**Output:**

Result for Maze #1 (should be reachable, expect 1): 1

Result for Maze #2 (should NOT be reachable, expect 0): 0

**Graph.h**

/\*

 \* graph.h

 \*/

#ifndef GRAPH\_H

#define GRAPH\_H

#include <stdlib.h>

#include "list.h"

#include "set.h"

/\* Define a structure for adjacency lists. \*/

typedef struct AdjList\_ {

    void \*vertex;

    Set adjacent;

} AdjList;

/\* Define a structure for graphs. \*/

typedef struct Graph\_ {

    int vcount;

    int ecount;

    int (\*match)(const void \*key1, const void \*key2);

    void (\*destroy)(void \*data);

    List adjlists;

} Graph;

/\* Define colors for vertices in graphs. \*/

typedef enum VertexColor\_ {

    white, gray, black

} VertexColor;

/\* Public Interface \*/

void graph\_init(Graph \*graph, int(\*match)(const void \*key1, const void \*key2),

        void(\*destroy)(void \*data));

void graph\_destroy(Graph \*graph);

int graph\_ins\_vertex(Graph \*graph, const void \*data);

int graph\_ins\_edge(Graph \*graph, const void \*data1, const void \*data2);

int graph\_rem\_vertex(Graph \*graph, void \*\*data);

int graph\_rem\_edge(Graph \*graph, void \*data1, void \*\*data2);

int graph\_adjlist(const Graph \*graph, const void \*data, AdjList \*\*adjlist);

int graph\_is\_adjacent(const Graph \*graph, const void \*data1, const void \*data2);

int isExitReachable(Graph \*pMaze, char entrance, char exit);

#define graph\_adjlists(graph) ((graph)->adjlists)

#define graph\_vcount(graph) ((graph)->vcount)

#define graph\_ecount(graph) ((graph)->ecount)

#endif

**Graph.c**

/\*

 \* graph.c

 \*/

#include <stdlib.h>

#include <string.h>

#include "graph.h"

#include "list.h"

#include "set.h"

void graph\_init(Graph \*graph, int(\*match)(const void \*key1, const void \*key2),

        void(\*destroy)(void \*data)) {

    /\* Initialize the graph. \*/

    graph->vcount = 0;

    graph->ecount = 0;

    graph->match = match;

    graph->destroy = destroy;

    /\* Initialize the list of adjacency-list structures. \*/

    list\_init(&graph->adjlists, NULL);

}

void graph\_destroy(Graph \*graph) {

    AdjList \*adjlist;

    /\* Remove each adjacency-list structure and destroy its adjacency list. \*/

    while (list\_size(&graph->adjlists) > 0) {

        if (list\_rem\_next(&graph->adjlists, NULL, (void \*\*) &adjlist) == 0) {

            set\_destroy(&adjlist->adjacent);

            if (graph->destroy != NULL)

                graph->destroy(adjlist->vertex);

            free(adjlist);

        }

    }

    /\* Destroy the list of adjacency-list structures, which is now empty. \*/

    list\_destroy(&graph->adjlists);

    /\* No operations are allowed now, but clear the structure as a

     \* precaution. \*/

    memset(graph, 0, sizeof(Graph));

}

int graph\_ins\_vertex(Graph \*graph, const void \*data) {

    ListElmt \*element;

    AdjList \*adjlist;

    int retval;

    /\* Do not allow the insertion of duplicate vertices. \*/

    for (element = list\_head(&graph->adjlists); element != NULL; element

            = list\_next(element)) {

        if (graph->match(data, ((AdjList \*) list\_data(element))->vertex))

            return 1;

    }

    /\* Insert the vertex. \*/

    if ((adjlist = (AdjList \*) malloc(sizeof(AdjList))) == NULL)

        return -1;

    adjlist->vertex = (void \*) data;

    set\_init(&adjlist->adjacent, graph->match, graph->destroy);

    if ((retval = list\_ins\_next(&graph->adjlists, list\_tail(&graph->adjlists),

            adjlist)) != 0) {

        return retval;

    }

    /\* Adjust the vertex count to account for the inserted vertex. \*/

    graph->vcount++;

    return 0;

}

int graph\_ins\_edge(Graph \*graph, const void \*data1, const void \*data2) {

    ListElmt \*element;

    int retval;

    /\* Do not allow insertion of an edge without both its vertices in the

     \* graph. \*/

    for (element = list\_head(&graph->adjlists); element != NULL; element

            = list\_next(element)) {

        if (graph->match(data2, ((AdjList \*) list\_data(element))->vertex))

            break;

    }

    if (element == NULL)

        return -1;

    for (element = list\_head(&graph->adjlists); element != NULL; element

            = list\_next(element)) {

        if (graph->match(data1, ((AdjList \*) list\_data(element))->vertex))

            break;

    }

    if (element == NULL)

        return -1;

    /\* Insert the second vertex into the adjacency list of the first vertex. \*/

    if ((retval

            = set\_insert(&((AdjList \*) list\_data(element))->adjacent, data2))

            != 0) {

        return retval;

    }

    /\* Adjust the edge count to account for the inserted edge. \*/

    graph->ecount++;

    return 0;

}

int graph\_rem\_vertex(Graph \*graph, void \*\*data) {

    ListElmt \*element, \*temp, \*prev;

    AdjList \*adjlist;

    int found;

    /\* Traverse each adjacency list and the vertices it contains. \*/

    prev = NULL;

    found = 0;

    for (element = list\_head(&graph->adjlists); element != NULL; element

            = list\_next(element)) {

        /\* Do not allow removal of the vertex if it is in an adjacency list. \*/

        if (set\_is\_member(&((AdjList \*) list\_data(element))->adjacent, \*data))

            return -1;

        /\* Keep a pointer to the vertex to be removed. \*/

        if (graph->match(\*data, ((AdjList \*) list\_data(element))->vertex)) {

            temp = element;

            found = 1;

        }

        /\* Keep a pointer to the vertex before the vertex to be removed. \*/

        if (!found)

            prev = element;

    }

    /\* Return if the vertex was not found. \*/

    if (!found)

        return -1;

    /\* Do not allow removal of the vertex if its adjacency list is not empty. \*/

    if (set\_size(&((AdjList \*)list\_data(temp))->adjacent) > 0)

        return -1;

    /\* Remove the vertex. \*/

    if (list\_rem\_next(&graph->adjlists, prev, (void \*\*) &adjlist) != 0)

        return -1;

    /\* Free the storage allocated by the abstract data type. \*/

    \*data = adjlist->vertex;

    free(adjlist);

    /\* Adjust the vertex count to account for the removed vertex. \*/

    graph->vcount--;

    return 0;

}

int graph\_rem\_edge(Graph \*graph, void \*data1, void \*\*data2) {

    ListElmt \*element;

    /\* Locate the adjacency list for the first vertex. \*/

    for (element = list\_head(&graph->adjlists); element != NULL; element

            = list\_next(element)) {

        if (graph->match(data1, ((AdjList \*) list\_data(element))->vertex))

            break;

    }

    if (element == NULL)

        return -1;

    /\* Remove the second vertex from the adjacency list of the first vertex. \*/

    if (set\_remove(&((AdjList \*) list\_data(element))->adjacent, data2) != 0)

        return -1;

    /\* Adjust the edge count to account for the removed edge. \*/

    graph->ecount--;

    return 0;

}

int graph\_adjlist(const Graph \*graph, const void \*data, AdjList \*\*adjlist) {

    ListElmt \*element, \*prev;

    /\* Locate the adjacency list for the vertex. \*/

    prev = NULL;

    for (element = list\_head(&graph->adjlists); element != NULL; element

            = list\_next(element)) {

        if (graph->match(data, ((AdjList \*) list\_data(element))->vertex))

            break;

        prev = element;

    }

    /\* Return if the vertex was not found. \*/

    if (element == NULL)

        return -1;

    /\* Pass back the adjacency list for the vertex. \*/

    \*adjlist = (AdjList\*)list\_data(element);

    return 0;

}

int graph\_is\_adjacent(const Graph \*graph, const void \*data1, const void \*data2) {

    ListElmt \*element, \*prev;

    /\* Locate the adjacency list of the first vertex. \*/

    prev = NULL;

    for (element = list\_head(&graph->adjlists); element != NULL; element

            = list\_next(element)) {

        if (graph->match(data1, ((AdjList \*) list\_data(element))->vertex))

            break;

        prev = element;

    }

    /\* Return if the first vertex was not found. \*/

    if (element == NULL)

        return 0;

    /\* Return whether the second vertex is in the adjacency list of the first. \*/

    return set\_is\_member(&((AdjList \*) list\_data(element))->adjacent, data2);

}

int match(const void \*key1, const void \*key2) {

    return \*(const char \*)key1 == \*(const char \*)key2;

}

static int dfs(Graph \*graph, AdjList \*adjlist, const void \*exit, Set \*visited) {

    /\* Check if this vertex is the exit. \*/

    if (graph->match(adjlist->vertex, exit)) {

        return 1;

    }

    /\* Add this vertex to the visited set. \*/

    set\_insert(visited, adjlist->vertex);

    /\* Recursively apply DFS to unvisited adjacent vertices. \*/

    ListElmt \*element;

    void \*adjacent;

    AdjList \*adjlist\_next;

    for (element = list\_head(&adjlist->adjacent); element != NULL; element = list\_next(element)) {

        adjacent = list\_data(element);

        if (!set\_is\_member(visited, adjacent)) {

            if (graph\_adjlist(graph, adjacent, &adjlist\_next) == 0 && dfs(graph, adjlist\_next, exit, visited)) {

                return 1;

            }

        }

    }

    return 0;

}

int isExitReachable(Graph \*pMaze, char entrance, char exit) {

    Set visited;

    AdjList \*adjlist;

    char entrance\_vertex = entrance;

    char exit\_vertex = exit;

    /\* Initialize the visited set. \*/

    set\_init(&visited, pMaze->match, NULL);

    /\* Start DFS from the entrance vertex. \*/

    if (graph\_adjlist(pMaze, &entrance\_vertex, &adjlist) == 0) {

        int result = dfs(pMaze, adjlist, &exit\_vertex, &visited);

        set\_destroy(&visited);

        return result;

    }

    set\_destroy(&visited);

    return 0;

}

**mazeTest.c**

#include "graph.h"

#include <stdio.h>

int match(const void \*key1, const void \*key2);

void (\*destroy)(void \*data);

int main() {

    Graph maze1, maze2;

    char vertices[] = {'A', 'B', 'C', 'D', 'E', 'F', 'G'};

    int i;

    /\* Initialize both mazes. \*/

    graph\_init(&maze1, match, \*destroy);

    graph\_init(&maze2, match, \*destroy);

    /\* Insert vertices for both mazes. \*/

    for (i = 0; i < sizeof(vertices) / sizeof(char); i++) {

        graph\_ins\_vertex(&maze1, &vertices[i]);

        graph\_ins\_vertex(&maze2, &vertices[i]);

    }

    /\* Insert edges for Maze #1. \*/

    graph\_ins\_edge(&maze1, &vertices[0], &vertices[2]); // A-C

    graph\_ins\_edge(&maze1, &vertices[0], &vertices[3]); // A-D

    graph\_ins\_edge(&maze1, &vertices[3], &vertices[1]); // D-B

    graph\_ins\_edge(&maze1, &vertices[2], &vertices[5]); // C-F

    graph\_ins\_edge(&maze1, &vertices[5], &vertices[6]); // F-G

    graph\_ins\_edge(&maze1, &vertices[3], &vertices[6]); // D-G

    graph\_ins\_edge(&maze1, &vertices[3], &vertices[4]); // D-E

    graph\_ins\_edge(&maze1, &vertices[4], &vertices[6]); // E-G

    /\* Insert edges for Maze #2. \*/

    graph\_ins\_edge(&maze2, &vertices[0], &vertices[2]); // A-C

    graph\_ins\_edge(&maze2, &vertices[0], &vertices[3]); // A-D

    graph\_ins\_edge(&maze2, &vertices[3], &vertices[1]); // D-B

    graph\_ins\_edge(&maze2, &vertices[2], &vertices[5]); // C-F

    graph\_ins\_edge(&maze2, &vertices[4], &vertices[6]); // E-G

    /\* Test the mazes. \*/

    int result1 = isExitReachable(&maze1, 'A', 'G');

    int result2 = isExitReachable(&maze2, 'A', 'G');

    printf("Result for Maze #1 (should be reachable, expect 1): %d\n", result1);

    printf("Result for Maze #2 (should NOT be reachable, expect 0): %d\n", result2);

    /\* Destroy both mazes. \*/

    graph\_destroy(&maze1);

    graph\_destroy(&maze2);

    return 0;

}