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: Иерархические списки

Студент гр. 9381 _____ Прибылов Н.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Изучить принципы рекурсии, реализовать рекурсивно определённую структуру данных — иерархический список.

Задание.

Вариант 12

Проверить идентичность двух иерархических списков.

Основные теоретические положения.

Иерархический список — рекурсивно определённая структура данных. Похожа на линейный список, однако её узлами могут быть в том числе другие иерархические списки. Структура иерархического списка элементов типа *elem*:

<Hlist(elem)> :== <head(elem)>

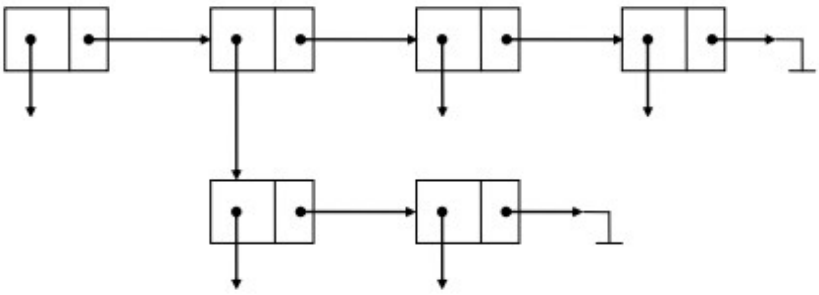
<head(elem)> :== <Node(elem)>

<Node(elem)> :== ((<Atom(elem)> | <Hlist(elem)>) <next>)

<Atom(elem)> :== <elem>

<next> :== <Node(elem)>

Графический пример списка:



Как видно, второй элемент списка сам представляет из себя иерархический список.

Описание алгоритма.

Проверка на идентичность двух иерархических списков состоит в том, чтобы установить идентичность каждой пары соответствующих узлов. Списки считаются идентичными, если длины списков одинаковы и каждая пара

соответствующих узлов идентична. Узлы считаются идентичными, если совпадает их тип (оба узла — либо атомы, либо вложенные списки) и внутреннее содержимое (либо атомы одинаковы, либо вложенные списки идентичны). Алгоритм проходит по спискам, сравнивая их узлы друг с другом. Отсюда косвенная рекурсия — для того, чтобы проверить идентичность списков, требуется проверять идентичность узлов. Если узлы сами являются списками, требуется проверить идентичность этих списков.

Описание структур данных и функций.

struct Node — узел списка.

Поля структуры:

std::variant<atom, HList> cur* — текущий элемент (либо атом, либо указатель на вложенный список).

Node next* — указатель на следующий узел.

Методы структуры:

explicit Node(std::variant<atom, HList> elem)* — конструктор, принимает элемент и создаёт на его основе узел.

bool operator==(const Node& other) const — оператор сравнения двух узлов. Принимает другой узел, возвращает результат сравнения. Проверяют, совпадает ли тип элементов, затем, проверяют их идентичность (либо сравниваются атомы, либо проверяются на идентичность вложенные списки с помощью оператора сравнения двух иерархических списков). Узлы равны, только если совпадает тип их элементов и сами элементы.

bool operator!=(const Node& other) const — инверсия предыдущего оператора.

std::string toString() const — возвращает узел в виде строки.

class HList — иерархический список.

Поля класса:

Node head* — указатель на голову списка.

Node tail* — указатели на хвост списка.

Методы класса:

Hlist() — конструктор класса, создаёт пустой список.

explicit HList(const std::string &from) — конструктор класса, принимает строку с сокращённой скобочной записью и создаёт список на основе этой строки. Если строка некорректна, создаёт пустой список.

~Hlist() — деструктор, очищает память.

bool operator==(const HList& other) const — оператор сравнения. Принимает другой список, возвращает результат сравнения. Проверяет идентичность списков, пробегая от головы до хвоста обоих и вызывая операторы сравнения пары узлов. Списки равны, только если каждая пара соответствующих узлов совпадает и концы списков достигнуты одновременно (то есть одинаковы по длине, нет «лишних» узлов на конце одного из них).

Node pushBack(std::variant<atom, HList*> elem)* — принимает элемент, вставляет его в конец списка, возвращает его же.

std::string toString() const — возвращает список в виде строки с сокращённой скобочной записью.

bool isStringCorrect(const std::string& str) — принимает строку с сокращённым скобочным представлением списка. Возвращает результат проверки строки на корректность.

void readFromString(const std::string& from, int &pos) — принимает строку и позицию, с которой нужно её читать. Добавляет элементы из неё в список.

class Logger — вспомогательный класс для логгирования промежуточных результатов.

Методы класса:

static Logger& instance() — возвращает экземпляр класса.

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

void logNodeOperatorEquals(const std::string& first, const std::string& second, bool res) — принимает две строки, соответствующие некоторым частям двух списков, и результат их сравнения для логирования.

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

~Logger()— деструктор, закрывает файл лога.

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

Разработанный программный код см. в приложении А.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные
1	(f) (f)	Первый список: (f) Второй список: (f) Узлы "f" и "f" идентичны. Списки идентичны.
2	(f(a)) (f(a))	Первый список: (f(a)) Второй список: (f(a)) Узлы "f" и "f" идентичны.

		Узлы "a" и "a" идентичны. Узлы "(a)" и "(a)" идентичны. Списки идентичны.
3	a ()	Первый список: a Второй список: () Строка "a" некорректна. Создан пустой список. Списки идентичны.
4	(a) (a))	Первый список: (a) Второй список: (a)) Строка "(a))" некорректна. Создан пустой список. Списки не идентичны.
5	(ab((cd)(e((fg))hi))) (ab((cd)(e((fg(x)))hi)))	Узлы "a" и "a" идентичны. Узлы "b" и "b" идентичны. Узлы "c" и "c" идентичны. Узлы "d" и "d" идентичны. Узлы "(cd)" и "(cd)" идентичны. Узлы "e" и "e" идентичны. Узлы "f" и "f" идентичны. Узлы "g" и "g" идентичны. Узлы "(fg)" и "(fg(x))" не идентичны. Узлы "((fg))" и "((fg(x)))" не идентичны. Узлы "(e((fg))hi)" и "(e((fg(x)))hi)" не идентичны. Узлы "((cd)(e((fg))hi))" и "((cd)(e((fg(x)))hi))" не идентичны. Списки не идентичны.
6	(ab((cd)(e((fg x))hi))) (ab((cd)(e((fg(x)))hi)))	Первый список: (ab((cd)(e((fgx))hi))) Второй список: (ab((cd)(e((fg(x)))hi))) Узлы "a" и "a" идентичны. Узлы "b" и "b" идентичны. Узлы "c" и "c" идентичны. Узлы "d" и "d" идентичны. Узлы "(cd)" и "(cd)" идентичны. Узлы "e" и "e" идентичны.

		Узлы "f" и "f" идентичны. Узлы "g" и "g" идентичны. Узлы "x" и "(x)" не идентичны. Узлы "(fgx)" и "(fg(x))" не идентичны. Узлы "((fgx))" и "((fg(x)))" не идентичны. Узлы "(e((fgx))hi)" и "(e((fg(x)))hi)" не идентичны. Узлы "((cd)(e((fgx))hi))" и "((cd)(e((fg(x)))hi))" не идентичны. Списки не идентичны.
7	(ab((cd)(e((fg( )))hi))) (ab((cd)(e((fg(x)))hi)))	Первый список: (ab((cd)(e((fg()))hi))) Второй список: (ab((cd)(e((fg(x)))hi))) Узлы "a" и "a" идентичны. Узлы "b" и "b" идентичны. Узлы "c" и "c" идентичны. Узлы "d" и "d" идентичны. Узлы "(cd)" и "(cd)" идентичны. Узлы "e" и "e" идентичны. Узлы "f" и "f" идентичны. Узлы "g" и "g" идентичны. Узлы "()" и "(x)" не идентичны. Узлы "(fg())" и "(fg(x))" не идентичны. Узлы "((fg()))" и "((fg(x)))" не идентичны. Узлы "(e((fg()))hi)" и "(e((fg(x)))hi)" не идентичны. Узлы "((cd)(e((fg()))hi))" и "((cd)(e((fg(x)))hi))" не идентичны. Списки не идентичны.
8	(ab((cd)(e((fg(x)))hi))) (ab((cd)(e((fg (x)))hi)))	Первый список: (ab((cd)(e((fg(x)))hi))) Второй список: (ab((cd)(e((fg(x)))hi))) Узлы "a" и "a" идентичны. Узлы "b" и "b" идентичны. Узлы "c" и "c" идентичны. Узлы "d" и "d" идентичны. Узлы "(cd)" и "(cd)" идентичны. Узлы "e" и "e" идентичны.

		Узлы "f" и "f" идентичны. Узлы "g" и "g" идентичны. Узлы "x" и "x" идентичны. Узлы "(x)" и "(x)" идентичны. Узлы "(fg(x))" и "(fg(x))" идентичны. Узлы "((fg(x)))" и "((fg(x)))" идентичны. Узлы "h" и "h" идентичны. Узлы "i" и "i" идентичны. Узлы "(e((fg(x)))hi)" и "(e((fg(x)))hi)" идентичны. Узлы "((cd)(e((fg(x)))hi))" и "((cd)(e((fg(x)))hi))" идентичны. Списки идентичны.

Выводы.

Были изучены принципы рекурсии, был реализован иерархический список и алгоритм проверки двух таких списков на идентичность.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include "HList.h"
#include "Logger.h"

char kDefaultStopString[] = "STOP";
char kDefaultFileName[] = "input.txt";

void printTask(); // печатает задание
void printMenu(); // печатает меню
void menu(); // вызывает меню
void consoleInput(); // организует ввод с консоли
void fileInput(); // организует ввод с файла
void performTask(std::istream& infile); // принимает поток чтения, начинает работу программы

void printTask() {
            Logger::instance().log("Проверка идентичности двух иерархических списков.\n");
}

void printMenu() {
    std::cout << "1. Ввести данные с клавиатуры.\n"
                 "2. Ввести данные с файла.\n"
                 "0. Выход из программы.\n";
}

void menu() {
    printTask();
    printMenu();
    char c = '1';
    do {
        std::cin >> c;
        switch(c) {
            case '1':
                std::cin.ignore(256, '\n');
                consoleInput();
                break;
            case '2':
                std::cin.ignore(256, '\n');
                fileInput();
                break;
            case '0':
                std::cout << "Выход из программы.\n";
                break;
            default:
                std::cout << "Неверное значение.\n";
                break;
        }
        if (c != '0') printMenu();
```

```
        } while (c != '0');
    }

    void consoleInput() {
        std::cout << "Вводите данные:\n"
                             "Чтобы вернуться в меню, введите \"" <<
kDefaultStopString << "\"\n";
        performTask(std::cin);
    }

    void fileInput() {
        std::string inputFileName;
        std::ifstream infile;
        std::cout << "Введите название файла:\n"
                          "По умолчанию данные читаются из файла \"" <<
kDefaultFileName << "\".\n";
        getline(std::cin, inputFileName);

        if (inputFileName.empty()) {
            inputFileName = kDefaultFileName;
        }

        infile.open(inputFileName);
        if (!infile) {
                 std::cout << "Файла \"" << inputFileName << "\" не
существует.\n";
        } else {
              std::cout << "Чтение данных прекратится на строке \"" <<
kDefaultStopString << "\".\n";
            performTask(infile);
        }

        if (infile.is_open()) {
            infile.close();
        }
    }

    void performTask(std::istream& infile)
    {
        std::string first, second;

        while (!infile.eof()) {
            getline(infile, first);
            // удаляются все пробелы
              first.erase(std::remove(first.begin(), first.end(), ' '),
first.end());
            if (!first.length()) continue;

            Logger::instance().log("Первый список: " + first);
            if (first == kDefaultStopString) {
                       Logger::instance().log("Встретилась терминальная
строка.\n");
                return;
            }

            getline(infile, second);
               second.erase(std::remove(second.begin(), second.end(), '
'), second.end());
```

```
            if (!second.length()) continue;

            Logger::instance().log("Второй список: " + second);
            if (second == kDefaultStopString) {
                        Logger::instance().log("Встретилась  терминальная
строка.\n");
                return;
            }

            hlist::HList x(first), y(second);
            x == y ?
                Logger::instance().log("Списки идентичны.\n")
            :
                Logger::instance().log("Списки не идентичны.\n");
        }
    }

    int main() {
        menu();
        return 0;
    }
```

Название файла: HList.h

```
    #ifndef ALG_LAB2_HLIST_H
    #define ALG_LAB2_HLIST_H

    #include <variant>
    #include <string>
    #include "Logger.h"

    namespace hlist
    {
        struct HList;
        using atom = char;
         using element = std::variant<atom, HList*>; // элемент узла -
либо атом, либо указатель на внутренний список

        struct Node
        {
            element cur;
            Node* next = nullptr;

            // создаёт узел с элементом
            explicit Node(element elem);

            // проверяет идентичность узлов
            bool operator==(const Node& other) const;
            bool operator!=(const Node& other) const;

            // возвращает узел в виде строки (для логгирования)
            std::string toString() const;
        };

        class HList
        {
        public:
```

```
            // создаёт пустой список
            HList();

             // создаёт список на основе строки с сокращённой скобочной
записью;
                // если строка содержит некорректный список, создаётся
пустой список
            explicit HList(const std::string &from);

            // очищает память
            ~HList();

            // проверяет идентичность списков
            bool operator==(const HList& other) const;

            // вставляет новый элемент в конец списка
            Node* pushBack(element elem);

            // возвращает сокращённое скобочное представление списка
            std::string toString() const;

        private:
            Node* head;
            Node* tail;

            // проверяет строку на корректность
            bool isStringCorrect(const std::string& str);

              // анализирует строку с сокращённой скобочной записью и
добавляет в список элементы
            void readFromString(const std::string& from, int &pos);
        };

    }
    #endif //ALG_LAB2_HLIST_H
```

Название файла: HList.cpp

```
    #include "HList.h"

    namespace hlist
    {
        Node::Node(element elem) : cur(elem) {}

        bool Node::operator==(const Node& other) const {
             // если типы текущих узлов различны - узлы однозначно не
идентичны
            if (cur.index() != other.cur.index()) {
                    Logger::instance().logNodeOperatorEquals(toString(),
other.toString(), false);
                return false;
            }
            if (std::holds_alternative<atom>(cur)) {
                // если текущие узлы - атомы, сравниваются атомы
                                bool   res   =   std::get<atom>(cur)   ==
std::get<atom>(other.cur);
```

```
                    Logger::instance().logNodeOperatorEquals(toString(),
other.toString(), res);
                return res;
            } else {
                    // если же текущие узлы - указатели на внутренний
список,
                  // сравниваются списки (вызывается оператор сравнения
класса HList) - косвенная рекурсия
                            bool  res  =  *(std::get<HList  *>(cur))  ==
*(std::get<HList *>(other.cur));
                    Logger::instance().logNodeOperatorEquals(toString(),
other.toString(), res);
                return res;
            }
        }

        bool Node::operator!=(const Node& other) const {
            return !(*this == other);
        }

        std::string Node::toString() const {
            if (std::holds_alternative<atom>(cur)) {
                return std::string(1, std::get<atom>(cur));
            } else {
                return std::get<HList *>(cur)->toString();
            }
        }


        HList::HList() : head(nullptr), tail(nullptr) {}

        HList::HList(const std::string &from) : HList() {
            if (!isStringCorrect(from)) {
                    Logger::instance().log("Строка \"" + from + "\"
некорректна. Создан пустой список.");
                return;
            }
            int pos = 0;
            readFromString(from, pos);
        }

        HList::~HList() {
            Node* tmp = head;
            while (tmp) {
                head = tmp->next;
                if (std::holds_alternative<HList *>(tmp->cur)) {
                       // если текущий узел - внутренний список, память
под него тоже очищается
                    delete *std::get_if<HList *>(&(tmp->cur));
                }
                delete tmp;
                tmp = head;
            }
        }

        bool HList::operator==(const HList& other) const {
            Node* i = head; Node* j = other.head;
```

```
            // продвигается по спискам, пока не достигнут конец хотя бы одного из них,
                // либо пока соответствующие узлы равны (вызывается оператор сравнения класса Node) - косвенная рекурсия
            while (i && j && *i == *j) {
                i = i->next, j = j->next;
            }
            // последняя проверка на идентичность - концы обоих списков достигнуты
            return !i && !j;
        }

        bool HList::isStringCorrect(const std::string &str) {
            int pos = 0, bracketPairsCount = 0;
            while (str[pos] == ' ') pos++;
            if (str[pos] != '(') return false;

            bracketPairsCount++;
            while (++pos != str.length() && bracketPairsCount != 0) {
                if (str[pos] == ' ') continue;
                if (str[pos] == '(') bracketPairsCount++;
                else if (str[pos] == ')') bracketPairsCount--;
            }
            return pos == str.length() && bracketPairsCount == 0;
        }

        void HList::readFromString(const std::string &from, int &pos) {
            while (from[++pos] != ')' && pos < from.length()) {
                if (from[pos] == ' ') continue;
                if (from[pos] != '(') {
                    pushBack(from[pos]);
                } else {
                    HList* tmp = new HList;
                    tmp->readFromString(from, pos);
                    pushBack(tmp);
                }
            }
        }

        Node* HList::pushBack(element elem) {
            Node* node = new Node(elem);
            if (!head) {
                head = node;
                tail = node;
            } else {
                tail->next = node;
                tail = tail->next;
            }
            return node;
        }

        std::string HList::toString() const {
            std::string str;
            str += "(";
            if (head) {
                for (auto i = head; i != nullptr; i = i->next) {
                    if (std::holds_alternative<atom>(i->cur)) {
                        str += std::get<atom>(i->cur);
```

```
                } else {
                    str += std::get<HList *>(i->cur)->toString();
                }
            }
        }
        str += ")";
        return str;
    }

}
```

Название файла: Logger.h

```
#ifndef ALG_LAB2_LOGGER_H
#define ALG_LAB2_LOGGER_H

#include <iostream>
#include <fstream>
#include <string>
#include <ctime>

class Logger {
public:
    static Logger& instance();
     void log(const std::string& str, bool toConsole = true, bool
toFile = true);
       void logNodeOperatorEquals(const std::string& first, const
std::string& second, bool res);
private:
    Logger();
    ~Logger();
    Logger(const Logger&) = delete;
    Logger(Logger&&) = delete;
    Logger& operator=(const Logger&) = delete;
    Logger& operator=(Logger&&) = delete;

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


#endif //ALG_LAB2_LOGGER_H
```

Название файла: Logger.cpp

```
#include "Logger.h"

Logger::Logger() {
    std::time_t t = std::time(nullptr);
    std::tm* now = std::localtime(&t);
    char logFileName[32];
    strftime(logFileName, 32, "log_%F_%T.txt", now);
    stream.open(logFileName);
}

Logger::~Logger() {
```

```
        stream.close();
    }

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

    void Logger::log(const std::string& str, bool toConsole, bool
toFile) {
        if (toConsole) std::cout << str << '\n';
        if (toFile) stream << str << '\n';
    }

    void Logger::logNodeOperatorEquals(const std::string &first, const
std::string &second, bool res) {
          std::cout << "Узлы \"" << first << "\" и \"" << second <<
(res ? "\" идентичны.\n" : "\" не идентичны.\n");
         stream << "Узлы \"" << first << "\" и \"" << second << (res ?
"\" идентичны.\n" : "\" не идентичны.\n");
    }
```