1. Suppose we are given an *undirected, unrooted* tree $T$ with $n$ vertices, represented using an adjacency list data structure. The tree $T$ necessarily has $n-1$ edges.

   For any two vertices $u$ and $v$ of $T$, let $path_T(u,v)$ denote the unique path from $u$ to $v$ in $T$. For any three vertices $u,v,w$ of $T$, let $meet_T(u,v,w)$ denote the unique vertex of $T$ that lies on all three paths $path_T(u,v)$ and $path_T(u,w)$ and $path_T(v,w)$.

   Describe and analyze a data structure that supports the following query:

   - MEET$(u,v,w)$: return the vertex $meet_T(u,v,w)$.

   For full credit, your solution should have the following components:

   - A description of your actual data structure
   - An analysis of the space used by your data structure
   - A preprocessing algorithm that builds your data structure from an adjacency list for $T$
   - An analysis of the running time of your preprocessing algorithm.
   - A query algorithm that implements MEET.
   - A brief argument that your query algorithm is correct.
   - An analysis of the running time of MEET.

**Solution:** I assume that the input tree $T$ is represented using a standard adjacency list. Each vertex $v$ points to a linked list of $b$'s neighbors; specifically, $v$ stores a pointer $v.first\_nbr$ to one of its neighbors, and each neighbor record $w$ stores a pointer $w.next\_nbr$ to the next neighbor in the list.

- **Data structure:** We choose a root vertex $r$ and treat $T$ as a rooted tree. Then we build a data structure that supports LCA queries in $T$ in $O(1)$ time, using $O(n)$ space, as described in the lecture notes.
- **Preprocessing:** First we need to transform $T$ into a *rooted* tree. We start by choosing an arbitrary vertex $r$ to be the root of $T$. For each node $v$, we compute the distance $v.depth$ from $r$ to $v$ in $T$ using a standard breadth-first search. Now we can treat $T$ as a rooted tree, *without building a separate data structure*, using the following algorithms to navigate through the children of each vertex:

```
⟨⟨Return pointer to first child⟩⟩
FIRSTKID(v):
  ⟨⟨We know that v.first_nbr ≠ NULL⟩⟩
  if v.first_nbr.depth < v.depth
      return v.first_nbr.next_nbr
  else
      return v.first_nbr
```

```
⟨⟨Return pointer to next sibling⟩⟩
NEXTSIB(w):
  if w.next_nbr = NULL
      return NULL
  else if w.next_nbr.depth < v.depth
      return w.next_nbr.next_nbr
  else
      return w.next_nbr
```

  (Alternatively, we could duplicate $T$ into a standard rooted tree data structure, but why waste memory?)

  Once we've converted $T$ into a rooted tree, we preprocess $T$ into a data structure that answers LCA queries using $O(n)$ space and $O(1)$ query time, exactly as described in the lecture notes. (Building this structure requires computing the vertex depths, but we've already done that.)

- **Preprocessing time:** The time to convert $T$ into a rooted tree is dominated by breadth-first search, which takes $O(n)$ time. Building the LCA-query data structure for $T$ takes $O(n)$ time, as described in the lecture notes.
- **Query algorithm:**

```
MEET(u, v, w):
  if LCA(u, v) = LCA(u, w)
      return LCA(v, w)
  else if LCA(u, w) = LCA(v, w)
      return LCA(u, v)
  else ⟨⟨LCA(u, v) = LCA(v, w)⟩⟩
      return LCA(u, w)
```

- **Proof of correctness:** There are two cases to consider:
  - If $lca(u, v) = lca(u, w) = lca(v, w) = x$, then $meet(u, v, w) = x$.
  - Otherwise, without loss of generality, $lca(u, v) \neq lca(v, w)$. Both $lca(u, v)$ and $lca(v, w)$ are ancestors of $v$, so one must be a proper ancestor of the other. Without loss of generality, suppose $lca(u, v)$ is a proper ancestor of $lca(v, w)$. Then $lca(u, v) = lca(u, w)$ and $meet(u, v, w) = lca(v, w)$.
- **Query time:** MEET makes at most five LCA queries, each of which is answered in $O(1)$ time, so its overall running time is $O(1)$.

■