photo by pixabay user @ulleo

COMP201

Computer Systems & Programming

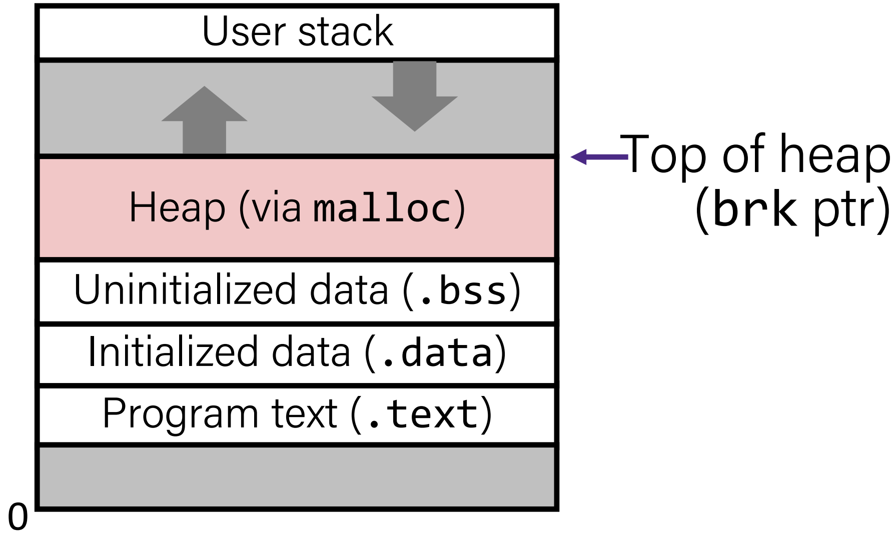
Lecture #26 – More on Managing The Heap

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2023

Recap: Dynamic memory allocation

- Allocator organizes heap as a collection of variable-sized blocks, which are either allocated or free
  - Allocator requests pages in the heap region; virtual memory hardware and OS kernel allocate these pages to the process
  - Application objects are typically smaller than pages, so the allocator manages blocks *within* pages
    - (Larger objects handled too; ignored here)



Recap: Types of heap allocators

- **Explicit allocator:** programmer allocates and frees space
  - Example: `malloc` and `free` in C

- **Implicit allocator:** programmer only allocates space (no free)
  - Example: garbage collection in Java, Caml, and Lisp

Recap: Heap Allocator Functions

```
void *malloc(size_t size);

void free(void *ptr);

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

Recap: Heap Allocator Requirements

A heap allocator must...

1. Handle arbitrary request sequences of allocations and frees
2. Keep track of which memory is allocated and which is available
3. Decide which memory to provide to fulfill an allocation request
4. Immediately respond to requests without delay

Recap: 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!

Recap: Fragmentation

- **Internal Fragmentation:** an allocated block is larger than what is needed (e.g. due to minimum block size)
- **External Fragmentation:** no single block is large enough to satisfy an allocation request, even though enough aggregate free memory is available

Recap: Bump Allocator

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

Recap: Implicit Free List Allocator

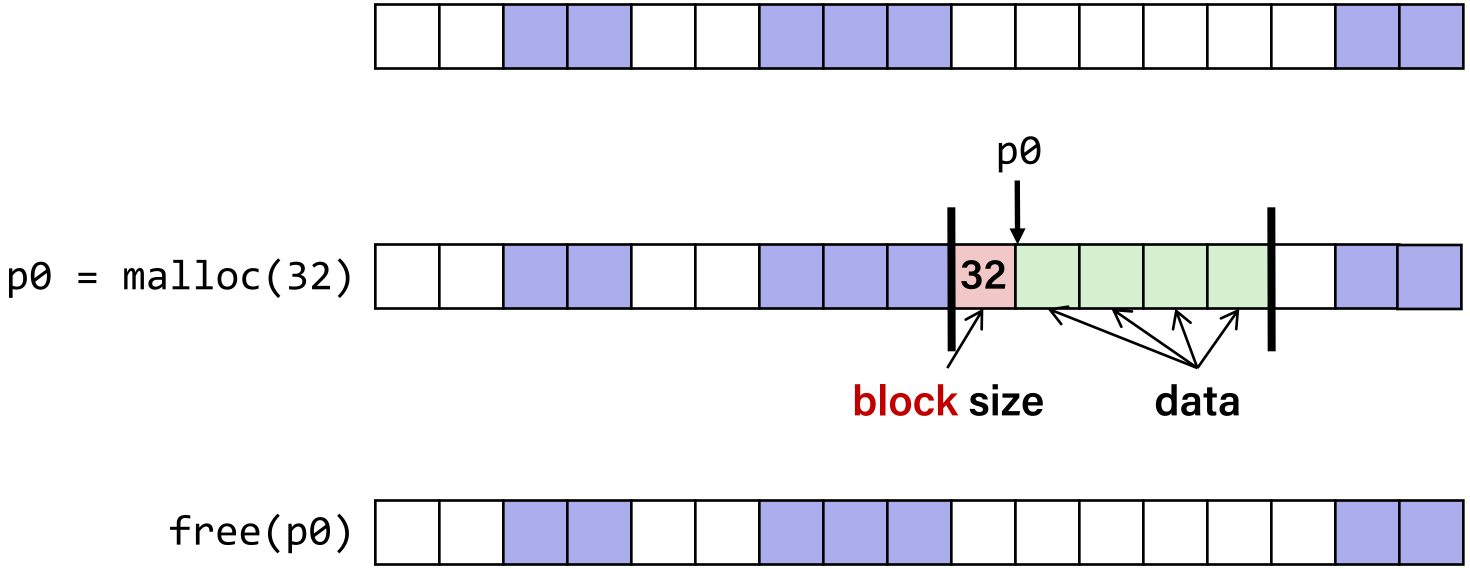
- **Key idea:** in order to reuse blocks, we need a way to track which blocks are allocated and which are free.
- We could store this information in a separate global data structure, but this is inefficient.
- Instead: let's allocate extra space before each block for a **header** storing its payload 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.

Recap: Implicit Free List Allocator

□ = 8-byte word (free)

🟦 = 8-byte word (allocated)

- Standard method
  - Keep the length of a block in the word preceding the data
    - This word is often called the header field or header
  - Requires an extra word for every allocated block



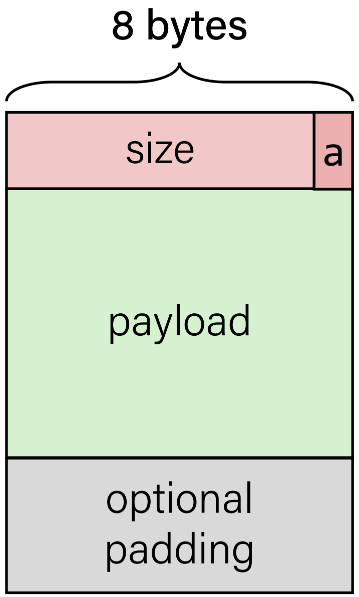
Recap: Representing Headers

e.g. with 8-byte alignment,
possible values for size:

```
00001000 = 8 bytes
00010000 = 16 bytes
00011000 = 24 bytes
...
```

- For each block we need: **`size`**, is-allocated?
  - Could store using two words, but wasteful
- Standard trick
  - If blocks are aligned, some low-order bits of `size` are always 0
  - Use lowest bit as an allocated/free flag (fine as long as aligning to K>1)
  - When reading `size`, must remember to mask out this bit!

Format of allocated and free blocks:



`a = 1`: allocated block
`a = 0:` free block

`size:` block size (in bytes)

`payload:` application data (allocated blocks only)

If x is first word (header):

```
x = size | a;

a = x & 1;

size = x & ~1;
```

Recap: 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?

Recap: 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!**

Recap: 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`?

Plan for Today

- Explicit Free List Allocator
- Garbage Collection

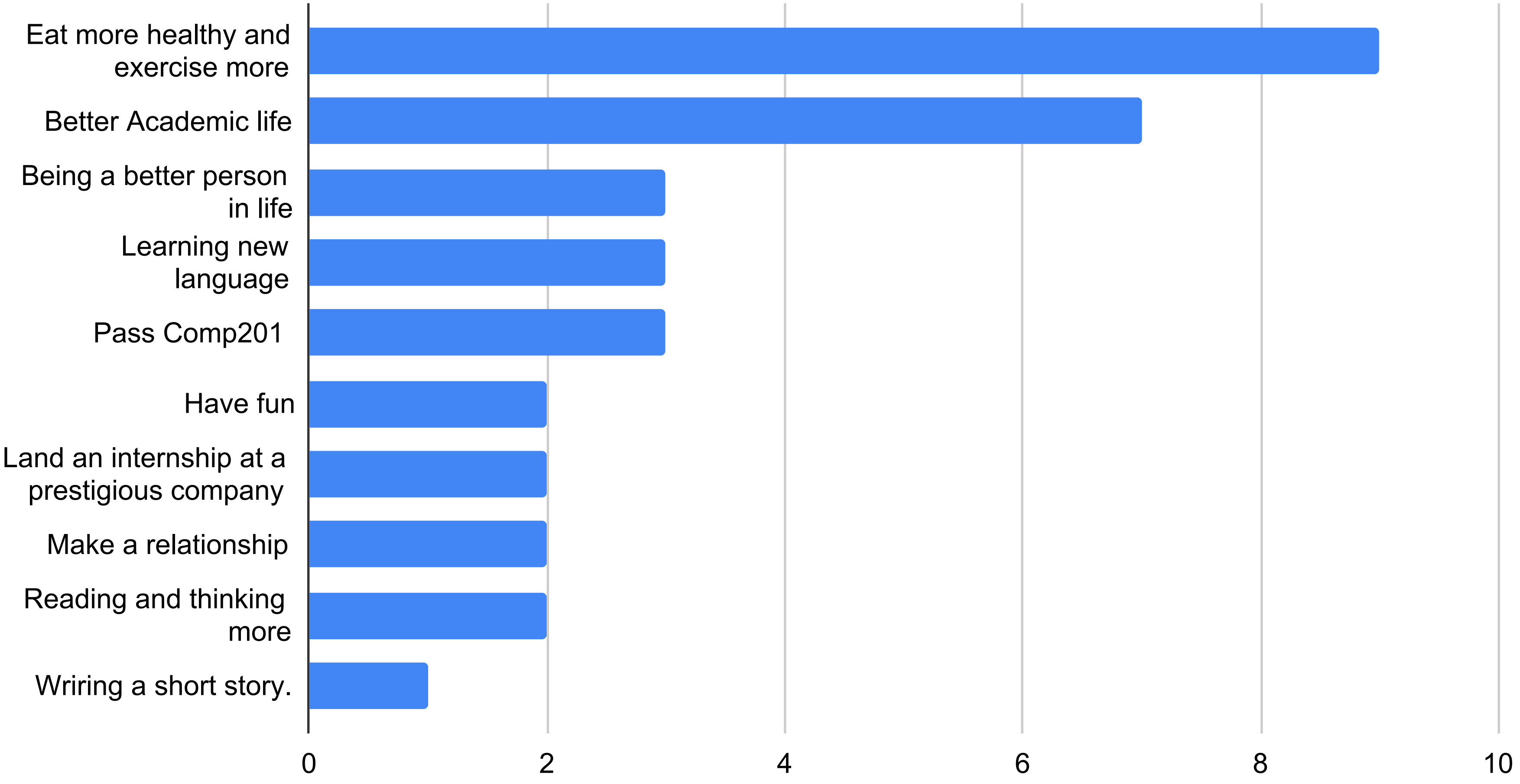
Disclaimer: Slides for this lecture were borrowed from

—Nick Troccoli's Stanford CS107 class

—Ruth Anderson’s UW CSE 351 class

Results of the Previous Attendance Question

What is your New Year’s resolution?



Attendance Question

What is your favorite programming language?

SCAN ME

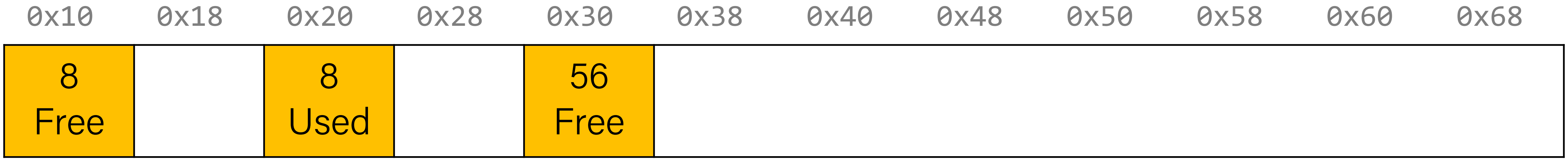
https://forms.gle/UWU9DTJGuJ4DZwsL6

Lecture Plan

- Explicit Free List Allocator
- Garbage Collection

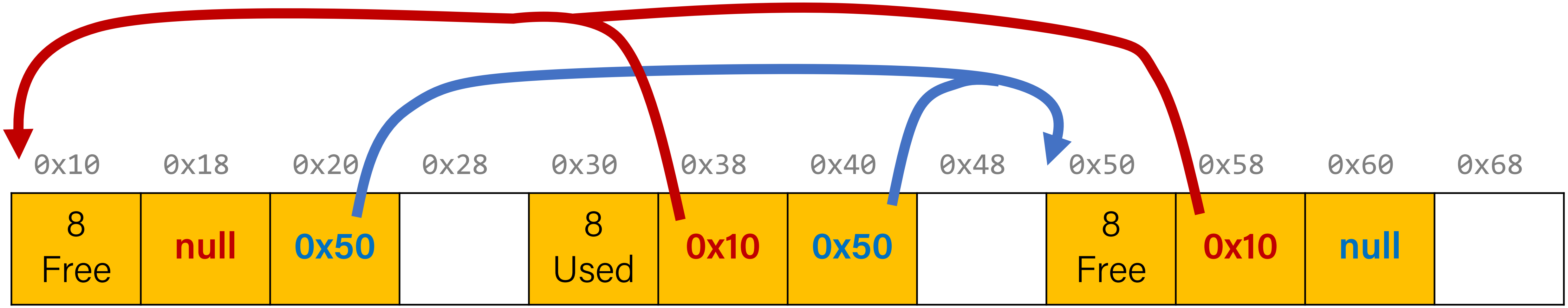
Can We Do Better?

- 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.

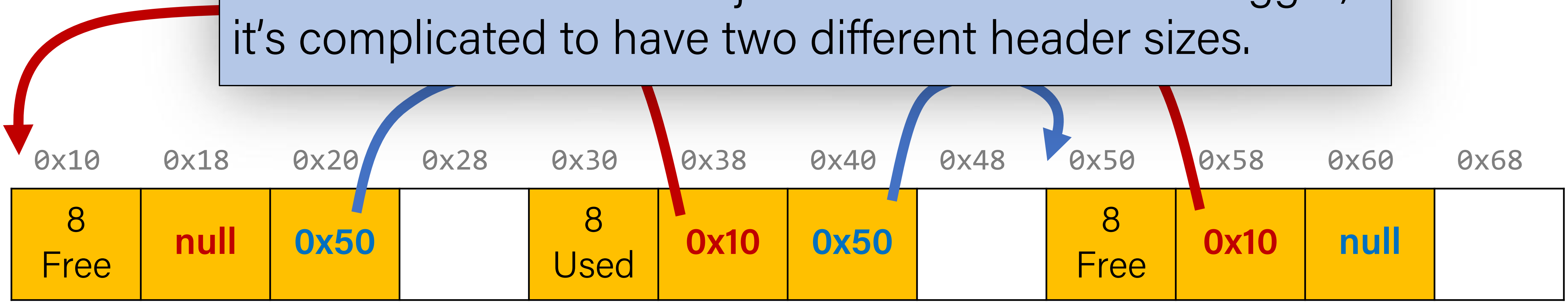


0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60	0x68
8 Free	null	0x50		8 Used	0x10	0x50		8 Free	0x10	null	

Can We Do Better?

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

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



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

Can We Do Better?

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

Can We Do Better?

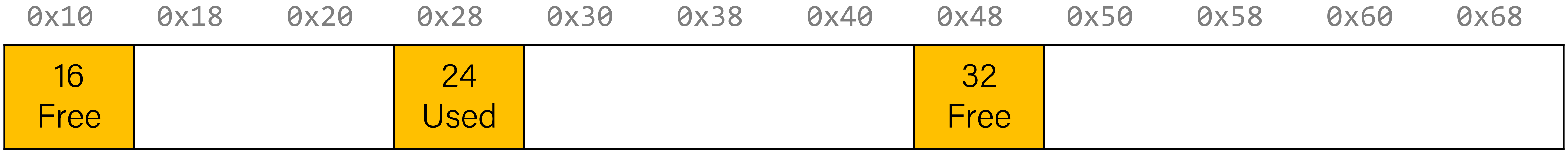
- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the previous free block and a pointer to the next free block. *This is inefficient / complicated.*
- **Where can we put these pointers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



0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48 0x50 0x58 0x60 0x68

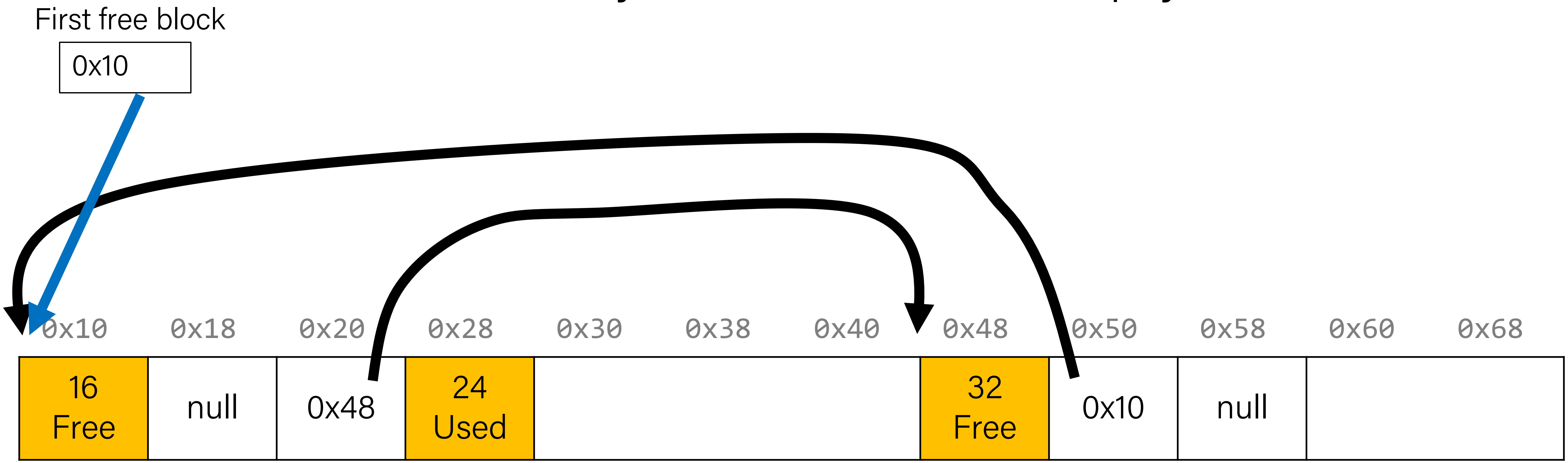
16 Free

24 Used

32 Free

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!



First free block

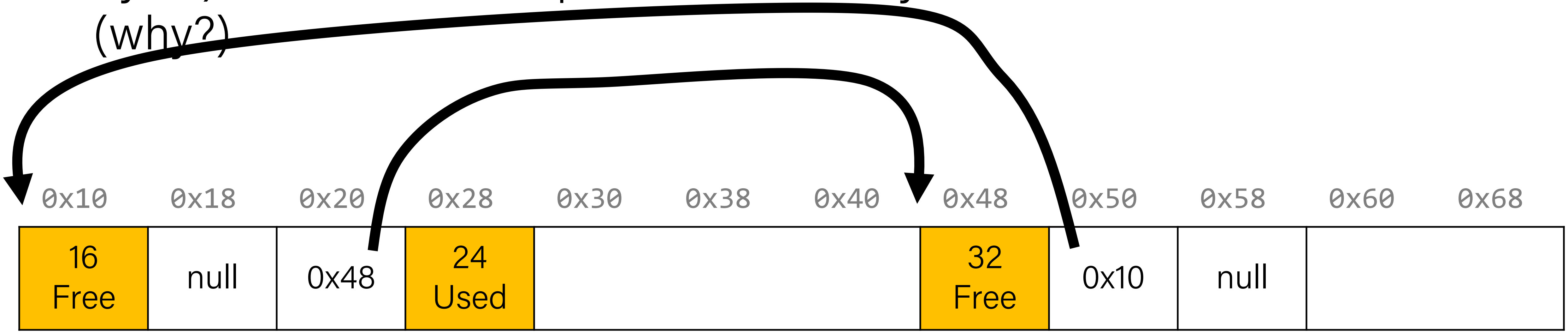
0x10

0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48 0x50 0x58 0x60 0x68

16 Free | null | 0x48 | 24 Used | | 32 Free | 0x10 | null |

Can We Do Better?

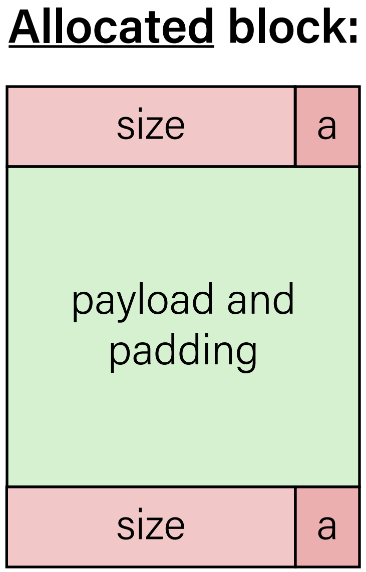
- **Key Insight:** the payloads of the free blocks aren't being used, because they're free.
- **Idea:** since we only need to store these pointers for free blocks, let's store them in the first 16 bytes of each free block's payload!
- This means each payload must be big enough to store 2 pointers (16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



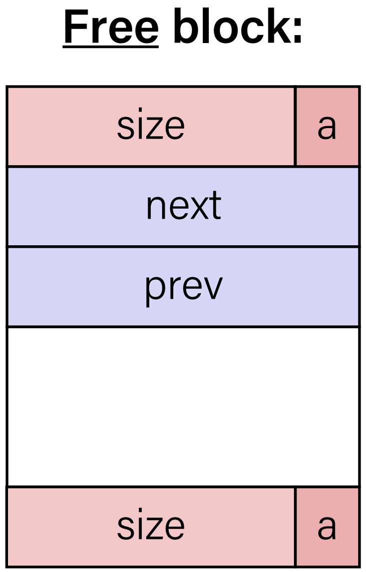
0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48 0x50 0x58 0x60 0x68

16 Free | null | 0x48 | 24 Used | | 32 Free | 0x10 | null |

Explicit Free Lists



(same as implicit free list)



- Use list(s) of free blocks, rather than implicit list of all blocks
  - The "next" free block could be anywhere in the heap
    - So we need to store next/previous pointers, not just sizes
  - Since we only track free blocks, so we can use "payload" for pointers
  - Still need boundary tags (header/footer) for coalescing

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)

Better memory util, Linear free

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

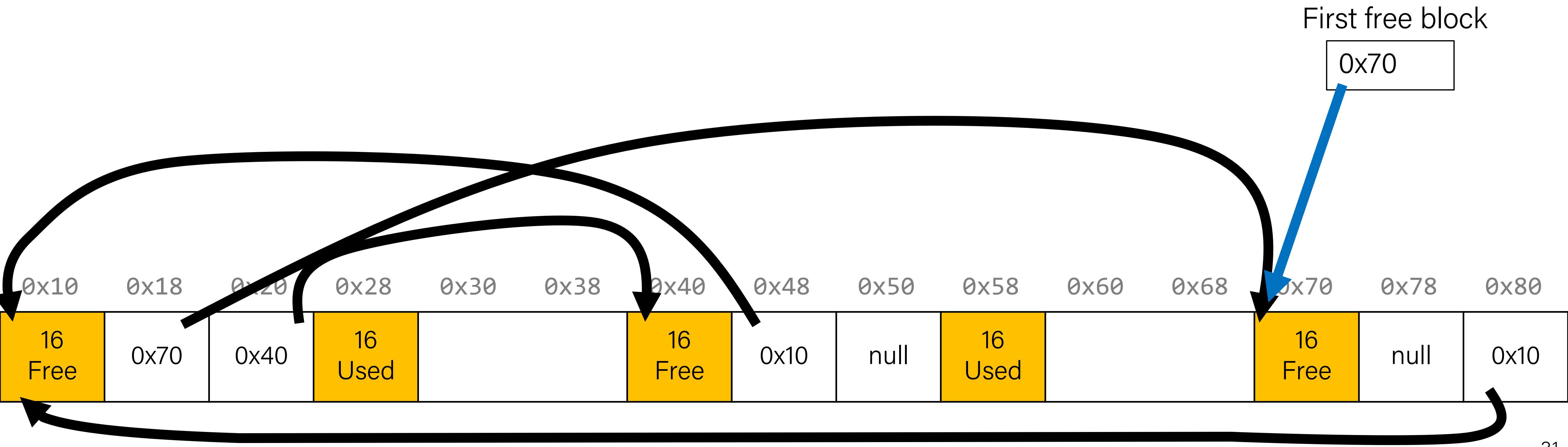
Constant free (push recent block onto stack)

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

(more at end of lecture)

Explicit Free List: List Design

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



Implicit vs. Explicit: So Far

Implicit Free List

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

Explicit Free List

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

Revisiting Our Goals

Can we do better?

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

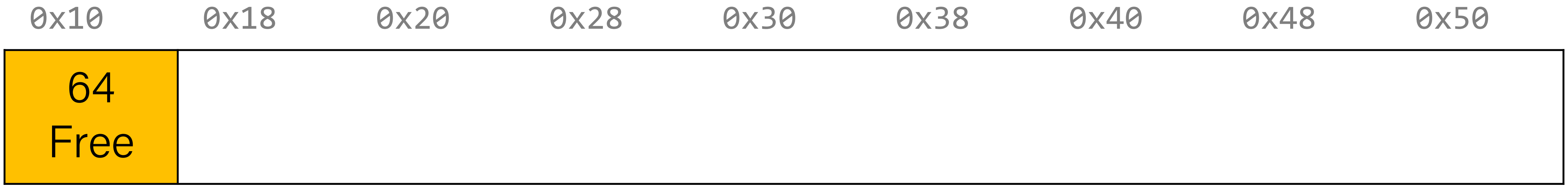
Revisiting Our Goals

Can we do better?

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

Coalescing

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

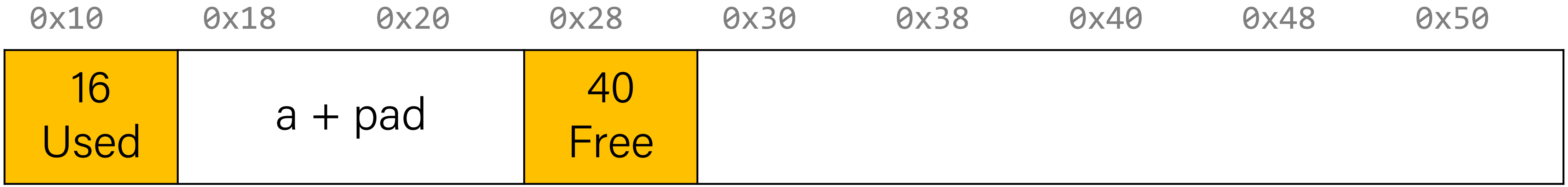


0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48 0x50

64
Free

Coalescing

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



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

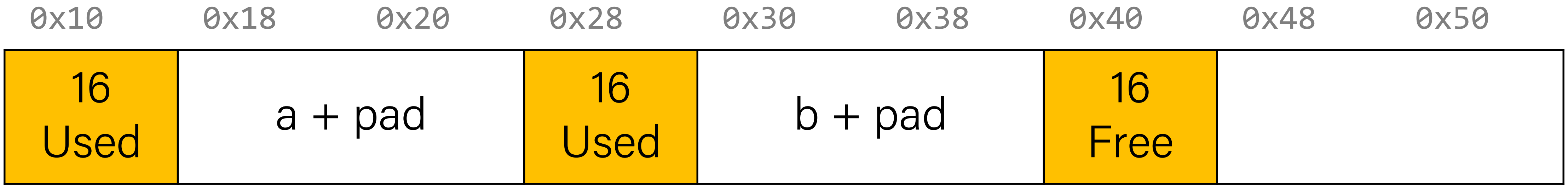
16
Used

a + pad

40
Free

Coalescing

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



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

16 Used

a + pad

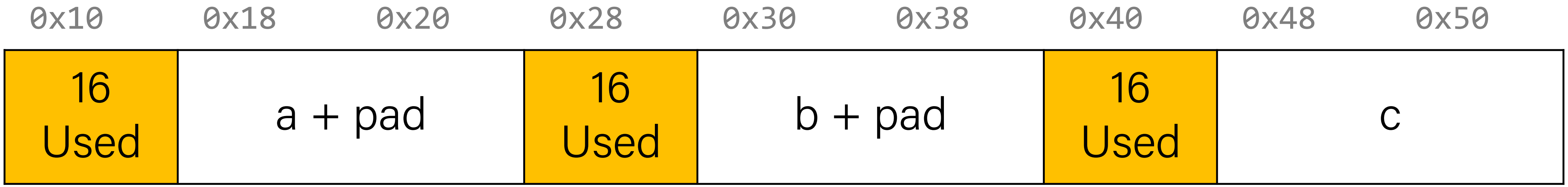
16 Used

b + pad

16 Free

Coalescing

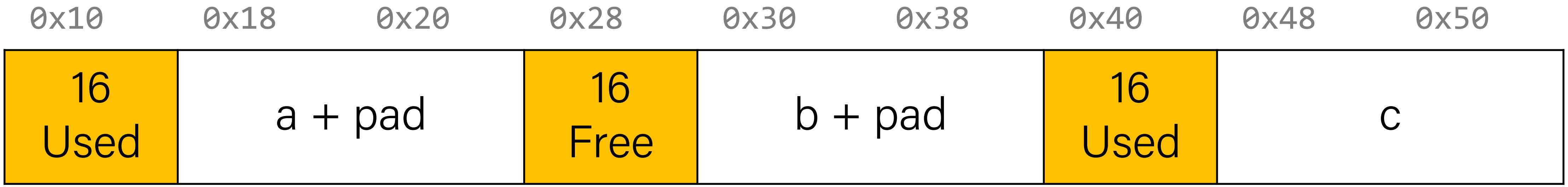
```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```



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

Coalescing

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

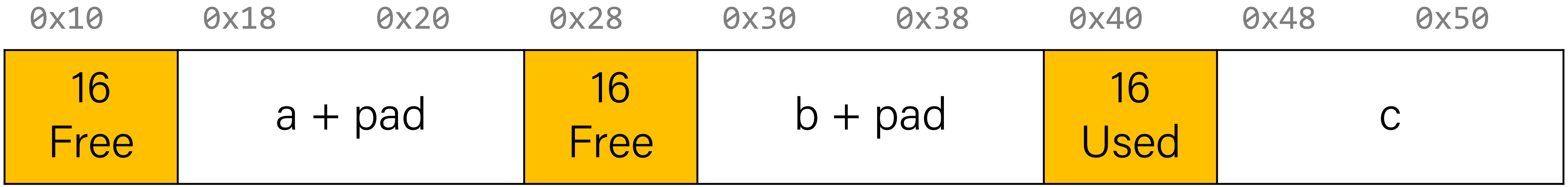


0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48 0x50

16 Used | a + pad | 16 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);
```



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

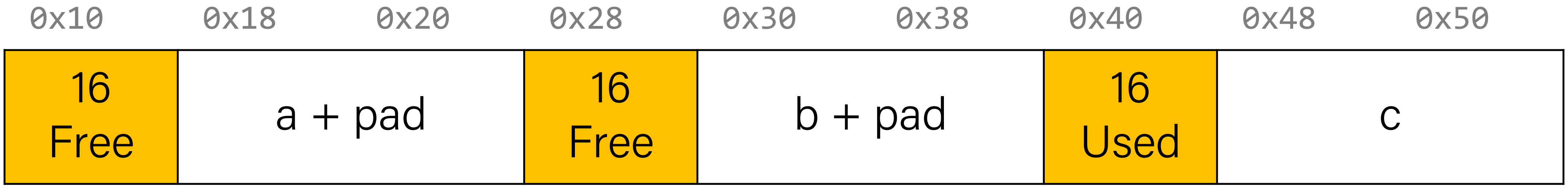
16
Free | a + pad | 16
Free | b + pad | 16
Used | c

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?



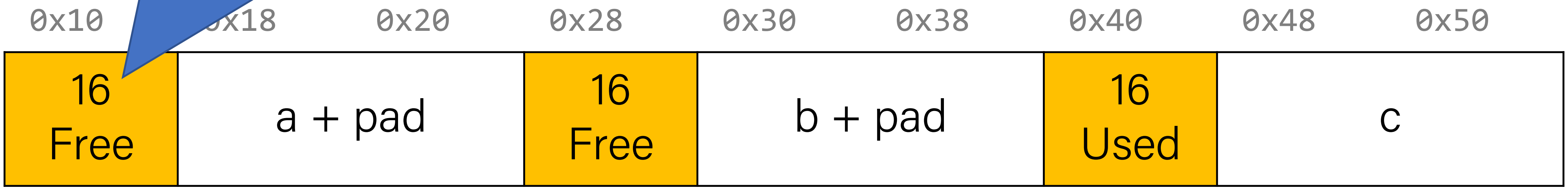
0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48 0x50

16 Free | a + pad | 16 Free | b + pad | 16 Used | c

Coalescing

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

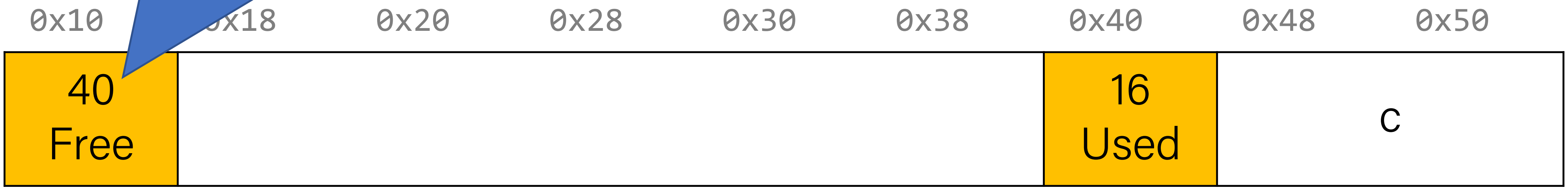
Hey, look! I have a free neighbor. Let's be friends! ☺



Coalescing

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

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

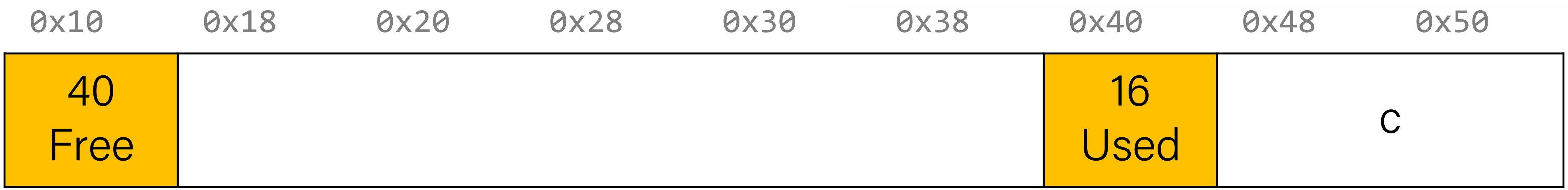


Coalescing

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

The process of combining adjacent free blocks is called coalescing.

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



Revisiting Our Goals

Can we do better?

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

Revisiting Our Goals

Can we do better?

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

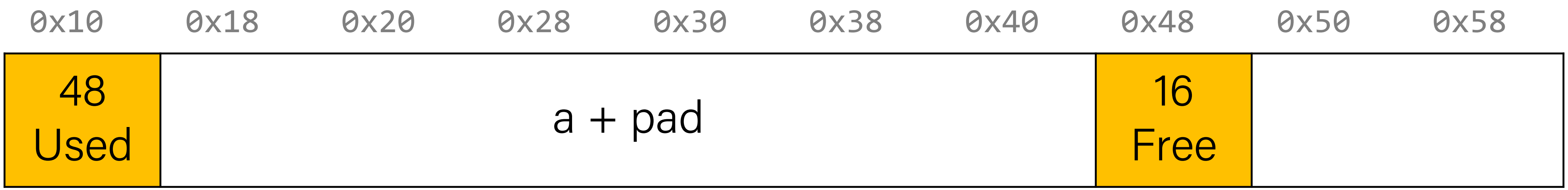
`realloc`

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

realloc: Growing In Place

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

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



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

48
Used

a + pad

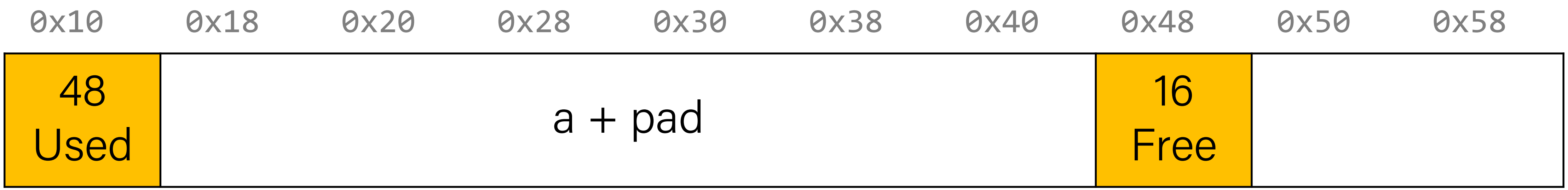
16
Free

realloc: Growing In Place

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

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

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

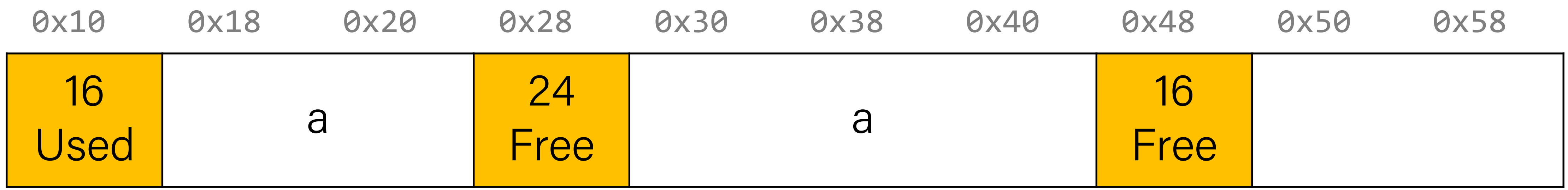


realloc: Growing In Place

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

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

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



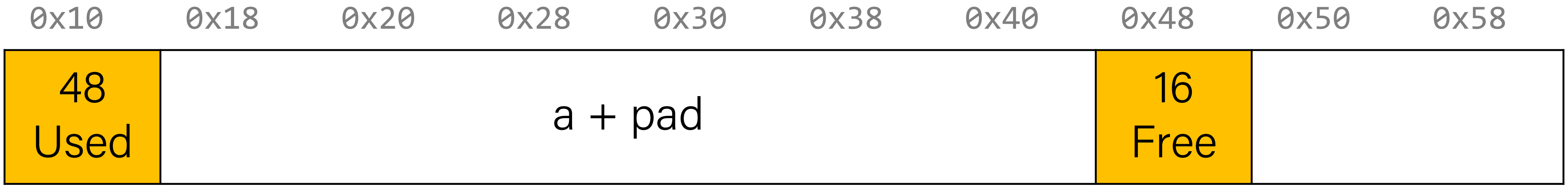
0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48 0x50 0x58

16 Used | a | 24 Free | a | 16 Free |

realloc: Growing In Place

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

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



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

48
Used

a + pad

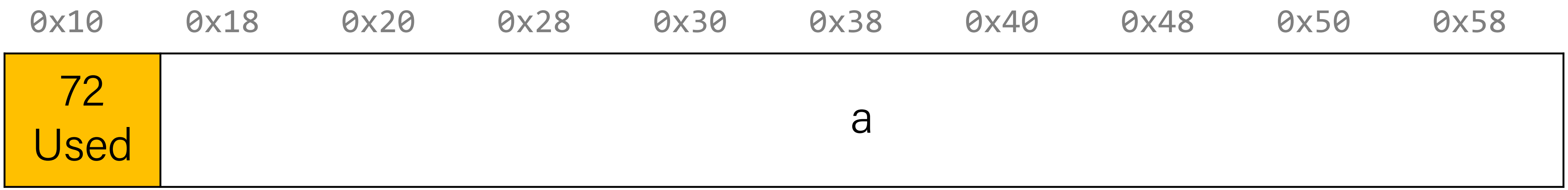
16
Free

realloc: Growing In Place

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



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

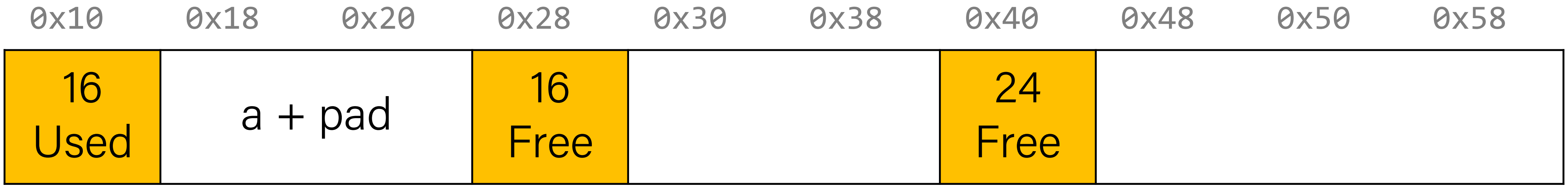
72
Used

a

`realloc`: Growing In Place

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

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



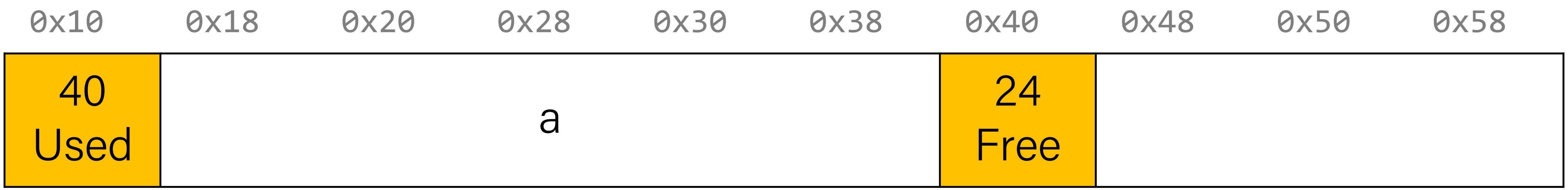
0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48 0x50 0x58

16 Used | a + pad | 16 Free | | 24 Free |

realloc: Growing In Place

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

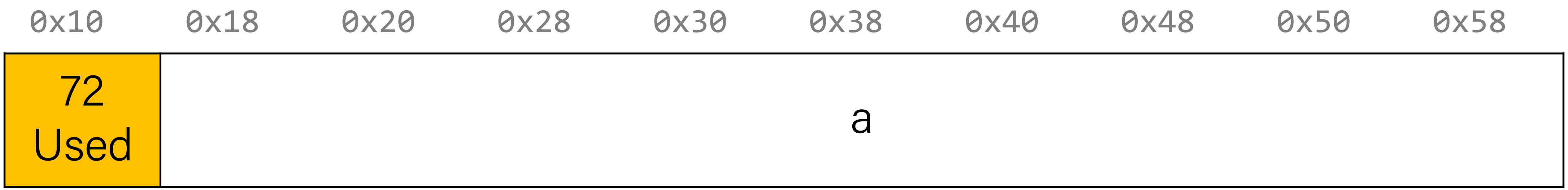
Here, you should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.



realloc: Growing In Place

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

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



`realloc`

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

Practice 3

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

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

```
free(B);
```

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

Practice 4

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

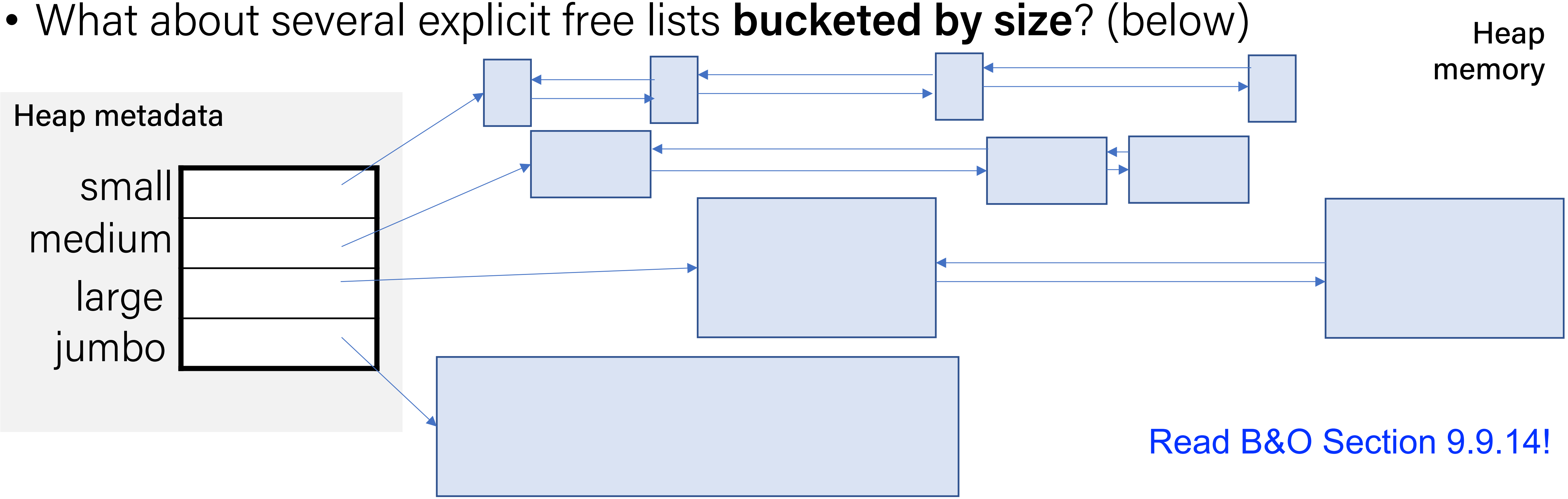
[16 byte payload, allocated for A] [32 byte payload, free] [16 byte payload, allocated for B]

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

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

Going beyond: Explicit list w/size buckets

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



Read B&O Section 9.9.14!

More Info on Allocators

- D. Knuth, "*The Art of Computer Programming*", 2nd edition, Addison Wesley, 1973
    - The classic reference on dynamic storage allocation

- Wilson et al, "*Dynamic Storage Allocation: A Survey and Critical Review*", Proc. 1995 Int'l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.
    - Comprehensive survey
    - Available from CS:APP student site (csapp.cs.cmu.edu)

Wouldn't it be nice…

- If we never had to free memory?
- Do you free objects in Java?
  - Reminder: *implicit* allocator

Lecture plan

- Explicit Free List Allocator
- Garbage Collection

Garbage Collection (GC) (Automatic Memory Management)

- **Garbage collection:** automatic reclamation of heap-allocated storage – application never explicitly frees memory

```
void foo() {
   int* p = (int*) malloc(128);
   return;  /* p block is now garbage! */
}
```

- Common in implementations of functional languages, scripting languages, and modern object oriented languages:
  - Lisp, Racket, Erlang, ML, Haskell, Scala, Java, C#, Perl, Ruby, Python, Lua, JavaScript, Dart, Mathematica, MATLAB, many more...

- Variants (“conservative” garbage collectors) exist for C and C++
  - However, cannot necessarily collect all garbage

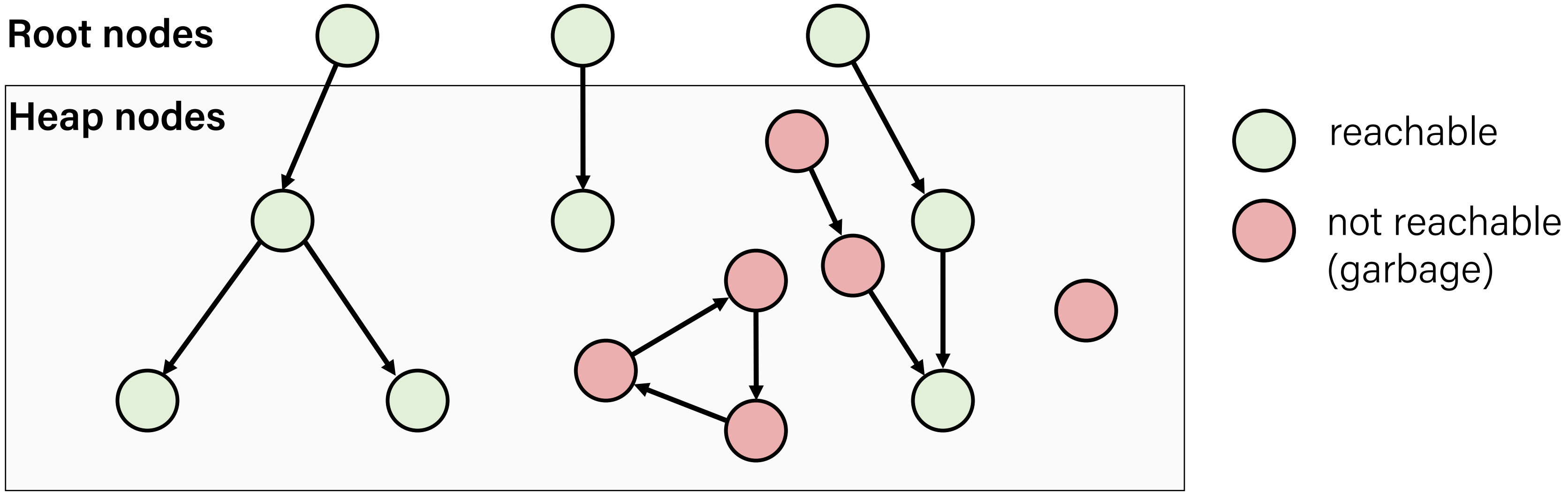
Garbage Collection

- How does the memory allocator know when memory can be freed?
  - In general, we cannot know what is going to be used in the future since it depends on conditionals
  - **But, we can tell that certain blocks cannot be used if they are unreachable (via pointers in registers/stack/globals)**

- Memory allocator needs to know what is a pointer and what is not – how can it do this?
  - Sometimes with help from the compiler

Memory as a Graph

- We view memory as a directed graph
  - Each allocated heap block is a node in the graph
  - Each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called ***root*** nodes (e.g. registers, stack locations, global variables)



A node (block) is reachable if there is a path from any root to that node
Non-reachable nodes are garbage (cannot be needed by the application)

Garbage Collection

- Dynamic memory allocator can free blocks if there are <u>no pointers to them</u>

- How can it know what is a pointer and what is not?

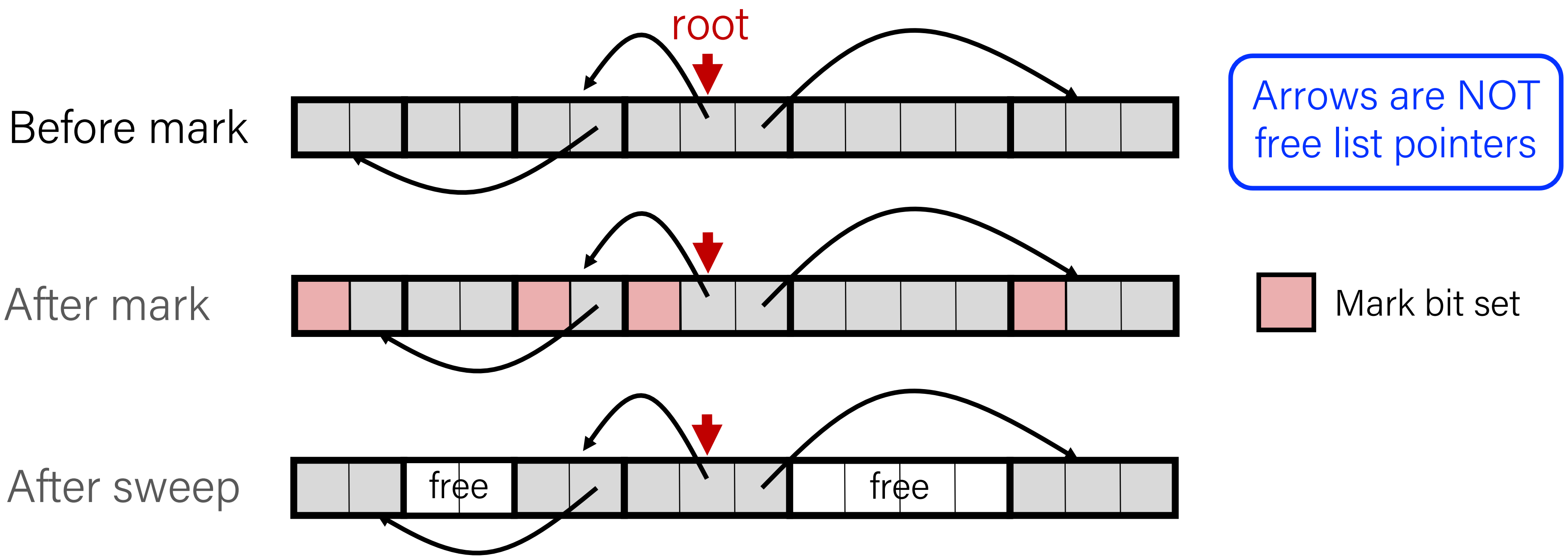
- We'll make some assumptions about pointers:
  - Memory allocator can distinguish pointers from non-pointers
  - All pointers point to the start of a block in the heap
  - Application cannot hide pointers
    (*e.g.* by coercing them to a `long`, and then back again)

Classical GC Algorithms

- **Mark-and-sweep collection** (McCarthy, 1960)
  - Does not move blocks (unless you also “compact”)
- Reference counting (Collins, 1960)
  - Does not move blocks (not discussed)
- Copying collection (Minsky, 1963)
  - Moves blocks (not discussed)
- Generational Collectors (Lieberman and Hewitt, 1983)
  - Most allocations become garbage very soon, so focus reclamation work on zones of memory recently allocated.
- For more information:
  - Jones, Hosking, and Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*, CRC Press, 2012.
  - Jones and Lin, *Garbage Collection: Algorithms for Automatic Dynamic Memory*, John Wiley & Sons, 1996.

Mark and Sweep Collecting

- Can build on top of `malloc/free` package
  - Allocate using `malloc` until you "run out of space"
- When out of space:
  - Use extra mark bit in the header of each block
  - **Mark:** Start at roots and set mark bit on each reachable block
  - **Sweep:** Scan all blocks and free blocks that are not marked



Arrows are NOT free list pointers

Assumptions For a Simple Implementation

- Application can use functions to allocate memory:
  - `b=malloc(…)` returns pointer, `b`, to new block with all locations cleared
  - `b[i]` read location `i` of block `b` into register
  - `b[i]=v` write `v` into location `i` of block `b`

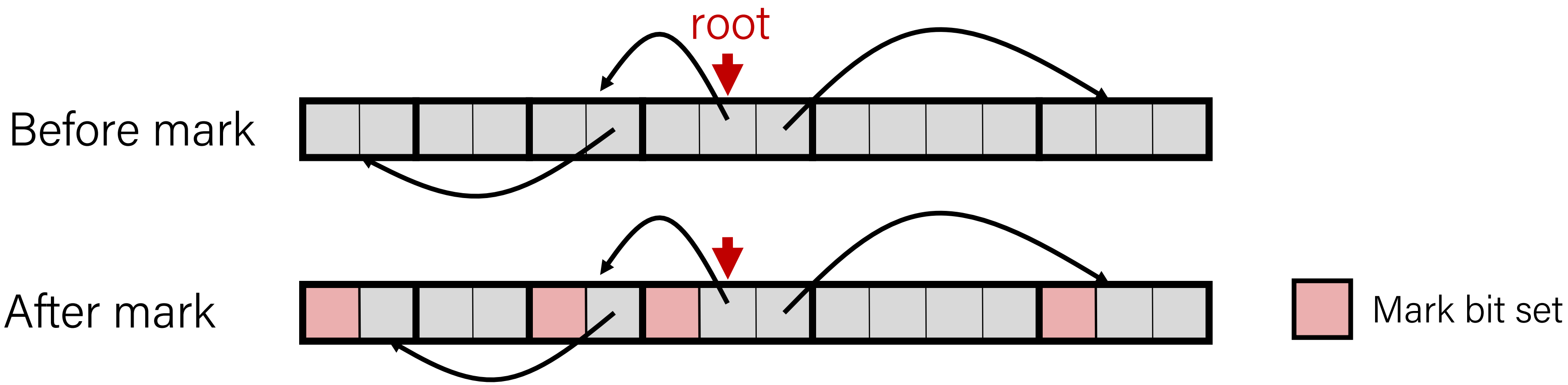
- Each block will have a header word (accessed at `b[-1]`)

- Functions used by the garbage collector:
  - `is_ptr(p)` determines whether `p` is a pointer to a block
  - `length(p)` returns length of block pointed to by `p`, not including header
  - `get_roots()` returns all the roots

Mark

- Mark using depth-first traversal of the memory graph

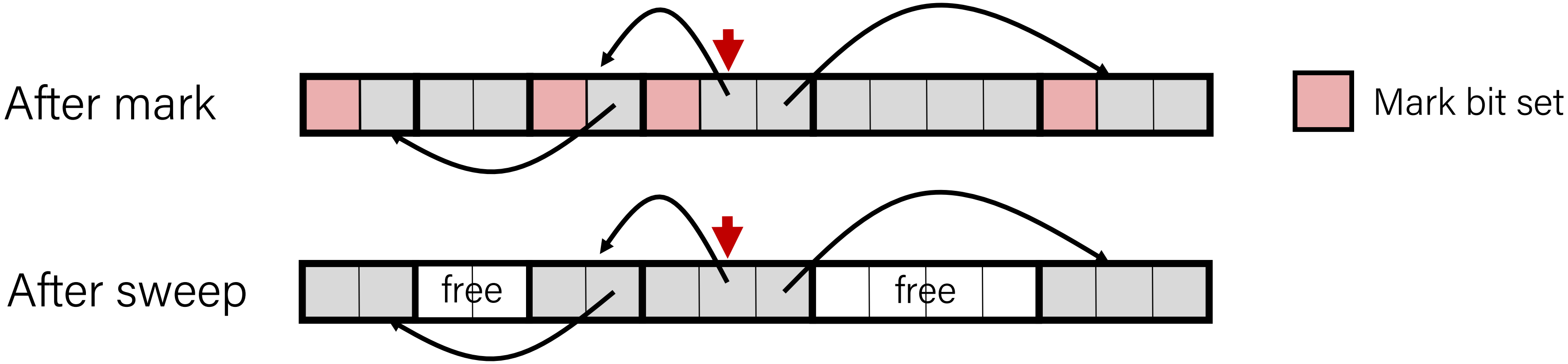
```
ptr mark(ptr p) {                           // p: some word in a heap block
   if (!is_ptr(p))    return;               // do nothing if not pointer
   if (markBitSet(p)) return;               // check if already marked
   setMarkBit(p);                           // set the mark bit
   for (i=0; i<length(p); i++)              // recursively call mark on
      mark(p[i]);                           //    all words in the block
   return;
}
```



Sweep

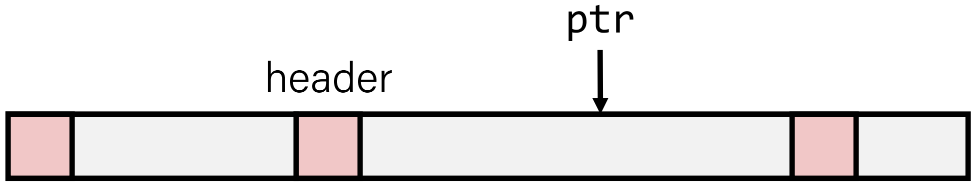
- Sweep using sizes in headers

```
ptr sweep(ptr p, ptr end) {       // ptrs to start & end of heap
   while (p < end) {              // while not at end of heap
      if (markBitSet(p))          // check if block is marked
         clearMarkBit(p);         // if so, reset mark bit
      else if (allocateBitSet(p)) // if not marked, but allocated
         free(p);                 // free the block
      p += length(p);             // adjust pointer to next block
   }
}
```



Conservative Mark & Sweep in C

- Would mark & sweep work in C?
  - `is_ptr` determines if a word is a pointer by checking if it points to an allocated block of memory
  - But in C, pointers can point into the middle of allocated blocks (not so in Java)
    - Makes it tricky to find all allocated blocks in mark phase



  - There are ways to solve/avoid this problem in C, but the resulting garbage collector is conservative:
    - Every reachable node correctly identified as reachable, but some unreachable nodes might be incorrectly marked as reachable
  - In Java, all pointers (*i.e.* references) point to the starting address of an object structure – the start of an allocated block

Recap

- Explicit Free List Allocator
- Garbage Collection

Next time: *Wrapping up*