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

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Алгоритмы и структуры данных»

Тема: Иерархические списки

Студентка гр. 9304 _____ Селезнёва А.В.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург

2020

Цель работы.

Ознакомиться с понятием иерархического списка. Реализовать иерархический список для решения поставленной задачи на языке программирования C++.

Задание.

Вариант – 24.

Пусть алгебраическое выражение представлено иерархическим списком. В выражение входят константы и переменные, которые являются атомами списка. Операции представляются в префиксной форме ((<*операция*> <*аргументы*>)).

Доступные операции: +, -, *, *power(,)*.

На входе дополнительно задаётся список значений переменных

$$\big((x_1\ c_1)(x_2\ c_2) \ldots\ (x_k\ c_k)\big),$$

где x_i – переменная, а c_i – её значение (константа).

Требуется выполнить вычисление введенного выражения.

Выполнение работы.

На вход программа получает две строки, каждая из которых содержит скобочную запись иерархического списка. Первая строка является алгебраическим выражением, а вторая списком значений переменных. Далее на основе этих строк создаются два иерархических списка. Пример хранения введенных данных «(* 6(- c 4))» и «((c 3))» в иерархических списках представлен на рисунке 1 и на рисунке 2 соответственно:

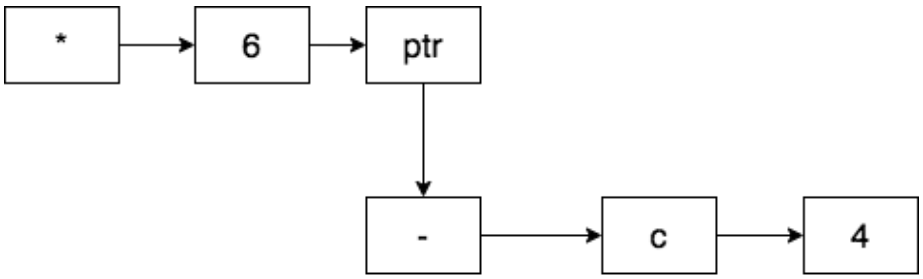

Рисунок 1 – Иллюстрация хранения алгебраического выражения в иерархическом списке

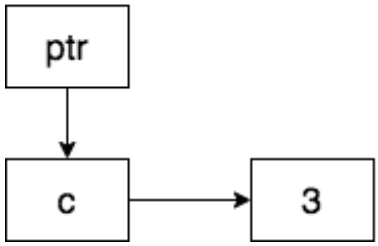

Рисунок 2 – Иллюстрация списка значений элементов в иерархическом списке

Далее вызывается метод *call_cal()* для вычисления алгебраического выражения. Результат работы этого метода выводится на экран.

Класс *Node*:

Класс содержит поле *Next* – умный указатель на следующий элемент списка. Поле *elem* – *std::varian*, который может содержать строку или умный указатель на иерархический список.

Класс *List_Hier*:

Класс содержит поле *Head* – умный указатель на первый элемент иерархического списка.

Конструктор класса принимает строку, проверяет ее на корректность и в случае ее корректности создает иерархический список, иначе выбрасывает исключение.

Метод *call_cal()* на вход получает список элементов, отправляет их в метод *calculating()* вместе с указателем на первый элемент списка и возвращает результат метода *calculating()*.

Метод *calculating()* рекурсивно вычисляет алгебраическое выражение, возвращает число – результат вычислений. Данный метод выбрасывает исключение в случае ошибки.

Метод *Search_for_an_element()* рекурсивно обходит список и находит в нем строку, равную строке, полученную в вызове метода, и возвращает число, которое соответствует числу данного элемента. Также метод выбрасывает исключение в зависимости от ошибки.

Метод *App_hier()* добавляет в конец иерархического списка полученный *Node*.

Разработанный программный код находится в приложении А.

Тестирование.

Тестирование осуществляется с помощью bash-скрипта ./script. Скрипт запускает программу и в качестве входных аргументов подает строки, прописанные в текстовых файлах, расположенных в папке ./Tests.

Результаты тестирования представлены в приложении Б.

Выводы.

В ходе выполнения лабораторной работы был реализован иерархический список на языке программирования C++.

Разработана программа, создающая иерархический список и вычисляющая алгебраическое выражение с его помощью. Использование иерархического списка при решении поставленной задачи не оправдано с точки зрения экономии памяти, так как используется рекурсивная обработка самого иерархического списка.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Lb2.cpp

```
#include "List_Hier.h"
#include <iostream>

int main() {
    std::string s_elem;
    std::string s_cal;
    std::getline(std::cin, s_cal);
    std::getline(std::cin, s_elem);
    try {
        List_Hier list_cal(s_cal);
        List_Hier list_elem(s_elem);
        std::cout << list_cal.call_cal(list_elem);
    }
    catch (const char* ex) {
        std::cout << ex;
    }
    return 0;
}
```

Название файла: List_Hier.h

```
#ifndef LIST_HEIR
#define LIST_HEIR

#include <variant>
#include <string>
#include <cmath>
#include <cctype>
#include <cstdlib>
#include <memory>

class List_Hier;

class Node {
    friend class List_Hier;
    std::variant<std::string, std::shared_ptr<List_Hier>>
elem;
    std::shared_ptr<Node> Next = nullptr;
};

class List_Hier
```

```
{
public:
    std::shared_ptr <Node> Head = nullptr;
    List_Hier(const std::string& str);
    int call_cal( List_Hier& List_elements);
    void App_hier(const Node& new_Node);
    int calculating(std::shared_ptr <Node> ptrNode,
List_Hier& List_elements);
    int Search_for_an_element(std::shared_ptr <Node> ptr,
const std::string str);
};
#endif
```

Название файла: List_Hier.cpp

#include "List_Hier.h"

```
size_t search_the_bracket(const std::string str) {
    size_t i = 0;
    int bracket = 0;
    do {
        if (str.at(i) == '(') {
            ++bracket;
        }
        else if (str.at(i) == ')') {
            --bracket;
        }
        ++i;
    } while (bracket != 0 && i < str.size());
    if (bracket != 0) {
        return 0;
    }
    return i;

}

List_Hier::List_Hier(const std::string& str)
{
    bool is_first_bracket = true;
    size_t i = 0;
    while (i < str.size()) {
        if (str.at(i) == '(') {
            if (search_the_bracket(str) == 0 &&
search_the_bracket(str) != str.size()) {
                throw ("input not true\n");
            }
            if (is_first_bracket) {
                is_first_bracket = false;
                ++i;
```

```
                }
                else {
                    std::shared_ptr <List_Hier> new_elem(new
List_Hier(str.substr(i, str.size())));
                    Node new_Node;
                    new_Node.elem = new_elem;
                    this->App_hier(new_Node);
                    i = str.size();
                }
            }
            else if (isdigit(str.at(i)) || isalpha(str.at(i)) ||
str.at(i) == '+' || str.at(i) == '-' || str.at(i) == '*') {
                if (is_first_bracket) {
                    throw ("input not true\n");
                }
                else {
                    if (str.find("power(", i) == i) {
                        if (str.find(',', i) == -1) {
                            throw ("input not true\n");
                        }
                        else {
                            const std::string new_elem =
str.substr(i, 5);
                            Node new_Node;
                            new_Node.elem =
(std::string)new_elem;
                            this->App_hier(new_Node);
                            i = i + 6;
                            size_t comma = str.find(',', i);
                            std::shared_ptr <List_Hier>
new_elem_1(new List_Hier(str.substr(i, comma - i)));
                            Node new_Node_1;
                            new_Node_1.elem = new_elem_1;
                            this->App_hier(new_Node_1);
                            size_t end_two_arg =
search_the_bracket(str.substr(comma + 1, str.size()));
                            std::shared_ptr <List_Hier>
new_elem_2(new List_Hier(str.substr(comma + 1, end_two_arg - comma
- 2)));
                            Node new_Node_2;
                            new_Node_2.elem = new_elem_2;
                            this->App_hier(new_Node_2);
                            i = end_two_arg+1;
                        }
                    }
                    else {
                        size_t i_1 = i;
                        i =
str.find_first_not_of("1234567890qwertyuiopasdfghjklzxcvbnmQWERTYU
IOPASDFGHJKLZXCVBNM-+*", i + 1);
                        const std::string new_elem =
str.substr(i_1, i - i_1);
                        Node new_Node;
```

```
                        new_Node.elem = (std::string)new_elem;
                        this->App_hier(new_Node);
                    }
                }
            }
            else if (str.at(i) == ' ' || str.at(i) == ')' ||
str.at(i) == ',') {
                ++i;
            }
            else {
                throw ("input not true\n");
            }
        }
    }

    void List_Hier::App_hier(const Node& new_Node) {
        std::shared_ptr <Node> node =
std::make_shared<Node>(new_Node);
        std::shared_ptr <Node> ptr = this->Head;
        if (this->Head != nullptr) {
            while (ptr->Next != nullptr) {
                ptr = ptr->Next;
            }
            ptr->Next = node;
        }
        else {
            this->Head = node;
        }
    }
    bool from_string_to_int(const std::string str) {
        size_t i = 0;
        if (str.at(0) == '-' || str.at(0) == '+') {
            ++i;
        }
        while (i < str.size()) {
            if (isdigit(str.at(i))) {
                ++i;
            }
            else {
                return false;
            }
        }
        return true;
    }

    int List_Hier::call_cal( List_Hier& List_elements) {
        return calculating(this->Head, List_elements);
    }

    int List_Hier::calculating(std::shared_ptr <Node> ptr_Node,
List_Hier& List_elements) {
        if (ptr_Node == nullptr) {
            throw ("error calculating\n");
```

```
        }
        else if (std::holds_alternative<std::string>(ptr_Node->elem)) {
            std::string str = std::get<std::string>(ptr_Node->elem);
            if (str == "+") {
                if (ptr_Node->Next == nullptr && ptr_Node->Next->Next == nullptr) {
                    throw ("error calculating\n");
                }
                int first = calculating(ptr_Node->Next, List_elements);
                int second = calculating(ptr_Node->Next->Next, List_elements);
                return first + second;
            }
            else if (str == "-") {
                if (ptr_Node->Next == nullptr && ptr_Node->Next->Next == nullptr) {
                    throw ("error calculating\n");
                }
                int first = calculating(ptr_Node->Next, List_elements);
                int second = calculating(ptr_Node->Next->Next, List_elements);
                return first - second;
            }
            else if (str == "*") {
                if (ptr_Node->Next == nullptr && ptr_Node->Next->Next == nullptr) {
                    throw ("error calculating\n");
                }
                int first = calculating(ptr_Node->Next, List_elements);
                int second = calculating(ptr_Node->Next->Next, List_elements);
                return first * second;
            }
            else if (str == "power") {
                if (ptr_Node->Next == nullptr && ptr_Node->Next->Next == nullptr) {
                    throw ("error calculating\n");
                }
                int first = calculating(ptr_Node->Next, List_elements);
                int second = calculating(ptr_Node->Next->Next, List_elements);
                return (int)pow(first, second);
            }
            else {
                if (from_string_to_int(str)) {
                    if (str.at(0) == '-') {
```

```
                        std::string str_1 = str.substr(1,
str.size() - 1);
                        return -atoi(str_1.c_str());
                    }
                    else if (str.at(0) == '+') {
                        std::string str_1 = str.substr(1,
str.size() - 1);
                        return atoi(str_1.c_str());
                    }
                    else {
                        return atoi(str.c_str());
                    }
                }
                else {
                    return
Search_for_an_element(List_elements.Head, str);
                }
            }
        }
        else if
(std::holds_alternative<std::shared_ptr<List_Hier>>(ptr_Node-
>elem)) {
            return calculating(std::get<
std::shared_ptr<List_Hier>>(ptr_Node->elem)->Head, List_elements);
        }
        throw ("error calculating\n");
    }

    int List_Hier::Search_for_an_element(std::shared_ptr <Node>
ptr, const std::string str) {
        if (ptr == nullptr) {
            throw ("list of elements is empty\n");
        }
        while (ptr != nullptr) {
            if (std::holds_alternative<std::string>(ptr->elem)) {
                std::string str_node = std::get<std::string>(ptr-
>elem);
                if (str_node == str) {
                    std::string str_elem =
std::get<std::string>(ptr->Next->elem);
                    if (from_string_to_int(str_elem)) {
                        if (str_elem.at(0) == '-') {
                            std::string str_1 =
str_elem.substr(1, str.size() - 1);
                            return -atoi(str_1.c_str());
                        }
                        else {
                            return atoi(str_elem.c_str());
                        }
                    }
                    else {
                        throw ("element is not an integer\n");
                    }
```

```
                }
            }
            else if
(std::holds_alternative<std::shared_ptr<List_Hier>>(ptr->elem)) {
                return Search_for_an_element(std::get<
std::shared_ptr<List_Hier>>(ptr->elem)->Head, str);
            }
            ptr = ptr->Next;
        }
        throw ("element is not found\n");
    }
```

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ

Таблица Б – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Результат проверки
1.	(- a(+ c(v))) ((a 3)(c 10)(v 11))	-18	correct
2.	(* 6(- c 4)) ((c 3))	-6	correct
3.	(power((+ 7(m)),(* l(4)))) ((m 3)(l 2))	100 000 000	correct
4.	(+ 6(a)) ((c 4))	element is not found	correct
5.	(- *(d)) ((d 4))	error calculating	correct
6.	(- a(+ 3(-4))) ((a 12))	13	correct
7.	(- 4(power((* b(2)),(+7(c))))) ((c 2)(b 4))	-134 217 724	correct
8.	(* a(+ c(b))) ((a 5)(b 2)(c 3))	25	correct