# Design

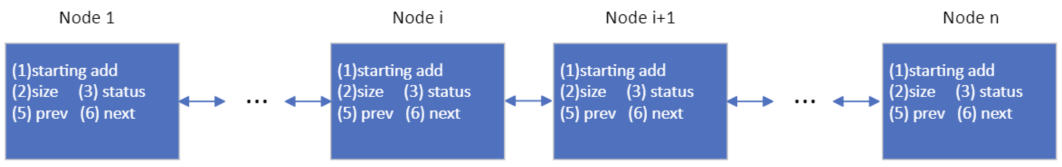
1. Data structures used for implementing the algorithm

## 1.1. Memory Map by linked list

### 1.1.1. Description

The Memory Map is a linked list that contains information on both free and allocated memory blocks in the memory. Each node in Memory Map contains parameters (1) the physical starting address of this block; (2) the size of this block; (3) the occupation status (whether free or occupied); (4) reference to the previous node; (5) reference to the next node. In real world memory maps, there should also be information like ownership and protection, but it is omitted in this project.

The graphical representation is shown below:



### 1.1.2. Supported operations

(a) insert(MemMapNode prev, MemMapNode curr)
Given the reference of the node before the to insert place (prev) and the node to be inserted (curr), node curr can be inserted into the Memory Map.
This operation can be done in O(1) time, because it needs to update at most 4 variables: the forward pointer of prev and curr, and the back pointer of curr and the node after curr.

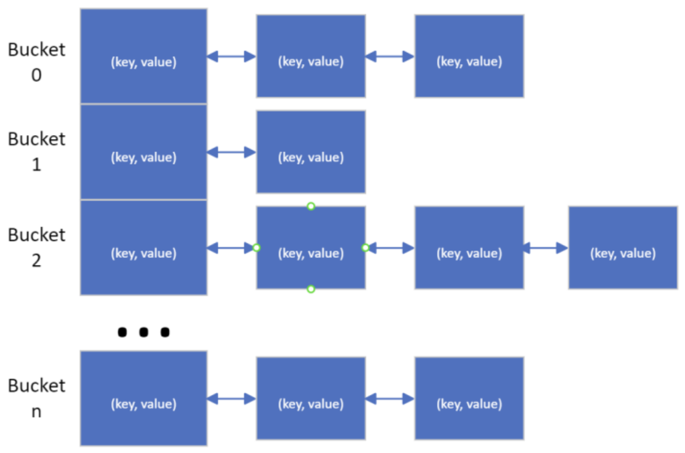
(b) delete(MemMapNode curr)
Given the reference to the Memory Map node (curr) to be deleted, this node can be deleted from memory map.
This operation can also be done in O(1) time, because it needs to update at most 2 variables: the forward pointer of the previous node and the back pointer of the next node.

## 1.2. Hash table pointing to Memory Map

### 1.2.1. Description

Although we can search for each Memory Map node linearly in the Memory Map, it would take O(n) time and would not be efficient enough. As a result, I designed 2 Hash tables of the Memory Map nodes indexing respectively on the starting Address of the Memory Map Node (htOfAddr) and the Size (htOfFreeSize) of the free Memory Map nodes. Since the number of free and allocated nodes will change in the memory managing process, the hash table are designed to expand and shrink automatically. Also the hash table is modified so that it can store multiple nodes with same key.

The graphical representation of the hash table:



1.2.2. Supported operations

(a) put(BigInteger key, MemMapNode memMapNode)
insert a key value pair. For htOfAddr, it will be (starting address, Memory Map Node reference), for htOfFreeSize, it will be (free size, Memory Map node reference)

(b) get(BigInteger key)
the basic report function. Given the key, it reports the first value found in hash table.

(c) getMinAddr(BigInteger key)
Modified get() method.It will loop through the whole bucked of the corresponding key and find the key value pair with the minimum value. It is used in the hash table of free size to find the block of required size with the smallest address.

(d) getPredKey(BigInteger key)
Modified get() method. It will get the key value pair with successor of theq. It is used in the hash table of starting address, so that given an address, the table can find out which memory map block is the address in. This can be done because in the hash table of starting address, the starting address + the size of the block = the starting address of the next block. So search in the hash table for the starting address of the next block until the suitable block is found.

(e) getSuccKeyMinAddr(BigInteger key)
Modified get() method. Similar to getPredKey(), it will will get the key value pair with successor of the key of smallest address. It is used in the BEST_FIT search to find the memory block that suits the requirement best.

(f) getMaxBlockSize()

It is used in the hash table of free size to return the maximum key (which is the max free size).

(g) remove (BigInteger key, MemMapNode memMapNode)
to remove the hash table element with the given key and value. Since my modified hash table can store elements with same key, we need to know both the key and the value to delete an element.

2. Explanation of algorithm

2.1. Request

2.1.1. validation

In the validation process, if the request has a target address, then we only need to check the locate the memory block of the address by htOfAddr.getPredKey() method, and validate if there is enough space to allocate.

If the request has no target address, then we have 3 different kinds of strategies. For FIRST_FIT, we search through the hash table of address with the similar way mentioned in getPredKey(BigInteger key) method in hash table. For BEST_FIT, we can directly use the method htOfFreeSize.getSuccKeyMinAddr() to search for the target memory block. For WORST_FIT, use htOfFreeSize. getMinAddr() to search for the free memory block with the max free block size, which is a variable that is maintained through the algorithm. I the three kinds of operations, if we successfully find a target memory block, then it means that the operation is valid and we can go to the allocation process. Otherwise it is an invalid request.

2.1.2. Allocation

In the allocation process, up to 3 new memory blocks will be generated : the newly allocated block N2, the free block N1 before N2 (if exists). We update the Memory Map (memMap) and the hash tables of starting adrress (htOfAddr) by inserting N1, N2, N3 and deleting the original memory block from them. And for the hash table of free size, we only need to delete the original block and insert N2. The update of max free block size is only performed when the original block is the biggest free block.

2.2. Release

In my definition, the release is only invalid when all memory in the target range is free. So we do not need a validation step before releasing, instead if we found that all memory is free after searching through the whole loop, we naturally find it invalid.

So in the release process, we first find the Memory Map block the target releasing starting address is in by htOfAddr.getPredKey().

Looping through all the memory blocks in the range (starting address of the the Memory Map block the releasing starting address is in, ending address of release), the big free block's starting address and size is calculated, and is inserted in memMap, htOfAddr and htOfFreeAddr. If the size is bigger than the max size of the free block then the max size of the free block is updated.

All the blocks that fully overlap with the big new block is deleted from memMap, htOfAddr and htOfFreeAddr. And before and after the newly inserted free block, 2 partial allocated blocks which are the sub-block of the partially overlapped original blocks might also be inserted depending on the inserting address and the allocation status of the partially overlapped original blocks.

## Evaluation

1. Trade offs and modification on the given structure
   1.1. Adding an extra memory map
   The memory map is designed to be a linked list, so that form each memory block we can get information about its previous and next memory block in O(1) time. If we don't have this, although there might be a decrease in memory size, but the change trivial (both O(n) before and after change) and accessing information of neighbor blocks will become a lot harder.
   1.2. Merging 2 hash tables of starting address into one
   I the sample method, hash table of starting address of free blocks and hash table of starting address of allocated blocks are separately created. In my design, I merged them into one because (1) it can save some place and make the algorithm and data structure simpler to understand and implement (2) if the starting address of all blocks are placed in one hash table, then for any memory block B, the key B.startingAddr + B.size (which is the starting address of the next memory block) is definitely in the hash table, this makes methods involving search the hash table of address like getPredKey() and getSuccKeyMinAddr() quicker. But for the FIRST_FIT, we are unable to only search for starting address of free memory blocks, which is a drawback of this modification.
   1.3. Making the hash table dynamic
   Since the total number of memory blocks is changing and are hard to estimate the value, I think the hash table should be made dynamic so that when the elements inside doubles we double the number of buckets in the hash table and when the elements inside halves we half the number of buckets in the hash table.
   This can make the amortized time of insertion, deletion and searching to be O(1).
   1.4. Keeping the max free block size in the WORST_FIT search
   In WORST_FIT search, we need to know the size of max free block so we can efficiently search it in the hash table of free size. If every time we search we loop through the table it will be very inefficient. But if we maintain a variable maxFreeBlockSize, we only have to update it when we delete a block with max size (then we need to loop through the hash table of free size and get the new maxFreeBlockSize: O(n)) and when we add a new block whose size

is bigger than the maxFreeBlockSize (for this situation, we only need to assign maxFreeBlockSize to the new size, so the complextity is O(1)!)

2. Space complexity analysis

   There are only three data structures maintain in the whole process, which are the Memory Map (linked list), the hash table of starting address and the hash table of free size. When there are n memory blocks handled by the memory manager, both linked list and hash table use O(n) space. Therefore the total space complexity is O(n).

3. Time complexity analysis

   3.1. Request

   3.1.1. Validation

   For the request with address, first we need to find the memory block with containing the target address. This is done by htOfAddr. getPredKey(target_address) which involves looping through the hash table of address. htOfAddr contains O(n) elements so the time for this step is O(n)

   For request without address, there are FIRST_FIT, BEST_FIT, WORST_FIT.

   For FIRST_FIT, it involves looping through the hash table of address to find the first free block of enough size. In worst case, there is no such block, so the complexity is the same with the number of elements in the hash table so it is O(n).

   For BEST_FIT, we can use htOfAddr.getSuccKeyMinAddr(target_size) to get the best fit memory block with the smallest address. The process also involves looping through the whole hash table so time complexity is O(n).

   For WORST_FIT, we first need to get the max free block size. This requires only O(1) time since the variable maxFreeBlockSize is maintain throughout the algorithm.

   3.1.2. Allocation

   For the allocation process, we need to insert and delelte O(1) number of elements in *memMap* and in 2 hash tables. Since we already know the reference to the node in the Memory Map and hash table is dynamic, so each insertion and deletion also takes only O(1) time (amortized for dynamic hash table), so the total time is O(1). But for WORST_FIT, if the maxFreeBlockSize is updated in the process, we need to update it by looping through the hash table of free size, so the complexity goes back to O(n). However, since we do not need to update the maxFreeBlockSize every time in real practice, it stills has a very high efficiency.

   3.1.3. Combined and amortized

   So combining validation and allocation, request takes and amortized O(n) time.

## 3.2. Release

When first finding the Memory Map block the target releasing starting address is in by htOfAddr.getPredKey(), we need to search the whole hash table of address and it takes O(n) time.

Then for looping through all the memory blocks in range, the big free block's starting address and calculating the new size of the freed and partially freed blocks, the number of blocks needed loop through is the number of blocks in the range of release operation. So in n operations we altogether need to loop through at most n blocks so the amortized number of blocks to go through is O(n) so for each operation, the amortized blocks we need to go through is O(1) so the amortized time complexity is O(1).

For updating the Memory Map and the 2 hash tables, as discussed in 3.1.2 each update (deletion and insertion) of a memory blocks requires O(1) time, and for an amortized O(1) blocks to update, the time complexity is O(1).

Similar to the update of the maxFreeBlockSize in request, each update requires O(n) time, but the frequency is much smaller than finding the maxFreeBlockSize every time when we do WORST_FIT validation.

So the total time complexity is O(n).

## 3.3. Amortized complexity

Since the amortized time complexity of request and release is both O(n), the amortized time complexity for the whole process is still O(n).