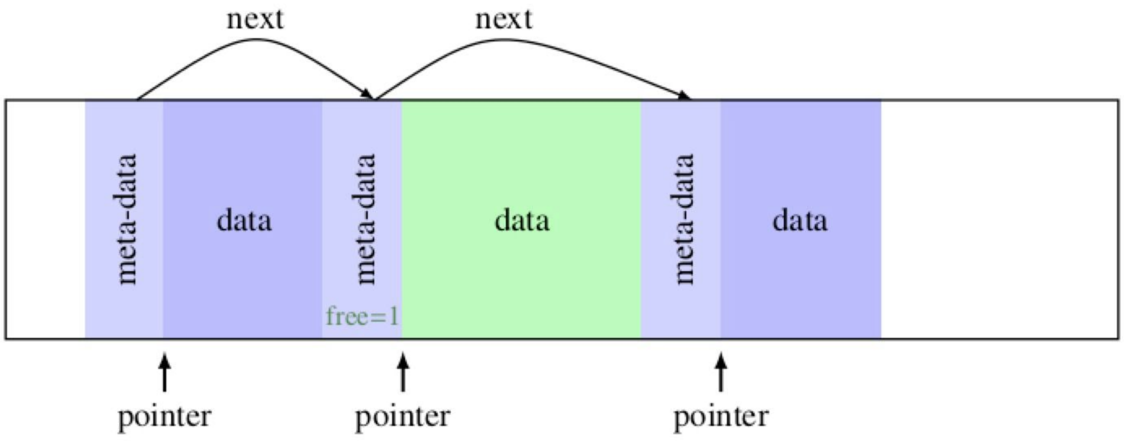
Overview

Requirements:

I first define the requirements for my malloc to be a valid malloc function. My malloc should be able to allocate the space no less than what the input (size_t size) requires. My malloc function return a pointer pointing to the start of valid memory address. When malloc is called multiple times, the allocated space cannot have overlaps unless it is freed before. Malloc function require address from the heap by using sbrk() function in my implementation.

Implementation:

I organized the avaliable heap memory as blocks. Block is a struct I define, which consists meta data region and data region. I connected these blocks using double linked list structure. Meta data region consists of information of blocks(including size of data region, next and prev pointer of block, "free" which marks if the block is avaliable for malloc). Data region is the real region that I allocate in malloc function and the address of first byte in data region is the address my malloc function will return. When malloc receives size as input, I first check if the head pointer of my block list is NULL. If it is, I need to extend the heap to create the first block, I wrote extend_heap() function to do this. If the head is not NULL, I use a pointer called "curr" to point to each block as I iterate through all the blocks. Here I use FF, BF, WF three policies to search for blocks. When I find proper block, I return the address of data region in this block and change the free to 0 representing "not avaliable for mallocing". If I have not find proper blocks, I called extend_heap() function to extending the heap and add a new block to the block list. A little detail in my implementation is that I want the data region to always be mutiple of 8 bytes, so I first adjust input size to the smallest mulptile of 8 bytes which is larger than input size.



For free, I need to first check if the input address is valid for me to free. By valid, I mean the input ptr should be the block malloced before and the address should be within the range of head pointer in the list and current break pointer in the list. Another important thing is that after many times of malloc and free, there would be many small blocks fragments. To make the free function useful, I need to combine these blocks together into one block. So I wrote cmombine function to combine nearby blocks into one block when the nearby block is also free.

Details of 3 Searching Algorithms:
First-Fit Allocation:
In the first fit algorithm, when receiving input request, “curr” pointer scans along the list for the first block that is large enough to satisfy the request. The first fit algorithm ensures that allocations are quick.
Algorithm for allocate (size)
```
size(block) = size + (BLOCK_SIZE)
Scan list for first block with blk->size >= size(block)
If head is NULL
Extend the heap to create first block
Else
Search for proper block
If block not found
Extend the heap to create new block
Else
If block->size - size > SIZE + threhold
Split the blocks
Set the blk->free to 0
return pointer to this region.
```
Algorithm test is O(K) time-complexity. K is the number of free blocks.
Best-Fit Allocation:
In the best fit algorithm, when receiving input request, “curr” pointer scans along the whole list for the block that is nearest to the size request. In my implementation, during iteration, if the block size equals to the request size I stop iterating and return this block.
Algorithm for Best-fit (size)
```
size(block) = size + (BLOCK_SIZE)
Set temp pointer to NULL
Set tmp pointer to head
Set variable Ans = (size_t)INT_MAX
Scan free list which block with tmp->size >= size(block)
If tmp->size equals to size
Set temp to tmp block.
break the while loop.
If ans > tmp->size - size
Set temp to tmp
Set ans equals to blk->size - size
Move tmp to the next block.
Return temp pointer as result.
```
Algorithm test is O(N) time-complexity. N is total number of blocks in the list.
Worst-Fit Allocation:
In the worst fit algorithm, when receiving input request, “curr” pointer scans along the whole list for the block that is largest to the size request.
Algorithm for Worst-fit (size)
```
size(block) = size + (BLOCK_SIZE)
```

```
Set temp pointer to NULL
Set tmp pointer to head
Set variable Ans = (size_t)INT_MIN
Scan free list which block with tmp->size >= size(block)
If ans < tmp->size - size
Set temp to tmp
Set ans equals to blk->size - size
Move tmp to the next block.
Return temp pointer as result.
```

Algorithm test is O(N) time-complexity. N is total number of blocks in the list.

Performance

My test results from the performance experiments are listed below:

First-Fit	Equal(10000 items)	Small	Large
Runtime	55.637005 s	108.233548 s	176.151175 s
Fragmentation	0.450000	0.047021	0.080707

Best-Fit	Equal(10000 items)	Small	Large
Runtime	551.670759 s	145.119683 s	457.58707 s
Fragmentation	0.450000	0.020526	0.039482

Worst-Fit	Equal(1000 items)	Small	Large
Runtime	1471.508609 s	322.02256 s	384.711149 s
Fragmentation	0.550000	0.390140	0.462437

Analysis

Runtime Perspective:

Observed from the tables, I can see that two significant runtime performance difference.

(1) Frist-Fit policy tends to have a better run-time performance than Best-Fit and Worst-Fit. And Best-Fit can have a better run-time performance than Worst-Fit policy. It is easy to understand the reason when I think about the mechanism. First-Fit policy would return whenever I find the proper block, so each time It will iterate from the beginning of list, return at the first proper block in the list, which is O(K) time complexity. However, for Best-Fit and Worst-Fit policy. The pointer ptr has to iterate throught the whole list to decide the most proper block. So the

run-time is O(N), which N is the total number of blocks in the list. One difference between Best-Fit and Worst-Fit is that for Best-Fit, when I find a block with the size equals to the required size I will stop iterating and just return this block. So the best run-time for Best-Fit policy in my implementation could be better than O(N). But for Worst-Fit policy, I cannot decide which block to return until iterating the whole list. In conclusion, First-Fit has the best runtime performance, then Best-Fit and last is Worst-Fit policy.

(2) For differenct allocate/free paterns, different search policy have different performance.

(2.1) For FF policy, Large-range-alloc tends to have longer running time than smallrange alloc, then equal-size-alloc has the best runnint time performance. The reason is that when malloc multiple times and free randomly, many block holes exists in the list. For equal-size-alloc, any of these holes is proper so the running time is great. Compared with large-range-alloc, required size in small-range-alloc tends to be more concentrated. So more requests can find the proper block in the list. However, large-range-alloc has more extrem data, as a result it is more likely to find no proper existing blocks and need extending heap. These elements all causing the runtime variance.

(2.2) For BF policy, large-range-alloc is also longer than small-range-alloc and the reason is the same as FF. However, runtime of equal-alloc is longer than other two malloc policy. For small-range-malloc and large-range-malloc, after random free there would be block whole which has random size in a range. When the input size equals to one of the existing holes, it do not need to search the whole list. However, for equal-alloc scenario, each malloc is the size. To find the next proper block, it needs to iterate through the whole list. As a result, the runtime would be longer than small-range and large-range malloc.

(3) For WF policy, small-range, large-range and equal size allocate scenario all take quite a long time. Because I need to iterate through the whole linked list to decide the largest proper block and the runtime is always O(N).

Fragmentation

Obeserved from the table, I can find out that Worst-Fit policy has the biggest fragmentation.

The Best-Fit policy has the smallest fragmentaion. Between three different allocate scenarios, equal-size-alloc has the biggest fragmentation.

(1) From the definition of fragmentation, it equals to free-data-region/whole data region. We know that Worst-Fit would allocate the largest block each time. As a result, it would free the largest block. Therefore, Worst-Fit has the largest fragmentation. Because Best-Fit policy each time will allocate the smallest block, as a result, it would free the small blocks in the list. Therefore, Best-Fit policy has the smallest fragmentation.

(2) Another interesting number is that for different allocate scenarios, equal-size-allocation tends to have the fragmentation. I think the reason is that each time I call for the same size of block, the list is more likely to not have proper blocks. So I need to extend the heap to get the proper block. Calling extend function in my implementation will build the same size of blocks. As a result, equal-size-alloc tends to have the same fragmentation. Also, I observe that

small-range-alloc and large-range-alloc have smaller fragmentation than equal-size-allocation. The reason is that small-range and large-range allocation would ask for random size blocks. Therefore, our block-list has random size blocks rather than the same size blocks. Therefore, during multiple free calls, different size of blocks will be freed. Some is small, some is larger. But the whole free-data-segment space is smaller than equal-szie-alloc. Therefore, small-range-alloc and large-range-alloc tend to have smaller fragmentation than equal-size-allocation.

Conclusions

In the end, both strength and weakness of each allocation policy are discussed discussed.

	Strenth	Weakness
First-Fit	Run-time is fastest	Malloc large size block for small size request
Best-Fit	Leave large blocks for large size requests. Each request get the most proper size block	Bad fragmentation.
Worst-Fit	Good fragmentation	Eliminate large blocks of memory, thus requests for large block cannot be met.