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

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Алгоритмы и структуры данных»
Тема: Рекурсивная обработка иерархических списков

Студент гр. 9381 Колованов Р.А.

Преподаватель Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Познакомиться со структурой данных иерархического списка, реализовать рекурсивную обработку иерархических списков на языке программирования C++.

Задание.

Вариант 7.

Решить следующие задачи с использованием базовых функций рекурсивной обработки списков:

7) удалить из иерархического списка все вхождения заданного элемента (атома) *x*;

Уточнение задания.

В данной лабораторной работе скобочные записи списков *"(ab(cd)(ef)g)"* и *"(a b (c d) (e f) g)"* считаются эквивалентными, поскольку в явном разделении элементов списка нет необходимости (так как в качестве типа атома списка используется тип *char*).

Основные теоретические положения.

Иерархический список – это список, элементами которого так же могут быть иерархические списки.

Описание алгоритма.

Для удаления из иерархического списка всех вхождений заданного элемента был реализован метод *deleteElements*. Рассмотрим его реализацию подробнее. В начале метода объявляются две переменные:

- *Node* temp* – текущий элемент иерархического списка (в начале равен голове списка);
- *size_t index* – индекс текущего элемента иерархического списка (в начале равен 0).

Далее метод итеративно пробегается по элементам иерархического списка и выполняем проверку каждого из них. Если текущий элемент является подсписком, то для этого подсписка (который в свою очередь так же является иерархическим списком) рекурсивно вызывается метод *deleteElements*. Если же элемент является атомом, который нужно удалить из списка, то вызывается метод *deleteElement*, который осуществляет удаление элемента иерархического списка по индексу. В этом случае переменная *index* декрементируется, так как текущий элемент был удален из списка. Иначе, если элемент является атомом, который не требуется удалять, то метод пропускает его. В конце итерации цикла происходит переход на следующий элемент списка.

В случае, когда для подсписков будет вызван метод *deleteElements*, произойдут те же самые действия, и в свою очередь для подсписков подсписка также будет вызван метод *deleteElements*.

Описание структур и классов.

Структура Node.

Представляет собой элемент иерархического списка. Содержится внутри класса *HierarchicalList* и имеет модификатор доступа *private*. Поля структуры приведены в таблице 1. Тип значения элемента списка *ListType* в данной лабораторной работе — *char*.

Таблица 1 - Поля структуры *Node*

Модификатор доступа	Тип и название поля	Предназначение	Значение по умолчанию
public	*ListType element_*	Хранит значение элемента списка. Если элемент содержит указатель на подсписок, то значение данного поля будет равно „\0“.	-
public	*Node* next_*	Хранит указатель на следующий элемент списка.	*nullptr*
public	*Node* previous_*	Хранит указатель на предыдущий элемент списка.	*nullptr*
public	*HierarchicalList**	Содержит указатель на	*nullptr*

	*sublist_*	подсписок. Если указатель равен nullptr, то данный элемент списка является атомом и хранит значение, иначе — является подсписком.	

Класс HierarchicalList.

Класс иерархического списка. Предоставляет интерфейс для создания иерархического списка по скобочной записи и работы с иерархическим списком. В данной лабораторной работе осуществляется удаление элементов с определенным значением из иерархического списка при помощи метода *deleteElements*. Поля и методы класса приведены в таблице 2 и 3.

Таблица 2 - Поля класса *HierarchicalList*

Модификатор доступа	Тип и название поля	Предназначение	Значение по умолчанию
private	*Node* head_*	Хранит указатель на голову списка.	*nullptr*
private	*size_t size_*	Хранит размер списка.	*0*

Таблица 3 - Методы класса *HierarchicalList*

Модификатор доступа	Возвращаемое значение	Название метода и принимаемые аргументы
private	*Node**	*getNode(size_t index)*
public	-	*HierarchicalList(const char*& character)*
public	*void*	*append(const ListType& element)*
public	*HierarchicalList**	*createSublist(size_t index, const char*& character)*
public	*size_t*	*getSize()*
public	*void*	*deleteElement(size_t index)*
public	*void*	*deleteElements(const ListType& element, int indent = 0)*
public	*std::string*	*getListString()*
public	-	*~HierarchicalList()*

Метод HierarchicalList::getNode.

Принимает на вход *index* — индекс элемента в списке. Возвращает указатель на элемент списка с индексом *index*. Если список пуст (*head_ == nullptr*)

или индекс выходит за пределы размера списка (*index >= size_*), то выбрасывается исключение.

Метод HierarchicalList::HierarchicalList.

Конструктор. Является взаимно рекурсивным с методом *createSublist*. Принимает на вход *character* – ссылку на указатель начала строки, содержащую скобочную запись списка. Создает иерархический список по заданной скобочной записи.

Метод HierarchicalList::append.

Принимает на вход *element* – значение атома списка. Создает и добавляет новый атом со значением *element* в конец иерархического списка. Ничего не возвращает.

Метод HierarchicalList::createSublist.

Является взаимно рекурсивным с конструктором *HierarchicalList(const char*& character)*. Принимает на вход два аргумента: *index* – индекс элемента, *character* - ссылку на указатель начала скобочной записи подсписка в строке для передачи её в конструктор подсписка. Создает подсписок и меняет атом, находящийся по индексу *index* в списке, на созданный подсписок. Возвращает указатель на созданный подсписок. Если подсписок не был создан, то из метода вернется значение *nullptr*.

Метод HierarchicalList::getSize.

Ничего не принимает. Возвращает количество элементов в иерархическом списке (не учитывая элементы подсписков).

Метод HierarchicalList::deleteElement.

Принимает на вход *index* — индекс элемента в списке. Удаляет элемент с индексом *index* из иерархического списка. Ничего не возвращает.

Метод HierarchicalList::deleteElements.

Рекурсивный метод. Принимает на вход два аргумента: *element* – значения элементов, которые требуется удалить из иерархического списка и *indent* – глубина рекурсии. Удаляет все элементы иерархического списка и его подсписков со значением *element*. Ничего не возвращает.

Метод HierarchicalList::getListString.

Рекурсивный метод. Ничего не принимает. Возвращает строку *std::string*, в которой содержится скобочная запись иерархического списка.

Метод HierarchicalList::~HierarchicalList.

Деструктор. Является рекурсивным методом. Очищает выделенную под элементы иерархического списка 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
private	*int indentSize_*	Хранит размер отступа в пробелах.	*4*
private	*bool silentMode_*	Хранит информацию о том, включен ли тихий режим. При тихом режиме будут печататься сообщения типа COMMON, сообщения типа DEBUG будут игнорироваться.	*false*
private	*bool fileOutput_*	Хранит информацию о том, нужно ли выводить сообщения в файл.	*false*

private	*std::string filePath_;*	Содержит путь к файлу для записи сообщений.	-
private	*std::ofstream file_*	Поток вывода данных в файл.	-

Таблица 5 - Методы класса *Logger*

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

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.

Метод 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.

Метод 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.

Метод Logger::setSilentMode.

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

Класс Exception.

Объекты класса используются для выбрасывания информации об ошибки в качестве исключения. Поля и методы класса приведены в таблицах 6 и 7.

Таблица 6 - Поля класса *Exception*

Модификатор доступа	Тип и название поля	Предназначение	Значение по умолчанию
private	*const std::string error_*	Хранит сообщение об ошибке.	-

Таблица 7 - Методы класса *Logger*

Модификатор доступа	Возвращаемое значение	Название метода и принимаемые аргументы
public	-	*Exception(const std::string error)*
public	*const std::string&*	*getError()*

Метод Exception::Exception.

Конструктор объекта класса. Принимает на вход *error* — сообщение об ошибке. Поле *error_* инициализируется значением *error*.

Метод Logger::getError.

Ничего не принимает. Возвращает сообщение об ошибке.

Выполнение работы.

Для решения поставленной задачи был написан класс *HierarchicalList,* предоставляющий функционал для работы с иерархическим списком. Для вывода основной и промежуточной информации на экран и в файл был использован класс *Logger,* для выбрасывания исключений — класс *Exception*. Для

тестирования работы класса *HierarchicalList* была написана функция *test*. Для вывода справки программы была написана функция *printHelp*. Генерации имени файла лога осуществляется с использованием функции *getCurrentDataTime*, 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 getCurrentDataTime.

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 списка, элемента для удаления и корректной скобочной записи результирующего списка, которые находятся на одной строке, разделенные символом «|», и их проверка на тестируемой функции с выводом информации о результатах. Строка имеет следующий формат: *<скобочная запись списка>|<значения элементов, которые необходимо удалить>|<скобочная запись списка, который должен получится после выполнения удаления элементов>*. Ничего не возвращает.

Функция main.

Для начала объявляются следующе 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 значение переменной *isTesting* происходит тестирование программы при помощи функции *test,* после чего происходит выход из программы. Если же флаг тестирования не был установлен, то в происходит считывание входных данных. В зависимости от значения переменной *isFromFile* происходит считывание либо с файла, либо с консоли.

После получения скобочной записи списка *expression*, создается переменная *const char* end*, которая содеражит адрес начала *C-style* строки *expression*. Далее происходит создание объекта иерархического списка по скобочной записи при помощи передачи указателя *end* в конструтор. Далее пользователь вводит значение элементов, которые необходимо удалить из иерархического списка, после чего происходит вызов метода *deleteElements* для удаления элементов с выбранным значением. 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 иерархического списка, реализована рекурсивная обработка иерархических списков на языке программирования C++.

Разработан класс иерархического списка с интерфейсом, при помощи которого можно создать иерархический список и удалить из него элементы с определенным значением. Для реализации функций создания списка и удаления элементов использовалась рекурсия.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <fstream>
#include <ctime>
#include "HierarchicalList.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 t = time(nullptr);
    char buffer[80] = {'\0'};

    strftime(buffer, sizeof(buffer), "%d-%m-%y_%H-%M-%S", localtime(&t));

    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, result;
        std::getline(file, line);

        // Поиск и проверка разделителя
        size_t separatorIndex = line.find('|');
        if (separatorIndex != -1) {
            std::string expression = line.substr(0, separatorIndex); //
Входная строка
            char elementToDelete = line.substr(separatorIndex + 1,
separatorIndex + 2)[0]; // Элемент для удаления
            std::string correctResult = line.substr(separatorIndex + 3);
// Корректный результат теста

            const char* end = expression.c_str();
            HierarchicalList list(end);

            // Проверка на корректность скобочной записи списка
            if (*end != ')' || *(end + 1) != '\0' || expression.size() <=
1) {
```

```
                    result = "invalid"; // Результат теста
                } else {
                    list.deleteElements(elementToDelete);
                    result = list.getListString(); // Результат теста
                }

                // Вывод результатов теста
                if (correctResult == result) {
                    successTestCount++;
                    Logger::log("\n[Test #" + std::to_string(++testCount) + "
OK]\n");
                    Logger::log("Input list: " + expression + "\n");
                    Logger::log("Deleting element: " + std::string(1,
elementToDelete) + "\n");
                    Logger::log("Correct result: " + correctResult + "\n");
                    Logger::log("Test result: " + result + "\n\n");
                }
                else {
                    Logger::log("\n[Test #" + std::to_string(++testCount) + "
WRONG]\n");
                    Logger::log("Input list: " + expression + "\n");
                    Logger::log("Deleting element: " + std::string(1,
elementToDelete) + "\n");
                    Logger::log("Correct result: " + correctResult + "\n");
                    Logger::log("Test result: " + result + "\n\n");
                }
            }
        }

        Logger::log("Passed 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 list expression] ";
            std::getline(std::cin, expression);
            Logger::log("Entered list expression: " + expression + "\n");
        }

        // Создание иерархического списка
        const char* end = expression.c_str();
        HierarchicalList list(end);

        // Проверка на корректность скобочной записи списка
        if (*end != ')' || *(end + 1) != '\0' || expression.size() <= 1) {
            Logger::log("Invalid list expression.\n");
            return 0;
        }
        Logger::log("Created list: " + list.getListString() + "\n\n");

        // Ввод значения элементов, которые требуется удалить
        char element;
        std::cout << "[Enter element to delete] ";
        std::cin >> element;
        Logger::log("Entered element to delete: " + std::string(1, element) +
"\n\n");

        // Удаляем элементы из списка
        Logger::log("Deleting elements...\n");
        list.deleteElements(element);

        // Вывод результата работы программы
        Logger::log("Deleting completed!\n\n");
        Logger::log("List with removed elements '" + std::string(1, element) +
"': " + list.getListString() + "\n\n");

        return 0;
    }
```

Название файла: Exception.h

```
#ifndef EXCEPTION_H
```

```
#define EXCEPTION_H

#include <string>


class Exception {
    const std::string error_; // Сообщение ошибки

public:
    Exception(const std::string& error);
    const std::string& getError();
};


#endif // EXCEPTION_H
```

Название файла: Exception.cpp

```
#include "Exception.h"

Exception::Exception(const std::string& error): error_(error) {};

const std::string& Exception::getError() {
    return error_;
}
```

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()) {
        Logger::log("Cannot open file: " + filePath + "\n");
        filePath_ = "";
        fileOutput_ = false;
        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: HierarchicalList.h

```
#ifndef HIERARCHICAL_LIST_H
#define HIERARCHICAL_LIST_H

#include <cstddef>
#include <string>
```

```
typedef char ListType;


class HierarchicalList {
    struct Node;            // Элемент иерархического списка
    Node* head_ = nullptr;  // Голова списка
    size_t size_ = 0;       // Размер списка

    Node* getNode(size_t index);

public:
    explicit HierarchicalList(const char*& character);
    void append(const ListType& element);
    HierarchicalList* createSublist(size_t index, const char*& character);
    size_t getSize();
    void deleteElement(size_t index);
    void deleteElements(const ListType& element, int indent = 0);
    std::string getListString();
    ~HierarchicalList();
};


struct HierarchicalList::Node {
    ListType element_;                      // Значение элемента
    Node* next_ = nullptr;                  // Следующий элемент
    Node* previous_ = nullptr;              // Предыдущий элемент
    HierarchicalList* sublist_ = nullptr;   // Подсписок
};


#endif // HIERARCHICAL_LIST_H
```

Название файла: HierarchicalList.cpp

```
#include "HierarchicalList.h"
#include "Exception.h"
#include "Logger.h"

HierarchicalList::HierarchicalList(const char*& character) {
    // Если скобочная запись списка не начинается с '(', то выходим
    if (*character != '(') {
        return;
    }

    // Считываем элементы до тех пор, пока не встретим конец списка
    while (*(++character) != ')') {
        // Если встретился подсписок, то добавляем подсписок в конец
списка
        if (*character == '(') {
            append('\0');
            createSublist(getSize() - 1, character);
        }
        // Если встретился элемент списка, то добавляем его в конец списка
        else if (*character != ' ' && *character != '\0') {
            append(*character);
        }

        // Если достигли конца выражения, то выходим
        if (*character == '\0') {
            return;
        }
    }
```

```
    }

    HierarchicalList::Node* HierarchicalList::getNode(size_t index) {
        // Если список пуст, то выбрасываем исключение
        if (head_ == nullptr) {
            throw Exception("In function HierarchicalList::getNode(): List is
null.");
        }

        // Если индекс превышает размер, то выбрасываем исключение
        if (size_ <= index) {
            throw Exception("In function HierarchicalList::getNode(): Out of
range.");
        }

        Node* temp = head_;

        // Пробегаемся по списку до нужного индекса
        for (size_t i = 0; i < index; i++) {
            temp = temp->next_;
        }

        return temp;
    }

    void HierarchicalList::append(const ListType& element) {
        // Если список пуст - создаем голову
        if (head_ == nullptr) {
            head_ = new Node;
            head_->element_ = element;
        } else {
            Node* temp = nullptr;

            // Получаем последний элемент списка
            try {
                temp = getNode(getSize() - 1);
            } catch (Exception e) {
                Logger::log(e.getError() + "\n");
                return;
            }

            // Привязываем новый элемент к последнему элементу списка
            temp->next_ = new Node;
            temp->next_->previous_ = temp;
            temp->next_->element_ = element;
        }

        size_++;
    }

    HierarchicalList* HierarchicalList::createSublist(size_t index, const
char*& character) {
        Node* temp = nullptr;

        // Получаем элемент списка
        try {
            temp = getNode(index);
        } catch (Exception e) {
            Logger::log(e.getError() + "\n");
            return nullptr;
        }

        // Если полученный элемент - подсписок, то очищаем его
        delete temp->sublist_;
```

```
            // Создаем подсписок
            temp->sublist_ = new HierarchicalList(character);
            return temp->sublist_;
        }

        size_t HierarchicalList::getSize() {
            return size_;
        }

        void HierarchicalList::deleteElement(size_t index) {
            Node* temp = nullptr;

            // Получаем элемент списка
            try {
                temp = getNode(index);
            } catch (Exception e) {
                Logger::log(e.getError() + "\n");
                return;
            }

            // Если полученный элемент - голова списка, то устанавливаем новую
голову списка
            if (index == 0) {
                head_ = temp->next_;
            }
            // Если полученный элемент - конец списка, то удаляем связь с
предыдущим элементом
            else if (index == getSize() - 1) {
                temp->previous_->next_ = nullptr;
            }
            // Если полученный элемент - не голова и не конец списка, то
устанавливаем связи между соседними элементами
            else {
                temp->previous_->next_ = temp->next_;
                temp->next_->previous_ = temp->previous_;
            }

            // Очищаем память элемента списка
            delete temp->sublist_;
            delete temp;

            size_--;
        }

        void HierarchicalList::deleteElements(const ListType& element, int indent)
{
            Logger::log("\n", DEBUG);
            Logger::log("Calling method deleteElements() for sublist " +
getListString() + ":\n", DEBUG, indent);

            Node* temp = head_;
            size_t index = 0;

            // Проходимся по всем элементам списка
            while (temp != nullptr) {
                Node* next = temp->next_;

                // Если элемент списка - подсписок, то рекурсивно вызываем для
подсписка метод deleteElements
                if (temp->sublist_ != nullptr) {
                    temp->sublist_->deleteElements(element, indent + 1);
                }
```

```
            // Если элемент списка - элемент, который нужно удлаить, то
удаляем его из списка
            else if (temp->element_ == element) {
                Logger::log("Checking element '" + std::string(1,
temp->element_) + "': Deleting.\n", DEBUG, indent + 1);
                deleteElement(index);
                index--;
            }
            // Иначе пропускаем элемент
            else {
                Logger::log("Checking element '" + std::string(1,
temp->element_) + "': Skip.\n", DEBUG, indent + 1);
            }

            temp = next;
            index++;
        }

        Logger::log("Method deleteElements() for sublist finished. Updated
sublist: " + getListString() + ".\n\n", DEBUG, indent);
    }

    std::string HierarchicalList::getListString() {
        std::string result = "(";

        // Пробегаемся по элементам списка
        for (size_t i = 0; i < getSize(); i++) {
            Node* node = getNode(i);

            // Если элемент - не подсписок, то добавляем его в строку
            if (node->sublist_ == nullptr) {
                result += node->element_;
            }
            // Иначе получаем скобочную запись подсписка и добавляем ее к
строке
            else {
                result += node->sublist_->getListString();
            }
        }

        result += ")";
        return result;
    }

    HierarchicalList::~HierarchicalList() {
        Node* temp = head_;

        // Очищаем память всех элементов списка и его подсписков
        while (temp != nullptr) {
            Node* next = temp->next_;
            delete temp->sublist_;
            delete temp;
            temp = next;
        }
    }
```

ПРИЛОЖЕНИЕ Б

ТЕСТИРОВАНИЕ

Таблица Б.1 - Примеры тестовых случаев на некорректных данных

№ п/п	Входные данные	Выходные данные	Комментарии
1.	UNCORRECT_DATA x	invalid	
2.	(x	invalid	
3.	) x	invalid	
4.	((a) a	invalid	
5.	(a)) a	invalid	
6.	(abc)hhh h	invalid	

Таблица Б.2 - Примеры тестовых случаев на корректных данных

№ п/п	Входные данные	Выходные данные	Комментарии
7.	(a) a	()	
8.	(((((b))))) b	((((()))))	
9.	(xab) x	(ab)	
10.	(axb) x	(ab)	
11.	(abx) x	(ab)	
12.	(abc) x	(abc)	
13.	(zzz) z	()	
14.	(xb(xd)ex) x	(b(d)e)	
15.	(ax(cx)xf) x	(a(c)f)	
16.	(ab(axd)ev) x	(ab(ad)ev)	
17.	(xx(abc)xx) x	((abc))	

18.	(ab(cccc)ab) c	(ab()ab)	
19.	(xaxbx(xaxbx)x(xaxbx)xabx) x	(ab(ab)(ab)ab)	
20.	(ab(abxanc)xxxmf) x	(ab(abanc)mf)	
21.	(axxb(xabx(abxx)axxxb(axb)xab)xaxb) x	(ab(ab(ab)ab(ab)ab)ab)	

Файл с тестами: tests.txt

```
UNCORRECT_DATA|x|invalid
(|x|invalid
)|x|invalid
((a)|a|invalid
(a))|a|invalid
(abc)hhh|h|invalid
(a)|a|()
((((b))))|b|(((())))
(xab)|x|(ab)
(axb)|x|(ab)
(abx)|x|(ab)
(abc)|x|(abc)
(zzz)|z|()
(xb(xd)ex)|x|(b(d)e)
(ax(cx)xf)|x|(a(c)f)
(ab(axd)ev)|x|(ab(ad)ev)
(xx(abc)xx)|x|((abc))
(ab(cccc)ab)|c|(ab()ab)
(xaxbx(xaxbx)x(xaxbx)xabx)|x|(ab(ab)(ab)ab)
(ab(abxanc)xxxmf)|x|(ab(abanc)mf)
(axxb(xabx(abxx)axxxb(axb)xab)xaxb)|x|(ab(ab(ab)ab(ab)ab)ab)
```