The graph data structure is implemented using the Graph class. There is a custom exception class called illegal_exception which inherits from the standard exception class.

Illegal_Exception Class - This class is used to throw exceptions when there is an illegal argument, such as entering a vertex with an illegal value, or assigning an illegal weight to an edge. It has a function what() which returns the string "illegal argument" so it can be returned in the program. Note: As I was unfamiliar with exception classes, I have adopted a variation of the implementation from https://www.learncpp.com/cpp-tutorial/exceptions-classes-andinheritance/.

Graph Class - There are 3 member variables.
`adj_list`: This stores an adjacency list representation of the graph G. It is a dynamic array of size 50000, where each element of the array represents a vertex in the graph. It stores the adjacent vertices and associated weight (— i.e. edge) in a tuple object, where the format is <Vertex A, Vertex B, Weight of Edge A-B>.

`Q`: This stores the priority queue, to be used when finding Minimum Spanning Tree of the graph later on. It is a dynamic array, and each element is an edge.

`costMST:` This integer stores a cached value of the MST. Essentially, if the MST was calculated and then the next command is to find the cost, it will use the pre-determined value. Additionally, it is reset to -1 when changes are made to the graph, indicating that the MST needs to be re-calculated.

The constructor and the destructor of this class initializes the `adj_list` member variable and deallocates the memory respectively. Hence, the runtime of this function is O(1).

The function `load_graph` and `insert_edge` share the same algorithm, with the exception of additional debugging statements. Given three integers that represent a Vertex A, Vertex B and weight W, they are added to the Graph G or the adjacency list.

First, confirm this edge doesn't already exist in the graph.

- The code first checks if either Vertex A or Vertex B has no adjacent vertices, which takes constant time. If either has no adjacent vertices, then the code returns immediately, and the runtime is O(1).
- Then, the code iterates over the adjacent vertices of the vertex with fewer adjacent vertices and checks if the other vertex is adjacent to it. So, it has a runtime of θ(min(adj_A, adj_B)) where adj_A and adj_B represents the number of adjacent vertices to Vertex A and Vertex B respectively.
- In the worst case, adj_A = adj_B = |E| and a runtime of O(|E|).

If we confirm this edge doesn't already exist, we add the edge as a tuple object to the adjacency list. Since the edge is undirected, both vertexes in the adjacency list need to keep track of it. This is constant time, O(1).

The function `print_adj_edges` takes in an integer that represents a vertex in Graph G. There is a constant lookup time for the vertex in the adjacency list matrix. Since we iterate through each adjacent vertex , this has a runtime of O(adj_A) i.e. the number of adjacent vertices to a given Vertex A.

The function `delete_node` takes in an integer that represents a vertex to delete in Graph G. There is a constant lookup time for the vertex in the adjacency list matrix. We iterate through each adjacent vertex and delete it from the list. Since the graph is undirected, we need to delete the edge from the adjacent node as well. If adj_A and adj_B represents the number of adjacent vertices to Vertex A and Vertex B respectively, the runtime is
θ(adj_A* adj_B). In the worst case, the size of both lists is |E| and the runtime is $O(|E|^2)$.

The function `find_mst` and `cost_mst` share the same algorithm, with the sole difference being the output statements.

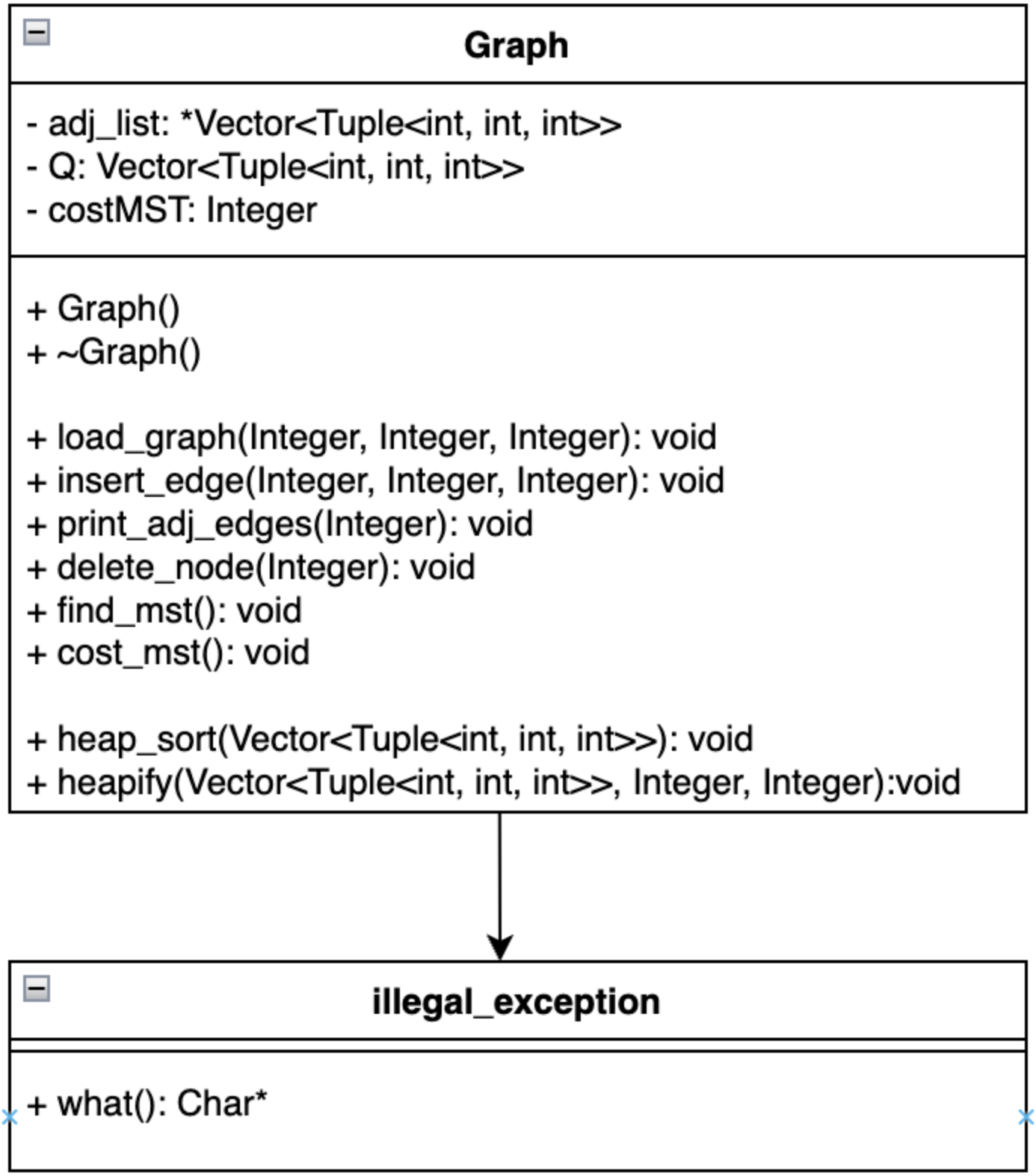
It is based on Prim's algorithm to generate the MST and it's cost. First, there are a few initialization procedures

- Initialize priority queue Q, cost of MST and two arrays, mst (i.e. adds the vertices covered in the MST) and mst_adj_list (i.e. a copy of the existing adjacent list)
- We iterate through all vertices to find a count of how many there are in the graph. This is stored in an integer variable n. Also, we find a source node from where we can build the MST.

While all vertices have not been found,

- from the current vertex, find all adjacent vertices.
- Add them to a priority queue, and run the heapsort algorithm. The pseudocode, adopted from CLRS, tells us that this runs in O(log V). It sorts the edges by weight.
- We remove the edge from the adjacency matrix so it's not considered in future iterations. This has a worst case runtime of O(V).
  So, the runtime of this algorithm is:
  V *O(log(V)) = O(V* log(V))
  and since the |V| < |E|^2 for the worst cases, it can be simplified to O(V log(E)).

UML Diagram:


Graph
- adj_list: *Vector<Tuple<int, int, int>>
- Q: Vector<Tuple<int, int, int>>
- costMST: Integer
+ Graph()
+ ~Graph()
+ load_graph(Integer, Integer, Integer): void
+ insert_edge(Integer, Integer, Integer): void
+ print_adj_edges(Integer): void
+ delete_node(Integer): void
+ find_mst(): void
+ cost_mst(): void
+ heap_sort(Vector<Tuple<int, int, int>>): void
+ heapify(Vector<Tuple<int, int, int>>, Integer, Integer):void
illegal_exception
+ what(): Char*