Федеральное государственное автономное образовательное учреждение  
высшего образования  
«Нижегородский государственный университет  
им. Н.И. Лобачевского»

Институт информационных технологий, математики и механики (ИИТММ)

Отчёт по лабораторной работе

**Алгоритм Дейкстры поиска кратчайшего пути**

Выполнил:  
студент ИИТММ гр. 0823-1  
 Красикова Е.А.

Проверил:

к.т.н., ассистент каф. ПрИнж  
ИИТММ

Сиднев А.А.

Нижний Новгород  
2017г.

Содержание

[Введение 3](#_Toc481097172)

[Постановка учебно-практической задачи 4](#_Toc481097173)

[Руководство пользователя 5](#_Toc481097174)

[Руководство программиста 6](#_Toc481097175)

[Общее описание структуры программного комплекса 6](#_Toc481097176)

[Описание структур данных 6](#_Toc481097177)

[Описание алгоритмов 8](#_Toc481097178)

[Алгоритм Дейкстры с использованием приоритетной очереди, реализованной на основе красно-черного дерева 8](#_Toc481097179)

[Алгоритм Дейкстры с использованием приоритетной очереди, реализованной на 4-кучи 9](#_Toc481097180)

[Алгоритмы реализации методов класса RBTree 9](#_Toc481097181)

[Алгоритмы реализации методов класса FourHeap 14](#_Toc481097182)

[Заключение 16](#_Toc481097183)

[Список литературы 17](#_Toc481097184)

[Приложение 1. Dijkstra.h 18](#_Toc481097185)

[Приложение 2. CreateGraph.cpp 18](#_Toc481097186)

[Приложение 3. DijkstraRBTree.cpp 18](#_Toc481097187)

[Приложение 4. DijkstraFourHeap.cpp 19](#_Toc481097188)

[Приложение 5. rbnode.h 20](#_Toc481097189)

[Приложение 6. RBTree.h 20](#_Toc481097190)

[Приложение 7. FourHeap.h 26](#_Toc481097191)

[Приложение 8. PQRBTree.h 28](#_Toc481097192)

[Приложение 9. PQFHeap.h 28](#_Toc481097193)

[Приложение 10. test\_RBTree.cpp 29](#_Toc481097194)

[Приложение 11. test\_FourHeap.cpp 31](#_Toc481097195)

[Приложение 12. test\_DijkstraRBTree.cpp 32](#_Toc481097196)

[Приложение 13. test\_DijkstraFourHeap.cpp 33](#_Toc481097197)

# Введение

Задача о нахождении кратчайших путей возникает в повседневной жизни довольно часто. К примеру, необходимо узнать, какую последовательность дорог лучше использовать, чтобы добраться из одного города до каждого из многих других, или в какие страны выгодней экспортировать нефть и тому подобное. Для решения указанной задачи можно использовать алгоритм Дейкстры - алгоритм на графах, изобретённый нидерландским ученым Э. Дейкстрой в 1959 году. [1]

# Постановка учебно-практической задачи

Необходимо реализовать алгоритм Дейкстры поиска кратчайшего пути с использованием приоритетной очереди. Приоритетную очередь необходимо реализовать двумя способами: с использованием красно-черных деревьев и  4-кучи. Граф необходимо читать из файла, в котором он будет описан списком смежности. Реализовать тесты, доказывающие корректность работы алгоритма и используемых структур данных.

# Руководство пользователя

Чтобы посчитать кратчайшие расстояния от заданной вершины графа до всех остальных, необходимо:

1. Задать граф при помощи списка смежности в следующем формате (в отдельном файле):

[количество вершин]

[количество вершин, смежных с вершиной] [номер смежной вершины] [вес ребра]

1. Считать граф из файла при помощи функции:

vector < vector < pair<int, int> > > CreateGraph(char filename[])

в качестве параметра указать “COPYONLY”

1. Воспользоваться функцией

vector<int> DijkstraRBTree(vector < vector< pair<int, int> > > g, int start)

либо

vector<int> DijkstraFourHeap(vector < vector< pair<int, int> > > g, int start);

в качестве первого параметра указать считанный из файла граф, в качестве второго – номер стартовой вершины.

# Руководство программиста

## Общее описание структуры программного комплекса

Разработанная программа включает:

1. заголовочный файл «Dijkstra.h» (содержит прототипы функций DijkstraRBTree, DijkstraFourHeap, CreateGraph)
2. файл исходного кода «CreateGraph.cpp» (содержит реализацию функции CreateGraph)
3. файл исходного кода «DijkstraRBTree.cpp» (содержит реализацию функции DijkstraRBTree)
4. файл исходного кода «DijkstraFourHeap.cpp» (содержит реализацию функции DijkstraFourHeap)
5. заголовочный файл «rbnode.h» (содержит объявление шаблонной структуры rbnode)
6. заголовочный файл «RBTree.h» (содержит интерфейс шаблонного класса RBTree и реализацию его методов)
7. заголовочный файл «FourHeap.h» (содержит интерфейс шаблонного класса FourHeap и реализацию его методов)
8. заголовочный файл «PQRBTree.h» (содержит интерфейс шаблонного класса PQRBTree и реализацию его методов)
9. заголовочный файл «PQFHeap.h» (содержит интерфейс шаблонного класса PQFHeap и реализацию его методов)
10. файл исходного кода «test\_RBTree.cpp» (содержит тесты, проверяющие корректность работы методов класса RBTree)
11. файл исходного кода «test\_FourHeap.cpp» (содержит тесты, проверяющие корректность работы методов класса FourHeap)
12. файл исходного кода «test\_DijkstraRBTree.cpp» (содержит тесты, проверяющие корректность работы функции DijkstraRBTree)
13. файл исходного кода «test\_DijkstraFourHeap.cpp» (содержит тесты, проверяющие корректность работы функции DijkstraFourHeap)

## Описание структур данных

template <class T>

struct rbnode { // Шаблонная структура для хранения вершины красно-

черного дерева

T val; // Ключ-значение

rbnode \*left, // указатель на левого потомка

\*right, // указатель на правого потомка

\*parent; // указатель на родителя

int color; // цвет вершины (BLACK = 0, RED = 1)

};

extern rbnode<int> NILnode; // внешняя переменная для хранения   
 фиктивной вершины

template <class T>

class RBTree { // шаблонный класс для представления красно-черного   
 дерева

rbnode<T>\* root; // корень дерева

void RotateLeft(rbnode<T> \*x); // метод, осуществляющий левый   
 поворот дерева вокруг вершины x

void RotateRight(rbnode<T> \*x); // метод, осуществляющий правый   
 поворот дерева вокруг вершины x

void InsertFixup(rbnode<T> \*x); // метод, восстанавливающий   
 свойства красно-черного   
 дерева после добавления   
 новой вершины

void DeleteFixup(rbnode<T> \*x); ); // метод, восстанавливающий   
 свойства красно-черного   
 дерева после удаления

вершины

public:

RBTree(); // конструктор без параметров

explicit RBTree(T key); // конструктор с одним параметром (ключом)

~RBTree(void); // деструктор

rbnode<T>\* GetRoot(); // метод, обеспечивающий доступ к корню

void Insert(T v); // добавление вершины

void Delete(T v); // удаление вершины

rbnode<T>\* Find(T v); // поиск вершины по ключу

rbnode<T>\* Min(); // вершина с минимальным ключом

bool IsRBTree(); // проверка свойств красно-черного дерева

};

template <class T>

class FourHeap { //шаблонный класс для представления 4-кучи

vector<T> heap; // вектор элементов

int child(int i); // метод, определяющий индекс ребенка по индексу   
 родителя

int parent(int i); // метод, определяющий индекс родителя по   
 индексу ребенка

public:

T Min(); // метод, определяющий минимальный элемент

void DeleteMin(); // удаление минимального элемента

void Insert(T key); // вставка нового элемента

bool empty(); // проверка на пустоту

};

template <class T>

class PQRBTree { // шаблонный класс для представления приоритетной   
 очереди с использованием красно-черных деревьев

RBTree<T> tree; // красно-черное дерево

public:

rbnode<T>\* Find(T key); // поиск элемента

rbnode<T>\* Min(); // определение минимального элемента

void Insert(T key); // вставка нового элемента

void Delete(T key); // удаление элемента

bool empty(); // проверка на пустоту

};

template <class T>

class PQFHeap { // шаблонный класс для представления приоритетной   
 очереди с использованием 4-кучи

FourHeap<T> h; // 4-куча

public:

T Min(); // определение минимального элемента

void DeleteMin(); // удаление минимального элемента

void Insert(T key); // вставка нового элемента

bool empty(); // проверка на пустоту

};

## Описание алгоритмов

Дан ориентированный или неориентированный взвешенный граф с*n*вершинами и*m*рёбрами. Веса всех рёбер неотрицательны. Указана некоторая стартовая вершинаs. Требуется найти длины кратчайших путей из вершиныsво все остальные вершины. Эта задача называется "задачей о кратчайших путях с единственным источником" (single-source shortest paths problem). В данной лабораторной работе будет рассмотрен алгоритм, который предложил голландский исследователь Дейкстра (Dijkstra) в 1959 г. [2]

### Алгоритм Дейкстры с использованием приоритетной очереди, реализованной на основе красно-черного дерева

d – массив расстояний от стартовой вершины до всех остальных ( в начальный момент расстояние до стартовой вершины равно 0, до всех остальных – бесконечность)

pq – приоритетная очередь, элементами которой являются пары (расстояние до вершины, номер вершины)

Начало алгоритма

1. Добавляем в очередь пару (расстояние до стартовой вершины = 0, номер стартовой вершины) (метод Insert)
2. Выполняем, пока очередь не пуста:
3. Выбираем из очереди вершину, расстояние до которой минимально (при помощи метода Min) и удаляем её из очереди (метод Delete)
4. Проверяем каждую смежную с ней вершину:
5. Если сумма расстояния до текущей вершины и веса ребра, соединяющего текущую и смежную вершину, меньше расстояния от стартовой вершины до смежной:
6. если расстояние от стартовой вершины до смежной не равно бесконечности, удаляем пару (расстояние от стартовой вершины до смежной, номер смежной вершины) из очереди
7. обновляем расстояние для смежной вершины в массиве d
8. вставляем пару (расстояние от стартовой вершины до смежной, номер смежной вершины) в очередь

Конец алгоритма

По окончании работы алгоритма массив d содержит кратчайшие расстояния от стартовой вершины до всех остальных вершин графа.

### Алгоритм Дейкстры с использованием приоритетной очереди, реализованной на 4-кучи

d – массив расстояний от стартовой вершины до всех остальных ( в начальный момент расстояние до стартовой вершины равно 0, до всех остальных – бесконечность)

pq – приоритетная очередь, элементами которой являются пары (расстояние до вершины, номер вершины)

Начало алгоритма

1. Добавляем в очередь пару (расстояние до стартовой вершины = 0, номер стартовой вершины) (метод Insert)
2. Выполняем, пока очередь не пуста:
3. Выбираем из очереди вершину, расстояние до которой минимально (при помощи метода Min) и удаляем её из очереди (метод DeleteMin)
4. Проверяем каждую смежную с ней вершину:
5. Если сумма расстояния до текущей вершины и веса ребра, соединяющего текущую и смежную вершину, меньше расстояния от стартовой вершины до смежной:
6. обновляем расстояние для смежной вершины в массиве d
7. вставляем пару (расстояние от стартовой вершины до смежной, номер смежной вершины) в очередь

Конец алгоритма

В силу невозможности реализации метода удаления произвольного элемента из 4-кучи с логарифмической сложностью, мы не удаляем пары с устаревшими расстояниями до вершин, но т.к. на каждой итерации мы выбираем вершину с кратчайшим расстоянием, этот факт не повлияет на корректность работы алгоритма.

### Алгоритмы реализации методов класса RBTree

Красно-черное дерево – это бинарное дерево поиска, для которого выполняются красно-черные свойства:

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 [3]

#### Поиск элемента

Сложность операции – О(logn) (n- количество вершин).

Начало алгоритма

1. Создаем переменную tmp типа указатель на вершину дерева и присваиваем ей значение указателя на корень
2. Пока tmp не равно NIL (указателю на фиктивное звено), выполнять:
3. Если ключ вершины tmp равен искомому ключу, вернуть tmp
4. Если искомый ключ меньше ключа вершины tmp, присвоить tmp значение указателя на левого потомка
5. Если искомый ключ больше ключа вершины tmp, присвоить tmp значение указателя на правого потомка
6. Вызвать исключение: в дереве нет искомого элемента

Конец алгоритма

#### Поиск минимального элемента

Сложность операции – О(logn)

Начало алгоритма

1. Создаем переменную tmp типа указатель на вершину дерева и присваиваем ей значение указателя на корень
2. Пока указатель на левого потомка вершины tmp не равен NIL, выполнять:
3. Присвоить tmp значение указателя на левого потомка
4. Вернуть tmp

Конец алгоритма

#### Вставка нового элемента

Сложность операции – О(logn)

Начало алгоритма

1. Находим лист для вставки нового элемента (результат операции поиска в случае, когда искомого элемента нет в дереве)
2. Создаем элемент и окрашиваем его в красный цвет
3. Вызываем метод, восстанавливающий красно-черные свойства после добавления нового элемента (InsertFixup)

Конец алгоритма

В случае, если элемент уже есть в дереве, будет вызвано исключение.

#### Восстановление свойств красно-черного дерева после добавления элемента

x – указатель на добавленный элемент

Начало алгоритма

1. Пока x не корень и родитель x красный, выполнять:
2. Если родитель x - левый потомок своего родителя, выполнять:
3. y – указатель на дядю x ( правый потомок родителя родителя x)
4. если y красный:
5. перекрашиваем y и родителя x в черный, а родителя родителя (дедушку) x в красный
6. присваиваем x указатель на дедушку
7. если y черный:
8. если х – правый потомок своего родителя, присваиваем x указатель на родителя и выполняем левый поворот вокруг x
9. перекрашиваем родителя x в черный, а дедушку x в красный
10. выполняем правый поворот вокруг дедушки x
11. Если родитель x - правый потомок своего родителя, выполнить симметричные действия
12. Покрасить корень в черный

Конец алгоритма

#### Левый поворот

y – правый потомок x

Начало алгоритма

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 0:
4. Если x – левый потомок своего родителя, присвоить указателю на левого потомка родителя x значение y
5. Если x – правый потомок своего родителя, присвоить указателю на правого потомка родителя x значение y
6. Если указатель на родителя x не равен 0, присвоить указателю на корень значение y
7. Присвоить указателю на левого потомка y значение x
8. Присвоить указателю на родителя x значение y

Конец алгоритма

#### Правый поворот

y – левый потомок x

Начало алгоритма

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 0:
4. Если x – левый потомок своего родителя, присвоить указателю на левого потомка родителя x значение y
5. Если x – правый потомок своего родителя, присвоить указателю на правого потомка родителя x значение y
6. Если указатель на родителя x не равен 0, присвоить указателю на корень значение y
7. Присвоить указателю на правого потомка y значение x
8. Присвоить указателю на родителя x значение y

Конец алгоритма

#### Удаление элемента

Сложность операции – О(logn)

Начало алгоритма

1. По заданному ключу находим элемент для удаления – y (у него должно быть не больше одного потомка):
2. Выполняем операцию поиска
3. Если найденный элемент равен NIL, вызвать исключение (такой вершины не существует в дереве)
4. Если у найденного элемента есть NIL-потомок, то присваиваем y значение указателя на него
5. Если у найденного элемента есть два потомка, находим элемент, следующий по ключу (переходим в правое поддерево и движемся влево, пока левый потомок не равен NIL) и присваиваем y значение указателя на него
6. Соединяем указателями родителя и потомка y
7. Если ключ y не совпадает с ключом, который нужно удалить, то записываем ключ y в вершину с удаляемым ключом
8. Если вершина по указателю y – черная, перекрашивая узлы и выполняя повороты восстанавливаем структуру красно-черного дерева (DeleteFixup)
9. Удаляем вершину по указателю y

Конец алгоритма

#### Восстановление свойств красно-черного дерева после удаления элемента

x – указатель на потомка удаленного элемента

Начало алгоритма

1. Пока x не корень и имеет черный цвет, выполнять:
2. Если x – левый потомок своего родителя:
3. y – правый потомок родителя x
4. если y – красный:
5. перекрашиваем y в черный, а родителя x - в красный
6. выполняем левый поворот вокруг родителя x
7. присваиваем y значение указателя на правого потомка родителя x
8. Если оба потомка y черные:
9. перекрашиваем y в красный
10. присваиваем x значение указателя на родителя x
11. Если один из потомков y не черный:
12. Если правый потомок y черный:

* Перекрашиваем левого потомка в черный
* Перекрашиваем y в красный
* Выполняем правый поворот вокруг y
* Присваиваем y значение указателя на правого потомка родителя x

1. Перекрашиваем y в цвет родителя x
2. Родителя x и правого потомка y перекрашиваем в черный
3. Выполняем левый поворот вокруг родителя x
4. Назначаем x корнем
5. Если x – правый потомок своего родителя, выполняем симметричные действия
6. Перекрашиваем x в черный

Конец алгоритма

#### Алгоритм проверки свойств красно-черного дерева

Начало алгоритма

1. q – очередь с элементами типа указатель на вершину
2. Если корень красный, вернуть false
3. Если корень равен NIL, вернуть true
4. В противном случае, добавить корень в очередь
5. Пока очередь не пуста, выполнять:
6. tmp – элемент из начала очереди
7. удалить элемент из очереди
8. если tmp и один из его потомков красные, вернуть false
9. если левый потомок не равен NIL, добавить его в очередь
10. если правый потомок не равен NIL, добавить его в очередь
11. s1 – стек с элементами типа указатель на вершину
12. s2 – стек с элементами типа int
13. s3 – стек с элементами типа bool
14. Добавляем корень в стек s1
15. Добавляем false в стек s3
16. Пока стек s1 не пуст, выполнять:
17. tmp – элемент с вершины стека s1
18. если элемент с вершины стека s3 равен false (попали в tmp в первый раз):
19. удаляем из стека false и кладем true
20. если tmp равен NIL:
21. кладем 1 в стек s2
22. удаляем элемент из стека s1
23. удаляем элемент из стека s3
24. кладем левого потомка tmp в стек s1 и false в s3
25. кладем правого потомка tmp в стек s1 и false в s3
26. если элемент с вершины стека s3 равен true (вернулись в tmp):
27. правая черная высота – элемент с вершины стека s2
28. удалить элемент с вершины стека s2
29. левая черная высота – элемент с вершины стека s2
30. удалить элемент с вершины стека s2
31. если правая черная высота не равна левой черной высоте, вернуть false
32. если правая черная высота равна левой черной высоте:
33. удалить элементы с вершин s1 и s3
34. если tmp черный, положить в стек s2 чёрную высоту, увеличенную на единицу
35. если tmp не черный, положить в стек s2 старую чёрную высоту
36. Вернуть true

Конец алгоритма

### Алгоритмы реализации методов класса FourHeap

4-heap (пирамида, сортирующее дерево, binary heap) – это 4-дерево, удовлетворяющее следующим условиям:

1. Приоритет любой вершины не больше приоритета ее потомков
2. Дерево является полным 4-деревом – все уровни заполнены слева направо

(возможно за исключением последнего) [4]

Каждая вершина дерева соответствует элементу массива. Если вершина имеет индекс i, то её родитель имеет индекс (i-1)/4, а её дети 4\*i+1, 4\*i+2, 4\*i+3, 4\*i+4. [5]

#### Поиск минимального элемента

Сложность операции – О(1). Минимальный элемент в 4-куче – первый элемент в массиве.

#### Вставка нового элемента

Сложность операции – О(logn)

Начало алгоритма

1. Добавляем новый элемент в конец массива
2. tmp – индекс добавленного элемента
3. Пока tmp больше нуля и родитель элемента tmp больше элемента tmp, выполнять:
4. Поменять местами элементы массива с индексами родителя tmp и tmp
5. Присвоить tmp индекс родителя tmp

Конец алгоритма

#### Удаление минимального элемента

Сложность операции – О(logn)

Начало алгоритма

1. Записать на первое место в массиве последний элемент
2. Удалить из массива последний элемент
3. tmp = 0
4. Пока индекс потомка tmp меньше размера массива, выполнять:
5. Найти минимальный элемент среди tmp и его потомков
6. Если минимальный элемент имеет индекс tmp, выйти из цикла
7. В противном случае поменять местами tmp и минимального потомка tmp
8. Присвоить tmp индекс минимального потомка

Конец алгоритма

# Заключение

Реализован алгоритм Дейкстры поиска кратчайшего пути с использованием приоритетной очереди. Реализованы классы красно-черных деревьев и  4-кучи, на основе их реализованы соответствующие классы приоритетных очередей. Реализованы необходимые тесты.

# Список литературы

1. Алгоритм Дейкстры – [https://prog-cpp.ru/deikstra/]
2. Алгоритм Дейкстры - [http://www.e-maxx-ru.1gb.ru/algo/dijkstra]
3. Красно-черные деревья – [http://www.mkurnosov.net/teaching/uploads/DSA/dsa-fall-lecture4.pdf]
4. Свойства кучи – [http://www.mkurnosov.net/teaching/uploads/DSA/dsa-fall2014-lec6.pdf]
5. Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн "Алгоритмы. Построение и анализ" Вильямс, 2013 год, 1324 стр. Издание 3-е
6. Бьёрн Страуструп.  Язык программирования C++ = TheC++ ProgrammingLanguage / Пер. с англ. — 3-е изд. — СПб.; М.: Невский диалект — Бином, 1999.

# Приложение 1. Dijkstra.h

#ifndef INCLUDE\_DIJKSTRA\_H\_

#define INCLUDE\_DIJKSTRA\_H\_

#include <vector>

#include <utility>

#include <stdlib.h>

#define INF 0x3F3F3F3F

#define vector std::vector

#define pair std::pair

vector<int> DijkstraRBTree(vector < vector<

pair<int, int> > > g, int start);

vector<int> DijkstraFourHeap(vector < vector<

pair<int, int> > > g, int start);

vector < vector < pair<int, int> > >

CreateGraph(char filename[]);

#endif // INCLUDE\_DIJKSTRA\_H\_

# Приложение 2. CreateGraph.cpp

#include <vector>

#include <fstream>

#include <utility>

#define pair std::pair

#define vector std::vector

vector < vector < pair<int, int> > > CreateGraph(char filename[]) {

std::ifstream file(filename);

int N, n;

file >> N;

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

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

file >> n;

g[i] = vector< pair<int, int>>(n);

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

file >> g[i][j].first >> g[i][j].second;

}

}

return g;

}

# Приложение 3. DijkstraRBTree.cpp

#include "PQRBTree.h"

#include "Dijkstra.h"

#include <vector>

#include <utility>

#include <stdexcept>

#define pair std::pair

#define vector std::vector

vector<int> DijkstraRBTree(vector < vector < pair<int, int> > > g, int start) {

if ( g.empty() || (start < 1) || (start > static\_cast<int>(g.size())) )

throw std::logic\_error("incorrect parametrs");

vector<int> d(g.size(), INF);

int v;

d[start-1] = 0;

PQRBTree< pair<int, int> > pq;

pq.Insert(pair<int, int>(0, start));

while (!pq.empty()) {

pair<int, int> tmp = pq.Min()->val;

pq.Delete(tmp);

v = tmp.second-1;

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

if (d[v] + g[v][i].second < d[g[v][i].first-1]) {

if (d[g[v][i].first-1] != INF)

pq.Delete(pair<int, int>(d[g[v][i].first-1],

g[v][i].first));

d[g[v][i].first-1] = d[v] + g[v][i].second;

pq.Insert(pair<int, int>(d[g[v][i].first-1], g[v][i].first));

}

}

}

return d;

}

# Приложение 4. DijkstraFourHeap.cpp

#include "PQFHeap.h"

#include "Dijkstra.h"

#include <vector>

#include <utility>

#include <stdexcept>

#define pair std::pair

#define vector std::vector

vector<int> DijkstraFourHeap(vector < vector < pair<int, int> > > g,

int start) {

if ( g.empty() || (start < 1) || (start > static\_cast<int>(g.size())) )

throw std::logic\_error("incorrect parametrs");

vector<int> d(g.size(), INF);

int v;

d[start-1] = 0;

PQFHeap< pair<int, int> > pq;

pq.Insert(pair<int, int>(0, start));

while (!pq.empty()) {

pair<int, int> tmp = pq.Min();

pq.DeleteMin();

v = tmp.second-1;

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

if (d[v] + g[v][i].second < d[g[v][i].first-1]) {

d[g[v][i].first-1] = d[v] + g[v][i].second;

pq.Insert(pair<int, int>(d[g[v][i].first-1], g[v][i].first));

}

}

}

return d;

}

# Приложение 5. rbnode.h

#ifndef INCLUDE\_RBNODE\_H\_

#define INCLUDE\_RBNODE\_H\_

template <class T>

struct rbnode {

T val;

rbnode \*left, \*right, \*parent;

int color; // BLACK = 0, RED = 1

};

extern rbnode<int> NILnode;

#define NIL &NILnode

#endif // INCLUDE\_RBNODE\_H\_

# Приложение 6. RBTree.h

#ifndef INCLUDE\_RBTREE\_H\_

#define INCLUDE\_RBTREE\_H\_

#include "rbnode.h"

#include <queue>

#include <stack>

#include <stdexcept>

template <class T>

class RBTree {

rbnode<T>\* root;

void RotateLeft(rbnode<T> \*x);

void RotateRight(rbnode<T> \*x);

void InsertFixup(rbnode<T> \*x);

void DeleteFixup(rbnode<T> \*x);

public:

RBTree();

explicit RBTree(T key);

~RBTree(void);

rbnode<T>\* GetRoot();

void Insert(T v);

void Delete(T v);

rbnode<T>\* Find(T v);

rbnode<T>\* Min();

bool IsRBTree();

};

template <class T>

RBTree<T>::RBTree() {

root = reinterpret\_cast<rbnode<T>\*>(NIL);

}

template <class T>

RBTree<T>::RBTree(T key) {

root = new rbnode<T>;

root->val = key;

root->color = 0;

root->left = root->right = reinterpret\_cast<rbnode<T>\*>(NIL);

root->parent = 0;

}

template <class T>

RBTree<T>::~RBTree() {

std::queue<rbnode<T>\*> q;

if (root != reinterpret\_cast<rbnode<T>\*>(NIL)) q.push(root);

rbnode<T>\* tmp;

while (!q.empty()) {

tmp = q.front();

q.pop();

if (tmp->left != reinterpret\_cast<rbnode<T>\*>(NIL))

q.push(tmp->left);

if (tmp->right != reinterpret\_cast<rbnode<T>\*>(NIL))

q.push(tmp->right);

delete tmp;

}

}

template <class T>

rbnode<T>\* RBTree<T>::GetRoot() {

return root;

}

template <class T>

void RBTree<T>::RotateLeft(rbnode<T> \*x) {

rbnode<T> \*y = x->right;

x->right = y->left;

if (y->left != reinterpret\_cast<rbnode<T>\*>(NIL))

y->left->parent = x;

y->parent = x->parent;

if (x->parent != 0) {

if (x == x->parent->left) {

x->parent->left = y;

} else {

x->parent->right = y;

}

} else {

root = y;

}

y->left = x;

x->parent = y;

}

template <class T>

void RBTree<T>::RotateRight(rbnode<T> \*x) {

rbnode<T> \*y = x->left;

x->left = y->right;

if (y->right != reinterpret\_cast<rbnode<T>\*>(NIL))

y->right->parent = x;

y->parent = x->parent;

if (x->parent != 0) {

if (x == x->parent->left) {

x->parent->left = y;

} else {

x->parent->right = y;

}

} else {

root = y;

}

y->right = x;

x->parent = y;

}

template <class T>

void RBTree<T>::InsertFixup(rbnode<T> \*x) {

while ( (x != root) && (x->parent->color == 1) ) {

if (x->parent == x->parent->parent->left) {

rbnode<T> \*y = x->parent->parent->right;

if (y->color == 1) {

x->parent->color = 0;

y->color = 0;

x->parent->parent->color = 1;

x = x->parent->parent;

} else {

if (x == x->parent->right) {

x = x->parent;

RotateLeft(x);

}

x->parent->color = 0;

x->parent->parent->color = 1;

RotateRight(x->parent->parent);

}

} else {

rbnode<T> \*y = x->parent->parent->left;

if (y->color == 1) {

x->parent->color = 0;

y->color = 0;

x->parent->parent->color = 1;

x = x->parent->parent;

} else {

if (x == x->parent->left) {

x = x->parent;

RotateRight(x);

}

x->parent->color = 0;

x->parent->parent->color = 1;

RotateLeft(x->parent->parent);

}

}

}

root->color = 0;

}

template <class T>

void RBTree<T>::Insert(T v) {

rbnode<T> \*tmp = root, \*ptmp = 0, \*x = 0;

while (tmp != reinterpret\_cast<rbnode<T>\*>(NIL)) {

if (tmp->val == v) throw std::logic\_error("this key already exist");

ptmp = tmp;

if (v < tmp->val) {

tmp = tmp->left;

} else {

tmp = tmp->right;

}

}

x = new rbnode<T>;

x->parent = ptmp;

x->left = x->right = reinterpret\_cast<rbnode<T>\*>(NIL);

x->color = 1;

x->val = v;

if (ptmp != 0) {

if (v < ptmp->val) {

ptmp->left = x;

} else {

ptmp->right = x;

}

} else {

root = x;

}

InsertFixup(x);

}

template <class T>

void RBTree<T>::DeleteFixup(rbnode<T> \*x) {

while ( (x != root) && (x->color == 0) ) {

if (x == x->parent->left) {

rbnode<T> \*y = x->parent->right;

if (y->color == 1) {

y->color = 0;

x->parent->color = 1;

RotateLeft(x->parent);

y = x->parent->right;

}

if ( (y->left->color == 0) && (y->right->color == 0) ) {

y->color = 1;

x = x->parent;

} else {

if (y->right->color == 0) {

y->left->color = 0;

y->color = 1;

RotateRight(y);

y = x->parent->right;

}

y->color = x->parent->color;

x->parent->color = 0;

y->right->color = 0;

RotateLeft(x->parent);

x = root;

}

} else {

rbnode<T> \*y = x->parent->left;

if (y->color == 1) {

y->color = 0;

x->parent->color = 1;

RotateRight(x->parent);

y = x->parent->left;

}

if ( (y->right->color == 0) && (y->left->color == 0) ) {

y->color = 1;

x = x->parent;

} else {

if (y->left->color == 0) {

y->right->color = 0;

y->color = 1;

RotateLeft(y);

y = x->parent->left;

}

y->color = x->parent->color;

x->parent->color = 0;

y->left->color = 0;

RotateRight(x->parent);

x = root;

}

}

}

x->color = 0;

}

template <class T>

void RBTree<T>::Delete(T v) {

rbnode<T> \*x, \*y, \*z;

z = root;

while (z != reinterpret\_cast<rbnode<T>\*>(NIL)) {

if (z->val == v) {

break;

} else {

if (v < z->val) {

z = z->left;

} else {

z = z->right;

}

}

}

if (z == reinterpret\_cast<rbnode<T>\*>(NIL))

throw std::logic\_error("this key does not exist");

if ( (z->left == reinterpret\_cast<rbnode<T>\*>(NIL)) ||

(z->right == reinterpret\_cast<rbnode<T>\*>(NIL)) ) {

y = z;

} else {

y = z->right;

while (y->left != reinterpret\_cast<rbnode<T>\*>(NIL)) y = y->left;

}

if (y->left != reinterpret\_cast<rbnode<T>\*>(NIL)) {

x = y->left;

} else {

x = y->right;

}

x->parent = y->parent;

if (y->parent != 0) {

if (y == y->parent->left) {

y->parent->left = x;

} else {

y->parent->right = x;

}

} else {

root = x;

}

if (y != z) z->val = y->val;

if (y->color == 0) DeleteFixup(x);

delete y;

}

template <class T>

rbnode<T>\* RBTree<T>::Find(T v) {

rbnode<T> \*tmp = root;

while (tmp != reinterpret\_cast<rbnode<T>\*>(NIL)) {

if (v == tmp->val) {

return tmp;

} else {

if (v < tmp->val) {

tmp = tmp->left;

} else {

tmp = tmp->right;

}

}

}

throw std::logic\_error("this key does not exist");

}

template <class T>

rbnode<T>\* RBTree<T>::Min() {

rbnode<T>\* tmp = root;

while (tmp->left != reinterpret\_cast<rbnode<T>\*>(NIL))

tmp = tmp->left;

return tmp;

}

template <class T>

bool RBTree<T>::IsRBTree() {

std::queue<rbnode<T>\*> q;

if (root->color == 1) return false;

if (root != reinterpret\_cast<rbnode<T>\*>(NIL)) {

q.push(root);

} else {

return true;

}

rbnode<T>\* tmp;

while (!q.empty()) {

tmp = q.front();

q.pop();

if ((tmp->color == 1) && ((tmp->left->color == 1) ||

(tmp->right->color == 1))) return false;

if (tmp->left != reinterpret\_cast<rbnode<T>\*>(NIL))

q.push(tmp->left);

if (tmp->right != reinterpret\_cast<rbnode<T>\*>(NIL))

q.push(tmp->right);

}

std::stack<rbnode<T>\*> s1;

std::stack<int> s2;

std::stack<bool> s3;

int lh, rh;

s1.push(root);

s3.push(false);

while (!s1.empty()) {

tmp = s1.top();

if (s3.top() == false) {

s3.pop();

s3.push(true);

if (tmp == NIL) {

s2.push(1);

s1.pop();

s3.pop();

continue;

}

s1.push(tmp->right);

s3.push(false);

s1.push(tmp->left);

s3.push(false);

} else {

rh = s2.top();

s2.pop();

lh = s2.top();

s2.pop();

if (rh != lh) {

return false;

} else {

s1.pop();

s3.pop();

if (tmp->color == 0) {

s2.push(lh+1);

} else {

s2.push(lh);

}

}

}

}

return true;

}

#endif // INCLUDE\_RBTREE\_H\_

# Приложение 7. FourHeap.h

#ifndef INCLUDE\_FOURHEAP\_H\_

#define INCLUDE\_FOURHEAP\_H\_

#include <vector>

#include <stdexcept>

#define vector std::vector

template <class T>

class FourHeap {

vector<T> heap;

int child(int i);

int parent(int i);

public:

T Min();

void DeleteMin();

void Insert(T key);

bool empty();

};

template <class T>

T FourHeap<T>::Min() {

return heap[0];

}

template <class T>

int FourHeap<T>::child(int i) {

if ( (i >= 0) && (4\*i+1 < static\_cast<int>(heap.size()))) return 4\*i+1;

throw std::logic\_error("incorrect index");

}

template <class T>

int FourHeap<T>::parent(int i) {

if ( (i > 0) && (i < static\_cast<int>(heap.size()))) return (i-1)/4;

throw std::logic\_error("incorrect index");

}

template <class T>

void FourHeap<T>::Insert(T key) {

heap.push\_back(key);

int tmp = static\_cast<int>(heap.size())-1;

while ((tmp > 0) && (heap[parent(tmp)] > heap[tmp])) {

T temp = heap[parent(tmp)];

heap[parent(tmp)] = heap[tmp];

heap[tmp] = temp;

tmp = parent(tmp);

}

}

template <class T>

void FourHeap<T>::DeleteMin() {

heap[0] = heap[static\_cast<int>(heap.size())-1];

heap.pop\_back();

int tmp = 0;

while (4\*tmp+1 < static\_cast<int>(heap.size())) {

int iofmin = tmp;

for (int j = 0; (j < 4) && (child(tmp)+j <

static\_cast<int>(heap.size())); j++)

if ( heap[child(tmp) + j] < heap[iofmin]) iofmin = child(tmp) + j;

if (tmp == iofmin) break;

T temp = heap[tmp];

heap[tmp] = heap[iofmin];

heap[iofmin] = temp;

tmp = iofmin;

}

}

template <class T>

bool FourHeap<T>::empty() {

return heap.empty();

}

#endif // INCLUDE\_FOURHEAP\_H\_

# Приложение 8. PQRBTree.h

#ifndef INCLUDE\_PQRBTREE\_H\_

#define INCLUDE\_PQRBTREE\_H\_

#include "RBTree.h"

template <class T>

class PQRBTree {

RBTree<T> tree;

public:

rbnode<T>\* Find(T key);

rbnode<T>\* Min();

void Insert(T key);

void Delete(T key);

bool empty();

};

template <class T>

rbnode<T>\* PQRBTree<T>::Find(T key) {

return tree.Find(key);

}

template <class T>

rbnode<T>\* PQRBTree<T>::Min() {

return tree.Min();

}

template <class T>

void PQRBTree<T>::Insert(T key) {

tree.Insert(key);

}

template <class T>

void PQRBTree<T>::Delete(T key) {

tree.Delete(key);

}

template <class T>

bool PQRBTree<T>::empty() {

return (tree.GetRoot() == reinterpret\_cast<rbnode<T>\*>(NIL));

}

#endif // INCLUDE\_PQRBTREE\_H\_

# Приложение 9. PQFHeap.h

#ifndef INCLUDE\_PQFHEAP\_H\_

#define INCLUDE\_PQFHEAP\_H\_

#include "FourHeap.h"

template <class T>

class PQFHeap {

FourHeap<T> h;

public:

T Min();

void DeleteMin();

void Insert(T key);

bool empty();

};

template <class T>

T PQFHeap<T>::Min() {

return h.Min();

}

template <class T>

void PQFHeap<T>::DeleteMin() {

h.DeleteMin();

}

template <class T>

void PQFHeap<T>::Insert(T key) {

h.Insert(key);

}

template <class T>

bool PQFHeap<T>::empty() {

return h.empty();

}

#endif // INCLUDE\_PQFHEAP\_H\_

# Приложение 10. test\_RBTree.cpp

#include <gtest/gtest.h>

#include "RBTree.h"

rbnode<int> NILnode = {0, 0, 0, 0, 0};

TEST(RBTree, can\_create\_empty\_tree) {

ASSERT\_NO\_THROW(RBTree<int> a);

RBTree<int> b;

EXPECT\_EQ(b.GetRoot(), NIL);

}

TEST(RBTree, can\_create\_tree) {

ASSERT\_NO\_THROW(RBTree<int> a(3));

RBTree<int> b(3);

EXPECT\_EQ(b.GetRoot()->val, 3);

}

TEST(RBTree, can\_insert\_node) {

RBTree<int> b(3);

ASSERT\_NO\_THROW(b.Insert(1));

}

TEST(RBTree, can\_insert\_node\_in\_empty\_tree) {

RBTree<int> b;

ASSERT\_NO\_THROW(b.Insert(1));

}

TEST(RBTree, throw\_when\_insert\_existing\_node) {

RBTree<int> b(3);

ASSERT\_ANY\_THROW(b.Insert(3));

}

TEST(RBTree, can\_find\_node) {

RBTree<int> b(3);

b.Insert(1);

ASSERT\_NO\_THROW(b.Find(1));

rbnode<int> \*res = b.Find(1);

EXPECT\_EQ(res->val, 1);

}

TEST(RBTree, throw\_when\_search\_absent\_node) {

RBTree<int> b(3);

b.Insert(1);

ASSERT\_ANY\_THROW(b.Find(2));

}

TEST(RBTree, can\_insert\_node\_in\_any\_case) {

RBTree<int> b(7);

ASSERT\_NO\_THROW(b.Insert(3));

ASSERT\_NO\_THROW(b.Insert(5));

ASSERT\_NO\_THROW(b.Insert(1));

ASSERT\_NO\_THROW(b.Insert(2));

ASSERT\_NO\_THROW(b.Insert(4));

ASSERT\_NO\_THROW(b.Insert(6));

ASSERT\_NO\_THROW(b.Find(2));

}

TEST(RBTree, throw\_when\_delete\_absent\_node) {

RBTree<int> b(3);

b.Insert(1);

ASSERT\_ANY\_THROW(b.Delete(2));

}

TEST(RBTree, can\_delete\_node\_with\_no\_childs) {

RBTree<int> b(3);

b.Insert(1);

ASSERT\_NO\_THROW(b.Delete(1));

}

TEST(RBTree, can\_delete\_node\_with\_one\_child) {

RBTree<int> b(7);

ASSERT\_NO\_THROW(b.Insert(3));

ASSERT\_NO\_THROW(b.Insert(5));

ASSERT\_NO\_THROW(b.Insert(1));

ASSERT\_NO\_THROW(b.Insert(2));

ASSERT\_NO\_THROW(b.Insert(4));

ASSERT\_NO\_THROW(b.Find(5));

ASSERT\_NO\_THROW(b.Delete(5));

ASSERT\_ANY\_THROW(b.Find(5));

}

TEST(RBTree, can\_delete\_node\_with\_two\_child) {

RBTree<int> b(7);

ASSERT\_NO\_THROW(b.Insert(3));

ASSERT\_NO\_THROW(b.Insert(5));

ASSERT\_NO\_THROW(b.Insert(1));

ASSERT\_NO\_THROW(b.Insert(2));

ASSERT\_NO\_THROW(b.Insert(4));

ASSERT\_NO\_THROW(b.Find(3));

ASSERT\_NO\_THROW(b.Delete(3));

ASSERT\_ANY\_THROW(b.Find(3));

}

TEST(RBTree, can\_find\_min) {

RBTree<int> b(7);

b.Insert(3);

b.Insert(5);

b.Insert(1);

b.Insert(2);

b.Insert(4);

b.Insert(6);

rbnode<int>\* res = b.Min();

EXPECT\_EQ(1, res->val);

}

TEST(RBTree, IsRBTree\_work\_right\_1) {

RBTree<int> b;

EXPECT\_TRUE(b.IsRBTree());

}

TEST(RBTree, IsRBTree\_work\_right\_2) {

RBTree<int> b(7);

EXPECT\_TRUE(b.IsRBTree());

}

TEST(RBTree, IsRBTree\_work\_right\_3) {

RBTree<int> b(7);

b.GetRoot()->left = new rbnode<int>;

b.GetRoot()->left->color = 1;

b.GetRoot()->right = new rbnode<int>;

b.GetRoot()->right->color = 1;

b.GetRoot()->left->left = NIL;

b.GetRoot()->left->right = NIL;

b.GetRoot()->right->left = NIL;

b.GetRoot()->right->right = NIL;

EXPECT\_TRUE(b.IsRBTree());

}

TEST(RBTree, remain\_RBTree\_after\_insert) {

RBTree<int> b(7);

b.Insert(3);

b.Insert(5);

b.Insert(1);

b.Insert(2);

b.Insert(4);

b.Insert(6);

EXPECT\_TRUE(b.IsRBTree());

}

TEST(RBTree, remain\_RBTree\_after\_delete) {

RBTree<int> b(7);

b.Insert(3);

b.Insert(5);

b.Insert(1);

b.Insert(2);

b.Insert(4);

b.Insert(6);

b.Delete(2);

EXPECT\_TRUE(b.IsRBTree());

}

# Приложение 11. test\_FourHeap.cpp

#include <gtest/gtest.h>

#include "FourHeap.h"

TEST(FourHeap, can\_create\_heap) {

ASSERT\_NO\_THROW(FourHeap<int> h);

}

TEST(FourHeap, can\_insert\_in\_heap) {

FourHeap<int> h;

ASSERT\_NO\_THROW(h.Insert(2));

ASSERT\_NO\_THROW(h.Insert(5));

ASSERT\_NO\_THROW(h.Insert(1));

}

TEST(FourHeap, can\_find\_min) {

FourHeap<int> h;

h.Insert(2);

h.Insert(5);

h.Insert(1);

EXPECT\_EQ(1, h.Min());

}

TEST(FourHeap, can\_delete\_min) {

FourHeap<int> h;

h.Insert(2);

h.Insert(5);

h.Insert(1);

ASSERT\_NO\_THROW(h.DeleteMin());

EXPECT\_EQ(2, h.Min());

}

TEST(FourHeap, can\_check\_empty) {

FourHeap<int> h;

EXPECT\_TRUE(h.empty());

}

# Приложение 12. test\_DijkstraRBTree.cpp

#include <gtest/gtest.h>

#include <Dijkstra.h>

#include <vector>

#include <utility>

#define pair std::pair

#define vector std::vector

TEST(DijkstraRBTree, throws\_when\_parametrs\_are\_incorrect) {

vector < vector < pair<int, int> > > gr1;

EXPECT\_ANY\_THROW(vector <int> res = DijkstraRBTree(gr1, 1));

vector < vector < pair<int, int> > > gr2(7);

EXPECT\_ANY\_THROW(vector <int> res = DijkstraRBTree(gr2, 8));

}

TEST(DijkstraRBTree, work\_right\_for\_graph\_with\_one\_vertex) {

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

vector <int> res = DijkstraRBTree(gr, 1);

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

}

TEST(DijkstraRBTree, work\_right\_for\_connected\_graph) {

char filename[] = "COPYONLY";

vector < vector < pair<int, int> > > gr = CreateGraph(filename);

vector <int> res = DijkstraRBTree(gr, 1);

int exp[] = {0, 4, 6, 5, 8, 6, 8};

for (int i = 0; i < 7; ++i)

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

}

TEST(DijkstraRBTree, work\_right\_for\_disconnected\_graph) {

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

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

vector <int> res = DijkstraRBTree(gr, 1);

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

EXPECT\_EQ(res[1], 5);

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

}

# Приложение 13. test\_DijkstraFourHeap.cpp

#include <gtest/gtest.h>

#include <Dijkstra.h>

#include <vector>

#include <utility>

#define pair std::pair

#define vector std::vector

TEST(DijkstraFourHeap, throws\_when\_parametrs\_are\_incorrect) {

vector < vector < pair<int, int> > > gr1;

EXPECT\_ANY\_THROW(vector <int> res = DijkstraFourHeap(gr1, 1));

vector < vector < pair<int, int> > > gr2(7);

EXPECT\_ANY\_THROW(vector <int> res = DijkstraFourHeap(gr2, 8));

}

TEST(DijkstraFourHeap, work\_right\_for\_graph\_with\_one\_vertex) {

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

vector <int> res = DijkstraFourHeap(gr, 1);

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

}

TEST(DijkstraFourHeap, work\_right\_for\_connected\_graph) {

char filename[] = "COPYONLY";

vector < vector < pair<int, int> > > gr = CreateGraph(filename);

vector <int> res = DijkstraFourHeap(gr, 1);

int exp[] = {0, 4, 6, 5, 8, 6, 8};

for (int i = 0; i < 7; ++i)

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

}

TEST(DijkstraFourHeap, work\_right\_for\_disconnected\_graph) {

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

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

vector <int> res = DijkstraFourHeap(gr, 1);

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

EXPECT\_EQ(res[1], 5);

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

}