**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра АПУ**

отчет

**по лабораторной работе № 4**

**по дисциплине «Программирование»**

**Тема: Шаблонная структура данных двухсвязного списка**

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

Санкт-Петербург

2024

## Постановка задачи.

Необходимо реализовать шаблонную структуру данных двухсвязанного списка List<T> (коллекция, обеспечивающая произвольный доступ к данным) для обработки последовательности данных о городах в состав которой входит:

* Название города
* Название региона
* Количество жителей

Данная структура должна обладать следующим набором функций:

* void add(T) - добавление элемента в конец списка
* void insert(int, T) - 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)
* T elementAt(int) - доступ к информационной части элемента в данной позиции (индексация от 0)
* int count() - возвращает количество элементов в коллекции
* 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 двухсвязанного списка. И обеспечивать реализацию следующего функционала:

* Вывод наименования регионов в упорядоченном по убыванию суммарной численности городского населения
* Удаление узлов, хранящих информацию о городах указанного региона

## Ход решения.

Данная программа реализует структуру данных двухсвязного списка, специализированного для хранения информации о городах, включая название города, регион и население. Основная структура List<T> обеспечивает базовые функции управления списком, такие как добавление, вставка, удаление и доступ к элементам списка, а также управление его размером.

Данный фрагмент кода (рис. 1.1) включает директивы предварительной обработки для подключения стандартных библиотек C++, которые обеспечивают функциональность ввода-вывода и работы со строками. Директива #include <iostream> подключает библиотеку, которая содержит определения для ввода и вывода, например std::cin и std::cout, используемые для чтения с консоли и записи в консоль соответственно. Директива #include <string> подключает библиотеку для работы со строками C++ типа std::string, позволяющая использовать различные функции для управления строками.

Также присутствует объявление using namespace std;, которое позволяет использовать элементы пространства имен std (стандартной библиотеки C++) без явного указания префикса std::. Это делает код более компактным и читаемым, позволяя обращаться к стандартным функциям и типам, таким как cin, cout и string, без необходимости добавления std:: перед ними.

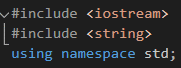


Рисунок 1.1 – Подключение библиотеки и пространство имён

Этот фрагмент кода (рис. 1.2) определяет структуру City, предназначенную для хранения информации о городе, включая его название, регион и население. В структуре также реализован конструктор, который инициализирует эти три поля. Конструктор принимает параметры для каждого поля с заданными значениями по умолчанию, что позволяет создавать объекты City с полными, частичными или отсутствующими данными, если при создании объекта аргументы не предоставлены. Значения по умолчанию для строковых полей — пустые строки, а для числового поля — ноль, что обеспечивает удобство и гибкость в использовании структуры.

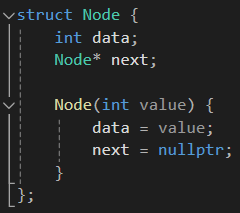


Рисунок 1.2 – Определение структуры City

Данный код (рис. 1.3) определяет шаблонный класс List, который реализует структуру данных двухсвязного списка. Шаблонный класс позволяет хранить данные любого типа T, делая структуру гибкой для различных применений. Внутри класса List определена вложенная структура Node, представляющая собой узел списка, который содержит данные типа T, а также указатели prev и next для связи с предыдущим и следующим узлами в списке. Конструктор Node инициализирует эти указатели как nullptr, что означает, что новый узел изначально не связан ни с какими другими узлами.

Класс List включает приватные члены head и tail, которые указывают на начало и конец списка соответственно, а также size, который отслеживает количество элементов в списке. Дополнительно, используются currentNode и currentIndex для оптимизации доступа к элементам списка: currentNode хранит ссылку на последний доступный узел, а currentIndex — его позицию, что ускоряет последующие операции доступа, особенно если они происходят близко к последнему использованному индексу.

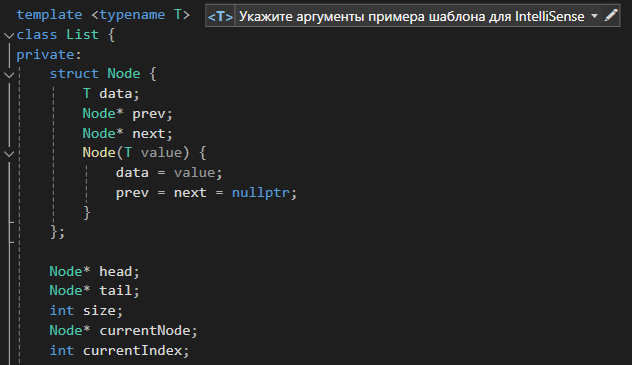


Рисунок 1.3 – Определение шаблонного класса List

Метод getNode (рис. 1.4) в классе List предназначен для получения узла по его индексу и является частью внутренней реализации двухсвязного списка. Он начинает с проверки валидности указанного индекса: если индекс меньше нуля или больше или равен количеству элементов в списке, метод возвращает nullptr, указывая на отсутствие узла по данному индексу. Если индекс равен нулю, метод сразу возвращает головной узел списка head, а если индекс равен индексу последнего элемента, возвращается хвостовой узел tail.

Для оптимизации доступа к элементам, если запрашиваемый индекс находится близко к последнему использованному индексу (хранится в currentIndex), метод выбирает направление движения от текущего узла (currentNode) в зависимости от того, больше или меньше запрашиваемый индекс по сравнению с текущим. Это уменьшает количество операций перемещения по списку, если последовательные доступы к элементам происходят в одной и той же области списка.

Если запрашиваемый индекс находится в первой половине списка, то поиск начинается с головы списка и двигается вперед, а если во второй половине — поиск ведется от хвоста списка назад. Эта логика обеспечивает наиболее эффективный доступ к элементам по индексу, минимизируя количество переходов от узла к узлу. В результате, после нахождения узла, метод обновляет currentNode и currentIndex, чтобы сохранить текущее положение в списке, что может ускорить последующие вызовы метода.

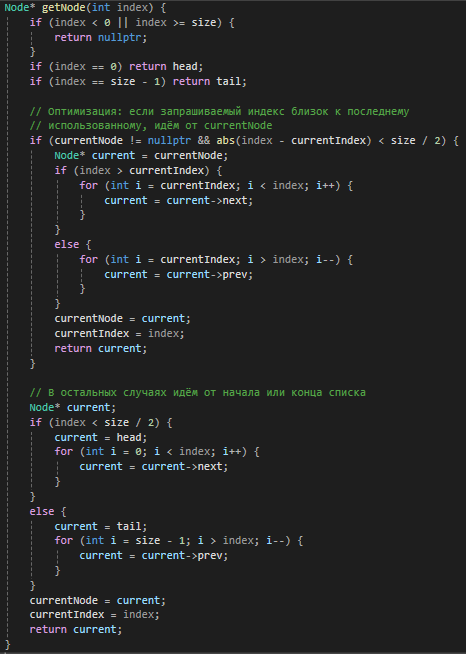


Рисунок 1.4 – Метод getNode для получения узла по его индексу

Конструктор List () (рис. 1.5) инициализирует класс, устанавливая все внутренние указатели (head, tail, currentNode) в nullptr, что указывает на то, что список изначально пуст. Также он устанавливает значения size и currentIndex в 0, подтверждая отсутствие элементов в списке и отсутствие последнего использованного индекса соответственно. Эта начальная настройка обеспечивает корректное состояние списка сразу после его создания, готовя его к безопасному добавлению элементов.

Деструктор ~List () вызывает метод clear(), который отвечает за освобождение всех ресурсов, используемых списком. Метод clear() последовательно удаляет все узлы списка, начиная с головы и двигаясь к хвосту, тем самым предотвращая утечки памяти, которые могут возникнуть, если бы элементы остались в динамической памяти после удаления списка. Использование clear() в деструкторе гарантирует, что все ресурсы, связанные со списком, будут корректно освобождены при его уничтожении, например, когда объект List выходит из области видимости или явно удаляется.

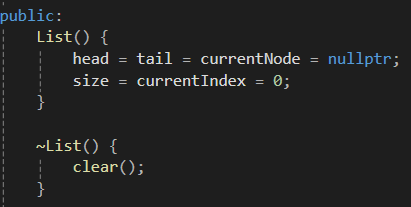


Рисунок 1.5 – Конструктор и деструктор класса List.

Метод add (рис. 1.6) предназначен для добавления нового элемента в конец двухсвязного списка, реализованного в классе List. Метод принимает данные обобщённого типа T, создаёт новый узел Node, инициализируя его этими данными. Если список пуст (что проверяется по head равному nullptr), новый узел становится одновременно и головой, и хвостом списка, так как это первый добавленный элемент. В противном случае, если в списке уже есть элементы, новый узел добавляется в конец списка: устанавливается связь next от текущего хвоста списка к новому узлу и обратная связь prev от нового узла к текущему хвосту. После этого, новый узел становится новым хвостом списка.

После добавления нового узла, счётчик размера списка size увеличивается на единицу, отражая добавление нового элемента. Это изменение обеспечивает актуализацию информации о количестве элементов в списке, что важно для корректной работы методов, зависящих от размера списка, таких как добавление, удаление или доступ к элементам по индексу. Этот метод является фундаментальным для работы со списком, так как позволяет динамически расширять коллекцию данных.

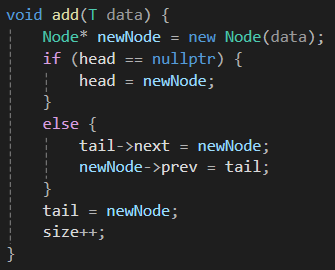


Рисунок 1.6 – Метод для добавления нового элемента в конец двухсвязного списка

Метод insert (рис. 1.7) предназначен для вставки нового элемента в произвольную позицию двухсвязного списка, представленного классом List. Он принимает целочисленный индекс и данные типа T, на основе которых создается новый узел. Если указанный индекс меньше нуля или больше текущего размера списка, метод завершается без изменений, обеспечивая защиту от вставки в недопустимые позиции.

Если индекс равен размеру списка, новый элемент добавляется в конец, используя метод add, который уже определен для этой операции. Это упрощает логику вставки в конец списка, переиспользуя существующий функционал.

В случае, когда индекс равен нулю, новый узел должен стать новой головой списка. Для этого узел связывается с текущей головой списка, и ссылки обновляются так, чтобы новый узел указывал на предыдущую голову, а предыдущая голова — на новый узел. Затем новый узел становится головой списка.

Для вставки в середину списка используется метод getNode, который извлекает узел по указанному индексу. Новый узел вставляется перед найденным узлом, обновляются ссылки так, чтобы новый узел связывался с предыдущим и следующим узлами корректно.

После вставки, currentNode и currentIndex обновляются для отражения позиции нового узла в списке, что улучшает производительность последующих операций доступа. Размер списка увеличивается на один, что поддерживает актуальность информации о количестве элементов.

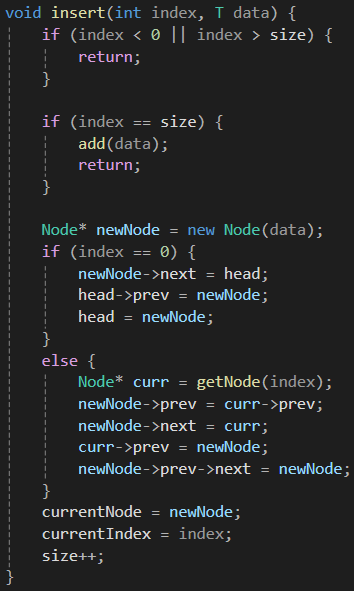


Рисунок 1.7 – Метод для удаления элемента

Метод removeAt (рис. 1.8) служит для удаления элемента из двухсвязного списка на указанной позиции по индексу. Прежде чем удалять элемент, метод проверяет корректность индекса: если он выходит за пределы допустимого диапазона (меньше нуля или больше либо равен текущему размеру списка), операция прерывается без изменений.

Если индекс равен нулю, удаляется головной элемент списка. Узел head перемещается на следующий элемент, и, если новый head не равен nullptr, его указатель prev обнуляется, чтобы разорвать связь с удаленным узлом. Если новый головной узел не найден (список стал пустым), то хвостовой узел также обнуляется, как и указатели на текущий узел и индекс.

Когда удаляется последний элемент списка (если индекс равен последнему индексу), удаляется хвостовой элемент. Указатель tail перемещается на предыдущий узел, и его указатель next обнуляется.

Если удаление происходит в середине списка, метод getNode используется для получения узла по индексу, который необходимо удалить. Затем связи соседних узлов обновляются: предыдущий узел связывается с последующим, а последующий — с предыдущим.

После удаления узла проверяется, является ли удаленный узел последним использованным (currentNode). Если это так, currentNode устанавливается на предыдущий узел, а индекс обновляется на максимум между нулем и index - 1. Если удаленный индекс меньше текущего индекса, текущий индекс уменьшается на один, чтобы корректно отражать положение в списке.

В завершение удаляемый узел освобождается из памяти с помощью delete, и размер списка уменьшается на единицу, чтобы поддерживать актуальные данные о количестве элементов.

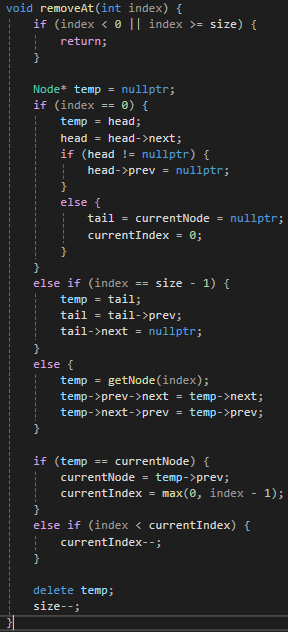


Рисунок 1.8 – Метод для удаления элемента из двухсвязного списка на указанной позиции по индексу

Метод elementAt (рис. 1.9) обеспечивает доступ к элементу списка по указанному индексу. Для начала он проверяет, находится ли индекс в допустимом диапазоне: если индекс меньше нуля или больше или равен текущему размеру списка, вызывается исключение out\_of\_range, сигнализирующее о попытке обращения к несуществующему элементу.

Если указанный индекс совпадает с текущим индексом, и текущий узел (currentNode) не равен nullptr, метод возвращает данные из этого узла, что оптимизирует доступ при последовательном обращении к соседним элементам списка.

В противном случае вызывается метод getNode, который находит и возвращает указатель на узел по заданному индексу. Метод elementAt извлекает данные из этого узла и возвращает их в качестве результата.

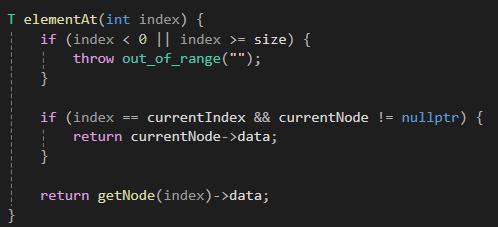


Рисунок 1.9 – Метод, который обеспечивает доступ к элементу списка по указанному индексу

Метод count (рис. 1.10) предназначен для получения количества элементов в двухсвязном списке, реализованном в классе List. Этот метод просто возвращает значение переменной size, которая хранит текущий размер списка и постоянно обновляется в других методах класса при добавлении, удалении или очистке узлов.

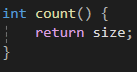


Рисунок 1.10 – Метод для получения количества элементов

Метод swappedAt (рис. 1.11) предназначен для замены данных в узле списка по заданному индексу новыми данными. Он принимает целочисленный индекс и новые данные типа T. Сначала с помощью метода getNode происходит поиск узла по указанному индексу. После получения указателя на этот узел его поле data, которое хранит текущие данные, заменяется значением, переданным в метод.

Этот метод позволяет изменить содержимое конкретного узла без необходимости удаления и повторного добавления элемента.

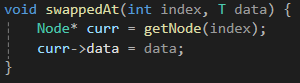


Рисунок 1.11 – Метод для замены данных в узле списка по заданному индексу новыми данными

Метод clear (рис. 1.12) предназначен для очистки двухсвязного списка, освобождая всю занятую память и сбрасывая его состояние до пустого. Процесс начинается с установки указателя current на первый элемент списка (голову). Затем в цикле, пока узел current не равен nullptr, метод сохраняет указатель на следующий узел в переменную next, удаляет текущий узел с помощью оператора delete, после чего перемещает указатель current на следующий узел. Это повторяется до тех пор, пока не будут освобождены все узлы в списке.

После завершения цикла указатели head и tail устанавливаются в nullptr, сигнализируя, что список теперь пуст. Размер списка (size) также сбрасывается до нуля, что отражает отсутствие элементов.

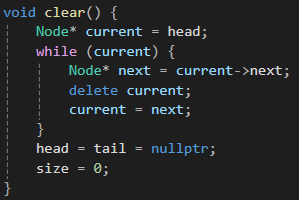


Рисунок 1.12 – Метод для очистки двухсвязного списка

Функция printList (рис. 1.13) предназначена для вывода на экран всех элементов списка городов, представленного структурой City. Она принимает в качестве параметра объект типа List <City> и с помощью цикла проходит по каждому элементу списка. Для этого используется метод count для определения количества элементов и метод elementAt для получения конкретного элемента по индексу.

На каждом шаге цикла данные текущего города, такие как название (name), регион (region) и население (population), извлекаются из объекта City и выводятся на экран в форматированном виде с помощью стандартного потока вывода cout.

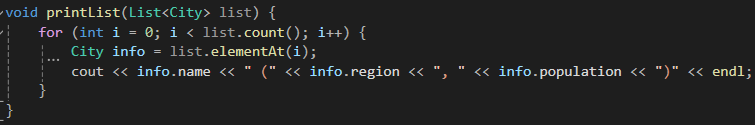


Рисунок 1.13 – Функция для вывода на экран всех элементов списка городов

Функция removeCitiesFromRegion (рис. 1.14) удаляет из списка все города, принадлежащие определенному региону. Она принимает строку regionName, представляющую название региона, и ссылку на список городов типа List <City>. Изначально устанавливается индекс i в ноль, чтобы начать обход списка с его начала.

Функция выполняет цикл, пока значение i меньше общего количества городов в списке. Для каждого города проверяется, совпадает ли его поле region с именем переданного региона regionName. Если совпадает, город удаляется с помощью метода removeAt, и индекс i не изменяется, так как после удаления на место удаленного элемента смещается следующий узел. В противном случае, если город принадлежит другому региону, индекс i увеличивается, чтобы перейти к следующему элементу.

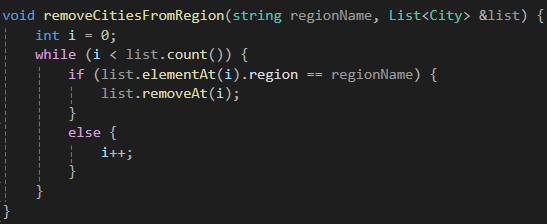


Рисунок 1.14 – Функция, которая удаляет из списка все города, принадлежащие определенному региону

Функция printRegionPopulation (рис. 1.15) выводит на экран регионы в порядке убывания общего числа жителей всех городов в каждом регионе. Для начала создается временный список temp, который будет использоваться для хранения информации о населении регионов. Затем выполняется цикл по всему исходному списку list, и каждый город рассматривается в контексте его региона. Внутри вложенного цикла проверяется, существует ли уже регион этого города в списке temp. Если регион найден, к его населению добавляется население текущего города, а обновленный элемент заменяется в списке. Если региона еще нет в temp, город добавляется целиком.

После заполнения временного списка выполняется сортировка регионов по убыванию населения. Для этого вложенный цикл сравнивает текущий регион с последующим, и, если население текущего меньше, они меняются местами с помощью метода swappedAt.

В завершение функция выводит все регионы и их суммарное население, используя стандартный поток вывода cout, упорядочив их по численности населения в порядке убывания.



Рисунок 1.15 – Функция, которая выводит на экран регионы в порядке убывания общего числа жителей всех городов в каждом регионе

Функция main (рис. 1.16) реализует пользовательский интерфейс для управления списком городов, используя цикл для приема и обработки команд, вводимых пользователем. В начале создается объект List <City>, предназначенный для хранения информации о городах. Переменные population, index, name, region и choice используются для обработки вводимых данных.

Функция запускает бесконечный цикл, в котором считывается команда пользователя с помощью cin в переменную choice. Затем на основании этой команды выполняются различные операции над списком cityList.

Если команда add, пользователь вводит данные о городе, включая название, регион и население, после чего город добавляется в конец списка. Команда insert требует указания индекса, куда будет добавлен новый город с введенными данными. Команда removeAt удаляет город по заданному индексу, а команда elementAt выводит информацию о городе по указанному индексу, обрабатывая исключения в случае некорректного ввода. Команда count выводит общее количество городов в списке.

Команда removeCitiesFromRegion удаляет все города определенного региона, а команда printRegionPopulation выводит все регионы, отсортированные по убыванию населения. Команда print выводит полный список городов с их данными, а clear очищает весь список. Команда stop завершает цикл и программу. В остальных случаях программа завершает выполнение, выходя из цикла.



Рисунок 1.16 – Функция main

**Выводы.**

В ходе выполнения лабораторной работы была разработана программа, реализующая структуру данных двухсвязного списка `List`, предназначенную для обработки информации о городах. Эта структура обеспечивает произвольный доступ к данным и включает в себя набор функций для управления элементами списка, таких как добавление, вставка, удаление и доступ к элементам, а также подсчет их количества. Дополнительно реализованы функции для удаления городов по региону и вывода списка регионов в порядке убывания суммарной численности городского населения. 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

// Lab4.cpp : Этот файл содержит функцию "main". Здесь начинается и заканчивается выполнение программы.

//

#include <iostream>

#include <string>

using namespace std;

struct City {

string name;

string region;

int population;

City(string city = "", string nameRegion = "", int populationK = 0) {

name = city;

region = nameRegion;

population = populationK;

}

};

template <typename T>

class List {

private:

struct Node {

T data;

Node\* prev;

Node\* next;

Node(T value) {

data = value;

prev = next = nullptr;

}

};

Node\* head;

Node\* tail;

int size;

Node\* currentNode;

int currentIndex;

Node\* getNode(int index) {

if (index < 0 || index >= size) {

return nullptr;

}

if (index == 0) return head;

if (index == size - 1) return tail;

// Оптимизация: если запрашиваемый индекс близок к последнему

// использованному, идём от currentNode

if (currentNode != nullptr && abs(index - currentIndex) < size / 2) {

Node\* current = currentNode;

if (index > currentIndex) {

for (int i = currentIndex; i < index; i++) {

current = current->next;

}

}

else {

for (int i = currentIndex; i > index; i--) {

current = current->prev;

}

}

currentNode = current;

currentIndex = index;

return current;

}

// В остальных случаях идём от начала или конца списка

Node\* current;

if (index < size / 2) {

current = head;

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

current = current->next;

}

}

else {

current = tail;

for (int i = size - 1; i > index; i--) {

current = current->prev;

}

}

currentNode = current;

currentIndex = index;

return current;

}

public:

List() {

head = tail = currentNode = nullptr;

size = currentIndex = 0;

}

~List() {

clear();

}

void add(T data) {

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

}

else {

tail->next = newNode;

newNode->prev = tail;

}

tail = newNode;

size++;

}

void insert(int index, T data) {

if (index < 0 || index > size) {

return;

}

if (index == size) {

add(data);

return;

}

Node\* newNode = new Node(data);

if (index == 0) {

newNode->next = head;

head->prev = newNode;

head = newNode;

}

else {

Node\* curr = getNode(index);

newNode->prev = curr->prev;

newNode->next = curr;

curr->prev = newNode;

newNode->prev->next = newNode;

}

currentNode = newNode;

currentIndex = index;

size++;

}

void removeAt(int index) {

if (index < 0 || index >= size) {

return;

}

Node\* temp = nullptr;

if (index == 0) {

temp = head;

head = head->next;

if (head != nullptr) {

head->prev = nullptr;

}

else {

tail = currentNode = nullptr;

currentIndex = 0;

}

}

else if (index == size - 1) {

temp = tail;

tail = tail->prev;

tail->next = nullptr;

}

else {

temp = getNode(index);

temp->prev->next = temp->next;

temp->next->prev = temp->prev;

}

if (temp == currentNode) {

currentNode = temp->prev;

currentIndex = max(0, index - 1);

}

else if (index < currentIndex) {

currentIndex--;

}

delete temp;

size--;

}

T elementAt(int index) {

if (index < 0 || index >= size) {

throw out\_of\_range("");

}

if (index == currentIndex && currentNode != nullptr) {

return currentNode->data;

}

return getNode(index)->data;

}

int count() {

return size;

}

void swappedAt(int index, T data) {

Node\* curr = getNode(index);

curr->data = data;

}

void clear() {

Node\* current = head;

while (current) {

Node\* next = current->next;

delete current;

current = next;

}

head = tail = nullptr;

size = 0;

}

};

void printList(List<City> list) {

for (int i = 0; i < list.count(); i++) {

City info = list.elementAt(i);

cout << info.name << " (" << info.region << ", " << info.population << ")" << endl;

}

}

void removeCitiesFromRegion(string regionName, List<City> &list) {

int i = 0;

while (i < list.count()) {

if (list.elementAt(i).region == regionName) {

list.removeAt(i);

}

else {

i++;

}

}

}

void printRegionPopulation(List<City> list) {

List<City> temp;

for (int i = 0; i < list.count(); i++) {

City listCity = list.elementAt(i);

bool regionInList = false;

for (int j = 0; j < temp.count(); j++) {

City tempCity = temp.elementAt(j);

if (tempCity.region == listCity.region) {

tempCity.population += listCity.population;

temp.swappedAt(j, tempCity);

regionInList = true;

break;

}

}

if (!regionInList)

temp.add(listCity);

}

for (int i = 0; i < temp.count(); i++) {

for (int j = 0; j < temp.count() - 1; j++) {

City curr = temp.elementAt(j);

City next = temp.elementAt(j + 1);

if (curr.population < next.population) {

temp.swappedAt(i, next);

temp.swappedAt(i + 1, curr);

}

}

}

for (int i = 0; i < temp.count(); i++) {

cout << temp.elementAt(i).region << " " << temp.elementAt(i).population << endl;

}

}

int main()

{

List<City> cityList;

int population, index;

string name, region, choice;

while (true) {

cin >> choice;

if (choice == "add") {

cin >> name;

cin >> region;

cin >> population;

cityList.add(City(name, region, population));

}

else if (choice == "insert") {

cin >> index;

cin >> name;

cin >> region;

cin >> population;

cityList.insert(index, City(name, region, population));

}

else if (choice == "removeAt") {

cin >> index;

cityList.removeAt(index);

}

else if (choice == "elementAt") {

cin >> index;

try {

City info = cityList.elementAt(index);

cout << info.name << " (" << info.region << ", " << info.population << ")" << endl;

}

catch (out\_of\_range) {}

}

else if (choice == "count") {

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

}

else if (choice == "removeCitiesFromRegion") {

cin >> region;

removeCitiesFromRegion(region, cityList);

}

else if (choice == "printRegionPopulation") {

printRegionPopulation(cityList);

}

else if (choice == "print") {

printList(cityList);

}

else if (choice == "clear") {

cityList.clear();

}

else if (choice == "stop") {

break;

}

else {

break;

}

}

}

Файл data1-1.txt:

add 10

add 20

add 15

count

remove 20

count

stop

Файл report1-1.txt:

10

20

15

3

2

Файл data2-1.txt:

add 30

add 40

add 35

count

remove 30

count

clear

count

stop

Файл report2-1.txt:

30

40

35

3

2

0

Файл data3-1.txt:

add 50

add 60

add 70

add 80

count

remove 60

remove 70

count

stop

Файл report3-1.txt:

50

60

70

80

4

2