Санкт-Петербургский национальный исследовательский университет

Информационных технологий, механики и оптики

Отчет по решению задач седьмой недели
По курсу «Алгоритмы и структуры данных»

на Openedu

Выполнил: Сыроватский Павел Валентинович

Группа P3218

Санкт-Петербург

2019

Проверка сбалансированности

0 возможных балла (не 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:

$$-1 \le B(V) \le 1$$

Обратите внимание, что, по историческим причинам, определение баланса в этой и последующих задачах этой недели "зеркально отражено" по сравнению с определением баланса в лекциях! Надеемся, что этот факт не доставит Вам неудобств. В литературе по алгоритмам — как российской, так и мировой — ситуация, как правило, примерно та же.

Дано двоичное дерево поиска. Для каждой его вершины требуется определить ее баланс.

Формат входного файла

Входной файл содержит описание двоичного дерева. В первой строке файла находится число $N (1 \le N \le 2 \cdot 10^5)$ — число вершин в дереве. В последующих N строках файла находятся описания вершин дерева. В $(i+1)$-ой строке файла $(1 \le i \le N)$ находится описание i-ой вершины, состоящее из трех чисел K_i, L_i, R_i, разделенных пробелами — ключа в i-ой вершине $(|K_i| \le 10^9)$, номера левого ребенка i-ой вершины ($i < L_i \le N$ или $L_i = 0$, если левого ребенка нет) и номера правого ребенка i-ой вершины ($i < R_i \le N$ или $R_i = 0$, если правого ребенка нет).

Все ключи различны. Гарантируется, что данное дерево является деревом поиска.

Формат выходного файла

Для i-ой вершины в i-ой строке выведите одно число — баланс данной вершины

Реализация на C++

```
#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);
}

//Найти высоту i элемента
int find(int x) {
	int i = 0;

	while (tree[i].key != x) {
		if (x < tree[i].key) {
			if (tree[i].left) {
				i = tree[i].left - 1;
			}
			else {
				return -1;
			}
		}
		else {
			if (tree[i].right) {
				i = tree[i].right - 1;
			}
			else {
```

```
                        return -1;
                    }
                }
            }

            return i;
}


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

            tree = new t_node[sz = n];
            h = new int[n];
            b = new int[n];
            //Заполняем дерево
            for (int i = 0; i < n; ++i) {
                        cin >> tree[i].key >> tree[i].left >> tree[i].right;

                        h[i] = 0;
            }

            //Заполняем массив высот
            cnt(0);

            for (int i = 0; i < n; ++i) {
                        cout << b[i] << nl;
            }


            return 0;
}
```
Результат выполнения первой задачи

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.015	2383872	25	11
1	OK	0.000	2371584	7	2
2	OK	0.015	2367488	8	3
3	OK	0.000	2383872	5	1
4	OK	0.015	2371584	5	1
5	OK	0.000	2383872	6	1
6	OK	0.000	2371584	9	4
7	OK	0.000	2367488	23	10
8	OK	0.000	2371584	25	11
9	OK	0.000	2371584	24	1
10	OK	0.000	2371584	24	1
11	OK	0.000	2371584	14	10
12	OK	0.015	2371584	23	10
13	OK	0.015	2371584	23	11
14	OK	0.000	2371584	20	9
15	OK	0.015	2371584	23	11
16	OK	0.015	2371584	20	9
17	OK	0.000	2371584	22	10
18	OK	0.000	2367488	23	11
19	OK	0.000	2371584	22	10
20	OK	0.000	2371584	22	10
21	OK	0.015	2371584	22	10

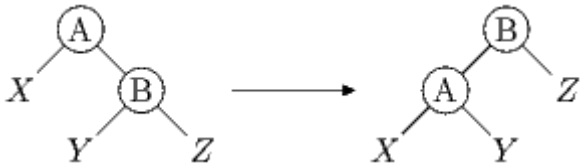
Делаю я левый поворот...

0 возможных балла (не оценивается)

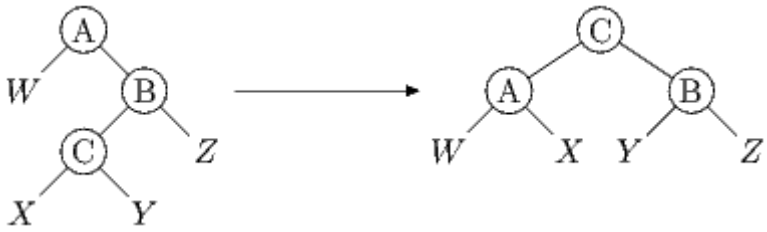
Для балансировки АВЛ-дерева при операциях вставки и удаления производятся *левые* и *правые*повороты. Левый поворот в вершине производится, когда баланс этой вершины больше 1, аналогично, правый поворот производится при балансе, меньшем −1.

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:



Дано дерево, в котором баланс корня равен 2. Сделайте левый поворот.

Формат входного файла

Входной файл содержит описание двоичного дерева. В первой строке файла находится число N ($3 \le N \le 2 \cdot 10^5$) — число вершин в дереве. В последующих N строках файла находятся описания вершин дерева. В $(i+1)$-ой строке файла ($1 \le i \le N$) находится описание i-ой вершины, состоящее из трех чисел K_i, L_i, R_i, разделенных пробелами — ключа в i-ой вершине ($|K_i| \le 10^9$), номера левого ребенка i-ой вершины ($i < L_i \le N$ или $L_i = 0$, если левого ребенка нет) и номера правого ребенка i-ой вершины ($i < R_i \le N$ или $R_i = 0$, если правого ребенка нет). Все ключи различны. Гарантируется, что данное дерево является деревом поиска. Баланс корня дерева (вершины с номером 1) равен 2, баланс всех остальных вершин находится в пределах от -1 до 1.

Формат выходного файла

Выведите в том же формате дерево после осуществления левого поворота. Нумерация вершин может быть произвольной при условии соблюдения формата. Так, номер вершины должен быть меньше номера ее детей.

Реализация программы на C++

```
#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);
}

int find(int x) {
	int i = 0;

	while (tree[i].key != x) {
		if (x < tree[i].key) {
			if (tree[i].left) {
```

```
                        i = tree[i].left - 1;
                }
                else {
                        return -1;
                }
        }
        else {
                if (tree[i].right) {
                        i = tree[i].right - 1;
                }
                else {
                        return -1;
                }
        }
    }

    return i;
}

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 решения задачи

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.031	2387968	25	19
1	OK	0.015	2367488	7	3
2	OK	0.031	2371584	8	3
3	OK	0.000	2371584	5	1
4	OK	0.015	2371584	5	1
5	OK	0.000	2383872	6	1
6	OK	0.015	2371584	6	1
7	OK	0.000	2371584	23	19
8	OK	0.015	2367488	25	18
9	OK	0.000	2387968	24	18
10	OK	0.000	2383872	24	19
11	OK	0.015	2371584	23	18
12	OK	0.015	2371584	23	18
13	OK	0.000	2371584	20	15
14	OK	0.000	2383872	23	18
15	OK	0.000	2371584	20	18
16	OK	0.000	2367488	22	18
17	OK	0.000	2383872	23	18
18	OK	0.000	2371584	22	17
19	OK	0.000	2367488	22	17
20	OK	0.000	2367488	22	18

Вставка в АВЛ-дерево

0 возможных балла (не 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.

Первый этап нуждается в пояснении. Спуск до будущего родителя вершины V осуществляется, начиная от корня, следующим образом:

- Пусть ключ текущей вершины равен Y.
- Если $X<Y$ и у текущей вершины есть левый ребенок, переходим к левому ребенку.
- Если $X<Y$ и у текущей вершины нет левого ребенка, то останавливаемся, текущая вершина будет родителем новой вершины.
- Если $X>Y$ и у текущей вершины есть правый ребенок, переходим к правому ребенку.
- Если $X>Y$ и у текущей вершины нет правого ребенка, то останавливаемся, текущая вершина будет родителем новой вершины.

Отдельно рассматривается следующий крайний случай — если до вставки дерево было пустым, то вставка новой вершины осуществляется проще: новая вершина становится корнем дерева.

Формат входного файла

Входной файл содержит описание двоичного дерева, а также ключа вершины, которую требуется вставить в дерево.

В первой строке файла находится число N ($0 \le N \le 2 \cdot 10^5$) — число вершин в дереве. В последующих N строках файла находятся описания вершин дерева. В $(i+1)$-ой строке файла ($1 \le i \le N$) находится описание i-ой вершины, состоящее из трех чисел K_i, L_i, R_i, разделенных пробелами — ключа в i-ой вершине ($|K_i| \le 10^9$), номера левого ребенка i-ой вершины ($i < L_i \le N$ или $L_i = 0$, если левого ребенка нет) и номера правого ребенка i-ой вершины ($i < R_i \le N$ или $R_i = 0$, если правого ребенка нет).

Все ключи различны. Гарантируется, что данное дерево является корректным АВЛ-деревом.

В последней строке содержится число X ($|X| \le 10^9$) — ключ вершины, которую требуется вставить в дерево. Гарантируется, что такой вершины в дереве нет.

Формат выходного файла

Выведите в том же формате дерево после осуществления операции вставки. Нумерация вершин может быть произвольной при условии соблюдения формата.

Реализация программы на C++

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


namespace Openedu.Week7
{
    class Lab7_3
    {
        public static void Main(string[] args)
        {
            var app = new Lab7_3();
            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 решения задачи

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.015	2375680	10415	14296
1	OK	0.000	2355200	25	40
2	OK	0.000	2363392	7	5
3	OK	0.000	2371584	12	12
4	OK	0.000	2355200	8	8
5	OK	0.000	2351104	10	12
6	OK	0.000	2367488	29	31
7	OK	0.000	2355200	10	12
8	OK	0.015	2355200	10	12
9	OK	0.000	2355200	10	12
10	OK	0.000	2351104	10	12
11	OK	0.000	2355200	10	12
12	OK	0.015	2355200	57	63
13	OK	0.000	2355200	56	62
14	OK	0.000	2355200	57	63
15	OK	0.015	2351104	77	87
16	OK	0.015	2355200	76	86
17	OK	0.000	2351104	77	87
18	OK	0.000	2355200	112	127
19	OK	0.015	2355200	111	127
20	OK	0.000	2367488	110	125
21	OK	0.015	2355200	949	1190
22	OK	0.000	2355200	960	1219
23	OK	0.000	2351104	957	1134
24	OK	0.000	2355200	1490	1888
25	OK	0.000	2351104	1486	1944
26	OK	0.000	2367488	1481	1761
27	OK	0.015	2355200	3723	4888
28	OK	0.000	2375680	3729	5047
29	OK	0.015	2355200	3727	4437
30	OK	0.015	2359296	8456	11338
31	OK	0.015	2359296	8471	11609
32	OK	0.015	2359296	8415	10035
33	OK	0.015	2359296	10415	14035
34	OK	0.015	2359296	10410	14296
35	OK	0.000	2375680	10393	12386