Long Assignment 1:

Time-Travelling File System

SHUBHRADEEP PAUL

ENTRY NO. - 2024MT10112

Architecture

Self Implemented Hashmap, Tree and Heap using only Vectors

Core Components

TreeNode

Represents individual versions in the file's version history tree.

- **Properties**: Version ID, content,message, timestamps, parent/children relationships
- **Features**: Snapshot capability, version structure
- **Usage**: version tracking

HashMap<k,v>

Custom hash table implementation with chaining for collision resolution.

- **Key Features**: Template-based, multiple hash functions for different int and strings
- **Operations**: Insert, find, remove with O(1) average complexity
- **Collision Handling**: Linked list chaining

MaxHeap<k,v>

Maxheap implementation with HashMap-based indexing for fast updates.

- **Key Features**: Template-based, supports key-based updates

- **Operations**: Push, pop, update with O(log n) complexity
- **Indexing**: Uses custom HashMap for index tracking

File

Manages individual file's version history and operations.

- **Version Tree**: Maintains hierarchical version relationships
- **Active Version**: Tracks current working version
- **Operations**: Insert, update, snapshot, rollback

FileSystem

Main system managing multiple files with advanced querying capabilities.

- **File Management**: Create, read, update, delete operations
- **Analytics**: Recent files and biggest version trees tracking

Features

Version Control

- **Hierarchical Versioning**: Tree-based version history
- **Rollback**: Navigate to any previous version

Advanced Queries

- **HISTORY**: View complete version timeline

  It gives in reverse chronological order from most recent to root

  Prints all the snapshot files giving their version id, timestamp and snapshot message
- **RECENT_FILES**: num number of Most recently modified files
- **BIGGEST_TREES**: num number of Files with most versions
- **ROLLBACK**: Active version Rolled back to desired version or parent version

Time Complexity Analysis

FileSystem Class Operations

Core File Management Functions

- **file_exists()** - **O(1)**: HashMap lookup with average constant time
- **create_file()** - **O(log n)**: HashMap insertion + 2 heap insertions + snapshot creation
- **read_file()** - **O(1)**: HashMap lookup + direct content access
- **insert_in_file()** - **O(log n)**: HashMap lookup + content append + 2 heap updates
- **update_file()** - **O(log n)**: HashMap lookup + content replacement + 2 heap updates

Version Control Functions

- **snapshot_file()** - **O(log n)**: HashMap lookup + snapshot setup + recent heap update
- **rollback_file()** - **O(1)**: HashMap lookup + HashMap-based version lookup
- **snap_message()** - **O(1)**: HashMap lookup + direct message access
- **total_versions()** - **O(1)**: HashMap lookup + counter access
- **version()** - **O(1)**: HashMap lookup + counter access

History and Navigation Functions

- **file_history()** - **O(h)**: HashMap lookup + tree traversal (h = tree height)
- **get_active_version()** - **O(1)**: HashMap lookup + pointer access

Analytics and Query Functions

- **recent_files()** - **O(k log n)**: MaxHeap top_num operation (k = requested count)
- **biggest_trees()** - **O(k log n)**: MaxHeap top_num operation (k = requested count)

Heap Management Functions

- **update_recent()** - **O(log n)**: MaxHeap update or insertion for recent time files tracking

- **update_biggest() - O(log n)**: MaxHeap update or insertion for version count tracking

Detailed Analysis

O(1) Operations

- **File Existence Check**: Direct HashMap lookup with constant average time
- **File Reading**: HashMap lookup followed by immediate content access
- **Version Rollback**: HashMap-based version indexing for instant version access and pointing
- **Data Access**: Direct property access after single HashMap lookup
- **Message Retrieval**: Direct access to snapshot message through active version pointer
- **Version Counting**: Simple counter access after file lookup

O(log n) Operations

- **File Creation**: Requires insertions into both recent and biggest MaxHeaps for analytics

  A slight compromise here O(log n) instead of O(1) but helps in the bigger picture of Recent Files and Biggest Trees
- **Content Modifications**: HashMap lookup is O(1), but subsequent heap updates are O(log n)
- **Snapshot Creation**: Updates recent files heap to maintain proper recency ranking
- **Heap Updates**: All heap maintenance operations follow binary heap properties

O(h) Operations

- **History Retrieval**: Traverses version tree from current node to root (h = tree height)
- **Worst Case**: O(v) where v is total versions if tree becomes completely linear (h = v)
- **Average Case**: O(log v) for balanced version trees

O(k log n) Operations

- **Top-K Queries**: MaxHeap's top_num() temporarily removes k elements and re-inserts them
- **Recent Files**: Extracts top k files by modification time while preserving heap structure
- **Biggest Trees**: Extracts top k files by version count (similar to recent files)

Performance Characteristics Summary

Best Case Scenarios

- **File Operations**: O(1) for read create insert update etc owing to Hashmap DS
- **Version Navigation**: O(1) with direct HashMap access to any version
- **File Queries**: O(1) for all basic information retrieval operations

Average Case Performance

- **Most Operations**: Dominated by heap maintenance at O(log n) complexity
- **Hash Collisions**: Minimal impact with good hash distribution and polynomial rolling hash
- **Version Tree**: Typically balanced structure keeps history traversal efficient

Worst Case Considerations

- **Hash Collisions**: O(n) for HashMap operations in pathological cases with poor hash function
- **Linear Version Tree**: O(v) for history traversal in sequential versioning patterns
- **Heap Operations**: Always O(log n) due to complete binary tree structure

Usage Examples

Basic File Operations

```
CREATE myfile.txt          # Create new file
INSERT myfile.txt "Hello"   # Add content
UPDATE myfile.txt "World"   # Replace content
READ myfile.txt            # View current content
```

Version Control

```
SNAPSHOT myfile.txt "Initial version"  # Create snapshot
INSERT myfile.txt " World"             # Modify file
ROLLBACK myfile.txt 0                  # Return to version 0
HISTORY myfile.txt                     # View version history
```

System Queries

```
RECENT_FILES 5      # Show 5 most recent files
BIGGEST_TREES 3     # Show 3 files with most versions
```

Command Reference

Command	Syntax	Description
CREATE	`CREATE <filename>`	Create new file with initial snapshot
READ	`READ <filename>`	Display current file content
INSERT	`INSERT <filename> <content>`	Append content to file and increase version if not a snapshot
UPDATE	`UPDATE <filename> <content>`	Replace file content and increase version if not a snapshot
SNAPSHOT	`SNAPSHOT <filename> <message>`	Create immutable version if not already snapshotted
ROLLBACK	`ROLLBACK <filename> [version_id]`	Navigate to specific version or parent if version not given
HISTORY	`HISTORY <filename>`	Show version timeline chronologically
RECENT_FILES	`RECENT_FILES <num>`	List num recently modified files
BIGGEST_TREES	`BIGGEST_TREES <num>`	List num files with most versions
EXIT	`EXIT`	Terminate program

Input Handling

The program works best if the input given is according to the given syntax however for specific wrong inputs , the errors are handled

For example:

Missing Arguments Detection

- **Command Parsing**: Uses `cin.peek()` to detect missing arguments before attempting to read them
- **Stream Recovery**: Uses `cin.clear()` to reset error flags and continue processing

File Existence Validation

- **Duplicate Creation Prevention**: Checks if file already exists before creation
- **Operation Prerequisites**: Validates file existence before all file operations

Version Validation

- **Boundary Checking**: Validates version IDs are within valid range
- **Version Map Verification**: Ensures version exists in the version mapping
- **Even if vession in ROLLBACK is** not given the program correctly handles it to roll it back to the parent version

String Input Support

- **Quoted String Support**: Custom `input()` function handles both quoted and unquoted strings
- **Plain Strings**: Single words without quotes usually used for second argument in the operations
- **Special Characters**: Full support for whitespace and symbols
- If user needs to give a message consisting of multiple words, he must put them in quotes

  Otherwise only the first word will be included in the message and the rest will be shown as invalid include

Performance Characteristics

Space Complexity

- **Per File**: O(v) where v is number of versions
- **HashMap**: O(n)
- **Heap Operations**: O(n log n) for maintenance

Build Instructions

Prerequisites

- C++23 or later compiler

Compilation

```
g++ -std=c++23 -o COL.exe FinalDraft.cpp
```

Execution

```
COL.exe
```

System Architecture Benefits

- **Efficient Indexing**: HashMap-based fast lookups
- **Memory Optimization**: Tree structure prevents duplicate storage
- **Query Performance**: Heap-based ranking for analytics

The time complexity analysis shows that the system achieves excellent performance for most operations, with the majority being either O(1) or O(log n), making it suitable for large-scale file management scenarios.