In this assignment, you will implement a binary heap-based priority queue (PQ). The requirements for the assignment are described below.

Step 0: Download the skeleton code and unzip

For this assignment, you are provided with starter code that defines the structures you'll be working with, prototypes the functions you'll be writing and provides data structures upon which to build a PQ implementation. Download the skeleton code for this assignment from Canvas, upload to the COE server and unzip. To unzip the file, use the following command: `unzip assignment4.zip`

Step 1: Implement a heap-based priority queue

Your task for this assignment is to implement a priority queue (PQ). A PQ is a structure that orders data elements based on assigned priority values. Specifically, elements can be inserted into a PQ in any order, but when removing an element from a PQ, the highest-priority element is always removed first.

You must build your PQ using a binary heap. A binary heap is a data structure that takes the form of a binary tree. There are two different kinds of binary heaps:

- A minimizing binary heap is organized so that the element with the *lowest* key is always at the root of the tree.
- A maximizing binary heap is organized so that the element with the *highest* key is always at the root of the tree. In this assignment, you will specifically base your PQ implementation on a minimizing binary heap.

The interface for the PQ you'll implement (i.e. the structures and the prototypes of functions a user of the PQ interacts with) is already defined for you in the file `pq.h`. Your job is to implement definitions for the functions that comprise this interface in `pq.c`.

Note that you may not modify the interface definition with which you are provided. Specifically, do not modify any of the already-defined PQ function prototypes. We will use a set of tests to verify your implementation, and if you change the PQ interface, it will break these tests, thereby (negatively) affecting your grade. Beyond these things, though, feel free to add any additional functions or structures your PQ implementation needs. In particular, you'll have to specify a definition of the main PQ structure, `struct pq`, in `pq.c`.

The PQ functions you'll need to implement are outlined briefly below. All of these functions use the type `struct pq`, which represents the PQ itself and which you'll have to define in `pq.c`. For more details, including information on function parameters and expected return values, see the documentation provided in `pq.c`. Here are the functions you'll have to implement:

- `pq_create()` – This function should allocate, initialize, and return a pointer to a new PQ structure.

- `pq_free()` – This function should free all the memory held within a PQ

structure created by `pq_create()` without any memory leaks. Note that this function only needs to free the memory held by the PQ itself. It does not need to free the individual elements stored in the PQ. This is the responsibility of the calling function.

- `pq_isempty()` – This function should return 1 if the PQ is empty and 0 otherwise.

- `pq_insert()` – This function should insert a new value with specified priority into the PQ. **This operation must have O(log n) runtime complexity.**

- `pq_first()` – This function should return the first element (i.e. the highest-priority value) from a PQ *without* removing it. **This operation must have O(1) runtime complexity.**

- `pq_first_priority()` – This function should return the *priority value* associated with the first element in a PQ *without* removing that element. **This operation must have O(1) runtime complexity.**

- `pq_remove_first()` – This function should remove the first element (i.e. the highest-priority value) from a PQ and return that value. **This operation must have O(log n) runtime complexity.**

Your priority queue must be implemented using a minimizing binary heap as the underlying data structure. This means that within the priority queue you implement, *lower* priority values should correspond to elements with *higher* priority. In other words, the first element in the priority queue should be the one with the *lowest* priority value among all elements in the collection. For example, your priority queue should return an element with priority value 0 before it returns one with priority value 10.

You are provided with a dynamic array implementation in `dynarray.c` and `dynarray.h` that you can use to implement your heap, if you'd like. In addition to this dynamic array implementation, you may implement any additional helper functions you need to make your priority queue work.

Step 2: Testing your work

In addition to the skeleton code provided here, you are also provided with some application code in `test_pq.c` to help verify that your PQ implementation, is behaving the way you want it to. In particular, the testing code calls the functions from `pq.c`, passing them appropriate arguments, and then prints the results. You can use the provided `Makefile` to compile all of the code in the project together, and then you can run the testing code as follows:

```
make
./test_pq
```

Example output of the testing program using a correct PQ implementation is provided

in the `example_output/` directory.

In order to verify that your memory freeing functions work correctly, it will be helpful to run the testing application through `valgrind`.

Step 3: Submission

In order to submit your homework assignment, you must create a **zip file** that contains `assignment4/` folder with your implementation. This zip file will be submitted in Canvas. In order to create the zip file, go to the directory where you can access the `assignment4/`, and use the following command:

```
zip onidusername_program4 assignment4 -r
```

Extra Credit: Dijkstra's Algorithm

One of the most well-known applications of the priority queue data structure is in the implementation of Dijkstra's algorithm, which is used to find the shortest paths in a weighted graph.

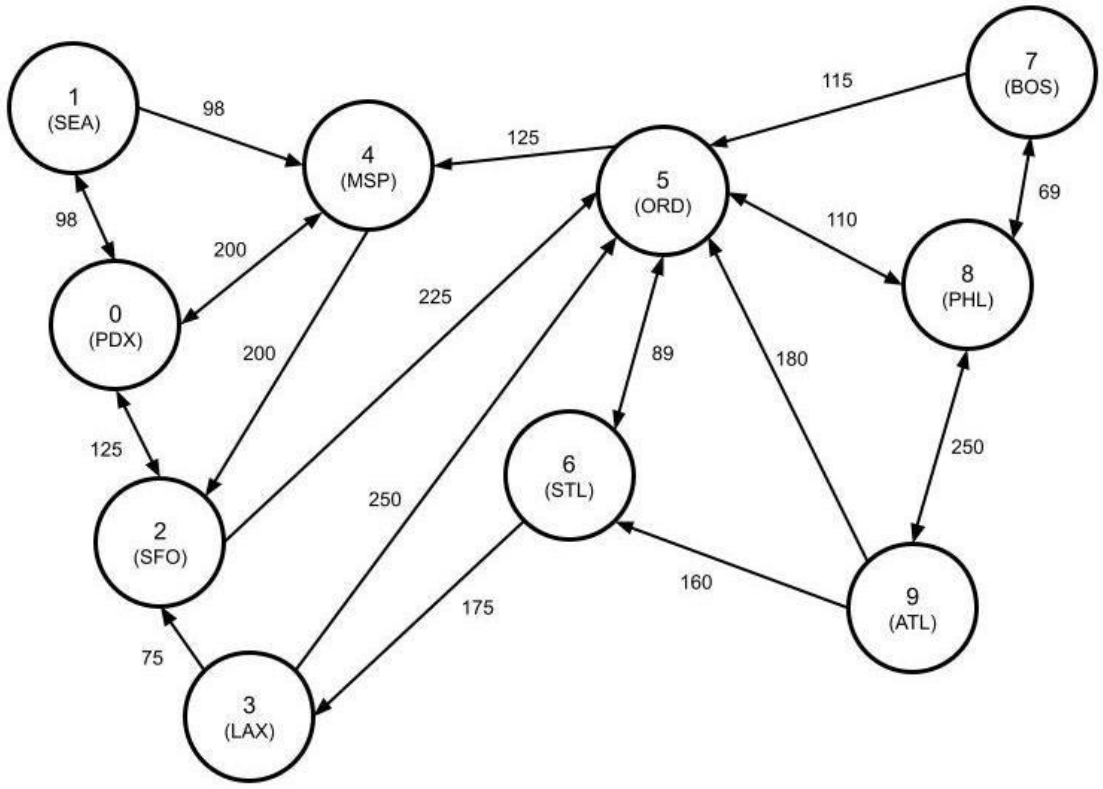
Your job here is to implement Dijkstra's algorithm to find the shortest paths in the graph specified in the file `airports.dat`. Here are some things you'll have to do to make this work:

- Read the data from the file `airports.dat`. You can use the functions `fopen()` and `fscanf()`, respectively, to open and read data from the file. The file has a special format that will make it easier to read:

  ```
  <num_nodes>
  <num_edges>
  <node_i> <node_j> <distance_i_j>
  ...
  ```

  Specifically, the first line of the file contains the number of nodes `n` in the graph, and the second line contains the number of edges specified in the file. The remaining lines each specify one edge in the graph. Each edge specification consists of 3 values:
  - An integer index between 0 and `n-1` representing the source node of the edge (i.e. the edge goes *from* this node).
  - An integer index between 0 and `n-1` representing the destination node of the edge (i.e. the edge goes *to* this node).
  - The cost associated with the edge.
- You can store the graph data in an adjacency list or adjacency matrix. Describe in in the program's comments all data structures used.
- Once the graph is read and stored, implement Dijkstra's algorithm as described in class. We will use this algorithm to find the minimum cost paths from the starting city (node) to all other cities (nodes).
- Allow the user to input the starting node. In other words, you want to find the minimum cost path from Portland (PDX) to all other cities in the graph, the starting node would be 0. Once you've found these paths, print them out, along with their total cost. If there is no path between cities, output "No flights available to city x".
- You will have to figure out how to store the necessary data in your priority queue. You may add functions to the priority queue pq.h and pq.c if necessary.

Below is a depiction of the graph defined in airports.dat:



This graph represents a hypothetical set of flights between airports in the US. In this graph, the nodes represent airports in the US, and each edge weight represents the cost of a flight between two specific airports. Your goal in finding the “minimum costs paths” using Dijkstra's algorithm. Above is a graph representation of the data in airport.dat You can refer to the cities by numbers. We may test your program with a different graph in the file airport.dat so do not “hard code” in the names of the cities.

Include with your submission instructions for compiling and running in a README text file. The README file should also include a description of all data structions used in your program along with the time complexity. Submit your extra credit to Canvas in a separate zip file named `onidusername_extracredit4.`