HW1 Malloc Library Part1

This file will explain different ways to implement malloc Library and compare the difference between their behavior

1 Implementation

1.1 Data Structure

This approach use a single linked list that link all the free blocks. This linked list is called FreeList in this report.

Each structure inside the FreeList is called memBlock and contents the things below to describe the information of a blocks:

```
size_t size;
void * add;
struct _memBlock * nextFree;
```

in which, size means the size of the block, add means the address of the beginning of the block, and nextFree points to the next free block if it is inside the free list.

As shown in my_malloc.c there a two global valuables:

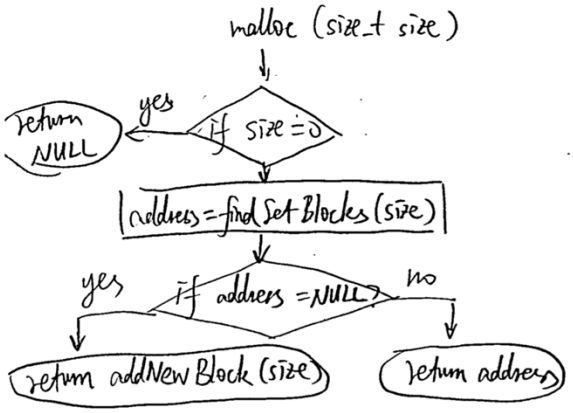
```
memBlock * freeHead;
unsigned long dataSeg;
```

freeHead pointed to the first free block and is initialized to NULL and dataSeg is the total size of the block that have been allocated(excluding the space that used by all the struct memBlock)

The reason that memBlock can act as a header of each block well be discuss in 1.2 inside addNewBlock function.

1.2 Function Description

The flow chart of malloc are shown below



picture 1.1 flow chart of malloc function

As shown in the picture, malloc calls two functions
  void * addNewBlock(size_t size);
  void * ff_findSetBlocks(size_t size) or void *bf_findSetBlocks(size_t size);

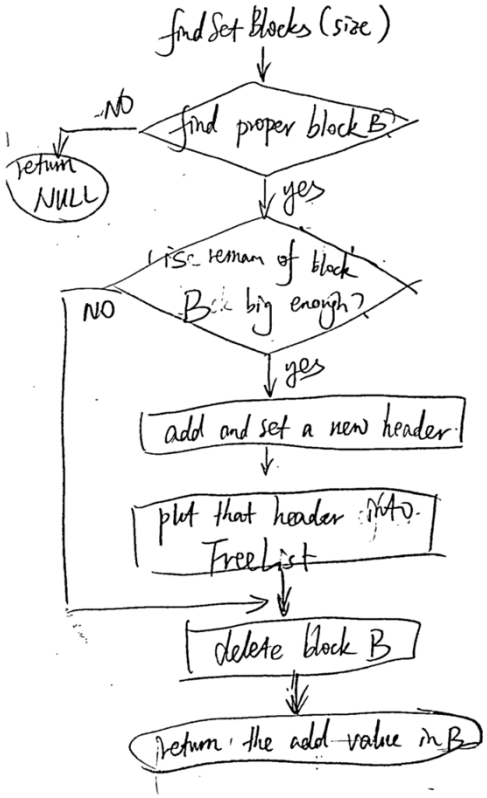
addNewBlock(size_t size) would first use sbrk() to allocate a sizeof(memBlock) and then use sbrk() again to allocate another size that is input to the function. So that the memBlock and block would adjacent to each other physically. Set the add value in the memBlock to be the start address of the block address so that the equation for every memBlock * Blk below is valid:

$$\text{Blk->add} - \text{sizeof(memBlock)} = \text{Blk} \qquad (1.1)$$

Then addNewBlock would return the add value in that newly allocated memBlock.

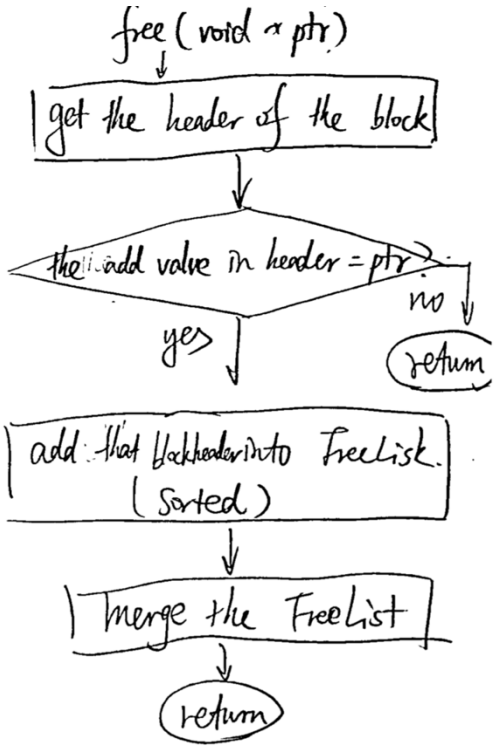
findSetBlocks(size_t size) would search all the node in the FreeList and find the proper block. The only difference between ff_findSetBlocks and bf_findSetBlocks is that they would have different strategy to find the proper block. If it cannot find that kind of block it would return NULL. If a block has been found, the function would try to reorganize the FreeList and then return the add value in that block.

To be more specific, during the reorganization we need add a new header at the beginning of the remain block address. Before that we need to check whether the remain block size is big enough to hold a memBlock as a header. If so we need to add that new block into the FreeList and then delete the proper chosen block, if not all we need to do is simply delete that chosen block. This process is shown below. By doing this, equation 1.1 is always valid for every memBlock * Blk.



picture 1.2 the flow chart of function findSetBlocks

As for void free(void * ptr), since the ff and bf share the same data structure so there is no difference between ff_free and bf_free. The flow chart of free is shown below.



picture 1.3 the flow chart of free function

As mentioned in equation 1.1 the header of the block can be get by

memBlock * toFree = (void*) (ptr - sizeof(memBlock));

As for the merge() function, it is used to prevent any two or three blocks adjacent to each other physically. It would check all the things inside the FreeList if there are any node memBlock * curr inside the list satisfy this equation:

curr->add + curr->size = curr->nextFree

than merge this node with the node after that.

2 Result

table 2.1 execution time and fragmentation of different test case

		First Fit	Best Fit
equal	exec time	28.666670s	56.559150s
	fragmentation	0.450000	0.450000
large	exec time	35.648731s	40.625189s
	fragmentation	0.086037	0.030463
small	exec time	64.741855s	18.244704s
	fragmentation	0.062818	0.021723

Note: this result is not base on the default parameter. The reason that the program takes a long time with default values is that it uses recursion to add or delete items in FreeList.

3 Analysis

According to table 2.1 it is clear that in equal test case, the FF and BF have the same fragmentation but BF have a longer execution time. It is because that this test case malloc and free a same size of memory. Because all the block are at the same size when it is malloc and free the freeblocks in side FreeList are all the same. So that the fragmentation is the same. However, in BF method it has to go though all the items inside the FreeList to make the decision, which is the reason why it take longer time than FF method.

When it comes to large test case and small test case BF method would always get less fragmentation. That is because BF method tend to have minimize waste on using freeblocks. For example, if in FreeList there a three free block at the size of 200B, 120B,100B. when malloc a 100B and 150B inorder the BF would first use the 120B block and then the 200B block then have a fragmentation around 0.40. The FF method, on the other hand, would first use 200B block and then allocate another location in the memory to put that 150B. This would cause a fragmentation around 0.6.

As for the run time of the large test case, it has the same reason in equal test case. The BB method have to go though all the FreeList to get the result while FB will return when it meet the first block that is big enough. However, when it comes to the small test case, this theory seems not working well.

This is because that with those very small blocks be allocated, the free blocks in FreeList are also tend to be small. Hence, the waste cause by FF is not trivial anymore. It is likely that a small blocks occupied a relatively large blocks and another blocks cannot use that rest space anymore. Also, the size of the header structure memBlock is also not trivial compare to the small size of those test blocks, which makes the waste cause by FF method even worse.

With those wastes, FF method have to allocate more memory to malloc memory blocks, which would make the FreeList longer. This can be prove by the out put of the date segment:

table 3.1 data segment lay out for small test case

	First Fit	Best Fit
Data segment size	3451744	3284704
Data segment free size	216832	71352

As shown in table 3.1 the FF method have more data segment size and more data segment free size then BF method, which means that FF method have a longer FreeLise.

Meanwhile, FF has less chance to find proper blocks inside the FreeList. What would happen if there is no proper block inside the FreeList? In FF method, it has to search all the List just as BF method do. So, with less chance to find block and longer FreeList, FF method tend to spend more time in the small test case.

In conclusion, BF would always have less or equal fragmentation then FF and likely to have longer execution time then FF. However, when it comes to some small pieces of memory blocks, the waste cause by FF method is not trivial anymore. In this kind of cases, the FreeLink with FF method tend to be longer, and the chance for FF to find a proper free block is lower. So in this particular case, BF method would be faster than FF method.

4 Other Approaches and Improvements

There are many other data structure that can be use in this malloc implementation. The approach discuss above is the third method I try. And I would like to share other 2 method I try and the code is given in the folder complexMethod

4.1 Seach All

This method creates a linked list that has all the blocksInfo. Inside the blocksInfo there is a int to indicate whether this block is occupied or not. So each time it need go in to the findSetBlocks function it has to go though all the blocks we have.

Also as you find in the code this method did not take advantage to make the blocksInfo physically adjacent to the blocks itself. So inside the free function it also has to go though every blocks to find out which one to free.

4.2 Double Link

This method is base on method 4.1 and add more feature in blocksInfo. In side blocksInfo it now has two more pointers that is prefree and nextfree, which point at the previous free block and the next free block. With this two pointers, every free blocks are in two linked list the normal one that have all the blocks (in order) and the freeList that have all the free blocks (no necessary in order). So inside findSetBlocks it only need to search the freeList to find the useful free blocks.

However this method is too complex with so many pointers to take care of. Then the method in the first part of this report come out.

4.3 Improvements

However, when using the test case provided, the parameter has to be change to make this work. The NUM_ITERS is changed to 20, 5, 100 in equal, large and small. This is because when implement delete and addSorted in FreeList, recursion has been used, and that recursion may cause lots of delay with a huge test case like this. To improve that, two pointers can be used when delete and addSorted.