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 поиска кратчайшего пути**

**с использованием приоритетной очереди**

Выполнил:

студент ИИТММ

группы 0823-1  
Ермолаев М.А.

Проверил:

к.т.н., ассистент каф. ПрИнж ИИТММ

Сиднев А.А.

Нижний Новгород

2017 г.

**Содержание**

[Введение 3](#_Toc484197352)

[Постановка задачи 4](#_Toc484197353)

[Руководство пользователя 5](#_Toc484197354)

[Руководство программиста 6](#_Toc484197355)

[Описание структуры программы 6](#_Toc484197356)

[Описание структур данных 7](#_Toc484197357)

[Описание алгоритмов 9](#_Toc484197358)

[Алгоритмы методов класса RB\_tree (красно-чёрных деревьев) 9](#_Toc484197359)

[Алгоритмы методов класса Six\_Heap (6-кучи) 14](#_Toc484197360)

[Алгоритм Дейкстры 15](#_Toc484197361)

[Заключение 16](#_Toc484197362)

[Литература 17](#_Toc484197363)

# **Введение**

**Алгоритм Дейкстры** назван в честь голландского ученого Эдсгера Дейкстры (Edsger Dijkstra). Алгоритм был предложен в 1959 году для нахождения кратчайших путей от одной вершины до всех остальных в ориентированном взвешенном графе, при условии, что все ребра в графе имеют неотрицательные веса (где под весом имеется в виду расстояние от одной вершины графа к другой).

# 

# **Постановка задачи**

Необходимо реализовать:

1. Алгоритм Дейкстры поиска кратчайшего пути с использованием приоритетной очереди. Приоритетную очередь следует реализовать следующими способами:

- c использованием красно-черных деревьев

- c использованием 6-кучи

2. Ввод данных из файла

3. Тесты, проверяющие работоспособность программы

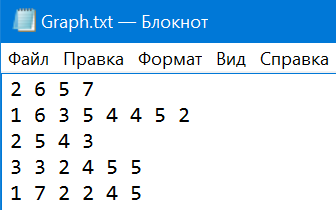
# 

# **Руководство пользователя**

Для работы с программой необходимо создать текстовый документ, который следует заполнить по следующим правилам:

1. Каждая строка описывает свою вершину графа (то есть первая строка описывает вершину с номером 1, вторая строка - вершину с номером 2 и т.д.)
2. Первое число – вершина, смежная с данной, второе число – вес ребра, соединяющего эти вершины
3. Можно вводить в одну строку несколько «вершин»

*Контрольный пример:*



После запуска программы необходимо ввести в консоль следующие данные:

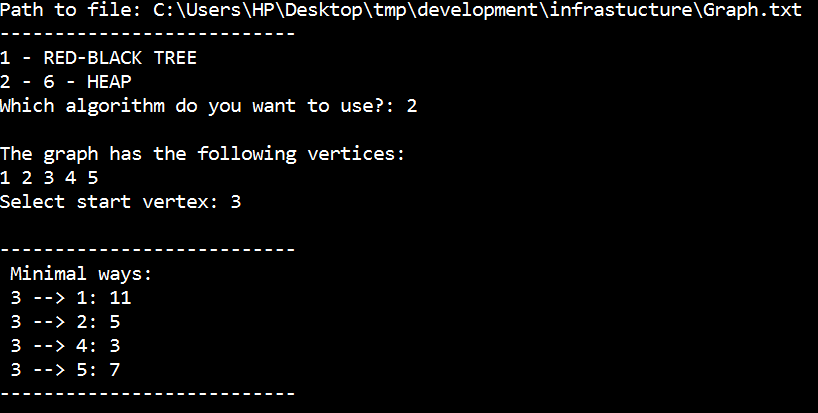
- Путь к ранее созданному текстовому файлу с описанием графа

- Алгоритм, с помощью которого будет реализована приоритетная очередь

- Стартовую вершины

Результат работы программы – таблица кратчайших путей от заданной вершины до всех остальных

*Контрольный пример:*



# **Руководство программиста**

## **Описание структуры программы**

Заголовочные файлы:

* **RB\_tree.h** – содержит описание структуры *RB\_node* звеньев красно-чёрного дерева, шаблонного класса *RB\_tree* и реализацию его методов
* **6-Heap.h** – содержит описание класса *Six\_Heap* 6-кучи и реализацию его методов
* **PriorityQueue.h** – содержит описание классов *PriorityQueue\_on\_HEAP* и *PriorityQueue\_on\_RBT*
* **Dijkstra.h** – содержит прототипы функций *Dijkstra\_Six\_Heap* и *Dijkstra\_RB\_tree*

Файлы исходного кода:

* **Digkstra.cpp** - содержит реализацию функций *Dijkstra\_Six\_Heap* и *Dijkstra\_RB\_tree*
* **test\_RB\_tree.cpp** – cодержит тесты, проверяющие корректность работы методов класса *RB\_tree*
* **test\_6-Heap.cpp -** cодержит тесты, проверяющие корректность работы методов класса *6-Heap*
* **test\_Dijkstra.cpp -** cодержит тесты, проверяющие корректность работы функций *Dijkstra\_Six\_Heap* и *Dijkstra\_RB\_tree*

- **main.cpp** – содержит интерфейс работы с пользователем

## **Описание структур данных**

- Структура **RB\_node**

struct RB\_node {

T value;

bool color;

struct RB\_node \*left;

struct RB\_node \*right;

struct RB\_node \*parent;

RB\_node<T>(T v, bool c, RB\_node<T>\* l, RB\_node<T>\* r, RB\_node<T>\* p) {

value = v;

color = c;

left = l;

right = r;

parent = p;

}

};

- Класс **RB\_tree**

template <typename T>  
class RB\_tree {

private:

void Rotate\_Left(RB\_node<T> \*x);

void Rotate\_Right(RB\_node<T> \*x);

void InsertFixup(RB\_node<T> \*x);

void DeleteFixup(RB\_node<T> \*x);

public:

RB\_tree();

~RB\_tree();

void Insert(T x);

void Delete(T x);

void Delete(RB\_node<T> \*x);

void DeleteMin();

RB\_node<T> \*root;

RB\_node<T>\* Find(T x);

T GetMin();

bool IsEmpty() {

if (root == reinterpret\_cast<RB\_node<T>\*>(NIL) || root == nullptr)

return true;

else

return false;

}

};

- Класс **Six-Heap**

template <typename T>

class Six\_Heap {

private:

vector<T> array;

void ShiftUp(int index);

void ShiftDown();

int MinSon(int index\_parent);

public:

void InsertElem(T e);

void DeleteMin();

T GetMin();

T operator[](int index) { return array[index]; }

bool IsEmpty() { return array.empty(); }

int GetSize() { return array.size(); }

};

- Классы **PriorityQueue\_on\_HEAP** и **PriorityQueue\_on\_RBT**

template <typename T>

class PriorityQueue\_on\_HEAP: public Six\_Heap<T> {

public:

};

template <typename T>

class PriorityQueue\_on\_RBT: public RB\_tree<T> {

public:

};

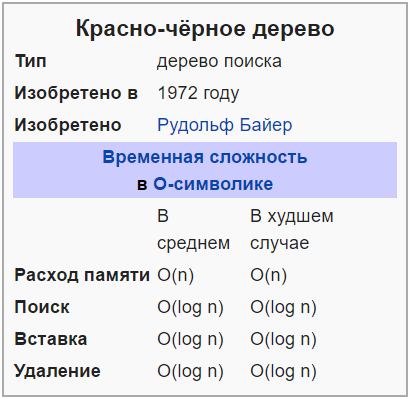
## 

## **Описание алгоритмов**

### **Алгоритмы методов класса RB\_tree (красно-чёрные деревья)**

**Теория:**

**Красно-чёрное дерево** – это бинарное дерево поиска, для которого выполняются красно-чёрные свойства (red-black properties):

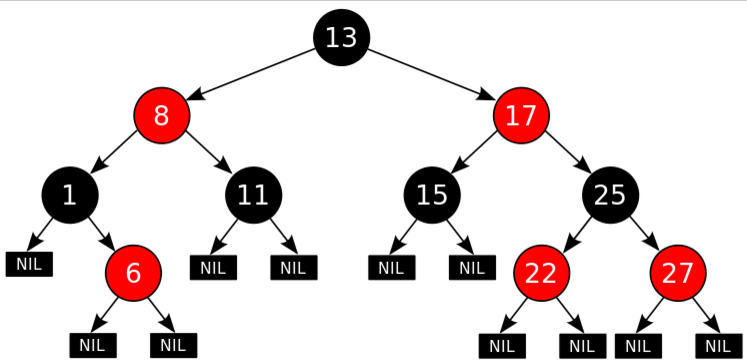
1. Каждый узел является красным или черным
2. Корень дерева является черным
3. Каждый лист дерева (NIL) является черным
4. У красного узла оба дочерних узла – черные
5. У любого узла все пути от него до листьев,

являющихся его потомками, содержат

одинаковое количество черных узлов

Данная структура гарантирует логарифмический рост   
высоты дерева от числа узлов и быстро выполняющиеся   
основные операции дерева поиска: добавление, удаление   
и поиск узла.

*Пример красно-чёрного дерева:*



**Операции:**

#### *Вставка нового элемента по ключу key*

1. Если дерево пусто, то под переменную root выделяем память. Инициализируем поля: ключ равен key, цвет – чёрный, сыны - NIL- листы, родитель - nullptr, иначе выполняется шаг 2
2. Создаем переменную current типа указатель на вершину дерева и присваиваем ей значение указателя на корень.
3. Создаем переменную parent равную nullptr.
4. Пока current не равен NIL:   
   Если значение v меньше, чем значение ключа переменной current, то присвоить current значение указателя на левого потомка, иначе присвоить current значение указателя на правого потомка
5. Создаем переменную типа указатель на новую вершину newnode: ключ равен v, цвет – красный, сыны - NIL- листы, родитель - parent
6. Если parent равен nullptr, то переменной root присваиваем значение newnode и выполняем шаг 9, иначе выполняется шаг 8
7. Если значение ключа newnode меньше значения ключа parent, то левым сыном parent становится newnode, иначе правым сыном parent становится newnode
8. Выполняется балансировка дерева после вставки

#### *Балансировка после вставки*

Пусть x – указатель на новый элемент, вставленный в дерево.  
Пока x не корень и имеет черный цвет:

* Если родитель х – правый потомок:

Назовём дядей (uncle) потомка от прародителя x

Пусть uncle – правый потомок родителя от родителя x

1. Eсли uncle – красный:
2. перекрашиваем uncle и родителя x - в черный
3. перекрашиваем правого прародителя x - в красный
4. присваиваем х значение указателя на прародителя x
5. Иначе, если x – правый потомок:
6. x присваиваем значение указателя на родителя х
7. Выполняем левый поворот узла х
8. Перекрашиваем родителя х в черный
9. Перекрашиваем родителя от родителя х в красный
10. Выполняем правый поворот узла прародителя х

* Если родитель х - левый потомок, то выполняется аналогично вышеописанному блоку с точностью до перестановки правого и левого поворотов.

***Повороты:***

#### *Правый поворот*

Пусть x – узел, поворот которого необходимо осуществить, y – левый потомок x.

1. Присвоить указателю на левого потомка x значение указателя на правого потомка y и в случае, если он не равен NIL, присвоить его указателю на родителя значение x
2. Присвоить указателю на родителя y значение указателя на родителя x
3. Если указатель на родителя x не равен nullptr:

* Если x – левый потомок своего родителя, присвоить указателю на левого потомка родителя x значение y
* Если x – правый потомок своего родителя, присвоить указателю на правого потомка родителя x значение y

1. Если указатель на родителя x не равен nullptr, присвоить указателю на корень значение y
2. Присвоить указателю на правого потомка y значение x
3. Присвоить указателю на родителя x значение y

#### *Левый поворот*

Пусть x – узел, поворот которого необходимо осуществить, а y – правый потомок x.

1. Присвоить указателю на правого потомка x значение указателя на левого потомка y и в случае, если он не равен NIL, присвоить его указателю на родителя значение x
2. Присвоить указателю на родителя y значение указателя на родителя x
3. Если указатель на родителя x не равен nullptr:

* Если x – левый потомок своего родителя, присвоить указателю на левого потомка родителя x значение y
* Если x – правый потомок своего родителя, присвоить указателю на правого потомка родителя x значение y

1. Если указатель на родителя x не равен nullptr, присвоить указателю на корень значение y
2. Присвоить указателю на левого потомка y значение x

**Удаление:**

#### *Удаление узла по значению ключа v*

1. Если дерево не пусто, то выполняется:

* Создаем переменную х типа указатель на узел и инициализируем ее возвращаемым значением функции поиска узла по заданному ключу v
* Выполняем удаление узла по адресу х

1. Иначе исключение (дерево пусто).

#### *Удаление минимального элемента*

1. Ищем значение адреса указателя на минимальный элемент в дереве
2. Выполняем удаление по адресу найденного узла

#### *Удаление узла node по адресу*

Если node не равно NIL – лист и не равно nullptr, то выполняем:

1. Если правый или левый потомки node - NIL – листы, то переменной у присваиваем значение указателя на node, иначе переменной у присваиваем значение указателя на левого потомка node
2. Пока левый потомок не равен NIL – листу, двигаемся по указателю влево
3. Если левый потомок у не равен NIL – листу, то переменной х присваиваем значение указателя на левого потомка, иначе присваиваем переменной х значение указателя правого потомка
4. Присваиваем родителю х значение родителя у
5. Если родитель у не nullptr и у - левый потомок, то значению у присваиваем значение х
6. Если родитель у не nullptr и у - правый потомок, то значению у присваиваем значение х
7. Если у не равен node, то значению ключа у присваиваем значение ключа node
8. Если узел у – красный, то выполняем балансировку после удаления
9. Если у равен root, то переменной root присваиваем значение nullptr, иначе удаляем у

#### *Поиск элемента по ключу key*

1. Если дерево не пусто, то создаем переменную current типа указатель на вершину дерева и присваиваем ей значение указателя на корень дерева
2. Пока current не равен NIL, выполняется:

2.1. Если ключ вершины current равен искомому ключу, вернуть current

2.2. Если искомый ключ меньше ключа вершины current, присвоить current значение указателя на левого потомка. Перейти к пункту 2.

2.3. Если искомый ключ больше ключа вершины current, присвоить current значение указателя на правого потомка. Перейти к пункту 2.

1. Вызвать исключение (т.е. если current равно NIL).

#### *Балансировка после удаления*

Пусть х - узел, для которого вызывается балансировка

1. Если х – не корень и х – не NIL- лист и х – не красный, выполняется:
2. Если х – левый потомок, выполняется:
3. Создаем новую переменную brother типа указатель на правого потомка родителя х
4. Если цвет brother – красный, то

* Перекрашиваем brother в черный
* Перекрашиваем родителя х в красный
* Выполняем левый поворот родителя х
* Переменной brother присваиваем значение указателя на правого потомка

Иначе переходим к шагу с.

1. Если правый и левый потомки brother – красные, то перекрашиваем brother в красный, а переменной х присваиваем значение указателя на родителя х, иначе выполняется:

* Если правый потомок brother – черный, то выполняется:
* Перекрашиваем левого потомка brother в черный
* Перекрашиваем brother в красный
* Выполняем правый поворот узла brother
* Переменной brother присваиваем значение указателя правого потомка родителя х
* Перекрашиваем brother в цвет родителя х
* Перекрашиваем родителя х в черный
* Перекрашиваем правого потомка brother в черный
* Выполняем левый поворот родителя х
* Переменной х присваиваем значение указателя на root(корень дерева)

1. Если х – правый потомок, выполняется блок аналогичный с точностью до перестановки правого и левого поворотов вышеописанному блоку.
2. Перекрашиваем х в черный

#### *Получение минимального элемента*

1. Если дерево не пусто, то создаем переменную rootcopy типа указатель на вершину дерева и присваиваем ей значение указателя на корень, иначе пункт 4.
2. Пока указатель на левого потомка вершины rootcopy не равен NIL, выполняется:
   * Присвоить rootcopy значение указателя на левого потомка
3. Вернуть значение ключа переменной rootcopy.
4. Вызывается исключение.

### **Алгоритмы методов класса Six\_Heap (6-куча)**

6-heap – шестеричное дерево, удовлетворяющее двум условиям:

1. Приоритет любой вершины не больше приоритета ее потомков
2. Дерево является полным 6-деревом, т.е. все уровни заполнены слева направо (за исключением может быть последнего)

Каждая вершина дерева – элемент массива. Если, например, рассматриваемая вершина имеет индекс i, то её родитель имеет индекс (i-1)/6, а её потомки: 6\*i+1, 6\*i+2, 6\*i+3, 6\*i+4, 6\*i+5, 6\*i+6.

#### *Поиск минимального элемента*

Минимальный элемент в куче – первый элемент массива.

#### *Вставка нового элемента*

1. Вставляем новый элемент в конец массива
2. Выполняем просеивание вверх

#### *Просеивание вверх*

1. На вход подается индекс i нового вставленного элемента
2. Пока i больше нуля и родитель элемента i больше элемента по индексу i, выполнять:
3. Поменять местами элементы массива с индексом i и индексом родителя i
4. Индексу i присвоить значение индекса родителя элемента, лежащего по текущему i.

#### *Удаление минимального элемента*

1. Меняем местами первый и последний элемент в массиве
2. Удаляем последний элемент в массиве
3. Выполняем просеивание вниз

#### *Просеивание вниз*

Индекс parent = 0, index\_min\_son равен индексу минимального сына нулевого элемента

Пока индекс минимального сына меньше индекса последнего и не меньше индекса первого элемента и значение родителя больше чем значение минимального сына

* Меняем местами значения родителя и минимального сына
* Индексу parent присваиваем значение index\_min\_son
* index\_min\_son присваиваем значение минимального сына для элемента с индексом parent

### **Алгоритм Дейкстры**

Пусть имеем неориентированный взвешенный граф. Веса всех рёбер неотрицательны. Указана некоторая стартовая вершинаstart\_v. Требуется найти длины кратчайших путей от вершиныstart\_vдо всех остальных вершин.

Алгоритм Дейкстры на приоритетной очереди в данной программе реализован двумя способами:

1. C использованием красно-чёрных деревьев
2. C использованием 6-кучи

На вход будем подавать стартовую вершину start\_v и вектор векторов пар, где пара – номер вершины и расстояние до неё.

***Описание алгоритма***

visited – массив типа bool - хранит информацию о том, была посещена вершина или нет

distance – вектор, в котором хранятся текущие значения расстояний до вершин

pq – очередь c приоритетом (реализована на красно–черном дереве, либо на 6 –куче), её элементами являются пары (номер вершины и расстояние до неё)

1. Пушим в очередь стартовую вершину
2. Пока очередь q не пуста, выполняем (если пуста, то переходим к шагу 3):

* В переменную current\_node кладем номер вершины с наименьшим расстоянием
* Удаляем элемент с минимальным расстоянием из очереди pq

Если вершина с номером current\_node не была посещена, то выполняем:

1. Отмечаем вершину посещённой
2. Проверяем каждую смежную с ней (current\_node) вершину node.
3. Если сумма расстояния до текущей вершины и веса ребра - len, соединяющего текущую и смежную вершину, меньше расстояния от стартовой вершины до смежной, то:
   * + В distance[node] кладем сумму distance[current\_node] + len
     + Пушим в очередь новую пару (node, distance[node])
4. Возвращаем вектор distance кратчайших расстояний от заданной вершины до всех остальных.

# **Заключение**

В данной работе разработаны классы красно-черных деревьев и 6-кучи. На их основе разработан класс приоритетной очереди, который используется для алгоритма Дейкстры поиска кратчайшего пути. Работоспособность реализации алгоритмов проверена тестами.

# **Литература**

Красно-чёрные деревья:

1. [https://ru.wikipedia.org/wiki/Красно-чёрное\_дерево]
2. [http://www.mkurnosov.net/teaching/uploads/DSA/dsa-fall-lecture4.pdf]
3. [http://algolist.manual.ru/ds/rbtree.php]
4. [http://mech.math.msu.su/~vvb/2course/Borisenko/lecTree.html]

Алгоритм Дейкстры:

1. [https://ru.wikipedia.org/wiki/Алгоритм\_Дейкстры]
2. [https://foxford.ru/wiki/informatika/algoritm-deykstry]