1. Overview of Classes

a. Vertex class - used as a node inside the graph and contains its edges. Has a friend class Graph.

- id - is the “researcher ID” of each node and is distinct.It is initialized to `inid` a parameter value
- key - used to find the MST and is initialized to `-1` to guarantee its smaller than any other weight
- visited - used to determine if we have visited the node before during MST. If `true` it is in the heap
- parent - points to the node that points to it. It is initialized to `nullptr`
- edges - a vector of edge pointer that keeps track of all the edges coming out of the node
- **getKey** - returns key

b. Edge class - used to connect two vertices in a certain direction. Has a friend class Graph.

- weight - holds the weight of the edge
- nextVertex - points to the other vertex (end node)

c. Graph class

- map - array of 23133 vertex pointers and represents the graph. Each entry initialized with `nullptr`
- numOfVertex - the number of vertices in the graph (map)
- **insert** - passes in a start, end node, and weight. Attempts to create a start and end node if they do not exist. Checks if an edge already exists between the two nodes, if so then return false. If not then dynamically allocate a new node and return true. Push back the edge to the start node's edge vector
- **print** - passes in the “researcher id” and checks if the node exists, if so then prints out all adjacent nodes connected through an edge by iterating through its edge vector. If it doesn’t exist print newline
- **remove** - passes in the “researcher id” and checks if the node exists, if so, iterate through all nodes and delete all the edges that point to the node. Then delete the node by deleting all of its edges and then deleting it and returning true. If the node does not exist, return false.
- **graphSize** - return numOfVertex
- **mst** - passes in the “researcher id” and uses a max heap to find the MST. It first checks if the node exists, if it does not exist return 0. Assign the root node a key of 0 and all other nodes a key of -1. Set the root's parent pointer to itself and all other parent pointers to `nullptr`. Then put all the nodes in the heap and use Prim’s to find the MST. Remove the max key from the heap and check if its parent is `nullptr`, if it is, we can stop here. If it isn't, we add it to a vector and check if the adjacent nodes are also in the heap and if its key is smaller than the weight of the edge. If it is then we can update the key to the new weight and set the parent to the max key’s node. Compare the key with its parent and swap if the compared node has a larger key and repeat till its parent’s key is larger. Repeat the above process till the heap is empty and return the size of the vector.

d. Heap class

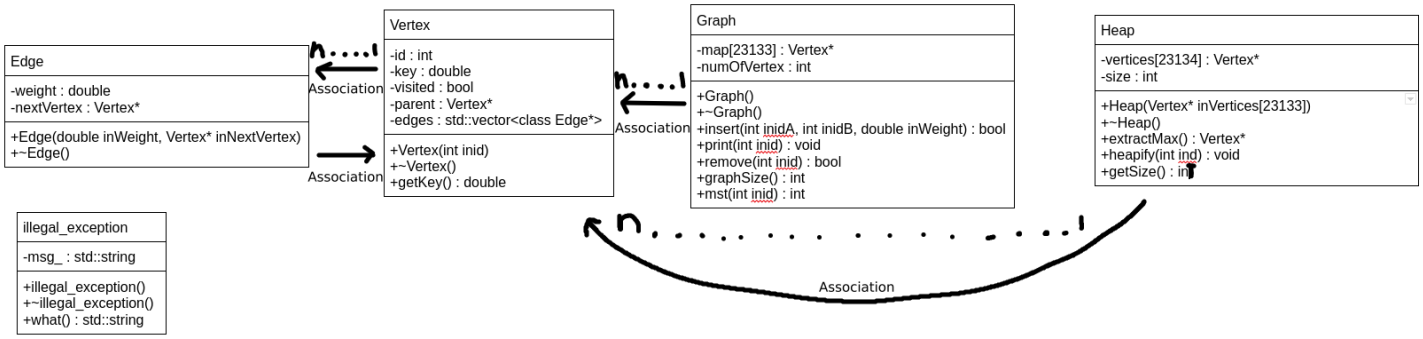
- vertices - an array of all the pointers in map (from Graph class)
- size - # of elements inside the heap
- **extractMax** - call heapify at index 1 and return the pointer to the vertex with the maximum key
- **heapify** - passes through an index and compares the keys of the left and right nodes with the max key. If the max key is not the current node, swap it and call heapify again at the max key’s index

- **getSize** - returns size

e. Illegal_exception class - Serves as an exception class and is used whenever we have an illegal input. Has data member msg_ which is a string and is the output msg.

- **Public member function**:
  - **what()**: returns msg_
- **Constructor and Destructor**:
  - **illegal_exception()**: sets msg_ to “illegal argument”
  - **~illegal_exception()**: does nothing as when msg_ leaves scope the string destructor is called

2. UML Class Diagram



3. Details on Class Decisions (constructors and destructors)

- No use of const functions (could have in getters but no point with our application) and no use of overloading. Edge and Vertex class are friends with graph class because graph uses the private data members of Edge and Vertex so much it makes no sense to use getters and setters.

a. Vertex class

- Vertex(int inid) - sets id to `inid` and all pointers to `nullptr` and all non-pointers to `0`
- ~Vertex() - iterates through the edge vector and deletes each edge. Sets all pointers to `nullptr`

b. Edge class

- Edge(double inWeight, Vertex* inNextVertex) - sets weight to `inWeight` and nextVertex to `inNextVertex`
- ~Edge() - sets nextVertex to `nullptr`

c. Graph class

- Graph() - sets all pointers in map to `nullptr` and numOfVertex to 0
- ~Graph() - Deletes each pointer in map and sets it to `nullptr`

d. Heap class

- Heap(Vertex* inVertices[23133]) - copy each node in map into vertices and call heapify for each index from the middle to 1 to build the heap.
- ~Heap() - sets all pointers in vertices to `nullptr`

e. Illegal_exception class

- illegal_exception() - sets _msg to "illegal argument"

- ~illegal_exception() - nothing

4. Test Cases

- General Case
  - print a vertex after deleting it (to see if delete worked)
  - insert/print/delete invalid weights/researcher id to get illegal_exceptions
  - print a vertex that does not exist and print one with adjacent vertices
  - check the size of the graph
- Edge case
  - Insert an edge from an existing vertex to a new vertex and check if size has increased by 1
  - Insert an edge when both vertices exist and check only an edge was created
  - Insert an edge when an edge already exists and checks the weight has not changed
  - Delete a vertex twice to see if we have a memory leak and if we have completely delete it
  - Print the vertex after calling MST on the vertex

5. Performance Decisions

- **i** calls insert(int inidA, int inidB, double inWeight) - Checks if vertex A and vertex B already exists (if not, create them) and then iterates through each edge vector of vertex A and checks if an edge already exists between them. Thus the running time is O(degree(A)) but note that there exists a scenario where vertex A has all the edges in the graph point out of it. In this case, degree(A) = E. Thus, the running time is **O(|E|)**
- **print** calls print(int inid) - Iterates through the edge vector at vertex A and prints each vertex (degree(a) times). Thus the runtime is **O(degree(A))**
- **d** calls remove(int inid) - Iterates through the edge vector and deletes every edge that involves the Vertex. Thus we will in the worst case iterate through V*E elements hence a runtime of **O(|V||E|)**
- **mst** calls mst(int inid) - First it initializes every vertex (sets key to -1, parent to `nullptr`, visited to `false`). This is O(|V|) as we iterate through every vertex. Then we build the heap which is O(|V|) run time. We extract the max key till the heap is empty which will take O(|V|lg|V|) because it is essentially heap sort. We check edges (E times) for adjacent nodes and “bubbling up” at the edge node (lg|V| time) which will take O(|E|)*O(lg|V|) = O(|E|lg|V|) time. Note that |V| = O(sqrt(|E|)) and O(V) = O(E). Thus the sum is O(|V|) + O(lg|V|) + O(|V|lg|V|) + O(|E|lg|V|) = **O(|E|lg|V|)** as the final answer
- **size** calls graphSize() - returns the int numOfVertex which is **O(1)**