**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра математического обеспечения и применения ЭВМ**

**КУРСОВАЯ РАБОТА**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: Рандомизированные бинарные деревья поиска**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 7382 |  | Дерябина П.С. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2018

**ЗАДАНИЕ**

**на курсовую работу**

|  |  |  |
| --- | --- | --- |
| Студентка Дерябина П.С. | | |
| Группа 7382 | | |
| Тема работы : Рандомизированные бинарные деревья поиска - исследование | | |
| Содержание пояснительной записки:   1. Введение 2. Формальная постановка задачи 3. Описание алгоритма 4. Описание структур данных и функций 5. Тестирование 6. Исследование 7. Заключение | | |
| Предполагаемый объем пояснительной записки:  Не менее 30 страниц. | | |
| Дата выдачи задания: 19.10.2018 | | |
| Дата сдачи курсовой работы: | | |
| Дата защиты курсовой работы: | | |
| Студентка |  | Дерябина П.С. |
| Преподаватель |  | Фирсов М.А. |

**Аннотация**

В ходе курсовой работы требуется разработать программу на языке программирования Cи. Программа должна производить работу с рандомизированным бинарным деревом поиска все элементы которого различны. Необходимо исследовать глубину полученного дерева и скорость работы алгоритма в зависимости от входных данных.

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 5](#_Toc531123314)

[1 ЦЕЛЬ РАБОТЫ 6](#_Toc531123315)

[2 ПОСТАНОВКА ЗАДАЧИ И СПЕЦИФИКАЦИЯ ПРОГРАММЫ 6](#_Toc531123316)

[2.1 Постановка задачи. 6](#_Toc531123317)

[2.2 Спецификация программы. 7](#_Toc531123318)

[3 ОПИСАНИЕ АЛГОРИТМА 7](#_Toc531123319)

[4 ОПИСАНИЕ ОСНОВНЫХ ФУНКЦИЙ И СТРУКТУР ПРОГРАММЫ 8](#_Toc531123320)

[4.1 Описание основных функций. 8](#_Toc531123321)

[4.2 Описание структур данных. 9](#_Toc531123322)

[5 ТЕСТИРОВАНИЕ 9](#_Toc531123323)

[6 ИССЛЕДОВАНИЕ 19](#_Toc531123324)

[ПРИЛОЖЕНИЕ А 21](#_Toc531123325)

[ПРИЛОЖЕНИЕ Б 22](#_Toc531123326)

[ПРИЛОЖЕНИЕ B 25](#_Toc531123327)

# ВВЕДЕНИЕ

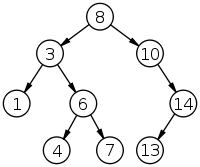
[](https://commons.wikimedia.org/wiki/File:Binary_search_tree.svg?uselang=ru)

Рис.1 - Пример бинарного(двоичного) дерева поиска

Двоичное дерево поиска - это [двоичное дерево](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE), для которого выполняются следующие дополнительные условия (свойства дерева поиска):

Оба поддерева — левое и правое — являются двоичными деревьями поиска.

У всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X.

У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равны, нежели значение ключа данных самого узла X.

Очевидно, данные в каждом узле должны обладать ключами, на которых определена операция сравнения меньше.

Особенность же рандомизированного дерева состоит в том, что каждый из узлов бинарного дерева может быть корнем с вероятностью , где — количество узлов дерева.

Для целей реализации двоичное дерево поиска можно определить так:

Двоичное дерево состоит из узлов (вершин) — записей вида (data, left, right), где data — некоторые данные, привязанные к узлу, left и right — ссылки на узлы, являющиеся детьми данного узла — левый и правый сыновья соответственно. Для осуществления рандомизированности дерева каждый узле также содержит поле size, хранящее значение количества узлов поддерева, корнем которого он является.

Данные (data) обладают ключом (key), на котором определена операция сравнения «меньше». В конкретных реализациях это может быть пара (key, value) — (ключ и значение), или ссылка на такую пару, или простое определение операции сравнения на необходимой структуре данных или ссылке на неё.

Для любого узла X выполняются свойства дерева поиска: key[left[X]] < key[X] ≤ key[right[X]], то есть ключи данных родительского узла больше ключей данных левого сына и нестрого меньше ключей данных правого.

Двоичное дерево поиска не следует путать с [двоичной кучей](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D0%BA%D1%83%D1%87%D0%B0), построенной по другим правилам.

Основным преимуществом двоичного дерева поиска перед другими [структурами данных](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) является возможная высокая эффективность реализации основанных на нём алгоритмов [поиска](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B8%D1%81%D0%BA_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) и [сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8).

Двоичное дерево поиска применяется для построения более абстрактных структур, таких, как [множества](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE_(%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)), [мультимножества](https://ru.wikipedia.org/wiki/%D0%9C%D1%83%D0%BB%D1%8C%D1%82%D0%B8%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE), [ассоциативные массивы](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%BE%D1%86%D0%B8%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2).

# ЦЕЛЬ РАБОТЫ

Разработать программу работы с рандомизированным бинарным деревом поиска все элементы которого различны. Сгенерировать тестовые данные с критериями. Исследовать зависимость скорости работы алгоритма и глубины полученного дерева от изменения количества входных данных.

# ПОСТАНОВКА ЗАДАЧИ И СПЕЦИФИКАЦИЯ ПРОГРАММЫ

## Постановка задачи.

Для 11 варианта задания необходимо:

По заданному файлу F, все элементы которого различны, построить случайное БДП с рандомизацией. Исследование в среднем и худшем случае.

## Спецификация программы.

Входные данные: последовательность различных символов, считанная с файла или введенная с консоли. Предусмотрена возможность отсутствия входных данных — программа их генерирует при соответствующей команде.

Выходные данные: дерево, реализованное по принципу БДП и БДП с рандомизацией, время работы алгоритма для обоих случаев, значение глубины, получившихся деревьев для обоих случаев.

# ОПИСАНИЕ АЛГОРИТМА

Для решения задачи построения РБДП используется динамическое представление бинарного дерева поиска. Изначально создается пустое бинарное дерево поиска. Далее последовательно создается бинарное дерево поиска с помощью функции CreateRBST.

Последовательно перебираются все ключи файла F, и для каждого

создается узел, отвечающий условиям: правый сын больше родителя, а левый сын — меньше. При этом каждая вставка элемента может происходить в корень с вероятностью , где — текущее количество узлов дерева.

# ОПИСАНИЕ ОСНОВНЫХ ФУНКЦИЙ И СТРУКТУР ПРОГРАММЫ

## Описание основных функций.

**int GetElements(float \*keys, char\* keys\_ch, char \*str, size\_t fixtype).**

Принимает указатель keys на float и указатель keys\_ch на char, в один из которых потом поместятся значения ключей файла F; строку str в которой хранятся значения ключей файла F; переменную size, хранящую размер типа char или float. В зависимости от значения переменной size функция будет осуществлять считываний символов в массив keys\_ch или в массив keys.

**int fixsize(Node\* tree).**

Принимает указатель на структуру Node. Функция присваивает полю size каждого узла дерева tree значение, равное весам узлов в дереве tree.

**Node\* RotateRight(Node \*tree, size\_t size, int level)** и

**Node\* RotateLeft(Node\* tree, size\_t size, int level).**

RotateRight принимает указатель на Node, который хранит дерево tree; переменную size, хранящую размер типа char или float; переменную level со значение глубины рекурсии функции, которая вызвала RotateRight. Функция осуществляет поворот вправо, благодаря чему левый сын корневого дерева поднимается на один уровень дерева tree.

Функция RotateLeft аналогична функции RotateRight.

**Node\* Insert(Node\* tree, void\* key, size\_t size, int level).**

Принимает указатель на структуру Node, которая хранит дерево tree, переменную key со значением ключа, который нужно вставить в дерево; переменную size со значением размера char или float; переменную level со значением глубины рекурсии. Функция производит обход КЛП дерева tree, и, если текущий узел пуст, то вставляем в него ключ key.

**Node\* InsertRoot(Node\* tree, void\* key, size\_t size, int level).**

Функция InsertRoot аналогична функции Insert, но после вставки в

первый пустой узел функции InsertRoot вызывает фукнции RotateRight

и/или RotateLeft для того, чтобы поднять узел key из положения листа в

положения корня.

**void showtree(Node\* tree, size\_t size).**

Принимает указатель на структуру Node, которая хранит дерево tree и переменную size, хранящую размер типа char или float. С помощью обхода КЛП дерева tree функция выводит на экран значения текущего корня и его детей.

**void delete\_BT(Node\* tree).**

Принимает указатель на структуру Node, которая хранит дерево tree и удаляет все его узлы, совершая обход ЛПК.

**Node\* CreateRBST(Node\* tree, float\* keys\_f, char\* keys\_ch, size\_t type, int size)** и **Node\* CreateBST(Node\* tree, float\* keys\_f, char\* keys\_ch, size\_t type, int size).**

Функции принимают указатель на дерево, указатели на массивы char и float, размер типа дерева и количество узлов дерева. Обе функции обращаются к функциям Insert и InsertRoot, чтобы с помощью цикла создать бинарное дерево поиска с заданным количеством узлов. В случае функции CreateRBST дерево с рандомизацией, CreateBST — обычное.

## Описание структур данных.

typedef struct Node {

void\* key; // значение ключа случайного типа

int size; // размер дерева, корнем которого является ключ

struct Node\* left; // указатель на левого сына

struct Node\* right; // указатель на правого сына

} Node;

# ТЕСТИРОВАНИЕ

Протестируем программу различными способами: с корректными данными и нет.

В табл. 1 предоставлено тестирование программы с корректным вводом данных.

Таблица 1 – Корректное тестирование

|  |  |
| --- | --- |
| Исходные данные | Выходные данные |
| 2  a b c d e | RBST:  [null] <-- [a] --> [b]  [null] <-- [b] --> [c]  [null] <-- [c] --> [d]  [null] <-- [d] --> [e]  [null] <-- [e] --> [null]  BST:  [null] <-- [a] --> [b]  [null] <-- [b] --> [c]  [null] <-- [c] --> [d]  [null] <-- [d] --> [e]  [null] <-- [e] --> [null]  RBST BST  Depth: 5 Depth: 5  Time: 0.0000 Time: 0.0000 |
| 2  a b c d e f | RBST:  [a] <-- [f] --> [null]  [null] <-- [a] --> [b]  [null] <-- [b] --> [c]  [null] <-- [c] --> [d]  [null] <-- [d] --> [e]  [null] <-- [e] --> [null]  BST:  [null] <-- [a] --> [b]  [null] <-- [b] --> [c]  [null] <-- [c] --> [d]  [null] <-- [d] --> [e]  [null] <-- [e] --> [f]  [null] <-- [f] --> [null]  RBST BST  Depth: 6 Depth: 6  Time: 0.0000 Time: 0.0000 |
| 2  2 3.3 3.33 11 100 5 | RBST:  [2] <-- [5] --> [11]  [null] <-- [2] --> [3.3]  [null] <-- [3.3] --> [3.33]  [null] <-- [3.33] --> [null]  [null] <-- [11] --> [100]  [null] <-- [100] --> [null]  BST:  [null] <-- [2] --> [3.3]  [null] <-- [3.3] --> [3.33]  [null] <-- [3.33] --> [11]  [5] <-- [11] --> [100]  [null] <-- [5] --> [null]  [null] <-- [100] --> [null]  RBST BST  Depth: 4 Depth: 5  Time: 0.0000 Time: 0.0000 |
| 2  6.7 6.1 7.2 5.5 3 100 200 | RBST:  [6.7] <-- [100] --> [200]  [6.1] <-- [6.7] --> [7.2]  [5.5] <-- [6.1] --> [null]  [3] <-- [5.5] --> [null]  [null] <-- [3] --> [null]  [null] <-- [7.2] --> [null]  [null] <-- [200] --> [null]  BST:  [6.1] <-- [6.7] --> [7.2]  [5.5] <-- [6.1] --> [null]  [3] <-- [5.5] --> [null]  [null] <-- [3] --> [null]  [null] <-- [7.2] --> [100]  [null] <-- [100] --> [200]  [null] <-- [200] --> [null]  RBST BST  Depth: 5 Depth: 4  Time: 0.0000 Time: 0.0000 |
| 2  h g f e d c b a | RBST:  [b] <-- [c] --> [h]  [a] <-- [b] --> [null]  [null] <-- [a] --> [null]  [g] <-- [h] --> [null]  [f] <-- [g] --> [null]  [e] <-- [f] --> [null]  [d] <-- [e] --> [null]  [null] <-- [d] --> [null]  BST:  [g] <-- [h] --> [null]  [f] <-- [g] --> [null]  [e] <-- [f] --> [null]  [d] <-- [e] --> [null]  [c] <-- [d] --> [null]  [b] <-- [c] --> [null]  [a] <-- [b] --> [null]  [null] <-- [a] --> [null]  RBST BST  Depth: 6 Depth: 8  Time: 0.0000 Time: 0.0000 |
| 2  a z h g b o | RBST:  [a] <-- [o] --> [z]  [null] <-- [a] --> [h]  [g] <-- [h] --> [null]  [b] <-- [g] --> [null]  [null] <-- [b] --> [null]  [null] <-- [z] --> [null]  BST:  [null] <-- [a] --> [z]  [h] <-- [z] --> [null]  [g] <-- [h] --> [o]  [b] <-- [g] --> [null]  [null] <-- [b] --> [null]  [null] <-- [o] --> [null]  RBST BST  Depth: 5 Depth: 5  Time: 0.0000 Time: 0.0000 |
| 2  2 3 105 1 39 | RBST BST  Depth: 4 Depth: 4  Time: 0.0000 Time: 0.0000 |
| 2  a | RBST:  [null] <-- [a] --> [null]  BST:  [null] <-- [a] --> [null]  RBST BST  Depth: 1 Depth: 1  Time: 0.0000 Time: 0.0000 |

В табл. 2 предоставлено тестирование программы с некорректным вводом данных.

Таблица 2 – Некорректное тестирование

|  |  |
| --- | --- |
| Исходные данные | Выходные данные |
| 2  b c d 3 e f | Wrong input!  RBST:  Tree is empty!  BST:  Tree is empty!  RBST BST  Depth: 0 Depth: 0  Time: 0.0000 Time: 0.0000 |
| 2 | wrong input! |

# ИССЛЕДОВАНИЕ

Исследуем как распределяется значение высоты РБДП для разного количества узлов в двух случаях: лучшем и худшем.

Для худшего случая в головной функции сгенерировали массив строго возрастающих чисел. При заполнении обычного БДП таким массивом (перебирая его элементы последовательно) получили бы вырожденный случай – список – и линейное время обхода. В БДП с рандомизацией такой случай не исключен но маловероятен. Запустим программу на тестах для худшего случая для разного количества узлов, причем для каждого количества узлов будем запускать программу 100 раз, а результат усредним. Продемонстрируем зависимость значения высоты РБДП от количества узлов для худшего случая на рис.1: красная линия соответствует линейной зависимости значения высоты БДП, зеленая – зависимости значения высоты РБДП для худшего случая.

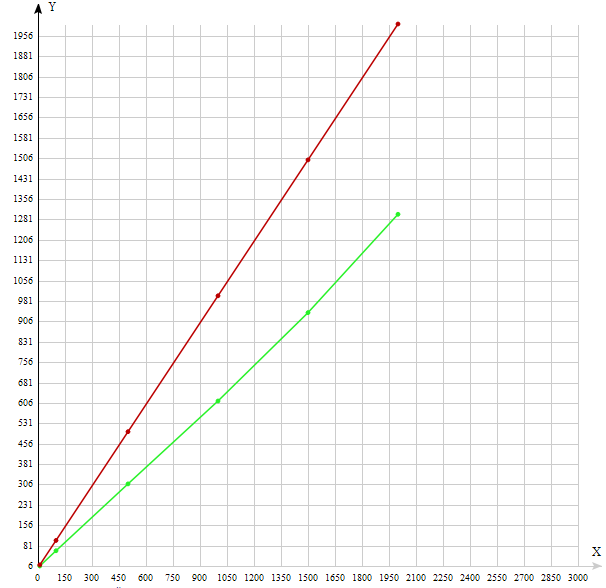


Рисунок 1 – Зависимость

Из рисунка видим, РБДП с рандомизацией является более сбалансированным даже в худшем случае.

Для среднего случая сгенерировали массив псевдослучайных чисел и таким же образом запустили программу по 100 раз для разного количества узлов, усреднив результат. Кроме, собственно, РБДП также создали БДП из тех же исходных данных и сравнили усредненные значения высоты РБДП и получившиеся значения высоты БДП. Полученная зависимость значения высоты дерева от количества узлов для двух случаев представлена на рис.2: фиолетовая линия – зависимость для БДП, желтая – РБДП.

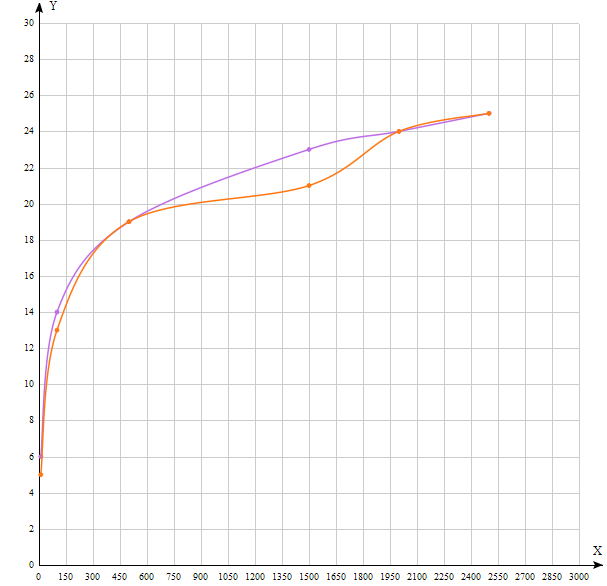


Рисунок 2 – Зависимость значения высоты РБДП и БДП от количества узлов в среднем случае.

Из рис. 2 видим, что зависимость для обоих алгоритмов примерно похожа в среднем случае.

Также исследовали скорость выполнения алгоритма построения РБДП, сравнив его со скоростью выполнения алгоритма построения БДП, результат представлен в табл.1 для худшего и лучшего случая.

Таблица 1 – Исследования скорости выполнения алгоритма.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кол-во узлов | РБДП | | БДП | |
| Средний случай | Худший случай | Средний случай | Худший случай |
| 10 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| 100 | 0.0011 | 0.0029 | 0.0007 | 0.0008 |
| 500 | 0.0077 | 0.0679 | 0.0055 | 0.0179 |
| 1000 | 0.0251 | 0.4626 | 0.0301 | 0.0315 |
| 1500 | 0.0482 | 3.9025 | 0.0483 | 0.0705 |
| 2000 | 0.0931 | 4.8556 | 0.1047 | 0.1432 |

Из табл.1 видим, что скорость выполнения алгоритма РБДП и БДП для среднего случая примерно похожи, а для худшего случая РБДП оказывается более медлительным, что объясняется необходимостью применять функции поворота направо и налево (RotateLeft и RotateRight) большее количество раз, так как доставка узлов из положения листа в положение корня требует будет занимать больше времени в худшем случае.

# ЗАКЛЮЧЕНИЕ

В ходе работы была разработана программа, реализующая работу рандомизированного бинарного дерева поиска случайного типа. Была исследована зависимость значения глубины дерева от количества узлов в нем: при увеличении количества узлов значение высоты РБДП будет существенно меньше, чем значение высоты БДП в худшем случае, также РБДП сводит вероятность вырождения бинарного дерева в связный список к минимуму. Согласно полученным данным также, оказалось, что время работы алгоритма построения РБДП в худшем случае будет существенно больше, чем время работы алгоритма построения БДП, а в среднем случае сильной разницы во времени работы двух алгоритмов разницы замечено не было. Полученные результаты примерно совпали теоретическими, согласно которым высота РБДП должна быть примерно в два раза больше, чем высота идеально сбалансированного дерева, — 2\*O(log n).

# ПРИЛОЖЕНИЕ А

**ИСХОДНЫЙ КОД API.h**

typedef struct Node {

void\* key;

int size;

struct Node\* left;

struct Node\* right;

} Node;

int explore(int a, int \*b, int\* c);

int GetDepth(Node\* tree);

Node\* CreateBST(Node\* tree, float\* keys\_f, char\* keys\_ch, size\_t type, int size);

Node\* CreateRBST(Node\* tree, float\* keys\_f, char\* keys\_ch, size\_t type, int size);

int fixsize(Node\* tree);

Node\* RotateRight(Node\* tree, size\_t size, int level);

Node\* RotateLeft(Node\* tree, size\_t size, int level);

Node\* InsertRoot(Node\* tree, void\* key, size\_t size, int level);

Node\* Insert(Node\* tree, void\* key, size\_t size, int level);

void showtree(Node\* tree, size\_t size);

void delete\_BT(Node\* tree);

int GetElements(float\* keys, char\* keys\_ch, char\* str, size\_t fixtype);

# ПРИЛОЖЕНИЕ Б

**ИСХОДНЫЙ КОД cw.c**

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#include <time.h>

#include "API.h"

#define N 1000

#define M 2000

int main()

{

srand(time(NULL)); // new settings for random number

char\* str = malloc(sizeof(char) \* N \* 2); // string for keys of RBST

char\* keys\_ch = NULL; // for char keys

float\* keys = NULL; // for float keys

int size = 0; // number of keys

void\* key;

void\* WorstKey = NULL;

void\* MiddleKeyR = NULL;

int flag = 0;

int format; // format of input

size\_t fixtype = 0; // type of keys

float\* middle = NULL;

float\* worst = NULL;

int WRDepth;

int MRDepth;

int WDepth;

int MDepth;

int times[6];

FILE \*f;

printf("Hello! This is randomized binary search tree (RBST) explorer!\nAvailable type of elements: char, float, int\n");

printf("Type '1', if you want input data from file 'input.txt'\nType '2' - from the keyboard\nType '3' - to generate input data: ");

scanf("%d", &format);

char c = getchar(); // takes '\n'

if (format != 1 && format != 2 && format != 3) {

printf("Wrong input format!\n");

return 0;

}

else if (format == 1) { // input from file

if ((f = fopen("input.txt", "r")) == NULL) {

printf("Couldn't open input.txt\n");

return 0;

}

fgets(str, N - 1, f);

printf("You entered: %s\n", str);

fclose(f);

}

else if (format == 2) {

printf("Input elements of RBST: ");

fgets(str, N - 1, stdin);

printf("You entered: %s\n", str);

}

else if (format == 3){

printf("Input number of elements (<=10000): ");

scanf("%d", &size);

c = getchar(); // takes '\n'

f = fopen("output.txt", "a");

fprintf(f, "\n\nИсследование рандомизированного бинарного дерева (RBST).\n\n");

middle = calloc(size, sizeof(float));

worst = calloc(size, sizeof(float));

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

middle[i] = rand()%(size\*10);

worst[i] = rand()%(size\*10);

if (i != 0){

while(worst[i] <= worst[i-1]){

worst[i] = worst[i-1] + 1 + rand()%10;

}

}

}

fprintf(f, "Сгенерированные данные для худшего случая (возрастающая последовательность): ");

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

fprintf(f, "[%g] ", worst[i]);

}

fprintf(f, "\n\n");

fprintf(f, "Сгенерированные данные для случайного распределения: ");

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

fprintf(f, "[%g] ", middle[i]);

}

fprintf(f, "\n\n");

fprintf(f, "Результаты построения RBST и BST на одних исходных данных\n");

fprintf(f, "\nRBST \t\t\t\tBST\n");

}

if (isalpha(str[0])) { // if first element of keys is symbol

fixtype = (sizeof(char));

keys\_ch = malloc(N \* sizeof(\*keys\_ch));

}

else if (isdigit(str[0])) { // if first element of keys is digit

fixtype = (sizeof(float));

keys = calloc(N, sizeof(float));

}

else if (format != 3){

printf("wrong input!\n");

return 0;

}

Node\* treeR = NULL;

Node\* tree = NULL;

Node\* WorstTreeR = NULL;

Node\* MiddleTreeR = NULL;

Node\* WorstTree = NULL;

Node\* MiddleTree = NULL;

size = GetElements(keys, keys\_ch, str, fixtype); // extract elements from string

if (format == 3){

times[0] = clock();

WorstTreeR = CreateRBST(WorstTreeR, worst, keys\_ch, sizeof(float), M);

times[1] = clock();

MiddleTreeR = CreateRBST(MiddleTreeR, middle, keys\_ch, sizeof(float), M);

times[2] = clock();

printf("\nworst randomized case:\n");

showtree(WorstTreeR, sizeof(float));

printf("\n\n");

printf("\nmiddle randomized case:\n");

showtree(MiddleTreeR, sizeof(float));

printf("\n\n");

WRDepth = GetDepth(WorstTreeR);

MRDepth = GetDepth(MiddleTreeR);

delete\_BT(WorstTreeR);

delete\_BT(MiddleTreeR);

times[4] = clock();

MiddleTree = CreateBST(MiddleTree, middle, keys\_ch, sizeof(float), M);

times[5] = clock();

printf("\nmiddle case:\n");

showtree(MiddleTree, sizeof(float));

printf("\n\n");

MDepth = GetDepth(MiddleTree);

fprintf(f, "Глубина (худший): %d\t\t\tГлубина (худший): %d\nГлубина (случайный): %d\t\t\tГлубина (случайный): %d\n", WRDepth, M, MRDepth, MDepth);

fprintf(f, "Время (худший): %.4f\t\t\tВремя (худший): -\nВремя (случайный): %.4f\t\tВремя (случайный): %.4f\n", (float)(times[1] - times[0])/CLOCKS\_PER\_SEC, (float)(times[2] - times[1])/CLOCKS\_PER\_SEC, (float)(times[5] - times[4])/CLOCKS\_PER\_SEC);

fclose(f);

}

else{

times[0] = clock();

treeR = CreateRBST(treeR, keys, keys\_ch, fixtype, size);

times[1] = clock();

times[2] = clock();

tree = CreateBST(tree, keys, keys\_ch, fixtype, size);

times[3] = clock();

printf("\nRBST:\n");

showtree(treeR, fixtype);

printf("\nBST:\n");

showtree(tree, fixtype);

printf("\n\n");

printf("RBST\t\t\t\tBST\n");

printf("Depth: %d\t\t\tDepth: %d\n", GetDepth(treeR), GetDepth(tree));

printf("Time: %.4f\t\t\tTime: %.4f\n", (float)(times[1]-times[0])/CLOCKS\_PER\_SEC, (float)(times[3]-times[2])/CLOCKS\_PER\_SEC);

}

int depths\_mid[100] = {0};

int depths\_wor[100] = {0};

int mid = 0;

int wor = 0;

/\* for(int i = 0; i < 100; i++){

explore(size, &depths\_mid[i], &depths\_wor[i]);

mid += depths\_mid[i];

wor +=depths\_wor[i];

}

printf("%d:\ndepth for worst: %g\ndepth for middle: %g\n", M, (float)(wor/100), (float)(mid/100));

\*/

free(str);

free(keys);

free(keys\_ch);

delete\_BT(tree);

delete\_BT(treeR);

delete\_BT(WorstTree);

delete\_BT(MiddleTree);

free(worst);

free(middle);

return 0;

}

# ПРИЛОЖЕНИЕ B

**ИСХОДНЫЙ КОД API.c**

#include <string.h>

#include <time.h>

#include "API.h"

#define N 1000

#define M 2000

int main()

{

srand(time(NULL)); // new settings for random number

char\* str = malloc(sizeof(char) \* N \* 2); // string for keys of RBST

char\* keys\_ch = NULL; // for char keys

float\* keys = NULL; // for float keys

int size = 0; // number of keys

void\* key;

void\* WorstKey = NULL;

void\* MiddleKeyR = NULL;

int flag = 0;

int format; // format of input

size\_t fixtype = 0; // type of keys

float\* middle = NULL;

float\* worst = NULL;

int WRDepth;

int MRDepth;

int WDepth;

int MDepth;

int times[6];

FILE \*f;

printf("Hello! This is randomized binary search tree (RBST) explorer!\nAvailable type of elements: char, float, int\n");

printf("Type '1', if you want input data from file 'input.txt'\nType '2' - from the keyboard\nType '3' - to generate input data: ");

scanf("%d", &format);

char c = getchar(); // takes '\n'

if (format != 1 && format != 2 && format != 3) {

printf("Wrong input format!\n");

return 0;

}

else if (format == 1) { // input from file

if ((f = fopen("input.txt", "r")) == NULL) {

printf("Couldn't open input.txt\n");

return 0;

}

fgets(str, N - 1, f);

printf("You entered: %s\n", str);

fclose(f);

}

else if (format == 2) {

printf("Input elements of RBST: ");

fgets(str, N - 1, stdin);

printf("You entered: %s\n", str);

}

else if (format == 3){

printf("Input number of elements (<=10000): ");

scanf("%d", &size);

c = getchar(); // takes '\n'

f = fopen("output.txt", "a");

fprintf(f, "\n\nИсследование рандомизированного бинарного дерева (RBST).\n\n");

middle = calloc(size, sizeof(float));

worst = calloc(size, sizeof(float));

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

middle[i] = rand()%(size\*10);

worst[i] = rand()%(size\*10);

if (i != 0){

while(worst[i] <= worst[i-1]){

worst[i] = worst[i-1] + 1 + rand()%10;

}

}

}

fprintf(f, "Сгенерированные данные для худшего случая (возрастающая последовательность): ");

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

fprintf(f, "[%g] ", worst[i]);

}

fprintf(f, "\n\n");

fprintf(f, "Сгенерированные данные для случайного распределения: ");

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

fprintf(f, "[%g] ", middle[i]);

}

fprintf(f, "\n\n");

fprintf(f, "Результаты построения RBST и BST на одних исходных данных\n");

fprintf(f, "\nRBST \t\t\t\tBST\n");

}

if (isalpha(str[0])) { // if first element of keys is symbol

fixtype = (sizeof(char));

keys\_ch = malloc(N \* sizeof(\*keys\_ch));

}

else if (isdigit(str[0])) { // if first element of keys is digit

fixtype = (sizeof(float));

keys = calloc(N, sizeof(float));

}

else if (format != 3){

printf("wrong input!\n");

return 0;

}

Node\* treeR = NULL;

Node\* tree = NULL;

Node\* WorstTreeR = NULL;

Node\* MiddleTreeR = NULL;

Node\* WorstTree = NULL;

Node\* MiddleTree = NULL;

size = GetElements(keys, keys\_ch, str, fixtype); // extract elements from string

if (format == 3){

times[0] = clock();

WorstTreeR = CreateRBST(WorstTreeR, worst, keys\_ch, sizeof(float), M);

times[1] = clock();

MiddleTreeR = CreateRBST(MiddleTreeR, middle, keys\_ch, sizeof(float), M);

times[2] = clock();

printf("\nworst randomized case:\n");

showtree(WorstTreeR, sizeof(float));

printf("\n\n");

printf("\nmiddle randomized case:\n");

showtree(MiddleTreeR, sizeof(float));

printf("\n\n");

WRDepth = GetDepth(WorstTreeR);

MRDepth = GetDepth(MiddleTreeR);

delete\_BT(WorstTreeR);

delete\_BT(MiddleTreeR);

times[4] = clock();

MiddleTree = CreateBST(MiddleTree, middle, keys\_ch, sizeof(float), M);

times[5] = clock();

printf("\nmiddle case:\n");

showtree(MiddleTree, sizeof(float));

printf("\n\n");

MDepth = GetDepth(MiddleTree);

fprintf(f, "Глубина (худший): %d\t\t\tГлубина (худший): %d\nГлубина (случайный): %d\t\t\tГлубина (случайный): %d\n", WRDepth, M, MRDepth, MDepth);

fprintf(f, "Время (худший): %.4f\t\t\tВремя (худший): -\nВремя (случайный): %.4f\t\tВремя (случайный): %.4f\n", (float)(times[1] - times[0])/CLOCKS\_PER\_SEC, (float)(times[2] - times[1])/CLOCKS\_PER\_SEC, (float)(times[5] - times[4])/CLOCKS\_PER\_SEC);

fclose(f);

}

else{

times[0] = clock();

treeR = CreateRBST(treeR, keys, keys\_ch, fixtype, size);

times[1] = clock();

times[2] = clock();

tree = CreateBST(tree, keys, keys\_ch, fixtype, size);

times[3] = clock();

printf("\nRBST:\n");

showtree(treeR, fixtype);

printf("\nBST:\n");

showtree(tree, fixtype);

printf("\n\n");

printf("RBST\t\t\t\tBST\n");

printf("Depth: %d\t\t\tDepth: %d\n", GetDepth(treeR), GetDepth(tree));

printf("Time: %.4f\t\t\tTime: %.4f\n", (float)(times[1]-times[0])/CLOCKS\_PER\_SEC, (float)(times[3]-times[2])/CLOCKS\_PER\_SEC);

}

int depths\_mid[100] = {0};

int depths\_wor[100] = {0};

int mid = 0;

int wor = 0;

/\* for(int i = 0; i < 100; i++){

explore(size, &depths\_mid[i], &depths\_wor[i]);

mid += depths\_mid[i];

wor +=depths\_wor[i];

}

printf("%d:\ndepth for worst: %g\ndepth for middle: %g\n", M, (float)(wor/100), (float)(mid/100));

\*/

free(str);

free(keys);

free(keys\_ch);

delete\_BT(tree);

delete\_BT(treeR);

delete\_BT(WorstTree);

delete\_BT(MiddleTree);

free(worst);

free(middle);

return 0;

}