```
#include <iostream>
#include <vector>

#define INT_MAX 10000000

using namespace std;

void DijkstrasTest();

int main()
{
    DijkstrasTest();
    return 0;
}

class Node;
class Edge;

void Dijkstras();
vector<Node *> *AdjacentRemainingNodes(Node *node);
Node *ExtractSmallest(vector<Node *> &nodes);
int Distance(Node *node1, Node *node2);
bool Contains(vector<Node *> &nodes, Node *node);
void PrintShortestRouteTo(Node *destination);

vector<Node *> nodes;
vector<Edge *> edges;

class Node
{
public:
    Node(char id)
        : id(id), previous(NULL), distanceFromStart(INT_MAX)
    {
        nodes.push_back(this);
    }

public:
    char id;
    Node *previous;
    int distanceFromStart;
};

class Edge
{
public:
    Edge(Node *node1, Node *node2, int distance)
        : node1(node1), node2(node2), distance(distance)
    {
        edges.push_back(this);
    }
    bool Connects(Node *node1, Node *node2)
    {
        return (
            (node1 == this->node1 &&
             node2 == this->node2) ||
```

```
            (node1 == this->node2 &&
             node2 == this->node1));
    }

public:
    Node *node1;
    Node *node2;
    int distance;
};

//////////////////
void DijkstrasTest()
{
    Node *a = new Node('a');
    Node *b = new Node('b');
    Node *c = new Node('c');
    Node *d = new Node('d');
    Node *e = new Node('e');
    Node *f = new Node('f');
    Node *g = new Node('g');

    Edge *e1 = new Edge(a, c, 1);
    Edge *e2 = new Edge(a, d, 2);
    Edge *e3 = new Edge(b, c, 2);
    Edge *e4 = new Edge(c, d, 1);
    Edge *e5 = new Edge(b, f, 3);
    Edge *e6 = new Edge(c, e, 3);
    Edge *e7 = new Edge(e, f, 2);
    Edge *e8 = new Edge(d, g, 1);
    Edge *e9 = new Edge(g, f, 1);

    a->distanceFromStart = 0; // set start node
    Dijkstras();
    PrintShortestRouteTo(f);
}

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

void Dijkstras()
{
    while (nodes.size() > 0)
    {
        Node *smallest = ExtractSmallest(nodes);
        vector<Node *> *adjacentNodes =
            AdjacentRemainingNodes(smallest);

        const int size = adjacentNodes->size();
        for (int i = 0; i < size; ++i)
        {
            Node *adjacent = adjacentNodes->at(i);
            int distance = Distance(smallest, adjacent) +
                           smallest->distanceFromStart;

            if (distance < adjacent->distanceFromStart)
            {
                adjacent->distanceFromStart = distance;
                adjacent->previous = smallest;
```

```
            }
        }
        delete adjacentNodes;
    }
}

// Find the node with the smallest distance,
// remove it, and return it.
Node *ExtractSmallest(vector<Node *> &nodes)
{
    int size = nodes.size();
    if (size == 0)
        return NULL;
    int smallestPosition = 0;
    Node *smallest = nodes.at(0);
    for (int i = 1; i < size; ++i)
    {
        Node *current = nodes.at(i);
        if (current->distanceFromStart <
            smallest->distanceFromStart)
        {
            smallest = current;
            smallestPosition = i;
        }
    }
    nodes.erase(nodes.begin() + smallestPosition);
    return smallest;
}

// Return all nodes adjacent to 'node' which are still
// in the 'nodes' collection.
vector<Node *> *AdjacentRemainingNodes(Node *node)
{
    vector<Node *> *adjacentNodes = new vector<Node *>();
    const int size = edges.size();
    for (int i = 0; i < size; ++i)
    {
        Edge *edge = edges.at(i);
        Node *adjacent = NULL;
        if (edge->node1 == node)
        {
            adjacent = edge->node2;
        }
        else if (edge->node2 == node)
        {
            adjacent = edge->node1;
        }
        if (adjacent && Contains(nodes, adjacent))
        {
            adjacentNodes->push_back(adjacent);
        }
    }
    return adjacentNodes;
}

// Return distance between two connected nodes
int Distance(Node *node1, Node *node2)
```

```
{
    const int size = edges.size();
    for (int i = 0; i < size; ++i)
    {
        Edge *edge = edges.at(i);
        if (edge->Connects(node1, node2))
        {
            return edge->distance;
        }
    }
    return -1; // should never happen
}

// Does the 'nodes' vector contain 'node'
bool Contains(vector<Node *> &nodes, Node *node)
{
    const int size = nodes.size();
    for (int i = 0; i < size; ++i)
    {
        if (node == nodes.at(i))
        {
            return true;
        }
    }
    return false;
}

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

void PrintShortestRouteTo(Node *destination)
{
    Node *previous = destination;
    cout << "Distance from start: "
         << destination->distanceFromStart << endl;
    while (previous)
    {
        cout << previous->id << " ";
        previous = previous->previous;
    }
    cout << endl;
}

// these two not needed
vector<Edge *> *AdjacentEdges(vector<Edge *> &Edges, Node *node);
void RemoveEdge(vector<Edge *> &Edges, Edge *edge);

vector<Edge *> *AdjacentEdges(vector<Edge *> &edges, Node *node)
{
    vector<Edge *> *adjacentEdges = new vector<Edge *>();

    const int size = edges.size();
    for (int i = 0; i < size; ++i)
    {
        Edge *edge = edges.at(i);
        if (edge->node1 == node)
        {
            cout << "adjacent: " << edge->node2->id << endl;
```

```
            adjacentEdges->push_back(edge);
        }
        else if (edge->node2 == node)
        {
            cout << "adjacent: " << edge->node1->id << endl;
            adjacentEdges->push_back(edge);
        }
    }
    return adjacentEdges;
}

void RemoveEdge(vector<Edge *> &edges, Edge *edge)
{
    vector<Edge *>::iterator it;
    for (it = edges.begin(); it < edges.end(); ++it)
    {
        if (*it == edge)
        {
            edges.erase(it);
            return;
        }
    }
}
```