**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

ОТЧЁТ

**по лабораторной работе №3**

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

**Тема: «Алгоритмы на графах»**

**Вариант 2.**

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

Санкт-Петербург

2021

## Постановка задачи и описание реализуемого класса и методов.

**Задание:**

Дан список возможных авиарейсов в текстовом файле в формате:

Город отправления 1;Город прибытия 1;цена прямого перелета 1;цена обратного перелета 1

Город отправления 2;Город прибытия 2;цена перелета 2;цена обратного перелета 1

…

Город отправления N;Город прибытия N;цена перелета N;цена обратного перелета N

 В случае, если нет прямого или обратного рейса, его цена будет указана как N/A (not available)

**Найти:** наиболее эффективный по стоимости перелет из города i в город j.

## Описание используемого алгоритма и структур данных.

Для этого мне понадобился класс «List» (взят из предыдущих лабораторных работ и переделан под нужную реализацию). Также используются функции вне класса. Алгоритм реализован так: сначала мы создаем список, в котором храним все уникальные (неповторяющиеся) города из текстового файла. Затем создаем матрицу смежности n\*n, где n- количество уникальных городов. Изначально заполняем эту матрицу нулями, затем вносим данные из текстового файла. Если пути из города А в город B не существует, то в матрице остается 0. Затем создаем матрицу кратчайших расстояний, которую изначально заполняем бесконечностями. Затем с помощью алгоритма Беллмана-Форда вносим кратчайшие расстояния. Пользователя мы просим ввести город отправки и город прибытия. Затем из полученный матрицы мы выводим наиболее эффективный по стоимости путь из города А в город B.

template<class data\_T>

class List

|  |  |  |
| --- | --- | --- |
| Поля | Описание | |
| Node \*head; | Указатель на начало списка | |
| Node \*tail; | Указатель на конец списка | |
| size\_t size; | Переменная, хранящая размер списка | |
| Методы | Описание | Оценка временной сложности |
| void push\_back(data\_T) | Добавление в конец списка | O(1) |
| void push\_front(data\_T) | Добавление в начало списка | O(1) |
| void pop\_back() | Удаление последнего элемента | O(1) |
| void pop\_front() | Удаление первого элемента | O(1) |
| void insert(size\_t, data\_T) | Добавление элемента по индексу | O(n) |
| data\_T 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) |
| bool isEmpty() | Проверка на пустоту списка | O(1) |
| void clear() | Удаление всех элементов списка | O(n) |
| void reverse | Изменение порядка элементов в списке на обратный | O(n) |
| void set(size\_t, data\_T) | Замена элемента по индексу | O(n) |
| data\_T top() | Получение первого элемента в списке | O(1) |
| int search(data\_T info) | Поиск индекса элемента по информации | O(n) |

class Node

|  |  |
| --- | --- |
| Поля | Описание |
| Node \*next; | Указатель на следующий элемент списка |
| Node \*prev; | Указатель на предыдущий элемент списка |
| data\_T data; | Информация, хранящаяся в элементах списка |

Функции:

|  |  |  |
| --- | --- | --- |
| Функция | Описание | O |
| void Search\_Unique\_Cities(List<string>\* Cities, ifstream& read); | Поиск уникальных (неповторяющихся) городов | O(n) |
| void Search\_Matrix(List<string>\* Cities, ifstream& read, int\*\* Adjacency\_Matrix); | Поиск матрицы смежности | O(n) |
| void Bellman\_Ford(int\*\* Adjacency\_Matrix, int\*\* Shortest\_Distances, int unique\_cities, int index\_from\_city); | Алгоритм Беллмана-Форда | O(n^3) |
| void Process(ifstream& read, int& result); | Процесс работы программы | O(n^3) |

## Описание реализованных unit-тестов.

|  |  |
| --- | --- |
| Проверка методов для класса «List» (взято из предыдущей лабораторной работы) | |
| Проверка функций: | |
| Название теста | Что проверяет |
| Search\_Unique\_Cities\_Test | Проверка поиска уникальных городов |
| Search\_Matrix\_Test | Проверка заполнения матрицы смежности |
| Belman\_Ford\_Test1 | Проверка алгоритма Беллмана-Форда при условии, что перелет из города А в город B сущестует |
| Belman\_Ford\_Test2 | Проверка алгоритма Беллмана-Форда при условии, что перелет из города А в город B не сущестует |

## Результат выполнения всех unit-тестов

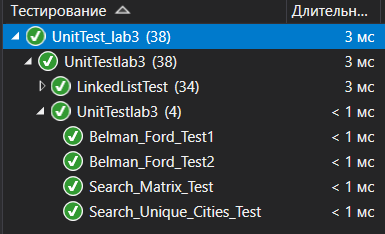
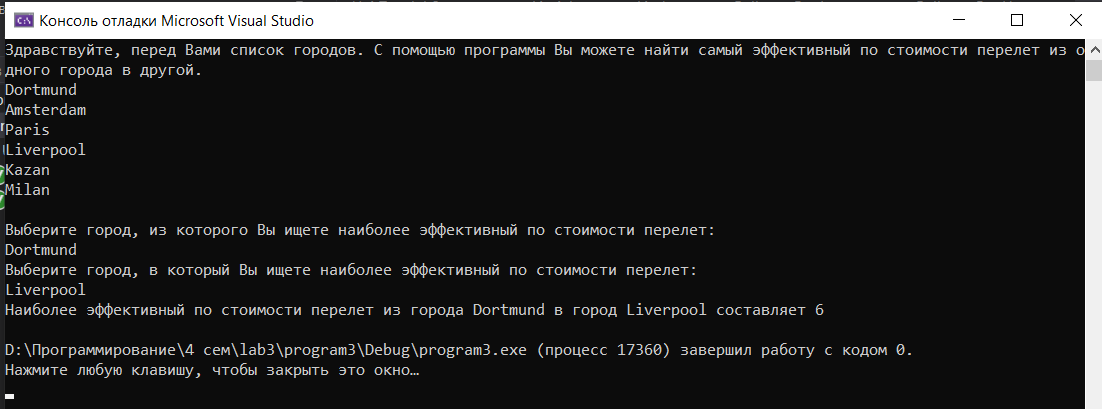
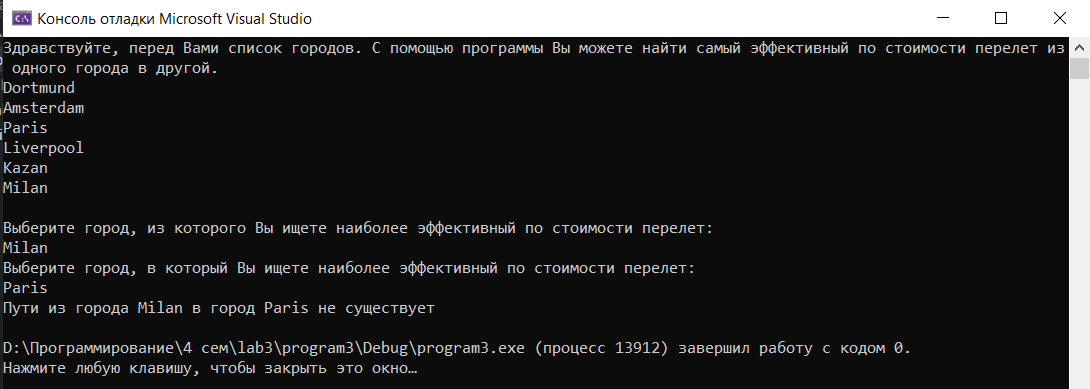


Рисунок 1. Unit-тесты

## Примеры работы программы





## Листинг

|  |
| --- |
| List.h |
| #pragma once  #include <iostream>  #include <string>  using namespace std;  template<class data\_T>  class List  {  public:  List();  ~List();  void push\_back(data\_T data);  void push\_front(data\_T data);  void pop\_back();  void pop\_front();  void clear();  size\_t GetSize();  data\_T at(const int index);  bool isEmpty();  void remove(size\_t index);  void set(size\_t index, data\_T data);  void insert(size\_t index, data\_T data);  void reverse();  void print\_to\_console();  data\_T top();  int search(data\_T info);  private:  class Node  {  public:  Node\* next;  Node\* prev;  data\_T data;  Node(data\_T data = "\0", Node\* next = nullptr, Node\* prev = nullptr)  {  this->data = data;  this->next = next;  this->prev = prev;  }  };  Node\* head;  Node\* tail;  size\_t size;  };  #include "List.inl" |
| List.inl |
| #pragma once  #include "List.h"  // constructor  template<class data\_T>  List<data\_T>::List()  {  size = 0;  head = nullptr;  tail = nullptr;  }  // destructor  template<class data\_T>  List<data\_T>::~List()  {  clear();  }  // inserting an element at the end of the list  template<class data\_T>  void List<data\_T>::push\_back(data\_T 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  template<class data\_T>  void List<data\_T>::push\_front(data\_T 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  template<class data\_T>  void List<data\_T>::pop\_back()  {  if (isEmpty())  throw "Error! Linked list is Empty.";  else  {  Node\* todelete = tail;  tail = tail->prev;  delete todelete;  size--;  }  }  // remove the first element  template<class data\_T>  void List<data\_T>::pop\_front()  {  if (head != nullptr) {  Node\* temp = this->head;  head = head->next;  delete temp;  size--;  }  else throw "Error! Linked list is Empty.";  }  // clear list  template<class data\_T>  void List<data\_T>::clear()  {  while (size) {  pop\_front();  }  }  // getting list size  template<class data\_T>  size\_t List<data\_T>::GetSize()  {  return size;  }  // output the list to the console  template<class data\_T>  void List<data\_T>::print\_to\_console()  {  Node\* cursor = head;  if (head)  {  while (cursor->next)  {  std::cout << cursor->data << " ";  cursor = cursor->next;  }  std::cout << cursor->data;  std::cout << std::endl;  }  else throw "Error! List is empty.";  }  // getting an element by index  template<class data\_T>  data\_T List<data\_T>::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++;  }  }  // checking the list for emptiness  template<class data\_T>  bool List<data\_T>::isEmpty()  {  if (head == nullptr)  return true;  else return false;  }  // replacing the element by index with the passed element  template<class data\_T>  void List<data\_T>::set(size\_t index, data\_T data)  {  if (index >= size || index < 0)  throw "Error! Incorrect input.";  Node\* cursor = head;  for (size\_t i = 0; i < index; i++)  cursor = cursor->next;  cursor->data = data;  }  // insert into an arbitrary place in the list by index  template<class data\_T>  void List<data\_T>::insert(size\_t index, data\_T 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++;  }  }  }  // кeverse the order of items in the list  template<class data\_T>  void List<data\_T>::reverse()  {  if (head == nullptr)  throw "Error! Linked list is empty.";  if (!head || !head->next)  return;  tail = head;  Node\* temp = nullptr;  Node\* current = head;  while (current != nullptr)  {  temp = current->prev;  current->prev = current->next;  current->next = temp;  current = current->prev;  }  head = temp->prev;  }  // deleting an element by index  template<class data\_T>  void List<data\_T>::remove(size\_t index)  {  if (index >= size || index < 0)  throw "Error! Incorrect input.";  if (index == 0)  this->pop\_front();  else if (index == size - 1)  this->pop\_back();  else {  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--;  }  }  // getting the first item in the list  template<class data\_T>  data\_T List<data\_T>::top()  {  if (head)  return head->data;  else throw "Error! Stack is empty.";  }  template<class data\_T>  inline int List<data\_T>::search(data\_T info)  {  if (size != 0) {  Node\* current = head;  for (size\_t i = 0; i < size; i++) {  if (info == current->data) {  return i;  }  current = current->next;  }  }  return -1;  } |
| Bellman-Ford.h |
| #pragma once  #include "list.h"  void Search\_Unique\_Cities(List<string>\* Cities, ifstream& read);  void Search\_Matrix(List<string>\* Cities, ifstream& read, int\*\* Adjacency\_Matrix);  void Bellman\_Ford(int\*\* Adjacency\_Matrix, int\*\* Shortest\_Distances, int unique\_cities, int index\_from\_city);  void Process(ifstream& read, int& result); |
| Bellman-Ford.cpp |
| #include <iostream>  #include <fstream>  #include <string>  #include "list.h"  #include "Bellman-Ford.h"  using namespace std;  #define inf INT\_MAX/2-1  // search for unique(non - recurring) cities from the list  void Search\_Unique\_Cities(List<string>\* Cities, ifstream& read)  {  string str, temp;  int counter, index;  while (!read.eof())  {  getline(read, str);  counter = 0;  for (size\_t i = 0; i < str.size(); i++)  {  temp = "";  while (str[i] != ';' && i != str.size())  temp += str[i++];  index = Cities->search(temp);  if (index == -1)  Cities->push\_back(temp);  counter++;  if (counter == 2)  break;  }  }  }  // finding the adjacency matrix  void Search\_Matrix(List<string>\* Cities, ifstream& read, int\*\* Adjacency\_Matrix)  {  string str, temp;  int indexI, indexJ, counter;  while (!read.eof())  {  getline(read, str);  counter = 0;  for (size\_t i = 0; i < str.size(); i++)  {  temp = "";  while (str[i] != ';' && i != str.size())  temp += str[i++];  if (counter == 0)  indexI = Cities->search(temp);  if (counter == 1)  indexJ = Cities->search(temp);  if (counter == 2 && temp != "N/A")  Adjacency\_Matrix[indexI][indexJ] = atoi(temp.c\_str());  if (counter == 3 && temp != "N/A")  Adjacency\_Matrix[indexJ][indexI] = atoi(temp.c\_str());  counter++;  }  }  }  // Bellman–Ford algorithm  void Bellman\_Ford(int\*\* Adjacency\_Matrix, int\*\* Shortest\_Distances, int unique\_cities, int index\_from\_city)  {  for (int i = 0; i < unique\_cities; i++) {  Shortest\_Distances[i][index\_from\_city] = 0;  }  for (int k = 1; k < unique\_cities; k++) {  for (int i = 0; i < unique\_cities; i++) {  for (int j = 0; j < unique\_cities; j++) {  if (Shortest\_Distances[k - 1][j] + Adjacency\_Matrix[j][i] < Shortest\_Distances[k][i] && Adjacency\_Matrix[j][i] != 0) {  Shortest\_Distances[k][i] = Shortest\_Distances[k - 1][j] + Adjacency\_Matrix[j][i];  }  }  }  }  }  // performs the work of the program  void Process(ifstream& read, int& result)  {  List<string>\* Cities = new List<string>;  Search\_Unique\_Cities(Cities, read);  read.clear();  read.seekg(0);  int unique\_cities = Cities->GetSize();  int\*\* Adjacency\_Matrix = new int\* [unique\_cities];  for (int i = 0; i < unique\_cities; i++)  Adjacency\_Matrix[i] = new int[unique\_cities];  for (int i = 0; i < unique\_cities; i++)  for (int j = 0; j < unique\_cities; j++)  Adjacency\_Matrix[i][j] = 0;  Search\_Matrix(Cities, read, Adjacency\_Matrix);  int\*\* Shortest\_Distances = new int\* [unique\_cities];  for (int i = 0; i < unique\_cities; i++)  Shortest\_Distances[i] = new int[unique\_cities];  for (int i = 0; i < unique\_cities; i++)  for (int j = 0; j < unique\_cities; j++)  Shortest\_Distances[i][j] = inf;  cout << "Здравствуйте, перед Вами список городов. С помощью программы Вы можете найти самый эффективный по стоимости перелет из одного города в другой." << endl;  Cities->print\_to\_console();  int from, control = 0;  string choice;  do  {  if (control == 0)  cout << "Выберите город, из которого Вы ищете наиболее эффективный по стоимости перелет: " << endl;  else  {  cout << "\nОшибка! Вы некорректно ввели название города, попробуйте еще раз." << endl;  cout << "Выберите город, из которого Вы ищете наиболее эффективный по стоимости перелет: " << endl;  }  cin >> choice;  from = Cities->search(choice);  control++;  } while (from == -1 && control < 3);  if (from == -1)  {  cout << "\nВы так и не смогли правильно ввести название нужного города, попробуйте в другой раз." << endl;  return;  }  Bellman\_Ford(Adjacency\_Matrix, Shortest\_Distances, unique\_cities, from);  int to;  control = 0;  choice = "";  do  {  if (control == 0)  cout << "Выберите город, в который Вы ищете наиболее эффективный по стоимости перелет: " << endl;  else  {  cout << "\nОшибка! Вы некорректно ввели название города, попробуйте еще раз." << endl;  cout << "Выберите город, в который Вы ищете наиболее эффективный по стоимости перелет: " << endl;  }  cin >> choice;  to = Cities->search(choice);  control++;  } while (to == -1 && control < 3);  if (to == -1)  {  cout << "\nВы так и не смогли правильно ввести название нужного города, попробуйте в другой раз." << endl;  return;  }  result = Shortest\_Distances[unique\_cities - 1][to];  if (result == inf)  cout << "Пути из города " << Cities->at(from) << " в город " << Cities->at(to) << " не существует" << endl;  else  cout << "Наиболее эффективный по стоимости перелет из города " << Cities->at(from) << " в город " << Cities->at(to) << " составляет " << result << endl;  for (unsigned j = 0; j < unique\_cities; ++j)  delete[] Adjacency\_Matrix[j];  delete[] Adjacency\_Matrix;  for (unsigned j = 0; j < unique\_cities; ++j)  delete[] Shortest\_Distances[j];  delete[] Shortest\_Distances;  } |
| UnitTest\_lab3.cpp |
| #include "pch.h"  #include "CppUnitTest.h"  #include <fstream>  #include "../program3/List.h"  #include "../program3/Bellman-Ford.h"  #include "../program3/Bellman-Ford.cpp"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace UnitTestlab3  {  TEST\_CLASS(BellmanFordTest)  {  public:    TEST\_METHOD(Search\_Unique\_Cities\_Test)  {  ifstream read;  read.open("D:\\Программирование\\4 сем\\lab3\\program3\\test\_1.txt", ios::in);  List<string>\* Cities = new List<string>;  Search\_Unique\_Cities(Cities, read);  Assert::IsTrue("Yerevan" == Cities->at(0));  Assert::IsTrue("Ufa" == Cities->at(1));  Assert::IsTrue("Moscow" == Cities->at(2));  }  TEST\_METHOD(Search\_Matrix\_Test)  {  ifstream read;  read.open("D:\\Программирование\\4 сем\\lab3\\program3\\test\_1.txt", ios::in);  List<string>\* Cities = new List<string>;  Search\_Unique\_Cities(Cities, read);  read.clear();  read.seekg(0);  int unique\_cities = Cities->GetSize();  int\*\* Adjacency\_Matrix = new int\* [unique\_cities];  for (int i = 0; i < unique\_cities; i++)  {  Adjacency\_Matrix[i] = new int[unique\_cities];  }  for (int i = 0; i < unique\_cities; i++)  {  for (int j = 0; j < unique\_cities; j++) {  Adjacency\_Matrix[i][j] = 0;  }  }  Search\_Matrix(Cities, read, Adjacency\_Matrix);  Assert::IsTrue(Adjacency\_Matrix[0][0] == 0);  Assert::IsTrue(Adjacency\_Matrix[0][1] == 2);  Assert::IsTrue(Adjacency\_Matrix[0][2] == 5);  Assert::IsTrue(Adjacency\_Matrix[1][0] == 10);  Assert::IsTrue(Adjacency\_Matrix[1][1] == 0);  Assert::IsTrue(Adjacency\_Matrix[1][2] == 2);  Assert::IsTrue(Adjacency\_Matrix[2][0] == 9);  Assert::IsTrue(Adjacency\_Matrix[2][1] == 4);  Assert::IsTrue(Adjacency\_Matrix[2][2] == 0);  for (unsigned j = 0; j < unique\_cities; ++j)  delete[] Adjacency\_Matrix[j];  delete[] Adjacency\_Matrix;  }  TEST\_METHOD(Belman\_Ford\_Test1)  {  ifstream read;  int result = 0;  read.open("D:\\Программирование\\4 сем\\lab3\\program3\\test\_1.txt", ios::in);  List<string>\* Cities = new List<string>;  Search\_Unique\_Cities(Cities, read);  read.clear();  read.seekg(0);  int unique\_cities = Cities->GetSize();  int\*\* Adjacency\_Matrix = new int\* [unique\_cities];  for (int i = 0; i < unique\_cities; i++)  {  Adjacency\_Matrix[i] = new int[unique\_cities];  }  for (int i = 0; i < unique\_cities; i++)  {  for (int j = 0; j < unique\_cities; j++)  {  Adjacency\_Matrix[i][j] = 0;  }  }  Search\_Matrix(Cities, read, Adjacency\_Matrix);  int\*\* Shortest\_Distances = new int\* [unique\_cities];  for (int i = 0; i < unique\_cities; i++)  {  Shortest\_Distances[i] = new int[unique\_cities];  }  for (int i = 0; i < unique\_cities; i++)  {  for (int j = 0; j < unique\_cities; j++) {  Shortest\_Distances[i][j] = inf;  }  }  string from\_choice = "Yerevan";  int from = Cities->search(from\_choice);  Bellman\_Ford(Adjacency\_Matrix, Shortest\_Distances, unique\_cities, from);  string to\_choice = "Moscow";  int to = Cities->search(to\_choice);  result = Shortest\_Distances[unique\_cities - 1][to];  // check: from Yerevan to Moscow  Assert::IsTrue(result == 4);  for (unsigned j = 0; j < unique\_cities; ++j)  delete[] Adjacency\_Matrix[j];  delete[] Adjacency\_Matrix;  for (unsigned j = 0; j < unique\_cities; ++j)  delete[] Shortest\_Distances[j];  delete[] Shortest\_Distances;    }  TEST\_METHOD(Belman\_Ford\_Test2)  {  ifstream read;  int result = 0;  read.open("D:\\Программирование\\4 сем\\lab3\\program3\\test\_2.txt", ios::in);  List<string>\* Cities = new List<string>;  Search\_Unique\_Cities(Cities, read);  read.clear();  read.seekg(0);  int unique\_cities = Cities->GetSize();  int\*\* Adjacency\_Matrix = new int\* [unique\_cities];  for (int i = 0; i < unique\_cities; i++)  {  Adjacency\_Matrix[i] = new int[unique\_cities];  }  for (int i = 0; i < unique\_cities; i++)  {  for (int j = 0; j < unique\_cities; j++)  {  Adjacency\_Matrix[i][j] = 0;  }  }  Search\_Matrix(Cities, read, Adjacency\_Matrix);  int\*\* Shortest\_Distances = new int\* [unique\_cities];  for (int i = 0; i < unique\_cities; i++)  {  Shortest\_Distances[i] = new int[unique\_cities];  }  for (int i = 0; i < unique\_cities; i++)  {  for (int j = 0; j < unique\_cities; j++) {  Shortest\_Distances[i][j] = inf;  }  }    string from\_choice = "Yerevan";  int from = Cities->search(from\_choice);  Bellman\_Ford(Adjacency\_Matrix, Shortest\_Distances, unique\_cities, from);  string to\_choice = "Ufa";  int to = Cities->search(to\_choice);    result = Shortest\_Distances[unique\_cities - 1][to];    // check: from Yerevan to Ufa  Assert::IsTrue(result == inf);  for (unsigned j = 0; j < unique\_cities; ++j)  delete[] Adjacency\_Matrix[j];  delete[] Adjacency\_Matrix;  for (unsigned j = 0; j < unique\_cities; ++j)  delete[] Shortest\_Distances[j];  delete[] Shortest\_Distances;  }  };  TEST\_CLASS(LinkedListTest)  {  public:  TEST\_METHOD(ConstructorTest)  {  List<int> lst;  Assert::IsTrue(lst.GetSize() == 0);  }  TEST\_METHOD(IsEmptyTest1)  {  List<int> lst;  Assert::IsTrue(lst.isEmpty());  }  TEST\_METHOD(IsEmptyTest2)  {  List<int> lst;  lst.push\_back(3);  Assert::IsTrue(lst.isEmpty() == false);  }  TEST\_METHOD(AtTest1)  {  List<int> lst;  lst.push\_back(3);  Assert::IsTrue(lst.at(0) == 3);  }  TEST\_METHOD(AtTest2)  {  List<int> 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<int> lst;  lst.push\_back(3);  try  {  lst.at(-5);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Incorrect input.");  }  }  TEST\_METHOD(GetSizeTest1)  {  List<int> lst;  Assert::IsTrue(lst.GetSize() == 0);  }  TEST\_METHOD(GetSizeTest2)  {  List<int> lst;  lst.push\_back(3);  lst.push\_front(2);  Assert::IsTrue(lst.GetSize() == 2);  }  TEST\_METHOD(GetSizeTest3)  {  List<int> lst;  lst.push\_back(3);  lst.push\_front(2);  lst.pop\_back();  lst.pop\_front();  Assert::IsTrue(lst.GetSize() == 0);  }  TEST\_METHOD(PushBackTest1)  {  List<int> 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<int> 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<int> 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<int> 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<int> 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<int> lst;  lst.push\_back(0);  lst.pop\_back();  Assert::IsTrue(lst.GetSize() == 0);  }  TEST\_METHOD(PopBackTest3)  {  List<int> lst;  try  {  lst.pop\_back();  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Linked list is Empty.");  }  }  TEST\_METHOD(PopFrontTest1)  {  List<int> 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<int> lst;  lst.push\_back(0);  lst.pop\_front();  Assert::IsTrue(lst.isEmpty(), L"Assert 1");  }  TEST\_METHOD(PopFrontTest3)  {  List<int> lst;  try  {  lst.pop\_front();  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Linked list is Empty.");  }  }  TEST\_METHOD(InsertTest1)  {  List<int> 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<int> 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<int> 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<int> 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<int> 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<int> 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<int> 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<int> 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<int> 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<int> 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(RemoveTest1)  {  List<int> 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<int> 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<int> 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<int> 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<int> 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 <fstream>  #include "Bellman-Ford.h"  using namespace std;  #define inf INT\_MAX/2-1  int main()  {  setlocale(LC\_ALL, "rus");  ifstream read;  int result = 0;  read.open("air\_flights.txt", ios::in);  if (read.bad())  {  cout << "Ошибка! Файл не открылся.";  return 0;  }  Process(read,result);  return 0;  } |