Authors: Allison Coopersmith & Diana Del Gaudio

Design: Our implementation used an array for memory storage. Our metadata simulation was a simple struct that held two pieces of information: how much memory the user had requested, and how much empty space followed the current piece of data. This meant that every block of empty space in the array was accounted for by whatever piece of data came before it. You can loop through the array by starting at a piece of metadata, adding the metadata size + the user memory size (to get to the end of the current metadata), and then adding the freespace following the metadata (to get to the beginning of the next metadata). We do this both to free blocks of memory, and to look for open blocks of memory to use for mallocing.
This leaves one glaringly obvious corner case: the free space at the beginning of the array does not follow any piece of metadata, and is therefore unaccounted for. To fix this, we used a variable to track how much free space there was at the front of the array (freeSpaceInFront). At the beginning, this is set to 4096, because the array is empty, and therefore freeSpaceInFront encompasses the entire array. Once something is inserted into the array, freeSpaceInFront becomes 0, because there is no longer any room at the beginning of the array. As things are inserted and deleted from the front, this variable is updated and always holds how much freespace there is in the array before the first metadata. Whenever iterating through the metadata stored array, the first position is *always* at the end of freeSpaceInFront.

Results: We included the -O flag in our makefile, and the compiler eventually realized we were only mallocing and freeing things, and optimized the runtime. Our runtimes for each workload were:
Microseconds for workload A: 0.640000
Microseconds for workload B: 10.170000
Microseconds for workload C: 7.250000
Microseconds for workload D: 4.430000
Microseconds for workload E: 37.120000
Microseconds for workload F: 0.010000
We were surprised by two things: that B took so much longer than C and D, when it was only performing 150 operations, and that C took so much longer than D even though they were doing the same thing but with different memory sizes. Other than that, the results were in line with our expectations. F was the least amount of operations (mallocing and freeing one string), and E was the most, so it made sense to us that they would be on those extreme ends of the spectrum.

Space Efficiency: Our metadata is 4 bytes. We were able to implement malloc() and free() using only two numbers (the user's memory size and the free space following the metadata). Since the cache is of size 4096, we were able to use the data type int16_t, which is only two bytes long.