20171248 안재형

CAUSWE 2021

Algorithm Course - Class#2

(Prof.Eunwoo Kim)

Assignment #3

Results(Output)

Problem #1

```
Problem #1
2
4
2
1
7
8
3
9
5
4

4
5
9
3
8
7
1
101
4
2
```

Problem #2

```
Problem #2
20
24
78
20
73
20
24
```

```
68
20
20
20
24
78
76
46
20
24
78
20
20

20
24
78
73
68
76
46
```

Problem #3

```
Problem #3
This is NOT a valid Binary Search Tree
```

Problem #4

```
Problem #4
Write the index of two Nodes to find lowest common ancestor
first node:3
second node:4
2


Write the index of two Nodes to find lowest common ancestor
first node:5
second node:6
8


Write the index of two Nodes to find lowest common ancestor
first node:2
second node:7
6
```

Problem #5

```
Problem #5 (0 is red, 1 is black)
[[[[None,8-0,None],12-1,None],19-0,[None,31-1,None]],38-1,[None,41-1,None]]
```

Codes

Problem #1, #2

```
import random


class LL:
    top_node = None

    @classmethod
    def last_node(cls):
        if cls.top_node is None:
            return None
        else:
            current_node = cls.top_node
            while current_node.next is not None:
                current_node = current_node.next
            return current_node

    @classmethod
    def num_of_nodes(cls):
        current_node = cls.top_node
        result = 0
        while current_node is not None:
            current_node = current_node.next
            result += 1
        return result

    @classmethod
    def search_node(cls, index):
        if index < 0:
            return None
        elif index >= cls.num_of_nodes():
            return None
        current_index = 0
        current_node = cls.top_node
        while current_index < index:
            current_index += 1
            current_node = current_node.next
        return current_node

    @classmethod
    def append(cls, value):
        if cls.top_node is None:
            cls.top_node = LL(value)
        else:
```

```
            cls.last_node().next = LL(value)

    @classmethod
    def insert(cls, at, value):
        if at > (cls.num_of_nodes() + 1):
            raise NameError("Index Overflow")
        else:
            node_new = LL(value)
            node_before = cls.search_node(at - 1)
            node_after = cls.search_node(at)

            if node_before is None:
                cls.top_node = node_new
            else:
                node_before.next = node_new

            if node_after is not None:
                node_new.next = node_after




    @classmethod
    def traverse(cls):
        current_node = cls.top_node
        while current_node is not None:
            print(current_node.value)
            current_node = current_node.next
        print("")

    @classmethod
    def remove(cls, index):
        if index >= cls.num_of_nodes():
            raise NameError("Index Overflow")
        else:
            if index == 0:
                cls.top_node = cls.top_node.next
            elif index == cls.num_of_nodes() - 1:
                cls.search_node(index - 1).next = None
            else:
                cls.search_node(index - 1).next =
cls.search_node(index + 1)

    # 1
    @classmethod
    def reverse(cls):
        if cls.num_of_nodes() <= 0:
            raise NameError("Empty List")
        else:
            new_top_node = cls.last_node()
            new_last_node = new_top_node
            cls.remove(cls.num_of_nodes() - 1)
            while cls.num_of_nodes() > 0:
```

```
                new_last_node.next = cls.last_node()
                new_last_node = new_last_node.next
                cls.remove(cls.num_of_nodes() - 1)
            cls.top_node = new_top_node

    # 2
    @classmethod
    def remove_duplicates(cls):
        current_node = cls.top_node
        current_node_index = 0
        values = []
        while current_node is not None:
            if current_node.value in values:
                LL.remove(current_node_index)
                current_node_index -= 1
            else:
                values.append(current_node.value)

            current_node = current_node.next
            current_node_index += 1

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


# Problem 1
print ("Problem #1")
LL.top_node = None
for i in range(10):
    LL.append(random.randrange(1, 10))
LL.traverse()
LL.insert(2, 101)
LL.remove(3)
LL.reverse()
LL.traverse()


# Problem 2
print ("Problem #2")
LL.top_node = None
LL.append(20)
LL.append(24)
LL.append(78)
LL.append(20)
LL.append(73)
LL.append(20)
LL.append(24)
LL.append(68)
LL.append(20)
LL.append(20)
LL.append(20)
LL.append(24)
```

```
LL.append(78)
LL.append(76)
LL.append(46)
LL.append(20)
LL.append(24)
LL.append(78)
LL.append(20)
LL.append(20)
LL.traverse()
LL.remove_duplicates()
LL.traverse()
```

Problem #3, #4

```
def descendant_indices(tree, node_index):
    if node_index >= len(tree):
        return []
    elif tree[node_index] is None:
        return []
    else:
        return [node_index] \
               + descendant_indices(tree, (node_index * 2) + 1) \
               + descendant_indices(tree, (node_index * 2) + 2)


def ancestor_indices(_, node_index):
    ancestors = [node_index]
    latest_ancestor_index = node_index
    while latest_ancestor_index > 0:
        if (latest_ancestor_index % 2) == 0:
            latest_ancestor_index -= 1
        latest_ancestor_index = ((latest_ancestor_index - 1) / 2)
        ancestors.append(latest_ancestor_index)
    ancestors.reverse()
    # print(ancestors)
    return ancestors


def check_if_CBT(tree):
    i = 0
    while i < len(tree):
        # print(i, tree[i])
        if tree[i] is None:
            i += 1
            continue
        else:
            left_node_values = list(map(lambda index: tree[index],
descendant_indices(tree, (i * 2) + 1)))
            if len(list(filter(lambda value: tree[i] < value,
left_node_values))) > 0:
                print("This is NOT a valid Binary Search Tree\n")
                return
```

```
            right_node_values = list(map(lambda index: tree[index],
descendant_indices(tree, (i * 2) + 2)))
            if len(list(filter(lambda value: tree[i] > value,
right_node_values))) > 0:
                print("This is NOT a valid Binary Search Tree\n")
                return
            i += 1
    print("This is a valid Binary Search Tree\n")


def find_lowest_common_ancestor(tree, node1_index, node2_index):
    node1_ancestors = ancestor_indices(tree, node1_index)
    node2_ancestors = ancestor_indices(tree, node2_index)
    i = 0
    while i < len(node1_ancestors):
        if node1_ancestors[i] != node2_ancestors[i]:
            return node1_ancestors[i - 1]
        else:
            i += 1
    return node1_ancestors[i - 1]


print("Problem #3")
check_if_CBT([8, 3, 9, None, None, 4, 7])

print("Problem #4")
tree = [6, 2, 8, 1, 3, 7, 9]
for i in range(3):
    print "Write the index of two Nodes to find lowest common
ancestor"
    a = input("first node:")
    b = input("second node:")
    print tree[find_lowest_common_ancestor(tree, a, b)]
    print '\n'
```

Problem #5

```
class rbt_node:
    def __init__(self, value):
        self.color = 0 #red
        self.value = value

        self.parent = None
        self.left = None
        self.right = None

    def insert_left(self, left):
        self.left = left
        left.parent = self

    def insert_right(self, right):
        self.right = right
        right.parent = self
```

```
    def swap(self, other_node):
        temp_value = other_node.value
        other_node.value = self.value
        self.value = temp_value

    def right_rotate(self):
        B = self
        A = self.left
        a = A.left
        b = A.right
        c = B.right

        B.left = a
        B.right = A
        A.left = b
        A.right = c

        if a is not None:
            a.parent = B
        if b is not None:
            b.parent = A
        if c is not None:
            c.parent = A

        B.swap(A)


    def left_rotate(self):
        A = self
        B = self.right
        b = B.left
        c = B.right
        a = A.left

        B.left = a
        B.right = b
        A.left = B
        A.right = c

        if a is not None:
            a.parent = B
        if b is not None:
            b.parent = B
        if c is not None:
            c.parent = A

        A.swap(B)

    def tree_insert(self, node):
        if node.value < self.value:
            if self.left is None:
                self.insert_left(node)
```

```
            else:
                self.left.tree_insert(node)
        if node.value > self.value:
            if self.right is None:
                self.insert_right(node)
            else:
                self.right.tree_insert(node)

    def uncle(self):
        if self.parent is None:
            return None
        parent = self.parent

        if parent.parent is None:
            return None
        grandparent = self.parent.parent

        if grandparent.left == parent:
            return grandparent.right
        else:
            return grandparent.left

    def insert(self, value):
        new_node = rbt_node(value)
        self.tree_insert(new_node)

        while new_node != self:
            if new_node.parent.color != 0:
                break

            if new_node.uncle() is not None:
                if new_node.uncle().color == 0:
                    new_node.parent.parent.color = 0
                    new_node.parent.color = 1
                    new_node.uncle().color = 1
                    new_node = new_node.parent.parent
                    continue

            if new_node == new_node.parent.left:
                if new_node.parent == new_node.parent.parent.left:
                    new_node.parent.parent.right_rotate()
                else:
                    new_node.parent.right_rotate()
            elif new_node == new_node.parent.right:
                if new_node.parent == new_node.parent.parent.left:
                    new_node.parent.left_rotate()
                else:
                    new_node.parent.parent.left_rotate()

        self.color = 1

    def printNode(self):
        left_print = None
```

```
        right_print = None
        if self.left is not None:
            left_print = self.left.printNode()
        if self.right is not None:
            right_print = self.right.printNode()

        return '['+str(left_print)+','+str(self.value)
+'-'+str(self.color)+','+str(right_print)+']'


root_node = rbt_node(41)
root_node.color = 1
root_node.insert(38)
root_node.insert(31)
root_node.insert(12)
root_node.insert(19)
root_node.insert(8)

print("Problem #5 (0 is red, 1 is black)")
print(root_node.printNode())
```