Please NOTE: All the header files except maze.h are used as is from the book

maze.h

```
#ifndef MAZE_H_
#define MAZE_H_

#include "graph.h"
#include "list.h"

typedef struct _RoomVertex{
	void* data;
	VertexColor color;
} RoomVertex;

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

#endif
```

maze.c

```
#include <stdio.h>
#include <stdlib.h>
#include "maze.h"
#include "graph.h"
#include "list.h"
#include "queue.h"

/*Assign labels for rooms*/
char roomA = 'A';
char roomB = 'B';
char roomC = 'C';
char roomD = 'D';
char roomE = 'E';
char roomF = 'F';
char roomG = 'G';

/*Pointers to vertices of a graph. These pointer variables reassigned for second maze */
RoomVertex *A,*B,*C,*D,*E,*F,*G;

/*Actual room label matching function*/
int matchRoomLabel(const void* key1, const void* key2){
	char* cKey1 = (char*) key1;
	char* cKey2 = (char*) key2;
```

```
        if(*cKey1 == *cKey2)
                return 1;
        else
                return 0;

}

/*Match the vertex by comparing their room labels*/
int matchVertex(const void* key1, const void* key2){
        RoomVertex* cKey1 = (RoomVertex*) key1;
        RoomVertex* cKey2 = (RoomVertex*) key2;
        return (matchRoomLabel(cKey1->data,cKey2->data));
}

/*Given a label, get the corresponding vertex by searching through the graph*/
int getVertex(Graph *graph, char* label, RoomVertex** vtx){
        ListElmt* element;
        *vtx = NULL;
        char* val;
        /*Return the vertex containing the "label" in vtx*/
  for (element = list_head(&graph->adjlists); element != NULL; element
       = list_next(element)) {
        *vtx = (RoomVertex*)((AdjList *) list_data(element))->vertex;
        val = (*vtx)->data;
    if (matchRoomLabel(label, (*vtx)->data))
       return 1;
  }
  return -1;
}


/*helper function to print the vertices in a maze*/
int printMaze(Graph *graph){
        ListElmt* element;
        char* val;
        RoomVertex* vtx;
        /*Return the vertex containing the "label" in vtx*/
  for (element = list_head(&graph->adjlists); element != NULL; element
       = list_next(element)) {
        vtx = (RoomVertex*)((AdjList *) list_data(element))->vertex;
        val = (vtx)->data;
        printf("%c\n",*val);
  }
  return 0;

}
```

```
int isExitReachable(Graph *pMaze, char entrance, char exit){
        RoomVertex* entranceVertex = malloc(sizeof(RoomVertex));
        RoomVertex* exitVertex  = malloc(sizeof(RoomVertex));
        Queue queue;
   AdjList *adjlist, *clr_adjlist;
   RoomVertex *clr_vertex, *adj_vertex;
   ListElmt *element, *member;

   /*Get the vertex for entrance*/
   if(getVertex(pMaze,&entrance,&entranceVertex) == -1){
                printf("entranceVertex not found\n");

                return -1;
        }
         /*Get the vertex for exit*/
        if(getVertex(pMaze,&exit,&exitVertex) == -1){

                printf("exitVertex not found\n");
                return -1;
        }

   /* Initialize all of the vertices in the graph. */
   for (element = list_head(&graph_adjlists(pMaze)); element != NULL; element
        = list_next(element)) {

      clr_vertex = (RoomVertex*)((AdjList *) list_data(element))->vertex;

      if (pMaze->match(clr_vertex,entranceVertex )) {
         /* Initialize the entrance vertex. */
         clr_vertex->color = gray;
      }
      else {
         /* Initialize vertices other than the entrance vertex. */
         clr_vertex->color = white;
      }
   }

   /* Initialize the queue with the adjacency list of the entrance vertex. */
   queue_init(&queue, NULL);
   if (graph_adjlist(pMaze, entranceVertex, &clr_adjlist) != 0) {
      queue_destroy(&queue);
      return -1;
   }

   if (queue_enqueue(&queue, clr_adjlist) != 0) {

      queue_destroy(&queue);
```

```
   return -1;
}

/* Perform breadth-first search. */
while (queue_size(&queue) > 0) {

   adjlist = (AdjList*) queue_peek(&queue);

   /* Traverse each vertex in the current adjacency list. */
   for (member = list_head(&adjlist->adjacent); member != NULL; member
         = list_next(member)) {

      adj_vertex = (RoomVertex*) list_data(member);

       /*If the adjacent vertex is exit vertex, empty the queue, return 1 i.e. success*/
       if (pMaze->match(adj_vertex,exitVertex )) {

                           queue_destroy(&queue);
                  return 1;
       }
      /* Determine the color of the next adjacent vertex. */
      if (graph_adjlist(pMaze, adj_vertex, &clr_adjlist) != 0) {

         queue_destroy(&queue);
         return -1;
      }

      clr_vertex = (RoomVertex*) clr_adjlist->vertex;

      /* Color each white vertex gray and enqueue its adjacency list. */
      if (clr_vertex->color == white) {
         clr_vertex->color = gray;
         if (queue_enqueue(&queue, clr_adjlist) != 0) {

            queue_destroy(&queue);
            return -1;
         }
      }
   }

   /* Dequeue the current adjacency list and color its vertex black. */
   if (queue_dequeue(&queue, (void **) &adjlist) == 0) {
      ((RoomVertex *) adjlist->vertex)->color = black;
   }
   else {

      queue_destroy(&queue);
```

```
            return -1;
        }
    }
    queue_destroy(&queue);
    /*vertex was not found*/
    return 0;
}

/*Create a Room vertex from the label*/
RoomVertex* createVertex(char* label){
        RoomVertex* vtx = (RoomVertex*) malloc(sizeof(RoomVertex));
        vtx->data = label;
        return vtx;
}

void buildFirstMaze(Graph* graph){
        /*Create all the vertices for maze1*/
        A= createVertex(&roomA);
        B= createVertex(&roomB);
        C= createVertex(&roomC);
        D= createVertex(&roomD);
        E= createVertex(&roomE);
        F= createVertex(&roomF);
        G= createVertex(&roomG);

        /*Insert the vertex into graph*/
        graph_ins_vertex(graph,A);
        graph_ins_vertex(graph,B);
        graph_ins_vertex(graph,C);
        graph_ins_vertex(graph,D);
        graph_ins_vertex(graph,E);
        graph_ins_vertex(graph,F);
        graph_ins_vertex(graph,G);

        /*Insert the edges: 2 per pair because this is undirected*/
        graph_ins_edge(graph,A,D);
        graph_ins_edge(graph,D,A);

        graph_ins_edge(graph,A,C);
        graph_ins_edge(graph,C,A);

        graph_ins_edge(graph,B,D);
        graph_ins_edge(graph,D,B);

        graph_ins_edge(graph,C,F);
        graph_ins_edge(graph,F,C);
```

```
	graph_ins_edge(graph,F,G);
	graph_ins_edge(graph,G,F);

	graph_ins_edge(graph,D,E);
	graph_ins_edge(graph,E,D);

	graph_ins_edge(graph,D,G);
	graph_ins_edge(graph,G,D);

	graph_ins_edge(graph,E,G);
	graph_ins_edge(graph,G,E);
}



void buildSecondMaze(Graph* graph){
	/*Create all the vertices for maze2*/
	A= createVertex(&roomA);
	B= createVertex(&roomB);
	C= createVertex(&roomC);
	D= createVertex(&roomD);
	E= createVertex(&roomE);
	F= createVertex(&roomF);
	G= createVertex(&roomG);

	/*Insert the vertex into graph*/
	graph_ins_vertex(graph,A);
	graph_ins_vertex(graph,B);
	graph_ins_vertex(graph,C);
	graph_ins_vertex(graph,D);
	graph_ins_vertex(graph,E);
	graph_ins_vertex(graph,F);
	graph_ins_vertex(graph,G);

	/*Insert the edges: 2 per pair because this is undirected*/
	graph_ins_edge(graph,A,C);
	graph_ins_edge(graph,C,A);

	graph_ins_edge(graph,A,D);
	graph_ins_edge(graph,D,A);

	graph_ins_edge(graph,B,D);
	graph_ins_edge(graph,D,B);

	graph_ins_edge(graph,C,F);
	graph_ins_edge(graph,F,C);
```

```
	graph_ins_edge(graph,E,G);
	graph_ins_edge(graph,G,E);
}

/*Free the vertices */
void destroyMaze(){
	if(A)
		free(A);
	if(B)
		free(B);
	if(C)
		free(C);
	if(D)
		free(D);
	if(E)
		free(E);
	if(F)
		free(F);
	if(G)
		free(G);

}

/*Wrapper for isExitReachable(). Not used in final submission*/
void testMaze(Graph* maze, char entrance, char exit){
	int retVal = -1;
	retVal = isExitReachable(maze, entrance, exit);
	if(retVal==1){
		printf("maze has a path from %c to %c\n",entrance, exit);
	}
	else if(retVal==0) {
		printf("maze has NO path from %c to %c\n",entrance, exit);
	}
	else{
		printf("Error:could not navigate maze\n");
	}
}


int main(){
	Graph* maze1 = malloc(sizeof(Graph));
	Graph* maze2 = malloc(sizeof(Graph));
	char entrance = roomA;
	char exit = roomG;

	/*Construct maze1 and test for path from entrance to exit*/
	graph_init(maze1, matchVertex,free);
```

```
        buildFirstMaze(maze1);
        if(isExitReachable(maze1, entrance, exit)==1)
                printf("Maze#1 has a path from %c to %c\n",entrance, exit);
        else
                printf("Maze#1 has NO path from %c to %c\n",entrance, exit);
        destroyMaze();

        /*Construct maze2 and test for path from entrance to exit*/
        graph_init(maze2, matchVertex,free);
        buildSecondMaze(maze2);
        if(isExitReachable(maze2, entrance, exit)==1)
                printf("Maze#2 has a path from %c to %c\n",entrance, exit);
        else
                printf("Maze#2 has NO path from %c to %c\n",entrance, exit);
        destroyMaze();

        free(maze1);
        free(maze2);

}
```

OUTPUT

```
Ram (master) Graph_BreadthFirst $ gcc -o hw8 list.c queue.c set.c
graph.c maze.c
Ram (master) Graph_BreadthFirst $ ./hw8
Maze#1 has a path from A to G
Maze#2 has NO path from A to G
```