**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 2.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9302 |  | Новокрещенов Д.К. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

## Постановка задачи и описание реализуемого класса и методов.

Для решения поставленной задачи необходимо реализовать двунаправленного связного списка с набором методов.

Для этого мне понадобились структура «Node» и класс «BiList».

struct Node

|  |  |
| --- | --- |
| **Компонент** | **Описание** |
| Node \*prev; | Указатель на предыдущий элемент списка |
| int value; | Тип данных, который хранится в списке. |
| Node \*next; | Указатель на следующий элемент списка |

**class BiList**

|  |  |  |
| --- | --- | --- |
| **Компонент** | **Описание** | |
| Node \*head; | Указатель на начало списка | |
| Node \*last; | Указатель на конец списка | |
| size\_t size; | Переменная, хранящая размер списка | |
| **Метод** | **Описание** | **Оценка временной сложности** |
| void push\_back(int) | Вставка нового элемента в конец списка | O(1) |
| void push\_front(int) | Вставка нового элемента в начало списка | O(1) |
| void pop\_back() | Удаление элемента из конца списка | O(1) |
| void pop\_front() | Удаление элемента из конца списка | O(1) |
| void insert(int, size\_t) | Вставка нового элемента по индексу | O(n) |
| int at(int) | Получение элемента по индексу | O(n) |
| void remove(size\_t) | Удаление элемента по индексу | O(n) |
| size\_t get\_size() | Получение размера списка | 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 | O(n) |
| bool isEmpty() | Проверка на пустоту списка | O(1) |
| void reverse() | Обращение списка | O(n) |

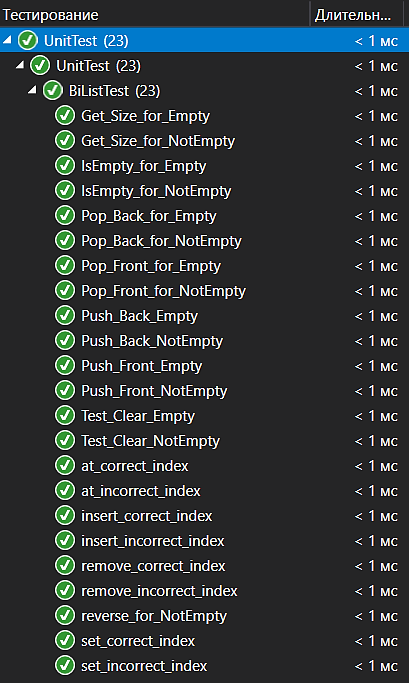
## Описание реализованных unit-тестов.

|  |  |
| --- | --- |
| **Наименование теста** | **Описание** |
| IsEmpty\_for\_NotEmpty | Проверка работы метода «isEmpty» на непустом списке |
| IsEmpty\_for\_Empty | Проверка работы метода «isEmpty» на пустом списке |
| Get\_Size\_for\_NotEmpty | Проверка работы метода «get\_size» на непустом списке |
| Get\_Size\_for\_Empty | Проверка работы метода «get\_size» на пустом списке |
| at\_correct\_index | Проверка работы метода «at» при корректном вводе |
| at\_incorrect\_index | Проверка работы метода «at» при некорректном вводе |
| Push\_Back\_NotEmpty | Проверка работы метода «push\_back» на непустом списке |
| Push\_Front\_NotEmpty | Проверка работы метода «push\_front» на непустом списке |
| Pop\_Back\_NotEmpty | Проверка работы метода «pop\_back» на непустом списке |
| Pop\_Front\_NotEmpty | Проверка работы метода «pop\_front» на непустом списке |
| Push\_Back\_Empty | Проверка работы метода «push\_back» на пустом списке |
| Push\_Front\_Empty | Проверка работы метода «push\_front» на пустом списке |
| Pop\_Back\_Empty | Проверка работы метода «pop\_back» на пустом списке |
| Pop\_Front\_Empty | Проверка работы метода «pop\_front» на пустом списке |
| remove\_correct\_index | Проверка работы метода «remove» при корректном вводе |
| remove\_incorrect\_index | Проверка работы метода «remove» при некорректном вводе |
| set\_correct\_index | Проверка работы метода «set» при корректном вводе |
| set\_incorrect\_index | Проверка работы метода «set» при некорректном вводе |
| Test\_Clear\_NotEmpty | Проверка работы метода «clear» на непустом списке |
| Test\_Clear\_Empty | Проверка работы метода «clear» на пустом списке |
| reverse\_for\_NotEmpty | Проверка работы метода «reverse» на непустом списке |
| insert\_correct\_index | Проверка работы метода «insert» при корректном вводе |
| insert\_incorrect\_index | Проверка работы метода «insert» при некорректном вводе |

## Листинги программы и тестов

|  |
| --- |
| List.h |
| #pragma once  #include <cstddef>  struct Node  {  int data;  Node\* prev;  Node\* next;  };  class BiList  {  public:  BiList();  ~BiList();  void push\_back(int);  void push\_front(int);  void pop\_back();  void pop\_front();  void insert(int, size\_t);  int at(size\_t);  void remove(size\_t);  size\_t get\_size();  void print\_to\_console();  void clear();  void set(size\_t, int);  bool isEmpty();  void reverse();  private:  Node\* head;  Node\* last;  size\_t size;  }; |
| List.cpp |
| #pragma once  #include "List.h"  #include <iostream>  BiList::BiList()  {  head = NULL;  last = NULL;  size = 0;  }  BiList::~BiList()  {  while (head)  {  last = head->next;  delete head;  head = last;  }  }  //The function inserts a new node at back of the list.  void BiList::push\_back(int value)  {  Node\* temp = new Node;  temp->next = NULL;  temp->data = value;  if (!this->isEmpty())  {  temp->prev = last;  last->next = temp;  last = temp;  }  else {  temp->prev = NULL;  head = last = temp;  }  size++;  }  //The function inserts a new node at front of the list.  void BiList::push\_front(int value)  {  Node\* temp = new Node;  temp->prev = NULL;  temp->data = value;  if (!this->isEmpty())  {  temp->next = head;  head->prev = temp;  head = temp;  }  else  {  temp->prev = NULL;  temp->next = NULL;  head = last = temp;  }  size++;  }  //The function deletes the last node in the list.  void BiList::pop\_back()  {  if(size==0)  return;  if (size == 1)  {  delete last;  last = head = NULL;  size--;  return;  }  delete last->next;  last = last->prev;  last->next = NULL;  size--;  }  //The function deletes the first node in the list.  void BiList::pop\_front()  {  if (size == 0)  return;  if (size == 1)  {  delete head;  head = last = NULL;  size--;  return;  }  delete head->prev;  head->prev = NULL;  head = head->next;  size--;  }  //The function inserts a new node after the node with the number "index".  void BiList::insert(int value, size\_t index)  {  if ((size == 0 && index==0) || index >= size)  throw "Incorrect index.";  if (index == 0)  {  this->push\_front(value);  return;  }  Node\* cursor = head;  for (size\_t i = 0; i < index; i++)  cursor = cursor->next;  Node\* temp = new Node;  temp->data = value;  temp->next = cursor;  temp->prev = cursor->prev;  cursor->prev = temp;  temp->prev->next = temp;  size++;  }  //The function gets the value from the node with the number "index".  int BiList::at(size\_t index)  {  if (index >= size || index<0)  throw "Incorrect index.";  Node\* cursor = head;  for (size\_t i = 0; i < index; i++)  cursor = cursor->next;  return cursor->data;  }  //The function deletes the node with the "index" number.  void BiList::remove(size\_t index)  {  if (index >= size || index <= 0)  throw "Incorrect index.";  if (index == 0)  {  this->pop\_front();  return;  }  if (index == size-1)  {  this->pop\_back();  return;  }  Node\* cursor = head;  for (size\_t i = 0; i < index; i++)  cursor = cursor->next;  Node\* temp = cursor;  cursor->prev->next = cursor->next;  cursor->next->prev = cursor->prev;  delete temp;  size--;  }  //The function gets the number of items in the list  size\_t BiList::get\_size()  {  size\_t size\_of\_list = 0;  Node\* cursor = head;  if (cursor == NULL)  return 0;  while (cursor)  {  size\_of\_list++;  cursor = cursor->next;  }  return size\_of\_list;  }  //The function prints all list in console.  void BiList::print\_to\_console()  {  Node\* cursor = head;  if (head)  {  while (cursor->next)  {  std::cout << cursor->data << "<-->";  cursor = cursor->next;  }  std::cout << cursor->data;  }  else throw "List is empty.";  }  //The function deletes all node in list.  void BiList::clear()  {  while (head!=NULL)  this->pop\_front();  head = last = NULL;  }  //The function sets a new value in the node with the number "index"  void BiList::set(size\_t index, int value)  {  if (index >= size || index<0)  throw"Incorrect index.";  Node\* cursor = head;  for (size\_t i = 0; i < index; i++)  cursor = cursor->next;  cursor->data = value;  }  //The function checks the list for emptiness.  bool BiList::isEmpty()  {  if (head==NULL) return true;  return false;  }  //The function reverses a list.  void BiList::reverse()  {  if (!head || !head->next)  return;  last = head;  Node\* temp = NULL;  Node\* current = head;  while (current != NULL)  {  temp = current->prev;  current->prev = current->next;  current->next = temp;  current = current->prev;  }  head = temp->prev;  } |
| UnitTest.cpp |
| #include "pch.h"  #include "CppUnitTest.h"  #include "../Lab1/List.h"  #include "../Lab1/List.cpp"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace UnitTest  {  TEST\_CLASS(BiListTest)  {  public:  BiList Empty;  BiList NotEmpty;  TEST\_METHOD\_INITIALIZE(SetUp)  {  for (size\_t i = 0; i < 3; i++)  NotEmpty.push\_back(i + 1);  }  TEST\_METHOD(IsEmpty\_for\_NotEmpty)  {  Assert::AreEqual(NotEmpty.isEmpty(), false);  }  TEST\_METHOD(IsEmpty\_for\_Empty)  {  Assert::AreEqual(Empty.isEmpty(), true);  }  TEST\_METHOD(Get\_Size\_for\_Empty)  {  size\_t real\_size = 0;  Assert::AreEqual(Empty.get\_size(), real\_size);  }  TEST\_METHOD(Get\_Size\_for\_NotEmpty)  {  size\_t real\_size = 3;  Assert::AreEqual(NotEmpty.get\_size(), real\_size);  }  TEST\_METHOD(at\_correct\_index)  {  Assert::AreEqual(NotEmpty.at(2), 3);  }  TEST\_METHOD(at\_incorrect\_index)  {  try  {  Empty.at(4);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Incorrect index.");  }  }  TEST\_METHOD(Push\_Back\_NotEmpty)  {  NotEmpty.push\_back(5);  Assert::AreEqual(NotEmpty.at(3), 5);  }  TEST\_METHOD(Push\_Front\_NotEmpty)  {  NotEmpty.push\_front(0);  Assert::AreEqual(NotEmpty.at(0), 0);  }  TEST\_METHOD(remove\_correct\_index)  {  int temp = NotEmpty.at(2);  NotEmpty.remove(1);  Assert::AreEqual(NotEmpty.at(1), temp);  }  TEST\_METHOD(Pop\_Back\_for\_NotEmpty)  {  size\_t real\_size = NotEmpty.get\_size() - 1;  NotEmpty.pop\_back();  Assert::AreEqual(NotEmpty.get\_size(), real\_size);  }  TEST\_METHOD(Pop\_Back\_for\_Empty)  {  Empty.pop\_back();  Assert::AreEqual(Empty.isEmpty(), true);  }  TEST\_METHOD(Pop\_Front\_for\_NotEmpty)  {  size\_t real\_size = NotEmpty.get\_size() - 1;  NotEmpty.pop\_front();  Assert::AreEqual(NotEmpty.get\_size(), real\_size);  }  TEST\_METHOD(insert\_correct\_index)  {  NotEmpty.insert(4, 1);  Assert::AreEqual(NotEmpty.at(1), 4);  }  TEST\_METHOD(insert\_incorrect\_index)  {  try  {  Empty.insert(2, 0);  }  catch (const char\* error)  {  Assert::AreEqual("Incorrect index.", error);  }  }  TEST\_METHOD(remove\_incorrect\_index)  {  try  {  Empty.remove(2);  }  catch (const char\* error)  {  Assert::AreEqual("Incorrect index.", error);  }  }  TEST\_METHOD(reverse\_for\_NotEmpty)  {  int last\_inf = NotEmpty.at(NotEmpty.get\_size()-1);  NotEmpty.reverse();  Assert::AreEqual(NotEmpty.at(0), last\_inf);  }  TEST\_METHOD(Pop\_Front\_for\_Empty)  {  Empty.pop\_front();  Assert::AreEqual(Empty.isEmpty(), true);  }  TEST\_METHOD(set\_correct\_index)  {  NotEmpty.set(2, 3);  Assert::AreEqual(NotEmpty.at(2), 3);  }  TEST\_METHOD(set\_incorrect\_index)  {  try  {  NotEmpty.set(10000000, 2);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Incorrect index.");  }  }  TEST\_METHOD(Test\_Clear\_NotEmpty)  {  NotEmpty.clear();  size\_t real\_size = 0;  Assert::AreEqual(NotEmpty.get\_size(), real\_size);  }  TEST\_METHOD(Test\_Clear\_Empty)  {  Empty.clear();  size\_t real\_size = 0;  Assert::AreEqual(Empty.get\_size(), real\_size);  }  TEST\_METHOD(Push\_Back\_Empty)  {  Empty.push\_back(5);  Assert::AreEqual(Empty.at(0), 5);  Empty.pop\_back();  }  TEST\_METHOD(Push\_Front\_Empty)  {  Empty.push\_front(0);  Assert::AreEqual(Empty.at(0), 0);  }  };  } |

## Результат выполнения всех unit-тестов



## Вывод

В ходе выполнения работы я научился реализовывать двунаправленные связные списки, проводить оценку временной сложности методов, а также создавать и использовать unit-тесты.