1. Here is the final configuration after linear probing:

<table>
<thead>
<tr>
<th>C42</th>
<th>D31</th>
<th>F08</th>
<th>B10</th>
<th>E12</th>
<th>A29</th>
<th>G34</th>
</tr>
</thead>
</table>

And after quadratic probing:

<table>
<thead>
<tr>
<th>C42</th>
<th>D31</th>
<th>F08</th>
<th>B10</th>
<th>E12</th>
<th>A29</th>
<th>G34</th>
</tr>
</thead>
</table>

For your convenience (this is not a part of the question), here is the detailed info about the keys:

<table>
<thead>
<tr>
<th>Key</th>
<th>Hash value</th>
<th>final position (linear, quad)</th>
<th># collisions (linear, quad)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A29</td>
<td>7</td>
<td>7, 7</td>
<td>0, 0</td>
</tr>
<tr>
<td>C42</td>
<td>0</td>
<td>0, 0</td>
<td>0, 0</td>
</tr>
<tr>
<td>E12</td>
<td>5</td>
<td>5, 5</td>
<td>0, 0</td>
</tr>
<tr>
<td>D31</td>
<td>1</td>
<td>1,1</td>
<td>0, 0</td>
</tr>
<tr>
<td>F08</td>
<td>2</td>
<td>2, 2</td>
<td>0, 0</td>
</tr>
<tr>
<td>G34</td>
<td>7</td>
<td>8, 8</td>
<td>1, 1</td>
</tr>
<tr>
<td>B10</td>
<td>0</td>
<td>3, 4</td>
<td>3, 2</td>
</tr>
</tbody>
</table>

2. Build an undirected graph of \( n \) nodes, each for one specimen. An edge connects every pair of specimens, \( i, j \), for which a judgment is made. The edges are also labeled as “same” or “different” (no need for ”ambiguous” since it doesn’t add any new info). Then run a modified version of BFS to find inconsistencies in the graph (there is some resemblance to the two-coloring algorithm, although the graph is not necessarily bipartite... What should be bipartite in case of a consistent graph is the subgraph with only “different” edges):

- Start from an arbitrary vertex \( s \). Assign it to species A (or B. It’s arbitrary at this point and doesn’t really matter).
- Run a modification of BFS starting from \( s \):
  - For each vertex \( v \) that’s visited right now, look at every neighbor \( w \):
    - If \( w \) is already visited, check for inconsistencies. That is – \( v \) and \( w \) are assigned to the same species but \( e = (v, w) \) is labeled “different” or vice versa. In this case stop the algorithm and declare the graph inconsistent.
    - If \( w \) is unassigned, assign it according to the information in \( v \) and the edge \( e = (v, w) \). If \( e \) is labeled “same”, assign to \( w \) the same species as \( v \). If \( e \) is labeled “different”, assign the other species to \( w \).
    - If \( w \) is unassigned, put it in the queue.
    - Notice that by construction, \( v \) is assigned by the time we visit it. We only visit vertices after they’ve been assigned by their predecessor, or if they’re \( s \) in which case we assign it in the beginning.
  - If not all the vertices have been explored by the time you finish, run another phase starting from an unassigned vertex.
  - Return “True” if no inconsistencies were found at any stage.
You can do it with DFS as well, it doesn’t matter. If you run BFS you don’t have to keep track of the distances. Notice that the runtime is the same as BFS/DFS, $O(m + n)$, since we still look at every edge twice (one from each side) and the consistency check adds constant time to each loop.

3. Let us call the shortest path from $s$ to $t$ $P = \{s, v_1, v_2, ..., t\}$. We know that $P$ has $> \frac{n}{2}$ edges where $n$ is the number of vertices in the graph. So $P$ has $> \frac{n}{2} + 1$ vertices, including $s$ and $t$ or $\geq \frac{n}{2}$ vertices not including $s$ and $t$, since the number of vertices along a simple path is always one more than the number of edges.

Let’s assume by contradiction that no vertex other than $s$ or $t$, when removed from the graph, can disconnect $s$ and $t$. This can only happen if there are at least two disjoint paths between $s$ and $t$ (so they share only $s$ and $t$, nothing else). In this case, the removal of any vertex would still leave the other paths intact. In other words, there is another path $P' = \{s, w_1, w_2, ..., t\}$ such that no $w_i$ is equal to no $v_j$ for any $i, j$. However, since $P$ is the shortest path between $s$ and $t$, $P'$ must be at least as long as $P$, which would imply that it also has $> \frac{n}{2}$ vertices not including $s$ and $t$. Since $P$ and $P'$ are disjoint, this would mean that the graph must have at least $n$ vertices not including $s$ and $t$ and at least $n + 2$ vertices in total, contradicting the fact that the graph has only $n$ vertices overall.

An algorithm to find such a node (called “an articulation point”) is based on DFS from $s$ in this case. In DFS tree, a vertex $u$ is articulation point if one of the following two conditions is true:

1) $u$ is the root of a DFS tree and it has at least two children on the tree (this can only happen if we can’t reach from one child to another in any other way but through $u$, think of DFS).

2) $u$ is not the root of a DFS tree and it has a child $v$ such that no vertex in subtree rooted in $v$ has a back edge to one of the ancestors (in DFS tree) of $u$. A back edge is a non-tree edge that goes from a descendant to its ancestor in the DFS tree (if such an edge exists, it’s a “backup route” from $v$ to another vertex above $u$, so disconnecting $u$ still leaves the backup intact).

4. The DFS order of visitation is as follows (only edges that lead to newly discovered vertices are depicted):

- $v_1$
- $v_1 - v_2$
- $v_2$
- $v_2 - v_4$
- $v_4$
- $v_1 - v_5$
- $v_5$
- $v_5 - v_3$
- $v_3$
- $v_3 - v_6$

The graph with tree edges marked looks like this:

5. Suppose the graph has a Hamiltonian path $P = \{v_1, v_2, ..., v_n\}$. It’s easy to see that the order of vertices along the path is a valid topological sort, since by definition for every $v_i, v_{i+1}$ along the path, there is a path (of size 1) from $v_i$ to $v_{i+1}$. The trickier part is to show that this is the only way to topologically sort the graph. Let’s assume by contradiction that there is another valid topological sort to the graph. In this case, there is at least one pair of vertices, say $v_i, v_j$ (not necessarily adjacent),
such that $v_i$ appears before $v_j$ in one topological sort, but after $v_j$ in the other. This can only happen if there is no path between $v_i$ and $v_j$ in the graph. But $P$, the Hamiltonian path, connects all the vertices in the graph, so there must be a sub-path from $v_i$ to $v_j$ (or vice versa) along $P$, a contradiction.

$\Leftarrow$ Suppose the graph has a only one topological ordering $T = \{v_1, v_2, ..., v_n\}$. We need to show that in this case there must be an edge (not just any path) from $v_i$ to $v_{i+1}$ for every consecutive pair of vertices along the topological order, hence a Hamiltonian path. Let’s assume by contradiction that there isn’t an edge from some $v_i$ to $v_{i+1}$, a pair of vertices appearing consecutively in $T$. The only two ways this can happen in a DAG are:

(a) There is a path of size $> 1$ from $v_i$ to $v_{i+1}$. This means there is at least another vertex $v_j$ along the path. However, by definition, any topological sorting of the graph would put $v_j$ between $v_i$ and $v_{i+1}$, a contradiction to the assumption that $v_i$ and $v_{i+1}$ are adjacent in the topological ordering of the graph.

(b) $v_i$ and $v_{i+1}$ are not connected by any path. However, this would imply that we can reverse their order in the topological sort and get another valid topological sort (since an order is only forced between two vertices connected by a path), a contradiction to the assumption that there is only one topological sort to the graph.

The algorithm is therefore – Topologically sort the graph. Follow the vertices in the order of the sort. If every consecutive pair of vertices along the sorted order is connected by an edge, you found a Hamiltonian path. The runtime is the same as the topological sort runtime, $O(m + n)$.

6. Here is a simple example. Vertices B and C are both a distance of 1 from A, but there is a path also from B to C, so a topological order must be A, B, C. However, B and C are equidistant from A, so a topological sorting based on the distance may put C before B if we explore C before B in BFS.

7. Show the process of Kruskal’s algorithm to find its minimal spanning tree. The format of your answer should be the following: write down the edges in the order in which they are processed, and indicate for each edge whether it appears in the final MST or not.

In Kruskal’s algorithm, we greedily select edges in increasing order of weight, as long as they don’t close a cycle.

- $v_3 - v_4$
- $v_2 - v_3$
- $v_2 - v_4$ (not a tree edge, closes a cycle)
- $v_1 - v_4$
- $v_1 - v_5$

The other edges are certainly not tree edges, so we can stop here.

8. Do the same with Prim’s algorithm. Start from $v_1$.

In Prim’s algorithm, we start from an arbitrary vertex and greedily grow the tree from it, selecting the smallest edge that crosses the cut.

- $v_1 - v_4$
- $v_3 - v_4$
9. Draw the final MST (despite possibly selecting the edges in a different order, the MST should be the same for (a) and (b)).