Name: Deep Suryavanshi

Roll No: 20220201077

MCA-1

* Binary Tree Implementation

class BinaryTreeNode

{

constructor(key, value = key, parent = null)

{

this.key = key;

this.value = value;

this.parent = parent;

this.left = null;

this.right = null;

}

get isLeaf()

{

return this.left === null && this.right === null;

}

get hasChildren()

{

return !this.isLeaf;

}

}

class BinaryTree

{

constructor(key, value = key)

{

this.root = new BinaryTreeNode(key, value);

}

\*inOrderTraversal(node = this.root)

{

if (node.left) yield\* this.inOrderTraversal(node.left);

yield node;

if (node.right) yield\* this.inOrderTraversal(node.right);

}

\*postOrderTraversal(node = this.root)

{

if (node.left) yield\* this.postOrderTraversal(node.left);

if (node.right) yield\* this.postOrderTraversal(node.right);

yield node;

}

\*preOrderTraversal(node = this.root)

{

yield node;

if (node.left) yield\* this.preOrderTraversal(node.left);

if (node.right) yield\* this.preOrderTraversal(node.right);

}

insert(

parentNodeKey,

key,

value = key,

{ left, right } = { left: true, right: true }

)

{

for (let node of this.preOrderTraversal())

{

if (node.key === parentNodeKey)

{

const canInsertLeft = left && node.left === null;

const canInsertRight = right && node.right === null;

if (!canInsertLeft && !canInsertRight) return false;

if (canInsertLeft)

{

node.left = new BinaryTreeNode(key, value, node);

return true;

}

if (canInsertRight)

{

node.right = new BinaryTreeNode(key, value, node);

return true;

}

}

}

return false;

}

remove(key)

{

for (let node of this.preOrderTraversal())

{

if (node.left.key === key)

{

node.left = null;

return true;

}

if (node.right.key === key)

{

node.right = null;

return true;

}

}

return false;

}

find(key)

{

for (let node of this.preOrderTraversal())

{

if (node.key === key) return node;

}

return undefined;

}

}

const tree = new BinaryTree(1, 'AB');

tree.insert(1, 11, 'AC');

tree.insert(1, 12, 'BC');

tree.insert(12, 121, 'BG', { right: true });

[...tree.preOrderTraversal()].map(x => x.value);

// ['AB', 'AC', 'BC', 'BCG']

[...tree.inOrderTraversal()].map(x => x.value);

// ['AC', 'AB', 'BC', 'BG']

tree.root.value; // 'AB'

tree.root.hasChildren; // true

tree.find(12).isLeaf; // false

tree.find(121).isLeaf; // true

tree.find(121).parent.value; // 'BC'

tree.find(12).left; // null

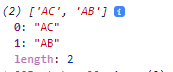
tree.find(12).right.value; // 'BG'

tree.remove(12);

[...tree.postOrderTraversal()].map(x => x.value);

// ['AC', 'AB']

OutPut:-



* Implementation of binary Search tree

class Node

{

constructor(data, left = null, right = null)

{

this.data = data;

this.left = left;

this.right = right;

}

}

class BST

{

constructor()

{

this.root = null;

}

add(data)

{

const node = this.root;

if (node === null)

{

this.root = new Node(data);

return;

}

else

{

const searchTree = function(node)

{

if (data < node.data)

{

if (node.left === null)

{

node.left = new Node(data);

return;

}

else if (node.left !== null)

{

return searchTree(node.left);

}

}

else if (data > node.data)

{

if (node.right === null)

{

node.right = new Node(data);

return;

}

else if (node.right !== null)

{

return searchTree(node.right);

}

}

else

{

return null;

}

};

return searchTree(node);

}

}

findMin()

{

let current = this.root;

while (current.left !== null)

{

current = current.left;

}

return current.data;

}

findMax()

{

let current = this.root;

while (current.right !== null)

{

current = current.right;

}

return current.data;

}

find(data)

{

let current = this.root;

while (current.data !== data)

{

if (data < current.data) {

current = current.left;

}

else

{

current = current.right;

}

if (current === null)

{

return null;

}

}

return current;

}

isPresent(data)

{

let current = this.root;

while (current)

{

if (data === current.data) {

return true;

}

if (data < current.data)

{

current = current.left;

}

else

{

current = current.right;

}

}

return false;

}

remove(data)

{

const removeNode = function(node, data)

{

if (node == null)

{

return null;

}

if (data == node.data)

{

// node has no children

if (node.left == null && node.right == null)

{

return null;

}

// node has no left child

if (node.left == null)

{

return node.right;

}

// node has no right child

if (node.right == null)

{

return node.left;

}

// node has two children

var tempNode = node.right;

while (tempNode.left !== null)

{

tempNode = tempNode.left;

}

node.data = tempNode.data;

node.right = removeNode(node.right, tempNode.data);

return node;

}

else if (data < node.data)

{

node.left = removeNode(node.left, data);

return node;

}

else

{

node.right = removeNode(node.right, data);

return node;

}

}

this.root = removeNode(this.root, data);

}

isBalanced()

{

return (this.findMinHeight() >= this.findMaxHeight() - 1)

}

findMinHeight(node = this.root)

{

if (node == null) {

return -1;

};

let left = this.findMinHeight(node.left);

let right = this.findMinHeight(node.right);

if (left < right)

{

return left + 1;

}

else

{

return right + 1;

};

}

findMaxHeight(node = this.root)

{

if (node == null)

{

return -1;

};

let left = this.findMaxHeight(node.left);

let right = this.findMaxHeight(node.right);

if (left > right)

{

return left + 1;

}

else

{

return right + 1;

};

}

inOrder()

{

if (this.root == null)

{

return null;

}

else

{

var result = new Array();

function traverseInOrder(node)

{

node.left && traverseInOrder(node.left);

result.push(node.data);

node.right && traverseInOrder(node.right);

}

traverseInOrder(this.root);

return result;

};

}

preOrder()

{

if (this.root == null)

{

return null;

}

Else

{

var result = new Array();

function traversePreOrder(node)

{

result.push(node.data);

node.left && traversePreOrder(node.left);

node.right && traversePreOrder(node.right);

};

traversePreOrder(this.root);

return result;

};

}

postOrder()

{

if (this.root == null)

{

return null;

}

Else

{

var result = new Array();

function traversePostOrder(node)

{

node.left && traversePostOrder(node.left);

node.right && traversePostOrder(node.right);

result.push(node.data);

};

traversePostOrder(this.root);

return result;

}

}

levelOrder()

{

let result = [];

let Q = [];

if (this.root != null)

{

Q.push(this.root);

while(Q.length > 0)

{

let node = Q.shift();

result.push(node.data);

if (node.left != null)

{

Q.push(node.left);

};

if (node.right != null)

{

Q.push(node.right);

};

};

return result;

}

else

{

return null;

};

};

}

const bst = new BST();

bst.add(9);

bst.add(4);

console.log(bst.findMinHeight());

console.log(bst.findMaxHeight());

console.log(bst.isBalanced());

bst.add(10);

console.log(bst.findMinHeight());

console.log(bst.findMaxHeight());

console.log(bst.isBalanced());

console.log('inOrder: ' + bst.inOrder());

console.log('preOrder: ' + bst.preOrder());

console.log('postOrder: ' + bst.postOrder());

console.log('levelOrder: ' + bst.levelOrder());

OutPut:-

