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 № 1

По дисциплине «Языки программирования»

Выполнил: ст. гр. ТКИ – 241

Борцов Ю. Д.

Проверил: к.т.н., доц.

Васильева М. А.

Москва 2023

**Задание:** Написать класс Single linked list

**Код программы:**

#include <iostream>

#include <sstream>

template <typename T>

struct Node

{

T data;

Node\* next;

Node\* previous;

/\*\*

\* @brief Конструктор класса Node.

\* @param data Значение, которое содержит элемент списка.

\* @param next Указатель на следующий элемент списка.

\* @param previous Указатель на предыдующий элемент списка.

\*/

Node(const T data, Node\* next = nullptr, Node\* previous = nullptr);

/\*\*

\* @brief Оператор сравнения двух объектов класса Node.

\* @param node Объект класса Node.

\* @return true/false в зависимости от объектов.

\*/

bool operator == (const Node& node);

/\*\*

\* @brief Оператор сравнения двух объектов класса Node.

\* @param node Объект класса Node.

\* @return true/false в зависимости от объектов.

\*/

bool operator != (const Node& node);

};

template <typename T>

Node<T>::Node(const T data, Node\* next, Node\* previous):data(data), next(next), previous(previous){}

template <typename T>

bool Node<T>::operator==(const Node &node)

{

return (this->data == node.data);

}

template <typename T>

bool Node<T>::operator!= (const Node& node)

{

return !(\*this == node);

}

template <typename T>

class List

{

private:

Node<T>\* head;

Node<T>\* tail;

public:

/\*\*

\* @brief Конструктор класса List.

\* @param head Указатель на первый элемент списка.

\* @param tail Указатель на последний элемент списка.

\*/

List(Node<T>\* head = nullptr, Node<T>\* tail = nullptr);

/\*\*

\* @brief Вставляет новый узел с заданными данными в конец списка

\* @param data Данные для вставки

\*/

void PushBack(const T data);

/\*\*

\* @brief Проверяет, пуст ли список

\* @return True, если список пуст, иначе false

\*/

bool IsEmpty();

/\*\*

\* @brief Деструктор класса List.

\*/

~List();

/\*\*

\* @brief Очищает список, удаляя все его элементы.

\*/

void clear();

/\*\*

\* @brief Вставляет новый узел с заданными данными в начало списка

\* @param data Данные для вставки

\*/

void PushAhead(const T data);

/\*\*

\* @brief Удаляет узел с конца списка

\*/

void DeleteBack();

/\*\*

\* @brief Удаляет узел с начала списка

\*/

void DeleteAhead();

/\*\*

\* @brief Перегружает оператор == для сравнения двух списков

\* @param list Список для сравнения

\* @return True, если списки равны, иначе false

\*/

bool operator==(List<T> list);

/\*\*

\* @brief Перегружает оператор != для сравнения двух списков

\* @param list Список для сравнения

\* @return True, если списки не равны, иначе false

\*/

bool operator!=(List<T> list);

/\*\*

\* @brief Возвращает строковое представление списка.

\* @return Строка, содержащая все элементы списка.

\*/

std::string toString();

/\*\*

\* @brief Конструктор копирования класса List.

\* @param other Объект для копирования.

\*/

List(const List<T> &other);

/\*\*

\* @brief Оператор присваивания копированием для класса List.

\* @param other Объект для копирования.

\* @return Ссылка на текущий объект.

\*/

List<T> &operator=(const List<T> &other);

/\*\*

\* @brief Конструктор перемещения класса List.

\* @param other Объект для перемещения.

\*/

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

/\*\*

\* @brief Оператор присваивания перемещением для класса List.

\* @param other Объект для перемещения.

\* @return Ссылка на текущий объект.

\*/

List<T> &operator=(List<T> &&other) noexcept;

/\*\*

\* @brief Перегружает оператор << для вывода списка

\* @param os Поток вывода

\* @param list Список для вывода

\* @return Поток вывода

\*/

template <typename U>

friend std::ostream& operator<<(std::ostream& os, const List<U>& list);

};

template <typename T>

List<T>::List(Node<T>\* head, Node<T>\* tail) : head(head), tail(tail) {}

template <typename T>

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

{

Node<T>\* current = list.head;

while (current != nullptr)

{

os << current->data << " ";

current = current->previous;

}

return os;

}

template <typename T>

bool List<T>::IsEmpty()

{

return (this->tail == nullptr);

}

template <typename T>

void List<T>::PushBack(const T data)

{

Node<T>\* node = new Node<T>(data);

if (this->IsEmpty())

{

this->tail = node;

this->head = node;

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

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

}

else if (this->tail == this->head)

{

this->tail = node;

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

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

}

else

{

node->data = this->tail->data;

Node<T>\* temp = this->tail->next;

this->tail->next->previous = node;

node->next = temp;

node->previous = this->tail;

this->tail->next = node;

this->tail->data = data;

}

}

template <typename T>

List<T>::~List()

{

clear();

}

template <typename T>

void List<T>::clear()

{

while (tail != head)

{

DeleteAhead();

}

if (tail == head)

{

delete head;

}

}

template <typename T>

void List<T>::PushAhead(const T data)

{

Node<T> \*node = new Node<T>(data);

if (IsEmpty())

{

tail = node;

head = node;

}

else if (tail == head)

{

head = node;

head->previous = tail;

tail->next = head;

}

else

{

node->data = head->data;

Node<T> \*temp = head->previous;

head->previous->next = node;

node->previous = temp;

node->next = head;

head->previous = node;

head->data = data;

}

}

template <typename T>

void List<T>::DeleteBack()

{

Node<T> \*next = tail->next;

if (next != nullptr)

{

delete next->previous;

next->previous = nullptr;

}

tail = next;

}

template <typename T>

void List<T>::DeleteAhead()

{

Node<T> \*previous = head->previous;

if (previous != nullptr)

{

delete previous->next;

previous->next = nullptr;

}

head = previous;

}

template <typename T>

bool List<T>::operator==(List<T> list)

{

return (toString() == list.toString());

}

template <typename T>

std::string List<T>::toString()

{

std::stringstream buffer;

buffer << \*this;

return buffer.str();

}

template <typename T>

bool List<T>::operator!=(List<T> list)

{

return !(\*this == list);

}

template <typename T>

List<T>::List(const List<T> &other) : head(nullptr), tail(nullptr)

{

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

while (temp != nullptr)

{

this->PushBack(temp->data);

temp = temp->previous;

}

}

template <typename T>

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

{

List<T> temp(other);

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

return \*this;

}

template <typename T>

List<T>::List(List<T> &&other) noexcept : head(other.head), tail(other.tail)

{

\*this = std::move(other);

}

template <typename T>

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

{

if (this != &other)

{

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

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

}

return \*this;

}

**Тесты:**

#include <gtest/gtest.h>

#include "doubleLinkedListTemplates.cpp"

TEST(NodeTest, Equality) {

Node<int> node1(42);

Node<int> node2(42);

Node<int> node3(10);

ASSERT\_TRUE(node1 == node2);

ASSERT\_FALSE(node1 == node3);

}

TEST(NodeTest, Inequality) {

Node<int> node1(42);

Node<int> node2(42);

Node<int> node3(10);

ASSERT\_FALSE(node1 != node2);

ASSERT\_TRUE(node1 != node3);

}

TEST(ListTest, PushBack) {

List<int> list;

list.PushBack(1);

list.PushBack(2);

list.PushBack(3);

ASSERT\_EQ(list.toString(), "1 2 3 ");

}

TEST(ListTest, PushAhead) {

List<int> list;

list.PushAhead(1);

list.PushAhead(2);

list.PushAhead(3);

ASSERT\_EQ(list.toString(), "3 2 1 ");

}

TEST(ListTest, DeleteBack) {

List<int> list;

list.PushBack(1);

list.PushBack(2);

list.PushBack(3);

list.DeleteBack();

ASSERT\_EQ(list.toString(), "1 2 ");

}

TEST(ListTest, DeleteAhead) {

List<int> list;

list.PushBack(1);

list.PushBack(2);

list.PushBack(3);

list.DeleteAhead();

ASSERT\_EQ(list.toString(), "2 3 ");

}

TEST(ListTest, CopyConstructor) {

List<int> original;

original.PushBack(1);

original.PushBack(2);

original.PushBack(3);

List<int> copy(original);

ASSERT\_EQ(original.toString(), copy.toString());

}

TEST(ListTest, CopyAssignmentOperator) {

List<int> original;

original.PushBack(1);

original.PushBack(2);

original.PushBack(3);

List<int> copy;

copy = original;

ASSERT\_EQ(original.toString(), copy.toString());

}

TEST(ListTest, MoveConstructor) {

List<int> original;

original.PushBack(1);

original.PushBack(2);

original.PushBack(3);

List<int> moved(std::move(original));

ASSERT\_EQ("", original.toString());

ASSERT\_EQ("1 2 3 ", moved.toString());

}

TEST(ListTest, MoveAssignmentOperator) {

List<int> original;

original.PushBack(1);

original.PushBack(2);

original.PushBack(3);

List<int> moved;

moved = std::move(original);

ASSERT\_EQ("", original.toString());

ASSERT\_EQ("1 2 3 ", moved.toString());

}

TEST(ListTest, Equality) {

List<int> list1;

list1.PushBack(1);

list1.PushBack(2);

list1.PushBack(3);

List<int> list2;

list2.PushBack(1);

list2.PushBack(2);

list2.PushBack(3);

List<int> list3;

list3.PushBack(3);

list3.PushBack(2);

list3.PushBack(1);

ASSERT\_TRUE(list1 == list2);

ASSERT\_FALSE(list1 == list3);

}

TEST(ListTest, Inequality) {

List<int> list1;

list1.PushBack(1);

list1.PushBack(2);

list1.PushBack(3);

List<int> list2;

list2.PushBack(1);

list2.PushBack(2);

list2.PushBack(3);

List<int> list3;

list3.PushBack(3);

list3.PushBack(2);

list3.PushBack(1);

ASSERT\_FALSE(list1 != list2);

ASSERT\_TRUE(list1 != list3);

}

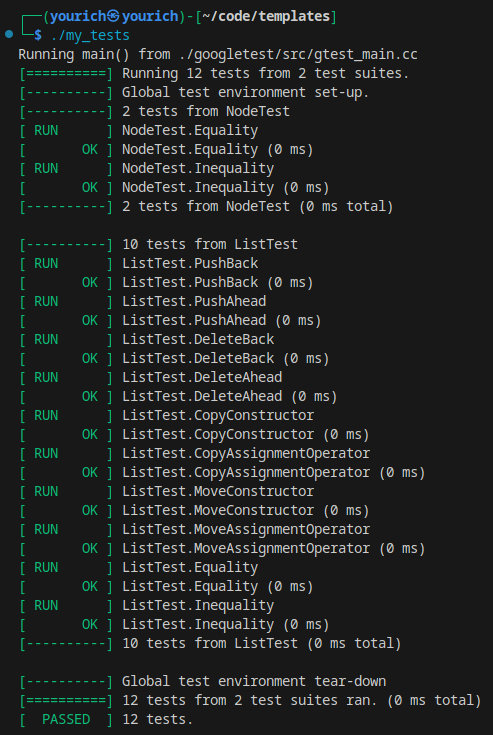
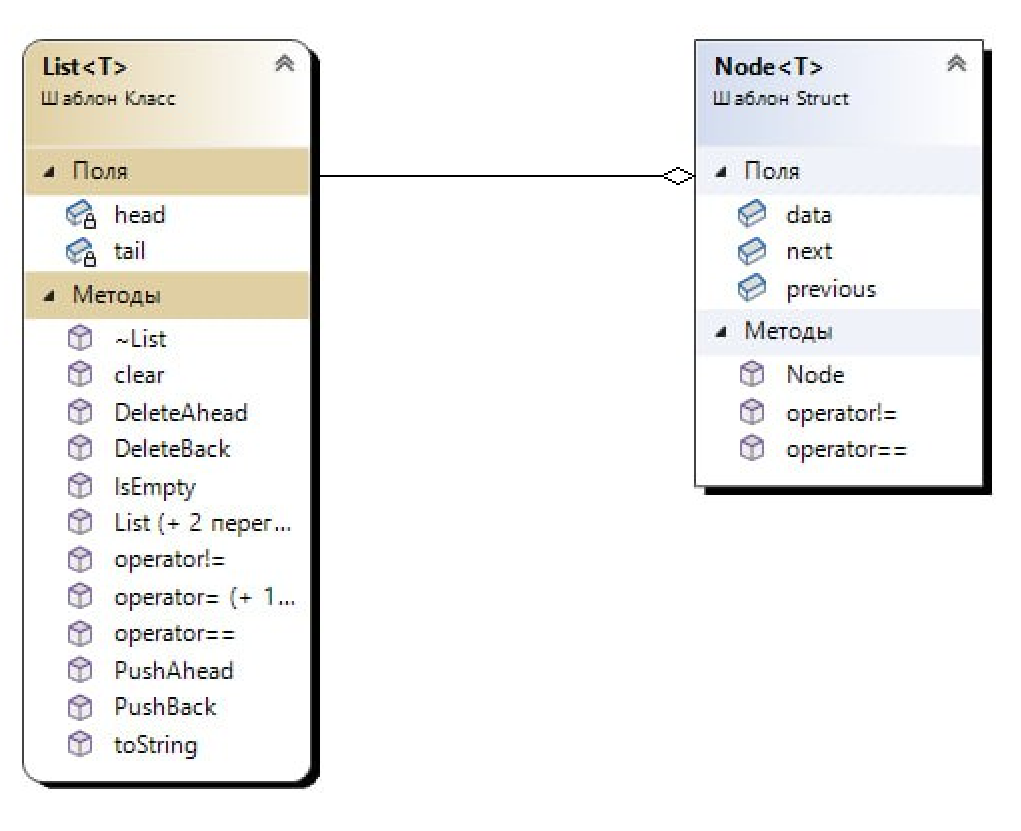


Рисунок 1 – прохождение тестов

****

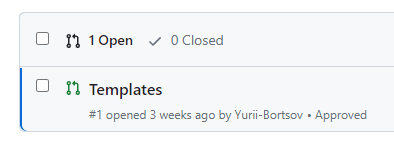
Рисунок 2 -UML – диаграмма проекта

Рисунок 3 – approve задания.