Федеральное государственное автономное образовательное учреждение

высшего образования

«Нижегородский государственный университет

им. Н.И. Лобачевского»

Институт Информационных Технологий, Математики и Механики

Отчёт по лабораторной работе

Реализация алгоритма Дейкстры с помощью приоритетной очереди

Выполнил:

студент ф-та ИИТММ гр. 0823-1

Ванеев Н.Ю.

Проверил:

к.т.н., ассистент каф. ПрИнж ИИТММ

Сиднев А.А.

Нижний Новгород

2017 г.

Содержание

[Введение 3](file:///D:\plak\3laba\Отчет.docx#_Toc405754030)

[Постановка задачи 4](file:///D:\plak\3laba\Отчет.docx#_Toc405754031)

[Руководство пользователя 5](file:///D:\plak\3laba\Отчет.docx#_Toc405754032)

[Руководство программиста 6](file:///D:\plak\3laba\Отчет.docx#_Toc405754033)

[Описание структуры программы 6](file:///D:\plak\3laba\Отчет.docx#_Toc405754034)

[Описание структур данных 6](file:///D:\plak\3laba\Отчет.docx#_Toc405754035)

[Описание алгоритмов 8](file:///D:\plak\3laba\Отчет.docx#_Toc405754036)

[Заключение 9](file:///D:\plak\3laba\Отчет.docx#_Toc405754037)

[Литература 10](file:///D:\plak\3laba\Отчет.docx#_Toc405754038)

[Приложения 11](file:///D:\plak\3laba\Отчет.docx#_Toc405754039)

[Приложение 1 11](file:///D:\plak\3laba\Отчет.docx#_Toc405754040)

[Приложение 2 11](file:///D:\plak\3laba\Отчет.docx#_Toc405754041)

[Приложение 3 12](file:///D:\plak\3laba\Отчет.docx#_Toc405754042)

# Введение

Выполненная лабораторная работа направлена на изучение алгоритма Дейкстры и его реализацию с помощью приоритетной очереди. В свою очередь приоритетная очередь была реализована двумя структурами данных. С помощью AVL-дерева и 3-кучи

Например, алгоритм Дейкстры помогает решать некоторые жизненные задачи:

Имеется некоторое количество авиарейсов между городами мира, для каждого известна стоимость. Стоимость перелёта из A в B может быть не равна стоимости перелёта из B в A. Найти маршрут минимальной стоимости (возможно, с пересадками) от Москвы до Бостона.

 Дана сеть автомобильных дорог, соединяющих города Московской области. Некоторые дороги односторонние. Найти кратчайшие пути от Москвы до каждого города области (если двигаться можно только по дорогам).

# 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. Пользователь указывает стартовую вершину, от которой программа будет искать пути до других вершин графа.

Алгоритм Дейкстры реализован с помощью приоритетной очереди. В свою очередь приоритетную очередь реализована с помощью 2-х структур данных.

* AVL-дерево
* 3-куча.

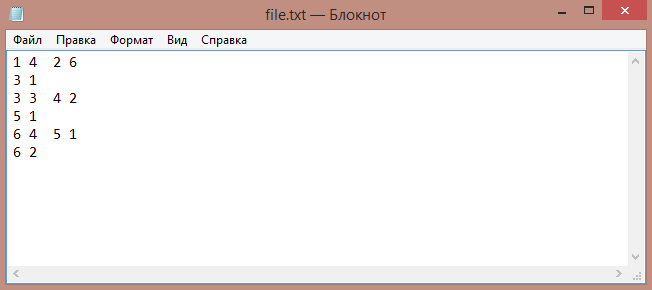
Реализованные тесты проверяют корректность работы алгоритма и используемых структур данных:

* Корректность Алгоритма Дейкстры;
* Корректность AVL-дерева;
* Корректность 3-кучи.

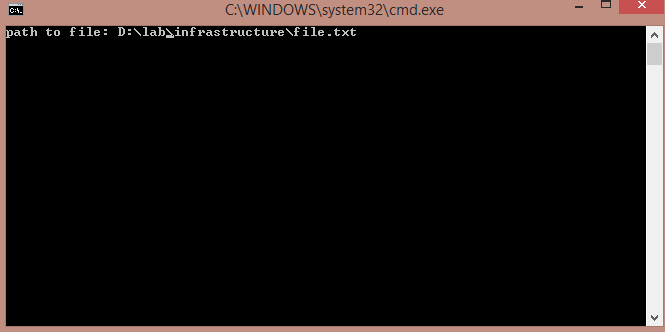
# Руководство пользователя

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

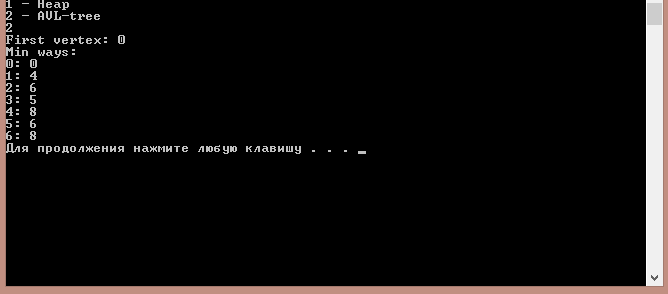
1. Создать файл, в котором будет граф заданный списком смежности.



1. Указать полный путь до этого файла

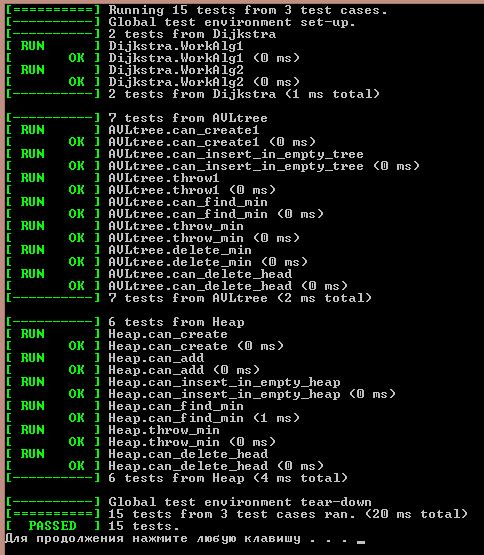


1. Выбрать метод и начальную вершину



В результате на экран будут выведены минимальные расстояние от заданной вершины до всех остальных.

Так же тесты подтверждают корректность реализации алгоритма.



# Руководство программиста

## Описание структуры программы

Программа состоит из следующих составных частей:

* Реализация Алгоритма Дейкстры.
* Реализация структуры данных AVL-дерево
* Реализация структуры данных 3-куча;
* Модуль test.cpp.

Данный модуль содержит тесты для проверки программы.

## Описание структур данных

В программе реализована структура данных — **3-куча.**

**3-Ку́ча** — это специализированная структура данных типа дерево, которая удовлетворяет свойству кучи: если B является узлом-потомком узла A, то ключ(A) ≥ ключ(B). Из этого следует, что элемент с наибольшим ключом всегда является корневым узлом кучи

Используется контейнер vector<T>. Поле head содержит в себе всю кучу.

Методы void shift\_up(int a); void shift\_down(int b); Используются для балансировки кучи.

Public методы:

min – возвращает значение в корне кучи.

add – добавляет узел в кучу.

delmin – удаляет минимальный.

empty – проверяет пуста ли куча.

Вторая структура данных в программе это **AVL-Дерево**

**AVL-дерево** — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

Используется структура данных CNode

template <typename T>

struct CNode {

T val;

int h;

CNode\* left;

CNode\* right;

CNode\* parent;

explicit CNode(T a = 0, int b = 1, CNode\* par = nullptr) : val(a), h(b),

left(nullptr), right(nullptr), parent(par) {}

};

CNode<T>\* head; - Ссылка на голову дерева

Методы которые нужны для балансировки дерева

private:

{

CNode<T>\* bal(CNode<T>\*);

CNode<T>\* rightrot(CNode<T>\*);

CNode<T>\* leftrot(CNode<T>\*);

};

public:

void ins(T); - Вставка по значению

void delmin(); -Удаление минимального элемента

T min(); - Минимальное значние узла

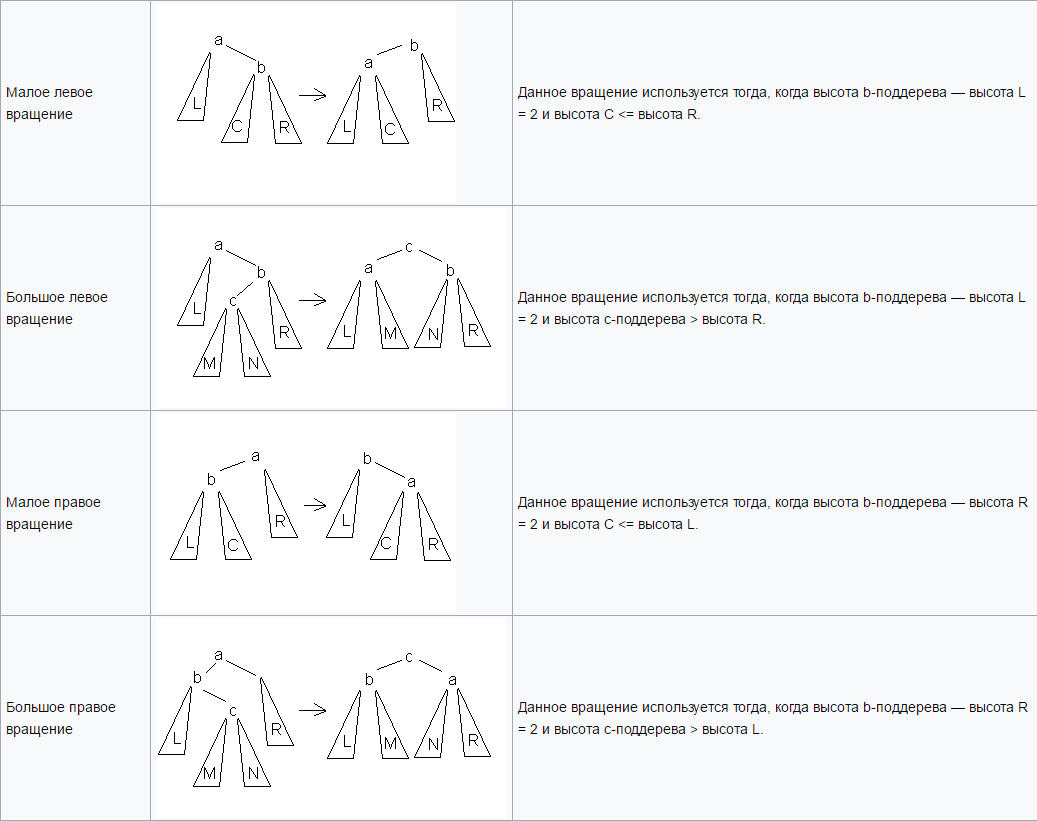
bool empty(); -Проверка дерева на пустоту

## Описание алгоритмов

**AVL-дерево**

**Балансировка**

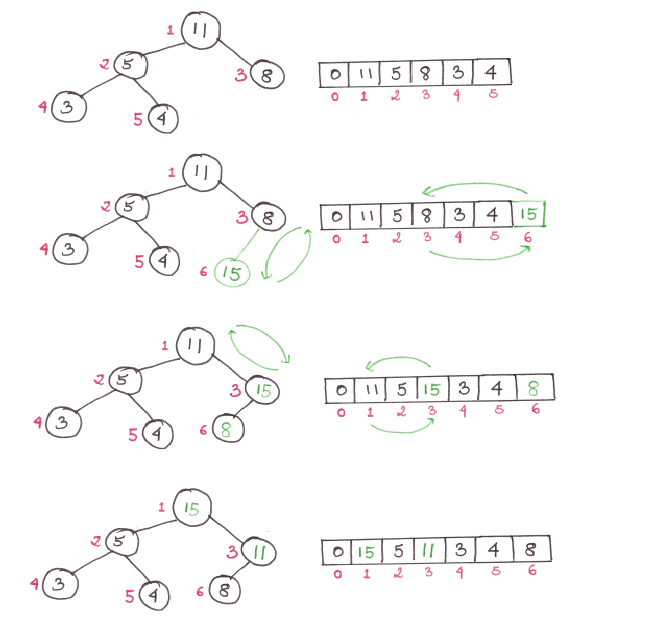
Относительно AVL-дерева балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев = 2, изменяет связи предок-потомок в поддереве данной вершины так, что разница становится <= 1, иначе ничего не меняет. Указанный результат получается вращениями поддерева данной вершины.



Из-за условия сбалансированности высота дерева О(log(N)), где N- количество вершин, поэтому добавление элемента требует O(log(N)) операций.

**Куча**

**Добавление элемента в существующую кучу**  
Для начала, мы добавляем элемент в самый низ кучи, т.е. в конец массива. Затем мы меняем его местами с родительским элементом до тех пор, пока он не встанет на свое место.

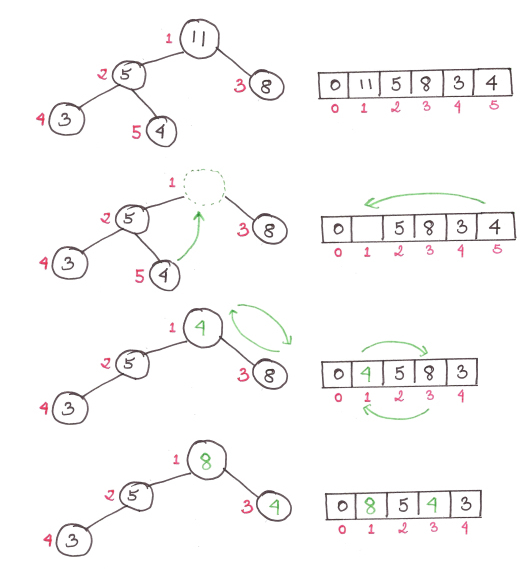


Алгоритм:

1. Добавляем элемент в самый низ кучи.
2. Сравниваем добавленный элемент с родительским; если порядок верный — останавливаемся.
3. Если нет — меняем элементы местами, и возвращаемся к предыдущему пункту.

**Удаление максимального элемента**

Первый элемент в куче — всегда максимальный, так что мы просто удалим его (предварительно запомнив), и заменим самым нижним. Затем мы приведем кучу в правильный порядок, используя функцию:



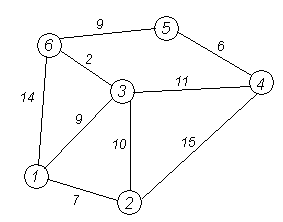
Алгоритм:

1. Заменить корневой элемент самым нижним.
2. Сравнить новый корневой элемент с дочерними. Если они в правильном порядке — остановиться.
3. Если нет — заменить корневой элемент на одного из дочерних (меньший для min-heap, больший для max-heap), и повторить шаг 2.

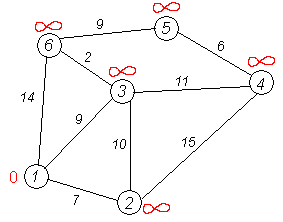
# Алгоритм Дейкстры

Рассмотрим выполнение алгоритма на примере графа, показанного на рисунке.

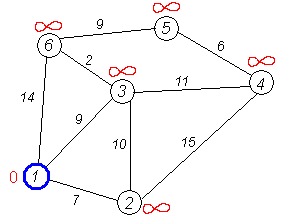
Пусть требуется найти кратчайшие расстояния от 1-й вершины до всех остальных.

[](https://commons.wikimedia.org/wiki/File:Dijkstra_graph0.PNG?uselang=ru)

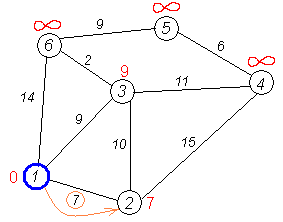
Кружками обозначены вершины, линиями — пути между ними (рёбра графа). В кружках обозначены номера вершин, над рёбрами обозначен их вес — длина пути. Рядом с каждой вершиной красным обозначена метка — длина кратчайшего пути в эту вершину из вершины 1.

[](https://commons.wikimedia.org/wiki/File:Dijkstra_graph1.PNG?uselang=ru)

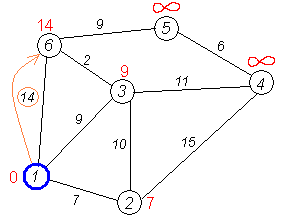
**Первый шаг**. Рассмотрим шаг алгоритма Дейкстры для нашего примера. Минимальную метку имеет вершина 1. Её соседями являются вершины 2, 3 и 6.

[](https://commons.wikimedia.org/wiki/File:Dijkstra_graph2.PNG?uselang=ru)

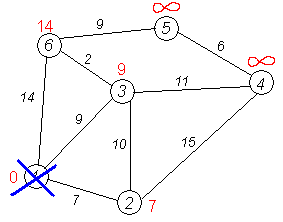
Первый по очереди сосед вершины 1 — вершина 2, потому что длина пути до неё минимальна. Длина пути в неё через вершину 1 равна сумме значения метки вершины 1 и длины ребра, идущего из 1-й в 2-ю, то есть 0 + 7 = 7. Это меньше текущей метки вершины 2, бесконечности, поэтому новая метка 2-й вершины равна 7.

[](https://commons.wikimedia.org/wiki/File:Dijkstra_graph3.PNG?uselang=ru)

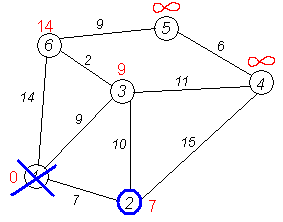
Аналогичную операцию проделываем с двумя другими соседями 1-й вершины — 3-й и 6-й.

[](https://commons.wikimedia.org/wiki/File:Dijkstra_graph5.PNG?uselang=ru)

Все соседи вершины 1 проверены. Текущее минимальное расстояние до вершины 1 считается окончательным и пересмотру не подлежит (то, что это действительно так, впервые доказал Дейкстра)Вычеркнем её из графа, чтобы отметить, что эта вершина посещена.

[](https://commons.wikimedia.org/wiki/File:Dijkstra_graph6.PNG?uselang=ru)

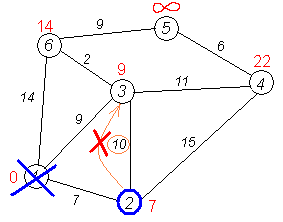
**Второй шаг**. Шаг алгоритма повторяется. Снова находим «ближайшую» из непосещённых вершин. Это вершина 2 с меткой 7.

[](https://commons.wikimedia.org/wiki/File:Dijkstra_graph7.PNG?uselang=ru)

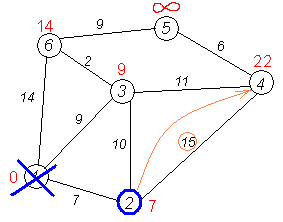
Снова пытаемся уменьшить метки соседей выбранной вершины, пытаясь пройти в них через 2-ю вершину. Соседями вершины 2 являются вершины 1, 3 и 4.

Первый (по порядку) сосед вершины 2 — вершина 1. Но она уже посещена, поэтому с 1-й вершиной ничего не делаем.

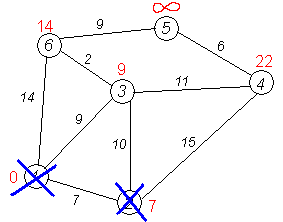
Следующий сосед вершины 2 — вершина 3, так как имеет минимальную метку из вершин, отмеченных как не посещённые. Если идти в неё через 2, то длина такого пути будет равна 17 (7 + 10 = 17). Но текущая метка третьей вершины равна 9, а это меньше 17, поэтому метка не меняется.

[](https://commons.wikimedia.org/wiki/File:Dijkstra_graph9.PNG?uselang=ru)

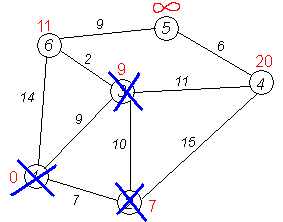
Ещё один сосед вершины 2 — вершина 4. Если идти в неё через 2-ю, то длина такого пути будет равна сумме кратчайшего расстояния до 2-й вершины и расстояния между вершинами 2 и 4, то есть 22 (7 + 15 = 22). Поскольку 22<{\displaystyle \infty }inf, устанавливаем метку вершины 4 равной 22.

[](https://commons.wikimedia.org/wiki/File:Dijkstra_graph8.PNG?uselang=ru)

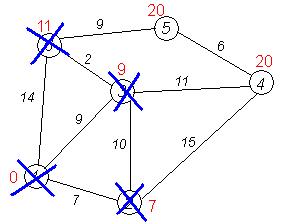
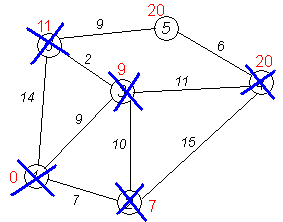
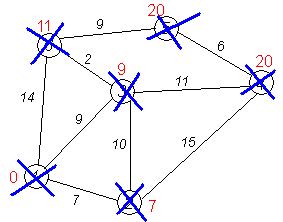
Все соседи вершины 2 просмотрены, замораживаем расстояние до неё и помечаем её как посещённую.

[](https://commons.wikimedia.org/wiki/File:Dijkstra_graph10.PNG?uselang=ru)

**Третий шаг**. Повторяем шаг алгоритма, выбрав вершину 3. После её «обработки» получим такие результаты:

[](https://commons.wikimedia.org/wiki/File:Dijkstra_graph11.PNG?uselang=ru)

**Дальнейшие шаги**. Повторяем шаг алгоритма для оставшихся вершин. Это будут вершины 6, 4 и 5, соответственно порядку.

[](https://commons.wikimedia.org/wiki/File:Dijkstra_graph12.PNG?uselang=ru) [](https://commons.wikimedia.org/wiki/File:Dijkstra_graph13.PNG?uselang=ru) [](https://commons.wikimedia.org/wiki/File:Dijkstra_graph14.PNG?uselang=ru)

**Завершение выполнения алгоритма**. Алгоритм заканчивает работу, когда нельзя больше обработать ни одной вершины. В данном примере все вершины зачёркнуты, однако ошибочно полагать, что так будет в любом примере — некоторые вершины могут остаться не зачёркнутыми, если до них нельзя добраться, т. е. если граф несвязный. Результат работы алгоритма виден на последнем рисунке: кратчайший путь от вершины 1 до 2-й составляет 7, до 3-й — 9, до 4-й — 20, до 5-й — 20, до 6-й — 11.

# Заключение

Реализован алгоритм Дейкстры поиска кратчайшего пути с использованием приоритетной очереди. Реализованы AVL-дерево и  3-куча. Реализованные тесты показывают корректность выполнения программы.

# Литература

1. Алгоритм Дейкстры — [https://ru.wikipedia.org/wiki/Алгоритм\_Дейкстры]
2. 3-куча [https://habrahabr.ru/post/263765/]
3. AVL-дерево [https://ru.wikipedia.org/wiki/АВЛ-дерево]
4. 3-куча [https://ru.wikipedia.org/wiki/Куча\_(структура\_данных)]

# Приложения

## Приложение 1

Реализация AVL- дерева

#ifndef INCLUDE\_AVL\_H\_

#define INCLUDE\_AVL\_H\_

#include <stack>

#include <stdexcept>

template <typename T>

struct CNode {

T val;

int h;

CNode\* left;

CNode\* right;

CNode\* parent;

explicit CNode(T a = 0, int b = 1, CNode\* par = nullptr) : val(a), h(b),

left(nullptr), right(nullptr), parent(par) {}

};

template <typename T>

class avltree {

CNode<T>\* head;

private:

CNode<T>\* bal(CNode<T>\*);

CNode<T>\* rightrot(CNode<T>\*);

CNode<T>\* leftrot(CNode<T>\*);

int diff(CNode<T>\*);

void fixtree(CNode<T>\*);

public:

explicit avltree(T a = T()) {

if (a == T())

head = nullptr;

else

head = new CNode<T>(a);

}

void ins(T);

void delmin();

T min();

bool empty();

};

template <typename T>

bool avltree<T>::empty() {

if (head)

return false;

else

return true;

}

template <typename T>

void avltree<T>::fixtree(CNode<T>\* a) {

CNode<T>\* par = a;

int h1, h2;

bool change = false;

while (par != nullptr) {

if ((diff(par) == 2) || diff(par) == -2) {

if (par == head)

change = true;

par = bal(par);

if (change)

head = par;

}

if (par->left)

h1 = par->left->h;

else

h1 = 0;

if (par->right)

h2 = par->right->h;

else

h2 = 0;

par->h = (h1 > h2 ? h1 : h2) + 1;

par = par->parent;

}

}

template <typename T>

int avltree<T>::diff(CNode<T>\* node) {

if ((node->left == nullptr) && (node->right == nullptr))

return 0;

if (node->left == nullptr)

return (-node->right->h);

if (node->right == nullptr)

return (node->left->h);

return (node->left->h - node->right->h);

}

template <typename T>

void avltree<T>::ins(T a) {

CNode<T>\*\* tmp = &head;

if (\*tmp == nullptr) {

head = new CNode<T>(a);

return;

}

CNode<T>\*\* tmp1;

while (\*tmp != nullptr) {

if (a == (\*tmp)->val)

throw std::logic\_error("eror");

if (a < (\*tmp)->val) {

tmp1 = tmp;

tmp = &((\*tmp)->left);

} else {

tmp1 = tmp;

tmp = &((\*tmp)->right);

}

}

\*tmp = new CNode<T>(a, 1, \*tmp1);

fixtree(\*tmp);

}

template <typename T>

CNode<T>\* avltree<T>::bal(CNode<T>\* node) {

int dif = diff(node);

if (dif == -2) {

dif = diff(node->right);

if (dif > 0)

node->right = rightrot(node->right);

node = leftrot(node);

} else if (dif == 2) {

dif = diff(node->left);

if (dif < 0)

node->left = leftrot(node->left);

node = rightrot(node);

}

return node;

}

template <typename T>

CNode<T>\* avltree<T>::rightrot(CNode<T>\* node) {

CNode<T>\* tmp = node->left;

node->left = tmp->right;

if (node->left)

node->left->parent = node;

tmp->right = node;

tmp->parent = node->parent;

CNode<T>\* par = tmp->parent;

if (par) {

if (par->left == node)

tmp->parent->left = tmp;

else

tmp->parent->right = tmp;

}

node->parent = tmp;

int h1, h2;

if (node->left != 0) {

h1 = node->left->h;

} else {

h1 = 0;

}

if (node->right != 0) {

h2 = node->right->h;

} else {

h2 = 0;

}

node->h = (h1 > h2 ? h1 : h2) + 1;

if (tmp->left)

h1 = tmp->left->h;

else

h1 = 0;

if (tmp->right)

h2 = tmp->right->h;

else

h2 = 0;

tmp->h = (h1 > h2 ? h1 : h2) + 1;

return tmp;

}

template <typename T>

CNode<T>\* avltree<T>::leftrot(CNode<T>\* node) {

CNode<T>\* tmp = node->right;

node->right = tmp->left;

if (node->right)

node->right->parent = node;

tmp->left = node;

tmp->parent = node->parent;

CNode<T>\* par = tmp->parent;

if (par) {

if (par->left == node)

tmp->parent->left = tmp;

else

tmp->parent->right = tmp;

}

node->parent = tmp;

int h1, h2;

if (node->left != 0) {

h1 = node->left->h;

} else {

h1 = 0;

}

if (node->right != 0) {

h2 = node->right->h;

} else {

h2 = 0;

}

node->h = (h1 > h2 ? h1 : h2) + 1;

if (tmp->left)

h1 = tmp->left->h;

else

h1 = 0;

if (tmp->right)

h2 = tmp->right->h;

else

h2 = 0;

tmp->h = (h1 > h2 ? h1 : h2) + 1;

return tmp;

}

template <typename T>

void avltree<T>::delmin() {

CNode<T>\* tmp = head;

if (!head)

throw std::logic\_error("Tree is empty!");

while (tmp->left != nullptr) {

tmp = tmp->left;

}

if (tmp->parent) {

if (tmp->right)

delete tmp->right;

tmp = tmp->parent;

delete tmp->left;

tmp->left = nullptr;

fixtree(tmp);

} else {

head = tmp->right;

if (head)

head->parent = nullptr;

delete tmp;

}

}

template <typename T>

T avltree<T>::min() {

if (!head)

throw std::logic\_error("empty!");

CNode<T>\* tmp = head;

while (tmp->left != nullptr) {

tmp = tmp->left;

}

return tmp->val;

}

#endif // INCLUDE\_AVL\_H\_

## Приложение 2

Реализация 3-кучи:

#ifndef INCLUDE\_HEAP\_H\_

#define INCLUDE\_HEAP\_H\_

#include <vector>

#include <stdexcept>

using std::vector;

using std::pair;

template <typename T>

class heap {

vector<T> head;

void shift\_up(int a);

void shift\_down(int b);

public:

T min();

void add(T);

void delmin();

bool empty();

};

template <typename T>

bool heap<T>::empty() {

return head.empty();

}

template <typename T>

void heap<T>::add(T pr) {

head.push\_back(pr);

shift\_up(head.size() - 1);

}

template <typename T>

T heap<T>::min() {

if (!head.empty())

return head[0];

else

throw std::logic\_error("empty!");

}

template <typename T>

void heap<T>::delmin() {

if (!head.empty()) {

head[0] = head[head.size() - 1];

head.erase(head.end() - 1);

shift\_down(0);

} else {

throw std::logic\_error("empty!");

}

}

template <typename T>

void heap<T>::shift\_up(int pos) {

int pospar = (pos - 1) / 3;

while (head[pospar] > head[pos]) {

std::swap(head[pospar], head[pos]);

pos = pospar;

pospar = (pospar - 1) / 3;

}

}

template <typename T>

void heap<T>::shift\_down(int pos) {

int end = head.size();

while (3\*pos + 1 < end) {

T min = head[3 \* pos + 1];

int posmin = 3 \* pos + 1;

for (int i = 2; (i < 4), (3 \* pos + i) < end; i++) {

if (head[3 \* pos + i] < min) {

min = head[3 \* pos + i];

posmin = 3 \* pos + i;

}

}

std::swap(head[pos], head[posmin]);

pos = posmin;

}

}

#endif // INCLUDE\_HEAP\_H\_

***Приложение 3***

Реализация алгоритма Дейкстры:

#include <vector>

#include <utility>

#include "avl.h"

#include "pr\_queue\_t.h"

#include <limits.h>

using std::vector;

using std::pair;

vector<int> Dijkstra\_Heap(vector< vector< pair<int, int> > > ls, int a) {

if (a >= static\_cast<int>(ls.size())) {

throw std::logic\_error("Vertex!");

}

pr\_queue\_h <pair <int, int> > T;

bool \*aw = new bool[ls.size()];

for (unsigned int i = 0; i < ls.size(); i++)

aw[i] = true;

vector<int> ret(ls.size(), INT\_MAX);

auto t = ls[a];

for (auto it = t.begin(); it != t.end(); it++) {

T.add(\*it);

ret[it->second] = it->first;

}

ret[a] = 0;

aw[a] = false;

while (!T.empty()) {

int vs = T.min().second;

T.delmin();

if (aw[vs] != false) {

aw[vs] = false;

t = ls[vs];

for (auto it = t.begin(); it != t.end(); it++) {

if (it->first + ret[vs] < ret[it->second])

ret[it->second] = it->first + ret[vs];

T.add(std::make\_pair(it->first + ret[vs], it->second));

}

}

}

return ret;

}

vector<int> Dijkstra\_Tree(vector< vector< pair<int, int> > > ls, int a) {

if (a >= static\_cast<int>(ls.size())) {

throw std::logic\_error("Vertex!");

}

pr\_queue\_t <pair <int, int> > T;

bool \*aw = new bool[ls.size()];

for (unsigned int i = 0; i < ls.size(); i++)

aw[i] = true;

vector<int> ret(ls.size(), INT\_MAX);

auto t = ls[a];

for (auto it = t.begin(); it != t.end(); it++) {

T.ins(\*it);

ret[it->second] = it->first;

}

ret[a] = 0;

aw[a] = false;

while (!T.empty()) {

int vs = T.min().second;

T.delmin();

if (aw[vs] != false) {

aw[vs] = false;

t = ls[vs];

for (auto it = t.begin(); it != t.end(); it++) {

if (it->first + ret[vs] < ret[it->second])

ret[it->second] = it->first + ret[vs];

T.ins(std::make\_pair(it->first + ret[vs], it->second));

}

}

}

return ret;

}