CMPT 300 Assignment 3

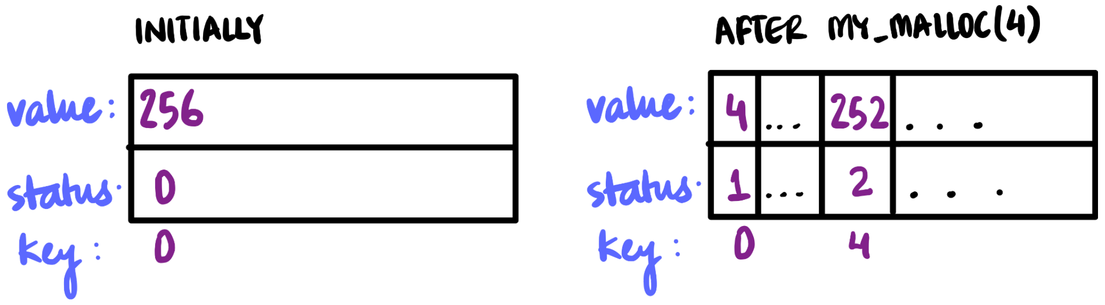
(due Monday, November 20 2023)

Submitted By:

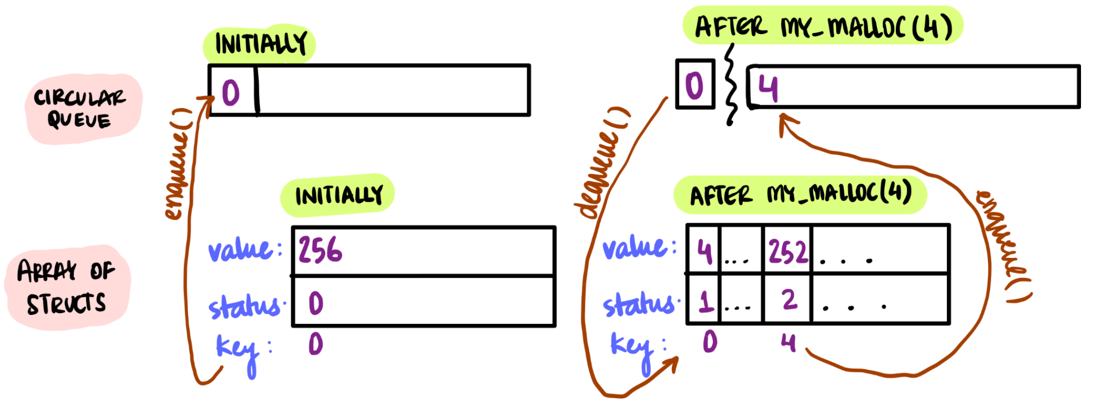
Rebecca Reedel (301454910), Asmita Srivastava (301436340)

For this assignment we had to implement malloc and free in C. In designing the solution for the assignment we took into consideration problems with dynamic memory allocation such as fragmentation, and optimization related concerns regarding the implementation of my_malloc() and my_free().

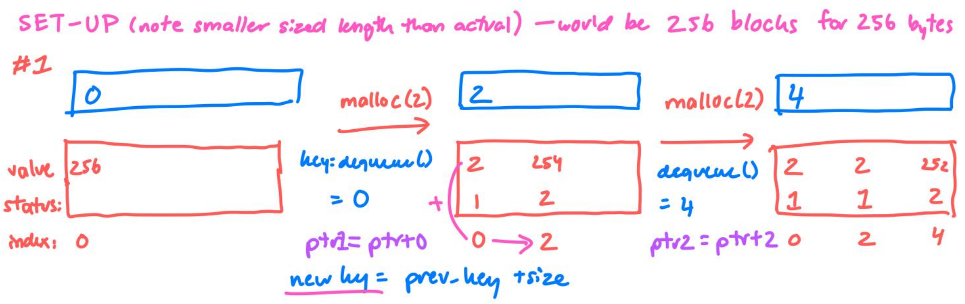
Array of Structs: To implement my_malloc and my_free, we imagined a hashmap-array like data structure, using an array of structs to organise the bytes allocated, where the struct has two fields, a value, containing the available memory bytes in the chunk after allocation, and a status which is used to determine whether the block is free/busy. This data structure is indexed by the key (index) which is holding the number of bytes since the starter pointer. The reasoning behind this choice was in hopes that by being able to index easily to get desired values instead of using linked-list pointers that our solution for malloc would be closer to **Ω(1)** time with an unlikely **O(n)** for the N values of the queue.



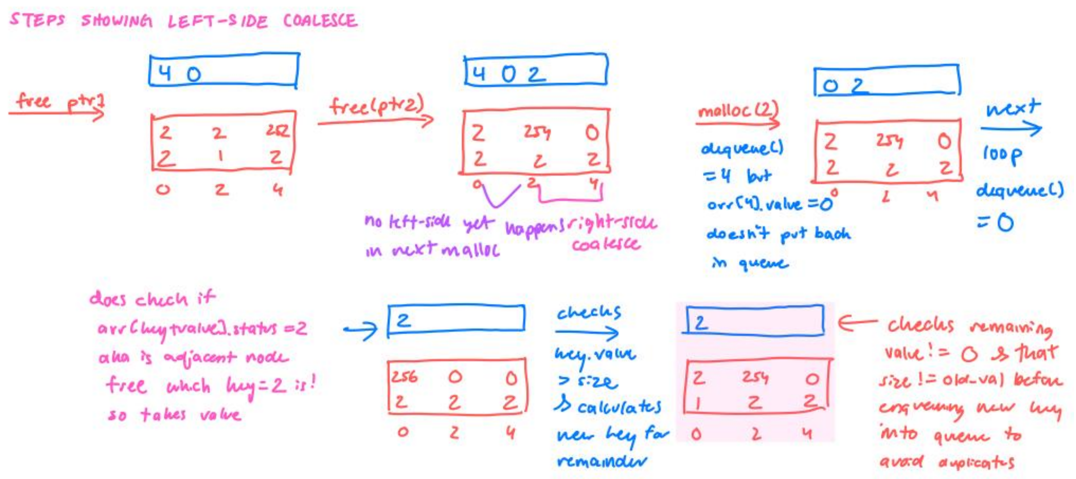
Circular Queue: Another data structure that aids the solution is a circular queue that stores the index from the array containing any available chunks (i.e., where status == 2 and value > 0). The queue functions by removing and adding in First In First Out order to accommodate user's my_malloc requests. This Queue helps speed up the search time for free chunks during allocation.



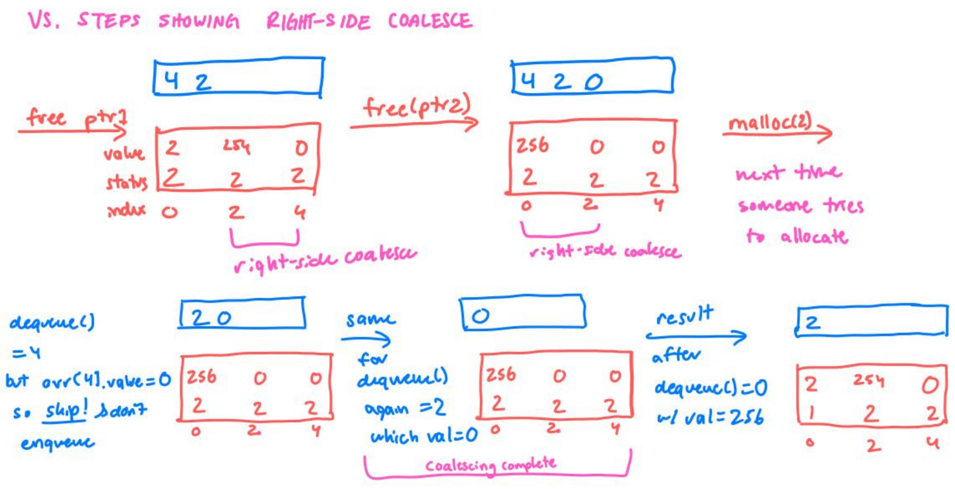
My_Malloc Process: When a user asks for memory worth the size using my_malloc(size), the program dequeues the first key from the circular queue and checks to see if the value at the key in the array of structs has available memory >= size requested by the user, if not enough size it will enqueue the key and repeat the process of enqueuing and dequeuing until it finds a bigger enough chunk or goes through all options. Hence, after the block is allocated, it is returned as a pointer to the user (ptr+key) where ptr is the base pointer of the contiguous block of memory. Moreover, a new key = key+size is formed if there is any more space after the requested size and this new_key goes to enqueue in the circular queue for future allocation.


SET-UP (note smaller sized length than actual) — would be 256 blocks for 256 bytes
#1
malloc(2)
malloc(2)
value
status:
index: 0
key: dequeue() = 0
ptr2 = ptr+0
new key = prev-key + size
dequeue() = 4
ptr2 = ptr+2

My_Free Process: When a user asks to free a chunk of memory using my_free(free_ptr), the program traces the ptr to the key in the array of structs using: key = free_ptr - ptr; (where ptr is the starter pointer of contiguous block of memory), and then using this key the function extracts the value acquired by the block in **O(1)** time, and resets the status of the chunk to free (arr[key].status = 2).


STEPS SHOWING LEFT-SIDE COALESCE
free ptr1
free(ptr2)
no left-side yet in next malloc
happens right-side coalesce
malloc(2)
dequeue() = 4 but arr[4].value = 0 doesn't put back in queue
next loop dequeue() = 0
does check if arr[key+value].status = 2 aka is adjacent node free which key = 2 is! so takes value
checks key.value > size & calculates new key for remainder
checks remaining value != 0 & that size != old-val before enqueuing new key into queue to avoid duplicates

Dealing with Fragmentation: Our solution aims to minimise fragmentation as much as possible by having coalescing constraints to ensure that two adjacent free blocks will be combined either during memory allocation or when the block is freed. We define coalescing as checking if the next block header after the current header is free and if so allocating the memory to the first one and setting the second to a value of 0; As well since we don't use fixed block sizes, we avoid internal fragmentation.


VS. STEPS SHOWING RIGHT-SIDE COALESCE
free ptr1
value
status
index
right-side coalesce
free(ptr2)
right-side coalesce
malloc(2)
next time someone tries to allocate
dequeue() = 4 but arr[4].value = 0 so skip! & don't enqueue
same for dequeue() again = 2 which val = 0
coalescing complete
result after dequeue() = 0 w/ val = 256

Reducing Memory Leaks: In the end of our program, we made sure to free the ptr we were using to provide the block of memory to implement my_malloc and my_free. The ptr was freed using free(ptr); and later the program was checked using valgrind to ensure there weren't any memory leaks.