**BST**

**BINARY SEARCH TREE**

'use strict'

var abs = Math.abs;

var max = Math.max;

var min = Math.min;

function Node(data){

    this.data = data;

    this.left = null;

    this.right = null;

}

Node.prototype.search = function(s){

    if(this.data === s){

        return this;

    }

    if(s <= this.data && this.left !== null){

        return this.left.search(s);

    }else if(s > this.data && this.right !== null){

        return this.right.search(s)

    }else{

        return undefined;

    }

}

Node.prototype.isBalanced = function(){

    if(!this.right && !this.left) return true

    if(abs(this.balanceFactor())>1){

        return false

    }

    if(!this.right){

        return this.left.isBalanced();

    }else if(!this.left){

        return this.right.isBalanced();

    }

    else{

        return this.right.isBalanced() && this.left.isBalanced();

    }

}

Node.prototype.height = function(){

    if(!this.left && !this.right){

        return 0;

    }else{

        if(this.left && !this.right){

            return this.left.height() + 1;

        }else if(!this.left && this.right){

            return this.right.height() + 1;

        }else{

            return max(this.left.height() + 1, this.right.height() + 1 );

        }

    }

}

Node.prototype.balanceFactor = function(){

    if(!this.left && !this.right){

        return 0;

    }else{

        if(this.left && !this.right){

            return  - this.left.height() - 1 ;

        }else if(!this.left && this.right){

            return this.right.height() + 1;

        }else{

            return this.right.height() - this.left.height() ;

        }

    }

}

Node.prototype.rotate = function(){

    var aux;

    if(this.balanceFactor() <= -2){

        if(this.left.left && this.left.balanceFactor() <= 0){

            //Left Left

            // console.log('ll','pivot:', this.data);

            aux = this.left;

            this.left = aux.right;

            aux.right = this;

        }else if(this.left.right){

            //Left Right

            // console.log('lr','pivot:', this.data);

            aux = this.left.right;

            this.left.right = aux.left;

            aux.left = this.left;

            this.left = aux.right;

            aux.right = this;

            }

    }else if (this.balanceFactor() >=2 ){

        if(this.right.right && this.right.balanceFactor() >= 0){

            //Right Right

            // console.log('rr','pivot:', this.data);

            aux = this.right;

            this.right = aux.left;

            aux.left = this;

        }else if(this.right.left){

            //Right Left

            // console.log('rl','pivot:', this.data);

            aux = this.right.left;

            this.right.left = aux.right;

            aux.right = this.right;

            this.right = aux.left;

            aux.left = this;

        }

    }

    return aux;

}

Node.prototype.print = function(s){

    if(!s){

        s=0;

        console.log('===================');

    }

    console.log('--'.repeat(s)+this.data);

    ++s;

    if(this.left) this.left.print(s);

    if(this.right) this.right.print(s);

}

Node.prototype.add = function(v,r){

    if(!r) var r = this;

    if( v < this.data){

        if(this.left === null){

            this.left = new Node(v);

        }else{

            this.left.add(v,r);

        }

    }

    if( v >= this.data){

        if(this.right === null){

            this.right = new Node(v);

        }else{

            this.right.add(v,r);

        }

    }

    return this;

}

Node.prototype.insert = function(v){

    var tree = this;

    this.add(v);

    while(!this.isBalanced()){

        tree = tree.balance();

    }

    return tree; // always return the root

}

Node.prototype.maxBalanceFactor = function(m){

    if(m === undefined) var m = 0;

    else{

        if (abs(this.balanceFactor()) > m) m = abs(this.balanceFactor());

    }

    if(!this.left && this.right) return max(this.right.maxBalanceFactor(m), m);

    if(!this.right && this.left) return max(this.left.maxBalanceFactor(m), m);

    if(this.right && this.left)  return max(this.right.maxBalanceFactor(m), this.left.maxBalanceFactor(m))

    if(!this.right && !this.left) return 1;

}

Node.prototype.balance = function(anterior, arbol, from){

    if(!arbol) arbol = this;

    if(!anterior) anterior = this;

    let thisbf  =   this.balanceFactor();

    let leftbf  = Infinity;

    let rightbf = Infinity;

    if(this.isBalanced()) return arbol;

    if(this.left)   leftbf  =   this.left.balanceFactor();

    if(this.right)  rightbf =   this.right.balanceFactor();

    // console.log( 'node:',this.data,'l:', leftbf, 'this:',  thisbf  , 'r:', rightbf);

    if((thisbf < 0 && this.maxBalanceFactor() >= abs(thisbf) )){

        //el desbalance está a la izquierda

        if(this.left){

            return this.left.balance(this,arbol, 'left');

        }

    }else if(( thisbf > 0 && this.maxBalanceFactor() >= thisbf)){

        //desbalance a la derecha

        if(this.right){

            return this.right.balance(this, arbol, 'right');

        }

    }else{

        if(this.right) if( !this.right.isBalanced()) return this.right.balance(this, arbol, 'right');

        if(this.left) if( !this.left.isBalanced()) return this.left.balance(this, arbol, 'right');

        //este arbol esta desbalanceado

        if(this == arbol) { return  this.rotate() }

        else{anterior[from] = this.rotate(); }

    }

    return arbol;

}

Node.prototype.findMin = function(){

    if(this.left && this.right){

        var l = this.left.findMin();

        var r = this.right.findMin();

        if(l.data < r.data) return l;

        else return r

    }else{

        if(this.left){

            minL = this.left.findMin();

            return minL.data  <= this.data ?  minL: this;

        }

        if(this.right){

            minR = this.right.findMin();

            return minR.data  <= this.data ?  minR: this;

        }

    }

    if(!this.left && !this.right) return this;

}

Node.prototype.findMax = function(){

    if(this.left && this.right){

        var l = this.left.findMax();

        var r = this.right.findMax();

        if(l.data > r.data) return l;

        else return r

    }else{

        if(this.left){

            let maxL = this.left.findMax();

            return maxL.data  > this.data ?  maxL: this;

        }

        if(this.right){

            let maxR = this.right.findMax()

            return maxR.data > this.data ? maxR: this

        }

    }

    if(!this.left && !this.right) return this;

}

Node.prototype.delete = function(v){

    var tree = this;

    var node = this.search(v);

    if(!this.left && !this.right && this === node) return null;

    if(!node) return this;

    if(node.left){

        var target = node.left.findMax();

        if(node.left == target) node.left = target.left;

        node.destroy(target);

        node.data = target.data;

    }else if(node.right){

        var target = node.right.findMin()

        if(node.left === target) node.left = target.left;

        node.destroy(target);

        node.data = target.data;

        node.right = target.right;

    }else{

        this.destroy(node);

    }

    while(!this.isBalanced()){

        tree = tree.balance();

    }

    return tree;

}

Node.prototype.destroy = function(n, anterior, next){

    if(!anterior) var anterior = this;

    if(this == n ){

        if( anterior == this){ //Is the root element

            return null;

        }else{

            return anterior[next] = null;

        }

    }

    if(n.data < this.data){

        if(this.left){

            this.left.destroy(n, this, 'left' )

        }

    }else{

        if(this.right){

            this.right.destroy(n, this, 'right' )

        }

    }

    return this;

}

// var tree =  new Node(22).insert(20).insert(62).insert(36).insert(60).insert(78).insert(68)

                // .insert(5).insert(37).insert(49)

 // var tree= new Node(60).add(68).add(62).add(78).add(22).add(20).add(5).add(36).add(37).add(49).balance2()

// tree = new Node(100).insert(62).insert(34).insert(15).insert(21).insert(86).insert(12)

                // .insert(8).insert(8).insert(92);

 // var tree = new Node(87).insert(93).insert(61).insert(80).insert(8).insert(89).insert(22)

                // .insert(91).insert(60).insert(33);

let tree = new Node(87).insert(93).insert(61).insert(80).insert(8).insert(91).insert(60).insert(33).insert(34).insert(35).insert(36);

                //.delete(8).delete(7).delete(6).delete(5).delete(4).delete(3).delete(2).delete(1);

// console.log('arbol:\n',tree);

console.log('arbol:\n',tree);

module.exports = Node;

/\*

 Implementar la clase BinarySearchTree, definiendo los siguientes métodos recursivos:

  - size: retorna la cantidad total de nodos del árbol

  - insert: agrega un nodo en el lugar correspondiente

  - contains: retorna true o false luego de evaluar si cierto valor existe dentro del árbol

  - depthFirstForEach: recorre el árbol siguiendo el orden depth first (DFS) en cualquiera de sus variantes, según se indique por parámetro ("post-order", "pre-order", o "in-order"). Nota: si no se provee ningún parámetro, hará el recorrido "in-order" por defecto.

  - breadthFirstForEach: recorre el árbol siguiendo el orden breadth first (BFS)

  El ábrol utilizado para hacer los tests se encuentra representado en la imagen bst.png dentro del directorio homework.

\*/

function BinarySearchTree(value) {

  this.value = value;

  this.left = null;

  this.right = null;

}

BinarySearchTree.prototype.insert = function(val){

  val < this.value ?

    (!this.left ? this.left = new BinarySearchTree(val) : this.left.insert(val)) :

    (!this.right ? this.right = new BinarySearchTree(val) : this.right.insert(val));

}

BinarySearchTree.prototype.size = function () {

  let cuenta=1;

  if(this.left) cuenta += this.left.size();

  if(this.right) cuenta += this.left.size();

  return cuenta

}

BinarySearchTree.prototype.contains = function (val) {

if (this.value === val) return true;

  if (this.left?.contains(val)) return true;

  if (this.right?.contains(val)) return true;

return false;

}

BinarySearchTree.prototype.depthFirstForEach = function (cb, order = "in-order") {

  switch (order) {

    case "in-order": {

      this.left?.depthFirstForEach(cb, order);

      cb(this.value);

      this.right?.depthFirstForEach(cb, order);

      break;

    }

    case "pre-order": {

      cb(this.value);

      this.left?.depthFirstForEach(cb, order);

      this.right?.depthFirstForEach(cb, order);

      break;

    }

    case "post-order": {

      this.left?.depthFirstForEach(cb, order);

      this.right?.depthFirstForEach(cb, order);

      cb(this.value);

      break;

    }

  }

}

//defino el [] como parametro para que se mantenga en todos los niveles

BinarySearchTree.prototype.breadthFirstForEach = function (cb, queue = []) {

  cb(this.value);//mando el valor del nodo actual

  if (this.left) queue.push(this.left);//meto en la cola el izq

  if (this.right) queue.push(this.right);//meto en la cola el der

  if (queue.length) {//si la cola tiene algo

    queue.shift().breadthFirstForEach(cb, queue);//saco el nodo y ejecuto el nodo

  }

};