NYU, Tandon School of Engineering
CS-1134: Data Structures and Algorithms — Fall 2018

CS-1134 – Midterm Exam

Tuesday, November 20, 2018

- You have one hour and 15 minutes.

- There are 5 questions all together, with 100 points total.

- Write your Name and NetID at the top of each page.

- The exam has **TWO Parts**:
  1. The first part of the exam contains:
     - This cover page.
     - Documentation with the interface of the `ArrayStack`, `ArrayQueue`, and `DoublyLinkedList` classes that we implemented in the lectures.
       You may use these classes and methods without implementing them, unless explicitly stated otherwise**.**
     - A page for scratch work. **What you write in this page will not be graded**, but you must hand it in with your exam.
  2. The second part of the exam contains the questions you need to answer, and a space for you to write your answers at. Write your answers clearly and concisely, in those spaces.

- **YOU MAY NOT USE THE BACKSIDE OF THE EXAM PAPERS**, as they will not be looked at. Also, try to avoid writing near the edge of the page.
  If you need extra space for an answer, use the **extra page at the end of the exam** and **mark it clearly**, so we can find it when we're grading.

- If you write with a pencil, press hard enough so that the writing will show up when scanned.

- Calculators are not allowed.

- Read every question completely before answering it.

- For any questions about runtime, show the worst-case asymptotic runtime, using big-Theta notation.

- You do not have to do error checking. Assume all inputs to your functions are as described

- Cell phones, and any other electronic gadgets must be turned **off**.

- Do not talk to any students during the exam. If you truly do not understand what a question is asking, you may raise your hand when one of the CS1134 instructors is in the room.

```
class ArrayStack:

    def __init__(self):
        """initializes an empty ArrayStack object. A stack object has:
        data – an array, storing the elements currently in the
        stack in the order they entered the stack"""

    def __len__(self):
        """returns the number of elements stored in the stack"""

    def is_empty(self):
        """returns True if and only if the stack is empty"""

    def push(self, elem):
        """inserts elem to the stack"""

    def pop(self):
        """removes and returns the item that entered the stack
        last (out of all the items currently in the stack),
        or raises an Exception, if the stack is empty"""

    def top(self):
        """returns (without removing) the item that entered the
        stack last (out of all the items currently in the stack),
        or raises an Exception, if the stack is empty"""
```

```
class ArrayQueue:

    def __init__(self):
        """initializes an empty ArrayQueue object.
        A queue object has the following data members:
        1. data – an array, holding the elements currently in the
           queue in the order they entered the queue. The elements
           are stored in the array in a “circular” way (not necessarily
           starting at index 0)
        2. front_ind – holds the index, where the (cyclic) sequence
           starts, or None if the queue is empty
        3. num_of_elems – holds the number of elements that are
           currently stored in the queue"""

    def __len__(self):
        """returns the number of elements stored in the queue"""

    def is_empty(self):
        """returns True if and only if the queue is empty"""

    def enqueue(self, elem):
        """inserts elem to the queue"""

    def dequeue(self):
        """removes and returns the item that entered the queue
        first (out of all the items currently in the queue),
        or raises an Exception, if the queue is empty"""

    def first(self):
        """returns (without removing) the item that entered the
        queue first (out of all the items currently in the queue),
        or raises an Exception, if the queue is empty"""

    def resize(self, new_cap):
        """resizes the capacity of the self.data array to be
        new_cap, while preserving the current contents of the
        queue"""
```

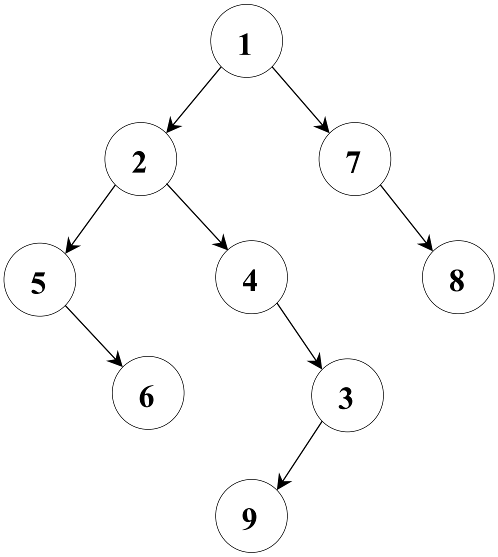
```
class DoublyLinkedList:

    class Node:
        def __init__(self, data=None, prev=None, next=None):
            """initializes a new Node object containing the
            following attributes:
            1. data - to store the current element
            2. next - a reference to the next node in the list
            3. prev - a reference to the previous node in the list """

        def disconnect(self):
            """detaches the node by setting all its attributes to None"""


    def __init__(self):
        """initializes an empty DoublyLinkedList object.
        A list object holds references to two "dummy" nodes:
        1. header - a node before the primary sequence
        2. trailer - a node after the primary sequence
        also a size count attribute is maintained"""

    def __len__(self):
        """returns the number of elements stored in the list"""

    def is_empty(self):
        """returns True if and only if the list is empty"""

    def first_node(self):
        """returns a reference to the node storing the
        first element in the list"""

    def last_node(self):
        """returns a reference to the node storing the
        last element in the list"""

    def add_after(self, node, data):
        """adds data to the list, after the element stored in node.
        returns a reference to the new node (containing data)"""

    def add_first(self, data):
        """adds data as the first element of the list"""

    def add_last(self, data):
        """adds data as the last element of the list"""

    def add_before(self, node, data):
        """adds data to the list, before the element stored in node.
        returns a reference to the new node (containing data)"""
```

```
def delete_node(self, node):
    """removes node from the list, and returns the data stored in it"""

def delete_first(self):
    """removes the first element from the list, and returns its value"""

def delete_last(self):
    """removes the last element from the list, and returns its value"""

def __iter__(self):
    """an iterator that allows iteration over the
    elements of the list from start to end"""

def __repr__(self):
    """returns a string representation of the list, showing
    data values separated by <--> """
```

Scratch
(This paper will not be graded)

Question 1 (15 points)

Give the preorder, postorder and inorder traversal sequences, for the following binary tree:



Preorder: 1,2,5,6,4,3,9,7,8

Postorder: 6,5,9,3,4,2,8,7,1

Inorder: 6,5,2,4,3,9,1,7,8

Question 2 (15 points)

a. Consider the algorithm we studied for evaluating postfix expressions:

```
def eval_postfix_exp(postfix_exp_str):
    operators = '+-*/'
    tokens_list = postfix_exp_str.split()
    args_stack = ArrayStack()
    for token in tokens_list:
        if token not in operators:
            args_stack.push(int(token))
        else:
            arg2 = args_stack.pop()
            arg1 = args_stack.pop()
            if (token == '+'):
                res = arg1 + arg2
            elif (token == '-'):
                res = arg1 - arg2
            elif (token == '*'):
                res = arg1 * arg2
            else: # token == '/'
                if (arg2 == 0):
                    raise ZeroDivisionError
                else:
                    res = arg1 / arg2
            args_stack.push(res)
    return args_stack.pop()
```

Under each token (number or operator symbol) in the postfix expression: "1 2 3 4 + 5 - * +", show what the stack looks like, **after** that token is processed:

1	**2**	**3**	**4**	**+**	**5**	**-**	*****	**+**
			4		5			
		3	3	7	7	2		
	2	2	2	2	2	2	4	
1	1	1	1	1	1	1	1	5

b. Write a **prefix** expression that is equivalent to "1 2 3 4 + 5 - * +"

+ 1 * 2 - + 4 3 5
____

Question 3 (20 points)

Implement the following method, that would be part of the `DoublyLinkedList` class:

```
    def move_to_end(self, node)
```

This method expects a parameter `node`, which is a reference to a node with one of the elements of the sequence that is represented in the list (`node` will not be the list's `header` or `trailer` nodes).
When called, the method should **mutate** the list object, by moving `node` to be the last element in the sequence (the node just before the `trailer`).
Note: This method does not return any value.

For example, if `lnk_lst` is `[2 <--> 4 <--> 6 <--> 8 <--> 10]`, and `node` is a reference to the node that holds `4`, after calling `lnk_lst.move_to_end(node)`, `lnk_lst` should be: `[2 <--> 6 <--> 8 <--> 10 <--> 4]` .

Implementation requirements:

1. Your implementation must run in **worst-case constant time**.
2. In this implementation, you are **not allowed** to create new `DoublyLinkedList.Node` objects, **nor to use** the `delete_node`, `delete_first`, `delete_last` `add_after`, `add_before`, `add_first` and the `add_last` methods of the `DoublyLinkedList` class.
   Instead, you **should** change the references `prev` and/or `next` for some nodes, to reflect the change in the order of the elements in the sequence.

Write your answer on the next page

```
def move_to_end(self, node):

    node.prev.next = node.next
    node.next.prev = node.prev

    node.next = self.trailer
    node.prev = self.trailer.prev

    self.trailer.prev.next = node
    self.trailer.prev = node
```

Question 4 (25 points)

Implement the following function:

```
def alternating_parity(lst)
```

This function is called with a list `lst`, containing $2n$ positive integers. Half of the numbers in `lst` are even, and half are odd.
When called, it should reorder the elements in `lst`, so that, at the end, the elements will be ordered in `lst` with alternating parity. Also, the **relative order** of the even numbers, and the relative order of the odd numbers, should **remain the same** as they were originally in `lst`.
That is, at the end:

- The first even number in `lst` (as it was ordered at the beginning of the call) should come first
- The first odd number in `lst` (as it was ordered at the beginning of the call) should come second
- The second even number in `lst` (as it was ordered at the beginning of the call) should come third
- The second odd number in `lst` (as it was ordered at the beginning of the call) should come forth
- Etc.

For example, if `lst = [2, 8, 1, 7, 3, 4]`, after calling `alternating_parity(lst)`, `lst` should be: `[2, 1, 8, 7, 4, 3]`

Implementation requirements:

1. Your function may only use:
   - One `ArrayQueue` object
   - One `ArrayStack` object
   - In addition to these two objects, you are allowed to use only $\theta(1)$ memory. That is, in addition to the queue and the stack, you may **not** use another data structure (such as a list, another stack, another queue, etc.) to store non-constant number of elements.

2. Your function should **run in worst-case linear time**.

Note: You should use the `ArrayQueue`, and `ArrayStack` objects as black boxes. That is, you may only use the interface provided by the methods these types support.

Write your answer on the next page

```
def alternating_parity(lst):
    queue = ArrayQueue()
    stack = ArrayStack()
    curr = True # True is even, False is odd
    count = 0
    while count < len(lst):
        elem = lst[count]
        if queue and ((curr is True and queue.first() % 2 == 0) or (curr is False and queue.first() % 2 != 0)):
            stack.push(queue.dequeue())
            curr = not curr
        elif (curr is True and elem % 2 == 0) or (curr is False and elem % 2 != 0):
            stack.push(elem)
            curr = not curr
            count += 1
        else:
            queue.enqueue(elem)
            count += 1
    while queue:
        stack.push(queue.dequeue())
    for i in range(len(lst)-1,-1,-1):
        lst[i] = stack.pop()
```

Question 5 (25 points) this question has 2 sections

A ***Flippable-Stack*** is an abstract data type that is like a regular stack, but in addition, it allows to flip (reverse) the order of the elements that are currently in it, so that the element at the bottom would become the top element, the second from the bottom would become the second top, etc.

A *Flippable-Stack* has the following operations:

- ***FlippableStack()*****:** creates a new *FlippableStack* object, with no elements in it
- ***is_empty()*****:** returns *false* if there are one or more items in the *FlippableStack*; *true* if there are no items in it
- ***push(item)*****:** inserts a new item at the top of the *FlippableStack*
- ***pop()*****:** removes and returns the item that is at the top of the *FlippableStack*
- ***top()*****:** returns (without removing) the item that is at the top of the *FlippableStack*
- ***flip()*****:** flips the order of the items that are currently in the *FlippableStack*

For example, you should expect the following interaction:

```
>>> fs = FlippableStack()
>>> fs.push(1)
>>> fs.push(2)
>>> fs.push(3)
>>> fs.push(4)
>>> fs.push(5)
>>> fs.pop()
5
>>> fs.pop()
4
```

Text

```
>>> fs.flip()
>>> fs.pop()
1
>>> fs.push(6)
>>> fs.pop()
6
>>> fs.pop()
2
>>> fs.pop()
3
```

A. Complete the implementation of the `FlippableStack` class.

Runtime requirement: All `FlippableStack` operations should run in $\theta(1)$ worst-case.

Notes:

1. You may use data types we implemented in class (such as `ArrayStack`, `ArrayQueue`, `DoublyLinkedList`), as data members in your implementation.
   a. Make sure to choose the most suitable data type, so you could satisfy the runtime requirement
   b. You can't change the implementation of any of these data types. You may only use them.
2. Make sure that your implementation for the `flip` method runs in constant worst-case time. As a friendly advice, you shouldn't change the actual order of all items, as that would take too much time.
3. If you need more space than what is provided, you are probably over complicating the implementation. However, in any case, do not write on the back of any page.

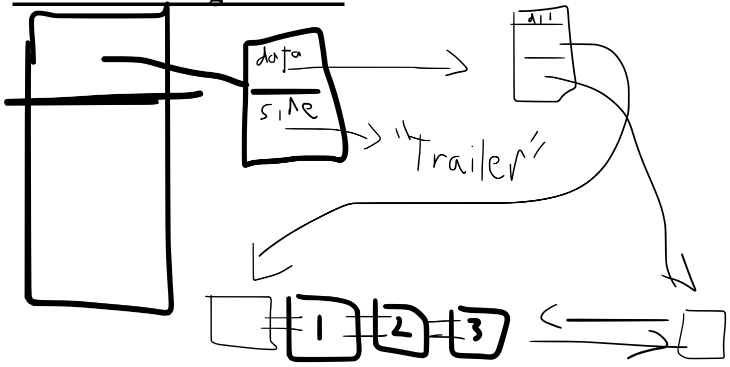
```
class FlippableStack:

    def __init__(self):
        self.data = DoublyLinkedList
        self.side = “Trailer”

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

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

    def push(self, item):
        if self.side == “Trailer”:
            self.data.add_last(item)
        else:
            self.data.add_first(item)

    def pop(self):
        if(self.is_empty()):
            raise Exception("FlippableStack is empty")
        if self.side == “Trailer”:
            return self.data.delete_last()
        else:
            return self.data.delete_first()
```

```
def top(self):
    if(self.is_empty()):
        raise Exception("FlippableStack is empty")
    if self.side == “Trailer”:
        return self.data.last_node().data
    else:
        return self.data.first_node().data



def flip(self):
    if self.side == “Trailer”:
        self.side = “Header”
    else:
        self.side = “Trailer”
```

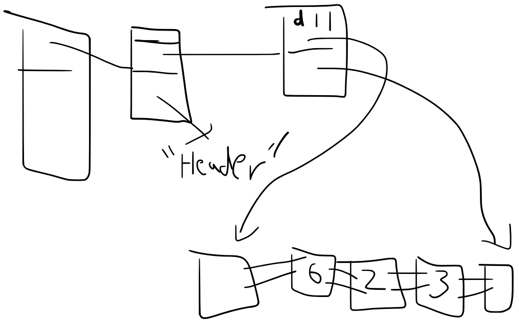
B. Draw the memory image, showing how the `fs` object, as you suggested in section (A), would look like, at **two points in time** (when running the following code):
   1. After executing only lines 1-8
   2. At the end of the execution (lines 1-11)

```
1.  fs = FlippableStack()
2.  fs.push(1)
3.  fs.push(2)
4.  fs.push(3)
5.  fs.push(4)
6.  fs.push(5)
7.  fs.pop()
8.  fs.pop()
9.  fs.flip()
10. fs.pop()
11. fs.push(6)
```

After executing lines 1-8:



After executing lines 1-11:



EXTRA PAGE IF NEEDED

Note question numbers of any questions or part of questions that you are answering here.
Also, write “ANSWER IS ON LAST PAGE” near the space provided for the answer.