```
// Copyright (c) 2016 Andrew Sutton
// All rights reserved

#ifndef IMDB_DB_HPP
#define IMDB_DB_HPP

#include "movies.hpp"
#include "actors.hpp"
#include "roles.hpp"

struct database
{
  database();

  int add_movie(const char* name, const char* year);
  int find_movie(const char* name);

  int add_actor(const char* name);
  int find_actor(const char* name);

  int find_actor(const std::string& name);
  int add_role(const char* act, const char* mov, const char* info);


  std::vector<int> get_roles(const std::string& name);
  std::vector<int> get_actors(const std::string& name);


  std::vector<int> BFS(const std::string& start);

  // Storage for movies and actors.
  movie_table movies;
  actor_table actors;
  role_table roles;

  // Efficient lookup for movie and actor names.
  name_index movie_lookup;
  name_index actor_lookup;

  int movie_lookup_errors = 0;
};


#endif
```



```
// Copyright (c) 2016 Andrew Sutton
// All rights reserved

#include "db.hpp"

#include "../imdb/actor_parser.hpp"
#include "../imdb/movie_parser.hpp"

#include <cassert>
#include <iostream>
#include <queue>



database::database() {
  // Pre-allocate a bunch of storage for these things.
  movies.reserve(4 << 20);
  actors.reserve(4 << 20);
  roles.reserve(32 << 20);
  movie_lookup.reserve(4 << 20);
  actor_lookup.reserve(4 << 20);
}

// Adds a movie to the movie table.
int
database::add_movie(const char* name, const char* year) {
  int id = movies.emplace(name, year);
  movie_lookup.emplace(movies[id].name.c_str(), id);
  return id;
}

int
database::find_movie(const char* name) {
  return movie_lookup.find(name);
}

// Adds an actor to the actor table.
int
database::add_actor(const char* name) {
  int id = actors.emplace(name);
  actor_lookup.emplace(actors[id].name.c_str(), id);
  return id;
}

// Returns the row id of an actor with the given name.
int
database::find_actor(const char* name) {
  return actor_lookup.find(name);
}

// Returns the row id of an actor with the given name.
int
database::find_actor(const std::string& name) {
  return find_actor(name.c_str());
}

// Adds a role (an edge) connecting the actor
int
database::add_role(const char* act, const char* mov, const char* info) {
  int a = actor_lookup.find(act);
  assert(a != -1);

  // OH NO! There are inaccuracies in the data set.
  int m = movie_lookup.find(mov);
  if (m == -1) {
    // std::cerr << "error: no movie named '" << mov << "'\n";
    ++movie_lookup_errors;
    return -1;
  }
```

```
  int id = roles.emplace(a, m, info);
  actors[a].add_role(id);
  movies[m].add_role(id);
  return id;
}
struct Vertex {
  int index;
  bool isActor;

  Vertex() = default;

  Vertex(int i, bool a)
  :index(i), isActor(a){}
};



struct movie_visitor
{
  movie_visitor(database& db)
    : db(db)
  { }

  // Nothing to do.
  void on_movie(const char* m) { }

  // Save each movie.
  void on_row(const char* n, const char* y) {
    db.add_movie(n, y);
  }

  database& db;
};


struct actor_visitor
{
  actor_visitor(database& db)
    : db(db)
  { }

  void on_actor(const char* n) {
    db.add_actor(n);
  }

  void on_row(const char* act, const char* mov, const char* info) {
    db.add_role(act, mov, info);
  }

  database& db;
};

std::vector<int>
database::BFS(const std::string& start){
  int target = find_actor(start.c_str());
  std::vector<int> visitedMovies(movies.size(), -1);
  std::vector<int> visitedActors(actors.size(), -1);

  Vertex current;
  std::queue<Vertex> q;
  q.push(Vertex(target, true));
  visitedActors[target] = 0;
  int actorError = 0;
  int actorSuccess = 0;
  int movieError = 0;
  int movieSuccess = 0;

  q.push(Vertex(target, true));
```



```
    while (!q.empty()){
      current = q.front();
      q.pop();
      if (current.isActor){
        for (int i : actors[current.index].roles) {
          int thisMovie = roles[i].movie;
          if (visitedMovies[thisMovie] == -1){
            visitedMovies[thisMovie] = visitedActors[current.index] +1 ;
            q.push(Vertex(thisMovie, false));
          }
        }
      }else{
        for (int j : movies[current.index].roles) {
          int thisActor = roles[j].actor;
          if(visitedActors[thisActor] == -1){
            visitedActors[thisActor] = visitedMovies[current.index] +1;
            q.push(Vertex(thisActor, true));
          }
        }
      }
  }
  return visitedActors;
}


int
main(int argc, char* argv[]) {
  // Emulate a simple shell.
  database db;

  // Actually parse the content.
  movie_visitor movie_vis(db);
  actor_visitor actor_vis(db);

  imdb::movie_parser<movie_visitor> movie_parser("movies.list", movie_vis);
  imdb::actor_parser<actor_visitor> actor_parser("actors.list", actor_vis);
  imdb::actor_parser<actor_visitor> actress_parser("actresses.list", actor_vis);

  std::cout << "* loading movies\n";
  movie_parser.parse();
  std::cout << "* loaded " << db.movies.size() << " movies\n";

  std::cerr << "* loading actors\n";
  actor_parser.parse();
  std::cout << "* loading actresses\n";
  actress_parser.parse();
  std::cout << "* loaded " << db.actors.size() << " actors\n";

  if (db.movie_lookup_errors)
    std::cerr << "!" << db.movie_lookup_errors << " movie lookup errors\n";
  while(true){
    std::cout << "enter target actor" << std::endl << ">";

    std::string target;
    std::getline(std::cin, target);
    if (!std::cin || target == "exit"){
      break;
    }
    if (db.find_actor(target) == -1){
      std::cout << "actor does not exist" << std::endl;
      continue;
    }
    std::vector<int> results = db.BFS(target);

    while (true) {
      std::string actor;
      std::cout << "enter actor" << std::endl << ">";
      std::getline(std::cin, actor);
      if (!std::cin || actor == "back")
```

```
      break;
    int actorNum = db.find_actor(actor);
    if (actorNum == -1){
      std::cout << "actor does not exist" << std::endl;
      continue;
    }
    std::cout << actor << " is " << results[actorNum] /2 << " degrees of seperation from
" << target << std::endl;
   }
  }
}
```