photo by pixabay user @ulleo

# COMP201

## Computer Systems & Programming

Lecture #35 – More on Managing The Heap

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2020

# Recap

- Shared Libraries
- Case study: Library interpositioning
- The heap so far
- What is a heap allocator?
- Heap allocator requirements and goals
- Method 0: Bump Allocator

# Plan for Today

- Method 1: Implicit Free List Allocator
- Method 2: Explicit Free List Allocator

**Disclaimer:** Slides for this lecture were borrowed from

—Nick Troccoli's Stanford CS107 class

# Recap: Heap Allocator Requirements

A heap allocator must...

1. Handle arbitrary request sequences of allocations and frees
2. Keep track of which memory is allocated and which is available
3. Decide which memory to provide to fulfill an allocation request
4. Immediately respond to requests without delay
5. Return addresses that are 8-byte-aligned (must be multiples of 8).

# Recap: Heap Allocator Goals

- <u>Goal 1:</u> Maximize **throughput**, or the number of requests completed per unit time. This means minimizing the average time to satisfy a request.
- <u>Goal 2:</u> Maximize memory **utilization**, or how efficiently we make use of the limited heap memory to satisfy requests.

# Recap: Bump Allocator

- A **bump allocator** is a heap allocator design that simply allocates the next available memory address upon an allocate request and does nothing on a free request.
- Throughput: each `malloc` and `free` execute only a handful of instructions:
    - It is easy to find the next location to use
    - Free does nothing!
- Utilization: we use each memory block at most once. No freeing at all, so no memory is ever reused. ☹

# Recap: Bump Allocator

- A bump allocator is an extreme heap allocator – it optimizes only for **throughput**, not **utilization**.
- Better allocators strike a more reasonable balance.  How can we do this?

Questions to consider:

1. How do we keep track of free blocks?
2. How do we choose an appropriate free block in which to place a newly allocated block?
3. After we place a newly allocated block in some free block, what do we do with the remainder of the free block?
4. What do we do with a block that has just been freed?

# Lecture Plan

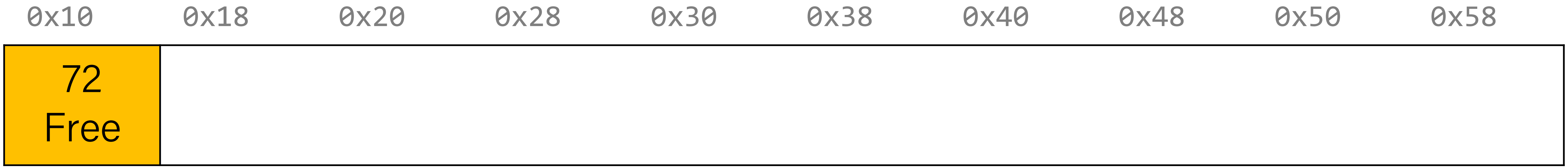
- Method 1: Implicit Free List Allocator
- Method 2: Explicit Free List Allocator

# Implicit Free List Allocator

- **Key idea:** in order to reuse blocks, we need a way to track which blocks are allocated and which are free.
- We could store this information in a separate global data structure, but this is inefficient.
- Instead: let's allocate extra space before each block for a **header** storing its payload size and whether it is allocated or free.
- When we allocate a block, we look through the blocks to find a free one, and we update its header to reflect its allocated size and that it is now allocated.
- When we free a block, we update its header to reflect it is now free.
- The header should be 8 bytes (or larger).
- By storing the block size of each block, we *implicitly* have a *list* of free blocks.

# Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```



# Implicit Free List Allocator

```c
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

| Variable | Value |
|---|---|
| a | 0x18 |

| 0x10 | 0x18 | 0x20 | 0x28 | 0x30 | 0x38 | 0x40 | 0x48 | 0x50 | 0x58 |
|---|---|---|---|---|---|---|---|---|---|
| 8 Used | a + pad | 56 Free | | | | | | | |

# Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

| Variable | Value |
|---|---|
| a | 0x18 |
| b | 0x28 |

| 0x10 | 0x18 | 0x20 | 0x28 | 0x30 | 0x38 | 0x40 | 0x48 | 0x50 | 0x58 |
|---|---|---|---|---|---|---|---|---|---|
| 8 Used | a + pad | 8 Used | b | 40 Free | | | | | |

# Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

| Variable | Value |
|---|---|
| a | 0x18 |
| b | 0x28 |
| c | 0x38 |

| 0x10 | 0x18 | 0x20 | 0x28 | 0x30 | 0x38 | 0x40 | 0x48 | 0x50 | 0x58 |
|---|---|---|---|---|---|---|---|---|---|
| 8<br>Used | a +<br>pad | 8<br>Used | b | 8<br>Used | c +<br>pad | 24<br>Free | | | |

# Implicit Free List Allocator

```c
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

| Variable | Value |
|---|---|
| a | 0x18 |
| b | 0x28 |
| c | 0x38 |

| 0x10 | 0x18 | 0x20 | 0x28 | 0x30 | 0x38 | 0x40 | 0x48 | 0x50 | 0x58 |
|---|---|---|---|---|---|---|---|---|---|
| 8 Used | a + pad | 8 Free | b | 8 Used | c + pad | 24 Free | | | |

# Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

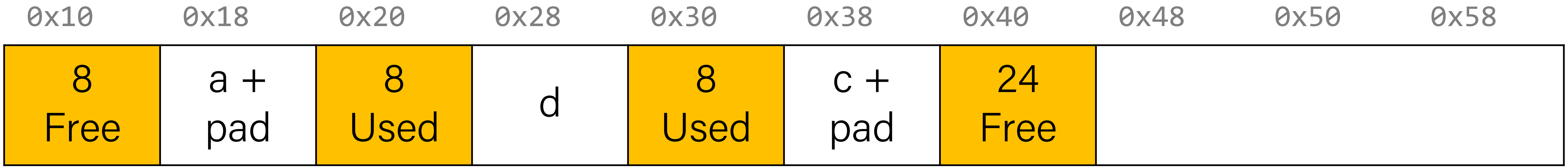
| Variable | Value |
|---|---|
| a | 0x18 |
| b | 0x28 |
| c | 0x38 |
| d | 0x28 |

| 0x10 | 0x18 | 0x20 | 0x28 | 0x30 | 0x38 | 0x40 | 0x48 | 0x50 | 0x58 |
|---|---|---|---|---|---|---|---|---|---|
| 8 Used | a + pad | 8 Used | d | 8 Used | c + pad | 24 Free | | | |

# Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

| Variable | Value |
|---|---|
| a | 0x18 |
| b | 0x28 |
| c | 0x38 |
| d | 0x28 |



# Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

| Variable | Value |
|---|---|
| a | 0x18 |
| b | 0x28 |
| c | 0x38 |
| d | 0x28 |
| e | 0x48 |

| 0x10 | 0x18 | 0x20 | 0x28 | 0x30 | 0x38 | 0x40 | 0x48 | 0x50 | 0x58 |
|---|---|---|---|---|---|---|---|---|---|
| 8 Free | a + pad | 8 Used | d | 8 Used | c + pad | 24 Used | e | | |

# Implicit Free List Allocator

```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```

| Variable | Value |
|---|---|
| a | 0x18 |
| b | 0x28 |
| c | 0x38 |
| d | 0x28 |
| e | 0x48 |

| 0x10 | 0x18 | 0x20 | 0x28 | 0x30 | 0x38 | 0x40 | 0x48 | 0x50 | 0x58 |
|---|---|---|---|---|---|---|---|---|---|
| 8 Free | a + pad | 8 Used | d | 8 Used | c + pad | 24 Used | e | | |

# Representing Headers

How can we store both a size and a status (Free/Allocated) in 8 bytes?

Int for size, int for status? **no! malloc/realloc use size_t for sizes!**

**Key idea:** block sizes will *always be multiples of 8*. (Why?)

- Least-significant 3 bits will be unused!
- *Solution:* use one of the 3 least-significant bits to store free/allocated status

# Implicit Free List Allocator

How can we choose a free block to use for an allocation request?

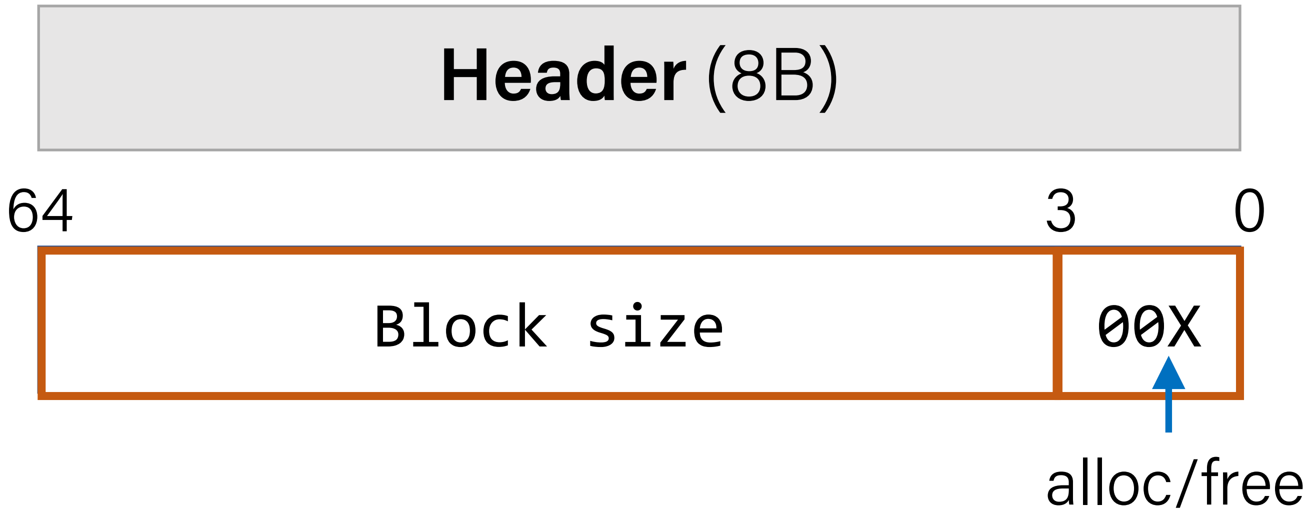
- **First fit:** search the list from beginning each time and choose first free block that fits.
- **Next fit:** instead of starting at the beginning, continue where previous search left off.
- **Best fit:** examine every free block and choose the one with the smallest size that fits.

- First fit/next fit easier to implement
- What are the pros/cons of each approach?

# Implicit Free List Summary

For **all blocks**,

- Have a header that stores size and status.
- Our list links *all* blocks, allocated (A) and free (F).




Keeping track of free blocks:

- Improves memory utilization (vs bump allocator)
- Decreases throughput (worst case allocation request has O(A + F) time)
- Increases design complexity ☺

# Splitting Policy

Up to you!

```
...
void *e = malloc(16);
```

So far, we have seen that a reasonable allocation request splits a free block into an allocated block and a free block with remaining space. **What about edge cases?**

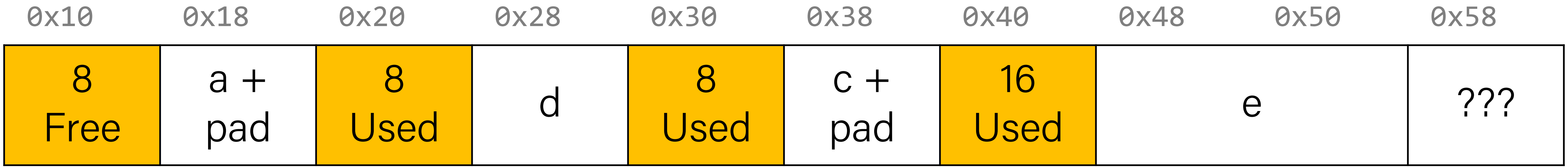
| 0x10 | 0x18 | 0x20 | 0x28 | 0x30 | 0x38 | 0x40 | 0x48 | 0x50 | 0x58 |
|---|---|---|---|---|---|---|---|---|---|
| 8 Free | a + pad | 8 Used | d | 8 Used | c + pad | 24 Free | | | |

# Splitting Policy

Up to you!

```
...
void *e = malloc(16);
```

So far, we have seen that a reasonable allocation request splits a free block into an allocated block and a free block with remaining space. **What about edge cases?**



# Splitting Policy

Up to you!

```
...
void *e = malloc(16);
```

So far, we have seen that a reasonable allocation request splits a free block into an allocated block and a free block with remaining space. **What about edge cases?**

**A. Throw into allocation for** e **as extra padding?** *Internal fragmentation – unused bytes because of padding*

| 0x10 | 0x18 | 0x20 | 0x28 | 0x30 | 0x38 | 0x40 | 0x48 | 0x50 | 0x58 |
|---|---|---|---|---|---|---|---|---|---|
| 8 Free | a + pad | 8 Used | d | 8 Used | c + pad | 24 Used | e + pad | | |

# Splitting Policy

Up to you!

```
...
void *e = malloc(16);
```

So far, we have seen that a reasonable allocation request splits a free block into an allocated block and a free block with remaining space. **What about edge cases?**

A. Throw into allocation for **e** as extra padding?
**B. Make a "zero-byte free block"?** *External fragmentation – unused free blocks*

| 0x10 | 0x18 | 0x20 | 0x28 | 0x30 | 0x38 | 0x40 | 0x48 | 0x50 | 0x58 |
|---|---|---|---|---|---|---|---|---|---|
| 8<br>Free | a +<br>pad | 8<br>Used | d | 8<br>Used | c +<br>pad | 16<br>Used | e | | 0<br>Free |

# Revisiting Our Goals

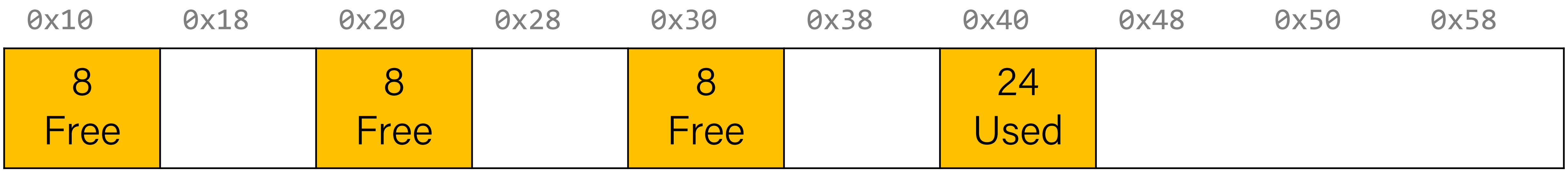
Questions we considered:

1. How do we keep track of free blocks? **Using headers!**
2. How do we choose an appropriate free block in which to place a newly allocated block? **Iterate through all blocks.**
3. After we place a newly allocated block in some free block, what do we do with the remainder of the free block? **Try to make the most of it!**
4. What do we do with a block that has just been freed? **Update its header!**

# Coalescing

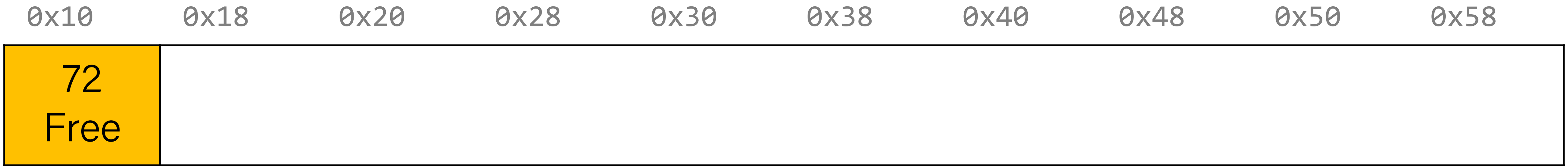
```
void *e = malloc(24);   // returns NULL!
```

You do not need to worry about this problem for the implicit allocator, but this is a requirement for the *explicit* allocator! (More about this later).



# In-Place Realloc

```
void *a = malloc(4);
void *b = realloc(a, 8);
```



# In-Place Realloc

```
void *a = malloc(4);
void *b = realloc(a, 8);
```

| Variable | Value |
|---|---|
| a | 0x10 |

| 0x10 | 0x18 | 0x20 | 0x28 | 0x30 | 0x38 | 0x40 | 0x48 | 0x50 | 0x58 |
|---|---|---|---|---|---|---|---|---|---|
| 8 Used | a + pad | 56 Free | | | | | | | |

# In-Place Realloc

```
void *a = malloc(4);
void *b = realloc(a, 8);
```

| Variable | Value |
|---|---|
| a | `0x10` |
| b | `0x28` |

> The implicit allocator can always move memory to a new location for a **`realloc`** request. The *explicit* allocator must support in-place **`realloc`** (more on this later).

| 0x10 | 0x18 | 0x20 | 0x28 | 0x30 | 0x38 | 0x40 | 0x48 | 0x50 | 0x58 |
|---|---|---|---|---|---|---|---|---|---|
| 8<br>Free | a +<br>pad | 8<br>Used | b | 40<br>Free | | | | | |

# Summary: Implicit Allocator

- An implicit allocator is a more efficient implementation that has reasonable **throughput** and **utilization** due to its recycling of blocks.

Can we do better?

1. Can we avoid searching all blocks for free blocks to reuse?
2. Can we merge adjacent free blocks to keep large spaces available?
3. Can we avoid always copying/moving data during `realloc`?

# Lecture Plan

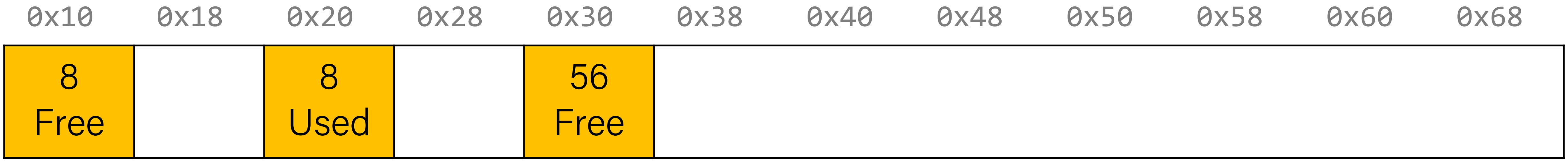
- Method 1: Implicit Free List Allocator
- Method 2: Explicit Free List Allocator

# Lecture Plan

- Method 1: Implicit Free List Allocator
- Method 2: Explicit Free List Allocator
  - Explicit Allocator
  - Coalescing
  - In-place realloc

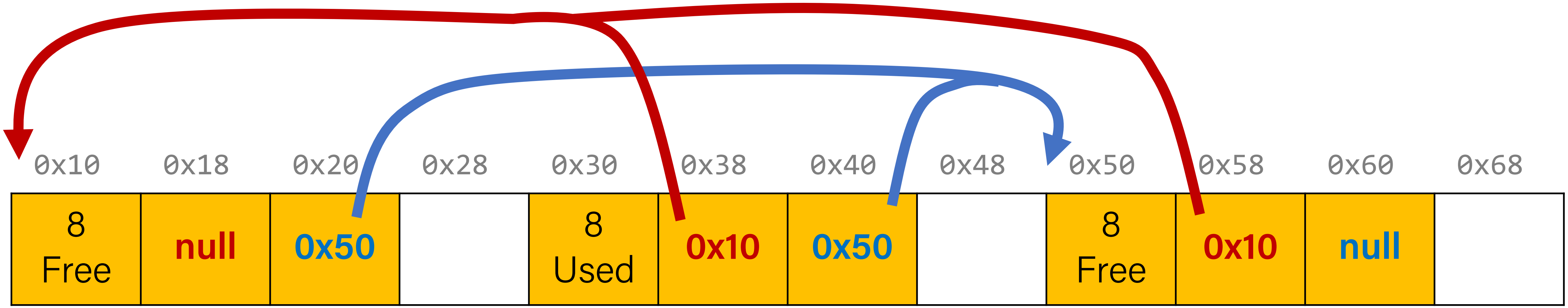
# Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the previous free block and a pointer to the next free block.



# Can We Do Better?

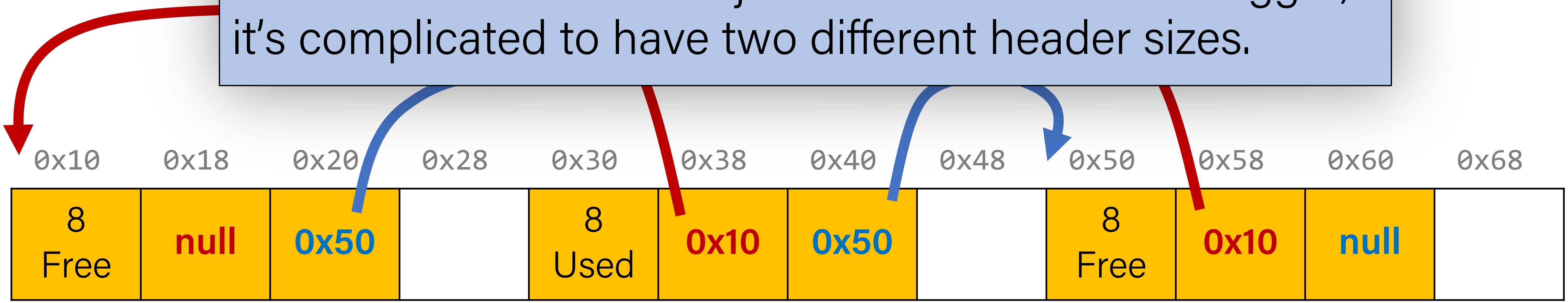
- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the **previous** free block and a pointer to the **next** free block.



# Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the **previous** free block and a pointer to the **next** free block.

This is inefficient – it triples the size of *every* header, when we just need to jump from one <u>free</u> block to another. And even if we just made free headers bigger, it's complicated to have two different header sizes.



# Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the previous free block and a pointer to the next free block.

# Can We Do Better?

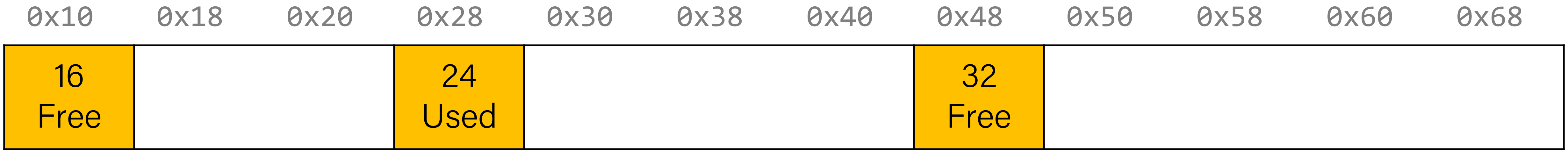
- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the previous free block and a pointer to the next free block. *This is inefficient / complicated.*
- **Where can we put these pointers to the next/previous free block?**
- **Idea:** In a separate data structure?

# Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the previous free block and a pointer to the next free block. *This is inefficient / complicated.*
- **Where can we put these pointers to the next/previous free block?**
- **Idea:** In a separate data structure? *More difficult to access in a separate place – prefer storing near blocks on the heap itself.*

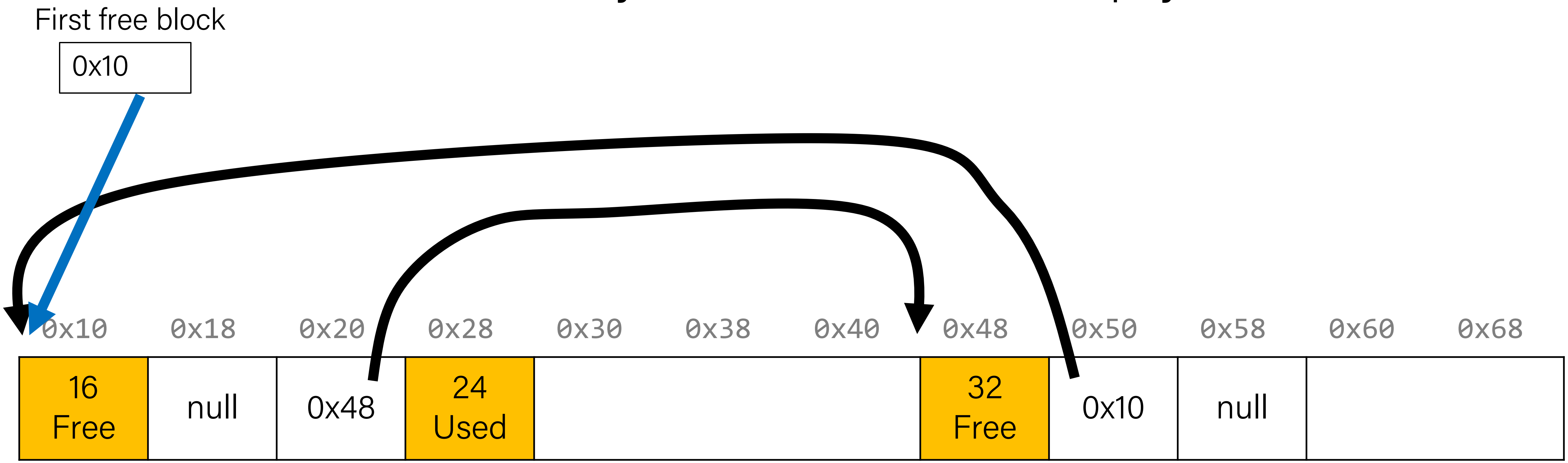
# Can We Do Better?

- **Key Insight:** the payloads of the free blocks aren't being used, because they're free.
- **Idea:** since we only need to store these pointers for free blocks, let's store them in the <u>first 16 bytes of each free block's payload!</u>



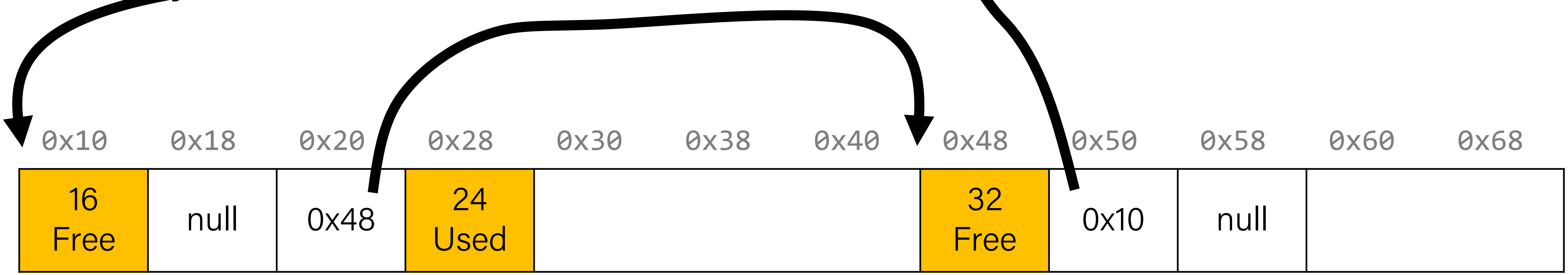
# Can We Do Better?

- **Key Insight:** the payloads of the free blocks aren't being used, because they're free.
- **Idea:** since we only need to store these pointers for free blocks, let's store them in the <u>first 16 bytes of each free block's payload!</u>



# Can We Do Better?

- **Key Insight:** the payloads of the free blocks aren't being used, because they're free.
- **Idea:** since we only need to store these pointers for free blocks, let's store them in the <u>first 16 bytes of each free block's payload!</u>
- This means each payload must be big enough to store 2 pointers (16 bytes). So we must require that for every block, free <u>and allocated.</u> (why?)



# Explicit Free List Allocator

- This design builds on the implicit allocator, but also stores pointers to the next and previous free block inside each free block's payload.
- When we allocate a block, we look through just the free blocks using our linked list to find a free one, and we update its header and the linked list to reflect its allocated size and that it is now allocated.
- When we free a block, we update its header to reflect it is now free and update the linked list.

This **explicit** list of free blocks increases request throughput, with some costs (design and internal fragmentation)

# Explicit Free List: List Design

How do you want to organize your explicit free list?
(compare utilization/throughput)

Up to you!

A. Address-order (each block's address is less than successor block's address)

   Better memory util, Linear free

B. Last-in first-out (LIFO)/like a stack, where newly freed blocks are at the beginning of the list

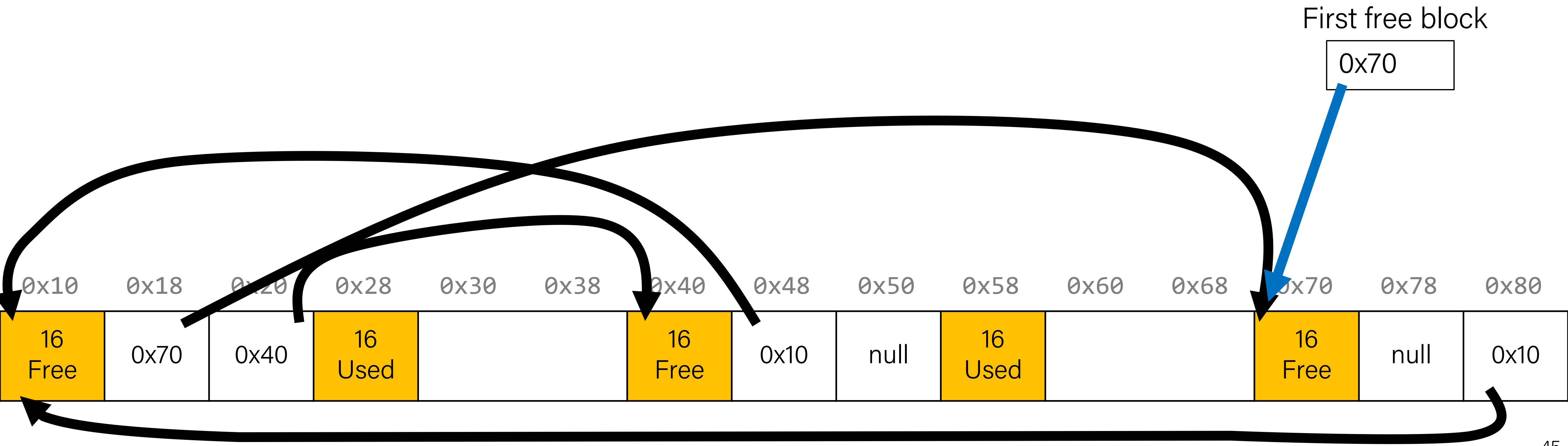
   Constant free (push recent block onto stack)

C. Other (e.g., by size, etc.)

   (more at end of lecture)

# Explicit Free List: List Design

Note that the doubly-linked list *does not have to be in address order*.



# Implicit vs. Explicit: So Far

## Implicit Free List

- 8B header for size + alloc/free status
- Allocation requests are worst-case linear in total number of blocks
- Implicitly address-order

## Explicit Free List

- 8B header for size + alloc/free status
- Free block payloads store prev/next free block pointers
- Allocation requests are worst-case linear in number of free blocks
- Can choose block ordering

# Revisiting Our Goals

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

# Revisiting Our Goals

Can we do better?

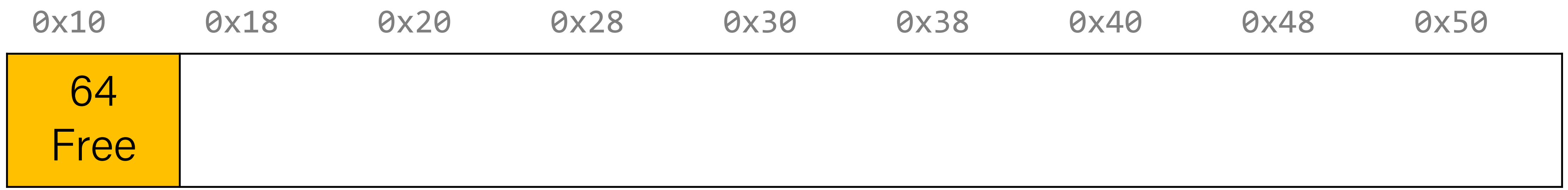
1. Can we avoid searching all blocks for free blocks to reuse? **Yes! We can use a doubly-linked list.**
2. Can we merge adjacent free blocks to keep large spaces available?
3. Can we avoid always copying/moving data during `realloc`?

# Lecture Plan

- Method 1: Implicit Free List Allocator
- Method 2: Explicit Free List Allocator
  - Explicit Allocator
  - Coalescing
  - In-place realloc

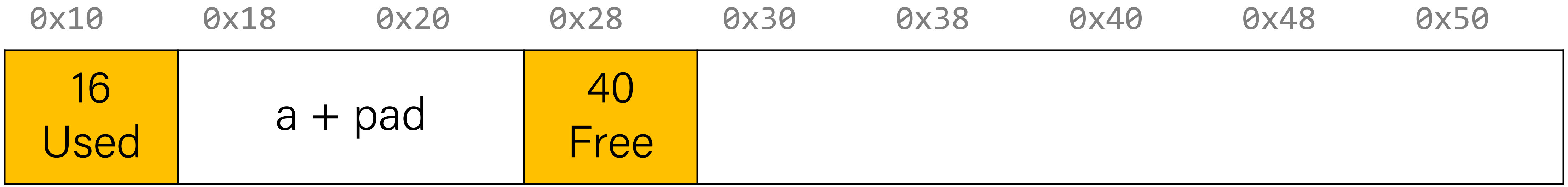
# Coalescing

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



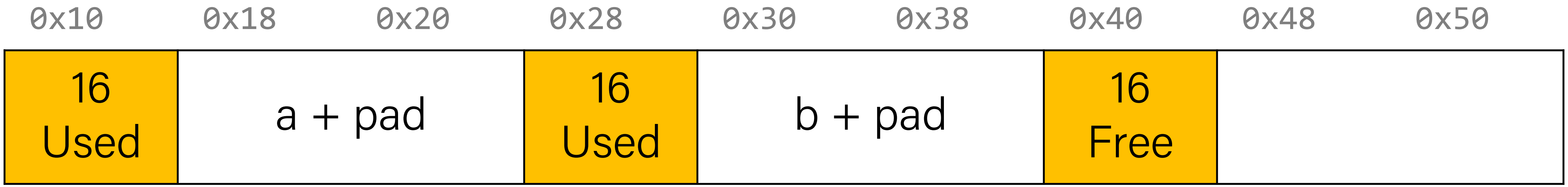
# Coalescing

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



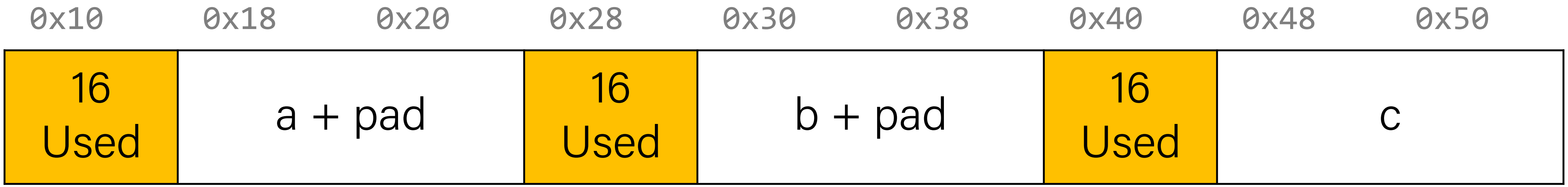
# Coalescing

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



# Coalescing

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



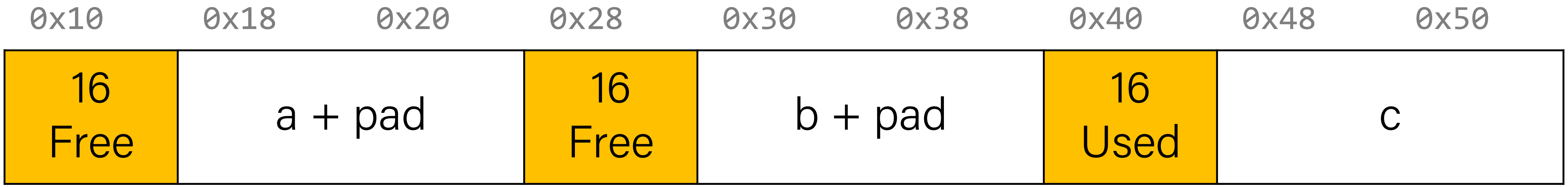
# Coalescing

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

| 0x10 | 0x18 | 0x20 | 0x28 | 0x30 | 0x38 | 0x40 | 0x48 | 0x50 |
|---|---|---|---|---|---|---|---|---|
| 16<br>Used | a + pad | | 16<br>Free | b + pad | | 16<br>Used | c | |

# Coalescing

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

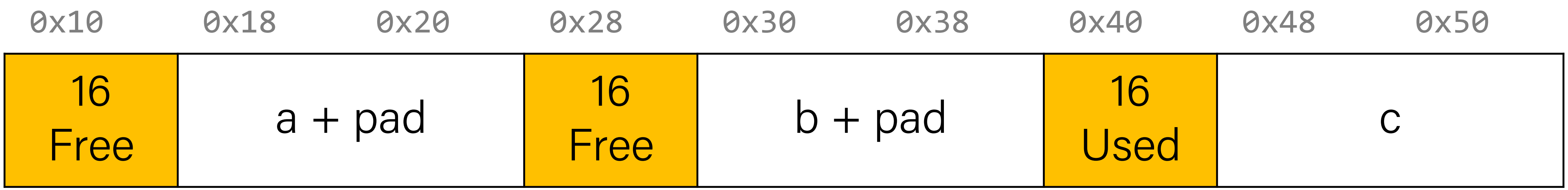


# Coalescing

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

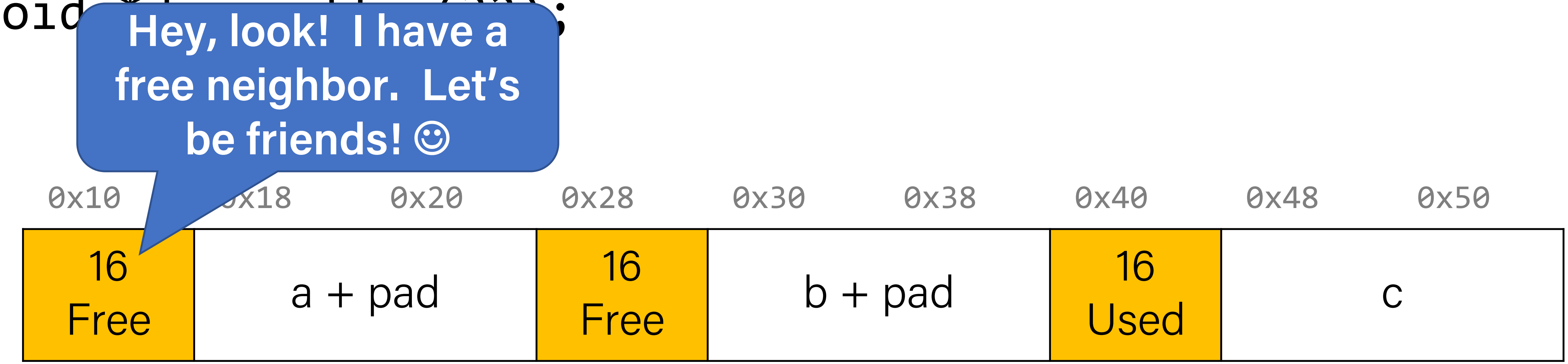
We have enough memory space, but it is fragmented into free blocks sized from earlier requests!

We'd like to be able to merge adjacent free blocks back together. How can we do this?



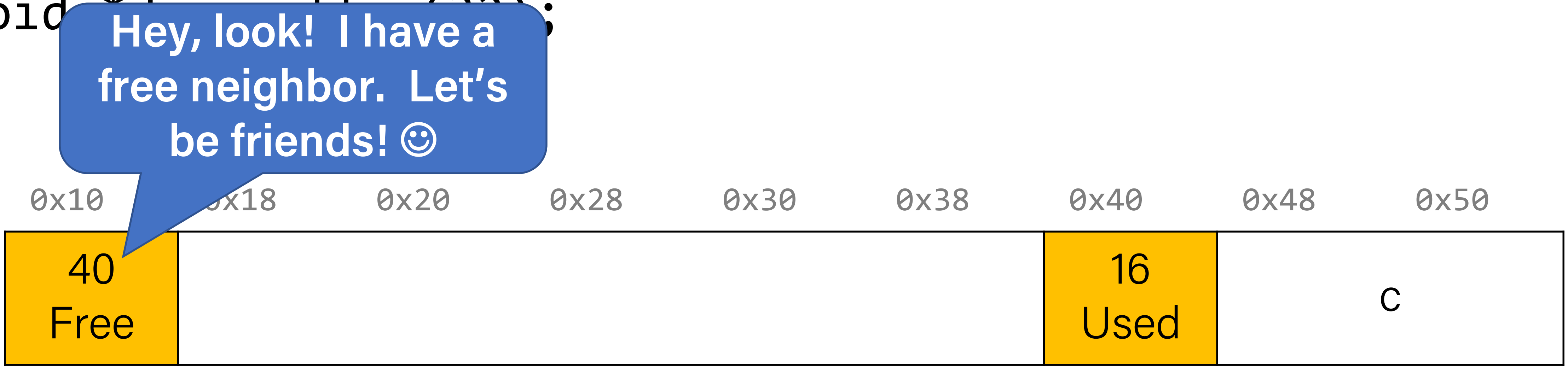
# Coalescing

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



# Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void [illegible];
```

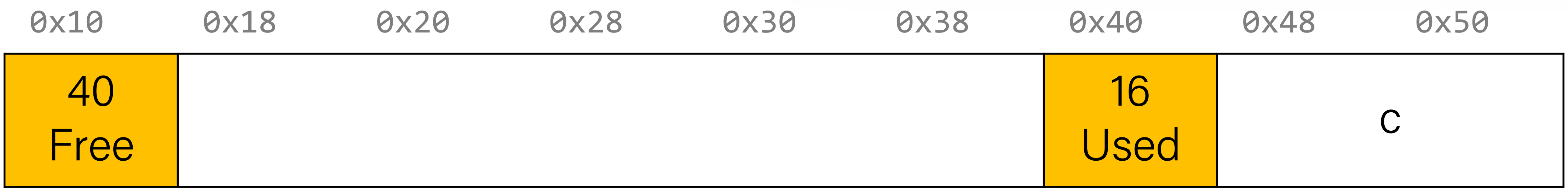


# Coalescing

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

The process of combining adjacent free blocks is called coalescing.

For your explicit heap allocator, you should coalesce if possible when a block is freed.  **You only need to coalesce the most immediate right neighbor.**



# Revisiting Our Goals

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 can coalesce on** `free()`.
3. Can we avoid always copying/moving data during `realloc`?

# Revisiting Our Goals

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 can coalesce on** `free()`.
3. Can we avoid always copying/moving data during `realloc`?

# Lecture Plan

- Method 1: Implicit Free List Allocator
- Method 2: Explicit Free List Allocator
    - Explicit Allocator
    - Coalescing
    - In-place `realloc`

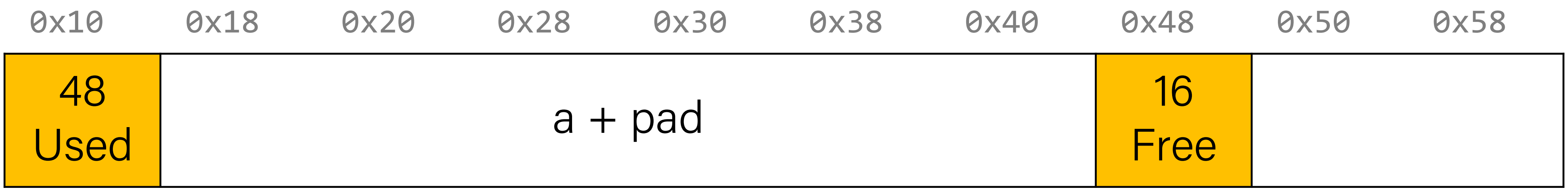
# `realloc`

- For the implicit allocator, we didn't worry too much about `realloc`. We always moved data when they requested a different amount of space.
  - Note: `realloc` can grow *or* shrink the data size.
- But sometimes we may be able to keep the data in the same place. How?
  - **Case 1:** size is growing, but we added padding to the block and can use that
  - **Case 2:** size is shrinking, so we can use the existing block
  - **Case 3:** size is growing, and current block isn't big enough, but adjacent blocks are free.

# realloc: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 48);
```

a's earlier request was too small, so we added padding. Now they are requesting a larger size we can satisfy with that padding! So `realloc` can return the same address.

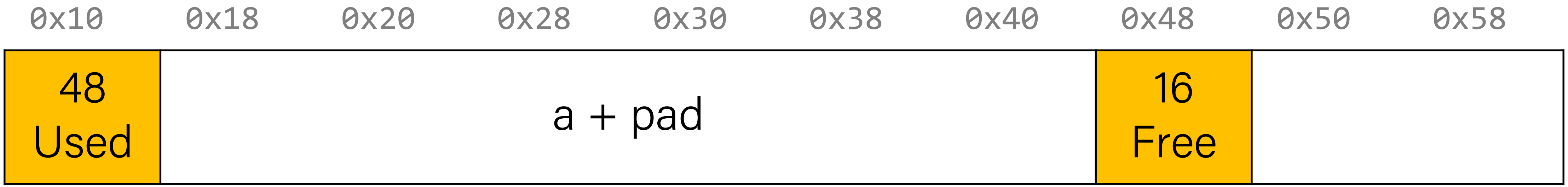


# `realloc`: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 16);
```

If a `realloc` is requesting to shrink, we can still use the same starting address.

If we can, we should try to recycle the now-freed memory into another freed block.

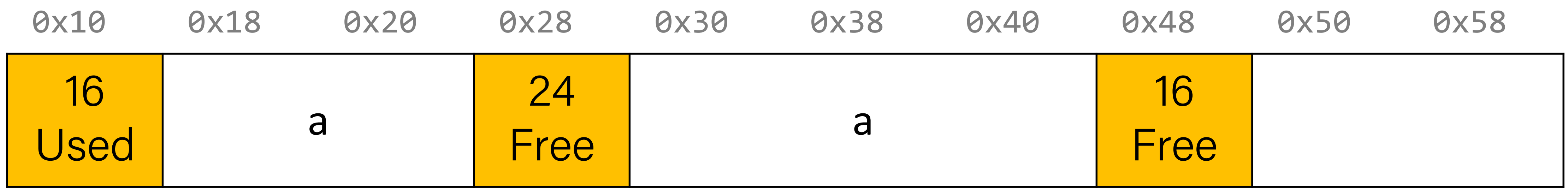


# realloc: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 16);
```

If a **realloc** is requesting to shrink, we can still use the same starting address.

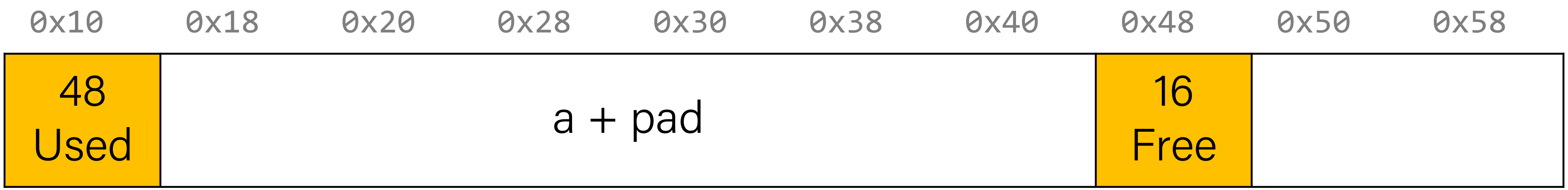
If we can, we should try to recycle the now-freed memory into another freed block.



# `realloc`: Growing In Place

```c
void *a = malloc(42);
...
void *b = realloc(a, 72);
```

Even with the padding, we don't have enough space to satisfy the larger size. But we have an adjacent neighbor that is free – let's team up!

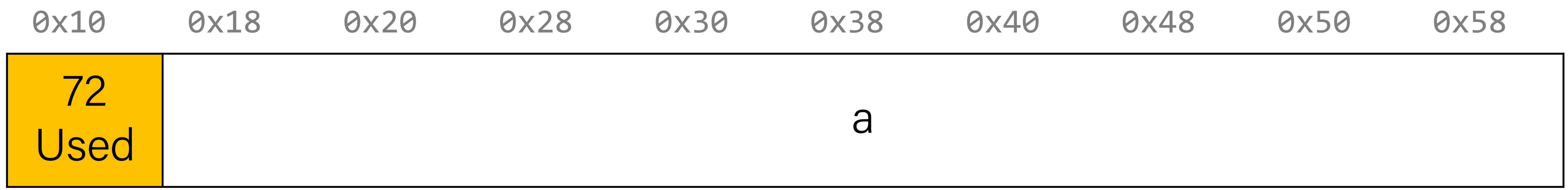


# `realloc`: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 72);
```

Even with the padding, we don't have enough space to satisfy the larger size.  But we have an adjacent neighbor that is free – let's team up!

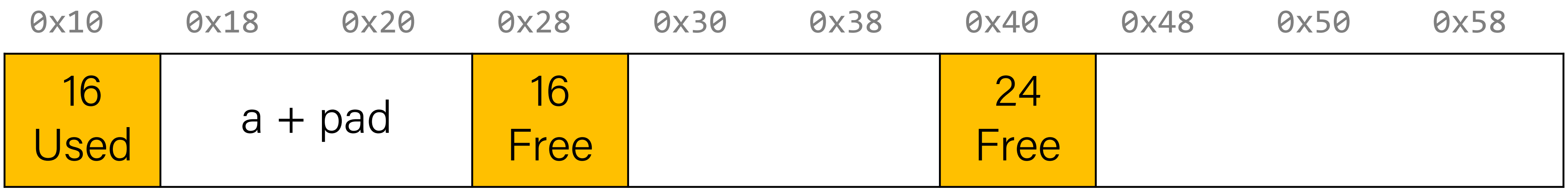
Now we can still return the same address.



# `realloc`: Growing In Place

```
void *a = malloc(8);
...
void *b = realloc(a, 72);
```

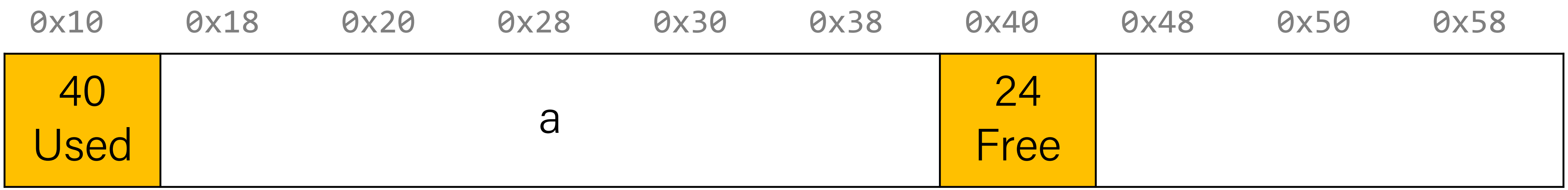
Here, you should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.



# `realloc`: Growing In Place

```c
void *a = malloc(8);
...
void *b = realloc(a, 72);
```

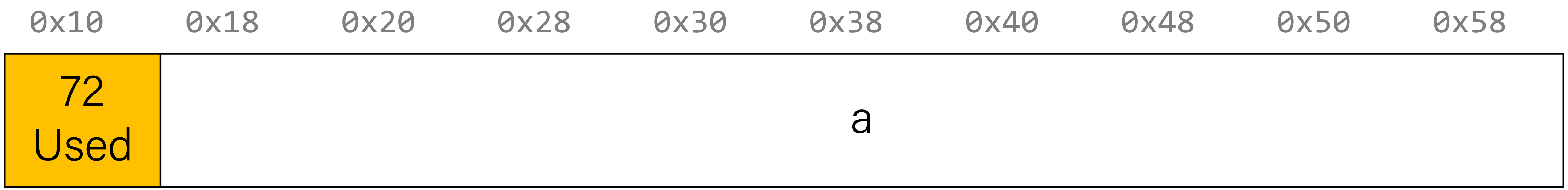
Here, you should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.



# realloc: Growing In Place

```
void *a = malloc(8);
...
void *b = realloc(a, 72);
```

Here, you should combine with your right neighbors as much as possible until we get enough space, or until we know we cannot get enough space.



# `realloc`

- For the implicit allocator, we didn't worry too much about `realloc`. We always moved data when they requested a different amount of space.
  - Note: `realloc` can grow *or* shrink the data size.
- But sometimes we may be able to keep the data in the same place. How?
  - **Case 1:** size is growing, but we added padding to the block and can use that
  - **Case 2:** size is shrinking, so we can use the existing block
  - **Case 3:** size is growing, and current block isn't big enough, but adjacent blocks are free.
- If you can't do an in-place `realloc`, then you should move the data elsewhere.

# Practice 3

- For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **explicit** free list allocator with a **first-fit** approach and **coalesce on free + realloc in-place**?

[24 byte payload, allocated for B] [16 byte payload, free] [16 byte payload, allocated for A]

```
free(B);
```

[48 byte payload, free] [16 byte payload, allocated for A]

# Practice 4

- For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **explicit** free list allocator with a **first-fit** approach and **coalesce on free + realloc in-place**?

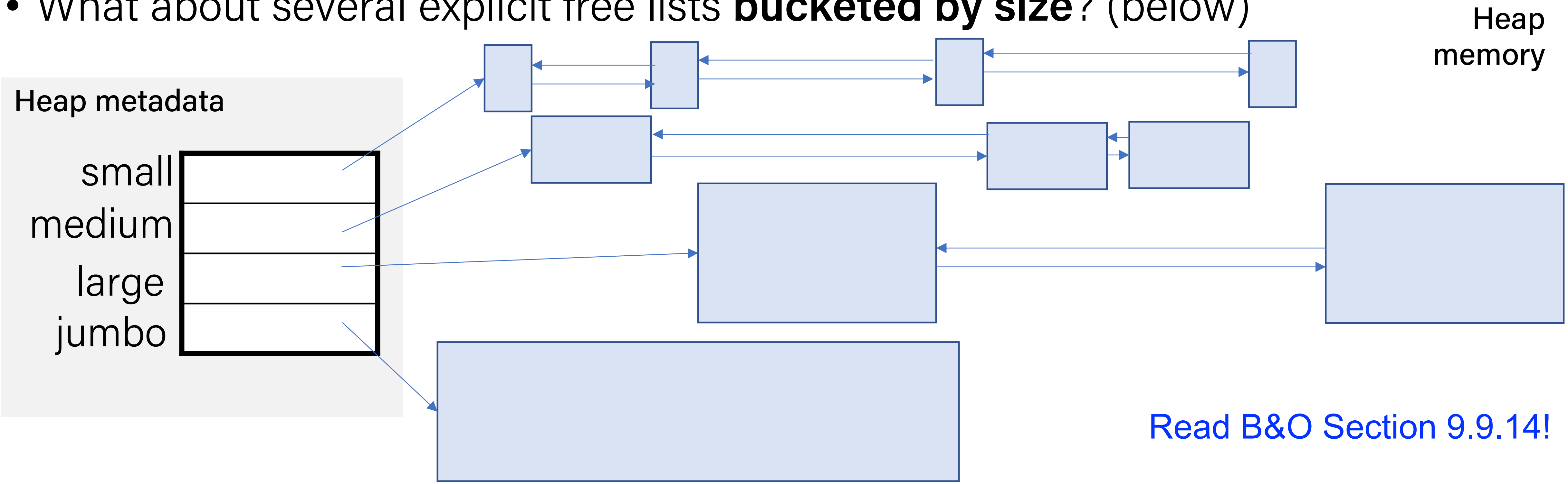
[16 byte payload, allocated for A] [32 byte payload, free] [16 byte payload, allocated for B]

```
realloc(A, 24);
```

[24 byte payload, allocated for A] [24 byte payload, free] [16 byte payload, allocated for B]

# Going beyond: Explicit list w/size buckets

- Explicit lists are much faster than implicit lists.
- However, a first-fit placement policy is still linear in total # of free blocks.
- What about an explicit free list **sorted by size** (e.g., as a tree)?
- What about several explicit free lists **bucketed by size**? (below)




Read B&O Section 9.9.14!

# Recap

- Method 1: Implicit Free List Allocator
- Method 2: Explicit Free List Allocator