PA1 DS

TAs: Amna Hassan 20100124, Aleena Abbas 26100109, Ali Azhar 27100083

August 2025

Contents

1 Introduction to the Assignment

Welcome to Programming Assignment 1 (PA1) for Data Structures! In this assignment, you will be implementing a simplified backend system for a social media platform. This project is designed to give you hands-on experience with fundamental data structures like linked lists and concepts such as memory management and transaction tracking. By the end of this assignment, you will have built a robust system capable of managing users, posts, and their relationships efficiently.

The goal of this manual is to guide you through the implementation process step-by-step. Each component is broken down into its purpose, key features, and detailed implementation steps. Please read through the entire manual carefully before you begin coding to understand the overall architecture and dependencies.

1.1 Submission Instructions

You must **ONLY** include the `src` folder in your submission. Do not include binaries, build files, or any other directories.

Use the following naming convention for your zip file:
`PA1_<roll number>.zip`

Important: You are **not allowed to modify any header files**. Your implementation must work with the provided headers as-is.

All submissions must be uploaded on LMS before the deadline. You are allowed 5 "free" late days during the semester (that can be applied to one or more assignments). The final assignment will be due tentatively on the final day of classes (i.e., before the dead week) and cannot be turned in late. The last day to do any late submission is also the final day of classes, even if you have free late days remaining.

2 System Overview

You are tasked with implementing a social media backend system that manages users, posts, and relationships between users. The system is designed around several key components that work together to provide a scalable and efficient platform.

2.1 Core Functionality

- **User Management:** Create, delete, and manage user accounts.
- **Social Relationships:** Handle follow/unfollow operations between users.
- **Post Management:** Create, store, and retrieve user posts efficiently.
- **Memory Pool Management:** Optimize memory allocation for posts.

- **Operation Tracking:** Implement undo/redo functionality for system operations.
- **Data Persistence:** Export/import user data via CSV files.

2.2 System Workflow

1. Users are created and stored in a doubly-linked list structure.
2. Each user maintains their own list of posts and following relationships.
3. Posts are allocated from a memory pool for efficiency.
4. Operations can be tracked and undone/redone.
5. The system supports bulk operations through an ingestion queue.
6. All data can be persisted to and restored from CSV files.

3 Architecture Components

This section details the core data structures and system management components required for the social media system.

3.1 Core Data Structures

3.1.1 LinkedList Template (`linked_list.h`)

A generic doubly-linked list that serves as the backbone for storing users.

- **Purpose:** Efficient insertion, deletion, and traversal of users.
- **Key Features:** O(1) insertion/deletion, bidirectional traversal.

3.1.2 User (`user.h`)

Represents a user in the social media system.

- **Components:** `userID`, `userName`, posts list, following list.
- **Relationships:** Each user owns their posts and following relationships.

3.1.3 Post (`post.h`)

Represents a social media post.

- **Components:** `postID`, `category`, `views`, `content`.

3.1.4 FollowList (`follow_list.h`)

Manages the list of users that a given user follows.

- **Structure:** Singly-linked list of User pointers.

3.1.5 PostList (`post_list.h`)

Manages posts belonging to a specific user.

- **Structure:** Singly-linked list of Post pointers.

3.2 System Management Components

3.2.1 UserManager (`user_manager.h`)

Central controller for all user-related operations.

- **Responsibilities:** User lifecycle, relationship management, data persistence.

3.2.2 PostPool (`post_pool.h`)

Memory pool for efficient Post allocation and deallocation.

- **Benefits:** Reduces memory fragmentation, improves performance.

3.2.3 IngestQueue (`ingest_queue.h`)

Circular buffer for batch processing of posts.

- **Use Case:** High-throughput post ingestion scenarios.

3.2.4 UndoRedoManager (`operation_stack.h`)

Tracks operations for undo/redo functionality.

- **Features:** Transaction support, operation rollback.

4 Implementation Guide

4.1 Development Order

Follow this sequence to ensure dependencies are resolved correctly:

1. **LinkedList Template:** Foundation for all list operations.
2. **Post Structure:** Simple data holder.
3. **FollowList:** Basic linked list for relationships.

4. **PostList:** Basic linked list for posts.
5. **User Class:** Combines PostList and FollowList.
6. **PostPool:** Memory management for posts.
7. **UserManager:** Central coordination.
8. **IngestQueue:** Batch processing capabilities.
9. **UndoRedoManager:** Operation tracking.

4.2 Key Design Principles

4.2.1 Memory Ownership

- **Users:** Owned by `UserManager`'s `LinkedList`.
- **Posts:** Allocated from `PostPool`, referenced by `PostList` nodes.
- **Following Relationships:** User owns their `FollowList`, which references other Users.
- **Nodes:** `PostList` and `FollowList` own their respective node objects.

4.2.2 Error Handling

- Return `nullptr` for failed allocations or lookups.
- Return `false` for failed operations.
- Use defensive programming (null pointer checks).
- Prevent circular references and memory leaks.

Function Specifications

This section provides detailed specifications for each function, including their purpose and step-by-step implementation instructions.

5 LinkedList Template

You are expected to implement a doubly-linked list data structure using the template class provided in the `linked_list.h` file. The basic layout and structure are already defined for you, including the LinkedList class with pointers to head and tail nodes, along with function declarations for essential operations. Your task is to complete the implementation of these core methods, ensuring proper memory management and correct pointer manipulation. The implementation should maintain the integrity of the doubly-linked structure while providing standard linked list functionality for adding, removing, and searching elements.

5.1 Constructor/Destructor

This constructor initializes an empty linked list with _head and _tail set to null and _size to 0. The destructor ensures proper cleanup by calling the clear() method to deallocate all memory used by the list nodes.

5.2 push_back(const T& val)

Adds a new element to the end of the list and returns a pointer to the newly created node.

5.3 push_front(const T& val)

Adds a new element to the beginning of the list and returns a pointer to the newly created node.

5.4 insert_after(Node* pos, const T& val)

Inserts a new element immediately after the specified node position in the list.

5.5 remove(Node* node)

Removes the specified node from the list and deallocates its memory.

5.6 find(function<bool(const T&)> pred)

Searches through the list and returns the first node whose data satisfies the given predicate function.This function takes a predicate pred of type std::function¡bool(const T)¿, meaning it accepts any callable (like a lambda or function) that operates on a const reference to type T and returns a bool. It iterates through the list, applying pred to each element, and returns the first node where pred returns true. If no match is found, it returns nullptr.

5.7 head() const

Returns a pointer to the first node in the list (or nullptr if empty).

5.8 tail() const

Returns a pointer to the last node in the list (or nullptr if empty).

5.9 size() const

Returns the current number of elements in the list.

5.10 `clear()`

Removes all nodes from the list and frees their allocated memory, resetting the list to empty state.

6 User (user.h)

You are expected to implement a User class that manages user data and social media functionality using the structure provided in the `user.h` file. This class handles user posts through a PostList and manages following relationships via a FollowList pointer. The implementation should provide proper resource management with copy/move semantics and maintain the integrity of user relationships while preventing issues like self-following and circular dependencies.

6.1 `User(int id, const string& name)`

This constructor initializes a User object with the given ID and name, creates a new FollowList for managing followed users, and uses PostList's default constructor for the posts.

6.2 `User(const User other)`

Copy constructor creates a deep copy of a User object by copying userID and userName directly, copying posts using PostList's copy constructor, and creating a new empty FollowList to prevent circular dependency issues.

6.3 `User operator=(const User other)`

Assigns values from another User object while ensuring proper deep copying of resources and handling self-assignment cases.

6.4 `User(User other) noexcept`

Move constructor efficiently transfers ownership of resources from another User object, taking ownership of the following pointer and moving other members appropriately.

6.5 `User operator=(User other) noexcept`

Transfers ownership of resources from another User object while properly cleaning up existing resources and handling self-assignment.

6.6 `~User()`

Cleans up owned resources by deleting the following pointer, while PostList destructor handles post cleanup automatically.

6.7 `addPost(int postID, const string& category)`

Adds a new post to the user's post list by creating a Post object with the given parameters and adding it to the PostList.

6.8 `followUser(User* otherUser)`

Establishes a following relationship with another user after validating the pointer and preventing self-following scenarios.

6.9 `displayFollowing() const`

Displays the list of users that this user is currently following.

7 Post (post.h)

You are expected to implement a `Post` class that represents a social media post. This class should encapsulate essential details such as `postID`, `category`, `views`, and `content`. The `Post` class will be a fundamental building block for managing user content within the social media platform.

8 FollowList (follow_list.h)

You are expected to implement a `FollowList` class that manages the list of users a particular user is following. This class will be structured as a **singly-linked list of `User` pointers**. It should provide functionality for adding, removing, finding, and displaying followed users, ensuring proper pointer management and preventing duplicate entries or self-following.

8.1 `addFollowing(User* u)`

Adds a user to the following list. Validates the user pointer, checks for duplicates, creates a new `FollowNode` with the user, and inserts it at the head of the list for O(1) efficiency, updating the head pointer.

8.2 `removeFollowing(int userID)`

Removes a user with the specified `userID` from the following list. It traverses the list, keeps track of the previous node, updates links to skip the target node when found, and deletes the `FollowNode`. Returns `true` if found and removed, `false` otherwise.

8.3 `findFollowing(int userID)`

Searches for a followed user by their `userID`. It traverses the list from the head, checks each node's user for a matching `userID`, adds safety checks to prevent issues, and returns the `User*` if found, or `nullptr` otherwise.

8.4 `displayFollowing() const`

Prints the names and IDs of all users currently being followed. It traverses the list from the head, prints the `userName` and `userID` for each valid user, handles null user pointers gracefully, and formats the output clearly.

9 PostList (post_list.h)

You are expected to implement a `PostList` class that manages the collection of posts belonging to a specific user. This class will be structured as a **singly-linked list of `Post` pointers**. It should provide functionalities for adding, removing, and finding posts, ensuring proper memory management for the `Post` objects.

9.1 `addPost(const Post& p)`

Adds a new post to the user's list of posts. It creates a new `Post` object on the heap (using copy constructor), creates a `PostNode` wrapper, inserts it at the head for O(1) efficiency, and updates the head pointer.

9.2 `removePost(int postID)`

Removes a post with the specified `postID` from the list. It traverses the list, keeps track of the previous node, updates links to skip the target node when found, and **deletes both the `Post` object and its `PostNode` wrapper**. Returns `true` if found and removed, `false` otherwise.

9.3 `findPost(int postID)`

Searches for a post by its `postID`. It traverses the list from the head, checks the `postID` of each contained `Post`, and returns the `Post*` if a match is found, or `nullptr` otherwise.

10 PostPool (post_pool.h)

You are expected to implement a memory pool management system using the PostPool class provided in the `post_pool.h` file. This class manages efficient allocation and reuse of Post objects (defined in `post.h`) by pre-allocating blocks of memory and maintaining a free list for recycled objects. The implementation

should minimize memory fragmentation and allocation overhead by reusing previously allocated Post objects when possible, while providing proper memory management and cleanup functionality.

The memory pool is organized into contiguous blocks of fixed size, typically 4096 bytes, with each block containing multiple Post objects arranged sequentially in memory (view figure 1). When the pool is initialized, the first block is allocated immediately, and posts are allocated from this block in order using a the current block index counter that tracks the next available position. As posts are allocated, the block index increments until it reaches the maximum capacity of the current block. When the current block becomes full and no posts are available in the free list for recycling, the system allocates an entirely new block of memory, adds it to the block vector, and resets the block index to zero to start allocating from the beginning of this new block (view figure 2).

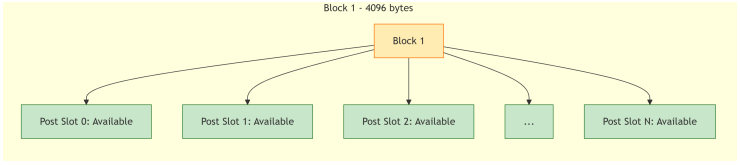

Figure 1: Block visualization

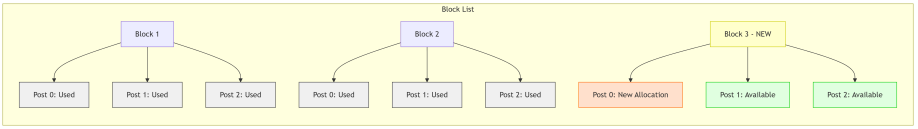

Figure 2: BlockList - The current block will be Block 3 and the current block index for this block will be 0

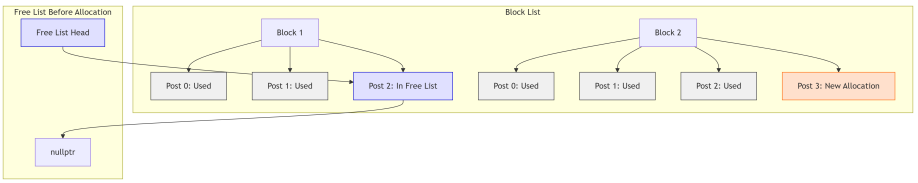

Figure 3: FreeList

The free list operates as a recycling mechanism for Post objects within the memory pool. When a post is freed through the freePost() method, the system first resets the post's internal data to a clean state, ensuring no residual content remains. The post's memory is not actually deallocated but is instead marked as available for reuse. The pointer to this recycled post is then pushed into the free list vector (as shown in figure 3), adding it to the collection of available posts

(keep in mind the pointer is not removed from the blocklist vector). Meanwhile, the reuse count is incremented to track how many posts have been returned to the pool for recycling. Other management variables like the current block index and total allocations remain unchanged during freeing, as the memory is simply being reclaimed rather than newly allocated. This approach allows the system to efficiently reuse existing memory locations without additional overhead when future allocation requests occur.

10.1 `Constructor`

Initializes the memory pool with a specified block size (default 4096) and allocates the first block immediately.

10.2 `allocPost()`

Allocates a Post object from the pool, either by reusing from the free list or allocating from the current block. Returns a pointer to an initialized Post object.

10.3 `freePost(Post* p)`

Returns a Post object to the free list for future reuse after resetting its data to a clean state.

10.4 `totalAllocations() const`

Returns the total number of memory chunk allocations performed by the pool.

10.5 `reuseCount() const`

Returns the number of times a previously freed Post object was reused instead of allocating new memory.

10.6 `purge()`

Releases all allocated memory blocks and resets the pool to its initial state for shutdown purposes.

10.7 `allocateBlock()`

Allocates a new block of Post objects and adds it to the pool's block storage. Updates allocation counters and resets block indexing.

11 UserManager (user_manager.h)

You are expected to implement a `UserManager` class, which acts as the central controller for all user-related operations. Its primary responsibilities include managing the user lifecycle (creation, deletion), handling user relationship management (following, unfollowing), and providing data persistence capabilities through CSV import/export.

11.1 `createUser(int userID, const std::string& username)`

Creates and adds a new user to the system. It first checks for duplicate `userID` and `username`. If either exists, it returns `nullptr`. Otherwise, it creates a `User` object, adds it to the internal `users` `LinkedList`, verifies the following pointer's validity, and returns a pointer to the newly created user's node.

11.2 `deleteUser(int userID)`

Removes a user with the specified `userID` and cleans up their references throughout the system. It finds the user's node by ID, removes this user from all other users' following lists, then removes the user from the main `users` list. The `User` destructor automatically handles the cleanup of the user's posts and their own following list. Returns `true` if the user was found and deleted, `false` otherwise.

11.3 `follow(int followerID, int followeeID)`

Establishes a following relationship between two users. It prevents self-following, validates that both users exist, checks the follower's following pointer, and verifies that the follower is not already following the followee. If all checks pass, it adds the followee to the follower's following list. Returns `true` on success, `false` on failure.

11.4 `unfollow(int followerID, int followeeID)`

Removes an existing following relationship between two users. It finds the follower user, validates that their following list exists, and then uses the `FollowList::removeFollowing` method to remove the specified relationship. Returns `true` on success, `false` on failure.

11.5 `isFollowing(int followerID, int followeeID)`

Checks if a following relationship exists between two users. It finds the follower user, checks if their following list exists, and then uses the `FollowList::findFollowing` method to determine if the followee is in the list. Returns `true` if the follower is following the followee, `false` otherwise.

11.6 addPost(int userID, Post* post)

Attaches a `Post` (obtained from `PostPool`) to a user's post list. It finds the user by ID, validates the `Post` pointer, and then uses the `User::addPost` method to add it to the user's collection of posts. Returns `true` on success, `false` on failure.

11.7 deletePost(int userID, PostID postID)

Removes a post from a user's list. It finds the user by ID and then uses the `PostList::removePost` method to remove and properly delete the post. Returns `true` on success, `false` on failure.

11.8 findUserByID(int userID)

Locates a user by their unique `userID`. It utilizes the `LinkedList::find` method with a lambda predicate that compares the `userID` field of each `User` object. Returns a pointer to the `LinkedList<User>::Node` if found, `nullptr` otherwise.

11.9 findUserByName(const std::string& username)

Locates a user by their `username`. It utilizes the `LinkedList::find` method with a lambda predicate that compares the `userName` field of each `User` object. Returns a pointer to the `LinkedList<User>::Node` if found, `nullptr` otherwise.

11.10 exportUsersCSV(const std::string& path)

Saves all user data to a CSV file at the specified path. The CSV format is: `userID,username,followeeID1|followeeID2|...,postID1:category1:views1|postID2:category2:views`
It opens the output file, iterates through all users, writing their ID, username, pipe-separated followee IDs, and pipe-separated post entries (with colon-separated fields). It handles empty lists appropriately.

11.11 importUsersCSV(const std::string& path)

Restores user data from a CSV file at the specified path. It first clears any existing users. It then performs a first pass to create all users and add their posts. A second pass is used to establish following relationships (ensuring all users exist before attempting to follow). It carefully parses CSV fields, handling empty values, and uses `createUser` and `follow` methods for proper initialization.

12 IngestQueue (ingest_queue.h)

The `IngestQueue` is a critical component for handling high-throughput post ingestion. It is implemented as a circular buffer, which is a fixed-size queue that efficiently reuses its memory by wrapping around from the end to the beginning.

This design allows the system to accept a large number of incoming posts quickly and process them in batches, which is far more efficient than handling them one by one. It decouples the fast act of receiving a post from the slower process of persisting it, improving overall system performance and responsiveness.

The state of the queue is managed by two indices: `head_idx`, which points to the next item to be removed, and `tail_idx`, which points to the next available slot for insertion.

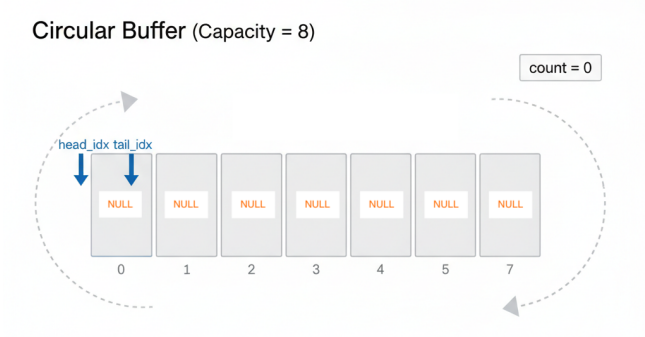

Figure 4: Initial State of an Empty IngestQueue.

As shown in Figure 4, the queue starts empty with both `head_idx` and `tail_idx` at the beginning of the buffer. As posts are added (*enqueued*) and removed (*dequeued*), these indices advance. The circular nature means that when an index reaches the end of the buffer, it wraps around to the start.

Figure 5 illustrates a queue in a later state, where several posts have been added and removed. Notice how the `head_idx` has advanced and the `tail_idx` will be wrapped around, with active posts occupying a contiguous block that spans the end and beginning of the buffer.

12.1 `explicit IngestQueue(std::size_t capacity = 8192)`

Purpose: Initializes the circular buffer. It allocates a fixed-size array of `Post*` with the given capacity, initializes all indices and the element `count` to 0, and sets all buffer entries to `nullptr` for safety.

12.2 `enqueue(Post* p) -> bool`

Purpose: Adds a `Post` pointer to the back of the queue. It first checks if the queue is full to prevent overflow. If space is available, it stores the pointer at the current `tail_idx`. The `tail_idx` is then advanced, wrapping around to the beginning of the buffer if necessary (using the modulo operator). The function returns `true` on success and `false` if the queue was full.

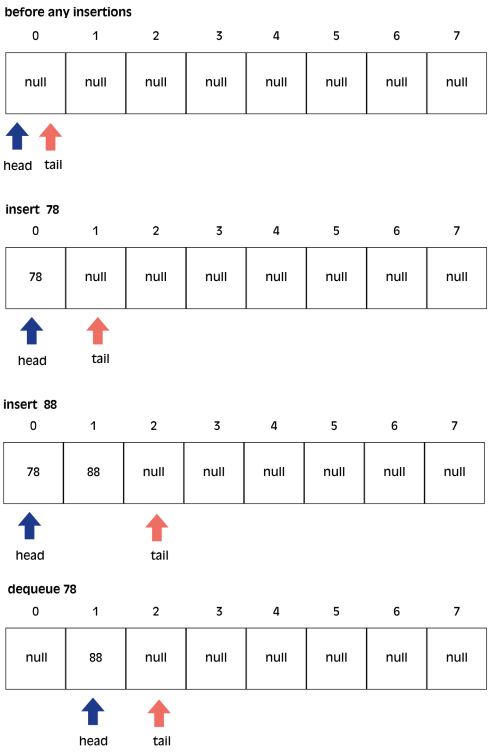

Figure 5: IngestQueue After Several Enqueue and Dequeue Operations.

12.3 `dequeue() -> Post*`

Purpose: Removes and returns the `Post` pointer from the front of the queue. It first checks if the queue is empty. If not, it retrieves the pointer from the current `head_idx`, clears the buffer slot for safety, and advances the `head_idx` (with wraparound). The function returns the retrieved `Post*` or `nullptr` if the queue was empty.

12.4 `dequeueBatch(Post** out_array, std::size_t max_k) -> std::size_t`

Purpose: Efficiently removes a batch of posts from the queue. This is more performant than dequeuing one by one. It calculates the number of items to dequeue (the minimum of the current count and `max_k`), copies that many `Post` pointers into the provided `out_array`, and updates the `head_idx` and `count` accordingly. It returns the number of items that were successfully dequeued.

13 UndoRedoManager (operation_stack.h)

You are expected to implement an UndoRedoManager class, provided in the `operation_stack.h` file, which provides transaction-based undo/redo functionality. This class manages operation history through undo and redo stacks, supports transaction scoping with rollback capabilities, and works with UserManager and PostPool to reverse various social media operations like user creation, post management, and follow relationships.

13.1 Constructor

Initializes UndoRedoManager with references to UserManager and PostPool to manage operations and memory allocation.

13.2 `beginTransaction()`

Starts a new transaction scope by creating a restore point. This allows multiple operations to be grouped together for potential rollback. The function pushes the current size of the undo stack to the transactionMarkers stack, creating a checkpoint that can be returned to during rollback.

13.3 `commitTransaction()`

Finalizes the current transaction by cleaning up transaction markers and freeing any objects in the trash list via PostPool.

13.4 `rollbackTransaction()`

Undoes all operations back to the last transaction start point by processing the undo stack backward and clearing the trash list. For each operation from the current position back to the transaction marker, it performs the reverse operation based on the `OpType`:

`DELETE_USER` → Call `userManager.createUser(frame.userID, frame.snapshot_username_or_content)`

- Reverses user deletion by recreating the user with original username stored in the snapshot

`CREATE_POST` → Call `userManager.deletePost(frame.userID, frame.postID)`

- Reverses post creation by deleting the post from the specified user

`DELETE_POST` →

- First allocate a new post object
- Restore the post ID and parse snapshot data to reconstruct category and views

- Call `userManager.addPost(frame.userID, newPost)` to restore the post to the user

`FOLLOW` → Call `userManager.unfollow(frame.userID, frame.postID)`

- Reverses a follow operation by unfollowing the target user (note: `frame.postID` likely represents `targetUserID`)

`UNFOLLOW` → Call `userManager.follow(frame.userID, frame.postID)`

- Reverses an unfollow operation by re-establishing the follow relationship

13.5 `record(const OpFrame& f)`

Records an operation frame to the undo stack for potential reversal and clears the redo stack since new operations invalidate redo history.

13.6 `undo()`

Reverses the most recent operation by popping from the undo stack, performing the reverse operation, and adding it to the redo stack if successful.

`DELETE_USER` → Call `userManager.createUser(frame.userID, frame.snapshot_username_or_content)`

- Reverses user deletion by recreating the user with their original username

`CREATE_POST` → Call `userManager.deletePost(frame.userID, frame.postID)`

- Reverses post creation by deleting the post from the user's post list

`DELETE_POST` →

- Allocate a new post from the post pool
- Restore the post ID and reconstruct category/views from snapshot data
- Call `userManager.addPost(frame.userID, newPost)` to restore the post to the user

`FOLLOW` → Call `userManager.unfollow(frame.userID, frame.postID)`

- Reverses a follow operation by removing the follow relationship

`UNFOLLOW` → Call `userManager.follow(frame.userID, frame.postID)`

- Reverses an unfollow operation by re-establishing the follow relationship

13.7 `redo()`

Reapplies a previously undone operation by popping from the redo stack, performing the original operation, and adding it back to the undo stack if successful.

`DELETE_USER` → Call `userManager.deleteUser(frame.userID)`

- Reapplies user deletion

`CREATE_POST` →

- Allocate a new post from the post pool
- Restore the post ID and reconstruct category/views from snapshot data
- Call `userManager.addPost(frame.userID, newPost)` to recreate the post

`DELETE_POST` → Call `userManager.deletePost(frame.userID, frame.postID)`

- Reapplies post deletion

`FOLLOW` → Call `userManager.follow(frame.userID, frame.postID)`

- Reapplies the follow operation

`UNFOLLOW` → Call `userManager.unfollow(frame.userID, frame.postID)`

- Reapplies the unfollow operation

14 Memory Management Guidelines

14.1 Ownership Rules

1. `LinkedList` owns its nodes - Don't delete nodes directly.
2. `User` owns `FollowList` - Deleted in `User` destructor.
3. `PostPool` owns Post objects - Use `allocPost/freePost`.
4. `PostList` owns `PostNode` objects - But references Posts from pool.
5. `UserManager` owns Users - Via `LinkedList`.

14.2 Common Pitfalls to Avoid

1. Double deletion - Don't delete objects owned by other components.
2. Dangling pointers - Always validate pointers before use.
3. Memory leaks - Ensure all new has corresponding delete.
4. Circular references - Be careful with User pointers in `FollowList`.

14.3 Best Practices

1. Always check pointers for `nullptr` before dereferencing.
2. Use RAII principles (Resource Acquisition Is Initialization).
3. Clear pointers after deletion to prevent accidental reuse.
4. Use the provided memory pools instead of direct `new/delete` for Posts.

Remember to run your tests frequently during development and always test edge cases such as empty lists, null pointers, and boundary conditions.

15 Debugging Tips

15.1 Memory Debugging

15.1.1 Segmentation Faults

- Always check if pointers are `nullptr` before dereferencing
- Use `gdb` to find crash location: `gdb ./your_program`, then `run`, and `bt` for backtrace
- Common causes: accessing deleted objects, uninitialized pointers, array bounds violations
- Add debug prints before suspected crash points

15.1.2 Memory Leaks

- Use `valgrind --leak-check=full ./your_program` to detect memory leaks
- Every `new` must have a corresponding `delete`
- Every `new[]` must have a corresponding `delete[]`
- Don't delete objects owned by other classes (e.g., don't delete Posts if PostPool owns them)

16 Testing

Test each component in isolation before integration:

1. **LinkedList:** Test all operations with simple data types.
2. **FollowList/PostList:** Test with mock User/Post objects.
3. **User:** Test construction, destruction, and basic operations.

4. **PostPool:** Test allocation, deallocation, and reuse.
5. **UserManager:** Test user lifecycle and relationship management.
6. **IngestQueue:** Test circular buffer wraparound and batch operations.
7. **UndoRedoManager:** Test transaction semantics.

16.1 Integration Testing

Test component interactions:

- User creation and following relationships.
- Post creation and attachment to users.
- CSV export/import round-trip.
- Memory pool reuse under load.

16.2 Error Condition Testing

- Null pointer handling.
- Duplicate prevention.
- Memory exhaustion scenarios.
- Invalid operation sequences.

16.3 Performance Testing

- Large user lists (10k+ users).
- Bulk post ingestion.
- Memory pool efficiency.
- Following list traversal performance.

16.4 Folder Structure

The project is organized into the following standard directories:

- `include/` - Contains all header files (`.h`).
- `src/` - Contains all C++ source implementation files (`.cpp`).
- `tests/` - Contains the test suites used for verifying functionality.

16.5 How to Compile and Run Tests

Here's a quick guide on how to run the test cases (in case you're feeling a little lost)

All commands must be executed from the root directory of the project (the `PA1` folder).

16.5.1 Unit Tests

This suite tests each component of the system in isolation.

1. **Compile the Test Executable:**

   ```
   g++ -std=c++17 -Iinclude src/*.cpp tests/tests.cpp -o run_unit_tests
   ```

2. **Run the Tests:**

   ```
   ./run_unit_tests
   ```

16.5.2 Integration & Error Condition Tests

This suite tests how different system components interact and also checks various error scenarios.

1. **Compile the Test Executable:**

   ```
   g++ -std=c++17 -Iinclude src/*.cpp tests/integration_tests.cpp
   -o run_integration_tests
   ```

2. **Run the Tests:**

   ```
   ./run_integration_tests
   ```

16.5.3 Stress Tests

This suite tests your code with large lists and bulk initializations, along with code efficiency and performance.

1. **Compile the Test Executable:**

   ```
    g++ -std=c++17 -Iinclude src/*.cpp tests/performance_tests.cpp
   -o run_performance_tests
   ```

2. **Run the Tests:**

   ```
   ./run_performance_tests
   ```