CS107 Lecture 21
Managing The Heap, Take I

Reading: B&O 9.9 and 9.11

CS107 Topic 6: Heap Allocators

How can a heap allocator's **core** **`malloc`**, **`realloc`**, **and** **`free`** **operations** be **designed** and **implemented**? What factors are crucial?

- They should handle **arbitrary sequences** of allocation and deallocation requests.
  - Allocators **shouldn't assume anything about the order** of **`malloc`**, **`realloc`**, and **`free`** requests.
- They should track what memory has been **allocated** and what memory is **free**.
  - Allocators must **maintain metadata** associated with each memory block and mark them as **in use** versus **free** if they're to distinguish between the two. Otherwise, it can't easily handle **`malloc`**, **`realloc`**, and **`free`** requests.
- They need to **decide what memory to use** to fulfill an allocation request.
  - Heap allocator often have **options** to choose from when fulfilling allocation requests. It must choose an appropriate block of memory **based on its allocation strategy**.
- They should service requests **as quickly as possible**.
  - Heap allocators should respond to allocation requests 🔥 **with blazing speed** 🔥. Otherwise, their implementations will be viewed as **inefficient** and the source of **bottlenecks** in programs that rely on them.
- Addresses returned by **`malloc`** and **`realloc`** must satisfy required **alignment restrictions** (e.g., addresses must be multiples of 8) to keep the hardware **happy**. 🤓

Heap Allocator Goals

The design and implementation of an allocator should strive to:

- maximize **throughput** — that is, requests completed per unit time — by **minimizing average allocation time**.
- maximize **utilization** — that's how efficiently we make use of **limited heap memory** to satisfy requests.

As we'll soon see, throughput and utilization fight each other.

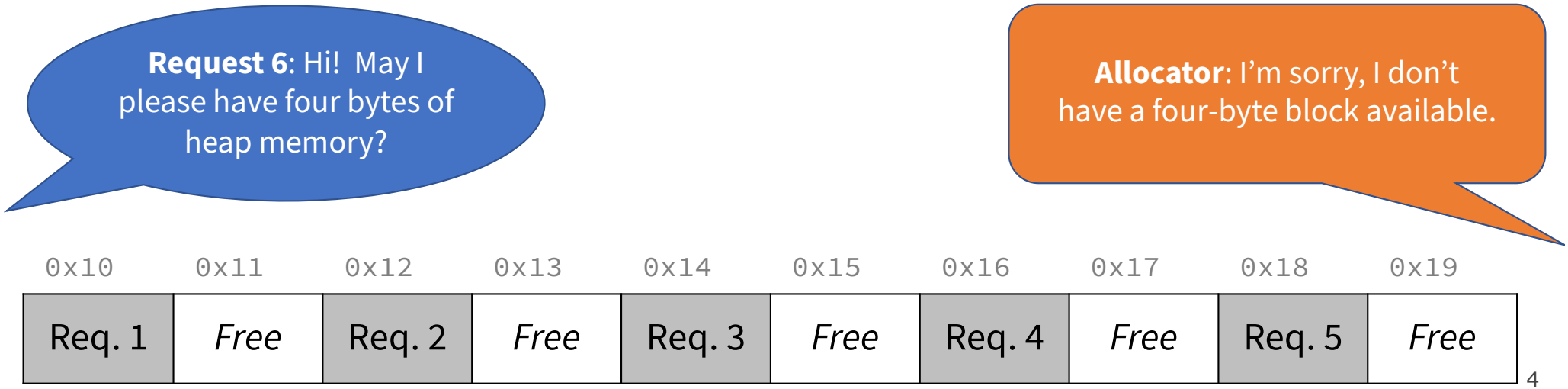
So, an allocator needs to **keep track of free memory blocks**, **decide which block to use**, and do it **quickly** and **compactly**.

That means we need a **concrete, in-memory representation** for each block.

Fragmentation: Utilization's Nemesis

The root cause of poor utilization is **fragmentation**. Fragmentation occurs when **otherwise unused memory isn't available to satisfy requests**.

- In this example, there's enough memory in aggregate to satisfy the allocation request, but no **single free block** is large enough.
- In general, we want the largest address used to be as low as possible.

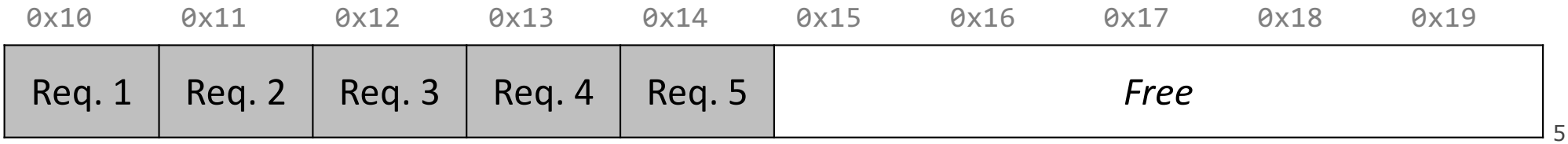


Fragmentation: Utilization's Nemesis

Question: What if we **coalesced** to unify free blocks? Can we do this?

- Yes, **clever idea**! Let's do that!
- **Sure**, it can be done, but doing so will **compromise throughput**.
- **No, it can't be done**.

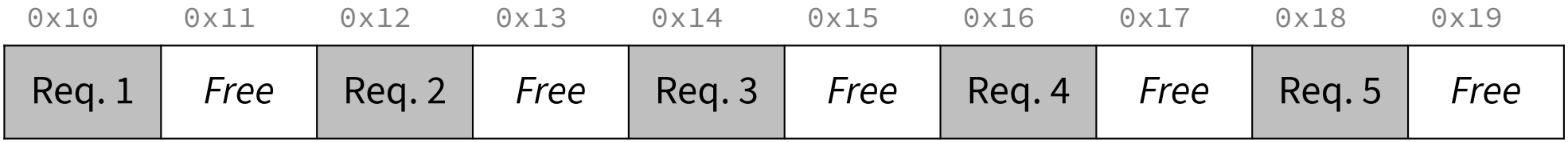
Nope. The original addresses have been **shared with client code** and are presumably being held in variables and data structures. We can't rearrange allocated memory blocks, since doing so would **invalidate** those addresses.



Fragmentation: Utilization's Nemesis

There are two types of fragmentation.

- **Internal Fragmentation**: an allocated block is **larger than what's needed**—e.g., due to minimum block size, or alignment restrictions.
  - An example? **`malloc(45)`** is internally promoted to **`malloc(48)`**. Those three extra bytes are **technically inaccessible** to the client but **can't be used** for future requests.
- **External Fragmentation**: no single block is large enough to satisfy an allocation request, even though enough aggregate free memory is available.
  - An example? Precisely what you saw two slides ago.



Heap Allocator Goals

The design and implementation of an allocator should strive to:

- maximize **throughput** — that is, requests completed per unit time — by **minimizing average allocation time**.
- maximize **utilization** — that's how efficiently we make use of **limited heap memory** to satisfy requests.

Other **noble goals**:

Locality ("similar" blocks allocated close to each other)

Robustness (handle client errors)

Ease of implementation

Case Study: Bump Allocator

Let's say we want to **prioritize throughput at all costs** but **not care about utilization** even one bit. This might even mean we **never reuse memory**—even after it's been freed!

Throughput

Ultra fast, short routines

What would this allocator look like?

Utilization

Never reuses memory

Case Study: Bump Allocator

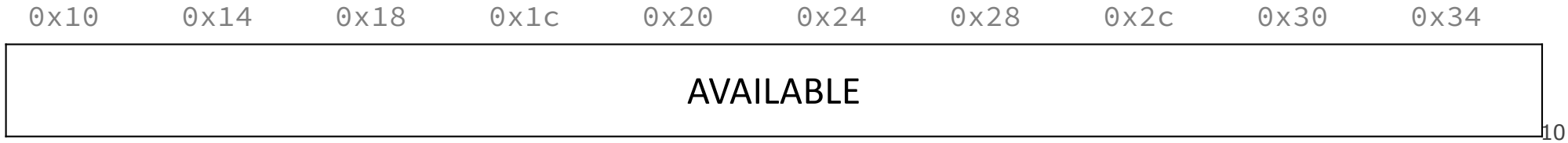
A **bump allocator** is an allocator that simply **allocates the next available memory location** in response to both **`malloc`** and **`realloc`** requests and does **nothing** in response to **`free`**.

- **Throughput**: each **`malloc`** call executes **only a handful of instructions**, and the implementation of **`free`** is empty—**essentially a no-op**.
  - It is easy to find the next memory location to service **`malloc`** and **`realloc`** requests.
  - **`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 a code reading exercise in your **`assign6`** repo.

Case Study: Bump Allocator

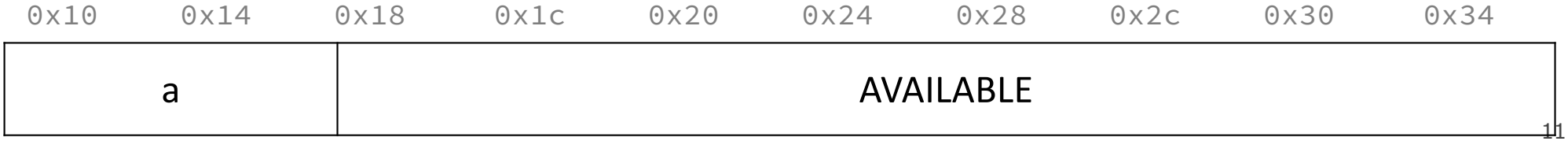
```
void *a = malloc(8);
void *b = malloc(4);
void *c = malloc(24);
free(b);
void *d = malloc(8);
```



Case Study: Bump Allocator

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

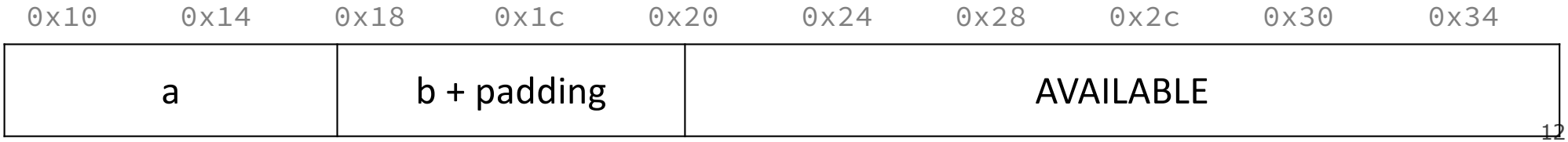
Variable	Value
a	0x10



Case Study: Bump Allocator

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

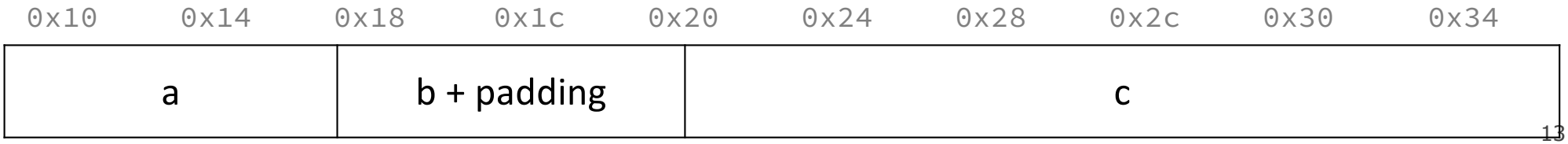
Variable	Value
a	0x10
b	0x18



Case Study: 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

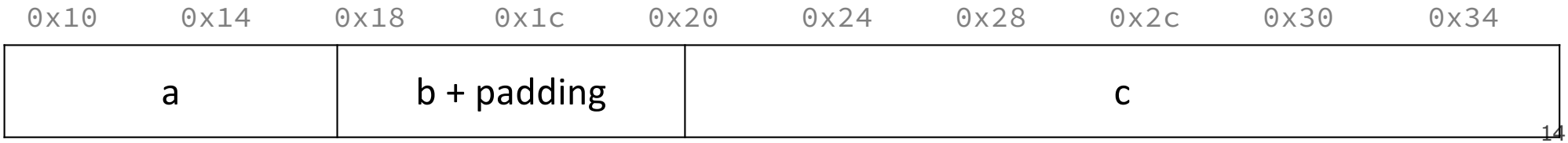


Case Study: Bump Allocator

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

Note that the value of **b** itself doesn't change. It's just that the memory at address **0x18** supposed to be dead.

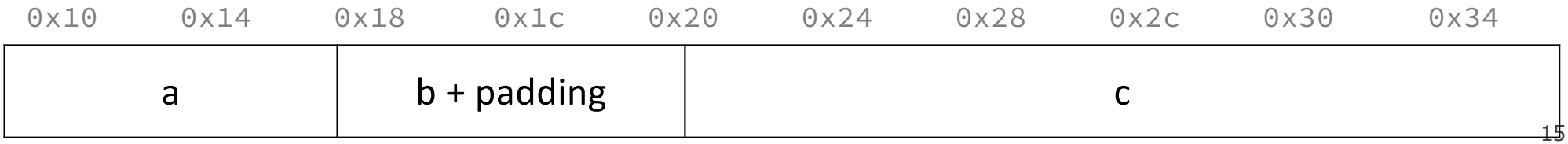
Variable	Value
a	0x10
b	0x18
c	0x20



Case Study: 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 **extreme**–it optimizes only for **throughput**.

Better allocators strike a more **reasonable balance** to achieve acceptable—even admirable—levels for both.

Questions to ponder:

1. How do we **keep track** of free blocks?
2. How do we **choose which free block** to service a request?
3. After choosing a block, what do we do with any **excess space**?
4. What do we do with a block **when it's freed**?

Case Study: Implicit Free List Allocator

Key idea: if we're to reuse blocks, we need a way to track which blocks are allocated and which ones are free.

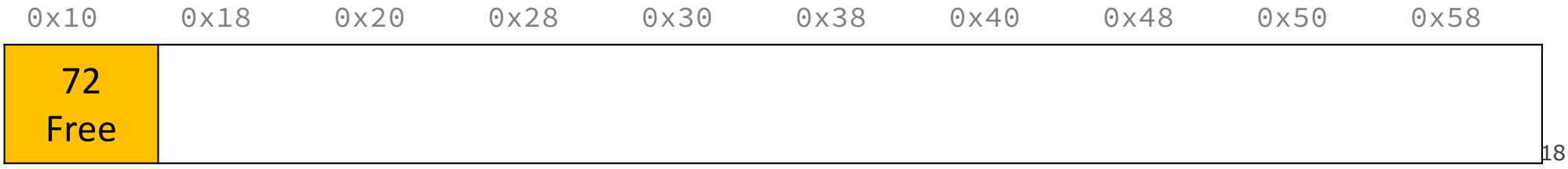
- We could store this information in a **separate global data structure**, but this is, in general, **inefficient** and requires substantial **overhead**.

Another Key Idea: let's allocate extra space before each block for a **header** storing its **payload size** and whether it's **free** or **in use**.

- When we **`malloc`** memory, we **search all blocks** to find a free one and **update its header** to reflect its allocation size and status, clipping off any excess into a free, smaller block.
- When we **`free`** a block, we update its header to **mark it as free**.
- By maintaining headers, we're **implicitly** maintaining a **list** of free blocks.

Case Study: Implicit Free List Allocator

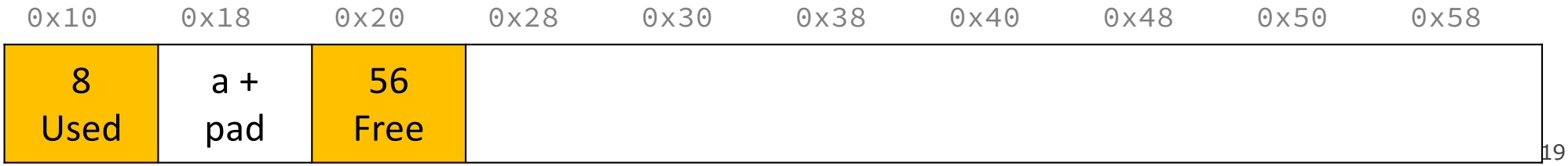
```
void *a = malloc(4);
void *b = malloc(8);
void *c = malloc(4);
free(b);
void *d = malloc(8);
free(a);
void *e = malloc(24);
```



Case Study: 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);
```

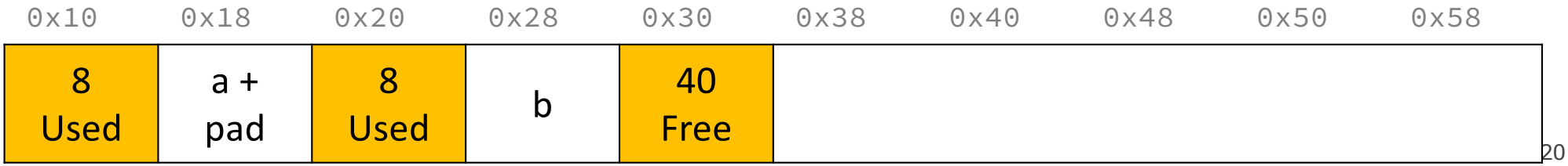
Variable	Value
a	0x18



Case Study: 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);
```

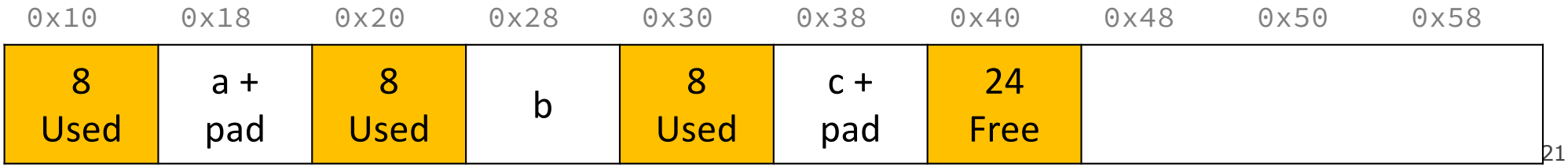
Variable	Value
a	0x18
b	0x28



Case Study: 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);
```

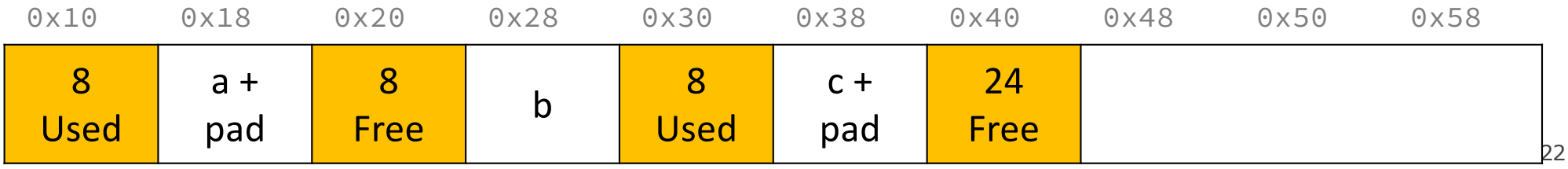
Variable	Value
a	0x18
b	0x28
c	0x38



Case Study: 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);
```

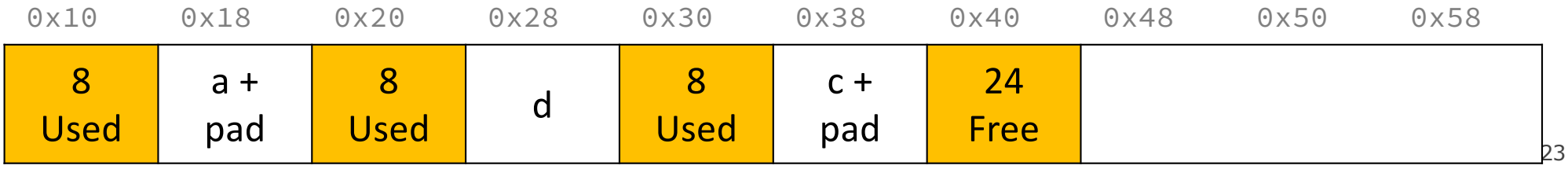
Variable	Value
a	0x18
b	0x28
c	0x38



Case Study: 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);
```

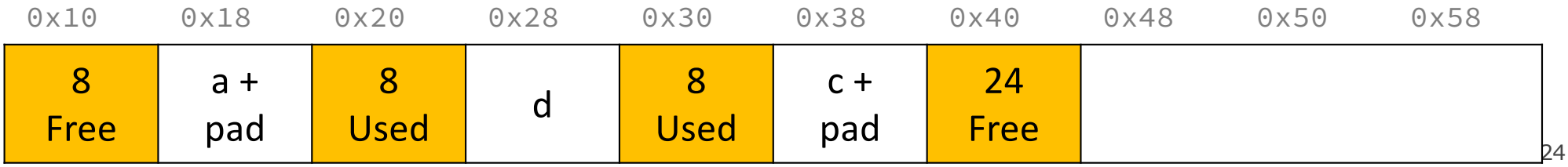
Variable	Value
a	0x18
b	0x28
c	0x38
d	0x28



Case Study: 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);
```

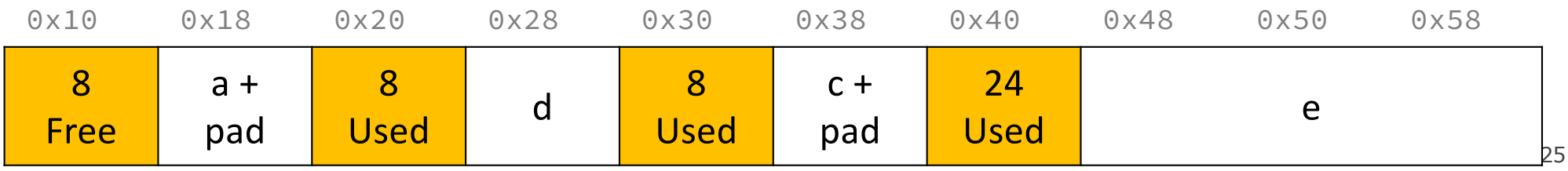
Variable	Value
a	0x18
b	0x28
c	0x38
d	0x28



Case Study: 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



Implicit Allocators: Representing Headers

We'll assume headers are **eight bytes** in size.

How can we store both **size** and **status**—that is, **free** vs. **in use**—in eight bytes?

Naïve Idea: Use four bytes to store the **size** (as an **`int`**) and use the other four bytes to store **allocation status** (also as an **`int`**).

Dilemma: **`malloc`** and **`realloc`** take **`size_t`**s as arguments, not **`int`**s.

This idea won't work.

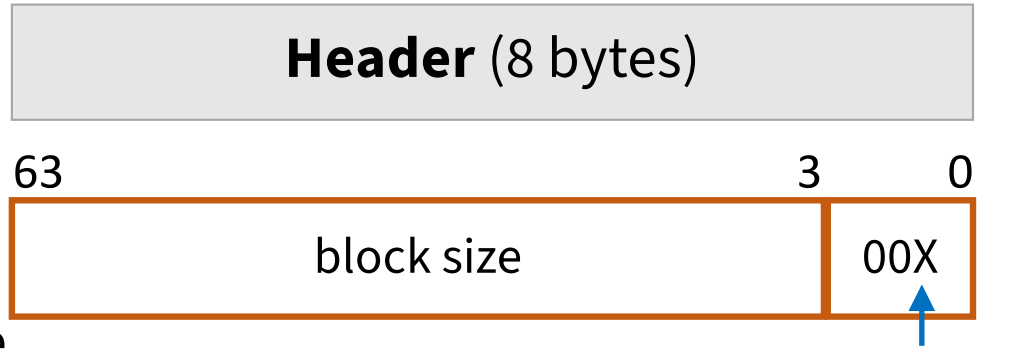
Observation: Since block sizes are **multiples of 8**, the three LSBs are always zero. So, use **all eight header bytes** to store the size.

Key Insight: Since the three LSBs are always zero, we can **co-opt** one of those bits as a **Boolean allocation flag**.

Implicit Allocators: Representing Headers

For **all blocks**:

- Maintain a header that stores **size** and **status** information.
- View sequence of blocks as an **implicit list**—implicit because there aren't any **`next`** or **`prev`** pointers—of **all** nodes, both free and in use.



Keeping track of free blocks:

- **Improves memory utilization** (versus bump allocator)
- **Decreases throughput** (worst case allocation request has O(A + F) time)
- Increases design complexity (but compared to bump, it's worth it)

Implicit Allocators: Trivia

Let's work through the answers to some questions about implicit allocators as we've described them so far. Some of this is useful for **`assign6`**, and some of it is just cool.

Question 1: How do I extract the size of a block **without being confused** by the allocation bit?

Answer 1:

```
size_t get_size(size_t *header) {
   return *header & (~0x7UL);
}
```

Question 2: How do I determine if a block is **free**?

Answer 2:

```
bool is_free(size_t *header) {
   return (*header & 0x1UL) == 0;
} // assumes 1 means in use
```

Question 3: How can I **`toggle`** the allocation bit?

Answer 3:

```
void toggle_free(size_t *header) {
   *header ^= 0x1UL;
}
```

Question 4: Wait, does the size **include the header bytes** or not? Or just the **payload bytes**?

Answer 4: **It's completely up to you!** All that's important is that you're consistent throughout your own implementation.

Question 5: Two of the three trailing bits aren't used for anything. Can they be used for **something**?

Answer 5: Of course! More **sophisticated allocators** often use them to track **allocation statuses of neighboring blocks**. **We don't do this in CS107!, though. This is just a neat-to-know!**

Implicit Allocators: Choosing Blocks

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

First fit: Search from the **beginning each time** and choose the 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**.

What are the pros/cons of each approach?

First fit: **Fast** and **simple** to implement and **stops early** once a block fits. 👍 **Tends to cluster small fragments near the front**, increasing fragmentation and search time over long runs.

Next fit: Avoids rescanning from the beginning, often **improving average search time**. Also simple, but **can skip good fits near the front** and can cause **uneven fragmentation patterns**.

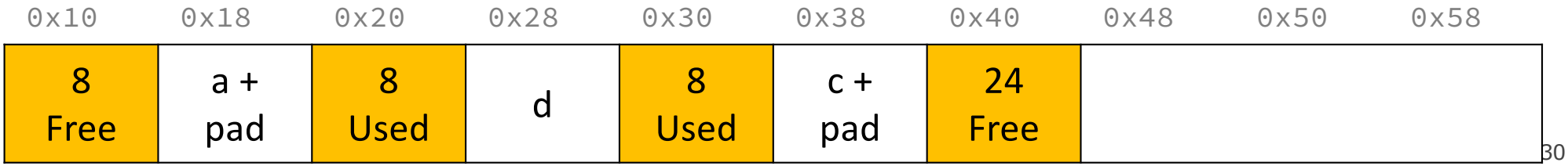
Best fit: Uses memory most tightly by **choosing the smallest fitting block**, **reducing leftover space**. Requires scanning all blocks, **so slower in practice** and can create many **tiny unusable fragments**.

Implicit Allocators: Splitting Policy

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

So far, we have seen that an allocation request often splits a free block into an allocated block and a second free block using the leftover space.

What about edge cases?



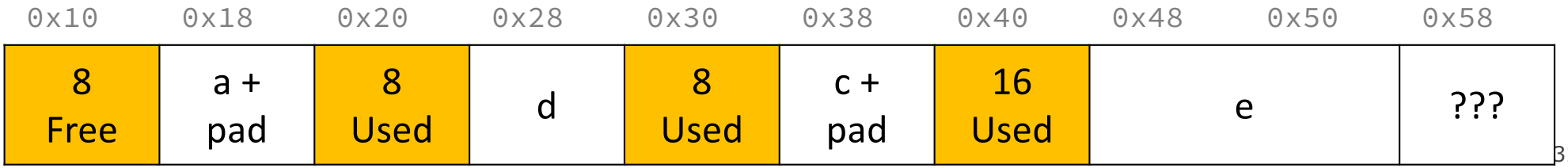
Implicit Allocators: Splitting Policy

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

So far, we have seen that an allocation request often splits a free block into an allocated block and a second free block using the leftover space.

What about edge cases?

Here, that **leftover space** is all of eight bytes. What to do?



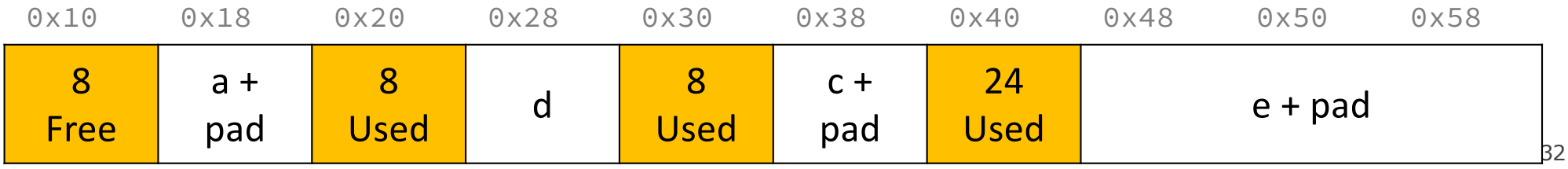
Implicit Allocators: Splitting Policy

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

So far, we have seen that an allocation request often splits a free block into an allocated block and a second free block using the leftover space.

What about edge cases?

One Idea: Give the leftover space to **e** as extra padding.
Pro: It's simple: Pretend the client requested those extra bytes.
Con: Those extra bytes aren't usable for future allocations. This is **internal fragmentation**: unusable bytes within an allocated node.



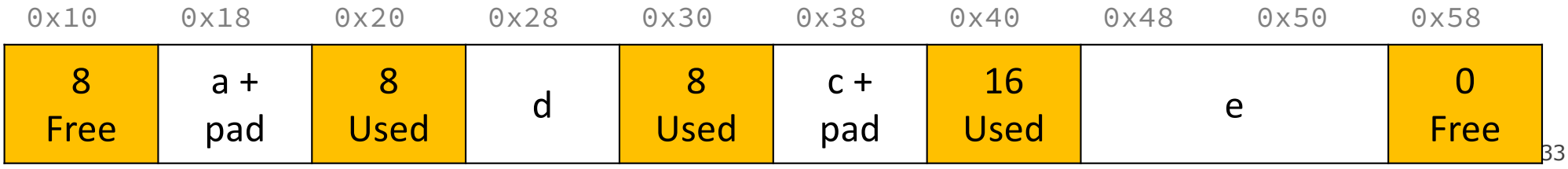
Implicit Allocators: Splitting Policy

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

So far, we have seen that an allocation request often splits a free block into an allocated block and a second free block using the leftover space.

What about edge cases?

Another Idea: Don't do anything special. Just create a free block with zero bytes of payload.
Pro: It's also simple. In fact, it's possible your code will just work without any special checks.
Con: Those bytes aren't usable for future allocations and will likely remain unused for the lifetime of the heap. This is called **external fragmentation**: unusable free space between allocated blocks.



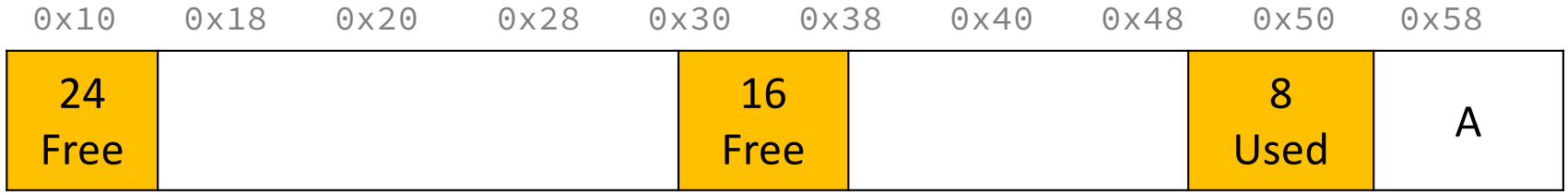
Heap Allocator Goals Revisited

Questions considered this far:

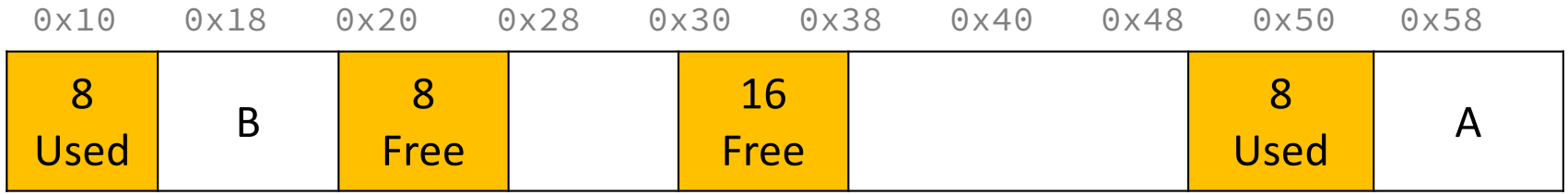
1. How do we keep track of free blocks? **We use headers!**
2. How do we choose an appropriate free block in which to place a newly allocated block? **We iterate through all blocks!**
3. After we place a newly allocated block in some free block, what do we do with the excess? **We try to make the most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

Implicit Allocators: Splitting Policy Etude 1

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?

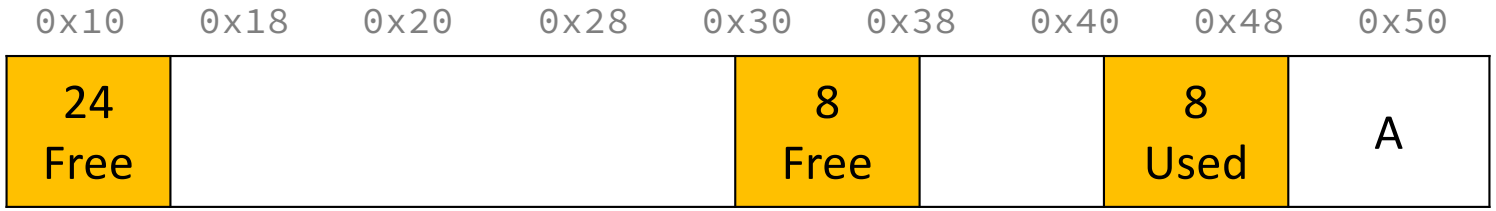



```
void *b = malloc(8);
```

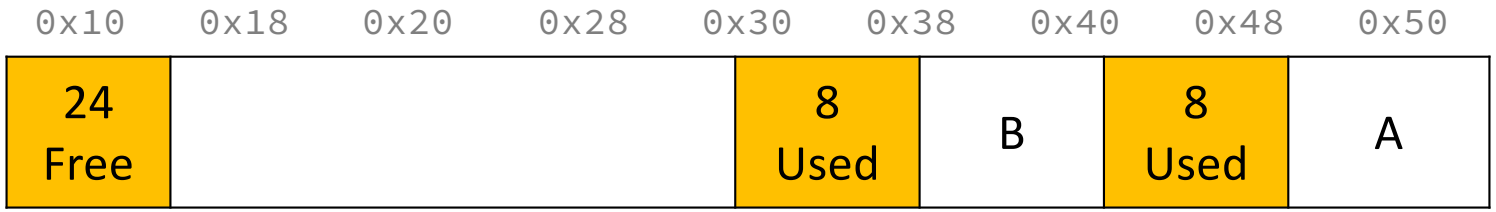


Implicit Allocators: Splitting Policy Etude 2

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

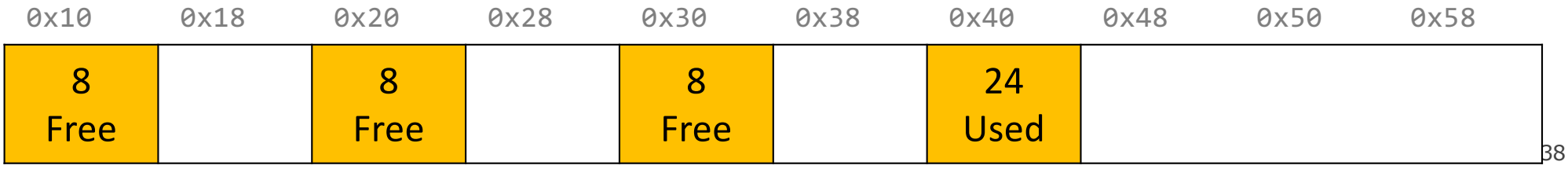
- **Must have** headers that track block information (size, status in-use or free) – you must use the eight-byte header size, storing the status using the free bits.
- **Must allow**, whenever possible, free blocks to be recycled and reused for subsequent **`malloc`** requests
- **Must have** a **`malloc`** implementation that searches the heap for free blocks via its implicit list (i.e., traverses block-by-block).

- **Does not need to** coalesce free blocks.
- **Does not need to** support in-place **`realloc`**.

Looking Forward: Coalescing

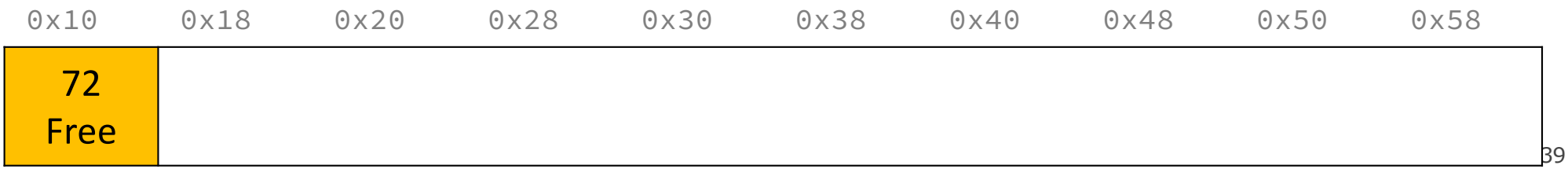
```
void *e = malloc(24);  // returns NULL!
```

You do not need to worry about this problem for the implicit allocator. You will, however, for the explicit allocator (to be discussed on Friday).



Looking Forward: In-Place `realloc`

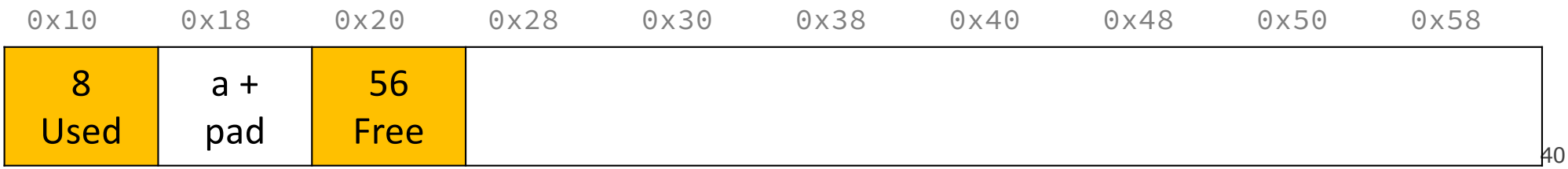
```
void *a = malloc(4);
void *b = realloc(a, 8);
```



Looking Forward: In-Place `realloc`

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

Variable	Value
a	0x18

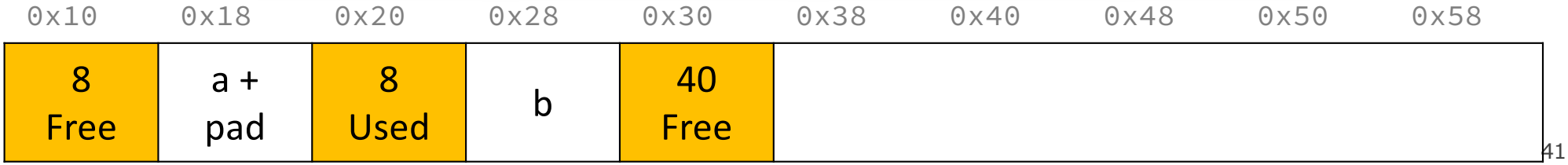


Looking Forward: In-Place `realloc`

```
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 later).



Implicit Allocators: Summary

An implicit allocator is an efficient implementation with reasonable **throughput** and **utilization**.

Can we do better?

1. How might we **avoid searching all blocks** for free blocks to choose from?
2. How can we **merge adjacent free blocks** into larger, more accommodating blocks without impacting throughput?
3. How might we **avoid `memcpy`–ing of data** with each **`realloc`** call? Can we **support the in-place `realloc`** we illustrated in the last few slides?