```python
import os
import time
import math
import random
import sys

class Colors:
    DEFAULT = '\033[0m'
    BLACK = '\033[30m'
    RED = '\033[31m'
    GREEN = '\033[32m'
    YELLOW = '\033[33m'
    BLUE = '\033[34m'
    PURPLE = '\033[35m'
    CYAN = '\033[36m'
    WHITE = '\033[37m'


class Node:
    def __init__(self, info, M=[]):
        self.p: Node = None  # parent
        self.l: Node = None  # left child
        self.r: Node = None  # right child
        self.h: int = 0  # height
        self.d: int = 0  # depth
        self.b = None  # balance
        self.i = info  # information
        self.M = M  # matrix

    def __str__(self):
        if self.l is None and self.r is None:
            print('<' + str(self.i) + '>' +
                  '(b:' + str(self.b) + ')', end='', flush=True)
        else:
            print(str(self.i) + '(b:' + str(self.b) + ')', end='', flush=True)
        print(Colors.DEFAULT)


class AVLTree:
    def __init__(self, all_info, balance, animate):
        self.root = None
        self.last = None
        self.animate = animate
        self.balance = balance
        self.check_ending_balance = False
        self.num_nodes = 0
        self.insertion_times = []
        self.deletion_times = []
        m1_size = math.ceil(math.sqrt(2**20))
        m2_size = math.ceil(math.sqrt(2**19 + 2**18))
        m3_size = math.ceil(math.sqrt(2**18 + 2**17))
        M1 = [[0 for _ in range(m1_size)] for _ in range(m1_size)]
        M2 = [[0 for _ in range(m2_size)] for _ in range(m2_size)]
        M3 = [[0 for _ in range(m3_size)] for _ in range(m3_size)]
        if animate == 0:
            block_print()
        for info in all_info:
            M = None
            if info % 3 == 0:
```

```python
                M = M1
            elif info % 3 == 1:
                M = M2
            elif info % 3 == 2:
                M = M3
            s1 = time.time()
            self.insert(Node(info, M), self.root)
            e1 = time.time()
            self.insertion_times.append(round(e1 - s1, 4))
            # time.sleep(0.25)
        self.last = None
        self.print_line(Colors.DEFAULT)
        self.update_balance(self.root, True)
        self.rebalance(self.root)
        if animate == 0:
            enable_print()
            print('Inserted all ' + str(len(array)) + ' values:')
            self.rev_order(self.root)
        print('Height of AVLTree: ' + str(self.height(self.root)))
        print('SEEING IF ONLINE-ALGORITHM WORKED... ', end='', flush=True)

        dir_path = os.path.dirname(__file__)
        output_file_path = os.path.join(dir_path, 'output.txt')
        with open(output_file_path, 'w') as f:
            i = 0
            f.write('average insertion time (s), average deletion time (s)\n')
            f.write(str(round((sum(self.insertion_times)/len(self.insertion_times)), 4)))
            f.write(', ')
            f.write(str(round((sum(self.deletion_times)/len(self.deletion_times)), 4)))
            f.write('\n\n')
            f.write('insertion times (s), deletions times (s)\n')
            while(i < len(self.insertion_times)):
                f.write(str(self.insertion_times[i]))
                f.write(',')
                if i < len(self.deletion_times):
                    f.write(' ' + str(self.deletion_times[i]))
                f.write('\n')
                i += 1


    def insert(self, node, root):
        if root == None:
            self.root = node
            self.last = node
            if animate >= 1:
                self.rev_order(self.root)
            if animate == 1:
                input()
        elif node.i == root.i:
            s2 = time.time()
            self.delete(root.i, root)
            e2 = time.time()
            self.deletion_times.append(round(e2 - s2, 4))
            node.d = 0
            self.insert(node, self.root)
        else:
```

```
            node.d += 1
            if node.i < root.i and self.num_nodes <= 50:
                if root.l is None:
                    self.num_nodes += 1
                    self.print_line(Colors.DEFAULT)
                    print('Inserting ' + Colors.GREEN + str(node.i) +
                          Colors.DEFAULT + ' as ' + str(root.i) + '\'s left child')
                    root.l = node
                    self.last = node
                    node.p = root
                    self.update_balance(node, False)
                    if animate >= 1:
                        self.rev_order(self.root)
                    if animate == 1:
                        input()
                else:
                    self.insert(node, root.l)
            elif root.i < node.i and self.num_nodes <= 50:
                if root.r is None:
                    self.num_nodes += 1
                    self.print_line(Colors.DEFAULT)
                    print('Inserting ' + Colors.GREEN + str(node.i) +
                          Colors.DEFAULT + ' as ' + str(root.i) + '\'s right child')
                    root.r = node
                    self.last = node
                    node.p = root
                    self.update_balance(node, False)
                    if animate >= 1:
                        self.rev_order(self.root)
                    if animate == 1:
                        input()
                else:
                    self.insert(node, root.r)


    def delete(self, info, node):
        if node != None:
            if node.i == info:
                self.num_nodes -= 1
                self.print_line(Colors.YELLOW)
                print('Deleting node: ' + str(node.i))
                if animate >= 1:
                    self.rev_order(self.root)
                print('* * *')
                root = None
                if node.l == None and node.r == None:
                    root = node.p
                    if root != None and root.l == node:
                        root.l = None
                    elif root != None and root.r == node:
                        root.r = None
                    elif root == None:  # only node in tree
                        self.root == None
                elif node.l != None and node.r != None:
                    """
                    If x has two children,
                        -find x's successor z [the leftmost node in the rightsubtree of x]
```

```
                        -replace x's contents with z's contents, and
                        -delete z.
                        (Note: z does not have a left child, but may have a right child)
                        [since z has at most one child, so we use case (1) or (2) to delete z]
                    """
                    lnode = self.left_most(node.r) # find left most node in right subtree of x
                    val = lnode.i
                    M = lnode.M
                    self.delete(val, self.root)
                    print('\tSwapping data from deleted node ' + str(val) + ' to ' + str(node.i))
                    node.i = val # data transfer
                    node.M = M # data transfer
                    self.update_balance(self.root, False)
                    self.update_depth(self.root)
                elif node.l != None or node.r != None:
                    root = node.p
                    if root != None and root.l == node:
                        root.l = node.l if node.l != None else node.r
                        root.l.p = root
                    elif root != None and root.r == node:
                        root.r = node.l if node.l != None else node.r
                        root.r.p = root
                    elif root == None:
                        self.root = node.l if node.l != None else node.r
                        self.root.p = None
                        root = self.root
                self.update_balance(root, True)
                self.update_depth(root)
                if animate >= 1:
                    self.rev_order(self.root)
            elif info < node.i:
                self.delete(info, node.l)
            elif node.i < info:
                self.delete(info, node.r)
        else:
            print('Could not find node')

    def update_balance(self, node, just_update):
        if node == None:
            return

        lh = self.height(node.l)
        rh = self.height(node.r)

        tempb = node.b  # used just to check if balance is correct at the end
        if lh != None and rh != None:
            node.b = lh - rh
        elif lh != None:
            node.b = lh
        elif rh != None:
            node.b = rh
        else:
            node.b = 0

        if just_update == False and (node.b > 1 or node.b < -1):
            self.rebalance(node)
```

```python
            self.update_balance(node, False)
            return
        else:
            self.update_balance(node.p, False)

    def height(self, root):
        if root == None:
            return -1
        if root != None:
            lh = self.height(root.l)
            rh = self.height(root.r)
            return max(lh, rh) + 1

    def left_rotation(self, root):
        if root.r == None:
            return
        self.print_line(Colors.PURPLE)
        print('\tLeft rotation from node: ' + str(root.i) + '\n')
        if animate >= 1:
            self.rev_order(root)
        print('\n\t* * *\n')
        new_root = root.r
        root.r = new_root.l

        if new_root.l != None:
            new_root.l.p = root
        new_root.p = root.p

        if root.p == None:
            self.root = new_root
            new_root.p = None
        else:
            if root.p.l == root:
                root.p.l = new_root
            elif root.p.r == root:
                root.p.r = new_root
        new_root.l = root
        root.p = new_root

        self.update_depth(new_root)
        self.update_balance(new_root, True)
        self.update_balance(new_root.l, True)
        self.update_balance(new_root.r, True)
        if animate >= 1:
            self.rev_order(new_root)
        self.print_line(Colors.PURPLE)
        if animate == 1:
            input()
        return root

    def right_rotation(self, root):
        if root.l == None:
            return
        self.print_line(Colors.RED)
        print('\tRight rotation from node: ' + str(root.i) + '\n')
        if animate >= 1:
            self.rev_order(root)
        print('\n\t* * *\n')
        new_root = root.l
        root.l = new_root.r
```

```

        if new_root.r != None:
            new_root.r.p = root
        new_root.p = root.p

        if root.p == None:
            self.root = new_root
            new_root.p = None
        else:
            if root.p.l == root:
                root.p.l = new_root
            elif root.p.r == root:
                root.p.r = new_root
        new_root.r = root
        root.p = new_root

        self.update_depth(new_root)
        self.update_balance(new_root, True)
        self.update_balance(new_root.l, True)
        self.update_balance(new_root.r, True)
        if animate >= 1:
            self.rev_order(new_root)
        self.print_line(Colors.RED)
        if animate == 1:
            input()
        return root

    def rebalance(self, node):
        if node.b > 1:
            if node.l.b < 0:
                print('LR Rotation Needed')
                self.right_rotation(self.left_rotation(node.l))
            else:
                self.right_rotation(node)
        elif node.b < -1:
            if node.r.b > 0:
                print('RL Rotation Needed')
                self.left_rotation(self.right_rotation(node.r))
            else:
                self.left_rotation(node)

    def point(self, n, left, right):
        if left != None:
            n.l = left
        if right != None:
            n.r = right

    def rev_order(self, node):
        if node is None:
            return
        self.rev_order(node.r)
        self.print_node(node)
        self.rev_order(node.l)

    def update_depth(self, node):
        if node is None:
            return
        self.update_depth(node.r)
        node.d = self.get_depth(node)
        self.update_depth(node.l)
```

```python

    def get_depth(self, node):
        if node == self.root:
            return 0
        else:
            return 1 + self.get_depth(node.p)

    def left_most(self, node):
        if node.l == None:
            return node
        else:
            return self.left_most(node.l)

    def print_node(self, node):
        for i in range(node.d):
            print('\t\t', end='', flush=True)
        if self.last is node:
            print(Colors.GREEN, end='', flush=True)
        else:
            print(Colors.CYAN, end='', flush=True)
        node.__str__()

    def print_line(self, color):
        rows, columns = os.popen('stty size', 'r').read().split()
        for _ in range(int(columns)):
            print(color + '-' + Colors.DEFAULT, end='', flush=True)
        print()

# Disable
def block_print():
    sys.stdout = open(os.devnull, 'w')

# Restore
def enable_print():
    sys.stdout = sys.__stdout__

print('Enter lower bound (0 for hw): ', end='', flush=True)
lb = int(input())
print('Enter upper bound (299 for hw): ', end='', flush=True)
ub = int(input())
print('Enter the amount of random integers ranging from ' + str(lb) + ' - ' + str(ub) + ' to insert (100000 for hw): ', end='', flush=True)
num = int(input())
print('No steps or animations (0),\nAnimate (1),\nJust steps (2)?: ', end='', flush=True)
animate = int(input())
array = []
#try:
#    array = random.sample(range(lb, ub), num)
#except ValueError:
#    print('Sample size exceeded population size.')
#array = [random.randint(lb,ub) for _ in range(num)]
array = [random.randint(lb,ub) for _ in range(num)]
if len(array) <= 100:
    print(array)
start = time.time()
AVLTree(array, True, animate)
end = time.time()
print("{0:.4f}".format(end - start) + ' seconds')

```