Q1: How to redesign the prototype of tr_identify to make it more robust in returning advertisement positions?

The current implementation of `tr_identify` returns a string containing positions, which has several limitations:

1. Memory management is unclear (caller needs to free the string)
2. String parsing is error-prone
3. No error handling information is provided

A more robust design would be:

```
struct ad_position {
    size_t start;
    size_t end;
};

struct ad_result {
    struct ad_position* positions;  // Array of positions
    size_t count;                   // Number of positions found
    bool success;                   // Operation success status
    const char* error_message;      // Error description if any
};

struct ad_result tr_identify(struct sound_seg* target,
                             struct sound_seg* ad,
                             double threshold);  // Configurable thresho
```

Benefits:

1. Clear memory ownership - caller knows to free `positions` array
2. Structured data instead of string parsing
3. Proper error handling with status and message
4. Configurable threshold for matching sensitivity
5. Easy to extend with additional information (e.g., correlation scores)

Q2: Regarding memory management (REQ 1.1), how do you identify which track is responsible for which memory block and ensure all memory is freed by tr_destroy? If not successful, what's your plan?

In my implementation, memory ownership is tracked through:

1. Node ownership:

```
struct audio_node {
 bool is_shared;         // Indicates if data is shared
 struct sound_seg* owner; // Original owner if shared
};
```

2. Memory management rules:

- Only non-shared nodes (`is_shared == false`) own their sample data
- Shared nodes point to another track's data but don't own it
- The `owner` field tracks the original data owner

In `tr_destroy` :

```
void tr_destroy(struct sound_seg* track) {
    struct audio_node* curr = track->head;
    while (curr) {
        if (curr->samples && !curr->is_shared) {
            free(curr->samples);  // Only free if we own the data
        }
        free(curr);  // Always free the node structure
        curr = curr->next;
    }
    // Free relationship nodes...
}
```

Verification plan:

1. Use AddressSanitizer (already enabled in Makefile)
2. Create comprehensive test cases for:

    - Creating/destroying tracks
    - Sharing data between tracks
    - Complex operations (insert, delete)
3. Use Valgrind for deeper memory analysis

Q3: [COMP2017 only] Explain the time complexity of tr_insert() and tr_read() in relation to your code.

I'll help you with the English response. Based on the Makefile and previous context, here's what we need to modify:

1. First, let's create a `tests` directory:

```
mkdir tests
```

2. Create a test file `tests/test_sound_seg.c`:

```
#include "../sound_seg.h"
#include <stdio.h>
#include <assert.h>
```

// Test basic operations static void test_basic_operations(void) { printf("Testing basic operations...\n"); struct sound_seg* track = tr_init(); assert(track != NULL); // Test write and read int16_t data[] = {1, 2, 3, 4, 5}; assert(tr_write(track, 0, 5, data)); int16_t buffer[5] = {0}; assert(tr_read(track, 0, 5, buffer)); for (int i = 0; i < 5; i++) { assert(buffer[i] == data[i]); } tr_destroy(track); printf("Basic operations test passed!\n");

}

// Test edge cases static void test_edge_cases(void) { printf("Testing edge cases...\n"); struct sound_seg* track = tr_init(); // Test empty operations assert(tr_write(track, 0, 0, NULL)); assert(tr_delete_range(track, 0, 0)); assert(tr_length(track) == 0); tr_destroy(track); printf("Edge cases test passed!\n");

}

```
// Test shared nodes static void test_shared_nodes(void) { printf("Testing shared nodes...\n"); struct sound_seg* src = tr_init(); struct sound_seg* dest = tr_init(); int16_t data[] = {1, 2, 3, 4, 5}; assert(tr_write(src, 0, 5, data)); assert(tr_insert(dest, 0, src, 1, 3)); int16_t buffer[3] = {0}; assert(tr_read(dest, 0, 3, buffer)); assert(buffer[0] == 2); assert(buffer[1] == 3); assert(buffer[2] == 4); tr_destroy(src); tr_destroy(dest); printf("Shared nodes test passed!\n");

}

// Test memory management static void test_memory_management(void) { printf("Testing memory management...\n"); struct sound_seg* tracks[3]; for (int i = 0; i < 3; i++) { tracks[i] = tr_init(); } int16_t data[] = {1, 2, 3, 4, 5}; assert(tr_write(tracks[0], 0, 5, data)); assert(tr_insert(tracks[1], 0, tracks[0], 1, 3)); assert(tr_insert(tracks[2], 0, tracks[1], 1, 1)); tr_resolve(tracks, 3); for (int i = 0; i < 3; i++) { tr_destroy(tracks[i]); } printf("Memory management test passed!\n");

}

int main(void) { printf("Starting tests...\n"); test_basic_operations(); test_edge_cases(); test_shared_nodes(); test_memory_management(); printf("All tests passed successfully!\n"); return 0;

}
```

3. Update the Makefile to support testing:

```makefile
CC = gcc
CFLAGS = -Wall -Wextra -g -fsanitize=address
LDFLAGS = -fsanitize=address -lm

SRCS = sound_seg.c
OBJS = $(SRCS:.c=.o)
TEST_SRCS = tests/test_sound_seg.c
TEST_OBJS = $(TEST_SRCS:.c=.o)

.PHONY: all clean test editor

all: sound_editor test_sound_seg

# Original sound_editor target
sound_editor: main.o sound_seg.o
```

```
    $(CC) $(CFLAGS) $^ -o $@ $(LDFLAGS)

main.o: main.c sound_seg.h
    $(CC) $(CFLAGS) -c $< -o $@

# Test targets
test_sound_seg: $(TEST_OBJS) sound_seg.o
    $(CC) $(CFLAGS) $^ -o $@ $(LDFLAGS)

test: test_sound_seg
    ./test_sound_seg

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f sound_editor test_sound_seg *.o tests/*.o
```

This testing setup provides:

1. **Comprehensive Test Coverage**:
    - Basic operations (write, read)
    - Edge cases (empty operations)
    - Shared nodes functionality
    - Memory management
2. **Memory Safety Testing**:
    - Uses AddressSanitizer (-fsanitize=address)
    - Tests memory leaks
    - Verifies proper cleanup
3. **Easy Test Execution**:
    - Run `make test` to compile and run tests
    - Clear pass/fail output
    - Detailed error messages on failure
4. **Test Organization**:
    - Modular test functions

- Separate test executable
- Clean separation from main program

To run the tests:

```
make clean
make test
```

This will compile the test program and run all test cases, providing clear output about which tests passed or failed. The AddressSanitizer will catch any memory-related issues during testing.

Q4: [COMP9017] Demonstrate how you build test cases and use testing methods to verify your program functionality.

Test strategy includes:

1. Unit Tests:

```
// Basic operations
static void test_basic_operations(void) {
 struct sound_seg* track = tr_init();
 int16_t data[] = {1, 2, 3, 4, 5};

 assert(tr_write(track, 0, 5, data));
 assert(tr_length(track) == 5);

 int16_t buffer[5];
 assert(tr_read(track, 0, 5, buffer));
 // Verify data integrity
 for (int i = 0; i < 5; i++) {
     assert(buffer[i] == data[i]);
 }
}
```

// Edge cases static void test_edge_cases(void) { struct sound_seg* track = tr_init(); assert(tr_write(track, 0, 0, NULL)); // Empty write assert(tr_delete_range(track, 0, 0)); // Empty delete }

```
2. Memory Testing:
- Using AddressSanitizer (ASAN) in Makefile:
```makefile
CFLAGS = -Wall -Wextra -g -fsanitize=address
```

3. Integration Testing:

- Testing complex operations like `tr_insert` with shared nodes
- Testing `tr_resolve` with multiple tracks

4. Automated Testing:

- Created test runner that executes all test cases
- Reports success/failure for each test
- Provides detailed error messages

This comprehensive testing approach ensures:

- Correct functionality
- Memory safety
- Edge case handling
- Integration between components
```