public class Worker {

int key;

String name;

int age;

public Worker(int key, String name, int age) {

this.key = key;

this.name = name;

this.age = age;

}

}

class Node {

Worker data;

Node left, right, parent;

public Node(Worker worker) {

data = worker;

left = right = parent = null;

}

}

class BinarySearchTree {

private Node root;

private int count;

public BinarySearchTree() {

root = null;

count = 0;

}

// Find node with given key

public Node findNode(int key) {

Node current = root;

Node parent = null;

while (current != null && current.data.key != key) {

parent = current;

if (key < current.data.key)

current = current.left;

else

current = current.right;

}

if (current != null)

current.parent = parent;

return current;

}

// Insert new worker to tree

public void insert(Worker worker) {

Node newNode = new Node(worker);

Node current = root;

Node parent = null;

while (current != null) {

parent = current;

if (worker.key < current.data.key)

current = current.left;

else if (worker.key > current.data.key)

current = current.right;

else

return; // duplicate key

}

if (parent == null)

root = newNode;

else if (worker.key < parent.data.key)

parent.left = newNode;

else

parent.right = newNode;

count++;

}

// Output workers on tree in descending order

public void printDescendingOrder() {

printDescendingOrder(root);

}

private void printDescendingOrder(Node node) {

if (node == null)

return;

printDescendingOrder(node.right);

System.out.println(node.data.name);

printDescendingOrder(node.left);

}

// Count number of workers less than 25 years old

public int countYoungerWorkers() {

return countYoungerWorkers(root);

}

private int countYoungerWorkers(Node node) {

if (node == null)

return 0;

int count = 0;

if (node.data.age < 25)

count++;

count += countYoungerWorkers(node.left) + countYoungerWorkers(node.right);

return count;

}

// Delete right-most node of tree

public void deleteRightmostNode() {

if (root == null)

return;

if (root.right == null) {

root = root.left;

count--;

return;

}

Node parent = null;

Node current = root;

while (current.right != null) {

parent = current;

current = current.right;

}

parent.right = current.left;

count--;

}

// Determine height of tree using level order traversal

public int getHeight() {

if (root == null)

return -1;

Queue<Node> queue = new LinkedList<>();

queue.add(root);

int height = 0;

while (!queue.isEmpty()) {

int size = queue.size();

while (size-- > 0) {

Node node = queue.poll();

if (node.left != null)

queue.add(node.left);

if (node.right != null)

queue.add(node.right);

}

height++;

}

return height - 1;

}

// Create binary search tree from sequence of workers

public static BinarySearchTree createFromSequence(Worker[] workers) {

Arrays.sort(workers, new Comparator<Worker>() {

public int compare(Worker w1, Worker w2) {

return Integer.compare(w1.age, w2.age);

}

});

return createFromSequence(workers, 0, workers.length - 1);

}

private static BinarySearchTree createFromSequence(Worker[] workers, int start, int end) {

if (start > end)

return null;

int mid = (start + end) / 2;

BinarySearchTree bst = new BinarySearchTree();

bst.insert(workers[mid]);

bst.root.left = createFromSequence(workers, start, mid - 1);

bst.root.right = createFromSequence(workers, mid + 1, end);

return bst;

}

}

``