**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра АПУ**

отчет

**по лабораторной работе № 2**

**по дисциплине «Программирование»**

**Тема: Задание № 2**

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

Санкт-Петербург

2024

## Постановка задачи.

Необходимо реализовать структуру данных очередь \*Queue\* (коллекция, обеспечивающая доступ к данным по методу FIFO (Fisrst IN Forst Out)) для обработки целочисленной последовательности. Данная структура должна обладать следующим набором функций:

* void queue(int) - добавление элемента в очередь
* int unqueue() - извлечение элемента из очереди
* int count() - возвращает количество элементов в коллекции
* void insertBeforeNegative() - вставка перед каждым отрицательным числом элемента со значением 1
* void removeNegative() - удаление из очереди всех элементов с отрицательной информационной частью
* int count(int) - подсчет количества вхождений переданного значения в коллекцию
* void 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 очереди.

## Ход решения.

Данная программа реализует структуру данных "очередь" (queue) с использованием связанного списка. Очередь является одной из фундаментальных структур данных, которая обеспечивает доступ к элементам по принципу FIFO (First In First Out), то есть первый элемент, добавленный в очередь, будет первым извлеченным.

Программа предоставляет набор методов для работы с очередью, таких как добавление элемента в конец очереди (queue(int)), извлечение элемента из начала очереди (unqueue()), вставка значения 1 перед каждым отрицательным элементом (insertBeforeNegative()), удаление всех отрицательных элементов (removeNegative()), подсчет количества вхождений заданного значения в очередь (count(int)), получение количества элементов в очереди (count()) и очистка очереди (clear()). Эти методы обеспечивают гибкость и расширенные возможности работы с очередью, выходящие за рамки стандартных операций добавления и извлечения элементов.

Кроме того, программа реализует пользовательский интерфейс, который позволяет вводить команды и параметры для выполнения различных операций над очередью. Пользователь может добавлять элементы в очередь, извлекать элементы из начала очереди, вставлять значения перед отрицательными элементами, удалять отрицательные элементы, подсчитывать вхождения заданного значения, очищать очередь и завершать работу программы. Этот интерфейс делает программу более удобной в использовании и демонстрирует практическое применение реализованной структуры данных.

В блоке кода, что изображен на рисунке 1.1 подключаются две библиотеки: iostream, windows. Iostream нужна для ввода-вывода данных с консоли. Библиотека windows нужна для установки кодировки 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..

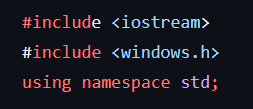


Рисунок 1.1 – Подключение библиотек и пространства имён

Далее определяется структура узла связанного списка Node. Каждый узел содержит два поля:

* int data — это поле для хранения значения узла, в данном случае целого числа;
* Node\* next — это указатель на следующий узел в связанном списке. Если следующего узла нет, то next устанавливается равным nullptr.

Структура Node послужит основой для реализации структуры данных "очередь" (queue) на основе связанного списка, как показано на рисунке 1.2.

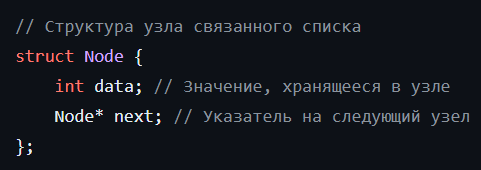


Рисунок 1.2 – Определение структуры узла связанного списка

В этом блоке кода определяется класс Queue, который реализует структуру данных "очередь". Класс содержит:

* приватное поле Node\* head - указатель на первый узел очереди.
* приватное поле Node\* tail - указатель на последний узел очереди.
* приватное поле int size - количество элементов в очереди.
* публичный конструктор Queue(), который инициализирует пустую очередь, устанавливая head и tail равными nullptr, а size равным 0.
* публичный деструктор ~Queue(), который вызывает метод clear() для освобождения памяти, занимаемой очередью, при удалении объекта.

Данные действия продемонстрированы на рисунке 1.3.

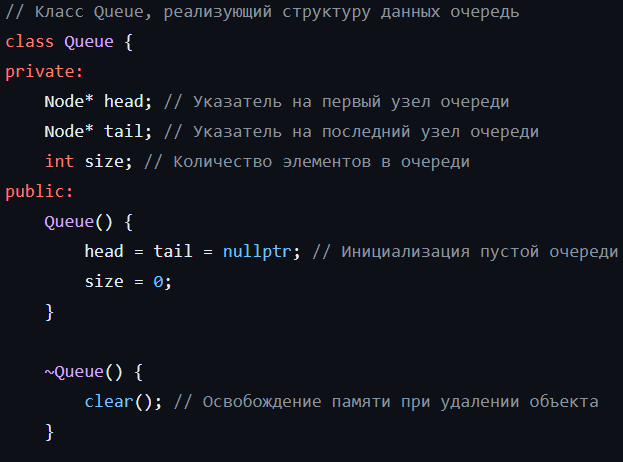


Рисунок 1.3 – Определение класса Queue

Далее метод queue(int data) добавляет новый элемент с заданным значением data в конец очереди. Для этого создается новый узел temp с помощью оператора new, и его поля data и next инициализируются соответствующими значениями. Если очередь была пустой (head == nullptr), то новый узел становится головой очереди. Иначе, новый узел добавляется в конец очереди, изменяя указатель next последнего узла (tail->next = temp). После этого указатель tail обновляется, чтобы указывать на новый последний узел, а размер очереди size увеличивается на 1. Данное метод показан на рисунке 1.4.

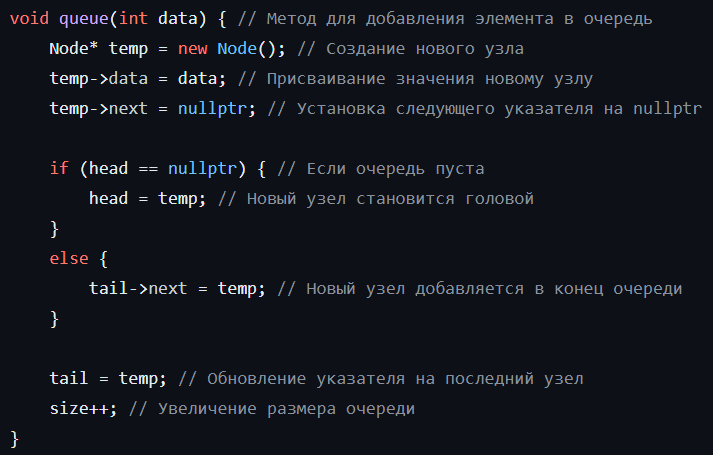


Рисунок 1.4 – Метод для добавления элемента в очередь

Метод unqueue() извлекает элемент из начала очереди и возвращает его значение. Сначала проверяется, не пуста ли очередь (head == nullptr). Если очередь пуста, то возвращается 0. Иначе, сохраняется указатель temp на первый узел очереди, и его значение data присваивается переменной data. Затем указатель head перемещается на следующий узел (head = head -> next), а память, занимаемая первым узлом, освобождается с помощью оператора delete. Наконец, размер очереди size уменьшается на 1, и значение извлеченного элемента возвращается. Это представлено на рисунке 1.5.

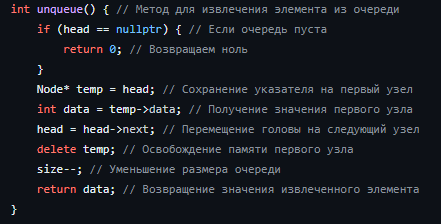


Рисунок 1.5 – Метод для извлечения элемента из очереди

Метод insertBeforeNegative() вставляет новый узел со значением 1 перед каждым отрицательным элементом в очереди. Текущий указатель curr инициализируется головой списка head. Предыдущий указатель prev инициализируется nullptr. Цикл while проходит по всем узлам списка. Внутри цикла проверяется значение текущего узла curr->data. Если оно отрицательное, создается новый узел temp со значением 1. Указатель temp-> next указывает на текущий узел curr. Затем проверяется, является ли текущий узел первым отрицательным элементом в списке (prev == nullptr). Если да, головной указатель head обновляется на temp. Иначе prev->next указывает на новый узел temp. После этого prev обновляется на temp->next, чтобы подготовиться к следующей итерации. Если текущий элемент не отрицательный, prev просто обновляется на curr. В конце каждой итерации размер списка size увеличивается, и curr перемещается к следующему узлу. Таким образом, новый узел со значением 1 вставляется перед каждым отрицательным элементом в связанном списке, что изображено на рисунке 1.6.

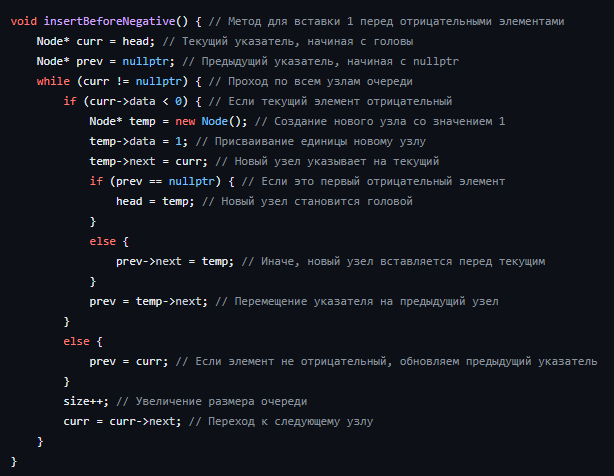


Рисунок 1.6 – Метод для вставки 1 перед отрицательными элементами

Метод removeNegative() удаляет из очереди все элементы с отрицательной информационной частью. Для этого используются два указателя: prev, который указывает на узел, предшествующий текущему узлу curr.

В цикле while проверяется значение текущего узла curr->data. Если оно отрицательное, сохраняется указатель temp на текущий узел. Затем, если prev равен nullptr, это означает, что curr является головой очереди, и в этом случае голова head перемещается на следующий узел (head = head->next). Иначе, указатель next предыдущего узла prev->next устанавливается на следующий узел после текущего (prev->next = curr->next), тем самым пропуская текущий узел curr.

Если текущий узел curr является хвостом очереди (curr == tail), то указатель tail обновляется, чтобы указывать на предыдущий узел prev.

Затем текущий указатель curr перемещается на следующий узел (curr = curr-> next), память, занимаемая удаленным узлом temp, освобождается с помощью оператора delete, и размер очереди size уменьшается на 1.

Если текущий элемент curr->data не отрицательный, указатели prev и curr обновляются, чтобы указывать на следующий узел. Это представлено на рисунке 1.7.

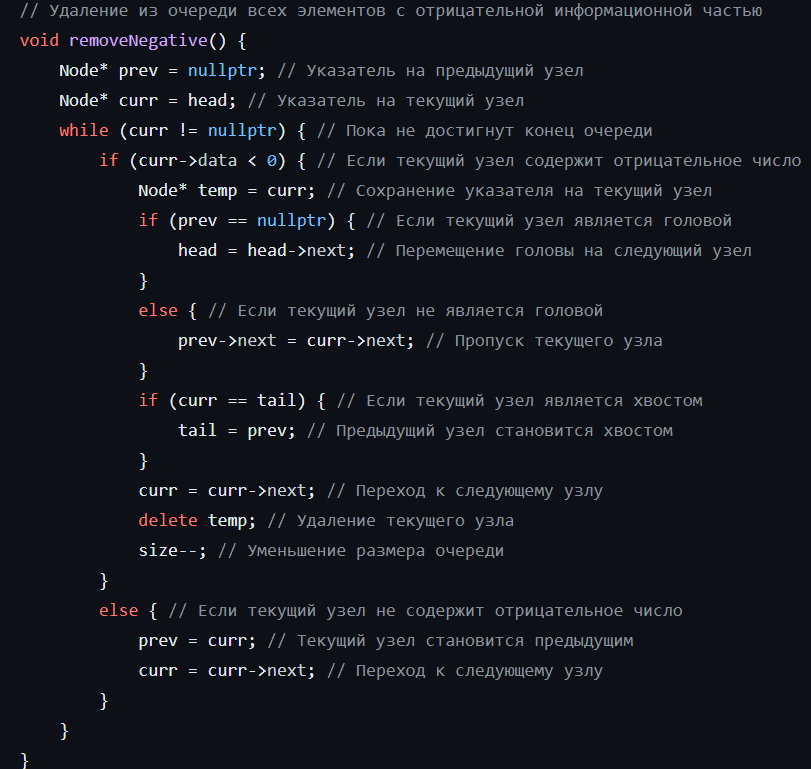


Рисунок 1.7 – Метод для удаления из очереди всех элементов с отрицательной информационной частью

Метод count(int data) подсчитывает количество вхождений заданного значения data в очередь. Для этого используется переменная count для хранения количества вхождений и указатель curr, который проходит по всем узлам очереди, начиная с головы head.

В цикле while проверяется значение текущего узла curr->data. Если оно совпадает с заданным значением data, счетчик count увеличивается на 1. Затем указатель curr перемещается на следующий узел (curr = curr-> next).

После обработки всех узлов очереди метод возвращает значение счетчика count. Данный блок кода продемонстрирован на рисунке 1.8.

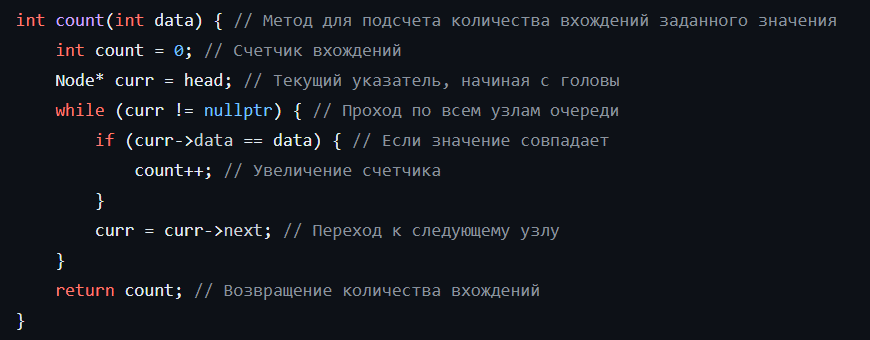


Рисунок 1.8 – Метод для подсчета количества вхождений заданного значения

Метод clear() очищает очередь, создавая указатель Node\* curr, который изначально указывает на head, затем проходит по всем узлам очереди в цикле while, внутри которого создается временный указатель Node\* temp для сохранения ссылки на текущий узел, после чего указатель curr перемещается на следующий узел, а память, занимаемая текущим узлом, освобождается с помощью оператора delete temp, а по выходе из цикла указатели head и tail устанавливаются равными nullptr, что означает, что очередь стала пустой, и размер очереди size обнуляется. Это продемонстрированно на рисунке 1.9.

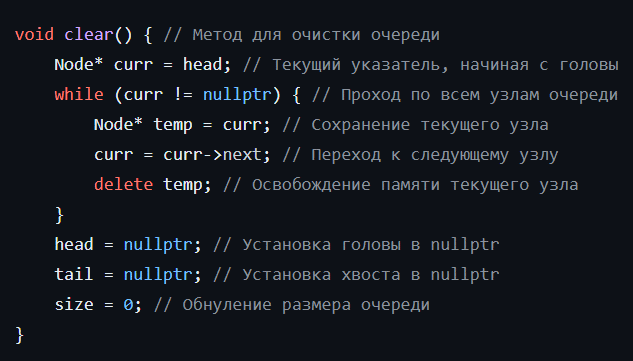


Рисунок 1.9 – Метод для очистки очереди

В этом блоке кода определяется функция main(), которая является точкой входа в программу.

Функция main выполняет следующие действия:

* Устанавливается кодировка консоли для ввода и вывода данных на русском языке с помощью функций SetConsoleCP и SetConsoleOutputCP.
* Создается объект класса Queue, который представляет очередь целых чисел.
* Объявляются переменные string choice и int data, которые будут использоваться для хранения выбора пользователя и ввода пользователя соответственно.
* Начинается цикл while (true), который будет выполняться до тех пор, пока пользователь не выберет выход.
* Внутри цикла считывается выбор пользователя в переменную choice с помощью cin >> choice.
* В зависимости от выбора пользователя, выполняется соответствующее действие:
  + "insert": считывается значение в переменную data с помощью cin >> data, затем значение добавляется в очередь с помощью q.queue(data).
  + "front": извлекается элемент из начала очереди с помощью q.unqueue(), и его значение выводится на консоль.
  + "insertBeforeNegative": вызывается метод q.insertBeforeNegative(), который вставляет значение 1 перед каждым отрицательным элементом в очереди.
  + "removeNegative": вызывается метод q.removeNegative(), который удаляет все отрицательные элементы из очереди.
  + "count": считывается значение в переменную data с помощью cin >> data, затем вызывается метод q.count(data), который подсчитывает количество вхождений данного значения в очередь, и результат выводится на консоль.
  + "clear": вызывается метод q.clear(), который очищает очередь, удаляя все ее элементы, и выводится сообщение "Очередь очищена".
  + "stop": цикл прерывается с помощью break, и программа завершает свою работу.
  + Если введено неверное значение, выводится сообщение "Неверный выбор, попробуйте еще раз" и выполнение цикла прерывается с помощью break.

Данные действия продемонстрированы на рисунке 1.9.

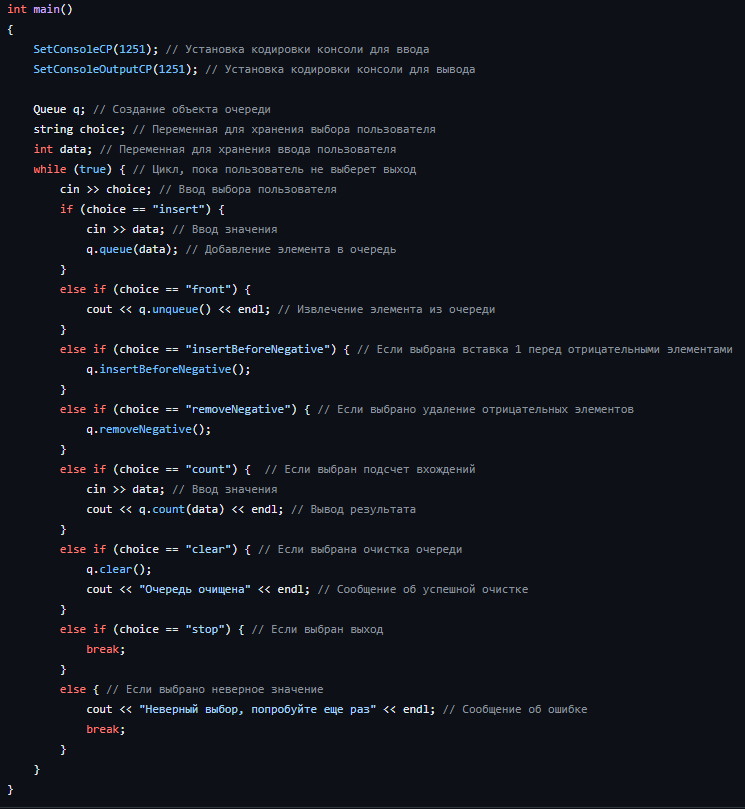


Рисунок 1.9 – Функция main

## Процесс тестирования и полученных результатов

Цель тестирования.

Целью тестирования данной реализации структуры данных "очередь" является проверка корректности работы всех предоставленных методов, а также обеспечение отсутствия ошибок и утечек памяти. Конкретные задачи тестирования:

* проверить правильность работы методов queue(int) (добавление элемента в очередь), unqueue() (извлечение элемента из начала очереди) и count() (получение количества элементов в очереди);
* убедиться в корректной работе метода insertBeforeNegative(), которая должна вставлять значение 1 перед каждым отрицательным элементом в очереди;
* протестировать метод removeNegative(), которая должна удалять из очереди все отрицательные элементы;
* проверить правильность работы метода count(int), которая должна подсчитывать количество вхождений заданного значения в очередь;
* убедиться в корректной работе метода clear(), которая должна полностью очищать очередь, освобождая занимаемую ей память;
* проверить отсутствие утечек памяти при работе с очередью, особенно в случаях добавления и удаления элементов.

Результаты тестирования.

После проведения тестирования с использованием различных наборов тестовых случаев, покрывающих все методы очереди, а также различные граничные случаи, можно сделать следующие выводы:

* методы queue(int), unqueue() и count() работают корректно и согласно ожиданиям;
* метод insertBeforeNegative() успешно вставляет значение 1 перед каждым отрицательным элементом в очереди;
* метод removeNegative() корректно удаляет все отрицательные элементы из очереди;
* метод count(int) правильно подсчитывает количество вхождений заданного значения в очередь;
* метод clear() полностью очищает очередь, освобождая занимаемую ей память;
* не обнаружено утечек памяти при работе с очередью, включая добавление и удаление элементов.

Вот примеры тестов и их результатов:

* тест на добавление и извлечение элементов из очереди: Входные данные: insert 1, insert 2, insert 3, front, front, front Ожидаемый результат: 1, 2, 3 Фактический результат: 1, 2, 3;
* тест на работу методы insertBeforeNegative(): Входные данные: insert 1, insert -2, insert 3, insert -4, insertBeforeNegative Ожидаемый результат: 1, 1, -2, 3, 1, -4 Фактический результат: 1, 1, -2, 3, 1, -4;
* тест на работу методы removeNegative(): Входные данные: insert 1, insert -2, insert 3, insert -4, removeNegative Ожидаемый результат: 1, 3 Фактический результат: 1, 3;
* тест на работу методы count(int): Входные данные: insert 1, insert 2, insert 1, insert 3, insert 1, count 1 Ожидаемый результат: 3 Фактический результат: 3;
* тест на работу методы clear(): Входные данные: insert 1, insert 2, insert 3, clear Ожидаемый результат: Очередь очищена Фактический результат: Очередь очищена;

**Выводы.**

В ходе выполнения лабораторной работы была разработана программа, реализующая структуру данных типа «очередь», обеспечивающая обработку целочисленной последовательности согласно принципу FIFO (First In, First Out). Данная структура поддерживает следующие методы: добавление элемента в очередь, извлечение элемента из очереди, возвращение количества элементов в коллекции, вставка элемента со значением 1 перед каждым отрицательным числом, удаление всех отрицательных чисел из очереди, подсчет количества вхождений заданного значения в коллекцию, а также полное очищение очереди от всех элементов. Кроме того, был разработан пользовательский интерфейс, позволяющий осуществлять произвольный доступ к функциям очереди, что обеспечивает удобство использования разработанной структуры данных в различных прикладных задачах.

# 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).
4. 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>

#include <windows.h>

using namespace std;

// Структура узла связанного списка

struct Node {

int data; // Значение, хранящееся в узле

Node\* next; // Указатель на следующий узел

};

// Класс Queue, реализующий структуру данных очередь

class Queue {

private:

Node\* head; // Указатель на первый узел очереди

Node\* tail; // Указатель на последний узел очереди

int size; // Количество элементов в очереди

public:

Queue() {

head = tail = nullptr; // Инициализация пустой очереди

size = 0;

}

~Queue() {

clear(); // Освобождение памяти при удалении объекта

}

void queue(int data) { // Метод для добавления элемента в очередь

Node\* temp = new Node(); // Создание нового узла

temp->data = data; // Присваивание значения новому узлу

temp->next = nullptr; // Установка следующего указателя на nullptr

if (head == nullptr) { // Если очередь пуста

head = temp; // Новый узел становится головой

}

else {

tail->next = temp; // Новый узел добавляется в конец очереди

}

tail = temp; // Обновление указателя на последний узел

size++; // Увеличение размера очереди

}

int unqueue() { // Метод для извлечения элемента из очереди

if (head == nullptr) { // Если очередь пуста

return 0; // Возвращаем ноль

}

Node\* temp = head; // Сохранение указателя на первый узел

int data = temp->data; // Получение значения первого узла

head = head->next; // Перемещение головы на следующий узел

delete temp; // Освобождение памяти первого узла

size--; // Уменьшение размера очереди

return data; // Возвращение значения извлеченного элемента

}

void insertBeforeNegative() { // Метод для вставки 1 перед отрицательными элементами

Node\* curr = head; // Текущий указатель, начиная с головы

Node\* prev = nullptr; // Предыдущий указатель, начиная с nullptr

while (curr != nullptr) { // Проход по всем узлам очереди

if (curr->data < 0) { // Если текущий элемент отрицательный

Node\* temp = new Node(); // Создание нового узла со значением 1

temp->data = 1; // Присваивание единицы новому узлу

temp->next = curr; // Новый узел указывает на текущий

if (prev == nullptr) { // Если это первый отрицательный элемент

head = temp; // Новый узел становится головой

}

else {

prev->next = temp; // Иначе, новый узел вставляется перед текущим

}

prev = temp->next; // Перемещение указателя на предыдущий узел

}

else {

prev = curr; // Если элемент не отрицательный, обновляем предыдущий указатель

}

size++; // Увеличение размера очереди

curr = curr->next; // Переход к следующему узлу

}

}

// Удаление из очереди всех элементов с отрицательной информационной частью

void removeNegative() {

Node\* prev = nullptr; // Указатель на предыдущий узел

Node\* curr = head; // Указатель на текущий узел

while (curr != nullptr) { // Пока не достигнут конец очереди

if (curr->data < 0) { // Если текущий узел содержит отрицательное число

Node\* temp = curr; // Сохранение указателя на текущий узел

if (prev == nullptr) { // Если текущий узел является головой

head = head->next; // Перемещение головы на следующий узел

}

else { // Если текущий узел не является головой

prev->next = curr->next; // Пропуск текущего узла

}

if (curr == tail) { // Если текущий узел является хвостом

tail = prev; // Предыдущий узел становится хвостом

}

curr = curr->next; // Переход к следующему узлу

delete temp; // Удаление текущего узла

size--; // Уменьшение размера очереди

}

else { // Если текущий узел не содержит отрицательное число

prev = curr; // Текущий узел становится предыдущим

curr = curr->next; // Переход к следующему узлу

}

}

}

int count(int data) { // Метод для подсчета количества вхождений заданного значения

int count = 0; // Счетчик вхождений

Node\* curr = head; // Текущий указатель, начиная с головы

while (curr != nullptr) { // Проход по всем узлам очереди

if (curr->data == data) { // Если значение совпадает

count++; // Увеличение счетчика

}

curr = curr->next; // Переход к следующему узлу

}

return count; // Возвращение количества вхождений

}

void clear() { // Метод для очистки очереди

Node\* curr = head; // Текущий указатель, начиная с головы

while (curr != nullptr) { // Проход по всем узлам очереди

Node\* temp = curr; // Сохранение текущего узла

curr = curr->next; // Переход к следующему узлу

delete temp; // Освобождение памяти текущего узла

}

head = nullptr; // Установка головы в nullptr

tail = nullptr; // Установка хвоста в nullptr

size = 0; // Обнуление размера очереди

}

};

int main()

{

SetConsoleCP(1251); // Установка кодировки консоли для ввода

SetConsoleOutputCP(1251); // Установка кодировки консоли для вывода

Queue q; // Создание объекта очереди

string choice; // Переменная для хранения выбора пользователя

int data; // Переменная для хранения ввода пользователя

while (true) { // Цикл, пока пользователь не выберет выход

cin >> choice; // Ввод выбора пользователя

if (choice == "insert") {

cin >> data; // Ввод значения

q.queue(data); // Добавление элемента в очередь

}

else if (choice == "front") {

cout << q.unqueue() << endl; // Извлечение элемента из очереди

}

else if (choice == "insertBeforeNegative") { // Если выбрана вставка 1 перед отрицательными элементами

q.insertBeforeNegative();

}

else if (choice == "removeNegative") { // Если выбрано удаление отрицательных элементов

q.removeNegative();

}

else if (choice == "count") { // Если выбран подсчет вхождений

cin >> data; // Ввод значения

cout << q.count(data) << endl; // Вывод результата

}

else if (choice == "clear") { // Если выбрана очистка очереди

q.clear();

cout << "Очередь очищена" << endl; // Сообщение об успешной очистке

}

else if (choice == "stop") { // Если выбран выход

break;

}

else { // Если выбрано неверное значение

cout << "Неверный выбор, попробуйте еще раз" << endl; // Сообщение об ошибке

break;

}

}

}