**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: «Изучение и практическое применение однонаправленных списков. Реализация модульного тестирования»**

**Вариант 7**

Студент гр. 9302 \_\_\_\_\_\_\_\_\_\_\_\_ Горбанев А.В

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_ Тутуева А.В.

Санкт-Петербург

2020

Оглавление

[1. Постановка задачи и описание реализуемого класса и методов 3](#_Toc55082721)

[2. Описание реализованных unit-тестов 3](#_Toc55082722)

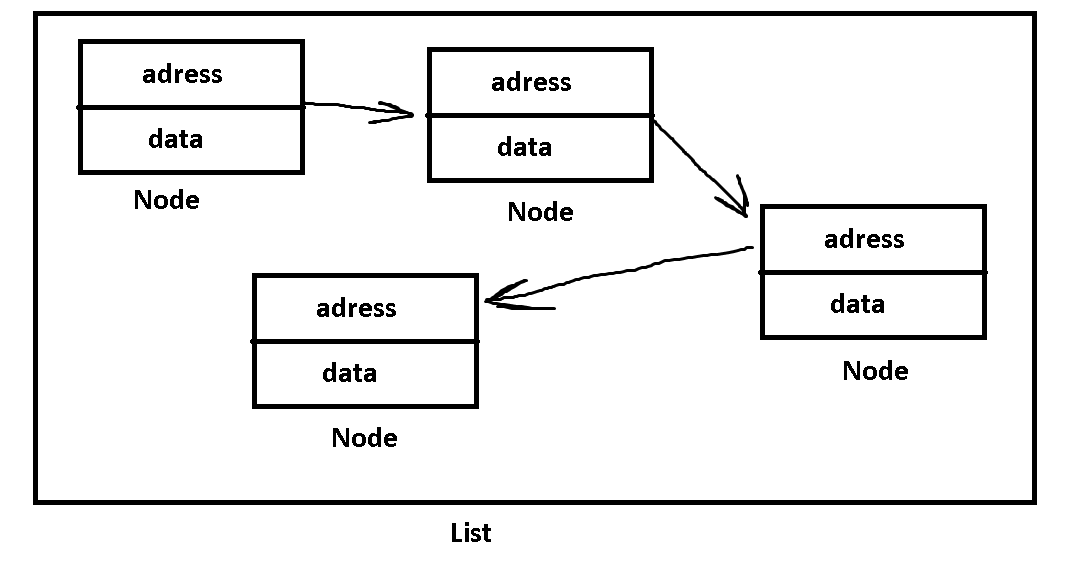
[3. Код программы 4](#_Toc55082723)

[4. Пример работы 11](#_Toc55082724)

[5. Вывод 12](#_Toc55082725)

# Постановка задачи и описание реализуемого класса и методов

Реализовать класс связного однонаправленного списка с набором методов. Использую для данной задачи класс List , содержащий в себе класс элементов Node.



|  |  |  |
| --- | --- | --- |
| **Метод** | **Описание** | **Оценка временной сложности** |
| void push\_back(int) | Добавление в конец списка | O(n) |
| void push\_front(int) | Добавление в начало списка | O(n) |
| void pop\_back() | Удаление последнего элемента | O(n-1) |
| void pop\_front() | Удаление первого элемента | O(1) |
| void insert(int, size\_t) | Добавление элемента по индексу | O(n) |
| int at(const int) | Получение элемента по индексу | O(n) |
| void remove(size\_t) | Удаление элемента по индексу | O(n) |
| 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) |
| bool isEmpty() | Проверка на пустоту списка | O(1) |
| void push\_front(list) | Вставка другого списка в начало | O(n\*m),n-размер вставляемого списка,m-размер списка в который вставляют |

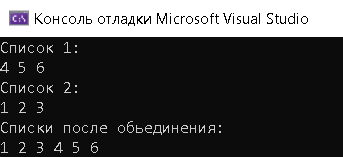
# Описание реализованных unit-тестов

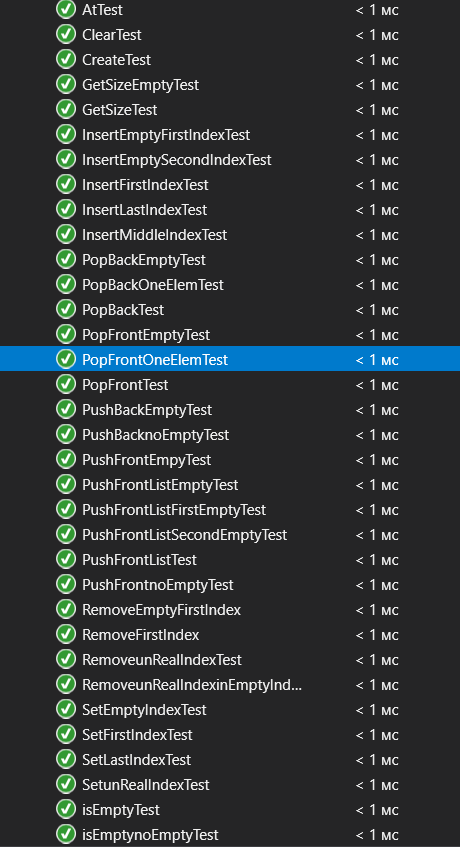
|  |  |
| --- | --- |
| Имя теста | Описание |
| CreateTest | Проверка работы конструктора |
| PushBackEmptyTest | Проверка добавления элемента в конец пустого списка |
| PushBacknoEmptyTest | Проверка добавления элемента в конец непустого списка |
| PushFrontEmpyTest | Проверка добавления элемента в начало пустого списка |
| PushFrontnoEmptyTest | Проверка добавления элемента в начало непустого списка |
| PopBackEmptyTest | Проверка удаления элемента из конца пустого списка |
| PopBackOneElemTest | Проверка удаления элемента из конца списка с одним элементом |
| PopBackTest | Проверка удаления элемента из конца непустого списка |
| PopFrontEmptyTest | Проверка удаления элемента из начала пустого списка |
| PopFrontOneElemTest | Проверка удаления элемента из начала списка с одним элементом |
| PopFrontTest | Проверка удаления элемента из начала непустого списка |
| InsertEmptySecondIndexTest | Проверка вставки элемента в пустой список на вторую позицию |
| InsertEmptyFirstIndexTest | Проверка вставки элемента в пустой список на первую позицию |
| InsertFirstIndexTest | Проверка вставки элемента в непустой список на первую позицию |
| InsertLastIndexTest | Проверка вставки элемента в непустой список на последнюю позицию |
| InsertMiddleIndexTest | Проверка вставки элемента в середину непустого списка |
| AtTest | Проверка получения элемента по индексу |
| RemoveEmptyFirstIndex | Проверка удаления элемента из пустого списка с первой позиции |
| RemoveFirstIndex | Проверка удаления элемента из непустого списка с первой позиции |
| RemoveunRealIndexinEmptyIndex | Проверка удаления элемента с несуществующей позиции в пустом списке |
| RemoveunRealIndexTest | Проверка удаления элемента с несуществующей позиции в непустом списке |
| GetSizeEmptyTest | Проверка получения размера пустого списка |
| GetSizeTest | Проверка получения размера непустого списка |
| ClearTest | Проверка на удаления списка |
| SetunRealIndexTest | Проверка замены несуществующего индекса в непустом списке |
| SetEmptyIndexTest | Проверка замены несуществующего индекса в пустом списке |
| SetFirstIndexTest | Проверка замены значения первого элемента в непустом списке |
| SetLastIndexTest | Проверка замены значения последнего элемента в непустом списке |
| isEmptynoEmptyTest | Проверка определения пустоты непустого списка |
| isEmptyTest | Проверка определения пустоты пустого списка |
| PushFrontListTest | Проверка вставки одного непустого в начало другого непустого |
| PushFrontListSecondEmptyTest | Проверка вставки пустого списка в непустой |
| PushFrontListFirstEmptyTest | Проверка вставки непустого списка в пустой |
| PushFrontListEmptyTest | Проверка вставки одного пустого в начало другого пустого |

# Код программы

|  |
| --- |
| List.h |
| #pragma once  #include<iostream>  #ifndef LIST\_H  #define LIST\_H  using namespace std;  class List {  public:  List();  void push\_back(int data);  void push\_front(int data);  void pop\_back();  void pop\_front();  void clear();  int GetSize() { return Size; };  void insert(int data, size\_t index);  int at(size\_t index);  void remove(size\_t index);  void print\_to\_console();  void set(size\_t index, int data);  bool isEmpty();  void push\_front(List lst);  private:  class Node {  public:  Node\* pNext;  int data;  Node(int data = NULL, Node\* pNext = nullptr) {  this->data = data;  this->pNext = pNext;  }  };  int Size;  Node\* head;  };  List::List() {  Size = 0;  head = nullptr;  }  void List::push\_back(int data)  {  if (head == nullptr) {  head = new Node(data);  }  else {  Node\* current = this->head;  while (current->pNext != nullptr) {  current = current->pNext;  }  current->pNext = new Node(data);  }  Size++;  }  void List::push\_front(int data)  {  if (head == nullptr) {  head = new Node(data);  }  else {  Node\* current = this->head;  int next\_item;  next\_item = current->data;  while (current->pNext != nullptr) {  int swap;  current = current->pNext;  swap = current->data;  current->data = next\_item;  next\_item = swap;  }  current->pNext = new Node(next\_item);  head->data = data;  }  Size++;  }  void List::pop\_back()  {  try {  Node\* current = this->head;  if (Size == 1) {  delete current;  Size--;  head = nullptr;  }  else if (Size > 1) {  for (int i = 0; i < Size - 2; i++) {  current = current->pNext;  }  Node\* temp = current->pNext;  delete temp;  current->pNext = nullptr;  Size--;  }  else throw "List is Empty!";  }  catch (const char\* warning) {  cout << warning << endl;  if (warning == "List is Empty!") return;  }    }  void List::pop\_front()  {  try{  Node\* temp = head;  if (Size == 1) {  delete temp;  Size--;  head = nullptr;  }  else if (Size > 1) {  head = head->pNext;  delete temp;  Size--;  }  else throw "List is Empty!";  }  catch (const char\* warning) {  cout << warning << endl;  if (warning == "List is Empty!") return;  }  }  void List::clear()  {  while (Size) {  pop\_front();  }  }  void List::insert(int data, size\_t index)  {  try {  Node\* current = this->head;  if (head == nullptr) throw "List is Empty!";  if (index > Size - 1) throw "Wrong Index!";  int counter = 0;  while (current != nullptr) {  if (counter == index) {  break;  }  current = current->pNext;  counter++;  }  int next\_item;  next\_item = current->data;  current->data = data;  while (current->pNext != nullptr) {  int swap;  current = current->pNext;  swap = current->data;  current->data = next\_item;  next\_item = swap;  }  current->pNext = new Node(next\_item);  Size++;  }  catch (const char\* warning) {  cout << warning << endl;  return;  }  }  int List::at(size\_t index)  {  Node\* current = this->head;  int counter = 0;  while (current != nullptr) {  if (counter == index) {  return current->data;  }  current = current->pNext;  counter++;  }  return 0;  }  void List::remove(size\_t index)  {  try {  if (head == nullptr) throw "List is Empty!";  if (index > Size - 1) {  throw "Wrong Index!";  }  else if (index == 0) pop\_front();  else if (index == Size - 1) pop\_back();  else {  Node\* temp = this->head;  Node\* current = this->head;  for (int i = 0; i < index; i++) {  temp = temp->pNext;  }  for (int i = 0; i < index - 1; i++) {  current = current->pNext;  }  current->pNext = temp->pNext;  delete temp;  Size--;  }  }  catch (const char\* warning) {  cout << warning << endl;  return;  }  }  void List::print\_to\_console()  {  Node\* current = this->head;  if (current != nullptr) {  while (current != nullptr) {  cout << current->data << " ";  current = current->pNext;  }  }  }  void List::set(size\_t index, int data)  {  try {  Node\* current = this->head;  if (current == nullptr) throw "List is Empty!";  if (index > Size - 1) throw "Wrong Index!";  for (int i = 0; i < index; i++) {  current = current->pNext;  }  current->data = data;  }  catch (const char\* warning) {  cout << warning << endl;  return;  }  }  bool List::isEmpty()  {  if (Size == 0) {  return true;  }  else return false;  }  void List::push\_front(List lst)  {  for (int i = lst.GetSize()-1; i >= 0; i--) {  push\_front(lst.at(i));  }  }  #endif |
| Main.cpp |
| #include<iostream>  #include"list.h"  int main() {  setlocale(LC\_ALL, "russian");  List lst;  List lst1;      lst.push\_back(4);  lst.push\_back(5);  lst.push\_back(6);  lst1.push\_back(1);  lst1.push\_back(2);  lst1.push\_back(3);  cout << "Список 1:" << endl;  lst.print\_to\_console();  cout << endl;  cout << "Список 2:" << endl;  lst1.print\_to\_console();  cout << endl;  cout << "Списки после обьединения:" << endl;  lst.push\_front(lst1);  lst.print\_to\_console();      } |
| Laba1test.cpp |
| #include "pch.h"  #include "CppUnitTest.h"  #include "../1laba/list.h"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace laba1Test  {  TEST\_CLASS(laba1Test)  {  public:    TEST\_METHOD(CreateTest)  {  List lst;  Assert::IsTrue(lst.GetSize() == 0);  }  TEST\_METHOD(PushBackEmptyTest)  {  List lst;  lst.push\_back(1);  Assert::IsTrue((lst.GetSize() == 1) && (lst.at(0) == 1));  }  TEST\_METHOD(PushBacknoEmptyTest)  {  List lst;  lst.push\_back(1);  lst.push\_back(2);  Assert::IsTrue((lst.GetSize() == 2) && (lst.at(1) == 2)&&(lst.at(0)==1));  }  TEST\_METHOD(PushFrontEmpyTest)  {  List lst;  lst.push\_front(1);  Assert::IsTrue((lst.GetSize() == 1)&&(lst.at(0)==1));  }  TEST\_METHOD(PushFrontnoEmptyTest)  {  List lst;  lst.push\_front(1);  lst.push\_front(2);  Assert::IsTrue((lst.GetSize() == 2) && (lst.at(1) == 1) && (lst.at(0) == 2));  }  TEST\_METHOD(PopBackEmptyTest)  {  List lst;  try {  lst.pop\_back();  }  catch (const char\* warning) {  Assert::AreEqual(warning, "List is Empty!");  }  Assert::IsTrue((lst.GetSize() == 0)&&(lst.at(0)==NULL));  }  TEST\_METHOD(PopBackOneElemTest)  {  List lst;  lst.push\_back(1);  lst.pop\_back();  Assert::IsTrue((lst.GetSize() == 0)&&(lst.at(0)==NULL));  }  TEST\_METHOD(PopBackTest)  {  List lst;  lst.push\_back(1);  lst.push\_back(2);  lst.pop\_back();  Assert::IsTrue((lst.GetSize() == 1)&&(lst.at(0)==1));  }  TEST\_METHOD(PopFrontEmptyTest)  {  List lst;  try {  lst.pop\_front();  }  catch (const char\* warning) {  Assert::AreEqual(warning, "List is Empty!");  }  Assert::IsTrue((lst.GetSize() == 0) && (lst.at(0) == NULL));  }  TEST\_METHOD(PopFrontOneElemTest)  {  List lst;  lst.push\_back(1);  lst.pop\_front();  Assert::IsTrue((lst.GetSize() == 0) && (lst.at(0) == NULL));  }  TEST\_METHOD(PopFrontTest)  {  List lst;  lst.push\_back(1);  lst.push\_back(2);  lst.pop\_front();  Assert::IsTrue((lst.GetSize() == 1) && (lst.at(0) == 2));  }  TEST\_METHOD(InsertEmptySecondIndexTest) {  List lst;  try {  lst.insert(2, 1);  }  catch (const char\* warning) {  Assert::AreEqual(warning, "Wrong Index!");  }  Assert::IsTrue((lst.GetSize() == 0) && (lst.at(0) == NULL));  }  TEST\_METHOD(InsertEmptyFirstIndexTest) {  List lst;  try {  lst.insert(2, 0);  }  catch (const char\* warning) {  Assert::AreEqual(warning, "List is Empty!");  }  Assert::IsTrue((lst.GetSize() == 0) && (lst.at(0) == NULL));  }  TEST\_METHOD(InsertFirstIndexTest) {  List lst;  lst.push\_back(1);  lst.insert(2, 0);  Assert::IsTrue((lst.GetSize() == 2) && (lst.at(0) == 2));  }  TEST\_METHOD(InsertLastIndexTest) {  List lst;  lst.push\_back(1);  lst.push\_back(2);  lst.insert(3, 1);  Assert::IsTrue((lst.GetSize() == 3) && (lst.at(1) == 3));  }  TEST\_METHOD(InsertMiddleIndexTest) {  List lst;  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  lst.push\_back(4);  lst.push\_back(5);  lst.insert(10, 2);  Assert::IsTrue((lst.GetSize() == 6) && (lst.at(2) == 10));  }  TEST\_METHOD(AtTest) {  List lst;  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  int index = lst.at(2);  Assert::IsTrue(index = 3);  }  TEST\_METHOD(RemoveEmptyFirstIndex) {  List lst;  try {  lst.remove(0);  }  catch (const char\* warning) {  Assert::AreEqual(warning, "List is Empty!");  }  Assert::IsTrue((lst.GetSize() == 0) && (lst.at(0) == NULL));  }  TEST\_METHOD(RemoveFirstIndex) {  List lst;  lst.push\_back(1);  lst.remove(0);  Assert::IsTrue((lst.GetSize() == 0) && (lst.at(0) == NULL));  }  TEST\_METHOD(RemoveunRealIndexinEmptyIndex) {  List lst;  try {  lst.remove(2);  }  catch (const char\* warning) {  Assert::AreEqual(warning, "Wrong Index!");  }  Assert::IsTrue((lst.GetSize() == 0) && (lst.at(0) == NULL));  }  TEST\_METHOD(RemoveunRealIndexTest) {  List lst;  try {  lst.push\_back(1);  lst.push\_back(2);  lst.remove(2);  }  catch (const char\* warning) {  Assert::AreEqual(warning, "Wrong Index!");  }  Assert::IsTrue((lst.GetSize() == 2) && (lst.at(0) == 1) && (lst.at(1) == 2));  }  TEST\_METHOD(GetSizeEmptyTest) {  List lst;  Assert::IsTrue(lst.GetSize() == 0);  }  TEST\_METHOD(GetSizeTest) {  List lst;  lst.push\_back(1);  lst.push\_back(2);  Assert::IsTrue(lst.GetSize() == 2);  }  TEST\_METHOD(ClearTest) {  List lst;  lst.push\_back(1);  lst.push\_back(2);  lst.clear();  Assert::IsTrue((lst.GetSize() == 0)&&(lst.at(0)==NULL));  }  TEST\_METHOD(SetunRealIndexTest) {  List lst;  try {  lst.push\_back(1);  lst.push\_back(2);  lst.set(3, 2);  }  catch (const char\* warning) {  Assert::AreEqual(warning, "Wrong Index!");  }  Assert::IsTrue((lst.GetSize() == 2)&&(lst.at(0)==1)&&(lst.at(1)==2));  }  TEST\_METHOD(SetEmptyIndexTest) {  List lst;  try {  lst.set(3, 2);  }  catch (const char\* warning) {  Assert::AreEqual(warning, "List is Empty!");  }  Assert::IsTrue((lst.GetSize() == 0) && (lst.at(0) == NULL));  }  TEST\_METHOD(SetFirstIndexTest) {  List lst;  lst.push\_back(1);  lst.set(0, 3);  Assert::IsTrue((lst.GetSize() == 1) && (lst.at(0) == 3));  }  TEST\_METHOD(SetLastIndexTest) {  List lst;  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  lst.set(2, 5);  Assert::IsTrue((lst.GetSize() == 3) && (lst.at(0) == 1)&&(lst.at(1)==2)&&(lst.at(2)==5));  }  TEST\_METHOD(isEmptynoEmptyTest) {  List lst;  lst.push\_back(1);  Assert::IsTrue(lst.isEmpty()==false);  }  TEST\_METHOD(isEmptyTest) {  List lst;  Assert::IsTrue(lst.isEmpty() == true);  }  TEST\_METHOD(PushFrontListTest) {  List lst;  List lst1;  lst.push\_back(2);  lst.push\_back(3);  lst1.push\_back(1);  lst.push\_front(lst1);  Assert::IsTrue((lst.GetSize() == 3) && (lst.at(0) == 1) && (lst.at(1) == 2) && (lst.at(2) == 3));  }  TEST\_METHOD(PushFrontListSecondEmptyTest) {  List lst;  List lst1;  lst.push\_back(2);  lst.push\_back(3);  lst.push\_front(lst1);  Assert::IsTrue((lst.GetSize() == 2) && (lst.at(0) == 2) && (lst.at(1) == 3));  }  TEST\_METHOD(PushFrontListFirstEmptyTest) {  List lst;  List lst1;  lst1.push\_back(2);  lst1.push\_back(3);  lst.push\_front(lst1);  Assert::IsTrue((lst.GetSize() == 2) && (lst.at(0) == 2) && (lst.at(1) == 3));  }  TEST\_METHOD(PushFrontListEmptyTest) {  List lst;  List lst1;  lst.push\_front(lst1);  Assert::IsTrue((lst.GetSize() == 0) && (lst.at(0) == NULL));  }  };  } |

# Пример работы





# Вывод

При выполнении данной лабораторной работы я познакомился с модульным тестированием написанного кода.