PROJECT DESCRIPTION

[COL 106 LONG ASSIGNMENT]

This project is a C++ implementation of a simplified version control system. It provides functionality for creating and managing files, taking snapshots, updating content, rolling back to earlier versions, and tracking file history. The design uses multiple data structures such as trees for organizing versions, heaps for retrieving files having largest version count or frequently updated files, and hash maps for fast indexing and lookup.

The system is organized into multiple files :

* **filefunctions.cpp** handles file operations such as create, read, update, snapshot, rollback, and history.
* **filestructure.cpp** defines the core classes (File, TreeNode) and manages file data, content, and version details.
* **heap.cpp** manages files based on time and version counts using heap structures, supporting efficient retrieval of recent or largest versioned files.
* **map.cpp** implements hash maps for mapping file names and versions to file objects, ensuring fast access and storage. It consists of 2 maps- one version\_map for mapping version\_id to Version (This is implemented via Vector). And another map for mapping file\_name and File (This is implemented via Hashmap).
* **mainfunction4.cpp** serve as drivers for testing and demonstrating different functionalities. **The mainfunction also contains layout for inputs. The program will run only on a specific type (layout) of input . The instructions for that are in the mainfunction. It is assumed that user will enter correct inputs only. Example :   
  if user has created 2 files and then asks for Biggest 3, then code might not run correctly.**

**Usage Guide**

**Running the Project**

The primary way to compile and run this project is through the run.sh script provided. This shell script is intended for **Linux/Unix** environments and will automatically compile the mainfunction4.cpp (it already includes all other program files ) and run the program. On Linux, you can execute it using:

***chmod +x run.sh***

***./run.sh***

**Windows**: A run.bat script and the compiled mainfunction4.exe are also provided. Running run.bat will automatically execute the .exe without needing manual compilation. Since all other files are directly included in mainfunction4.cpp, the executable already contains the complete functionality.

**Command Usage**

This program is a simple **file version management system** that allows users to create files, edit them, and maintain multiple versions through snapshots. The user interacts with the program by typing commands into the console.

**Features / Commands**

* **CREATE <filename> [content]** : Creates a new file with optional initial content.
* **READ <filename>** : Displays the current content of the given file.
* **INSERT <filename> <extra\_content>** : Appends new content to the file.
* **UPDATE <filename> <new\_content>** : Replaces the file’s content with new content.
* **SNAPSHOT <filename> [message]** : Creates a snapshot (like a commit) of the file with a message.
* **ROLLBACK <filename> <version\_id>** : Rolls back the file to a previous version by version ID.
* **HISTORY <filename>** : Shows all snapshots of the file with version ID, content, message, and timestamp. Time stamp is shown in the local time of the users machine .
* **RECENT\_FILES <n>** : Displays the n most recently edited files. **Note: If the time interval between two consecutive inputs is shorter than the resolution provided by the ctime library, the program may produce incorrect results, as it will be unable to distinguish between the two recorded timestamps**.
* **BIGGEST\_TREES <n>** : Shows the n files with the largest number of versions (biggest version trees). **NOTE – The program might cause undefined behaviour if user has created k files only but commands BIGGEST\_TREES <N> where N>k. In that case kill the terminal and run again.**
* **END** : Ends the program and clears all stored data.

**Error Handling in File Management System**

The system handles invalid operations and edge cases to avoid crashes and give clear messages to the user.

**1. File Checks:**  
Before doing read, insert, update, snapshot, or rollback, it checks if the file exists. If not, it prints "No such file exists."

**2. Version Checks:**  
For operations like rollback, it checks if the version ID is valid. If not, it prints *"No such version exists."*

**3. Snapshot Rules:**  
Snapshots are immutable. If the active version is a snapshot, any insert or update creates a new version instead of changing the snapshot.

**4. Duplicate Files:**  
createFile checks if a file already exists and asks the user to choose a new name.

**5. Handling Invalid Inputs:**  
Insert and update check for valid content. Invalid operations are rejected safely. **However, incomplete commands like read or update without arguments can cause a crash.**

**6. Branch-Aware History:**  
History only shows snapshots along the branch of the active version, so unrelated branches are not shown.

**7. Default Parameters:**  
Rollback has default version ID 0, so it safely rolls back to root if no version is specified.

**8. Safety Measures:**  
All pointers are checked before use. Heaps for recent files and version tracking are updated only after valid operations. Invalid actions print messages instead of crashing.

**Overall, the project demonstrates how fundamental data structures can be applied to build a functional version control system**.

**filefunctions.cpp**  
This file provides the primary operations for file handling and version control. It includes:

* createFile() – Creates a new file entry in the system.
* read() – Reads the contents of a file.
* insert() – Inserts new content into a file.
* update() – Updates existing file content.
* snapshot() – Captures the current state of a file (like a commit). Note that snapshotting same file twice merely changes the message and timestamp of snapshot.
* rollback() – Reverts a file back to a previous version.
* history() – Displays the version history of a file. The time stamp

Classes: None defined in this file.

**filestructure.cpp**  
This file defines the structures and classes used to represent files and their versions.

* **Classes**:
  + File – Represents a file, storing its name, content, creation time, snapshots, and metadata.
  + TreeNode – Represents a node in a version tree, linking snapshots and changes.
* **Functions**:
  + commitsnapshot() – Saves a new snapshot of a file.
  + status() – Returns the current status of a file.
  + get\_version\_id() – Retrieves the version identifier of a file.
  + get\_content() – Gets the current file content.
  + change\_content() – Modifies the file’s content.
  + get\_snapshot\_time() – Returns the time a snapshot was created.
  + get\_created\_time() – Returns the file creation time.
  + getname() – Retrieves the file’s name.

**heap.cpp**  
This file manages files using heap data structures for efficient retrieval.

* **Classes**:
  + File – Represents file metadata within heaps.
  + heapManager\_time – Manages a heap that sorts files based on last edited time.
  + heapManager\_versions – Manages a heap that sorts files based on version count. The versions include snapshotted as well as non snapshotted versions .
* **Functions**:
  + addFile() – Inserts a file into the heap.
  + recentFiles() – Retrieves the most recently edited files.
  + heapify\_time\_down() / heapify\_time\_up() – Maintains heap order for time-based management.
  + updateFile\_time() – Updates a file’s timestamp in the heap.
  + removeFile() – Removes a file from the heap.
  + biggestTrees() – Retrieves the files with the largest version trees.
  + heapify\_version\_down() / heapify\_version\_up() – Maintains heap order for version-based management.
  + updateFile\_version() – Updates a file’s version count in the heap.

**map.cpp**  
This file implements hash map structures for mapping files and versions.

* **Classes**:
  + fileMap – Maps file names to their corresponding file objects. Uses HashMap based implementation.
  + versionMap – Maps version numbers to file objects. Uses simple vector for implementation.
* **Functions**:
  + insert() – Inserts files or versions into the map.
  + get\_size() – Returns the size of the map.
  + hashFunction() – Generates hash values for efficient indexing.
  + rehash() – Resizes and reorganizes the map when load factor exceeds limits.