ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

Факультет программной инженерии и компьютерной техники

ЛАБОРАТОРНАЯ РАБОТА № 7

ПО ДИСЦИПЛИНЕ «Алгоритмы и структуры данных»

Выполнил: Мещеряков Владимир Алевтинович

Группа: P3218

Преподаватель: Романов Алексей Андреевич

Волчек Дмитрий Геннадьевич

Санкт-Петербург

2019 г.

Week 7

1)Упорядоченное множество на АВЛ-дереве

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.

Формат входного файла

В первой строке файла находится число N ($1 \le N \le 2 \cdot 10^5$) — число операций над множеством. Изначально множество пусто. В каждой из последующих N строк файла находится описание операции.

Операции бывают следующих видов:

- `A x` — вставить число x в множество. Если число x там уже содержится, множество изменять не следует.
- `D x` — удалить число x из множества. Если числа x нет в множестве, множество изменять не следует.
- `C x` — проверить, есть ли число x в множестве.

Формат выходного файла

Для каждой операции вида `C x` выведите `Y`, если число x содержится в множестве, и `N`, если не содержится.
Для каждой операции вида `A x` или `D x` выведите баланс корня дерева после выполнения операции. Если дерево пустое (в нем нет вершин), выведите 0.

Вывод для каждой операции должен содержаться на отдельной строке.

Пример

input.txt	output.txt
6 A 3 A 4 A 5 C 4 C 6 D 5	0 1 0 Y N -1

```
#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


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;
}


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

    tree = new t_node[sz = n];
    h = new int[n];
    b = new int[n];

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

        h[i] = 0;
    }

    cnt(0);

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


    return 0;
}
```

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.093	19304448	3986010	1688889
1	OK	0.015	2437120	46	19
2	OK	0.000	2453504	10	3
3	OK	0.000	2441216	17	6
4	OK	0.015	2437120	17	7
5	OK	0.000	2437120	24	9
6	OK	0.015	2437120	24	10
7	OK	0.015	2437120	24	9
8	OK	0.015	2437120	24	10
9	OK	0.000	2461696	24	11
10	OK	0.031	2437120	31	12
11	OK	0.000	2445312	31	13
12	OK	0.000	2433024	31	12
13	OK	0.000	2224128	31	13
14	OK	0.000	2215936	31	14
15	OK	0.031	2215936	31	12
16	OK	0.000	2220032	31	13
17	OK	0.000	2220032	31	13
18	OK	0.000	2232320	31	14
19	OK	0.031	2232320	31	13
20	OK	0.015	2220032	31	14
21	OK	0.031	2224128	31	13
22	OK	0.015	2215936	31	14
23	OK	0.015	2220032	31	15

2)Делаю я левый поворот

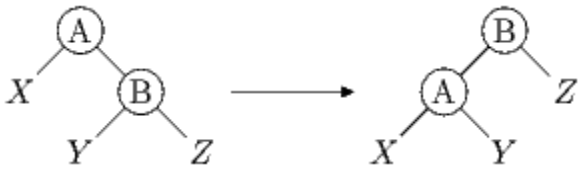
Имя входного файла:	input.txt
Имя выходного файла:	output.txt
Ограничение по времени:	2 секунды
Ограничение по памяти:	256 мегабайт

Для балансировки АВЛ-дерева при операциях вставки и удаления производятся *левые* и *правые*повороты. Левый поворот в вершине производится,

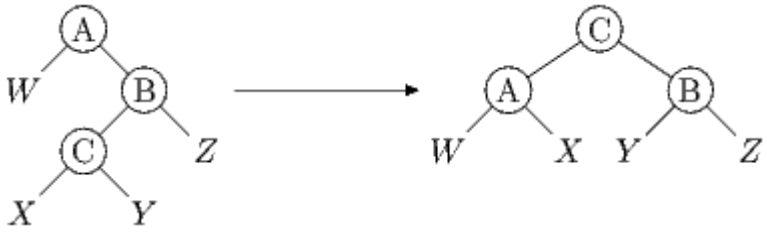
когда баланс этой вершины больше 1, аналогично, правый поворот производится при балансе, меньшем −1.

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:



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.

Пример

input.txt	output.txt

7 -2 7 2 8 4 3 9 0 0 3 6 5 6 0 0 0 0 0 -7 0 0	7 3 2 3 -2 4 5 8 6 7 -7 0 0 0 0 0 6 0 0 9 0 0

```
#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;
}
```

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.171	10612736	3986416	3986416
1	OK	0.015	2433024	54	54
2	OK	0.015	2449408	24	24
3	OK	0.000	2437120	24	24
4	OK	0.015	2437120	31	31
5	OK	0.000	2437120	45	45
6	OK	0.015	2445312	45	45
7	OK	0.000	2449408	45	45
8	OK	0.000	2441216	45	45
9	OK	0.015	2437120	52	52
10	OK	0.015	2433024	52	52
11	OK	0.015	2437120	52	52
12	OK	0.015	2441216	52	52
13	OK	0.015	2220032	52	52
14	OK	0.000	2220032	52	52
15	OK	0.000	2220032	59	59
16	OK	0.000	2215936	59	59
17	OK	0.000	2224128	59	59
18	OK	0.000	2224128	59	59
19	OK	0.000	2224128	66	66
20	OK	0.000	2220032	75	75
21	OK	0.015	2215936	75	75
22	OK	0.000	2220032	75	75
23	OK	0.000	2224128	75	75
24	OK	0.000	2220032	75	75
25	OK	0.000	2220032	75	75
26	OK	0.015	2220032	75	75

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.

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.

Пример

input.txt	output.txt
2 3 0 2 4 0 0 5	3 4 2 3 3 0 0 5 0 0

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

namespace Week7
{
    public class main
    {
        public static void Main(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;
}

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);
}


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;
        }
    }
}
```

4) Удаление из АВЛ-дерева

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.

Формат входного файла

Входной файл содержит описание двоичного дерева, а также ключа вершины, которую требуется удалить из 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.

В последней строке содержится число X ($|X| \le 10^9$) — ключ вершины, которую требуется удалить из дерева. Гарантируется, что такая вершина в дереве существует.

Формат выходного файла

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

Пример

input.txt	output.txt
3 4 2 3 3 0 0 5 0 0 4	2 3 0 2 5 0 0

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

namespace Week7
{
    public class main
    {
        public static void Main(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;
            }
        }
}
```

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.609	73056256	4077288	3877258
1	OK	0.046	11415552	27	17
2	OK	0.031	10825728	13	3
3	OK	0.031	11345920	20	11
4	OK	0.031	11354112	20	11
5	OK	0.031	11341824	20	11
6	OK	0.031	11354112	20	11
7	OK	0.046	11345920	27	17
8	OK	0.031	11403264	27	17
9	OK	0.031	11427840	27	17
10	OK	0.031	11390976	34	23
11	OK	0.046	11403264	34	23
12	OK	0.046	11382784	34	23
13	OK	0.031	11259904	34	23
14	OK	0.031	11206656	34	23
15	OK	0.031	11141120	34	23
16	OK	0.046	11169792	34	23
17	OK	0.031	11149312	34	23
18	OK	0.031	11165696	34	23
19	OK	0.046	11145216	34	23
20	OK	0.046	11157504	34	23
21	OK	0.031	11214848	34	23
22	OK	0.031	11268096	34	23
23	OK	0.031	11182080	34	23
24	OK	0.031	11223040	34	23
25	OK	0.046	11120640	34	23
26	OK	0.031	11137024	41	29
27	OK	0.031	11161600	41	29
28	OK	0.046	11169792	41	29
29	OK	0.031	11165696	41	29

5)Упорядоченное множество на АВЛ-дереве

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.

Формат входного файла

В первой строке файла находится число N ($1 \le N \le 2 \cdot 10^5$) — число операций над множеством. Изначально множество пусто. В каждой из последующих N строк файла находится описание операции.

Операции бывают следующих видов:

- `A x` — вставить число x в множество. Если число x там уже содержится, множество изменять не следует.
- `D x` — удалить число x из множества. Если числа x нет в множестве, множество изменять не следует.
- `C x` — проверить, есть ли число x в множестве.

Формат выходного файла

Для каждой операции вида `C x` выведите `Y`, если число x содержится в множестве, и `N`, если не содержится.
Для каждой операции вида `A x` или `D x` выведите баланс корня дерева после выполнения операции. Если дерево пустое (в нем нет вершин), выведите 0.

Вывод для каждой операции должен содержаться на отдельной строке.

Пример

input.txt	output.txt
6 A 3 A 4 A 5 C 4	0 1 0 Y N -1

C 6 D 5	

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

namespace Week7
{
    public class main
    {
        public static void Main(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;
            }
        }
    }
}
```

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		1.703	45563904	2678110	731071
1	OK	0.046	12173312	33	19
2	OK	0.046	12550144	114	66
3	OK	0.046	12533760	154	90
4	OK	0.046	12574720	154	91
5	OK	0.046	12673024	154	90
6	OK	0.031	12648448	154	95
7	OK	0.062	12566528	154	91
8	OK	0.046	12636160	154	94
9	OK	0.046	12652544	154	95
10	OK	0.046	12627968	154	90
11	OK	0.046	12705792	154	90
12	OK	0.046	11743232	154	90
13	OK	0.031	12472320	154	95
14	OK	0.031	12484608	154	97
15	OK	0.031	12521472	154	94
16	OK	0.046	12460032	154	93
17	OK	0.031	12374016	154	90
18	OK	0.031	12443648	154	90
19	OK	0.046	12333056	154	98
20	OK	0.046	12382208	154	93
21	OK	0.046	12472320	154	92
22	OK	0.046	12451840	154	98
23	OK	0.968	30310400	1000008	616458
24	OK	1.109	30343168	1000008	622272
25	OK	1.203	30380032	1000008	625335
26	OK	1.218	30334976	1000008	628546
27	OK	1.265	30392320	1000008	631472
28	OK	1.250	30310400	1000008	632217
29	OK	1.218	30355456	1000008	631772