

МИНОБРНАУКИ РОССИИ

федеральное государственное бюджетное образовательное учреждение высшего образования

«Санкт-Петербургский государственный технологический институт (технический университет)»

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| УГНС | | 09.00.00 | Информатика и вычислительная техника | | |
| Направление подготовки | | 09.03.01 | Информатика и вычислительная техника | | |
| Направленность (профиль) | |  | Системы автоматизированного проектирования | | |
| Форма обучения | |  | очная | | |
|  | |  |  | | |
| Факультет | |  | Информационных технологий и управления | | |
| Кафедра | |  | Систем автоматизированного проектирования и управления | | |
| Учебная дисциплина | |  | Разработка программных систем | | |
| Курс | IIII | | | Группа | 404 |

Отчёт по контрольной работе № 1

Вариант № 2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Исполнитель: |  |  |  |  |
| обучающийся группы 403 |  |  |  | Азаров Даниил Константинович |
|  |  | (дата, подпись) |  |  |
|  |  |  |  |  |
| Проверили: |  |  |  | Корниенко Иван Григорьевич |
|  |  | (дата, подпись) |  | Федин Алексей Константинович |

# 1. Постановка задачи

Необходимо реализовать красно-чёрное дерево, продемонстрировать характерные особенности, реализовать возможность добавления и удаления элементов, визуализировать дерево. В программе должны быть предусмотрены три варианта заполнения: пользователем с клавиатуры, из файла и случайными числами.

# 2. Исходные данные

В качестве исходных данных программа использует вводимые пользователем целые числа, которые будут храниться в виде красно-чёрного дерева.

# 3. Особые ситуации

— Если пользователь при выборе случайного ввода решит задать верхнюю границу выше заданной ниже границы, то он получит предупреждение и должен будет ввести нормальную верхнюю границу.

— Если пользователь ввёл существующее имя файла для вывода, то предоставить ему выбор перезаписать файл или выбрать новое имя.

— Если пользователь выбрал несуществующий файл для ввода или файл с неподходящими данными, то сообщить ему об ошибке прочтения файла.

# 4. Математические методы и алгоритмы решения задачи

У красно-чёрного дерева как у структуры данных есть ряд своих особенностей:

* Каждый узел является либо красным, либо чёрным.
* Корень дерева является чёрным узлом.
* Каждый лист дерева является чёрным узлом.
* Если узел красный, то все его дочерние узлы чёрные.
* Для каждого узла все простые пути от него до листьев содержат одинаковое количество чёрных узлов.

При выполнении вставки или удаления из дерева некоторые из этих особенностей могут нарушаться, поэтому после выполнения этих действий необходимо проверять соблюдение этих условий.

## 4.1 Вставка

При вставке внутрь красно-чёрного дерева могут нарушиться его 2 и 4 свойства, т.к. узел, который мы вставляем, мы будем окрашивать в красный цвет. Второе свойство будет нарушено, если вставляемый узел окажется корнем, а четвёртое, если его родитель будет обладать красным цветом.

Чтобы исправить получившиеся нарушения, мы будем регулировать дерево за счёт изменения цвета узлов дерева, правых поворотов и левых поворотов.

Если узел станет красным, то мы просто его перекрашиваем, однако ситуация будет сложнее, если необходимо будет исправить два идущих подряд красных узла. В этой ситуации у нас есть несколько разных случаев:

1. Дядя узла красный.
2. Дядя узла чёрный (Треугольник).
3. Дядя узла чёрный (Линия).

Рассмотрим первый случай:

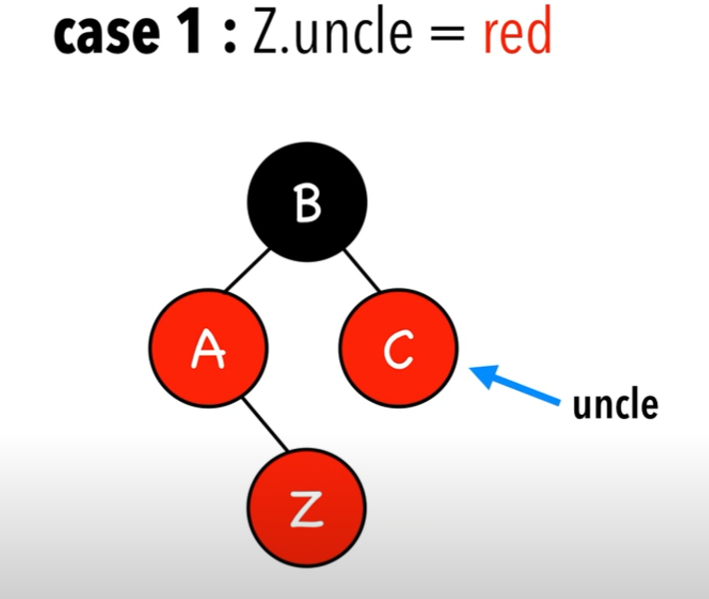


Рисунок 1 — 1 случай

Для исправления нам необходимо перекрасить родителя и дядю вставляемого элемента в чёрный, а дедушку в красный.

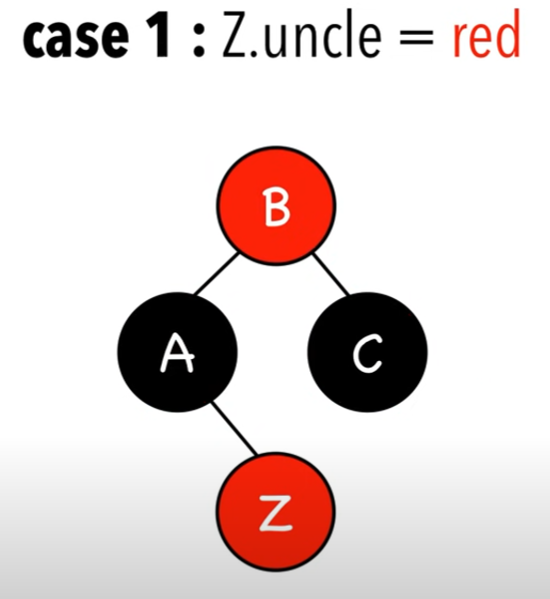


Рисунок 2 — 1 случай после исправления

Теперь рассмотрим второй случай:

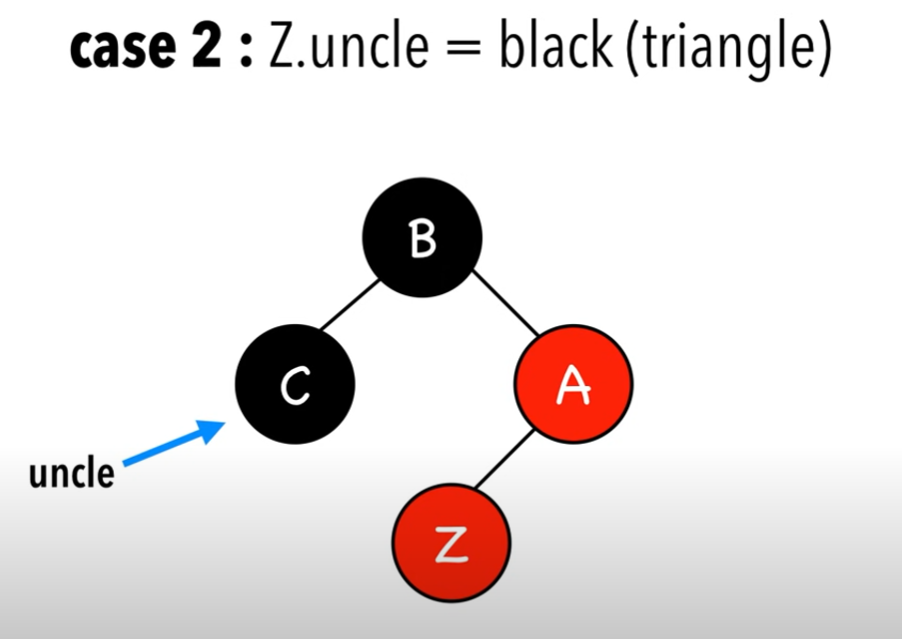


Рисунок 3 — 2 случай

Для исправления четвёртого свойства здесь нам будет необходимо повернуть дерево по родителю вставляемого элемента в противоположную сторону от вставляемого элемента (если он левый ребёнок — вправо и наоборот). Таким образом мы перейдём к случаю 3.

Рассмотрим третий случай:

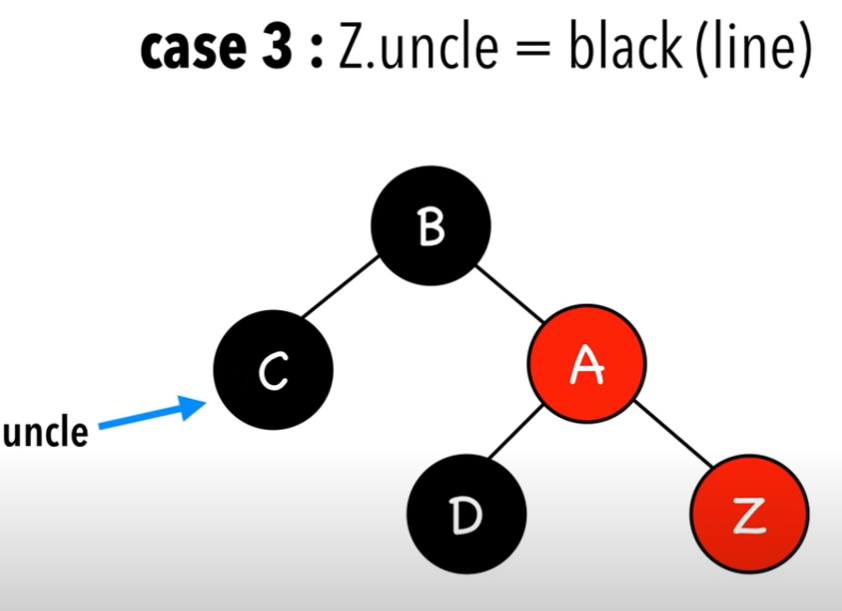


Рисунок 4 — 3 случай

Для исправления нарушения нам опять нужно повернуть дерево по тем же правилам, но теперь опираясь на дедушку вставляемого узла. После поворота нам нужно будет перекрасить родителя вставляемого узла в чёрный, а его брата после поворота в красный. Тогда никаких нарушений в дереве не будет.

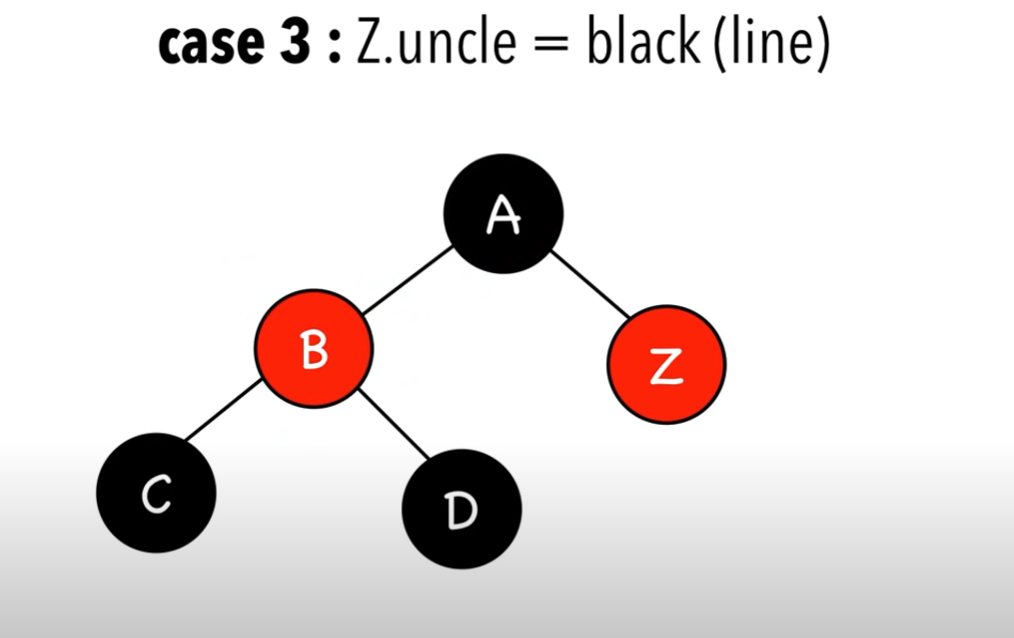


Рисунок 5 — 3 случай после исправления

## 4.2 Удаление

При удалении могут возникнуть проблемы только в том случае, если удаляемый узел будет чёрным. Могут быть нарушены 2, 4 и 5 свойства. Эти нарушения могут возникнуть при нескольких случаях:

* Брат удаляемого узла красный.
* Брат удаляемого узла чёрный, а его дочерние узлы тоже чёрные.
* Брат удаляемого узла чёрный, его левый дочерний красный, а правый чёрный.
* Брат удаляемого узла чёрный, его левый дочерний чёрный, а правый красный.

Рассмотрим первый случай:

Его нам необходимо привести к одному из следующих случаев. Делается это за счёт обмена цветами между братом и родителем, и левого поворота.

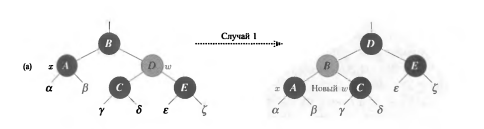


Рисунок 6 — 1 случай и его решение

Рассмотрим второй случай:

Необходимо перекрасить брата удаляемого узла в красный, и далее применять балансировку к отцу.

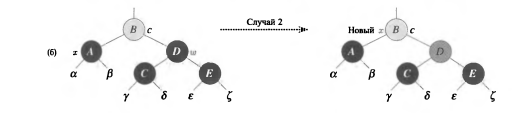


Рисунок 7 — 2 случай и его решение

Рассмотрим 3 случай:

Необходимо поменять цвета брата и его красного ребёнка, а затем совершить правый поворот с опорой на брата. Что в свою очередь приведёт нас к 4 случаю.

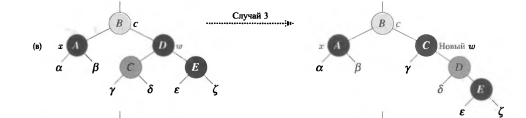


Рисунок 8 — 3 случай и его решение

Рассмотрим четвёртый случай:

Необходимо перекрасить брата в цвет отца, а ребёнка и отца в чёрный. После чего повернуть влево с опором на родителя.

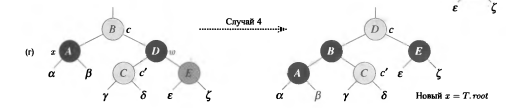


Рисунок 9 — 4 случай и его решение

# 5. Форматы представления данных

Таблица 1 — Классы, используемые в программе.

|  |  |  |  |
| --- | --- | --- | --- |
| **Название** | **Имя переменной** | **Тип** | **Описание** |
| Node | - | - | Представляет структуру узла красно-черного дерева |
| Data | double | Данные узла |
|  | IsRedColor | bool | Цвет |
|  | IsRightChild | bool | Расположение от родителя |
|  | Parent | Node | Родитель |
|  | \_rightChild | Node | Правый ребенок |
|  | \_leftChild | Node | Левый ребенок |
|  | **Метод** | **Тип** | **Описание** |
|  | GetBrother | Node | Возвращает брата узла |
|  | IsBrotherColorRed | bool | Возвращает цвет брата |
| RedBlackTree | - | - | Реализует структуру дерева |
| \_root | Node | Хранит корень дерева |
| **Метод** | **Тип** | **Описание** |
| AddData | void | Добавляет узел |
| DeleteData | void | Удаляет узел |
| GetTreeDataArray | List<List<double?[]>> | Возвращает все данные |
| FixInsertNode | void | Балансирует после вставки |
| LeftRotation | void | Поворачивает узел влево |
| RightRotation | void | Поворачивает узел вправо |
| FindNode | Node | Находит узел |
|  | FindeMinimumRightNode | Node | Находит минимальный правый узел |
|  | DeleteFixedUp | void | Балансирует после удаления |

# 6. Структура программы

Таблица 2 — Модули, на которые разбита программа.

|  |  |
| --- | --- |
| **Имя** | **Описание** |
| Maim | Основной модуль. |
| RedBlackTreeProject | Модуль, содержащий реализацию класса RedBlackTreeProject. |
| UI | Модуль, содержащий реализацию класса UI. |

На рисунке 10 представлена блок-схема алгоритма вставки в дерево, представленная в программе:



Рисунок 10 — блок-схема вставки в дерево.

# 7. Описание хода выполнения лабораторной работы

1. Были реализованы первые функции пользовательского интерфейса.
2. Были написаны функции вставки в дерево, удаления, поворота и вывода дерева.
3. Была исправлена функция удаления, т.к. при удалении оба ребёнка могли быть больше родителя.
4. Был добавлен ввод с клавиатуры.
5. Было добавлено меню.
6. Был исправлен вывод дерева после удаления элемента. Функция вывода сохраняло и множило детей листа из-за чего выводимое дерево бесконечно двоило своих детей.
7. Была исправлена функция поиска элемента, т.к. она выдавала исключение при попытке поиска несуществующего элемента.
8. Был добавлен ввод случайными числами.
9. Была добавлена возможность сохранения в файл.
10. Была исправлена ошибка при в воде границ генерации случайных чисел.
11. Была добавлена возможность ввода из файла.
12. Были написаны модульные тесты.
13. Был написан отчёт.

# 8. Результаты работы программы

На рисунках 11 - 13 представлены скриншоты работы программы.

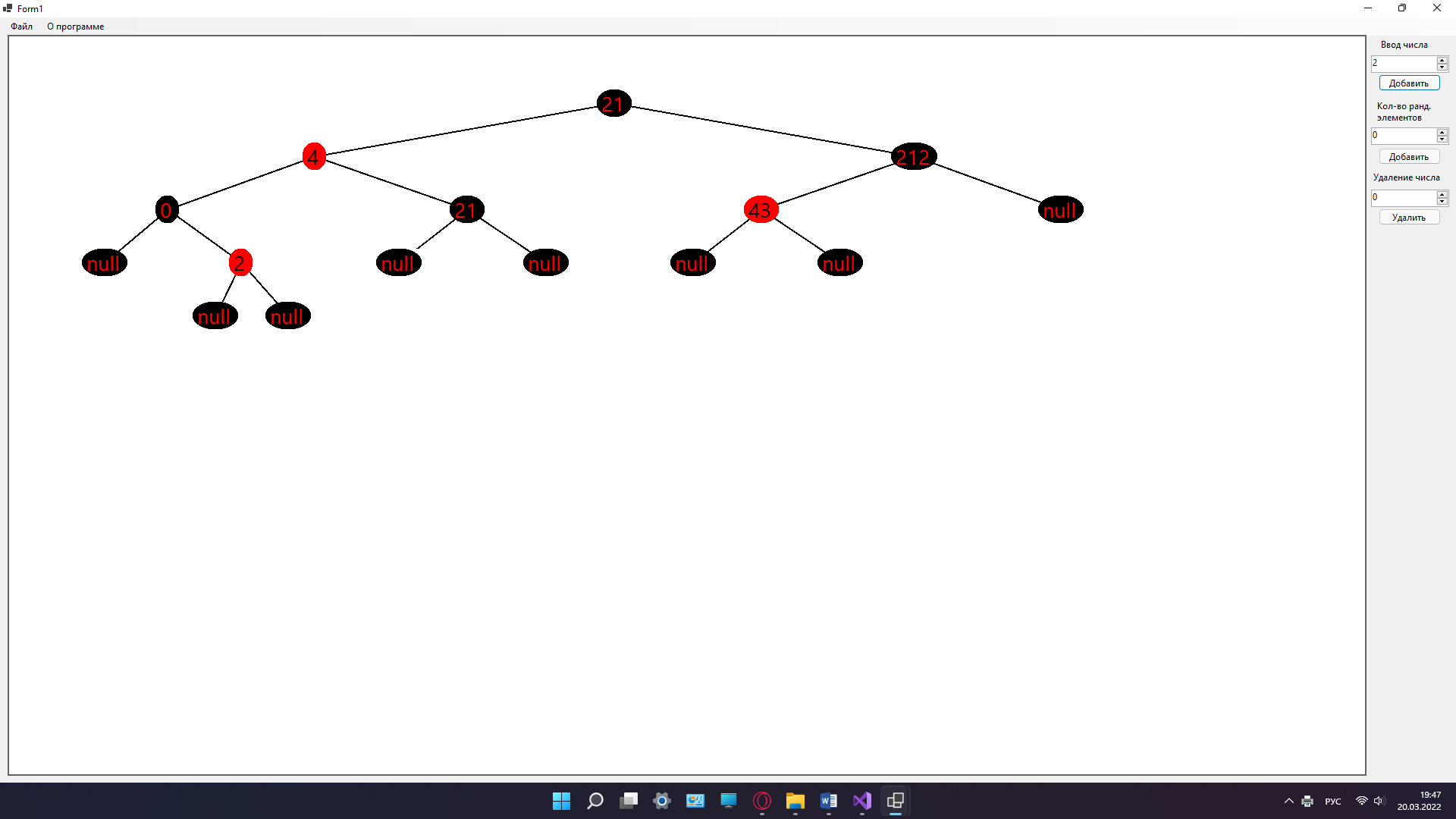


Рисунок 11 — Начало работы программы.

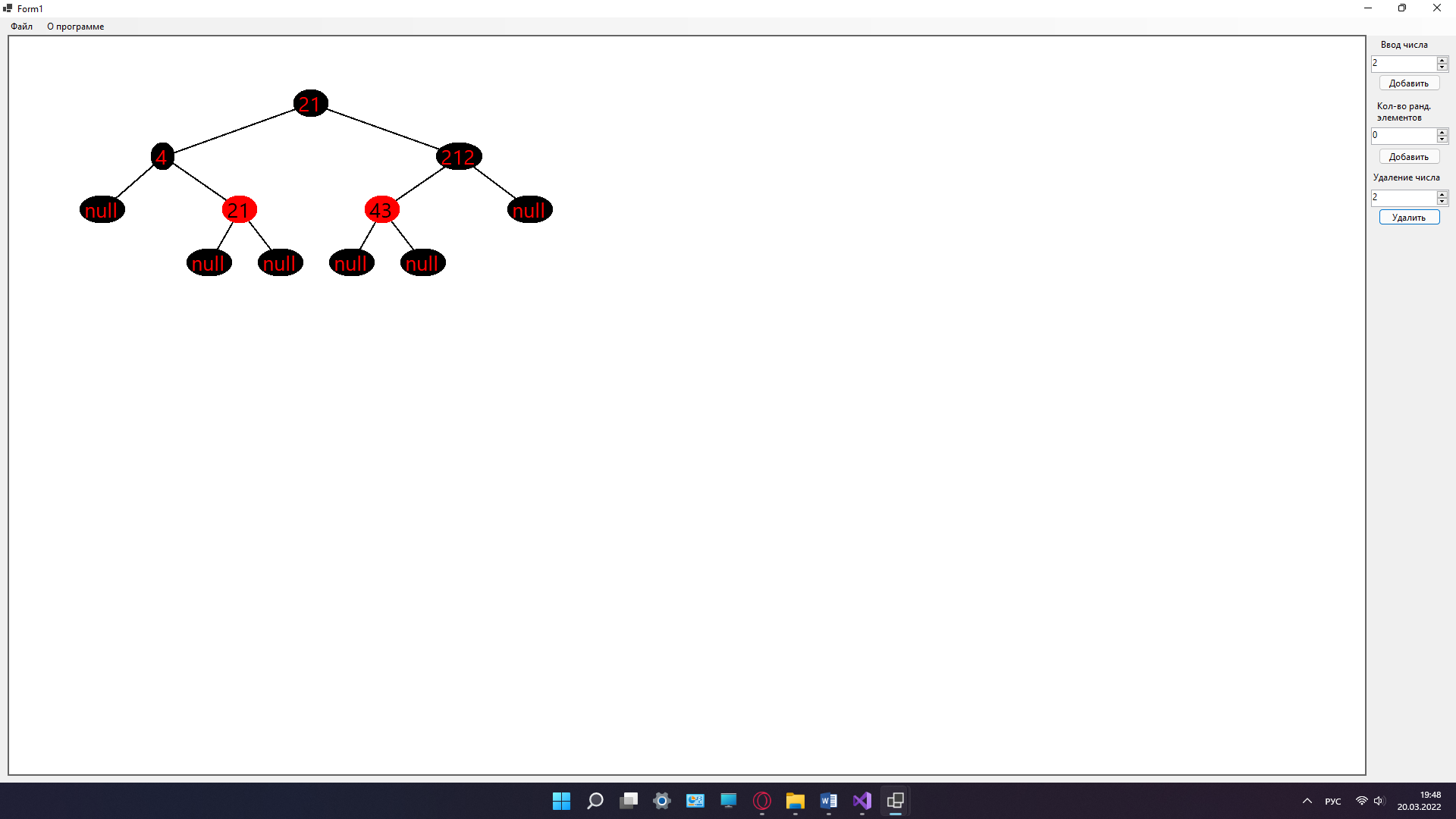


Рисунок 12 — Выбор варианта ввода узлов.

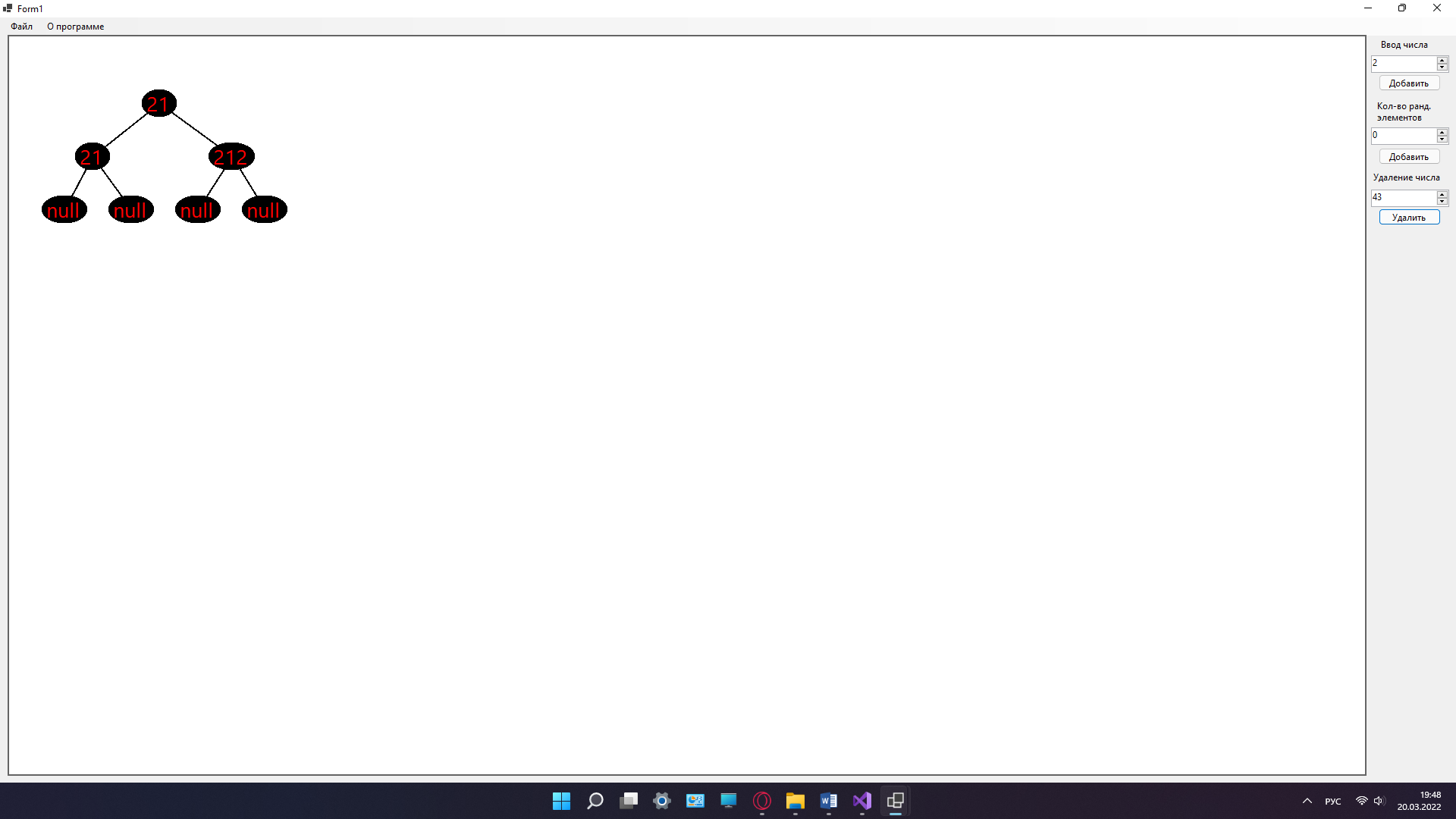


Рисунок 13 — Дерево, заполненное случайными числами.

# 9. Исходный текст программы

[Начало Node.cs ---]

using System;

namespace RedBlackTreeProject

{

class Node

{

public double Data { get; set; }

public bool IsRedColor { get; set; }

public bool IsRightChild { get; set; }

public Node? Parent { get; set; }

private Node? \_rightChild;

public Node? RightChild

{

get { return \_rightChild; }

set

{

\_rightChild = value;

if (\_rightChild != null)

{

\_rightChild.IsRightChild = true;

\_rightChild.Parent = this;

}

}

}

private Node? \_leftChild;

public Node? LeftChild

{

get { return \_leftChild; }

set

{

\_leftChild = value;

if (\_leftChild != null)

{

\_leftChild.IsRightChild = false;

\_leftChild.Parent = this;

}

}

}

public Node(double data, Node? parent)

{

Data = data;

IsRedColor = true;

Parent = parent;

RightChild = null;

LeftChild = null;

}

public Node? GetBrother()

{

if (Parent == null)

{

throw new InvalidOperationException("Root can't has brother");

}

if (IsRightChild)

{

return Parent.LeftChild;

}

else

{

return Parent.RightChild;

}

}

public bool IsBrotherColorRed()

{

if (this.Parent == null)

{

throw new InvalidOperationException("Root hasn't brother.");

}

Node? brother;

if (this.IsRightChild)

{

brother = Parent.LeftChild;

}

else

{

brother = Parent.RightChild;

}

if (brother == null)

{

return false;

}

return brother.IsRedColor;

}

}

}

[--- Конец Node.cs]

[Начало RedBlackTree.cs ---]

using System;

using System.Collections.Generic;

namespace RedBlackTreeProject

{

class RedBlackTree

{

private Node? \_root;

public RedBlackTree ()

{

}

public void AddData(double data)

{

if (\_root == null)

{

\_root = new Node (data, null);

\_root.Data = data;

\_root.IsRedColor = false;

return;

}

Node? node = \_root;

Node? parent;

bool isRightChild;

do

{

parent = node;

if (data > node.Data)

{

isRightChild = true;

node = node.RightChild;

}

else

{

isRightChild = false;

node = node.LeftChild;

}

}

while (node != null);

node = new Node (data, parent);

if (isRightChild)

{

parent.RightChild = node;

}

else

{

parent.LeftChild = node;

}

FixInsertNode(node);

while (\_root.Parent != null)

{

\_root = \_root.Parent;

}

}

public void DeleteData(double data)

{

Node? node = FindNode(data, \_root);

if (node == null)

{

return;

}

Node minimumRightNode = FindeMinimumRightNode(node);

if (minimumRightNode != node)

{

node.Data = minimumRightNode.Data;

node = minimumRightNode;

}

if (node.RightChild != null)

{

node.Data = node.RightChild.Data;

node = node.RightChild;

}

else if (node.LeftChild != null)

{

node.Data = node.LeftChild.Data;

node = node.LeftChild;

}

else if (node.Parent == null)

{

\_root = null;

return;

}

if (node.IsRedColor)

{

if (node.IsRightChild)

{

node.Parent!.RightChild = null;

}

else

{

node.Parent!.LeftChild = null;

}

}

else

{

/\* Node parent = node.Parent;

if (node.IsRightChild)

{

parent.RightChild = null;

}

else

{

parent.LeftChild = null;

}

if (parent == \_root)

{

return;

}\*/

//DeleteFixedUp(parent);

DeleteFixedUp(node);

if (node.IsRightChild)

{

node.Parent.RightChild = null;

}

else

{

node.Parent.LeftChild = null;

}

}

while (\_root.Parent != null)

{

\_root = \_root.Parent;

}

}

public List<List<double?[]>> GetTreeDataArray()

{

List<List<double?[]>> treeDataArray = new List<List<double?[]>>();

List<Node> foundTierNodes = new List<Node>();

if (\_root == null)

{

treeDataArray.Add(new List<double?[]>());

treeDataArray[0].Add(new double?[2]);

treeDataArray[0][0][0] = null;

treeDataArray[0][0][1] = 0;

}

else

{

treeDataArray.Add(new List<double?[]>());

treeDataArray[0].Add(new double?[2]);

treeDataArray[0][0][0] = \_root.Data;

treeDataArray[0][0][1] = Convert.ToInt32(\_root.IsRedColor);

foundTierNodes.Add(\_root);

}

while (foundTierNodes.Count != 0)

{

treeDataArray.Add(new List<double?[]>());

int indexLastTier = treeDataArray.Count - 1;

List<Node> newFoundTierNodes = new List<Node>();

for (int i = 0; i < foundTierNodes.Count; i++)

{

treeDataArray[indexLastTier].Add(new double?[2]);

if (foundTierNodes[i].LeftChild == null)

{

treeDataArray[indexLastTier][treeDataArray[indexLastTier].Count - 1][0] = null;

treeDataArray[indexLastTier][treeDataArray[indexLastTier].Count - 1][1] = 0;

}

else

{

treeDataArray[indexLastTier][treeDataArray[indexLastTier].Count - 1][0] = foundTierNodes[i].LeftChild.Data;

treeDataArray[indexLastTier][treeDataArray[indexLastTier].Count - 1][1] = Convert.ToInt32(foundTierNodes[i].LeftChild.IsRedColor);

newFoundTierNodes.Add(foundTierNodes[i].LeftChild);

}

treeDataArray[indexLastTier].Add(new double?[2]);

if (foundTierNodes[i].RightChild == null)

{

treeDataArray[indexLastTier][treeDataArray[indexLastTier].Count - 1][0] = null;

treeDataArray[indexLastTier][treeDataArray[indexLastTier].Count - 1][1] = 0;

}

else

{

treeDataArray[indexLastTier][treeDataArray[indexLastTier].Count - 1][0] = foundTierNodes[i].RightChild.Data;

treeDataArray[indexLastTier][treeDataArray[indexLastTier].Count - 1][1] = Convert.ToInt32(foundTierNodes[i].RightChild.IsRedColor);

newFoundTierNodes.Add(foundTierNodes[i].RightChild);

}

}

foundTierNodes = newFoundTierNodes;

}

return treeDataArray;

}

public List<double> GetDataArray()

{

List<double> dataAray = null;

if (\_root != null)

{

dataAray = new List<double>();

FindeAllNodeData(\_root, dataAray);

}

return dataAray;

}

private static void FixInsertNode(Node node)

{

Node parent = node.Parent!;

Node grandParent = parent.Parent!;

if (!parent.IsRedColor)

{

return;

}

if (parent.IsBrotherColorRed())

{

grandParent.RightChild!.IsRedColor = false;

grandParent.LeftChild!.IsRedColor = false;

if(grandParent.Parent != null)

{

grandParent.IsRedColor = true;

if (grandParent.Parent.IsRedColor)

{

FixInsertNode(grandParent);

}

}

}

else

{

if (parent.IsRightChild)

{

if (!node.IsRightChild)

{

RightRotation(parent);

Node? temp = node;

node = parent;

parent = temp;

}

LeftRotation(grandParent);

}

else

{

if (node.IsRightChild)

{

LeftRotation(parent);

Node? temp = node;

node = parent;

parent = temp;

}

RightRotation(grandParent);

}

parent.IsRedColor = false;

grandParent.IsRedColor = true;

node.IsRedColor = true;

}

}

private static void LeftRotation(Node node)

{

Node? parent = node.Parent;

Node rightChild = node.RightChild!;

if (parent != null)

{

if (node.IsRightChild)

{

parent.RightChild = rightChild;

}

else

{

parent.LeftChild = rightChild;

}

}

rightChild.Parent = node.Parent;

node.RightChild = rightChild.LeftChild;

rightChild.LeftChild = node;

}

private static void RightRotation(Node node)

{

Node? parent = node.Parent;

Node leftChild = node.LeftChild!;

if (parent != null)

{

if (node.IsRightChild)

{

parent.RightChild = leftChild;

}

else

{

parent.LeftChild = leftChild;

}

}

leftChild.Parent = node.Parent;

node.LeftChild = leftChild.RightChild;

leftChild.RightChild = node;

}

private static Node? FindNode(double data, Node? root)

{

if (root == null)

{

return null;

}

Node? node = root;

do

{

if (data == node.Data)

{

break;

}

else if (data > node.Data)

{

node = node.RightChild;

}

else

{

node = node.LeftChild;

}

if (node == null)

{

return null;

}

}

while (node != null);

return node;

}

private static Node FindeMinimumRightNode(Node node)

{

Node? minimumRightNode = node.RightChild;

if (minimumRightNode == null)

{

return node;

}

while (minimumRightNode.LeftChild != null)

{

minimumRightNode = minimumRightNode.LeftChild;

}

return minimumRightNode;

}

private void DeleteFixedUp(Node node)

{

Node parent = node.Parent!;

Node brother = node.GetBrother()!;

if (brother.IsRedColor)

{

if (node.IsRightChild)

{

parent.IsRedColor = true;

brother.IsRedColor = false;

RightRotation(parent);

parent = node.Parent!;

brother = node.GetBrother()!;

}

else

{

parent.IsRedColor = true;

brother.IsRedColor = false;

LeftRotation(parent);

parent = node.Parent!;

brother = node.GetBrother()!;

}

}

if ((brother.RightChild == null || !brother.RightChild.IsRedColor) &&

(brother.LeftChild == null || !brother.LeftChild.IsRedColor))

{

brother.IsRedColor = true;

if (parent.IsRedColor)

{

parent.IsRedColor = false;

}

else if (parent != \_root)

{

DeleteFixedUp(parent);

}

}

else

{

if (node.IsRightChild)

{

if (brother.LeftChild == null || !brother.LeftChild.IsRedColor)

{

LeftRotation(brother);

brother = node.GetBrother();

brother.IsRedColor = false;

brother.LeftChild.IsRedColor = true;

}

brother.IsRedColor = parent.IsRedColor;

parent.IsRedColor = false;

brother.LeftChild.IsRedColor = false;

RightRotation(parent);

}

else

{

if (brother.RightChild == null || !brother.RightChild.IsRedColor)

{

RightRotation(brother);

brother = node.GetBrother();

brother.IsRedColor = false;

brother.RightChild.IsRedColor = true;

}

brother.IsRedColor = parent.IsRedColor;

parent.IsRedColor = false;

brother.RightChild.IsRedColor = false;

LeftRotation(parent);

}

}

}

private void FindeAllNodeData(Node node, List<double> dataArray)

{

dataArray.Add(node.Data);

if (node.LeftChild != null)

{

FindeAllNodeData(node.LeftChild, dataArray);

}

if (node.RightChild != null)

{

FindeAllNodeData(node.RightChild, dataArray);

}

}

}

}

[--- Конец RedBlackTree.cs]

[Начало Program.cs ---]

using System;

using System.Windows.Forms;

using UI;

namespace Main

{

static class Program

{

/// <summary>

/// The main entry point for the application.

/// </summary>

[STAThread]

static void Main()

{

Console.WriteLine("Корниенко лапочка");

Application.SetHighDpiMode(HighDpiMode.SystemAware);

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

Application.Run(new Form1());

}

}

}

[--- Конец Program.cs]

[Начало Form1.cs ---]

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Drawing.Drawing2D;

using System.Windows.Forms;

using RedBlackTreeProject;

namespace UI

{

public partial class Form1 : Form

{

AboutForm \_aboutDialog;

Graphics \_graphics;

PictureBox \_pictureBox;

RedBlackTree \_tree;

const int \_DistanceBetweenTiers = 70;

const int \_NodeDiametr = 50;

List<List<Label>> \_associatedNodes;

bool \_isMouseDownToNode = false;

public Form1()

{

InitializeComponent();

\_associatedNodes = new List<List<Label>>();

\_tree = new RedBlackTree();

\_pictureBox = new PictureBox();

\_pictureBox.Dock = DockStyle.Fill;

\_pictureBox.BorderStyle = BorderStyle.FixedSingle;

\_pictureBox.BackColor = Color.White;

\_pictureBox.Resize += new EventHandler(pictureBox\_Resize);

panel.Controls.Add(\_pictureBox);

\_graphics = \_pictureBox.CreateGraphics();

\_aboutDialog = new AboutForm();

if (Properties.Settings.Default.isShowAboutMenu)

{

\_aboutDialog.Show();

}

}

private void DisplayedTree()

{

List<List<double?[]>> treeDataArray = \_tree.GetTreeDataArray();

int maxWidth = (2 \* Convert.ToInt32(Math.Pow(2, treeDataArray.Count - 1)) - 1) \* \_NodeDiametr;

List<Label> previosTier = new List<Label>();

previosTier.Add(new Label { Location = new Point(maxWidth, 0) });

for (int i = 0; i < treeDataArray.Count; i++)

{

IEnumerator<Label> iterator = previosTier.GetEnumerator();

iterator.MoveNext();

List<Label> newTier = new List<Label>();

int biasX = Convert.ToInt32(maxWidth / Math.Pow(2, i + 1) );

for (int j = 0; j < treeDataArray[i].Count; j++)

{

Label label = new Label();

newTier.Add(label);

label.AutoSize = true;

label.Font = new Font("Segoe UI", 20F, FontStyle.Regular, GraphicsUnit.Point);

label.MouseDown += new MouseEventHandler(this.Node\_MouseDown);

label.MouseUp += new MouseEventHandler(this.Node\_MouseUp);

label.MouseMove += new MouseEventHandler(this.Node\_MouseMove);

while (iterator.Current.Text == "null")

{

iterator.MoveNext();

}

if (i != 0 || j != 0)

{

\_associatedNodes.Add(new List<Label>());

\_associatedNodes[\_associatedNodes.Count - 1].Add(iterator.Current);

\_associatedNodes[\_associatedNodes.Count - 1].Add(label);

}

int y = iterator.Current.Location.Y + \_DistanceBetweenTiers;

int x = 0;

if (j % 2 == 0)

{

x = iterator.Current.Location.X - biasX;

}

else

{

x = iterator.Current.Location.X + biasX;

iterator.MoveNext();

}

label.Location = new Point(x, y);

if (treeDataArray[i][j][0].HasValue)

{

label.Text = Convert.ToString(treeDataArray[i][j][0]);

}

else

{

label.Text = "null";

}

if (treeDataArray[i][j][1] == 0)

{

label.ForeColor = Color.Red;

label.BackColor = Color.Black;

}

else

{

label.ForeColor = Color.Black;

label.BackColor = Color.Red;

}

panel.Controls.Add(label);

GraphicsPath path = new GraphicsPath();

path.AddEllipse(0, 0, label.Width, label.Height);

label.Region = new Region(path);

label.BringToFront();

}

previosTier = newTier;

}

}

private void DisplayedTreeAssociated()

{

foreach (List<Label> node in \_associatedNodes)

{

DrawLines(node[0], node[1], Color.Black);

}

}

private void Node\_MouseDown(object sender, MouseEventArgs e)

{

\_isMouseDownToNode = true;

}

private void Node\_MouseUp(object sender, MouseEventArgs e)

{

\_isMouseDownToNode = false;

}

private void Node\_MouseMove(object sender, MouseEventArgs e)

{

if (\_isMouseDownToNode)

{

Label movedNode = (Label)sender;

Point moveTo = PointToClient(Control.MousePosition);

moveTo.X -= movedNode.Width / 2;

moveTo.Y -= movedNode.Height / 2;

foreach (List<Label> node in \_associatedNodes)

{

if (node.Contains(movedNode))

{

DrawLines(node[0], node[1], Color.White);

}

}

movedNode.Location = moveTo;

foreach (List<Label> node in \_associatedNodes)

{

if (node.Contains(movedNode))

{

DrawLines(node[0], node[1], Color.Black);

}

}

}

}

private void DrawLines(Label node1, Label node2, Color color)

{

int x1 = node1.Location.X;

int y1 = node1.Location.Y;

int x2 = node2.Location.X;

int y2 = node2.Location.Y;

\_graphics.DrawLine(new Pen(color, 2), new Point(x1 + node1.Width / 2, y1 + node1.Height / 2),

new Point(x2 + node2.Width / 2, y2 + node2.Height / 2));

}

private void pictureBox\_Resize(object sender, EventArgs e)

{

\_graphics = \_pictureBox.CreateGraphics();

}

private void addButton\_Click(object sender, EventArgs e)

{

\_tree.AddData(Convert.ToDouble(addNumericUpDown.Value));

ClearGraphics();

DisplayedTree();

DisplayedTreeAssociated();

}

private void deleteButton\_Click(object sender, EventArgs e)

{

\_tree.DeleteData(Convert.ToDouble(deleteNumericUpDown.Value));

\_graphics.Clear(\_pictureBox.BackColor);

int countControls = panel.Controls.Count;

for (int i = 0; i < countControls; i++)

{

if (panel.Controls[i] is Label)

{

panel.Controls.RemoveAt(i);

i--;

countControls--;

}

}

\_associatedNodes.Clear();

DisplayedTree();

DisplayedTreeAssociated();

}

private void aboutProgramToolStripMenuItem\_Click(object sender, EventArgs e)

{

\_aboutDialog.ShowDialog();

}

private void ClearGraphics()

{

\_graphics.Clear(\_pictureBox.BackColor);

int countControls = panel.Controls.Count;

for (int i = 0; i < countControls; i++)

{

if (panel.Controls[i] is Label)

{

panel.Controls.RemoveAt(i);

i--;

countControls--;

}

}

\_associatedNodes.Clear();

}

private void openToolStripMenuItem\_Click(object sender, EventArgs e)

{

DialogResult result = openFileDialog.ShowDialog();

if (result == DialogResult.OK)

{

try

{

using System.IO.StreamReader sr = new System.IO.StreamReader(openFileDialog.OpenFile());

string inputValue;

while ((inputValue = sr.ReadLine()) != null)

{

double value = Convert.ToDouble(inputValue);

\_tree.AddData(value);

}

}

catch (System.IO.IOException)

{

MessageBox.Show("Файл не может быть прочитан.", "Ошибка!");

}

ClearGraphics();

DisplayedTree();

DisplayedTreeAssociated();

}

}

private void saveToolStripMenuItem1\_Click(object sender, EventArgs e)

{

DialogResult result = saveFileDialog.ShowDialog();

if (result == DialogResult.OK)

{

try

{

using System.IO.StreamWriter sw = new System.IO.StreamWriter(saveFileDialog.OpenFile());

List<double> dataArray = \_tree.GetDataArray();

foreach(double data in dataArray)

{

sw.WriteLine(data);

}

}

catch (System.IO.IOException)

{

MessageBox.Show("Не удалось сохранить данные.", "Ошибка!");

}

}

}

private void randAddButton\_Click(object sender, EventArgs e)

{

Random rand = new Random();

for (int i = 0; i < countRandNumericUpDown.Value; i++)

{

\_tree.AddData(rand.Next(-1000, 1000));

}

ClearGraphics();

DisplayedTree();

DisplayedTreeAssociated();

}

}

}

[--- Конец Form1.cs]

[Начало UnitTest1.cs ---]

using Microsoft.VisualStudio.TestTools.UnitTesting;

using System.Collections.Generic;

using RedBlackTreeProject;

namespace RedBlackTreeTest

{

[TestClass]

public class UnitTest1

{

[TestMethod]

public void InsertTest()

{

// Arrange

double[] insertValues = { 12, 32, 54, 23, 87, 43, 0, 98 };

double?[,] valuesInTree = new double?[17, 2] { {32, 0},

{12, 0}, {54, 1},

{0, 1}, {23, 1}, {43, 0}, {87, 0},

{null, 0}, {null, 0}, {null, 0}, {null, 0}, {null, 0}, {null, 0}, {null, 0}, {98, 1},

{null, 0}, {null, 0}};

// Act

RedBlackTree tree = new RedBlackTree();

foreach(double value in insertValues)

{

tree.AddData(value);

}

List<List<double?[]>> treeDataArray = tree.GetTreeDataArray();

// Assert

int indexValue = 0;

foreach (List<double?[]> itemsList in treeDataArray)

{

foreach (double?[] value in itemsList)

{

Assert.AreEqual(valuesInTree[indexValue, 0], value[0]);

Assert.AreEqual(valuesInTree[indexValue, 1], value[1]);

indexValue++;

}

}

}

[TestMethod]

public void DeleteTest()

{

// Arrange

double[] insertValues = { 12, 32, 54, 23, 87, 43, 0, 98 };

double deleteValue = 98;

double?[,] valuesInTree = new double?[15, 2] { {32, 0},

{12, 0}, {54, 1},

{0, 1}, {23, 1}, {43, 0}, {87, 0},

{null, 0}, {null, 0}, {null, 0}, {null, 0}, {null, 0}, {null, 0}, {null, 0}, {null, 0}};

// Act

RedBlackTree tree = new RedBlackTree();

foreach (double value in insertValues)

{

tree.AddData(value);

}

tree.DeleteData(deleteValue);

List<List<double?[]>> treeDataArray = tree.GetTreeDataArray();

// Assert

int indexValue = 0;

foreach (List<double?[]> itemsList in treeDataArray)

{

foreach (double?[] value in itemsList)

{

Assert.AreEqual(valuesInTree[indexValue, 0], value[0]);

Assert.AreEqual(valuesInTree[indexValue, 1], value[1]);

indexValue++;

}

}

}

}

}

[--- Конец UnitTest1.cs]