUE: |......e.|
              ^
e
QUEUE: |........|
               ^
```

In [13]:

```
b.enqueue('a')
b.display()
b.enqueue('b')
b.display()
b.enqueue('c')
b.display()
b.enqueue('d')
b.display()
b.enqueue('e')
b.display()
print(b.dequeue())
b.display()
print(b.dequeue())
b.display()
print(b.dequeue())
b.display()
print(b.dequeue())
b.display()
print(b.dequeue())
b.display()
```

```
QUEUE: |.......a|
                ^
QUEUE: |b......a|
                ^
QUEUE: |bc.....a|
                ^
QUEUE: |bcd....a|
                ^
QUEUE: |bcde...a|
                ^
a
QUEUE: |bcde....|
```

```
        ^
b
QUEUE: |.cde....|
         ^
c
QUEUE: |..de....|
           ^
d
QUEUE: |...e....|
             ^
e
QUEUE: |........|
              ^
```

In [14]:

```
b.dequeue()
```

```
---------------------------------------------------------------------------
EmptyException                            Traceback (most recent call last)
<ipython-input-14-63b240b5c5de> in <module>
----> 1 b.dequeue()

~/Documents/Materials/Lecture/2020-Spring DGIST Data Structure/Lecture/Week
4/notebooks/dsLecture3.py in dequeue(self)
     58         if self.is_empty():
     59             raise EmptyException('Queue is empty')
---> 60         answer = self._data[self._front]
     61         self._data[self._front] = None  # help garbage collection
     62

EmptyException: Queue is empty
```

Double-Ended Queue (Deque)

We next consider a queue-like data structure that supports insertion and deletion at both the front and the back of the queue. Such a structure is called a ***double-ended queue, or deque***, which is usally pronounced "deck" to avoid confusion with the dequeue method of the regular queue ADT, which is pronounced like the abbreviation "D.Q."

The Deque Abstract Data Type (ADT)

To provide a symmetrical abstraction, the deque ADT is defined so that deque D supports the following methods:

Mehod	Function
`D.add_first(e)`	Add element e to the fornt of deque **D**.
`D.add_last(e)`	Add element e to the back of deque **D**.
`e = D.delete_first()`	Remove and return the first element from deque **D**; an error occurs if the deque is empty.
`e = 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:

Mehod	Function
`e = D.first()`	Return (but do not remove) the first element of deque **D**; an error occurs if the deque is empty.
`e = 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__`.

HOMEWORK: Implement Deque using a Circular Array