1- Code Structure:

My code uses structures similar to those in the book. I have a header_t that contains the size and a magic number and is used to refer to newly allocated memory spaces. I have a node_t that contains size and a next pointer and is used to represent a free list allowing for accurate tracking of freed memory regions that can be malloced.

There are four main functions implemented. psumalloc() and psufree() are the core functions that take care of the bulk of the work. Psumeminit is used to initialize the original mmap mapping and the structures. There is a function called split() that allows correct allocation of memory requested from free memory. Another important function is coalesce() that takes care of contiguous areas of free space that have to be merged to avoid external fragmentation. We finally have a psumemdump() function that was used a couple of times in the process of development for testing purposes (displaying the list). However, it does not hold any functional purpose.

I use a simple linked list instead of other structures such as an array and a balanced tree. Let's discuss. An array provides excellent access, insertion and deletion time. However, an array is resource-heavy and compromises space for efficiency. Since I am told not to use one, I did not. A balanced tree (AVL, Red-black) could have been used. It provides a search runtime of O(log n) and is relatively memory friendly. However, a balanced tree is very hard to implement in C with the lack of libraries compared to other languages. It is also less readable compared to other simpler data structures. And so, I chose to use a normal linked-list, which is memory-friendly and has an good O(n) access time. It is readable and is referenced to in the book, making it the data structure of choice.

```
typedef struct __header_t
{
        int size;
        int magic;
} header_t;

typedef struct __node_t
{
        int size;
        struct __node_t* next;
} node_t;
```

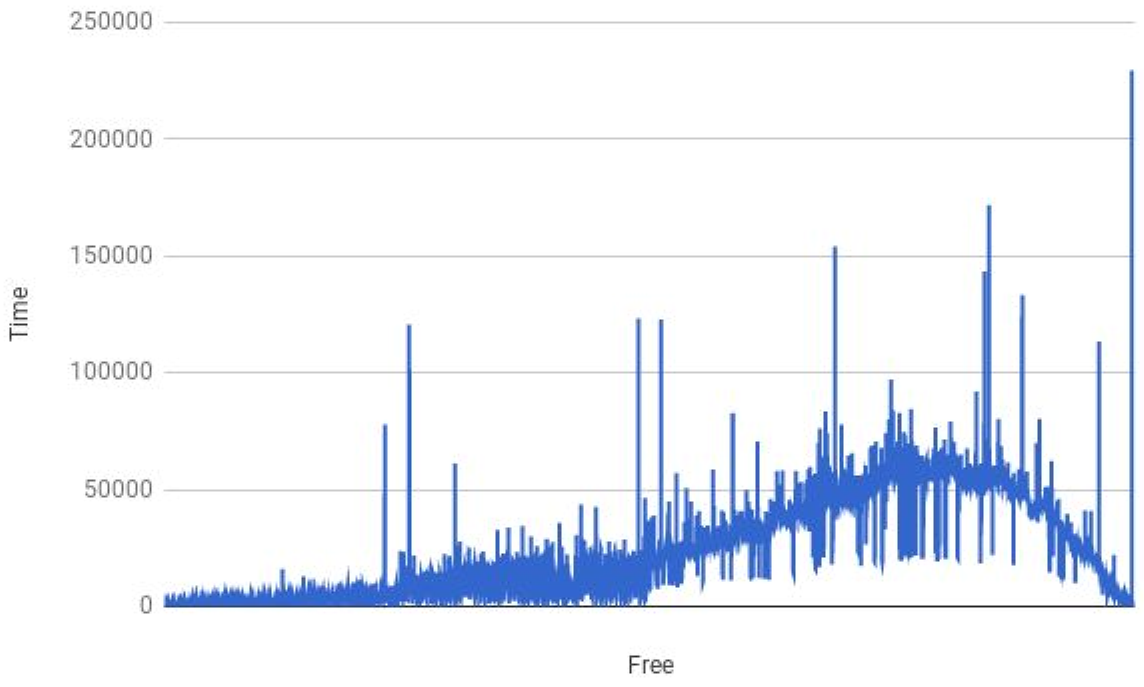
For testing purposes, I use the FILE structure for I/O and use clock_gettime() to measure time differences. I used google sheets to process the data and draw the graphs.

2- Measurements:

a) Small workload using best fit (0):

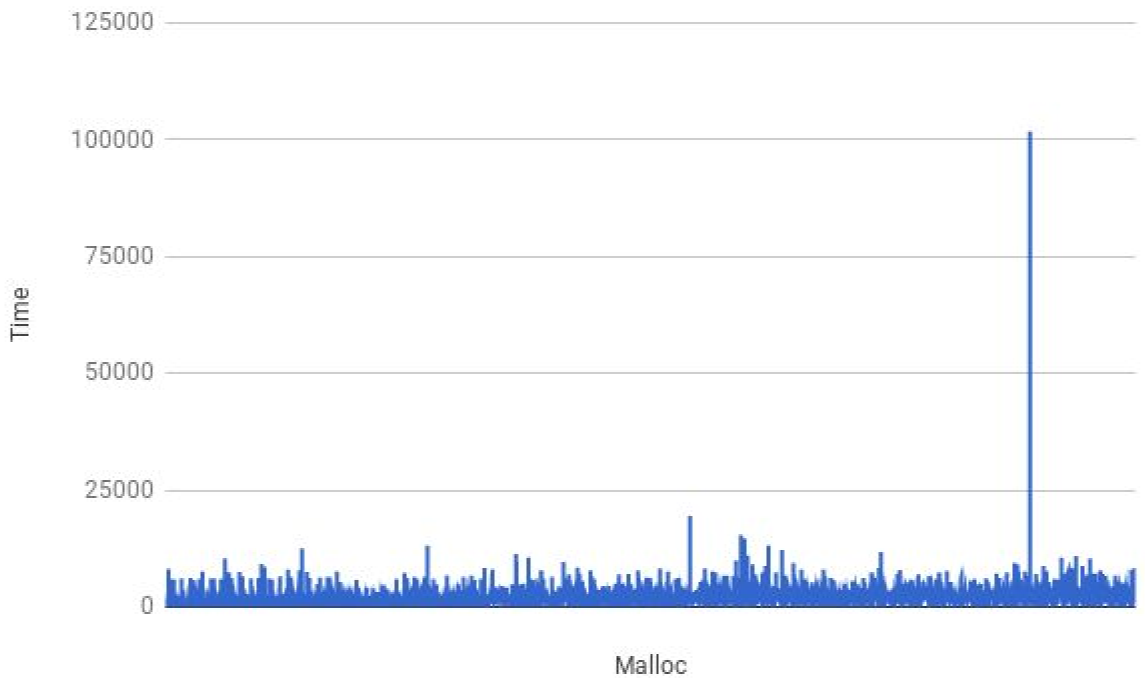
- Free:

22473.1683	Average
17251	Median
4121	25th percentile
38972	75th percentile



- <u>Malloc:</u>

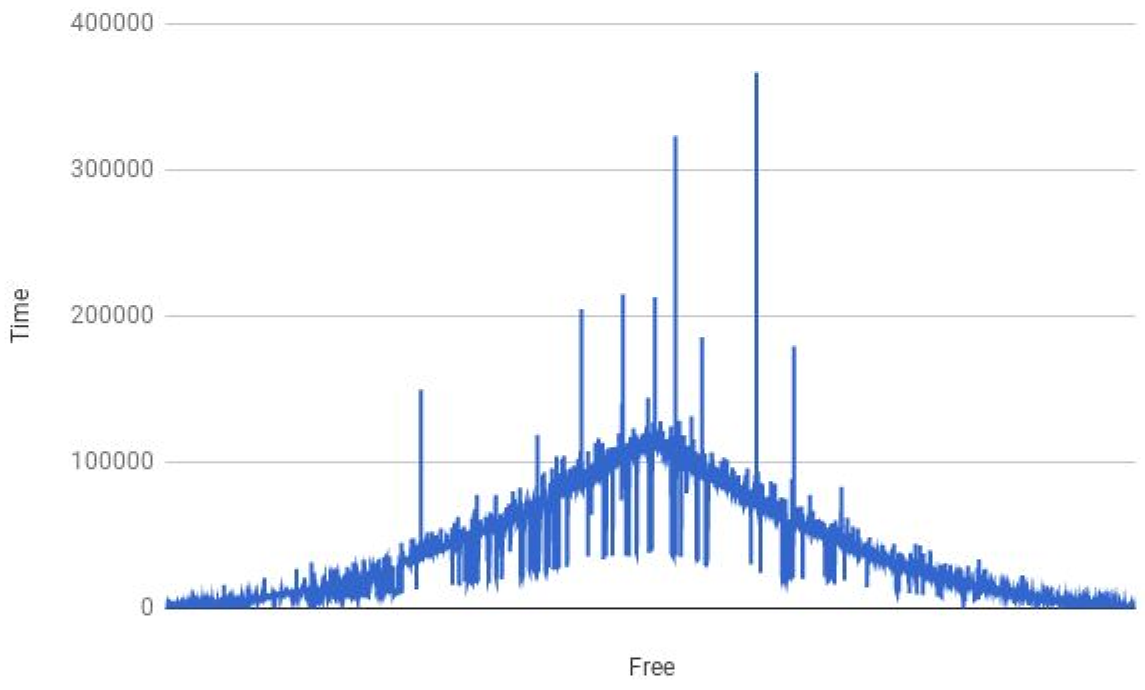
1729.9856	Average
1466	Median
978	25th percentile
2095	75th percentile



b) Small workload using worst fit (1):

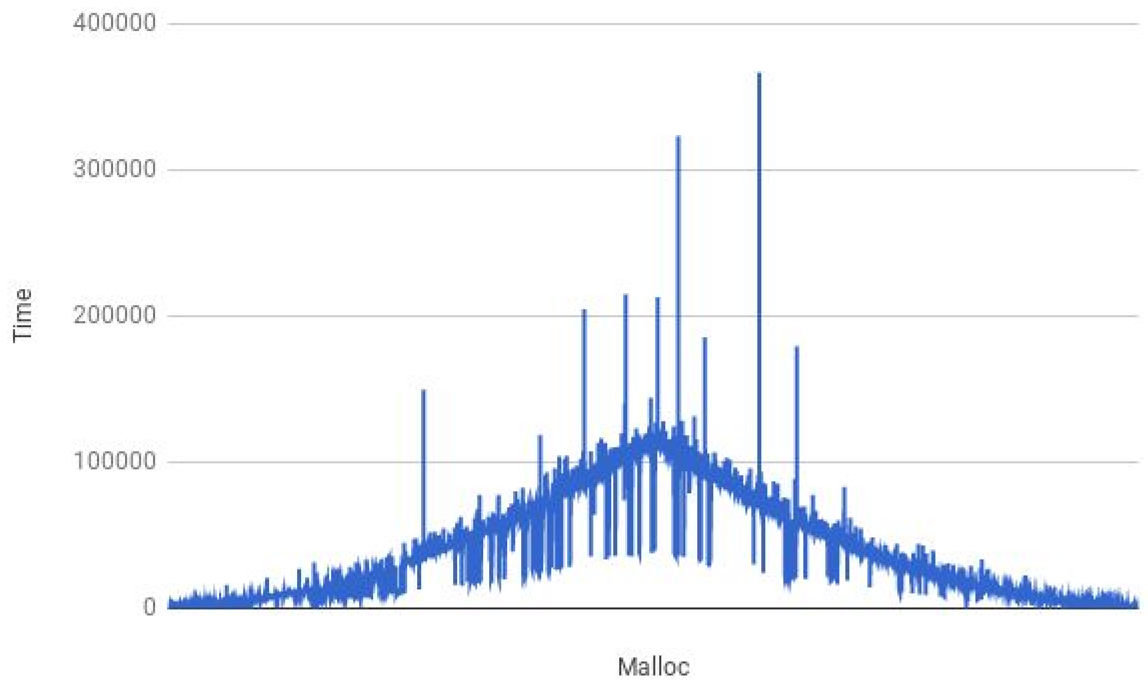
- Free:

38345.5452	Average
28705	Median
8451	25th percentile
62875.5	75th percentile



- Malloc:

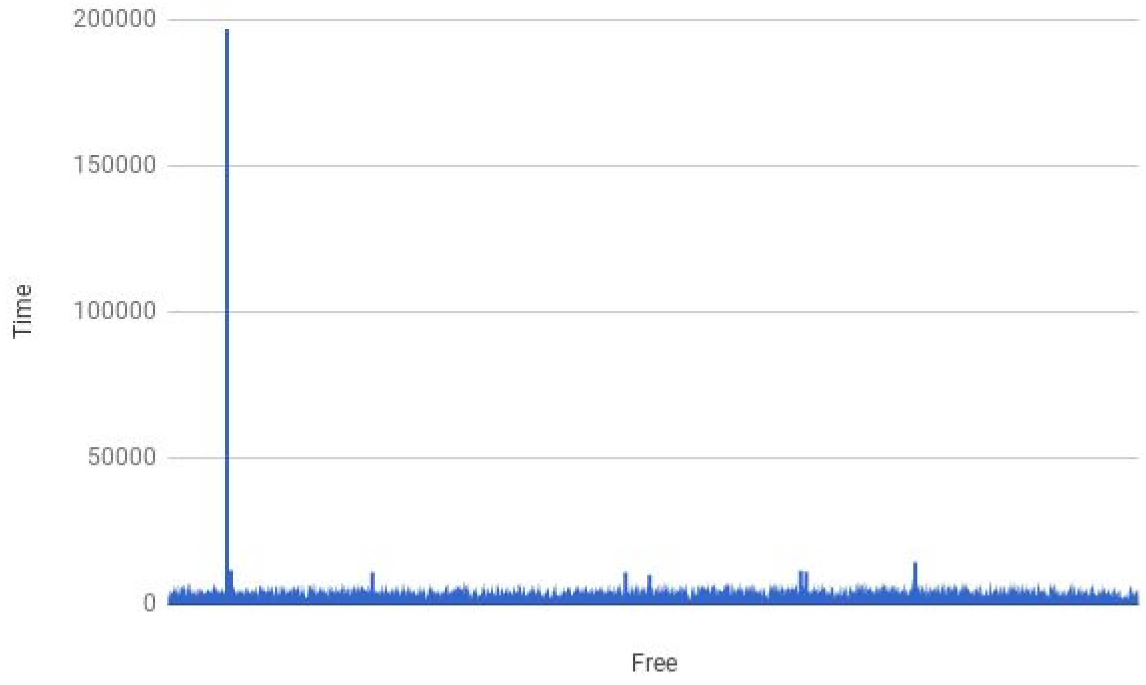
38345.7453	Average
28705	Median
8451	25th percentile
62875.5	75th percentile



c) <u>Mixed workload using best fit (0):</u>

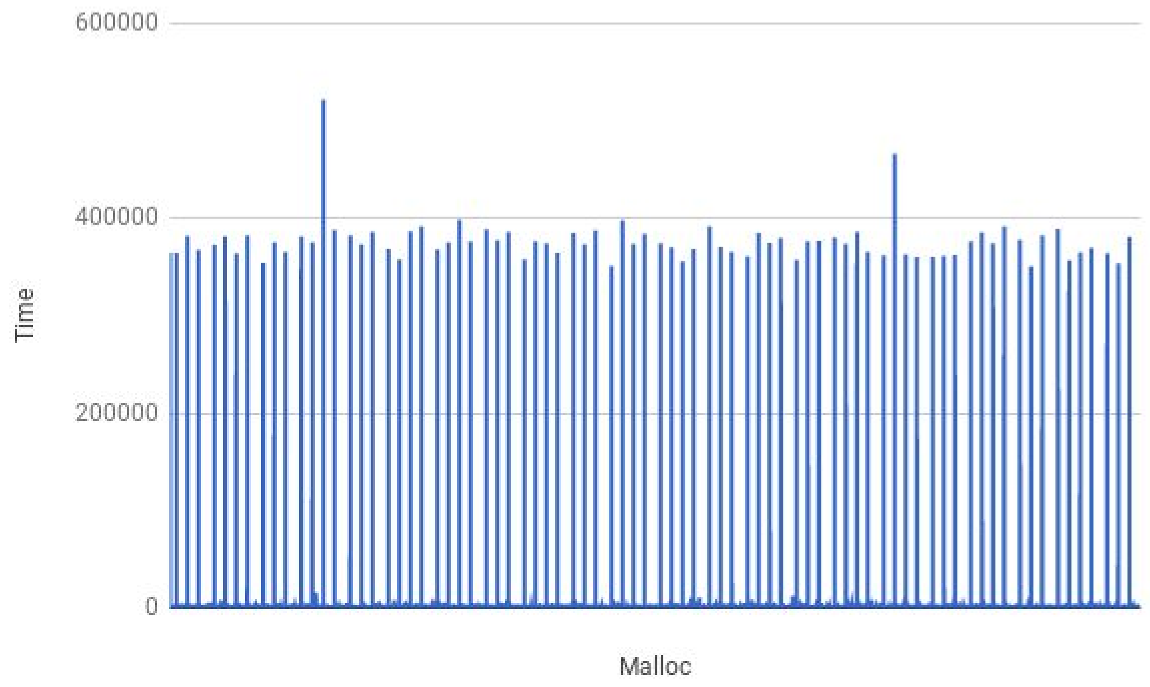
- <u>Free:</u>

993.2549	Average
908	Median
838	25th percentile
1048	75th percentile



- Malloc:

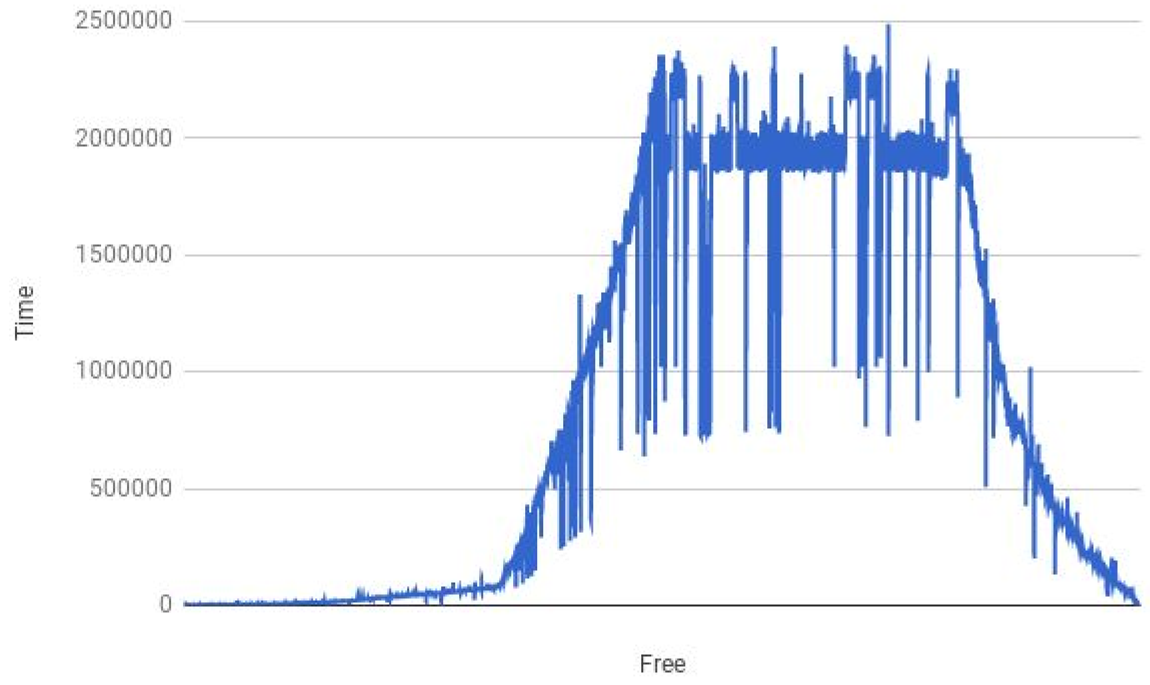
3773.4115	Average
769	Median
699	25th percentile
838	75th percentile



d) Mixed workload worst fit (1):

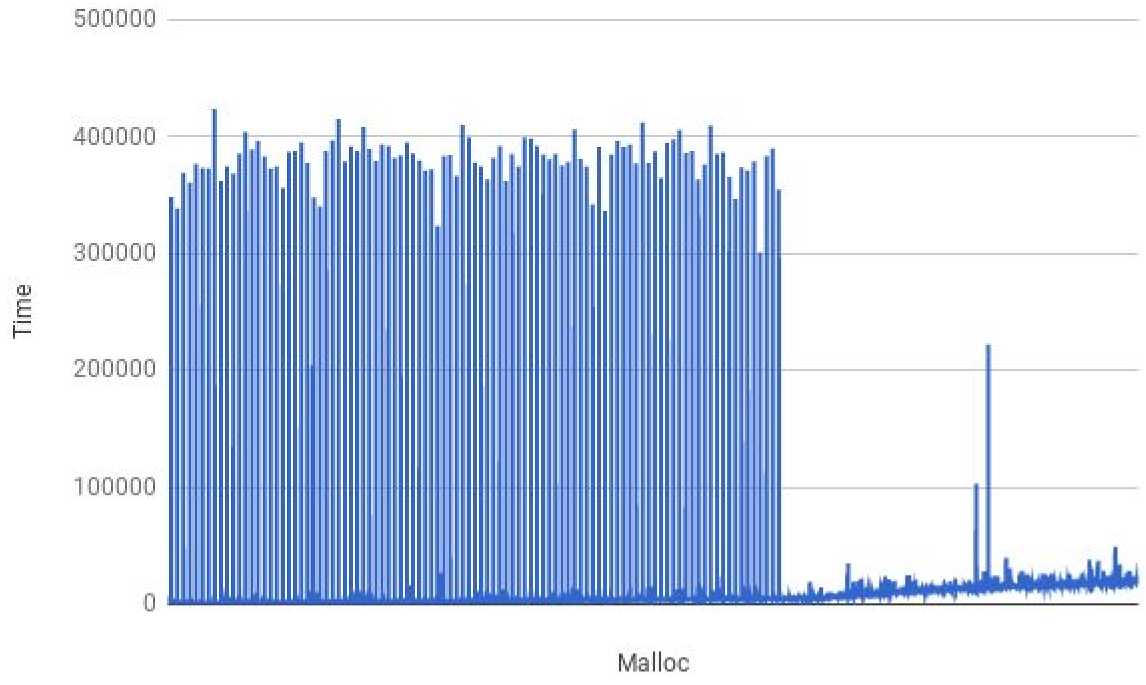
- Free:

891735.6025	Average
597149	Median
44001	25th percentile
1889223	75th percentile



- Malloc:

10301.0011	Average
3911	Median
2305	25th percentile
11314	75th percentile



We notice, on average, that best-fit is more time-efficient compared to worse-fit. This is simply caused by chance and the workload being given. On average, worst-fit and best-fit should have the same runtime for mallocs and frees.
In all 8 cases , the number of occasions when psumalloc() could not be satisfied due to a lack of memory is 0. This is similar to the sample tests performed by the TAs and uploaded to canvas.

Concerning internal fragmentation, there are also none that occur in my code.

These two claims were tested at the end of malloc, when an area to malloc was not found.