Санкт-Петербургский Национальный Исследовательский
Университет Информационных Технологий, Механики и Оптики
Мегафакультет компьютерных технологий и управления

Дисциплина
«Алгоритмы и структуры данных»

Лабораторная работа №7

Выполнил:
Студент группы P3218
Рябов Сергей Витальевич
Преподаватель:
Муромцев Дмитрий Ильич

Санкт-Петербург,
2018

1. Проверка сбалансированности

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 Ki, Li, Ri, разделенных пробелами — ключа в i-ой вершине $(|Ki| \le 10^9)$, номера левого ребенка i-ой вершины ($i < Li \le N$ или Li=0, если левого ребенка нет) и номера правого ребенка i-ой вершины ($i < Ri \le N$ или Ri=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.

Исходный код (C#):

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Globalization;

namespace ADS.Week7
{

    public class BinaryTree
    {
        private struct Node
        {
            public int Key { get; set; }
            public int Left { get; set; }
            public int Right { get; set; }
            public int Height { get; set; }

            public Node(int key, int left, int right)
            {
                Key = key;
                Left = left;
                Right = right;
                Height = 0;
```

```
            }
        }
        private Node[] nodes;
        private int root;
        private int size;

        public BinaryTree(int size)
        {
            nodes = new Node[size];
            root = 0;
            size = 0;
        }
        public void Add(int key, int left, int right)
        {
            nodes[size] = new Node(key, left, right);
            if (nodes[size].Left == root || nodes[size].Right == root)
                root = size;
            size++;
        }

        public void GetHeights()
        {
            List<int> childs = new List<int>(nodes.Length);
            childs.Add(root);
            for (int i = 0; i < childs.Count; i++)
            {
                if (nodes[childs[i]].Left != -1)
                    childs.Add(nodes[childs[i]].Left);
                if (nodes[childs[i]].Right != -1)
                    childs.Add(nodes[childs[i]].Right);
            }
            for (int i = childs.Count - 1; i >= 0; i--)
            {
                if (nodes[childs[i]].Left == nodes[childs[i]].Right)
                {
                    nodes[childs[i]].Height = 1;
                }
                else
                {
                    if (nodes[childs[i]].Left != -1)
                        nodes[childs[i]].Height = nodes[nodes[childs[i]].Left].Height;
                    if (nodes[childs[i]].Right != -1)
                        nodes[childs[i]].Height = Math.Max(nodes[nodes[childs[i]].Right].Height,
nodes[childs[i]].Height);
                    nodes[childs[i]].Height++;
                }
            }
        }
        public int GetBalance(int index)
        {
            int left = nodes[index].Left == -1 ? 0 : nodes[nodes[index].Left].Height;
            int right = nodes[index].Right == -1 ? 0 : nodes[nodes[index].Right].Height;
            return right - left;
        }
    }
    public class Task1
    {
        public static void Main(string[] args)
        {
            using (StreamReader streamReader = new StreamReader("input.txt"))
            using (StreamWriter streamWriter = new StreamWriter("output.txt"))
            {
                int n = int.Parse(streamReader.ReadLine());
                BinaryTree tree = new BinaryTree(n);
```

```
                for (int i = 0; i < n; i++)
                {
                    string[] str = streamReader.ReadLine().Split(' ');
                    tree.Add(int.Parse(str[0]), int.Parse(str[1]) - 1, int.Parse(str[2]) - 1);
                }
                tree.GetHeights();
                for (int i = 0; i < n; i++)
                {
                    streamWriter.WriteLine(tree.GetBalance(i));
                }
            }
        }
    }
}
```

Результат:

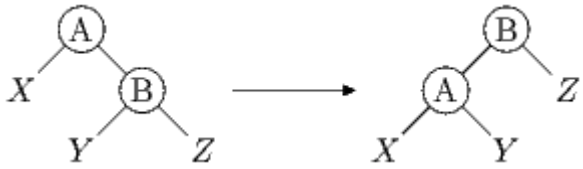
№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.265	15609856	3986010	1688889
1	OK	0.015	10645504	46	19
2	OK	0.031	10686464	10	3
3	OK	0.015	10641408	17	6
4	OK	0.031	10637312	17	7
5	OK	0.031	10633216	24	9
6	OK	0.031	10698752	24	10
7	OK	0.031	10612736	24	9
8	OK	0.031	10657792	24	10
9	OK	0.031	10661888	24	11
10	OK	0.046	10657792	31	12
11	OK	0.031	10641408	31	13
12	OK	0.031	10620928	31	12
13	OK	0.015	10653696	31	13
14	OK	0.031	10629120	31	14
15	OK	0.031	10637312	31	12
16	OK	0.031	10661888	31	13
17	OK	0.031	10649600	31	13
18	OK	0.031	10653696	31	14
19	OK	0.015	10633216	31	13
20	OK	0.046	10665984	31	14

2. Делаю я левый поворот...

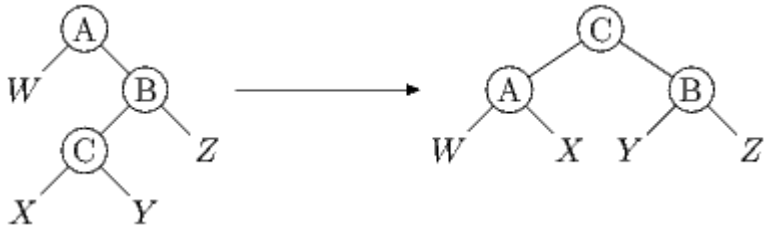
Для балансировки АВЛ-дерева при операциях вставки и удаления производятся левые и правыеповороты. Левый поворот в вершине производится, когда баланс этой вершины больше 1, аналогично, правый поворот производится при балансе, меньшем −1.

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 Ki, Li, Ri, разделенных пробелами — ключа в i-ой вершине ($|Ki| \le 10^9$), номера левого ребенка i-ой вершины ($i < Li \le N$ или Li=0, если левого ребенка нет) и номера правого ребенка i-ой вершины ($i < Ri \le N$ или Ri=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.

Исходный код (C#):

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Globalization;

namespace ADS.Week7
{
    public class BinaryTree
    {
        public struct Node
        {
            public int Key { get; set; }
            public int Left { get; set; }
            public int Right { get; set; }
            public int Height { get; set; }
            public int Parent { get; set; }
```

```
            public Node(int key, int left, int right)
            {
                Key = key;
                Left = left;
                Right = right;
                Height = 0;
                Parent = -1;
            }
        }
        private Node[] nodes;
        public int Root { get; set; }
        private int size;

        public Node this[int index]
        {
            get { return nodes[index]; }
        }

        public BinaryTree(int size)
        {
            nodes = new Node[size];
            Root = 0;
            size = 0;
        }
        public void Add(int key, int left, int right)
        {
            nodes[size] = new Node(key, left, right);
            if (nodes[size].Left == Root || nodes[size].Right == Root)
                Root = size;
            size++;
        }

        public void GetHeights()
        {
            List<int> childs = new List<int>(nodes.Length);
            childs.Add(Root);
            for (int i = 0; i < childs.Count; i++)
            {
                if (nodes[childs[i]].Left != -1)
                {
                    childs.Add(nodes[childs[i]].Left);
                    nodes[nodes[childs[i]].Left].Parent = childs[i];
                }
                if (nodes[childs[i]].Right != -1)
                {
                    childs.Add(nodes[childs[i]].Right);
                    nodes[nodes[childs[i]].Right].Parent = childs[i];
                }
            }
            for (int i = childs.Count - 1; i >= 0; i--)
            {
                if (nodes[childs[i]].Left == nodes[childs[i]].Right)
                {
                    nodes[childs[i]].Height = 1;
                }
                else
                {
                    if (nodes[childs[i]].Left != -1)
                        nodes[childs[i]].Height = nodes[nodes[childs[i]].Left].Height;
                    if (nodes[childs[i]].Right != -1)
                        nodes[childs[i]].Height = Math.Max(nodes[nodes[childs[i]].Right].Height,
nodes[childs[i]].Height);
                    nodes[childs[i]].Height++;
```

```
                }
            }
        }
        public int GetBalance(int index)
        {
            int left = nodes[index].Left == -1 ? 0 : nodes[nodes[index].Left].Height;
            int right = nodes[index].Right == -1 ? 0 : nodes[nodes[index].Right].Height;
            return right - left;
        }
        public void Balance()
        {
            if (GetBalance(nodes[Root].Right) == -1)
            {
                int A = Root;
                int B = nodes[A].Right;
                int C = nodes[B].Left;
                nodes[A].Right = nodes[C].Left;
                nodes[B].Left = nodes[C].Right;
                nodes[C].Left = A;
                nodes[C].Right = B;
                Root = C;
            }
            else
            {
                int A = Root;
                int B = nodes[A].Right;
                nodes[A].Right = nodes[B].Left;
                nodes[B].Left = A;
                Root = B;
            }
        }

        public void Print(StreamWriter streamWriter)
        {
            int[] indices = new int[size];
            int currentIndex = 0;
            GetIndices(indices, ref currentIndex, Root);
            Print(indices, Root, streamWriter);
        }

        private void GetIndices(int[] indices, ref int currentIndex, int index)
        {
            if (index == -1)
                return;

            Node node = nodes[index];
            indices[index] = currentIndex;
            currentIndex++;
            GetIndices(indices, ref currentIndex, node.Left);
            GetIndices(indices, ref currentIndex, node.Right);
        }
        private void Print(int[] indices, int index, StreamWriter streamWriter)
        {
            if (index == -1)
                return;

            Node node = nodes[index];
            streamWriter.WriteLine("{0} {1} {2}", node.Key, node.Left == -1 ? 0 : indices[node.Left] + 1,
node.Right == -1 ? 0 : indices[node.Right] + 1);

            Print(indices, node.Left, streamWriter);
            Print(indices, node.Right, streamWriter);
        }
    }
```

```
    public class Task2
    {
        public static void Main(string[] args)
        {
            using (StreamReader streamReader = new StreamReader("input.txt"))
            using (StreamWriter streamWriter = new StreamWriter("output.txt"))
            {
                int n = int.Parse(streamReader.ReadLine());
                BinaryTree tree = new BinaryTree(n);
                for (int i = 0; i < n; i++)
                {
                    string[] str = streamReader.ReadLine().Split(' ');
                    tree.Add(int.Parse(str[0]), int.Parse(str[1]) - 1, int.Parse(str[2]) - 1);
                }
                tree.GetHeights();
                tree.Balance();

                streamWriter.WriteLine(n);
                tree.Print(streamWriter);
            }
        }
    }
}
```

Результат:

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.359	17289216	3986416	3986416
1	OK	0.031	10915840	54	54
2	OK	0.046	10838016	24	24
3	OK	0.031	10866688	24	24
4	OK	0.031	10878976	31	31
5	OK	0.062	10833920	45	45
6	OK	0.031	10850304	45	45
7	OK	0.031	10883072	45	45
8	OK	0.031	10858496	45	45
9	OK	0.031	10858496	52	52
10	OK	0.031	10838016	52	52
11	OK	0.078	10829824	52	52
12	OK	0.031	10887168	52	52
13	OK	0.031	10813440	52	52
14	OK	0.031	10854400	52	52
15	OK	0.015	10825728	59	59
16	OK	0.031	10911744	59	59
17	OK	0.046	10899456	59	59
18	OK	0.015	10878976	59	59
19	OK	0.031	10891264	66	66
20	OK	0.031	10821632	75	75

3. Вставка в АВЛ-дерево

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 Ki, Li, Ri, разделенных пробелами — ключа в i-ой вершине ($|Ki| \le 10^9$), номера левого ребенка i-ой вершины ($i < Li \le N$ или Li=0, если левого ребенка нет) и номера правого ребенка i-ой вершины ($i < Ri \le N$ или Ri=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.

Исходный код (C#):

```
using System;
using System.IO;
using System.Collections.Generic;
```

```
using System.Globalization;

namespace ADS.Week7
{

    public class BinaryTree
    {
        public struct Node
        {
            public int Key { get; set; }
            public int Left { get; set; }
            public int Right { get; set; }
            public int Height { get; set; }
            public int Parent { get; set; }

            public Node(int key, int left, int right)
            {
                Key = key;
                Left = left;
                Right = right;
                Height = 0;
                Parent = -1;
            }
        }
        private Node[] nodes;
        public int Root { get; set; }
        private int size;

        public Node this[int index]
        {
            get { return nodes[index]; }
        }

        public BinaryTree(int size)
        {
            nodes = new Node[size + 1];
            Root = 0;
        }
        public void Add(int key, int left, int right)
        {
            nodes[size] = new Node(key, left, right);
            if (nodes[size].Left == Root || nodes[size].Right == Root)
                Root = size;
            size++;
        }

        public void GetHeights()
        {
            List<int> childs = new List<int>(nodes.Length);
            childs.Add(Root);
            for (int i = 0; i < childs.Count; i++)
            {
                if (nodes[childs[i]].Left != -1)
                {
                    childs.Add(nodes[childs[i]].Left);
                    nodes[nodes[childs[i]].Left].Parent = childs[i];
                }
                if (nodes[childs[i]].Right != -1)
                {
                    childs.Add(nodes[childs[i]].Right);
                    nodes[nodes[childs[i]].Right].Parent = childs[i];
                }
            }
            for (int i = childs.Count - 1; i >= 0; i--)
```

```
            {
                if (nodes[childs[i]].Left == nodes[childs[i]].Right)
                {
                    nodes[childs[i]].Height = 1;
                }
                else
                {
                    if (nodes[childs[i]].Left != -1)
                        nodes[childs[i]].Height = nodes[nodes[childs[i]].Left].Height;
                    if (nodes[childs[i]].Right != -1)
                        nodes[childs[i]].Height = Math.Max(nodes[nodes[childs[i]].Right].Height,
nodes[childs[i]].Height);
                    nodes[childs[i]].Height++;
                }
            }
        }
        public int GetBalance(int index)
        {
            int left = nodes[index].Left == -1 ? 0 : nodes[nodes[index].Left].Height;
            int right = nodes[index].Right == -1 ? 0 : nodes[nodes[index].Right].Height;
            return right - left;
        }
        public void Balance(int node)
        {
            int balance = GetBalance(node);
            if (balance == 2)
                LeftBalance(node);
            else if (balance == -2)
                RightBalance(node);
        }
        private void RightBalance(int node)
        {
            if (GetBalance(nodes[node].Left) == 1)
            {
                int A = node;
                int B = nodes[A].Left;
                int C = nodes[B].Right;
                nodes[B].Right = nodes[C].Left;
                nodes[A].Left = nodes[C].Right;
                nodes[C].Left = B;
                nodes[C].Right = A;
                if (node == Root)
                    Root = C;
                else
                {
                    if (nodes[nodes[node].Parent].Left == node)
                        nodes[nodes[node].Parent].Left = C;
                    else
                        nodes[nodes[node].Parent].Right = C;
                }
            }
            else
            {
                int A = node;
                int B = nodes[A].Left;
                nodes[A].Left = nodes[B].Right;
                nodes[B].Right = A;
                if (node == Root)
                    Root = B;
                else
                {
                    if (nodes[nodes[node].Parent].Left == node)
                        nodes[nodes[node].Parent].Left = B;
                    else
```

```
                nodes[nodes[node].Parent].Right = B;
        }
    }
}
private void LeftBalance(int node)
{
    if (GetBalance(nodes[node].Right) == -1)
    {
        int A = node;
        int B = nodes[A].Right;
        int C = nodes[B].Left;
        nodes[A].Right = nodes[C].Left;
        nodes[B].Left = nodes[C].Right;
        nodes[C].Left = A;
        nodes[C].Right = B;
        if (node == Root)
            Root = C;
        else
        {
            if (nodes[nodes[node].Parent].Left == node)
                nodes[nodes[node].Parent].Left = C;
            else
                nodes[nodes[node].Parent].Right = C;
        }
    }
    else
    {
        int A = node;
        int B = nodes[A].Right;
        nodes[A].Right = nodes[B].Left;
        nodes[B].Left = A;
        if (node == Root)
            Root = B;
        else
        {
            if (nodes[nodes[node].Parent].Left == node)
                nodes[nodes[node].Parent].Left = B;
            else
                nodes[nodes[node].Parent].Right = B;
        }
    }
}
public void Insert(int key)
{
    if (size == 0)
    {
        Add(key, -1, -1);
        return;
    }
    int index = Insert(key, Root);
    GetHeights();
    Balance(index);
    while (nodes[index].Parent != -1)
    {
        index = nodes[index].Parent;
        Balance(index);
    }
}
private int Insert(int key, int node)
{
    if (key < nodes[node].Key)
    {
        if (nodes[node].Left == -1)
        {
```

```
                    nodes[node].Left = size;
                    Add(key, -1, -1);
                    nodes[size - 1].Parent = node;
                    return node;
                }
                else
                    return Insert(key, nodes[node].Left);
            }
            else
            {
                if (nodes[node].Right == -1)
                {
                    nodes[node].Right = size;
                    Add(key, -1, -1);
                    nodes[size - 1].Parent = node;
                    return node;
                }
                else
                    return Insert(key, nodes[node].Right);
            }
        }

        public void Print(StreamWriter streamWriter)
        {
            int[] indices = new int[size];
            int currentIndex = 0;
            GetIndices(indices, ref currentIndex, Root);
            Print(indices, Root, streamWriter);
        }

        private void GetIndices(int[] indices, ref int currentIndex, int index)
        {
            if (index == -1)
                return;

            Node node = nodes[index];
            indices[index] = currentIndex;
            currentIndex++;
            GetIndices(indices, ref currentIndex, node.Left);
            GetIndices(indices, ref currentIndex, node.Right);
        }
        private void Print(int[] indices, int index, StreamWriter streamWriter)
        {
            if (index == -1)
                return;

            Node node = nodes[index];
            streamWriter.WriteLine("{0} {1} {2}", node.Key, node.Left == -1 ? 0 : indices[node.Left] + 1,
node.Right == -1 ? 0 : indices[node.Right] + 1);

            Print(indices, node.Left, streamWriter);
            Print(indices, node.Right, streamWriter);
        }
    }
    public class Task3
    {
        public static void Main(string[] args)
        {
            using (StreamReader streamReader = new StreamReader("input.txt"))
            using (StreamWriter streamWriter = new StreamWriter("output.txt"))
            {
                int n = int.Parse(streamReader.ReadLine());
                BinaryTree tree = new BinaryTree(n);
                for (int i = 0; i < n; i++)
```

```
                {
                    string[] str = streamReader.ReadLine().Split(' ');
                    tree.Add(int.Parse(str[0]), int.Parse(str[1]) - 1, int.Parse(str[2]) - 1);
                }
                tree.Insert(int.Parse(streamReader.ReadLine()));
                streamWriter.WriteLine(n + 1);
                tree.Print(streamWriter);
            }
        }
    }
}
```

Результат:

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.546	50696192	4011957	4011966
1	OK	0.046	10948608	20	24
2	OK	0.031	10956800	6	10
3	OK	0.031	10973184	14	18
4	OK	0.031	10985472	13	17
5	OK	0.046	10981376	21	25
6	OK	0.031	10981376	20	24
7	OK	0.031	10993664	20	24
8	OK	0.031	11079680	21	25
9	OK	0.031	11042816	20	24
10	OK	0.015	10969088	20	24
11	OK	0.031	10969088	28	32
12	OK	0.031	10960896	27	31
13	OK	0.031	10928128	27	31
14	OK	0.031	10903552	27	31
15	OK	0.015	10989568	35	39
16	OK	0.031	10952704	34	38
17	OK	0.015	10928128	34	38
18	OK	0.031	10932224	34	38
19	OK	0.031	10924032	34	38
20	OK	0.031	11001856	35	39

4. Удаление из АВЛ-дерева

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 Ki, Li, Ri, разделенных пробелами — ключа в i-ой вершине ($|Ki| \le 109$), номера левого ребенка i-ой вершины ($i < Li \le N$ или Li=0, если левого ребенка нет) и номера правого ребенка i-ой вершины ($i < Ri \le N$ или Ri=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.

Исходный код (C#):

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Globalization;

namespace ADS.Week7
{

    public class BinaryTree
    {
        public struct Node
        {
            public int Key { get; set; }
            public int Left { get; set; }
            public int Right { get; set; }
            public int Height { get; set; }
            public int Parent { get; set; }
```

```
            public Node(int key, int left, int right)
            {
                Key = key;
                Left = left;
                Right = right;
                Height = 0;
                Parent = -1;
            }
        }
        private Node[] nodes;
        public int Root { get; set; }
        private int size;

        public Node this[int index]
        {
            get { return nodes[index]; }
        }

        public BinaryTree(int size)
        {
            nodes = new Node[size];
            Root = 0;
        }
        public void Add(int key, int left, int right)
        {
            nodes[size] = new Node(key, left, right);
            if (nodes[size].Left == Root || nodes[size].Right == Root)
                Root = size;
            size++;
        }

        public void GetHeights()
        {
            List<int> childs = new List<int>(nodes.Length);
            childs.Add(Root);
            for (int i = 0; i < childs.Count; i++)
            {
                if (nodes[childs[i]].Left != -1)
                {
                    childs.Add(nodes[childs[i]].Left);
                    nodes[nodes[childs[i]].Left].Parent = childs[i];
                }
                if (nodes[childs[i]].Right != -1)
                {
                    childs.Add(nodes[childs[i]].Right);
                    nodes[nodes[childs[i]].Right].Parent = childs[i];
                }
            }
            for (int i = childs.Count - 1; i >= 0; i--)
            {
                if (nodes[childs[i]].Left == nodes[childs[i]].Right)
                {
                    nodes[childs[i]].Height = 1;
                }
                else
                {
                    if (nodes[childs[i]].Left != -1)
                        nodes[childs[i]].Height = nodes[nodes[childs[i]].Left].Height;
                    if (nodes[childs[i]].Right != -1)
                        nodes[childs[i]].Height = Math.Max(nodes[nodes[childs[i]].Right].Height,
nodes[childs[i]].Height);
                    nodes[childs[i]].Height++;
                }
```

```
    }
}
public int GetBalance(int index)
{
    int left = nodes[index].Left == -1 ? 0 : nodes[nodes[index].Left].Height;
    int right = nodes[index].Right == -1 ? 0 : nodes[nodes[index].Right].Height;
    return right - left;
}
public void Balance(int node)
{
    int balance = GetBalance(node);
    if (balance == 2)
        LeftBalance(node);
    else if (balance == -2)
        RightBalance(node);
}
private void RightBalance(int node)
{
    if (GetBalance(nodes[node].Left) == 1)
    {
        int A = node;
        int B = nodes[A].Left;
        int C = nodes[B].Right;
        nodes[B].Right = nodes[C].Left;
        nodes[A].Left = nodes[C].Right;
        nodes[C].Left = B;
        nodes[C].Right = A;
        if (node == Root)
            Root = C;
        else
        {
            if (nodes[nodes[node].Parent].Left == node)
                nodes[nodes[node].Parent].Left = C;
            else
                nodes[nodes[node].Parent].Right = C;
        }
    }
    else
    {
        int A = node;
        int B = nodes[A].Left;
        nodes[A].Left = nodes[B].Right;
        nodes[B].Right = A;
        if (node == Root)
            Root = B;
        else
        {
            if (nodes[nodes[node].Parent].Left == node)
                nodes[nodes[node].Parent].Left = B;
            else
                nodes[nodes[node].Parent].Right = B;
        }
    }
}
private void LeftBalance(int node)
{
    if (GetBalance(nodes[node].Right) == -1)
    {
        int A = node;
        int B = nodes[A].Right;
        int C = nodes[B].Left;
        nodes[A].Right = nodes[C].Left;
        nodes[B].Left = nodes[C].Right;
        nodes[C].Left = A;
```

```
            nodes[C].Right = B;
            if (node == Root)
                Root = C;
            else
            {
                if (nodes[nodes[node].Parent].Left == node)
                    nodes[nodes[node].Parent].Left = C;
                else
                    nodes[nodes[node].Parent].Right = C;
            }
        }
        else
        {
            int A = node;
            int B = nodes[A].Right;
            nodes[A].Right = nodes[B].Left;
            nodes[B].Left = A;
            if (node == Root)
                Root = B;
            else
            {
                if (nodes[nodes[node].Parent].Left == node)
                    nodes[nodes[node].Parent].Left = B;
                else
                    nodes[nodes[node].Parent].Right = B;
            }
        }
    }
    public void Delete(int key)
    {
        if (size == 1)
        {
            size = 0;
            Root = -1;
            return;
        }
        GetHeights();
        int node = Find(key, Root);
        if (nodes[node].Left == -1)
        {
            if (nodes[node].Right == -1)
            {
                if (nodes[nodes[node].Parent].Left == node)
                    nodes[nodes[node].Parent].Left = -1;
                else
                    nodes[nodes[node].Parent].Right = -1;
                BalanceFrom(nodes[node].Parent);
            }
            else
            {
                int parent = nodes[node].Parent;
                nodes[node] = nodes[nodes[node].Right];
                nodes[node].Parent = parent;
                BalanceFrom(node);
            }
        }
        else
        {
            int max = FindMax(nodes[node].Left);
            nodes[node].Key = nodes[max].Key;

            int newChild = -1;
            if (nodes[max].Left != -1)
            {
```

```
                newChild = nodes[max].Left;
            }
            if (nodes[nodes[max].Parent].Right == max)
                nodes[nodes[max].Parent].Right = newChild;
            else
                nodes[nodes[max].Parent].Left = newChild;
            BalanceFrom(nodes[max].Parent);
        }
    }
    private void BalanceFrom(int index)
    {
        Balance(index);
        while (nodes[index].Parent != -1)
        {
            index = nodes[index].Parent;
            Balance(index);
        }
    }
    private int Find(int key, int node)
    {
        if (key < nodes[node].Key)
            return Find(key, nodes[node].Left);
        else if (key > nodes[node].Key)
            return Find(key, nodes[node].Right);
        else
            return node;
    }
    private int FindMax(int node)
    {
        if (nodes[node].Right == -1)
            return node;
        else
            return FindMax(nodes[node].Right);
    }

    public void Print(StreamWriter streamWriter)
    {
        int[] indices = new int[size];
        int currentIndex = 0;
        GetIndices(indices, ref currentIndex, Root);
        Print(indices, Root, streamWriter);
    }

    private void GetIndices(int[] indices, ref int currentIndex, int index)
    {
        if (index == -1)
            return;

        Node node = nodes[index];
        indices[index] = currentIndex;
        currentIndex++;
        GetIndices(indices, ref currentIndex, node.Left);
        GetIndices(indices, ref currentIndex, node.Right);
    }
    private void Print(int[] indices, int index, StreamWriter streamWriter)
    {
        if (index == -1)
            return;

        Node node = nodes[index];
        streamWriter.WriteLine("{0} {1} {2}", node.Key, node.Left == -1 ? 0 : indices[node.Left] + 1,
node.Right == -1 ? 0 : indices[node.Right] + 1);

        Print(indices, node.Left, streamWriter);
```

```
            Print(indices, node.Right, streamWriter);
        }
    }
    public class Task4
    {
        public static void Main(string[] args)
        {
            using (StreamReader streamReader = new StreamReader("input.txt"))
            using (StreamWriter streamWriter = new StreamWriter("output.txt"))
            {
                int n = int.Parse(streamReader.ReadLine());
                BinaryTree tree = new BinaryTree(n);
                for (int i = 0; i < n; i++)
                {
                    string[] str = streamReader.ReadLine().Split(' ');
                    tree.Add(int.Parse(str[0]), int.Parse(str[1]) - 1, int.Parse(str[2]) - 1);
                }
                tree.Delete(int.Parse(streamReader.ReadLine()));
                streamWriter.WriteLine(n - 1);
                tree.Print(streamWriter);
            }
        }
    }
}
```

Результат:

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.546	50782208	4077288	4077255
1	OK	0.031	10964992	27	17
2	OK	0.031	10874880	13	3
3	OK	0.031	10960896	20	10
4	OK	0.031	10969088	20	10
5	OK	0.031	10981376	20	10
6	OK	0.062	10952704	20	10
7	OK	0.031	10952704	27	17
8	OK	0.031	10981376	27	17
9	OK	0.031	10948608	27	17
10	OK	0.046	10977280	34	24
11	OK	0.031	10932224	34	24
12	OK	0.031	10940416	34	24
13	OK	0.031	10960896	34	24
14	OK	0.031	10936320	34	24
15	OK	0.031	10936320	34	24
16	OK	0.031	10981376	34	24
17	OK	0.031	10928128	34	24
18	OK	0.031	10936320	34	24
19	OK	0.031	10928128	34	24
20	OK	0.031	10956800	34	24

5. Упорядоченное множество на АВЛ-дереве

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.

Исходный код (C#):

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Globalization;

namespace ADS.Week7
{

    public class BinaryTree
    {
        public struct Node
        {
            public int Key { get; set; }
            public int Left { get; set; }
            public int Right { get; set; }
            public int Height { get; set; }
            public int Parent { get; set; }

            public Node(int key, int left, int right)
            {
```

```
        Key = key;
        Left = left;
        Right = right;
        Height = 1;
        Parent = -1;
    }
}
private Node[] nodes;
public int Root { get; set; }
private int size;

public Node this[int index]
{
    get { return nodes[index]; }
}

public BinaryTree(int size)
{
    nodes = new Node[size];
    Root = 0;
}
public void Add(int key, int left, int right)
{
    nodes[size] = new Node(key, left, right);
    if (nodes[size].Left == Root || nodes[size].Right == Root)
        Root = size;
    size++;
}

public int GetBalance(int index)
{
    int left = nodes[index].Left == -1 ? 0 : nodes[nodes[index].Left].Height;
    int right = nodes[index].Right == -1 ? 0 : nodes[nodes[index].Right].Height;
    return right - left;
}
public void Balance(int node)
{
    int balance = GetBalance(node);
    if (balance == 2)
        LeftBalance(node);
    else if (balance == -2)
        RightBalance(node);
}
private void RightBalance(int node)
{
    if (GetBalance(nodes[node].Left) == 1)
    {
        int A = node;
        int B = nodes[A].Left;
        int C = nodes[B].Right;
        nodes[B].Right = nodes[C].Left;
        nodes[A].Left = nodes[C].Right;
        nodes[C].Left = B;
        nodes[C].Right = A;

        nodes[C].Parent = nodes[A].Parent;
        nodes[A].Parent = C;
        nodes[B].Parent = C;
        if (node == Root)
            Root = C;
        else
        {
            if (nodes[nodes[node].Parent].Left == node)
                nodes[nodes[node].Parent].Left = C;
```

```
            else
                nodes[nodes[node].Parent].Right = C;
        }
        RecalculateHeights(B);
        RecalculateHeights(C);
    }
    else
    {
        int A = node;
        int B = nodes[A].Left;
        nodes[A].Left = nodes[B].Right;
        nodes[B].Right = A;

        nodes[B].Parent = nodes[A].Parent;
        nodes[A].Parent = B;
        if (node == Root)
            Root = B;
        else
        {
            if (nodes[nodes[node].Parent].Left == node)
                nodes[nodes[node].Parent].Left = B;
            else
                nodes[nodes[node].Parent].Right = B;
        }
        RecalculateHeights(A);
    }
}
private void LeftBalance(int node)
{
    if (GetBalance(nodes[node].Right) == -1)
    {
        int A = node;
        int B = nodes[A].Right;
        int C = nodes[B].Left;
        nodes[A].Right = nodes[C].Left;
        nodes[B].Left = nodes[C].Right;
        nodes[C].Left = A;
        nodes[C].Right = B;
        nodes[C].Parent = nodes[A].Parent;
        nodes[A].Parent = C;
        nodes[B].Parent = C;
        if (node == Root)
            Root = C;
        else
        {
            if (nodes[nodes[node].Parent].Left == node)
                nodes[nodes[node].Parent].Left = C;
            else
                nodes[nodes[node].Parent].Right = C;
        }
        RecalculateHeights(B);
        RecalculateHeights(C);
    }
    else
    {
        int A = node;
        int B = nodes[A].Right;
        nodes[A].Right = nodes[B].Left;
        nodes[B].Left = A;
        nodes[B].Parent = nodes[A].Parent;
        nodes[A].Parent = B;
        if (node == Root)
            Root = B;
        else
```

```
        {
            if (nodes[nodes[node].Parent].Left == node)
                nodes[nodes[node].Parent].Left = B;
            else
                nodes[nodes[node].Parent].Right = B;
        }
        RecalculateHeights(A);
    }
}
private void RecalculateHeights(int index)
{
    int height = 1;
    if (nodes[index].Left != -1)
        height = nodes[nodes[index].Left].Height + 1;
    if (nodes[index].Right != -1)
        height = Math.Max(nodes[nodes[index].Right].Height, height) + 1;
    nodes[index].Height = height;
}
public void Insert(int key)
{
    if (size == 0)
    {
        Add(key, -1, -1);
        return;
    }
    int index = Insert(key, Root);
    if (index == -1)
        return;
    if (nodes[index].Left == -1 || nodes[index].Right == -1)
    {
        RecalculateHeights(index);
    }
    Balance(index);
    while (nodes[index].Parent != -1)
    {
        index = nodes[index].Parent;
        Balance(index);
    }
}
private int Insert(int key, int node)
{
    if (key == nodes[node].Key)
        return -1;
    if (key < nodes[node].Key)
    {
        if (nodes[node].Left == -1)
        {
            nodes[node].Left = size;
            Add(key, -1, -1);
            nodes[size - 1].Parent = node;
            return node;
        }
        else
            return Insert(key, nodes[node].Left);
    }
    else
    {
        if (nodes[node].Right == -1)
        {
            nodes[node].Right = size;
            Add(key, -1, -1);
            nodes[size - 1].Parent = node;
            return node;
        }
```

```
        else
            return Insert(key, nodes[node].Right);
    }
}

public void Delete(int key)
{
    if (size == 1)
    {
        size = 0;
        Root = -1;
        return;
    }
    int node = Find(key, Root);
    if (node == -1)
        return;
    if (nodes[node].Left == -1)
    {
        if (nodes[node].Right == -1)
        {
            if (nodes[nodes[node].Parent].Left == node)
                nodes[nodes[node].Parent].Left = -1;
            else
                nodes[nodes[node].Parent].Right = -1;
            BalanceFrom(nodes[node].Parent);
        }
        else
        {
            int parent = nodes[node].Parent;
            nodes[node] = nodes[nodes[node].Right];
            nodes[node].Parent = parent;
            BalanceFrom(node);
        }
    }
    else
    {
        int max = FindMax(nodes[node].Left);
        nodes[node].Key = nodes[max].Key;

        int newChild = -1;
        if (nodes[max].Left != -1)
        {
            newChild = nodes[max].Left;
        }
        if (nodes[nodes[max].Parent].Right == max)
            nodes[nodes[max].Parent].Right = newChild;
        else
            nodes[nodes[max].Parent].Left = newChild;
        BalanceFrom(nodes[max].Parent);
    }
}
private void BalanceFrom(int index)
{
    Balance(index);
    while (nodes[index].Parent != -1)
    {
        index = nodes[index].Parent;
        Balance(index);
    }
}
public int Find(int key, int node)
{
    if (node == -1)
        return node;
```

```
            if (key < nodes[node].Key)
                return Find(key, nodes[node].Left);
            else if (key > nodes[node].Key)
                return Find(key, nodes[node].Right);
            else
                return node;
        }
        private int FindMax(int node)
        {
            if (nodes[node].Right == -1)
                return node;
            else
                return FindMax(nodes[node].Right);
        }
    }
    public class Task5
    {
        public static void Main(string[] args)
        {
            using (StreamReader streamReader = new StreamReader("input.txt"))
            using (StreamWriter streamWriter = new StreamWriter("output.txt"))
            {
                int n = int.Parse(streamReader.ReadLine());
                BinaryTree tree = new BinaryTree(n);
                for (int i = 0; i < n; i++)
                {
                    string[] str = streamReader.ReadLine().Split(' ');
                    switch(str[0][0])
                    {
                        case 'A':
                            tree.Insert(int.Parse(str[1]));
                            streamWriter.WriteLine(tree.GetBalance(tree.Root));
                            break;
                        case 'C':
                            streamWriter.WriteLine(tree.Find(int.Parse(str[1]), tree.Root) == -1 ? 'N' :
'Y');
                            break;
                        case 'D':
                            tree.Delete(int.Parse(str[1]));
                            streamWriter.WriteLine(tree.GetBalance(tree.Root));
                            break;
                    }
                }

            }
        }
    }
}
```

Результат:

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.687	46579712	2678110	731071
1	OK	0.031	10387456	33	19
2	OK	0.031	10432512	114	66
3	OK	0.031	10321920	154	90
4	OK	0.046	10346496	154	91
5	OK	0.031	10330112	154	90
6	OK	0.031	10383360	154	95
7	OK	0.031	10342400	154	91
8	OK	0.031	10407936	154	94
9	OK	0.046	10444800	154	95
10	OK	0.031	10362880	154	90
11	OK	0.031	10391552	154	90
12	OK	0.031	10342400	154	90
13	OK	0.031	10383360	154	95
14	OK	0.046	10444800	154	97
15	OK	0.031	10371072	154	94
16	OK	0.015	10358784	154	93
17	OK	0.031	10428416	154	90
18	OK	0.031	10465280	154	90
19	OK	0.078	10391552	154	98
20	OK	0.046	10334208	154	93