[mymalloc]:
A program sends the sizeOf value representing the number of bytes which it desires to dynamically allocate to MYMALLOC.C's 'main' method: [mymalloc]. This method checks to make sure the total amount of memory asked for is within the confines of the last 'SBRK' which was called (in this case manually set to (2^9 * 14) = 7168), and creates a root Node if one does not already exist using the [storeMeta] method.

[storeMeta]:
This method is in charge of storing the metadata for a given node. The metadata size is 14 bytes total and stores the following information in order:

- [2 Bytes]: address where the list of the desired memory starts
- [2 Bytes]: total size of the block (node)
- [2 Bytes]: address within the array where the parent of this node has its malloc stored
- [2 Bytes]: left child of this node (malloc array)
- [2 Bytes]: right child of this node (malloc array)
- [2 Bytes]: how much of this block is taken up by METADATA
- [2 Bytes]: how much free space is available in this block

It then starts the list which keeps track of all Nodes in the tree, and sends the start address of the root Node and the size of the memory to be allocated to the recursive method [storeNode].

[storeNode]:
This recursive method first checks to make sure that the Node passed along has space available and returns a 0 if no space is available [base case].
It then checks to see if the given Node does NOT have children [must be true to save data], and if it does not, checks whether or not the data is within the range (space left) <= meta requested <= (space left / 2), saving it in that block if it does. It also makes sure that if it is not within that range, that there is at least enough space to make two additional nodes AND store data within them, otherwise, you will have two Nodes holding only their own METADATA which is pointless. Then, if it does not fit in that range, it creates the two additional children, and accesses them using recursion starting with the RIGHT child first.
Finally, if it already has children, it checks the right child first, and then the left child, return 0 if memory cannot be granted anywhere.

[myfree]:
This is the main method to free data given a pointer to its array. First, the size of the metadata is subtracted from the location to find the METADATA. Then, the method first will free ONLY the array that was allocated for the Node and if it has a parent, send it to the recursive method [recFree] to free and merge nodes.

[recFree]:
This method checks whether or not the parent of the Node destroyed has two children, BOTH of which have arrays with the maximum amount of free space. If it does, it clears the metadata for

the two nodes using the [freeNode] method, updates the parent, and if the node has a parent itself, recursively accesses that node as well.

Throughout the implementation, the metadata and space left are updated when necessary using the recursive [updateMeta] and [updateSpace] respectively, to make sure that every node in the tree is updated.

Given Test Cases

TEST CASE A:

Time elapsed for Workload A: 0.214549
Average for Workload A: 0.002145

TEST CASE B:
Time elapsed for Workload B: 0.497344
Average for Workload B: 0.004973

TEST CASE C:
Time elapsed for Workload C: 0.215478
Average for Workload C: 0.002155

TEST CASE D:
Time elapsed for Workload D: 0.215714
Average for Workload D: 0.002157

Our Test Cases
Description: Refer to testcases.txt

Time elapsed for Workload E: 0.001753
Average for Workload E: 0.000018

Time elapsed for Workload F: 0.005532
Average for Workload F: 0.000055