**Лабораторная работа №1**

**Линейные списки**

**10 ВАРИАНТ**

**Даны два стека целых чисел. Из элементов пеpвого стека меньших сpеднего аpифметического элементов втоpого сфоpмиpовать очередь.**

Выполнить задания, используя для пpедставления очеpедей и стеков: а) массивы; б) динамические списки.

Требования к программам:

1. Количество элементов исходных линейных списков заранее не определено и задается случайным образом. При дальнейшей обработке считается, что количество элементов списка неизвестно, т.е. обработка производится, пока не достигнут конец списка.

2. Программа должна сформировать исходные линейные списки, вывести их содержимое на экран (при этом данные из списков не должны быть потеряны), произвести обработку и вывести содержимое итогового списка на экран.

Стек - это структура данных, которая работает по принципу **FILO** (first in - last out; первый пришел - последний ушел).

В стеке элемент, который вошел самый **первый** - выйдет самым **последним**. Получается, если вы добавили три элемента в стек первым будет удален последний добавленный элемент.

Очередь - структура данных с доступом к элементам по принципу FIFO (First In First Out - Первый пришел - Первый вышел). Данные добавляются в конец, а извлекаются из начала.

а) массивы

Программный код:

#include<iostream>

using namespace std;

//реализация стека массивом

struct Stack

{

int top;

int\* data;

};

//Эта функция инициализирует стек через динамическое выделение памяти для целочисленного массива в структуре Stack. Она устанавливает вершину стека

//(изначально -1) и ёмкость на основе входного параметра.

void InitStack(Stack& st, int capacity)

{

st.data = new int[capacity];

st.top = -1;

}

//Данная функция добавляет элемент на вершину стека путем инкрементирования вершины стека

//и присваивания значения соответствующему индексу в массиве данных.

void push(Stack& st, int value)

{

st.data[++st.top] = value;

}

//Эта функция удаляет и возвращает верхний элемент стека, обращаясь к элементу и уменьшая индекс вершины.

int pop(Stack& st)

{

return st.data[st.top--];

}

//Эта функция сбрасывает индекс вершины стека на -1, указывая на пустой стек.

void nullStack(Stack& st)

{

st.top = -1;

}

//Эта функция проверяет, пуст ли стек, сравнивая индекс вершины с -1.

bool empty(Stack& st)

{

return st.top == -1;

}

//Эта функция отображает элементы стека снизу вверх, перебирая массив данных и выводя элементы. Если стек пуст, выводится соответствующее сообщение.

void show(Stack& st)

{

if (!empty(st))

{

for (int i = 0; i <= st.top; i++)

{

cout << st.data[i] << " ";

}

cout << endl;

}

else cout << "Стек пуст!" << endl;

}

//Реализация очереди массивом

struct Queue

{

int head, tail, size;

int\* data;

};

//Эта функция инициализирует начало и конец (голову и хвост) структуры очереди (Queue) в зависимости от размера очереди.

void nullQueue(Queue& q)

{

q.head = 0; q.tail = q.size - 1;

}

//Данная функция инициализирует очередь через динамическое выделение памяти для целочисленного массива в структуре Queue. Устанавливает начало, конец, размер

//и ёмкость входного параметра.

void InitQueue(Queue& q, int capacity)

{

q.size = capacity + 1;

q.data = new int[q.size];

nullQueue(q);

}

//Эта функция вычисляет следующий индекс в циклической очереди на основе текущего индекса и размера очереди.

int next(Queue& q, int n)

{

return (n + 1) % q.size;

}

//Эта функция проверяет, пуста ли очередь, сравнивая следующий индекс хвоста с индексом головы.

bool empty(Queue& q)

{

return next(q, q.tail) == q.head;

}

//Данная функция добавляет элемент в очередь, обновляя индекс хвоста и присваивая значение соответствующему индексу в массиве данных.

void add(Queue& q, int value)

{

if (next(q, next(q, q.tail)) == q.head)

cout << "Очередь переполнена" << endl;

else

{

q.tail = next(q, q.tail);

q.data[q.tail] = value;

}

}

//Эта функция удаляет и возвращает элемент в голове очереди, обращаясь и обновляя индекс головы.

int del(Queue& q)

{

if (empty(q))

{

cout << "Очередь пуста!" << endl;

return 0;

}

else

{

int d = q.data[q.head];

q.head = next(q, q.head);

return d;

}

}

//Данная функция выводит все элементы в очереди, извлекая элементы один за другим и выводя их.

//Затем она сбрасывает индекс головы для сохранения исходного состояния очереди.

void print(Queue& q)

{

if (empty(q))

cout << "Очередь пуста!" << endl;

else

{

int save = q.head;

q.head = save;

while (!empty(q))

{

int p = del(q);

cout << p << " ";

}

cout << endl;

q.head = save;

}

}

int main() {

setlocale(LC\_ALL, "ru");

srand(time(0));

Stack Stack1;

Stack Stack2;

Queue result;

double product = 0;//переменная для подсчета произведения всех чисел второго стека

double counter = 0;//переменная для подсчета количества элементов второго стека

int capacity = rand() % 50;//емкость первого стека

InitStack(Stack1, capacity);//инициализация первого стека

//заполнение первого стека

for (int i = 0; i < capacity; i++)

{

int y = rand() % 1001 - 500;

push(Stack1, y);

}

// Вывод содержимого стека

cout << "Первый стек (линейный список) целых чисел" << endl;

show(Stack1);

cout << "\n" << endl;

int capacity2 = rand() % 50;//емкость второго стека

InitStack(Stack2, capacity2);//инициализация второго стека

//заполнение второго стека

for (int i = 0; i < capacity2; i++)

{

int y = rand() % 1001 - 500;

push(Stack2, y);

product += y;

counter++;

}

// Выводим содержимое стека, "выталкивая" элементы из стека функцией pop()

cout << "Второй стек (линейный список) целых чисел" << endl;

show(Stack2);

cout << "\n" << endl;

if (counter == 0) {

cout << "Среднее арифметическое второго стека невозможно посчитать, так как на 0 делить нельзя" << "\n" << endl;

delete[] Stack1.data; // Освобождение памяти

delete[] Stack2.data; // Освобождение памяти

return 0;

}

double average = product / counter;//переменная, отвечающая за подсчет среднего арифметического 2 стека

InitQueue(result, capacity); // Инициализация очереди

while (!empty(Stack1))

{

int queueCapacity = pop(Stack1);

if (queueCapacity < average)

{

add(result, queueCapacity);

}

}

cout << "Среднее арифметическое второго стека: " << average << "\n\n" << endl;

cout << "Очередь из элементов пеpвого стека меньших сpеднего аpифметического элементов втоpого " << endl;

print(result);

delete[] result.data;

delete[] Stack1.data; // Освобождение памяти

delete[] Stack2.data; // Освобождение памяти

return 0;

}

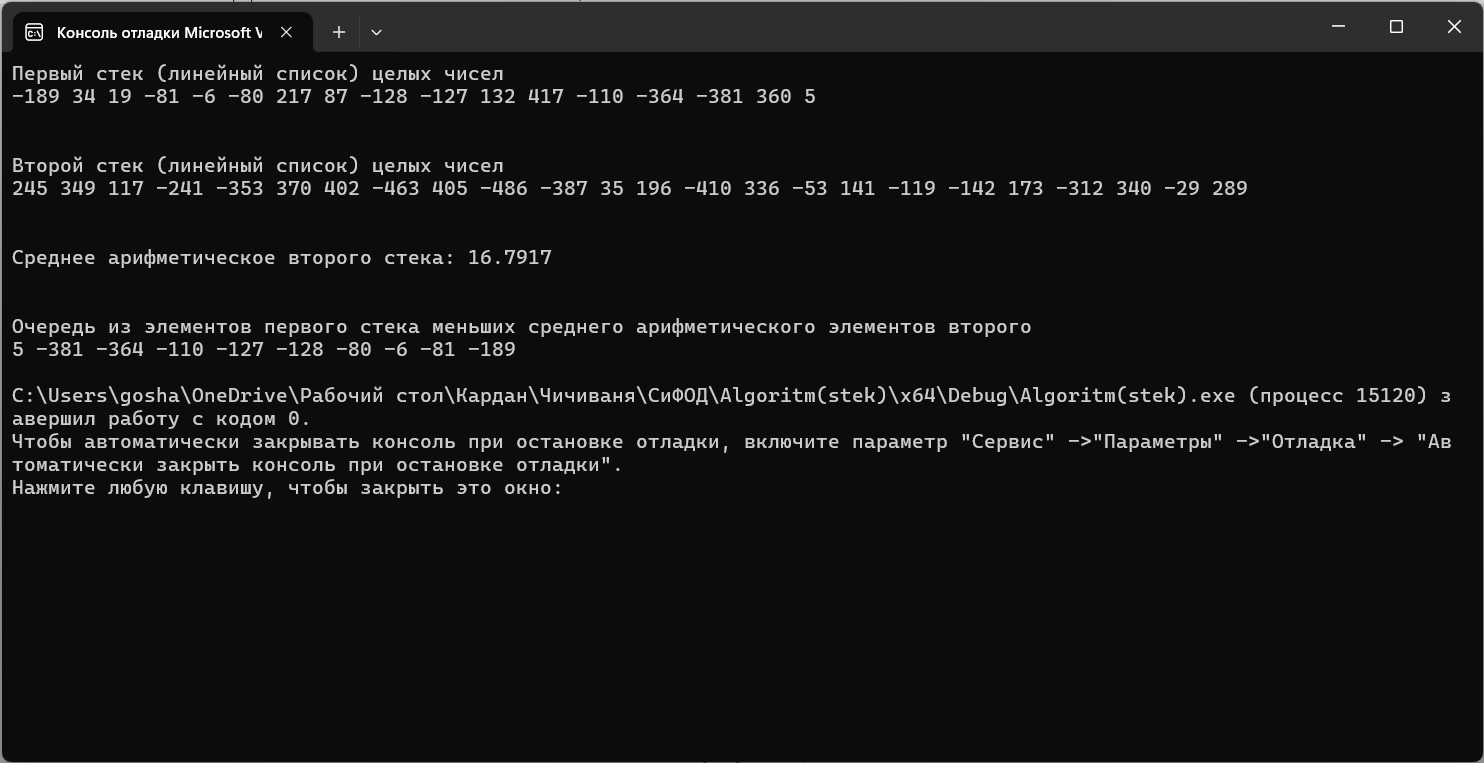


Рис. 1. Вывод программы, если элементы найдены

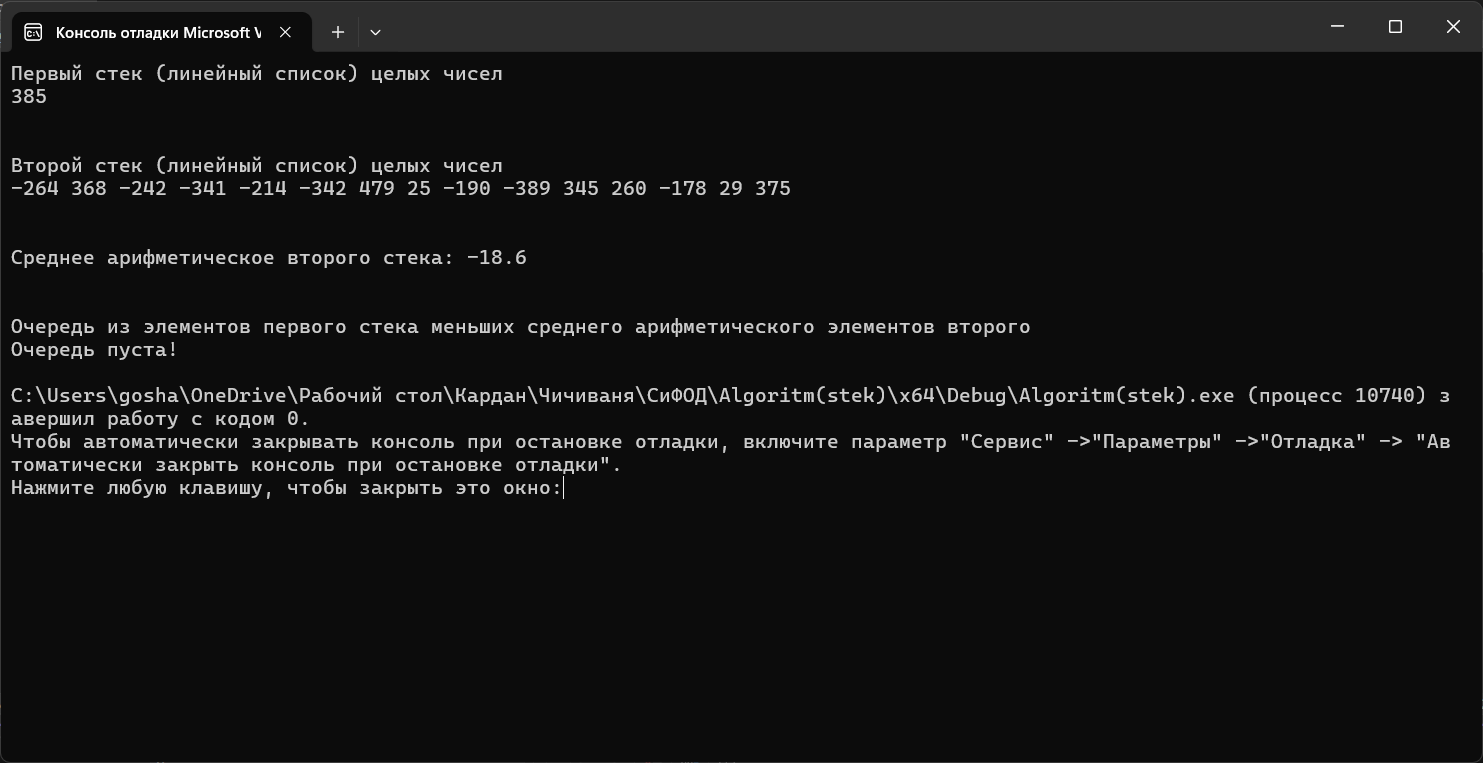


Рис. 2. Вывод программы, если элементы не найдены

В функции push, используется префиксный инкремент ++st.top, потому что нужно увеличить значение st.top до записи значения в массив st.data. Сначала мы увеличиваем top, чтобы указать на следующий свободный элемент, а затем присвоить значение.

В функции pop используется постфиксный декремент st.top--, потому что сначала нужно использовать текущее значение st.top для доступа к элементу массива st.data, а затем уменьшить top, чтобы указать на новый верхний элемент стека.

По итогу для того, чтобы правильно управлять индексом массива.

Описание алгоритма работы программы

Инициализация структур данных:

- Создаются структуры Stack для представления стеков и Queue для представления очереди.

- В структурах Stack и Queue хранятся данные о вершине стека (top) и размере очереди (head и tail), а также массивы для хранения данных.

2. Заполнение первого стека (Stack1):

- Сгенерирована случайная емкость capacity для первого стека.

- Инициализирован первый стек Stack1 заданной емкостью и заполнен случайными целыми числами из диапазона [-500, 500].

- Выводится содержимое первого стека Stack1.

3. Заполнение второго стека (Stack2):

- Сгенерирована случайная емкость capacity2 для второго стека.

- Инициализирован второй стек Stack2 заданной емкостью и заполнен случайными целыми числами из диапазона [-500, 500].

- Рассчитывается сумма значений второго стека product и количество элементов в нем counter.

- Выводится содержимое второго стека Stack2.

4. Вычисление среднего арифметического:

- Вычисляется среднее арифметическое значений второго стека и сохраняется в переменной average, но если во втором стеке отсутствуют элементы программа завершает работу с кодом 0.

5. Формирование очереди (Queue) из элементов первого стека меньших среднего арифметического второго стека:

- Инициализируется очередь result.

- Пока первый стек Stack1 не пуст, извлекается элемент queueCapacity.

- Если queueCapacity меньше чем average, то этот элемент добавляется в очередь result.

6. Вывод результатов:

- Выводится значение среднего арифметического второго стека.

- Отображается содержимое очереди result, содержащей элементы из первого стека, которые меньше среднего арифметического второго стека.

7. Освобождение динамической памяти:

- Освобождается память для массивов данных первого стека (Stack1.data), второго стека (Stack2.data) и очереди (result.data).

Функция **int next(Queue& q, int n)** реализована таким образом для обеспечения циклической навигации по элементам в структуре данных очереди.

1. **Циклическая структура данных**: В очереди обычно используется циклическая структура данных, где элементы добавляются в конец очереди и удаляются из начала. При достижении конца массива (кругового буфера) следующий элемент должен быть добавлен в начало массива. Для этого используется операция остатка от деления на размер очереди (**q.size**).
2. **Обход кольцевого буфера**: Путем выполнения операции **(n + 1) % q.size**, обеспечивается корректный переход к следующему элементу в круговом буфере. Для начала массива это приводит к **0**, а для последнего элемента - к возврату к началу массива.
3. **Эффективность и простота**: Данная реализация обеспечивает эффективный и простой способ обновления индекса n на следующий элемент в круговом буфере без необходимости дополнительной логики проверки на выход за пределы границ массива.

Таким образом, функция **next** реализована именно так для обеспечения корректного обхода элементов в циклической структуре данных очереди, где индекс следующего элемента определяется операцией остатка от деления.

**Недостаток реализации в массиве** заключается в том, что когда хвост очереди дойдёт до последнего элемента массива, логично предположить, что очередь переполнена. Однако к этому времени из ‘головы’ могут быть забраны несколько элементов, и реально в массиве свободное место будет. Чтобы в такой ситуации не перегонять всю очередь в начало массива, было предложено считать массив закольцованным: когда за последним элементом массива следует первый. В этом случае для вычисления номера элемента массива, следующего за i-ым, используется формула (i+1) % size

Чтобы при такой реализации различать пустую и переполненные очереди, решили считать очередь пустой, если сразу за хвостом следует голова, а переполненной – если голова через 1 элемент от хвоста. То есть, в массиве из N элементов максимально может поместиться N-1 элемент очереди.

б) динамические списки.

#include<iostream>

using namespace std;

// Создается структура Node, которая содержит два поля: data (хранит целочисленное значение) и next (указатель на следующий узел в списке).

struct Node

{

int data;

Node\* next;

};

// Функция InitStack инициализирует стек, устанавливая указатель на вершину стека в NULL.

void InitStack(Node\*& top)

{

top = NULL;

}

// Функция push добавляет новый узел с заданным значением в вершину стека. Происходит выделение памяти под новый узел,

// установка указателя next на текущий верхний элемент, а указатель top указывает на новую вершину.

void push(Node\*& top, int value)

{

Node\* tmp = new Node;

tmp->next = top;

top = tmp;

top->data = value;

}

// Функция pop удаляет верхний узел стека, сохраняет его данные, перемещает указатель top на следующий элемент и освобождает память, занятую удаленным узлом.

// Возвращает данные удаленного узла.

int pop(Node\*& top)

{

Node\* tmp = top;

int d = top->data;

top = top->next;

delete(tmp);

return d;

}

// Функция empty проверяет, пуст ли стек, сравнивая указатель top с NULL. Возвращает true, если стек пуст, и false в противном случае.

bool empty(Node\*& top)

{

return top == NULL;

}

// Функция nullStack освобождает всю память, занимаемую узлами стека. Происходит удаление всех узлов стека.

void nullStack(Node\*& top)

{

Node\* tmp;

while (!empty(top))

{

tmp = top;

top = top->next;

delete(tmp);

}

}

// Функция print выводит на экран содержимое стека, начиная с верхнего элемента. Если стек пуст, выводится сообщение "Стек пуст!".

void print(Node\* top)

{

if (!empty(top))

{

Node\* cur = top;

while (cur != 0)

{

cout << cur->data << " ";

cur = cur->next;

}

cout << endl;

}

else cout << "Стек пуст!" << endl;

}

// Определяется класс Queue, используемый для представления очереди на основе односвязного списка.

class Queue

{

private:

struct Node

{

int data;

Node\* next;

};

Node\* head;

Node\* tail;

public:

// Конструктор класса создает пустую очередь, устанавливая указатели head и tail в NULL.

Queue()

{

head = nullptr;

tail = nullptr;

}

// Метод empty() класса Queue проверяет пуста ли очередь, возвращая true, если head равен NULL.

bool empty()

{

return head == nullptr;

}

// Метод add() добавляет новый узел с заданным значением в хвост очереди. Если очередь пуста, устанавливает head и tail на новый узел.

void add(int value)

{

Node\* newNode = new Node;

newNode->data = value;

newNode->next = nullptr;

if (empty())

{

head = newNode;

tail = newNode;

} else

{

tail->next = newNode;

tail = newNode;

}

}

// Метод del() удаляет и возвращает значение из головы очереди, смещая указатель head.

int del()

{

if (empty())

{

cout << "Очередь пуста!" << endl;

return 0;

}

else

{

int deletedData = head->data;

Node\* tmp = head;

head = head->next;

delete tmp;

return deletedData;

}

}

// Метод nullQueue() удаляет все узлы из очереди, освобождая память.

void nullQueue()

{

Node\* tmp;

while (head != nullptr)

{

tmp = head;

head = head->next;

delete tmp;

}

tail = nullptr; // Обнуляем указатель на конец очереди

}

// Метод show() возвращает значение из головы очереди, удаляя его и смещая указатель head.

int show()

{

if (!empty())

{

return head->data;

}

else

{

cout << "Очередь пуста!" << endl;

return 0;

}

}

// Метод print() выводит содержимое очереди, удаляя элементы из головы и выводя их значения. После вывода указатели возвращаются в исходное состояние.

void print()

{

Node\* current = head;

while (current != nullptr)

{

cout << current->data << " ";

current = current->next;

}

cout << endl;

}

};

int main()

{

setlocale(LC\_ALL, "ru");

srand(time(0));

Node\* Stack1;

Node\* Stack2;

Queue result;

double product = 0;//переменная для подсчета произведения всех чисел второго стека

double counter = 0;//переменная для подсчета количества элементов второго стека

int capacity = rand() % 50;//емкость первого стека

int capacity2 = rand() % 50;//емкость второго стека

InitStack(Stack1);

for (int i = 0; i < capacity; i++)

{

int y = rand() % 1001 - 500;

push(Stack1, y);

}

// Вывод содержимого первого стека

cout << "Первый стек (линейный список) целых чисел" << endl;

print(Stack1);

cout << "\n" << endl;

InitStack(Stack2);

for (int i = 0; i < capacity2; i++)

{

int y = rand() % 1001 - 500;

push(Stack2, y);

product += y;

counter++;

}

//Вывод содержимого второго стека

cout << "Второй стек (линейный список) целых чисел" << endl;

print(Stack2);

cout << "\n" << endl;

if (counter == 0) {

cout << "Среднее арифметическое второго стека невозможно посчитать, так как на 0 делить нельзя" << "\n" << endl;

return 0;

}

double average = product / counter;//переменная, отвечающая за подсчет среднего арифметического 2 стека

while (!empty(Stack1))

{

int queueCapacity = pop(Stack1);

if (queueCapacity < average)

{

result.add(queueCapacity);

}

}

cout << "Среднее арифметическое второго стека: " << average << "\n\n" << endl;

cout << "Очередь из элементов пеpвого стека меньших сpеднего аpифметического элементов втоpого " << endl;

result.print();

return 0;

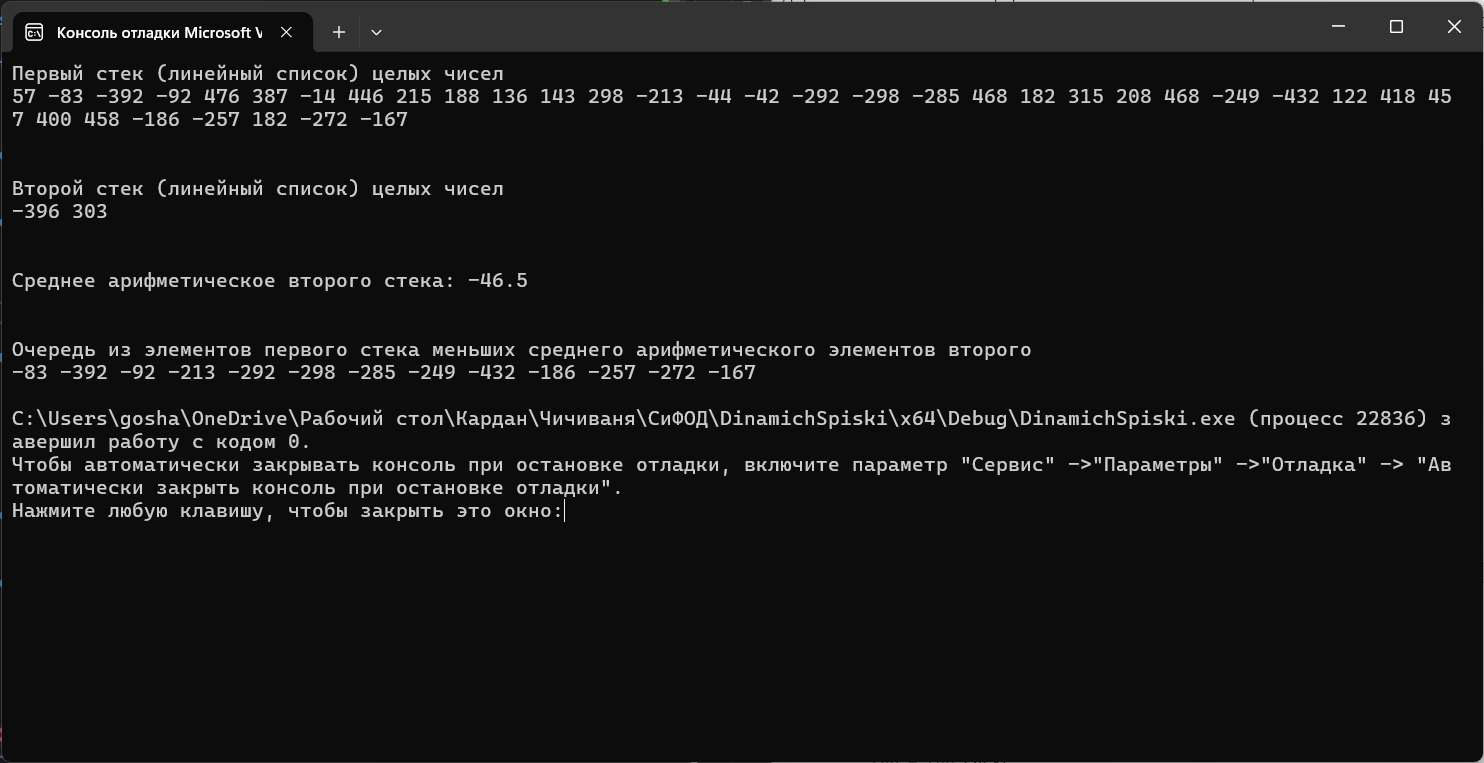
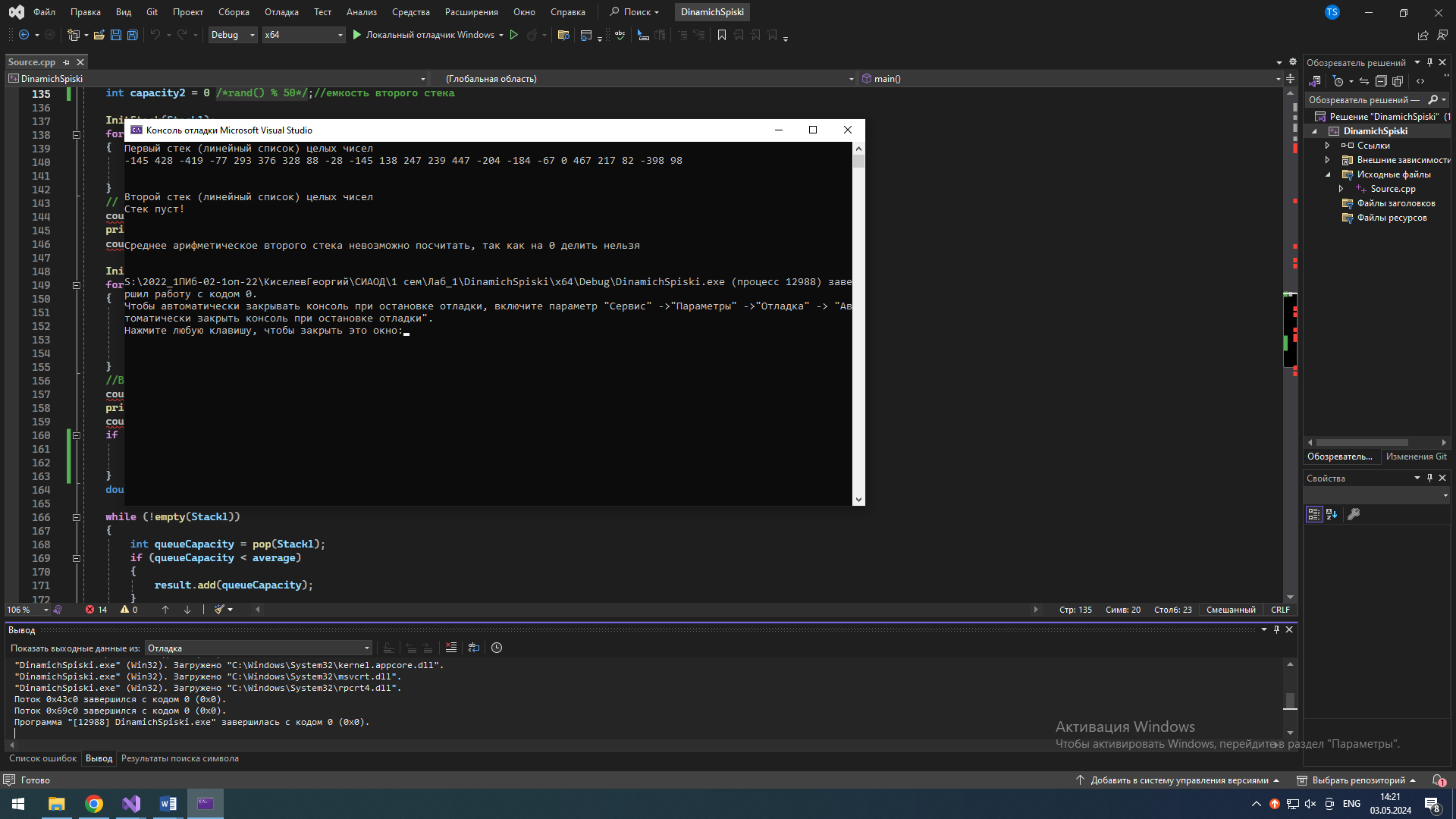
}

Рис. 3. Вывод программы, если элементы найдены

  
Рис. 4. Вывод программы, если элементы не найдены

Описание алгоритма работы программы

1. Инициализация структуры стека (struct Node):

- Узел Node представляет элемент стека с данными data и указателем next на следующий элемент.

2. Реализация операций для стеков (push, pop, empty, nullStack, print):

- InitStack: Инициализация стека, устанавливая указатель верхушки стека в NULL.

- push: Добавление нового элемента в стек, создание нового узла и его добавление в верхнюю часть стека.

- pop: Удаление и возврат значения из верхушки стека.

- empty: Проверка на пустоту стека.

- nullStack: Очистка стека путем удаления всех его элементов.

- print: Вывод содержимого стека на экран.

3. Класс Queue (очередь) с методами для работы с очередью:

- Описан класс Queue с приватными полями и методами для работы с очередью, использующие структуру Node.

- Методы включают: add (добавление элемента в конец очереди), del (удаление и возврат элемента из начала очереди), empty (проверка на пустоту), nullQueue (очистка очереди), show (удаление и возврат элемента из начала очереди), print (вывод содержимого очереди).

4. Основная часть программы в функции main():

- Создаются указатели Stack1 и Stack2, а также объект result типа Queue.

- Генерируются случайные величины для задания емкости capacity и capacity2 первого и второго стеков соответственно.

- Инициализируются стеки Stack1 и Stack2, заполняются случайными числами, а также подсчитывается среднее арифметическое значений второго стека.

- Элементы из первого стека, меньшие среднего значения второго стека, добавляются в очередь result.

- Выводится среднее арифметическое второго стека и содержимое очереди result.

5. Освобождение памяти и завершение программы:

- Память освобождается после использования массивов данных стеков и очереди.

- Программа завершает свою работу с кодом возврата 0.