

**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 структуры данных»**  
**Тема: Списки**  
**Вариант 4**

Студент гр. 9381

\_\_\_\_\_

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
<code>std::shared_ptr&lt;Node&gt;</code>	<i>prev</i>	Указатель на следующий элемент
<code>std::shared_ptr&lt;Node&gt;</code>	<i>next</i>	Указатель на предыдущий элемент
<code>int</code>	<i>a</i>	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
<code>std::shared_ptr&lt;Node&gt;</code>	<code>head_</code>	Указатель на первый элемент
<code>std::shared_ptr&lt;Node&gt;</code>	<code>tail_</code>	Указатель на последний элемент
<code>size_t</code>	<code>size_</code>	Длина списка (количество элементов)

Согласно варианта работы были реализованы следующие методы:

- `void push_back(int a)` — 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 a)` - 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 a, size\_t i)* — добавляет элемент со значением *a* по произвольному индексу. В случае указания индекса за пределами индексации 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 i)* — удаляет элемент по произвольному индексу. В случае указания индекса за пределами индексации 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 i)* — возвращает 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 i, int a)* — 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 проводится 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 сравнением поля *size\_* с нулём. Тестируется проверкой возвращаемого значения для пустого списка и списка, содержащего один элемент.
- *void insert(DLinkedList& list, size\_t i)* — 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 —  $O(n)$ . После этого головной и предшествующий и хвостовой и последующий элементы начинают попарно указывать друг на друга. После вставки элементов *list* перестаёт владеть этой цепочкой элементов — указатели *head\_* и *tail\_* приравниваются *nullptr*. 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.

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 списка. Базовые тесты проверяют инвариант создаваемого теста и функции, на которые полагаются все остальные тесты для проверки состояния списка — *get\_size()* и *is\_empty()*.

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, работающих 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 стаке 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() { 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 {

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

void DLinkedList::push_front(int a)
{
    std::shared_ptr<Node> n(new Node(a));
    if (!head_) {
        tail_ = n;
    } else {
        head_>prev = n;
        n->next = head_;
    }
    head_ = n;
    ++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 a, 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(a);
    else if (i == size_)
        return push_back(a);
    std::shared_ptr<Node> n(new Node(a));
    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;
    }
    n->prev = tmp->prev;
    n->next = tmp;
    tmp->prev->next = n;
    tmp->prev = n;
    ++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_->a;
    } else if (i == size_ - 1) {
        return tail_->a;
    }
    std::shared_ptr<Node> tmp = head_->next;
    for (size_t j = 1; j != i; ++j)
        tmp = tmp->next;
    return tmp->a;
}

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

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;

    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 = list.head_;
        list.head_->prev = prev;
    } else {
        head_ = list.head_;
    }
    list.head_ = nullptr;
    // if cur is nullptr, then inserting into the tail of the list
    if (cur) {
        cur->prev = list.tail_;
        list.tail_->next = cur;
    } else {
        tail_ = list.tail_;
    }
    list.tail_ = nullptr;
    size_ += list.size_;
    list.size_ = 0;
}
```

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

```

        tmp = tmp->next;
    }
    os << "}";
    return os;
}

} // 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());
}

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(), 1);
    ASSERT_FALSE(list_.is_empty());
    ASSERT_EQ(list_.at(0), 0);

    ASSERT_EQ(list.get_size(), 0);
    ASSERT_TRUE(list.is_empty());

    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(), 3);
    ASSERT_FALSE(list_.is_empty());

    ASSERT_EQ(list.get_size(), 0);
    ASSERT_TRUE(list.is_empty());
}

```

```

}

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);
    ASSERT_FALSE(list_.is_empty());

    ASSERT_EQ(list.get_size(), 0);
    ASSERT_TRUE(list.is_empty());
}

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);
    s += 3;
    std::cout << "list " << list << std::endl;
    std::cout << "list_ " << list_ << std::endl;
    ASSERT_EQ(list_.get_size(), s);
    ASSERT_FALSE(list_.is_empty());

    ASSERT_EQ(list.get_size(), 0);
    ASSERT_TRUE(list.is_empty());
}

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);
    s += 3;
    std::cout << "list " << list << std::endl;
    std::cout << "list_ " << list_ << std::endl;
    ASSERT_EQ(list_.get_size(), s);
    ASSERT_FALSE(list_.is_empty());

    ASSERT_EQ(list.get_size(), 0);
    ASSERT_TRUE(list.is_empty());
}

```