

Санкт-Петербургский национальный исследовательский университет
информационных технологий, механики и оптики

Кафедра информатики и прикладной математики

Лабораторная работа №7
Дисциплина «Алгоритмы и структуры данных»

Выполнил:
Молодецкий Арсений Алексеевич
Группа Р3217

Санкт-Петербург
2019

Задание №1

AVL-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 AVL-дерева, приведенное выше, можно сформулировать следующим образом: для любой ее вершины V выполняется следующее неравенство:

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

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.

```
#include <iostream>

using namespace std;

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

typedef struct Node{
    Node* left;
    Node* right;
    Node* parent;
    int height;
    int value;
} Node;

int updateHeight(Node* node) {
    int depth = 1;
    int depthLeft = 0;
    if (node->left != nullptr) depthLeft = updateHeight(node->left);
    int depthRight = 0;
    if (node->right != nullptr) depthRight = updateHeight(node->right);
    node->height = depth + max(depthLeft,depthRight);
    return node->height;
}

int getBalance(Node* node) {
    int heightRight = 0;
    if (node->right != nullptr) heightRight = node->right->height;
    int heightLeft = 0;
    if (node->left != nullptr) heightLeft = node->left->height;
    return heightRight - heightLeft;
}

int main() {
    int n;
    cin >> n;
    Node *nodes = new Node[n];
    for (int i = 0; i < n; ++i) {
```

```
    nodes[i].right = nullptr;
    nodes[i].left = nullptr;
    nodes[i].parent = nullptr;
    nodes[i].height = 0;
}
for (int i = 0; i < n; ++i) {
    int k,l,r;
    cin >> k >> l >> r;
    nodes[i].value = k;
    if (l != 0){
        nodes[i].left = &(nodes[l - 1]);
    }
    if (r != 0){
        nodes[i].right = &(nodes[r - 1]);
    }
}
updateHeight(&nodes[0]);

for (int i = 0; i < n; ++i) {
    cout << getBalance(&nodes[i]) << '\n';
}

return 0;
}
```


Задание №2

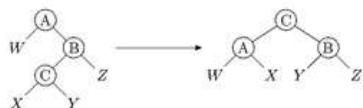
Для балансировки АВЛ-дерева при операциях вставки и удаления производятся *левые и правые* повороты. Левый поворот в вершине производится, когда баланс этой вершины больше 1, аналогично, правый поворот производится при балансе, меньшем -1.

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

Малый левый поворот осуществляется следующим образом:



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



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

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

using namespace std;

#ifndef 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 Node {
    int left, right, key;
} *tree;

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

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

```

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

        if (tree[i].right) {
            d = max(count(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) {
    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;
    }

    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; }

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

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

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

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

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

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

    tree = new 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;  
}  
  
count(0);  
left_rotation(0);  
  
calc_index(1);  
cout << n << nl;  
prinNode(1);  
  
return 0;  
}
```


Задание №3

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.

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

namespace ItmoAlgos
{
    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.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.ToString());
    }
}
}
```


Задача №4

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.

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

namespace ItmoAlgos
{
    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 _currentIndex;

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

    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.ToString());
        }
    }
}
}

```

Номер	Препаратор	Срок	Вид	Родитель-препаратор	Проверка	Проверяющий
1	ДК	03/07/2003	05/07/2003			007024
2	ДК	03/07/2004	07/07/2004			007025
3	ДК	03/07/2005	07/07/2005			007026
4	ДК	03/07/2006	07/07/2006			007027
5	ДК	03/07/2007	07/07/2007			007028
6	ДК	03/07/2008	07/07/2008			007029
7	ДК	03/07/2009	07/07/2009			007030
8	ДК	03/07/2010	07/07/2010			007031
9	ДК	03/07/2011	07/07/2011			007032
10	ДК	03/07/2012	07/07/2012			007033
11	ДК	03/07/2013	07/07/2013			007034
12	ДК	03/07/2014	07/07/2014			007035
13	ДК	03/07/2015	07/07/2015			007036
14	ДК	03/07/2016	07/07/2016			007037
15	ДК	03/07/2017	07/07/2017			007038
16	ДК	03/07/2018	07/07/2018			007039
17	ДК	03/07/2019	07/07/2019			007040
18	ДК	03/07/2020	07/07/2020			007041
19	ДК	03/07/2021	07/07/2021			007042
20	ДК	03/07/2022	07/07/2022			007043
21	ДК	03/07/2023	07/07/2023			007044
22	ДК	03/07/2024	07/07/2024			007045
23	ДК	03/07/2025	07/07/2025			007046
24	ДК	03/07/2026	07/07/2026			007047
25	ДК	03/07/2027	07/07/2027			007048
26	ДК	03/07/2028	07/07/2028			007049
27	ДК	03/07/2029	07/07/2029			007050
28	ДК	03/07/2030	07/07/2030			007051
29	ДК	03/07/2031	07/07/2031			007052
30	ДК	03/07/2032	07/07/2032			007053
31	ДК	03/07/2033	07/07/2033			007054
32	ДК	03/07/2034	07/07/2034			007055
33	ДК	03/07/2035	07/07/2035			007056
34	ДК	03/07/2036	07/07/2036			007057
35	ДК	03/07/2037	07/07/2037			007058
36	ДК	03/07/2038	07/07/2038			007059
37	ДК	03/07/2039	07/07/2039			007060
38	ДК	03/07/2040	07/07/2040			007061
39	ДК	03/07/2041	07/07/2041			007062
40	ДК	03/07/2042	07/07/2042			007063
41	ДК	03/07/2043	07/07/2043			007064
42	ДК	03/07/2044	07/07/2044			007065
43	ДК	03/07/2045	07/07/2045			007066
44	ДК	03/07/2046	07/07/2046			007067
45	ДК	03/07/2047	07/07/2047			007068
46	ДК	03/07/2048	07/07/2048			007069
47	ДК	03/07/2049	07/07/2049			007070
48	ДК	03/07/2050	07/07/2050			007071
49	ДК	03/07/2051	07/07/2051			007072
50	ДК	03/07/2052	07/07/2052			007073
51	ДК	03/07/2053	07/07/2053			007074
52	ДК	03/07/2054	07/07/2054			007075
53	ДК	03/07/2055	07/07/2055			007076
54	ДК	03/07/2056	07/07/2056			007077
55	ДК	03/07/2057	07/07/2057			007078
56	ДК	03/07/2058	07/07/2058			007079
57	ДК	03/07/2059	07/07/2059			007080
58	ДК	03/07/2060	07/07/2060			007081
59	ДК	03/07/2061	07/07/2061			007082
60	ДК	03/07/2062	07/07/2062			007083
61	ДК	03/07/2063	07/07/2063			007084
62	ДК	03/07/2064	07/07/2064			007085
63	ДК	03/07/2065	07/07/2065			007086
64	ДК	03/07/2066	07/07/2066			007087
65	ДК	03/07/2067	07/07/2067			007088
66	ДК	03/07/2068	07/07/2068			007089
67	ДК	03/07/2069	07/07/2069			007090
68	ДК	03/07/2070	07/07/2070			007091
69	ДК	03/07/2071	07/07/2071			007092
70	ДК	03/07/2072	07/07/2072			007093
71	ДК	03/07/2073	07/07/2073			007094
72	ДК	03/07/2074	07/07/2074			007095
73	ДК	03/07/2075	07/07/2075			007096
74	ДК	03/07/2076	07/07/2076			007097
75	ДК	03/07/2077	07/07/2077			007098
76	ДК	03/07/2078	07/07/2078			007099
77	ДК	03/07/2079	07/07/2079			007100
78	ДК	03/07/2080	07/07/2080			007101
79	ДК	03/07/2081	07/07/2081			007102
80	ДК	03/07/2082	07/07/2082			007103
81	ДК	03/07/2083	07/07/2083			007104
82	ДК	03/07/2084	07/07/2084			007105
83	ДК	03/07/2085	07/07/2085			007106
84	ДК	03/07/2086	07/07/2086			007107
85	ДК	03/07/2087	07/07/2087			007108
86	ДК	03/07/2088	07/07/2088			007109
87	ДК	03/07/2089	07/07/2089			007110
88	ДК	03/07/2090	07/07/2090			007111
89	ДК	03/07/2091	07/07/2091			007112
90	ДК	03/07/2092	07/07/2092			007113
91	ДК	03/07/2093	07/07/2093			007114
92	ДК	03/07/2094	07/07/2094			007115
93	ДК	03/07/2095	07/07/2095			007116
94	ДК	03/07/2096	07/07/2096			007117
95	ДК	03/07/2097	07/07/2097			007118
96	ДК	03/07/2098	07/07/2098			007119
97	ДК	03/07/2099	07/07/2099			007120
98	ДК	03/07/2100	07/07/2100			007121
99	ДК	03/07/2101	07/07/2101			007122
100	ДК	03/07/2102	07/07/2102			007123
101	ДК	03/07/2103	07/07/2103			007124
102	ДК	03/07/2104	07/07/2104			007125
103	ДК	03/07/2105	07/07/2105			007126
104	ДК	03/07/2106	07/07/2106			007127
105	ДК	03/07/2107	07/07/2107			007128
106	ДК	03/07/2108	07/07/2108			007129
107	ДК	03/07/2109	07/07/2109			007130
108	ДК	03/07/2110	07/07/2110			007131
109	ДК	03/07/2111	07/07/2111			007132
110	ДК	03/07/2112	07/07/2112			007133
111	ДК	03/07/2113	07/07/2113			007134
112	ДК	03/07/2114	07/07/2114			007135
113	ДК	03/07/2115	07/07/2115			007136
114	ДК	03/07/2116	07/07/2116			007137
115	ДК	03/07/2117	07/07/2117			007138
116	ДК	03/07/2118	07/07/2118			007139
117	ДК	03/07/2119	07/07/2119			007140
118	ДК	03/07/2120	07/07/2120			007141
119	ДК	03/07/2121	07/07/2121			007142
120	ДК	03/07/2122	07/07/2122			007143
121	ДК	03/07/2123	07/07/2123			007144
122	ДК	03/07/2124	07/07/2124			007145
123	ДК	03/07/2125	07/07/2125			007146
124	ДК	03/07/2126	07/07/2126			007147
125	ДК	03/07/2127	07/07/2127			007148
126	ДК	03/07/2128	07/07/2128			007149
127	ДК	03/07/2129	07/07/2129			007150
128	ДК	03/07/2130	07/07/2130			007151
129	ДК	03/07/2131	07/07/2131			007152
130	ДК	03/07/2132	07/07/2132			007153
131	ДК	03/07/2133	07/07/2133			007154
132	ДК	03/07/2134	07/07/2134			007155
133	ДК	03/07/2135	07/07/2135			007156
134	ДК	03/07/2136	07/07/2136			007157
135	ДК	03/07/2137	07/07/2137			007158
136	ДК	03/07/2138	07/07/2138			007159
137	ДК	03/07/2139	07/07/2139			007160
138	ДК	03/07/2140	07/07/2140			007161
139	ДК	03/07/2141	07/07/2141			007162
140	ДК	03/07/2142	07/07/2142			007163
141	ДК	03/07/2143	07/07/2143			007164
142	ДК	03/07/2144	07/07/2144			007165
143	ДК	03/07/2145	07/07/2145			007166
144	ДК	03/07/2146	07/07/2146			007167
145	ДК	03/07/2147	07/07/2147			007168
146	ДК	03/07/2148	07/07/2148			007169
147	ДК	03/07/2149	07/07/2149			007170
148	ДК	03/07/2150	07/07/2150			007171
149	ДК	03/07/2151	07/07/2151			007172
150	ДК	03/07/2152	07/07/2152			007173
151	ДК	03/07/2153	07/07/2153			007174
152	ДК	03/07/2154	07/07/2154			007175
153	ДК	03/07/2155	07/07/2155			007176
154	ДК	03/07/2156	07/07/2156			007177
155	ДК	03/07/2157	07/07/2157			007178
156	ДК	03/07/2158	07/07/2158			007179
157	ДК	03/07/2159	07/07/2159			007180
158	ДК	03/07/2160	07/07/2160			007181
159	ДК	03/07/2161	07/07/2161			007182
160	ДК	03/07/2162	07/07/2162			007183
161	ДК	03/07/2163	07/07/2163			007184
162	ДК	03/07/2164	07/07/2164			007185
163	ДК	03/07/2165	07/07/2165			007186
164	ДК	03/07/2166	07/07/2166			007187
165	ДК	03/07/2167	07/07/2167			007188
166	ДК	03/07/2168	07/07/2168			007189
167	ДК	03/07/2169	07/07/2169			007190
168	ДК	03/07/2170	07/07/2170			007191
169	ДК	03/07/2171	07/07/2171			007192
170	ДК	03/07/2172	07/07/2172			007193
171	ДК	03/07/2173	07/07/2173			007194
172	ДК	03/07/2174	07/07/2174			007195
173	ДК	03/07/2175	07/07/2175			007196
174	ДК	03/07/2176	07/07/2176			007197
175	ДК	03/07/2177	07/07/2177			007198
176	ДК	03/07/2178	07/07/2178			007199
177	ДК	03/07/2179	07/07/2179			007200
178	ДК	03/07/2180	07/07/2180			007201
179	ДК	03/07/2181	07/07/2			

Задача №5

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 ($1 \leq N \leq 2 \cdot 10^5$) — 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.

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

namespace ItmoAlgos
{
    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;
                }
            }
        }
    }
}

```


№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		1.640	45580288	2678110	731071
1	OK	0.046	11907072	33	19
2	OK	0.031	12341248	114	66
3	OK	0.046	12337152	154	90
4	OK	0.046	12324864	154	91
5	OK	0.046	12419072	154	90
6	OK	0.031	12455936	154	95
7	OK	0.031	12427264	154	91
8	OK	0.031	12468224	154	94
9	OK	0.046	12447744	154	95
10	OK	0.031	12505088	154	90
11	OK	0.046	12427264	154	90
12	OK	0.031	11554816	154	90
13	OK	0.031	12439552	154	95
14	OK	0.046	12402688	154	97
15	OK	0.031	12431360	154	94
16	OK	0.031	12533760	154	93
17	OK	0.031	12365824	154	90
18	OK	0.046	12484608	154	90
19	OK	0.046	12386304	154	98
20	OK	0.031	12390400	154	93
21	OK	0.046	12423168	154	92
22	OK	0.046	12488704	154	98
23	OK	0.953	30302208	1000008	616458
24	OK	1.093	30302208	1000008	622272
25	OK	1.171	30351360	1000008	625335
26	OK	1.218	30289920	1000008	628546
27	OK	1.234	30330880	1000008	631472
28	OK	1.218	30334976	1000008	632217
29	OK	1.296	30351360	1000008	631772
30	OK	1.296	30351360	1000008	631071
31	OK	1.250	30347264	1000008	630132
32	OK	1.218	30457856	1017957	630451
33	OK	0.937	30285824	1000008	616595
34	OK	1.140	30412800	1000008	622199
35	OK	1.156	30310400	1000008	625057
36	OK	1.187	30310400	1000008	628040
37	OK	1.203	30306304	1000008	631495
38	OK	1.234	30339072	1000008	632086
39	OK	1.203	30318592	1000008	631753
40	OK	1.234	30310400	1000008	630849
41	OK	1.265	30380032	1000008	630110
42	OK	1.250	30527488	1018151	630800
43	OK	0.031	11960320	756	369
44	OK	0.031	12034048	758	432
45	OK	0.031	11980800	1659	408
46	OK	0.046	12099584	723	383
47	OK	0.031	11956224	723	385
48	OK	0.078	11972608	723	415
49	OK	0.031	11968512	723	415
50	OK	0.046	11952128	1668	377
51	OK	0.046	12357632	1660	396
52	OK	0.031	12226560	5348	2337
53	OK	0.046	12267520	5350	2848
54	OK	0.046	12193792	10439	2648
55	OK	0.046	12140544	5238	2343
56	OK	0.046	12206080	5238	2465
57	OK	0.046	12173312	5238	2719
58	OK	0.046	12177408	5238	2719
59	OK	0.031	12238848	10450	2421
60	OK	0.046	12800000	10439	2405
61	OK	0.046	13127680	32784	12708
62	OK	0.046	13074432	32787	14896
63	OK	0.046	13094912	56716	12715
64	OK	0.031	13123584	31674	12778
65	OK	0.062	13172736	31674	13220
66	OK	0.046	13082624	31674	14383
67	OK	0.046	13152256	31674	14825
68	OK	0.046	13148160	56748	13671
69	OK	0.078	13991936	56716	13193
70	OK	0.078	18468864	162462	57855
71	OK	0.078	18554880	162466	68948
72	OK	0.078	19185664	258205	71756
73	OK	0.062	17813504	152067	59306
74	OK	0.062	17805312	152067	59903
75	OK	0.062	17801216	152067	66900
76	OK	0.062	17874944	152067	67497
77	OK	0.078	18702336	258312	70001
78	OK	0.171	20357120	258332	58111
79	OK	0.296	28905472	811002	274035
80	OK	0.312	28884992	811006	332612
81	OK	0.281	29798400	1222794	299942
82	OK	0.234	28766208	799892	286940
83	OK	0.218	28737536	799892	282227
84	OK	0.256	28667904	799892	324420
85	OK	0.234	28663808	799892	319707
86	OK	0.234	29155328	1222871	284516
87	OK	0.734	28893184	1223246	288111
88	OK	0.640	45088768	1888898	600000
89	OK	0.609	45006848	1888903	731071
90	OK	0.656	45580288	2677526	600067
91	OK	0.506	39329792	1777788	601696
92	OK	0.484	39333888	1777788	632768
93	OK	0.453	39276544	1777788	698302
94	OK	0.484	39276544	1777788	698303
95	OK	0.593	43900928	2678110	611713
96	OK	1.640	37195776	2677266	600286