Implémentation d'ABR en Python

Étude de code

Code de l'implémentation

```
class DuplicateKey(KeyError):
    pass


class BSTNode(object):
    """
    Implementation for the node of Binary Search Tree
    """

    def __init__(self, key, value, parent=None, left=None, right=None):
        """
        The class constructor

        @param key: the key of the node
        @param value: the value of the node
        @param parent: pointer to the parent node
        @param left: pointer to the left child
        @param right: pointer to the right child

        @return: None
        """
        self.key = key
        self.value = value
        self.parent = parent
        self.left = left
        self.right = right


class BinarySearchTree(object):
    """
    Implementation for Binary Search Trees
    """

    def __init__(self):
        self.root = None

    def find_recursive(self, node, key):
        """
        Search the key from node, recursively

        @param node: a BST Node
        @param key: a key value
        @return: the node with the key; ``None`` if the key is not found
        """
        if None == node or key == node.key:
            return node
        elif key < node.key:
            return self.find_recursive(node.left, key)
        else:
            return self.find_recursive(node.right, key)
```

```
def find_iterative(self, node, key):
        """
        Search the key from node, iteratively

        @param node: a BST Node
        @param key: a key value
        @return: the node with the key; None if the key is not found
        """
        current_node = node
        while current_node:
            if key == current_node.key:
                return current_node
            if key < current_node.key:
                current_node = current_node.left
            else:
                current_node = current_node.right
        return None

    def search(self, key):
        """
        Find the node with the key

        @param key: the target key
        @return: the node with the key; None if the key is not found
        """
        return self.find_iterative(self.root, key)

    def insert(self, key, value):
        """
        Insert the (key, value) to the BST

        @param key: the key to insert
        @param value: the value to insert
        @return: True if insert successfully; otherwise return False
        """
        if None == self.root:
            self.root = BSTNode(key, value)
            return True

        current_node = self.root
        while current_node:
            if key == current_node.key:
                raise DuplicateKey
            elif key < current_node.key:
                if current_node.left:
                    current_node = current_node.left
                else:
                    current_node.left = BSTNode(key, value, current_node)
                    return True
            else:
                if current_node.right:
                    current_node = current_node.right
                else:
                    current_node.right = BSTNode(key, value, current_node)
                    return True
```

```
def replace_node(self, node, new_node):
        """
        Replace the node by new_node, update in its parent node

        @param node: node to replace
        @param new_node: the new node
        @return: None
        """
        # special case: replace the root
        if node == self.root:
            self.root = new_node
            return
        parent = node.parent
        if new_node:
            new_node.parent = parent
        if parent.left and parent.left == node:
            parent.left = new_node
        elif parent.right and parent.right == node:
            parent.right = new_node
        # else:
        #     raise RuntimeError("Incorrect parent-children relation!")

    def remove_node(self, node):
        """
        Remove the node from the tree

        @param node: the node to remove
        @return: None
        """
        if node.left and node.right:    # the node has two children
            # Find its in-order successor
            successor = node.right
            while successor.left:
                successor = successor.left
            # Copy the node
            node.key = successor.key
            node.value = successor.value
            # Remove the successor
            self.remove_node(successor)
        elif node.left:     # the node only has a left child
            self.replace_node(node, node.left)
        elif node.right:    # the node only has a right child
            self.replace_node(node, node.right)
        else:               # the node has no children
            self.replace_node(node, None)

    def delete(self, key):
        """
        Delete the node with the key

        @param key: a key value
        @return: The key deleted, None if key does not exist
        """
        node = self.search(key)
        deleted_key = None
        if node:
            deleted_key = node.key
            self.remove_node(node)
        return deleted_key
```

Tests unitaires

```
import unittest
import random

from bst import BinarySearchTree

class TestBinarySearchTree(unittest.TestCase):
    def setUp(self):
        '''Create new sequence and search tree.'''
        self.bst = BinarySearchTree()
        self.seq = list(range(1, 1000))


    def testFiewKeys(self):
        keys_to_insert = [4,2,5,1,3]
        keys_to_insert = list(range(1,10))
        random.shuffle(keys_to_insert)
        for key in keys_to_insert:
            self.bst.insert(key, key)

        for key in keys_to_insert:
            deleted_node_key = self.bst.delete(key)
            self.assertEqual(deleted_node_key, key)
            node_should_be_none = self.bst.search(key)
            self.assertEqual(node_should_be_none, None)
        self.assertEqual(self.bst.root, None)

    def testRandom(self):
        '''Inserts, finds, and deletes on a random sequence.'''

        random.shuffle(self.seq)
        for a in self.seq:
            self.bst.insert(a, a)
            self.assertEqual(self.bst.search(a).value, a)

        # find each, delete it, and make sure they are removed
        random.shuffle(self.seq)
        for a in self.seq:
            node_to_delete = self.bst.search(a)
            self.assertEqual(node_to_delete.key, a)
            deleted_node_key = self.bst.delete(a)
            self.assertEqual(deleted_node_key, a)
            node_should_be_none = self.bst.search(a)
            self.assertEqual(node_should_be_none, None)
        self.assertEqual(self.bst.root, None)

        random.shuffle(self.seq)
        for a in self.seq:
            self.assertEqual(self.bst.delete(a), None)


if __name__ == '__main__':
    unittest.main()
```