THE CODE OF STACK CLASS

```python
class ListStack:

    def __init__(self):
        self.__data = list()

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

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

    def push(self,e):
        self.__data.append(e)

    def top(self):
        if self.is_empty():
            print('The stack is empty.')
        else:
            return self.__data[self.__len__() 1]

    def pop(self):
        if self.is_empty():
            print('The stack is empty.')
        else:
            return self.__data.pop()
```

THE CODE TO USE STACK CLASS

```python
def main():
    s = ListStack()
    print('The stack is empty? ',s.is_empty())
    s.push(100)
    s.push(200)
    s.push(300)
    print(s.top())
    print(s.pop())
    print(s.top())
```

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

SOLUTION

```python
from stack import ListStack

def is_matched_html(raw):
    s = ListStack()
    j = raw.find('<')

    while j!=-1:
        k = raw.find('>', j+1)
        if k==-1:
            return False
        tag = raw[j+1:k]

        if not tag.startswith('/'):
            s.push(tag)
        else:
            if s.is_empty():
                return False
            if tag[1:]!=s.pop():
                return False
        j = raw.find('<',k+1)

    return s.is_empty()
```

```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>
```

```python
def main():
    fhand = open('sampleHTML.txt','r')
    raw = fhand.read()
    print(raw)
    print(is_matched_html(raw))
```

ALGORITHM

Phase 1: Scanning the expression

The program scans the expression from left to right to extract operands, operators, and the parentheses.

1.1. If the extracted item is an operand, push it to **operandStack**.

1.2. If the extracted item is a + or - operator, process all the operators at the top of **operatorStack** and push the extracted operator to **operatorStack**.

1.3. If the extracted item is a * or / operator, process the * or / operators at the top of **operatorStack** and push the extracted operator to **operatorStack**.

1.4. If the extracted item is a (symbol, push it to **operatorStack**.

1.5. If the extracted item is a) symbol, repeatedly process the operators from the top of **operatorStack** until seeing the (symbol on the stack.

Phase 2: Clearing the stack

Repeatedly process the operators from the top of operatorStack until operatorStack is empty.

Expression	*Scan*	*Action*	operandStack	operatorStack
(1 + 2)*4 − 3	(	Phase 1.4		(
(1 + 2)*4 − 3	1	Phase 1.1	1	(
(1 + 2)*4 − 3	+	Phase 1.2	1	+ (
(1 + 2)*4 − 3	2	Phase 1.1	2 1	(
(1 + 2)*4 − 3	)	Phase 1.5	3	
(1 + 2)*4 − 3	*	Phase 1.3	3	*
(1 + 2)*4 − 3	4	Phase 1.1	4 3	*
(1 + 2)*4 − 3	−	Phase 1.2	12	−
(1 + 2)*4 − 3	3	Phase 1.1	3 12	−
(1 + 2)*4 − 3	none	Phase 2	9	

The queue class may contain the following methods:

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.

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.

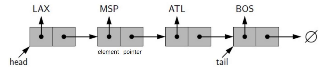
Q.is_empty(): Return True if queue Q does not contain any elements.

len(Q): Return the number of elements in queue Q; in Python, we implement this with the special method __len__.

```python
class ListQueue:
    default_capacity = 5

    def __init__(self):
        self.__data = [None]*ListQueue.default_capacity
        self.__size = 0
        self.__front = 0
        self.__end = 0

    def __len__(self):
        return self.__size

    def is_empty(self):
        return self.__size ==0

    def first(self):
        if self.is_empty():
            print('Queue is empty.')
        else:
            return self.__data[self.__front]

    def dequeue(self):

        if self.is_empty():
            print('Queue is empty.')
            return None

        answer = self.__data[self.__front]
        self.__data[self.__front] = None
        self.__front = (self.__front+1) \
                        % ListQueue.default_capacity
        self.__size -=1
        return answer

    def enqueue(self,e):
        if self.__size == ListQueue.default_capacity:
            print('The queue is full.')
            return None

        self.__data[self.__end] = e
        self.__end = (self.__end+1) \
                      % ListQueue.default_capacity
        self.__size += 1

    def outputQ(self):
        print(self.__data)
```

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

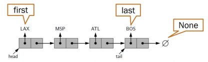
DEFINING A LINKED LIST IN PYTHON




```python
class Node:
    def __init__(self,element,pointer):
        self.element = element
        self.pointer = pointer

class LinkedList:

    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
```

```python
myList = LinkedList()

node1 = Node("BOS", None)
node2 = Node("ATL", node1)
node3 = Node("MSP", node2)
node4 = Node("LAX", node3)

myList.head = node4
myList.tail = node1
myList.size = 4
```

TRAVERSING THE LINKED LIST



- By starting at the head, and moving from one node to another by following each node's *pointer* reference, we can reach the tail of the list
- also known as link hopping or pointer hopping

```python
class Node:
    def __init__(self,element,pointer):
        self.element = element
        self.pointer = pointer

class LinkedList:

    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
```

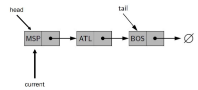
```python
myList = LinkedList()

node1 = Node("BOS", None)
node2 = Node("ATL", node1)
node3 = Node("MSP", node2)
node4 = Node("LAX", node3)

myList.head = node4
myList.tail = node1
myList.size = 4
```

```python
i = myList.size
current = myList.head
while i>0:
    print(current.element)
    current = current.pointer
    i-=1
```

COMPLETE CODE OF LINKED LIST

```python
class Node:
    def __init__(self,element,pointer):
        self.element = element
        self.pointer = pointer

class LinkedList:

    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def add_first(self,e):
        newest = Node(e,None)
        newest.pointer = self.head
        self.head = newest
        self.size = self.size+1

        if self.size == 1:
            self.tail = newest

    def add_last(self,e):
        newest = Node(e,None)
        if self.size > 0:
            self.tail.pointer = newest
        else:
            self.head = newest
        self.tail = newest
        self.size = self.size+1

    def remove_first(self):
        if self.size == 0:
            print('The linked list is empty')
        elif self.size == 1:
            answer = self.head.element
            self.head = None
            self.tail = None
            self.size -= 1
            return answer
        else:
            answer = self.head.element
            self.head = self.head.pointer
            self.size = self.size - 1
            return answer
```

REMOVING AN ELEMENT FROM THE TAIL OF A SINGLY LINKED LIST




```python
def remove_last(self):
    if self.size==0:
        print('The linked list is empty')
    elif self.size ==1:
        answer = self.tail.element
        self.head = None
        self.tail = None
        self.size = 0
        return answer
    else:
        answer = self.tail.element
        current = self.head
        while current.pointer != self.tail:
            current = current.pointer
        self.tail = current
        self.tail.pointer = None
        self.size = self.size - 1
        return answer
```

complexity: O(n)

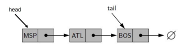
we cannot efficiently delete a node at the tail of the list

香港中文大學（深圳）

COMPLEXITY COMPARISON

Methods	Compact Array	Singly Linked List
add(e: E)	$O(1)$	$O(1)$
add(index: int, e: E)	$O(n)$	$O(n)$
clear()	$O(1)$	$O(1)$
contains(e: E)	$O(n)$	$O(n)$
get(index: int)	$O(1)$	$O(n)$
indexOf(e: E)	$O(n)$	$O(n)$
isEmpty()	$O(1)$	$O(1)$
lastIndexOf(e: E)	$O(n)$	$O(n)$
remove(e: E)	$O(n)$	$O(n)$
size()	$O(1)$	$O(1)$
remove(index: int)	$O(n)$	$O(n)$
set(index: int, e: E)	$O(1)$	$O(n)$
addFirst(e: E)	$O(n)$	$O(1)$
removeFirst()	$O(n)$	$O(1)$

MSP	ATL	BOS
0	1	2



SOLUTION: IMPLEMENT STACK WITH A SINGLY LINKED LIST

```python
class Node:
    def __init__(self, element, pointer):
        self.element = element
        self.pointer = pointer

class LinkedStack:

    def __init__(self):
        self.head = None
        self.size = 0

    def __len__(self):
        return self.size

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

    def push(self,e):
        self.head = Node(e,self.head)
        self.size += 1

    def top(self):
        if self.is_empty():
            print('Stack is empty.')
        else:
            return self.head.element

    def pop(self):
        if self.is_empty():
            print('Stack is empty.')
        else:
            answer = self.head.element
            self.head = self.head.pointer
            self.size -=1
            return answer
```

SOLUTION: IMPLEMENT QUEUE WITH A SINGLY LINKED LIST

```python
class LinkedQueue:

    def __init__(self):
        self. head = None
        self. tail = None
        self. size = 0

    def __len__(self):
        return self.size

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

    def first(self):
        if self.is_empty():
            print('Queue is empty.')
        else:
            return self.head.element

    def dequeue(self):
        if self.is_empty():
            print('Queue is empty.')
        else:
            answer = self.head.element
            self.head = self.head.pointer
            self.size -= 1
            if self.is_empty():
                self.tail = None
            return answer

    def enqueue(self,e):
        newest = Node(e,None)

        if self.is_empty():
            self.head = newest
        else:
            self.tail.pointer = newest
        self.tail = newest
        self.size += 1
```

IMPLEMENT A QUEUE WITH A CIRCULARLY LINKED LIST

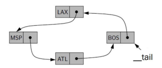
```python
class Node:
    def __init__(self, element, pointer):
        self.element = element
        self.pointer = pointer

class CQueue:

    def __init__(self):
        self.__tail = None
        self.__size = 0

    def __len__(self):
        return self.__size

    def is_empty(self):
        return self.__size == 0

    def first(self):

        if self.is_empty():
            print('Queue is empty.')
        else:
            head = self.__tail.pointer
            return head.element
```

Commonly used HTML tags:

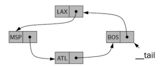
- body: document body
- h1: section header
- center: center justify
- p: paragraph
- ol: numbered (ordered) list
- li: list item

IMPLEMENT A QUEUE WITH A CIRCULARLY LINKED LIST

```python
def dequeue(self):
    if self.is_empty():
        print('Queue is empty.')
    else:
        oldhead = self.__tail.pointer
        if self.__size == 1:
            self.__tail = None
        else:
            self.__tail.pointer = oldhead.pointer
        self.__size -= 1
        return oldhead.element
```




```python
def enqueue(self,e):
    newest = Node(e,None)
    if self.is_empty():
        newest.pointer = newest
    else:
        newest.pointer = self.__tail.pointer
        self.__tail.pointer = newest
    self.__tail = newest
    self.__size += 1
```



CODE FOR THE DOUBLY LINKED LIST

```python
class Node:
    def __init__(self,element,prev,nxt):
        self.element = element
        self.prev = prev
        self.nxt = nxt

class DLList:

    def __init__(self):
        self.header = Node(None,None,None)
        self.trailer = Node(None,None,None)
        self.header.nxt = self.trailer
        self.trailer.prev = self.header
        self.size = 0

    def __len__(self):
        return self.size

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

    def insert_between(self,e,predecessor,successor):
        newest = Node(e,predecessor,successor)
        predecessor.nxt = newest
        successor.prev = newest
        self.size+=1
        return newest

    def delete_node(self,node):
        predecessor = node.prev
        successor = node.nxt
        predecessor.nxt = successor
        successor.prev = predecessor
        self.size -=1
        element = node.element
        node.prev = node.nxt = node.element = None
        return element

    def iterate(self):
        pointer = self.header.nxt
        print('The elements in the list:')
        while pointer != self.trailer:
            print(pointer.element)
            pointer = pointer.nxt
```

SORT ALGORITHM
SOLUTION:

```python
from LinkedQueue import LinkedQueue

def LinkedBubble(q):
    listLength = q.size

    while listLength > 0:
        index = 0
        pointer = q.head
        while index < listLength-1:
            if pointer.element > pointer.pointer.element:
                buf = pointer.element
                pointer.element = pointer.pointer.element
                pointer.pointer.element = buf
            index += 1
            pointer = pointer.pointer
        listLength -= 1
    return q

def outputQ(q):
    pointer = q.head

    while pointer:
        print(pointer.element)
        pointer = pointer.pointer

def main():
    oldList = [9,8,6,10,45,67,21,1
    q = LinkedQueue()

    for i in oldList:
        q.enqueue(i)

    print('Before the sorting...')
    outputQ(q)

    q = LinkedBubble(q)
    print()
    print('After the sorting...')
    outputQ(q)
```