CS107, Lecture 17
Heap Allocators

Reading: B&O 9.9, 9.11

Lecture Plan

- **What is a heap allocator?**
- Heap allocator requirements and goals
- Method 0: Bump Allocator

Your role so far: Client

```
void *malloc(size_t size);
```

Returns a pointer to a block of heap memory of at least size bytes, or NULL if an error occurred.

```
void free(void *ptr);
```

Frees the heap-allocated block starting at the specified address.

```
void *realloc(void *ptr, size_t size);
```

Changes the size of the heap-allocated block starting at the specified address to be the new specified size. Returns the address of the new, larger allocated memory region.

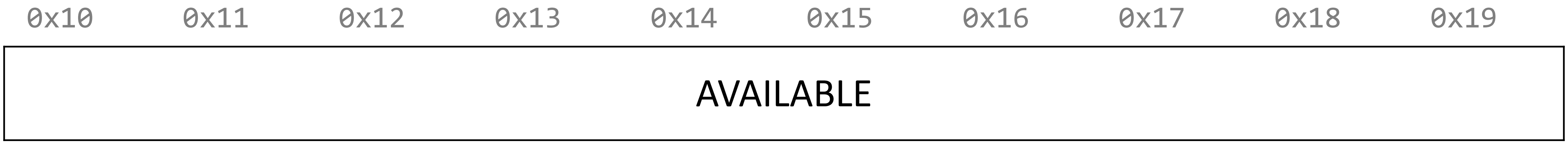
Your role now: Heap Hotel Concierge

(aka **Heap Allocator**)

http://screencrave.com/wp-content/uploads/2014/03/the-grand-budapest-hotel-anderson-image-2.jpg

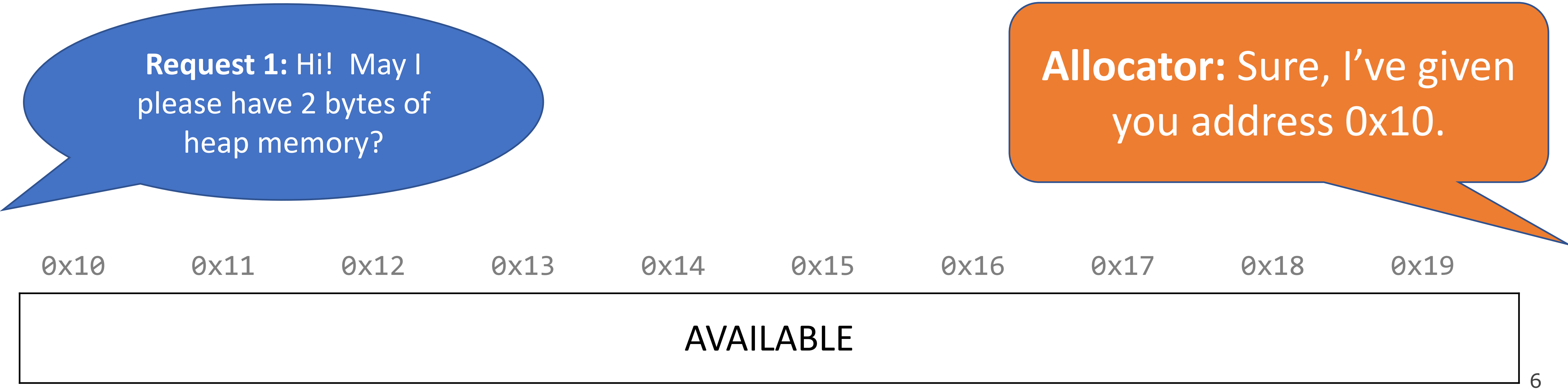
What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).



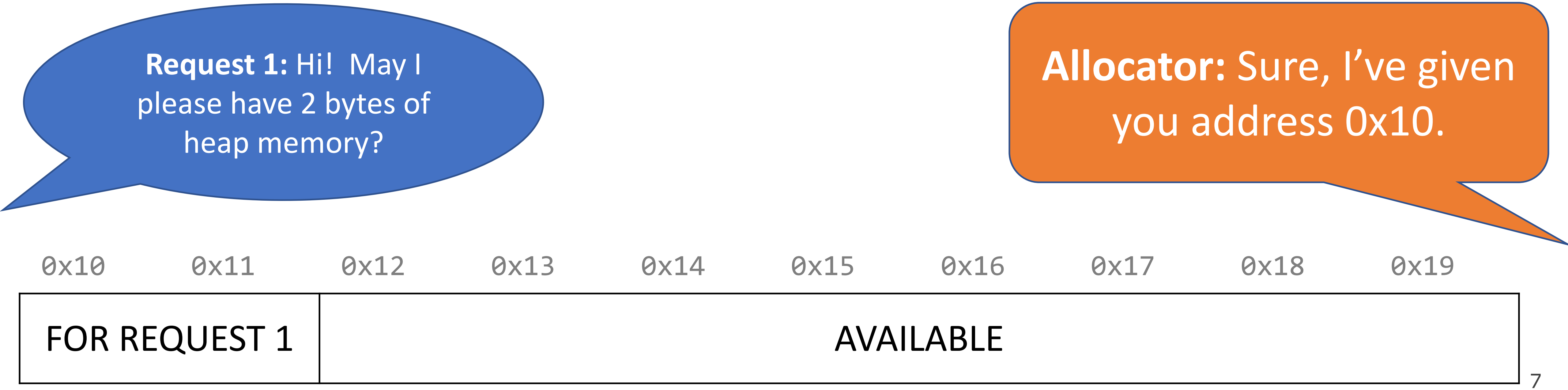
What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).
- A heap allocator must manage this memory as clients request or no longer need pieces of it.



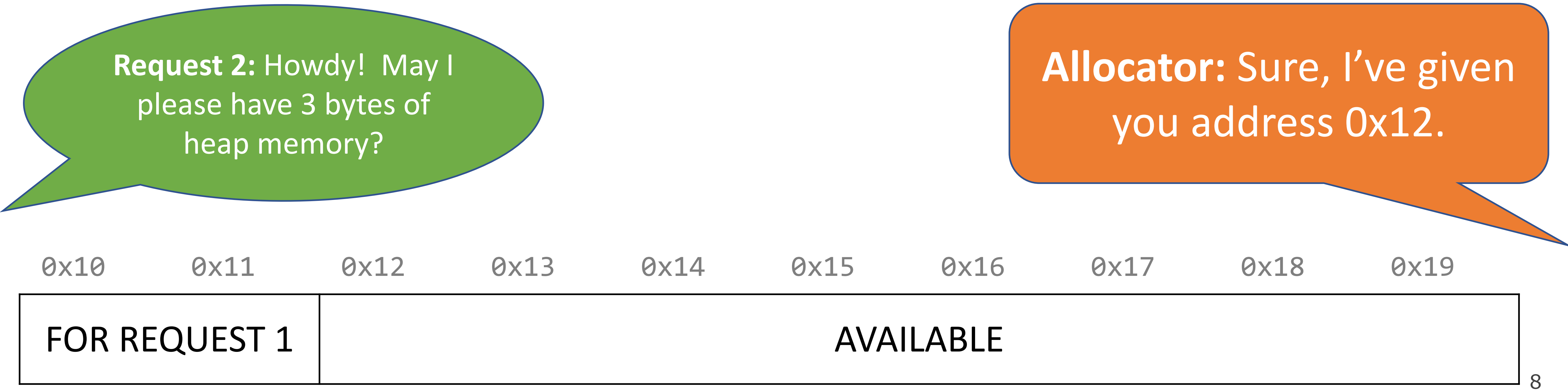
What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).
- A heap allocator must manage this memory as clients request or no longer need pieces of it.



What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).
- A heap allocator must manage this memory as clients request or no longer need pieces of it.



What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).
- A heap allocator must manage this memory as clients request or no longer need pieces of it.

Request 2: Howdy! May I please have 3 bytes of heap memory?

Allocator: Sure, I’ve given you address 0x12.

0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17 0x18 0x19

FOR REQUEST 1	FOR REQUEST 2	AVAILABLE

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).
- A heap allocator must manage this memory as clients request or no longer need pieces of it.

Request 1: I'm done with the memory I requested. Thank you!

Allocator: Thanks. Have a good day!

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19
FOR REQUEST 1		FOR REQUEST 2			AVAILABLE				

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).
- A heap allocator must manage this memory as clients request or no longer need pieces of it.

Request 1: I'm done with the memory I requested. Thank you!

Allocator: Thanks. Have a good day!

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19
AVAILABLE		FOR REQUEST 2			AVAILABLE				

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).
- A heap allocator must manage this memory as clients request or no longer need pieces of it.

Request 3: Hello there! I'd like to request 2 bytes of heap memory, please.

Allocator: Sure thing. I've given you address 0x10.

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19
AVAILABLE		FOR REQUEST 2			AVAILABLE				

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).
- A heap allocator must manage this memory as clients request or no longer need pieces of it.

Request 3: Hello there! I'd like to request 2 bytes of heap memory, please.

Allocator: Sure thing. I've given you address 0x10.

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19
FOR REQUEST 3		FOR REQUEST 2			AVAILABLE				

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).
- A heap allocator must manage this memory as clients request or no longer need pieces of it.

Request 3: Hi again! I'd like to request the region of memory at 0x10 be reallocated to 4 bytes.

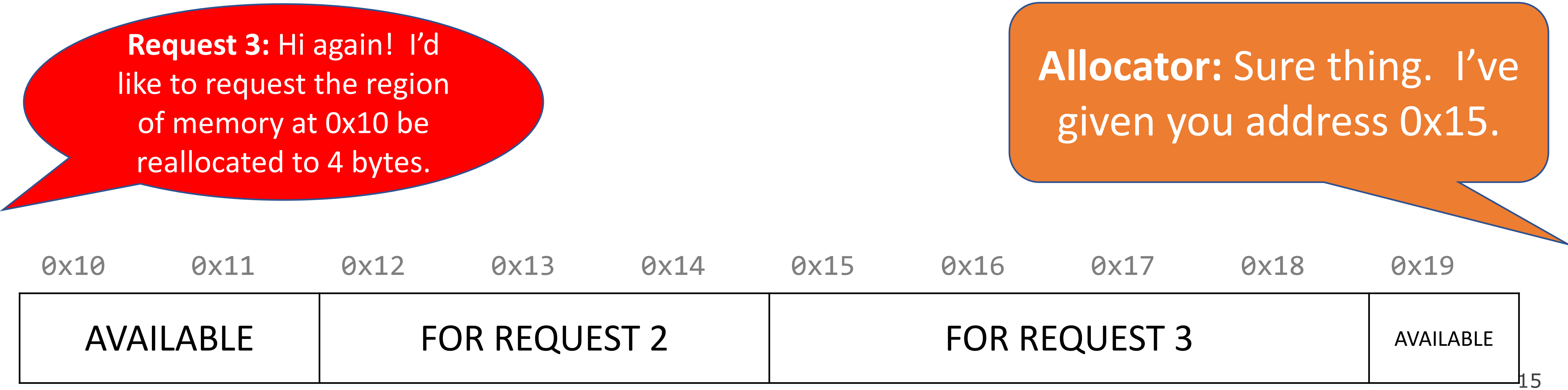
Allocator: Sure thing. I've given you address 0x15.

0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17 0x18 0x19

FOR REQUEST 3	FOR REQUEST 2	AVAILABLE

What is a heap allocator?

- A heap allocator is a set of functions that fulfills requests for heap memory.
- On initialization, a heap allocator is provided the starting address and size of a large contiguous block of memory (the heap).
- A heap allocator must manage this memory as clients request or no longer need pieces of it.



Lecture Plan

- What is a heap allocator?
- **Heap allocator requirements and goals**
- Method 0: Bump Allocator

Heap Allocator Functions

```
void *malloc(size_t size);

void free(void *ptr);

void *realloc(void *ptr, size_t size);
```

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.

Utilization

- The primary cause of poor utilization is **fragmentation**. **Fragmentation** occurs when otherwise unused memory is not available to satisfy allocation requests.
  - **External Fragmentation (this example)**: no single space is large enough to satisfy a request, even though enough aggregate free memory is available
  - **Internal Fragmentation**: space allocated for a block is larger than needed (more later).
- In general: we want the largest address used to be as low as possible.

Request 6: Hi! May I please have 4 bytes of heap memory?

Allocator: I'm sorry, I don't have a 4 byte block available…

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19
Req. 1	*Free*	Req. 2	*Free*	Req. 3	*Free*	Req. 4	*Free*	Req. 5	*Free*

Utilization

Question: Can we / should we shift these blocks down to make more space?

- YES, good idea!
- YES, but not a good idea for some reason
- NO, it can't be done!

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19
Req. 1	*Free*	Req. 2	*Free*	Req. 3	*Free*	Req. 4	*Free*	Req. 5	*Free*

↓

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19
Req. 1	Req. 2	Req. 3	Req. 4	Req. 5	*Free*				

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.

These are seemingly conflicting goals – for instance, it may take longer to better plan out heap memory use for each request. **Heap allocators must find an appropriate balance between these two goals!**

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.

Other desirable goals:

Locality ("similar" blocks allocated close in space)

Robust (handle client errors)

Ease of implementation/maintenance

Lecture Plan

- What is a heap allocator?
- Heap allocator requirements and goals
- **Method 0: Bump Allocator**

Bump Allocator

Let’s say we want to entirely prioritize throughput, and do not care about utilization at all. This means we do not care about reusing memory. How could we do this?

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.

Bump Allocator Performance

1. Utilization

😱

Never reuses memory

2. Throughput

⭐

Ultra fast, short routines

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. ☹
- We provide a bump allocator implementation as part of assign6 as a code reading exercise.

Bump Allocator

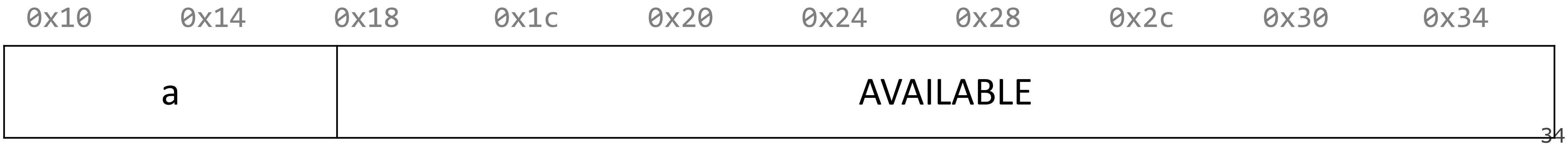
```
void *a = malloc(8);
void *b = malloc(4);
void *c = malloc(24);
free(b);
void *d = malloc(8);
```

0x10	0x14	0x18	0x1c	0x20	0x24	0x28	0x2c	0x30	0x34
AVAILABLE									

Bump Allocator

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

Variable	Value
a	0x10



Bump Allocator

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

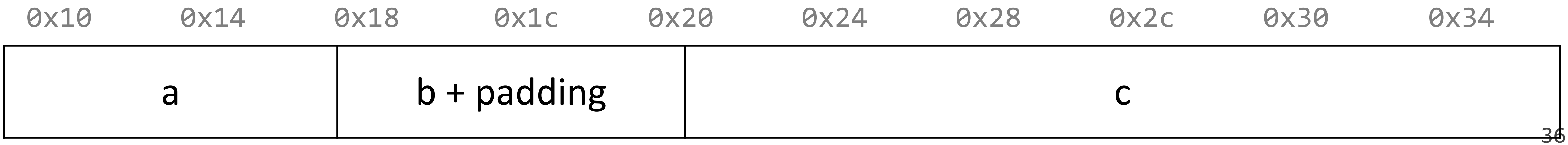
Variable	Value
a	0x10
b	0x18

0x10	0x14	0x18	0x1c	0x20	0x24	0x28	0x2c	0x30	0x34
a		b + padding		AVAILABLE					

Bump Allocator

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

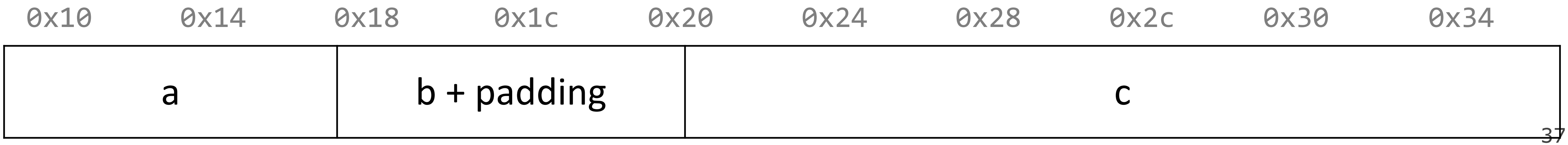
Variable	Value
a	0x10
b	0x18
c	0x20



Bump Allocator

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

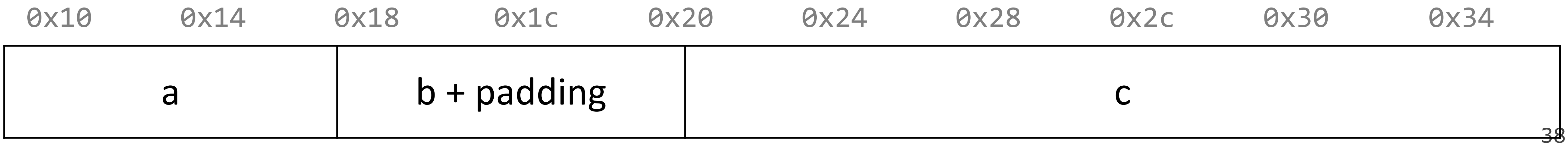
Variable	Value
a	0x10
b	0x18
c	0x20



Bump Allocator

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

Variable	Value
a	0x10
b	0x18
c	0x20
d	NULL



Summary: 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?

Recap

- What is a heap allocator?
- Heap allocator requirements and goals
- Method 0: Bump Allocator

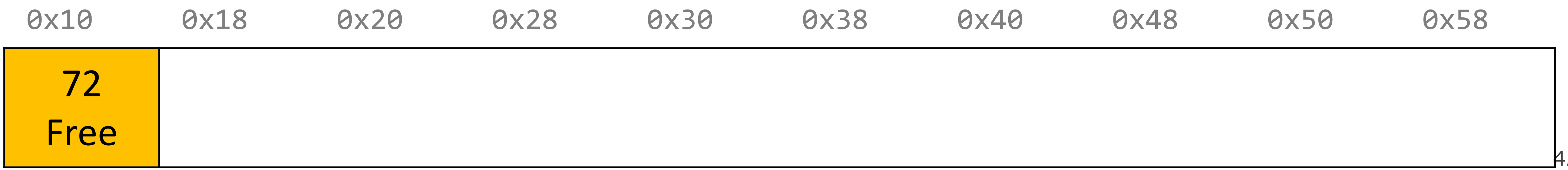
Monday takeaways:
A heap allocator is a set of functions that fulfills requests for heap memory. Seemingly-conflicting goals of maximizing throughput and memory utilization!

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

```
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);
```

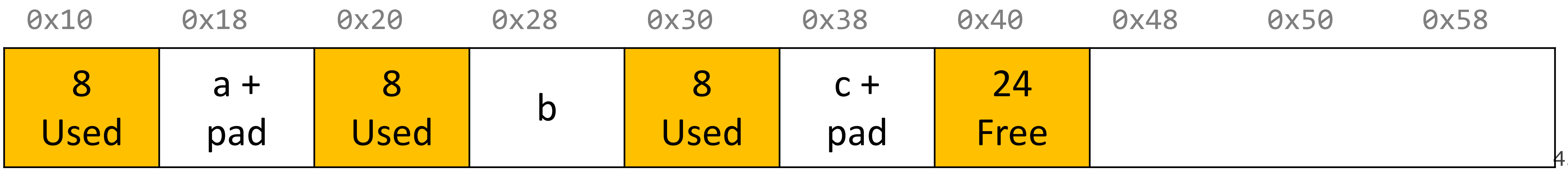
Variable	Value
a	0x18
b	0x28

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



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 Used	a + pad	8 Free	b	8 Used	c + pad	24 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
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

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

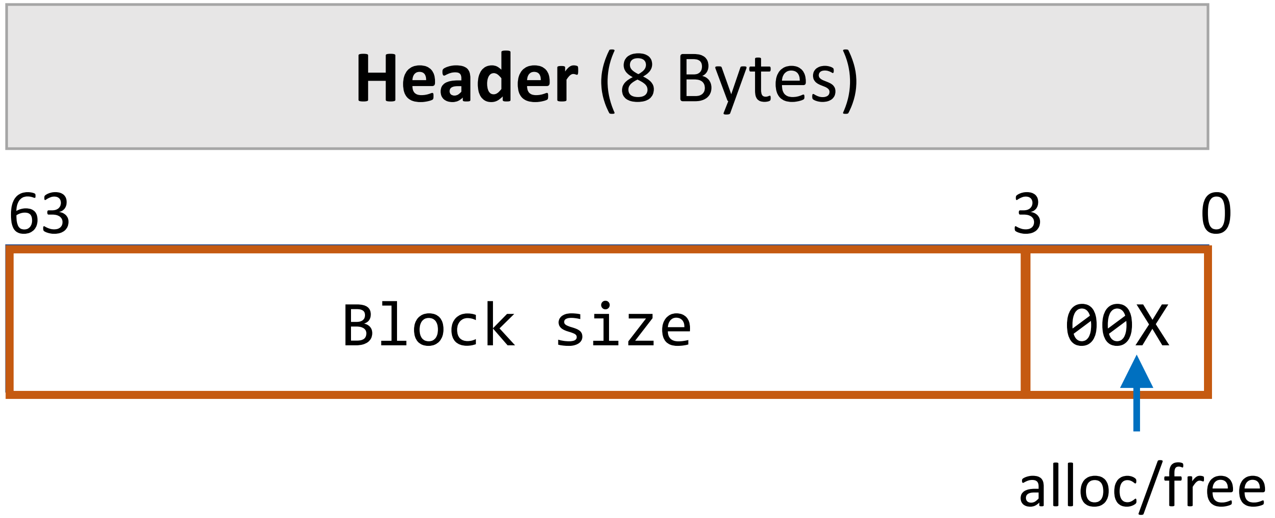
- How can we choose a free block to use for an allocation request?
  - **First fit:** search the list from beginning each time and choose first free block that fits.
  - **Next fit:** instead of starting at the beginning, continue where previous search left off.
  - **Best fit:** examine every free block 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 ☺

Up to you!

Implicit free list header design

Should we store the **block size** as
(A) payload size, or
(B) header + payload size?

Up to you! Your decision affects how you traverse the list (be careful of off-by-one)

Up to you!

Splitting Policy

```
...
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			

Up to you!

Splitting Policy

```
...
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	16 Used	e		???

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		

Up to you!

Splitting Policy

```
...
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 Free	a + pad	8 Used	d	8 Used	c + pad	16 Used	e		0 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!**

Practice 1: Implicit (first-fit)

For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **implicit** free list allocator with a **first-fit** approach?


```
void *b = malloc(8);
```

Practice 1: Implicit (first-fit)

For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **implicit** free list allocator with a **first-fit** approach?

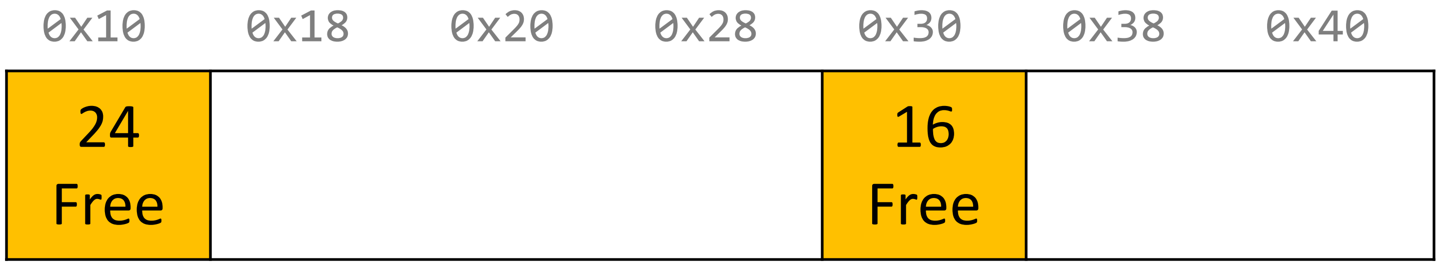
0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
8 Free		32 Free						8 Used	A

```
void *b = malloc(8);
```

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
8 Used	B	32 Free						8 Used	A

Practice 2: Implicit (first-fit)

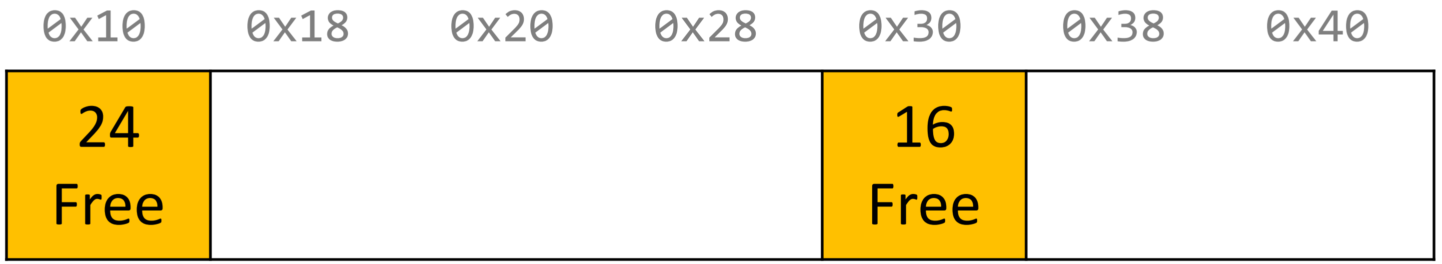
For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **implicit** free list allocator with a **first-fit** approach?



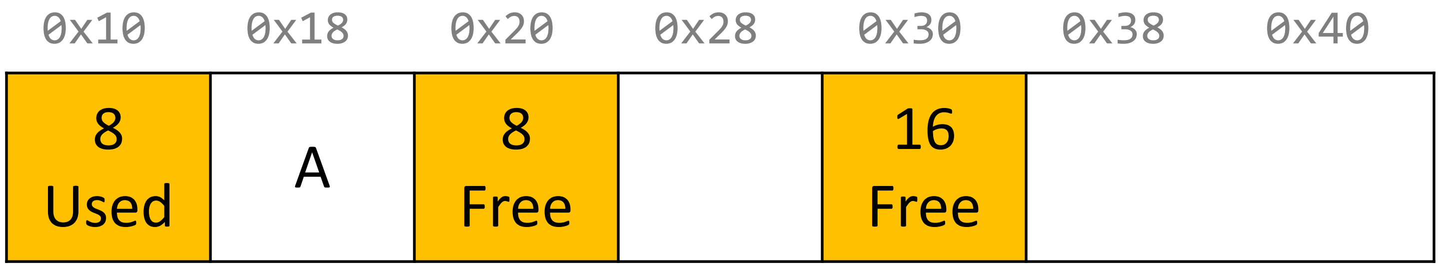

```
void *a = malloc(8);
```

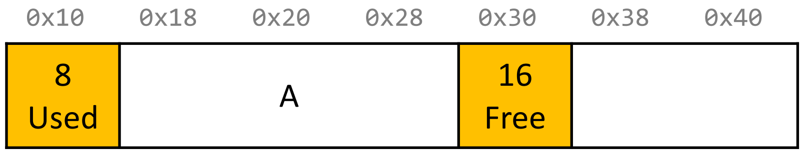
Practice 2: Implicit (first-fit)

For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **implicit** free list allocator with a **first-fit** approach?

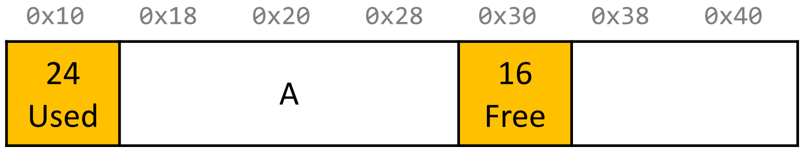



```
void *a = malloc(8);
```

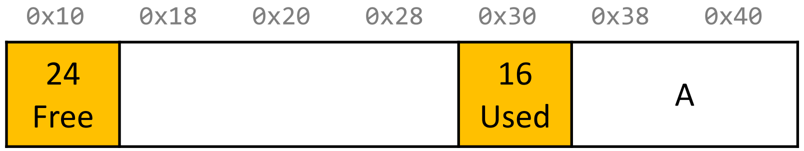




✗ *Space not tracked correctly*



✗ *We can save extra for later*



✗ *First fit chooses first available*

Practice 3: Implicit (best-fit)

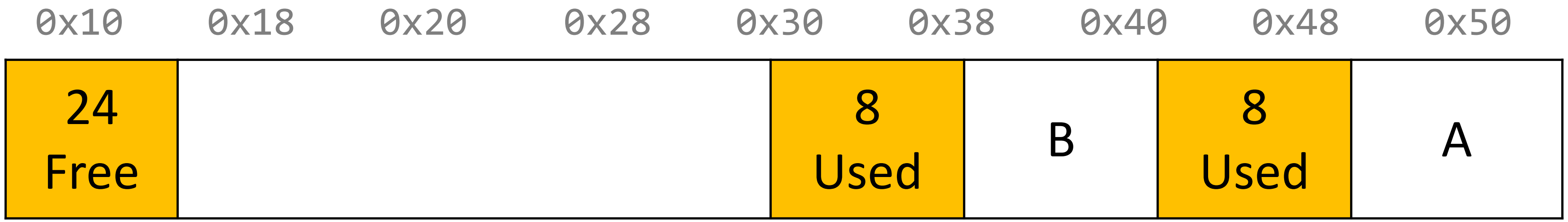
For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **implicit** free list allocator with a **best-fit** approach?


```
void *b = malloc(8);
```

Practice 3: Implicit (best-fit)

For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **implicit** free list allocator with a **best-fit** approach?


```
void *b = malloc(8);
```



Final Assignment: Implicit Allocator

- **Must have** headers that track block information (size, status in-use or free) – you must use the 8 byte header size, storing the status using the free bits (this is larger than the 4 byte headers specified in the book, as this makes it easier to satisfy the alignment constraint and store information).
- **Must have** free blocks that are recycled and reused for subsequent malloc requests if possible
- **Must have** a malloc implementation that searches the heap for free blocks via an implicit list (i.e. traverses block-by-block).

- **Does not need to** have coalescing of free blocks
- **Does not need to** support in-place realloc

(Note: these could be part of an implicit allocator, it's just not a requirement for this assignment)

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);
```

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

In-Place Realloc

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

Variable	Value
a	0x18

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

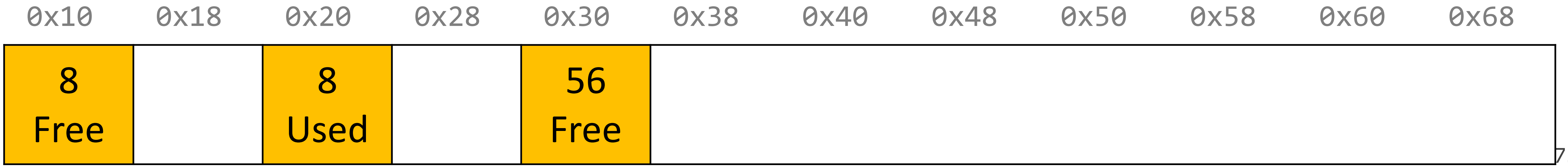
- **Recap:** heap allocators so far
- Method 0: Bump Allocator
- Method 1: Implicit Free List Allocator
- **Method 2: Explicit Free List Allocator**

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

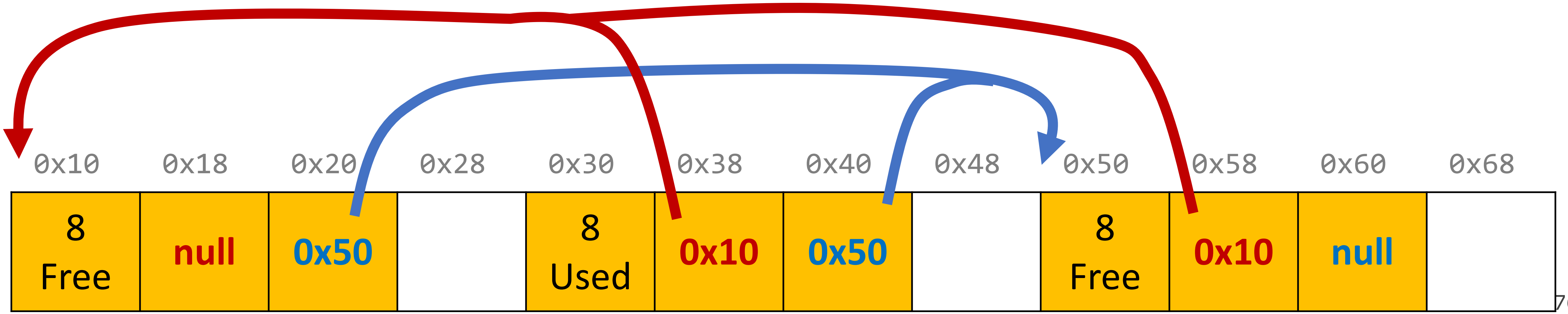
Can We Do Better?

- 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.



Can We Do Better?

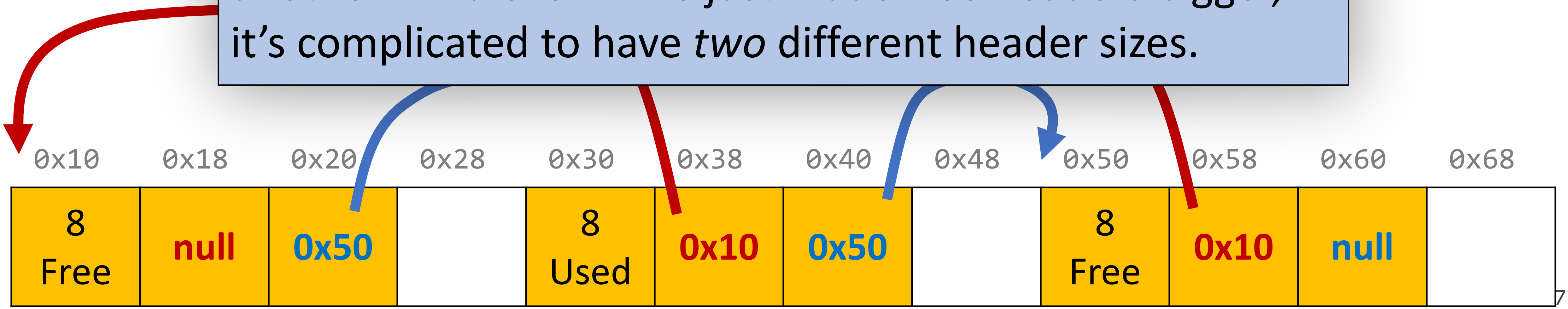
- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the **previous** free block and a pointer to the **next** free block.



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 free 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?

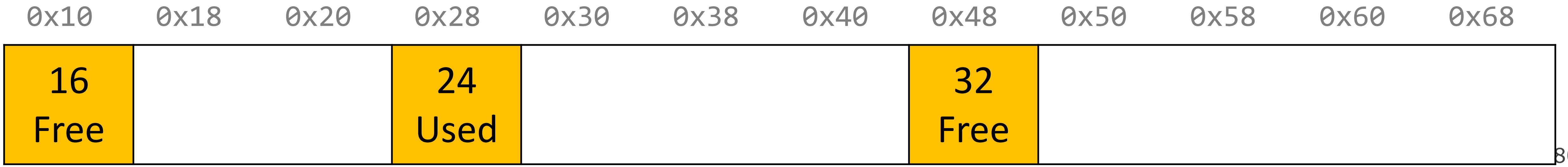
- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the previous free block and a pointer to the next free block. *This is inefficient / complicated.*
- **Where can we put these pointers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

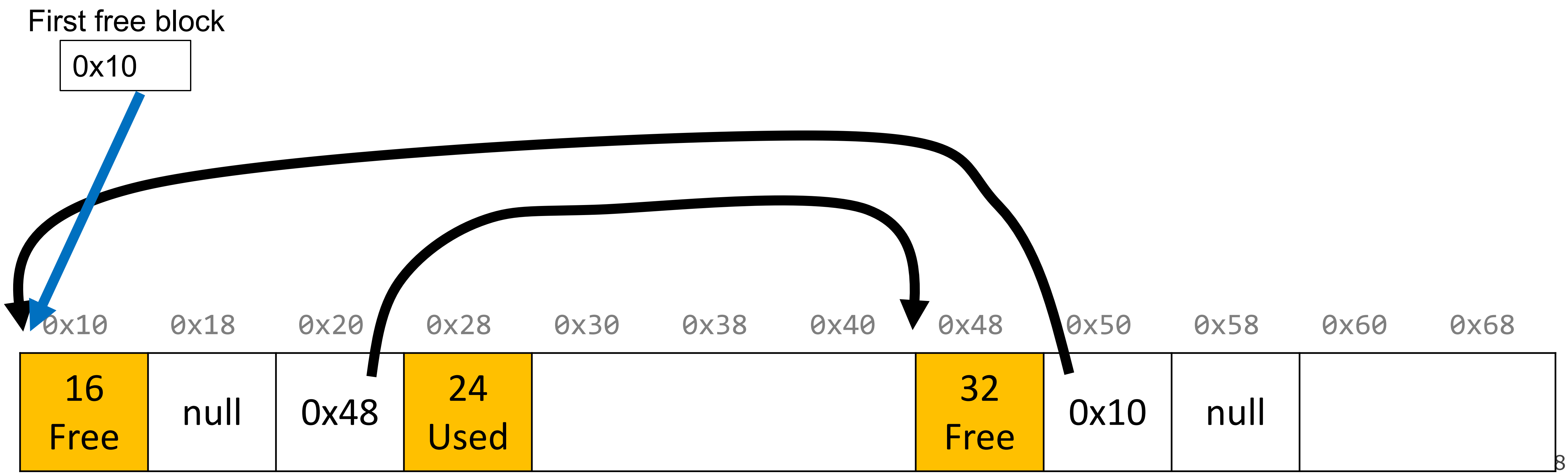
Can We Do Better?

- **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!



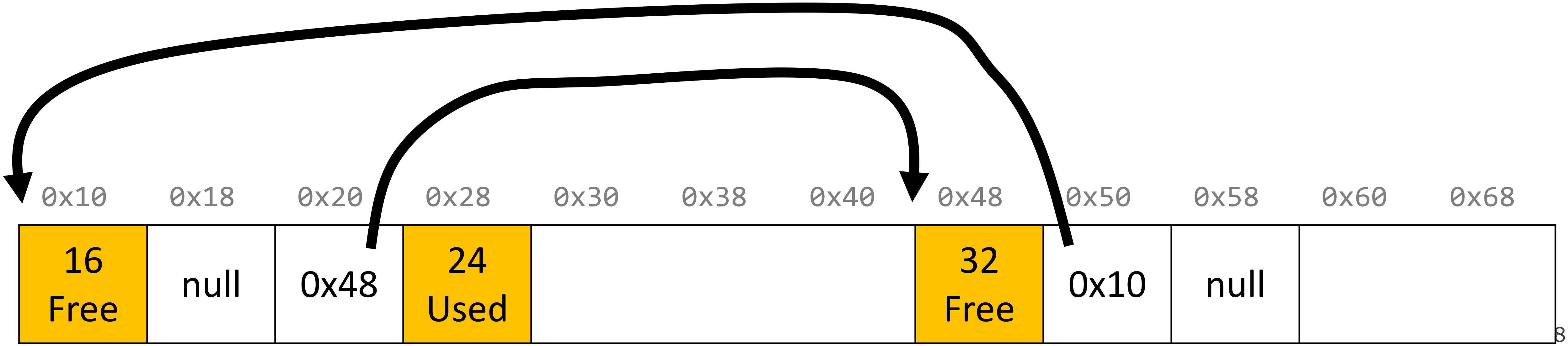
Can We Do Better?

- **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 <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)

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.)

Up to you!

Better memory util, Linear free

Constant free (push recent block onto stack)

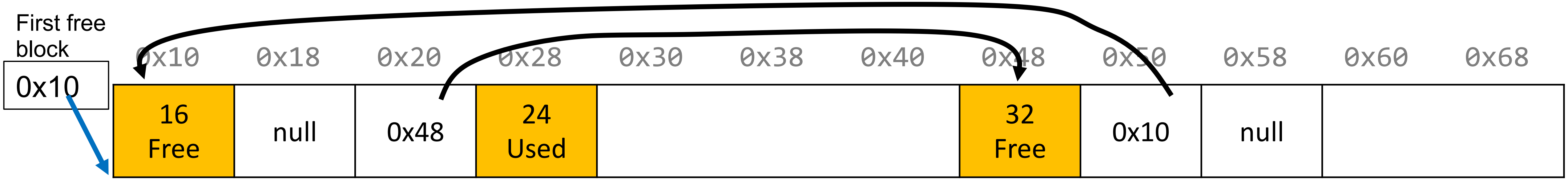
(more at end of lecture)

Explicit free list design

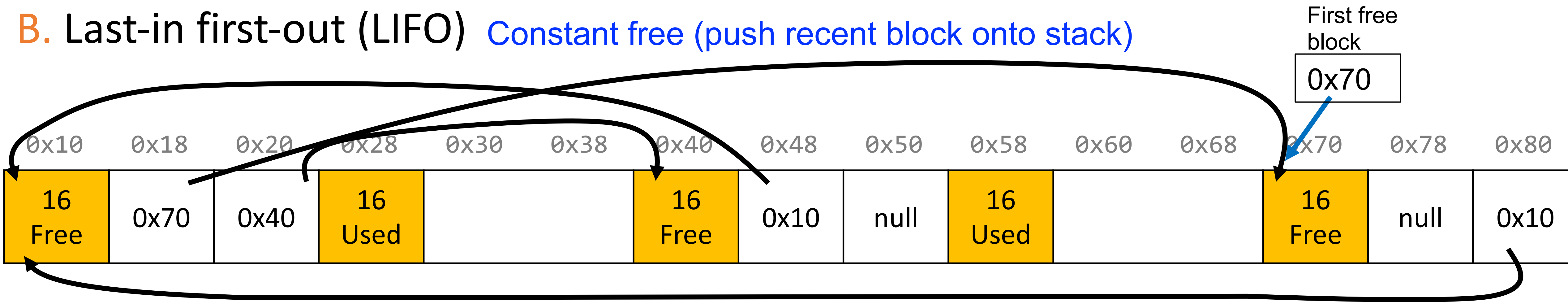
Up to you!

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

A. Address-order Better memory util, linear free



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



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 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?

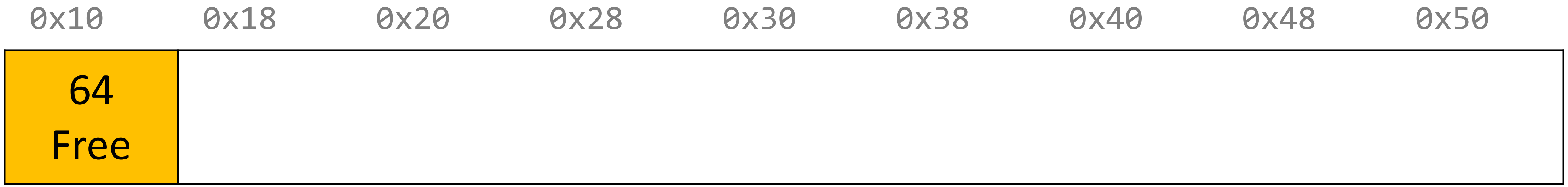
1. Can we avoid searching all blocks for free blocks to reuse? **Yes! We can use a doubly-linked list.**
2. **Can we merge adjacent free blocks to keep large spaces available?**
3. Can we avoid always copying/moving data during realloc?

Lecture Plan

- **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

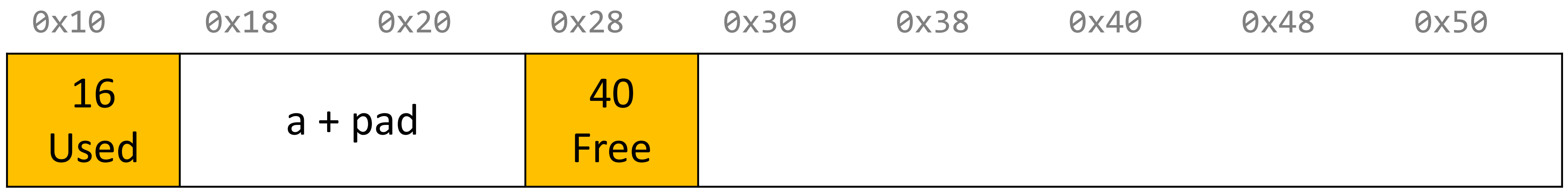
Coalescing

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



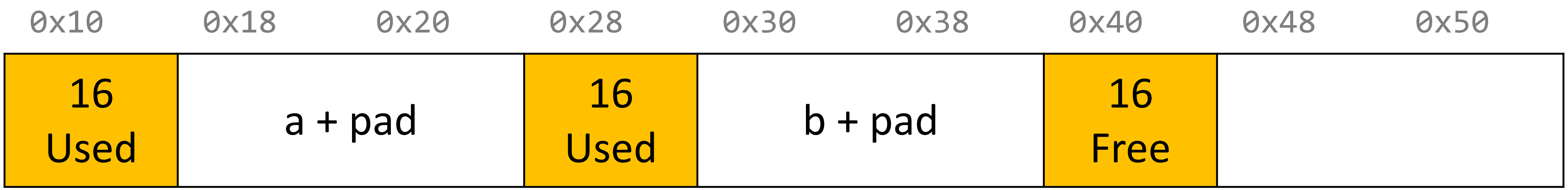
Coalescing

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



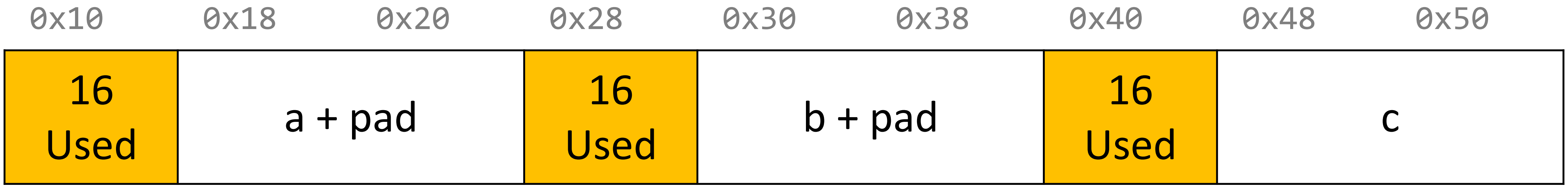
Coalescing

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



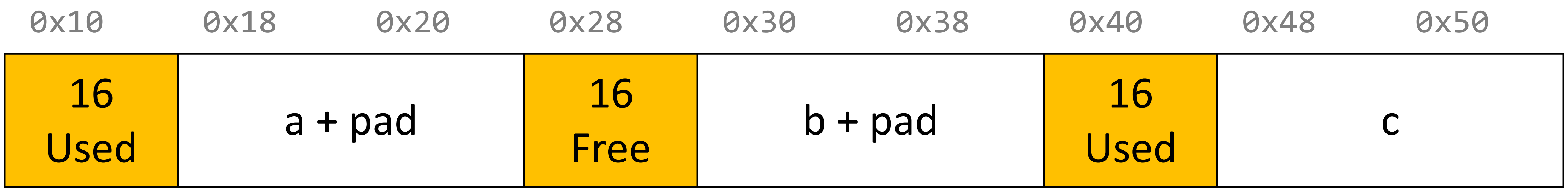
Coalescing

```
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);
```

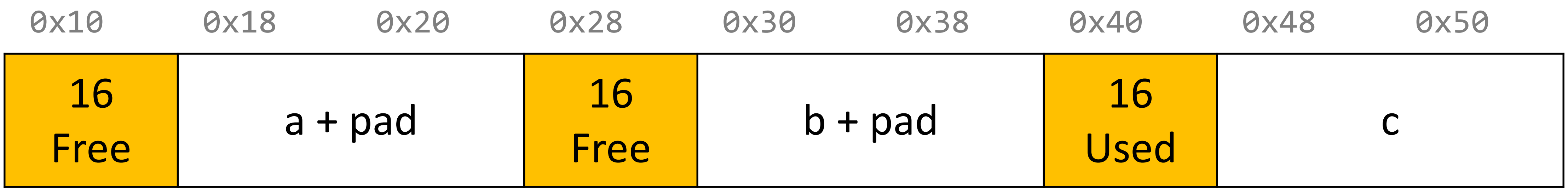
0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50
16 Free	a + pad		16 Free	b + pad		16 Used	c	

Coalescing

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

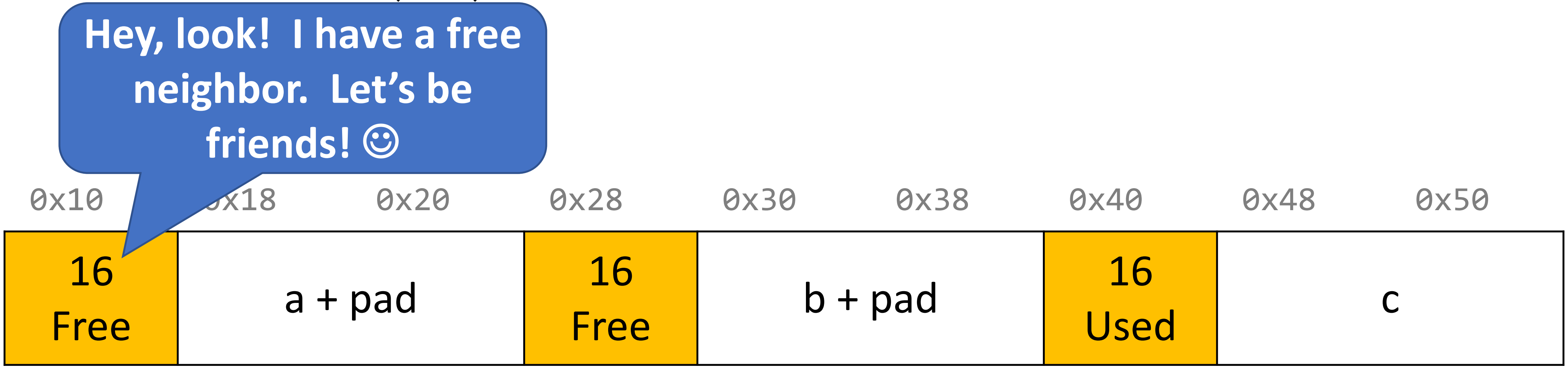
We have enough memory space, but it is fragmented into free blocks sized from earlier requests!

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



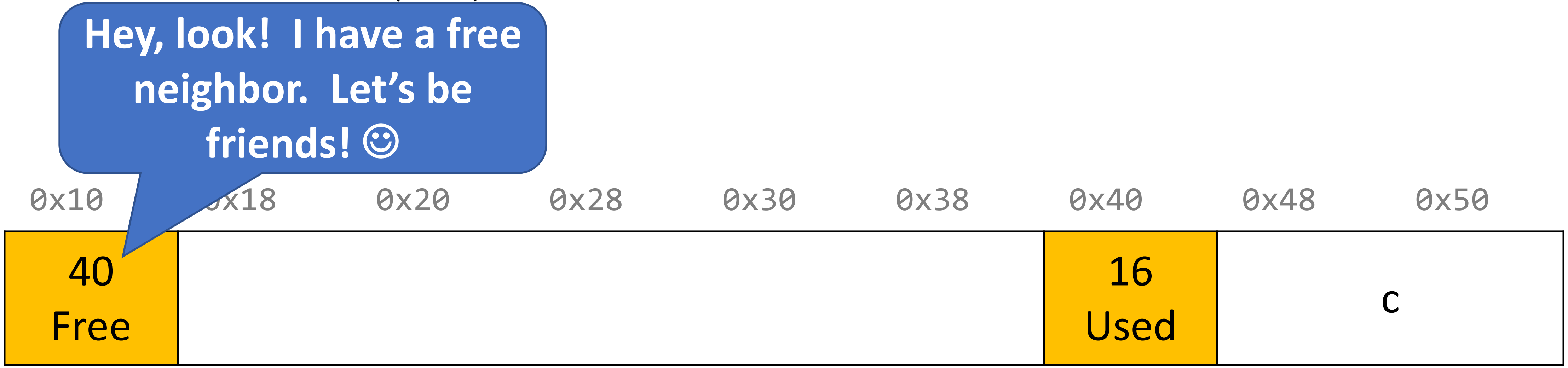
Coalescing

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



Coalescing

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

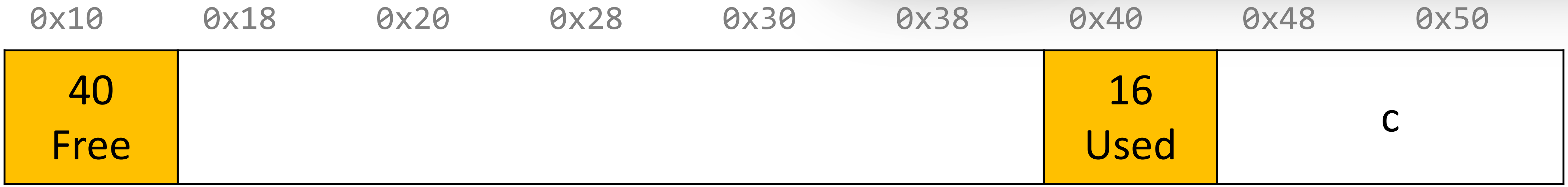


Coalescing

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

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**?

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

```
free(b);
```

0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
48 Free							16 Used	A	

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**

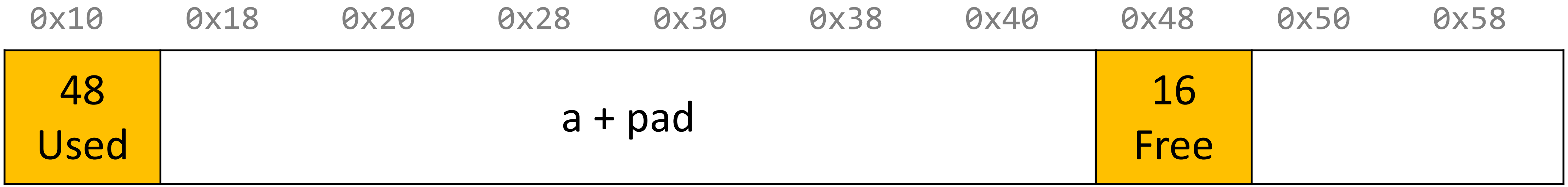
Realloc

- 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.

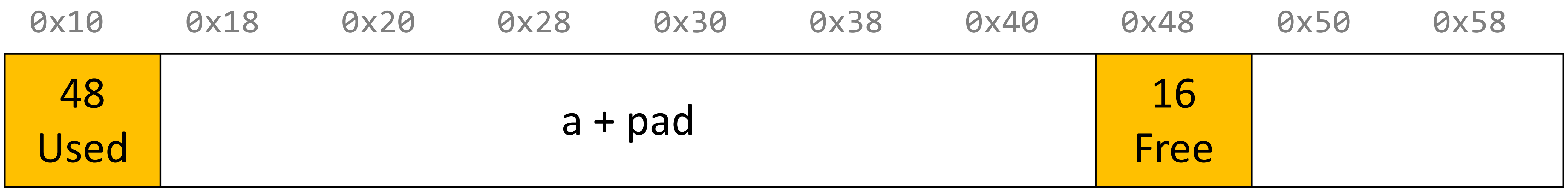


Realloc: Growing In Place

```
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.

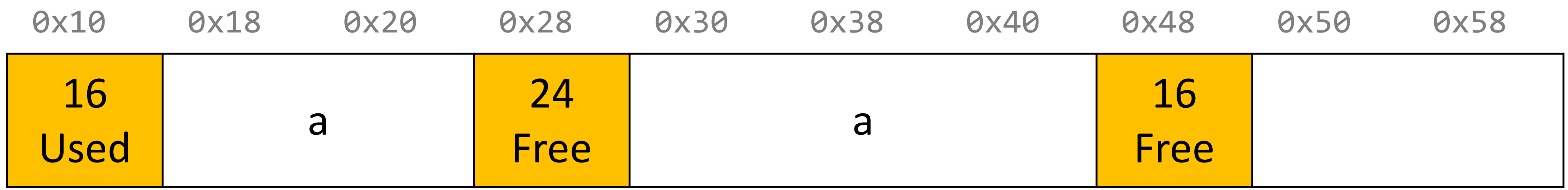


Realloc: Growing In Place

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

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

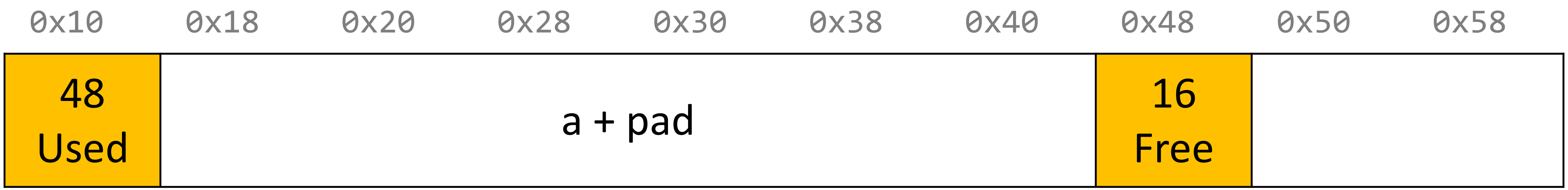
If we can, we should try to recycle the now-freed memory into another freed block.



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!

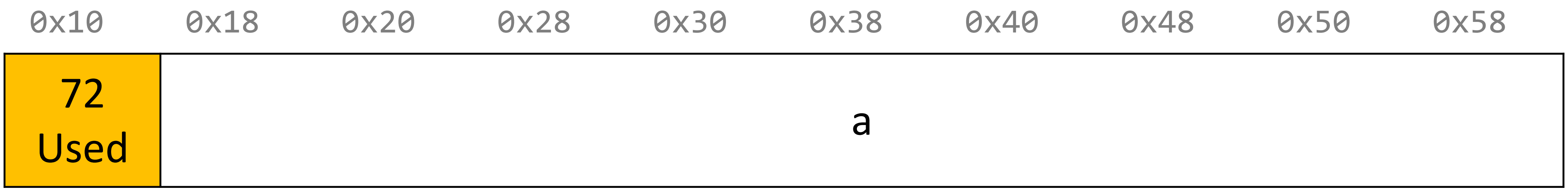


Realloc: Growing In Place

```
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!

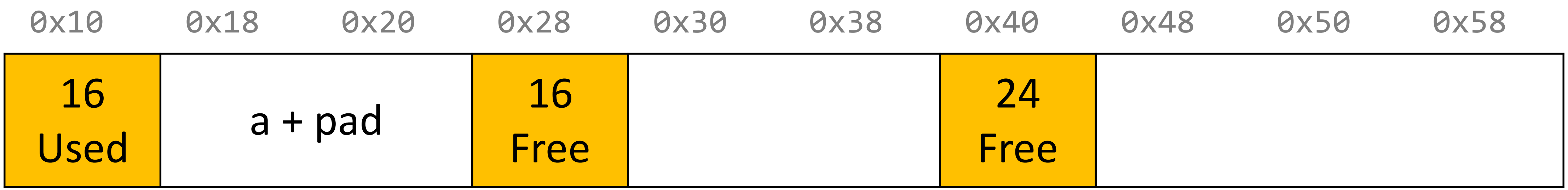
Now we can still return the same address.



Realloc: Growing In Place

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

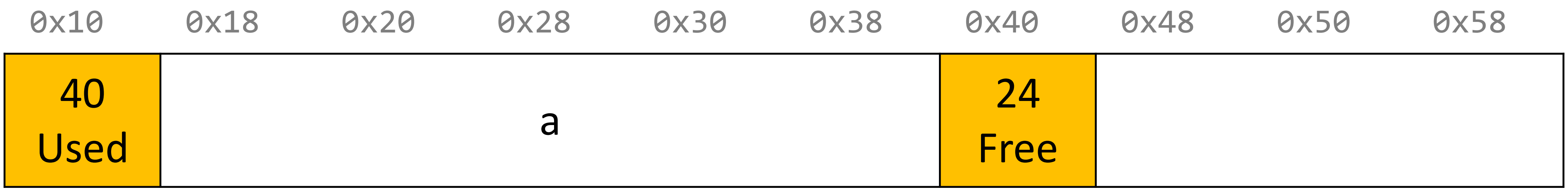
For your project (explicit only), you should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.



Realloc: Growing In Place

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

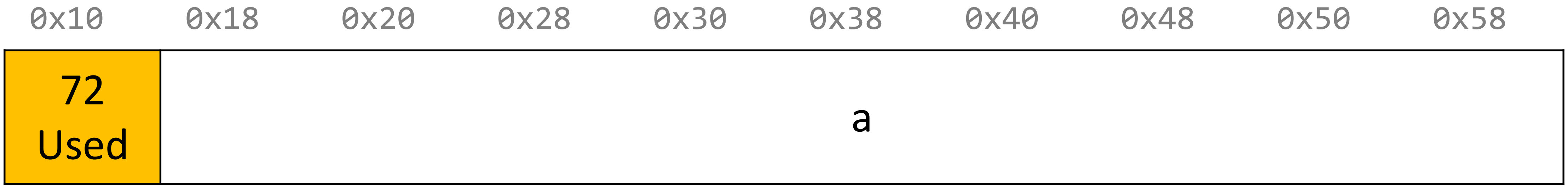
For your project (explicit only), you should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.



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.

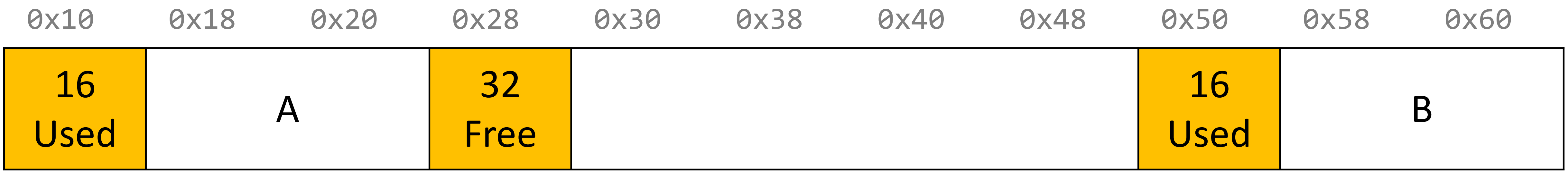


Realloc

- 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**?




```
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**?

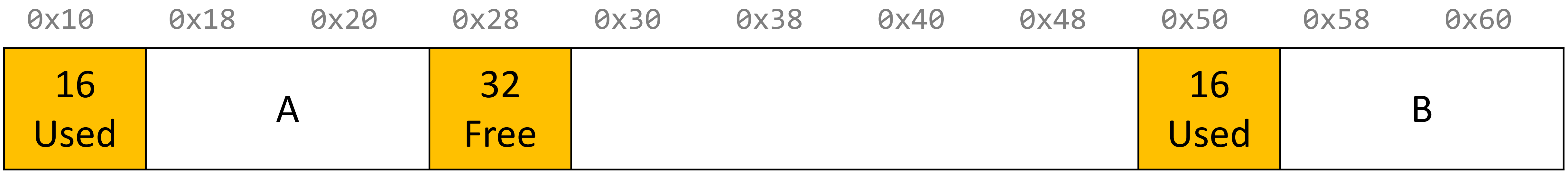
0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60
16 Used	A		32 Free					16 Used	B	

```
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**?

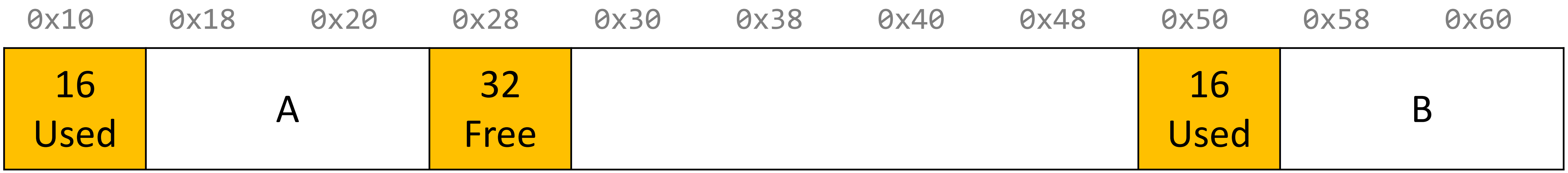
0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60
16 Used	A		32 Free					16 Used	B	

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

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

Practice 3: Explicit (realloc)

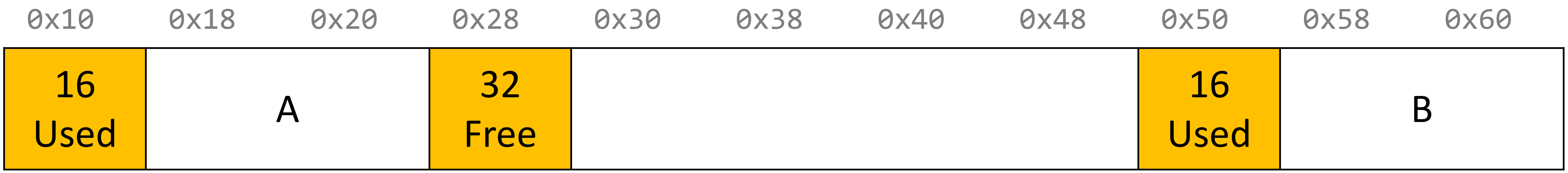
For the following heap layout, what would the heap look like after the following request is made, assuming we are using an explicit free list allocator with a **first-fit** approach and **coalesce on free + realloc in-place**?



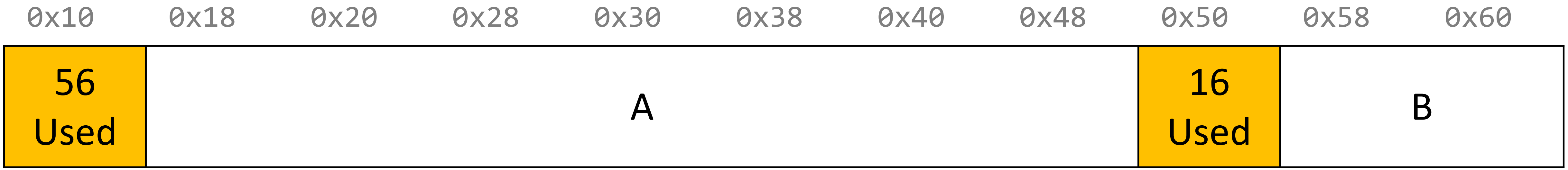

```
realloc(A, 48);
```

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**?

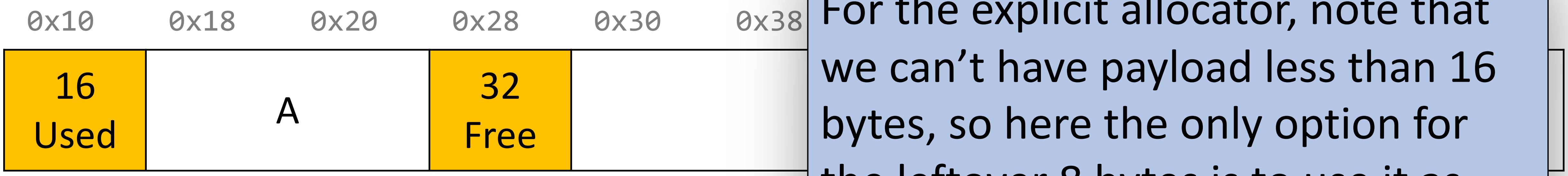



```
realloc(A, 48);
```



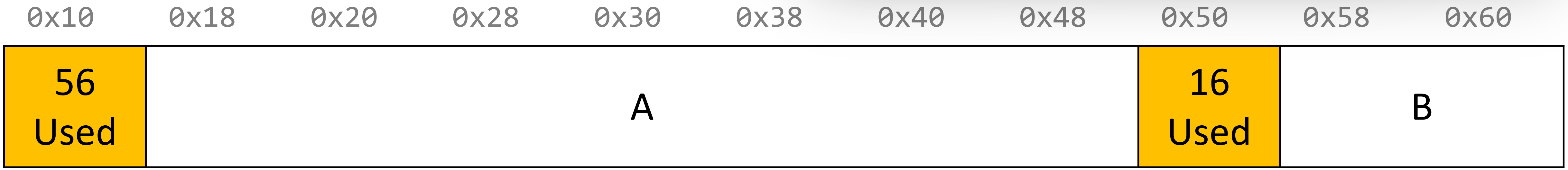
Practice 3: Explicit (realloc)

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**?



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);
```



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 23 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: optimization