Time-Travelling File System

Amaan Khan

September, 2025

§1 Introduction

This project implements a file version-management system with the following features:

- Creating new files and their versions and efficiently managing them in a version-tree.
- Reading existing files
- Inserting, updating and snapshotting existing file versions
- Rollback to older versions
- Printing the history of modifications of a file
- Printing recently edited files and files with the largest version trees.

§2 Command Reference

The following section specifies the command-syntax for performing operations.

- **INPUT:** The input is taken interactively with each command being executed in real-time. To terminate the program type the command `EXIT` or press `CTRL + D`.
- **CREATE FILE:** Use the command `CREATE <filename>`. This command creates a file with a root version (ID, 0), empty content, and an initial snapshot message. The root is marked as a snapshot. The code skips the command and displays an error message if a file with that name already exists, otherwise prints a message to confirm that the command has been executed successfully.

- **READ FILE:** Use the command `READ <filename>`. This command displays the content of the file's currently active version. The code skips the command and prints an error message if the file does not exist.

- **INSERT FILE:** Use the command `INSERT <filename> <content>`. This command appends the content to the file. If the file is a snapshot it creates a new version, otherwise modifies the version in place. The code skips the command and displays an error message if the file does not exist, otherwise prints a message to confirm that the command has been executed successfully.

- **UPDATE FILE:** Use the command `UPDATE <filename> <content>`. This command replaces the file's content. Same versioning logic as insert. The code also skips the command and displays an error message if the file does not exist, otherwise prints a message to confirm that the command has been executed successfully.

- **SNAPSHOT:** Use the command `SNAPSHOT <filename> <message>`. This command marks the active version as a snapshot, making its contents immutable. It stores the provided message and the current time (empty message if its empty). The code skips the command and displays an error message if the file does not exist. If the file is already snapshotted, it displays a message signifying that as well, otherwise prints a message to confirm that the command has been executed successfully.

- **ROLLBACK:** Use the command `ROLLBACK <filename> [versionID]`. This command sets the active version pointer to the specified `versionID` and displays a message on successful execution. If no ID is provided it rolls back to the parent and displays a message regarding this as well. If the file doesnt exist it prints an error message. Otherise if the `versionID` is invalid/ doesnt follow input constraints prints an error message as well.

- **HISTORY** Use the command `HISTORY <filename>`. Lists all snapshotted versions of the file chronologically, which lie on the path from the active node to the root in the file tree, showing their ID, timestamp and message. If the file does not exist prints an error message.

- **RECENT_FILES** Use the command `RECENT_FILES [num]`. Lists the files in descending order of their last modification time restricted to the first `num` entries. If `num` doesn't follow the required input format the prints an error message. Otherwise if `num` > total_size of the heap then it lists all the entries and displays a message showing not enough elements present in the heap.

- **BIGGEST_TREES** Use the command `BIGGEST_TREES [num]`. Lists the files in descending order of their total version count restricted to the first `num` entries. If `num` doesn't follow the required input format the prints an error message. Otherwise if `num` > total_size of the heap then it lists all the entries and displays a message showing not enough elements present in the heap.

- **INVALID COMMAND:** If the command matches none of the defined commands in this section, then the code skips the command and prints an error message.

§3 System Architecture

§3.1 Tree Class

Tree data structure is used to maintain the version history of each individual file. Each node within the tree represents a specific version of the file. The tree node structure includes

- `version_ID`: A unique sequentially assigned integer for each version, starting from 0.
- `content`: The textual content of the file version.
- `parent`: A pointer to the parent node of the tree.
- `children`: A vector of pointers to its child nodes in the tree.
- `created_timestamp`: The time at which the version was created.
- `message`: A snapshot message associated with the version, which is empty if it is not a snapshot.
- `snapshot_timestamp`: The time at which the version was snapshotted, which is null if it is not a snapshot.

§3.2 File Class

The File class manages the version history of a single file. It encapsulates all the logic and data structures necessary to create, modify, snapshot, and navigate through a file's various versions. The version history of this class is structured as a tree. It consists of:

- `version_map`: A hashmap that maps a unique `versionID` to its corresponding tree node pointer, allowing for efficient $O(1)$ access to any version within the file's history.
- `filename`: String storing the name of the file.
- `total_versions`: An integer that keeps a running count of all versions created for this file, used to assign unique `versionID`.
- `root`: A pointer to the tree node that represents the initial, root version of the file (ID 0).
- `active_version`: A pointer to the tree node that is currently active. All operations are applied relative to this version.
- `last_modified_time`: A timestamp `(time_t)` recording the time of the last modification to the file (e.g. `CREATE, INSERT, UPDATE`)
- `last_execution_number`: An integer used for system-wide analytics, to track the order of operations (used for tiebreaks).

§3.3 FileSystem class

The FileSystem class manages all files and providing system-wide analytics. It acts as the central hub, utilizing heaps to efficiently track and report on file metrics. It consists of:

- `recent_heap`: A heap that stores `FileLastMod` objects, which contain a pointer to a `File` object and its `last_modified_time`. This heap is organized to allow for $O(\texttt{num} \log n)$ retrieval of files in descending order of their last modification time.
- `biggest_heap`: A heap that stores `FileSize` objects, which contain a pointer to a `File` object and its `total_versions` count. This heap is organized to efficiently list files in $O(\texttt{num} \log n)$ time with the files having largest number of versions in descending order.

§3.4 MaxHeap

Efficiently manages and retrieves the maximum element from a collection of data. It uses a `index_map(HashMap)` to track the positions of elements in the heap, enabling $O(\log n)$ updates and deletions. It consists of:

- `heap`: A vector that stores the heap elements. The heap property is maintained such that the parent node is always greater than or equal to its children.
- `index_map`: A `HashMap` that maps a unique key (in this case, the filename of a file) to its corresponding index within the heap vector. This allows for $O(1)$ average-time lookups to find an element's position facilitating efficient updates and deletions.

§3.5 HashMap and HashMap file class

The `HashMap` and `HashMap_File` classes are implementations of a hash map data structure which provide fast, $O(1)$ average-time lookups. Each bucket in the hash table is a linked list of nodes. The `HashMap` class handles integer keys (like `versionID`) and tree node pointers as values, while `HashMap_File` handles string keys (like `filename`) and any value type. They both consist of:

- `table`: A vector of pointers to `HashNode`'s representing the hash table.
- `max_size`: An integer tracking the current capacity of the hash table.
- `curr_size`: An integer counting the number of elements currently stored in the hash map.
- `load_factor`: A constant that determines the ratio of elements to table size, triggering a rehash when exceeded (dynamic resizing).

§4 Setup Instructions

- Enter the following commands in terminal

```
cd directory_of_project
chmod +x col106_run_assignment_1.sh
./col106_run_assignment_1.sh
```

- Enter the commands in the input format specified in the previous section.

§5 Requirements

- `C++` compiler (e.g. `g++` or `clang++`)
- Install all the files locally and put them in the same directory before running the code.
- Preferred OS: Linux/MacOS.

§6 Important Notes

- Used `<ctime>` library for timestamps.
- On modifying a snapshotted version, the new version created is set as the active version.
- `INSERT, UPDATE, CREATE` are considered as modifications.
- `HISTORY` command prints all the timestamps in the ascending order of node creation time (i.e. root(ID 0) is printed first)
- Assumed `filename` has no spaces.
- In command `BIGGEST_TREES` if there is a tie in `total_versions` then the file which was the last modified is printed first.
- In command `RECENT_FILES` if there is a tie in `timestamp` then the file which had a later modification statement is printed first.