Version Control System Documentation

Overview

This project implements a custom version control system in C++ that manages file versioning, snapshots, and provides various file management operations. The system uses a tree-based structure to represent version hierarchies and employs max heaps for efficient file organization based on timestamps and version counts.

Architecture

The system consists of three main components:

1. **Data Structures** (`structs.hpp`) - Defines core structures for nodes, files, and comparators
2. **Max Heap and Hash Map Implementation** (`classes.hpp`) - Generic max heap for priority-based operations, and a hash map for fast retrieval of Version Node as well as file object with filename in O(1) time complexity.
3. **Main Application** (`main.cpp`) - Command-line interface and core functionality

Compilation Instructions

You basically have to compile the main.cpp file and you will be able to provide instructions.

```
g++ main.cpp -o {Name of output file}
```

Core Components

1. TreeNode Structure

The `TreeNode` represents a single version of a file in the version tree.

Private Members:

- `int version_id` - Unique identifier for the version
- `string content` - File content for this version
- `string message` - Snapshot message (empty if not a snapshot)
- `time_t created_timestamp` - When this version was created
- `time_t snapshot_timestamp` - When this version was snapshotted (-1 if not a snapshot)

Public Members:

- `TreeNode* parent` - Pointer to parent version
- `vector<TreeNode*> children` - List of child versions

Key Methods:

- `TreeNode(int version_id, time_t created_timestamp, string content="", string message="")` - Constructor
- `void addChild(TreeNode* child)` - Adds a child node and sets parent relationship
- Getter methods: `get_version_id()`, `get_content()`, `get_created_timestamp()`, etc.
- Setter methods: `set_content()`, `set_message()`, `set_snapshot_timestamp()`

Important Behavior:

- Content can only be modified if `snapshot_timestamp == -1` (not snapshotted)
- Attempting to modify snapshotted content prints "IMMUTABLE"

2. File Structure

The `file` structure represents a complete file with its version history.

Members:

- `string fileName` - Name of the file
- `TreeNode* root` - Root version of the file
- `TreeNode* active_version` - Currently active version
- `map<int, TreeNode*> version_map` - Maps version IDs to TreeNode pointers
- `int total_versions` - Total number of versions created

Constructor:

Creates initial version (version 0) with current timestamp and sets it as both root and active version.

3. Comparator Structures

Comparator_time

Compares files based on their active version's creation timestamp (ascending order).

Comparator_vcount

Compares files based on their total version count (ascending order).

4. MaxHeap Class Template

A generic max heap implementation using arrays.

Key Features:

- Template-based with customizable comparator
- Array representation with standard heap indexing:
  - Parent: `(i-1)/2`
  - Left child: `(2*i)+1`
  - Right child: `(2*i)+2`

Methods:

- `void insertKey(T* node)` - Insert new element and maintain heap property
- `T* getMax()` - Return maximum element without removing
- `T* extractMax()` - Remove and return maximum element
- `void deleteKey(int i)` - Delete element at index i
- `bool isEmpty()` - Check if heap is empty

System Operations

Global Variables

- `map<string, file*> fileMap` - Maps filenames to file objects
- `MaxHeap<file, Comparator_time> timeHeap` - Organizes files by timestamp
- `MaxHeap<file, Comparator_vcount> versionHeap` - Organizes files by version count

Core Functions

File Management

- **`create(string filename)`** - Creates new file and adds to both heaps
- **`read(string filename)`** - Displays content of active version
- **`write(string filename, string newContent)`** - Modifies file content

Version Control

- **`snapshot(string filename, string message)`** - Creates snapshot with message
- **`rollback(string filename, int version_id)`** - Reverts to specified version
- **`history(string filename)`** - Shows version history with snapshots only

Display Functions

- **`show_files_by_time()`** - Lists files ordered by creation time
- **`show_files_by_version_count()`** - Lists files ordered by version count

Command Interface

The system accepts the following commands through standard input:

Command	Syntax	Description
CREATE	`CREATE filename`	Creates a new file
READ	`READ filename`	Displays file content
WRITE	`WRITE filename content`	Writes content to file
SNAPSHOT	`SNAPSHOT filename message`	Creates snapshot with message
ROLLBACK	`ROLLBACK filename [version_id]`	Rolls back to version (parent if no ID)

Command	Syntax	Description
HISTORY	`HISTORY filename`	Shows snapshot history
SHOW_VERSION	`SHOW_VERSION filename`	Shows current version ID
SHOW_FILES_BY_TIME	`SHOW_FILES_BY_TIME`	Lists files by creation time
SHOW_FILES_BY_VERSION_COUNT	`SHOW_FILES_BY_VERSION_COUNT`	Lists files by version count

Version Control Logic

Writing Behavior

1. **If active version is not snapshotted** (`snapshot_timestamp == -1`):
    - Modifies content directly in place
    - No new version created
2. **If active version is snapshotted** (`snapshot_timestamp != -1`):
    - Creates new version with incremented version ID
    - New version becomes child of current version
    - Updates active version pointer

Snapshot Behavior

- Sets snapshot timestamp to current time
- Adds message to current version
- Makes version immutable (content cannot be changed)

Rollback Behavior

- Auto-snapshots current version before rollback if not already snapshotted
- If no version ID specified: rolls back to parent version
- If version ID specified: switches to that specific version

Key Design Decisions

Immutability

Once a version is snapshotted, its content becomes immutable. Any attempt to modify it will display "IMMUTABLE" message.

Tree Structure

Versions form a tree where each version can have multiple children, allowing for branching version histories.

Heap Organization

Files are maintained in two separate heaps for efficient retrieval based on different criteria (time vs version count).

Error Handling

The system includes basic error handling for:

- File not found operations
- Invalid heap indices
- Command parsing edge cases

Limitations and Potential Improvements

1. **Memory Management**: No explicit cleanup of dynamically allocated objects
2. **Persistence**: No file system persistence - data lost when program exits
3. **Concurrency**: No thread safety mechanisms
4. **Error Recovery**: Limited error handling and recovery mechanisms