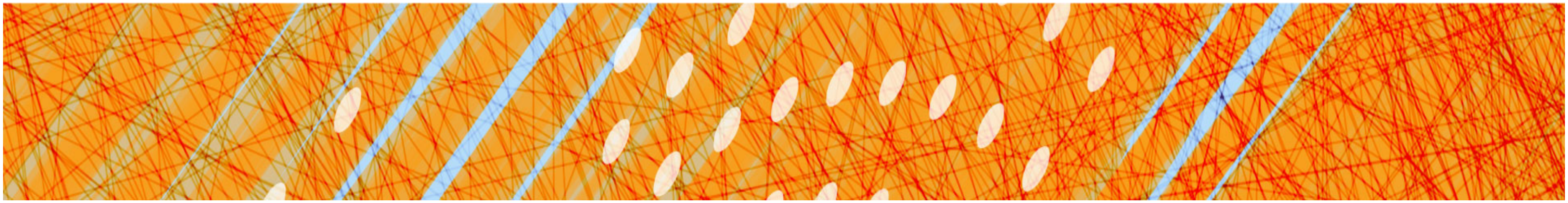


Dynamic Memory Allocation

CS 3410
Computer System Organization & Programming

Note: these slides derive from those by Markus Püschel at CMU.

My favorite kind of cookie is

A. Chocolate Chip
B. Chocolate Chocolate Chip
C. Oatmeal Raisin
D. Snickerdoodle
E. Other

Recommended Approach

```
while (TRUE) {
    code a little;
    test a little;
}
```

Get something that works!

“Premature Optimization is the Root of all Evil”

—Donald Knuth

Today

- Basic concepts
- Basic Implementation
  - Implicit Free Lists
  - Explicit Free Lists
- Implementation Optimizations

Note: there are *many* ways to implement malloc; these slides show the version that most 3410 students have found most intuitive in the past.

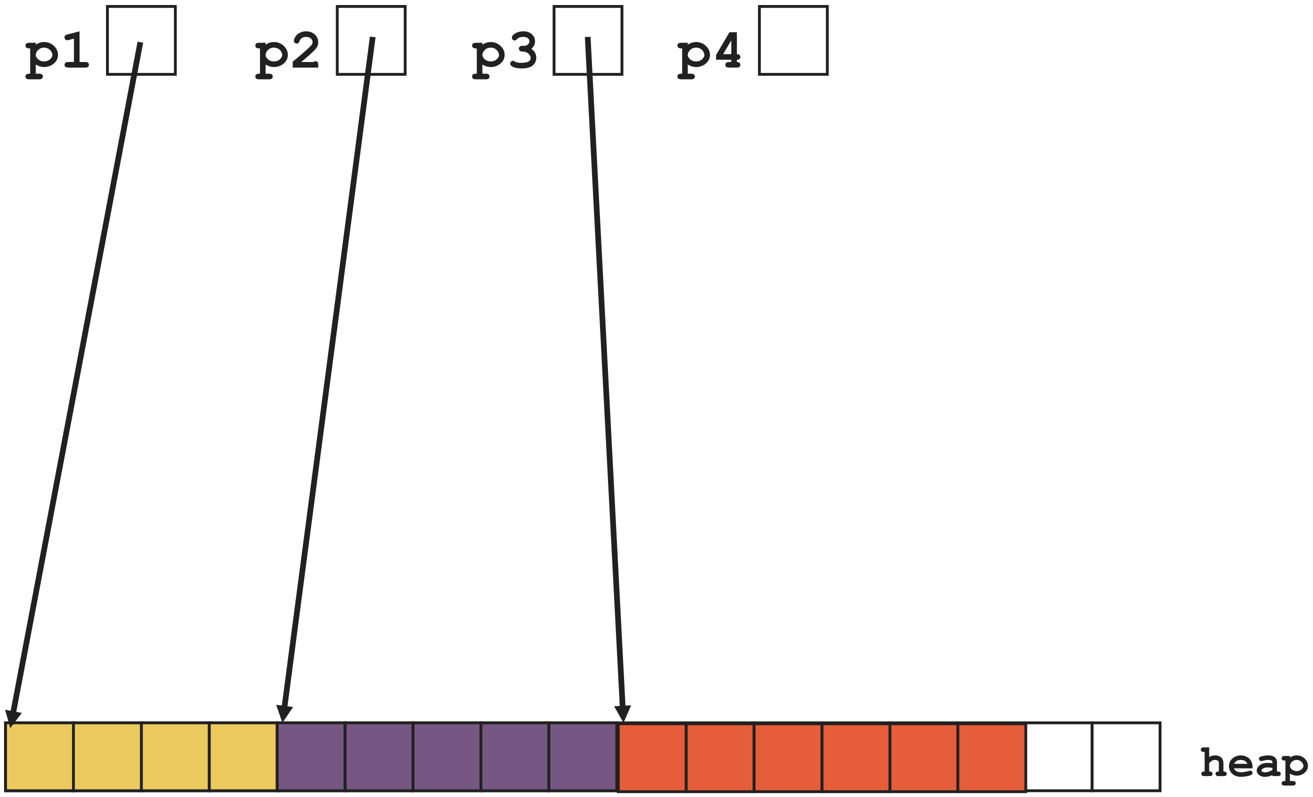
Dynamic Memory Allocation

An allocator:

- maintains the heap as collection of variable sized *blocks*, which are either *allocated* or *free*
  - Some languages free the memory for you (Java, ML, Lisp)
  - Some do not: C

Visualizing Malloc

```
p1 = malloc(16)
p2 = malloc(20)
p3 = malloc(24)
free(p2)
```

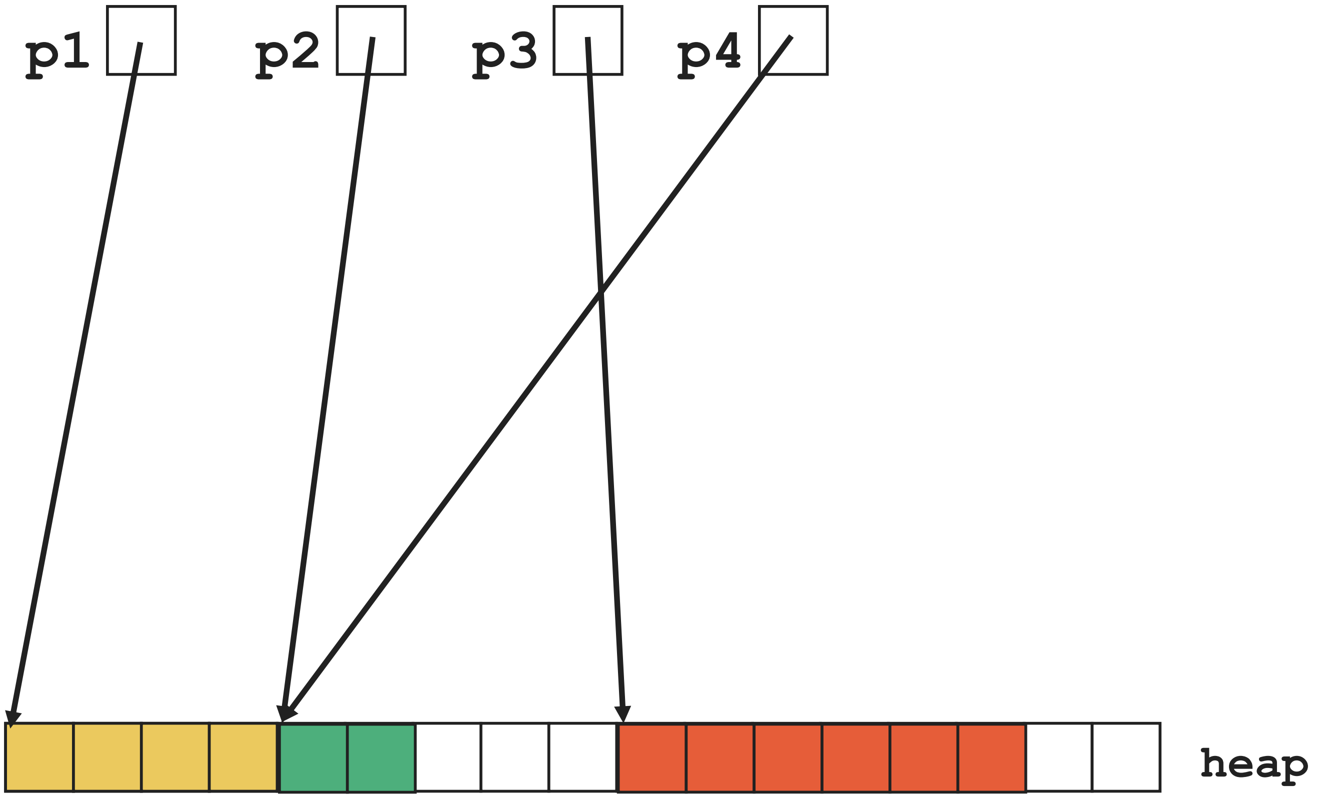


Note: the user should never make any expectations about ***where*** *the block will be allocated with respect to other blocks. That is not part of the library interface.*

To simplify the drawing, **each box is 4 bytes**

Visualizing Malloc

```
p1 = malloc(16)
p2 = malloc(20)
p3 = malloc(24)
free(p2)
p4 = malloc(8)
p5 = malloc(16)
```



Notice p2 still points to its original location after free is called

Fragmentation!

*Note: the user should never make any expectations about **where** the block will be allocated with respect to other blocks. That is not part of the library interface.*

To simplify the drawing, **each box is 4 bytes**

Constraints

Applications (users of malloc)

- Can issue arbitrary sequence of **`malloc`** and **`free`** requests
- **`free`** request must be to a **`malloc`**’d block

Allocators (implementors of malloc)

- Can’t control number or size of allocated blocks
- Must respond immediately to **`malloc`** requests
  - *i.e.*, can’t reorder or buffer requests
- Must allocate blocks from free memory
  - *i.e.*, can only place allocated blocks in free memory
- Must align blocks so they satisfy all alignment requirements
  - 8 byte alignment for GNU **`malloc`** (**`libc malloc`**) on Linux boxes
- Can manipulate and modify only free memory
- Can’t move the allocated blocks once they are **`malloc`**’d
  - *i.e.*, compaction is not allowed

Implementation Issues: 5 Questions

1. How do we keep track of the size of a block?
2. How do we keep track of which blocks are in use and which ones are free?
3. When the request for a block is smaller than the free block we find, what do we do with the extra space?
4. How do we pick a block to use for allocation?
   (if a few work)
5. How do we reinsert freed block?

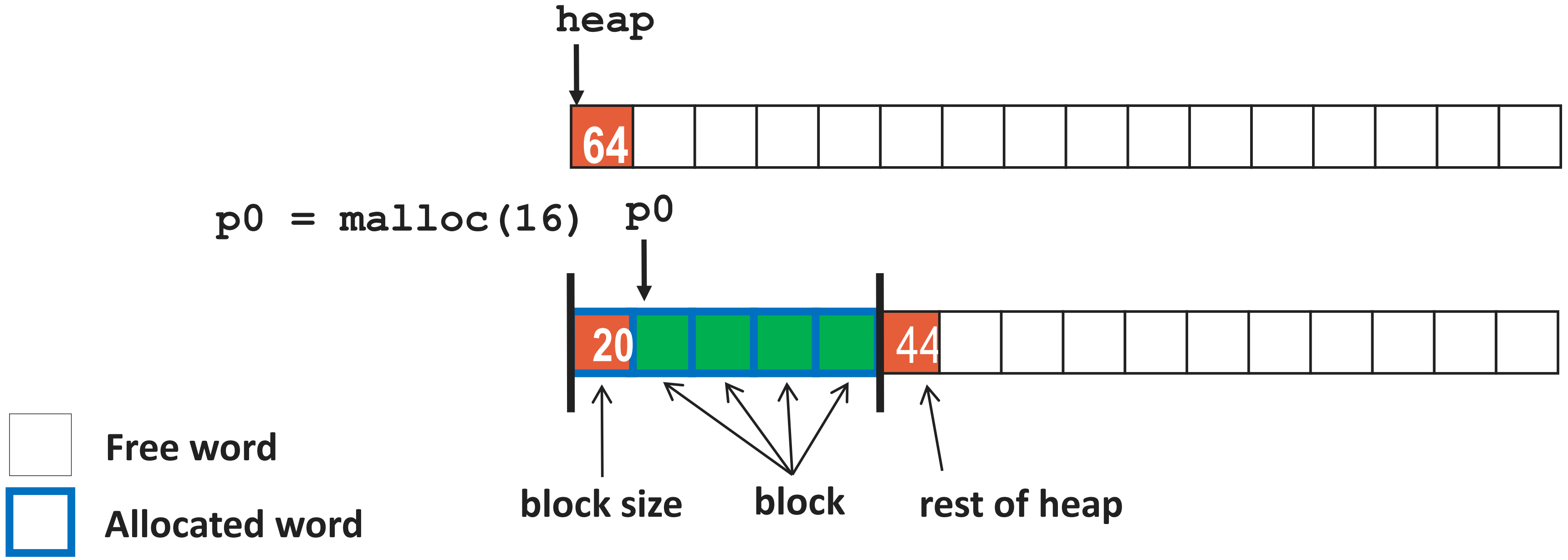
Q1: How big is each block?

Store block length before the block (called the ***header***)

- +1 word for every allocated block

Heap: initially 1 large, free block

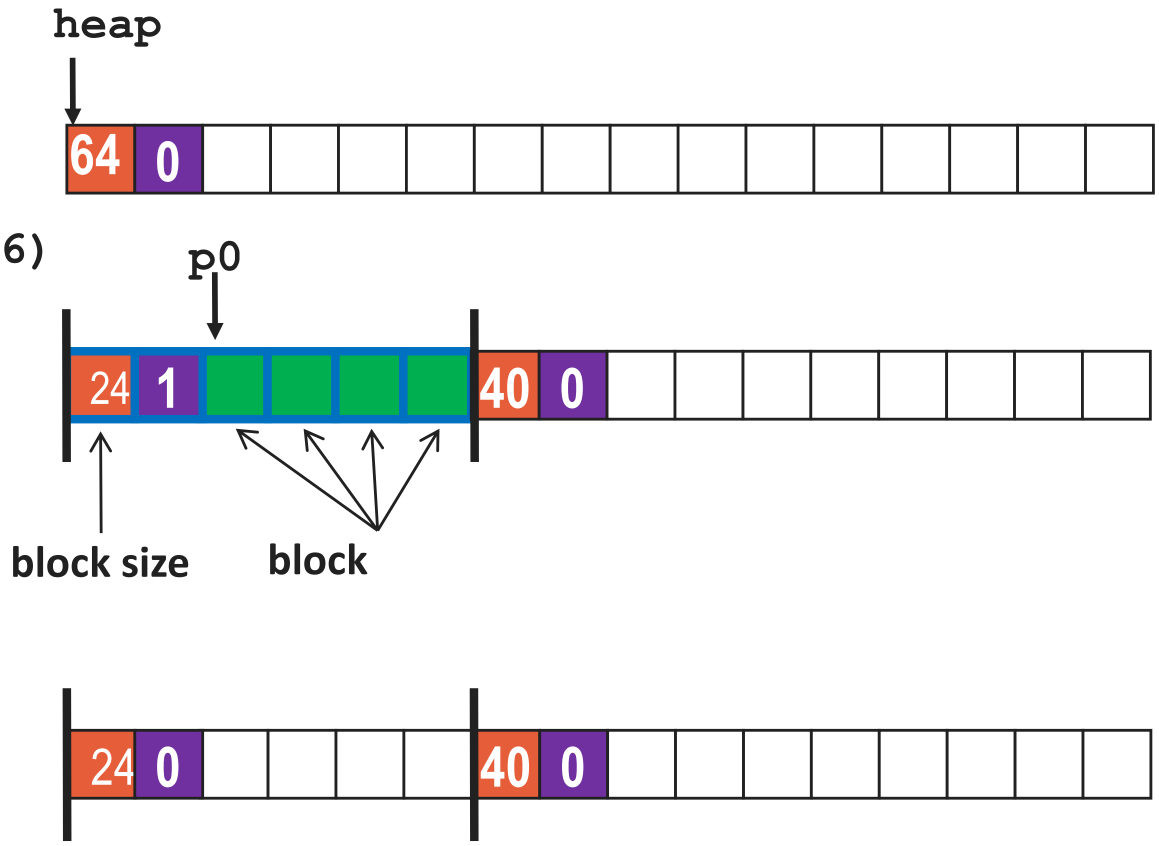
- 1st request splits 1 block into 2 blocks (1 used, 1 free)
  - User gets a pointer to the **block**
  - User does not know about the **header**
  - Notice size is size of **block** + **header**



Q2: Is this block taken?

Simple solution:

- Keep **allocation status** in ***header*** (0=free, 1=allocated)
- Requires *another* extra word for every block
- User *still* does not know about the header



`heap`

64 0

`p0 = malloc(16)`

`p0`

24 1 | 40 0

block size

block

`free(p0)`

24 0 | 40 0

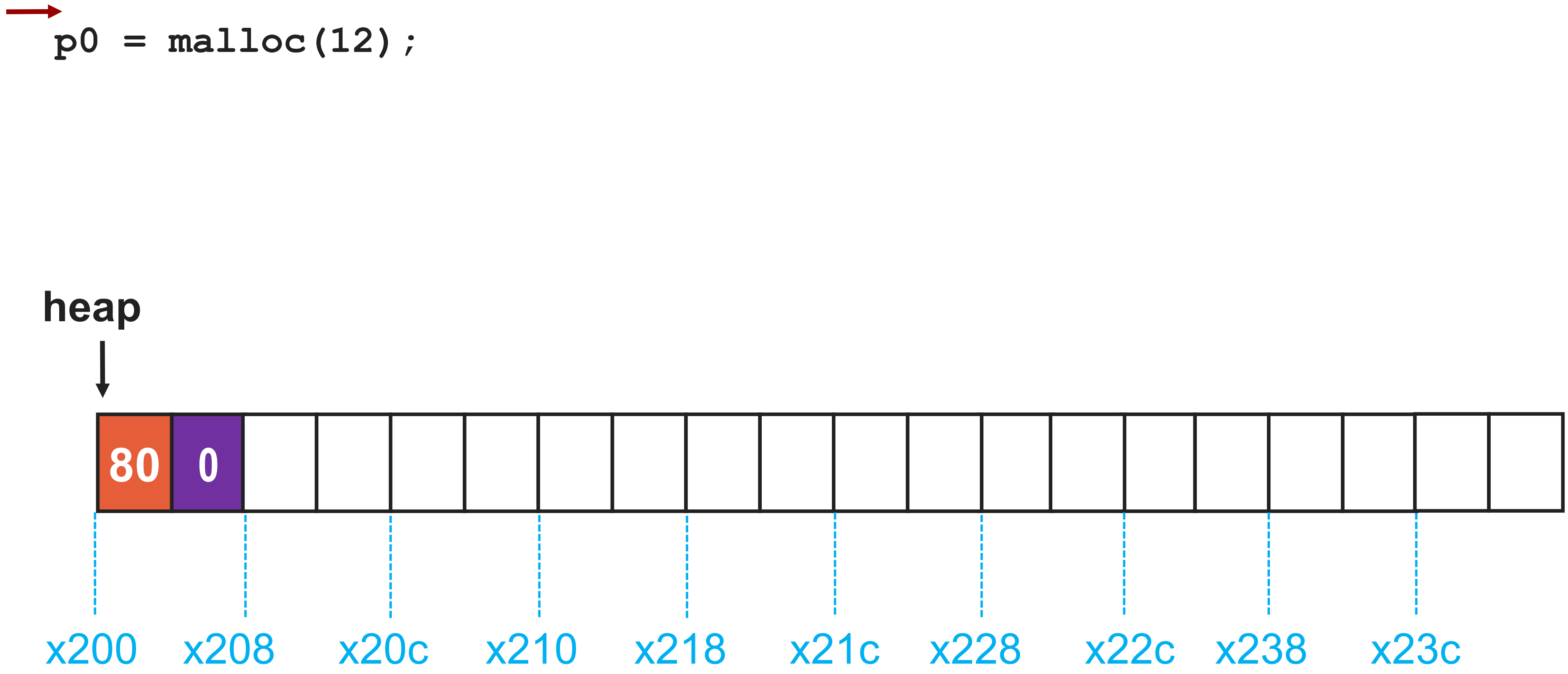
☐ Free word

🟦 Allocated word

What about 8-byte alignment?

If block pointer must be 8-byte aligned, and the header is 8 bytes, then the header should also be 8-byte aligned.

```
p0 = malloc(12);
```



Free blocks: white

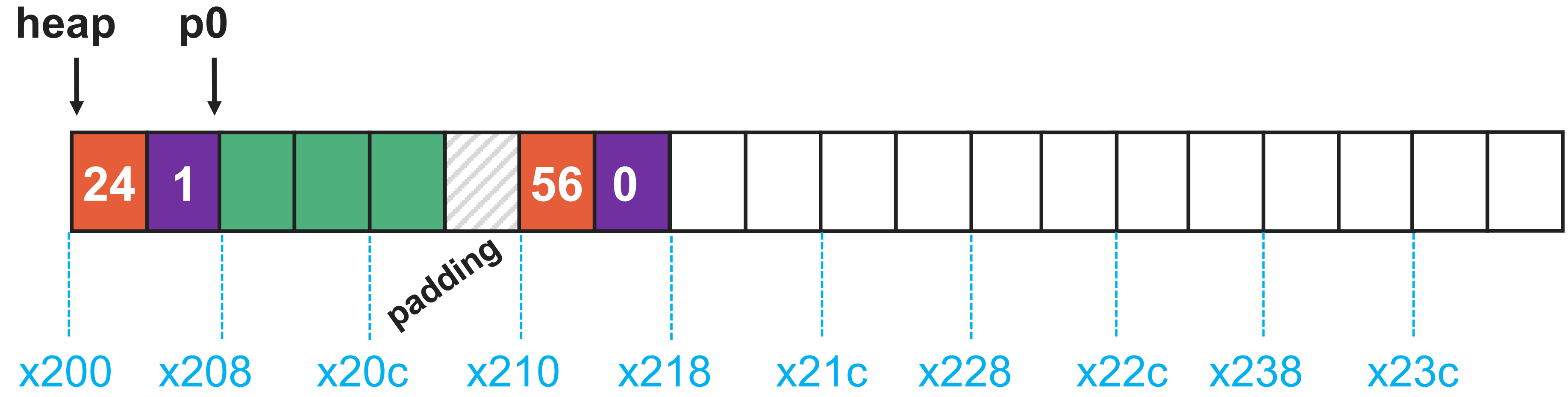
Headers: size in bytes, allocated bit

Each box is 4 bytes.

What about 8-byte alignment?

If block pointer must be 8-byte aligned, and the header is 8 bytes, then the header should also be 8-byte aligned.

```
p0 = malloc(12);
p1 = malloc(24);
```



Free blocks: white

Headers: size in bytes, allocated bit

Padding for alignment: shaded grey

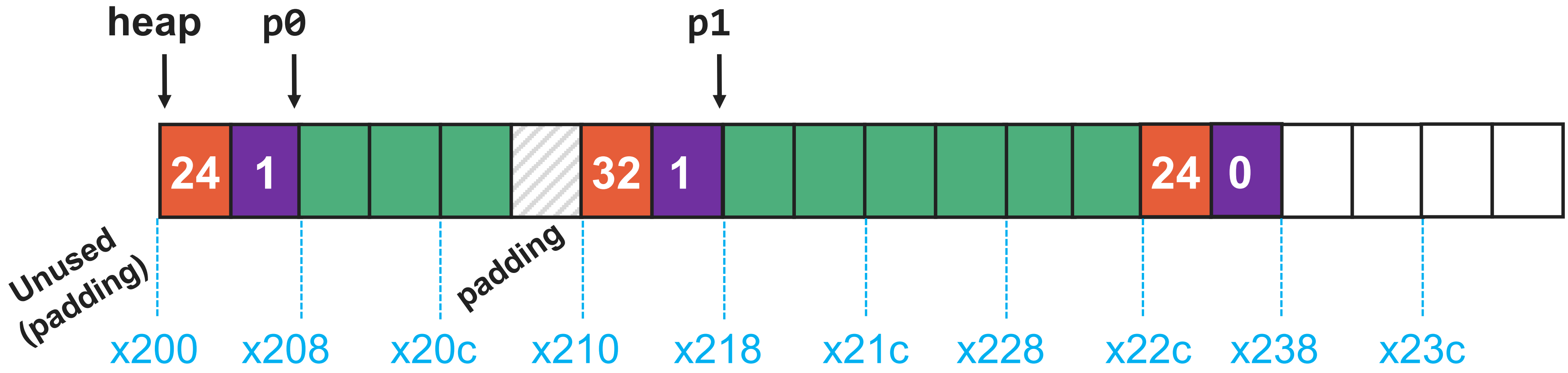
Block (what the user knows about)

Each box is 4 bytes.

What about 8-byte alignment?

If block pointer must be 8-byte aligned, and the header is 8 bytes, then the header should also be 8-byte aligned.

```
p0 = malloc(12);
p1 = malloc(24);
p3 = malloc(8);
```



Free blocks: white

Headers: size in bytes, allocated bit

Padding for alignment: shaded grey

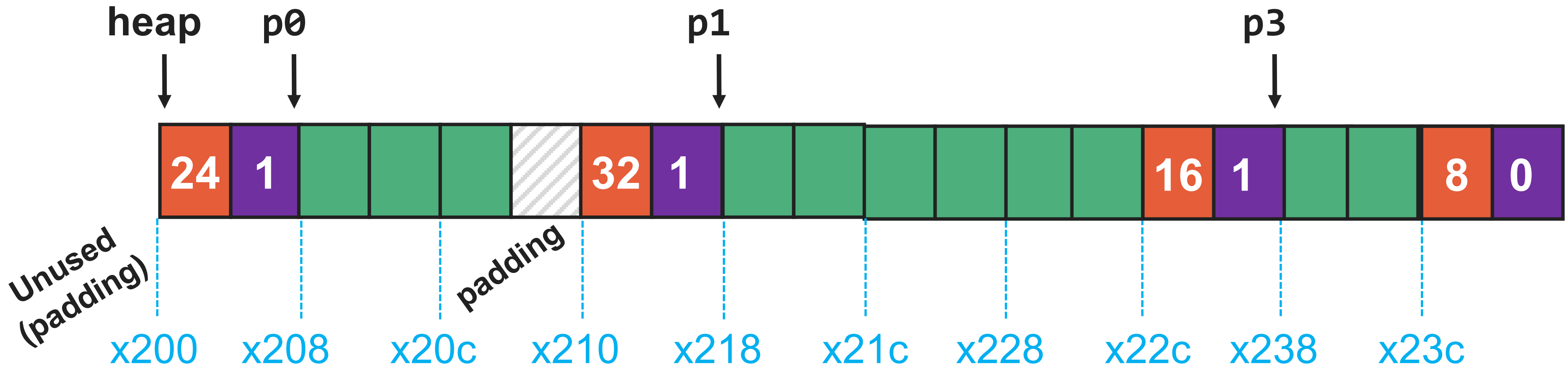
Block (what the user knows about)

Each box is 4 bytes.

What about 8-byte alignment?

If block pointer must be 8-byte aligned, and the header is 8 bytes, then the header should also be 8-byte aligned.

```
p0 = malloc(12);
p1 = malloc(24);
p3 = malloc(8);
free(p1);
```



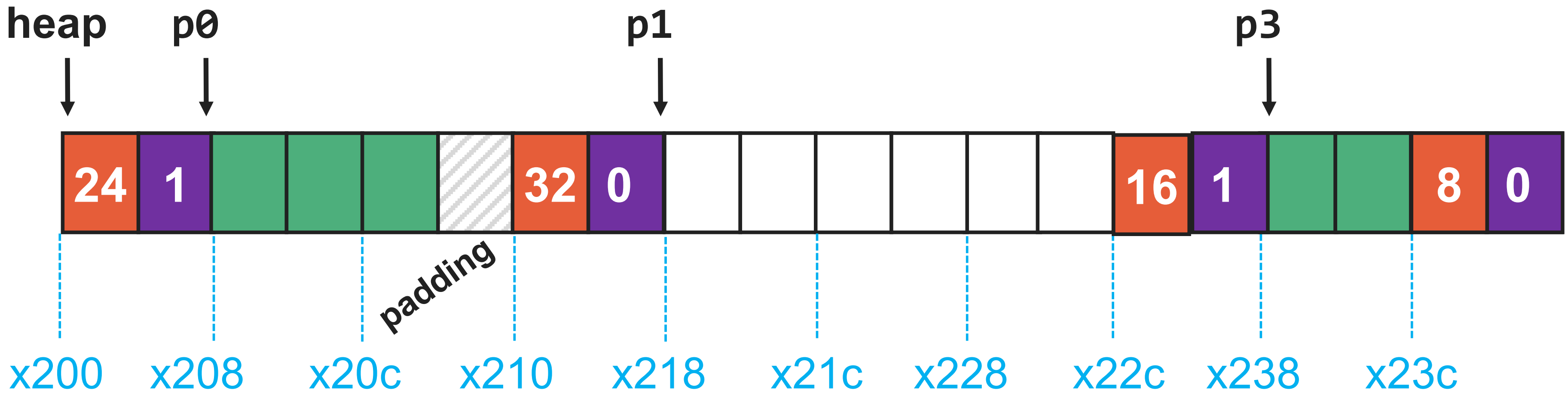
Free blocks: white
Headers: size in bytes, allocated bit
Padding for alignment: shaded grey
Block (what the user knows about)

Each box is 4 bytes.

What about 8-byte alignment?

If block pointer must be 8-byte aligned, and the header is 8 bytes, then the header should also be 8-byte aligned.

```
p0 = malloc(12);
p1 = malloc(24);
p3 = malloc(8);
free(p1);
```



Free blocks: white

Headers: size in bytes, allocated bit

Padding for alignment: shaded grey

Block (what the user knows about)

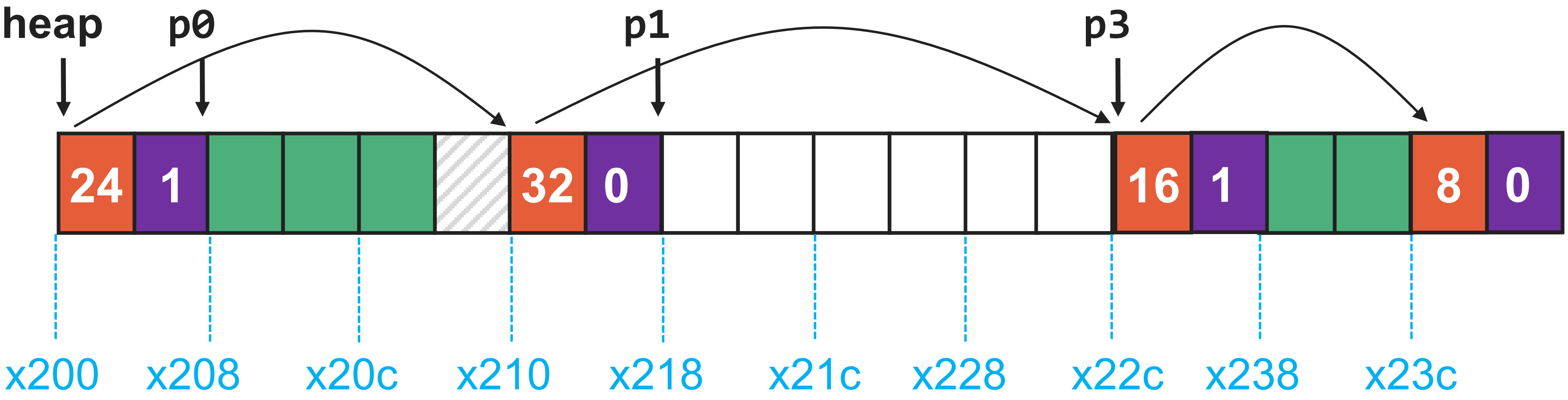
Each box is 4 bytes.

What about 8-byte alignment?

If block pointer must be 8-byte aligned, and the header is 8 bytes, then the header should also be 8-byte aligned.

```
p0 = malloc(12);
p1 = malloc(24);
p3 = malloc(8);
free(p1);
```

There are no actual pointers. You traverse through the heap by starting at the heap ptr and adding size to current block.



Free blocks: white

Headers: size in bytes, allocated bit

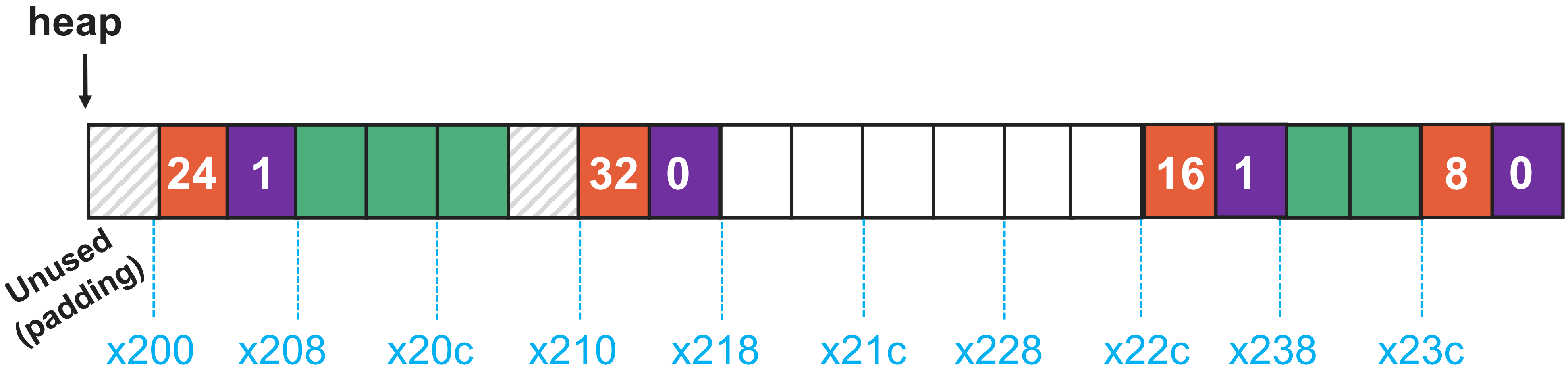
Padding for alignment: shaded grey

Block (what the user knows about)

Each box is 4 bytes.

Also, the heap might not be aligned

Might need to align the heap before you do anything.
(Also need to keep track of where the heap ends so you don't run off it.)

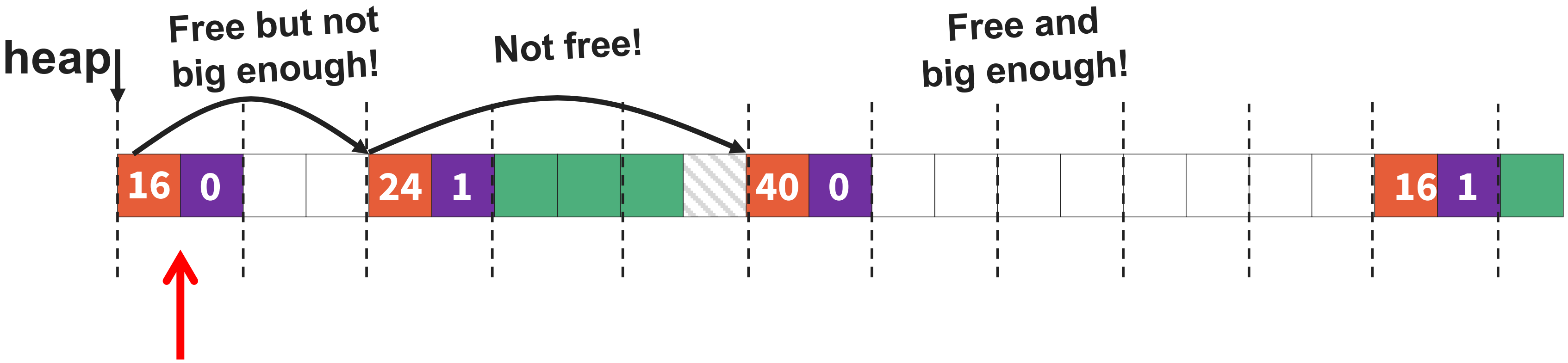


Free blocks: white
Headers: size in bytes, allocated bit
Padding for alignment: shaded grey
Block (what the user knows about)

Each box is 4 bytes.

Q3: Allocating a New Block

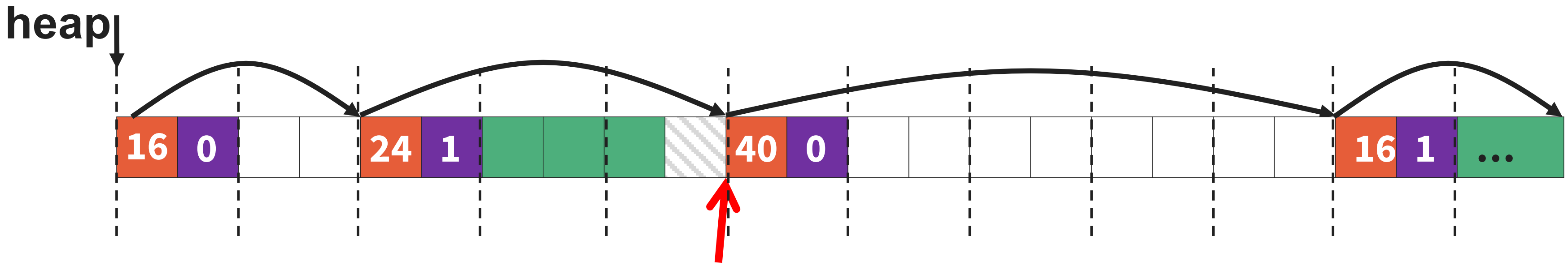
Suppose we need to allocate 12 bytes



This is our free block of choice

Q3: Allocating a New Block

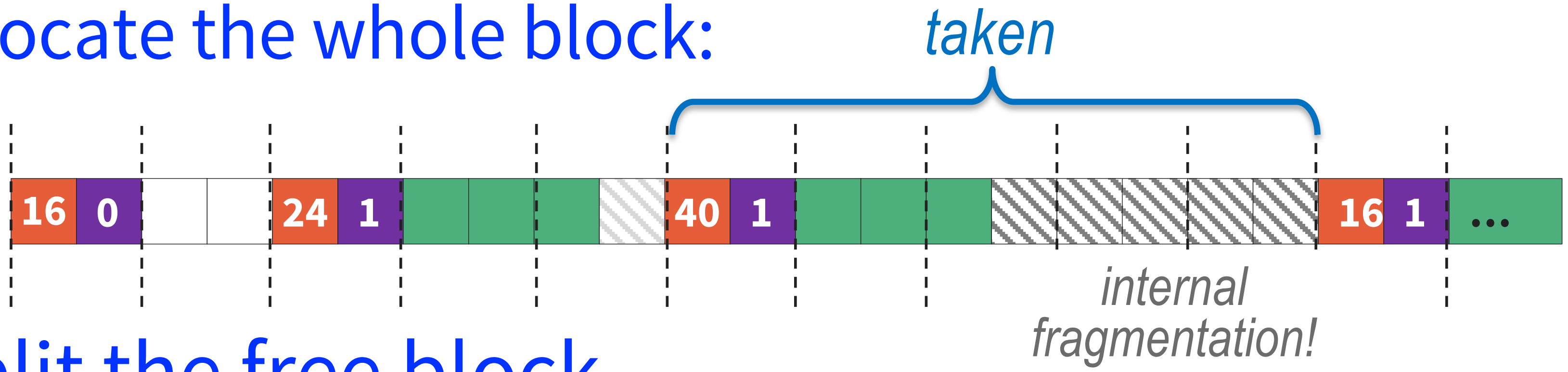
Suppose we need to allocate 12 bytes



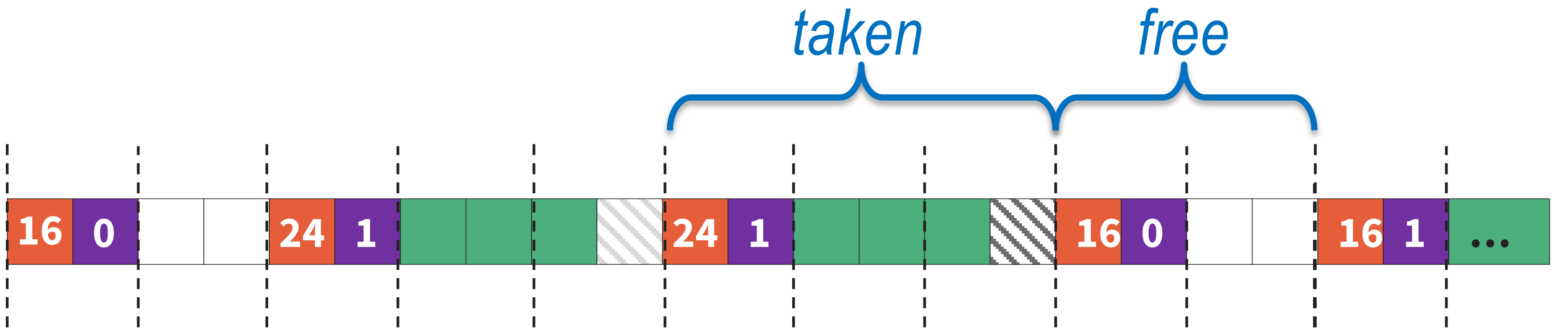
This is our free block of choice

Two options:

1. Allocate the whole block:



2. Split the free block



Q4: Finding a Free Block

First fit

- Search from beginning, choose ***first*** free block that fits:
- Linear time in total number of blocks (allocated and free)
- Can cause “splinters” (of small free blocks) at beginning of list

Next fit

- Like first fit, but search list starting where previous search finished
- Often faster than first fit: avoids re-scanning unhelpful blocks
- Some research suggests that fragmentation is worse

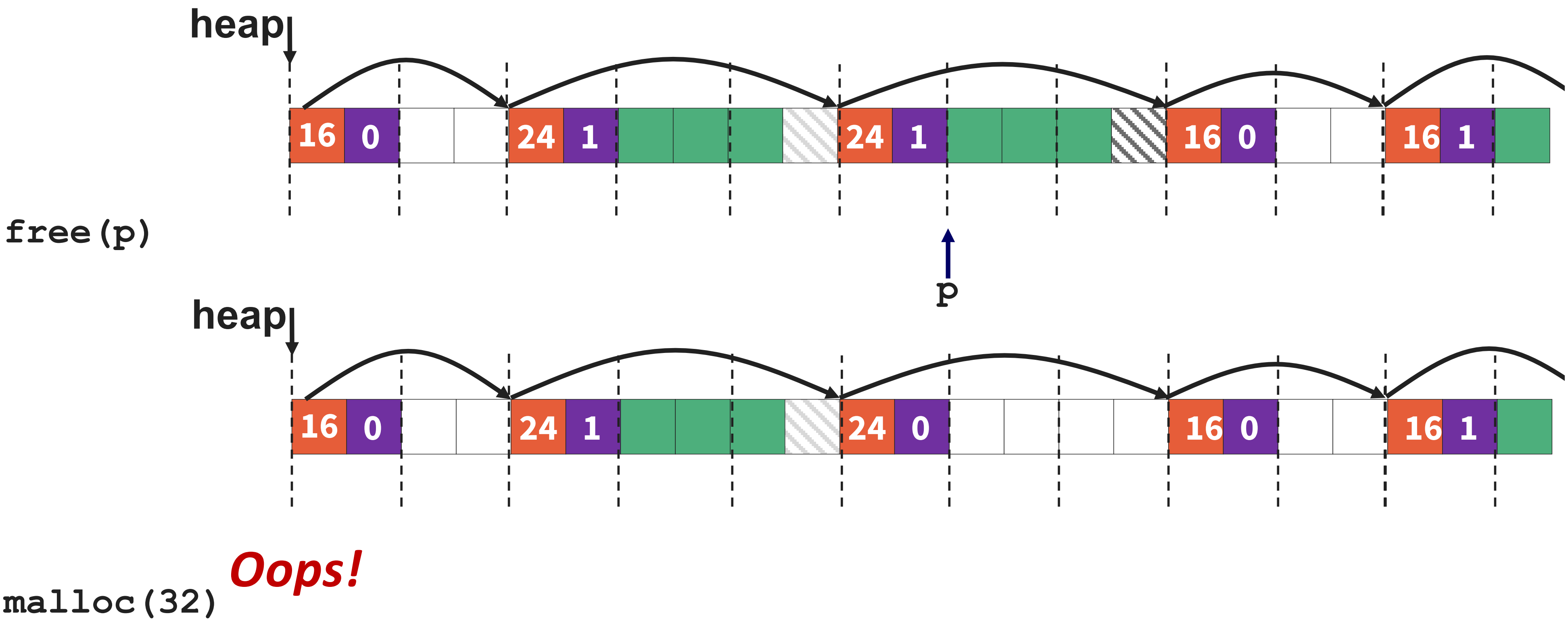
Best fit

- Search list, choose the ***best*** free block: fits, with fewest bytes left over
- Keeps fragments small—usually helps fragmentation
- Typically runs slower than first fit

Q5: Freeing a Block

Simplest implementation: clear the “allocated” flag

- But can lead to “false fragmentation”



Oops!

There is enough free space, but the allocator won’t be able to find it

Today

- Basic concepts
- Basic Implementation
  - Implicit Free Lists: *because pointers are calculated via the size field rather than with actual pointers.*
  - Explicit Free Lists
- Implementation Optimizations

Note: it is your choice whether you do an explicit or implicit list for this project!

This will blow your mind

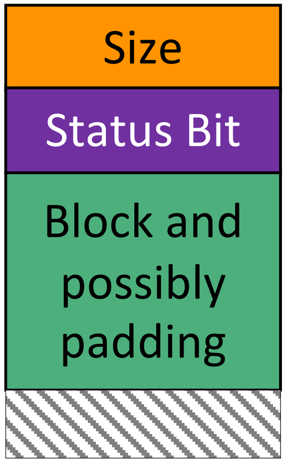
We don’t need to track all the blocks.
We only need to track the free ones.

Dynamic Memory Allocation Library *only frees allocated blocks when user says so:*

- User provides the pointer to be freed
- Cannot ever move or use the allocated blocks in the meantime

Two Types of Lists

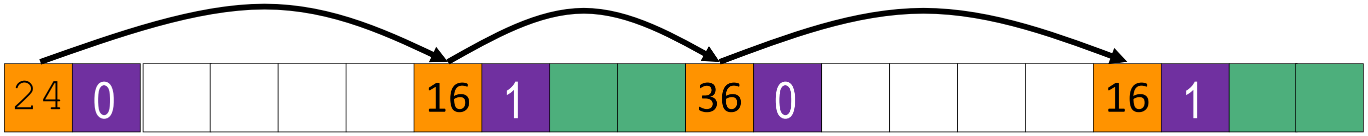
Implicit free list links **all** blocks using length



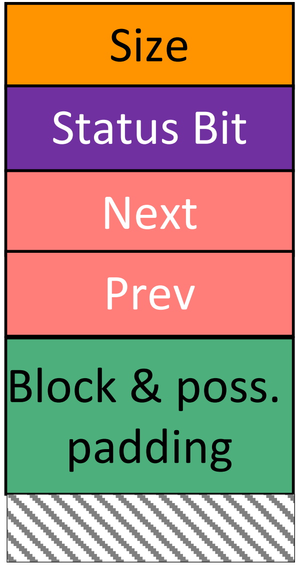
Size

Status Bit

Block and possibly padding



Explicit free list links **free** blocks using ptrs



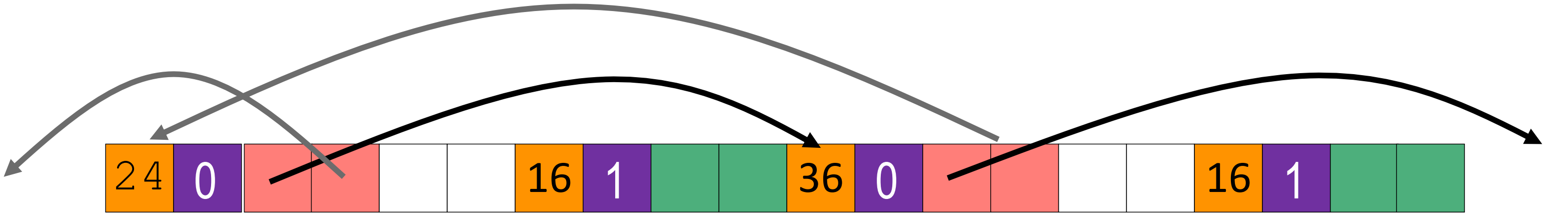
Size

Status Bit

Next

Prev

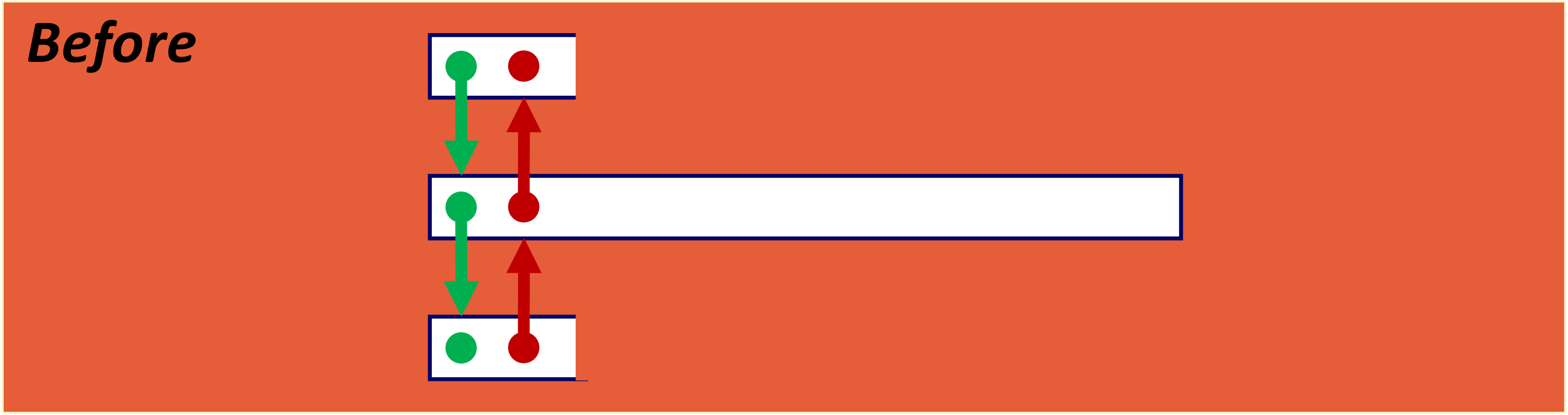
Block & poss. padding

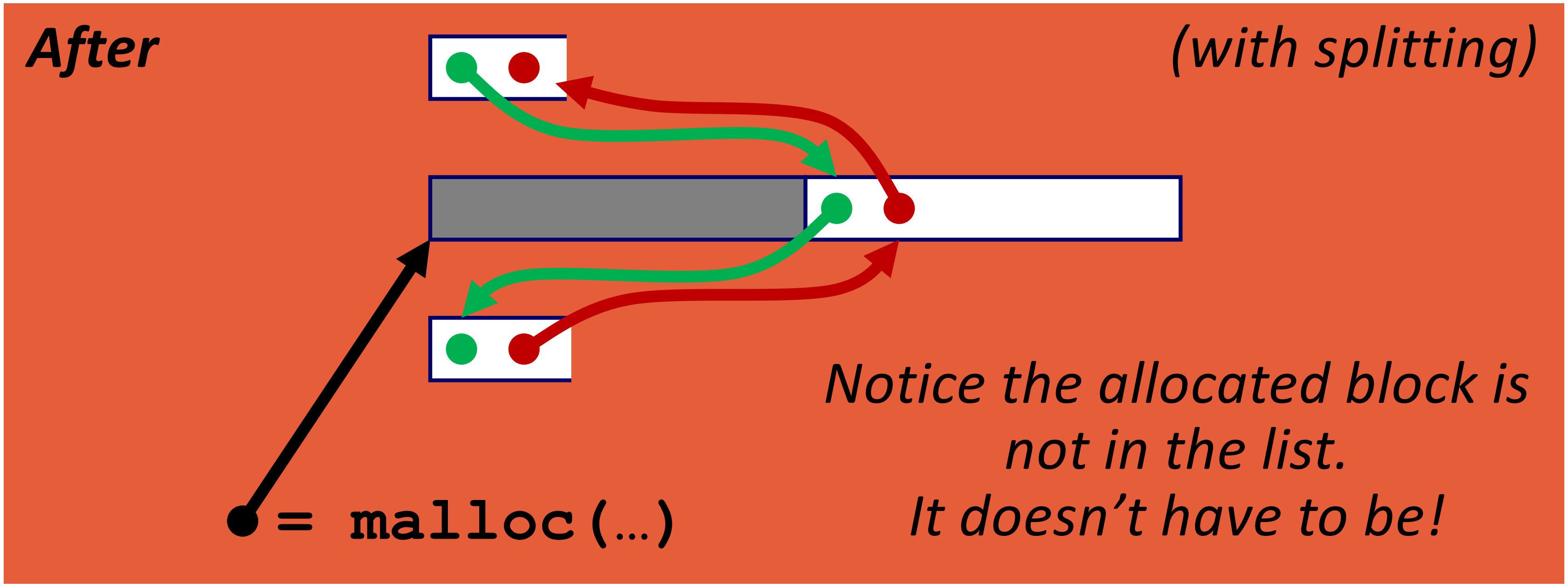


- “next” free block could be anywhere
- next pointer goes away when block is allocated
- (in C: two ways of casting the same block)

Allocating From Explicit Free Lists

conceptual graphic





Notice the allocated block is not in the list.
It doesn’t have to be!

Explicit List Summary

Comparison to implicit list:

- Allocate: linear in # of ***free*** blocks (instead of ***all*** blocks)
  - ***Much faster*** when most of the memory is full
- More complicated allocate/free (needs to splice blocks in/out of list)
- extra space for the links (2 extra words needed for each free block)

Most common use of linked lists is in conjunction with segregated free lists

- Keep multiple linked lists of different size classes, or possibly for different types of objects

Beyond Correctness

Utilization:

- make best use of the heap as possible

Performance:

- respond as quickly as possible

Beyond Implicit and Explicit

Method 3: *Segregated free list*

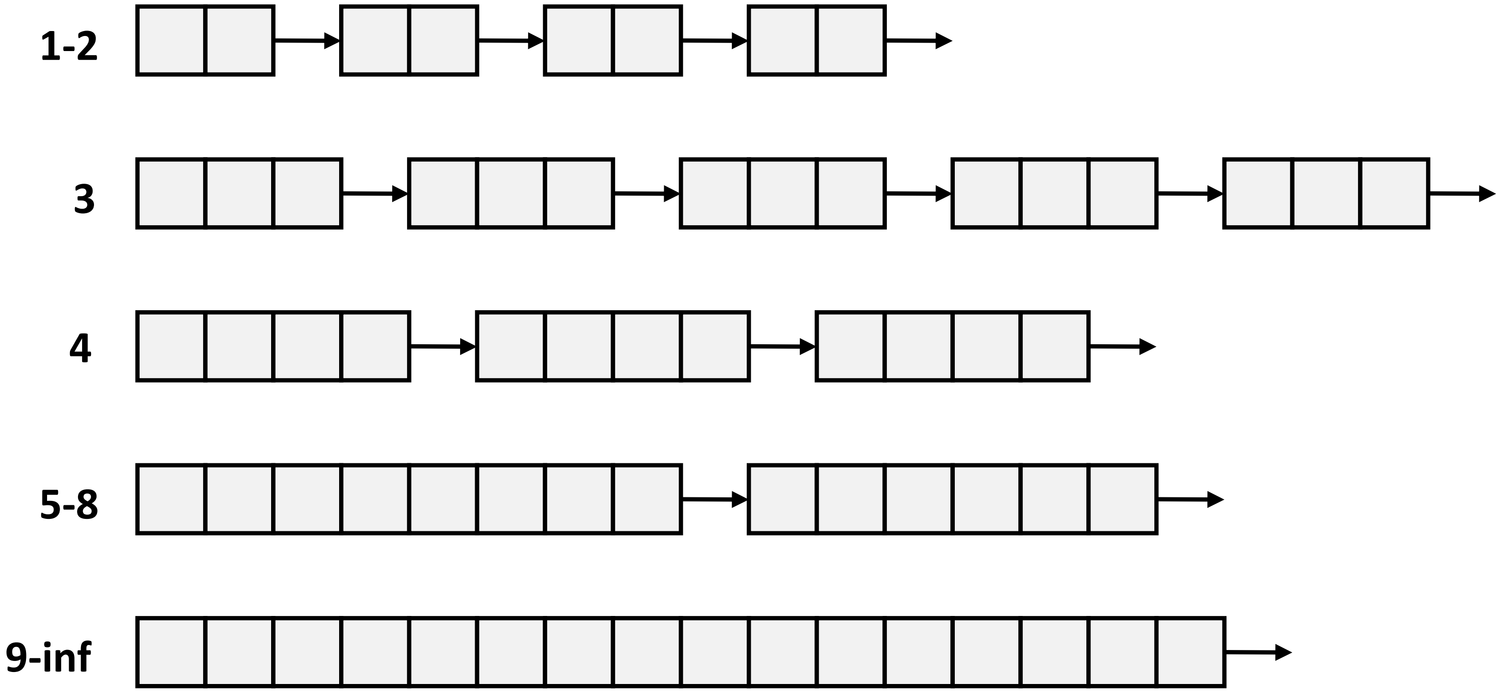
- different free lists for different size classes

Method 4: *Blocks sorted by size*

- Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Segregated List (Seglist) Allocators

- Each *size class* of blocks has its own free list



- Often have separate classes for each small size
- For larger sizes: One class for each two-power size

Seglist Allocator

- **Given an array of free lists, each one for some size class**

- **To allocate a block of size *n*:**
  - Search appropriate free list for block of size $m > n$
  - If found: split block, optionally place fragment on appropriate list
  - If no block is found, try next larger class
  - Repeat until block is found

- **If no block found:**
  - Real World:
    - Request additional heap memory from OS (using **`sbrk()`**)
    - Allocate block of *n* bytes from new memory
    - Place remainder as a single free block in largest size class
  - CS 3410:
    - Return NULL

Seglist Allocator (cont.)

- **To free a block:**
  - Coalesce and place on appropriate list (optional)

- **Advantages of seglist allocators**
  - Higher throughput
    - log time for power-of-two size classes
  - Better memory utilization
    - First-fit search of segregated free list approximates a best-fit search of entire heap
    - Extreme case: giving each block its own size class is equivalent to best-fit

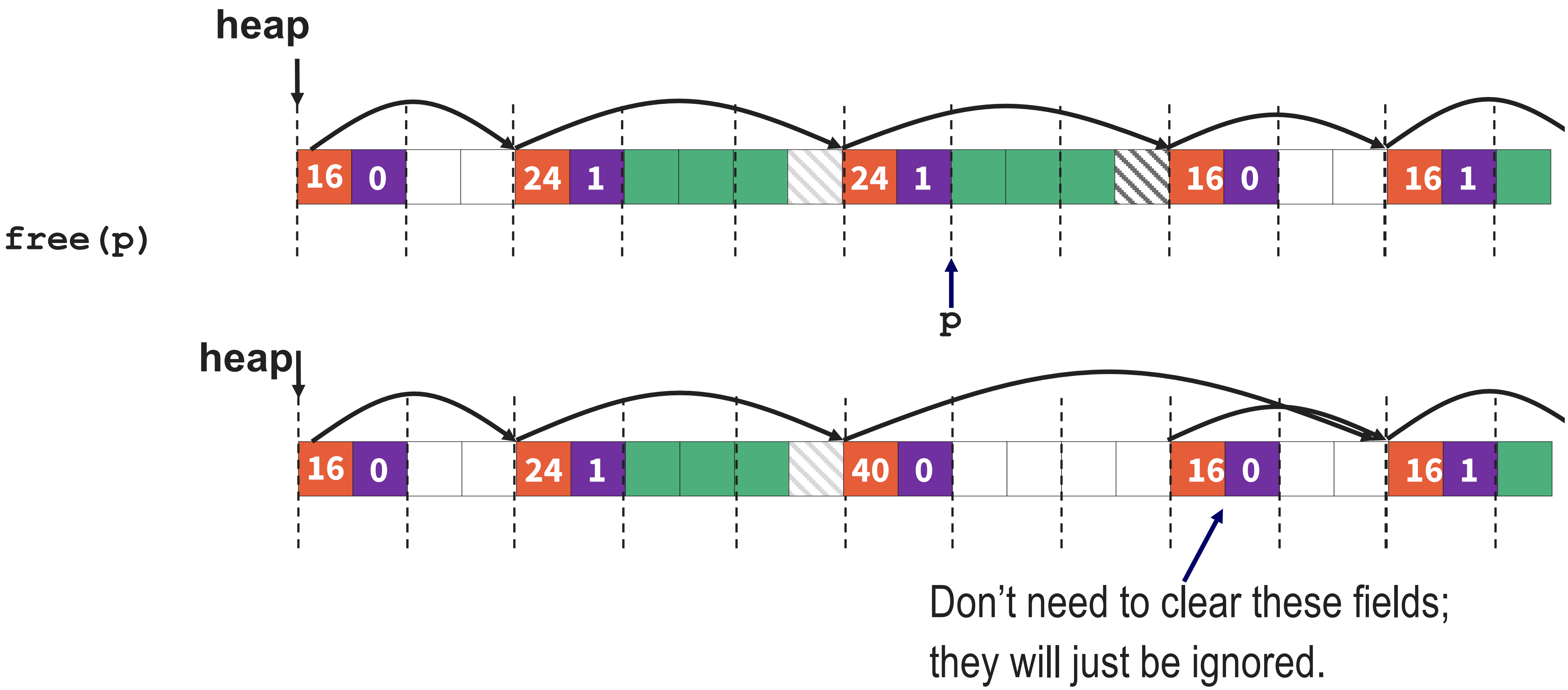
Today

- Basic concepts
- Basic Implementation
- Implementation Optimizations
    - Coalescing
    - Header Optimization

Do not try these optimizations until you have the basic implementation working.

Coalescing

Joining blocks, if they are free

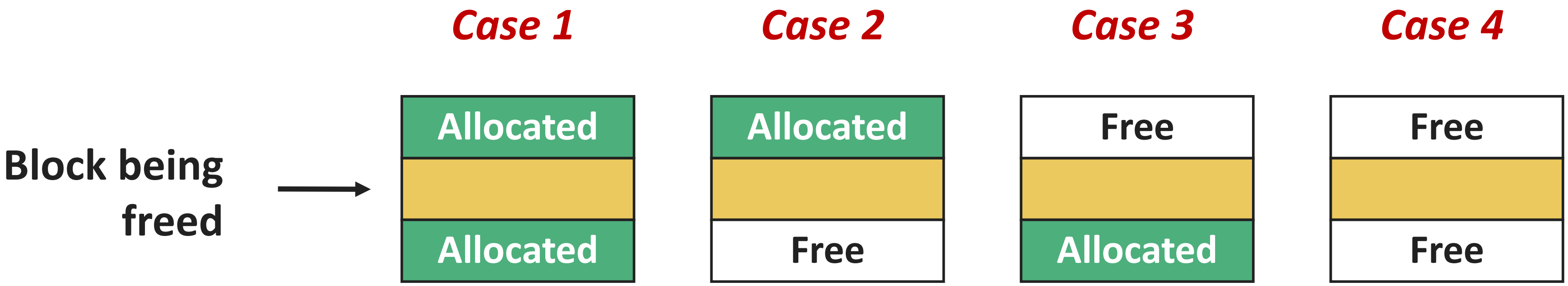


Don't need to clear these fields; they will just be ignored.

Easy to find the blocks *after* the block being freed.

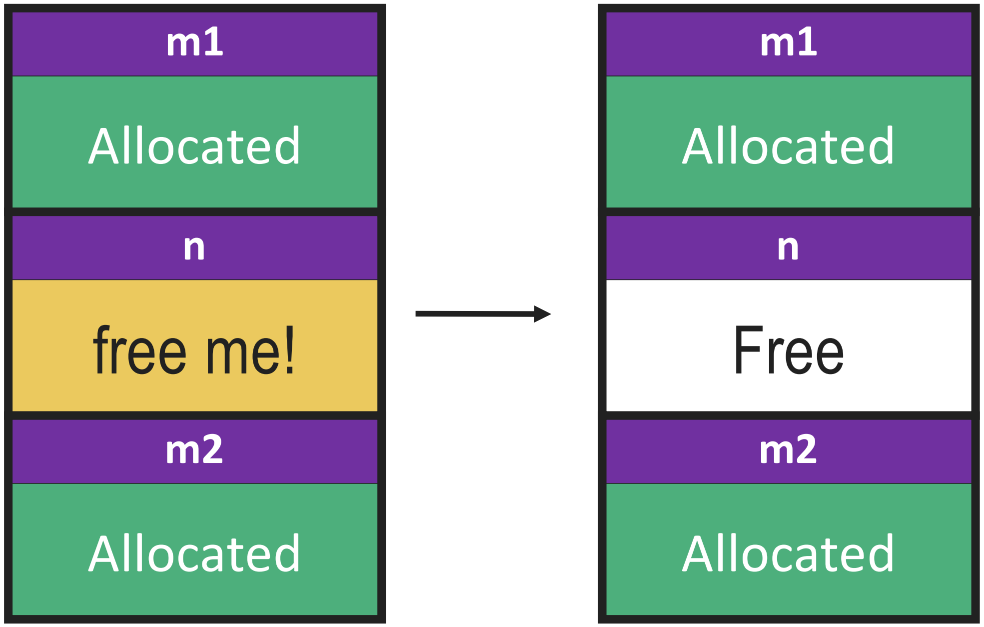
Harder to find the blocks *before* the block being freed.

Coalescing Cases



	Case 1	*Case 2*	*Case 3*	*Case 4*
	Allocated	Allocated	Free	Free
Block being freed →				
	Allocated	Free	Allocated	Free

Coalescing: Case 1



m1

Allocated

n

free me!

m2

Allocated

m1

Allocated

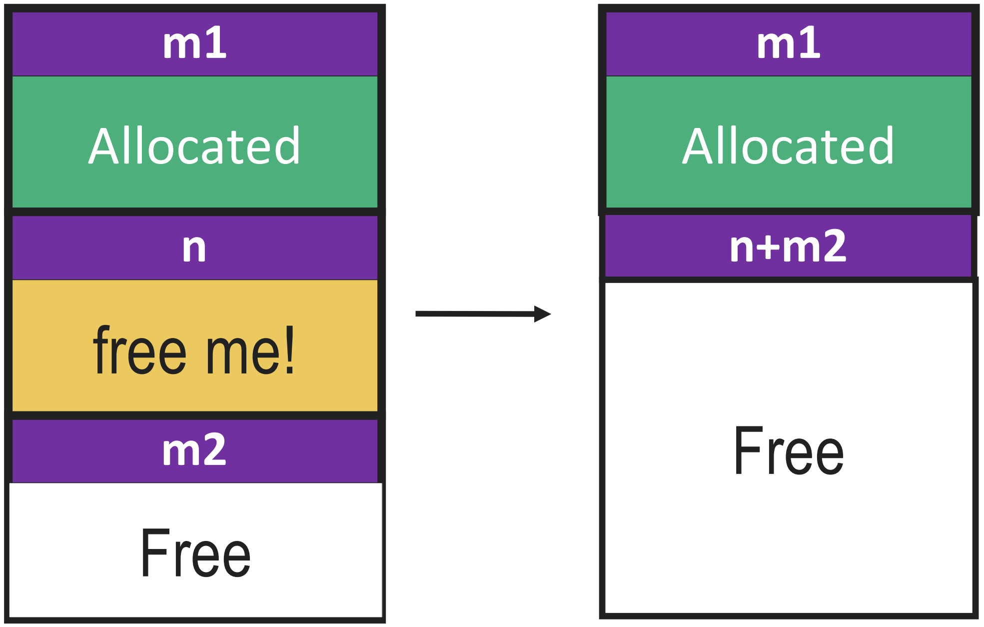
n

Free

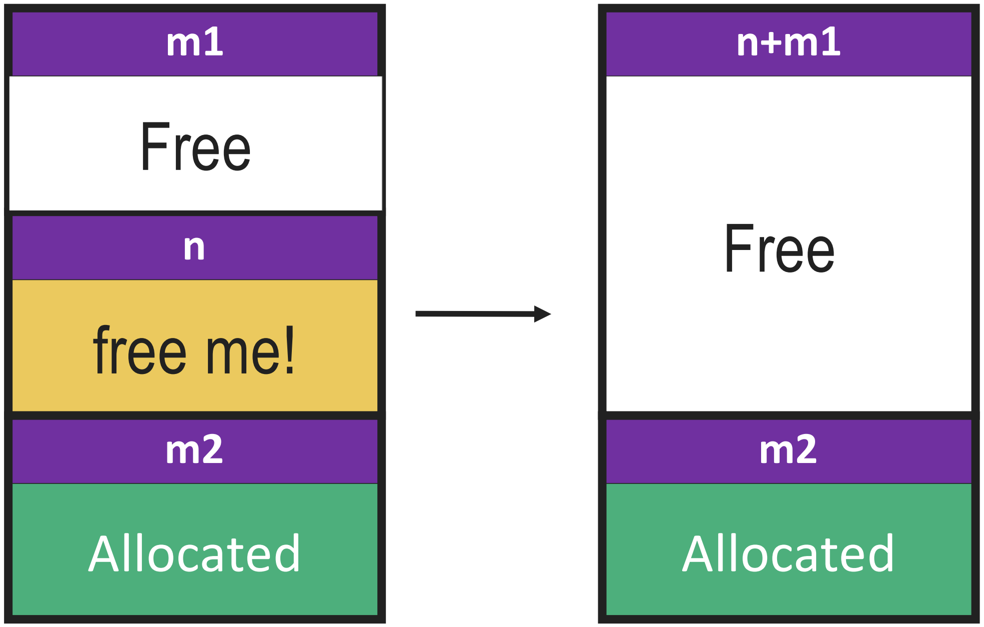
m2

Allocated

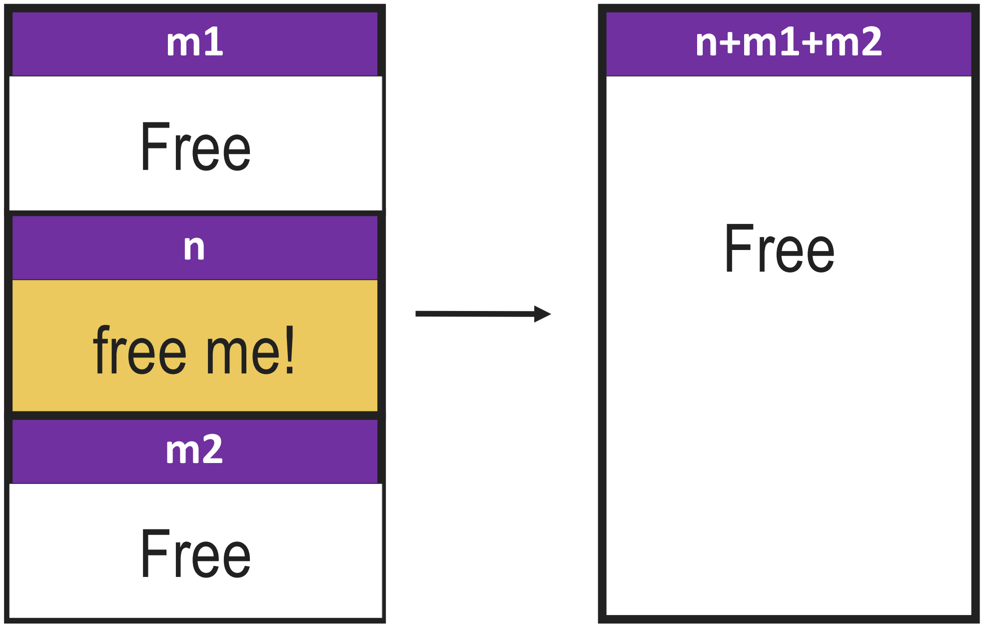
Coalescing: Case 2



Coalescing: Case 3



Coalescing: Case 4



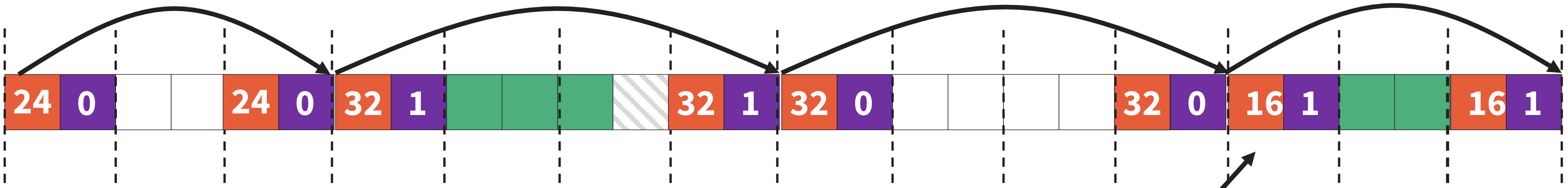
Wait a Minute…

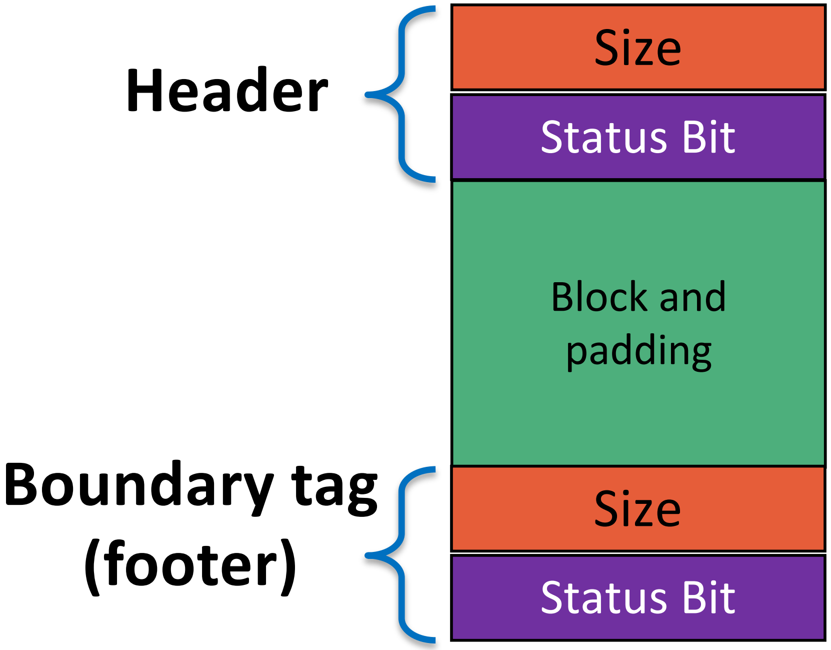
- How to we coalesce with the block in front?
- How do we know what block is in front of another block?

Implicit List: Bidirectional Coalescing

Boundary tags [Knuth73]

- Replicate size/allocated word at end of free blocks
- Allows us to traverse the "list" backwards, but requires extra space
- Important and general technique!





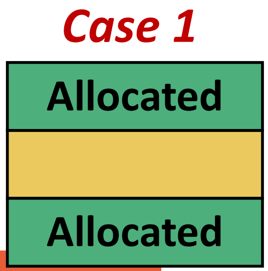
Now if you want to free this block, you know how to check both neighbors' status bits

Freeing With Explicit Free Lists

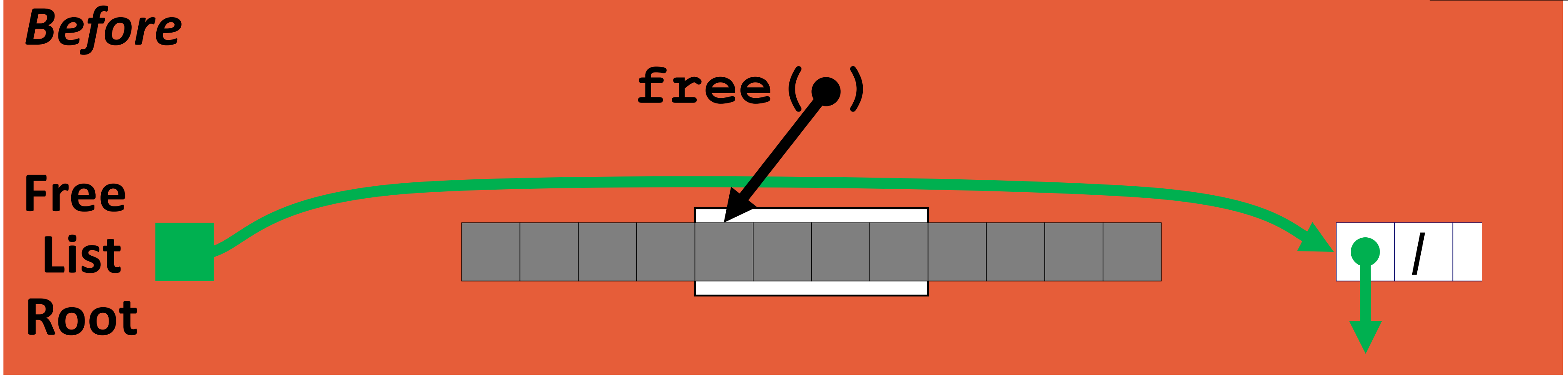
- ***Insertion policy*: Where do you put a newly freed block?**
  - **LIFO (last-in-first-out) policy**
    - Insert freed block at the beginning of the free list
    - ***Pro:*** simple and constant time
    - ***Con:*** studies suggest fragmentation worse than addr-ordered
  - **Address-ordered policy**
    - Insert freed blocks so free list blocks always in address order:

      addr(prev) < addr(curr) < addr(next)
    - ***Con:*** requires search
    - ***Pro:*** studies suggest fragmentation is lower than LIFO

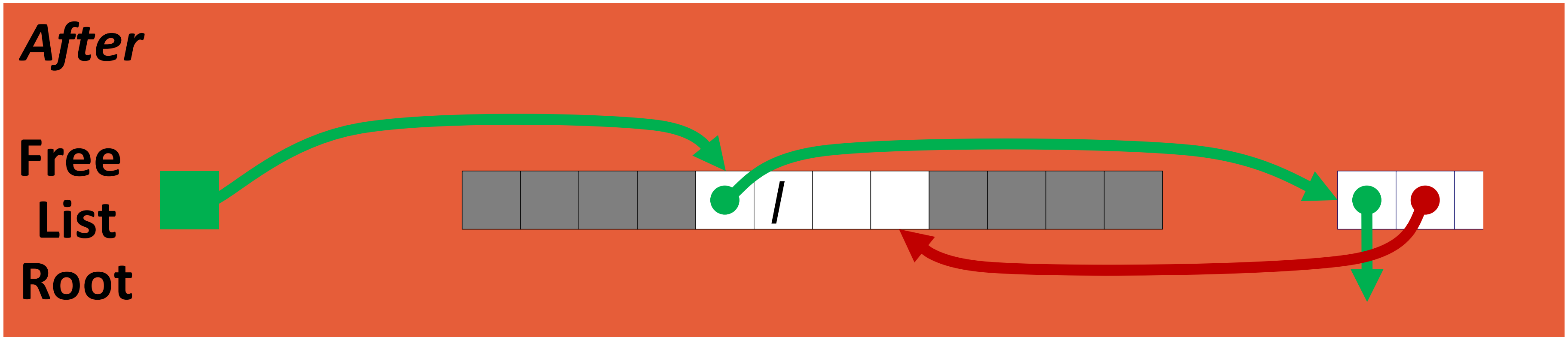
Freeing With a LIFO Policy (Case 1)



conceptual graphic



- Insert the freed block at the root of the list



Freeing With a LIFO Policy (Case 2)

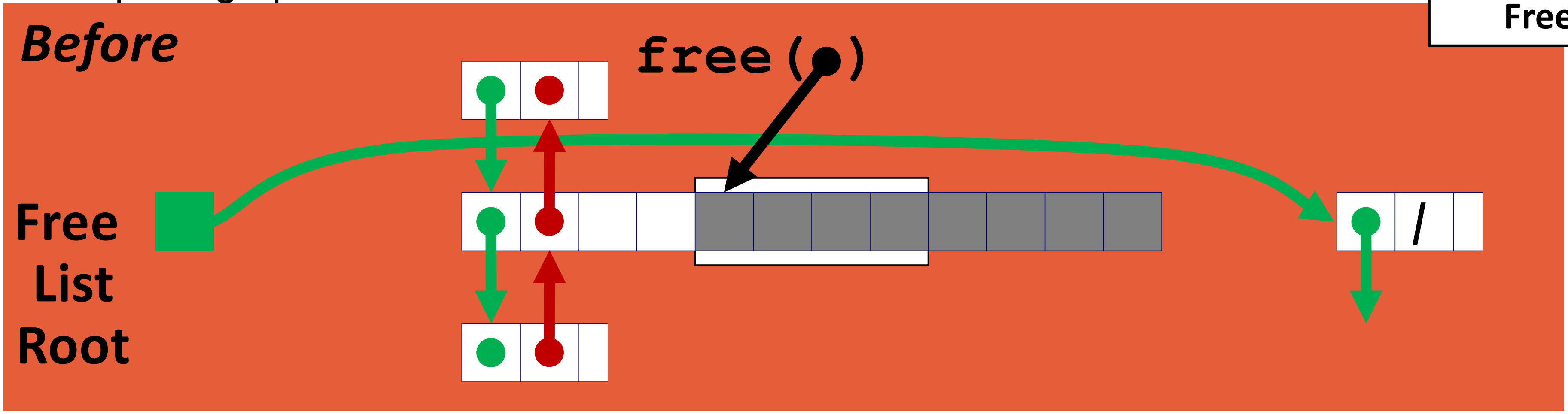
Case 2

🟩 Allocated

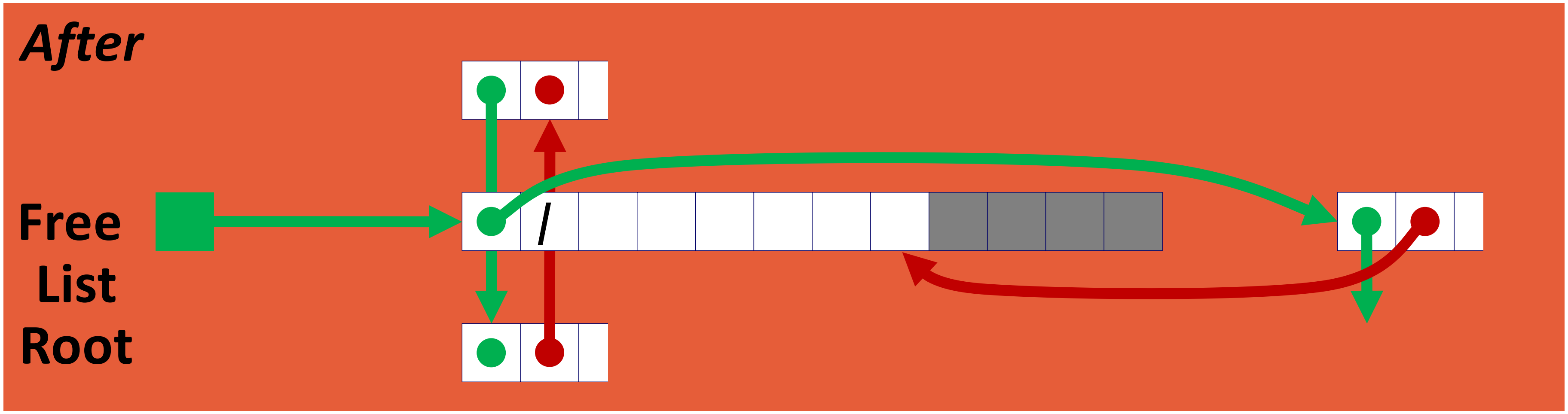
🟨

☐ Free

conceptual graphic



- **Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list**



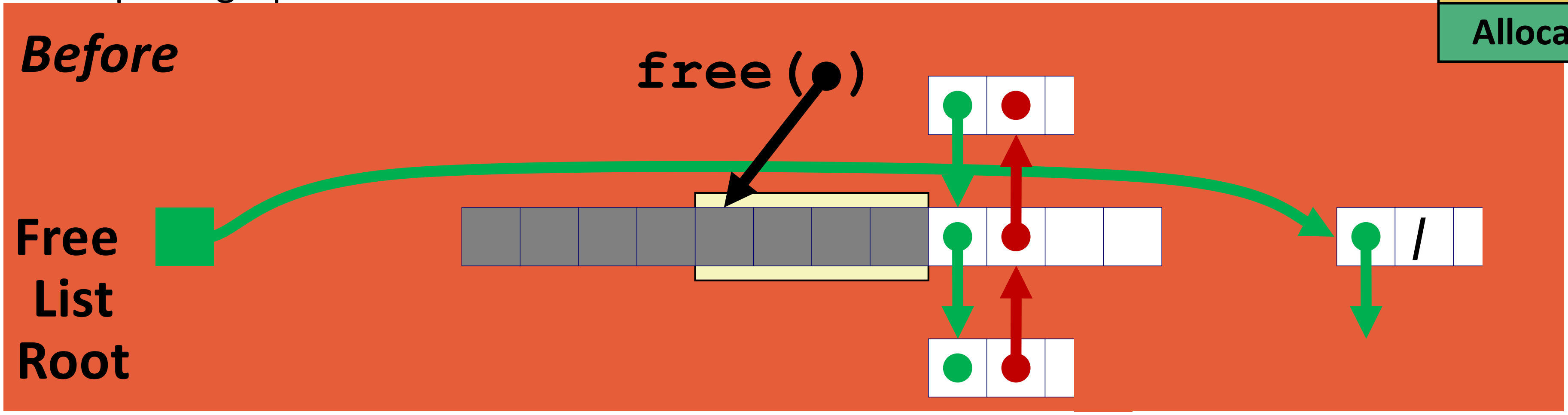
Freeing With a LIFO Policy (Case 3)

conceptual graphic

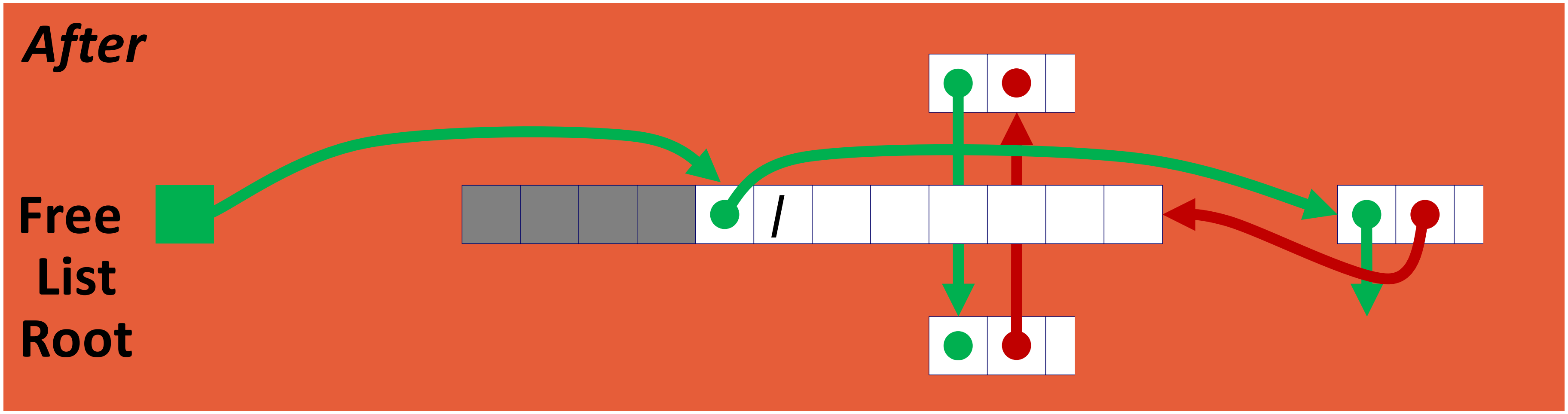
Case 3

Free

Allocated

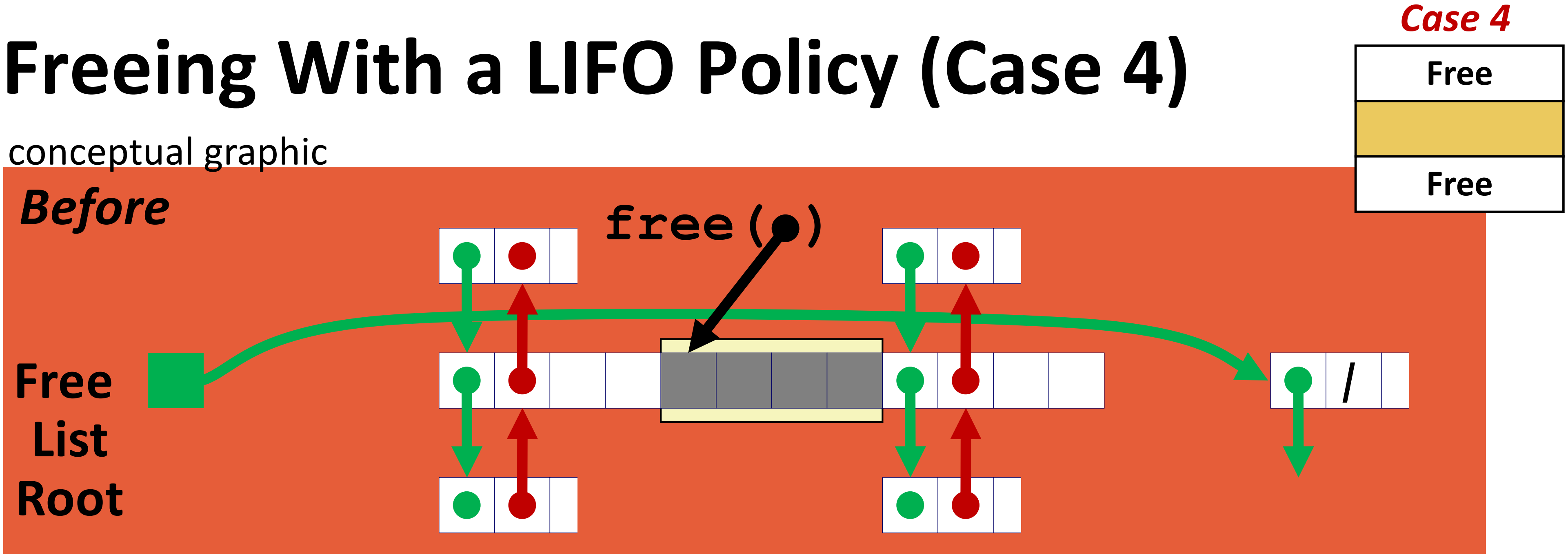


- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list

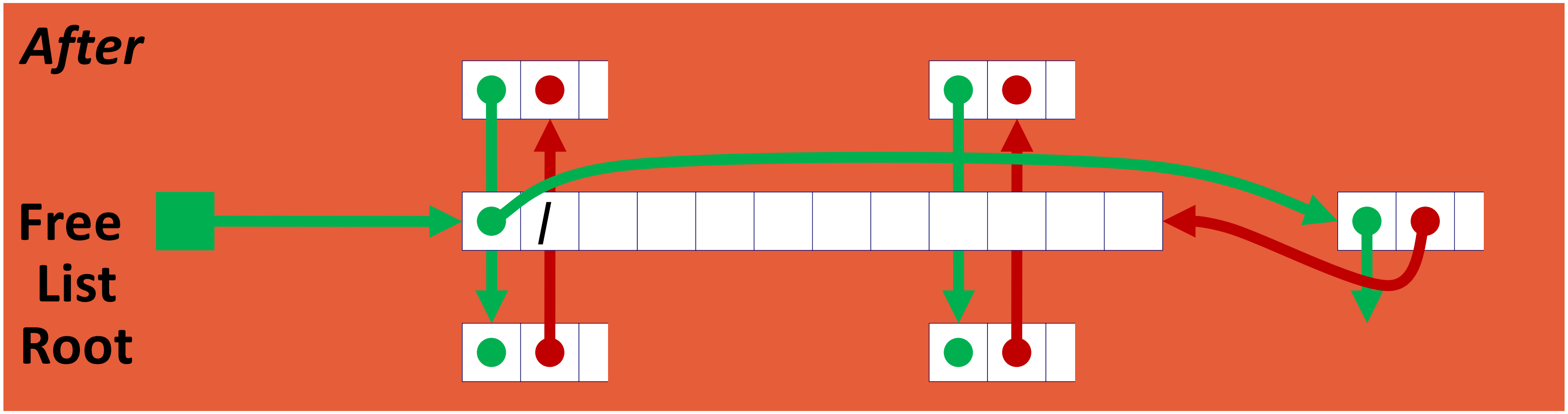


Freeing With a LIFO Policy (Case 4)

conceptual graphic



- **Splice out predecessor and successor blocks, coalesce all 3 memory blocks and insert the new block at the root of the list**



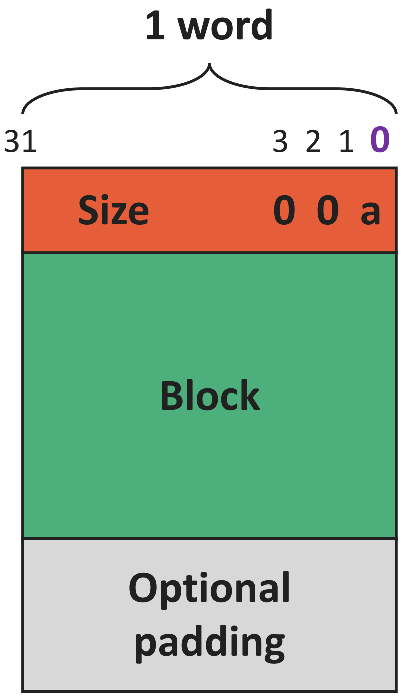
Today

- Basic concepts
- Basic Implementation
- Implementation Optimizations
  - Coalescing
  - Header Optimization

Header Optimization!

- Standard trick to keep overhead low:
  - If blocks are aligned, size is never odd, LSB always 0
  - Instead of storing 0, use LSB as allocated/free flag
  - Merge the **size** & **status** fields into 1 word
  - When reading size word, must mask out this bit

Format of allocated and free blocks



a = 1: Allocated block
a = 0: Free block

Size: block size

block: application data (allocated blocks only)

Summary of Key Allocator Policies

Placement policy:

- First-fit, next-fit, best-fit, etc.
- Tradeoffs: throughput vs. fragmentation

Splitting policy:

- When do we go ahead and split free blocks?
- How much internal fragmentation are we willing to tolerate?

More Info on Allocators

Bryant & O’Hallaron, “Computer Systems: A Programmer's Perspective” Sections 9.9-9.13

- A great book about System Software

D. Knuth, “*The Art of Computer Programming*”, 2nd edition, Addison Wesley, 1973

- The classic reference on dynamic storage allocation

Wilson et al, “*Dynamic Storage Allocation: A Survey and Critical Review*”, Proc. 1995 Int’l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.

- Comprehensive survey
- Available from CS:APP student site (csapp.cs.cmu.edu)