**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: Рандомизированное БДП

|  |  |  |
| --- | --- | --- |
| Студентка гр. 8381 |  | Гречко В.Д. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2019

# ЗАДАНИЕ

# на курсовую работу

|  |  |  |
| --- | --- | --- |
| Студентка Гречко В.Д. | | |
| Группа 8381 | | |
| Тема работы: Рандомизированное БДП | | |
| Исходные данные: необходимо провести исследование алгоритма вставки и удаления в рандоминизированное бинарное дерево поиска, включающее генерацию входных данных, использование их для измерения количественных характеристик алгоритмов, сравнение экспериментальных результатов с теоретическими. | | |
| Содержание пояснительной записки:  «Содержание», «Введение», «Задание», «Описание программы», «Тестирование», «Исследование», «Заключение», «Список использованных источников». | | |
| Предполагаемый объем пояснительной записки:  Не менее 30 страниц. | | |
| Дата выдачи задания: | | |
| Дата сдачи реферата: | | |
| Дата защиты реферата: | | |
| Студент |  | Гречко В.Д. |
| Преподаватель |  | Жангиров Т.Р. |

# Аннотация

В ходе выполнения курсовой работы была разработана программа с GUI, позволяющая исследовать алгоритм вставки в рандоминизированное бинарное дерево поиска, а также удаление заданного элемента. Программа обладает следующей функциональностью: создание файла, содержащего входные данные для построения бдп, создание рандоминизированного бдп по заданному файлу, вывод полученных результатов в файл.

# Summary

In the course work a program with a GUI was developed that allows you to examine the algorithm for inserting into a randomized binary search tree, as well as deleting a given element. The program has the following functionality: creating a file containing input data for building a BDP, creating a randomized BDP for a given file, outputting the results to a file.

# Содержание

|  |  |  |
| --- | --- | --- |
|  | Введение | 6 |
| 1. | Задание | 7 |
| 2. | Описание программы | 8 |
| 2.1. | Описание интерфейса пользователя | 8 |
| 2.2. | Описание основного класса для рандоминизированных бдп | 8 |
| 2.3. | Описание алгоритма вставки и удаления в рандоминизированном бдп | 10 |
| 2.4. | Описание генерирования входных значений | 10 |
| 3 | Тестирование | 11 |
| 3.1. | Вид программы | 11 |
| 4 | Исследование | 12 |
| 4.1 | План экспериментального исследования | 12 |
| 4.2 | Исследование зависимостей от полученной высоты дерева для алгоритма вставки | 12 |
| 4.3 | Исследование зависимостей от количества итераций для алгоритма вставки | 20 |
| 4.4 | Исследование зависимостей значений, вставленных в корень и размеров исходных данных | 23 |
| 4.5 | Исследование зависимостей от количества итераций для алгоритма удаления | 24 |
| 4.6 | Выводы об исследовании алгоритма | 27 |
|  | Заключение  Список использованных источников | 28  29 |
|  | Приложение А. Исходный код программы. MAIN.CPP | 30 |
|  | Приложение Б. Исходный код программы. MAINWINDOW.H | 31 |
|  | Приложение В. Исходный код программы.MAINWINDOW.CPP | 32 |
|  | Приложение Г. Исходный код программы. BIN\_STRUCT.H | 35 |
|  | Приложение Д. Исходный код программы. TREE\_FUN.CPP | 37 |
|  | Приложение Е. Исходный код программы. MAIN\_FUN.H | 39 |
|  | Приложение Ж. Исходный код программы. DRAW\_TREE.CPP | 40 |

# ВВЕДЕНИЕ

## Цель работы

Реализация и экспериментальное машинное исследование алгоритмов кодирования (Фано-Шеннона, Хаффмана), быстрого поиска на основе БДП или хеш-таблиц, сортировок.

## 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 Elementary OS в среде разработки QtCreator. Для создания графической оболочки использовался редактор интерфейса в QtCreator и система сигналов-слотов Qt.

# Задание

Необходимо провести исследование алгоритма вставки и удаления в рандоминизированном бинарном дереве поиска в среднем и худшем случаях.

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. Описание программы

## 2.1. Описание интерфейса пользователя

Интерфейс программы представляет из себя: панель ввода данных для генерации файла, панель ввода данных для построения дерева, панель ввода данных для исследования алгоритма удаления из бдп и панель вывода. Основные виджеты панели генерации файла и их назначение представлены в табл. 1.

Таблица 1 – Основные виджеты панели генерации файла

|  |  |  |
| --- | --- | --- |
| Класс объекта | Название виджета | Назначение |
| QLineEdit | num | Поле ввода размера бдп |
| QPushButton | generate | Кнопка запуска генерации файла |

Основные виджеты панели создания бдп и их назначение представлены в табл. 2.

Таблица 2 – Основные виджеты панели создания бдп

|  |  |  |
| --- | --- | --- |
| Класс объекта | Название виджета | Назначение |
| QLineEdit | Input\_tree | Поле ввода данных для считывания |
| QPushButton | Choose\_file | Кнопка выбора файла для считывания |
| QLineEdit | Input\_del | Поле для ввода элемента, подлежащего удалению |
| QPushButton | PrintTree | Кнопка создания бдп |
| QPushButton | Delete\_elem | Кнопка запуска удаления элемента |
| QGraphicsView | graphicsView | Поле для графического представления дерева |

## 2.2. Описание основного класса для рандоминизированных бдп

Для реализации бдп был создан класс BinTree. Основные методы класса представлены в табл. 5.

Таблица 5 – Основные функции работы с бинарным деревом

|  |  |
| --- | --- |
| Функция | Назначение |
| void BinTree() | Создает пустое бинарное дерево |
| Node\* insert(Node\* p, int k) | Рандомизированная вставка нового узла с ключом k в дерево p |
| int max\_depth(Node \*hd) | Возвращает максимальную глубину дерева |
| Node\* insertroot(Node\* p, int k) | Вставка нового узла с ключом k в корень дерева p |
| Node\* rotateright(Node\* p) | Правый поворот вокруг узла p |
| Node\* rotateleft(Node\* q) | Левый поворот вокруг узла q |
| int getsize(Node\* p) | Получение размера дерева |
| void fixsize(Node\* p) | Установление корректного размера дерева |
| Node\* join(Node\* p, Node\* q) | Объединение двух деревьев |
| Node\* remove(Node\* p, int k) | Удаление из дерева p первого найденного узла с ключом k |

Программа имеет возможность графического отображения полученного бинарного дерева с помощью виджета QGraphicsView. Функции, необходимые для графического представления дерева, представлены в табл. 6.

Таблица 6 – Функции, связующие графический интерфейс и алгоритмы

|  |  |
| --- | --- |
| Функция | Назначение |
| QGraphicsScene \*graphic(BinTree \*tree, QGraphicsScene \*&scene, int depth) | По заданному бинарному дереву выполняет рисование в объекте QGraphicsScene |
| int treePainter(QGraphicsScene \*&scene, Node \*node, int w, int h, int wDelta, int hDelta, QPen &pen, QBrush &brush, QFont &font, int depth) | Рекурсивный алгоритм обхода дерева и рисования узлов в заданном объекте QGraphicsScene |

## 2.3. Описание алгоритма вставки и удаления в рандоминизированном бдп

Известно, что если заранее перемешать как следует все ключи и потом построить из них дерево (ключи вставляются по стандартной схеме в полученном после перемешивания порядке), то построенное дерево окажется неплохо сбалансированным (его высота будет порядка 2 против для идеально сбалансированного дерева). Любой вводимый ключ может оказаться корнем с вероятностью (— размер дерева до вставки), следовательно выполняется с указанной вероятностью вставка в корень, а с вероятностью 1-— рекурсивную вставка в правое или левое поддерево в зависимости от значения ключа в корне.

Удаление происходит по ключу — ищется узел с заданным ключом и этот узел удаляется из дерева. Основное свойство дерева поиска — любой ключ в левом поддереве меньше корневого ключа, а в правом поддереве — больше корневого ключа. Это свойство позволяет очень просто организовать поиск заданного ключа, перемещаясь от корня вправо или влево в зависимости от значения корневого ключа. Далее происходит объединение левого и правого поддеревьев найденного узла, удаляется узел и возвращается корень объединенного дерева.

## 2.4. Описание генерирования входных значений

Для возможности генерации входных данных было реализовано поле ввода размера требуемого массива. Далее, происходит генерация значений с помощью функции random () и проверка на уникальность значения. В конце данные записываются в файл.

# 3. тестирование

## 3.1. Вид программы

Вид программы после запуска представлен на рис. 2.

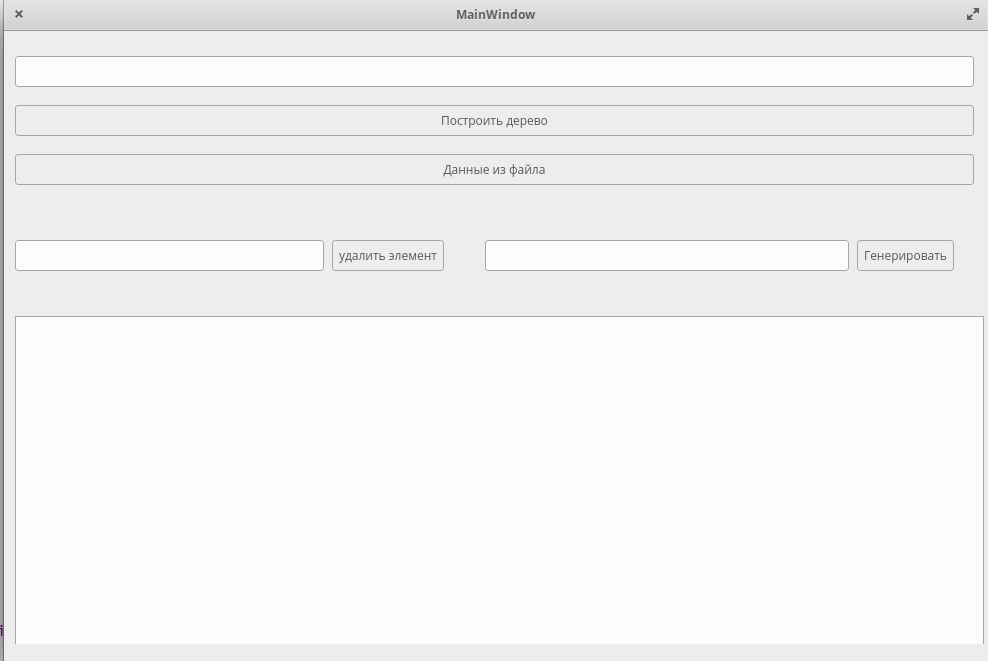


Рисунок 2 – Вид программы после запуска

Вид программы после создания дерева представлен на рис. 3.

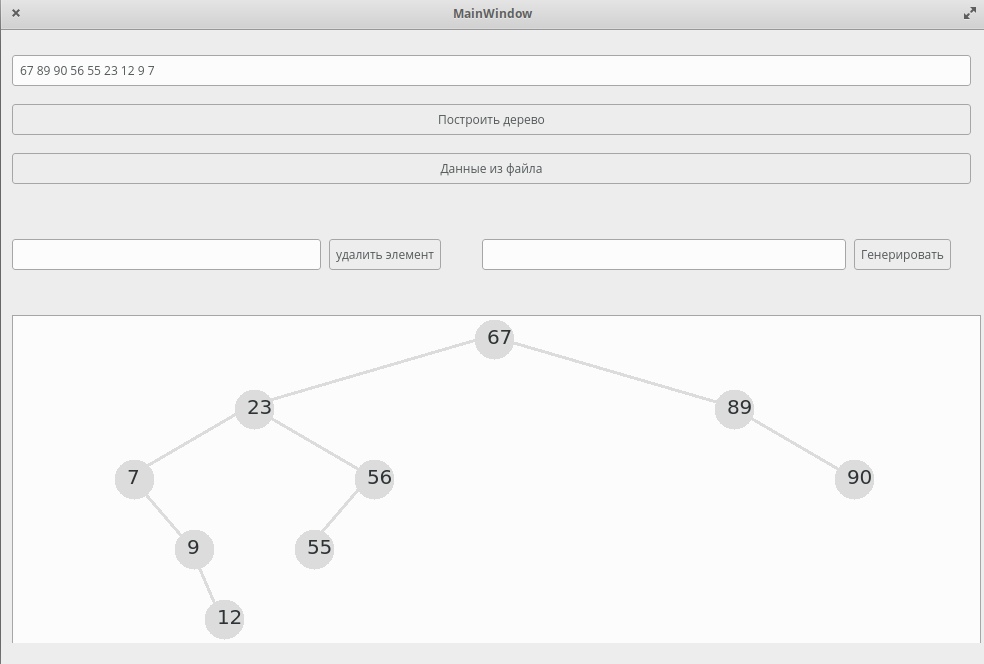


Рисунок 3 – Вид программы после создания дерева

# 4. Исследование

## 4.1. План экспериментального исследования.

Для проведения исследования сложности алгоритма вставки в рандоминизированное бдп необходимо понимать, что на одном и том же наборе данных при повторном запуске будут получаться различные бдп. Следовательно, необходимо провести исследование не только для различного количества исходных данных, но и для одним и тех же. Опорным элементом при исследовании алгоритма станет высота получаемого дерева и число совершенных итераций. По аналогичному плану будет проводиться и исследование алгоритма удаления. После накопления данных необходимо провести сравнение результатов и сделать выводы об эффективности алгоритмов. Кроме того, объединив всю статистику, следует сравнить полученные экспериментальные зависимости от теоретических и сделать выводы о сложности алгоритма вставки в среднем и худшем случаях.

План проведения исследования:

* Получение информации о зависимости высоты дерева, числа итераций и количестве элементов, вставляемых в корень
  + 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 вставки и удаления.

## 4.2. Исследование зависимостей от полученной высоты дерева для алгоритма вставки

Так как при построении бдп вставка в корень определяется рандомно, то при одних и тех же значениях полученные деревья будет различаться. Был проведён ряд тестов: сначала для одного набора данных, а далее построение зависимостей для изменяющего размера входных данных.

С учетом того, что размер входного массива начинался от 500 элементов, его содержимое, а также содержимое файла можно приведено в приложении.

Далее будут представлены только результирующие графики. Для сравнения высоты в одном и том же наборе данных на графике отмечены три точки: максимальная, минимальная и средняя арифметическая высота дерева. Также построен теоретическое значение получаемого результата для большей наглядности. Теоретическая функция - .

Таблица 7 – Легенда

|  |  |
| --- | --- |
| Цвет | Значение |
| Красный | Максимальное значение |
| Синий | Среднее значение |
| Жёлтый | Наименьшее значение |
| Зелёный | Теоретическое значение |

Для массива, состоящего из 500 элементов, результат представлен в табл. 7.

Таблица 8 – Результаты тестирования

|  |  |  |
| --- | --- | --- |
| Максимальная высота | Минимальная высота | Средняя высота |
| 23 | 16 | 19 |

Важно отметить, что количество бдп, не превышающих среднюю высоту, составило 26 из 50 возможных. Далее был построен сравнительный график, где представленная теоретическая функция - . Кроме того, в полученных результатах значение максимальной высоты дерева встречалось единожды, в то время как минимальное шесть раз. Полученные данные представлены на рис. 4.

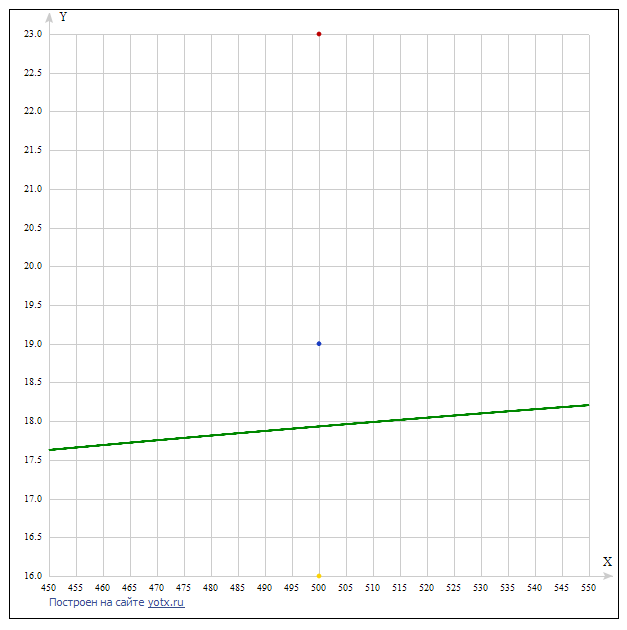


Рисунок 4 – Результаты тестирования 8 относительно теоретических значений

Для массива, состоящего из 1000 элементов, результат представлен в табл.9.

Таблица 9 – Результаты тестирования

|  |  |  |
| --- | --- | --- |
| Максимальная высота | Минимальная высота | Средняя высота |
| 25 | 16 | 20 |

Важно отметить, что количество бдп, лежащих в диапозоне от средней высоты до минимальной, составило 31 из 50 возможных. Кроме того, в полученных результатах значение максимальной высоты дерева встречалось единожды.

Для массива, состоящего из 2000 элементов, результат представлен в табл.10.

Таблица 10 – Результаты тестирования

|  |  |  |
| --- | --- | --- |
| Максимальная высота | Минимальная высота | Средняя высота |
| 31 | 21 | 23 |

Важно отметить, что количество бдп, лежащих в диапозоне от средней высоты до минимальной, составило 32 из 50 возможных. Кроме того, в полученных результатах значение максимальной и минимальной высоты дерева встречалось единожды.

Для массива, состоящего из 3000 элементов, результат представлен в табл.11.

Таблица 11 – Результаты тестирования

|  |  |  |
| --- | --- | --- |
| Максимальная высота | Минимальная высота | Средняя высота |
| 33 | 23 | 27 |

Важно отметить, что количество бдп, не превышающих среднюю высоту, составило 35 из 50 возможных. Кроме того, в полученных результатах значение максимальной высоты дерева встречалось единожды, в то время как минимальное четырежды. Полученные данные представлены на рис. 5.

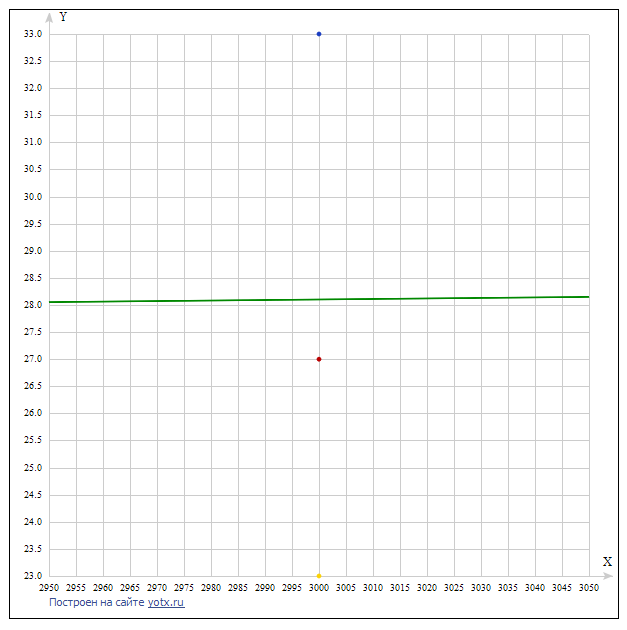


Рисунок 5 – Результаты тестирования 11 относительно теоретических значений

Для массива, состоящего из 4000 элементов, результат представлен в табл.12.

Таблица 12 – Результаты тестирования

|  |  |  |
| --- | --- | --- |
| Максимальная высота | Минимальная высота | Средняя высота |
| 33 | 24 | 28 |

Важно отметить, что количество бдп, лежащих в диапозоне от средней высоты до минимальной, составило 35 из 50 возможных. Кроме того, в полученных результатах значение максимальной высоты дерева встречалось единожды, в то время как минимальное два раза.

Для массива, состоящего из 5000 элементов, результат представлен в табл.13.

Таблица 13 – Результаты тестирования

|  |  |  |
| --- | --- | --- |
| Максимальная высота | Минимальная высота | Средняя высота |
| 34 | 24 | 28 |

Важно отметить, что количество бдп, не превышающих среднюю высоту, составило 33 из 50 возможных.

Для массива, состоящего из 7000 элементов, результат представлен в табл.14.

Таблица 14 – Результаты тестирования

|  |  |  |
| --- | --- | --- |
| Максимальная высота | Минимальная высота | Средняя высота |
| 34 | 25 | 29 |

Важно отметить, что количество бдп, не превышающих среднюю высоту, составило 27 из 50 возможных. Полученные данные представлены на рис. 6.

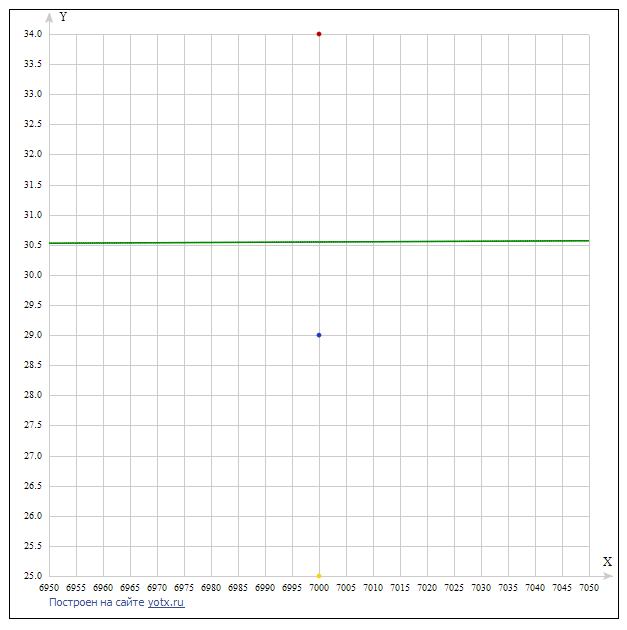


Рисунок 6 – Результаты тестирования 14 относительно теоретических значений

Далее, были получены результаты для набора данных размером от 500 до 7000 элементов с шагом 500, для построения общей картины. Полученные результаты сведены в общую таблицу (табл. 15) и на её основе было построено 4 графика: максимальная, минимальная, средняя и теоретическая высота (рис. 7 и рис.8).

Таблица 15 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| Размер массива | Минимальная высота | Средняя высота | Максимальная высота |
| 500 | 16 | 19 | 23 |
| 1000 | 16 | 20 | 25 |
| 1500 | 21 | 23 | 28 |
| 2000 | 21 | 23 | 31 |
| 2500 | 22 | 25 | 31 |
| 3000 | 23 | 27 | 33 |

Продолжение таблицы 15

|  |  |  |  |
| --- | --- | --- | --- |
| 3500 | 22 | 27 | 32 |
| 4000 | 24 | 28 | 33 |
| 4500 | 25 | 28 | 33 |
| 5000 | 24 | 28 | 34 |
| 5500 | 25 | 29 | 36 |
| 6000 | 25 | 29 | 36 |
| 6500 | 26 | 29 | 37 |
| 7000 | 25 | 29 | 34 |
| 27000 | 32 | 35 | 44 |
| 90000 | 36 | 40 | 45 |
| 150000 | 39 | 42 | 48 |
| 340000 | 42 | 45 | 50 |
| 750000 | 46 | 49 | 56 |
| 1000000 | 46 | 50 | 56 |

На основании данных, представленных в табл. 15, были построены два графика: зависимость высоты дерева от размера исходного массива для небольших значений и для всех значений таблицы.

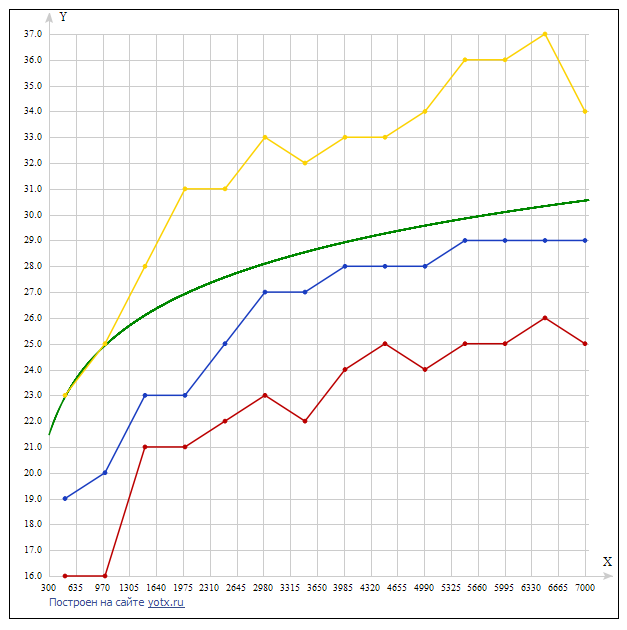


Рисунок 7 – График зависимости высоты дерева при построении дерева для размера массива, не превыщающего 7000

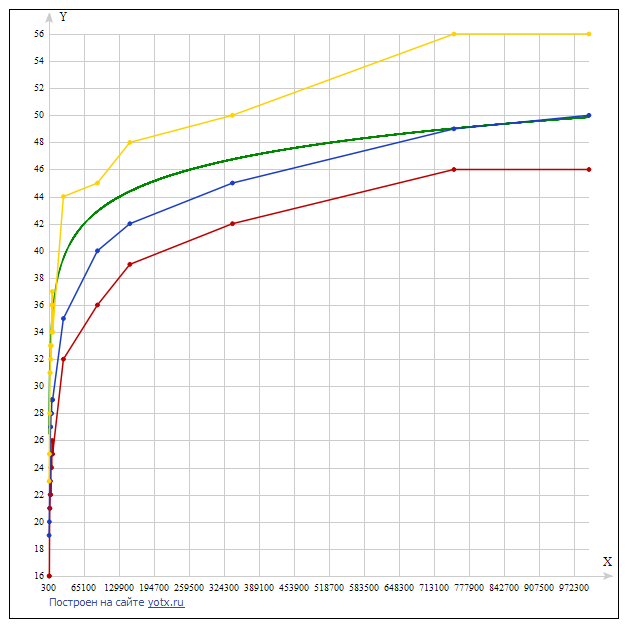


Рисунок 8 – График зависимости высоты дерева при построении дерева

Исходя из полученных результатов можно увидеть, что сложность алгоритма вставки является логарифмической. Стоит отметить, что худший случай сгенерировать самостоятельно не получится, так как алгоритм использует рандоминизацию для определения вероятности вставки в корень и, далее, высота дерева фиксируется. Это приводит к тому, что вероятность получения несбарансированного дерева оказывается пренебрежимо малой при больших размерах деревьев. Как можно видеть из полученных результатов максимальная высота дерева также имеет логарифмическую сложность, но при других константах.

## 4.3. Исследование зависимостей от количества итераций для алгоритма вставки

Был проведен ряд тестов, где переменным параметром бралось количество итераций, необходимое для построения бинарного дерева поиска. Также, как и в прошлом подпункте сначала исследовалось количество итераций на одном и том же наборе исходных данных, а затем общие результаты сводились в таблицу.

Ниже представлены табл. 16 и рис. 9, рис. 10, иллюстрирующие полученные результаты.

Таблица 16 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| Размер массива | Минимальное число итераций | Среднее арифметическое число итераций | Максимальное число итераций |
| 500 | 5432 | 5718 | 6184 |
| 1000 | 11850 | 12651 | 13993 |
| 1500 | 19077 | 20235 | 21374 |
| 2000 | 26659 | 28182 | 29487 |
| 2500 | 34764 | 36222 | 39661 |
| 3000 | 42567 | 44721 | 46994 |

Продолжение таблицы 16

|  |  |  |  |
| --- | --- | --- | --- |
| 3500 | 49309 | 52970 | 58127 |
| 4000 | 58815 | 61908 | 65222 |
| 4500 | 67411 | 70094 | 75059 |
| 5000 | 75867 | 78992 | 84705 |
| 5500 | 83954 | 88761 | 98453 |
| 6000 | 92722 | 97488 | 105155 |
| 6500 | 101613 | 107615 | 113993 |
| 7000 | 111698 | 116621 | 122881 |
| 27000 | 504724 | 519154 | 543403 |
| 90000 | 1015481 | 1104609 | 1235836 |
| 150000 | 1737703 | 1903858 | 2191343 |
| 340000 | 4330971 | 4671355 | 5082435 |
| 750000 | 9926353 | 10863589 | 11919314 |
| 1000000 | 13443080 | 14716258 | 16020418 |

На основании данных, представленных в табл. 16, были построены два графика: зависимость итераций от небольших размеров исходных данных и общий график для всех значений таблицы. Первый график был построен для удобства анализирования.

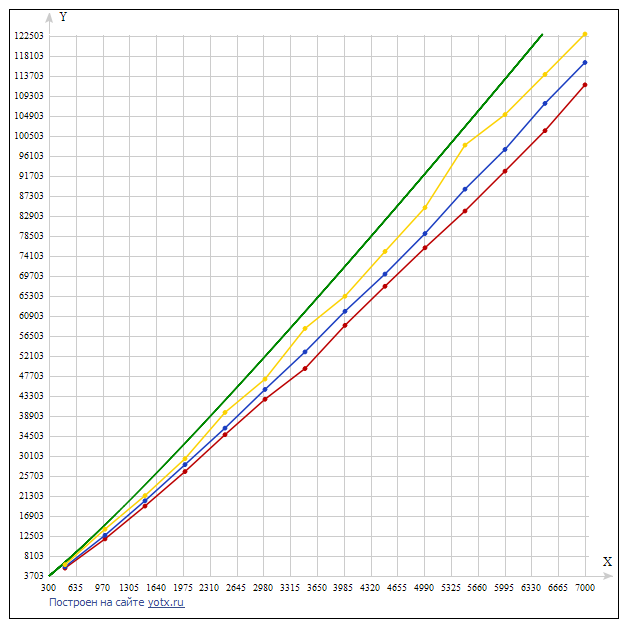


Рисунок 9 – График зависимости количества итераций от размера исходных данных

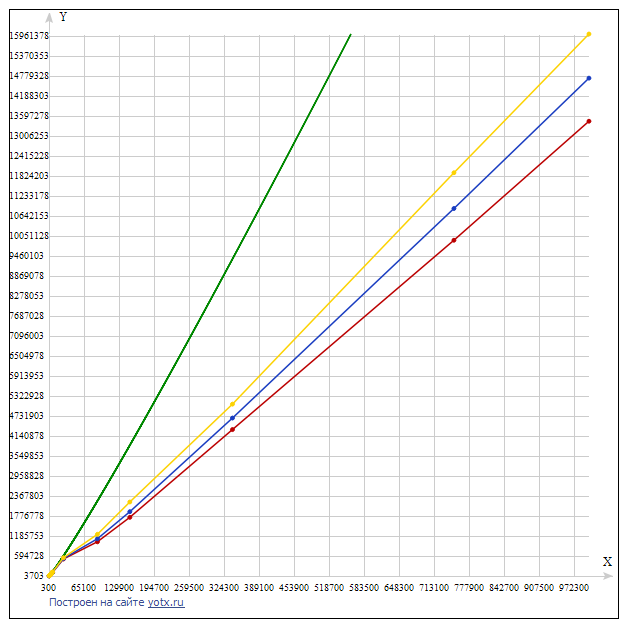


Рисунок 10 – График зависимости количества итераций от размера исходных данных

Количество итераций необходимое для вставки одного элемента в дерево составляет . Учитывая, что дерево имеет размер , итоговая сложность алгоритма вставки для всего дерева составит

Из графиков можно увидеть, что количество итераций даже в своём наибольшем количестве не превосходит указанную сложность. Опять же, стоит отметить, что худший случай был взят во время исследования алгоритма на одном и том же наборе данных.

В целом можно сделать вывод, что алгоритм вставки в рандоминизированном бинарном дереве поиска является крайне эффективным и с большой вероятностью строит сбалансированное дерево, избегая самого худшего случая – вырождения дерева в односвязный список.

## 4.4. Исследование зависимостей значений, вставленных в корень и размеров исходных данных

В ходе тестов собирались данных о количестве элементов исходного массива, для которых выполнилась вставка в корень. Также, как и в прошлом подпункте сначала исследовалось количество таких значений на одном и том же наборе исходных данных, а затем общие результаты сводились в таблицу. Результирующая табл. 17 представлена ниже.

Таблица 17 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| Размер массива | Минимальное число элементов | Среднее арифметическое число элементов | Максимальное число элементов |
| 500 | 322 | 350 | 372 |
| 1000 | 667 | 699 | 731 |
| 2000 | 1349 | 1401 | 1442 |
| 4000 | 2720 | 2802 | 2891 |
| 6000 | 4091 | 4198 | 4272 |
| 7000 | 4817 | 4907 | 4982 |

Продолжение таблицы 17

|  |  |  |  |
| --- | --- | --- | --- |
| 27000 | 18714 | 18937 | 19097 |
| 90000 | 48125 | 48512 | 48687 |
| 150000 | 80586 | 80846 | 81171 |
| 340000 | 182747 | 183194 | 183599 |
| 750000 | 403487 | 404202 | 404992 |
| 1000000 | 538236 | 538997 | 539910 |

Можно отметить, что большая часть элементов исходного массива вставляется в корень, что говорит о достаточно сбалансированном дереве.

## 4.5. Исследование зависимостей от количества итераций для алгоритма удаления

Для проведения тестов генерировался индекс числа, которое требовалось удалить и затем запускался ряд тестов для одного и того же набора значений. Так как генерируемое каждый раз дерево получалось различным, выбранный элемент также менял своё местоположение. На основе этих данных была сведена таблица 18, где указано наибольшее, наименьшее и среднее арифметическое число итераций.

Таблица 18 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| Размер массива | Минимальное число итераций | Среднее арифметическое число итераций | Максимальное число итераций |
| 500 | 13 | 19 | 26 |
| 1000 | 12 | 22 | 31 |
| 1500 | 12 | 18 | 28 |
| 2000 | 14 | 25 | 31 |
| 2500 | 15 | 22 | 29 |

Продолжение таблицы 18

|  |  |  |  |
| --- | --- | --- | --- |
| 3000 | 13 | 24 | 33 |
| 3500 | 9 | 18 | 28 |
| 4000 | 19 | 28 | 32 |
| 4500 | 12 | 28 | 37 |
| 5000 | 14 | 23 | 30 |
| 5500 | 19 | 27 | 36 |
| 6000 | 14 | 26 | 34 |
| 6500 | 14 | 22 | 28 |
| 7000 | 14 | 24 | 29 |
| 27000 | 16 | 23 | 33 |
| 90000 | 14 | 20 | 28 |
| 150000 | 18 | 24 | 31 |
| 340000 | 17 | 22 | 31 |
| 750000 | 17 | 27 | 36 |
| 1000000 | 19 | 25 | 33 |

На основании данных, представленных в табл. 18, были построены два графика: зависимость итераций от небольших размеров исходных данных и общий график для всех значений таблицы. Первый график был построен для удобства анализирования (рис. 11 и рис. 12 соотвественно)

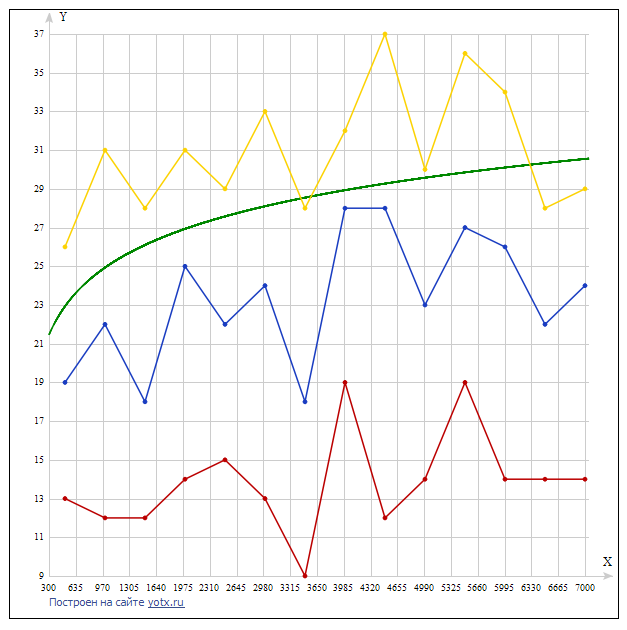


Рисунок 11 – График зависимости количества итераций от размера исходных данных

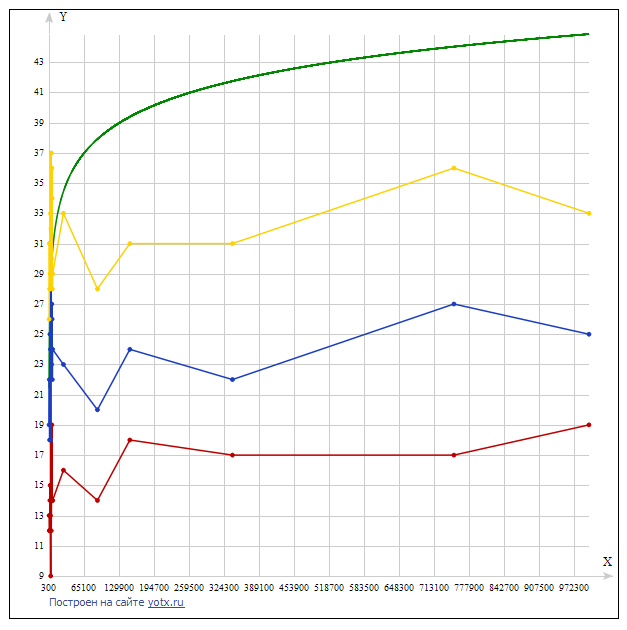


Рисунок 12 – График зависимости количества итераций от размера исходных данных

В результате можно сделать вывод, что несмотря на довольно резкие скачки получаемых значений (происходящие из-за случайной генерации удаляемого числа и повторного запуска теста для этого же набора данных: что меняет его положение в дереве), алгоритм удаления одного элемента из бинарного дерева поиска обладает логарифмической сложностью.

## 4.6. Выводы об исследовании алгоритма

В результате исследования было подтвержденно, что сложность алгоритмов вставки и удаления в рандоминизированном бинарном дереве поиска не превышает логарифмическую. Использование такого бинарного дерева поиска позволит избежать полностью несбалансированных случаев, те вырождения дерева в односвязный список.

Одним из основополагающих моментов эффективного поиска в таком бинарном дереве (для дальнейшего удаления элемента, к примеру) является правило хранения ключей: любой ключ в левом поддереве меньше корневого ключа, а в правом поддереве — больше корневого ключа.

Следует отметить, что значительную роль в построении такого дерева играет генерация вероятностей: именно от неё зависит наиболее оптимальная высота получаемого дерева. Тем не менее, как показывает исследование, даже худший случай получаемого значения высоты не является критическим и дерево продолжает оставаться достаточно сбалансированным.

# Заключение

В ходе выполнения курсовой работы была разработана программа, которая обладает следующей функциональностью: создание файла, содержащего входные данные для построения бинарного дерева поиска, создание рандоминизированного бинарного дерева поиска по заданному файлу, удаление выбранного пользователем элемента, а также запись результатов исследований в файл. С помощью программы было проведено исследование различных случаев алгоритма вставки в бинарное дерево и алгоритма удаления заданного значения. В ходе исследования была выявлена зависимость эффективности алгоритма от различных параметров. В результате было выявлено, что на эффективность влияет генерация вероятности вставки в корень, так как в дальнейшем это определяет дальнейшее построение дерева. В среднем случае дерево получается хорошо сбалансированным, особенно на больших объёмах данных.

# Список использованных источников

1. Bjarne Stroustrup. A Tour of C++. М.: Addison-Wesley, 2018. 217 с.
2. Treap // GeeksforGeeks <https://www.geeksforgeeks.org/treap-a-randomized-binary-search-tree/> (дата обращения: 18.12.2000)
3. Qt Documentation // Qt. URL: <https://doc.qt.io/qt-5/index.html> (дата обращения: 18.12.2000)
4. Рандомизированные деревья поиска URL: <https://habr.com/ru/post/145388/#reed> (дата обращения: 17.12.2000)
5. The Height of a Random Binary Search Tree // BRUCE REED URL: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.152.1289&rep=rep1&type=pdf> (дата обращения: 18.12.2000)

# Приложение А Исходный код программы. MAIN.cpp

#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. mainwindow.h

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QMainWindow>

#include <QGraphicsItem>

#include <QGraphicsView>

#include <QGraphicsEffect>

namespace Ui {

class MainWindow;

}

class MainWindow : public QMainWindow

{

Q\_OBJECT

public:

explicit MainWindow(QWidget \*parent = 0);

~MainWindow();

private slots:

void on\_printTree\_clicked();

void on\_delete\_elem\_clicked();

void on\_choose\_file\_clicked();

private:

Ui::MainWindow \*ui;

QGraphicsScene \*scene;

};

#endif // MAINWINDOW\_H

# Приложение В Исходный код программы. mainwindow.cPP

#include "mainwindow.h"

#include "ui\_mainwindow.h"

#include<bin\_struct.h>

MainWindow::MainWindow(QWidget \*parent) :

QMainWindow(parent),

ui(new Ui::MainWindow)

{

ui->setupUi(this);

scene = new QGraphicsScene;

ui->graphicsView->setScene(scene);

}

MainWindow::~MainWindow()

{

delete ui;

}

void MainWindow::on\_printTree\_clicked()

{

QString data = ui->input\_tree->text();

QStringList abc = data.split(' ');

int\* mas = new int[100];

int i =0;

bool go = true;

for (auto x:abc){

bool convertOK;

x.toInt(&convertOK);

if(convertOK == false){

go = false;

}

else{

mas[i] = x.toInt();

i++;

}

}

if(!go){

QMessageBox::warning(this,"Error", "Incorrect input");

return;

}

BinTree\* BT = new (BinTree);

for(int j = 0; j < i; j++){

BT->Head = BT->insert(BT->Head, mas[j]);

}

int depth = BT->max\_depth(BT->Head);

graphic(BT, scene, depth);

}

void MainWindow::on\_delete\_elem\_clicked()

{

QString data = ui->input\_tree->text();

QStringList abc = data.split(' ');

int\* mas = new int[100];

int i =0;

bool go = true;

for (auto x:abc){

bool convertOK;

x.toInt(&convertOK);

if(convertOK == false){

go = false;

}

else{

mas[i] = x.toInt();

i++;

}

}

if(!go){

QMessageBox::warning(this,"Error", "Incorrect input");

return;

}

BinTree\* BT = new (BinTree);

for(int j = 0; j < i; j++){

BT->Head = BT->insert(BT->Head, mas[j]);

}

QString elem\_ = ui->input\_del->text();

bool convert\_;

int elem = elem\_.toInt(&convert\_);

if(!convert\_){

QMessageBox::warning(this,"Error", "Incorrect input");

return;

}

BT->Head = BT->remove(BT->Head, elem);

int depth = BT->max\_depth(BT->Head);

QString out;

for(int l = 0; l < i - 1; l++){

if (mas[l] == elem){

for(int k = l; k < i - 1; k++){

mas[k] = mas[k + 1];

}

}

out.append(QString::number(mas[l]));

if(l != i -2) out.append(" ");

}

ui->input\_tree->setText(out);

graphic(BT, scene, depth);

}

void MainWindow::on\_choose\_file\_clicked()

{

QString file\_name = QFileDialog().getOpenFileName();

if(!file\_name.isNull()){

QFile file(file\_name);

if(file.open(file.ReadOnly)){

QString data = file.readAll();

if (data == "") {

QMessageBox::critical(this, "Error!", "Р’РІРµРґРёС‚Рµ РґРµСЂРµРІРѕ");

return;

}

else{

QStringList abc = data.split(' ');

int\* mas = new int[100];

int i =0;

bool go = true;

for (auto x:abc){

bool convertOK;

x.toInt(&convertOK);

if(convertOK == false){

go = false;

}

else{

mas[i] = x.toInt();

i++;

}

}

if(!go){

QMessageBox::warning(this,"Error", "Incorrect input");

return;

}

BinTree\* BT = new (BinTree);

for(int j = 0; j < i; j++){

BT->Head = BT->insert(BT->Head, mas[j]);

}

int depth = BT->max\_depth(BT->Head);

graphic(BT, scene, depth);

}

}

}

else QMessageBox::warning(this,"Error", "Not Found");

}

# Приложение Г Исходный код программы. bin\_struct.H

#ifndef BIN\_STRUCT\_H

#define BIN\_STRUCT\_H

#include <QGraphicsItem>

#include <QGraphicsView>

#include <QGraphicsEffect>

#include <QString>

#include <QFileDialog>

#include <QMessageBox>

#include <QTextEdit>

#include <QMainWindow>

#include <QStandardPaths>

#include <QtGui>

#include <QColorDialog>

#include <QInputDialog>

#include <QPushButton>

#include <QStringList>

struct Node // СЃС‚СЂСѓРєС‚СѓСЂР° РґР»СЏ РїСЂРµРґСЃС‚Р°РІР»РµРЅРёСЏ СѓР·Р»РѕРІ РґРµСЂРµРІР°

{

int key;

int size;

Node\* left;

Node\* right;

Node(int k) { key = k; left = right = 0; size = 1; }

};

class BinTree

{

private:

Node\* Current = nullptr;

public:

Node\* Head = nullptr;

BinTree();

Node\* insert(Node\* p, int k);

Node\* insertroot(Node\* p, int k);

Node\* rotateright(Node\* p);

Node\* rotateleft(Node\* q);

int max\_depth(Node \*hd);

int getsize(Node\* p);

void fixsize(Node\* p);

Node\* join(Node\* p, Node\* q);

Node\* remove(Node\* p, int k);

};

int treePainter(QGraphicsScene \*scene, Node \*node, int w, int h, int wDelta, int hDelta, QPen &pen, QBrush &brush, QFont &font, int depth, int c);

QGraphicsScene \*graphic(BinTree \*tree, QGraphicsScene \*scene, int depht);

#endif // BIN\_STRUCT\_H

# Приложение Д Исходный код программы. TREE\_fun.cpp

#include <bin\_struct.h>

BinTree::BinTree(){

Head = nullptr;

Current = Head;

}

int BinTree::max\_depth(Node \*hd){

if (hd == NULL) return 0;

else{

int lDepth = max\_depth(hd->left);

int rDepth = max\_depth(hd->right);

if (lDepth > rDepth) return(lDepth + 1);

else return(rDepth + 1);

}

}

int BinTree::getsize(Node\* p) // РѕР±РµСЂС‚РєР° РґР»СЏ РїРѕР»СЏ size, СЂР°Р±РѕС‚Р°РµС‚ СЃ РїСѓСЃС‚С‹РјРё РґРµСЂРµРІСЊСЏРјРё (t=NULL)

{

if( !p ) return 0;

return p->size;

}

void BinTree::fixsize(Node\* p) // СѓСЃС‚Р°РЅРѕРІР»РµРЅРёРµ РєРѕСЂСЂРµРєС‚РЅРѕРіРѕ СЂР°Р·РјРµСЂР° РґРµСЂРµРІР°

{

p->size = getsize(p->left)+getsize(p->right)+1;

}

Node\* BinTree::rotateright(Node\* p) // РїСЂР°РІС‹Р№ РїРѕРІРѕСЂРѕС‚ РІРѕРєСЂСѓРі СѓР·Р»Р° p

{

Node\* q = p->left;

if( !q ) return p;

p->left = q->right;

q->right = p;

q->size = p->size;

fixsize(p);

return q;

}

Node\* BinTree::rotateleft(Node\* q) // Р»РµРІС‹Р№ РїРѕРІРѕСЂРѕС‚ РІРѕРєСЂСѓРі СѓР·Р»Р° q

{

Node\* p = q->right;

if( !p ) return q;

q->right = p->left;

p->left = q;

p->size = q->size;

fixsize(q);

return p;

}

Node\* BinTree::insertroot(Node\* p, int k) // РІСЃС‚Р°РІРєР° РЅРѕРІРѕРіРѕ СѓР·Р»Р° СЃ РєР»СЋС‡РѕРј k РІ РєРѕСЂРµРЅСЊ РґРµСЂРµРІР° p

{

if( !p ) return new Node(k);

if( k < p->key )

{

p->left = insertroot(p->left,k);

return rotateright(p);

}

else

{

p->right = insertroot(p->right,k);

return rotateleft(p);

}

}

Node\* BinTree::insert(Node\* p, int k) // СЂР°РЅРґРѕРјРёР·РёСЂРѕРІР°РЅРЅР°СЏ РІСЃС‚Р°РІРєР° РЅРѕРІРѕРіРѕ СѓР·Р»Р° СЃ РєР»СЋС‡РѕРј k РІ РґРµСЂРµРІРѕ p

{

if( !p ) return new Node(k);

if( rand()%(p->size+1)==0 )

return insertroot(p,k);

if( p->key>k )

p->left = insert(p->left,k);

else

p->right = insert(p->right,k);

fixsize(p);

return p;

}

Node\*BinTree:: join(Node\* p, Node\* q) // РѕР±СЉРµРґРёРЅРµРЅРёРµ РґРІСѓС… РґРµСЂРµРІСЊРµРІ

{

if( !p ) return q;

if( !q ) return p;

if( rand()%(p->size+q->size) < p->size )

{

p->right = join(p->right,q);

fixsize(p);

return p;

}

else

{

q->left = join(p,q->left);

fixsize(q);

return q;

}

}

Node\*BinTree::remove(Node\* p, int k) // СѓРґР°Р»РµРЅРёРµ РёР· РґРµСЂРµРІР° p РїРµСЂРІРѕРіРѕ РЅР°Р№РґРµРЅРЅРѕРіРѕ СѓР·Р»Р° СЃ РєР»СЋС‡РѕРј k

{

if( !p ) return p;

if( p->key==k )

{

Node\* q = join(p->left,p->right);

delete p;

return q;

}

else if( k<p->key )

p->left = remove(p->left,k);

else

p->right = remove(p->right,k);

return p;

}

# Приложение Е Исходный код программы. main\_fun.h

#ifndef MAIN\_FUN\_H

#define MAIN\_FUN\_H

#include <bin\_struct.h>

int treePainter(QGraphicsScene \*scene, Node \*node, int w, int h, int wDelta, int hDelta, QPen &pen, QBrush &brush, QFont &font, int depth);

QGraphicsScene \*graphic(BinTree \*tree, QGraphicsScene \*scene, int depht);

#endif // MAIN\_FUN\_H

# Приложение Ж Исходный код программы. draw\_Tree.cpp

#include<bintree.h>

#include<functionstree.h>

#include<cmath>

QGraphicsScene \*graphic(BinTree \*tree, QGraphicsScene \*&scene, int depth)

{

if (tree == nullptr)

return scene;

scene->clear();

QPen pen;

QColor color;

color.setRgb(220, 220, 220);

pen.setColor(color);

QBrush brush (color);

QFont font;

font.setFamily("Tahoma");

pen.setWidth(3);

int wDeep = static\_cast<int>(pow(2, depth + 2));

int hDelta = 70;

int wDelta = 15;

font.setPointSize(wDelta);

int width = (wDelta\*wDeep)/2;

treePainter(scene, tree->Head, width/2, hDelta, wDelta, hDelta, pen, brush, font, wDeep);

return scene;

}

int treePainter(QGraphicsScene \*&scene, Node \*node, int w, int h, int wDelta, int hDelta, QPen &pen, QBrush &brush, QFont &font, int depth)

{

if ((node == nullptr) || (node->data == '^'))

return 0;

QString out;

out += node->data;

QGraphicsTextItem \*textItem = new QGraphicsTextItem;

textItem->setPos(w, h);

textItem->setPlainText(out);

textItem->setFont(font);

scene->addEllipse(w-wDelta/2, h, wDelta\*5/2, wDelta\*5/2, pen, brush);

if ((node->left != nullptr) && (node->left->data != '^') )

scene->addLine(w+wDelta/2, h+wDelta, w-(depth/2)\*wDelta+wDelta/2, h+hDelta+wDelta, pen);

if (node->right != nullptr)

scene->addLine(w+wDelta/2, h+wDelta, w+(depth/2)\*wDelta+wDelta/2, h+hDelta+wDelta, pen);

scene->addItem(textItem);

treePainter(scene, node->left, w-(depth/2)\*wDelta, h+hDelta, wDelta, hDelta, pen, brush, font, depth/2);

treePainter(scene, node->right, w+(depth/2)\*wDelta, h+hDelta, wDelta, hDelta, pen, brush, font, depth/2);

return 0;

}