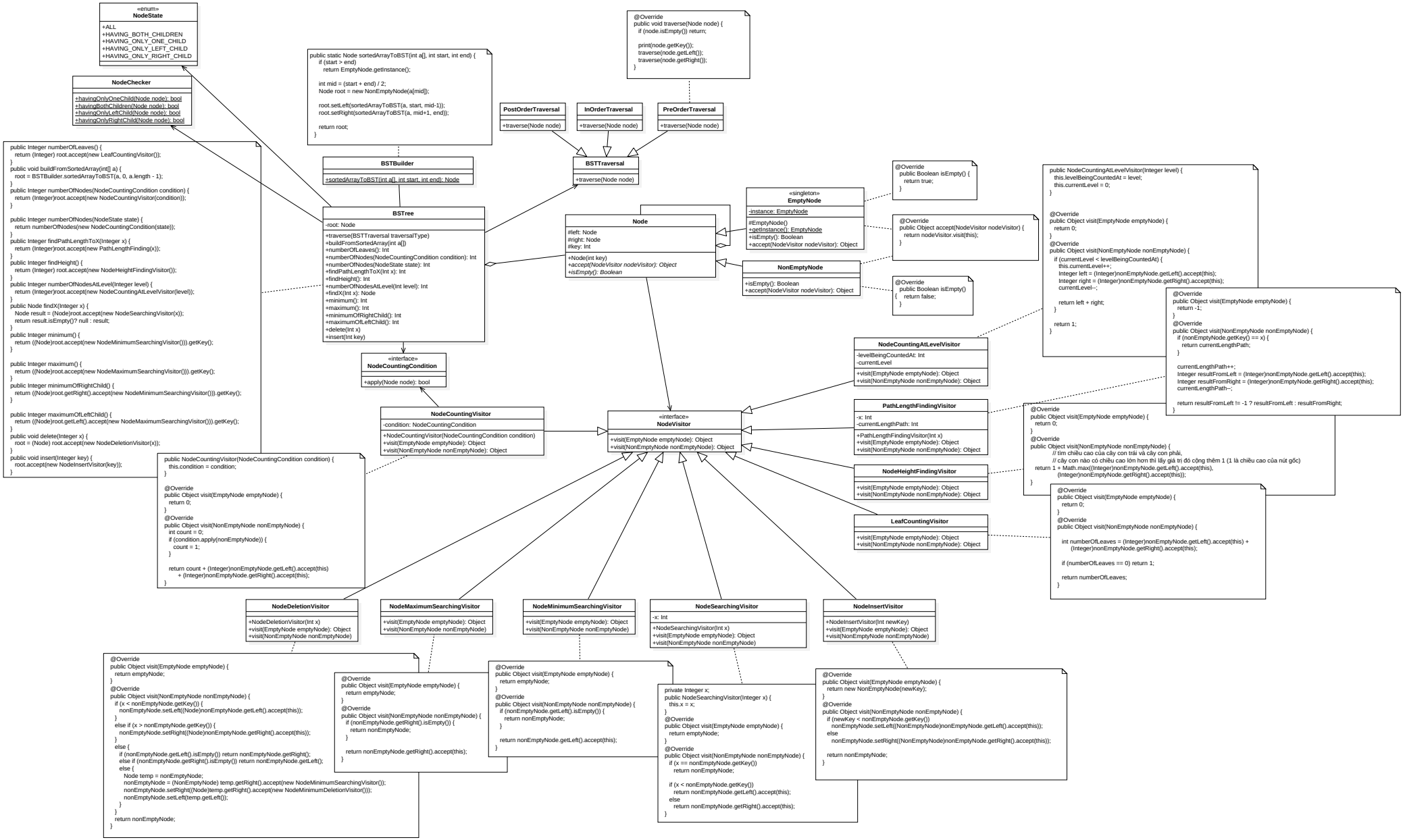
Model::Main


«enum»
NodeState
+ALL
+HAVING_BOTH_CHILDREN
+HAVING_ONLY_ONE_CHILD
+HAVING_ONLY_LEFT_CHILD
+HAVING_ONLY_RIGHT_CHILD
NodeChecker
+havingOnlyOneChild(Node node): bool
+havingBothChildren(Node node): bool
+havingOnlyLeftChild(Node node): bool
+havingOnlyRightChild(Node node): bool
public Integer numberOfLeaves() {
return (Integer) root.accept(new LeafCountingVisitor());
}
public void buildFromSortedArray(int[] a) {
root = BSTBuilder.sortedArrayToBST(a, 0, a.length - 1);
}
public Integer numberOfNodes(NodeCountingCondition condition) {
return (Integer)root.accept(new NodeCountingVisitor(condition));
}
public Integer numberOfNodes(NodeState state) {
return numberOfNodes(new NodeCountingCondition(state));
}
public Integer findPathLengthToX(Integer x) {
return (Integer)root.accept(new PathLengthFinding(x));
}
public Integer findHeight() {
return (Integer) root.accept(new NodeHeightFindingVisitor());
}
public Integer numberOfNodesAtLevel(Integer level) {
return (Integer)root.accept(new NodeCountingAtLevelVisitor(level));
}
public Node findX(Integer x) {
Node result = (Node)root.accept(new NodeSearchingVisitor(x));
return result.isEmpty()? null : result;
}
public Integer minimum() {
return ((Node)root.accept(new NodeMinimumSearchingVisitor())).getKey();
}
public Integer maximum() {
return ((Node)root.accept(new NodeMaximumSearchingVisitor())).getKey();
}
public Integer minimumOfRightChild() {
return ((Node)root.getRight().accept(new NodeMinimumSearchingVisitor())).getKey();
}
public Integer maximumOfLeftChild() {
return ((Node)root.getLeft().accept(new NodeMaximumSearchingVisitor())).getKey();
}
public void delete(Integer x) {
root = (Node) root.accept(new NodeDeletionVisitor(x));
}
public void insert(Integer key) {
root.accept(new NodeInsertVisitor(key));
}
public static Node sortedArrayToBST(int a[], int start, int end) {
if (start > end)
return EmptyNode.getInstance();
int mid = (start + end) / 2;
Node root = new NonEmptyNode(a[mid]);
root.setLeft(sortedArrayToBST(a, start, mid-1));
root.setRight(sortedArrayToBST(a, mid+1, end));
return root;
}
BSTBuilder
+sortedArrayToBST(int a[], int start, int end): Node
BSTTree
-root: Node
+traverse(BSTTraversal traversalType)
+buildFromSortedArray(int a[])
+numberOfLeaves(): Int
+numberOfNodes(NodeCountingCondition condition): Int
+numberOfNodes(NodeState state): Int
+findPathLengthToX(Int x): Int
+findHeight(): Int
+numberOfNodesAtLevel(Int level): Int
+findX(Int x): Node
+minimum(): Int
+maximum(): Int
+minimumOfRightChild(): Int
+maximumOfLeftChild(): Int
+delete(Int x)
+insert(Int key)
«interface»
NodeCountingCondition
+apply(Node node): bool
NodeCountingVisitor
-condition: NodeCountingCondition
+NodeCountingVisitor(NodeCountingCondition condition)
+visit(EmptyNode emptyNode): Object
+visit(NonEmptyNode nonEmptyNode): Object
public NodeCountingVisitor(NodeCountingCondition condition) {
this.condition = condition;
}
@Override
public Object visit(EmptyNode emptyNode) {
return 0;
}
@Override
public Object visit(NonEmptyNode nonEmptyNode) {
int count = 0;
if (condition.apply(nonEmptyNode)) {
count = 1;
}
return count + (Integer)nonEmptyNode.getLeft().accept(this)
+ (Integer)nonEmptyNode.getRight().accept(this);
}
PostOrderTraversal
+traverse(Node node)
InOrderTraversal
+traverse(Node node)
PreOrderTraversal
+traverse(Node node)
@Override
public void traverse(Node node) {
if (node.isEmpty()) return;
print(node.getKey());
traverse(node.getLeft());
traverse(node.getRight());
}
BSTTraversal
+traverse(Node node)
Node
#left: Node
#right: Node
#key: Int
+Node(int key)
+accept(NodeVisitor nodeVisitor): Object
+isEmpty(): Boolean
«singleton»
EmptyNode
-instance: EmptyNode
#EmptyNode()
+getInstance(): EmptyNode
+isEmpty(): Boolean
+accept(NodeVisitor nodeVisitor): Object
@Override
public Boolean isEmpty() {
return true;
}
@Override
public Object accept(NodeVisitor nodeVisitor) {
return nodeVisitor.visit(this);
}
NonEmptyNode
+isEmpty(): Boolean
+accept(NodeVisitor nodeVisitor): Object
@Override
public Boolean isEmpty()
{ return false;
}
«interface»
NodeVisitor
+visit(EmptyNode emptyNode): Object
+visit(NonEmptyNode nonEmptyNode): Object
NodeCountingAtLevelVisitor
-levelBeingCountedAt: Int
-currentLevel
+visit(EmptyNode emptyNode): Object
+visit(NonEmptyNode nonEmptyNode): Object
public NodeCountingAtLevelVisitor(Integer level) {
this.levelBeingCountedAt = level;
this.currentLevel = 0;
}
@Override
public Object visit(EmptyNode emptyNode) {
return 0;
}
@Override
public Object visit(NonEmptyNode nonEmptyNode) {
if (currentLevel < levelBeingCountedAt) {
this.currentLevel++;
Integer left = (Integer)nonEmptyNode.getLeft().accept(this);
Integer right = (Integer)nonEmptyNode.getRight().accept(this);
currentLevel--;
return left + right;
}
return 1;
}
PathLengthFindingVisitor
-x: Int
-currentLengthPath: Int
+PathLengthFindingVisitor(Int x)
+visit(EmptyNode emptyNode): Object
+visit(NonEmptyNode nonEmptyNode): Object
@Override
public Object visit(EmptyNode emptyNode) {
return 0;
}
@Override
public Object visit(NonEmptyNode nonEmptyNode) {
if (nonEmptyNode.getKey() == x) {
return currentLengthPath;
}
currentLengthPath++;
Integer resultFromLeft = (Integer)nonEmptyNode.getLeft().accept(this);
Integer resultFromRight = (Integer)nonEmptyNode.getRight().accept(this);
currentLengthPath--;
return resultFromLeft != -1 ? resultFromLeft : resultFromRight;
}
NodeHeightFindingVisitor
+visit(EmptyNode emptyNode): Object
+visit(NonEmptyNode nonEmptyNode): Object
@Override
public Object visit(EmptyNode emptyNode) {
return 0;
}
@Override
public Object visit(NonEmptyNode nonEmptyNode) {
// tìm chiều cao của cây con trái và cây con phải,
// cây con nào có chiều cao lớn hơn thì lấy giá trị đó cộng thêm 1 (1 là chiều cao của nút gốc)
return 1 + Math.max((Integer)nonEmptyNode.getLeft().accept(this),
(Integer)nonEmptyNode.getRight().accept(this));
}
LeafCountingVisitor
+visit(EmptyNode emptyNode): Object
+visit(NonEmptyNode nonEmptyNode): Object
@Override
public Object visit(EmptyNode emptyNode) {
return 0;
}
@Override
public Object visit(NonEmptyNode nonEmptyNode) {
int numberOfLeaves = (Integer)nonEmptyNode.getLeft().accept(this) +
(Integer)nonEmptyNode.getRight().accept(this);
if (numberOfLeaves == 0) return 1;
return numberOfLeaves;
}
NodeDeletionVisitor
+NodeDeletionVisitor(Int x)
+visit(EmptyNode emptyNode): Object
+visit(NonEmptyNode nonEmptyNode)
@Override
public Object visit(EmptyNode emptyNode) {
return emptyNode;
}
@Override
public Object visit(NonEmptyNode nonEmptyNode) {
if (x < nonEmptyNode.getKey()) {
nonEmptyNode.setLeft((Node)nonEmptyNode.getLeft().accept(this));
}
else if (x > nonEmptyNode.getKey()) {
nonEmptyNode.setRight((Node)nonEmptyNode.getRight().accept(this));
}
else {
if (nonEmptyNode.getLeft().isEmpty()) return nonEmptyNode.getRight();
else if (nonEmptyNode.getRight().isEmpty()) return nonEmptyNode.getLeft();
else {
Node temp = nonEmptyNode;
nonEmptyNode = (NonEmptyNode) temp.getRight().accept(new NodeMinimumSearchingVisitor());
nonEmptyNode.setRight((Node)temp.getRight().accept(new NodeMinimumDeletionVisitor()));
nonEmptyNode.setLeft(temp.getLeft());
}
}
return nonEmptyNode;
}
NodeMaximumSearchingVisitor
+visit(EmptyNode emptyNode): Object
+visit(NonEmptyNode nonEmptyNode)
@Override
public Object visit(EmptyNode emptyNode) {
return emptyNode;
}
@Override
public Object visit(NonEmptyNode nonEmptyNode) {
if (nonEmptyNode.getRight().isEmpty()) {
return nonEmptyNode;
}
return nonEmptyNode.getRight().accept(this);
}
NodeMinimumSearchingVisitor
+visit(EmptyNode emptyNode): Object
+visit(NonEmptyNode nonEmptyNode)
@Override
public Object visit(EmptyNode emptyNode) {
return emptyNode;
}
@Override
public Object visit(NonEmptyNode nonEmptyNode) {
if (nonEmptyNode.getLeft().isEmpty()) {
return nonEmptyNode;
}
return nonEmptyNode.getLeft().accept(this);
}
NodeSearchingVisitor
-x: Int
+NodeSearchingVisitor(Int x)
+visit(EmptyNode emptyNode): Object
+visit(NonEmptyNode nonEmptyNode)
private Integer x;
public NodeSearchingVisitor(Integer x) {
this.x = x;
}
@Override
public Object visit(EmptyNode emptyNode) {
return emptyNode;
}
@Override
public Object visit(NonEmptyNode nonEmptyNode) {
if (x == nonEmptyNode.getKey())
return nonEmptyNode;
if (x < nonEmptyNode.getKey())
return nonEmptyNode.getLeft().accept(this);
else
return nonEmptyNode.getRight().accept(this);
}
NodeInsertVisitor
+NodeInsertVisitor(Int newKey)
+visit(EmptyNode emptyNode): Object
+visit(NonEmptyNode nonEmptyNode)
@Override
public Object visit(EmptyNode emptyNode) {
return new NonEmptyNode(newKey);
}
@Override
public Object visit(NonEmptyNode nonEmptyNode) {
if (newKey < nonEmptyNode.getKey())
nonEmptyNode.setLeft((NonEmptyNode)nonEmptyNode.getLeft().accept(this));
else
nonEmptyNode.setRight((NonEmptyNode)nonEmptyNode.getRight().accept(this));
return nonEmptyNode;
}