Douglas Rudolph
Brandon Yu
Systems Programming
Dr. Fransisco
Assignment 1 – *malloc*

Design:
In my malloc project I split the project into two different sections, the malloc section and the free section:

Within the malloc process I split I start off the process seeing if I initialized cleaned the data in my char * array of size 5000. I also name this array heap. Now, every time someone mallocs for information, I check to see if there is any data in the heap. They way I check for data is by traversing through specificed sections of memory that strictly store if there is data written to an address as well as the size of the memory the user requested. If there is not enough room in the heap. I return an error message saying that there is not enough room in the heap for the user.

As for the freeing section to my project, I split this process into two sub functions. When the user sends in a memory address, I loop through the heap to see if any memory address shares the one the user is trying to free. If there is a memory address that aligns with the one the user is requesting, I then call a function named purge_heap(). Within the purge heap function, I start off by freeing the node the user wanted to free. Following this, I also loop through the rest of the memory nodes starting from the requested memory address and join together any adjacent nodes that were previously freed. Overall, this process defragments any blocks of data that aren't used.

Data:

Test Case A:
- Run Time: 0.030
- Findings: I found that because of the way malloc creates adjacent nodes, and the way I delete them, this test case runs very efficiently.

Test Case B:
- Run Time: 0.0025s
- Findings: I found that when I run Test Case B, mallocing and freeing 1 pointer on and off is extremely efficient because I never trigger a recursive call while freeing.

Test Case C:
- Run Time: 0.02ms
- Findings: In Test Case C, I found it to be unpredictable on time, but generally, I saw the listed run time on average. The reason this happens is because I only malloc when the number randomly generated is divisible by three, and free when the number randomly generated is divisible by two. A number divisible by two is more common than a number divisible by three, therefore not a lot of time was spent freeing the pointer.

Test Case D:
- Run Time: 0.0013ms
- Findings: For test case D, I did the inverse of Test Case C. I only malloc when the number randomly generated is divisible by two, and free when the number randomly generated is divisible by three. A

number divisible by 2 is more common than a number divisible by 3, therefore, this test case spent a lot more time mallocing then it did freeing. As you can see, the run time suffered a little from having to mallloc more often, but ultimately was still pretty quick.

Personal Test Case E:
- Run Time: 13.020000ms
- Findings: I found that this test case ran the longest because I purposely trigger the most inefficient recursive case within *purge_heap()*. Doing this 100 times causes for there to be a delay within each iteration.

Personal Test Case F:
- Run Time: 0.013ms
- Findings: For Test Case F, I run a script similar to Test Case E, but without triggering the recursive step twice. Seeing that the function ran in a much slower case in comparison to Test Case E, I suspect that deleting the middle reference like I do in Test Case E causes for there to be several extra recursive iterations.