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. 9382 _____ Демин В.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Ознакомиться с базовыми рекурсивными функциями обработки иерархических и линейных списков на языке программирования C++.

Задание.

Вариант 11.

Сформировать линейный список атомов исходного иерархического списка таким образом, что скобочная запись полученного линейного списка будет совпадать с сокращённой скобочной записью исходного иерархического списка после устранения всех внутренних скобок

Основные теоретические сведения.

Традиционно иерархические списки представляют или графически или в виде скобочной записи. На рисунке 1 приведен пример графического изображения иерархического списка. Соответствующая этому изображению сокращенная скобочная запись – это (a (b c) d e).

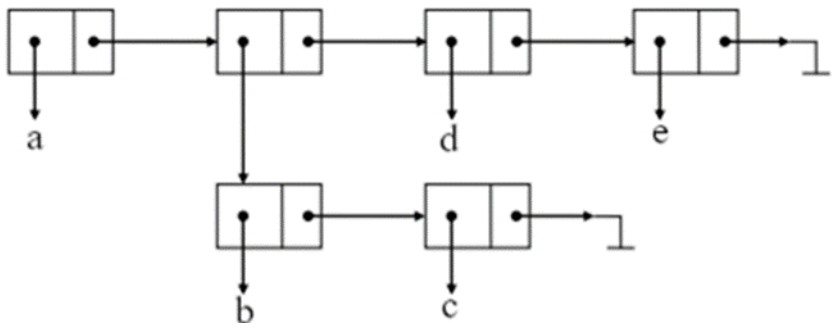

рис. 1

Алгоритм.

1. Для выполнения задания изначально необходимо получить иерархический список из которого мы будем получать линейный.
    a. Считываем символ, и производим проверку:
        i. если символ == ‘)’, то выводим ошибко записи списка
        ii. если символ != ‘(‘, то введен один атом списка.
        iii. если первый символ == ‘(‘, то переходим к пункту b

b. Если символ был ‘(‘, то считываем следующий символ, который не равен пробелу.
    i. Если следующий символ “)” , то возвращаем пустой список.
    ii. Иначе переходим к пункту a, работая уже с пустым списком p1, после этого переходи к пункту b, работая с пустым списком p2. Далее склеиваем в один список p1 и p2. (как видим происходит рекурсивный вызов функций)
c. Таким образом будет происходить склеивание атомов с другими сигментам в скобках.

2. Далее после того мы считали иерархический список, мы можем его как-нибудь изменить, если требуется. После этого чтобы получить список нам нужен сокращенная запись иерархического списка.
    a. Создадим пустую строку, в которую будем записывать сокращенную запись списка.
    b. Далее работаем со списком:
        i. если начало списка является атомом, то записываем его в строку.
        ii. если он не атом, то добаляем “(“ к строке, и работаем со списком по пункту c. После чего добавляем к строке “)”
        iii. если список пустой то добавляем “ ()”.
    c. Рассматриваем список, если он не пустой, то переходим пунку b, работая с head списка. Далее переходим к пункту c, работая с tail списка.
    d. Таким образом, будут рекурсивно вызваны функции получения списка или сигмента списка.
3. Получив скобочную запись спика, будет достаточно удалить все скобки в строке, и связать атомы в линейный список.

Функции и структуры данных.

1. Линейный список

a. Struct elem{ char atom, elem* next, *prev} – структура элемента списка, которая хранит в себе атом и указатели на следующий и предыдущий элемент списка.
b. struct list {elem *Head, *Tail; int Count;} – структура списка хранит указатели на начало и хвост списка.
c. list(); - конструктор списка
d. list(const list &); - конструктор копирование
e. ~list() – деструктор списка
f. elem *getElem(int); - функция получения элемента по порядковому номеру
g. void delAll(); - функция, которая удаляет все элементы списка
h. void del(int); - функция, которая удаляет элемент по позиции в списке
i. void addTail(char n); - добавление в конец списка
j. void addHead(char n); - Добавление в начало списка
k. void print(); -/ Печать списка
l. void print(int pos); - Печать определенного элемента
m. void makeListByAbbreviatedParenthesis(basic_string<char> str);- создание списка из сокращенной записи иерархического списка

2. Иерархический список
    a. struct two_ptr { s_expr *hd; s_expr *tl; }; -структура, которая хранит в себе два указателя s_expr – два элемента списка
    b. struct s_expr {bool tag; union { base atom; two_ptr pair;} node; - элемент иерархического списка, который в зависимости от tag будет хранить в себе атом, или pair указателей.
    c. typedef s_expr *lisp; - замена указателя на элемент как lisp.
    d. lisp head(const lisp s); - функция которая возращает head структуры two_ptr элемента списка, если элемент списка не является атомом.
    e. lisp tail(const lisp s); - функция которая возращает tail структуры two_ptr элемента списка, если элемент списка не является атомом.

f. lisp cons(const lisp h, const lisp t); - функция которая соеденяет два списка в один

g. lisp makeAtom(const base x); - функция, которая создает элемента списка как атом

h. bool isAtom(const lisp s); - проверяет является ли элемент списка атомом

i. bool isNull(const lisp s); - проверяет является ли элемент списка пустым

j. void destroy(lisp s); - рекурсивная функция, которая очищает иерархический список

k. base getAtom(const lisp s); - функция, которая возвращает атом элемента списка

l. void readLisp(lisp &y); - функция считываения списка из файла или строки, вызывает рекурсивную функцию readSExpr.

m. void readSExpr(base prev, lisp &y, std::ifstream &file); - функция считывания элемента списка

n. void readSeq(lisp &y, std::ifstream &file); -функция считывания сигмента списка

o. lisp copyLisp(const lisp x); - функция копирования списка

p. string getLisp(const lisp x, string &str);

void getSeg(lisp const x, string &str); - рекурсивные функции для получения иерархического списка в сокращенной скобочной записи

Выводы.

В данной задаче были изучены основные методы получение сокращенных записей списка через рекурсивные функции.

Тестирование.

№ п/п	Входные данные	Выходные данные	Комментарии
1.	)	! list2.Error 1	
2.	ab	error: selection of output type	
3.	((a))	(a)	
4.	(a)	(a)	
5.	(a (b c) d e)	(a b c d e)	
6.	(a (() (b c) d) e)	(a b c d e)	
7.	(a(b(c(d(e(f(g(h))))))))	(a b c d e f g h)	

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

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

using namespace std;
namespace HList {
    typedef char base;    // базовый тип элементов (атомов)

    struct s_expr;
    struct two_ptr {
        s_expr *hd;
        s_expr *tl;
    };    //end two_ptr;

    struct s_expr {
        bool tag; // true: elem, false: pair
        union {
            base atom;
            two_ptr pair;
        } node;        //end union node
    };            //end s_expr

    typedef s_expr *lisp;

// функции

    // базовые функции:
    lisp head(const lisp s);

    lisp tail(const lisp s);

    lisp cons(const lisp h, const lisp t);

    lisp makeAtom(const base x);

    bool isAtom(const lisp s);
```

```
bool isNull(const lisp s);

void destroy(lisp s);

base getAtom(const lisp s);

// функции ввода:
void readLisp(lisp &y);            // основная
void readSExpr(base prev, lisp &y, std::ifstream &file);

void readSeq(lisp &y, std::ifstream &file);

// функции вывода:
lisp copyLisp(const lisp x);

string getLisp(const lisp x, string &str);        // основная

void getSeg(lisp const x, string &str);

lisp head(const lisp s) {// PreCondition: not null (s)
    if (s != nullptr)
        if (!isAtom(s)) return s->node.pair.hd;
        else {
            cerr << "Error: Head(elem) \n";
            exit(1);
        }
    else {
        cerr << "Error: Head(nil) \n";
        exit(1);
    }
}

bool isAtom(const lisp s) {
    if (s == nullptr) return false;
    else return (s->tag);
}
```

```
bool isNull(const lisp s) {
    return s == nullptr;
}

lisp tail(const lisp s) {// PreCondition: not null (s)
    if (s != nullptr)
        if (!isAtom(s)) return s->node.pair.tl;
        else {
            cerr << "Error: Tail(elem) \n";
            exit(1);
        }
    else {
        cerr << "Error: Tail(nil) \n";
        exit(1);
    }
}

lisp cons(const lisp h, const lisp t)
{
    lisp p;
    if (isAtom(t)) {
        cerr << "Error: Cons(*, elem)\n";
        exit(1);
    } else {
        p = new s_expr;
        if (p == nullptr) {
            cerr << "Memory not enough\n";
            exit(1);
        } else {
            p->tag = false;
            p->node.pair.hd = h;
            p->node.pair.tl = t;
            return p;
        }
    }
}

lisp makeAtom(const base x) {
```

```
    lisp s;
    s = new s_expr;
    s->tag = true;
    s->node.atom = x;
    return s;
}

void destroy(lisp s) {
    if (s != nullptr) {
        if (!isAtom(s)) {
            destroy(head(s));
            destroy(tail(s));
        }
        delete s;
        // s = NULL;
    };
}

base getAtom(const lisp s) {
    if (!isAtom(s)) {
        cerr << "Error: getAtom(s) for !isAtom(s) \n";
        exit(1);
    } else return (s->node.atom);
}

lisp copyLisp(const lisp x) {
    if (isNull(x)) return nullptr;
    else if (isAtom(x)) return makeAtom(x->node.atom);
    else return cons(copyLisp(head(x)), copyLisp(tail(x)));
} //end copy-lisp

void readLisp(lisp &y) {
    int type = 0;
    cout << "Enter from console -1\n";
    cout << "Enter from file -2\n";
    cin >> type;
    ifstream file;
    string name_f;
```

```
        base x;


        switch (type) {
            case 1:
                do cin >> x;
                while (x == ' ');
                readSExpr(x, y, file);
                break;
            case 2:
                cout << "Enter file name\n";
                cin >> name_f;
                file.open(name_f);
                if (file.is_open()) {
                    do file >> x; while (x == ' ');
                    readSExpr(x, y, file);
                    file.close();

                } else cout << "Unable to open file";

                break;
            default:
                cout << "error";
                break;
        }
    } //end read_lisp

    void readSExpr(base prev, lisp &y, ifstream &file) { //prev -
ранее прочитанный символ}
        if (prev == ')') {
            cerr << " ! list2.Error 1 " << endl;
            exit(1);
        } else if (prev != '(') y = makeAtom(prev);
        else readSeq(y, file);
    } //end readSExpr

    void readSeq(lisp &y, ifstream &file) {
        base x;
```

```
            lisp p1, p2;
            if (file.is_open()) {
                if (!(file >> x)) {
                    cerr << " ! list2.Error 2 " << endl;
                    exit(1);
                } else {
                    while (x == ' ') file >> x;
                    if (x == ')') y = nullptr;
                    else {
                        readSExpr(x, p1, file);
                        readSeq(p2, file);
                        y = cons(p1, p2);
                    }
                }
            } else {
                if (!(cin >> x)) {
                    cerr << " ! list2.Error 2 " << endl;
                    exit(1);
                } else {
                    while (x == ' ') cin >> x;
                    if (x == ')') y = nullptr;
                    else {
                        readSExpr(x, p1, file);
                        readSeq(p2, file);
                        y = cons(p1, p2);
                    }
                }
            }
        } //end readSeq


        string getLisp(const lisp x, string &str) {//пустой список
выводится как ()

            if (isNull(x)) str.append(" ()");
            else if (isAtom(x)) {
                str.append(" ");
                str.append(1, x->node.atom);
```

```
            } else { //непустой список}
                str.append(" (");
                getSeg(x, str);
                str.append(" )");
            }
            return str;
        } // end write_lisp

        void getSeg(const lisp x, string &str) {//выводит
последовательность элементов списка без обрамляющих его скобок
            if (!isNull(x)) {
                getLisp(head(x), str);
                getSeg(tail(x), str);
            }
        }
    }
    namespace list2 {
        struct elem {
            char atom; // данные
            elem *next, *prev;
        };

        struct list {
            // Голова, хвост
            elem *Head, *Tail;
            // Количество элементов
            int Count;

            // Конструктор
            list();

            // Конструктор копирования
            list(const list &);

            // Деструктор
            ~list();

            // Получить элемент списка
```

```
        elem *getElem(int);

        // Удалить весь список
        void delAll();

        // Удаление элемента, если параметр не указывается,
        // то функция его запрашивает
        void del(int pos = 0);

        // Добавление в конец списка
        void addTail(char n);

        // Добавление в начало списка
        void addHead(char n);

        // Печать списка
        void print();

        // Печать определенного элемента
        void print(int pos);

        // Создание списка из сокращенной записи списка
        void    makeListByAbbreviatedParenthesis(basic_string<char>
str);
    };

    list::list() {
        // Изначально список пуст
        Head = Tail = nullptr;
        Count = 0;
    }

    list::list(const list &L) {
        Head = Tail = nullptr;
        Count = 0;

        // Голова списка, из которого копируем
        elem *temp = L.Head;
```

```
    // Пока не конец списка
    while (temp != nullptr) {
        // Передираем данные
        addTail(temp->atom);
        temp = temp->next;
    }
}

list::~list() {
    // Удаляем все элементы
    delAll();
}

void list::addHead(char n) {
    // новый элемент
    elem *temp = new elem;

    // Предыдущего нет
    temp->prev = nullptr;
    // Заполняем данные
    temp->atom = n;
    // Следующий - бывшая голова
    temp->next = Head;

    // Если элементы есть?
    if (Head != nullptr)
        Head->prev = temp;

    // Если элемент первый, то он одновременно и голова и хвост
    if (Count == 0)
        Head = Tail = temp;
    else
        // иначе новый элемент - головной
        Head = temp;

    Count++;
}
```

```
void list::addTail(char n) {
    // Создаем новый элемент
    elem *temp = new elem;
    // Следующего нет
    temp->next = nullptr;
    // Заполняем данные
    temp->atom = n;
    // Предыдущий - бывший хвост
    temp->prev = Tail;

    // Если элементы есть?
    if (Tail != nullptr)
        Tail->next = temp;

    // Если элемент первый, то он одновременно и голова и хвост
    if (Count == 0)
        Head = Tail = temp;
    else
        // иначе новый элемент - хвостовой
        Tail = temp;

    Count++;
}

void list::del(int pos) {
    // если параметр отсутствует или равен 0, то запрашиваем
его
    if (pos == 0) {
        cout << "Input position: ";
        cin >> pos;
    }
    // Позиция от 1 до Count?
    if (pos < 1 || pos > Count) {
        // Неверная позиция
        cout << "Incorrect position !!!\n";
        return;
    }
```

```
    // Счетчик
    int i = 1;

    elem *Del = Head;

    while (i < pos) {
        // Доходим до элемента,
        // который удаляется
        Del = Del->next;
        i++;
    }

    // Доходим до элемента,
    // который предшествует удаляемому
    elem *PrevDel = Del->prev;
    // Доходим до элемента, который следует за удаляемым
    elem *AfterDel = Del->next;

    // Если удаляем не голову
    if (PrevDel != nullptr && Count != 1)
        PrevDel->next = AfterDel;
    // Если удаляем не хвост
    if (AfterDel != nullptr && Count != 1)
        AfterDel->prev = PrevDel;

    // Удаляются крайние?
    if (pos == 1)
        Head = AfterDel;
    if (pos == Count)
        Tail = PrevDel;

    // Удаление элемента
    delete Del;

    Count--;
}

void list::print(int pos) {
```

```
    // Позиция от 1 до Count?
    if (pos < 1 || pos > Count) {
        // Неверная позиция
        cout << "Incorrect position !!!\n";
        return;
    }

    elem *temp;

    // Определяем с какой стороны
    // быстрее двигаться
    if (pos <= Count / 2) {
        // Отсчет с головы
        temp = Head;
        int i = 1;

        while (i < pos) {
            // Двигаемся до нужного элемента
            temp = temp->next;
            i++;
        }
    } else {
        // Отсчет с хвоста
        temp = Tail;
        int i = 1;

        while (i <= Count - pos) {
            // Двигаемся до нужного элемента
            temp = temp->prev;
            i++;
        }
    }
    // Вывод элемента
    cout << pos << " element: ";
    cout << temp->atom << endl;
}

void list::print() {
```

```
            // Если в списке присутствуют элементы, то пробегаем по
нему
            // и печатаем элементы, начиная с головного
            int type = 0;
            cout << "Print to console -1\n";
            cout << "Print to file -2\n";
            cin >> type;

            ofstream file;
            string name_f;


            switch (type) {
                case 1:
                    if (Count != 0) {
                        elem *temp = Head;
                        cout << "( ";
                        while (temp->next != nullptr) {
                            cout << temp->atom << " ";
                            temp = temp->next;
                        }

                        cout << temp->atom << " )\n";
                    }
                    break;
                case 2:
                    cout << "Enter file name\n";
                    cin >> name_f;
                    file.open(name_f);
                    if (file.is_open()) {
                        if (Count != 0) {
                            elem *temp = Head;
                            file << "( ";
                            while (temp->next != nullptr) {
                                file << temp->atom << " ";
                                temp = temp->next;
                            }
                            file << temp->atom << " )\n";
```

```
                        }
                        cout << "the final list is written to the file
\"" + name_f << "\"" << endl;
                    } else cout << "Unable to open file";
                    break;
                default:
                    cout << "error: selection of output type";
                    break;
            }

        }

        void list::delAll() {
            // Пока остаются элементы, удаляем по одному с головы
            while (Count != 0)
                del(1);
        }

        elem *list::getElem(int pos) {
            elem *temp = Head;

            // Позиция от 1 до Count?
            if (pos < 1 || pos > Count) {
                // Неверная позиция
                cout << "Incorrect position !!!\n";
                return nullptr;
            }

            int i = 1;
            // Ищем нужный нам элемент
            while (i < pos && temp != nullptr) {
                temp = temp->next;
                i++;
            }

            if (temp == nullptr)
                return nullptr;
            else
```

```
                return temp;
        }

        void   list::makeListByAbbreviatedParenthesis(basic_string<char>
str) {
            this->delAll();
            for (int i = 0; i < str.size(); ++i) {
                if (str[i] == '(' || str[i] == ')' || str[i] == ' ' ||
str[i] == ',') {
                    continue;
                }
                this->addTail(str[i]);
            }
        }
    }
    using namespace HList;
    using namespace list2;

    // Тестовый пример
    int main() {

        lisp h;
        readLisp(h);
        list l=list();

        string str;

        l.makeListByAbbreviatedParenthesis(getLisp(h, str));
        l.print();
    }
```