Балтийский Государственный технический университет «ВОЕНМЕХ» им. Д. Ф. Устинова

Кафедра И5

«Информационные системы и программная инженерия»

Практическое задание №2

по дисциплине «Структуры данных»

На тему **«Нелинейные структуры данных»**

**Выполнил:**

Студент : Дубровский В.И.

Группа: И582

**Преподаватель:**

Палехова О. А.

Санкт-Петербург

1. г.

**8 вариант.**

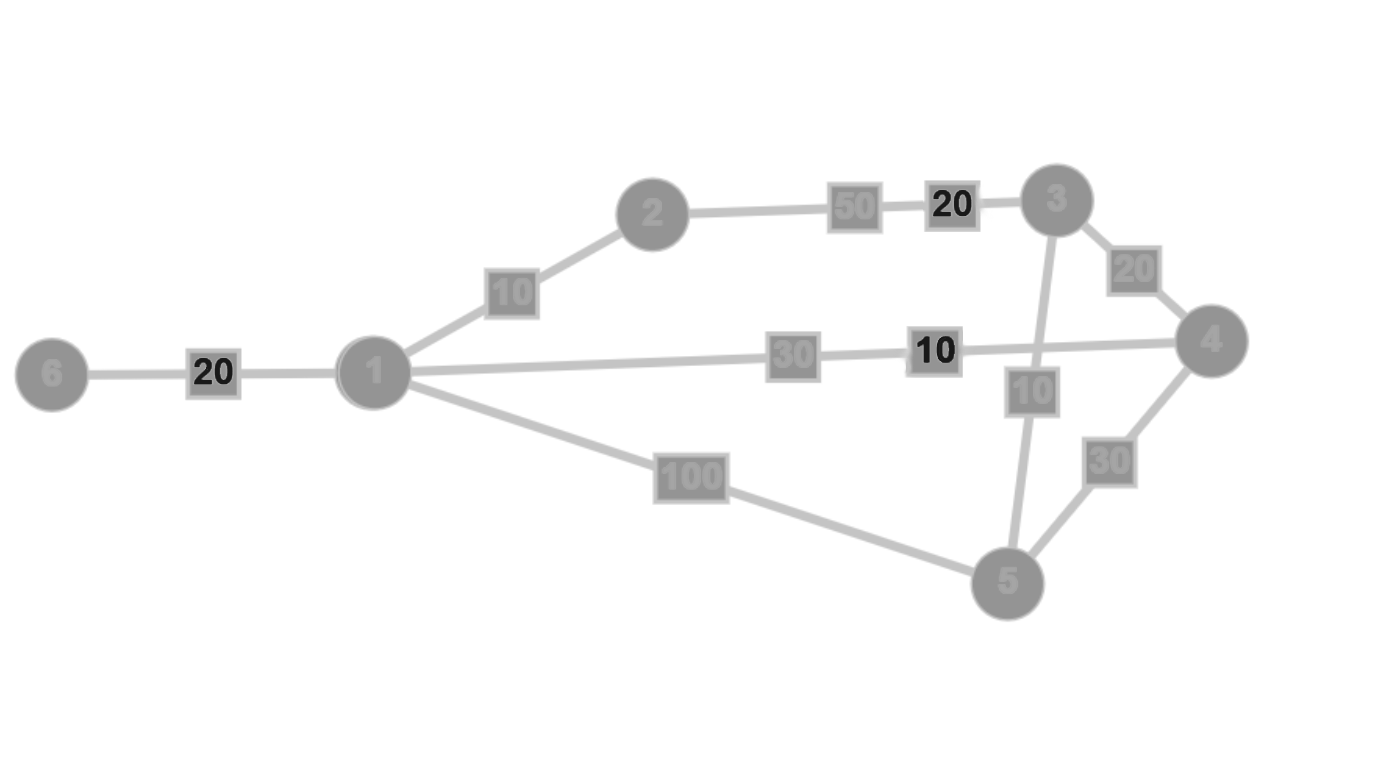
Постановка задачи:написать две программы согласно номеру индивидуального варианта. Использовать при работе с графами и деревьями рекурсивные алгоритмы обработки данных, если они упрощают решение задачи.

*Задание 1:*

Граф реализовать двумя способами (матрицей весов и списками смежности). Все структуры данных оформить в виде классов. Все данные считывать из файлов.

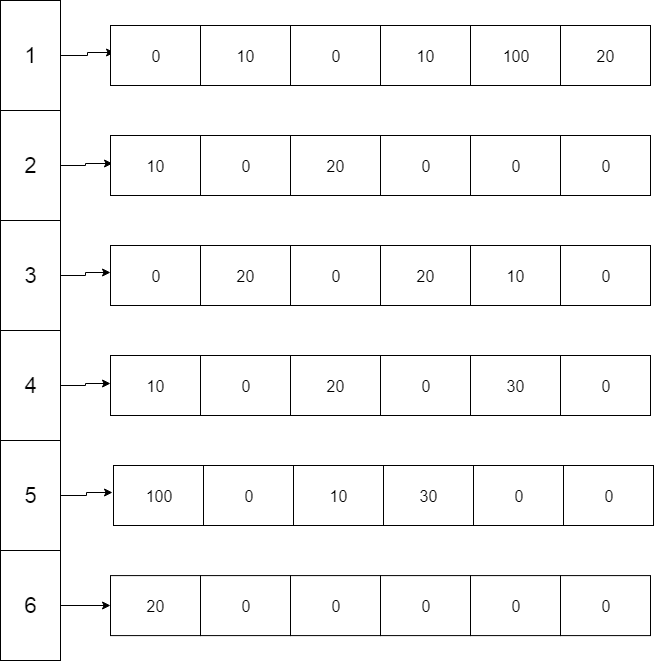
Заданы две системы двусторонних дорог с одним и тем же множеством городов (железные и шоссейные дороги). Найти минимальный по длине путь из города А в город В (который может проходить как по железным, так и по шоссейным дорогам), и места пересадок с одного вида транспорта на другой на этом пути. А и В вводятся с клавиатуры.

*Структура данных – неориентированный взвешенный граф (нелинейная):*

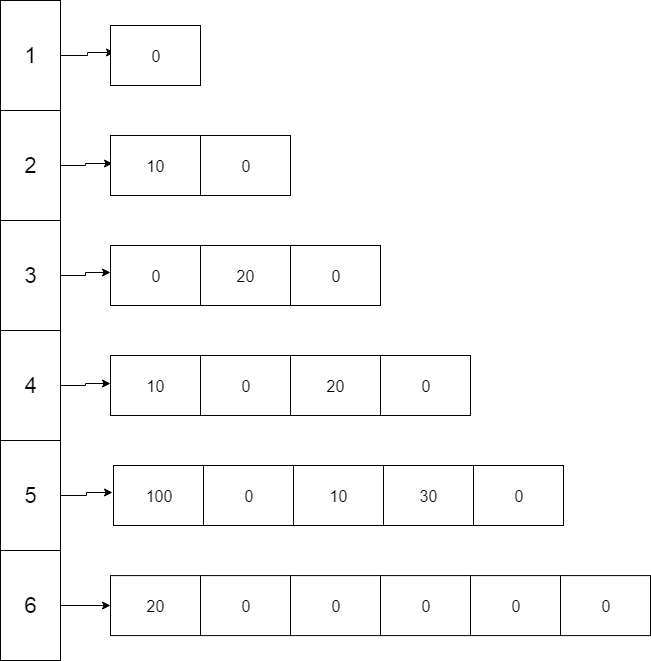
*Граф* – это набор вершин (узлов) и соединяющих их ребер (дуг). *Неориентированный граф* – это граф, в котором ребра не имеют направления. *Взвешенный граф* - это граф, некоторым элементам которого (вершинам, ребрам или дугам) сопоставлены числа. **

*Структура хранения –матрица весов:*

*Матрица весов* – это матрица, элемент M[i][j] которой равен весу ребра, если существует ребро из вершины i в вершину j, и равен 0, если такого нет.

**

*Матрица весов.*

**

*Треугольная матрица весов.*

*Текст программы:*

#include "file\_func.h"

#include "graph.h"

#include <memory>

using namespace std;

/\* ERROR LOG:

1 - workdir is empthy

2 - bad structure of file

3 - error to open file

4 - graphs erros

\*/

template<class T>

void cin\_clear(T &x);

constexpr auto PATH = "C:\\Users\\Владислав\\source\\repos\\LAB2\_struct\_graph\\LAB2\_struct\_graph\\graphs";

constexpr auto EXTENSION = ".txt";

enum TYPE { ADJACENCY\_MATRIX ,TRIANGLE\_MATRIX, ADJACENCY\_LIST };

int main(int argc, char\* argv[])

{

try

{

bool quit = true;

auto files = make\_shared<dirfile>(PATH, EXTENSION); //new dirfile(PATH, EXTENSION);

enum TYPE init;

shared\_ptr<Graph> graph;

files->menu();

{

char ch\_type = files->read\_type();

if (ch\_type == 't' || ch\_type == 'T')

init = TRIANGLE\_MATRIX;

else if (ch\_type == 'l' || ch\_type == 'L')

init = ADJACENCY\_LIST;

else if (ch\_type == 'm' || ch\_type == 'M')

init = ADJACENCY\_MATRIX;

else if (ch\_type == 'f' || ch\_type == 'F') { //if was opened a FAQ file program will close

files->read\_whole();

cout << "Press any key to continue" << endl;

std::cin.clear(); std::cin.ignore(numeric\_limits<streamsize>::max(), '\n'); std::cin.get();

return 0;

}

else throw Myexception("Wrong structure of txt file\nReduct and try again!", 2);

}

size\_t towns = files->count\_town();

if (towns < 1) throw Myexception("Not enough towns", 2);

if (init == ADJACENCY\_MATRIX)

graph = make\_unique<MatrixNeoGraph>(towns);

else if (init == TRIANGLE\_MATRIX)

graph = make\_unique<TriangleMatrixNeoGraph>(towns);

else

graph = make\_unique<ListNeoGraph>(towns);

while (!files->from.eof())

{

if (files->read\_road()) {

if (files->road\_type()) // first type of roads

graph->add\_path(files->road\_data.from, files->road\_data.width, files->road\_data.to);

else // second type

graph->add\_path(files->road\_data.from, files->road\_data.width, files->road\_data.to, 1);

}

}

files.reset(); //reset files coz dirfile obj is not needed anymore

graph->print\_d();

char answer = 48;

bool floyd\_fl = true;

size\_t dijkastra\_fl=numeric\_limits<size\_t>::max(); //size\_t is unsigned type, so we can just put in var 0 or -value

while (answer != 'n' && answer != 'N') //to minimize a coincidence with A(town num) variable

{

size\_t A = 0, B = 0;

do {

cout << "Type Origin: ";

cin\_clear(A);

cout << "Type Destination: ";

cin\_clear(B);

if (A == B || A > towns || B > towns)

cout << "Wrong input!" << endl;

else break;

} while (true);

A--; B--;

if (init == ADJACENCY\_MATRIX)

{

if (floyd\_fl)

{

((MatrixNeoGraph\*)&(\*graph))->floyd\_find();

{

cout << endl;

for (int i = -33; i < 0; i++)

cout << static\_cast<char>(-78);

cout << endl;

cout << static\_cast<char>(-78) << "\t After algorithm \t" << static\_cast<char>(-78) << endl;

for (int i = -33; i < 0; i++)

cout << static\_cast<char>(-78);

cout << endl;

graph->print\_d();

}

floyd\_fl = false;

}

((MatrixNeoGraph\*)&(\*graph))->restore(A,B);

}

else if (init == TRIANGLE\_MATRIX) {

if (floyd\_fl) //for make floyd warshall algorithm work once

{

((TriangleMatrixNeoGraph\*)&(\*graph))->floyd\_find();

{

cout << endl;

for (int i = -33; i < 0; i++)

cout << static\_cast<char>(-78);

cout << endl;

cout << static\_cast<char>(-78) << "\t After algorithm \t" << static\_cast<char>(-78) << endl;

for (int i = -33; i < 0; i++)

cout << static\_cast<char>(-78);

cout << endl;

graph->print\_d();

}

floyd\_fl = false;

}

((TriangleMatrixNeoGraph\*)&(\*graph))->restore(A, B);

}

else {

if (A!=dijkastra\_fl) { //if node will be the same its not necessary to use dijkstra algorithm one more time

((ListNeoGraph\*) & (\*graph))->dijkstra\_find(A);

dijkastra\_fl = A;

}

((ListNeoGraph\*)& (\*graph))->restore(A, B);

}

cout << "Do u want to find another journey?(y/n)" << endl;

while ((cin >> answer) && !(answer == 'y' || answer == 'Y' || answer == 'n' || answer == 'N'));

}

std::cin.clear(); std::cin.ignore(numeric\_limits<streamsize>::max(), '\n'); std::cin.get();

return 0;

}

catch (Myexception& e)

{

cout << e.GetName() << endl;

cout << "Press any key to continue" << endl;

std::cin.clear(); std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n'); std::cin.get();

return e.GetVal();

};

}

template<class T>

void cin\_clear(T &x)

{

while (!(cin >> x))

{

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

cerr << "Wrong input!" << endl;

}

}

#include "Myexception.h"

#include <iostream>

#include <stack>

#include <vector>

using namespace std;

typedef unsigned int weight\_t;

//constexpr auto WEIGHT\_INF = std::numeric\_limits<weight\_t>::max();

constexpr auto WEIGHT\_INF = 100000;

constexpr auto SIZE\_INF = std::numeric\_limits<size\_t>::max();

graph.h

class Graph

{

protected:

const size\_t nodes\_s;

explicit Graph(const size\_t n) : nodes\_s(n) {}

public:

virtual ~Graph() = default;

/\*add node from first type of road\*/

virtual void add\_path(size\_t from ,weight\_t path, size\_t to) = 0;

/\*add node from second type of road

last arg is just stub to differ two overload functions\*/

virtual void add\_path(size\_t from, weight\_t path, size\_t to, int) = 0;

/\*print whole Graph\*/

virtual void print\_d(void) = 0;

/\*weight - weight of node

just make printing weight visual better\*/

static void print\_w(weight\_t weight)

{

if (weight == WEIGHT\_INF) {

std::cout << "NONE";

}

else {

std::cout << weight;

}

}

/\*return count of nodes\*/

size\_t size()

{

return nodes\_s;

};

};

*File\_func.h*

#include <iostream>

#include <fstream>

#include <filesystem>

#include <vector>

#include <sstream>

#include "Myexception.h"

namespace fl = std::experimental::filesystem;

class dirfile

{

public:

std::ifstream from; //in file stream

struct {

size\_t from;

unsigned int width;

size\_t to;

} road\_data;

private:

std::vector<std::string> path; //pathes of txt files in graphes dir

const fl::path workdir;

bool road\_t = true;

public:

//append to vector path pathes of files ended with ".finder"

dirfile(const char\* work\_path, const char\* finder);

~dirfile(void) {

from.close();

};

bool road\_type(void) { return road\_t; };

//read type of storage structure

char read\_type(void)

{

char c;

from.get(c);

while (from.get() != '\n'); //get another part of line to move ptr

return c;

};

//read count of towns

std::size\_t count\_town(void);

bool read\_road(void);

void read\_whole(void)

{

while (!from.eof())

{

std::string str;

getline(from, str);

std::cout << str << std::endl;

}

};

//out to screen files and suggest to choose one

void menu(void);

};

*File\_func.cpp*

#include "file\_func.h"

//dirfile::dirfile(std::string work\_path,std::string finder) : workdir(work\_path)

dirfile::dirfile(const char\* work\_path, const char\* finder) : workdir(work\_path)

{

for (const auto & entry : fl::directory\_iterator(workdir)) { //iterating all files in dir

std::size\_t txt\_pos = entry.path().string().rfind(finder); //finding typed format

if (txt\_pos != std::string::npos) //if file with right format is existing

path.push\_back(entry.path().string()); //add to vector

}

}

std::size\_t dirfile::count\_town(void)

{

std::string line;

std::getline(from, line);

std::size\_t towns = line.find("Towns:");

try {

if (towns == std::string::npos) throw Myexception("Wrong structure of txt file\nReduct and try again!", 2);

}

catch (Myexception &e)

{

throw;

};

std::stringstream count(line.substr(line.find(':')+1));

while ((count >> towns));

return towns;

}

bool dirfile::read\_road(void)

{

std::string path;

getline(from, path);

if (path == "Railway:")

road\_t = true;

else if (path == "Highway:") {

road\_t = false;

}

else if (path == "");

else {

std::stringstream num(path);

num >> road\_data.from;

num >> road\_data.width;

num >> road\_data.to;

road\_data.from--;

road\_data.to--;

return true;

}

return false;

}

void dirfile::menu(void)

{

int i = 0;

int c = 0;

std::cout << "Choose one to open." << std::endl

<< "Txt files from work dir:" << std::endl;

try {

if (path.empty()) throw Myexception("No txt files in path", 1); //if our dir empty throw to exception and exit from programme

}

catch (Myexception &e)

{

throw;

};

for (const auto& str : path)

{

i++;

std::size\_t filename = str.rfind("\\"); // find last \ for clear output

std::cout << i << ".\t" << str.substr(filename + 1) << std::endl;

}

while (true) //loop for choosing

{

std::cout << "Num->";

if(!(std::cin >> c)) //c is not digit

{

std::cin.clear();

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

std::cerr << "Wrong input!" << std::endl;

continue;

}

if (c<1 || c>i) { // c is not in interval

std::cerr << "No such file!" << std::endl;

}

else break;

}

try {

from.open(path[c-1]); //open choosed file

if (from.is\_open()) std::cout << "File was successfully open!" << std::endl;

else throw Myexception("Failure to open file!", 3);

}

catch (Myexception &e)

{

throw;

};

}

*Текст класса:*

class MatrixNeoGraph :public Graph

{

private:

weight\_t\*\* weight;

bool\*\* transfer;

size\_t\*\* re\_path;

public:

MatrixNeoGraph(size\_t n);

~MatrixNeoGraph();

void add\_path(size\_t from, weight\_t path, size\_t to);

void add\_path(size\_t from, weight\_t path, size\_t to, int);

void print\_d(void);

/\*floyd warshall algorithm

will find shorted pathes for all nodes in matrix\*/

void floyd\_find(void);

/\*restore path from A town to B\*/

void restore(size\_t from, size\_t to);

};

MatrixNeoGraph::MatrixNeoGraph(size\_t n) : Graph(n)

{

weight = new weight\_t\*[nodes\_s];

transfer = new bool\* [nodes\_s];

re\_path = new size\_t\* [nodes\_s];

for (size\_t i = 0; i < nodes\_s; i++)

{

weight[i] = new weight\_t[nodes\_s];

transfer[i] = new bool[nodes\_s];

re\_path[i] = new size\_t[nodes\_s];

for (size\_t j = 0; j < nodes\_s; j++)

{

if (i == j) weight[i][j] = 0;

else weight[i][j] = WEIGHT\_INF;

transfer[i][j] = false;

}

}

}

MatrixNeoGraph::~MatrixNeoGraph()

{

for (int i = 0; i < nodes\_s; i++)

{

delete[] weight[i];

delete[] transfer[i];

delete[] re\_path[i];

}

delete[] weight;

delete[] transfer;

delete[] re\_path;

}

void MatrixNeoGraph::add\_path(size\_t from, weight\_t path, size\_t to)

{

try {

if (from == to) throw Myexception("Attention!Path from one town to them!", 4);

if (from > nodes\_s || to > nodes\_s) throw Myexception("From or To more then size of graph", 4);

}

catch (Myexception & e)

{

throw;

};

weight[from][to] = path;

weight[to][from] = path;

}

void MatrixNeoGraph::add\_path(size\_t from, weight\_t path, size\_t to, int)

{

try {

if (from == to) throw Myexception("Attention!Path from one town to them!", 4);

if (from > nodes\_s || to > nodes\_s) throw Myexception("From or To more then size of graph", 4);

}

catch (Myexception & e)

{

throw;

};

if (weight[from][to] > path) //if weight from second type of road is smaller then add in matrix

{

weight[from][to] = path;

weight[to][from] = path;

transfer[from][to] = true;

transfer[to][from] = true;

}

}

void MatrixNeoGraph::print\_d(void)

{

cout << "\t GRAPH:" << endl;

cout << " ";

cout << static\_cast<char>(-77);

for (int i = 1; i <= nodes\_s; i++)

cout << i << '\t';

cout << endl;

cout << static\_cast<char>(-60);

cout << static\_cast<char>(-59);

int temp = (nodes\_s) \* 7;

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

cout << static\_cast<char>(-60);

cout << endl;

for (int i = 0; i < nodes\_s; i++)

{

cout << i + 1 << static\_cast<char>(-77);

for (int j = 0; j < nodes\_s; j++)

{

print\_w(weight[i][j]);

cout << '\t';

}

cout << endl;

}

}

void MatrixNeoGraph::floyd\_find()

{

// Floyd-Warshall algorithm.

for (size\_t i = 0; i < nodes\_s; i++) {

for (size\_t j = 0; j < nodes\_s; j++) {

if (weight[i][j] == WEIGHT\_INF || i == j)

re\_path[i][j] = SIZE\_INF;

else

re\_path[i][j] = i;

}

}

for (size\_t k = 0; k < nodes\_s; k++)

{

for(size\_t i = 0 ; i< nodes\_s; i++)

for (size\_t j = 0; j < nodes\_s; j++)

{

if (weight[i][k] + weight[k][j] < weight[i][j])

{

weight[i][j] = weight[i][k] + weight[k][j];

re\_path[i][j] = re\_path[k][i];

}

}

}

}

void MatrixNeoGraph::restore(size\_t from, size\_t to)

{

if (from > to)

{

size\_t temp = to;

to = from;

from = temp;

}

if (re\_path[from][to] == SIZE\_INF)

{

cout << "Path is not exist" << endl;

return;

}

cout << "\tPATH:\n";

size\_t quit = from;

bool flag = re\_path[quit][to]!= quit ? transfer[quit][re\_path[quit][to]] : transfer[quit][to];

cout << (flag == true ? "H:" : "R:") <<from + 1 << "-(" << (re\_path[quit][to]!=quit ? weight[quit][re\_path[quit][to]] : weight[quit][to]) << ")->";

while (re\_path[quit][to] != quit)

{

if (transfer[quit][re\_path[quit][to]] != flag)

{

cout << "(Transfer from " << ((flag == true) ? "Highway to Railway)->\nR:" : "Railway to Highway)->\nH:");

flag = transfer[quit][re\_path[quit][to]];

}

cout << re\_path[quit][to] + 1;

quit = re\_path[quit][to];

cout << "-(" << (quit != re\_path[quit][to] ? weight[quit][re\_path[quit][to]] : weight[quit][to])<< ")->";

}

cout << to+1 << " reached.\n";

cout << "Sum path:" << weight[from][to] << endl;

}

///////////////////////

////////////

//////////////////////

/////////////////////

////////////

////////////////////

class TriangleMatrixNeoGraph:public Graph

{

private:

weight\_t \*\*weight;

bool \*\*transfer;

size\_t\*\* re\_path;

public:

TriangleMatrixNeoGraph(size\_t n);

~TriangleMatrixNeoGraph();

void add\_path(size\_t from, weight\_t path, size\_t to);

void add\_path(size\_t from, weight\_t path, size\_t to, int);

void print\_d(void);

/\*floyd warshall algorithm

will find shorted pathes for all nodes in matrix\*/

void floyd\_find();

/\*restore path from A town to B\*/

void restore(size\_t from, size\_t to);

};

TriangleMatrixNeoGraph::~TriangleMatrixNeoGraph()

{

for (int i=0; i<nodes\_s; i++)

{

delete[] weight[i];

delete[] transfer[i];

delete[] re\_path[i];

}

delete[] weight;

delete[] transfer;

delete[] re\_path;

}

void TriangleMatrixNeoGraph::add\_path(size\_t from, weight\_t path, size\_t to)

{

try {

if (from == to) throw Myexception("Attention!Path from one town to them!", 4);

if (from > nodes\_s || to > nodes\_s) throw Myexception("From or To more then size of graph", 4);

}

catch (Myexception & e)

{

throw;

}

if (from < to)

{

weight[to][from] = path;

}

else

{

weight[from][to] = path;

}

}

void TriangleMatrixNeoGraph::add\_path(size\_t from, weight\_t path, size\_t to, int)

{

try {

if (from == to) throw Myexception("Attention!Path from one town to them!", 4);

if (from > nodes\_s || to > nodes\_s) throw Myexception("From or To more then size of graph", 4);

}

catch (Myexception &e)

{

throw;

};

if (from < to) {

if (weight[to][from] > path) {

weight[to][from] = path;

transfer[to][from] = true;

}

}

else {

if (weight[from][to] > path) {

weight[from][to] = path;

transfer[from][to] = true;

}

}

}

void TriangleMatrixNeoGraph::print\_d(void)

{

cout << "\t GRAPH:" << endl;

cout << " ";

cout << static\_cast<char>(-77);

for (int i = 1; i <= nodes\_s; i++)

cout << i << '\t';

cout << endl;

cout << static\_cast<char>(-60);

cout << static\_cast<char>(-59);

int temp = (nodes\_s) \* 7;

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

cout << static\_cast<char>(-60);

cout << endl;

for (int i = 0 ; i< nodes\_s; i++)

{

cout << i + 1 << static\_cast<char>(-77);

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

{

print\_w(weight[i][j]);

cout << '\t';

}

cout << endl;

}

}

void TriangleMatrixNeoGraph::floyd\_find()

{

for (size\_t i = 0; i < nodes\_s; i++)

{

for (size\_t j = 0; j <= i; j++)

{

if (weight[i][j] == WEIGHT\_INF || i == j)

re\_path[i][j] = SIZE\_INF;

else

re\_path[i][j] = 0;

}

}

for (size\_t k = 0; k < nodes\_s; k++)

{

for (size\_t i = 0; i < nodes\_s; i++)

{

for (size\_t j = 0; j <= i ; j++)

{

weight\_t a = k > i ?

weight[k][i] : weight[i][k];

weight\_t b = k > j ?

weight[k][j] : weight[j][k];

if (weight[i][j] > a + b)

{

weight[i][j] = a + b;

re\_path[i][j] = k+1;

}

}

}

}

}

void TriangleMatrixNeoGraph::restore(size\_t from, size\_t to)

{

string s1;

string s2;

bool reached;

if (from < to)

{

size\_t temp = to;

to = from;

from = temp;

s1 = "-(";

s2 = ")->";

reached = true;

}

else

{

s1 = "<-(";

s2 = ")-";

reached = false;

}

stack<size\_t> path\_way;

size\_t quit = from;

size\_t part;

path\_way.push(from + 1);

while (re\_path[quit][to] != 0)

{

path\_way.push(re\_path[quit][to]);

quit = re\_path[quit][to] - 1;

}

part = path\_way.top();

path\_way.push(to + 1);

quit = path\_way.top();

bool flag = quit > part ? transfer[quit - 1][part - 1] : transfer[part - 1][quit - 1];

cout << (reached == true ? "|BEGIN| " : "|reached| ");

cout << (flag == true ? "H: " : "R: ");

while (quit != from + 1)

{

path\_way.pop();

part = path\_way.top();

if (flag != quit > part ? transfer[quit - 1][part - 1] : transfer[part - 1][quit - 1]) {

cout << s1 << "Transfer from " << (flag == true ?

"Highway to Railway" + s2 + "\nR: " : "Railway to Highway" + s2 + "\nH: ");

flag = quit > part ? transfer[quit - 1][part - 1] : transfer[part - 1][quit - 1];

}

cout << quit << s1 << (quit > part ? weight[quit-1][part-1] : weight[part-1][quit-1]) << s2;

quit = part;

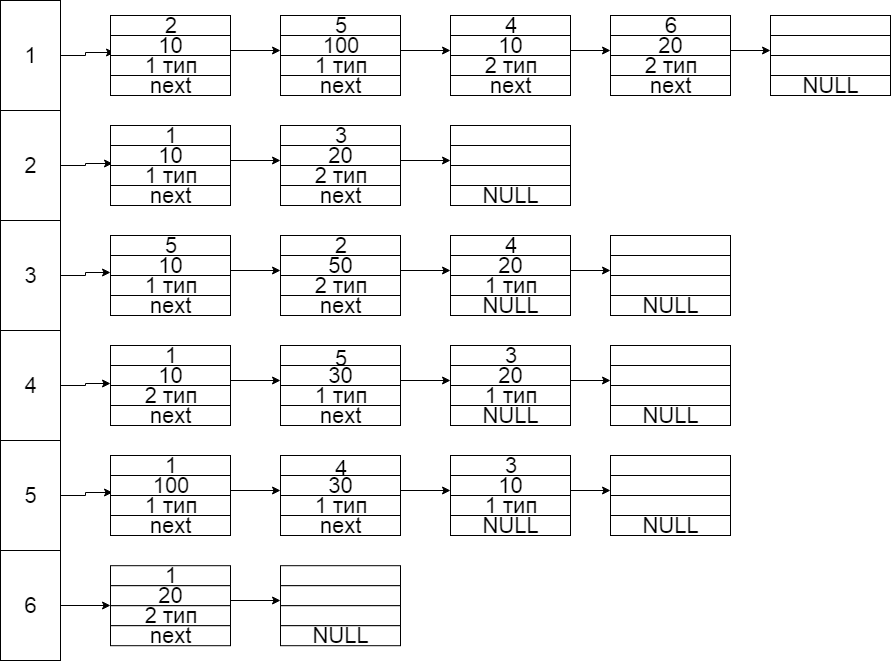
}

cout << quit << (reached == true ? " |reached|" : " |BEGIN|") << endl;

}

*Структура хранения – список смежности*

*Список смежности* формирует массив по количеству вершин, каждый элемент которого указывает на односвязный линейный список смежных вершин. В каждом элементе списка будет хранится индекс и вес ребра.

**

*Текст класса:*

template <class Type>

struct ListData {

Type data;

struct ListData\* next;

ListData() = default;

explicit ListData(const Type value) : data(value) {};

};

template <class Type>

class AdjancecyList

{

private:

ListData<Type>\* front;

ListData<Type>\* back;

public:

explicit AdjancecyList() : front(nullptr), back(nullptr) {};

AdjancecyList(const AdjancecyList<Type> &obj)

{

ListData<Type>\* iter = obj.pbegin();

while (iter != nullptr)

{

this.push\_back(iter->data);

iter = iter->next;

}

};

~AdjancecyList()

{

ListData<Type>\* temp;

while (front != back)

{

temp = front;

front = front->next;

delete temp;

}

delete front;

};

ListData<Type>\* pbegin(void) const { return front; };

ListData<Type>\* pend(void) const { return back; };

const bool empty() const {

return front == nullptr;

};

void push\_front(const Type value);

void push\_back(const Type value);

void insert(ListData<Type>\* pos, const Type value);

void pop\_front(void);

};

template<class Type>

void AdjancecyList<Type>::push\_front(const Type value)

{

ListData<Type>\* temp = new ListData<Type>(value);

if (front == nullptr)

{

temp->next = nullptr;

front = back = temp;

return;

}

temp->next = front;

front = temp;

}

template<class Type>

void AdjancecyList<Type>::push\_back(const Type value)

{

ListData<Type>\* temp = new ListData<Type>(value);

temp->next = nullptr;

if (front == nullptr)

{

front = back = temp;

return;

}

back->next = temp;

back = temp;

}

template<class Type>

void AdjancecyList<Type>::insert(ListData<Type> \* pos, Type value)

{

ListData<Type>\* temp = new ListData<Type>(value);

ListData<Type>\* temp2;

temp2 = pos->next;

pos->next = temp;

temp->next = temp2;

}

template<class Type>

void AdjancecyList<Type>::pop\_front(void)

{

ListData<Type>\* temp;

temp = front;

front = front->next;

delete temp;

}

template<class Type, typename cmp\_T>

class my\_priority\_queue

{

private:

vector<Type> data;

cmp\_T cmp ;

void sift\_up(size\_t pos);

void sift\_down(size\_t pos);

public:

explicit my\_priority\_queue() = default;

explicit my\_priority\_queue(cmp\_T\* c) : cmp(c) {};

const Type top() const {

return data[0];

}

const bool empty() const{

return data.size() == 0;

}

void pop()

{

swap(data[0], data.back());

data.pop\_back();

sift\_down(0);

};

void push(const Type& value)

{

data.push\_back(value);

sift\_up(data.size() - 1);

};

};

template<class Type, class cmp\_T>

void my\_priority\_queue<Type, cmp\_T>::sift\_up(size\_t pos)

{

size\_t parent\_pos = pos / 2;

if (!cmp(data[pos], data[parent\_pos])) {

swap(data[pos], data[parent\_pos]);

if(pos !=parent\_pos)

sift\_up(parent\_pos);

}

}

template<class Type, class cmp\_T>

void my\_priority\_queue<Type, cmp\_T>::sift\_down(size\_t pos)

{

size\_t cmp\_pos = pos;

if (2 \* pos < data.size() && !cmp(data[2 \* pos], data[cmp\_pos])) {

cmp\_pos = 2 \* pos;

}

if (2 \* pos + 1 < data.size() && !cmp(data[2 \* pos + 1], data[cmp\_pos])) {

cmp\_pos = 2 \* pos + 1;

}

if (cmp\_pos != pos) {

swap(data[cmp\_pos], data[pos]);

sift\_down(cmp\_pos);

}

}

struct NodeData {

size\_t to;

weight\_t weight;

bool road\_type;

};

struct cmp\_weight {

bool operator ()(const NodeData& f, const NodeData& s)

{

return f.weight > s.weight;

};

};

class ListNeoGraph : public Graph

{

private:

AdjancecyList<NodeData> \*weight\_list;

pair<size\_t,bool>\* re\_path;

public:

explicit ListNeoGraph(size\_t num) : Graph(num) {

weight\_list = new AdjancecyList<NodeData>[nodes\_s];

re\_path = new pair<size\_t,bool>[nodes\_s];

};

~ListNeoGraph()

{

delete[] weight\_list;

};

void add\_path(size\_t from, weight\_t path, size\_t to);

void add\_path(size\_t from, weight\_t path, size\_t to, int);

void dijkstra\_find(size\_t from);

void restore(size\_t from, size\_t to);

void print\_d(void);

};

template <class Type>

struct ListData {

Type data;

struct ListData\* next;

ListData() = default;

explicit ListData(const Type value) : data(value) {};

};

template <class Type>

class AdjancecyList

{

private:

ListData<Type>\* front;

ListData<Type>\* back;

public:

explicit AdjancecyList() : front(nullptr), back(nullptr) {};

AdjancecyList(const AdjancecyList<Type> &obj)

{

ListData<Type>\* iter = obj.pbegin();

while (iter != nullptr)

{

this.push\_back(iter->data);

iter = iter->next;

}

};

~AdjancecyList()

{

ListData<Type>\* temp;

while (front != back)

{

temp = front;

front = front->next;

delete temp;

}

delete front;

};

ListData<Type>\* pbegin(void) const { return front; };

ListData<Type>\* pend(void) const { return back; };

const bool empty() const {

return front == nullptr;

};

void push\_front(const Type value);

void push\_back(const Type value);

void insert(ListData<Type>\* pos, const Type value);

void pop\_front(void);

};

template<class Type>

void AdjancecyList<Type>::push\_front(const Type value)

{

ListData<Type>\* temp = new ListData<Type>(value);

if (front == nullptr)

{

temp->next = nullptr;

front = back = temp;

return;

}

temp->next = front;

front = temp;

}

template<class Type>

void AdjancecyList<Type>::push\_back(const Type value)

{

ListData<Type>\* temp = new ListData<Type>(value);

temp->next = nullptr;

if (front == nullptr)

{

front = back = temp;

return;

}

back->next = temp;

back = temp;

}

template<class Type>

void AdjancecyList<Type>::insert(ListData<Type> \* pos, Type value)

{

ListData<Type>\* temp = new ListData<Type>(value);

ListData<Type>\* temp2;

temp2 = pos->next;

pos->next = temp;

temp->next = temp2;

}

template<class Type>

void AdjancecyList<Type>::pop\_front(void)

{

ListData<Type>\* temp;

temp = front;

front = front->next;

delete temp;

}

template<class Type, typename cmp\_T>

class my\_priority\_queue

{

private:

vector<Type> data;

cmp\_T cmp ;

void sift\_up(size\_t pos);

void sift\_down(size\_t pos);

public:

explicit my\_priority\_queue() = default;

explicit my\_priority\_queue(cmp\_T\* c) : cmp(c) {};

const Type top() const {

return data[0];

}

const bool empty() const{

return data.size() == 0;

}

void pop()

{

swap(data[0], data.back());

data.pop\_back();

sift\_down(0);

};

void push(const Type& value)

{

data.push\_back(value);

sift\_up(data.size() - 1);

};

};

template<class Type, class cmp\_T>

void my\_priority\_queue<Type, cmp\_T>::sift\_up(size\_t pos)

{

size\_t parent\_pos = pos / 2;

if (!cmp(data[pos], data[parent\_pos])) {

swap(data[pos], data[parent\_pos]);

if(pos !=parent\_pos)

sift\_up(parent\_pos);

}

}

template<class Type, class cmp\_T>

void my\_priority\_queue<Type, cmp\_T>::sift\_down(size\_t pos)

{

size\_t cmp\_pos = pos;

if (2 \* pos < data.size() && !cmp(data[2 \* pos], data[cmp\_pos])) {

cmp\_pos = 2 \* pos;

}

if (2 \* pos + 1 < data.size() && !cmp(data[2 \* pos + 1], data[cmp\_pos])) {

cmp\_pos = 2 \* pos + 1;

}

if (cmp\_pos != pos) {

swap(data[cmp\_pos], data[pos]);

sift\_down(cmp\_pos);

}

}

struct NodeData {

size\_t to;

weight\_t weight;

bool road\_type;

};

struct cmp\_weight {

bool operator ()(const NodeData& f, const NodeData& s)

{

return f.weight > s.weight;

};

};

class ListNeoGraph : public Graph

{

private:

AdjancecyList<NodeData> \*weight\_list;

pair<size\_t,bool>\* re\_path;

public:

explicit ListNeoGraph(size\_t num) : Graph(num) {

weight\_list = new AdjancecyList<NodeData>[nodes\_s];

re\_path = new pair<size\_t,bool>[nodes\_s];

};

~ListNeoGraph()

{

delete[] weight\_list;

};

void add\_path(size\_t from, weight\_t path, size\_t to);

void add\_path(size\_t from, weight\_t path, size\_t to, int);

void dijkstra\_find(size\_t from);

void restore(size\_t from, size\_t to);

void print\_d(void);

};

void ListNeoGraph::add\_path(size\_t from, weight\_t path, size\_t to)

{

try {

if (from == to) throw Myexception("Attention!Path from one town to them!", 4);

if (from > nodes\_s || to > nodes\_s) throw Myexception("From or To more then size of graph", 4);

}

catch (Myexception & e)

{

throw;

};

NodeData value;

value.to = to; value.weight = path; value.road\_type = false;

weight\_list[from].push\_back(value);

value.to = from;

weight\_list[to].push\_back(value);

}

void ListNeoGraph::add\_path(size\_t from, weight\_t path, size\_t to, int)

{

try {

if (from == to) throw Myexception("Attention!Path from one town to them!", 4);

if (from > nodes\_s || to > nodes\_s) throw Myexception("From or To more then size of graph", 4);

}

catch (Myexception & e)

{

throw;

};

NodeData value;

value.weight = path; value.road\_type = true;

ListData<NodeData>\* iter = weight\_list[from].pbegin();

ListData<NodeData>\* iter2 = weight\_list[to].pbegin();

while (iter != nullptr)

{

if (iter->data.to == to && iter->data.weight > path)

{

value.to = to;

iter->data = value;

}

iter = iter->next;

}

while (iter && iter2 != nullptr)

{

if (iter2->data.to == from && iter2->data.weight > path)

{

value.to = from;

iter2->data = value;

}

iter2 = iter2->next;

}

if (!iter && !iter2)

{

value.to = to;

weight\_list[from].push\_back(value);

value.to = from;

weight\_list[to].push\_back(value);

}

}

void ListNeoGraph::dijkstra\_find(size\_t from)

{

weight\_t \*D = new weight\_t[nodes\_s];

my\_priority\_queue<NodeData, cmp\_weight> q;

for (size\_t i = 0; i < nodes\_s; i++) //init

{

D[i] = WEIGHT\_INF;

re\_path[i].first = SIZE\_INF;

re\_path[i].second = false;

}

D[from] = 0;

ListData<NodeData>\* iter = weight\_list[from].pbegin();

while (iter != nullptr) //pushing existing pathes from "FROM var" in queue

{

q.push(iter->data);

D[iter->data.to] = iter->data.weight;

re\_path[iter->data.to].first = from;

re\_path[iter->data.to].second = iter->data.road\_type;

iter = iter->next;

}

while (!q.empty())

{

NodeData t=q.top();

q.pop();

/\*if (t.weight > D[t.to]) //it happens really rarely

{

cout << "TAB " << endl;

continue;

}\*/

iter = weight\_list[t.to].pbegin();

while (iter != nullptr) //looking if path from queue's node is shorter then existing FROM pathes

{

if (D[iter->data.to] > D[t.to] + iter->data.weight)

{

D[iter->data.to] = D[t.to] + iter->data.weight;

q.push(iter->data);

re\_path[iter->data.to].first = t.to;

re\_path[iter->data.to].second = iter->data.road\_type;

}

iter = iter->next;

}

}

cout << "\tAFTER ALGORITHM:" << endl; //print array D with shortest pathes

for (size\_t i = 1; i <= nodes\_s; i++)

{

cout << i << '\t';

}

cout << endl;

size\_t temp = (nodes\_s) \* 7;

for (size\_t i = 0; i <= temp; i++)

cout << static\_cast<char>(-60);

cout << endl;

for (size\_t i = 0; i < nodes\_s; i++)

{

print\_w(D[i]); cout << '\t';

}

cout << endl;

/\*for (size\_t i = 0; i < nodes\_s; i++)

{

cout << re\_path[i].first << '\t';

}

cout << endl;\*/

//cout << "Sum weight from " << from+1 << " town to " << to+1 << " town = " << D[to] << endl;

}

void ListNeoGraph::restore(size\_t from, size\_t to)

{

if (re\_path[to].first == SIZE\_INF)

{

cout << "Path is not exist" << endl;

return;

}

stack<pair<size\_t, bool>> st;

size\_t quit = to;

while (re\_path[quit].first != from)

{

st.push(re\_path[quit]);

quit = re\_path[quit].first;

}

st.push(re\_path[quit]);

bool flag = st.top().second;

cout << (flag == true ? "H: " : "R: ");

while (!st.empty())

{

pair<size\_t, bool> temp = st.top();

cout << temp.first + 1 << "->";

if (flag != temp.second)

{

cout << "Transfer from " << (flag == true ? "Highway to Railway->\nR: " : "Railway to Highway->\nH: ");

flag = temp.second;

}

st.pop();

}

cout << to + 1 << " reached." << endl;

}

void ListNeoGraph::print\_d(void)

{

cout << '\t' << "LIST GRAPH:" << endl;

cout << "F\tW\tT" << endl;

for (size\_t i = 0; i < nodes\_s; i++) {

ListData<NodeData>\* iter = weight\_list[i].pbegin();

while (iter != nullptr)

{

cout << i+1 << '\t';

print\_w(iter->data.weight); cout << '\t' <<

iter->data.to+1 << endl;

iter = iter->next;

}

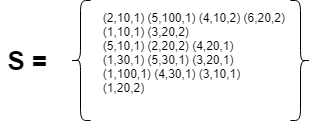
}

}

*Результаты тестирования:*

Пример работы алгоритма Дейкстры.

Начало:

- список смежности

D={(INF), (INF), (INF),(INF), (0) , (INF)} R={(INF,1),(INF,1),(INF,1),(INF,1),(INF,1),(INF,1)} //множество для восстановления пути

V={1,2,3,4,5,6} – множество посещённых вершин графа

1 итерация:

Пока(V не пустое)

Пусть u вершина с минимальным весом из D[u]: u ϵ V;

u = 5

V={1,2,3,4,6};

Для всех i ϵ D:

Если [u][i] ϵ S тогда

Если D[i] > D[u] + S[u][i]

D[i]=D[u]+S[u][i][2]

R[i][1]=u; R[i][2]=S[u][i][3];

===> D={100,INF,10,30, 0 , INF}; R={(5,1),(INF,1),(5,1),(5,1), (INF,1), (INF,1)}

2 итерация:

V не пустое, тогда

u=3

V={1,2,4,6}

D[2] = 10 + 20 = 30;

D={100,30,10,30,0,INF}; R={(5,1),(3,2),(5,1),(5,1),(INF,1),(INF,1)};

3 итерация:

V не пустое, тогда

u=2;

V={1,4,6};

D[1]=30+10=40;

D={40,60,10,30,0,INF}; R={(2,1), (3,2),(5,1),(5,1),(INF,1), (INF,1)};

4 итерация:

u=4;

V={1,6};

D={40,60,10,30,0,INF}; R={(2,1), (3,2),(5,1),(5,1),(INF,1), (INF,1)};

5 итерация:

u=1;

V={6};

D[6]=40+20=60

D={60,60,10,30,0,60}; R={(2,1), (3,2),(5,1),(5,1),(INF,1), (1,2)};

6 итерация:

u=6;

V=0;

D={60,60,10,30,0,80}; R={(2,1), (3,2),(5,1),(5,1),(INF,1), (1,2)};

V = 0 => конец.

Восстановление пути:

Из 5 в 6 => Пока R[i][1] != 5;

1 итер: R[6][1] == 1; R[6][2]==2 ==> из 6 в 1 по второму типу дорог

2 итер: R[1][1] == 2; R[1][2] == 1 ==> из 1 в 2 по первому типу дорог

3 итер:R[2][1] == 3; R[1][2] == 2 ==> из 2 в 3 по второму типу дорог

4 итер:R[3][1] == 5; R[3][2] ==1 ==> из 3 в 5 по первому типу дорог

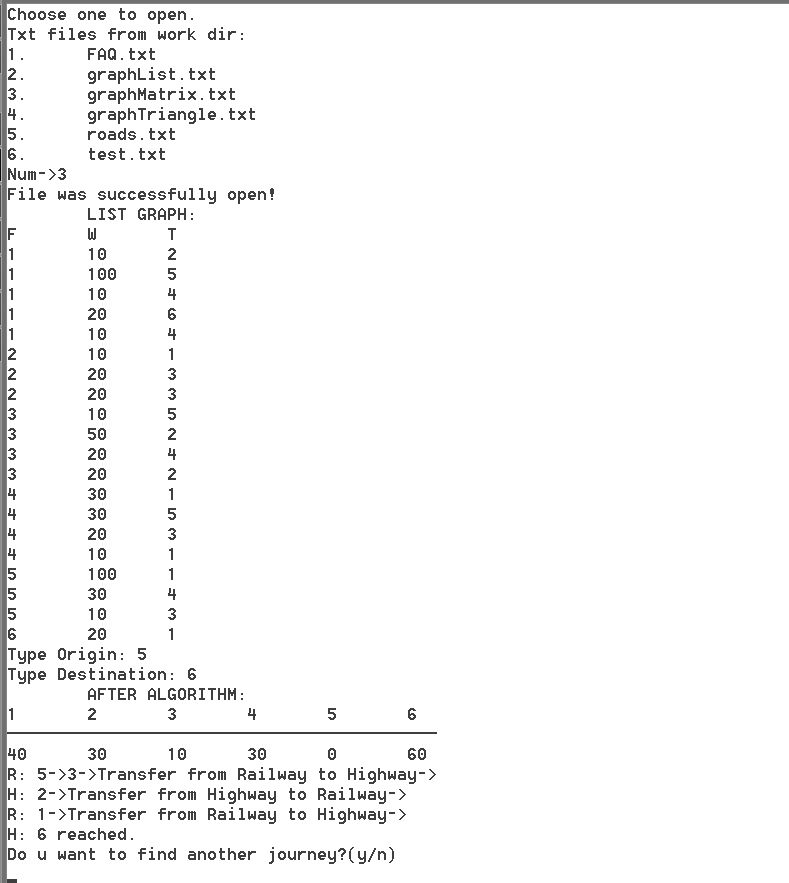
====> (1) 5->3 ->(ПЕРЕСАДКА)->

(2) 2->(ПЕРЕСАДКА)->

(1) 1->(ПЕРЕСАДКА)->

(2) 6 Конец.

*Результаты работы программы:*

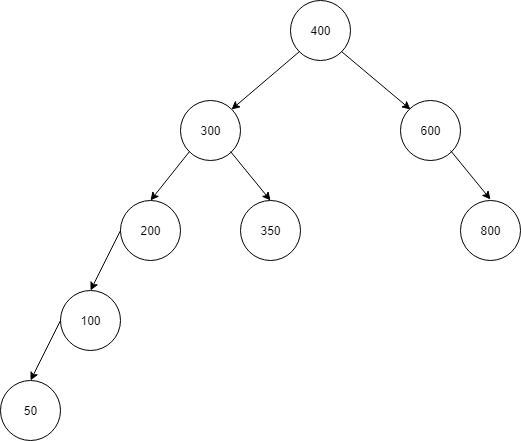
**

2. Реализация дерева – по выбору студента. Все структуры данных оформить в виде классов.

Построить дерево бинарного поиска и реализовать рекурсивную функцию его копирования. Обойти полученное дерево в симметричном порядке.

Входные данные из файла: 400 300 200 350 600 800 100 50

*Структура данных – бинарное дерево поиска:*

**

*Структура хранения – списковая:*

*(по сравнению с векторной предпочтительнее, если используется вставка и удаление элементов, размер структуры не фиксирован)*

*Текст программы:*

#include "files.h"

#include "bst.h"

#include <memory>

using namespace std;

constexpr auto PATH = "C:\\Users\\Владислав\\source\\repos\\LAB2\_sturct\_bst\\LAB2\_sturct\_bst\\treemap";

constexpr auto EXTENSION = ".txt";

int main(const int argc, const char\* argv[])

{

try {

auto files = make\_unique<dirfile>(PATH, EXTENSION);

files->menu();

auto tree = bst<int>();

while (!files->from.eof())

{

int x;

files->from >> x;

tree.add\_node(x);

}

files.reset();

cout << "Tree:" << endl;

tree.print\_visual();

cout << "Preorder:" << endl;

tree.print\_preorder();

cout << "Inorder:" << endl;

tree.print\_inorder();

cout << "Postorder:" << endl;

tree.print\_postorder();

cout << "On levels:" << endl;

tree.print\_onlevel();

cout << "Copy test:" << endl;

{

auto tree\_copy = bst<int>(tree);

tree\_copy.print\_visual();

}

cout << (tree.balanced()==true? "Balanced" : "Not balanced") << endl;

cin.get();

return 0;

}

catch (Myexception &e)

{

cout << e.GetName() << endl;

cout << "Press any key to continue" << endl;

cin.clear(); cin.ignore(numeric\_limits<streamsize>::max(), '\n'); cin.get();

return e.GetVal();

};

}

#pragma once

#include "Myexception.h"

#include <iostream>

#include <queue>

template<class Type>

struct node {

Type data;

struct node\* left;

struct node\* right;

node() = default;

explicit node(const Type x) : data(x) {

left = right = nullptr;

};

};

template<class Type>

class bst

{

private:

node<Type> \*root;

void sub\_add(node<Type> \*&sub, const Type &value);

void distruct\_help(node<Type>\* sub);

void inorder\_help(node<Type>\* sub);

void preorder\_help(node<Type>\* sub);

void postorder\_help(node<Type>\* sub);

void print\_tree(node<Type>\* sub, size\_t level);

void copy\_help(node<Type>\*& sub\_this, node<Type>\* sub\_copy);

void del\_help(node<Type>\*& sub, const Type& value);

void minimum(node<Type>\* min, node<Type>\* cur);

bool check(node<Type>\* sub, int &height);

public:

explicit bst()

{

root = nullptr;

};

~bst()

{

if (root)

{

distruct\_help(root);

}

};

bst(const bst& copy)

{

if (copy.root)

{

copy\_help(root, copy.root);

}

};

const bool empty (void) const

{

return root==nullptr;

};

void add\_node(const Type &value)

{

if (root == nullptr)

root = new node<Type>(value);

else

sub\_add(root, value );

};

void print\_inorder()

{

if (root) {

inorder\_help(root);

std::cout << std::endl;

} else std::cout << "empty" << std::endl;

};

void print\_preorder()

{

if (root) {

preorder\_help(root);

std::cout << std::endl;

} else std::cout << "empty" << std::endl;

};

void print\_postorder()

{

if (root) {

postorder\_help(root);

std::cout << std::endl;

} else std::cout << "empty" << std::endl;

};

void print\_onlevel();

void print\_visual()

{

print\_tree(root, 0);

};

bool balanced()

{

int h;

return check(root, h);

};

void del\_node(const Type& value)

{

del\_help(root, value);

};

};

template<class Type>

void bst<Type>::sub\_add(node<Type> \*&sub, const Type &value)

{

if (sub->data > value)

{

if (sub->left)

sub\_add(sub->left, value);

else

sub->left = new node<Type>(value);

}

else

{

if (sub->right)

sub\_add(sub->right, value);

else

sub->right = new node<Type>(value);

}

}

template<class Type>

void bst<Type>::distruct\_help(node<Type>\* sub)

{

if (sub->left)

distruct\_help(sub->left);

if (sub->right)

distruct\_help(sub->right);

delete sub;

}

template<class Type>

void bst<Type>::inorder\_help(node<Type>\* sub)

{

if(sub->left)

inorder\_help(sub->left);

std::cout << sub->data << ' ';

if(sub->right)

inorder\_help(sub->right);

}

template<class Type>

void bst<Type>::preorder\_help(node<Type>\* sub)

{

std::cout << sub->data << ' ';

if (sub->left)

preorder\_help(sub->left);

if (sub->right)

preorder\_help(sub->right);

}

template<class Type>

void bst<Type>::postorder\_help(node<Type>\* sub)

{

if (sub->left)

postorder\_help(sub->left);

if(sub->right)

postorder\_help(sub->right);

std::cout << sub->data << ' ';

}

template<class Type>

void bst<Type>::print\_tree(node<Type>\* sub, size\_t level)

{

if (sub)

{

print\_tree(sub->right, level + 1);

for (size\_t i = 0; i <= level; i++) std::cout << " ";

std::cout << sub->data << std::endl;

print\_tree(sub->left, level + 1);

}

}

template<class Type>

void bst<Type>::copy\_help(node<Type>\*& sub\_this, node<Type>\* sub\_copy)

{

sub\_this = new node<Type>(sub\_copy->data);

if(sub\_copy->left) copy\_help(sub\_this->left, sub\_copy->left);

if(sub\_copy->right) copy\_help(sub\_this->right, sub\_copy->right);

}

template<class Type>

void bst<Type>::del\_help(node<Type>\*& sub, const Type& value)

{

if (sub)

{

if (value < sub.data)

del\_help(sub->left, value);

else if (value > sub.data)

del\_help(sub->right, value);

else

{

node<Type> temp = sub;

if (sub->left == nullptr)

sub = sub->right;

else if (sub->right == nullptr)

sub = sub->left;

else

delete temp;

}

}

}

template<class Type>

void bst<Type>::minimum(node<Type>\* min, node<Type>\* cur)

{

if (min->left)

minimum(min->left);

else

{

cur->data = min->data;

cur = min;

min = min->right;

}

}

template<class Type>

bool bst<Type>::check(node<Type>\* sub,int &height)

{

if (sub == nullptr)

{

height = 0;

return true;

}

int left=0;

int right=0;

bool flag\_left = check(sub->left, left);

bool flag\_right = check(sub->right, right);

height = (left > right ? left : right) + 1;

if (std::abs(left - right) > 1)

return false;

return flag\_left && flag\_right;

}

template<class Type>

void bst<Type>::print\_onlevel()

{

std::queue<node<Type> \*> q;

q.push(root);

while (!q.empty())

{

node<Type> \*temp = q.front();

q.pop();

std::cout << temp->data << ' ';

if (temp->left)

q.push(temp->left);

if (temp->right)

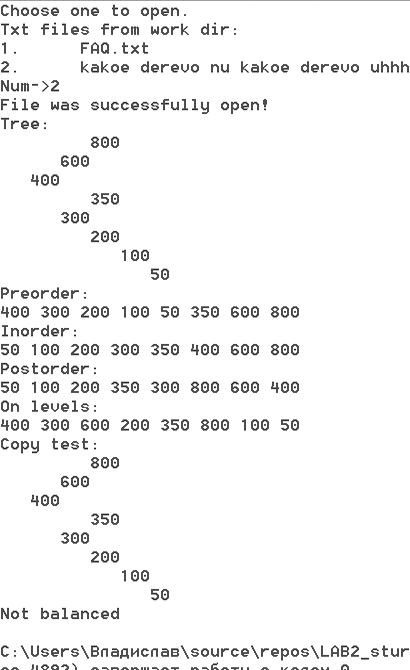
q.push(temp->right);

}

std::cout << std::endl;

}

*Результаты работы программы:*

**