

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

ОТЧЕТ
по лабораторной работе №3
по дисциплине «Алгоритмы и структуры данных»
Тема: Алгоритмы обработки бинарных деревьев

Студент гр. 9381

Колованов Р.А.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Познакомиться со структурой данных бинарного дерева, реализовать класс бинарных деревьев и методы для его обработки на языке программирования C++.

Задание.

Вариант 2д.

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

- определить максимальную глубину дерева b , т. е. число ветвей в самом длинном из путей от корня дерева до листьев;
- вычислить длину внутреннего пути дерева b , т. е. сумму по всем узлам длин путей от корня до узла.

Уточнение задания.

В данной лабораторной работе скобочные записи бинарных деревьев “ $(a(b)(c))$ ” и “ $(a(b)(c))$ ” считаются эквивалентными. Символ ‘/’ используется для обозначения пустого бинарного дерева. Бинарные деревья “ (a) ”, “ $(a/)$ ” и “ $(a/)$ ” также считаются эквивалентными.

Основные теоретические положения.

Дерево – конечное множество T , состоящее из одного или более узлов, таких, что

- а) имеется один специально обозначенный узел, называемый *корнем* данного дерева;
- б) остальные узлы (исключая корень) содержатся в $m \geq 0$ попарно не пересекающихся множествах T_1, T_2, \dots, T_m , каждое из которых, в свою очередь, является деревом. Деревья T_1, T_2, \dots, T_m называются *поддеревьями* данного дерева.

Бинарное дерево – конечное множество узлов, которое либо пусто, либо состоит из корня и двух непересекающихся бинарных деревьев, называемых правым поддеревом и левым поддеревом.

Описание алгоритма.

1) Для поиска максимальной глубины дерева был реализован метод *getMaximumDepth*. Максимальная глубина текущего дерева определяется как максимальная глубина одного из поддеревьев данного дерева (выбирается наибольшая по значению), увеличенная на 1. У листьев дерева максимальная глубина равна 0. Алгоритм заключается в следующем: Если у дерева есть правое или левое поддерева, то для них рекурсивно вызывается метод *getMaximumDepth*, который вернет максимальную глубину правого и левого поддеревьев, к которым далее прибавляется 1. Если у дерева отсутствует правое или левое поддерево, то тогда максимальная глубина отсутствующего поддерева равна 0. В конце из метода возвращается наибольшее из двух полученных значений для правого и левого поддеревьев, тем самым получая максимальную глубину текущего бинарного дерева.

2) Для поиска внутреннего пути дерева был реализован метод *getInternalPathLength*. Внутренний путь текущего дерева определяется как сумма внутренних путей его поддеревьев и глубины текущего узла дерева. Если у текущего дерева отсутствует одно из поддеревьев, то длина внутреннего пути отсутствующего поддерева равна 0. У листьев длина внутреннего пути равна глубине листа. Алгоритм заключается в следующем: Если у дерева есть правое или левое поддерева, то для них рекурсивно вызывается метод *getInternalPathLength*, который вернет длину внутреннего пути правого и левого поддеревьев (длина внутреннего пути отсутствующих поддеревьев равна 0). В конце из метода возвращается сумма длин внутренних путей его поддеревьев и глубины текущего узла дерева, тем самым получая длину внутреннего пути текущего бинарного дерева.

Описание структур и классов.

Класс *BinaryTree*.

Класс бинарного дерева. Для реализации класса используется шаблон, который определяет тип элементов дерева. Предоставляет интерфейс для создания бинарного дерева по скобочной записи и работы с бинарным деревом. В данной лабораторной работе осуществляется поиск максимальной глубины и длины внутреннего пути дерева при помощи рекурсивных методов *getMaximumDepth* и *getInternalPathLength*. Связь элементов бинарного дерева реализована при помощи указателей. Поля и методы класса приведены в таблице 2 и 3.

Таблица 2 - Поля класса *BinaryTree*

Модификатор доступа	Тип и название поля	Предназначение	Значение по умолчанию
<i>private</i>	<i>T element</i>	Хранит значение корня дерева.	-
<i>private</i>	<i>BinaryTree* right</i>	Хранит указатель на правое поддерево.	<i>nullptr</i>
<i>private</i>	<i>BinaryTree* left</i>	Хранит указатель на левое поддерево.	<i>nullptr</i>

Таблица 3 - Методы класса *BinaryTree*

Модификатор доступа	Возвращаемое значение	Название метода и принимаемые аргументы
<i>public</i>	-	<i>BinaryTree()</i>
<i>public</i>	<i>bool</i>	<i>createFromString(const char*& character)</i>
<i>public</i>	<i>bool</i>	<i>isEmpty()</i>
<i>public</i>	<i>size_t</i>	<i>getMaximumDepth(int depth = 0)</i>
<i>public</i>	<i>size_t</i>	<i>getInternalPathLength(int depth = 0)</i>
<i>public</i>	<i>std::string</i>	<i>getString()</i>
<i>public</i>	-	<i>~ BinaryTree()</i>

Метод *BinaryTree::BinaryTree*.

Конструктор. Создает пустое бинарное дерево.

Memod BinaryTree::createFromString.

Является рекурсивным методом. Принимает на вход *character* — ссылку на указатель начала строки, содержащую скобочную запись бинарного дерева. Создает бинарное дерево по заданной скобочной записи. Если бинарное дерево было успешно создано по скобочной записи, то возвращает *true*. Если скобочная запись оказалась некорректной, то возвращает *false*.

Memod BinaryTree::isEmpty.

Ничего не принимает. Если бинарное дерево является пустым (*element == '\0', left = nullptr, right == nullptr*), то возвращает *true*. Иначе возвращает *false*.

Memod BinaryTree::getMaximumDepth.

Является рекурсивным методом. Принимает на вход *depth* — уровень текущего узла дерева (считается, что корень дерева имеет глубину 0). Возвращает максимальную глубину дерева.

Memod BinaryTree::getInternalPathLength.

Является рекурсивным методом. Принимает на вход *depth* — уровень текущего узла дерева (считается, что корень дерева имеет глубину 0). Возвращает длину внутреннего пути дерева.

Memod BinaryTree::getString.

Является рекурсивным методом. Ничего не принимает. Возвращает строку класса *std::string*, в которой содержится скобочная запись бинарного дерева.

Memod BinaryTree::~~BinaryTree.

Деструктор. Является рекурсивным методом. Очищает выделенную под элементы бинарного дерева динамическую память.

Класс `Logger`.

Класс предоставляет функционал для вывода сообщений в консоль и файл из любой точки программы. Реализован с использованием паттерна *Singleton*. Поля и методы класса приведены в таблицах 4 и 5.

Таблица 4 - Поля класса *Logger*

Модификатор доступа	Тип и название поля	Предназначение	Значение по умолчанию
<i>private</i>	<i>int indentSize_</i>	Хранит размер отступа в пробелах.	<i>4</i>
<i>private</i>	<i>bool silentMode_</i>	Хранит информацию о том, включен ли тихий режим. При тихом режиме будут печататься сообщения типа <code>COMMON</code> , сообщения типа <code>DEBUG</code> будут игнорироваться.	<i>false</i>
<i>private</i>	<i>bool fileOutput_</i>	Хранит информацию о том, нужно ли выводить сообщения в файл.	<i>false</i>
<i>private</i>	<i>std::string filePath_;</i>	Содержит путь к файлу для записи сообщений.	-
<i>private</i>	<i>std::ofstream file_</i>	Поток вывода данных в файл.	-

Таблица 5 - Методы класса *Logger*

Модификатор доступа	Возвращаемое значение	Название метода и принимаемые аргументы
<i>private</i>	<i>Logger&</i>	<i>getInstance()</i>
<i>private</i>	<i>void</i>	<i>log(const std::string& message, MessageType type = COMMON, int indents = 0)</i>
<i>private</i>	<i>void</i>	<i>setSilentMode(bool value)</i>
<i>private</i>	<i>void</i>	<i>setFileOutput(const std::string& filePath)</i>

Метод `Logger::getInstance`.

Ничего не принимает. Создает статическую переменную объекта класса *Logger* (создается только один раз — при первом вызове данного метода). Возвращает ссылку на созданный объект.

Method Logger::log.

Принимает на вход три аргумента: *message* — сообщение, *type* — тип сообщения и *indents* — количество отступов. Для начала метод получает единственный объект класса *Logger* — *logger*. Далее проверяется, если включен тихий режим и тип сообщения — *DEBUG*, то происходит выход из функции. Иначе создает строку отступа, которая состоит из пробелов, количество которых равно *indentSize_ * indents*. Далее функция выводит сообщение с отступом на консоль, а также при наличии флага *fileOutput_* — в файл. Ничего не возвращает.

Method Logger::setFileOutput.

Принимает на вход *filePath* — путь к файлу для записи сообщений. Присваивает полю *filePath_* значение *filePath*, открывает поток вывода в файл и присваивает значение полю *fileOutput_* значение *true*. Ничего не возвращает.

Method Logger::setSilentMode.

Принимает на вход *value* — новое значение флага тихого режима. Устанавливает полю *silentMode_* значение *value*. Ничего не возвращает.

Выполнение работы.

Для решения поставленной задачи был написан класс *BinaryTree*, предоставляющий функционал для работы с бинарным деревом. Для вывода основной и промежуточной информации на экран и в файл был использован класс *Logger*. Для тестирования работы класса *BinaryTree* была написана функция *test*. Для вывода справки программы была написана функция *printHelp*. Генерации имени файла лога осуществляется с использованием функции *getCurrentDateTime*, которая возвращает текущую дату и время в виде строки. Помимо этого, был реализован CLI-интерфейс для удобной работы с программой.

Функция *printHelp*.

Выводит информацию о принимаемых программой аргументах на консоль.

Ничего не принимает; ничего не возвращает.

Функция *getCurrentDateTime*.

Ничего не принимает. Возвращает текущие дату и время в виде следующей строки: `<день>-<месяц>-<год>_<часы>-<минуты>-<секунды>`. Используется для генерации имени файла с логами.

Функция *test*.

Проводит тестирование программы при помощи заготовленных тестов, находящихся в файле. На вход принимает *path* — путь к файлу с тестами. Для начала открывает файл, если не удалось открыть — происходит выход из функции. Далее из файла тестов происходит считывание скобочной записи бинарного дерева и корректных значений его глубины и длины внутреннего пути, которые находятся на одной строке, разделенные символом «|», и их проверка на тестируемой функции с выводом информации о результатах. Строка имеет следующий формат: `<скобочная запись бд>|<максимальная глубина бд>|<длина внутреннего пути бд>`. Ничего не возвращает.

Функция *main*.

Для начала объявляются следующие переменные:

- *isFromFile* — хранит информацию о способе считывания входных данных;
- *isTesting* — хранит информацию о режиме тестирования;
- *isSilentMode* — хранит информацию о тихом режиме;
- *expression* — хранит строку, содержащую скобочную запись иерархического списка;
- *logger* — ссылка на единственный объект класса `Logger`.

После у логгера *logger* вызывается метод *setFileOutput* для установки файла для вывода сообщений. Далее происходит проверка аргументов, подаваемых на вход программе, и в зависимости от переданных аргументов инициализируются переменные *isFromFile*, *isTesting*, *isSilentMode* новым значениями. Если один из аргументов неверен, то происходит печать информации об этом и завершение программы. После устанавливается тихий режим при помощи метода *setSilentMode*.

Далее в зависимости от значения переменной *isTesting* происходит тестирование программы при помощи функции *test*, после чего происходит выход из программы. Если же флаг тестирования не был установлен, то в зависимости от значения переменной *isFromFile* происходит считывание либо с файла, либо с консоли.

После получения скобочной записи бинарного дерева *expression*, создается объект бинарного дерева и переменная *const char* end*, которая содержит адрес начала *C-style* строки *expression*. Далее происходит создание бинарного дерева по скобочной записи при помощи метода *createFromString*, которая принимает на вход переменную *end*. Далее происходит вызов методов *getMaximumDepth* и *getInternalPathLength*, при помощи которых осуществляется поиск максимальной глубины и длины внутреннего пути дерева. В конце происходит вывод результата и завершение работы программы.

Разработанный программный код см. в приложении А.

Результаты тестирования см. в приложении Б.

Выводы.

Была изучена структура данных бинарного, реализована рекурсивная обработка бинарных деревьев на языке программирования C++.

Разработан класс бинарного дерева *BinaryTree*, при помощи которого можно создать бинарное дерево по скобочной записи и найти максимальную

глубину и длину внутреннего пути этого дерева. При реализации методов класса *BinaryTree* использовалась рекурсия.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <ctime>
#include "BinaryTree.h"
#include "Logger.h"

void printHelp() {
    std::cout << "List of available options:\n";
    std::cout << "    -f    Input from file.\n";
    std::cout << "    -t    Conduct testing.\n";
    std::cout << "    -s    Enable silent mode.\n";
    std::cout << "    -h    Print help.\n";
    std::cout << "\n";
}

std::string getCurrentDateTime() {
    time_t timestamp; // Временная метка
    tm timeinfo;       // Структура с информацией о времени
    char buffer[80] = { '\0' };

    time(&timestamp); // Получение временной метки
    localtime_s(&timeinfo, &timestamp); // Получение информации о времени
    strftime(buffer, sizeof(buffer), "%d-%m-%y_%H-%M-%S", &timeinfo);

    return std::string(buffer);
}

void test(const std::string& path) {
    size_t testCount = 0; // Общее количество тестов
    size_t successTestCount = 0; // Колчество успешных тестов
    std::ifstream file(path);

    // Проверка на то, что файл был открыт
    if (!file.is_open()) {
        Logger::log("Cannot open file: " + path + "\n");
        return;
    }

    Logger::log("File with tests: " + path + "\n");

    while (!file.eof()) { // Пока не пройдемся по всем строкам файла
        std::string line, result1, result2;
        std::getline(file, line);

        // Поиск и проверка разделителя
        size_t separatorIndex1 = line.find('|');
        size_t separatorIndex2 = line.rfind('|');
        if (separatorIndex1 != -1 && separatorIndex2 != -1 &&
separatorIndex1 != separatorIndex2) {
            std::string expression = line.substr(0, separatorIndex1); //
Входная строка
            std::string correctResult1 = line.substr(separatorIndex1 + 1,
separatorIndex2 - separatorIndex1 - 1); // Корректный результат теста 1
            std::string correctResult2 = line.substr(separatorIndex2 + 1);
// Корректный результат теста 2
```

```

BinaryTree<char> tree;
const char* end = expression.c_str();
bool correct = tree.createFromString(end);

// Проверка на корректность скобочной записи списка
if (*end != '\0' || !correct) {
    result1 = "invalid"; // Результат теста 1
    result2 = "invalid"; // Результат теста 2
} else if (tree.isEmpty()) {
    result1 = "empty";
    result2 = "empty";
} else {
    result1 = std::to_string(tree.getMaximumDepth()); //
Результат теста 1
    result2 = std::to_string(tree.getInternalPathLength()); //
Результат теста 2
}

// Вывод результатов теста
if (result1 == correctResult1 && result2 == correctResult2) {
    successTestCount++;
    Logger::log("\n[Test #" + std::to_string(++testCount) + "
OK]\n");
} else {
    Logger::log("\n[Test #" + std::to_string(++testCount) + "
WRONG]\n");
}

Logger::log("Input binary tree: " + expression + "\n");
Logger::log("Correct result: Maximum depth = " +
correctResult1 + " and internal path length = " + correctResult2 + "\n");
Logger::log("Test result: Maximum depth = " + result1 + " and
internal path length = " + result2 + "\n");
}

}

Logger::log("\nPassed tests: " + std::to_string(successTestCount) +
"/" + std::to_string(testCount) + "\n");
}

int main(int argc, char* argv[]) {
    std::string expression;
    bool isFromFile = false;
    bool isTesting = false;
    bool isSilentMode = false;

    // Создание и настройка логгера
    Logger& logger = Logger::getInstance();
    logger.setFileOutput("Logs\\" + getCurrentDateTime() + ".txt");

    // Обработка аргументов командной строки
    if (argc > 0) {
        for (int i = 1; i < argc; i++) {
            if (strcmp(argv[i], "-f") == 0) {
                isFromFile = true;
            }
            else if (strcmp(argv[i], "-t") == 0) {
                isTesting = true;
            }
            else if (strcmp(argv[i], "-s") == 0) {
                isSilentMode = true;
            }
            else if (strcmp(argv[i], "-h") == 0) {
                printHelp();
            }
        }
    }
}

```

```

        return 0;
    }
    else {
        Logger::log("Unknown option: " + std::string(argv[i]) +
"\n");
        return 0;
    }
}

// Установка тихого режима
logger.setSilentMode(isSilentMode);

// Тестирование алгоритма при помощи набора тестов
if (isTesting) {
    test("Tests\\tests.txt");
    return 0;
}

// Ввод выражения из файла
if (isFromFile) {
    std::fstream file("input.txt");

    // Проверка на то, что файл был открыт
    if (!file.is_open()) {
        Logger::log("Cannot open file: input.txt\n");
        return 0;
    }

    std::getline(file, expression);
    Logger::log("Expression from file: " + expression + "\n");
}

// Ввод выражения с клавиатуры
else {
    std::cout << "[Enter binary tree expression] ";
    std::getline(std::cin, expression);
    Logger::log("Entered binary tree expression: " + expression +
"\n");
}

// Создание бинарного дерева
BinaryTree<char> tree;
const char* end = expression.c_str();
bool correct = tree.createFromString(end);

// Проверка на корректность скобочной записи списка
if (*end != '\0' || !correct) {
    Logger::log("Invalid binary tree expression.\n");
    return 0;
} else if (tree.isEmpty()) {
    Logger::log("Binary tree is empty.\n");
    return 0;
}

Logger::log("Created binary tree: " + tree.getString() + "\n\n");

// Получение результатов
size_t maximumDepth = tree.getMaximumDepth();
size_t internalPathLength = tree.getInternalPathLength();

// Вывод результата работы программы
Logger::log("Binary tree maximum depth: " +
std::to_string(maximumDepth) + "\n");

```

```

        Logger::log("Binary tree internal path length: " +
std::to_string(internalPathLength) + "\n");

        return 0;
    }

```

Название файла: Logger.h

```

#ifndef LOGGER_H
#define LOGGER_H

#include <fstream>

enum MessageType {
    COMMON,
    DEBUG
};

class Logger {
    int indentSize_ = 4;           // Размер отступа
    bool silentMode_ = false;      // Тихий режим
    bool fileOutput_ = false;      // Вывод сообщений в файл
    std::string filePath_;         // Путь к выходному файлу
    std::ofstream file_;           // Дескриптор выходного файла

    Logger() = default;
    Logger(const Logger&) = delete;
    Logger(Logger&) = delete;
    Logger& operator=(const Logger&) = delete;
    Logger& operator=(Logger&) = delete;
    ~Logger() = default;

public:
    static Logger& getInstance();
    static void log(const std::string& message, MessageType type = COMMON,
int indents = 0);
    void setSilentMode(bool value);
    void setFileOutput(const std::string& filePath);
};

#endif // LOGGER_H

```

Название файла: Logger.cpp

```

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

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

void Logger::setSilentMode(bool value) {
    silentMode_ = value;
}

void Logger::setFileOutput(const std::string& filePath) {
    file_.close();
    file_.open(filePath);
}

```

```

        // Проверка открытия файла
        if (!file_.is_open()) {
            filePath_ = "";
            fileOutput_ = false;
            Logger::log("Cannot open file: " + filePath + "\n");
            return;
        }

        filePath_ = filePath;
        fileOutput_ = true;
    }

    void Logger::log(const std::string& message, MessageType type, int
indents) {
        Logger& logger = Logger::getInstance();

        // Если включен тихий режим и сообщение - отладочное, то происходит
выход из функции
        if (logger.silentMode_ && type == DEBUG) {
            return;
        }

        std::string indent(logger.indentSize_ * indents, ' '); // Получение
отступа

        std::cout << indent << message; // Вывод на консоль
        if (logger.fileOutput_) {
            logger.file_ << indent << message; // Вывод в файл
        }
    }
}

```

Название файла: BinaryTree.h

```

#ifndef BINARY_TREE_H
#define BINARY_TREE_H

#include <cstddef>
#include <iostream>
#include "Logger.h"

template <typename T>
class BinaryTree {
private:
    T element; // Корень дерева
    BinaryTree* right = nullptr; // Правое поддерево
    BinaryTree* left = nullptr; // Левое поддерево

public:
    BinaryTree();
    bool createFromString(const char*& character);
    bool isEmpty();
    size_t getMaximumDepth(int depth = 0);
    size_t getInternalPathLength(int depth = 0);
    std::string getString();
    ~BinaryTree();
};

template<>
inline BinaryTree<char>::BinaryTree(): element('\0') {}

```

```

template <>
bool BinaryTree<char>::createFromString(const char*& character) {
    // Очищаем поддеревья (в случае, если до вызова метода дерево уже
использовалось)
    delete right;
    delete left;
    right = nullptr;
    left = nullptr;

    // Если скобочная запись начинается с '\', то это пустое БД
    if (*character == '/') {
        character++;
        return true;
    }

    // Если скобочная запись начинается с '(', то это непустое БД
    if (*character == '(') {
        character++;

        // Если нам встречается значение узла дерева, то записываем его в
узел
        if (*character != '(' && *character != ')' && *character != '/' &&
*character != ' ' && *character != '\0') {
            element = *character;
            character++;
        } else {
            return false;
        }

        // Если встречаем пробел, то пропускаем его
        if (*character == ' ') {
            character++;
        }
        // Если встречаем конец скобочной записи, то выходим
        else if (*character == ')') {
            character++;
            return true;
        }

        // Создаем левое поддерево
        if (*character != '/') {
            left = new BinaryTree;
            bool correct = left->createFromString(character);

            // Если не удалось корректно считать скобочную запись, то
ВЫХОДИМ
            if (!correct) {
                return false;
            }
        } else {
            character++;
        }

        // Если встречаем пробел, то пропускаем его
        if (*character == ' ') {
            character++;
        }
        // Если встречаем конец скобочной записи, то выходим
        else if (*character == ')') {
            character++;
            return true;
        }
    }
}

```



```

        // Создаем правое поддерево
        if (*character != '/') {
            right = new BinaryTree;
            bool correct = right->createFromString(character);

            // Если не удалось корректно считать скобочную запись, то
ВЫХОДИМ
            if (!correct) {
                return false;
            }

        } else {
            character++;
        }

        // Если встречаем конец скобочной записи, то выходим
        if (*character == ')') {
            character++;
            return true;
        }
    }

    return false;
}

template<>
inline bool BinaryTree<char>::isEmpty() {
    if (right == nullptr && left == nullptr && element == '\0') {
        return true;
    }
    return false;
}

template <typename T>
size_t BinaryTree<T>::getMaximumDepth(int depth) {
    Logger::log("Calling method getMaximumDepth() for binary tree " +
getString() + ":\n", DEBUG, depth);
    size_t rightDepth = 0; // Глубина правого поддерева
    size_t leftDepth = 0; // Глубина левого поддерева

    // Если у узла есть левое поддерево
    if (left != nullptr) {
        Logger::log("Right binary subtree:\n", DEBUG, depth);
        leftDepth = left->getMaximumDepth(depth + 1) + 1; // Получаем
глубину левого поддерева и к ней прибавляем 1 (для учета текущего узла)
    }

    // Если у узла есть правое поддерево
    if (right != nullptr) {
        Logger::log("Left binary subtree:\n", DEBUG, depth);
        rightDepth = right->getMaximumDepth(depth + 1) + 1; // Получаем
глубину правого поддерева и к ней прибавляем 1 (для учета текущего узла)
    }

    // Возвращаем наибольшую глубину дерева
    if (rightDepth > leftDepth) {
        Logger::log("Method getMaximumDepth() for binary tree " +
getString() + " finished: Maximum depth: " + std::to_string(rightDepth) +
"\n\n", DEBUG, depth);
        return rightDepth;
    } else {
        Logger::log("Method getMaximumDepth() for binary tree " +
getString() + " finished: Maximum depth: " + std::to_string(leftDepth) + "\n\n",
DEBUG, depth);
    }
}

```

```

        return leftDepth;
    }
}

template <typename T>
size_t BinaryTree<T>::getInternalPathLength(int depth) {
    Logger::log("Calling method getInternalPathLength() for binary tree "
+ getString() + ":\n", DEBUG, depth);
    size_t leftLength = 0; // Длина правого поддерева
    size_t righthLength = 0; // Длина левого поддерева

    // Если у узла есть левое поддерево
    if (left != nullptr) {
        Logger::log("Right binary subtree:\n", DEBUG, depth);
        leftLength = left->getInternalPathLength(depth + 1); // Получаем
внутренний путь левого поддерева
    }

    // Если у узла есть правое поддерево
    if (right != nullptr) {
        Logger::log("Left binary subtree:\n", DEBUG, depth);
        righthLength = right->getInternalPathLength(depth + 1); // Получаем
внутренний путь правого поддерева
    }

    Logger::log("Method getInternalPathLength() for binary tree " +
getString() + " finished: Internal path length: " + std::to_string(leftLength +
righthLength + depth) + "\n\n", DEBUG, depth);
    return leftLength + righthLength + depth; // Возвращаем внутренний путь
правого, левого поддерева и глубину данного узла для получения внутреннего пути
данного дерева
}

template <>
std::string BinaryTree<char>::getString() {
    std::string result = "(";

    result += std::string(1, element); // Записываем значение узла

    // Если левое поддерево не пусто, то добавляем его скобочную запись,
иначе добавляем скобочную запись пустого БД
    if (left != nullptr) {
        result += left->getString();
    } else {
        result += "/";
    }

    // Если правое поддерево не пусто, то добавляем его скобочную запись,
иначе добавляем скобочную запись пустого БД
    if (right != nullptr) {
        result += right->getString();
    } else {
        result += "/";
    }

    return result + ")";
}

template <typename T>
BinaryTree<T>::~BinaryTree() {
    delete right;
    delete left;
}

```

```
#endif // BINARY_TREE_H
```

ПРИЛОЖЕНИЕ Б

ТЕСТИРОВАНИЕ

Таблица Б.1 - Примеры тестовых случаев на некорректных данных

№ п/п	Входные данные	Выходные данные	Комментарии
1.	INVALID_DATA	invalid	OK
2.	(invalid	OK
3.)	invalid	OK
4.	()	invalid	OK
5.	(/)	invalid	OK
6.	x	invalid	OK
7.	(a)x	invalid	OK
8.	x(a)	invalid	OK
9.	(a(x)	invalid	OK
10.	(a x x)	invalid	OK

Таблица Б.2 - Примеры тестовых случаев на корректных данных

№ п/п	Входные данные	Выходные данные	Комментарии
11.	/	empty	OK
12.	(a)	0 0	OK
13.	(a / /)	0 0	OK
14.	(a (b) /)	1 1	OK
15.	(a / (b))	1 1	OK
16.	(a (b) (c))	1 2	OK
17.	(a (b) (a (b) (c)))	2 6	OK
18.	(a / (a / (a / (a))))	3 6	OK
19.	(a (a (a (a) /) /) /)	3 6	OK
20.	(f (f (f)) (f (f)))	2 6	OK
21.	(f (f (f / (c))) (f (f)))	3 9	OK
22.	(a (a (B) (a (B))) (a (a (B) (c)) (a (B) (c))))	3 25	OK
23.	(A (B (C (D) (E (F) (G))) (H (I) (H))) (K))	4 26	OK
24.	(x (x (x (x) (x)) (x (x) (x))) (x (x (x) (x)) (x (x) (x))))	3 34	OK

```

INVALID_DATA|invalid|invalid
(|invalid|invalid
)|invalid|invalid
()|invalid|invalid
(/)|invalid|invalid
x|invalid|invalid
(a)x|invalid|invalid
x(a)|invalid|invalid
(a(x)|invalid|invalid
(a x x)|invalid|invalid
/|empty|empty
(a)|0|0
(a / /)|0|0
(a (b) /)|1|1
(a / (b))|1|1
(a (b) (c))|1|2
(a (b) (a (b) (c)))|2|6
(a / (a / (a / (a))))|3|6
(a (a (a (a) /) /) /)|3|6
(f (f (f)) (f (f)))|2|6
(f (f (f / (c))) (f (f)))|3|9
(a (a (B) (a (B))) (a (a (B) (c)) (a (B) (c))))|3|25
(A (B (C (D) (E (F) (G))) (H (I) (H))) (K))|4|26
(x (x (x (x) (x)) (x (x) (x))) (x (x (x) (x)) (x (x) (x))))|3|34

```