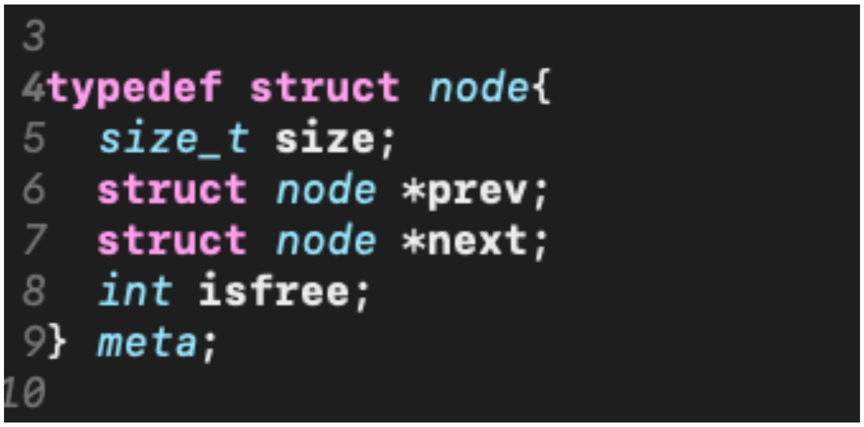
1.Implementation

Data Structure:

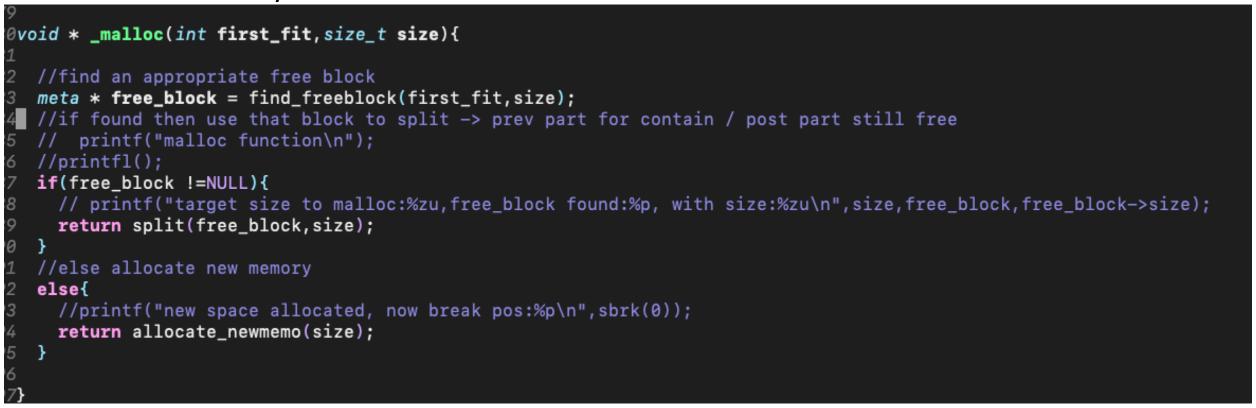


```
typedef struct node{
  size_t size;
  struct node *prev;
  struct node *next;
  int isfree;
} meta;
```

Use a doubly linked list to store those free blocks. All these blocks are defined as a data structure called meta data which has the attributes: the memory it occupies, if this memory is free to use, and the pointer to the previous meta data, the pointer to the post meta data.

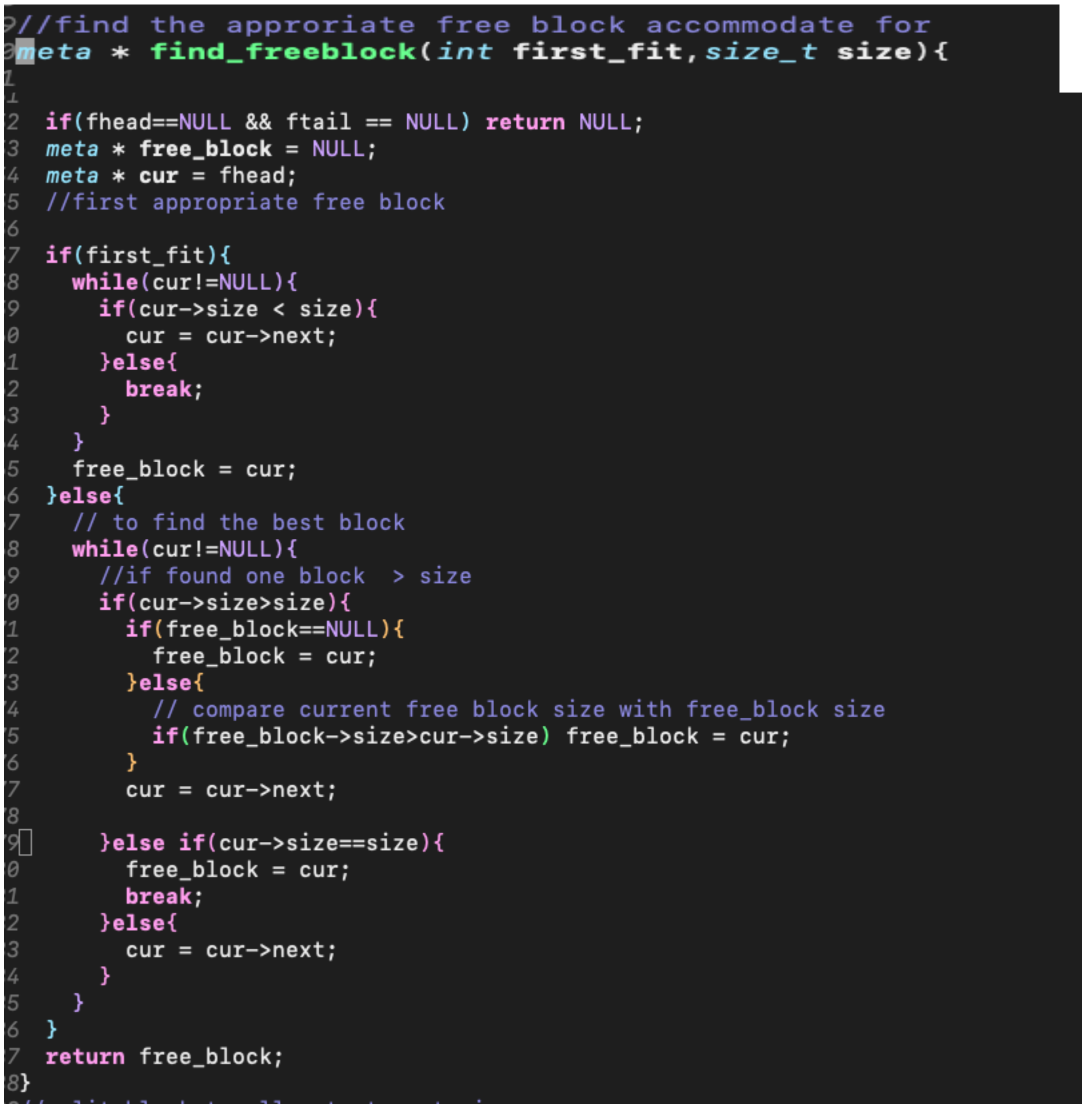
Details:

As to allocate a memory space, we should first consider whether there is a free space meeting the requirement. So, I try to find a block (meta data) through the free block list(the doubly linked list used to store those blocks which have been freed). Then, if there is a block found, we can use that block to allocate memory for the target. Otherwise, we should expand the heap to allocate new memory.



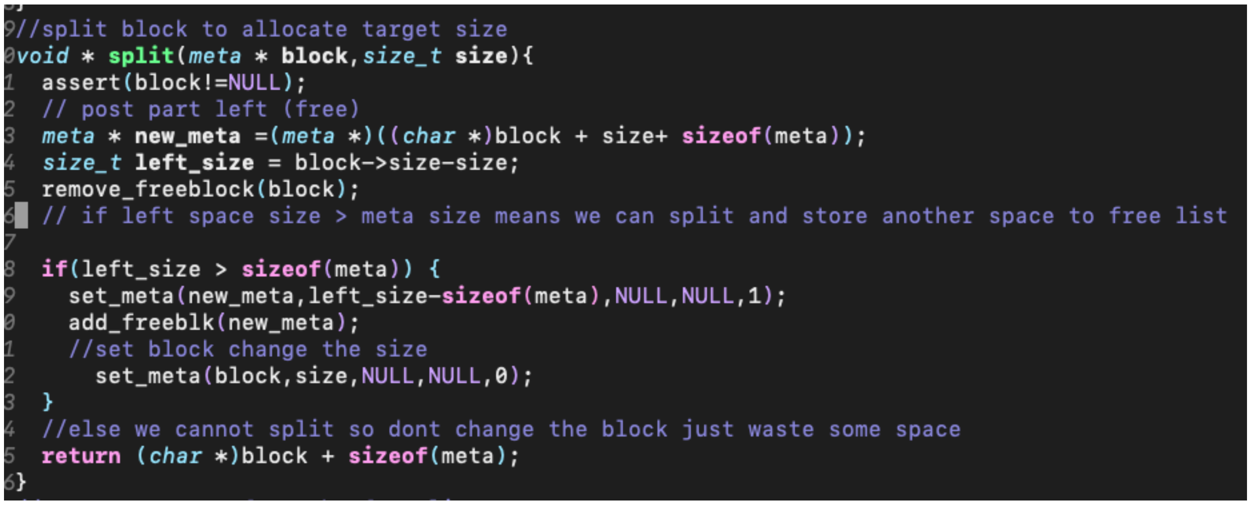
```
void * _malloc(int first_fit,size_t size){

  //find an appropriate free block
  meta * free_block = find_freeblock(first_fit,size);
  //if found then use that block to split -> prev part for contain / post part still free
  //  printf("malloc function\n");
  //printfl();
  if(free_block !=NULL){
    // printf("target size to malloc:%zu,free_block found:%p, with size:%zu\n",size,free_block,free_block->size);
    return split(free_block,size);
  }
  //else allocate new memory
  else{
    //printf("new space allocated, now break pos:%p\n",sbrk(0));
    return allocate_newmemo(size);
  }

}
```

As to find the appropriate block, I use an int parameter first_fit to decide whether to return the first appropriate block or the best block. It is just a flag to choose a policy. The implementation is quite eazy. Always, I use the fhead and ftail which is defined as the head and the tail of the free list to main the list.



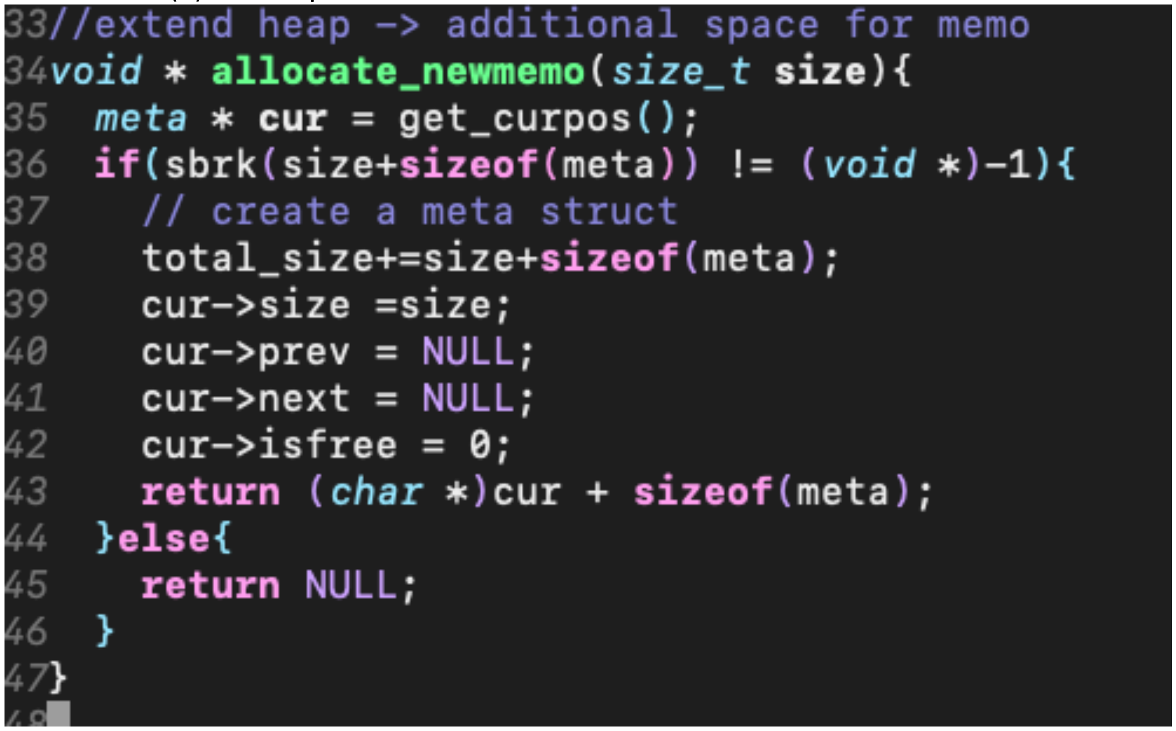
```
//find the approriate free block accommodate for
meta * find_freeblock(int first_fit,size_t size){

  if(fhead==NULL && ftail == NULL) return NULL;
  meta * free_block = NULL;
  meta * cur = fhead;
  //first appropriate free block

  if(first_fit){
    while(cur!=NULL){
      if(cur->size < size){
        cur = cur->next;
      }else{
        break;
      }
    }
    free_block = cur;
  }else{
    // to find the best block
    while(cur!=NULL){
      //if found one block  > size
      if(cur->size>size){
        if(free_block==NULL){
          free_block = cur;
        }else{
          // compare current free block size with free_block size
          if(free_block->size>cur->size) free_block = cur;
        }
        cur = cur->next;

      }else if(cur->size==size){
        free_block = cur;
        break;
      }else{
        cur = cur->next;
      }
    }
  }
  return free_block;
}
```

As to split the block, I split the block into two blocks. I create a new meta data at the corresponding position (the second part). The new created block has the size which equals to the original block size minus target size. Then the first split part is used for storage and is not free anymore. So, I remove it from the free list. Next, I check if the new block (second part) size is bigger than the meta data struct size. This step is to see whether the left part can at least store a meta data struct. If so, we set the new created block and add it to the free list.



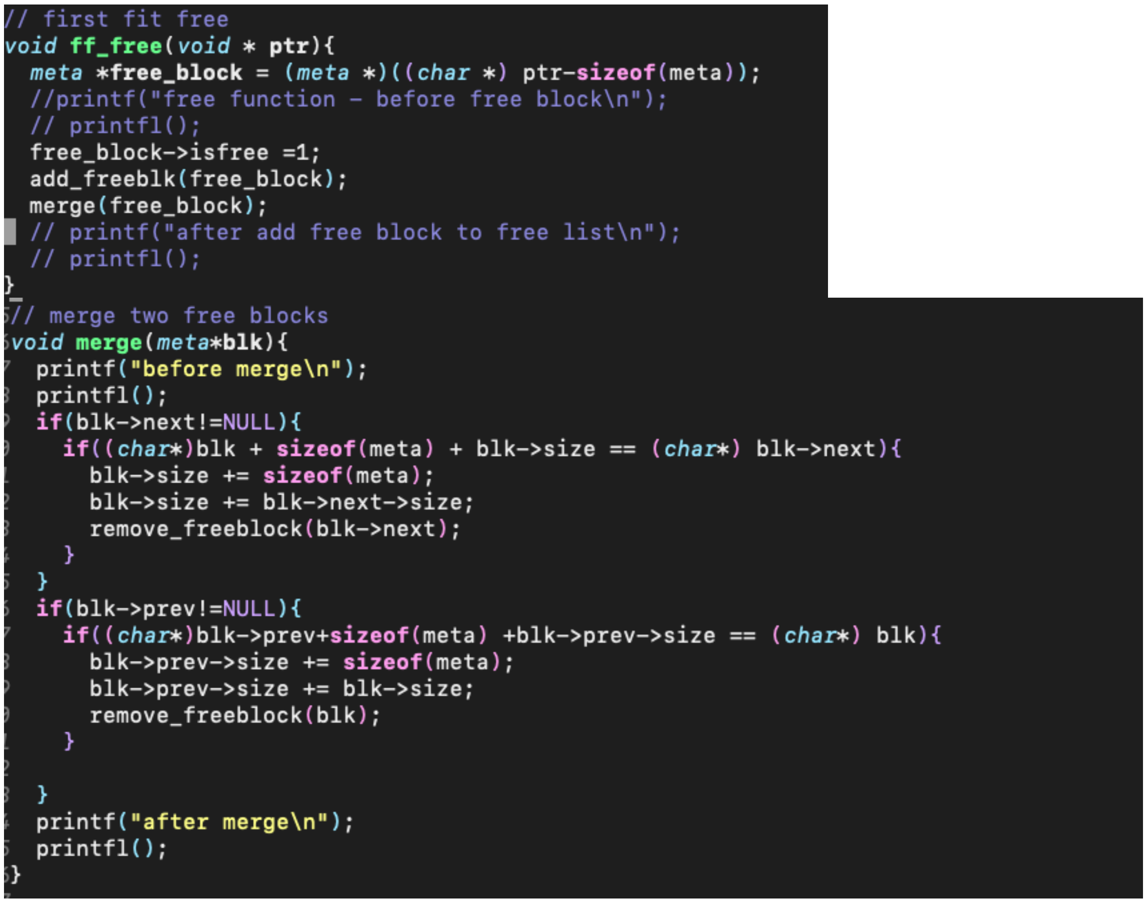
```
//split block to allocate target size
void * split(meta * block,size_t size){
  assert(block!=NULL);
  // post part left (free)
  meta * new_meta =(meta *)((char *)block + size+ sizeof(meta));
  size_t left_size = block->size-size;
  remove_freeblock(block);
  // if left space size > meta size means we can split and store another space to free list

  if(left_size > sizeof(meta)) {
    set_meta(new_meta,left_size-sizeof(meta),NULL,NULL,1);
    add_freeblk(new_meta);
    //set block change the size
      set_meta(block,size,NULL,NULL,0);
  }
  //else we cannot split so dont change the block just waste some space
  return (char *)block + sizeof(meta);
}
```

As to allocate new memory, I create a meta data at the current break position using the function sbrk(0). Then update meta.



```
//extend heap -> additional space for memo
void * allocate_newmemo(size_t size){
  meta * cur = get_curpos();
  if(sbrk(size+sizeof(meta)) != (void *)-1){
    // create a meta struct
    total_size+=size+sizeof(meta);
    cur->size =size;
    cur->prev = NULL;
    cur->next = NULL;
    cur->isfree = 0;
    return (char *)cur + sizeof(meta);
  }else{
    return NULL;
  }
}
```

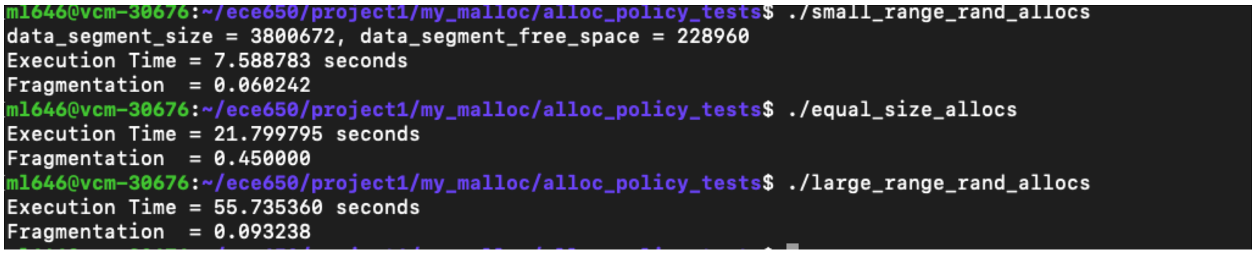
As to free a block, first is to find the head meta data position. Then set the attribute isfree to 1 and add the block to the free list. In the end, to see if we can merge the block according to the position at the list. Merge Function is not complex, just check the previous block and the post block to see whether the prev block address + sizeof(meta) + prev block size is equal to current block address.



```c
// first fit free
void ff_free(void * ptr){
  meta *free_block = (meta *)((char *) ptr-sizeof(meta));
  //printf("free function - before free block\n");
  // printfl();
  free_block->isfree =1;
  add_freeblk(free_block);
  merge(free_block);
  // printf("after add free block to free list\n");
  // printfl();
}
// merge two free blocks
void merge(meta*blk){
  printf("before merge\n");
  printfl();
  if(blk->next!=NULL){
    if((char*)blk + sizeof(meta) + blk->size == (char*) blk->next){
      blk->size += sizeof(meta);
      blk->size += blk->next->size;
      remove_freeblock(blk->next);
    }
  }
  if(blk->prev!=NULL){
    if((char*)blk->prev+sizeof(meta) +blk->prev->size == (char*) blk){
      blk->prev->size += sizeof(meta);
      blk->prev->size += blk->size;
      remove_freeblock(blk);
    }

  }
  printf("after merge\n");
  printfl();
}
```

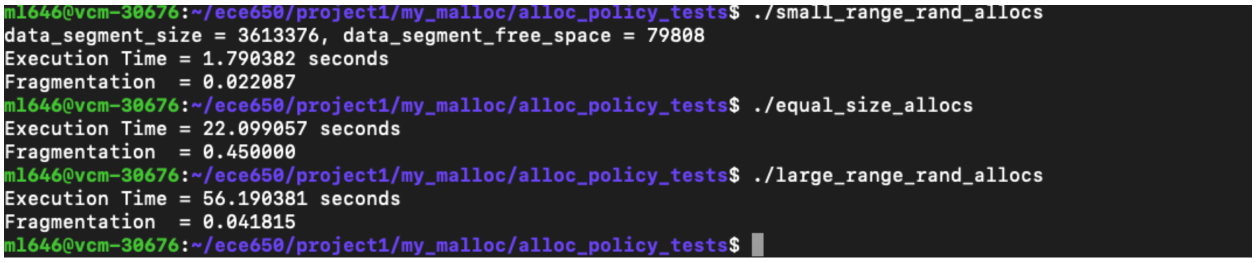
2.Results

FF:



```
ml646@vcm-30676:~/ece650/project1/my_malloc/alloc_policy_tests$ ./small_range_rand_allocs
data_segment_size = 3800672, data_segment_free_space = 228960
Execution Time = 7.588783 seconds
Fragmentation  = 0.060242
ml646@vcm-30676:~/ece650/project1/my_malloc/alloc_policy_tests$ ./equal_size_allocs
Execution Time = 21.799795 seconds
Fragmentation  = 0.450000
ml646@vcm-30676:~/ece650/project1/my_malloc/alloc_policy_tests$ ./large_range_rand_allocs
Execution Time = 55.735360 seconds
Fragmentation  = 0.093238
```

BF:



```
ml646@vcm-30676:~/ece650/project1/my_malloc/alloc_policy_tests$ ./small_range_rand_allocs
data_segment_size = 3613376, data_segment_free_space = 79808
Execution Time = 1.790382 seconds
Fragmentation  = 0.022087
ml646@vcm-30676:~/ece650/project1/my_malloc/alloc_policy_tests$ ./equal_size_allocs
Execution Time = 22.099057 seconds
Fragmentation  = 0.450000
ml646@vcm-30676:~/ece650/project1/my_malloc/alloc_policy_tests$ ./large_range_rand_allocs
Execution Time = 56.190381 seconds
Fragmentation  = 0.041815
ml646@vcm-30676:~/ece650/project1/my_malloc/alloc_policy_tests$
```

The results are interesting. In common sense, from the logic of the two policies, first fit policy tends to have shorter execution time while best fit policy tends to have better memory usage (lower fragmentation). However, the result of small_range_rand_allocs is not as expected. BF has lower fragmentaion, which is reasonable, but takes shorter to execute. I think this is because in this case the required memory size varies in a small range. Imagine three free blocks have the size 5, 3, 7, respectively. With BF policy, when we require the size 3, 5, 7, it takes all the three buckets. However, with FF policy, it takes 5 and 7 and must require a piece of new memory. This may account for the result that BF runs faster than FF.

For the large_range_rand_allocs and equal_size_allocs, BF takes longer to run while has lower or same fragmentaion, which is what we expected.

Generally, I would recommend first fit policy because the large_range_rand_allocs setting is the closest to real cases. Most of the time the required sizes vary in a great scope. Consequently, trying to traverse and find a best fit block is difficult and meaningless. If we know ahead that the required sizes are similar, I will recommend best fit policy instead according to the experiment results.