```cpp
// galaxy.h
//
// Declarations for a graph representing Old Republic Spaceways' route
// structure.  Nodes are system planets edges hold the list of ships'
// legs traveling from the origin to destination planet.
//
// Additional fields are defined to allow implementation of Dijkstra's
// algorithm to find the minimum cost (earliest arrival time) between
// pairs of planets.
//
// Copyright 2013, 2018 Systems Deployment, LLC
// Author: Morris Bernstein (morris@systems-deployment.com)

#if !defined(GALAXY_H)
#define GALAXY_H

#include <climits>
#include <iostream>
#include <ostream>
#include <string>
#include <vector>

#include "priority.h"

typedef int Time;
const Time MAX_TIME = INT_MAX;
const Time TURNAROUND_TIME = 4;
const Time TRANSFER_TIME = 6;

typedef int Ship_ID;

class Planet;
class Galaxy;

// Class Fleet maps internal ship ID to the ship's name .
class Fleet {
public:
  Ship_ID add(const std::string& name) {names.push_back(name); return names.size() - 1;}
  const std::string& name(Ship_ID id) const {return names[id];}

private:
  std::vector<std::string> names;
};


// Class Leg represents a single leg of an itinerary, consisting of a
// ship ID, departure time, and arrival time.  Legs are associated
// with an edge between two planets (vertices) in the galaxy map.
//
// A pair of legs may be compared to find the earliest arrival time.
class Leg {
```

```
public:
  Leg(): id(-1), departure_time(MAX_TIME), arrival_time(MAX_TIME) {}
  Leg(Ship_ID id, Time departure_time, Time arrival_time)
    : id(id), departure_time(departure_time), arrival_time(arrival_time) {
  }

  // Return negative, zero, or positive for left leg arriving before,
  // same time, or after the right leg (respectively
  static int compare(const Leg& left, const Leg& right) {
    return left.arrival_time - right.arrival_time;
  }

  static bool less_than(const Leg& left, const Leg& right) {
    return compare(left, right) < 0;
  }

  Ship_ID id;
  Time departure_time;
  Time arrival_time;
};


// Class Itinerary is a sequence of legs with a parallel sequence of
// destinaion planets. i.e. destinations[i] is the destination of
// leg[i].
class Itinerary {
public:
  Itinerary(Planet* origin): origin(origin) {}
  void print(Fleet& fleet, std::ostream& out=std::cout);

  Planet* origin;
  std::vector<Planet*> destinations;
  std::vector<Leg> legs;
};


// Class Edge is a single edge in the route graph.  It consists of a
// destination planet and a sequence of legs departing from the origin
// planet (vertex) to the destination planet.
class Edge {
public:
  Edge(Planet* destination): destination(destination) {}
  void add(Leg& leg) {departures.push_back(leg);}

  // sort(): sort the legs of this edge by arrival time to the
  // destination planet.
  void sort();

  void dump(Galaxy* galaxy);

  Planet* destination;
  std::vector<Leg> departures;
```

```cpp
};


//  Class Planet is a node in the route graph.  It contains a sequence
//  of edges plus additional fields to allow implementation of
//  Dijkstra's shortest-path algorithm.
class Planet {
public:
  Planet(const std::string& name): name(name) {}
  void add(Edge* e) {edges.push_back(e);}

  // reset() clears the fields set by Dijkstra's algorithm so the
  // algorithm may be re-run with a different origin planet.
  void reset() {predecessor = nullptr; best_leg = Leg();}

  // search() computes the shortest path from the Planet to each of the
  // other planets and returns the furthest planet by travel time.
  Planet* search(PriorityQueue<Planet, int (*)(Planet*, Planet*)>& queue);

  // make_itinerary() builds the itinerary with the earliest arrival
  // time from this planet to the given destination planet.
  Itinerary* make_itinerary(Planet* destination);

  // arrival_time() is the time to arrive at this planet from the
  // origin planet that was used to compute the most recent search().
  Time arrival_time() const {return best_leg.arrival_time;}

  // Debug-friendly output.
  void dump(Galaxy* galaxy);

  // Functions for priority queue:
  int get_priority() {return priority;}
  void set_priority(int new_priority) {priority = new_priority;}
  static int compare(Planet* left, Planet* right) {
    return Leg::compare(left->best_leg, right->best_leg);
  }

  const std::string name;

private:
  // relax_neighbors(): for each neighboring planet of this planet,
  // determine if the route to the neighbor via this planet is faster
  // than the previously-recorded travel time to the neighbor.
  void relax_neighbors(PriorityQueue<Planet, int (*)(Planet*, Planet*)>& queue);

  // edges shows the connections between this planet and it's
  // neighbors.  See class Edge.
  std::vector<Edge*> edges;

  // For Dijkstra's algorithm:
  Planet* predecessor;
  Leg best_leg;
```

```
  int priority;
};


// Class galaxy holds the graph of Old Republic Spaceways' route
// structure, consisting of a sequence of planets (vertices).  The
// graph is constructed by adding new planets to the Galaxy object and
// adding edges to the planet objects.
class Galaxy {
public:
  void add(Planet * planet) {planets.push_back(planet);}

  void reset() {for (auto planet: planets) {planet->reset();}}

  // For each planet, apply Dijkstra's algorithm to find the minimum
  // travel time to the other planets.  Print the itinerary to the
  // furthest planet. Terminate with EXIT_FAILURE if the graph is not
  // strongly connnected (you can't get there from here).  Finally,
  // print the diameter of the galaxy and its itinerary.
  void search();

  void dump();
  void dump_routes(Planet* origin, std::ostream& out=std::cerr);

  Fleet fleet;
  std::vector<Planet*> planets;
};

#endif
```