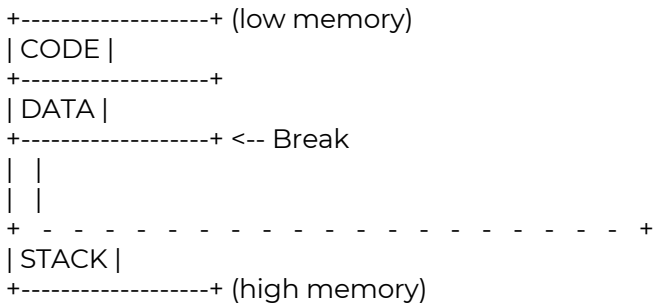
1 Assignment

In this assignment, you will be writing the dynamic memory allocation and deallocation functions of malloc, free, realloc, and calloc. These functions are confusing to write, so we have provided an in-depth guide below. Please read through this entire pdf before beginning. The specifics for each function are located in malloc.c as well as subsections 1.6 - 1.9 below.

1.1 The Basics

It is the job of the memory allocator to process and satisfy the memory requests of the user. But where does the allocator get its memory? Let us recall the structure of a program's memory footprint.



When a program is loaded into memory there are various "segments" created for different purposes: code, stack, data, etc. In order to create some dynamic memory space, otherwise known as the heap, it is possible to move the "break", which is the first address after the end of the process's uninitialized data segment. A function called brk() is provided to set this address to a different value. There is also a function called sbrk() which moves the break by some amount specified as a parameter.

For simplicity, a wrapper for the system call sbrk() has been provided for you in the file named my sbrk.c.

Make sure to use this call rather than a real call to sbrk, as doing this can potentially cause a lot of problems. Note that any problems introduced by calling the real sbrk will not be regraded, so make sure that everything is correct before turning in.

Ifyouglanceatthecodefor ,youwillquicklynoticethatuponthefirstcallitalwaysallocates8

KiB. For the purposes of your program, you should treat the returned amount as whatever you requested. Forinstance,thefirsttimeIcall itwillbedonelikethis:

```
my sbrk()

my sbrk()

my_sbrk(SBRK_SIZE); /* SBRK_SIZE == 2 KB */

------------------------------------------
| 8KB |
------------------------------------------
^
|
\_____ The pointer returned to me by my_sbrk
```

Even though you have a full 8 KiB, you should treat it as if you were only returned SBRK_SIZE bytes. Now when you run out of memory and need more heap space you will need to call my sbrk() again. Once again, the call is simply:

```
my_sbrk(SBRK_SIZE);

----------------------------------------
|2KB| 6KB |
----------------------------------------
^
|
        \____ The pointer returned to me by my_sbrk
```

Notice how it returned a pointer to the address after the end of the 2 KB I had requested the first time. my sbrk() remembers the end of the data segment you request each time and is able to return that value to you as the beginning of the new data segment on a following call. Keep this in mind as you write the assignment!

1.2 Block Allocation

Trying to use sbrk() (or brk()) exclusively to provide dynamic memory allocation to your program would be very difficult and inefficient. Calling sbrk involves a certain amount of system overhead, and we would prefer not to have to call it every single time a small amount of memory is required. In addition, deallocation would be a problem. Say we allocated several 100 byte chunks of memory and then decided we were done with the first. Where would the break be? There's no handy function to move the break back, so how could we reuse that first 100 byte chunk?

What we need are a set of functions that manage a pool of memory allowing us to allocate and deallocate efficiently. Typically, such schemes start out with no free memory at all. The first time the user requests memory, the allocator will call sbrk() as discussed above to obtain a relatively large chunk of memory. The user will be given a block with as much free space as they requested, and if there is any memory left over it will be managed by placing information about it in a data structure where information about all such free blocks is kept. This is called the freelist and we will return to this later.

In order to keep track of allocated blocks we will create a structure to store the information we need to know about a block. Where should we put this structure? Can we simply call malloc() to allocate space for the information?

No we can't! We're writing malloc(); we can't use it or we'd end up with infinite recursion. However, there's an easier way that will keep our bookkeeping structure right with the data we're allocating for easy access.

In order to keep track of allocated blocks, we will create a structure to store the information we need to know about a block. We will store this information about the block, called metadata, inside the block itself! A crucial part of the metadata is the canary. Canaries are integers that we generate via information about the block itself. They buffer the user data, so if the canary is incorrect, the user data has been altered. For more information about canaries see https://en.wikipedia.org/wiki/Buffer_overflow_protection#Canaries, but note that the canary we implement will be a one for memory allocated by malloc, not static arrays.

Metadata(containsbeg.canary) UserData EndCanary		

Figure 1. The beginning and end canaries buffer the area for user data, creating a 'block'

Whenever you malloc, you will set both of the beginning and end canaries. Since the canaries are random numbers used for verification purposes, we will calculate them by xor'ing the address of the block with CANARY MAGIC NUMBER and adding 1 for fun.

```
unsigned long canary = ((uintptr_t)block ^ CANARY_MAGIC_NUMBER) + 1;
```

We will need to take into consideration the leading metadata and end canary whenever we allocate blocks. To let the user have as much space as they requested, when they request a block of size n bytes we will allocate a block of size sizeof(the metadata) + n + sizeof(tail canary). Along with the beginning canary, this size will be stored in the metadata. As well, the metadata will contain four pieces of information that is critical for the freelist discussed in the next section. As depicted in my malloc.h, this is the struct definition for the metadata:

```
typedef struct metadata {
        struct metadata *prev_addr;
        struct metadata *next_addr;
         struct metadata *prev_size;
         struct metadata *next_size;
unsigned long size;
unsigned long canary;
} metadata_t;
```

The size portion of the metadata struct contains the size that the user requested and the TOTAL_METADATA SIZE, a macro holding the size in bytes of the metadata and end canary which will be described in detail in the next section. For ease of reading, this macro will be represented as TMS in all of our block representations. The user does not care about the metadata for the block, they just want the size they requested. Therefore, when you return a block to the user, you will need to use pointer arithmetic to 'step over' the metadata and return the address of the data. What this looks like:

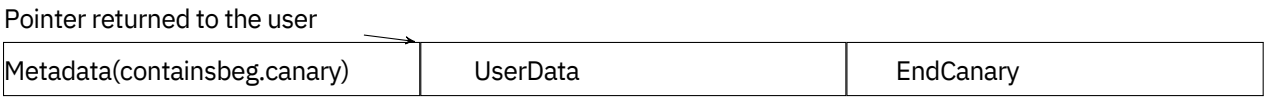


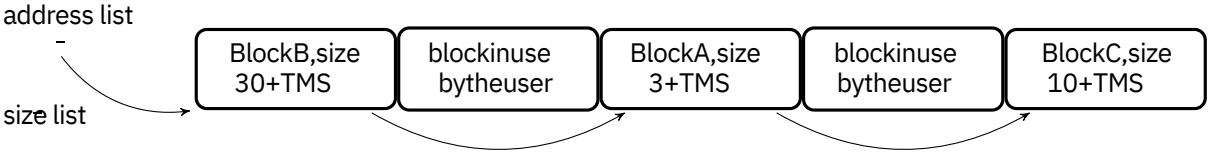
Figure 2. When a block is returned to the user, the pointer returned points to the address of the area used by the user

1.3 The Freelist

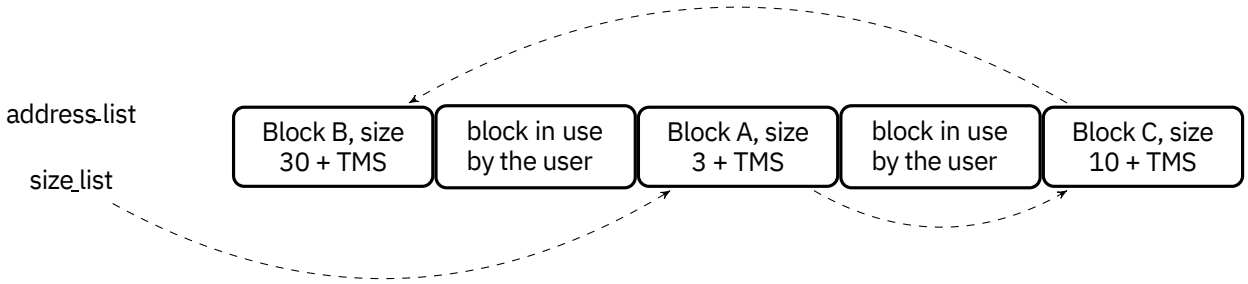
When we split up memory, we give one piece/block to the user. The remaining pieces/blocks are placed in a linked list, called the freelist, to be used at a later time. For this semester, we are representing our freelist as two separate doubly linked lists, one organized by address, the other organized by size. Both of these linked lists should be defined as global file variables and to help you out, we have already defined them for you.

```
metadata_t *size_list;
metadata_t *address_list;
```

This may be a source of some confusion, as we have two linked lists for the representation of our freelist. In reality, each block will be placed two linked lists, one ascending in address:

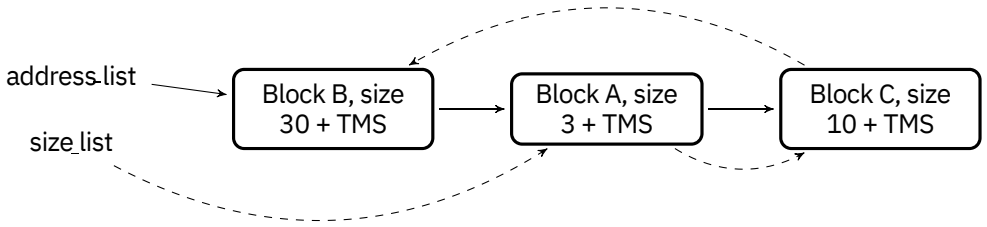


The other ascending by size:



Note: If two blocks are the same size, place the newer block before the older block in the size list.

For the remainder of the pdf, we will represent the freelist without spaces for the blocks currently in use by the user like so:



Both the address_list and size_list are doubly linked, which is why each block has both a previous and next pointer for both of the lists. Whenever you remove or add to the lists, make sure to update all four of the pointers present in the metadata.

1.4 Simple Linked List: Allocating

When we first allocate space for the heap, it is in our best interest not to just request what we need immediately but rather to get a sizable amount of space, use a piece of it now, and keep the rest around in the freelist until we need it. This reduces the amount of times we need to call sbrk(), the real version of which, as we discussed earlier, involves significant system overhead. So how do we know how much to allocate, how much to give to the user, and how much to keep?

For this assignment we will request blocks of size 2048 bytes from my sbrk(). We don't want to waste space, though, so we want to give to the user the smallest size block in which their request would fit. For example, the user may request 256 bytes of space. It is tempting to give them a block that is 256 bytes, but remember we are also storing the metadata inside the block. If our metadata and canaries takes up sizeof (metadata_t) + sizeof (int) = 20 bytes for example, we need at least a

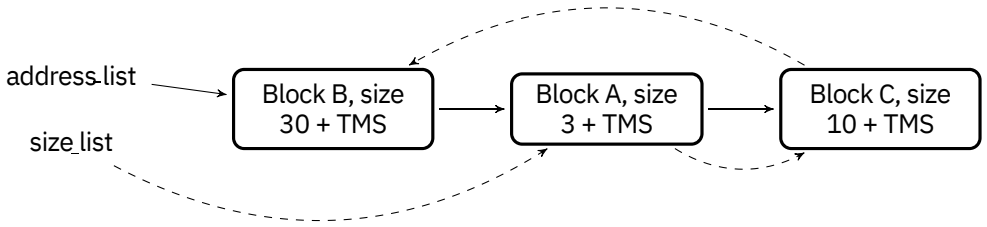
$$256 + 20 = 276$$

byte block.

Note that the size of your metadata will vary based on your computer's architecture and platform. Use sizeof() to avoid depending on the platform, and the macro TOTAL_METADATA_SIZE that sums the beginning metadata and end canary so you don't have to worry about it.

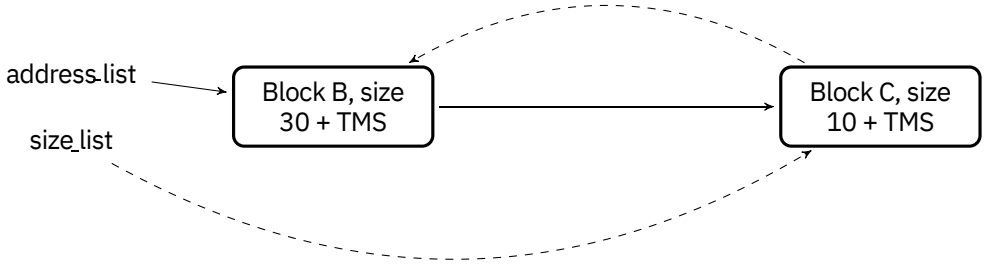
How do we get from one big free block of size 2048 bytes to the block of size 276 bytes we want to give to the user? In this simple implementation, you will traverse the size list to find the best block to satisfy the user's request, which should be equal or greater than the size requested, and "split" off however much you need from the front or the back. For this assignment, you must split off from the back.

Say we have the following situation:



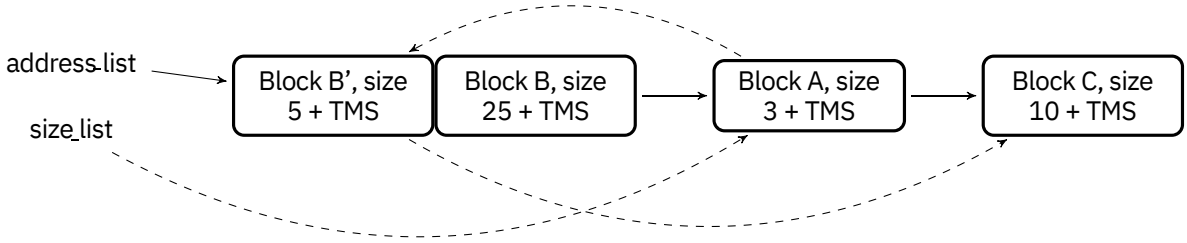
When we malloc for a certain size, we first want to use a block of that exact size, remove it from both the address list and size list and return it to the user.

Ex: malloc(3) would leave the freelist as so:



If we do not have a perfectly sized block, then find the next block that is big enough to split. i.e. A block that is big enough for the size of the malloc call + TMS with room for another block, MIN BLOCK SIZE. (In our case, MIN BLOCK SIZE is defined to be 1 byte + TMS)

Ex: my malloc(25) would split block B into two blocks B(size 25) and B'(size 5). Remember to split your block from the back, in which the left portion of the block will remain in the freelist.



Don't forget to set both canaries and move the pointer to the beginning of the space the user uses after the end of the metadata before returning the block to the user.

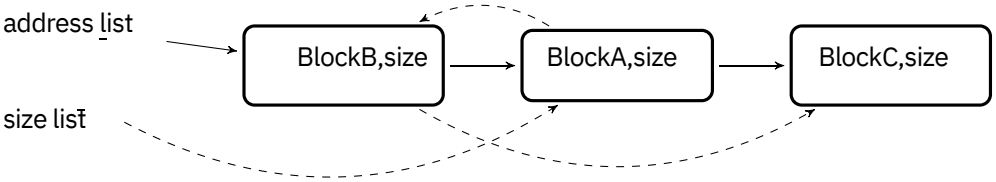
1.5 Simple Linked List: Deallocating

When we deallocate memory, we simply check the block's canaries and return the block to the address list and size list in the appropriate position. When the user calls the free function with a block body pointer, we do some pointer arithmetic to find the starting point of the entire block (i.e. the metadata). Notice we don't clear out all the data. That really just takes too long when we're not supposed to care about what's in memory after we free it anyway. For all of you who were wondering why sometimes you can still access data in a dynamically allocated block even after you call free on its pointer, this is why! We like the freelists to contain fairly large blocks so that large requests can be allocated quickly, so if the block on either side of the block we're freeing is also free, we can coalesce them, or join them into the bigger block like they were before we split them.

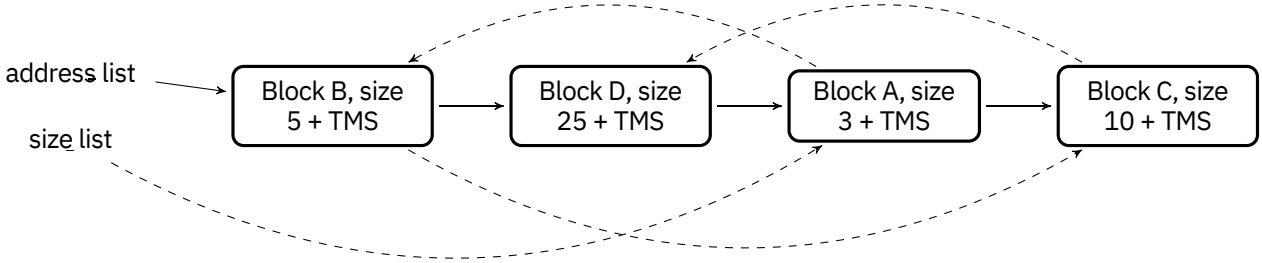
How do we know what blocks we can join with? The left side one will have its address + its size = your block's address, and the right one will be your block's size + it's address.

To deallocate blocks, we would first iterate through the address list for the correct location of the block. If the block could be merged with a curr block to the right or left, we would remove the curr block from the size list, combine the blocks and renter it into the size list. If the block could not be merged, we would insert it in the appropriate positions in both the size list and address list. The following examples demonstrate a few of the possibilities with deallocation.

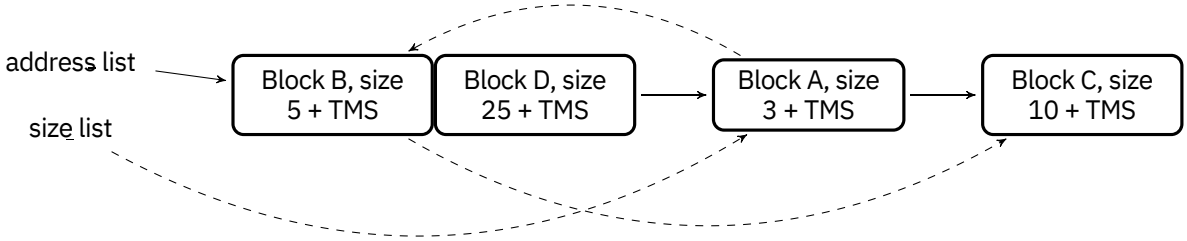
Let's start with this situation:

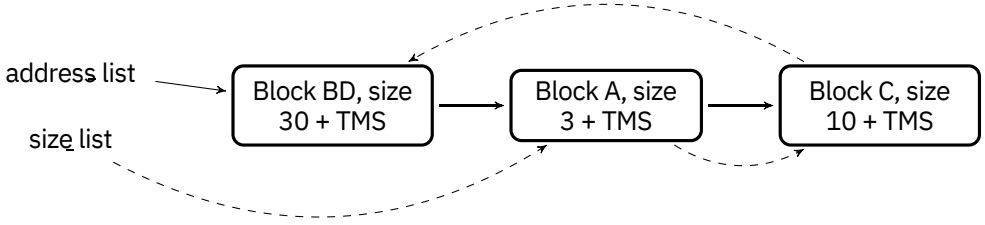


If we deallocated a block of size 25, we would first iterate through the address list for the correct location of the block and check to see if the block needs to be merged either to the right of left. In this example, the block to be entered is not directly next to any other blocks, address wise, so we would just insert it into the address list. Finally, we would insert the block in the correct position in the size list leaving the freelist as seen below (assuming Block D's address places it in between Blocks B and A)



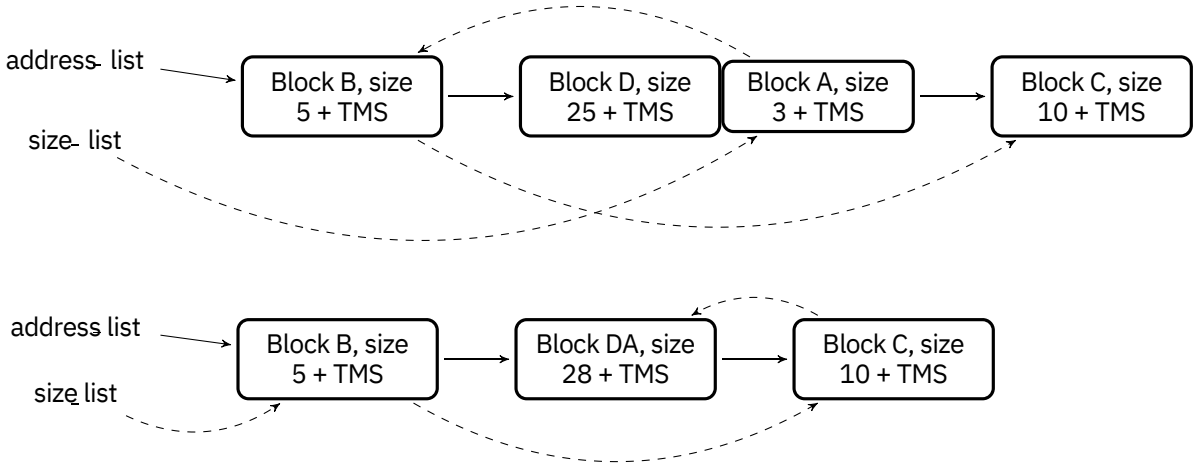
If Block B and D were right next to eachother in memory (i.e. the address at end of block B is equal to the address at the beginning of block D), then we would need to perform a merge. To perform this left merge, pop block B from the size list , add block D to it, reset the size and canaries, and find the new block its' proper home in the size list (Note: there is a way to perform a left merge without removing and inserting blocks from the address list).





If Block D and A were right next to eachother in memory (i.e. the address at the end of block D is equal

to the address at the beginning of block A), then we would need to perform a merge. To perform this right merge, pop block A from the size_list, add block D to it, move block A's metadata to block D, reset the size and canaries, and find the new block its' proper home in the size list.



1.6 mymalloc()

You are to write your own version of malloc that implements simple linked-list based allocation:

1. Figure out what size block you need to satisfy the user's request by adding the TOTAL_METADATA_SIZE to the requested block body size to include the size of the metadata and the tail canary, that will be the real block size we need. (Note: if this size in bytes is over SBRK SIZE, set the error SINGLE_REQUEST_TOO_LARGE and return NULL. If the request size is 0, then mark NO ERROR and return NULL).

2. Now that we have the size we care about, we need to iterate through our freelist to find a block that best fits. Best fit is defined as a block that is exactly the same size, or the smallest block big enough to split and house a new block (MIN BLOCK SIZE is defined for you). If the block is not big enough to split, it is not a valid block and cannot be used.

   (a) If the block is exactly the same size, you can simply remove it from the both the address list and size list, set the canaries, and return a pointer to the body of the block.

   (b) If the block is big enough to house a new block, we need to split off the portion we will use. Remember: pointer arithmetic can be tricky, make sure you are casting to a uint8_t * before adding the size (in bytes) to find the split pointer!

   (c) If no suitable blocks are found at all, then call my sbrk() with SBRK SIZE to get more memory. You must use this macro; failure to do so will result in a lower grade. After setting up its metadata and merging it if possible (in this assignment, there must never be two different blocks in the freelist who are directly adjacent in memory), go through steps (a)-(c). In the event that my sbrk() returns failure (by returning NULL), you should set the error code OUT OF MEMORY and return NULL.

   Remember that you want the address you return to be at the start of the block body, not the metadata. This is sizeof (metadata_t) bytes away from the metadata pointer. Since pointer arithmetic is in multiples of the sizeof the data type, you can just add 1 to a pointer of type metadata_t* pointing to the metadata to get a pointer to the body. If you have not specifically set the error code during this operation, set the error code to NO ERROR before returning.

3. The first call to my malloc() should call my sbrk(). Note that malloc should call my sbrk() when it doesn't have a block to satisfy the user's request anyway, so this isn't a special case.

1.7 myfree()

You are also to write your own version of free that implements deallocation. This means:

1. Calculate the proper address of the block to be freed, keeping in mind that the pointer passed to any call of my free() is a pointer to the block body and not to the block's metadata.
2. Check the canaries of the block, starting with the head canary (so that if it is wrong you don't try to use corrupted metadata to find the tail canary) to make sure they are still their original value. If the canary has been corrupted, set the CANARY CORRUPTED error and return.
3. Attempt to merge the block with blocks that are consecutive in address space with it if those blocks are free. That is, try to merge with the block to its left and its right in memory if they are in the freelist. Finally, place the resulting block in both the address list and size list.

Just like the free() in the C standard library, if the pointer is NULL, no operation should be performed.

1.8 my realloc()

You are to write your own version of realloc that will use your my malloc() and my free() functions. my realloc() should accept two parameters, void *ptr and size t size. If the block's canaries are valid, it will attempt to effectively change the size of the memory block pointed to by ptr to size bytes, and return a pointer to the beginning of the new memory block. If the canaries are invalid, it returns NULL and sets my malloc errno to CANARY CORRUPTED.

Do not directly change the freelist or blocks in my realloc() — leave that to my malloc() and my free(). This means you don't need to worry about shrinking or extending blocks in place[1]; if size is nonzero, just always call my malloc() to attempt to allocate a new block of the new size. Make sure to copy as much data as will fit in the new block from the old block to the new block. The rest of the data in the new block (if any) should be uninitialized.

Your my realloc() implementation must have the same features as the realloc() function in the standard library. For details on what realloc() does and edge cases involved in its implementation, read the realloc manual page by opening a terminal in Ubuntu and typing man realloc.

If my malloc() returns NULL, do not set any error codes (as my malloc will have taken care of that) and just return NULL directly.

1.9 mycalloc()

You are to write your own version of calloc that will use your my malloc() function. my calloc() should accept two parameters, size t nmemb and size t size. It will allocate a region of memory for nmemb number of elements, each of size size, zero out the entire block, and return a pointer to that block.

If my malloc() returns NULL, do not set any error codes (as my malloc() will have taken care of that) and just return NULL directly.

1.10 Error Codes

For this assignment, you will also need to handle cases where users of your malloc do improper things with their code. For instance, if a user asks for 12 gigabytes of memory, this will clearly be too much for your 8 kilobyte heap. It is important to let the user know what they are doing wrong. This is where the enum in

[1] Even though we don't extend or shrink blocks in place in this homework, keep in mind that real-world implementations (which are not written in a panic right before finals) very well could.

the my_malloc.h comes into play. You will see the four types of error codes for this assignment listed inside of it. They are as follows:

- NO_ERROR: set whenever my_calloc(), my_malloc(), my_realloc(), and my_free() complete successfully.
- OUT_OF_MEMORY : set whenever the user's request cannot be met because there's not enough heap space.
- SINGLE_REQUEST_TOO_LARGE : set whenever the user's requested size plus the total metadata size is beyond SBRK_SIZE.
- CANARY_CORRUPTED: set whenever either canary is corrupted in a block passed to free() or realloc().

Inside the .h file, you will see a variable of type enum my_malloc_err called my_malloc_errno. Whenever any of the cases above occur, you are to set this variable to the appropriate type of error. You may be wondering what happens if a single request is too large AND it causes malloc to run out of memory. In this case, we will let the SINGLE_REQUEST_TOO_LARGE take precedence over OUT_OF_MEMORY. So in the case of a request of 9kb, which is clearly beyond our biggest block and total heap size, we set ERRNO to SINGLE_REQUEST_TOO_LARGE.

1.11 UsingtheMakefile

Before running the Makefile, you need to install Check, a C unit testing library the provided tests use. The following command should install the packages you need for this homework:

```
sudo apt-get install pkg-config check gdb
```

You can run the provided tests with `make run-tests` and run gdb with `make run-gdb`.

1.12 Deliverables

Submit only my_malloc.c to GRADESCOPE under "Homework 11." Please don't zip it.

Do NOT modify or submit the header file, my_malloc.h. We will grade with the original copy. Any functions or variables you add should be marked static so they do not conflict with the grader.

Also, please note that the tests are not weighted, so the grade you get in your terminal will NOT be the grade you get on this assignment. You can submit to Gradescope to get a better idea of that, but we reserve the right to add test cases later.

1.13 Debugging

Yes, we assigned malloc
which makes us pretty cruel.
But here are some debugging tips
because we are actually kind of cool

When you run the tests, you will see a pretty hefty output in your terminal. Each line of the output provides critical information depicting which tests you are failing/passing. The general format of:

```
suite_filename.c:420:fun_test_case:test_description
```

states a test named test_description is failing/passing in an individual test case named fun_test_case, located in that specific test suite suite_filename.c at line 420. That is, test suites contain test cases which contain tests. For example,

```
malloc_suite.c:37:Malloc Perf Block1:test malloc perf block1 lists
```

tells us whether the address list and size list is correct when we malloc for a perfectly sized block. More information about the test is written in malloc suite.c, and the assertion that failed is on line 37.

To run an individual test case, run

```
make run-tests TEST=Malloc_Perf_BLock1
```

To debug an individual test case with gdb, run

```
make run-gdb TEST=Malloc_Perf_BLock1
```