

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: Бинарные деревья поиска

Студент гр. 9381

Прибылов Н.А.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

Студент Прибылов Н.А.

Группа 9381

Тема работы: Рандомизированная дерамида поиска - демонстрация

Исходные данные:

- ОС Linux
- Язык программирования C++
- Компилятор GCC
- IDE CLion

Содержание пояснительной записки:

- Введение
- Формулировка задания
- Написание исходного кода программы
- Тестирование работы программы
- Заключение
- Список использованных источников

Предполагаемый объем пояснительной записки:

Не менее 30 страниц.

Дата выдачи задания: 31.10.2019

Дата сдачи реферата: 30.12.2019

Дата защиты реферата: 30.12.2019

Студент

Прибылов Н.А.

Преподаватель

Фирсов М.А.

АННОТАЦИЯ

Была разработана программа, реализующая рандомизированную дерамиду поиска (*treap*) и позволяющая добавлять и удалять элементы из неё. Программа выводит дерево в наглядном виде, а так же действия пошагово с пояснениями на экран.

SUMMARY

A program was developed that implements treap data structure and allows user to add and remove elements from it. The program displays a tree in a clear form, as well as step by step actions with explanations on the screen.

СОДЕРЖАНИЕ

	Введение	5
1.	Формулировка задания	6
2.	Описание структур данных и функций	7
2.1.	Файл Tgear.h/cpp	7
2.2	Структура Tgear	7
2.3	Функции для работы с Tgear	7
3.	Описание алгоритма	9
4.	Тестирование программы	15
	 Заключение	 24
	Список использованных источников	25
	Приложение А. Исходный код программы	26

ВВЕДЕНИЕ

Цель работы.

Изучение бинарного дерева поиска, а именно рандомизированной дерамиды поиска (РДП), а так же операций вставки и удаления элементов.

Задачи.

Реализация РДП с операциями вставки и удаления, а также наглядный вывод структуры на экран.

Основные теоретические сведения.

Пусть имеются данные дерева — ключи x . Добавим к ним еще один параметр в пару — y , и назовем его приоритетом. Теперь построим такое дерево, которое хранит в каждой вершине по два параметра, и при этом по ключам является двоичным деревом поиска, а по приоритетам — кучей. Такое дерево называется декартовым. Если приоритеты генерировать случайным образом, получается рандомизированная дерамида поиска. Такая структура с очень высокой, стремящейся к 100% вероятностью, будет иметь высоту, не превосходящую $4 \log_2 N$. А значит, хоть оно может и не быть идеально сбалансированным, время поиска ключа в таком дереве будет в основном порядка $O(\log_2 N)$.

В англоязычной литературе такая структура имеет название *treap*, которое наглядно отражает её суть: *tree* + *heap*. В русскоязычной же иногда можно встретить составленные по такому же принципу: уже упомянутая дерамида (дерево + пирамида), дуча (дерево + куча) или курево (куча + дерево).

1. ФОРМУЛИРОВКА ЗАДАНИЯ

Вариант 12.

Рандомизированная дерамида поиска — демонстрация.

2. ОПИСАНИЕ СТРУКТУР ДАННЫХ И ФУНКЦИЙ

2.1. Файл Treap.h/cpp

Содержит реализацию дерамиды в виде структуры и ряда функций, а также вспомогательную структуру и набор функций для наглядного вывода дерамиды на экран.

Вспомогательные псевдонимы:

using TreapPtr = std::shared_ptr<class Treap> – указатель на объект класса Treap;

using TreapPair = std::pair<TreapPtr, TreapPtr> – пара из двух указателей на объекты класса Treap;

2.2. Структура Treap

Дерамиды. Имеет следующие поля:

- int key – ключ текущего элемента;
- int priority – приоритет текущего элемента;
- TreapPtr left – указатель на левое поддерево;
- TreapPtr right – указатель на правое поддерево;

2.3 Функции для работы с Treap

- Treap(int key, int priority = rand(), TreapPtr left = nullptr, TreapPtr right = nullptr) – конструктор, принимает ключ key, приоритет priority, указатели на левое и правое поддерева left и right;
- TreapPtr merge(const TreapPtr& L, const TreapPtr& R) – принимает два дерева L и R, сливает их в одно и возвращает его;
- TreapPair split(const TreapPtr& T, int key) – принимает дерево T и по ключу key расщепляет его, возвращает пару из левой и правой половины;

- `TreapPtr insert(TreapPtr& T, int key)` – принимает дерево `T` и вставляет элемент с ключом `key` в него, возвращает это же дерево;
- `TreapPtr remove(TreapPtr& T, int key)` – принимает дерево `T` и удаляет элемент с ключом `key` из него, возвращает это же дерево;
- `TreapPtr build(const std::vector<int>& keys)` – принимает вектор ключей `keys` и строит на их основе РДП.

3. ОПИСАНИЕ АЛГОРИТМА

Вся работа с декартовым деревом заключается в двух основных операциях: *Merge* и *Split*. С помощью них элементарно выражаются все остальные популярные операции.

3.1. Merge

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

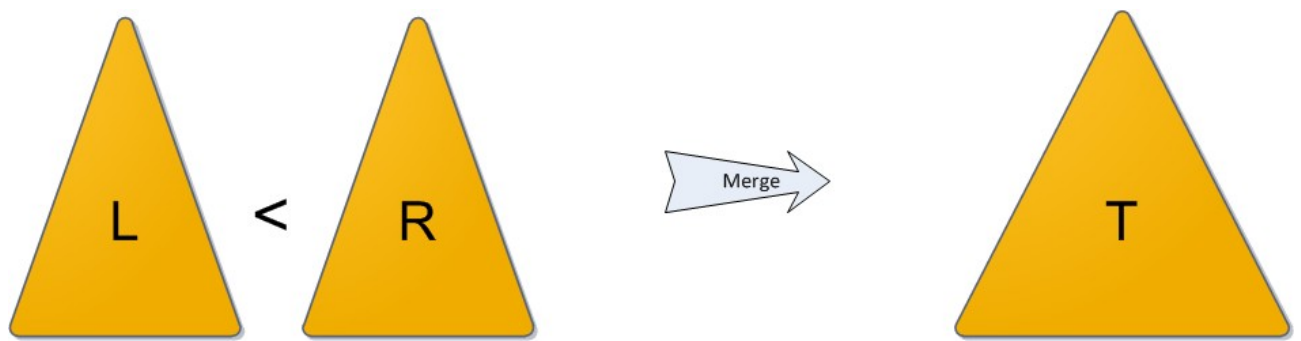


Рис. 1

Корнем будущего дерева станет, очевидно, элемент с наибольшим приоритетом. Кандидатов на максимальный приоритет два — только корни двух исходных деревьев. Сравним их приоритеты; пусть для однозначности приоритет у левого корня больше, а ключ в нем равен x . Новый корень определен, теперь нужно решить, какие элементы окажутся в его правом поддереве, а какие — в левом.

Легко понять, что все дерево R окажется в правом поддереве нового корня: его ключи больше x по условию. Точно так же левое поддерево старого корня L .*Left* имеет все ключи, меньшие x , и должно остаться левым поддеревом.

Остаётся правое поддерево $L.Right$: рекурсивно вызываем $Merge$ для $L.Right$ и дерева R , и возвращенное ею дерево используем как новое правое поддерево.

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

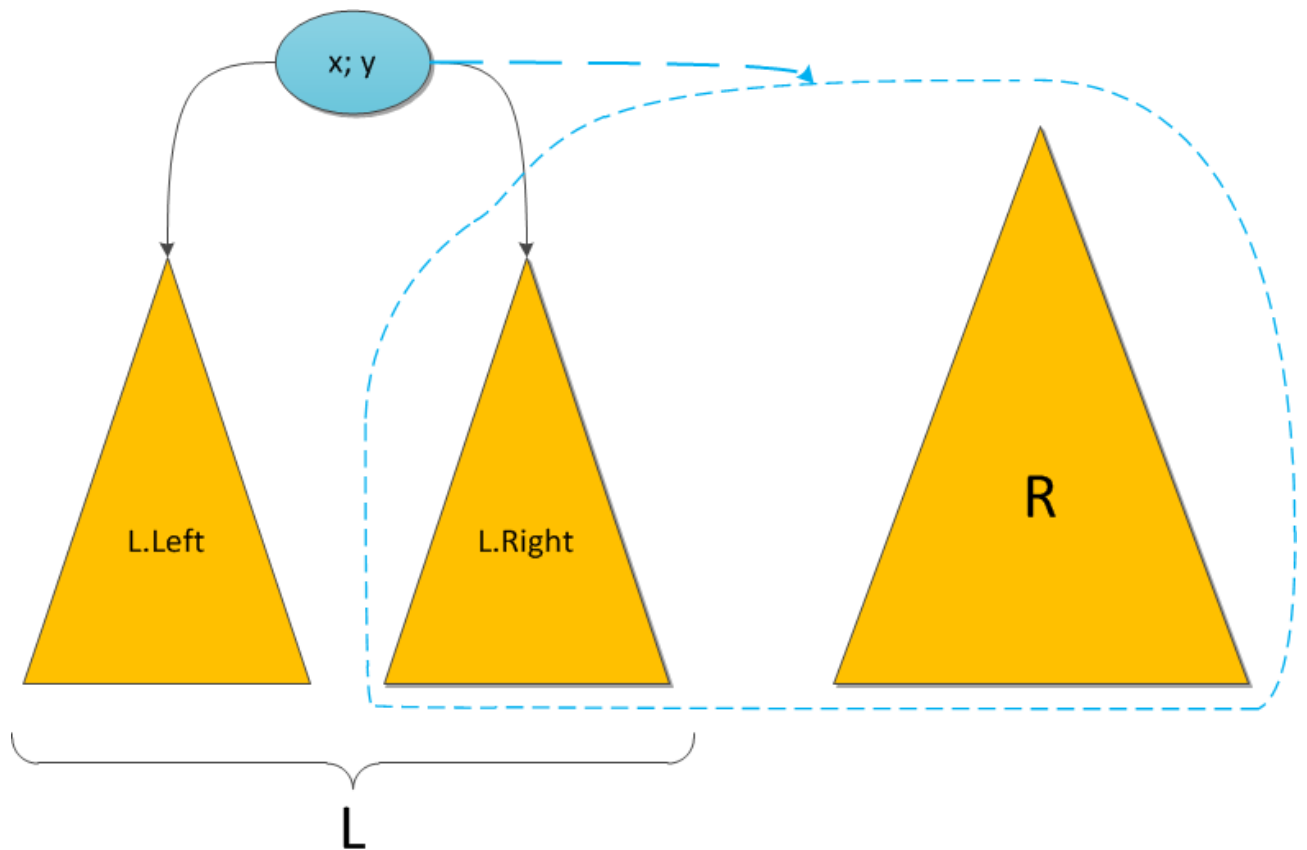


Рис. 2

Симметричный случай — когда приоритет в корне дерева R выше — разбирается аналогично.

3.2. Split

Операция *Split*: на вход ей поступает корректное декартово дерево T и некий ключ x_0 . Задача операции — разделить дерево на два так, чтобы в одном из них (L) оказались все элементы исходного дерева с ключами, меньшими x_0 , а в другом (R) — с большими. Никаких особых ограничений на дерево не накладывается.

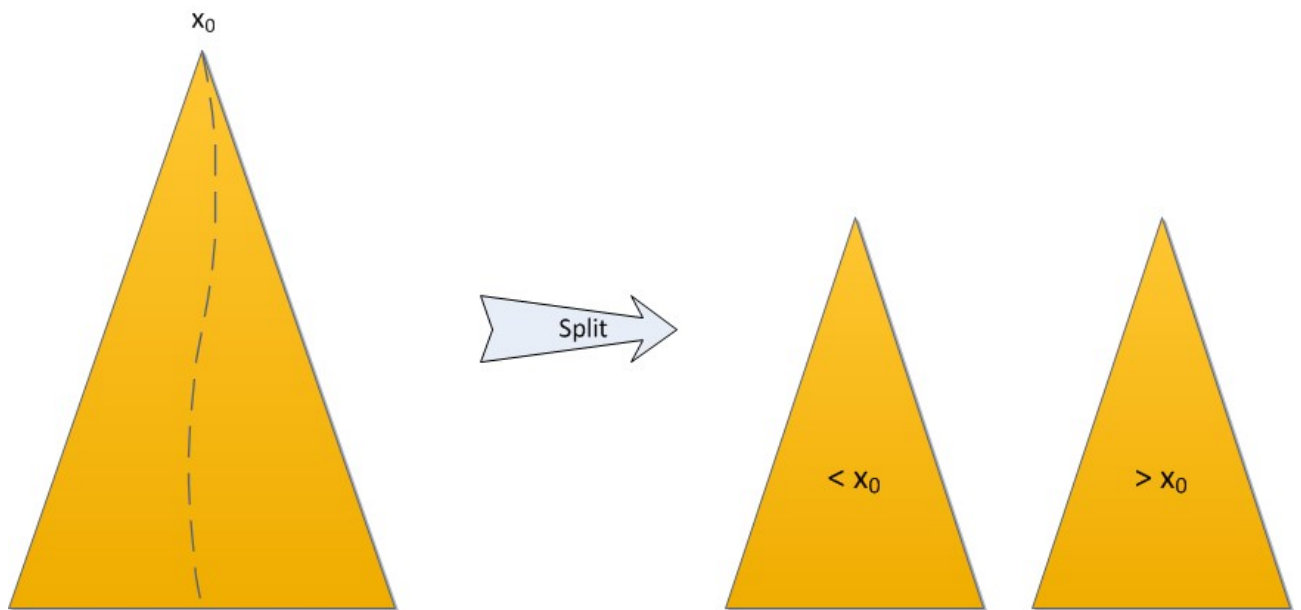


Рис. 3

Если ключ корня меньше x_0 , то корень окажется в L , иначе в R . Предположим для однозначности, что ключ корня оказался меньше x_0 .

Тогда очевидно, что все элементы левого поддерева T также окажутся в L — их ключи тоже будут меньше x_0 . Более того, корень T будет и корнем L , поскольку его приоритет наибольший во всем дереве. Левое поддерево корня полностью сохранится без изменений, а вот правое уменьшится — из него придется убрать элементы с ключами, большими x_0 , и вынести в дерево R . А остаток ключей сохранить как новое правое поддерево L . Здесь снова можно прибегнуть к рекурсии. Возьмем правое поддерево и рекурсивно разрежем его по тому же ключу x_0 на два дерева L' и R' . После чего становится ясно, что L'

станет новым правым поддеревом дерева L , а R' и есть непосредственно дерево R — оно состоит из тех и только тех элементов, которые больше $x\theta$.

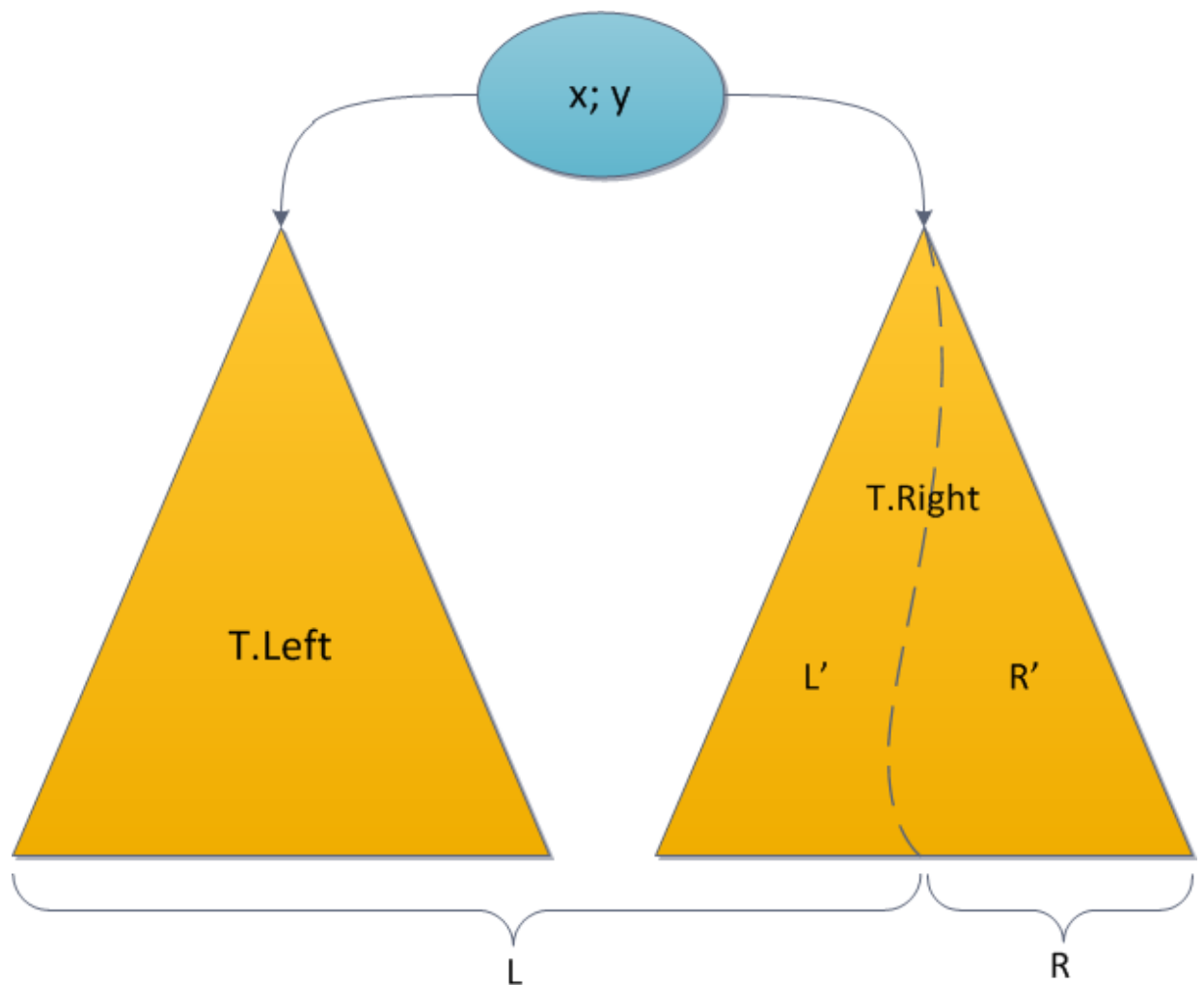


Рис. 4

Симметричный случай, при котором ключ корня больше, чем $x\theta$, тоже совершенно идентичен.

Деревья, выдаваемые на выход операцией *Split*, подходят как входные данные для операции *Merge*: все ключи левого дерева не превосходят ключей в правом.

Время работы *Merge* и *Split*:

Из описания алгоритма видно, что *Merge* за каждую итерацию рекурсии уменьшает суммарную высоту двух сливаемых деревьев как минимум на

единицу, так что общее время работы не превосходит $2N$, то есть $O(N)$. А используя *Split*, мы работаем с единственным деревом, его высота уменьшается с каждой итерацией тоже как минимум на единицу, и асимптотика работы операции тоже $O(N)$. А поскольку это декартово дерево со случайными приоритетами, оно с высокой вероятностью имеет близкую к логарифмической высоту, то *Merge* и *Split* работают за $O(\log_2 N)$.

3.3. Insert

Добавление элемента:

Используя универсальность операций *Split/Merge*:

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

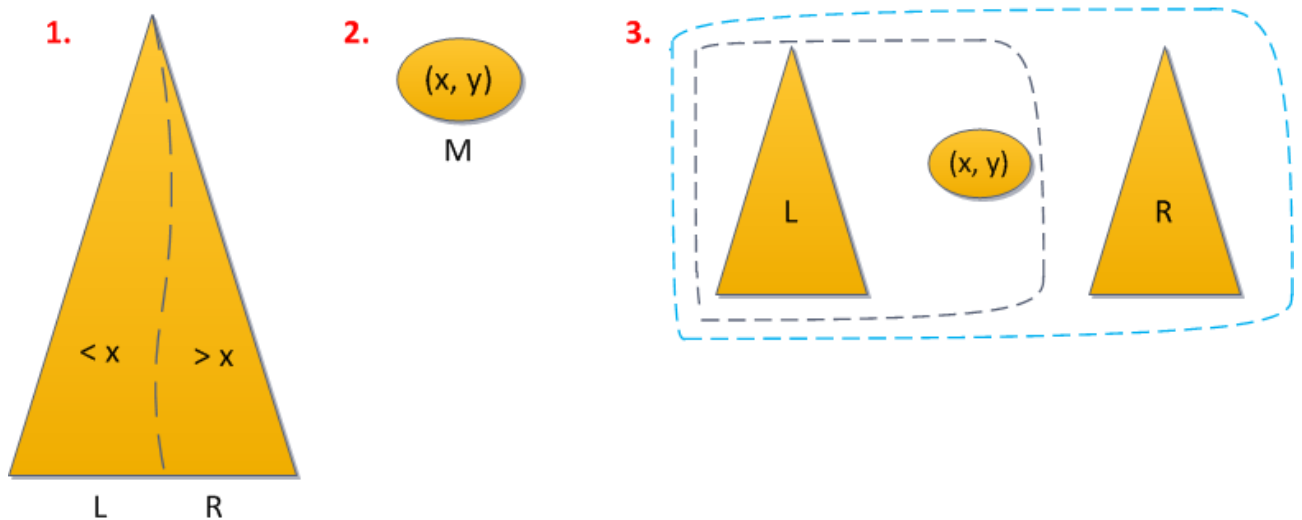


Рис. 5

Имеется 1 применение *Split* и 2 применения *Merge* — общее время работы $O(\log_2 N)$.

3.4. Remove

Удаление элемента:

Пусть нужно удалить из декартова дерева элемент с ключом x . Так как ключи в дереве, равные данному, могут встретиться только в левом поддереве вершины с ключом x , в правом не могут. Тогда совершим следующую последовательность действий:

1. Разделим сначала дерево по ключу $x-1$. Все элементы, меньшие либо равные $x-1$, отправляются в левый результат, значит, искомым элемент — в правом.

2. Разделим правый результат по ключу x . В новый правый результат отправляются все элементы с ключами, большими x , а в «средний» (левый от правого) — все меньшие либо равные x . Но поскольку строго меньшие после первого шага все были отсеяны, то среднее дерево и есть искомым элемент.

Теперь просто объединим снова левое дерево с правым, без среднего, и дерамида осталась без ключей x

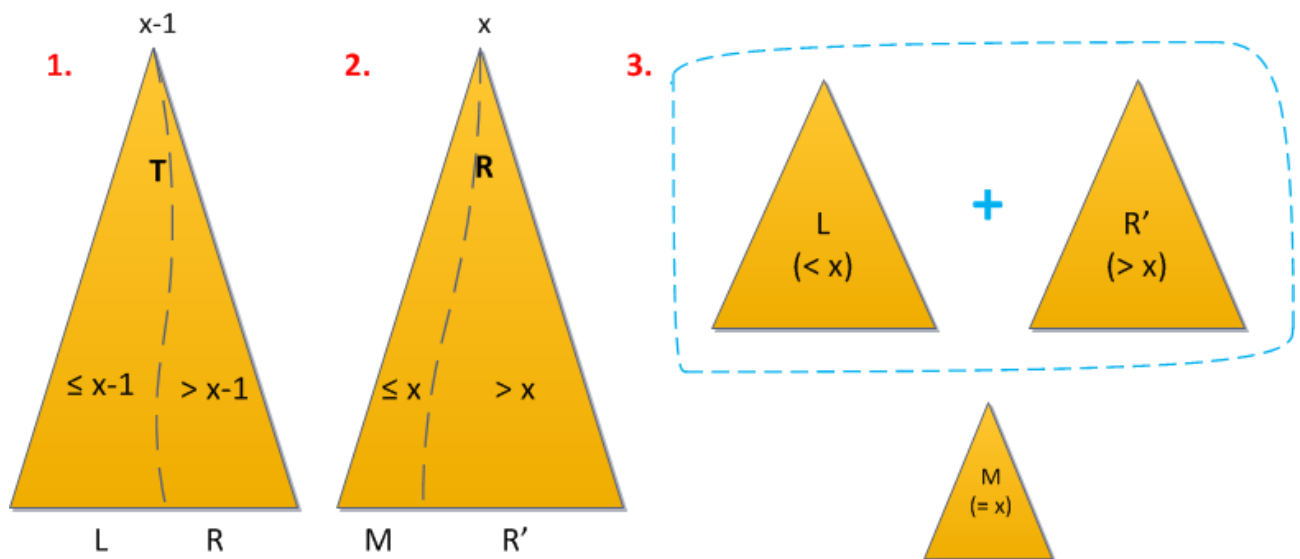


Рис. 6

Время работы операции все так же $O(\log_2 N)$, поскольку мы применили 2 раза *Split* и 1 раз *Merge*.

4. ОПИСАНИЕ ИНТЕРФЕЙСА

Взаимодействие с программой происходит через консоль:

- Ввод '1' – ввести данные с клавиатуры
- Ввод '2' – ввести данные с файла
- Ввод '0' – выход из программы

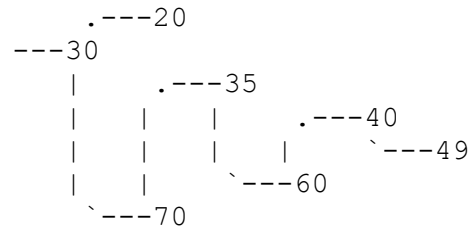
После выбора варианта ввода данных, пользователь может вводить данные для построения дерева. Дальнейшая работа зависит от команды пользователя:

- Ввод '1' – добавление элемента (далее пользователь вводит ключ и элемент вставляется в дерево)
- Ввод '2' – удаление элемента (далее пользователь вводит ключ и элемент удаляется из дерева)
- Ввод '0' – вернуться назад и начать вводить новые данные (далее ожидается ввод очередного дерева или строки STOP для выхода в меню)

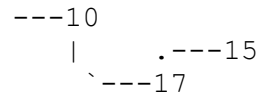
5. ТЕСТИРОВАНИЕ ПРОГРАММЫ

№	Входные данные	Выходные данные
1	10 60 20 70 30 49 40 35 15	<p>Построено дерево:</p> <pre> .---10 .---15 `---20 ---30 .---35 .---40 `---49 `---60 `---70 `---70 </pre> <p>Добавление элемента с ключом 5: Дерево T расщеплено по ключу 5: Левая половинка: Правая половинка:</p> <pre> .---10 .---15 `---20 ---30 .---35 .---40 `---49 `---60 `---70 `---70 </pre> <p>Левая половинка дерева сцеплена с элементом с ключом 5: ---5 Получившееся дерево сцеплено с правой половинкой:</p> <pre> .---5 .---10 .---15 `---20 ---30 .---35 .---40 `---49 `---60 `---70 `---70 </pre> <p>Добавление элемента завершено.</p>
2	10 60 20 70 30 49 40 35 15	<p>Построено дерево:</p> <pre> .---10 .---15 `---20 ---30 .---35 .---40 `---49 `---60 `---70 `---70 </pre> <p>Добавление элемента M с ключом 17: Дерево T расщеплено по ключу 17: Левая половинка L:</p> <pre> ---10 `---15 </pre>

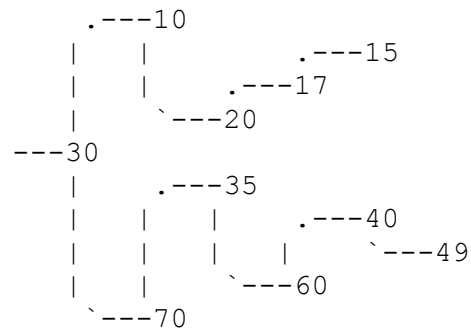
Правая половинка R:



Левая половинка L сцеплена с элементом M с ключом 17:

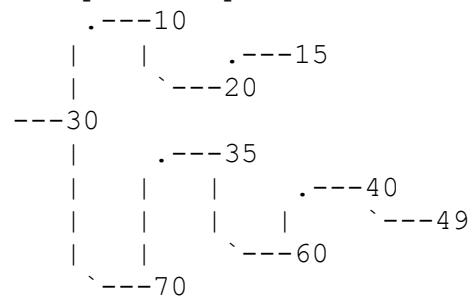


Получившееся дерево L+M сцеплено с правой половинкой R:



Добавление элемента завершено.

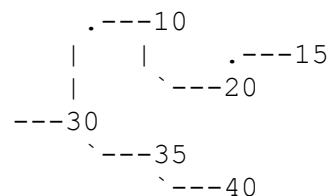
3 10 60 20 70 30 49 40 35 15 Построено дерево:



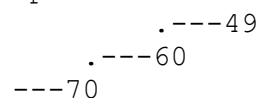
Добавление элемента M с ключом 41:

Дерево T расщеплено по ключу 41:

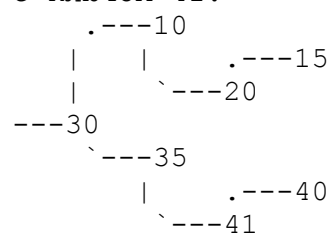
Левая половинка L:



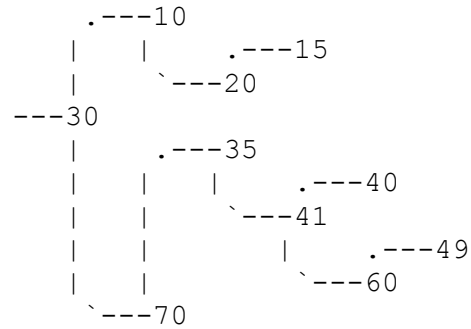
Правая половинка R:



Левая половинка L сцеплена с элементом M с ключом 41:

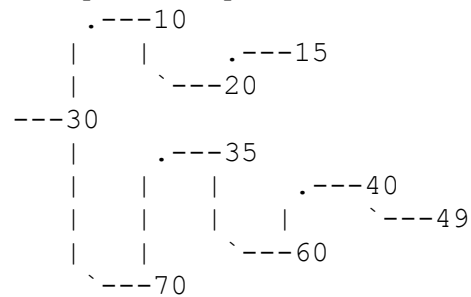


Получившееся дерево L+M сцеплено с правой половиной R:



Добавление элемента завершено.

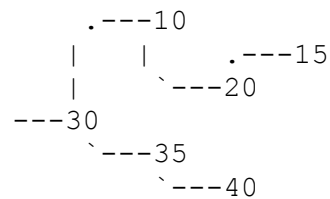
4 10 60 20 70 30 49 40 35 15 Построено дерево:



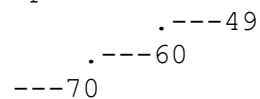
Добавление элемента М с ключом 41:

Дерево Т расщеплено по ключу 41:

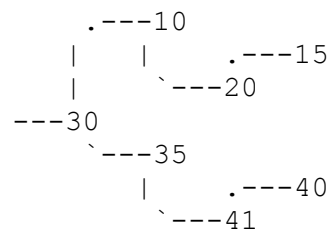
Левая половина L:



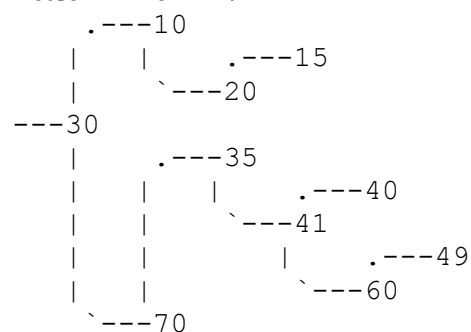
Правая половина R:



Левая половина L сцеплена с элементом М с ключом 41:

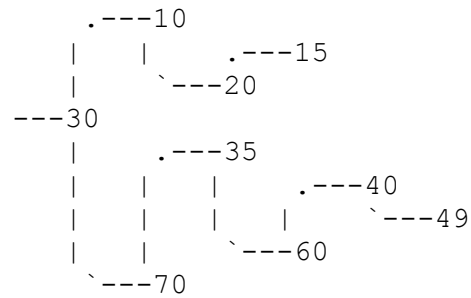


Получившееся дерево L+M сцеплено с правой половиной R:



Добавление элемента завершено.

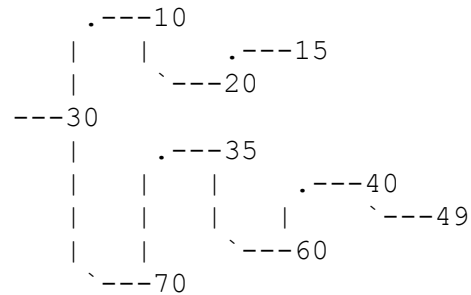
5 10 60 20 70 30 49 40 35 15 Построено дерево:



Добавление элемента М с ключом 420:

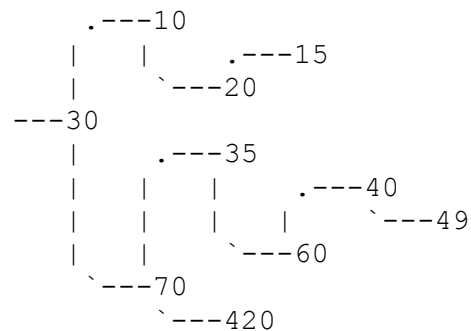
Дерево Т расщеплено по ключу 420:

Левая половинка L:

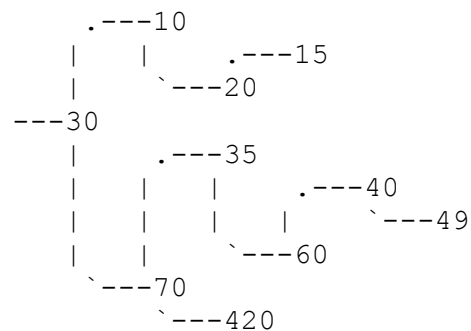


Правая половинка R:

Левая половинка L сцеплена с элементом М с ключом 420:

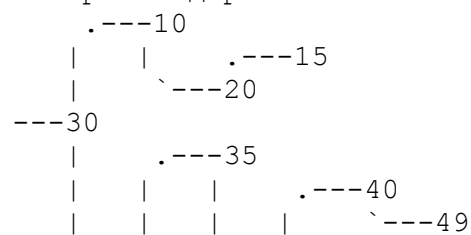


Получившееся дерево L+M сцеплено с правой половинкой R:



Добавление элемента завершено.

6 10 60 20 70 30 49 40 35 15 Построено дерево:



```

      |  |      `---60
      |  |      `---70

```

Удаление элемента с ключом 49:
 Дерево T расщеплено по ключу 48:
 Левая половина L:

```

      .---10
      |  |      .---15
      |  |      `---20
---30
      |  |      .---35
      |  |      `---40

```

Правая половина R:

```

      .---49
      .---60
---70

```

Правая половина расщеплена по ключу 48:

Её левая половина M:

```

---49

```

Её правая половина R':

```

      .---60
---70

```

Левая половина L сцеплена с правой
 половиной правого поддерева R'
 (исключили M):

```

      .---10
      |  |      .---15
      |  |      `---20
---30
      |  |      .---35
      |  |      |      .---40
      |  |      |      `---60
      |  |      |      `---70

```

Удаление элемента завершено.

7 10 60 20 70 30 49 40 35 15 Построено дерево:

```

      .---10
      |  |      .---15
      |  |      `---20
---30
      |  |      .---35
      |  |      |      .---40
      |  |      |      |      `---49
      |  |      |      |      `---60
      |  |      |      |      `---70

```

Удаление элемента с ключом 35:

Дерево T расщеплено по ключу 34:

Левая половина L:

```

      .---10
      |  |      .---15
      |  |      `---20
---30

```

Правая половина R:

```

      .---35
      |  |      .---40
      |  |      |      `---49
      |  |      |      `---60
---70

```

Правая половина расщеплена по ключу 34:

Её левая половина M:

---35

Её правая половинка R':

```
      .---40
      |    `---49
      .---60
---70
```

Левая половинка L сцеплена с правой половинкой правого поддерева R'

(исключили M):

```
      .---10
      |    |    .---15
      |    |    `---20
---30
      |    .---40
      |    |    `---49
      |    .---60
      `---70
```

Удаление элемента завершено.

8 10 60 20 70 30 49 40 35 15 Построено дерево:

```
      .---10
      |    |    .---15
      |    |    `---20
---30
      |    .---35
      |    |    |    .---40
      |    |    |    `---49
      |    |    |    `---60
      `---70
```

Удаление элемента с ключом 10:

Дерево T расщеплено по ключу 9:

Левая половинка L:

Правая половинка R:

```
      .---10
      |    |    .---15
      |    |    `---20
---30
      |    .---35
      |    |    |    .---40
      |    |    |    `---49
      |    |    |    `---60
      `---70
```

Правая половинка расщеплена по ключу 9:

Её левая половинка M:

---10

Её правая половинка R':

```
      .---15
      .---20
---30
      |    .---35
      |    |    |    .---40
      |    |    |    `---49
      |    |    |    `---60
      `---70
```

Левая половинка L сцеплена с правой половинкой правого поддерева R'

(исключили M):

```
      .---15
      .---20
```

```

---30
|
|   .---35
|   |   |   .---40
|   |   |   \---49
|   |   |   \---60
|   |   \---70

```

Удаление элемента завершено.

9 10 60 20 70 30 49 40 35 15 Построено дерево:

```

.---10
|   |   .---15
|   |   \---20
---30
|   .---35
|   |   |   .---40
|   |   |   \---49
|   |   |   \---60
|   |   \---70

```

Удаление элемента с ключом 30:

Дерево T расщеплено по ключу 29:

Левая половинка L:

```

---10
|   .---15
|   \---20

```

Правая половинка R:

```

---30
|   .---35
|   |   |   .---40
|   |   |   \---49
|   |   |   \---60
|   |   \---70

```

Правая половинка расщеплена по ключу 29:

Её левая половинка M:

```

---30
|   .---35
|   |   .---40
|   |   \---49
|   |   \---60
---70

```

Левая половинка L сцеплена с правой половинкой правого поддерева R'

(исключили M):

```

---10
|   .---15
|   .---20
|   |   \---35
|   |   |   .---40
|   |   |   \---49
|   |   |   \---60
|   |   \---70

```

Удаление элемента завершено.

Построено дерево:

```

.---10
|   \---15
.---20
|   \---30
|   |   .---35
|   |   \---40

```


Введён Вектор: 25
 Построено дерево:
 ---25
 1. Добавить элемент в дерамиду.
 2. Удалить элемент из дерамиды.
 0. Вернуться назад и ввести новые данные.
 2
 Введите значение ключа: 25
 Удаление элемента с ключом 25:
 Дерево T расщеплено по ключу 24:
 Левая половинка L:
 Правая половинка R:
 ---25
 Правая половинка расщеплена по ключу 24:
 Её левая половинка M:
 ---25
 Её правая половинка R':
 Левая половинка L сцеплена с правой
 половинкой правого поддерева R'
 (исключили M):
 Удаление элемента завершено.
 1. Добавить элемент в дерамиду.
 2. Удалить элемент из дерамиды.
 0. Вернуться назад и ввести новые данные.
 2
 Введите значение ключа: 30
 Дерево пусто. Удаление не проводится

ЗАКЛЮЧЕНИЕ

Была разработана программа, реализующая рандомизированную дерамиду поиска и операции вставки и удаления элемента в/из неё, а также наглядный вывод структуры и пояснения к операциям на экран.

Программа была успешно скомпилирована и протестирована.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Декартово дерево: Часть 1. Описание, операции, применения // Habr.
URL: <https://habr.com/ru/post/101818/> (дата обращения 29.12.2020)
2. Standard C++ Library reference // The C++ Resources Network. URL:
<http://www.cplusplus.com/reference/> (дата обращения 29.12.2020)

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <algorithm>
#include "Treap.h"
#include "Logger.h"

using namespace treap;

char kDefaultStopString[] = "STOP";
char kDefaultFileName[] = "input.txt";

void printTask(); // печатает задание
void printMenu(); // печатает меню
void printVector(const std::vector<int>& vec); // печатает вектор
void menu(); // вызывает меню
void consoleInput(); // организует ввод с консоли
void fileInput(); // организует ввод с файла
void performTask(std::istream& infile); // принимает поток чтения,
начинает работу программы
void treapMenu(TreapPtr& tree); // вызывает меню добавления/исключения
элемента из дерамиды
void printTreapMenu(); // печатает меню добавления/исключения элемента из
дерамиды

void printTask() {
    Logger::instance().log("Демонстрация добавления/удаления элемента в
рандомизированной дерамиде поиска.\n"
                           "Предполагается, что в дереве не могут
находиться элементы с одинаковыми ключами.\n");
}

void printMenu() {
```

```

        std::cout << "1. Ввести данные с клавиатуры.\n"
                    "2. Ввести данные с файла.\n"
                    "0. Выход из программы.\n";
    }

void printVector(const std::vector<int>& vec) {
    Logger::instance().log("Вектор: ");
    for (auto v : vec) {
        Logger::instance().log(std::to_string(v) + " ");
    }
    Logger::instance().log("\n");
}

void printTreapMenu() {
    std::cout <<"1. Добавить элемент в дерамиду.\n"
                "2. Удалить элемент из дерамиды.\n"
                "0. Вернуться назад.\n";
}

void treapMenu(TreapPtr& tree) {
    printTreapMenu();
    char c = '1';
    int key;
    do {
        std::cin >> c;
        std::cin.ignore(256, '\n');
        switch(c) {
            case '1':
                std::cout << "Введите значение ключа: ";
                std::cin >> key;
                insert(tree, key);
                break;
            case '2':
                std::cout << "Введите значение ключа: ";
                std::cin >> key;
                remove(tree, key);
                break;
            case '0':
                std::cout << "Возвращаюсь назад.\n\n";

```

```

        return;
    default:
        std::cout << "Неверное значение.\n";
        break;
    }
    if (c != '0') printTreapMenu();
} while (c != '0');
}

void menu() {
    printTask();
    printMenu();
    char c = '1';
    do {
        std::cin >> c;
        std::cin.ignore(256, '\n');
        switch(c) {
            case '1':
                consoleInput();
                break;
            case '2':
                fileInput();
                break;
            case '0':
                std::cout << "Выход из программы.\n";
                break;
            default:
                std::cout << "Неверное значение.\n";
                break;
        }
        if (c != '0') printMenu();
    } while (c != '0');
}

void consoleInput() {
    std::cout << "Вводите данные:\n"
                "Чтобы вернуться в меню, введите \"\" <<
kDefaultStopString << "\"\n";
    performTask(std::cin);
}

```

```

}

void fileInput() {
    std::string inputFileName;
    std::ifstream infile;
    std::cout << "Введите название файла:\n"
                "По умолчанию данные читаются из файла \"" <<
kDefaultFileName << "\".\n";
    getline(std::cin, inputFileName);

    if (inputFileName.empty()) {
        inputFileName = kDefaultFileName;
    }

    infile.open(inputFileName);
    if (!infile) {
        std::cout << "Файла \"" << inputFileName << "\" не существует.\n";
    } else {
        std::cout << "Чтение данных прекратится на строке \"" <<
kDefaultStopString << "\".\n";
        performTask(infile);
    }

    if (infile.is_open()) {
        infile.close();
    }
}

void performTask(std::istream& infile)
{
    std::string str;
    std::vector<int> vec;

    while (!infile.eof()) {
        getline(infile, str);
        if (str.empty()) continue;
        if (str == kDefaultStopString) {

```

```

        Logger::instance().log("Встретилась терминальная строка.\n\n");
        return;
    }

    const char *cstr = str.c_str();
    for (;;) { // преобразование строки в вектор чисел
        char* pEnd;
        const long i = std::strtoul(cstr, &pEnd, 10);
        if (cstr == pEnd) break;
        cstr = pEnd;
        vec.push_back(i);
    }

    Logger::instance().log("\nВведён "); printVector(vec);
    if (vec.empty()) continue;
    auto tree = build(vec);
    Logger::instance().log("Построено дерево:\n");
    printTree(tree, nullptr, false);

    treapMenu(tree);

    vec.clear();
}

}

int main() {
    try {
        srand(time(nullptr));
        menu();
    } catch (std::exception&) {
        std::cout << "menu(): Exception caught\n";
    }
    return 0;
}

```

Название файла: Treap.h

```
#ifndef TREAP_TREAP_H
#define TREAP_TREAP_H

#include <utility>
#include <memory>
#include <ctime>
#include <vector>
#include "Logger.h"

namespace treap {

    using TreapPtr = std::shared_ptr<class Treap>;
    using TreapPair = std::pair<TreapPtr, TreapPtr>;

    // Рандомизированная дерамида поиска (РДП)
    struct Treap {
        int key; // ключ
        int priority; // приоритет
        TreapPtr left;
        TreapPtr right;

        // конструктор, принимает ключ key, приоритет priority, указатели
на левое и правое поддеревья left и right
        Treap(int key, int priority = rand(), TreapPtr left = nullptr,
TreapPtr right = nullptr);

    };

    // принимает два дерева L и R, сливает их в одно и возвращает его
    TreapPtr merge(const TreapPtr& L, const TreapPtr& R);

    // принимает дерево T и по ключу key расщепляет его, возвращает пару
из левой и правой половинки
    TreapPair split(const TreapPtr& T, int key);

    // принимает дерево T и вставляет элемент с ключом key в него,
возвращает это же дерево
```



```

    TreapPtr insert(TreapPtr& T, int key);

    // принимает дерево T и удаляет элемент с ключом key из него,
    // возвращает это же дерево
    TreapPtr remove(TreapPtr& T, int key);

    // принимает вектор ключей keys и строит на их основе РДП
    TreapPtr build(const std::vector<int>& keys);

    // вспомогательная структура и набор функций для печати дерева
    struct Trunk
    {
        Trunk *prev;
        std::string str;

        Trunk(Trunk *prev, std::string str)
        {
            this->prev = prev;
            this->str = str;
        }
    };

    void showTrunks(Trunk *p);
    void printTree(TreapPtr root, Trunk *prev, bool isLeft);
}

#endif //TREAP_TREAP_H

```

Название файла: Treap.cpp

```

#include <iostream>
#include "Treap.h"

namespace treap {

    TreapPtr build(const std::vector<int>& keys) {
        Logger::instance().stream.setstate(std::ios_base::failbit);
        std::cout.setstate(std::ios_base::failbit);
    }
}

```

```

    auto tr = std::make_shared<Treap>(keys[0]);
    for (int i = 1; i < keys.size(); i++) {
        insert(tr, keys[i]);
    }

    Logger::instance().stream.clear();
    std::cout.clear();
    return tr;
}

Treap::Treap(int key, int priority, TreapPtr left, TreapPtr right)
    : key(key), priority(priority), left(left), right(right) {}

TreapPtr makeNode(int key) {
    return std::make_shared<Treap>(key);
}

TreapPtr merge(const TreapPtr& L, const TreapPtr& R) {
    if (L == nullptr) return R;
    if (R == nullptr) return L;

    TreapPtr merged = nullptr;
    if (L->priority > R->priority) {
        merged = merge(L->right, R);
        merged = std::make_shared<Treap>(L->key, L->priority, L-
>left, merged);
    } else {
        merged = merge(L, R->left);
        merged = std::make_shared<Treap>(R->key, R->priority, merged,
R->right);
    }
    return merged;
}

TreapPair split(const TreapPtr& T, int key) {
    if (T == nullptr) return TreapPair(nullptr, nullptr);

    if (T->key <= key) {
        TreapPair R = split(T->right, key);

```

```

        TreapPtr L = std::make_shared<Treap>(T->key, T->priority, T-
>left, R.first);
        return TreapPair(L, R.second);
    } else {
        TreapPair L = split(T->left, key);
        TreapPtr R = std::make_shared<Treap>(T->key, T->priority,
L.second, T->right);
        return TreapPair(L.first, R);
    }
}

TreapPtr insert(TreapPtr& T, int key) {
    Logger::instance().log("Добавление элемента М с ключом " +
std::to_string(key) + ":\n");
    TreapPtr newNode = std::make_shared<Treap>(key);

    TreapPair pair = split(T, key);
    Logger::instance().log("Дерево Т расщеплено по ключу " +
std::to_string(key) + ":\n");
    Logger::instance().log("Левая половинка L: \n");
    printTree(pair.first, nullptr, false);
    Logger::instance().log("Правая половинка R: \n");
    printTree(pair.second, nullptr, false);

    TreapPtr left = merge(pair.first, newNode);
    Logger::instance().log("Левая половинка L сцеплена с элементом М
с ключом " + std::to_string(key) + ":\n");
    printTree(left, nullptr, false);

    T = merge(left, pair.second);
    Logger::instance().log("Получившееся дерево L+М сцеплено с правой
половинкой R:\n");
    printTree(T, nullptr, false);

    Logger::instance().log("Добавление элемента завершено.\n");
    return T;
}

TreapPtr remove(TreapPtr& T, int key) {

```

```

        Logger::instance().log("Удаление элемента с ключом " +
std::to_string(key) + ":\n");

        TreapPair pair = split(T, key-1);
        Logger::instance().log("Дерево T расщеплено по ключу " +
std::to_string(key-1) + ":\n");
        Logger::instance().log("Левая половинка L: \n");
        printTree(pair.first, nullptr, false);
        Logger::instance().log("Правая половинка R: \n");
        printTree(pair.second, nullptr, false);

        TreapPair pairR = split(pair.second, key);
        Logger::instance().log("Правая половинка расщеплена по ключу " +
std::to_string(key-1) + ":\n");
        Logger::instance().log("Её левая половинка M: \n");
        printTree(pairR.first, nullptr, false);
        Logger::instance().log("Её правая половинка R': \n");
        printTree(pairR.second, nullptr, false);

        T = merge(pair.first, pairR.second);
        Logger::instance().log("Левая половинка L сцеплена с правой
половинкой правого поддерева R' (исключили M):\n");
        printTree(T, nullptr, false);

        Logger::instance().log("Удаление элемента завершено.\n");
        return T;
    }

    void print(const TreapPtr& T) {
        if (T == nullptr) return;
        Logger::instance().log("x=" + std::to_string(T->key) + "; y=" +
std::to_string(T->priority) + /*"; c=" + std::to_string(count) + */"\n");
        print(T->left);
        print(T->right);
    }
}

```

```

void showTrunks(Trunk *p)
{
    if (p == nullptr) return;
    showTrunks(p->prev);
    Logger::instance().log(p->str);
}

void printTree(TreapPtr root, Trunk *prev, bool isLeft)
{
    if (root == nullptr) return;

    std::string prev_str = "    ";
    Trunk *trunk = new Trunk(prev, prev_str);
    printTree(root->left, trunk, true);

    if (!prev)
        trunk->str = "---";
    else if (isLeft)
    {
        trunk->str = ".---";
        prev_str = "    |";
    }
    else
    {
        trunk->str = "`---";
        prev->str = prev_str;
    }

    showTrunks(trunk);
    Logger::instance().log(std::to_string(root->key) + "\n");

    if (prev)
        prev->str = prev_str;
    trunk->str = "    |";

    printTree(root->right, trunk, false);
}

```

```
}
```

Название файла: Logger.h

```
#ifndef TREAP_LOGGER_H
#define TREAP_LOGGER_H

#include <iostream>
#include <fstream>
#include <string>
#include <ctime>

class Logger {
public:
    static Logger& instance();
    void log(const std::string& str, bool toConsole = true, bool toFile =
true);

private:
    Logger();
    ~Logger();
    Logger(const Logger&) = delete;
    Logger(Logger&&) = delete;
    Logger& operator=(const Logger&) = delete;
    Logger& operator=(Logger&&) = delete;

    std::ofstream stream;
};

#endif //TREAP_LOGGER_H
```

Название файла: Logger.cpp

```
#include "Logger.h"

Logger::Logger() {
    std::time_t t = std::time(nullptr);
    std::tm* now = std::localtime(&t);
```

```

        char logFileName[32];
        strftime(logFileName, 32, "log_%F_%T.txt", now);
        stream.open(logFileName);
    }

    Logger::~~Logger() {
        stream.close();
    }

    Logger& Logger::instance() {
        static Logger instance;
        return instance;
    }

    void Logger::log(const std::string& str, bool toConsole, bool toFile) {
        if (toConsole) std::cout << str;
        if (toFile) stream << str;
    }

```