# DSA Assignment 3

Fa21-bcs-073

Muhammad Aqib Mehmood

## AVLTree Class

package LabAssign3;

public class AVLTree {

    private int key, height;

    private AVLTree left, right;

    public static final AVLTree NIL = new AVLTree();

    public AVLTree(int key) {

        this.key = key;

        left = right = NIL;

    }

    public boolean add(int key) {

        int oldSize = size();

        grow(key);

        return size() > oldSize;

    }

    public AVLTree grow(int key) {

        if (this == NIL)

            return new AVLTree(key);

        if (key == this.key)

            return this; // prevent key duplication

        if (key < this.key)

            left = left.grow(key);

        else

            right = right.grow(key);

        rebalance();

        height = 1 + Math.max(left.height, right.height);

        return this;

    }

    public int size() {

        if (this == NIL)

            return 0;

        return 1 + left.size() + right.size();

    }

    public String toString() {

        if (this == NIL)

            return "";

        return left + " " + key + " " + right;

    }

    private AVLTree() { // constructs the empty tree

        left = right = this;

        height = -1;

    }

    private AVLTree(int key, AVLTree left, AVLTree right) {

        this.key = key;

        this.left = left;

        this.right = right;

        height = 1 + Math.max(left.height, right.height);

    }

    private void rebalance() {

        if (right.height > left.height + 1) {

            if (right.left.height > right.right.height)

                right.rotateRight();

            rotateleft();

        }

        else if (left.height > right.height + 1) {

            if (left.right.height > left.left.height)

                left.rotateleft();

            rotateRight();

        }

    }

    private void rotateleft() {

        left = new AVLTree(key, left, right.left);

        key = right.key;

        right = right.right;

    }

    private void rotateRight() {

        right = new AVLTree(key, left.right, right);

        key = left.key;

        left = left.left;

    }

    public void printTree() {

        printTreeHelper(this);

    }

    private void printTreeHelper(AVLTree node) {

        if (node != NIL) {

            printTreeHelper(node.left);

            System.out.print(node.key + " ");

            printTreeHelper(node.right);

        }

    }

    public boolean isAVLTreeHelper(AVLTree node){

        if(node == NIL)

            return true;

        if(node.left != NIL && node.left.key > node.key)

            return false;

        if(node.right != NIL && node.right.key < node.key)

            return false;

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

    }

    public boolean isAVLTree(){

        return isAVLTreeHelper(this);

    }

    public boolean contains(int x){

        if(this == NIL)

            return false;

        if(this.key == x)

            return true;

        if(x < this.key)

            return left.contains(x);

        else

            return right.contains(x);

    }

    public AVLTree(int [] a){

        this.key = a[0];

        left = right = NIL;

        for(int i = 1; i < a.length; i++){

            this.add(a[i]);

        }

    }

}

## Heaptree Class

package LabAssign3;

import java.util.Stack;

class Heaptree<T extends Comparable<T>> {

    private Node<T> root;

    private Node<T> last;

    private Node<T> parent(Node<T> node) {

        if (node == root)

            return null;

        Node<T> parent = root;

        while (parent.left != node && parent.right != node) {

            parent = (parent.left != null) ? parent.left : parent.right;

        }

        return parent;

    }

    private Node<T> sibling(Node<T> node) {

        Node<T> parent = parent(node);

        if (parent == null)

            return null;

        return (parent.left == node) ? parent.right : parent.left;

    }

    private void heapifyUp(Node<T> node) {

        if (node == root)

            return;

        Node<T> parent = parent(node);

        if (parent.data.compareTo(node.data) < 0) {

            T temp = parent.data;

            parent.data = node.data;

            node.data = temp;

            heapifyUp(parent);

        }

    }

    private void heapifyDown(Node<T> node) {

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

            return;

        Node<T> largest = node;

        if (node.left != null && node.left.data.compareTo(largest.data) > 0)

            largest = node.left;

        if (node.right != null && node.right.data.compareTo(largest.data) > 0)

            largest = node.right;

        if (largest != node) {

            T temp = node.data;

            node.data = largest.data;

            largest.data = temp;

            heapifyDown(largest);

        }

    }

    public Heaptree() {

        this.root = null;

        this.last = null;

    }

    public void insert(T data) {

        Node<T> newNode = new Node<>(data);

        Node<T> parent = parent(last);

        if (isEmpty())

            root = last = newNode;

        else if (root == last)

            root.left = newNode;

        else if (parent.left == last)

            parent.right = newNode;

        else {

            Node<T> curr = last;

            while (parent != root && parent.left != curr) {

                curr = parent;

                parent = parent(parent);

            }

            if (parent == root) {

                while (parent.left != null)

                    parent = parent.left;

                parent.left = newNode;

            } else if (parent.left == curr) {

                parent.right = newNode;

            }

        }

        heapifyUp(newNode);

        last = newNode;

    }

    public T remove() {

        if (isEmpty())

            throw new RuntimeException("Heap is empty!");

        T data = last.data;

        if (root == last) {

            root.destroy();

            root = last = null;

            return data;

        }

        T temp = root.data;

        root.data = last.data;

        last.data = temp;

        Node<T> parent = parent(last);

        Node<T> curr = last;

        if (parent.right == last) {

            parent.right.destroy();

            parent.right = null;

            last = parent.left;

        } else if (parent.left == last) {

            parent.left.destroy();

            parent.left = null;

            while (parent != root && parent.right != curr) {

                curr = parent;

                parent = parent(parent);

            }

            if (parent == root) {

                while (parent.right != null)

                    parent = parent.right;

                last = parent.right;

            } else if (parent.right == curr) {

                Node<T> node = sibling(curr);

                while (node.right != null)

                    parent = node.right;

                last = node.right;

            }

        }

        heapifyDown(root);

        return data;

    }

    public boolean isEmpty() {

        return root == null;

    }

    public void print() {

        Stack<Node<T>> nodeStack = new Stack<>();

        Stack<Integer> marginStack = new Stack<>();

        nodeStack.push(root);

        marginStack.push(0);

        while (!nodeStack.isEmpty()) {

            Node<T> node = nodeStack.pop();

            int marginLevel = marginStack.pop();

            for (int i = 0; i < marginLevel; i++)

                System.out.print("   ");

            System.out.print("|- ");

            if (node != null) {

                if (node.left != null || node.right != null) {

                    nodeStack.push(node.right);

                    marginStack.push(marginLevel + 1);

                    nodeStack.push(node.left);

                    marginStack.push(marginLevel + 1);

                }

                System.out.println(node.data);

            } else {

                System.out.println("NULL");

            }

        }

    }

}

## Node Class

package LabAssign3;

class Node<T> {

    public T data;

    public Node<T> left;

    public Node<T> right;

    public Node(T data) {

        this.data = data;

        this.left = null;

        this.right = null;

    }

    public void destroy() {

        if (left != null)

            left.destroy();

        if (right != null)

            right.destroy();

    }

}

## Driver Class

package LabAssign3;

public class Driver {

    public static void main(String[] args) {

        AVLTree tree = new AVLTree(20);

        tree.add(30);

        tree.add(40);

        tree.add(50);

        tree.add(60);

        tree.add(70);

        tree.add(80);

        System.out.println(tree);

        tree.printTree();

        System.out.println();

        System.out.println("Is that an AVL Tree: " +tree.isAVLTree());

        System.out.println("Is that tree contains 40: " +tree.contains(40));

        int[] arr = {100,200,300,400,500,600,700,800};

        AVLTree T2 = new AVLTree(arr);

        T2.printTree();

        System.out.println();

        System.out.println("-----------------Q2----------------");

        Heaptree<Integer> heap = new Heaptree<>();

        heap.insert(8);

        heap.insert(4);

        heap.insert(10);

        heap.insert(9);

        heap.insert(3);

        heap.insert(5);

        heap.print();

        System.out.println("Removing elements from the heap:");

        System.out.println(heap.remove());

        heap.print();

    }

}

## Output:

