**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ "ЛЬВІВСЬКА ПОЛІТЕХНІКА"**

Кафедра систем та мереж



Звіт

До лабораторної роботи № 1

З дисципліни: «Теорія прийняття рішень»

На тему:   
«Дослідження методів представлення бінарних відношень та операцій на ними»

**Виконав:**

студент

групи КН-318

Карпюк Р.Г.

**Перевірила:**

Асистент кафедри ІСМ Заяць М.М.

Львів – 2019

Мета роботи: Вивчення та порівняння різних способів представлення бінарних відношень. Засвоєння практичних навиків маніпулювання бінарними відношеннями в різних представленнях.

**Варіант 5**

Завдання на роботу

1.Реалізувати методами ООП базовий віртуальний об’єкт “ВІДНОШЕННЯ” і на його основі реалізувати невіртуальні об’єкти - нащадки “ВІДНОШЕННЯ\_МАТР” для представлення бінарних відношень R у вигляді матриці Bі “ВІДНОШЕННЯ\_ЗРІЗ” для представлення у вигляді верхніх перетинів R+. Базовий віртуальний клас повинен містити конструктори та повністю віртуальні функції пп.2-5 а об’єкти нащадки реалізовувати ці віртуальні функції відповідно до особливостей представлення.

2.Для обох методів представлення реалізувати конструктори по замовчуванню, конструктори копіювання та визначити операції присвоєння.

3.Реалізувати конструктори створення елементарних бінарних відношень (пустого, повного, діагонального, антидіагонального).

4.Реалізувати функції-члени для виконання наступних дій над відношеннями: перетину, об’єднання, різниці, симетричної різниці, доповнення, знаходження оберненого відношення, композиції, звуження.

5.Реалізувати функцію-член перевірки умови включення одного відношення в інше.

6.За допомогою отриманої програми здійснити обчислення виразу: K = (PQ)\Rd при використанні обох методів представлення. Оцінити складність реалізації, об’єм використаної пам’яті та час обрахунку. Значення відношень P,Q,R згідно варіанту.

7.Оформити звіт про виконану роботу.

**Код програми**

#include "stdafx.h"

#define n 5

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

class Relation {

public:

int s;

virtual void print() {};

virtual void operator& (Relation const &obj) {}; // Intersection

virtual void operator| (Relation const &obj) {}; // Union

virtual void operator> (Relation const &obj) {}; // Difference

virtual void operator- (Relation const &obj) {}; // Symmetric Difference

virtual void operator\* (Relation const &obj) {}; // Composition

virtual void operator^ (Relation const &obj) {}; // Include

virtual void operator= (Relation const &obj) {}; // Include

virtual void operator++ () {}; // Inversion

virtual void operator! () {}; // P\_

virtual void decrease() {}; // P\_

void Pd() {}; // P\_

Relation() {}

~Relation() {}

};

class RelationMatrix : public Relation

{

public:

bool \*\*arr;

const static int s = n;

void print() {

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

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

cout << arr[i][j] << " ";

}

cout << endl;

}

cout << endl;

};

void operator& (RelationMatrix const &obj) {

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

{

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

{

arr[i][j] = obj.arr[i][j] & arr[i][j];

}

}

}; // Intersection

void operator| (RelationMatrix const &obj)

{

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

{

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

{

arr[i][j] = obj.arr[i][j] | arr[i][j];

}

}

}; // Union

void operator> (RelationMatrix const &obj)

{

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

{

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

{

arr[i][j] = arr[i][j] > obj.arr[i][j];

}

}

}; // Difference

void operator- (RelationMatrix const &obj)

{

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

{

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

{

arr[i][j] = (arr[i][j] > obj.arr[i][j]) | (obj.arr[i][j] > arr[i][j]);

}

}

}; // Symmetric Difference

void operator\* (RelationMatrix const &obj)

{

bool \*\*tmp;

tmp = new bool\*[s];

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

{

tmp[i] = new bool[s];

}

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

{

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

{

tmp[i][j] = 0;

for (int k = 0; k < s; k++)

{

tmp[i][j] = tmp[i][j] || (arr[i][k] && obj.arr[k][j]);

}

}

}

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

{

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

{

arr[i][j] = tmp[i][j];

}

}

}; // Composition

bool operator^ (RelationMatrix const &obj) // Include

{

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

{

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

{

if ((arr[i][j] != obj.arr[i][j]) && arr[i][j] == 0) {

return 0;

}

}

}

return 1;

};

void operator++ () // Inverse

{

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

{

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

{

swap(arr[i][j], arr[j][i]);

}

}

};

void operator! () // P\_

{

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

{

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

{

arr[i][j] = !arr[i][j];

}

}

};

void decrease(int const obj[], bool \*\*res, int size) // Decrease (zv)

{

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

{

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

{

res[i][j] = arr[obj[i]][obj[j]];

}

}

};

void Pd() // PD

{

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

{

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

{

arr[i][j] = !arr[i][j];

}

}

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

{

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

{

swap(arr[i][j], arr[j][i]);

}

}

};

void operator= (const RelationMatrix& M)

{

arr = new bool\*[s];

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

arr[i] = new bool[s];

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

{

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

{

arr[i][j] = M.arr[i][j];

}

}

}

RelationMatrix(char rel\_type = 'e') {

/\*

e - empty

1 - full

d - diagonal

a - anti\_diagonal

\*/

arr = new bool\*[s];

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

{

arr[i] = new bool[s];

memset(arr[i], 0, s \* sizeof(bool));

}

switch (rel\_type)

{

case '1': {

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

{

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

{

arr[i][j] = 1;

}

}

break;

}

case 'd': {

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

{

arr[i][i] = 1;

}

break;

}

case 'a': {

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

{

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

{

arr[i][j] = (i != j);

}

}

break;

}

default:

break;

}

}

RelationMatrix(int \*\*matr) {

arr = new bool\*[s];

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

arr[i] = new bool[s];

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

{

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

{

arr[i][j] = matr[i][j];

}

}

}

RelationMatrix(int matr[s][s]) {

arr = new bool\*[s];

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

arr[i] = new bool[s];

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

{

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

{

arr[i][j] = matr[i][j];

}

}

}

~RelationMatrix() {}

};

class RelationVector : public Relation

{

public:

vector<vector<int>> vec;

const static int s = n;

void print() {

for (int i = 0; i < vec.size(); i++) {

cout << "R(" << i << ") = ";

for (int j = 0; j < vec[i].size(); j++) {

cout << vec[i][j] << " ";

}

cout << endl;

}

cout << endl;

};

void operator& (RelationVector const &obj) {

vector<vector<int>> tmp\_vec = RelationVector().vec;

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

{

for (int j = 0; j < obj.vec[i].size(); j++)

{

if (binary\_search(vec[i].begin(), vec[i].end(), obj.vec[i][j])

&& binary\_search(obj.vec[i].begin(), obj.vec[i].end(), obj.vec[i][j])) {

tmp\_vec[i].push\_back(obj.vec[i][j]);

}

}

}

vec = tmp\_vec;

}; // Intersection

void operator| (RelationVector const &obj)

{

vector<vector<int>> tmp\_vec = vec;

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

{

for (int j = 0; j < obj.vec[i].size(); j++)

{

if (!binary\_search(vec[i].begin(), vec[i].end(), obj.vec[i][j])

&& binary\_search(obj.vec[i].begin(), obj.vec[i].end(), obj.vec[i][j])) {

tmp\_vec[i].push\_back(obj.vec[i][j]);

sort(tmp\_vec[i].begin(), tmp\_vec[i].end());

}

}

}

vec = tmp\_vec;

}; // Union

void operator> (RelationVector const &obj)

{

vector<vector<int>> tmp\_vec = RelationVector().vec;

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

{

for (int j = 0; j < vec[i].size(); j++)

{

if (binary\_search(vec[i].begin(), vec[i].end(), vec[i][j])

&& !binary\_search(obj.vec[i].begin(), obj.vec[i].end(), vec[i][j])) {

tmp\_vec[i].push\_back(vec[i][j]);

}

}

}

vec = tmp\_vec;

}; // Difference

void operator- (RelationVector const &obj)

{

vector<vector<int>> tmp\_vec = RelationVector().vec;

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

{

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

{

if (binary\_search(vec[i].begin(), vec[i].end(), j)

!= binary\_search(obj.vec[i].begin(), obj.vec[i].end(), j)) {

tmp\_vec[i].push\_back(j);

}

}

}

vec = tmp\_vec;

};

void operator\* (RelationVector const &obj)

{

vector<vector<int>> tmp\_vec = RelationVector().vec;

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

{

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

{

bool tmp = false;

for (int k = 0; k < s; k++)

{

tmp = tmp || (binary\_search(obj.vec[i].begin(), obj.vec[i].end(), k)

&& binary\_search(vec[k].begin(), vec[k].end(), j));

}

if (tmp)

tmp\_vec[i].push\_back(j);

}

}

vec = tmp\_vec;

};

bool include(RelationVector const &obj) // Include

{

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

{

for (int j = 0; j < obj.vec[i].size(); j++)

{

if (!binary\_search(vec[i].begin(), vec[i].end(), obj.vec[i][j]))

return 0;

}

}

return 1;

};

void operator++ () // Inverse

{

vector<vector<int>> tmp\_vec = RelationVector().vec;

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

{

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

{

if (binary\_search(vec[i].begin(), vec[i].end(), j)) {

tmp\_vec[j].push\_back(i);

}

}

}

vec = tmp\_vec;

};

void operator! () // P\_

{

vector<vector<int>> tmp\_vec = RelationVector().vec;

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

{

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

{

if (!binary\_search(vec[i].begin(), vec[i].end(), j)) {

tmp\_vec[i].push\_back(j);

}

}

}

vec = tmp\_vec;

};

void decrease(int const obj[], int const &size) // Decrease (zv)

{

vector<vector<int>> tmp\_vec = RelationVector().vec;

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

{

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

{

if (binary\_search(vec[obj[i]].begin(), vec[obj[i]].end(), obj[j])) {

tmp\_vec[i].push\_back(j);

}

}

}

vec = tmp\_vec;

};

void operator= (const RelationVector &obj)

{

vec.clear();

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

vector<int> row;

for (int j = 0; j < obj.vec[i].size(); j++) {

row.push\_back(obj.vec[i][j]);

}

vec.push\_back(row);

}

}

void Pd() // PD

{

vector<vector<int>> tmp\_vec = RelationVector().vec;

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

{

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

{

if (!binary\_search(vec[i].begin(), vec[i].end(), j)) {

tmp\_vec[i].push\_back(j);

}

}

}

vec = tmp\_vec;

tmp\_vec = RelationVector().vec;

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

{

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

{

if (binary\_search(vec[i].begin(), vec[i].end(), j)) {

tmp\_vec[j].push\_back(i);

}

}

}

vec = tmp\_vec;

};

RelationVector(char rel\_type = 'e') {

/\*

e - empty

1 - full

d - diagonal

a - anti\_diagonal

\*/

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

vector<int> row;

vec.push\_back(row);

}

switch (rel\_type)

{

case '1': {

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

{

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

{

vec[i].push\_back(j);

}

}

break;

}

case 'd': {

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

{

vec[i].push\_back(i);

}

break;

}

case 'a': {

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

{

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

{

if (i != j) {

vec[i].push\_back(j);

};

}

}

break;

}

default:

break;

}

}

RelationVector(int \*\*matr) {

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

vector<int> row;

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

if (matr[j][i]) {

row.push\_back(j);

}

}

vec.push\_back(row);

}

}

RelationVector(int matr[s][s]) {

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

vector<int> row;

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

if (matr[j][i]) {

row.push\_back(j);

}

}

vec.push\_back(row);

}

}

~RelationVector() {}

};

int main()

{

int P[RelationMatrix::s][RelationMatrix::s] =

{ { 0,0,1,0,0 },

{ 0,0,1,0,0 },

{ 1,0,0,1,0 },

{ 0,0,1,1,0 },

{ 1,0,0,0,0 } };

int Q[RelationMatrix::s][RelationMatrix::s] =

{ { 0,1,0,0,0 },

{ 0,1,0,0,0 },

{ 0,0,1,0,0 },

{ 0,1,0,0,1 },

{ 1,1,0,0,0 } };

int R[RelationMatrix::s][RelationMatrix::s] =

{ { 0,0,0,1,0 },

{ 0,1,1,0,0 },

{ 0,1,0,0,1 },

{ 0,0,0,1,0 },

{ 1,0,1,0,0 } };

RelationVector p(P);

RelationVector q(Q);

RelationVector r(R);

RelationMatrix pm(P);

RelationMatrix qm(Q);

RelationMatrix rm(R);

p\*q;

r.Pd();

p - r;

p.print();

cout << endl;

pm\*qm;

rm.Pd();

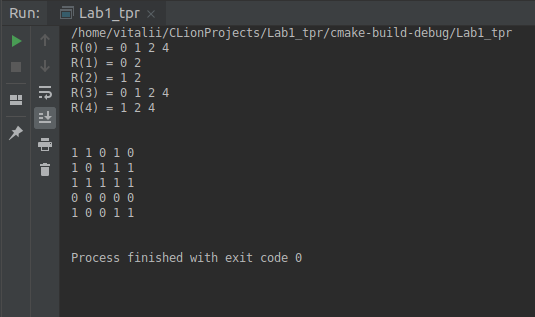
pm - rm;

pm.print();

system("pause");

return 0;

}

****

**Висновок:** в ході роботи було виконано елементарні операції над бінарними відношення. Серед них: сукупність, перетин, різниця тощо. Реалізовано два методи представлення відношень: матричний, верхній зріз, конструктори створення елементарних відношень: діагонального, порожнього, повного, антидіагонального.