Department of education and science of Ukraine

National technical university of Ukraine

«Kyiv polytechnic institute the name of Igor Sikorsky»

Faculty of informatics and computing engineering

Department of the computing engineering

Laboratory work №6

Discipline: «The algorithms theory»

Topic: «Binary search tree»

EXECUTED:

The first-year student

of FICT group IP-95

*Guskov Danil*

The Student book number- 9505

CHECKED:

Associate Professor

The Computer Engineering Department

PhD, SR

Antoniuk A.I.

Kyiv - 2020

**TASK**

**Goal:**

Convert an input binary tree into a binary search tree. Search for sums of consecutive nodes in a tree.

Binary search trees are binary trees that have the following property: For each node X, the elements in the left subtree X will be less than X, and the elements in the right subtree are greater than X.

**Task variant: 5**

1. Convert an input binary tree into a binary search tree.

A binary tree with a fixed structure (that is, links between nodes, their parent, and descendants) is fed into the input. You must overwrite the values ​​of the tree nodes so that:

a) their new values ​​were taken only from the set present in the input tree;

b) the internal structure of the tree (the links between the parent node and the descendant node) was maintained.

2. Search for sums of consecutive nodes in the tree

After the input tree has been converted to a binary search tree, the next task must be resolved. In addition, a certain number of S. is given. In the obtained binary search tree, it is necessary to find all such monotone paths (which do not necessarily come from the root, but all go from top to bottom) that the sum of the values ​​of the nodes belonging to the found paths is equal to S.

**SOFTWARE CODE**

**File:**

**6-laba.js**

**'use strict';**

**const BinaryTree = require('./BinaryTree');**

**const BinarySearchTree = require('./BinarySearchTree');**

**// Node instance**

**class Node {**

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

**this.value = value;**

**this.left = left;**

**this.right = right;**

**}**

**};**

**// Creating this binary tree:**

**// 0**

**// / \**

**// 1 2**

**// / \ /**

**// 3 4 5**

**// \ \**

**// 6 7**

**// \**

**// 8**

**const n8 = new Node(8);**

**const n7 = new Node(7);**

**const n4 = new Node(4);**

**const n6 = new Node(6, null, n8);**

**const n3 = new Node(3, null, n6);**

**const n1 = new Node(1, n3, n4);**

**const n5 = new Node(5, null, n7);**

**const n2 = new Node(2, n5);**

**const n0 = new Node(0, n1, n2),**

**root = n0;**

**const bt = new BinaryTree(root);**

**// Function that change BT nodes values in in-order**

**// fashion to the values of "inorder" array**

**const changeNodeValuesInorder = (currentNode, inorder, index) => {**

**if(currentNode === null) return null;**

**changeNodeValuesInorder(currentNode.left, inorder, index);**

**currentNode.value = inorder[index[0]];**

**index[0] += 1;**

**changeNodeValuesInorder(currentNode.right, inorder, index);**

**}**

**// Function that convert binary tree to binary search tree**

**const binaryTree2BinarySearchTree = (bt) => {**

**if(!(bt instanceof BinaryTree)){**

**throw 'argument must be an "BinaryTree" instance';**

**}**

**const inorder = [];**

**const index = [ 0 ];**

**// Make in-order traversal of binary tree**

**bt.inOrder(inorder, index[0]);**

**// Sort the in-ordered array**

**const sorted = inorder.sort();**

**index[0] = 0;**

**// Copy inputed binary tree...**

**const btCopy = new Object(bt);**

**// Modifies tree by changing node values while traversing**

**// tree inorder fashion...');**

**changeNodeValuesInorder(btCopy.root, sorted, index);**

**return btCopy;**

**}**

**const sumsOfConsecutiveNodes = (bst, s) => {**

**if(!(bst instanceof BinarySearchTree)){**

**throw 'argument must be an "BinarySearchTree" instance';**

**}**

**const findSums = (node, result, s, currSeq = []) => {**

**if(!node) return;**

**currSeq.push(node.data);**

**const currSum = currSeq.reduce( (a, b) => a + b );**

**if(currSum > s){**

**return;**

**}**

**if(currSum === s){**

**result.push(currSeq);**

**return;**

**}**

**findSums(node.left, result, s);**

**findSums(node.right, result, s);**

**findSums(node.left, result, s, Array.from(currSeq));**

**findSums(node.right, result, s, Array.from(currSeq));**

**}**

**const result = [];**

**const root = bst.getRootNode();**

**findSums(root, result, s);**

**return result;**

**}**

**const s = 13;**

**// USE //**

**console.log('\nInput parameter "S": ' + s + '\n');**

**console.log('\nInput binary tree bfs: ');**

**console.log(bt.bfs());**

**console.log('\nInput binary tree inorder: ');**

**const inorder1 = [];**

**const i = [ 0 ];**

**bt.inOrder(inorder1, i);**

**console.log(inorder1);**

**console.log('\nVisualise input binary tree')**

**console.log(`**

**0**

**/ \\**

**1 2**

**/ \\ /**

**3 4 5**

**\\ \\**

**6 7**

**\\**

**8`);**

**console.log('\nConvert an input binary tree into a binary search tree.');**

**console.log('Binary search tree creating...');**

**var bst = new BinarySearchTree();**

**console.log('Binary search tree has been created!');**

**console.log('Converting data from BT to BST...');**

**const btEx = binaryTree2BinarySearchTree(bt);**

**btEx.bfs().forEach( el => bst.insert(el));**

**console.log('\nBinary tree has been converted to Binary Search tree!\n');**

**console.log('Binary search tree bfs: ');**

**console.log(bst.bfs());**

**console.log('\nBinary search tree inorder: ');**

**const inorder2 = [];**

**bst.inorder(inorder2);**

**console.log(inorder2);**

**console.log('\nVisualise outpute binary search tree');**

**console.log(`**

**5**

**/ \\**

**3 8**

**/ \\ /**

**0 4 6**

**\\ \\**

**1 7**

**\\**

**2`);**

**console.log('\nSearch for sums of consecutive nodes in the tree');**

**console.log('Sum ("s" parameter) = ' + s);**

**console.log('Sequences: ');**

**console.dir(sumsOfConsecutiveNodes(bst, s));**

**console.log('\n');**

**File:**

**BinaryTree.js**

**'use strict';**

**// base node object**

**class Node {**

**constructor(key, value){**

**this.key = key;**

**this.value = value;**

**this.left = null;**

**this.right = null;**

**this.count = 0;**

**}**

**}**

**class BinaryTree {**

**constructor(bt){**

**// reference to root node**

**this.root = bt;**

**}**

**// in order traversing**

**inOrder(result, index) {**

**inOrder(this.root, result, index);**

**}**

**// post order traversing**

**bfs() {**

**return bfs(this.root);**

**}**

**addNode(value){**

**this.root = addNode(this.root, value);**

**}**

**}**

**function addNode(node, value){**

**if (!node) return new Node(value, value);**

**if (node.value > value ){**

**node.left = addNode( node.left, value);**

**} else if (node.value < value) {**

**node.right = addNode(node.right, value);**

**} else if (node.value == value){**

**node.value = value;**

**}**

**node.count = 1 + getSize(node.left) + getSize(node.right);**

**return node;**

**}**

**function inOrder(node, result, index){**

**if (node === null) return null;**

**inOrder(node.left, result, index);**

**result.push(node.value);**

**index[0]++;**

**inOrder(node.right, result, index);**

**}**

**function bfs(node){**

**const queue = [];**

**const values = [];**

**queue.push(node);**

**while(queue.length > 0){**

**const tempNode = queue.shift();**

**values.push(tempNode.value);**

**if (tempNode.left){**

**queue.push(tempNode.left);**

**}**

**if (tempNode.right){**

**queue.push(tempNode.right);**

**}**

**}**

**return values;**

**}**

**module.exports = BinaryTree;**

**File:**

**BinarySearchTree.js**

**'use strict';**

**// Node class**

**class Node {**

**constructor(data) {**

**this.data = data;**

**this.left = null;**

**this.right = null;**

**}**

**}**

**// Binary Search tree class**

**class BinarySearchTree {**

**constructor(root = null) {**

**// root of a binary seach tree**

**this.root = root;**

**}**

**// helper method which creates a new node to**

**// be inserted and calls insertNode**

**insert(data) {**

**// Creating a node and initailising**

**// with data**

**var newNode = new Node(data);**

**// root is null then node will**

**// be added to the tree and made root.**

**if(this.root === null) {**

**this.root = newNode;**

**} else {**

**// find the correct position in the**

**// tree and add the node**

**this.insertNode(this.root, newNode);**

**}**

**}**

**// Method to insert a node in a tree**

**// it moves over the tree to find the location**

**// to insert a node with a given data**

**insertNode(node, newNode) {**

**// if the data is less than the node**

**// data move left of the tree**

**if(newNode.data < node.data) {**

**// if left is null insert node here**

**if(node.left === null) {**

**node.left = newNode;**

**} else {**

**// if left is not null recurr until**

**// null is found**

**this.insertNode(node.left, newNode);**

**}**

**} else {**

**// if the data is more than the node**

**// data move right of the tree**

**// if right is null insert node here**

**if(node.right === null) {**

**node.right = newNode;**

**} else {**

**// if right is not null recurr until**

**// null is found**

**this.insertNode(node.right,newNode);**

**}**

**}**

**}**

**// returns root of the tree**

**getRootNode() {**

**return this.root;**

**}**

**// Performs inorder traversal of a tree**

**inorder(result) {**

**return inOrder(this.root, result);**

**}**

**bfs() {**

**return bfs(this.root);**

**}**

**}**

**function inOrder(node, result) {**

**if(node !== null) {**

**inOrder(node.left, result);**

**result.push(node.data);**

**inOrder(node.right, result);**

**}**

**return result;**

**}**

**function bfs(node){**

**var queue = [];**

**var values = [];**

**queue.push(node);**

**while(queue.length > 0){**

**var tempNode = queue.shift();**

**values.push(tempNode.data);**

**if (tempNode.left){**

**queue.push(tempNode.left);**

**}**

**if (tempNode.right){**

**queue.push(tempNode.right);**

**}**

**}**

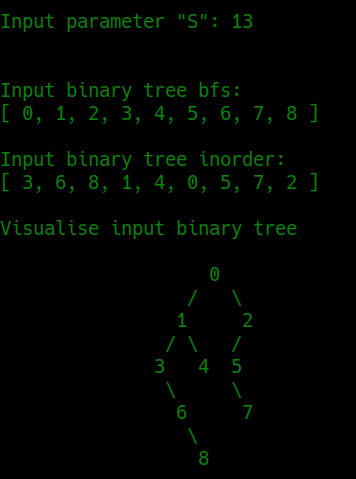
**return values;**

**}**

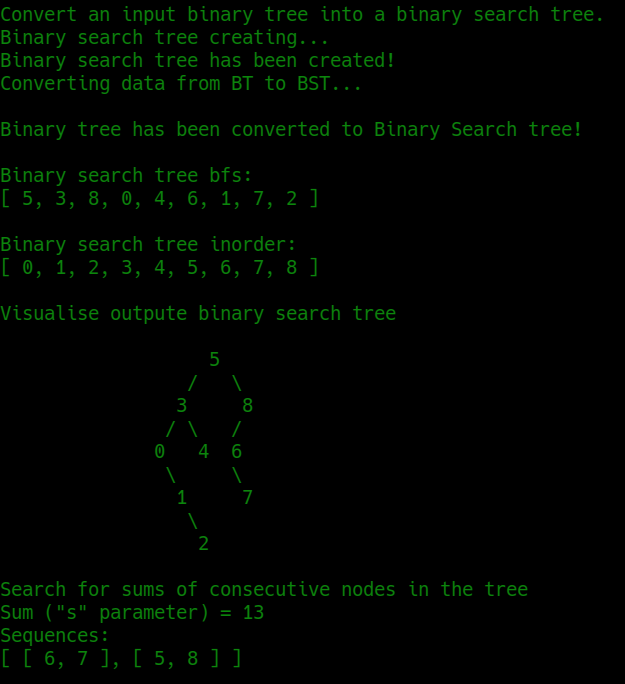
**module.exports = BinarySearchTree;**

**RESULTS OF THE PROGRAM WORK**

The input:



Output:



**CONCLUSIONS**

Familiarized with the topic of laboratory work.

Have acquired relevant work skills.

An appropriate test program has been developed.

The results of the successful work of the test program above confirm the correctness of the chosen decisions, the ultimate goal of the work has been achieved.