1. Purpose

In this project you will create a Red Black Tree, which is a balanced binary search tree. In a Red Black tree the longest path from the root to a leaf cannot be more than twice of the shortest path from the root to a leaf. This means that the tree is always balanced and the operations are always O(lgn).

2. Properites

In order to build the tree, we need to understand the properties of Red Black Tree.

1. Every node is either red or black

2. Every leaf (NULL pointer) is black

3. If a node is red, both children are black

4. Every path from node to descendent leaf contains the same number of black nodes

5. The root is always black

3. Methods

- getSibling(RBNode): returns the sibling node of the parameter If the sibling does not exist, then return null.

```
public RedBlackTree.Node<String> getSibling(RedBlackTree.Node<String> node)
{
        if (isEnd(node.parent))
            return null;
        else if (node == node.parent.left)
            return node.parent.right;
        else
            return node.parent.left;
}
```

- getAunt(RBNode): returns the aunt of the parameter or the sibling of the parent node. If the aunt node does not exist, then return null.

```
public RedBlackTree.Node<String> getAunt(RedBlackTree.Node<String> node)
{
        if (!isEnd(getGrandparent(node)))
        {
            if (node.parent == getGrandparent(node).left)
                return getGrandparent(node).right;
            else
                return getGrandparent(node).left;
        }
        else
            return null;

    }
```

- getGrandparent(RBNode): Similar to getAunt() and getSibling(), returns the parent of your parent node, if it doesn't exist return null.

```
public RedBlackTree.Node<String> getGrandparent(RedBlackTree.Node<String> node)
{
        if (!isEndnode.parent))
        {
            if (!isEnd(node.parent.parent))
                return node.parent.parent;
            else
                return null;
        }
        else
            return null;

    }
```

- rotateLeft(RBNode) and rotateRight(RBNode) functions: left, resp. right, rotate around the node parameter.

```
private void leftRotate(RedBlackTree.Node<String> node)
{
        leftRotateFixup(node);
```

```
        RedBlackTree.Node<String> newNode;
        newNode = node.right;
        node.right = newNode.left;

        if (!isNil(newNode.left))
            newNode.left.parent = node;
        newNode.parent = node.parent;

        if (isNil(node.parent))
            root = newNode;
        else if (node.parent.left == node)
            node.parent.left = newNode;
        else
            node.parent.right = newNode;

        newNode.left = node;
        node.parent = newNode;
}
```

4. Conclusion

When I put the test case into the project, I got the correct answer which is "B A E C B D H G

I ".

```
B A E C B D H G I
Correct Tree: B A E C B D H G I
```

After run the Junit, I got this answer.

```
In order to insert the dictionary, we use: 178 ms.
So, there exist 324749 words in our dictionary.
In order to search these words, we use: 49 ms.
```

That shows when we put the dictionary into the project, we use 158ms and if we search words, we use 58ms.