To implement the Graph I created a Graph class to handle all the required functionality as given by the requiremnets. High level functions to use the Graph were made public, while member variables were made private to create this ADT. Additionally, helper classes were made to implement this structure. In the main file there exists a loop which awaits user inputs. This loop is not apart of the Graph as there are situations where you could use one without such a loop.

Class Graph:
> Contains all the functionality to manipulate and use the graph, as well as necessary data members. High level functions are public, while members are private to ensure proper use, The implementation comes in the form of an adjacency list, or an array of length 50000 containing linked lists storing all the adjacent nodes to each vertex. There also exist a boolean array storing weather or not a node exists to account for the case in which all the edges to a node get deleted. To ensure proper use, members are only accessed through the usage of methods. Upon creation, the values of the adjacency list and vertex flag array are set to nullptr and false, respectively. In order to prevent memory leaks, upon destruction, the adjacency list is totally deleted.

Class MinHeap:
> Contains all the functionality of a minimum heap to allow for log(n) insertion and deletion. High level function are public, while low level functions and members are made private to ensure proper use. The implementation comes in the form of a "one indexed" (to make position calulations simpler) vector, storing all the values of the heap in a binary tree format. The heap is ordered with respect to the weights of the edges. The heapify function is made private as it should only be used in the context of a deletion. Upon creation, a vector with an initial entry of nullptr is initialized to allow for "one indexing". In order to prevent memory leaks each edge of the vector is delete upon destruction.

Class Node:
> Stores the vertex being pointed to, the weight of the edge, a pointer to the next edge, as well as an edge object to allow for seamless use. To ensure proper use, relevant public getter and setter methods are used, while data members kept private. Upon creation, the next node and edge are initialized to the input values given. In order to prevent memory leaks, the edge is deleted upon destruction. This node class is particularly useful in the constuction of the adjacency list

Class Edge:
> Stores the origin, weight and destination node of the edge. This edge class is particularly useful in the construction of the min heap used in the MST and cost functions of Graph. In order to ensure proper use, relevant getter and setter methods were implemented while making all members private.

Class illegal_exception: public std::illegal_exception:
> Contians all the members of std::exception. Effectively acts as a name change in its current form, but could act as an extention to the standard libraries exception handling if modified and added on to. When an illegal exception occurs, an error will be thrown, and handled by a try/catch. This functionality is used in the Graph class and main loop.

Runtimes:

Insert: O(degree(a))
> Inorder to insert a node, we first must check to see if such a connection has already been made. As a result we must traverse all the elements in the linked list of a. the insertion portion is O(1) as we insert at the head. This gives O(len(adjacency[a-1])) or O(degree(A))

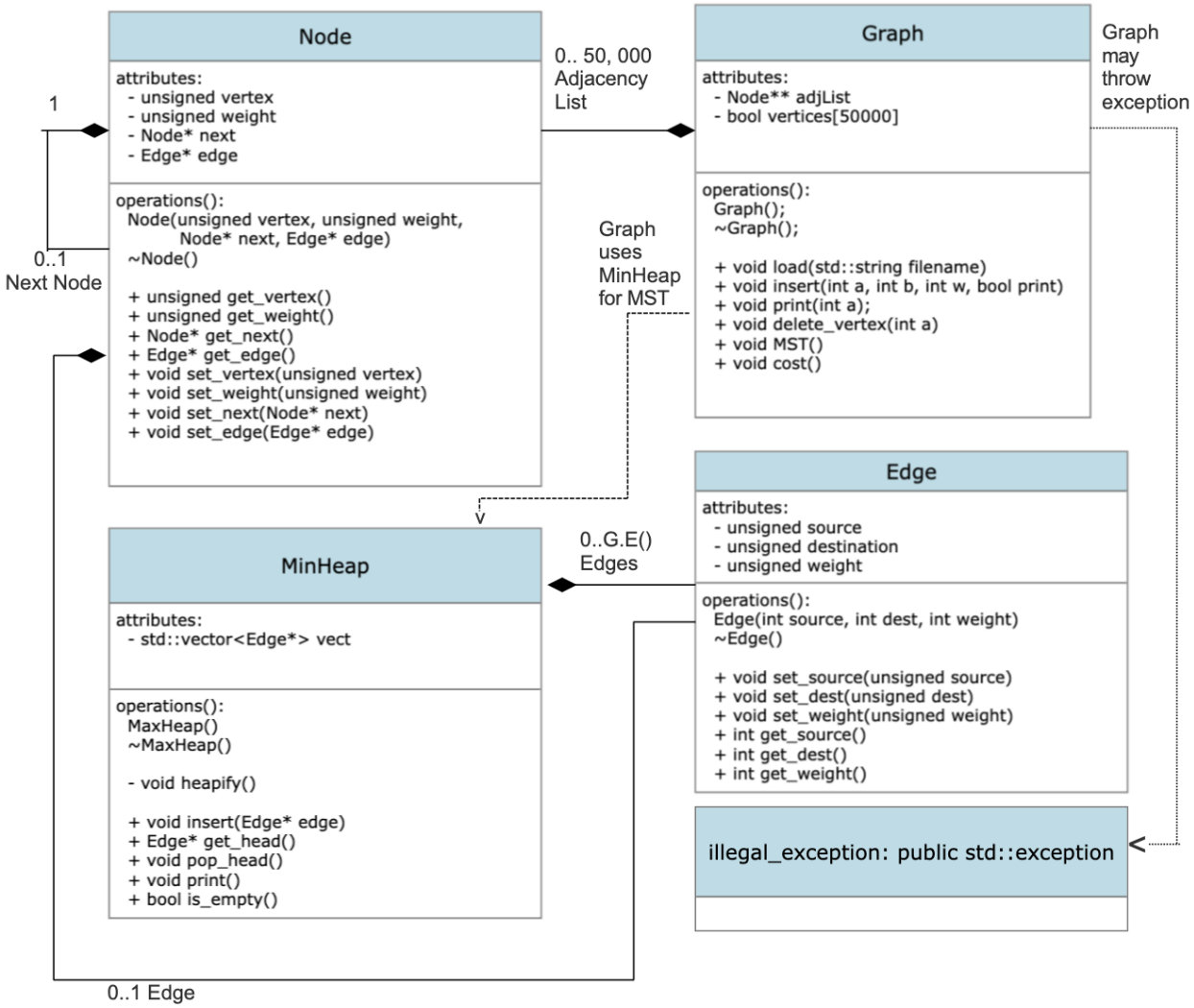
Print: O(degree(a))
> To print each adjacent vertices, we must traverse a's adjacency list. Printing is O(1). Therefore we get O(degree(a))

Delete: O(E)
> When deleting a node, we also must delete all edges to adjacent nodes. As all connections are undirected we must also delete all the references to node a in all the nodes contained in the edge list of a. This gives a worst case of O(E) as if a is connected to every other vertex, we might have to traverse every edges or such vertex to delete the reference to a, and as a result, every edge in the graph. Therefore O(E).

MST: O(E * log(V))
> To implement the MST I used the Prim Jarnik algorithm with the use of a minimum heap as the priority queue. In order to construct the MST, we must create a priority queue of all the edges in the first vertices adjacency list (O(E) as the vertex could contain all edges). Then we begin by dequeuing the heap until it is empty (O(E)). as we dequeue the heap we must then check all the vertices of the edges destination, and if such edge leads to a destination that has not been visited, we must then load all the edges of that vertex to the queue O(E). Then we must insert the edges into the queue (taking O(log(V)) as insertion into a binary heap takes O(log(E)) and E is at most V^2 -> log(v^2) = 2log(E) therefore O(log(V))). While there are two while loops which have an upper bound of O(E), such loops depend on eachother, in which the total amount of edges loaded into the heap will be E, giving O(E) for both loops combine. Finally, the printing of the edge takes O(1). As the insertion within the loops takes log(V), we henceforth get O(E * Log(V)) time complexity.

Cost: O(E * log(V))
> Same procedure as MST, but instead of printing which takes O(1), we add the weight of the current edge to the current cost (starting from 0), also O(1). Therefore O(E * log(V))

UML Diagram


Node
attributes:
- unsigned vertex
- unsigned weight
- Node* next
- Edge* edge
operations():
Node(unsigned vertex, unsigned weight, Node* next, Edge* edge)
~Node()
+ unsigned get_vertex()
+ unsigned get_weight()
+ Node* get_next()
+ Edge* get_edge()
+ void set_vertex(unsigned vertex)
+ void set_weight(unsigned weight)
+ void set_next(Node* next)
+ void set_edge(Edge* edge)
1
0..1
Next Node
0.. 50, 000
Adjacency
List
Graph
attributes:
- Node** adjList
- bool vertices[50000]
operations():
Graph();
~Graph();
+ void load(std::string filename)
+ void insert(int a, int b, int w, bool print)
+ void print(int a);
+ void delete_vertex(int a)
+ void MST()
+ void cost()
Graph
may
throw
exception
Graph
uses
MinHeap
for MST
Edge
attributes:
- unsigned source
- unsigned destination
- unsigned weight
operations():
Edge(int source, int dest, int weight)
~Edge()
+ void set_source(unsigned source)
+ void set_dest(unsigned dest)
+ void set_weight(unsigned weight)
+ int get_source()
+ int get_dest()
+ int get_weight()
MinHeap
attributes:
- std::vector<Edge*> vect
operations():
MaxHeap()
~MaxHeap()
- void heapify()
+ void insert(Edge* edge)
+ Edge* get_head()
+ void pop_head()
+ void print()
+ bool is_empty()
0..G.E()
Edges
illegal_exception: public std::exception
0..1 Edge