15-213 Recitation 11: Malloc

28 March 2016

Ralf Brown and the 15-213 staff

Agenda

- Reminders
- Memory Allocation
- Debugging: Heap Checker
- C Pointer Arithmetic
- Appendix

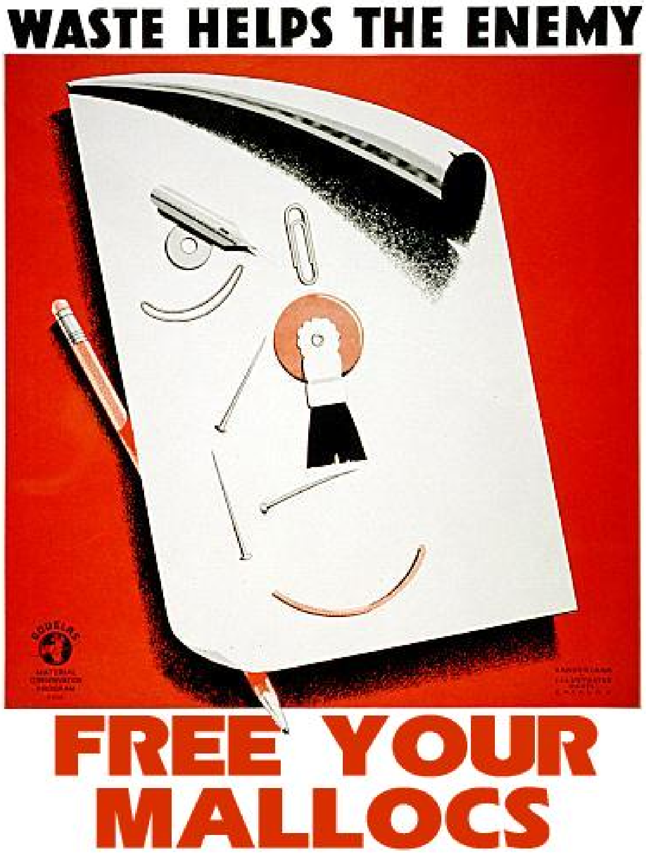


Image: https://nick-black.com/images/gt-coc-propaganda/propaganda/

Reminders

- Malloclab Checkpoint is due on **Thursday**
  - **hard** deadline – no grace days, no late days

Dynamic Memory Allocation

- When do we use it?

Dynamic Memory Allocation

- Used when
  - we don't know at compile-time how much memory we will need
  - when a particular chunk of memory is not needed for the entire run
    - lets us re-use that memory for storing other things
- Important terms:
  - malloc/calloc/realloc/free
  - sbrk
  - payload
  - fragmentation
  - coalescing
    - immediate vs. deferred

Internal Fragmentation

- When does it occur?

Internal Fragmentation

- Occurs when the *payload* is smaller than the block size
  - due to alignment requirements
  - due to management overhead
  - as the result of a decision to use a larger-than-necessary block
- Depends on the current allocations, i.e. the pattern of *previous* requests

Internal Fragmentation

- Due to alignment requirements – the allocator doesn't know how you'll be using the memory, so it has to use the strictest alignment:
  - `void *m1 = malloc(13); void *m2 = malloc(11);`
  - `m1` and `m2` both have to be aligned on 8-byte boundaries
- Due to management overhead:

l	e	n	1	p	a	y	l	o	a	d	1	l	e	n	2	p	a	y	l	o	a	d	2

External Fragmentation

- When does it occur?

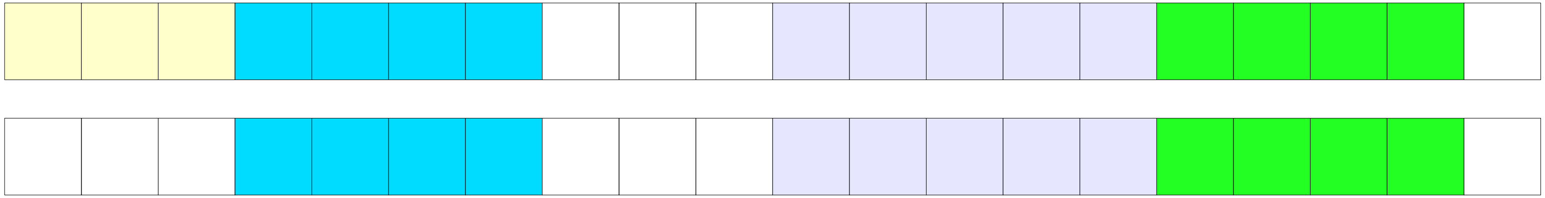
External Fragmentation

- Occurs when the total free space is sufficient, but no single free block is large enough to satisfy the request
- Depends on the pattern of *future* requests
  - thus difficult to predict, and any measurement is at best an estimate

p5 = malloc(4)

free(p1)

p6 = malloc(5)



Oops! **Seven bytes available, but not in one chunk....**

Keeping Track of Blocks

- How do we know where the blocks are?
- How do we know how big the blocks are?
  - remember, free() takes a pointer, but not a size
- How do we know which blocks are free?

Keeping Track of Blocks

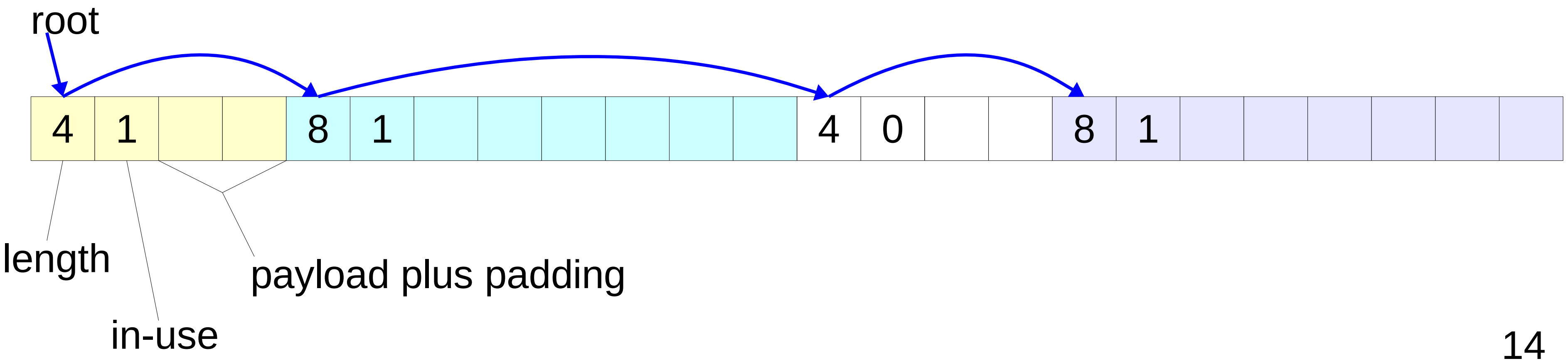
- How do we know where the blocks are?
- How do we know how big the blocks are?
  - remember, free() takes a pointer, but not a size
- How do we know which blocks are free?
- Solution: A data structure to store information about memory blocks
  - Where to keep this structure? We can't malloc() space for it....

Memory-Block Lists

- Common Types
  - Implicit List
    - Root -> block1 ... block2 ... block3 ...
  - Explicit List
    - Root -> free-block 1 -> free-block 2 -> free-block 3 -> ...
  - Segregated List
    - small-malloc root -> free small block 1 -> free small block 2 -> ...
    - medium-malloc root -> free medium block 1 -> ...
    - large-malloc root -> free large block 1 -> free large block 2 -> ...

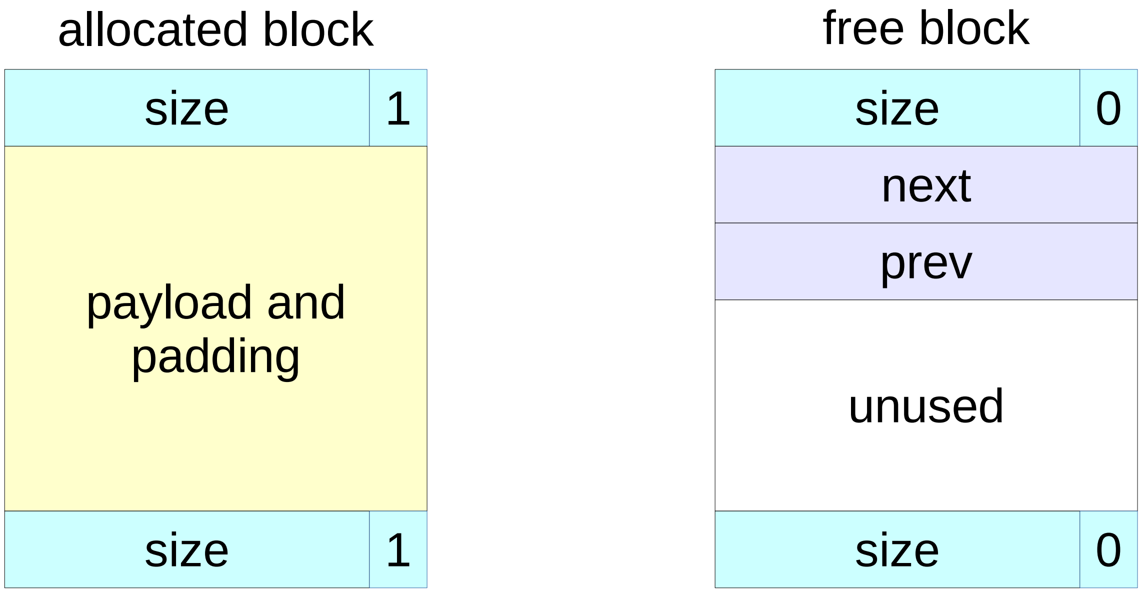
Tracking Blocks: Implicit List

- Use the length field (which is needed anyway) to find the next block
- Scan finds both allocated and free blocks
- Standard trick to save memory – use low bit of length field to store allocation status
  - alignment requirements mean that lowest bit of the length must always be zero



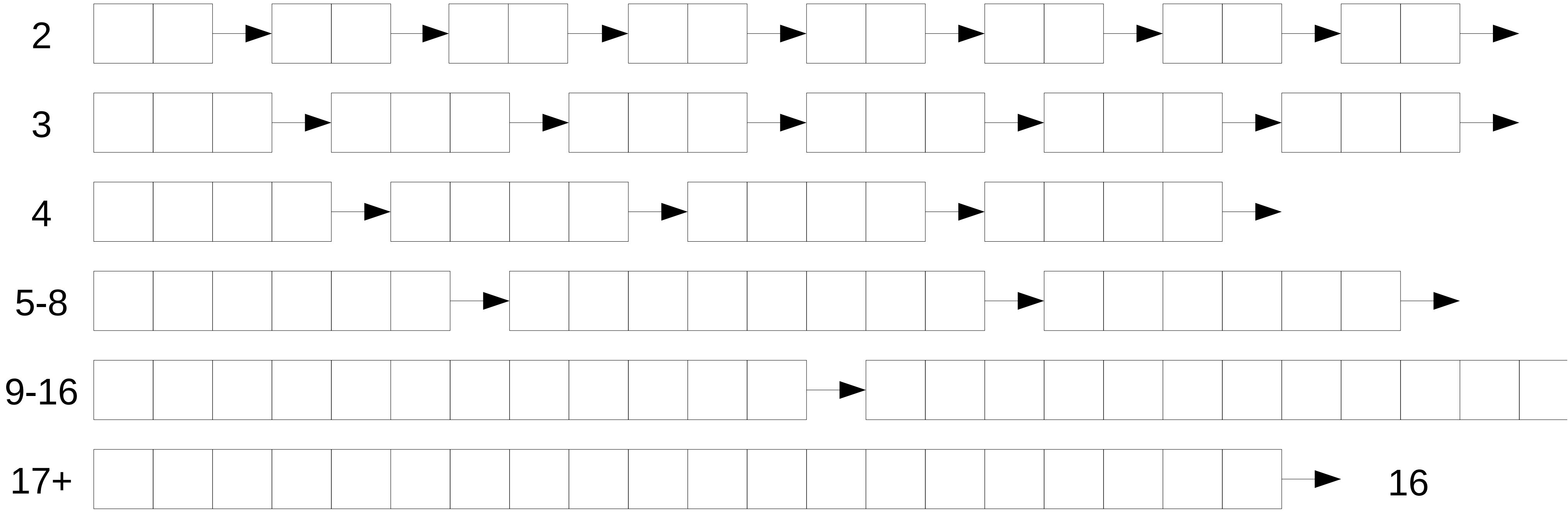
Tracking Blocks: Explicit List

- Maintain a list of *free* blocks instead of *all* blocks
  - means we need to store forward/backward pointers, not just sizes
  - we only track free blocks, so we can store the pointers in the payload area!
  - need to store size at end of block too, for coalescing



Tracking Blocks: Segregated List

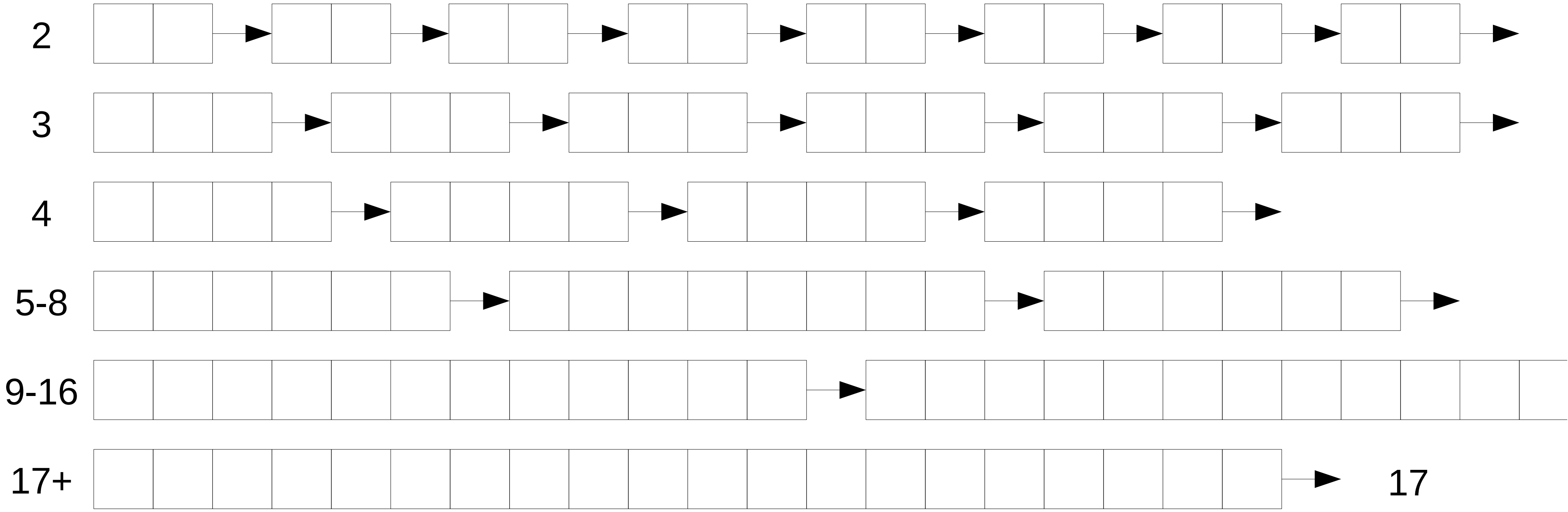
- Keep a separate free list for each *size class* of blocks
  - commonly many different lists for small(ish) block sizes, one class for each power of two for larger sizes, plus a catch-all for really large blocks



Tracking Blocks: Segregated List

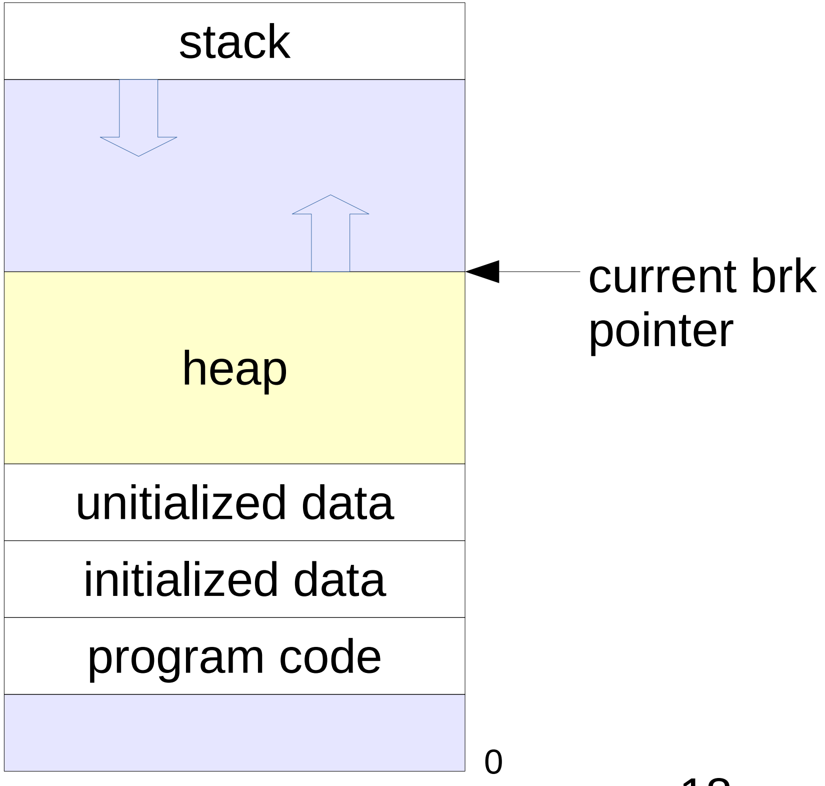
Why is there no list for size-1 blocks?

- Keep a … ach *size class* of blocks
  - comm… sts for small(ish) block sizes, one class for each … power of two for larger sizes, plus a catch-all for really large blocks



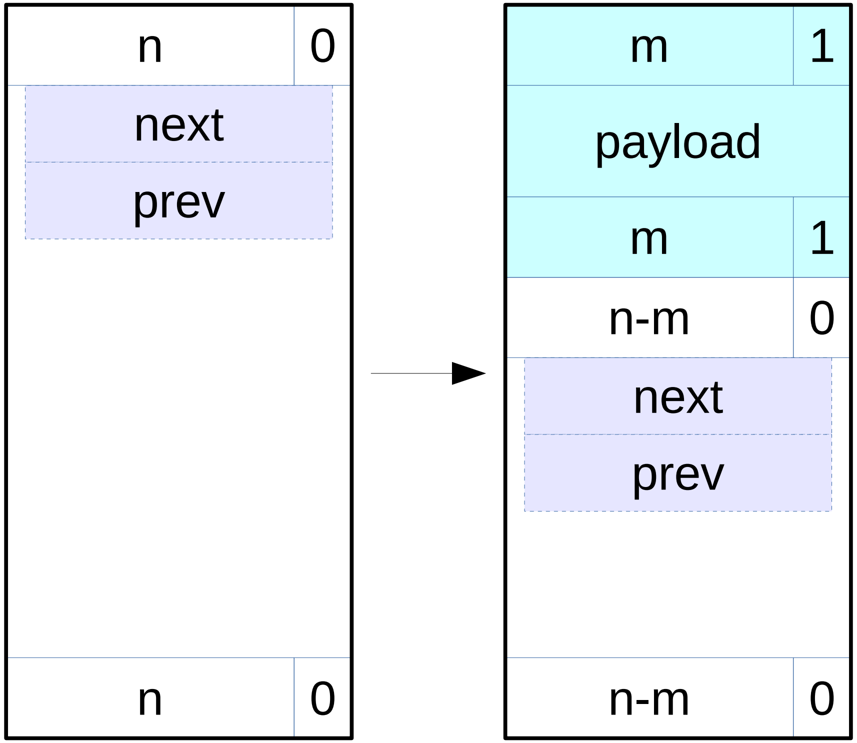
If We Can't Find a Usable Free Block

- Need to extend the heap
  - use the `brk()` or `sbrk()` system calls
  - in Malloclab, use `mem_sbrk()`
    - `sbrk(requested_bytes)` allocates `requested_bytes` of space and returns pointer to start
    - `sbrk(0)` returns a pointer to the end of the current heap
- For speed, extend the heap by a little more than you need immediately
  - use what you need out of the new space, add the rest as a free block

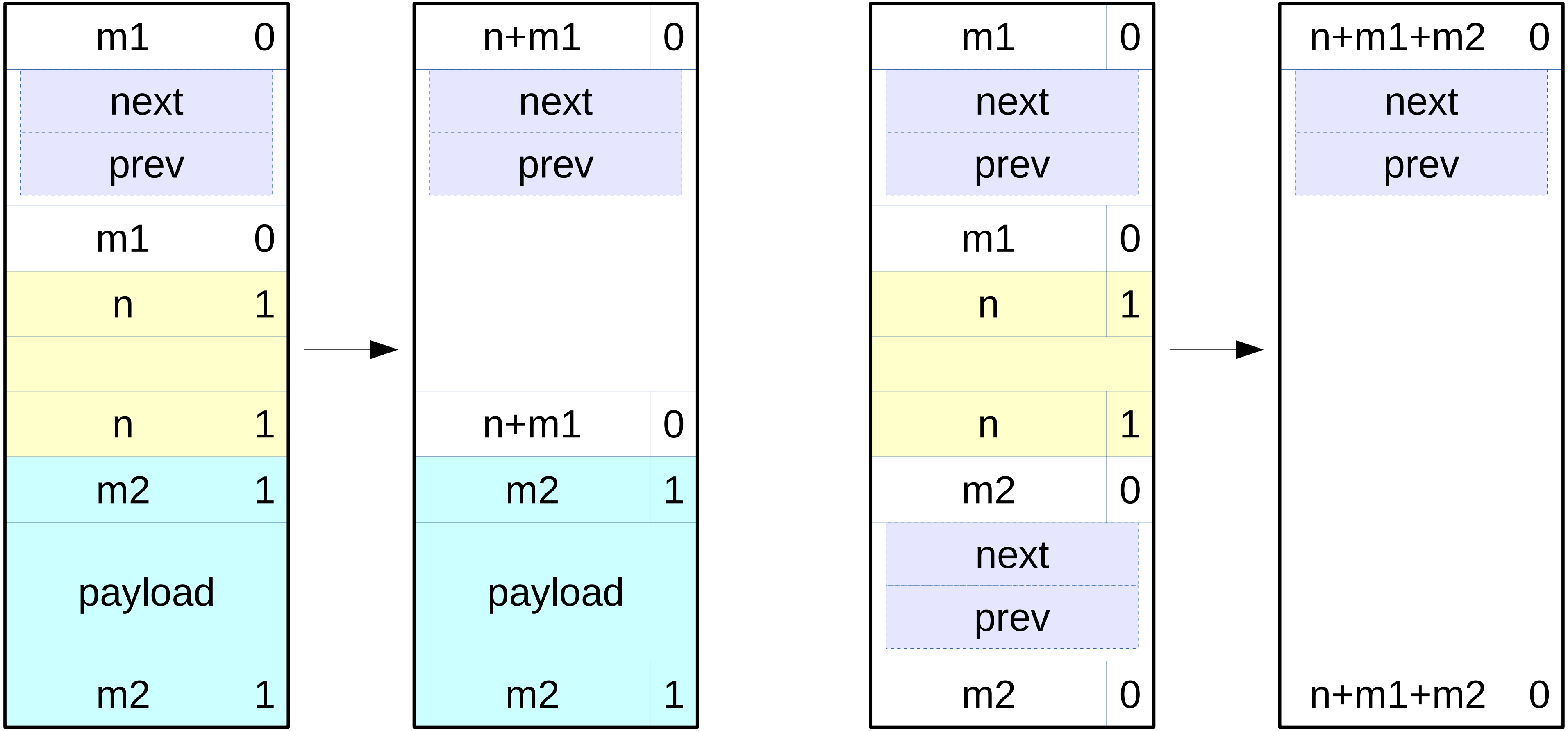


Splitting a Block

- If the block we find is larger than we need, split it and leave the remainder for a future allocation
  - implicit lists: correct the block sizes of the two parts
  - explicit lists: correct previous and next pointers
  - segregated lists: determine the correct size list and insert according to the insertion policy (more on this later)
- When would we **not** split a block?



Coalescing Memory



Design Considerations

- What are some of the design decisions you need to make when implementing malloc()?

Design Considerations

- Finding a matching free block
  - First fit vs. next fit vs. best fit
  - continue searching for a closer fit after finding a big-enough free block?
- Free block ordering
  - LIFO, FIFO, or address-ordered?
- When to coalesce
  - while freeing a block or while searching for free memory?
- How much memory to request with sbrk()
  - larger requests save time in system calls but increase maximum memory use

Heap Checker

- `int mm_checkheap(int verbose);`
- critical for debugging
  - **write this function early!**
  - update it when you change your freelist implementation
  - check all heap invariants (next slide), make sure you haven't lost track of any part of your heap
    - check should pass if and only if the heap is truly well-formed
  - should only generate output if a problem is found, to avoid cluttering up your program's output
- meant to be correct, not efficient
- call before/after major operations when the heap should be well-formed

Heap Invariants (Non-Exhaustive)

- Block level
  - What are some things which should always be true of every block in the heap?

Heap Invariants (Non-Exhaustive)

- Block level
  - header and footer match
  - payload area is aligned, size is valid
  - no contiguous free blocks unless you defer coalescing
- List level
  - What are some things which should always be true of every element of a free list?

Heap Invariants (Non-Exhaustive)

- Block level
  - header and footer match
  - payload area is aligned, size is valid
  - no contiguous free blocks unless you defer coalescing
- List level
  - next/prev pointers in consecutive free blocks are consistent
  - no allocated blocks in free list, all free blocks are in the free list
  - no cycles in free list unless you use a circular list
  - each segregated list contains only blocks in the appropriate size class
- Heap level
  - What are some things that should be true of the heap as a whole?

Heap Invariants (Non-Exhaustive)

- Block level
  - header and footer match
  - payload area is aligned, size is valid
  - no contiguous free blocks unless you defer coalescing
- List level
  - next/prev pointers in consecutive free blocks are consistent
  - no allocated blocks in free list, all free blocks are in the free list
  - no cycles in free list unless you use a circular list
  - each segregated list contains only blocks in the appropriate size class
- Heap level
  - all blocks between heap boundaries, correct sentinel blocks (if used)
- Add your own invariants (e.g. address order)

C: Pointer Arithmetic

- Adding an integer to a pointer is different from adding two integers
- The value of the integer is always multiplied by the size of the type that the pointer points at
- Example:
  - `type_a *ptr = …;`
  - `type_a *ptr2 = ptr + a;`
- is really computing
  - `ptr2 = ptr + (a * sizeof(type_a));`
  - i.e. `lea (ptr, a, sizeof(type_a)), ptr2`
- Pointer arithmetic on `void*` is undefined (what's the size of a `void`?)

C: Pointer Arithmetic

- ```
  int *ptr = (int*)0x152130;
  int *ptr2 = ptr + 1;
  ```
- ```
  char *ptr = (char*)0x152130;
  char *ptr2 = ptr + 1;
  ```
- ```
  char *ptr = (char*)0x152130;
  void *ptr2 = ptr + 1;
  ```
- ```
  char *ptr = (char*)0x152130;
  char *p2 = ((char*)(((int*)ptr)+1));
  ```

C: Pointer Arithmetic

- ```
  int *ptr = (int*)0x152130;
  int *ptr2 = ptr + 1; // ptr2 is 0x152134
  ```

- ```
  char *ptr = (char*)0x152130;
  char *ptr2 = ptr + 1; // ptr2 is 0x152131
  ```

- ```
  char *ptr = (char*)0x152130;
  void *ptr2 = ptr + 1; // ptr2 is still 0x152131
  ```

- ```
  char *ptr = (char*)0x152130;
  char *p2 = ((char*)(((int*)ptr)+1));// p2 is 0x152134
  ```

Beyond Debugging: *Preventing* Errors

- Good coding practices can make your code less error-prone
- What do **you** think are good practices?

Beyond Debugging: *Preventing* Errors

- Good coding practices can make your code less error-prone
- Plan what each function does before writing it
  - consider edge cases – block at start/end of list, single item on list, etc.
  - draw pictures to help you visualize linked lists, memory layout, etc.
- Document your code as you write it
- Encapsulate common operations (e.g. macro to access block header)
- Check for common errors:
  - dereferencing invalid pointers / reading uninitialized memory
  - overwriting memory
  - freeing blocks multiple times (or not at all) / referencing freed blocks
  - incorrect pointer arithmetic

How to Ask for Help

- Be specific about what the problem is, and how to cause it
  - **BAD:** “My program segfaults.”
  - **GOOD:** “On the third free() in trace 4, I get an invalid pointer in my free list while coalescing memory.”
  - Try to figure out which part of the trace file triggers the problem
  - What sequence of events do you expect around the time of the error? What part of the sequence has already happened?
- Have you written your mm_heapcheck function, and is it working?
  - We **WILL** ask to see it!
- Practice asking your rubber duck about the problem (see Recitation 9) before asking a TA or instructor

If You Get Stuck

- ***Please read the writeup!***
- CS:APP Chapter 9
- View lecture notes and course FAQ at http://www.cs.cmu.edu/~213
- Office hours Sunday through Thursday 5:00-9:00pm in WeH 5207
- Post a **private** question on Piazza
- Obtain a rubber duck....

APPENDIX

Dynamic Memory Allocation: Example

p1 = malloc(3)

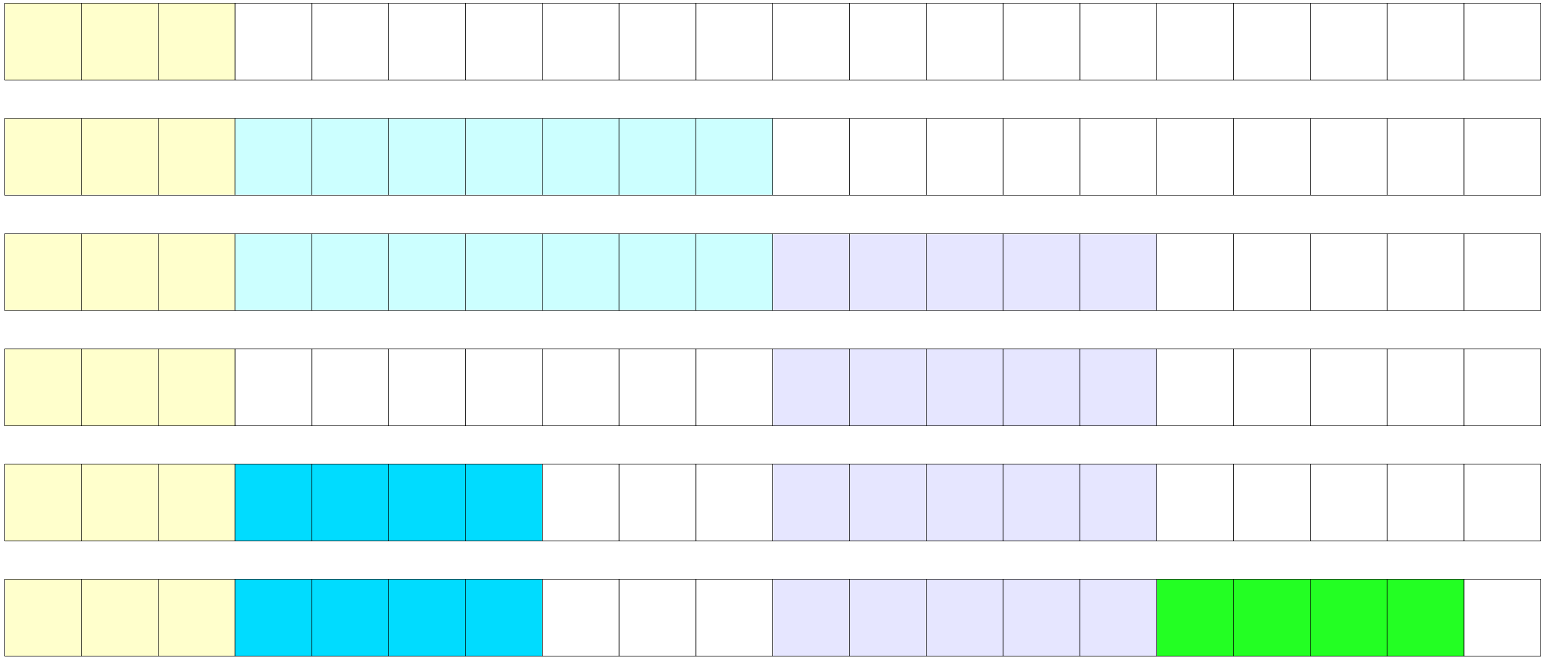
p2 = malloc(7)

p3 = malloc(5)

free(p2)

p4 = malloc(4)

p5 = malloc(4)



The Memory-Block Information Data Structure

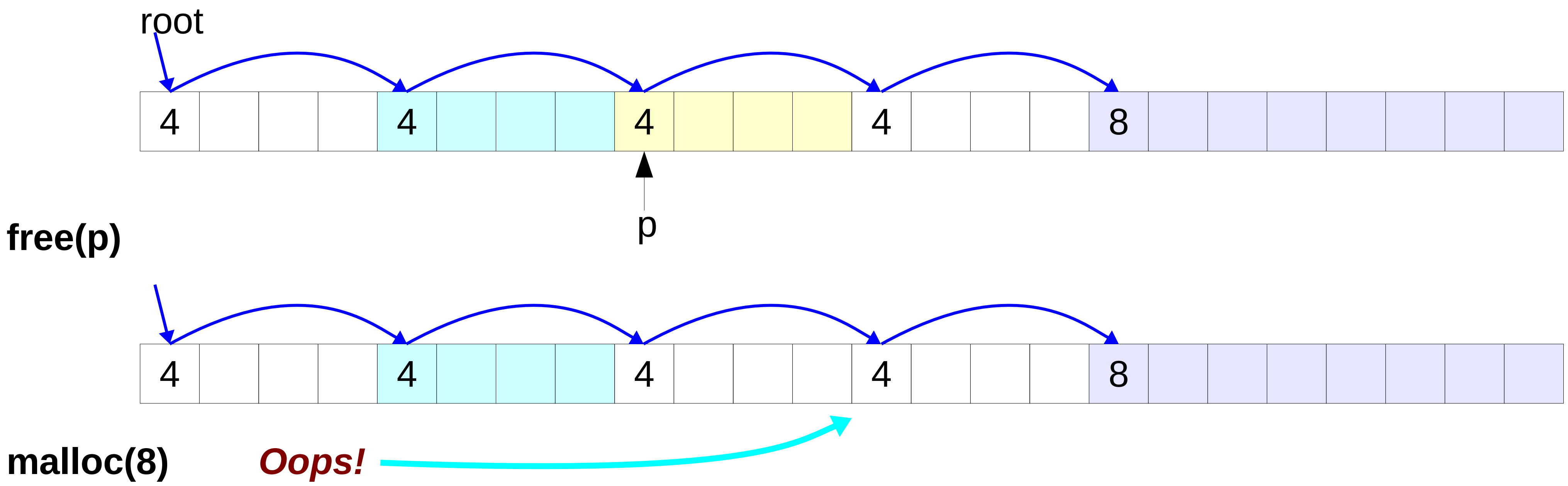
- Requirements:
  - tells us where the blocks are, how big they are, and whether they are free
  - must be able to update the data during calls to `malloc` and `free`
  - need to be able to find the **next free block** which is a “good enough fit” for a given payload
  - need to be able to quickly mark a block as free or allocated
  - need to be able to detect when we run out of blocks
    - what do we do in that case?
- The only memory we have is what we're handing out
  - ...but not all of it needs to be payload! We can use part of it to store the block information.

Finding a Free Block

- First Fit
  - search from beginning, use first block that's big enough
  - linear time in total number of blocks
  - can cause small “splinters” at beginning of list
- Next Fit
  - start search from where previous search finished
  - often faster than first fit, but some research suggests worse fragmentation
- Best Fit
  - search entire list, use smallest block that's big enough
  - keeps fragments small (less wasted memory), but slower than first fit

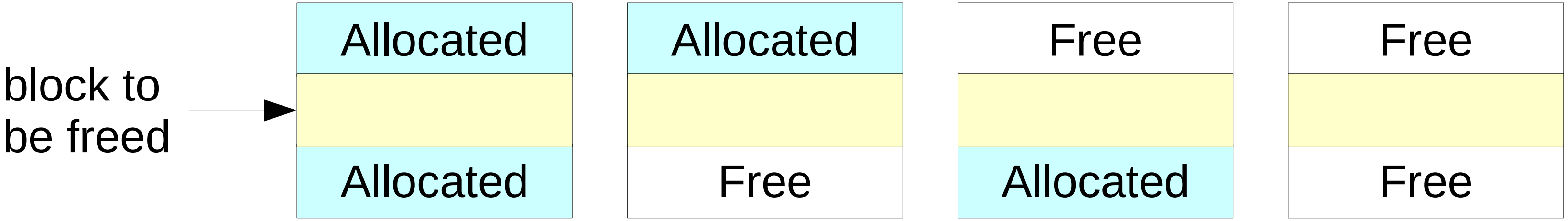
Freeing Blocks

- Simplest implementation is just clearing the “allocated” flag
  - but leads to external fragmentation



Coalescing Memory

- Combine adjacent blocks if both are free
  - implicit lists: look forward and backward using block sizes
    - easily deferred until next allocation request (coalesce while scanning blocks)
  - explicit lists: look forward and backward using block sizes, **not** next/prev
  - segregated lists: look forward and backward using block sizes, then
    - use the size of the coalesced block to determine the proper list
    - insert into list using the insertion policy (LIFO, address-ordered, etc.)
- Four cases:



Insertion Policy

- Where do you put a newly-freed block in the free list?
  - LIFO (last-in-first-out) policy
    - add to the beginning of the free list
    - pro: simple and constant time (very fast)

      `block->next = freelist; freelist = block;`
    - con: studies suggest fragmentation is worse
  - Address-ordered policy
    - insert blocks so that free list blocks are always sorted by address

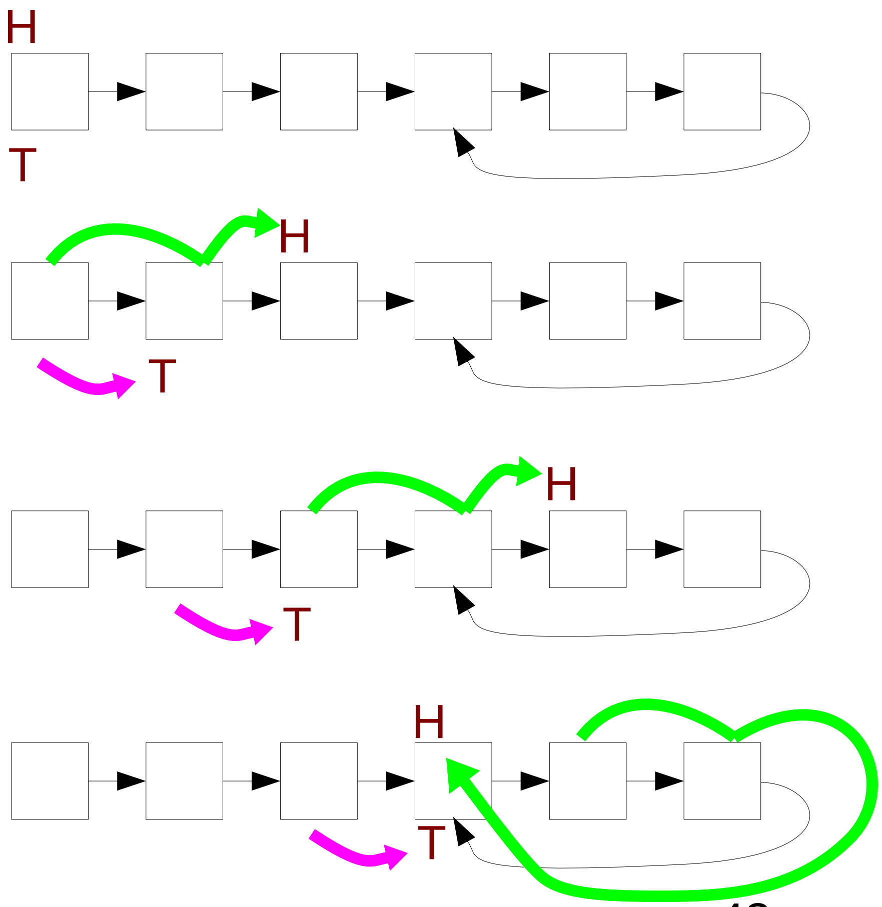
      addr(prev) < addr(curr) < addr(next)
    - pro: lower fragmentation than LIFO
    - con: requires search

C: Pointer Casting

- Notation: `(b*)a` "casts" `a` to be of type `b*`
- Casting a pointer doesn't change the bits!
  - `type_a *ptr_a=...; type_b *ptr_b=(type_b*)ptr_a;` makes ptr_a and ptr_b contain identical bits
- But it does change the behavior when dereferencing
  - because we *interpret* the bits differently
- Can cast `type_a*` to long/unsigned long and back
  - pointers are really just 64-bit numbers
  - such casts are important for malloclab
  - but be careful – this can easily lead to hard-to-find errors

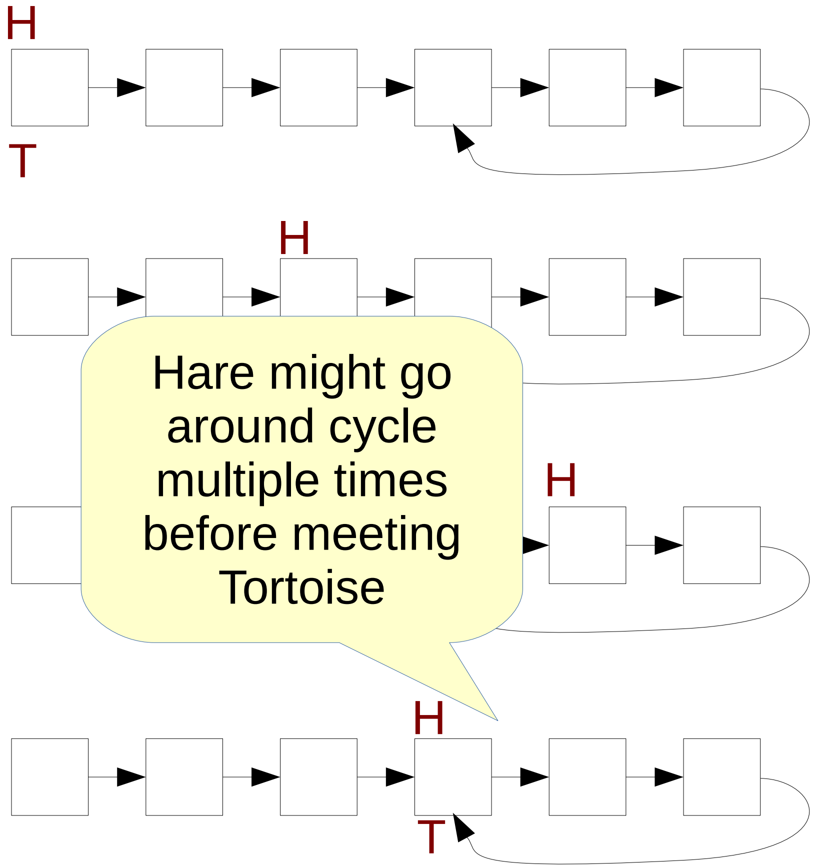
Cycle Checking: Hare and Tortoise Algorithm

- This algorithm detects cycles in linked lists
- Set two pointers, called "hare" and "tortoise", to the beginning of the list
- During each iteration, move "hare" forward by two nodes, "tortoise" by one node
  - if "tortoise" reaches the end of the list, there is no cycle
  - if "tortoise" equals "hare", the list has a cycle



Cycle Checking: Hare and Tortoise Algorithm

- This algorithm detects cycles in linked lists
- Set two pointers, called “hare” and “tortoise”, to the beginning of the list
- During each iteration, move “hare” forward by two nodes, “tortoise” by one node
  - if “tortoise” reaches the end of the list, there is no cycle
  - if “tortoise” equals “hare”, the list has a cycle



Debugging Tip: Using the Preprocessor

- Use conditional compilation with #if or #ifdef to easily turn debugging code on or off

```
#ifdef DEBUG
# define DBG_PRINTF(...) fprintf(stderr, __VA_ARGS__)
# define CHECKHEAP(verbose) mm_checkheap(verbose)
#else
# define DBG_PRINTF(...)
# define CHECKHEAP(verbose)
#endif /* DEBUG */
```

```
void free(void *p)
{
   DBG_PRINTF("freeing %lx\n",(long)p);
   CHECKHEAP(1);
   ...
}
```

Debugging Tip: GDB

- Use breakpoints / conditional breakpoints
  - `break` {address} `if` {condition}
- Use watchpoints
  - like breakpoints, but stop the program when the watched expression changes or location is written
  - `watch {expression}` `watch block->next`
    - break any time the expression changes value; can be extremely slow!
  - `watch -l {expression}` `watch -l *0x15213`
    - evaluate the expression and watch the memory location at that address
    - program runs at full speed if GDB can set a hardware watchpoint
  - `rwatch` to stop on *reading* a location, `awatch` to stop on **any** access