```
!date
```

```
Mon Apr 18 03:22:31 UTC 2022
```

Please run the above line to refresh the date before your submission.

CSCI-SHU 210 Data Structures

Recitation 10 Trees/Binary trees

You should work on the 5 tasks as written in the worksheet.

Name: your name

NetID: your netid

Please submit the following items to the Gradescope:

- Your Colab notebooklink (by clicking the Share button at the top-right corner of the Colab notebook, share to anyone)
- The printout of your run in Colab notebook in pdf format
- No late submission is permitted. All solutions must be from your own work. Total points of the assignment is 100.

LinkedQueue class (provided as a separate ADT. Use it only if you need to use it for your solution).

```
class LinkedQueue:
    """FIFO queue implementation using a singly linked list for storage."""

    #-------------------------- nested _Node class --------------------------
    class _Node:
        """Lightweight, nonpublic class for storing a singly linked node."""
        __slots__ = '_element', '_next'                # streamline memory usage

        def __init__(self, element, next):
            self._element = element
            self._next = next

    #------------------------------- queue methods -------------------------------
    def __init__(self):
        """Create an empty queue."""
        self._head = None
        self._tail = None
        self._size = 0                          # number of queue elements

    def __len__(self):
        """Return the number of elements in the queue."""
        return self._size

    def is_empty(self):
        """Return True if the queue is empty."""
        return self._size == 0

    def first(self):
        """Return (but do not remove) the element at the front of the queue.

        Raise Empty exception if the queue is empty.
        """
        if self.is_empty():
```

```
        if self.is_empty():
            raise Exception('Queue is empty')
        return self._head._element                    # front aligned with head of list

    def dequeue(self):
        """Remove and return the first element of the queue (i.e., FIFO).

        Raise Empty exception if the queue is empty.
        """
        if self.is_empty():
            raise Exception('Queue is empty')
        answer = self._head._element
        self._head = self._head._next
        self._size -= 1
        if self.is_empty():                          # special case as queue is empty
            self._tail = None                         # removed head had been the tail
        return answer

    def enqueue(self, e):
        """Add an element to the back of queue."""
        newest = self._Node(e, None)                  # node will be new tail node
        if self.is_empty():
            self._head = newest                       # special case: previously empty
        else:
            self._tail._next = newest
        self._tail = newest                           # update reference to tail node
        self._size += 1

    def __str__(self):
        result = []
        curNode = self._head
        while (curNode is not None):
            result.append(str(curNode._element) + " --> ")
            curNode = curNode._next
        result.append("None")
        return "".join(result)
```

The Binary Tree with OOP, as we defined in lecture

```
class Tree:
    class TreeNode:
        def __init__(self, element, parent = None, left = None, right = None):
            self._parent = parent
            self._element = element
            self._left = left
            self._right = right

        def element(self):
            return self._element

    #-------------------------- binary tree constructor --------------------------
    def __init__(self):
        """Create an initially empty binary tree."""
        self._root = None
        self._size = 0

    #-------------------------- public accessors ---------------------------------
    def __len__(self):
        """Return the total number of elements in the tree."""
        return self._size

    def is_root(self, node):
        """Return True if a given node represents the root of the tree."""
        return self._root == node
```

```python
    def is_leaf(self, node):
        """Return True if a given node does not have any children."""
        return self.num_children(node) == 0

    def is_empty(self):
        """Return True if the tree is empty."""
        return len(self) == 0

    def __iter__(self):
        """Generate an iteration of the tree's elements."""
        for node in self.nodes():                           # use same order as nodes()
            yield node._element                             # but yield each element

    def depth(self, node):
        """Return the number of levels separating a given node from the root."""
        if self.is_root(node):
            return 0
        else:
            return 1 + self.depth(self.parent(node))

    def height(self, node = None):     # time is linear in size of subtree
        if node is None:
            node = self._root
        if self.is_leaf(node):
            return 0
        else:
            if node._left:
                l = self.height(node._left)
            else:
                l = 0
            if node._right:
                r = self.height(node._right)
            else:
                r = 0
            return 1 + max(l, r)

    def nodes(self):
        """Generate an iteration of the tree's nodes."""
        return self.preorder()                              # return entire preorder iteration

    def preorder(self):
        """Generate a preorder iteration of nodes in the tree."""
        if not self.is_empty():
            for node in self._subtree_preorder(self._root):   # start recursion
                yield node

    def _subtree_preorder(self, node):
        """Generate a preorder iteration of nodes in subtree rooted at node."""
        yield node                                          # visit node before its subtrees
        for c in self.children(node):                       # for each child c
            for other in self._subtree_preorder(c):         # do preorder of c's subtree
                yield other                                 # yielding each to our caller

    def postorder(self):
        """Generate a postorder iteration of nodes in the tree."""
        if not self.is_empty():
            for node in self._subtree_postorder(self._root):   # start recursion
                yield node

    def _subtree_postorder(self, node):
        """Generate a postorder iteration of nodes in subtree rooted at node."""
        for c in self.children(node):               # for each child c
            for other in self._subtree_postorder(c): # do postorder of c's subtree
                yield other                          # yielding each to our caller
        yield node                                  # visit node after its subtrees
    def inorder(self):
        """Generate an inorder iteration of nodes in the tree."""
```

```
        if not self.is_empty():
            for node in self._subtree_inorder(self._root):
                yield node

    def _subtree_inorder(self, node):
        """Generate an inorder iteration of nodess in subtree rooted at p."""
        if node._left is not None:          # if left child exists, traverse its subtree
            for other in self._subtree_inorder(node._left):
                yield other
        yield node                          # visit p between its subtrees
        if node._right is not None:         # if right child exists, traverse its subtree
            for other in self._subtree_inorder(node._right):
                yield other


    def breadthfirst(self):
        """Generate a breadth-first iteration of the nodes of the tree."""
        if not self.is_empty():
                fringe = LinkedQueue()              # known nodes not yet yielded
                fringe.enqueue(self._root)          # starting with the root
                while not fringe.is_empty():
                        node = fringe.dequeue()     # remove from front of the queue
                        yield node                  # report this node
                        for c in self.children(node):
                                fringe.enqueue(c)   # add children to back of queue


    def root(self):
        """Return the root of the tree (or None if tree is empty)."""
        return self._root

    def parent(self, node):
        """Return node's parent (or None if node is the root)."""
        return node._parent

    def left(self, node):
        """Return node's left child (or None if no left child)."""
        return node._left

    def right(self, node):
        """Return node's right child (or None if no right child)."""
        return node._right

    def children(self, node):
        """Generate an iteration of nodes representing node's children."""
        if node._left is not None:
                yield node._left
        if node._right is not None:
                yield node._right

    def num_children(self, node):
        """Return the number of children of a given node."""
        count = 0
        if node._left is not None:      # left child exists
                count += 1
        if node._right is not None:     # right child exists
                count += 1
        return count

    def sibling(self, node):
        """Return a node representing given node's sibling (or None if no sibling)."""
        parent = node._parent
        if parent is None:                          # p must be the root
                return None                         # root has no sibling
        else:
                if node == parent._left:
                        return parent._right        # possibly None
```

```
            else:
                return parent._left                 # possibly None

    #-------------------------- nonpublic mutators --------------------------
    def add_root(self, e):
        """Place element e at the root of an empty tree and return the root node.

        Raise ValueError if tree nonempty.
        """
        if self._root is not None:
            raise ValueError('Root exists')
        self._size = 1
        self._root = self.TreeNode(e)
        return self._root

    def add_left(self, node, e):
        """Create a new left child for a given node, storing element e in the new node.

        Return the new node.
        Raise ValueError if node already has a left child.
        """
        if node._left is not None:
            raise ValueError('Left child exists')
        self._size += 1
        node._left = self.TreeNode(e, node)                # node is its parent
        return node._left

    def add_right(self, node, e):
        """Create a new right child for a given node, storing element e in the new node.

        Return the new node.
        Raise ValueError if node already has a right child.
        """
        if node._right is not None:
            raise ValueError('Right child exists')
        self._size += 1
        node._right = self.TreeNode(e, node)               # node is its parent
        return node._right

    def _replace(self, node, e):
        """Replace the element at given node with e, and return the old element."""
        old = node._element
        node._element = e
        return old

    def _delete(self, node):
        """Delete the given node, and replace it with its child, if any.

        Return the element that had been stored at the given node.
        Raise ValueError if node has two children.
        """
        if self.num_children(node) == 2:
            raise ValueError('Node has two children')
        child = node._left if node._left else node._right   # might be None
        if child is not None:
            child._parent = node._parent          # child's grandparent becomes parent
        if node is self._root:
            self._root = child                          # child becomes root
        else:
            parent = node._parent
            if node is parent._left:
                parent._left = child
            else:
                parent._right = child
        self._size -= 1
        return node._element
```

```
    def _attach(self, node, t1, t2):
        """Attach trees t1 and t2, respectively, as the left and right subtrees of the external node.

        As a side effect, set t1 and t2 to empty.
        Raise TypeError if trees t1 and t2 do not match type of this tree.
        Raise ValueError if node already has a child. (This operation requires a leaf node!)
        """
        if not self.is_leaf(node):
            raise ValueError('Node must be leaf')
        if not type(self) is type(t1) is type(t2):      # all 3 trees must be same type
            raise TypeError('Tree types must match')
        self._size += len(t1) + len(t2)
        if not t1.is_empty():               # attached t1 as left subtree of node
            t1._root._parent = node
            node._left = t1._root
            t1._root = None                         # set t1 instance to empty
            t1._size = 0
        if not t2.is_empty():               # attached t2 as right subtree of node
            t2._root._parent = node
            node._right = t2._root
            t2._root = None                         # set t2 instance to empty
            t2._size = 0
```

```
def pretty_print(tree):
    # ----------------------- Need to enter height to work -----------------
    levels = tree.height() + 1
    print("Levels:", levels)
    print_internal([tree._root], 1, levels)

def print_internal(this_level_nodes, current_level, max_level):
    if (len(this_level_nodes) == 0 or all_elements_are_None(this_level_nodes)):
        return  # Base case of recursion: out of nodes, or only None left

    floor = max_level - current_level;
    endgeLines = 2 ** max(floor - 1, 0);
    firstSpaces = 2 ** floor - 1;
    betweenSpaces = 2 ** (floor + 1) - 1;
    print_spaces(firstSpaces)
    next_level_nodes = []
    for node in this_level_nodes:
        if (node is not None):
            print(node._element, end = "")
            next_level_nodes.append(node._left)
            next_level_nodes.append(node._right)
        else:
            next_level_nodes.append(None)
            next_level_nodes.append(None)
            print_spaces(1)

        print_spaces(betweenSpaces)
    print()
    for i in range(1, endgeLines + 1):
        for j in range(0, len(this_level_nodes)):
            print_spaces(firstSpaces - i)
            if (this_level_nodes[j] == None):
                    print_spaces(endgeLines + endgeLines + i + 1);
                    continue
            if (this_level_nodes[j]._left != None):
                    print("/", end = "")
            else:
                    print_spaces(1)
            print_spaces(i + i - 1)
            if (this_level_nodes[j]._right != None):
                    print("\\", end = "")
            else:
                    print_spaces(1)
```

```
            print_spaces(endgeLines + endgeLines - i)
        print()

    print_internal(next_level_nodes, current_level + 1, max_level)

def all_elements_are_None(list_of_nodes):
    for each in list_of_nodes:
        if each is not None:
            return False
    return True

def print_spaces(number):
    for i in range(number):
        print(" ", end = "")
```

Task 1: create and perform some operations on a Binary Search Tree

```
class BinarySearchTree(Tree):

    #------------------------------- nonpublic utilities -------------------------------
    def _subtree_search(self, node, v):
        """Return the node having value v, or last node searched."""
        if v == node._element:                      # found match
            return node
        elif v < node._element:                     # search left subtree
            if node._left is not None:
                return self._subtree_search(node._left, v)
        else:                                       # search right subtree
            if node._right is not None:
                return self._subtree_search(node._right, v)
        return node                                 # unsucessful search

    def _subtree_first_node(self, node):
        """Return the node that contains the first item in subtree rooted at given node."""
        walk = node
        while walk._left is not None:    # keep walking left
            walk = walk._left
        return walk

    def _subtree_last_node(self, node):
        """Return the node that contains the last item in subtree rooted at given node."""
        walk = node
        while walk._right is not None:      # keep walking right
            walk = walk._right
        return walk

    #--------------------- public methods providing Binary Search Tree support ---------------------
    def first(self):
        """Return the first node (smallest node) in the tree (or None if empty)."""
        return self._subtree_first_node(self.root()) if len(self) > 0 else None

    def last(self):
        """Return the last node (largest node) in the tree (or None if empty)."""
        return self._subtree_last_node(self.root()) if len(self) > 0 else None

    def before(self, node):
        """Return the node that is just before the given node in the natural order.

        Return None if the given node is the first node.
        """
        if node._left is not None:
            return self._subtree_last_node(node._left)
        else:
            # walk upward
            walk = node
```

```
            walk = node
            above = walk._parent
            while above is not None and walk == above._left:
                walk = above
                above = walk._parent
            return above

    def after(self, node):
        """Return the node that is just after the given node in the natural order.

        Return None if the given node is the last node.
        """
        if node._right is not None:
            return self._subtree_first_node(node._right)
        else:
            walk = node
            above = walk._parent
            while above is not None and walk == above._right:
                walk = above
                above = walk._parent
            return above

    def delete(self, node):
        """Remove the given node."""
        if node._left and node._right:                      # node has two children
            replacement = self._subtree_last_node(node._left)
            self._replace(node, replacement._element)       # from BinaryTree(class Tree)
            node = replacement
        # now node has at most one child
        self._delete(node)
        #self._rebalance_delete(parent)

    #-------------------- public methods for accessing/mutating --------------------
    def get_node(self, v):
        """Return the node associated with value (raise Error if not found)."""
        if self.is_empty():
            raise Exception('Tree is empty')
        else:
            node = self._subtree_search(self._root, v)
            if v != node._element:
                raise ValueError('Not found: ' + repr(v))
            return node

    def insert(self, v):
        """Insert value v into the Binary Search Tree"""
        if self.is_empty():
            leaf = self.add_root(v)          # from BinaryTree (class Tree)
        else:
            node = self._subtree_search(self._root, v)
            if node._element < v:
                leaf = self.add_right(node, v)  # inherited from BinaryTree (class Tree)
            else:
                leaf = self.add_left(node, v)  # inherited from BinaryTree (class Tree)
        self._rebalance_insert(leaf)                     # (This line only works in AVL Tree)

    def delete_value(self, v):
        """Remove the node within the Tree that contains value v (raise Error if not found)."""
        if not self.is_empty():
            node = self._subtree_search(self._root, v)
            if v == node._element:
                self.delete(node)                  # reuse the delete node function
                return                             # successful deletion complete
        raise ValueError('Not found: ' + repr(v))

    def _rebalance_insert(self, p):          # Do nothing in BST, going to be overidden in AVLTree.
        pass

    def _rebalance_delete(self, p):          # Do nothing in BST, going to be overidden in AVLTree.
```

```
    def _rebalance_delete(self, p):          # Do nothing in BST, going to be overridden in AVLTree.
        pass

    def __iter__(self):
        """Generate an iteration of all values in order."""
        node = self.first()
        while node is not None:
            yield node._element
            node = self.after(node)

    def __reversed__(self):
        """Generate an iteration of all values in reverse order."""
        node = self.last()
        while node is not None:
            yield node._element
            node = self.before(node)

    ## Task 2 ##
    def minimum(self):
        return self.first()._element

    ## Task 3 ##
    def second_minimum(self):
        return self.after(self.first())._element

    ## Task 4 ##
    def is_valid(self):
        values = [x._element for x in self.inorder()]
        return values == sorted(values)

    ## Task 5 ##
    def iter_range(self, start, end):
        values = [x._element for x in self.inorder() \
                    if start <= x._element <= end]
        for each in values:
            yield each
```

```
print("------------------Task 1 Build BST-----------------")
# Constuct a BST
# 1. Insert 0, 1, 2, 3, 4 into the tree.
# 2. Get the Node of 2 by calling get_node(self, value) function.
# 3. Use before(self, node) function to get node of 1.
# 4. Use after(self, node) function to get node of 3.
# 5. Delete 0, 1, 2, 3, 4 from the tree.

## Task 1 ##
test_BST = BinarySearchTree()
test_BST.insert(0)
test_BST.insert(1)
test_BST.insert(2)
test_BST.insert(3)
test_BST.insert(4)
pretty_print(test_BST)
node_of_2 = test_BST.get_node(2)
print(test_BST.before(node_of_2).element())
print(test_BST.after(node_of_2).element())

for i in range(5):
    test_BST.delete_value(i)
```

```
------------------Task 1 Build BST-----------------
Levels: 5
               0
                \
                 \
                  \
                   \
```

```
                    \
                     \
                      \
                       \
                       1
                        \
                         \
                          \
                           \
                           2
                            \
                             \
                             3
                              \
                              4

1
3
```

Task 2: Minimum(self)

Implement function `minimum(self)` above. When called, the minimum element within the tree is returned.

```
######### Below part is for task 2 to task 5 ##########
# Construct a BST t2
#               3
#             /  \
#           /      \
#         /          \
#       /              \
#       1              7
#    /  \           /  \
#  /       \      /       \
#  0       2      5       8
#                 /  \       \
#                 4  6       9
t2 = BinarySearchTree()
t2.insert(3)
t2.insert(1)
t2.insert(7)
t2.insert(0)
t2.insert(2)
t2.insert(5)
t2.insert(8)
t2.insert(4)
t2.insert(6)
t2.insert(9)
print("------------------Task 2 minimum-----------------")
print("Minimum of tree is: ", t2.minimum(), ", Expected: 0")
```

```
------------------Task 2 minimum-----------------
Minimum of tree is:  0 , Expected: 0
```

Task 3: second_minimum(self)

Implement function second_minimum(self). When called, the second smallest element within the tree is returned.

```
print("------------------Task 3 second_minimum-----------------")
print("Second smallest of tree is: ", t2.second_minimum(), ", Expected: 1")
```

```
------------------Task 3 second_minimum-----------------
Second smallest of tree is:  1 , Expected: 1
```

Task 4: is_valid(self)

Implement function `is_valid(self)`. When called, returns True if the self tree is a valid binary search tree. Returns false otherwise.

```
print("-----------------Task 4 is_valid----------------")
print("Is the tree a valid BST?: ", t2.is_valid(), ", should be True")

############# Mess up t2 ###################
node_three = BinarySearchTree.TreeNode(3, t2.last(), None, None)
t2.last()._left = node_three
#                              3
#                            /  \
#                          /      \
#                        /          \
#                      /              \
#                    /                  \
#                  /                      \
#                /                          \
#              /                              \
#              1                              7
#            /  \                          /  \
#          /      \                      /      \
#        /          \                  /          \
#      /              \              /              \
#      0              2              5              8
#                                    /  \              \
#                                  /      \              \
#                                  4      6              9
#                                                        /
#                                                        3          <===  Problem!
print("Is the tree a valid BST?: ", t2.is_valid(), ", should be False")
```

```
-----------------Task 4 is_valid----------------
Is the tree a valid BST?:  True , should be True
Is the tree a valid BST?:  False , should be False
```

Task 5: iter_range(self, start, stop)

Implement function `iter_range(self, start, stop)`. When called, Yield a generator that contains elements in order, such that start <= elements <= stop.

```
print("-----------------Task 5 iter_range----------------")
t2 = BinarySearchTree()
t2.insert(3)
t2.insert(1)
t2.insert(7)
t2.insert(0)
t2.insert(2)
t2.insert(5)
t2.insert(8)
t2.insert(4)
t2.insert(6)
t2.insert(9)
print([x for x in t2.iter_range(3, 7)], ", Expected: [3, 4, 5, 6, 7]")
```

```
-----------------Task 5 iter_range----------------
[3, 4, 5, 6, 7] , Expected: [3, 4, 5, 6, 7]
```