Time Travelling File System README

Adarsh Raj

September 10, 2025

1 Overview

This project implements a **File Versioning and Snapshot System** in C++ with the following features:

- Create and manage multiple files.
- Insert and update file contents.
- Read file content.
- Take snapshots of file states with messages.
- Rollback to previous versions or specified version_id.
- View file history (snapshot logs from active node to root node).
- System-wide queries:
  - `BIGGEST_TREES` – Find files with the most versions.
  - `RECENT_FILES` – Find most recently updated files.
- Implemented with custom:
  - Hash maps for file lookups.
  - Heaps for recent/biggest file tracking.
  - Tree structure for file versions.

2 Data Structures

2.1 TreeNode

Represents a file version node.

- Stores content, snapshot flag, timestamps,message ,version_id and parent/children relations.
- Supports deletion of all children via destructor.

2.2 MyHashMap

- Custom hash map for mapping `version_id` to `TreeNode*`.
- Handles collisions using separate chaining.
- Supports rehashing (whenever loadFactor > 0.75 rehashing is done).

2.3 file_struct

Represents a single file with filename.

- Maintains root node, active version,total version and version history.
- It also contain the indices that required during heap implementation and the time when the file is last updated via command CREATE,INSERT,UPDATE.
- Supports `read`,`insert`, `update`, `snapshot`, `rollback`, and `history`.

2.4 HashMap (Root Map)

- Maps `filename` to its corresponding `file_struct*`.
- Implemented by standard string hashing i.e roll hash.
- Used for global file management.

2.5 MyHeap

- Two heaps are maintained: vector Heap1,Heap2.
- Each heap store pointer to file_struct i.e file_struct*.

2.5.1 Functionality:

- **push**:Push every new file_struct* to it's correct place.
- **update**:Update every updated file_struct* i.e during insertion or updation to it's correct place.
- **heapify_down**:Whenever pop used it is popped by swapping top value with last value in heap then use heapify down top element.
- **pop**:Pop file_struct*.

1. By `total_versions` (largest file trees) .Every thing maintained in Heap1.
2. By `last_updated` timestamp (recent files).Every thing maintained in Heap2.

3 Supported Commands

- `CREATE <filename>`
  Creates a new file.
- `READ <filename>`
  Reads current content.
- `INSERT <filename> <content>`
  Appends content to file if not snapshotted otherwise form new version.
- `UPDATE <filename> <content>`
  Replaces file content.
- `SNAPSHOT <filename> <message>`
  Marks current state as snapshot.
- `ROLLBACK <filename> [version_id]`
  Rollback to parent(when version_id is not given) or specific given version.
- `HISTORY <filename>`
  Displays snapshot history.
- `BIGGEST_TREES <num>`
  Shows top files with most versions.
- `RECENT_FILES <num>`
  Shows most recently updated files.
- `EXIT`
  Terminates the program.It is must to give `EXIT` command to terminate the program.

4 Example Usage for each file

```
CREATE file1
INSERT file1 Hello
SNAPSHOT file1 Initial commit
INSERT file1 World
HISTORY file1
ROLLBACK file1 0
READ file1
```

5 SYSTEM WIDE ANALYTICS

More than `one file` are created and updated.

```
RECENT_FILES n
BIGGEST_TREES n
```

(n is the number filename required)

6 Compilation and Execution

```
g++ long_assignment.cpp -o output
./output.exe
```

7 Assumptions

- **File Representation**
  - Each file is represented using a `file_struct` object.
  - The root of the file is a `TreeNode`, which forms the base of a version tree.
  - Every version of the file corresponds to a new `TreeNode`.
  - A snapshot node indicates a frozen version where further edits create child nodes.
- **Versioning Logic**
  - Version identifiers (`version_id`) start from 0 and increment with each new version.
  - Inserts or updates on non-snapshot nodes directly modify content.
  - Inserts or updates on snapshot nodes create new child versions.
  - Rollback moves to the parent version (when no ID is given) or to a specific version ID if provided.
- **HashMaps**
  - Two hashmaps are used:
    1. `MyHashMap`: Maps integer version IDs to `TreeNode` pointers for individual files.
    2. `HashMap`: Maps string filenames to `file_struct` pointers for the system.
  - Integer modulus hashing is used for version IDs.
  - Polynomial rolling hash is used for strings.
  - A load factor threshold of 0.75 is assumed sufficient to avoid excessive collisions.
- **Heap Structures**
  - Two max-heaps maintain global ordering of files:
    1. `heap1`: Ordered by total number of versions (`total_versions`).
    2. `heap2`: Ordered by last updated timestamp (`last_updated`).
  - File indices (`idx1`, `idx2`) are assumed to remain consistent during heap operations.
- **Command Parsing**
  - Input is read line by line and split into three parts: `cmd`, `filename`, and `rest`.

  - Commands follow a strict format such as:
    * `CREATE filename`
    * `INSERT filename content`
    * `UPDATE filename content`
    * `SNAPSHOT filename message`
  - Extra arguments where not expected are treated as errors.

- **Memory Management**
  - The `TreeNode` destructor recursively deletes all child nodes.
  - The `HashMap` destructor deletes all file structures.
  - It is assumed there are no memory leaks, since destructors handle cleanup.

- **Timestamps**
  - `created_timestamp` records when a version is created.
  - `snapshot_timestamp` records when a version is snapshotted.
  - `last_updated` stores the last modification time of a file.
  - `time(nullptr)` with second-level granularity is assumed sufficient.

- **Rollback and Validations**
  - Rollback without argument moves to the parent version.
  - Rollback with an integer argument jumps to the corresponding version.
  - Version IDs are assumed to be continuous integers starting from 0.

- **Error Handling**
  - Errors are reported using console messages instead of exceptions.
  - Invalid operations include accessing non-existent files, invalid arguments, or requesting more files than available.

- **Termination**
  - The program runs in an infinite loop until the `EXIT` command is entered with no arguments.
  - It is assumed that users follow the correct termination command format.

8 Notes

- All data structures are implemented from scratch (no STL maps or heaps).
- Memory management is carefully handled with destructors.
- Designed for learning low-level implementation of file versioning systems.