```
//============================================================
// BST.cpp
// This file contains excerpts from the Binary Search Tree
// class and the Red Black Tree class.
//============================================================

//============================================================
// BST
//============================================================

//============================================================
// insert
// inserts the value given as the parameter into the correct
// node in the bst.
// pre-condition: It is assumed that the bst is valid
// post-condition: The bst is valid after insertion
//============================================================
template <class KeyType>
void BST<KeyType>::insert (KeyType *k)
{
        Node<KeyType> *K = new Node<KeyType>;           //node to be
        K->data = k;                                                    //inserted
        K->left = NULL;
        K->right = NULL;
        Node<KeyType> *qtr = NULL;
        Node<KeyType> *ptr = root;
        while (ptr != NULL)
        {
                qtr = ptr;
                if (*(K->data) < *(ptr->data))
                        ptr = ptr->left;
                else
                        ptr = ptr->right;
        }
        if (qtr == NULL)
        {
                root = K;
        }
        else if (*(K->data) < *(qtr->data))
        {
                qtr->left = K;
        }
        else
                qtr->right = K;
}
//============================================================
// remove
// removes and deletes the node of the first instance of the
// value passed as a parameter
// pre-condition: the bst is valid
// post-condition: the bst is still valid after removal
//============================================================
template <class KeyType>
void BST<KeyType>::remove (const KeyType& k)
{
        Node<KeyType> *K = find(root, k);
        Node<KeyType> *parent = findParent(root, k);
        if (K->left == NULL && K->right ==NULL)                 //case 1:
        {                                                                       //no c
hildren
                if (*(parent->data) < *(K->data))
                        parent->right = NULL;
                else
```

```
                        parent->left = NULL;
                delete K;
        }
        else if (K->left == NULL && K->right != NULL) //case 2: one
        {                                                                       //chil
d (right)
                if (*(K->right->data) < *(parent->data))
                {
                        parent->left = K->right;
                        delete K;
                }
                else
                {
                        parent->right = K->right;
                        delete K;
                }
        }
        else if (K->left != NULL && K->right == NULL)   //one child
        {
// (left)
                if (*(K->left->data) < *(parent->data))
                {
                                parent->left = K->left;
                                delete K;
                }
                else
                {
                        parent->right = K->left;
                        delete K;
                }
        }
        else                            //case 3: two children
        {
                Node<KeyType> *succ = find(root, *successor(k));
                KeyType temp = *(succ->data);
                remove(*(succ->data));
                *(K->data) = temp;
        }
}

//============================================================
// maximum
// returns a KeyType pointer to the maximum element in the bst
//============================================================
template<class KeyType>
KeyType* BST<KeyType>::maximum (void) const
{
        Node<KeyType> *ptr = root;
        if (ptr == NULL)
        {
                cout << "Error: empty tree" << endl;
                //exit(1);
        }
        while (ptr->right != NULL)
                ptr = ptr->right;
        return ptr->data;
}
//============================================================
// minimum
// returns a KeyType pointer to the minimum element in the bst
//============================================================
template<class KeyType>
KeyType* BST<KeyType>::minimum (void) const
```

```
{
        Node<KeyType> *ptr = root;
        if (ptr == NULL)
        {
                cout << "Error: empty tree" << endl;
                //exit(1);
        }
        while (ptr->left != NULL)
    ptr = ptr->left;
  return ptr->data;
}
//================================================================
// successor
// returns a KeyType pointer to the successor of the first
// instance of the KeyType parameter value
//================================================================
template<class KeyType>
KeyType* BST<KeyType>::successor  (const KeyType& k) const
{
        Node<KeyType> *K = find(root, k);               //case 1: find
        if (K->right != NULL)   //smallest value in right subtree
        {
                K = K->right;
                while (K->left != NULL)
                {
                        K = K->left;
                }
                return K->data;
        }
        Node<KeyType> *parent = findParent(root, k);
        while (parent != NULL && K == parent->right)
        {
                K = parent;
                parent = findParent(root, *(parent->data));
        }
        return parent->data;
}
//================================================================
// predecessor
// returns a KeyType pointer to the predecessor of the first
// instance of the KeyType parameter value
//================================================================
template<class KeyType>
KeyType* BST<KeyType>::predecessor  (const KeyType& k) const
{
        Node<KeyType> *K = find(root, k);
        if (K->left != NULL)
        {
                K = K->left;
                while (K->right != NULL)
                {
                        K = K->right;
                }
                return K->data;
        }
        Node<KeyType> *parent = findParent(root, k);
        while (parent != NULL && K == parent->left)
        {
                K = parent;
                parent = findParent(root, *(parent->data));
        }
        return parent->data;
}
```

```
//=============================================================
// RBT
//=============================================================

template <class KeyType>
class Node
{
public:
        KeyType         *data;
        Node                    *left;
        Node                    *right;
        Node                    *p;
        string          color;

        Node            (void)                          //default constructor
        {
                data = NULL;
                left = NULL;
                right = NULL;
                p = NULL;
                color = "red";
        };
        Node            (KeyType *item) //constructor with item
        {
                data = item;
                left = NULL;
                right = NULL;
                p = NULL;
                color = "red";
        };
        Node            (KeyType *item, Node<KeyType>* nil)//nil pointer
        {                                                                                       //cons
tuctor
                data = item;
                left = nil;
                right = nil;
                p = nil;
                color = "red";
        };


};

template <class KeyType>
class RBT
{
protected:
        void    RBTFix                          (Node<KeyType> *current);
        void    LeftRotate                      (Node<KeyType> *z);
        void    RightRotate                     (Node<KeyType> *z);
        string  inOrderHelper   (Node<KeyType> *z,stringstream &s);
        string  preOrderHelper  (Node<KeyType> *z, stringstream &s);
        string  postOrderHelper (Node<KeyType> *z, stringstream &s);
        Node<KeyType>*  copy                                    (Node<KeyType> *z);
        void                                            clear           (Node<KeyType> *z);
        Node<KeyType>*  find    (Node<KeyType> *r, KeyType k) const;

public:

        Node<KeyType> * root;     // root pointer for red black tree
        Node<KeyType> * nil;    // null pointer for leaf nodes

                        RBT                     (void);
```

```
                ~RBT                            (void);
                        RBT             (const RBT<KeyType> & r);

    bool                Empty                   (void) const;
    KeyType             *get                    (const KeyType& k);
    void                insert                  (KeyType *k);
    void                remove                  (const KeyType& k);
    KeyType             *maximum                (void) const;
    KeyType             *minimum                (void) const;
    KeyType             *successor              (const KeyType& k)const;
    KeyType             *predecessor            (const KeyType& k)const;
    string              inOrder                 (void) const;
    string              preOrder                (void) const;
    string              postOrder               (void) const;

};


class EmptyError {};


//=============================================================
//insert
//inserts item into RBT
//Pre-Condition:
//Post-Condition:
//Parameters: KeyType *k - pointer to item to be inserted
//Return Value: void
//=============================================================
template <class KeyType>
void            RBT<KeyType>::insert    (KeyType *k)
{
        Node<KeyType> *parent = nil;
        Node<KeyType> *current = root;
        Node<KeyType> *newNode = new Node<KeyType>(k, nil);
        //newNode->left = nil;
        //newNode->right = nil;
        while (current != nil)          //find where to insert newNode
        {
                parent = current;
                if (*newNode->data > *current->data)
                {
                        current = current->right;
                }
                else
                {
                        current = current->left;
                }
        }
        if (current == nil)                     //newNode inserted at root
        {
                root = newNode;
        }
        if (current == current->p->left)        //newNode should be
        {                                                                       //inserted to
left
                parent->left = newNode;
        }
        else
        {
                parent->right = newNode;                //inserted to right
        }
```

```
        RBTFix(newNode);                //fix black heights and balance tree


}
//=============================================================
//LeftRotate
//rotates the node left
//Pre-Condition:
//Post-Condition:
//Parameters: Node *current - pointer to node to be rotated
//Return Value: void
//=============================================================
template <class KeyType>
void            RBT<KeyType>::LeftRotate(Node<KeyType> *z)
{
        Node<KeyType> y = z->right;     //sets y to z's right child
        z->right = y->left;             //sets z's right subtree to y's
        if (y->left != nil)                             //left subtree
        {
                y->left->p = z;         //sets y's left subtree parent to z
        }
        y->p = z->p;                            //sets y's parent to z's parent
        if (z->p == nil)                                                //case 1: z is root
        {
                root = y;
        }
        else if (z == z->p->left)               //case 2: z is left child
        {
                z->p->left = y;
        }
        else                                                    //case 3: z is right child
        {
                z->p->right = y;
        }
        y->left = z;                                                    //put z on y's left
        z->p = y;
}
//=============================================================
//RightRotate
//rotates node right
//Pre-Condition:
//Post-Condition:
//Parameters:  Node *current - pointer to node to be rotated
//Return Value: void
//=============================================================
template <class KeyType>
void            RBT<KeyType>::RightRotate(Node<KeyType> *z)
{
        Node<KeyType> y = z->left;              //sets y to z's left child
        z->left = y->right;             //sets y's right subtree to
        if (y->right != nil)            //z's left subtree
        {
                y->right->p = z;
        }
        y->p = z->p;
        if (z->p == nil)                                //case 1: z is root
        {
                root = y;
        }
        else if (z == z->p->left)               //case 2: z is left child
        {
                z->p->left = y;
        }
```

```
        else                                    //case 3: z is right child
        {
                z->p->right = y;
        }
        y->right = z;                           //put z on y's right
        z->p = y;
}
//=============================================================
//RBTFix
//"fixes" RBT to uphold RBT properties after an insert
//Pre-Condition:
//Post-Condition:
//Parameters: Node *current - pointer to node to be rotated
//Return Value: void
//=============================================================
template <class KeyType>
void            RBT<KeyType>::RBTFix    (Node<KeyType> *current)
{
        while (current->p->color == "red")
        {
                if(current->p == current->p->p->left)//parent is l child
                {
                        Node<KeyType> *uncle = current->p->p->right;
                        if (uncle->color == "red")      //if uncle red can just
                        {                                                       //change color
 of parent
                                current->p->color = "black";//and uncle to fix
                                uncle->color = "black";                 //black height
                                current->p->p->color = "red";
                                current = current->p->p;
                        }
                        else
                        {
                                if (current == current->p->right)
                                {
                                        current = current->p;   //makes current a
                                        LeftRotate(current);    //left child
                                }
                                current->p->color = "black";
                                current->p->p->color = "red";//fixes colors
                                RightRotate(current->p->p);
                        }
                }
                else                    //parent is right child
                {
                        Node<KeyType> *uncle = current->p->p->left;
                        if (uncle->color == "red")
                        {                                                               //switches col
or
                                current->p->color = "black";//of uncle
                                uncle->color = "black"; //and parent to fix
                                current->p->p->color = "red";//black height
                                current = current->p->p;
                        }
                        else
                        {
                                if (current == current->p->left)
                                {
                                        current = current->p;
                                        RightRotate(current);//makes current a right
                                }                                                               //chil
d
                                current->p->color = "black";
```

```
                                current->p->p->color = "red";//fixes colors
                                LeftRotate(current->p->p);

                            }
                        }
                }
                root->color = "black";          //fixes root color to black
}
```