**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

ОТЧЁТ

**по курсовой работе**

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

**Тема: «Потоки в сетях»**

**Вариант 2.**

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

Санкт-Петербург

2021

## Постановка задачи.

**Задание:**

Входные данные: текстовый файл со строками в формате V1, V1, P, где V1, V2 направленная дуга транспортной сети, а P – ее пропускная способность. Исток всегда обозначен как S, сток – как T

**Найти:** максимальный поток в сети используя алгоритм Эдмондса-Карпа.

## Описание используемого алгоритма и структур данных.

Для этого мне понадобился классы «Graph» и «List» (взят из предыдущих лабораторных работ и переделан под нужную реализацию). Также используются функции вне класса. Алгоритм реализован так: сначала мы считываем из файла все неповторяющиеся символы в строку. Эти символы и есть вершины графа. Затем мы создаем экземпляр класса «Graph», в который вносим пропускные способности, которые также считываем из файла. Затем применяется алгоритм Эдмондса-Карпа, который находит максимальный поток в транспортной сети. Результат выводится на экран.

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 |

class Graph

|  |  |  |
| --- | --- | --- |
| Поля | Описание | |
| int dimension; | Размерность графа | |
| int\*\* matrix; | Двумерный массив (для хранения пропускной способности) | |
| Методы | Описание | Оценка временной сложности |
| void SetValue(int, int, int); | Установление значения по индексу | O(1) |
| int GetValue(int, int ); | Получение значения по индексу | O(1) |
| List<int>\* BFS(Graph& ); | Поиск в ширину | O(n^2) |

Функции:

|  |  |  |
| --- | --- | --- |
| Функция | Описание | O |
| void SearchUniqueVertexes(string&, int&, ifstream&) | Поиск уникальных (неповторяющихся) вершин | O(n^2) |
| void SearchThroughput(int, Graph\*, char\*, ifstream&) | Поиск пропускных способностей и внесение их в двумерный массив | O(n^2) |
| void BubbleSort(int, char\*) | Сортировка массива | O(n^2) |
| unsigned EdmondsKarp(int, Graph&) | Алгоритм Эдмондса-Карпа | O(v^3) |
| void Process(ifstream&, int&); | Процесс работы программы | O(v^3) |
| v-количество вершин графа | | |

## Описание реализованных unit-тестов.

|  |  |
| --- | --- |
| Проверка методов для класса «List» (взято из предыдущей лабораторной работы) | |
| Проверка функций: | |
| Название теста | Что проверяет |
| SearchUniqueVertexesTest | Проверка поиска уникальных вершин |
| EdmondsKarpTest\_1 | Проверка работы программы на текстовом файле «input.txt» |
| EdmondsKarpTest\_2 | Проверка работы программы на текстовом файле «test.txt» |
| Проверка методов для класса «Graph»: | |
| GetValueTest | Проверка метода GetValue |
| SetValueTest | Проверка метода SetValue |
| BFSTest | Проверка метода BFS |

## Результат выполнения всех unit-тестов

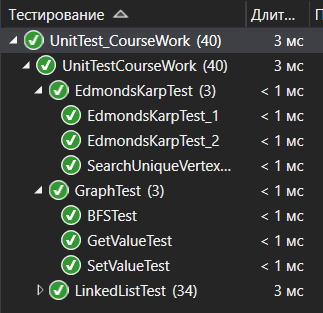
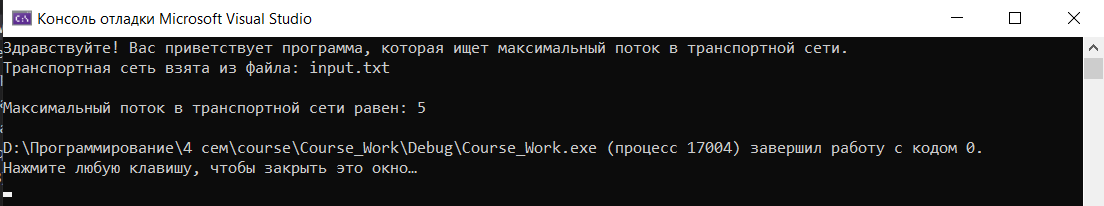
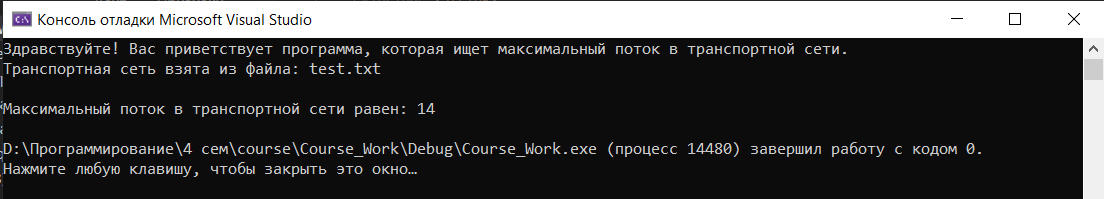


Рисунок 1. Unit-тесты

## 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;  } |
| Edmonds-Karp.h |
| #pragma once  #include "Graph.h"  void SearchUniqueVertexes(string& UniqueVertexes, int& VertexesNumber, ifstream& read);  void SearchThroughput(int VertexesNumber, Graph\* Throughput, char\* Vertexes, ifstream& read);  void BubbleSort(int VertexesNumber, char\* Vertexes);  unsigned EdmondsKarp(int VertexesNumber, Graph& Throughput);  void Process(unsigned& MaxFlow, ifstream& read); |
| Edmonds-Karp.cpp |
| #include <iostream>  #include <fstream>  #include <string>  #include "Edmonds-Karp.h"  // Search for unique vertices  void SearchUniqueVertexes(string& UniqueVertexes, int& VertexesNumber, ifstream& read)  {  char temp = ' ';  while (!read.eof())  {  temp = read.get();  if (temp == ' ' || isdigit(temp) || temp == '\n')  continue;  if (!isdigit(temp))  {  for (int i = 0; i < VertexesNumber; ++i)  {  if (UniqueVertexes[i] == temp)  {  temp = ' ';  break;  }  }  if (temp == ' ')  continue;  else  {  UniqueVertexes += temp;  VertexesNumber++;  }  }  }  }  // Search for bandwidth and add it to a two-dimensional array  void SearchThroughput(int VertexesNumber, Graph\* Throughput, char\* Vertexes, ifstream& read)  {  char temp = ' ';  unsigned from = 0, to = 0;  string temp\_number = "";  while (!read.eof())  {  temp = read.get();  if (temp == '\n' || temp == ' ')  continue;  else if (!isdigit(temp))  {  for (unsigned i = 0; i < VertexesNumber; ++i)  {  if (Vertexes[i] == temp)  from = i;  }  temp = read.get();  temp = read.get();  for (unsigned i = 0; i < VertexesNumber; ++i)  {  if (Vertexes[i] == temp)  to = i;  }  }  else if (isdigit(temp))  {  temp\_number.push\_back(temp);  while (isdigit(temp = read.get()))  temp\_number.push\_back(temp);  Throughput->SetValue(from, to, atoi(temp\_number.c\_str()));  //std::cout << "i: " << from << ", j: " << to << ", number: " << temp\_number << endl;  temp\_number.clear();  if (read.eof())  break;  }  }  }  // Sorting an array  void BubbleSort(int VertexesNumber, char\* Vertexes)  {  for (int i = 0; i < VertexesNumber - 1; ++i)  {  for (int j = 0; j < VertexesNumber - 1; ++j)  {  if (Vertexes[j] > Vertexes[j + 1] && (Vertexes[j] != 'S' && Vertexes[j + 1] != 'T'))  {  char temp = Vertexes[j];  Vertexes[j] = Vertexes[j + 1];  Vertexes[j + 1] = temp;  }  }  }  }  // The Edmonds-Karp algorithm  unsigned EdmondsKarp(int VertexesNumber, Graph& Throughput)  {  Graph\* Flow = new Graph(VertexesNumber);  List<int>\* ShortestPath = new List<int>;  ShortestPath = Throughput.BFS(\*Flow);  int\* ResidualThroughput = new int[ShortestPath->GetSize() - 1];  while (!(ShortestPath->isEmpty()) || ShortestPath->GetSize() == 1)  {  ShortestPath = Throughput.BFS(\*Flow);  ResidualThroughput = new int[ShortestPath->GetSize() - 1];  if (ShortestPath->isEmpty() || ShortestPath->GetSize() == 1)  break;  for (int i = 0; i < ShortestPath->GetSize() - 1; ++i)  {  unsigned to = Throughput.GetValue(ShortestPath->at(i), ShortestPath->at(i + 1));  unsigned from = Throughput.GetValue(ShortestPath->at(i + 1), ShortestPath->at(i));  if (to != 0)  ResidualThroughput[i] = to - Flow->GetValue(ShortestPath->at(i), ShortestPath->at(i + 1));  else if (from != 0)  ResidualThroughput[i] = Flow->GetValue(ShortestPath->at(i + 1), ShortestPath->at(i));  else  ResidualThroughput = 0;  }  unsigned min = ResidualThroughput[0];  for (int i = 1; i < ShortestPath->GetSize() - 1; ++i)  if (ResidualThroughput[i] < min)  min = ResidualThroughput[i];  for (int i = 0; i < ShortestPath->GetSize() - 1; ++i)  {  unsigned to = Flow->GetValue(ShortestPath->at(i), ShortestPath->at(i + 1));  unsigned from = Flow->GetValue(ShortestPath->at(i + 1), ShortestPath->at(i));  if (from == 0)  Flow->SetValue(ShortestPath->at(i), ShortestPath->at(i + 1), to + min);  else  Flow->SetValue(ShortestPath->at(i + 1), ShortestPath->at(i), from - min);  }  }  unsigned MaxFlow = 0;  for (int i = 0; i < VertexesNumber; ++i)  {  if (Flow->GetValue(0, i) != 0)  MaxFlow += Flow->GetValue(0, i);  }  delete[] ResidualThroughput;  delete ShortestPath;  delete Flow;  return MaxFlow;  }  // How the program works  void Process(unsigned& MaxFlow, ifstream& read)  {  int VertexesNumber = 0;  string UniqueVertexes = "";  SearchUniqueVertexes(UniqueVertexes, VertexesNumber, read);  VertexesNumber = VertexesNumber - 1;  char\* Vertexes = new char[VertexesNumber];  for (int k = 0; k < VertexesNumber; ++k)  Vertexes[k] = UniqueVertexes[k];  BubbleSort(VertexesNumber, Vertexes);  read.clear();  read.seekg(0, ios::beg);  Graph\* Throughput = new Graph(VertexesNumber);  SearchThroughput(VertexesNumber, Throughput, Vertexes, read);  MaxFlow = EdmondsKarp(VertexesNumber, \*Throughput);  delete Throughput;  delete[] Vertexes;  } |
| Graph.h |
| #pragma once  #include "List.h"  class Graph  {  private:  int dimension;  int\*\* matrix;  public:  Graph(int V);  ~Graph();  void SetValue(int V1, int V2, int data);  int GetValue(int V1, int V2);  List<int>\* BFS(Graph& Flow);  }; |
| Graph.cpp |
| #include "Graph.h"  // Constructor  Graph::Graph(int dimension)  {  this->dimension = dimension;  matrix = new int\* [dimension];  for (int i = 0; i < dimension; ++i)  matrix[i] = new int[dimension];  for (int k = 0; k < dimension; ++k)  for (int l = 0; l < dimension; ++l)  matrix[k][l] = 0;  }  // Destructor  Graph::~Graph()  {  for (int i = 0; i < this->dimension; ++i)  delete[] matrix[i];  delete[] matrix;  }  // Setting the value by index  void Graph::SetValue(int V1, int V2, int data)  {  matrix[V1][V2] = data;  }  // Getting a value by index  int Graph::GetValue(int V1, int V2)  {  return matrix[V1][V2];  }  // Breadth-first search  List<int>\* Graph::BFS(Graph& Flow)  {  List<int> queue;  List<int>\* path = new List<int>;  int\* level = new int[dimension];  bool\* visited = new bool[dimension];  std::fill(visited, visited + dimension, false);  level[0] = 0;  queue.push\_back(0);  visited[0] = true;  while (!(queue.isEmpty()))  {  int u = queue.at(0);  queue.pop\_front();  for (int i = 0; i < dimension; i++)  {  if (u == i)  continue;  int V = matrix[u][i];  int V0 = Flow.GetValue(u, i);  int V1 = Flow.GetValue(i, u);  int V2 = matrix[i][u];  if (!visited[i] && ((V != V0) || ((V2 != V1) && V1 != 0)))  {  queue.push\_back(i);  visited[i] = true;  level[i] = level[u] + 1;  }  }  }  int Max = level[dimension - 1];  path->push\_front(dimension - 1);  while (Max != 0 && Max >= 0)  {  for (int j = 0; j < dimension; ++j)  {  int V = matrix[j][path->at(0)];  int V0 = Flow.GetValue(j, path->at(0));  int V1 = Flow.GetValue(path->at(0), j);  int V2 = matrix[path->at(0)][j];  if (level[j] == Max - 1 && ((V != V0) || ((V2 != V1) && V1 != 0)))  {  if (level[j] == Max - 1)  {  path->push\_front(j);  Max--;  break;  }  }  }  }  delete[] level;  delete[] visited;  return path;  } |
| UnitTest\_CourseWork.cpp |
| #include "pch.h"  #include "CppUnitTest.h"  #include <fstream>  #include "..\Course\_Work\Edmonds-Karp.cpp"  #include "..\Course\_Work\Graph.cpp"  #include "..\Course\_Work\List.h"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace UnitTestCourseWork  {  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.");  }  }  };  TEST\_CLASS(GraphTest)  {  public:  TEST\_METHOD(GetValueTest)  {  int dim = 3;  Graph Test(dim);  Assert::IsTrue(Test.GetValue(0, 1) == 0);  }  TEST\_METHOD(SetValueTest)  {  int dim = 3;  Graph Test(dim);  Test.SetValue(2, 2, 5);  Assert::IsTrue(Test.GetValue(2, 2) == 5);  }  TEST\_METHOD(BFSTest)  {  int dim = 3;  Graph\* Test = new Graph(dim);  Graph\* Flow = new Graph(3);  Test->SetValue(0, 1, 1);  Test->SetValue(0, 2, 1);  Test->SetValue(1, 2, 1);  Assert::IsTrue(Test->BFS(\*Flow)->at(0) == 0);  Assert::IsTrue(Test->BFS(\*Flow)->at(1) == 2);  }  };  TEST\_CLASS(EdmondsKarpTest)  {  public:  TEST\_METHOD(SearchUniqueVertexesTest)  {  ifstream read("D:\\Программирование\\4 сем\\сourse\\Course\_Work\\input.txt");  if (!read.is\_open())  return;  int VertexesNumber = 0;  string UniqueVertexes = "";  SearchUniqueVertexes(UniqueVertexes, VertexesNumber, read);  VertexesNumber = VertexesNumber - 1;  char\* Vertexes = new char[VertexesNumber];  for (int k = 0; k < VertexesNumber; ++k)  Vertexes[k] = UniqueVertexes[k];  read.close();  Assert::IsTrue(Vertexes[0] == 'S');  Assert::IsTrue(Vertexes[1] == 'O');  Assert::IsTrue(Vertexes[2] == 'P');  Assert::IsTrue(Vertexes[3] == 'Q');  Assert::IsTrue(Vertexes[4] == 'R');  Assert::IsTrue(Vertexes[5] == 'T');  Assert::IsTrue(VertexesNumber == 6);  delete[] Vertexes;  }  TEST\_METHOD(EdmondsKarpTest\_1)  {  ifstream read("D:\\Программирование\\4 сем\\сourse\\Course\_Work\\input.txt");  if (!read.is\_open())  return;  unsigned MaxFlow = 0;  Process(MaxFlow, read);  read.close();  Assert::IsTrue(MaxFlow == 5);  }  TEST\_METHOD(EdmondsKarpTest\_2)  {  ifstream read("D:\\Программирование\\4 сем\\сourse\\Course\_Work\\test.txt");  if (!read.is\_open())  return;  unsigned MaxFlow = 0;  Process(MaxFlow, read);  read.close();  Assert::IsTrue(MaxFlow == 14);  }  };  } |
| Main.cpp |
| #include <iostream>  #include <fstream>  #include <string>  #include "Graph.h"  #include "List.h"  #include "Edmonds-Karp.h"  using namespace std;  int main()  {  setlocale(LC\_ALL, "rus");  ifstream read;  unsigned MaxFlow = 0;  string file\_name = "test.txt";  read.open(file\_name, ios::in);  if (read.bad())  {  cout << "Ошибка! Файл не открылся.";  return 0;  }  cout << "Здравствуйте! Вас приветствует программа, которая ищет максимальный поток в транспортной сети." << endl;  cout << "Транспортная сеть взята из файла: " << file\_name << endl << endl;  Process(MaxFlow, read);  cout << "Максимальный поток в транспортной сети равен: " << MaxFlow << endl;  read.close();  return 0;  } |