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

ОТЧЕТ

по лабораторной работе №5
по дисциплине «Алгоритмы и структуры данных»
Тема: Бинарные деревья поиска

Студент гр. 9381 _____ Прибылов Н.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Изучить и реализовать рандомизированную дерамиду поиска (декартово дерево, treap).

Задание.

Вариант 12

БДП: Рандомизированная дерамида поиска (treap); действие: 1+2a:

1) По заданной последовательности элементов Elem построить структуру данных определённого типа – БДП или хеш-таблицу;

2a) Для построенной структуры данных проверить, входит ли в неё элемент e типа Elem, и если входит, то в скольких экземплярах. Добавить элемент e в структуру данных. Предусмотреть возможность повторного выполнения с другим элементом.

Основные теоретические положения.

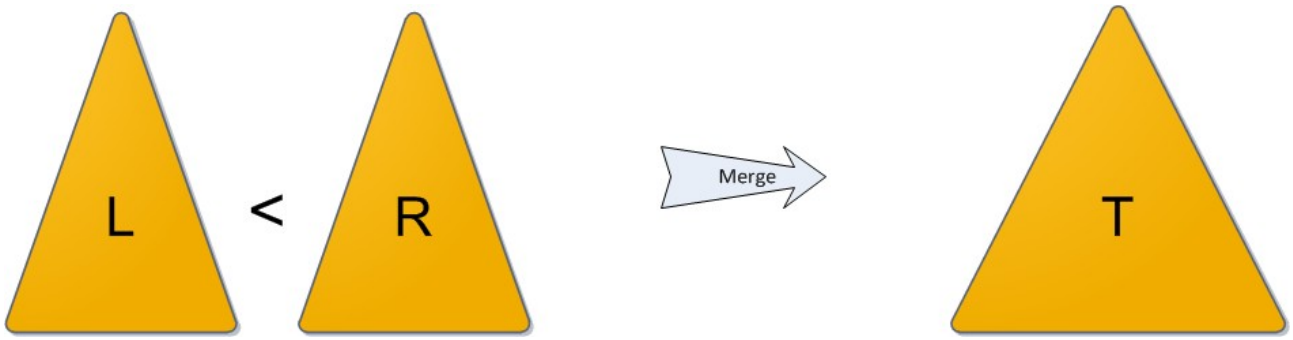
Пусть имеются данные дерева — ключи x. Добавим к ним еще один параметр в пару — y, и назовем его приоритетом. Теперь построим такое дерево, которое хранит в каждой вершине по два параметра, и при этом по ключам является двоичным деревом поиска, а по приоритетам — кучей. Такое дерево называется и декартовым. Если приоритеты генерировать случайным образом, такая структура называется рандомизированной дерамидой поиска.

В англоязычной литературе такая структура имеет название treap, которое наглядно показывает 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).

Описание алгоритмов.

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

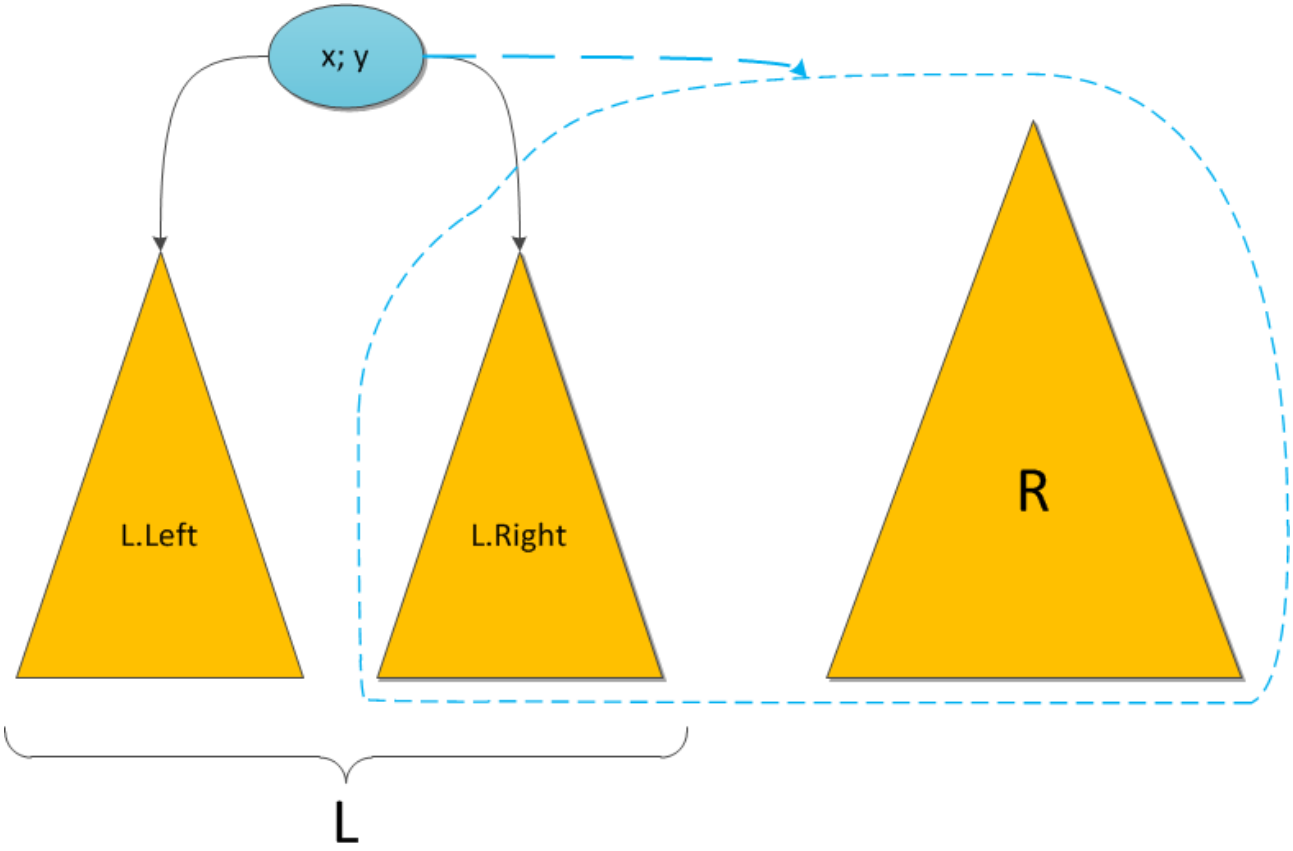
Операция *Merge*: принимает на вход два декартовых дерева *L* и *R*. От нее требуется слить их в одно, тоже корректное, декартово дерево *T*. Следует заметить, что работать операция *Merge* может не с любыми парами деревьев, а только с теми, у которых все ключи одного дерева (*L*) не превышают ключей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



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 *y* 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.

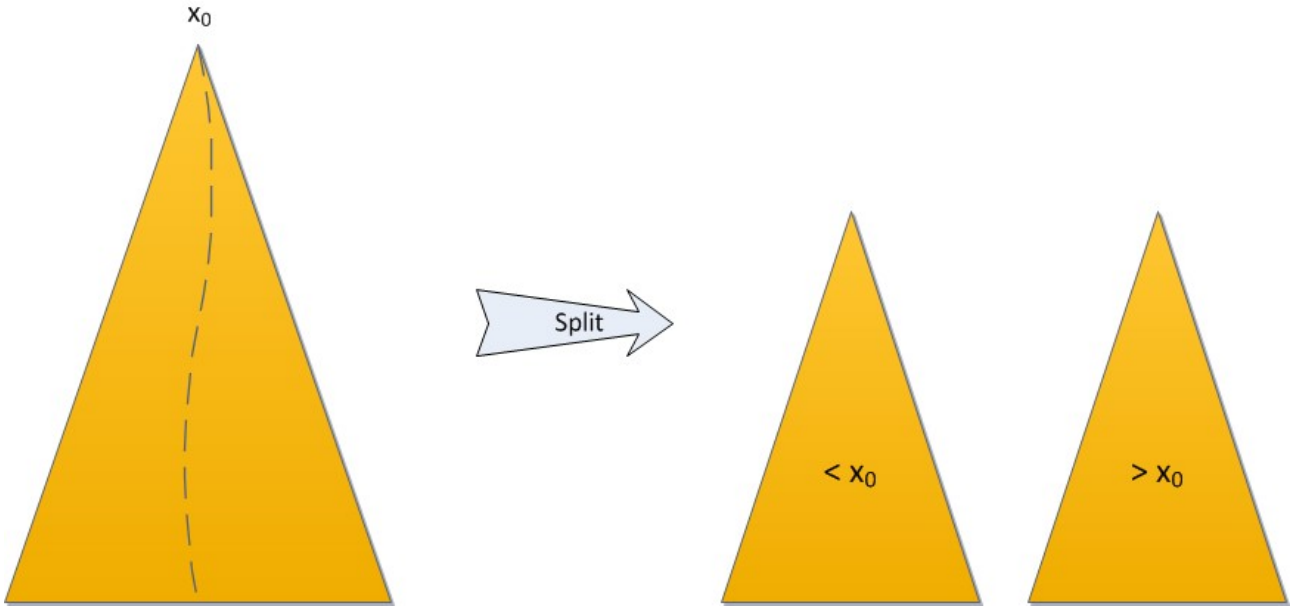
Легко понять, что все дерево *R* окажется в правом поддереве нового корня: его ключи больше *x* по условию. Точно так же левое поддерево старого корня *L.Left* имеет все ключи, меньшие *x*, и должно остаться левым поддеревом. Остаётся правое поддерево *L.Right:* рекурсивно вызываем *Merge* для *L.Right*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



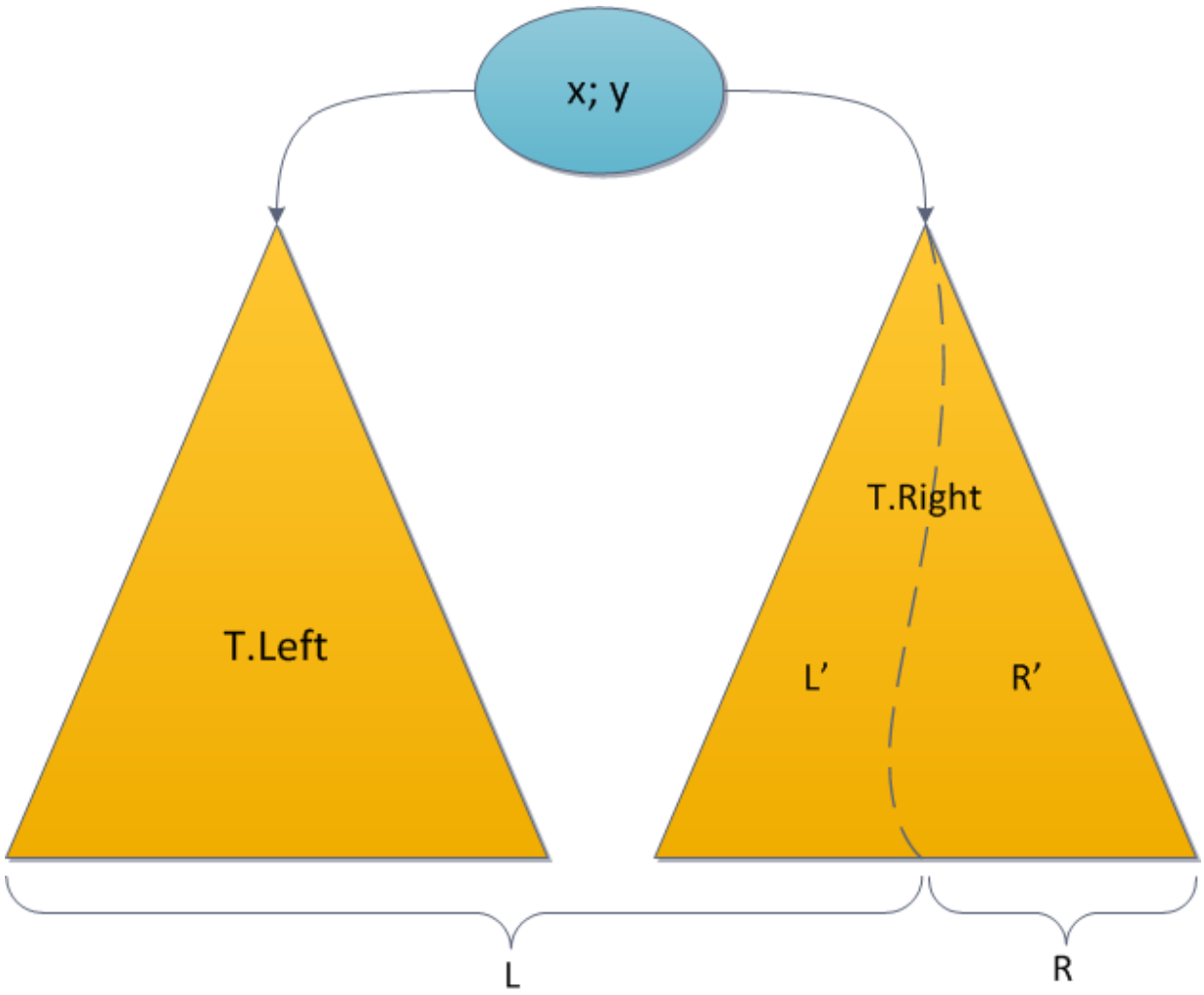
Симметричный случай — когда приоритет в корне дерева *R* выше — разбирается аналогично.

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



Если ключ корня меньше *x0*, то корень окажется в *L*, иначе в *R*. Предположим для однозначности, что ключ корня оказался меньше *x0*.

Тогда очевидно, что все элементы левого поддерева *T* также окажутся в *L* — их ключи тоже будут меньше *x0*. Более того, корень *T* будет и корнем *L*, поскольку его приоритет наибольший во всем дереве. Левое поддерево корня полностью сохранится без изменений, а вот правое уменьшится — из него придется убрать элементы с ключами, большими *x0*, и вынести в дерево *R*. А остаток ключей сохранить как новое правое поддерево *L*. Здесь снова можно прибегнуть к рекурсии. Возьмем правое поддерево и рекурсивно разрежем его по тому же ключу *x0* на два дерева *L′* и *R′*. После чего становится ясно, что *L′* станет новым правым поддеревом дерева *L*, а *R′* и есть непосредственно дерево *R* — оно состоит из тех и только тех элементов, которые больше *x0*.



Симметричный случай, при котором ключ корня больше, чем *x0*, 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.

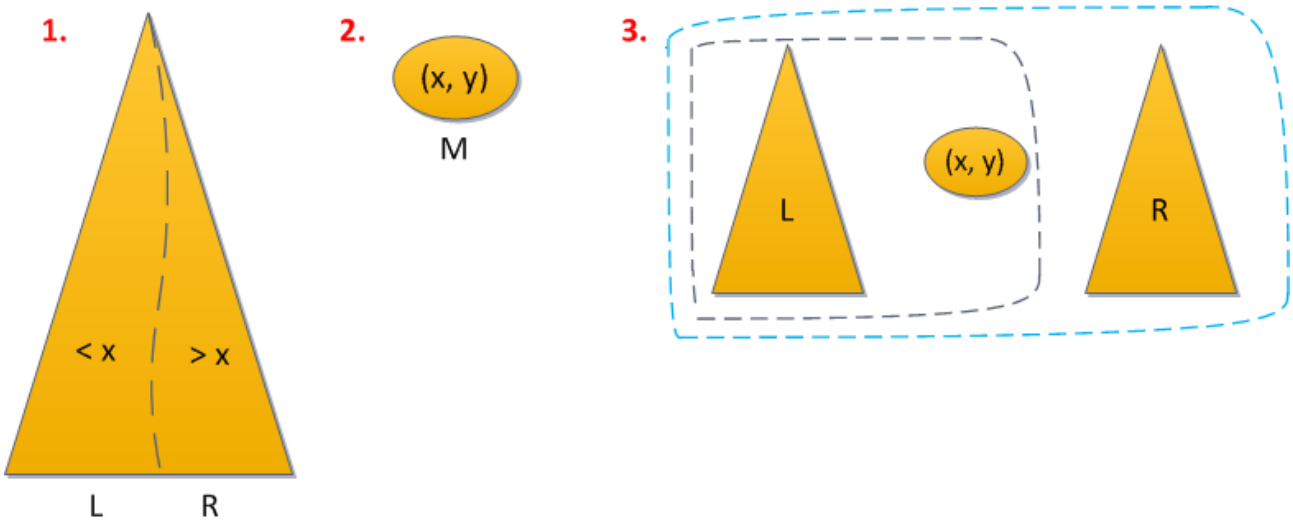
Добавление элемента:

Используя универсальность операций *Split*/*Merge*:

1. Разделим (*split*) дерево по ключу *x* на дерево *L*, с ключами меньше икса, 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*.



Описание структур данных и функций.

void printTask(); // печатает задание

void printMenu(); // печатает меню

void printVector(const std::vector<int>& vec); // принимает вектор и печатает его

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

struct Treap — структура дерамиды.

Поля класса:

int key — ключ

int priority — приоритет

TreapPtr left — указатель на левое поддерево

TreapPtr right — указатель на правое поддерево

int count — счётчик одинаковых ключей

Методы класса:

Treap(int key, int priority, TreapPtr left = nullptr, TreapPtr right = nullptr) — конструктор, принимает ключ, приоритет, указатели на левое и правое поддерево

static TreapPtr merge(TreapPtr L, TreapPtr R) — принимает указатели на два дерева и сливает их в одно, возвращает полученное дерево

void split(int key, TreapPtr& L, TreapPtr& R) const — принимает ключ, ссылки на указатели на два дерева и по ключу расщепляет дерево, помещая левую и правую части в переданные указатели

TreapPtr add(int key) — принимает ключ, добавляет его в дерево, возвращает дерево с новым ключом (не меняя исходное)

Treap search(int key)* — принимает ключ и ищет его в дереве, возвращает узел с этим ключом (по сути, поддерево, в корне которого нужный ключ), либо, если ключ не найден, *nullptr*

void print() — печатает узлы дерева в КЛП-порядке

Дополнительные функции, не являющиеся методами класса:

int generateRandom() — генерирует псевдослучайное целое число и возвращает его

TreapPtr buildTreap(const std::vector<int>& keys) — принимает вектор ключей и создаёт дерево на его основе, возвращает указатель на дерево

class Logger — вспомогательный класс для логгирования промежуточных результатов.

Методы класса:

static Logger& instance() — возвращает экземпляр класса.

void log(const std::string& str, bool toConsole = true, bool toFile = true) — принимает строку, которую нужно внести в лог, и две опции — печатать в консоль и/или в файл.

Logger() — конструктор, создаёт файл лога и открывает его.

~Logger()— деструктор, закрывает файл лога.

Конструкторы копирования, перемещения, операторы присваивания объявлены удалёнными во избежание случайного дублирования экземпляра класса.

Разработанный программный код см. в приложении А.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные
1	23 69 45 23	Элемент с ключом 23 встретился 1 раз(а).
2	-10 20 -30 40 -50 60 -70 20	Элемент с ключом 20 встретился 1 раз(а).
3	5 6 -8 -7 7 8 -6 -5 1 2 8 -4 -3 3 4 -2 -1 8	Элемент с ключом 8 встретился 2 раз(а).
4	9 8 7 6 5 4 3 2 1 0	Элемент с ключом 0 встретился 0 раз(а).

Выводы.

Была изучена и реализована рандомизированная дерамида поиска.

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 printTask() {
    Logger::instance().log("Рандомизированная дерамида поиска.\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 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::strtol(cstr, &pEnd, 10);
                if (cstr == pEnd) break;
                cstr = pEnd;
                vec.push_back(i);
            }

            Logger::instance().log("\nВведён "); printVector(vec);
            auto tree = buildTreap(vec);
            tree->print();
            auto found = tree->search(vec.back());
                      Logger::instance().log("Элемент с ключом " +
std::to_string(vec.back())
                                                    + " встретился " +
std::to_string(found ? found->count : 0) + " раз(а).\n");

            vec.clear();
        }
    }

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

Название файла: Logger.h

```
    #ifndef ALG_LAB3_LOGGER_H
    #define ALG_LAB3_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);
           void logNodeOperatorEquals(const std::string& first, const
std::string& second, bool res);
    private:
        Logger();
        ~Logger();
```

```
    Logger(const Logger&) = delete;
    Logger(Logger&&) = delete;
    Logger& operator=(const Logger&) = delete;
    Logger& operator=(Logger&&) = delete;

    static Logger logger;
    std::ofstream stream;
};


#endif //ALG_LAB3_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 << '\n';
    if (toFile) stream << str << '\n';
}
```

Название файла: Treap.h

```
#ifndef TREAP_TREAP_H
#define TREAP_TREAP_H

#include <utility>
#include <memory>
#include <ctime>
#include <vector>

namespace treap {

    using TreapPtr = std::shared_ptr<class Treap>;

    struct Treap {
        int key;
        int priority;
```

```
            TreapPtr left;
            TreapPtr right;
            int count = 1;

                Treap(int key, int priority, TreapPtr left = nullptr,
TreapPtr right = nullptr);
            static TreapPtr merge(TreapPtr L, TreapPtr R);
            void split(int key, TreapPtr& L, TreapPtr& R) const;
            TreapPtr add(int key);
            Treap* search(int key);
            void print();
        };

        int generateRandom();
        TreapPtr buildTreap(const std::vector<int>& keys);
    }

    #endif //TREAP_TREAP_H
```

Название файла: Treap.cpp

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

    namespace treap {

        int generateRandom() {
            return std::rand();
        }

        TreapPtr buildTreap(const std::vector<int>& keys) {
                        auto tr = std::make_shared<Treap>(keys[0],
generateRandom());
            for (int i = 1; i < keys.size()-1; i++) {
                auto node = tr->search(keys[i]);
                if (node)
                    node->count++;
                else
                    tr = tr->add(keys[i]);
            }
            return tr;
        }

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

        TreapPtr Treap::merge(TreapPtr L, TreapPtr R) {
            if (L == nullptr) return R;
            if (R == nullptr) return L;

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

```
                    return std::make_shared<Treap>(R->key, R->priority,
merge(L, R->left), R->right);
            }
        }

        void Treap::split(int key, TreapPtr& L, TreapPtr& R) const {
            TreapPtr newTreap = nullptr;
            if (this->key <= key) {
                if (right == nullptr)
                    R = nullptr;
                else
                    right->split(key, newTreap, R);
                 L = std::make_shared<Treap>(this->key, priority, left,
newTreap);
            } else {
                if (left == nullptr)
                    L = nullptr;
                else
                    left->split(key, L, newTreap);
                      R = std::make_shared<Treap>(this->key, priority,
newTreap, right);
            }
        }

        TreapPtr Treap::add(int key) {
            TreapPtr l = nullptr, r = nullptr;
            split(key, l, r);
                        TreapPtr tmp = std::make_shared<Treap>(key,
generateRandom());
            return merge(merge(l, tmp), r);
        }

        Treap* Treap::search(int key) {
            if (key == this->key) return this;
            if (key < this->key) {
                if (left) return left->search(key);
            } else {
                if (right) return right->search(key);
            }
            return nullptr;
        }

        void Treap::print() {
            std::cout << "x=" << key << "; y=" << priority << "\n";
            if (left) left->print();
            if (right) right->print();
        }

    }
```