```
!date
```

```
Sat Apr  9 01:50:34 UTC 2022
```

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

```
1
```

CSCI-SHU 210 Data Structures

Recitation 9 Trees/Binary trees

- For students who have recitation on Wednesday, you should submit your solutions by Friday 11:59pm.
- For students who have recitation on Thursday, you should submit your solutions by Saturday 11:59pm.
- For students who have recitation on Friday, you should submit your solutions by 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.

Part I: Simple Binary Tree (Just the Tree node class):

Subpart 1: implement depth first traversals: Pre-order, Post-order and In-order

```
class TreeWithoutParent:
```

```
 2     def __init__(self, element, left=None, right=None):
 3         self._element = element
 4         self._left = left
 5         self._right = right
 6 
 7     def __str__(self):
 8         return str(self._element)
 9 
10 def PreOrderTraversal(tree: TreeWithoutParent):
11     # Prints all the elements with pre order traversal,
12     if tree == None:
13         return
14     # to do
15     print(tree, end=' ')
16     PreOrderTraversal(tree._left)
17     PreOrderTraversal(tree._right)
18 
19 def PostOrderTraversal(tree: TreeWithoutParent):
20     # Prints all the elements with post order traversal
21     if tree == None:
22         return
23     # to do
24     PostOrderTraversal(tree._left)
25     PostOrderTraversal(tree._right)
26     print(tree, end=' ')
27 
28 def InOrderTraversal(tree: TreeWithoutParent):
29     # Prints all the elements with in order traversal,
30     if tree == None:
31         return
32     # to do
33     InOrderTraversal(tree._left)
34     print(tree, end=' ')
35     InOrderTraversal(tree._right)
36 
37 
38 ### Uncomment the following code if you want to print t
39 ### We assume that you had variables:  _left, _right, _
40 
41 
42 def pretty_print(A):
```

```
43      levels = 3      # Need a function to calculate leve
44      print_internal([A], 1, levels)
45
46  def print_internal(this_level_nodes, current_level, max
47      if (len(this_level_nodes) == 0 or all_elements_are_
48          return  # Base case of recursion: out of nodes,
49
50      floor = max_level - current_level;
51      endgeLines = 2 ** max(floor - 1, 0);
52      firstSpaces = 2 ** floor - 1;
53      betweenSpaces = 2 ** (floor + 1) - 1;
54      print_spaces(firstSpaces)
55      next_level_nodes = []
56      for node in this_level_nodes:
57          if (node is not None):
58              print(node._element, end = "")
59              next_level_nodes.append(node._left)
60              next_level_nodes.append(node._right)
61          else:
62              next_level_nodes.append(None)
63              next_level_nodes.append(None)
64              print_spaces(1)
65
66          print_spaces(betweenSpaces)
67      print()
68      for i in range(1, endgeLines + 1):
69          for j in range(0, len(this_level_nodes)):
70              print_spaces(firstSpaces - i)
71              if (this_level_nodes[j] == None):
72                      print_spaces(endgeLines + endgeLine
73                      continue
74              if (this_level_nodes[j]._left != None):
75                      print("/", end = "")
76              else:
77                      print_spaces(1)
78              print_spaces(i + i - 1)
79              if (this_level_nodes[j]._right != None):
80                      print("\\", end = "")
81              else:
82                      print_spaces(1)
83              print_spaces(endgeLines + endgeLines - i)
```

```
        print()

    print_internal(next_level_nodes, current_level + 1,

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 = "")
```

Subpart 2: Create a tree

```
##Create a expression tree for this expression:  3*2 +
## to do
head=TreeWithoutParent('+')
head._left=TreeWithoutParent('*')
head._right=TreeWithoutParent("-")
head._left._left=TreeWithoutParent(3)
head._left._right=TreeWithoutParent(2)
head._right._left=TreeWithoutParent(5)
head._right._right=TreeWithoutParent(2)
tree=head
```

```
print("\nUsing Tree Data Structure Without a parent")


##pretty_print(tree) #Call pretty_print to print the tr

print("\nPreOrder:")
PreOrderTraversal(tree)       # should print: + * 3 2 -
print("\nPostOrder:")
PostOrderTraversal(tree)    # should print: 3 2 * 5 2 -
print("\nInOrder:")
InOrderTraversal(tree)        # should print: 3 * 2 + 5
```

```
Using Tree Data Structure Without a parent
```

```
PreOrder:
+ * 3 2 - 5 2
PostOrder:
3 2 * 5 2 - +
InOrder:
3 * 2 + 5 - 2
```

Subpart 3: implement breadth first traversal: Level-order

- Use a Queue in your implementation as a temporal storage place

```
 1 class LinkedQueue:
 2     """FIFO queue implementation using a singly linked
 3
 4     #------------------------- nested _Node class ----
 5     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)
```

```
1 def LevelOrderTraversal(tree):
2     # Prints all the elements with level order traversa
3     if tree == None:
```

```
        return
    # to do
    level=LinkedQueue()
    level.enqueue(tree)
    while not level.is_empty():
        temp=level.dequeue()
        print(temp._element, end=' ')
        if temp._left:
            level.enqueue(temp._left)
        if temp._right:
            level.enqueue(temp._right)

print("\nLevelOrderOrder:")
LevelOrderTraversal(tree)        # should print: + * -
```

```
LevelOrderOrder:
+ * - 3 2 5 2
```

Part 2: Binary Tree with OOP

- get an idea of this method of implementing a tree
- utilize the methods already defined
- work on Task1 to Task7 as asked in the Worksheet

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

        def __str__(self):
            return str(self._element)

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

    #                                   public accessors
```

```
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 nodes(self):
41          """Generate an iteration of the tree's nodes.""
42          return self.preorder()
43
44      def preorder(self):
45          """Generate a preorder iteration of nodes in th
46          if not self.is_empty():
47              for node in self._subtree_preorder(self._ro
48                  yield node
49
50      def _subtree_preorder(self, node):
51          """Generate a preorder iteration of nodes in su
52          yield node
53          for c in self.children(node):
54              for other in self._subtree_preorder(c):
55                  yield other
56
57      def postorder(self):
58          """Generate a postorder iteration of nodes in t
59          if not self.is_empty():
```

```
60              for node in self._subtree_postorder(self._r
61                  yield node
62
63      def _subtree_postorder(self, node):
64          """Generate a postorder iteration of nodes in s
65          for c in self.children(node):
66              for other in self._subtree_postorder(c):
67                  yield other
68          yield node
69      def inorder(self):
70          """Generate an inorder iteration of positions i
71          if not self.is_empty():
72            for node in self._subtree_inorder(self._root)
73              yield node
74
75      def _subtree_inorder(self, node):
76          """Generate an inorder iteration of positions i
77          if node._left is not None:          # if left c
78            for other in self._subtree_inorder(node._left
79              yield other
80          yield node                                  # visi
81          if node._right is not None:         # if right
82            for other in self._subtree_inorder(node._righ
83              yield other
84
85
86      def breadthfirst(self):
87          """Generate a breadth-first iteration of the no
88          if not self.is_empty():
89              fringe = LinkedQueue()              # known
90              fringe.enqueue(self._root)        # startin
91              while not fringe.is_empty():
92                  node = fringe.dequeue()             # r
93                  yield node                          # r
94                  for c in self.children(node):
95                      fringe.enqueue(c)             # ad
96
97
98      def root(self):
99          """Return the root of the tree (or None if tree
100         return self._root
101
```

```
102     def parent(self, node):
103         """Return node's parent (or None if node is the
104         return node._parent
105 
106     def left(self, node):
107         """Return node's left child (or None if no left
108         return node._left
109 
110     def right(self, node):
111         """Return node's right child (or None if no rig
112         return node._right
113 
114     def children(self, node):
115         """Generate an iteration of nodes representing
116         if node._left is not None:
117             yield node._left
118         if node._right is not None:
119             yield node._right
120 
121     def num_children(self, node):
122         """Return the number of children of a given nod
123         count = 0
124         if node._left is not None:     # left child exi
125             count += 1
126         if node._right is not None:    # right child ex
127             count += 1
128         return count
129 
130     def sibling(self, node):
131         """Return a node representing given node's sibl
132         parent = node._parent
133         if parent is None:                      # p must
134             return None                           # root
135         else:
136             if node == parent._left:
137                 return parent._right          # possibly
138             else:
139                 return parent._left          # possibly
140 
141     #------------------------- nonpublic mutators ----
142     def add_root(self, e):
143         """Place element e at the root of an empty tree
```

```
144
145         Raise ValueError if tree nonempty.
146         """
147         if self._root is not None:
148             raise ValueError('Root exists')
149         self._size = 1
150         self._root = self.TreeNode(e)
151         return self._root
152
153     def add_left(self, node, e):
154         """Create a new left child for a given node, st
155
156         Return the new node.
157         Raise ValueError if node already has a left chi
158         """
159         if node._left is not None:
160             raise ValueError('Left child exists')
161         self._size += 1
162         node._left = self.TreeNode(e, node)
163         return node._left
164
165     def add_right(self, node, e):
166         """Create a new right child for a given node, s
167
168         Return the new node.
169         Raise ValueError if node already has a right ch
170         """
171         if node._right is not None:
172             raise ValueError('Right child exists')
173         self._size += 1
174         node._right = self.TreeNode(e, node)
175         return node._right
176
177     def _replace(self, node, e):
178         """Replace the element at given node with e, an
179         old = node._element
180         node._element = e
181         return old
182
183     def _delete(self, node):
184         """Delete the given node, and replace it with i
185
```

```
186         Return the element that had been stored at the
187         Raise ValueError if node has two children.
188         """
189         if self.num_children(node) == 2:
190             raise ValueError('Position has two children
191         child = node._left if node._left else node._rig
192         if child is not None:
193             child._parent = node._parent     # child's
194         if node is self._root:
195             self._root = child             # child beco
196         else:
197             parent = node._parent
198             if node is parent._left:
199                 parent._left = child
200             else:
201                 parent._right = child
202         self._size -= 1
203         return node._element
204
205
206
207     def _attach(self, node, t1, t2):
208         """Attach trees t1 and t2, respectively, as the
209
210         As a side effect, set t1 and t2 to empty.
211         Raise TypeError if trees t1 and t2 do not match
212         Raise ValueError if node already has a child. (
213         """
214         if not self.is_leaf(node):
215             raise ValueError('position must be leaf')
216         if not type(self) is type(t1) is type(t2):    #
217             raise TypeError('Tree types must match')
218         self._size += len(t1) + len(t2)
219         if not t1.is_empty():          # attached t1 as
220             t1._root._parent = node
221             node._left = t1._root
222             t1._root = None              # set t1 instan
223             t1._size = 0
224         if not t2.is_empty():          # attached t2 as
225             t2._root._parent = node
226             node._right = t2._root
227             t2._root = None              # set t2 instan
```

```
228             t2._size = 0
229
230     def preorderPrint(self,node):
231         """
232         :param node: TreeNode -- a given TreeNode.
233
234         Prints all the elements with pre order traversa
235
236         :return: nothing.
237         """
238         # temp=Tree()
239         # temp._root=node
240         # temp._size=self._size
241         # for i in temp.preorder():
242         #     print(i, end=' ')
243         for i in self._subtree_preorder(node):
244             print(i, end=' ')
245
246
247
248     def postorderPrint(self,node):
249         """
250         :param node: TreeNode -- a given TreeNode.
251
252         Prints all the elements with post order travers
253
254         :return: nothing.
255         """
256         # temp=Tree()
257         # temp._root=node
258         # temp._size=self._size
259         # for i in temp.postorder():
260         #     print(i, end=' ')
261         for i in self._subtree_postorder(node):
262             print(i, end=' ')
263
264     def inorderPrint(self,node):
265         """
266         :param node: TreeNode -- a given TreeNode.
267
268         Prints all the elements with in order traversal
269
```

```
270         :return: nothing.
271         """
272         for i in self._subtree_inorder(node):
273             print(i, end=' ')
274         # temp=Tree()
275         # temp._root=node
276         # temp._size=self._size
277         # for i in temp.inorder():
278         #     print(i, end=' ')
279         # if not node:
280         #     return None
281         # self.inorderPrint(node._left)
282         # print(node._element, end=' ')
283         # self.inorderPrint(node._right)
284 
285 
286     def levelorderPrint(self, node):
287         """
288         :param node: TreeNode -- a given TreeNode.
289 
290         Prints all the elements with level order traver
291 
292         :return: nothing.
293         """
294         temp=Tree()
295         temp._root=node
296         temp._size=self._size
297         for i in temp.breadthfirst():
298             print(i, end=' ')
299 
300 
301 
302     def height(self, node = None):
303         """
304         :param node: TreeNode -- a given TreeNode.
305 
306         Return the height of the subtree rooted at a gi
307         If node is None, return the height of the entir
308 
309         :return: height of subtree, integer.
310         """
311         if not node:
```

```
312             return self.height(self._root)
313         if self.is_leaf(node):
314             return 0
315         cur_max=-1
316         for i in self.children(node):
317             temp=self.height(i)
318             if temp>cur_max:
319                 cur_max=temp
320         return cur_max+1
321
322
323
324
325     def depth(self, node):
326         """
327         :param node: TreeNode -- a given TreeNode.
328
329         Return the number of levels separating a given
330
331         :return: depth of a node, integer
332         """
333         # pass
334         walk=node
335         count=-1
336         while walk:
337             walk=self.parent(walk)
338             count+=1
339         return count
340
341
342     def return_max(self):
343         """
344         :return: maximum value stored within self tree.
345         """
346         temp_max=-float("inf")
347         for i in self.preorder():
348             if i._element>temp_max:
349                 temp_max=i._element
350         return temp_max
351
352
353
```

```
354     def flip_node(self, node):
355         """
356         :param node: TreeNode -- a given TreeNode.
357 
358         flips the left and right children of a given no
359 
360         :return: nothing, modify self Tree in place.
361         """
362         original_size=self._size
363         l=node._left
364         r=node._right
365         new_l=Tree()
366         new_l._size=1
367 
368         new_l._root=l
369         new_r=Tree()
370         new_r._root=r
371         new_r._size=1
372         node._left=node._right=None
373         self._attach(node, new_r, new_l)
374         self._size=original_size
375         return
376 
377 
378     def flip_tree(self, node = None):
379         """
380         :param node: a given TreeNode.
381 
382         flips the left and right children all nodes in
383         and if parameter node is omitted it flips the e
384 
385         :return: nothing, modify self Tree in place.
386         """
387         if not node:
388             self.flip_tree(self._root)
389         if self.is_leaf(node):
390             return
391         self.flip_node(node)
392         self.flip_tree(node._left)
393         self.flip_tree(node._right)
394 
395 
```

```
396
397 def pretty_print(tree):
398     # ---------------------- Need to enter height to w
399     levels = tree.height() + 1
400     print("Levels:", levels)
401     print_internal([tree._root], 1, levels)
402
403 def print_internal(this_level_nodes, current_level, max
404     if (len(this_level_nodes) == 0 or all_elements_are_
405         return  # Base case of recursion: out of nodes,
406
407     floor = max_level - current_level;
408     endgeLines = 2 ** max(floor - 1, 0);
409     firstSpaces = 2 ** floor - 1;
410     betweenSpaces = 2 ** (floor + 1) - 1;
411     print_spaces(firstSpaces)
412     next_level_nodes = []
413     for node in this_level_nodes:
414         if (node is not None):
415             print(node._element, end = "")
416             next_level_nodes.append(node._left)
417             next_level_nodes.append(node._right)
418         else:
419             next_level_nodes.append(None)
420             next_level_nodes.append(None)
421             print_spaces(1)
422
423         print_spaces(betweenSpaces)
424     print()
425     for i in range(1, endgeLines + 1):
426         for j in range(0, len(this_level_nodes)):
427             print_spaces(firstSpaces - i)
428             if (this_level_nodes[j] == None):
429                     print_spaces(endgeLines + endgeLine
430                     continue
431             if (this_level_nodes[j]._left != None):
432                     print("/", end = "")
433             else:
434                     print_spaces(1)
435             print_spaces(i + i - 1)
436             if (this_level_nodes[j]._right != None):
437                     print("\\", end = "")
```

```
                    print(" \\", end = "")
                else:
                    print_spaces(1)
                print_spaces(endgeLines + endgeLines - i)
            print()

    print_internal(next_level_nodes, current_level + 1,

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 = "")



def main():
    ''' The following code will construct this tree:
                  -
                 / \
                /   \
               /     \
              /       \
              *       +
             / \     / \
            /   \   /   \
            +   4   -   2
           / \     / \
           3 1     9 5
    '''
    t = Tree()
    a = t.add_root("-")
    b = t.add_left(a, "*")
    c = t.add_right(a, "+")
    d = t.add_left(b, "+")
    e = t.add_right(b, 4)
    t.add_left(d, 3)
    t.add_right(d, 1)
```

```
479     t.add_right(d, 1)
480     f = t.add_left(c, "-")
481     t.add_right(c, 2)
482     t.add_left(f, 9)
483     k = t.add_right(f, 5)
484     #pretty_print(t)
485
486     ''' The following code will construct this tree:
487                 1
488               / \
489              /   \
490             /     \
491            /       \
492            2       3
493           / \
494          /   \
495          4   5
496         / \
497         6 7
498     '''
499
500     t2 = Tree()
501     a2 = t2.add_root(1)
502     b2 = t2.add_left(a2, 2)
503     c2 = t2.add_right(a2, 3)
504     d2 = t2.add_left(b2, 4)
505     e2 = t2.add_right(b2, 5)
506     f2 = t2.add_left(d2, 6)
507     g2 = t2.add_right(d2, 7)
508     #pretty_print(t2)
509
510     print("-------------------Testing task 1 preorder--
511     t.preorderPrint(a)  # a is the root
512     print()
513     print("-------------------Testing task 2 inorder---
514     t.inorderPrint(a) # a is the root
515     print()
516     print("-------------------Testing task 3 postorder-
517     t.postorderPrint(a)  # a is the root
518     print()
519     print("-------------------Testing task 4 levelorder
520     t.levelorderPrint(a)  # a is the root
521     print()
```

```
521     print()
522     print("-------------------Testing task 5 height----
523     print("Height of Tree 1: Expected: 3;   Your answer
524     print("Height of Tree 2: Expected: 3;   Your answer
525     print("-------------------Testing task 6 depth-----
526     print("Depth of '*' in Tree 1: Expected: 1;    Your
527     print("Depth of root in Tree 1: Expected: 0;    You
528     print("Depth of leaf in Tree 1: Expected: 3;    You
529     print("-------------------Testing task 7 return_max
530     print("Max value within Tree 2: Expected: 7;    You
531     print("-------------------Testing task 8 flip_node-
532     print("Tree 2 before flip_node:")
533     pretty_print(t2)
534     t2.flip_node(t2._root)
535     print("Tree 2 after flip_node:")
536     pretty_print(t2)
537     print("-------------------Testing task 9 flip_tree-
538     print("Tree 1 before flip_tree:")
539     pretty_print(t)
540     t.flip_tree(t._root)
541     print("Tree 1 after flip_tree:")
542     pretty_print(t)
543
544 main()
```

```
-------------------Testing task 1 preorder--------------------
- * + 3 1 4 + - 9 5 2
-------------------Testing task 2 inorder---------------------
3 + 1 * 4 - 9 - 5 + 2
-------------------Testing task 3 postorder-------------------
3 1 + 4 * 9 5 - 2 + -
-------------------Testing task 4 levelorderPrint-------------------
- * + + 4 - 2 3 1 9 5
-------------------Testing task 5 height----------------------
Height of Tree 1: Expected: 3;   Your answer: 3
Height of Tree 2: Expected: 3;   Your answer: 3
-------------------Testing task 6 depth-----------------------
Depth of '*' in Tree 1: Expected: 1;    Your answer: 1
Depth of root in Tree 1: Expected: 0;    Your answer: 0
Depth of leaf in Tree 1: Expected: 3;    Your answer: 3
-------------------Testing task 7 return_max------------------
Max value within Tree 2: Expected: 7;    Your answer: 7
-------------------Testing task 8 flip_node-------------------
Tree 2 before flip_node:
Levels: 4
       1
      / \
     /   \
    /     \
   /       \
```

```
   2       3
  / \
 /   \
 4   5
/ \
6 7

Tree 2 after flip_node:
Levels: 4
        1
       / \
      /   \
     /     \
    /       \
   /         \
   3         2
            / \
           /   \
           4   5
          / \
          6 7

--------------------Testing task 9 flip_tree---------------------
Tree 1 before flip_tree:
Levels: 4
        -
       / \
      /   \
     /     \
    /       \
   /         \
   *         +
  / \       / \
 /   \     /   \
 +   4     -   2
```