Binary search tree

Used Node and Class/Interface in single link list

**public** **class** BTNode<T> {

**public** T t;

**public** BTNode<T> lt, rt, pt, next, prev; // next point to successor/prdecessor of node

**public** **int** h, ind;

**public** BTNode(T t) {

**this**.t = t;

**this**.lt = **this**.rt = **this**.pt = **this**.next = **this**.prev = **null**;

**this**.h = 0;

**this**.ind = 0;

}

}

-------------------------------------------------------------------------------------

Used interface

----------------------------------------------------

/\*\* 1. Write a program to Delete a Tree \*\*/

**public** BTNode<T> deleteTree(BTNode<T> t);

/\*\* 2. Write a Program to Find the Maximum Depth or Height of a Tree \*\*/

**public** **int** maxHeightOrDepth(BTNode<T> t);

/\*\* 3. Write Code to Determine if Two Trees are Identical \*\*/

**public** **boolean** isTwoTreeIdentical(BTNode<T> t1, BTNode<T> t2);

/\*\* 4. Write a program to Calculate Size of a tree \*\*/

**public** **int** sizeOfTree(BTNode<T> t);

/\*\* 5. Root to leaf path sum equal to a given number \*\*/

**public** **boolean** rootToLeavPathSum(BTNode<T> node, **int** number);

**public** **boolean** pathWiseSum(BTNode<T> node, **int**[] path, **int** size, **int** number);

**public** **boolean** validate(**int**[] a, **int** size, **int** element);

/\*\* 6. How to determine if a binary tree is height-balanced? \*\*/

**public** **boolean** isBalanced(BTNode<T> t);

/\*\* 7. Diameter of a Binary Tree \*\*/

**public** **int** diameter(BTNode<T> t);

/\*\* 8. Check for Children Sum Property in a Binary Tree \*\*/

**public** **boolean** isChildrenSum(BTNode<T> t);

/\*\* 9. Program to count leaf nodes in a binary tree \*\*/

**public** **int** countLeafNode(BTNode<T> t);

/\*\* 10. The Great Tree-List Recursion Problem \*\*/

// @See IBTUtils.java...bstToListUtils(....)

/\*\*

\* 11. Given a binary tree, print out all of its root-to-leaf paths one per line

\*\*/

**public** **void** printAllPathPerLine(BTNode<T> t);

**public** **void** pathWisePrint(BTNode<Integer> node, **int**[] path, **int** size);

**public** **void** printArray(**int**[] a, **int** size);

/\*\* 12. Populate Inorder Successor for all nodes \*\*/

**public** **void** populateNext(BTNode<T> t);

**public** **void** printSuccConnected(BTNode<T> t);

/\*\* 13. Connect nodes at same level using constant extra space \*\*/

**public** **void** connectLevelNodes(BTNode<T> t);

**public** **void** printLevelConnect(BTNode<T> t);

/\*\* 14. Connect nodes at same level \*\*/

**public** **void** connectLevelNodesRec(BTNode<T> t);

/\*\* 15. Check if a binary tree is subtree of another binary tree | Set 1 \*\*/

**public** **boolean** isSubtree(BTNode<T> main, BTNode<T> sub);

**public** **boolean** isOverlap(BTNode<T> main, BTNode<T> sub);

**public** **boolean** isOverlapTree(BTNode<T> t1, BTNode<T> t2);

/\*\* 16. Check if a given Binary Tree is SumTree \*\*/

**public** **boolean** isSumTree(BTNode<T> t);

/\*\* 17. Print Ancestors of a given node in Binary Tree \*\*/

**public** **void** printAllAncestor(BTNode<T> t, **int** key);

/\*\* 18. Get Level of a node in a Binary Tree \*\*/

**public** **int** levelOfNode(BTNode<T> node, **int** k);

/\*\* 19. Print nodes at k distance from root \*\*/

**public** **void** printKdistNode(BTNode<T> root, **int** k);

/\*\* 20. Foldable Binary Trees \*\*/

**public** **boolean** isFoldable(BTNode<T> node);

**public** **boolean** leftRightFoldable(BTNode<T> a, BTNode<T> b);

/\*\* 21. Maximum width of a binary tree \*\*/

**public** **int** maxWidth(BTNode<T> t);

/\*\* 22. Double Tree \*\*/

**public** **void** doubleTree(BTNode<T> t);

/\*\* 23. Given a binary tree, print all root-to-leaf paths \*\*/

// @See 11. problem

/\*\* 24. Linked complete binary tree & its creation \*\*/

/\*

\* this is create complete binary tree @See BinaryTreeUtil.java...create(....)

\*/

/\*\*

\* 25. Check whether a given Binary Tree is Complete or not | Set 1 (Iterative

\* Solution)

\*\*/

**public** **boolean** isCompletBinrayTree(BTNode<T> t);

/\*\* 26. Find the maximum sum leaf to root path in a Binary Tree \*\*/

**public** **int** maxSumRootToLeaf(BTNode<T> t);

**public** **void** getTargetLeaf(BTNode<T> node, **int** maxSum, **int** currSum);

/\*\* 27. Vertical Sum in a given Binary Tree | Set 1 \*\*/

**public** Map<Integer, Integer> getVerticalSum(BTNode<T> t);

/\*\* 28. Sum of all the numbers that are formed from root to leaf paths \*\*/

**public** List<Integer> sumOfPathLeafToRoot(BTNode<T> t);

**public** List<Integer> getRootToLeafSum(BTNode<Integer> node, **int** sum);

/\*\* 29. Find next right node of a given key \*\*/

**public** BTNode<T> nextRightNode(BTNode<T> t, T key);

**public** BTNode<T> nextRightKthNode(BTNode<T> t, T key, **int** k);

**public** BTNode<T> nextLeftKthNode(BTNode<T> t, T key, **int** k);

/\*\* 30. Deepest left leaf node in a binary tree \*\*/

**public** BTNode<T> deepestLeftNode(BTNode<T> t);

/\*\* 31. Extract Leaves of a Binary Tree in a Doubly Linked List \*\*/

**public** BTNode<T> extractLeafNodeAsList(BTNode<T> t);

/\*\* 32. Remove all nodes which don’t lie in any path with sum>= k \*/

**public** BTNode<T> delAllPathLessSum(BTNode<T> t, **int** k);

**public** **void** pruneUtil(BTNode<Integer> node, **int** k, **int** sum);

/\*\* 33. Print Left View of a Binary Tree \*\*/

// @See all view of tree DAIHashingWithTree.java

/\*\* 34. Check if all leaves are at same level \*\*/

**public** **boolean** checkAllLeavesAtSameLevel(BTNode<T> t);

/\*\* 35. Find depth of the deepest odd level leaf node \*\*/

**public** BTNode<T> deepestOddlevelNode(BTNode<T> t);

/\*\*

\* 36. Difference between sums of odd level and even level nodes of a Binary

\* Tree

\*\*/

**public** **int** diffSumOfOddAndEvenLevel(BTNode<T> t);

/\*\* 37. Custom Tree Problem \*\*/

**public** BTNode<T> createCustomTree(BTNode<T> t, T p, T c);

/\*\* 38. Iterative Method to find Height of Binary Tree \*\*/

**public** **int** heightIterative(BTNode<T> t);

/\*\* 39. Tree Isomorphism Problem \*\*/

**public** **boolean** isIsomorphic(BTNode<T> t1, BTNode<T> t2);

/\*\* 40. Check if a binary tree is subtree of another binary tree | Set 2 \*\*/

// @See 15.

}

=====================================================================================Implementation of interface

-------------------------------------------------------------------------------------

**public** **class** MiscBinaryTreeImpl **implements** IMiscBinaryTree<Integer> {

/\*\* 1. Write a program to Delete a Tree \*\*/

@Override

**public** BTNode<Integer> deleteTree(BTNode<Integer> node) {

**if** (node != **null**) {

BTNode<Integer> lt = node.lt;

BTNode<Integer> rt = node.rt;

node = **null**;

deleteTree(lt);

deleteTree(rt);

}

**return** node;

}

/\*\* 2. Write a Program to Find the Maximum Depth or Height of a Tree \*\*/

@Override

**public** **int** maxHeightOrDepth(BTNode<Integer> node) {

**if** (node != **null**) {

**if** (node.lt == **null** && node.rt == **null**)

**return** 1;

**return** 1 + Math.*max*(maxHeightOrDepth(node.lt), maxHeightOrDepth(node.rt));

} **else**

**return** 0;

}

/\*\* 3. Write Code to Determine if Two Trees are Identical \*\*/

@Override

**public** **boolean** isTwoTreeIdentical(BTNode<Integer> t1, BTNode<Integer> t2) {

**if** (t1 == **null** && t2 == **null**)

**return** **true**;

**if** (t1 != **null** && t2 != **null**) {

**if** (t1.t.equals(t2.t))

**return** isTwoTreeIdentical(t1.lt, t2.lt) && isTwoTreeIdentical(t1.rt, t2.rt);

**else**

**return** **false**;

} **else**

**return** **false**;

}

/\*\* 4. Write a program to Calculate Size of a tree \*\*/

@Override

**public** **int** sizeOfTree(BTNode<Integer> node) {

**if** (node == **null**)

**return** 0;

**else**

**return** 1 + sizeOfTree(node.lt) + sizeOfTree(node.rt);

}

/\*\* 5. Root to leaf path sum equal to a given number \*\*/

@Override

**public** **boolean** rootToLeavPathSum(BTNode<Integer> node, **int** number) {

**if** (node == **null**) {

**return** **false**;

}

**int**[] path = **new** **int**[10];

**return** pathWiseSum(node, path, 0, number);

}

@Override

**public** **boolean** pathWiseSum(BTNode<Integer> node, **int**[] path, **int** size, **int** number) {

**if** (node != **null**) {

path[size++] = node.t;

**if** (node.lt == **null** && node.rt == **null**) {

**return** validate(path, size, number);

} **else** {

**return** pathWiseSum(node.lt, path, size, number) || pathWiseSum(node.rt, path, size, number);

}

} **else**

**return** **false**;

}

@Override

**public** **boolean** validate(**int**[] a, **int** size, **int** element) {

**int** sum = 0;

**int** i = 0;

**while** (i < size) {

sum += a[i];

++i;

}

**if** (element == sum) {

**return** **true**;

}

**return** **false**;

}

/\*\* 6. How to determine if a binary tree is height-balanced? \*\*/

@Override

**public** **boolean** isBalanced(BTNode<Integer> node) {

**int** lh; /\* for height of left subtree \*/

**int** rh; /\* for height of right subtree \*/

/\* If tree is empty then return true \*/

**if** (node == **null**)

**return** **true**;

/\* Get the height of left and right sub trees \*/

lh = BinaryTreeUtil.*height*(node.lt);

rh = BinaryTreeUtil.*height*(node.rt);

**if** (Math.*abs*(lh - rh) <= 1 && isBalanced(node.lt) && isBalanced(node.rt))

**return** **true**;

/\* If we reach here then tree is not height-balanced \*/

**return** **false**;

}

/\*\* 7. Diameter of a Binary Tree \*\*/

@Override

**public** **int** diameter(BTNode<Integer> node) {

**if** (node == **null**) {

**return** 0;

}

**int** lh = BinaryTreeUtil.*height*(node.lt);

**int** rh = BinaryTreeUtil.*height*(node.rt);

**int** ld = diameter(node.lt);

**int** rd = diameter(node.rt);

**return** Math.*max*(lh + rh + 1, Math.*max*(ld, rd));

}

/\*\* 8. Check for Children Sum Property in a Binary Tree \*\*/

@Override

**public** **boolean** isChildrenSum(BTNode<Integer> node) {

**int** ltd, rtd;

ltd = rtd = 0;

**if** (node == **null** || (node.lt == **null** && node.rt == **null**))

**return** **true**;

**else** {

**if** (node.lt != **null**)

ltd = node.lt.t;

**if** (node.rt != **null**)

rtd = node.rt.t;

**if** ((node.t == ltd + rtd) && isChildrenSum(node.lt) && isChildrenSum(node.rt))

**return** **true**;

**else**

**return** **false**;

}

}

/\*\* 9. Program to count leaf nodes in a binary tree \*\*/

@Override

**public** **int** countLeafNode(BTNode<Integer> node) {

**if** (node == **null**)

**return** 0;

**if** (node.lt == **null** && node.rt == **null**)

**return** 1;

**return** countLeafNode(node.lt) + countLeafNode(node.rt);

}

/\*\* 10. The Great Tree-List Recursion Problem \*\*/

// @See IBTUtils.java...bstToListUtils(....)

/\*\*

\* 11. Given a binary tree, print out all of its root-to-leaf paths one per line

\*\*/

@Override

**public** **void** printAllPathPerLine(BTNode<Integer> node) {

**if** (node == **null**) {

**return**;

}

**int**[] path = **new** **int**[20];

pathWisePrint(node, path, 0);

}

@Override

**public** **void** pathWisePrint(BTNode<Integer> node, **int**[] path, **int** size) {

**if** (node != **null**) {

path[size++] = node.t;

**if** (node.lt == **null** && node.rt == **null**) {

printArray(path, size);

} **else** {

pathWisePrint(node.lt, path, size);

pathWisePrint(node.rt, path, size);

}

}

}

@Override

**public** **void** printArray(**int**[] a, **int** size) {

**int** i = 0;

**while** (i < size) {

System.***out***.print(a[i] + "->");

++i;

}

System.***out***.println();

}

/\*\* 12. Populate Inorder Successor for all nodes \*\*/

**static** BTNode<Integer> *next* = **null**;

@Override

**public** **void** populateNext(BTNode<Integer> node) {

**if** (node != **null**) {

// First set the next pointer in right subtree

populateNext(node.rt);

// Set the next as previously visited node in reverse Inorder

node.next = *next*;

// Change the prev for subsequent node

*next* = node;

// Finally, set the next pointer in left subtree

populateNext(node.lt);

}

}

@Override

**public** **void** printSuccConnected(BTNode<Integer> node) {

**while** (node.lt != **null**)

node = node.lt;

**while** (node != **null**) {

System.***out***.print(node.t + "->");

node = node.next;

}

}

/\*\* 13. Connect nodes at same level using constant extra space \*\*/

@Override

**public** **void** connectLevelNodes(BTNode<Integer> node) {

**if** (node != **null**) {

Queue<BTNode<Integer>> que = **new** LinkedList<>();

que.add(node);

**while** (!que.isEmpty()) {

List<BTNode<Integer>> list = **new** ArrayList<>();

**while** (!que.isEmpty()) {

list.add(que.poll());

}

**for** (**int** i = 0; i < list.size(); i++) {

BTNode<Integer> bt = list.get(i);

**if** (i != 0) {

list.get(i - 1).next = bt;

}

**if** (bt.lt != **null**)

que.add(bt.lt);

**if** (bt.rt != **null**)

que.add(bt.rt);

}

}

}

}

@Override

**public** **void** printLevelConnect(BTNode<Integer> node) {

**while** (node != **null**) {

BTNode<Integer> bt = node;

**while** (bt != **null**) {

System.***out***.print(bt.t + "->");

bt = bt.next;

}

node = node.lt;

System.***out***.println();

}

}

/\*\* 14. Connect nodes at same level \*\*/

@Override

**public** **void** connectLevelNodesRec(BTNode<Integer> node) {

// Base case

**if** (node == **null**)

**return**;

// Set the nextRight pointer for p's left child

**if** (node.lt != **null**)

node.lt.next = node.rt;

// Set the next pointer for node's right child

// p->next will be null if node is the right most child

// at its level

**if** (node.rt != **null**)

node.rt.next = (node.next != **null**) ? node.next.lt : **null**;

// Set nextRight for other nodes in pre order fashion

connectLevelNodesRec(node.lt);

connectLevelNodesRec(node.rt);

}

/\*\* 15. Check if a binary tree is subtree of another binary tree | Set 1 \*\*/

@Override

**public** **boolean** isSubtree(BTNode<Integer> t, BTNode<Integer> subT) {

/\* base cases \*/

**if** (subT == **null**)

**return** **true**;

**if** (t == **null**)

**return** **false**;

/\* Check the tree with root as current node \*/

**if** (BinaryTreeUtil.*areIdentical*(t, subT))

**return** **true**;

/\*

\* If the tree with root as current node doesn't match then try left and right

\* subtrees one by one

\*/

**return** isSubtree(t.lt, subT) || isSubtree(t.rt, subT);

}

@Override

**public** **boolean** isOverlap(BTNode<Integer> t, BTNode<Integer> subT) {

BTNode<Integer> node = BinaryTreeUtil.*search*(t, subT);

**if** (node != **null**)

**return** isOverlapTree(node, subT);

**return** **false**;

}

@Override

**public** **boolean** isOverlapTree(BTNode<Integer> b1, BTNode<Integer> b2) {

**if** (b1 == **null** && b2 == **null**)

**return** **true**;

**if** (b2 == **null**)

**return** **true**;

**return** b1.t.equals(b2.t) && isOverlapTree(b1.lt, b2.lt) && isOverlapTree(b1.rt, b2.rt);

}

/\*\* 16. Check if a given Binary Tree is SumTree \*\*/

@Override

**public** **boolean** isSumTree(BTNode<Integer> node) {

**int** ls, rs;

**if** ((node == **null**) || (node.lt == **null** && node.rt == **null**))

**return** **true**;

ls = sum(node.lt);

rs = sum(node.rt);

**if** ((node.t == ls + rs) && (isSumTree(node.lt)) && (isSumTree(node.rt)))

**return** **true**;

**return** **false**;

}

**public** **int** sum(BTNode<Integer> node) {

**if** (node == **null**)

**return** 0;

**return** sum(node.lt) + node.t + sum(node.rt);

}

/\*\* 17. Print Ancestors of a given node in Binary Tree \*\*/

@Override

**public** **void** printAllAncestor(BTNode<Integer> node, **int** key) {

BTNode<Integer> x = BinaryTreeUtil.*search*(node, **new** BTNode<Integer>(key));

**if** (x != **null**)

x = x.pt;

**while** (x != **null**) {

System.***out***.print(x.t + "->");

x = x.pt;

}

}

/\*\* 18. Get Level of a node in a Binary Tree \*\*/

@Override

**public** **int** levelOfNode(BTNode<Integer> node, **int** k) {

**if** (node == **null**) {

**return** 0;

}

BTNode<Integer> n = **new** BTNode<Integer>(k);

BTNode<Integer> x = BinaryTreeUtil.*search*(node, n);

**int** lev = 0;

**while** (x != **null**) {

lev++;

x = x.pt;

}

**return** lev;

}

@Override

**public** **void** printKdistNode(BTNode<Integer> root, **int** k) {

**if** (root == **null**) {

**return**;

}

**if** (k == 1) {

System.***out***.print(String.*valueOf*(root.t) + "->");

} **else** **if** (k > 1) {

printKdistNode(root.lt, k - 1);

printKdistNode(root.rt, k - 1);

}

}

@Override

**public** **boolean** isFoldable(BTNode<Integer> node) {

**if** (node == **null**) {

**return** **true**;

}

**return** leftRightFoldable(node.lt, node.rt);

}

@Override

**public** **boolean** leftRightFoldable(BTNode<Integer> a, BTNode<Integer> b) {

**if** (a == **null** && b == **null**)

**return** **true**;

**return** (a != **null** && b != **null**) && leftRightFoldable(a.lt, b.lt) && leftRightFoldable(a.rt, b.rt);

}

/\*\* 21. Maximum width of a binary tree \*\*/

@Override

**public** **int** maxWidth(BTNode<Integer> node) {

**int** maxWith = 0;

**if** (node != **null**) {

maxWith = 1;

Queue<BTNode<Integer>> que = **new** LinkedList<>();

que.add(node);

**while** (!que.isEmpty()) {

List<BTNode<Integer>> list = **new** ArrayList<>();

**while** (!que.isEmpty()) {

list.add(que.poll());

}

**for** (**int** i = 0; i < list.size(); i++) {

BTNode<Integer> bt = list.get(i);

**if** (bt.lt != **null**)

que.add(bt.lt);

**if** (bt.rt != **null**)

que.add(bt.rt);

}

**if** (maxWith < list.size())

maxWith = list.size();

}

}

**return** maxWith;

}

/\*\* 22. Double Tree \*\*/

**public** **void** doubleTree(BTNode<Integer> node) {

BTNode<Integer> oldleft;

**if** (node == **null**)

**return**;

/\* do the subtrees \*/

doubleTree(node.lt);

doubleTree(node.rt);

/\* duplicate this node to its left \*/

oldleft = node.lt;

node.lt = **new** BTNode<>(node.t);

node.lt.lt = oldleft;

}

/\*\* 23. Given a binary tree, print all root-to-leaf paths \*\*/

// @See 11. problem

/\*\* 24. Linked complete binary tree & its creation \*\*/

/\*

\* this is create complete binary tree @See BinaryTreeUtil.java...create(....)

\*/

/\*\*

\* 25. Check whether a given Binary Tree is Complete or not | Set 1 (Iterative

\* Solution)

\*\*/

@Override

**public** **boolean** isCompletBinrayTree(BTNode<Integer> node) {

**if** (node != **null**) {

Queue<BTNode<Integer>> que = **new** LinkedList<>();

que.add(node);

**while** (!que.isEmpty()) {

List<BTNode<Integer>> list = **new** ArrayList<>();

**while** (!que.isEmpty()) {

list.add(que.poll());

}

**for** (**int** i = 0; i < list.size(); i++) {

BTNode<Integer> bt = list.get(i);

**if** (bt.lt != **null**)

que.add(bt.lt);

**if** (bt.rt != **null**)

que.add(bt.rt);

**if** (bt.lt == **null** && bt.rt != **null**)

**return** **false**;

}

}

}

**return** **true**;

}

/\*\* 26. Find the maximum sum leaf to root path in a Binary Tree \*\*/

@Override

**public** **int** maxSumRootToLeaf(BTNode<Integer> node) {

getTargetLeaf(node, 0, 0);

BTNode<Integer> tgNode = targetNode;

**int** maxSum = 0;

**while** (tgNode != **null**) {

maxSum += tgNode.t;

System.***out***.print(tgNode.t + "->");

tgNode = tgNode.pt;

}

**return** maxSum;

}

**public** BTNode<Integer> targetNode = **null**;

@Override

**public** **void** getTargetLeaf(BTNode<Integer> node, **int** maxSum, **int** currSum) {

**if** (node == **null**)

**return**;

currSum += node.t;

**if** (node.lt == **null** && node.rt == **null**) {

**if** (currSum > maxSum) {

maxSum = currSum;

targetNode = node;

}

}

getTargetLeaf(node.lt, maxSum, currSum);

getTargetLeaf(node.rt, maxSum, currSum);

}

/\*\* 27. Vertical Sum in a given Binary Tree | Set 1 \*\*/

@Override

**public** Map<Integer, Integer> getVerticalSum(BTNode<Integer> node) {

Map<Integer, Integer> map = **new** TreeMap<>();

**if** (node != **null**) {

Queue<BTNode<Integer>> que = **new** LinkedList<>();

que.add(node);

**while** (!que.isEmpty()) {

BTNode<Integer> bt = que.poll();

**if** (map.containsKey(bt.ind))

map.put(bt.ind, map.get(bt.ind) + bt.t);

**else**

map.put(bt.ind, bt.t);

**if** (bt.lt != **null**)

que.add(bt.lt);

**if** (bt.rt != **null**)

que.add(bt.rt);

}

}

**return** map;

}

/\*\* 28. Sum of all the numbers that are formed from root to leaf paths \*\*/

@Override

**public** List<Integer> sumOfPathLeafToRoot(BTNode<Integer> node) {

**return** getRootToLeafSum(node, 0);

}

**public** List<Integer> sumList = **null**;

@Override

**public** List<Integer> getRootToLeafSum(BTNode<Integer> node, **int** sum) {

**if** (sum == 0 || sumList == **null**)

sumList = **new** ArrayList<>();

**if** (node != **null**) {

sum += node.t;

**if** (node.lt == **null** && node.rt == **null**) {

sumList.add(sum);

}

getRootToLeafSum(node.lt, sum);

getRootToLeafSum(node.rt, sum);

}

**return** sumList;

}

/\*\* 29. Find next right node of a given key \*\*/

@Override

**public** BTNode<Integer> nextRightNode(BTNode<Integer> node, Integer key) {

BinaryTreeUtil.*linkNextPrevLevelWise*(node);

BTNode<Integer> keyNode = BinaryTreeUtil.*search*(node, **new** BTNode<Integer>(key));

**if** (keyNode != **null**)

**return** keyNode.next;

**return** **null**;

}

@Override

**public** BTNode<Integer> nextRightKthNode(BTNode<Integer> node, Integer key, **int** k) {

BinaryTreeUtil.*linkNextPrevLevelWise*(node);

BTNode<Integer> keyNode = BinaryTreeUtil.*search*(node, **new** BTNode<Integer>(key));

**int** i = 0;

**while** (keyNode != **null** && i++ != k)

keyNode = keyNode.next;

**return** keyNode;

}

@Override

**public** BTNode<Integer> nextLeftKthNode(BTNode<Integer> node, Integer key, **int** k) {

BinaryTreeUtil.*linkNextPrevLevelWise*(node);

BTNode<Integer> keyNode = BinaryTreeUtil.*search*(node, **new** BTNode<Integer>(key));

**int** i = 0;

**while** (keyNode != **null** && i++ != k)

keyNode = keyNode.prev;

**return** keyNode;

}

/\*\* 30. Deepest left leaf node in a binary tree \*\*/

BTNode<Integer> deepestLt = **null**;

@Override

**public** BTNode<Integer> deepestLeftNode(BTNode<Integer> node) {

**if** (node != **null**) {

**if** (node.lt == **null** && node.rt == **null** && node.pt.lt != **null** && node.pt.lt.equals(node)) {

**if** (deepestLt == **null**)

deepestLt = node;

**else** {

**if** (node.h > deepestLt.h)

deepestLt = node;

}

}

deepestLeftNode(node.lt);

deepestLeftNode(node.rt);

}

**return** deepestLt;

}

/\*\* 31. Extract Leaves of a Binary Tree in a Doubly Linked List \*\*/

List<BTNode<Integer>> list = **null**;

@Override

**public** BTNode<Integer> extractLeafNodeAsList(BTNode<Integer> node) {

List<BTNode<Integer>> l = collectLeafNode(node);

**int** i = 1;

**if** (l.size() > 1) {

BTNode<Integer> x = l.get(0);

BTNode<Integer> y = l.get(i);

**for** (; i < l.size(); y = l.get(i)) {

x.next = y;

y.prev = x;

x = y;

i++;

**if** (i == l.size())

**break**;

}

}

**return** l.get(0);

}

**public** List<BTNode<Integer>> collectLeafNode(BTNode<Integer> node) {

**if** (list == **null**)

list = **new** ArrayList<>();

**if** (node != **null**) {

**if** (node.lt == **null** && node.rt == **null**)

list.add(node);

collectLeafNode(node.lt);

collectLeafNode(node.rt);

}

**return** list;

}

/\*\* 32. Remove all nodes which don’t lie in any path with sum>= k \*/

BTNode<Integer> rootNode = **null**;

@Override

**public** BTNode<Integer> delAllPathLessSum(BTNode<Integer> node, **int** k) {

rootNode = node;

pruneUtil(node, k, 0);

**return** rootNode;

}

@Override

**public** **void** pruneUtil(BTNode<Integer> node, **int** k, **int** sum) {

// Base Case

**if** (node == **null**)

**return**;

// Initialize left and right sums as sum from root to

// this node (including this node)

**int** lsum = sum + (node.t);

**int** rsum = lsum;

// Recursively prune left and right subtrees

pruneUtil(node.lt, k, lsum);

pruneUtil(node.rt, k, rsum);

// Get the maximum of left and right sums

sum = Math.*max*(lsum, rsum);

// If maximum is smaller than k, then this node

// must be deleted and deleted node must be leaf node

**if** (sum < k && node.lt == **null** && node.rt == **null**) {

System.***out***.println("deleted:-" + node.t);

BinaryTreeUtil.*delete*(rootNode, node.t);

}

}

/\*\* 34. Check if all leaves are at same level \*\*/

**int** leafLevel = 0;

@Override

**public** **boolean** checkAllLeavesAtSameLevel(BTNode<Integer> node) {

**if** (node == **null**)

**return** **true**;

**if** (node.lt == **null** && node.rt == **null**) {

**if** (leafLevel == 0)

leafLevel = node.h;

**else** **if** (node.h != leafLevel)

**return** **false**;

**else**

**return** **true**;

}

**return** checkAllLeavesAtSameLevel(node.lt) && checkAllLeavesAtSameLevel(node.rt);

}

/\*\* 35. Find depth of the deepest odd level leaf node \*\*/

BTNode<Integer> oddNode = **null**;

**int** deepestOddLevel = 0;

**public** BTNode<Integer> deepestOddlevelNode(BTNode<Integer> node) {

**if** (node != **null**) {

**if** (node.h % 2 != 0) {

**if** (deepestOddLevel <= node.h) {

deepestOddLevel = node.h;

oddNode = node;

}

}

deepestOddlevelNode(node.lt);

deepestOddlevelNode(node.rt);

}

**return** oddNode;

}

/\*\*

\* 36. Difference between sums of odd level and even level nodes of a Binary

\* Tree

\*\*/

**int** sumOddLevl = 0;

**int** sumEvenLevel = 0;

@Override

**public** **int** diffSumOfOddAndEvenLevel(BTNode<Integer> node) {

**if** (node != **null**) {

**if** (node.h % 2 == 0)

sumEvenLevel += node.t;

**else**

sumOddLevl += node.t;

diffSumOfOddAndEvenLevel(node.lt);

diffSumOfOddAndEvenLevel(node.rt);

}

**return** sumOddLevl - sumEvenLevel;

}

/\*\* 37. Custom Tree Problem \*\*/

@Override

**public** BTNode<Integer> createCustomTree(BTNode<Integer> node, Integer parent, Integer child) {

**return** BinaryTreeUtil.*createCustomTree*(node, parent, child);

}

/\*\* 38. Iterative Method to find Height of Binary Tree \*\*/

@Override

**public** **int** heightIterative(BTNode<Integer> node) {

**int** height = -1;

**if** (node != **null**) {

Queue<BTNode<Integer>> que = **new** LinkedList<>();

que.add(node);

**while** (!que.isEmpty()) {

BTNode<Integer> bt = que.poll();

**if** (bt.lt == **null** && bt.rt == **null**) {

BTNode<Integer> x = bt;

**int** curHeight = 0;

**while** (x.pt != **null**) {

curHeight++;

x = x.pt;

}

**if** (height == 0)

height = curHeight;

**else** **if** (curHeight > height)

height = curHeight;

}

**if** (bt.lt != **null**)

que.add(bt.lt);

**if** (bt.rt != **null**)

que.add(bt.rt);

}

}

**return** height;

}

/\*\* 39. Tree Isomorphism Problem \*\*/

@Override

**public** **boolean** isIsomorphic(BTNode<Integer> n1, BTNode<Integer> n2) {

// Both roots are NULL, trees isomorphic by definition

**if** (n1 == **null** && n2 == **null**)

**return** **true**;

**if** (n1 == **null** || n2 == **null**)

**return** **false**;

**if** (n1.t != n2.t)

**return** **false**;

**return** (isIsomorphic(n1.lt, n2.lt) && isIsomorphic(n1.rt, n2.rt))

|| (isIsomorphic(n1.lt, n2.rt) && isIsomorphic(n1.rt, n2.lt));

}

}

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

Test case:

-----------------------------------------------------------------

**public** **class** IMiscBinaryTreeTest {

**public** IMiscBinaryTree<Integer> imbt = **null**;

@Before

**public** **void** init() {

imbt = **new** MiscBinaryTreeImpl();

}

/\*\* 1. Write a program to Delete a Tree \*\*/

@Test

**public** **void** deleteTreeTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7 };

// Integer result[] = { 1, 2, 4, 5, 3, 6, 7 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

// BinaryTreeUtil.levelTravelLineByline(root);

root = imbt.deleteTree(root);

Assert.*assertTrue*(root == **null**);

}

/\*\* 2. Write a Program to Find the Maximum Depth or Height of a Tree \*\*/

@Test

**public** **void** maxHeightOrDepthTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8 };

// Integer result[] = { 1, 2, 4, 5, 3, 6, 7 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

// BinaryTreeUtil.levelTravelLineByline(root);

Assert.*assertTrue*(imbt.maxHeightOrDepth(root) == 4);

}

/\*\* 3. Write Code to Determine if Two Trees are Identical \*\*/

@Test

**public** **void** isTwoTreeIdenticalTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8 };

// Integer result[] = { 1, 2, 4, 5, 3, 6, 7 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

BTNode<Integer> root1 = **null**;

**for** (**int** i = 0; i < a.length; i++)

root1 = BinaryTreeUtil.*create*(root1, a[i]);

Integer b[] = { 1, 2, 3, 4, 5, 6, 7 };

BTNode<Integer> root2 = **null**;

**for** (**int** i = 0; i < b.length; i++)

root2 = BinaryTreeUtil.*create*(root2, b[i]);

Assert.*assertTrue*(imbt.isTwoTreeIdentical(root, root1));

Assert.*assertTrue*(!imbt.isTwoTreeIdentical(root2, root1));

}

/\*\* 4. Write a program to Calculate Size of a tree \*\*/

@Test

**public** **void** sizeOfTreeTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8 };

Integer b[] = { 1, 2, 4, 5, 3, 6, 7 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

Assert.*assertTrue*(imbt.sizeOfTree(root) == 8);

root = **null**;

**for** (**int** i = 0; i < b.length; i++)

root = BinaryTreeUtil.*create*(root, b[i]);

Assert.*assertTrue*(imbt.sizeOfTree(root) == 7);

}

/\*\* 5. Root to leaf path sum equal to a given number \*\*/

@Test

**public** **void** rootToLeavPathSum() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

Assert.*assertTrue*(imbt.rootToLeavPathSum(root, 15));

Assert.*assertTrue*(imbt.rootToLeavPathSum(root, 10));

Assert.*assertTrue*(imbt.rootToLeavPathSum(root, 11));

Assert.*assertTrue*(!imbt.rootToLeavPathSum(root, 16));

Assert.*assertTrue*(!imbt.rootToLeavPathSum(root, 4));

Assert.*assertTrue*(!imbt.rootToLeavPathSum(root, 9));

}

/\*\* 6. How to determine if a binary tree is height-balanced? \*\*/

@Test

**public** **void** isBalanced() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

Assert.*assertTrue*(imbt.isBalanced(root));

}

/\*\* 7. Diameter of a Binary Tree \*\*/

@Test

**public** **void** diameterOfTreeTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

Assert.*assertTrue*(imbt.diameter(root) == 6);

}

/\*\* 8. Check for Children Sum Property in a Binary Tree \*\*/

@Test

**public** **void** isChildrenSumTest() {

Integer a[] = { 10, 2, 8, 2, 0, 6, 2 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

Assert.*assertTrue*(imbt.isChildrenSum(root));

}

/\*\* 9. Program to count leaf nodes in a binary tree \*\*/

@Test

**public** **void** countLeafNodeTest() {

Integer a[] = { 10, 2, 8, 2, 0, 6, 2, 8, 1, 2, 3, 4, 5, 6, 7 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

Assert.*assertTrue*(imbt.countLeafNode(root) == 8);

}

/\*\* 10. The Great Tree-List Recursion Problem \*\*/

// @See IBTUtils.java...bstToListUtils(....)

/\*\*

\* 11. Given a binary tree, print out all of its root-to-leaf paths one per line

\*\*/

@Test

**public** **void** printAllPathPerLineTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

imbt.printAllPathPerLine(root);

}

/\*\* 12. Populate Inorder Successor for all nodes \*\*/

@Test

**public** **void** populateNextTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

imbt.populateNext(root);

imbt.printSuccConnected(root);

}

/\*\* 13. Connect nodes at same level using constant extra space \*\*/

@Test

**public** **void** connectLevelNodesTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

imbt.connectLevelNodes(root);

imbt.printLevelConnect(root);

}

/\*\* 14. Connect nodes at same level \*\*/

@Test

**public** **void** connectLevelNodesRecTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

imbt.connectLevelNodesRec(root);

imbt.printLevelConnect(root);

}

/\*\* 15. Check if a binary tree is subtree of another binary tree | Set 1 \*\*/

@Test

**public** **void** isSubtreeTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

Integer b[] = { 2, 4, 5, 8, 9 };

Integer c[] = { 1, 2, 3, 4, 5 };

BTNode<Integer> root = **null**;

BTNode<Integer> root1 = **null**;

BTNode<Integer> root2 = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

**for** (**int** i = 0; i < b.length; i++)

root1 = BinaryTreeUtil.*create*(root1, b[i]);

**for** (**int** i = 0; i < c.length; i++)

root2 = BinaryTreeUtil.*create*(root2, c[i]);

Assert.*assertTrue*(imbt.isSubtree(root, root1));

Assert.*assertTrue*(!imbt.isSubtree(root, root2));

}

@Test

**public** **void** isOverlapTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

Integer b[] = { 1, 2, 4, 5, 8, 9 };

Integer c[] = { 1, 2, 3, 4, 5 };

BTNode<Integer> root = **null**;

BTNode<Integer> root1 = **null**;

BTNode<Integer> root2 = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

**for** (**int** i = 0; i < b.length; i++)

root1 = BinaryTreeUtil.*create*(root1, b[i]);

**for** (**int** i = 0; i < c.length; i++)

root2 = BinaryTreeUtil.*create*(root2, c[i]);

Assert.*assertTrue*(!imbt.isOverlap(root, root1));

Assert.*assertTrue*(imbt.isOverlap(root, root2));

}

/\*\* 16. Check if a given Binary Tree is SumTree \*\*/

@Test

**public** **void** isSumTreeTest() {

Integer a[] = { 26, 10, 3, 4, 6, 3 };

Integer b[] = { 26, 10, 3, 4, 6, 4 };

BTNode<Integer> root = **null**;

BTNode<Integer> root1 = **null**;

**for** (**int** i = 0; i < a.length; i++) {

root = BinaryTreeUtil.*create*(root, a[i]);

root1 = BinaryTreeUtil.*create*(root1, b[i]);

}

Assert.*assertTrue*(imbt.isSumTree(root));

Assert.*assertTrue*(!imbt.isSumTree(root1));

}

/\*\* 17. Print Ancestors of a given node in Binary Tree \*\*/

@Test

**public** **void** printAllAncestorTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++) {

root = BinaryTreeUtil.*create*(root, a[i]);

imbt.printAllAncestor(root, 9);

}

}

/\*\* 18. Get Level of a node in a Binary Tree \*\*/

@Test

**public** **void** levelOfNodeTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

Assert.*assertTrue*(imbt.levelOfNode(root, 9) == 4);

}

/\*\* 19. Print nodes at k distance from root \*\*/

@Test

**public** **void** printKdistNodeTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

imbt.printKdistNode(root, 4);

}

/\*\* 20. Foldable Binary Trees \*\*/

@Test

**public** **void** isFoldableTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

Assert.*assertTrue*(imbt.isFoldable(root));

root = BinaryTreeUtil.*create*(root, 8);

Assert.*assertTrue*(!imbt.isFoldable(root));

}

/\*\* 21. Maximum width of a binary tree \*\*/

@Test

**public** **void** maxWidthTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

Assert.*assertTrue*(imbt.maxWidth(root) == 4);

}

/\*\* 22. Double Tree \*\*/

@Test

**public** **void** doubleTreeTest() {

Integer a[] = { 1, 2, 3 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

imbt.doubleTree(root);

imbt.printAllPathPerLine(root);

}

/\*\* 23. Given a binary tree, print all root-to-leaf paths \*\*/

// @See 11. problem

/\*\* 24. Linked complete binary tree & its creation \*\*/

/\*

\* this is create complete binary tree @See BinaryTreeUtil.java...create(....)

\*/

/\*\*

\* 25. Check whether a given Binary Tree is Complete or not | Set 1 (Iterative

\* Solution)

\*\*/

@Test

**public** **void** isCompletBinaryTreeTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

Integer b[] = { 5, 3, 7, 2, 4, 6, 8, 9 };

BTNode<Integer> root = **null**;

BTNode<Integer> root1 = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

**for** (**int** i = 0; i < b.length; i++)

root1 = BinaryTreeUtil.*createBst*(root1, b[i]);

Assert.*assertTrue*(imbt.isCompletBinrayTree(root));

Assert.*assertTrue*(!imbt.isCompletBinrayTree(root1));

}

/\*\* 26. Find the maximum sum leaf to root path in a Binary Tree \*\*/

@Test

**public** **void** maxSumRootToLeafTest() {

Integer a[] = { 10, -2, 7, 8, -4 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

Assert.*assertTrue*(imbt.maxSumRootToLeaf(root) == 17);

}

/\*\* 27. Vertical Sum in a given Binary Tree | Set 1 \*\*/

@Test

**public** **void** getVerticalSumTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

**int** b[] = { 4, 2, 12, 3, 7 };

**int** i = 0;

**for** (Map.Entry<Integer, Integer> set : imbt.getVerticalSum(root).entrySet())

Assert.*assertTrue*(set.getValue() == b[i++]);

}

/\*\* 28. Sum of all the numbers that are formed from root to leaf paths \*\*/

@Test

**public** **void** sumOfPathLeafToRootTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

**int** b[] = { 4, 2, 12, 3, 7 };

**int** i = 0;

**for** (Integer in : imbt.sumOfPathLeafToRoot(root))

Assert.*assertTrue*(in == b[i]);

}

/\*\* 29. Find next right node of a given key \*\*/

@Test

**public** **void** nextRightNodeTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

Assert.*assertTrue*(imbt.nextRightNode(root, 5).t == 6);

Assert.*assertTrue*(imbt.nextRightNode(root, 2).t == 3);

Assert.*assertTrue*(imbt.nextRightNode(root, 7) == **null**);

Assert.*assertTrue*(imbt.nextRightNode(root, 9) == **null**);

}

@Test

**public** **void** nextRightKthNodeTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

Assert.*assertTrue*(imbt.nextRightKthNode(root, 5, 2).t == 7);

Assert.*assertTrue*(imbt.nextRightKthNode(root, 4, 2).t == 6);

Assert.*assertTrue*(imbt.nextRightKthNode(root, 3, 2) == **null**);

Assert.*assertTrue*(imbt.nextRightKthNode(root, 4, 3).t == 7);

}

@Test

**public** **void** nextLeftKthNodeTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

Assert.*assertTrue*(imbt.nextLeftKthNode(root, 5, 1).t == 4);

Assert.*assertTrue*(imbt.nextLeftKthNode(root, 4, 2) == **null**);

Assert.*assertTrue*(imbt.nextLeftKthNode(root, 7, 2).t == 5);

Assert.*assertTrue*(imbt.nextLeftKthNode(root, 7, 3).t == 4);

}

/\*\* 30. Deepest left leaf node in a binary tree \*\*/

@Test

**public** **void** deepestLeftNodeTest() {

Integer a[] = { 7, 4, 9, 6, 5, 3, 2, 9, 8, 12, 11, 10 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*createBst*(root, a[i]);

Assert.*assertTrue*(imbt.deepestLeftNode(root).t == 10);

}

/\*\* 31. Extract Leaves of a Binary Tree in a Doubly Linked List \*\*/

@Test

**public** **void** extractLeafNodeAsListTest() {

Integer a[] = { 7, 4, 9, 6, 5, 3, 2, 9, 8, 12, 11, 10 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*createBst*(root, a[i]);

root = imbt.extractLeafNodeAsList(root);

**while** (root != **null**) {

System.***out***.print(root.t + "->");

root = root.next;

}

}

/\*\* 32. Remove all nodes which don’t lie in any path with sum>= k \*/

@Test

**public** **void** delAllPathLessSumTest() {

Integer a[] = { 7, 4, 9, 6, 5, 3, 2, 9, 8, 12, 11, 10 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*createBst*(root, a[i]);

root = imbt.delAllPathLessSum(root, 21);

BinaryTreeUtil.*levelTravelLineByline*(root);

}

/\*\* 34. Check if all leaves are at same level \*\*/

@Test

**public** **void** checkAllLeavesAtSameLevelTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

// BinaryTreeUtil.levelTravelLineByline(root);

Assert.*assertTrue*(imbt.checkAllLeavesAtSameLevel(root));

root = BinaryTreeUtil.*create*(root, 8);

Assert.*assertTrue*(!imbt.checkAllLeavesAtSameLevel(root));

}

/\*\* 35. Find depth of the deepest odd level leaf node \*\*/

@Test

**public** **void** deepestOddlevelNodeTest() {

Integer a[] = { 7, 4, 9, 6, 5, 3, 2, 9, 8, 12, 11, 10 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*createBst*(root, a[i]);

Assert.*assertTrue*(imbt.deepestOddlevelNode(root).t == 11);

}

/\*\*

\* 36. Difference between sums of odd level and even level nodes of a Binary

\* Tree

\*\*/

@Test

**public** **void** diffSumOfOddAndEvenLevelTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

Assert.*assertTrue*(imbt.diffSumOfOddAndEvenLevel(root) == 9);

}

/\*\* 37. Custom Tree Problem \*\*/

@Test

**public** **void** createCustomTreeTest() {

Integer parent[] = { 1, 2, 2, 1 };

Integer child[] = { 2, 3, 4, 5 };

BTNode<Integer> node = **null**;

**for** (**int** i = 0; i < parent.length; i++)

node = imbt.createCustomTree(node, parent[i], child[i]);

BinaryTreeUtil.*levelTravelLineByline*(node);

}

/\*\* 38. Iterative Method to find Height of Binary Tree \*\*/

@Test

**public** **void** heightIterativeTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8 };

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < a.length; i++)

root = BinaryTreeUtil.*create*(root, a[i]);

Assert.*assertTrue*(imbt.heightIterative(root) == 3);

}

/\*\* 39. Tree Isomorphism Problem \*\*/

@Test

**public** **void** isIsomorphicTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7 };

Integer b[] = { 1, 3, 2, 7, 6, 4, 5 };

BTNode<Integer> root = **null**;

BTNode<Integer> root1 = **null**;

**for** (**int** i = 0; i < a.length; i++) {

root = BinaryTreeUtil.*create*(root, a[i]);

root1 = BinaryTreeUtil.*create*(root, b[i]);

}

Assert.*assertTrue*(imbt.isIsomorphic(root, root1));

}

}