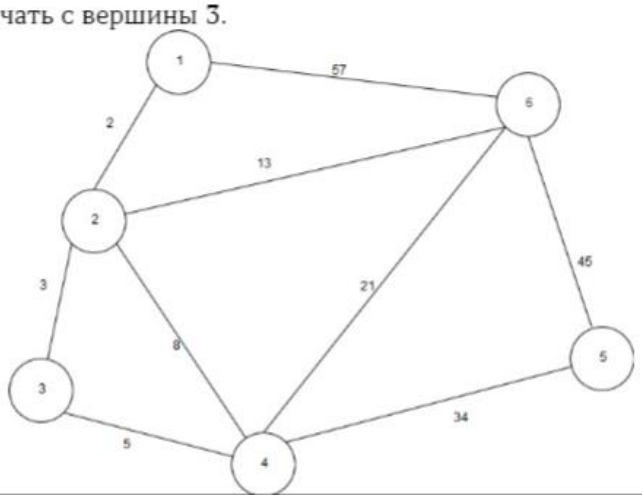
Романов Артём Алексеевич (ИВТ-23-1Б). Графы, вариант 14.

Постановка задачи:

14	Реализовать граф, а также алгоритм Дейкстры, выполнив все необходимые действия. Выполнение начать с вершины 3. 

Анализ задачи:

Граф - упорядоченная структура, следовательно для её реализации нужно создать класс графа и его узлов (class Graph и class Node).

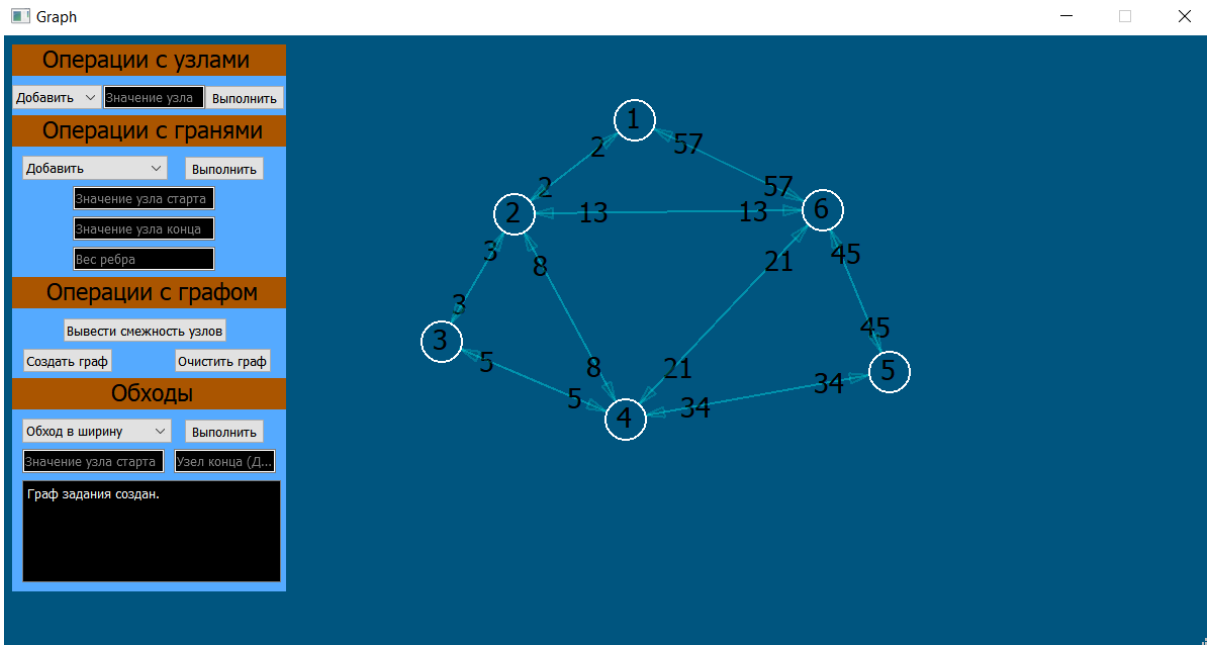
Граф создаётся из вершин (узлов), следовательно для построения и изменения графа нужно использовать функцию создания и удаления вершин(узлов), а также создания/удаления/редактирования рёбер [AddNode()/removeNode, addEdge/removeEdge/updateEdgeWeight]

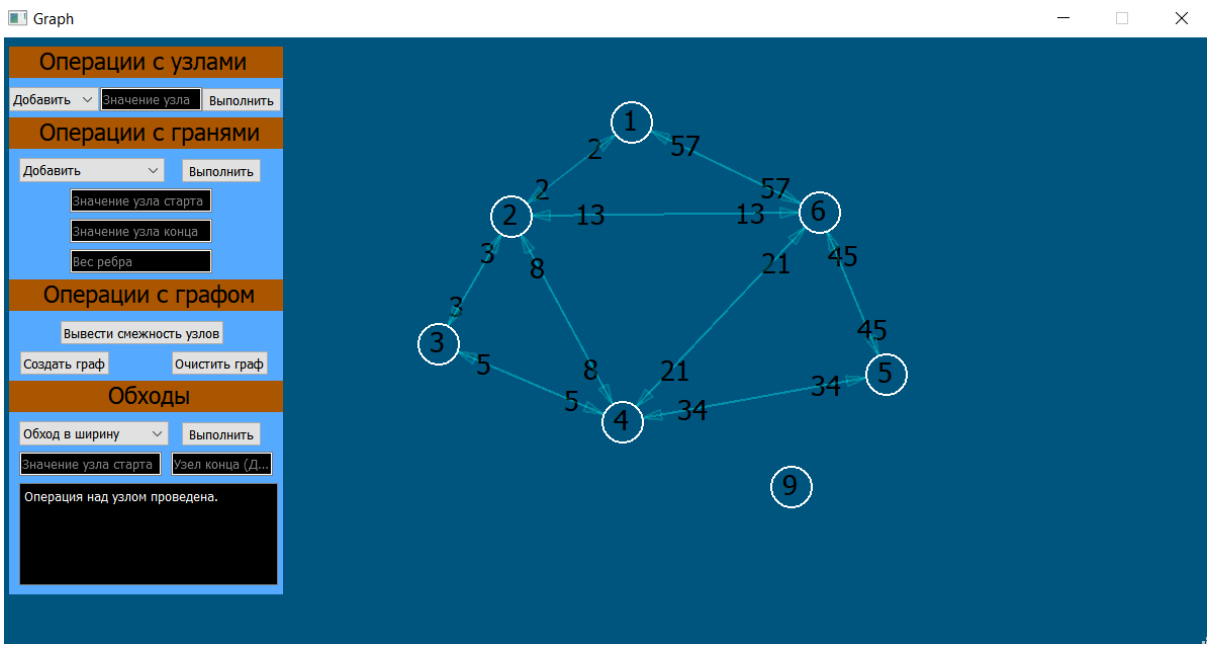
Вывод графа реализован с помощью Qt.

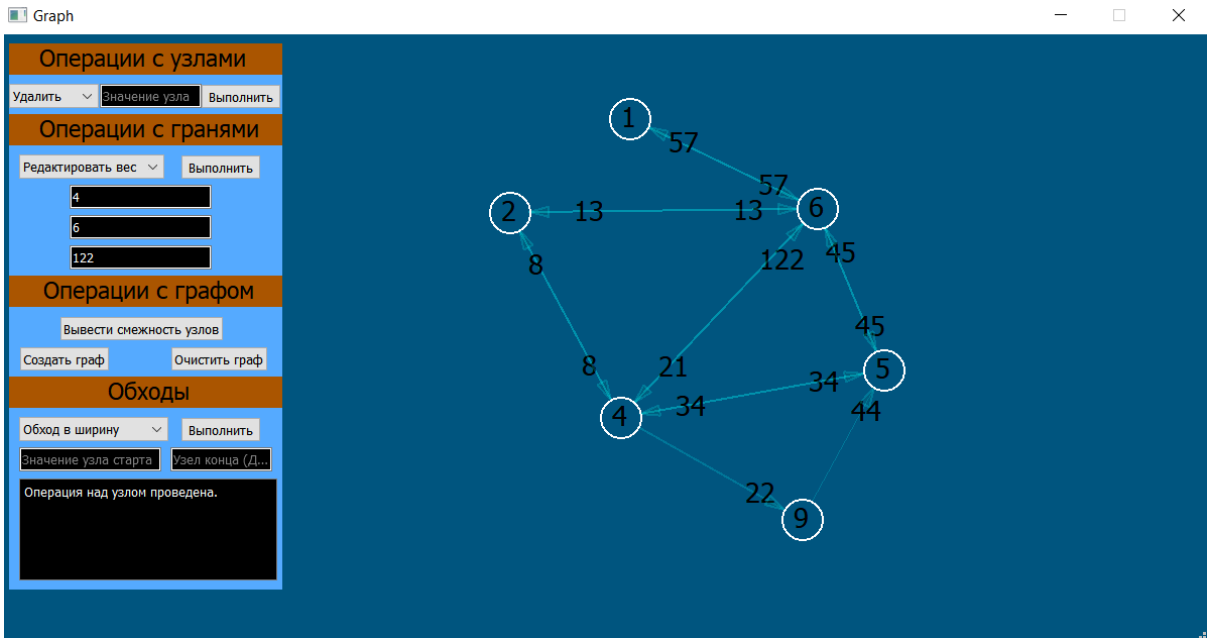
Вывод смежности узлов происходит через проход всех узлов и вывода связанных с ними рёбер (printAdjTable())

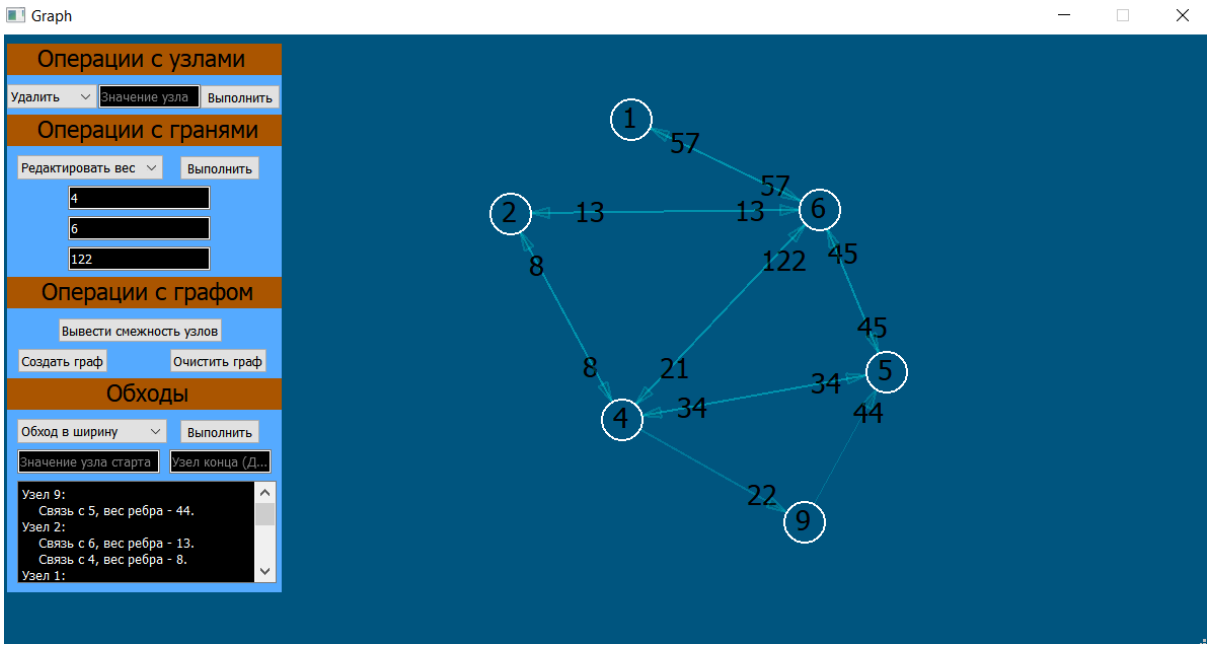
Код программы находится в файлах проекта Qt.

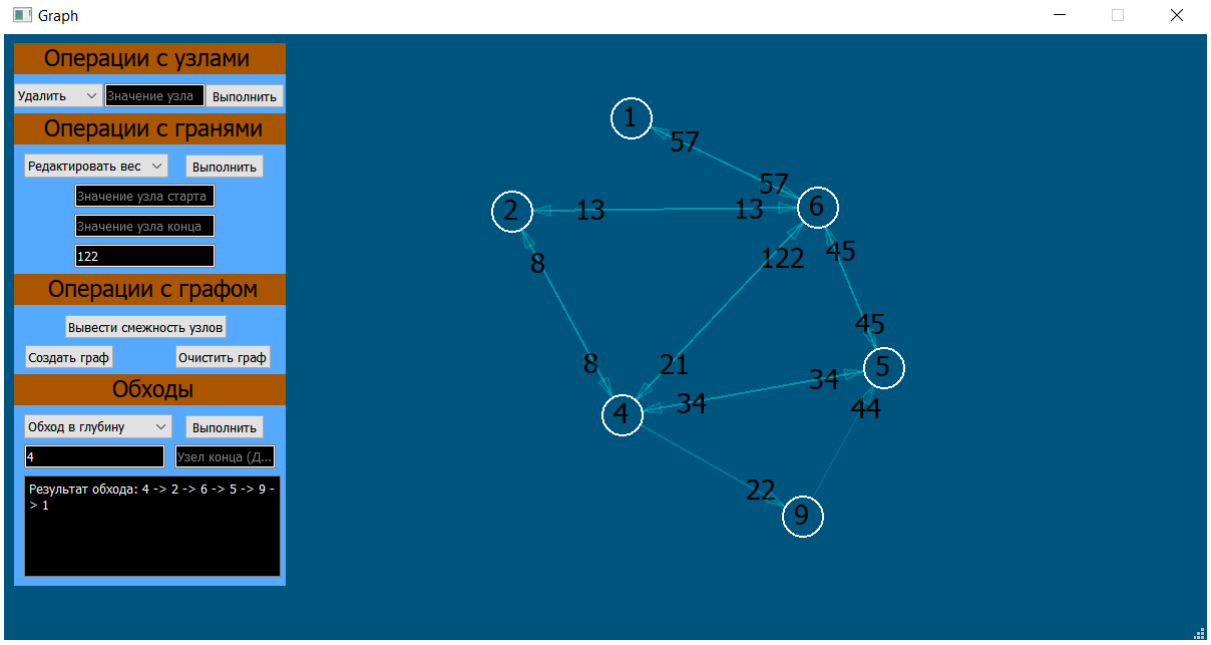
Скриншоты работы программы:


Graph
Операции с узлами
Добавить
Значение узла
Выполнить
Операции с гранями
Добавить
Выполнить
Значение узла старта
Значение узла конца
Вес ребра
Операции с графом
Вывести смежность узлов
Создать граф
Очистить граф
Обходы
Обход в ширину
Выполнить
Значение узла старта
Узел конца (Д...
Граф задания создан.
1
2
3
4
5
6
57
2
57
2
13
13
3
8
21
45
3
45
5
8
21
34
5
34


Graph
Операции с узлами
Добавить
Значение узла
Выполнить
Операции с гранями
Добавить
Выполнить
Значение узла старта
Значение узла конца
Вес ребра
Операции с графом
Вывести смежность узлов
Создать граф
Очистить граф
Обходы
Обход в ширину
Выполнить
Значение узла старта
Узел конца (Д...
Операция над узлом проведена.
1
2
3
4
5
6
9
57
2
57
2
13
13
3
8
21
45
3
45
5
8
21
34
5
34


Graph
Операции с узлами
Удалить
Значение узла
Выполнить
Операции с гранями
Редактировать вес
Выполнить
4
6
122
Операции с графом
Вывести смежность узлов
Создать граф
Очистить граф
Обходы
Обход в ширину
Выполнить
Значение узла старта
Узел конца (Д...
Операция над узлом проведена.
1
57
57
2
13
13
6
8
122
45
45
8
21
34
5
4
34
44
22
9


Graph
Операции с узлами
Удалить
Значение узла
Выполнить
Операции с гранями
Редактировать вес
Выполнить
4
6
122
Операции с графом
Вывести смежность узлов
Создать граф
Очистить граф
Обходы
Обход в ширину
Выполнить
Значение узла старта
Узел конца (Д...
Узел 9:
Связь с 5, вес ребра - 44.
Узел 2:
Связь с 6, вес ребра - 13.
Связь с 4, вес ребра - 8.
Узел 1:
1
57
57
2
13
13
6
8
122
45
45
8
21
34
5
4
34
44
22
9



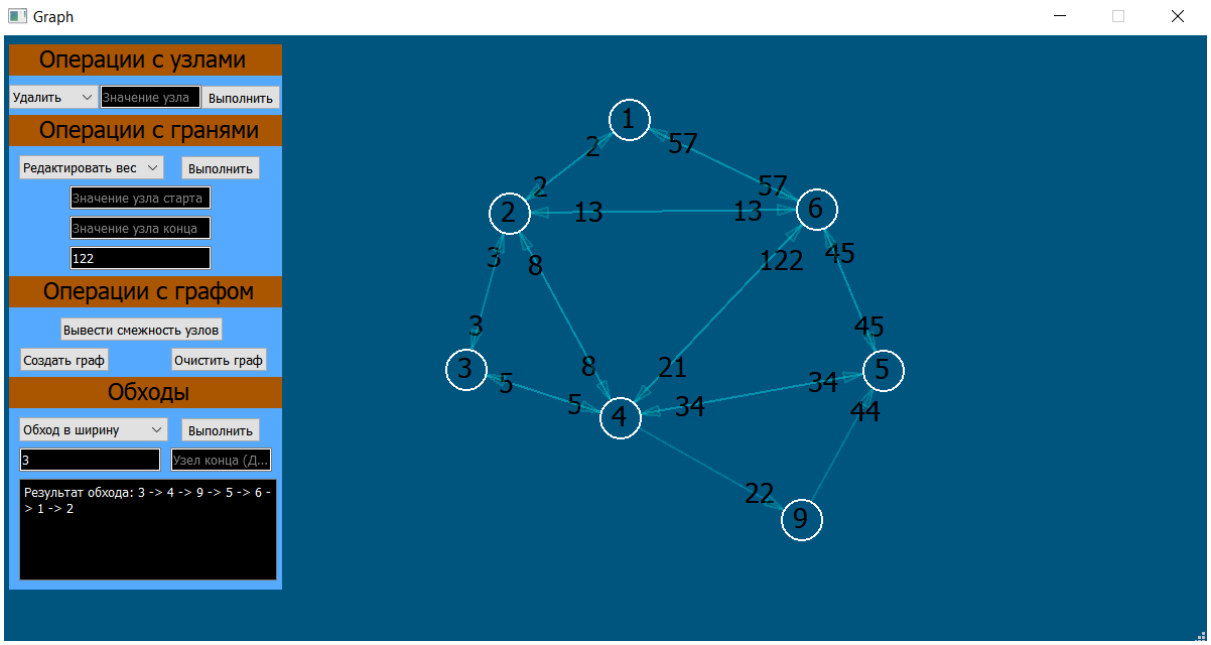
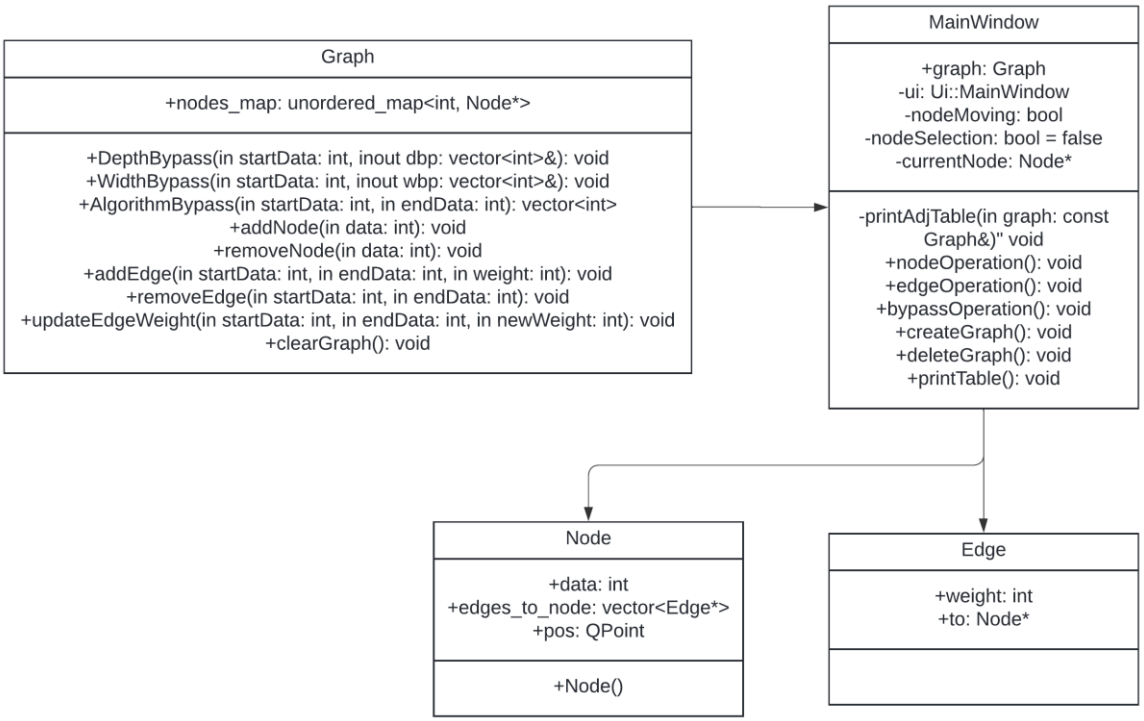


Диаграмма классов:



Код программы:

Mainwindow.h:

```cpp
#include <QWidget>
#include <QMouseEvent>
#include <ui_mainwindow.h>
#include <unordered_map>
#include <unordered_set>

using namespace std;

// Предопределение классов
class Edge;
class Node;
class Graph;

class Node{
public:
    int data;
    vector<Edge*> edges_to_node;
    QPoint pos;
    Node(){
        pos = QPoint(800 + (rand()%400 - 400), 300 + (rand()%300 - 300));
    }
};

class Edge{
public:
    int weight;
    Node* to;
};

class Graph{
public:
    unordered_map<int, Node*> nodes_map;

    void DepthBypass(int startData, vector<int>& dbp);
    void WidthBypass(int startData, vector<int>& wbp);
    vector<int> AlgorithmBypass(int startData, int endData);

    void addNode(int data);
    void removeNode(int data);

    void addEdge(int startData, int endData, int weight);
    void removeEdge(int startData, int endData);
    void updateEdgeWeight(int startData, int endData, int newWeight);

    void clearGraph();
};

class MainWindow : public QMainWindow{
    Q_OBJECT
public:
    MainWindow(QWidget* parent = nullptr);
    ~MainWindow();
    Graph graph;

protected:
    void paintEvent(QPaintEvent* event) override;
    void mousePressEvent(QMouseEvent* event) override;
    void mouseMoveEvent(QMouseEvent* event) override;
    void mouseReleaseEvent(QMouseEvent* event) override;
```

```cpp
private:
    Ui::MainWindow ui;
    Node* selectedNode;
    bool nodeMoving;
    bool nodeSelection = false;
    Node* currentNode;

    void printAdjTable(const Graph& graph);

public slots:
    void nodeOperation();
    void edgeOperation();
    void bypassOperation();
    void createGraph();
    void deleteGraph();
    void printTable();
};
```

Main.cpp:

```cpp
#include "mainwindow.h"

#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

Mainwindow.cpp:

```cpp
#include "mainwindow.h"

#include <QPainter>
#include <cmath>
#include <QTimer>
#include <queue>
#include <stack>

MainWindow::MainWindow(QWidget* parent): QMainWindow(parent){
    ui.setupUi( this );

    connect(ui.nodeOperationButton, &QPushButton::pressed, this, &MainWindow::nodeOperation);
    connect(ui.edgeOperationButton, &QPushButton::pressed, this, &MainWindow::edgeOperation);
    connect(ui.bypassOperationButton, &QPushButton::pressed, this, &MainWindow::bypassOperation);
    connect(ui.createGraphButton, &QPushButton::clicked, this, &MainWindow::createGraph);
    connect(ui.deleteGraphButton, &QPushButton::clicked, this, &MainWindow::deleteGraph);
    connect(ui.printTableButton, &QPushButton::clicked, this, &MainWindow::printTable);
}

// Операции над узлами
void Graph::addNode(int data){
    if (nodes_map.find(data) == nodes_map.end()){
        Node* newNode = new Node;
        newNode->data = data;
        nodes_map[data] = newNode;
    }
}
void Graph::removeNode(int data){
    for (auto& pair : nodes_map){
        Node* node = pair.second;
        vector<Edge*> edges_to_remove;
        for (Edge* edge : node->edges_to_node){
            if (edge->to->data == data){
                edges_to_remove.push_back(edge);
            }
        }
        for (Edge* edge : edges_to_remove){
            auto it = find(node->edges_to_node.begin(), node->edges_to_node.end(), edge);
            if (it != node->edges_to_node.end()){
                node->edges_to_node.erase(it);
                delete edge;
            }
        }
    }
    auto it = nodes_map.find(data);
    if (it != nodes_map.end()){
        delete it->second;
        nodes_map.erase(it);
    }
}
void MainWindow::nodeOperation(){
```

```cpp
void MainWindow::nodeOperation(){
    if (ui.nodeValue->text().isEmpty()){ return; }

    int operation = ui.nodeOperations->currentIndex();
    int nodeValue = ui.nodeValue->text().toInt();

    // 0 - Добавить, 1 - Удалить
    switch(operation){
    case 0: graph.addNode(nodeValue); break;
    case 1: graph.removeNode(nodeValue); break;
    }
    ui.nodeValue->clear(); update();
    ui.statusText->setText("Операция над узлом проведена.");
}

// Операции над рёбрами
void Graph::addEdge(int startData, int endData, int weight){
    for (Edge* edge : nodes_map[startData]->edges_to_node){
        if (edge->to == nodes_map[endData]){
            return;
        }
    }
    Edge* newEdge = new Edge();
    newEdge->to = nodes_map[endData];
    newEdge->weight = weight;
    nodes_map[startData]->edges_to_node.push_back(newEdge);
}
void Graph::removeEdge(int startData, int endData){
    auto startNodeIt = nodes_map.find(startData);
    auto endNodeIt = nodes_map.find(endData);
    if (startNodeIt == nodes_map.end() || endNodeIt == nodes_map.end())
    {
        return;
    }
    Node* startNode = startNodeIt->second;
    Edge* edgeToRemove = nullptr;

    for (Edge* edge : startNode->edges_to_node)
    {
        if (edge->to->data == endData)
        {
            edgeToRemove = edge;
            break;
        }
    }
    if (edgeToRemove)
    {
        auto it = find(startNode->edges_to_node.begin(), startNode->edges_to_node.end(), edgeToRemove);
        if (it != startNode->edges_to_node.end())
        {
            startNode->edges_to_node.erase(it);
            delete edgeToRemove;
        }
    }
}
```

```
void Graph::updateEdgeWeight(int startData, int endData, int newWeight){
    if (nodes_map.find(startData) == nodes_map.end() || nodes_map.find(endData) == nodes_map.end()){
        return;
    }
    Node* startNode = nodes_map[startData];
    Node* endNode = nodes_map[endData];
    for (Edge* edge : startNode->edges_to_node){
        if (edge->to == endNode){
            edge->weight = newWeight;
            return;
        }
    }
}
void MainWindow::edgeOperation(){
    if (ui.nodeStart->text().isEmpty() || ui.nodeEnd->text().isEmpty()) { return; }

    int nodeStart = ui.nodeStart->text().toInt();
    int nodeEnd = ui.nodeEnd->text().toInt();
    int edgeWeight = ui.edgeWeight->text().toInt();

    int operation = ui.edgeOperations->currentIndex();

    // 0 - Добавить, 1 - Удалить, 2 - Редактировать вес
    switch(operation){
    case 0: if(ui.edgeWeight->text().isEmpty()){ return; } graph.addEdge(nodeStart, nodeEnd, edgeWeight);  break;
    case 1: graph.removeEdge(nodeStart, nodeEnd); break;
    case 2: if(ui.edgeWeight->text().isEmpty()){ return; } graph.updateEdgeWeight(nodeStart, nodeEnd, edgeWeight);  break;

    ui.nodeStart->clear(); ui.nodeEnd->clear(); ui.edgeWeight->clear(); update();
    ui.statusText->setText("Операция над гранью проведена.");
    }
}

// Обходы графа
void Graph::DepthBypass(int startData, vector<int> &dbp){
    stack<Node*> nodeStack;
    nodeStack.push(nodes_map[startData]);
    unordered_set<int> visited;
    visited.insert(startData);
    while (!nodeStack.empty()){
        Node* currentNode = nodeStack.top();
        nodeStack.pop();
        dbp.push_back(currentNode->data);
        for (Edge* edge : currentNode->edges_to_node){
            if (visited.find(edge->to->data) == visited.end()){
                nodeStack.push(edge->to);
                visited.insert(edge->to->data);
            }
        }
    }
    for (auto const& pair : nodes_map){
        if (visited.find(pair.first) == visited.end()){
            dbp.push_back(pair.first);
            visited.insert(pair.first);
        }
    }
}
```

```cpp
void Graph::WidthBypass(int startData, vector<int> &wbp){
    queue<Node*> q;
    unordered_map<int, bool> visited;
    Node* startNode = nodes_map[startData];
    q.push(startNode);
    visited[startData] = true;
    while (!q.empty()){
        Node* currentNode = q.front();
        q.pop();
        wbp.push_back(currentNode->data);
        for (Edge* edge : currentNode->edges_to_node){
            Node* neighborNode = edge->to;
            if (!visited[neighborNode->data]){
                visited[neighborNode->data] = true;
                q.push(neighborNode);
            }
        }
    }
    for (const auto& pair : nodes_map){
        Node* node = pair.second;
        if (!visited[node->data]){
            q.push(node);
            visited[node->data] = true;
            while (!q.empty()){
                Node* currentNode = q.front();
                q.pop();
                wbp.push_back(currentNode->data);
                for (Edge* edge : currentNode->edges_to_node){
                    Node* neighborNode = edge->to;
                    if (!visited[neighborNode->data]){
                        visited[neighborNode->data] = true;
                        q.push(neighborNode);
                    }
                }
            }
        }
    }
}
```

```cpp
vector<int> Graph::AlgorithmBypass(int startData, int endData){
    unordered_map<int, int> dist;
    unordered_map<int, int> prev;
    vector<int> result;
    for (auto& pair : nodes_map){
        dist[pair.first] = INT_MAX;
        prev[pair.first] = -1;
    }
    dist[startData] = 0;
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pq.push({ 0, startData });
    while (!pq.empty()){
        int u = pq.top().second;
        pq.pop();
        if (u == endData) break;
        for (Edge* edge : nodes_map[u]->edges_to_node){
            int v = edge->to->data;
            int alt = dist[u] + edge->weight;

            if (alt < dist[v]){
                dist[v] = alt;
                prev[v] = u;
                pq.push({ alt, v });
            }
        }
    }
    for (int at = endData; at != -1; at = prev[at]){
        result.push_back(at);
    }
    reverse(result.begin(), result.end());
    if (result[0] == endData) { result.pop_back(); }
    return result;
}
```

```cpp
void MainWindow::bypassOperation(){
    if( ui.bypassStart->text().isEmpty() ){ return; }

    int nodeStart = ui.bypassStart->text().toInt();
    int nodeEnd = ui.bypassEnd->text().toInt();
    vector<int> passed;
    QString bypassResult;

    int operation = ui.bypassOperations->currentIndex();

    // 0 - Обход в ширину, 1 - Обход в глубину, 2 - Обход алгоритмом Дейкстры
    switch(operation){
    case 0: graph.DepthBypass(nodeStart, passed); break;
    case 1: graph.WidthBypass(nodeStart, passed); break;
    case 2: if( ui.bypassEnd->text().isEmpty() ){ return; } passed = graph.AlgorithmBypass(nodeStart, nodeEnd); break;
    }

    // Добавление в строку вывода результата обхода
    for (unsigned int i = 0; i < passed.size(); i++){
        bypassResult.append(QString::number(passed[i]));
        if (i < passed.size() - 1){
            bypassResult.append(" -> ");
        }
    }

    // Таймер и отображение движения по узлам
    static unsigned int idx = 0;
    QTimer* nodeMoveTimer = new QTimer(this);
    connect(nodeMoveTimer, &QTimer::timeout, [=](){
        if (passed.size() != 0 and idx < passed.size()){
            Node* nod = graph.nodes_map[passed[idx]];
            currentNode = nod;
            nodeSelection = true;
            update();
            idx++;
        } else {
            ui.statusText->setText("Результат обхода: " + bypassResult);
            nodeMoveTimer->stop();
            nodeMoveTimer->deleteLater();
            nodeSelection = false;
            update();
            idx = 0;
        }
    });

    ui.nodeStart->clear();
    ui.nodeEnd->clear();
    nodeMoveTimer->start(1000);
}


// Удаление и создание графа
void Graph::clearGraph(){
    for (auto& pair : nodes_map){
        Node* node = pair.second;
        delete node;
    }

    nodes_map.clear();
}
```

```
void MainWindow::createGraph(){
    graph.addNode(1);
    graph.addNode(2);
    graph.addNode(3);
    graph.addNode(4);
    graph.addNode(5);
    graph.addNode(6);

    graph.addEdge(1, 2, 2);
    graph.addEdge(1, 6, 57);

    graph.addEdge(2, 1, 2);
    graph.addEdge(2, 6, 13);
    graph.addEdge(2, 4, 8);
    graph.addEdge(2, 3, 3);

    graph.addEdge(3, 2, 3);
    graph.addEdge(3, 4, 5);

    graph.addEdge(4, 3, 5);
    graph.addEdge(4, 2, 8);
    graph.addEdge(4, 6, 21);
    graph.addEdge(4, 5, 34);

    graph.addEdge(5, 6, 45);
    graph.addEdge(5, 4, 34);

    graph.addEdge(6, 1, 57);
    graph.addEdge(6, 2, 13);
    graph.addEdge(6, 4, 21);
    graph.addEdge(6, 5, 45);


    update();
    ui.statusText->setText("Граф задания создан.");
}
void MainWindow::deleteGraph(){
    graph.clearGraph();

    ui.statusText->setText("Граф удалён.");
    update();
}
```

```cpp
// Визуализация графа
void MainWindow::paintEvent(QPaintEvent* event){
    QPainter painter(this);

    QFont font = painter.font();
    font.setPointSize(16);
    painter.setFont(font);
    painter.setPen(QPen(Qt::cyan));

    for (const auto& pair : graph.nodes_map) {
        Node* node = pair.second;

        for (Edge* edge : node->edges_to_node) {
            painter.setOpacity(0.2);
            QPoint edgeStart;
            QPoint edgeEnd;

            double angles = atan2(-(edge->to->pos.y() - node->pos.y()), (edge->to->pos.x()-node->pos.x()));

            edgeStart = QPoint(node->pos.x()+20*cos(angles), node->pos.y() - 20*sin(angles));
            edgeEnd = QPoint(edge->to->pos.x()- 20 * cos(angles), edge->to->pos.y() + 20*sin(angles));

            painter.drawLine(edgeStart, edgeEnd);

            int x_t = edgeStart.x() + 4 * (edgeEnd.x() - edgeStart.x()) / 5 ;
            int y_t = edgeStart.y() - 4 * (edgeStart.y() - edgeEnd.y()) / 5;

            painter.setPen(QPen(Qt::black, 2));
            painter.setOpacity(1);
            painter.drawText(x_t-10, y_t+10, QString::number(edge->weight));
            painter.setOpacity(0.2);
            painter.setPen(QPen(Qt::cyan, 2));

            QLine line(edgeStart, edgeEnd);

            double angle = atan2(-line.dy(), line.dx())-M_PI/2;
            double arrowSize = 20;

            QPointF arrowP1 = edgeEnd + QPointF(sin(angle - M_PI / 12) * arrowSize, cos(angle - M_PI / 12) * arrowSize);
            QPointF arrowP2 = edgeEnd + QPointF(sin(angle + M_PI / 12) * arrowSize, cos(angle + M_PI / 12) * arrowSize);

            QPolygonF arrowHead;

            arrowHead << edgeEnd << arrowP1 << arrowP2;

            QPainterPath path;

            path.moveTo(edgeEnd);
            path.lineTo(arrowP1);
            path.lineTo(arrowP2);
            painter.fillPath(path, Qt::darkCyan);
            painter.drawPolygon(arrowHead);
            painter.setOpacity(1);
        }
    }
    painter.setBrush(Qt::NoBrush);
    painter.setPen(QPen(Qt::white, 2));

    for (const auto& pair : graph.nodes_map) {
```

```
    for (const auto& pair : graph.nodes_map) {
        Node* node = pair.second;
        painter.drawEllipse(node->pos, 20, 20);
        painter.setPen(QPen(Qt::black, 2));
        painter.drawText(node->pos.x() - 9, node->pos.y() + 8, QString::number(node->data));
        painter.setPen(QPen(Qt::white, 2));
    }

    if (nodeSelection){
        painter.drawEllipse(100,100, 40, 40);
        painter.setBrush(Qt::yellow);
        painter.drawEllipse(currentNode->pos, 20, 20);
        painter.setPen(QPen(Qt::black, 2));
        painter.drawText(currentNode->pos.x() - 9, currentNode->pos.y() + 8, QString::number(currentNode->data));
    }
}
void MainWindow::mousePressEvent(QMouseEvent* event){
    if (event->button() == Qt::LeftButton){
        nodeMoving = false;
        for (const auto& pair : graph.nodes_map){
            Node* node = pair.second;
            if ((event->pos() - node->pos).manhattanLength() < 30){
                selectedNode = node;
                nodeMoving = true;
                break;
            }
        }
        update();
    }
}
void MainWindow::mouseMoveEvent(QMouseEvent* event){
    if (nodeMoving && selectedNode){
        selectedNode->pos = event->pos();
        update();
    }
}
void MainWindow::mouseReleaseEvent(QMouseEvent* event){
    if (event->button() == Qt::LeftButton && nodeMoving){
        nodeMoving = false;
        selectedNode = nullptr;
        update();
    }
}
```

```
// Вывод таблицы смежностей узлов
void MainWindow::printTable(){
    printAdjTable(graph);
}
void MainWindow::printAdjTable(const Graph& graph){
    QString result;
    for (const auto& pair : graph.nodes_map){
        int node = pair.first;
        Node* nodeConnections = pair.second;

        result += "Узел " + QString::number(node) + ": \n";
        unordered_set<int> printedNodes;
        for (Edge* edge : nodeConnections->edges_to_node){
            if (printedNodes.find(edge->to->data) == printedNodes.end()){
                result += "    Связь с " + QString::number(edge->to->data) + ", вес ребра - " + QString::number(edge->weight) + ". \n";
                printedNodes.insert(edge->to->data);
            }
        }
        ui.statusText->setText(result);
    }
}

MainWindow::~MainWindow(){}
```