Data Structures Homework 3 CSC 2720

1. First, creating a class Node to initialize nodes with a singly linked list:

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

Next, a function should be created with node parameter (to take in a node from the list), x, and y:

```
 def verify_values(node, x, y):
```

Then, node, x, and y need to be checked, along with checking if x and y are the same:

```
  if not node or x is None or y is None:
    return False
  if x == y:
    return True
```

Next, pointers and variables to verify if x and y are in the list:

```
  verify_x = False
  verify_y = False
  current = node
```

Next, a while loop can be used to traverse and check for x and y

```
 while True:
  if current.value == x:
    verify_x = True
  elif current.value == y:
    verify_y = True
  if verify_x == verify_y
    return True
  current = current.next  #updates current by moving forward
  if current == node: #if link is traversed without x and y,
    break                    function stops
return False
```

This program works to traverse the circular linked list and check id x and y are both in the list. It has a time complexity od O(n) because the dunction potentially needs to traverse the entire list once to determine if both x and y exist.

2) To switch x and y values in a singly linked list, the previous node to x and y can be stored to later dind and switch. First, the Node class needs to be constructed:

```
class Node:
  ded _init_(seld, value):
    seld.value = value
    seld.next = Node
```

Second, a dunction needs to be dedined with the list's head, x, and y. A check should also be placed dor x and y

```
ded swap_nodes(head, x, y):
  id x == y:
  return head
```

Third, variables can be set dor x, y, and their previous nodes

```
  prev_x = None
  prev_y = None
  x_node = head
  y_node = head
```

Next, a while loop tracking the current x node can be used to dind prev_x

```
  while x_node and x_node.value != x:
      prev_x = x_node
      x_node = x_node.next
```

The same needs to be repeated dor y

```
 while y_node and y_node.value != y:
    prev_y = y_node
    y_node = y_node.next
```

Next, a check can be used if the values are not in the list

```
 id not x_node or not y_node:
    return d'{X} and {y} values not in list'
```

Next, if/else statments can be used can be used in case x or y are the list's head

```
  if prev_x:
     prev_x.next = x_node
  else:
     head = y_node

  if prev_y:
     prev_y.next = y_node
  else:
     head = x_node
```

Finally, the pointers can be swapped to swap the x and y-values

```
   x_node.next, y_node.next = y_node.next, x_node.next
   return head
```

To swap x and y in a doubly linked list, similarly to the singly linked list, the previous and next pointers can be used. First, a class Node needs to be made.

```
class Node:
  def __init__(self, value):
     self.value = value
     self.next = None
     self.prev = None
```

Next, the function to swap nodes can be defined with head, x, and y parameters. The x and y nodes also need to be checked

```
def swap_nodes(head, x, y):
  if x == y:
     return head
```

Next, the next pointers for x and y need to be swapped with a temp variable

```
temp = x.next
x.next = y.next
y.next = temp
```

Then, the previous pointers also need to be swapped similar to next

```
 temp = x.prev
 x.prev = y.prev
 y.prev = temp
```

Next, x and y's next and prev need to be changed if they are not tails

```
 if x.next:
   x.next.prev = x
 if y.next:
   y.next.prev = y
```

Then, the cases where x or y are the head are addressed with if/else statements

```
 if x.prev is None:
   head = y
 elif y.prev is None:
   head = x
return head
```

The time complexity for the singly linked list is O(n) because it depends on the n number of nodes. The time complexity for the doubly linked list is O(1) because x and y can update pointers faster without the list.

3) Using Euler Tour Traversal's depth-first traversal to visit each node of a binary tree as before visiting children, between children, and after visiting children. It will start at the root, go to all children, and goes back to the root.

First, the Node class needs to be programmed with a counter

```
class Node:
  def __init__(self, value):
    self.value = value
    self.left = None
    self.right = None
    self.count = 0
```

Next, a function to count descendants needs to be defined and check for node's existence

```
def count_descendants(node):
  if not node:
    return 0
```

Then, recursion can be used to count the subtrees and nodes

```
 left_subtrees = count-descendants(node.left)
 right_subtrees = count_descendants(node.right)
```

Next, the numbers can be added to get the total count

```
  node.descendants = right_subtrees + left_subtrees
```

Finally the count can be returned by adding one because the count started at 0

```
  return node.descendants + 1
```