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

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

**public** **interface** ITraversal<T> {

/\*\* 0. create binary tree level wise \*\*/

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

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

/\*\* 1. Tree Traversals \*\*/

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

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

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

/\*\* 2. Level Order Tree Traversal \*\*/

**public** **void** levelTravel(BTNode<T> t, **boolean** isLeftToRight);

/\*\* 3. Print level order traversal line by line | Set 1 \*\*/

**public** **int** height(BTNode<T> node);

**public** **void** printGivenLevel(BTNode<T> root, **int** level);

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

/\*\* 4. Inorder Tree Traversal without Recursion \*\*/

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

/\*\* 5. Inorder Tree Traversal without recursion and without stack! \*\*/

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

/\*\* 6. Iterative Preorder Traversal \*\*/

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

/\*\* 7. Morris traversal for Preorder \*\*/

// this same of 5.0

/\*\* 8. Iterative Postorder Traversal | Set 1 (Using Two Stacks) \*\*/

**public** **void** iterativePostorder2Stack(BTNode<T> t);

/\*\* 9. Iterative Postorder Traversal | Set 2 (Using One Stack) \*\*/

**public** **void** iterativePostorder1Stack(BTNode<T> t);

/\*\* 10. Reverse Level Order Traversal \*\*/

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

/\*\* 11. Print Postorder traversal from given Inorder and Preorder traversals \*\*/

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

// **TODO**

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

/\*\* 12. Level order traversal line by line | Set 2 (Using Two Queues) \*\*/

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

/\*\* 13. Diagonal Traversal of Binary Tree \*\*/

/\* from top-left to bottom-right corner \*/

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

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

/\* form top-right to bottom-left \*/

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

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

/\*\* 0. create binary tree level wise \*\*/

// @See 0.

/\*\*

\* 14. Inorder Non-threaded Binary Tree Traversal without Recursion or Stack

\*\*/

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

/\*\* 15. Check if leaf traversal of two Binary Trees is same? \*\*/

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

/\*\* 16. Print a Binary Tree in Vertical Order | Set 1 \*\*/

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

/\*\* 17. Print a Binary Tree in Vertical Order | Set 2 (Hashmap based Method) \*\*/

// @See DAIHashingWithTree.java @method printColumWise(....)

/\*\* 18. Boundary Traversal of binary tree \*\*/

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

/\* 19. Perfect Binary Tree Specific Level Order Traversal \*/

/\* 20. Perfect Binary Tree Specific Level Order Traversal | Set 2 \*/

/\*

\* 21. If you are given two traversal sequences, can you construct the binary

\* tree?

\*/

}

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

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

**public** **class** TraversalImpl **implements** ITraversal<Integer> {

/\*\* 0. create binary tree level wise \*\*/

@Override

**public** BTNode<Integer> create(BTNode<Integer> node, Integer dt) {

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

node = **new** BTNode<>(dt);

node.pt = **null**;// this will be update in else except root

} **else** {

BTNode<Integer> temp = findInsertPostion(node);

**if** (temp.lt == **null**) {

temp.lt = create(temp.lt, dt);

temp.lt.pt = temp;

}

**else** {

temp.rt = create(temp.rt, dt);

temp.rt.pt = temp;

}

}

**return** node;

}

@Override

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

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

queue.add(node);

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

BTNode<Integer> temp = queue.poll();

**if** (temp.lt == **null** || temp.rt == **null**)

**return** temp;

**else** **if** (temp.lt != **null** && temp.rt != **null**) {

queue.add(temp.lt);

queue.add(temp.rt);

}

}

**return** **null**;

}

/\*\* 1. Tree Traversals \*\*/

// these are depth first search

@Override

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

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

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

preOrder(node.lt);

preOrder(node.rt);

}

}

@Override

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

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

inOrder(node.lt);

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

// System.out.print("(" + node.ind + "::" + node.h + "::" + node.t + ")->");

inOrder(node.rt);

}

}

@Override

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

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

postOrder(node.lt);

postOrder(node.rt);

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

}

}

/\*\* 2. Level Order Tree Traversal \*\*/

@Override

**public** **void** levelTravel(BTNode<Integer> node, **boolean** isLeftToRight) {

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

que.add(node);

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

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

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

**if** (isLeftToRight) {

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

que.add(x.lt);

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

que.add(x.rt);

} **else** {

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

que.add(x.rt);

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

que.add(x.lt);

}

}

}

/\*\* 3. Print level order traversal line by line | Set 1 \*\*/

@Override

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

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

**return** 0;

}

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

}

@Override

**public** **void** printGivenLevel(BTNode<Integer> root, **int** level) {

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

**return**;

}

**if** (level == 1) {

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

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

printGivenLevel(root.lt, level - 1);

printGivenLevel(root.rt, level - 1);

}

}

@Override

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

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

System.***out***.println("No any Node present:-");

**return**;

}

**int** height = height(node);

**int** i = 1;

**while** (i <= height) {

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

printGivenLevel(node, i);

++i;

}

}

/\*\* 4. Inorder Tree Traversal without Recursion \*\*/

@Override

**public** **void** inorderWithoutRecursion(BTNode<Integer> root) {

Stack<BTNode<Integer>> stack = **new** Stack<BTNode<Integer>>();

BTNode<Integer> node = root;

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

stack.push(node);

node = node.lt;

}

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

node = stack.pop();

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

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

node = node.rt;

// the next node to be visited is the leftmost

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

stack.push(node);

node = node.lt;

}

}

}

}

/\*\* 5. Inorder Tree Traversal without recursion and without stack! \*\*/

@Override

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

BTNode<Integer> current, pre;

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

**return**;

current = node;

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

**if** (current.lt == **null**) {

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

current = current.rt;

} **else** {

/\* Find the inorder predecessor of current \*/

pre = current.lt;

**while** (pre.rt != **null** && pre.rt != current)

pre = pre.rt;

/\* Make current as right child of its inorder predecessor \*/

**if** (pre.rt == **null**) {

pre.rt = current;

current = current.lt;

}

/\*

\* Revert the changes made in if part to restore the original tree i.e., fix the

\* right child of predecssor

\*/

**else** {

pre.rt = **null**;

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

current = current.rt;

}

}

}

}

/\*\* 6. Iterative Preorder Traversal \*\*/

@Override

**public** **void** iterativePreorder(BTNode<Integer> t) {

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

Stack<BTNode<Integer>> stk = **new** Stack<>();

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

stk.add(t);

**while** (!stk.empty()) {

x = stk.pop();

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

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

stk.add(x.rt);

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

stk.add(x.lt);

}

}

/\*\* 7. Morris traversal for Preorder \*\*/

// this same of 5.0

/\*\* 8. Iterative Postorder Traversal | Set 1 (Using Two Stacks) \*\*/

@Override

**public** **void** iterativePostorder2Stack(BTNode<Integer> t) {

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

Stack<BTNode<Integer>> stk1 = **new** Stack<>();

Stack<BTNode<Integer>> stk2 = **new** Stack<>();

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

stk1.add(t);

**while** (!stk1.isEmpty()) {

x = stk1.pop();

stk2.add(x);

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

stk1.add(x.lt);

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

stk1.add(x.rt);

}

**while** (!stk2.isEmpty())

System.***out***.print("(" + stk2.pop().t + ")->");

}

/\*\* 9. Iterative Postorder Traversal | Set 2 (Using One Stack) \*\*/

@Override

**public** **void** iterativePostorder1Stack(BTNode<Integer> t) {

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

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

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

Stack<BTNode<Integer>> stk = **new** Stack<>();

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

stk.add(t);

**while** (!stk.empty()) {

x = stk.peek();

// adding in stack part

**if** ((x.lt != **null** && x.rt != **null**) && !list.contains(x.lt.t) && !list.contains(x.rt.t)) {

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

stk.add(x.rt);

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

stk.add(x.lt);

} **else** **if** ((x.lt != **null** && x.rt == **null**) && !list.contains(x.lt.t)) {

stk.add(x.lt);

} **else** **if** ((x.lt == **null** && x.rt != **null**) && !list.contains(x.rt.t)) {

stk.add(x.rt);

}

// removing from stack part

**else** **if** (x.lt == **null** && x.rt == **null**) {

y = stk.pop();

list.add(y.t);

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

} **else** **if** ((x.lt != **null** && x.rt != **null**) && list.contains(x.lt.t) && list.contains(x.rt.t)) {

y = stk.pop();

list.add(y.t);

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

} **else** **if** ((x.lt != **null** && x.rt == **null**) && list.contains(x.lt.t)) {

y = stk.pop();

list.add(y.t);

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

} **else** **if** ((x.lt == **null** && x.rt != **null**) && list.contains(x.rt.t)) {

y = stk.pop();

list.add(y.t);

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

}

}

}

/\*\* 10. Reverse Level Order Traversal \*\*/

@Override

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

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

System.***out***.println("No any Node present:-");

**return**;

}

**int** height = height(node);

**int** i = height;

**while** (i >= 0) {

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

printGivenLevel(node, i);

--i;

}

}

/\*\* 11. Print Postorder traversal from given Inorder and Preorder traversals \*\*/

@Override

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

postOrder(node);

}

// **TODO**

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

}

/\*\* 12. Level order traversal line by line | Set 2 (Using Two Queues) \*\*/

@Override

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

Queue<BTNode<Integer>> q1 = **new** LinkedList<>();

Queue<BTNode<Integer>> q2 = **new** LinkedList<>();

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

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

q1.add(node);

**while** (!q1.isEmpty()) {

**while** (!q1.isEmpty()) {

x = q1.poll();

q2.add(x);

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

}

**while** (!q2.isEmpty()) {

x = q2.poll();

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

q1.add(x.lt);

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

q1.add(x.rt);

}

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

}

}

/\*\* 13. Diagonal Traversal of Binary Tree \*\*/

/\* from top-left to bottom-right corner \*/

**public** BTNode<Integer> createforTopLeftToBtmRightTravel(BTNode<Integer> node, Integer dt) {

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

node = **new** BTNode<>(dt);

node.pt = node;// this will be update in else except root

} **else** {

BTNode<Integer> temp = findInsertPostion(node);

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

**if** (temp.lt == **null**) {

x = create(temp.lt, dt);

x.ind = temp.ind - 1;

temp.lt = x;

temp.lt.pt = temp;

}

**else** {

x = create(temp.rt, dt);

x.ind = temp.ind;

temp.rt = x;

temp.rt.pt = temp;

}

}

**return** node;

}

@Override

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

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

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

que.add(node);

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

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

list.add(x);

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

que.add(x.lt);

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

que.add(x.rt);

}

Collections.*sort*(list, BinaryTreeUtil.*indComparator*);

**int** i = list.get(0).ind;

**for** (BTNode<Integer> y : list) {

**if** (y.ind != i) {

i++;

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

}

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

}

}

/\* form top-right to bottom-left \*/

@Override

**public** BTNode<Integer> createforTopRightToBottomLeftTravel(BTNode<Integer> node, Integer dt) {

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

node = **new** BTNode<>(dt);

node.pt = node;// this will be update in else except root

} **else** {

BTNode<Integer> temp = findInsertPostion(node);

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

**if** (temp.lt == **null**) {

x = create(temp.lt, dt);

x.ind = temp.ind;

temp.lt = x;

temp.lt.pt = temp;

}

**else** {

x = create(temp.rt, dt);

x.ind = temp.ind + 1;

temp.rt = x;

temp.rt.pt = temp;

}

}

**return** node;

}

@Override

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

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

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

que.add(node);

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

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

list.add(x);

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

que.add(x.lt);

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

que.add(x.rt);

}

Collections.*sort*(list, BinaryTreeUtil.*indComparator*);

**int** i = list.get(0).ind;

**for** (BTNode<Integer> y : list) {

**if** (y.ind != i) {

i++;

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

}

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

}

}

@Override

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

**boolean** leftdone = **false**;

// Start traversal from root

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

// If left child is not traversed, find the

// leftmost child

**if** (!leftdone) {

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

node = node.lt;

}

}

// Print root's data

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

// Mark left as done

leftdone = **true**;

// If right child exists

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

leftdone = **false**;

node = node.rt;

}

// If right child doesn't exist, move to parent

**else** **if** (node.pt != **null**) {

// If this node is right child of its parent,

// visit parent's parent first

**while** (node.pt != **null** && node == node.pt.rt)

node = node.pt;

**if** (node.pt == **null**)

**break**;

node = node.pt;

} **else**

**break**;

}

}

/\*\* 15. Check if leaf traversal of two Binary Trees is same? \*\*/

@Override

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

Stack<BTNode<Integer>> stack = **new** Stack<BTNode<Integer>>();

Stack<BTNode<Integer>> stack1 = **new** Stack<BTNode<Integer>>();

Stack<BTNode<Integer>> stack2 = **new** Stack<BTNode<Integer>>();

// inorder tarverse of n1 and collecting leaf node

BTNode<Integer> node = n1;

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

stack.push(node);

node = node.lt;

}

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

node = stack.pop();

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

stack1.add(node);

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

node = node.rt;

// the next node to be visited is the leftmost

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

stack.push(node);

node = node.lt;

}

}

}

// inorder tarverse of n2 and collecting leaf node

node = n2;

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

stack.push(node);

node = node.lt;

}

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

node = stack.pop();

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

stack2.add(node);

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

node = node.rt;

// the next node to be visited is the leftmost

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

stack.push(node);

node = node.lt;

}

}

}

// checking

**if** (stack1.size() != stack2.size())

**return** **false**;

**while** (stack1.isEmpty() && stack2.isEmpty()) {

**if** (stack1.pop() != stack2.pop())

**return** **false**;

}

**return** **true**;

}

/\*\* 16. Print a Binary Tree in Vertical Order | Set 1 \*\*/

// **TODO**

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

}

/\*\* 17. Print a Binary Tree in Vertical Order | Set 2 (Hashmap based Method) \*\*/

// @See DAIHashingWithTree.java @method printColumWise(....)

/\*\* 18. Boundary Traversal of binary tree \*\*/

**public** **void** boundryTraverse(BTNode<Integer> t) {

Stack<BTNode<Integer>> stack = **new** Stack<BTNode<Integer>>();

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

// inorder tarverse of n1 and collecting leaf node

BTNode<Integer> node = t;

BTNode<Integer> x = t;

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

stack.push(node);

node = node.lt;

}

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

node = stack.pop();

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

que.add(node);

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

node = node.rt;

// the next node to be visited is the leftmost

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

stack.push(node);

node = node.lt;

}

}

}

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

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

x = x.lt;

**if** (x.lt == **null**)

**break**;

}

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

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

x = t;

**while** (x.rt != **null**)

x = x.rt;

x = x.pt;

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

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

x = x.pt;

}

}

}

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

Test case:

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

**public** **class** ITraversalTest {

**public** ITraversal<Integer> it = **null**;

@Before

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

it = **new** TraversalImpl();

}

/\*\* 1. Tree Traversals \*\*/

@Test

**public** **void** preOrderTest() {

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

it.preOrder(root);

}

@Test

**public** **void** inOrderTest() {

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

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

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

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

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

it.inOrder(root);

}

@Test

**public** **void** postOrderTest() {

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

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

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

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

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

it.postOrder(root);

}

/\*\* 2. Level Order Tree Traversal \*\*/

@Test

**public** **void** levelTravelTest() {

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

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

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

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

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

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

System.***out***.println("Left to right Traversal");

it.levelTravel(root, **true**);

System.***out***.println("\nRight to left Traversal");

it.levelTravel(root, **false**);

}

/\*\* 3. Print level order traversal line by line | Set 1 \*\*/

@Test

**public** **void** levelTravelLineBylineTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 };

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

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

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

it.levelTravelLineByline(root);

}

/\*\* 4. Inorder Tree Traversal without Recursion \*\*/

@Test

**public** **void** inorderWithoutRecursionTest() {

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

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

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

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

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

it.inorderWithoutRecursion(root);

}

/\*\* 5. Inorder Tree Traversal without recursion and without stack! \*\*/

@Test

**public** **void** inorderWithoutRecAndStackTest() {

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

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

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 };

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

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

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

it.inOrder(root);

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

it.inorderWithoutRecAndStack(root);

}

/\*\* 6. Iterative Preorder Traversal \*\*/

@Test

**public** **void** iterativePreorderTest() {

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

it.iterativePreorder(root);

}

/\*\* 7. Morris traversal for Preorder \*\*/

// this same of 5.0

/\*\* 8. Iterative Postorder Traversal | Set 1 (Using Two Stacks) \*\*/

@Test

**public** **void** iterativePostorder2StackTest() {

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

// (4)->(5)->(2)->(6)->(7)->(3)->(1)->

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

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

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

it.iterativePostorder2Stack(root);

}

/\*\* 9. Iterative Postorder Traversal | Set 2 (Using One Stack) \*\*/

@Test

**public** **void** iterativePostorder1StackTest() {

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

// (4)->(5)->(2)->(6)->(7)->(3)->(1)->

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

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

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

it.iterativePostorder1Stack(root);

}

/\*\* 10. Reverse Level Order Traversal \*\*/

@Test

**public** **void** reverseLevelOrderTraversalTest() {

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

// (4)->(5)->(2)->(6)->(7)->(3)->(1)->

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

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

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

it.reverseLevelOrderTraversal(root);

}

/\*\* 11. Print Postorder traversal from given Inorder and Preorder traversals \*\*/

@Test

**public** **void** inorderToPostorderTest() {

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

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

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

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

it.inorderToPostorder(root);

}

// **TODO**

**public** **void** preorderToPostorderTest() {

}

/\*\* 12. Level order traversal line by line | Set 2 (Using Two Queues) \*\*/

@Test

**public** **void** levelByLineTrvelTest() {

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

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

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

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

it.levelByLineTrvel(root);

}

/\*\* 13. Diagonal Traversal of Binary Tree \*\*/

/\* from top-left to bottom-right corner \*/

@Test

**public** **void** printTopLeftToBtmRightDigonallyTest() {

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

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

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

root = it.createforTopLeftToBtmRightTravel(root, a[i]);

it.printTopLeftToBtmRightDigonally(root);

}

/\* form top-right to bottom-left \*/

@Test

**public** **void** printTopRightToBottomLeftDigonallyTest() {

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

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

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

root = it.createforTopRightToBottomLeftTravel(root, a[i]);

it.printTopRightToBottomLeftDigonally(root);

}

/\*\*

\* 14. Inorder Non-threaded Binary Tree Traversal without Recursion or Stack

\*\*/

@Test

**public** **void** inorderNoUseRecurStackTest() {

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

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

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

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

it.inorderNoUseRecurStack(root);

}

/\*\* 15. Check if leaf traversal of two Binary Trees is same? \*\*/

@Test

**public** **void** isSameLeafTravelOfTwoTreeTest() {

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

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

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

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

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

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

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

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

Assert.*assertTrue*(it.isSameLeafTravelOfTwoTree(root1, root2));

}

/\*\* 16. Print a Binary Tree in Vertical Order | Set 1 \*\*/

// **TODO**

@Test

**public** **void** printVerticalOrderTest() {

}

/\*\* 17. Print a Binary Tree in Vertical Order | Set 2 (Hashmap based Method) \*\*/

// @See DAIHashingWithTree.java @method printColumWise(....)

/\*\* 18. Boundary Traversal of binary tree \*\*/

@Test

**public** **void** boundryTraverseTest() {

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

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

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

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

it.boundryTraverse(root);

}

}