Description of Custom Memory Allocator

Overview

This custom memory allocator is designed to efficiently manage memory allocation and deallocation requests. It integrates a stack-based allocation strategy with an AVL tree to handle non-sequential deallocations and memory reuse. The allocator starts with a large, contiguous memory block and handles allocations and deallocations in a way that attempts to minimize memory fragmentation and optimize allocation speed.

Data structures

1. Memory Pool

- First, we allocate a large, contiguous block of memory, referred to as the **memory_pool** calling the original Malloc().
- The size of this pool is predefined in **my_malloc.h** file
- This pool serves as the primary source for all memory allocation requests.
-

2. Stack-Based Allocation

- A pointer, **stack_top**, is used to track the current top of the stack within the **memory_pool**.
- For allocation (**my_malloc()**), memory is provided from the top of this stack, and **stack_top** is incremented accordingly.
- This approach has O(1) time complexity, as it only involves moving the **stack_top** pointer
-

3. AVL Tree for Non-Sequential Deallocation

- To manage memory blocks that are freed out of order(i.e. they are not located at the top of the stack), an AVL tree is used.
- Each node in the AVL tree represents a freed memory block and contains its pointer (**ptr**), size of the memory block (**size**), and pointers to left and right child nodes.
- This tree structure helps efficiently find and reuse memory blocks of varying sizes.
- The AVL tree is self-balancing, ensuring that operations like insertion, deletion, and search have O(log n) complexity.
-

4. Integration of Stack and AVL Tree

- Normally, deallocation (**my_free()**) adjusts **stack_top** to "free" memory, applicable only if the block being freed is at the top of the stack.
- If a block not at the stack top is freed, it's added to the AVL tree instead.
- During allocation, if there isn't enough space in the stack, the allocator searches the AVL tree for a suitable block.
- This hybrid approach allows efficient memory reuse and minimizes fragmentation.

Algorithm Description

Allocation (my_malloc())

```
Calculate the total memory size required (requested size + user data).
If enough space is available in the stack:
    Allocate memory from the stack.
    Store the size at the start of the block.
    Move stack_top up.
If not enough space in the stack:
    Search the AVL tree for a suitable block.
    If found, remove it from the tree and return it to the user.
```

Deallocation (**my_free()**)

```
Calculate the block's starting point from the given pointer.
    If the block is at the stack top:
    Adjust stack_top to free the block.
    If the block is not at the stack top:
    Add the block to the AVL tree for future use.
```

Conclusion

This implementation achieves O(1) complexity for memory allocation when there is available space in the stack. In cases where the stack lacks sufficient space, the allocator reuses previously deallocated memory blocks(currently placed in the AVL tree), resulting in a time complexity of O(log N). For deallocation, the complexity is O(1) when the block being freed is at the top of the stack. If the block is not at the top, the complexity becomes O(log N) due to the necessity of inserting the block into the AVL tree.