Exercise 10

You're familiar with the malloc()/free() family of functions from the C standard library. These functions get memory from the operating system in big blocks, carve it up to into smaller regions as memory is requested and reorganize memory into large, available blocks as it is freed. The standard library has to manage a data structure to do this, and it has to build that structure out of the same region of memory it's using to satisfy memory requests. We're going to try this ourselves.

In the files myMalloc.h and myMalloc.c, I'm giving you a partial implementation of our own malloc(), free(), calloc() and realloc(). Most of this implementation is done for you. It's not nearly as clever as a real heap allocation subsystem would be, but it will be enough to let us see how this kind of thing could work, and why errors in dynamic memory allocation can be particularly tricky to debug. You just have to complete the implementations of two functions, getBlock(), which selects and removes a node from the free list, and returnBlock() which puts a node back on the free list, potentially merging it with neighboring blocks.

Our implementation keeps free memory organized as a single, big, linked list, linking together all the unused blocks of memory and ordered them by memory address (so, blocks with lower memory addresses will be earlier on the list). Part of the memory in each free block is used to build this list. The first few bytes of each free block are used as an instance of the **`Hole`** structure. The following shows how Hole is defined. It contains a size field containing the size of the hole and a next pointer pointing to the next node on the free list.

```
typedef struct HoleStruct {
  size_t size;

  char mem[ 0 ];

  struct HoleStruct *next;
} Hole;
```

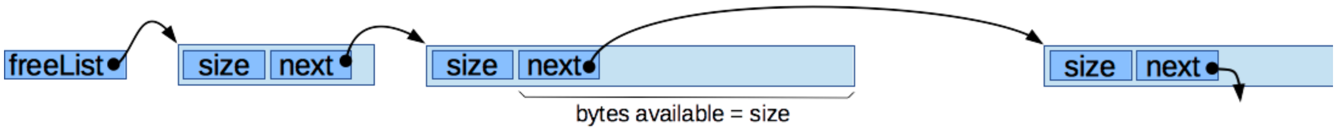
I'm the size of the hole, in bytes.

I don't take up any space in the struct.

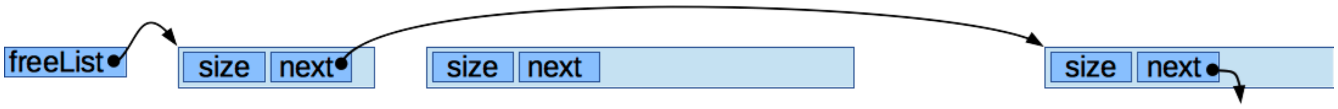
I'm a pointer to the start of the next hole.

This struct should take 16 bytes on an EOS Linux machine. A free block of memory might be larger than that. That's OK. We use the first part of the block as an instance of Hole and the rest of the block is unused, until it is given out via a call to malloc.

The size measures the number of bytes in the hole, counting the size of the next pointer but not the size of the size field.

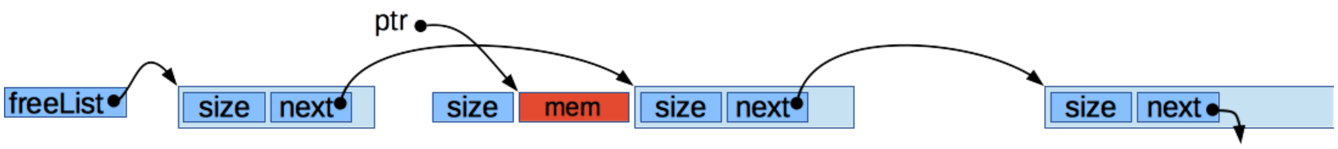


When our implementation of malloc() needs to allocate a block, it calls the getBlock() function. The job of this function is to walk down the linked list of free blocks and unlink and return the first one that's big enough to satisfy the request.



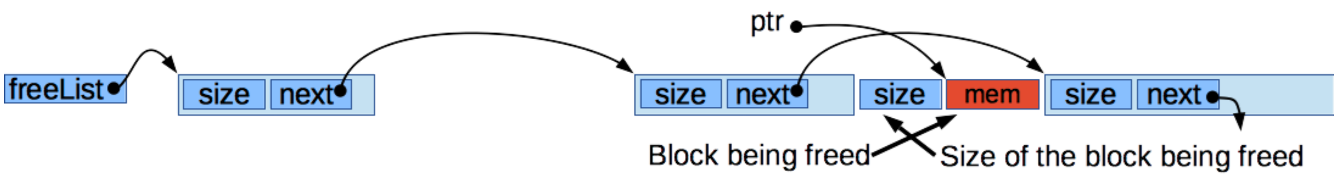
If there's no sufficiently large block (initially, there will be no blocks at all), getBlock() should return NULL. When getBlock() returns NULL, our malloc() will go to the operating system to ask for more memory. It will use part of that memory to satisfy the request and use returnBlock() to put the rest of the memory on the freeList.

The block returned by getBlock() may be larger than the current request. If there's enough extra space, it may be possible to give away just the first part of the block and re-insert the remaining portion as a new node on the freeList.

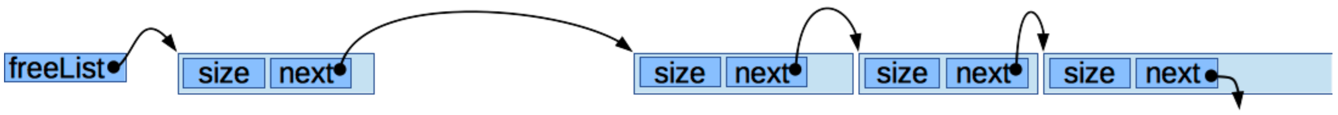


If the block isn’t large enough to make use of the left-over part, we just return the whole block, an example of *internal* fragmentation. The malloc() function takes care of splitting the block if possible, and it uses returnBlock() to put any extra memory back on the free list.

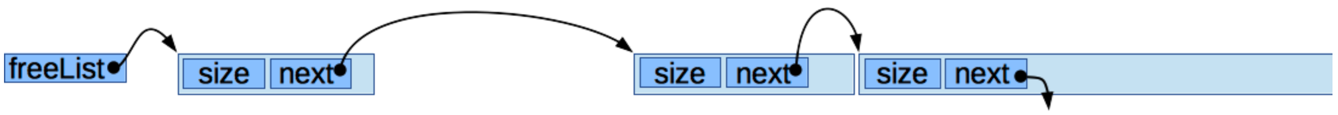
When malloc() gives away a pointer to a block of memory, it returns the address right after the size field inside the Hole structure (the orange block in the picture above). The contents of the size field are left in memory, right before the address malloc() returns. This is important for when the block is eventually freed. The free() function can back up in memory and look in this size field to figure out how much memory is being returned to the free list. It then calls returnBlock() to re-link the freed memory back into the right spot in the free list (the block in the figure below is supposed to be a different block from the one allocated above).



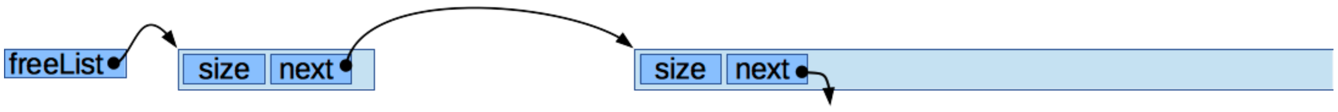
The returnBlock() function has two jobs. It has to re-link the freed block back into the right spot in the free list, with blocks sorted by memory address. I’ve already written this part of the function for you.



Then, if there’s no gap between the freed block and its predecessor and/or successor blocks on the freeList, returnBlock() needs to merge blocks to into a single, larger contiguous block. I’ve already written part of this code for you. After a block is linked back into the free list, I check to see if there's no gap between the freed block and its successor. If the freed block is adjacent to its successor, I merge them into a single, large block.



The same thing needs to happen with the predecessor. If there's a predecessor block and there's no gap between it and the freed block, they should be merged into a single, large block. You get to add the code to do this.



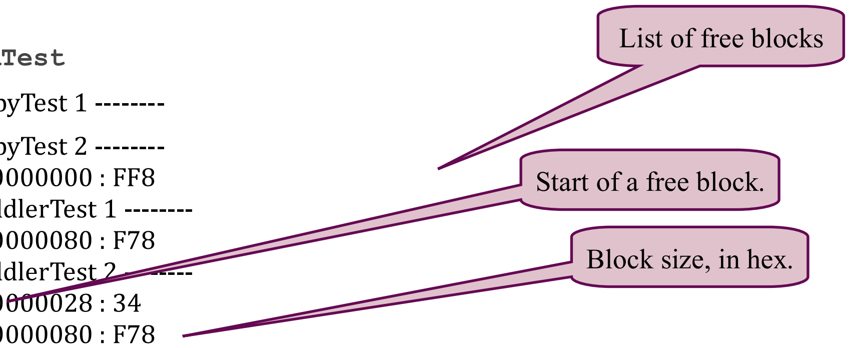
Be aware that once you start using your own malloc()/free(), your program will be using it for **everything**, including memory requests by code you didn't write (e.g., if printf() needs to allocate some memory). You may want to debug this code using gdb, instead of depending on printouts in your code. The printf() function may not work if malloc() is broken, and, even if it does, you don't want it to try to allocate memory while you're debugging your malloc() implementation.

To help you try out your substitute heap allocation subsystem, I've written a test program, memTest.c. This program uses the reportFreeList() function, a debug function that lets us see all the holes on the free list, to make sure the malloc implementation is working correctly. This program first tries out some simple allocate/free operations where it's easy to predict what the free list should look like. Then, it does a series of random malloc()/free() requests to try to find other, less obvious bugs. You should be able to build this test driver with your malloc implementation using a command like:

```
$ gcc -Wall -std=c99 -g -D_DEFAULT_SOURCE memTest.c myMalloc.c -o memTest
```

When you run this test, you should get output like the following. The first free list report is empty; there shouldn't be any blocks on the free list part-way through the first test (babyTest). At the end of babyTest, all the free memory should be in one block on the free list, exactly the size of a page, minus the size of the block's size field (8 bytes).

```
$ ./memTest
-------- babyTest 1 --------
-------- babyTest 2 --------
0x600000000000 : FF8
-------- toddlerTest 1 --------
0x600000000080 : F78
-------- toddlerTest 2 --------
0x600000000028 : 34
0x600000000080 : F78
-------- toddlerTest 3 --------
0x600000000000 : 5C
```



```
0x600000000080 : F78
-------- toddlerTest 4 --------
0x600000000000 : FF8
-------- randomTest --------
0x600000000000 : 188FF8
```

As you're debugging your implementation, have a look at the tests in the memTest.c program. They're fairly simple, and understanding what each test is doing will help you to figure out what's going wrong. You could even create a few simple test cases yourself. Debugging with a tiny little test case could be a lot easier than trying to debug the randomTest at the end.

Once your replacement malloc is working, submit just the myMalloc.c file to the Gradescope assignment named EX10. This should be the only file you need to change.