Memory allocation, release

- Given our layout/admin data for arenas, heaps, chunks, we need data structures and algorithms to control allocation/release of chunks
- Tries to balance need to minimize memory fragmentation against need to maximize runtime efficiency
- Each arena divides the chunks in its heap(s) into 4(5) categories based on size, different data structures for each

Chunk categories

- Fast: set of linked lists of small chunks of fixed size (e.g. 32-byte chunks, 64-byte chunks, etc), with inserts/removes from ends of list
- Unsorted: list of recently-release chunks, not yet put in appropriate other category. Makes release really quick, just add to unsorted list, moves to “right” category later
- Small: set of bins containing chunks of equal sizes (e.g. search tree whose nodes are doubly-linked lists containing chunks of size N), adjacent chunks can be coalesced into larger chunk and put in other bin

Chunk categories (cont.)

- Large: like small category, except chunks within a list fall into a size range (e.g. 4k-8k) instead of a fixed size

  When handling a request, must search the linked list for that range to find a big enough chunk for the request, and maybe split off the unused part as a new smaller chunk
- Memory mapped: special case for really big request, actually passes request off to OS to be handled there
- Cached: each thread can be given a fixed sized cache of chunks that it can use without going to the general memory pool

Allocate algorithm

- First, see if a suitable chunk is in the cache (use it if so)
- If it’s a really big request, use mmap (memory mapping)
- Check the fastbin, use if available (might top up cache)
- Check the small bins, use if available (might top up cache)
- Go through unsorted, move them to small/large, coalesce where appropriate, if you find a good chunk then use it
- Go through large bins, use if available (and split off unused)
- Go through fastbins, coalese where possible and go back to unsorted
- If we get here we need to split a new chunk off the top chunk

Free algorithm

- Put in cache if space available
- If small enough, put in fastbin
- If mmap’d then unmap it
- If adjacent to another free chunk coalesce them
- If it’s the new top then update top, otherwise put in unsorted
- Note how structure of free chunks (with flags for “is prev free?” and their sizes stored at both ends) allows rapid coalescing of adjacent chunks