1. Draw an adjacency list and adjacency matrix representation of the undirected graph shown in Figure 14.1.

**Adjacency List.**

Vertices are given an alternate name for brevity. Reference pointers between objects are assumed – omitted for clarity.

```
V
- Snoeyink (a) → {a, b}
- Goodrich (b) → {b, a}, {b, c}, {b, d}, {b, f}, {b, g}, {b, i}
- Garg (c) → {c, b}, {c, g}
- Goldwasser (d) → {d, b}, {d, g}
- Tollis (e) → {e, f}, {e, g}
- Vitter (f) → {f, b}, {f, e}, {f, g}, {f, h}
- Tamassia (g) → {g, b}, {g, c}, {g, d}, {g, e}, {g, f}, {g, h}, {g, i}
- Preparata (h) → {h, f}, {h, g}, {h, i}
- Chiang (i) → {i, b}, {i, g}, {i, h}
```

```
E
- {a, b}
- {b, c}
- {b, d}
- {b, f}
- {b, g}
- {b, i}
- {c, g}
- {d, g}
- {e, g}
- {e, f}
- {f, g}
- {f, h}
- {g, i}
- {g, h}
- {h, i}
```
Adjacency Matrix.

Vertices are given an alternate name for brevity. Reference pointers between objects are assumed – omitted for clarity. A 1 in the adjacency matrix implies the edge exists and really represents a pointer to the correct edge object – omitted for clarity.
2. Suppose we represent a graph $G$ having $n$ vertices and $m$ edges with the edge list structure. Why, in this case, does the `insertVertex` method run in $O(1)$ time while the `removeVertex` method runs in $O(m)$ time? Statement and proof of complexity required.

**Theorem 1.** When storing a graph as an edge list, the `insertVertex` takes $O(1)$ time.

**Proof.** Inserting a vertex is equivalent to adding an element to the end of a List. This can easily be done in $O(1)$ time with a linked list or an array list structure. □

**Theorem 2.** When storing a graph as an edge list, the `removeVertex` takes $O(m)$ time.

**Proof.** In order to remove a vertex from a graph, we also must remove all edges with a source or target as the vertex. This in an edge list requires searching the whole list of edges for incident edges. This operation thus take $O(m)$ time. □
3. Would you use the adjacency matrix structure or the adjacency list structure in each of the following cases? Justify your choice.

(a) The graph has 10,000 vertices and 20,000 edges, and it is important to use as little space as possible.

(b) The graph has 10,000 vertices and 20,000,000 edges, and it is important to use as little space as possible.

(c) You need to answer the query $\text{getEdge}(u, v)$ as fast as possible, no matter how much space you use.

(a) Use an adjacency list. This is the most memory efficient data structure for graphs, while maintaining a good cost for edge operations.

(b) Use an adjacency list. This is the most memory efficient data structure for graphs, while maintaining a good cost for edge operations.

(c) Use an adjacency matrix. This graph has the fastest complexity for accessing a single edge.
4. **Bonus.** Suppose we wish to represent an $n$-vertex graph $G$ using the edge list structure, assuming that we identify the vertices with the integers in the set $\{0, 1, \ldots, n-1\}$. Describe how to implement the collection $E$ to support $O(\log n)$-time performance for the $\text{getEdge}(u, v)$ method. How are you implementing the method in this case?

Store $E$ as a map of map between the source vertex and target vertex using balanced binary search trees. Looking up the source vertex takes $O(\log n^2)$ time, and looking up the destination vertex takes $O(\log n)$ time, as each vertex has at most $n$ edges. So in total, a search would take $O(\log n)$ time.