**Министерство науки и высшего образования Российской Федерации**

**федеральное государственное образовательное бюджетное учреждение высшего образования**

**«Вологодский государственный университет»**

**(ВоГУ)**

**РАСЧЁТНО-ГРАФИЧЕСКАЯ РАБОТА**

**по дисциплине «Алгоритмы и структуры данных»**

|  |  |
| --- | --- |
| **НА ТЕМУ:** | разработка визуализатора алгоритма |
| «Декартово дерево» | |

**Направление подготовки: 09.03.01 Информатика и вычислительная техника**

**Направленность (профиль): Программное обеспечение средств вычислительной техники и автоматизированных систем**

**Форма обучения: \_\_\_ очная \_\_\_\_**

***(очная / заочная)***

**Институт: Математики, естественных и компьютерных наук**

***(указывается наименование института, к которому относится данное направление подготовки / специальность)***

**Кафедра: Автоматики и вычислительной техники**

***(указывается наименование кафедры, обеспечивающей подготовку)***

**Группа: 4Б09 ПО-31**

***(аббревиатура)***

**Студент: Пихтина Е.М.**

***(ФИО)***

**Преподаватель: Андрианов И.А.**

***(ФИО)***

**Вологда**

**2021 г.**

Оглавление

1. Введение3
2. Marge и Split 4
3. Операции с деревом8
4. Разработка визуализатора10
   1. Проектирование интерфейса пользователя 10
   2. Разработка алгоритмов и структур данных для выполнения пошаговой визуализации.11
   3. Тестирование визуализатора13
5. Заключение19
6. **Введение**

Итак, у нас есть данные дерева — ключи x (здесь и далее предполагается, что ключ и является той самой информацией, которую мы храним в дереве; когда впоследствии потребуется отделять по смыслу пользовательскую информацию от ключей, я скажу особо). Давайте добавим к ним еще один параметр в пару — y, и назовем его приоритетом. Теперь построим такое дерево, которое хранит в каждой вершине по два параметра, и при этом по ключам является деревом поиска, а по приоритетам — кучей. Такое дерево и будем далее называть декартовым.

Почему дерево называется декартовым? Это сразу станет ясно, как только мы попробуем его нарисовать. Возьмем какой-нибудь набор пар «ключ-приоритет» и расставим на координатной сетке соответствующие точки (x, y). А потом соединим соответствующие вершины линиями, образуя дерево. Таким образом, декартово дерево отлично укладывается на плоскости благодаря своим ограничениям, а два его основных параметра — ключ и приоритет — в некотором смысле, координаты. Результат построения показан на рисунке: слева в стандартной нотации дерева, справа — на декартовой плоскости.

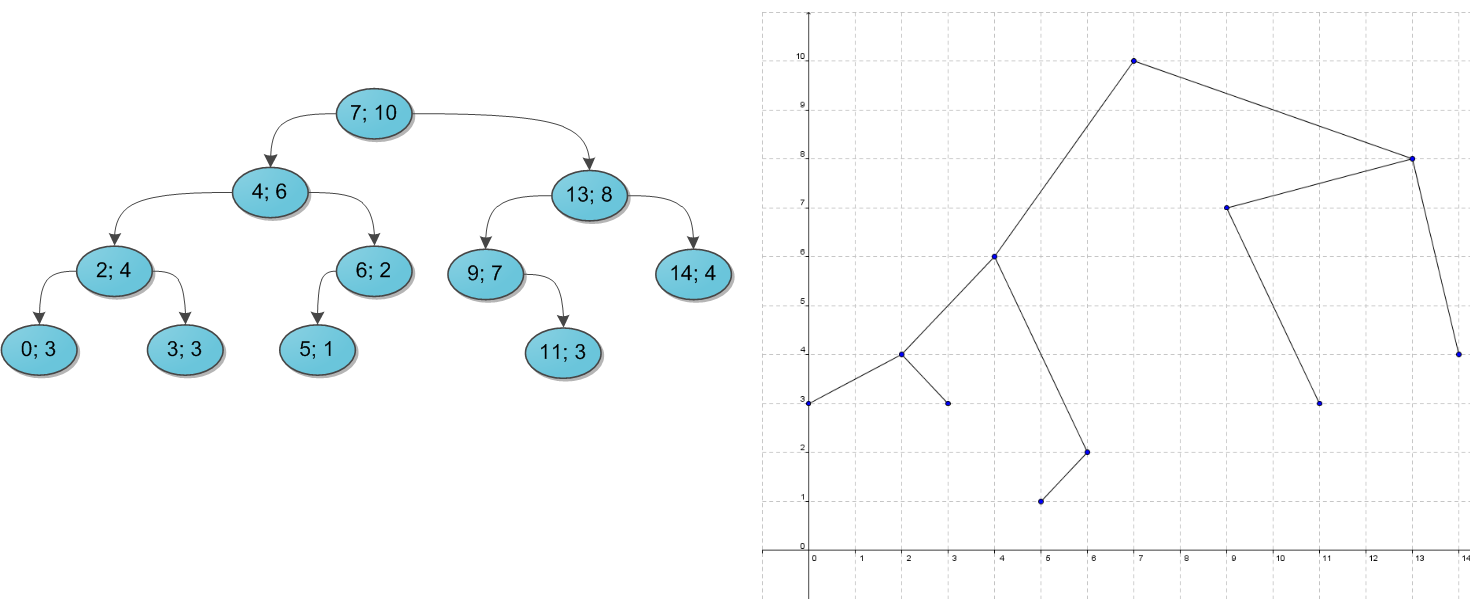


Рисунок 1. Декартово дерево

Пока что не очень понятно, зачем такое нужно. А разгадка проста, и кроется она в следующих утверждениях. Во-первых, пусть дано множество ключей: корректных деревьев поиска из них можно построить много различных, в том числе и спископодобное. А вот после добавления к ним приоритетов дерево из данных ключей можно построить уже лишь одно-единственное, вне зависимости от порядка поступления ключей. Это довольно очевидно.

А во-вторых, давайте теперь сделаем наши приоритеты случайными. То есть просто ассоциируем с каждым ключом случайное число из достаточно большого диапазона, и именно оно и будет служить соответствующим игреком. Тогда полученное декартово дерево с очень высокой, стремящейся к 100% вероятностью, будет иметь высоту, не превосходящую 4 log2 N. (Оставлю этот факт здесь без доказательства.) А значит, хоть оно может и не быть идеально сбалансированным, время поиска ключа в таком дереве все равно будет порядка O(log2 N), чего мы, собственно, и добивались.

1. **Marge и Split**

Вся подноготная работы с декартовым деревом заключается в двух основных операциях: Merge и Split. С помощью них элементарно выражаются все остальные популярные операции, так что начнем с основ.

Операция Merge принимает на вход два декартовых дерева L и R. От нее требуется слить их в одно, тоже корректное, декартово дерево T. Следует заметить, что работать операция Merge может не с любыми парами деревьев, а только с теми, у которых все ключи одного дерева ( L ) не превышают ключей второго ( R ).

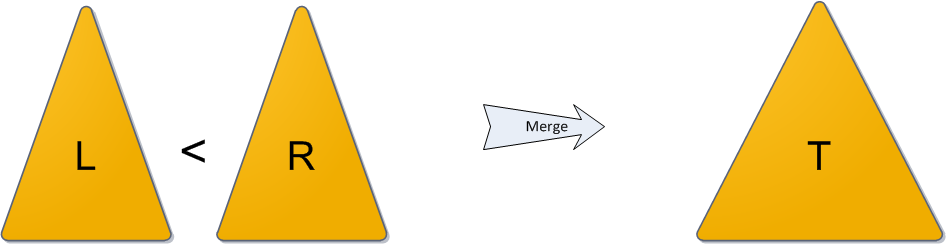


Рисунок 2. Операция Merge

Алгоритм работы Merge очень прост. Корнем будущего дерева будет элемент с наибольшим приоритетом. Кандидатов на максимальный приоритет у нас два — только корни двух исходных деревьев. Сравним их приоритеты; пускай для однозначности приоритет y левого корня больше, а ключ в нем равен x. Новый корень определен, теперь стоит подумать, какие же элементы окажутся в его правом поддереве, а какие — в левом.

Легко понять, что все дерево R окажется в правом поддереве нового корня, ведь ключи-то у него больше x по условию. Точно так же левое поддерево старого корня L.Left имеет все ключи, меньшие x, и должно остаться левым поддеревом, а правое поддерево L.Right просто рекурсивно вызываем Merge для L.Right и дерева R, и возвращенное ею дерево используем как новое правое поддерево.

На рисунке синим цветом показано правое поддерево результирующего дерева после операции Merge и связь от нового корня к этому поддереву.

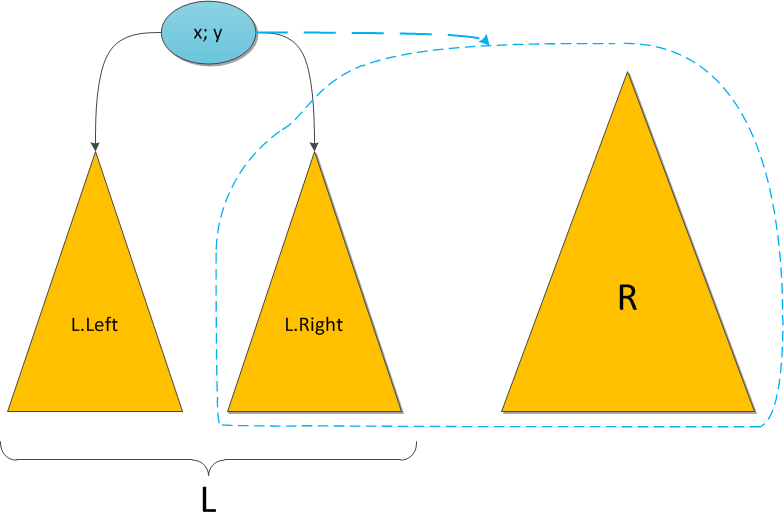
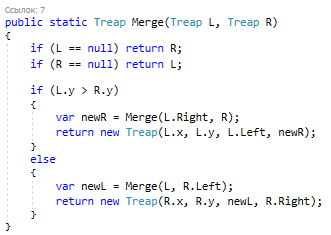


Рисунок 3. Операция Merge для правого поддерева

Симметричный случай — когда приоритет в корне дерева R выше — разбирается аналогично. И, конечно, надо не забыть про основу рекурсии, которая в нашем случае наступает, если какое-то из деревьев L и R, или сразу оба, являются пустыми.

Исходный код Merge:



Теперь об операции Split. На вход ей поступает корректное декартово дерево T и некий ключ x0. Задача операции — разделить дерево на два так, чтобы в одном из них ( L ) оказались все элементы исходного дерева с ключами, меньшими x0, а в другом ( R ) — с большими. Никаких особых ограничений на дерево не накладывается.

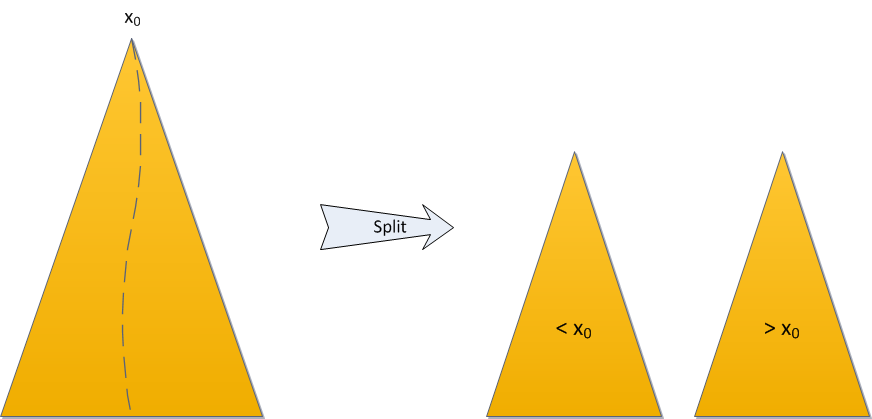


Рисунок 4. Операция Split

Можно сразу сказать, что все элементы левого поддерева T также окажутся в L — их ключи ведь тоже все будут меньше x0. Более того, корень T будет и корнем L, поскольку его приоритет наибольший во всем дереве. Левое поддерево корня полностью сохранится без изменений, а вот правое уменьшится — из него придется убрать элементы с ключами, большими x0, и вынести в дерево R. А остаток ключей сохранить как новое правое поддерево L. Снова видим идентичную задачу, снова напрашивается рекурсия.

Возьмем правое поддерево и рекурсивно разрежем его по тому же ключу x0 на два дерева L' и R'. После чего становится ясно, что L' станет новым правым поддеревом дерева L, а R' и есть непосредственно дерево R — оно состоит из тех и только тех элементов, которые больше x0.

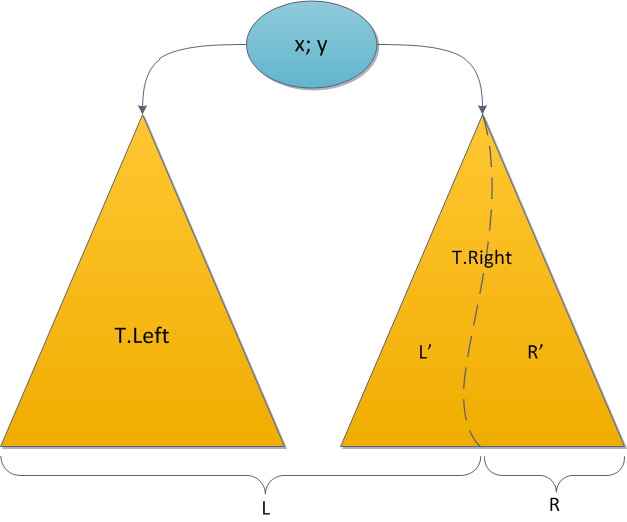
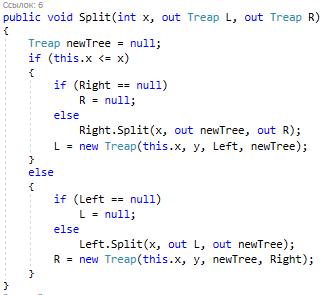


Рисунок 5. Split для правого поддерева.

Симметричный случай, при котором ключ корня больше, чем x0, тоже совершенно идентичен. Основа рекурсии здесь — случаи, когда какое-то из поддеревьев пустое. Исходный код функции.



Кстати, обратите внимание: деревья, выдаваемые на выход операцией Split, подходят как входные данные для операции Merge: все ключи левого дерева не превосходят ключей в правом. Это ценное обстоятельство пригодится нам уже через несколько абзацев.

Последний вопрос — это время работы Merge и Split. Из описания алгоритма видно, что Merge за каждую итерацию рекурсии уменьшает суммарную высоту двух сливаемых деревьев как минимум на единицу, так что общее время работы не превосходит 2H, то есть O(H). А со Split все совсем просто — мы работаем с единственным деревом, его высота уменьшается с каждой итерацией тоже как минимум на единицу, и ассимптотика работы операции тоже O(H). А поскольку декартово дерево со случайными приоритетами, как уже говорилось, с высокой вероятностью имеет близкую к логарифмической высоту, то Merge и Split работают за желаемый O(log2 N), и это дает нам потрясающий простор для их применения.

1. **Операции с деревом**

Теперь, когда мы с вами в совершенстве владеем клеем и ножницами, не составляет совершенно никакого труда только с помощью них реализовать самые необходимые действия с декартовым деревом: добавление элемента в дерево и удаление его. Я приведу самый простой вариант их реализации, основанный целиком на Merge и Split. Он будет работать за все то же логарифмическое время, однако отличаться, как говорят ACM-олимпийцы, большей константой: то есть порядок зависимости времени работы от размера дерева будет все так же O(log2 N), но точное время работы отличаться в несколько раз — в константу раз. Скажем, 4 log2 N против просто log2 N. На практике это различие почти не ощущается, пока размер дерева не достигнет поистине галактических размеров.

Итак, пускай нам дано декартово дерево и некий элемент x, который требуется в него вставить (как мы помним, в контексте статьи предполагается, что все элементы различны и икса в дереве еще нет). Хочется применить подход из двоичного дерева поиска: идти вниз по ключам, выбирая каждый раз путь влево или вправо, пока не найдем место, куда можно вставить наш x, и дописать его. Но это решение неправильное, ведь мы забыли о приоритетах. Место, куда алгоритм дерева поиска захочет добавить новую вершину, однозначно удовлетворяет ограничениям дерева поиска по x, однако может нарушить ограничение кучи по y. Значит, придется действовать немного выше и абстрактней.

Второй вариант решения — представить новый ключ как дерево из единственной вершины (со случайным приоритетом y), и слить его с исходным с помощью Merge. Это опять неверно: в исходном дереве могут быть вершины с ключами, большими x, и тогда мы нарушаем обещание, данное функции Merge касательно взаимоотношения между ее входными деревьями.

Проблему можно исправить. Помня универсальность операций Split/Merge, решение напрашивается практически сразу:

* Разделим (split) дерево по ключу x на дерево L, с ключами меньше икса, и дерево R, с большими.
* Создадим из данного ключа дерево M из единственной вершины (x, y), где y — только что сгенерированный случайный приоритет.
* Объединим (merge) по очереди L с M, то что получилось — с R.
* Все шаги алгоритма можно и проиллюстрировать.

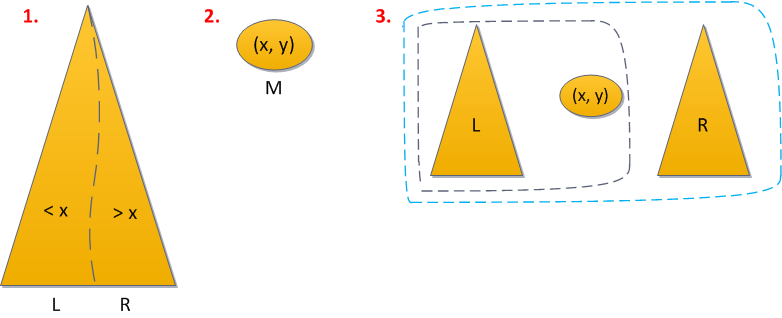
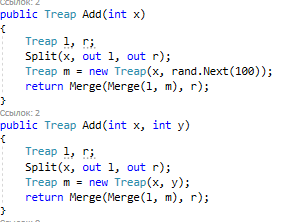


Рисунок 6. Работа метода Add.

Исходный код:



1. **Разработка визуализатора**

Для разработки визуализатора был выбран язык программирования C#, так как это наиболее изученный мною языков. Данный язык программирования имеет необходимые качества для создания программы визуализации, а именно:

• Удобство создания GUI

• Функциональная полнота

• Кроссплатформенность

Само приложение будет написано на Windows Presentation Foundation. Windows Presentation Foundation (WPF) — аналог WinForms, система для построения клиентских приложений Windows с визуально привлекательными возможностями взаимодействия с пользователем, графическая (презентационная) подсистема в составе. NET Framework (начиная с версии 3.0), использующая язык XAML.

* 1. **Проектирование интерфейса пользователя**

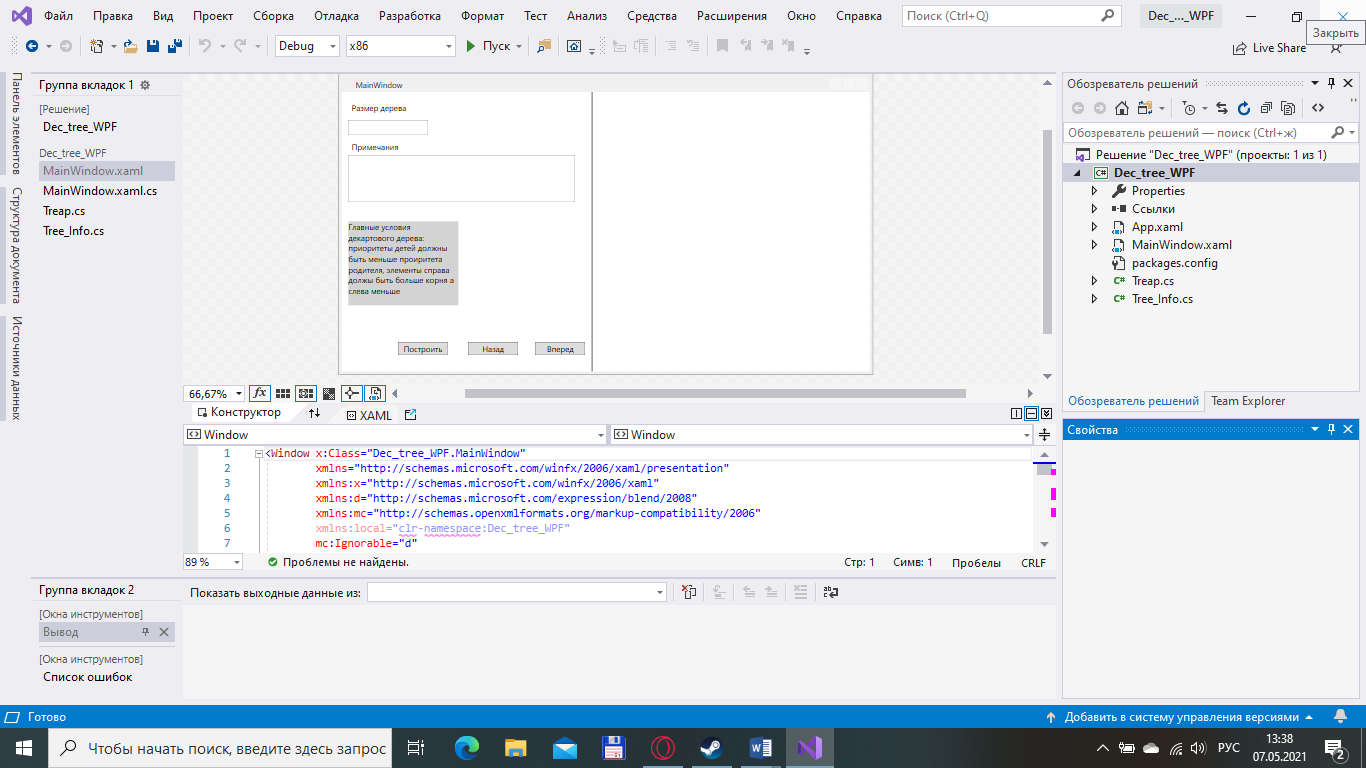


Рисунок 7. Разработка интерфейса в среде разработки

Интерфейс программы получился следующим.

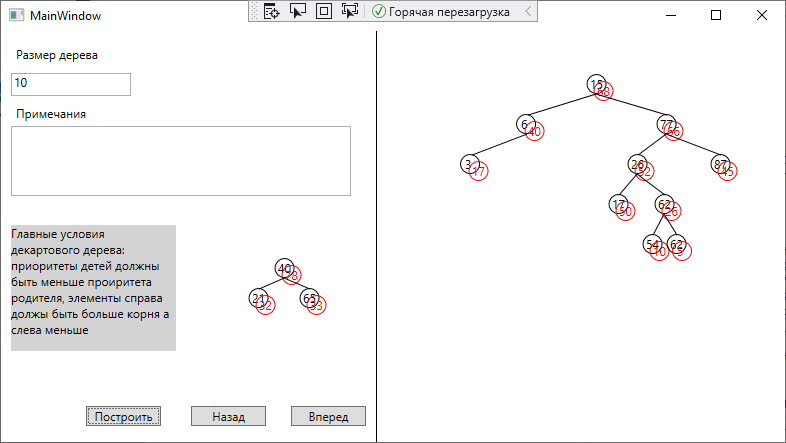


Рисунок 8. Интерфейс программы

Здесь мы видим поле для ввода размера дерева, поле с примечаниями, условия построения дерева, его пример, кнопку построения дерева, и кнопки для шагов вперед и назад.

* 1. **Разработка алгоритмов и структур данных для выполнения пошаговой визуализации.**

Для визуализации был разработан класс Tree\_info в котором хранится вся информация о дереве для пошаговой визуализации.

class Tree\_Info

    {

        public int size = 0;

        public List<(int, int)> elements\_on\_lvl = new List<(int, int)>();

        public List<(int, int)> elements\_priority = new List<(int, int)>();

        public List<List<(int, int)>> drawed\_elements = new List<List<(int, int)>>();

        public Treap treap;

        public int lvl = 0;

        public int number\_slide = -1;

        public STEPS step = STEPS.ONE;

        public List<List<UIElement>> previous = new List<List<UIElement>>();

        public List<string> comments = new List<string>();

        public enum STEPS

        {

            ZERO,

            ONE,

            TWO,

        }

        public void Reset()

        {

            step = STEPS.ZERO;

            elements\_on\_lvl.Clear();

            elements\_priority.Clear();

            drawed\_elements.Clear();

            drawed\_elements = new List<List<(int, int)>>();

            previous.Clear();

            comments.Clear();

            treap = null;

            lvl = 0;

            number\_slide = -1;

        }

        public List<(int, int)> Get\_Element()

        {

            List<(int, int)> result = new List<(int, int)>();

            if (elements\_on\_lvl[lvl].Item1 != 0)

            {

                result = drawed\_elements[lvl];

            }

            return result;

        }

    }

Визуализация осуществляется следующим кодом. Для шага вперед

public void Next\_Step(object sender, RoutedEventArgs e)

        {

            if (info.number\_slide == -1 || info.number\_slide + 1 == info.previous.Count)

            {

                if (info.elements\_priority.Count == 0 && info.step == Tree\_Info.STEPS.ONE)

                    return;

                canvas.Children.Clear();

                if (info.step == Tree\_Info.STEPS.TWO)

                {

                    info.step = Tree\_Info.STEPS.ZERO;

                }

                info.step++;

                Draw\_Step(ref info.elements\_priority);

            }

            else

            {

                canvas.Children.Clear();

                info.number\_slide++;

                foreach(UIElement elem in info.previous[info.number\_slide])

                {

                    canvas.Children.Add(elem);

                }

                note.Text = info.comments[info.number\_slide];

            }

        }

И для шага назад:

public void Previous\_Step(object sender, RoutedEventArgs e)

        {

            if(info.number\_slide > 0)

            {

                canvas.Children.Clear();

                info.number\_slide--;

                foreach (UIElement elem in info.previous[info.number\_slide])

                {

                    canvas.Children.Add(elem);

                }

                note.Text = info.comments[info.number\_slide];

            }

        }

* 1. **Тестирование визуализатора**

Если попяться построить, не введя его размер то будет выведено предупреждение в разделе «Примечания».

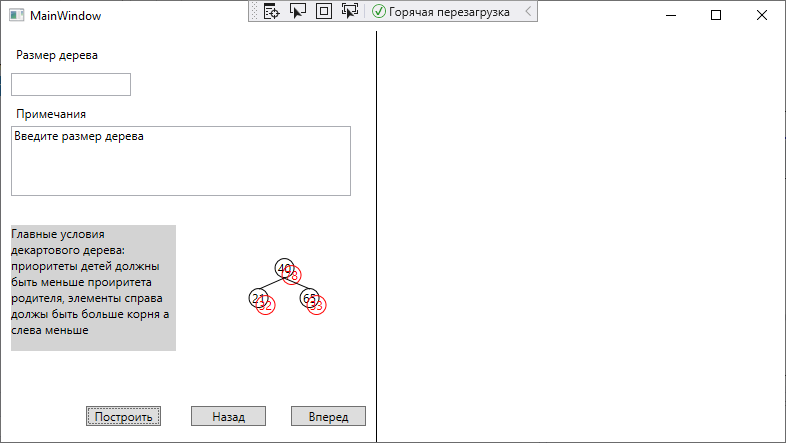


Рисунок 9. Ошибка при построении

При вводе размера дерева оно будет простроено.

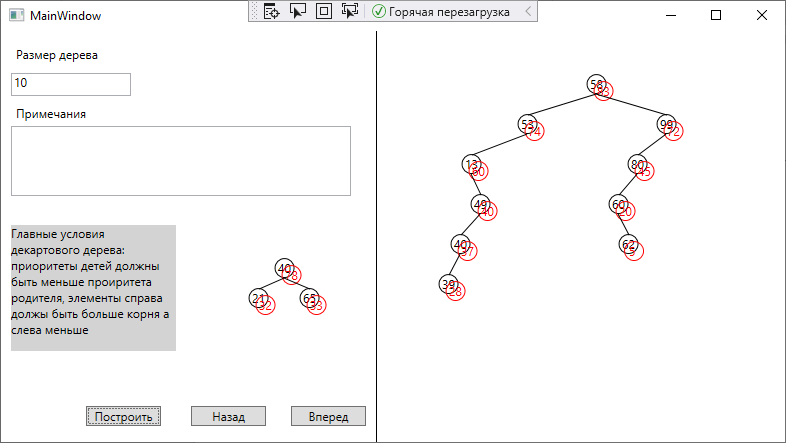
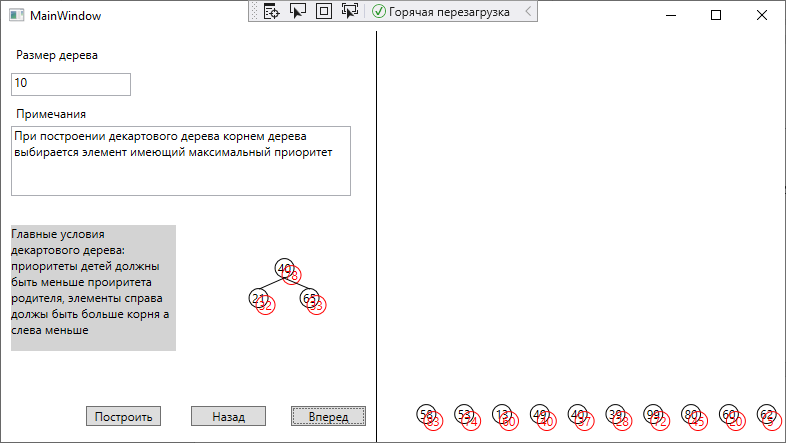
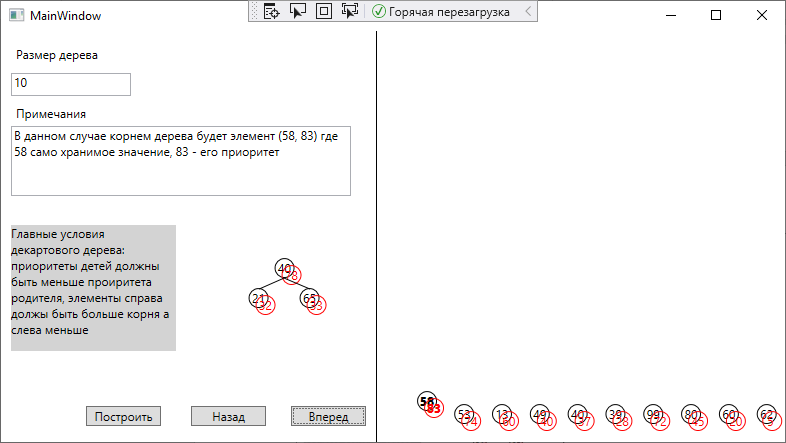
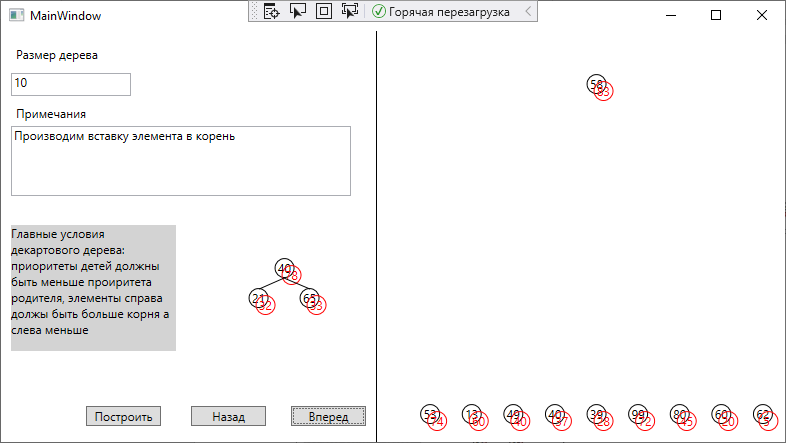


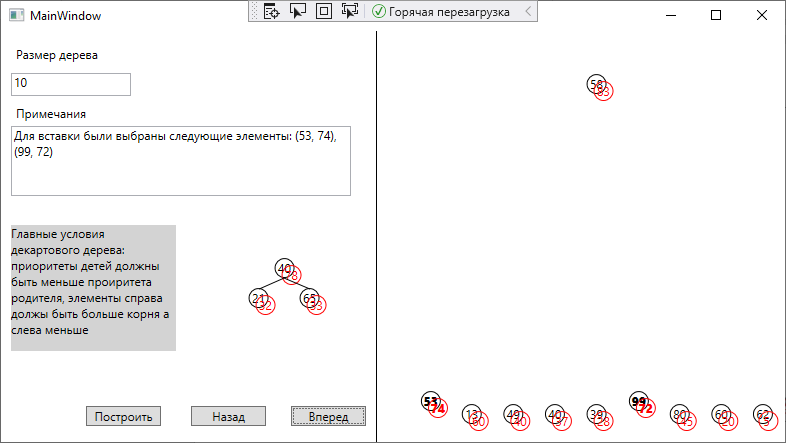
Рисунок 10. Построенное дерево.

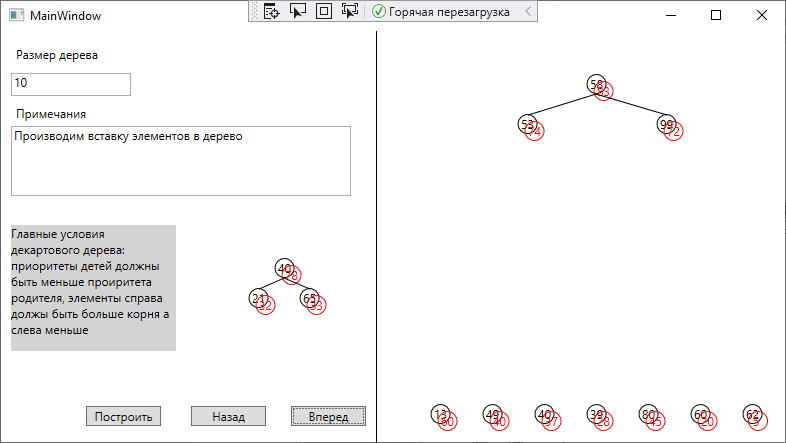
При нажатии клавиши вперед будет выполнена пошаговое построение дерева и к каждому шагу будут писаться комментарии.











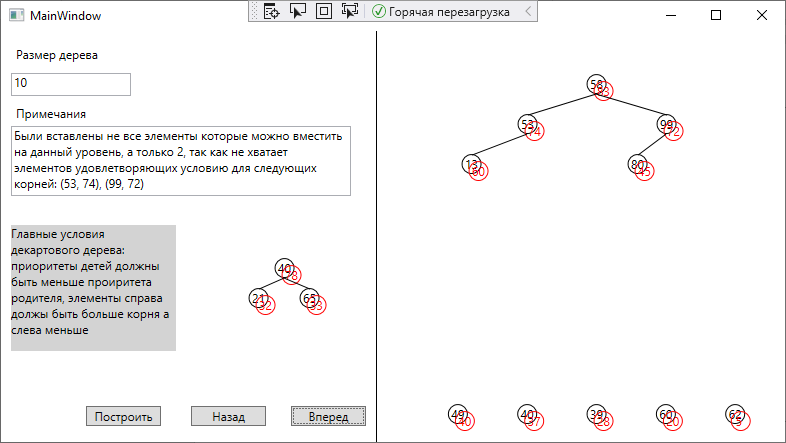


Рисунок 11 – 16. Пошаговое построение дерева

При использовании клавиши назад, все так же корректно работает, и мы видим предыдущий шаг построения

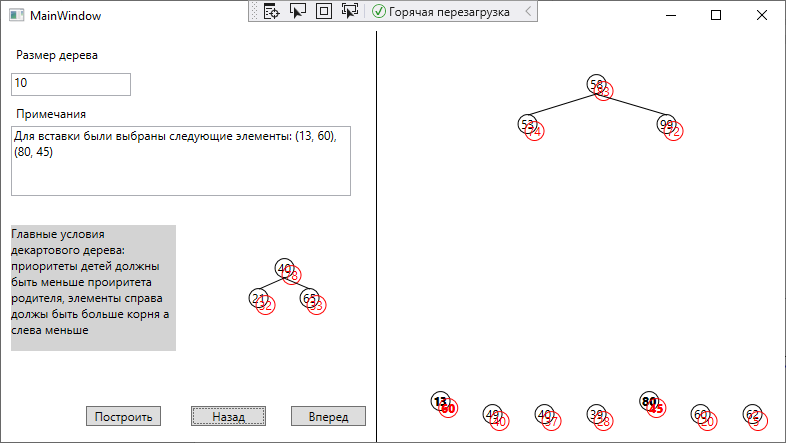


Рисунок 17. Шаг назад.

**Заключение.**

В ходе выполнения расчётно-графической работы был разработан визуализатор алгоритма «Декартово дерево». Знание работы определенного алгоритма поможет при необходимости реализовать его и эффективно выполнить поставленную задачу.

Визуализатор разработанный в ходе расчётно-графической работы также выполняет все поставленные в задаче функции.