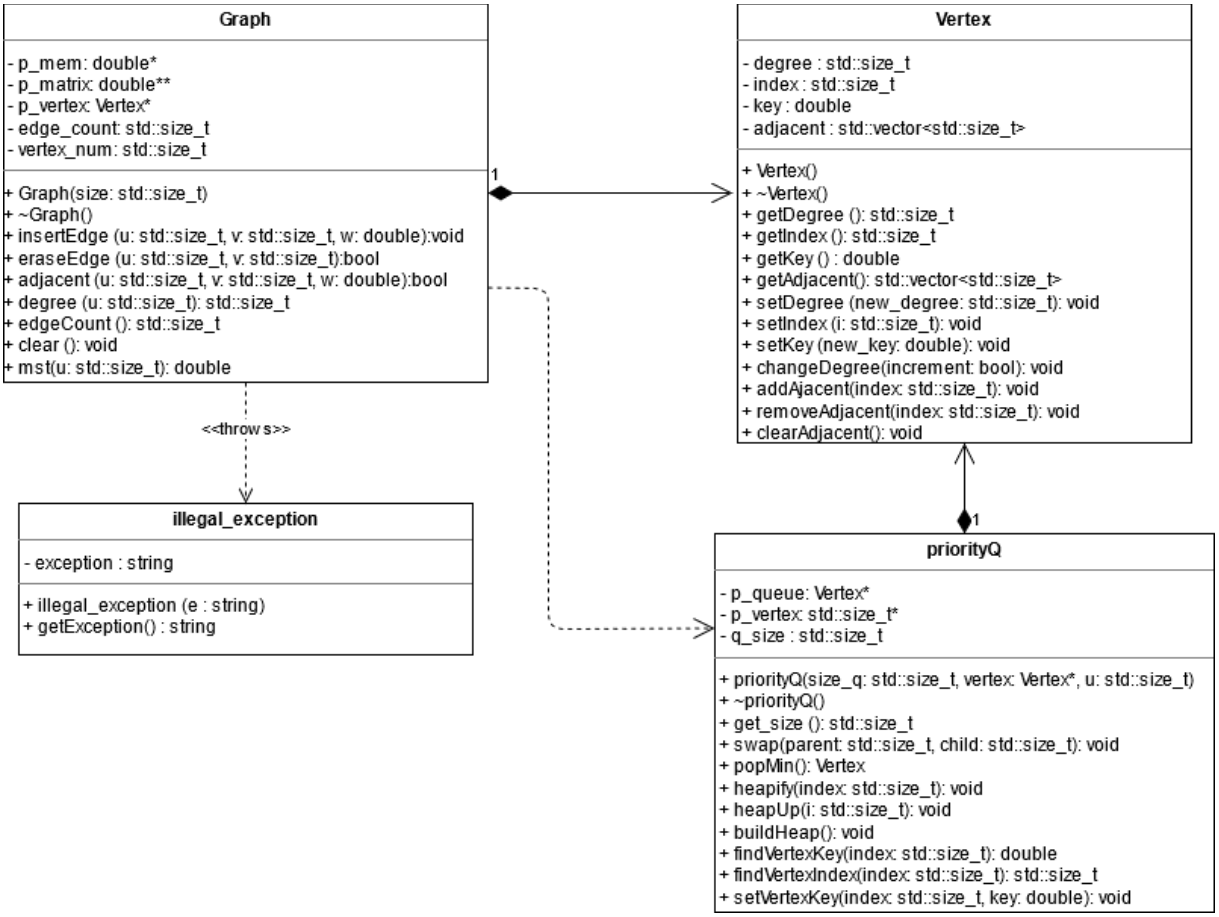
Design Document Project 3

Overview of Classes

Four classes were designed, a Graph class, a Vertex class, a priorityQ class and an illegal_exception class. The Graph class was designed to create and manage (insert/erase/edge_count/degree/adjacent/mst/clear etc.) an undirected Graph of a specified size. The Vertex class was designed to contain information about individual vertices contained in the Graph such as the degree of the vertices, key value, and the list of vertices which are adjacent. The purpose of the priorityQ class is to help in the implementation of the Prim-Jarnik algorithm used to calculate the MST value of the Graph. The illegal_exception class handles the illegal exceptions caused by invalid inputs to the insert, erase, adjacent, degree, and mst commands.

The Graph class uses the Vertex objects to represent nodes in the Graph and store information about each vertex in the Graph. The Graph class can manipulate the data regarding the Graph by either updating its own member variables (edge_count, weight matrix) or using the public member functions of the Vertex class. The Graph class uses the priorityQ class to create a priority queue object which is used in the calculation of MST. Furthermore, the priorityQ class uses Vertex objects as members of its priority queue. The illegal_exception class is used by the Graph class to throw exceptions when the input to its member functions is invalid.

UML Class Diagram



Design Decisions

Graph

The constructor used for this class dynamically allocates 3 arrays, 2 of which combine to make up the adjacency matrix (2D array n*n; n = size of graph) and the other which stores the Vertex objects contained in the Graph. It also initializes every value in the matrix to 0 since there are no edges in the graph initially and assigns each vertex an index number so it can be identified later. The *edge_count* is set to 0 and the *vertex_num* is initialized to the size of Graph. The destructor deallocates the memory for the three arrays which was allocated in the constructor. No operators were overwritten. Passing by reference was not used.

For the following member functions:

```
void insertEdge (const std::size_t u, const std::size_t v, const double w);
bool eraseEdge (const std::size_t u, const std::size_t v);
void adjacent (const std::size_t u, const std::size_t v, const double w) const;
std::size_t degree (std::size_t u) const; double mst (const std::size_t u) const;
```

The keyword *const* was used at the parameter level since in each case the parameter being passed should not be altered within the function.

For the following member functions:

```
void adjacent (const std::size_t u, const std::size_t v, const double w) const;
std::size_t degree (std::size_t u) const; std::size_t edgeCount() const;
double mst (const std::size_t u) const;
```

The keyword *const* was used at the function level since in each case the function should not change the value of any member variable.

Note: The keyword const was not used at either the function level or parameter level for the `void clear ();` function as it has no parameters and must be able to alter private member variables of the Graph.

Vertex

The constructor for the Vertex initializes the private member variables - it sets *degree* = 0 and the *key* = INFINITY (*index* is initialized in the Graph and *adjacent* does not need to be initialized). The destructor clears the adjacency list. No memory needs to be deallocated since none was allocated in the constructor. No operators were overwritten. Passing by reference was not used.

For the following member functions:

```
void setDegree (const std::size_t new_degree); void setKey (const double new_key);
void setIndex (const std::size_t i); void changeDegree (const bool increment);
void addAdjacent (const std::size_t index); removeAdjacent (const std::size_t index);
```

The keyword *const* was used at the parameter level since in each case the parameter being passed should not be altered within the function.

For the following member functions:

```
std::size_t getDegree() const; double getKey() const; std::size_t getIndex() const;
std::vector<std::size_t> getAdjacent() const;
```

The keyword *const* was used at the function level since in each case the function should not change the value of any member variable.

Note: the `void clearAdjacent ();` function is the only function in which the keyword const was not used at either the parameter or function level since it needs to be able to change the values of the member variables and it has no parameter.

priorityQ

The constructor for the priorityQ initializes the *size_q* to the size of the graph. It also dynamically allocates two arrays of the same size as the graph. One is for the priority queue itself (*p_queue*) and the other is to keep track of the location of each vertex in the queue (*p_vertex*). The vertices from the Graph are copied into the priority queue and their position is recorded in the auxiliary array. The min heap is built by calling heapify up the tree starting at the lowest level non leaf nodes. The destructor deletes the arrays which were allocated in the constructor. No operators were overwritten. Passing by reference was not used.

For the following member functions:

```
void swap (const std::size_t parent, const std::size_t child); void heapify (const std::size_t index);
void heapUp (const std::size_t i); double findVertexKey (const std::size_t index) const;
std::size_t findVertexIndex (const std::size_t index) const;
void setVertexKey(const std::size_t index, const double key);
```

The keyword *const* was used at the parameter level since in each case the parameter being passed should not be altered within the function.

For the following member functions:

```
std::size_t get_size() const; double findVertexKey (const std::size_t index) const;
std::size_t findVertexIndex (const std::size_t index) const;
```

The keyword *const* was used at the function level since in each case the function should not change the value of any member variable.

Note: the `void buildHeap (); Vertex popMin()` functions are the only functions in which the keyword const was not used at either the parameter or function level since they needs to be able to change the values of the member variables and they both have no parameters.

Illegal_exception

The constructor for the *illegal_exception* class is passed a string when the exception is thrown and assigns it to the *exception* member variable. No destructor was required since there was no dynamic memory allocation in the constructor. No operators were overwritten. Passing by reference was not required.

The keyword const is used in the `string getException()` function as no member variables should be updated in this function. The keyword const is not used at the parameter level as `string getException()` is the only member function of this class.

Test Cases

To test if my program met the specified requirements, I created specific inputs to try and cover the many possible cases which would reveal problems with my code (stack dump, memory leak, dangling pointer, etc.)

Some specific cases which I tested for:

- Adding edges with negative weights
- Adding edges between vertices which are not contained in the Graph
- Adding an edge between the same vertices
- Checking the adjacency of two vertices which do not share an edge
- MST of the same graph using a different starting node each time (mst should be same)
- MST of an unconnected graph
- MST of a vertices not in the Graph
- Clearing the function midway through the test case
- Checked to make sure the two entries were being updated in the adjacency matrix each time an edge was inserted/erased
- Tested using Valgrind for memory leaks

Performance Considerations

Expected run time of O(ElgV)

The *mst* operation has an expected running time of **O(ElgV)** where E is the number of edges and V is the number of vertices. The *mst* operation involves implementing the Prim-Jarnik algorithm. This involves initializing values for all the vertices and creating the priority queue by copying the vertices into the queue (takes **O(V)** time). Then the min heap must be built (takes **O(lgV)**). After the priority queue is created, the program will remove the minimum vertex from the queue until there is the queue is empty (takes **O(VlgV)** – lgV to fix the priority queue and remake the min heap, must be done for each vertex). As the vertices are removed from the queue each Edge in the graph is examined once and in the worst case, for each Edge a key value would have to updated in the priorityQ. This would mean that for each edge, the priorityQ would need to be fixed so it is once again a min heap which would have a cost of **O(ElgV)** – lgV is the cost of fixing the priorityQ and remake the min heap, must be done for all edges. Hence the total run time would be **O(V + lgV + VlgV + ElgV).** O(V), O(lgV) and O(VlgV) can be ignored and it can be simplified to **O(ElgV)** since the number of edged in a graph is much larger than the number of vertices. Hence the asymptotic upper bound of the mst operation is **O(ElgV)**.

Expected run time of O(1)

Operations *degree* and *edge_count* have an expected running time of O(1) since they only require accessing of a member variable. The *edge_count* operation can be computed in a constant time since it is stored as a member variable in the Graph and updated (incremented/decremented) whenever edges are inserted or deleted from the Graph. The *degree* operation checks the member variable (which stores the degree of the Vertex) of a specific Vertex *u* in the Graph and returns how many edges have been inserted between *u* and other Vertices. This design choice allows *degree* and *edge_count* operations have an asymptotic tight bound of O(1).