**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра АПУ**

отчет

**по лабораторной работе № 3**

**по дисциплине «Программирование»**

**Тема: Задание № 3**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3371 |  | Манешов Д. В. |
| Преподаватель |  | Писарев А.С. |

Санкт-Петербург

2024

## Постановка задачи.

Необходимо реализовать структуру данных односвязного циклического списка List (коллекция, обеспечивающая произвольный доступ к данным) для обработки целочисленной последовательности. Данная структура должна обладать следующим набором функций:

* void add(int) - добавление элемента в конец списка;
* void insert(int, int) - вставка в данную позицию (первый аргумент, индексация от 0) данного значения (второй аргумент);
* void removeAt(int) - удаление из коллекции элемента в заданной позиции (индексация от 0);
* int elementAt(int) - доступ к информационной части элемента в данной позиции (индексация от 0);
* int count() - возвращает количество элементов в коллекции;
* void insertBeforeNegative() - вставка перед каждым отрицательным числом элемента со значением 1;
* void removeNegative() - удаление из списка всех элементов с отрицательной информационной частью;
* int count(int) - подсчет количества вхождений переданного значения в коллекцию;
* clear() - удаление всех элементов коллекции.

Данная структура данных будет подвергнута тестированию автоматическими unit-тестами, поэтому наименование функций (и названия структуры) и сигнатуру изменять не следует.

Так же необходимо предусмотреть реализацию пользовательского интерфейса, обеспечивающего произвольный доступ к функциям Циклического списка.

## Ход решения.

Данная программа реализует структуру данных односвязного циклического списка List (коллекция, обеспечивающая произвольный доступ к данным) для обработки целочисленной последовательности.

В блоке кода (рис. 1.1) подключается библиотека: iostream, она нужна для ввода-вывода данных с консоли. И используется пространство имен std, чтобы не писать std:: перед каждым именем из этого пространства. Пространство имен std содержит множество полезных классов, функций и переменных, таких как cout, cin, ofstream и т.д.



Рисунок 1.1 – Подключение библиотеки и пространство имён

Далее определяется структура Node, которая представляет собой элемент односвязного списка (рис. 1.2). Детальное описание каждой части этой структуры:

1. структура Node:

* базовый блок для построения односвязного списка, где каждый узел содержит данные и указатель на следующий узел в списке;

1. поля структуры:

* int data - целочисленное поле data используется для хранения данных, которые могут быть любым целочисленным значением;
* Node\* next - указатель на следующий элемент списка. Тип Node\* означает, что это указатель на другой объект структуры Node. По умолчанию инициализируется как nullptr, указывая на то, что следующий элемент отсутствует (т.е. текущий узел является последним в списке);

1. конструктор Node (int value):

* это конструктор структуры, который принимает один аргумент целочисленного типа (value);
* внутри конструктора переменной data присваивается значение value, а next устанавливается в nullptr, что означает инициализацию узла как конечного элемента в списке, не имеющего последующих элементов.

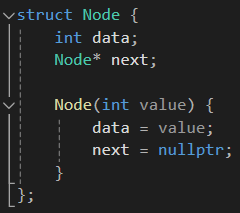


Рисунок 1.2 – Определение структуры узла односвязного списка

В этом блоке кода представлен класс List (рис 1.3), который реализует функционал связного списка с помощью структуры Node. Класс содержит несколько приватных атрибутов и метод для навигации по элементам списка.

Приватные атрибуты:

* Node\* tail - указатель на последний элемент списка; в контексте циклических списков tail->next обычно указывает на первый элемент списка;
* Node\* currentNode - указатель на текущий элемент списка, используемый для доступа к элементам;
* int size - хранит количество элементов в списке;
* int currentIndex - позиция текущего элемента относительно начала списка, используемая в методе getNode.

Метод Node\* getNode(int index) в классе List служит для получения узла по указанному индексу в списке (рис. 1.3). Работа метода начинается с проверки текущего индекса: если запрашиваемый индекс меньше значения currentIndex, происходит переинициализация — currentIndex обнуляется, а currentNode устанавливается в nullptr. Это необходимо для оптимизации доступа, особенно когда требуется получить доступ к элементу, находящемуся ближе к началу списка.

Далее, метод проверяет состояние currentNode. Если этот указатель равен nullptr, что часто бывает после сброса или при первом вызове метода, поиск начинается с элемента, следующего за tail, который является первым элементом списка. Такая организация позволяет методу эффективно переходить к нужной позиции, минимизируя количество необходимых операций.

Затем метод переходит к поиску узла. Он последовательно перемещается по элементам списка, начиная с текущего узла (currentNode), при этом на каждом шаге увеличивается currentIndex. Как только currentIndex достигает запрашиваемого индекса, поиск прекращается, и метод возвращает текущий узел.

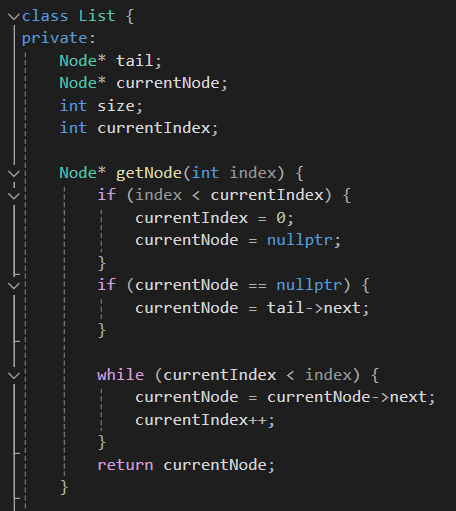


Рисунок 1.3 – Реализация метода доступа к элементу в связном списке

Следующий фрагмент кода (рис. 1.4) содержит реализацию конструктора и деструктора для класса `List`. Конструктор занимается инициализацией основных атрибутов списка: он устанавливает указатели `tail` и `currentNode` в `nullptr`, что сигнализирует о том, что список пуст на начальном этапе, и инициализирует счётчики `size` и `currentIndex` нулевыми значениями, где `size` отражает общее количество элементов в списке, а `currentIndex` используется для отслеживания текущей позиции внутри списка при помощи метода `getNode`. Деструктор класса `List` вызывает метод `clear()`, который предназначен для полной очистки списка, удаляя все его элементы и освобождая занятую ими память, тем самым предотвращая утечки памяти. Это также служит мерой безопасности, поскольку автоматический вызов `clear()` при уничтожении объекта списка гарантирует корректное освобождение всех ресурсов, независимо от того, будет ли список удалён или выйдет из области видимости.

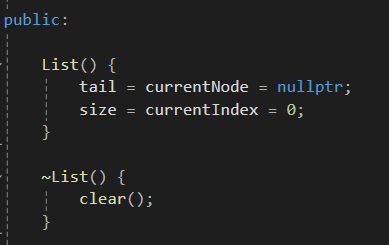


Рисунок 1.4 – Конструктор и деструктор класса List

Метод add (рис. 1.5) представляет собой метод для добавления нового элемента в циклический связный список. При вызове этого метода сначала создается новый узел с помощью конструктора Node, который инициализируется переданным значением value. Если список пуст, что определяется по nullptr в указателе tail, новый узел становится хвостом списка, и его указатель next ссылается на самого себя, формируя цикл. В случае, если в списке уже есть элементы, новый узел вставляется после текущего хвоста, и указатель tail обновляется, чтобы указывать на новый узел, сохраняя цикличность списка. После добавления узла, счетчик размера списка size увеличивается, а переменные currentIndex и currentNode сбрасываются, что необходимо для поддержания корректной работы списка при последующих операциях.

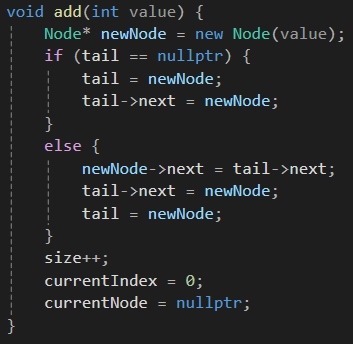


Рисунок 1.5 – Метод для добавления элемента

Метод insert (рис. 1.6) предназначен для вставки нового элемента в произвольную позицию циклического связного списка. Параметр index указывает на позицию, куда должен быть вставлен новый элемент, а value — это значение, которое будет храниться в новом узле списка.

Первым шагом метод корректирует index с учетом текущего размера списка size, используя операцию остатка от деления, чтобы гарантировать, что индекс не выйдет за пределы допустимых значений. Если список пуст, index устанавливается в 0. Если index равен size, то новый элемент добавляется в конец списка с помощью уже существующий метод add.

Затем создается новый узел newNode с помощью конструктора Node, инициализированный значением value. Если index равен 0, новый узел вставляется в начало списка. В противном случае, с помощью вспомогательного метода getNode, находится узел, который в данный момент занимает позицию index - 1, и новый узел вставляется после него.

После вставки нового узла, size увеличивается на единицу, обновляя размер списка. Также обновляются currentIndex и currentNode, которые теперь указывают на позицию и узел, добавленные в список.

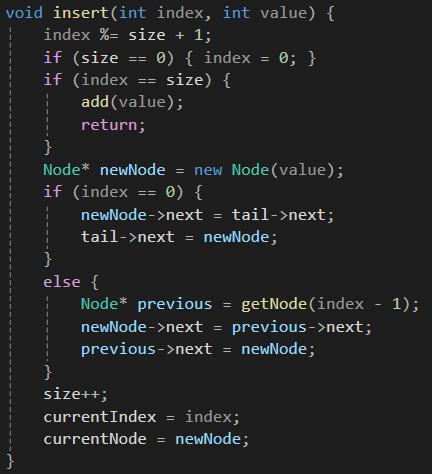


Рисунок 1.6 – Метод для вставки элемента

Метод removeAt (рис. 1.7) удаляет элемент из циклического связного списка по заданному индексу. Если список пуст, что проверяется по указателю tail, метод ничего не делает и возвращает управление. Индекс корректируется с учетом размера списка, чтобы предотвратить выход за его пределы.

Если в списке всего один элемент, он удаляется, и все связанные с ним указатели и счетчики обнуляются. В случае, если индекс равен нулю, удаляется первый элемент списка. Если удаляемый элемент является текущим узлом, на который указывает currentNode, то currentNode и currentIndex обновляются.

Для удаления элемента по индексу, отличному от нуля, сначала находится узел, предшествующий удаляемому. Затем удаляемый узел извлекается из списка, и его память освобождается. Если удаляемый узел является хвостом списка, то указатель tail обновляется на предшествующий узел. После удаления элемента размер списка уменьшается.

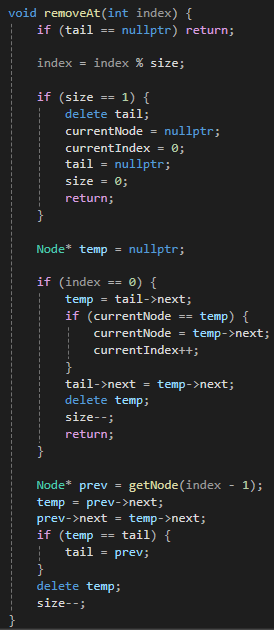


Рисунок 1.7 – Метод для удаления элемента

Метод elementAt (рис. 1.8) служит для извлечения данных из узла циклического связного списка по заданному индексу. Если список пуст, метод возвращает -1, сигнализируя об ошибке. Индекс нормализуется относительно размера списка, чтобы обеспечить доступ к элементам в рамках его границ. С помощью вспомогательного метода getNode происходит поиск узла по индексу. Если узел не найден, возвращается -1. В случае успеха, метод возвращает значение data, содержащееся в узле, что позволяет получить доступ к конкретным данным списка. Этот метод является ключевым для операций чтения данных в связном списке.

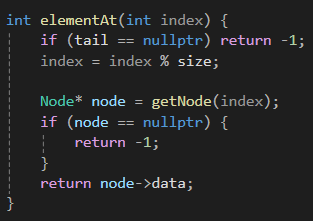


Рисунок 1.8 – Метод для получения данных элемента по индексу

Метод count (рис. 1.9) возвращает текущее количество элементов в циклическом связном списке. Она возвращает значение переменной size, которая отслеживает размер списка.

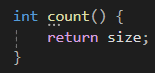


Рисунок 1.9 – Метод подсчета элементов в списке

Метод insertBeforeNegative (рис. 1.10) предназначен для обхода циклического связного списка и вставки нового узла со значением 1 перед каждым узлом, содержащим отрицательное число. Процесс начинается с проверки, не является ли список пустым, что указывало бы на nullptr в tail. Если список не пуст, начинается обход с первого узла списка.

Для каждого узла curr проверяется, содержит ли он отрицательное значение в поле data. Если это так, создается новый узел newNode со значением 1. Если текущий узел является первым узлом списка (следующим за tail), новый узел вставляется в начало списка. В противном случае, новый узел вставляется между prev и curr. После вставки нового узла, размер списка увеличивается на единицу.

Если индекс текущего узла совпадает с currentIndex, то currentNode обновляется на newNode. Если индекс меньше currentIndex, то currentIndex увеличивается на единицу, чтобы отразить изменение позиций узлов в списке. Обход продолжается до тех пор, пока не будет достигнут первый узел списка снова, завершая цикл. Эта метод обеспечивает динамическую модификацию списка, позволяя вставлять новые узлы в соответствии с определенным условием.

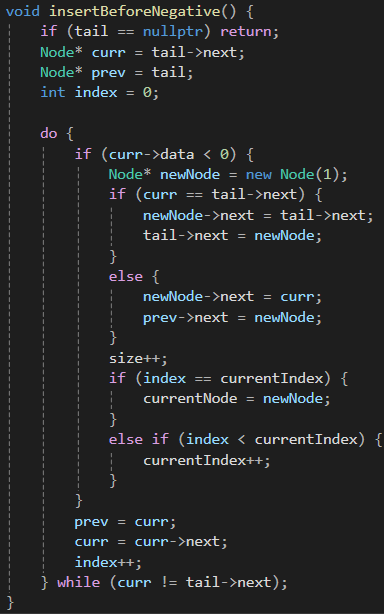


Рисунок 1.10 – Метод для вставки 1 перед отрицательными значениями

Метод removeNegative (рис. 1.11) предназначен для удаления всех узлов с отрицательными значениями из циклического связного списка. Процесс начинается с проверки, не пуст ли список, и, если это так, метод завершается. В противном случае, начинается обход списка с первого узла.

Во время обхода, если текущий узел curr содержит отрицательное значение, он удаляется из списка. Если удаляемый узел является текущим узлом currentNode, то currentNode и currentIndex обновляются. После удаления узла, указатель prev обновляется, чтобы указывать на следующий узел, и размер списка уменьшается.

Если хвост списка tail содержит отрицательное значение, он также удаляется, и tail обновляется на предыдущий узел. Если после удаления всех отрицательных элементов список оказывается пустым, все указатели устанавливаются в nullptr, и размер списка обнуляется.

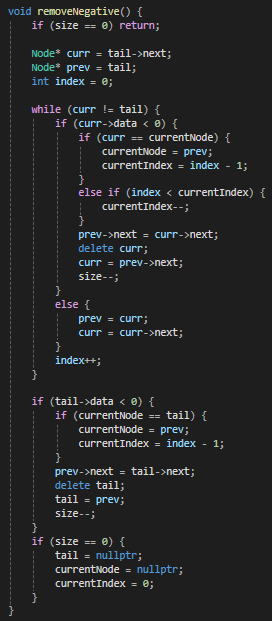


Рисунок 1.11 – Метод для удаления отрицательных элементов

Метод count (рис. 1.12) принимает один параметр value и подсчитывает количество узлов в циклическом связном списке, которые содержат это значение. Счетчик k инициализируется нулем и увеличивается каждый раз, когда значение в узле списка совпадает с value. Обход списка начинается с первого узла и продолжается до тех пор, пока не будут проверены все узлы. Если список пуст, метод немедленно возвращает k, которое в этом случае равно нулю. В конце метод возвращает k, предоставляя общее количество узлов с заданным значением.

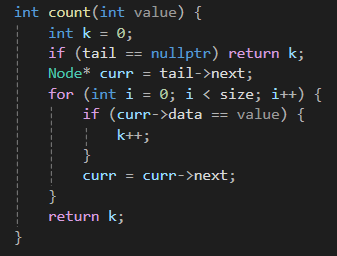


Рисунок 1.12 – Метод для подсчета количества узлов с заданным значением

Метод clear (рис. 1.13) предназначен для полной очистки циклического связного списка, удаляя все его узлы и сбрасывая все связанные с ним управляющие переменные. Если список уже пуст, что определяется по nullptr в tail, метод ничего не выполняет и завершает работу. В противном случае, начинается процесс удаления узлов, начиная с первого узла после tail. Каждый узел удаляется с помощью оператора delete, освобождая выделенную под него память. После удаления всех узлов, указатель tail устанавливается в nullptr, размер списка size обнуляется, а указатели currentNode и currentIndex сбрасываются.

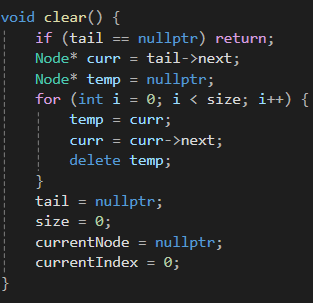


Рисунок 1.13 – Метод для очистки списка

В этом блоке кода определяется функция main() (рис. 1.14), которая является точкой входа в программу. Пользователь может вводить команды для выполнения различных операций со списком, таких как добавление нового элемента (add), вставка элемента в определенную позицию (insert), удаление элемента по индексу (removeAt), получение элемента по индексу (elementAt), подсчет количества элементов в списке (count), вставка узла перед каждым отрицательным значением (insertBeforeNegative), удаление всех отрицательных элементов (removeNegative), подсчет количества узлов с определенным значением (countData) и очистка всего списка (clear). Цикл продолжается до тех пор, пока пользователь не введет команду stop или другую команду, не соответствующую ни одной из перечисленных, что приведет к выходу из программы.

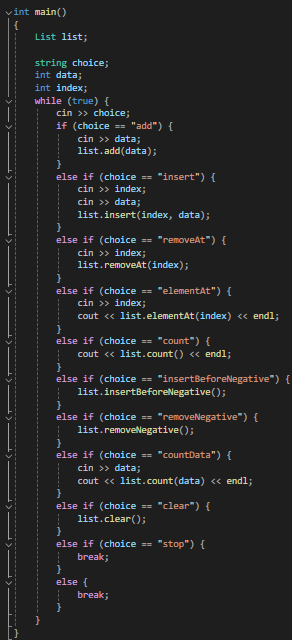


Рисунок 1.14 – Функция main

**Выводы.**

В ходе выполнения лабораторной работы была разработана программа, реализующая структуру данных односвязного циклического списка List, предназначенная для обработки целочисленных последовательностей. Структура обеспечивает произвольный доступ к данным и включает в себя комплекс функций для управления элементами списка, таких как добавление, вставка, удаление, доступ к элементам, подсчет их количества, а также операции для работы с отрицательными числами. Пользовательский интерфейс, включенный в программу, позволяет взаимодействовать со списком через консольные команды, предоставляя удобный доступ к его функциям.

# Список использованной литературы

1. Павловская Т.А., Чаевников В.В., Юрков Н.К. Программирование на языке С++. Электронное методическое пособие. СПб.: Издательство СПбГЭТУ «ЛЭТИ». 2014.
2. Практикум по процедурному программированию на языке C++ С. А. Ивановский, А. А. Лисс, В. П. Самойленко, О. М. Шолохова. Практикум по процедурному программированию на языке C++: учеб. пособие. СПб.: Издательство СПбГЭТУ«ЛЭТИ», 2016.
3. Справочник по стандартной библиотеке C++ (STL) // Справочный сайт learn.microsoft.com URL: https://learn.microsoft.com/ru-ru/cpp/standard-library/cpp-standard-library-reference?view=msvc-170 (дата обращения 09.03.2024).

Документация по языку C++ // Справочный сайт learn.microsoft.com URL: https://learn.microsoft.com/ru-ru/cpp/cpp/?view=msvc-170 (дата обращения 10.03.2024).

# Приложение

## Код написанной программы

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

Node(int value) {

data = value;

next = nullptr;

}

};

class List {

private:

Node\* tail;

Node\* currentNode;

int size;

int currentIndex;

Node\* getNode(int index) {

if (index < currentIndex) {

currentIndex = 0;

currentNode = nullptr;

}

if (currentNode == nullptr) {

currentNode = tail->next;

}

while (currentIndex < index) {

currentNode = currentNode->next;

currentIndex++;

}

return currentNode;

}

public:

List() {

tail = currentNode = nullptr;

size = currentIndex = 0;

}

~List() {

clear();

}

void add(int value) {

Node\* newNode = new Node(value);

if (tail == nullptr) {

tail = newNode;

tail->next = newNode;

}

else {

newNode->next = tail->next;

tail->next = newNode;

tail = newNode;

}

size++;

currentIndex = 0;

currentNode = nullptr;

}

void insert(int index, int value) {

index %= size + 1;

if (size == 0) { index = 0; }

if (index == size) {

add(value);

return;

}

Node\* newNode = new Node(value);

if (index == 0) {

newNode->next = tail->next;

tail->next = newNode;

}

else {

Node\* previous = getNode(index - 1);

newNode->next = previous->next;

previous->next = newNode;

}

size++;

currentIndex = index;

currentNode = newNode;

}

void removeAt(int index) {

if (tail == nullptr) return;

index = index % size;

if (size == 1) {

delete tail;

currentNode = nullptr;

currentIndex = 0;

tail = nullptr;

size = 0;

return;

}

Node\* temp = nullptr;

if (index == 0) {

temp = tail->next;

if (currentNode == temp) {

currentNode = temp->next;

currentIndex++;

}

tail->next = temp->next;

delete temp;

size--;

return;

}

Node\* prev = getNode(index - 1);

temp = prev->next;

prev->next = temp->next;

if (temp == tail) {

tail = prev;

}

delete temp;

size--;

}

int elementAt(int index) {

if (tail == nullptr) return -1;

index = index % size;

Node\* node = getNode(index);

if (node == nullptr) {

return -1;

}

return node->data;

}

int count() {

return size;

}

void insertBeforeNegative() {

if (tail == nullptr) return;

Node\* curr = tail->next;

Node\* prev = tail;

int index = 0;

do {

if (curr->data < 0) {

Node\* newNode = new Node(1);

if (curr == tail->next) {

newNode->next = tail->next;

tail->next = newNode;

}

else {

newNode->next = curr;

prev->next = newNode;

}

size++;

if (index == currentIndex) {

currentNode = newNode;

}

else if (index < currentIndex) {

currentIndex++;

}

}

prev = curr;

curr = curr->next;

index++;

} while (curr != tail->next);

}

void removeNegative() {

if (size == 0) return;

Node\* curr = tail->next;

Node\* prev = tail;

int index = 0;

while (curr != tail) {

if (curr->data < 0) {

if (curr == currentNode) {

currentNode = prev;

currentIndex = index - 1;

}

else if (index < currentIndex) {

currentIndex--;

}

prev->next = curr->next;

delete curr;

curr = prev->next;

size--;

}

else {

prev = curr;

curr = curr->next;

}

index++;

}

if (tail->data < 0) {

if (currentNode == tail) {

currentNode = prev;

currentIndex = index - 1;

}

prev->next = tail->next;

delete tail;

tail = prev;

size--;

}

if (size == 0) {

tail = nullptr;

currentNode = nullptr;

currentIndex = 0;

}

}

int count(int value) {

int k = 0;

if (tail == nullptr) return k;

Node\* curr = tail->next;

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

if (curr->data == value) {

k++;

}

curr = curr->next;

}

return k;

}

void clear() {

if (tail == nullptr) return;

Node\* curr = tail->next;

Node\* temp = nullptr;

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

temp = curr;

curr = curr->next;

delete temp;

}

tail = nullptr;

size = 0;

currentNode = nullptr;

currentIndex = 0;

}

};

int main()

{

List list;

string choice;

int data;

int index;

while (true) {

cin >> choice;

if (choice == "add") {

cin >> data;

list.add(data);

}

else if (choice == "insert") {

cin >> index;

cin >> data;

list.insert(index, data);

}

else if (choice == "removeAt") {

cin >> index;

list.removeAt(index);

}

else if (choice == "elementAt") {

cin >> index;

cout << list.elementAt(index) << endl;

}

else if (choice == "count") {

cout << list.count() << endl;

}

else if (choice == "insertBeforeNegative") {

list.insertBeforeNegative();

}

else if (choice == "removeNegative") {

list.removeNegative();

}

else if (choice == "countData") {

cin >> data;

cout << list.count(data) << endl;

}

else if (choice == "clear") {

list.clear();

}

else if (choice == "stop") {

break;

}

else {

break;

}

}

}

Файл data1-1.txt:

add 5

add -10

add 15

count

removeAt 1

elementAt 1

stop

Файл report1-1.txt:

5

-10

15

3

-10

15

Файл data2-1.txt:

add 20

add -1

insert 0 25

count

countData 20

removeNegative

count

stop

Файл report2-1.txt:

25

20

-1

3

1

2

Файл data3-1.txt:

add 30

add -40

add 50

count

insertBeforeNegative

count

clear

count

stop

Файл report3-1.txt:

30

-40

50

3

2

Очередь очищена

0