I implemented the specified methods (initHeap(), myMalloc(), myFree(), and print()) and variable (a linked list representing the heap; “memory”) in my HeapManager class. I also provided a private method (coalesce()) to aid myFree() in coalescing adjacent free spaces. I checked the correctness of my implementation using test.cpp, which [when compiled] takes the sample run number as an input on the terminal and runs the operations corresponding to the relevant run in the guiding document on an initialised heap. I did not replicate Sample Run #4 exactly, since random numbers were involved, but I ran it multiple times and checked the outputs.

To represent the memory, I used list<node> from <list>, where node is a struct I created consisting of ID, size, and index. I implemented default and parameterised constructors as well as a == operator for the struct for better convenience.

I used one lock (a private variable called “lock”) for synchronisation. Furthermore, I used it only within myMalloc() and myFree(), as initHeap() and print() are supposed to be used outside threads from what I understand from the sample outputs (except for printing within myMalloc() and myFree() in the case of print(), which is already handled by my lock), and coalesce() is only called internally and thus protected by the lock as part of myMalloc() and myFree().

Both myMalloc() and myFree() obtain the lock before doing anything else, and both release it right before returning (either successfully or unsuccessfully). Therefore, each thread makes sure that it is the only running thread before modifying any shared variable (that is, using myMalloc() or myFree()). This is why my class works as expected with multiple threads; atomicity is guaranteed by preventing data races for the heap with a mutex lock. Since I use a single lock to synchronise everything and in a way that results in what is basically a sequential run, my approach is coarse-grained. The class could be made more efficient by fine-grained locking (i.e. locking nodes instead of the method).

Below, you can find the pseudocode for myMalloc(), myFree(), and coalesce():

* myMalloc(int ID, int size):
  1. The lock is obtained.
  2. The heap is traversed.
     + If a node with ID = -1 and size bigger than *size* is found:
       1. The node for the remaining free space is created with size node\_size – *size* and index node\_index + *size*.
       2. The node for the remaining space is inserted right after the current node.
       3. The current (previously free) node is updated with *size* and *ID*.
       4. The text for successful allocation is printed, the lock is released, and the index of the allocated node is returned.
     + Otherwise, nothing is done.
       - If the list was traversed from the beginning to the end and still no space was found: the text for unsuccessful allocation is printed, the lock is released, and -1 is returned.
* myFree(int ID, int index):
  1. The lock is obtained.
  2. The heap is traversed.
     + If a node with *ID* and *index* is found:
       1. The ID of the node is set to -1.
       2. coalesce() is called.
       3. The text for successful freeing is printed, the lock is released, and 1 is returned.
     + Otherwise, nothing is done.
       - If the list was traversed from the beginning to the end and no such free node was found: a warning about unsuccessful freeing is printed, the lock is released, and -1 is returned.
* coalesce():
  1. A list of nodes to remove (delete\_list) is initialised.
  2. The heap is traversed.
     + If the index of a node is the sum of the previous node’s index and size:
       1. The size of the node is incremented by the previous node’s size.
       2. The index of the node is changed to the previous node’s index.
       3. The previous node is pushed to delete\_list.
     + Otherwise, nothing is done.
  3. delete\_list is traversed.
     + If an element in delete\_list is in the heap, it is removed from the heap.
     + Otherwise, nothing is done.