Санкт-Петербургский Национальный Исследовательский Университет
Информационных Технологий, Механики и Оптики

Лабораторная работа №7

по дисциплине

«Алгоритмы и структуры данных»

Выполнил: Студент группы P3217

Сергачев Данила Дмитриевич

Преподаватели:

Романов Алексей Андреевич и

Волчек Дмитрий Геннадьевич

Санкт-Петербург

2018 г

Задание №1

Проверка сбалансированности

1.0 из 1.0 балла (оценивается)

Имя входного файла:	input.txt
Имя выходного файла:	output.txt
Ограничение по времени:	2 секунды
Ограничение по памяти:	256 мегабайт

АВЛ-дерево является сбалансированным в следующем смысле: для любой вершины высота ее левого поддерева отличается от высоты ее правого поддерева не больше, чем на единицу.

Введем понятие *баланса вершины*: для вершины дерева V ее баланс $B(V)$ равен разности высоты правого поддерева и высоты левого поддерева. Таким образом, свойство АВЛ-дерева, приведенное выше, можно сформулировать следующим образом: для любой ее вершины V выполняется следующее неравенство:

Решение

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace week7
{
   class Task1
   {
      public static void Main(string[] args)
      {
         var app = new Task1();
         app.DoWork(args);
      }

      private void DoWork(string[] args)
      {
         using (var sw = new StreamWriter("output.txt"))
         {
            var stdin = File.ReadAllLines("input.txt");
            var n = long.Parse(stdin[0]);
            var tree = new Tree(n);
            tree.Parse(stdin);

            for (int i = 0; i < n; i++)
               sw.WriteLine(TreeNode<long>.GetBalance(tree.Nodes[i]));

         }
      }
   }

   public class Tree
   {
```

```
    public TreeNode<long>[] Nodes { get; private set; }
    private long _nodesCount;

    public Tree(long n)
    {
        _nodesCount = n;
        this.Nodes = new TreeNode<long>[n];
    }
    //Parsing
    public void Parse(string[] stdin)
    {
        for (int i = 1; i <= _nodesCount; i++)
        {
            var temp = stdin[i].Split(' ').Select(x => long.Parse(x)).ToArray();

            if (this.Nodes[i - 1] == null)
                this.Nodes[i - 1] = new TreeNode<long>();
            this.Nodes[i - 1].Key = temp[0];
            //Left child
            if (temp[1] != 0)
            {
                if (this.Nodes[temp[1] - 1] == null)
                    this.Nodes[temp[1] - 1] = new TreeNode<long>()
                    {
                        Parent = this.Nodes[i - 1]
                    };
                this.Nodes[i - 1].Left = this.Nodes[temp[1] - 1];
            }
            //Right child
            if (temp[2] != 0)
            {
                if (temp[2] != 0 && this.Nodes[temp[2] - 1] == null)
                    this.Nodes[temp[2] - 1] = new TreeNode<long>() { Parent = this.Nodes[i - 1] };
                this.Nodes[i - 1].Right = this.Nodes[temp[2] - 1];
            }

            //Calc Height
            if (temp[1] == 0 & temp[2] == 0)
            {
                TreeNode<long> leaf = this.Nodes[i - 1];
                Stack<TreeNode<long>> curr = new Stack<TreeNode<long>>();
                while (leaf != null)
                {
                    curr.Push(leaf);
                    leaf = leaf.Parent;
                }
                while (curr.Count != 0)
                {
                    leaf = curr.Pop();
                    if (leaf.Height < curr.Count)
                        leaf.Height = curr.Count;
                }
            }
        }
    }
}

public class TreeNode<T> where T : IComparable<T>
{
    public T Key { get; set; }
    public TreeNode<T> Parent { get; set; }
    public TreeNode<T> Left { get; set; }
    public TreeNode<T> Right { get; set; }

    public long Height { get; set; }
```

```
        public static long GetBalance(TreeNode<T> tree)
        {
            if (tree == null)
                throw new ArgumentNullException("Tree does not exist!");

            if (tree.Left != null && tree.Right != null)
                return tree.Right.Height - tree.Left.Height;
            if (tree.Left == null && tree.Right != null)
                return tree.Right.Height + 1;
            if (tree.Left != null && tree.Right == null)
                return -1 - tree.Left.Height;
            else
                return 0;
        }
    }
}
```

Результат работы

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.515	51191808	3986010	1688889
1	OK	0.031	11448320	46	19
2	OK	0.031	11436032	10	3
3	OK	0.031	11468800	17	6
4	OK	0.031	11493376	17	7
5	OK	0.031	11464704	24	9
6	OK	0.031	11431936	24	10
7	OK	0.031	11681792	24	9
8	OK	0.046	11452416	24	10

Задание №2

Делаю я левый поворот...

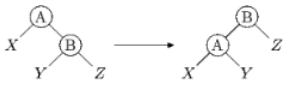
1.0 из 1.0 балла (оценивается)

Имя входного файла:	input.txt
Имя выходного файла:	output.txt
Ограничение по времени:	2 секунды
Ограничение по памяти:	256 мегабайт

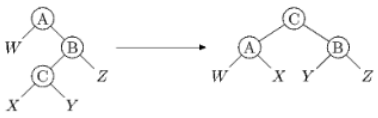
Для балансировки АВЛ-дерева при операциях вставки и удаления производятся *левые* и *правые* повороты. Левый поворот в вершине производится, когда баланс этой вершины больше 1, аналогично, правый поворот производится при балансе, меньшем −1.

Существует два разных левых (как, разумеется, и правых) поворота: *большой* и *малый* левый поворот.

Малый левый поворот осуществляется следующим образом:



Заметим, что если до выполнения малого левого поворота был нарушен баланс только корня дерева, то после его выполнения все вершины становятся сбалансированными, за исключением случая, когда у правого ребенка корня баланс до поворота равен −1. В этом случае вместо малого левого поворота выполняется большой левый поворот, который осуществляется так:



Решение

```
#include <iostream>
#include <string>
#include <queue>
#include <deque>

using namespace std;

#ifdef LOCAL

#define cin std::cin
#define cout std::cout

#else

#include "edx-io.hpp"
#define cin io
#define cout io

#endif

#define nl "\n"

struct t_node {
	int left, right, key;
} *tree;

int *keys, *h, *b;
int sz;

int cnt(int i) {
	int d = b[i] = 0;

	if (tree[i].left) {
		d = max(cnt(tree[i].left - 1), d);
		b[i] -= h[tree[i].left - 1];
	}

	if (tree[i].right) {
		d = max(cnt(tree[i].right - 1), d);
		b[i] += h[tree[i].right - 1];
	}


	return h[i] = (d + 1);
}

#include <iostream>
```

```cpp
#include <string>
#include <queue>
#include <deque>

using namespace std;

#ifdef LOCAL

#define cin std::cin
#define cout std::cout

#else

#include "edx-io.hpp"
#define cin io
#define cout io

#endif

#define nl "\n"

struct t_node {
	int left, right, key;
} *tree;

int *keys, *h, *b;
int sz;

int cnt(int i) {
	int d = b[i] = 0;

	if (tree[i].left) {
		d = max(cnt(tree[i].left - 1), d);
		b[i] -= h[tree[i].left - 1];
	}

	if (tree[i].right) {
		d = max(cnt(tree[i].right - 1), d);
		b[i] += h[tree[i].right - 1];
	}


	return h[i] = (d + 1);
}

void big_left_rotation(int root_index) {
	t_node
		a = tree[root_index],
		b = tree[a.right - 1],
		c = tree[b.left - 1];

	int x_ind = c.left - 1,
		y_ind = c.right - 1,
		old_c_ind = b.left - 1,
		b_ind = a.right - 1;

	c.left = old_c_ind + 1;
	c.right = b_ind + 1;

	b.left = y_ind + 1;
	a.right = x_ind + 1;

	tree[root_index] = c;
	tree[old_c_ind] = a;
	tree[b_ind] = b;
```

```
}

void left_rotation(int root_index) {
	if (b[tree[root_index].right - 1] < 0) {
		big_left_rotation(root_index);
		return;
	}

	t_node
		a = tree[root_index],
		b = tree[a.right - 1];

	int y_ind = b.left - 1,
		old_b_index = a.right - 1;

	b.left = old_b_index + 1;
	a.right = y_ind + 1;

	tree[root_index] = b;
	tree[old_b_index] = a;
}

int *indexes;
int curr_index = 1;

void calc_index(int i) {
	if (!i) { return; }

	t_node n = tree[i - 1];
	indexes[i] = curr_index++;

	calc_index(n.left);
	calc_index(n.right);
}

void print_node(int i) {
	if (!i) { return; }

	t_node n = tree[i - 1];
	cout << n.key << " " << indexes[n.left] << " " << indexes[n.right] << nl;

	print_node(n.left);
	print_node(n.right);
}


int main() {
	int n, k, m, x;
	cin >> n;

	tree = new t_node[sz = n];
	h = new int[n];
	b = new int[n];
	indexes = new int[n + 1];
	indexes[0] = 0;

	for (int i = 0; i < n; ++i) {
		cin >> tree[i].key >> tree[i].left >> tree[i].right;

		h[i] = 0;
	}

	cnt(0);
	left_rotation(0);
```

```
	calc_index(1);
	cout << n << nl;
	print_node(1);

	return 0;
}
void big_left_rotation(int root_index) {
	t_node
		a = tree[root_index],
		b = tree[a.right - 1],
		c = tree[b.left - 1];

	int x_ind = c.left - 1,
		y_ind = c.right - 1,
		old_c_ind = b.left - 1,
		b_ind = a.right - 1;

	c.left = old_c_ind + 1;
	c.right = b_ind + 1;

	b.left = y_ind + 1;
	a.right = x_ind + 1;

	tree[root_index] = c;
	tree[old_c_ind] = a;
	tree[b_ind] = b;
}

void left_rotation(int root_index) {
	if (b[tree[root_index].right - 1] < 0) {
		big_left_rotation(root_index);
		return;
	}

	t_node
		a = tree[root_index],
		b = tree[a.right - 1];

	int y_ind = b.left - 1,
		old_b_index = a.right - 1;

	b.left = old_b_index + 1;
	a.right = y_ind + 1;

	tree[root_index] = b;
	tree[old_b_index] = a;
}

int *indexes;
int curr_index = 1;

void calc_index(int i) {
	if (!i) { return; }

	t_node n = tree[i - 1];
	indexes[i] = curr_index++;

	calc_index(n.left);
	calc_index(n.right);
}

void print_node(int i) {
	if (!i) { return; }

	t_node n = tree[i - 1];
	cout << n.key << " " << indexes[n.left] << " " << indexes[n.right] << nl;
```

```
        print_node(n.left);
        print_node(n.right);
}


int main() {
        int n, k, m, x;
        cin >> n;

        tree = new t_node[sz = n];
        h = new int[n];
        b = new int[n];
        indexes = new int[n + 1];
        indexes[0] = 0;

        for (int i = 0; i < n; ++i) {
                cin >> tree[i].key >> tree[i].left >> tree[i].right;

                h[i] = 0;
        }

        cnt(0);
        left_rotation(0);

        calc_index(1);
        cout << n << nl;
        print_node(1);


        return 0;
                                                        }
```

Результат работы

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.140	21630976	25	21
1	OK	0.125	21594112	7	5
2	OK	0.109	21561344	8	5
3	OK	0.109	21561344	5	3
4	OK	0.109	21532672	5	3
5	OK	0.093	21610496	6	3
6	OK	0.125	21614592	6	3
7	OK	0.125	21577728	23	21
8	OK	0.125	21553152	25	20
9	OK	0.109	21557248	24	20
10	OK	0.109	21630976	24	21
11	OK	0.125	21594112	23	20
12	OK	0.140	21532672	23	20
13	OK	0.109	21594112	20	17
14	OK	0.125	21565440	23	20
15	OK	0.125	21553152	20	20
16	OK	0.125	21614592	22	20
17	OK	0.125	21565440	23	20
18	OK	0.125	21573632	22	19
19	OK	0.125	21614592	22	19
20	OK	0.109	21553152	22	20

Задание №3

Вставка в АВЛ-дерево

1.0 из 1.0 балла (оценивается)

Имя входного файла:	input.txt
Имя выходного файла:	output.txt
Ограничение по времени:	2 секунды
Ограничение по памяти:	256 мегабайт

Вставка в АВЛ-дерево вершины V с ключом X при условии, что такой вершины в этом дереве нет, осуществляется следующим образом:

- находится вершина W, ребенком которой должна стать вершина V;
- вершина V делается ребенком вершины W;
- производится подъем от вершины W к корню, при этом, если какая-то из вершин несбалансирована, производится, в зависимости от значения баланса, левый или правый поворот.

Решение

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Week7
{
   class Task3
   {
      public static void Main(string[] args)
      {
         var app = new Task3();
         app.DoWork(args);
      }

      private void DoWork(string[] args)
      {
         using (var sw = new StreamWriter("output.txt"))
```

```
    {
        string[] stdin = File.ReadAllLines("input.txt");

        TreeNode<long> root = null;
        for (int i = 1; i <= long.Parse(stdin[0]); i++)
            root = TreeNode<long>.Insert(root, new TreeNode<long> { Key = long.Parse(stdin[i].Split(' ')[0]) });

        for (int i = int.Parse(stdin[0]) + 1; i < stdin.Length; i++)
        {
            TreeNode<long> node = new TreeNode<long> { Key = long.Parse(stdin[i].Split(' ')[0]) };
            root = TreeNode<long>.Insert(root, node);
            root = TreeNode<long>.Balance(node);
        }

        sw.WriteLine(stdin.Length - 1);
        TreeNode<long>.PrintTree(sw, root);
    }
}

class TreeNode<T> where T : IComparable<T>
{
    public T Key { get; set; }
    public TreeNode<T> Parent { get; set; }
    public TreeNode<T> Left { get; set; }
    public TreeNode<T> Right { get; set; }

    private long Depth { get; set; }
    public long Height { get; private set; }

    public static TreeNode<T> Previous(TreeNode<T> node)
    {
        if (node.Left == null)
            return node;
        return Maximum(node.Left);
    }

    public static TreeNode<T> Maximum(TreeNode<T> node)
```

```
{
    while (node.Right != null)
        node = node.Right;
    return node;
}

/// <returns>Root of tree after remove</returns>
public static TreeNode<T> Remove(TreeNode<T> item)
{
    TreeNode<T> parent = item.Parent;

    //Leaf
    if (item.Left == null && item.Right == null)
    {
        if (parent == null)
            return null;
        if (parent.Left == item)
            parent.Left = null;
        else
            parent.Right = null;

        UpdateHeight(parent);
        return Balance(parent);
    }

    //One child
    if ((item.Left == null) ^ (item.Right == null))
        if (item.Left != null)
        {
            if (parent != null)
            {
                if (parent.Left == item)
                    parent.Left = item.Left;
                else
                    parent.Right = item.Left;

                UpdateHeight(parent);
```

```
            }

            item.Left.Parent = parent;
            return Balance(item.Left);
        }
        else
        {
            if (parent != null)
            {
                if (parent.Left == item)
                    parent.Left = item.Right;
                else
                    parent.Right = item.Right;

                UpdateHeight(parent);
            }

            item.Right.Parent = parent;
            return Balance(item.Right);
        }


    //Two child
    if ((item.Left != null) && (item.Right != null))
    {
        TreeNode<T> prev = Previous(item);
        Remove(prev);
        item.Key = prev.Key;
    }

    return Balance(item);
}

/// <returns>Root of tree after insert</returns>
public static TreeNode<T> Insert(TreeNode<T> root, TreeNode<T> node)
{
    if (root == null)
```

```
            return node;
        TreeNode<T> current = root;
        while (true)
        {
            if (current.Key.CompareTo(node.Key) == 0)
                throw new ArgumentException("Not unique key");
            if (current.Key.CompareTo(node.Key) < 0)
            {
                if (current.Right != null)
                    current = current.Right;
                else
                {
                    current.Right = node;
                    node.Parent = current;
                    UpdateHeight(node);
                    return root;
                    //return Balance(node);
                }
            }
            else
            {
                if (current.Left != null)
                    current = current.Left;
                else
                {
                    current.Left = node;
                    node.Parent = current;
                    UpdateHeight(node);
                    return root;
                    //return Balance(node);
                }
            }
        }
    }

    private static void UpdateHeight(TreeNode<T> node)
    {
```

```
    while (node != null)
    {
        long rH = node.Right != null ? node.Right.Height : -1;
        long lH = node.Left != null ? node.Left.Height : -1;

        long currentH = node.Height;
        if (rH > lH)
            node.Height = rH + 1;
        else
            node.Height = lH + 1;

        node = node.Parent;
    }
}

/// <returns>Root of tree after balance</returns>
public static TreeNode<T> Balance(TreeNode<T> leaf)
{
    TreeNode<T> current = leaf;
    while (current != null)
    {
        long balance = GetBalance(current);
        if (balance > 1)
        {
            if (GetBalance(current.Right) == -1)
                current = BigLeftTurn(current);
            else
                current = SmallLeftTurn(current);
        }
        if (balance < -1)
        {
            if (GetBalance(current.Left) == 1)
                current = BigRightTurn(current);
            else
                current = SmallRightTurn(current);
        }
        if (current.Parent == null)
```

```
            return current;
        else
            current = current.Parent;
    }
    return current;
}

public static void PrintTree(StreamWriter sw, TreeNode<T> root)
{
    if (root == null)
        return;
    Queue<TreeNode<T>> bfsQueue = new Queue<TreeNode<T>>();
    long counter = 1;
    bfsQueue.Enqueue(root);
    while (bfsQueue.Count != 0)
    {
        TreeNode<T> current = bfsQueue.Dequeue();
        sw.Write(current.Key);

        if (current.Left != null)
        {
            bfsQueue.Enqueue(current.Left);
            sw.Write(" " + ++counter);
        }
        else
            sw.Write(" " + 0);

        if (current.Right != null)
        {
            bfsQueue.Enqueue(current.Right);
            sw.WriteLine(" " + ++counter);
        }
        else
            sw.WriteLine(" " + 0);
    }
}
```

```
public static long GetBalance(TreeNode<T> tree)
{
    if (tree == null)
        return 0;

    if (tree.Left != null && tree.Right != null)
        return tree.Right.Height - tree.Left.Height;
    if (tree.Left == null && tree.Right != null)
        return tree.Right.Height + 1;
    if (tree.Left != null && tree.Right == null)
        return -1 - tree.Left.Height;
    else
        return 0;
}

/// <returns>Root of tree after turn</returns>
public static TreeNode<T> SmallLeftTurn(TreeNode<T> root)
{
    TreeNode<T> child = root.Right;
    TreeNode<T> parent = root.Parent;
    TreeNode<T> x = root.Left;
    TreeNode<T> y = root.Right.Left;
    TreeNode<T> z = root.Right.Right;

    //Parents
    child.Parent = parent;
    root.Parent = child;
    if (x != null)
        x.Parent = root;
    if (y != null)
        y.Parent = root;
    if (z != null)
        z.Parent = child;

    //Childs
    root.Left = x;
    root.Right = y;
```

```
    child.Left = root;
    child.Right = z;
    if (parent != null)
        if (parent.Right == root)
            parent.Right = child;
        else
            parent.Left = child;

    //Heights
    long xH = x != null ? x.Height : -1;
    long yH = y != null ? y.Height : -1;
    long zH = z != null ? z.Height : -1;

    if (xH > yH)
        root.Height = xH + 1;
    else
        root.Height = yH + 1;
    if (root.Height > zH)
        child.Height = root.Height + 1;
    else
        child.Height = zH + 1;

    UpdateHeight(child);
    return child;
}

/// <returns>Root of tree after turn</returns>
public static TreeNode<T> SmallRightTurn(TreeNode<T> root)
{
    TreeNode<T> child = root.Left;
    TreeNode<T> parent = root.Parent;
    TreeNode<T> x = root.Right;
    TreeNode<T> y = root.Left.Left;
    TreeNode<T> z = root.Left.Right;

    //Parents
    child.Parent = parent;
```

```
    root.Parent = child;
    if (x != null)
        x.Parent = root;
    if (y != null)
        y.Parent = child;
    if (z != null)
        z.Parent = root;

    //Childs
    root.Left = z;
    root.Right = x;
    child.Left = y;
    child.Right = root;
    if (parent != null)
        if (parent.Right == root)
            parent.Right = child;
        else
            parent.Left = child;

    //Heights
    long xH = x != null ? x.Height : -1;
    long yH = y != null ? y.Height : -1;
    long zH = z != null ? z.Height : -1;

    if (zH > xH)
        root.Height = zH + 1;
    else
        root.Height = xH + 1;

    if (y.Height > root.Height)
        child.Height = yH + 1;
    else
        child.Height = root.Height + 1;

    UpdateHeight(child);
    return child;
}
```

```
/// <returns>Root of tree after turn</returns>
public static TreeNode<T> BigRightTurn(TreeNode<T> root)
{
    TreeNode<T> w = root.Right;
    TreeNode<T> parent = root.Parent;
    TreeNode<T> b = root.Left;
    TreeNode<T> c = root.Left.Right;
    TreeNode<T> z = b.Left;
    TreeNode<T> x = c.Left;
    TreeNode<T> y = c.Right;

    //Parents
    c.Parent = parent;
    b.Parent = c;
    root.Parent = c;
    if (w != null)
        w.Parent = root;
    if (z != null)
        z.Parent = b;
    if (y != null)
        y.Parent = root;
    if (x != null)
        x.Parent = b;

    //Childs
    if (parent != null)
        if (parent.Right == root)
            parent.Right = c;
        else
            parent.Left = c;
    c.Left = b;
    c.Right = root;
    b.Left = z;
    b.Right = x;
    root.Left = y;
    root.Right = w;
```

```
    //Heights
    long xH = x != null ? x.Height : -1;
    long yH = y != null ? y.Height : -1;
    long zH = z != null ? z.Height : -1;
    long wH = w != null ? w.Height : -1;

    if (zH > xH)
        b.Height = zH + 1;
    else
        b.Height = xH + 1;

    if (yH > wH)
        root.Height = yH + 1;
    else
        root.Height = wH + 1;

    if (b.Height > root.Height)
        c.Height = b.Height + 1;
    else
        c.Height = root.Height + 1;

    UpdateHeight(c);
    return c;
}

/// <returns>Root of tree after turn</returns>
public static TreeNode<T> BigLeftTurn(TreeNode<T> root)
{
    TreeNode<T> w = root.Left;
    TreeNode<T> parent = root.Parent;
    TreeNode<T> b = root.Right;
    TreeNode<T> c = root.Right.Left;
    TreeNode<T> z = b.Right;
    TreeNode<T> x = c.Left;
    TreeNode<T> y = c.Right;
```

```
//Parents
c.Parent = parent;
b.Parent = c;
root.Parent = c;
if (w != null)
    w.Parent = root;
if (z != null)
    z.Parent = b;
if (y != null)
    y.Parent = b;
if (x != null)
    x.Parent = root;

//Childs
if (parent != null)
    if (parent.Right == root)
        parent.Right = c;
    else
        parent.Left = c;
c.Left = root;
c.Right = b;
b.Left = y;
b.Right = z;
root.Left = w;
root.Right = x;

//Heights
long xH = x != null ? x.Height : -1;
long yH = y != null ? y.Height : -1;
long zH = z != null ? z.Height : -1;
long wH = w != null ? w.Height : -1;

if (wH > xH)
    root.Height = wH + 1;
else
    root.Height = xH + 1;
```

```
            if (yH > zH)
                b.Height = yH + 1;
            else
                b.Height = zH + 1;

            if (b.Height > root.Height)
                c.Height = b.Height + 1;
            else
                c.Height = root.Height + 1;

            UpdateHeight(c);
            return c;
        }
    }
  }
}
```

Результат работы

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.546	50700288	4011957	4011966
1	OK	0.031	11014144	20	24
2	OK	0.031	11005952	6	10
3	OK	0.031	10960896	14	18
4	OK	0.031	10973184	13	17
5	OK	0.031	10985472	21	25
6	OK	0.031	10948608	20	24
7	OK	0.046	11071488	20	24
8	OK	0.031	10977280	21	25

Задание №4

Удаление из АВЛ-дерева

1.0 из 1.0 балла (оценивается)

Имя входного файла:	input.txt
Имя выходного файла:	output.txt
Ограничение по времени:	2 секунды
Ограничение по памяти:	256 мегабайт

Удаление из АВЛ-дерева вершины с ключом X, при условии ее наличия, осуществляется следующим образом:

- путем спуска от корня и проверки ключей находится V — удаляемая вершина;
- если вершина V — лист (то есть, у нее нет детей):
  - удаляем вершину;
  - поднимаемся к корню, начиная с бывшего родителя вершины V, при этом если встречается несбалансированная вершина, то производим поворот.
- если у вершины V не существует левого ребенка:
  - следовательно, баланс вершины равен единице и ее правый ребенок — лист;
  - заменяем вершину V ее правым ребенком;
  - поднимаемся к корню, производя, где необходимо, балансировку.
- иначе:
  - находим R — самую правую вершину в левом поддереве;
  - переносим ключ вершины R в вершину V;
  - удаляем вершину R (у нее нет правого ребенка, поэтому она либо лист, либо имеет левого ребенка, являющегося листом);
  - поднимаемся к корню, начиная с бывшего родителя вершины R, производя балансировку.

Исключением является случай, когда производится удаление из дерева, состоящего из одной вершины — корня. Результатом удаления в этом случае будет пустое дерево.

Указанный алгоритм не является единственно возможным, но мы просим Вас реализовать именно его, так как тестирующая система проверяет точное равенство получающихся деревьев.

Решение

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Week7
{
    class Task4
```

```
{
    public static void Main(string[] args)
    {
        var app = new Task4();
        app.DoWork(args);
    }

    private void DoWork(string[] args)
    {
        using (var sw = new StreamWriter("output.txt"))
        {
            string[] stdin = File.ReadAllLines("input.txt");
            int n = int.Parse(stdin[0]);

            TreeNode<long> root = null;
            for (int i = 1; i <= n; i++)
                root = TreeNode<long>.Insert(root, new TreeNode<long> { Key = long.Parse(stdin[i].Split('
')[0]) });

            for (int i = n + 1; i < stdin.Length; i++)
            {
                TreeNode<long> node = TreeNode<long>.Search(root, long.Parse(stdin[i]));
                if (node != null)
                    root = TreeNode<long>.Remove(node);
            }

            sw.WriteLine(n - (stdin.Length - 1 - n));
            TreeNode<long>.PrintTree(sw, root);
        }
    }

    class TreeNode<T> where T : IComparable<T>
    {
        public T Key { get; set; }
        public TreeNode<T> Parent { get; set; }
        public TreeNode<T> Left { get; set; }
        public TreeNode<T> Right { get; set; }

        private long Depth { get; set; }
        public long Height { get; private set; }

        public static TreeNode<T> Search(TreeNode<T> root, T key)
        {
            while (root != null && root.Key.CompareTo(key) != 0)
                if (root.Key.CompareTo(key) > 0)
                    root = root.Left;
                else
                    root = root.Right;

            return root;
        }

        public static TreeNode<T> Previous(TreeNode<T> node)
        {
            if (node.Left == null)
                return node;
            return Maximum(node.Left);
```

```
}

public static TreeNode<T> Maximum(TreeNode<T> node)
{
   while (node.Right != null)
      node = node.Right;
   return node;
}

/// <returns>Root of tree after remove</returns>
public static TreeNode<T> Remove(TreeNode<T> item)
{
   TreeNode<T> parent = item.Parent;

   //Leaf
   if (item.Left == null && item.Right == null)
   {
      if (parent == null)
         return null;
      if (parent.Left == item)
         parent.Left = null;
      else
         parent.Right = null;

      UpdateHeight(parent);
      return Balance(parent);
   }

   //One child
   if ((item.Left == null) ^ (item.Right == null))
      if (item.Left != null)
      {
         if (parent != null)
         {
            if (parent.Left == item)
               parent.Left = item.Left;
            else
               parent.Right = item.Left;

            UpdateHeight(parent);
         }

         item.Left.Parent = parent;
         return Balance(item.Left);
      }
      else
      {
         if (parent != null)
         {
            if (parent.Left == item)
               parent.Left = item.Right;
            else
               parent.Right = item.Right;

            UpdateHeight(parent);
         }
```

```
            item.Right.Parent = parent;
            return Balance(item.Right);
        }


    //Two child
    if ((item.Left != null) && (item.Right != null))
    {
        TreeNode<T> prev = Previous(item);
        Remove(prev);
        item.Key = prev.Key;
    }

    return Balance(item);
}

/// <returns>Root of tree after insert</returns>
public static TreeNode<T> Insert(TreeNode<T> root, TreeNode<T> node)
{
    if (root == null)
        return node;
    TreeNode<T> current = root;
    while (true)
    {
        if (current.Key.CompareTo(node.Key) == 0)
            throw new ArgumentException("Not unique key");
        if (current.Key.CompareTo(node.Key) < 0)
        {
            if (current.Right != null)
                current = current.Right;
            else
            {
                current.Right = node;
                node.Parent = current;
                UpdateHeight(node);
                return root;
                //return Balance(node);
            }
        }
        else
        {
            if (current.Left != null)
                current = current.Left;
            else
            {
                current.Left = node;
                node.Parent = current;
                UpdateHeight(node);
                return root;
                //return Balance(node);
            }
        }
    }
}

private static void UpdateHeight(TreeNode<T> node)
{
```

```
    while (node != null)
    {
        long rH = node.Right != null ? node.Right.Height : -1;
        long lH = node.Left != null ? node.Left.Height : -1;

        long currentH = node.Height;
        if (rH > lH)
            node.Height = rH + 1;
        else
            node.Height = lH + 1;

        node = node.Parent;
    }
}

/// <returns>Root of tree after balance</returns>
public static TreeNode<T> Balance(TreeNode<T> leaf)
{
    TreeNode<T> current = leaf;
    while (current != null)
    {
        long balance = GetBalance(current);
        if (balance > 1)
        {
            if (GetBalance(current.Right) == -1)
                current = BigLeftTurn(current);
            else
                current = SmallLeftTurn(current);
        }
        if (balance < -1)
        {
            if (GetBalance(current.Left) == 1)
                current = BigRightTurn(current);
            else
                current = SmallRightTurn(current);
        }
        if (current.Parent == null)
            return current;
        else
            current = current.Parent;
    }
    return current;
}

public static void PrintTree(StreamWriter sw, TreeNode<T> root)
{
    if (root == null)
        return;
    Queue<TreeNode<T>> bfsQueue = new Queue<TreeNode<T>>();
    long counter = 1;
    bfsQueue.Enqueue(root);
    while (bfsQueue.Count != 0)
    {
        TreeNode<T> current = bfsQueue.Dequeue();
        sw.Write(current.Key);

        if (current.Left != null)
```

```
        {
            bfsQueue.Enqueue(current.Left);
            sw.Write(" " + ++counter);
        }
        else
            sw.Write(" " + 0);

        if (current.Right != null)
        {
            bfsQueue.Enqueue(current.Right);
            sw.WriteLine(" " + ++counter);
        }
        else
            sw.WriteLine(" " + 0);
    }
}

public static long GetBalance(TreeNode<T> tree)
{
    if (tree == null)
        return 0;

    if (tree.Left != null && tree.Right != null)
        return tree.Right.Height - tree.Left.Height;
    if (tree.Left == null && tree.Right != null)
        return tree.Right.Height + 1;
    if (tree.Left != null && tree.Right == null)
        return -1 - tree.Left.Height;
    else
        return 0;
}

/// <returns>Root of tree after turn</returns>
public static TreeNode<T> SmallLeftTurn(TreeNode<T> root)
{
    TreeNode<T> child = root.Right;
    TreeNode<T> parent = root.Parent;
    TreeNode<T> x = root.Left;
    TreeNode<T> y = root.Right.Left;
    TreeNode<T> z = root.Right.Right;

    //Parents
    child.Parent = parent;
    root.Parent = child;
    if (x != null)
        x.Parent = root;
    if (y != null)
        y.Parent = root;
    if (z != null)
        z.Parent = child;

    //Childs
    root.Left = x;
    root.Right = y;
    child.Left = root;
    child.Right = z;
    if (parent != null)
```

```
        if (parent.Right == root)
            parent.Right = child;
        else
            parent.Left = child;

    //Heights
    long xH = x != null ? x.Height : -1;
    long yH = y != null ? y.Height : -1;
    long zH = z != null ? z.Height : -1;

    if (xH > yH)
        root.Height = xH + 1;
    else
        root.Height = yH + 1;
    if (root.Height > zH)
        child.Height = root.Height + 1;
    else
        child.Height = zH + 1;

    UpdateHeight(child);
    return child;
}

/// <returns>Root of tree after turn</returns>
public static TreeNode<T> SmallRightTurn(TreeNode<T> root)
{
    TreeNode<T> child = root.Left;
    TreeNode<T> parent = root.Parent;
    TreeNode<T> x = root.Right;
    TreeNode<T> y = root.Left.Left;
    TreeNode<T> z = root.Left.Right;

    //Parents
    child.Parent = parent;
    root.Parent = child;
    if (x != null)
        x.Parent = root;
    if (y != null)
        y.Parent = child;
    if (z != null)
        z.Parent = root;

    //Childs
    root.Left = z;
    root.Right = x;
    child.Left = y;
    child.Right = root;
    if (parent != null)
        if (parent.Right == root)
            parent.Right = child;
        else
            parent.Left = child;

    //Heights
    long xH = x != null ? x.Height : -1;
    long yH = y != null ? y.Height : -1;
    long zH = z != null ? z.Height : -1;
```

```
    if (zH > xH)
        root.Height = zH + 1;
    else
        root.Height = xH + 1;

    if (y.Height > root.Height)
        child.Height = yH + 1;
    else
        child.Height = root.Height + 1;

    UpdateHeight(child);
    return child;
}

/// <returns>Root of tree after turn</returns>
public static TreeNode<T> BigRightTurn(TreeNode<T> root)
{
    TreeNode<T> w = root.Right;
    TreeNode<T> parent = root.Parent;
    TreeNode<T> b = root.Left;
    TreeNode<T> c = root.Left.Right;
    TreeNode<T> z = b.Left;
    TreeNode<T> x = c.Left;
    TreeNode<T> y = c.Right;

    //Parents
    c.Parent = parent;
    b.Parent = c;
    root.Parent = c;
    if (w != null)
        w.Parent = root;
    if (z != null)
        z.Parent = b;
    if (y != null)
        y.Parent = root;
    if (x != null)
        x.Parent = b;

    //Childs
    if (parent != null)
        if (parent.Right == root)
            parent.Right = c;
        else
            parent.Left = c;
    c.Left = b;
    c.Right = root;
    b.Left = z;
    b.Right = x;
    root.Left = y;
    root.Right = w;

    //Heights
    long xH = x != null ? x.Height : -1;
    long yH = y != null ? y.Height : -1;
    long zH = z != null ? z.Height : -1;
    long wH = w != null ? w.Height : -1;
```

```
    if (zH > xH)
        b.Height = zH + 1;
    else
        b.Height = xH + 1;

    if (yH > wH)
        root.Height = yH + 1;
    else
        root.Height = wH + 1;

    if (b.Height > root.Height)
        c.Height = b.Height + 1;
    else
        c.Height = root.Height + 1;

    UpdateHeight(c);
    return c;
}

/// <returns>Root of tree after turn</returns>
public static TreeNode<T> BigLeftTurn(TreeNode<T> root)
{
    TreeNode<T> w = root.Left;
    TreeNode<T> parent = root.Parent;
    TreeNode<T> b = root.Right;
    TreeNode<T> c = root.Right.Left;
    TreeNode<T> z = b.Right;
    TreeNode<T> x = c.Left;
    TreeNode<T> y = c.Right;

    //Parents
    c.Parent = parent;
    b.Parent = c;
    root.Parent = c;
    if (w != null)
        w.Parent = root;
    if (z != null)
        z.Parent = b;
    if (y != null)
        y.Parent = b;
    if (x != null)
        x.Parent = root;

    //Childs
    if (parent != null)
        if (parent.Right == root)
            parent.Right = c;
        else
            parent.Left = c;
    c.Left = root;
    c.Right = b;
    b.Left = y;
    b.Right = z;
    root.Left = w;
    root.Right = x;
```

```
            //Heights
            long xH = x != null ? x.Height : -1;
            long yH = y != null ? y.Height : -1;
            long zH = z != null ? z.Height : -1;
            long wH = w != null ? w.Height : -1;

            if (wH > xH)
               root.Height = wH + 1;
            else
               root.Height = xH + 1;

            if (yH > zH)
               b.Height = yH + 1;
            else
               b.Height = zH + 1;

            if (b.Height > root.Height)
               c.Height = b.Height + 1;
            else
               c.Height = root.Height + 1;

            UpdateHeight(c);
            return c;
         }
      }
   }
}
```

Результат работы

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.531	50827264	4077288	4077255
1	OK	0.031	10960896	27	17
2	OK	0.031	10940416	13	3
3	OK	0.046	11145216	20	10
4	OK	0.031	10956800	20	10
5	OK	0.031	11001856	20	10
6	OK	0.031	10977280	20	10
7	OK	0.015	10944512	27	17
8	OK	0.046	11153408	27	17

Задание №5

Упорядоченное множество на АВЛ-дереве

1.0 из 1.0 балла (оценивается)

Имя входного файла:	input.txt
Имя выходного файла:	output.txt
Ограничение по времени:	2 секунды
Ограничение по памяти:	256 мегабайт

Если Вы сдали все предыдущие задачи, Вы уже можете написать эффективную реализацию упорядоченного множества на АВЛ-дереве. Сделайте это.

Для проверки того, что множество реализовано именно на АВЛ-дереве, мы просим Вас выводить баланс корня после каждой операции вставки и удаления.

Операции вставки и удаления требуется реализовать точно так же, как это было сделано в предыдущих двух задачах, потому что в ином случае баланс корня может отличаться от требуемого.

Решение

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;


namespace Week7
{
   class Task5
   {
      public static void Main(string[] args)
      {
         var app = new Task5();
         app.DoWork(args);
      }


      private void DoWork(string[] args)
      {
         using (var sw = new StreamWriter("output.txt"))
         {
            string[] stdin = File.ReadAllLines("input.txt");
```

```
        TreeNode<long> root = null;

        for (int i = 1; i < stdin.Length; i++)
        {
            string[] temp = stdin[i].Split(' ');
            switch (temp[0])
            {
                case "A":
                    TreeNode<long> s = TreeNode<long>.Search(root, long.Parse(temp[1]));
                    if (s == null)
                        root = TreeNode<long>.Insert(root, new TreeNode<long> { Key = long.Parse(temp[1]) });
                    sw.WriteLine(TreeNode<long>.GetBalance(root));
                    break;
                case "D":
                    TreeNode<long> t = TreeNode<long>.Search(root, long.Parse(temp[1]));
                    if (t != null)
                        root = TreeNode<long>.Remove(t);
                    sw.WriteLine(TreeNode<long>.GetBalance(root));
                    break;
                case "C":
                    TreeNode<long> x = TreeNode<long>.Search(root, long.Parse(temp[1]));
                    if (x != null)
                        sw.WriteLine("Y");
                    else
                        sw.WriteLine("N");
                    break;
            }
        }
    }
}

class TreeNode<T> where T : IComparable<T>
{
    public T Key { get; set; }
    public TreeNode<T> Parent { get; set; }
    public TreeNode<T> Left { get; set; }
    public TreeNode<T> Right { get; set; }
```

```
public long Height { get; set; }

public static TreeNode<T> Next(TreeNode<T> node)
{
   if (node.Right == null)
      return node;
   return Minimum(node.Right);
}

public static TreeNode<T> Previous(TreeNode<T> node)
{
   if (node.Left == null)
      return node;
   return Maximum(node.Left);
}

public static TreeNode<T> Maximum(TreeNode<T> node)
{
   while (node.Right != null)
      node = node.Right;
   return node;
}

public static TreeNode<T> Minimum(TreeNode<T> node)
{
   while (node.Left != null)
      node = node.Left;
   return node;
}

/// <returns>Root of tree after remove</returns>
public static TreeNode<T> Remove(TreeNode<T> item)
{
   TreeNode<T> parent = item.Parent;

   //Leaf
```

```
if (item.Left == null && item.Right == null)
{
    if (parent == null)
        return null;
    if (parent.Left == item)
        parent.Left = null;
    else
        parent.Right = null;

    UpdateHeight(parent);
    return Balance(parent);
}

//One child
if ((item.Left == null) ^ (item.Right == null))
    if (item.Left != null)
    {
        if (parent != null)
        {
            if (parent.Left == item)
                parent.Left = item.Left;
            else
                parent.Right = item.Left;

            UpdateHeight(parent);
        }

        item.Left.Parent = parent;
        return Balance(item.Left);
    }
    else
    {
        if (parent != null)
        {
            if (parent.Left == item)
                parent.Left = item.Right;
            else
```

```
                parent.Right = item.Right;

            UpdateHeight(parent);
        }

        item.Right.Parent = parent;
        return Balance(item.Right);
    }


  //Two child
  if ((item.Left != null) && (item.Right != null))
  {
    TreeNode<T> prev = Previous(item);
    Remove(prev);
    item.Key = prev.Key;
  }

  return Balance(item);
}

/// <returns>Root of tree after insert</returns>
public static TreeNode<T> Insert(TreeNode<T> root, TreeNode<T> node)
{
  if (root == null)
    return node;
  TreeNode<T> current = root;
  while (true)
  {
    if (current.Key.CompareTo(node.Key) == 0)
      throw new ArgumentException("Not unique key");
    if (current.Key.CompareTo(node.Key) < 0)
    {
      if (current.Right != null)
        current = current.Right;
      else
      {
```

```
                current.Right = node;
                node.Parent = current;
                UpdateHeight(node);
                return Balance(node);
            }
        }
        else
        {
            if (current.Left != null)
                current = current.Left;
            else
            {
                current.Left = node;
                node.Parent = current;
                UpdateHeight(node);
                return Balance(node);
            }
        }
    }
}

private static void UpdateHeight(TreeNode<T> node)
{
    while (node != null)
    {
        long rH = node.Right != null ? node.Right.Height : -1;
        long lH = node.Left != null ? node.Left.Height : -1;

        long currentH = node.Height;
        if (rH > lH)
            node.Height = rH + 1;
        else
            node.Height = lH + 1;

        node = node.Parent;
    }
}
```

```
public static TreeNode<T> Search(TreeNode<T> root, T key)
{
    while (root != null && root.Key.CompareTo(key) != 0)
        if (root.Key.CompareTo(key) > 0)
            root = root.Left;
        else
            root = root.Right;

    return root;
}

/// <returns>Root of tree after balance</returns>
public static TreeNode<T> Balance(TreeNode<T> leaf)
{
    TreeNode<T> current = leaf;
    while (current != null)
    {
        long balance = GetBalance(current);
        if (balance > 1)
        {
            if (GetBalance(current.Right) == -1)
                current = BigLeftTurn(current);
            else
                current = SmallLeftTurn(current);
        }
        if (balance < -1)
        {
            if (GetBalance(current.Left) == 1)
                current = BigRightTurn(current);
            else
                current = SmallRightTurn(current);
        }
        if (current.Parent == null)
            return current;
        else
            current = current.Parent;
```

```
    }
    return current;
}

public static void PrintTree(TreeNode<T> root)
{
    if (root == null)
        return;
    Queue<TreeNode<T>> bfsQueue = new Queue<TreeNode<T>>();
    long counter = 1;
    bfsQueue.Enqueue(root);
    while (bfsQueue.Count != 0)
    {
        TreeNode<T> current = bfsQueue.Dequeue();
        Console.Write(current.Key);

        if (current.Left != null)
        {
            bfsQueue.Enqueue(current.Left);
            Console.Write(" " + ++counter);
        }
        else
            Console.Write(" " + 0);

        if (current.Right != null)
        {
            bfsQueue.Enqueue(current.Right);
            Console.WriteLine(" " + ++counter);
        }
        else
            Console.WriteLine(" " + 0);
    }
}

public static long GetBalance(TreeNode<T> tree)
{
    if (tree == null)
```

```
        return 0;

    if (tree.Left != null && tree.Right != null)
        return tree.Right.Height - tree.Left.Height;
    if (tree.Left == null && tree.Right != null)
        return tree.Right.Height + 1;
    if (tree.Left != null && tree.Right == null)
        return -1 - tree.Left.Height;
    else
        return 0;
}

/// <returns>Root of tree after turn</returns>
public static TreeNode<T> SmallLeftTurn(TreeNode<T> root)
{
    TreeNode<T> child = root.Right;
    TreeNode<T> parent = root.Parent;
    TreeNode<T> x = root.Left;
    TreeNode<T> y = root.Right.Left;
    TreeNode<T> z = root.Right.Right;

    //Parents
    child.Parent = parent;
    root.Parent = child;
    if (x != null)
        x.Parent = root;
    if (y != null)
        y.Parent = root;
    if (z != null)
        z.Parent = child;

    //Childs
    root.Left = x;
    root.Right = y;
    child.Left = root;
    child.Right = z;
    if (parent != null)
```

```
        if (parent.Right == root)
            parent.Right = child;
        else
            parent.Left = child;

    //Heights
    long xH = x != null ? x.Height : -1;
    long yH = y != null ? y.Height : -1;
    long zH = z != null ? z.Height : -1;

    if (xH > yH)
        root.Height = xH + 1;
    else
        root.Height = yH + 1;
    if (root.Height > zH)
        child.Height = root.Height + 1;
    else
        child.Height = zH + 1;

    UpdateHeight(child);
    return child;
}

/// <returns>Root of tree after turn</returns>
public static TreeNode<T> SmallRightTurn(TreeNode<T> root)
{
    TreeNode<T> child = root.Left;
    TreeNode<T> parent = root.Parent;
    TreeNode<T> x = root.Right;
    TreeNode<T> y = root.Left.Left;
    TreeNode<T> z = root.Left.Right;

    //Parents
    child.Parent = parent;
    root.Parent = child;
    if (x != null)
        x.Parent = root;
```

```
    if (y != null)
        y.Parent = child;
    if (z != null)
        z.Parent = root;

    //Childs
    root.Left = z;
    root.Right = x;
    child.Left = y;
    child.Right = root;
    if (parent != null)
        if (parent.Right == root)
            parent.Right = child;
        else
            parent.Left = child;

    //Heights
    long xH = x != null ? x.Height : -1;
    long yH = y != null ? y.Height : -1;
    long zH = z != null ? z.Height : -1;

    if (zH > xH)
        root.Height = zH + 1;
    else
        root.Height = xH + 1;

    if (y.Height > root.Height)
        child.Height = yH + 1;
    else
        child.Height = root.Height + 1;

    UpdateHeight(child);
    return child;
}

/// <returns>Root of tree after turn</returns>
public static TreeNode<T> BigRightTurn(TreeNode<T> root)
```

```
{
    TreeNode<T> w = root.Right;
    TreeNode<T> parent = root.Parent;
    TreeNode<T> b = root.Left;
    TreeNode<T> c = root.Left.Right;
    TreeNode<T> z = b.Left;
    TreeNode<T> x = c.Left;
    TreeNode<T> y = c.Right;

    //Parents
    c.Parent = parent;
    b.Parent = c;
    root.Parent = c;
    if (w != null)
        w.Parent = root;
    if (z != null)
        z.Parent = b;
    if (y != null)
        y.Parent = root;
    if (x != null)
        x.Parent = b;

    //Childs
    if (parent != null)
        if (parent.Right == root)
            parent.Right = c;
        else
            parent.Left = c;
    c.Left = b;
    c.Right = root;
    b.Left = z;
    b.Right = x;
    root.Left = y;
    root.Right = w;

    //Heights
    long xH = x != null ? x.Height : -1;
```

```
    long yH = y != null ? y.Height : -1;
    long zH = z != null ? z.Height : -1;
    long wH = w != null ? w.Height : -1;

    if (zH > xH)
        b.Height = zH + 1;
    else
        b.Height = xH + 1;

    if (yH > wH)
        root.Height = yH + 1;
    else
        root.Height = wH + 1;

    if (b.Height > root.Height)
        c.Height = b.Height + 1;
    else
        c.Height = root.Height + 1;

    UpdateHeight(c);
    return c;
}

/// <returns>Root of tree after turn</returns>
public static TreeNode<T> BigLeftTurn(TreeNode<T> root)
{
    TreeNode<T> w = root.Left;
    TreeNode<T> parent = root.Parent;
    TreeNode<T> b = root.Right;
    TreeNode<T> c = root.Right.Left;
    TreeNode<T> z = b.Right;
    TreeNode<T> x = c.Left;
    TreeNode<T> y = c.Right;

    //Parents
    c.Parent = parent;
    b.Parent = c;
```

```
root.Parent = c;
if (w != null)
    w.Parent = root;
if (z != null)
    z.Parent = b;
if (y != null)
    y.Parent = b;
if (x != null)
    x.Parent = root;

//Childs
if (parent != null)
    if (parent.Right == root)
        parent.Right = c;
    else
        parent.Left = c;
c.Left = root;
c.Right = b;
b.Left = y;
b.Right = z;
root.Left = w;
root.Right = x;

//Heights
long xH = x != null ? x.Height : -1;
long yH = y != null ? y.Height : -1;
long zH = z != null ? z.Height : -1;
long wH = w != null ? w.Height : -1;

if (wH > xH)
    root.Height = wH + 1;
else
    root.Height = xH + 1;

if (yH > zH)
    b.Height = yH + 1;
else
```

```
                b.Height = zH + 1;

            if (b.Height > root.Height)
                c.Height = b.Height + 1;
            else
                c.Height = root.Height + 1;

            UpdateHeight(c);
            return c;
        }
    }
  }
}
```

Результат работы

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.656	46452736	2678110	731071
1	OK	0.031	10563584	33	19
2	OK	0.031	10428416	114	66
3	OK	0.031	10387456	154	90
4	OK	0.031	10551296	154	91
5	OK	0.031	10350592	154	90
6	OK	0.031	10551296	154	95
7	OK	0.031	10547200	154	91
8	OK	0.046	10457088	154	94
9	OK	0.046	10543104	154	95
10	OK	0.031	10547200	154	90
11	OK	0.015	10416128	154	90
12	OK	0.031	10530816	154	90