**Overview**

This project implements a simplified, in-memory version control system inspired by Git. It allows users to create and manage versioned files, supporting branching, snapshots, history inspection, and analytics. The system is entirely in-memory and leverages custom implementations of core data structures: trees, hash maps, and heaps.

**Core Data Structures**

***Tree****:* Represents the version history for each file. Each version is a node, with branching support for snapshots and rollbacks. At each node, pointers are stored for its children and parent nodes, along with a modularized public interface that prevents direct manipulation from the outside, except for the specified parameters (content and message only).

***File:*** Object that holds the most recently changed time and the tree pointer to the root of the file structure tree along with a hash map

(implemented as a vector due to the dense and incremental nature of version IDs for efficient rollback in O(1)).

***Val***: This is a file wrapper to implement a data structure port, which I have used to store the pointer to the file object and the IDs of its position in both the heaps named appropriately. This wrapper also holds pointers to two functions, the size comparison function and the time comparison function, so that when the heaps call update on this wrapper object, it would be able to detect which parameter to update by validating that the pointer to the function stored in the heap is the same as the one corresponding to the value to be updated.

This was done so that the file objects can have independent existence irrespective of the heaps and hash maps so that in the future the files can be manipulated in other ways if needed. A shared pointer to this object has been used, as it is being accessed by various data structures through pointers, and raw pointers in such a case can cause memory leaks or floating pointers.

***HashMap:*** HashMap uses polynomial hashing and a custom-made structure buck to store the Val pointers whose hash is colliding. I have given the a capacity parameter so that rehashing takes place when the structure is storing about 5 times as many values as capacity. The reason I have used the threshold of 5 times instead of standard 0.7 times would be clear on going forward.

***Buck:*** I have tried to innovate and use a bit of knowledge that I had previously in this data structure. I have used a vector in it to store key-value pairs, but in addition to that, if the number of files exceeds 10, I have used a red-black tree in place of the plain vector (my reasoning behind this was that hash maps have an average complexity of O(1), but in case of excessive collisions or delibrate attack on the system by making hashes collide the complexity can eventually exceed to O(n) (too many key value pairs in the same bucket) to prevent this we can use a red black tree based map which would reduce the worst case complexity to O(logn) making the file system imvulnerable to attacks and bad luck.

Heap: I have implemented the Heap in such a way that it stores a pointer to the function that is used to maintain the heap ordering and updates the object whose pointer is stored in it whenever the position of the pointer is being changed. The indices of the Val pointer in the heap are maintained as correct after each bubble up or bubble down operation but passing the function pointer stored in the heap so the Val wrapper can appropriately update itself.

***Red Black Tree:*** The red-black tree implemented in the appropriate header file is insert only and does not support erasures as it was not required by the assignment to support file deletions the 0 color encodes Black nodes and 1 encodes Red nodes. As its a balanced binary search tree we can find the required node in O(logn) time and rotate left and rotate right functions have been implemented so that the binary tree can be rebalanced after each insertion and required recoloring is also done so that the constraint of red nodes not having red children, Black root and each path from a node to leaves having same number of black nodes is maintained. The rebalance approach in an insertion only tree is simple as we only have to add red children and balance the tree upward until the parent of the child is red.

Main Components

File, TreeNode (in File.hpp): Represent files and their versioned structure.

Val (in Data\_struct\_wrapper.hpp) : wrapper for file object to connect them to the data structures.

Hashmap.hpp & RBTree.hpp: File storage and version map, with dynamic switching between vector and tree for buckets.

Heap.hpp: Manages heaps for recency and size analytics.

2024CS10204.cpp: Main program logic and UI.

**How Inputs and Outputs Are Handled**

Input

The program reads commands from standard input (stdin). Each command should be entered as a single line following the format specified below. Inputs are parsed and sanitized to extract file names, content, and additional flags. The input is sanitized in such a manner that the whitespaces in the front and end of the input would be automatically removed but while adding inputs 1 whitespace is always added in front of the previous message. To add multiple whitespace and new lines escape characters should be used.

**Supported Commands**

CREATE <filename>

Creates a new file with an initial empty snapshot with the default snapshot message “root node of file”.

READ <filename>

Prints the content of the current version of the file.

INSERT <filename> <content>

Appends content to a file. Creates a new version if active version is a snapshot; else, modifies active version in-place. If new version is created it would be displayed.

UPDATE <filename> <content>

Replaces the content of the file. New version logic is similar to INSERT. If new version is created it would be displayed.

SNAPSHOT <filename> <message>

Marks the active version as a snapshot with a custom message and displays error message if the version is already a snapshot.

ROLLBACK <filename> [versionID]

Sets the file’s active version pointer to the specified version, or parent if no ID is given.

HISTORY <filename>

Lists all snapshots in chronological order with details.

RECENT\_FILES [num]

Lists [num] most recently modified files and their version counts. If [num] is not specified, lists all files.

BIGGEST\_TREES [num]

Lists [num] files with the most versions, by descending count. Prints all if [num] is omitted.

EXIT

Exits the system, cleaning up all files from memory.

Output

Most commands produce informative output via standard output (stdout), such as file content, version lists, error messages, and confirmations.

Error Handling: If a command is not recognized, is malformed, or refers to a missing file, the system prints a descriptive error message and continues. If the error message is not specifies for a particular input a generic message has been encoded in the catch block to error handle and continue the program.

Commands expecting filenames or version IDs sanitize input and check existence; operations on missing files or versions are not performed and explained with errors.

Error Handling Philosophy

Snapshots, Inserts and updates with content missing are initialized with an empty string, Rollbacks without version id are rolled backed to their parent except root which when rollbacked would print an error message.

**Compilation Instructions**

All .cpp and .hpp files should be in the same working directory.

To Compile for Windows (I have a windows machine)

Run the following in your terminal after putting the folder in the current working directory by cd command

cl /EHsc /std:c++17 /Fe:<Program\_name>.exe 2024CS10204.cpp

<Program\_name>.exe

to run the program

**Design Philosophy**

I have templated the code wherever possible maintaining the independence of various structures and tried to make the best design decisions. The heap, hash map, File object have been made as generic as possible so that the code can be easily debugged and more features can be easily added while (trying) following best practices of public and private variables.