Федеральное государственное автономное образовательное учреждение

высшего образования

“Нижегородский государственный университет

им. Н.И. Лобачевского”

Институт информационных технологий, математики и механики

Отчёт по лабораторной работе

Алгоритм Дейкстры

Выполнила:

студент группы 0823-1

Ермакова Е.А.

Проверил:

к.т.н., ассистент каф. пр. инж. ИИТММ

Сиднев А.А.

Нижний Новгород

2017 г.

Содержание

[Введение 3](#_Toc481018162)

[1 Постановка задачи 4](#_Toc481018163)

[2 Руководство пользователя 5](#_Toc481018164)

[3 Руководство программиста 6](#_Toc481018165)

[3.1 Описание структуры программы 6](#_Toc481018166)

[3.2 Описание модульной структуры программы 6](#_Toc481018167)

[3.3 Описание структур данных 12](#_Toc481018168)

[3.4 Описание алгоритмов 14](#_Toc481018169)

[Алгоритмы АВЛ-деревьев. 14](#_Toc481018170)

[Алгоритмы 5-куч. 18](#_Toc481018171)

[Алгоритм Дейкстры. 19](#_Toc481018172)

[Заключение 21](#_Toc481018173)

[Литература 22](#_Toc481018174)

[Приложения 23](#_Toc481018175)

[Приложение 1 (файл main.cpp) 23](#_Toc481018176)

[Приложение 2 (файл avl\_tree.h) 23](#_Toc481018177)

[Приложение 3 (файл five\_heap.h) 32](#_Toc481018178)

[Приложение 4 (файл priority\_queue.h) 34](#_Toc481018179)

[Приложение 4 (файл dijkstra.h) 36](#_Toc481018180)

[Приложение 5 (файл dijkstra.cpp) 36](#_Toc481018181)

[Приложение 6 (файл test\_tree.cpp) 37](#_Toc481018182)

[Приложение 7 (файл test\_heap.cpp) 40](#_Toc481018183)

[Приложение 8 (файл test\_queue.cpp) 44](#_Toc481018184)

[Приложение 9 (файл test\_dijkstra.cpp) 47](#_Toc481018185)

[Приложение 10 (файл main\_test.cpp) 49](#_Toc481018186)

# Введение

В современном мире развитая дорожная сеть является преимуществом любой страны. Однако с увеличением количества дорог между населенными пунктами усложняется задача поиска кратчайшего пути, которая является актуальной как для деловых людей, которые берегут свое время, так и для транспортных логистических компаний, ищущих путь наименьшей длины для сокращения расходов.

Карту можно представить в виде графа, где вершины – населенные пункты, а ребра – дороги. Поэтому возникает необходимость в программе, которая бы находила кратчайший путь между вершинами графа.

# 1 Постановка задачи

Требуется реализовать алгоритм Дейкстры поиска кратчайшего пути от заданной вершины до всех остальных вершин в графе.

Программа принимает на вход граф в виде списка смежности, где для каждой вершины указывается список смежных с ней вершин с указанием веса ребра между соответствующими вершинами. Также передается количество вершин в графе и стартовая вершина, от которой программа будет искать пути до других вершин графа.

Для реализации алгоритма необходимо использовать приоритетную очередь, в основе которой лежит: а) АВЛ-деревья; б) 5-куча.

Исходные данные:

g – граф в виде списка смежности;

s – стартовая вершина;

n – количество вершин в графе;

Требуемый рез-тат:

d – массив кратчайших расстояний от стартовой вершины до всех остальных вершин графа.

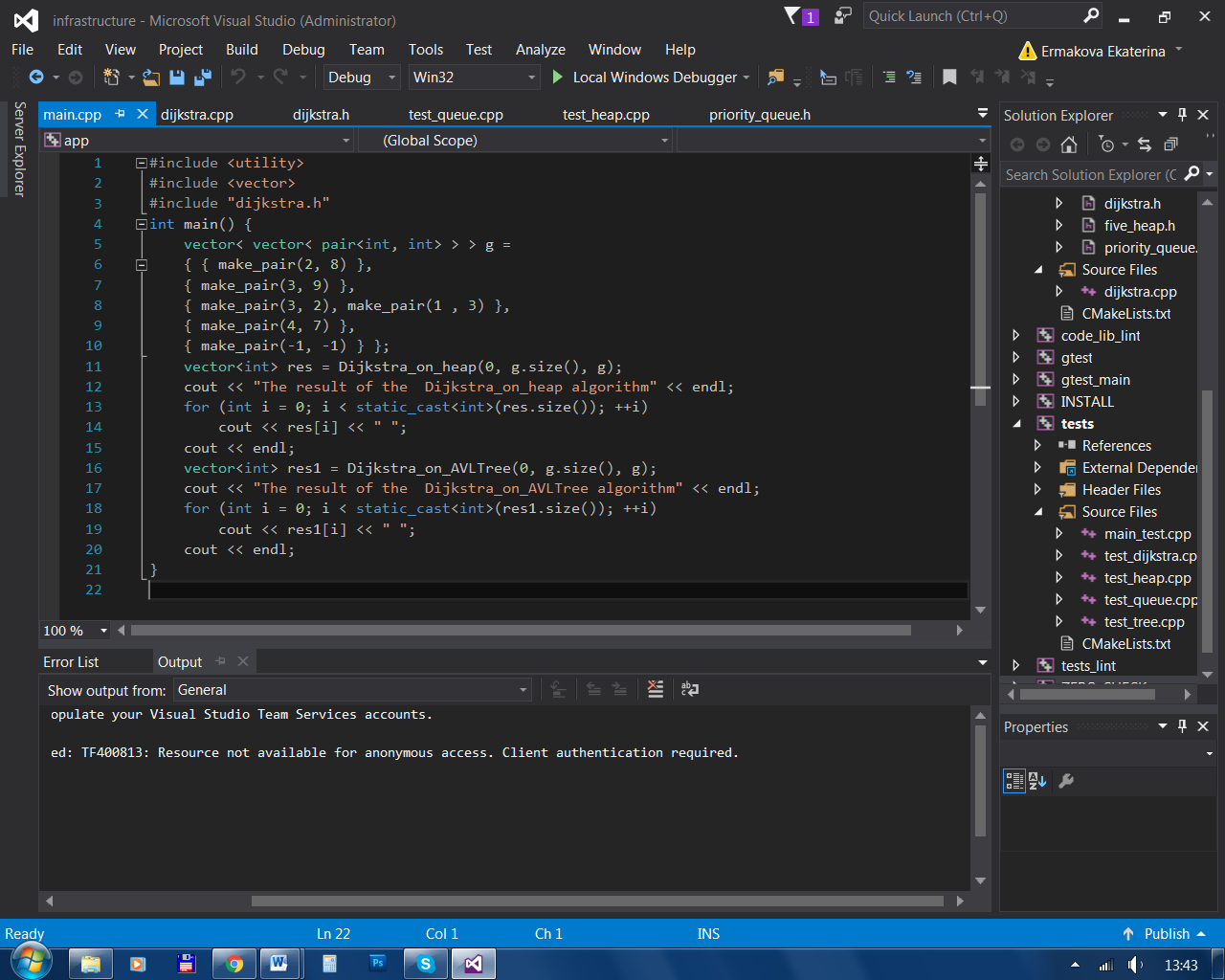
# 2 Руководство пользователя

Для нахождения массива кратчайших расстояний от заданной вершины до всех остальных вершин графа, пользователю доступны две программы:

1. Dijkstra\_on\_heap – поиск кратчайшего пути алгоритмом Дейкстры, реализованном на приоритетной очереди с использованием 5-кучи.
2. Dijkstra\_on\_AVLTree – поиск кратчайшего пути алгоритмом Дейкстры, реализованном на приоритетной очереди с использованием 5-кучи.

Для получения массива кратчайших путей необходимо:

1. Создать граф и инициализировать его.

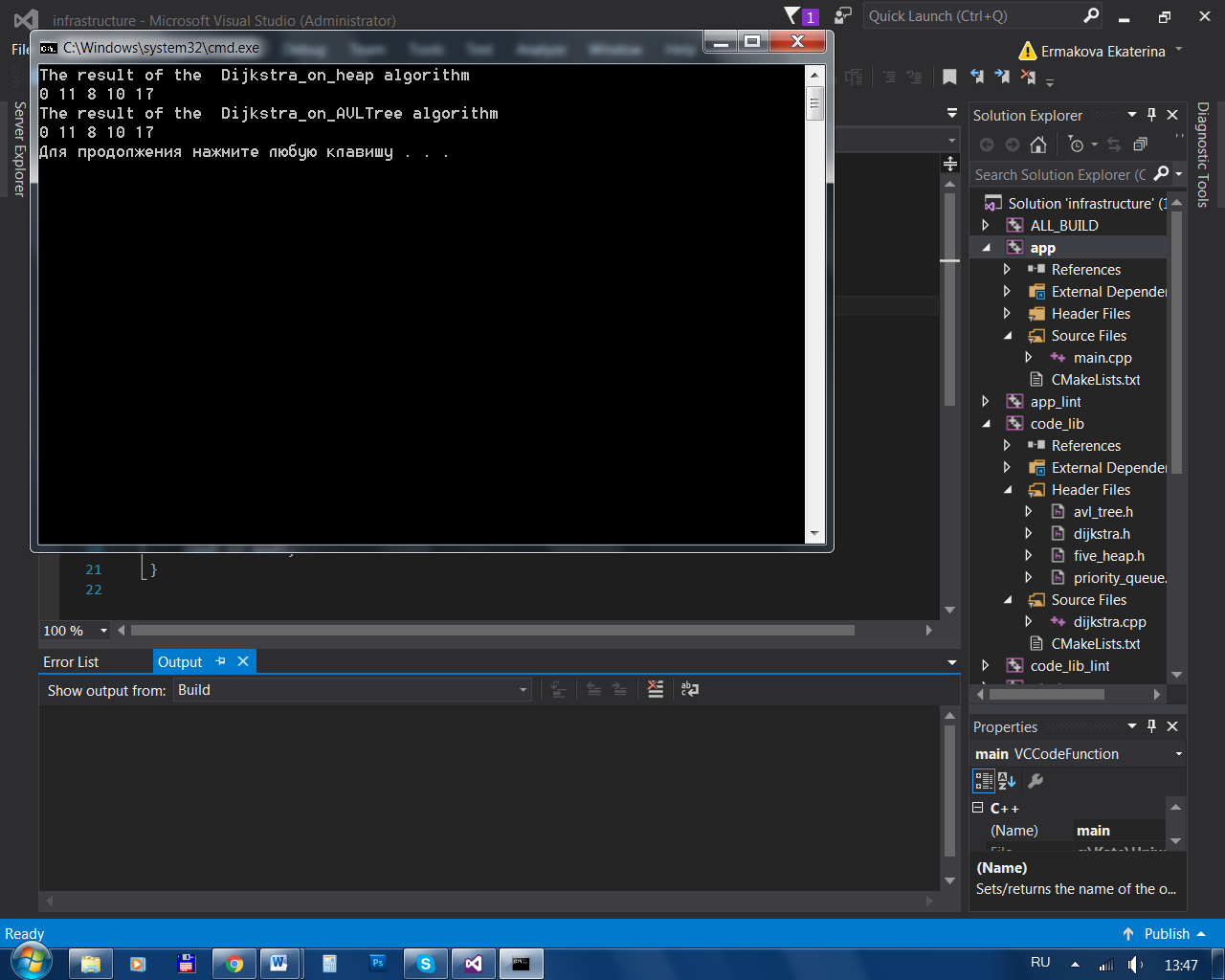


1. Вызвать интересующую функцию, передав соответствующие параметры (стартовая вершина, количество вершин, граф).

vector<int> res = Dijkstra\_on\_heap(0, g.size(), g);

vector<int> res1 = Dijkstra\_on\_AVLTree(0, g.size(), g);

1. Результат работы функций будет выведен в консоль



# 3 Руководство программиста

## 3.1 Описание структуры программы

Разработано решение, состоящее из следующих проектов:

1. *code.lib –* статическая библиотека, содержащая реализацию алгоритма Дейкстры и реализации вспомогательных структур (АВЛ-деревья, 5-кучи, приоритетные очереди). Содержит заголовочные файлы avl\_tree.h, dijkstra.h, five\_heap.h и priority\_queue.h, а также файл dijkstra.cpp.
2. *gtest.lib –* статическая библиотека. Содержит заголовочные файлы, а также файлы-реализации, необходимые для работы Google Tests, с помощью которых происходит проверка корректности программы.
3. *app.exe –* консольное приложение. Содержит файл main.cpp.
4. *mytest.exe -* консольное приложение, тестирующее работоспособность программы. Содержит файлы main\_test.cpp, test\_dijkstra.cpp, test\_heap.cpp, test\_tree.cpp, test\_queue.cpp.

## 3.2 Описание модульной структуры программы

* Файл main.cpp содержит функцию int main(), в которой осуществляются создание и инициализация графа, вызов функций по нахождению кратчайших путей от заданной вершины до остальных вершин и вывод результатов работы этих функций в консоль.
* Файл avl\_tree.h содержит:
  1. Директивы предпроцессора

#ifndef INCLUDE\_AVL\_TREE\_H\_

#define INCLUDE\_AVL\_TREE\_H\_

#include <stack>

#include <vector>

#include <stdexcept>

using std::stack;

using std::vector;

using std::logic\_error;

#endif // INCLUDE\_AVL\_TREE\_H\_

* 1. Объявление шаблонного класса avlnode с реализацией конструкторов

template <class Type>

class avlnode {

public:

Type val;

avlnode\* parent;

avlnode\* left;

avlnode\* right;

int balance;

avlnode() {

val = 0;

parent = left = right = nullptr;

balance = 0;

}

avlnode(const Type& v, avlnode\* par) {

val = v;

parent = par;

left = right = nullptr;

balance = 0;

}

};

* 1. Объявление шаблонного класса AVLTree

template <class Type>

class AVLTree {

void LeftRotation(avlnode<Type>\*\* knot);

void RightRotation(avlnode<Type>\*\* knot);

void DoubleLeftRotation(avlnode<Type>\*\* knot);

void DoubleRightRotation(avlnode<Type>\*\* knot);

public:

avlnode<Type>\* root;

AVLTree();

~AVLTree();

vector<avlnode<Type>\*> GetAllKnots();

int GetSize();

int Height(avlnode<Type>\* knot);

void Balancing(avlnode<Type>\*\* knot);

avlnode<Type>\* Findmin(avlnode<Type>\* knot);

avlnode<Type>\* Findnext(avlnode<Type>\* knot);

void Add(const Type& v);

void Insert(avlnode<Type> \*\*knot, avlnode<Type> \*\*new\_node);

void Erase(avlnode<Type>\*\* knot, const Type& value);

void Erase(const Type& value);

void TotalErase(avlnode<Type>\*\* knot);

};

* 1. Реализацию методов класса AVLTree
     + Конструктор:

AVLTree<Type>::AVLTree();

Метод инициализирует корень дерева нулем.

* + - Деструктор

AVLTree<Type>::~AVLTree();

Метод удаляет дерево.

* + - Функция обхода дерева:

vector<avlnode<Type>\*> AVLTree<Type>::GetAllKnots();

Функция обходит дерево и создает вектор всех узлов.

* + - Получение размера дерева:

int AVLTree<Type>::GetSize();

Функция возвращает количество вершин в дереве

* + - Функция простого левого поворота:

void AVLTree<Type>::LeftRotation(avlnode<Type>\*\* knot);

Функция получает указатель на узел, в котором нарушен баланс и выполняет поворот поддерева так, чтобы оно вновь стало сбалансированным.

* + - Функция простого правого поворота:

void AVLTree<Type>::RightRotation(avlnode<Type>\*\* knot)

Функция получает указатель на узел, в котором нарушен баланс и выполняет поворот поддерева так, чтобы оно вновь стало сбалансированным.

* + - Функция двойного левого поворота:

void AVLTree<Type>::DoubleLeftRotation(avlnode<Type>\*\* knot);

Функция получает указатель на узел, в котором нарушен баланс и выполняет поворот поддерева так, чтобы оно вновь стало сбалансированным.

* + - Функция двойного поворота:

void AVLTree<Type>::DoubleRightRotation(avlnode<Type>\*\* knot);

Функция получает двойной указатель на узел, в котором нарушен баланс и выполняет поворот поддерева так, чтобы оно вновь стало сбалансированным.

* + - Получение высоты поддерева:

int AVLTree<Type>::Height(avlnode<Type>\* knot);

Функция получает узел и возвращает высоту поддерева, корнем которого является полученный узел.

* + - Балансировка:

void AVLTree<Type>::Balancing(avlnode<Type>\*\* knot)

Функция получает указатель на корень поддерева и выполняет балансировку поддерева (делает дерево сбалансированным).

* + - Поиск узла с минимальным значением:

avlnode<Type>\* AVLTree<Type>::Findmin(avlnode<Type>\* knot);

Функция получает корень поддерева и возвращает узел с минимальным в этом поддереве значением.

* + - Поиск следующего по значению узла:

avlnode<Type>\* AVLTree<Type>::Findnext(avlnode<Type>\* knot);

Функция получает корень поддерева и возвращает узел со значением, следующим после значения корня в поддереве.

* + - Добавление узла:

void AVLTree<Type>::Add(const Type& value);

Функция получает значение, создает узел с таким значением и вставляет узел в дерево.

* + - Вставка узла в поддерево:

void AVLTree<Type>::Insert(avlnode<Type>\*\* knot, avlnode<Type>\*\* new\_node);

Функция получает узел и вставляет его в поддерево, также полученное через параметры функции.

* + - Удаление узла по значению из поддерева:

void AVLTree<Type>::Erase(avlnode<Type>\*\* knot, const Type& value);

Функция получает поддерево и значение, находит узел с таким значением и удаляет узел из поддерева.

* + - Удаление значения из дерева:

void AVLTree<Type>::Erase(const Type& value);

Функция получает значение и с помощью предыдущего метода удаляет соответствующий узел из дерева.

* + - Удаление поддерева:

void AVLTree<Type>::TotalErase(avlnode<Type>\*\* knot)

Функция получает указатель на поддерево, удаляет все поддерево и инициализирует его корень нулевым указателем.

* Файл five\_heap.h содержит:
  1. Директивы предпроцессора и объявление констант:

#ifndef INCLUDE\_FIVE\_HEAP\_H\_

#define INCLUDE\_FIVE\_HEAP\_H\_

#include <stdexcept>

const int d\_ = 5;

const int MAX\_SIZE = 1000;

using std::logic\_error;

#endif // INCLUDE\_FIVE\_HEAP\_H\_

* 1. Объявление шаблонного класса FIVEheap:

template <class Type>

class FIVEheap{

int size;

int d;

public:

Type\* val;

FIVEheap();

~FIVEheap();

int GetSize() { return size; }

bool operator == (const FIVEheap<Type>& B);

bool operator != (const FIVEheap<Type>& B);

Type& operator[] (int i);

void Reduction(int i, const Type& con);

void Swap(int i, int j);

int MinChild(int i);

void Insert(const Type& val\_);

void Erase(int i);

void EraseMin();

void Immertion(int i); // down

void Emertion(int i); // up

void DoingHeaps();

bool IsFull() { return (size >= MAX\_SIZE); }

bool IsEmpty() { return size == 0; }

};

* 1. Реализацию методов класса FIVEheap:
     + Конструктор:

FIVEheap<Type>::FIVEheap();

Метод создает пустой массив для элементов кучи и устанавливает значение размера, равное 0.

* + - Деструктор:

FIVEheap<Type>::~FIVEheap();

Метод удаляет кучу.

* + - Перегрузка операции ==:

bool FIVEheap<Type>::operator==(const FIVEheap<Type>& B);

Функция получает ссылку на кучу и возвращает true, если элементы кучи и полученной кучи полностью совпадают. В противном случае метод возвращает false.

* + - Перегрузка операции !=:

bool FIVEheap<Type>::operator!=(const FIVEheap<Type>& B);

Функция получает ссылку на кучу и возвращает true, кучи не совпадают. В противном случае метод возвращает false.

* + - Перегрузка []:
    - Type& FIVEheap<Type>::operator[](int i);

Функция получает ключ и возвращает значение элемента, лежащему по этому ключу.

* + - Уменьшение значения на константу:

void FIVEheap<Type>::Reduction(int i, const Type & con);

Функция получает ключ и константу, находит элемент, лежащий по этому ключу, и уменьшает его значение на указанную константу, после чего производится «всплытие» элемента.

* + - Транспонирование:

void FIVEheap<Type>::Swap(int i, int j);

Функция получает два ключа и меняет местами значения элементов, лежащих по этим ключам.

* + - Поиск минимального потомка:

int FIVEheap<Type>::MinChild(int i);

Функция получает ключ и возвращает ключ минимального по значению потомка, принадлежащего элементу с данным ключом.

* + - Вставка:

void FIVEheap<Type>::Insert(const Type& val\_);

Функция получает значение и добавляет элемент с этим значением в кучу.

* + - Удаление:

void FIVEheap<Type>::Erase(int i);

Функция получает ключ и удаляет элемент, лежащий по этому ключу, из кучи.

* + - Удаление минимального элемента:

void FIVEheap<Type>::EraseMin();

Функция находит минимальный элемент в куче и удаляет его.

* + - Погружение:

void FIVEheap<Type>::Immertion(int i);

Функция получает ключ и «опускает» соответствующий элемент на нужное для восстановления структуры кучи место.

* + - Всплытие:

void FIVEheap<Type>::Emertion(int i);

Функция получает ключ и «поднимает» соответствующий элемент на нужное для восстановления структуры кучи место.

* + - Окучивание:

void FIVEheap<Type>::DoingHeaps();

Функция превращает массив вершин в 5-кучу.

* Файл priority\_queue.h содержит:
  1. Директивы предпроцессора

#ifndef INCLUDE\_PRIORITY\_QUEUE\_H\_

#define INCLUDE\_PRIORITY\_QUEUE\_H\_

#include "avl\_tree.h"

#include "five\_heap.h"

template <class Type>

#endif // INCLUDE\_PRIORITY\_QUEUE\_H\_

* 1. Описание шаблонного класса Queue\_on\_heap:

template <class Type>

class Queue\_on\_heap {

FIVEheap<Type>\* heap;

public:

Queue\_on\_heap();

~Queue\_on\_heap();

int GetSize() { return heap->GetSize(); }

bool IsFull() { return heap->IsFull(); }

bool IsEmpty() { return heap->IsEmpty(); }

void Push(const Type& val);

void Pop();

Type Top();

};

* 1. Реализацию методов класса Queue\_on\_heap:
     + Конструктор:

Queue\_on\_heap<Type>::Queue\_on\_heap();

Метод создает вершину кучи.

* + - Деструктор:

Queue\_on\_heap<Type>::~Queue\_on\_heap()

Метод удаляет кучу.

* + - Вставка:

void Queue\_on\_heap<Type>::Push(const Type & val);

Функция получает значение элемента и добавляет его в очередь в соответствие с величиной значения.

* + - Удаление первого в очереди:

void Queue\_on\_heap<Type>::Pop();

Функция удаляет элемент, стоящий вначале очереди.

* + - Получение первого в очереди:

Type Queue\_on\_heap<Type>::Top();

Функция возвращает значение первого в очереди элемента.

* 1. Описание шаблонного класса Queue\_on\_AVLTree:

template <class Type>

class Queue\_on\_AVLTree {

AVLTree<Type>\* tree;

public:

Queue\_on\_AVLTree();

~Queue\_on\_AVLTree();

int GetSize() { return tree->GetSize(); }

bool IsFull();

bool IsEmpty() { return (tree->GetSize() == 0); }

void Push(const Type& val) { tree->Add(val); }

void Pop();

Type Top();

};

* 1. Реализацию методов класса Queue\_on\_AVLTree:
     + Конструктор:

Queue\_on\_AVLTree<Type>::Queue\_on\_AVLTree();

Метод создает вершину дерева.

* + - Деструктор:

Queue\_on\_AVLTree<Type>::~Queue\_on\_AVLTree();

Метод удаляет дерево.

* + - Проверка на полноту:

bool Queue\_on\_AVLTree<Type>::IsFull();

Функция возвращает 1 в случае, если невозможно добавить новый элемент в очередь.

* + - Удаление первого в очереди:

void Queue\_on\_AVLTree<Type>::Pop();

Функция удаляет элемент, стоящий вначале очереди.

* + - Получение первого в очереди:

Type Queue\_on\_AVLTree<Type>::Top()

Функция возвращает значение первого в очереди элемента.

* Файл dijkstra.h содержит:
  1. Директивы предпроцессора и объявление констант:

#ifndef INCLUDE\_DIJKSTRA\_H\_

#define INCLUDE\_DIJKSTRA\_H\_

#include "priority\_queue.h"

#include <iostream>

#include <vector>

#include <string>

#include <algorithm>

#include <set>

#include <utility>

using std::cin;

using std::cout;

using std::endl;

using std::set;

using std::vector;

using std::pair;

using std::make\_pair;

const int INF = 1000000;

#endif // INCLUDE\_DIJKSTRA\_H\_

* 1. Прототипы функций Dijkstra\_on\_heap (алгоритм Дейкстры с использованием 5-кучи) и Dijkstra\_on\_AVLTree (алгоритм Дейкстры с использованием АВЛ-деревьев)

vector<int> Dijkstra\_on\_heap(int s, int n, vector<vector<pair<int, int>>> g);

vector<int> Dijkstra\_on\_AVLTree(int s, int n, vector<vector<pair<int, int>>> g);

* Файл dijkstra.cpp содержит реализацию следующих функций:
  1. Алгоритм Дейкстры с использованием 5-кучи:

vector<int> Dijkstra\_on\_heap(int s, int n, vector<vector<pair<int, int>>> g);

Функция принимает граф в виде списка смежности с указанием веса ребер, стартовую вершину и количество вершин в графе и возвращает массив кратчайших расстояний между стартовой и всеми остальными вершинами, при этом в реализации используется приоритетная очередь на 5-куче.

* 1. Алгоритм Дейкстры с использованием АВЛ-деревьев:

vector<int> Dijkstra\_on\_AVLTree(int s, int n, vector<vector<pair<int, int>>> g);

Функция принимает граф в виде списка смежности с указанием веса ребер, стартовую вершину и количество вершин в графе и возвращает массив кратчайших расстояний между стартовой и всеми остальными вершинами, при этом в реализации используется приоритетная очередь на АВЛ-деревьях.

* Файл main\_test.cpp содержит функцию для запусков всех тестов:

int main(int ac, char\* av[]);

* Файл test\_tree содержит тесты для проверки корректности методов класса AVLTree.
* Файл test\_heap содержит тесты для проверки корректности методов класса FIVEheap.
* Файл test\_queue содержит тесты для проверки корректности методов классов Queue\_on\_AVLTree и Queue\_on\_heap.
* Файл test\_dijkstra содержит тесты для проверки корректности функций Dijkstra\_on\_heap и Dijkstra\_on\_AVLTree.

## 3.3 Описание структур данных

В разработанной программе использованы следующие структуры данных:

1. Класс для представления узлов АВЛ – деревьев:

template <class Type>

class avlnode {

public:

Type val;

avlnode\* parent;

avlnode\* left;

avlnode\* right;

int balance;

avlnode() {

val = 0;

parent = left = right = nullptr;

balance = 0;

}

avlnode(const Type& v, avlnode\* par) {

val = v;

parent = par;

left = right = nullptr;

balance = 0;

}

};

1. Класс АВЛ-деревьев:

template <class Type>

class AVLTree {

void LeftRotation(avlnode<Type>\*\* knot);

void RightRotation(avlnode<Type>\*\* knot);

void DoubleLeftRotation(avlnode<Type>\*\* knot);

void DoubleRightRotation(avlnode<Type>\*\* knot);

public:

avlnode<Type>\* root;

AVLTree();

~AVLTree();

vector<avlnode<Type>\*> GetAllKnots();

int GetSize();

int Height(avlnode<Type>\* knot);

void Balancing(avlnode<Type>\*\* knot);

avlnode<Type>\* Findmin(avlnode<Type>\* knot);

avlnode<Type>\* Findnext(avlnode<Type>\* knot);

void Add(const Type& v);

void Insert(avlnode<Type> \*\*knot, avlnode<Type> \*\*new\_node);

void Erase(avlnode<Type>\*\* knot, const Type& value);

void Erase(const Type& value);

void TotalErase(avlnode<Type>\*\* knot);

};

1. Класс 5 – куч:

template <class Type>

class FIVEheap{

int size;

int d;

public:

Type\* val;

FIVEheap();

~FIVEheap();

int GetSize() { return size; }

bool operator == (const FIVEheap<Type>& B);

bool operator != (const FIVEheap<Type>& B);

Type& operator[] (int i);

void Reduction(int i, const Type& con);

void Swap(int i, int j);

int MinChild(int i);

void Insert(const Type& val\_);

void Erase(int i);

void EraseMin();

void Immertion(int i); // down

void Emertion(int i); // up

void DoingHeaps();

bool IsFull() { return (size >= MAX\_SIZE); }

bool IsEmpty() { return size == 0; }

};

1. Класс приоритетной очереди на 5-куче:

template <class Type>

class Queue\_on\_heap {

FIVEheap<Type>\* heap;

public:

Queue\_on\_heap();

~Queue\_on\_heap();

int GetSize() { return heap->GetSize(); }

bool IsFull() { return heap->IsFull(); }

bool IsEmpty() { return heap->IsEmpty(); }

void Push(const Type& val);

void Pop();

Type Top();

};

1. Класс приоритетной очереди на АВЛ-деревьях:

template <class Type>

class Queue\_on\_AVLTree {

AVLTree<Type>\* tree;

public:

Queue\_on\_AVLTree();

~Queue\_on\_AVLTree();

int GetSize() { return tree->GetSize(); }

bool IsFull();

bool IsEmpty() { return (tree->GetSize() == 0); }

void Push(const Type& val) { tree->Add(val); }

void Pop();

Type Top();

};

1. Вектор g с данными типа vector<pair<int, int>>> из стандартной библиотеки std для представления графа:

vector<vector<pair<int, int>>> g;

1. Вектор d с данными типа int из стандартной библиотеки std для хранения кратчайших путей:

vector<int> d(n, INF);

1. Пары из стандартной библиотеки std для хранения вершины и веса ребра до этой вершины из стартовой вершины.

## 3.4 Описание алгоритмов

В лабораторной работе для нахождения кратчайших расстояний от заданной вершины до остальных вершин графа были использованы следующие алгоритмы:

### Алгоритмы АВЛ-деревьев.

1. Алгоритм простого левого поворота в АВЛ-дереве.

Такая операция выполняется, когда в правое поддерево правого потомка добавляется узел, в результате чего нарушается баланс.

На входе функция получает указатель на корень поддерева, которое нужно сбалансировать.

* 1. Правым потомком исходного корня поддерева делаем левое поддерево правого потомка исходного корня поддерева.
  2. Родителем левого поддерева правого потомка исходного корня поддерева делаем исходный корень поддерева.
  3. Левым потомком правого потомка исходного корня поддерева делаем исходный корень поддерева.
  4. Родителем правого потомка исходного корня поддерева делаем родитель исходного корня поддерева.
  5. Родителем исходного корня поддерева делаем правый потомок исходного корня поддерева.
  6. Находим новые значения высот исходного корня поддерева и его правого потомка.
  7. Правый потомок исходного корня поддерева назначается новым корнем поддерева.

1. Алгоритм простого правого поворота в АВЛ-дереве.

Такая операция выполняется, когда в левое поддерево левого потомка добавляется узел, в результате чего нарушается баланс.

На входе функция получает указатель на корень поддерева, которое нужно сбалансировать.

* 1. Левым потомком исходного корня поддерева делаем правое поддерево левого потомка исходного корня.
  2. Родителем правого поддерева левого потомка исходного корня поддерева делаем исходный корень поддерева.
  3. Правым потомком левого потомка исходного корня поддерева делаем исходный корень поддерева.
  4. Родителем левого потомка исходного корня поддерева делаем родитель исходного корня поддерева.
  5. Родителем исходного корня поддерева делаем левый потомок исходного корня поддерева.
  6. Находим новые значения высот исходного корня поддерева и его левого потомка.
  7. Левый потомок исходного корня поддерева назначается новым корнем поддерева.

1. Алгоритм двойного левого поворота в АВЛ-дереве.

Такая операция применяется, когда в левое поддерево правого потомка добавляется узел, в результате чего нарушается баланс.

Функция получает на входе указатель на корень поддерева, которое нужно сбалансировать.

* 1. Левым потомком добавленного узла делаем исходный корень поддерева.
  2. Родителем добавленного узла делаем родитель исходного корня поддерева.
  3. Родителем исходного корня поддерева делаем добавленный узел.
  4. Правым потомком добавленного узла делаем правый потомок исходного корня поддерева.
  5. Родителем правого потомка исходного корня поддерева делаем добавленный узел.
  6. Левым потомком правого потомка исходного корня поддерева делаем правый потомок добавленного узла.
  7. Родителем правого потомка добавленного узла делаем правый потомок исходного корня поддерева.
  8. Правым потомком исходного корня поддерева делаем левый потомок добавленного узла.
  9. Родителем левого потомка добавленного узла делаем исходный корень поддерева.
  10. Находим новые значения высот для исходного корня поддерева, его правого потомка и добавленного узла.
  11. Новым корнем поддерева делаем добавленный узел.

1. Алгоритм двойного правого поворота в АВЛ-дереве.

Такая операция применяется, когда в правое поддерево левого потомка добавляется узел, в результате чего нарушается баланс.

Функция получает на входе указатель на корень поддерева, которое нужно сбалансировать.

* 1. Правым потомком добавленного узла делаем исходный корень поддерева.
  2. Родителем добавленного узла делаем родитель исходного корня поддерева.
  3. Родителем исходного корня поддерева делаем добавленный узел.
  4. Левым потомком добавленного узла делаем левый потомок исходного корня поддерева.
  5. Родителем левого потомка исходного корня поддерева делаем добавленный узел.
  6. Правым потомком левого потомка исходного корня поддерева делаем левый потомок добавленного узла.
  7. Родителем левого потомка добавленного узла делаем левый потомок исходного корня поддерева.
  8. Левым потомком исходного корня поддерева делаем правый потомок добавленного узла.
  9. Родителем правого потомка добавленного узла делаем исходный корень поддерева.
  10. Находим новые значения высот для исходного корня поддерева, его левого потомка и добавленного узла.
  11. Новым корнем поддерева делаем добавленный узел.

1. Алгоритм получения высоты поддерева.

Функция получает на входе корень поддерева.

* 1. Если корень нулевой – функция возвращает значение -1.
  2. Вычисляем значение высот левого и правого поддерева корня.
  3. В баланс корня записывается разность высот левого и правого поддерева.

Функция возвращает наибольшее из значений, полученных в пункте 2, увеличенное на 1.

1. Алгоритм балансировки.

Функция получает на входе указатель на корень поддерева.

* 1. Функция вычисляет высоту поддерева.
  2. Если баланс поддерева равен 2, значит, его левое поддерево больше правого.

Определяем баланс левого потомка корня поддерева. Если баланс не отрицателен – значит, левое поддерево левого потомка больше его правого поддерева, и необходимо сделать простой правый поворот. В противном случае функция выполняет двойной правый поворот.

* 1. Если баланс поддерева равен -2, то его правое поддерево больше левого. Определяем баланс правого потомка корня поддерева. Если баланс не положителен – значит, правое поддерево правого потомка больше его левого поддерева, и необходимо сделать простой левый поворот. В противном случае функция выполняет двойной левый поворот.

1. Нахождение минимального элемента.

Функция получает на входе корень поддерева.

* 1. Если корень равен нулю – функция возвращает 0.
  2. Создается локальный корень дерева, и пока левый потомок текущего узла не равен 0, локальному корню присваивается значение его левого потомка.

Функция возвращает узел с минимальным значением.

1. Нахождение следующего по значению элемента в дереве.

Функция получает на входе корень поддерева.

* 1. Функция работает только с ненулевым значением корня.
  2. Если правый поддерево не пусто, то функция возвращает узел с минимальным значением из правого поддерева.
  3. Если правого поддерева не существует, то функция ищет узел, который приходится левым сыном для рассматриваемого элемента.

Функция возвращает узел со следующим по возрастанию значением после значения корня поддерева.

1. Алгоритм вставки узла в АВЛ-дерево.

Функция получает на входе указатель на узел и указатель на корень поддерева, в которое нужно вставить узел.

* 1. Если значение узла меньше значения корня поддерева, тогда:
     1. Если у корня поддерева не существует левого потомка, то добавляем на его место данный функции узел.
     2. В противном случае функция вставляет узел в левое поддерево.
  2. Если значение узла больше значения корня поддерева, тогда:
     1. Если у корня поддерева не существует правого потомка, то добавить на его место данный функции узел.
     2. В противном случае функция вставляет узел в правое поддерево.
  3. Проводим балансировку дерева.

1. Алгоритм удаления узла по значению из дерева.

На входе функция получает указатель на поддерево и значение узла, которое нужно удалить.

* 1. Если полученное значение меньше значение корня – функция продолжает удаление в левом поддереве корня.
  2. Если полученное значение больше значения корня – функция продолжает удаление в правом поддереве корня.
  3. Если полученное значение совпадает со значением корня, тогда:
     1. Функция определяет потомков корня. В случае, когда оба равны нулю, функция удаляет корень.
     2. Если только правый потомок корня равен нулю, тогда левый потомок становится корнем поддерева, а сам корень удаляется.
     3. Если только левый потомок равен нулю, то правый потомок становится корнем поддерева, а исходный корень удаляется.
     4. Если существуют и левый, и правый потомки корня поддерева, тогда функция ищет следующий по значению узел, добавляет его на место корня и удаляет исходный корень поддерева.
  4. Функция проводит балансировку полученного дерева.

1. Алгоритм полного удаления дерева.

Функция получает на входе указатель на корень поддерева, которое нужно удалить.

* 1. Если корень равен 0, функция удаляет его.
  2. Если существует левый потомок корня, функция удаляет левое поддерево.
  3. Если существует правый потомок корня, функция удаляет правое поддерево.
  4. Функция удаляет корень поддерева.

### Алгоритмы 5-куч.

1. Алгоритм вставки элемента по значению в кучу.

Функция получает на входе значение элемента, требующего вставки.

* 1. Функция добавляет элемент в конец кучи.
  2. Функция увеличивает размер кучи.
  3. Функция производит «всплытие» добавленного элемента.

1. Алгоритм удаления элемента по ключу.

Функция получает ключ элемента, который нужно удалить.

* 1. На место элемента, который требуется удалить, функция вставляет последний элемент кучи.
  2. Если вставленный элемент меньше своего родителя, функция проводит «всплытие» этого элемента. В противном случае происходит «погружение» элемента.

1. Алгоритм удаления минимального элемента кучи.
   1. На место первого элемента кучи функция вставляет последний элемент и производит «погружение» этого элемента.
2. Алгоритм поиска минимального потомка элемента.

Функция получает ключ элемента, минимального ребенка которого нужно найти.

* 1. Если ключ первого потомка больше размера кучи – значит, у элемента нет детей.
  2. Функция получает ключ первого ребенка элемента и сохраняет значение этого потомка как минимальное.
  3. Функция получает ключ последнего потомка элемента.
  4. Функция перебирает всех детей элемента от первого до последнего и сравнивает их значение с минимальным. Если был найден потомок с меньшим значением, функция сохраняет его ключ, а значение сохраняет как новое минимальное.

Функция возвращает ключ минимального потомка.

1. Алгоритм «погружения»

Функция получает на входе ключ элемента, который нужно «погрузить».

* 1. Функция находит минимального потомка элемента.
  2. Функция меняет элемент и его минимального потомка местами.
  3. Находим элемент, который является новым минимальным потомком для исходного элемента.
  4. Функция повторяет пункты 2 и 3 до тех пор, пока значение элемента больше значение минимального потомка.

1. Алгоритм «всплытия».

Функция получает на входе ключ элемента, который нужно «поднять».

* 1. Функция находит родителя элемента.
  2. Производится транспозиция элемента и его родителя.
  3. Функция находит нового родителя элемента.
  4. Пункты 2 и 3 повторяются до тех пор, пока значение элемента меньше значения его родителя.

1. Алгоритм «окучивания».

Функция проводит «погружение» последовательное погружение узлов size-1…0.

### Алгоритм Дейкстры.

Функция получает на входе граф в виде списка смежности, стартовую вершину и количество вершин в графе.

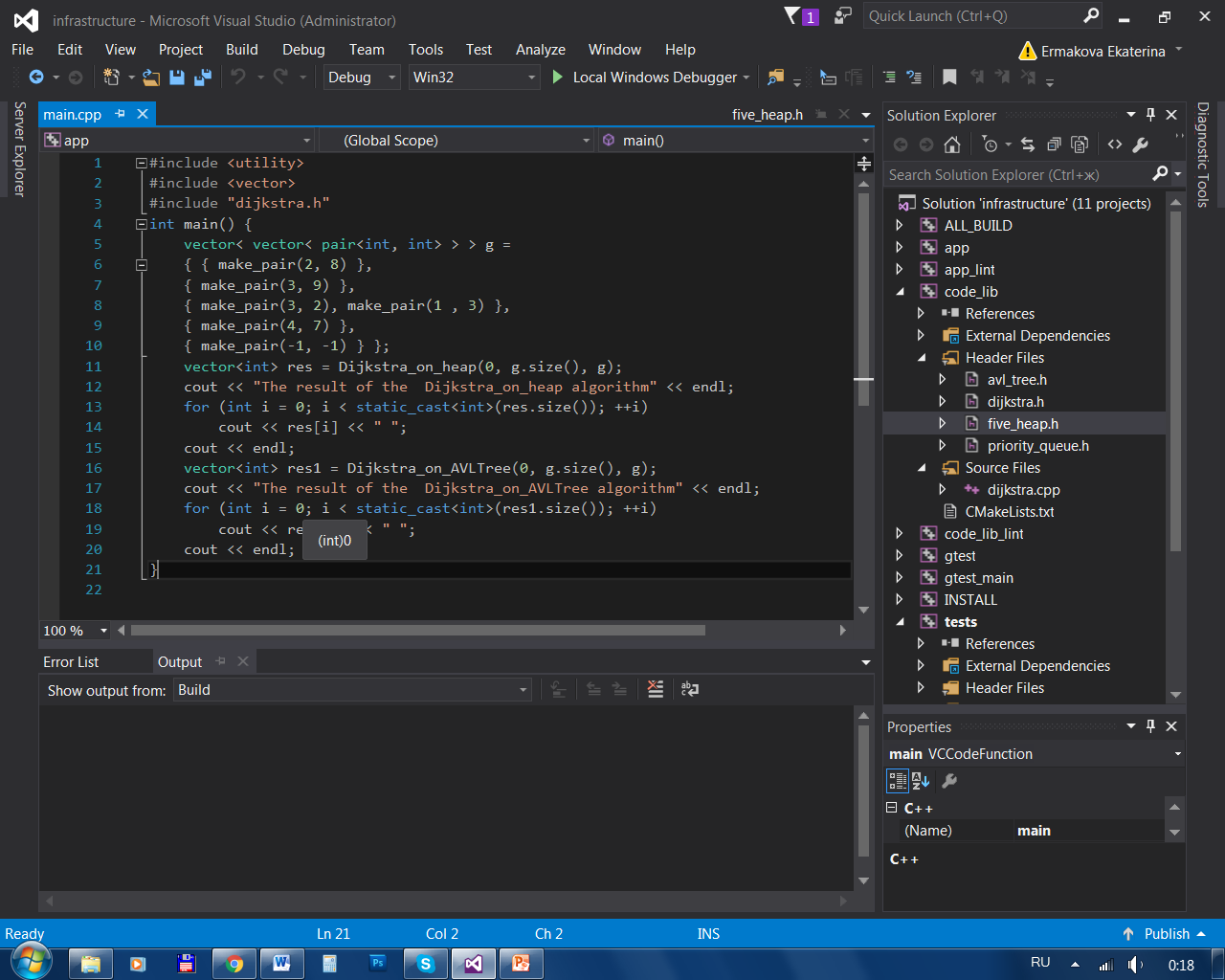
* 1. Функция создает массив дистанций до каждой вершины. Каждая дистанция кладется равной глобальной константе.
  2. Расстояние до стартовой вершины обнуляется.
  3. Функция создает приоритетную очередь пар «расстояние-вершина».
  4. Функция создает пару «расстояние до стартовой вершины (то есть 0) - стартовая вершина» и кладет пары в очередь.
  5. Функция получает вершину из очереди, расстояние до которой минимально, и само расстояние.
  6. Полученная в пункте 4 вершина удаляется из очереди.
  7. Функция сравнивает значение расстояния до полученной вершины, которое было получено в пункте 4, со значением уже имеющегося до этой вершины пути. Если новое расстояние больше имеющегося, возвращаемся к пункту 4.
  8. Если новое расстояние меньше, то функция перебирает все вершины, смежные с полученной вершиной. Функция получает расстояние до каждой смежной вершины от стартовой. Если текущее расстояние до смежной вершины больше, чем расстояние до полученной вершины от стартовой плюс расстояние от полученной вершины, то функция присваивает расстоянию до смежной вершины новое значение и добавляет пару «расстояние до смежной вершины – смежная вершина».
  9. Пункты 5 – 8 повторяются, пока очередь не пуста.

Функция возвращает массив минимальных расстояний от стартовой вершины до всех остальных вершин графа.

# Заключение

В ходе лабораторной работы был реализован алгоритм Дейкстры. Для возможности его применения были разработаны шаблонные классы приоритетных очередей на АВЛ-деревьях Queue\_on\_AVLTree и на 5-кучах Queue\_on\_heap, для реализации который в свою очередь были реализованы шаблонные классы АВЛ-деревьев AVLTree (с использованием разработанного класса avlnode) и 5-куч FIVEheap.

Результаты работы представлены ниже.



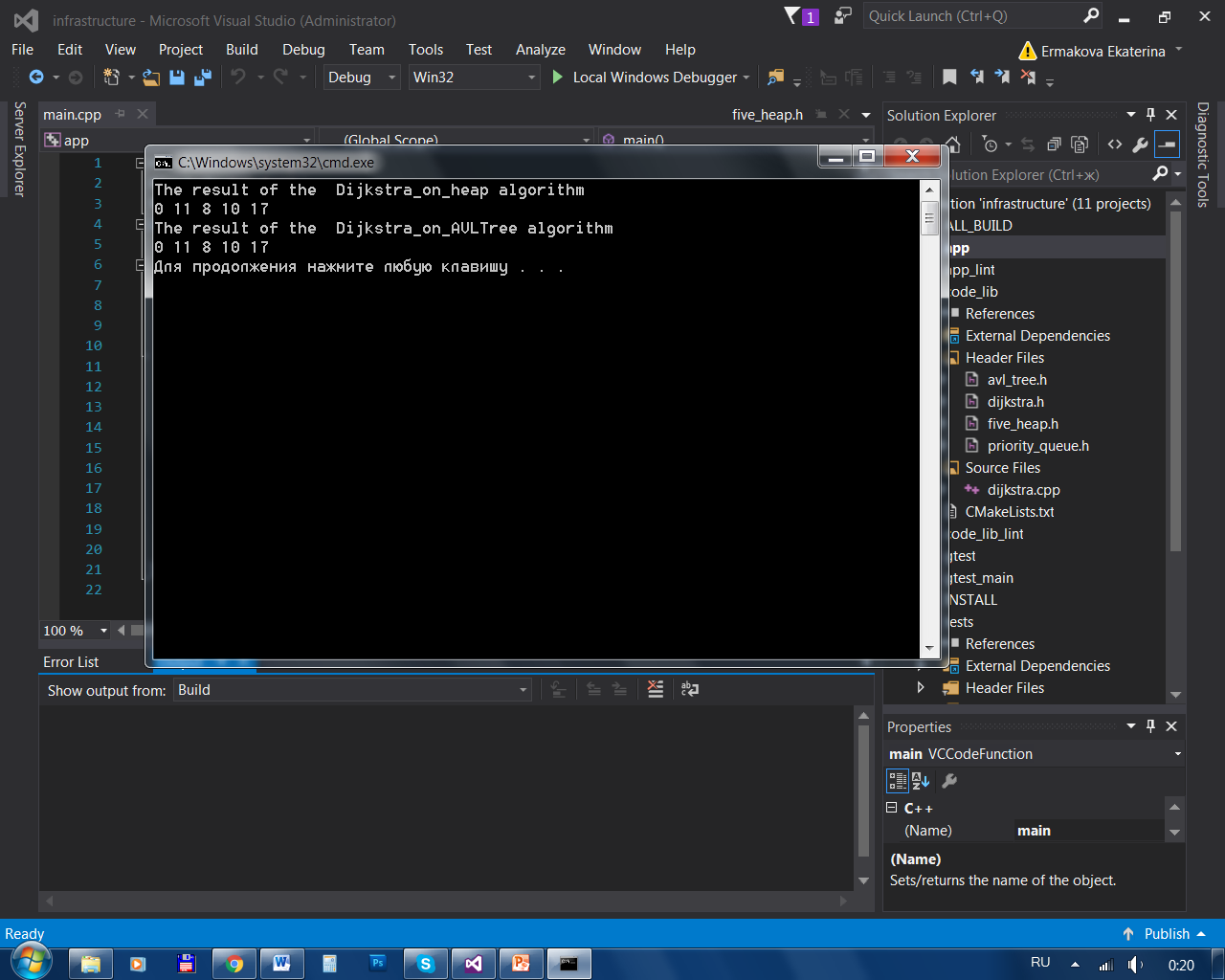


Рис.1 Результат работы программы при указанных начальных условиях.

Полученные результаты полностью удовлетворяют поставленным учебно-практическим задачам.

# Литература

1. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. 2-е издание. – М.: Вильямс, 2010.
2. Вирт Н. Алгоритмы и структуры данных. - М.: Мир, 1989.
3. Алексеев В.Е., Таланов В.А. Графы. Модели вычислений. Структуры данных: Учебник. – Нижний Новгород ННГУ, 2005.

# 

# Приложения

## Приложение 1 (файл main.cpp)

#include <utility>

#include <vector>

#include "dijkstra.h"

int main() {

vector< vector< pair<int, int> > > g =

{ { make\_pair(2, 8) },

{ make\_pair(3, 9) },

{ make\_pair(3, 2), make\_pair(1 , 3) },

{ make\_pair(4, 7) },

{ make\_pair(-1, -1) } };

vector<int> res = Dijkstra\_on\_heap(0, g.size(), g);

cout << "The result of the Dijkstra\_on\_heap algorithm" << endl;

for (int i = 0; i < static\_cast<int>(res.size()); ++i)

cout << res[i] << " ";

cout << endl;

vector<int> res1 = Dijkstra\_on\_AVLTree(0, g.size(), g);

cout << "The result of the Dijkstra\_on\_AVLTree algorithm" << endl;

for (int i = 0; i < static\_cast<int>(res1.size()); ++i)

cout << res1[i] << " ";

cout << endl;

}

## Приложение 2 (файл avl\_tree.h)

#ifndef INCLUDE\_AVL\_TREE\_H\_

#define INCLUDE\_AVL\_TREE\_H\_

#include <stack>

#include <vector>

#include <stdexcept>

using std::stack;

using std::vector;

using std::logic\_error;

template <class Type>

class avlnode {

public:

Type val;

avlnode\* parent;

avlnode\* left;

avlnode\* right;

int balance;

avlnode() {

val = 0;

parent = left = right = nullptr;

balance = 0;

}

avlnode(const Type& v, avlnode\* par) {

val = v;

parent = par;

left = right = nullptr;

balance = 0;

}

};

template <class Type>

class AVLTree {

void LeftRotation(avlnode<Type>\*\* knot);

void RightRotation(avlnode<Type>\*\* knot);

void DoubleLeftRotation(avlnode<Type>\*\* knot);

void DoubleRightRotation(avlnode<Type>\*\* knot);

public:

avlnode<Type>\* root;

AVLTree();

~AVLTree();

vector<avlnode<Type>\*> GetAllKnots();

int GetSize();

int Height(avlnode<Type>\* knot);

void Balancing(avlnode<Type>\*\* knot);

avlnode<Type>\* Findmin(avlnode<Type>\* knot);

avlnode<Type>\* Findnext(avlnode<Type>\* knot);

void Add(const Type& v);

void Insert(avlnode<Type> \*\*knot, avlnode<Type> \*\*new\_node);

void Erase(avlnode<Type>\*\* knot, const Type& value);

void Erase(const Type& value);

void TotalErase(avlnode<Type>\*\* knot);

};

template<class Type>

AVLTree<Type>::AVLTree() {

root = NULL;

}

template<class Type>

AVLTree<Type>::~AVLTree() {

TotalErase(&root);

}

template<class Type>

vector<avlnode<Type>\*> AVLTree<Type>::GetAllKnots() {

if (root == nullptr)

vector<avlnode<Type>\*> r(0);

stack<avlnode<Type>\*> s, res;

s.push(root);

while (!s.empty()) {

auto tmp = s.top();

s.pop();

if (tmp != nullptr) {

res.push(tmp);

if (tmp->left != nullptr)

s.push(tmp->left);

if (tmp->right != nullptr)

s.push(tmp->right);

}

}

vector<avlnode<Type>\*> v(res.size());

while (!res.empty()) {

v[res.size() - 1] = res.top();

res.pop();

}

return v;

}

template<class Type>

int AVLTree<Type>::GetSize() {

return GetAllKnots().size();

}

template<class Type>

void AVLTree<Type>::LeftRotation(avlnode<Type>\*\* knot) {

avlnode<Type>\* tmp1 = (\*knot);

avlnode<Type>\* tmp2 = tmp1->right;

avlnode<Type>\* tmp3 = tmp2->left;

tmp1->right = tmp3;

if (tmp3 != nullptr)

tmp3->parent = tmp1;

tmp2->left = tmp1;

tmp2->parent = tmp1->parent;

tmp1->parent = tmp2;

Height(tmp1);

Height(tmp2);

(\*knot) = tmp2;

}

template<class Type>

void AVLTree<Type>::RightRotation(avlnode<Type>\*\* knot) {

avlnode<Type>\* tmp1 = (\*knot);

avlnode<Type>\* tmp2 = tmp1->left;

avlnode<Type>\* tmp3 = tmp2->right;

tmp1->left = tmp3;

if (tmp3 != nullptr)

tmp3->parent = tmp1;

tmp2->right = tmp1;

tmp2->parent = tmp1->parent;

tmp1->parent = tmp2;

Height(tmp1);

Height(tmp2);

(\*knot) = tmp2;

}

template<class Type>

void AVLTree<Type>::DoubleLeftRotation(avlnode<Type>\*\* knot) {

avlnode<Type>\* tmp1 = (\*knot);

avlnode<Type>\* tmp2 = tmp1->left;

avlnode<Type>\* tmp3 = tmp2->right;

avlnode<Type>\* tmp4 = tmp3->left;

avlnode<Type>\* tmp5 = tmp3->right;

tmp3->right = tmp1;

tmp3->left = tmp2;

tmp3->parent = tmp1->parent;

tmp1->parent = tmp3;

tmp2->parent = tmp3;

tmp2->right = tmp4;

tmp1->left = tmp5;

if (tmp4 != nullptr)

tmp4->parent = tmp2;

if (tmp5 != nullptr)

tmp5->parent = tmp1;

Height(tmp1);

Height(tmp2);

Height(tmp3);

(\*knot) = tmp3;

}

template<class Type>

void AVLTree<Type>::DoubleRightRotation(avlnode<Type>\*\* knot) {

avlnode<Type>\* tmp1 = (\*knot);

avlnode<Type>\* tmp2 = tmp1->left;

avlnode<Type>\* tmp3 = tmp2->right;

avlnode<Type>\* tmp4 = tmp3->left;

avlnode<Type>\* tmp5 = tmp3->right;

tmp3->right = tmp1;

tmp3->parent = tmp1->parent;

tmp1->parent = tmp3;

tmp3->left = tmp2;

tmp2->parent = tmp3;

tmp2->right = tmp4;

if (tmp4 != nullptr)

tmp4->parent = tmp2;

tmp1->left = tmp5;

if (tmp5 != nullptr)

tmp5->parent = tmp1;

Height(tmp1);

Height(tmp2);

Height(tmp3);

(\*knot) = tmp3;

}

template<class Type>

int AVLTree<Type>::Height(avlnode<Type>\* knot) {

if (knot == nullptr)

return -1;

int left = Height(knot->left);

int right = Height(knot->right);

knot->balance = left - right;

if (left > right)

return (left + 1);

else

return (right + 1);

}

template<class Type>

void AVLTree<Type>::Balancing(avlnode<Type>\*\* knot) {

if ((\*knot) == nullptr)

return;

Height((\*knot));

if ((\*knot)->balance == 2) {

if (((\*knot)->left)->balance > 0)

RightRotation(knot);

else

DoubleRightRotation(knot);

}

if ((\*knot)->balance == -2) {

if ((\*knot)->right->balance <= 0)

LeftRotation(knot);

else

DoubleLeftRotation(knot);

}

}

template<class Type>

avlnode<Type>\* AVLTree<Type>::Findmin(avlnode<Type>\* knot) {

avlnode<Type>\* tmp;

if (knot == nullptr)

return 0;

else

tmp = knot;

while (tmp->left != nullptr)

tmp = tmp->left;

return tmp;

}

template<class Type>

avlnode<Type>\* AVLTree<Type>::Findnext(avlnode<Type>\* knot) {

if (knot == nullptr)

throw logic\_error("Entered empty knot");

if (knot->right != nullptr)

return Findmin(knot->right);

while ((knot->parent != nullptr) && knot->parent->right == knot)

knot = knot->parent;

if (knot->parent == nullptr)

throw logic\_error("Next knot isn`t found");

return knot->parent;

}

template<class Type>

void AVLTree<Type>::Add(const Type& value) {

avlnode<Type>\* tmp = new avlnode<Type>(value, nullptr);

if (tmp == 0)

throw logic\_error("You don`t have a memory");

if (root == nullptr) {

root = tmp;

root->parent = nullptr;

return;

}

Insert(&root, &tmp);

}

template<class Type>

void AVLTree<Type>::Insert(avlnode<Type>\*\* knot, avlnode<Type>\*\* new\_node) {

if ((\*new\_node)->val < (\*knot)->val) {

if ((\*knot)->left == nullptr) {

(\*knot)->left = (\*new\_node);

(\*new\_node)->parent = (\*knot);

} else {

Insert(&((\*knot)->left), new\_node);

}

} else {

if ((\*knot)->right == nullptr) {

(\*knot)->right = (\*new\_node);

(\*new\_node)->parent = (\*knot);

} else {

Insert(&((\*knot)->right), new\_node);

}

}

Balancing(knot);

}

template<class Type>

void AVLTree<Type>::Erase(avlnode<Type>\*\* knot, const Type& value) {

if ((\*knot) == nullptr)

return;

if (value < (\*knot)->val) {

Erase(&((\*knot)->left), value);

} else if (value > (\*knot)->val) {

Erase(&((\*knot)->right), value);

} else {

if ((\*knot)->left == nullptr && (\*knot)->right == nullptr) {

avlnode<Type>\* tmp1 = (\*knot);

if ((\*knot)->parent != nullptr) {

if ((\*knot)->parent->left == (\*knot))

(\*knot)->parent->left = nullptr;

else

(\*knot)->parent->right = nullptr;

} else {

root = nullptr;

}

delete tmp1;

return;

}

if ((\*knot)->right == nullptr) {

avlnode<Type>\* tmp1 = (\*knot);

avlnode<Type>\* tmp2 = (\*knot)->left;

tmp2->parent = tmp1->parent;

if (tmp1->parent != nullptr) {

if (tmp1->parent->left == tmp1)

tmp1->parent->left = tmp2;

else

tmp1->parent->right = tmp2;

} else {

root = tmp2;

}

delete tmp1;

return;

}

if ((\*knot)->left == nullptr) {

avlnode<Type>\* tmp1 = (\*knot);

avlnode<Type>\* tmp2 = (\*knot)->right;

tmp2->parent = tmp1->parent;

if (tmp1->parent != nullptr) {

if (tmp1->parent->left == tmp1)

tmp1->parent->left = tmp2;

else

tmp1->parent->right = tmp2;

} else {

root = tmp2;

}

delete tmp1;

return;

} else {

avlnode<Type>\* tmp1 = Findnext((\*knot));

avlnode<Type>\* tmp2 = tmp1->parent;

if (tmp1->parent->left == tmp1)

tmp2->left = tmp1->right;

else

tmp2->right = tmp1->right;

tmp1->parent = (\*knot)->parent;

tmp1->left = (\*knot)->left;

tmp1->right = (\*knot)->right;

if (tmp1->left != nullptr)

tmp1->left->parent = tmp1;

if (tmp1->right != nullptr)

tmp1->right->parent = tmp1;

while (tmp2 != nullptr && tmp2 != tmp1) {

Balancing(&tmp2);

tmp2 = tmp2->parent;

}

avlnode<Type>\* temp = (\*knot);

(\*knot) = tmp1;

delete temp;

}

}

Balancing(knot);

}

template<class Type>

void AVLTree<Type>::Erase(const Type& value) {

if (root == nullptr)

throw logic\_error("Tree is empty");

Erase(&root, value);

}

template<class Type>

void AVLTree<Type>::TotalErase(avlnode<Type>\*\* knot) {

if ((\*knot) == nullptr)

return;

if ((\*knot)->left != nullptr)

TotalErase(&((\*knot)->left));

if ((\*knot)->right != nullptr)

TotalErase(&((\*knot)->right));

delete (\*knot);

(\*knot) = nullptr;

}

#endif // INCLUDE\_AVL\_TREE\_H\_

## Приложение 3 (файл five\_heap.h)

#ifndef INCLUDE\_FIVE\_HEAP\_H\_

#define INCLUDE\_FIVE\_HEAP\_H\_

#include <stdexcept>

const int d\_ = 5;

const int MAX\_SIZE = 1000;

using std::logic\_error;

template <class Type>

class FIVEheap{

int size;

int d;

public:

Type\* val;

FIVEheap();

~FIVEheap();

int GetSize() { return size; }

bool operator == (const FIVEheap<Type>& B);

bool operator != (const FIVEheap<Type>& B);

Type& operator[] (int i);

void Reduction(int i, const Type& con);

void Swap(int i, int j);

int MinChild(int i);

void Insert(const Type& val\_);

void Erase(int i);

void EraseMin();

void Immertion(int i); // down

void Emertion(int i); // up

void DoingHeaps();

bool IsFull() { return (size >= MAX\_SIZE); }

bool IsEmpty() { return size == 0; }

};

template<class Type>

FIVEheap<Type>::FIVEheap() {

d = d\_;

size = 0;

val = new Type[MAX\_SIZE];

}

template<class Type>

FIVEheap<Type>::~FIVEheap() {

delete[] val;

}

template<class Type>

bool FIVEheap<Type>::operator==(const FIVEheap<Type>& B) {

if (size != B.size)

return false;

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

if (val[i] != B.val[i])

return false;

return true;

}

template<class Type>

bool FIVEheap<Type>::operator!=(const FIVEheap<Type>& B) {

return !((\*this) == B);

}

template<class Type>

Type& FIVEheap<Type>::operator[](int i) {

if (i< 0 || i >= size)

throw logic\_error("Out of range");

return val[i];

}

template<class Type>

void FIVEheap<Type>::Reduction(int i, const Type & con) {

if (i< 0 || i >= size)

throw logic\_error("Out of range");

val[i] -= con;

Emertion(i);

}

template<class Type>

void FIVEheap<Type>::Swap(int i, int j) {

if ((i < 0) || (j < 0) || (i >= size) || (j >= size))

throw logic\_error("Out of range");

Type tmp = val[i];

val[i] = val[j];

val[j] = tmp;

}

template<class Type>

void FIVEheap<Type>::Insert(const Type& val\_) {

if (size >= MAX\_SIZE)

throw logic\_error("Out of range");

val[size] = val\_;

size++;

Emertion(size-1);

}

template<class Type>

void FIVEheap<Type>::Erase(int i) {

if (size ==0)

throw logic\_error("Heap is empty");

if (i< 0 || i >= size)

throw logic\_error("Out of range");

val[i] = val[size - 1];

size--;

if ((i != 0) && (val[i] < val[(i - 1) / d]))

Emertion(i);

else

Immertion(i);

}

template<class Type>

void FIVEheap<Type>::EraseMin() {

if (size == 0)

throw logic\_error("Heap is empty");

val[0] = val[size-1];

size--;

if (size != 0)

Immertion(0);

}

template<class Type>

int FIVEheap<Type>::MinChild(int i) {

if (i< 0 || i >= size)

throw logic\_error("Out of range");

if (d\*i + 1 >= size)

return 0;

int first = d\*i + 1;

Type min\_val = val[first];

int last = (i + 1) \*d;

if (last >= size)

last = size - 1;

for (int k = first + 1; k <= last; ++k)

if (val[k] < min\_val) {

min\_val = val[k];

first = k;

}

return first;

}

template<class Type>

void FIVEheap<Type>::Immertion(int i) {

if (i< 0 || i >= size)

throw logic\_error("Out of range");

int s = MinChild(i);

while ((s != 0) && (val[i] > val[s])) {

Swap(i, s);

i = s;

s = MinChild(i);

}

}

template<class Type>

void FIVEheap<Type>::Emertion(int i) {

if (i < 0 || i >= size)

throw logic\_error("Out of range");

int parent = (i - 1) / d;

while ((i != 0) && (val[parent] > val[i])) {

Swap(i, parent);

i = parent;

parent = (i - 1) / d;

}

}

template<class Type>

void FIVEheap<Type>::DoingHeaps() {

if (size > 0) {

for (int i = (size - 1) / d; size > 0; i++)

Immertion(i);

}

}

#endif // INCLUDE\_FIVE\_HEAP\_H\_

## Приложение 4 (файл priority\_queue.h)

#ifndef INCLUDE\_PRIORITY\_QUEUE\_H\_

#define INCLUDE\_PRIORITY\_QUEUE\_H\_

#include "avl\_tree.h"

#include "five\_heap.h"

template <class Type>

class Queue\_on\_heap {

FIVEheap<Type>\* heap;

public:

Queue\_on\_heap();

~Queue\_on\_heap();

int GetSize() { return heap->GetSize(); }

bool IsFull() { return heap->IsFull(); }

bool IsEmpty() { return heap->IsEmpty(); }

void Push(const Type& val);

void Pop();

Type Top();

};

template<class Type>

Queue\_on\_heap<Type>::Queue\_on\_heap() {

heap = new FIVEheap<Type>();

}

template<class Type>

Queue\_on\_heap<Type>::~Queue\_on\_heap() {

delete heap;

}

template<class Type>

void Queue\_on\_heap<Type>::Push(const Type & val) {

if (IsFull())

throw logic\_error("Heap is full");

heap->Insert(val);

}

template<class Type>

void Queue\_on\_heap<Type>::Pop() {

if (IsEmpty())

throw logic\_error("Heap is empty");

heap->EraseMin();

}

template<class Type>

Type Queue\_on\_heap<Type>::Top() {

if (IsEmpty())

throw logic\_error("Heap is empty");

return (\*heap)[0];

}

template <class Type>

class Queue\_on\_AVLTree {

AVLTree<Type>\* tree;

public:

Queue\_on\_AVLTree();

~Queue\_on\_AVLTree();

int GetSize() { return tree->GetSize(); }

bool IsFull();

bool IsEmpty() { return (tree->GetSize() == 0); }

void Push(const Type& val) { tree->Add(val); }

void Pop();

Type Top();

};

template<class Type>

Queue\_on\_AVLTree<Type>::Queue\_on\_AVLTree() {

tree = new AVLTree<Type>();

}

template<class Type>

Queue\_on\_AVLTree<Type>::~Queue\_on\_AVLTree() {

delete tree;

}

template<class Type>

bool Queue\_on\_AVLTree<Type>::IsFull() {

avlnode<Type>\* tmp = new avlnode<Type>();

if (tmp == nullptr)

return 1;

delete tmp;

return 0;

}

template<class Type>

void Queue\_on\_AVLTree<Type>::Pop() {

avlnode<Type>\* min = tree->Findmin(tree->root);

tree->Erase(min->val);

}

template<class Type>

Type Queue\_on\_AVLTree<Type>::Top() {

if (IsEmpty())

throw logic\_error("Heap is empty");

return (tree->Findmin(tree->root))->val;

}

#endif // INCLUDE\_PRIORITY\_QUEUE\_H\_

## Приложение 4 (файл dijkstra.h)

#ifndef INCLUDE\_DIJKSTRA\_H\_

#define INCLUDE\_DIJKSTRA\_H\_

#include "priority\_queue.h"

#include <iostream>

#include <vector>

#include <string>

#include <algorithm>

#include <set>

#include <utility>

using std::cin;

using std::cout;

using std::endl;

using std::set;

using std::vector;

using std::pair;

using std::make\_pair;

const int INF = 1000000;

vector<int> Dijkstra\_on\_heap(int s, int n, vector<vector<pair<int, int>>> g);

vector<int> Dijkstra\_on\_AVLTree(int s, int n, vector<vector<pair<int, int>>> g);

#endif // INCLUDE\_DIJKSTRA\_H\_

## Приложение 5 (файл dijkstra.cpp)

#include "dijkstra.h"

vector<int> Dijkstra\_on\_heap(int s, int n, vector<vector<pair<int, int>>> g) {

if ((s < 0) || (s > n) || (g.empty())) {

throw logic\_error("Errors in the start values");

}

vector<int> d(n, INF);

d[s] = 0;

Queue\_on\_heap<pair<int, int>> q;

q.Push(make\_pair(0, s));

while (!q.IsEmpty()) {

int v = q.Top().second;

int cur\_d = q.Top().first;

q.Pop();

if (cur\_d > d[v])

continue;

for (int j = 0; j < static\_cast<int>(g[v].size()); ++j) {

int to = g[v][j].first;

if (to < 0)

break;

int len = g[v][j].second;

if (d[v] + len < d[to]) {

d[to] = d[v] + len;

q.Push(make\_pair(d[to], to));

}

}

}

return d;

}

vector<int> Dijkstra\_on\_AVLTree(int s, int n, vector<vector<pair<int, int>>> g) {

if ((s < 0) || (s > n) || (g.empty())) {

throw logic\_error("Errors in the start values");

}

vector<int> d(n, INF);

d[s] = 0;

Queue\_on\_AVLTree<pair<int, int>> q;

q.Push(make\_pair(0, s));

while (!q.IsEmpty()) {

int v = q.Top().second;

int cur\_d = q.Top().first;

q.Pop();

if (cur\_d > d[v])

continue;

for (int j = 0; j < static\_cast<int>(g[v].size()); ++j) {

int to = g[v][j].first;

if (to < 0)

break;

int len = g[v][j].second;

if (d[v] + len < d[to]) {

d[to] = d[v] + len;

q.Push(make\_pair(d[to], to));

}

}

}

return d;

}

## Приложение 6 (файл test\_tree.cpp)

#include <gtest/gtest.h>

#include "avl\_tree.h"

TEST(AVLTree, Can\_create\_empty\_tree) {

EXPECT\_NO\_THROW(AVLTree<int> a);

}

TEST(AVLTree, Size\_of\_empty\_tree\_is\_zero) {

AVLTree<int> empty;

EXPECT\_EQ(0, empty.GetSize());

}

TEST(AVLTree, Can\_insert\_a\_key) {

AVLTree<int> a;

EXPECT\_NO\_THROW(a.Add(7));

}

TEST(AVLTree, LeftRotation\_works\_correct) {

AVLTree<int> a;

for (int i = 2; i < 10; ++i)

a.Add(i);

int res = a.root->balance;

EXPECT\_TRUE((res == -1) || (res == 0) || (res == 1));

a.Add(1);

res = a.root->balance;

EXPECT\_TRUE((res == -1) || (res == 0) || (res == 1));

}

TEST(AVLTree, RightRotation\_works\_correct) {

AVLTree<int> a;

for (int i = 2; i < 10; ++i)

a.Add(i);

int res = a.root->balance;

EXPECT\_TRUE((res == -1) || (res == 0) || (res == 1));

a.Add(10);

res = a.root->balance;

EXPECT\_TRUE((res == -1) || (res == 0) || (res == 1));

}

TEST(AVLTree, DoubleRightRotation\_works\_correct) {

AVLTree<int> a;

for (int i = 30; i <= 70; i=i+10)

a.Add(i);

a.Add(5);

int res = a.root->balance;

EXPECT\_TRUE((res == -1) || (res == 0) || (res == 1));

a.Add(20);

res = a.root->balance;

EXPECT\_TRUE((res == -1) || (res == 0) || (res == 1));

}

TEST(AVLTree, Single\_tree\_is\_balanced) {

AVLTree<int> a;

a.Add(7);

EXPECT\_TRUE(((a.root)->balance == -1) ||

((a.root)->balance == 0) || ((a.root)->balance == 0));

}

TEST(AVLTree, Can\_insert\_to\_tree) {

AVLTree<int> a;

a.Add(2);

a.Add(6);

a.Add(7);

avlnode<int>\* eight = new avlnode<int>(8, nullptr);

EXPECT\_NO\_THROW(a.Insert(&(a.root->left), &eight));

}

TEST(AVLTree, Adding\_increases\_size\_tree) {

AVLTree<int> a;

int tmp = a.GetSize();

a.Add(7);

EXPECT\_EQ(tmp + 1, a.GetSize());

}

TEST(AVLTree, Created\_tree\_is\_correct) {

AVLTree<int> a;

for (int i = 0; i < 10; ++i)

a.Add(i);

EXPECT\_EQ(a.root->val, 3);

EXPECT\_EQ(a.root->left->val, 1);

EXPECT\_EQ(a.root->right->val, 7);

EXPECT\_EQ(a.root->left->left->val, 0);

EXPECT\_EQ(a.root->left->right->val, 2);

EXPECT\_EQ(a.root->right->left->val, 5);

EXPECT\_EQ(a.root->right->right->val, 8);

EXPECT\_EQ(a.root->right->left->left->val, 4);

EXPECT\_EQ(a.root->right->left->right->val, 6);

EXPECT\_EQ(a.root->right->right->right->val, 9);

}

TEST(AVLTree, Created\_tree\_is\_correct\_check\_with\_func) {

AVLTree<int> a;

for (int i = 0; i < 10; ++i)

a.Add(i);

vector<int> v = {3, 7, 8, 9, 5, 6, 4, 1, 2, 0};

vector<avlnode<int>\*> knots = a.GetAllKnots();

for (int i = 0; i < 10; ++i)

EXPECT\_EQ(v[i], knots[i]->val);

}

TEST(AVLTree, Adding\_increases\_size\_tree\_2) {

AVLTree<int> a;

int tmp = a.GetSize();

for (int i = 0; i < 10; ++i)

a.Add(i);

EXPECT\_EQ(tmp + 10, a.GetSize());

}

TEST(AVLTree, Created\_tree\_is\_balanced) {

AVLTree<int> a;

for (int i = 0; i < 10; ++i)

a.Add(i);

int res = a.root->balance;

EXPECT\_TRUE((res ==-1) || (res == 0) || (res == 1));

res = a.root->left->balance;

EXPECT\_TRUE((res == -1) || (res == 0) || (res == 1));

res = a.root->right->right->balance;

EXPECT\_TRUE((res == -1) || (res == 0) || (res == 1));

}

TEST(AVLTree, Can\_erase\_by\_key) {

AVLTree<int> a;

for (int i = 0; i < 10; ++i)

a.Add(i);

EXPECT\_NO\_THROW(a.Erase(7));

}

TEST(AVLTree, Can\_erase\_by\_knot) {

AVLTree<int> a;

for (int i = 0; i < 10; ++i)

a.Add(i);

EXPECT\_NO\_THROW(a.Erase(a.root->balance));

}

TEST(AVLTree, Erasing\_decreases\_size\_tree\_) {

AVLTree<int> a;

for (int i = 0; i < 10; ++i)

a.Add(i);

int tmp = a.GetSize();

a.Erase(1);

a.Erase(4);

EXPECT\_EQ(tmp -2, a.GetSize());

}

TEST(AVLTree, Erase\_saves\_balance) {

AVLTree<int> a;

for (int i = 0; i < 10; ++i)

a.Add(i);

int res = a.root->balance;

EXPECT\_TRUE((res == -1) || (res == 0) || (res == 1));

res = a.root->left->balance;

EXPECT\_TRUE((res == -1) || (res == 0) || (res == 1));

res = a.root->right->right->balance;

EXPECT\_TRUE((res == -1) || (res == 0) || (res == 1));

a.Erase(3);

a.Erase(9);

EXPECT\_TRUE((res == -1) || (res == 0) || (res == 1));

res = a.root->left->balance;

EXPECT\_TRUE((res == -1) || (res == 0) || (res == 1));

}

## Приложение 7 (файл test\_heap.cpp)

#include <gtest/gtest.h>

#include "five\_heap.h"

TEST(FIVEheap, Can\_create\_empty\_heap) {

EXPECT\_NO\_THROW(FIVEheap<int> a);

}

TEST(FIVEheap, Return\_correct\_size) {

FIVEheap<int> a;

EXPECT\_EQ(0, a.GetSize());

}

TEST(FIVEheap, Can\_insert) {

FIVEheap<int> a;

EXPECT\_NO\_THROW(a.Insert(7));

}

TEST(FIVEheap, Adding\_increase\_size) {

FIVEheap<int> a;

int size1 = a.GetSize();

a.Insert(7);

int size2 = a.GetSize();

EXPECT\_EQ(size1 + 1, size2);

}

TEST(FIVEheap, Cant\_insert\_to\_full\_heap) {

FIVEheap<int> a;

for (int i = 0; i < 1000; i++)

a.Insert(i);

EXPECT\_ANY\_THROW(a.Insert(1001));

}

TEST(FIVEheap, Can\_compare\_equal\_heaps) {

FIVEheap<int> a;

FIVEheap<int> b;

for (int i = 0; i < 10; i++) {

a.Insert(i);

b.Insert(i);

}

EXPECT\_TRUE(a == b);

}

TEST(FIVEheap, Cant\_compare\_not\_equal\_heaps) {

FIVEheap<int> a;

FIVEheap<int> b;

for (int i = 0; i < 10; i++) {

a.Insert(i);

b.Insert(10-i);

}

EXPECT\_FALSE(a == b);

}

TEST(FIVEheap, Cant\_compare\_not\_equal\_heaps2) {

FIVEheap<int> a;

FIVEheap<int> b;

for (int i = 0; i < 10; i++) {

a.Insert(i);

b.Insert(10 - i);

}

EXPECT\_TRUE(a != b);

}

TEST(FIVEheap, Can\_get\_elem\_by\_index) {

FIVEheap<int> a;

a.Insert(7);

a.Insert(2);

a.Insert(9);

EXPECT\_EQ(9, a[2]);

}

TEST(FIVEheap, Square\_brackets\_cant\_work\_with\_neg\_index) {

FIVEheap<int> a;

for (int i = 0; i < 3; i++)

a.Insert(i);

EXPECT\_ANY\_THROW(a[-5]);

}

TEST(FIVEheap, Square\_brackets\_cant\_work\_with\_too\_much\_index) {

FIVEheap<int> a;

for (int i = 0; i < 3; i++)

a.Insert(i);

EXPECT\_ANY\_THROW(a[5]);

}

TEST(FIVEheap, Square\_brackets\_cant\_change\_size) {

FIVEheap<int> a;

for (int i = 0; i < 3; i++)

a.Insert(i);

int size1 = a.GetSize();

int size2 = a[2];

size2 = a.GetSize();

EXPECT\_EQ(size1, size2);

}

TEST(FIVEheap, MinChild\_cant\_work\_with\_neg\_index) {

FIVEheap<int> a;

for (int i = 0; i < 3; i++)

a.Insert(i);

EXPECT\_ANY\_THROW(a.MinChild(-10));

}

TEST(FIVEheap, MinChild\_cant\_work\_with\_too\_much\_index) {

FIVEheap<int> a;

for (int i = 0; i < 3; i++)

a.Insert(i);

EXPECT\_ANY\_THROW(a.MinChild(10));

}

TEST(FIVEheap, MinChild\_cant\_change\_size) {

FIVEheap<int> a;

a.Insert(4);

a.Insert(8);

a.Insert(3);

int size1 = a.GetSize();

int size2 = a.MinChild(2);

size2 = a.GetSize();

EXPECT\_EQ(size1, size2);

}

TEST(FIVEheap, Can\_correct\_swap) {

FIVEheap<int> a;

a.Insert(4);

a.Insert(8);

a.Insert(3);

a.Insert(12);

a.Swap(0, 3);

EXPECT\_EQ(a[0], 12);

EXPECT\_EQ(a[3], 3);

}

TEST(FIVEheap, Swap\_cant\_work\_with\_neg\_index) {

FIVEheap<int> a;

a.Insert(4);

a.Insert(8);

a.Insert(3);

a.Insert(12);

EXPECT\_ANY\_THROW(a.Swap(0, -3));

}

TEST(FIVEheap, Swap\_cant\_work\_with\_too\_much\_index) {

FIVEheap<int> a;

a.Insert(4);

a.Insert(8);

a.Insert(3);

a.Insert(12);

EXPECT\_ANY\_THROW(a.Swap(10, 2));

}

TEST(FIVEheap, Swap\_cant\_change\_size) {

FIVEheap<int> a;

a.Insert(4);

a.Insert(8);

a.Insert(3);

a.Insert(12);

int size1 = a.GetSize();

a.Swap(0, 1);

int size2 = a.GetSize();

EXPECT\_EQ(size1, size2);

}

TEST(FIVEheap, Can\_correctly\_reduction) {

FIVEheap<int> a;

a.Insert(7);

a.Reduction(0, -1);

EXPECT\_EQ(a[0], 8);

}

TEST(FIVEheap, Reduction\_cant\_work\_with\_neg\_index) {

FIVEheap<int> a;

for (int i = 0; i < 3; i++)

a.Insert(i);

EXPECT\_ANY\_THROW(a.Reduction(-2, 5));

}

TEST(FIVEheap, Reduction\_cant\_work\_with\_too\_much\_index) {

FIVEheap<int> a;

for (int i = 0; i < 3; i++)

a.Insert(i);

EXPECT\_ANY\_THROW(a.Reduction(5, 5));

}

TEST(FIVEheap, Reduction\_cant\_change\_size) {

FIVEheap<int> a;

for (int i = 0; i < 3; i++)

a.Insert(i);

int size1 = a.GetSize();

a.Reduction(2, 4);

int size2 = a.GetSize();

EXPECT\_EQ(size1, size2);

}

TEST(FIVEheap, Immertion\_cant\_work\_with\_neg\_index) {

FIVEheap<int> a;

for (int i = 0; i < 3; i++)

a.Insert(i);

EXPECT\_ANY\_THROW(a.Immertion(-10));

}

TEST(FIVEheap, Immertion\_cant\_work\_with\_too\_much\_index) {

FIVEheap<int> a;

for (int i = 0; i < 3; i++)

a.Insert(i);

EXPECT\_ANY\_THROW(a.Immertion(10));

}

TEST(FIVEheap, Immertion\_cant\_change\_size) {

FIVEheap<int> a;

a.Insert(4);

a.Insert(8);

a.Insert(3);

int size1 = a.GetSize();

a.Immertion(0);

int size2 = a.GetSize();

EXPECT\_EQ(size1, size2);

}

TEST(FIVEheap, Emertion\_cant\_work\_with\_neg\_index) {

FIVEheap<int> a;

for (int i = 0; i < 3; i++)

a.Insert(i);

EXPECT\_ANY\_THROW(a.Emertion(-10));

}

TEST(FIVEheap, Emertion\_cant\_work\_with\_too\_much\_index) {

FIVEheap<int> a;

for (int i = 0; i < 3; i++)

a.Insert(i);

EXPECT\_ANY\_THROW(a.Emertion(10));

}

TEST(FIVEheap, Emertion\_cant\_change\_size) {

FIVEheap<int> a;

a.Insert(4);

a.Insert(8);

a.Insert(3);

int size1 = a.GetSize();

a.Emertion(0);

int size2 = a.GetSize();

EXPECT\_EQ(size1, size2);

}

TEST(FIVEheap, Can\_erase) {

FIVEheap<int> a;

for (int i = 0; i < 10; i++)

a.Insert(i);

a.Erase(0);

a.Erase(1);

a.Erase(4);

int mas[6] = {1, 7, 2, 3, 8, 5 };

for (int i = 0; i < 6; i++)

EXPECT\_EQ(a[i], mas[i]);

}

TEST(FIVEheap, Cant\_erase\_from\_empty\_heap) {

FIVEheap<int> a;

EXPECT\_ANY\_THROW(a.Erase(0));

}

TEST(FIVEheap, Cant\_erase\_min\_from\_empty\_heap) {

FIVEheap<int> a;

EXPECT\_ANY\_THROW(a.EraseMin());

}

TEST(FIVEheap, Erasing\_decreases\_size) {

FIVEheap<int> a;

for (int i = 0; i < 10; i++)

a.Insert(i);

int size1 = a.GetSize();

a.Erase(0);

a.Erase(1);

a.Erase(4);

int size2 = a.GetSize();

EXPECT\_EQ(size1 - 3, size2);

}

TEST(FIVEheap, ErasingMin\_decreases\_size) {

FIVEheap<int> a;

for (int i = 0; i < 10; i++)

a.Insert(i);

int size1 = a.GetSize();

a.EraseMin();

int size2 = a.GetSize();

EXPECT\_EQ(size1 - 1, size2);

}

TEST(FIVEheap, Erasing\_cant\_work\_with\_neg\_index) {

FIVEheap<int> a;

for (int i = 0; i < 10; i++)

a.Insert(i);

a.Erase(0);

EXPECT\_ANY\_THROW(a.Erase(-4));

}

TEST(FIVEheap, Erace\_cant\_work\_with\_too\_much\_index) {

FIVEheap<int> a;

for (int i = 0; i < 10; i++)

a.Insert(i);

a.Erase(0);

EXPECT\_ANY\_THROW(a.Erase(20));

}

TEST(FIVEheap, IsFull\_works\_correctly) {

FIVEheap<int> a;

for (int i = 0; i < 1000; i++)

a.Insert(i);

EXPECT\_TRUE(a.IsFull());

}

TEST(FIVEheap, IsEmpty\_works\_correctly) {

FIVEheap<int> a;

EXPECT\_TRUE(a.IsEmpty());

}

## Приложение 8 (файл test\_queue.cpp)

#include <gtest/gtest.h>

#include "priority\_queue.h"

/\*on Heap\*/

TEST(QUEUEonHeap, Can\_create\_empty\_queue) {

EXPECT\_NO\_THROW(Queue\_on\_heap<int> a);

}

TEST(QUEUEonHeap, Is\_Empty\_works\_correctly) {

Queue\_on\_heap<int> a;

EXPECT\_TRUE(a.IsEmpty());

}

TEST(QUEUEonHeap, Size\_of\_empty\_queue\_is\_zero) {

Queue\_on\_heap<int> a;

EXPECT\_EQ(0, a.GetSize());

}

TEST(QUEUEonHeap, Can\_push) {

Queue\_on\_heap<int> a;

EXPECT\_NO\_THROW(a.Push(7));

}

TEST(QUEUEonHeap, Pushing\_increase\_size) {

Queue\_on\_heap<int> a;

int size1 = a.GetSize();

a.Push(7);

a.Push(3);

int size2 = a.GetSize();

EXPECT\_EQ(size1 + 2, size2);

}

TEST(QUEUEonHeap, Is\_Full\_works\_correctly) {

Queue\_on\_heap<int> a;

for (int i = 0; i < 1000; ++i)

a.Push(i);

EXPECT\_TRUE(a.IsFull());

}

TEST(QUEUEonHeap, Cant\_push\_to\_full\_queue) {

Queue\_on\_heap<int> a;

for (int i = 0; i < 1000; ++i)

a.Push(i);

EXPECT\_ANY\_THROW(a.Push(1001));

}

TEST(QUEUEonHeap, Can\_pop) {

Queue\_on\_heap<int> a;

for (int i = 0; i < 10; ++i)

a.Push(i);

EXPECT\_NO\_THROW(a.Pop());

}

TEST(QUEUEonHeap, Cant\_pop\_from\_empty\_queue) {

Queue\_on\_heap<int> a;

EXPECT\_ANY\_THROW(a.Pop());

}

TEST(QUEUEonHeap, Popping\_decrease\_size) {

Queue\_on\_heap<int> a;

for (int i = 0; i < 10; ++i)

a.Push(i);

int size1 = a.GetSize();

a.Pop();

int size2 = a.GetSize();

EXPECT\_EQ(size1 - 1, size2);

}

TEST(QUEUEonHeap, Top\_works\_correctly) {

Queue\_on\_heap<int> a;

for (int i = 0; i < 10; ++i)

a.Push(i);

EXPECT\_EQ(0, a.Top());

}

TEST(QUEUEonHeap, Cant\_Top\_from\_empty\_queue) {

Queue\_on\_heap<int> a;

EXPECT\_ANY\_THROW(a.Top());

}

TEST(QUEUEonHeap, After\_pop\_Top\_is\_still\_correct) {

Queue\_on\_heap<int> a;

for (int i = 0; i < 10; ++i)

a.Push(i);

a.Pop();

EXPECT\_EQ(1, a.Top());

}

TEST(QUEUEonHeap, Top\_plus\_pop\_works\_correctly) {

Queue\_on\_heap<int> a;

for (int i = 0; i < 10; ++i)

a.Push(i);

int mas[10];

for (int i = 0; i < 10; ++i) {

mas[i] = a.Top();

a.Pop();

}

int exp[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

for (int i = 0; i < 10; ++i)

EXPECT\_EQ(mas[i], exp[i]);

}

TEST(QUEUEonHeap, Pushing\_update\_queue) {

Queue\_on\_heap<int> a;

for (int i = 1; i < 10; ++i)

a.Push(i);

EXPECT\_EQ(1, a.Top());

a.Push(0);

EXPECT\_EQ(0, a.Top());

}

TEST(QUEUEonHeap, After\_total\_pop\_heap\_is\_empty) {

Queue\_on\_heap<int> a;

for (int i = 0; i < 100; ++i)

a.Push(i - i/2);

for (int i = 0; i < 100; ++i)

a.Pop();

EXPECT\_TRUE(a.IsEmpty());

}

/\*on AVL-tree\*/

TEST(QUEUEonAVLtree, Can\_create\_empty\_queue) {

EXPECT\_NO\_THROW(Queue\_on\_AVLTree<int> a);

}

TEST(QUEUEonAVLtree, Is\_Empty\_works\_correctly) {

Queue\_on\_AVLTree<int> a;

EXPECT\_TRUE(a.IsEmpty());

}

TEST(QUEUEonAVLtree, Size\_of\_empty\_queue\_is\_zero) {

Queue\_on\_AVLTree<int> a;

EXPECT\_EQ(0, a.GetSize());

}

TEST(QUEUEonAVLtree, Can\_push) {

Queue\_on\_AVLTree<int> a;

EXPECT\_NO\_THROW(a.Push(7));

}

TEST(QUEUEonAVLtree, Pushing\_increase\_size) {

Queue\_on\_AVLTree<int> a;

int size1 = a.GetSize();

a.Push(7);

a.Push(3);

int size2 = a.GetSize();

EXPECT\_EQ(size1 + 2, size2);

}

TEST(QUEUEonAVLtree, Is\_Full\_works\_correctly) {

Queue\_on\_AVLTree<int> a;

for (int i = 0; i < 1000; ++i)

a.Push(i);

EXPECT\_TRUE(!a.IsFull());

}

TEST(QUEUEonAVLtree, Can\_pop) {

Queue\_on\_AVLTree<int> a;

for (int i = 0; i < 10; ++i)

a.Push(i);

EXPECT\_NO\_THROW(a.Pop());

}

TEST(QUEUEonAVLtree, Cant\_pop\_from\_empty\_queue) {

Queue\_on\_AVLTree<int> a;

EXPECT\_ANY\_THROW(a.Pop());

}

TEST(QUEUEonAVLtree, Popping\_decrease\_size) {

Queue\_on\_AVLTree<int> a;

for (int i = 0; i < 10; ++i)

a.Push(i);

int size1 = a.GetSize();

a.Pop();

int size2 = a.GetSize();

EXPECT\_EQ(size1 - 1, size2);

}

TEST(QUEUEonAVLtree, Top\_works\_correctly) {

Queue\_on\_AVLTree<int> a;

for (int i = 0; i < 10; ++i)

a.Push(i);

EXPECT\_EQ(0, a.Top());

}

TEST(QUEUEonAVLtree, Cant\_Top\_from\_empty\_queue) {

Queue\_on\_AVLTree<int> a;

EXPECT\_ANY\_THROW(a.Top());

}

TEST(QUEUEonAVLtree, After\_pop\_Top\_is\_still\_correct) {

Queue\_on\_AVLTree<int> a;

for (int i = 0; i < 10; ++i)

a.Push(i);

a.Pop();

EXPECT\_EQ(1, a.Top());

}

TEST(QUEUEonAVLtree, Top\_plus\_pop\_works\_correctly) {

Queue\_on\_AVLTree<int> a;

for (int i = 0; i < 10; ++i)

a.Push(i);

int mas[10];

for (int i = 0; i < 10; ++i) {

mas[i] = a.Top();

a.Pop();

}

int exp[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

for (int i = 0; i < 10; ++i)

EXPECT\_EQ(mas[i], exp[i]);

}

TEST(QUEUEonAVLtree, Pushing\_update\_queue) {

Queue\_on\_AVLTree<int> a;

for (int i = 1; i < 10; ++i)

a.Push(i);

EXPECT\_EQ(1, a.Top());

a.Push(0);

EXPECT\_EQ(0, a.Top());

}

TEST(QUEUEonAVLtree, After\_total\_pop\_heap\_is\_empty) {

Queue\_on\_AVLTree<int> a;

for (int i = 0; i < 100; ++i)

a.Push(i - i / 2);

for (int i = 0; i < 100; ++i)

a.Pop();

EXPECT\_TRUE(a.IsEmpty());

}

## Приложение 9 (файл test\_dijkstra.cpp)

#include <gtest/gtest.h>

#include <vector>

#include <utility>

#include "dijkstra.h"

/\*on Heap\*/

TEST(DIJKSTRAonHeap, Can\_find\_way) {

vector< vector< pair<int, int> > > g =

{ { make\_pair(2, 8) },

{ make\_pair(3, 9) },

{ make\_pair(1, 3), make\_pair(3, 2) },

{ make\_pair(4, 7) },

{ make\_pair(-1, -1) } };

vector<int> res = Dijkstra\_on\_heap(0, g.size(), g);

EXPECT\_EQ(res[1], 11);

EXPECT\_EQ(res[2], 8);

EXPECT\_EQ(res[3], 10);

EXPECT\_EQ(res[4], 17);

}

TEST(DIJKSTRAonHeap, Cant\_use\_negative\_start) {

vector< vector< pair<int, int> > > g =

{ { make\_pair(2, 8) },

{ make\_pair(3, 9) },

{ make\_pair(1, 3), make\_pair(3, 2) },

{ make\_pair(-1, -1) } };

EXPECT\_ANY\_THROW(Dijkstra\_on\_heap(-10, static\_cast<int>(g.size()), g));

}

TEST(DIJKSTRAonHeap, Cant\_use\_too\_big\_start) {

vector< vector< pair<int, int> > > g =

{ { make\_pair(2, 8) },

{ make\_pair(3, 9) },

{ make\_pair(1, 3), make\_pair(3, 2) },

{ make\_pair(-1, -1) } };

EXPECT\_ANY\_THROW(Dijkstra\_on\_heap(10, static\_cast<int>(g.size()), g));

}

TEST(DIJKSTRAonHeap, Cant\_use\_empty\_graph) {

vector< vector< pair<int, int> > > g;

EXPECT\_ANY\_THROW(Dijkstra\_on\_heap(0, static\_cast<int>(g.size()), g));

}

TEST(DIJKSTRAonHeap, Can\_use\_one\_vertex\_graph) {

vector< vector< pair<int, int> > > g(1);

vector<int> res = Dijkstra\_on\_heap(0, static\_cast<int>(g.size()), g);

EXPECT\_EQ(res[0], 0);

}

TEST(DIJKSTRAonHeap, Can\_use\_disconnected\_graph) {

vector < vector < pair<int, int> > > g(3);

g[0].push\_back(pair<int, int>(2, 5));

vector <int> res = Dijkstra\_on\_heap(0, static\_cast<int>(g.size()), g);

EXPECT\_EQ(res[0], 0);

EXPECT\_EQ(res[1], INF);

EXPECT\_EQ(res[2], 5);

}

TEST(DIJKSTRAonHeap, Can\_find\_min\_way) {

vector< vector< pair<int, int> > > g =

{ { make\_pair(2, 8) },

{ make\_pair(3, 9), make\_pair(4, 20) },

{ make\_pair(3, 15), make\_pair(1, 3) },

{ make\_pair(4, 7) },

{ make\_pair(-1, -1) } };

vector<int> res = Dijkstra\_on\_heap(0, static\_cast<int>(g.size()), g);

EXPECT\_EQ(res[3], 20);

EXPECT\_EQ(res[4], 27);

}

/\*on AVL-tree\*/

TEST(DIJKSTRAonAVLTree, Can\_find\_way) {

vector< vector< pair<int, int> > > g =

{ { make\_pair(2, 8) },

{ make\_pair(3, 9) },

{ make\_pair(1, 3), make\_pair(3, 2) },

{ make\_pair(4, 7) },

{ make\_pair(-1, -1) } };

vector<int> res = Dijkstra\_on\_AVLTree(0, g.size(), g);

EXPECT\_EQ(res[1], 11);

EXPECT\_EQ(res[2], 8);

EXPECT\_EQ(res[3], 10);

EXPECT\_EQ(res[4], 17);

}

TEST(DIJKSTRAonAVLTree, Cant\_use\_negative\_start) {

vector< vector< pair<int, int> > > g =

{ { make\_pair(2, 8) },

{ make\_pair(3, 9) },

{ make\_pair(1, 3), make\_pair(3, 2) },

{ make\_pair(-1, -1) } };

EXPECT\_ANY\_THROW(Dijkstra\_on\_AVLTree(-10, static\_cast<int>(g.size()), g));

}

TEST(DIJKSTRAonAVLTree, Cant\_use\_too\_big\_start) {

vector< vector< pair<int, int> > > g =

{ { make\_pair(2, 8) },

{ make\_pair(3, 9) },

{ make\_pair(1, 3), make\_pair(3, 2) },

{ make\_pair(-1, -1) } };

EXPECT\_ANY\_THROW(Dijkstra\_on\_AVLTree(10, static\_cast<int>(g.size()), g));

}

TEST(DIJKSTRAonAVLTree, Cant\_use\_empty\_graph) {

vector< vector< pair<int, int> > > g;

EXPECT\_ANY\_THROW(Dijkstra\_on\_AVLTree(0, static\_cast<int>(g.size()), g));

}

TEST(DIJKSTRAonAVLTree, Can\_use\_one\_vertex\_graph) {

vector< vector< pair<int, int> > > g(1);

vector<int> res = Dijkstra\_on\_AVLTree(0, static\_cast<int>(g.size()), g);

EXPECT\_EQ(res[0], 0);

}

TEST(DIJKSTRAonAVLTree, Can\_use\_disconnected\_graph) {

vector < vector < pair<int, int> > > g(3);

g[0].push\_back(pair<int, int>(2, 5));

vector <int> res = Dijkstra\_on\_AVLTree(0, static\_cast<int>(g.size()), g);

EXPECT\_EQ(res[0], 0);

EXPECT\_EQ(res[1], INF);

EXPECT\_EQ(res[2], 5);

}

TEST(DIJKSTRAonAVLTree, Can\_find\_min\_way) {

vector< vector< pair<int, int> > > g =

{ { make\_pair(2, 8) },

{ make\_pair(3, 9), make\_pair(4, 20) },

{ make\_pair(3, 15), make\_pair(1, 3) },

{ make\_pair(4, 7) },

{ make\_pair(-1, -1) } };

vector<int> res = Dijkstra\_on\_AVLTree(0, static\_cast<int>(g.size()), g);

EXPECT\_EQ(res[3], 20);

EXPECT\_EQ(res[4], 27);

}

## Приложение 10 (файл main\_test.cpp)

#include <gtest/gtest.h>

int main(int ac, char\* av[]) {

testing::InitGoogleTest(&ac, av);

return RUN\_ALL\_TESTS();

}