Lab 5 Overview

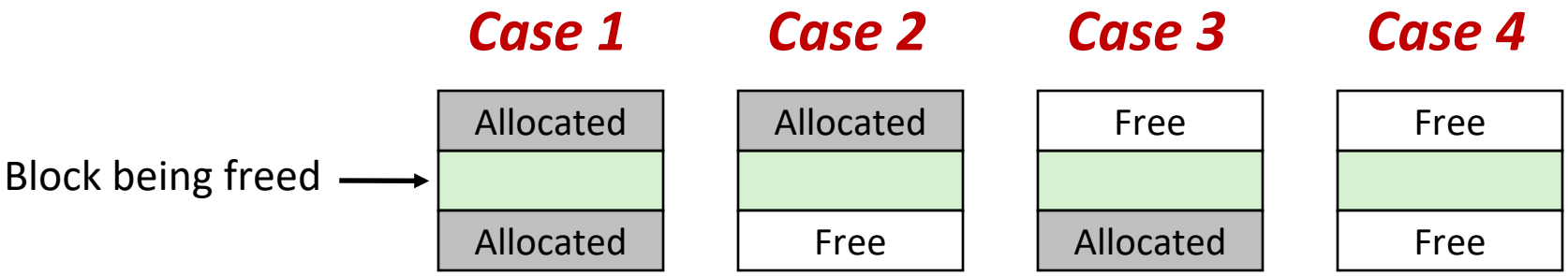
Section 9

Dynamic Memory Allocation

Coalescing

- What?
  - Combining consecutive free blocks
- Why?
  - If we didn't coalesce then the free list would consist of a bunch of small blocks.
  - Upon an allocation request, we might mistakenly think we don't have enough contiguous free space.
- When?
  - When we free a block, we check the *preceding* and *following* blocks to see if they are free.
  - If at least one is free, we coalesce.

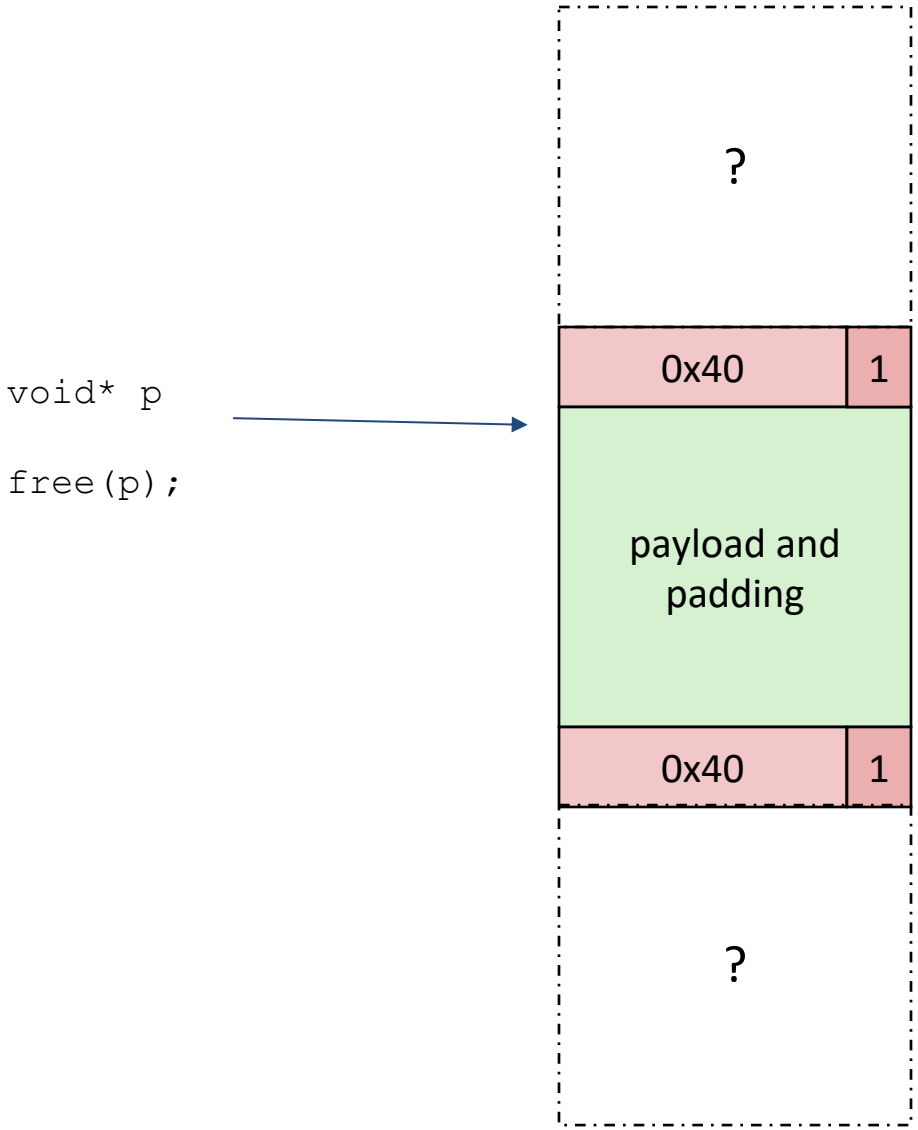
Coalescing in Explicit Free Lists



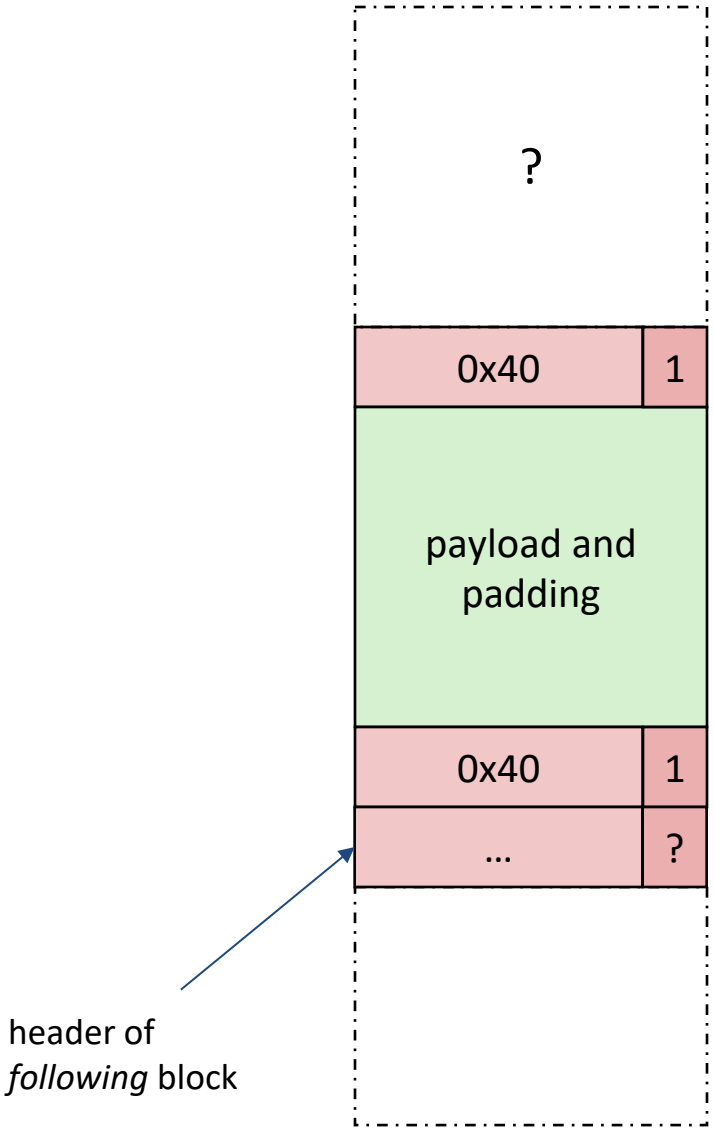
Neighboring free blocks are *already part of the free list*

1) Remove old block from free list
2) Create new, larger coalesced block
3) Add new block to free list (insertion policy)

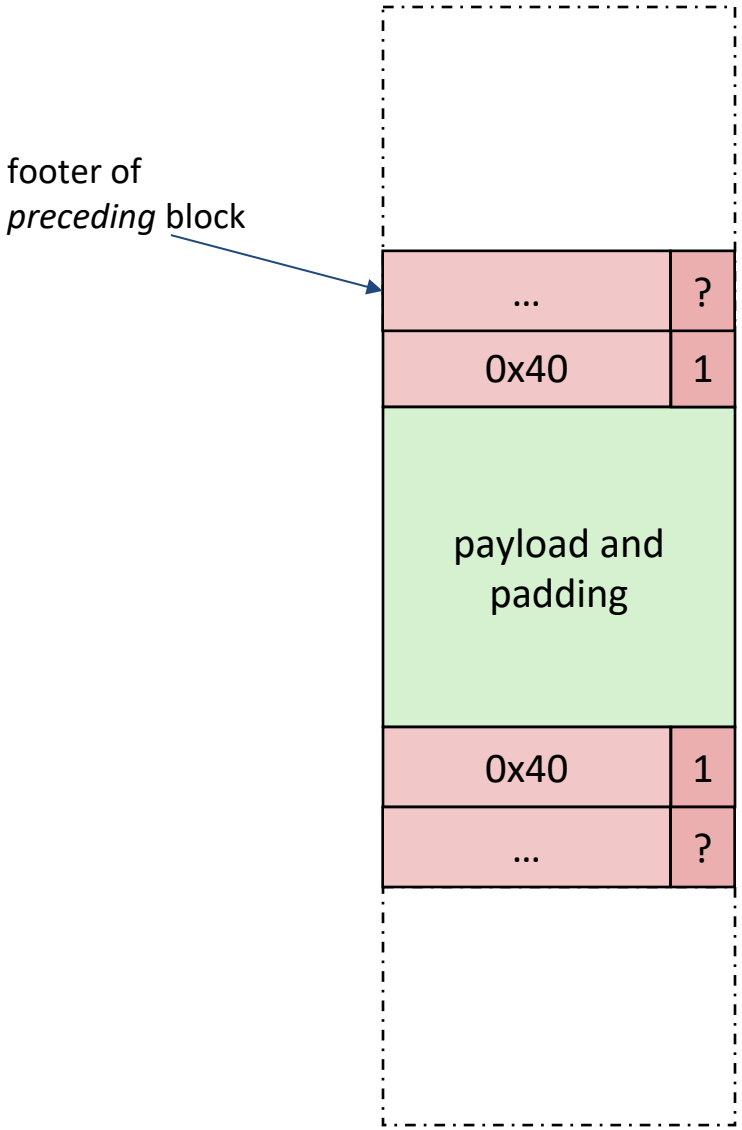
How do we tell if a neighboring block if free?


void* p
free(p);
?
0x40
1
payload and
padding
0x40
1
?

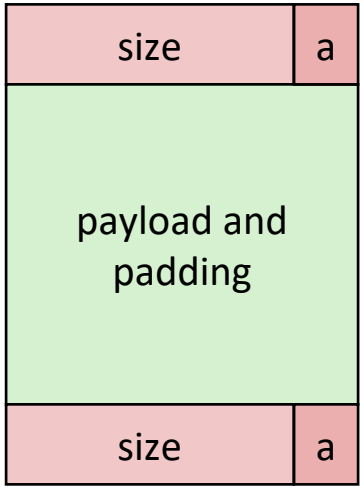
- To determine if the following block is allocated we move by 0x40 bytes to the *following* block and read the header



- To determine if the following block is allocated we move by 0x40 bytes to the *following* block and read the header
- To determine if the previous block is allocated we check the *preceding* block's footer.

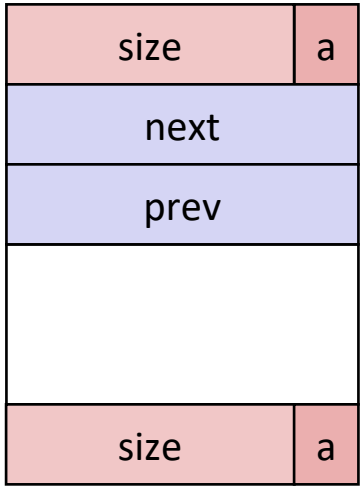


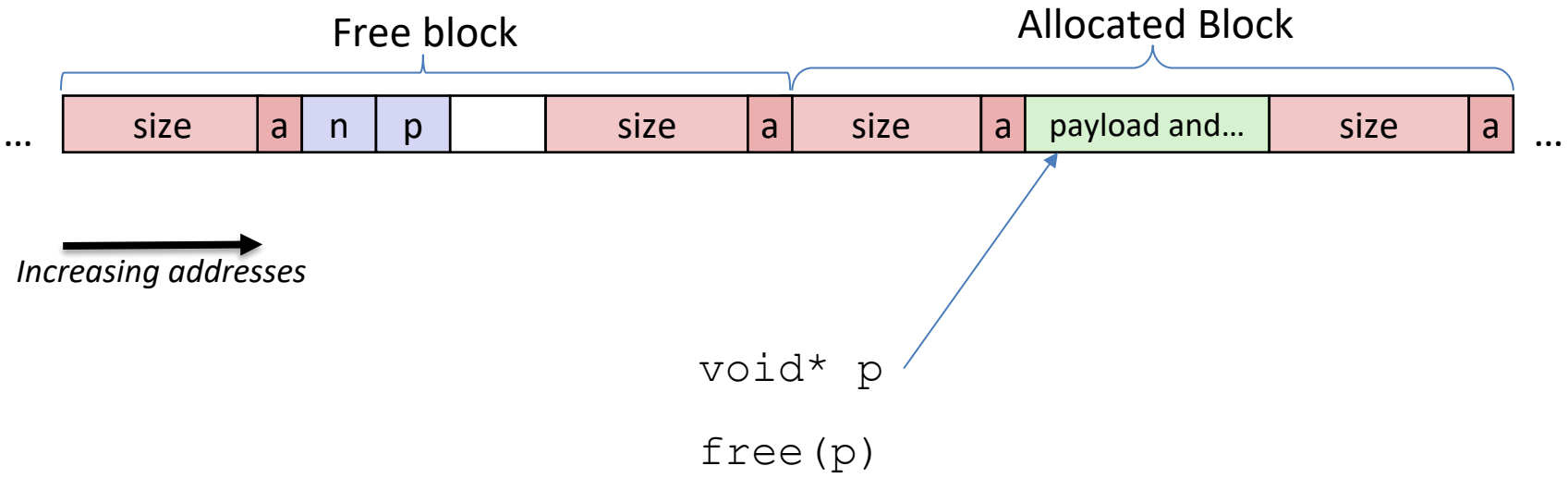
Allocated block:



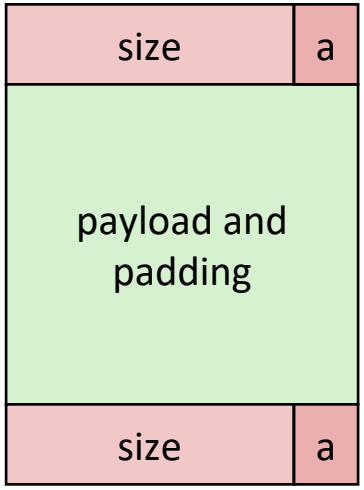
(same as implicit free list)

Free block:



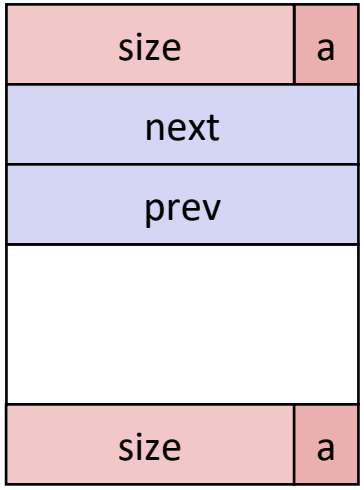


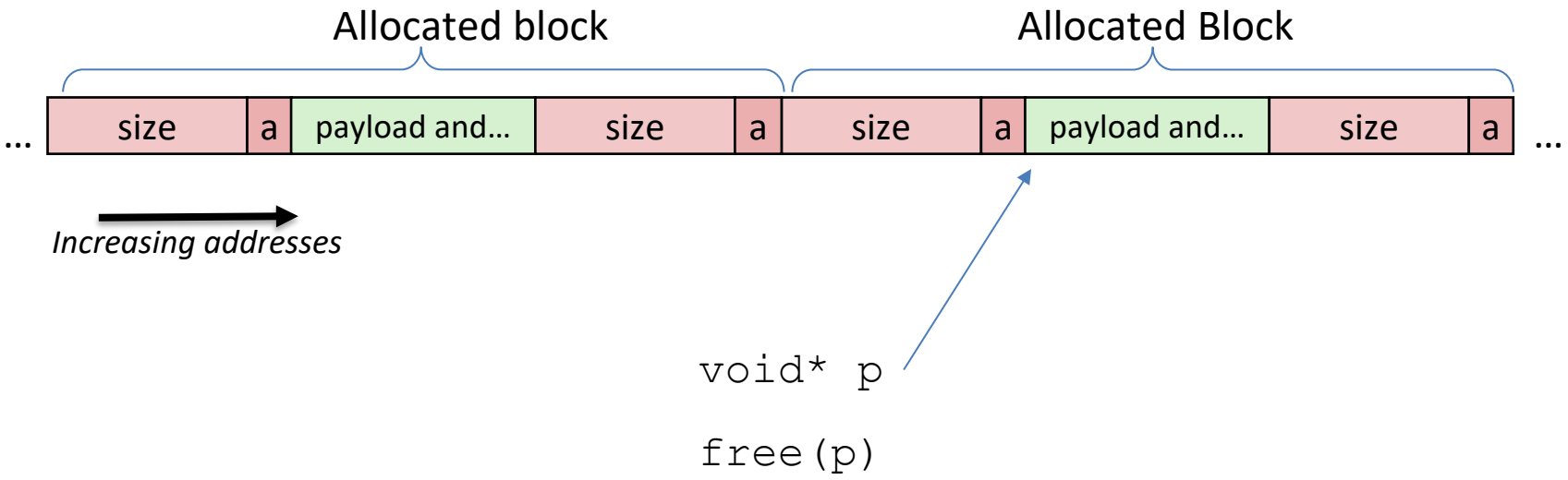
Allocated block:

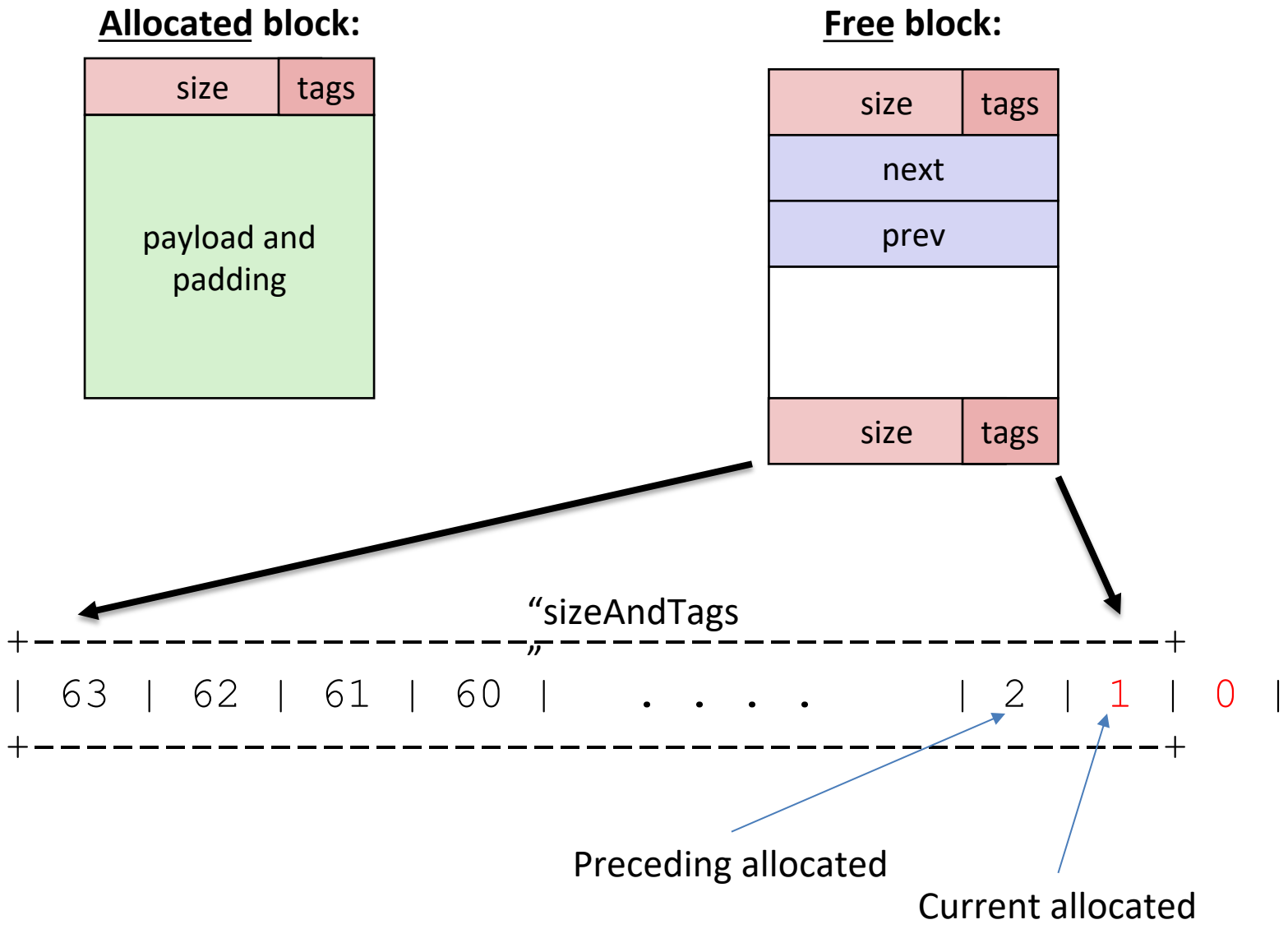


(same as implicit free list)

Free block:

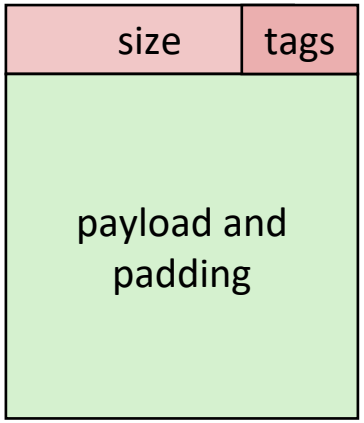





Allocated block:
size
tags
payload and padding
Free block:
size
tags
next
prev
size
tags
"sizeAndTags"
| 63 | 62 | 61 | 60 | . . . . | 2 | 1 | 0 |
Preceding allocated
Current allocated

BlockInfo Struct

Allocated block:



Free block:


```
struct BlockInfo {
  size_t sizeAndTags;
  struct BlockInfo* next;
  struct BlockInfo* prev;
};
typedef struct BlockInfo BlockInfo;
```

Page 2 Section Handout (10 min)

Macros

- Pre-compile time “find and replace”
  - `#define NUM_ENTRIES 100`

- Can be dangerous!
  - `#define twice(x) 2*x`
    - `twice(x+1) => 2*x+1`

  - `#define twice(x) 2*(x)`
    - `twice(x+1) => 2*(x+1)`

Some Provided Macros

- `UNSCALED_POINTER_ADD(p,x)`

  Add without using “pointer arithmetic”

- `UNSCALED_POINTER_SUB(p,x)`

  Subtract without using “pointer arithmetic”

- `MIN_BLOCK_SIZE`

  The size of the smallest block that is safe to allocate

- `SIZE(x)`

  Gets the size from ‘sizeAndTags’

- `TAG_USED`

  Mask for the used tag

- `TAG_PRECEDING_USED`

  Mask for the preceding used tag

- …

There are more. Don’t forget to use them! (or risk losing points on the lab).

Lab 5

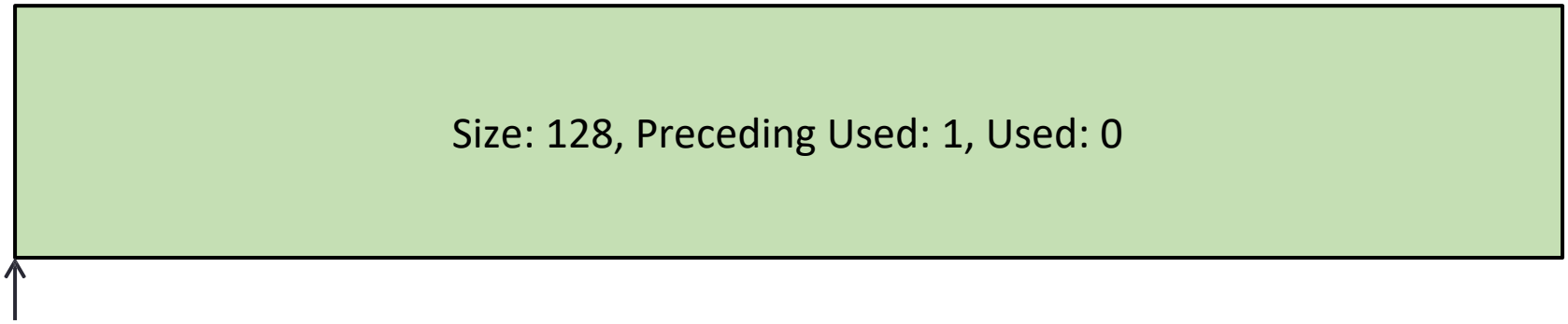
Implement `malloc()` and `free()`

- Before you start to feel overwhelmed…
- We *give you* many functions already including:
  - `searchFreeList()`
  - `insertFreeBlock()`
  - `removeFreeBlock()`
  - `coalesceFreeBlock()`
  - `requestMoreSpace()`

Putting it All Together

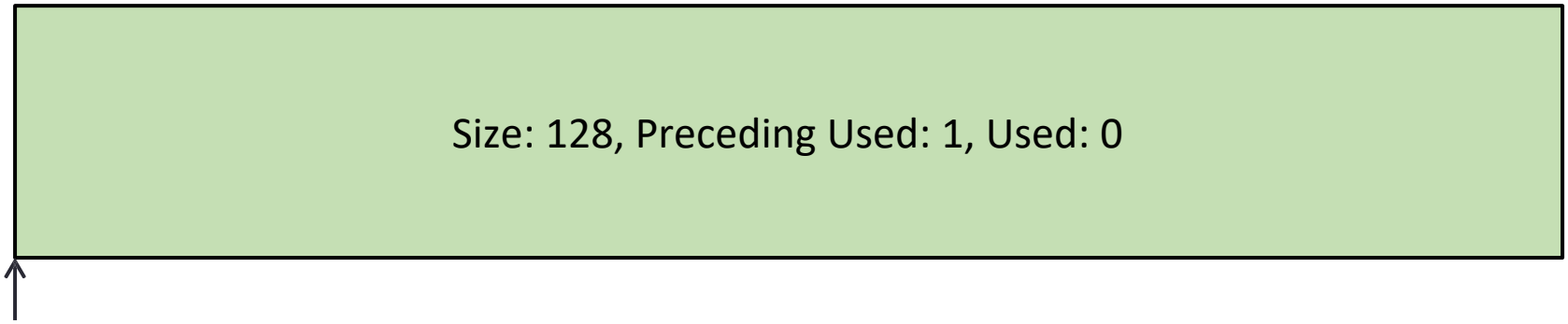
Initial 128-byte heap layout:

- `BlockInfo* FREE_LIST_HEAD` always points to the first block in the free list
- The `BlockInfo` for this free block would look like this:
  - sizeAndTags: 130 (128 + 0x2)
  - next: null
  - prev: null
- The PrecedingUsed tag is set because the previous block is not free (comes into play when we look at coalescing later)



Allocating Blocks – What Happens?

```
void* a = malloc(32)
```

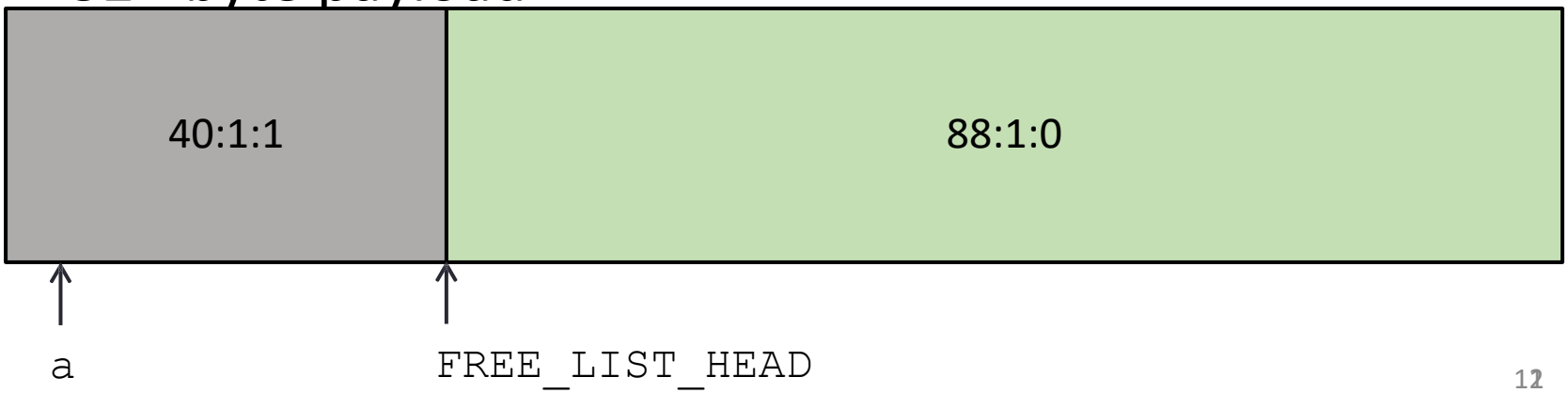


Allocating Blocks

Note: “a” does not point to sizeAndTags! Points to payload, or where the “next” pointer would be stored in the BlockInfo

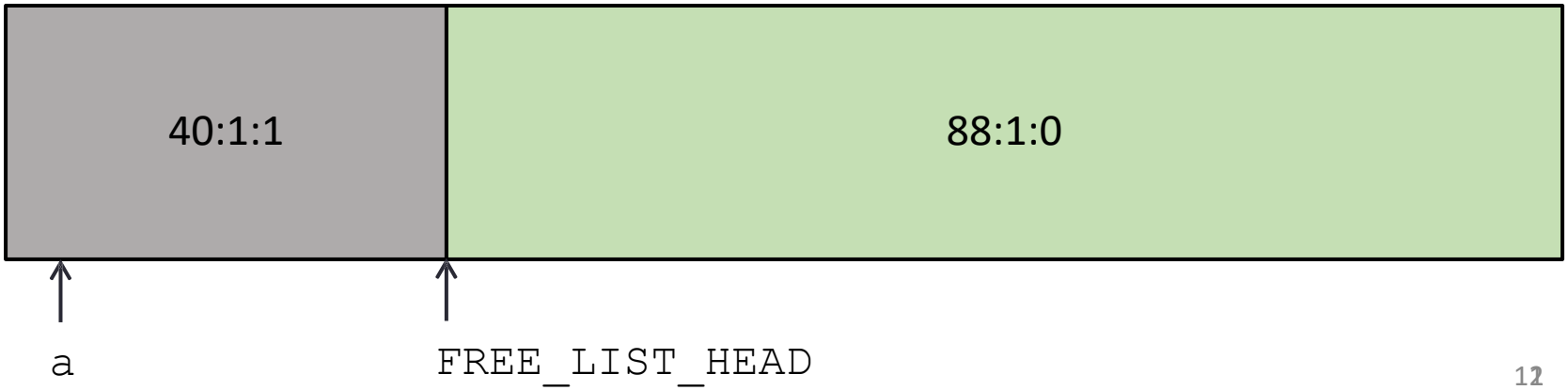
```
void* a = malloc(32)
```

- Searches the free list for a block big enough
- The first (and only) block is 128 bytes, which will work
- Bad implementation: return a 120-byte payload (8-byte header)
- Good implementation: split off 40 bytes, return a 32- byte payload



Allocating Blocks – What Happens?

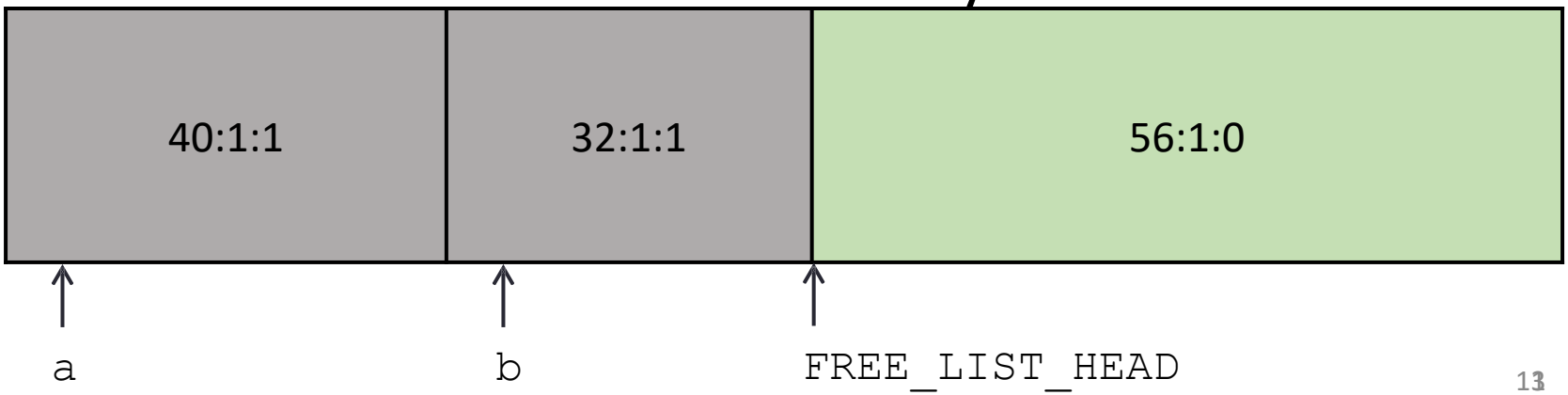
```
void* b = malloc(16)
```



Allocating Blocks

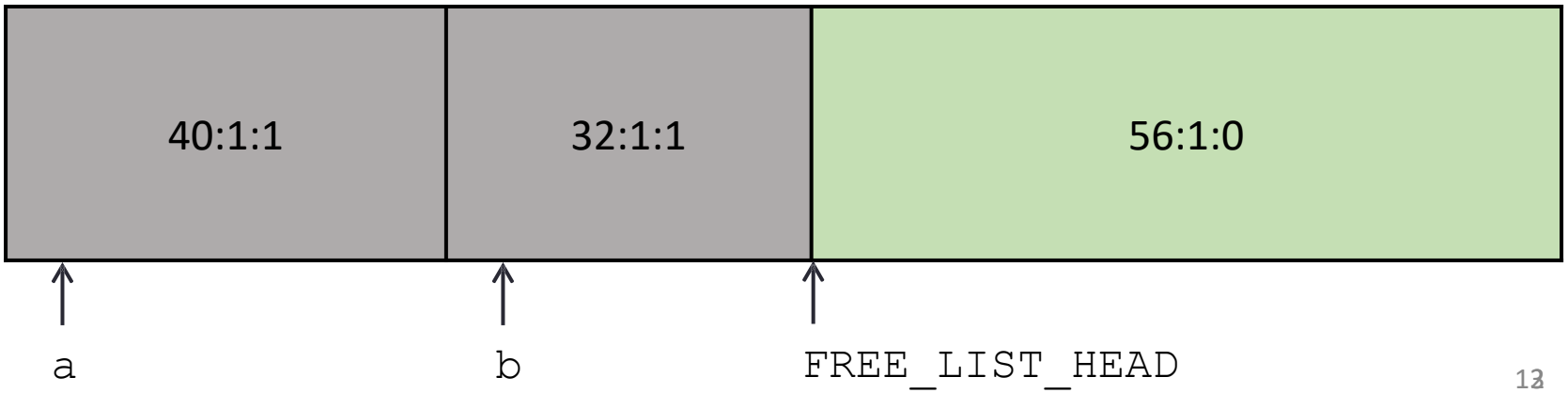
```
void* b = malloc(16)
```

- Only needs a block of 16 + 8 = 24 bytes, but if we were to free this block in the future, we would need at least 32 bytes to create a free block.
- The minimum block size is 32 bytes



Allocating Blocks – What Happens?

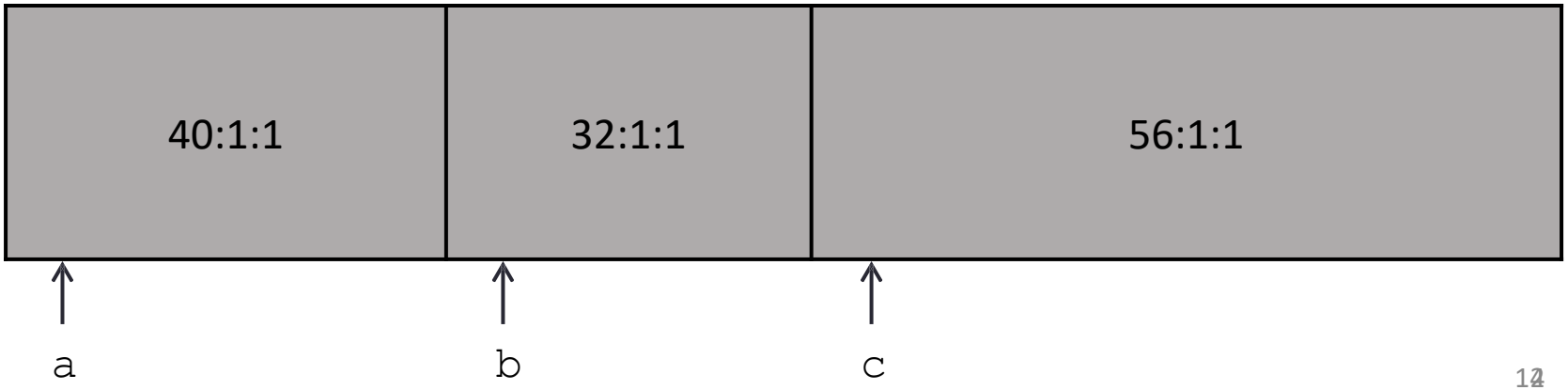
```
void* c = malloc(48)
```



Allocating Blocks

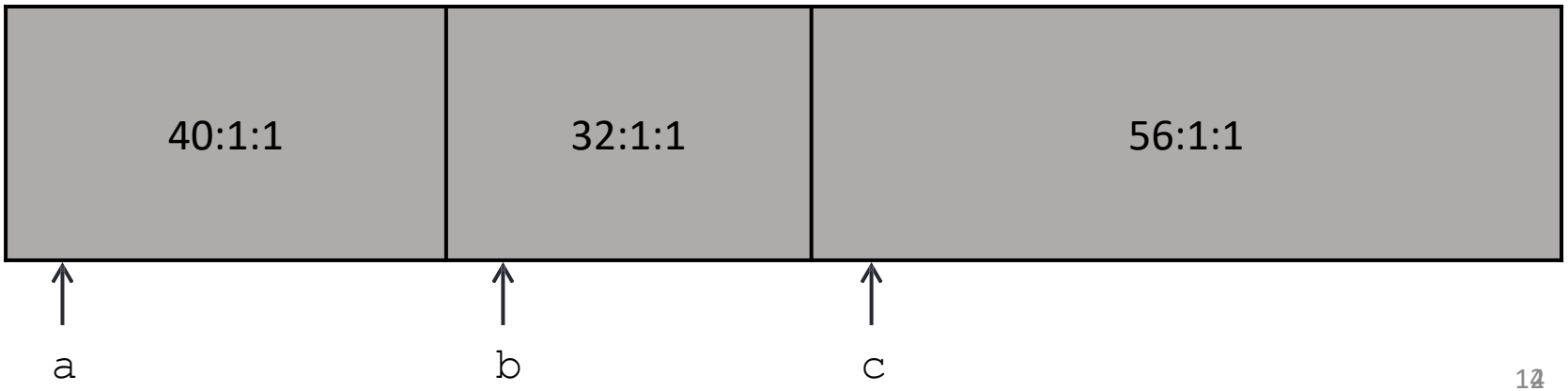
```
void* c = malloc(48)
```

- FREE_LIST_HEAD = null



Freeing Blocks – What Happens?

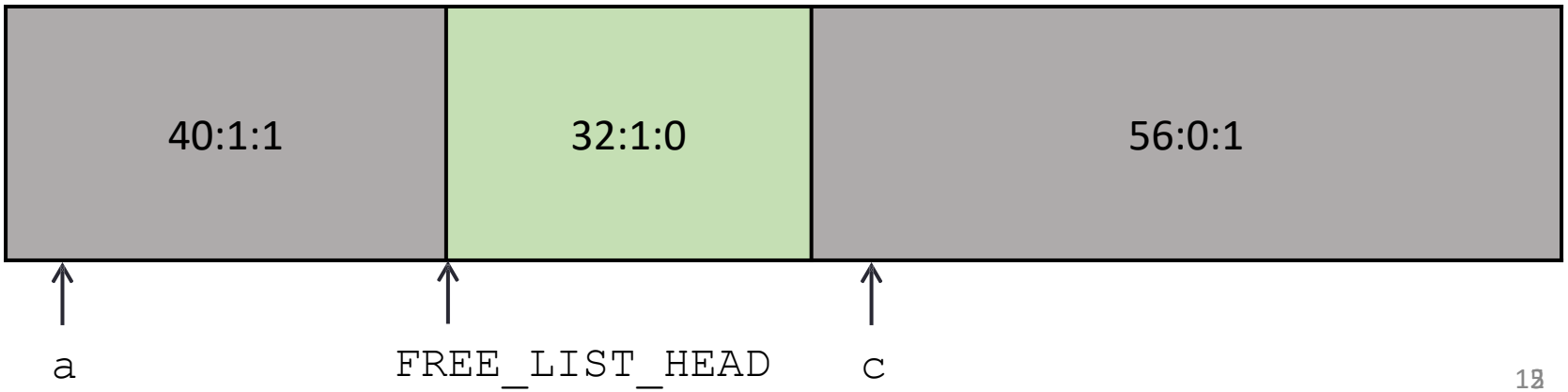
```
free(b)
```



Freeing Blocks

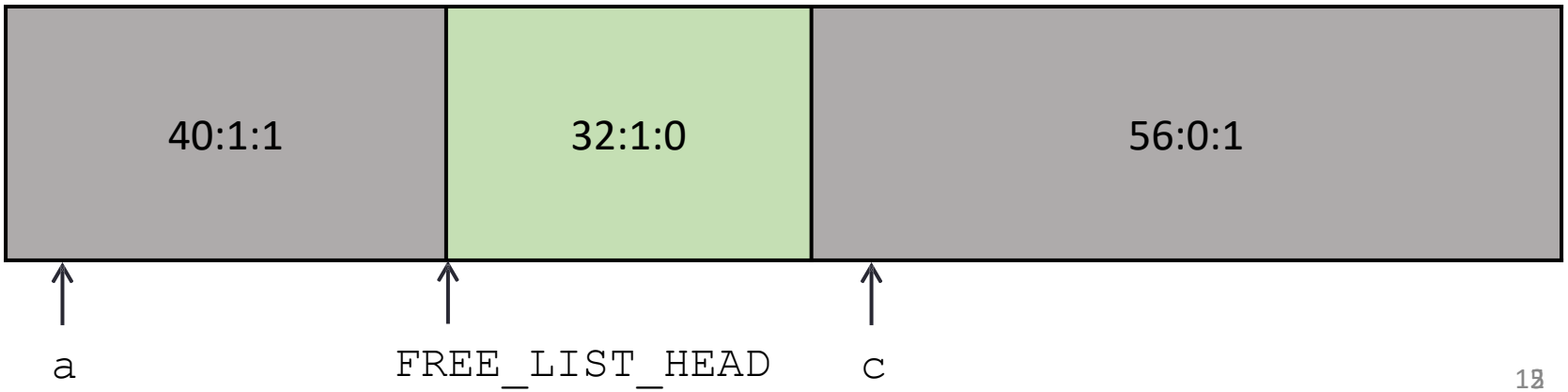
`free(b)`

- Inserts block b into the beginning of the free list
- Notice how the tags in the block after needed to be updated



Freeing Blocks – What Happens?

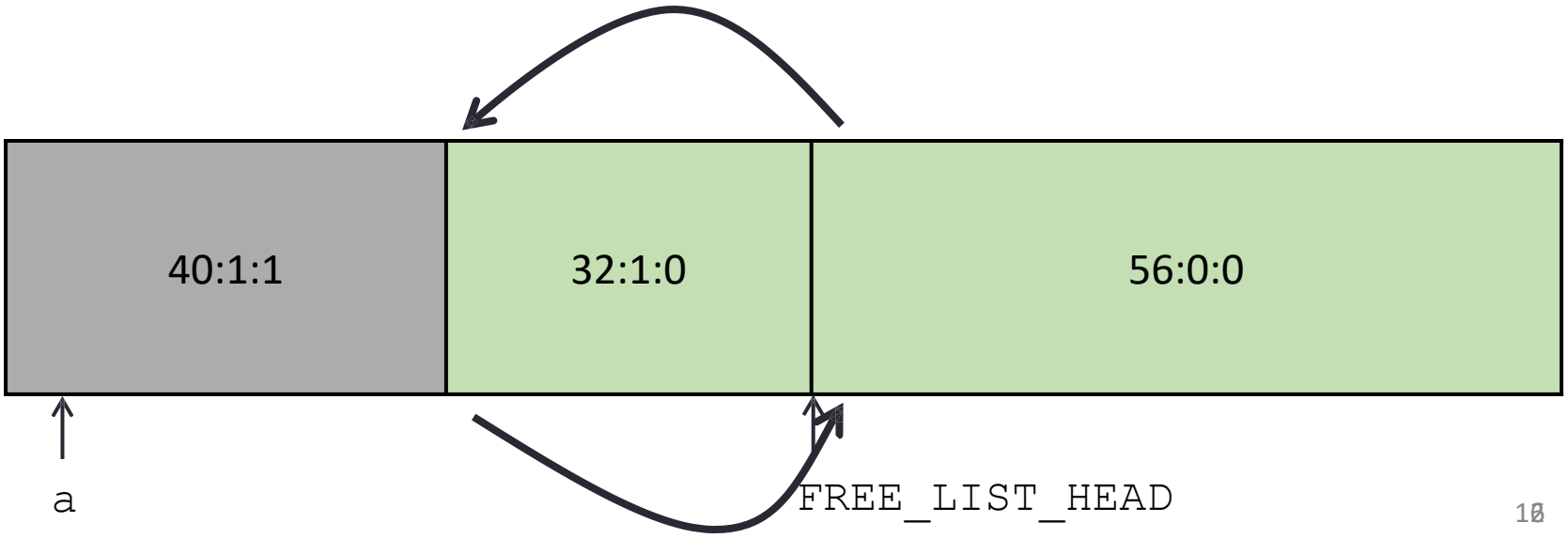
```
free(c)
```



Freeing Blocks

`free(c)`

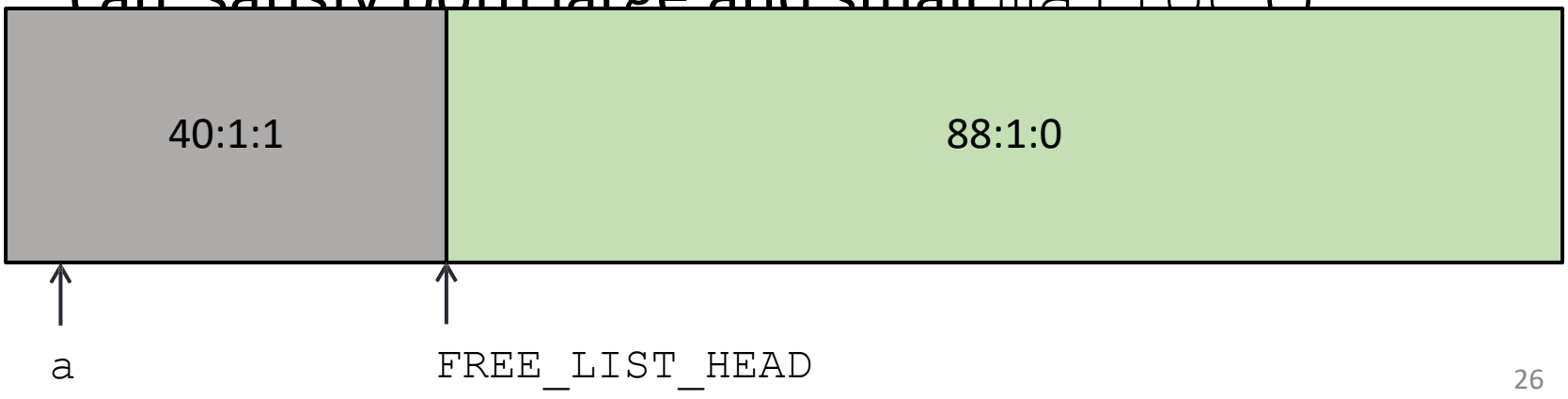
- Is this what the heap should look like at the end of `free(c)`?



Coalesce Free Blocks

When we have multiple free blocks adjacent to each
other in memory, we should coalesce them.

- Coalescing basically combines free blocks together
- Bigger blocks are always better; a large block can satisfy both large and small `malloc()`



Implementing `malloc()`

- Figure out how big a block you need
- Call `searchFreeList()` to get a free block that
  is large enough
  - NOTE: If you request 16 bytes, it might give you a block that is 500 bytes
- Remove that block from the list
- Update size + tags appropriately
- Return a pointer to the payload of that block

Implementing `free()`

- **Remember, the pointer you are passed is to the payload**
- Convert the given used block into a free block
- Insert it into the free list
- Update size + tags appropriately
- Coalesce if necessary by calling `coalesceFreeBlock()`