

```

1 import os
2 import time
3 import math
4 import random
5 import sys
6
7 class Colors:
8     DEFAULT = '\033[0m'
9     BLACK = '\033[30m'
10    RED = '\033[31m'
11    GREEN = '\033[32m'
12    YELLOW = '\033[33m'
13    BLUE = '\033[34m'
14    PURPLE = '\033[35m'
15    CYAN = '\033[36m'
16    WHITE = '\033[37m'
17
18
19 class Node:
20     def __init__(self, info, M=[]):
21         self.p: Node = None # parent
22         self.l: Node = None # left child
23         self.r: Node = None # right child
24         self.h: int = 0 # height
25         self.d: int = 0 # depth
26         self.b = None # balance
27         self.i = info # information
28         self.M = M # matrix
29
30     def __str__(self):
31         if self.l is None and self.r is None:
32             print('<' + str(self.i) + '>' +
33                 '(b:' + str(self.b) + ')', end='', flush=True)
34         else:
35             print(str(self.i) + '(b:' + str(self.b) + ')', end='',
36 flush=True)
37             print(Colors.DEFAULT)
38
39 class AVLTree:
40     def __init__(self, all_info, balance, animate):
41         self.root = None
42         self.last = None
43         self.animate = animate
44         self.balance = balance
45         self.check_ending_balance = False
46         self.num_nodes = 0
47         self.insertion_times = []
48         self.deletion_times = []
49         m1_size = math.ceil(math.sqrt(2**20))
50         m2_size = math.ceil(math.sqrt(2**19 + 2**18))
51         m3_size = math.ceil(math.sqrt(2**18 + 2**17))
52         M1 = [[0 for _ in range(m1_size)] for _ in range(m1_size)]
53         M2 = [[0 for _ in range(m2_size)] for _ in range(m2_size)]
54         M3 = [[0 for _ in range(m3_size)] for _ in range(m3_size)]
55         if animate == 0:
56             block_print()
57         for info in all_info:
58             M = None
59             if info % 3 == 0:

```

```

60         M = M1
61         elif info % 3 == 1:
62             M = M2
63         elif info % 3 == 2:
64             M = M3
65         s1 = time.time()
66         self.insert(Node(info, M), self.root)
67         e1 = time.time()
68         self.insertion_times.append(round(e1 - s1, 4))
69         # time.sleep(0.25)
70     self.last = None
71     self.print_line(Colors.DEFAULT)
72     self.update_balance(self.root, True)
73     self.rebalance(self.root)
74     if animate == 0:
75         enable_print()
76         print('Inserted all ' + str(len(array)) + ' values:')
77         self.rev_order(self.root)
78     print('Height of AVLTree: ' + str(self.height(self.root)))
79     print('SEEING IF ONLINE-ALGORITHM WORKED ...', end='', flush=True)
80
81     dir_path = os.path.dirname(__file__)
82     output_file_path = os.path.join(dir_path, 'output.txt')
83     with open(output_file_path, 'w') as f:
84         i = 0
85         f.write('average insertion time (s), average deletion time
(s)\n')
86
87         f.write(str(round((sum(self.insertion_times)/len(self.insertion_times)),
4)))
88         f.write(', ')
89
90         f.write(str(round((sum(self.deletion_times)/len(self.deletion_times)), 4)))
91         f.write('\n\n')
92         f.write('insertion times (s), deletions times (s)\n')
93         while(i < len(self.insertion_times)):
94             f.write(str(self.insertion_times[i]))
95             f.write(',')
96             if i < len(self.deletion_times):
97                 f.write(' ' + str(self.deletion_times[i]))
98             f.write('\n')
99             i += 1
100
101     def insert(self, node, root):
102         if root == None:
103             self.root = node
104             self.last = node
105             if animate >= 1:
106                 self.rev_order(self.root)
107             if animate == 1:
108                 input()
109         elif node.i == root.i:
110             s2 = time.time()
111             self.delete(root.i, root)
112             e2 = time.time()
113             self.deletion_times.append(round(e2 - s2, 4))
114             node.d = 0
115             self.insert(node, self.root)
116         else:

```

```

116         node.d += 1
117         if node.i < root.i and self.num_nodes ≤ 50:
118             if root.l is None:
119                 self.num_nodes += 1
120                 self.print_line(Colors.DEFAULT)
121                 print('Inserting ' + Colors.GREEN + str(node.i) +
122                     Colors.DEFAULT + ' as ' + str(root.i) + '\s left
child')
123                 root.l = node
124                 self.last = node
125                 node.p = root
126                 self.update_balance(node, False)
127                 if animate ≥ 1:
128                     self.rev_order(self.root)
129                 if animate = 1:
130                     input()
131             else:
132                 self.insert(node, root.l)
133         elif root.i < node.i and self.num_nodes ≤ 50:
134             if root.r is None:
135                 self.num_nodes += 1
136                 self.print_line(Colors.DEFAULT)
137                 print('Inserting ' + Colors.GREEN + str(node.i) +
138                     Colors.DEFAULT + ' as ' + str(root.i) + '\s right
child')
139                 root.r = node
140                 self.last = node
141                 node.p = root
142                 self.update_balance(node, False)
143                 if animate ≥ 1:
144                     self.rev_order(self.root)
145                 if animate = 1:
146                     input()
147             else:
148                 self.insert(node, root.r)
149
150
151     def delete(self, info, node):
152         if node ≠ None:
153             if node.i = info:
154                 self.num_nodes -= 1
155                 self.print_line(Colors.YELLOW)
156                 print('Deleting node: ' + str(node.i))
157                 if animate ≥ 1:
158                     self.rev_order(self.root)
159                 print('* * *')
160                 root = None
161                 if node.l = None and node.r = None:
162                     root = node.p
163                     if root ≠ None and root.l = node:
164                         root.l = None
165                     elif root ≠ None and root.r = node:
166                         root.r = None
167                     elif root = None: # only node in tree
168                         self.root = None
169                 elif node.l ≠ None and node.r ≠ None:
170                     """
171                     If x has two children,
172                     -find x's successor z [the leftmost node in the
rightsubtree of x]

```

```

173         -replace x's contents with z's contents, and
174         -delete z.
175         (Note: z does not have a left child, but may have a
right child)
176         [since z has at most one child, so we use case (1) or
(2) to delete z]
177         """
178         lnode = self.left_most(node.r) # find left most node in
right subtree of x
179         val = lnode.i
180         M = lnode.M
181         self.delete(val, self.root)
182         print('\tSwapping data from deleted node ' + str(val) + '
to ' + str(node.i))
183         node.i = val # data transfer
184         node.M = M # data transfer
185         self.update_balance(self.root, False)
186         self.update_depth(self.root)
187         elif node.l != None or node.r != None:
188             root = node.p
189             if root != None and root.l == node:
190                 root.l = node.l if node.l != None else node.r
191                 root.l.p = root
192             elif root != None and root.r == node:
193                 root.r = node.l if node.l != None else node.r
194                 root.r.p = root
195             elif root == None:
196                 self.root = node.l if node.l != None else node.r
197                 self.root.p = None
198                 root = self.root
199             self.update_balance(root, True)
200             self.update_depth(root)
201             if animate >= 1:
202                 self.rev_order(self.root)
203             elif info < node.i:
204                 self.delete(info, node.l)
205             elif node.i < info:
206                 self.delete(info, node.r)
207         else:
208             print('Could not find node')
209
210     def update_balance(self, node, just_update):
211         if node == None:
212             return
213
214         lh = self.height(node.l)
215         rh = self.height(node.r)
216
217         tempb = node.b # used just to check if balance is correct at the end
218         if lh != None and rh != None:
219             node.b = lh - rh
220         elif lh != None:
221             node.b = lh
222         elif rh != None:
223             node.b = rh
224         else:
225             node.b = 0
226
227         if just_update == False and (node.b > 1 or node.b < -1):
228             self.rebalance(node)

```

```
229         self.update_balance(node, False)
230         return
231     else:
232         self.update_balance(node.p, False)
233
234 def height(self, root):
235     if root == None:
236         return -1
237     if root != None:
238         lh = self.height(root.l)
239         rh = self.height(root.r)
240         return max(lh, rh) + 1
241
242 def left_rotation(self, root):
243     if root.r == None:
244         return
245     self.print_line(Colors.PURPLE)
246     print('\tLeft rotation from node: ' + str(root.i) + '\n')
247     if animate ≥ 1:
248         self.rev_order(root)
249     print('\n\t* * *\n')
250     new_root = root.r
251     root.r = new_root.l
252
253     if new_root.l != None:
254         new_root.l.p = root
255     new_root.p = root.p
256
257     if root.p == None:
258         self.root = new_root
259         new_root.p = None
260     else:
261         if root.p.l == root:
262             root.p.l = new_root
263         elif root.p.r == root:
264             root.p.r = new_root
265     new_root.l = root
266     root.p = new_root
267
268     self.update_depth(new_root)
269     self.update_balance(new_root, True)
270     self.update_balance(new_root.l, True)
271     self.update_balance(new_root.r, True)
272     if animate ≥ 1:
273         self.rev_order(new_root)
274     self.print_line(Colors.PURPLE)
275     if animate == 1:
276         input()
277     return root
278
279 def right_rotation(self, root):
280     if root.l == None:
281         return
282     self.print_line(Colors.RED)
283     print('\tRight rotation from node: ' + str(root.i) + '\n')
284     if animate ≥ 1:
285         self.rev_order(root)
286     print('\n\t* * *\n')
287     new_root = root.l
288     root.l = new_root.r
```

```
289
290     if new_root.r != None:
291         new_root.r.p = root
292     new_root.p = root.p
293
294     if root.p == None:
295         self.root = new_root
296         new_root.p = None
297     else:
298         if root.p.l == root:
299             root.p.l = new_root
300         elif root.p.r == root:
301             root.p.r = new_root
302     new_root.r = root
303     root.p = new_root
304
305     self.update_depth(new_root)
306     self.update_balance(new_root, True)
307     self.update_balance(new_root.l, True)
308     self.update_balance(new_root.r, True)
309     if animate >= 1:
310         self.rev_order(new_root)
311     self.print_line(Colors.RED)
312     if animate == 1:
313         input()
314     return root
315
316 def rebalance(self, node):
317     if node.b > 1:
318         if node.l.b < 0:
319             print('LR Rotation Needed')
320             self.right_rotation(self.left_rotation(node.l))
321         else:
322             self.right_rotation(node)
323     elif node.b < -1:
324         if node.r.b > 0:
325             print('RL Rotation Needed')
326             self.left_rotation(self.right_rotation(node.r))
327         else:
328             self.left_rotation(node)
329
330 def point(self, n, left, right):
331     if left != None:
332         n.l = left
333     if right != None:
334         n.r = right
335
336 def rev_order(self, node):
337     if node is None:
338         return
339     self.rev_order(node.r)
340     self.print_node(node)
341     self.rev_order(node.l)
342
343 def update_depth(self, node):
344     if node is None:
345         return
346     self.update_depth(node.r)
347     node.d = self.get_depth(node)
348     self.update_depth(node.l)
```

```

349
350     def get_depth(self, node):
351         if node == self.root:
352             return 0
353         else:
354             return 1 + self.get_depth(node.p)
355
356     def left_most(self, node):
357         if node.l == None:
358             return node
359         else:
360             return self.left_most(node.l)
361
362     def print_node(self, node):
363         for i in range(node.d):
364             print('\t\t', end='', flush=True)
365         if self.last is node:
366             print(Colors.GREEN, end='', flush=True)
367         else:
368             print(Colors.CYAN, end='', flush=True)
369         node.__str__()
370
371     def print_line(self, color):
372         rows, columns = os.popen('stty size', 'r').read().split()
373         for _ in range(int(columns)):
374             print(color + '-' + Colors.DEFAULT, end='', flush=True)
375         print()
376
377     # Disable
378     def block_print():
379         sys.stdout = open(os.devnull, 'w')
380
381     # Restore
382     def enable_print():
383         sys.stdout = sys.__stdout__
384
385     print('Enter lower bound (0 for hw): ', end='', flush=True)
386     lb = int(input())
387     print('Enter upper bound (299 for hw): ', end='', flush=True)
388     ub = int(input())
389     print('Enter the amount of random integers ranging from ' + str(lb) + ' - ' +
390           str(ub) + ' to insert (100000 for hw): ', end='', flush=True)
391     num = int(input())
392     print('No steps or animations (0),\nAnimate (1),\nJust steps (2)? ', end='',
393           flush=True)
394     animate = int(input())
395     array = []
396     #try:
397     #    array = random.sample(range(lb, ub), num)
398     #except ValueError:
399     #    print('Sample size exceeded population size.')
400     #array = [random.randint(lb,ub) for _ in range(num)]
401     array = [random.randint(lb,ub) for _ in range(num)]
402     if len(array) ≤ 100:
403         print(array)
404     start = time.time()
405     AVLTree(array, True, animate)
406     end = time.time()
407     print("{0:.4f}".format(end - start) + ' seconds')

```