**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по лабораторной работе №2**

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

Тема: «Алгоритмы на графах»(2 вариант)

|  |  |  |
| --- | --- | --- |
| Студент гр. 8309 |  | Ильин Д.Д. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

# Цель работы

Реализовать программу, принимающую список городов и цен на прямой и обратный перелёт между ними и выдающую информацию о наиболее выгодном перелёте между какимилибо городами.

# Описание реализуемого класса

Класс Matrix, содержит поля double\*\* matrix (двумерный массив цен на рейсы, схожий с матрицей смежности), int size\_of\_matrix (размер матрицы смежности),Map<string, int>\* map\_City\_name\_to\_index (для хранения названия и получения его индекса), Map<int, string>\* map\_index\_to\_name\_City (для хранения индекса и получения его названия города), const int INF (значение «бесконечности», которого невозможно достичь (означает, что между пунктами нет пути)). Класс содержит следующие методы:

* Конструктор – получает на вход список строк, обрабатывает их и в результате выдает матрицу смежности, где по горизонтали начальный пункт, по вертикали – конечный, и на пересечении строк – цена перелёта из одного пункта в другой.
* Деструктор – вызывает метод clear (на основе обычного удаления двоичного дерева).
* *string Ford\_Bellman(string start\_City,string end\_City) –* функция поиска кратчайшего пути из города отправления *start\_City* в город пребывания *end\_City* по алгоритму Форда-Беллмана*.*
* *void print\_way (int i, int j, int\*\* p, Map<int, string>\* map\_index\_to\_name\_City, string&cur)* – функция, рекурсивно записывающая путь в строку.

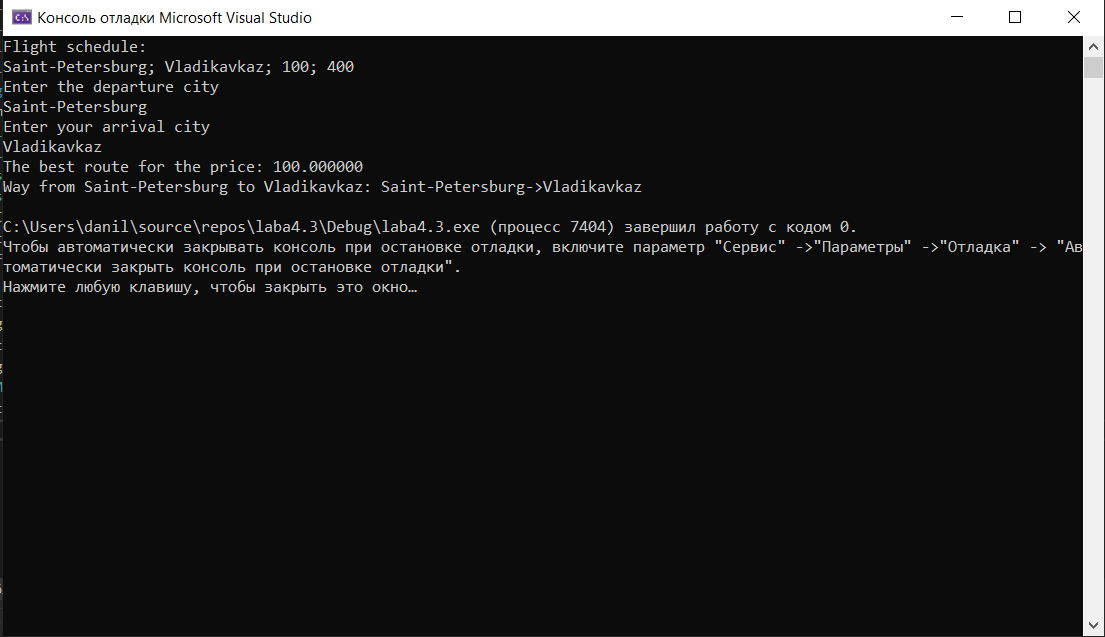
# Оценка временной сложности алгоритмов

* *string Ford\_Bellman(string start\_City, string end\_City)* – O(n^3)
* *print\_path (int i, int j, int\*\* p, Map<int, string>\* map\_index\_to\_name\_City, string&cur)* – O(n^2)

# Описание реализованных unit-тестов

Реализованные мною тесты проверяют нахождение кратчайшего (наиболее выгодного) маршрута: когда между городами есть маршрут и когда нет.

# Примеры работы программы



**Программа**

**laba4.3.cpp**

#include "iostream"

#include "Map.h"

#include "fstream"

#include "string"

#include "adjacency\_matrix.h"

#include "Used\_function.h"

using namespace std;

int main() {

ifstream input("input.txt");

List<string>\* list\_fly = new List<string>();

string city\_Start;

string city\_End;

InputDataFromFile(list\_fly, input);

cout << "Flight schedule: " << endl;

for (int i = 0; i < list\_fly->get\_size(); i++)

cout << list\_fly->at(i) << endl;

cout << "Enter the departure city" << endl;

getline(cin, city\_Start);

cout << "Enter your arrival city" << endl;

getline(cin, city\_End);

Matrix\* matrix\_ford\_bellman = new Matrix(list\_fly);

cout << matrix\_ford\_bellman->Ford\_Bellman(city\_Start, city\_End);

}

**adjacency\_matrix.h**

pragma once

#include"List.h"

#include"Map.h"

#include<string>

#include<iostream>

class Matrix {

public:

Matrix(List<string>\* data) {

map\_City\_name\_to\_index = new Map<string, int>();

map\_index\_to\_name\_City = new Map<int, string>();

int N = data->get\_size();

int index\_city = 0;

int index\_start\_vertex = -1; //starting vertex

for (int i = 0; i < N; i++) //we filled in all the indexes of different cities and count their number

{

string str\_cur = data->at(i);

int cur = str\_cur.find(';'); //first occurrence ";"

int cur1 = str\_cur.find(';', cur + 1); //second occurrence ";"

string str\_name\_city1 = str\_cur.substr(0, cur); //getting the first city

string str\_name\_city2 = str\_cur.substr(cur + 1, cur1 - cur - 1); //getting the second city

str\_name\_city2.erase(0, 1);

if (!map\_City\_name\_to\_index->find\_is(str\_name\_city1))

{

map\_City\_name\_to\_index->insert(str\_name\_city1, index\_city);

map\_index\_to\_name\_City->insert(index\_city, str\_name\_city1);

index\_city++;

}

if (!map\_City\_name\_to\_index->find\_is(str\_name\_city2))

{

map\_City\_name\_to\_index->insert(str\_name\_city2, index\_city);

map\_index\_to\_name\_City->insert(index\_city, str\_name\_city2);

index\_city++;

}

}

// filling in the path matrix

size\_of\_matrix = index\_city;

matrix = new double\* [size\_of\_matrix];

for (int i = 0; i < size\_of\_matrix; i++)

matrix[i] = new double[size\_of\_matrix];

for (int i = 0; i < size\_of\_matrix; i++)

for (int j = 0; j < size\_of\_matrix; j++)

matrix[i][j] = INF;

// filling in the price matrix

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

{

int price\_1\_to\_2 = INF;

int price\_2\_to\_1 = INF;

string str\_cur = data->at(i);

int cur = str\_cur.find(';'); //first entering ;

int cur1 = str\_cur.find(';', cur + 1); //second entering ;

int cur2 = str\_cur.find(';', cur1 + 1); //third entering ;

int cur3 = str\_cur.find(';', cur2 + 1); //fourth entering ;

string str\_name\_city1 = str\_cur.substr(0, cur); //getting the first city

string str\_name\_city2 = str\_cur.substr(cur + 1, cur1 - cur - 1); //getting the second city

str\_name\_city2.erase(0, 1);

if (str\_cur.substr(cur1 + 2, cur2 - 2 - cur1) != "N/A")

price\_1\_to\_2 = stof(str\_cur.substr(cur1 + 2, cur2 - 2 - cur1));

if (str\_cur.substr(cur2 + 2, cur3 - 1) != "N/A")

price\_2\_to\_1 = stoi(str\_cur.substr(cur2 + 2, cur3 - 2 - cur2));

matrix[map\_City\_name\_to\_index->find(str\_name\_city1)][map\_City\_name\_to\_index->find(str\_name\_city2)] = price\_1\_to\_2;

matrix[map\_City\_name\_to\_index->find(str\_name\_city2)][map\_City\_name\_to\_index->find(str\_name\_city1)] = price\_2\_to\_1;

}

}

string Ford\_Bellman(string start\_City, string end\_City)

{

string cur;

while (!map\_City\_name\_to\_index->find\_is(start\_City))

{

std::cout << "The departure city is missing, enter it again" << endl;

std::cin >> start\_City;

}

while (!map\_City\_name\_to\_index->find\_is(end\_City))

{

std::cout << "The arrival city is missing, enter it again" << endl;

std::cin >> end\_City;

}

int index\_start\_vertex = map\_City\_name\_to\_index->find(start\_City); //find the index of the city of departure

int index\_end\_vertex = map\_City\_name\_to\_index->find(end\_City); //find the index of the city of departure

double\* distance = new double[size\_of\_matrix];

int\* pre = new int[size\_of\_matrix]; //ancestors

for (int i = 0; i < size\_of\_matrix; i++)

{

distance[i] = INF;

pre[i] = -1;

}

distance[map\_City\_name\_to\_index->find(start\_City)] = 0;

for (int i = 0; i < size\_of\_matrix - 1; i++)

{

for (int j = 0; j < size\_of\_matrix; j++)

{

for (int k = 0; k < size\_of\_matrix; k++)

{

if (matrix[j][k] != INF)

{

if (distance[k] > distance[j] + matrix[j][k])

{

distance[k] = distance[j] + matrix[j][k];

pre[k] = j;

}

}

}

}

}

if (distance[map\_City\_name\_to\_index->find(end\_City)] != INF)

{

cur = "The best route for the price: " + to\_string(distance[map\_City\_name\_to\_index->find(end\_City)]) + '\n';

this->print\_way(cur, start\_City, end\_City, distance, pre);

cur.erase(cur.size() - 2);

cur += '\n';

}

else {

cur = "This route can't be built, try waiting for the flight schedule for tomorrow!";

}

return cur;

}

private:

void print\_way(string& cur, string start\_City, string end\_City, double\* distance, int\* pre)

{

List<int> way;

for (int cur = map\_City\_name\_to\_index->find(end\_City); cur != -1; cur = pre[cur])

way.push\_back(cur);

way.reverse();

cur = cur + "Way from " + start\_City + " to " + end\_City + ": ";

for (int i = 0; i < way.get\_size(); ++i)

cur = cur + map\_index\_to\_name\_City->find(way.at(i)) + "->";

}

double\*\* matrix;

int size\_of\_matrix;

Map<string, int>\* map\_City\_name\_to\_index;

Map<int, string>\* map\_index\_to\_name\_City;

const double INF = 1000000000;

};

**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;

}

};

**List.h**

#pragma once

using namespace std;

template<typename 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) // add to the end of the list

{

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) // adding to the top of the list

{

if (head != NULL)

{

Node\* current = new Node;

current->data = obj;

current->Next = this->head;

this->head = current;

}

else {

this->head = new Node(obj);

}

this->Size++;

}

void pop\_back() { // deleting the last element

if (head != NULL)

{

Node\* current = head;

while (current->Next != tail)//that is we are looking for the penultimate one

current = current->Next;

delete tail;

tail = current;

tail->Next = NULL;

Size--;

}

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

}

void pop\_front() { //deleting the first element

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) // adding an item by index (insert before an item that was previously available by this index)

{

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;//to add an element

Node\* current1 = head;//to search for the total element

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

{

current1 = current1->Next;

}

current->data = obj;

current->Next = current1->Next;//points to the next element

current1->Next = current;

Size++;

}

}

}

else {

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

}

}

T at(size\_t k) {// getting an item by index

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) { // deleting an item by index

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++) //go to the pre element

{

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() { // getting the list size

return Size;

}

void print\_to\_console() // output list items to the console using a separator, do not use at

{

if (this->head != NULL)

{

Node\* current = head;

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

{

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

current = current->Next;

}

}

}

void clear() // deleting all list items

{

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) // replacing an element by index with the passed element

{

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() // checking for an empty list

{

return (bool)(head);

}

void reverse() // changes the order of items in the list

{

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;

};

**Used\_function.h**

#pragma once

#include<iostream>

#include<fstream>

void InputDataFromFile(List<string>\* data, ifstream& file) //entering from a file

{

while (!file.eof()) {

string s1;

getline(file, s1);

data->push\_back(s1);

}

}

# Вывод

В данной лабораторной работе я познакомился с алгоритмом Форда-Беллмана и реализовал программу, находящую наиболее выгодный путь с помощью этого алгоритма.