Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО

ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«ОРЕНБУРГСКИЙ ГОСУДАРСТВЕННЙ УНИВЕРСИТЕТ»

Факультет математики и информационных технологий

Кафедра компьютерной безопасности и математического обеспечения информационных систем

**РАСЧЕТНО-ГРАФИЧЕСКОЕ ЗАДАНИЕ**

по дисциплине «Технология создания прикладного программного обеспечения»

**Демонстрационная программа "Структурированные данные. Работа с деревьями"**

ОГУ 090301.65.3017.017 ОО

Руководитель

канд. техн. наук, доцент

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Влацкая И.В.

“\_\_\_\_\_”\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2017г.

Исполнитель

студент гр. 13КБ(с)РЗПО

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Чернова Е.В.

“\_\_\_\_\_”\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2017г.

Оренбург 2017

**Содержание**

[Введение 3](#_Toc469479369)

[1 Разработка модели демонстрационной программы "Структурированные данные. Работа с деревьями" 4](#_Toc469479370)

[1.1 Концептуальная модель демонстрационной программы "Структурированные данные. Работа с деревьями"](#_Toc469479370) 4

[1.2 Описание параметров качества 6](#_Toc469479370)

[2 Реализация демонстрационной программы"Структурированные данные. Работа с деревьями"](#_Toc469479370) 6

[[2.1 Организация модульной структуры программного средства 7](#_Toc469479370)](#_Toc469479370)

[2.2 Модули 8](#_Toc469479370)

[2.3 Метрики качества программногос редства 8](#_Toc469479370)

[3 Тестирование демонстрационной программы "Структурированные данные. Работа с деревьями" 9](#_Toc469479370)

[Заключение 14](#_Toc469479373)

[Список использованных источников 15](#_Toc469479374)

[Приложение А. Исходный код 16](#_Toc469479375)

# **Введение**

Структура данных – это совокупность элементов данных, между которыми существуют некоторые отношения, причем элементами данных могут быть как простые данные, так и структуры данных. Дерево — одна из наиболее широко распространённых структур данных в информатике, эмулирующая древовидную структуру в виде набора связанных узлов. Это абстрактный тип данных для хранения информационных элементов имеющих нелинейные отношения. Дерево определяется рекурсивно. Оно является либо пустым, либо состоит из узла (корня), являющегося родительским узлом для некоторого количества деревьев [1, 2].

При изучении студентами подобных структур данных у них могут возникать трудности. Ведь на практике деревья представляют в виде массивов или с помощью ссылок. Даже опытным студентам иногда сложно представить себе деревья с большим количеством элементов. Для лучшего понимания часто используют графическое представление деревьев, однако чаще всего рисуют вручную. Этим объясняется *актуальность* данной работы: необходима простая программа, способная быстро визуализировать деревья.

Таким образом, поставленная задача - создать программный продукт, демонстрирующий простые действия с деревьями поиска. Так же необходима справка, содержащая в себе описание демонстрируемых процессов и объясняющая основные принципы работы с программой.

***Целью*** *данной расчетно-графического задания является помощь студентам при изучении структур данных.*

# **1 Разработка модели демонстрационной программы "Структурированные данные. Работа с деревьями"**

**1.1 Концептуальная модель демонстрационной программы "Структурированные данные. Работа с деревьями"**

Программа предназначена для студентов, изучающих алгоритмы и структуры данных. Главной целью является улучшение понимания предмета студентов. Для этого будет реализовано графическое отображение демонстрируемых процессов.

Пользователь выбирает готовый шаблон дерева поиска, т.е. оно является входным параметром. Затем пользователь выбирает действие, которое хочет выполнить с деревом («добавить элемент в дерево», «поиск элемента по дереву», «удалить элемент дерева» и т.д.). В качестве выходного параметра будет модифицированное дерево и его графическое представление. Пользователь не ограничен в количестве действий с деревом.

Планируется, что программа будет работать с деревьями (ацикличными графами), рандомизированными и двоичными деревьями поиска и демонстрировать следующие простые операции:

* вставка нового элемента;
* удаление элемента;
* перебор всех элементов дерева (визуализация дерева);
* поиск элемента.

Это показано на диаграмме прецедентов (рис. 1).

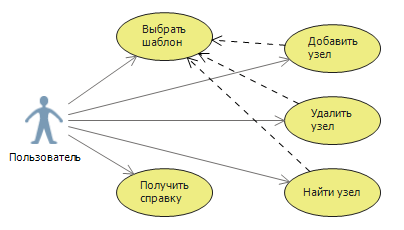


Рисунок 1 – Диаграмма прецедентов

Диаграмма активности показывает программный алгоритм как последовательность действий (рис. 2).

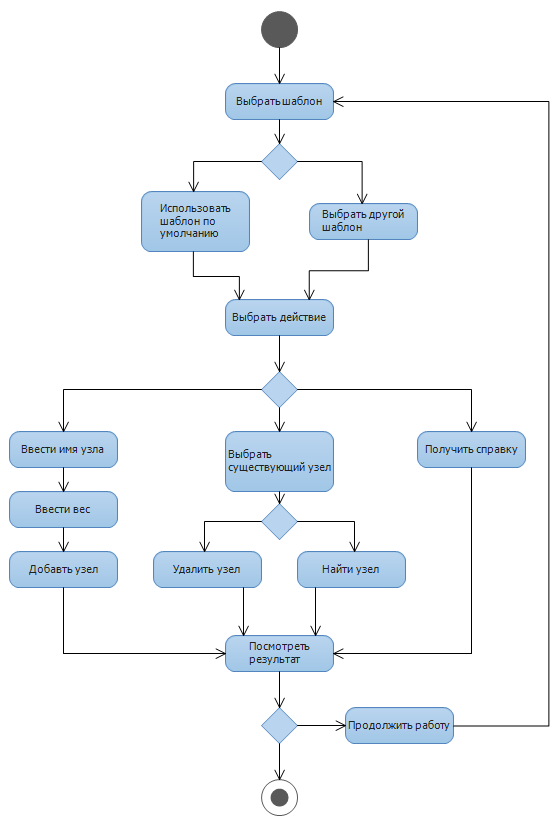


Рисунок 2 – Диаграмма активности

Диаграмма состояний показывает набор состояний системы и представляет собой конечный автомат.

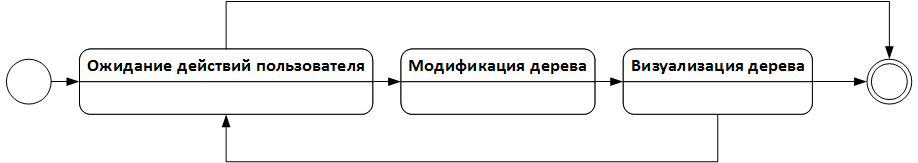


Рисунок 3 – Диаграмма состояний

Для поставленной задачи программист выбрал каскадную модель жизненного цикла потому, что:

* задача проста;
* требования окончательно согласованы;
* переход на очередную стадию проекта возможен только после полного завершения работ на предыдущей стадии.

В качестве метода программирования будет использоваться архитектурный подход, т.к. для данной предметной области можно выделить примитивные функции, которые будут использоваться несколько раз при решении задачи.

**1.2 Описание параметров качества**

Разрабатываемое программное средство обязательно должно обладать следующими параметрами качества [3]:

* пригодность для применения, чтобы программа выполняла объявленный набор функции и имела внешнее описание;
* надежность, чтобы программа работала безотказно с достаточно большой вероятностью и время выполнения не превышало разумных норм;
* эффективность для того, чтобы программа обеспечивала требуемую производительность решения функциональных задач с учетом количества используемых вычислительных ресурсов в установленных условиях.
* применимость означает, что разрабатываемая программа должна обладать легким, понятным и интуитивным интерфейсом.

Программа должна работать на OS Windows, требования к работе на других платформах не предъявляется. Поэтому параметр «переносимость», так же как и «сопровождаемость» в данном проекте не принимается в расчет.

**2 Реализация демонстрационной программы "Структурированные данные. Работа с деревьями"**

Для реализации проекта будет использоваться среда разработки Visual Studio 2013, язык программирования C# с использованием технологии WPF, а так же открытая библиотека GraphSharp для визуализации деревьев.

Архитектура программного обеспечения – цельная программа, поскольку задача в целом несложная, а функция программы ярко выраженна.

Итоговый продукт - программа, которая должна демонстрировать работу деревьев поиска.

**2.1 Организация структуры программного средства**

На рисунке 4 представлена SADT-диаграмма программного средства. Она показывает функциональную структуру программы, т.е. производимые им действия и связи между ними.

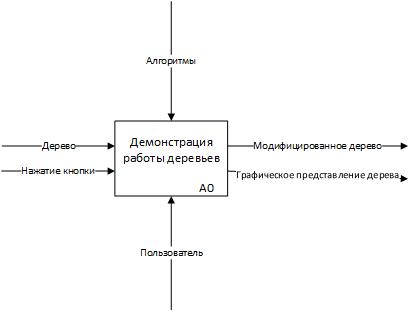


Рисунок 4 — SADT-диаграмма А0

Была произведена детализация программного средства (рис.5), которая отображает подфункции основного блока программы и движение информационных потоков между ними.

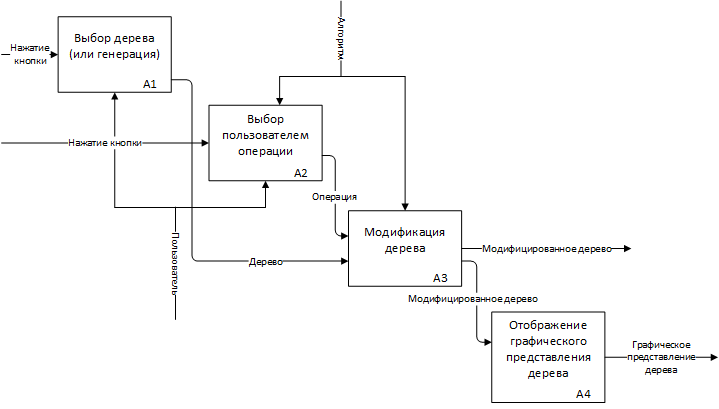


Рисунок 5 — Декомпозиция SADT-диаграммы А0

Рисунок 5 показывает наличие коммуникационной (4) и последовательной (5) связности модулей.

**2.2 Модули**

Программа содержит несколько модулей. Модуль работы с деревьями имеет функции модификации деревьев (удаление и добавление вершин), нахождение вершины по деревьям поиска, а так же простой интерфейс для взаимодействия с пользователем.

Модуль справки представляет собой окно, показывающее дополнительную информацию о работе программы.

Для визуализации деревьев программа дополнительно использует внешнюю библиотеку GraphSharp. Она включает в себя некоторые алгоритмы компоновки элемент управления GraphLayout для приложений WPF. Для корректной работы необходимы библиотеки QuickSharp, GraphSharp.Controls, WPFExtensions.

Таким образом, если рассматривать характеристики модулей по Майерсу, данные модули вызывают друг друга, поэтому их тип сцепления «По данным», а уровень значимости 1. Модули справки и внешних библиотек являются рутинными, поскольку результат их работы зависит только от значений параметров. Модуль работы с деревьями зависит от предыстории, это необходимо для правильного функционирования программы.

Функционально прочным модулем является модуль справки, он реализует одну определенную функцию. Информационно прочный модуль – модуль работы с деревьями, он работает с одной структурой данных и выполняет множество функций. Определить прочность внешних библиотек не представляется возможным.

**2.3 Метрики качества программного средства**

В общем случае применение метрик позволяет изучить сложность разработанного, оценить объем работ, стилистику разрабатываемой программы и усилия, потраченные для реализации того или иного решения.

Количественные метрики вычисляются на основе исходного кода программы. Метрика Холстеда показывает уровень качества программирования сложность понимания программы (таблица 1).

Таблица 1 – Расчет метрики Холстеда

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Название | n1 | n2 | n | N1 | N2 | N | V |
| Окно справки | 4 | 13 | 17 | 14 | 13 | 27 | 33,22 |
| Основное окно | 78 | 203 | 281 | 165 | 133 | 298 | 729,7 |

Метрика Джилба так же относится к количественным метрикам и показывает сложность программы (таблица 2).

Таблица 2 – Расчет метрики Джилба

|  |  |  |  |
| --- | --- | --- | --- |
| Название | CL | n | cl |
| Окно справки | 1 | 17 | 0,06 |
| Основное окно | 31 | 281 | 0,11 |

Метрики сложности потока управления программ основаны уже не на количественных показателях, а на анализе управляющего графа программы. Метрика МакКейба показывает требуемое количество проходов для покрытия всех контуров сильносвязанного графа или, другими словами, количество тестовых прогонов программы, необходимых для исчерпывающего тестирования по критерию «работает каждая ветвь». Для расчета метрики необходимо знать следующие параметры: количество вершин графа (функций) V=23, количество ребер E=64, число компонентов связности графа p=5; тогда цикломатическое число Маккейба Z=9. Метрика граничных значений определяется по формуле S0=1-(V-1)/Sa, где V – число вершин, Sa – абсолютная граничная сложность программы (в данном случае она равна 3). Число вершин уже известно, поэтому можно найти относительную граничную сложность программы, она равна7,3.

Метрика Чепина является мерой трудности понимания программ на основе входных и выходных данных и относится к метрикам сложности потока данных (таблица 3).

Таблица 3 – Расчет метрики Чепина

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Название | P | M | C | T | Q |
| Окно справка | 2 | 0 | 0 | 0 | 2 |
| Основное окно | 2 | 24 | 3 | 11 | 64,5 |

Рассчитав данные метрики, можно сделать вывод, что данное программное средство несколько сложное для понимания и, возможно, его можно сделать более простым.

**3 Тестирование демонстрационной программы "Структурированные данные. Работа с деревьями"**

Было проведено тестирование с целью проверить соответствие между ожидаемыми результатами работы программы и реальными. Сначала были установлены требования к тестированию (таблица 4).

Таблица 4 – Требования к тестированию

|  |  |  |
| --- | --- | --- |
| Название | Тип | Требование |
| Список шаблонов | ListBox | Обязательный параметр, отвечает за выбор шаблона, на основе которого будет проводится демонстрация |
| Значение новой вершины | TextBox | Параметр нужный при добавлении нового элемента в дерево |
| Вершины | ComboBox | Параметр, используемый для выполнения операций «Найти» и «Удалить» (в случае простого дерева еще и «Добавить») |

На основе требований был разработан универсальный шаблон тестов (таблица 5).

Таблица 5 – Разработанный шаблон тестов

|  |  |
| --- | --- |
| Действие | Ожидаемый результат |
| 1. Выбор шаблона из списка | * Отображение дерева шаблона * Заполнение полей «Количество вершин» и «Количество ребер» * Появление записи в поле «Пояснения» * Если выбран шаблон типа «Дерево», то кнопка «Найти неактивна» |
| 2. Выбор вершины из списка | * Отображение выбора в списке |
| 3. Нажатие кнопки «Найти» | * Появление записи в поле «Пояснения», в том числе пути от корневой до искомой вершины (если не выбран шаблон типа «Дерево», иначе никаких действий программы не должно быть) |
| 4. Нажатие кнопки «Удалить» | * Отображение измененного дерева * Изменение полей «Количество вершин» и «Количество ребер» * Появление записи в поле «Пояснения» |
| 5. Выбор вершины из списка | * Отображение выбора в списке |
| 6. Заполнение поля «Значение новой вершины» и нажатие кнопки «Добавить новую» | * Отображение измененного дерева * Изменение полей «Количество вершин» и «Количество ребер» * Появление записи в поле «Пояснения» |

В таблице 6 представлены примеры позитивных и негативных тест-кейсов для тестирования программы. Для шаблонов «Двоичное дерево поиска» и «Рандомизированное дерево поиска» подходит один тип тестов, для «Деревьев» были подобраны немного другие тесты в связи с особенностями программы.

Таблица 6 – Примеры разработанных тестов

|  |  |
| --- | --- |
| Действие | Ожидаемый результат |
| Позитивный тест-кейс | |
| 1. Выбор шаблона из списка  «Дерево» | * Отображение дерева шаблона * Заполнение полей «Количество вершин» и «Количество ребер» * Появление записи в поле «Пояснения» * Кнопка «Найти» неактивна |
| 2. Выбор вершины из списка: «i» | * Отображение выбора в списке |
| 3. Нажатие кнопки «Удалить» | * Отображение измененного дерева * Изменение полей «Количество вершин» и «Количество ребер» * Появление записи в поле «Пояснения» |
| 4. Выбор вершины из списка: «y» | * Отображение выбора в списке |
| 5. Заполнение поля «Значение новой вершины»: «test» и нажатие кнопки «Добавить новую | * Отображение измененного дерева * Изменение полей «Количество вершин» и «Количество ребер» * Появление записи в поле «Пояснения» |
| Позитивный тест-кейс | |
| 1. Выбор шаблона из списка  «Двоичное дерево поиска» | * Отображение дерева шаблона * Заполнение полей «Количество вершин» и «Количество ребер» * Появление записи в поле «Пояснения» * Кнопка «Найти» активна |
| 2. Выбор вершины из списка «qwe» | * Отображение выбора в списке |
| 3. Нажатие кнопки «Найти» | * Появление записи в поле «Пояснения», в том числе пути от корневой до искомой вершины |
| 4. Нажатие кнопки «Удалить» | * Отображение измененного дерева * Изменение полей «Количество вершин» и «Количество ребер» * Появление записи в поле «Пояснения» |
| 5. Выбор вершины из списка | * Отображение выбора в списке |
| 6. Заполнение поля «Значение новой вершины»: «test» и нажатие кнопки «Добавить новую» | * Отображение измененного дерева * Изменение полей «Количество вершин» и «Количество ребер» * Появление записи в поле «Пояснения» |
| Негативный тест-кейс | |
| 1. Выбор шаблона из списка  «Дерево» | * Отображение дерева шаблона * Заполнение полей «Количество вершин» и «Количество ребер» * Появление записи в поле «Пояснения» * Кнопка «Найти» неактивна |
| 2. Игнорирование выбора вершины из списка | * Программа не делает никаких действий |
| 3. Нажатие кнопки «Удалить» | * Сообщение об ошибке и просьба выбрать вершину |
| 5. Заполнение поля «Значение новой вершины»: «test» и нажатие кнопки «Добавить новую» | * Сообщение об ошибке и просьба выбрать вершину |
| 6. Выбор вершины из списка | * Отображение выбора в списке |
| 7. Игнорирование поля «Значение новой вершины»: и нажатие кнопки «Добавить новую» | * Сообщение об ошибке и просьба ввести значение в поле «Значение новой вершины» |
| Негативный тест-кейс | |
| 1. Выбор шаблона из списка  «Двоичное дерево поиска» | * Отображение дерева шаблона * Заполнение полей «Количество вершин» и «Количество ребер» * Появление записи в поле «Пояснения» * Кнопка «Найти» активна |
| 2. Игнорирование выборы вершины из списка | * Программа не делает никаких действий |
| 3. Нажатие кнопки «Найти» | * Сообщение об ошибке и просьба выбрать вершину |
| 4. Нажатие кнопки «Удалить» | * Сообщение об ошибке и просьба выбрать вершину |
| 6. Игнорирование поля «Значение новой вершины»: и нажатие кнопки «Добавить новую» | * Сообщение об ошибке и просьба ввести значение в поле «Значение новой вершины» |
| 7. Заполнение поля «Значение новой вершины» уже существующим значением: «qwe» и нажатие кнопки «Добавить новую» | * Сообщение об ошибке и просьба ввести новое уникальное для дерева значение в поле «Значение новой вершины» |

Тестирование показало, что пользователь не получает сообщения о невозможности добавления нового элемента в дерево в случае заполнения текстового поля «Значение новой вершины» пустой строкой. Данная ошибка была исправлена.

**Заключение**

Разработанное программное средство помогает студентам при изучении структур данных, а именно деревьев. Особенностью программы является визуализация как самих деревьев, так и действий совершаемых над ними (например, удаление или добавление вершин). Программа имеет свою справку с описанием всех процессов, происходящих с деревьями и кратким руководством пользования.

Для разработки данного программного средства были изучены научные статьи, посвященные проектированию и разработке программных средств. Были рассмотрены основные возможности Windows Presentation Foundation, такие как создание оконного приложения и удобного пользовательского интерфейса, корректно го отображения информации. Так как дополнительно для визуализации деревьев использовалась внешняя библиотека GraphSharp, были изучены ее основные функции.

Тестирование программного средства выявило небольшие ошибки, которые были исправлены.

В будущем планируется улучшение данного программного средства – доработка пользовательского интерфейса, ускорение обработки больших объемов данных, добавление нескольких новых функций, а так же устранение найденных при тестировании багов.

**Список использованных источников**

1. Дж. Макконелл Анализ алгоритмов. Активный обучающий подход. — 3-е дополненное издание. М: Техносфера, 2009. -416с.
2. Скиена С. Алгоритмы. Руководство по разработке. 2-е изд.: Пер. с англ. — СПб.: БХВ-Петербург. 2011. — 720 с.: ил.
3. Влацкая И. В., Заельская Н. А., Надточий Н. С. Проектирование и реализация прикладного программного обеспечения: учебное пособие //Оренбург: ОГУ. – 2015. – С. 11.
4. Дональд Э. Кнут. Глава 2.3. Деревья // Искусство программирования = The Art of Computer Programming. — 3-е изд. — М.: Вильямс, 2000. — Т. 2. Основные алгоритмы. — 832 с. — 7000 экз. — ISBN 5-8459-0081-6 (рус.) ISBN 0-201-89684-2 (англ.).
5. Майерс Г. Надежность программного обеспечения.- М.: Мир, 1980

# **Приложение А**

# **Исходный код**

public partial class Window1 : Window

{

public BidirectionalGraph<object, IEdge<object>> g;

Block tree = new Block();

private Help help=new Help();

public class Block

{

public string name = "null";

public int count = 0;

public Block left;

public Block right;

}

public Window1()

{

g = new BidirectionalGraph<object, IEdge<object>>();

DataContext = g;

InitializeComponent();

}

public void MakeListVertex()

{

cmVertex.Items.Clear();

foreach (var i in g.Vertices)

cmVertex.Items.Add(i);

}

public static int GetCount(Block t)

{

if (t != null)

return t.count;

else

return 0;

}

public static void PovL(ref Block t)

{

Block s = new Block();

s = t.left;

int c\_s = GetCount(t);

int c\_t = GetCount(s.right) + GetCount(t.right) + 1;

t.left = s.right;

s.right = t;

s.count = c\_s;

t.count = c\_t;

t = s;

}

public static void PovP(ref Block t)

{

Block s = new Block();

s = t.right;

int c\_s = GetCount(t);

int c\_t = GetCount(s.left) + GetCount(t.left) + 1;

t.right = s.left;

s.left = t;

s.count = c\_s;

t.count = c\_t;

t = s;

}

public static void AddRoot(ref Block t, string na)

{

if (t == null || t.count == 0)

{

t = new Block();

t.name = na;

t.left = null;

t.right = null;

t.count = 1;

}

else

{

if (String.Compare(t.name, na) >= 0)

{

AddRoot(ref t.left, na);

t.count++;

PovL(ref t);

}

else

{

AddRoot(ref t.right, na);

t.count++;

PovP(ref t);

}

}

}

public static void AddRand(ref Block t, string na)

{

Random r = new Random();

if (r.Next(GetCount(t) + 1) == 0)

AddRoot(ref t, na);

else

{

if (String.Compare(t.name, na) >= 0) Add(ref t.left, na);

else Add(ref t.right, na);

t.count++;

}

}

public static void Add(ref Block t, string na)

{

if (t == null || t.count == 0)

{

t = new Block();

t.name = na;

t.left = null;

t.right = null;

t.count = 1;

}

else

{

if (String.Compare(t.name, na) >= 0) Add(ref t.left, na);

else Add(ref t.right, na);

t.count++;

}

}

public void Obxod(Block t, ref int c,Block tparent = null)

{

if (t != null)

{

g.AddVertex(t.name);

cmVertex.Items.Add(t.name);

Obxod(t.left, ref c, t);

c++;

Obxod(t.right, ref c,t);

if (tparent != null)

g.AddEdge(new Edge<object>(tparent.name, t.name));

}

}

public static void Find(Block t, string na, ref string mes)

{

if (t == null)

//Console.WriteLine("Empty list");

mes += "Вершина не найдена";

else

{

mes += "";

if (string.Compare(t.name, na) == 0)

mes += t.name;

else if (string.Compare(t.name, na) > 0)

{

mes += string.Format("{0} -> ", t.name);

Find(t.left, na, ref mes);

}

else

{

mes += string.Format("{0} -> ", t.name);

Find(t.right, na, ref mes);

}

}

}

public static Block Join(Block lev, Block prav)

{

if (prav == null || prav.count == 0)

return lev;

if (lev == null || lev.count == 0)

return prav;

int c = GetCount(lev) + GetCount(prav);

Random r = new Random();

if (r.Next(c + 1) < GetCount(lev))

{

lev.right = Join(lev.right, prav);

lev.count = c;

return lev;

}

else

{

prav.left = Join(lev, prav.left);

prav.count = c;

return prav;

}

}

public static bool Del(ref Block t, string key)

{

if (t == null && t.count == 0)

return false;

else

{

if (String.Compare(key, t.name) == 0)

{

t = Join(t.left, t.right);

return true;

}

else if (String.Compare(key, t.name) > 0)

{

if (Del(ref t.right, key))

{

t.count--;

return true;

}

else

return false;

}

else

{

if (Del(ref t.left, key))

{

t.count--;

return true;

}

else

return false;

}

}

}

public void Update()

{

MakeListVertex();

tbEdges.Text = g.EdgeCount.ToString();

tbVertex.Text = g.VertexCount.ToString();

}

public void Rebuild()

{

int c = 0;

g = new BidirectionalGraph<object, IEdge<object>>();

Obxod(tree, ref c);

DataContext = g;

Update();

}

public bool Check()

{

foreach (var i in g.Vertices)

if (i.ToString() == tbNew.Text)

return false;

return true;

}

private void btNew\_Click(object sender, RoutedEventArgs e)

{

if (tbNew.Text != "")

{

tbNew.Text = tbNew.Text.Trim();

if (Check())

{

switch (lbList.SelectedIndex)

{

case 0:

case 1:

if (g.VertexCount >= 1)

if (cmVertex.SelectedIndex != -1)

{

g.AddVertex(tbNew.Text);

g.AddEdge(new Edge<object>(cmVertex.Text, tbNew.Text));

}

else

MessageBox.Show("Выберете вершину-родителя для новой вершины");

else

g.AddVertex(tbNew.Text);

tbOut.Text =

"Операция 'Добавление' в дереве.\n Простейшая операция, которая добавляет к дереву новую вершину-лист.";

break;

case 2:

case 3:

Add(ref tree, tbNew.Text);

Rebuild();

tbOut.Text =

"Операция 'Добавление' в двоичном дереве.\n Если дерево пустое, то добавляемая вершина становится корневой вершиной дерева. Иначе значение добавляемой вершины" +

" сравнивается с значением корневой вершины. В случае, когда значение добавляемой больше или равно корневой,вершина рекурсивно добавляется в правое поддерево. Когда меньше - в левое поддерево. ";

break;

case 4:

case 5:

AddRand(ref tree, tbNew.Text);

Rebuild();

tbOut.Text =

"Операция 'Добавление' в двоичном дереве.\n Если дерево пустое, то добавляемая вершина становится корневой вершиной дерева. Иначе значение добавляемой вершины сравнивается с значением корневой вершины. В случае, когда значение добавляемой больше или равно корневой,вершина рекурсивно добавляется в правое поддерево. Когда меньше - в левое поддерево. ";

break;

default:

Add(ref tree, tbNew.Text);

Rebuild();

break;

}

}

else

MessageBox.Show("Нельзя добавить вершину: уже существует вершина с таким значением");

Update();

}

else

MessageBox.Show("Введите значение новой вершины");

}

private void btDel\_Click(object sender, RoutedEventArgs e)

{

if (cmVertex.SelectedIndex != -1)

{ switch (lbList.SelectedIndex)

{

case 0:

case 1:

g.RemoveVertex(cmVertex.Text);

tbOut.Text =

"Операция 'Удаление' в дереве.\nВ данном случае демонстрируется удаление вершины и всех дуг, с которыми она соединена. Если вершина не является листом, возможно появление дополнительных деревьев или отдельных вершин.";

break;

case 2:

case 3:

case 4:

case 5:

tbOut.Text =

"Операция 'Удаление' в двоичном дереве.\nЕсли удалаемая вершина является листом дерева, то она просто удаляется из дерева.\nОднако чаще бывает так, что она имеет потомков. Если потомок один, то вершина заменяется потомком (точнее заменяется поддеревом, корнем которого является потомок). Если потомков два, то выбирается либо на самый левый узел его правого поддерева, причем они должны иметь не более одного потомка.";

if(!Del(ref tree, cmVertex.Text))

tbOut.Text += "Не получилось";

Rebuild();

break;

default:

break;

}

Update();

}

else

MessageBox.Show("Выберете вершину для удаления");

}

private void btFind\_Click(object sender, RoutedEventArgs e)

{

if (cmVertex.SelectedIndex != -1)

switch (lbList.SelectedIndex)

{

case 2:

case 3:

case 4:

case 5:

string mes = null;

Find(tree, cmVertex.Text,ref mes);

tbOut.Text = String.Format("Операция 'Найти' проверяет наличие вершины в двоичном дереве поиска.\nЕсли дерево не пусто, сравниваются значения корневого узла и искомого. Если они равны, то вершина найдена. Если значение искомого узла больше корневого, то вершина рекурсивно ищется в правом поддереве.Если меньше,то в левом поддереве.\nИскомый узел: {0}.\nПуть от корневой вершины к искомой: {1}.", cmVertex.Text, mes);

break;

default:

break;

}

else

MessageBox.Show("Выберете вершину для поиска");

}

private void lbList\_OnSelectionChanged(object sender, SelectionChangedEventArgs e)

{

g = new BidirectionalGraph<object, IEdge<object>>();

switch (lbList.SelectedIndex)

{

case 1:

tbOut.Text = "Пустое дерево для ваших экспериментов :)";

Update();

break;

case 0:

btFind.IsEnabled = false;

g.AddVertex("p");

g.AddVertex("r");

g.AddVertex("i");

g.AddVertex("v");

g.AddVertex("e");

g.AddVertex("t");

g.AddVertex("y");

g.AddVertex("o");

g.AddVertex("u");

g.AddEdge(new Edge<object>("p", "r"));

g.AddEdge(new Edge<object>("p", "i"));

g.AddEdge(new Edge<object>("r", "v"));

g.AddEdge(new Edge<object>("r", "e"));

g.AddEdge(new Edge<object>("i", "t"));

g.AddEdge(new Edge<object>("e", "y"));

g.AddEdge(new Edge<object>("e", "o"));

g.AddEdge(new Edge<object>("e", "u"));

tbOut.Text = "Дерево - структура данных, эмулирующая древовидную структуру, является связным графом, не содержащим циклы. Элемент дерева, не имеющий родителя является корнем (в данном примере вершина p). Элемент, не имеющий поддеревьев, называется листом или терминальной вершиной (v,t,y,o,u). Внутренний узел — любой узел дерева, имеющий потомков и не являющийся листовым узлом.";

Update();

break;

case 2:

tree = null;

btFind.IsEnabled = true;

Add(ref tree, "j");

Add(ref tree, "a");

Add(ref tree, "v");

Add(ref tree, "t");

Add(ref tree, "z");

Add(ref tree, "b");

Rebuild();

tbOut.Text = "Двоичное дерево представляет собой структуру, в которой каждая вершина имеет не более двух вершин-потомков и ровно одного родителя.";

break;

case 4:

tree = null;

btFind.IsEnabled = true;

AddRand(ref tree, "qwe");

AddRand(ref tree, "assd");

AddRand(ref tree, "z");

AddRand(ref tree, "zx");

AddRand(ref tree, "t");

AddRand(ref tree, "n");

Rebuild();

tbOut.Text = "Рандомизированные двоичные деревья поиска - это двоичные деревья поиска с добавлением случайности при генерации дерева.\nЕсли добавлять в дерево отсортированные элементы может получиться наименее сбалансированное дерево. Необходимо упорядочить их случайным образом, тогда вероятность получить дерево близкое к сбалансированному выше. Поэтому в рандомизированные деревья новые элементы добавляются случайно либо в лист, либо в корень.";

break;

case 5:

case 3:

tree = null;

tbOut.Text = "Пустое дерево для ваших экспериментов :)";

Rebuild();

break;

default:

tree = null;

break;

}

DataContext = g;

}

private void btS\_Click(object sender, RoutedEventArgs e)

{

help = new Help();

help.Show();

}

private void Window1\_OnClosed(object sender, EventArgs e)

{

help.Close();

}

}