MEMBERS
Moksha Shah
Sneh Phauja
Varsha Gunturu
Vineesha Vuppala

1.Data Structure for Tracking Memory Blocks
Array:
Advantages:

- Simplicity in design.
- Efficient iteration through the array.
- O(1) time complexity for accessing and modifying any index directly.
- O(N) time complexity for scanning the entire array.

In comparison, a linked list may incur O(N) just to reach the starting block when freeing memory, making arrays more efficient for this use case. None of the group members have worked with bitmap before so we decided to use something we knew how to work with.

2. Character Representation:
'a' for allocated blocks.
'f' for free blocks.

3. Functions
Initialise

- The initialize_array function initializes all memory blocks to 'f' (free)

Load

- The function doesn't take any parameters as it is only supposed to load the characters from the text file into the memory.
- We kept the function's return type as int so that if the function works without any error 0 is returned otherwise if there is an error 1 is returned.
- We have declared *storage* which is a file data type pointer that will help in performing file operations.
- The file is opened and closed using the *fopen()* and *fclose()*.
- We have declared another variable *i* to keep track of the memory array index and *ch* to get and store character from the file using *getc()*.
- We have a *while* loop that runs till the end of the file, and at each iteration, the *ch* is inputted into the array and the next character is gotten from the file.
- Then we close the file.

- We have two error conditions: one when there is an error when the file is not opened and when the file is unable to close.
- One assumption we have made here is that the file contains exactly 100 characters of *f* or *a* or both and not an empty file.
- This function takes O(n) as its best and worst case, as the loop needs to run till the total length of characters in the file.
- The space complexity is O(1) as the function uses a constant amount of additional space for both the *memory* array and a file pointer *storage*.

Save

- Return Parameter - int for returning error
- No parameters because not needed
- The purpose of the function is to save the current state of the memory into the text file so it can be loaded next time.
- The file is opened and closed using the *fopen()* and *fclose()*. If it is unable to open or close, it gives an error message.
- If able to open, it stores the memory pool blocks into the file by iterating through the array and putting character after character in the file (with *putc*) till the end of the array.
- This function takes O(n) as its best and worst case, as the loop needs to run till the total length of characters in the file.
- The space complexity is O(1) as the function uses a constant amount of additional space for both the file pointer and loop variables.

Compact

- No parameters and no return type since it only has to defragment what's already there in the memory pool
- Purpose is to make all the allocated blocks contiguous at the start of the array and to ensure there are no free blocks in between the allocated blocks
- Using a swap method
- One pointer at the start of the array
- One pointer at the end
- If free blocks are found at the start of the array they are swapped with allocated ones from the end
- Used this because easily interpretable and if everything is already defragmented, there will only be a simple linear scan of array
- This function takes O(n) as its best and worst case, as the loop needs to run to check the entire array and the space complexity is again O(1).

Free

We have formulated two options for free but we have opted for the first option.

Selected Option: final free algorithm

- Input Parameters: Takes start_address and size as input, which define the range of memory blocks to be freed.

- Return Type - Integer to return if error, 1 for error and 0 for executed
- The function allows partial memory freeing, meaning it continues even if some blocks are already free. This choice was made to avoid unnecessary interruptions and allow the user to free memory in sections, regardless of whether some blocks are already freed.
- It iterates through the specified memory range and frees each block whether it is allocated or already free. This allows the function to continue freeing memory without stopping if a block is already free, ensuring flexibility and smooth execution.
- The time complexity is O(n), where n is the number of blocks in the specified range, since it iterates through the entire range.

Original version Unusable

- Input Parameters: Takes start_address and size as input parameters.
- The function requires that all blocks within the specified range must be allocated before any can be freed. This approach ensures strict control over memory management and prevents partial freeing.
- It first iterates through the specified range to check if all blocks are allocated (memory[i] == 1). If any block is free (memory[i] == 0), the function immediately stops and prints an error message. This prevents partial memory freeing and ensures that memory is only freed when the entire specified range is valid and allocated. Once all blocks are confirmed to be allocated, it proceeds to free them by setting memory[i] = 0.
- The time complexity is O(n), where n is the size of the range being freed. The space complexity is O(1).
- When the loop encounters a free space, it throws an error but the allocated blocks before that could have been freed, before the error. Thus it only does half of the job and hence it has not been used.

Alloc

- Return Type- integer to return error or success
- Input Parameters - size of memory to be allocated

- The function iterates over the array searching for the first space where the number of free blocks is greater than or equal to the size through a counter. If the space has fewer blocks than required, then it resets the counter, and continues searching.
- Once it finds a large enough space, it allocates the required number of free blocks.
- We have chosen the first fit model , even though it can lead to external fragmentation as our memory pool size is quite small, and our compact function can defragment quick enough, so we have opted for the quicker, more efficient first fit instead.

Print Map

- We have used a *for* loop to print the *memory* array with a new line character after every 10 characters so that it would be easier to visualise, interpret and understand the index of the memory array.
- This function takes O(n) time, n being the length of the memory array and the operations such as *if* condition checking and *printf* take O(1) time. Space complexity is O(1) for the loop index *i*.