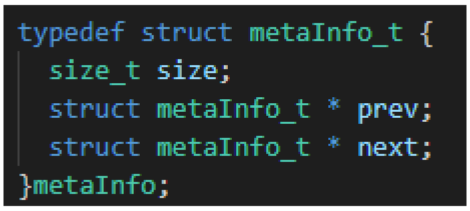
Report of Malloc Library Project

Name: Junfeng Zhi | NetID: jz399

1. Overview of Implementation

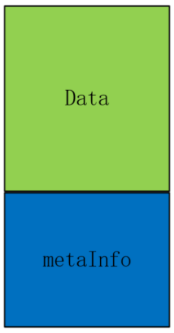
1.1 Definition of allocated block

In my implementation, I use a double linked list to record all the free blocks. There are two parts in each allocated block. First, it is the metadata. It contains three fields. ***Size_t size*** records how many bytes the data part of the block contains. ***metaInfo_t * prev*** points to the previous node of the current node in the double linked list. Similarly, ***metaInfo_t * next*** points to the next node of the current node. These two pointers help us locate current block and merge it with its neighbor.



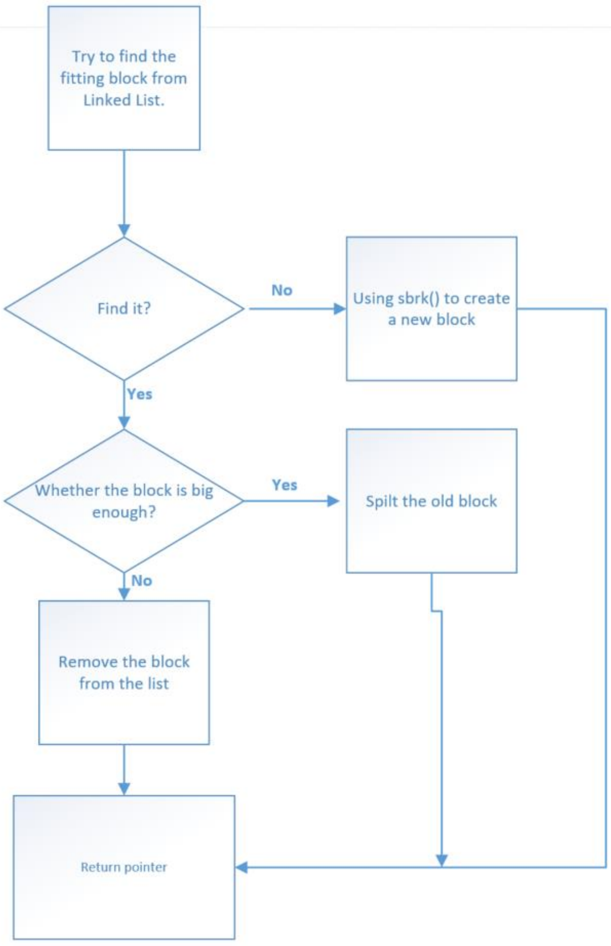
```
typedef struct metaInfo_t {
  size_t size;
  struct metaInfo_t * prev;
  struct metaInfo_t * next;
}metaInfo;
```

Second, it is the data part of the block, which is the memory requested by malloc function. The overview of the allocated block is shown below.



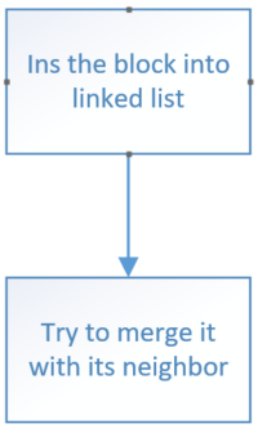
1.2 Basic idea of Malloc()

Malloc function will return a pointer which points to the allocated memory. The figure below shows the procedure of Malloc function. In the beginning, program will try to find the fitting block from linked list based on two policy: First Fit and Best Fit (Details can be found in README)**.** If fitting block does not exist, it will use sbrk() function to increase the data segment size and create a new allocated block. Otherwise, it will try to remove the fitting block or spilt it based on whether the fitting block is big enough. Finally, function will return the pointer. Importantly, pointer need to add the size of metaInfo to point at the data region. The runtime complexity of Malloc is O(n), n is the length of the linked list.


Try to find the fitting block from Linked List.
Find it?
No
Using sbrk() to create a new block
Yes
Whether the block is big enough?
Yes
Spilt the old block
No
Remove the block from the list
Return pointer

1.3 Basic idea of Free()

As shown in the figure above, there are two steps in Free function. First, the free function will insert the block into linked list. In order to merge the adjacent block to shorten the list, the nodes in the linked list are maintained in ascending order based on the address while inserting. Second, we merge the block with its neighbor if possible. The runtime complexity for Free function is O(n), n is the length of linked list.


Ins the block into linked list
Try to merge it with its neighbor

2. Results and Analysis of performance test

2.1 Experiment results

Pattern	First-Fit		Best-Fit	
	Execution time(s)	Fragmentation	Execution time(s)	Fragmentation
Small	21.83	0.14	1.94	0.04
Equal	22.54	0.45	23.13	0.45
Large	62.29	0.12	118.02	0.07

2.2 Analysis

For equal_size_alloc, program uses the same number of bytes (128) in all of its malloc calls. The results show that two malloc policy have similar runtime and fragmentation. First-Fit and Best-Fit will always find the first free block available and use it because all the blocks have the same size. Therefore, these two strategies work the same way.

For small_range_rand_allocs, program works with allocations of random size, ranging from 128 - 512 bytes (in 32B increments). In this situation, Best-Fit works much better than First-Fit. Although Best-Fit takes more time to search the fitting free block than First-Fit, is uses the free blocks more efficiently. Therefore, program call sbrk() less frequently. Result shows that the sbrk call has more impact on the result than traversing the block in this test.

For large_range_rand_allocs, this program works with allocations of random size, ranging from 32 - 64K bytes (in 32B increments). Results shows that First-Fit policy works better than Best-Fit. Best-Fit is more difficult to stop the search of blocks early, because the block size range is larger and it is difficult to find a block with an exact matching size. Therefore, the list iteration takes more time.

In conclusion, I recommend First-Fit policy because real life situation is similar to large_range_rand_allocs.