# Implementations

---

## Stack

- Python list
- Linked list

### List Implementation

The elements in the stack have an order to them, so using a sequence makes sense. We can use a Python list. Due to the way Python manages memory for lists, we should add and delete at the end of the list to do it most efficiently.

- We add to the end of the list
- We take from the end of the list

#### Code

```
class Stack(object):
    def__init__(self):
        self._lst = []

    def push(self, element):
        self._lst.append(element)

    def pop(self):
        if len(self._lst) == 0:
            return None
        return self._lst.pop()

    def top(self):
        if len(self._lst) == 0:
            return None
        return self._lst[-1]

    def length(self):
        return len(self._lst)

    def is_empty(self):
        return len(self._lst) == 0
```

## Complexity

- `List.append` and `List.pop` are O(1) on average (due to Python's memory reshuffling).
    - Our `push` and `pop` methods are O(1) on average.
- List index lookup is O(1).
    - Our `top` method is O(1).
- List length is O(1).
    - Our `length` method is O(1).

- Our `is_empty` method is O(1).

## Linked List Implementation

```
def add_first(self, element):
    node = SSLNode(element, self.first)
    self.first = node
    self.size = self.size + 1

def get_first(self):
    if self.size == 0:
        return None
    return self.first.element

def remove_first(self):
    if self.size == 0:
        return None
    item = self.first.element
    self.first = self.first.next
    self.size = self.size - 1
    return item

def add_last(self, element):
    newnode = SLLNode(element, None)
    if self.first == None:
        self.first = newnode
    else:
        node = self.first
        while node.next is not None:
            node = node.next
        node.next = newnode
    self.size += 1
```

```
def get_last(self):
    if self.size == 0:
        return None
    node = self.first
    while node.next is not None:
        node = node.next
    return node.element

def remove_last(self):
    if self.size == 0:
        return None
    node = self.first
    while node.next is not None:
        node = node.next

def push(self, item):
    self.add_first(item)

def pop(self):
    return self.remove_first()

def top(self):
    return self.get_first()

def length(self):
    return self.size

def is_empty(self):
    return self.size == 0
```

# Queue

The elements clearly have an order, so we can use a sequence.

## List Implementation

- We'll maintain a head and tail reference.
- Every time we add, we move the tail reference forward one position.
  - If the reference would go beyond the end of the list, we wrap around to the start.
- Every time we remove, we move the head reference forward one position and replace the removed item with `None`.
  - If the reference would go beyond the end of the list, we wrap around to the start.
- If our list becomes full, we grow the list to twice the size.
  - When we do this, we might as well undo the wrapping by putting the head element back at the start of the new list.

### Code

```
class Queue:
    def __init__(self):
        self.body = [None] * 10
        self.front = 0
        self.size = 0

    def __str__(self):
        output = '<-'
        i = self.front

        while i != (self.front + self.size) %
len(self.body):
            output += str(self.body[i]) + '-'
            i = (i + 1) % len(self.body)
```

```
        output = output + '<'
        return output

    def grow(self):
        oldbody = self.body
        self.body = [None] * (2*len(self.body))
        oldpos = self.front
        pos = 0

        for _ in range(self.size):
            self.body[pos] = oldbody[oldpos]
            oldbody[oldpos] = None
            pos += 1
            oldpos = (oldpos + 1) % len(oldbody)

        self.front = 0

    def shrink(self):
        oldbody = self.body
        self.body = [None] * (math.ceil(0.5 *
len(self.body)))
        oldpos = self.front
        pos = 0

        for _ in range(self.size):
            self.body[pos] = oldbody[oldpos]
            oldbody[oldpos] = None
            pos += 1
            oldpos = (oldpos + 1) % len(oldbody)

        self.front = 0

    def enqueue(self,item):
```

```
        """Note: uses modular arithmetic to wrap around"""
        if self.size == 0:
            self.body[0] = item      #assumes an empty queue
has head at 0
            self.size = 1
        else:
            self.body[(self.front + self.size) %
len(self.body)] = item
            self.size += 1
            if self.size == len(self.body):  #list is now
full
                self.grow()                  #so grow it
ready for next enqueue

    def dequeue(self):
        if self.size == 0:     #empty queue
            return None
        item = self.body[self.front]
        self.body[self.front] = None
        if self.size == 1:               #just removed last
element, so rebalance
            self.front = 0
            self.size = 0
        self.front = (self.front + 1) % len(self.body)
        self.size = self.size - 1
        if self.size / len(self.body) < 0.25:
            self.shrink()

        return item

    def length(self):
        return self.size
```

```
    def first(self):
        return self.body[self.front]      # will return None
if queue is empty

    def is_empty(self):
        return self.size == 0
```

### Complexity

`enqueue` and `dequeue` are O(1) on average, and everything else is O(1)