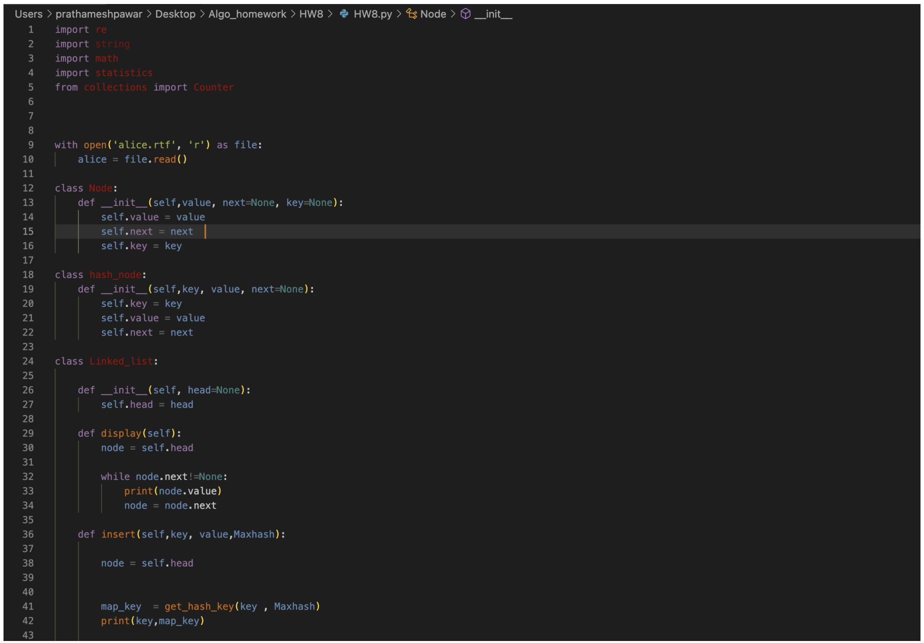
# HW8

Prathamesh Pawar

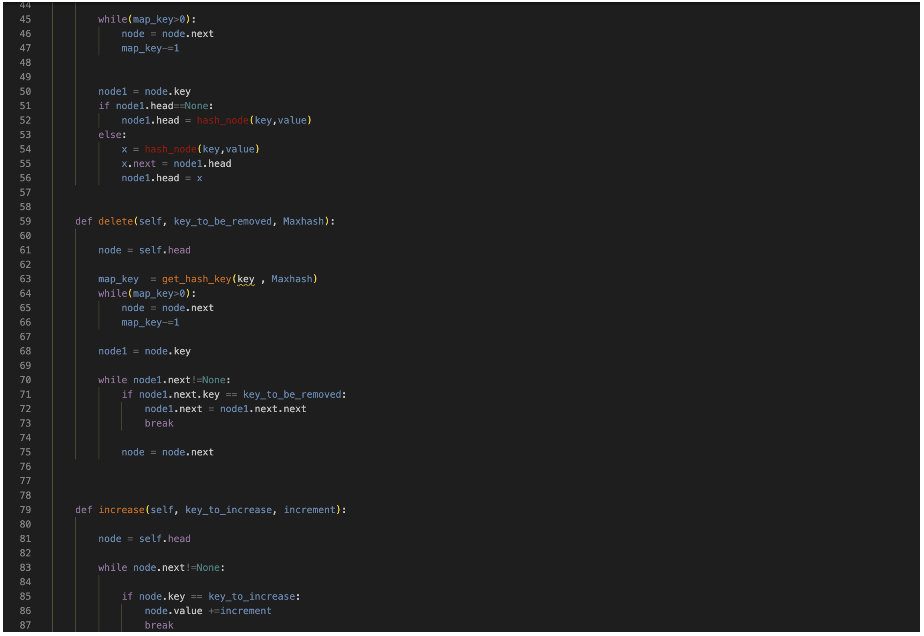
March 2023

Sources: stackoverflow, GeeksforGeeks

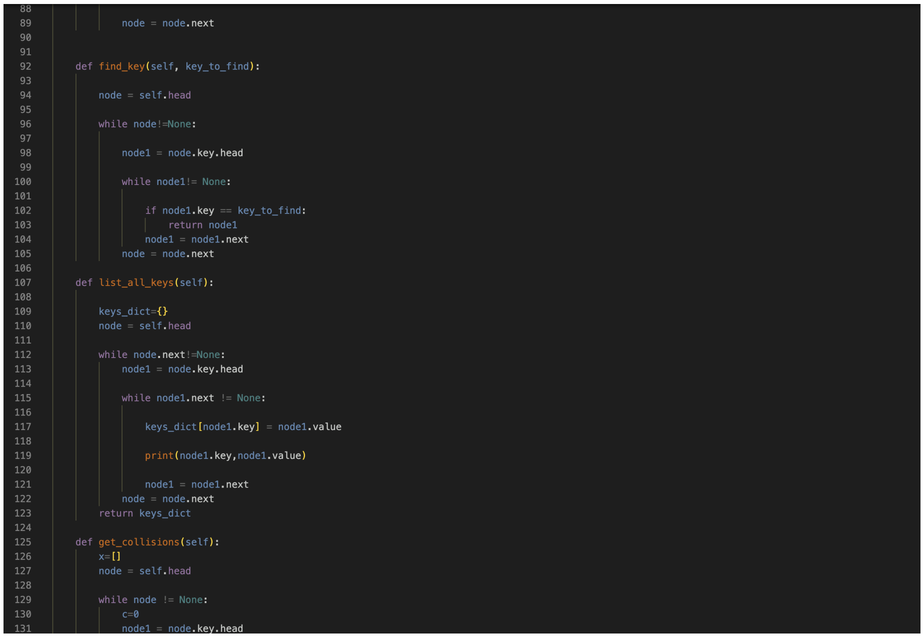
# 1 Problem 1: Hash Function


Users > prathameshpawar > Desktop > Algo_homework > HW8 > HW8.py > Node > __init__

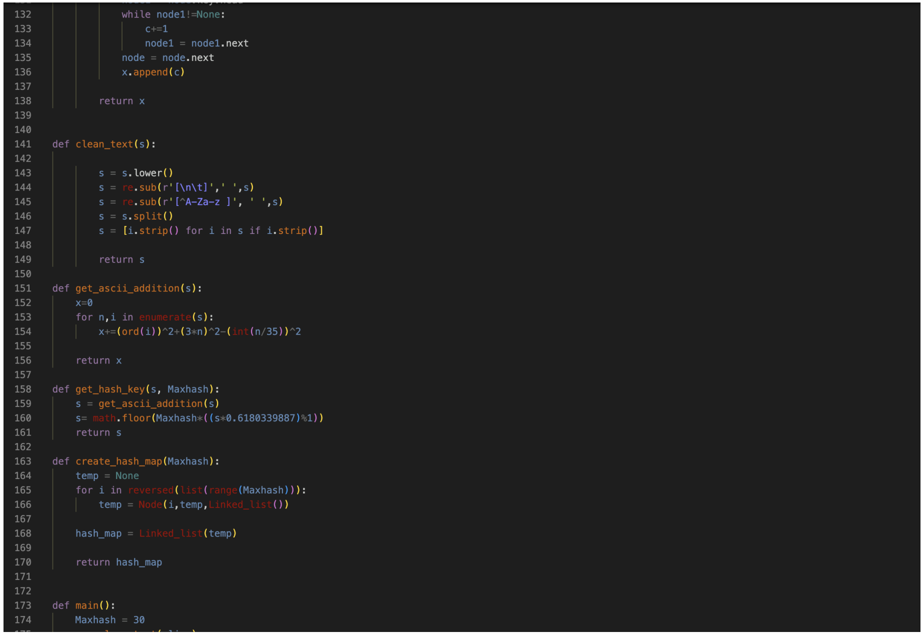
```python
import re
import string
import math
import statistics
from collections import Counter



with open('alice.rtf', 'r') as file:
    alice = file.read()

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

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

class Linked_list:

    def __init__(self, head=None):
        self.head = head

    def display(self):
        node = self.head

        while node.next!=None:
            print(node.value)
            node = node.next

    def insert(self,key, value,Maxhash):

        node = self.head


        map_key  = get_hash_key(key , Maxhash)
        print(key,map_key)

```



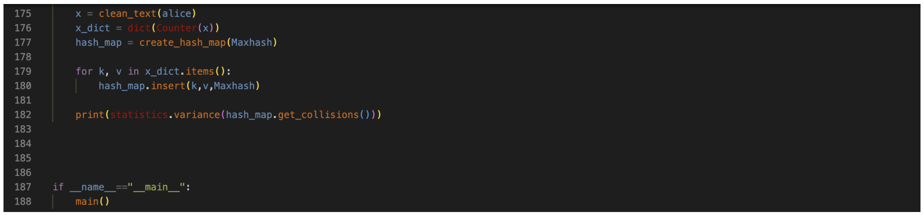
```python

        while(map_key>0):
            node = node.next
            map_key-=1


        node1 = node.key
        if node1.head==None:
            node1.head = hash_node(key,value)
        else:
            x = hash_node(key,value)
            x.next = node1.head
            node1.head = x


    def delete(self, key_to_be_removed, Maxhash):

        node = self.head

        map_key  = get_hash_key(key , Maxhash)
        while(map_key>0):
            node = node.next
            map_key-=1

        node1 = node.key

        while node1.next!=None:
            if node1.next.key == key_to_be_removed:
                node1.next = node1.next.next
                break

            node = node.next



    def increase(self, key_to_increase, increment):

        node = self.head

        while node.next!=None:

            if node.key == key_to_increase:
                node.value +=increment
                break
```



```
            node = node.next


    def find_key(self, key_to_find):

        node = self.head

        while node!=None:

            node1 = node.key.head

            while node1!= None:

                if node1.key == key_to_find:
                    return node1
                node1 = node1.next
            node = node.next

    def list_all_keys(self):

        keys_dict={}
        node = self.head

        while node.next!=None:
            node1 = node.key.head

            while node1.next != None:

                keys_dict[node1.key] = node1.value

                print(node1.key,node1.value)

                node1 = node1.next
            node = node.next
        return keys_dict

    def get_collisions(self):
        x=[]
        node = self.head

        while node != None:
            c=0
            node1 = node.key.head
```

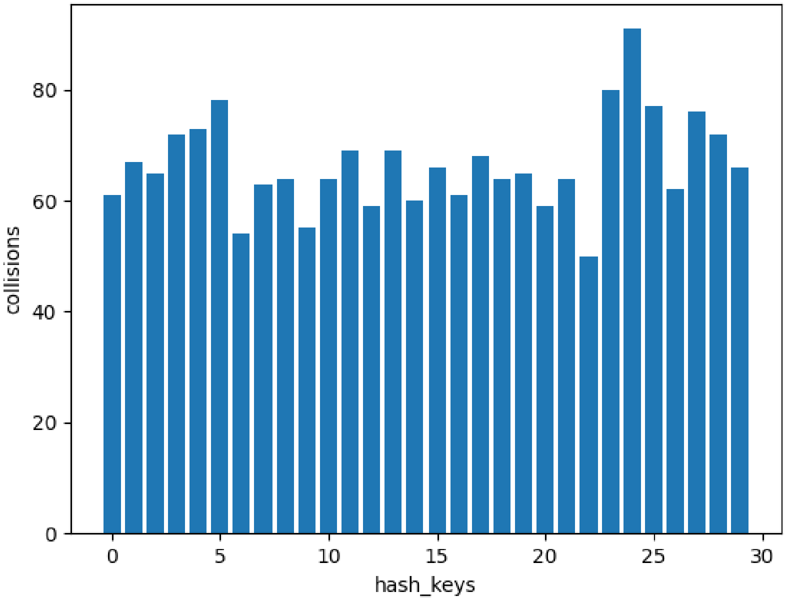


```
            while node1!=None:
                c+=1
                node1 = node1.next
            node = node.next
            x.append(c)

        return x


def clean_text(s):

        s = s.lower()
        s = re.sub(r'[\n\t]',' ',s)
        s = re.sub(r'[^A-Za-z ]', ' ',s)
        s = s.split()
        s = [i.strip() for i in s if i.strip()]

        return s

def get_ascii_addition(s):
    x=0
    for n,i in enumerate(s):
        x+=(ord(i))^2+(3*n)^2-(int(n/35))^2

    return x

def get_hash_key(s, Maxhash):
    s = get_ascii_addition(s)
    s= math.floor(Maxhash*((s*0.6180339887)%1))
    return s

def create_hash_map(Maxhash):
    temp = None
    for i in reversed(list(range(Maxhash))):
        temp = Node(i,temp,Linked_list())

    hash_map = Linked_list(temp)

    return hash_map


def main():
    Maxhash = 30
```



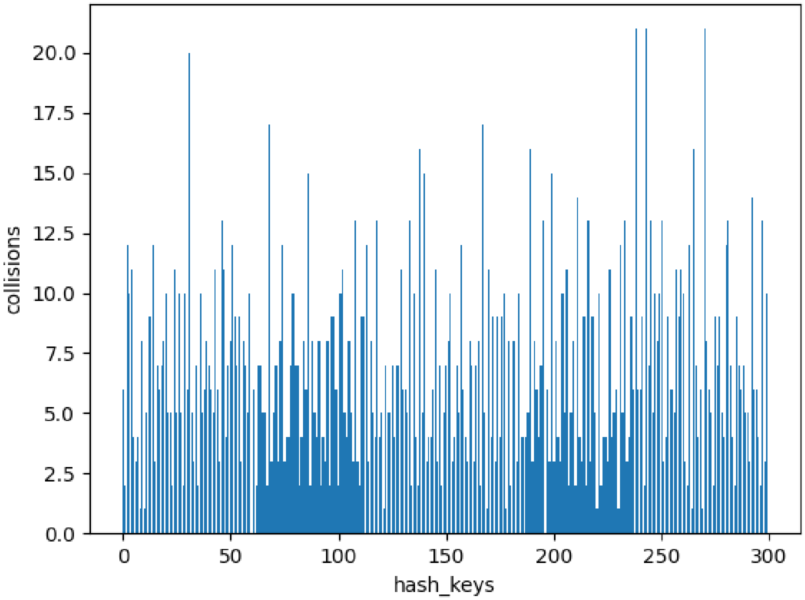
```
    x = clean_text(alice)
    x_dict = dict(Counter(x))
    hash_map = create_hash_map(Maxhash)

    for k, v in x_dict.items():
        hash_map.insert(k,v,Maxhash)

    print(statistics.variance(hash_map.get_collisions()))




if __name__=="__main__":
    main()
```

## 1.1 Results 1:



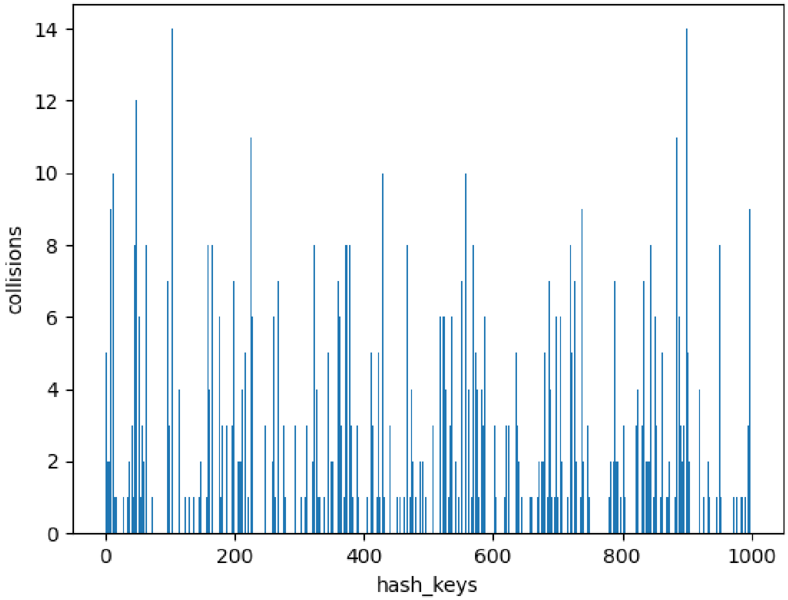
Variance = 71.4

## 1.2 Results 2:



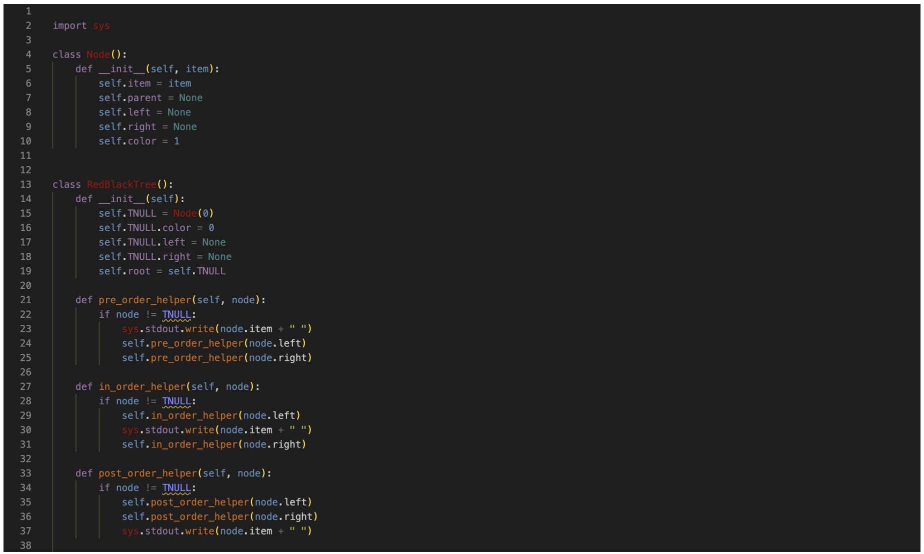
Variance = 16.08

### 1.3 Results 3:

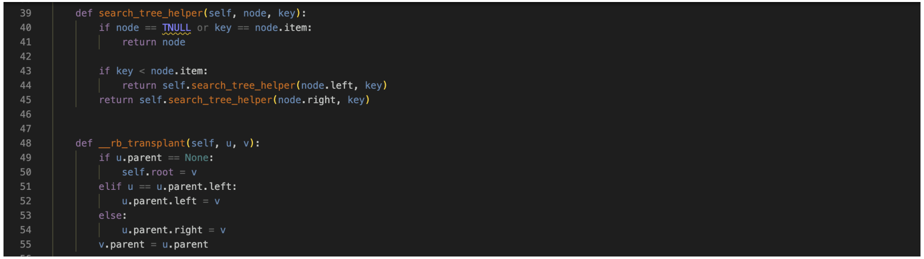


Variance = 5.70

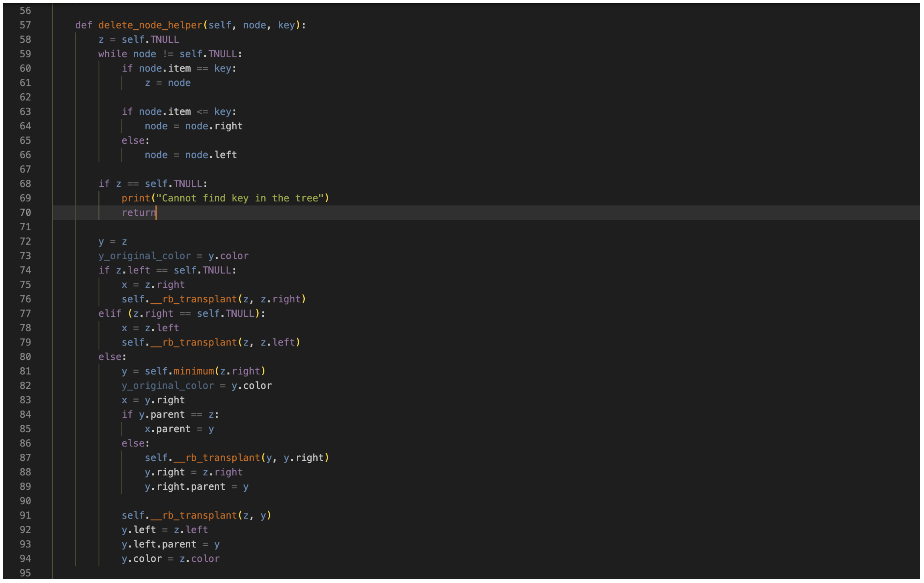
## 2 Problem 2: RB Tree



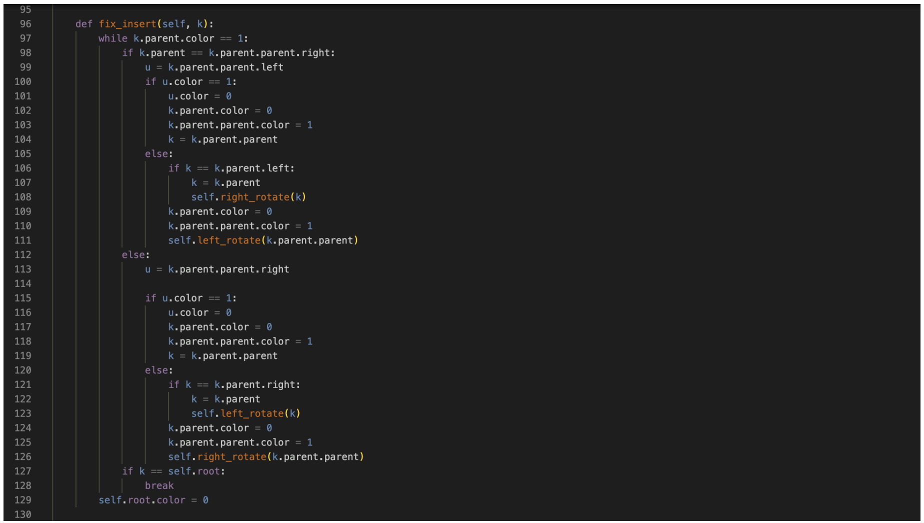
```python

import sys

class Node():
    def __init__(self, item):
        self.item = item
        self.parent = None
        self.left = None
        self.right = None
        self.color = 1


class RedBlackTree():
    def __init__(self):
        self.TNULL = Node(0)
        self.TNULL.color = 0
        self.TNULL.left = None
        self.TNULL.right = None
        self.root = self.TNULL

    def pre_order_helper(self, node):
        if node != TNULL:
            sys.stdout.write(node.item + " ")
            self.pre_order_helper(node.left)
            self.pre_order_helper(node.right)

    def in_order_helper(self, node):
        if node != TNULL:
            self.in_order_helper(node.left)
            sys.stdout.write(node.item + " ")
            self.in_order_helper(node.right)

    def post_order_helper(self, node):
        if node != TNULL:
            self.post_order_helper(node.left)
            self.post_order_helper(node.right)
            sys.stdout.write(node.item + " ")

```



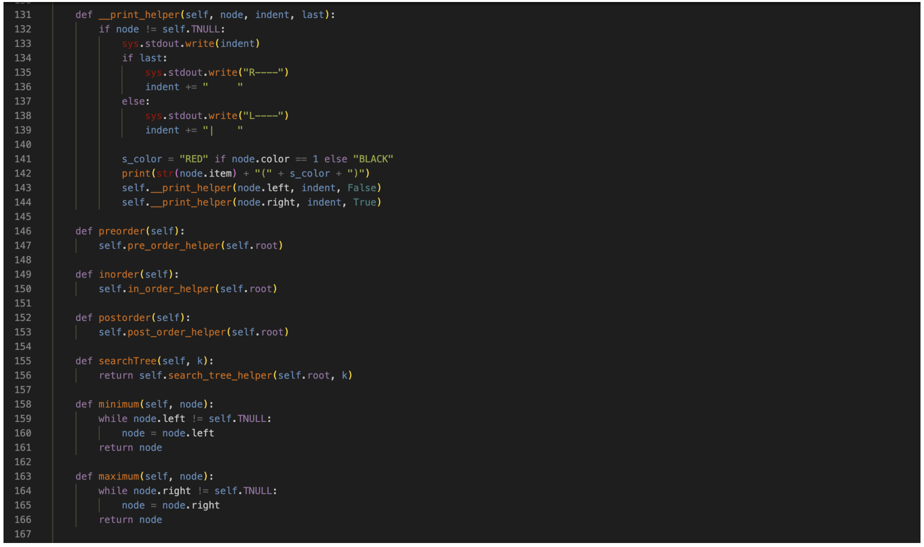
```python
    def search_tree_helper(self, node, key):
        if node == TNULL or key == node.item:
            return node

        if key < node.item:
            return self.search_tree_helper(node.left, key)
        return self.search_tree_helper(node.right, key)


    def __rb_transplant(self, u, v):
        if u.parent == None:
            self.root = v
        elif u == u.parent.left:
            u.parent.left = v
        else:
            u.parent.right = v
        v.parent = u.parent
```



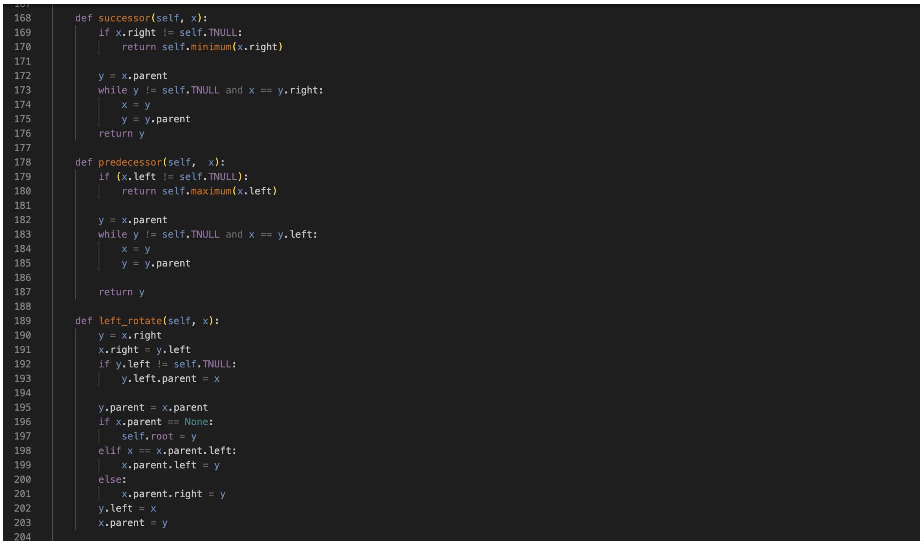
```python

    def delete_node_helper(self, node, key):
        z = self.TNULL
        while node != self.TNULL:
            if node.item == key:
                z = node

            if node.item <= key:
                node = node.right
            else:
                node = node.left

        if z == self.TNULL:
            print("Cannot find key in the tree")
            return

        y = z
        y_original_color = y.color
        if z.left == self.TNULL:
            x = z.right
            self.__rb_transplant(z, z.right)
        elif (z.right == self.TNULL):
            x = z.left
            self.__rb_transplant(z, z.left)
        else:
            y = self.minimum(z.right)
            y_original_color = y.color
            x = y.right
            if y.parent == z:
                x.parent = y
            else:
                self.__rb_transplant(y, y.right)
                y.right = z.right
                y.right.parent = y

            self.__rb_transplant(z, y)
            y.left = z.left
            y.left.parent = y
            y.color = z.color

```



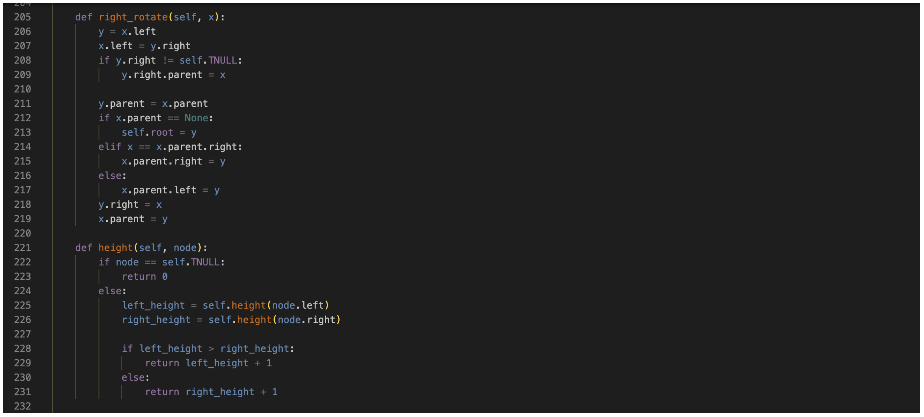
```
    def fix_insert(self, k):
        while k.parent.color == 1:
            if k.parent == k.parent.parent.right:
                u = k.parent.parent.left
                if u.color == 1:
                    u.color = 0
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    k = k.parent.parent
                else:
                    if k == k.parent.left:
                        k = k.parent
                        self.right_rotate(k)
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    self.left_rotate(k.parent.parent)
            else:
                u = k.parent.parent.right

                if u.color == 1:
                    u.color = 0
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    k = k.parent.parent
                else:
                    if k == k.parent.right:
                        k = k.parent
                        self.left_rotate(k)
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    self.right_rotate(k.parent.parent)
            if k == self.root:
                break
        self.root.color = 0

```



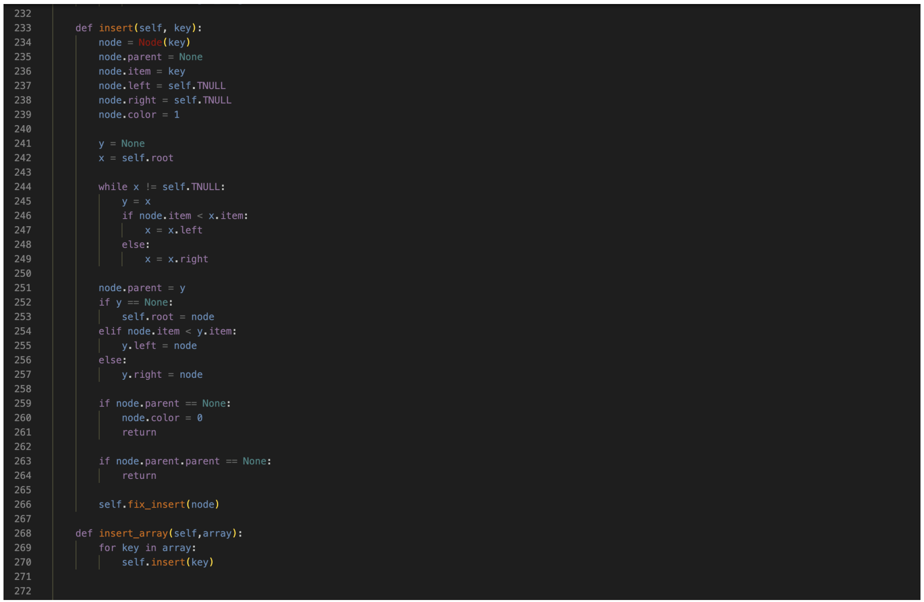
```
    def __print_helper(self, node, indent, last):
        if node != self.TNULL:
            sys.stdout.write(indent)
            if last:
                sys.stdout.write("R----")
                indent += "     "
            else:
                sys.stdout.write("L----")
                indent += "|    "

            s_color = "RED" if node.color == 1 else "BLACK"
            print(str(node.item) + "(" + s_color + ")")
            self.__print_helper(node.left, indent, False)
            self.__print_helper(node.right, indent, True)

    def preorder(self):
        self.pre_order_helper(self.root)

    def inorder(self):
        self.in_order_helper(self.root)

    def postorder(self):
        self.post_order_helper(self.root)

    def searchTree(self, k):
        return self.search_tree_helper(self.root, k)

    def minimum(self, node):
        while node.left != self.TNULL:
            node = node.left
        return node

    def maximum(self, node):
        while node.right != self.TNULL:
            node = node.right
        return node

```



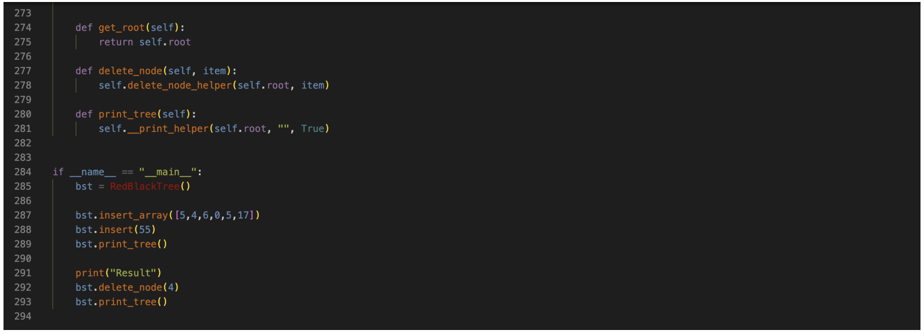
```
    def successor(self, x):
        if x.right != self.TNULL:
            return self.minimum(x.right)

        y = x.parent
        while y != self.TNULL and x == y.right:
            x = y
            y = y.parent
        return y

    def predecessor(self,  x):
        if (x.left != self.TNULL):
            return self.maximum(x.left)

        y = x.parent
        while y != self.TNULL and x == y.left:
            x = y
            y = y.parent

        return y

    def left_rotate(self, x):
        y = x.right
        x.right = y.left
        if y.left != self.TNULL:
            y.left.parent = x

        y.parent = x.parent
        if x.parent == None:
            self.root = y
        elif x == x.parent.left:
            x.parent.left = y
        else:
            x.parent.right = y
        y.left = x
        x.parent = y

```



```
    def right_rotate(self, x):
        y = x.left
        x.left = y.right
        if y.right != self.TNULL:
            y.right.parent = x

        y.parent = x.parent
        if x.parent == None:
            self.root = y
        elif x == x.parent.right:
            x.parent.right = y
        else:
            x.parent.left = y
        y.right = x
        x.parent = y

    def height(self, node):
        if node == self.TNULL:
            return 0
        else:
            left_height = self.height(node.left)
            right_height = self.height(node.right)

            if left_height > right_height:
                return left_height + 1
            else:
                return right_height + 1

```

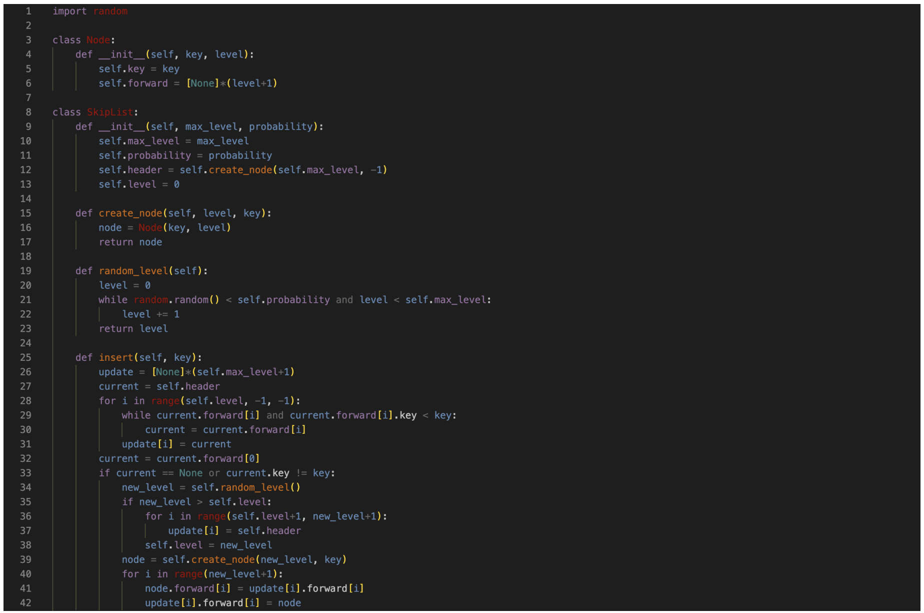


```python

    def insert(self, key):
        node = Node(key)
        node.parent = None
        node.item = key
        node.left = self.TNULL
        node.right = self.TNULL
        node.color = 1

        y = None
        x = self.root

        while x != self.TNULL:
            y = x
            if node.item < x.item:
                x = x.left
            else:
                x = x.right

        node.parent = y
        if y == None:
            self.root = node
        elif node.item < y.item:
            y.left = node
        else:
            y.right = node

        if node.parent == None:
            node.color = 0
            return

        if node.parent.parent == None:
            return

        self.fix_insert(node)

    def insert_array(self,array):
        for key in array:
            self.insert(key)


```

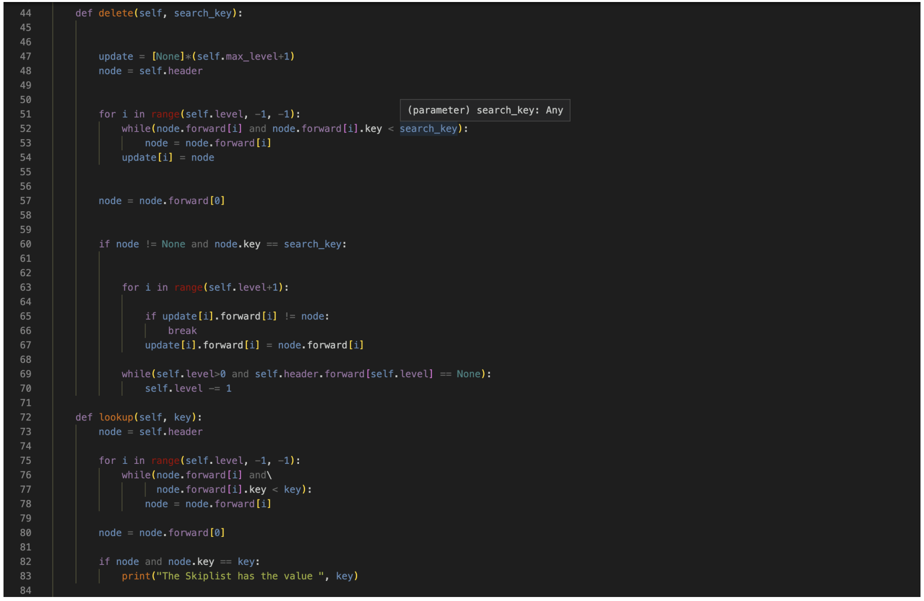


```python

    def get_root(self):
        return self.root

    def delete_node(self, item):
        self.delete_node_helper(self.root, item)

    def print_tree(self):
        self.__print_helper(self.root, "", True)


if __name__ == "__main__":
    bst = RedBlackTree()

    bst.insert_array([5,4,6,0,5,17])
    bst.insert(55)
    bst.print_tree()

    print("Result")
    bst.delete_node(4)
    bst.print_tree()

```

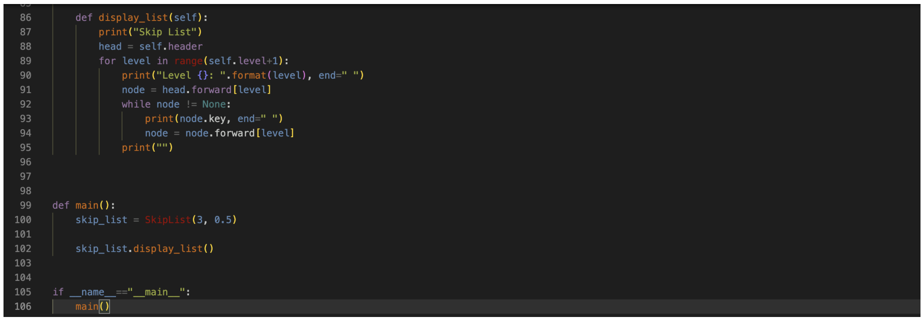
# 3 Problem 2: Skiplist



```python
import random

class Node:
    def __init__(self, key, level):
        self.key = key
        self.forward = [None]*(level+1)

class SkipList:
    def __init__(self, max_level, probability):
        self.max_level = max_level
        self.probability = probability
        self.header = self.create_node(self.max_level, -1)
        self.level = 0

    def create_node(self, level, key):
        node = Node(key, level)
        return node

    def random_level(self):
        level = 0
        while random.random() < self.probability and level < self.max_level:
            level += 1
        return level

    def insert(self, key):
        update = [None]*(self.max_level+1)
        current = self.header
        for i in range(self.level, -1, -1):
            while current.forward[i] and current.forward[i].key < key:
                current = current.forward[i]
            update[i] = current
        current = current.forward[0]
        if current == None or current.key != key:
            new_level = self.random_level()
            if new_level > self.level:
                for i in range(self.level+1, new_level+1):
                    update[i] = self.header
                self.level = new_level
            node = self.create_node(new_level, key)
            for i in range(new_level+1):
                node.forward[i] = update[i].forward[i]
                update[i].forward[i] = node
```



```python
    def delete(self, search_key):


        update = [None]*(self.max_level+1)
        node = self.header


        for i in range(self.level, -1, -1):
            while(node.forward[i] and node.forward[i].key < search_key):
                node = node.forward[i]
            update[i] = node


        node = node.forward[0]


        if node != None and node.key == search_key:


            for i in range(self.level+1):

                if update[i].forward[i] != node:
                    break
                update[i].forward[i] = node.forward[i]

            while(self.level>0 and self.header.forward[self.level] == None):
                self.level -= 1

    def lookup(self, key):
        node = self.header

        for i in range(self.level, -1, -1):
            while(node.forward[i] and\
              node.forward[i].key < key):
                node = node.forward[i]

        node = node.forward[0]

        if node and node.key == key:
            print("The Skiplist has the value ", key)

```



```python
    def display_list(self):
        print("Skip List")
        head = self.header
        for level in range(self.level+1):
            print("Level {}: ".format(level), end=" ")
            node = head.forward[level]
            while node != None:
                print(node.key, end=" ")
                node = node.forward[level]
            print("")



def main():
    skip_list = SkipList(3, 0.5)

    skip_list.display_list()


if __name__=="__main__":
    main()
```