# Algorithms 2024/25: Tick 2

Your task is to implement a form of Doug Lea's Malloc Algorithm in Java. The use-case is to support a program that will use your implementation to create a single, long array of primitive integers (type `int[]`), to be used by variable-sized chunks of integers that need to be allocated within it. Doug Lea's linked list pointers (prev/next) can be represented as the array indices where neighbouring list cells begin.

Your class, `Storage`, will manage free/busy space within an array and offer three methods:

1. `void initialise(int length)` – should create a new `int[]` with the specified length, and initialise any elements of the array that your solution needs to represent an initially empty storage space (see slide "The free/busy list"). This method should throw an `OutOfMemoryException` if it is unable to allocate an array of the specified size.
2. `int malloc(int numInts)` – provided `initialise(..)` has previously been called on your `Storage` object, `malloc(..)` should find a large enough region within the array to hold `numInts`-many integers (plus whatever overhead your linked list cells require), and leave the remaining space in the array free for future allocations. The allocated space should be marked as busy, preventing it from being used to satisfy future calls to `malloc(..)`. The method should return the array index where the allocated chunk begins (N.B.: *not* the start of your linked list cell; the start of the space where the user can store their `numInts`-many integers). If allocation was impossible, your method should return -1.
3. `void free(int index)` – given a (non-negative) index that was previously returned by `malloc(..)`, this method should mark the space as free, such that a future call to `malloc(..)` can reuse the space (perhaps together with neighbouring free space) to satisfy future allocation requests. (See slides "Freeing up memory: FREE(p) [1]" and "[2]".)

Your implementation should not require any libraries or external code, and should succeed in finding free space whenever it is reasonable to do so – e.g. you cannot mark the entire space as busy, always return -1 from `malloc(..)`, and expect to be awarded this tick!

What to submit:

1. Your Java implementation: a single Java source containing a class that implements the interface provided. Helper functions are allowed and encouraged to make your code readable! You may write unit tests and a class containing a `main(String[])` function in other files but are not asked to submit those.

**Submit >> here <<** You may re-submit if you wish. **The deadline is 12:00 on Fri 14 Mar 2025**.

```
interface Algs202425Tick2 {
  void initialise(int length);
  int malloc(int numInts);
  void free(int index);
}
```

Save this ^^ into Algs202425Tick2.java, then write your solution in Storage.java:

```
public class Storage implements Algs202425Tick2 {...}
```

There is a starred tick available – read on!

Dr John Fawcett, Feb 2025

# Algorithms 2024/25: Tick 2

## Tick 2*

Add the following *lazy-update* heuristic to your implementation.

Each cell in the linked list maintained by Doug Lea's Malloc Algorithm should store the free/busy status of the neighbouring chunks (in addition to what was already stored for Tick 2).

- When `malloc(..)` marks a chunk as busy, you *do* update the copies of its free/busy status in the neighbouring cells.
- When `free(..)` marks a chunk as free, you *do not* update the copies of its free/busy status in the neighbouring cells (which saves CPU cache misses), and you only merge with previous/next chunks if your copy of their free/busy status tells you that they are free.
- As `malloc(..)` walks down the list, it updates any cell's free/busy status flag for the previous node if it does not match what it just saw as it walked over the previous node; and it merges any adjacent free blocks.

**What's going on here?** Because we *lazily* do not update the free/busy flags for the prev/next chunks when freeing a chunk, they can become out-of-date and must only be used as a hint, not as definitive knowledge of the status of the neighbours. In this case, a neighbour marked as 'free' really is free but a 'busy' neighbour could be either free or busy. While `malloc(..)` is walking down the list, it corrects any stale free/busy flags it encounters along the way: this is 'free' because the CPU data cache has already had to load the data in order to walk along the list.

Dr John Fawcett, Feb 2025