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

Студентка гр. 9382 _____ Круглова В.Д.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Получение знаний в области рекурсивной обработки иерархических списков. Освоение основных функций для работы с иерархическими списками.

Основные теоретические положения.

Иерархический список представляет собой список, элементы, которых могут быть также списки и т.д. Иерархические списки позволяют хранить информацию в соответствии с подчинением одних данных другим.

Задание.

Вариант 7.

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

Описание структур данных для реализации иерархических списков.

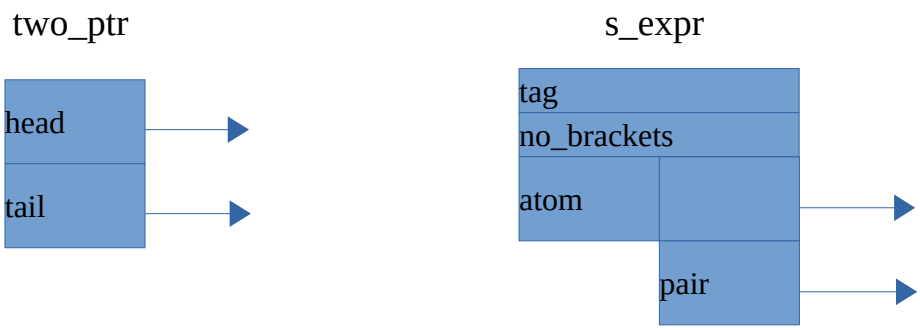
```
struct two_ptr
{
   s_expr *head;
   s_expr *tale;
};

struct s_expr
{
   bool tag;  // true: atom, false: pair
   int no_brackets = 0;
   union
   {
      base atom;
      two_ptr pair;
   } node;
};
typedef s_expr* lisp;
```

two_ptr состоит из двух указателей на объекты head и tale структуры s_expr.

s_expr состоит из логической переменной tag которая выставляется в зависимости от того хранится в этой структуре символ или объект первой структуры, флаг no_brackets, позволяющий при желании не выводить ненужные сковки, union в котором хранится либо атом, либо пара.

С помощью typedef создается возможность использовать lisp вместо указателя на s_expr.

Графическая схема примера иерархического списка.



Описание основных функций.

void destroy (lisp s)

Назначение: Очищает память, выделенную под пару и под сам переданный объект.

Описание аргументов: Указатель на иерархический список.

Возвращаемое значение: Функция ничего не возвращает.

int del_needed_atoms(lisp s, base atom, int depth)

Назначение: Удаляет атом, если это единственный элемент списка и он нужный, запускает рекурсию. Если это атом, то проверяет его на соответствие искомому, иначе вызывает рекурсивную функцию del_atoms_Add.

Описание аргументов: Указатель на иерархический список s, искомый atom, depth (глубина рекурсии).

Возвращаемое значение: 1, если вызов del_atoms_Add вернул единицу, иначе 0.

int del_atoms_Add(lisp s, base atom, int depth, int loc)

Назначение: Рекурсивно проверяет голову и хвост списка на соответствие введенному атому.

Описание аргументов: Указатель на иерархический список s, искомый atom, depth (глубина рекурсии), loc (флаг для вывода).
Возвращаемое значение: 1, если в переданном списке найден искомый атом, иначе 0.

Описание алгоритма.

Вызывается функция del_needed_atoms, в которой происходит проверка на пустой список (выводятся пустые скобки и сообщение, о том, что список пуст, возвращается 0), список, состоящий из одного искомого элемента (в этом случае список выводится и элемент удаляется, возвращается 1) или не из искомого (выводится список, завершается работа функции).

Если список – пара, то вызывается рекурсивная функция del_atoms_Add, где с учетом глубины рекурсии выполняются следующие проверки:

1. Голова пуста
    1. Хвост пуст (возвращается 0)
    2. Хвост атом
        1. Искомый (очищается помять, выделенную под хвост, обнуляется указатель на хвост, возвращается 1)
        2. Не искомый (возвращается 0)
    3. Хвост пара (вызывается эта же функция для хвоста)
2. Голова атом
    1. Искомый
        1. Хвост пуст (очищается память, выделенная под голову, обнуляется указатель на голову, возвращается 1)
        2. Хвост атом
            1. Искомый (очищается память, выделенная под голову, обнуляется указатель на голову, очищается помять, выделенную под хвост, обнуляется указатель на хвост, возвращается 1)
            2. Не искомый (то, что было в указателе на хвост, помещается в указатель на голову, обнуляется указатель на хвост, возвращается 1)
            3. Хвост пара (то, что было в указателе на хвост, помещается в указатель на голову, обнуляется указатель на хвост, вызывается эта же функция для головы)
    2. Не искомый
        1. Хвост пуст (возвращается 0)
        2. Хвост атом

1. Искомый (очищается помять, выделенную под хвост, обнуляется указатель на хвост, возвращается 1)
2. Не искомый (возвращается 0)

3. Хвост пара (вызывается эта же функция для хвоста)

3. Голова пара
1. Хвост пуст (вызывается эта же функция для головы)
2. Хвост атом
1. Искомый (очищается помять, выделенную под хвост, обнуляется указатель на хвост, вызывается эта же функция для головы)
2. Не искомый (вызывается эта же функция для головы)

3. Хвост пара (вызывается эта же функция для головы и для хвоста).

Пример работы программы.

Таблица 1 – Пример работы

Входные данные	Выходные данные
((ewe)t) e	Введите список Вы ввели: ((e w e) t) Введите атом Вы ввели: e Program start---------- [] Now in second function [] ((e w e) t) [] HEAD is a 'pair' and TAIL is a 'pair' too. [] [] Now in the HEAD of lisp [] [] (e w e) [] [] HEAD is needed atom and TAIL is a 'pair'. [] [] [] Now in the HEAD of lisp [] [] [] (w e) [] [] [] HEAD is a wrong atom and TAIL is a 'pair'. [] [] [] [] Now in the TAIL of lisp [] [] [] [] (e) [] [] [] [] HEAD is needed atom and TAIL is empty. [] [] Now in the TAIL of lisp [] [] (t) [] [] HEAD is a wrong atom and TAIL is empty Изначальный список: ((e w e) t) Форматированный список: ((w) t) Хорошая работа!

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 — Результаты тестирования

No	Входные данные	Выходные данные	Комментарии
1	1 1	Изначальный список: 1 Форматированный список:	Простейший случай
2	asdfgh r	asdfgh	Некорректный ввод
3	(asf)asdfgh f	asdfgh	Некорректный ввод
4	(e(e(e(ebe)))) e	Изначальный список: (e (e (e (e b e)))) Форматированный список: ((((b))))	Большая вложенность
5	(1234567890(1234567890(1234567890)))) (123)	Изначальный список: (1 2 3 4 5 6 7 8 9 0 (1 2 3 4 5 6 7 8 9 0 (1 2 3 4 5 6 7 8 9 0))) Форматированный список: (1 2 3 4 5 6 7 8 9 0 (1 2 3 4 5 6 7 8 9 0 (1 2 3 4 5 6 7 8 9 0)))	Много элементов, некорректный ввод атома(читается только первый символ)
6	(#[[A#[[A#[[A#[[B#[[B#[[B#[[C#[[C#[[C#[[D#[[D#[[D) A	(# [A # [A # [A # [B # [B # [B # [C # [C # [C # [D # [D # [D) Форматированный список: (# [# [# [# [B # [B # [B # [C # [C # [C # [D # [D # [D)	При засоренном символами вводе
7	() a	Изначальный список: () Форматированный список: ()	При вводе пустых скобок
8	)	! List.Error 1 :: no open bracket	Некорректный ввод
9	(qqqqqqqqqqqqqqq) q	Изначальный список: (q q q q q q q q q q q q q q q) Форматированный список: ( )	Граничные данные

10	(o(o(o(o(o))))) o	Изначальный список: (o (o (o (o (o))))) Форматированный список: ((((( )))))	Граничные данные

Выводы.

Получены знания в области работы с иерархическими списками. Освоены методы работы с такими списками (рекурсивные и нет). Написана работающая программа на языке C++, способная удалять из иерархического списка все вхождения введенного атома, сохраняя уровень вложенности.

ПРИЛОЖЕНИЕ С КОДОМ

main.cpp :

```
#include "headers.h"

using namespace std;

int main()
{
    lisp l;
    base atom;

    cout << "Введите список" << endl;
    read_lisp (l);
    cout << "Вы ввели: " << endl;
    write_lisp (l);
    cout << endl;

    free_cin(); // чтобы не оставалось ничего лишнего в cin

    cout << "Введите атом" << endl;
    cin >> atom;
    cout << "Вы ввели:" << endl;
    cout << atom << endl;

    free_cin();

    lisp old_lisp = copy_lisp(l); // чтобы вывести, что было изначально

    del_needed_atoms(l, atom, 0);

    cout << "Изначальный список:     ";
    write_lisp(old_lisp);
    cout << endl;
    cout << "Форматированный список:     ";
    write_lisp(l);
    cout << endl;

    destroy(l); // рекурсивно отчищаем выделенную под списки память
    destroy(old_lisp);

    cout << "Хорошая работа!" << endl;
    return 0;
}
```

funcs.cpp :

```
#include "headers.h"

using namespace std;

// Создание и удаление списка
lisp head (const lisp s) //возвращает именно указатель на голову из поданного
списка, если подать атом -> будет ошибка (поэтому нужно проверять на атом)
{
    if (s != NULL)
    {
        if (!isAtom(s))
            return s->node.pair.head;
```

```
        else
        {
            cerr << "Error: Head(atom) \n";
            exit(1);
        }
    }
    else
    {
        cerr << "Error: Head(nil) \n";
        exit(1);
    }
}

lisp tail (const lisp s) // возвращает именно указатель на хвост из поданного
списка, если подать атом -> ошибка
{
    if (s != NULL)
    {
        if (!isAtom(s))
            return s->node.pair.tale;
        else
        {
            cerr << "Error: Tail(atom) \n";
            exit(1);
        };
    }
    else
    {
        cerr << "Error: Tail(nil) \n";
        exit(1);
    };
}

lisp cons (const lisp h, const lisp t) // создает и возвращает новый lisp из
головы и хвоста
{
    lisp p;
    if (isAtom(t))
    {
        cerr << "Error: cons(*, atom) \n";
        exit(1);
    }
    else
    {
        p = new s_expr;
        if ( p == NULL) // не удалось выделить память
        {
            cerr << "Memory ...\n";
            exit(1);
        }
        else
        {
            p->tag = false; // сделали значение tag = пара и заполнили поля
            p->node.pair.head = h;
            p->node.pair.tale = t;
            return p;
        };
    };
}

lisp make_atom (const base x) // из символа char делает lisp и возвращает
{
    lisp s;
```

```
    s = new s_expr;
    s -> tag = true; // перевели tag в значение атом и записали этот символ
    s->node.atom = x;
    return s; // возвращаем указатель
}

void destroy (lisp s) // рекурсивно все очищаем (и объект, и память выделенную
внутри)
{
    if ( s != NULL)
    {
        if (!isAtom(s)) // если это пара, то рекурсивно очищаем выделенную
память для головы и хвоста
        {
            destroy ( head (s));
            destroy ( tail(s));
        };
        delete s; // очищаем память для самого объекта
    };
}


// Работа со списком
bool isAtom (const lisp s) // возвращает tag из поданого списка
{
    if(s == NULL)
        return false;
    else
        return (s -> tag);
}

bool isNull (const lisp s) {return s==NULL;} // проверка на пустоту списка

lisp copy_lisp (const lisp x) // делает и возвращает копию поданного списка
{
    if (isNull(x))
        return NULL;
    else if (isAtom(x))
        return make_atom (x->node.atom);
    else
        return cons (copy_lisp (head (x)), copy_lisp (tail(x)));
}

base getAtom (const lisp s) // вернет значение атома (переведет из lisp в char),
если не атом - exit(1)
{
    if (!isAtom(s))
    {
        cerr << "Error: getAtom(s) for !isAtom(s) \n";
        exit(1);
    }
    else
        return (s->node.atom);
}


//вывод списка на консоль
void write_lisp (const lisp x) // выведет пусто или атом и вызовет следующую
функцию
{
    if (isNull(x))   //пустой список
    {
        cout << " ()";
```

```
    }
    else if (isAtom(x))  // весь список - атом
    {
        cout << ' ' << x->node.atom;
    }
    else //непустой список
    {
        cout << " (" ;
        write_seq(x); // вызов следующей функции
        cout << " )";
    }
}

void write_seq (const lisp x)//выводит список без внешних скобок (их выведет
прошлая функция)
{
    if (!isNull(x)) // эта проверка нужна для нормальной работы прямой рекурсии
этой функции
    {
        if (isNull(head(x)) && isNull(tail(x))) // если хвост и голова пусты и
не должно ничего выводиться
        {
            if (x->no_brackets == 1)
                return;
        }
        else if (isAtom(head(x)) && isNull(tail(x))) // если в голову перенесли
атом
        {
            if (x->no_brackets == 1)
            {
                cout << ' ' << x->node.atom;
                return;
            }
        }
        else if (isNull(tail(x)))                    // если в голову перенесли
пару
        {
            if (x->no_brackets == 1)
            {
                write_seq(head(x));
                return;
            }
        }

        write_lisp(head (x)); // если head будет NULL -  будет выведено - ()
        write_seq(tail (x));  // а если tail будет NULL - ничего не выведется

    };
}


// ввод списка с консоли
void read_lisp (lisp& y) // считывает символ и передает его со списком дальше
{
    base x;
    do cin >> x; while (x==' ');
    read_s_expr (x, y);
}

void read_s_expr (base prev, lisp& y)//'(' -> вызов следующей функции; иное ->
записали как atom
{
```

```
    if ( prev == ')' ) // есть закрывающая скобка - нет открывающей (словит и
обратное) -> exit(1)
    {
        cerr << " ! List.Error 1 :: no open bracket" << endl;
        exit(1);
    }
    else if (prev != '(')
        y = make_atom (prev);
    else
        read_seq (y); // передали в следующую функцию
}

void read_seq (lisp& y)  // рекурсивное считывание всего, что внутри скобок и
создание иерарх. списка в lisp y
{
    base x;
    lisp p1, p2;
    if (!(cin >> x)) // после открывающей скобки ничего -> exit(1)
    {
        cerr << " ! List.Error 2 :: nothing after open bracket" << endl;
        exit(1);
    }
    else
    {
        while (x==' ') // это пропускает пробелы
            cin >> x;
        if (x == ')') // пустые скобки
            y = NULL;
        else
        {
            read_s_expr (x, p1);
            read_seq (p2);
            y = cons (p1, p2); // последовательное создание иерархического
списка с самого глуокого вложения
        }
    };
}
```

my_funcs.cpp :

```
#include "headers.h"

using namespace std;

void print_depth(int depth) // печать чтобы показать глубину
{
    for (int j = 0; j < depth; j++)
        cout << "[] ";
    return;
}

void free_cin() // чтобы очистить поток ввода
{
    do{}
    while(getchar() != '\n');
}

int del_needed_atoms(lisp s, base atom, int depth)
{
    cout << "Program start----------------------------------------------------"
<< endl;
```

```
    if (isNull(s))      // если весь список = ( )
    {
        print_depth(depth);
        cout << "( )" << endl;

        print_depth(depth);
        cout << "It is empty Lisp." << endl;
        return 0;
    }

    else if (isAtom(s)) // если весь список состоит из одного атома
    {
        if (getAtom(s) == atom) // и это искомый атом
        {
            print_depth(depth);
            write_lisp(s);
            cout << endl;

            print_depth(depth);
            cout << "This lisp was needed atom." << endl;

            s->node.atom = '\0'; // просто удалить с пом. destroy и обнулить
указатель нельзя, т.к.
                                 // сам указатель в main не изменится -> будет
ошибка сегментирования
            return 1;
        }
        else // и это не тот атом
        {
            print_depth(depth);
            write_lisp(s);
            cout << endl;

            print_depth(depth);
            cout << "This lisp is a wrong atom." << endl;
            return 0;
        };
    }

    else                // список состоит из пары указателей head и tail
    {
        if (del_atoms_Add(s, atom, depth + 1,0))
            return 1;
        else
            return 0;
    }
}


int del_atoms_Add(lisp s, base atom, int depth, int loc)
{
    print_depth(depth);
    if (loc == 0)
        cout << "Now in second function" << endl;
    else if (loc == 1)
        cout << "Now in the HEAD of lisp" << endl;
    else if (loc == 2)
        cout << "Now in the TAIL of lisp" << endl;

    if (isNull(head(s)))     // у переданной пары голова = ()
    {
        print_depth(depth);
        write_lisp(s);
```

```
        cout << endl;

        print_depth(depth);
        cout << "HEAD = ()";

        if (isNull(tail(s)))      // а хвост пуст
        {
            cout << " and TAIL is empty." << endl;
            return 0;
        }

        else if (isAtom(tail(s))) // а хвост - это атом
        {
            if (getAtom(tail(s)) == atom) // который нужно удалить
            {
                destroy(tail(s)); // удалили хвост
                s->node.pair.tale = NULL; // обнулили указатель

                cout << " and TAIL is needed atom." << endl;
                return 1;
            }
            else                          // который нам не подходит
            {
                cout << " and TAIL is a wrong atom." << endl;
                return 0;
            }
        }

        else                      // а хвост - пара указателей head и tail
        {
            cout << " and TAIL is a \'pair\'" << endl;
            del_atoms_Add(tail(s), atom, depth + 1, 2); // вызываем эту же
функцию для хвоста
        }
    }

    else if(isAtom(head(s))) // у переданной пары голова - атом
    {
        print_depth(depth);
        write_lisp(s);
        cout << endl;

        if (getAtom(head(s)) == atom) // причем тот, который нужно удалить
        {
            print_depth(depth);
            cout << "HEAD is needed atom";

            if (isNull(tail(s)))      // а хвост пустой
            {
                cout << " and TAIL is empty." << endl;

                destroy(head(s));         // удалили голову
                s->node.pair.head = NULL; // обнулили указатель в паре
                s->no_brackets = 1;       // сделали так, чтобы не выводились
лишние скобки
                return 1;
            }

            else if (isAtom(tail(s))) // а хвост тоже атом
            {
                if (getAtom(tail(s)) == atom)   // и тоже тот, который нужно
удалить
                {
```

```
                    cout << " and TAIL is needed atom too." << endl;

                    destroy(head(s));         // удалили голову
                    s->node.pair.head = NULL; // обнулили указатель

                    destroy(tail(s));         // удалили голову
                    s->node.pair.tale = NULL; // обнулили указатель

                    s->no_brackets = 1;       // чтобы не выводились лишние
скобки
                    return 1;
                }
                else                            // но не тот, что нужен
                {
                    cout << " and TAIL is a wrong atom." << endl;

                    s->node.pair.head = s->node.pair.tale; // теперь в голове -
ненужный атом
                    s->node.pair.tale = NULL;              // а в хвосте -
ничего
                    s->no_brackets = 1;                    // чтобы не
воводились лишние скобки
                    return 1;                              // и весь список
теперь = а
                }
            }

            else                      // а хвост - пара указателей head и tail
            {
                cout << " and TAIL is a \'pair\'." << endl;

                s->node.pair.head = s->node.pair.tale; // теперь в голове - пара
указателей из хвоста
                s->node.pair.tale = NULL;              // а в хвосте - ничего
                s->no_brackets = 1;                    // чтобы не выводились
лишние скобки

                del_atoms_Add(head(s), atom, depth + 1, 1); // вызываем
следующую функцию для head
            }
        }

        else                          // но это не искомый атом
        {
            print_depth(depth);
            cout << "HEAD is a wrong atom";

            if (isNull(tail(s)))     // а хвост пустой
            {
                cout << " and TAIL is empty" << endl;

                return 0;
            }

            else if(isAtom(tail(s))) // а хвост - атом
            {
                if (getAtom(tail(s)) == atom) // причем тот, который нужно
удалить
                {
                    cout << " and TAIL is needed atom." << endl;

                    destroy(tail(s));         // очистили хвост
                    s->node.pair.tale = NULL; // и обнулили указатель
```

```
                    return 1;
                }
                else                            // но он тоже неподходящий
                {
                    cout << " and TAIL is a wrong atom too. " << endl;

                    return 0;
                }
            }

            else                        // а хвост - пара указателей head и tail
            {
                cout << " and TAIL is a \'pair\'." << endl;

                del_atoms_Add(tail(s), atom, depth + 1, 2);
            }
        }
    }

    else                        // у переданной пары голова - пара указателей head
и tail
    {
        print_depth(depth);
        write_lisp(s);
        cout << endl;

        print_depth(depth);
        cout << "HEAD is a \'pair\'";

        if(isNull(tail(s)))      // а хвост пустой
        {
            cout << " and TAIL is empty." << endl;

            del_atoms_Add(head(s), atom, depth + 1, 1);
        }

        else if(isAtom(tail(s))) // а хвост - атом
        {
            if (getAtom(tail(s)) == atom)   // причем тот, который нужно удалить
            {
                cout << " and TAIL is needed atom." << endl;

                destroy(tail(s));         // очищаем хвост
                s->node.pair.tale = NULL; // и обнуляем указатель

                del_atoms_Add(head(s), atom, depth + 1, 1); // вызываем для
головы
            }
            else                            // но который не надо удалять
            {
                cout << " and TAIL is a wrong atom." << endl;

                del_atoms_Add(head(s), atom, depth + 1, 1);
            }
        }

        else                        // и хвост тоже пара указателей
        {
            cout << " and TAIL is a \'pair\' too." << endl;

            del_atoms_Add(head(s), atom, depth + 1, 1);
            del_atoms_Add(tail(s), atom, depth + 1, 2);
        }
```

```
    }
}
```

headers.h :

```
#ifndef HEADERS_H
#define HEADERS_H

#include <iostream>
#include <fstream>

typedef char base; // базовый тип элементов (атомов)

struct s_expr;

struct two_ptr // содержит голову и хвост
{
    s_expr *head;
    s_expr *tale;
};

struct s_expr // голова и хвост хранят либо атом, либо пару (рекурсия) и
соответствующее значение в tag
{
    bool tag; // true: atom, false: pair
    int no_brackets = 0; // добавлено, чтобы при желании можно было не выводить
скобки
    union
    {
        base atom; // 1 символ
        two_ptr pair; // голова и хвост
    } node;
};
typedef s_expr *lisp;

// объявления основных функций
lisp head (const lisp s);
lisp tail (const lisp s);
lisp cons (const lisp h, const lisp t);
lisp make_atom (const base x);
lisp copy_lisp (const lisp x);
void destroy (lisp s);

// объявления предикатов
```

```
bool isAtom (const lisp s);
bool isNull (const lisp s);

// объявления функций вывода
void write_lisp (const lisp x); // основная
void write_seq (const lisp x);

// объявления функций считывания
void read_lisp (lisp& y); // основная
void read_s_expr (base prev, lisp& y);
void read_seq (lisp& y);

base getAtom (const lisp s);


void print_depth(int depth);

void free_cin(); // чтобы в cin ничего не оставалось при неправильном вводе.

int del_needed_atoms(lisp s, base atom, int depth);

int del_atoms_Add(lisp s, base atom, int depth, int loc);


#endif
```