Index_min_queue: Implementation: Top: The top function's logic is to return the root or min of the Index_min_queue. This is done using the heap_to_index vector in which root is represented as 1. If the size of the vector is empty throw an underflow error for lack of values for index.
IMQ:Implementation: Push: Push logic is to add key at an index from the end. If this index already exists or goes beyond the capacity of the queue, we throw an error. If a key-value pair is added, the overall size of the queue should increase by 1. We then set heap_to_index and its inverse to map the values to the correct index. Then we associate the correct key to the index given in the parameters. The reason we have a push function is to take a key value pair from a vector to a min priority queue.
IMQ:Implementation:Pop: Pop logic exists to delete the top key value pair. The function reduces the size of the queue and percolates down to reorganize the queue again by min priority. When the root is reassigned the previous value of the root becomes null.
IMQ:Implementation:ChangeKey: The change key logic updates an index to a new key value. The key is updated through the key vector. Then the program either percolates up or down based on whether the key is higher or lower than its previous value.
IMQ:Test:all: Our first custom test looks for error checking abilities of the code. The second test places char variable types inside the queue. The third test is an extension of the given test, with more inputs and int data types. The fourth test compares integers from a vector, essentially an int checker, but with extra steps.

Prim_mst.cc:
We started by making the implementation of the graph with three classes: edge, vertex, and graph.
Edge: Just a class with a starting point, end point, and weight. It has a default constructor that sets the weight to max.
Vertex: Class that has two variations of Add edge, one that makes an edge that has the start as the vertex, and one that makes an edge that has the end as the vertex. This allows us to keep the order that an edge is entered from the file. Find edge simply returns the edge for a given end point, this allows us to add edges later in the Prim algorithm. Adj just goes through all the edges, we have to add if statements because we don't know if our begin is the vertex we are at, or if it's the end.
Graph: Creating a graph, we start by allocating space for all of our vertices, then go through each line and add each edge.We then add the edge to the two vertices, start and end. We call two different add edges in order to keep the edge in the same order. Add edge and adj simply call add edge and adj for the corresponding vertex. Then there is the Prim function which simply runs through the pseudo code given to us in the prompt. We just had to add the declaration for the minpq and dist, edge, and marked vectors.

Prim_mst.cc:
This file goes through simple error checking, we used peek in order to check if the first line is an integer. We add our entire part of our code inside a try and catch statement in order to allow us to return exit code 1 if we had an error in the individual points. In our try statement, we make a graph with our input file stream, call Prim on the graph, then go through our vector of edges and print out each one with the correct formatting. We also add up the weights to print out the total weight as the end.