A7: LINEAR DATA STRUCTURES

Bad programmers worry about the code. Good programmers worry about data structures and their relationships. — Linus Torvalds (Dec 28, 1969)

Course: CS 5002
Fall 2018
Due: November 4, 2018, Midnight

OBJECTIVES

After you complete this assignment, you will be comfortable with:

- The notion of a data structures
- The need for different data structures
- Linked lists
- Doubly linked lists
- Stacks
- Queues

RELEVANT READING

- Introduction to Data Structures
- Basic Data Structures in Python
- Linked List
- Doubly Linked List
- Stack
- Queue

NEXT WEEK'S READING

- Rosen
  - 6.1 The Basics of Counting
  - 6.2 The Pigeonhole Principle
  - 6.3 Permutations and Combinations
  - 6.4 Binomial Coefficients and Identities
  - 6.5 Generalized Permutations and Combinations

EXERCISES

Problem 1: What is a Linked List?

Determine whether or not the following statements are true for the linked list:

(a) (points) Linked list is a linear data structure with elements stored at a contiguous location.

(a) ______________________________

(b) (points) Linked list is a linear data structure with elements not stored at a contiguous location.

(b) ______________________________

(c) (points) All elements of a linked list are integers.

(c) ______________________________

(d) (points) Elements in a linked list are connected using pointers.

(d) ______________________________

Points: ______________ out of 3

(e) (points) An element of a linked list is a node, containing two attributes: value and pointer to the next element in the list.

(e) ____

(f) (points) An element of a linked list is a node, containing two attributes: value and pointer to the previous element in the list.

(f) ____

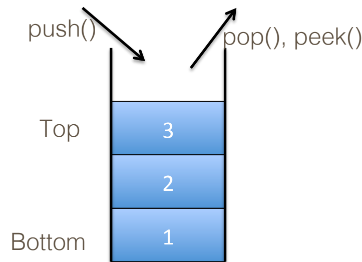


Figure 1: Graphical representation of a stack, used in Problem 2.

Problem 2: What is a Stack?

(a) (1 point) In class, we used a stack of book or a stack of pancakes as real life examples of stacks. Give three more real life examples of stacks.

(a) ____

(b) (1 point) Consider the following graphical representation of a stack, depicted in Figure 1. What is the size of the given stack?

(b) ____

(c) (1 point) Figure 1 also depicts operation `peek()`. What is the difference between operations `peek()` and `pop()`?

(c) ____

(d) (1 point) Redraw the stack from figure 1 after we execute operation `pop()` two times.

(e) (1 point) Redraw the stack from figure 1 after we execute operation `push(5)` two times.

Figure 2: Graphical representation of a stack, used in Problem 3.

Problem 3: What is a Queue?

(a) (1 point) In class, we used an example of people waiting in line as a real life example of a queue. Give three more real life examples of queues.

(a) ______________________________

(b) (1 point) Consider the following graphical representation of a queue, depicted in Figure 2. What is the size of the given queue?

(b) ______________________________

(c) (1 point) Redraw the queue from Figure 2, after we dequeue one element from the given queue, and enqueue element 20 into it.

(d) (1 point) Redraw the queue from part (c), after we dequeue two elements from it, and enqueue elements 25 and 30 into it.

(e) (1 point) Redraw the queue from part (d), after we dequeue three elements from it.

Problem 4: Simple Linked List Problem

Consider the following simple Python program, involving linked lists. State what gets printed out after every `print()` method call.

```
1  #A7-A1 Simple linked list problem
2
3  myLinkedList = [1, 2, 3, 4, 5];
4  print(myLinkedList);
5
6  myLinkedList.append(6);
7  print(myLinkedList);
8
9  myLinkedList.remove(4);
```

```
10 print(myLinkedList);
11
12 myLinkedList.insert(3, 55);
13 print(myLinkedList);
14
15 myLinkedList.reverse();
16 print(myLinkedList);
17
18 print(myLinkedList.__len__());
```

(a) In line 4?

(a) ____________________

(b) In line 7?

(b) ____________________

(c) In line 10?

(c) ____________________

(d) In line 13?

(d) ____________________

(e) In line 16?

(e) ____________________

(f) In line 18?

(f) ____________________

Problem 5: Simple Stack Problem

Consider the following simple Python program, containing our own implementation of a Stack, `MySimpleStack`. State what gets printed out after every `print()` method call.

```
1  class MySimpleStack:
2      def __init__(self):
3          self.items = []
4
5      def push(self, item):
6          self.items.append(item)
7
8      def pop(self):
9          return self.items.pop()
10
11 myTestStack = MySimpleStack();
12 myTestStack.push('This');
13 myTestStack.push('is');
14 myTestStack.push('my');
15 myTestStack.push('Simple');
16 myTestStack.push('Stack');
17
18 print(myTestStack.pop());
19 print(myTestStack.pop());
20 print(myTestStack.pop());
21 print(myTestStack.pop());
22 print(myTestStack.pop());
```

```
23
24 #print(mySimpleStack.pop());
```

(a) In line 18?

(a) ____________________

(b) In line 19?

(b) ____________________

(c) In line 20?

(c) ____________________

(d) In line 21?

(d) ____________________

(e) In line 22?

(e) ____________________

(f) If we uncomment line 24?

(f) ____________________

Problem 6: Simple Queue Problem

Consider the following simple Python program, containing our own implementation of a Queue, `MySimpleQueue`. State what gets printed out after every `print()` method call.

```
1  class MySimpleQueue:
2      def __init__(self):
3          self.items = []
4
5      def enqueue(self, item):
6          self.items.insert(0,item)
7
8      def dequeue(self):
9          return self.items.pop()
10
11 myTestQueue = MySimpleQueue();
12 myTestQueue.enqueue('This');
13 myTestQueue.enqueue('is');
14 myTestQueue.enqueue('my');
15 myTestQueue.enqueue('Simple');
16 myTestQueue.enqueue('Queue');
17
18 print(myTestQueue.dequeue());
19 print(myTestQueue.dequeue());
20 print(myTestQueue.dequeue());
21 print(myTestQueue.dequeue());
22 print(myTestQueue.dequeue());
23
24 #print(myTestQueue.dequeue());
```

(a) In line 18?

(a) ____________________

(b) In line 19?

(b) ____________________

(c) In line 20?

(c) ____________________

(d) In line 21?

(d) ____________________

(e) In line 22?

(e) ____________________

(f) If we uncomment line 24?

(f) ____________________

Problem 7: Stack ADT

Consider our simple implementation of a stack, `MySimpleStack`, repeated below:

```
1 class MySimpleStack:
2     def __init__(self):
3         self.items = []
4
5     def push(self, item):
6         self.items.append(item)
7
8     def pop(self):
9         return self.items.pop()
```

The given implementation is correct, but a bit too simple for us - we don't have a way to peek at the top of the stack, or to determine whether or not the stack is empty, or what its size might be.

This is where you come in. Please expand our simple implementation of a stack by implementing the following simple methods:

- `isEmpty()` - returns `True` if the stack is empty, and `False` otherwise.
- `size()` - returns the number of elements on the stack.
- `peek()` - returns the value of the element on the top of the stack, but does not remove it.

(a) (1 point) Method `isEmpty()`:

(b) (3 points) Method `size()`:

(c) (1 point) Method `peek()`:

Problem 8: List ADT

Consider our simple implementation of a list, `MySimpleList`, given below:

```
1  class MySimpleList:
2      def __init__(self):
3          self.head = None
4
5      def add(self, item):
6          temp = Node(item)
7          temp.setNext(self.head)
8          self.head = temp
9
10 class Node:
11     def __init__(self, initdata):
12         self.data = initdata
13         self.next = None
14
15     def getData(self):
16         return self.data
17
18     def getNext(self):
19         return self.next
20
21     def setData(self, newdata):
22         self.data = newdata
23
24     def setNext(self, newnext):
25         self.next = newnext
26
27 # myList = MySimpleList();
28 # myList.add(5);
29 # myList.add(6);
30
31 # print(myList.isEmpty());
32 # print(myList.size());
33 # print(myList.contains(5));
34 # print(myList.contains(7));
```

Once again, the given implementation is correct, but a bit too simple for us - we don't have a way to see whether or not our list contains some element, or to determine whether or not the list is empty, or what its size might be.

And again, this is where you come in. Please expand our simple implementation of a list by implementing the following simple methods:

- `isEmpty()` - returns `True` if the list is empty, and `False` otherwise.
- `size()` - returns the number of elements in the list.
- `contain(element)` - returns `True` if the list contains `element`, and `False` otherwise.

(a) (1 point) Method `isEmpty()`:

(b) (2 points) Method `size()`:

(c) (3 points) Method `contains()`:

Problem 9: Lists and Sets

A few weeks ago, we talked about sets, and we defined a set as a group of objects, usually with some relationship or similar property. As it turns out, sets are also an important data structure in computer science, with the distinguishing feature that **elements of a set are unique**.

Consider our simple, but incorrect implementation of a `Set`, provided below.

```
1  class MySimpleSet:
2      def __init__(self):
3          self.head = None
4  
5      def add(self, item):
6          temp = Node(item)
7          temp.setNext(self.head)
8          self.head = temp
9  
10     def remove(self, item):
11         current = self.head
12         previous = None
13         found = False
```

```
14          while not found:
15              if current.getData() == item:
16                  found = True
17              else:
18                  previous = current
19                  current = current.getNext()
20
21          if previous == None:
22              self.head = current.getNext()
23          else:
24              previous.setNext(current.getNext())
25
26  class Node:
27      def __init__(self, initdata):
28          self.data = initdata
29          self.next = None
30
31      def getData(self):
32          return self.data
33
34      def getNext(self):
35          return self.next
36
37      def setData(self, newdata):
38          self.data = newdata
39
40      def setNext(self, newnext):
41          self.next = newnext
42
43  # mySet = MySimpleSet();
44  # mySet.add(5);
45  # mySet.add(6);
46  # mySet.add(5);
```

(a) (3 points) As you can observe, there is currently no difference between our implementations of `List` and `Set`. That is intentional, because we want to implement a set using our simple linked list. Our implementation, however, is not correct, because it allows our set to have duplicate elements in a `Set`. Please explain how do we need to modify methods `add(item)` and `remove(item)` so that our implementation becomes correct.

(b) (3 points) Implement the necesssary changes in our code, so that our implementation of a `Set` becomes correct.

(c) (3 points) When thinking about sets as groups of objects, we know that we often care about finding a union or an intersection of two or more sets. Assuming we now have a correct implementation of a `Set`, please explain how would you implement a function `union(anotherSet)`, that finds a union of your current set, and the set `anotherSet`.

Problem 10: Lists and Matrices

A few weeks ago, we talked about matrices, and defined a matrix as a rectangular array of numbers. We said that a matrix with m rows and n columns is called an $m \times n$ matrix.

An important class of matrices in data science are so called **tall-and-skinny matrices**, defined as matrices such that the number of rows is much bigger than the number of columns ($m >> n$).

(a) (4 points) Consider the following simple implementation of a matrix, `SimpleMatrix`, provided below.

```
1  class MySimpleMatrix:
2      def __init__(self):
3          self.items = []
4
5      def initialize(self, items):
6          self.items = items
7
8      def isEmpty(self):
9          return self.items == []
10
11     def numRows(self):
12         return len(self.items)
13
14     def numColumns(self):
15         return len(self.items[0])
16
```

```
17  myMatrix = MySimpleMatrix();
18  myMatrix.initialize([[1, 2, 3], [2, 3, 4], [4, 5, 6], [5, 6, 7]]);
19  print(myMatrix.numRows());
20  print(myMatrix.numColumns());
```

Please explain how would you implement method `isTallSkinny(muchBigger)` that, given an input argument to define what "much bigger" means, returns `True` if the given matrix is tall and skinny, and `False` otherwise.

For example, given a 10×3, matrix, a function `isTallSkinny(muchBigger)` with `muchBigger` = 10 returns `False`, because it is not true that $m \geq 10 \cdot n$. On the other hand, if `muchBigger` = 2, the method returns `True`, because $m \geq 2 \cdot n$.

(b) (4 points) Modify the given simple implementation of a matrix, `SimpleMatrix`, to implement method `isTallSkinny(muchBigger)`.

Problem 11: Bag-Of-Words Model

A Bag-Of-Words model is one of the fundamental data structures in Natural Language Processing (NLP). In this model, some text is represented as a multiset (a bag) of its words, where we disregard grammar, and often also the order of words.

In this problem, a bag-of-words is a data collection, containing words (Strings), where the words do not have to be unique (i.e., duplicates are allowed), and there is no order within the collection.

(a) (4 points) Explain which of the data structures that we are already familiar with (list, stack, queue) could we use to implement our `BagOrWords`, and how might we do that.

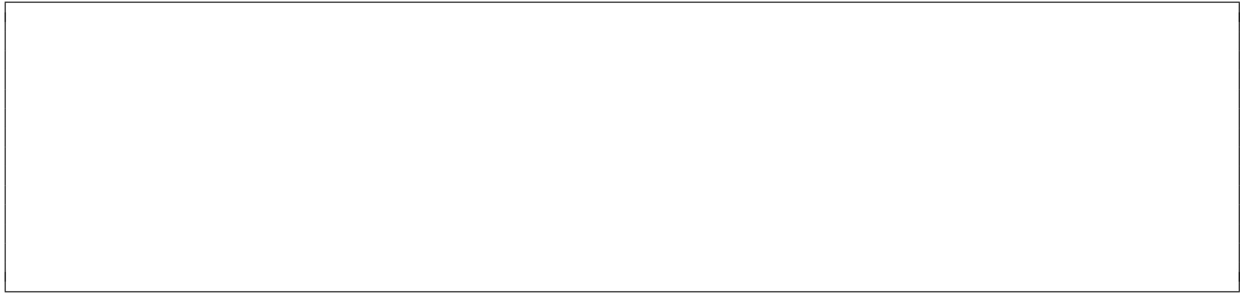
(b) (6 points) Implement the `BagOfWords` data structure, such that it includes at least the following methods:

- `add(item)`
- `remove(item)`
- `contains(item)`
- `size()`

Problem 12: Reversing a String

In many real life problems, we encounter the need to reverse some string. One way to do so is using a stack.

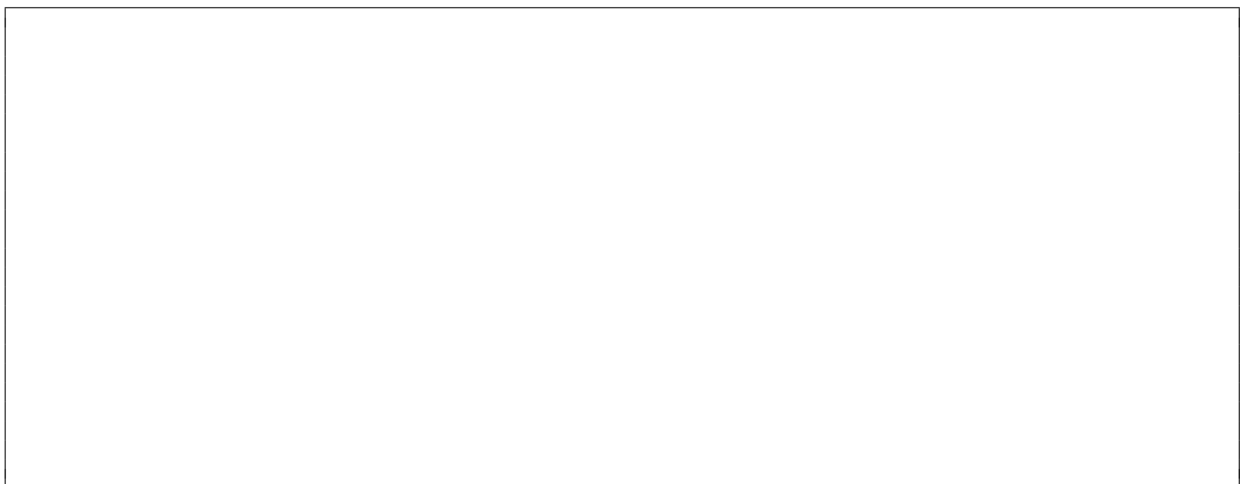
(a) (3 points) Explain how might we use a stack to reverse a string.



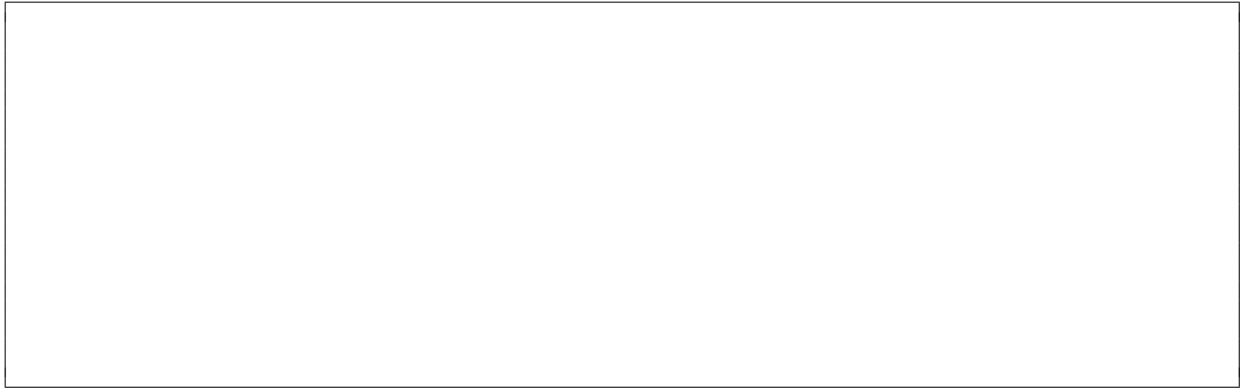
(b) (4 points) Consider our simple implementation of a stack, `MySimpleStack`, repreated here for convenience.

```
1  class MySimpleStack:
2      def __init__(self):
3          self.items = []
4
5      def push(self, item):
6          self.items.append(item)
7
8      def pop(self):
9          return self.items.pop()
10
11 myTestStack = MySimpleStack();
12 myTestStack.push('This');
13 myTestStack.push('is');
14 myTestStack.push('my');
15 myTestStack.push('Simple');
16 myTestStack.push('Stack');
17
18 print(myTestStack.pop());
19 print(myTestStack.pop());
20 print(myTestStack.pop());
21 print(myTestStack.pop());
22 print(myTestStack.pop());
```

Implement method `reverseString(string)` that reverses the given `string`, and prints out the reversed string, using our implementation of a stack, `MySimpleStack`.



(c) (3 points) A **palindrome** is a word or phrase that reads the same when read from left to right, as well as when read from right to left. Some examples of palindromes in English include: **mom, Anna, kayak, radar, level** and **noon**. Explain how might you use a stack to check whether or not some string is a palindrome.



Problem 13: Mixing Stacks and Queues

Consider the following code, containing our simple implementations of a stack and a queue, `MySimpleStack` and `MySimpleQueue`. State what gets printed out after every `prettyPrint()` method call.

```
1  class MySimpleStack:
2      def __init__(self):
3          self.items = []
4
5      def push(self, item):
6          self.items.append(item)
7
8      def pop(self):
9          return self.items.pop()
10
11     def prettyPrint(self):
12         for i in reversed(range (len(self.items))):
13             print(self.items[i])
14
15 class MySimpleQueue:
16     def __init__(self):
17         self.items = []
18
19     def enqueue(self, item):
20         self.items.insert(0, item)
21
22     def dequeue(self):
23         return self.items.pop()
24
25     def prettyPrint(self):
26         for i in reversed(range(len(self.items))):
27             print(self.items[i])
28
29
30 myStack = MySimpleStack();
31 myStack.push('This');
32 myStack.push('is');
33 myStack.push('Test');
34
35 myStack.prettyPrint();
36
37 myStack.pop()
38 myStack.prettyPrint();
39
40 myQueue = MySimpleQueue();
41 myQueue.enqueue(22);
42 myQueue.enqueue(55);
43 myQueue.enqueue(125);
44
45 myQueue.prettyPrint();
46
47 myQueue.dequeue();
```

```
48 myQueue.prettyPrint();
49
50 myNewStack = MySimpleStack();
51 myNewQueue = MySimpleQueue();
52
53 myNewQueue.enqueue('A');
54 myNewQueue.enqueue('B');
55 myNewQueue.enqueue('C');
56
57 print("Queue:")
58 myNewQueue.prettyPrint();
59
60 for i in reversed(range(len(myNewQueue.items))):
61     myNewStack.push(myNewQueue.dequeue());
62
63 print("Stack:")
64 myNewStack.prettyPrint();
```

(a) In line 35?

(a) ______________________________

(b) In line 38?

(b) ______________________________

(c) In line 45?

(c) ______________________________

(d) In line 48?

(d) ______________________________

(e) In line 58?

(e) ______________________________

(f) In line 64?

(f) ______________________________

Problem 14: Doubly Linked List

Similar to the linked list, a **doubly linked list** is a linked data structure that consists of sequentially linked records, referred to as **nodes**. In the doubly linked list, however, every node contains three fields:

- Value,
- Link to the previous node in the sequnece,
- Link to the next node in the sequence.

Consider the following code, containing our simple implementation of a doubly linked list, `SimpleDoublyLinkedList`. State what gets printed out after every `prettyPrint()` method call.

```
1 class Node(object):
2
3     def __init__(self, data, prev, next):
4         self.data = data
5         self.prev = prev
6         self.next = next
```

```
7
8
9  class SimpleDoublyLinkedList(object):
10     head = None
11     tail = None
12
13     def add(self, data):
14         temp = Node(data, None, None)
15         if self.head is None:
16             self.head = self.tail = temp
17         else:
18             temp.prev = self.tail
19             temp.next = None
20             self.tail.next = temp
21             self.tail = temp
22
23     def prettyPrint(self):
24         current_node = self.head
25         while current_node is not None:
26             print(current_node.data)
27             current_node = current_node.next
28
29 print("Case 1:");
30 myDLList = SimpleDoublyLinkedList();
31 myDLList.add(5);
32 myDLList.add(6);
33 myDLList.add(7);
34 myDLList.prettyPrint();
35
36 print("Case 2:");
37 myDLList.add(10);
38 myDLList.add(12);
39 myDLList.prettyPrint();
40
41 print("Case 3:");
42 myDLList.add(5);
43 myDLList.add(6);
44 myDLList.add(7);
45 myDLList.prettyPrint();
```

(a) (1 point) In line 33?

(a) ______________________________

(b) (1 point) In line 39?

(b) ______________________________

(c) (1 point) In line 45?

(d) (4 points) Explain how would you implement method `prettyPrintReverse()` that prints the whole doubly linked list in a reverse order.

(e) (4 points) Implement method `prettyPrintReverse()` that prints the whole doubly linked list in a reverse order.

Problem 15: Double-Ended Queue (Deque)

Deque, typically pronounced deck, is a double-ended-queue. It is a linear collection of elements that supports the insertion and removal of elements at both end points (front and rear). Deque is a richer abstract data type than both Stack and Queue because it implements both stacks and queues at the same time.

Consider the following code, containing our simple implementations of a deque, `MySimpleDeque`. State what gets printed out after every `prettyPrint()` method call.

```
1  class MySimpleDeque:
2      def __init__(self):
3          self.items = []
4
5      def isEmpty(self):
6          return self.items == []
7
8      def addFront(self, item):
9          self.items.append(item)
10
11     def addRear(self, item):
12         self.items.insert(0,item)
13
14     def removeFront(self):
15         return self.items.pop()
16
17     def removeRear(self):
18         return self.items.pop(0)
19
20     def size(self):
21         return len(self.items)
22
23     def prettyPrint(self):
24         for i in reversed(range(len(self.items))):
```

```
25              print(self.items[i])
26
27
28  myDeque = MySimpleDeque();
29  myDeque.addFront(1);
30  myDeque.addFront(2);
31  myDeque.addFront(3);
32
33  myDeque.prettyPrint();
34
35  myDeque.addRear(10);
36  myDeque.addRear(20);
37  myDeque.addRear(30);
38
39  myDeque.prettyPrint();
40
41  myDeque.removeFront();
42  myDeque.removeRear();
43
44  myDeque.prettyPrint();
45
46  myDeque.addRear(5);
47  myDeque.addRear(6);
48  myDeque.removeFront();
49  myDeque.addFront(7);
50
51  myDeque.prettyPrint();
52
53  myDeque.removeFront();
54  myDeque.removeRear();
55  myDeque.removeFront();
56  myDeque.removeRear();
57  myDeque.removeRear();
58
59  myDeque.prettyPrint();
60
61  myDeque.removeRear();
62  #myDeque.removeRear();
```

(a) (1 point) In line 33?

(a) ______________________________

(b) (1 point) In line 39?

(b) ______________________________

(c) (1 point) In line 44?

(c) ______________________________

(d) (1 point) In line 51?

(d) ______________________________

(e) (1 point) In line 59?

(e) ______________________________

(f) (1 point) What would happen if we uncomment line 62?

(f) ______________________________

(g) (2 points) Explain how would you implement method `sumFrontBack(flag)`, that removes the front and the back elements from the deque, sums them up, and depending on the `flag`, returns the sum back into the deque. If `flag = "Front"`, the sum is added to the front of the deque, if `flag = "Rear"`, the sum is added to the back of the deque. Otherwise, the sum is not added to the deque.

(h) (2 points) Implement method `sumFrontBack(flag)`, that removes the front and the back elements from the deque, sums them up, and depending on the `flag`, returns the sum back into the deque. If `flag = "Front"`, the sum is added to the front of the deque, if `flag = "Rear"`, the sum is added to the back of the deque. Otherwise, the sum is not added to the deque.

Question	Points	Score
What is a Linked List?	3	
What is a Stack?	5	
What is a Queue?	5	
Simple Linked List Problem	3	
Simple Stack Problem	6	
Simple Queue Problem	3	
Stack ADT	5	
List ADT	6	
Lists and Sets	9	
Lists and Matrices	8	
Bag-Of-Words Model	10	
Reversing a String	10	
Mixing Stacks and Queues	6	
Doubly Linked List	11	
Double-Ended Queue (Deque)	10	
Total:	100	

SUBMISSION DETAILS

Things to submit:

- Submit the following on Blackboard for Assignment 7:
  - The written parts of this assignment as a .pdf named "CS5002_[lastname]_A7.pdf". For example, my file would be named "CS5002_Bonaci_A7.pdf". (There should be no brackets around your name).
  - Make sure your name is in the document as well (e.g., written on the top of the first page).