template <class ItemType>

void TreeType<ItemType> :: InsertItem(ItemType item)

// Calls recursive function Insert to insert item into tree.

{

bool taller;

Insert (root, item, taller);

}

=====================================================================

template <class ItemType>

void Insert (TreeNode<ItemType>\*& tree, ItemType item, bool & taller)

// Inserts item into tree.

// Post: item is in tree; search property is maintained.

{

if (tree == NULL)

{ // Insertion place found.

tree = new TreeNode<ItemType>;

tree->left = NULL;

tree->right = NULL;

tree->info = item;

tree->bf = EH;

taller = true;

}

else if ( item == tree->info)

{

cerr << "Duplicate key is not allowed in AVL tree." << endl;

}

else if (item < tree->info)

{

Insert (tree->left, item, taller);

if (taller)

switch (tree->bf)

{

case LH: LeftBalance (tree, taller); break;

case EH: tree->bf = LH; break;

case RH: tree->bf = EH; taller = false;break;

}

}

else if (item > tree->info)

{

Insert (tree->right, item, taller);

if (taller)

switch (tree->bf)

{

case RH: RightBalance (tree, taller); break;

case EH: tree->bf = RH; break;

case LH: tree->bf = EH; taller = false;break;

}

}

}

template <class ItemType>

void RightBalance (TreeNode<ItemType> \*& tree, bool & taller)

{

TreeNode<ItemType> \* rs = tree->right;

TreeNode<ItemType> \* ls;

switch (rs->bf)

{

case RH: tree->bf = rs->bf = EH;

RotateLeft(tree);

taller = false;

break;

case EH: cerr << "Tree already balanced " << endl;

break;

case LH: ls = rs->left;

switch (ls->bf)

{

case RH: tree->bf = LH;

rs->bf = EH; break;

case EH: tree->bf = rs->bf = EH; break;

case LH: tree->bf = EH;

rs->bf = RH; break;

}

ls->bf = EH;

RotateRight(tree->right);

RotateLeft(tree);

taller = false;

}

}

template <class ItemType>

void LeftBalance (TreeNode<ItemType> \*& tree, bool & taller)

{

TreeNode<ItemType> \* ls = tree->left;

TreeNode<ItemType> \* rs;

switch (ls->bf)

{

case LH: tree->bf = ls->bf = EH;

RotateRight(tree);

taller = false;

break;

case EH: cerr << "Tree already balanced " << endl;

break;

case RH: rs = ls->right;

switch (rs->bf)

{

case LH: tree->bf = RH;

ls->bf = EH; break;

case EH: tree->bf = ls->bf = EH; break;

case RH: tree->bf = EH;

ls->bf = LH; break;

}

rs->bf = EH;

RotateLeft(tree->left);

RotateRight(tree);

taller = false;

}

}

template <class ItemType>

void RotateLeft (TreeNode<ItemType> \* & tree)

{

TreeNode<ItemType> \* rs;

if (tree == NULL)

cerr << "Cannot rotate an empty tree in RotateLeft"

<< endl;

else if (tree->right == NULL)

cerr << "Cannot make an empty subtree the root in

RotateLeft" << endl;

else

{

rs = tree->right;

tree->right = rs->left;

rs->left = tree;

tree = rs;

}

}

template <class ItemType>

void RotateRight (TreeNode<ItemType> \* & tree)

{

TreeNode<ItemType> \* ls;

if (tree == NULL)

cerr << "Cannot rotate an empty tree in RotateRight"

<< endl;

else if (tree->left == NULL)

cerr << "Cannot make an empty subtree the root in

RotateRight" << endl;

else

{

ls = tree->left;

tree->left = ls->right;

ls->right = tree;

tree = ls;

}

}