**Final Report**

**1. Architecture Diagram**

The application follows a Model-View-Controller (MVC)-like pattern, separating the user interface from the business logic and data persistence.

* **View (ZettelkastenApp):** The main GUI class built with customtkinter. It is responsible for all user-facing elements, such as windows, buttons, and text fields. It handles user input and delegates actions to the NoteManager.
* **Controller (NoteManager):** The central orchestrator. It acts as the bridge between the GUI and the backend. It initializes all data structures and the database connection, and it contains the high-level logic for CRUD operations (e.g., create\_note calls methods on both the NoteDatabase and the in-memory Graph).
* **Model (NoteDatabase, Graph):** The data layer.
  + NoteDatabase: Handles all direct communication with the SQLite database for persistence. It performs the raw SQL queries for creating, reading, updating, and deleting notes, links, and tags.
  + Graph: An in-memory representation of the note relationships, implemented as an adjacency list. This allows for fast queries of note connections without constantly hitting the database.

**UML Component Diagram (Description):**

A UML diagram would show the ZettelkastenApp having a dependency on the NoteManager. The NoteManager would then have dependencies on both the NoteDatabase and the Graph classes.

graph TD

A[User] --> B(ZettelkastenApp - GUI);

B --> C{NoteManager - Controller};

C --> D[NoteDatabase - Model];

C --> E[Graph - Model];

D -- Interacts with --> F[(SQLite Database)];

**2. Data Structures & Algorithm Analysis**

**a. Tree (Jeremy)**

* **Implementation:** A logical tree structure is implemented using a parent\_id foreign key in the notes table of the SQLite database. The GUI uses a ttk.Treeview to visually represent this hierarchy.
* **Operations & Complexity:**
  + **Insertion (\_add\_note):** O(N) in the worst case, as building the visual tree requires iterating through the notes to place them correctly. The database insertion itself is O(log N) due to indexing.
  + **Search (\_search\_notes):** O(N) where N is the number of notes, as the search query performs a LIKE scan on the title and content columns.
  + **Deletion (\_delete\_selected\_note):** O(log N) for the database deletion due to the primary key index. Reloading the visual tree is O(N).

**b. Graph (Adrian)**

* **Implementation:** A graph is implemented as an adjacency list using Python's collections.defaultdict(list). This in-memory graph is loaded at startup for fast access. Visualization is handled by networkx and matplotlib.
* **Operations & Complexity:**
  + **Add/Remove Node:** O(1) on average for the in-memory dictionary operation.
  + **Add/Remove Edge (Link):** O(1) on average.
  + **Get Connected Notes:** O(1) on average to look up the note ID in the dictionary.

**c. Hash Map (Kindness)**

* **Implementation:** A persistent hash map is implemented using the tags and note\_tags tables in SQLite. The tags.name column has a UNIQUE index, which acts as the hash key.
* **Operations & Complexity:**
  + **Tagging a Note (update\_note\_tags):** For each tag, this involves a lookup (SELECT) and a potential insertion (INSERT). Due to database indexing, these operations are highly efficient, averaging O(log K) where K is the number of unique tags.
  + **Finding Notes for a Tag (get\_notes\_for\_tag):** O(log K + M) where K is the number of tags and M is the number of notes with that tag. The lookup is fast due to the index on the tag name.

**3. Challenges & Solutions**

* **Challenge:** Creating a modern and responsive user interface with standard Tkinter.
  + **Solution:** We migrated the GUI to the customtkinter library. This provided modern, theme-able widgets and better layout control, significantly improving the application's appearance and user experience.
* **Challenge:** Visually representing the network of linked notes.
  + **Solution:** We integrated the networkx and matplotlib libraries. networkx was used to build a graph object from our data, and matplotlib was used to draw and render that graph in a separate Tkinter window, providing an intuitive visualization of the knowledge base.
* **Challenge:** Managing the application's state and decoupling the UI from the data logic.
  + **Solution:** We implemented a NoteManager class that acts as a central controller. The GUI only calls methods on the NoteManager, which then coordinates operations between the database (NoteDatabase) and the in-memory data structures (Graph), following a clear separation of concerns.

**4. References**

* **CustomTkinter:** <https://github.com/TomSchimansky/CustomTkinter>
* **NetworkX:** <https://networkx.org/documentation/stable/>
* **Matplotlib:** <https://matplotlib.org/stable/contents.html>
* **SQLite:** <https://www.sqlite.org/docs.html>