CS107, Lecture 24
Explicit Free List Allocator

Reading: B&O 9.9, 9.11, 5 (Optimization)

CS107 Topic 6

How do the core malloc/realloc/free memory-allocation operations work?

Why is answering this question important?

- Combines techniques from across the quarter (bits/bytes, pointers, memory, generics, assembly, efficiency, testing, and more) to understand a real-world system that you have relied on all quarter!
- Learning about the design and tradeoffs in a real-world large system gives us a great example of how to evaluate different designs when there's no one "right" answer.

assign6: implement two different possible designs for a heap allocator, implementing malloc/realloc/free.

Learning Goals

- Learn about how we can implement coalescing of blocks and in-place realloc
- Understand the tradeoffs between bump, implicit and explicit free list allocators

Lecture Plan

- **Recap:** heap allocators so far
- Method 2: Explicit Free List Allocator

Lecture Plan

- **Recap: heap allocators so far**
- Method 2: Explicit Free List Allocator

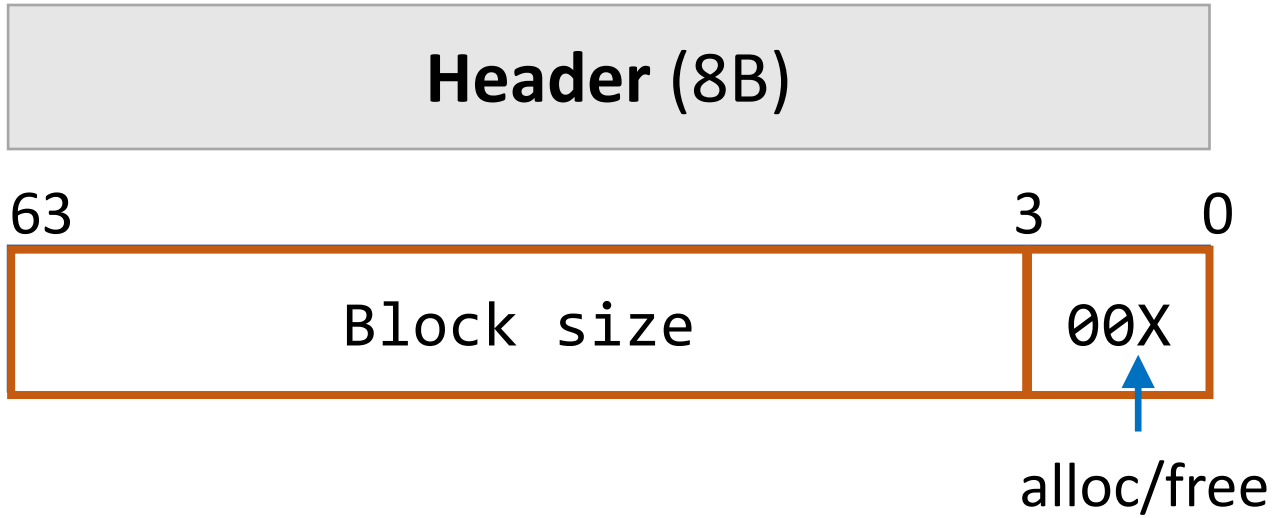
Bump Allocator

- 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 the final assignment as a code reading exercise.

Implicit Free List Allocator

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 ☺

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)*

Lecture Plan

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

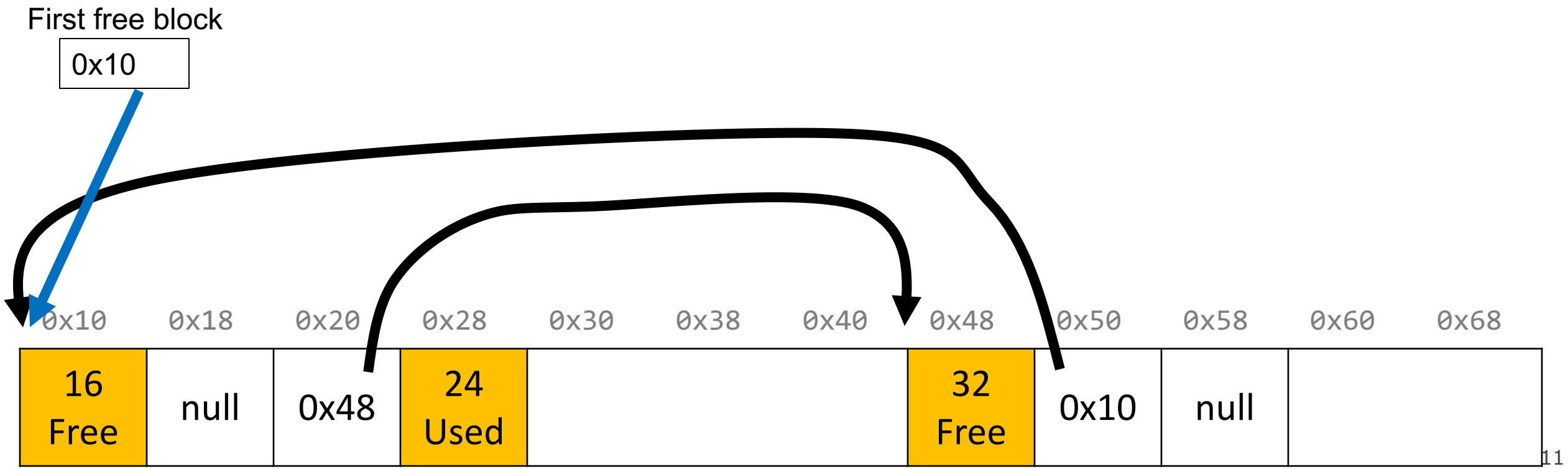
Explicit Free List Allocator

- 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 Allocator

- **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!



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.)

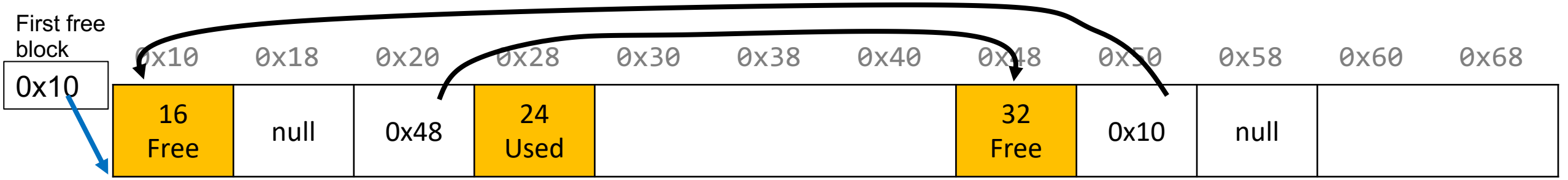
(more at end of lecture)

Explicit free list design

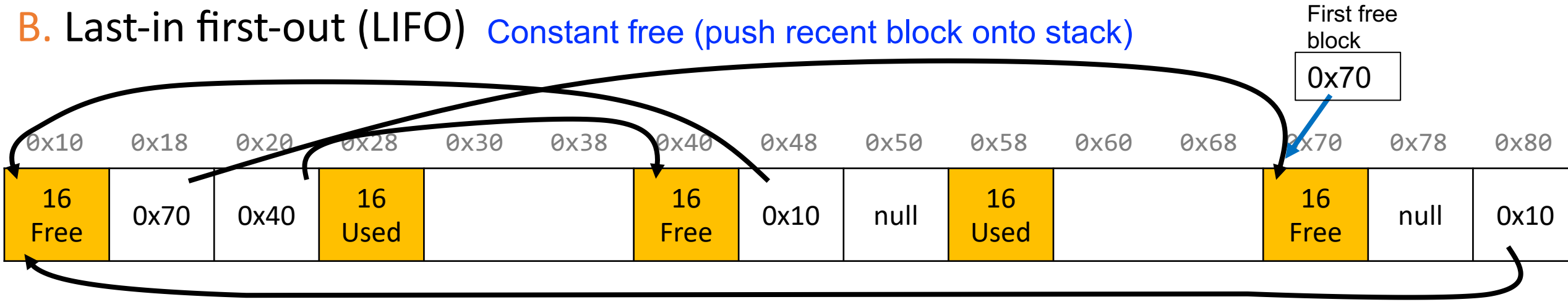
Up to you!

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?

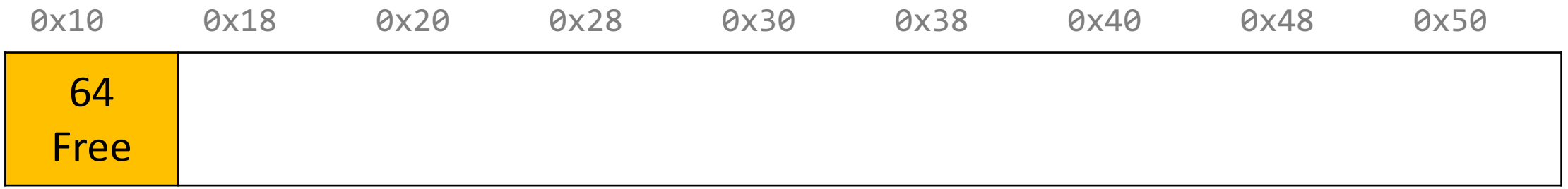
1. Can we avoid searching all blocks for free blocks to reuse? **Yes! We can use a doubly-linked list.**
2. **Can we merge adjacent free blocks to keep large spaces available?**
3. Can we avoid always copying/moving data during realloc?

Lecture Plan

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

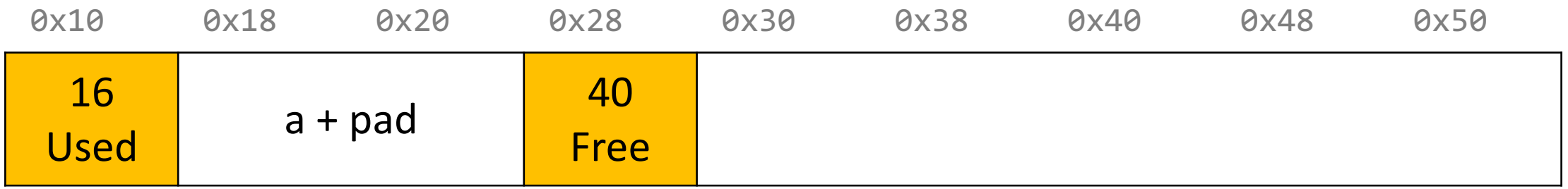
Coalescing

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



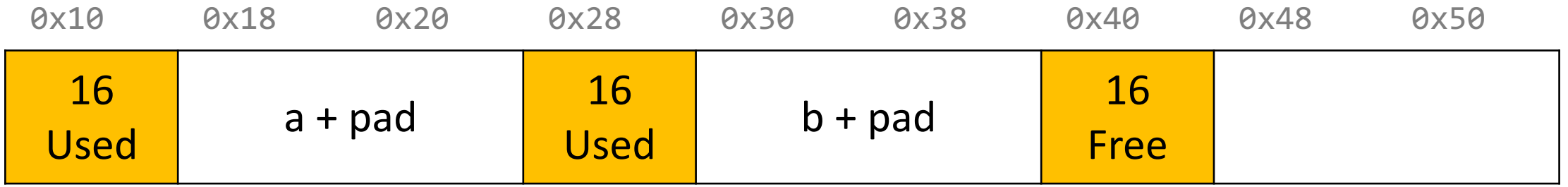
Coalescing

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



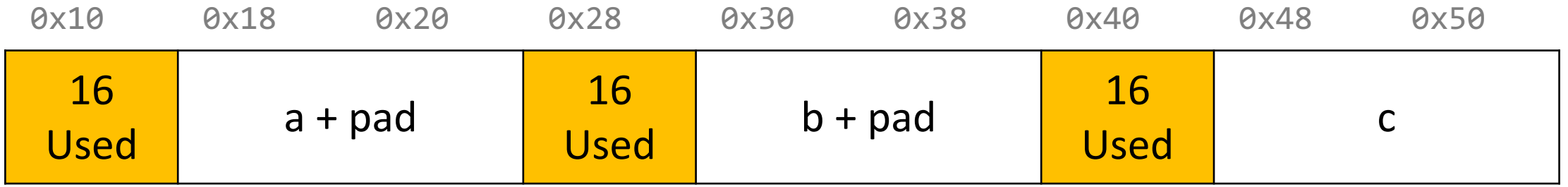
Coalescing

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



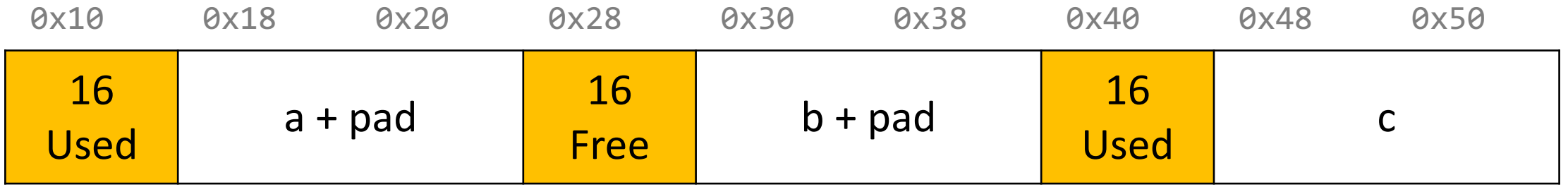
Coalescing

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



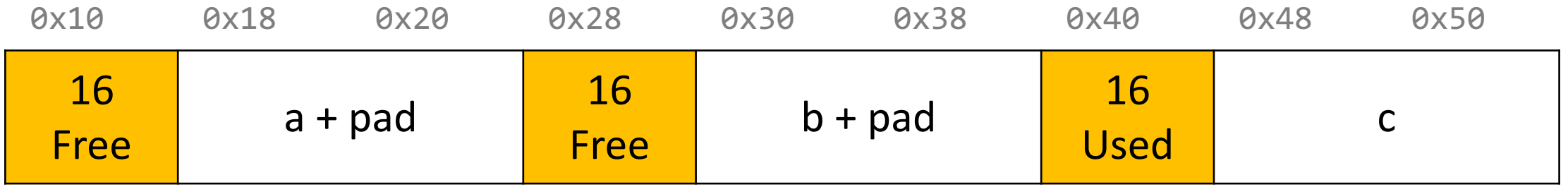
Coalescing

```
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);
```

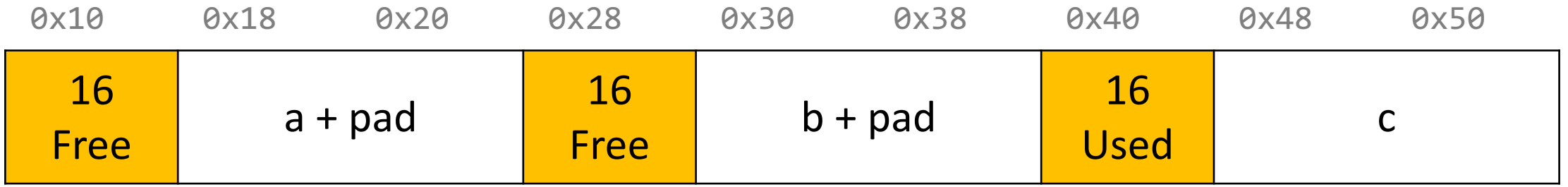


Coalescing

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

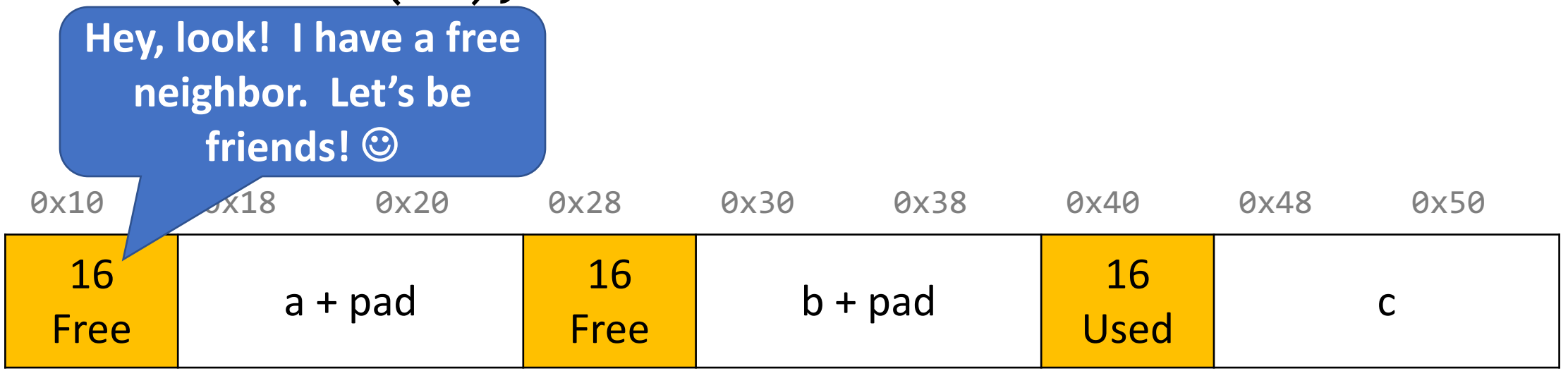
We have enough memory space, but it is fragmented into free blocks sized from earlier requests!

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



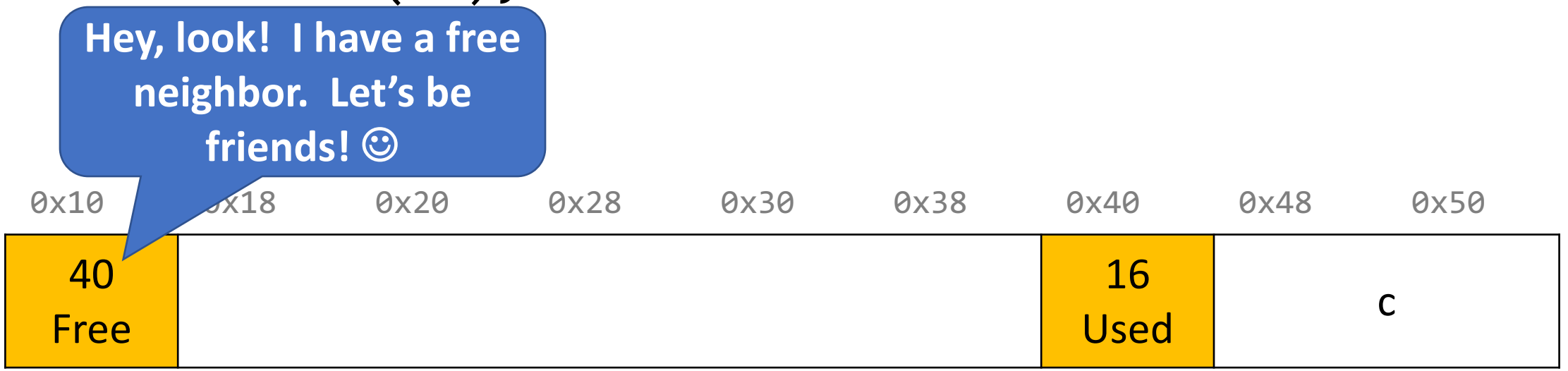
Coalescing

```
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);
```

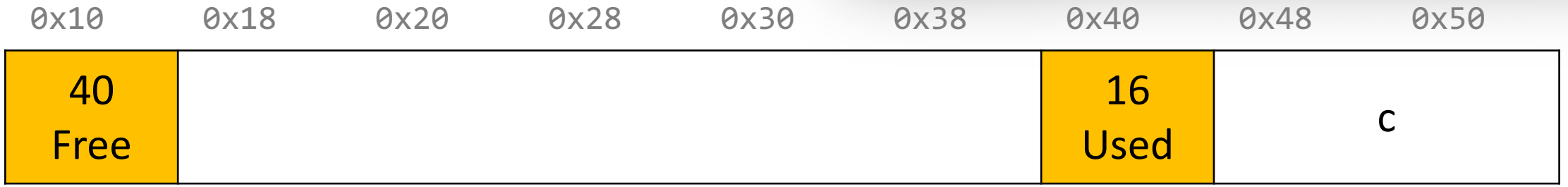


Coalescing

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

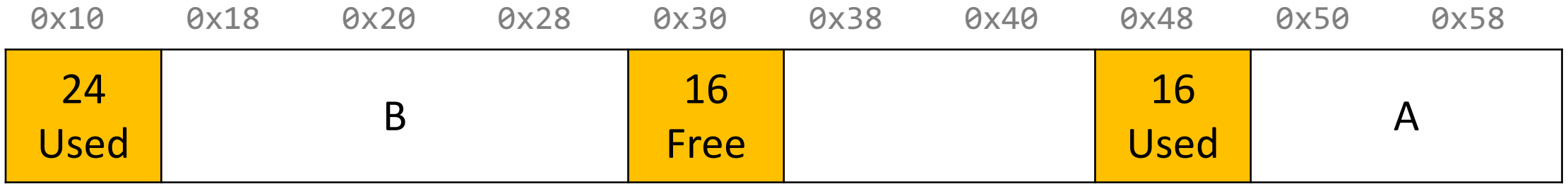
The process of combining adjacent free blocks is called *coalescing*.

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)

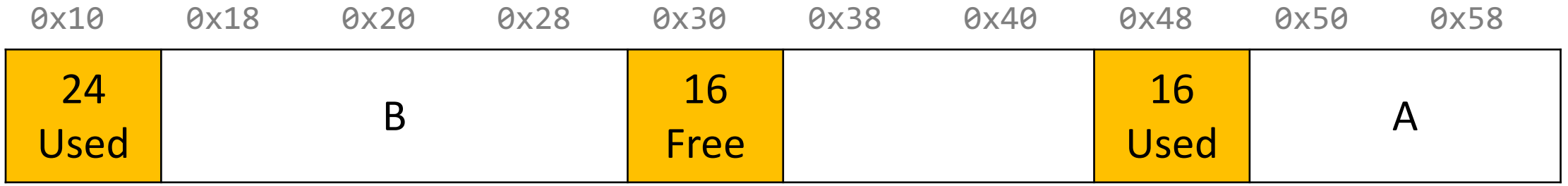
For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **explicit** free list allocator with a **first-fit** approach and **coalesce on free**?



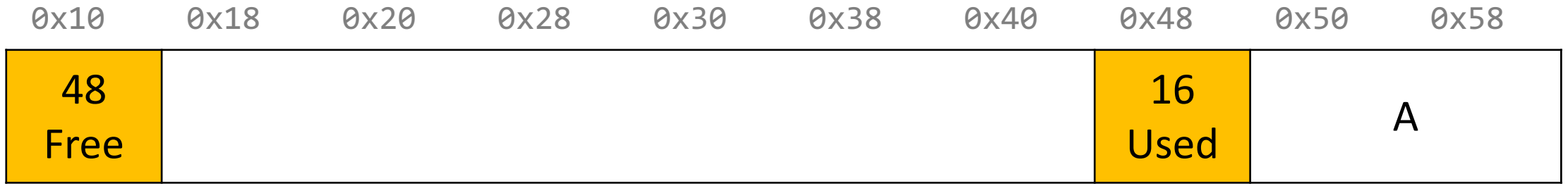

```
free(b);
```

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 2: Explicit Free List Allocator**
  - Explicit Allocator
  - Coalescing
  - **In-place realloc**

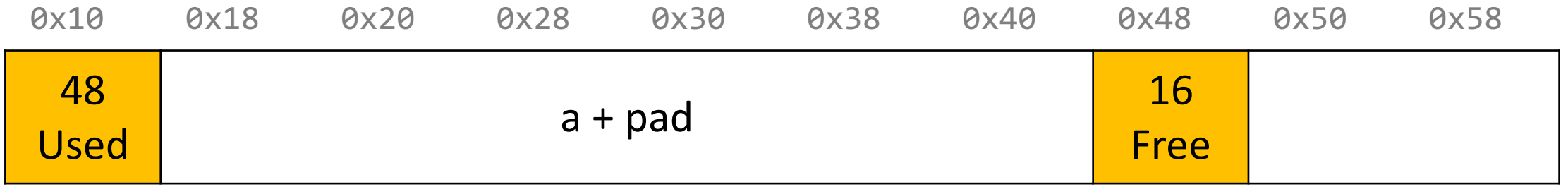
Realloc

- 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.

In-place realloc, case 1: size is growing, but we can use prior padding

```
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.

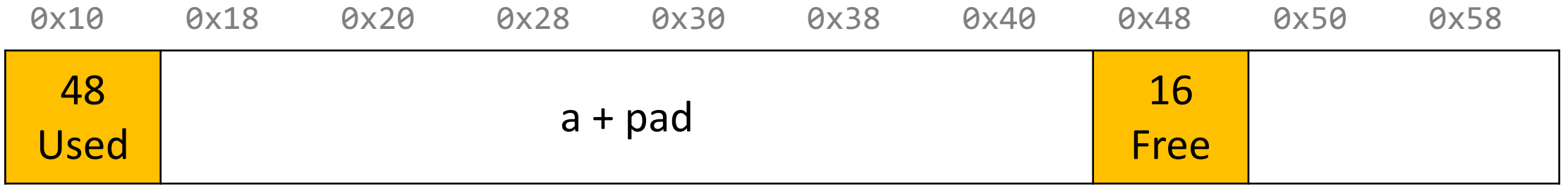


In-place realloc, case 2: size is shrinking, keep using existing block

```
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.

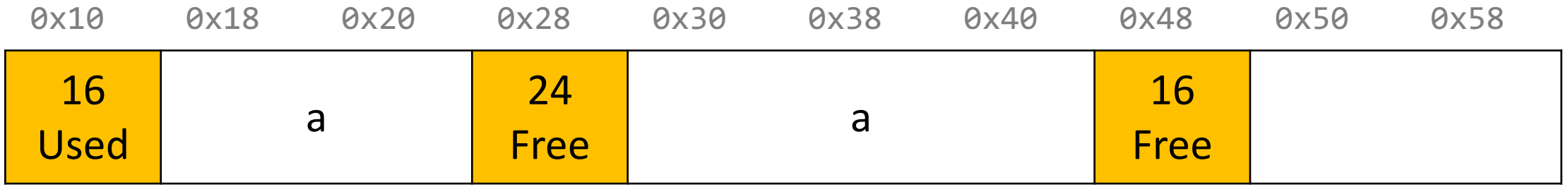


In-place realloc, case 2: size is shrinking, keep using existing block

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

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

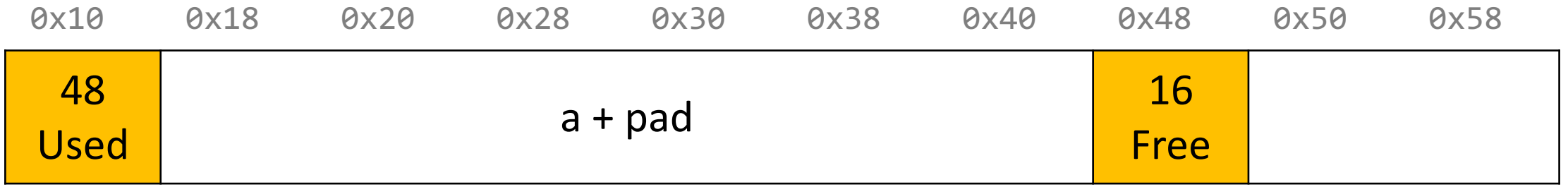
If we can, we should try to recycle the now-freed memory into another freed block.



In-place realloc, case 3: size is growing, block too small, but adjacent space available

```
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!

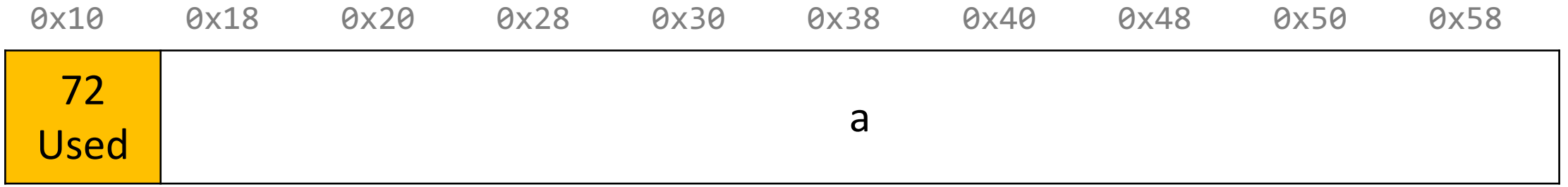


In-place realloc, case 3: size is growing, block too small, but adjacent space available

```
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!

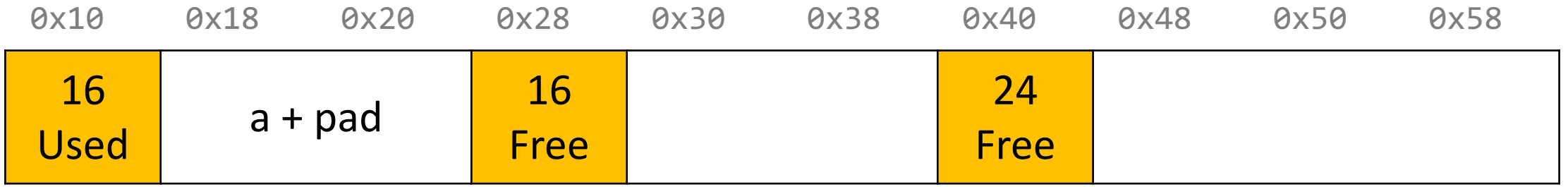
Now we can still return the same address.



In-place realloc, case 3: size is growing, block too small, but adjacent space available

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

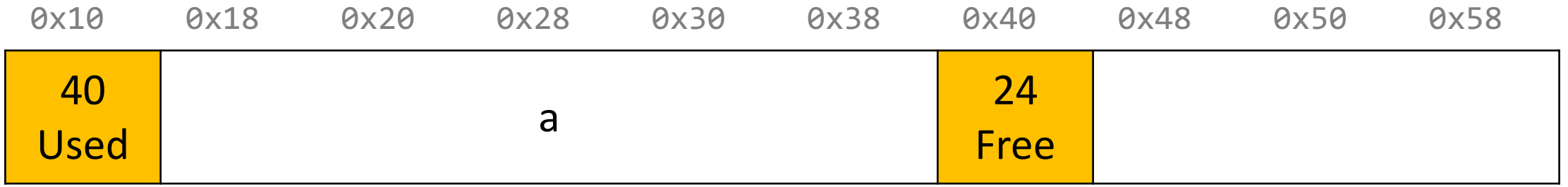
For your project (explicit only), you should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.



In-place realloc, case 3: size is growing, block too small, but adjacent space available

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

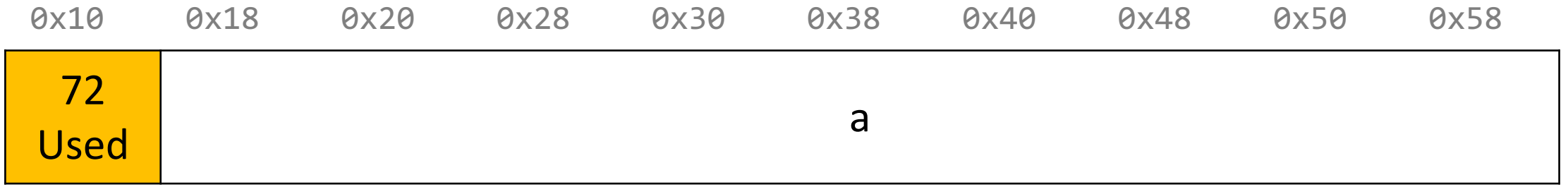
For your project (explicit only), you should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.



In-place realloc, case 3: size is growing, block too small, but adjacent space available

```
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.

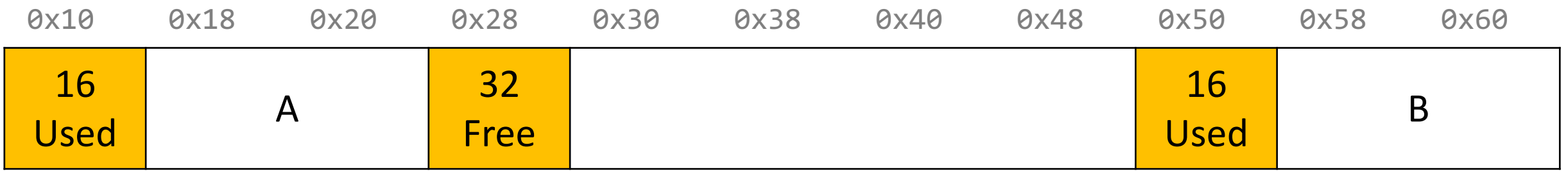


Realloc

- 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)

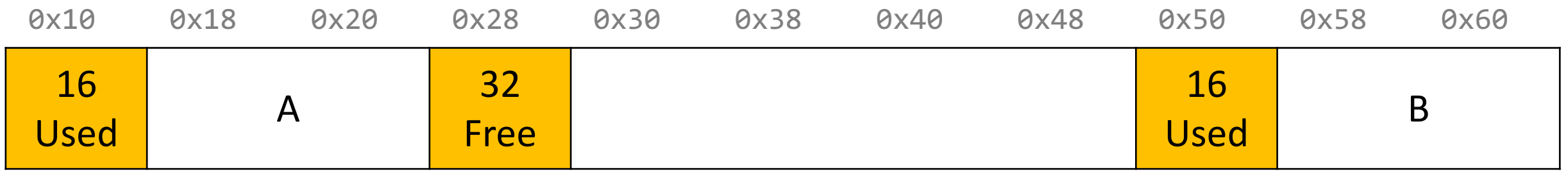
For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **explicit** free list allocator with a **first-fit** approach and **coalesce on free + realloc in-place**?



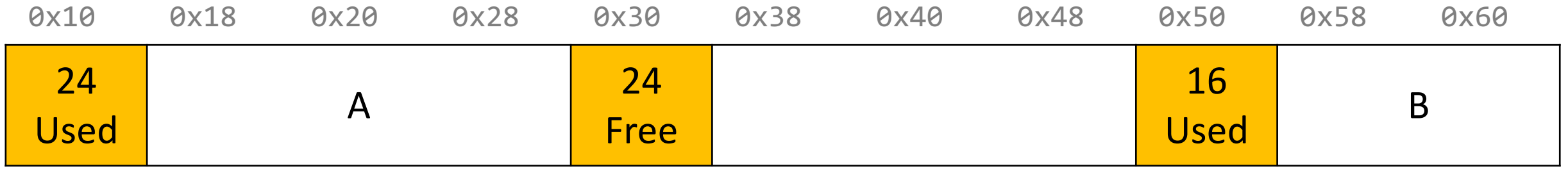

```
realloc(A, 24);
```

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**?

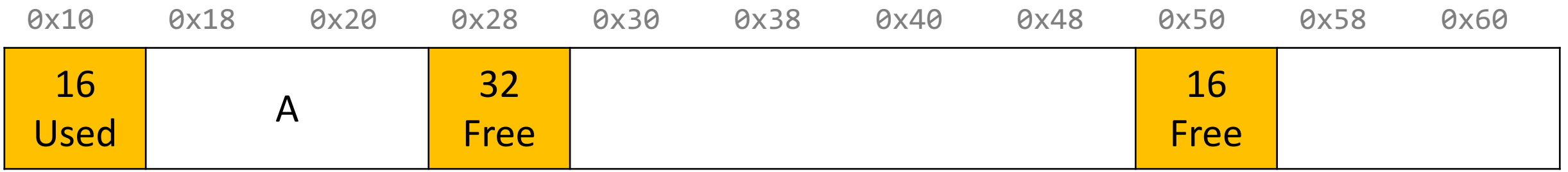



```
realloc(A, 24);
```



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**?

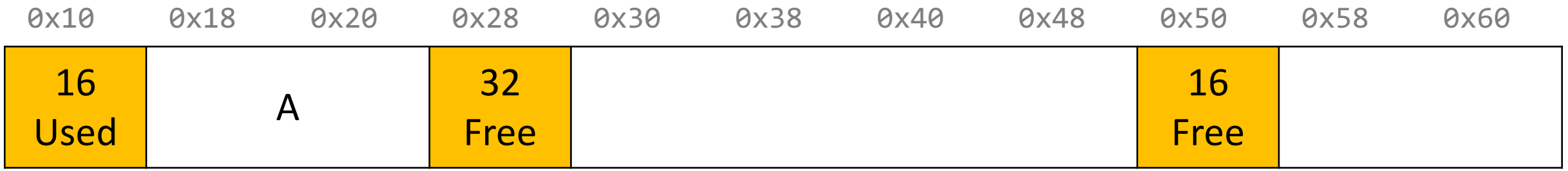



```
realloc(A, 56);
```

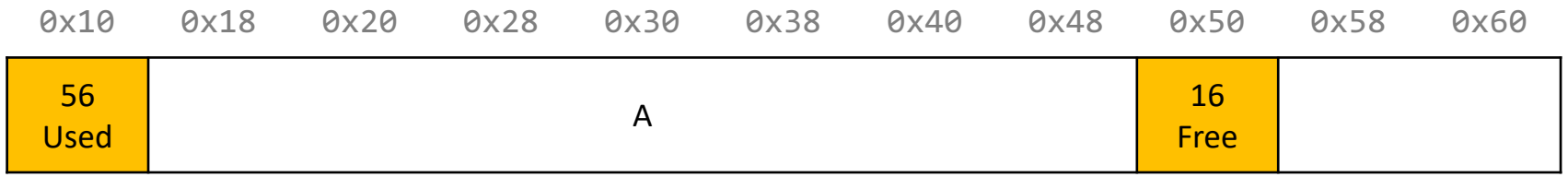
Respond on PollEv: pollev.com/cs107
or text CS107 to 22333 once to join.

Which image represents what the heap would look like following the request?

Practice 2: Explicit: Options

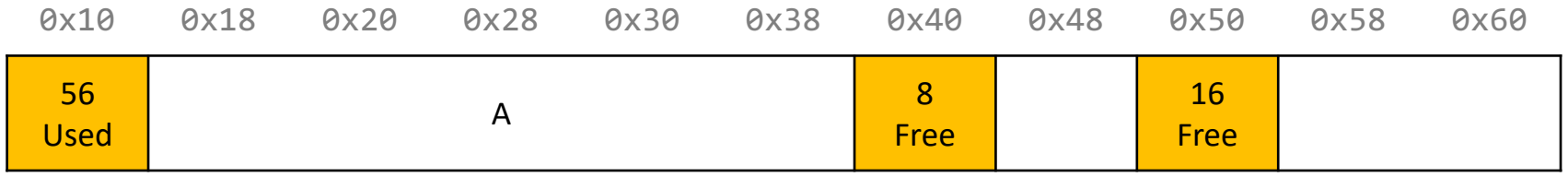


`realloc(A, 56);`

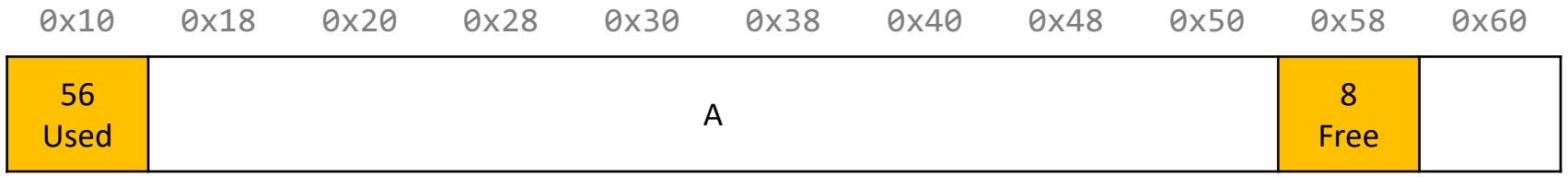


✅ *combine just as much as needed*

❌ *Only combine as much as needed*



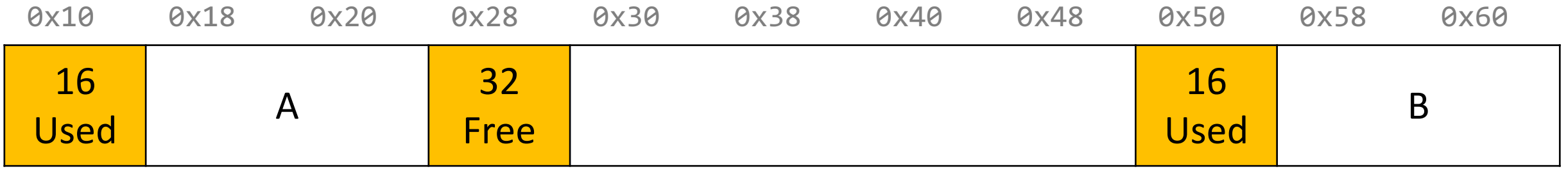
❌ *Space not tracked correctly*



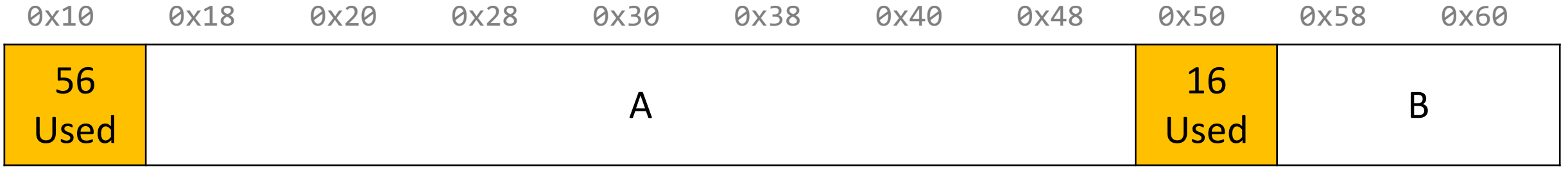
❌ *Space not tracked correctly*

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**?

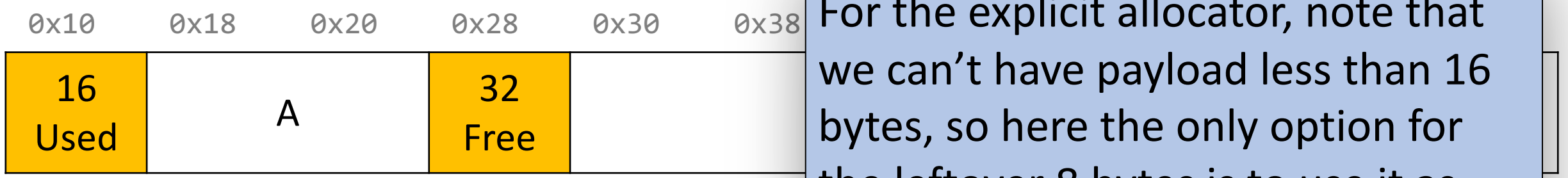



```
realloc(A, 48);
```



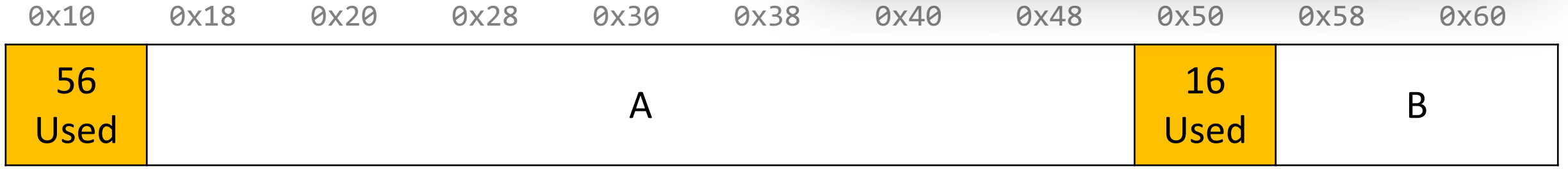
Practice 3: Explicit (realloc)

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**?



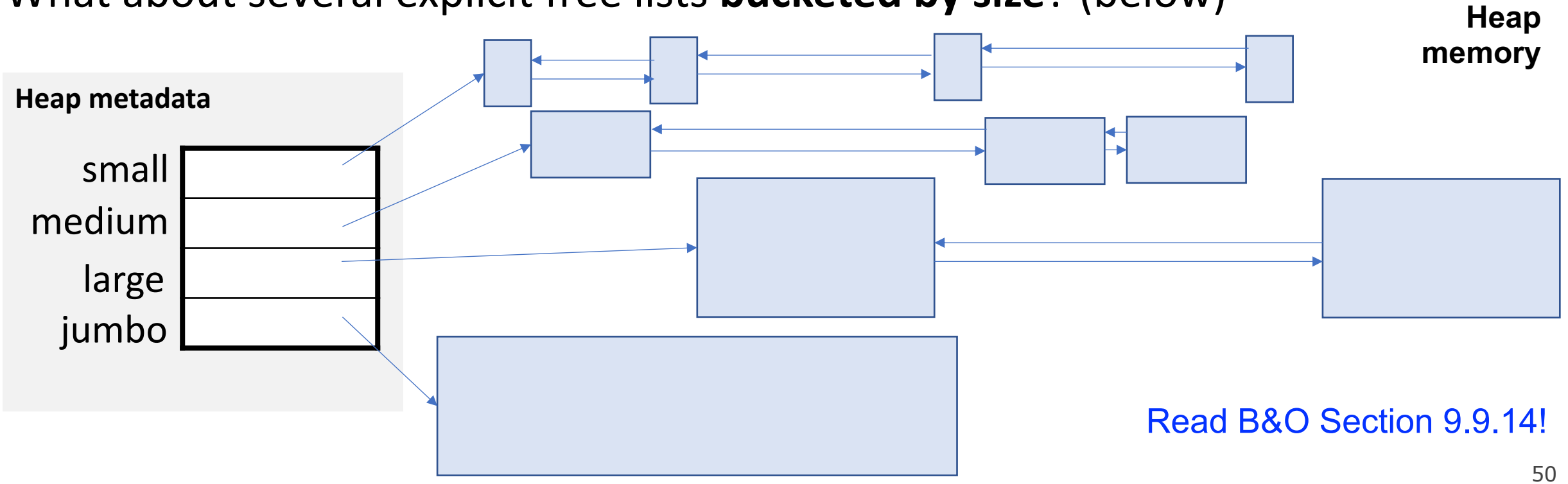
For the explicit allocator, note that we can't have payload less than 16 bytes, so here the only option for the leftover 8 bytes is to use it as padding for the existing block.

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



Going beyond: Explicit list w/size buckets

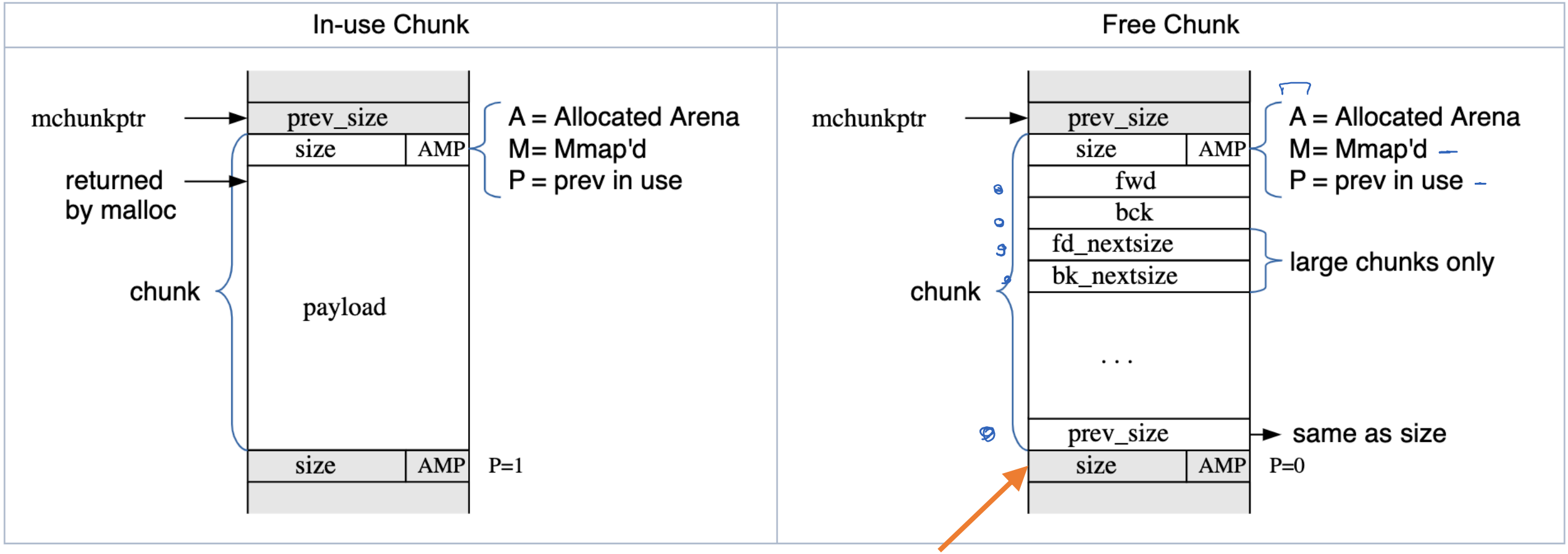
- Explicit lists are much faster than implicit lists.
- However, a first-fit placement policy is still linear in total # of free blocks.
- What about an explicit free list **sorted by size** (e.g., as a tree)?
- What about several explicit free lists **bucketed by size**? (below)



Read B&O Section 9.9.14!

In the wild: glibc allocator

- https://sourceware.org/glibc/wiki/MallocInternals



Footer/Boundary tag (see textbook)

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

Work Incrementally

- Don’t implement all features at once; instead, implement features gradually, testing thoroughly at each step. For instance, start just by implementing **myinit** and a basic **mymalloc**, and then test them. Then add **myfree**, and later **myrealloc**.

Shrink test cases when debugging and leverage debugging checklist

- When encountering a bug, shrink the test case as much as possible before diving deeper into debugging. Large test case files are difficult to debug with GDB; instead, shrink the test case down first.

Read B&O textbook.

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

Final Project Tips

Helper Hours

- Our focus is to support you in tracking down your own bugs and leveraging the debugging strategies we've developed this quarter! For this reason, we are happy to help with code-level questions, but we cannot look at assignment code in helper hours.
- We can provide good debugging techniques and strategies!
- Come and discuss design tradeoffs!

Honor Code/collaboration

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

Recap

- **Recap:** heap allocators so far
- Method 2: Explicit Free List Allocator

Lecture 24 takeaway: The explicit free list allocator uses headers and also stores pointers to free blocks in free block payloads. 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