**Data Structure and Algorithm Practicals**

8. Practical based on binary search tree implementation with its operations

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<script src="n.js"></script>

<title>Document</title>

</head>

<body>

</body>

</html>

class Node {

constructor(value = null, left = null, right = null) {

this.value = value;

this.right = right;

this.left = left;

}

toString() {

return JSON.stringify(this);

}

}

class BinarySearchTree {

constructor() {

this.root = null;

}

/\*

\* A recursive in-order traversal. Takes a callback function, process, which is applied to each node.

\*/

printInOrder(process) {

let inOrder = (node) => {

if (node.left !== null) {

inOrder(node.left);

}

process.call(this, node);

if (node.right !== null) {

inOrder(node.right);

}

};

inOrder(this.root);

}

/\*

\* A recursive pre-order traversal.

\*/

printPreOrder(process) {

let preOrder = (node) => {

process.call(this, node);

if (node.left !== null) {

preOrder(node.left);

}

if (node.right !== null) {

preOrder(node.right);

}

}

preOrder(this.root);

}

/\*

\* A recursive post-order traversal.

\*/

printPostOrder(process) {

let postOrder = (node) => {

if (node.left !== null) {

postOrder(node.left);

}

if (node.right !== null) {

postOrder(node.right);

}

process.call(this, node);

}

postOrder(this.root);

}

traverseBFS() {

let result = []

let queue = [this.root];

while (queue.length > 0) {

let node = queue.shift();

result.push(node.value);

if (node.left) {

queue.push(node.left);

}

if (node.right) {

queue.push(node.right);

}

}

return result;

}

traverseZigZag() {

let stack = [this.root];

// store next level node in nextLevel because order changes

let nextLevel = [];

let fromLeft = true;

let result = [];

while(stack.length) {

let len = stack.length;

for (let i=0; i<len; i++) {

let el = stack.pop();

result.push(el.value);

if (fromLeft) {

el.left && nextLevel.push(el.left);

el.right && nextLevel.push(el.right);

} else {

el.right && nextLevel.push(el.right);

el.left && nextLevel.push(el.left);

}

}

fromLeft = !fromLeft;

stack = nextLevel;

nextLevel = [];

}

return result;

}

/\*

\* Searches for a value in the tree and returns a node.

\*/

find(value) {

let traverse = (node) => {

if (node == null || node.value === value) {

return node;

} else if (value < node.value) {

traverse(node.left);

} else {

traverse(node.right);

}

};

return traverse(this.root);

}

/\*

\* Takes a value to insert into the tree.

\*/

insert(value) {

if (this.root === null) {

this.root = new Node(value);

} else {

let current = this.root;

while (true) {

if (value > current.value) {

if (current.right === null) {

current.right = new Node(value);

break;

} else {

current = current.right;

}

} else if (value < current.value) {

if (current.left === null) {

current.left = new Node(value);

break;

} else {

current = current.left;

}

}

}

}

}

/\*

\* get min value from the tree.

\*/

getMin(node = this.root) {

while(node.left) {

node = node.left;

}

return node.value;

}

/\*

\* get min value from the tree.

\*/

getMax(node = this.root) {

while(node.right) {

node = node.right;

}

return node.value;

}

/\*

\* Remove value from the tree.

\*/

remove(val, node = this.root) {

if (!node) {

return null;

}

if (val < node.value) {

node.left = this.remove(val, node.left);

} else if (val > node.value) {

node.right = this.remove(val, node.right);

} else {

if (!node.left) {

return node.right;

} else if (!node.right) {

return node.left;

} else {

node.value = this.getMin(node.right);

node.right = this.remove(node.value, node.right);

}

}

return node;

}

/\*\*

\* Find the least /lowest common ancestor of two value

\*/

leastCommonAncestor(n1, n2) {

if (this.root == null) {

return this.root;

}

let queue = [this.root];

while (queue.length) {

let root = queue.shift();

if (root.value === n1.value ||

root.value === n2.value ||

(root.value >= n1.value && root.value <= n2.value) ||

(root.value <= n1.value && root.value >= n2.value)

){

return root;

} else {

if(root.value > n1.value && root.value > n2.value) {

root.left && queue.push(root.left);

} else {

root.right && queue.push(root.right);

}

}

}

return null;

}

findHeight(root = this.root) {

let height = (node) => {

if (node === null) {

return -1;

}

let lefth = height(node.left);

let righth = height(node.right);

return 1 + Math.max(lefth, righth);

}

return height(root);

}

/\*

\* check if binary tree is balanced or not

\*/

isBalanced(){

let balanced = function(node) {

if (node === null) { // Base case

return true;

}

let heightDifference = Math.abs(this.findHeight(node.left) - this.findHeight(node.right));

if (heightDifference > 1) {

return false;

} else {

return balanced(node.left) && balanced(node.right);

}

}

return balanced(this.root);

}

/\*

\* Returns a boolean indicating whether a given value is contained in the tree.

\*/

contains(value) {

return !!this.find(value);

}

/\*

\* Returns an integer indicating the number of nodes in the tree.

\*/

size() {

let length = 0;

this.printInOrder(() => {

length++;

});

return length;

}

/\*

\* Returns an array containing the tree's nodes, in ascending order.

\*/

toArray() {

let arr = [];

this.printInOrder((node) => {

arr.push(node.value);

});

return arr;

}

/\*

\* Returns the tree in order as a serialized JSON string.

\*/

toString() {

let str = '';

this.printInOrder((node) => {

str += JSON.stringify(node.value) + '\n';

});

return str;

}

/\*

\* Returns the node with the nth-largest value in the tree.

\*/

nthLargest(n) {

let arr = this.toArray();

return arr[arr.length - (n + 1)];

}

/\*

\* Returns the node with the nth-smallest value in the tree.

\*/

nthSmallest(n) {

let arr = this.toArray();

return arr[n];

}

}

var tree = new BinarySearchTree();

tree.insert(6);

tree.insert(2);

tree.insert(8);

tree.insert(0);

tree.insert(4);

tree.insert(7);

tree.insert(9);

tree.insert(3);

tree.insert(5);

console.log(tree.findHeight());

console.log(tree.leastCommonAncestor(new Node(2),new Node(8)));

console.log(tree.leastCommonAncestor(new Node(2),new Node(4)));

console.log(tree.toArray()); // [0, 2, 3, 4, 5, 6, 7, 8, 9]

console.log(tree.nthLargest(1)) // second largest

console.log(tree.nthLargest(0)) // largest

console.log(tree.traverseZigZag()); // [6, 8, 2, 0, 4, 7, 9, 5, 3]

console.log(tree.traverseBFS()); // 6, 2, 8, 0, 4, 7, 9, 3, 5]

console.log(tree.remove(4))

console.log(tree.traverseBFS()); // [6, 2, 8, 0, 5, 7, 9, 3

console.log(tree.getMin()); // 0

console.log(tree.getMax()); //9