```
public class Tree extends BTreePrinter {
    Node root;
public Tree(Node root) { this.root = root;}
public Tree() {}
public void printTree() {
        if (root == null) print empty
        else super.printTree(root);    }
public static void printNode(Node node) {
        if (node == null) print not found
        else print note.key; }
public Node find(int search_key) { // recursive
        return find(root, search_key);}
public static Node find(Node node, int search_key) {
        if (node == null) return null;
        if (node.key == search_key) return node;
        else if (node.key < search_key && node.right != null)
            return find(node.right, search_key);
        else if (node.key > search_key && node.left != null)
            return find(node.left, search_key);
        else return null; }
public Node findMin() { // recursive
        return findMin(root); }
public static Node findMin(Node node) {
        if (node.left == null) return node;
        return findMin(node.left); }
public Node findMax() { // recursive
        return findMax(root); }
public static Node findMax(Node node) {
        if (node.right == null) return node;
        return findMax(node.right); }
public Node findClosestLeaf(int search_key) { //recursive
        return findClosestLeaf(root, search_key); }
public static Node findClosestLeaf(Node node, int search_key)
        if (node == null) return null;
        if (node.key > search_key) {
            if (node.left != null)
                return findClosestLeaf(node.left, search_key);
            else return node;
        } else if (node.key < search_key) {
            if (node.right != null)
                return findClosestLeaf(node.right, search_key)
            else return node;
        } return null; }
public Node findClosest(int search_key) {
        Node current, closest;
        closest = current = root;
        int min_diff = Integer.MAX_VALUE;
        while (current != null) {
            if (Math.abs(current.key - search_key) < min_diff)
                min_diff = Math.abs(current.key - search_key);
                closest = current; }
            if (current.key < search_key)
                current = current.right;
            else if (current.key > search_key)
                current = current.left;
            else break; } return closest; }
public void insert(int key) {
        Node Parent = findClosestLeaf(root, key);
        Node Child = new Node(key);
        if (Parent == null) root = Child;
        else {
            Child.parent = Parent;
            if (Parent.key > key) Parent.left = Child;
            else Parent.right = Child; } }
public void printPreOrderDFT() {
        System.out.print("PreOrder DFT node sequence [ ");
        printPreOrderDFT(root); System.out.println("]"); }
public static void printPreOrderDFT(Node node) {
        if (node == null) return;
        System.out.print(node.key + " ");
        printPreOrderDFT(node.left);printPreOrderDFT(node.right);}
public void printInOrderDFT() {
        System.out.print("InOrder DFT node sequence [ ");
        printInOrderDFT(root); System.out.println("]"); }
public static void printInOrderDFT(Node node) {
        if (node == null) return;
        printInOrderDFT(node.left);System.out.print(node.key + " ");
        printInOrderDFT(node.right); }
public void printPostOrderDFT() {
        System.out.print("PostOrder DFT node sequence [ ");
        printPostOrderDFT(root);System.out.println("]"); }
public static void printPostOrderDFT(Node node) {
        if (node == null) return;
        printPostOrderDFT(node.left); printPostOrderDFT(node.right);
        System.out.print(node.key + " "); }
public static int height(Node node) {
        if (node == null) return -1;
        return Math.max(height(node.left), height(node.right)) + 1; }
public static int size(Node node) {
        if (node == null) return 0;
        return size(node.left) + size(node.right) + 1; }
public class Node {
    Node left; Node right; Node parent; int key;
    public Node(int key) {this.key = key;} }
```

```
public class List {
    Node[] arr; int capacity; int size;
    public List(int cap) {
        arr = new Node[cap]; capacity = cap; }
    public void append(Node node) {
        if (size < capacity) { arr[size] = node; size++; } }
    public void printList() {
        System.out.print("[Head] ");
        for (int i = 0; i < size; i++) {
            System.out.print(arr[i].key + " ");
        } System.out.println("[Tail] "); } }
```

```
public static int depth(Node root, Node node) {
        if (node == null) return -1;
        if (node == root) return 0;
        return depth(root, node.parent) + 1; }
public int height() { return height(root); }
public int size() { return size(root); }
public int depth() { return height(root); }
public Node findKthSmallest(int k) {return findKthSmallest(root, k);}
public static Node findKthSmallest(Node node, int k) {
        int l = size(node.left);
        if (k == l + 1) return node;
        else if (k < l + 1) return findKthSmallest(node.left, k);
        else return findKthSmallest(node.right, k - l - 1); }
public static Node findNext(Node node) {
        if (node.right != null) return leftDescendant(node.right);
        else return rightAncestor(node); }
public static Node leftDescendant(Node node) {
        if (node.left == null) return node;
        else return leftDescendant(node.left); }
public static Node rightAncestor(Node node) {
        if (node.parent == null) return null;
        if (node.key < node.parent.key) return node.parent;
        else return rightAncestor(node.parent); }
public List rangeSearch(int x, int y) {
        List L = new List(100); Node N = findClosest(x);
        while (N != null && N.key <= y) {
            if (N.key >= x) L.append(N);
            N = findNext(N); } return L; }
public void delete(int key) {
        Node del = find(key);
        if (root == null) System.out.println("Empty Tree!!!");
        else if (del == null) System.out.println("Key not found!!!");
        else if (del == root) {
            if (root.left == null && root.right == null) root = null;
            else if (root.left != null && root.right == null) {
                Node newRoot = root.left; newRoot.parent = null;
                root.left = null; root = newRoot; }
            else if (root.left == null && root.right != null) {
                Node newRoot = root.right; newRoot.parent = null;
                root.right = null; root = newRoot; }
            else {
                Node min = findMin(root.right); int min_value = min.key;
                delete(min); root.key = min_value; } }
        else {delete(del);} }
public static void delete(Node node) {
        if (node.left == null && node.right == null) {
            Node par = node.parent; node.parent = null;
            if (par.key < node.key) par.right = null;
            else par.left = null; }
        else if (node.left == null && node.right != null) {
            Node par = node.parent; Node child = node.right;
            node.parent = null; node.right = null;
            if (par.key < node.key) par.right = child;
            else par.left = child; <-- ดูผลลัพธ์ๆ else นี้มีแค่บรรทัดเดียว
            child.parent = par; }
        else if (node.left != null && node.right == null) {
            Node par = node.parent; Node child = node.left;
            node.parent = null; node.right = null;
            if (par.key < node.key) par.right = child;
            else par.left = child; <-- ดูผลลัพธ์ๆ else นี้มีแค่บรรทัดเดียว
            child.parent = par; }
        else {
            Node min = findMin(node.right); int min_value = min.key;
            delete(min); node.key = min_value; } } }
```

```
AVLTree.java
public class AVLTree extends BTreePrinter {
    Node root;
public AVLTree(Node root) {this.root = root; root.parent = null;}
public void singleRotateFromLeft(Node y) {}
public void singleRotateFromRight(Node y) {}
public void doubleRotateFromLeft(Node y) {}
public void doubleRotateFromRight(Node y) {}
public static void rebalance(AVLTree tree, Node node) {
    if (node == null) return;
    int balanceFactor = height(node.left) - height(node.right);
    if (Math.abs(balanceFactor) > 1) {
        if (balanceFactor > 0) { // left-heavy
            int balanceFactor_next = height(node.left.left) - height(node.left.right);
            if (balanceFactor_next > 0) {
                tree.singleRotateFromLeft(node);
                System.out.println("Perform SingleRotationFromLeft(Node " + node.key + ")");
            } else {
                tree.doubleRotateFromLeft(node);
                System.out.println("Perform DoubleRotationFromLeft(Node " + node.key + ")"
            } else {
                int balanceFactor_next = height(node.right.left) - height(node.right.right);
                if (balanceFactor_next > 0) {
                    tree.doubleRotateFromRight(node);
                    System.out.println("Perform DoubleRotationFromRight(Node " + node.key + ")");
                } else {
                    tree.singleRotateFromRight(node);
                    System.out.println("Perform SingleRotationFromRight(Node " + node.key + ")"); } } } }
```

```
BSTree.java
public class BSTree extends BTreePrinter {
    Node root;
public void singleRotateFromLeft(Node y) {
    Node x = y.left; Node w = y.parent; Node b = x.right; y.left = b;
        if (b != null) b.parent = y;
        y.parent = x; x.right = y;
        if (w != null) { x.parent = w;
            if (w.key < x.key) w.right = x;
            else w.left = x; }
        else { root = x; root.parent = null; } }
public void singleRotateFromRight(Node y) {
        Node x = y.right; Node w = y.parent; Node b = x.left; y.right = b;
        if (b != null) b.parent = y;
        y.parent = x; x.left = y;
        if (w != null) { x.parent = w;
            if (w.key < x.key) w.right = x;
            else w.left = x; }
        else { root = x; root.parent = null; } }
public void doubleRotateFromLeft(Node y) {
        Node x = y.left; Node z = x.right; Node w = y.parent;
        Node b = z.left; Node c = z.right; x.right = b;
        if (b != null) b.parent = x;
        y.left = c;
        if (c != null) c.parent = y;
        z.left = x; z.right = y; x.parent = z; y.parent = z;
        if (w != null) { z.parent = w;
            if (z.key < w.key) w.left = z;
            else w.right = z; }
        else { root = z; root.parent = null; } }
public void doubleRotateFromRight(Node y) {
        Node x = y.right; Node z = x.left; Node w = y.parent;
        Node b = z.left; Node c = z.right; y.right = b;
        if (b != null) b.parent = y;
        x.left = c;
        if (c != null) c.parent = x;
        z.left = y; z.right = x; x.parent = z; y.parent = z;
        if (w != null) { .parent = w;
            if (z.key < w.key) w.left = z;
            else w.right = z; }
        else { root = z; root.parent = null; } }
public Node find(int search_key) { //copy hw6 }
public static Node find(Node node, int search_key) { //copy hw6 }
public static Node findMin(Node node){ //copy hw6 }
public static Node findMax(Node node){ //copy hw6 }
public void insert(int key){ //copy hw6 }
public static void insert(Node node, int key){ //copy hw6 }
public void delete(int key){ //copy hw6 }
public static void delete(Node node){ //copy hw6 }
public static boolean isMergeable(Node r1, Node r2) {
        Node maxR1 = findMax(r1); Node minR2 = findMin(r2);
        return maxR1.key < minR2.key; }
public static Node mergeWithRoot(Node r1, Node r2, Node t) {
        if (isMergeable(r1, r2)) { t.left = r1; t.right = r2;
            if (r1.parent != null) r1.parent = t;
            if (r2.parent != null) r2.parent = t; return t; }
        else {
            System.out.println("All nodes in T1 must be smaller than all nodes from T2");
            return r1; } }
public void merge(BSTree tree2) {
        if (isMergeable(this.root, tree2.root)) {
            Node maxR1 = findMax(this.root); Node t = new Node(maxR1.key);
            delete(maxR1); this.root = mergeWithRoot(this.root, tree2.root, t);
        } else {
            System.out.println("All nodes in T1 must be smaller than all nodes from T2"); } }

public void printTree() {
        if (root == null) { System.out.println("Empty tree!!!");
        } else { super.printTree(root); } }
} // End BSTree.java
```

```
public static Node mergeWithRoot(Node r1, Node r2, Node t) {
    if (isMergeable(r1, r2)) {
        if (Math.abs(height(r1) - height(r2)) <= 1) {
            t.left = r1;
            if (r1 != null) r1.parent = t;
            t.right = r2;
            if (r2 != null) r2.parent = t;
            return t;
        } else if (height(r1) > height(r2)) {
            Node sub_root = mergeWithRoot(r1.right, r2, t); r1.right = sub_root;
            if (sub_root != null) sub_root.parent = r1;
            r1.parent = null; AVLTree newRoot = new AVLTree(r1);
            rebalance(newRoot, newRoot.root); return newRoot.root;
        } else if (height(r1) < height(r2)) {
            Node sub_root = mergeWithRoot(r1, r2.left, t); r2.left = sub_root;
            if (sub_root != null) sub_root.parent = r2;
            r2.parent = null; AVLTree newRoot = new AVLTree(r2);
            rebalance(newRoot, newRoot.root); return newRoot.root; } return null;
    } else {
        System.out.println("All nodes in T1 must be smaller than all nodes from T2");
        return r1; } }
```

```
public void insert(int key) {
    if (root == null) { root = new Node(key);
    } else { insert(this, root, key); } }
public static void insert(AVLTree tree, Node node, int key) {
    if (key == node.key) {
        System.out.println("Duplicated key:" + key);
    } else if (key < node.key) {
        if (node.left == null) { node.left = new Node(key); node.left.parent = node;
        } else { insert(tree, node.left, key); rebalance(tree, node); }
    } else {
        if (node.right == null) { node.right = new Node(key); node.right.parent = node;
        } else { insert(tree, node.right, key); rebalance(tree, node); } } }
public void delete(int key) {
    Node del = find(key);
    if (root == null) System.out.println("Empty Tree!!!");
    else if (del == null) System.out.println("Key not found!!!");
    else if (del == root) {
        if (root.left == null && root.right == null) root = null;
        else if (root.left != null && root.right == null) {
            Node newRoot = root.left; newRoot.parent = null;
            root.left = null; root = newRoot; }
        else if (root.left == null && root.right != null) {
            Node newRoot = root.right; newRoot.parent = null;
            root.right = null; root = newRoot; }
        else {
            Node min = findMin(root.right); int min_value = min.key;
            root.key = min_value; delete(this, min); } }
    else { delete(this, del); } }
public static void delete(AVLTree tree, Node node) {
    Node current = node.parent;
    if (node.left == null && node.right == null) {
        Node par = node.parent;
        if (par.right != null && par.right.key == node.key) par.right = null;
        else par.left = null;
        node.parent = null; }
    else if (node.left == null && node.right != null) {
        Node par = node.parent; Node child = node.right;
        node.parent = null; node.right = null;
        if (par.right != null && par.right.key == node.key) par.right = child;
        else par.left = child;
        child.parent = par; }
    else if (node.left != null && node.right == null) {
        Node par = node.parent; Node child = node.left;
        node.parent = null; node.right = null;
        if (par.right != null && par.right.key == node.key) par.right = child;
        else par.left = child;
        child.parent = par; }
    else {
        Node min = findMin(node.right); int min_value = min.key;
        node.key = min_value; delete(tree, min);
    while (current != null) { rebalance(tree, current); current = current.parent; }}
public Node find(int search_key) { return find(root, search_key); }
```

```
public static Node find(Node node, int search_key) {
    if (node == null) return null;
    if (node.key == search_key) return node;
    else if (node.key < search_key && node.right != null)
        return find(node.right, search_key);
    else if (node.key > search_key && node.left != null)
        return find(node.left, search_key);
    else return null; }
public static Node findMin(Node node) {
    if (node.left == null) return node;
    return findMin(node.left); }
public static Node findMax(Node node) {
    if (node.right == null) return node;
    return findMax(node.right); }
public static boolean isMergeable(Node r1, Node r2) {
    if (r1 == null || r2 == null) return true;
    int max_left_tree = findMax(r1).key;
    int min_right_tree = findMin(r2).key;
    return max_left_tree < min_right_tree; }
```

```
public void merge(AVLTree tree2) {
    if (isMergeable(this.root, tree2.root)) { // ทำmerge ได้
        Node maxR1 = findMax(this.root); Node t = new Node(maxR1.key);
        delete(this, maxR1); root = mergeWithRoot(this.root, tree2.root, t);
    } else {
        System.out.println("All nodes in T1 must be smaller than all nodes from T2"); } }
public Node[] split(int key) {return split(root, key);}
public Node[] split(Node r, int key) {
    Node[] roots = new Node[2];
    if (r == null) { roots[0] = roots[1] = null; }
    else if (r.key <= key) { roots = split(r.right, key);
        roots[0] = mergeWithRoot(r.left, roots[0], r);
    } else {
        roots = split(r.left, key); roots[1] = mergeWithRoot(roots[1], r.right, r);
    } return roots; }
public static int height(Node node) {
    if (node == null) return -1;
    else return 1 + Math.max(height(node.left), height(node.right)); }
public void printTree() {
    if (root == null) { System.out.println("Empty tree!!!");
    } else { super.printTree(root); } }
public AVLTree() {} // Dummy Constructor, no need to edit
} End AVLTree.java
```