CSCI-SHU 210 Data Structures

Recitation 5 Stacks and Queues

Bad Queue Example

```
class  ArrayQueue():

        DEFAULT_CAPACITY  =  10

        def  __init__(self):
                self._data  =  []

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

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

        def  first(self):
                return  self._data[0]

        def  dequeue(self):
                return  self._data.pop(0)

        def  enqueue(self,  e):
                self._data.append(e)

        def  __str__(self):
                '''  You  can  simply  print  self._data  '''
                return  str(self._data)

def  main():
        #  Empty  Queue,  size  10.
        queue  =  ArrayQueue()

        #  Enqueue  0,  1,  2,  3,  4,  5,  6,  7
        for  i  in  range(8):
                queue.enqueue(i)
        print(queue)      #  [0,  1,  2,  3,  4,  5,  6,  7,  None,  None]

        #  Dequeue  5  times.
        for  j  in  range(5):
                queue.dequeue()
        print(queue)    #  [None,  None,  None,  None,  None,  5,  6,  7,  None,  None]

        #  Enqueue  8,  9,  10,  11,  12
        for  k  in  range(5):
```

```
            queue.enqueue(k + 8)
        print(queue)    # [10, 11, 12, None, None, 5, 6, 7, 8, 9]


if __name__ == '__main__':
        main()
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
[5, 6, 7]
[5, 6, 7, 8, 9, 10, 11, 12]
```

1. Array Queue

```
class ArrayQueue():

        DEFAULT_CAPACITY = 10

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

        def __len__(self):
                return self._size

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

        def first(self):
                if self.is_empty():
                        raise Exception("Queue is Empty")
                return self._data[self._front]

        def dequeue(self):
                if self.is_empty():
                        raise Exception("Queue is Empty")
                ans = self._data[self._front]
                self._data[self._front] = None
                self._front = (self._front + 1) % len(self._data)
                self._size -= 1
                return ans

        def enqueue(self, e):
                if self._size == len(self._data):
                        raise Exception("Queue is Full")
                loc = (self._front + self._size) % len(self._data)
                self._data[loc] = e
                self._size += 1

        def __str__(self):
                return str(self._data)
```

```
def  main():
        #  Empty  Queue,  size  10.
        queue  =  ArrayQueue()

        #  Enqueue  0,  1,  2,  3,  4,  5,  6,  7
        for  i  in  range(8):
                queue.enqueue(i)
        print(queue)      #  [0,  1,  2,  3,  4,  5,  6,  7,  None,  None]

        #  Dequeue  5  times.
        for  j  in  range(5):
                (queue.dequeue())
        print(queue)    #  [None,  None,  None,  None,  None,  5,  6,  7,  None,  None]

        #  Enqueue  8,  9,  10,  11,  12
        for  k  in  range(5):
                queue.enqueue(k  +  8)
        print(queue)    #  [10,  11,  12,  None,  None,  5,  6,  7,  8,  9]


if  __name__  ==  '__main__':
        main()
```

```
[0, 1, 2, 3, 4, 5, 6, 7, None, None]
[None, None, None, None, None, 5, 6, 7, None, None]
[10, 11, 12, None, None, 5, 6, 7, 8, 9]
```

2. Computing Spans

```
class  ArrayStack:
        '''  Stack  implemented  with  python  list  append/pop'''
        def  __init__(self):
                self.array  =  []

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

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

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

        def  top(self):
                if  self.is_empty():
                        raise  Exception()
                return  self.array[-1]

        def  pop(self):
                if  self.is_empty():
```

```
                        raise  Exception()
                return  self.array.pop(-1)

        def  __repr__(self):
                        return  str(self.array)

def  spans1(X):      #  O(N^2)
        ans  =  []
        for  i  in  range(len(X)):
                span  =  1
                while  i  -  span  >=  0  and  X[i  -  span]  <=  X[i]:
                        span  +=  1
                ans.append(span)
        return  ans

def  spans2(X):    #  O(N):  Think  about  how  many  times  a  stack  is  pushed  and  popped.  Th
        ans  =  []
        stack  =  ArrayStack()
        for  i  in  range(len(X)):
                while  len(stack)  >  0  and  X[stack.top()]  <=  X[i]:
                        stack.pop()

                if  len(stack)  ==  0:
                        ans.append(i  +  1)
                else:
                        ans.append(i  -  stack.top())

                stack.push(i)

        return  ans


def  main():
        print(spans1([6,3,4,5,2]))  #  [1,  1,  2,  3,  1]
        print(spans1([6,7,1,3,4,5,2]))    #  [1,  2,  1,  2,  3,  4,  1]
        print(spans2([6,3,4,5,2]))  #  [1,  1,  2,  3,  1]
        print(spans2([6,7,1,3,4,5,2]))    #  [1,  2,  1,  2,  3,  4,  1]

if  __name__  ==  '__main__':
        main()
```

```
[1, 1, 2, 3, 1]
[1, 2, 1, 2, 3, 4, 1]
[1, 1, 2, 3, 1]
[1, 2, 1, 2, 3, 4, 1]
```

3. Double ended queue

```
class  ArrayDeque:
        DEFAULT_CAPACITY  =  10
```

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

    def __len__(self):
        return self._size

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

    def is_full(self):
        return self._size == len(self._data)

    def first(self):
        if self.is_empty():
            raise Exception("Queue is empty")
        return self._data[self._front]

    def last(self):
        if self.is_empty():
            raise Exception("Queue is empty")
        loc = (self._front + self._size - 1) % len(self._data)
        return self._data[loc]

    def delete_first(self):
        if self.is_empty():
            raise Exception("Queue is empty")
        ans = self._data[self._front]
        self._data[self._front] = None
        self._front = (self._front + 1) % len(self._data)
        self._size -= 1
        return ans

    def add_first(self, e):
        if self.is_full():
            raise Exception("Queue is full")
        loc = (self._front - 1) % len(self._data)
        self._data[loc] = e
        self._front = (self._front - 1) % len(self._data)
        self._size += 1

    def delete_last(self):
        if self.is_empty():
            raise Exception("Queue is empty")
        loc = (self._front + self._size - 1) % len(self._data)
        ans = self._data[loc]
        self._data[loc] = None
        self._size -= 1
        return ans

    def add_last(self, e):
        if self.is_full():
            raise Exception("Queue is full")
        loc = (self._front + self._size) % len(self._data)
```

```
                self._data[loc] = e
                self._size += 1

        def __str__(self):
                return str(self._data)

def main():
        # Empty Queue, size 10.
        deque = ArrayDeque()

        # Add 0, 1, 2, 3 following FIFO.
        for i in range(4):
                deque.add_first(i)
        print(deque)    # [None, None, None, None, None, None, 3, 2, 1, 0]

        # Add 4, 5, 6, 7 following LIFO.
        for j in range(4):
                deque.add_last(j + 4)
        print(deque)    # [4, 5, 6, 7, None, None, 3, 2, 1, 0]

        # Remove first one
        print(deque.delete_first()) # 3

        # Remove last one
        print(deque.delete_last()) # 7


if __name__ == '__main__':
        main()
```

```
[None, None, None, None, None, None, 3, 2, 1, 0]
[4, 5, 6, 7, None, None, 3, 2, 1, 0]
3
7
```

4. Evaluation of arithmetic expressions

```
class ArrayStack:
        ''' Stack implemented with python list append/pop'''
        def __init__(self):
                self.array = []

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

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

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

        def top(self):
```

```
                if  self.is_empty():
                        raise  Exception()
                return  self.array[-1]

        def  pop(self):
                if  self.is_empty():
                        raise  Exception()
                return  self.array.pop(-1)

        def  __repr__(self):
                return  str(self.array)

def  compute(left,  right,  operator):
        if  (operator  ==  "+"):
                return  left  +  right
        elif  (operator  ==  "-"):
                return  left  -  right
        elif  (operator  ==  "/"):
                return  left  /  right
        elif  (operator  ==  "*"):
                return  left  *  right

def  evaluate(string):
        operator_stack  =  ArrayStack()
        operand_stack  =  ArrayStack()
        table  =  {"+":2,  "-":2,  "*":3,  "/":3,  "(":1,  ")":1}
        tokens  =  string.split()
        for  token  in  tokens:
                if  token  not  in  table.keys():    #  operand
                        operand_stack.push(int(token))

                elif  token  ==  '(':
                        operator_stack.push(token)

                elif  token  ==  ')':
                        operator  =  operator_stack.pop()
                        while  operator  !=  '(':
                                operand1  =  operand_stack.pop()
                                operand2  =  operand_stack.pop()
                                operand_stack.push(compute(operand2,  operand1,  operator))
                                operator  =  operator_stack.pop()

                else:      #  operator
                        while  (not  operator_stack.is_empty())  and  \
                        (table[operator_stack.top()]  >=  table[token]):
                                operand1  =  operand_stack.pop()
                                operand2  =  operand_stack.pop()
                                operator  =  operator_stack.pop()
                                operand_stack.push(compute(operand2,  operand1,  operator))
                        operator_stack.push(token)

        while  (not  operator_stack.is_empty()):
                operand1  =  operand_stack.pop()
                operand2  =  operand_stack.pop()
                operator  =  operator_stack.pop()
```

```
                operand_stack.push(compute(operand2,  operand1,  operator))

        return  operand_stack.pop()

if  __name__  ==  '__main__':
        print(evaluate("9  +  8  *  (  7  -  6  )  /  (  2  /  8  )"))    #41
        print(evaluate("9  +  8  *  7  /  (  6  +  5  )  -  (  4  +  3  )  *  2"))    #  0.09090909
        print(evaluate("9  +  8  *  7  /  (  (  6  +  5  )  -  (  4  +  3  )  *  2  )"))  #  -9.666
```

```
41.0
0.09090909090908994
-9.666666666666668
```