**Name: Anubhav Shrivastava**

**USN: 1PI10CS020**

**Name: Arpan Madanmohan Gupta**

**USN: 1PI10CS040**

**Generic Programming Project**

Project Name – Red Black Tree

A red–black tree is a type of self-balancing binary search tree.

In addition to the requirements imposed on a binary search trees, with red–black trees:[4]

* A node is either red or black.
* The root is black. (This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice-versa, this rule has little effect on analysis.)
* All leaves (NIL) are black. (All leaves are same colour as the root.)
* Both children of every red node are black.
* Every simple path from a given node to any of its descendant leaves contains the same number of black nodes.

We have implemented this as a tree class which has a Node class as its private member for holding the structure of the Node and a public Iterator class for iterating through the tree.

1. Structure of a Node:

a)right link

b)left link

c)parent link : this link points to the parent. Is used during the inorder traversal of the tree using the overloaded ++ /-- operators in the Iterator class.

d)a generic “key” to hold data.

1. Functions in Tree
2. addNode1: The main insertion function, for adding nodes in the tree.
3. addNode : A wrapper around the main function addNode1.
4. Display1: To show an INORDER traversal of keys.
5. Display: a wrapper around Display1.
6. Iterator end(): gives an iterator pointing to the last element.
7. Iterator begin(): gives an iterator pointing to the first element of the tree.
8. EmptyTree: this function is called by the destructor when to Tree object goes out of scope. Its work is to traverse the whole tree and remove/ FREE all the allocated Node on of the tree.
9. Functions of iterator
10. Operator++: pre / post
11. Operator-- : pre/post
12. Operator==:equality
13. Operator\*:dereferencing for rhs
14. Operator!=: inequality

Procedure for insertion in a Red-Black Tree .

After successfully inserting a node as in binary tree fashion.

The following functions are used for balancing and to check the validity of properties of Red-Black Tree:

1. insertCase1:   
   The current node **N** is at the root of the tree. In this case, it is repainted black to satisfy property 2 (the root is black). Since this adds one black node to every path at once, property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is not violated.
2. insertCase2:

The current node's parent **P** is black, so property 4 (both children of every red node are black) is not invalidated. In this case, the tree is still valid. property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is not threatened, because the current node **N** has two black leaf children, but because **N** is red, the paths through each of its children have the same number of black nodes as the path through the leaf it replaced, which was black, and so this property remains satisfied.

1. insertCase3:

 If both the parent **P** and the uncle **U** are red, then both of them can be repainted black and the grandparent **G** becomes red (to maintain property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes)). Now, the current red node **N**has a black parent. Since any path through the parent or uncle must pass through the grandparent, the number of black nodes on these paths has not changed. However, the grandparent **G** may now violate properties 2 (The root is black) or 4 (Both children of every red node are black) (property 4 possibly being violated since **G** may have a red parent). To fix this, the entire procedure is recursively performed on **G** from case 1. Note that this is a tail-recursive call, so it could be rewritten as a loop; since this is the only loop, and any rotations occur after this loop, this proves that a constant number of rotations occur.

1. insertCase4:

The parent **P** is red but the uncle **U** is black; also, the current node **N** is the right child of **P**, and **P** in turn is the left child of its parent**G**. In this case, a [left rotation](http://en.wikipedia.org/wiki/Tree_rotation) that switches the roles of the current node **N** and its parent **P** can be performed; then, the former parent node **P** is dealt with using case 5 (relabeling **N** and **P**) because property 4 (both children of every red node are black) is still violated. The rotation causes some paths (those in the sub-tree labelled "1") to pass through the node **N** where they did not before. It also causes some paths (those in the sub-tree labelled "3") not to pass through the node **P** where they did before. However, both of these nodes are red, so property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is not violated by the rotation. After this case has been completed, property 4 (both children of every red node are black) is still violated, but now we can resolve this by continuing to case 5.

1. insertCase5:

The parent P is red but the uncle U is black; also, the current node N is the right child of P, and P in turn is the left child of its parentG. In this case, a left rotation that switches the roles of the current node N and its parent P can be performed; then, the former parent node P is dealt with using case 5 (relabeling N and P) because property 4 (both children of every red node are black) is still violated. The rotation causes some paths (those in the sub-tree labelled "1") to pass through the node N where they did not before. It also causes some paths (those in the sub-tree labelled "3") not to pass through the node P where they did before. However, both of these nodes are red, so property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is not violated by the rotation. After this case has been completed, property 4 (both children of every red node are black) is still violated, but now we can resolve this by continuing to case 5.