CGL Data Structures Specification Sheet

<u>Node Class:</u>
Contains an ID to identify itself. Equality comparisons would check if the IDs are matching. A node precedes another if its ID is smaller than the other's.

```
class Node {
     public:
          int64_t id;
          Node(int64_t id);
          bool operator==(const Node& other);
};
```

<u>Edge Class:</u>
Contains two nodes in a vector in this format {upstream node id, downstream node id}. Also keep track if it's a directed or undirected edge. If it's directed, the relationship would be upstream node->downstream node. If it's undirected, the relationship would be upstream node<->downstream node.

```
class Edge {
     private:
          vector<int64_t> nodes;
          bool isDirected;
          bool isBackwards;
     public:
          Edge(int64_t id1, int64_t id2, bool isDirected,
               bool isBackwards = false);
          Edge reverse();
               // Returns an identical Edge object with
                  isBackwards = !isBackwards.
          int64_t getUpstreamId();
               // Both up/downstream depends if the edge is
                  backwards
          int64_t getDownstreamId();
          int64_t traverse(int64_t id);
               // Returns the downstreamId if upstreamId is
given
          bool operator==(const Edge& other);
               // Checks if both edges have the same upstream
                  node
}
```

NodeTraversal Class:
Node wrapper class that traverses the given node. Maintains a backwards value that could reverse edges. Returns any edge that would be traversed from node as the upstream node. Think of it as combining the node and edge into one object.

```
class NodeTraversal {
     private:
          Node* node;
          bool isBackwards;
     public:
          NodeTraversal(Node* node, bool isBackwards = false);
          int64_t getId();
               // Returns if of the current node.
          bool reverse();
               // Returns the current isBackwards state
          vector<Edge*> getTraversedEdges(vector<Edge*> edges);
               // Returns all vertices that have the same
                  upstream node id. These edges will be reversed
                  if isBackwards is true.
}
```

Graph Class:
This is a directed graph that contains a vector of NodeTraversals and Edges.

```
class Graph {
     private:
          vector<NodeTraversal*> vertices;
               // Collection of vertices in this graph.
          vector<Edge*> edges;
               // Collection of all edges in this graph.
          inline Node newNode(int64_t id);
               // Creates a new Node object.
          inline void addEdge(int64_t upstream_id,
               int64_t downstream_id, bool isDirected);
               // General purpose addEdge function that creates
                  a directed/undirected edge with the given ids.
     public:
          void addVertex(int64_t id);
               // Creates a new NodeTraversal object (needs to
                  make a new Node object with the corresponding
                  id with Graph::newNode;
          void addDirectedEdge(int64_t upstream_id,
               int64_t downstream_id);
               // Uses Graph::addEdge to create a directed edge
                  from upstream_id and downstream_id. This new
```

```
                 edge would be added into the <edges> vector.
         void addUndirectedEdge(int64_t id1, int64_t id2);
              // Uses Graph::addEdge to create an undirected
                 edge from id1 and id2. This new edge would be
                 added into the <edges> vector.
         vector<Node*> connectedNodes(int64_t parentId);
              // Return all nodes that are downstream of
                 parentId.
}
```