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

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Рандомизированные дерамиды поиска – вставка и исключение. Демонстрация

Студент гр. 9304 _____ Арутюнян В.В.

Преподаватель _____ Филатов Ар.Ю.

Санкт-Петербург

2020

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент Арутюнян В.В.

Группа 9304

Тема работы: Рандомизированные дерамиды поиска – вставка и исключение. Демонстрация

Исходные данные:

Содержание пояснительной записки:

- Аннотация
- Содержание
- Введение
- Формальная постановка задачи
- Описание структур данных и функций, описание алгоритма
- Тестирование / Демонстрация
- Заключение
- Список использованных источников

Предполагаемый объем пояснительной записки:

Не менее 24 страниц.

Дата выдачи задания: 23.11.2020

Дата сдачи реферата: 28.12.2020

Дата защиты реферата: 28.12.2020

Студент		Арутюнян В.В.
Преподаватель		Филатов Ар.Ю.

АННОТАЦИЯ

В данной курсовой работе производится демонстрация операций вставка и исключение элементов в структуре данных рандомизированная дерамида поиска. Демонстрация сопровождается наличием интерактивного меню, с помощью которого удобно взаимодействовать с соответствующей структурой данных.

SUMMARY

This course work demonstrates the operations of inserting and excluding elements in the data structure of a randomized search deramide. The demonstration is accompanied by the presence of an interactive menu, with the help of which it is convenient to interact with the corresponding data structure.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы: Изучить структуру данных рандомизированная дерамида поиска. Демонстрация операций вставки и удаления элементов.

Задача: Реализовать программу, считывающую и обрабатывающую команды управления и элементы посредством командной строки.

1. ФОРМАЛЬНАЯ ПОСТАНОВКА ЗАДАЧИ

Реализовать структуру данных рандомизированная дерамида поиска и выполнить демонстрацию операций вставки и удаления. Предоставить некоторый функционал для управления и выполнения необходимых операций над деревом.

2. ХОД ВЫПОЛНЕНИЯ РАБОТЫ

2.1. Описание алгоритма

Для использования рандомизированной дерамиды поиска требуется несколько вспомогательных методов: Split(), Merge().

Split()

Метод Split() разрезает исходное дерево tree по значению data. Возвращает указатель на элемент типа Tree, где потомками являются две дерамиды поиска, которые и необходимо было получить.

Пусть необходимо разрезать дерево по значению data, которое больше, чем значение в корне. Назовем исходное дерево tree, а возвращаемые деревья – left и right, где в left находятся все элементы со значениями меньше, чем data, а в right – все остальные.

Левое поддерево left будет совпадать с левым поддеревом tree. Поэтому для нахождения правого поддерева left и всего right, необходимо разрезать правое поддерево tree на left“ и right“ по значению data. Тогда left“ будет правым поддеревом left, а right совпадет с right“.

Если data меньше (или равно) значения в корне, то ситуация рассматривается симметрично.

Merge()

Метод Merge() сливает два дерева left и right в одно tree.

Корнем tree должна стать вершина из left или right с наибольшим приоритетом priority. Так как наибольший приоритет в дереве находится в корне, то корнем tree станет либо корень left, либо корень right.

Пусть priority больше у left. Тогда левое поддерево tree совпадет с левым поддеревом left. Правым поддеревом будет Merge() правого поддерева left и дерева right.

Ситуация, когда priority больше у right рассматривается симметрично.

Также присутствует несколько основных методов, которые необходимы для реализации операций вставки и удаления.

Insert()

Метод Insert() добавляет в дерево tree элемент elem, elem.priority – приоритет элемента, elem.data – данные элемента.

Сначала необходимо, спускаясь по дереву, найти первый элемент, в котором значение приоритета priority окажется меньше, чем elem.priority. После достаточно разбить дерево от найденного элемента с помощью Split(), полученные left и right подвязать как потомков к elem, а elem вставить на место найденного элемента в дереве. Тем самым новый элемент вставится в дерамиду, а её некоторая переформируется в ходе выполнения данной операциии.

Remove()

Метод Remove() удаляет из дерева tree элемент со значением data.

Сначала необходимо, спускаясь по дереву, найти удаляемый элемент del. В случае, когда data меньше значения данных на текущем рассматриваемом элементе дерамиды, то спуск производится к левому потомку, в ином случае – к правому. Далее необходимо с помощью Merge() слить потомков del, а результат слияния вставить вместо удаляемого элемента del. Таким образом удаляемый элемент просто заменится переформированными потомками.

Еще несколько полезных методов: Count(), CountAndThenInsert(), GetHeight():

Count()

Метод Count() возвращает целое число – количество элементов е типа Elem в дереве tree. Подсчёт элементов в дереве ведется простым рекурсивным обходом с помощью метода RecursiveCount().

CountAndThenInsert()

В данном методе используются уже готовые методы Count() и Insert(). Сначала сохраняется значение, которое вернет Count() для элемента e типа Elem, затем с помощью Insert() элемент e добавляется в дерево. После возвращается сохраненное значение.

GetHeight()

GetHeight() возвращает высоту дерева tree. Подсчёт ведется с помощью рекурсивного обхода в методе GetHeightPrivate().

2.2. Описание структур данных и функций

1. Class Node – шаблонный класс, представляющий собой узел дерева:

   Поля:

   data_ – элемент с данными, которые хранятся в узле дерева, помощью данного поля поддерживаются свойства бинарного дерева в дерамиде поиска;

   priority_ – случайно сгенерированный приоритет в виде целого числа, необходим для поддержания свойств кучи в дерамиде поиска;

   left_ – указатель на левого потомка;

   right_ – указатель на правого потомка.

2. Class Treap – класс, представляющий собой рандомизированную дерамиду поиска:

   Поля:

   head_ – указатель на элемент типа Node, является головным элементом дерамиды поиска.

   Методы:

   Insert() – производит вставку элемента в дерамиду;

   Remove() – производит удаление элемента из дерамиды;

Count() – узнает, в каком количестве присутствует определенный элемент в дереве;

GetHeight() – узнает высоту дерева;

CountAndThenInsert() – аналогично Count(), только после еще вставляет элемент в дерево, по которому изначально производился поиск;

operator<< – вывод дерамиды поиска в удобном виде;

InsertPrivate() – необходим для работы Insert();

RemovePrivate() – необходим для работы Remove();

Split() – метод, выполняющий разделение дерева на два, по определенному значению data. Метод возвращает две дерамиды поиска, где в левой находятся все элементы меньше data, а в правом – все остальные;

Merge() – метод, выполняющий слияние двух дерамид поиска в одну. Возвращает одну дерамиду поиска;

GetHeightPrivate() – необходим для работы GetHeight();

DumpFullBinaryTree() – необходим для работы более удобного вывода всего дерева через перегруженный оператор operator<<;

RecursiveDumpFullBinaryTree() – необходим для работы DumpFullBinaryTree();

RecursiveCount() – необходим для работы Count();

PrintDataAndPriority() – выводит данные и приоритет узла дерамиды;

PaintPartOfTree() – раскрашивает часть дерева, начиная от определенного узла и продолжая всеми потомками;

MaxLenghtOfElement() – возвращает максимальную длину элемента во всём дереве, необходимо для корректного вывода;

PrintFullBinaryTree() – выводит дерамиду с поддержкой различных цветов узлов;

PrintSeparator() – выводит разделитель с различным форматированием;

PrintAndPaintFullBinaryTree() – выводит и расшкрашивает дерамиду, объединяет в себе методы DumpFullBinaryTree(), PaintPartOfTree(), PrintFullBinaryTree();

PrintRemoveInfo() – вывод информацию о текущем удаляемом элементе;

PrintRemoveIfEqual() – вывод дополнительной информации, в случае совпадения значения элемента с элементов дерамиды;

PrintMerge() – вывод дополнительной информации о слиянии дерамид.

3. Class ColoredNode – класс для хранения раскрашенных узлов дерамиды:

Поля:

node – указатель на узел дерамиды;

color – цвет, в который покрашен узел node.

4. Enum Class Color – предназначен для хранения кодов цветов, необходимых для раскрашивания элементов в терминале с помощью escape-последовательностей.

Исходный код представлен в приложении А.

3. ТЕСТИРОВАНИЕ

Программа на вход ожидает строку для анализа сразу после флага “-s”, иначе ожидается путь до файла со строкой для обработки. Возможно совместное использование:

./cw some/path/1 -s “some_string_1” -s “some_string_2” some/path/2

При таком вызове программа обработает строку из файла some/path/1, затем строку some_string_1, после строку some_string_2, далее строку из файла some/path/2.

Далее полученная строка анализируется, в строке могут быть как команды управления, так и сами элементы, над которыми производятся операции. Доступны следующие команды управления:

1. «#Insert» – вставка элемента в дерево;
2. «#Remove» – удаление элемента из дерева, если он существует, иначе команда просто игнорируется;
3. «#Count» – узнать, в каком количестве присутствует элемент е типа Elem в дереве;
4. «#Task» – узнать, в каком количестве присутствует элемент е типа Elem в дереве, а затем добавить его в дерево;
5. «#Print» – вывод текущего дерева;
6. «#Exit» – завершение программы.

Если на вход не подать команды, то по умолчанию используется команда «#Insert», то есть все, введенные элементы далее, будут добавляться в дерамиду.

Пример ввода:

«2 3 1 2 3 5 6 #Remove 3 5 6 #Task 3 3 #Count 3 3 #Insert 0»

ассмотрим, как будет анализироваться данная строка:

1. Так как команды не поступило, все полученные далее элементы просто будут добавляться в дерево. То есть элементы «2», «3», «1», «2», «3», «5», «6». На рисунке 1 продемонстрировано, как выглядит дерево послед данного этапа.

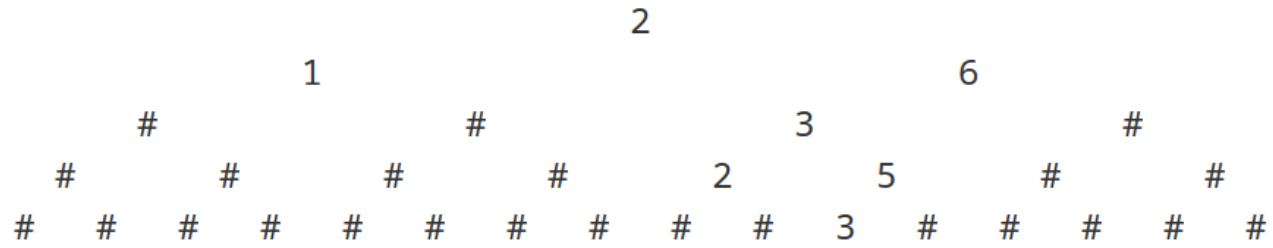


Рисунок 1 – Представление дерева на первом шаге

2. Далее идёт команда «#Remove», а элементы дальше будут удаляться из дерева, если это возможно, либо команда проигнорируется. То есть будет попытка удалить элементы «3», «5», «6». Все три попытки успешно выполнятся, так как такие элементы в дереве уже присутствовали. На рисунке 2 продемонстрировано, как выглядит дерево послед данного этапа.

Рисунок 2 – Представление дерева на втором шаге

3. Поступает команда «#Task», далее для каждого элемента e типа Elem выполнится проверка, которая покажет, в каком количестве присутствует данный элемент в дереве. А затем добавится элемент e в дерево. В ходе работы были получены две строки: «Count = 1 (для элемента "3")» и «Count = 2 (для элемента "3")». Первая говорит о том, что в дереве есть элемент «3» в количестве 1 шт., а вторая – существует элемент «3» в двух экземплярах. На рисунке 3 продемонстрировано, как выглядит дерево послед данного этапа.

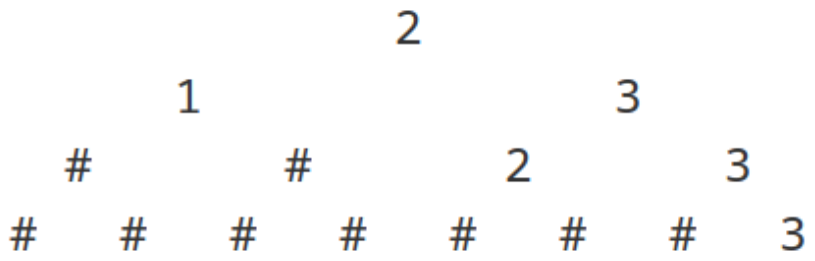


Рисунок 3 – Представление дерева на третьем шаге

4. После следует команда «#Count». Далее для соответствующих элементов вернется, в каком количестве они присутствуют. В ходе работы были получены следующие строки: «Count = 3 (для элемента "3")» и «Count = 3 (для элемента "3")». Первая сообщает, что элемент «3» присутствует в дереве в количестве 3 шт., вторая – существует элемент «3» в трёх экземлярах. Дерево на данном шаге никак не изменяется.

5. Далее идёт команда «#Insert», после которой все элементы будут добавляться в дерево. А именно: «0». На рисунке 4 продемонстрировано, как выглядит дерево после данного этапа.

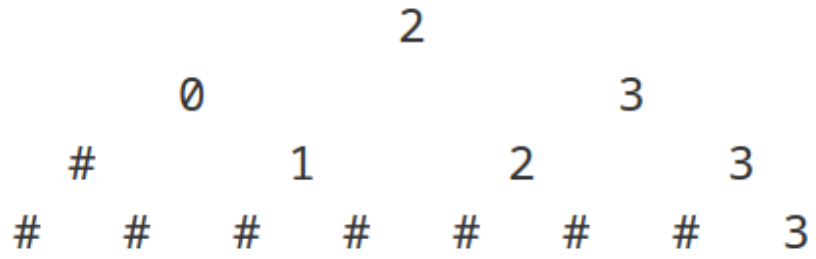
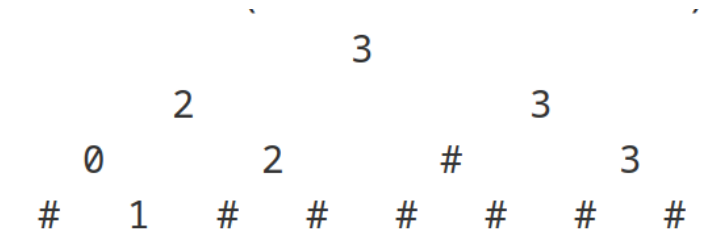
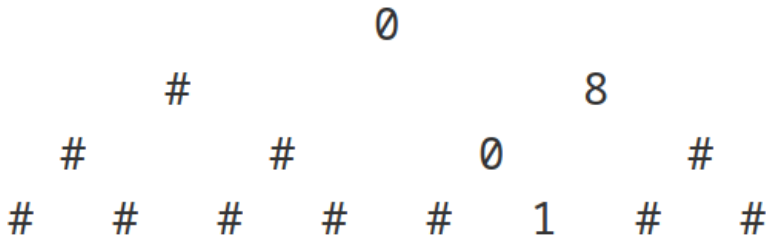
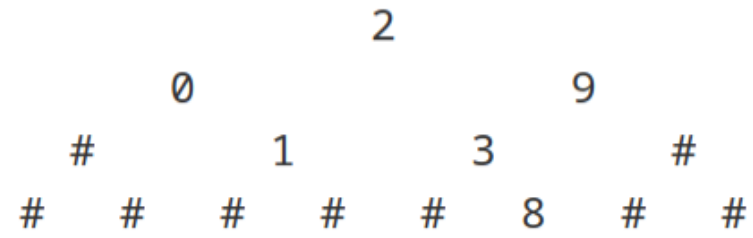


Рисунок 4 – Представление дерева на пятом шаге

Таблица 1 — Результаты тестирования

№	Входные данные	Выходные данные
1	2 3 1 2 3 5 6 #Remove 3 5 6 #Task 3 3 #Count 3 3 #Insert 0	Count = 1 (для элемента "3") Count = 2 (для элемента "3") Count = 3 (для элемента "3") Count = 3 (для элемента "3") 
2	#Insert 3 8 0 #Insert 1 0 #Remove 7 3	
3	#Insert 3 8 0 #Task 3 #Count 3 #Remove 7 3 #Count 3	Count = 1 (для элемента "3") Count = 2 (для элемента "3") Count = 1 (для элемента "3") 
4	#Count 1 #Insert 1 #Count 1 #Remove 1 #Count 1	Count = 0 (для элемента "1") Count = 1 (для элемента "1") Count = 0 (для элемента "1")
5	#Remove 0 1 2 #Insert 2 0 1 # #Task 0	Count = 1 (для элемента "0")

		0 # 1 # # 0 2
6	#Task 0 0 0 #Insert 1 #Remove 0 0 0 #Task 0	Count = 0 (для элемента "0") Count = 1 (для элемента "0") Count = 2 (для элемента "0") Count = 0 (для элемента "0") 1 0 #
7	#Task 1 0 2 5 4 #Insert 4 #Count 4	Count = 0 (для элемента "1") Count = 0 (для элемента "0") Count = 0 (для элемента "2") Count = 0 (для элемента "5") Count = 0 (для элемента "4") Count = 2 (для элемента "4") 4 0 4 # 1 # 5 # # # 2 # # # #
8	3 2 5 6 1	3 1 5 # 2 # 6
9	3 2 5 6 1 #Insert 10 9	10 1 5 # # 2 6 # # # # # 3 # 9
10	#Task 8 3 #Insert 1 2 #Task 0	Count = 0 (для элемента "8") Count = 0 (для элемента "3") Count = 0 (для элемента "0")

	#Insert 9 #Count 3	Count = 1 (для элемента "3") 

4. ДЕМОНСТРАЦИЯ

4.1. Демонстрация операций рандомизированной дерамиды поиска

4.1.1. Вставка

Проведем последовательную вставку в дерамиду последовательности элементов «2», «3», «1».

Первый шаг (Рисунок 5).

```
------------------------------------------------------------------
Начало вставки
------------------------------------------------------------------
Вставляемый элемент:
  Данные: 2
  Приоритет:  1832372109
Рассматриваемый элемент (T) дерева:
  Данные: #
  Приоритет:  #
Текущее дерево:
 #

Вставка элемента в пустую область
------------------------------------------------------------------
Конец вставки
Дерамида после вставки:
 2
------------------------------------------------------------------
```

Рисунок 5 – Представление дерева на первом шаге вставки

Второй шаг (Рисунок 6).

```
--------------------------------------------------------------------------------
Начало вставки
--------------------------------------------------------------------------------
Вставляемый элемент:
  Данные: 3
  Приоритет:  1309320536
Рассматриваемый элемент (T) дерева:
  Данные: 2
  Приоритет:  1832372109
Текущее дерево:
 2

Приоритет вставляемого элемента оказался не больше, чем приоритет рассматриваемого элемента дерева
Переход в правое поддерево, так как данные вставляемого элемента оказались не меньше, чем данные рассматриваемого элемента дерева
--------------------------------------------------------------------------------
Вставляемый элемент:
  Данные: 3
  Приоритет:  1309320536
Рассматриваемый элемент (T) дерева:
  Данные: #
  Приоритет:  #
Текущее дерево:
 #

Вставка элемента в пустую область
--------------------------------------------------------------------------------
Конец вставки
Дерамида после вставки:
   2
 #   3
--------------------------------------------------------------------------------
```

Рисунок 6 – Представление дерева на втором шаге вставки

Шаг третий (Рисунок 7).

```
--------------------------------------------------------------------------------
Начало вставки
--------------------------------------------------------------------------------
Вставляемый элемент:
  Данные: 1
  Приоритет:  568873360
Рассматриваемый элемент (T) дерева:
  Данные: 2
  Приоритет:  1832372109
Текущее дерево:
   2
 #   3

Приоритет вставляемого элемента оказался не больше, чем приоритет рассматриваемого элемента дерева
Переход в левое поддерево, так как данные вставляемого элемента оказались меньше, чем данные рассматриваемого элемента дерева
--------------------------------------------------------------------------------
Вставляемый элемент:
  Данные: 1
  Приоритет:  568873360
Рассматриваемый элемент (T) дерева:
  Данные: #
  Приоритет:  #
Текущее дерево:
 #

Вставка элемента в пустую область
--------------------------------------------------------------------------------
Конец вставки
Дерамида после вставки:
   2
 1   3
--------------------------------------------------------------------------------
```

Рисунок 7 – Представление дерева на третьем шаге вставки

4.1.2. Удаление

Теперь продемонстрируем последовательное удаление элементов из дерева полученного в предыдущем пункте.

Удалим элементы «3», «0», «1», «2».

Первый шаг (Рисунок 8).

```
--------------------------------------------------------------------------------------------------------------------------------
Начало удаления
--------------------------------------------------------------------------------------------------------------------------------
Удаляемый элемент:
  Данные: 3
  Приоритет: #
Рассматриваемый элемент (T) дерева:
  Данные: 2
  Приоритет:  1832372109
Текущее дерево:
   2
 1   3

Данные удаляемого элемента не меньше, чем данные рассматриваемого элемента дерева. Попытка поиска элемента в правом поддереве
--------------------------------------------------------------------------------------------------------------------------------
Удаляемый элемент:
  Данные: 3
  Приоритет: #
Рассматриваемый элемент (T) дерева:
  Данные: 3
  Приоритет:  1309320536
Текущее дерево:
   2
 1   3

Данные совпадают

Слияние двух дерамид, двух потомков рассматриваемого элемента дерева в одну дерамиду:
Левый потомок (T1):
 #
Правый потомок (T2):
 #
Слияние:
 #
Добавление слитой дерамиды вместо удаляемоего элемента:
   2
 1   #
--------------------------------------------------------------------------------------------------------------------------------
Конец удаления
Дерамида после удаления:
   2
 1   #
--------------------------------------------------------------------------------------------------------------------------------
```

Рисунок 8 – Представление дерева на первом шаге удаления

Второй шаг (Рисунок 9).

```
--------------------------------------------------------------------------------------------------------------------------------
Начало удаления
--------------------------------------------------------------------------------------------------------------------------------
Удаляемый элемент:
  Данные: 0
  Приоритет: #
Рассматриваемый элемент (T) дерева:
  Данные: 2
  Приоритет:  1832372109
Текущее дерево:
   2
 1   #

Данные удаляемого элемента меньше, чем данные рассматриваемого элемента дерева. Попытка поиска элемента в левом поддереве
--------------------------------------------------------------------------------------------------------------------------------
Удаляемый элемент:
  Данные: 0
  Приоритет: #
Рассматриваемый элемент (T) дерева:
  Данные: 1
  Приоритет:  568873360
Текущее дерево:
   2
 1   #

Данные удаляемого элемента меньше, чем данные рассматриваемого элемента дерева. Попытка поиска элемента в левом поддереве
--------------------------------------------------------------------------------------------------------------------------------
Элемент не найден. Попытка удаления из пустой области, удаление игнорируется
--------------------------------------------------------------------------------------------------------------------------------
Конец удаления
Дерамида после удаления:
   2
 1   #
--------------------------------------------------------------------------------------------------------------------------------
```

Рисунок 9 – Представление дерева на втором шаге удаления

Третий шаг (Рисунок 10).

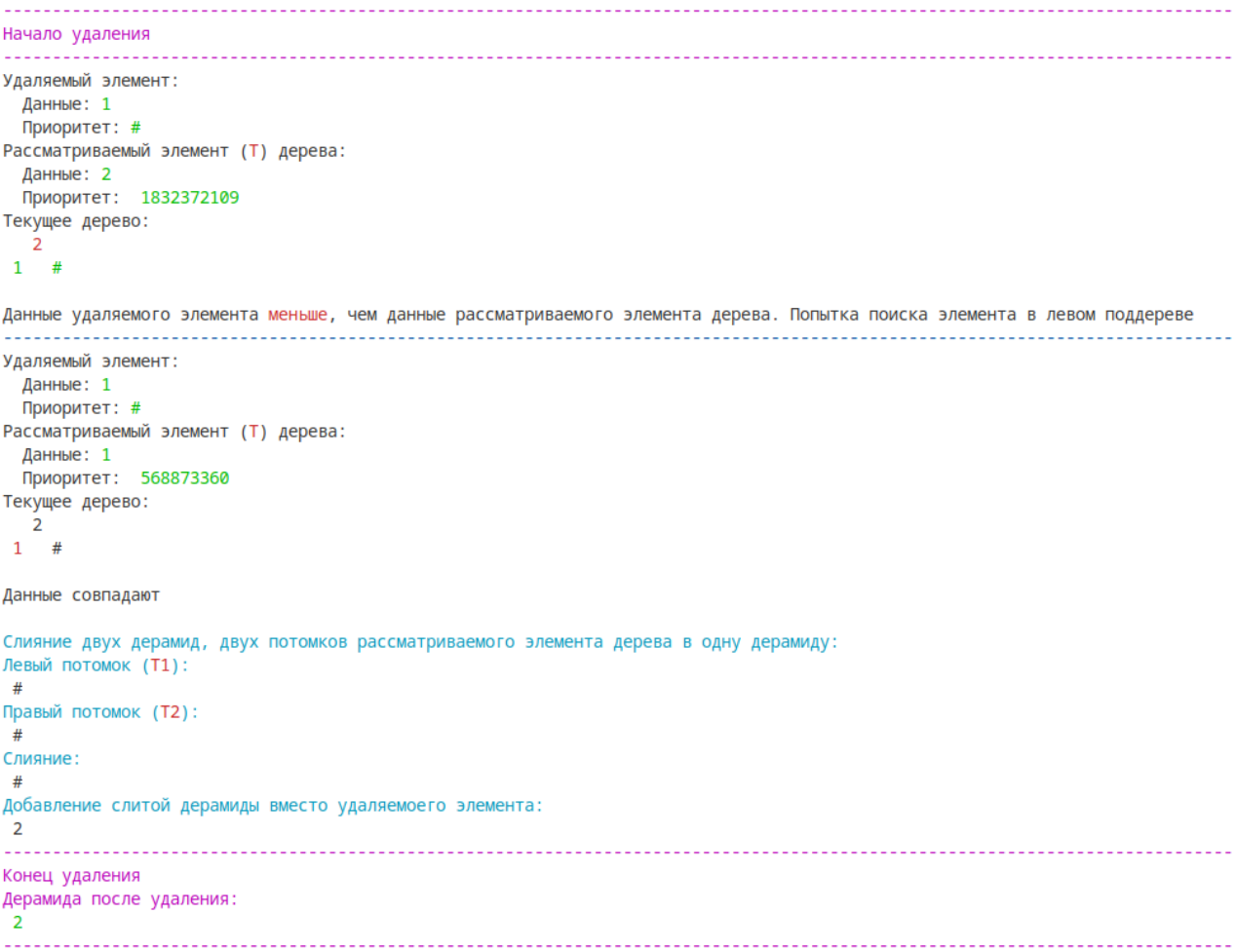


Рисунок 10 – Представление дерева на третьем шаге удаления

Четвертый шаг (Рисунок 11)

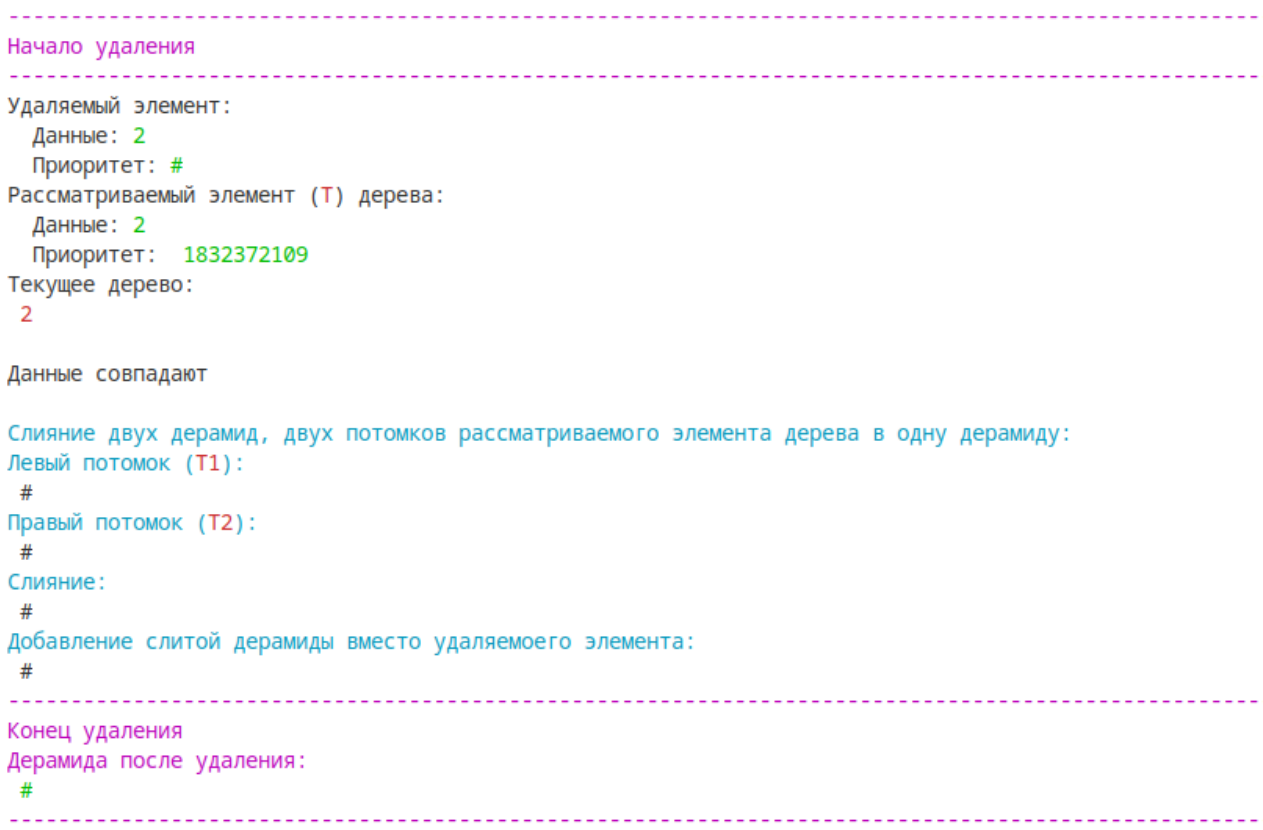


Рисунок 11 – Представление дерева на четвёртом шаге удаления

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы была реализована структура рандомизированная дерамида поиска, а также операции вставки и удаления для него.

Был реализован последовательный вывод тех действий, которые выполнялись при вставке или удалении элемента из дерамиды. Для более явной демонстрации выводилась рандомизированная дерамида поиска в явном виде.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Декартово дерево или дерамида URL: https://neerc.ifmo.ru/wiki/index.php?title=%D0%94%D0%B5%D0%BA%D0%B0%D1%80%D1%82%D0%BE%D0%B2%D0%BE_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE (дата обращения: 14.12.2020).

2. Декартово дерево (treap, дерамида) URL: https://e-maxx.ru/algo/treap (дата обращения: 14.12.2020).

3. Декартово дерево: Часть 1. Описание, операции, применения URL: hhttps://habr.com/ru/post/101818 (дата обращения: 14.12.2020).

ПРИЛОЖЕНИЕ А
НАЗВАНИЕ ПРИЛОЖЕНИЯ

main.cpp

```
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>

#include "../lib/treap.h"

enum class Operation {
  kInsert = 0,
  kRemove,
  kCountAndThenInsert,
  kCount,
  kPrint,
  kExit
};

template <typename T>
void Analyze(const std::string& str, Treap<T> treap) {
  Operation current_operation = Operation::kInsert;
  std::stringstream ss;
  std::string in;
  ss << str;
  bool is_end = false;
  bool is_check_str = true;

  while (!is_end) {
    if (is_check_str && !(ss >> in)) {
      is_check_str = false;
    }
    if (!is_check_str) {
      std::cout << "Ввод: ";
      std::cin >> in;
      std::cout << '\n';
    }

    if (in == "#Insert") {
      current_operation = Operation::kInsert;
    } else if (in == "#Remove") {
      current_operation = Operation::kRemove;
    } else if (in == "#Task") {
      current_operation = Operation::kCountAndThenInsert;
```

```
    } else if (in == "#Count") {
      current_operation = Operation::kCount;
    } else if (in == "#Print") {
      current_operation = Operation::kPrint;
      std::cout << '\n' << treap << "\n\n";
    } else if (in == "#Exit") {
      current_operation = Operation::kExit;
      is_end = true;
    } else {
      if (current_operation == Operation::kInsert) {
        treap.Insert(in);
      } else if (current_operation == Operation::kRemove) {
        treap.Remove(in);
      } else if (current_operation == Operation::kCount) {
        std::cout << "Count = " << treap.Count(in) << " (для
элемента \"" << in
                  << "\")" << '\n';
      } else if (current_operation ==
Operation::kCountAndThenInsert) {
        size_t len = treap.CountAndThenInsert(in);
        std::cout << "Count = " << len << " (для элемента \"" <<
in << "\")"
                  << '\n';
      } else if (current_operation == Operation::kPrint) {
        std::cout << '\n' << treap << "\n\n";
      }
    }
  }
  std::cout << '\n';
}

int main(int argc, char** argv) {
  if (argc == 1) {
    std::cout << "Слишком мало аргументов.\n"
              << "Выражение ожидается после флага \"-s\", "
              << "иначе ожидается путь к файлу.\n\n"
              << "например: ./cw -s \"2 3 1 2 3 5 6 #Remove 3 5 6
#Task 2\" "
                 "Tests/test/test1.txt\n";
  } else {
    bool is_string = false;

    for (int i = 1; i < argc; ++i) {
      bool is_open = false;
      std::string arg = argv[i];
      Treap<std::string> treap;
```

```
      if (arg == "-s" && !is_string) {
        is_string = true;
      } else {
        std::string str = arg;
        if (!is_string) {
          std::ifstream file_in(arg);
          if (file_in.is_open()) {
            is_open = true;
            std::getline(file_in, str);
            file_in.close();
          } else {
            std::cout << "Не удалось открыть файл.\n";
          }
        }
        if (!(!is_string && !is_open)) {
          Analyze<std::string>(str, treap);
        }
      }
    }  // for
  }    // else
  return 0;
}  // main
```

treap.h

```
#ifndef TREAP_H_
#define TREAP_H_

#include <iomanip>
#include <memory>
#include <queue>
#include <random>
#include <sstream>
#include <vector>

#include "color.h"
#include "colored_node.h"
#include "node.h"

template <typename T>
class Treap {
 public:
  Treap() = default;
  void Insert(const T& data);
  void Remove(const T& data);
```

```
  size_t Count(const T& data) const;
  size_t GetHeight() const;
  size_t CountAndThenInsert(const T& data);

  template <typename TT>
  friend std::ostream& operator<<(std::ostream& out, const
Treap<TT>& tree);

  ~Treap() = default;

 private:
  std::shared_ptr<Node<T>> head_;
  Treap(std::shared_ptr<Node<T>> head);
  Treap<T>& operator=(Node<T>& tree);
  std::shared_ptr<Node<T>> InsertPrivate(std::shared_ptr<Node<T>>
tree,
                                         std::shared_ptr<Node<T>>
elem);
  void RemovePrivate(std::shared_ptr<Node<T>> tree, const T& key);
  std::shared_ptr<Node<T>> Split(std::shared_ptr<Node<T>> tree,
const T& data);
  std::shared_ptr<Node<T>> Merge(std::shared_ptr<Node<T>> left,
                                 std::shared_ptr<Node<T>> right);
  size_t GetHeightPrivate(const std::shared_ptr<Node<T>> tree)
const;
  std::vector<std::vector<std::shared_ptr<ColoredNode<T>>>>
DumpFullBinaryTree()
      const;
  void RecursiveDumpFullBinaryTree(
      std::vector<std::vector<std::shared_ptr<ColoredNode<T>>>>&
vec,
      std::shared_ptr<Node<T>> elem, long long current_level = 0)
const;
  size_t RecursiveCount(std::shared_ptr<Node<T>> tree, const T&
data) const;
  void PrintDataAndPriority(Color color, std::shared_ptr<Node<T>>
elem) const;
  void PaintPartOfTree(
      std::vector<std::vector<std::shared_ptr<ColoredNode<T>>>>
tree,
      std::shared_ptr<Node<T>> elem, Color color_vertex,
      Color color_other = Color::kDefault) const;
  size_t MaxLenghtOfElement() const;
  std::ostream& PrintFullBinaryTree(
      std::ostream& out,
```

```
      std::vector<std::vector<std::shared_ptr<ColoredNode<T>>>>
tree) const;
  void PrintSeparator(Color color = Color::kPurple, int length =
175,
                      char separator = '-') const;
  void PrintAndPaintFullBinaryTree(std::shared_ptr<Node<T>> tree,
                                   Color color_vertex,
                                   Color color_other =
Color::kDefault) const;
  void PrintRemoveInfo(const T& data) const;
  void PrintRemoveIfEqual(std::shared_ptr<Node<T>> tree) const;
  void PrintMerge(std::shared_ptr<Node<T>> tree) const;
};

#include "treap.inl"

#endif  // TREAP_H_
```

treap.inl

```
#include <chrono>
#include <random>

#include "treap.h"

template <typename T>
Treap<T>::Treap(std::shared_ptr<Node<T>> head) : head_(head) {}

template <typename T>
Treap<T>& Treap<T>::operator=(Node<T>& tree) {
  if (this->head_ == &tree) {
    return *this;
  }
  head_ = tree;
  return *this;
}

template <typename T>
void Treap<T>::Insert(const T& data) {
  PrintSeparator();
  std::cout << Color::kPurple << "Начало вставки\n" <<
Color::kDefault;
  PrintSeparator();
  // std::random_device rd;
  // std::mt19937 mersenne_twister(rd());
```

```
srand(std::chrono::high_resolution_clock::now().time_since_epoch()
.count());
  std::shared_ptr<Node<T>> elem(new Node<T>(data, rand()));
  head_ = InsertPrivate(head_, elem);

  PrintSeparator();
  std::cout << Color::kPurple
            << "Конец вставки\nДерамида после вставки:" <<
Color::kDefault;
  std::cout << Color::kGreen << '\n';
  PrintAndPaintFullBinaryTree(head_, Color::kGreen,
Color::kGreen);
  std::cout << Color::kDefault;
  PrintSeparator();

  PrintSeparator(Color::kDefault, 12, '\n');
}

template <typename T>
void Treap<T>::Remove(const T& data) {
  PrintSeparator();
  std::cout << Color::kPurple << "Начало удаления\n" <<
Color::kDefault;
  PrintSeparator();

  RemovePrivate(head_, data);

  PrintSeparator();
  std::cout << Color::kPurple
            << "Конец удаления\nДерамида после удаления:" <<
Color::kDefault;
  std::cout << Color::kGreen << '\n';
  PrintAndPaintFullBinaryTree(head_, Color::kGreen,
Color::kGreen);
  std::cout << Color::kDefault;
  PrintSeparator();

  std::cout << "\n\n\n\n";  // FIXIT
}

template <typename T>
size_t Treap<T>::Count(const T& data) const {
  return RecursiveCount(head_, data);
}
```

```
template <typename T>
size_t Treap<T>::GetHeight() const {
  return GetHeightPrivate(head_);
}

template <typename T>
size_t Treap<T>::CountAndThenInsert(const T& data) {
  size_t save_count = Count(data);
  Insert(data);
  return save_count;
}

template <typename T>
std::ostream& operator<<(std::ostream& out, const Treap<T>& tree)
{
  auto vec = tree.DumpFullBinaryTree();
  tree.PrintFullBinaryTree(out, vec);
  return out;
}

template <typename T>
std::shared_ptr<Node<T>> Treap<T>::InsertPrivate(
    std::shared_ptr<Node<T>> tree, std::shared_ptr<Node<T>> elem)
{
  std::cout << "Вставляемый элемент:";
  PrintDataAndPriority(Color::kGreen, elem);
  std::cout << "\nРассматриваемый элемент (" << Color::kRed << "T"
            << Color::kDefault << ") дерева:";
  PrintDataAndPriority(Color::kGreen, tree);
  std::cout << '\n' << "Текущее дерево:\n";
  if (!tree) {
    std::cout << " #";
  }

  std::shared_ptr<Node<T>> new_element(nullptr);
  if (tree == nullptr) {
    std::cout << "\n\nВставка элемента в пустую область\n";
    new_element = elem;

  } else if (elem->priority_ > tree->priority_) {
    PrintAndPaintFullBinaryTree(tree, Color::kRed);
    std::cout << "\nПриоритет вставляемого элемента оказался" <<
Color::kRed
              << " больше" << Color::kDefault
              << ", чем приоритет рассматриваемого элемента
дерева\n";
```

```
    std::cout << "Вставка элемента на место рассматриваемого
элемента дерева\n";

    std::cout << Color::kWave << "\nРазделение дерева (" <<
Color::kRed << "T"
              << Color::kWave << ") на два поддерева (" <<
Color::kRed << "T1"
              << Color::kWave << " и " << Color::kRed << "T2" <<
Color::kWave
              << ") так, что в" << Color::kRed << " T1 " <<
Color::kWave
              << "будет находиться элементы, данные которых" <<
Color::kRed
              << " меньше " << Color::kWave
              << "данных вставляемого элемента, а в" <<
Color::kRed << " T2 "
              << Color::kWave << "— все остальные:\n"
              << Color::kDefault;

    new_element = Split(tree, elem->data_);

    std::cout << Color::kWave << "Левое поддерево (" <<
Color::kRed << "T1"
              << Color::kWave << "):\n"
              << Color::kDefault;
    Treap temp(new_element->left_);
    std::cout << temp << '\n';
    std::cout << Color::kWave << "Правое поддерево (" <<
Color::kRed << "T2"
              << Color::kWave << "):\n"
              << Color::kDefault;
    temp = new_element->right_;
    std::cout << temp << '\n';

    elem->left_ = new_element->left_;
    elem->right_ = new_element->right_;
    new_element = elem;

    std::cout << Color::kWave << "Подвязываем к вставляемому
элементу"
              << Color::kRed << " T1 " << Color::kWave << "левым
потомком, а"
              << Color::kRed << " T2 " << Color::kWave
              << "— правым. Вместо рассматриваемого элемента" <<
Color::kRed
              << " T " << Color::kWave
```

```
              << "дерева добавляем вставляемый элемент:\n"
              << Color::kDefault;
    temp = new_element;
    std::cout << temp << '\n';

  } else {
    PrintAndPaintFullBinaryTree(tree, Color::kRed, Color::kGreen);

    std::cout << "\nПриоритет вставляемого элемента оказался" <<
Color::kRed
              << " не больше" << Color::kDefault
              << ", чем приоритет рассматриваемого элемента
дерева\n";

    if (elem->data_ < tree->data_) {
      std::cout << "Переход в" << Color::kRed << " левое " <<
Color::kDefault
                << "поддерево, так как данные вставляемого
элемента оказались"
                << Color::kRed << " меньше" << Color::kDefault
                << ", чем данные рассматриваемого элемента дерева\
n";
      PrintSeparator(Color::kBlue);
      tree->left_ = InsertPrivate(tree->left_, elem);
    } else {
      std::cout << "Переход в" << Color::kRed << " правое " <<
Color::kDefault
                << "поддерево, так как данные вставляемого
элемента оказались"
                << Color::kRed << " не меньше" << Color::kDefault
                << ", чем данные рассматриваемого элемента дерева\
n";
      PrintSeparator(Color::kBlue);
      tree->right_ = InsertPrivate(tree->right_, elem);
    }
    new_element = tree;
  }
  return new_element;
}

template <typename T>
void Treap<T>::RemovePrivate(std::shared_ptr<Node<T>> tree, const
T& data) {
  PrintRemoveInfo(data);
  PrintDataAndPriority(Color::kGreen, tree);
  std::cout << '\n' << "Текущее дерево:\n";
```

```
  if (!tree) {
    std::cout << " #";
  } else {
    PrintAndPaintFullBinaryTree(tree, Color::kRed, Color::kGreen);
  }
  std::cout << '\n';

  if (tree == nullptr) {
    std::cout << "Элемент не найден. Попытка удаления из пустой
области, "
                 "удаление игнорируется\n";
    return;
  } else if (tree->data_ != data) {
    if (data < tree->data_) {
      std::cout << "Данные удаляемого элемента" << Color::kRed <<
" меньше"
                << Color::kDefault
                << ", чем данные рассматриваемого элемента дерева.
Попытка "
                   "поиска элемента в левом поддереве\n";
      PrintSeparator(Color::kBlue);
      if (tree->left_ != nullptr) {
        if (tree->left_->data_ != data) {
          RemovePrivate(tree->left_, data);
        } else {
          PrintRemoveInfo(data);
          PrintDataAndPriority(Color::kGreen, tree->left_);
          std::cout << '\n' << "Текущее дерево:\n";
          PrintAndPaintFullBinaryTree(tree->left_, Color::kRed,
Color::kGreen);
          std::cout << '\n';

          PrintRemoveIfEqual(tree->left_);

          tree->left_ = Merge(tree->left_->left_, tree->left_-
>right_);

          PrintMerge(tree->left_);
        }
      } else {
        std::cout << "Элемент не найден. Попытка удаления из
пустой области, "
                     "удаление игнорируется\n";
      }

    } else {
```

```
      std::cout << "Данные удаляемого элемента" << Color::kRed <<
" не меньше"
                << Color::kDefault
                << ", чем данные рассматриваемого элемента дерева.
Попытка "
                   "поиска элемента в правом поддереве\n";
      PrintSeparator(Color::kBlue);
      if (tree->right_ != nullptr) {
        if (tree->right_->data_ != data) {
          RemovePrivate(tree->right_, data);
        } else {
          PrintRemoveInfo(data);
          PrintDataAndPriority(Color::kGreen, tree->right_);
          std::cout << '\n' << "Текущее дерево:\n";
          PrintAndPaintFullBinaryTree(tree->right_, Color::kRed,
Color::kGreen);
          std::cout << '\n';

          PrintRemoveIfEqual(tree->right_);

          tree->right_ = Merge(tree->right_->left_, tree->right_-
>right_);

          PrintMerge(tree->right_);
        }
      } else {
        std::cout << "Элемент не найден. Попытка удаления из
пустой области, "
                     "удаление игнорируется\n";
      }
    }
  } else {
    PrintRemoveIfEqual(head_);
    head_ = Merge(head_->left_, head_->right_);
    PrintMerge(head_);
  }
}

template <typename T>
std::shared_ptr<Node<T>> Treap<T>::Split(std::shared_ptr<Node<T>>
tree,
                                         const T& data) {
  std::shared_ptr<Node<T>> new_element(nullptr);
  if (tree == nullptr) {
    new_element.reset(new Node<T>(nullptr, nullptr));
  } else if (data > tree->data_) {
```

```
    new_element = Split(tree->right_, data);
    tree->right_ = new_element->left_;
    new_element->left_ = tree;
  } else {
    new_element = Split(tree->left_, data);
    tree->left_ = new_element->right_;
    new_element->right_ = tree;
  }
  return new_element;
}

template <typename T>
std::shared_ptr<Node<T>> Treap<T>::Merge(std::shared_ptr<Node<T>>
left,
                                         std::shared_ptr<Node<T>>
right) {
  std::shared_ptr<Node<T>> new_element(nullptr);
  if (left == nullptr) {
    new_element = right;
  } else if (right == nullptr) {
    new_element = left;
  } else if (left->priority_ > right->priority_) {
    left->right_ = Merge(left->right_, right);
    new_element = left;
  } else {
    right->left_ = Merge(left, right->left_);
    new_element = right;
  }
  return new_element;
}

template <typename T>
size_t Treap<T>::GetHeightPrivate(const std::shared_ptr<Node<T>>
tree) const {
  if (tree == nullptr) {
    return 0;
  }
  return 1 + std::max(GetHeightPrivate(tree->left_),
                      GetHeightPrivate(tree->right_));
}

template <typename T>
std::vector<std::vector<std::shared_ptr<ColoredNode<T>>>>
Treap<T>::DumpFullBinaryTree() const {
  std::vector<std::vector<std::shared_ptr<ColoredNode<T>>>>
vec(GetHeight());
```

```
  RecursiveDumpFullBinaryTree(vec, head_);
  return vec;
}

template <typename T>
void Treap<T>::RecursiveDumpFullBinaryTree(
    std::vector<std::vector<std::shared_ptr<ColoredNode<T>>>>&
vec,
    std::shared_ptr<Node<T>> elem, long long current_level) const
{
  if (current_level == vec.size()) {
    return;
  }

  vec[current_level].push_back(
      std::shared_ptr<ColoredNode<T>>(new ColoredNode(elem)));

  RecursiveDumpFullBinaryTree(vec, elem == nullptr ? nullptr :
elem->left_,
                              current_level + 1);
  RecursiveDumpFullBinaryTree(vec, elem == nullptr ? nullptr :
elem->right_,
                              current_level + 1);
}

template <typename T>
size_t Treap<T>::RecursiveCount(std::shared_ptr<Node<T>> tree,
                                const T& data) const {
  if (tree == nullptr) {
    return 0;
  }

  return (tree->data_ == data) + (tree->data_ > data
                                      ? RecursiveCount(tree-
>left_, data)
                                      : RecursiveCount(tree-
>right_, data));
}

template <typename T>
void Treap<T>::PrintDataAndPriority(Color color,
                                    std::shared_ptr<Node<T>> elem)
const {
  std::cout << "\n  Данные: " << color;

  if (elem) {
```

```
    std::cout << elem->data_;
  } else {
    std::cout << '#';
  }

  std::cout << Color::kDefault;
  std::cout << "\n  Приоритет:  " << color;

  if (elem) {
    std::cout << elem->priority_;
  } else {
    std::cout << '#';
  }
  std::cout << Color::kDefault;
}

template <typename T>
void Treap<T>::PaintPartOfTree(
    std::vector<std::vector<std::shared_ptr<ColoredNode<T>>>>
tree,
    std::shared_ptr<Node<T>> elem, Color color_vertex,
    Color color_other) const {
  if (!elem) {
    return;
  }

  bool is_find = false;
  size_t first_colored = 0, last_colored = 0;
  size_t len_colored = 1;
  for (size_t i = 0; i < tree.size(); ++i) {
    for (size_t j = 0; j < tree[i].size(); ++j) {
      if (tree[i][j] && elem == tree[i][j]->node) {
        is_find = true;
        first_colored = j;
        tree[i][j]->color = color_vertex;
      } else if (is_find && first_colored <= j && j <
last_colored) {
        tree[i][j]->color = color_other;
      }
    }
    if (is_find) {
      first_colored <<= 1;
      len_colored <<= 1;
      last_colored = first_colored + len_colored;
    }
  }
```

```
}

template <typename T>
size_t Treap<T>::MaxLenghtOfElement() const {
  std::queue<std::shared_ptr<Node<T>>> queue;
  size_t element_length = 0;

  if (head_ != nullptr) {
    queue.push(head_);
  }

  while (!queue.empty()) {
    std::shared_ptr<Node<T>> elem = queue.front();
    queue.pop();

    if (elem != nullptr) {
      std ::stringstream ss;
      ss << elem->data_;
      element_length = std::max(element_length, ss.str().size());

      queue.push(elem->left_);
      queue.push(elem->right_);
    }
  }
  return element_length;
}

template <typename T>
std::ostream& Treap<T>::PrintFullBinaryTree(
    std::ostream& out,
    std::vector<std::vector<std::shared_ptr<ColoredNode<T>>>>
tree) const {
  const size_t height = GetHeight();
  size_t element_length = MaxLenghtOfElement();
  for (int current_level = 0; current_level < height; +
+current_level) {
    if (current_level) {
      out << '\n';
    }
    for (int i = 0; i < tree[current_level].size(); ++i) {
      size_t cell_width = height - current_level + 1;
      if (!i) {
        --cell_width;
      }
      out << tree[current_level][i]->color;
      out << std::setw((1 << cell_width) * element_length);
```

```
      if (tree[current_level][i]->node) {
        out << tree[current_level][i]->node->data_;
      } else {
        out << '#';
      }
    }
    out << Color::kDefault;
  }

  if (!height) {
    out << " #";
  }

  return out;
}

template <typename T>
void Treap<T>::PrintSeparator(Color color, int length, char
separator) const {
  std::cout << color << std::setfill(separator) <<
std::setw(length) << ' '
            << std::setfill(' ') << '\n'
            << Color::kDefault;
}

template <typename T>
void
Treap<T>::PrintAndPaintFullBinaryTree(std::shared_ptr<Node<T>>
tree,
                                      Color color_vertex,
                                      Color color_other)
const {
  auto vec = DumpFullBinaryTree();
  PaintPartOfTree(vec, tree, color_vertex, color_other);
  PrintFullBinaryTree(std::cout, vec);
  std::cout << '\n';
}

template <typename T>
void Treap<T>::PrintRemoveInfo(const T& data) const {
  std::cout << "Удаляемый элемент:";
  std::cout << "\n  Данные: " << Color::kGreen << data <<
Color::kDefault
            << '\n';
  std::cout << "  Приоритет: " << Color::kGreen << "#" <<
Color::kDefault;
```

```
  std::cout << "\nРассматриваемый элемент (" << Color::kRed << "T"
            << Color::kDefault << ") дерева:";
}

template <typename T>
void Treap<T>::PrintRemoveIfEqual(std::shared_ptr<Node<T>> tree)
const {
  std::cout << "Данные совпадают\n\n";
  std::cout << Color::kWave
            << "Слияние двух дерамид, двух потомков
рассматриваемого "
               "элемента дерева в одну дерамиду:\n"
            << Color::kDefault;
  std::cout << Color::kWave << "Левый потомок (" << Color::kRed <<
"T1"
            << Color::kWave << "):\n"
            << Color::kDefault;
  Treap temp(tree->left_);
  std::cout << temp << '\n';

  std::cout << Color::kWave << "Правый потомок (" << Color::kRed
<< "T2"
            << Color::kWave << "):\n"
            << Color::kDefault;
  temp = tree->right_;
  std::cout << temp << '\n';
}

template <typename T>
void Treap<T>::PrintMerge(std::shared_ptr<Node<T>> tree) const {
  std::cout << Color::kWave << "Слияние:\n" << Color::kDefault;
  Treap temp(tree);
  std::cout << temp << '\n';

  std::cout << Color::kWave
            << "Добавление слитой дерамиды вместо удаляемоего
элемента:\n"
            << Color::kDefault;
  PrintAndPaintFullBinaryTree(tree, Color::kRed, Color::kGreen);
}
```

node.h

```
#ifndef TREE_H_
#define TREE_H_
```

```
#include <memory>

template <typename T>
class Treap;

template <typename T>
class Node {
 public:
  Node();
  Node(const T& data, long long priority = 0,
       std::shared_ptr<Node<T>> left = nullptr,
       std::shared_ptr<Node<T>> right = nullptr);
  Node(std::shared_ptr<Node<T>> left, std::shared_ptr<Node<T>>
right);

 private:
  friend class Treap<T>;

  T data_;
  long long priority_;
  std::shared_ptr<Node<T>> left_;
  std::shared_ptr<Node<T>> right_;
};

#include "node.inl"

#endif  // TREE_H_
```

node.inl

```
#include "node.h"

template <typename T>
Node<T>::Node() : priority_(0), right_(nullptr), left_(nullptr) {}

template <typename T>
Node<T>::Node(const T& data, long long priority,
std::shared_ptr<Node<T>> left,
              std::shared_ptr<Node<T>> right)
    : data_(data), priority_(priority), left_(left), right_(right)
{}

template <typename T>
Node<T>::Node(std::shared_ptr<Node<T>> left,
std::shared_ptr<Node<T>> right)
    : priority_(0), left_(left), right_(right) {}
```

colored_node.h

```
#ifndef COLORED_NODE_H_
#define COLORED_NODE_H_

#include "color.h"
#include "node.h"

template <typename T>
class ColoredNode {
 public:
  std::shared_ptr<Node<T>> node = nullptr;
  Color color = Color::kDefault;

  ColoredNode(std::shared_ptr<Node<T>> elem, Color color =
Color::kDefault);
  template <typename TT>
  friend std::ostream& operator<<(std::ostream& out,
ColoredNode<TT>& elem);
};

#include "colored_node.inl"

#endif  // COLORED_NODE_H_
```

colored_node.inl

```
#include "colored_node.h"

template <typename T>
ColoredNode<T>::ColoredNode(std::shared_ptr<Node<T>> node_, Color
color_)
    : node(node_), color(color_) {}

template <typename T>
std::ostream& operator<<(std::ostream& out, ColoredNode<T>& elem)
{
  std::cout << elem.color;
  if (elem.node) {
    std::cout << elem.node->data_;
  } else {
    std::cout << '#';
  }
  std::cout << Color::kDefault;
  return out;
```

```
}
```

color.h

```
#ifndef COLOR_H_
#define COLOR_H_

#include <iostream>
#include <memory>

#include "node.h"

enum Color {
  kDefault = 0,
  kBlack = 30,
  kRed,
  kGreen,
  kYellow,
  kBlue,
  kPurple,
  kWave,
  kGrey
};

std::ostream& operator<<(std::ostream& out, Color color) {
  out << "\033[" << (int)color << "m";
  return out;
}

#endif  // COLOR_H_
```

Makefile

```
.PHONY: all clean rebuild

CXX = g++
TARGET = cw
CXXFLAGS = -g -c -std=c++17
CXXOBJFLAGS = -g -std=c++17
LIBDIR = source/lib
SRCDIR = source/src
SRCS = $(wildcard $(SRCDIR)/*.cc)
OBJS = $(SRCS:.cpp=.o)

all: $(TARGET)
```

```
$(TARGET): $(OBJS)
     @echo "Compiling:"
     $(CXX) $(CXXOBJFLAGS) $(OBJS) -o $(TARGET)

%.o: $(SRCDIR)/%.cpp $(LIBDIR)/*.h
     $(CXX) $(CXXFLAGS) $<

clean:
     @echo "Cleaning build files:"
     rm -rf $(SRCDIR)/*.o $(TARGET)

rebuild: clean all
     clear
     ./cw -s ""
```