МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

ИНСТИТУТ НЕПРЕРЫВНОГО И ДИСТАНЦИОННОГО ОБРАЗОВАНИЯ

|  |
| --- |
| КАФЕДРА компьютерных технологий и программной инженерии |

ОЦЕНКА

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ |
| «ЛИНЕЙНЫЕ И ЦИКЛИЧЕСКИЕ СПИСКИ» |
| по дисциплине: Структуры и алгоритмы обработки данных |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | Z9431 |  |  |  | Андреев Д.И. |
|  | номер группы |  | подпись, дата |  | инициалы, фамилия |
| Студенческий билет № | 2019/3781 | |  |  |  |

Санкт-Петербург 2020

Оглавление

[1. Цель работы 3](#_Toc53095307)

[2. Вариант задания 4](#_Toc53095308)

[3. Листинг программы 5](#_Toc53095309)

[4. Контрольные примеры 13](#_Toc53095310)

[5. Выводы 14](#_Toc53095311)

1. Цель работы

Целью работы является изучение структур данных «линейный список» и «циклический список», а также получение практических навыков их реализации.

1. Вариант задания

Дана последовательность повторяющихся целых чисел , , … , . Получить последовательность, содержащую среднее арифметическое  элементов исходной последовательности, затем  элементов и т.д. (,, … , , ). Реализовать структуру данных циклический односвязный список (далее по тексту список).

Дополнительно программа должна осуществлять следующие операции:

1) Добавление/удаление элемента в список (с клавиатуры);

2) Вывод исходного и результирующего списков на экран;

1. Листинг программы

Файл main.cpp.

Содержит код пользовательского интерфейса программы и реализацию алгоритма выполнения основной задачи варианта лабораторной работы.

#include "LinkList.h"

#include <iostream>

using namespace std;

//Объявления функций

LinkList<double>::iterator insertElements(LinkList<double>& list, const LinkList<double>::iterator& iter);

void eraseElement(LinkList<double>& list);

void showAll(const LinkList<double>& inList, const LinkList<double>& outList);

void showHint();

short getCommand();

void computeNewList(const LinkList<double> &list, LinkList<double>& outList);

int main() {

setlocale(LC\_ALL, "Russian");

cout << "Лабораторная работа №2.\nЛинейные и циклические списки\nВариант " << (81 % 23) + 1 << "\n\n";

short command = -1;

//входной список

LinkList<double> inputList;

//Итератор на конец списка

LinkList<double>::iterator endInputIter = inputList.begin();

//выходной список

LinkList<double> outputList;

do {

command = getCommand();

switch (command)

{

case 0:

endInputIter = insertElements(inputList, endInputIter);

break;

case 1:

eraseElement(inputList);

break;

case 2:

computeNewList(inputList, outputList);

break;

case 3:

showAll(inputList, outputList);

break;

case 4:

showHint();

break;

case 5:

break;

default:

cout << "Неизвестная команда\n";

break;

}

} while (command != 5);

return 0;

}

//Объявления функций

//Добавление элементов в список

//Добавляет елемент введенный пользователем по итератору inputIterator в список list

//Возвращает итератор на вставленный элемент

//Символы, не являющиеся числами будут игнорироваться

LinkList<double>::iterator insertElements(LinkList<double>& list, const LinkList<double>::iterator& iter)

{

double elem;

LinkList<double>::iterator tempIter = iter;

cout << "Введите элементы для добавления в список: ";

while(cin.peek() != '\n') {

cin >> elem;

if (!cin) {

cin.clear();

cin.ignore(sizeof(char));

continue;

}

tempIter = list.insert(tempIter, elem);

}

return tempIter;

}

//Удаление элементов из списка

//Удаляет элемент введенный пользователем из списка

void eraseElement(LinkList<double>& list)

{

if (list.isEmpty()) {

cout << "Список пуст. Нучего удалять\n";

return;

}

int count = 0;

double elem;

cout << "Введите элементы для удаления: ";

while (cin.peek() != '\n') {

cin >> elem;

if (!cin) {

cin.clear();

cin.ignore(sizeof(char));

continue;

}

LinkList<double>::iterator it = list.find(elem);

if (elem == \*it) {

list.erase(it);

++count;

}

}

cout << "Удалено " << count << " элементов\n";

}

//Вывод начального и конечного списков на экран

void showAll(const LinkList<double> &inList, const LinkList<double> &outList)

{

cout << "Входной список:\n";

display\_all(inList);

cout << "Выходной список:\n";

display\_all(outList);

}

//Вывод подсказки

void showHint()

{

cout << "0 - добавить элементы в список\n";

cout << "1 - удалить элемент из входного списка\n";

cout << "2 - создать новый список\n";

cout << "3 - вывести списки на экран\n";

cout << "4 - вывести подсказку\n";

cout << "5 - выход" << endl;

}

//Чтение команды введеной пользователем

//Читает и возвращает введенную пользователем команду

short getCommand()

{

short com;

cout << "Введите команду для продолжения (4 - показать все команды): ";

cin >> com;

while (!cin) {

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

cout << "Введите коректной значение (5 для выхода):";

cin >> com;

}

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

return com;

}

//Создать выходной список

//Генерирует выходной список newList из выходного списка oldList в соответствии с заданием

void computeNewList(const LinkList<double>& oldList, LinkList<double>& newList)

{

//Очистка выходного списка, если он не пуст

newList.clear();

//Глубокое копирование входного списка, чтобы сохранить его целостность

LinkList<double> li = oldList;

LinkList<double>::iterator it = li.begin();

LinkList<double>::iterator outIt = newList.begin();

//Пока входной список не пуст

while (!li.isEmpty()) {

//Вставка нового элемента в выходной список

outIt = newList.insert(outIt, \*it++);

//Пока не закончится обход входного списка

while (li.begin() != it->next()) {

//Прибавление к созданному в выходном списке элементу всех

//значений входного списка, кроме последнего

\*outIt += \*it++;

}

//Если во входном списке 1 элемент,

//необходимости в еще одном добавлении и делении нет

if (li.size() > 1) {

//Добавить последний элемент из входного списка

//к созданному элементу в выходном списке

\*outIt += \*it;

//Деление полученного результата на количество

//элементов входного списка (получение среднего арифметического)

\*outIt /= li.size();

}

//Удаление последнего элемента из входного списка и переход к его началу

it = li.erase(it);

}

}

Файл LinkList.h.

Содержит реализацию алгоритмов двунаправленного связанного списка, паттерна итератор и класса хранения данных.

#pragma once

#include <iostream>

template<typename T> class LinkList;

template<typename Node> class ListIterator;

//Шаблонный класс для хранения элементов и их указателей

template<typename T>

class node

{

public:

using Value = typename T;

node() : m\_data(), m\_next(this) {}

node(Value\* data) : m\_data(data) { m\_next = this; }

node(Value\* data, node\* next) : m\_data(data), m\_next(next) {}

~node() { delete m\_data; }

//функции получения значений

//указатуль на следующий элемент в списке

node\* next() const { return m\_next; }

//даные

Value\* data() const { return m\_data; }

//функции установки значений

void setData(Value\* data) { m\_data = data; }

void setNext(node\* next) { m\_next = next; }

private:

node\* m\_next;

Value\* m\_data;

};

//Класс итератора для упращения обхода элементов

template<typename Node>

class ListIterator

{

using ValueType = typename Node::Value;

public:

ListIterator(Node\* data = nullptr) : m\_ptr(data) {}

//Перегруженные операторы

//operator Node() { return m\_ptr; }

ValueType& operator\*() { return \*m\_ptr->data(); }

Node\* operator->() { return m\_ptr; }

bool operator==(const ListIterator& rightIter) { return m\_ptr == rightIter.m\_ptr; }

bool operator!=(const ListIterator& rightIter) { return m\_ptr != rightIter.m\_ptr; }

bool operator==(Node\* rightPtr) { return m\_ptr == rightPtr; }

bool operator!=(Node\* rightPtr) { return m\_ptr != rightPtr; }

bool operator!() { return !m\_ptr; }

operator ValueType\* () { return m\_ptr->data(); }

ListIterator operator++(int)

{

if (m\_ptr == nullptr) return \*this;

ListIterator temp = \*this;

++(\*this);

return temp;

}

ListIterator operator++()

{

if (m\_ptr == nullptr) return \*this;

m\_ptr = m\_ptr->next();

return \*this;

}

//Чтобы не добавлять класс как дружественный в LinkList<T>,

//функция объявлена как public

Node\* \_getPtr() const { return m\_ptr; }

private:

Node\* m\_ptr;

};

//Класс циклического списка

template<typename T>

class LinkList

{

using Node = node<T>;

friend class Node;

public:

using iterator = ListIterator<Node>;

LinkList() : m\_begin(nullptr), m\_size(0) {}

//Конструктор копирования

LinkList(const LinkList<T>& li)

{

LinkList<T>::iterator it = li.begin();

LinkList<T>::iterator thisIt;

thisIt = insert(begin(), \*it++);

while (it != li.begin())

thisIt = insert(thisIt, \*it++);

}

~LinkList()

{

clear();

}

//служебные методы возвращают размер списка,

//проверку на пустоту списка и итератор на начало соответственно

int size() const { return m\_size; }

bool isEmpty() const { return !m\_size; }

iterator begin() const { return iterator(m\_begin); }

LinkList<T>& operator=(const LinkList<T>& li)

{

if (this == &li)

return \*this;

LinkList<T>::iterator it = li.begin();

LinkList<T>::iterator thisIt;

thisIt = insert(begin(), \*it++);

while(it != li.begin())

thisIt = insert(thisIt, \*it++);

return \*this;

}

bool operator==(const LinkList<T>& li)

{

bool isEqual = true;

if (li.size() != size())

return false;

LinkList<T>::iterator it = li.begin();

LinkList<T>::iterator thisIt = begin();

if (\*it++ == \*thisIt++) isEqual = false;

while (it != li.begin() || isEqual != false)

if (\*it++ == \*thisIt++) isEqual = false;

return isEqual;

}

/\*

\* Произовдит вставку в связанный список после элемента,

\* на который указывает итератор pos

\* Возвращает итератор, указывающий на вставленный элемент

\*/

iterator insert(iterator pos, const T& elem)

{

//Проверка недействительность переданного итератора

//Если итератор недействителен и список пуст, то итератор указывает

//на начало списка и необходимо создание первого элемента

//Иначе передан недействительный итератор (функция возвращает

//переданный недействительный итератор)

if (!pos) {

if (!isEmpty())

return pos;

m\_begin = new Node(new T(elem));

++m\_size;

return iterator(m\_begin);

}

//Вставка нового элемена

pos->setNext(new Node(new T(elem), pos->next()));

++m\_size;

return iterator(pos->next());

}

/\*

\* Производит удаление элемента, на который указывает итератор pos из списка

\* Возвращает итератор на элемент после удаляемого

\* Передаваемый в функцию итератор становится недействительным

\*/

iterator erase(iterator pos)

{

//проверяем, является ли элемент первым

//если это так передвигаем указатель на начало на один элемент вперед

if (pos == begin())

m\_begin = m\_begin->next();

Node\* iter = pos.\_getPtr();

//поиск элемент предшествующий данному

while (pos != iter->next())

iter = iter->next();

//Сохранение элемента следующего за данным

Node\* newNext = iter->next()->next();

//Удаляем переданный в функцию элемент

delete iter->next();

//Связываем разорванный список

iter->setNext(newNext);

--m\_size;

return iterator(newNext);

}

//очищает связанный список

void clear()

{

Node\* iter = m\_begin;

while (!isEmpty()) {

if (iter->next() == m\_begin) {

m\_begin = m\_begin->next();

}

Node\* newNext = iter->next()->next();

delete iter->next();

iter->setNext(newNext);

--m\_size;

}

}

/\*

\* Производит поиск элемента elem в связанном списке

\* Возвращает итератор на (нужно чтобы на предидущий) найденный элемент, если элемент elem есть в списке

\* Иначе возвращает итератор на начало списка

\*/

iterator find(const T& elem)

{

//если список пуст поиск не имеет смысла

if (isEmpty())

return begin();

//Если искомый элемент равен первому возвратить его

if (elem == \*m\_begin->data())

return begin();

//Итераторы для обхода списка

iterator beg = begin();

iterator it = beg;

//В цикле с передаваемым значением сравниваются все элементы кроме первого

//Если элемент не найден, возвращается итератор на начало списка

while (++it != beg) {

if(\*it == elem)

return it;

}

return beg;

}

private:

Node\* m\_begin;

int m\_size;

};

//Выводит все элементы в списке начаная с первого в консоль

void display\_all(const LinkList<double>& li)

{

LinkList<double>::iterator it = li.begin();

if (li.isEmpty()) {

std::cout << "list is empty" << std::endl;

return;

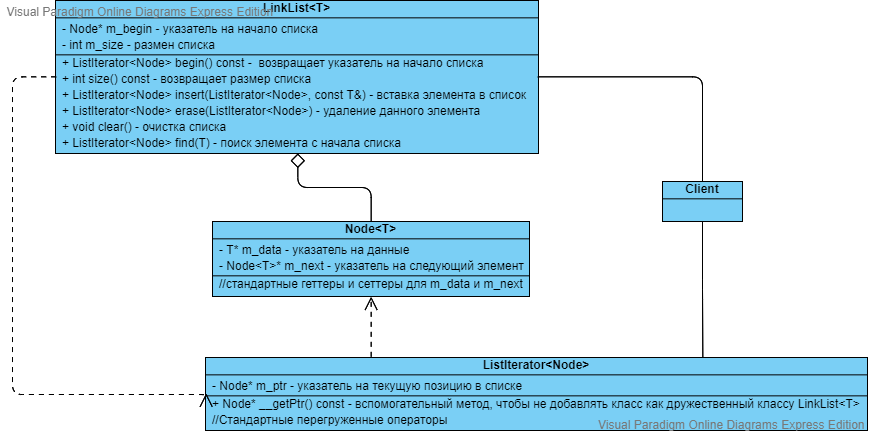
}

for (int i = 0; i < li.size(); ++i) {

std::cout << "element N" << i + 1 << ": " << \*it++ << std::endl;

}

}

Диаграмма классов:

В данной программе реализован паттерн итератор для удобства работы со списком из пользовательского интерфейса.

Описание классов:

Класс Node<T> - шаблонный класс. Реализован как структура для хранения данных списка. Инкапсулирует указатель на данные соответствующий данному узлу списка и указатель на следующий элемент списка. Также содержит стандартные методы get и set для установки и получения соответственно значений указателя на следующий элемент и данных текущего элемента.

Класс ListIterator<Node> - шаблонный класс. Реализует алгоритм обхода списка. Инкапсулирует указатель на текущее звено списка, а также содержит переопределенные методы стандартных операторов, что делает в плане использования его похожим на обычный указатель.

Класс LinkList<T> - шаблонный класс. Реализует алгоритмы списка. Инкапсулирует указатель на начало списка, а также содержит служебные методы get для получения размера списка, указателя на начало списка и проверки списка на отсутствие элементов, метод clear для полной очистки списка (реализован для упрощения кода деструктора) и реализацию алгоритмов добавления, удаления и поиска элементов списка.

Реализация алгоритмов добавления, удаления и поиска элементов:

Алгоритм добавления элемента в список:

iterator insert(iterator pos, const T& elem)

{

//Проверка недействительность переданного итератора

//Если итератор недействителен и список пуст, то итератор указывает

//на начало списка и необходимо создание первого элемента

//Иначе передан недействительный итератор (функция возвращает

//переданный недействительный итератор)

if (!pos) {

if (!isEmpty())

return pos;

m\_begin = new Node(new T(elem));

++m\_size;

return iterator(m\_begin);

}

//Вставка нового элемена

pos->setNext(new Node(new T(elem), pos->next()));

++m\_size;

return iterator(pos->next());

}

Описание:

Входные данные:

pos – итератор, указывающий на позицию в списке, за которой необходимо вставить элемент.

elem – элемент, который необходимо вставить в список.

A1: [проверка указателя переданного в функцию] Если pos – нулевой итератор, перейти к шагу A2, иначе перейти к шагу А5

A2: [проверка пустой ли список] Если список пуст перейти к шагу А3, иначе вставка элемента имеет неопределенное поведение – завершить выполнение алгоритма

A3: [вставка нового элемента в пустой список] Создать новый элемент по адресу на который указывает указатель m\_begin. Завершить выполнение алгоритма

A5: [вставка нового элемента в не пустой список] Создать новый элемент со значением elem и указателем next на элемент следующий за итератором pos и установить указатель next итератора pos на только что созданный элемент. Завершить выполнение алгоритма

Алгоритм удаления элемента из списка:

iterator erase(iterator pos)

{

//проверяем, является ли элемент первым

//если это так передвигаем указатель на начало на один элемент вперед

if (pos == begin())

m\_begin = m\_begin->next();

Node\* iter = pos.\_getPtr();

//поиск элемент предшествующий данному

while (pos != iter->next())

iter = iter->next();

//Сохранение элемента следующего за данным

Node\* newNext = iter->next()->next();

//Удаляем переданный в функцию элемент

delete iter->next();

//Связываем разорванный список

iter->setNext(newNext);

--m\_size;

return iterator(newNext);

}

Описание:

Входные данные:

pos – итератор, указывающий на элемент который необходима удалить

В1: [проверка указателя, переданного в функцию] Если pos – указатель на начало списка перейти к шагу B2, иначе перейти к шагу B3

B2: [перемещение начала списка] переместить указатель на начало списка m\_begin на одну позицию вправо

B3: [сохранение текущего положения] Создать указатель iter, указывающий на тот же элемент что и итератор pos

B4: [поиск элемента следующего за удаляемым] Если указатель next элемента, на который указывает iter, равен итератору pos, перейти к шагу B5, иначе сделать инкремент указателя iter и перейти к шагу B4

B5: [удаление элемента] Удалить элемент на который указывает указатель iter, предварительно сохранив указатель next удаляемого элемента

В6: [восстановление списка] Установить указатель next удаляемого элемента как указатель next элемента на который указывает итератор pos. Завершить выполнение алгоритма

Алгоритм поиска:

iterator find(const T& elem)

{

//если список пуст поиск не имеет смысла

if (isEmpty())

return begin();

//Если искомый элемент равен первому возвратить его

if (elem == \*m\_begin->data())

return begin();

//Итераторы для обхода списка

iterator beg = begin();

iterator it = beg;

//В цикле с передаваемым значением сравниваются все элементы кроме первого

//Если элемент не найден, возвращается итератор на начало списка

while (++it != beg) {

if(\*it == elem)

return it;

}

return beg;

}

Описание:

Входные данные:

elem – значение искомого элемента

C1: [проверка списка] Если список пуст поиск элемента имеет неопределенное поведение. Завершить выполнение алгоритма. Иначе перейти к шагу C2

С2: [проверка переданного элемента] Если элемент elem равен значению элемента в начале списка, то элемент найден. Завершить выполнение алгоритма. Иначе перейти к шагу С3

С3: [создание вспомогательных итераторов] Создать итераторы beg и it, указывающие на начало списка и на элемент следующий за началом списка соответственно

С3: [линейный поиск] если элемент на который указывает it равен переданному элементу, то элемент найден. Завершить выполнение алгоритма. Иначе перейти на шаг C3

Описание алгоритма заданного вариантом лабораторной работы:

void computeNewList(const LinkList<double>& oldList, LinkList<double>& newList)

{

//Очистка выходного списка, если он не пуст

newList.clear();

//Глубокое копирование входного списка, чтобы сохранить его целостность

LinkList<double> li = oldList;

//Итераторы для обхода обоих списков

LinkList<double>::iterator it = li.begin();

LinkList<double>::iterator outIt = newList.begin();

//Пока входной список не пуст

while (!li.isEmpty()) {

//Вставка нового элемента в выходной список

outIt = newList.insert(outIt, \*it++);

//Пока не закончится обход входного списка

while (li.begin() != it->next()) {

//Прибавление к созданному в выходном списке элементу всех значений входного списка, кроме последнего

\*outIt += \*it++;

}

//Если во входном списке 1 элемент, необходимости в еще одном добавлении и делении нет

if (li.size() > 1) {

//Добавить последний элемент из входного списка к созданному элементу в выходном списке

\*outIt += \*it;

//Деление полученного результата на количество элементов входного списка (получение среднего арифметического)

\*outIt /= li.size();

}

it = li.erase(it);

}

}

Описание:

Входные данные:

oldList – исходный список

newList – список для сохранения полученных значений

D1: [очистка списка] Произвести очистку списка newList

D2: [копирование исходного списка] Копировать данные исходного списка во временный список li, создать итераторы it и outIt на начало списков li и newList соответственно

D3: [основной цикл] Если список li не пуст перейти к шагу D4 иначе завершить выполнение алгоритма

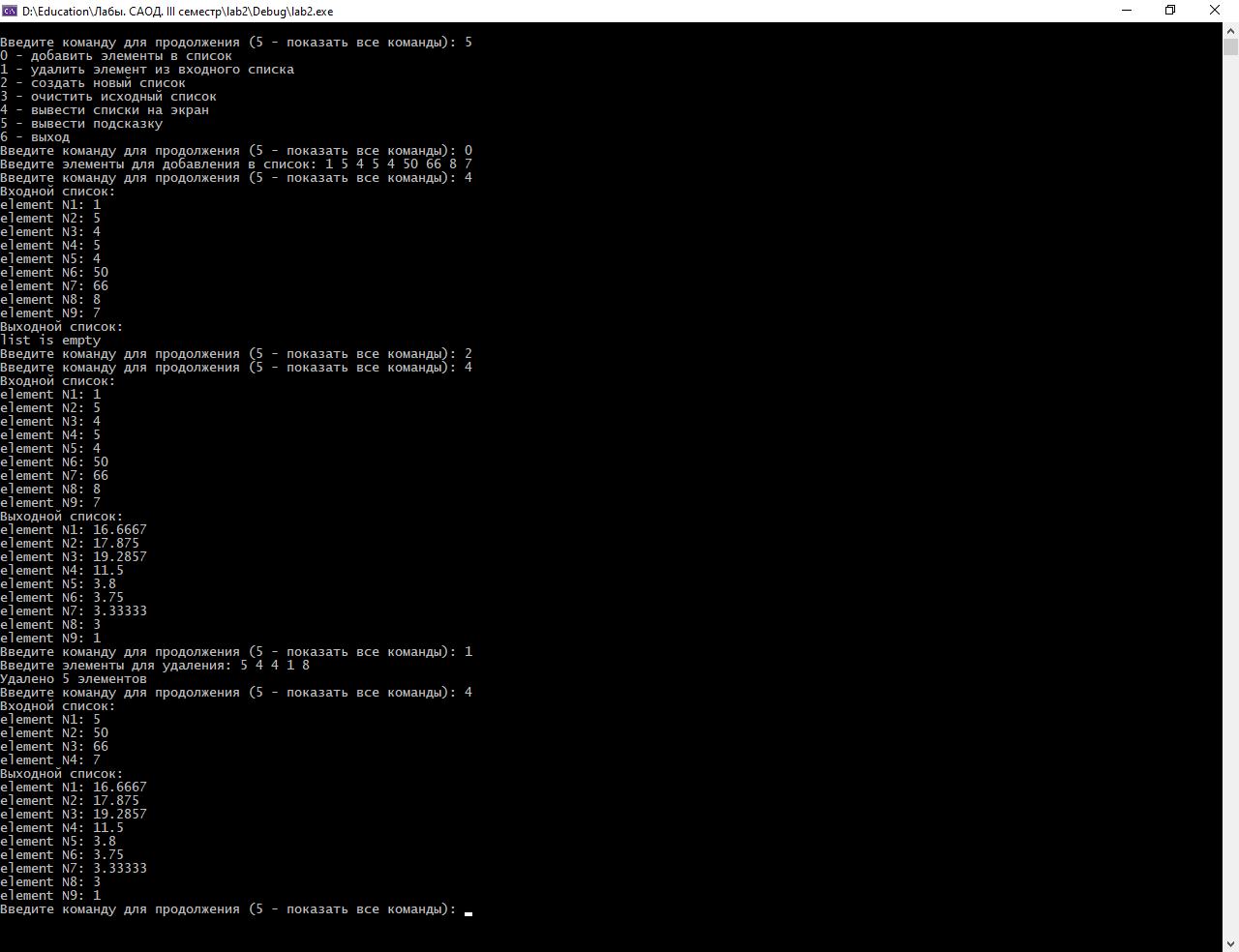
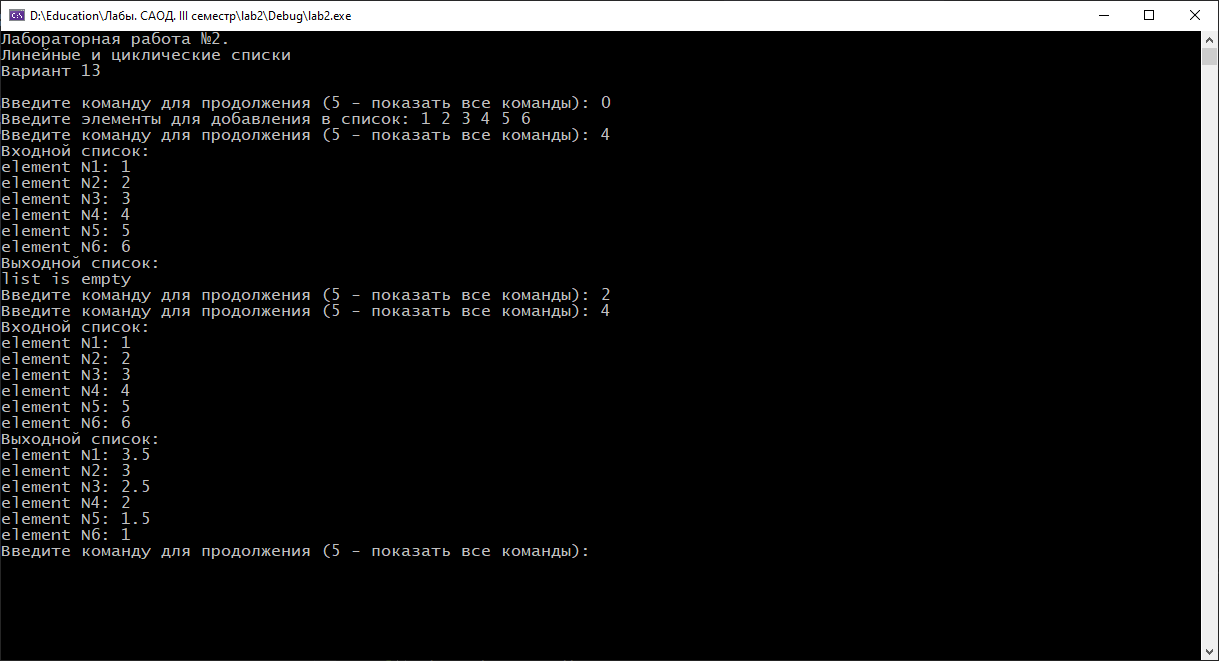
D4: [вставка нового элемента] Вставить элемент по итератору it в список newList в позицию за итератором outIt. Увеличить итератор it на 1. Итератор outIt установить на только что вставленный элемент

D5: [обход исходного списка] Если указатель next итератора it указывает на начало списка li перейти к шагу D7, иначе перейти к шагу D6

D6: [сумма элементов] Прибавить элемент по итератору it к элементу по итератору outIt, увеличить итератор it и перейти к шагу D5

D7: [добавление последнего элемента] Прибавить к элементу по итератору outIt элемент по итератору it и разделить элемент по итератору outIt на размер списка. Сохранить получившееся значение в списке newList в позицию по итератору outIt

D8: [удаление элемента из исходного списка] удалить элемент по итератору it из исходного списка li и перейти к шагу D3

1. Контрольные примеры  
2. Выводы

В рамках лабораторной работы был создан циклический односвязный список с применение паттерна итератор, реализованы алгоритмы вставки и удаления, а также алгоритм линейного поиска в списке.

Как можно заметить, для решения поставленной в лабораторной работе задачи необходимо удаление последнего элемента из входного списка на каждой итерации основного цикла. Данная реализация циклического односвязного списка предполагает удаление элемента, переданного в функцию. В этом случае временная сложность операции удаления O(n), так как для удаления необходимо найти элемент перед удаляемым, чтобы предотвратить разрыв списка. Если при реализации решения данной задачи использовать двусвязный список, то временная сложность операции удаления станет O(1), так как в данном случае каждый элемент списка хранит указатель предыдущий и поиск элемента перед удаляемым более не требуется. Что уменьшает время выполнения алгоритма при большом количестве переменных.

Также для повышения производительности удаления односвязного списка при большом количестве элементов функция удаления может производить удаление элемента после того, который был ей передан. Например:

/\*

\* Производит удаление элемента следующего за тем, на который указывает итератор pos из списка

\* Возвращает итератор на элемент перед удаляемым

\* Передаваемый в функцию итератор становится недействительным

\*/

iterator erase\_after(iterator pos)

{

if (pos->next() == m\_begin) {

m\_begin = m\_begin->next();

}

Node\* iter = pos.\_getPtr();

Node\* newNext = iter->next()->next();

delete iter->next();

iter->setNext(newNext);

--m\_size;

return iterator(iter);

}

В данном случае обход списка не требуется и удаление элемента происходит на первой итерации. Временная сложность данного алгоритма O(1).

Из вышесказанного можно сделать вывод, что в данном случае использование циклического двусвязного списка повышает производительность программы при большом количестве элементов.