# Homework 1 Report – ECE650

Jing Wang jw844

## *I.* Objectives

This assignment aims to implement the dynamic memory allocation functions from the C standard library in C code. It mainly focuses on malloc() and free(). What is more, the malloc() function takes in the number of bytes for a memory allocation, locates an address in the heap region where there is enough space to fit the specified number of bytes, and returns this address. The free() takes an address as the argument and marks that data region as available.

The implementation of the malloc() function can be achieved by using the sbrk() system call. With the help of the sbrk() function, it is able to return the address that represents the current end of the processes data segment and grow the size of the processes data segment by the amount specified by the input argument. In addition, there are two strategies that need to be implemented when calling the malloc() function to allocate memory. They are the first fit and best fit. For the first fit case, during tracking the free list, the address from the first free region with enough space to fit the specific allocation size can be allocated. As for the best fit, the address from the available region has the smallest number of bytes greater than or equal to the requested allocation size.

The requirements can be divided into two parts, one is for the malloc() function and the other is for the free() function. As for the malloc() function, the first-fit policy, and best-fit policy are required and if there is free space that fits an allocation request, the sbrk() function should not be used to increase the memory segment. As for the free() implementations, the first-fit case and best-fit case are required as well. Besides, if a free region of memory is adjacent to other free memory regions, it is required to merge the adjacent free regions into a single free region of memory.

In this assignment, I used Visual Studio Code to code in C and this report consists of three parts. The first one is the aims and requirements of the assignment which are already mentioned. Then, the design and implementation details will be covered. Finally, the performed analysis is carried out.

## *II.* Implementation

The double-linked list is applied to achieve the dynamic allocation. One element of this linked list can be represented as follows:

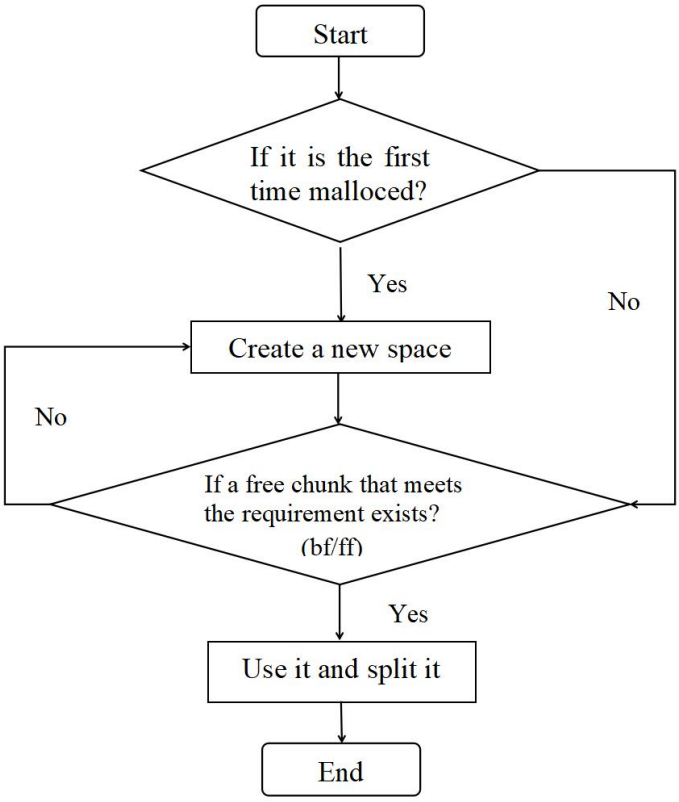
| Metadata | Payload |
|---|---|

The metadata part is used to store the information, such as the pointers to the previous one and the next one, the requested allocation size, and the variable to record whether this data chunk is available. Its structure can be shown in the figure below:

```
typedef struct metaData{
    bool isFree;// whether the block is avaliable
    size_t size;//the size that a user asks
    struct metaData * prev;
    struct metaData * next;
} mData;
```

There are four functional functions used in the malloc() function. The first one is generation(), and it is used to create a new space if there is no more free space that fits in the request allocation size. The second one is the ffChunk() which is to find the first qualified space in the free data region (using first fit policy) and the next is the bfChunk() function which is used to find the best-qualified space in the free data region (using best-fit policy). If the free data chunk has a larger size, the split() function is applied to cut this chunk into a used chunk and an available chunk to maximize the space efficiency.

The whole algorithm of the malloc() function can be described as the following flow chart.




The realization of the first fit malloc() function is traversing the whole linked list, and if there is a free data chunk that meets the requirement, then the chunk is allocated. Similarly, the best-fit malloc() function traverses the whole linked list first. If there is an available candidate, it will not stop traversing but continue to traverse until it finds the chunk with the minimum size which fits the required allocation size. It will not return immediately unless it finds a free chunk has the same size as we request.

As for the free() function, there is no difference between the two functions. With the given pointer, it checks whether it is valid (not empty), then it returns an address after freeing and

marks the data region as free. Then, it checks whether its adjacent chunks are available. If they are, it combines their size and adjusts their pointers.

When I finished my code, I tried to malloc some int pointers and free them to see the changes in their addresses and test whether my code can work well. In addition, I also tried the offered test cases and keep optimizing my algorithm.

## *III.* Performance Analysis

According to the table below, we can see that in terms of execution time, the performance of the best-fit policy is near the performance of the first-fit policy in the equal-size case. However, in the large and unequal size case, the best-fit policy executes obviously much longer time than the first-fit policy executes, and the best-fit policy takes a slightly longer time than the first-fit policy does in the small and unequal case. In conclusion, the first-fit algorithm has a better performance than the best-fit algorithm in terms of execution time.


As for the fragmentation, the two policies have the same performance in the equal size case. But for the small and unequal size case, the fragmentation calculated by the first fit policy is much smaller, which means that it merges more free chunks together. Therefore, it has a better performance in this case. However, in the larger and unequal size case, the best-fit policy has a smaller fragmentation, and this means that the best-fit policy performs much better than the first-fit policy in this case. The detailed information is illustrated in the table below.