МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Сцепляемая очередь на основе декартова дерева

Студентка гр. 8381 _____ Лисок М.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2019

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студентка Лисок М.А.

Группа 8381

Тема работы: сцепляемая очередь на основе декартова дерева.

Исходные данные: необходимо провести исследование алгоритмов операций сцепления (Merge) и разделения (Split) в декартовом дерево по неявному ключу, включающее генерацию входных данных, использование их для измерения количественных характеристик алгоритмов, сравнение экспериментальных результатов с теоретическими.

Содержание пояснительной записки:
«Содержание», «Введение», «Задание», «Теоретические положения», «Тестирование», «Исследование», «Заключение», «Список использованных источников».

Предполагаемый объем пояснительной записки:
Не менее 50 страниц.

Дата выдачи задания:

Дата сдачи реферата:

Дата защиты реферата:

Студент		Лисок М.А.
Преподаватель		Жангиров Т.Р.

АННОТАЦИЯ

В ходе выполнения курсовой работы была разработана программа с GUI, позволяющая исследовать алгоритмы операций разделения (Split) и слияния (Merge). Программа обладает следующей функциональностью: генерация данных, для получения декартово дерева поиска по неявному ключу с заданными параметрами, вывод полученных результатов в консольный вывод, а также возможность через взаимодействие с интерфейсом получать данные для различных операций над деревом.

SUMMARY

During the course work, was developed a program with a GUI, that allows to explore the algorithms of the operations of separation (Split) and merge (Merge). The program has the following functionality: data generation, to obtain a Cartesian search tree by implicit key with specified parameters, output of the results to console output, and also the ability to receive data for various operations on the tree through interaction with the interface.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы

Реализация и экспериментальное машинное исследование алгоритмов слияния (Merge) и разделения (Split) для декартово дерева по неявному ключу.

Основные задачи

Генерация входных данных, использование их для измерения количественных характеристик структур данных, алгоритмов, действий, сравнение экспериментальных результатов с теоретическими.

Методы решения

Разработка программы велась на базе операционной системы Mac OS в среде разработки QtCreator. Для создания графической оболочки использовался редактор интерфейса в QtCreator и система сигналов-слотов Qt.

1. ЗАДАНИЕ

Необходимо провести исследование алгоритмом операций разделения (Split) и слияния (Merge) для декартово дерева по неявному ключу.

Исследование должно содержать:

1. Анализ задачи, цели, технологию проведения и план экспериментального исследования.
2. Генерацию представительного множества реализаций входных данных (с заданными особенностями распределения (для среднего и для худшего случаев)).
3. Выполнение исследуемых алгоритмов на сгенерированных наборах данных. При этом в ходе вычислительного процесса фиксируется как характеристики (например, время) работы программы, так и количество произведенных базовых операций алгоритма.
4. Фиксацию результатов испытаний алгоритма, накопление статистики.
5. Представление результатов испытаний, их интерпретацию и сопоставление с теоретическими оценками.

2. ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

2.1 Декартово дерево

Декартово дерево (пирамида поиска, деректуча) это абстрактная структура данных, бинарное дерево, в узлах которого хранятся:

- ссылки на правое и левое поддерево;
- ссылка на родительский узел (необязательно);
- ключи x и y, которые являются двоичным деревом поиска по ключу x и двоичной кучей по ключу y; а именно, для любого узла дерева n:
    - ключи x узлов правого (левого) поддерева больше (меньше) либо равны ключа x узла n;
    - ключи y узлов правого и левого детей больше либо равны ключу y узла n.

Пример графического представления декартово дерева приведён на рис. 1.

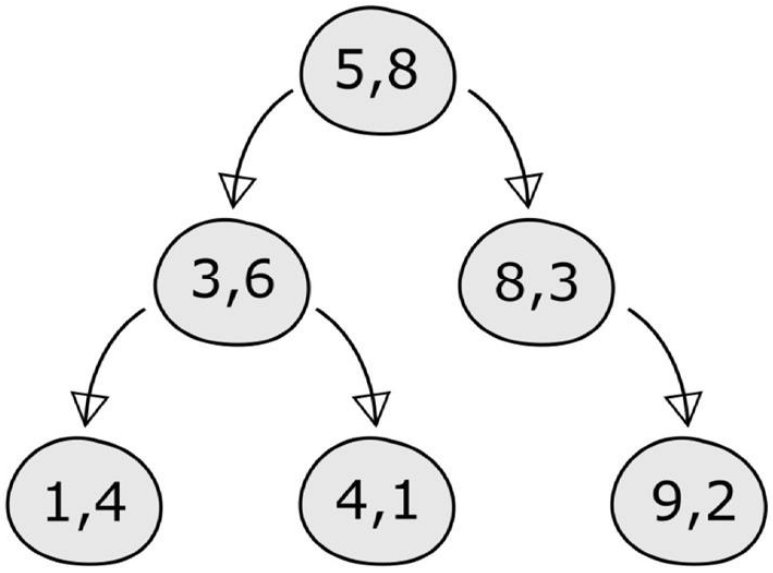

Рисунок 1 – Графическое представление декартового дерева

На рис. 1 первое значение в узле – ключ, а второе – приоритет. Соответственно, данное дерево – бинарная куча по приоритету, и БПД по ключу

В задании данной курсовой работы указано использование сцепляемой очереди, следовательно, воспользуемся модификацией изначальной структуры данных - декартовым деревом по неявному ключу.

Данная структуры получена следующим образом: оставим в первоначальном дереве только приоритет y, а вместо ключа x будем использовать следующую величину: количество элементов в нашей структуре, находящихся левее нашего элемента. Иначе говоря, будем считать ключом порядковый номер нашего элемента в дереве, уменьшенный на единицу.

Заметим, что при этом сохранится структура двоичного дерева поиска по этому ключу (то есть модифицированное декартово дерево так и останется декартовым деревом). Однако, с этим подходом появляется проблема: операции добавления и удаления элемента могут поменять нумерацию, и при наивной реализации на изменение всех ключей потребуется $O(n)$ времени, где n - количество элементов в дереве.

Графическое представление изначального декартового дерева с указанной модификацией изображено на рис. 2.

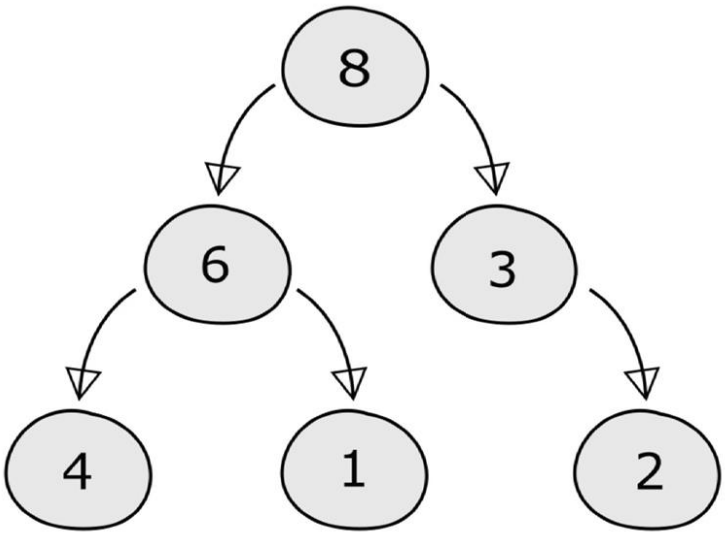

Рисунок 2 – Графическое представление декартового дерева по неявному ключу

На рис. 3 представлено обозначение связи переменных в разработанной программе и неявным индексом элементов в изображаемой структуре.

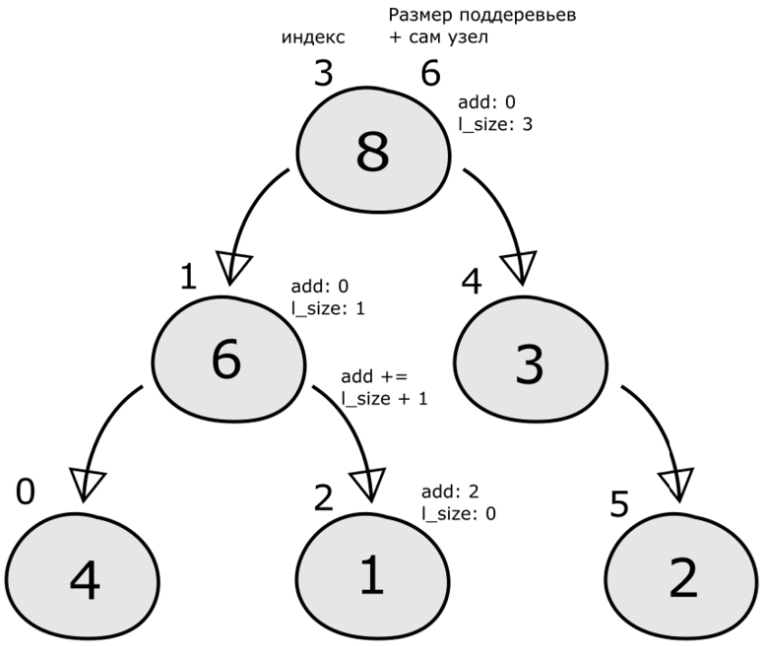

Рисунок 3 – Неявные индексы декартового дерева по неявному ключу

Приведём пример нахождения элемента под индексом 2. Подсчёт индексов начинается с корня. Если индекс корня больше 2, то идем вправо, если меньше, то влево. Индекс рассчитывается как сумма размера левого поддерева и добавочного числа (оно увеличивается, если на очередной итерации переходим к правому потомку на 1 + размер левого поддерева узла).

2 < 3(root index), а значит идем влево. Считаем индекс узла с приоритетом шесть: index = add + l_size = 0 + 1 = 1, 1 < 2, а значит идем вправо, увеличивая add на 1 + l_size (на 2). Считаем индекс узла с приоритетом 1: add + l_size = 2 + 0 = 2, элемент найден.

2.2 Операция Merge

Данная функция, наряду с функцией Split (см пункт 2.3), является основой любого декартового дерева. Её задача – слияние двух деревьев при условии, что ключи правого дерева больше ключей левого. В случае неявных ключей это условие выполняется автоматически, потому что операция Merge в этом случае добавляет одно дерево «в конец» другого, то есть после слияния окажется так, что неявные индексы элементов добавленного дерева больше индексов элементов дерева, в которое добавлялись элементы.

Допустим, необходимо произвести операцию Merge для двух декартовых деревьев, изображённых на рис. 4:

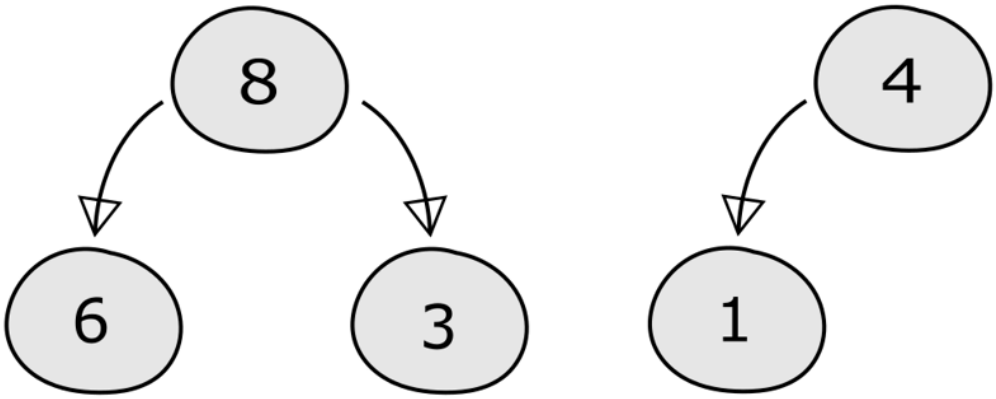

Рисунок 4 – Два декартовых дерева по неявному ключу

Весь алгоритм построен на сравнении приоритетов двух узлов. Если приоритет левого больше, то рекурсивно вызывается Merge для правого поддерева левой дерекучи и правой дерекучи, иначе вызывается Merge для левой дерекучи и левого поддерева правой дерекучи. Делается сравнение: 8 > 4, значит идет переход по 1 случаю, когда приоритет левого больше чем правого. Графически данное сравнение представлено на рис. 5.

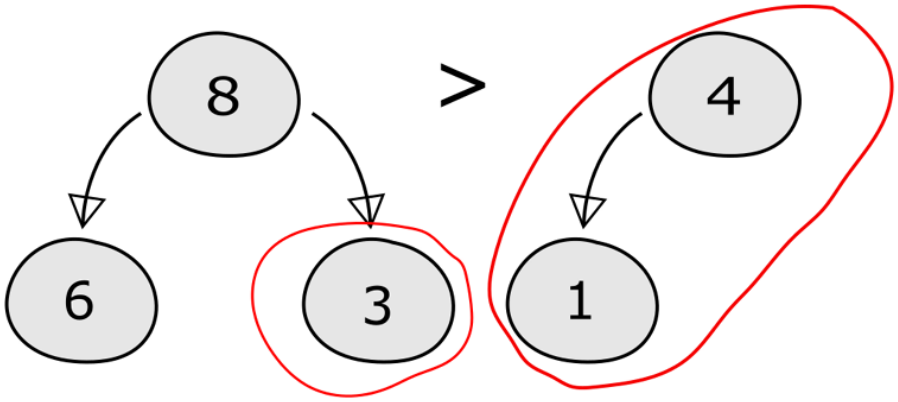

Рисунок 5 – Сравнение индексов деревьев

Теперь условный корень левой дерекучи – 3. Снова делается сравнение: 3 < 4, значит в правой дерекуче переходим к левому потомку. Данное сравнение представлено на рис. 6.

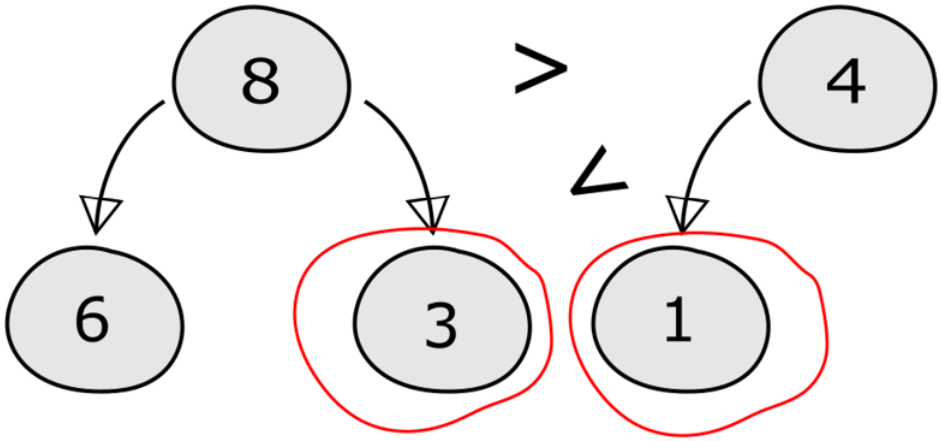

Рисунок 6 – Второе сравнение индексов деревьев

Сменился корень правой дерекучи, снова идет сравнение: 3 > 1, значит по логике, нужно переходить к правому потомку левой дерекучи, однако такого потомка нет. Как только обнаруживается, что дальше потомка нет, делается следующее: в обратной рекурсивной последовательности те узлы, у которых приоритет меньше становятся потомками тех, у которых приоритеты больше. В данном случае, поскольку 1 меньше 3, она становится правым сыном левой дерекучи. Графическая иллюстрация данной операции приведена на рис.7.

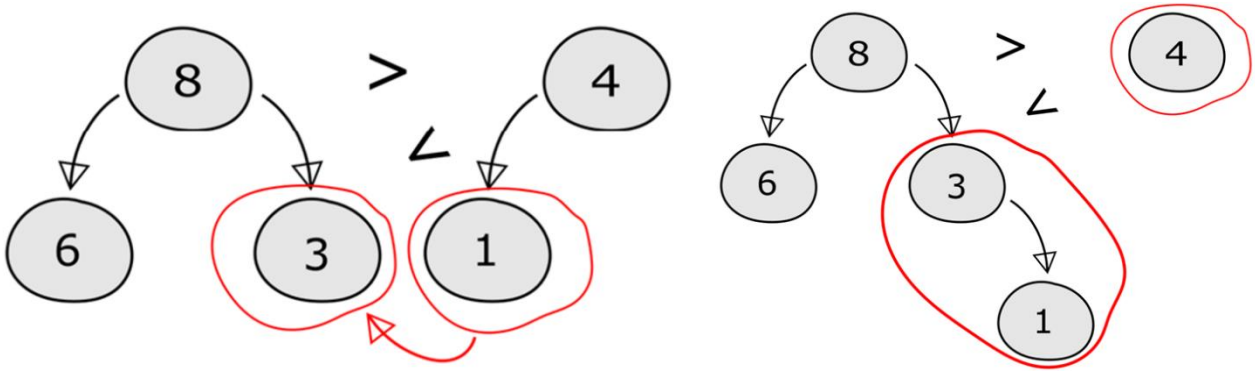

Рисунок 7 – Процесс проведения операции Merge

Теперь узлу 3 соответствует правый потомок 1, а значит далее они рассматриваются уже вместе. На следующем шаге получившееся поддерево присваивается узлу с большим приоритетом, то есть с 4. Графическая иллюстрация данной операции приведена на рис. 8.

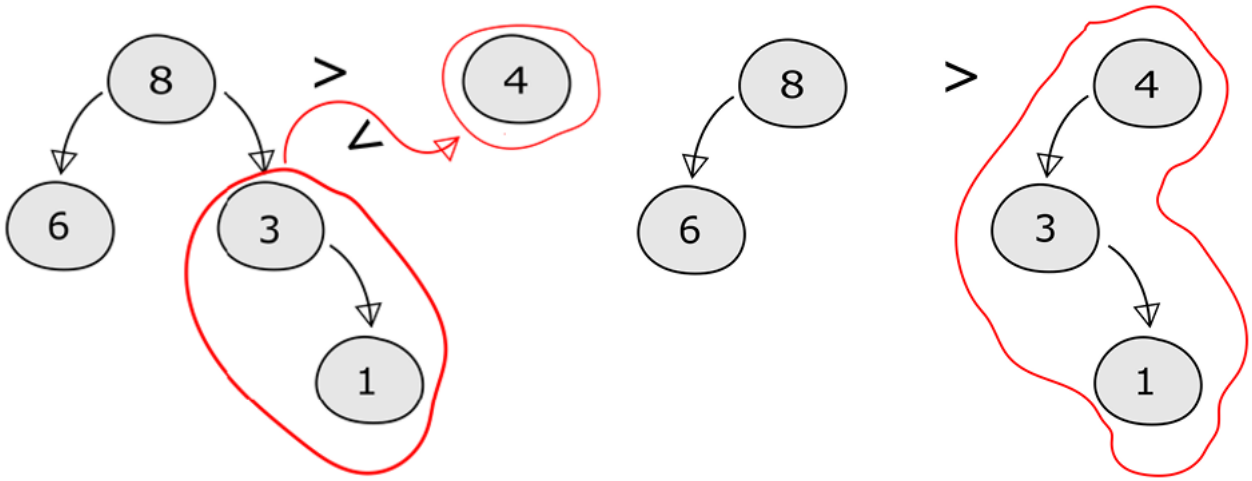

Рисунок 8 - Процесс проведения операции Merge

На последнем шаге дерекуча с меньшим приоритетом корня становится потомком дерекучи с большим приоритетом корня. Графическая иллюстрация данной операции приведена на рис. 9.

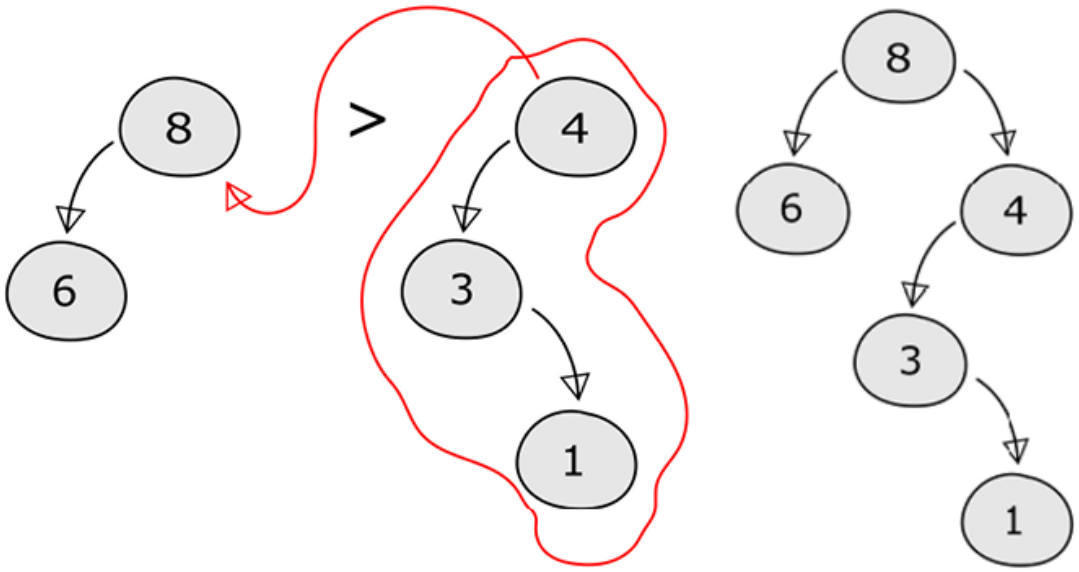

Рисунок 9 – Результат работы операции Merge

Как можно видеть дерекучи склеились в одну, правило кучи выполнено, правило бинарного дерева поиска также выполнено, если проверить индексы узлов.

2.3 Операция Split

Функция Split расщепляет одну дерекучу на две по заданному ключу. В данном случае в качестве ключа будет выступать неявный индекс. То есть, по

сути, это операция разделения массива на 2 части, в один массив пойдет то, что было до i-го индекса, в другой всё что после.

Данная функция работает следующим образом – осуществляется проход по «среднему» звену узлов и распределение их в правую или левую дерекучу:

- Если индекс текущего узла больше либо равен индексу-разделителю, то текущий узел вместе со всеми правыми потомками включается в новую правую дерекучу как левый потомок.
- Если индекс текущего узла меньше индекса-разделителя, то текущий узел вместе со всеми левыми потомками включается в новую левую дерекучу как правый потомок.

Покажем работу этой функции расщепив по индексу 3 декартово дерево, полученное функцией Merge.

Проходим по дерекуче похожим образом, только теперь сравниваются индексы, а не приоритеты. Начинаем с корня. Его индекс меньше тройки, значит отправляем корень вместе с левым потомком в левую дерекучу и переходим к правому потомку. Полученные данные проиллюстрированы на рис. 10.

Вид новых дерекуч представлен на рис. 11.

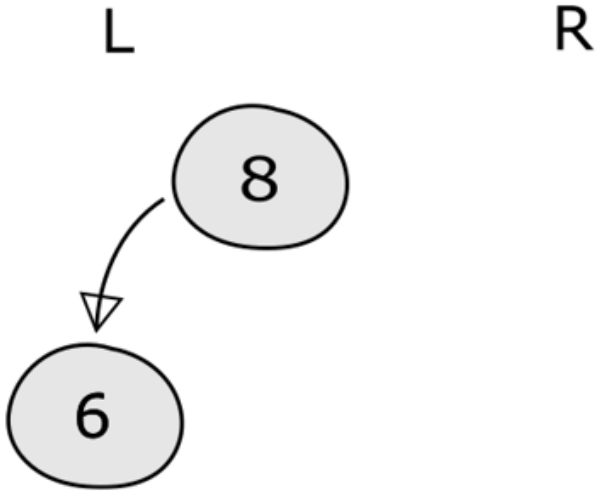

Рисунок 10 – Процесс проведения операции Split

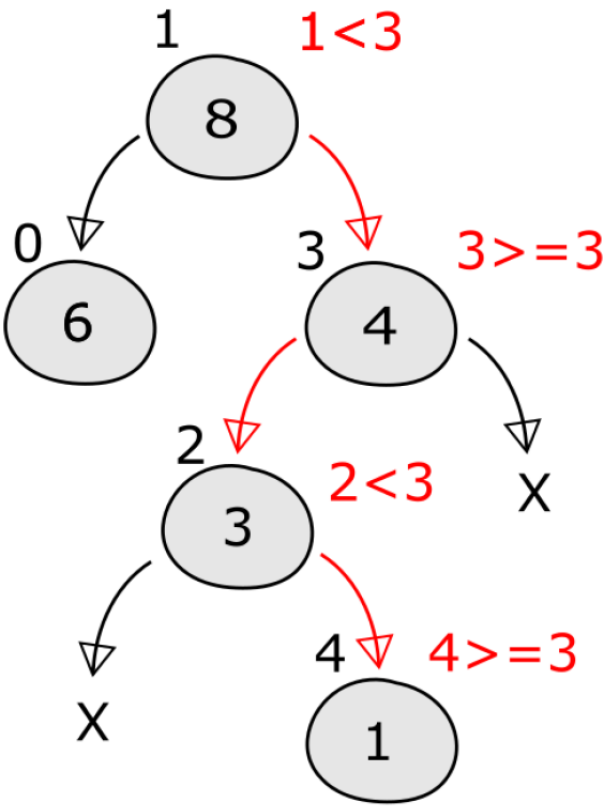

Рисунок 11 – Индексы декартового дерева

На следующем шаге индексы совпали, а значит, что нужно в новую правую дерекучу перенести узел с приоритетом 4 и его правых потомков. Графическая иллюстрация данной операции приведена на рис. 12.

Вид новых дерекуч представлен на рис. 13.

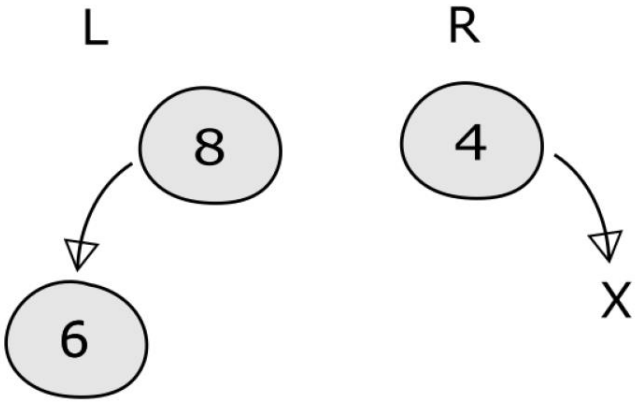

Рисунок 12 – Процесс проведения операции Split

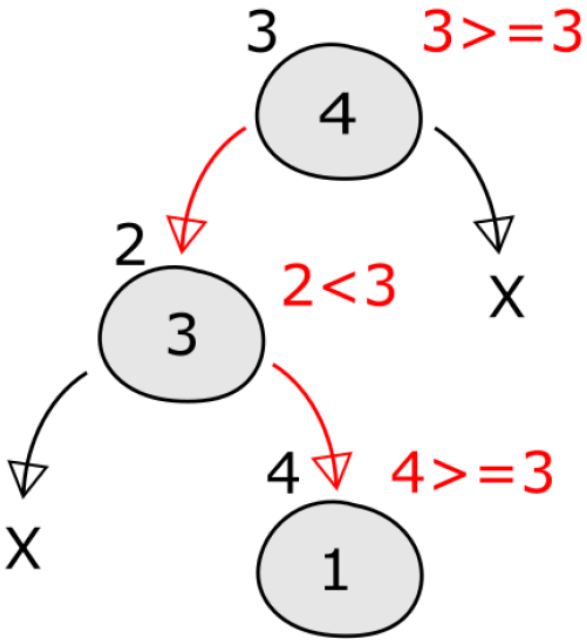

Рисунок 13 - Индексы декартового дерева

Теперь снова индекс разделитель больше текущего, поэтому узел с приоритетом 3 переходит в новую левую дерекучу правым потомком. Графическая иллюстрация данной операции приведена на рис. 14.

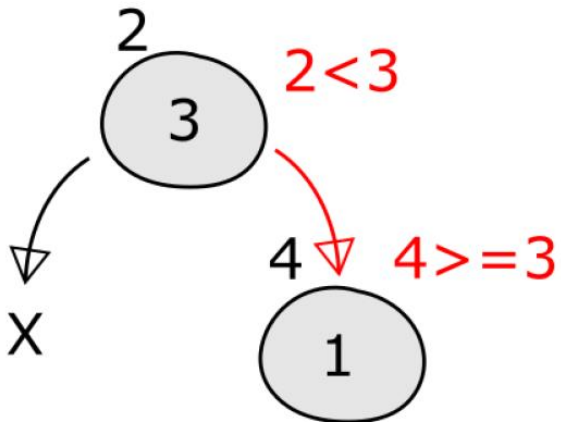

Рисунок 14 - Индексы декартового дерева

Вид новых дерекуч представлен на рис. 15.

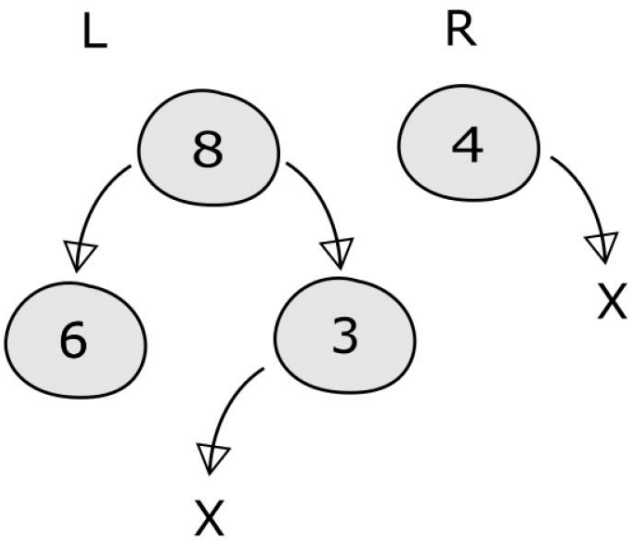

Рисунок 15 – Процесс проведения операции Split

Остался последний узел, и он переходит в новую правую дерекучу левым потомком, поскольку его индекс больше индекса разделителя. Графическая иллюстрация данной операции приведена на рис. 16.

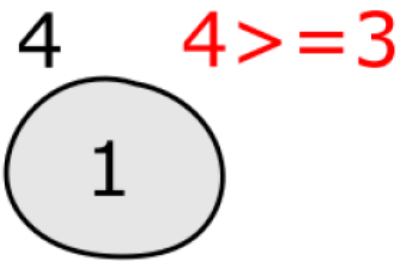

Рисунок 16 - Индексы декартового дерева

Убирая указатели на пустые узлы, получились те же дерекучи, что и до слияния, представленные на рис. 17.

2.4 Применение описанной структуры

Таким образом, описана структура, от которой можно отрезать слева часть произвольной длины и слить две любые части в одну в нужном порядке. Благодаря данной структуре данных появляются следующие возможность по работе с данным:

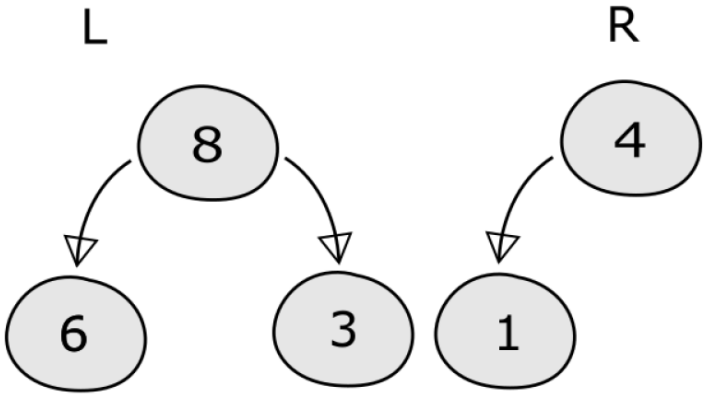

Рисунок 17 – Результат работы операции Split

- вставить элемент в любое место (отрежем нужное количество элементов слева, сольем левое дерево с деревом из одного добавленного элемента и результат — с правым деревом),
- переставить любой кусок массива куда угодно (сделаем нужные разрезы и слияния в правильном порядке),
- совершать групповые операции с элементами,
- сделав на одном исходном массиве два дерева из элементов разной четности, можно решить задачу про смену мест четных и нечетных на отрезке.

3. ТЕСТИРОВАНИЕ

3.1 Вид программы

Программа представляет собой окно с графическим интерфейсом, в котором пользователю предоставляется возможность ввести количество генерируемых узлов и выбрать операцию для тестирования. Вид программы после запуска представлен на рис. 18.

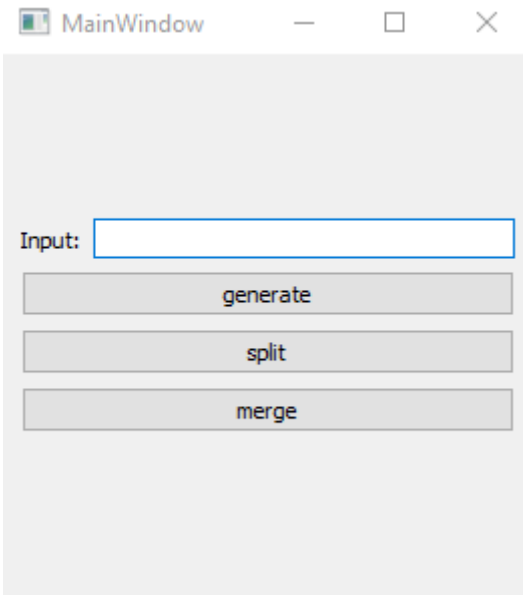

Рисунок 18 – Вид программы после запуска

3.2 Тестирование генерации дерева для выполнения операции

Одним из ключевых моментов работы программы является генерация псевдослучайных деревьев с аналогичными приоритетами (для соблюдения относительной уравновешенности). Для тестирования данной функциональности был разработан фрагмент программы, осуществляющий вывод текущего дерева в файл в виде схематического представления. Пример одного из таких выводов представлен в Приложении И.

3.3 Получение информации о выполнении операции

После выполнения операции, выбранной пользователем, в консольный вывод qDebug() выводится соответствующее сообщение, содержащее необходимую информацию о работе программы.

Пример данного вывода для операции Merge с 2000 сгенерированных узлов равен: `min 6 avg 14.0105  max 21 \n ( 2000 ; 14.0105 )`.

Пример данного вывода для операции Split с 1000 сгенерированных узлов равен: `min 6 avg 12.9063  max 20 \n ( 1000 ; 12.9063 )`.

4. ИССЛЕДОВАНИЕ

4.1. План экспериментального исследования.

Для проведения исследования сложности алгоритма расщепления сцепляемой очереди на основе рандомизированной пирамиды поиска необходимо учесть факт того, что на одном и том же наборе данных при повторном запуске будут получаться различные декартовы деревья (в данной работе по неявному ключу). Следовательно, необходимо провести исследование не только для деревьев с различным количеством узлов, но и для деревьев одинаковой длины. Опорным параметром для исследования алгоритма были выбраны количество узлов и число совершенных операций. По аналогичному плану было проведено исследование алгоритма сцепления. После накопления данных было проведено сравнение результатов и выведены зависимости, отвечающие за эффективность алгоритмов. После анализа полученной статистики, а также экспериментальных и теоретических зависимостей, были сделаны выводы о сложности операций и соответствующих им алгоритмов расщепления и сцепления в среднем и худшем случаях.

План проведения исследования:

- Получение информации о зависимости количества узлов и числа итераций
    - Для одного и того же набора данных
    - Для различных наборов
- Анализ собранной информации, выводы о зависимостях эффективности алгоритма от указанных параметров
- Анализ собранной информации, сравнение экспериментальных значений с теоретическими, выводы о сложности алгоритма расщепления и сцепления.

4.2. Исследование зависимости количества итераций от размера массива входных данных для алгоритма расщепления (Split)

Был проведен ряд тестов, где переменным параметром бралось количество узлов, необходимое для построения декартова дерева. Сначала исследовалось количество итераций на одном и том же наборе исходных данных, а затем общие результаты сводились в таблицу.

Ниже представлены табл. 1 и рис. 19, иллюстрирующие полученные результаты.

Таблица 1 – Результаты тестирования зависимости числа итераций от количества входных элементов для операции разделения (Split)

Размер массива	Минимальное число итераций	Среднее арифметическое число итераций	Максимальное число итераций
1000	4	13.4388	23
2000	8	14.3711	24
3000	7	15.6289	26
4000	8	15.2268	24
5000	8	15.6458	26
6000	6	15.6907	25
7000	9	16.6632	24
8000	10	16.7708	27
9000	8	17.0313	26
10000	9	17.3711	28

Окончание таблицы 1

11000	9	17.5876	26
12000	6	17.8163	36
13000	10	18.3878	28
14000	10	18.3918	28
15000	11	18.6042	30
16000	9	18.7732	27
…			
25000	11	18.9388	29
35000	10	20.0714	33
45000	7	20.6224	34

Количество итераций необходимое для расщепления декартова дерева по заданному ключу составляет $O(log_2 n)$. Учитывая, что дерево имеет размер n, итоговая сложность алгоритма расщепления для всего дерева составит $O(2{,}99 \cdot log_2 n)$, т.к. в данном случае используется “O”-большая верхняя асимптотика, в которой символ Брахмана нивелирует константу, которая уравновешивает постоянные затраты алгоритма.

На рис. 19 представлен график зависимости количества элементарных операций, осуществляемых алгоритмом операции Split от количества входных элементов (узлов). Представленная визуализация данных о работе алгоритма явно указывает, что количество итераций даже при большом количестве входных данных показывает рост функции не больший, чем теоретический. Аналогично, стоит отметить, что худший случай был взят во время исследования алгоритма на одном и том же наборе данных.

В итоге на основе анализа экспериментальных данных был сделан вывод, что алгоритм расщепления в декартовом дереве является крайне эффективным и с большой вероятностью строит сбалансированное дерево, избегая самого худшего случая – вырождения дерева в односвязный список, где его сложность становится линейной, т.е. равной $O(n)$.

Рисунок 19 – График зависимости количества итераций от размера исходных данных для операции Split

4.3 Исследование зависимости количества итераций от размера массива входных данных для алгоритма слияния (Merge)

Согласно постановке задачи исследования, аналогичные пункту 4.2 исследования были проведены также для операции сцепления (Merge) для декартового дерева по неявному ключу. Результаты представлены в табл. 2.

На основании тестов, полностью аналогичных предыдущим для всех случаев были получены практические данные, подтверждающие теоретическое положение о верхней асимптотике операции слияния равной $O(2{,}99 \cdot log_2\, n)$.

Таблица 2 - Результаты тестирования зависимости числа итераций от количества входных элементов для операции слияния (Merge)

Размер массива	Минимальное число итераций	Среднее арифметическое число итераций	Максимальное число итераций
1000	4	11.8776	21
2000	4	13.6211	21
3000	5	15.3196	26
4000	9	15.5361	25
5000	6	16.4592	27
6000	7	16.2143	31
7000	7	16.1735	30
8000	7	15.8061	29
9000	8	17.1563	26
10000	8	17.1327	29
11000	9	17.5155	29
12000	3	16.6224	26
13000	9	18.1031	32
14000	7	17.7245	29
15000	10	17.9794	29
16000	5	17.7835	26

Окончание таблицы 2

…			
25000	12	19.0213	27
35000	9	19.0714	34
45000	11	20.7113	31

Единственным отличием от тестов для операции Split является поле “размер массива”, равное количеству элементов в итоговом дереве, когда же сцепляемые имеют одинаковый размер, равный половине итогового.

Визуальная репрезентация данных, представленных в таблице, изображено на графике рис. 20.

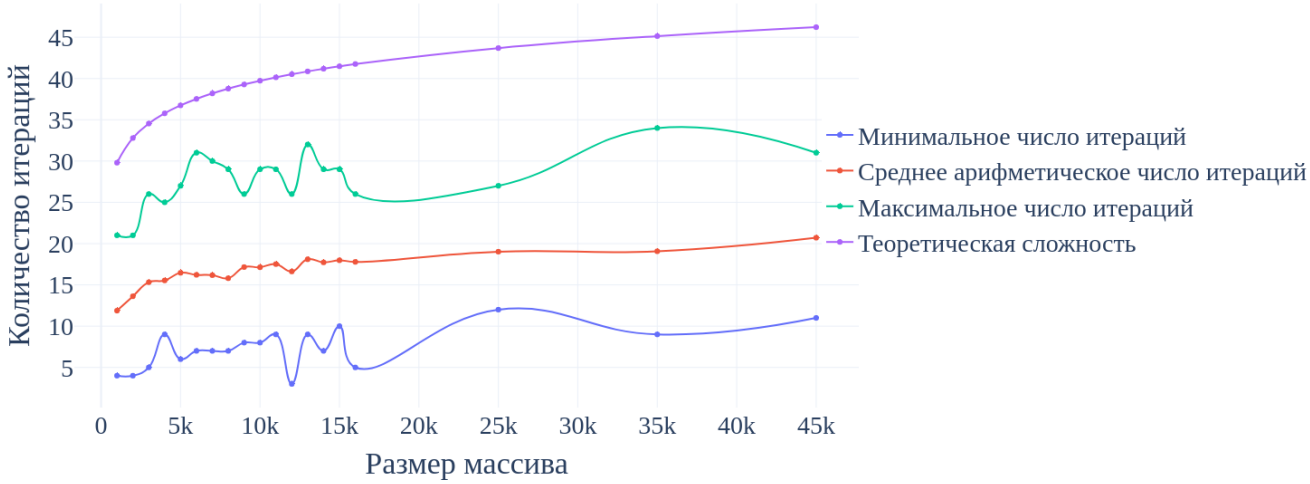

Рисунок 20 - График зависимости количества итераций от размера исходных данных для операции Merge

Стоит отметить факт того, что при проведении тестирования для операции Merge итоговый график, несмотря на обширную выбору (массивы от 1000 до 45000 случайно сгенерированных входных данных) и большой количество тестов

(100 генераций для каждого размера), имеет “скачки” для функций зависимости минимального и максимального количества итераций от размера массива.

Данное наблюдение объясняется невозможностью гарантировать генерацию абсолютно равных по высоте деревьев для соединения, а, следовательно и доступ к их элементам на разных уровнях, что даёт некоторые колебания на представленных графиках.

4.4 Исследование зависимости количества итераций от уникальности псевдослучайно сгенерированных приоритетов дерева

В ходе проведения исследований, описанных в пунктах 4.2 и 4.3, было получено практическое подтверждение теоремы о том, что количество итераций (сложность работы) алгоритма операций Split и Merge для декартового дерева по неявному ключу обладает единственной зависимостью вида $O(2{,}99 \cdot log_2 n)$, где n – размер массива входных данных (узлов). Однако, сложность выполнения операций с дерекучей корректнее связать с высотой рабочего дерева – величиной, напрямую зависящей от n.

На практике возможна ситуация, когда высота декартова дерева может быть линейной относительно его размеров. Например, высота декартова дерева, построенного по набору ключей $(1, 1), \dots, (n, n)$, будет равна n. Во избежание таких случаев, необходимым является обеспечение уникальности генерируемых приоритетов. Об этом также говорит следующая теорема: в декартовом дереве из n узлов, приоритеты y которого являются случайными величинами с равномерным распределением, средняя глубина вершины $O(log_2 n)$ (Викиконспект: [сайт]. URL: https://goo-gl.su/IMsQ2dsB).

Для практического подтверждения данного утверждения был проведён ряд тестов, представленных в табл. 3. Данные, представленные в ней демонстрируют, что при достаточном большом диапазоне генерируемых псевдослучайных чисел – приоритетов наблюдается логарифмический рост сложности алгоритма Split (частный случай всех операций с деревом), когда же количество узлов превышает,

начинают появляться коллизии, вырождающие дерево в линейных список с соответствующей сложностью $O(n)$. Визуальное представление данного факта изображено в виде графика на рис. 21.

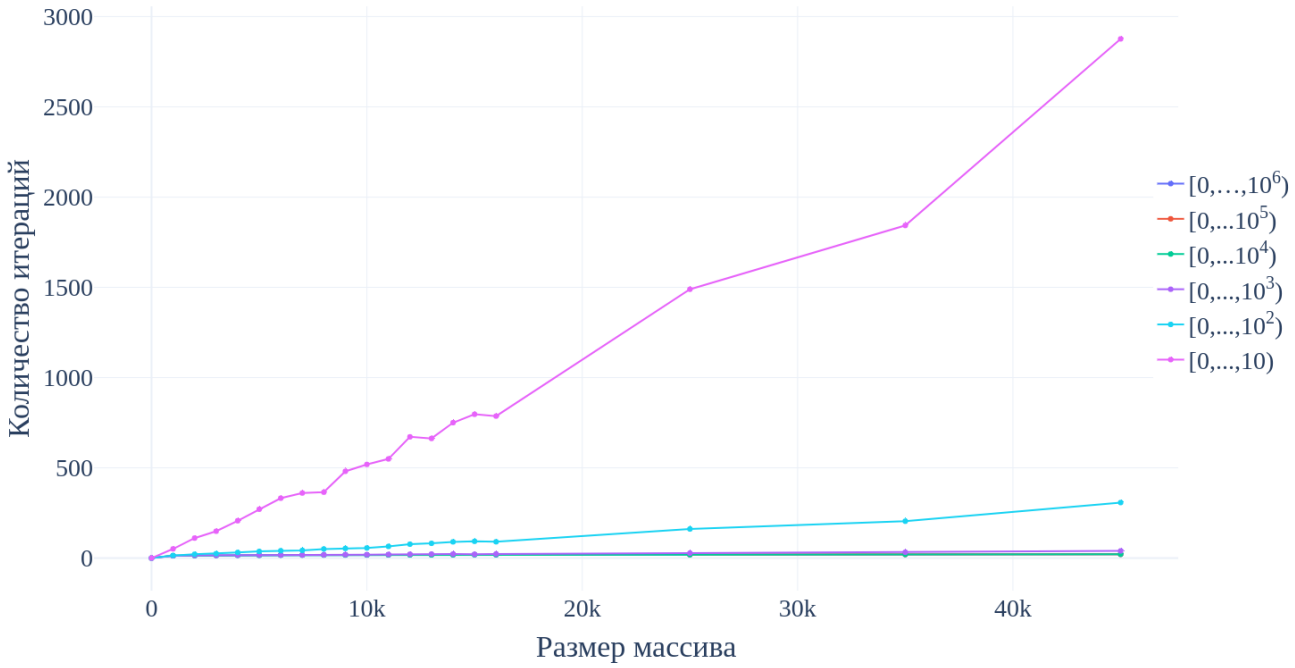

Рисунок 21 – Зависимость сложности от количества коллизий

Таблица 3 – Результаты тестирования зависимости числа итераций от уникальности приоритетов

Размер массива	Доступный размер случайно генерируемых приоритетов					
	$[0,\ldots,10^6)$	$[0,\ldots,10^5)$	$[0,\ldots,10^4)$	$[0,\ldots,10^3)$	$[0,\ldots,10^2)$	$[0,\ldots,10)$
1000	12.6771	12.6771	13.2784	12.5306	14.449	50.8367
2000	14.1429	14.1429	13.6947	14.9375	21.134	110.724
3000	14.8776	14.298	14.8351	15.0309	24.268	148.784
4000	15.7526	15.8125	15.5833	16.4796	31.051	206.714
5000	16.1531	16.2857	15.3711	17.1531	36.9167	270.541
6000	16.6327	16.6563	16.299	16.8776	40.5816	331.626
7000	16.8454	16.3505	16.7245	17.1122	42.449	360.306
8000	17.4286	16.2577	16.7526	17.7551	50.1531	364.908
9000	17.7872	17.4227	16.8571	18.8163	53.2653	480.633
10000	16.9381	16.6122	17.6667	19.1474	56.0103	518.835
11000	17.4184	17.551	17.9691	19.9388	64.5816	549.296
12000	17.9691	17.5102	17.5521	21.3061	76.9388	671.612
13000	17.9592	17.9381	17.6327	22.2143	81.6837	662.714
14000	17.8866	17.8571	18.2887	22.3878	89.8144	750.133
15000	18.551	18.4082	18.7347	20.9897	92.7347	796.592
16000	18.1122	18.4694	19.0306	22.9592	90.1327	786.602
…						
25000	18.9381	19.7835	18.7732	27.4639	161.092	1489.23
35000	20.3367	20.3265	20.1979	32.2268	204.163	1842.76
45000	20.8041	21.0532	21.6224	39.7245	307.454	2876.43

График, представленный на рис. 21 даёт явную картину скорости вырождения дерева в линейный список при много большем количестве входных данных в сравнении с количеством доступных псевдослучайных приоритетов. Однако, из-за вынужденного использования масштаба с большими значениями он не даёт представления о поведении деревьев с меньшим количеством данных. Поэтому на рис. 22 представлен аналогичный график в меньшем масштабе, который позволяет оценить зависимость сложности алгоритма от количества коллизий при меньшем количестве узлов.

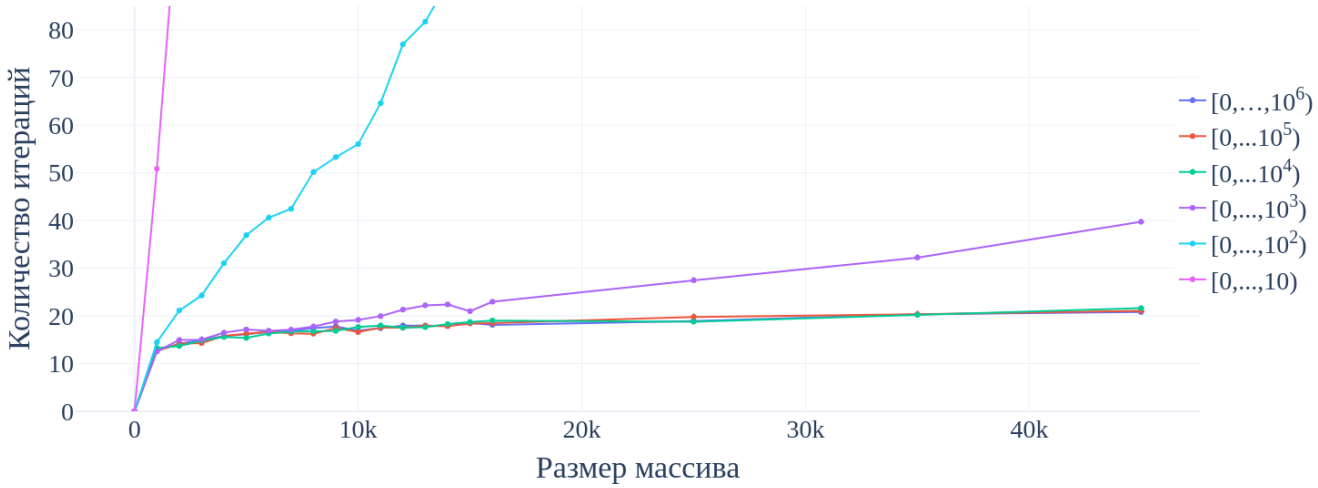

Рисунок 22 - Зависимость сложности от количества коллизий

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы была разработана программа, которая обладает следующей функциональностью: генерация входных данных для построения декартово дерева по неявному ключу, применение различных операций и получение статистического результата их выполнения. С помощью программы было проведено исследование различных случаев алгоритмом слияния (Merge) и разделения (Split). В ходе исследования была выявлена зависимость эффективности алгоритма от различных параметров. В результате было выявлено, что эффективность напрямую зависит от количества узлов во входном / входных деревьях, а также от уникальности сгенерированных псевдослучайных приоритетов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Bjarne Stroustrup. A Tour of C++. M.: Addison-Wesley, 2018. 217 с.
2. Макс Шлее. Qt5.10. Профессиональное программирование на C++. М.: BHV-СПб, 2018, 513 с.
3. Перевод и дополнение документации QT // CrossPlatform.RU. URL: http://doc.crossplatform.ru/
4. Qt Documentation // Qt. URL: https://doc.qt.io/qt-5/index.html (дата обращения: 15.11.2019).
5. Коллективный блог про программирование // Хабр. URL: https://habr.com/ru/post/102364/ (дата обращения: 12.12.2019).
6. Справочник по алгоритмам и структурам данных // MAXimal. URL: https://e-maxx.ru/algo/treap (дата обращения: 03.12.2019).
7. Викиконспект // Neerc IFMO. URL: https://goo-gl.su/ZACdRmxP (дата обращения: 29.11.2019).
8. Кнут Д. Э. Искусство программирования. Том 3. Сортировка и поиск = The Art of Computer Programming. Volume 3. Sorting and Searching / под ред. В. Т. Тертышного (гл. 5) и И. В. Красикова (гл. 6). — 2-е изд. — Москва: Вильямс, 2007. — Т. 3. — 832 с. — ISBN 5-8459-0082-1.

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД ПРОГРАММЫ. MAIN.C

```
#include "mainwindow.h"

#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

ПРИЛОЖЕНИЕ Б.

ИСХОДНЫЙ КОД ПРОГРАММЫ. IMPLICIT_TREAP_NODE.CPP

```
#include "implicit_treap_node.h"

implicit_treap_node::implicit_treap_node()
{

}
```

ПРИЛОЖЕНИЕ В.

ИСХОДНЫЙ КОД ПРОГРАММЫ. IMPLICIT_TREAP_NODE.H

```
#ifndef BINARYTREAP_H
#define BINARYTREAP_H

#include <cstdlib>
#include <iostream>
#include <QDebug>
#include <chrono>
#include <cstring>

#define ONE_NODE 1
#define RAND_FACTOR 20000000

using namespace std;
typedef unsigned long ul;

template<class TYPE>
class ImplicitTreapNode
{
public:
    ImplicitTreapNode();
    ~ImplicitTreapNode();
    ImplicitTreapNode(const TYPE* _data, ul _priority = std::rand() %
RAND_FACTOR);

    static ImplicitTreapNode* Merge(bool flag,ImplicitTreapNode* _left,
ImplicitTreapNode* _right);                    // Merging two treaps. Need agree
with: keys of one of treaps needs to be bigger, than keys of another Treap.
    //static     ImplicitTreapNode*     MergeNew(ImplicitTreapNode*     _left,
ImplicitTreapNode* _right);                    // Merging two treaps. Need agree
with: keys of one of treaps needs to be bigger, than keys of another Treap.
    static void Split(ImplicitTreapNode* _node, size_t _index, bool flag,
ImplicitTreapNode*& _left, ImplicitTreapNode*& _right, size_t _add = 0);
// Splits up treap. First output treap has all keys smaller than keys of the
second one.

    static void Clear(ImplicitTreapNode*& _treap);                              //
Destroy root this and all childs.
    static size_t SizeOf(ImplicitTreapNode* _treap);                            //
Returns mSizeOf field of _treap

    ImplicitTreapNode* Insert(const TYPE* mData, size_t _index,bool, ul _prior =
std::rand() % RAND_FACTOR);
    ImplicitTreapNode*                  Remove(size_t                   _index);
// Remove Node from NON EMPTY Treap by key.
```

```
    TYPE& GetData(size_t _index);                                          //
Returns key of node by index in upping order of keys.

    inline void RecalcSizeOf();                                            //
Recalculate SizeOf
    inline void AddData(const TYPE* _data);
    static size_t iterationsSplit;
    static size_t iterationsMerge;
    static size_t height(ImplicitTreapNode<TYPE>*);
    ImplicitTreapNode* getMLeft(){
        return mLeft;
    }
    void copy(ImplicitTreapNode<TYPE>*, ImplicitTreapNode<TYPE>*);
private:

    ImplicitTreapNode* mLeft;
    ImplicitTreapNode* mRight;

    ul mPriority;
    size_t mSizeOf;                                                        //
Count of subtreaps + 1 (root exactly)

    TYPE* mData;

};



//_____________________________________NODE_METHODS_____________________________
_________________//

//____CONSTRUCTORS____//
template<class TYPE> size_t ImplicitTreapNode<TYPE>::iterationsSplit;
template<class TYPE> size_t ImplicitTreapNode<TYPE>::iterationsMerge;

template<class TYPE>
ImplicitTreapNode<TYPE>::ImplicitTreapNode()
{
    mSizeOf = ONE_NODE;
    mLeft = nullptr;
    mRight = nullptr;
    mData = nullptr;
    mPriority = 0;
    iterationsSplit = 0;
    iterationsMerge = 0;
}
template<class TYPE>
size_t ImplicitTreapNode<TYPE>::height(ImplicitTreapNode<TYPE>* node){
```

```
    if (node == nullptr)
        return 0;
    else
    {

        int lDepth = height(node->mLeft);
        int rDepth = height(node->mRight);

        // use the larger one
        if (lDepth > rDepth)
            return(lDepth + 1);
        else
            return(rDepth + 1);
    }
}

template<class TYPE>
void     ImplicitTreapNode<TYPE>::copy(ImplicitTreapNode<TYPE>     *     dest,
ImplicitTreapNode<TYPE> *src)
{
    if(!src){
        dest = nullptr;
        return;
    }
    cout << "Copy" << src->data << endl;
    dest = new ImplicitTreapNode<TYPE> {src->mData, src->mPriority};
    copy(dest->left, src->left);
    copy(dest->right, src->right);

}
template<class TYPE>
ImplicitTreapNode<TYPE>::~ImplicitTreapNode()
{
    delete mData;
}

template<class TYPE>
ImplicitTreapNode<TYPE>::ImplicitTreapNode(const TYPE* _data, ul _priority) :
ImplicitTreapNode()
{
    AddData(_data);
    mPriority = _priority;
}

//____TREAP_EDITING____//

template<class TYPE>
```

```
ImplicitTreapNode<TYPE>*        ImplicitTreapNode<TYPE>::Merge(bool        flag,
ImplicitTreapNode* _left, ImplicitTreapNode* _right)
{
    ImplicitTreapNode<TYPE>* result;
    if(flag){
        iterationsMerge++;
    }
    if (!_right) {
        return _left;
    }
    if (!_left) {
        return _right;
    }
    else if (_left->mPriority > _right->mPriority)
    {
        _left->mRight = Merge(flag, _left->mRight, _right);
        result = _left;
    }
    else
    {
        _right->mLeft = Merge(flag, _left, _right->mLeft);
        result = _right;
    }

    result->RecalcSizeOf();
    return result;
}

template<class TYPE>
void  ImplicitTreapNode<TYPE>::Split(ImplicitTreapNode*  _node,  size_t  _index,
bool flag, ImplicitTreapNode*& _left, ImplicitTreapNode*& _right, size_t _add)
{

    if (!_node)
    {
        _left = _right = nullptr;
        return;
    }
    if(flag){
        iterationsSplit++;
    }
    size_t curr_index = 1 + SizeOf(_node->mLeft);
    if (_index <= curr_index) {
        Split(_node->mLeft, _index,flag, _left, _node->mLeft, _add);
        _right = _node;
        _node->RecalcSizeOf();
    }
    else {
```

```
        Split(_node->mRight, _index-curr_index, flag, _node->mRight, _right,
_add);
        _left = _node;
        _node->RecalcSizeOf();
    }
}

template<class TYPE>
ImplicitTreapNode<TYPE>*  ImplicitTreapNode<TYPE>::Insert(const  TYPE*  _data,
size_t _index,bool flag, ul _prior)
{
    ImplicitTreapNode* _l = nullptr;
    ImplicitTreapNode* _r = nullptr;
    ImplicitTreapNode* _in = new ImplicitTreapNode(_data, _prior);
    Split(this, _index,flag, _l, _r);
    return Merge(false, Merge(false,_l, _in), _r);
}

template<class TYPE>
ImplicitTreapNode<TYPE>* ImplicitTreapNode<TYPE>::Remove(size_t _index)
{
    ImplicitTreapNode<TYPE>* _r = nullptr;
    ImplicitTreapNode<TYPE>* _l = nullptr;
    ImplicitTreapNode<TYPE>* _el = nullptr;
    Split(this, _index, false, _l, _r);
    Split(_r, 1, false, _el, _r);
    delete _el;
    _el = nullptr;
    return Merge(false, _l, _r);
}

template<class TYPE>
void ImplicitTreapNode<TYPE>::Clear(ImplicitTreapNode<TYPE>*& _treap)
{
    if(_treap)
    {
        if(_treap->mLeft)
            Clear(_treap->mLeft);
        if(_treap->mRight)
            Clear(_treap->mRight);

        delete _treap;
        _treap = nullptr;
    }
}

//____SIZE____//
```

```
template<class TYPE>
size_t ImplicitTreapNode<TYPE>::SizeOf(ImplicitTreapNode* _treap)
{
    return !_treap ? 0 : _treap->mSizeOf;
}

template<class TYPE>
void ImplicitTreapNode<TYPE>::RecalcSizeOf()
{
    mSizeOf = SizeOf(mLeft) + SizeOf(mRight) + ONE_NODE;
}

template<class TYPE>
void ImplicitTreapNode<TYPE>::AddData(const TYPE* _data)
{
    mData = new TYPE;
    std::memmove(mData, _data, sizeof (TYPE));
}



template<class TYPE>
TYPE& ImplicitTreapNode<TYPE>::GetData(size_t _index)
{
    ImplicitTreapNode<TYPE>* _node = this;
    size_t _add = 0;
    size_t curr_index = _add + SizeOf(_node->mLeft);
    while (_index != curr_index)
    {
        if(_index < curr_index)
            _node = _node->mLeft;

        else if(_index > curr_index) {
            _add = _add + 1 + SizeOf(_node->mLeft);
            _node = _node->mRight;
        }
        curr_index = _add + SizeOf(_node->mLeft);
    }
    return *_node->mData;
}


#endif // BINARYTREAP_H
```

ПРИЛОЖЕНИЕ Г.

ИСХОДНЫЙ КОД ПРОГРАММЫ. MAINWINDOW.CPP

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include "math.h"
#include <sstream>
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    queue=nullptr;
    keys = nullptr;
    avr = 0;
    getNumRand();
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::on_actionAddRandUnit_triggered()
{
    /*next_permutation(begin(numsForRand), end(numsForRand));
    std::stringstream bufData;
    copy(numsForRand.begin(), numsForRand.end(), ostream_iterator<int>(bufData));
    int data = stoi(bufData.str().c_str());*/
    int data = std::rand()%1000;
    queue->Push(&data);
}
void MainWindow::generateKeys(){
    for(int i=0; i<100; i++){
        keys->push_back(std::rand() % length);
    }
}
void MainWindow::generateValues(){
    for(int i=0; i<length; i++){
        on_actionAddRandUnit_triggered();
    }
}

void MainWindow::on_pushButton_clicked()
{
    if(queue){
        queue->~MergeQueue();
    }
```

```
    if(keys){
        keys->clear();
    }
    queue = new MergeQueue<int>();
    string str = ui->input->text().toUtf8().constData();
    this->length = stoi(str);
    keys = new vector<size_t>();
    generateKeys();
    generateValues();
}
void MainWindow::findMinIter(){
    avr = 0;
    minKey = static_cast<long>(queue->alliterationsForSplit->at(0));
    maxKey = static_cast<long>(queue->alliterationsForSplit->at(0));
    for(int i=1; i<queue->alliterationsForSplit->size();i++){
        if(minKey > queue->alliterationsForSplit->at(i)){
            minKey = queue->alliterationsForSplit->at(i);
        }
        if(maxKey < queue->alliterationsForSplit->at(i)){
            maxKey = queue->alliterationsForSplit->at(i);
        }
    }
    int len=1;
    for(int i=1; i<queue->alliterationsForSplit->size();i++){
        if(minKey   ==   queue->alliterationsForSplit->at(i)   ||   maxKey   ==
queue->alliterationsForSplit->at(i)){
            continue;
        }else{
            len++;
            avr+=queue->alliterationsForSplit->at(i);
        }
    }
    avr = avr/len;
    qDebug() << "min" << minKey << "avg" << avr <<" max"<< maxKey << endl;
}
void MainWindow::on_split_clicked()
{

    MergeQueue<int>* q1 = new MergeQueue<int>();
    MergeQueue<int>* q2 = new MergeQueue<int>();
    ImplicitTreapNode<int>::iterationsSplit = 0;
    for(int i=0; i<static_cast<long>(keys->size()); i++){
        queue->Split(q1, q2, keys->at(i), true);
        generateValues();
    }
    findMinIter();
    qDebug() << "(" <<length<<";"<<avr<<")"<<" ";
    q1->~MergeQueue();
```

```
    q2->~MergeQueue();
}

void MainWindow::on_logMath_clicked()
{
    string str = ui->input->text().toUtf8().constData();
    this->length = stoi(str);
    double log_res = log(length)/log(2);
    qDebug() << "(" <<length<<";"<<log_res<<")"<<" ";

}

void MainWindow::getNumRand() {
    for(int i = 1; i < 10; i++) numsForRand.push_back(i);
}


void MainWindow::on_merge_clicked()
{
    string str = ui->input->text().toUtf8().constData();
    this->length = stoi(str);
    ImplicitTreapNode<int>::iterationsMerge = 0;
    //for(int i = 0; i<100; i++){
        MergeQueue<int>* q1 = new MergeQueue<int>();

        for(int i=0; i<length; i++){
            int data = std::rand()%1000;
            q1->Push(&data);
        }

        qDebug()<<"height                                                    q1:"<<
ImplicitTreapNode<int>::height(q1->getMRoot())<<endl;
        MergeQueue<int>* q2 = new MergeQueue<int>();
        for(int i=0; i<length; i++){
            int data = std::rand()%1000;
            q2->Push(&data);
        }
        qDebug()<<"height                                                    q2:"<<
ImplicitTreapNode<int>::height(q2->getMRoot())<<endl;
        q1->Merge(q2);
        qDebug()<<"all:";
        qDebug()<<"height                                                final:"<<
ImplicitTreapNode<int>::height(q1->getMRoot())<<endl;
    //}
}
```

ПРИЛОЖЕНИЕ Д.

ИСХОДНЫЙ КОД ПРОГРАММЫ. MAINWINDOW.H

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include "mergequeue.h"
QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    void on_actionAddRandUnit_triggered();
    ~MainWindow();

private slots:
    void on_pushButton_clicked();

    void on_split_clicked();

    void setMaxKey(long key){
        this->maxKey = key;
    }

    long getMaxKey(){
        return maxKey;
    }
    void setMinKey(long key){
        this->minKey = key;
    }

    long getMinKey(){
        return minKey;
    }
    void generateKeys();
    void generateValues();
    void findMinIter();
    void on_logMath_clicked();



    void on_merge_clicked();
```

```
private:
    Ui::MainWindow * ui;
    MergeQueue<int> * queue;

    vector<size_t>* keys;
    long long int maxKey;
    long long int minKey;
    double avr;
    long length;

    vector<int> numsForRand;
    void getNumRand();
    vector<long>* heights;
};
#endif // MAINWINDOW_H
```

ПРИЛОЖЕНИЕ Е.

ИСХОДНЫЙ КОД ПРОГРАММЫ. MERGEQUEUE.H

```
#ifndef MERGEQUEUE_H
#define MERGEQUEUE_H

#include "implicit_treap_node.h"

template <class TYPE>
class MergeQueue
{
public:
    MergeQueue();
    ~MergeQueue();

    void Push(TYPE* data);
    void Pop();
    void Merge(MergeQueue* queue);
    void Split(MergeQueue* q1, MergeQueue* q2, size_t index, bool);

    TYPE& Top();
    void DisplayTreap(std::string&);
    void PrintData(std::string&);

    bool Empty();
    size_t Size();
    vector<size_t>* alliterationsForSplit;
    vector<size_t>* alliterationsForMerge;
    ImplicitTreapNode<TYPE>* getMRoot(){
        return mRoot;
    }

private:
    ImplicitTreapNode<TYPE>* mRoot;

};

#endif // MERGEQUEUE_H

template<class TYPE>
MergeQueue<TYPE>::MergeQueue() {     mRoot = nullptr;   alliterationsForSplit =
nullptr; alliterationsForMerge = nullptr; }

template<class TYPE>
MergeQueue<TYPE>::~MergeQueue() {
    ImplicitTreapNode<TYPE>::Clear(mRoot);
}
```

```
template<class TYPE>
void MergeQueue<TYPE>::Push(TYPE* data)
{

    if(Empty())
    {
        mRoot = new ImplicitTreapNode<TYPE>(data);
        return;
    }

     mRoot = mRoot->Insert(data, 0, false);


}

template<class TYPE>
void MergeQueue<TYPE>::Pop()
{
    if(Empty())
        throw std::out_of_range("Call POP operation in empty queue");
    mRoot = mRoot->Remove(ImplicitTreapNode<TYPE>::SizeOf(mRoot)-1);
}

template<class TYPE>
void MergeQueue<TYPE>::Split(MergeQueue *q1, MergeQueue *q2, size_t index, bool
flag)
{

    if(index >= Size())
        throw std::out_of_range("Invalid implicit index");
    ImplicitTreapNode<TYPE>::Split(mRoot, index, flag, q1->mRoot, q2->mRoot);
    mRoot = nullptr;
    if(!alliterationsForSplit){
        alliterationsForSplit = new vector<size_t>();
    }
    alliterationsForSplit->push_back(ImplicitTreapNode<TYPE>::iterationsSplit);
    ImplicitTreapNode<int>::iterationsSplit = 0;

}

template<class TYPE>
TYPE &MergeQueue<TYPE>::Top()
{
    if(Empty())
        throw std::out_of_range("Call TOP operation in empty queue");
    return mRoot->GetData(ImplicitTreapNode<TYPE>::SizeOf(mRoot)-1);
}
```

```
template<class TYPE>
void MergeQueue<TYPE>::DisplayTreap(std::string& out)
{
    ImplicitTreapNode<TYPE>::Display(mRoot, out);
}

template<class TYPE>
void MergeQueue<TYPE>::PrintData(std::string &out)
{
    ImplicitTreapNode<TYPE>::PrintData(mRoot, out);
}

template<class TYPE>
void MergeQueue<TYPE>::Merge(MergeQueue<TYPE>* queue)
{
    mRoot = ImplicitTreapNode<TYPE>::Merge(true, queue->mRoot, mRoot);
    //size_t height = ImplicitTreapNode<TYPE>::height(this->mRoot);
    queue->mRoot = nullptr;
    if(!alliterationsForMerge){
        alliterationsForSplit = new vector<size_t>();
    }

//alliterationsForMerge->push_back(ImplicitTreapNode<TYPE>::iterationsMerge);
   // ImplicitTreapNode<int>::iterationsMerge = 0;
}

template<class TYPE>
bool MergeQueue<TYPE>::Empty() {    return !mRoot ? true : false;    }

template<class TYPE>
size_t       MergeQueue<TYPE>::Size()       {                            return
ImplicitTreapNode<TYPE>::SizeOf(mRoot);    }
```

ПРИЛОЖЕНИЕ Ж.

ИСХОДНЫЙ КОД ПРОГРАММЫ. MERGEQUEUE.H

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>513</width>
    <height>410</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>MainWindow</string>
  </property>
  <widget class="QWidget" name="centralwidget">
   <widget class="QPushButton" name="pushButton">
    <property name="geometry">
     <rect>
      <x>10</x>
      <y>40</y>
      <width>221</width>
      <height>31</height>
     </rect>
    </property>
    <property name="text">
     <string>generate</string>
    </property>
   </widget>
   <widget class="QLineEdit" name="input">
    <property name="geometry">
     <rect>
      <x>20</x>
      <y>10</y>
      <width>113</width>
      <height>21</height>
     </rect>
    </property>
   </widget>
   <widget class="QPushButton" name="split">
    <property name="geometry">
     <rect>
      <x>10</x>
      <y>90</y>
      <width>221</width>
```

```
      <height>31</height>
     </rect>
    </property>
    <property name="text">
     <string>split</string>
    </property>
   </widget>
   <widget class="QPushButton" name="merge">
    <property name="geometry">
     <rect>
      <x>10</x>
      <y>140</y>
      <width>221</width>
      <height>31</height>
     </rect>
    </property>
    <property name="text">
     <string>merge</string>
    </property>
   </widget>
   <widget class="QPushButton" name="logMath">
    <property name="geometry">
     <rect>
      <x>20</x>
      <y>190</y>
      <width>211</width>
      <height>31</height>
     </rect>
    </property>
    <property name="text">
     <string>log2_n</string>
    </property>
   </widget>
  </widget>
  <widget class="QStatusBar" name="statusbar"/>
 </widget>
 <resources/>
 <connections/>
</ui>
```

ПРИЛОЖЕНИЕ И.

ПСЕВДОСЛУЧАЙНОЕ ДЕРЕВО. LOG.TXT

392(15759) -> 824(5601) -> 213(23992) -> 215(8683) -> 601(6077) -> 813(29174) -> 129(31240) -> 467(29541) -> 561(31627) -> 549(29556) -> 292(5997) -> 775(20608) -> 503(3829) -> 264(12181) -> 192(7605) -> 335(8759) -> 44(11258) -> 651(26740) -> 353(19314) -> 990(15145) -> 497(2589) -> 685(19090) -> 40(26488) -> 169(23831) -> 112(30695) -> 17(14932) -> 818(610) -> 875(20072) -> 40(9313) -> 296(12673) -> 752(18443) -> 424(32678) -> 611(30877) -> 142(11462) -> 855(24855) -> 413(16641) -> 881(300) -> 962(10548) -> 176(25705) -> 556(1993) -> 637(26534) -> 193(12734) -> 279(22701) -> 850(17398) -> 205(31783) -> 626(31934) -> 221(17870) -> 459(17825) -> 741(20175) -> 754(6511) -> 223(22142) -> 675(12938) -> 67(22848) -> 911(11635) -> 607(2483) -> 532(2963) -> 457(14369) -> 654(16972) -> 584(29734) -> 53(16962) -> 296(29855) -> 559(8932) -> 229(4565) -> 423(24129) -> 585(24041) -> 717(18696) -> 617(10112) -> 896(13022) -> 869(29361) -> 337(23271) -> 526(13357) -> 626(9357) -> 556(23216) -> 823(14485) -> 72(29350) -> 835(32356) -> 111(28704) -> 405(6540) -> 993(27384) -> 460(22428) -> 801(22850) -> 113(14887) -> 477(5108) -> 357(2324) -> 303(12760) -> 58(6439) -> 477(1200) -> 365(13007) -> 958(17578) -> 900(17713) -> 662(13829) -> 940(20851) -> 188(21763) -> 790(14924) -> 153(21520) -> 411(25547) -> 11(30771) -> 30(20326) -> 202(8116) -> 264(8260) -> 195(32525) -> 737(13261) -> 123(24596) -> 902(14962) -> 423(28520) -> 629(31928) -> 625(25627) -> 577(14474) -> 565(3487) -> 882(17086) -> 164(5109) -> 43(22758) -> 576(30227) -> 170(20315) -> 10(4757) -> 60(1926) -> 800(18087) -> 119(23152) -> 18(28464) -> 851(25484) -> 678(22593) -> 371(22466) -> 658(26302) -> 576(21694) -> 19(2125) -> 695(21624) -> 140(13694) -> 285(12550) -> 896(4667) -> 711(25760) -> 235(30833) -> 270(29170) -> 413(30974) -> 187(8360) -> 64(7900) -> 222(31286) -> 492(142) -> 725(13031) -> 505(27593) -> 3(27432) -> 61(14181) -> 23(29972) -> 543(1924) -> 527(2625) -> 802(4099) -> 213(7627) -> 102(8480) -> 192(13985) -> 763(2668) -> 773(32270) -> 518(8177) -> 487(28297) -> 833(28070) -> 372(20159) -> 334(25874) -> 314(25824) -> 477(4414) -> 202(3625) -> 958(7391) -> 787(9905) -> 168(14018) -> 474(28022) -> 186(8313)