|  |  |
| --- | --- |
| **Российский университет транспорта (МИИТ)**  **Институт транспортной техники и систем управления**  **Кафедра «Управление и защита информации»** | |
| **Отчет**  **по практическому заданию**  **по теме «Двусвязный список»**  **по дисциплине «Системы управления базами данных»** | |
|  | Выполнил:  Студент группы ТКИ-441  Шибзухова Д.А. |
|  | Проверил:  к.т.н. доцент  Васильева М.А. |
| Москва 2023 | |

Оглавление

[Оглавление 2](#_Toc153830922)

[1 Задание 3](#_Toc153830923)

[2 Описание структуры – Двусвязного списка 4](#_Toc153830924)

[2.1 UML-диаграмма 4](#_Toc153830925)

[2.2 Листинг кода 5](#_Toc153830927)

[2.2.1 List.h 5](#_Toc153830928)

[2.2.2 Main.cpp 12](#_Toc153830929)

[2.2.3 test.cpp 12](#_Toc153830930)

[3 Тестирование программы 17](#_Toc153830931)

[4. Заключение 18](#_Toc153830932)

1. Задание

Разработать структуру данных на языке программирования С++ в ООП парадигме. Реализовать библиотеку классов, обеспечить выполнение операций CRUD (create, read, update, delete) для любого типа данных (template). Переопределить операторы сдвига влево и вправо для возможности чтения и записи из любого потока (консоль, строка, файл). Ни один метод не должен зависеть от консоли или файла, используйте представление в строку. Написать итератор для работы с коллекцией (разработанной структурой данных). По возможности использовать умные указатели (smart pointer), заместо «сырых» (raw). Использование стандартных структур данных возможно в качестве вспомогательных, например, при обходе дерева можно использовать std::vector. Для избежания недоразумений, лучше обсудить использование с преподавателем индивидуально.

Все методы покрыть тестами (модульное тестирование).

Выбранная структура данных: Двусвязный список

1. Описание структуры – Двусвязного списка
   1. UML-диаграмма

На рисунке 1 представлена UML-диаграмма, показывающая связь классов в проекте.

|  |
| --- |
|  |
| 1. – UML-диаграмма |

* 2. Листинг кода
     1. List.h

#pragma once

#include <iostream>

#include <stdexcept>

#include <limits>

#include <vector>

/\*

\* @brief Структура для реализации класса List, в которой хранится информации об элементе, а также ссылки на предыдущий и следующий элементы.

\*/

template <typename T>

struct Element {

T data;

Element\* previous;

Element\* next;

};

/\*

\* @brief Класс List. Представляется из себя двусвязный ациклический список.

\*/

template <typename T>

class List {

private:

Element<T>\* head;

Element<T>\* tail;

size\_t size;

public:

/\*

\* @breif Конструктор по умолчанию.

\*/

List();

/\*

\* @brief Метод, позволяющий получить список в виде вектора.

\* @return Список, в виде вектора T.

\*/

std::vector<T> getListAsVector() const;

/\*

\* @brief Метод, позволяющий получить размер списка.

\* @return Размер списка.

\*/

size\_t getSize() const;

/\*

\* @brief Метод, позволяющий добавить элемент в конец списка.

\* @param newData - информация о новом элементе

\*/

void pushBack(T newData);

/\*

\* @brief Метод, позволяющий добавить элемент в начало списка.

\* @param newData - информация о новом элементе.

\*/

void pushFront(T newData);

/\*

\* @brief Метод, позволяющий получить информацию о первом элементе списка.

\* @return Информация о первом элементе.

\*/

T getFront() const;

/\*

\* @brief Метод, позволяющий получить информацию о последнем элементе списка.

\* @return Информация о последнем элементе.

\*/

T getBack() const;

/\*

\* @brief Метод, удаляющий первый элемент списка.

\*/

void popFront();

/\*

\* @brief Метод, удаляющий последний элемент списка.

\*/

void popBack();

/\*

\* @brief Метод, удаляющий элемент списка по ключу.

\* @param position - ключ (позиция элемента в списке).

\*/

void deleteELement(size\_t position);

/\*

\* @brief Метод, позволяющий вставить элемент в список.

\* @param data - информация об элементе (его размер).

\* @param position - позиция, на которую нужно элемент поставить.

\*/

void insertElement(T data, size\_t position);

/\*

\* @brief Метод, позволяющий получить данные об элементе списка.

\* @param position - позиция элемента в списке.

\* @return Информация об элементе - его размер.

\*/

T getElement(size\_t position) const;

/\*

\* @brief Метод, позволяющий поменять местами значения 2 элементов списка.

\* @param positionOne - позиция первого элемента.

\* @param positionTwo - позиция второго элемента.

\*/

void swap(size\_t positionOne, size\_t positionTwo);

/\*

\* @brief Метод, удаляющий все элементы списка.

\*/

void deleteAll();

/\*

\* @brief Конструктор копирования.

\*/

List(const List& other);

/\*

\* @brief Оператор копирования.

\*/

List& operator = (const List& other);

/\*

\* @brief Конструктор перемещения.

\*/

List(List&& other) noexcept;

/\*

\* @brief Оператор перемещения

\*/

List& operator = (List&& other) noexcept;

/\*

\* @brief Деструктор

\*/

~List();

};

template <typename T>

List<T>::List() : head(nullptr), tail(nullptr), size(0) {};

template <typename T>

List<T>::List(const List& other) : List()

{

Element<T>\* temp = other.head;

while (temp != nullptr)

{

this->pushBack(temp->data);

temp = temp->next;

}

this->size = other.size;

}

template <typename T>

List<T>& List<T>::operator = (const List& other) {

if (this == &other)

return \*this;

List<T> copy(other);

std::swap(this->tail, copy.tail);

std::swap(this->head, copy.head);

std::swap(this->size, copy.size);

return \*this;

}

template <typename T>

size\_t List<T>::getSize() const{

return this->size;

}

template <typename T>

std::vector<T> List<T>::getListAsVector() const {

std::vector<T> out;

auto temp = this->head;

while (temp != nullptr) {

out.push\_back(temp->data);

temp = temp->next;

}

return out;

}

template <typename T>

void List<T>::pushBack(T newData) {

Element<T>\* temp = new Element<T>();

temp->data = newData;

temp->next = nullptr;

if (this->getSize() == 0) {

temp->previous = nullptr;

this->head = temp;

this->tail = temp;

}

else {

temp->previous = this->tail;

tail->next = temp;

this->tail = temp;

}

++this->size;

}

template <typename T>

void List<T>::pushFront(T newData) {

Element<T>\* temp = new Element<T>();

temp->data = newData;

temp->previous = nullptr;

if (this->getSize() == 0) {

temp->next = nullptr;

this->head = temp;

this->tail = temp;

}

else {

temp->next = this->head;

head->previous = temp;

this->head = temp;

}

++this->size;

}

template <typename T>

T List<T>::getFront() const {

return this->head->data;

}

template <typename T>

T List<T>::getBack() const {

return this->tail->data;

}

template <typename T>

void List<T>::popBack() {

if (this->getSize() == 0)

throw std::logic\_error("Нельзя удалять элементы в пустом листе");

if (this->getSize() == 1) {

delete this->tail;

this->head = nullptr;

this->tail = nullptr;

--this->size;

return;

}

else if (this->getSize() == 2) {

delete this->tail;

this->head->next = nullptr;

this->tail = this->head;

--this->size;

return;

}

auto temp = this->tail;

this->tail = this->tail->previous ;

delete temp;

--this->size;

}

template <typename T>

void List<T>::popFront() {

if (this->getSize() == 0)

throw std::logic\_error("Нельзя удалять элементы в пустом листе");

if (this->getSize() == 1) {

delete this->head;

this->head = nullptr;

this->tail = nullptr;

--this->size;

return;

}

else if (this->getSize() == 2) {

delete this->head;

this->tail->previous = nullptr;

this->head = this->tail;

--this->size;

return;

}

auto temp = this->head;

this->head = this->head->next;

delete temp;

--this->size;

}

template <typename T>

void List<T>::deleteELement(size\_t position) {

if (position >= this->getSize())

throw std::out\_of\_range("Позиция больше размера списка");

if (position == 0) {

this->popFront();

return;

}

else if (position == this->getSize() - 1) {

this->popBack();

return;

}

auto temp = this->head;

for (size\_t i = 0; i < position; i++)

{

temp = temp->next;

}

temp->previous->next = temp->next;

temp->next->previous = temp->previous;

delete temp;

--this->size;

}

template <typename T>

void List<T>::insertElement(T data, size\_t position) {

if (position > this->getSize())

throw std::out\_of\_range("Позиция больше размера списка");

if (position == 0) {

this->pushFront(data);

return;

}

else if (position == this->getSize()) {

this->pushBack(data);

return;

}

auto insert = this->head;

for (size\_t i = 0; i < position; i++)

{

insert = insert->next;

}

Element<T>\* temp = new Element<T>();

temp->data = data;

if (insert->previous != nullptr && this->getSize() != 1)

insert->previous->next = temp;

temp->next = insert;

temp->previous = insert->previous;

insert->previous = temp;

++this->size;

}

template <typename T>

T List<T>::getElement(size\_t position) const{

if (position >= this->getSize())

throw std::out\_of\_range("Позиция больше размера списка");

auto get = this->head;

for (size\_t i = 0; i < position; i++)

{

get = get->next;

}

return get->data;

}

template <typename T>

void List<T>::swap(size\_t positionOne, size\_t positionTwo) {

if (positionOne > this->getSize() || positionTwo > this->getSize())

throw std::out\_of\_range("Позиция больше размера списка");

auto tempOne = this->head;

auto tempTwo = this->head;

if (positionOne >= positionTwo) {

for (size\_t i = 0; i < positionTwo; i++)

{

tempOne = tempOne->next;

tempTwo = tempTwo->next;

}

for (size\_t i = positionTwo; i < positionOne; i++)

{

tempOne = tempOne->next;

}

}

else {

for (size\_t i = 0; i < positionOne; i++)

{

tempOne = tempOne->next;

tempTwo = tempTwo->next;

}

for (size\_t i = positionOne; i < positionTwo; i++)

{

tempTwo = tempTwo->next;

}

}

std::swap(tempOne->data, tempTwo->data);

}

template <typename T>

void List<T>::deleteAll() {

while (this->getSize() != 0)

this->popBack();

}

template <typename T>

List<T>::~List() {

this->deleteAll();

}

template <typename T>

List<T>::List(List&& other) noexcept : List()

{

\*this = other;

}

template <typename T>

List<T>& List<T>::operator = (List&& other) noexcept {

std::swap(this->tail, other.tail);

std::swap(this->head, other.head);

std::swap(this->size, other.size);

return \*this;

}

template <typename T>

std::ostream& operator << (std::ostream& os, const List<T>& list) {

std::vector<T> out = list.getListAsVector();

for (size\_t i = 0; i < out.size(); i++)

{

os << out[i] << " ";

}

return os;

}

* + 1. Main.cpp

#include "List.h"

using namespace std;

/\*

\* @brief Точка входа в программу. В ней представлен пример работы переопределённого оператора <<.

\*/

int main() {

List<int> l1;

l1.pushBack(1);

l1.pushBack(5);

l1.pushBack(10);

cout << l1;

return 0;

}

* + 1. test.cpp

#include "pch.h"

#include "../List/List.h"

#include <string>

using namespace std;

class ListTest : public testing::Test {

protected:

List<int> list1;

List<string> list2;

std::vector<int> vector1 = { 3, 7 };

std::vector<string> vector2 = { "rutik" };

void SetUp() override {

list1.pushBack(3);

list1.pushBack(11);

list2.pushFront("miitik");

}

};

TEST\_F(ListTest, CopyConstructorIntWorks) {

List<int> list3 = List<int>(list1);

auto v = list1.getListAsVector();

for (size\_t i = 0; i < vector1.size(); i++)

{

EXPECT\_EQ(v[i], vector1[i]);

}

}

TEST\_F(ListTest, CopyConstructorStringWorks) {

List <string> list3 = List<string>(list2);

auto v = list3.getListAsVector();

for (size\_t i = 0; i < vector2.size(); i++)

{

EXPECT\_EQ(v[i], vector2[i]);

}

}

TEST\_F(ListTest, PushBackIntWorks) {

list1.pushBack(10);

vector1.push\_back(10);

auto v = list1.getListAsVector();

for (size\_t i = 0; i < vector1.size(); i++)

{

EXPECT\_EQ(v[i], vector1[i]);

}

}

TEST\_F(ListTest, PushBackStringWorks) {

list2.pushBack("tki");

vector2.push\_back("super");

auto v = list2.getListAsVector();

for (size\_t i = 0; i < vector2.size(); i++)

{

EXPECT\_EQ(v[i], vector2[i]);

}

}

TEST\_F(ListTest, PushFrontIntWorks) {

list1.pushFront(10);

std::vector<int> vector3 = { 22, 6, 9 };

auto v = list1.getListAsVector();

for (size\_t i = 0; i < vector3.size(); i++)

{

EXPECT\_EQ(v[i], vector3[i]);

}

}

TEST\_F(ListTest, PushFrontStringWorks) {

list2.pushFront("tutti");

std::vector<string> vector3 = { "tutti", "frutti" };

auto v = list2.getListAsVector();

for (size\_t i = 0; i < vector3.size(); i++)

{

EXPECT\_EQ(v[i], vector3[i]);

}

}

TEST\_F(ListTest, PopFrontThrowsLogicError) {

List <int> list3;

EXPECT\_THROW(list3.popFront(), logic\_error);

}

TEST\_F(ListTest, PopFrontWorks ) {

list1.popFront();

EXPECT\_EQ(list1.getSize(), 1);

EXPECT\_EQ(list1.getListAsVector()[0], 5);

list2.popFront();

EXPECT\_EQ(list2.getSize(), 0);

}

TEST\_F(ListTest, PopBackThrowsLogicError) {

List <int> list3;

EXPECT\_THROW(list3.popBack(), logic\_error);

}

TEST\_F(ListTest, PopBackWorks) {

list1.popBack();

EXPECT\_EQ(list1.getSize(), 1);

EXPECT\_EQ(list1.getListAsVector()[0], 1);

list2.popBack();

EXPECT\_EQ(list2.getSize(), 0);

}

TEST\_F(ListTest, DeleteElementThrowsOutOfRange) {

EXPECT\_THROW(list1.deleteELement(4), out\_of\_range);

}

TEST\_F(ListTest, DeleteElementWorks) {

list1.pushFront(10);

list1.deleteELement(1);

std::vector<int> vector3 = { 10, 5 };

auto v = list1.getListAsVector();

for (size\_t i = 0; i < vector3.size(); i++)

{

EXPECT\_EQ(v[i], vector3[i]);

}

}

TEST\_F(ListTest, InsertElementThrowsOutOfRange) {

EXPECT\_THROW(list1.insertElement(14, 4), out\_of\_range);

}

TEST\_F(ListTest, InserElementIntWorks) {

list1.insertElement(3, 1);

auto v = list1.getListAsVector();

std::vector<int> vector3 = { 2, 4, 6 };

for (size\_t i = 0; i < vector3.size(); i++)

{

EXPECT\_EQ(v[i], vector3[i]);

}

}

TEST\_F(ListTest, InsertElementStringWorks) {

list2.pushBack("rut");

list2.insertElement("---", 3);

std::vector<string> vector3 = { "kuku", "---", "rut" };

auto v = list2.getListAsVector();

for (size\_t i = 0; i < vector3.size(); i++)

{

EXPECT\_EQ(v[i], vector3[i]);

}

}

TEST\_F(ListTest, GetFrontIntWorks) {

EXPECT\_EQ(list1.getFront(), 4);

}

TEST\_F(ListTest, GetFrontStringWorks) {

EXPECT\_EQ(list2.getFront(), "qugu");

}

TEST\_F(ListTest, GetBackIntWorks) {

EXPECT\_EQ(list1.getBack(), 7);

}

TEST\_F(ListTest, GetBackStringWorks) {

EXPECT\_EQ(list2.getBack(), "kukareku");

}

TEST\_F(ListTest, GetElementThrowsOutOfRange) {

EXPECT\_THROW(list1.deleteELement(4), out\_of\_range);

}

TEST\_F(ListTest, GetElementIntWorks) {

list1.pushBack(10);

list1.pushBack(15);

EXPECT\_EQ(list1.getElement(2), 10);

EXPECT\_EQ(list1.getElement(3), 15);

EXPECT\_EQ(list1.getElement(0), 1);

}

TEST\_F(ListTest, GetElementStringWorks) {

list2.pushBack("---");

list2.pushBack("ruty");

EXPECT\_EQ(list2.getElement(1), "---");

}

TEST\_F(ListTest, SwapThrowsOutOfRange) {

EXPECT\_THROW(list1.swap(1, 3), out\_of\_range);

}

TEST\_F(ListTest, SwapIntWorks) {

list1.pushBack(10);

list1.pushBack(20);

list1.swap(0, 3);

list1.swap(1, 2);

auto v = list1.getListAsVector();

std::vector<int> vector3 = { 10, 11, 7, 3 };

for (size\_t i = 0; i < vector3.size(); i++)

{

EXPECT\_EQ(v[i], vector3[i]);

}

}

TEST\_F(ListTest, SwapStringWorks) {

list2.pushBack("rty");

list2.swap(0, 1);

auto v = list2.getListAsVector();

std::vector<string> vector3 = { "qwerty","sad" };

for (size\_t i = 0; i < vector3.size(); i++)

{

EXPECT\_EQ(v[i], vector3[i]);

}

}

TEST\_F(ListTest, DeleteAllWorks) {

EXPECT\_NO\_FATAL\_FAILURE(list1.deleteAll());

EXPECT\_EQ(list1.getSize(), 0);

}

1. Тестирование программы

Чтобы оценить, что программа работает верно, её нужно протестировать. Результаты работы тестов показаны на рисунке 3.

|  |
| --- |
|  |
| 1. – Результаты работы тестов |

4. Заключение

В ходе задания была изучена теория, связанная с двусвязными списками, были освоены методы работы с ними.

Итогом выполнения задания стала реализация структуры двусвязный список на языке программирования C++.