The metadata is storing information about allocated space in the struct entry, that has 4 fields:

- char free - indicates whether memory block is freed
- dataBlock - a pointer to actual block of data
- blockSize - integer that stores the size of data block of the entry
- next - pointer to the next entry.

In mymmaloc, array myblock is initialized when called for the first time, by setting first 4 chars to be equal to certain values.

We use isInitialized function to check if array is initialized already.

First, we check if the required data size is too large , or 0.

If there are no structs in myblock yet, the first struct is created and the pointer is return to its 'dataBlock' field.

If there are structs in myblock, we go through each entry, looking for the struct with dataBlock of at least the required size (including metadata) , which 'free' value is '1'. If such entry exist, we check if there is enough data to split it , or not. If it's possible to split, we create another struct, and insert it into the list . It has the amount of bytes that are left after allocating for requested size (but no smaller than 1). If we can't split the block into two, we simply change free field to '0' and return pointer to entry's dataBlock.

If we reached the end of the list and still did not find the entry with the dataBlock of the right size, we return null if there is not enough memory left after the end of the linked list, or create another entry and append it to the end of the list, returning pointer to its dataBlock.

In myfree, we check if parameter is a pointer. If not, we return an error.

If parameter is a pointer, we go through each entry in the list, and check if its dataBlock entry is equal to that block. If it is , and the 'free' field of the entry is not 1, we freeing the data, setting 'free' field to 0. Otherwise, we return NUll, while also printing an error. We also check if adjacent block are free , and if so, stitch them together.

Memgrind:

By looking at the reaults of first 4 testcases, we can conclude the following:

CASE A: Allocating and immidiately freeing one byte of memory many times takes relatively short amount of time. TestCase A took on average 8.94 microseconds to complete.

CASE B: If we allocate large amount of bytes, and then free them all, it takes significantly longer. However, this could also be due to the fact that we are using an array to store allocated pointers. TestCase B took on average 35.14 microseconds to complete.

CASE C: Randomly choosing between allocating and freeing 1 byte, until total amount of allocations is 50, and then freeing everything, takes slightly longer then testA, but much shorter than testB(probably due to the fact that we store less pointers in the array). TestCase C took on average 3.41 microseconds to complete.

CASE D: Randomly choosing between allocating and freeing from 1 to 64 bytes, until total amount of allocations is 50, and then freeing everything, takes slightly longer than TestC, probably due to the fact that we might have sometimes to break and stitch memory blocks. TestCase D took on average 4.09 microseconds to complete.

CASES E&F are explained in the testcases.txt file. TestCase E took on average 9.60 microseconds to complete. TestCase F took on average 10.16 microseconds to complete.