CS107, Lecture 18

Heap Allocators Episode II

Reading: B&O 9.9, 9.11

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).

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).

A heap allocator cannot assume anything about the order of allocation and free requests, or even that every allocation request is accompanied by a matching free request.

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).

A heap allocator marks memory regions as **allocated** or **available**. It must remember which is which to properly provide memory to clients.

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).

A heap allocator may have options for which memory to use to fulfill an allocation request. It must decide this based on a variety of factors.

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).

A heap allocator must respond immediately to allocation requests and should not e.g. prioritize or reorder certain requests to improve performance.

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).**

Heap Allocator Goals

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

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 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.

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!**

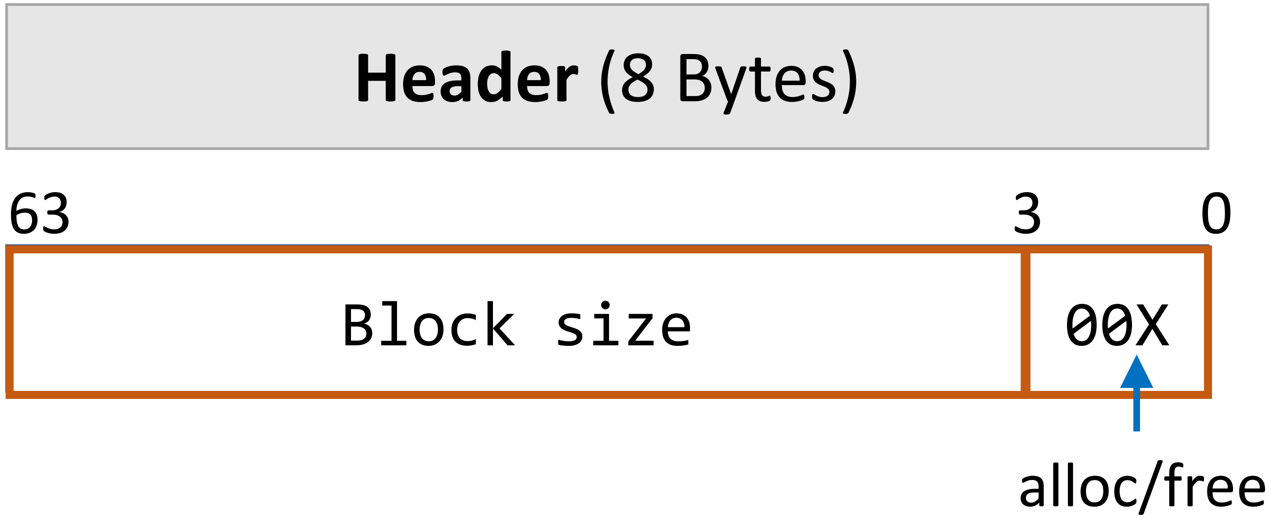
Key idea: block sizes will *always be multiples of 8*. (Why?)

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

Implicit Free List Summary

For **all blocks**,

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



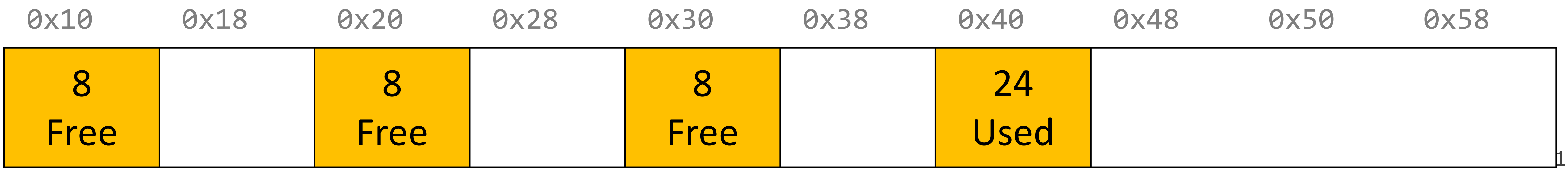
Keeping track of free blocks:

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

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).



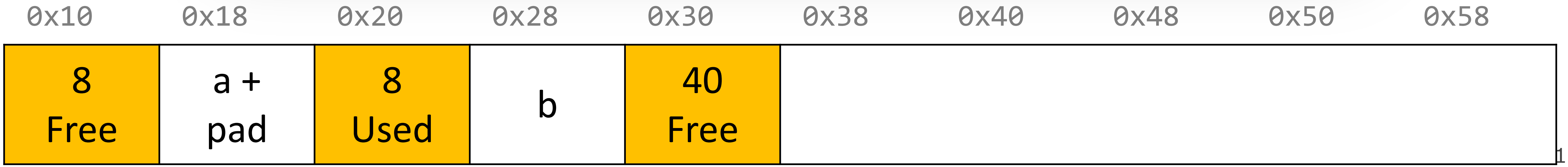
0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
8 Free		8 Free		8 Free		24 Used			

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 Free	a + pad	8 Used	b	40 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?

Checkpoint Review

Heap allocator terminology: What do the below terms mean/imply?

- Payload, Header, Free/Used(Allocated) status
- Splitting policy
- Memory utilization vs Throughput
- Bump allocator, Implicit free list Allocator
- First-fit approach, Best-fit approach
- Coalescing
- Realloc in place
- Fragmentation

Lecture Plan

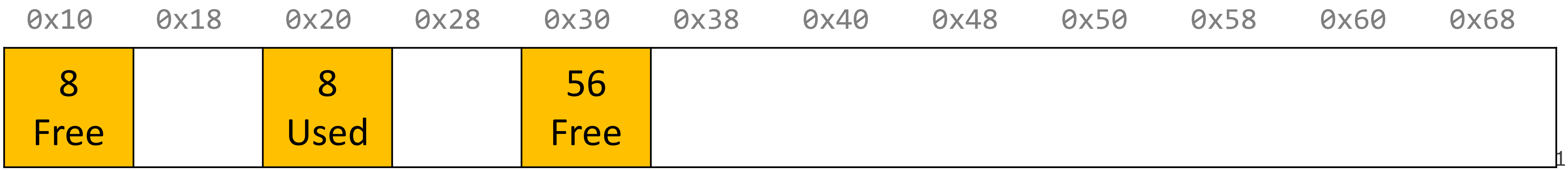
- **Recap:** heap allocators so far
- Method 0: Bump Allocator
- Method 1: Implicit Free List Allocator
- **Method 2: Explicit Free List Allocator**

Lecture Plan

- **Recap:** heap allocators so far
- Method 0: Bump Allocator
- Method 1: Implicit Free List Allocator
- **Method 2: Explicit Free List Allocator**
  - **Explicit Allocator**
  - Coalescing
  - In-place realloc

Can We Do Better?

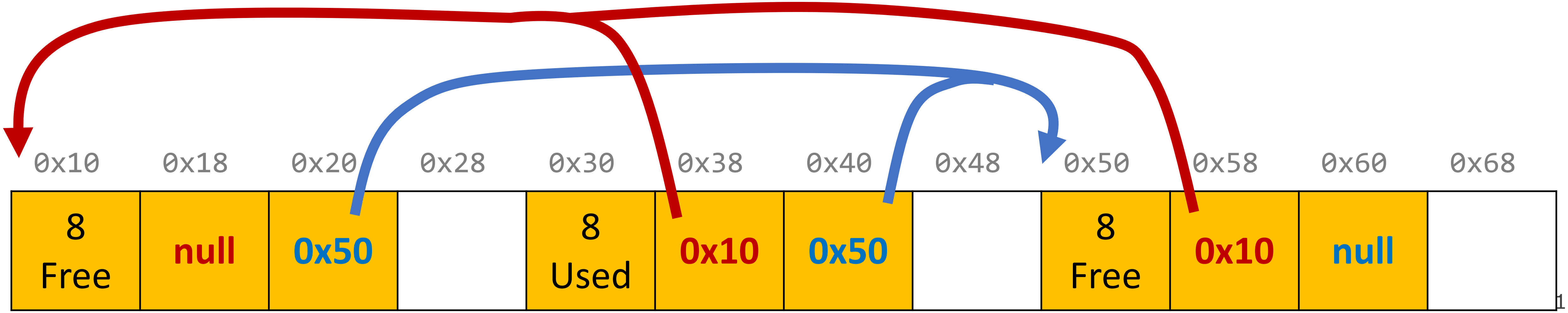
- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let’s modify each header to add a pointer to the previous free block and a pointer to the next free block.



0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60	0x68
8 Free		8 Used		56 Free							

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.

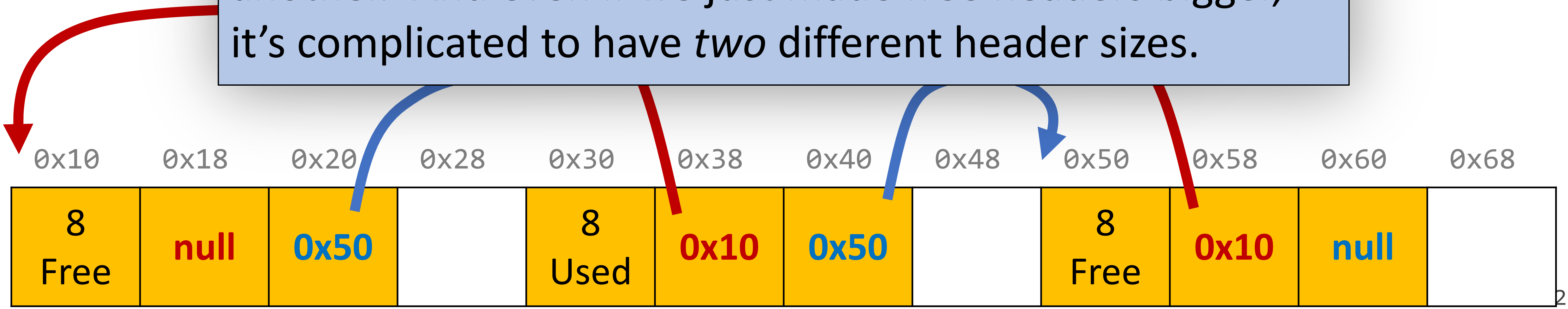


0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60	0x68
8 Free	**null**	**0x50**		8 Used	**0x10**	**0x50**		8 Free	**0x10**	**null**	

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 free block to another. And even if we just made free headers bigger, it's complicated to have *two* different header sizes.



0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60	0x68
8 Free	**null**	**0x50**		8 Used	**0x10**	**0x50**		8 Free	**0x10**	**null**	

Can We Do Better?

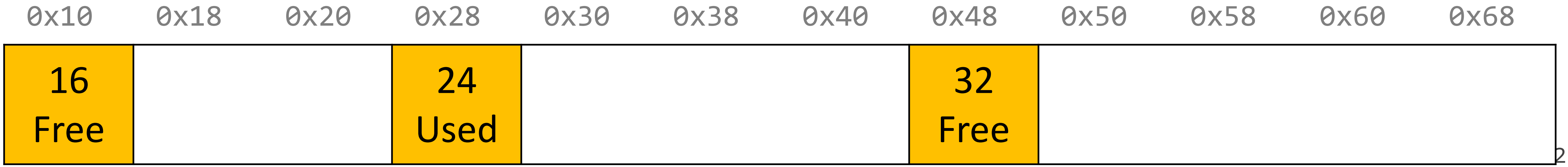
- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let’s modify each header to add a pointer to the previous free block and a pointer to the next free block. *This is inefficient / complicated.*
- **Where can we put these pointers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

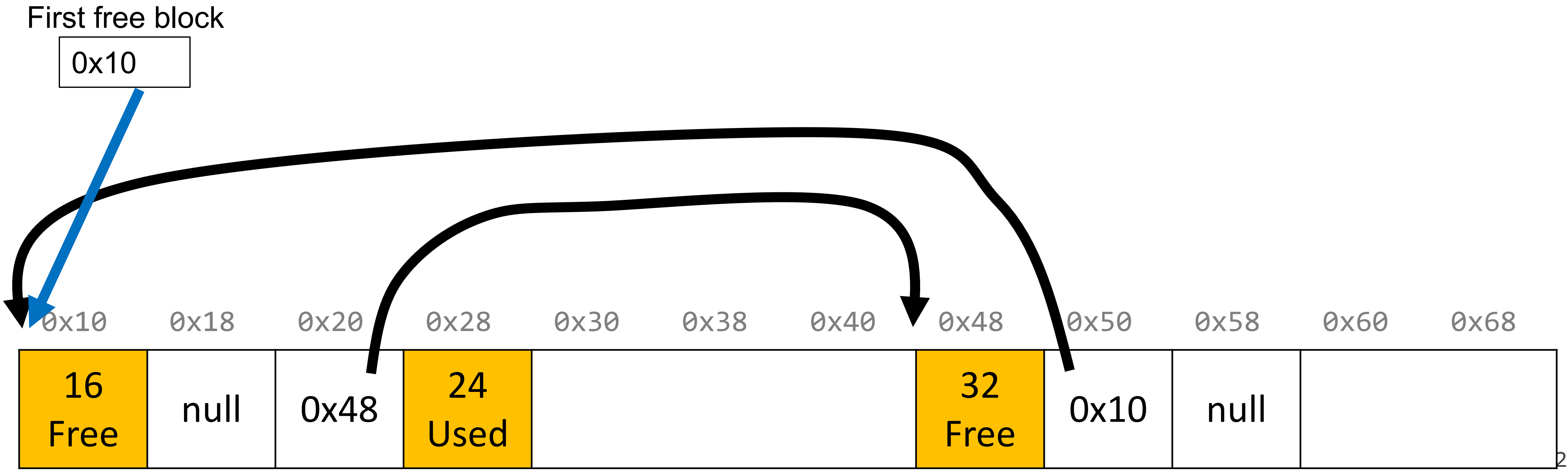
Can We Do Better?

- **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 first 16 bytes of each free block’s payload!



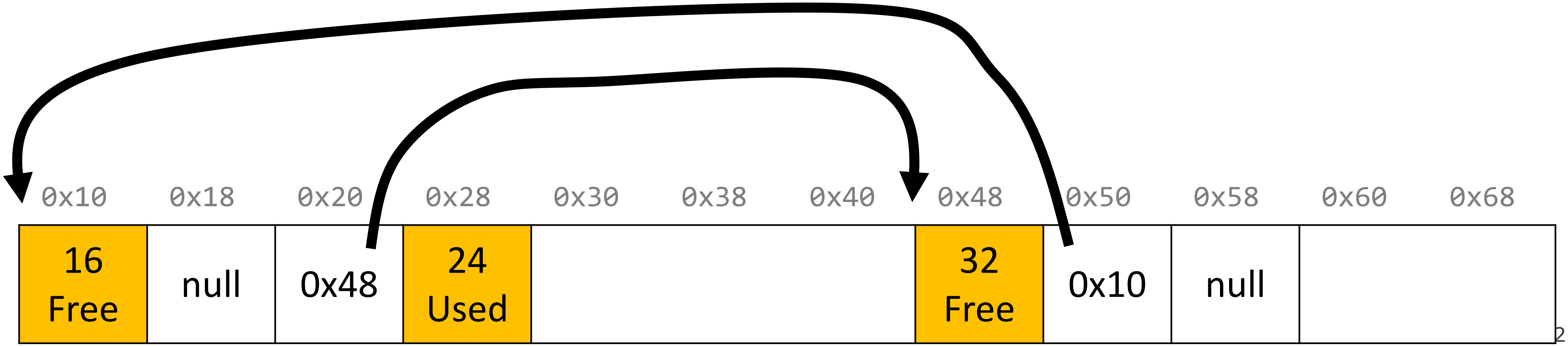
Can We Do Better?

- **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 first 16 bytes of each free block's payload!



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 first 16 bytes of each free block's payload!
- This means each payload must be big enough to store 2 pointers (16 bytes). So we must require that for every block, free and allocated. (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)
B. Last-in first-out (LIFO)/like a stack, where newly freed blocks are at the beginning of the list
C. Other (e.g., by size, etc.)

Better memory util, Linear free

Constant free (push recent block onto stack)

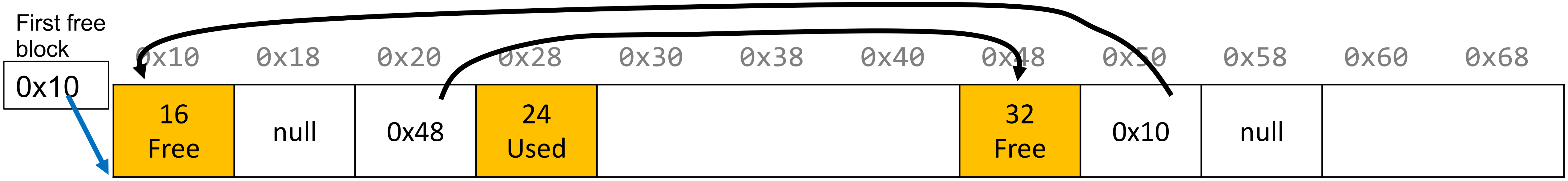
(more at end of lecture)

Explicit free list design

Up to you!

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

A. Address-order Better memory util, linear free

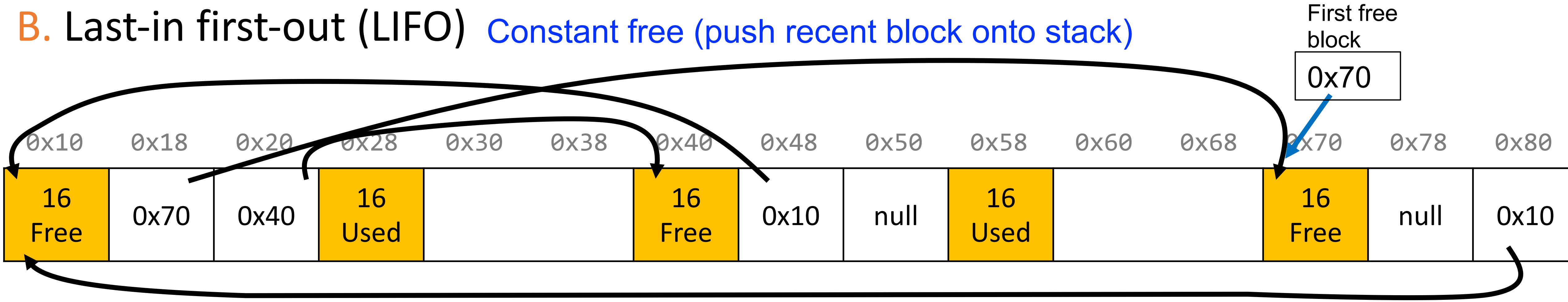


First free block

0x10

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60	0x68
16 Free	null	0x48	24 Used				32 Free	0x10	null		

B. Last-in first-out (LIFO) Constant free (push recent block onto stack)



First free block

0x70

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60	0x68	0x70	0x78	0x80
16 Free	0x70	0x40	16 Used			16 Free	0x10	null	16 Used			16 Free	null	0x10

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

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 when freeing?
   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?

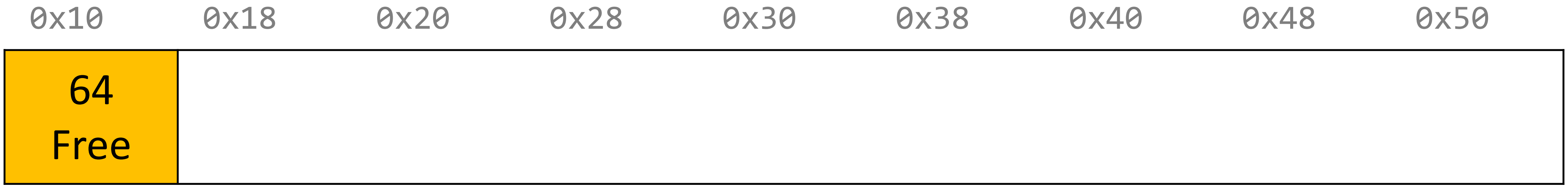
1. Can we avoid searching all blocks when freeing?
   Yes! We can use a doubly-linked list.
2. **Can we merge adjacent free blocks to keep large spaces available?**
3. Can we avoid always copying/moving data during realloc?

Lecture Plan

- **Recap:** heap allocators so far
- Method 0: Bump Allocator
- Method 1: Implicit Free List Allocator
- **Method 2: Explicit Free List Allocator**
  - Explicit Allocator
  - **Coalescing**
  - In-place realloc

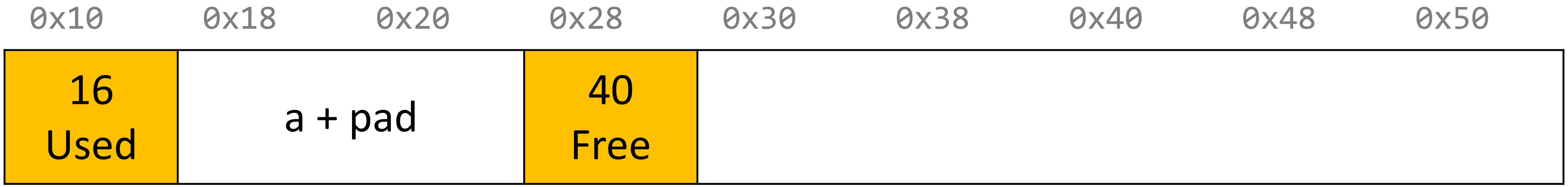
Coalescing

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



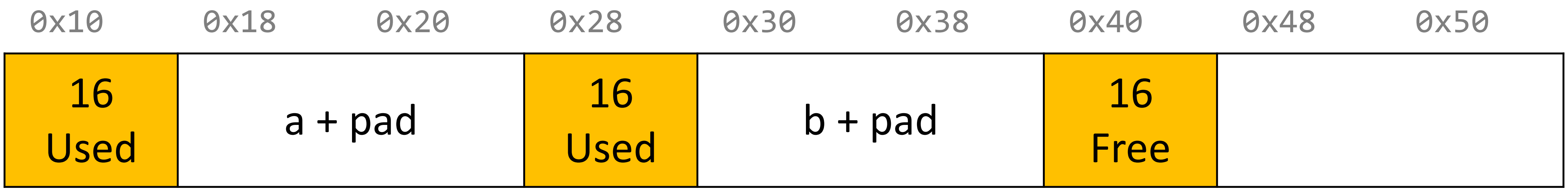
Coalescing

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



Coalescing

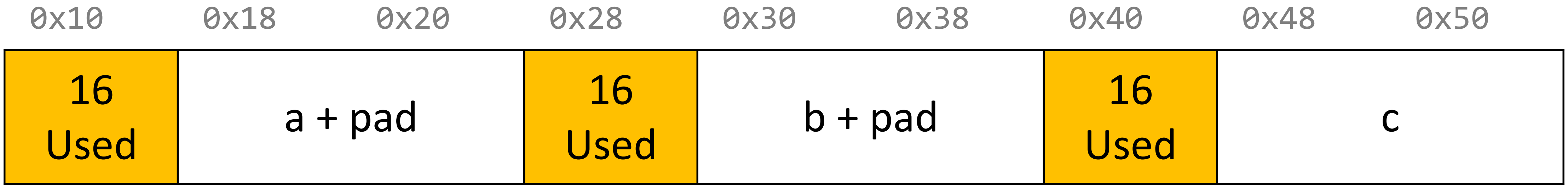
```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```



0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50
16 Used	a + pad		16 Used	b + pad		16 Free		

Coalescing

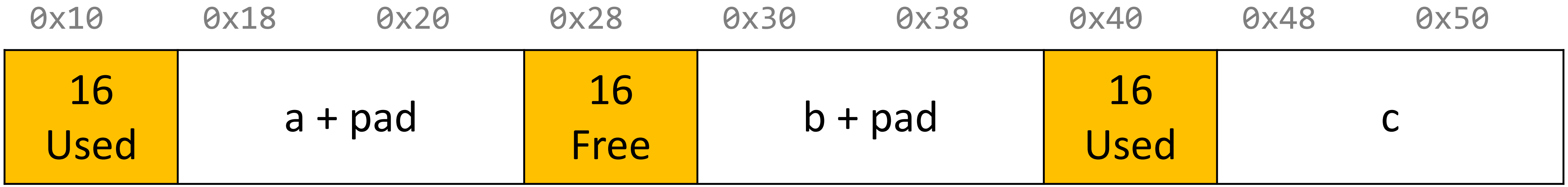
```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```



0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50
16 Used	a + pad		16 Used	b + pad		16 Used	c	

Coalescing

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

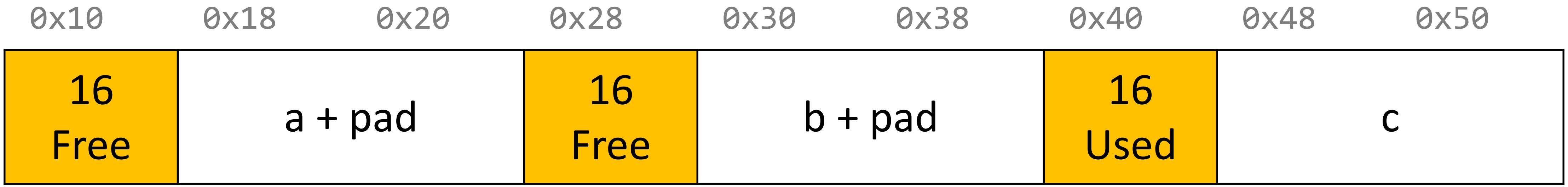


0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48 0x50

0x10	0x18–0x20	0x28	0x30–0x38	0x40	0x48–0x50
16 Used	a + pad	16 Free	b + pad	16 Used	c

Coalescing

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

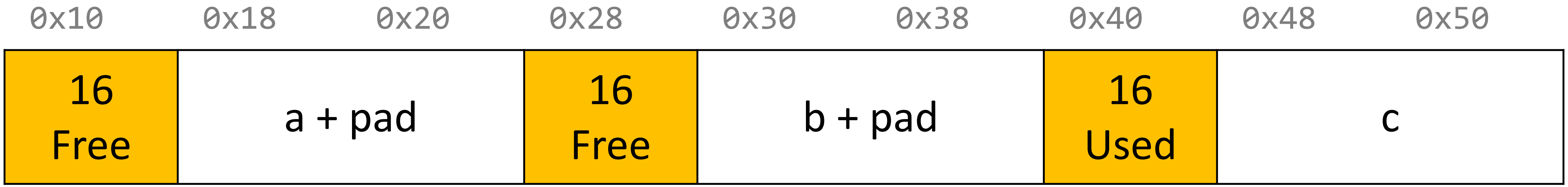


Coalescing

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

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

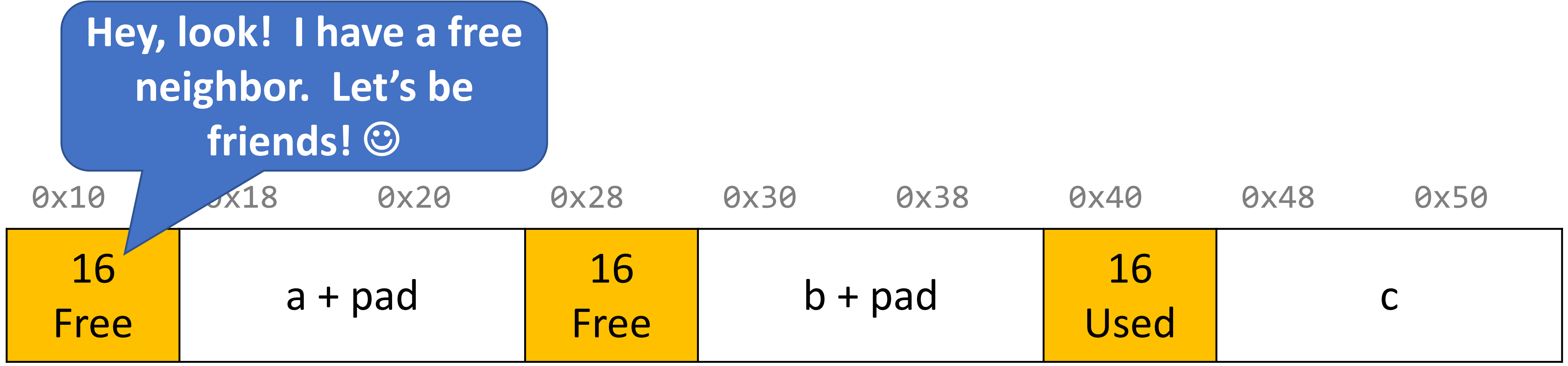
We'd like to be able to merge adjacent free blocks back together. How can we do this?



0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50
16 Free	a + pad		16 Free	b + pad		16 Used	c	

Coalescing

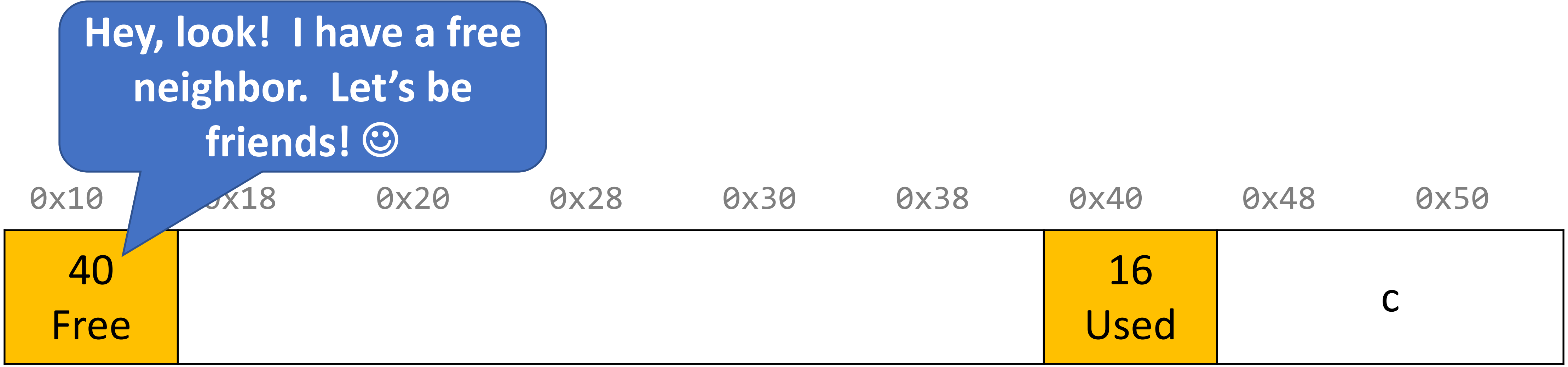
```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```



Hey, look! I have a free neighbor. Let's be friends! ☺

Coalescing

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



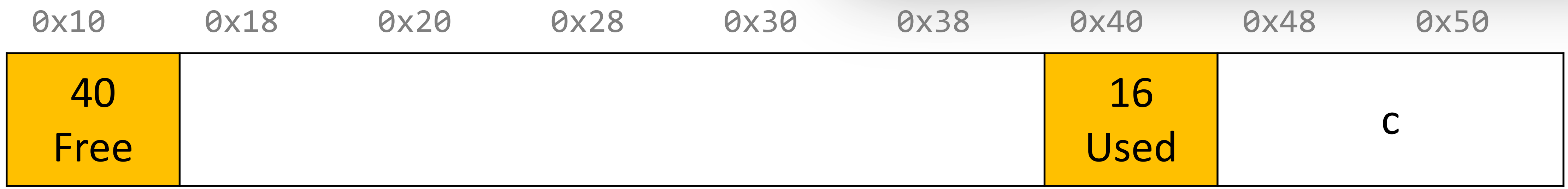
Hey, look! I have a free neighbor. Let's be friends! ☺

Coalescing

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

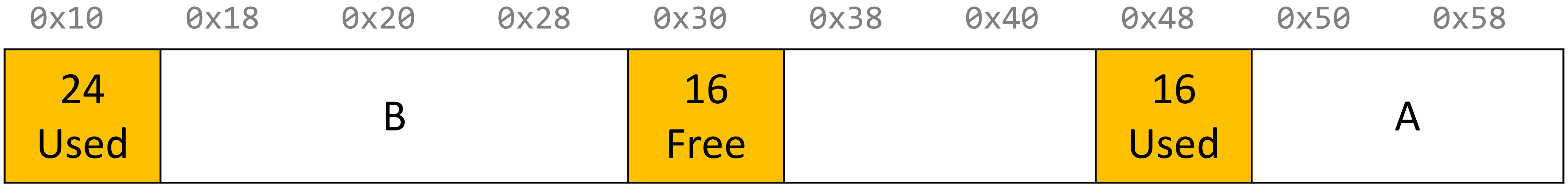
The process of combining adjacent free blocks is called *coalescing*.

For your explicit heap allocator only (not required for implicit), you should coalesce if possible when a block is freed. **You only need to coalesce the most immediate right neighbor.**



Practice 1: Explicit (coalesce)

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**?



0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48 0x50 0x58

24 Used

B

16 Free

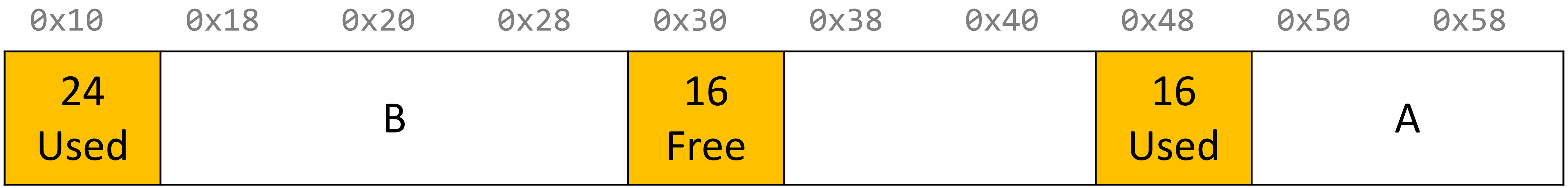
16 Used

A

```
free(b);
```

Practice 1: Explicit (coalesce)

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**?



```
free(b);
```

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

- **Recap:** heap allocators so far
- Method 0: Bump Allocator
- Method 1: Implicit Free List Allocator
- **Method 2: Explicit Free List Allocator**
    - Explicit Allocator
    - Coalescing
    - **In-place realloc**

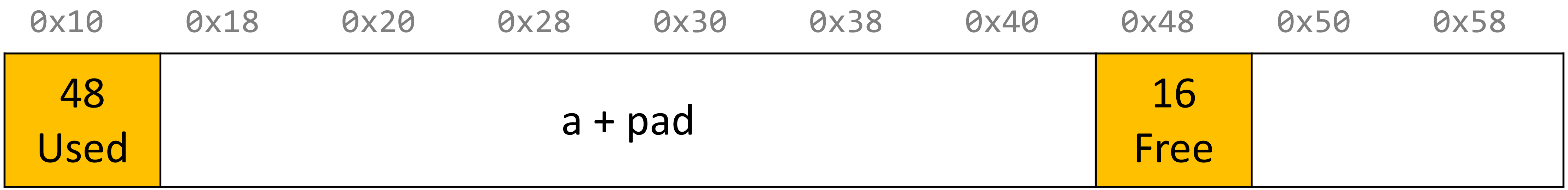
Realloc

- For the implicit free list 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.

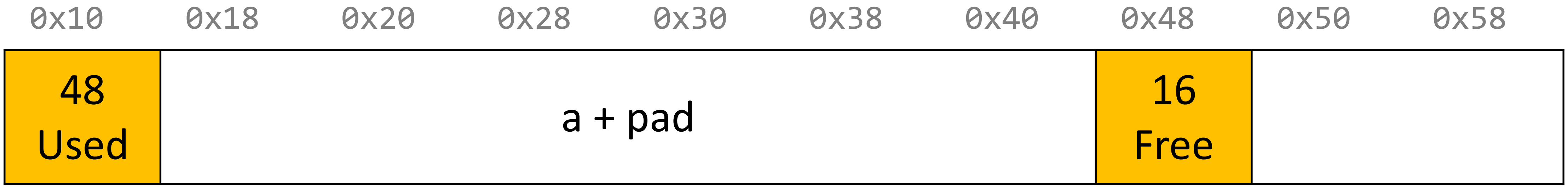


Realloc: Growing In Place

```
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.



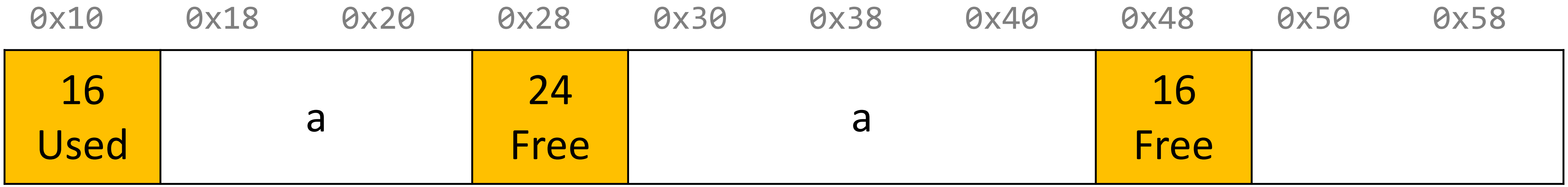
0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
48 Used	a + pad						16 Free		

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.

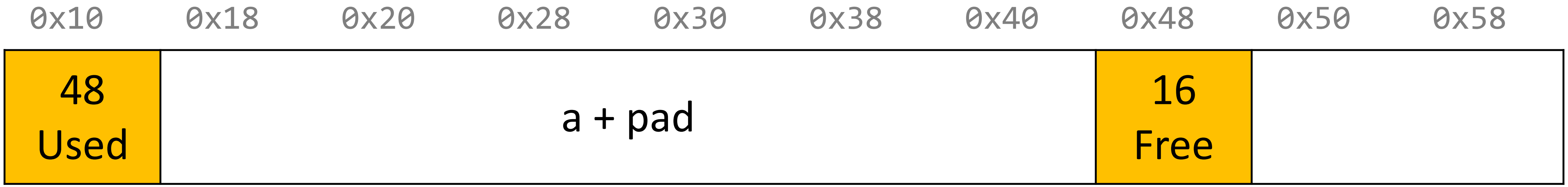


0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
16 Used	a		24 Free	a			16 Free		

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!

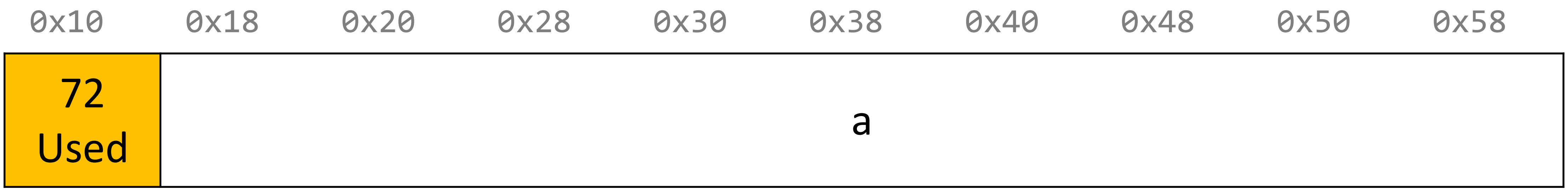


Realloc: Growing In Place

```
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!

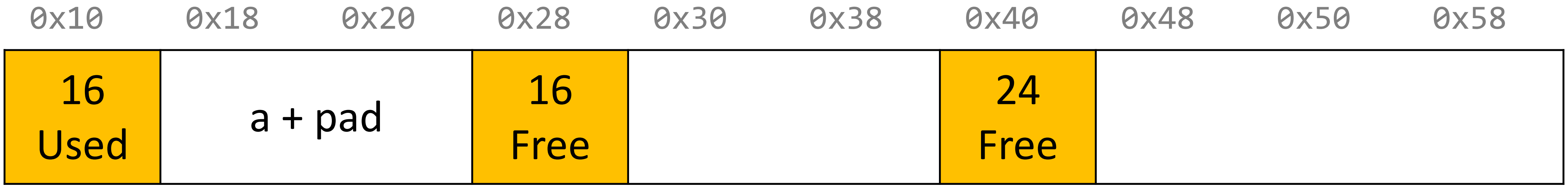
Now we can still return the same address.



Realloc: Growing In Place

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

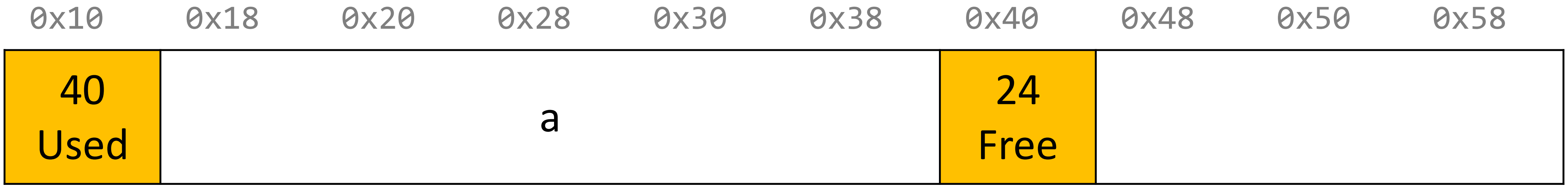
For your project (explicit only), you should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.



Realloc: Growing In Place

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

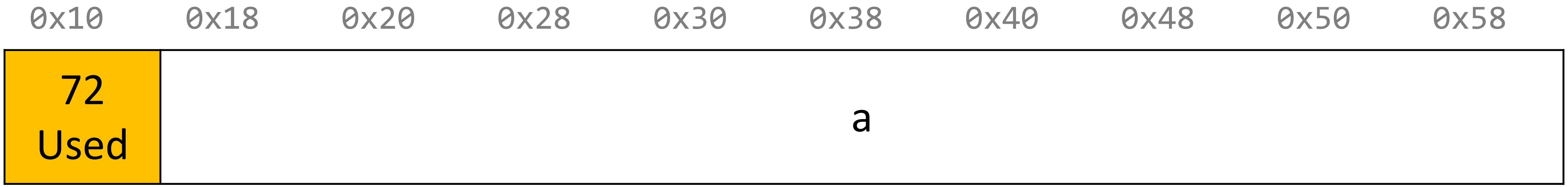
For your project (explicit only), you should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.



Realloc: Growing In Place

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

For your project (explicit only), 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.

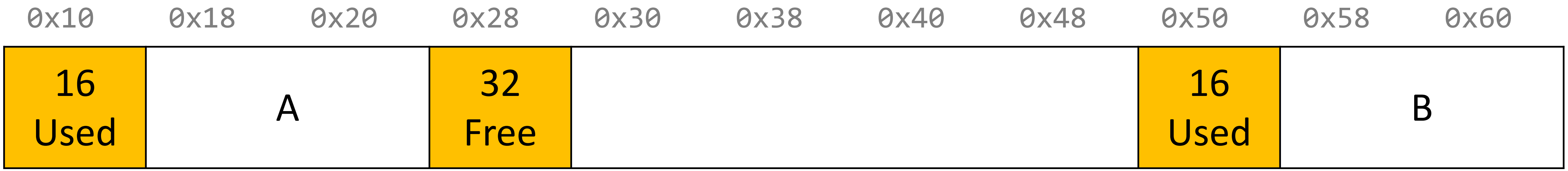


Realloc

- For the implicit free list 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 1: Explicit (realloc)

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?

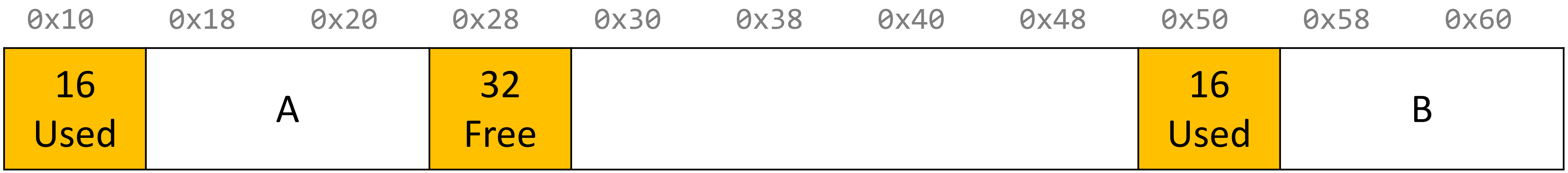


0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60
16 Used	A		32 Free					16 Used	B	

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

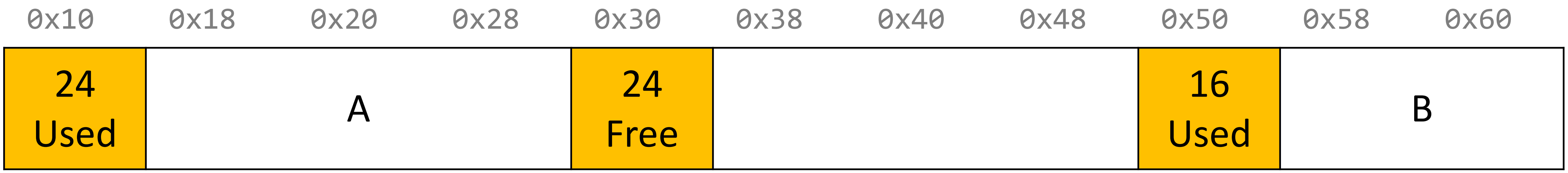
Practice 1: Explicit (realloc)

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**?



0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60
16 Used	A		32 Free					16 Used	B	

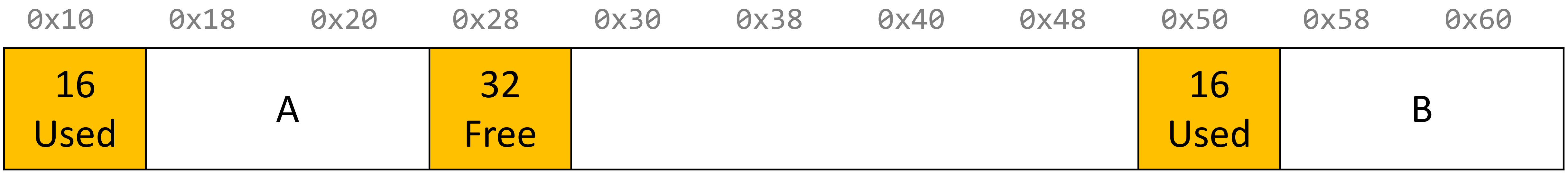
`realloc(A, 24);`



0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60
24 Used	A			24 Free				16 Used	B	

Practice 2: Explicit (realloc)

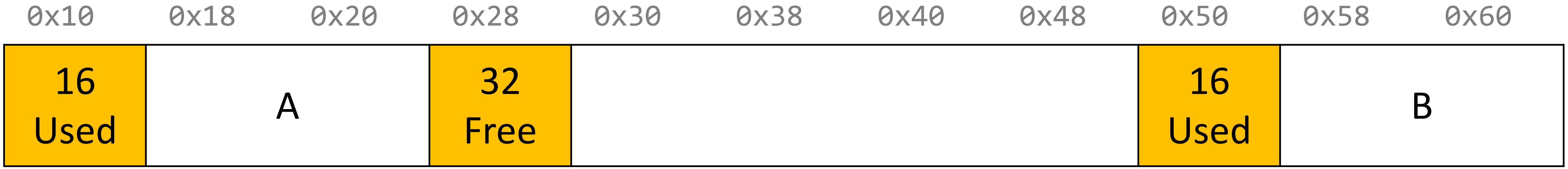
For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **explicit** free list allocator with a **first-fit** approach and **coalesce on free + realloc in-place**?



```
realloc(A, 56);
```

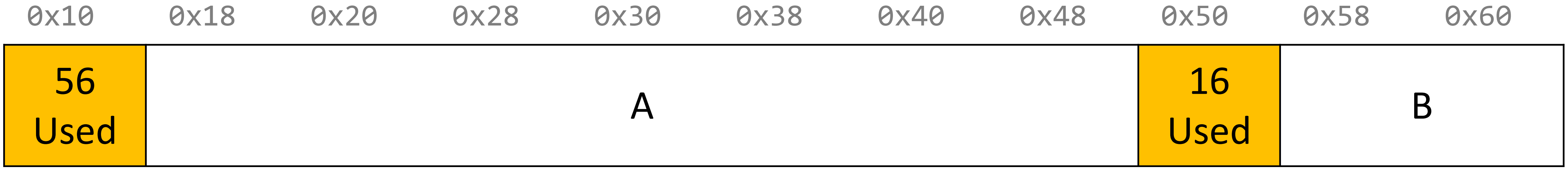
Practice 2: Explicit (realloc)

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**?



0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60
16 Used	A		32 Free					16 Used	B	

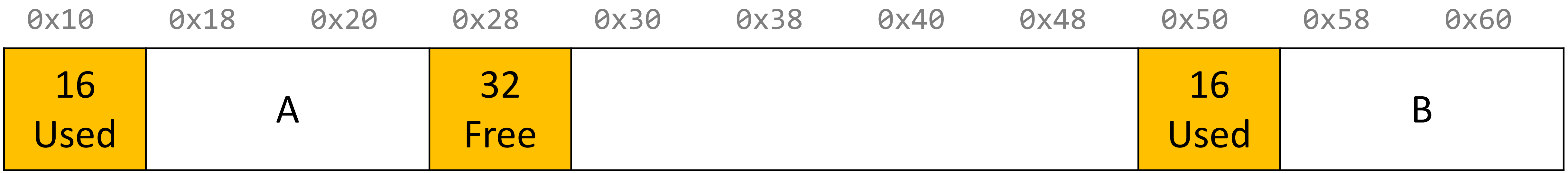
```
realloc(A, 56);
```



0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60
56 Used	A							16 Used	B	

Practice 3: Explicit (realloc)

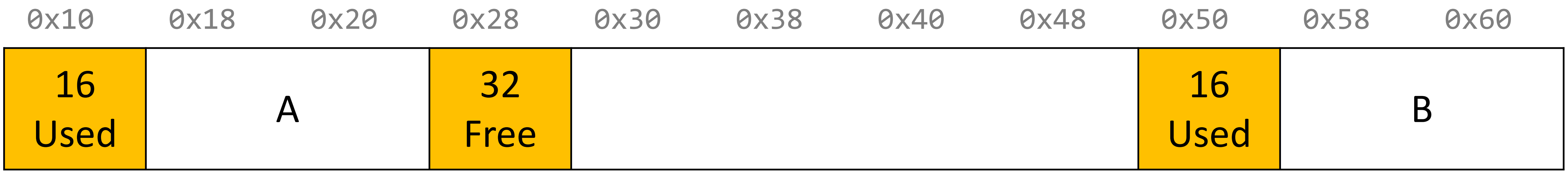
For the following heap layout, what would the heap look like after the following request is made, assuming we are using an explicit free list allocator with a **first-fit** approach and **coalesce on free + realloc in-place**?



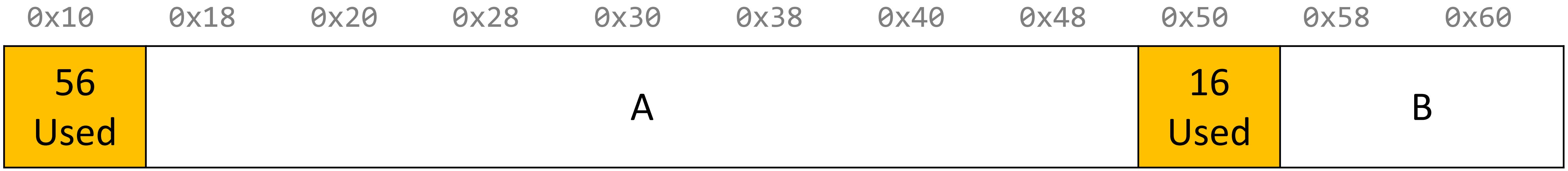
```
realloc(A, 48);
```

Practice 3: Explicit (realloc)

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**?



```
realloc(A, 48);
```



Practice 3: Explicit (realloc)

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**?

0x10	0x18	0x20	0x28	0x30	0x38
16 Used	A		32 Free		

For the explicit allocator, note that we can't have payload less than 16 bytes, so here the only option for the leftover 8 bytes is to use it as padding for the existing block.

```
realloc(A, 48);
```

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60
56 Used	A							16 Used	B	

Final Assignment: Explicit Allocator

- **Must have** headers that track block information like in implicit (size, status in-use or free) – you can copy from your implicit version
- **Must have** an explicit free list managed as a doubly-linked list, using the first 16 bytes of each free block's payload for next/prev pointers.
- **Must have** a malloc implementation that searches the explicit list of free blocks.
- **Must** coalesce a free block in free() whenever possible with its immediate right neighbor. (only required for explicit)
- **Must** do in-place realloc when possible (only required for explicit). Even if an in-place realloc is not possible, you should still absorb adjacent right free blocks as much as possible until you either can realloc in place or can no longer absorb and must realloc elsewhere.

Final Project Tips

Read B&O textbook.

- Offers some starting tips for implementing your heap allocators.
- Make sure to cite any design ideas you discover.

Honor Code/collaboration

- All non-textbook code is off-limits.
- Please do not discuss discuss code-level specifics with others.
- Your code should be designed, written, and debugged by you independently.

Helper Hours

- We will provide good debugging techniques and strategies!
- Come and discuss design tradeoffs!

Recap

- **Recap:** heap allocators so far
- Method 0: Bump Allocator
- Method 1: Implicit Free List Allocator
- Method 2: Explicit Free List Allocator

Lecture 18 takeaway: Bump, implicit free list and explicit free list are 3 heap allocator designs, each with their own tradeoffs. The implicit free list and explicit free list designs use headers to keep track of blocks. Allocators can support techniques like realloc-in-place and coalesce-on-free (both only required for your explicit allocator) to try and better handle requests.

Next time: Review session with our wonder-ca Daniel!