



CS107 Lecture 22

Managing The Heap, Take II

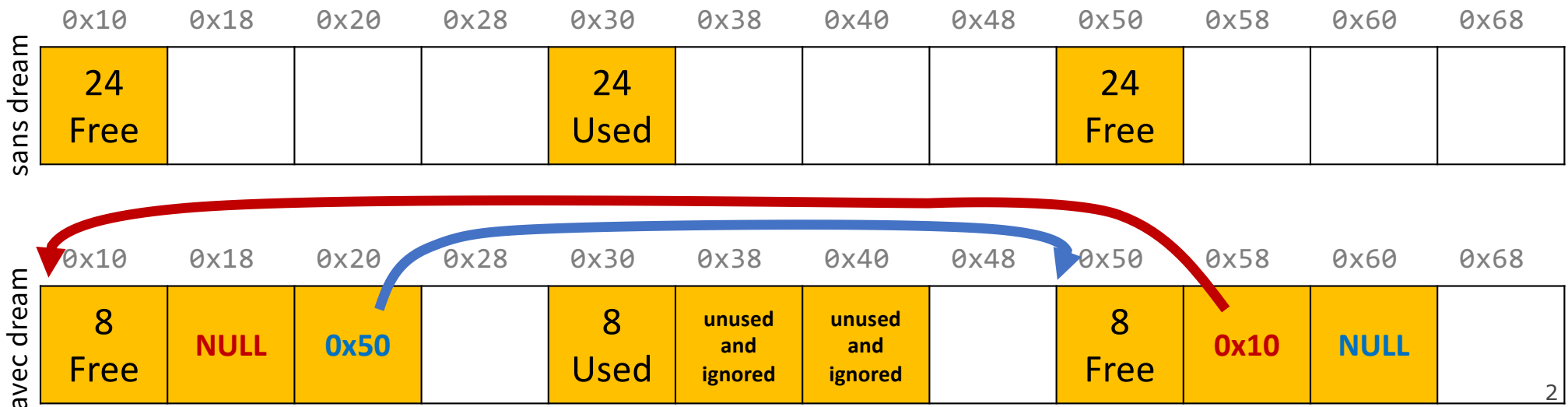
Reading: B&O 9.9 and 9.11

Implicit Lists: Can We Do Better?

The Dream: It would be nice if we could **traverse just the free blocks**, rather than **all of them**, when searching for a fit.

The Idea: Let's increase the header from 8 bytes to 24, using the additional 16 bytes to store **prev** and **next** pointers to other free blocks.

Restated, thread a **doubly-linked list** through just the free nodes.



Implicit Lists: Can We Do Better?

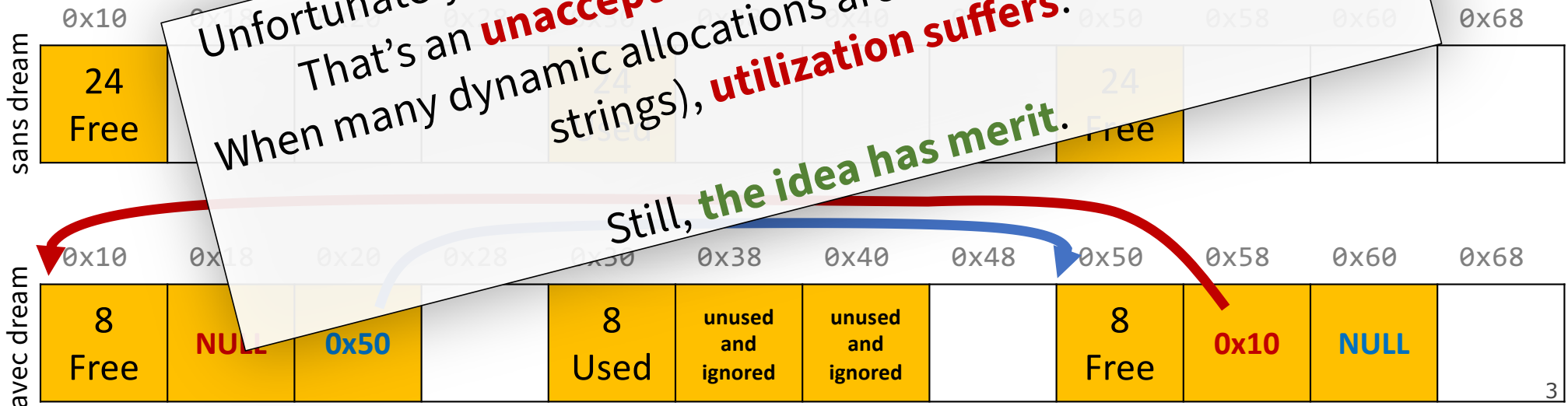
The Dream: It would be nice if we could **traverse just the free blocks**, rather than **all of them**, when searching for a fit.

The Idea: Let's increase the header from 8 bytes to 16 bytes to store **prev** and **next** pointers, using the additional 16 bytes to store **prev** and **next** pointers.

Restated, though, **this isn't ideal**: we've **tripled** the header size. Unfortunately, **this isn't ideal**: we've **tripled** the header size.

That's an **unacceptably large per-block overhead**. When many dynamic allocations are small (e.g., lots of short C strings), **utilization suffers**.

Still, **the idea has merit**.

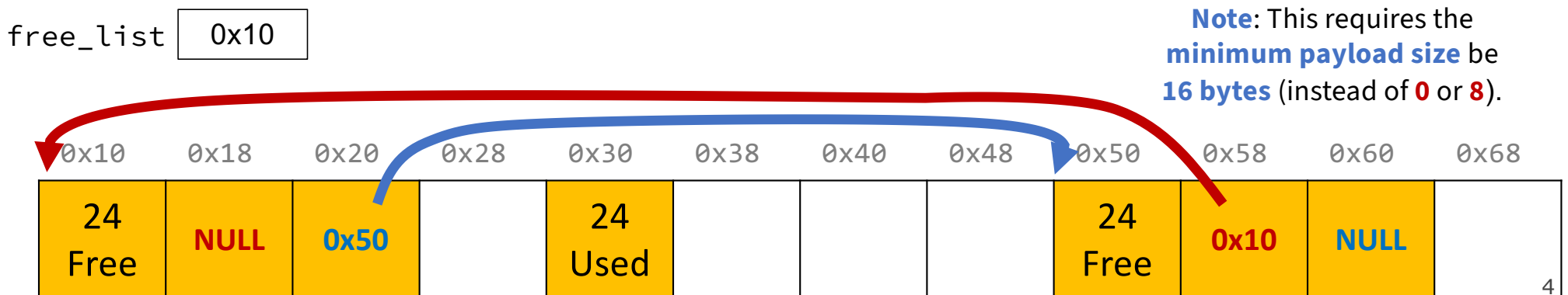


Implicit Lists: Can We Do Better?

The Dream: It would be nice if we could **traverse just the free blocks**, rather than **all of them**, when searching for a fit.

The Idea: Let's increase the header from 8 bytes to 24, using the additional 16 bytes to store **prev** and **next** pointers to other free blocks.

Better Idea: Since free blocks aren't **using their payloads anyway**, we can **secretly stash** the two pointers in the first 16 bytes.



Case Study: Explicit Free List Allocator

The idea **extends our implicit allocator** but stores pointers to the **next** and **previous** free blocks using the first 16 bytes of the free block's payload.

- In response to each **malloc** call, we:
 - **search our list of free blocks**—that is, the **explicit free list**—to find an appropriately sized one,
 - **update its header** to note the block is now allocated,
 - **splice** the now allocated block **out of the free list**, and potentially
 - **create a node** out of any substantial excess payload and **thread it back into the free list**.
- In response to each **free** call, we:
 - **update its header** to note the block is now free, and
 - **thread it into the free list** so it's discoverable by **malloc**

Our list of free blocks is called an **explicit free list**, because there's a doubly-linked list structure **literally weaving its way through all free nodes**.

Case Study: Explicit Free List Allocator

How do you want to **organize your explicit free list**?

You get to choose, but **most choose LIFO**.

- A. address-order** (each block's address is **less than** that of its successor)
- This comes with **better utilization scores**, since first fit—the most commonly used search heuristic—**tends to select nodes at lower addresses**.
 - **free** runs in **linear time**, since the block being freed may be at a higher address and the search for its splice point typically starts at the **front of the free list**.
- B. last-in first-out** (push nodes onto a stack-like data structure, so recently freed blocks appear at the front of the free list)
- **free** runs in **constant time**—that's a huge win.
 - **Utilization scores can suffer**, since lower addresses **aren't prioritized** as they are with an address-order scheme.

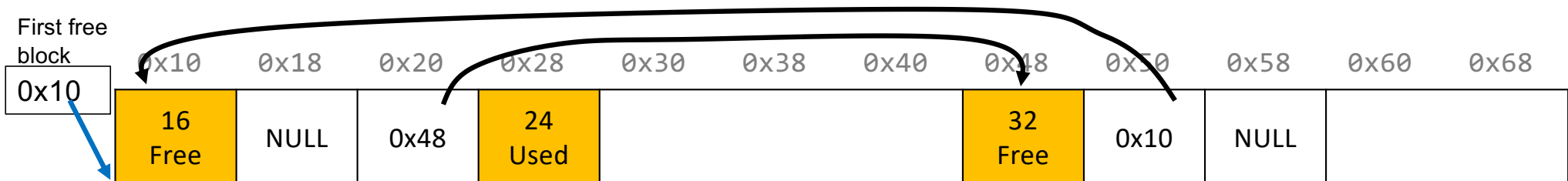
Remember: throughput and utilization fight each other.



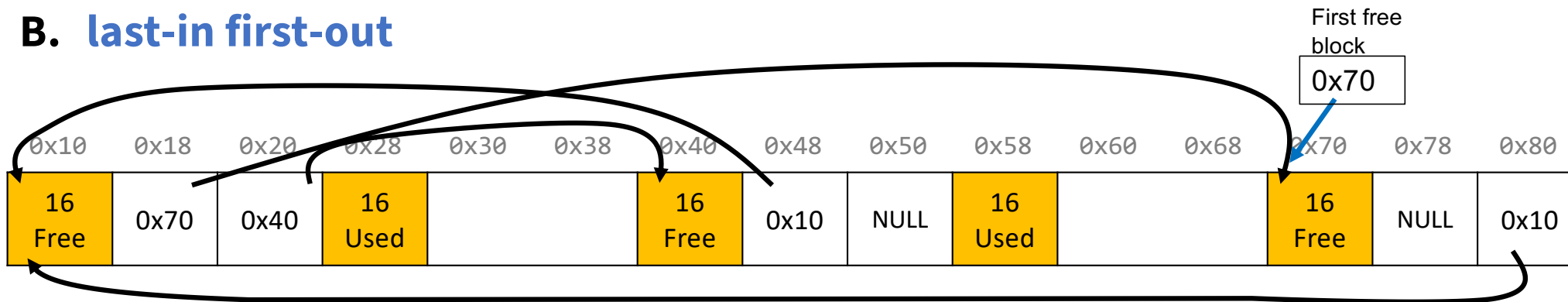
Case Study: Explicit Free List Allocator

How do you want to **organize your explicit free list**?

A. address-order



B. last-in first-out



Case Studies: Implicit versus Explicit

Implicit Free List

- Eight-byte header for **size + allocation** status



- Allocation requests are **worst-case** linear in **total number of blocks**
- Implicitly **address-order**

Explicit Free List

- Eight-byte header for **size + allocation** status
- **Free block payloads store prev and next free block pointers**
- Allocation requests are **worst-case** linear in **number of free blocks**
- Can **choose** block ordering

Heap Allocator Goals Revisited

Can we do better?

1. Can we avoid searching all blocks for free blocks to reuse?

Yes! We can use a doubly-linked list.

2. Can we merge adjacent free blocks to keep large spaces available?

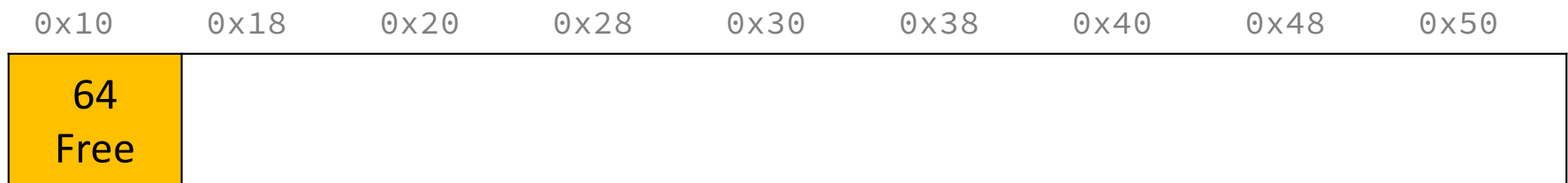
Yes! We'll illustrate this in a few minutes.

3. Can we avoid copying/moving data during **realloc**?

Sometimes! We'll illustrate this next time.

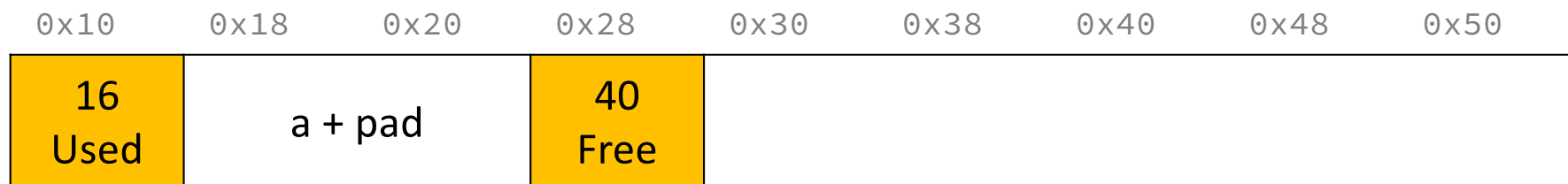
Explicit Allocators and Coalescing

```
void *a = malloc(8);  
void *b = malloc(8);  
void *c = malloc(16);  
free(b);  
free(a);  
void *d = malloc(32);
```



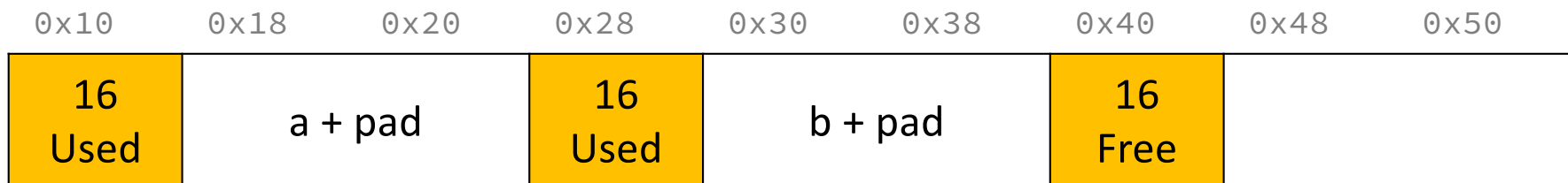
Explicit Allocators and Coalescing

```
void *a = malloc(8);  
void *b = malloc(8);  
void *c = malloc(16);  
free(b);  
free(a);  
void *d = malloc(32);
```



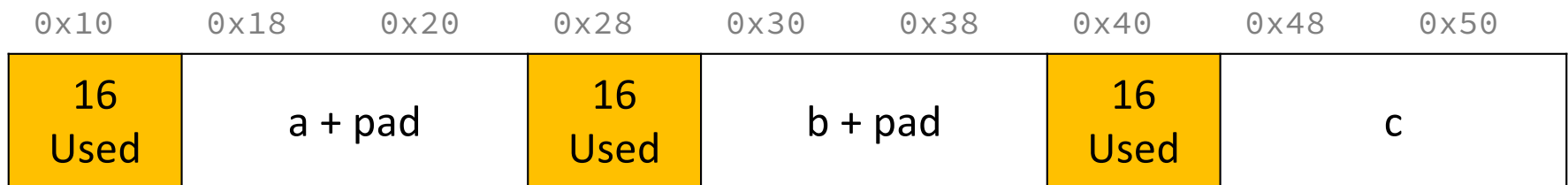
Explicit Allocators and Coalescing

```
void *a = malloc(8);  
void *b = malloc(8);  
void *c = malloc(16);  
free(b);  
free(a);  
void *d = malloc(32);
```



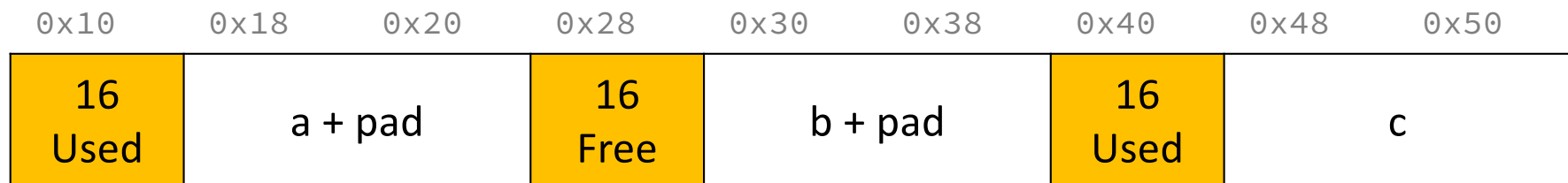
Explicit Allocators and Coalescing

```
void *a = malloc(8);  
void *b = malloc(8);  
void *c = malloc(16);  
free(b);  
free(a);  
void *d = malloc(32);
```



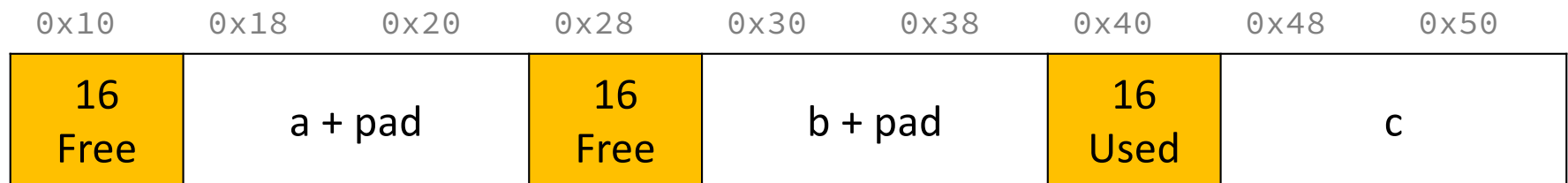
Explicit Allocators and Coalescing

```
void *a = malloc(8);  
void *b = malloc(8);  
void *c = malloc(16);  
free(b);  
free(a);  
void *d = malloc(32);
```



Explicit Allocators and Coalescing

```
void *a = malloc(8);  
void *b = malloc(8);  
void *c = malloc(16);  
free(b);  
free(a);  
void *d = malloc(32);
```

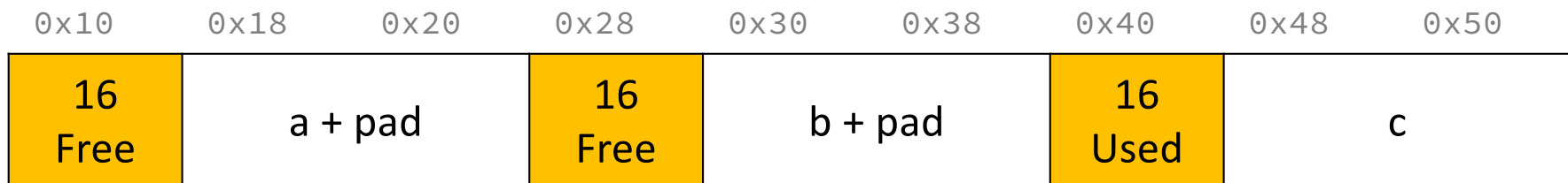


Explicit Allocators and Coalescing

```
void *a = malloc(8);  
void *b = malloc(8);  
void *c = malloc(16);  
free(b);  
free(a);  
void *d = malloc(32);
```

We have the space, but it's **split across two neighboring blocks**. Because the second of the two blocks being freed is the left neighbor of the first, the **first can easily absorb the second** to create a larger, more flexible block.

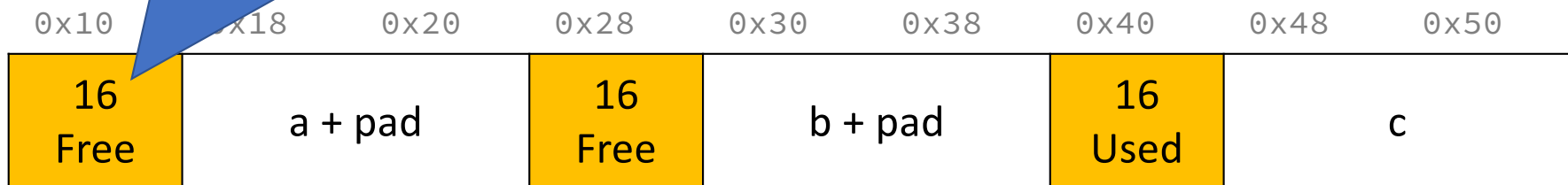
Let's rewind to the second **free** call to see what can be done to **right-coalesce**.



Explicit Allocators and Coalescing

```
void *a = malloc(8);  
void *b = malloc(8);  
void *c = malloc(16);  
free(b);  
free(a);  
void *d = malloc(32);
```

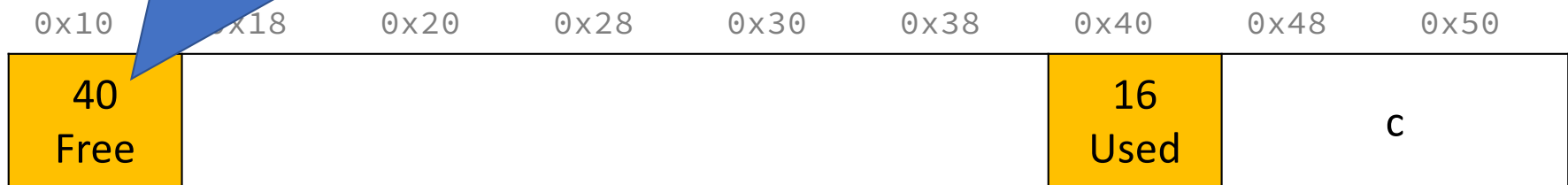
Hey, look! I have a free right neighbor. Let's merge households.



Explicit Allocators and Coalescing

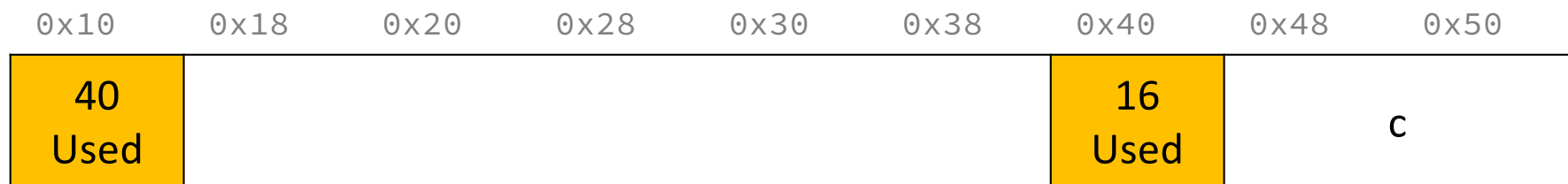
```
void *a = malloc(8);  
void *b = malloc(8);  
void *c = malloc(16);  
free(b);  
free(a);  
void *d = malloc(32);
```

Hey, look! I have a free right neighbor. Let's merge households.



Explicit Allocators and Coalescing

```
void *a = malloc(8);  
void *b = malloc(8);  
void *c = malloc(16);  
free(b);  
free(a);  
void *d = malloc(32);
```



Heap Allocator Goals Revisited

Can we do better?

1. Can we avoid searching all blocks for free blocks to reuse?

Yes! We can use a **doubly-linked list**.

2. Can we merge adjacent free blocks to keep large spaces available?

Yes! We'll illustrate this in a few minutes.

3. Can we avoid copying/moving data during **realloc**?

Sometimes! We'll illustrate this next time.

We'll focus on this come Monday, but if you understand right coalescing, you then you can guess what we do to support in-place **realloc**.