Week 7
Red-Black Tree

Preliminary: Class lecture

Workshop: Application of Red-Black Tree class in program

Materials:

- RedBlackTree.py (Red-Black Tree Python class)
- BinarySearchTree.py (Binary Search Tree Python class)
- Test_program.py
- RBTree.pdf (the lecture note)

1) Examine the Python file that contains Binary Search Tree in order to understand how the class is to be used in a program.

   - For each Binary Search Tree operation, create a short code to test the provided function for such operation. You need to develop correct understanding of arguments defined for each of these functions.

2) Create a list of 16 integers arbitrarily. To enforce randomization, use "shuffle" function of "random" module to shuffle the order of these integers. If need, learn more here https://www.w3schools.com/python/ref_random_shuffle.asp .
   Insert these integers into a Binary Search Tree. Then, use the print_BSTree() function to view the resulted Binary Search Tree.

3) Create a loop that generates integers from a to b (about 10 integers) and insert them into a Binary Search Tree in the order they are generated. Then use the print_BSTree() function to view the resulted Binary Search Tree.

4) Redo step 3 but with n integers. After all the keys are inserted, use the following code to search the resulted Binary Search Tree. Add running time code as before to measure the running time of the given searching code.

   Try n = 1000, 2000, 4000, and 8000. Observe the running time.

```
import random

counter = 0
k = 2*n                          # n is the number of keys inserted
for i in range(n):
    v = random.randint(0, k)
    x = bst.Tree_Search(v)       # bst is the Binary Search Tree
    if x != None:
        counter += 1
print(counter)
```

5) Estimate the upperbound on the running time of each search. T(n) = O(__n__)

6) Examine the Python file that contains Red-Black Tree in order to understand how the class is to be used in a program.

   - For each basic Red-Black Tree operation, create a short code to test the provided function for such operation. You need to develop correct understanding of arguments defined for each of these functions.
   - If the Red-Black Tree is created as a variable named rbt, the value that represents None for rbt is rbt.NULL.

7) Create a loop that generates integers from a to b (about 10 integers) and insert them into a Red-Black Tree in the order they are generated. Then use the print_RBTree() function to view the resulted Red-Black Tree.

8) Redo step 3 but with n integers. After all the keys are inserted, use the following code to search the resulted Red-Black Tree. Add running time code as before to measure the running time of the given searching code.

   Try n = 1000, 2000, 4000, and 8000. Observe the significantly improved running time.

```
import random

counter = 0
k = 2*n                          # n is the number of keys inserted
for i in range(n):
    v = random.randint(0, k)
    x = rbt.Tree_Search(v)       # rbt is the Red-Black Tree
    if x != rbt.NULL:
        counter += 1
print(counter)
```

9) Why does using Red-Black Tree is significantly faster than using Binary Search Tree?