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

print(sys.version)


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, char, freq):
        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.char = char
        self.code = ''
        self.freq = freq # frequency

    def char_info(self):
        # TODO: add this functionality
        char = str(self.char) if self.char != '\n' else '\\n'
        return '\'' + char + '\'\t\t\t' + ' code: ' + str(self.code) +
'\t\t\t freq: ' + str(self.freq)

    def get_char(self):
        return str(self.char)

    def __str__(self):
        char = str(self.char) if self.char != '\n' else '\\n'
        if self.l is None and self.r is None:
            print('<' + char + '>' +
                  '(f:' + str(self.freq) + ')', end='', flush=True)
        else:
            print(char + '(f:' + str(self.freq) + ')', end='', flush=True)
        print(Colors.DEFAULT)


class HuffmanBinaryTree:
    def __init__(self, all_info, animate):
        self.root = None
        self.last = None
        self.error = False
        self.num_nodes = 0
        self.animate = animate
        if self.animate == 0:
```

```
60              block_print()
61          #for info in all_info:
62          #    self.insert(Node(info), self.root)
63              # time.sleep(0.25)
64          self.last = None
65          #if self.animate == 0:
66          #    self.enable_print()
67          #    print('Inserted all ' + str(len(array)) + ' values:')
68          #    self.rev_order(self.root)
69
70      def insert(self, root, rootl, rootr):
71          if self.root == None:
72              self.root = root
73              self.point(root, rootl, rootr)
74              self.update_balance(rootl)
75              self.update_balance(rootr)
76              self.update_depth(self.root)
77              self.get_codes(self.root)
78          else:
79              self.root = root
80              self.point(root, rootl, rootr)
81              self.update_balance(rootl)
82              self.update_balance(rootr)
83              self.update_depth(self.root)
84              self.get_codes(self.root)
85
86      def update_balance(self, node):
87          if node == None:
88              return
89
90          lh = self.height(node.l)
91          rh = self.height(node.r)
92
93          tempb = node.b  # used just to check if balance is correct at the end
94          if lh != None and rh != None:
95              node.b = lh - rh
96          elif lh != None:
97              node.b = lh
98          elif rh != None:
99              node.b = rh
100         else:
101             node.b = 0
102
103         self.update_balance(node.p)
104
105     def height(self, root):
106         if root == None:
107             return -1
108         if root != None:
109             lh = self.height(root.l)
110             rh = self.height(root.r)
111             return max(lh, rh) + 1
112
113     def point(self, root, left, right):
114         if left != None:
115             root.l = left
116             left.p = root
117         if right != None:
118             root.r = right
119             right.p = root
```

```
120
121     def rev_order(self, node):
122         if node is None:
123             return
124         self.rev_order(node.r)
125         self.print_node(node)
126         self.rev_order(node.l)
127
128     def update_depth(self, node):
129         if node is None:
130             return
131         self.update_depth(node.r)
132         node.d = self.get_depth(node)
133         self.update_depth(node.l)
134
135     def get_depth(self, node):
136         if node == self.root:
137             return 0
138         else:
139             return 1 + self.get_depth(node.p)
140
141     def print_tree(self):
142         self.rev_order(self.root)
143
144     def is_empty(self):
145         if self.root == None:
146             return True
147         else:
148             return False
149
150     def get_root(self):
151         return self.root
152
153     def print_node(self, node):
154         for i in range(node.d):
155             print('\t\t', end='', flush=True)
156         if self.last is node:
157             print(Colors.GREEN, end='', flush=True)
158         else:
159             print(Colors.CYAN, end='', flush=True)
160         node.__str__()
161
162     def print_line(self, color):
163         rows, columns = os.popen('stty size', 'r').read().split()
164         for _ in range(int(columns)):
165             print(color + '-' + Colors.DEFAULT, end='', flush=True)
166         print()
167
168     def get_codes(self, node):
169         if node is None:
170             return
171         self.get_codes(node.r)
172         if node.l == None and node.r == None:
173             node.code = str(self.get_code(node))[::-1]
174         self.get_codes(node.l)
175
176     def get_code(self, node):
177         if node.p != None and node.p.l == node:
178             return '0' + str(self.get_code(node.p))
179         elif node.p != None and node.p.r == node:
```

```
180             return '1' + str(self.get_code(node.p))
181         else:
182             return ''
183 
184     def find_char(self, node, code):
185         if node.l == None and node.r == None:
186             return node.get_char()
187         elif code[0] == '0':
188             if node != None:
189                 return self.find_char(node.l, code[1:])
190         elif code[0] == '1':
191             if node != None:
192                 return self.find_char(node.r, code[1:])
193 
194 
195 # Disable
196 def block_print():
197     sys.stdout = open(os.devnull, 'w')
198 
199 # Restore
200 def enable_print():
201     sys.stdout = sys.__stdout__
202 
```