**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по лабораторной работе №1**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: «Реализация линейных списков через классы в C++»**

**Вариант 15**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 9302 |  | Ленина М.Р. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

**Оглавление**

[1. Постановка задачи. Описание реализуемого класса и методов. 3](#_Toc55768246)

[2. Описание реализованных unit-тестов. 3](#_Toc55768247)

[3. Пример работы. 3](#_Toc55768248)

[4. Листинг. 3](#_Toc55768249)

[5. Вывод 3](#_Toc55768250)

# Постановка задачи. Описание реализуемого класса и методов.

Цель работы: Реализовать класс односвязного списка с набором методов.

Class Elem

|  |  |
| --- | --- |
| Элемент | Описание |
| int data | Целочисленные данные, хранящиеся в списке |
| Elem\* next | Указатель на следующий элемент списка |

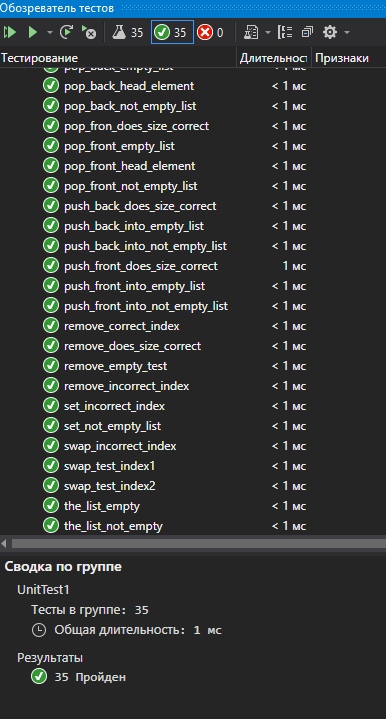
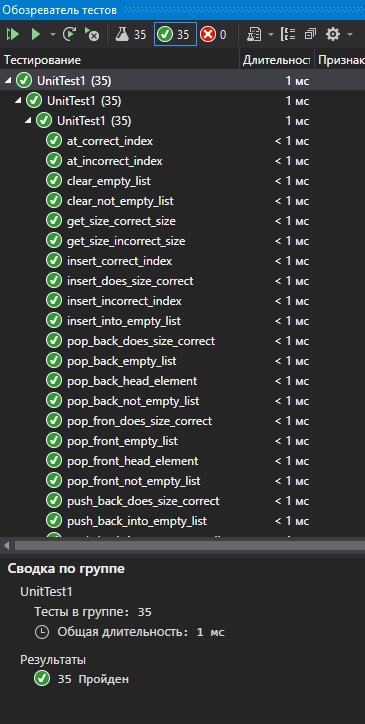
Class List

|  |  |  |  |
| --- | --- | --- | --- |
| Элемент | Описание | | |
| Elem\* head | Указатель на головной элемент списка | | |
| Elem\* tail | Указатель на последний элемент списка | | |
| size\_t size | Размер списка | | |
| Метод | | Описание | Оценка временной сложности |
| size\_t get\_size() const | | Возвращает значение размера списка | О(1) |
| bool isEmpty() | | Проверка пустой список или нет | О(1) |
| void push\_back(int newElem) | | Добавление элемента в конец списка | О(1) |
| void push\_front(int newElem) | | Добавление элемента в начало списка | О(1) |
| void pop\_back() | | Удаление последнего элемента | О(n) |
| void pop\_front() | | Удаление первого элемента | O(1) |
| void insert(int, size\_t) | | Добавление элемента по индексу (вставка перед элементом, который был ранее доступен по этому индексу) | О(n) |
| int at(size\_t index) const | | Получение элемента по индексу | O(n) |
| void remove(size\_t) | | Удаление элемента по индексу | O(n) |
| void print\_to\_console() | | Вывод элементов в консоль через разделитель | O(n) |
| void clear() | | Удаление всех элементов списка | O(n) |
| void set(size\_t, int) | | Замена элемента по индексу на передаваемый элемент | O(n) |
| void swap(size\_t, size\_t) | | Обмен двух элементов списка по индексам | O(n) |

# Описание реализованных unit-тестов.

|  |  |
| --- | --- |
| the\_list\_empty | Проверка метода isEmpty, пустой список |
| the\_list\_not\_empty | Проверка метода isEmpty, непустой список |
| get\_size\_correct\_size | Проверка метода get\_size, пустой список |
| get\_size\_incorrect\_size | Проверка метода get\_size, непустой список |
| at\_correct\_index | Проверка метода at, верный индекс |
| at\_incorrect\_index | Проверка метода at, индекс больше размера списка |
| push\_back\_into\_empty\_list | Проверка метода push\_back, вставка в пустой список |
| push\_back\_into\_not\_empty\_list | Проверка метода push\_back, вставка в непустой список |
| push\_back\_does\_size\_correct | Проверка метода push\_back, верный ли размер списка |
| push\_front\_into\_empty\_list | Проверка метода push\_front, вставка в пустой список |
| push\_front\_into\_not\_empty\_list | Проверка метода push\_front, вставка в непустой список |
| push\_front\_does\_size\_correct | Проверка метода push\_front, верный ли размер списка |
| pop\_back\_empty\_list | Проверка метода pop\_back, проверка на исключение при удалении из пустого списка |
| pop\_back\_head\_element | Проверка метода pop\_back, удаление головного элемента, проверка что список стал пустым |
| pop\_back\_not\_empty\_list | Проверка метода pop\_back, удаление из непустого списка |
| pop\_back\_does\_size\_correct | Проверка метода pop\_back, верный ли размер списка |
| pop\_front\_empty\_list | Проверка метода pop\_front, проверка на исключение при удалении из пустого списка |
| pop\_front\_head\_element | Проверка метода pop\_front, удаление головного элемента, проверка что список стал пустым |
| pop\_front\_not\_empty\_list | Проверка метода pop\_front, удаление из непустого списка |
| pop\_fron\_does\_size\_correct | Проверка метода pop\_front, верный ли размер списка |
| insert\_incorrect\_index | Проверка метода insert, индекс больше размера списка |
| insert\_correct\_index | Проверка метода insert, вставка в непустой список |
| insert\_into\_empty\_list | Проверка метода insert, вставка в пустой список |
| insert\_does\_size\_correct | Проверка метода insert, верный ли размер списка |
| remove\_incorrect\_index | Проверка метода remove, индекс больше размера списка |
| remove\_correct\_index | Проверка метода remove, удаление из непустого списка |
| remove\_does\_size\_correct | Проверка метода remove, верный ли размер списка |
| remove\_empty\_test | Проверка метода remove, проверка на исключение при удалении из пустого списка |
| clear\_empty\_list | Проверка метода clear, проверка на исключение при удалении из пустого списка |
| clear\_not\_empty\_list | Проверка метода clear, удаление непустого списка |
| set\_incorrect\_index | Проверка метода set, индекс больше размера списка |
| set\_not\_empty\_list | Проверка метода set, выставление значений в непустом списке |
| swap\_incorrect\_index | Проверка метода swap, индекс больше размера списка |
| swap\_test\_index1 | Проверка метода swap, верно ли значение у элемента с первым индексом |
| swap\_test\_index2 | Проверка метода swap, верно ли значение у элемента со вторым индексом |

# Пример работы.



# Листинг.

|  |
| --- |
| list.h |
| #pragma once  class List {  private:  class Elem  {  public:  Elem(int data, Elem\* next = nullptr) {  this->data = data;  this->next = next;  };  ~Elem() {};  int data;  Elem\* next;  };  Elem\* head;  Elem\* tail;  size\_t size;  public:  List();  ~List() {};  void reset\_list();  void add\_first(int newElem);  size\_t get\_size() const;  bool isEmpty();  void push\_back(int newElem);  void push\_front(int newElem);  void pop\_back();  void pop\_front();  void insert(int, size\_t);  int at(size\_t index) const;  void remove(size\_t);  void print\_to\_console();  void clear();  void set(size\_t, int);  void swap(size\_t, size\_t);  }; |
| list.cpp |
| #include "list.h"  #include <stdexcept>  #include <string>  #include <iostream>  using namespace std;  void List::reset\_list()  {  head = nullptr;  tail = nullptr;  }  List::List()  {  reset\_list();  size = 0;  }  void List::add\_first(int newElem)  {  head = new Elem(newElem);  tail = head;  }  //checks is the list is empty  bool List::isEmpty() {  return (head == nullptr);  }  //gets size of list  size\_t List::get\_size() const  {  return size;  }  //insert element in the end of list  void List::push\_back(int newElem)  {  if (size == 0) {  add\_first(newElem);  }  else {  tail->next = new Elem(newElem);  tail = tail->next;  }  size++;  }  //insert element in the begining of the list  void List::push\_front(int newElem)  {  if (size == 0) {  add\_first(newElem);  }  else {  head = new Elem(newElem, head);  }  size++;  }  //removes the last element  void List::pop\_back()  {  if (size == 0) {  throw out\_of\_range("Removing elements in an empty list is not possible");  return;  }  if (size == 1) {  delete head;  reset\_list();  }  else {  Elem\* current = head;  while (current->next != tail) {  current = current->next;  }  current->next = nullptr;  delete tail;  tail = current;  }  size--;  }  //removes the first element  void List::pop\_front() {  if (size == 0) {  throw out\_of\_range("Removing elements in an empty list is not possible");  return;  }  if (size == 1) {  delete head;  reset\_list();  }  else {  Elem\* current = head;    head = current->next;  delete current;  }  size--;  }  //insert the element at index (insertion before an item that was previously accessible at that index)  void List::insert(int insertData, size\_t index) {    if (index == 0) {  push\_front(insertData);  } else  if (index >= size) {  throw out\_of\_range("Index is greater than list size");  }  else {  size\_t count = 0;  Elem\* current = head;  while (count != index - 1) {  current = current->next;  count++;  }  Elem\* insertElem = new Elem(insertData, current->next);  current->next = insertElem;  size++;  }  }  //gets an element by index  int List::at(size\_t index) const  {  if (index >= size) {  throw out\_of\_range("Index is greater than list size");  }  else {  size\_t counter = 0;  Elem\* current = head;  while (counter != index) {  current = current->next;  counter++;  }  return current->data;  }  }  //deleting an element by index  void List::remove(size\_t index) {  if (index >= size) {  throw out\_of\_range("Index is greater than list size");  }  if (index == 0)  pop\_front();  else {  size\_t count = 0;  Elem\* current = head;  while (count != index - 1) {  current = current->next;  count++;  }  Elem\* deleting\_Element = current->next;  current->next = deleting\_Element->next;  delete deleting\_Element;  size--;  }  }  //output of list elements to the console  void List::print\_to\_console() {  Elem\* current = head;  while (current->next != tail) {  cout << current->data<<' ';  current = current->next;  }  cout << endl;  }  //removing all the list elements  void List::clear() {  if (size == 0) {  throw out\_of\_range("Removing elements in an empty list is not possible");  }  Elem\* current = head;  while (!(this->isEmpty())) {  current = head;  head = current->next;  delete current;  }  }  //replacing the elemen by index with the passed element  void List::set( size\_t index, int newData) {  if (index >= size) {  throw out\_of\_range("Index is greater than list size");  }  size\_t count = 0;  Elem\* current = head;  while (count != index) {  current = current->next;  count++;  }  current->data = newData;  }  //swapping two list elements by indices  void List::swap(size\_t index1, size\_t index2) {  if ((index1 >= size) || (index2 >= size)) {  throw out\_of\_range("Index is greater than list size");  }  Elem\* current\_1 = head;  size\_t count=0;  while (count != index1) {  current\_1 = current\_1->next;  count++;  }  count = 0;  Elem\* current\_2 = head;  while (count != index2) {  current\_2 = current\_2->next;  count++;  }  int buffer=current\_1->data;  current\_1->data = current\_2->data;  current\_2->data = buffer;  } |
| UnitTest1.cpp |
| #include "CppUnitTest.h"  #include "../astra\_l1/list.h"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace UnitTest1  {  TEST\_CLASS(UnitTest1)  {  public:  List\* lst;  TEST\_METHOD\_INITIALIZE(setUp)  {  lst = new List();  }  TEST\_METHOD\_CLEANUP(cleanUp)  {  delete lst;  }  TEST\_METHOD(the\_list\_empty)  {  Assert::IsTrue(lst->isEmpty());  }  TEST\_METHOD(the\_list\_not\_empty)  {  lst->push\_back(1);  Assert::IsFalse(lst->isEmpty());  }  TEST\_METHOD(get\_size\_correct\_size)  {  Assert::AreEqual(int(lst->get\_size()),0);  }  TEST\_METHOD(get\_size\_incorrect\_size)  {  lst->push\_back(1);  Assert::AreNotEqual(int(lst->get\_size()), 0);  }  TEST\_METHOD(at\_correct\_index)  {  lst->push\_back(1);  lst->push\_back(2);  lst->push\_back(3);  Assert::AreEqual(lst->at(1), 2);  }  TEST\_METHOD(at\_incorrect\_index)  {  try {  lst->at(-1);  }  catch (std::out\_of\_range e) {  Assert::AreEqual("Index is greater than list size", e.what());  }  }  TEST\_METHOD(push\_back\_into\_empty\_list)  {  lst->push\_back(1);  Assert::AreEqual(lst->at(0), 1);  }  TEST\_METHOD(push\_back\_into\_not\_empty\_list)  {  lst->push\_back(1);  lst->push\_back(2);  Assert::AreEqual(lst->at(1), 2);  }  TEST\_METHOD(push\_back\_does\_size\_correct)  {  lst->push\_back(1);  lst->push\_back(1);  lst->push\_back(1);  Assert::AreEqual(int(lst->get\_size()), 3);  }  TEST\_METHOD(push\_front\_into\_empty\_list)  {  lst->push\_front(1);  Assert::AreEqual(lst->at(0), 1);  }  TEST\_METHOD(push\_front\_into\_not\_empty\_list)  {  lst->push\_back(2);  lst->push\_front(1);  Assert::AreEqual(lst->at(0), 1);  }  TEST\_METHOD(push\_front\_does\_size\_correct)  {  lst->push\_front(1);  lst->push\_front(1);  lst->push\_front(1);  Assert::AreEqual(int(lst->get\_size()), 3);  }  TEST\_METHOD(pop\_back\_empty\_list)  {  try {  lst->pop\_back();  }  catch (std::out\_of\_range e) {  Assert::AreEqual("Removing elements in an empty list is not possible", e.what());  }  }  TEST\_METHOD(pop\_back\_head\_element)  {  lst->push\_back(1);  lst->pop\_back();  Assert::IsTrue(lst->isEmpty());  }  TEST\_METHOD(pop\_back\_not\_empty\_list)  {  lst->push\_back(1);  lst->push\_back(2);  lst->push\_back(3);  lst->pop\_back();  try {  lst->at(2);  }  catch (std::out\_of\_range e) {  Assert::AreEqual("Index is greater than list size", e.what());  }  }  TEST\_METHOD(pop\_back\_does\_size\_correct)  {  lst->push\_back(1);  lst->push\_back(2);  lst->push\_back(3);  lst->pop\_back();  Assert::AreEqual(int(lst->get\_size()), 2);  }  TEST\_METHOD(pop\_front\_empty\_list)  {  try {  lst->pop\_front();  }  catch (std::out\_of\_range e) {  Assert::AreEqual("Removing elements in an empty list is not possible", e.what());  }  }  TEST\_METHOD(pop\_front\_head\_element)  {  lst->push\_back(1);  lst->pop\_front();  Assert::IsTrue(lst->isEmpty());  }  TEST\_METHOD(pop\_front\_not\_empty\_list)  {  lst->push\_back(1);  lst->push\_back(2);  lst->push\_back(3);  lst->pop\_front();  Assert::AreEqual(lst->at(0), 2);  }  TEST\_METHOD(pop\_fron\_does\_size\_correct)  {  lst->push\_back(1);  lst->push\_back(2);  lst->push\_back(3);  lst->pop\_front();  Assert::AreEqual(int(lst->get\_size()), 2);  }  TEST\_METHOD(insert\_incorrect\_index)  {  try {  lst->insert(1,1);  }  catch (std::out\_of\_range e) {  Assert::AreEqual("Index is greater than list size", e.what());  }  }  TEST\_METHOD(insert\_correct\_index)  {  lst->push\_back(1);  lst->push\_back(3);  lst->push\_back(4);  lst->insert(2, 1);  Assert::AreEqual(lst->at(1), 2);  }  TEST\_METHOD(insert\_into\_empty\_list)  {  lst->insert(1, 0);  Assert::AreEqual(lst->at(0), 1);  }  TEST\_METHOD(insert\_does\_size\_correct)  {  lst->push\_back(2);  lst->push\_back(3);  lst->insert(1, 0);  Assert::AreEqual(int(lst->get\_size()), 3);  }  TEST\_METHOD(remove\_incorrect\_index)  {  try {  lst->remove(2);  }  catch (std::out\_of\_range e) {  Assert::AreEqual("Index is greater than list size", e.what());  }  }  TEST\_METHOD(remove\_correct\_index)  {  lst->push\_back(1);  lst->push\_back(2);  lst->remove(0);  Assert::AreEqual(lst->at(0), 2);  }  TEST\_METHOD(remove\_does\_size\_correct)  {  lst->push\_back(1);  lst->push\_back(2);  lst->push\_back(3);  lst->remove(1);  Assert::AreEqual(int(lst->get\_size()), 2);  }  TEST\_METHOD(remove\_empty\_test)  {  lst->push\_back(1);  lst->remove(0);  Assert::IsTrue(lst->isEmpty());  }  TEST\_METHOD(clear\_empty\_list)  {  try {  lst->clear();  }  catch (std::out\_of\_range e) {  Assert::AreEqual("Removing elements in an empty list is not possible", e.what());  }  }  TEST\_METHOD(clear\_not\_empty\_list)  {  lst->push\_back(1);  lst->push\_back(2);  lst->push\_back(3);  lst->clear();  Assert::IsTrue(lst->isEmpty());  }  TEST\_METHOD(set\_incorrect\_index)  {  try {  lst->set(2,1);  }  catch (std::out\_of\_range e) {  Assert::AreEqual("Index is greater than list size", e.what());  }  }  TEST\_METHOD(set\_not\_empty\_list)  {  lst->push\_back(1);  lst->push\_back(2);  lst->push\_back(3);  lst->set(1,5);  Assert::AreEqual(lst->at(1), 5);  }  TEST\_METHOD(swap\_incorrect\_index)  {  try {  lst->swap(2, 1);  }  catch (std::out\_of\_range e) {  Assert::AreEqual("Index is greater than list size", e.what());  }  }  TEST\_METHOD(swap\_test\_index1)  {  lst->push\_back(1);  lst->push\_back(2);  lst->push\_back(3);  lst->swap(0, 2);  Assert::AreEqual(lst->at(0), 3);  }  TEST\_METHOD(swap\_test\_index2)  {  lst->push\_back(1);  lst->push\_back(2);  lst->push\_back(3);  lst->swap(0, 2);  Assert::AreEqual(lst->at(2), 1);  }  };  } |

# Вывод

Я научилась реализовывать связные списки через классы и работать с ними. Также научилась проверять их корректную работу через unit-тесты.