UniHub CLI — Academic Resource Hub for NIT Trichy

Team Name: UniHub

Team Members:

Adarsh – 106124008
Sarvesh – 106124048
Sagar – 106124108

October 29, 2025

Abstract

UniHub CLI is a command-line application designed to serve as a centralized platform for students at the National Institute of Technology, Trichy (NITT) to share and access academic resources efficiently. Motivation stems from the need for a structured system tailored to NITT's specific academic organization (branches, years, semesters, sections), enhancing resource accessibility and collaboration. The primary objective is to implement and apply fundamental data structures learned in the CSLR31 Data Structures course to solve a real-world problem, creating a practical tool for the NITT community.

1 UniHub CLI: Design and Module Explanation

1.1 1. Introduction

UniHub CLI aims to be a dedicated platform for NIT Trichy students to manage and share academic resources. It addresses the lack of a centralized, NITT-specific system by providing features tailored to the institute's academic structure (branches, semesters, sections) and leveraging various data structures for efficient operation. The core idea is to apply concepts from the Data Structures course (CSLR31) to build a practical, useful tool. This document details the design, modular architecture, and the role of different data structures within the system (see the project codebase under `UniHub-CLI/Code/`).

1.2 2. Overall Architecture

The application follows a modular design, coordinated by a central `UniHubCore` class (`include/unihub_core.h`). This core class integrates functionalities from specialized manager classes responsible for different aspects of the application: User Management, Academic Management, Resource Indexing, and Navigation. A dedicated `EnhancedMenu` class handles the command-line interface (CLI) interactions (`include/enhanced_menu.h`). Low-level file operations are abstracted by a `Storage` module (`include/storage.h`).

The primary modules are:

1. **Enhanced Menu (UI Layer):** Manages user interaction, displays menus and prompts, and invokes `UniHubCore` functions.
2. **UniHub Core (Coordinator):** Central hub holding instances of manager classes, maintaining application state (like current user), and delegating tasks.
3. **User Manager (User Data & Auth):** Handles user registration, login, profile updates, and manages user data using hybrid data structures. Relies on the `Auth` module for credential handling.
4. **Auth (Authentication Logic):** Implements password hashing, salting, and credential file I/O (`include/auth.h`, `src/auth.cpp`).

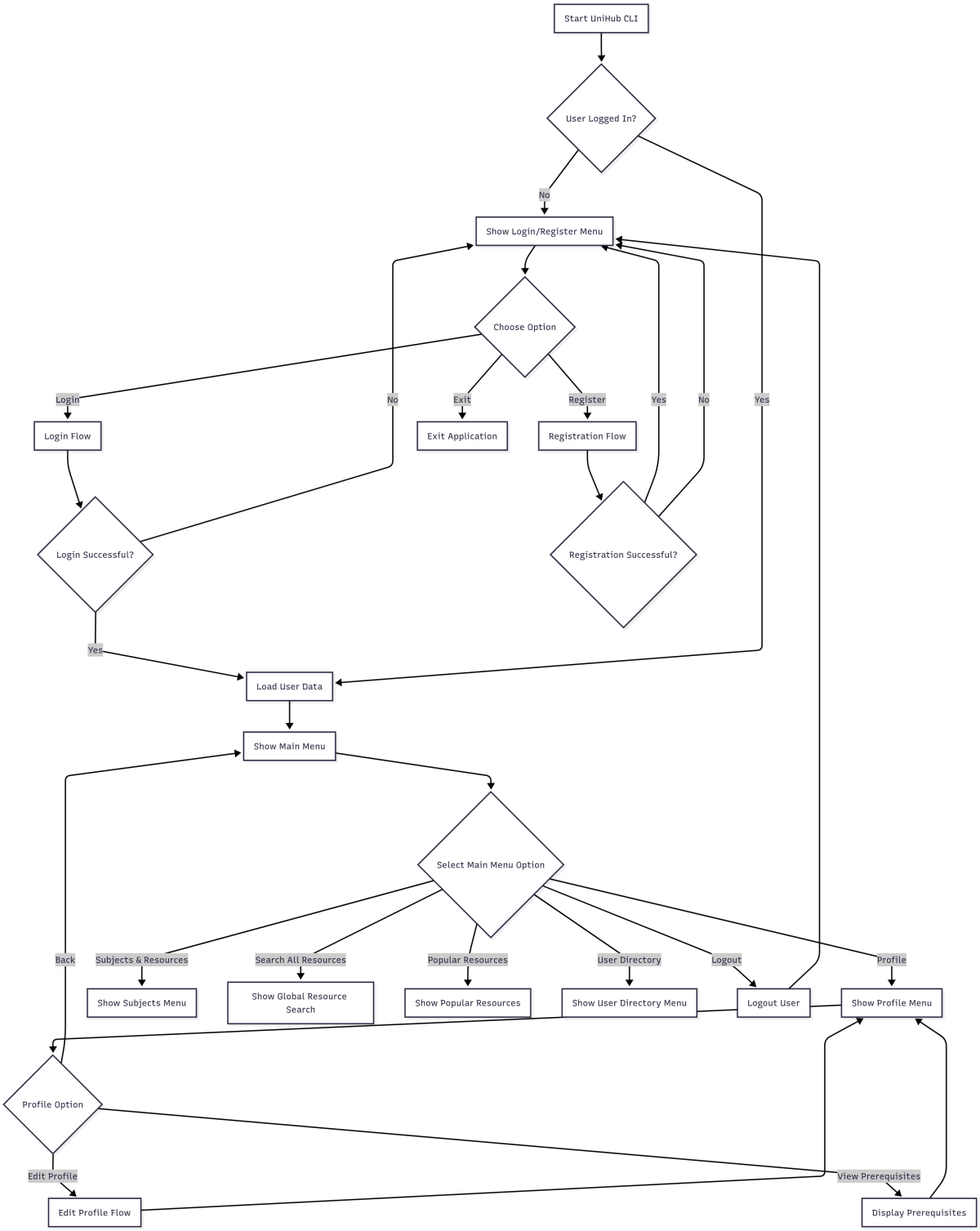

Figure 1: Flowchart of UniHub CLI System

5. **Academic Manager (Academic Structure):** Manages NITT's academic hierarchy, subjects, curriculum variations, and prerequisites.

6. **Resource Index (Resource Metadata & Search):** Indexes resources using multiple data structures for efficient searching, ranking, and retrieval. Relies on the `Resources` module for basic file operations.

7. **Resources (Resource File Operations):** Provides functions to list, upload, and download

resource files (`include/resources.h`, `src/resources.cpp`).

8. **Navigation Manager (UI State):** Tracks user navigation using a stack for back functionality and breadcrumbs.

9. **Storage (File System Abstraction):** Provides basic file and directory manipulation utilities.

1.3 3. Module Descriptions

1.3.1 3.1. Enhanced Menu (`enhanced_menu.h`)

Purpose: Provides the command-line interface for the user. It handles displaying menus, getting user input, parsing commands, and interacting with the `UniHubCore`.

Working: It runs the main application loop. Based on whether a user is logged in, it presents different menus (Login/Register or Main Menu). It interprets user choices (numbers or letters) and calls corresponding methods in `UniHubCore`. It also utilizes the `NavigationManager` (via `UniHubCore`) to display breadcrumbs, enhancing user orientation.

Implementation: Implemented as a class `EnhancedMenu`. It contains methods for different screens (login flow, registration flow, profile view, subjects menu, resource menus, etc.). It uses `std::cin` for input and `std::cout` for output. Input validation (e.g., using `std::stoi`, checking ranges) is performed.

1.3.2 3.2. UniHub Core (`unihub_core.h`)

Purpose: Acts as the central orchestrator, connecting all other modules and managing the overall application state, particularly the currently logged-in user session.

Working: It holds instances of `UserManager`, `AcademicManager`, `ResourceIndex`, and `NavigationManager`. When the `EnhancedMenu` requests an action (e.g., login, view subjects, search resources), `UniHubCore` delegates the call to the appropriate manager. It maintains the `currentUser` state using `std::optional<UserRecord>`.

Implementation: Implemented as the `UniHubCore` class. It provides a unified interface to the functionalities offered by the underlying managers.

1.3.3 3.3. User Manager & Auth (`user_manager.h`, `auth.h`, `auth.cpp`)

Purpose: Manages all user-related data, authentication, and session information. The `Auth` part specifically handles secure credential storage and verification.

Working:

- **Registration:** Takes `Profile` data and a password. Uses the `Auth` module to generate a salt, hash the password (the current implementation uses `std::hash`), and stores credentials (`.cred` file) and profile (`.profile` file) via the `Storage` module. Adds the new user's email to internal data structures.
- **Login:** Takes email and password. Reads credentials using `Auth`, re-hashes the input password with the stored salt, and compares hashes. If successful, loads the `UserRecord` and updates the LRU cache.
- **Data Management:** Uses a hybrid approach:
  - `std::unordered_map` (Hash Table): `emailIndex` provides average O(1) lookup of `UserRecord` pointers by email for fast login checks and profile retrieval.
  - `AVLTree<std::string>`: `sortedEmails` stores emails in a self-balancing binary search tree, allowing efficient retrieval of all users in sorted order.
  - `std::list` & `std::unordered_map` (LRU Cache): `recentUsers` and `recentIndex` track the most recently accessed users. Accessing a user moves them to the front of the list in O(1) time.
  - `Graph<std::string>`: `socialGraph` allows storing connections between users (e.g., friends, study groups).
- **Profile Update:** Saves updated profile data to the `.profile` file and updates the in-memory record if present.

Implementation: `UserManager` class uses the data structures mentioned. The `Auth` functions (`registerUser`, `login`, `loadProfile`, `saveProfile` in `auth.cpp`) handle file I/O via the `Storage` module and perform hashing (currently `std::hash` with salt). Profile data is stored as simple comma-separated values.

1.3.4 3.4. Academic Manager (`academic_manager.h`, `subjects.h`, `subjects.cpp`)

Purpose: Represents and manages the academic structure of NITT, including branches, subjects, curriculum variations, and prerequisites.

Working:

- **Hierarchy:** Intended to use a Tree structure (`TreeNode`) to represent University → Branch → Year → Semester, although the current implementation mainly uses maps for direct access.
- **Subjects:** Stores `EnhancedSubject` details (code, name, teacher, credits, year, semester, branch, section, prerequisites) in a `std::unordered_map` (`subjectMap`) for quick O(1) average lookup by subject code.
- **Branches:** Stores `Branch` information (code, full name, max years) in a `std::unordered_map` (`branches`).
- **Prerequisites:** Uses a `DAG<std::string>` (`prerequisiteGraph`) where nodes are subject codes and directed edges represent prerequisite relationships (e.g., edge from `CSE11A` to `CSE12A`). Allows checking prerequisites and generating topological sorts (potential course sequences).
- **Curriculum:** Initializes a default curriculum (e.g., `initializeCSECurriculum`) and allows for overrides based on branch/year/semester/section.

Implementation: `AcademicManager` class encapsulates the logic. `EnhancedSubject` struct holds detailed subject info. `DAG` class is implemented using `std::unordered_map` for the adjacency list and in-degree tracking.

1.3.5 3.5. Resource Index & Resources (`resource_index.h`, `resources.h`, `resources.cpp`)

Purpose: Manages metadata about academic resources, provides efficient search and retrieval capabilities, and interacts with the file system for actual file operations.

Working:

- **Metadata Storage:** Stores `ResourceMetadata` (filename, display name, path, type, subject, uploader, size, timestamp, download count, rating, tags).
- **Indexing:** Uses multiple structures for different search needs:
  - **`BST<ResourceMetadata>`:** `resourceBST` stores metadata, enabling O(log n) average search/insertion based on the chosen key.
  - **Simple Autocomplete (Array-based):** `resourceNameAutocomplete` stores resource display names in a sorted `std::vector`, providing prefix-based suggestions.
  - **`std::priority_queue<ResourceMetadata>`:** `popularResources` maintains resources ordered by download count (or rating), allowing quick retrieval of top items.
  - **`Graph<std::string>`:** `resourceGraph` stores relationships between resources (e.g., related lecture notes).
  - **`std::unordered_map` (Inverted Index):** `invertedIndex` maps keywords (lowercase, punctuation removed) from titles, subjects, types, and tags to lists of resource filenames, enabling basic full-text search.
- **File Operations:** The `Resources` module interacts with the `Storage` module to list files in directories (`listResources`), copy files for upload (`uploadResource`), and copy files for download (`downloadResource`).

Implementation: `ResourceIndex` class manages the metadata and indexes. `ResourceMetadata` struct holds resource details. `BST`, Simple Autocomplete, and `Graph` are implemented in `data_structures.h`. The `Resources` functions use `std::filesystem` via the `Storage` module.

1.3.6 3.6. Navigation Manager (`unihub_core.h`)

Purpose: Manages the user's navigation state within the application's menu system.

Working: Uses a `std::stack<NavigationState>` (`history`) to keep track of visited menus. When navigating forward (`MapsTo`), the current state is pushed onto the stack. When going back (`goBack`), the state is popped from the stack. It also stores context (like the current subject code) relevant to the current menu state in a `std::unordered_map` within `NavigationState`. Provides the `getBreadcrumbs` function, which reconstructs the path from the stack for display.

Implementation: Implemented as the `NavigationManager` class within `unihub_core.h`. `NavigationState` struct holds location, description, and context.

1.3.7 3.7. Storage (`storage.h`, `storage.cpp`)

Purpose: Provides a low-level abstraction layer for interacting with the file system.

Working: Offers functions to get standard directory paths (`dataDir`, `resourcesDir`), ensure directories exist (`ensureDir` using `std::filesystem::create_directories`), read/write text files (`readTextFile`, `writeTextFile` using `std::ifstream`, `std::ofstream`), list directory contents (`listFiles` using `std::filesystem::dire` and copy files (`copyFile` using `std::filesystem::copy_file`).

Implementation: Consists of standalone functions within the `uni` namespace using C++17 `std::filesystem` features. Error handling is basic (e.g., returning `std::optional` or boolean flags).

2 4. Conclusion

The design of UniHub CLI effectively utilizes a variety of data structures taught in a typical undergraduate course. Each structure is chosen to optimize specific operations: hash tables for fast lookups, trees (AVL, BST) for ordered data and efficient searching, DAGs for dependency tracking, priority queues for ranking, graphs for relationships, arrays for simple lists/autocomplete, stacks for navigation, and lists/maps for caching. The modular architecture, coordinated by `UniHubCore`, allows for separation of concerns and integrates these components into a functional command-line application for academic resource sharing at NITT. The hybrid use of multiple data structures demonstrates how different structures can be combined to address the varied performance requirements of a real-world application.