```
!date

```

```
Fri Apr 15 09:48:21 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.

- For students who have recitation on Wednesday, you should submit your solutions by **Apr 15th** Friday 11:59pm.
- For students who have recitation on Thursday, you should submit your solutions by **Apr 16th** Saturday 11:59pm.
- For students who have recitation on Friday, you should submit your solutions by **Apr 17th** Sunday 11:59pm.

Name: Peter Yuncheng Yao

NetID: yy4108

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

    #------------------------- nested _Node class ----
    class _Node:
```

```
 6         """Lightweight, nonpublic class for storing a s
 7         __slots__ = '_element', '_next'         # strea
 8
 9         def __init__(self, element, next):
10             self._element = element
11             self._next = next
12
13     #------------------------------ queue methods ----
14     def __init__(self):
15         """Create an empty queue."""
16         self._head = None
17         self._tail = None
18         self._size = 0                          # numbe
19
20     def __len__(self):
21         """Return the number of elements in the queue."
22         return self._size
23
24     def is_empty(self):
25         """Return True if the queue is empty."""
26         return self._size == 0
27
28     def first(self):
29         """Return (but do not remove) the element at th
30
31         Raise Empty exception if the queue is empty.
32         """
33         if self.is_empty():
34             raise Exception('Queue is empty')
35         return self._head._element              # front
36
37     def dequeue(self):
38         """Remove and return the first element of the q
39
40         Raise Empty exception if the queue is empty.
41         """
42         if self.is_empty():
43             raise Exception('Queue is empty')
44         answer = self._head._element
45         self._head = self._head._next
46         self._size -= 1
```

```
47          if self.is_empty():                         # speci
48              self._tail = None                         # rem
49          return answer
50
51      def enqueue(self, e):
52          """Add an element to the back of queue."""
53          newest = self._Node(e, None)                # node
54          if self.is_empty():
55              self._head = newest                       # spe
56          else:
57              self._tail._next = newest
58          self._tail = newest                         # updat
59          self._size += 1
60
61      def __str__(self):
62          result = []
63          curNode = self._head
64          while (curNode is not None):
65              result.append(str(curNode._element) + " -->
66              curNode = curNode._next
67          result.append("None")
68          return "".join(result)
```

The Binary Tree with OOP, as we defined in lecture

```
 1 class Tree:
 2     class TreeNode:
 3         def __init__(self, element, parent = None, left
 4             self._parent = parent
 5             self._element = element
 6             self._left = left
 7             self._right = right
 8
 9         def element(self):
10             return self._element
11
12     #------------------------- binary tree constructor
13     def __init__(self):
14         """Create an initially empty binary tree."""
15         self._root = None
```

```
16          self._size = 0
17
18      #------------------------- public accessors ------
19      def __len__(self):
20          """Return the total number of elements in the t
21          return self._size
22
23      def is_root(self, node):
24          """Return True if a given node represents the r
25          return self._root == node
26
27      def is_leaf(self, node):
28          """Return True if a given node does not have an
29          return self.num_children(node) == 0
30
31      def is_empty(self):
32          """Return True if the tree is empty."""
33          return len(self) == 0
34
35      def __iter__(self):
36          """Generate an iteration of the tree's elements
37          for node in self.nodes():
38              yield node._element
39
40      def depth(self, node):
41          """Return the number of levels separating a giv
42          if self.is_root(node):
43              return 0
44          else:
45              return 1 + self.depth(self.parent(node))
46
47      def height(self, node = None):   # time is linear i
48          if node is None:
49              node = self._root
50          if self.is_leaf(node):
51              return 0
52          else:
53              if node._left:
54                  l = self.height(node._left)
55              else:
56                  l = 0
```

```
57              if node._right:
58                  r = self.height(node._right)
59              else:
60                  r = 0
61              return 1 + max(l, r)
62
63      def nodes(self):
64          """Generate an iteration of the tree's nodes.""
65          return self.preorder()
66
67      def preorder(self):
68          """Generate a preorder iteration of nodes in th
69          if not self.is_empty():
70              for node in self._subtree_preorder(self._ro
71                  yield node
72
73      def _subtree_preorder(self, node):
74          """Generate a preorder iteration of nodes in su
75          yield node
76          for c in self.children(node):
77              for other in self._subtree_preorder(c):
78                  yield other
79
80      def postorder(self):
81          """Generate a postorder iteration of nodes in t
82          if not self.is_empty():
83              for node in self._subtree_postorder(self._r
84                  yield node
85
86      def _subtree_postorder(self, node):
87          """Generate a postorder iteration of nodes in s
88          for c in self.children(node):
89              for other in self._subtree_postorder(c):
90                  yield other
91          yield node
92      def inorder(self):
93          """Generate an inorder iteration of nodes in th
94          if not self.is_empty():
95            for node in self._subtree_inorder(self._root)
96              yield node
97
```

```
 98     def _subtree_inorder(self, node):
 99         """Generate an inorder iteration of nodess in s
100         if node._left is not None:          # if left c
101           for other in self._subtree_inorder(node._left
102             yield other
103         yield node                                  # visi
104         if node._right is not None:         # if right
105           for other in self._subtree_inorder(node._righ
106             yield other
107
108
109     def breadthfirst(self):
110         """Generate a breadth-first iteration of the no
111         if not self.is_empty():
112             fringe = LinkedQueue()              # known
113             fringe.enqueue(self._root)        # startin
114             while not fringe.is_empty():
115                 node = fringe.dequeue()             # r
116                 yield node                          # r
117                 for c in self.children(node):
118                     fringe.enqueue(c)             # ad
119
120
121     def root(self):
122         """Return the root of the tree (or None if tree
123         return self._root
124
125     def parent(self, node):
126         """Return node's parent (or None if node is the
127         return node._parent
128
129     def left(self, node):
130         """Return node's left child (or None if no left
131         return node._left
132
133     def right(self, node):
134         """Return node's right child (or None if no rig
135         return node._right
136
137     def children(self, node):
138         """Generate an iteration of nodes representing
```

```
139         if node._left is not None:
140             yield node._left
141         if node._right is not None:
142             yield node._right
143 
144     def num_children(self, node):
145         """Return the number of children of a given nod
146         count = 0
147         if node._left is not None:     # left child exi
148             count += 1
149         if node._right is not None:    # right child ex
150             count += 1
151         return count
152 
153     def sibling(self, node):
154         """Return a node representing given node's sibl
155         parent = node._parent
156         if parent is None:                      # p must
157             return None                           # root
158         else:
159             if node == parent._left:
160                 return parent._right          # possibly
161             else:
162                 return parent._left          # possibly
163 
164     #--------------------------- nonpublic mutators ----
165     def add_root(self, e):
166         """Place element e at the root of an empty tree
167 
168         Raise ValueError if tree nonempty.
169         """
170         if self._root is not None:
171             raise ValueError('Root exists')
172         self._size = 1
173         self._root = self.TreeNode(e)
174         return self._root
175 
176     def add_left(self, node, e):
177         """Create a new left child for a given node, st
178 
179         Return the new node.
```

```
180         Raise ValueError if node already has a left chi
181         """
182         if node._left is not None:
183             raise ValueError('Left child exists')
184         self._size += 1
185         node._left = self.TreeNode(e, node)
186         return node._left
187
188     def add_right(self, node, e):
189         """Create a new right child for a given node, s
190
191         Return the new node.
192         Raise ValueError if node already has a right ch
193         """
194         if node._right is not None:
195             raise ValueError('Right child exists')
196         self._size += 1
197         node._right = self.TreeNode(e, node)
198         return node._right
199
200     def _replace(self, node, e):
201         """Replace the element at given node with e, an
202         old = node._element
203         node._element = e
204         return old
205
206     def _delete(self, node):
207         """Delete the given node, and replace it with i
208
209         Return the element that had been stored at the
210         Raise ValueError if node has two children.
211         """
212         if self.num_children(node) == 2:
213             raise ValueError('Node has two children')
214         child = node._left if node._left else node._rig
215         if child is not None:
216             child._parent = node._parent     # child's
217         if node is self._root:
218             self._root = child             # child beco
219         else:
220             parent = node._parent
```

```
221             if node is parent._left:
222                 parent._left = child
223             else:
224                 parent._right = child
225         self._size -= 1
226         return node._element
227
228     def _attach(self, node, t1, t2):
229         """Attach trees t1 and t2, respectively, as the
230
231         As a side effect, set t1 and t2 to empty.
232         Raise TypeError if trees t1 and t2 do not match
233         Raise ValueError if node already has a child. (
234         """
235         if not self.is_leaf(node):
236             raise ValueError('Node must be leaf')
237         if not type(self) is type(t1) is type(t2):    #
238             raise TypeError('Tree types must match')
239         self._size += len(t1) + len(t2)
240         if not t1.is_empty():          # attached t1 as
241             t1._root._parent = node
242             node._left = t1._root
243             t1._root = None              # set t1 instan
244             t1._size = 0
245         if not t2.is_empty():          # attached t2 as
246             t2._root._parent = node
247             node._right = t2._root
248             t2._root = None              # set t2 instan
249             t2._size = 0
```

```
  1 def pretty_print(tree):
  2     # ---------------------- Need to enter height to w
  3     levels = tree.height() + 1
  4     print("Levels:", levels)
  5     print_internal([tree._root], 1, levels)
  6
  7 def print_internal(this_level_nodes, current_level, max
  8     if (len(this_level_nodes) == 0 or all_elements_are_
  9         return  # Base case of recursion: out of nodes,
 10
 11     floor = max_level - current_level;
```

```
12      endgeLines = 2 ** max(floor - 1, 0);
13      firstSpaces = 2 ** floor - 1;
14      betweenSpaces = 2 ** (floor + 1) - 1;
15      print_spaces(firstSpaces)
16      next_level_nodes = []
17      for node in this_level_nodes:
18          if (node is not None):
19              print(node._element, end = "")
20              next_level_nodes.append(node._left)
21              next_level_nodes.append(node._right)
22          else:
23              next_level_nodes.append(None)
24              next_level_nodes.append(None)
25              print_spaces(1)
26
27          print_spaces(betweenSpaces)
28      print()
29      for i in range(1, endgeLines + 1):
30          for j in range(0, len(this_level_nodes)):
31              print_spaces(firstSpaces - i)
32              if (this_level_nodes[j] == None):
33                      print_spaces(endgeLines + endgeLine
34                      continue
35              if (this_level_nodes[j]._left != None):
36                      print("/", end = "")
37              else:
38                      print_spaces(1)
39              print_spaces(i + i - 1)
40              if (this_level_nodes[j]._right != None):
41                      print("\\", end = "")
42              else:
43                      print_spaces(1)
44              print_spaces(endgeLines + endgeLines - i)
45          print()
46
47      print_internal(next_level_nodes, current_level + 1,
48
49  def all_elements_are_None(list_of_nodes):
50      for each in list_of_nodes:
51          if each is not None:
52              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 utilitie
    def _subtree_search(self, node, v):
        """Return the node having value v, or last node
        if v == node._element:
            return node
        elif v < node._element:
            if node._left is not None:
                return self._subtree_search(node._left,
        else:
            if node._right is not None:
                return self._subtree_search(node._right
        return node

    def _subtree_first_node(self, node):
        """Return the node that contains the first item
        walk = node
        while walk._left is not None:                #
            walk = walk._left
        return walk

    def _subtree_last_node(self, node):
        """Return the node that contains the last item
        walk = node
        while walk._right is not None:               #
            walk = walk._right
        return walk

    #--------------------- public methods providing Bin
```

```
31      def first(self):
32          """Return the first node (smallest node) in the
33          return self._subtree_first_node(self.root()) if
34
35      def last(self):
36          """Return the last node (largest node) in the t
37          return self._subtree_last_node(self.root()) if
38
39      def before(self, node):
40          """Return the node that is just before the give
41
42          Return None if the given node is the first node
43          """
44          if node._left is not None:
45              return self._subtree_last_node(node._left)
46          else:
47              # walk upward
48              walk = node
49              above = walk._parent
50              while above is not None and walk == above._
51                  walk = above
52                  above = walk._parent
53              return above
54
55      def after(self, node):
56          """Return the node that is just after the given
57
58          Return None if the given node is the last node.
59          """
60          if node._right is not None:
61              return self._subtree_first_node(node._right
62          else:
63              walk = node
64              above = walk._parent
65              while above is not None and walk == above._
66                  walk = above
67                  above = walk._parent
68              return above
69
70      def delete(self, node):
71          """Remove the given node."""
```

```
72          if node._left and node._right:           # node
73              replacement = self._subtree_last_node(node.
74              self._replace(node, replacement._element)
75              node = replacement
76          # now node has at most one child
77          self._delete(node)
78          #self._rebalance_delete(parent)
79
80      #---------------------- public methods for accessing
81      def get_node(self, v):
82          """Return the node associated with value (raise
83          if self.is_empty():
84              raise Exception('Tree is empty')
85          else:
86              node = self._subtree_search(self._root, v)
87              if v != node._element:
88                  raise ValueError('Not found: ' + repr(v
89              return node
90
91      def insert(self, v):
92          """Insert value v into the Binary Search Tree""
93          if self.is_empty():
94              leaf = self.add_root(v)     # from BinaryTr
95          else:
96              node = self._subtree_search(self._root, v)
97              if node._element < v:
98                  leaf = self.add_right(node, v)        #
99              else:
100                 leaf = self.add_left(node, v)         #
101         self._rebalance_insert(leaf)                  #
102
103     def delete_value(self, v):
104         """Remove the node within the Tree that contain
105         if not self.is_empty():
106             node = self._subtree_search(self._root, v)
107             if v == node._element:
108                 self.delete(node)
109                 return
110         raise ValueError('Not found: ' + repr(v))
111
112     def _rebalance_insert(self, p):     # Do nothing in
```

```
113         pass
114
115     def _rebalance_delete(self, p):     # Do nothing in
116         pass
117
118     def __iter__(self):
119         """Generate an iteration of all values in order
120         node = self.first()
121         while node is not None:
122             yield node._element
123             node = self.after(node)
124
125     def __reversed__(self):
126         """Generate an iteration of all values in rever
127         node = self.last()
128         while node is not None:
129             yield node._element
130             node = self.before(node)
131
132     ## Task 2 ##
133     def minimum(self):
134         return self.first()._element
135
136     ## Task 3 ##
137     def second_minimum(self):
138         return self.after(self.first())._element
139
140     ## Task 4 ##
141     def is_valid(self):
142         if self.is_empty():
143             return True
144         walk=self.first()
145         # future=self.after(walk
146         while walk:
147             temp=self.after(walk)
148             if temp and temp._element<=walk._element:
149                 return False
150             walk=temp
151         return True
152
153
```

```
154     ## Task 5 ##
155     def iter_range(self, start, end):
156         begin=self._subtree_search(self._root, start)
157         if begin._element<start:
158             begin=self.after(begin)
159         last=self._subtree_search(self._root, end)
160         if last._element > end:
161             last=self.before(last)
162         walk=begin
163         while walk!=last:
164             yield walk._element
165             walk=self.after(walk)
166         yield(last._element)
167
168
169
170
```

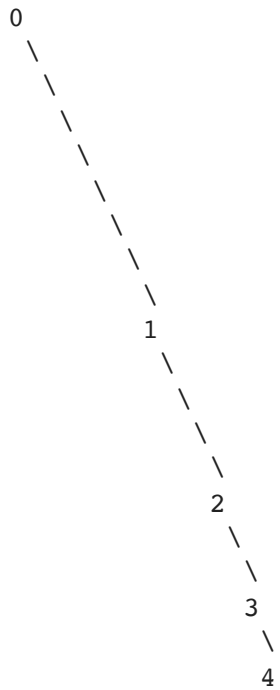
```
 1 print("------------------Task 1 Build BST-------------
 2 # Constuct a BST
 3 # 1. Insert 0, 1, 2, 3, 4 into the tree.
 4 # 2. Get the Node of 2 by calling get_node(self, value)
 5 # 3. Use before(self, node) function to get node of 1.
 6 # 4. Use after(self, node) function to get node of 3.
 7 # 5. Delete 0, 1, 2, 3, 4 from the tree.
 8
 9 ## Task 1 ##
10 t=BinarySearchTree()
11 t.insert(0)
12 t.insert(1)
13 t.insert(2)
14 t.insert(3)
15 t.insert(4)
16 # t.pretty_print()
17 pretty_print(t)
18 temp=t.get_node(2)
19 print(temp._element)
20 # pretty_print(t)
21
22 before_temp=t.before(temp)
23 # print(before_temp)
24 # pretty_print(t)
```

```
# pretty_print(t)
print(before_temp._element)


after_temp=t.after(temp)
# print(after_temp)
# pretty_print(t)
print(after_temp._element)


for i in t.inorder():
    t.delete(i)
# pretty_print(t)
```

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




```
2
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
```

```
------------------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
```

```
------------------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(), ", s

############# Mess up t2 ###################
node_three = BinarySearchTree.TreeNode(3, t2.last(), No
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(), ", s
```

```
------------------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-------------
```

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

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