ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«РОССИЙСКИЙ УНИВЕРСИТЕТ ТРАНСПОРТА»  
(РУТ (МИИТ))

Институт транспортной техники и систем управления

Кафедра «Управление и защита информации»

ОТЧЁТ  
О ПРАКТИЧЕСКОЙ РАБОТЕ № 3

По дисциплине «Языки программирования»

Выполнил: ст. гр. ТКИ – 241

Боди Итшан

Проверил: к.т.н., доц.

Васильева М. А.

Москва 2024

**Задание:** Повторите задание 2, используя интеллектуальные указатели

**Код программы:**

-------------------------------------------------------------------------

#pragma once

#include <memory>

/\*

\* @brief Define a struct for the Node of a double linked list.

\*/

struct Node

{

int value;

std::shared\_ptr<Node> next;

std::weak\_ptr<Node> previous;

/\*

\* @brief Constructor for Node, initializes the Node with a given value.

\* @param value The integer with which the Node will be initialized.

\*/

Node(int value);

};

#include "Node.h"

Node::Node(int value) : value{ value }, next{ nullptr } {}

**---------------------------------------------------------------------------**

#pragma once

#include <iostream>

#include "Node.h"

#include <initializer\_list>

#include <sstream>

/\*\*

\* @brief Class DoubleLinkedList.

\*/

class DoubleLinkedList {

std::shared\_ptr<Node> head;

std::shared\_ptr<Node> tail;

std::size\_t size;

public:

/\*\*

\* @brief Constructs a new empty linked list.

\*/

DoubleLinkedList();

/\*\*

\* @brief Constructs a new linked list initialized with the elements of the provided initializer list.

\* @param lst The initializer list of elements to initialize the linked list with.

\*/

DoubleLinkedList(std::initializer\_list<int> lst);

/\*\*

\* @brief Constructs a new linked list with the same elements as another linked list.

\* @param other The linked list to copy the elements from.

\*/

DoubleLinkedList(const DoubleLinkedList& other) = default;

/\*\*

\* @brief Move constructor for DoubleLinkedList.

\* @param other The linked list to move elements from.

\*/

DoubleLinkedList(DoubleLinkedList&& other) noexcept = default;

/\*\*

\* @brief Destroys the linked list and deallocates the memory used by the nodes.

\*/

~DoubleLinkedList();

/\*\*

\* @brief Removes all nodes from the linked list.

\*/

void clear();

/\*\*

\* @brief Checks if the linked list is empty.

\* @return true If the linked list is empty.

\* @return false If the linked list is not empty.

\*/

bool empty() const;

/\*\*

\* @brief Returns the number of nodes in the linked list.

\* @return size\_t The number of nodes in the linked list.

\*/

size\_t length() const;

/\*\*

\* @brief Adds a new node with the given data to the front of the linked list.

\* @param data The data to store in the new node.

\*/

void push\_front(int value);

/\*\*

\* @brief Adds a new node with the given data to the back of the linked list.

\* @param data The data to store in the new node.

\*/

void push\_back(int value);

/\*

@brief a function which delete a Node in front of List.

\*/

void pop\_front();

/\*

@brief a function which delete a Node in back of List.

\*/

void pop\_back();

/\*\*

\* @brief Returns a reference to the data stored in the node at the given index.

\* @param index The index of the node to retrieve the data from.

\* @return T& A reference to the data stored in the node at the given index.

\* @throws std::out\_of\_range If the index is out of bounds.

\*/

int& operator[](size\_t index);

/\*\*

\* @brief Returns a string representation of the linked list, with the elements separated by commas and enclosed in square brackets.

\* @return std::string The string representation of the linked list.

\*/

void print();

/\*\*

\* @brief Removes the node at the given index from the linked list.

\* @param index The index of the node to remove.

\* @throws std::out\_of\_range If the index is out of bounds.

\*/

void remove(size\_t index);

/\*\*

\* @brief Assignment Operator= for DoubleLinkedList.

\* @param other The linked list to copy the elements from.

\* @return DoubleLinkedList& A reference to the assigned linked list.

\*/

DoubleLinkedList& operator=(const DoubleLinkedList& other) = default;

/\*\*

\* @brief Move Assignment Operator= for DoubleLinkedList.

\* @param other The linked list to move elements from.

\* @return DoubleLinkedList& A reference to the assigned linked list.

\*/

DoubleLinkedList& operator=(DoubleLinkedList&& other) noexcept = default;

#include "doubleLinkedList.h"

DoubleLinkedList::DoubleLinkedList()

: head(nullptr), tail(nullptr), size(0) {}

DoubleLinkedList::DoubleLinkedList(std::initializer\_list<int> lst)

: DoubleLinkedList() {

for (auto& elem : lst) {

push\_back(elem);

}

}

DoubleLinkedList::~DoubleLinkedList() { clear(); }

void DoubleLinkedList::clear() {

while (head) {

head = head->next;

}

tail.reset();

size = 0;

}

bool DoubleLinkedList::empty() const {

return size == 0;

}

size\_t DoubleLinkedList::length() const {

return size;

}

void DoubleLinkedList::push\_front(int value) {

auto new\_node = std::make\_shared<Node>(value);

if (head == nullptr)

{

tail = new\_node;

}

else

{

head->previous = new\_node;

}

new\_node->next = head;

head = new\_node;

++size;

}

void DoubleLinkedList::push\_back(int value) {

auto new\_node = std::make\_shared<Node>(value);

if (head == nullptr)

{

head = new\_node;

}

else

{

tail->next = new\_node;

}

new\_node->previous = tail;

tail = new\_node;

++size;

}

void DoubleLinkedList::pop\_front()

{

if (head == nullptr)

{

return;

}

auto temp = head;

head = head->next;

--size;

};

void DoubleLinkedList::pop\_back()

{

if (nullptr == this->head)

{

return;

}

auto temp = this->tail;

this->tail = this->tail->previous.lock();

tail->next = nullptr;

--size;

};

int& DoubleLinkedList::operator[](size\_t index) {

auto curr = head.get();

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

if (!curr) throw std::out\_of\_range("Index out of bounds");

curr = curr->next.get();

}

if (!curr) throw std::out\_of\_range("Index out of bounds");

return curr->value;

}

void DoubleLinkedList::print() {

std::stringstream ss;

if (empty()) {

ss << "[]";

}

else {

ss << "[" << head->value;

auto curr = head->next;

while (curr) {

ss << ", " << curr->value;

curr = curr->next;

}

ss << "]";

}

std::cout << ss.str() << std::endl;

}

void DoubleLinkedList::remove(size\_t index) {

if (index == 0)

{

pop\_front();

}

else if (index == size - 1)

{

pop\_back();

}

else

{

auto temp = head;

std::size\_t i = 0;

while (i < index - 1)

{

temp = temp->next;

}

auto to\_delete = temp->next;

temp->next = to\_delete->next;

--size;

}

}

};

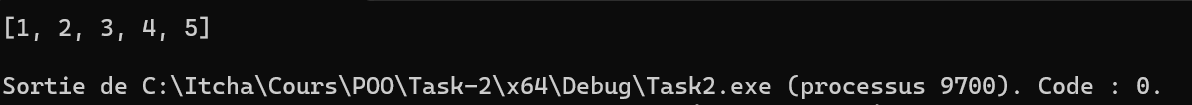


Рисунок 1 – прохождение задач

**Тесты:**

#include <gtest/gtest.h>

#include "DoubleLinkedList.h"

// Test fixture for the DoubleLinkedList class

class DoubleLinkedListTest : public ::testing::Test {

protected:

// Create a linked list object before each test

void SetUp() override {

list = DoubleLinkedList<int>();

}

DoubleLinkedList<int> list;

};

// Test the empty() function

TEST\_F(DoubleLinkedListTest, Empty) {

// Verify that the list is initially empty

EXPECT\_TRUE(list.empty());

// Add an element to the list

list.push\_back(42);

// Verify that the list is not empty after adding an element

EXPECT\_FALSE(list.empty());

}

// Test the length() function

TEST\_F(DoubleLinkedListTest, Length) {

// Verify that the initial length of the list is zero

EXPECT\_EQ(list.length(), 0);

// Add elements to the list

list.push\_back(1);

list.push\_back(2);

list.push\_back(3);

// Verify that the length of the list is updated correctly

EXPECT\_EQ(list.length(), 3);

}

// Test the push\_front() function

TEST\_F(DoubleLinkedListTest, PushFront) {

// Add elements to the front of the list

list.push\_front(3);

list.push\_front(2);

list.push\_front(1);

// Verify that the elements are inserted in the correct order

EXPECT\_EQ(list[0], 1);

EXPECT\_EQ(list[1], 2);

EXPECT\_EQ(list[2], 3);

}

// Test the push\_back() function

TEST\_F(DoubleLinkedListTest, PushBack) {

// Add elements to the back of the list

list.push\_back(1);

list.push\_back(2);

list.push\_back(3);

// Verify that the elements are inserted in the correct order

EXPECT\_EQ(list[0], 1);

EXPECT\_EQ(list[1], 2);

EXPECT\_EQ(list[2], 3);

}

// Test the operator[] function

TEST\_F(DoubleLinkedListTest, OperatorIndex) {

// Add elements to the list

list.push\_back(1);

list.push\_back(2);

list.push\_back(3);

// Verify that the operator[] returns the correct element at each index

EXPECT\_EQ(list[0], 1);

EXPECT\_EQ(list[1], 2);

EXPECT\_EQ(list[2], 3);

}

// Test the remove() function

TEST\_F(DoubleLinkedListTest, Remove) {

// Add elements to the list

list.push\_back(1);

list.push\_back(2);

list.push\_back(3);

// Remove the element at index 1

list.remove(1);

// Verify that the element is removed and the list is updated correctly

EXPECT\_EQ(list.length(), 2);

EXPECT\_EQ(list[0], 1);

EXPECT\_EQ(list[1], 3);

}

// Test the to\_string() function

TEST\_F(DoubleLinkedListTest, ToString) {

// Add elements to the list

list.push\_back(1);

list.push\_back(2);

list.push\_back(3);

// Verify that the string representation is correct

std::string expected = "[1, 2, 3]";

EXPECT\_EQ(list.to\_string(), expected);

}

// Test the stream insertion operator

TEST\_F(DoubleLinkedListTest, StreamInsertion) {

// Add elements to the list

list.push\_back(1);

list.push\_back(2);

list.push\_back(3);

// Create an output stringstream

std::stringstream ss;

// Insert the list into the stringstream

ss << list;

// Verify that the string representation is correct

std::string expected = "[1, 2, 3]";

EXPECT\_EQ(ss.str(), expected);

}

int main(int argc, char\* argv[]) {

::testing::InitGoogleTest(&argc, argv);

return RUN\_ALL\_TESTS();

}

**Uml – диаграмма:**

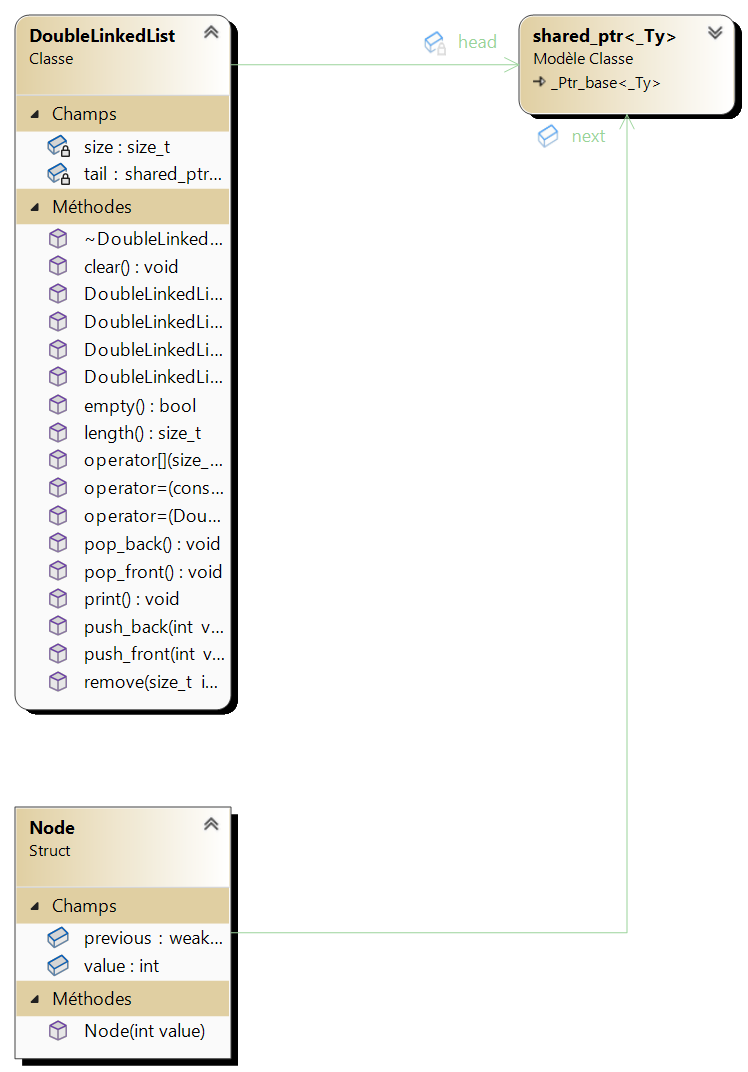


Рисунок 2 – диаграмма проекта

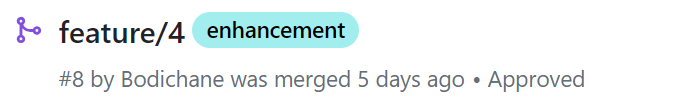


Рисунок 3 – approve задания