**Step 1: Add an "Edit Tree" Mode**

Before adding full user-driven functionality, let's create the infrastructure for tree editing. This means adding buttons or menu options in the app that allow users to interact with the tree (e.g., adding or editing nodes).

**Task:**

* Add a button labeled "Edit Tree" to toggle between "view mode" and "edit mode."
* When in "edit mode," allow users to select a node and perform basic actions like adding or renaming a child.

**Step 2: Allow Adding New Nodes**

**ability to add a clade with multiple genera and multiple species under that genera at once?**

Once "Edit Tree" mode is active:

* Add a user interface component (like an input box and button) for the user to add a child to the selected node.

**Task:**

* Implement the ability to select a node and add a new child to it dynamically.
* Example: If the user selects "Amniota," they should be able to add children like "Synapsida" or "Sauropsida."

**Step 3: Allow Editing Existing Nodes**

Allow the user to rename nodes or update their data, or change the placement of a parent or child

**Task:**

* Create a simple "Rename" feature that enables users to update the name of a node.
* Move an existing node under an existing parent node

**Step 4: Allow Deleting Nodes**

Users should be able to delete nodes, but this will require careful handling:

* Deleting a node should also delete all of its descendants.
* You must ensure that users cannot delete the root node.

**Task:**

* Add a "Delete Node" feature with a confirmation dialog.

**Step 5: Add Ability to Insert Nodes Between Parent and Children. And add ability to create new parent nodes**

This is more advanced. Users should be able to insert a node between a parent and its current children.

**Task:**

* Implement a way to select a parent, then add a child that becomes the parent of the current children.

**Step 6: Save and Load Trees**

Provide functionality for users to save their trees to a file (e.g., JSON) and load saved trees for editing.

**Task:**

* Create a "Save" button to save the tree structure.
* Create a "Load" button to load a tree into the app.

**Step 7: Finalize and Polish**

After implementing the above functionality:

* Refine the user interface to make it intuitive.
* Add error handling (e.g., ensure nodes cannot have duplicate children).
* Allow users to switch between multiple saved trees.

Yes, that’s correct. The functionality to fix the typo and make **Oviraptorisauria** (or any existing node) a child of **Maniraptora** should be part of the **edit node** function rather than the **add node** function.

Here’s why:

* **Add Node**: This function is for creating new nodes and optionally assigning them as children or parents. It’s not designed to manage existing nodes beyond adding relationships at the moment of creation.
* **Edit Node**: This function is where you would manage relationships between existing nodes, such as:
  1. Changing the parent of an existing node.
  2. Reassigning a node to a different parent.
  3. Fixing typos in names or relationships.

**Solution: Adding Reassign Functionality to edit\_node**

The **edit node** function should:

1. Allow the user to select or type the node they want to edit.
2. Provide an option to change its parent or add/remove children.

For your specific case:

* You would use the **edit node** function to make **Oviraptorisauria** a child of **Maniraptora**.
* This involves finding **Oviraptorisauria**, removing it from its current parent (if any), and adding it under **Maniraptora**.

Would you like to proceed with implementing this feature in the **edit node** function? If yes, we can build it to handle your case and allow broader editing capabilities for nodes.

**Checklist for Add Node Functionality**

1. **Edge Case Handling**:
   * **Duplicate Nodes**: Confirm that duplicate names are properly handled and errors are displayed consistently.
   * **Empty Fields**: Verify all relevant fields are required, and appropriate messages are shown for missing fields.
   * **New Root Node**: Ensure the new root node functionality works correctly, reassigning the current root as its child.
2. **Input Validation**:
   * **Parent Name Matching**: Confirm that searching for a parent node is case-insensitive.
   * **Existing Children Names**: Ensure children reassignment works even when the input contains leading/trailing spaces or empty entries.
   * **Reserved Names**: (Optional) Consider disallowing reserved names like “Root” or “None” for node names to avoid ambiguity.
3. **UI Feedback**:
   * All error or success messages should be consistent and intuitive.
   * The application’s response (e.g., tree display) should update dynamically after a node is added.
4. **Performance Testing**:
   * Test with a large number of nodes to ensure the app can handle the tree structure efficiently.
5. **Code Robustness**:
   * Ensure all try-except blocks are correctly placed to catch errors without breaking the program.
6. **Unit Testing**:
   * Create test cases to verify the following:
     + Adding a valid node under an existing parent.
     + Adding a new root node.
     + Handling invalid parent or child node names.

**Suggestions Before Moving On**

1. **Refactor Duplicate Message Strings: To maintain consistency, store common messages as constants or variables. For example:**

**ERROR\_NODE\_NOT\_FOUND = "Node '{}' not found.\n" ERROR\_DUPLICATE\_NODE = "Node '{}' already exists under '{}'.\n"**

**Then use these constants in the code:**

**self.display\_area.insert(tk.END, ERROR\_NODE\_NOT\_FOUND.format(child\_name))**

1.  **Confirm Add Functionality with Complex Trees**: Test edge cases with deeply nested trees, missing children, and invalid inputs. This ensures your add functionality is robust before moving on to edit and delete.
2.  **Document the Add Node Process**: Add in-code comments or a README section explaining the add node functionality, covering its parameters and limitations. This will help with future debugging or collaboration.