МИНОБРНАУКИ РОССИИ
САНКТ–ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №4

по дисциплине «Алгоритмы и структуры данных»

Тема: Деревья

Студент гр. 7381 _____ Ильясов А.В.

Преподаватель _____ Фирсов М.А.

Санкт–Петербург

2018

Задание

Вариант 2(а–г):

Для заданного бинарного дерева b типа BT с произвольным типом элементов:

а) определить максимальную глубину дерева b, т. е. число ветвей в самом длинном из путей от корня дерева до листьев;

б) вычислить длину внутреннего пути дерева b, т. е. сумму по всем узлам длин путей от корня до узла;

в) напечатать элементы из всех листьев дерева b;

г) подсчитать число узлов на заданном уровне n дерева b (корень считать узлом 1–го уровня);

Реализация дерева должна быть основана на динамической памяти.

Пояснение задания

Наиболее важным типом деревьев являются бинарные деревья. Удобно дать следующее формальное определение. Бинарное дерево – конечное множество узлов, которое либо пусто, либо состоит из корня и двух непересекающихся бинарных деревьев, называемых правым поддеревом и левым поддеревом.

Например, бинарное дерево, изображенное на рис. 1, имеет скобочное представление:

$(a\ (b\ (d\ \Lambda\ (h\ \Lambda\ \Lambda))\ (e\ \Lambda\ \Lambda))\ (c\ (f\ (i\ \Lambda\ \Lambda)\ (j\ \Lambda\ \Lambda))\ (g\ \Lambda\ (k\ (l\ \Lambda\ \Lambda)\ \Lambda))))$.

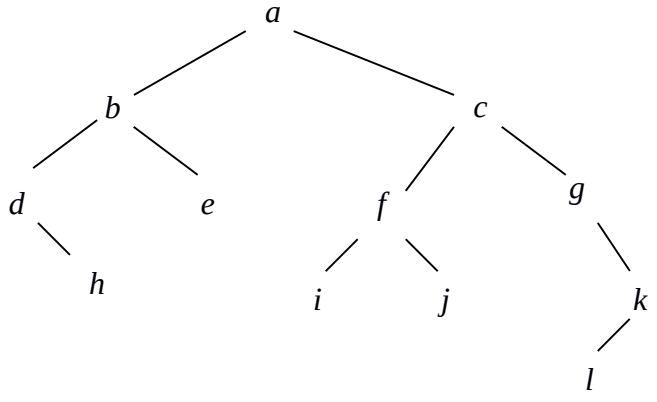

Рис. 1 – Бинарное дерево

Определим скобочное представление бинарного дерева (БД):

< БД > ::= < пусто > | < непустое БД >,

< пусто > ::= ʌ,

< непустое БД > ::= (< корень > < БД > < БД >).

Задача каждого из пунктов сводится к рекурсивному обходу дерева, заданного скобочным выражением, и выполнение определенных действий.

Описание алгоритмов

а) По определению, корень дерева имеет глубину 1, значит глубина непустого дерева не меньше 1. При переходе в левое или правое поддерево, глубина дерева увеличивается на 1, то есть равна 2. При дальнейшем переходе к поддеревьям текущих корней, глубина увеличивается на 1. И это происходит до тех пор, пока мы не натыкаемся на лист – узел дерева, не имеющий детей.

При каждом вызове метода подсчета глубины производится проверка на наличие поддеревьев, и если поддеревья есть, то рекурсивно вызывается этот же метод для поддеревьев с прибавлением единицы к промежуточному значению глубины деревьев, являющихся поддеревьями заданного. После этого выбирается наибольшее значение глубины левого или правого поддерева, которое возвращается, как результат выполнения метода, и также будет сравнен со значением, полученным в другом поддереве, и также будет сравнен, только на предыдущем шаге рекурсии.

б) Чтобы посчитать длину пути в дереве, необходимо пройти по всем поддеревьям и с каждым шагом прибавлять единицу к длине.

При каждом рекурсивном дереве создается 2 переменные, отвечающие за длину пути в левом и правом поддереве, инициализирующиеся 0. Это сделано исключительно для подробного вывода работы программы. Далее вызывается этот же метод для каждого поддерева, если оно есть. И так продолжается до тех пор, пока не встречается лист, и тогда длины путей в поддеревьях складываются и возвращаются с последующим складыванием с полученным значением в другом поддереве на предыдущем шаге рекурсии.

в) Пока текущий просматриваемый узел не лист, происходит переход в его левое или правое поддерево. Когда встречается лист, значение этого узла записывается в выходную строку, также исключительно для подробного вывода работы программы, которая потом выводится на экран.

г) Первоначально производится проверка на введенное значение, которое должно быть строго положительным и не превышать максимальную глубину дерева, посчитанную в пункте а). Если значение удовлетворяет этим условиям, то это значение передается, как аргумент в метод, в котором происходит такой же обход дерева, который описывался в предыдущих пунктах, с тем отличием, что в этом методе при каждом погружении на следующий уровень дерева уменьшается значение получаемого аргумента на единицу. При достижении нужного уровня глубины, значение аргумента равно 1, так как корень дерева лежит на первом уровне. Если на этом уровне в поддереве есть узел, то счетчик узлов увеличивается на единицу. Таким образом при обходе всего дерева, будут обнаружены все узлы на заданном уровне.

Описание функций

`void print_tabs(size_t tabs);` – функция, печатающая необходимое количество символов табуляции.

`size_t tabs` – количество выводимых символов табуляции.

Возвращаемое значение: функция ничего не возвращает.

`enum side_t {LEFT, RIGHT};` - перечисление, используемое для выбора стороны поддерева при создании бинарного дерева.

Описание класса `BinaryTree`**:**

поля:

`char value;` – значение корня.

`size_t depth;` – глубина дерева

`BinaryTree *left;` – указатель на левое поддерево.

`BinaryTree *root;` – указатель на родителя.

`BinaryTree *right;` – указатель на правое поддерево.

конструкторы:

`BinaryTree();` – стандартный конструктор. Инициализирует указатели `left`, `right`, `root` нулевыми указателями. Используется при создании поддеревьев.

`BinaryTree(std::string &string);` – конструктор, передающий выходную строку в метод чтения и создания дерева.

`~BinaryTree();` – деструктор, удаляющий узлы дерева с конца.

приватные методы:

`void read__binary_tree(std::string &string);` – основной метод чтения входной строки и создания дерева по его скобочной записи. В этом методе происходит анализ корректности строки, а также ее разделение для последующего перенаправления в методы создания дерева.

`std::string &string` – строка, содержащая скобочную запись дерева.

Возвращаемое значение: метод ничего не возвращает.

`void record_to_root(char value);` – метод, записывающий значение value в узел.

`char value` – символ, который будет записан в узел, как его значение. Должен быть либо цифрой, либо буквой.

Возвращаемое значение: метод ничего не возвращает.

`void create_tree_branches(std::string &string, side_t side);` – метод, создающий поддеревья в дереве. В нем выделяется память под дерево и вызывается `read__binary_tree` для дальнейшего создания дерева.

`std::string &string` – строка, содержащая скобочную запись дерева.

`side_t side` – переменная, определяющая сторону создаваемого поддерева.

Возвращаемое значение: метод ничего не возвращает.

`size_t get_depth(size_t lvl);` – метод, вычисляющий глубину заданного дерева с одновременным выводом процесса работы.

`size_t lvl` – количество отступов, необходимое для читаемого вывода процесса работы метода.

Возвращаемое значение: глубина дерева в формате целого беззнакового.

`size_t get_path_length(size_t lvl);` – метод, вычисляющий длину пути в заданном дереве с одновременным выводом процесса работы.

`size_t lvl` – количество отступов, необходимое для читаемого вывода процесса работы метода.

Возвращаемое значение: длина пути в дереве в формате целого беззнакового.

`void print_leaves(size_t lvl, std::string &leaves);` – метод, печатающий все листья в заданном дереве с одновременным выводом процесса работы.

`size_t lvl` – количество отступов, необходимое для читаемого вывода процесса работы метода.

`std::string &leaves` – строка, в которую помещаются все найденные листья.

Возвращаемое значение: метод ничего не возвращает.

`size_t get_nodes_number(int depth_lvl, size_t lvl);` – метод, вычисляющий количество узлов на заданном уровне в дереве с одновременным выводом процесса работы.

`int depth_lvl` – глубина, на которой нужно посчитать количество узлов. Вводится пользователем.

`size_t lvl` – количество отступов, необходимое для читаемого вывода процесса работы метода.

Возвращаемое значение: количество узлов на заданном уровне в формате целого беззнакового.

публичные метода:

`size_t get_depth();` – метод, вызывающий перегруженный метод `get_depth(size_t lvl)` с начальным значением `lvl` равным 1. Именно этот метод вызывает пользователь.

Возвращаемое значение: то же самое значение, что и перегруженный метод.

`size_t get_path_length();` – метод, вызывающий перегруженный метод `get_path_length(size_t lvl)` с начальным значением `lvl` равным 1. Именно этот метод вызывает пользователь.

Возвращаемое значение: то же самое значение, что и перегруженный метод.

`void print_leaves();` – метод, вызывающий перегруженный метод `print_leaves(size_t lvl)` с начальным значением `lvl` равным 1. Именно этот метод вызывает пользователь.

Возвращаемое значение: метод ничего не возвращает.

`size_t get_nodes_number(int depth_lvl);` – метод, вызывающий перегруженный метод `get_nodes_number(int depth_lvl, size_t lvl)` с начальным значением `lvl` равным 1. Именно этот метод вызывает пользователь.

`int depth_lvl` - глубина, на которой нужно посчитать количество узлов. Вводится пользователем.

Возвращаемое значение: то же самое значение, что и перегруженный метод.

Тестирование

Для проверки работоспособности программы был создан скрипт(см. ПРИЛОЖЕНИЕ) для автоматического ввода и вывода тестовых данных:

correct test1.txt: (a(b)(c));

correct test2.txt: (a(b#(c))(d(e)(f(g))));

correct test3.txt: (a(b(c(d)(e))(f(g)(h)))(i(j(k)(l))(m(n)(o))));

correct test4.txt: (a(b(c(d)(e)))(i#(m(n)(o))));

correct test5.txt: (ab(cd)ef(gh(ij))kl(mn(op(qr))));

correct test6.txt: (a(b(c(d(e(f(g))))))(h#(i#(j#(k#(l#(m)))))));

incorrect test1.txt: q(d)(e);

incorrect test2.txt: (a(b#(c))(d(e)(f())));

incorrect test3.txt: (a(b#(c))(d(e)(f(g))))).

Результаты тестирования сохраняются в файл testsresult.txt.

Ниже представлены результаты тестирования.

Скобочная запись дерева	Результат тестирования
(a(b)(c))	The depth of this tree is equal to 2 the length of the path in this tree is equal to 2 All the leaves of this tree: b c At level 2 is 2 nodes
(a(b#(c))(d(e)(f(g))))	The depth of this tree is equal to 4 the length of the path in this tree is equal to 6 All the leaves of this tree: c e g At level 3 is 3 nodes
(a(b(c(d)(e))(f(g)(h)))(i(j(k)(l))(m(n)(o))))	The depth of this tree is equal to 4 the length of the path in this tree is equal to 14 All the leaves of this tree: d e g h k l n o At level 3 is 4 nodes
(a(b(c(d)(e)))(i#(m(n)(o))))	The depth of this tree is equal to 4

	the length of the path in this tree is equal to 8 All the leaves of this tree: d e n o At level 4 is 4 nodes
(a(b(c(d(e(f(g))))))(h#(i#(j#(k#(l#(m)))))))	The depth of this tree is equal to 7 the length of the path in this tree is equal to 12 All the leaves of this tree: g m At level 3 is 2 nodes
q(d)(e)	--Error! Tree elements must be enclosed in parentheses--
(a(b#(c))(d(e)(f())))	--Error! Invalid character ' '--
(a(b#(c))(d(e)(f(g))))	--Error! Expected brackets or '#', but was ')'--

Разберем более подробно работу программы на примере первого теста:

Было введено (a(b)(c)).

а) подсчет глубины

Первым делом инициализируются единицей переменные для глубины левого и правого поддеревьев. У корня есть и левое, и правое поддерево, поэтому переменные увеличиваются на 1. Но корни поддеревьев являются листьями, поэтому вызовы методов завершаются и происходит сравнение полученных значений. Но 2 = 2, значит глубина заданного дерева = 2.

б) длина пути

Первым делом инициализируются нулем переменные для длины пути в левом и правом поддеревьях. У корня есть и левое, и правое поддерево, поэтому переменные увеличиваются на 1 и на значение, возвращаемое рекурсивным вызовом метода для каждого из поддеревьев. Но корни поддеревьев являются листьями, поэтому вызовы методов возвращают 0, что дает в итоге длину пути в левом и правом поддереве по 1. Значит суммарный путь имеет длину 2.

в) вывод листьев

У корня есть и левое, и правое поддерево, поэтому вызывается метод для поддеревьев, в которых корни уже являются листьями со значениями b и c. И так как для левого поддерева метод вызывается раньше, то сначала в выходную строку записывается левый лист - b, и потом уже правый - c. В итоге получается строка "b c".

г) подсчет узлов на заданном уровне

При тестировании выбор уровня производится рандомно в пределах от 1 до глубины дерева включительно. В данном случае это 2. При первом вызове метода с аргументом `depth_lvl` = 2 инициализируется нулем переменная, хранящая количество узлов на уровне 2. У корня есть и левое, и правое поддерево, поэтому переменная увеличивается на значение возвращаемое рекурсивным вызовом метода для поддеревьев, но уже с аргументом `depth_lvl` = 1. Условие равенства аргумента единице является условием выхода из рекурсии, и завершается он возвращением 1. Таким образом сначала переменная увеличивается на 1, за счет левого поддерева, а потом еще на 1, за счет правого. В итоге получаем количество узлов на 2 уровне равное 2.

Выводы

В процессе выполнения лабораторной работы были поли получены знания и навыки по ООП в C++. Были изучены бинарные деревья, повторена работа с рекурсивным функциям, bash–скриптам и автоматизации тестирования. Работа была написана на C/C++.

ПРИЛОЖЕНИЕ А

КОД MAIN.CPP

```
#include <iostream>
#include <string>
#include <algorithm>
#include <ctime>

#include "BTree.hpp"

#define TEST

int main() {
    std::string input_data;
    std::cout << "Enter the bracket tree view: ";
    std::getline(std::cin, input_data);
    input_data.erase(remove_if(input_data.begin(), input_data.end(),
isspace), input_data.end());
    BinaryTree binary_tree(input_data);

    size_t depth = binary_tree.get_depth();
    std::cout << "The depth of this tree is equal to " << depth <<
std::endl;
    std::cout << std::endl;

    size_t path_length = binary_tree.get_path_length();
    std::cout << "the length of the path in this tree is equal to " <<
path_length << std::endl;
    std::cout << std::endl;

    binary_tree.print_leaves();
    std::cout << std::endl;

    int lvl;
    #ifndef TEST
        std::cout << "Enter a number between 1 and " << depth << ": ";
        std::cin >> lvl;
    #else
        srand(time(0));
        lvl = depth - rand() % depth;
    #endif
    size_t nodes_on_lvl = binary_tree.get_nodes_number(lvl);
    std::cout << "At level " << lvl << " is " << nodes_on_lvl << " nodes"
<< std::endl;
return 0;
}
```

ПРИЛОЖЕНИЕ Б
ФАЙЛ BTREE.HPP

```
#ifndef __BTREE_HPP__
#define __BTREE_HPP__

enum side_t {LEFT, RIGHT};

class BinaryTree {
private:
    char value;                                 // Переменная для хранения
значения в корне
    size_t depth;                               // Переменная для хранения
глубины дерева
    BinaryTree *left;
    BinaryTree *root;
    BinaryTree *right;
    void read__binary_tree(std::string &string); // Методы четния
    void record_to_root(char value);             // входной строки и
    void create_tree_branches(std::string &string, side_t side);
                                                 // создания дерева

    size_t get_depth(size_t lvl);
    size_t get_path_length(size_t lvl);
    void print_leaves(size_t lvl, std::string &leaves);
    size_t get_nodes_number(int depth_lvl, size_t lvl);
public:
    BinaryTree();
    BinaryTree(std::string &string);

    size_t get_depth();                          // Метод для пункта а)
    size_t get_path_length();                    // Метод для пункта б)
    void print_leaves();                         // Метод для пункта в)
    size_t get_nodes_number(int depth_lvl);      // Метод для пункта г)

    ~BinaryTree();
};

#endif
```

ПРИЛОЖЕНИЕ В
ФАЙЛ BTREE.CPP

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <algorithm>
#include <cmath>
#include "BTree.hpp"
```

```
#define TEST

void print_tabs(size_t tabs) {
	for (size_t count = 0; count < tabs; count++)
		std::cout << ".\t";
}

BinaryTree::BinaryTree() {
	left = NULL;
	root = NULL;
	right = NULL;
}

BinaryTree::BinaryTree(std::string &string) {
	read__binary_tree(string);
}

void BinaryTree::read__binary_tree(std::string &string) {
	if (string[0] == '(' && string[string.length()-1] == ')') {
		string = string.substr(1, string.length()-2);
		record_to_root(string[0]);
		string = string.substr(1, string.length()-1);

		if (!string.length())
			return;

		if (string[0] == '(') {
			size_t index = 0;
			size_t brackets_count = 0;
			do {
				if (string[index] == '(')
					brackets_count++;
				if (string[index] == ')')
					brackets_count--;
				index++;
			} while (brackets_count > 0);
			std::string substring_left = string.substr(0, index);
			create_tree_branches(substring_left, LEFT);
			string = string.substr(index);
		}
		else if (string[0] == '#') {
			string = string.substr(1, string.length()-1);
		}
		else {
			std::cout << "--Error! Expected brackets or '#', but was
'" << string[0] << "'--" << std::endl;
			exit(0);
		}

		if (string.length() == 0)
			return;
```

```
            if (string[0] == '(') {
                  create_tree_branches(string, RIGHT);
            }
            else if (string[0] == '#') {
                  string = string.substr(1, string.length()-1);
            }
            else {
                  std::cout << "--Error! Expected brackets or '#', but was
'" << string[0] << "'--" << std::endl;
                  exit(0);
            }
      }
      else {
            std::cout << "--Error! Tree elements must be enclosed in
parentheses--" << std::endl;
            exit(0);
      }
}

void BinaryTree::record_to_root(char value) {
      if (isdigit(value) || isalpha(value)) {
            this->value = value;
            left = NULL;
            right = NULL;
      }
      else {
            std::cout << "--Error! Invalid character '" << value << "'--"
<< std::endl;
            exit(0);
      }
}

void BinaryTree::create_tree_branches(std::string &string, side_t side) {
      if (side == LEFT) {
            left = new BinaryTree();
            left->root = this;
            left->read__binary_tree(string);
      }
      else {
            right = new BinaryTree();
            right->root = this;
            right->read__binary_tree(string);
      }
}

size_t BinaryTree::get_depth() {
      #ifndef TEST
            std::cout << "Call depth calculation method" << std::endl;
      #endif
      size_t depth = get_depth(1);
```

```
	this->depth = depth;
	return depth;
}

size_t BinaryTree::get_depth(size_t lvl) {
	#ifndef TEST
		print_tabs(lvl);
		std::cout << "method call. Current root - " << value << ".
Current depth is " << lvl << std::endl;
	#endif
	size_t left_depth = 1;
	size_t right_depth = 1;
	if (left) left_depth += left->get_depth(lvl+1);
	if (right) right_depth += right->get_depth(lvl+1);
	// каждый раз выбирается максимальная глубина поддеревьев
	size_t max_depth = std::max(left_depth, right_depth);
	#ifndef TEST
		print_tabs(lvl);
		std::cout << "exit method. compare the depth of the subtrees:
left = " << left_depth-1 << ", right = " << right_depth-1
		<< ". Choose more, that is " << max_depth-1 << std::endl;
	#endif
	return max_depth;
}

size_t BinaryTree::get_path_length() {
	#ifndef TEST
		std::cout << "Call the method of calculating the length of the
path in this tree" << std::endl;
	#endif
	return get_path_length(1);
}

size_t BinaryTree::get_path_length(size_t lvl) {
	#ifndef TEST
		print_tabs(lvl);
		std::cout << "method call. Current root - " << value <<
std::endl;
	#endif
	if (!left && !right) {
		#ifndef TEST
			print_tabs(lvl);
			std::cout << "exit method. It's a leave" << std::endl;
		#endif
		return 0;
	}
	else {
		size_t left_length = 0;
		size_t right_length = 0;
		if (left) left_length += 1 + left->get_path_length(lvl+1);
		if (right) right_length += 1 + right->get_path_length(lvl+1);
```

```
            size_t length = left_length + right_length;
            #ifndef TEST
                  print_tabs(lvl);
                  std::cout << "exit method. look at the length of the
subtree paths: left = " << left_length << ", right = " << right_length
                  << ". total path is " << left_length << " + " <<
right_length << " = " << length << std::endl;
            #endif
            return length;
      }
}

void BinaryTree::print_leaves() {
      std::string leaves = "";
      #ifndef TEST
            std::cout << "Call the method of printing the leaves of this
tree" << std::endl;
      #endif
      print_leaves(1, leaves);
      std::cout << "All the leaves of this tree: " << leaves << std::endl;
}

void BinaryTree::print_leaves(size_t lvl, std::string &leaves) {
      #ifndef TEST
            print_tabs(lvl);
            std::cout << "method call. Current root - " << value <<
std::endl;
      #endif
      if (!left && !right) {
            #ifndef TEST
                  print_tabs(lvl);
                  std::cout << "exit method. It's a leave" << std::endl;
            #endif
            leaves += value;
            leaves += " ";
      }
      else {
            if (left) left->print_leaves(lvl+1, leaves);
            if (right) right->print_leaves(lvl+1, leaves);
            #ifndef TEST
                  print_tabs(lvl);
                  std::cout << "exit method. look at the leaves found: " <<
leaves << std::endl;
            #endif
      }
}

size_t BinaryTree::get_nodes_number(int depth_lvl) {
      if (depth_lvl < 1) {
            std::cout << "--Error! Expected number not less than 1--" <<
std::endl;
```

```
            exit(0);
        }
        if (depth_lvl > depth) {
        // аргумент не должен превышать максимальную глубину дерева
            std::cout << "--Error! The argument entered is greater than the
tree depth--" << std::endl;
            exit(0);
        }
        #ifndef TEST
            std::cout << "Call the method that counts the number of nodes
at a given level" << std::endl;
        #endif
        return get_nodes_number(depth_lvl, 1);
}

size_t BinaryTree::get_nodes_number(int depth_lvl, size_t lvl) {
        #ifndef TEST
            print_tabs(lvl);
            std::cout << "method call. Current lvl is " << lvl << std::endl;
        #endif
        if (depth_lvl == 1) {
        // необходимый уровень достигается, когда lvl уменьшается до 1
            #ifndef TEST
                print_tabs(lvl);
                std::cout << "exit method. reached the desired level" <<
std::endl;
            #endif
            return 1;
        }
        else {
            size_t nodes_number = 0;
            if (left) nodes_number += left->get_nodes_number(depth_lvl-1,
lvl+1);
            if (right) nodes_number += right->get_nodes_number(depth_lvl-
1, lvl+1);
            #ifndef TEST
                print_tabs(lvl);
                std::cout << "exit method. found " << nodes_number << "
nodes" << std::endl;
            #endif
            return nodes_number;
        }
}

BinaryTree::~BinaryTree() {
        if (left)
            delete left;
        if (right)
            delete right;
}
```

ПРИЛОЖЕНИЕ Г
MAKEFILE

```
all: main.o BTree.o
    g++ main.o BTree.o -o lab4
main.o: ./Source/main.cpp
    g++ -c ./Source/main.cpp
BTree.o: ./Source/BTree.hpp
    g++ -c ./Source/BTree.cpp
clean:
    rm main.o BTree.o
```

ПРИЛОЖЕНИЕ Д
ФАЙЛ RUNTESTS.SH

```
#!/bin/bash

if test ! -f "lab4" ; then
    make make clean
fi
if test -f "testsresult.txt"; then
    rm testsresult.txt
fi

touch testsresult.txt

for i in $(ls ./Tests/correct); do
    echo "running correct $i: $(cat Tests/correct/$i | more)"
        sleep 0.1s
    echo "correct "$i"" >> testsresult.txt
    echo "test data: $(cat Tests/correct/$i | more)" >> testsresult.txt
    echo "result: " >> testsresult.txt
    ./lab4 < ./Tests/correct/$i >> testsresult.txt
    echo "##################################################" >>
testsresult.txt
done
for i in $(ls ./Tests/incorrect); do
    echo "running incorrect $i: $(cat Tests/incorrect/$i | more)"
        sleep 0.1s
    echo "incorrect "$i"" >> testsresult.txt
    echo "test data: $(cat Tests/incorrect/$i | more)" >> testsresult.txt
    echo "result: " >> testsresult.txt
    ./lab4 < ./Tests/incorrect/$i >> testsresult.txt
    echo "##################################################" >>
testsresult.txt
done

sleep 0.2s
echo  "test results are saved in testsresult.txt"

rm lab4
```

ПРИЛОЖЕНИЕ Е
ФАЙЛ RUN.SH

```
#!/bin/bash

if test ! -f "lab4"; then
    make
    make clean
fi

clear
./lab4

make clean
```