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

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Алгоритмы и струтуры данных»

Тема: Списки

Вариант 13

Студент гр. 0302 _____ Устинов Г.А.

Преподаватель _____ Родионова Е.А.

Санкт-Петербург

2022

Цель работы

Изучить структуру данных «Односвязный список». Реализовать односвязный список и заданные методы.

Задание

Вариант 13

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. Пример работы.
5. Листинг.

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

1. добавление в конец списка
2. добавление в начало списка
3. удаление последнего элемента
4. удаление первого элемента
5. добавление элемента по индексу (вставка перед элементом, который был ранее доступен по этому индексу)
6. получение элемента по индексу
7. удаление элемента по индексу
8. получение размера списка
9. удаление всех элементов списка

10. замена элемента по индексу на передаваемый
элемент

11. проверка на пустоту списка

Список методов, которые реализуются в отдельных вариантах:

12. меняет порядок элементов в списке на обратный

13. вставка другого списка в список, начиная с
индекса

14. вставка другого списка в конец

15. вставка другого списка в начало

16. проверка на содержание другого списка в списке, можно сделать типа int

17. поиск первого вхождения другого списка в список

18. поиск последнего вхождения другого списка в список

19. обмен двух элементов списка по индеусам

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

Выполнение работы

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

Таблица 1 — Поля класса *Node*

Тип	Название поля	Описание
std::shared_ptr<Node<T>>	*next*	Указатель на предыдущий элемент
T	*value*	Хранимый элемент

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

Класс *LinkedList* 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<T>>	*head_*	Указатель на первый элемент
std::shared_ptr<Node<T>>	*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).
- *void push_front(int)* - добавляет элемент в начало списка. Добавление производится за константное время — не зависит от длины списка - O(1).
- *void pop_back()* - удаляет элемент из конца списка. В случае применения для пустого списка ничего не делает. Поскольку нужно обновить значение указателя на последний элемент предыдущего элемента, на который указателя нет, нужно пройти весь список от начала до конца, поэтому метод работает за линейное время - O(n).
- *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).
- *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).
- *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).

- *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).
- *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.
- *void clear()* - удаляет все элементы из списка. Для начала «затирается» *tail_*, затем *head_*. 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).
- *bool is_empty()* - возвращает *true*, если список пуст и *false* в противном случае. Время работы константно, поскольку пустота определяется сравнение поля *size_* с нулём.
- *void swap(size_t first, size_t second)* — меняет местами два элемента по указанным индексам. Находит элементы по номерам, а также пердыдущие элементы, после чего меняет значения указателей, для обеспечения нового порядка следования элементов. Для нахождения

элементов проходит список от начала до конца, поэтому время работы линейно — O(n).

Дополнительно, для копирования элементов списка при создании был создан копирующий конструктор — *DlinkedList(const DlinkedList&)*, который копирует все элементы из списка-аргумента в создаваемый список.

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

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 односвязный список, а так же принципы методы её обработки. 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/list.hh**

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

#include <memory>
#include <stdexcept>

namespace list {

template<class T>
class LinkedList {
public:
   LinkedList() : head_(nullptr), tail_(nullptr), size_(0) {}
   LinkedList(const LinkedList &src);
   LinkedList(LinkedList &&src);
   ~LinkedList() { clear(); }

   void push_back(T value);
   void push_front(T value);
   void pop_back();
   void pop_front();

   void insert(T value, size_t i);
   void remove(size_t i);
   T &at(size_t i);
   void set(size_t i, T value);

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

   void swap(size_t first, size_t second);
private:
   template<class U>
   struct Node;

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

```
   size_t size_;
};


template<class T>
template<class U>
struct LinkedList<T>::Node {
   Node(U value) : next(nullptr), value(value) {}

   std::shared_ptr<Node> next;

   U value;
};


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

template<class T>
LinkedList<T>::LinkedList(LinkedList<T> &&src) : size_(src.size_)
{
   head_.swap(src.head_);
   tail_.swap(src.tail_);
}

template<class T>
void LinkedList<T>::push_back(T value)
{
   std::shared_ptr<Node<T>> node(new Node<T>(value));
   if (!tail_)
       head_ = node;
   else
       tail_->next = node;
   tail_ = node;
   ++size_;
}
```

```
template<class T>
void LinkedList<T>::push_front(T value)
{
   std::shared_ptr<Node<T>> node(new Node<T>(value));
   if (!head_)
       tail_ = node;
   else
       node->next = head_;
   head_ = node;
   ++size_;
}

template<class T>
void LinkedList<T>::pop_back()
{
   if (is_empty())
       return;
   if (size_ == 1) {
       head_.reset();
       tail_.reset();
   } else {
       std::shared_ptr<Node<T>> cur = head_;
       while (cur != tail_ && cur->next != tail_)
           cur = cur->next;
       tail_ = cur;
       tail_->next.reset();
   }
   --size_;
}

template<class T>
void LinkedList<T>::pop_front()
{
   if (is_empty())
       return;
   head_ = head_->next;
   if (!head_)
       tail_.reset();
   --size_;
}

template<class T>
void LinkedList<T>::insert(T value, size_t i)
{
```

```
   if (i > size_) // if i == size_, node will be inserted after the
last one
       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<T>> node(new Node<T>(value));
   std::shared_ptr<Node<T>> tmp;
   if (i == size_ - 1) {
       tmp = tail_;
   } else {
       tmp = head_;
       for (size_t j = 1; j != i; ++j)
           tmp = tmp->next;
   }
   node->next = tmp->next;
   tmp->next = node;
   ++size_;
}

template<class T>
void LinkedList<T>::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<T>> cur = head_;
   for (size_t j = 1; j != i; ++j)
       cur = cur->next;
   std::shared_ptr<Node<T>> del = cur->next;
   cur->next = del->next;
   del->next.reset();
   --size_;
}

template<class T>
T &LinkedList<T>::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<T>> tmp = head_->next;
    for (size_t j = 1; j != i; ++j)
        tmp = tmp->next;
    return tmp->value;
}

template<class T>
void LinkedList<T>::set(size_t i, T 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<T>> tmp = head_->next;
    for (size_t j = 1; j != i; ++j)
        tmp = tmp->next;
    tmp->value = value;
}

template<class T>
void LinkedList<T>::clear()
{
    if (is_empty())
        return;
    tail_.reset();
    head_.reset();
    size_ = 0;
}

template<class T>
void LinkedList<T>::swap(size_t first, size_t second)
{
    if (first >= size_ || second >= size_)
```

```
        throw std::out_of_range("Index out of range.");
    if (first == second)
        return;
    if (first > second) {
        size_t tmp = first;
        first = second;
        second = tmp;
    }
    std::shared_ptr<Node<T>> first_prev;
    std::shared_ptr<Node<T>> first_cur;
    std::shared_ptr<Node<T>> second_prev;
    std::shared_ptr<Node<T>> second_cur;

    std::shared_ptr<Node<T>> prev = nullptr;
    std::shared_ptr<Node<T>> cur = head_;
    for (size_t i = 0; i < size_; ++i) {
        if (i == first) {
            first_prev = prev;
            first_cur = cur;
        } else if (i == second) {
            second_prev = prev;
            second_cur = cur;
        }
        prev = cur;
        cur = cur->next;
    }

    if (tail_ == second_cur)
        tail_ = first_cur;
    if (head_ == first_cur)
        head_ = second_cur;
    else
        first_prev->next = second_cur;
    second_prev->next = first_cur;
    first_cur->next.swap(second_cur->next);
}

}       // namespace dlist

#endif  // DLIST_HH
```