Лабораторная работа №2

Артамоновой Анастасии

**Конспект**

Под *рекурсией* понимают способ задания функции через саму себя, например, способ задания факториала в виде N! = (n – 1)!\*n.

Под итерацией понимают результат многократно повторяемой какой-либо операции, например, представление факториала в виде n! = 1\*2\*3... \*n.

Рекурсивные структуры данных

Список - набор элементов, расположенных в определенном порядке.

Список очередности - список, в котором последний поступающий элемент добавляется к нижней части списка.

Список с использованием указателей - список, в котором каждый элемент содержит указатель на следующий элемент списка.

Очередь - тип данных, при котором новые данные располагаются следом за существующими в порядке поступления; поступившие первыми данные при этом обрабатываются первыми.

Стек - линейный список, в котором все включения и исключения осуществляются в одном конце списка.

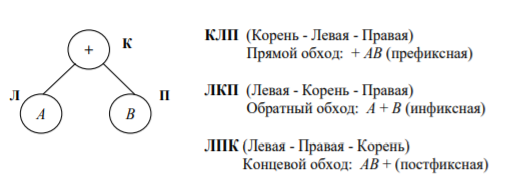
Дек (стек с двумя концами) - линейный список, в котором все включения и исключения делаются на обоих концах списка.

Виды обхода бинарных деревьев

1) обход в направлении слева направо (обратный порядок, инфиксная запись);

2) обход сверху вниз (прямой порядок, префиксная запись);

3) обход снизу вверх (концевой порядок, постфиксная запись).



Параллельные методы умножения блочно-представленных матриц

Алгоритм Фокса

Для организации параллельных вычислений при блочном представлении матриц предположим, что процессоры образуют логическую прямоугольную решетку размером k×k (обозначим через pij процессор, располагаемый на пересечении i строки и j столбца решетки).

Основные положения параллельного метода, известного как алгоритм Фокса, состоят в следующем:

• каждый из процессоров решетки отвечает за вычисление одного блока матрицы C;

• в ходе вычислений на каждом из процессоров pij располагаются четыре матричных блока: блок Сij матрицы C, вычисляемый процессором; блок Aij матрицы A, размещенный в процессоре перед началом вычислений; блоки Aij ′ и Bij ′ - матриц A и B, получаемые процессором в ходе выполнения вычислений.

Алгоритм Кэннона

Отличие алгоритма Кэннона от алгоритма Фокса состоит в изменении схемы начального распределения блоков перемножаемых матриц между процессорами вычислительной системы. Начальное расположение блоков в алгоритме Кэннона подбирается таким образом, чтобы располагаемые блоки на процессорах могли быть перемножены без каких-либо дополнительных передач данных между процессорами. При этом подобное распределение блоков может быть организовано таким образом, что перемещение блоков между процессорами в ходе вычислений может осуществляться с использованием более простых коммуникационных операций.

С учетом высказанных замечаний этап инициализации алгоритма Кэннона включает выполнение следующих операций передач данных:

• на каждый процессор pij передаются блоки Aij , Bij;

• для каждой строки i процессорной «решетки» блоки матрицы A сдвигаются на (i − 1) позиций влево;

• для каждого столбца j процессорной «решетки» блоки матрицы В сдвигаются на (j − 1) позиций вверх.

В ходе вычислений на каждой итерации алгоритма Кэннона каждый блок матрицы A сдвигается на один процессор влево по решетке, а каждый блок матрицы В - на один процессор вверх.

Ленточный алгоритм

При ленточной схеме разделения данных исходные матрицы разбиваются на горизонтальные (для матрицы A) и вертикальные (для матрицы B) полосы.



Получаемые полосы распределяются по процессорам, при этом на каждом из имеющегося набора процессоров располагается только по одной полосе матриц A и B. Перемножение полос (эту операцию процессоры могут выполнить параллельно) приводит к получению части блоков результирующей матрицы C. Для вычисления оставшихся блоков матрицы C сочетания полос матриц A и B на процессорах должны быть изменены. В наиболее простом виде это может быть обеспечено, например, при кольцевой топологии вычислительной сети (при числе процессоров, равном количеству полос), т.е. необходимое для матричного умножения изменение положения данных может быть обеспечено циклическим сдвигом полос матрицы B по кольцу. После многократного выполнения описанных действий (количество необходимых повторений является равным числу процессоров) на каждом процессоре получается набор блоков, образующий горизонтальную полосу матрицы C.

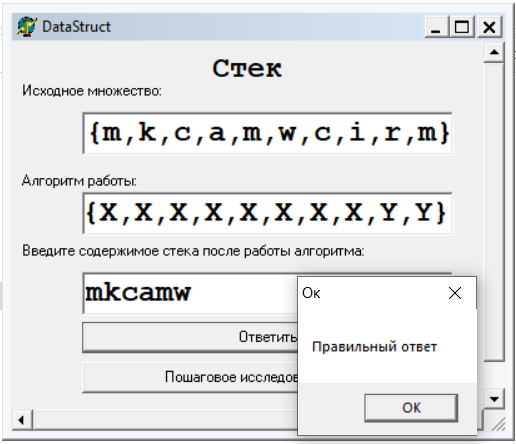
Метод Крускала

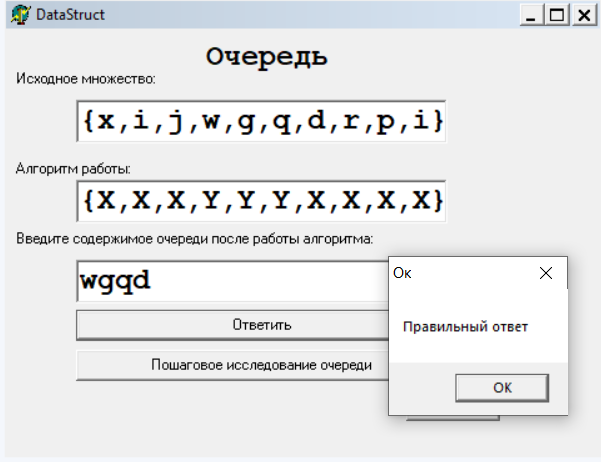
Вначале осуществляется предварительная сортировка весов ребер в порядке их возрастания. В начале работы алгоритма принимается, что в искомом остове не проведено ни одно ребро (т.е. остов состоит из изолированного множества вершин v1, v2,…, vm, где m - количество вершин графа). Считается, что множество Ws имеет вид: Ws={{v1},{v2},…, {vn}}, где {vi} обозначает множество, состоящее из единственной изолированной вершины i v . Проверка k l Ws v ,v ∈ предполагает установление факта: входят ли вершины k l v ,v во множество Ws как изолированные, или они сами входят в подмножества постепенно увеличивающихся связных вершин Wk , Wl , каждое из которых имеет вид: Wk = {…,vk ,…} и Wl = {…,vl ,…}. Если обе вершины k l v ,v содержатся в одном из подмножеств Wk , Wl , то ребро (k, l) в остов не включается, в противном случае данное ребро включается в остов, а множества Wk Wl , объединяются. Работа метода Крускала заканчивается, когда множество Ws совпадет по мощности с множеством всех вершин графа V.

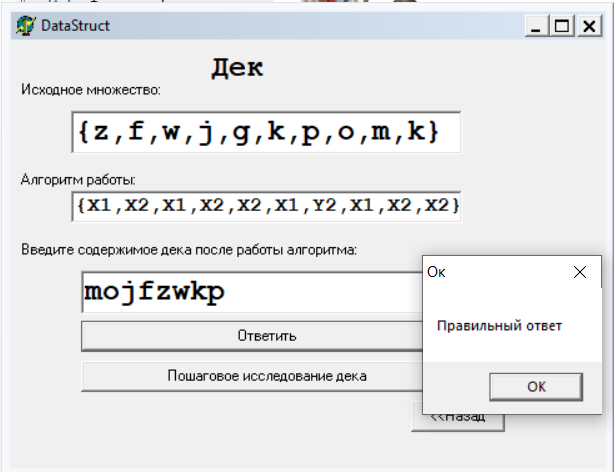
Метод Прима

При использовании метода Прима от исходного графа переходим к его представлению в виде матрицы смежности. На графе выбирается ребро минимального веса. Выбранное ребро вместе с вершинами образует первоначальный фрагмент остовного дерева. Затем анализируются веса ребер от каждой вершины фрагмента до оставшихся невыбранных вершин. Выбирается минимальное ребро, которое присоединяется к первоначальному фрагменту и т.д. Процесс продолжается до тех пор, пока в остовное дерево не будут включены все вершины исходного графа.

**Задание 3.1**



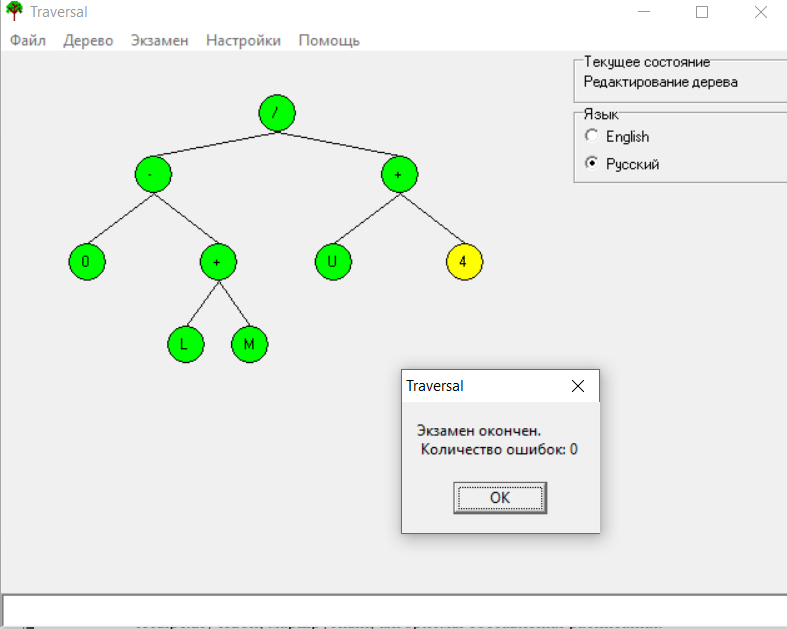




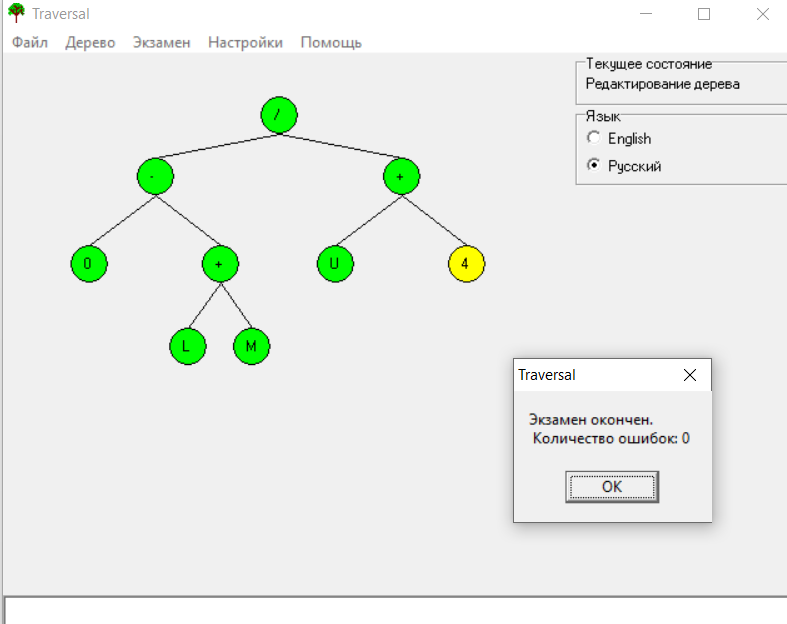
**Задание 3.2**

8 элементов

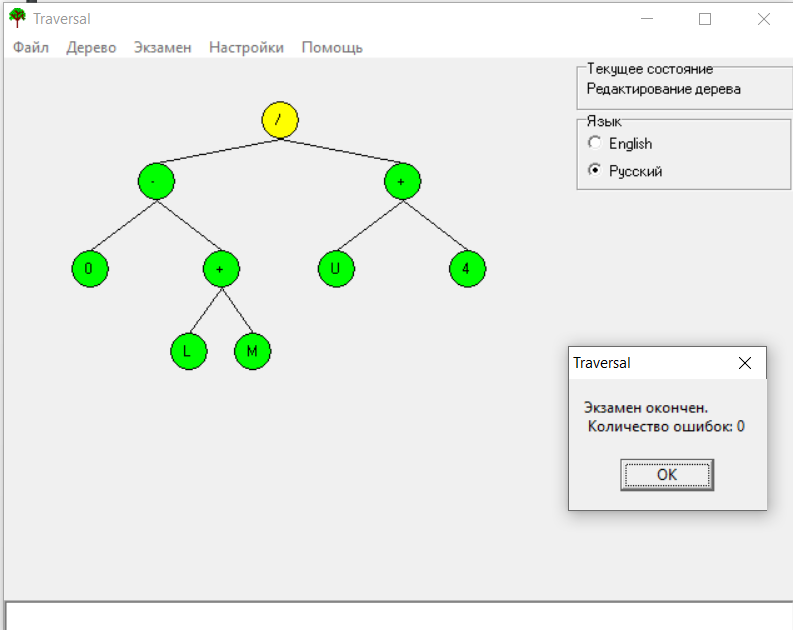
Префиксный



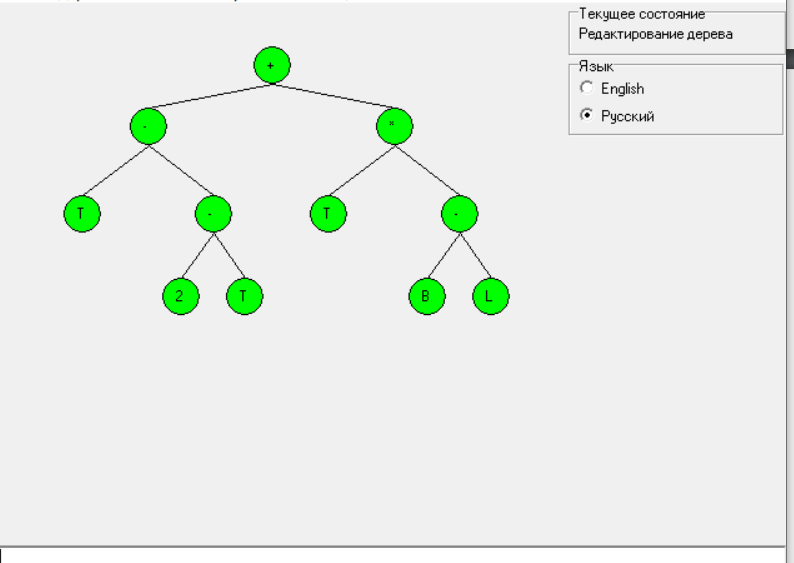
Инфиксный



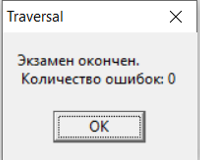
Постфиксный



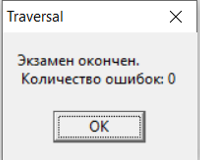
10 элементов



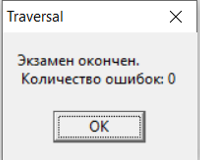
Префиксный



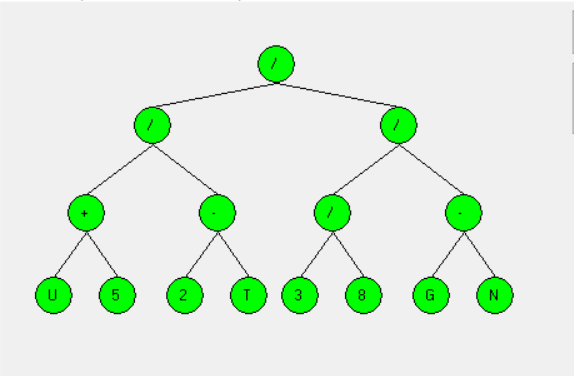
Инфиксный



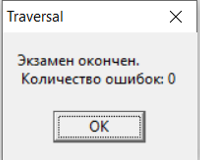
Постфиксный



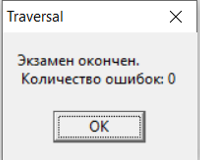
12 элементов



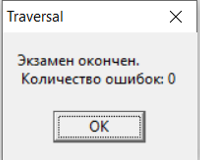
Префиксный



Инфиксный



Постфиксный



**Задание 3.3**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <iostream>

using namespace std;

#define NMAX 100

typedef struct stack

{

char element[NMAX]; //хранилище

size\_t top; // индекс элемента в вершине стека.

} stack;

void init(stack\* st)

{

st->top = 0;

}

void push(stack\* st, char e)

{

if (st->top < NMAX)

{

st->element[st->top] = e;

st->top++;

}

else

cout << "Error: stack overflow " << st->top << endl;

}

char pop(stack\* st)

{

char element;

if ((st->top) > 0)

{

st->top--;

element = st->element[st->top];

return element;

}

else

{

cout<<"Stack is empty!"<<endl;

return 0;

}

}

int stack\_top(stack\* st)

{

if ((st->top) > 0)

return st->element[st->top - 1];

else

return 0;

}

int isempty(stack\* st)

{

if ((st->top) == 0)

return 1;

else

return 0;

}

int main()

{

stack\* st;

char el[NMAX];

st = new stack[NMAX];

init(st);

cin>> el;

int n = strlen(el) + 1;

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

{

if (el[i] == '(' || el[i] == '[' || el[i] == '{')

{

push(st, el[i]);

}

else if

(

(el[i] == ')' && stack\_top(st) != '(') ||

(el[i] == ']' && stack\_top(st) != '[') ||

(el[i] == '}' && stack\_top(st) != '{')

)

{

push(st, el[i]);

}

else if

(

(el[i] == ')' && stack\_top(st) == '(') ||

(el[i] == ']' && stack\_top(st) == '[') ||

(el[i] == '}' && stack\_top(st) == '{'))

{

pop(st);

}

}

if (isempty(st) == 1)

cout << "Success" << endl;

else

cout << "No success" << endl;

delete[] st;

}

**Задание 3.4**

#include <iostream>

using namespace std;

double function(int n)

{

double ans;

if (n == 1.0)

return(2.0 / 3.0);

ans = ((2\* (double)function(double(n) - 1.0)) / 3.0);

return(ans);

}

int main()

{

setlocale(LC\_ALL, "ru");

int n;

cout << "Введите n: ";

cin >> n;

cout << function(n) << endl;

return 0;

}

**Задание 3.5**

#include <iostream>

#include <vector>

#include <time.h>

using namespace std;

double function(int n)

{

double ans;

if (n == 1.0)

return(2.0 / 3.0);

ans = ((2\* (double)function(double(n) - 1.0)) / 3.0);

return(ans);

}

int main()

{

vector<double> v;

clock\_t time;

setlocale(LC\_ALL, "ru");

int n;

cout << "Введите количество элементов массива: ";

cin >> n;

time = clock();

for (int i = 1; i < n; ++i)

v.push\_back(function(i));

time = clock() - time;

for (auto i : v)

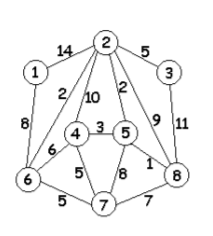
cout << i << "\n";

cout << endl;

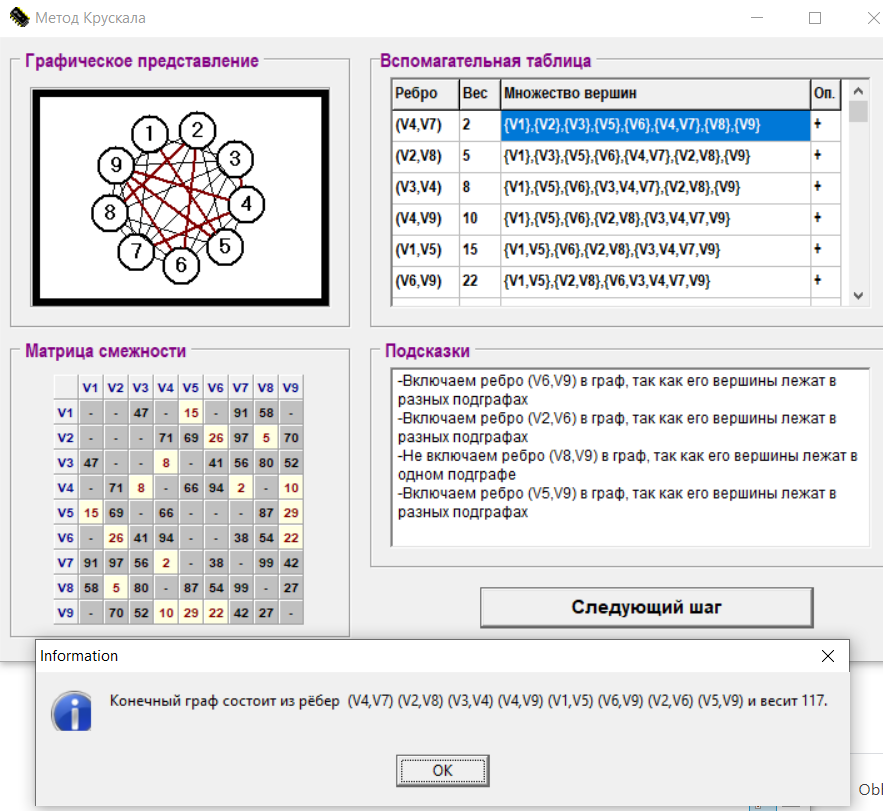
printf("Время: %f секунд.\n", ((float)time) / CLOCKS\_PER\_SEC);

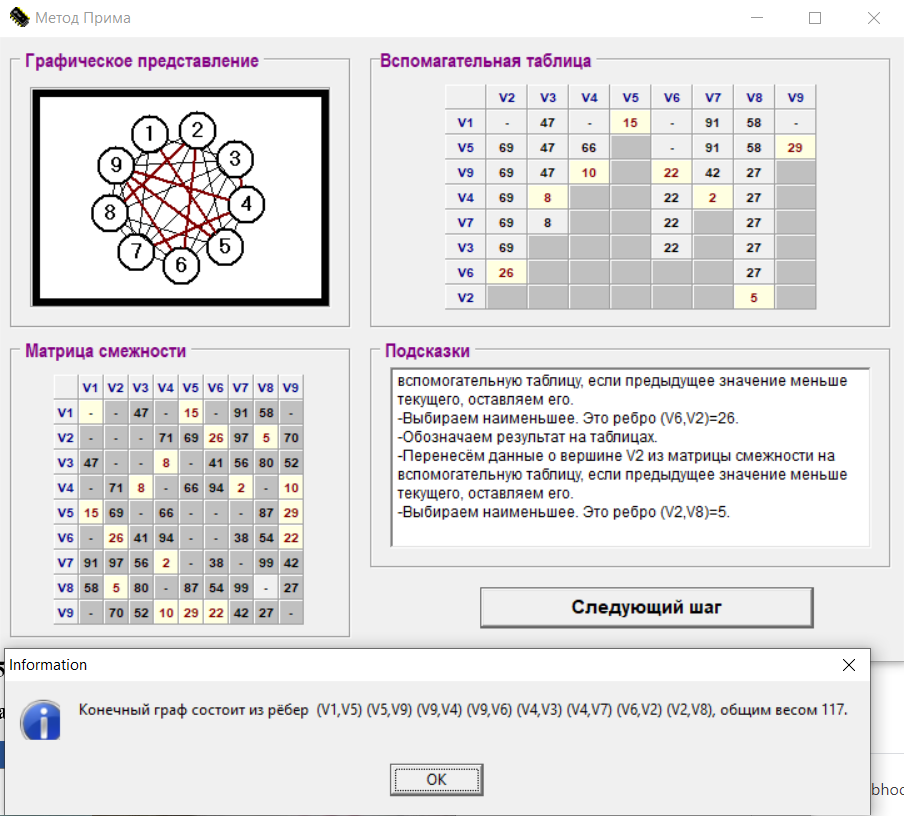
return 0;

}



**Задание 4.3**





**Задание 4.4**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

#define edge pair<int,int>

class Graph {

private:

vector<pair<int, edge>> G;

vector<pair<int, edge>> T;

int\* variety;

int V;

public:

Graph(int V);

void AddWeightedEdge(int u, int v, int w);

int find\_set(int i);

void union\_set(int u, int v);

void kruskal();

void print();

};

Graph::Graph(int V) {

variety = new int[V];

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

variety[i] = i;

G.clear();

T.clear();

}

void Graph::AddWeightedEdge(int u, int v, int w) {

G.push\_back(make\_pair(w, edge(u, v)));

G.push\_back(make\_pair(w, edge(v, u)));

}

int Graph::find\_set(int i) {

if (i == variety[i])

return i;

else

return find\_set(variety[i]);

}

void Graph::union\_set(int u, int v) {

variety[u] = variety[v];

}

void Graph::kruskal() {

size\_t i, uRep, vRep;

sort(G.begin(), G.end());

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

uRep = find\_set(G[i].second.first);

vRep = find\_set(G[i].second.second);

if (uRep != vRep) {

T.push\_back(G[i]);

union\_set(uRep, vRep);

}

}

}

void Graph::print() {

cout << "Ребро :" << " Вес" << endl;

int minw = 0;

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

cout << T[i].second.first +1 << "-" << T[i].second.second +1 << " : "

<< T[i].first;

cout << endl;

minw += T[i].first;

}

cout << "Минимальный вес: " << minw << endl;

}

int main() {

setlocale(LC\_ALL, "ru");

Graph g(8);

g.AddWeightedEdge(0, 1, 14);

g.AddWeightedEdge(0, 5, 8);

g.AddWeightedEdge(1, 5, 2);

g.AddWeightedEdge(1, 3, 10);

g.AddWeightedEdge(1, 4, 2);

g.AddWeightedEdge(1, 7, 9);

g.AddWeightedEdge(1, 2, 5);

g.AddWeightedEdge(2, 7, 11);

g.AddWeightedEdge(3, 5, 6);

g.AddWeightedEdge(3, 4, 3);

g.AddWeightedEdge(3, 6, 5);

g.AddWeightedEdge(4, 6, 8);

g.AddWeightedEdge(4, 7, 1);

g.AddWeightedEdge(5, 6, 5);

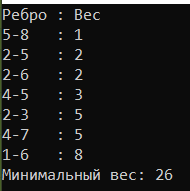
g.AddWeightedEdge(6, 7, 7);

g.kruskal();

g.print();

return 0;

}



**Задание 4.5**

**Метод Крускала**

Цикломатическое число: γ = n – m + 1 = 15 – 8 +1 = 8 ребер необходимо удалить.

|  |  |  |  |
| --- | --- | --- | --- |
| Ребро | Вес | Множества вершин | Операция |
| {1}{2}{3}{4}{5}{6}{7}{8} | | | |
| 58 | 1 | {1}{2}{3}{4}{58}{6}{7} | Включение |
| 62 | 2 | {1}{26}{3}{4}{58}{7} | Включение |
| 25 | 2 | {1}{2658}{3}{4}{7} | Включение |
| 45 | 3 | {1}{26584}{3}{7} | Включение |
| 47 | 5 | {1}{265847}{3} | Включение |
| 67 | 5 | --------------------------------------- | Исключение |
| 23 | 5 | {1}{2658473} | Включение |
| 64 | 6 | -------------------------------------- | Исключение |
| 78 | 7 | -------------------------------------- | Исключение |
| 16 | 8 | {26584731} | Включение |
| 57 | 8 | -------------------------------------- | Исключение |
| 28 | 9 | -------------------------------------- | Исключение |
| 24 | 10 | -------------------------------------- | Исключение |
| 38 | 11 | -------------------------------------- | Исключение |
| 12 | 14 | -------------------------------------- | Исключение |

**Метод Прима**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | V1 | V2 | V3 | V4 | V5 | V6 | V7 |
| V8 | \* | 9 | 11 | \* | 1 | \* | 7 |
| V5 | \* | 2 | 11 | 3 | \* | \* | 7 |
| V2 | 14 | \* | 5 | 3 | \* | 2 | 7 |
| V6 | 8 | \* | 5 | 3 | \* | \* | 5 |
| V7 | 8 | \* | 5 | 3 | \* | \* | \* |
| V4 | 8 | \* | 5 | \* | \* | \* | \* |
| V3 | 8 | \* | \* | \* | \* | \* | \* |

**Задание 4.6**

