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. 9303 _____ Максимов Е.А.

Преподаватель _____ Филатов Ар. Ю.

Санкт-Петербург

2020

Цель работы.

Познакомиться с нелинейными (иерархическими) списками и способами их создания и рекурсивной обработки. Решить поставленную задачу для обработки иерархических списков с использованием рекурсивных функций.

Задание.

Вариант №12 лабораторной работы.

Проверить идентичность двух иерархических списков.

Основные теоретические положения.

В практических приложениях возникает необходимость работы с более сложными, чем линейные списки, нелинейными конструкциями. Рассмотрим одну из них, называемую иерархическим списком элементов базового типа *El* или *S*-выражением.

Определим соответствующий тип данных *S_expr* (*El*) рекурсивно, используя определение линейного списка (типа *L_list*):

< S_expr (El) > ::= < Atomic (El) > | < L_list (S_expr (El)) >,

< Atomic (E) > ::= < El >.

< L_list(El) > ::= < Null_list > | < Non_null_list(El) >

< Null_lis t> ::= Nil

< Non_null_list(El) > ::= < Pair(El) >

< Pair(El) > ::= (< Head_l(El) > . < Tail_l(El) >)

< Head_l(El) > ::= < El >

< Tail_l(El) > ::= < L_list(El) >

Выполнение работы.

В программе реализованы следующие структуры:

1. Структура *Elem* представляет собой элемент иерархического списка. Структура имеет 4 поля:

a) поле *bool dataTag* представляет собой булево значение, которое показывает, хранится ли в данном элементе иерархического списка значение;

b) поле *char data* представляет собой данные в элемента иерархического списка (символ);

c) поле *Elem* tail* представляет собой указатель на следующий элемент, принадлежащий одному иерархическому подсписку;

d) поле *Elem* child* представляет собой указатель на первый элемент иерархического подсписка, который относится к данному элементу иерархического списка.

В программе реализованы следующие функции:

1. Функция *void lineToLists(string& line, vector<Elem*>& pointerCollector)* принимает на вход переменные *line* типа *string&* – ссылку на строку, содержащую два иерархических списка, и *pointerCollector* типа *vector<Elem*>* – ссылка на вектор элементов типа *Elem**. Функция обрабатывает исходную строку, разделяет её на две подстроки при помощи функции *splitString*, создаёт иерархические списки при помощи функции *getHList*, проверяет их на корректность и сравнивает их при помощи функции *equalLists.* В вектор *pointerCollector* записываются указатели на все созданные элементы иерархических списков для последующей очистки перед завершением функции. Функция не имеет возвращаемого значения. Функция была реализована для сокращения исходного кода программы.

2. Функция *bool splitString(string& line, string& line1, string& line2)*

принимает на вход три указателя на строки символов типа *string*: исходная строка, первая и вторая подстрока соответственно. Функция находит разделитель иерархических списков (символ пробела) и записывает подстроки в соответствующие ссылки. В случае успеха, функция возвращает *true.* Если разделитель не был найден или строка содержит разделитель в начале строки, функция вернёт *false.*

3. Функция *Elem* getHList(string line, int& i, Elem* hListHead, vector<Elem*> pointerCollector, int depth = 0)* принимает на вход переменную типа *string*, содержащую иерархический список, ссылку на переменную *i* типа *int&* – индекс символа строки *line*, указатель на иерархический список *hListHead*, вектор указателей *pointerCollector* и переменную *depth* типа *int* – текущая глубина рекурсии, параметр умолчанию равен 0.

Рекурсивная функция обрабатывает иерархический подсписок, который содержится во вхождении строки *line*, начиная с символа с индексом *i.* Следует отметить, что весь иерархический список является иерархическим подсписком, поэтому функция является точкой входа рекурсивного алгоритма. На каждом шаге обработки функция создаёт новый объект иерархического списка (структура *Elem*), указатель на который помещается в вектор *pointerCollector*.

Функция проверяет встречающиеся символы подстроки *string*. Если текущим обрабатываемым символом является допустимый символ, проверяемый функцией *isData*, то элемент с данными, которые устанавливаются с помощью функции *setData,* добавляется к иерархическому подсписку текущего уровня вложенности с помощью функции *push*. Если функция при обработке встречает иерархический подсписок вложенности больше, чем текущая (символ «(»), то функция посредством вызова самой себя с другими аргументами рекурсивно обрабатывает иерархический подсписок и добавляет подсписок к иерархическому подсписку текущего уровня вложенности с помощью функции *push*. В процессе обработки

функция печатает символы и уровень вложенности рекурсии с помощью функции *recursionDepth*.

При достижении символа конца иерархического подписка (символ «)»), функция возвращает указатель, который был передан функции или *NULL* в случае возникновения ошибки анализа строки. Если рекурсивная функция завершает свою работу на первом уровне вложенности рекурсии, то производится проверка на лишние символы подстроки и проверка на совпадающие символы в элементах иерархического списка с помощью функции *isUniqueData*. В случае успешной проверки, функция возвращает указатель, который был передан функции, или *NULL* в противном случае.

4. Функция *bool isData(char c)* принимает на вход символ *c.* Функция возвращает *true,* если символ является символом английского алфавита или цифрой, и *false* в противном случае.

5. Функция *void setData(Elem* hListElem, char data)* принимает на вход указатель на элемент иерархического списка *hListElem* и данные *data* типа *char.* Функция устанавливает поле *dataTag* на *true* и присваивает в поле *data* значения из переменной *data.* Функция не имеет возвращаемого значения.

6. Функция *void push(Elem*& hListHead, Elem* hListTail)* принимает на вход ссылку на указатель первой структуры и указатель на структуру – элементы, образующие иерархический список на данном уровне вложенности. Функция присваивает в поле *tail* первой структуры значения адреса второй структуры и меняет значение переменной первой переменной на вторую. Функция не имеет возвращаемого значения. С помощью этой функции достигается последовательное присоединение элементов иерархического списка.

7. Функция *void recursionDepth(char s, int n, bool sublist = false)* принимает на вход обрабатываемый символ *char s,* текущий уровень вложенности рекурсии *int n* и переменную типа *bool* – индикатор способа

печати, параметр по умолчанию равен *false.* В зависимости от параметра *sublist* функция печатает подзаголовок «Sublist:» с определённым отступом от начала строки или символ со значением уровня вложенности рекурсии. Функция не имеет возвращаемого значения.

8. Функция *bool isUniqueData(Elem* hList, vector<char>& dataCollector)* принимает на вход указатель на иерархический список *hList* и ссылку на вектор сборщика данных *dataCollector*. Рекурсивная функция обходит иерархический список. Если встречается элемент с данными, то производится проверка на наличие такого элемента в списка. В случае провала проверки возвращается *false*. Если рекурсивный алгоритм не вернул *false* на каком-то шаге работы функции, то функция вернёт *true*.

9. Функция *bool equalLists (Elem* hList1, Elem* hList2)* принимает на вход указатели на два иерархических списка. Рекурсивная функция обходит иерархический список. На каждом шаге сравниваются элементы из двух списков. Если встречается элемент с данными, то производится сравнение элементов. Если поля *child, tail* у обоих указателей не равны *NULL*, то функция рекурсивно проверяет соответствующие подсписки. Если рекурсивный алгоритм не вернул *false* на каком-то шаге работы функции, то функция вернёт *true*.

10. Функция *Elem* error(int n)* принимает на вход переменную *n* типа *int* – номер ошибки, из-за которой не выполнилось одно из условий при проверке в функциях, описанных выше. Функция печатает на экран сообщение об ошибке. Функция всегда возвращает *NULL.* Функция реализована для сокращения исходного кода программы.

11. Функция *int main(int argc, char *argv[])* вызывает функцию *lineToLists* и обрабатывает входные данные. После запуска программы пользователю предлагается ввести выражение, и программа напечатает на экран шаги алгоритма и результат работы.

Пользователь может при запуске исполняемого файла ввести в качестве аргумента файл, в котором построчно содержатся пары иерархических списков, разделённые символом переноса строки. Если файл не пуст, то программа обработает каждую строку и запишет все шаги программы в файл «*HLists.txt*».

Исходный код программы представлен в приложении А.

Результаты тестирования программы представлены в приложении Б.

Выводы.

Была реализована программа для создания и анализа иерархических списков. Программа проверяет иерархические списки на эквивалентность. Программа включает в себя рекурсивные функции, благодаря которым осуществляется создание и анализ.

Были выполнены следующие требования: возможность ввода данных из файла или с консоли, вывод сообщений о глубине рекурсии, вывод информации о работе программы в отдельный файл или консоль.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp.

```
#include <iostream>
#include <fstream>
#include <vector>

using namespace std;

typedef struct Elem{
    Elem* tail = NULL;
    Elem* child = NULL;
    char data = 0;
    bool dataTag = 0;
} Elem;

void lineToLists(string& line, vector<Elem*>& pointerCollector);
bool splitString(string &line, string &line1, string &line2);
Elem* getHList(string line, int& i, Elem* hListHead, vector<Elem*>
& pointerCollector, int depth = 0);
bool isUniqueData(Elem* hList, vector<char>& dataCollector);
bool equalLists(Elem* hList1, Elem* hList2);
bool isData(char c);
void recursionDepth(char s, int n, bool sublist = false);
void push(Elem*& hListHead, Elem* hListTail);
void setData(Elem* hListElem, char data);
Elem* error(int n);

int main(int argc, char *argv[]){
    string line;
    vector<Elem*> pointerCollector;
    if(argc==1){
        cout << "Write input lists:\t";
        getline(cin, line);
        lineToLists(line, pointerCollector);
    }else if(argc==2){
        line = argv[1];
        ifstream in(line);
        if(!in){
            error(10);
            return 0;
        }
        ofstream out("HLists.txt");
        streambuf *coutbuf = cout.rdbuf();
        cout.rdbuf(out.rdbuf());
        int lineNum;
        for(lineNum = 1; getline(in, line); lineNum++){
            cout << "\n\tString #" << lineNum << endl;
            lineToLists(line, pointerCollector);
        }
        cout.rdbuf(coutbuf);
        if(lineNum == 1) error(7); else cout << "Results are saved
```

```
 in HLists.txt file." << endl;
    }else error(6);
    for(int i=0; i<pointerCollector.size(); i++) delete pointerCol
lector[i];
    return 0;
}

void lineToLists(string& line, vector<Elem*>& pointerCollector){
    string line1, line2;
    int i;
    if(splitString(line, line1, line2) == 0) error(1);
    else {
        Elem* hList1 = new Elem;
        pointerCollector.push_back(hList1);
        Elem* hList2 = new Elem;
        pointerCollector.push_back(hList2);
        cout << "=======\nList #1:" << endl;
        hList1 = getHList(line1, i=0, hList1, pointerCollector);
        if(hList1 == NULL) error(8);
        cout << "=======\nList #2:" << endl;
        hList2 = getHList(line2, i=0, hList2, pointerCollector);
        if(hList2 == NULL) error(9);
        cout << "=======" << endl;
        if((hList1 == NULL)||(hList2 == NULL)) cout << "Lists are
not comparable." << endl;
        else if(equalLists(hList1, hList2)) cout << "Lists are equ
al." << endl; else cout << "Lists are NOT equal." << endl;
    }
}

bool splitString(string &line, string &line1, string &line2){
    string delimiter = " ";
    int delimiterIndex = line.find(delimiter);
    if(delimiterIndex <= 0) return false;
    line1 = line.substr(0, delimiterIndex);
    line2 = line.substr(delimiterIndex+1, line.length());
    return true;
}

Elem* getHList(string line, int& i, Elem* hListHead, vector<Elem*>
& pointerCollector, int depth){
    if(line[i] == '('){
        Elem* hListLocalTail = new Elem;
        pointerCollector.push_back(hListLocalTail);
        hListHead->child = hListLocalTail;
        if(depth != 0) recursionDepth(0, depth, true);
        for(++i; ((i < line.length())&&(line[i] != ')')); i++){
            Elem* hListBlank = new Elem;
            pointerCollector.push_back(hListBlank);
            if(isData(line[i])){
                setData(hListLocalTail, line[i]);
                recursionDepth(line[i], depth);
            }
            else if(line[i] == '('){
                if(getHList(line, i, hListLocalTail, pointerCollec
```

```
tor, depth+1) == NULL) return NULL;
            }
            else return error(3);
            push(hListLocalTail, hListBlank);
        }
        if(i == line.length()) return error(4);
        if((depth == 0)&&(i+1 < line.length())) return error(5);
        vector<char> dataCollector;
        if(isUniqueData(hListHead, dataCollector) == 0) return err
or(11);
        return hListHead;
    }
    return error(2);
}

bool equalLists(Elem* hList1, Elem* hList2){
    if(hList1->data != hList2->data) return false;
    if((hList1->child != NULL)^(hList2-
>child != NULL)) return false;
    if((hList1->tail != NULL)^(hList2->tail != NULL)) return false;
    if((hList1->child != NULL)&&(hList2-
>child != NULL)) if (equalLists(hList1->child, hList2-
>child) == 0) return false;
    if((hList1->tail != NULL)&&(hList2-
>tail != NULL)) if (equalLists(hList1->tail, hList2-
>tail) == 0) return false;
    return true;
}

void recursionDepth(char s, int n, bool sublist){
    for(int i=0; i<n; i++) cout << "\t";
    if(sublist) cout << "Sublist:" << endl; else cout << "Symbol:
" << s << "\tdepth: " << n << endl;
    return;
}

bool isData(char c){
    return ((c>=48)&&(c<=57))||((c>=65)&&(c<=90))||((c>=97)&&(c<=1
22));
}

void push(Elem*& hListHead, Elem* hListTail){
    hListHead->tail = hListTail;
    hListHead = hListTail;
    return;
}

void setData(Elem* hListElem, char data){
    hListElem->dataTag = 1;
    hListElem->data = data;
    return;
}

bool isUniqueData(Elem* hList, vector<char>& dataCollector){
    if(hList->dataTag){
```

```
        for(int i=0; i<dataCollector.size(); i++) if(dataCollector
[i] == hList->data) return false;
        dataCollector.push_back(hList->data);
    }
    if(hList->child != NULL) if(isUniqueData(hList-
>child, dataCollector) == false) return false;
    if(hList->tail != NULL) if(isUniqueData(hList-
>tail, dataCollector) == false) return false;
    return true;
}

Elem* error(int n){
    cout << "Error!\t";
    switch (n){
        case 0: cout << "Empty string." << endl; break;
        case 1: cout << "Incorrect list string. Need 2 lists separ
ated by space." << endl; break;
        case 2: cout << "Expected \'(\' symbol." << endl; break;
        case 3: cout << "Non-
permitted symbol is string. Permitted symbols for list are: [A-Za-
z0-9()]." << endl; break;
        case 4: cout << "Expected \')\' symbol." << endl; break;
        case 5: cout << "Unexpected sequence after brackets." << e
ndl; break;
        case 6: cout << "Incorrect input arguments. Write file pat
h or two hierarchical lists." << endl; break;
        case 7: cout << "Empty file." << endl; break;
        case 8: cout << "List #1 has incorrect syntax." << endl; b
reak;
        case 9: cout << "List #2 has incorrect syntax." << endl; b
reak;
        case 10: cout << "File not found." << endl; break;
        case 11: cout << "List must have unique data." << endl; br
eak;
    };
    return NULL;
}
```

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ

Таблица 1 - Тестирование программы.

№	Входные данные	Выходные данные	Комментарий
1.		Error! Incorrect list string. Need 2 lists separated by space.	Тест обработки пустой строки.
2.		Error! Incorrect list string. Need 2 lists separated by space.	Тест обработки строки с пробелом.
3.	123678	Error! Incorrect list string. Need 2 lists separated by space.	Тест обработки некорректной строки.
4.	((( )()(	======= List #1: Sublist: Sublist: Error! Expected ')' symbol. Error! List #1 has incorrect syntax. ======= List #2: Error! Unexpected sequence after brackets. Error! List #2 has incorrect	Тест обработки некорректной строки.

		syntax. ======= Lists are not comparable.	
5.	((a()) (b	======= List #1: Sublist: Symbol: a depth: 1 Sublist: Error! Expected ')' symbol. Error! List #1 has incorrect syntax. ======= List #2: Symbol: b depth: 0 Error! Expected ')' symbol. Error! List #2 has incorrect syntax. ======= Lists are not comparable.	Тест обработки некорректной строки.
6.	b) ###?	======= List #1: Error! Expected '(' symbol.	Тест обработки некорректной строки с недопустимыми символами.

		Error! List #1 has incorrect syntax. ======= List #2: Error! Expected '(' symbol. Error! List #2 has incorrect syntax. ======= Lists are not comparable.	
7.	))))))(	======= List #1: Error! Expected '(' symbol. Error! List #1 has incorrect syntax. ======= List #2: Error! Expected '(' symbol. Error! List #2 has incorrect syntax. ======= Lists are not comparable.	Тест обработки некорректной строки.
8.	(((aaa))) (((abcd)ef)bd)	======= List #1:	Тест обработки строки с

		Sublist: Sublist: Symbol: a depth: 2 Symbol: a depth: 2 Symbol: a depth: 2 Error! List must have unique data. Error! List #1 has incorrect syntax. ======= List #2: Sublist: Sublist: Symbol: a depth: 2 Symbol: b depth: 2 Symbol: c depth: 2 Symbol: d depth: 2 Symbol: e depth: 1 Symbol: f depth: 1 Symbol: b depth: 0 Symbol: d depth: 0 Error! List must have unique	неэквивалентными списками.

		data. Error! List #2 has incorrect syntax. ======= Lists are not comparable.	
9.	(a) (b)	======= List #1: Symbol: a depth: 0 ======= List #2: Symbol: b depth: 0 ======= Lists are NOT equal.	Тест обработки строки с неэквивалентными списками.
10.	((a)b) (b(a))	======= List #1: Sublist: Symbol: a depth: 1 Symbol: b depth: 0 ======= List #2: Symbol: b depth: 0 Sublist:	Тест обработки строки с неэквивалентными списками.

		Symbol: a depth: 1 ======= Lists are NOT equal.	
11.	(abcdef) (bcde)	======= List #1: Symbol: a depth: 0 Symbol: b depth: 0 Symbol: c depth: 0 Symbol: d depth: 0 Symbol: e depth: 0 Symbol: f depth: 0 ======= List #2: Symbol: b depth: 0 Symbol: c depth: 0 Symbol: d depth: 0 Symbol: e depth: 0 ======= Lists are NOT equal.	Тест обработки строки с неэквивалентными списками.
12.	(a(bc(d(e)f)ghi (lm(no)))jk) (a(bc(d(e)f)ghi	======= List #1: Symbol: a depth: 0	Тест обработки строки с неэквивалентными

	)jl)	Sublist: Symbol: b depth: 1 Symbol: c depth: 1 Sublist: Symbol: d depth: 2 Sublist: Symbol: e depth: 3 Symbol: f depth: 2 Symbol: g depth: 1 Symbol: h depth: 1 Symbol: i depth: 1 Sublist: Symbol: l depth: 2 Symbol: m depth: 2 Sublist: Symbol: n depth: 3 Symbol: o depth: 3 Symbol: j depth: 0 Symbol: k depth: 0	списками.

		======= List #2: Symbol: a depth: 0 Sublist: Symbol: b depth: 1 Symbol: c depth: 1 Sublist: Symbol: d depth: 2 Sublist: Symbol: e depth: 3 Symbol: f depth: 2 Symbol: g depth: 1 Symbol: h depth: 1 Symbol: i depth: 1 Symbol: j depth: 0 Symbol: l depth: 0 ======= Lists are NOT equal.	
13.	(abcdef) (abcdef)	======= List #1: Symbol: a depth: 0	Тест обработки строки с эквивалентными списками.

		Symbol: b depth: 0 Symbol: c depth: 0 Symbol: d depth: 0 Symbol: e depth: 0 Symbol: f depth: 0 ======= List #2: Symbol: a depth: 0 Symbol: b depth: 0 Symbol: c depth: 0 Symbol: d depth: 0 Symbol: e depth: 0 Symbol: f depth: 0 ======= Lists are equal.	
14.	(a(bc(d(e)f)ghi (lm(no)))jk) (a(bc(d(e)f)ghi (lm(no)))jk)	======= List #1: Symbol: a depth: 0 Sublist: Symbol: b depth: 1 Symbol: c depth: 1 Sublist:	Тест обработки строки с эквивалентными списками.

		Symbol: d depth: 2 Sublist: Symbol: e depth: 3 Symbol: f depth: 2 Symbol: g depth: 1 Symbol: h depth: 1 Symbol: i depth: 1 Sublist: Symbol: l depth: 2 Symbol: m depth: 2 Sublist: Symbol: n depth: 3 Symbol: o depth: 3 Symbol: j depth: 0 Symbol: k depth: 0 ======= List #2: Symbol: a depth: 0 Sublist:	

		Symbol: b depth: 1 Symbol: c depth: 1 Sublist: Symbol: d depth: 2 Sublist: Symbol: e depth: 3 Symbol: f depth: 2 Symbol: g depth: 1 Symbol: h depth: 1 Symbol: i depth: 1 Sublist: Symbol: l depth: 2 Symbol: m depth: 2 Sublist: Symbol: n depth: 3 Symbol: o depth: 3 Symbol: j depth: 0 Symbol: k depth: 0 =======	

		Lists are equal.	
15.	() ()	======= List #1: ======= List #2: ======= Lists are equal.	Тест обработки строки с эквивалентными списками.