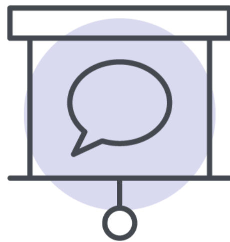
OneClass

# CP164

## FINAL EXAM STUDY GUIDE

*Fall 2018*



# Lecture Notes

OneClass find more resources at oneclass.com

```
"""
-------------------------------------------------------
food.py
Food class definition.
-------------------------------------------------------
Author:  David Brown
ID:      999999999
Email:   dbrown@wlu.ca
__updated__ = "2018-03-12"
-------------------------------------------------------
"""


class Food:
    """
    Defines an object for a single food: name, origin, vegetarian, calories.
    """
    # Constants
    ORIGIN = ("Canadian", "Chinese", "Indian", "Ethiopian",
              "Mexican", "Greek", "Japanese", "Italian", "American",
              "Scottish", "New Zealand", "English")

    @staticmethod
    def origins():
        """
        -------------------------------------------------------
        Creates a string list of food origins in the format:
           0 Canadian
           1 Chinese
           2 Indian
           ...
        Use: s = Food.origins()
        Use: print(Food.origins())
        -------------------------------------------------------
        Postconditions:
            returns
            string - A numbered list of valid food origins.
        -------------------------------------------------------
        """
        string = ""

        for i in range(len(Food.ORIGIN)):
            string += """{:2d} {}
""".format(i, Food.ORIGIN[i])
        return string

    def __init__(self, name, origin, is_vegetarian, calories):
        """
        -------------------------------------------------------
        Initialize a food object.
        Use: f = Food( name, origin, is_vegetarian, calories )
        -------------------------------------------------------
        Preconditions:
            name - food name (str)
            origin - food origin (int)
            is_vegetarian - whether food is vegetarian (boolean)
            calories - caloric content of food (int > 0)
        Postconditions:
            food values are set.
        -------------------------------------------------------
        """
        assert origin in range(len(Food.ORIGIN)), "Invalid origin ID"
        assert is_vegetarian in (True, False, None), "Must be True or False"
        assert calories is None or calories >= 0, "Calories must be >= 0"
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        self.name = name
        self.origin = origin
        self.is_vegetarian = is_vegetarian
        self.calories = calories
        return

    def __str__(self):
        """
        -------------------------------------------------------
        Creates a formatted string of food data.
        Use: print(f)
        Use: s = str(f)
        -------------------------------------------------------
        Postconditions:
            returns:
            string - the formatted contents of food (str)
        -------------------------------------------------------
        """
        if self.calories is None:
            # is a key
            string = "{}, {}".format(self.name, Food.ORIGIN[self.origin])
        else:
            # full data set
            string = """Name:       {}
Origin:     {}
Vegetarian: {}
Calories:   {:,d}""".format(self.name, Food.ORIGIN[self.origin],
self.is_vegetarian, self.calories)
        return string

    def __eq__(self, rs):
        """
        -------------------------------------------------------
        Compares this food against another food for equality.
        Use: f == rs
        -------------------------------------------------------
        Preconditions:
            rs - [right side] food to compare to (Food)
        Postconditions:
            returns:
            result - True if name and origin match, False otherwise (boolean)
        -------------------------------------------------------
        """
        result = (self.name.lower(), self.origin) == (
            rs.name.lower(), rs.origin)
        return result

    def __lt__(self, rs):
        """
        -------------------------------------------------------
        Determines if this food comes before another.
        Use: f < rs
        -------------------------------------------------------
        Preconditions:
            rs - [right side] food to compare to (Food)
        Postconditions:
            returns:
            result - True if food precedes rs, False otherwise (boolean)
        -------------------------------------------------------
        """
        result = (self.name.lower(), self.origin) < \
            (rs.name.lower(), rs.origin)
        return result
```

OneClass find more resources at oneclass.com

```
    def __le__(self, rs):
        """
        -------------------------------------------------------
        Determines if this food precedes or is or equal to another.
        Use: f <= rs
        -------------------------------------------------------
        Preconditions:
            rs - [right side] food to compare to (Food)
        Postconditions:
            returns:
            result - True if this food precedes or is equal to rs,
              False otherwise (boolean)
        -------------------------------------------------------
        """
        result = self < rs or self == rs
        return result

    def write(self, file_variable):
        """
        -------------------------------------------------------
        Writes a single line of food data to an open file.
        Use: f.write( file_variable )
        -------------------------------------------------------
        Preconditions:
            file_variable - an open file of food data (file)
        Postconditions:
            The contents of food are written as a string in the format
              name|origin|is_vegetarian to file_variable.
        -------------------------------------------------------
        """
        print("{}|{}|{}|{}"
              .format(self.name, self.origin, self.is_vegetarian,
self.calories),
              file=file_variable)
        return

    def key(self):
        """
        -------------------------------------------------------
        Creates a formatted string of food key data.
        Use: key = f.key()
        -------------------------------------------------------
        Postconditions:
            returns:
            the formatted contents of food key (str)
        -------------------------------------------------------
        """
        return "{}, {}".format(self.name, self.origin)

    def __hash__(self):
        """
        -------------------------------------------------------
        Generates a hash value from a food name.
        Use: h = hash(f)
        -------------------------------------------------------
        Postconditions:
            returns
            value - the total of the characters in the name string (int > 0)
        -------------------------------------------------------
        """
        value = 0

        for c in self.name:
```

```
        value = value + ord(c)
    return value
```

```
"""
-------------------------------------------------------
food_utilities.py
Utilities for working with a Food object.
-------------------------------------------------------
Author:  David Brown
ID:      999999999
Email:   dbrown@wlu.ca
__updated__ = "2018-01-21"
-------------------------------------------------------
"""
from food import Food


def get_food():
    """
    -------------------------------------------------------
    Creates a food object by requesting data from a user.
    Use: f = get_food()
    -------------------------------------------------------
    Postconditions:
        returns
        food - a completed food object (Food).
    -------------------------------------------------------
    """
    name = input("Name: ")
    print("Origin")
    print(Food.origins())
    origin = int(input(": "))
    s = input("Vegetarian (Y/N): ")
    # Accept a range of values for vegetarian
    is_vegetarian = s.upper() in ('Y', 'TRUE', '1', 'T', 'YES')
    calories = int(input("Calories: "))
    food = Food(name, origin, is_vegetarian, calories)
    return food


def read_food(line):
    """
    -------------------------------------------------------
    Creates and returns a food object from a line of string data.
    Use: f = read_food(line)
    -------------------------------------------------------
    Preconditions:
        line - a vertical bar-delimited line of food data in the format
          name|origin|is_vegetarian|calories (str)
    Postconditions:
        returns
        food - contains the data from line (Food)
    -------------------------------------------------------
    """
    data = line.strip().split("|")
    food = Food(data[0], int(data[1]), data[2] == "True", int(data[3]))
    return food


def read_foods(file_variable):
    """
    -------------------------------------------------------
    Reads a file of food strings into a list of Food objects.
    Use: foods = read_food(file_variable)
    -------------------------------------------------------
    Preconditions:
        file_variable - a file of food data (file)
```

```
    Postconditions:
        returns
        foods - a list of food objects (list of food)
    -------------------------------------------------------
    """
    file_variable.seek(0)
    foods = []

    for line in file_variable:
        food = read_food(line)
        foods.append(food)
    return foods


def write_foods(file_variable, foods):
    """
    -------------------------------------------------------
    Writes a list of food objects to a file.
    Use: write_foods(file_variable, foods)
    -------------------------------------------------------
    Preconditions:
        file_variable - an open file of food data (file)
        foods - a list of Food objects (list of Food)
    Postconditions:
        file_variable contains the objects in foods as strings in the format
          name|origin|is_vegetarian|calories
    -------------------------------------------------------
    """
    for food in foods:
        food.write(file_variable)
    return


def get_vegetarian(foods):
    """
    -------------------------------------------------------
    Creates a list of vegetarian foods.
    Use: v = get_vegetarian(foods)
    -------------------------------------------------------
    Preconditions:
        foods - a list of Food objects (list of Food)
    Postconditions:
        returns
        veggies - Food objects from foods that are vegetarian (list of Food)
    -------------------------------------------------------
    """
    veggies = []

    for food in foods:
        if food.is_vegetarian:
            veggies.append(food)
    return veggies


def by_origin(foods, origin):
    """
    -------------------------------------------------------
    Creates a list of foods by origin.
    Use: v = by_origin(foods, origin)
    -------------------------------------------------------
    Preconditions:
        foods - a list of Food objects (list of Food)
        origin - a food origin (int)
    Postconditions:
```

OneClass find more resources at oneclass.com

```
        returns
        origins - Food objects from foods that are of a particular origin (list
of Food)
    -------------------------------------------------------
    """
    assert origin in range(len(Food.ORIGIN))

    origins = []

    for food in foods:
        if food.origin == origin:
            origins.append(food)
    return origins


def average_calories(foods):
    """
    -------------------------------------------------------
    Determines the average calories in a list of foods.
    Use: avg = average_calories(foods)
    -------------------------------------------------------
    Preconditions:
        foods - a list of Food objects (list of Food)
    Postconditions:
        returns
        avg - average calories in all Food objects of foods (int)
    -------------------------------------------------------
    """
    total = 0
    count = len(foods)

    for food in foods:
        total += food.calories

    if count > 0:
        avg = total // count
    else:
        avg = 0
    return avg


def calories_by_origin(foods, origin):
    """
    -------------------------------------------------------
    Determines the average calories in a list of foods.
    Use: a = calories_by_origin(foods, origin)
    -------------------------------------------------------
    Preconditions:
        foods - a list of Food objects (list of Food)
        origin - the origin of the Food objects to find (int)
    Postconditions:
        returns
        avg - average calories for all Foods of the requested origin (int)
    -------------------------------------------------------
    """
    assert origin in range(len(Food.ORIGIN))

    total = 0
    count = 0

    for food in foods:
        if food.origin == origin:
            total += food.calories
            count += 1
```

OneClass find more resources at oneclass.com

```
    if count > 0:
        avg = total // count
    else:
        avg = 0
    return avg


def food_table(foods):
    """
    -------------------------------------------------------
    Prints a formatted table of foods, sorted by name.
    Use: food_table(foods)
    -------------------------------------------------------
    Preconditions:
        foods - a list of Food objects (list of Food)
    Postconditions:
        prints
        a table of the foods sorted by name
    -------------------------------------------------------
    """
    foods.sort()
    print("{:35s} {:12s} {:10s} {:8s}".format(
        "Food", "Origin", "Vegetarian", "Calories"))
    print("{} {} {} {}".format("-" * 35, "-" * 12, "-" * 10, "-" * 8))

    for food in foods:
        print("{:35s} {:12s} {:>10} {:8,d}".format(
            food.name, Food.ORIGIN[food.origin], str(food.is_vegetarian),
food.calories))
    return


def food_search(foods, origin, max_cals, is_veg):
    """
    -------------------------------------------------------
    Searches for foods that fit certain conditions.
    Use: results = food_search(foods, origin, max_cals, is_veg)
    -------------------------------------------------------
    Preconditions:
        foods - a list of Food objects (list of Food)
        origin - the origin of the food; if -1, accept any origin (int)
        max_cals - the maximum calories for the food; if 0, accept any calories
value (int)
        is_veg - whether the food is vegetarian or not; if False accept any food
(boolean)
    Postconditions:
        returns
        result - a list of foods that fit the conditions (list of Food)
            foods parameter must be unchanged
    -------------------------------------------------------
    """
    assert origin in range(-1, len(Food.ORIGIN))

    result = []

    for food in foods:
        if (origin == -1 or food.origin == origin) and (max_cals == 0 or
food.calories <= max_cals) and (not is_veg or food.is_vegetarian):
            result.append(food)
    return result


def food_test():
```

OneClass find more resources at oneclass.com

```
fv = open('foods.txt', 'r', encoding='utf-8')
foods = read_foods(fv)
fv.close()

print("Length: {}".format(len(foods)))
r = food_search(foods, -1, 300, True)
print("Length: {}".format(len(r)))

for f in r:
    print(f)
return
```

OneClass find more resources at oneclass.com

```
"""
-------------------------------------------------------
sorts_linked.py
Linked versions of various sorts. Implemented on linked Deques.
-------------------------------------------------------
Author:  David Brown
ID:      999999999
Email:   dbrown@wlu.ca
__updated__ = "2017-08-25"
-------------------------------------------------------
"""
from math import log
from list_linked import List


class Sorts:
    """
    -------------------------------------------------------
    Defines a number of linked sort operations.
    Uses class attribute 'swaps' to determine how many times
    elements are swapped by the class.
    Use: print(Sorts.swaps)
    Use: Sorts.swaps = 0
    -------------------------------------------------------
    """
    swaps = 0  # Tracks swaps performed.

    # The Sorts

    @staticmethod
    def selection_sort(a):
        """
        -------------------------------------------------------
        Sorts a linked list using the Selection Sort algorithm.
        Finds maximum value each pass.
        Use: selection_sort(a)
        -------------------------------------------------------
        Preconditions:
            a - a linked list of comparable elements (List)
        Postconditions:
            Contents of a are sorted.
        -------------------------------------------------------
        """
        # Split the list into the sorted (_front) and unsorted parts.
        unsorted = a._front
        a._front = None
        # Go through each node in the unsorted list and find the max value
        # to insert at the front of the sorted list.
        while unsorted is not None:
            max_prev = None
            max_node = unsorted
            previous = unsorted
            current = max_node._next

            while current is not None:
                if current._data > max_node._data:
                    max_prev = previous
                    max_node = current
                previous = current
                current = current._next
            # Remove the max node from the list.
            Sorts.swaps += 1

            if max_prev is None:
```

OneClass find more resources at oneclass.com

```
                unsorted = max_node._next
            else:
                max_prev._next = max_node._next
            # Move the next max node to the front of the sorted list.
            max_node._next = a._front
            a._front = max_node
        return

    @staticmethod
    def bubble_sort(a):
        """
        -------------------------------------------------------
        Sorts a linked list using the Bubble Sort algorithm.
        Use: bubble_sort(a)
        -------------------------------------------------------
        Preconditions:
            a - a linked list of comparable elements (?)
        Postconditions:
            Contents of a are sorted.
        -------------------------------------------------------
        """
        done = False
        last = None

        while not done:
            # if no elements have been swapped, then the list is sorted
            done = True
            # Get the front of the list.
            previous = None
            current = a._front
            swapped = a._front

            while current is not last and current._next is not None:

                if current._data > current._next._data:
                    # If you swapped you need another pass.
                    done = False
                    # The pair current, current._next is out of order.
                    Sorts.swaps += 1
                    a._swap(previous, current)
                    # Keep track of last node swapped
                    swapped = current
                    # current is unchanged - update previous
                    if previous is None:
                        previous = a._front
                    else:
                        previous = previous._next
                else:
                    # Move to next node.
                    previous = current
                    current = current._next
            last = swapped
        # done == True iff no pair of keys was swapped on the last pass.
        return

    @staticmethod
    def comb_sort(a):
        """
        -------------------------------------------------------
        Sorts an List using the Comb Sort algorithm.
        Use: comb_sort(a)
        -------------------------------------------------------
        Preconditions:
          a - a linked list of comparable elements (?)
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        Postconditions:
          Contents of a are sorted.
        -------------------------------------------------------
        """
        n = len(a)

        if n > 0:
            gap = n
            done = False

            while gap > 1 or not done:
                done = True
                previous = None
                current = a._front
                gap = int(gap / 1.3)

                if gap < 1:
                    gap = 1

                i = 0
                prev_far = current
                far = current._next
                # Move to the far node for comparison.
                while i < gap - 1 and far is not None:
                    prev_far = far
                    far = far._next
                    i += 1

                while current is not None and far is not None:
                    if current._data > far._data:
                        Sorts.swaps += 1
                        a._swap(previous, prev_far)
                        done = False
                    # Increment all nodes.
                    prev_far = far
                    far = far._next
                    previous = current
                    current = current._next
        return

    @staticmethod
    def insertion_sort(a):
        """
        -------------------------------------------------------
        Sorts a linked list using the Insertion Sort algorithm.
        Use: insertion_sort(a)
        -------------------------------------------------------
        Preconditions:
            a - a linked list of comparable elements (?)
        Postconditions:
            Contents of a are sorted.
        -------------------------------------------------------
        """
        # Split the list into the sorted (_front) and unsorted parts.
        unsorted = a._front
        a._front = None

        # Go through each node in the unsorted list and insert it into the
        # proper position in the sorted list.
        while unsorted is not None:
            # Isolate the first node of the unsorted list.
            node = unsorted
            unsorted = unsorted._next
            # Find the proper place for the new node in the sorted so far list.
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        # (Very similar to priorityQueue insertion.)
        previous = None
        current = a._front

        while current is not None and current._data < node._data:
            previous = current
            current = current._next

        # Insert node into the proper place in the sorted list.
        Sorts.swaps += 1

        if previous is None:
            node._next = a._front
            a._front = node
        else:
            node._next = current
            previous._next = node
    return

@staticmethod
def merge_sort(a):
    """
    -------------------------------------------------------
    Sorts a linked list using the Merge Sort algorithm.
    Use: merge_sort(a)
    -------------------------------------------------------
    Preconditions:
        a - a linked list of comparable elements (List)
    Postconditions:
        Contents of a are sorted.
    -------------------------------------------------------
    """
    a._front = Sorts._merge_sort_aux(a._front)
    return

@staticmethod
def _merge_sort_aux(current):
    """
    -------------------------------------------------------
    Divides a linked list into halves, sorts each half, then
    merges the halves back together.
    Use: current = _merge_sort_aux(current)
    -------------------------------------------------------
    Preconditions:
      current - pointer to a List node (_ListNode)
    Postconditions:
      returns:
      current - pointer to sorted List segment (_ListNode)
    -------------------------------------------------------
    """
    # Split the list only if there are at least two elements.
    if current is not None and current._next is not None:
        # Generate the left and right lists.
        left, right = Sorts._merge_split(current)
        # Put the left list in order.
        left = Sorts._merge_sort_aux(left)
        # Put the right list in order.
        right = Sorts._merge_sort_aux(right)
        # Merge the left and right lists.
        current = Sorts._merge(left, right)
    return current

@staticmethod
def _merge_split(current):
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        """
        -------------------------------------------------------
        Split the list by putting alternating nodes into left and right lists.
        Use: left, right = _merge_split(current)
        -------------------------------------------------------
        Preconditions:
          current - pointer to a List node (_ListNode)
        Postconditions:
          returns:
          left - pointer to left List segment (_ListNode)
          right - pointer to right list segment (_ListNode)
        -------------------------------------------------------
        """
        #
        left = None
        right = None
        toggle = 'l'

        while current is not None:
            next_node = current._next

            if toggle == 'l':
                current._next = left
                left = current
                toggle = 'r'
            else:
                current._next = right
                right = current
                toggle = 'l'

            current = next_node
        return left, right

    @staticmethod
    def _merge(left, right):
        """
        -------------------------------------------------------
        Merges two parts of a linked list together.
        Use: new = _merge(left, right)
        -------------------------------------------------------
        Preconditions:
          left - pointer to a List node (_ListNode)
          right - pointer to a List node (_ListNode)
        Postconditions:
          returns:
          new - pointer to sorted merge of left and right (_ListNode)
        -------------------------------------------------------
        """
        # Initialize the new list.
        if left._data <= right._data:
            new = left
            left = left._next
        else:
            new = right
            right = right._next

        # Create a pointer to traverse the new list.
        current = new

        # Traverse both lists appending larger value to the end of the list.
        while left is not None and right is not None:

            if left._data <= right._data:
                current._next = left
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
                current = current._next
                left = left._next
            else:
                current._next = right
                current = current._next
                right = right._next

        # Append the remaining list.
        if left is not None:
            current._next = left
        elif right is not None:
            current._next = right
        return new

    @staticmethod
    def quick_sort(a):
        """
        -------------------------------------------------------
        Sorts a linked list using the Quick Sort algorithm.
        Use: quick_sort(a)
        -------------------------------------------------------
        Preconditions:
          a - a linked list of comparable elements (?)
        Postconditions:
          Contents of a are sorted.
        -------------------------------------------------------
        """
        a._front = Sorts._quick_sort_aux(a._front)
        return

    @staticmethod
    def _quick_sort_aux(current):
        """
        -------------------------------------------------------
        Divides a linked list into halves, sorts each half, then
        concatenates the halves back together.
        Use: current = _quick_sort_aux(current)
        -------------------------------------------------------
        Preconditions:
          current - pointer to a List node (_ListNode)
        Postconditions:
          returns:
          current - pointer to a sorted List (_ListNode)
        -------------------------------------------------------
        """
        # to sort the sublist a[p:q] of linked list a into ascending order
        if current is not None:
            # partitions a[first:last] into a[first:pivot] and a[pivot+1:last]
            lesser, equals, greater = Sorts._partition(current)
            lesser = Sorts._quick_sort_aux(lesser)
            greater = Sorts._quick_sort_aux(greater)
            current = Sorts._append(lesser, equals)
            current = Sorts._append(current, greater)
        return current

    @staticmethod
    def _partition(current):
        """
        -------------------------------------------------------
        Divides a linked list into three parts.
        Use: l, e, g = _partition(current)
        -------------------------------------------------------
        Preconditions:
          current - pointer to a List node containing pivot value (_ListNode)
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
    Postconditions:
      returns:
      lesser - a list of values less than the pivot value (_ListNode)
      greater - a list of values greater than the pivot value (_ListNode)
      equals - a list of values equal to the pivot value (_ListNode)
    -------------------------------------------------------
    """
    pivotValue = current._data
    lesser = None
    greater = None
    equals = None

    while current is not None:
        next_node = current._next

        if pivotValue > current._data:
            current._next = lesser
            lesser = current
        elif pivotValue < current._data:
            current._next = greater
            greater = current
        else:
            current._next = equals
            equals = current

        current = next_node
    return lesser, equals, greater

@staticmethod
def _append(head, tail):
    """
    -------------------------------------------------------
    Combines two lists together in order.
    Use: current = _append(head, tail)
    -------------------------------------------------------
    Preconditions:
      head - pointer to a List node of the first list (_ListNode)
      tail - pointer to a List node of the second list (_ListNode)
    Postconditions:
      returns:
      head - pointer to the combined nodes in order (_ListNode)
    -------------------------------------------------------
    """
    current = head
    previous = None

    while current is not None:
        previous = current
        current = current._next

    if previous is None:
        head = tail
    else:
        previous._next = tail
    return head

@staticmethod
def radix_sort(a):
    """
    -------------------------------------------------------
    Performs a base 10 radix sort.
    Use: radix_sort(a)
    -------------------------------------------------------
    Preconditions:
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
            a - a List of base 10 integers (List)
        Postconditions:
            Contents of a are sorted.
        -------------------------------------------------------
        """
        if len(a) > 0:
            # Determine the maximum digit in the numbers in a.
            max_val = a.max()
            passes = int(log(max_val, 10) + 1)
            # Create the empty buckets.
            buckets = []

            for _ in range(10):
                buckets.append(List())

            for digit in range(passes):
                # Calculate the digit extraction numerator and denominator.
                d = 10 ** digit
                n = d * 10

                while not a.empty():
                    # Move the list nodes to the appropriate bucket.
                    # Extract the individual digit.
                    index = a._front._data % n // d
                    # Move the front node to the proper bucket.
                    buckets[index]._move_front(a)

                for index in range(len(buckets) - 1, -1, -1):
                    # Move the nodes back to the original list.
                    # (Start with right-most bucket).
                    while not buckets[index].empty():
                        a._move_front(buckets[index])
        return

    # Sort Utilities

    @staticmethod
    def to_array(a):
        """
        -------------------------------------------------------
        Copies list values to a Python list.
        Use: values = to_array(a)
        -------------------------------------------------------
        Preconditions:
            a - a linked list of comparable elements (?)
        Postconditions:
            returns
            values - the contents of a in a Python list.= (list of ?)
        -------------------------------------------------------
        """
        values = []
        current = a._front

        while current is not None:
            values.append(current._data)
            current = current._next
        return values

    @staticmethod
    def sort_test(a):
        """
        -------------------------------------------------------
        Determines whether a linked list is is_sorted or not.
        Use: sort_test(a)
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
    -------------------------------------------------------
    Preconditions:
      a - a linked list of comparable elements (?)
    Postconditions:
      returns:
      is_sorted - True if contents of a are sorted, False otherwise.
    -------------------------------------------------------
    """
    is_sorted = True
    current = a._front

    while is_sorted and current is not None and \
            current._next is not None:

        if current._data <= current._next._data:
            current = current._next
        else:
            is_sorted = False
    return is_sorted
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
"""
-------------------------------------------------------
sorts_array.py
Array versions of various sorts.
-------------------------------------------------------
Author:  David Brown
ID:      999999999
Email:   dbrown@wlu.ca
__updated__ = "2017-11-21"
-------------------------------------------------------
"""
# Imports
from math import log
from bst_linked import BST


class Sorts:
    """
    -------------------------------------------------------
    Defines a number of array-based sort operations.
    Uses class attribute 'swaps' to determine how many times
    elements are swapped by the class.
    Use: print(Sorts.swaps)
    Use: Sorts.swaps = 0
    -------------------------------------------------------
    """
    swaps = 0  # Tracks swaps performed.

    # The Sorts

    @staticmethod
    def selection_sort(a):
        """
        -------------------------------------------------------
        Sorts an array using the Selection Sort algorithm.
        Use: Sorts.selection_sort(a)
        -------------------------------------------------------
        Preconditions:
            a - a Python list of comparable elements (?)
        Postconditions:
            Contents of a are sorted.
        -------------------------------------------------------
        """
        n = len(a)

        for i in range(n):
            # Walk through entire array
            m = i
            j = i + 1

            for j in range(i + 1, n):
                # Find smallest value in unsorted part of array

                if a[m] > a[j]:
                    # Track smallest value so far
                    m = j

            if m != i:
                # swap elements only if necessary
                Sorts._swap(a, m, i)
        return

    @staticmethod
    def insertion_sort(a):
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        """
        -------------------------------------------------------
        Sorts an array using the Insertion Sort algorithm.
        Use: Sorts.insertion_sort(a)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (?)
        Postconditions:
            Contents of a are sorted.
        -------------------------------------------------------
        """
        n = len(a)

        for i in range(1, n):
            # Walk through entire array
            # Move each key bigger than save in a[1:n-1] one space to right.
            save = a[i]
            # Count as a swap
            Sorts.swaps += 1
            j = i - 1

            while j >= 0 and a[j] > save:
                Sorts._shift(a, j + 1, j)
                j -= 1
            # Move save into hole opened up by moving previous keys to right.
            a[j + 1] = save
        return

    @staticmethod
    def bubble_sort(a):
        """
        -------------------------------------------------------
        Sorts an array using the Bubble Sort algorithm.
        Use: Sorts.bubble_sort(a)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (?)
        Postconditions:
            Contents of a are sorted.
        -------------------------------------------------------
        """
        done = False
        last = len(a) - 1

        while not done:
            # assume done is true
            done = True
            last_swapped = 0
            i = 0

            while i < last:
                if a[i] > a[i + 1]:
                    # Save the list index swapped.
                    last_swapped = i
                    # The pair (a[i], a[i+1]) is out of order.
                    # Exchange a[i] and a[i + 1] to put them in sorted order.
                    Sorts._swap(a, i, i + 1)
                    # If you swapped you need another pass.
                    done = False
                i += 1
            last = last_swapped
            # Decreases 'last' because everything after last_swapped is already
            # in order. done == False iff no pair of keys swapped on last pass.
        return
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
    @staticmethod
    def bst_sort(a):
        """
        -------------------------------------------------------
        Sorts an array using the Tree Sort algorithm.
        Use: Sorts.bst_sort(a)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (?)
        Postconditions:
            Contents of a are sorted.
        -------------------------------------------------------
        """
        bst = BST()

        for v in a:
            bst.insert(v)

        a[:] = bst.inorder()
        return

    @staticmethod
    def shell_sort(a):
        """
        -------------------------------------------------------
        Sorts an array using the Shell Sort algorithm.
        Use: Sorts.shell_sort(a)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (?)
        Postconditions:
            Contents of a are sorted.
        -------------------------------------------------------
        """
        n = len(a)
        increment = n // 3

        while increment > 0:
            i = increment

            while i < n:
                j = i
                temp = a[i]

                while j >= increment and a[j - increment] > temp:
                    Sorts._shift(a, j, j - increment)
                    j = j - increment
                a[j] = temp
                i += 1

            increment = increment // 3
        return

    @staticmethod
    def merge_sort(a):
        """
        -------------------------------------------------------
        Sorts an array using the Merge Sort algorithm.
        Use: Sorts.merge_sort(a)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (?)
        Postconditions:
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
            Contents of a are sorted.
        -------------------------------------------------------
        """
        Sorts._merge_sort_aux(a, 0, len(a) - 1)
        return

    @staticmethod
    def _merge_sort_aux(a, first, last):
        """
        -------------------------------------------------------
        Divides an array into halves, sorts each half, then
        merges the halves back together.
        Use: Sorts._merge_sort_aux(a, first, last)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (?)
            first - beginning of subarray of a (int)
            last - end of subarray of a (int)
        Postconditions:
            Contents of a from first to last are sorted.
        -------------------------------------------------------
        """

        if first < last:
            middle = (last - first) // 2 + first
            Sorts._merge_sort_aux(a, first, middle)
            Sorts._merge_sort_aux(a, middle + 1, last)
            Sorts._merge(a, first, middle, last)
        return

    @staticmethod
    def _merge(a, first, middle, last):
        """
        -------------------------------------------------------
        Merges two parts of an array together.
        Use: Sorts._merge(a, first, middle, last)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (?)
            first - beginning of subarray of a (int)
            middle - middle of subarray of a (int)
            last - end of subarray of a (int)
        Postconditions:
            Contents of a are sorted.
        -------------------------------------------------------
        """
        temp = []
        i = first
        j = middle + 1

        while i <= middle and j <= last:

            if a[i] <= a[j]:
                # put leftmost element of left array into temp array
                temp.append(a[i])
                i += 1
            else:
                # put leftmost element of right array into temp array
                temp.append(a[j])
                j += 1

        # copy any remaining elements from the left half
        while i <= middle:
            temp.append(a[i])
```

OneClass find more resources at oneclass.com

```
            i += 1

        # copy any remaining elements from the right half
        while j <= last:
            temp.append(a[j])
            j += 1

        # copy the temporary array back to the passed array
        i = 0

        while i < len(temp):
            a[first + i] = temp[i]
            i += 1
        return

    @staticmethod
    def quick_sort(a):
        """
        -------------------------------------------------------
        Sorts an array using the Quick Sort algorithm.
        Use: Sorts.quick_sort(a)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (?)
        Postconditions:
            Contents of a are sorted.
        -------------------------------------------------------
        """
        Sorts._quick_sort_aux(a, 0, len(a) - 1)
        return

    @staticmethod
    def _quick_sort_aux(a, first, last):
        """
        -------------------------------------------------------
        Divides an array into halves, sorts each half, then
        concatenates the halves back together.
        Use: Sorts._quick_sort_aux(a, first, last)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (?)
            first - beginning of subarray of a (int)
            last - end of subarray of a (int)
        Postconditions:
            Contents of a from first to last are sorted.
        -------------------------------------------------------
        """
        # to sort the subarray a[p:q] of array a into ascending order
        if first < last:
            # partitions a[first:last] into a[first:pivot] and a[pivot+1:last]
            i = Sorts._partition(a, first, last)
            Sorts._quick_sort_aux(a, first, i - 1)
            Sorts._quick_sort_aux(a, i + 1, last)
        return

    @staticmethod
    def _partition(a, first, last):
        """
        -------------------------------------------------------
        Divides an array into two parts.
        Use: Sorts._partition(a, first, last)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (?)
```

OneClass find more resources at oneclass.com

```
            first - beginning of subarray of a (int)
            last - last of subarray of a (int)
        Postconditions:
            Contents of a from first to last are sorted.
        -------------------------------------------------------
        """
        pivot_index = first
        low = first
        high = last - 1  # After we remove pivot it will be one smaller
        # print("{} - {} - {}".format(a[first:pivot_index], a[pivot_index],
        #                         a[pivot_index + 1:last]))
        pivot_value = a[pivot_index]
        a[pivot_index] = a[last]

        while low <= high:
            # print("{} - {} - {}".format(a[first:pivot_index],
            #        a[pivot_index], a[pivot_index + 1:last]))
            while low <= high and a[low] < pivot_value:
                low = low + 1
            while low <= high and a[high] >= pivot_value:
                high = high - 1
            if low <= high:
                Sorts._swap(a, low, high)
        Sorts._shift(a, last, low)
        # a[last] = a[low]
        a[low] = pivot_value
        # print("{} - {} - {}".format(a[first:pivot_index], a[pivot_index],
        #                    a[pivot_index + 1:last]))
        # Return the new pivot position.
        return low

    @staticmethod
    def heap_sort(a):
        """
        -------------------------------------------------------
        Sorts an array using the Heap Sort algorithm.
        Use: Sorts.heap_sort(a)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (?)
        Postconditions:
            Contents of a are sorted.
        -------------------------------------------------------
        """
        first = 0
        last = len(a) - 1
        Sorts._build_heap(a, first, last)
        i = last

        while i > first:
            Sorts._swap(a, i, first)
            Sorts._reheap(a, first, i - 1)
            i -= 1
        return

    @staticmethod
    def _build_heap(a, first, last):
        """
        -------------------------------------------------------
        Creates a heap.
        Use: Sorts._build_heap(a, first, last)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (list)
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
            first - first element in array to process (int)
            last - last element in array to process (int)
        Postconditions:
            Contents of a from first to last are sorted.
        -------------------------------------------------------
        """
        i = last // 2

        while i >= first:
            Sorts._reheap(a, i, last)
            i -= 1
        return

    @staticmethod
    def _reheap(a, first, last):
        """
        -------------------------------------------------------
        Establishes heap property in a.
        Use: Sorts._reheap(a, first, last)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (?)
            first - first element in array to process (int)
            last - last element in array to process (int)
        Postconditions:
            Contents of a from first to last are heaped.
        -------------------------------------------------------
        """
        done = False

        while 2 * first + 1 <= last and not done:
            k = 2 * first + 1

            if k < last and a[k] < a[k + 1]:
                k += 1

            if a[first] >= a[k]:
                done = True
            else:
                Sorts._swap(a, first, k)
                first = k
        return

    @staticmethod
    def comb_sort(a):
        """
        -------------------------------------------------------
        Sorts an array using the Comb Sort algorithm.
        Use: Sorts.comb_sort(a)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (?)
        Postconditions:
            Contents of a are sorted.
        -------------------------------------------------------
        """
        n = len(a)
        gap = n
        done = False

        while gap > 1 or not done:
            done = True
            gap = int(gap / 1.3)
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
            if gap < 1:
                gap = 1

            i = 0
            j = gap

            while j < n:
                if a[i] > a[j]:
                    Sorts._swap(a, i, j)
                    done = False
                i += 1
                j += 1
        return

    @staticmethod
    def cocktail_sort(a):
        """
        -------------------------------------------------------
        Sorts an array using the Cocktail Sort algorithm.
        Use: Sorts.cocktail_sort(a)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (?)
        Postconditions:
            Contents of a are sorted.
        -------------------------------------------------------
        """
        n = len(a)
        first = 0
        last = n - 1

        while first < last:
            # Initialize last_swapped to the beginning of the array.
            # Stops bottom loop from executing if no swaps done by top loop.
            last_swapped = 0
            i = first

            while i < last:
                if a[i] > a[i + 1]:
                    # test whether the two elements are in the correct order
                    Sorts._swap(a, i, i + 1)
                    last_swapped = i
                i += 1
            last = last_swapped

            # Initialize first_swapped to the end of the array.
            # Stops top loop from executing if no swaps done by bottom loop.
            first_swapped = n - 1
            i = last

            while i > first:
                if a[i] < a[i - 1]:
                    # test whether the two elements are in the correct order
                    Sorts._swap(a, i, i - 1)
                    first_swapped = i
                i -= 1
            first = first_swapped
        return

    @staticmethod
    def gnome_sort(a):
        """
        -------------------------------------------------------
        Sorts an array using the Gnome Sort algorithm.
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        Use: gnome_sort(a)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (list)
        Postconditions:
            Contents of a are sorted.
        -------------------------------------------------------
        """
        pos = 0
        n = len(a)

        while pos < n - 1:

            if a[pos] <= a[pos + 1]:
                # Compared elements are in order
                pos += 1
            else:
                # Compared elements must be swapped
                Sorts._swap(a, pos, pos + 1)

                if pos > 0:
                    # Go back to compare newly-swapped element
                    pos -= 1
                else:
                    # index 0 contains smallest element so far
                    pos += 1
        return

    @staticmethod
    def gnome_sort2(a):
        """
        -------------------------------------------------------
        Sorts an array using the Gnome Sort algorithm.
        Use: gnome_sort(a)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (?)
        Postconditions:
            Contents of a are sorted.
        -------------------------------------------------------
        """
        n = len(a)
        i = 0

        while i < n - 1:
            pos = i

            while pos >= 0 and a[pos] > a[pos + 1]:
                # Walk backwards through elements, swapping if necessary
                Sorts._swap(a, pos, pos + 1)
                pos -= 1
            i += 1
        return

    @staticmethod
    def gnome_r(a):
        """
        -------------------------------------------------------
        Sorts an array using the Gnome Sort algorithm.
        Use: gnome_sort(a)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (?)
        Postconditions:
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
            Contents of a are sorted.
        -------------------------------------------------------
        """
        Sorts.gnome_r_aux(a, 0)
        return

    @staticmethod
    def gnome_r_aux(a, pos):
        """
        -------------------------------------------------------
        Sorts an array using the Gnome Sort algorithm.
        Use: gnome_sort(a)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (?)
            pos - index of first value to compare (int)
        Postconditions:
            Contents of a up to index p are sorted.
        -------------------------------------------------------
        """
        if pos < len(a) - 1:

            if a[pos] <= a[pos + 1]:
                Sorts.gnome_r_aux(a, pos + 1)
            else:
                Sorts._swap(a, pos, pos + 1)

                if pos > 0:
                    Sorts.gnome_r_aux(a, pos - 1)
                else:
                    Sorts.gnome_r_aux(a, pos + 1)
        return

    @staticmethod
    def binary_insert_sort(a):
        """
        -------------------------------------------------------
        Sorts an array using the Binary Insertion Sort algorithm.
        Use: Sorts.binary_insert_sort(a)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (list)
        Postconditions:
            Contents of a are sorted.
        -------------------------------------------------------
        """
        n = len(a)
        i = 1

        while i < n:
            ins = Sorts._bin_srch_i(a, 0, i, a[i])
            # Move key bigger than save in a[1:n-1] one space to the right.
            if ins < i:
                save = a[i]
                j = i - 1

                while j > ins - 1:
                    Sorts._shift(a, j + 1, j)
                    # a[j + 1] = a[j]
                    j -= 1
                a[ins] = save
            i += 1
        return
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
    @staticmethod
    def _bin_srch_r(a, low, high, value):
        """
        -------------------------------------------------------
        Sorts an array using the Binary Insertion Sort algorithm.
        Use: Sorts._bin_srch_r(a)
        Recursive algorithm.
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (list)
            low - starting point of a to search for value (int)
            high - end point of a to search for value (int)
            value - value to search for position in a (?)
        Postconditions:
            returns
            mid - the insert point for value in a between low and high
        -------------------------------------------------------
        """
        if low == high:
            mid = low
        else:
            mid = low + ((high - low) // 2)

            if value > a[mid]:
                mid = Sorts._bin_srch_r(a, mid + 1, high, value)
            elif value < a[mid]:
                mid = Sorts._bin_srch_r(a, low, mid, value)
        return mid

    @staticmethod
    def _bin_srch_i(a, low, high, value):
        """
        -------------------------------------------------------
        Sorts an array using the Binary Insertion Sort algorithm.
        Use: Sorts._bin_srch_r(a)
        Iterative algorithm.
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (list)
            low - starting point of a to search for value (int)
            high - end point of a to search for value (int)
            value - value to search for position in a (?)
        Postconditions:
            returns
            mid - the insert point for value in a between low and high
        -------------------------------------------------------
        """
        found = False
        mid = (low + high) // 2

        while low < high and not found:
            # Find the middle of the current subarray.
            if value > a[mid]:
                # Search the right subarray.
                low = mid + 1
                mid = (low + high) // 2
            elif value < a[mid]:
                # Search the left subarray.
                high = mid
                mid = (low + high) // 2
            else:
                found = True
        return mid
```

OneClass find more resources at oneclass.com

```
    @staticmethod
    def radix_sort(a):
        """
        -------------------------------------------------------
        Performs a base 10 radix sort.
        Use: Sorts.radix_sort(a)
        -------------------------------------------------------
        Preconditions:
            a - an array of base 10 integers (list)
        Postconditions:
            Contents of a are sorted.
        -------------------------------------------------------
        """
        if len(a) > 0:
            # Find the largest number in a.
            max_val = max(a)
            # Find the number of digits in the largest number.
            passes = int(log(max_val, 10) + 1)
            # Create the empty buckets.
            buckets = []

            for _ in range(10):
                buckets.append([])

            for digit in range(passes):
                # Calculate the digit extraction numerator and denominator.
                d = 10 ** digit
                n = d * 10

                for v in a:
                    # Assign the array values to the appropriate bucket.
                    # Extract the individual digit.
                    index = v % n // d
                    # Add the number to the proper bucket.
                    buckets[index].append(v)

                # Put the values back into the original array.
                index = 0

                for bucket in buckets:
                    while bucket != []:
                        a[index] = bucket.pop(0)
                        index += 1
        return

    @staticmethod
    def counting_sort(a):

        if len(a) > 0:
            m = max(a) + 1
            b = [0] * m

            for i in range(m):
                n = a.count(i)
                b[i] = n

            a[:] = []
            i = 0

            for v in b:
                for _ in range(v):
                    a.append(i)
                i += 1
        return
```

```
    @staticmethod
    def counting_sort_2(a):
        m = max(a) + 1
        b = [0] * m

        while a != []:
            i = a.pop()
            b[i] += 1

        for i in range(len(b)):
            for _ in range(b[i]):
                a.append(i)
        return

    @staticmethod
    def counting_sort_3(a):
        m = max(a) + 1
        b = [0] * m

        for v in a:
            b[v] += 1

        i = 0

        for j in range(len(b)):
            for _ in range(b[j]):
                a[i] = j
                i += 1
        return

    # Sort Utilities

    @staticmethod
    def sort_test(a):
        """
        -------------------------------------------------------
        Determines whether an array is is_sorted or not.
        Use: sort_test(a)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (?)
        Postconditions:
            returns
            is_sorted - True if contents of a are sorted,
                False otherwise.
        -------------------------------------------------------
        """
        is_sorted = True
        n = len(a)
        i = 0

        while is_sorted and i < n - 1:

            if a[i] <= a[i + 1]:
                i += 1
            else:
                is_sorted = False
        return is_sorted

    @staticmethod
    def _swap(a, i, j):
        """
        -------------------------------------------------------
```

OneClass find more resources at oneclass.com

```
        Swaps the data contents of two array elements.
        Updates 'swaps'.
        Use: Sorts._swap(a, i, j)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (?)
            i - index of one value (int 0 <= i < len(a))
            j - index of another value (int 0 <= j < len(a))
        Postconditions:
            Values in a[i] and a[j] are swapped.
        -------------------------------------------------------
        """
        Sorts.swaps += 1

        temp = a[i]
        a[i] = a[j]
        a[j] = temp
        return

    @staticmethod
    def _shift(a, i, j):
        """
        -------------------------------------------------------
        Shifts the contents of a[j] to a[i].
        Updates 'swaps' - worth 1/3 of _swap
        Use: Sorts._shift(a, i, j)
        -------------------------------------------------------
        Preconditions:
            a - an array of comparable elements (?)
            i - index of one value (int 0 <= i < len(a))
            j - index of another value (int 0 <= j < len(a))
        Postconditions:
            Value in a[j] is copied to a[i].
        -------------------------------------------------------
        """
        Sorts.swaps += 0.34

        a[i] = a[j]
        return
```

OneClass find more resources at oneclass.com

```
"""
-------------------------------------------------------
list_array.py
Array version of the list ADT.
-------------------------------------------------------
Author:  David Brown
ID:      999999999
Email:   dbrown@wlu.ca
__updated__ = "2018-03-21"
-------------------------------------------------------
"""
from copy import deepcopy


class List:

    def __init__(self):
        """
        -------------------------------------------------------
        Initializes an empty list.
        Use: l = List()
        -------------------------------------------------------
        Postconditions:
            Initializes an empty list.
        -------------------------------------------------------
        """
        self._values = []
        return

    def empty(self):
        """
        -------------------------------------------------------
        Determines if the list is empty.
        Use: b = l.empty()
        -------------------------------------------------------
        Postconditions:
            returns
            True if the list is empty, False otherwise.
        -------------------------------------------------------
        """
        return len(self._values) == 0

    def is_empty(self):
        """
        -------------------------------------------------------
        Determines if the list is empty.
        Use: b = l.empty()
        -------------------------------------------------------
        Postconditions:
            returns
            True if the list is empty, False otherwise.
        -------------------------------------------------------
        """
        return len(self._values) == 0

    def __len__(self):
        """
        -------------------------------------------------------
        Returns the size of the list.
        Use: n = len(l)
        -------------------------------------------------------
        Postconditions:
            Returns the number of values in the list.
        -------------------------------------------------------
```

OneClass find more resources at oneclass.com

```
        """
        return len(self._values)

    def insert(self, i, value):
        """
        -------------------------------------------------------
        Inserts a copy of value into the list at index i.
        Use: l.insert(i, value)
        -------------------------------------------------------
        Preconditions:
            i - index value (int)
            value - a data element (?)
        Postconditions:
            a copy of value is added to index i, all other values are pushed
right
            If i outside of range of length of list, appended to end
        -------------------------------------------------------
        """
        self._values = self._values[:i] + [deepcopy(value)] + self._values[i:]
        return

    def _linear_search(self, key):
        """
        -------------------------------------------------------
        Searches for the first occurrence of key in the list.
        Private helper method - used only by other ADT methods.
        Use: i = self._linear_search(key)
        -------------------------------------------------------
        Preconditions:
            key - a partial data element (?)
        Postconditions:
            returns
            i - the index of key in the list, -1 if key is not found (int)
        -------------------------------------------------------
        """
        n = len(self._values)
        i = 0

        while i < n and self._values[i] != key:
            i += 1

        if i == n:
            i = -1
        return i

    def remove(self, key):
        """
        -------------------------------------------------------
        Finds, removes, and returns the first value in list that matches key.
        Use: value = l.remove(key)
        -------------------------------------------------------
        Preconditions:
            key - a partial data element (?)
        Postconditions:
            returns
            value - the full value matching key, otherwise None (?)
        -------------------------------------------------------
        """
        assert len(self._values) > 0, "Cannot remove from an empty list"

        # Key argument exists - search list for key.
        i = self._linear_search(key)

        if i > -1:
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
            value = self._values.pop(i)
        else:
            value = None
        return value

    def find(self, key):
        """
        -------------------------------------------------------
        Finds and returns a copy of the first value in list that matches key.
        Use: value = l.find(key)
        -------------------------------------------------------
        Preconditions:
            key - a partial data element (?)
        Postconditions:
            returns
            value - a copy of the full value matching key, otherwise None (?)
        -------------------------------------------------------
        """
        i = self._linear_search(key)

        if i > -1:
            value = deepcopy(self._values[i])
        else:
            value = None
        return value

    def peek(self):
        """
        -------------------------------------------------------
        Returns a copy of the first value in list.
        Use: value = l.peek()
        -------------------------------------------------------
        Postconditions:
            returns
            value - a copy of the first value in the list (?)
        -------------------------------------------------------
        """
        assert len(self._values) > 0, "Cannot peek at an empty list"

        value = deepcopy(self._values[0])
        return value

    def index(self, key):
        """
        -------------------------------------------------------
        Finds location of a value by key in list.
        Use: n = l.index(key)
        -------------------------------------------------------
        Preconditions:
            key - a partial data element (?)
        Postconditions:
            returns
            i - the index of the location of key in the list, -1 if
              key is not in the list. (int)
        -------------------------------------------------------
        """
        i = self._linear_search(key)
        return i

    def _valid_index(self, i):
        """
        -------------------------------------------------------
        Private helper method to validate an index value.
        Python index values can be positive or negative and range from
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
          -len(list) to len(list) - 1
        Use: assert self._valid_index(i)
        -------------------------------------------------------
        Preconditions:
            i - an index value (int)
        Postconditions:
            returns
            True if i is a valid index, False otherwise.
        -------------------------------------------------------
        """
        n = len(self._values)
        return -n <= i < n

    def __getitem__(self, i):
        """
        ---------------------------------------------------------
        Returns a copy of the nth element of the list.
        Use: value = l[i]
        -------------------------------------------------------
        Preconditions:
            i - index of the element to access (int)
        Postconditions:
            returns
            value - the i-th element of list (?)
        -------------------------------------------------------
        """
        assert self._valid_index(i), "Invalid index value"

        value = deepcopy(self._values[i])
        return value

    def __setitem__(self, i, value):
        """
        ---------------------------------------------------------
        Places a copy of value into the list at position n.
        Use: l[i] = value
        -------------------------------------------------------
        Preconditions:
            i - index of the element to access (int)
            value - a data value (?)
        Postconditions:
            The i-th element of list contains a copy of value. The existing
                value at i is overwritten.
        -------------------------------------------------------
        """
        assert self._valid_index(i), "Invalid index value"

        self._values[i] = deepcopy(value)
        return

    def __contains__(self, key):
        """
        ---------------------------------------------------------
        Determines if the list contains key.
        Use: b = key in l
        -------------------------------------------------------
        Preconditions:
            key - a partial data element (?)
        Postconditions:
            returns
            True if the list contains key, False otherwise. (boolean)
        -------------------------------------------------------
        """
        i = self._linear_search(key)
```

OneClass find more resources at oneclass.com

```
        return i > -1

    def max(self):
        """
        -------------------------------------------------------
        Finds the maximum value in list.
        Use: value = l.max()
        -------------------------------------------------------
        Postconditions:
            returns
            value - a copy of the maximum value in the list (?)
        -------------------------------------------------------
        """
        assert len(self._values) > 0, "Cannot find maximum of an empty list"

        max_index = 0
        i = 1
        n = len(self._values)

        while i < n:
            if self._values[i] > self._values[max_index]:
                max_index = i
            i += 1
        value = deepcopy(self._values[max_index])
        return value

    def min(self):
        """
        -------------------------------------------------------
        Finds the minimum value in list.
        Use: value = l.min()
        -------------------------------------------------------
        Postconditions:
            returns
            value - a copy of the minimum value in the list (?)
        -------------------------------------------------------
        """
        assert len(self._values) > 0, "Cannot find minimum of an empty list"

        min_index = 0
        i = 1
        n = len(self._values)

        while i < n:
            if self._values[i] < self._values[min_index]:
                min_index = i
            i += 1
        value = deepcopy(self._values[min_index])
        return value

    def count(self, key):
        """
        -------------------------------------------------------
        Finds the number of times key appears in list.
        Use: n = l.count(key)
        -------------------------------------------------------
        Preconditions:
            key - a partial data element (?)
        Postconditions:
            returns
            number - number of times key appears in list (int)
        -------------------------------------------------------
        """
        assert len(self._values) > 0, "Cannot count keys in an empty list"
```

```
        number = 0
        i = 1
        n = len(self._values)

        while i < n:
            if self._values[i] == key:
                number += 1
            i += 1
        return number

    def reverse(self):
        """
        -------------------------------------------------------
        Reverses the order of the elements in list.
        -------------------------------------------------------
        Postconditions:
            The contents of list are reversed in order with respect
            to their order before the operation was called.
        -------------------------------------------------------
        """
        n = len(self._values)
        mid = n // 2
        i = 0

        while i < mid:
            j = n - i - 1
            self._values[i], self._values[j] = self._values[j], self._values[i]
            i += 1
        return

    def append(self, value):
        """
        ---------------------------------------------------------
        Appends a copy of value to the end of the list.
        Use: l.append(value)
        -------------------------------------------------------
        Preconditions:
            value - a data element (?)
        Postconditions:
            a copy of value is added to the end of the list.
        -------------------------------------------------------
        """
        self._values.append(deepcopy(value))
        return

    def clean(self):
        """
        ---------------------------------------------------------
        Removes duplicates from the list.
        Use: l.clean()
        -------------------------------------------------------
        Postconditions:
            The list contains one and only one of each value formerly present
            in the list. The first occurrence of each value is preserved.
        -------------------------------------------------------
        """
        i = 0

        while i < len(self._values):
            # Check current value against the rest of the values.
            j = i + 1

            while j < len(self._values):
```

OneClass find more resources at oneclass.com

```
                if self._values[i] == self._values[j]:
                    # Delete the extra value.
                    self._values.pop(j)
                else:
                    j += 1
            i += 1
        return

    def pop(self, *args):
        """
        -------------------------------------------------------
        Finds, removes, and returns the value in list whose index matches i.
        Use: value = l.pop()
        Use: value = l.pop(i)
        -------------------------------------------------------
        Preconditions:
            args - an array of arguments (tuple of int)
                args[0], if it exists, is the index i
        Postconditions:
            returns
            value - if args exists, the value at position args[0], otherwise the
last
                value in the list, value is removed from the list (?)
        -------------------------------------------------------
        """
        assert len(self._values) > 0, "Cannot pop from an empty list"
        assert len(args) <= 1, "No more than 1 argument allowed"

        if len(args) == 1:
            # pop the element at position i
            i = args[0]
            value = self._values.pop(i)
        else:
            # pop the last element
            value = self._values.pop()
        return value

    def _swap(self, i, j):
        """
        -------------------------------------------------------
        Swaps the position of two elements in the data list.
        Private helper operations called only from within class.
        -------------------------------------------------------
        Preconditions:
            i - index of one element to switch (int, 0 <= i < len(self._values))
            j - index of other element to switch (int, 0 <= j <
len(self._values))
        Postconditions:
            The element originally at position i is now at position j,
            and visa versa.
        -------------------------------------------------------
        """
        assert self._valid_index(i), "Invalid index i"
        assert self._valid_index(j), "Invalid index j"

        temp = self._values[i]
        self._values[i] = self._values[j]
        self._values[j] = temp
        return

    def _linear_search_r(self, key):
        """
        -------------------------------------------------------
        Searches for the first occurrence of key in the list.
```

OneClass find more resources at oneclass.com

```
        Private helper method - used only by other ADT methods.
        Use: i = self._linear_search_r(key)
        -------------------------------------------------------
        Preconditions:
            key - a partial data element (?)
        Postconditions:
            returns
            i - the index of key in the list, -1 if key is not found (int)
        -------------------------------------------------------
        """
        i = self._linear_search_r_aux(key, 0, len(self._values))
        return i

    def _linear_search_r_aux(self, key, i, n):
        """
        -------------------------------------------------------
        Searches for the first occurrence of key in the list.
        Private helper method - used only by other ADT methods.
        Use: i = self._linear_search(key)
        -------------------------------------------------------
        Preconditions:
            key - a partial data element (?)
        Postconditions:
            returns
            i - the index of key in the list, -1 if key is not found (int)
        -------------------------------------------------------
        """
        if i == n:
            i = -1
        elif self._values[i] != key:
            i = self._linear_search_r_aux(key, i + 1, n)
        return i

    def insert_front(self, value):
        """
        -------------------------------------------------------
        Inserts a copy of value into the front of the list.
        Use: l.insert_front(value)
        -------------------------------------------------------
        Preconditions:
            value - a data element. (?)
        Postconditions:
            value is added to the front of the list.
        -------------------------------------------------------
        """
        self._values.insert(0, deepcopy(value))
        return

    def remove_front(self):
        """
        -------------------------------------------------------
        Removes the first node in the list.
        Use: value = l.remove_front()
        -------------------------------------------------------
        Postconditions:
            returns
            value - the first value in the list (?)
        -------------------------------------------------------
        """
        assert len(self._values) > 0, "Cannot remove from an empty list"

        value = self._values.pop(0)
        return value
```

OneClass find more resources at oneclass.com

```
    def remove_many(self, key):
        """
        -------------------------------------------------------
        Finds and removes all values in the list that match key.
        Use: l.remove_many(key)
        -------------------------------------------------------
        Preconditions:
            key - a data element (?)
        Postconditions:
            Removes all values matching key.
        -------------------------------------------------------
        """
        i = 0

        while i < len(self._values):
            if self._values[i] == key:
                self._values.pop(i)
            else:
                i = i + 1
        return

    def identical(self, rs):
        """
        ---------------------------------------------------------
        Determines whether two lists are identical, i.e. same values appear
        in the same locations in both lists. (iterative version)
        Use: b = l.identical(rs)
        -------------------------------------------------------
        Preconditions:
            rs - another list (List)
        Postconditions:
            returns
            is_identical - True if this list contains the same values as rs
            in the same order, otherwise False. (boolean)
        -------------------------------------------------------
        """
        is_identical = False
        n = len(self._values)

        # Lists must be the same length to be identical.
        if len(rs) == n:
            i = 0

            # Compare each element, stop when the elements are not the same
            # or the end of the list is reached.
            while i < n and self._values[i] == rs._values[i]:
                i += 1

            if i == n:
                is_identical = True
        return is_identical

    def identical_r(self, rs):
        """
        ---------------------------------------------------------
        Determines whether two lists are identical, i.e. same values appear
        in the same locations in both lists. (recursive version)
        Use: b = l.identical_r(rs)
        -------------------------------------------------------
        Preconditions:
            rs - another list (List)
        Postconditions:
            returns
            is_identical - True if this list contains the same values as rs
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
            in the same order, otherwise False. (boolean)
        -------------------------------------------------------
        """
        if len(rs) == len(self._values):
            is_identical = self._identical_r_aux(self._values, rs._values)
        else:
            is_identical = False
        return is_identical

    def _identical_r_aux(self, data1, data2):
        """
        ---------------------------------------------------------
        Recursive auxiliary function for identical_r.
        Use: b = self._identical_r_aux(self._values[?:], rs._values[?:])
        -------------------------------------------------------
        Preconditions:
            data1 - the current Python list (list)
            data2 - the rs Python list (list)
        Postconditions:
            returns
            is_identical - True if this list contains the same values as rs
            in the same order, otherwise False. (boolean)
        -------------------------------------------------------
        """
        if len(data1) == 0:
            is_identical = True
        elif data1[0] == data2[0]:
            is_identical = self._identical_r_aux(data1[1:], data2[1:])
        else:
            is_identical = False
        return is_identical

    def intersection(self, rs):
        """
        -------------------------------------------------------
        Returns a list that contains only values that appear in both
        the current List and rs.
        Use: l2 = l1.intersection(rs)
        -------------------------------------------------------
        Preconditions:
            rs - another List (List)
        Postconditions:
            returns
            new_list - A List that contains only the values found in both
            the current List and rs. Values do not repeat. (List)
        -------------------------------------------------------
        """
        new_list = List()
        m = len(self._values)
        n = len(rs._values)
        i = 0

        while i < m:
            # Walk through every value in self
            key = self._values[i]
            j = 0

            # Look in new_list for key
            while j < len(new_list._values) and key != new_list._values[j]:
                j += 1

            if j == len(new_list._values):
                # key is not in new_list
                k = 0
```

OneClass find more resources at oneclass.com

```
            # look in rs for key
            while k < n and key != rs._values[k]:
                k += 1

            if k < n:
                # key is in rs - add key to new_list
                new_list._values.append(key)
        i += 1
    return new_list

def union(self, rs):
    """
    -------------------------------------------------------
    Returns a list that contains all values in both
    the current List and rs.
    Use: nl = l.union(rs)
    -------------------------------------------------------
    Preconditions:
        rs - another list (List)
    Postconditions:
        returns
        new_list - contains all values found in both the current
        List and rs. Values do not repeat. (List)
    -------------------------------------------------------
    """
    new_list = List()
    i = 0
    n = len(self._values)

    while i < n:
        v = self._values[i]
        # Walk through current list.
        if new_list._linear_search(v) == -1:
            # Value is not in new list.
            new_list._values.append(v)
        i += 1

    i = 0
    n = len(rs._values)

    while i < n:
        v = rs._values[i]
        # Walk through other list.
        if new_list._linear_search(v) == -1:
            # Value is not in new list.
            new_list._values.append(v)
        i += 1
    return new_list

def split_alt(self):
    """
    -------------------------------------------------------
    Split a List into two parts. even contains the even indexed
    elements, odd contains the odd indexed elements.
    Order of even and odd is not significant. (iterative version)
    Use: even, odd = l.split_alt()
    -------------------------------------------------------
    Postconditions:
        returns
        even - the even indexed elements of the list (List)
        odd - the odd indexed elements of the list (List)
            The List is empty.
    -------------------------------------------------------
```

OneClass find more resources at oneclass.com

```
        """
        even = List()
        odd = List()
        i = 0

        while len(self._values) > 0:

            if i % 2 == 0:
                even._values.append(self._values.pop(0))
            else:
                odd._values.append(self._values.pop(0))
            i += 1
        return even, odd

    def split(self):
        """
        -------------------------------------------------------
        Splits list into two parts. ls contains the first half,
        rs the second half. Current list becomes empty.
        Use: ls, rs = l.split()
        -------------------------------------------------------
        Postconditions:
            returns
            ls - a new List with >= 50% of the original List (List)
            rs - a new List with <= 50% of the original List (List)
        -------------------------------------------------------
        """
        ls = List()
        rs = List()
        # Split
        middle = len(self._values) // 2 + len(self._values) % 2
        ls._values = self._values[:middle]
        rs._values = self._values[middle:]
        self._values = []
        return ls, rs

    def split_apply(self, func):
        """
        -------------------------------------------------------
        Splits list into two parts. ls contains all the values
        where the result of calling func(value) is True,
        rs contains the remaining values.
        Use: ls, rs = l.split_apply(func)
        -------------------------------------------------------
        Preconditions:
            func - a function that given a value in the list returns
                True for some condition, otherwise returns False.
        Postconditions:
            returns
            ls - a new List with values where func(value) is True (List)
            rs - a new List with values where func(value) is False (List)
            self is empty. Order of values is new lists is maintained.
        -------------------------------------------------------
        """
        ls = List()
        rs = List()
        # Split
        while len(self._values) > 0:
            value = self._values.pop()

            if(func(value)):
                ls.insert_front(value)
            else:
                rs.insert_front(value)
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        return ls, rs

    def copy(self):
        """
        -------------------------------------------------------
        Duplicates the current list to a new list in the same order.
        Use: new_list = l.copy()
        -------------------------------------------------------
        Postconditions:
            returns
            new_list - a copy of self (List)
        -------------------------------------------------------
        """
        new_list = List()
        new_list._values = deepcopy(self._values)
        return new_list

    def combine(self, s2):
        """
        -------------------------------------------------------
        Combines contents of two lists into a third.
        Use: new_list = l1.combine(s2)
        -------------------------------------------------------
        Preconditions:
            s2- an array-based List (List)
        Postconditions:
            returns
            new_list - the contents of the current List and s2
                are interlaced into new_list - current List and s2
                are empty (List)
        -------------------------------------------------------
        """
        new_list = List()

        while len(self._values) > 0 and len(s2._values) > 0:
            new_list._values.append(self._values.pop(0))
            new_list._values.append(s2._values.pop(0))

        while len(self._values) > 0:
            new_list._values.append(self._values.pop(0))

        while len(s2._values) > 0:
            new_list._values.append(s2._values.pop(0))

        return new_list

    def __iter__(self):
        """
        USE FOR TESTING ONLY
        -------------------------------------------------------
        Generates a Python iterator. Iterates through the list
        from front to rear.
        Use: for v in q:
        -------------------------------------------------------
        Postconditions:
            returns
            value - the next value in the list (?)
        -------------------------------------------------------
        """
        for value in self._values:
            yield value
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
"""
-------------------------------------------------------
bst_linked.py
Linked version of the BST ADT.
-------------------------------------------------------
Author:  David Brown
ID:      999999999
Email:   dbrown@wlu.ca
__updated__ = "2018-03-21"
-------------------------------------------------------
"""
# Imports
from copy import deepcopy
from stack_array import Stack


class _BSTNode:

    def __init__(self, value):
        """
        -------------------------------------------------------
        Creates a node containing a copy of value.
        Use: node = _BSTNode(value)
        -------------------------------------------------------
        Preconditions:
            value - data for the node (?)
        Postconditions:
            Initializes a BST node containing value. Child pointers are None,
            height is 1.
        -------------------------------------------------------
        """
        self._data = deepcopy(value)
        self._left = None
        self._right = None
        self._height = 1
        return

    def _update_height(self):
        """
        -------------------------------------------------------
        Updates the height of the current node.
        Use: node._update_height()
        -------------------------------------------------------
        Postconditions:
            _height is 1 plus the maximum of the node's (up to) two children.
        -------------------------------------------------------
        """
        if self._left is None:
            left_height = 0
        else:
            left_height = self._left._height

        if self._right is None:
            right_height = 0
        else:
            right_height = self._right._height

        self._height = max(left_height, right_height) + 1
        return

    def __str__(self):
        """
        USE FOR TESTING ONLY
        -------------------------------------------------------
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        Returns node height and value as a string - for debugging.
        -------------------------------------------------------
        """
        return "h: {}, v: {}".format(self._height, self._data)


class BST:

    def __init__(self):
        """
        -------------------------------------------------------
        Initializes an empty BST.
        Use: bst = BST()
        -------------------------------------------------------
        Postconditions:
            Initializes an empty bst.
        -------------------------------------------------------
        """
        self._root = None
        self._count = 0
        return

    def empty(self):
        """
        -------------------------------------------------------
        Determines if bst is empty.
        Use: b = bst.empty()
        -------------------------------------------------------
        Postconditions:
            returns
            True if bst is empty, False otherwise.
        -------------------------------------------------------
        """
        return self._root is None

    def is_empty(self):
        """
        -------------------------------------------------------
        Determines if bst is empty.
        Use: b = bst.empty()
        -------------------------------------------------------
        Postconditions:
            returns
            True if bst is empty, False otherwise.
        -------------------------------------------------------
        """
        return self._root is None

    def __len__(self):
        """
        -------------------------------------------------------
        Returns the number of nodes in the BST.
        Use: n = len(bst)
        -------------------------------------------------------
        Postconditions:
            returns
            the number of nodes in the BST.
        -------------------------------------------------------
        """
        return self._count

    def insert(self, value):
        """
        -------------------------------------------------------
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        Inserts a copy of value into the bst.
        Use: b = bst.insert(value)
        -------------------------------------------------------
        Preconditions:
            value - data to be inserted into the bst (?)
        Postconditions:
            returns
            inserted - True if value is inserted into the BST,
            False otherwise. Values may appear only once in a tree. (boolean)
        -------------------------------------------------------
        """
        self._root, inserted = self._insert_aux(self._root, value)
        return inserted

    def _insert_aux(self, node, value):
        """
        -------------------------------------------------------
        Inserts a copy of _data into node.
        Private recursive operation called only by insert.
        Use: node, inserted = self._insert_aux(node, value)
        -------------------------------------------------------
        Preconditions:
            node - a bst node (_BSTNode)
            value - data to be inserted into the node (?)
        Postconditions:
            returns
            node - the current node (_BSTNode)
            inserted - True if value is inserted into the BST,
            False otherwise. Values may appear only once in a tree. (boolean)
        -------------------------------------------------------
        """
        if node is None:
            # Base case: add a new node containing the value.
            node = _BSTNode(value)
            self._count += 1
            inserted = True
        elif node._data > value:
            # General case: check the left subtree.
            node._left, inserted = self._insert_aux(node._left, value)
        elif node._data < value:
            # General case: check the right subtree.
            node._right, inserted = self._insert_aux(node._right, value)
        else:
            # Base case: value is already in the BST.
            inserted = False

        if inserted:
            # Update the node height if any of its children have been changed.
            node._update_height()
        return node, inserted

    def retrieve(self, key):
        """
        -------------------------------------------------------
        Retrieves a copy of a value matching key in a BST. (Iterative)
        Use: v = bst.retrieve(key)
        -------------------------------------------------------
        Preconditions:
            key - data to search for (?)
        Postconditions:
            returns
            value - value in the node containing key, otherwise None (?)
        -------------------------------------------------------
        """
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        node = self._root
        value = None

        while node is not None and value is None:

            if node._data > key:
                node = node._left
            elif node._data < key:
                node = node._right
            elif node._data == key:
                # for comparison counting
                value = deepcopy(node._data)
        return value

    def remove(self, key):
        """
        -------------------------------------------------------
        Removes a node with a value matching key from the bst.
        Returns the value matched.
        Use: value = bst.remove(key)
        -------------------------------------------------------
        Preconditions:
            key - data to search for (?)
        Postconditions:
            returns
            value - value matching key if found,
            otherwise returns None. Update structure of bst as required.
        -------------------------------------------------------
        """
        assert self._root is not None, "Cannot remove from an empty BST"

        self._root, value = self._remove_aux(self._root, key)
        return value

    def _remove_aux(self, node, key):
        """
        -------------------------------------------------------
        Attempts to find a value matching key in a BST node. Deletes the node
        if found and returns the sub-tree root.
        Private recursive operation called only by remove.
        Use: node, value = self._remove_aux(node, key)
        -------------------------------------------------------
        Preconditions:
            node - a bst node to search for key (_BSTNode)
            key - data to search for (?)
        Postconditions:
            returns
            node - the current node or its replacement (_BSTNode)
            value - value in node containing key, None otherwise.
        -------------------------------------------------------
        """
        if node is None:
            # Base Case: the key is not in the tree.
            value = None
        elif key < node._data:
            # Search the left subtree.
            node._left, value = self._remove_aux(node._left, key)
        elif key > node._data:
            # Search the right subtree.
            node._right, value = self._remove_aux(node._right, key)
        else:
            # Value has been found.
            value = node._data
            self._count -= 1
```

OneClass find more resources at oneclass.com

```
            # Replace this node with another node.
            if node._left is None and node._right is None:
                # node has no children.
                node = None
            elif node._left is None:
                # node has no left child.
                node = node._right
            elif node._right is None:
                # node has no right child.
                node = node._left
            else:
                # Node has two children
                if node._left._right is None:
                    # left child is replacement node
                    repl_node = node._left
                else:
                    # find replacement node in right subtree of left node
                    repl_node = self._delete_node_left(node._left)
                    repl_node._left = node._left

                repl_node._right = node._right
                node = repl_node

        if node is not None and value is not None:
            # If the value was found, update the ancestor heights.
            node._update_height()
        return node, value

    def _delete_node_left(self, parent):
        """
        -------------------------------------------------------
        Finds a replacement node for a node to be removed from the tree.
        Private operation called only by _remove_aux.
        Use: repl_node = self._delete_node_left(node, node._right)
        -------------------------------------------------------
        Preconditions:
            parent - node to search for largest value (_BSTNode)
        Postconditions:
            returns
            repl_node - the node that replaces the deleted node. This node
            is the node with the maximum value in the deleted node's left
            subtree (_BSTNode)
        -------------------------------------------------------
        """
        child = parent._right

        if child._right is None:
            # child has largest value in left subtree
            repl_node = child
            # move child's left tree up
            parent._right = child._left
        else:
            repl_node = self._delete_node_left(child)

        # Recursively update all parent node heights
        parent._update_height()
        return repl_node

    def remove_root(self):
        """
        -------------------------------------------------------
        Removes the root node and returns its value.
        Use: value = bst._remove_root()
        -------------------------------------------------------
```

```
        Postconditions:
            returns
            value - value in root.
        -------------------------------------------------------
        """
        assert self._root is not None, "Cannot remove the room of an empty BST"

        # Value has been found.
        value = self._root._data
        self._count -= 1
        # Replace the root with another node.

        if self._root._left is None and self._root._right is None:
            # root has no children.
            self._root = None
        elif self._root._left is None:
            # root has no left child.
            self._root = self._root._right
        elif self._root._right is None:
            # root has no right child.
            self._root = self._root._left
        else:
            # root has two children
            if self._root._left._right is None:
                # left child is replacement node
                repl_node = self._root._left
            else:
                # find replacement node in right subtree of left node
                repl_node = self._delete_node_left(self._root._left)
                repl_node._left = self._root._left

            repl_node._right = self._root._right
            self._root = repl_node

        if self._root is not None:
            # Update the root height.
            self._root._update_height()
        return value

    def __contains__(self, key):
        """
        ---------------------------------------------------------
        Determines if the bst contains key.
        Use: b = key in bst
        -------------------------------------------------------
        Preconditions:
            key - a comparable data element (?)
        Postconditions:
            returns
            True if the bst contains key, False otherwise.
        -------------------------------------------------------
        """
        value = self.retrieve(key)
        return value is not None

    def height(self):
        """
        -------------------------------------------------------
        Returns the maximum height of a BST, i.e. the length of the
        largest path from root to a leaf node in the tree.
        Use: h = bst.height()
        -------------------------------------------------------
        Postconditions:
            returns
```

OneClass find more resources at oneclass.com

```
            maximum height of bst (int)
        -------------------------------------------------------
        """
        if self._root is None:
            h = 0
        else:
            h = self._root._height
        return h

    def identical(self, rs):
        """
        ---------------------------------------------------------
        Determines whether two BSTs are identical.
        Use: b = bst.identical(rs)
        -------------------------------------------------------
        Preconditions:
            rs - another bst (BST)
        Postconditions:
            returns
            is_identical - True if this bst contains the same values
            in the same order as rs, otherwise returns False (boolean)
        -------------------------------------------------------
        """
        if self._count != rs._count:
            is_identical = False
        else:
            is_identical = self._identical_aux(self._root, rs._root)
        return is_identical

    def _identical_aux(self, node1, node2):
        """
        ---------------------------------------------------------
        Determines whether two subtrees are identical.
        Use: b = self._identical_aux(node1, node2)
        -------------------------------------------------------
        Preconditions:
            node1 - node of the current BST (_BSTNode)
            node2 - node of the rs BST (_BSTNode)
        Postconditions:
            returns
            result - True if this stubtree contains the same values as rs
                subtree in the same order, otherwise returns False (boolean)
        -------------------------------------------------------
        """
        if node1 is None and node2 is None:
            # Reached a bottom of the tree.
            result = True
        elif node1 is not None and node2 is not None \
                and node1._data == node2._data and node1._height ==
node2._height:
            result = self._identical_aux(node1._left, node2._left) \
                and self._identical_aux(node1._right, node2._right)
        else:
            result = False
        return result

    def parent_i(self, key):
        """
        ---------------------------------------------------------
        Returns the value of the parent node of a key node in a bst.
        ---------------------------------------------------------
        Preconditions:
            key - a key value (?)
        Postconditions:
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
            returns
            value - a copy of the value in a node that is the parent of the
            key node, None if the key is not found. (?)
        -------------------------------------------------------
        """
        assert self._root is not None, "Cannot locate a parent in an empty BST"

        # Find the node containing the key.
        node = self._root
        parent = None
        found = False

        while node is not None and found is False:

            if key < node._data:
                parent = node
                node = node._left
            elif key > node._data:
                parent = node
                node = node._right
            else:
                found = True

        if parent is None or not found:
            value = None
        else:
            value = deepcopy(parent._data)
        return value

    def parent_r(self, key):
        """
        -------------------------------------------------------
        Returns the value of the parent node in a bst given a key.
        -------------------------------------------------------
        Preconditions:
            key - a key value (?)
        Postconditions:
            returns
            value - a copy of the value in a node that is the parent of the
            key node, None if the key is not found.
        -------------------------------------------------------
        """
        assert self._root is not None, "Cannot locate a parent in an empty BST"

        # Find the node containing the key _data.
        return self._parent_aux(self._root, key, None)

    def _parent_aux(self, node, key, parent):
        """
        -------------------------------------------------------
        Returns the _data of the parent node in a bst given a _data.
        Private recursive operation called only by parent_r.
        Use: v = self._parent_aux(node, key, parent of node)
        -------------------------------------------------------
        Preconditions:
            node - a BST node
            key - a key _data
            parent - the parent node of the current node
        Postconditions:
            returns
            value - the value of the parent node, None if it has no parent (?)
        -------------------------------------------------------
        """
        if node is None:
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
            # Base Case: the key is not in the tree.
            value = None
        elif key < node._data:
            # General Case: Search the left subtree.
            value = self._parent_aux(node._left, key, node)
        elif key > node._data:
            # General Case: Search the right subtree.
            value = self._parent_aux(node._right, key, node)
        elif parent is None:
            # Base Case: Value has been found, but without a parent node.
            value = None
        else:
            # Base Case: Value has been found. Return the parent value.
            value = deepcopy(parent._data)
        return value

    def max(self):
        """
        -------------------------------------------------------
        Finds the maximum value in BST. (Iterative algorithm)
        Use: value = bst.max()
        -------------------------------------------------------
        Postconditions:
            returns
            value - a copy of the maximum value in the BST (?)
        -------------------------------------------------------
        """
        assert self._root is not None, "Cannot find maximum of an empty BST"
        # Find the node containing the largest _data.
        # (It is the right-most node.)
        node = self._root

        while node._right is not None:
            node = node._right

        value = deepcopy(node._data)
        return value

    def max_r(self):
        """
        ---------------------------------------------------------
        Returns the largest value in a bst. (Recursive algorithm)
        Use: value = bst.max_r()
        ---------------------------------------------------------
        Postconditions:
            returns
            value - a copy of the maximum value in the BST (?)
        ---------------------------------------------------------
        """
        assert self._root is not None, "Cannot find maximum of an empty BST"

        value = self._max_aux(self._root)
        return value

    def _max_aux(self, node):
        """
        ---------------------------------------------------------
        Returns the largest value in a BST node. (Recursive algorithm)
        Use: v = self._max_aux(node)
        ---------------------------------------------------------
        Preconditions:
            node - valid linked BST node (_BSTNode)
        Postconditions:
            returns
```

OneClass find more resources at oneclass.com

```
            value - a copy of the largest value in the node subtree (?)
        -------------------------------------------------------
        """
        # Find the node containing the largest _data.
        # (It is the right-most node.)
        if node._right is None:
            value = deepcopy(node._data)
        else:
            value = self._max_aux(node._right)
        return value

    def min(self):
        """
        -------------------------------------------------------
        Finds the minimum value in BST. (Iterative algorithm)
        Use: value = bst.min()
        -------------------------------------------------------
        Postconditions:
            returns
            value - a copy of the minimum value in the BST (?)
        -------------------------------------------------------
        """
        assert self._root is not None, "Cannot find minimum of an empty BST"
        # Find the node containing the smallest _data.
        # (It is the left-most node.)
        node = self._root

        while node._left is not None:
            node = node._left

        value = deepcopy(node._data)
        return value

    def min_r(self):
        """
        -------------------------------------------------------
        Returns the minimum value in a bst. (Recursive algorithm)
        Use: value = bst.min_r()
        -------------------------------------------------------
        Postconditions:
            returns
            value - a copy of the minimum value in the BST (?)
        -------------------------------------------------------
        """
        assert self._root is not None, "Cannot find minimum of an empty BST"

        value = self._min_aux(self._root)
        return value

    def _min_aux(self, node):
        """
        -------------------------------------------------------
        Returns the minimum value in a BST node. (Recursive algorithm)
        Use: v = self._max_aux(node)
        -------------------------------------------------------
        Preconditions:
            node - valid linked BST node (_BSTNode)
        Postconditions:
            returns
            value - a copy of the minimum value in the node subtree (?)
        -------------------------------------------------------
        """
        # Find the node containing the minimum _data.
        # (It is the left-most node.)
```

```
        if node._left is None:
            value = deepcopy(node._data)
        else:
            value = self._min_aux(node._left)
        return value

    def leaf_count(self):
        """
        -------------------------------------------------------
        Returns the number of leaves (nodes with no children) in bst.
        Use: n = bst.leaf_count()
        (Recursive algorithm)
        -------------------------------------------------------
        Postconditions:
            returns
            count - number of nodes with no children in bst (int)
        -------------------------------------------------------
        """
        count = self._leaf_count_aux(self._root)
        return count

    def _leaf_count_aux(self, node):
        """
        -------------------------------------------------------
        Returns the number of leaves (nodes with no children) in bst.
        Use: n = bst.leaf_count()
        (Recursive algorithm)
        -------------------------------------------------------
        Preconditions:
            node - a BST node (_BSTNode)
        Postconditions:
            returns
            count - number of nodes with no children below node (int)
        -------------------------------------------------------
        """
        if node is None:
            count = 0
        elif node._left is None and node._right is None:
            # Base case: node has no children.
            count = 1
        else:
            count = self._leaf_count_aux(node._left) + \
                self._leaf_count_aux(node._right)
        return count

    def two_child_count(self):
        """
        -------------------------------------------------------
        Returns the number of the three types of nodes in a BST.
        Use: zero, one, two = bst.node_counts()
        ------------------------------------------------------
        Postconditions:
            returns
            zero - number of nodes with zero children (int)
            one - number of nodes with one child (int)
            two - number of nodes with two children (int)
        --------------------------------------------------------
        """
        return self._two_child_count_aux(self._root)

    def _two_child_count_aux(self, node):
        """
        -------------------------------------------------------
        Returns the number of types of nodes in a BST node.
```

OneClass find more resources at oneclass.com

```
        -------------------------------------------------------
        Preconditions:
            node - a BST node (_BSTNode)
        Postconditions:
            returns
            zero - number of nodes with zero children (int)
            one - number of nodes with one child (int)
            two - number of nodes with two children (int)
        ----------------------------------------------------------
        """
        if node is None:
            # Base case: node is empty.
            count = 0
        elif node._left is not None and node._right is not None:
            # General case: node has two children.
            count = 1 + self._two_child_count_aux(node._left) + \
                self._two_child_count_aux(node._right)
        else:
            # General case: node has one child.
            count = self._two_child_count_aux(node._left) + \
                self._two_child_count_aux(node._right)
        return count

    def one_child_count(self):
        """
        ---------------------------------------------------------
        Returns the number of the three types of nodes in a BST.
        Use: zero, one, two = bst.node_counts()
        -------------------------------------------------------
        Postconditions:
            returns
          zero - number of nodes with zero children (int)
          one - number of nodes with one child (int)
          two - number of nodes with two children (int)
        ----------------------------------------------------------
        """
        return self._one_child_count_aux(self._root)

    def _one_child_count_aux(self, node):
        """
        ---------------------------------------------------------
        Returns the number of types of nodes in a BST node.
        -------------------------------------------------------
        Preconditions:
          node - a BST node (_BSTNode)
        Postconditions:
            returns
            zero - number of nodes with zero children (int)
            one - number of nodes with one child (int)
            two - number of nodes with two children (int)
        ----------------------------------------------------------
        """
        if node is None:
            # Base case: empty node..
            count = 0
        elif node._left is None and node._right is not None:
            # General case: node has one child.
            count = 1 + self._one_child_count_aux(node._right)
        elif node._left is not None and node._right is None:
            # General case: node has one child.
            count = 1 + self._one_child_count_aux(node._left)
        else:
            # General case: node has two children.
            count = self._one_child_count_aux(node._left) + \
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
                self._one_child_count_aux(node._right)
        return count

    def node_counts(self):
        """
        ---------------------------------------------------------
        Returns the number of the three types of nodes in a BST.
        Use: zero, one, two = bst.node_counts()
        -------------------------------------------------------
        Postconditions:
            returns
            zero - number of nodes with zero children (int)
            one - number of nodes with one child (int)
            two - number of nodes with two children (int)
        ----------------------------------------------------------
        """
        zero, one, two = self._node_counts_aux(self._root)
        return zero, one, two

    def _node_counts_aux(self, node):
        """
        ---------------------------------------------------------
        Returns the number of types of nodes in a BST node.
        -------------------------------------------------------
        Preconditions:
            node - a BST node (_BSTNode)
        Postconditions:
            returns
            zero - number of nodes with zero children (int)
            one - number of nodes with one child (int)
            two - number of nodes with two children (int)
        ----------------------------------------------------------
        """
        if node is None:
            # Base case: no node.
            one = 0
            two = 0
            zero = 0
        elif node._left is None and node._right is None:
            # Base case: node has no children.
            one = 0
            two = 0
            zero = 1
        elif node._left is not None and node._right is None:
            # General case: node has a single left child.
            zero, one, two = self._node_counts_aux(node._left)
            one += 1
        elif node._left is None and node._right is not None:
            # General case: node has a single right child.
            zero, one, two = self._node_counts_aux(node._right)
            one += 1
        else:
            # General case: node has two children. Get node counts
            # from both children.
            zero_l, one_l, two_l = self._node_counts_aux(node._left)
            zero_r, one_r, two_r = self._node_counts_aux(node._right)
            zero = zero_l + zero_r
            one = one_l + one_r
            # Count the current node as one with two children.
            two = two_l + two_r + 1
        return zero, one, two

    def total_depth(self):
        """
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        -------------------------------------------------------
        Returns the total depth of a bst.
        -------------------------------------------------------
        Postconditions:
            returns
            the total depth count - i.e. the sum of all the node depths
            in the tree (int)
        -------------------------------------------------------
        """
        return self._total_depth_aux(self._root)

    def _total_depth_aux(self, node):
        """
        -------------------------------------------------------
        Returns the total depth of a bst.
        -------------------------------------------------------
        Preconditions:
            node - a BST node (_BSTNode)
        Postconditions:
            returns
            the total depth count - i.e. the sum of all the node depths
            in the tree (int)
        -------------------------------------------------------
        """
        if node is None:
            total_depth = 0
        else:
            total_depth = node._height + self._total_depth_aux(node._left) + \
                self._total_depth_aux(node._right)
        return total_depth

    def mirror(self):
        """
        -------------------------------------------------------
        Creates a mirror version of a BST. All nodes are swapped with nodes on
        the other side the tree. Nodes may take the place of an empty spot.
        The resulting tree is a mirror image of the original tree. (Note that
        the mirrored tree is not a BST.) The original BST is unchanged.
        Use: tree = bst.mirror()
        -------------------------------------------------------
        Postconditions:
            returns
            tree - a mirror version of subtree of node.
        -------------------------------------------------------
        """
        tree = BST()
        tree._root = self._mirror_aux(self._root)
        return tree

    def _mirror_aux(self, node):
        """
        -------------------------------------------------------
        Creates a mirror version of a BST. All nodes are swapped with nodes on
        the other side the tree. Nodes may take the place of an empty spot.
        The resulting tree is a mirror image of the original tree. (Note that
        the mirrored BST is no longer a BST itself.)
        Use: self._mirror_aux(node)
        -------------------------------------------------------
        Preconditions:
            node - a binary tree node (_BSTNode)
        Postconditions:
            returns
            tree - a mirror version of subtree of node.
        -------------------------------------------------------
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        """
        if node is not None:
            new_node = _BSTNode(node._data)
            new_node._right = self._mirror_aux(node._left)
            new_node._left = self._mirror_aux(node._right)
        else:
            new_node = None
        return new_node

    def balanced(self):
        """
        ---------------------------------------------------------
        Returns whether a bst is balanced, i.e. the difference in
        height between all the bst's node's left and right subtrees is <= 1.
        Use: b = bst.balanced()
        ---------------------------------------------------------
        Postconditions:
            returns
            is_balanced - True if the bst is balanced, False otherwise (boolean)
        ---------------------------------------------------------
        """
        is_balanced = self._balanced_aux(self._root)
        return is_balanced

    def _balanced_aux(self, node):
        """
        ---------------------------------------------------------
        Determines whether the BST is is_balanced.
        Private operation called only by _is_valid_aux.
        Use: b = self._balanced_aux(node)
        ---------------------------------------------------------
        Preconditions:
            node - the node to check the balance of (_BSTNode)
        Postconditions:
            returns
            balanced - True if node is is balanced, False otherwise (boolean)
        ---------------------------------------------------------
        """
        if node is None or node._height == 1:
            # Base case: node is empty or a leaf, so no children.
            is_balanced = True
        elif abs(self._node_height(node._left) -
                 self._node_height(node._right)) > 1:
            # Base case: left or right subtree is too deep.
            is_balanced = False
        else:
            # General case: check the children of node.
            is_balanced = self._balanced_aux(node._left) and \
                self._balanced_aux(node._right)
        return is_balanced

    def _node_height(self, node):
        """
        ---------------------------------------------------------
        Helper function to determine the height of node - handles empty node.
        Private operation called only by _is_valid_aux.
        Use: h = self._node_height(node)
        ---------------------------------------------------------
        Preconditions:
            node - the node to get the height of (_BSTNode)
        Postconditions:
            returns
            height - 0 if node is None, node._height otherwise (int)
        ---------------------------------------------------------
```

OneClass find more resources at oneclass.com

```
        """
        if node is None:
            height = 0
        else:
            height = node._height
        return height

    def retrieve_r(self, key):
        """
        -------------------------------------------------------
        Retrieves a _data in a BST. (Recursive)
        Use: v = bst.retrieve(key)
        -------------------------------------------------------
        Preconditions:
            key - data to search for (?)
        Postconditions:
            returns
            value - If bst contains key, returns value, else returns None.
        -------------------------------------------------------
        """
        # Find the node containing the key _data.
        value = self._retrieve_r_aux(self._root, key)
        return value

    def _retrieve_r_aux(self, current, key):
        """
        -------------------------------------------------------
        Retrieves a _data in a BST.
        -------------------------------------------------------
        Preconditions:
            current - a bst node (_BSTNode)
            key - data to search for (?)
        Postconditions:
            returns
            value - contains key, else returns None (?)
        -------------------------------------------------------
        """
        if current is None:
            # Base case: at bottom of tree and key not found.
            value = None
        else:
            if key < current._data:
                value = self._retrieve_r_aux(current._left, key)
            elif key > current._data:
                value = self._retrieve_r_aux(current._right, key)
            else:
                value = deepcopy(current._data)
        return value

    def average_depth(self):
        """
        ---------------------------------------------------------
        Returns the average depth of a bst.
        ---------------------------------------------------------
        Postconditions:
            returns
            avg-depth - total depth count divided by the number of nodes
                in the tree (int)
        ---------------------------------------------------------
        """
        count = self._count
        total_depth = self.total_depth()
        avg_depth = total_depth / count
        return avg_depth
```

OneClass find more resources at oneclass.com

```
    def valid(self):
        """
        -------------------------------------------------------
        Determines if a tree is a valid BST, i.e. the values in all left nodes
        are smaller than their parent, and the values in all right nodes are
        larger than their parent, and height of any node is 1 + max height of
        its children.
        Use: b = bst.valid()
        -------------------------------------------------------
        Postconditions:
            returns
            is_valid - True if tree is a BST, False otherwise (boolean)
        -------------------------------------------------------
        """
        is_valid = self._valid_aux(self._root)
        return is_valid

    def _valid_aux(self, node):
        """
        -------------------------------------------------------
        Determines if a subtree is a valid BST.
        -------------------------------------------------------
        Preconditions:
            node - a binary tree node (_BSTNode)
        Postconditions:
            returns
            is_valid - True if node is root of a valid BST, False otherwise
(boolean)
        -------------------------------------------------------
        """
        if node is None:
            is_valid = True
        elif node._left is not None and node._left._data >= node._data \
                or node._right is not None and node._right._data <= node._data:
            # print("BST Violation at value: {}".format(node._data))
            is_valid = False
        elif node._height <= self._node_height(node._left) \
                or node._height <= self._node_height(node._right):
            # print("Height Violation at value: {}".format(node._data))
            is_valid = False
        else:
            is_valid = self._valid_aux(node._left) and \
                self._valid_aux(node._right)
        return is_valid

    def update(self, value, update):
        """
        -------------------------------------------------------
        Updates a value in a bst by applying a function to it.
        Use: bst.update(value, func)
        -------------------------------------------------------
        Preconditions:
            value - a comparable part of a data element (?)
            update - an update function compatible with value (function)
        Postconditions:
            returns
            updated - True if value is in bst and is updated, False if
            value is not in bst, but adds value to bst in that case.
            (Iterative algorithm.)
        ------------------------------------------------------- -
        """
        parent = None
        current = self._root
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        updated = False

        while current is not None:
            # Fine the location for the new node.
            parent = current

            if value < current._data:
                location = BST._LEFT
                current = current._left
            elif value > current._data:
                location = BST._RIGHT
                current = current._right
            else:
                # value is already in bst.
                current._data = update(current._data)
                updated = True
                location = None
                current = None

        if parent is None:
            self._root = _BSTNode(value)
        elif location == BST._LEFT:
            parent._left = _BSTNode(value)
        elif location == BST._RIGHT:
            parent._right = _BSTNode(value)
        return updated

    def update_r(self, key, update):
        """
        ---------------------------------------------------------
        Updates a value in a bst by applying a function to it.
        Use: bst.update(value, func)
        ---------------------------------------------------------
        Preconditions:
            value - a comparable part of a data element (?)
            update - an update function compatible with value (function)
        Postconditions:
            returns
            updated - True if value is in bst and is updated, False if
            value is not in bst, but adds value to bst in that case.
            (Recursive algorithm.)
        --------------------------------------------------------- -
        """
        # Find the node containing the key _data.
        updated = self._update_r_aux(self._root, key, update)
        return updated

    def _update_r_aux(self, node, key, update):
        """
        ---------------------------------------------------------
        Updates a _data in a bst.
        ---------------------------------------------------------
        Preconditions:
            node - a BST node
            key - a comparable part of a data element.
            update - an update function compatible with key.
        Postconditions:
            returns
          Attempts to key in node. Returns the full node _data if found,
          otherwise returns None. Updates the _data with update if found.
          (Recursive algorithm.)
        --------------------------------------------------------- -
        """
        # Find the node containing the key _data.
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        if node is not None:
            result = self._cf(key, node._data)

            if result == self.LESSER:
                value = self._update_r_aux(node._left, key, update)
            elif result == self.GREATER:
                value = self._update_r_aux(node._right, key, update)
            else:
                # Update the node data.
                update(node._data)
                value = deepcopy(node._data)
        else:
            value = None
        return value

    """
    ---------------------------------------------------------
    """

    def join(self, bst2):
        self._root = self._join_aux(self._root, bst2._root)
        return

    def _join_aux(self, node1, node2):
        return

    def apply(self):
        self._apply_aux(self._root)
        return

    def _apply_aux(self, node):
        if node is not None:
            self._apply_aux(node._left)
            node._data.apply()
            self._apply_aux(node._right)
        return

    def inorder(self):
        """
        -------------------------------------------------------
        Generates a list of the contents of the tree in inorder order.
        Use: bst.inorder()
        -------------------------------------------------------
        Postconditions:
            returns
            a - copy of the contents of the tree in inorder (list of ?)
        -------------------------------------------------------
        """
        a = []
        self._inorder_aux(self._root, a)
        return a

    def _inorder_aux(self, node, a):
        """
        ---------------------------------------------------------
        Traverses node subtree in inorder.
        Private recursive operation called only by inorder.
        Use: self._inorder_aux(node, a)
        ---------------------------------------------------------
        Preconditions:
            node - an BST node (_BSTNode)
            a - target list of data (list of ?)
        Postconditions:
            a contains the contents of node and its children in inorder.
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        ---------------------------------------------------------
        """
        if node is not None:
            self._inorder_aux(node._left, a)
            a.append(deepcopy(node._data))
            self._inorder_aux(node._right, a)
        return

    def preorder(self):
        """
        -------------------------------------------------------
        Generates a list of the contents of the tree in preorder order.
        Use: bst.preorder()
        -------------------------------------------------------
        Postconditions:
            returns
            a - copy of the contents of the tree in preorder (list of ?)
        -------------------------------------------------------
        """
        a = []
        self._preorder_aux(self._root, a)
        return a

    def _preorder_aux(self, node, a):
        """
        ---------------------------------------------------------
        Traverses node subtree in preorder.
        Private recursive operation called only by preorder.
        Use: self._preorder_aux(node, a)
        ---------------------------------------------------------
        Preconditions:
            node - an BST node (_BSTNode)
            a - target of data (list of ?)
        Postconditions:
            a contains the contents of node and its children in preorder.
        ---------------------------------------------------------
        """
        if node is not None:
            a.append(deepcopy(node._data))
            self._preorder_aux(node._left, a)
            self._preorder_aux(node._right, a)
        return

    def postorder(self):
        """
        -------------------------------------------------------
        Generates a list of the contents of the tree in postorder order.
        Use: bst.postorder()
        -------------------------------------------------------
        Postconditions:
            returns
            a - copy of the contents of the tree in postorder (list of ?)
        -------------------------------------------------------
        """
        a = []
        self._postorder_aux(self._root, a)
        return a

    def _postorder_aux(self, node, a):
        """
        ---------------------------------------------------------
        Traverses node subtree in postorder.
        Private recursive operation called only by postorder.
        Use: self._postorder_aux(node, a)
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        ---------------------------------------------------------
        Preconditions:
            node - an BST node (_BSTNode)
            a - target of data (list of ?)
        Postconditions:
            a contains the contents of node and its children in postorder.
        ---------------------------------------------------------
        """
        if node is not None:
            self._postorder_aux(node._left, a)
            self._postorder_aux(node._right, a)
            a.append(deepcopy(node._data))
        return

    def levelorder(self):
        """
        -------------------------------------------------------
        Copies the contents of the tree in levelorder order to a list.
        Use: values = bst.levelorder()
        -------------------------------------------------------
        Postconditions:
            returns
            values - a list containing the values of bst in levelorder.
            (list of ?)
        -------------------------------------------------------
        """
        values = []

        if self._root is not None:
            # Put the nodes for one level into a queue.
            queue = []
            queue.append(self._root)

            while len(queue) > 0:
                # Add a copy of the data to the sublist
                node = queue.pop(0)
                values.append(deepcopy(node._data))

                if node._left is not None:
                    queue.append(node._left)
                if node._right is not None:
                    queue.append(node._right)
        return values

    """
    ---------------------------------------------------------
    """

    def valid_avl(self):
        """
        ---------------------------------------------------------
        Determines if an AVL is valid.
        Use: b = avl.valid()
        ---------------------------------------------------------
        Postconditions:
            returns:
            is_valid - True if the tree is an AVL, False otherwise.
        ---------------------------------------------------------
        """
        is_valid = self.valid_avl_aux(self._root)
        return is_valid

    def valid_avl_aux(self, node):
        """
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        -------------------------------------------------------
        Helper function to determine the AVL validity of node.
        Private operation called only by is_valid.
        Use: b = self._is_valid_aux(node)
        -------------------------------------------------------
        Preconditions:
            node - the node to check the validity of (_AVLNode)
        Postconditions:
            returns
            is_valid - True if node is an AVL, False otherwise (boolean)
        -------------------------------------------------------
        """
        if node is None or (node._left is None and node._right is None):
            # Base case: node is empty or a leaf, so tree must be an AVL.
            is_valid = True
        elif abs(self._node_height(node._left) -
                 self._node_height(node._right)) > 1:
            # Base case: left or right subtree is too deep.
            # print("Height Violation at value: {}".format(node._data))
            is_valid = False
        elif (node._left is not None and node._left._data > node._data) \
                or (node._right is not None and node._right._data < node._data):
            # Base case: does not follow the BST property.
            # print("Binary Tree Violation")
            is_valid = False
        else:
            # General case: check the nodes children for validity.
            is_valid = self.valid_avl_aux(node._left) and \
                self.valid_avl_aux(node._right)
        return is_valid

    def preorder_i(self):
        """
        ------------------------------------------------------
        Generates a list of the contents of the tree in preorder order.
        (Iterative algorithm)
        Use: bst.preorder_i()
        ------------------------------------------------------
        Postconditions:
            returns
            a - copy of the contents of the tree in preorder (list of ?)
        ------------------------------------------------------
        """
        a = []
        stack = Stack()
        stack.push(self._root)

        while not stack.empty():
            node = stack.pop()

            if node is not None:
                a.append(deepcopy(node._data))
                stack.push(node._right)
                stack.push(node._left)
        return a

    def count(self):
        """
        -------------------------------------------------------
        Returns the number of nodes in a BST.
        Use: number = bst.count()
        ------------------------------------------------------
        Postconditions:
            returns
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
            number - count of nodes in tree (int)
        ----------------------------------------------------------
        """
        number = self._count_aux(self._root)
        return number

    def _count_aux(self, node):
        """
        ---------------------------------------------------------
        Returns the number of nodes in a BST subtree.
        -------------------------------------------------------
        Preconditions:
            node - a BST node (_BSTNode)
        Postconditions:
            returns
            number - count of nodes in the current subtree (int)
        ----------------------------------------------------------
        """
        if node is None:
            # Base case: node does not exist
            number = 0
        else:
            # General case: node exists.
            number = 1 + self._count_aux(node._left) + \
                self._count_aux(node._right)
        return number

    def count_apply(self, func):
        """
        ---------------------------------------------------------
        Returns the number of values in a BST where func(value) is True.
        Use: number = bst.count_apply(func)
        -------------------------------------------------------
        Preconditions:
            func - a function that given a value in the bst returns
                True for some condition, otherwise returns False.
        Postconditions:
            returns
            number - count of nodes in tree where func(value) is True (int)
        ----------------------------------------------------------
        """
        number = self._count_apply_aux(func, self._root)
        return number

    def _count_apply_aux(self, func, node):
        """
        ---------------------------------------------------------
        Returns the number of nodes in a BST subtree
        where the result of calling func(value) is True.
        Use: number = self.__count_apply_aux(func, node)
        -------------------------------------------------------
        Preconditions:
            node - a BST node (_BSTNode)
        Postconditions:
            returns
            number - count of nodes in the current subtree (int)
        ----------------------------------------------------------
        """
        if node is None:
            # Base case: node does not exist
            number = 0
        else:
            # General case: node exists.
            if func(node._data):
```

OneClass find more resources at oneclass.com

```
                number = 1
            else:
                number = 0
            number = number + self._count_apply_aux(func, node._left) + \
                self._count_apply_aux(func, node._right)
        return number

    def __iter__(self):
        """
        -------------------------------------------------------
        Generates a Python iterator. Iterates through a BST node
        in level order.
        Use: for v in bst:
        -------------------------------------------------------
        Postconditions:
            yields
            value - the values in the BST node and its children (?)
        -------------------------------------------------------
        """
        if self._root is not None:
            # Put the nodes for one level into a queue.
            queue = []
            queue.append(self._root)

            while len(queue) > 0:
                # Add a copy of the data to the sublist
                node = queue.pop(0)
                yield node._data

                if node._left is not None:
                    queue.append(node._left)
                if node._right is not None:
                    queue.append(node._right)
```

```
"""
-------------------------------------------------------
list_linked.py
Linked version of the list ADT.
-------------------------------------------------------
Author:  David Brown
ID:      999999999
Email:   dbrown@wlu.ca
__updated__ = "2018-03-21"
-------------------------------------------------------
"""
from copy import deepcopy


class _ListNode:

    def __init__(self, value, next_):
        """
        -------------------------------------------------------
        Initializes a list node.
        Use: node = _ListNode(value, _next)
        -------------------------------------------------------
        Preconditions:
            _data - data value for node (?)
            _next - another list node (_ListNode)
        Postconditions:
            Initializes a list node that contains a copy of value
            and a link to the next node in the list.
        -------------------------------------------------------
        """
        self._data = deepcopy(value)
        self._next = next_
        return


class List:

    def __init__(self):
        """
        -------------------------------------------------------
        Initializes an empty list.
        Use: l = List()
        -------------------------------------------------------
        Postconditions:
            Initializes an empty list.
        -------------------------------------------------------
        """
        self._front = None
        self._count = 0
        return

    def empty(self):
        """
        -------------------------------------------------------
        Determines if the list is empty.
        Use: b = l.empty()
        -------------------------------------------------------
        Postconditions:
            returns
            True if the list is empty, False otherwise.
        -------------------------------------------------------
        """
        return self._front is None
```

OneClass find more resources at oneclass.com

```
    def is_empty(self):
        """
        -------------------------------------------------------
        Determines if the list is empty.
        Use: b = l.is_empty()
        -------------------------------------------------------
        Postconditions:
            returns
            True if the list is empty, False otherwise.
        -------------------------------------------------------
        """
        return self._front is None

    def __len__(self):
        """
        -------------------------------------------------------
        Returns the size of the list.
        Use: n = len(l)
        -------------------------------------------------------
        Postconditions:
            returns
            the number of values in the list.
        -------------------------------------------------------
        """
        return self._count

    def insert(self, i, value):
        """
        -------------------------------------------------------
        Inserts a copy of value into the list at index i.
        Use: l.insert(i, value)
        -------------------------------------------------------
        Preconditions:
            i - index value (int)
            value - a data element (?)
        Postconditions:
            a copy of value is added to index i, all other values are pushed
right
            If i outside of range of length of list, appended to end
        -------------------------------------------------------
        """
        if i < 0:
            # negative index
            i = self._count + i

        n = 0
        previous = None
        current = self._front

        while n < i and current is not None:
            # find the proper location in the list
            previous = current
            current = current._next
            n += 1

        if previous is None:
            # Insert a new node into the front of the list.
            self._front = _ListNode(value, self._front)
        else:
            # Insert a new node elsewhere in the list
            previous._next = _ListNode(value, current)
        self._count += 1
        return
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
    def insert_front(self, value):
        """
        -------------------------------------------------------
        Inserts a copy of value into the front of the list.
        Use: l.insert_front(value)
        -------------------------------------------------------
        Preconditions:
            value - a data element. (?)
        Postconditions:
            value is added to the front of the list.
        -------------------------------------------------------
        """
        self.insert(0, value)
        return

    def _linear_search(self, key):
        """
        -------------------------------------------------------
        Searches for the first occurrence of key in the list.
        Private helper methods - used only by other ADT methods.
        Use: p, c, i = self._linear_search(key)
        -------------------------------------------------------
        Preconditions:
            key - a partial data element (?)
        Postconditions:
            returns
            previous - pointer to the node previous to the node containing key
(_ListNode)
            current - pointer to the node containing key (_ListNode)
            index - index of the node containing key, -1 if key not found (int)
        -------------------------------------------------------
        """
        previous = None
        current = self._front
        index = 0

        while current is not None and current._data != key:
            previous = current
            current = current._next
            index += 1

        if current is None:
            index = -1

        return previous, current, index

    def remove(self, key):
        """
        -------------------------------------------------------
        Finds, removes, and returns the first value in list that matches key.
        Use: value = l.remove(key)
        -------------------------------------------------------
        Preconditions:
            key - a partial data element (?)
        Postconditions:
            returns
            value - the full value matching key, otherwise None (?)
        -------------------------------------------------------
        """
        # search list for key.
        previous, current, _ = self._linear_search(key)

        if current is None:
            # Key is not found.
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
            value = None
        else:
            value = current._data
            self._count -= 1

            if previous is None:
                # Remove the first element.
                self._front = current._next
            else:
                # Remove any other element.
                previous._next = current._next
        return value

    def remove_front(self):
        """
        -------------------------------------------------------
        Removes the first node in the list.
        Use: value = l.remove_front()
        -------------------------------------------------------
        Postconditions:
            returns
            value - the first value in the list (?)
        -------------------------------------------------------
        """
        assert self._front is not None, "Cannot remove from an empty list"

        value = self._front._data
        self._front = self._front._next
        self._count -= 1
        return value

    def remove_many(self, key):
        """
        -------------------------------------------------------
        Finds and removes all values in the list that match key.
        Use: l.remove_many(key)
        -------------------------------------------------------
        Preconditions:
            key - a data element (?)
        Postconditions:
            Removes all values matching key.
        -------------------------------------------------------
        """
        previous = None
        current = self._front

        while current is not None:

            if current._data == key:
                # Do not update previous
                self._count -= 1

                if previous is None:
                    # Remove the first element.
                    self._front = current._next
                else:
                    # Remove any other element.
                    previous._next = current._next
            else:
                previous = current
            current = current._next
        return

    def find(self, key):
```

OneClass find more resources at oneclass.com

```
        """
        -------------------------------------------------------
        Finds and returns a copy of the first value in list that matches key.
        Use: value = l.find(key)
        -------------------------------------------------------
        Preconditions:
            key - a partial data element (?)
        Postconditions:
            returns
            value - a copy of the full value matching key, otherwise None (?)
        -------------------------------------------------------
        """
        _, current, _ = self._linear_search(key)

        if current is not None:
            value = deepcopy(current._data)
        else:
            value = None
        return value

    def peek(self):
        """
        -------------------------------------------------------
        Returns a copy of the first value in list.
        Use: value = l.peek()
        -------------------------------------------------------
        Postconditions:
            returns
            value - a copy of the first value in the list (?)
        -------------------------------------------------------
        """
        assert self._front is not None, "Cannot peek at an empty list"

        value = deepcopy(self._front._data)
        return value

    def index(self, key):
        """
        -------------------------------------------------------
        Finds location of a value by key in list.
        Use: n = l.index(key)
        -------------------------------------------------------
        Preconditions:
            key - a partial data element (?)
        Postconditions:
            returns
            i - the index of the location of key in the list, -1 if
                key is not in the list.
        -------------------------------------------------------
        """
        _, _, i = self._linear_search(key)
        return i

    def _valid_index(self, i):
        """
        -------------------------------------------------------
        Private helper method to validate an index value.
        Python index values can be positive or negative and range from
          -len(list) to len(list) - 1
        Use: assert self._valid_index(i)
        -------------------------------------------------------
        Preconditions:
            i - an index value (int)
        Postconditions:
```

OneClass find more resources at oneclass.com

```
            returns
            True if i is a valid index, False otherwise.
        -------------------------------------------------------
        """
        n = self._count
        return -n <= i < n

    def __getitem__(self, i):
        """
        ---------------------------------------------------------
        Returns a copy of the nth element of the list.
        Use: value = l[i]
        -------------------------------------------------------
        Preconditions:
            i - index of the element to access (int)
        Postconditions:
            returns
            value - the i-th element of list (?)
        -------------------------------------------------------
        """
        assert self._valid_index(i), "Invalid index value"

        current = self._front

        if i < 0:
            # negative index - convert to positive
            i = self._count + i
        j = 0

        while j < i:
            current = current._next
            j += 1

        value = deepcopy(current._data)
        return value

    def __setitem__(self, i, value):
        """
        ---------------------------------------------------------
        Places a copy of value into the list at position n.
        Use: l[i] = value
        -------------------------------------------------------
        Preconditions:
            i - index of the element to access (int)
            value - a data value (?)
        Postconditions:
            The i-th element of list contains a copy of value. The
                existing value at i is overwritten.
        -------------------------------------------------------
        """
        assert self._valid_index(i), "Invalid index value"

        current = self._front

        if i < 0:
            # negative index - convert to positive
            i = self._count + i
        j = 0

        while j < i:
            current = current._next
            j += 1

        current._data = deepcopy(value)
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        return

    def __contains__(self, key):
        """
        -------------------------------------------------------
        Determines if the list contains key.
        Use: b = key in l
        -------------------------------------------------------
        Preconditions:
            key - a partial data element (?)
        Postconditions:
            returns
            True if the list contains key, False otherwise.
        -------------------------------------------------------
        """
        _, current, _ = self._linear_search(key)
        return current is not None

    def max(self):
        """
        -------------------------------------------------------
        Finds the maximum value in list.
        Use: value = l.max()
        -------------------------------------------------------
        Postconditions:
            returns
            max_data - a copy of the maximum value in the list (?)
        -------------------------------------------------------
        """
        assert self._front is not None, "Cannot find maximum of an empty list"

        max_data = self._front._data
        current = self._front._next

        while current is not None:
            if max_data < current._data:
                max_data = current._data
            current = current._next
        max_data = deepcopy(max_data)
        return max_data

    def min(self):
        """
        -------------------------------------------------------
        Finds the minimum value in list.
        Use: value = l.min()
        -------------------------------------------------------
        Postconditions:
            returns
            min_data - a copy of the minimum value in the list (?)
        -------------------------------------------------------
        """
        assert self._front is not None, "Cannot find maximum of an empty list"

        min_data = self._front._data
        current = self._front._next

        while current is not None:
            if min_data > current._data:
                min_data = current._data
            current = current._next
        min_data = deepcopy(min_data)
        return min_data
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
    def count(self, key):
        """
        -------------------------------------------------------
        Finds the number of times key appears in list.
        Use: n = l.count(key)
        -------------------------------------------------------
        Preconditions:
            key - a partial data element (?)
        Postconditions:
            returns
            number - number of times key appears in list (int)
        -------------------------------------------------------
        """
        number = 0
        current = self._front

        while current is not None:
            if key == current._data:
                number += 1
            current = current._next
        return number

    def reverse(self):
        """
        -------------------------------------------------------
        Reverses the order of the elements in list.
        Use: l.reverse()
        -------------------------------------------------------
        Postconditions:
            The contents of list are reversed in order with respect
            to their order before the method was called.
        -------------------------------------------------------
        """
        new_front = None

        while self._front is not None:
            temp = self._front._next
            self._front._next = new_front
            new_front = self._front
            self._front = temp

        self._front = new_front
        return

    def reverse_r(self):
        """
        -------------------------------------------------------
        Reverses the order of the elements in list.
        Use: l.reverse_r()
        -------------------------------------------------------
        Postconditions:
            The contents of list are reversed in order with respect
            to their order before the method was called.
        -------------------------------------------------------
        """
        self._reverse_r_aux(None)
        return

    def _reverse_r_aux(self, new_front):
        """
        -------------------------------------------------------
        Auxiliary recursive function for reverse_r (fruitful).
        Use: l.reverse_r_aux(new_front)
        -------------------------------------------------------
```

OneClass find more resources at oneclass.com

```
        Preconditions:
            new_front - the new front node to process (_ListNode)
        Postconditions:
            The list is in reverse order.
        -------------------------------------------------------
        """
        if self._front is None:
            self._front = new_front
        else:
            temp = self._front._next
            self._front._next = new_front
            new_front = self._front
            self._front = temp
            self._reverse_r_aux(new_front)
        return

    def reverse_r2(self):
        """
        -------------------------------------------------------
        Reverses the order of the elements in list.
        Use: l.reverse_r()
        -------------------------------------------------------
        Postconditions:
            The contents of list are reversed in order with respect
            to their order before the method was called.
        -------------------------------------------------------
        """
        self._front = self._reverse_r2_aux(self._front, None)
        return

    def _reverse_r2_aux(self, current, previous):
        """
        -------------------------------------------------------
        Auxiliary recursive function for reverse_r (fruitful).
        -------------------------------------------------------
        Preconditions:
            current - the current node to process (_ListNode)
            previous - the node previous to current (_ListNode)
        Postconditions:
            returns
            new_front - the new front of the list so far (_ListNode)
        -------------------------------------------------------
        """
        if current is None:
            new_front = previous
        else:
            new_front = self._reverse_r2_aux(current._next, current)
            current._next = previous
        return new_front

    def append(self, value):
        """
        ---------------------------------------------------------
        Appends a copy of value to the end of the List.
        Use: l.append(value)
        -------------------------------------------------------
        Preconditions:
            value - a data element (?)
        Postconditions:
            a copy of value is added to the end of the List.
        -------------------------------------------------------
        """
        previous = None
        current = self._front
```

OneClass
find more resources at oneclass.com

```
        while current is not None:
            # Find the last node in the list
            previous = current
            current = current._next

        if previous is None:
            # Insert into the front of the list.
            self._front = _ListNode(deepcopy(value), self._front)
        else:
            # Insert elsewhere in the list
            previous._next = _ListNode(deepcopy(value), current)
        self._count += 1
        return

    def clean(self):
        """
        ---------------------------------------------------------
        Removes duplicates from the list. (iterative algorithm)
        Use: l.clean()
        -------------------------------------------------------
        Postconditions:
            The list contains one and only one of each value formerly present
            in the list. The first occurrence of each value is preserved.
        -------------------------------------------------------
        """
        key_node = self._front

        while key_node is not None:
            # Loop through every node - compare each node with the rest
            previous = key_node
            current = key_node._next

            while current is not None:
                # Always search to the end of the list (may have > 1 duplicate)
                if current._data == key_node._data:
                    # Remove the current node by connecting the node before it
                    # to the node after it.
                    self._count -= 1
                    previous._next = current._next
                else:
                    previous = current
                # Move to the _next node.
                current = current._next
            # Check for duplicates of the _next remaining node in the list
            key_node = key_node._next
        return

    def pop(self, *i):
        """
        -------------------------------------------------------
        Finds, removes, and returns the value in list whose index matches i.
        Use: value = l.pop(i)
        -------------------------------------------------------
        Preconditions:
            i - an array of arguments (?)
                i[0], if it exists, is the index
        Postconditions:
            returns
            value - if i exists, the value at position i, otherwise the last
                value in the list, value is removed from the list (?)
        -------------------------------------------------------
        """
        assert self._front is not None, "Cannot pop from an empty list"
```

OneClass
find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        assert len(i) <= 1, "No more than 1 argument allowed"

        previous = None
        current = self._front

        if len(i) == 1:

            if i[0] < 0:
                # index is negative
                i[0] = self._count + i[0]
            j = 0

            while j < i[0]:
                previous = current
                current = current._next
                j += 1
        else:
            # find and pop the last element
            j = 0

            while j < (self._count - 1):
                previous = current
                current = current._next
                j += 1

        value = current._data

        if previous is None:
            # Update the front
            self._front = current._next
        else:
            # Update any other node
            previous._next = current._next
        self._count -= 1
        return value

    def _swap(self, pln, prn):
        """
        -------------------------------------------------------
        Swaps the position of two nodes.
        Use: self._swap(pln, prn)
        -------------------------------------------------------
        Preconditions:
            pln - node before list node to swap (_ListNode)
            prn - node before list node to swap (_ListNode)
        Postconditions:
            The nodes in pln.next and prn.next have been swapped,
                and all links to them updated.
        -------------------------------------------------------
        """
        if pln is not prn:
            # Swap only if two nodes are not the same node

            if pln is None:
                # Make r the new front
                l = self._front
                self._front = prn._next
            else:
                l = pln._next
                pln._next = prn._next

            if prn is None:
                # Make l the new front
                r = self._front
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
                self._front = l
            else:
                r = prn._next
                prn._next = l

            # Swap next pointers
            # l._next, r._next = r._next, l._next
            temp = l._next
            l._next = r._next
            r._next = temp
        return

    def identical(self, rs):
        """
        ---------------------------------------------------------
        Determines whether two lists are identical. (iterative version)
        Use: b = l.identical(rs)
        -------------------------------------------------------
        Preconditions:
            rs - another list (List)
        Postconditions:
            returns
            is_identical - True if this list contains the same values as
                rs in the same order, otherwise False.
        -------------------------------------------------------
        """
        if self._count != rs._count:
            is_identical = False
        else:
            current1 = self._front
            current2 = rs._front

            while current1 is not None and current1._data == current2._data:
                current1 = current1._next
                current2 = current2._next

            is_identical = current1 is None
        return is_identical

    def identical_r(self, rs):
        """
        ---------------------------------------------------------
        Determines whether two lists are identical. (recursive version)
        Use: b = l.identical_r(rs)
        -------------------------------------------------------
        Preconditions:
            rs - another list (List)
        Postconditions:
            returns
            is_identical - True if this list contains the same values as rs
                in the same order, otherwise False.
        -------------------------------------------------------
        """
        if self._count != rs._count:
            is_identical = False
        else:
            is_identical = self._identical_r_aux(self._front, rs._front)
        return is_identical

    def _identical_r_aux(self, node1, node2):
        """
        -------------------------------------------------------
        An auxiliary function for identical_r
        -------------------------------------------------------
```

OneClass find more resources at oneclass.com

```
        Preconditions:
            node1 - a List node (_ListNode)
            node2 - a List node (_ListNode)
        Postconditions:
            returns
            identical - True if node1 contains the same values as node2
                in the same order, otherwise False.
        -------------------------------------------------------
        """
        if node1 is None:
            is_identical = True
        elif node1._data != node2._data:
            is_identical = False
        else:
            is_identical = self._identical_r_aux(node1._next,
                                                 node2._next)
        return is_identical

    def split_alt(self):
        """
        -------------------------------------------------------
        Split a List into two parts. even contains the even indexed
        elements, odd contains the odd indexed elements.
        Order of even and odd is not significant. (iterative version)
        Use: even, odd = l.split_alt()
        -------------------------------------------------------
        Postconditions:
            returns
            even - the even indexed elements of the list (List)
            odd - the odd indexed elements of the list (List)
                The List is empty.
        -------------------------------------------------------
        """
        even = List()
        odd = List()
        i = 0

        while self._front is not None:

            if i % 2 == 0:
                even._move_front(self)
            else:
                odd._move_front(self)
            i += 1

        even.reverse()
        odd.reverse()
        return even, odd

    def split_alt_r(self):
        """
        -------------------------------------------------------
        Split a list into two parts. even contains the even indexed
        elements, odd contains the odd indexed elements.
        Order of even and odd is not significant. (recursive version)
        Use: even, odd = l.split_alt()
        -------------------------------------------------------
        Postconditions:
            returns
            even - the even numbered elements of the list (List)
            odd - the odd numbered elements of the list (List)
                The List is empty.
        -------------------------------------------------------
        """
```

OneClass find more resources at oneclass.com

```
        even = List()
        odd = List()
        self._split_alt_r_aux(even, odd)
        return even, odd

    def _split_alt_r_aux(self, even, odd):
        """
        -------------------------------------------------------
        Split a list into two parts. even contains the even indexed
        elements, odd contains the odd numbered elements.
        Order of even and odd is not significant.
        -------------------------------------------------------
        Postconditions:
            returns
            even - the even numbered elements of the list (List)
            odd - the odd numbered elements of the list (List)
                The List is empty.
        -------------------------------------------------------
        """
        if self._front is not None:
            even._move_front(self)
            # Reverse the order of the arguments.
            self._split_alt_r_aux(odd, even)
        return

    def _linear_search_r(self, key):
        """
        -------------------------------------------------------
        Searches for the first occurrence of key in the list.
        Private helper methods - used only by other ADT methods.
        (recursive version)
        Use: p, c, i = self._linear_search(key)
        -------------------------------------------------------
        Preconditions:
            key - a partial data element (?)
        Postconditions:
            returns
            previous - pointer to the node previous to the node containing key
(_ListNode)
            current - pointer to the node containing key (_ListNode)
            index - index of the node containing key, -1 if key not found (int)
        -------------------------------------------------------
        """
        previous, current, index = self._linear_search_r_aux(
            key, None, self._front, 0)
        return previous, current, index

    def _linear_search_r_aux(self, key, previous, current, index):
        """
        -------------------------------------------------------
        Auxiliary method for _linear_search.
        Use: p, c, i = self._linear_search(key, previous, current, index)
        -------------------------------------------------------
        Preconditions:
            key - a partial data element (?)
            previous - pointer to the node previous to the node containing key
(_ListNode)
            current - pointer to the node containing key (_ListNode)
            index - index of the node containing key, -1 if key not found (int)
        Postconditions:
            returns
            previous - pointer to the node previous to the node containing key
(_ListNode)
            current - pointer to the node containing key (_ListNode)
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
            index - index of the node containing key, -1 if key not found (int)
        -------------------------------------------------------
        """
        # Note: can be done with three parameters, if these cases added:
        #         if previous is None:
        #             current = self._front
        #         else:
        #             current = previous._next

        if current is None:
            index = -1
        elif current._data != key:
            previous, current, index = self._linear_search_r_aux(
                key, current, current._next, index + 1)

        return previous, current, index

    def intersection(self, rs):
        """
        -------------------------------------------------------
        Returns a list that contains only values that appear in both
        the current List and rs. (iterative version)
        Use: l2 = l1.intersection(rs)
        -------------------------------------------------------
        Preconditions:
            rs - another List (List)
        Postconditions:
            returns
            new_list - A List that contains only the values found in both
            the current List and rs. Values do not repeat. (List)
        -------------------------------------------------------
        """
        new_list = List()
        temp = rs._front

        while temp is not None:
            _, current, _ = self._linear_search(temp._data)

            if current is not None:
                # Value exists in both lists.
                _, current, _ = new_list._linear_search(temp._data)

                if current is None:
                    # Value does not appear in new list.
                    new_list._front = _ListNode(temp._data, new_list._front)
                    new_list._count += 1

            temp = temp._next
        return new_list

    def intersection_r(self, rs):
        """
        -------------------------------------------------------
        Returns a list that contains only values that appear in both
        the current List and rs. (recursive version)
        Use: l2 = l1.intersection_r(rs)
        -------------------------------------------------------
        Preconditions:
            rs - another List (List)
        Postconditions:
            returns
            new_list - A List that contains only the values found in both
            the current List and rs. Values do not repeat. (List)
        -------------------------------------------------------
```

OneClass find more resources at oneclass.com

```
        """
        new_list = List()
        self._intersection_r_aux(rs._front, new_list)
        new_list.reverse()
        return new_list

    def _intersection_r_aux(self, rs_node, new_list):
        """
        -------------------------------------------------------
        Auxiliary function for intersection_r.
        Use: self._intersection_r_aux(rs_node, new_list)
        -------------------------------------------------------
        Preconditions:
            rs_node - right side list node (_ListNode)
            new_list - the list to contain the intersection (List)
        Postconditions:
            new_list contains one copy of values common to current list
                and rs
        -------------------------------------------------------
        """
        if rs_node is not None:
            _, current, _ = self._linear_search(rs_node._data)

            if current is not None:
                # Value exists in both lists.
                _, current, _ = new_list._linear_search(rs_node._data)

                if current is None:
                    # Value does not appear in new list.
                    new_list.insert(0, rs_node._data)
            self._intersection_r_aux(rs_node._next, new_list)
        return

    def union(self, rs):
        """
        -------------------------------------------------------
        Returns a list that contains all values in both
        the current List and rs. (iterative algorithm)
        Use: nl = l.union(rs)
        -------------------------------------------------------
        Preconditions:
            rs - another list (List)
        Postconditions:
            returns
            new_list - contains all values found in both the current
            List and rs. Values do not repeat. (List)
        -------------------------------------------------------
        """
        new_list = List()
        temp = self._front

        while temp is not None:
            _, current, _ = new_list._linear_search(temp._data)

            if current is None:
                # Value does not exist in new list.
                new_list.insert(0, temp._data)
            temp = temp._next

        temp = rs._front

        while temp is not None:
            _, current, _ = new_list._linear_search(temp._data)
```

OneClass find more resources at oneclass.com

```
            if current is None:
                # Value does not exist in new list.
                new_list.insert(0, temp._data)

            temp = temp._next
        new_list.reverse()
        return new_list

    def union_r(self, rs):
        """
        -------------------------------------------------------
        Returns a list that contains all values in both
        the current List and rs. (recursive algorithm)
        Use: nl = l.union(rs)
        -------------------------------------------------------
        Preconditions:
            rs - another list (List)
        Postconditions:
            returns
            new_list - contains all values found in both the current
            List and rs. Values do not repeat. (List)
        -------------------------------------------------------
        """
        new_list = List()
        new_list._union_r_aux(self._front)
        new_list._union_r_aux(rs._front)
        new_list.reverse()
        return new_list

    def _union_r_aux(self, rs_node):
        """
        -------------------------------------------------------
        Auxiliary function for union_r.
        Use: self._union_r_aux(rs_node)
        -------------------------------------------------------
        Preconditions:
            rs_node - right side list rs_node (_ListNode)
        Postconditions:
            self contains one copy of all values in rs_node
        -------------------------------------------------------
        """
        if rs_node is not None:
            _, current, _ = self._linear_search(rs_node._data)

            if current is None:
                # Value does not exist in current list.
                self.insert(0, rs_node._data)
            self._union_r_aux(rs_node._next)
        return

    def clean_r(self):
        """
        ---------------------------------------------------------
        Removes duplicates from the list. (recursive algorithm)
        Use: l.clean_r()
        -------------------------------------------------------
        Postconditions:
            The list contains one and only one of each value formerly present
            in the list. The first occurrence of each value is preserved.
        -------------------------------------------------------
        """
        self._clean_r_aux(self._front)
        return
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
    def _clean_r_aux(self, key_node):
        """
        -------------------------------------------------------
        Auxiliary function to remove duplicates from a list.
        Use: self._clean_r_aux(key_node)
        -------------------------------------------------------
        Preconditions:
            key_node - node containing the current key value to be searched
                for (_ListNode)
        Postconditions:
            All nodes containing the same value as key_node are removed
                from the rest of the list.
        -------------------------------------------------------
        """
        # Compares key_node against the rest of the list.
        if key_node is not None:
            # Update the _next node in the list.
            self._clean_r_aux_2(key_node._data, key_node)
            # Move to the _next key node in the list for comparison.
            self._clean_r_aux(key_node._next)
        return

    def _clean_r_aux_2(self, data, previous):
        """
        -------------------------------------------------------
        Auxiliary function to remove duplicates from a list.
        Use: self._clean_r_aux_2(data, node)
        -------------------------------------------------------
        Preconditions:
            data - the key value to be searched for (?)
            previous - the node pointing to the rest of the list to
                be searched for data (_ListNode)
        Postconditions:
            All nodes containing the same value as data are removed from
            the rest of the list.
        -------------------------------------------------------
        """
        # Removes all duplicates to the key node from the rest of the list.
        if previous._next is not None:
            current = previous._next

            if current._data == data:
                # Remove the current node.
                self._count -= 1
                previous._next = current._next
                self._clean_r_aux_2(data, previous)
            else:
                # Move to the next node.
                self._clean_r_aux_2(data, previous._next)
        return

    def split_th(self):
        """
        -------------------------------------------------------
        Splits list into two parts. ls contains the first half,
        rs the second half. Uses Tortoise / Hare algorithm.
        Current list is empty.
        Use: ls, rs = l.split_th()
        -------------------------------------------------------
        Postconditions:
            returns
            ls - a new List with >= 50% of the original List (List)
            rs - a new List with <= 50% of the original List (List)
        -------------------------------------------------------
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        """
        ls = List()
        rs = List()
        # Initialize both temporary pointers to beginning of the list.
        hare = self._front
        tortoise = self._front

        while hare is not None and hare._next is not None:
            # Move hare down two nodes.
            hare = hare._next._next

            if hare is not None:
                # Update tortoise only if hare is not None.
                tortoise = tortoise._next

        # Split the list.
        if tortoise is not None:
            ls._front = self._front
            rs._front = tortoise._next
            tortoise._next = None
            rs._count = self._count // 2
            ls._count = self._count - rs._count
            self._count = 0
        return ls, rs

    def split(self):
        """
        -------------------------------------------------------
        Splits list into two parts. ls contains the first half,
        rs the second half. Uses counting algorithm.
        Current list is empty.
        Use: ls, rs = l.split_th()
        -------------------------------------------------------
        Postconditions:
            returns
            ls - a new List with >= 50% of the original List (List)
            rs - a new List with <= 50% of the original List (List)
        -------------------------------------------------------
        """
        ls = List()
        rs = List()
        # Find the middle of the list.
        middle = self._count // 2 + self._count % 2 - 1
        current = self._front
        i = 0

        while i < middle:
            current = current._next
            i += 1

        if current is not None:
            ls._front = self._front
            rs._front = current._next
            current._next = None
            rs._count = self._count // 2
            ls._count = self._count - rs._count
            self._count = 0
        return ls, rs

    def split_apply(self, func):
        """
        -------------------------------------------------------
        Splits list into two parts. ls contains all the values
        where the result of calling func(value) is True,
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        rs contains the remaining values.
        Use: ls, rs = l.split_apply(func)
        -------------------------------------------------------
        Postconditions:
            returns
            ls - a new List with values where func(value) is True (List)
            rs - a new List with values where func(value) is False (List)
            self is empty. Order of values is new lists is maintained.
        -------------------------------------------------------
        """
        ls = List()
        rs = List()
        # Split
        while self._front is not None:

            if func(self._front._data):
                ls._move_front(self)
            else:
                rs._move_front(self)
        ls.reverse()
        rs.reverse()
        return ls, rs

    def copy(self):
        """
        -------------------------------------------------------
        Duplicates the current list to a new list in the same order.
        (iterative version)
        Use: new_list = l.copy()
        -------------------------------------------------------
        Postconditions:
            returns
            new_list - a copy of self (List)
        -------------------------------------------------------
        """
        new_list = List()

        if self._front is not None:
            # Set up the new list front.
            new_list._front = _ListNode(self._front._data, None)
            previous = new_list._front
            current = self._front._next

            while current is not None:
                # Add a node in the new list.
                new_node = _ListNode(current._data, None)
                previous._next = new_node
                previous = new_node
                # Move to the next node in the current list.
                current = current._next
            new_list._count = self._count
        return new_list

    def copy_r(self):
        """
        -----------------------------------------------------------
        Duplicates the current list to a new list in the same order.
        (recursive verstion)
        Use: new_list = l.copy()
        -----------------------------------------------------------
        Postconditions:
            returns
            new_list - a copy of self (List)
        -----------------------------------------------------------
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        """
        new_list = List()
        new_list._front = self._copy_r_aux(self._front)
        new_list._count = self._count
        return new_list

    def _copy_r_aux(self, current):
        """
        -------------------------------------------------------------
        Auxiliary function for copy_r. Private helper function
        Use: new_node = self._copy_r_aux(current)
        -------------------------------------------------------------
        Preconditions:
            current - current list node to be copied (_ListNode)
        Postconditions:
            returns
            new_node - new node of new list (_ListNode)
        -------------------------------------------------------------
        """
        if current is None:
            new_node = None
        else:
            next_node = self._copy_r_aux(current._next)
            new_node = _ListNode(current._data, next_node)
        return new_node

    def reverse_pc(self):
        """
        -------------------------------------------------------
        Reverses a list through partitioning and concatenation.
        Use: l.reverse_pc()
        -------------------------------------------------------
        Postconditions:
            The contents of the current list are reversed.
        -------------------------------------------------------
        """
        self._front = self._reverse_pc_aux(self._front)

    def _reverse_pc_aux(self, current):
        """
        -------------------------------------------------------
        Auxiliary function for reverse_pc.
        Use: node = self._reverse_pc_aux(node)
        -------------------------------------------------------
        Preconditions:
            current - the current node to process (_ListNode)
        Postconditions:
            returns
            result -
          Recursively split and concatenate the list until the end
          of the list is reached.
        -------------------------------------------------------
        """
        if current is None:
            result = None
        else:
            head, tail = self._partition(current)
            tail = self._reverse_pc_aux(tail)
            result = self._concantenate(tail, head)
        return result

    def _partition(self, current):
        """
        -------------------------------------------------------
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        Partitions a list at the current node.
        -------------------------------------------------------
        Parameters:
          current - the current node to process
        Preconditions:
          current - a valid node dictionary
        Postconditions:
          Partition a list by removing the first node as 'head' and
          naming the remainder of the list 'tail'. 'node' cannot be None.
        -------------------------------------------------------
        """
        tail = current._next
        head = current
        head._next = None
        return head, tail

    def _concantenate(self, tail, head):
        """
        -------------------------------------------------------
        [function description]
        -------------------------------------------------------
        Parameters:
          head - the head node of a list
          tail - the tail node of a list
        Preconditions:
          head - a valid node dictionary
          tail - a valid node dictionary
        Postconditions:
         Appends the 'head' node to the end of 'tail'.
        -------------------------------------------------------
        """
        previous = None
        current = tail

        # Find the end of 'tail'.
        while current is not None:
            previous = current
            current = current._next

        # Append 'head' to the end of 'tail'.
        if previous is None:
            tail = head
        else:
            previous._next = head
        # 'tail' is the new head of the list.
        return tail

    def _move_front(self, rs):
        """
        -------------------------------------------------------
        Moves the front node from the rs List to the front
        of the current List. Private helper method.
        Use: self._move_front(rs)
        -------------------------------------------------------
        Preconditions:
            rs - a non-empty linked List (List)
        Postconditions:
            The current List contains the old front of the rs List and
            its count is updated. The rs List front and count are updated.
        -------------------------------------------------------
        """
        assert rs._front is not None, \
            "Cannot move the front of an empty List"
```

OneClass find more resources at oneclass.com

```
        # Move the front node from rs to self
        temp = rs._front
        rs._front = rs._front._next
        temp._next = self._front
        self._front = temp
        # Update the list counts
        self._count += 1
        rs._count -= 1
        return

    def combine(self, s2):
        """
        -------------------------------------------------------
        Combines contents of two lists into a third.
        Use: new_list = l1.combine(s2)
        -------------------------------------------------------
        Preconditions:
            s2- an linked-based List (List)
        Postconditions:
            returns
            new_list - the contents of the current List and s2
            are interlaced into new_list - current List and s2
            are empty (List)
        -------------------------------------------------------
        """
        new_list = List()

        while self._front is not None and s2._front is not None:
            new_list._move_front(self)
            new_list._move_front(s2)

        while self._front is not None:
            new_list._move_front(self)

        while s2._front is not None:
            new_list._move_front(s2)

        new_list.reverse()
        return new_list

    def combine_r(self, rs):
        """
        -------------------------------------------------------
        Combines contents of two lists into a third.
        Use: new_list = l1.combine(rs)
        -------------------------------------------------------
        Preconditions:
            rs- an linked-based List (List)
        Postconditions:
            returns
            new_list - the contents of the current List and rs
            are interlaced into new_list - current List and rs
            are empty (List)
        -------------------------------------------------------
        """
        new_list = List()
        new_list._combine_r_aux(self, rs)
        new_list.reverse()
        return new_list

    def _combine_r_aux(self, l1, l2):
        """
        -------------------------------------------------------
        Auxiliary recursive function for combine_r.
```

```
        Use: self._combine_aux(l1, l2)
        -------------------------------------------------------
        Preconditions:
            l1 - (List)
            l2 - (List)
        Postconditions:
            Contents of l1 and l2 are interlaced into the current List.
            l1 and L2 are empty.
        -------------------------------------------------------
        """
        if l1._front is not None and l2._front is not None:
            self._move_front(l1)
            self._combine_r_aux(l2, l1)
        elif l1._front is not None:
            self._move_front(l1)
            self._combine_r_aux(l1, l2)
        elif l2._front is not None:
            self._move_front(l2)
            self._combine_r_aux(l2, l1)
        return

    def __iter__(self):
        """
        USE FOR TESTING ONLY
        -------------------------------------------------------
        Generates a Python iterator. Iterates through the list
        from front to rear.
        Use: for v in s:
        -------------------------------------------------------
        Postconditions:
            yields
            value - the next value in the list (?)
        -------------------------------------------------------
        """
        current = self._front

        while current is not None:
            yield current._data
            current = current._next
```

OneClass find more resources at oneclass.com

```
"""
-------------------------------------------------------
queue_circular.py
Array version of the CircularQueue ADT.
-------------------------------------------------------
Author:  David Brown
ID:      999999999
Email:   dbrown@wlu.ca
__updated__ = "2018-02-03"
-------------------------------------------------------
"""
from copy import deepcopy


class CircularQueue:

    def __init__(self, max_size):
        """
        -------------------------------------------------------
        Initializes an empty queue. Data is stored in a list.
        Use: cq = CircularQueue(max_size)
        -------------------------------------------------------
        Preconditions:
            max_size - maximum size of the queue (int > 0)
        Postconditions:
            Initializes an empty queue.
        -------------------------------------------------------
        """
        assert max_size > 0, "CircularQueue size must be > 0"

        self._max_size = max_size
        self._values = [None] * self._max_size
        self._front = 0
        self._rear = 0
        self._count = 0
        return

    def empty(self):
        """
        -------------------------------------------------------
        Determines if the queue is empty.
        Use: b = cq.empty()
        -------------------------------------------------------
        Postconditions:
            Returns True if the queue is empty, False otherwise.
        -------------------------------------------------------
        """
        return self._count == 0

    def is_empty(self):
        """
        -------------------------------------------------------
        Determines if the queue is empty.
        Use: b = cq.is_empty()
        -------------------------------------------------------
        Postconditions:
            Returns True if the queue is empty, False otherwise.
        -------------------------------------------------------
        """
        return self._count == 0

    def full(self):
        """
        -------------------------------------------------------
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        Determines if the queue is full.
        Use: b = cq.full()
        -------------------------------------------------------
        Postconditions:
            Returns True if the queue is full, False otherwise.
        -------------------------------------------------------
        """
        return self._count == self._max_size

    def is_full(self):
        """
        -------------------------------------------------------
        Determines if the queue is full.
        Use: b = cq.is_full()
        -------------------------------------------------------
        Postconditions:
            Returns True if the queue is full, False otherwise.
        -------------------------------------------------------
        """
        return self._count == self._max_size

    def __len__(self):
        """
        -------------------------------------------------------
        Returns the length of the queue.
        Use: n = len(cq)
        -------------------------------------------------------
        Postconditions:
            Returns the number of values in the queue.
        -------------------------------------------------------
        """
        return self._count

    def insert(self, value):
        """
        -------------------------------------------------------
        Inserts a copy of value into the queue.
        Use: cq.insert( value )
        -------------------------------------------------------
        Preconditions:
            value - a data element (?)
        Postconditions:
            a copy of value is added to the rear of the queue.
        -------------------------------------------------------
        """
        assert self._count < self._max_size, "queue is full"

        self._values[self._rear] = deepcopy(value)
        self._rear = (self._rear + 1) % self._max_size
        self._count += 1
        return

    def remove(self):
        """
        -------------------------------------------------------
        Removes and returns value from the queue.
        Use: v = cq.remove()
        -------------------------------------------------------
        Postconditions:
            returns
            value - the value at the front of the queue - the value is
            removed from the queue (?)
        -------------------------------------------------------
        """
```

OneClass find more resources at oneclass.com

OneClass find more resources at oneclass.com

```
        assert self._count > 0, "Cannot remove from an empty queue"

        value = self._values[self._front]
        self._values[self._front] = None
        self._front = (self._front + 1) % self._max_size
        self._count -= 1
        return value

    def peek(self):
        """
        -------------------------------------------------------
        Peeks at the front of queue.
        Use: v = cq.peek()
        -------------------------------------------------------
        Postconditions:
            returns
            value - a copy of the value at the front of the queue -
            the value is not removed from the queue (?)
        -------------------------------------------------------
        """
        assert self._count > 0, "Cannot peek at an empty queue"

        value = deepcopy(self._values[self._front])
        return value

    def __iter__(self):
        """
        USE FOR TESTING ONLY
        -------------------------------------------------------
        Generates a Python iterator. Iterates through the queue
        from front to rear.
        Use: for v in cq:
        -------------------------------------------------------
        Postconditions:
            returns
            value - the next value in the queue (?)
        -------------------------------------------------------
        """
        j = self._front
        i = 0

        while i < self._count:
            yield self._values[j]
            i += 1
            j = (j + 1) % self._max_size
```

OneClass find more resources at oneclass.com