```
/*
 * CS 261 Data Structure
 * Assignment 7, graph BFS & DFS implementation
 * Name: Jacob Karcz
 * Date: 11.18.2016
 */

/
    ****************************************************************************
    **************

 ******************                    File: graph.h
     ****************


     ***************************************************************************
     **************/

#ifndef GRAPH_H
#define GRAPH_H

typedef struct Vertex Vertex;
typedef struct Graph Graph;

struct Vertex {
    int label;
    int isVisited;
    int numNeighbors;
    Vertex** neighbors;
};

struct Graph {
    int numEdges;
    int numVertices;
    Vertex* vertexSet;
};

int dfsRecursive(Graph* graph, Vertex* source, Vertex* destination);
int dfsIterative(Graph* graph, Vertex* source, Vertex* destination);
int bfsIterative(Graph* graph, Vertex* source, Vertex* destination);

Graph* randomGraph(int numVertices, int numEdges);
Graph* loadGraph(const char* fileName);
void freeGraph(Graph* graph);
void printGraph(Graph* graph);

#endif


/
    ****************************************************************************
    **************

 ******************                    File: graph.c
     ****************
```

```
     ****************************************************************************
     **************/

#include "graph.h"
#include "deque.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

/
    ****************************************************************************
    **************
 * Clears isVisited in all nodes in the graph.
 * param graph

     ****************************************************************************
     **************/
static void clearVisited(Graph* graph)
{
    for (int i = 0; i < graph->numVertices; ++i)
    {
        graph->vertexSet[i].isVisited = 0;
    }
}

/
    ****************************************************************************
    **************
 * Recursive helper function for DfsRecursive. Determines if there is a path
 * from the given vertex to the destination using a recursive depth-first
 * search.
 * param graph
 * param vertex
 * param destination
 * return 1 if there is a path, 0 otherwise.

     ****************************************************************************
     **************/
static int DfsRecursiveHelper(Graph* graph, Vertex* vertex, Vertex* destination
    )
{
    vertex->isVisited = 1;
    if (vertex == destination) {
        return 1;
    }
    for (int i = 0; i < vertex->numNeighbors; ++i) {
        Vertex* neighbor = vertex->neighbors[i];
        if (!neighbor->isVisited) {
            if (DfsRecursiveHelper(graph, neighbor, destination) == 1) {
                return 1;
            }
        }
    }
    return 0;
```

```
}

/
    **************************************************************************
    **************
 * Determines if an edge (v1, v2) exists.
 * param v1
 * param v2
 * return 1 if the edge exists, 0 otherwise.

     **************************************************************************
     **************/
static int isAdjacent(Vertex* v1, Vertex* v2) {
    if (v1 == v2) {
        return 1;
    }
    for (int i = 0; i < v1->numNeighbors; ++i) {
        if (v1->neighbors[i] == v2) {
            return 1;
        }
    }
    return 0;
}

/
    **************************************************************************
    **************
 * Connects two vertices by adding each other to their neighbors lists.
 * param v1
 * param v2

     **************************************************************************
     **************/
static void createEdge(Vertex* v1, Vertex* v2)
{
    v1->neighbors = realloc(v1->neighbors,
                            sizeof(Vertex*) * (v1->numNeighbors + 1));
    v2->neighbors = realloc(v2->neighbors,
                            sizeof(Vertex*) * (v2->numNeighbors + 1));
    v1->neighbors[v1->numNeighbors] = v2;
    v2->neighbors[v2->numNeighbors] = v1;
    ++(v1->numNeighbors);
    ++(v2->numNeighbors);
}

/
    **************************************************************************
    **************
 * Determines if there is a path from the source to the destination using a
 * recursive depth-first search starting at the source.
 *
 * You can use this function to test the correctness of the others.
 *
 * param graph
 * param source
 * param destination
```

```
 * return 1 if there is a path, 0 otherwise.

     ***************************************************************************
     **************/
int dfsRecursive(Graph* graph, Vertex* source, Vertex* destination)
{
    clearVisited(graph);
    return DfsRecursiveHelper(graph, source, destination);
}

/
    ***************************************************************************
    **************
 * Determines if there is a path from the source to the destination using an
 * iterative depth-first search starting at the source.
 *
 * Remember to call clearVisited() before starting the search.
 *
 * param graph
 * param source <------get from here
 * param destination <------ to here
 * return 1 if there is a path, 0 otherwise.

     ***************************************************************************
     **************/
int dfsIterative(Graph* graph, Vertex* source, Vertex* destination) { //
    <------------------<<<<<<<<<<<<
    // FIXME: Implement
    //Depth First Search, implemented with a linked list deque (using stack
        functions)
    /*
     1. create a set of reachable vertices, initially empty. reachable stack?
     2. create a container for vertices known to be reachable. ignore for this
         function
     3. add starting vertex/node to the stack
     4. enter while loop
        while (stack is not empty) {
            pop top from stack, assign it to v
            if (v is not already in set of reachable vertices (reachable array)
                {
                add v to the array of reachable nodes
                push the neighbors of v (not already in reachable array) to the
                    stack
            }
        }
    */

    clearVisited(graph);
    Vertex *current = source;
    Deque *reachables = dequeNew();  //deque of reachable vertices

    dequePushFront(reachables, current);
    current->isVisited = 1;

    if (source == destination) {
        dequeDelete(reachables);
```

```
        return 1;
    }
    while (! dequeIsEmpty(reachables)) {
        current = dequeFront(reachables);
        dequePopFront(reachables);

        if (current->isVisited == 0)
            current->isVisited = 1;
        if (current == destination) {
            dequeDelete(reachables);
            return 1;
        }
        for (int i = 0; i < current->numNeighbors; i++) {
            if (current->neighbors[i]->isVisited == 0)
                dequePushFront(reachables, current->neighbors[i]);
            //if (current->neighbors[i] == destination)
             //   return 1;

        }
    }
    dequeDelete(reachables);
    return 0;

}

/
    ****************************************************************************
    **************
 * Determines if there is a path from the source to the destination using an
 * iterative breadth-first search starting at the source.
 *
 * Remember to call clearVisited() before starting the search.
 *
 * param graph
 * param source
 * param destination
 * return 1 if there is a path, 0 otherwise.

     ****************************************************************************
     **************/
int bfsIterative(Graph* graph, Vertex* source, Vertex* destination) { //
    <------------------<<<<<<<<<<<<
    // FIXME: Implement
    //Depth First Search, implemented with a linked list deque (using queue
        functions)
    /*
     1. create a set of reachable vertices, initially empty. reachable  queue
     2. create a container for vertices known to be reachable. ignore for this
         function
     3. add starting vertex/node to the queue
     4. enter while loop
        while (queue is not empty) {
            dequeue last entry, assign it to v
            if (v is not already in set of reachable vertices (reachable array)
                {
                add v to the array of reachable nodes
```

```
                enqueue neighbors of v not already in reachable array
            }
        }
     */
    clearVisited(graph);
    Vertex *current = source;
    Deque *reachable = dequeNew();  //deque of reachable vertices

    dequePushFront(reachable, current);
    current->isVisited = 1;

    if (source == destination) {
        dequeDelete(reachable);
        return 1;
    }

    while (! dequeIsEmpty(reachable)) {
        current = dequeBack(reachable);
        dequePopBack(reachable);

        if (! current->isVisited)
            current->isVisited = 1;
        if (current == destination) {
            dequeDelete(reachable);
            return 1;
        }
        for (int i = 0; i < current->numNeighbors; i++) {
            if (current->neighbors[i]->isVisited == 0)
                dequePushFront(reachable, current->neighbors[i]);
        }
    }
    dequeDelete(reachable);
    return 0;
}


/
    ***************************************************************************
    **************
 * Generates a set of random unique edges of size numEdges sampled from the
 * set of all possible edges.
 * param numVertices
 * param numEdges
 * return An array of numEdges edges.

     ***************************************************************************
     **************/
typedef struct Edge Edge;

struct Edge
{
    int i;
    int j;
};
```

```
Edge* randomEdges(int numVertices, int numEdges)
{
    assert(numVertices > 0);
    int maxEdges = numVertices * (numVertices - 1) / 2;
    assert(numEdges >= 0);
    assert(numEdges <= maxEdges);

    // Generate all possible edges
    Edge* edges = malloc(sizeof(Edge) * maxEdges);
    int k = 0;
    for (int i = 0; i < numVertices; ++i) {
        for (int j = i + 1; j < numVertices; ++j) {
            edges[k].i = i;
            edges[k].j = j;
            ++k;
        }
    }

    // Shuffle edges
    for (int i = maxEdges - 1; i > 0; --i) {
        int j = rand() % (i + 1);
        Edge temp = edges[i];
        edges[i] = edges[j];
        edges[j] = temp;
    }

    // Take only the number of edges needed
    edges = realloc(edges, sizeof(Edge) * numEdges);
    return edges;
}

/
    ****************************************************************************
    **************
 * Given a number of vertices and a number of edges, generates a graph
 * connecting random pairs of vertices. The edges are unique, and thus their is
 * a maximum number of edges allowed in proportion to the number of vertices.
 * numEdges must be in the interval [0, numVertices * (numVertices + 1) / 2].
 * param numVertices
 * param numEdges
 * return

     ***************************************************************************
     **************/
Graph* randomGraph(int numVertices, int numEdges)
{
    assert(numVertices > 0);
    assert(numEdges >= 0);
    assert(numEdges <= numVertices * (numVertices - 1) / 2);

    Graph* graph = malloc(sizeof(Graph));
    graph->numVertices = numVertices;
    graph->numEdges = numEdges;
    graph->vertexSet = malloc(sizeof(Vertex) * numVertices);

    // Initialize vertices
```

```
    for (int i = 0; i < graph->numVertices; ++i) {
        Vertex* vertex = &graph->vertexSet[i];
        vertex->label = i;
        vertex->isVisited = 0;
        vertex->numNeighbors = 0;
        vertex->neighbors = NULL;
    }

    // Randomly connect vertices
    Edge* edges = randomEdges(numVertices, numEdges);
    for (int i = 0; i < numEdges; ++i) {
        Vertex* v1 = &graph->vertexSet[edges[i].i];
        Vertex* v2 = &graph->vertexSet[edges[i].j];
        createEdge(v1, v2);
    }
    free(edges);

    return graph;
}

/
    ****************************************************************************
    **************
 * Loads a graph from the given file. The file's first line must be the number
 * of vertices in the graph and each consecutive line must be a list of numbers
 * separated by spaces. The first number is the next vertex and the following
 * numbers are its neighbors.
 * param fileName
 * return

     ***************************************************************************
     **************/
Graph* loadGraph(const char* fileName)
{
    FILE* file = fopen(fileName, "r");
    char buffer[512];

    // Get the number of vertices
    fgets(buffer, sizeof buffer, file);
    int numVertices = (int) strtol(buffer, NULL, 10);
    Graph* graph = malloc(sizeof(Graph));
    graph->numVertices = numVertices;
    graph->numEdges = 0;

    // Initialize vertices
    graph->vertexSet = malloc(sizeof(Vertex) * numVertices);
    for (int i = 0; i < numVertices; ++i) {
        Vertex* vertex = &graph->vertexSet[i];
        vertex->isVisited = 0;
        vertex->label = i;
        vertex->neighbors = NULL;
        vertex->numNeighbors = 0;
    }

    // Create edges
    while (fgets(buffer, sizeof buffer, file) != NULL) {
```

```
        char* begin = buffer;
        char* end = NULL;

        // Get vertex
        int i = (int) strtol(begin, &end, 10);
        Vertex* vertex = &graph->vertexSet[i];
        begin = end;

        // Create edges
        for (int i = (int) strtol(begin, &end, 10);
            end != begin;
            i = (int) strtol(begin, &end, 10)) {
            Vertex* neighbor = &graph->vertexSet[i];
            if (!isAdjacent(vertex, neighbor)) {
                createEdge(vertex, neighbor);
                ++(graph->numEdges);
            }
            begin = end;
        }
    }
    fclose(file);

    return graph;
}

/
    ****************************************************************************
    **************
 * Frees all memory allocated for a graph and the graph itself.
 * param graph

     ***************************************************************************
     **************/
void freeGraph(Graph* graph) {
    for (int i = 0; i < graph->numVertices; ++i) {
        free(graph->vertexSet[i].neighbors);
    }
    free(graph->vertexSet);
    free(graph);
}

/
    ****************************************************************************
    **************
 * Prints the vertex count, edge count, and adjacency list for each vertex.
 * param graph

     ***************************************************************************
     **************/
void printGraph(Graph* graph) {
    printf("Vertex count: %d\n", graph->numVertices);
    printf("Edge count: %d\n", graph->numEdges);
    for (int i = 0; i < graph->numVertices; ++i) {
        Vertex* vertex = &graph->vertexSet[i];
        printf("%d :", vertex->label);
        for (int j = 0; j < vertex->numNeighbors; ++j) {
```

```
            printf(" %d", vertex->neighbors[j]->label);
        }
        printf("\n");
    }
}









/
    ******************************************************************************
    **************

 ******************                     File: deque.h
     ****************


     ******************************************************************************
     **************/

#ifndef DEQUE_H
#define DEQUE_H


typedef void* Type;
typedef struct Link Link;
typedef struct Deque Deque;

struct Link {
    Type value;
    Link* next;
    Link* prev;
};

struct Deque {
    int size;
    Link* sentinel;
};

Deque* dequeNew();
void dequeDelete(Deque* deque);
void dequePushFront(Deque* deque, Type value); //push
void dequePushBack(Deque* deque, Type value);  //enqueue
Type dequeFront(Deque* deque);                  //top
Type dequeBack(Deque* deque);                   //peek?
void dequePopFront(Deque* deque);               //pop
void dequePopBack(Deque* deque);                //dequeue
int dequeIsEmpty(Deque* deque);
void dequeClear(Deque* deque);
```

```
#endif



/
    ***************************************************************************
    **************

 ******************                    File: deque.c
     ****************


     ***************************************************************************
     **************/

#include "deque.h"
#include <stdlib.h>
#include <assert.h>

/**
 * Frees the given link and connects its previous and next links.
 * @param link
 */
static void removeLink(Link* link)
{
    link->prev->next = link->next;
    link->next->prev = link->prev;
    free(link);
}

/**
 * Creates a new link with the given value and inserts it after the given link.
 * @param link
 * @param value
 */
static void addLinkAfter(Link* link, Type value)
{
    Link* newLink = malloc(sizeof(Link));
    newLink->value = value;
    newLink->next = link->next;
    newLink->prev = link;
    newLink->next->prev = newLink;
    newLink->prev->next = newLink;
}

/**
 * Allocates a new deque, initializes it, and returns it.
 * @return new initialized deque.
 */
Deque* dequeNew()
{
    Deque* deque = malloc(sizeof(Deque));
    assert(deque != NULL);
    Link* sentinel = malloc(sizeof(Link));
    sentinel->next = sentinel;
```

```c
    sentinel->prev = sentinel;
    deque->sentinel = sentinel;
    deque->size = 0;
    return deque;
}

/**
 * Frees allocated memory in the given deque and the deque itself.
 * @param deque
 */
void dequeDelete(Deque* deque)
{
    dequeClear(deque);
    free(deque->sentinel);
    free(deque);
}

/**
 * Creates a new link with the given value and inserts it at the front of the
 * deque. Also increments the size of the deque.
 * @param deque
 * @param value
 */
void dequePushFront(Deque* deque, Type value)
{
    addLinkAfter(deque->sentinel, value);
    ++(deque->size);
}

/**
 * Creates a new link with the given value and inserts it at the back of the
 * deque. Also increments the size of the deque.
 * @param deque
 * @param value
 */
void dequePushBack(Deque* deque, Type value)
{
    addLinkAfter(deque->sentinel->prev, value);
    ++(deque->size);
}

/**
 * Returns the value of the link at the front of the deque.
 * @param deque
 * @return value of front link.
 */
Type dequeFront(Deque* deque)
{
    assert(deque->size > 0);
    return deque->sentinel->next->value;
}

/**
 * Returns the value of the link at the back of the deque.
 * @param deque
 * @return value of back link.
```

```
 */
Type dequeBack(Deque* deque)
{
    assert(deque->size > 0);
    return deque->sentinel->prev->value;
}

/**
 * Removes and frees the link at the front of the deque. Also decrements the
 * size of the deque.
 * @param deque
 */
void dequePopFront(Deque* deque)
{
    assert(deque->size > 0);
    removeLink(deque->sentinel->next);
    --(deque->size);
}

/**
 * Removes and frees the link at the back of the deque. Also decrements the
 * size of the deque.
 * @param deque
 */
void dequePopBack(Deque* deque)
{
    assert(deque->size > 0);
    removeLink(deque->sentinel->prev);
    --(deque->size);
}

/**
 * Returns 1 if the deque is empty and 0 otherwise.
 * @param deque
 * @return 1 if empty, 0 otherwise.
 */
int dequeIsEmpty(Deque* deque)
{
    return deque->size == 0;
}

/**
 * Removes and frees all links in the deque, except the sentinel.
 * @param deque
 */
void dequeClear(Deque* deque)
{
    while (!dequeIsEmpty(deque))
    {
        dequePopFront(deque);
    }
}
```