МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Алгоритмы и струтуры данных»

Тема: Списки

Вариант 4

Студент гр. 0302 _____ Устинов Г.А.

Преподаватель _____ Тутуева А.В.

Санкт-Петербург

2021

Цель работы

Изучить структуру данных «Двусвязный список». Реализовать двусвязный список и заданные методы.

Задание

Вариант 4

Реализовать класс связного списка с набором методов. Данные, хранящиеся в списке могут быть любого типа на ваш выбор (например, int).

Отчет по лабораторной работе должен содержать:

1. Титульный лист с указанием варианта.

2. Постановка задачи. Описание реализуемого класса и методов.

3. Оценка временной сложности каждого метода.

4. Описание реализованных unit-тестов.

5. Пример работы.

6. Листинг.

Наличие unit-тестов ко всем реализуемым методам и предусмотренные исключительные ситуации является обязательными требованиями. При выполнении задания запрещено пользоваться библиотекой <algorithm>.

Обязательна реализация конструктора и деструктора.

Список методов, которые реализует каждый вариант (приведено для типа данных int):

1. void push_back(int); // добавление в конец списка

2. void push_front(int); // добавление в начало списка

3. void pop_back(); // удаление последнего элемента

4. void pop_front(); // удаление первого элемента

5. void insert(int, size_t) // добавление элемента по индексу (вставка перед элементом, который был ранее доступен по этому индексу)

6. int at(size_t); // получение элемента по индексу

7. void remove(size_t); // удаление элемента по индексу

8. size_t get_size(); // получение размера списка

9. void clear(); // удаление всех элементов списка

10. void set(size_t, int); // замена элемента по индексу на передаваемый элемент

11. bool isEmpty(); // проверка на пустоту списка

12. Перегрузка оператора вывода <<;

Список методов, которые реализуются в отдельных вариантах:

13. void reverse(); // меняет порядок элементов в списке на обратный

14. void insert(List, size_t); // вставка другого списка в список, начиная с индекса

15. void push_back(List); // вставка другого списка в конец

16. void push_front(List); // вставка другого списка в начало

17. bool contains(List); // проверка на содержание другого списка в списке, можно сделать типа int

18. int find_first(List); // поиск первого вхождения другого списка в список

19. int find_last(List); // поиск последнего вхождения другого списка в список

20. void swap(size_t, size_t); // обмен двух элементов списка по индексам

Варианты

№ Варианта	Тип списка	Реализуемые функции
1	односвязный список	1-12, 13
2	двусвязный список	1-12, 13
3	односвязный список	1-12, 14
4	двусвязный список	1-12, 14

5	односвязный список	1-12, 15
6	двусвязный список	1-12, 15
7	односвязный список	1-12, 16
8	двусвязный список	1-12, 16
9	односвязный список	1-12, 17
10	двусвязный список	1-12, 17
11	односвязный список	1-12, 18
12	двусвязный список	1-12, 18
13	односвязный список	1-12, 19
14	двусвязный список	1-12, 19
15	односвязный список	1-12, 20
16	двусвязный список	1-12, 20

Выполнение работы

Для типа списка были реализованы два класса — *DlinkedList* и *Node*.

Таблица 1 — Поля класса *Node*

Тип	Название поля	Описание
std::shared_ptr<Node>	*prev*	Указатель на следующий элемент
std::shared_ptr<Node>	*next*	Указатель на предыдущий элемент
int	*a*	Хранимый элемент

Класс *Node* является скрытым типом для внутреннего использования типом списка и представляет собой элемент двусвязного списка, хранящий указатели на предыдущий и следующий элемент, а также поле данных.

Класс *DlinkedList* представляет собой тип самого списка, который хранит указатели на первый и последний элемент, что помогает ускорить

выполнение некоторых типовых операций над списком, и количество элементов в списке — его размер или «длину».

Таблица 2 — Поля класса *DLinkedList*

Тип	Название поля	Описание
std::shared_ptr<Node>	*head_*	Указатель на первый элемент
std::shared_ptr<Node>	*tail_*	Указатель на последний элемент
size_t	*size_*	Длина списка (количество элементов)

Согласно варианта работы были реализованы следующие методы:

- *void push_back(int)* — добавляет элемент в конец списка. Благодаря тому, что в списке есть указатель на «хвост» списка — добавление производится за константное время — не зависит от длины списка - O(1). Данный метод тестируется в двух условиях — при пустом и непустом списке.
- *void push_front(int)* - добавляет элемент в начало списка. Добавление производится за константное время — не зависит от длины списка - O(1). Метод тестируется в двух условиях — при пустом и непустом списке.
- *void pop_back()* - удаляет элемент из конца списка. В случае применения для пустого списка ничего не делает. Также как и *push_back()* выполняется за время, не зависящее от количества элементов в списке - O(1). Тестируется как в условиях пустого, так и непустого списка.
- *void pop_front()* - удаляет элемент из начала списка. В случае применения для пустого списка ничего не делает. Также как и *push_front()* выполняется за время, не зависящее от количества

элементов в списке - O(1). Тестируется как в условиях пустого, так и непустого списка.

- *void insert(int, size_t)* — добавляет элемент со значением *a* по произвольному индексу. В случае указания индекса за пределами индекскации списка, выбрасывает исключение *std::out_of_range*. В общем случае выполняется за линейное время, зависящее от количества элементов в списке, поскольку нужно последовательно переходить от элемента к элементу до достижения нужной позиции — O(n). Тестируется как в случае пустого, так непустого списка, причем для непустого списка производится вставка в начало, в конец и в середину. Также в обоих случаях отрабатывается исключение при передаче индекса за границей списка.
- *void remove(size_t)* — удаляет элемент по произвольному индексу. В случае указания индекса за пределами индекскации списка, выбрасывает исключение *std::out_of_range*. Подобно *insert()* работает за линейное время — O(n). Тестируется как в случае пустого, так непустого списка, причем для непустого списка производится удаление из начала, из конца и из середины. Также в обоих случаях отрабатывается исключение при передаче индекса за границей списка.
- *int at(size_t)* — возвращяет значение, которое хранит элемент с номером *i*. При обращении к элементу с слишком большим номером выбрасывает исключение *std::out_of_range*. Из-за необходимости последовательного прохождения элементов до достижения указанного номера работает за время O(n). Тестируется как для пустого списка — на исключение — так и для непустого — последовательно обходятся все элементы с помощью *at()*.

- *void set(size_t, int)* — устанавливает значение элемента по индексу *i* значение *n*. Для слишком высоких индексов выбрасывает исключение *std::out_of_range*. Так же как и *at()* работает за линейное время O(n). Тестируется для непустого — последовательной установкой элементам значений и на исключение — так и для пустого — проверяется только исключение.
- *size_t get_size()* - возвращает размер списка — значение *size_*. Время работы не зависит от размера списка — длина списка хранится в отдельном поле. Тестируется однажды — в одном из самых первых тестов провожится проверка — сравнивается значение, которое возвращает этот метод для пустого списка с эталоном — нулём.
- *void clear()* - удаляет все элементы из списка. Для начала «затирается» *tail_*, затем последовательно *head_* перемещается по элементам и «затирает» указатель на предыдущий элемент. Таким образом, удаление объектов и освобождение памяти отдаётся на откуп *std::shared_ptr.* При использовании для пустого списка ничего не делает. Выполняется тем дольше, чем больше элементов в списке, потому что последовательно обходит все элементы - O(n). Тестируется как для пустого, так и для непустого списка.
- *bool is_empty()* - возвращает *true*, если список пуст и *false* в противном случае. Время работы константно, поскольку пустота определяется сравнение поля *size_* с нулём. Тестируется проверкой возвращаемого значения для пустого списка и списка, содержащего один элемент.
- *void insert(DLinkedList&, size_t)* — вставляет список по ссылке *list* в произвольное место так, чтобы первый его номер оказался под индексом *i*. При указании слишком большого индекса выбрасывает исключение *std::shared_ptr*. Работает за линейное время, поскольку

последовательно обходит все элементы списка, в который вставляются элементы для нахождения нужной позиции, а также копирует каждый элемент из вставляемого списка — O(n). После этого головной и предшествующий и хвостовой и последующий элементы начинают попарно указывать друг на друга. Тестируется как для пустого, так и для непустого списка вставкой как пустого так и непустого списка. Также проверяется вставка по ошибочному индексу на выброс исключения.

Дополнительно, для копирования элементов списка при создании был создан копирующий конструктор — *DlinkedList(const DlinkedList&)*, который копирует все элементы из списка-аргумента в создаваемый список. Тестируется в группе *Base* копированием заполненного списка и последовательным сравнением всех элементов и размеров списков.

Для очистки списка перед удалением и освобождения памяти деструктор использует функцию *clear()*.

Тестирование

Тесты были поделены на три группы — базовые, проверка пустого списка и проверка непустого писка. Базовые тесты проверяют инвариант создаваемого теста и функции, на которые полагаются все остальные тесты для проверки состояния списка — *get_size()* и *is_empty()*, а также копирующий конструктор.

Тесты проверки пустого списка проверяют все методы кроме двух, указанных выше и проверяют работу методов при их вызове для пустого списка.

Тесты проверки непустого списка проверяют работу методов с непустым списком. Из-за появления размерности у списка некоторые тесты поделены на несколько — проверка для начала, для середины и для конца.

Для тестирования используется фреймворк googletest.

Пример работы

Следующий код служит примером работы с разработанным списком.

```
int n;
cin >> n;
int a;
for (size_t i = 0; i < n; ++i) {
    cin >> a;
    // Если в списке ещё есть элементы, то перезаписать их
    if (i < list.get_size())
        list.set(i, a);
    else  // Если же нет, то добавить новые
        list.push_back(a);
}
cout << list << endl;
```

Разработанный программный код см. в приложении А.

Выводы

Была изучена такая структура данных как двусвязный список, а так же принципы функций, работающий с ней. Также была разработана собственная реализация, которая оптимизирует некоторые функции обработки двусвязного списка ценой незначительных дополнительных затрат памяти на хранение списка. В процессе выполнения работы были изучены такие инструменты как cmake для упрощения работы с проектом, состоящим из разных библиотек, и фреймворк googletest, облегчающий написание модульных тестов.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл **src/dlist.hh**

```
#pragma once
#ifndef DLIST_HH
#define DLIST_HH

#include <iostream>
#include <ostream>
#include <memory>


namespace dlist {

class DLinkedList {
public:
   DLinkedList() : head_(nullptr), tail_(nullptr), size_(0) {}
   DLinkedList(const DLinkedList& src);
   ~DLinkedList() { clear(); }

   void push_back(int a);
   void push_front(int a);
   void pop_back();
   void pop_front();

   void insert(int a, size_t i);
   void remove(size_t i);
   int at(size_t i);
   void set(size_t i, int a);

   size_t get_size() { return size_; }
   void clear();
   bool is_empty() { return size_ == 0; }

   void insert(DLinkedList& list, size_t i);

   friend std::ostream& operator<<(std::ostream& os, const
DLinkedList& list);
private:
   struct Node;

   std::shared_ptr<Node> head_;
   std::shared_ptr<Node> tail_;

   size_t size_;
};


struct DLinkedList::Node {
```

```
   Node(int a) : prev(nullptr), next(nullptr), a(a) {}

   std::shared_ptr<Node> prev;
   std::shared_ptr<Node> next;

   int a;
};

}    // namespace dlist

#endif  // DLIST_HH
```

Файл **src/dlist.cpp**

```
#include "dlist.hh"

#include <stdexcept>


namespace dlist {


DLinkedList::DLinkedList(const DLinkedList& src) : size_(0)
{
   std::shared_ptr<Node> cur = src.head_;
   while (cur) {
       push_back(cur->value);
       cur = cur->next;
   }
}


void DLinkedList::push_back(int value)
{
   std::shared_ptr<Node> node(new Node(value));
   if (!tail_) {
       head_ = node;
   } else {
       tail_->next = node;
       node->prev = tail_;
   }
   tail_ = node;
   ++size_;
}

void DLinkedList::push_front(int value)
{
   std::shared_ptr<Node> node(new Node(value));
   if (!head_) {
       tail_ = node;
   } else {
       head_->prev = node;
```

```
        node->next = head_;
    }
    head_ = node;
    ++size_;
}

void DLinkedList::pop_back()
{
    if (!is_empty()) {
        tail_ = tail_->prev;
        if (!tail_)
            head_ = nullptr;
        else
            tail_->next = nullptr;
        --size_;
    }
}

void DLinkedList::pop_front()
{
    if (!is_empty()) {
        head_ = head_->next;
        if (!head_)
            tail_ = nullptr;
        else
            head_->prev = nullptr;
        --size_;
    }
}


void DLinkedList::insert(int value, size_t i)
{
    if (i > size_) // if i == size_, node will be inserted after the
last
        throw std::out_of_range("Index out of range.");
    else if (i == 0)
        return push_front(value);
    else if (i == size_)
        return push_back(value);
    std::shared_ptr<Node> node(new Node(value));
    std::shared_ptr<Node> tmp;
    if (i == size_ - 1) {
        tmp = tail_;
    } else {
        tmp = head_->next;
        for (size_t j = 1; j != i; ++j)
            tmp = tmp->next;
    }
    node->prev = tmp->prev;
    node->next = tmp;
    tmp->prev->next = node;
```

```
    tmp->prev = node;
    ++size_;
}

void DLinkedList::remove(size_t i)
{
    if (i >= size_) {
        throw std::out_of_range("Index out of range.");
    } else if (i == 0) {
        return pop_front();
    } else if (i == size_ - 1) {
        return pop_back();
    }
    std::shared_ptr<Node> tmp = head_->next;
    for (size_t j = 1; j != i; ++j)
        tmp = tmp->next;
    tmp->prev->next = tmp->next;
    tmp->next->prev = tmp->prev;
    --size_;
}

int DLinkedList::at(size_t i)
{
    if (i >= size_) {
        throw std::out_of_range("Index out of range.");
    } else if (i == 0) {
        return head_->value;
    } else if (i == size_ - 1) {
        return tail_->value;
    }
    std::shared_ptr<Node> tmp = head_->next;
    for (size_t j = 1; j != i; ++j)
        tmp = tmp->next;
    return tmp->value;
}

void DLinkedList::set(size_t i, int value)
{
    if (i >= size_) {
        throw std::out_of_range("Index out of range.");
    } else if (i == 0) {
        head_->value = value;
        return;
    } else if (i == size_ - 1) {
        tail_->value = value;
        return;
    }
    std::shared_ptr<Node> tmp = head_->next;
    for (size_t j = 1; j != i; ++j)
        tmp = tmp->next;
    tmp->value = value;
}
```

```
void DLinkedList::clear()
{
   if (is_empty())
       return;
   tail_.reset();
   while (head_) {
      head_->prev.reset();
      head_ = head_->next;
   }
   size_ = 0;
}


void DLinkedList::insert(DLinkedList& list, size_t i)
{
   if (i > size_)
       throw std::out_of_range("Index out of range.");
   else if (list.is_empty())
       return;

   DLinkedList copy = list;
   std::shared_ptr<Node> prev;
   std::shared_ptr<Node> cur;
   if (i == size_) {
       prev = tail_;
       cur = nullptr;
   } else {
       prev = nullptr;
       cur = head_;
       for (size_t j = 0; j != i; ++j) {
           prev = cur;
           cur = cur->next;
       }
   }
   // if prev is nullptr, then inserting into the head of the list
   if (prev) {
       prev->next = copy.head_;
       copy.head_->prev = prev;
   } else {
       head_ = copy.head_;
   }
   // if cur is nullptr, then inserting into the tail of the list
   if (cur) {
       cur->prev = copy.tail_;
       copy.tail_->next = cur;
   } else {
       tail_ = copy.tail_;
   }
   size_ += copy.size_;
}
```

```
std::ostream& operator<<(std::ostream& outstream, const DLinkedList&
list)
{
   outstream << "Doubly Linked List: {";
   std::shared_ptr<DLinkedList::Node> tmp = list.head_;
   for (size_t i = 0; i != list.size_; ++i) {
       outstream << tmp->value;
       if (i != list.size_ - 1)
           outstream << ", ";
       tmp = tmp->next;
   }
   outstream << "}";
   return outstream;
}

}   // namespace dlist
```

Файл **test/lab1-test.cpp**

```
#include "../src/dlist.hh"

#include <iostream>
#include <stdexcept>

#include <gtest/gtest.h>

using dlist::DLinkedList;


TEST(Base, Creation)
{
   DLinkedList list;
   std::cout << list << std::endl;
   ASSERT_EQ(list.get_size(), 0);
   ASSERT_TRUE(list.is_empty());
}

TEST(Base, GetSize)
{
   DLinkedList list;
   ASSERT_EQ(list.get_size(), 0);
}

TEST(Base, IsEmpty)
{
   DLinkedList list;
   ASSERT_TRUE(list.is_empty());
   list.push_back(0);
   ASSERT_FALSE(list.is_empty());
```

```
}

TEST(Base, CopyConstruct)
{
   DLinkedList list0;
   list0.push_back(0);
   list0.push_back(1);
   list0.push_back(2);
   list0.push_back(3);
   list0.push_back(4);
   DLinkedList list1 = list0;
   ASSERT_FALSE(list1.is_empty());
   ASSERT_EQ(list1.get_size(), list0.get_size());
   for (size_t i = 0; i < list0.get_size(); ++i)
       ASSERT_EQ(list0.at(i), list1.at(i));
}


class DLinkedListEmptyTest : public ::testing::Test {
protected:
   DLinkedList list_;
};

TEST_F(DLinkedListEmptyTest, PushBack)
{
   list_.push_back(0);
   std::cout << list_ << std::endl;
   ASSERT_EQ(list_.get_size(), 1);
   ASSERT_FALSE(list_.is_empty());
}

TEST_F(DLinkedListEmptyTest, PushFront)
{
   list_.push_front(0);
   std::cout << list_ << std::endl;
   ASSERT_EQ(list_.get_size(), 1);
   ASSERT_FALSE(list_.is_empty());
}

TEST_F(DLinkedListEmptyTest, PopBack)
{
   list_.pop_back();
   std::cout << list_ << std::endl;
   ASSERT_EQ(list_.get_size(), 0);
   ASSERT_TRUE(list_.is_empty());
}

TEST_F(DLinkedListEmptyTest, PopFront)
{
   list_.pop_front();
   std::cout << list_ << std::endl;
   ASSERT_EQ(list_.get_size(), 0);
```

```
   ASSERT_TRUE(list_.is_empty());
}

TEST_F(DLinkedListEmptyTest, Insert)
{
   ASSERT_THROW(list_.insert(0, 1), std::out_of_range);
   std::cout << list_ << std::endl;
   list_.insert(0, 0);
   ASSERT_EQ(list_.get_size(), 1);
   ASSERT_FALSE(list_.is_empty());
}

TEST_F(DLinkedListEmptyTest, Remove)
{
   ASSERT_THROW(list_.remove(0), std::out_of_range);
   ASSERT_THROW(list_.remove(2), std::out_of_range);
   std::cout << list_ << std::endl;
   ASSERT_EQ(list_.get_size(), 0);
   ASSERT_TRUE(list_.is_empty());
}

TEST_F(DLinkedListEmptyTest, At)
{
   ASSERT_THROW(list_.at(0), std::out_of_range);
   ASSERT_THROW(list_.at(2), std::out_of_range);
}

TEST_F(DLinkedListEmptyTest, Set)
{
   ASSERT_THROW(list_.set(0, 1), std::out_of_range);
   ASSERT_THROW(list_.set(2, 1), std::out_of_range);
}

TEST_F(DLinkedListEmptyTest, Clear)
{
   list_.clear();
   ASSERT_EQ(list_.get_size(), 0);
   ASSERT_TRUE(list_.is_empty());
   std::cout << list_ << std::endl;
}


TEST_F(DLinkedListEmptyTest, InsertList)
{
   DLinkedList list;
   ASSERT_THROW(list_.insert(list, 1), std::out_of_range);
   list_.insert(list, 0);
   std::cout << "list_ " << list_ << std::endl;
   ASSERT_EQ(list_.get_size(), 0);
   ASSERT_TRUE(list_.is_empty());

   list.push_back(0);
```

```
    std::cout << "list " << list << std::endl;
    std::cout << "list_ "<< list_ << std::endl;
    list_.insert(list, 0);
    std::cout << "list " << list << std::endl;
    std::cout << "list_ "<< list_ << std::endl;
    ASSERT_EQ(list_.get_size(), list.get_size());
    ASSERT_FALSE(list.is_empty());
    ASSERT_FALSE(list_.is_empty());
    for (size_t i = 0; i < list.get_size(); ++i)
        ASSERT_EQ(list.at(i), list_.at(i));

    list_.clear();
    list.push_back(0);
    list.push_back(1);
    list.push_back(2);
    std::cout << "list " << list << std::endl;
    std::cout << "list_ "<< list_ << std::endl;
    list_.insert(list, 0);
    std::cout << "list " << list << std::endl;
    std::cout << "list_ "<< list_ << std::endl;
    ASSERT_EQ(list_.get_size(), list.get_size());
    ASSERT_FALSE(list.is_empty());
    ASSERT_FALSE(list_.is_empty());
    for (size_t i = 0; i < list.get_size(); ++i)
        ASSERT_EQ(list.at(i), list_.at(i));
}


class DLinkedListTest : public ::testing::Test {
protected:
    void SetUp() override
    {
        list_.push_back(0);
        list_.push_back(1);
        list_.push_back(2);
        list_.push_back(3);
    }

    DLinkedList list_;
};

TEST_F(DLinkedListTest, PushBack)
{
    size_t s = list_.get_size() + 1;
    list_.push_back(0);
    std::cout << list_ << std::endl;
    ASSERT_EQ(list_.get_size(), s);
    ASSERT_FALSE(list_.is_empty());
}

TEST_F(DLinkedListTest, PushFront)
{
```

```
   size_t s = list_.get_size() + 1;
   list_.push_front(0);
   std::cout << list_ << std::endl;
   ASSERT_EQ(list_.get_size(), s);
   ASSERT_FALSE(list_.is_empty());
}

TEST_F(DLinkedListTest, PopBack)
{
   size_t s = list_.get_size() - 1;
   list_.pop_back();
   std::cout << list_ << std::endl;
   ASSERT_EQ(list_.get_size(), s);
   ASSERT_FALSE(list_.is_empty());
}

TEST_F(DLinkedListTest, PopFront)
{
   size_t s = list_.get_size() - 1;
   list_.pop_front();
   std::cout << list_ << std::endl;
   ASSERT_EQ(list_.get_size(), s);
   ASSERT_FALSE(list_.is_empty());
}

TEST_F(DLinkedListTest, Insert)
{
   size_t s = list_.get_size() + 1;
   ASSERT_THROW(list_.insert(0, s), std::out_of_range);
   std::cout << list_ << std::endl;

   list_.insert(5, 0);
   std::cout << list_ << std::endl;
   ASSERT_EQ(list_.get_size(), s);
   ASSERT_FALSE(list_.is_empty());

   list_.insert(5, s++);
   std::cout << list_ << std::endl;
   ASSERT_EQ(list_.get_size(), s);
   ASSERT_FALSE(list_.is_empty());

   list_.insert(10, s/2);
   std::cout << "s/2: " << s/2 << std::endl;
   std::cout << list_ << std::endl;
   ASSERT_EQ(list_.get_size(), ++s);
   ASSERT_FALSE(list_.is_empty());
}

TEST_F(DLinkedListTest, Remove)
{
   size_t s = list_.get_size();
   ASSERT_THROW(list_.remove(s), std::out_of_range);
```

```
    std::cout << list_ << std::endl;
    ASSERT_EQ(list_.get_size(), s);
    ASSERT_FALSE(list_.is_empty());

    list_.remove(s/2);
    std::cout << list_ << std::endl;
    ASSERT_EQ(list_.get_size(), --s);
    ASSERT_FALSE(list_.is_empty());

    list_.remove(0);
    std::cout << list_ << std::endl;
    ASSERT_EQ(list_.get_size(), --s);
    ASSERT_FALSE(list_.is_empty());

    list_.remove(--s);
    std::cout << list_ << std::endl;
    ASSERT_EQ(list_.get_size(), s);
    ASSERT_FALSE(list_.is_empty());
}

TEST_F(DLinkedListTest, At)
{
    ASSERT_THROW(list_.at(10), std::out_of_range);
    for (size_t i = 0; i < list_.get_size(); ++i)
        ASSERT_EQ(list_.at(i), i);
}

TEST_F(DLinkedListTest, Set)
{
    size_t s = list_.get_size();
    ASSERT_THROW(list_.set(s, 1), std::out_of_range);
    for (size_t i = 0; i < s; ++i) {
        list_.set(i, s);
        ASSERT_EQ(list_.at(i), s);
    }
    std::cout << list_ << std::endl;
}

TEST_F(DLinkedListTest, Clear)
{
    list_.clear();
    ASSERT_EQ(list_.get_size(), 0);
    ASSERT_TRUE(list_.is_empty());
    std::cout << list_ << std::endl;
}

TEST_F(DLinkedListTest, InsertEmptyList)
{
    DLinkedList list;
    size_t s = list_.get_size();
    list_.insert(list, 0);
    std::cout << "list " << list << std::endl;
```

```
   std::cout << "list_ " << list_ << std::endl;
   ASSERT_EQ(list_.get_size(), s);
   ASSERT_FALSE(list_.is_empty());
}

TEST_F(DLinkedListTest, InsertListAtHead)
{
   DLinkedList list;
   size_t s = list_.get_size();
   list.push_back(10);
   std::cout << "list " << list << std::endl;
   std::cout << "list_ "<< list_ << std::endl;
   list_.insert(list, 0);
   std::cout << "list " << list << std::endl;
   std::cout << "list_ "<< list_ << std::endl;
   ASSERT_EQ(list_.get_size(), s + list.get_size());
   ASSERT_FALSE(list.is_empty());
   ASSERT_FALSE(list_.is_empty());
   for (size_t i = 0; i < list.get_size(); ++i)
       ASSERT_EQ(list.at(i), list_.at(i));
}

TEST_F(DLinkedListTest, InsertListInMiddle)
{
   DLinkedList list;
   size_t s = list_.get_size();
   list.push_back(0);
   list.push_back(1);
   list.push_back(2);
   std::cout << "list " << list << std::endl;
   std::cout << "list_ "<< list_ << std::endl;
   list_.insert(list, s/2);
   std::cout << "list " << list << std::endl;
   std::cout << "list_ "<< list_ << std::endl;
   ASSERT_EQ(list_.get_size(), s + list.get_size());
   ASSERT_FALSE(list.is_empty());
   ASSERT_FALSE(list_.is_empty());
   for (size_t j = 0, i  = s/2; j < list.get_size(); ++i, ++j)
       ASSERT_EQ(list.at(j), list_.at(i));
}

TEST_F(DLinkedListTest, InsertListAtTail)
{
   DLinkedList list;
   size_t s = list_.get_size();
   list.push_front(0);
   list.push_front(1);
   list.push_front(2);
   std::cout << "list " << list << std::endl;
   std::cout << "list_ "<< list_ << std::endl;
   list_.insert(list, s);
   std::cout << "list " << list << std::endl;
```

```
    std::cout << "list_ "<< list_ << std::endl;
    ASSERT_EQ(list_.get_size(), s + list.get_size());
    ASSERT_FALSE(list.is_empty());
    ASSERT_FALSE(list_.is_empty());
    for (size_t j = 0, i  = s; j < list.get_size(); ++i, ++j)
        ASSERT_EQ(list.at(j), list_.at(i));
}
```