**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

ОТЧЁТ

**по лабораторной работе №1**

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

**Тема: «Программирование двунаправленных связных списков»**

**Вариант 8.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9302 |  | Давтян С.Д. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

## Постановка задачи и описание реализуемого класса и методов.

Задание: Реализовать класс двусвязного списка с набором методов.

Для этого мне понадобились классы «Node» и «List».

class Node

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

**class List**

|  |  |  |
| --- | --- | --- |
| **Элемент** | **Описание** | |
| Node \*head; | Указатель на начало списка | |
| Node \*tail; | Указатель на конец списка | |
| size\_t size; | Переменная, хранящая размер списка | |
| **Метод** | **Описание** | **Оценка временной сложности** |
| void push\_back(int) | Добавление в конец списка | O(1) |
| void push\_front(int) | Добавление в начало списка | O(1) |
| void pop\_back() | Удаление последнего элемента | O(1) |
| void pop\_front() | Удаление первого элемента | O(1) |
| void insert(int, size\_t) | Добавление элемента по индексу | O(n/2) |
| int at(const int) | Получение элемента по индексу | O(n) |
| void remove(size\_t) | Удаление элемента по индексу | O(n/2) |
| size\_t GetSize() | Получение размера списка | O(1) |
| 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/2) |
| bool isEmpty() | Проверка на пустоту списка | O(1) |
| void push\_front(List) | Вставка другого списка в начало | O(n) |

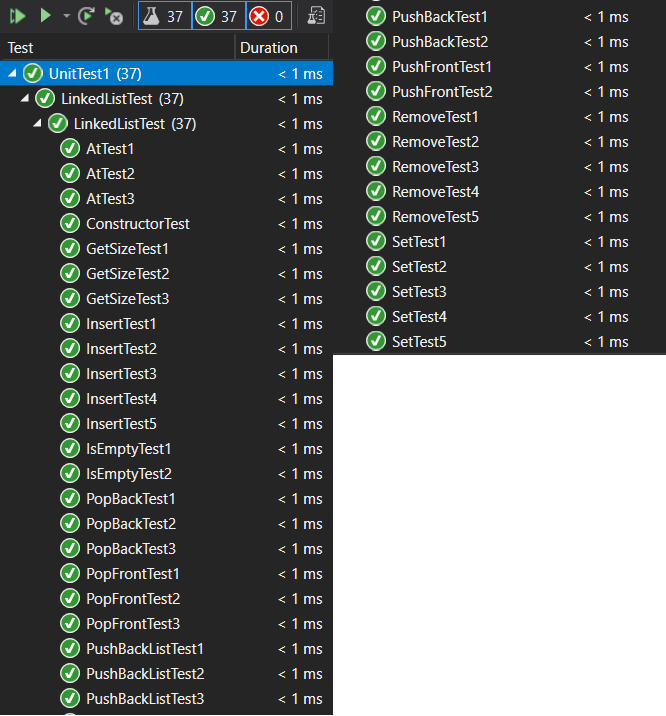
## Описание реализованных unit-тестов.

|  |  |
| --- | --- |
| Название теста | Что проверяет |
| AtTest1 | Проверка метода «at». |
| AtTest2 | Проверка метода «at». |
| AtTest3 | Проверка метода «at». Некорректный ввод |
| ConstructorTest | Проверка работы конструктора класса «List» |
| GetSizeTest1 | Проверка метода «GetSize». Пустой список |
| GetSizeTest2 | Проверка метода «GetSize». Непустой список |
| GetSizeTest3 | Проверка работы метода «GetSize» в непустом списке, в котором произведена какая-то работа |
| InsertTest1 | Проверка метода «insert». Вставка в начало непустого списка. |
| InsertTest2 | Проверка метода «insert». Вставка в начало пустого списка. |
| InsertTest3 | Проверка метода «insert». Вставка в конец непустого списка. |
| InsertTest4 | Проверка метода «insert». Вставка в произвольное место непустого списка. |
| InsertTest5 | Проверка метода «insert». Некорректный ввод |
| IsEmptyTest1 | Проверка метода «IsEmpty». Пустой список |
| IsEmptyTest2 | Проверка метода «IsEmpty». Непустой список |
| PopBackTest1 | Проверка метода «pop\_back». Непустой список |
| PopBackTest2 | Проверка метода «pop\_back». Список с одним элементом |
| PopBackTest3 | Проверка метода «pop\_back». Пустой список |
| PopFrontTest1 | Проверка метода «pop\_front». Непустой список |
| PopFrontTest2 | Проверка метода «pop\_front». Список с одним элементом |
| PopFrontTest3 | Проверка метода «pop\_front». Пустой список |
| PushBackListTest1 | Проверка метода «push\_back» для списков. Вставка непустого списка в начало непустого списка. |
| PushBackListTest2 | Проверка метода «push\_back» для списков. Вставка непустого списка в начало пустого списка. |
| PushBackListTest3 | Проверка метода «push\_back» для списков. Некорректное действие (вставка пустого списка в начало непустого) |
| PushBackTest1 | Проверка метода «push\_back». Пустой список |
| PushBackTest2 | Проверка метода «push\_back». Непустой список |
| PushFrontTest1 | Проверка метода «push\_front». Пустой список |
| PushFrontTest2 | Проверка метода «push\_front». Пустой список |
| RemoveTest1 | Проверка метода «remove». Удаление первого элемента непустого списка |
| RemoveTest2 | Проверка метода «remove». Удаление элемента из первой половины списка |
| RemoveTest3 | Проверка метода «remove». Удаление элемента из второй половины списка |
| RemoveTest4 | Проверка метода «remove». Удаление последнего элемента списка |
| RemoveTest5 | Проверка метода «remove». Некорректный ввод. |
| SetTest1 | Проверка метода «set». Замена первого элемента списка. |
| SetTest2 | Проверка метода «set». Замена элемента из первой половины списка |
| SetTest3 | Проверка метода «set». Замена элемента из второй половины списка |
| SetTest4 | Проверка метода «set». Замена последнего элемента списка. |
| SetTest5 | Проверка метода «set». Некорректный ввод. |

## Листинг

|  |
| --- |
| List.h |
| #pragma once  #include <iostream>  class List  {  public:  List();  List(const List &lst);  ~List();  void push\_back(int data);  void push\_front(int data);  void pop\_back();  void pop\_front();  void insert(size\_t index, int data);  void remove(size\_t index);  void clear();  void print\_to\_console();  size\_t GetSize();  int at(const int index);  void set(size\_t index, int data);  bool isEmpty();  void push\_front(List lst);  private:  class Node  {  public:  Node\* next;  Node\* prev;  int data;  Node(int data = 0, Node\* next = nullptr, Node\* prev =nullptr)  {  this->data = data;  this->next = next;  this->prev = prev;  }  };  Node\* head;  Node\* tail;  size\_t size;  }; |
| List.cpp |
| #include "List.h"  // constructor  List::List()  {  size = 0;  head = nullptr;  tail = nullptr;  }  // copy constructor  List::List(const List &lst)  {  size = 0;  head = nullptr;  tail = nullptr;  Node\* temp = lst.head;  for (int i = 0; i < lst.size; i++)  {  push\_back(temp->data);  temp = temp->next;  }  }  // destructor  List::~List()  {  clear();  }  // inserting an element at the end of the list  void List::push\_back(int data)  {  if (head == nullptr)  {  this->head = this->tail = new Node(data);  }  else  {  Node\* temp = new Node(data, nullptr, tail);  tail->next = temp;  tail = temp;  }  size++;  }  // inserting an element at the beginning of the list  void List::push\_front(int data)  {  if (!(isEmpty()))  {  Node\* lasthead = head;  head = new Node(data, head);  lasthead->prev = head;  }  else  {  head = new Node(data, head);  }  size++;  }  // remove the last element  void List::pop\_back()  {  if (isEmpty())  throw "Error! Linked list is Empty.";  else  {  Node\* todelete = tail;  tail = tail->prev;  delete todelete;  size--;  }  }  // remove the first element  void List::pop\_front()  {  if (head != nullptr) {  Node\* temp = this->head;  head = head->next;  delete temp;  size--;  }  else throw "Error! Linked list is Empty.";  }  // insert into an arbitrary place in the list by index  void List::insert(size\_t index, int data)  {  if (!isEmpty())  {  if (index >= size || index < 0)  throw "Error! Incorrect input.";  }  else  {  if (index != size)  throw "Error! Incorrect input.";  }  if (index == 0)  {  push\_front(data);  }  else  {  if (index == size)  push\_back(data);  else {  if (index < (size - 1) / 2)  {  Node\* previous = this->head;  for (size\_t i = 0; i < index - 1; i++)  {  previous = previous->next;  }  previous->next = new Node(data, previous->next, previous);  }  else  {  Node\* previous = this->tail;  for (size\_t i = size - 1; i > index - 1; i--)  {  previous = previous->prev;  }  previous->next = new Node(data, previous->next, previous);  }  size++;  }  }  }  // deleting an element by index  void List::remove(size\_t index)  {  if (index >= size || index < 0)  throw "Error! Incorrect input.";  if (index == 0)  pop\_front();  else  {  if (index < (size - 1) / 2)  {  Node\* previous = this->head;  for (size\_t i = 0; i < index - 1; i++)  {  previous = previous->next;  }  Node\* todelete = previous->next;  previous->next = todelete->next;  delete todelete;  }  else  {  Node\* previous = this->tail;  for (size\_t i = size - 1; i >= index; i--)  {  previous = previous->prev;  }  Node\* todelete = previous->next;  previous->next = todelete->next;  delete todelete;  }  size--;  }  }  // clear list  void List::clear()  {  while (size) {  pop\_front();  }  }  // output the list to the console  void List::print\_to\_console()  {  if (isEmpty())  throw "Linked List is empty.\n";  else  {  Node\* current = head;  for (int i = 0; i < GetSize()-1; i++)  {  std::cout << current->data << " - ";  current = current->next;  }  std::cout << current->data;  }  }  // getting list size  size\_t List::GetSize()  {  return size;  }  // getting an element by index  int List::at(const int index)  {  if (index >= size || index < 0)  throw "Error! Incorrect input.";  int counter = 0;  Node\* current = head;  while (current != nullptr)  {  if (counter == index)  return current->data;  current = current->next;  counter++;  }  }  // replacing the element by index with the passed element  void List::set(size\_t index, int data)  {  if (index >= size || index < 0)  throw "Error! Incorrect input.";  if (index == 0)  {  pop\_front();  push\_front(data);  }  else  {  if (index < (size - 1) / 2)  {  Node\* previous = this->head;  for (size\_t i = 0; i < index - 1; i++)  {  previous = previous->next;  }  Node\* todelete = previous->next;  previous->next = todelete->next;  delete todelete;  previous->next = new Node(data, previous->next, previous);  }  else  {  Node\* previous = this->tail;  for (size\_t i = size - 1; i >= index; i--)  {  previous = previous->prev;  }  Node\* todelete = previous->next;  previous->next = todelete->next;  delete todelete;  previous->next = new Node(data, previous->next, previous);  }    }  }  // checking the list for emptiness  bool List::isEmpty()  {  if (head == nullptr)  return true;  else return false;  }  // insert at the beginning of the list of another list  void List::push\_front(List lst)  {  if (!isEmpty() && lst.isEmpty())  throw "Error! Inserting an empty list at the beginning is not possible.";  Node\* temp = lst.tail;  for (size\_t i = 0; i < lst.GetSize(); i++) {  this->push\_front(temp->data);  temp = temp->prev;  }  } |
| UnitTest1.cpp |
| #include "pch.h"  #include "CppUnitTest.h"  #include "..\Program1\List.h"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace LinkedListTest  {  TEST\_CLASS(LinkedListTest)  {  public:    TEST\_METHOD(ConstructorTest)  {  List lst;  Assert::IsTrue (lst.GetSize() == 0);  }  TEST\_METHOD(IsEmptyTest1)  {  List lst;  Assert::IsTrue(lst.isEmpty());  }  TEST\_METHOD(IsEmptyTest2)  {  List lst;  lst.push\_back(3);  Assert::IsTrue(lst.isEmpty() == false);  }  TEST\_METHOD(AtTest1)  {  List lst;  lst.push\_back(3);  Assert::IsTrue(lst.at(0) == 3);  }  TEST\_METHOD(AtTest2)  {  List lst;  lst.push\_back(3);  lst.push\_front(2);  Assert::IsTrue(lst.at(0) == 2, L"Assert 1");  Assert::IsTrue(lst.at(1) == 3, L"Assert 2");  Assert::IsTrue(lst.GetSize() == 2, L"Assert 3");  }  TEST\_METHOD(AtTest3)  {  List lst;  lst.push\_back(3);  try  {  lst.at(-5);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Incorrect input.");  }  }  TEST\_METHOD(GetSizeTest1)  {  List lst;  Assert::IsTrue(lst.GetSize() == 0);  }  TEST\_METHOD(GetSizeTest2)  {  List lst;  lst.push\_back(3);  lst.push\_front(2);  Assert::IsTrue(lst.GetSize() == 2);  }  TEST\_METHOD(GetSizeTest3)  {  List lst;  lst.push\_back(3);  lst.push\_front(2);  lst.pop\_back();  lst.pop\_front();  Assert::IsTrue(lst.GetSize() == 0);  }  TEST\_METHOD(PushBackTest1)  {  List lst;  lst.push\_back(3);  Assert::IsTrue(lst.GetSize() == 1, L"Assert 1");  Assert::IsTrue(lst.at(0) == 3, L"Assert 2");  }  TEST\_METHOD(PushBackTest2)  {  List lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  Assert::IsTrue(lst.GetSize() == 4, L"Assert 1");  Assert::IsTrue(lst.at(3) == 3, L"Assert 2");  }  TEST\_METHOD(PushFrontTest1)  {  List lst;  lst.push\_front(3);  Assert::IsTrue(lst.GetSize() == 1, L"Assert 1");  Assert::IsTrue(lst.at(0) == 3, L"Assert 2");  }  TEST\_METHOD(PushFrontTest2)  {  List lst;  lst.push\_front(0);  lst.push\_front(1);  lst.push\_front(2);  lst.push\_front(3);  Assert::IsTrue(lst.GetSize() == 4, L"Assert 1");  Assert::IsTrue(lst.at(3) == 0, L"Assert 2");  }  TEST\_METHOD(PopBackTest1)  {  List lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  lst.pop\_back();  Assert::IsTrue(lst.GetSize() == 3);  }  TEST\_METHOD(PopBackTest2)  {  List lst;  lst.push\_back(0);  lst.pop\_back();  Assert::IsTrue(lst.GetSize() == 0);  }  TEST\_METHOD(PopBackTest3)  {  List lst;  try  {  lst.pop\_back();  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Linked list is Empty.");  }  }  TEST\_METHOD(PopFrontTest1)  {  List lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  lst.pop\_front();  Assert::IsTrue(lst.at(0) == 1, L"Assert 1");  Assert::IsTrue(lst.at(1) == 2, L"Assert 2");  Assert::IsTrue(lst.GetSize() == 3, L"Assert 3");  }  TEST\_METHOD(PopFrontTest2)  {  List lst;  lst.push\_back(0);  lst.pop\_front();  Assert::IsTrue(lst.isEmpty(), L"Assert 1");  }  TEST\_METHOD(PopFrontTest3)  {  List lst;  try  {  lst.pop\_front();  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Linked list is Empty.");  }  }  TEST\_METHOD(InsertTest1)  {  List lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.insert(0, 21);  Assert::IsTrue(lst.at(0) == 21, L"Assert 1");  Assert::IsTrue(lst.at(1) == 0, L"Assert 2");  Assert::IsTrue(lst.GetSize() == 4, L"Assert 3");  }  TEST\_METHOD(InsertTest2)  {  List lst;  lst.insert(0, 21);  Assert::IsTrue(lst.at(0) == 21, L"Assert 1");  Assert::IsTrue(lst.GetSize() == 1, L"Assert 2");  }  TEST\_METHOD(InsertTest3)  {  List lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.insert(2, 21);  Assert::IsTrue(lst.at(2) == 21, L"Assert 1");  Assert::IsTrue(lst.at(3) == 2, L"Assert 2");  Assert::IsTrue(lst.GetSize() == 4, L"Assert 3");  }  TEST\_METHOD(InsertTest4)  {  List lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  lst.push\_back(4);  lst.insert(2, 21);  Assert::IsTrue(lst.at(2) == 21, L"Assert 1");  Assert::IsTrue(lst.at(3) == 2, L"Assert 2");  Assert::IsTrue(lst.GetSize() == 6, L"Assert 3");  }  TEST\_METHOD(InsertTest5)  {  List lst;  lst.push\_back(0);  lst.push\_back(1);  try  {  lst.insert(-3, 21);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Incorrect input.");  }  }  TEST\_METHOD(SetTest1)  {  List lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.set(0, 21);  Assert::IsTrue(lst.at(0) == 21, L"Assert 1");  Assert::IsTrue(lst.at(1) == 1, L"Assert 2");  Assert::IsTrue(lst.GetSize() == 3, L"Assert 3");  }  TEST\_METHOD(SetTest2)  {  List lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  lst.push\_back(4);  lst.push\_back(5);  lst.push\_back(6);  lst.push\_back(7);  lst.push\_back(8);  lst.set(2, 21);  Assert::IsTrue(lst.at(2) == 21, L"Assert 1");  Assert::IsTrue(lst.at(3) == 3, L"Assert 2");  Assert::IsTrue(lst.GetSize() == 9, L"Assert 3");  }  TEST\_METHOD(SetTest3)  {  List lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  lst.push\_back(4);  lst.push\_back(5);  lst.push\_back(6);  lst.push\_back(7);  lst.push\_back(8);  lst.set(7, 21);  Assert::IsTrue(lst.at(7) == 21, L"Assert 1");  Assert::IsTrue(lst.at(8) == 8, L"Assert 2");  Assert::IsTrue(lst.GetSize() == 9, L"Assert 3");  }  TEST\_METHOD(SetTest4)  {  List lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  lst.push\_back(4);  lst.push\_back(5);  lst.push\_back(6);  lst.push\_back(7);  lst.push\_back(8);  lst.set(8, 21);  Assert::IsTrue(lst.at(8) == 21, L"Assert 1");  Assert::IsTrue(lst.at(7) == 7, L"Assert 2");  Assert::IsTrue(lst.GetSize() == 9, L"Assert 3");  }  TEST\_METHOD(SetTest5)  {  List lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  try  {  lst.set(-3, 21);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Incorrect input.");  }  }  TEST\_METHOD(PushBackListTest1)  {  List lst;  lst.push\_back(5);  lst.push\_back(6);  List lst2;  lst2.push\_back(3);  lst2.push\_back(4);  lst.push\_front(lst2);  Assert::IsTrue(lst.at(1) == 4, L"Assert 1");  Assert::IsTrue(lst.GetSize() == 4, L"Assert 2");  }  TEST\_METHOD(PushBackListTest2)  {  List lst;  List lst2;  lst2.push\_back(0);  lst2.push\_back(1);  lst.push\_front(lst2);  Assert::IsTrue(lst.at(0) == 0, L"Assert 1");  Assert::IsTrue(lst.GetSize() == 2, L"Assert 2");  }  TEST\_METHOD(PushBackListTest3)  {  List lst;  lst.push\_back(5);  lst.push\_back(6);  List lst2;  try  {  lst.push\_front(lst2);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Inserting an empty list at the beginning is not possible.");  }  }  TEST\_METHOD(RemoveTest1)  {  List lst;  lst.push\_back(0);  lst.push\_back(1);  lst.remove(0);  Assert::IsTrue(lst.at(0) == 1, L"Assert 1");  Assert::IsTrue(lst.GetSize() == 1, L"Assert 2");  }  TEST\_METHOD(RemoveTest2)  {  List lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  lst.push\_back(4);  lst.push\_back(5);  lst.push\_back(6);  lst.push\_back(7);  lst.push\_back(8);  lst.remove(2);  Assert::IsTrue(lst.at(2) == 3, L"Assert 1");  Assert::IsTrue(lst.GetSize() == 8, L"Assert 2");  }  TEST\_METHOD(RemoveTest3)  {  List lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  lst.push\_back(4);  lst.push\_back(5);  lst.push\_back(6);  lst.push\_back(7);  lst.push\_back(8);  lst.remove(6);  Assert::IsTrue(lst.at(6) == 7, L"Assert 1");  Assert::IsTrue(lst.GetSize() == 8, L"Assert 2");  }  TEST\_METHOD(RemoveTest4)  {  List lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  lst.push\_back(4);  lst.push\_back(5);  lst.remove(5);  Assert::IsTrue(lst.at(4) == 4, L"Assert 1");  Assert::IsTrue(lst.GetSize() == 5, L"Assert 2");  }  TEST\_METHOD(RemoveTest5)  {  List lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  try  {  lst.remove(-2);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Incorrect input.");  }  }  };  } |
| Main.cpp |
| #include <iostream>  #include "List.h"  using namespace std;  int main()  {  List lst;  lst.push\_back(5);  lst.push\_back(6);  lst.push\_back(7);  lst.push\_back(8);    List lst2;  lst2.push\_back(0);  lst2.push\_back(1);  lst2.push\_back(2);  lst2.push\_back(3);  lst2.push\_back(4);  lst.push\_front(lst2);  cout << lst2.at(2) << endl << endl;  cout << lst.at(2) << endl << endl;  lst.print\_to\_console();  return 0;  } |

## Результат выполнения всех unit-тестов



## Вывод

В данной лабораторной работе мы научились реализовывать связные списки и работать с ними. Также мы научились проверять работу написанной программы с помощью unit-тестов.