Submission Instructions Please submit a single python file (.py) containing the class RedBlackBST. If you would like to include any test code, please include at bottom using:

```
if __name__ == "__main__":
```

This allows the class to be imported for grading/testing without executing your tests. Also, please refrain from including print statements in your submission.

Finally, please name the file as follows:

```
RedBlackBST.py
```

Implement Insert using Iteration (not recursion) (50 points) The sample code provided for RedBlackTrees in lecture uses recursion to perform the insert() function. Your assignment is to implement the same functionality, but without the use of recursion. To do so effectively, you should maintain a parent pointer in your nodes, so that you may traverse back up the tree. Maintaining a parent requires an update to the provide _rotate_right() and _rotate_left() methods as well, as parents will have changed during the rotation. If you successfully manage parents in the rotation functions, and set the parent correctly when inserting the new node into the base of the tree, you should then be able to traverse back up the tree using $curr = curr.parent$ and assess the current node for the left rotation, right rotation and color flip scenarios until curr.parent is None (you're at the root).

Description of the provided code

The RedBlackBST class provided for this assignment includes the following methods for reference and testing:

```
def insert_r(self, key, value):
""" A functioning recursive implementation of insert.  Does not update or maintain parent
links for nodes.  It is used for student reference and testing.  It will not be included
as part of grading, and can be optionally omitted from the submission. """

def _put(self, node, key, value):
""" The recursive protected method used by insert_r to insert nodes recursively into the
data structure. It is used for student reference and testing.  It will not be included as
part of grading, and can be optionally omitted from the submission. """

def _rotate_left_r(self, node):
""" A functioning left rotation that is used by insert_r and _put methods.  It does NOT
maintain links to parent nodes and will thus not work as is for the iterative implementation.
This should be used for student reference and testing.  It will not be included as part of
grading, and can be optionally omitted from the submission. """

def _rotate_right_r(self, node):
""" A functioning right rotation that is used by insert_r and _put methods.  It does NOT
maintain links to parent nodes and will thus not work as is for the iterative implementation.
This should be used for student reference and testing.  It will not be included as part of
grading, and can be optionally omitted from the submission. """
```

The RedBlackBST class provided for this assignment includes the following shell methods and classes that can be used as is:

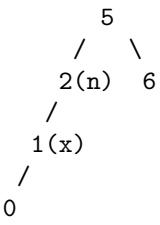
```
class RedBlackNode:
""" This subclass comes prepared with a standard constructor and support for a parent link.

Inside of RedBlackBST:
```

```
def _flip_colors(self, node):
""" The same method that works for the recursive implementation also works as is for the
iterative implementation."""

def search(self, key):
""" Regardless of the insert implementation, the search method is the same.
This can be used as is for testing."""

def _node_search(self, key):
""" Supports search().  returns the RedBlackNode object that contains the provided key."""

def _right_is_red(self, node):
""" Explanation in provided file. """

def _left_is_red(self, node):
""" Explanation in provided file. """

def _left_left_is_red(self, node):
""" Explanation in provided file. """

Testing Code:
def test_bst(bst):
""" Runs a few basic tests of the bst provided. """
```

Finally, below is the list of stub methods that have been included in the sample file for you to complete as a student:

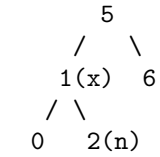
```
def insert_i(self, key, value):
""" Should produce the same results as insert_r but done so in an iterative, not
recursive manner.  The traversal down to insert, and the bubble up of rotations and
flips should be contained within this method. """

def _rotate_left_i(self, node):
""" Rotation left that includes the maintenance of parent links. """

def _rotate_right_i(self, node):
""" Rotation right that includes the maintenance of parent links. """
```

One final hint: When rotating, the new root node needs to be set as the original parent's .left node or .right node depending on whether the original child is smaller or larger than the parent. For example:



`n (node) and x are the variable names used in _rotate_left and _rotate_right`

Assume the links between 2-1 and 1-0 are both red links. We would want to _rotate_right(n). When doing so, the resulting structure will look like this:



This means that x.parent should be set to 5, but also that 5.left must be set to x. Do so using something akin to: "if n.parent.left is n then n.parent.left = x else n.parent.right = x"