**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 |  | Михеева Е.Е. |
| Преподаватель |  | Шолохова О. М. |

Санкт-Петербург

2017

**Цель работы.**

Познакомиться 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.

**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.

Задание: 3. заменить в иерархическом списке все вхождения заданного элемента (атома) x на заданный элемент (атом) y;

**Основные теоретические положения.**

Традиционно иерархические списки представляют или графически или в виде скобочной записи. На рисунке 2.1 приведен пример графического изображения иерархического списка. Соответствующая этому изображению сокращенная скобочная запись － это (a (b c) d e).

Рис. 2.1. Пример представления иерархического списка в виде двумерного рисунка.

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*) >

Функциональная спецификация иерархического списка включает:

функции － селекторы Head и Tail , выделяющие соответственно «голову» и «хвост» списка

функции － конструкторы: Cons, создающая точечную пару (новый список из «головы» и «хвоста»), и Make\_Atom, создающая атомарное S-выражение.

предикаты Is Null, проверяющий список на отсутствие в нем элементов, и Atom, проверяющий, является ли список атомом.

Функция Destroy, позволяющая уничтожить созданный список, т.е. освободить память от ставших ненужными списочных структур.

**Спецификация программы.**

*Назначение программы*.

Программа предназначается для подсчета количества в иерархическом списке различных атомов.

*Описание программы*.

Программа написана на языке C++ с использованием компилятора gcc. Входными данными для программы является: строка, представляющая список, элемент для замены (символ), заданный элемент (символ). Входные данные вводятся из консоли.

Выходными данными являются:

В случае соответствия исходной строки сокращенной скобочной записи иерархического списка — сокращенная скобочная запись результирующего списка в виде строки, записанной в файле output.txt, расположенном в каталоге с программой. В случае несоответствия — сообщение с кодом ошибки, выведенное на консоль.

*Пример исходных данных*

*(a b b c)*

*Пример выходных данных*

2

(a c)

*Реализация*.

void find(list x, base elem);

Рекурсивные функции для поиска повторяющихся элементов

Предусловие: s — содержит линейный список уникальных элементов.

Предусловие: list — содержит входной список.

Постусловие: list — содержит выходной список.

**Тестирование.**

Во всех тестах в таблице предполагается, что from=“а“, to=“X“.

|  |  |
| --- | --- |
| Входная строка | Ожидаемый результат |
| (a) | 1  (a) |
| (b(aa)) | 1  (ba) |
| (a a c c b(dk)) | 3  (a b c d k ) |

**Выводы.**

В ходе работы я познакомилась с одной из часто используемых на практике нелинейных конструкций, способами её организации и рекурсивной обработки, получила навыки решения задач обработки иерархических списков, как с использованием базовых функций их рекурсивной обработки, так и без использования рекурсии.

Приложение A. Код программы.

Файл main.cpp

// Михеева Екатеринв

// группа 6383

//вариант 4

//09.10.17

//

#include <iostream>

#include <cstdlib>

#include "lisp.h"

#include "list.h"

using namespace std;

void count(list head, int &k) {

if (head != NULL) {

count(tail\_l(head),k);

k++;

}

else return;

}

bool find(list head, base c){

if (head != nullptr) {

if (head->value == c)

return true;

else return find(tail\_l(head), c);

}

else return false;

}

list make\_list(lisp s1);

void make\_list\_private( lisp s1, list &x);

void make\_list\_private\_seq( lisp s1, list &x);

int main(){

lisp s1;

int k = 0;

cout << "Введите list1:" << endl;

read\_lisp(s1);

cout << endl;

list head;

head = make\_list(s1);

cout << "flatten = ";

cout<<"( ";

write\_l((head));

cout<<")"<<endl;

count(head, k);

cout<<"Количество = ";

cout << k << endl;

destroy\_list(head);

destroy\_lisp(s1);

return 0;

}

void make\_list\_private ( lisp s1, list &x){

if (is\_null(s1)) return ;

if (is\_atom(s1)) {

if (!find(x, s1->node.atom))

add(x,s1->node.atom);

}

else make\_list\_private\_seq(s1,x);

}

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

void make\_list\_private\_seq ( lisp s1,list &x){

if (!is\_null(s1)) {

make\_list\_private(head (s1),x);

make\_list\_private\_seq(tail (s1),x);

}

}

list make\_list(lisp s1){

list x = nullptr;

make\_list\_private(s1,x);

return x;

}

файл lisp.cpp

#include "lisp.h"

#include <iostream>

#include <stdlib.h>

using namespace std;

lisp head (const lisp s){

if (s != NULL) if (!is\_atom(s)) return s->node.pair.hd;

else {

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

exit(1);

}

else {

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

exit(1);

}

}

bool is\_atom (const lisp s){

if(s == NULL) return false;

else return !(s -> tag);

}

bool is\_null (const lisp s){

return s==NULL;

}

lisp tail (const lisp s){

if (s != NULL) if (!is\_atom(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){

lisp p;

if (is\_atom(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 = PAIR;

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 = ATOM;

s->node.atom = x;

return s;

}

base get\_atom (const lisp s){

if (!is\_atom(s)) {

cerr << "Error: getAtom(s) for !isAtom(s) \n";

exit(1);

}

else return (s->node.atom);

}

void read\_lisp (lisp &y){

base x;

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

read\_s\_expr ( x, y);

}

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

if ( prev == ')' ) {

cerr << " ! List.Error 1 " << endl;

exit(1);

}

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

else read\_seq (y);

}

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);

}

}

}

void write\_lisp (const lisp x){

if (is\_null(x)) cout << " ()";

else if (is\_atom(x))

cout << ' ' << x->node.atom;

else {

cout << " (" ;

write\_seq(x);

cout << " )";

}

}

void write\_seq (const lisp x){

if (!is\_null(x)) {

write\_lisp(head (x));

write\_seq(tail (x));

}

}

void destroy\_lisp(lisp s)

{

if ( s != NULL) {

if (!is\_atom(s)) {

destroy\_lisp(head(s));

destroy\_lisp(tail(s));

}

delete s;

};

}

файл lisp.h

#ifndef LISP\_H

#define LISP\_H

enum TAG {

ATOM,

PAIR

};

typedef char base;

struct s\_expr;

struct two\_ptr{

s\_expr \*hd;

s\_expr \*tl;

};

struct s\_expr {

TAG tag;

union{

base atom;

two\_ptr pair;

} node;

};

typedef s\_expr \*lisp;

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 is\_atom (const lisp s);

bool is\_null (const lisp s);

base get\_atom (const lisp s);

void read\_lisp ( lisp& y);

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

void read\_seq ( lisp& y);

void write\_lisp (const lisp x);

void write\_seq (const lisp x);

void destroy\_lisp (lisp s) ;

#endif //

файл list.cpp

#include "list.h"

#include <iostream>

using namespace std;

list tail\_l(list head){

return head->next;

}

void write\_l(list head){

if (head != NULL){

cout << head->value << " ";

write\_l(tail\_l(head));

}

else return ;

}

void add(list &x, char elem){

list node = new Node;

node->value = elem;

node ->next = x;

x = node;

}

void destroy\_list(list x){

if(x!=NULL)

destroy\_list(tail\_l(x));

delete(x);

}

файл list.h

#ifndef LIST

#define LIST

typedef char base;

struct Node {

base value;

Node \*next;

};

typedef Node \*list;

list tail\_l(list head);

void write\_l(list head);

void add(list &x, char elem);

void destroy\_list(list x);

#endif

// LIST