**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра САПР**

**ОТЧЕТ**

**По курсовой работе**

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

**Вариант №3**

|  |  |  |
| --- | --- | --- |
| Студент гр. 8302 | **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** | Никулин Л.А. |
| Преподаватель | **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** | Тутуева А.В. |

1. **Цель работы**

Реализовать программу определяющую максимальный поток в заданном графе. Программа считывает из файла список ребер и их пропускные способности.

**2. Описание реализуемого класса и методов**

|  |  |
| --- | --- |
| FlowPushRelabel | Содержит поля:  int\* excessFlowArray (массив избытков вершин), int\*\* capacity (остаточная сеть),  int\* height (функция высоты),  int vertexCount (количество вершин),  int sourceVertex (исток),  int destinationVertex (сток).  Содержит следующие методы:  Дефолтный конструктор.  Конструктор с параметром ifstream&– вызывает метод setInfo(ifstream&);  Деструктор – вызывает метод clear(). |
| void push(int edge, int vertex) | Функция, проталкивающая поток из u в v, равный min{e[edge], cf(edge, vertex)}, и подсчитывающая остаточную сеть и избытки |
| void lift(int edge) | Поднимает вершину на минимальную высоту, достаточную для возможности проталкивания потока |
| void discharge(int edge) | Выполняет лифтинг и проталкивание, пока это возможно |
| int maximalFlow() | Вычисляет максимальный поток в сети |
| void clear() | Очищение на основе обычного удаления двоичного дерева |
| void setInfo(ifstream&) | Получает на вход файл с списком строк, обрабатывает их и выдает список смежности |

**3. Оценка временной сложности алгоритмов**

|  |  |
| --- | --- |
| void push(int edge, int vertex) | O(1) |
| void lift(int edge) | O(|V|) |
| void discharge(int edge) | O(|V| |E|) |
| int maximalFlow() | O(|V|E|) |

**4. Описание реализованных unit-тестов**

|  |  |
| --- | --- |
| Test\_Сorrect\_output | Проверяет ситуацию c 20 вершинами |
| Test\_Exception\_entering\_empty\_character | Некорректное введение символа |
| Test\_Exception\_entering\_the\_bandwidth | Некорректное введение пропускной способности |
| Test\_Exception\_empty\_string | Ввод пустой строки |
| Test\_Exception\_there\_is\_a\_path\_from\_the\_vertex\_to\_itself | Некорректный путь от вершины к самой себе |

**5. Обоснование выбора используемых структур данных**

Был выбран МАР для того чтобы индивидуализировать вершины индексами. Данную структура используется потому, что она позволяет не сохранять повторяющиеся данные и быстрый доступ к ним. List используется для перебора вершин сети в функции maximalFlow.

Со структурой List удобнее работать, нежели с обычным массивом, так как не нужно хранить его размер, а также быстро добавлять и удалять элементы, без траты времени на их перезапись.

**6. Примеры работы программы**

|  |  |  |
| --- | --- | --- |
| № | Входные данные: | Результат: |
| 1 |  |  |
| 2 |  |  |
| 3 |  |  |
| 4 |  |  |
| 5 |  |  |
| 6 |  |  |
| 7 |  |  |

**7. Листинг**

**CourseWork.cpp:**

#include <iostream>

#include <fstream>

#include "Flow.h"

int main()

{

ifstream input("input.txt");

FlowPushRelabel example(input);

std::cout << "Максимальный поток: " << example.maximalFlow();

}

**Flow.h:**

#pragma once

#include <fstream>

#include "List.h"

#include <string>

#include"Map.h"

using namespace std;

class FlowPushRelabel {

private:

#pragma region VARIABLES

int\* excessFlowArray;

int\*\* capacity;

int\* height;

int vertexCount, sourceVertex, destinationVertex;

#pragma endregion

#pragma region FUNCTIONS

int min(int, int);

#pragma endregion

public:

FlowPushRelabel() = default;

FlowPushRelabel(ifstream&);

~FlowPushRelabel();

int maximalFlow();

void setInfo(ifstream&);

void push(int, int);

void clear();

void lift(int);

void discharge(int);

};

**Flow.cpp:**

#pragma once

#include "Flow.h"

FlowPushRelabel::~FlowPushRelabel() {

clear();

}

FlowPushRelabel::FlowPushRelabel(ifstream& file)

{

setInfo(file);

}

int FlowPushRelabel::maximalFlow() {

if (vertexCount > 2) {

for (int i = 0; i < vertexCount; i++)

{

if (i == sourceVertex)

continue;

excessFlowArray[i] = capacity[sourceVertex][i];

capacity[i][sourceVertex] += capacity[sourceVertex][i];

}

height[sourceVertex] = vertexCount;

List<int> l;

int cur;

int cur\_index = 0;

int old\_height;

for (int i = 0; i < vertexCount; i++)

if (i != sourceVertex && i != destinationVertex)

l.push\_front(i);

cur = l.at(0);

while (cur\_index < l.get\_size())

{

old\_height = height[cur];

discharge(cur);

if (height[cur] != old\_height)

{

l.push\_front(cur);

l.remove(++cur\_index);

cur = l.at(0);

cur\_index = 0;

}

++cur\_index;

if (cur\_index < l.get\_size())

cur = l.at(cur\_index);

}

return excessFlowArray[destinationVertex];

}

else

return capacity[0][1];

}

void FlowPushRelabel::setInfo(ifstream& file)

{

Map<char, int>\* cardCharNumber = new Map<char, int>();

vertexCount = 0;

int str\_num = 1;

while (!file.eof()) {

string s1;

getline(file, s1);

if (s1.size() >= 5) {

if (!((s1[0] >= 'A' && s1[0] <= 'Z') && (s1[1] == ' ')) || !((s1[2] >= 'A' && s1[2] <= 'Z') && (s1[3] == ' '))) {

throw std::exception("Error entering a character in the string or missing a space after it.");

}

string cur;

for (int i = 4; i < s1.size(); ++i) {

if (s1[i] >= '0' && s1[i] <= '9')

cur += s1[i];

else {

throw std::exception("Error entering the third character (bandwidth) in the string or the presence of a space after it.");

}

}

if (!cardCharNumber->find\_is(s1[0])) {

cardCharNumber->insert(s1[0], vertexCount);

++vertexCount;

}

if (!cardCharNumber->find\_is(s1[2])) {

cardCharNumber->insert(s1[2], vertexCount);

++vertexCount;

}

}

else

{

throw std::exception(string(("A data-entry error. Check the correctness of the input in the file and correct these errors in the line under the number: " + to\_string(str\_num))).c\_str());

}

++str\_num;

}

if (cardCharNumber->find\_is('S'))

sourceVertex = cardCharNumber->find('S');

else {

throw std::exception("Source is missing");

}

if (cardCharNumber->find\_is('T'))

destinationVertex = cardCharNumber->find('T');

else {

throw std::exception("Sink is missing");

}

file.clear();

file.seekg(ios::beg);

excessFlowArray = new int[vertexCount];

height = new int[vertexCount];

capacity = new int\* [vertexCount];

for (int i = 0; i < vertexCount; ++i) {

excessFlowArray[i] = 0;

height[i] = 0;

}

for (int i = 0; i < vertexCount; ++i) {

capacity[i] = new int[vertexCount];

for (int j = 0; j < vertexCount; ++j)

capacity[i][j] = 0;

}

str\_num = 1;

while (!file.eof()) {

string s1;

int vert1, vert2, cap;

getline(file, s1);

vert1 = cardCharNumber->find(s1[0]);

vert2 = cardCharNumber->find(s1[2]);

if (vert1 == vert2)

throw std::exception(string("The path from the vertex to itself is impossible in the string under the number: " + to\_string(str\_num)).c\_str());

capacity[vert1][vert2] = stoi(s1.substr(4));

++str\_num;

}

}

void FlowPushRelabel::push(int edge, int vertex)

{

int interVariable = min(excessFlowArray[edge], capacity[edge][vertex]);

excessFlowArray[edge] -= interVariable;

excessFlowArray[vertex] += interVariable;

capacity[edge][vertex] -= interVariable;

capacity[vertex][edge] += interVariable;

}

void FlowPushRelabel::lift(int edge)

{

int min = 2 \* vertexCount + 1;

for (int i = 0; i < vertexCount; i++)

if (capacity[edge][i] && (height[i] < min))

min = height[i];

height[edge] = min + 1;

}

void FlowPushRelabel::clear()

{

delete[] excessFlowArray;

delete[] height;

for (int i = 0; i < vertexCount; ++i)

{

delete[] capacity[i];

}

}

void FlowPushRelabel::discharge(int edge)

{

int vertex = 0;

while (excessFlowArray[edge] > 0)

{

if (capacity[edge][vertex] && height[edge] == height[vertex] + 1)

{

push(edge, vertex);

vertex = 0;

continue;

}

++vertex;

if (vertex == vertexCount)

{

lift(edge);

vertex = 0;

}

}

}

int FlowPushRelabel::min(int data1, int data2) {

return data1 > data2 ? data2 : data1;

}

**Map.h:**

#pragma once

#include"List.h"

using namespace std;

enum Color

{

RED, BLACK

};

template<typename TypeKey, typename TypeValue>

class Map {

public:

class Node

{

public:

Node(bool color = RED, TypeKey key = TypeKey(), Node\* parent = NULL, Node\* left = NULL, Node\* right = NULL, TypeValue value = TypeValue()) :color(color), key(key), parent(parent), left(left), right(right), value(value) {}

TypeKey key;

TypeValue value;

bool color;

Node\* parent;

Node\* left;

Node\* right;

};

~Map()

{

if (this->Root != NULL)

this->clear();

Root = NULL;

delete TNULL;

TNULL = NULL;

}

Map(Node\* Root = NULL, Node\* TNULL = new Node(0)) :Root(TNULL), TNULL(TNULL) {}

void printTree()

{

if (Root)

{

print\_helper(this->Root, "", true);

}

else throw std::out\_of\_range("Tree is empty!");

}

void insert(TypeKey key, TypeValue value)

{

if (this->Root != TNULL)

{

Node\* node = NULL;

Node\* parent = NULL;

/\* Search leaf for new element \*/

for (node = this->Root; node != TNULL; )

{

parent = node;

if (key < node->key)

node = node->left;

else if (key > node->key)

node = node->right;

else if (key == node->key)

throw std::out\_of\_range("key is repeated");

}

node = new Node(RED, key, TNULL, TNULL, TNULL, value);

node->parent = parent;

if (parent != TNULL)

{

if (key < parent->key)

parent->left = node;

else

parent->right = node;

}

insert\_fix(node);

}

else

{

this->Root = new Node(BLACK, key, TNULL, TNULL, TNULL, value);

}

}

List<TypeKey>\* get\_keys() {

List<TypeKey>\* list = new List<TypeKey>();

this->ListKey(Root, list);

return list;

}

List<TypeValue>\* get\_values() {

List<TypeValue>\* list = new List<TypeValue>();

this->ListValue(Root, list);

return list;

}

TypeValue find(TypeKey key)

{

Node\* node = Root;

while (node != TNULL && node->key != key)

{

if (node->key > key)

node = node->left;

else

if (node->key < key)

node = node->right;

}

if (node != TNULL)

return node->value;

else

throw std::out\_of\_range("Key is missing");

}

void remove(TypeKey key)

{

this->delete\_node(this->find\_key(key));

}

void clear()

{

this->clear\_tree(this->Root);

this->Root = NULL;

}

bool find\_is(TypeKey key) {

Node\* node = Root;

while (node != TNULL && node->key != key) {

if (node->key > key)

node = node->left;

else

if (node->key < key)

node = node->right;

}

if (node != TNULL)

return true;

else

return false;

}

void increment\_value(TypeKey key) {

Node\* cur = this->find\_value(key);

cur->value++;

}

private:

Node\* Root;

Node\* TNULL;

//delete functions

void delete\_node(Node\* find\_node)

{

Node\* node\_with\_fix, \* cur\_for\_change;

cur\_for\_change = find\_node;

bool cur\_for\_change\_original\_color = cur\_for\_change->color;

if (find\_node->left == TNULL)

{

node\_with\_fix = find\_node->right;

transplant(find\_node, find\_node->right);

}

else if (find\_node->right == TNULL)

{

node\_with\_fix = find\_node->left;

transplant(find\_node, find\_node->left);

}

else

{

cur\_for\_change = minimum(find\_node->right);

cur\_for\_change\_original\_color = cur\_for\_change->color;

node\_with\_fix = cur\_for\_change->right;

if (cur\_for\_change->parent == find\_node)

{

node\_with\_fix->parent = cur\_for\_change;

}

else

{

transplant(cur\_for\_change, cur\_for\_change->right);

cur\_for\_change->right = find\_node->right;

cur\_for\_change->right->parent = cur\_for\_change;

}

transplant(find\_node, cur\_for\_change);

cur\_for\_change->left = find\_node->left;

cur\_for\_change->left->parent = cur\_for\_change;

cur\_for\_change->color = find\_node->color;

}

delete find\_node;

if (cur\_for\_change\_original\_color == RED)

{

this->delete\_fix(node\_with\_fix);

}

}

//swap links(parent and other) for rotate

void transplant(Node\* current, Node\* current1)

{

if (current->parent == TNULL)

{

Root = current1;

}

else if (current == current->parent->left)

{

current->parent->left = current1;

}

else

{

current->parent->right = current1;

}

current1->parent = current->parent;

}

void clear\_tree(Node\* tree)

{

if (tree != TNULL)

{

clear\_tree(tree->left);

clear\_tree(tree->right);

delete tree;

}

}

//find functions

Node\* minimum(Node\* node)

{

while (node->left != TNULL)

{

node = node->left;

}

return node;

}

Node\* maximum(Node\* node)

{

while (node->right != TNULL)

{

node = node->right;

}

return node;

}

Node\* grandparent(Node\* current)

{

if ((current != TNULL) && (current->parent != TNULL))

return current->parent->parent;

else

return TNULL;

}

Node\* uncle(Node\* current)

{

Node\* current1 = grandparent(current);

if (current1 == TNULL)

return TNULL; // No grandparent means no uncle

if (current->parent == current1->left)

return current1->right;

else

return current1->left;

}

Node\* sibling(Node\* n)

{

if (n == n->parent->left)

return n->parent->right;

else

return n->parent->left;

}

Node\* find\_key(TypeKey key)

{

Node\* node = this->Root;

while (node != TNULL && node->key != key)

{

if (node->key > key)

node = node->left;

else

if (node->key < key)

node = node->right;

}

if (node != TNULL)

return node;

else

throw std::out\_of\_range("Key is missing");

}

//all print function

void print\_helper(Node\* root, string indent, bool last)

{

if (root != TNULL)

{

cout << indent;

if (last)

{

cout << "R----";

indent += " ";

}

else

{

cout << "L----";

indent += "| ";

}

string sColor = !root->color ? "black" : "red";

cout << root->key << " (" << sColor << ")" << endl;

print\_helper(root->left, indent, false);

print\_helper(root->right, indent, true);

}

}

void list\_key\_or\_value(int mode, List<TypeKey>\* list)

{

if (this->Root != TNULL)

this->key\_or\_value(Root, list, mode);

else

throw std::out\_of\_range("Tree empty!");

}

void key\_or\_value(Node\* tree, List<TypeKey>\* list, int mode)

{

if (tree != TNULL)

{

key\_or\_value(tree->left, list, mode);

if (mode == 1)

list->push\_back(tree->key);

else

list->push\_back(tree->value);

key\_or\_value(tree->right, list, mode);

}

}

//fix

void insert\_fix(Node\* node)

{

Node\* uncle;

/\* Current node is RED \*/

while (node != this->Root && node->parent->color == RED)//

{

/\* node in left tree of grandfather \*/

if (node->parent == this->grandparent(node)->left)//

{

/\* node in left tree of grandfather \*/

uncle = this->uncle(node);

if (uncle->color == RED)

{

/\* Case 1 - uncle is RED \*/

node->parent->color = BLACK;

uncle->color = BLACK;

this->grandparent(node)->color = RED;

node = this->grandparent(node);

}

else {

/\* Cases 2 & 3 - uncle is BLACK \*/

if (node == node->parent->right)

{

/\*Reduce case 2 to case 3 \*/

node = node->parent;

this->left\_rotate(node);

}

/\* Case 3 \*/

node->parent->color = BLACK;

this->grandparent(node)->color = RED;

this->right\_rotate(this->grandparent(node));

}

}

else {

/\* Node in right tree of grandfather \*/

uncle = this->uncle(node);

if (uncle->color == RED)

{

/\* Uncle is RED \*/

node->parent->color = BLACK;

uncle->color = BLACK;

this->grandparent(node)->color = RED;

node = this->grandparent(node);

}

else {

/\* Uncle is BLACK \*/

if (node == node->parent->left)

{

node = node->parent;

this->right\_rotate(node);

}

node->parent->color = BLACK;

this->grandparent(node)->color = RED;

this->left\_rotate(this->grandparent(node));

}

}

}

this->Root->color = BLACK;

}

void delete\_fix(Node\* node)

{

Node\* sibling;

while (node != this->Root && node->color == BLACK)//

{

sibling = this->sibling(node);

if (sibling != TNULL)

{

if (node == node->parent->left)//

{

if (sibling->color == BLACK)

{

node->parent->color = BLACK;

sibling->color = RED;

this->left\_rotate(node->parent);

sibling = this->sibling(node);

}

if (sibling->left->color == RED && sibling->right->color == RED)

{

sibling->color = BLACK;

node = node->parent;

}

else

{

if (sibling->right->color == RED)

{

sibling->left->color = RED;

sibling->color = BLACK;

this->left\_rotate(sibling);

sibling = this->sibling(node);

}

sibling->color = node->parent->color;

node->parent->color = RED;

sibling->right->color = RED;

this->left\_rotate(node->parent);

node = this->Root;

}

}

else

{

if (sibling->color == BLACK);

{

sibling->color = RED;

node->parent->color = BLACK;

this->right\_rotate(node->parent);

sibling = this->sibling(node);

}

if (sibling->left->color == RED && sibling->right->color)

{

sibling->color = BLACK;

node = node->parent;

}

else

{

if (sibling->left->color == RED)

{

sibling->right->color = RED;

sibling->color = BLACK;

this->left\_rotate(sibling);

sibling = this->sibling(node);

}

sibling->color = node->parent->color;

node->parent->color = RED;

sibling->left->color = RED;

this->right\_rotate(node->parent);

node = Root;

}

}

}

}

this->Root->color = BLACK;

}

//Rotates

void left\_rotate(Node\* node)

{

Node\* right = node->right;

/\* Create node->right link \*/

node->right = right->left;

if (right->left != TNULL)

right->left->parent = node;

/\* Create right->parent link \*/

if (right != TNULL)

right->parent = node->parent;

if (node->parent != TNULL)

{

if (node == node->parent->left)

node->parent->left = right;

else

node->parent->right = right;

}

else {

this->Root = right;

}

right->left = node;

if (node != TNULL)

node->parent = right;

}

void right\_rotate(Node\* node)

{

Node\* left = node->left;

/\* Create node->left link \*/

node->left = left->right;

if (left->right != TNULL)

left->right->parent = node;

/\* Create left->parent link \*/

if (left != TNULL)

left->parent = node->parent;

if (node->parent != TNULL)

{

if (node == node->parent->right)

node->parent->right = left;

else

node->parent->left = left;

}

else

{

this->Root = left;

}

left->right = node;

if (node != TNULL)

node->parent = left;

}

void ListValue(Node\* tree, List<TypeValue>\* list) {

if (tree != TNULL) {

ListValue(tree->left, list);

list->push\_back(tree->value);

ListValue(tree->right, list);

}

}

void ListKey(Node\* tree, List<TypeKey>\* list) {

if (tree != TNULL) {

ListKey(tree->left, list);

list->push\_back(tree->key);

ListKey(tree->right, list);

}

}

Node\* find\_value(TypeKey key) {

Node\* node = Root;

while (node != TNULL && node->key != key) {

if (node->key > key)

node = node->left;

else

if (node->key < key)

node = node->right;

}

if (node != TNULL)

return node;

}

};

**List.h:**

#pragma once

#include<iostream>

using namespace std;

template<class TypeKey>

class List

{

private:

class Node {

public:

Node(TypeKey data = TypeKey(), Node\* Next = NULL) {

this->data = data;

this->Next = Next;

}

Node\* Next;

TypeKey data;

};

public:

void push\_back(TypeKey obj) { // добавление в конец списка

if (head != NULL) {

this->tail->Next = new Node(obj);

tail = tail->Next;

}

else {

this->head = new Node(obj);

this->tail = this->head;

}

Size++;

}

void push\_front(TypeKey obj) { // добавление в начало списка

if (head != NULL) {

Node\* current = new Node;

current->data = obj;

current->Next = this->head;

this->head = current;

}

else {

this->head = new Node(obj);

tail = head;

}

this->Size++;

}

void pop\_back() { // удаление последнего элемента

if (head != NULL) {

Node\* current = head;

while (current->Next != tail)//то есть ищем предпоследний

current = current->Next;

delete tail;

tail = current;

tail->Next = NULL;

Size--;

}

else throw std::out\_of\_range("out\_of\_range");

}

void pop\_front() { // удаление первого элемента

if (head != NULL) {

Node\* current = head;

head = head->Next;

delete current;

Size--;

}

else throw std::out\_of\_range("out\_of\_range");

}

void insert(TypeKey obj, size\_t k) {// добавление элемента по индексу (вставка перед элементом, который был ранее доступен по этому индексу)

if (k >= 0 && this->Size > k) {

if (this->head != NULL) {

if (k == 0)

this->push\_front(obj);

else

if (k == this->Size - 1)

this->push\_back(obj);

else

{

Node\* current = new Node;//для добавления элемента

Node\* current1 = head;//для поиска итого элемента

for (int i = 0; i < k - 1; i++) {

current1 = current1->Next;

}

current->data = obj;

current->Next = current1->Next;//переуказывает на след элемент

current1->Next = current;

Size++;

}

}

}

else {

throw std::out\_of\_range("out\_of\_range");

}

}

TypeKey at(size\_t k) {// получение элемента по индексу

if (this->head != NULL && k >= 0 && k <= this->Size - 1) {

if (k == 0)

return this->head->data;

else

if (k == this->Size - 1)

return this->tail->data;

else

{

Node\* current = head;

for (int i = 0; i < k; i++) {

current = current->Next;

}

return current->data;

}

}

else {

throw std::out\_of\_range("out\_of\_range");

}

}

void remove(int k) { // удаление элемента по индексу

if (head != NULL && k >= 0&&k<=Size-1) {

if (k == 0) this->pop\_front();

else

if (k == this->Size - 1) this->pop\_back();

else

if (k != 0) {

Node\* current = head;

for (int i = 0; i < k - 1; i++) {//переходим на предэлемент

current = current->Next;

}

Node\* current1 = current->Next;

current->Next = current->Next->Next;

delete current1;

Size--;

}

}

else {

throw std::out\_of\_range("out\_of\_range");

}

}

size\_t get\_size() { // получение размера списка

return Size;

}

void print\_to\_console() { // вывод элементов списка в консоль через разделитель

if (this->head != NULL) {

Node\* current = head;

for (int i = 0; i < Size; i++) {

cout << current->data << ' ';

current = current->Next;

}

}

}

void clear() { // удаление всех элементов списка

if (head != NULL) {

Node\* current = head;

while (head != NULL) {

current = current->Next;

delete head;

head = current;

}

Size = 0;

}

}

void set(size\_t k, TypeKey obj) // замена элемента по индексу на передаваемый элемент

{

if (this->head != NULL && this->get\_size() >= k && k >= 0) {

Node\* current = head;

for (int i = 0; i < k; i++) {

current = current->Next;

}

current->data = obj;

}

else {

throw std::out\_of\_range("out\_of\_range");

}

}

bool isEmpty() { // проверка на пустоту списка

return (bool)(head);

}

void reverse() { // меняет порядок элементов в списке

int Counter = Size;

Node\* HeadCur = NULL;

Node\* TailCur = NULL;

for (int j = 0; j <Size; j++) {

if (HeadCur != NULL) {

if(head!=NULL&&head->Next==NULL){

TailCur->Next = head;

TailCur = head;

head = NULL;

}

else {

Node \* cur = head;

for (int i = 0; i < Counter - 2; i++)

cur = cur->Next;

TailCur->Next = cur->Next;

TailCur = cur->Next;

cur->Next = NULL;

tail = cur;

Counter--;

}

}

else {

HeadCur = tail;

TailCur = tail;

Node\* cur = head;

for (int i = 0; i < Size - 2; i++)

cur = cur->Next;

tail = cur;

tail->Next = NULL;

Counter--;

}

}

head = HeadCur;

tail = TailCur;

}

public:

List(Node\* head = NULL, Node\* tail = NULL, int Size = 0) :head(head), tail(tail), Size(Size) {}

~List() {

if (head != NULL) {

this->clear();

}

};

private:

Node\* head;

Node\* tail;

int Size;

};

**CourseWorkTests.cpp:**

#include "CppUnitTest.h"

#include "../CourseWork/Flow.h"

#include "../CourseWork/Flow.cpp"

#include <fstream>

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

#define FILE1 "C:\\Users\\ASUS\\Desktop\\CourseWork\\CourseWorkTests\\test1.txt"

#define FILE2 "C:\\Users\\ASUS\\Desktop\\CourseWork\\CourseWorkTests\\test2.txt"

#define FILE3 "C:\\Users\\ASUS\\Desktop\\CourseWork\\CourseWorkTests\\test3.txt"

#define FILE4 "C:\\Users\\ASUS\\Desktop\\CourseWork\\CourseWorkTests\\test4.txt"

#define FILE5 "C:\\Users\\ASUS\\Desktop\\CourseWork\\CourseWorkTests\\test5.txt"

#define FILE6 "C:\\Users\\ASUS\\Desktop\\CourseWork\\CourseWorkTests\\test6.txt"

namespace UnitTestCourseWork

{

TEST\_CLASS(UnitTestCourseWork)

{

public:

TEST\_METHOD(Test\_Сorrect\_output)

{

ifstream stream(FILE1);

FlowPushRelabel flow(stream);

int excepted = 19;

Assert::AreEqual(flow.maximalFlow(), excepted);

}

TEST\_METHOD(Test\_Exception\_entering\_empty\_character) {

try {

ifstream stream(FILE2);

FlowPushRelabel flow(stream);

}

catch (exception& ex) {

Assert::AreEqual(ex.what(), "Error entering a character in the string or missing a space after it.");

}

}

TEST\_METHOD(Test\_Exception\_entering\_the\_bandwidth) {

try {

ifstream stream(FILE3);

FlowPushRelabel flow(stream);

}

catch (exception& ex) {

Assert::AreEqual(ex.what(), "Error entering the third character (bandwidth) in the string or the presence of a space after it.");

}

}

TEST\_METHOD(Test\_Exception\_empty\_string) {

try {

ifstream stream(FILE4);

FlowPushRelabel flow(stream);

}

catch (exception& ex) {

Assert::AreEqual(ex.what(), "A data-entry error. Check the correctness of the input in the file and correct these errors in the line under the number: 2");

}

}

TEST\_METHOD(Test\_Exception\_there\_is\_a\_path\_from\_the\_vertex\_to\_itself) {

try {

ifstream stream(FILE6);

FlowPushRelabel flow(stream);

}

catch (exception& ex) {

Assert::AreEqual(ex.what(), "The path from the vertex to itself is impossible in the string under the number: 2");

}

}

};

}