package KW.CH06;

/\*\*

\* Class for a binary tree that stores type E objects.

\* @param <E> The element type

\*/

public class BinaryTree<E>

{

/\*\*

\* Inner class to encapsulate a tree node.

\* @param <E> The element type

\*/

protected static class Node<E>

{

public E data; // The information stored in this node.

public Node<E> left; //Reference to the left child

public Node<E> right; //Reference to the right child.

// Constructors

/\*\*

\* Construct a node with given data and no children.

\* @param data The data to store in this node

\*/

public Node(E data)

{

this.data = data;

left = null;

right = null;

}

/\*\*

\* Returns a string representation of the node.

\* @return A string representation of the data fields

\*/

@Override

public String toString() {

return data.toString();

}

}

protected Node<E> root; //The root of the binary tree

/\*\*

\* Construct an empty BinaryTree

\*/

public BinaryTree( )

{

root = null;

}

/\*\*

\* Construct a BinaryTree with a specified root. Should only be used

\* by subclasses.

\* @param root The node that is the root of the tree.

\*/

protected BinaryTree(Node<E> root)

{

this.root = root;

}

/\*\*

\* Constructs a new binary tree with data in its root, leftTree as its

\* left subtree and rightTree as its right subtree.

\* @param data The data item to store in the root

\* @param leftTree the left child

\* @param rightTree the right child

\*/

public BinaryTree(E data, BinaryTree<E> leftTree,

BinaryTree<E> rightTree)

{

root = new Node<>(data);

if (leftTree != null) {

root.left = leftTree.root;

} else {

root.left = null;

}

if (rightTree != null) {

root.right = rightTree.root;

} else {

root.right = null;

}

}

/\*\*

\* Return the left subtree.

\* @return The left subtree or null if either the root or the left

\* subtree is null

\*/

public BinaryTree<E> getLeftSubtree()

{

if (root != null && root.left != null)

{

return new BinaryTree<>(root.left);

} else {

return null;

}

}

/\*\*

\* Return the right sub-tree

\* @return the right sub-tree or null if either the root or the right

\* subtree is null.

\*/

public BinaryTree<E> getRightSubtree() {

if (root != null && root.right != null)

{

return new BinaryTree<>(root.right);

} else {

return null;

}

}

/\*\*

\* Return the data field of the root

\* @return the data field of the root or null if the root is null

\*/

public E getData() {

if (root != null) {

return root.data;

} else {

return null;

}

}

/\*\*

\* Determine whether this tree is a leaf.

\* @return true if the root has no children

\*/

public boolean isLeaf() {

return (root == null || (root.left == null && root.right == null));

}

/\*\*

\* Starter method for preorder traversal

\*/

public void preOrderTraverse( )

{

preOrder(root);

}

/\*\*

\* Performs a recursive pre-order traversal of the tree,

\* whose root is at node

\* @param node The local root

\*/

private void preOrder(Node<E> node)

{

if (node == null) {

return;

} else {

System.out.print(node.data + “ ”);

preOrder(node.left);

preOrder(node.right);

}

}

/\*\*

\* Starter method for post-order traversal of the tree.

\*/

public void postOrderTraverse( )

{

postOrder(root);

}

/\*\*

\* Performs a recursive post-order traversal of the tree,

\* whose root is at node

\* @param node The local root

\*/

private void postOrder(Node<E> node)

{

if (node == null) {

return;

} else {

postOrder(node.left);

postOrder(node.right);

System.out.print(node.data + “ ”);

}

}

/\*\*

\* Starter method for in-order traversal of the tree.

\*/

public void inOrderTraverse( )

{

inOrderTraverse(root);

}

/\*\*

\* Performs a recursive in-order traversal of the tree,

\* whose root is at node

\* @param node The local root

\*/

private void inOrder(Node<E> node) {

if (node == null) {

return;

} else {

inOrder(node.left);

System.out.print(node.data + “ ”);

inOrder(node.right);

}

}

/\*\*

\* Starter method for height of the binary tree.

\*/

public int treeHeight( )

{

return height(root);

}

/\*\*

\* Finds the height of a binary tree recursively,

\* whose root is at node

\* @param node The local root

\* @return Returns the height of the binary tree

private int height(Node<E> node)

{

if (node == null) {

return 0 ;

else {

return (1 + max (height(node.left), height(node.right) ) );

}

/\*\*

\* Private method to compare two integers, called from method

\* height

\* @param x First integer to compare

\* @param y Second integer to compare

\* @return Returns the maximum of x and y

\*/

private int max(int x, int y)

{

If( x >= y )

return x;

else

return y;

}

/\*\*

\* Starter method for nodeCount of the binary tree.

\*/

public int treeNodeCount( )

{

return nodeCount(root);

}

/\*\*

\* Finds the number of nodes in a binary tree recursively,

\* whose root is at node

\* @param node The local root

\* @return Returns the number of nodes in the binary tree

private int nodeCount(Node<E> node)

{

if (node == null) {

return 0 ;

else {

return (1 + nodeCount(node.left) + nodeCount(node.right));

}

} /\*</listing>\*/