- This lab will cover Linked Lists, Binary Trees, Depth-First Search, and Breadth-First Search.
- It is assumed that you have reviewed chapter 6, 7 & 8 of the textbook. You may want to refer to the text and your lecture notes during the lab as you solve the problems.
- When approaching the problems, think before you code. Doing so is good practice and can help you lay out possible solutions.
- Think of any possible test cases that can potentially cause your solution to fail!
- You must stay for the duration of the lab. If you finish early, you may help other students. If you don't finish by the end of the lab, we recommend you complete it on your own time. Ideally you should not spend more time than suggested for each problem.
- Your TAs are available to answer questions in lab, during office hours, and on Piazza.

Vitamins (maximum 50 minutes)

1. Draw the memory image of the linked list object as the following code executes: (5 minutes)

```
from DoublyLinkedList import DoublyLinkedList


dll = DoublyLinkedList()
dll.add_first(1)
dll.add_last(3)
dll.add_last(5)
dll.add_after(dll.header.next, 2)
dll.add_before(dll.trailer.prev, 4)
dll.delete_node(dll.trailer.prev)
dll.add_first(0)

print(dll)
```

What is the output of the code?

2. During lecture you learned about the different methods of a doubly linked list.

   Provide the following worst-case runtime for those methods:

   a. `def __len__(self):`

   b. `def is_empty(self):`

   c. `def add_after(self, node, data):`

   d. `def add_first(self, data):`

   e. `def add_last(self, data):`

   f. `def add_before(self, node, data):`

   g. `def delete_node(self, node):`

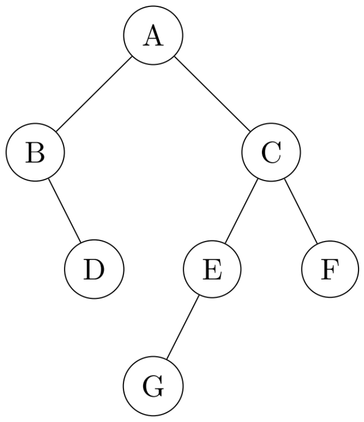
   h. `def delete_first(self):`

   i. `def delete_last(self):`

3. Trace the following function. What is the output of the following code using the doubly linked list from question 3? Give mystery an appropriate name.

```
#dll = Doubly Linked List
def mystery(dll):

    if len(dll) >= 2:
        node = dll.trailer.prev.prev
        node.prev.next = node.next
        node.next.prev = node.prev

        node.next = None
        node.prev = None
        return node

    else:
        raise Exception("dll must have length of 2 of
        greater")

print(mystery(dll))
```

4. Given the following binary tree, complete the following (5 minutes):



a. Give the preorder, inorder, and postorder traversals of the tree:

b. Give the level order traversal (Breadth-First) of the tree:

c. What is the height of the tree?

d. What are the depths of nodes E, B, and G?

5. Draw the expression tree for the following expressions (given in prefix, postfix, or infix): Remember that the numbers should be leaf nodes. (10 minutes)

   a. 3 4 - 2 + 5 *

   b. (3 * 2) + (4 / 6)

   c. / + 9 9 2

6. Draw the binary tree given the following traversals (10 minutes):

   preorder : 11, 6, 4, 5, 8, 10, 19, 17, 43, 31, 49

   inorder : 4, 5, 6, 8, 10, 11, 17, 19, 31, 43, 49

   Is it possible to draw a unique binary tree given only its preorder and postorder? If not, draw two trees with the same preorder and postorder traversal.

Coding

In this section, it is strongly recommended that you solve the problem on paper before writing code. For each problem, **you may not call any methods defined in the `LinkedBinaryTree` class. Specifically, you should manually traverse the tree in your function. Each node node of the tree contains the following references:** `data, left, right, parent`**.**

Download the **DoublyLinkedList.py & LinkedBinaryTree.py** files attached on NYU Brightspace.

1. In class, we defined the `stack ADT` using a dynamic array as the underlying data structure. Because of the resizing, the ArrayStack run-time for its operations is not exactly in Θ(1). Instead, the cost is Θ(1) amortized. (10 minutes)

   Define the stack ADT that guarantees each method to always run in Θ(1) **worst case**.

```
class LinkedStack:

    def __ init__(self):
        …

    def __len__(self):
    ''' Returns the number of elements in the stack. '''


    def is_empty(self):
    ''' Returns true if the stack is empty,false otherwise.
    '''


    def push(self, e):
    ''' Adds an element, e, to the top of the stack. '''


    def top(self):
    ''' Returns the element at the top of the stack.
        An exception is raised if the stack is empty. '''


    def pop(self):
```

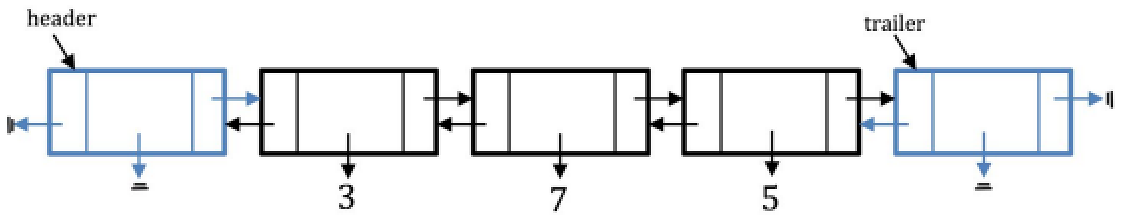
```
''' Removes and returns the element at the top of the
stack.
    An exception is raised if the stack is empty. '''
```

2. Implement the following method for the `DoublyLinkedList` class. The get item operator takes in an index, i, and returns the value at the ith node of the Doubly Linked List.

   You only need to support non-negative indices. Your solution should try to optimize the get item method. This means that you should decide whether to iterate from either the header or trailer based on whichever is closer to i. Index should start at 0. (30 minutes)

```
def __getitem__(self, i):
    '''Return the value at the ith node. If i is out of range,
    an IndexError is raised'''
```

   For example, if your Doubly Linked List looks like this:



```
print(dll[0]) # 3 (should iterate from the header)

print(dll[1]) # 7 (either way works)

print(dll[2]) # 5 (should iterate from the trailer)

print(dll[3]) # IndexError
```

   What is the worst-case run-time of the get item operator? Can we do better?

3. In homework 3, you were asked to implement a `MidStack` ADT using an ArrayStack and an ArrayDeque. This time, you will create the MidStack using a **Doubly Linked List** with Θ(1) extra space. All methods of this MidStack should have a Θ(1) run-time.

   The middle is defined as the $(n + 1)//2\ th$ element, where n is the number of elements in the stack.

   Hint: To access the middle of the stack in constant time, you may want to define an additional data member to reference the middle of the Doubly Linked List. (40 minutes)

```
class MidStack:

    def __ init__(self):
        self.data = DoublyLinkedList( )
        …

    def __len__(self):
    ''' Returns the number of elements in the stack. '''


    def is_empty(self):
    ''' Returns true if stack is empty and false otherwise.
    '''


    def push(self, e):
    ''' Adds an element, e, to the top of the stack. '''


    def top(self):
    ''' Returns the element at the top of the stack.
        An exception is raised if the stack is empty. '''


    def pop(self):
    ''' Removes and returns the element at the top of the
    stack.
        An exception is raised if the stack is empty. '''


    def mid_push(self, e):
    ''' Adds an element, e, to the middle of the stack.
```

```
            An exception is raised if the stack is empty. '''
```

4. Implement the `SinglyLinkedList`. The SinglyLinkedList differs from the DoublyLinkedList in that there is only a header and no trailer. In addition, each node only references the node after it. The last node in the linked list will reference its next Node as `None` (this would've been self.trailer for a DoublyLinkedList).

> **You may add additional data members that are O(1) extra space.** Analyze the run-time of each method after completing the implementation. Is it possible to make add_last and or delete_last work in O(1) constant run-time? (40 minutes)

```
class SinglyLinkedList:
    class Node:
        def __init__(self, data=None, next=None):
            self.data = data
            self.next = next

        def disconnect(self):
            self.data = None
            self.next = None


    def __init__(self):
        self.header = SinglyLinkedList.Node()
        self.size = 0

    def __len__(self):
        return self.size

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

    def add_after(self, node, val):
     ''' Creates a new node containing val as its data and adds
     it after an existing node in the SinglyLinkedList'''

    def add_before(self, node, val):
     ''' Creates a new node containing val as its data and adds
     it before an existing node in the SinglyLinkedList'''

    def add_first(self, val):
     ''' Creates a new node containing val as its data and adds
     it to the front of the SinglyLinkedList'''
```

```
        def add_last(self, val):
         ''' Creates a new node containing val as its data and adds
         it to the back of the SinglyLinkedList'''


        def delete_node(self, node):
         ''' Removes an existing node from the SinglyLinkedList and
         returns its value'''

        def delete_first(self):
         ''' Removes an existing node from the front of the
         SinglyLinkedList and returns its value'''

        def delete_last(self):
         ''' Removes an existing node from the back of the
         SinglyLinkedList and returns its value'''

        def __iter__(self):
            cursor = self.header.next
            while(cursor is not None):
                yield cursor.data
                cursor = cursor.next

        def __repr__(self):
            return "[" + " -> ".join([str(elem) for elem in self])
    + "]"
```

5. Write a *recursive* function that returns the sum of all even integers in a `LinkedBinaryTree`. Your function should take one parameter, *root* node. You may assume that the tree only contains integers. (5 minutes)

```
    def bt_even_sum(root):
         ''' Returns the sum of all even integers in the binary
         tree'''
```

6. Write a *recursive* function that determines whether or not a value exists in a `LinkedBinaryTree`. Your function should take two parameters, a *root* node and *val*, and return True if *val* exists or False if not. (10 minutes)

```
def bt_contains(root, val):
    ''' Returns True if val exists in the binary tree and
    false if not'''
```

7. Write a function that will add (merge) two `LinkedBinaryTree`. The function will take two parameters, root1, root2, of 2 binary trees and return the root of a new binary tree containing nodes created by merging each node of the same positions from the original trees. If only one node exists in a specific position, simply use that value as the node for the new tree. **You may also define additional helper functions.** (25 minutes)

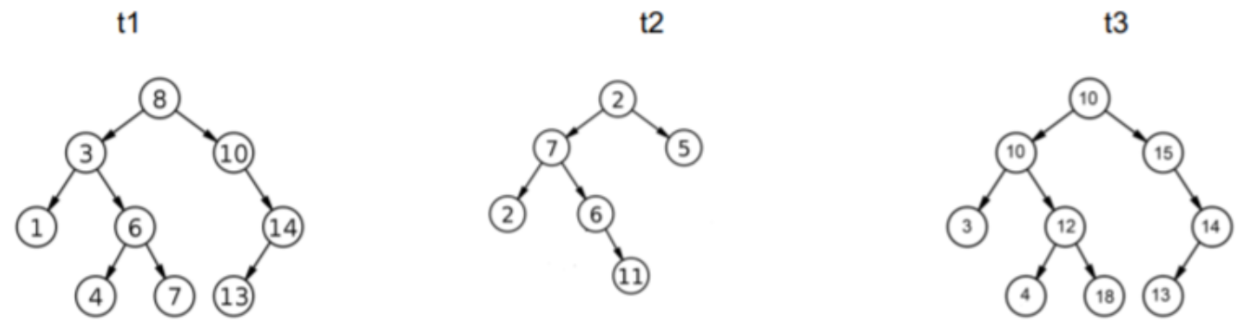
```
def add_bts(root1, root2):
     ''' Creates a new binary tree merging tree1 and tree2
     and returns its root. '''
```

In the following example, notice that the new root is 10, the result of merging root1 and root2 (8 + 2).

In the case where there is only one node at a given position (14 in t1, there is no corresponding node in t2), the value 14 is used instead for t3.

The root node of t3, containing 10, should be returned by the add_bts function.

Note that all of the nodes must be newly created. t3 should not have any nodes referencing a subtree of t1 or t2.



8. Write a function that will invert a `LinkedBinaryTree` in-place (mutate the input tree)

You will write two implementations, one recursive, and one non recursive.

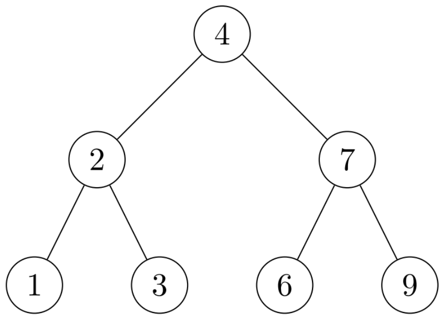
Both functions will be given a parameter, *root*, which is the root node of the binary tree.
(25 minutes)

```
def invert_bt(root):
     ''' Inverts the binary tree using recursion '''
```

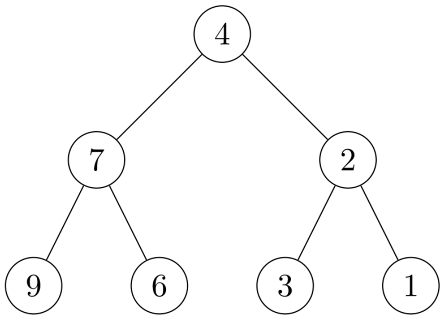
```
def invert_bt(root):
     ''' Inverts the binary tree without recursion '''
```

<u>Hint:</u> For the non recursive implementation, you should use the *breadth-first search* with an ArrayQueue.

ex)



will become



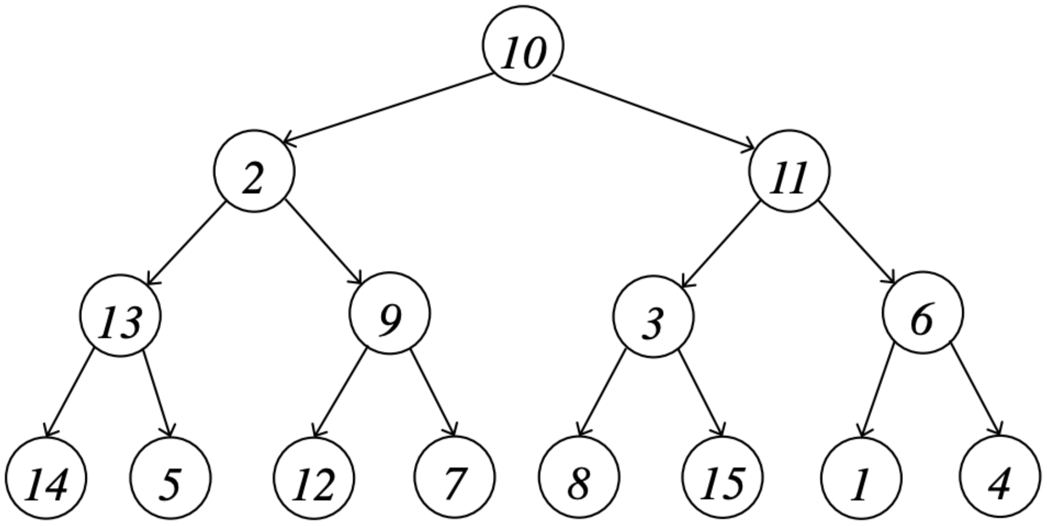
9. A **complete binary tree** is a **binary tree** in which every level of the tree contains all possible nodes.

Implement the following function, which takes in the root node of a `LinkedBinaryTree` and determines whether or not it is a full tree. This function should be iterative.
(20 minutes)

```
def is_complete(root):
    ''' Returns True if the Binary Tree is complete and
    false if not '''
```

Hint: For the non recursive implementation, you should use the *breadth-first search* with an ArrayQueue.

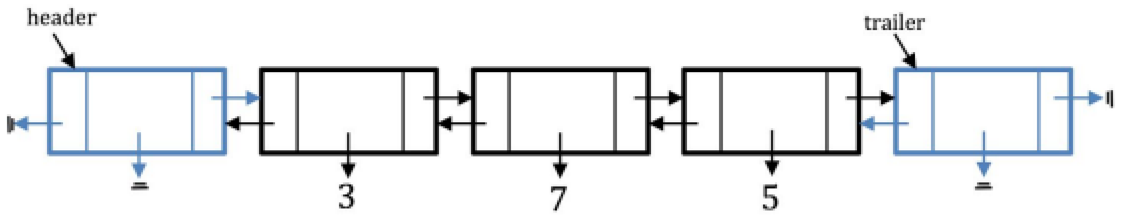
ex)



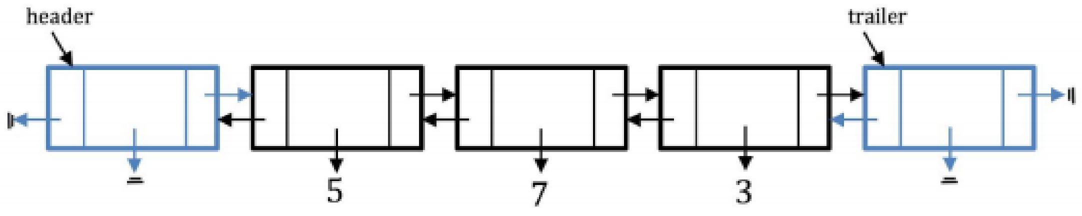
OPTIONAL

10. Implement a method to reverse a doubly linked list. This method should be non-recursive and done **in-place** (do not return a new linked list).

For example if your list looks like:



After calling the method on it, it will look like:



You will implement the reversal in two ways:

a. First implement a function which reverses the data in the list, but does not move any nodes. Instead, change the value at each node so that the order is reversed.

```
def reverse_dll_by_data(dll):
''' Reverses the linked list '''
```

b. Next, implement a function which reverses the order of the nodes in the list. That is, you should move the nodes objects around, without changing their data value and without creating any new node objects.

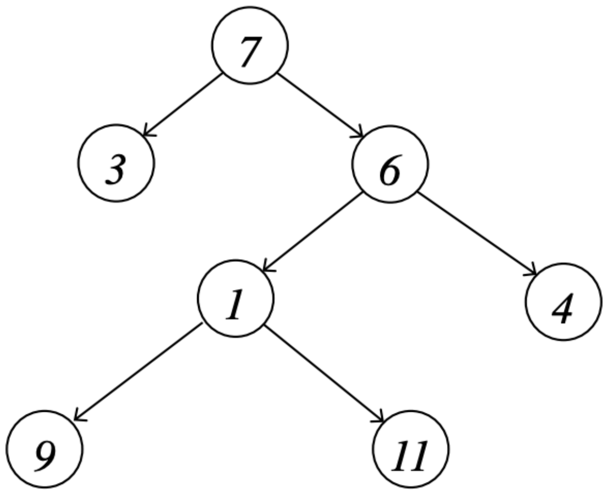
```
def reverse_dll_by_node(dll):
''' Reverses the linked list '''
```

11. A **full binary tree** (or **proper binary tree**) is a **binary tree** in which every node in the tree has 2 or 0 children.

Implement the following function, which takes in the root node of a `LinkedBinaryTree` and determines whether or not it is a full tree. This function should be recursive.
(20 minutes)

```
def is_full(root):
     ''' Returns True if the Binary Tree is full and false
     if not '''
```

ex)



12. Add the following method to the `LinkedBinaryTree` class (35 minutes).

```
def preorder_with_stack(self):
     ''' Returns a generator function that iterates through
     the tree using the preorder traversal '''
```

The method is a generator function that when called, will iterate through the binary tree using preorder traversal without recursion. **You will use one ArrayStack and Θ(1) extra space.**

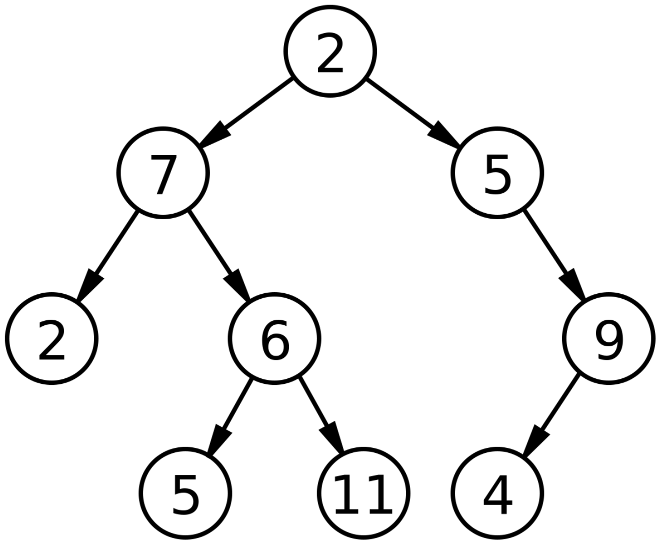
Preorder is as followed: **Data, Left, RIght**. When you're tracing the preorder traversal of the binary tree, imagine how you would place the nodes in the stack, and when you would pop the nodes from the stack. Think of recursion and the call stack!

ex) Given the following code using the tree example below:

```
#t is a LinkedBinaryTree
for item in t.preorder_with_stack():
     print(item, end = ' ')
print()
```

You should expect the following output:

```
2 7 2 6 5 11 5 9 4
```


2
7
5
2
6
9
5
11
4