**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. 8301 |  | Бобров А.И. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

# **Постановка задачи**

Дан текстовый файл со строками в формате V1, V2, P, где V1, V2 направленная дуга транспортной сети, а P это её пропускная способность. Исток обозначается как S, а сток как T. Найти максимальный поток в сети реализовав это с помощью алгоритма проталкивания предпотока.

# Описание реализуемых классов и методов

Flow – класс в котором содержатся основные функции реализованные для поиска максимального потока.

int MaxFlow() – метод для поиска максимального потока в сети

void push(int u, int v) – функция реализующая операцию проталкивания

void lift(int u) – метод используемый для подъёма вершины

void discharge(int u) - применяется к переполненной вершине, для того чтобы протолкнуть поток через допустимые ребра в смежные вершины.

# Оценка временной сложности

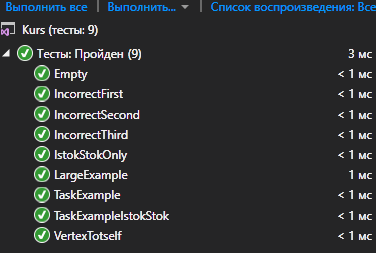
int MaxFlow() - **O(V2E)**

void push(int u, int v) - **O(1)**

void lift(int u) - **O(V)**

void discharge(int u) - **O(VE)**

# Описание реализованных Unit-тестов



IncorrectFirst – неправельный ввод V1

IncorrectSecond – неправельный ввод V2

IncorrectThird - неправельный ввод P

Empty – не введены данные

IstokStokOnly – пример только с Истоком и Стоком

LargeExample – пример с большим количеством вершин

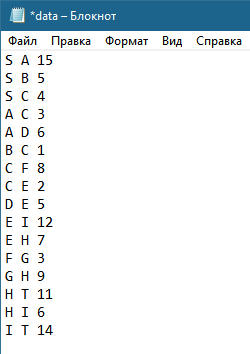
TaskExample – пример из задания

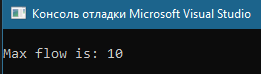
TaskExampleIstokStok – модернизированный пример из задания

VertexToItself – проверка ребра Исток Исток

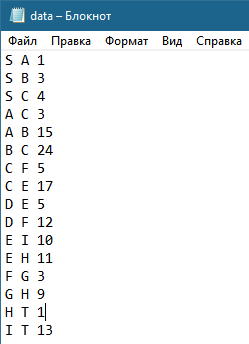
# Примеры работы программы

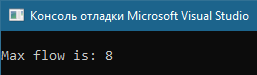
**Пример 1:**



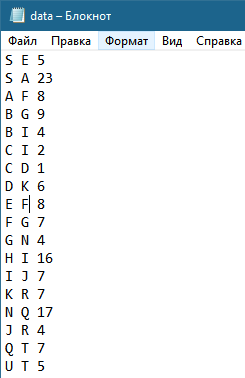


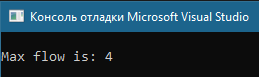
**Пример 2:**



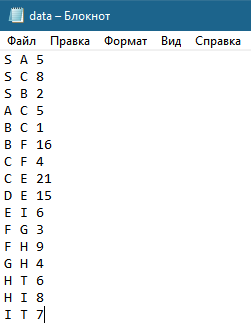


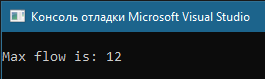
**Пример 3:**



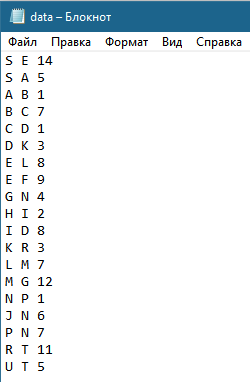


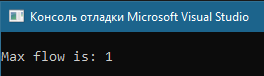
**Пример 4:**



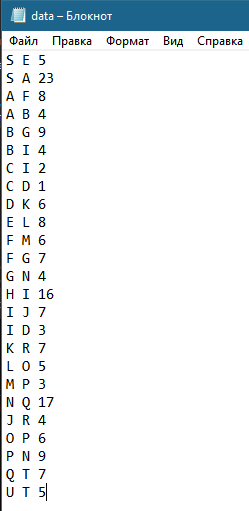


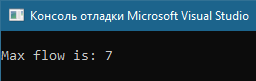
**Пример 5:**



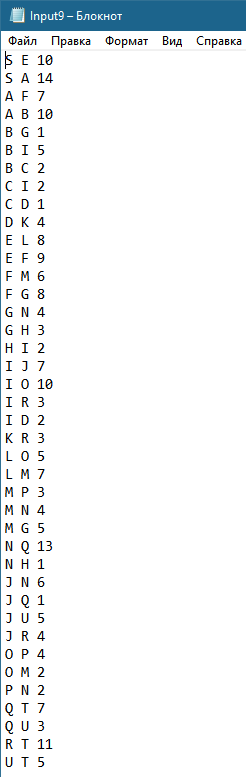


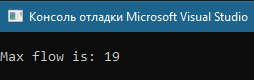
**Пример 6:**





**Пример 7:**





# Обоснование выбора используемых структур данных

List используется при переборе вершин сети в методе MaxFlow. List как структура превосходит стандартный массив. За счёт своего функционала он более удобный и с его помощью можно оптимизировать по времени работы функции, в которых он используется.

Мap используется для индексирования вершин. Как и в предыдущем случае, позволяет улучшить работу программы.

# Листинг

## Main.cpp:

#include <iostream>

#include <fstream>

#include "MaxFlow.h"

int main()

{

try

{

ifstream input("data.txt");

Flow flow(input);

std::cout << "\nMax flow is: " << flow.MaxFlow() << "\n";

}

catch (exception& exception)

{

std::cout << exception.what();

}

return 0;

## }

## 

## MaxFlow.h:

#pragma once

#include <fstream>

#include "List.h"

#include<string>

#include"Map.h"

using namespace std;

template<typename T>

T min(T a, T b)

{

return a > b ? b : a;

}

class Flow

{

public:

//the push-relabel algorithm for the maximal network flow//

int MaxFlow()

{

if (Amount > 2)

{

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

{

if (i == s)

continue;

e[i] = c[s][i]; c[i][s] += c[s][i];

}

h[s] = Amount;

List<int> l;

int current\_;

int current = 0;

int old;

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

if (i != s && i != t)

l.push\_front(i);

current\_ = l.at(0);

while (current != l.get\_size())

{

old = h[current\_];

discharge(current\_);

if (h[current\_] != old)

{

l.push\_front(current\_); l.remove(++current);

current\_ = l.at(0); current = 0;

}

current++;

if (current < l.get\_size())

current\_ = l.at(current);

}

return e[t];

}

else

return c[0][1];

}

//push//

void push(int u, int v)

{

int f = min(e[u], c[u][v]);

e[u] -= f; e[v] += f;

c[u][v] -= f; c[v][u] += f;

}

//lift//

void lift(int u)

{

int min = 2 \* Amount + 1;

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

if (c[u][i] && (h[i] < min))

min = h[i];

h[u] = min + 1;

}

//discharge//

void discharge(int u)

{

int V = 0;

while (e[u] > 0)

{

if (c[u][V] && h[u] == h[V] + 1)

{

push(u, V); V = 0; continue;

}

V++;

if (V == Amount)

{

lift(u); V = 0;

}

}

}

~Flow()

{

delete[] e;

delete[] h;

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

delete[] c[i];

}

Flow(ifstream& file)

{

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

Amount = 0;

int StringN = 1;

while (!file.eof())

{

string String;

getline(file, String);

if (String.size() >= 5) //cause five is min length of string to correct input

{

if (!((String[0] >= 'A' && String[0] <= 'Z') && (String[1] == ' ')))

{

throw std::exception(string(("There is no space after first symbol. Input is incorrect! Line: " + to\_string(StringN))).c\_str());

}

if (!((String[2] >= 'A' && String[2] <= 'Z') && (String[3] == ' ')))

{

throw std::exception(string(("There is no space after second symbol. Input is incorrect! Line: " + to\_string(StringN))).c\_str());

}

string cur;

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

{

if (String[i] >= '0' && String[i] <= '9') cur += String[i];

else

{

throw std::exception(string(("There is some trouble with third symbol. Input is incorrect! Line: " + to\_string(StringN))).c\_str());

}

}

if (!CharToNum->find\_is(String[0]))

{

CharToNum->insert(String[0], Amount);

++Amount;

}

if (!CharToNum->find\_is(String[2]))

{

CharToNum->insert(String[2], Amount);

++Amount;

}

}

else throw std::exception(string(("Data input is incorrect! Line: " + to\_string(StringN))).c\_str());

StringN++;

}

//stok//

if (CharToNum->find\_is('S')) s = CharToNum->find('S');

else throw std::exception("There is no Istok!");

//istok//

if (CharToNum->find\_is('T')) t = CharToNum->find('T');

else throw std::exception("There is no Stok!");

file.clear();

file.seekg(ios::beg);

e = new int[Amount];

h = new int[Amount];

c = new int\*[Amount];

for (int i = 0; i < Amount; ++i) {e[i] = 0; h[i] = 0;}

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

{

c[i] = new int[Amount];

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

c[i][j] = 0;

}

StringN = 1;

//itself//

while (!file.eof())

{

string s1;

int V1, V2;

getline(file, s1);

V1 = CharToNum->find(s1[0]);

V2 = CharToNum->find(s1[2]);

if (V1 == V2) throw std::exception(string("Vertex path to itself is impossible! Line: " + to\_string(StringN)).c\_str());

c[V1][V2] = stoi(s1.substr(4));

StringN++;

}

}

private:

int\* e;

int\*\* c;

int\* h;

int Amount, s, t;

};

## List.h:

#pragma once

#include<iostream>

using namespace std;

template<class T>

class List

{

private:

class Node {

public:

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

this->data = data;

this->Next = Next;

}

Node\* Next;

T data;

};

public:

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

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(T obj) { // добавление в начало списка bc

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 bc

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 bc-+

if (head != NULL) {

Node\* current = head;

head = head->Next;

delete current;

Size--;

}

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

}

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

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");

}

}

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

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 bc

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 bc

return Size;

}

void print\_to\_console() { // вывод элементов списка в консоль через разделитель, не использовать at bc

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, T 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 bc

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;

};

## Map.h:

#pragma once

#define COLOR\_RED 1

#define COLOR\_BLACK 0

#include"List.h"

using namespace std;

template<typename T, typename T1>

class Map {

public:

class Node

{

public:

Node(bool color = COLOR\_RED, T key = T(), Node\* parent = NULL, Node\* left = NULL, Node\* right = NULL, T1 value = T1()) :color(color), key(key), parent(parent), left(left), right(right), value(value) {}

T key;

T1 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(T key, T1 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(COLOR\_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(COLOR\_BLACK, key, TNULL, TNULL, TNULL, value);

}

}

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

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

this->ListKey(Root, list);

return list;

}

List<T1>\* get\_values() {

List<T1>\* list = new List<T1>();

this->ListValue(Root, list);

return list;

}

T1 find(T 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(T key)

{

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

}

void clear()

{

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

this->Root = NULL;

}

bool find\_is(T 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(T 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 == 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(T 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<T>\* 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<T>\* 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 COLOR\_RED \*/

while (node != this->Root && node->parent->color == 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 == COLOR\_RED)

{

/\* Case 1 - uncle is COLOR\_RED \*/

node->parent->color = COLOR\_BLACK;

uncle->color = COLOR\_BLACK;

this->grandparent(node)->color = COLOR\_RED;

node = this->grandparent(node);

}

else {

/\* Cases 2 & 3 - uncle is COLOR\_BLACK \*/

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

{

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

node = node->parent;

this->left\_rotate(node);

}

/\* Case 3 \*/

node->parent->color = COLOR\_BLACK;

this->grandparent(node)->color = COLOR\_RED;

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

}

}

else {

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

uncle = this->uncle(node);

if (uncle->color == COLOR\_RED)

{

/\* Uncle is COLOR\_RED \*/

node->parent->color = COLOR\_BLACK;

uncle->color = COLOR\_BLACK;

this->grandparent(node)->color = COLOR\_RED;

node = this->grandparent(node);

}

else {

/\* Uncle is COLOR\_BLACK \*/

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

{

node = node->parent;

this->right\_rotate(node);

}

node->parent->color = COLOR\_BLACK;

this->grandparent(node)->color = COLOR\_RED;

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

}

}

}

this->Root->color = COLOR\_BLACK;

}

void delete\_fix(Node\* node)

{

Node\* sibling;

while (node != this->Root && node->color == COLOR\_BLACK)//

{

sibling = this->sibling(node);

if (sibling != TNULL)

{

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

{

if (sibling->color == COLOR\_BLACK)

{

node->parent->color = COLOR\_BLACK;

sibling->color = COLOR\_RED;

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

sibling = this->sibling(node);

}

if (sibling->left->color == COLOR\_RED && sibling->right->color == COLOR\_RED)

{

sibling->color = COLOR\_BLACK;

node = node->parent;

}

else

{

if (sibling->right->color == COLOR\_RED)

{

sibling->left->color = COLOR\_RED;

sibling->color = COLOR\_BLACK;

this->left\_rotate(sibling);

sibling = this->sibling(node);

}

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

node->parent->color = COLOR\_RED;

sibling->right->color = COLOR\_RED;

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

node = this->Root;

}

}

else

{

if (sibling->color == COLOR\_BLACK);

{

sibling->color = COLOR\_RED;

node->parent->color = COLOR\_BLACK;

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

sibling = this->sibling(node);

}

if (sibling->left->color == COLOR\_RED && sibling->right->color)

{

sibling->color = COLOR\_BLACK;

node = node->parent;

}

else

{

if (sibling->left->color == COLOR\_RED)

{

sibling->right->color = COLOR\_RED;

sibling->color = COLOR\_BLACK;

this->left\_rotate(sibling);

sibling = this->sibling(node);

}

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

node->parent->color = COLOR\_RED;

sibling->left->color = COLOR\_RED;

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

node = Root;

}

}

}

}

this->Root->color = 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<T1>\* list) {

if (tree != TNULL) {

ListValue(tree->left, list);

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

ListValue(tree->right, list);

}

}

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

if (tree != TNULL) {

ListKey(tree->left, list);

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

ListKey(tree->right, list);

}

}

Node\* find\_value(T 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;

}

## };

## Unit-тесты:

#include "pch.h"

#include "CppUnitTest.h"

#include <fstream>

#include "../Kurs/MaxFlow.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace KursTest

{

TEST\_CLASS(Exceptions)

{

public:

TEST\_METHOD(IncorrectFirst)

{

try

{

ifstream input("C:\\Kurs\\Input2.txt");

Flow flow(input);

}

catch (exception & ex)

{

Assert::AreEqual(ex.what(), "There is no space after first symbol. Input is incorrect! Line: 2");

}

}

TEST\_METHOD(IncorrectSecond)

{

try

{

ifstream input("C:\\Kurs\\Input3.txt");

Flow flow(input);

}

catch (exception & ex)

{

Assert::AreEqual(ex.what(), "There is no space after second symbol. Input is incorrect! Line: 2");

}

}

TEST\_METHOD(IncorrectThird)

{

try

{

ifstream input("C:\\Kurs\\Input4.txt");

Flow flow(input);

}

catch (exception & ex)

{

Assert::AreEqual(ex.what(), "There is some trouble with third symbol. Input is incorrect! Line: 2");

}

}

TEST\_METHOD(Empty)

{

try

{

ifstream input("C:\\Kurs\\Input5.txt");

Flow flow(input);

}

catch (exception & ex)

{

Assert::AreEqual(ex.what(), "Data input is incorrect! Line: 2");

}

}

TEST\_METHOD(VertexTotself)

{

try

{

ifstream input("C:\\Kurs\\Input8.txt");

Flow flow(input);

}

catch (exception & ex)

{

Assert::AreEqual(ex.what(), "Vertex path to itself is impossible! Line: 2");

}

}

};

TEST\_CLASS(KursTest)

{

public:

TEST\_METHOD(TaskExample)

{

ifstream input("C:\\Kurs\\Input1.txt");

Flow flow(input);

Assert::AreEqual(flow.MaxFlow(), 5);

}

TEST\_METHOD(TaskExampleIstokStok)

{

ifstream input("C:\\Kurs\\Input6.txt");

Flow flow(input);

Assert::AreEqual(flow.MaxFlow(), 25);

}

TEST\_METHOD(IstokStokOnly)

{

ifstream input("C:\\Kurs\\Input7.txt");

Flow flow(input);

Assert::AreEqual(flow.MaxFlow(), 20);

}

TEST\_METHOD(LargeExample)

{

ifstream input("C:\\Kurs\\Input9.txt");

Flow flow(input);

Assert::AreEqual(flow.MaxFlow(), 19);

}

};

}