Memory Allocator Using Buddy System

Introduction:

At the program start, we allocate a block of memory of a configurable size by calling C malloc (we call it initBlock), which is used during the program's runtime as a total available memory. We also use ***Binary Tree as our main data structure*** to keep track of each block's meta-data that was allocated/deallocated by my_malloc() and my_free(). By using Binary Tree, we achieved an ***O(logn) running time for my_malloc() and O(n) running time for my_free()***.

Each node of the binary tree stores meta-data of a particular memory block. The meta-data includes the size of the block (in bytes), the starting memory address of the block, the state of the block (either free, partially free, or full), the pointers to the tree node's children and parent, and a boolean whether the node is a parent. When the binary tree is initialized, a single free node with the size of initBlock is created.

We acknowledge that it would be better to store the meta-data inside initBlock, however, we've discovered that it's quite difficult to do so and it complicates the Buddy System algorithm a lot. Storing the Binary Tree outside of the initBlock is not that critical, since, during the program execution, the binary tree adjusts its size according to malloc/free calls - it shrinks when my_free() is called, and expands when my_malloc() is called. Hence, depending on the number of malloc/free calls, the binary tree should not take much memory.

The logic behind my_malloc()

A tree node is ***FULL*** if its both children are full or if the memory block pointed by the node was allocated, ***PARTIAL*** if one of its children is FULL, or ***FREE*** if it's a leaf that wasn't allocated. We start at the root of the binary tree and compare the requested memory size with the size of the current node.

- If the requested size is smaller than half of the size of the current node (n/2):
    - if the current node's state is FREE, we split the current node into two nodes of size n/2 and recurse to the right child. If the current node's state is PARTIAL (it's been already split), we just recurse to the right child. We recurse until we find a node such that half of its size is less than the requested size or until we reach a full node.
    - If a node that can satisfy the requested size was not found in the right subtree, we recurse to the left subtree and repeat the logic
- When we recurse, if half of the node's size is less than the requested size, it means we've found a block that keeps internal fragmentation to a minimum. Hence, we allocate the block by marking it to be full and exiting from recursion
- When we split a node, the memory address of the block into where the right child points to is set to the parent's memory address, whereas the left child's memory address equals the parent's address + parent's size/2. Hence, we are effectively splitting our

block of memory into two partitions and making two new nodes point to the two blocks of memory of equal size.

We acknowledge that recursively partitioning the memory block into blocks of half-size might cause a little bit of internal fragmentation, e.g. a request of size 30 will be put into a partition of size 32 if initBlock is of size 256. We also acknowledge that by using partitioning we also cause some external fragmentation. However, we've tried to keep fragmentation as minimal as possible and chosen efficiency over having no internal/external fragmentation.

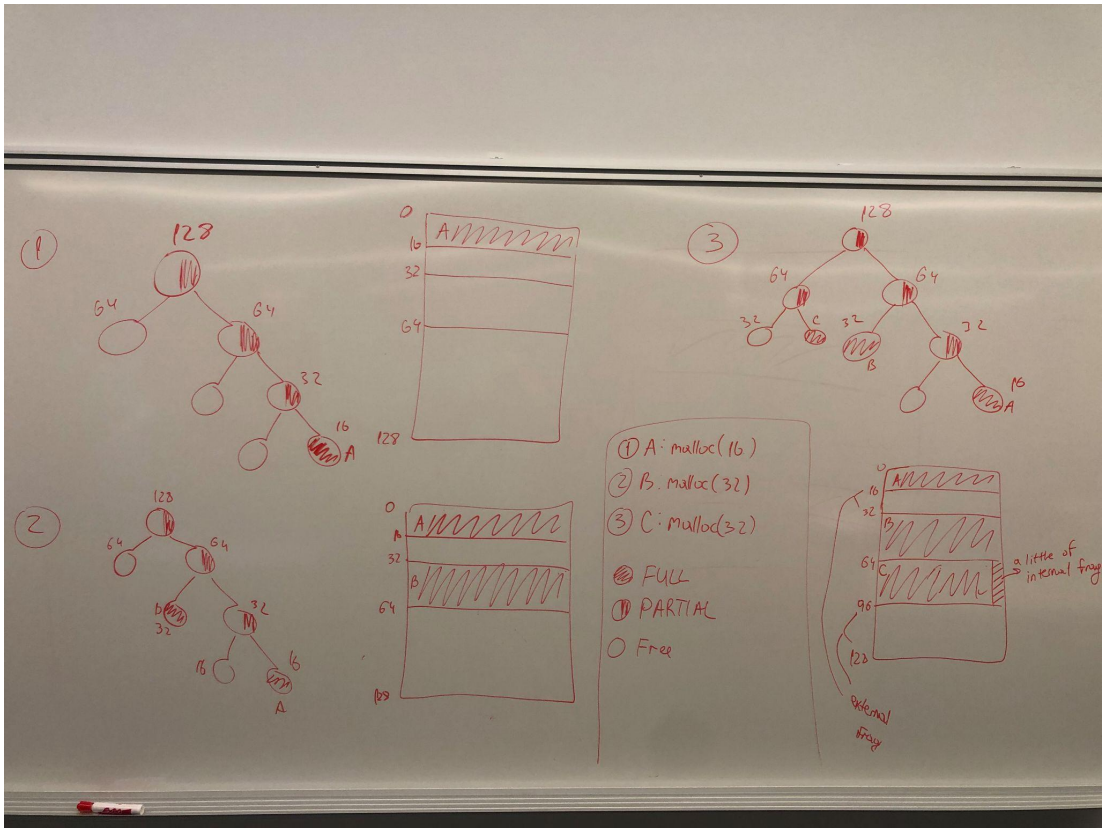
The logic behind my_free()

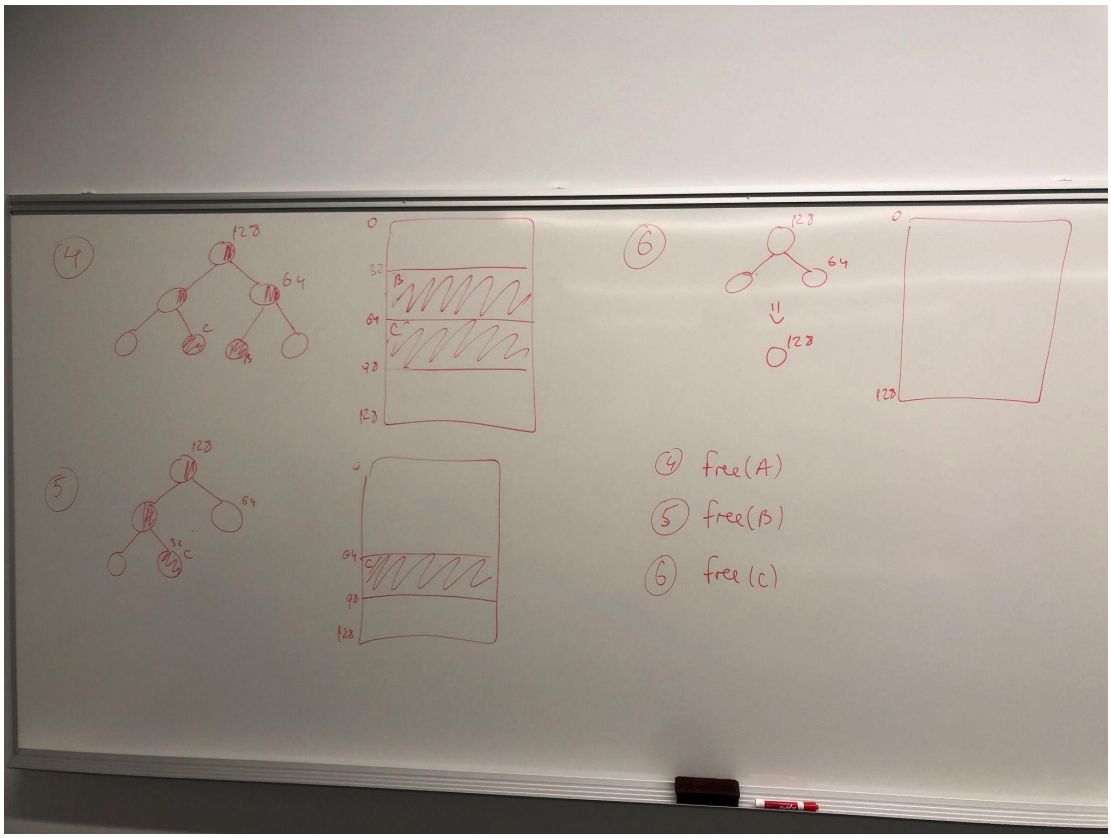
After we malloc a particular block of memory, we add a tree node that points to the block into a list. When we free, we search this list for a tree node that stores the pointer which was passed into my_free() call. Once we've found the tree node, we look at its sibling (so-called buddy). If the buddy is FREE, we merge the current node with its sibling to one node, i.e. we remove the two from the tree and update their parent to be FREE.

See an example run of the algorithm on the next page.

References:

Inspiration for the Buddy System is taken from https://www.youtube.com/watch?v=1pCC6pPAtio


1
128
64
64
32
16
A
0
16
32
64
128
A
3
128
64
64
32
C
32
B
32
16
A
2
128
64
64
32
B
32
16
16
A
0
A
32
B
64
128
① A: malloc(16)
② B. malloc(32)
③ C: malloc(32)
FULL
PARTIAL
Free
0
16
32
B
64
C
96
128
a little of internal frag
external frag


4
128
64
C
B
0
32
B
64
C
96
128
6
128
64
128
0
128
5
128
64
32
C
0
64
C
96
128
④ free(A)
⑤ free(B)
⑥ free(C)