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 №5

Неделя пятая

Выполнил:

Жумиков Егор Олегович

Преподаватели:

Романов Алексей Андреевич

Волчек Дмитрий Геннадьевич

Санкт-Петербург, 2019

Оглавление

Задача «Проверка сбалансированности»

Условие

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

Формат выходного файла

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

Решение

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

using namespace std;

#ifdef LOCAL

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

#else

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

#endif

#define nl "\n"

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

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

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

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

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


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

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;
}
```

Результат

Верное решение!
Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.468	53780480	3986010	1688889
1	OK	0.031	10604544	46	19
2	OK	0.031	10596352	10	3

Задача «Делаю я левый поворот...»

Условие

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

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.

Решение

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

using namespace std;

#ifdef LOCAL

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

#else

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

#endif

#define nl "\n"

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

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

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

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

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


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

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

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

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

        return i;
}

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

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

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

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

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

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

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

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

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

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

int *indexes;
int curr_index = 1;

void calc_index(int i) {
```

```
        if (!i) { return; }

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

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

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

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

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


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

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

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

                h[i] = 0;
        }

        cnt(0);
        left_rotation(0);

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


        return 0;
}
```

Результат

Верное решение!
Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.593	72486912	3986416	3786418
1	OK	0.031	11137024	54	49

Задача «Вставка в АВЛ-дерево»

Условие

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.

Решение

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

```
namespace ItmoAlgos
{
    public struct ModificationResult
    {
        public bool Modified { get; set; }
        public int HeightDelta { get; set; }
    }

    public sealed class Node<T> where T : IComparable<T>,
IEquatable<T>
    {
        public Node(T v, Node<T> left = null, Node<T> right = null,
bool refreshProperties = true)
        {
            if (v == null) throw new ArgumentNullException(nameof(v));

            Value = v;
            SetLeftChild(left, refreshProperties);
            SetRightChild(right, refreshProperties);
        }

        public T Value { get; }
        public Node<T> Left { get; private set; }
        public Node<T> Right { get; private set; }
        public Node<T> Parent { get; private set; }

        public int Height { get; private set; }

        public int Balance { get; private set; }

        public bool HasChildren => Left != null || Right != null;

        public void SetChild(Node<T> child, bool refreshProperties =
true)
        {
            if (child is null) throw new
InvalidOperationException("Use explicit methods for null values.");

            int comp = Value.CompareTo(child.Value);
            if (comp < 0)
                SetRightChild(child, refreshProperties);
            else if (comp > 0)
                SetLeftChild(child, refreshProperties);
            else
                throw new InvalidOperationException("Cannot set child
with the same value");
        }

        public void RemoveChild(Node<T> child, bool refreshProperties
= true)
        {
```

```
            if (child is null) throw new
InvalidOperationException("Use explicit methods for null values.");

            int comp = Value.CompareTo(child.Value);
            if (comp < 0)
                SetRightChild(null, refreshProperties);
            else if (comp > 0)
                SetLeftChild(null, refreshProperties);
            else
                throw new InvalidOperationException("Cannot set child
with the same value");
        }

        public void SetLeftChild(Node<T> left, bool refreshProperties
= true)
        {
            if (left ≠ null)
            {
                if (left.Value.CompareTo(Value) ⩾ 0) throw new
InvalidOperationException();

                left.Parent?.RemoveChild(left);
                left.Parent = this;
            }

            if (Left ≠ null) Left.Parent = null;

            Left = left;
            if (refreshProperties) RefreshOwnAndParentProperties();
        }

        public void SetRightChild(Node<T> right, bool
refreshProperties = true)
        {
            if (right ≠ null)
            {
                if (right.Value.CompareTo(Value) ⩽ 0) throw new
InvalidOperationException();

                right.Parent?.RemoveChild(right);
                right.Parent = this;
            }

            if (Right ≠ null) Right.Parent = null;

            Right = right;

            if (refreshProperties) RefreshOwnAndParentProperties();
        }

        public Node<T> LeftTurn()
```

```
{
    if (Right?.Balance < 0) return BigLeftTurn();

    Node<T>
        a = this,
        b = Right,
        y = b.Left;

    if (a.Parent ≠ null)
    {
        a.Parent.SetChild(b);
    }
    else
    {
        b.Parent = a.Parent;
    }

    a.SetRightChild(y);
    b.SetLeftChild(a);

    return b;
}

public Node<T> RightTurn()
{
    if (Left?.Balance > 0) return BigRightTurn();

    Node<T>
        a = this,
        b = a.Left,
        y = b.Right;

    if (a.Parent ≠ null)
    {
        a.Parent.SetChild(b);
    }
    else
    {
        b.Parent = a.Parent;
    }

    a.SetLeftChild(y);
    b.SetRightChild(a);

    return b;
}

public override string ToString()
{
    return $"Node({Value}, ({Left}, {Right})";
}
```

```
public string ToPrintableString()
{
    var sb = new StringBuilder();
    var queue = new Queue<Node<T>>();
    long counter = 1;
    queue.Enqueue(this);

    while (queue.Count > 0)
    {
        Node<T> current = queue.Dequeue();
        sb.Append(current.Value);

        if (current.Left ≠ null)
        {
            queue.Enqueue(current.Left);
            sb.Append($" {++counter}");
        }
        else
        {
            sb.Append(" 0");
        }


        if (current.Right ≠ null)
        {
            queue.Enqueue(current.Right);
            sb.Append($" {++counter}");
        }
        else
        {
            sb.Append(" 0");
        }

        sb.Append("\n");
    }

    return sb.ToString();
}

public Node<T> Insert(T x)
{
    Node<T> current = this;
    var inserted = false;

    while (!inserted)
    {
        int comp = current.Value.CompareTo(x);
        if (comp == 0)
        {
            throw new DuplicateNameException();
```

```
            }

            if (comp < 0)
            {
                if (current.Right ≠ null)
                {
                    current = current.Right;
                }
                else
                {
                    current.SetRightChild(new Node<T>(x));
                    inserted = true;
                }
            }
            else
            {
                if (current.Left ≠ null)
                {
                    current = current.Left;
                }
                else
                {
                    current.SetLeftChild(new Node<T>(x));
                    inserted = true;
                }
            }
        }

        Node<T> res = current;

        while (current ≠ null)
        {
            if (current.Balance > 1)
            {
                current = current.LeftTurn();
            }
            else if (current.Balance < -1)
            {
                current = current.RightTurn();
            }
            else
            {
                res = current;
                current = current.Parent;
            }
        }

        return res;
    }

    public void RefreshTree()
```

```
        {
            Left?.RefreshTree();
            Right?.RefreshTree();

            RefreshOwnProperties();
        }

        public void RefreshOwnProperties()
        {
            Height = 1 + Math.Max(Left?.Height ?? 0, Right?.Height ??
0);
            Balance = (Right?.Height ?? 0) - (Left?.Height ?? 0);
        }

        private void RefreshOwnAndParentProperties()
        {
            Node<T> node = this;

            while (node ≠ null)
            {
                node.RefreshOwnProperties();
                node = node.Parent;
            }
        }

        private Node<T> BigLeftTurn()
        {
            Node<T>
                a = this,
                b = a.Right,
                c = b.Left,
                x = c.Left,
                y = c.Right;

            // replace a with c
            if (a.Parent ≠ null)
            {
                a.Parent.SetChild(c);
            }
            else
            {
                c.Parent = a.Parent;
            }


            b.SetLeftChild(y);
            a.SetRightChild(x);

            c.SetLeftChild(a);
            c.SetRightChild(b);
```

```
        return c;
    }

    private Node<T> BigRightTurn()
    {
        Node<T>
            a = this,
            b = a.Left,
            c = b.Right,
            x = c.Left,
            y = c.Right;

        // replace a with c
        if (a.Parent ≠ null)
        {
            a.Parent?.SetChild(c);
        }
        else
        {
            c.Parent = a.Parent;
        }

        b.SetRightChild(x);
        a.SetLeftChild(y);

        c.SetLeftChild(b);
        c.SetRightChild(a);

        return c;
    }
}

public class Program
{
    private static string[] _input;
    private static int _currentLineIndex;

    private static string ReadLine()
    {
        return _input[_currentLineIndex++];
    }

    public static void Main(string[] args)
    {
        _input = File.ReadAllLines("input.txt");

        int n = int.Parse(ReadLine());
        var parents = new int[n];
        var nodes = new List<Node<int>>();
```

```
            for (var i = 0; i < n; i++)
            {
                string[] nodeParts = ReadLine().Split();

                // setting parents to child nodes
                int leftChild = int.Parse(nodeParts[1]) - 1,
                    rightChild = int.Parse(nodeParts[2]) - 1;

                if (leftChild ⩾ 0)
                    parents[leftChild] = nodes.Count;
                if (rightChild ⩾ 0)
                    parents[rightChild] = nodes.Count;


                nodes.Add(new Node<int>(int.Parse(nodeParts[0]),
refreshProperties: false));

                // checking for parents for non-root nodes
                if (i > 0)
nodes[parents[i]].SetChild(nodes[nodes.Count - 1], false);
            }

            for (int i = n - 1; i ⩾ 0; i--)
nodes[i].RefreshOwnProperties();

            using (var sw = new StreamWriter("output.txt"))
            {
                Node<int> root;
                if (nodes.Count > 0)
                {
                    root = nodes[0];
                    root = root.Insert(int.Parse(ReadLine()));
                }
                else
                {
                    root = new Node<int>(int.Parse(ReadLine()));
                }

                sw.WriteLine(n + 1);
                sw.WriteLine(root.ToPrintableString());
            }
        }
    }
}
```

Результат

Верное решение!
Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.593	74256384	4011957	3811967
1	OK	0.031	12406784	20	23
2	OK	0.031	11104256	6	11
3	OK	0.046	12406784	14	18

Задача «Удаление из АВЛ-дерева»

Условия

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 выходного файла

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

Решение

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

```
using System.Text;

namespace ItmoAlgos
{
    public struct ModificationResult
    {
        public bool Modified { get; set; }
        public int HeightDelta { get; set; }
    }

    public sealed class Node<T> where T : IComparable<T>,
IEquatable<T>
    {
        public Node(T v, Node<T> left = null, Node<T> right = null,
bool refreshProperties = true)
        {
            if (v == null) throw new ArgumentNullException(nameof(v));

            Value = v;
            SetLeftChild(left, refreshProperties);
            SetRightChild(right, refreshProperties);
        }

        public T Value { get; private set; }
        public Node<T> Left { get; private set; }
        public Node<T> Right { get; private set; }
        public Node<T> Parent { get; private set; }

        public int Height { get; private set; }

        public int Balance { get; private set; }

        public bool HasChildren => Left != null || Right != null;

        public void SetChild(Node<T> child, bool refreshProperties =
true)
        {
            if (child is null) throw new
InvalidOperationException("Use explicit methods for null values.");

            int comp = Value.CompareTo(child.Value);
            if (comp < 0)
                SetRightChild(child, refreshProperties);
            else if (comp > 0)
                SetLeftChild(child, refreshProperties);
            else
                throw new InvalidOperationException("Cannot set child
with the same value");
        }
```

```
        public void RemoveChild(Node<T> child, bool refreshProperties
= true)
        {
            if (child is null) throw new
InvalidOperationException("Use explicit methods for null values.");

            int comp = Value.CompareTo(child.Value);
            if (comp < 0)
                SetRightChild(null, refreshProperties);
            else if (comp > 0)
                SetLeftChild(null, refreshProperties);
            else
                throw new InvalidOperationException("Cannot set child
with the same value");
        }

        public void SetLeftChild(Node<T> left, bool refreshProperties
= true)
        {
            if (left ≠ null)
            {
                if (left.Value.CompareTo(Value) ⩾ 0) throw new
InvalidOperationException();

                left.Parent?.RemoveChild(left);
                left.Parent = this;
            }

            if (Left ≠ null) Left.Parent = null;

            Left = left;
            if (refreshProperties) RefreshOwnAndParentProperties();
        }

        public void SetRightChild(Node<T> right, bool
refreshProperties = true)
        {
            if (right ≠ null)
            {
                if (right.Value.CompareTo(Value) ⩽ 0) throw new
InvalidOperationException();

                right.Parent?.RemoveChild(right);
                right.Parent = this;
            }

            if (Right ≠ null) Right.Parent = null;

            Right = right;

            if (refreshProperties) RefreshOwnAndParentProperties();
```

```
}

public Node<T> LeftTurn()
{
    if (Right?.Balance < 0) return BigLeftTurn();

    Node<T>
        a = this,
        b = Right,
        y = b.Left;

    if (a.Parent ≠ null)
    {
        a.Parent.SetChild(b);
    }
    else
    {
        b.Parent = a.Parent;
    }

    a.SetRightChild(y);
    b.SetLeftChild(a);

    return b;
}

public Node<T> RightTurn()
{
    if (Left?.Balance > 0) return BigRightTurn();

    Node<T>
        a = this,
        b = a.Left,
        y = b.Right;

    if (a.Parent ≠ null)
    {
        a.Parent.SetChild(b);
    }
    else
    {
        b.Parent = a.Parent;
    }

    a.SetLeftChild(y);
    b.SetRightChild(a);

    return b;
}

public override string ToString()
```

```
{
    return $"Node({Value}, ({Left}, {Right})";
}

public string ToPrintableString()
{
    var sb = new StringBuilder();
    var queue = new Queue<Node<T>>();
    long counter = 1;
    queue.Enqueue(this);

    while (queue.Count > 0)
    {
        Node<T> current = queue.Dequeue();
        sb.Append(current.Value);

        if (current.Left ≠ null)
        {
            queue.Enqueue(current.Left);
            sb.Append($" {++counter}");
        }
        else
        {
            sb.Append(" 0");
        }


        if (current.Right ≠ null)
        {
            queue.Enqueue(current.Right);
            sb.Append($" {++counter}");
        }
        else
        {
            sb.Append(" 0");
        }

        sb.Append("\n");
    }

    return sb.ToString();
}

public Node<T> Insert(T x)
{
    Node<T> current = this;
    var inserted = false;

    while (!inserted)
    {
        int comp = current.Value.CompareTo(x);
```

```
            if (comp == 0)
            {
                throw new DuplicateNameException();
            }

            if (comp < 0)
            {
                if (current.Right != null)
                {
                    current = current.Right;
                }
                else
                {
                    current.SetRightChild(new Node<T>(x));
                    inserted = true;
                }
            }
            else
            {
                if (current.Left != null)
                {
                    current = current.Left;
                }
                else
                {
                    current.SetLeftChild(new Node<T>(x));
                    inserted = true;
                }
            }
        }

        return current.BalanceUpward();
    }

    public Node<T> Remove(T x)
    {
        Node<T> current = this;
        var removed = false;

        while (!removed)
        {
            int comp = current.Value.CompareTo(x);
            if (comp == 0)
            {
                current = RemoveFoundVertex(current);

                removed = true;
            }
            else if (comp < 0)
            {
                if (current.Right != null)
```

```
                {
                    current = current.Right;
                }
                else
                {
                    throw new InvalidOperationException();
                }
            }
            else
            {
                if (current.Left ≠ null)
                {
                    current = current.Left;
                }
                else
                {
                    throw new InvalidOperationException();
                }
            }
        }

        return current.BalanceUpward();
    }

    private static Node<T> RemoveFoundVertex(Node<T> current)
    {
        Node<T> toRemove = current;
        current = current.Parent;

        if (toRemove.HasChildren)
        {
            if (toRemove.Left == null)
            {
                if (current is null)
                {
                    current = toRemove.Right;
                    current.Parent = null;
                }
                else
                {
                    current.SetChild(toRemove.Right);
                }
            }
            else
            {
                Node<T> maximum = toRemove.Left.Maximum();
                current = maximum.Parent;
                if (maximum.HasChildren)
                {
                    current.SetChild(maximum.Left);
                }
```

```
            else
            {
                current.RemoveChild(maximum);
            }

            toRemove.Value = maximum.Value;
        }
    }
    else
    {
        current.RemoveChild(toRemove);
    }

    return current;
}

private Node<T> Maximum()
{
    Node<T> current = this;
    while (current.Right ≠ null)
    {
        current = current.Right;
    }

    return current;
}

private Node<T> BalanceUpward()
{
    Node<T> current = this;
    Node<T> res = current;

    while (current ≠ null)
    {
        if (current.Balance > 1)
        {
            current = current.LeftTurn();
        }
        else if (current.Balance < -1)
        {
            current = current.RightTurn();
        }
        else
        {
            res = current;
            current = current.Parent;
        }
    }

    return res;
}
```

```
        public void RefreshTree()
        {
            Left?.RefreshTree();
            Right?.RefreshTree();

            RefreshOwnProperties();
        }

        public void RefreshOwnProperties()
        {
            Height = 1 + Math.Max(Left?.Height ?? 0, Right?.Height ??
0);
            Balance = (Right?.Height ?? 0) - (Left?.Height ?? 0);
        }

        private void RefreshOwnAndParentProperties()
        {
            Node<T> node = this;

            while (node ≠ null)
            {
                node.RefreshOwnProperties();
                node = node.Parent;
            }
        }

        private Node<T> BigLeftTurn()
        {
            Node<T>
                a = this,
                b = a.Right,
                c = b.Left,
                x = c.Left,
                y = c.Right;

            // replace a with c
            if (a.Parent ≠ null)
            {
                a.Parent.SetChild(c);
            }
            else
            {
                c.Parent = a.Parent;
            }


            b.SetLeftChild(y);
            a.SetRightChild(x);

            c.SetLeftChild(a);
```

```
        c.SetRightChild(b);

        return c;
    }

    private Node<T> BigRightTurn()
    {
        Node<T>
            a = this,
            b = a.Left,
            c = b.Right,
            x = c.Left,
            y = c.Right;

        // replace a with c
        if (a.Parent ≠ null)
        {
            a.Parent?.SetChild(c);
        }
        else
        {
            c.Parent = a.Parent;
        }

        b.SetRightChild(x);
        a.SetLeftChild(y);

        c.SetLeftChild(b);
        c.SetRightChild(a);

        return c;
    }
}

public class Program
{
    private static string[] _input;
    private static int _currentLineIndex;

    private static string ReadLine()
    {
        return _input[_currentLineIndex++];
    }

    public static void Main(string[] args)
    {
        _input = File.ReadAllLines("input.txt");

        int n = int.Parse(ReadLine());
        var parents = new int[n];
        var nodes = new List<Node<int>>();
```

```
            for (var i = 0; i < n; i++)
            {
                string[] nodeParts = ReadLine().Split();

                // setting parents to child nodes
                int leftChild = int.Parse(nodeParts[1]) - 1,
                    rightChild = int.Parse(nodeParts[2]) - 1;

                if (leftChild ⩾ 0)
                    parents[leftChild] = nodes.Count;
                if (rightChild ⩾ 0)
                    parents[rightChild] = nodes.Count;


                nodes.Add(new Node<int>(int.Parse(nodeParts[0]),
refreshProperties: false));

                // checking for parents for non-root nodes
                if (i > 0)
nodes[parents[i]].SetChild(nodes[nodes.Count - 1], false);
            }

            for (int i = n - 1; i ⩾ 0; i--)
nodes[i].RefreshOwnProperties();

            using (var sw = new StreamWriter("output.txt"))
            {
                Node<int> root = nodes[0];
                int x = int.Parse(ReadLine());
                if (n == 1)
                {
                    sw.WriteLine(0);
                }
                else
                {
                    root = root.Remove(x);
                    sw.WriteLine(n - 1);
                    sw.WriteLine(root.ToPrintableString());
                }

            }
        }
    }
}
```

Результат

Верное решение!
Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.578	73199616	4077288	3877258
1	OK	0.046	11153408	27	17
2	OK	0.031	10686464	13	3
3	OK	0.031	11120640	20	11

Задача «Упорядоченное множество на АВЛ-дереве»

Условие

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 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.

Решение

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

namespace ItmoAlgos
{
    public struct ModificationResult
    {
        public bool Modified { get; set; }
        public int HeightDelta { get; set; }
    }

    public sealed class Node<T> where T : IComparable<T>
    {
        public Node(T v, Node<T> left = null, Node<T> right = null,
bool refreshProperties = true)
```

```
        {
            if (v == null) throw new ArgumentNullException(nameof(v));

            Value = v;
            SetLeftChild(left, refreshProperties);
            SetRightChild(right, refreshProperties);
        }

        public T Value { get; private set; }
        public Node<T> Left { get; private set; }
        public Node<T> Right { get; private set; }
        public Node<T> Parent { get; private set; }

        public int Height { get; private set; }

        public int Balance { get; private set; }

        public bool HasChildren => Left != null || Right != null;

        public void SetChild(Node<T> child, bool refreshProperties =
true)
        {
            if (child is null) throw new
InvalidOperationException("Use explicit methods for null values.");

            int comp = Value.CompareTo(child.Value);
            if (comp < 0)
                SetRightChild(child, refreshProperties);
            else if (comp > 0)
                SetLeftChild(child, refreshProperties);
            else
                throw new InvalidOperationException("Cannot set child
with the same value");
        }

        public void RemoveChild(Node<T> child, bool refreshProperties
= true)
        {
            if (child is null) throw new
InvalidOperationException("Use explicit methods for null values.");

            int comp = Value.CompareTo(child.Value);
            if (comp < 0)
                SetRightChild(null, refreshProperties);
            else if (comp > 0)
                SetLeftChild(null, refreshProperties);
            else
                throw new InvalidOperationException("Cannot set child
with the same value");
        }
```

```
        public void SetLeftChild(Node<T> left, bool refreshProperties
= true)
        {
            if (left ≠ null)
            {
                if (left.Value.CompareTo(Value) ⩾ 0) throw new
InvalidOperationException();

                left.Parent?.RemoveChild(left);
                left.Parent = this;
            }

            if (Left ≠ null) Left.Parent = null;

            Left = left;
            if (refreshProperties) RefreshOwnAndParentProperties();
        }

        public void SetRightChild(Node<T> right, bool
refreshProperties = true)
        {
            if (right ≠ null)
            {
                if (right.Value.CompareTo(Value) ⩽ 0) throw new
InvalidOperationException();

                right.Parent?.RemoveChild(right);
                right.Parent = this;
            }

            if (Right ≠ null) Right.Parent = null;

            Right = right;

            if (refreshProperties) RefreshOwnAndParentProperties();
        }

        public Node<T> LeftTurn()
        {
            if (Right?.Balance < 0) return BigLeftTurn();

            Node<T>
                a = this,
                b = Right,
                y = b.Left;

            if (a.Parent ≠ null)
                a.Parent.SetChild(b);
            else
                b.Parent = a.Parent;
```

```
    a.SetRightChild(y);
    b.SetLeftChild(a);

    return b;
}

public Node<T> RightTurn()
{
    if (Left?.Balance > 0) return BigRightTurn();

    Node<T>
        a = this,
        b = a.Left,
        y = b.Right;

    if (a.Parent ≠ null)
        a.Parent.SetChild(b);
    else
        b.Parent = a.Parent;

    a.SetLeftChild(y);
    b.SetRightChild(a);

    return b;
}

public override string ToString()
{
    return $"Node({Value}, ({Left}, {Right})";
}

public string ToPrintableString()
{
    var sb = new StringBuilder();
    var queue = new Queue<Node<T>>();
    long counter = 1;
    queue.Enqueue(this);

    while (queue.Count > 0)
    {
        Node<T> current = queue.Dequeue();
        sb.Append(current.Value);

        if (current.Left ≠ null)
        {
            queue.Enqueue(current.Left);
            sb.Append($" {++counter}");
        }
        else
        {
            sb.Append(" 0");
```

```
            }


            if (current.Right ≠ null)
            {
                queue.Enqueue(current.Right);
                sb.Append($" {++counter}");
            }
            else
            {
                sb.Append(" 0");
            }

            sb.Append("\n");
        }

        return sb.ToString();
    }

    public Node<T> Insert(T x)
    {
        Node<T> current = this;
        var inserted = false;

        while (!inserted)
        {
            int comp = current.Value.CompareTo(x);
            if (comp == 0) throw new DuplicateNameException();

            if (comp < 0)
            {
                if (current.Right ≠ null)
                {
                    current = current.Right;
                }
                else
                {
                    current.SetRightChild(new Node<T>(x));
                    inserted = true;
                }
            }
            else
            {
                if (current.Left ≠ null)
                {
                    current = current.Left;
                }
                else
                {
                    current.SetLeftChild(new Node<T>(x));
                    inserted = true;
```

```
            }
        }
    }

    return current.BalanceUpward();
}

public Node<T> Remove(T x)
{
    Node<T> current = this;
    var removed = false;

    while (!removed)
    {
        int comp = current.Value.CompareTo(x);
        if (comp == 0)
        {
            current = RemoveFoundVertex(current);

            removed = true;
        }
        else if (comp < 0)
        {
            if (current.Right ≠ null)
                current = current.Right;
            else
                throw new InvalidOperationException();
        }
        else
        {
            if (current.Left ≠ null)
                current = current.Left;
            else
                throw new InvalidOperationException();
        }
    }

    return current.BalanceUpward();
}

private static Node<T> RemoveFoundVertex(Node<T> current)
{
    Node<T> toRemove = current;
    current = current.Parent;

    if (toRemove.HasChildren)
    {
        if (toRemove.Left == null)
        {
            if (current is null)
            {
```

```
                current = toRemove.Right;
                current.Parent = null;
            }
            else
            {
                current.SetChild(toRemove.Right);
            }
        }
        else
        {
            Node<T> maximum = toRemove.Left.Maximum();
            current = maximum.Parent;
            if (maximum.HasChildren)
                current.SetChild(maximum.Left);
            else
                current.RemoveChild(maximum);

            toRemove.Value = maximum.Value;
        }
    }
    else
    {
        current.RemoveChild(toRemove);
    }

    return current;
}

private Node<T> Maximum()
{
    Node<T> current = this;
    while (current.Right ≠ null) current = current.Right;

    return current;
}

private Node<T> BalanceUpward()
{
    Node<T> current = this;
    Node<T> res = current;

    while (current ≠ null)
        if (current.Balance > 1)
        {
            current = current.LeftTurn();
        }
        else if (current.Balance < -1)
        {
            current = current.RightTurn();
        }
        else
```

```
            {
                res = current;
                current = current.Parent;
            }

        return res;
    }

    public void RefreshTree()
    {
        Left?.RefreshTree();
        Right?.RefreshTree();

        RefreshOwnProperties();
    }

    public void RefreshOwnProperties()
    {
        Height = 1 + Math.Max(Left?.Height ?? 0, Right?.Height ??
0);
        Balance = (Right?.Height ?? 0) - (Left?.Height ?? 0);
    }

    private void RefreshOwnAndParentProperties()
    {
        Node<T> node = this;

        while (node ≠ null)
        {
            node.RefreshOwnProperties();
            node = node.Parent;
        }
    }

    private Node<T> BigLeftTurn()
    {
        Node<T>
            a = this,
            b = a.Right,
            c = b.Left,
            x = c.Left,
            y = c.Right;

        // replace a with c
        if (a.Parent ≠ null)
            a.Parent.SetChild(c);
        else
            c.Parent = a.Parent;


        b.SetLeftChild(y);
```

```
    a.SetRightChild(x);

    c.SetLeftChild(a);
    c.SetRightChild(b);

    return c;
}

private Node<T> BigRightTurn()
{
    Node<T>
        a = this,
        b = a.Left,
        c = b.Right,
        x = c.Left,
        y = c.Right;

    // replace a with c
    if (a.Parent ≠ null)
        a.Parent?.SetChild(c);
    else
        c.Parent = a.Parent;

    b.SetRightChild(x);
    a.SetLeftChild(y);

    c.SetLeftChild(b);
    c.SetRightChild(a);

    return c;
}

public bool Contains(T x)
{
    Node<T> current = this;

    while (true)
    {
        int comp = current.Value.CompareTo(x);
        if (comp == 0)
        {
            return true;
        }

        if (comp < 0)
        {
            if (current.Right ≠ null)
                current = current.Right;
            else
                return false;
        }
```

```
                else
                {
                    if (current.Left ≠ null)
                        current = current.Left;
                    else
                        return false;
                }
            }
        }
    }

    public class Tree<T> where T : IComparable<T>
    {
        private Node<T> _root;
        public int Balance ⇒ _root?.Balance ?? 0;

        public void Insert(T x)
        {
            try
            {
                _root = _root == null ? new Node<T>(x) :
_root.Insert(x);
            }
            catch (DuplicateNameException)
            {
                // nothing.
            }
        }

        public void Remove(T x)
        {
            if (_root is null) return;

            if (_root.Value.CompareTo(x) == 0 && !_root.HasChildren)
            {
                _root = null;
            }
            else
            {
                try
                {
                    _root = _root.Remove(x);
                }
                catch (InvalidOperationException)
                {
                    // nothing
                }
            }
        }

        public bool ContainsValue(T x)
```

```
        {
            return _root?.Contains(x) ?? false;
        }
    }

    public class Program
    {
        private static string[] _input;
        private static int _currentLineIndex;

        private static string ReadLine()
        {
            return _input[_currentLineIndex++];
        }

        public static void Main(string[] args)
        {
            _input = File.ReadAllLines("input.txt");

            int n = int.Parse(ReadLine());
            var t = new Tree<int>();


            using (var sw = new StreamWriter("output.txt"))
            {
                for (var i = 0; i < n; i++)
                {
                    string[] cmdParts = ReadLine().Split();
                    int x = int.Parse(cmdParts[1]);

                    switch (cmdParts[0][0])
                    {
                        case 'A':
                            t.Insert(x);
                            sw.WriteLine(t.Balance);
                            break;

                        case 'D':
                            t.Remove(x);
                            sw.WriteLine(t.Balance);
                            break;

                        case 'C':
                            sw.WriteLine(t.ContainsValue(x) ? "Y" :
"N");
                            break;
                    }
                }
            }
        }
    }
```

```
}
```

Результат

Верное решение!
Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		1.671	45584384	2678110	731071
1	OK	0.031	11960320	33	19
2	OK	0.046	12337152	114	66
3	OK	0.031	12349440	154	90