Memory Manager Design Document

Dean.Cochran Xu.Song

# Purpose

In this lab, we are asked to create a data structure that can simulate a simple memory manager that can keep track of the usage of memory allocation. Specifically, our job is to implement two functions of the memory manager, request memory and return the memory. This project can help us deepen the understanding of the structure of the doubly linked list and how does memory manager work.

# Specification

When a memory manager object is created, the size of the memory manager (how many stuff can the memory manager contains) is restricted. The consumer can request memory by giving the size of the request and the owner of the requester. When memory allocation is required, the program will search for appropriate free allocation in the memory manager. If there is enough free space for the requester, the requester will be given a position that indicates where this requester located in the memory manager and then the program returns the requester. Otherwise, the requester can’t store in the manager, the program will return null. The consumer can also return a memory allocation by giving the object. When a memory allocation is asked to return to the memory manager, it will become a free space which has exactly the same size as the object is requested before. If there is free space next to this new deallocated one, these free spaces will merge into one free space that has a bigger size, so it can use to store new requester in the future.

# Design Overview

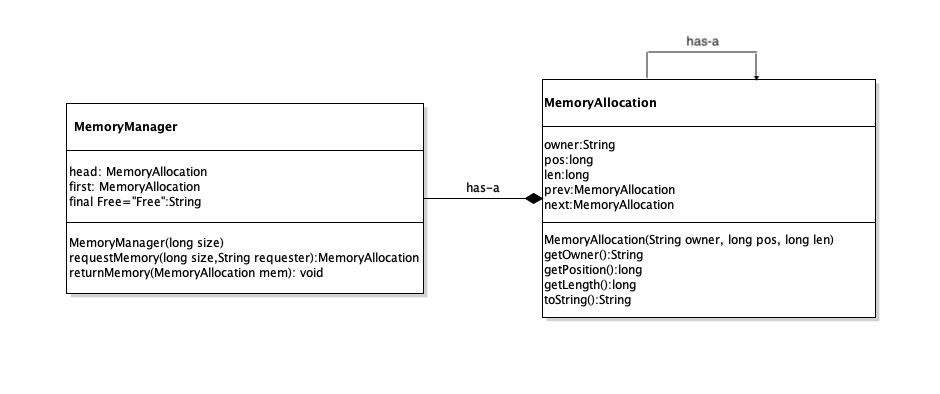


Figure 1: Class Diagram

Consider that we need to check memory allocations that besides the one we need to deallocate, a circular doubly-linked list will be an appropriate ADT to implement the memory management. As shown in the class diagram, the memory manager object can be seen as a linked-list composed of memory allocations. Each memory allocation consists of 5 attributions. Owner is a string to store the requester information, pos indicates the position of the specific memory allocation in the memory manager, len is the size of the allocation, or how many spaces this allocation occupied in the manager. There are also next and prev that point to the next and previous memory allocation object of the current one. MemoryAllocation() constructor will set owner, pos and len, these three attributes for each memory allocation. And the other three methods in the MemoryAllocation class can return the corresponding attributes’ value of the specific memory allocation. For MemoryManager, it contains a head and a first. The head and the first are both memoryAllocation objects, they will be created when the constructor of memoryManger is called. The constructor of MemoryManager takes a long integer as a parameter, this integer will be the max free spaces the manager has. requestMemory() method will return a memory allocation of the requested size if there is enough free size in the manager. returnMemory() release the memory allocation back to free space than can be used in the future memory request.

# Detailed Design Overview

## Constructor

When the constructor of MemoryManager is called, two memory allocation objects will be created. One is head, and the other is first. The head is just a dummy head used as a reference later in other methods, it doesn’t occupy any free spaces in the memory manager. Therefore, when creating the head using requestMemory(), the len of head is 0; the owner of head we call it “head”; the pos is -1. The first is the actual memory allocation in the memory manager. Since the manager is just created by the constructor, the manager is all free. So the first’s owner is “Free”; len is the size parameter passed into the constructor, and the pos is 0. Since the manager is a circular doubly linked list, we set both first’s prev and next to head, and both head’s prev and next to first.

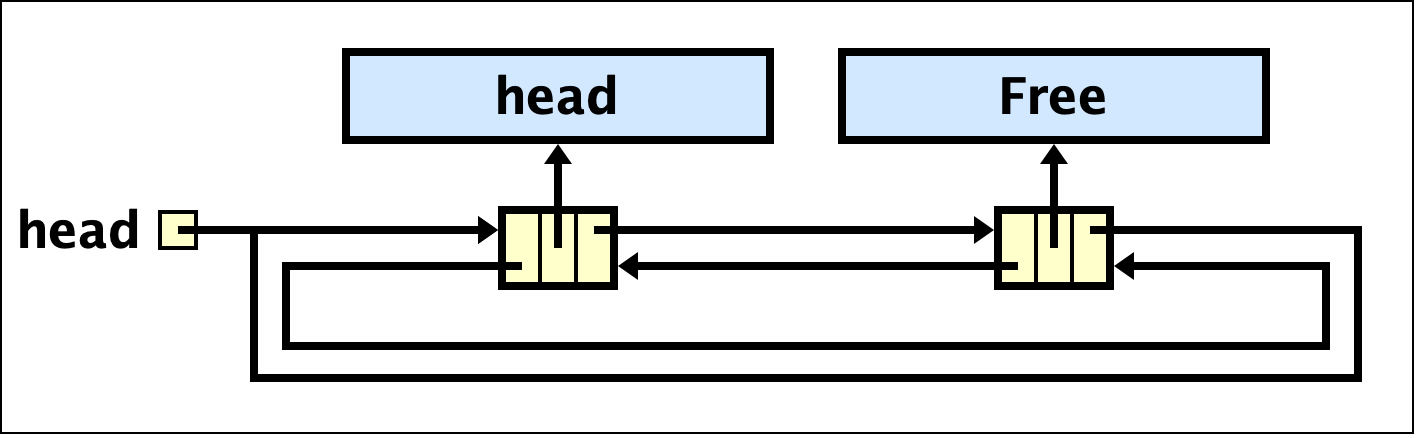


Figure 2: Diagram of the constructor

## requestMemory() method

For the requestMemory() method, we designed a while loop to loop through each memory allocation in the manager until the requester is inserted or find out that there is no space for the new requester. To start searching, the first allocation we need to check is the one next to the head, so we created a memory allocation called mem and set the mem to head.next. If the current allocation is not available for inserting the new requester, the mem will be set to mem.next at the end of the while loop. The advantage of a circular linked list, in this case, is that when the current allocation is head, we know that we already loop through every allocation in the manager and can’t find one that fits the new requester, so we can just stop the while loop and return null. Therefore, the condition of the while loop is that the mem is not equal to head, when the condition is false (mem==head), we can return null. To insert a new requester, the allocation has to fulfill two requirements: owner is “Free”, len is bigger or equal to the size of the new requester. Inside while loop, we have two if statements to identify two situations:

1. len is bigger than the size of the new requester and mem’s owner is “Free”
2. len is equal to the size of the new requester and mem’s owner is “Free”

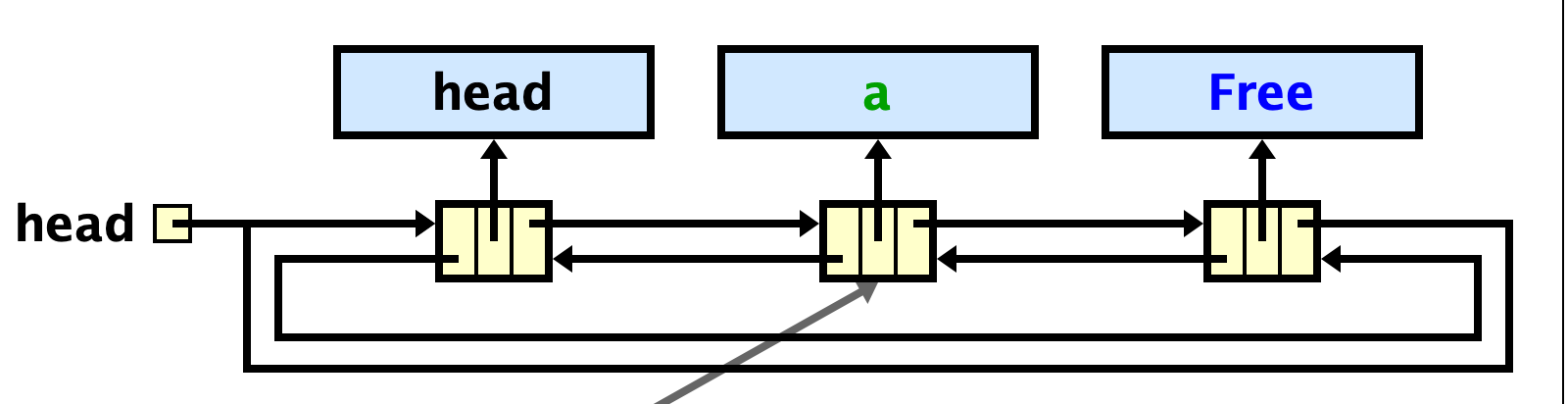


Figure 3: visualization of len is bigger than the size of the new requester

When len is bigger than the size of the new requester and the owner is “Free”, the strategy is to create a new memory allocation for the requester and reset the length of the mem. To visualize this strategy would be we split the free size into two, the first part is allocated by the requester, and the second part is still free but with shorter size and different position. Because of the new memory allocation is inserted before the current mem, the position is the same as the mem’s position; owner and size would be the corresponding parameters. After creating the new allocation we can set links to put it in the manager. The new allocation’s previous allocation is the mem’s previous allocation and the next allocation is the mem.

When len is happened to equal to the size of the new requester and the owner is “Free”, the only thing we need to do is change the mem’s owner into the new requester.

## returnMemory() method

For returnMemory() method, we need to set the memory allocation object, mem, that passed into as a parameter to free. We have four if statements in the implementation to identify four situations:

1. The mem’s previous and next allocations are both free, we need to merge three of them
2. The mem’s next allocation is free, we need to merge two allocations
3. The mem’s previous allocation is free, we need to merge two allocations
4. The mem’s previous and next allocations are both allocated, no need to merge free spaces in this situation

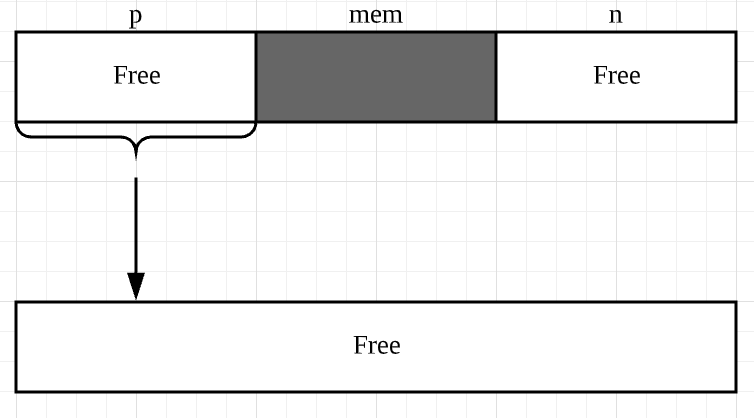


Figure 4: visualization of situation 1

In situation 1, mem is the memory allocation we need to deallocate. p is the previous allocation of mem; n is the next allocation of mem. We need to merge p, mem, and n into one big free spaces, the best strategy is to modify p and delete mem and n, because modifying p only need to reset p’s size and p’s next allocation. The new size of p is the sum of the size of three allocations. To merge three of them, we link p to the next allocation of n.

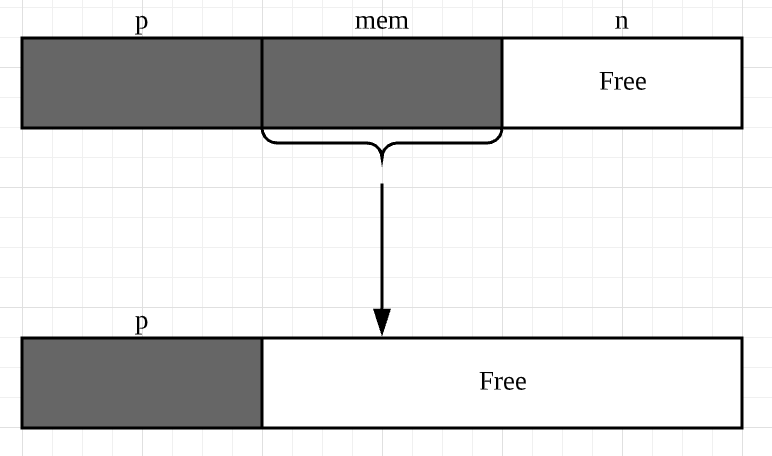


Figure 5: visualization of situation 2

In situation 2, mem is the memory allocation we need to deallocate. p is the previous allocation of mem; n is the next allocation of mem. We need to merge mem and n into one big free spaces, the best strategy is to modify mem and delete n, because modifying mem doesn’t need to reset position. The new size of mem is the sum of the size of itself and n, and new owner will be “Free”. To merge two of them, we link mem to the next allocation of n.

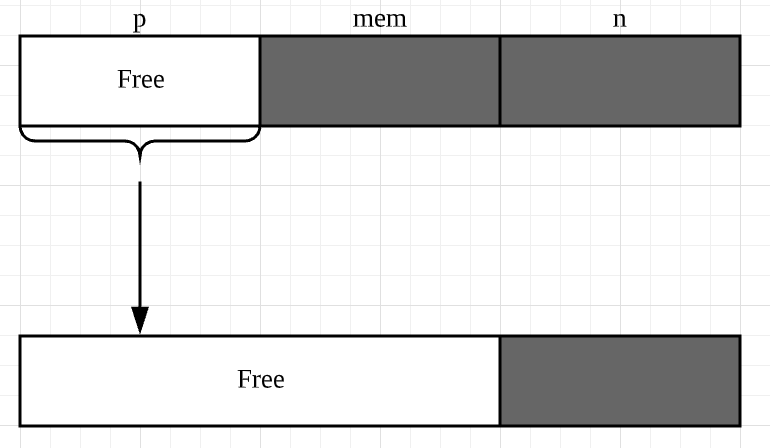


Figure 6: visualization of situation 3

In situation 3, mem is the memory allocation we need to deallocate. p is the previous allocation of mem; n is the next allocation of mem. We need to merge mem and p into one big free spaces, the best strategy is to modify p and delete mem, because modifying p doesn’t need to reset position. The new size of p is the sum of the size of itself and mem. To merge two of them, we link p to n.

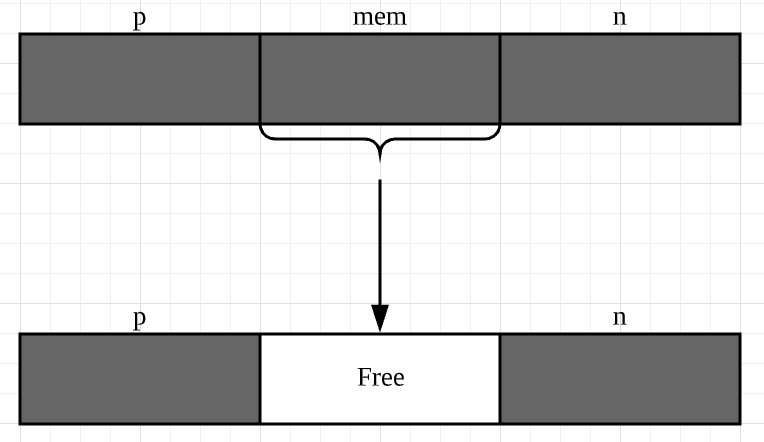


Figure 13: visualization of situation 4

In situation 4 is the simplest one, the only thing we need to modify is the mem’s owner. Since it is deallocated, we reset its owner to free is enough.

# Analysis

Not only to mention to applicability of the code, but to highlight the key trait that is consistent with each of the previously mentioned situations; the code provides opportunity for a wide variety of sequence operation. The double-linked list ADT is an extremely powerful data structure by which, when implemented correctly, allows the programmer to create a valuable piece of memory in which can be manipulated to the extreme. To represent this idea, imagine a collection of contacts in your address book. The literal book is a representation of our Memory Manager in which you have the “operations” to write, add, remove, and search through your contacts. The usefulness of this structure can not be more effectively portrayed after saying the possibilities with such a memory space are limitless within reason.

From the data that was uncovered, patterns began to take form, allowing us as the programmers to deeply understanding the inner workings of each operation. Each operation works with one another by simply using effective single lines of code. This further developed our code to be as elegant as we believe was possible with our circumstance.

Through analysis of our data, the elegance of the code became visualized as the MemoryManager used MemoryAllocation to connect each new double linked node when allocated to a memory space, and used ReturnMemory to connect nodes when removed from a memory space. One of the most notable forms of “elegance” in our code would be our usage of overriding links between nodes using the “garbage” system to remove the deallocated nodes from memory. Anytime a new piece of memory is needed to be returned, we simply altered pre existing free space to access the memory space being removed and increase the free space associated with what was removed.

Alternatively, we could have used recursion to implement the same result as our iterations. Using observation the literal code for recursion would be other fewer lines of code. However, with this information it is important to consider the run time for the iteration/recursion methods to truly determine if one is better/more efficient than the other. The run time for both recursion and iteration is the same in our scenario. Θ(n) would be the run for both methods since the code would have to search through the memory to identify free space, then return an expression with the location of where newly allocated memory could be placed. For ReturnMemory, the run time would be Θ(1) since there is no search for the target linked node, but rather an alteration of the links associated with the returned memory space.

Respectively, using iteration was very efficient in creating a data structure that keeps track of the usage of memory allocation, and by doing so we deepened our understanding of the double linked list and how it can be applied.

# Conclusion

In this design document, we lay out the purpose of the project and then describe the overall layout of our UML diagram including relationships of classes and objects and how each method is implemented in detail. We also analyzed the time complexity of returnMemory() method and two different ways to achieve the goal of implementing the requestMemory() method. After the lab, we have a better understanding of the structure of the doubly linked list and how does memory manager work.

To sum up, our design can be considered as successful; it passed the unit testing and fulfills all the requirements that are asked.