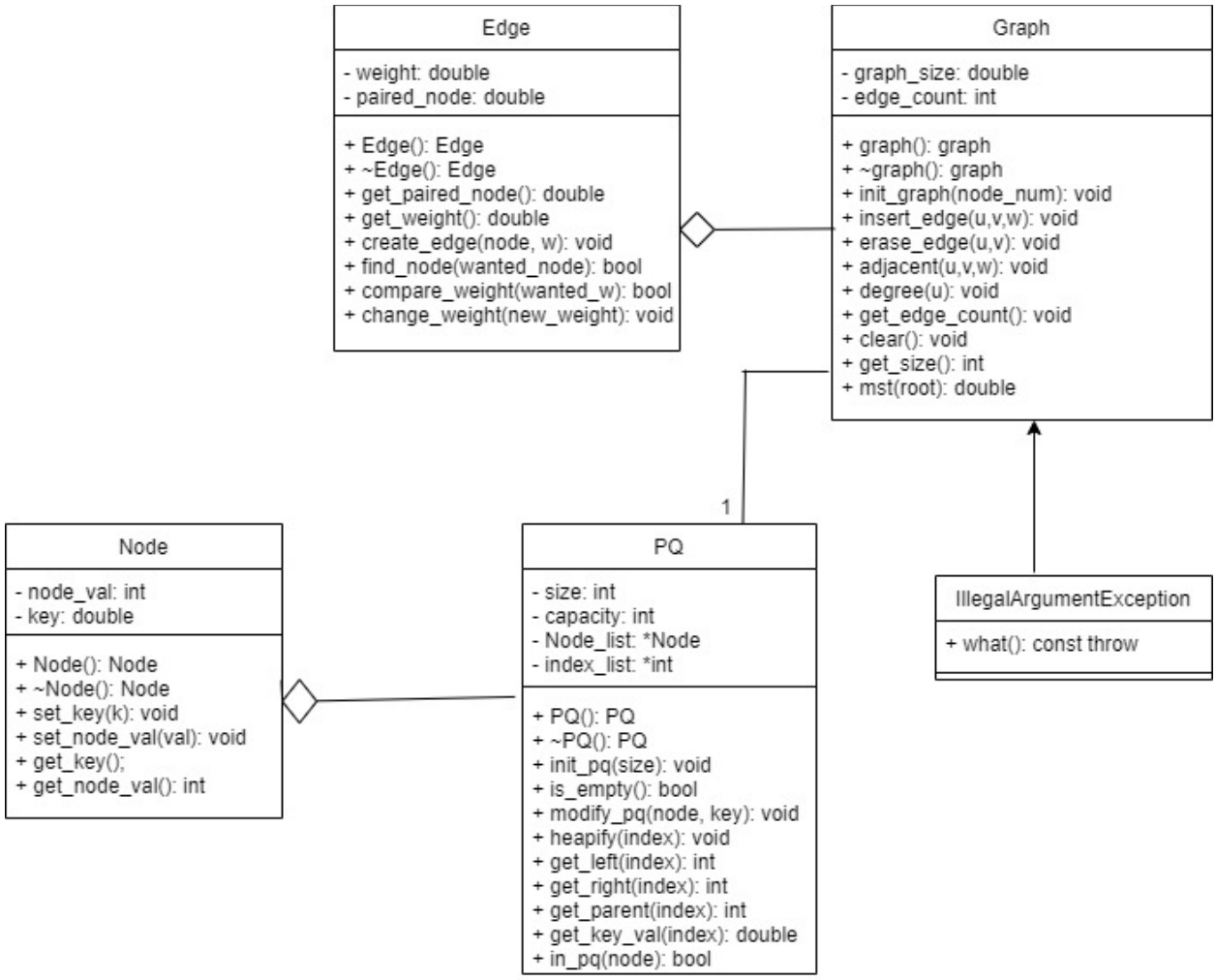
1. **Overview of Classes:**
   There are three classes implemented in my project design: Edge, graph, Node, PQ and illegalArgumentException.
   - The Edge class stores an edge between two verticies, thus it contains a destination vertex and a corresponding weight.
   - The graph class is used to create the graph itself by storing vertices using an adjacency list and invokes all functions associated with the graph
   - The Node class is used within the PQ, the purpose of the node data structure is to store the value of the node and the key.
   - The PQ is a min heap that is indexed based off the node key values and is used to traverse through the graph to calculate the value of mst.
   - illegalArgumentException class is used to throw error statement if operations that are invoked on the graph that contain illegal parameters.
2. **UML Class Design**



3. **Details on Design Decisions**
   Edge Class:
   - Used in graph class to store edges between nodes in a graph.

- Stores the destination node and the weight of the edge. To access this data within the graph classes, setter and getter functions were created for each of the private member variables.

Graph Class:

- Creates graph containing nodes using adjacency list to the adjacent nodes at each node in graph.
- Init graph is used to create the adjacency list to have the correct number of nodes based on size of the graph.
- Insert edge is used to insert an edge between nodes u and v, when inserting we check nodes are in the bounds and if the nodes already have a connecting edge using the adjacency list. If the edge already exists, then we update the weight if they do not then we create a new edge in the adjacency list.
- Erase edge first checks if the nodes are in the bounds of graph and then if edge between u and v exist then we delete it and print corresponding message to console.
- Adjacent function determines if there is an edge between u and v with weight w by searching adjacency list to see if it contains proper edge with proper weight.
- Degree u returns number of adjacent nodes by extracting vector size of adjacency list at index u.
- Clear removes all edges created by invoking vector.clear() and resets edge count to zero.
- Mst calculates the min spanning tree using the prim jarnik algorithm by initializing a min heap to graph size, while min heap is not empty we extract the min value and add its weight to mst count we then loop through all the adjacent nodes and update their keys within the min heap.

Node Class:

- Invoked in the PQ class, stores the node value and the corresponding key value. To access these private member variables outside the class, getter and setter functions were implemented.

PQ Class:

- Replicates a min heap that is used in mst function to extract the smallest node and add it to the mst count.
- We use two pointers; one is a Node pointer to the min heap array and the other is a int pointer that is used to indicate the position of the node within the heap array to ensure swapping and extraction can occur in a linear time.
- The modify function takes in the node and the new key value we want to change it to. We get the position of the node using the index array and update the node key at that position in the node list array. We then invoke heapify on the parent node to ensure the min heap is still balanced.
- The heapify function is a recursive function that takes in an index and determines whether the minimum node is at the top index of the heap. This is done by comparing the passed in node with its children if one of the children are smaller we swap the input with the child and recursively call heapify on the parent.

- The in_pq function is a Boolean that takes in a node and determines if it is still in the priority queue by checking the index array to see where it is positioned in the node array, if the position is greater than the current size of the heap, then the node is no longer in the priority queue.
- Get_parent, get_left and get_right all take in an index and return the corresponding index of the parent, left and right nodes respectively.

4. **Test Cases**
   - Added additional test cases to implement basic functionality and specifications of the project.
   - Added test cases to check if graph is not connected to ensure mst returns the proper statement to the compiler.
   - Added test cases to ensure that illegal argument class works as expected.
   - Added additional test cases to ensure that calling mst on same graph and passing in a different node returns the same mst value.
   - Added test case to ensure that inserting and erasing edges between a single node through and instance of exception class
   -
5. **Performance Considerations**
   - To ensure that the project met the specific run time requirements, the storage of edges in the graph uses an adjacency list this was we can extract the degree of a node in a constant time, O(1) by simply accessing the size of the vector at that specific index. To ensure the edge count can be computed in a constant time, I created a private member variable in the graph class to store the total number of edges in the graph. Thus the run time to extract the total number of edges is O(1). When invoking mst function on the graph, the use of the min priority queue is used to ensure the proper run time is met. In the priority queue was also have the index pointer to indicate where each node is in the priority queue. Therefore when we need to check if a node is still in the queue or if we have to swap elements it can all be done in a constant time O(1). In the Prim Jarniks algorithm, we extract the minimum number from the priority queue until it is empty done in logarithmic time due to heapify being invoked for each node in the graph. When we loop through the adjacent nodes of the current minimum and modify the keys, the run time is invoking heapify thus done in logarithmic time and repeated for each of the adjacent nodes. Therefore the overall run time for mst is O(ElgV) where E is the number of edges and V is the number of nodes in the graph.