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

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Алгоритмы и структуры данных»
Тема: Иерархические списки

Студент гр. 7381 _____ Ильясов А.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2018

Задание

13 вариант:

Вычислить глубину (число уровней вложения) иерархического списка как максимальное число одновременно открытых левых скобок в сокращённой скобочной записи списка; принять, что глубина пустого списка и глубина атомарного S-выражения равны нулю; например, глубина списка (a (b ( ) c) d) равна двум.

В работе используется язык программирования C++.

Пояснение задания

Задача состоит в том, чтобы пройтись по всем уровням вложенности иерархического списка, и из них выбрать наибольший, так как может быть ситуация, когда максимальная глубина списка достигается в его середине или конце.

Описание алгоритма

Создана рекурсивная функция вычисления глубины иерархического списка, принимающая в качестве аргумента проверяемый список, в которой при удачном создании иерархического списка, производится проход по первому уровню вложенности до тех пор, пока это возможно. Если просматриваемый узел списка указывает на атом, функция возвращает 1, в противном же случае, узел указывает на подсписок, являющийся также иерархическим списком, и тогда снова вызывается эта же функция, только уже с подсписком в качестве аргумента, который в свою очередь также обрабатывается, как и основной. Этот процесс продолжается дло тех пор, пока не встречается пустой список, в этом случае возвращается 0, либо же проверка достигает последнего элемента в списке.

После каждого завершения вызова данной функции, имеется результат в виде глубины подсписка, который сравнивается с текущим максимальным. И если он больше текущего максимального, то перезаписывается в переменную,

хранящую максимальное значение глубины списка. Таким образом производится выбор максимальной глубины иерархического списка.

Описание функций

Базовые функции:

lisp head(const lisp list) - функция перехода в голову узла списка.

const lisp list - список, в котором требуется перейти в голову первого узла.

Возвращаемое значение: если list пуст, то выводится сообщение об ошибке и программа завершает работу, если же список не пуст и первый узел указывает на подсписок, то возвращается этот подсписок, в противном случае узел указывает на атом, при этом выводится сообщение об ошибке и программа завершает работу.

lisp tail(const lisp list) - функция перехода в хвост списка.

const lisp list - список, в котором требуется перейти в хвост.

Возвращаемое значение: если list пуст, то выводится сообщение об ошибке и программа завершает работу. Если же список не пуст и он не атом , то возвращается этот же список, начиная со второго узла, в противном случае список состоит из одного атома, при этом выводится сообщение об ошибке и программа завершает работу.

lisp cons(const lisp interim_list_1, const lisp interim_list_2) - функция создания нового списка из interim_list_1 и interim_list_2.

const lisp interim_list_1 - список, который будет вставлен в голову нового списка.

const lisp interim_list_2 — список, который будет хвостом нового списка.

Возвращаемое значение: если interim_list_2 атом, то выводится сообщение об ошибке и программа завершает работу. Если interim_list_2 -

список, то создается новый список, в хвост которого помещается interim_list_2, а в голову interim_list_1.

lisp make_atom(const char value) - функция создания списка, состоящего из одного атома.

const char value - символ, который будет значением атома в списке.

Возвращаемое значение: список, с одним атомом.

bool isAtom(const lisp list) - функция, проверяющая является ли list атомом.

const lisp list - проверяемый список.

Возвращаемое значение: true - атом, false - в противном случае.

bool isNull(const lisp list)- функция, проверяющая является ли list пустым списком.

const lisp list - проверяемый список.

Возвращаемое значение: true - список пуст, false - в противном случае.

char getAtom(const lisp list) - функция, возвращающая значения атома list.

const lisp list - список, являющийся атомом.

Возвращаемое значение: зачение поля atom, если список - атом. Если же список не является атомом, то выводится сообщение об ошибке и программа завершает работу.

void destroy(lisp list) - функция удаления списка.

lisp list - список, который требуется удалить.

Возвращаемое значение: функция ничего не возвращает.

Функции ввода:

`void read_lisp(lisp &list)` - основная функция создания списка. Считывает символ с клавиатуры и направляет его в следующую функцию `read_s_expr`.

`lisp &list` - ссылка на создаваемый список.

Возвращаемое значение: функция ничего не возвращает.

`void read_s_expr(char prev, lisp &list)` - функция, определяющая дальнейшее создание списка по аргументу `prev`. Если `prev` - символ `)`, то выводится сообщение об ошибке и программа завершает работу, так как список не может начинаться с этого символа. Если `prev` является символом `(`, то вызывается следующая функция `read_seq` для создания списка или подсписков.

`char prev` - символ, считанный в функции `read_lisp` и который определяет, как дальше будет создаваться список.

`lisp &list` - ссылка на создаваемый список.

Возвращаемое значение: функция ничего не возвращает.

`void read_seq(lisp &list)` - функция, считывающая и объединяющая основной список и подсписки.

`lisp &list` - ссылка на создаваемый список.

Возвращаемое значение: функция ничего не возвращает.

Функция задания лабораторной работы:

`unsigned int list_depth_count(const lisp list, unsigned int counter)` - функция, вычисляющая максимальную глубину иерархического списка. Описание работы функции описано в предыдущем пункте.

`const lisp list` - список, максимальную глубину которого требуется найти.

unsigned int counter - счетчик промежуточных значений глубины списка для вывода промежуточных данных.

Возвращаемое значение: функция возвращает максимальное значение глубины иерархического списка.

Тестирование

Для проверки работоспособности программы был создан скрипт(см. ПРИЛОЖЕНИЕ Д) для автоматического ввода и вывода тестовых данных:

correct test1.txt: ();

correct test2.txt: d;

correct test3.txt: (ab(cd(ef)));

correct test4.txt: (ab(cd)ef(gh(ij))kl(mn(op(qr))));

correct test5.txt: (ab(cd())ef);

incorrect test1.txt:)(;

incorrect test2.txt: (abcd(df()).

Результаты тестирования сохраняются в файл testsresult.txt.

Ниже представлена таблица тестирования программы с полным выводом для одного из тестов.

Входные данные	Выходные данные
()	correct test: test1.txt test data: () result: List is empty or was not created
d	correct test: test2.txt test data: d result: The depth of this hierarchical list is 0.
(ab(cd(ef)))	correct test: test3.txt test data: (ab(cd(ef))) result: a b ------> c

	d -------> e f The depth of this hierarchical list is 3.
(ab(cd)ef(gh(ij))kl(mn(op(qr))))	correct test: test4.txt test data: (ab(cd)ef(gh(ij))kl(mn(op(qr)))) result: a b -------> c d e f -------> g h -------> i j k l -------> m n -------> o p -------> q r The depth of this hierarchical list is 4.
(ab(cd())ef)	correct test: test5.txt test data: (ab(cd())ef) result: a b -------> c d -------> e

	f The depth of this hierarchical list is 2.
)(	incorrect test: test1.txt test data:)(result: --Error: unexpected ')'--
(abcd(df())	incorrect test: test2.txt test data: (abcd(df()) result: --Error: data entry expected--

Выводы

В процессе выполнения лабораторной работы были поли получены знания и навыки по иерархическим спискам, рекурсивным функциям, bash-скриптам и автоматизации тестирования. Работа была написана на C++.

ПРИЛОЖЕНИЕ А

КОД MAIN_LAB2.CPP

```
#include <iostream>
#include <cstdlib>

#include "hierar_list.hpp"

int main() {
	lisp list;
	read_lisp(list);
	if (isNull(list))
		std::cout << "List is empty or was not created" << std::endl;
	else {
		std::cout << "The depth of this hierarchical list is " <<
list_depth_count(list, 0) << "." << std::endl;
		destroy(list);
	}
return 0;
}
```

ПРИЛОЖЕНИЕ Б

ФАЙЛ HIERAR_LIST.CPP

```
#include <iostream>
#include <cstring>
#include <cstdlib>
#include <cctype>

#include "hierar_list.hpp"

// базовые функции:
lisp head(const lisp list) {
	if (list) {
		if (!isAtom(list))
			return list->node.pair.head;
		else {
			std::cout << "--Error: Head(atom)--" << std::endl;
			exit(0);
		}
	}
	else {
```

```
                std::cout << "--Error: Head(nil)--" << std::endl;
                exit(0);
        }
}

lisp tail(const lisp list) {
        if (list) {
                if (!isAtom(list))
                        return list->node.pair.tail;
                else {
                        std::cout << "--Error: Tail(atom)--" << std::endl;
                        exit(0);
                }
        }
        else {
                std::cout << "--Error: Tail(nil)--" << std::endl;
                exit(0);
        }
}

lisp cons(const lisp interim_list_1, const lisp interim_list_2) {
        lisp list;
        if (isAtom(interim_list_2)) {
                std::cout << "--Error: Cons(*, atom)--" << std::endl;
                exit(0);
        }
        else {
                list = new s_expr;
                if (!list) {
                        std::cout << "--Memory not enough--" << std::endl;
                        exit(0);
                }
                else {
                        list->tag = false;
                        list->node.pair.head = interim_list_1;
                        list->node.pair.tail = interim_list_2;
                        return list;
                }
        }
}

lisp make_atom(const char value) {
        lisp list = new s_expr;
        list->tag = true;
        list->node.atom = value;
```

```
	return list;
}

bool isAtom(const lisp list) {
	return list ? list->tag : false;
}

bool isNull(const lisp list) {
	return !list;
}

char getAtom(const lisp list) {
	if (!isAtom(list)) {
		std::cout << "--Error: getAtom(!atom)--" << std::endl;
		exit(0);
	}
	else
		return list->node.atom;
}

void destroy(lisp list) {
	if (list) {
		if (!isAtom(list)) {
			destroy(head(list));
			destroy(tail(list));
		}
	delete list;
	}
}
// функции ввода:
void read_lisp(lisp &list) {
	char x;
	do std::cin >> x; while (x == ' ');
	read_s_expr(x, list);
}

void read_s_expr(char prev, lisp &list) {
	if (prev == ')') {
		std::cout << "--Error: unexpected ')'--" << std::endl;
		exit(0);
	}
	else if (prev != '(')
		list = make_atom(prev);
	else
		read_seq(list);
```

```
}

void read_seq(lisp &list) {
	char x;
	lisp p1, p2;
	if (!(std::cin >> x)) {
		std::cout << "--Error: data entry expected--" << std::endl;
		exit(0);
	}
	else {
		while (x == ' ') std::cin >> x;
		if (x == ')')
			list = NULL;
		else {
			read_s_expr(x, p1);
			read_seq(p2);
			list = cons(p1, p2);
		}
	}
}
// Функция подсчета глубины списка
unsigned int list_depth_count(const lisp list, unsigned int counter) {
	unsigned int max_depth_count = 1;
	unsigned int depth_count = 0;

	if (isNull(list) || isAtom(list))                                    //
Глубина пустого списка и глубина
		return 0;                                                    //
атомарного S-выражения равны нулю

	for (lisp viewed = list; viewed; viewed = viewed->node.pair.tail) {
		if (!isNull(viewed)) {
			depth_count++;
			if (!isAtom(viewed)) {

				if (isAtom(head(viewed))) {                          //
выражение
					for (int i = 0; i < counter; i++)            //
для
						std::cout << "\t";          //
вывода
					std::cout << head(viewed)->node.atom <<
std::endl;    // промежуточного
				}                                                    //
результата,
```

```
                        else {                                          //
демонстрирующего
                                for (int i = 0; i < counter; i++)       //
визуализацию
                                        std::cout << "\t";             //
уровней
                                std::cout << "------->" << std::endl;
        // вложенности
                        }                                               // списка

                        depth_count += list_depth_count(viewed-
>node.pair.head, counter+1);
                }
                else {
                        return 1;
                }
        }
        else {
                return 0;
        }

        if (depth_count > max_depth_count)
        // выбор максимальной
                max_depth_count = depth_count;
        // глубины вложенности

        depth_count = 0;
    }
return max_depth_count;
}
```

ПРИЛОЖЕНИЕ В
ФАЙЛ HIERAR_LIST.HPP

```
#ifndef HIERAR_LIST_HPP
#define HIERAR_LIST_HPP

// описание структуры узла иерархического списка
    struct s_expr;
    struct two_ptr {
        s_expr *head;
        s_expr *tail;
    };
```

```
        struct s_expr {
                bool tag; // true: atom, false: pair
                union {
                        char atom;
                        two_ptr pair;
                } node;
        };
        typedef s_expr *lisp;
// базовые функции:
        lisp head(const lisp list);
        lisp tail(const lisp list);
        lisp cons(const lisp interim_list_1, const lisp interim_list_2);
        lisp make_atom(const char value);
        bool isAtom(const lisp list);
        bool isNull(const lisp list);
        char getAtom(const lisp list);
        void destroy(lisp list);
// функции ввода:
        void read_lisp(lisp& list);
        void read_s_expr(char prev, lisp &list);
        void read_seq(lisp &list);
// Функция подсчета глубины списка
        unsigned int list_depth_count(const lisp list, unsigned int counter);
#endif
```

ПРИЛОЖЕНИЕ Г
MAKEFILE

```
all: lab2

lab2: main_lab2.o hierar_list.o
        g++ main_lab2.o hierar_list.o -o lab2

main_lab2.o: ./Source/main_lab2.cpp
        g++ -c ./Source/main_lab2.cpp

hierar_list.o: ./Source/hierar_list.cpp
        g++ -c ./Source/hierar_list.cpp

clean:
        rm main_lab2.o hierar_list.o
```

ПРИЛОЖЕНИЕ Д
ФАЙЛ RUNTESTS.SH

```
#!/bin/bash

if test ! -f "lab2" ; then
        make
```

```
        clear
fi
if test -f "testsresult.txt"; then
        rm testsresult.txt
fi

touch testsresult.txt

for i in $(ls ./Tests/correct); do
        echo "running correct $i: $(cat Tests/correct/$i | more)"
                sleep 0.1s
        echo "correct test: "$i"" >> testsresult.txt
        echo "test data: $(cat Tests/correct/$i | more)" >> testsresult.txt
        echo "result: " >> testsresult.txt
        ./lab2 < ./Tests/correct/$i >> testsresult.txt
        echo -e >> testsresult.txt
        echo
"##########################################################
####" >> testsresult.txt
done
for i in $(ls ./Tests/incorrect); do
        echo "running incorrect $i: $(cat Tests/incorrect/$i | more)"
                sleep 0.1s
        echo "incorrect test: "$i"" >> testsresult.txt
        echo "test data: $(cat Tests/incorrect/$i | more)" >> testsresult.txt
        echo "result: " >> testsresult.txt
        ./lab2 < ./Tests/incorrect/$i >> testsresult.txt
        echo -e >> testsresult.txt
        echo
"##########################################################
####" >> testsresult.txt
done

sleep 0.2s
echo -e
echo  "test results are saved in testsresult.txt"
```

ПРИЛОЖЕНИЕ Е
ФАЙЛ RUN.SH

```
#!/bin/bash

if test ! -f "lab2"; then
        make
        clear
fi
if test -f "result.txt"; then
        rm result.txt
fi
```

```
echo -n "Select the output method: 1 - on the screen; 2 - to the file: "
read output_method

if [[ $output_method == 1 || $output_method == 2 ]]; then
        echo -n "Enter abbreviated entry of the hierarchical list: "
        if test $output_method -eq 1; then
                ./lab2
                exit
        fi
        if test $output_method -eq 2; then
                echo "result:" >> result.txt
                ./lab2 >> result.txt
                echo "Result of the program is saved in result.txt"
                exit
        fi
else
        echo "Input error! Enter 1 or 2!"
        exit
fi
```