**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. 6383 |  | Спас А.А. |
| Преподаватель |  | Шолохова О.М. |

Санкт-Петербург

2017

**Цель работы.**

Ознакомиться с одной из часто используемых на практике нелинейных конструкций, способами её организации и рекурсивной обработки. Получить навыки решения задач обработки иерархических списков, как с использованием базовых функций их рекурсивной обработки, так и без использования рекурсии.

1. **Постановка задачи.**

Задание предполагает самостоятельную разработку студентом одного или нескольких модулей на языке С++, реализующих согласованный с преподавателем набор операций над иерархическими списками, а также главной программы, непосредственно решающей поставленную задачу. Предполагается выполнение задания в двух вариантах: с использованием базовых функций рекурсивной обработки иерархических списков и без использования рекурсии.

Во всех случаях, когда это не оговорено особо, предполагается, что исходные и результирующие списки размещаются в файлах подходящего типа. Для представления иерархических списков рекомендуется использовать сокращенную скобочную запись.

Вариант 2.

Удалить в иерархическом списке все вхождения заданного элемента (атома) x;

1. **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

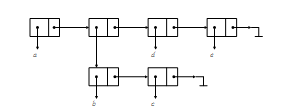
< Non\_null\_list(El) > ::= < Pair(El) >

< Pair(El) > ::= ( < Head\_l(El) > . < Tail\_l(El) > )

< Head\_l(El) > ::= < El >

< Tail\_l(El) > ::= < L\_list(El) >

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



*Рисунок 1. Пример представления иерархического списка в виде двумерного рисунка*

Переход от полной скобочной записи, соответствующей определению иерархического списка, к сокращенной производится путем отбрасывания конструкции . Nill и удаления необходимое число раз пары скобок вместе с предшествующей открывающей скобке точкой. Согласно приведенному определению иерархического списка, структура непустого иерархического списка － это элемент размеченного объединения множества атомов и множества пар «голова-хвост».

1. **Спецификация программы.**

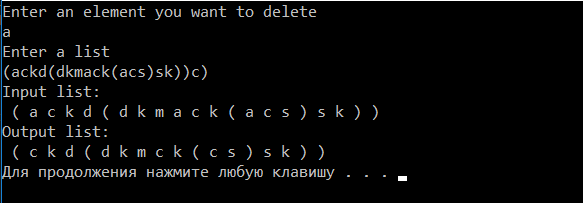
Программа предназначена для удаления всех вхождений заданного атома.

Входными данными для программы являются:

элемент типа char – атом, который требуется удалить;

иерархический список в сокращенной скобочной записи.

1. **Пример диалога с пользователем.**



1. **Реализация.**

Функции:

void delet(lisp l, const base &x);

*Функция, которая удаляет все вхождения атома x*

*предусловие: a – удаляемый атом*

*l – иерархический список в сокращенной скобочной записи*

1. **Тестирование.**

Удаляем а

|  |  |
| --- | --- |
| (a b c b a) | ( b c b ) |
| (b x a h j) | (b x h j) |
| (d s (ae) c d) | (d s (e) c d) |
| (a b) | ( b ) |
| (b a) | ( b ) |
| (c b c) | (c b c) |
| (a) | ( ) |
| (5 7 3) | (5 7 3) |
| (z) | (z) |
| ((A) b (d a) A) | ((A) b (d ) A) |

**Выводы****.**

В результате выполнения лабораторной работы были получены основные сведения о иерархических списках. Ознакомились с базовыми функциями обработки иерархических списков. Была написана функция, позволяющая удалить все вхождения заданного атома.

**ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД**

А.1. l\_intrfc.h

namespace h\_list

{

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: atom, false: pair

union

{

base atom;

two\_ptr pair;

} node; //end union node

}; //end s\_expr

typedef s\_expr \*lisp;

// функции

void print\_s\_expr(lisp s);

// базовые функции:

lisp head(const lisp s);

lisp tail(const lisp s);

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

lisp make\_atom(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 read\_lisp(lisp& y); // основная

void read\_s\_expr(base prev, lisp& y);

void read\_seq(lisp& y);

// функции вывода:

void write\_lisp(const lisp x); // основная

void write\_seq(const lisp x);

lisp copy\_lisp(const lisp x);

}

A.2. l\_intrfc.cpp

#include "l\_intrfc.h"

#include <iostream>

#include <fstream>

#include <cstdlib>

using namespace std;

namespace h\_list

{

//....................................

lisp head(const lisp s)

{// PreCondition: not null (s)

if (s != NULL) if (!isAtom(s)) return s->node.pair.hd;

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

else {

cerr << "Error: Head(nil) \n";

exit(1);

}

}

//.......................................

bool isAtom(const lisp s)

{

if (s == NULL) return false;

else return (s->tag);

}

//.......................................

bool isNull(const lisp s)

{

return s == NULL;

}

//.......................................

lisp tail(const lisp s)

{// PreCondition: not null (s)

if (s != NULL) if (!isAtom(s)) return s->node.pair.tl;

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

else {

cerr << "Error: Tail(nil) \n";

exit(1);

}

}

//.......................................

lisp cons(const lisp h, const lisp t)

// PreCondition: not isAtom (t)

{

lisp p;

if (isAtom(t)) { cerr << "Error: Cons(\*, atom)\n"; exit(1); }

else {

p = new s\_expr;

if (p == NULL) { cerr << "Memory not enough\n"; exit(1); }

else {

p->tag = false;

p->node.pair.hd = h;

p->node.pair.tl = t;

return p;

}

}

}

//...........................

lisp make\_atom(const base x)

{

lisp s;

s = new s\_expr;

s->tag = true;

s->node.atom = x;

return s;

}

//...........................

void destroy(lisp s)

{

if (s != NULL) {

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);

}

//...........................

// ввод списка с консоли

void read\_lisp(lisp& y)

{

base x;

do cin >> x; while (x == ' ');

read\_s\_expr(x, y);

} //end read\_lisp

//...........................

void read\_s\_expr(base prev, lisp& y)

{ //prev － ранее прочитанный символ}

if (prev == ')') { cerr << " ! List.Error 1 " << endl; exit(1); }

else if (prev != '(') y = make\_atom(prev);

else read\_seq(y);

} //end read\_s\_expr

//...........................

void read\_seq(lisp& y)

{

base x;

lisp p1, p2;

if (!(cin >> x)) { cerr << " ! List.Error 2 " << endl; exit(1); }

else {

while (x == ' ') cin >> x;

if (x == ')') y = NULL;

else {

read\_s\_expr(x, p1);

read\_seq(p2);

y = cons(p1, p2);

}

}

} //end read\_seq

//...........................

// Процедура вывода списка с обрамляющими его скобками － write\_lisp,

// а без обрамляющих скобок － write\_seq

void write\_lisp(const lisp x)

{//пустой список выводится как ()

if (isNull(x)) cout << " ()";

else if (isAtom(x)) cout << ' ' << x->node.atom;

else { //непустой список}

cout << " (";

write\_seq(x);

cout << " )";

}

} // end write\_lisp

//...........................

void write\_seq(const lisp x)

{//выводит последовательность элементов списка без обрамляющих его скобок

if (!isNull(x)) {

write\_lisp(head(x));

write\_seq(tail(x));

}

}

//...........................

lisp copy\_lisp(const lisp x)

{

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)));

} //end copy-lisp

}

A.3. main.cpp

#include <iostream>

#include <fstream>

#include "l\_intrfc.h"

using namespace h\_list;

using namespace std;

lisp delet(lisp l, base x);

int main()

{

lisp l = nullptr;

base x;

cout << "Enter an element you want to delete" << endl;

cin >> x;

cout << "Enter a list" << endl;

read\_lisp(l);

cout << "Input list:" << endl;

write\_lisp(l);

cout << endl;

cout << "Output list:" << endl;

l = delet(l, x);

write\_lisp(l);

cout << endl;

destroy(l);

system("pause");

}

lisp delet(lisp l, base x)

{

if (isNull(l)) return nullptr;

if (isAtom(l))

{

if (getAtom(l) != x) return l;

else

{

delete l;

return nullptr;

}

}

l->node.pair.hd = delet(head(l), x);

l->node.pair.tl = delet(tail(l), x);

return ((head(l) == nullptr) ? tail(l) : l);

}