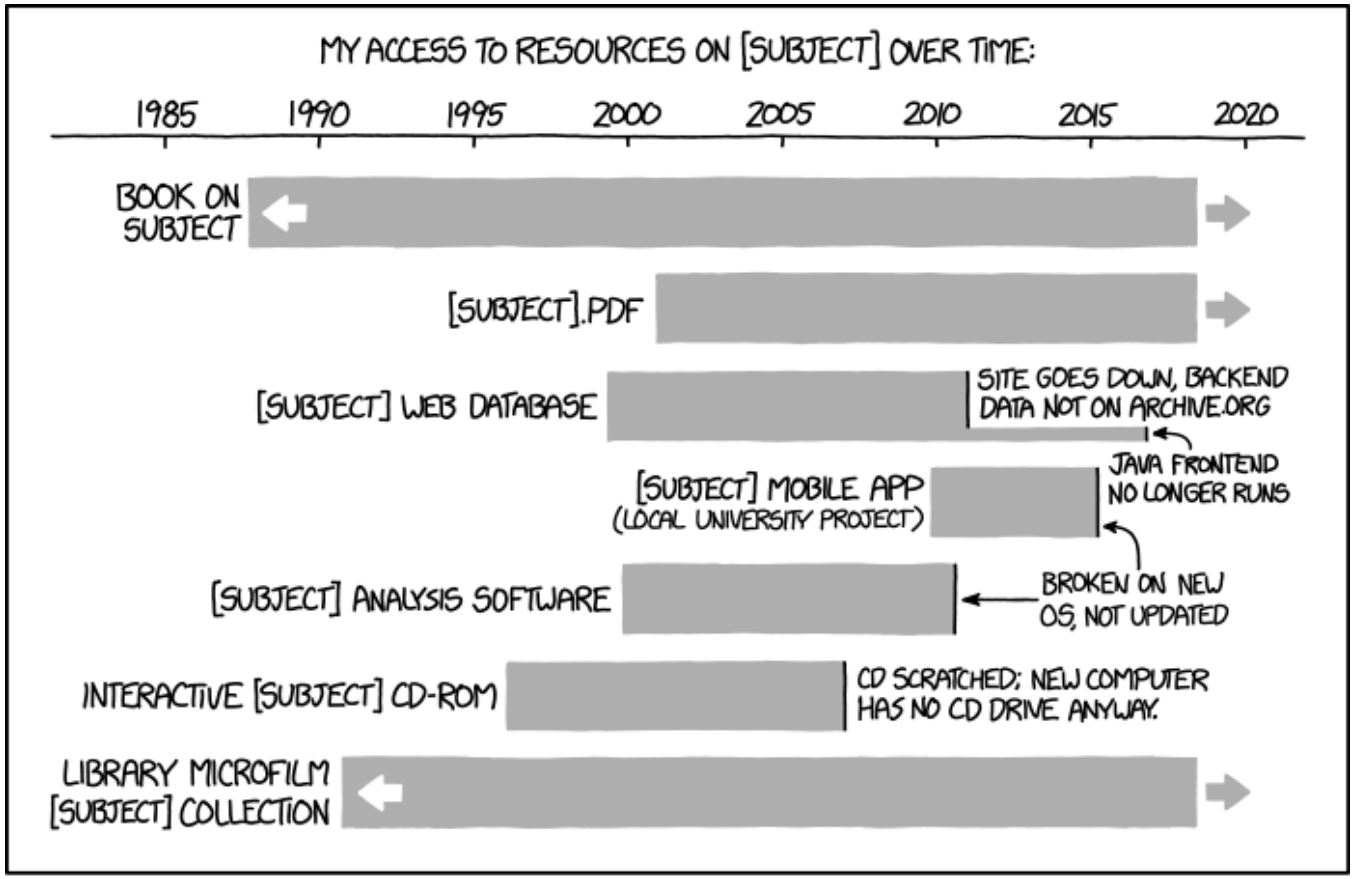
Memory Allocation II

CSE 351 Summer 2020

Instructor:
Porter Jones

Teaching Assistants:
Amy Xu
Callum Walker
Sam Wolfson
Tim Mandzyuk



IT'S UNSETTLING TO REALIZE HOW QUICKLY DIGITAL RESOURCES CAN DISAPPEAR WITHOUT ONGOING WORK TO MAINTAIN THEM.

http://xkcd.com/1909/

End of Quarter Approaching!

- You've done a ton of work and learning in a condensed schedule, be proud of that!

- End of the quarter can get hectic (especially in the summer) balancing final assignments/projects/exams/etc.
  - Could help to take 15 minutes to take a deep breath and organize your last week of the quarter.

- Please start Lab 5/Unit Summary 3 early so you can get them out of the way and focus on other classes!
  - Also allows you to make use of office hours and the message board if you have questions or get stuck
  - Let the course staff know if situations arise that are making it difficult to complete your work, we want to support you the best that we can!

Administrivia

- Questions doc: https://tinyurl.com/CSE351-8-14

- hw19 is optional
  - Can complete it at any point before the quarter ends
  - Practice with virtual memory concepts
- hw21 due Monday (8/17) – 10:30am
- hw22 due Wednesday (8/19) – 10:30am
  - Helpful for Lab 5!
- hw23 due Monday (8/24) – 10:30am
  - Won't cover material until Mon/Wed of next week

Administrivia

- Lab 5 due last day of quarter (Friday 8/21)
  - ***Cutoff is Saturday 8/22 @11:59pm (only one late day can be used!)***
  - The most significant amount of C programming you will do in this class – combines lots of topics from this class: pointers, bit manipulation, structs, examining memory
  - Understanding the concepts *first* and efficient *debugging* will save you lots of time
  - Can be difficult to debug so please start early and use OH
  - Light style grading
  - hw22 will help get you started!

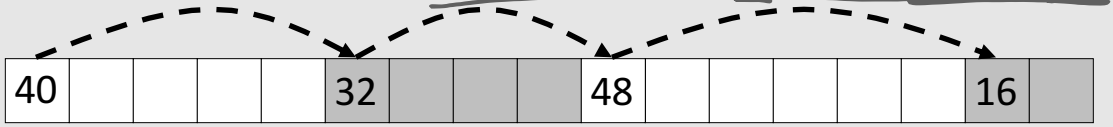
- Unit Summary 3 due last day of quarter (Friday 8/21)
  - ***Cutoff is Saturday 8/22 @11:59pm (only one late day can be used!)***

Keeping Track of Free Blocks

= 8-byte word (free)

= 8-byte word (allocated)

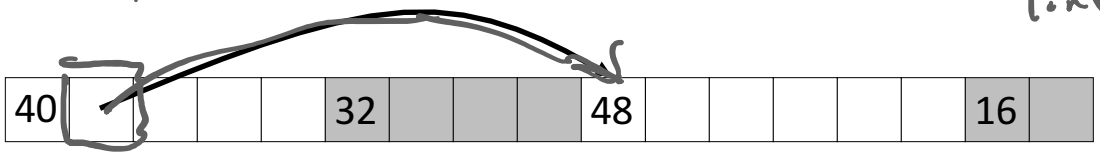
1) *Implicit free list* using length – links all blocks using math
   - No actual pointers, and must check each block if allocated or free



add size to get to next block

2) *Explicit free list* among only the free blocks, using pointers

linked list of free blocks



3) *Segregated free list*
   - Different free lists for different size "classes"

4) *Blocks sorted by size*
   - Can use a balanced binary tree (*e.g.* red-black tree) with pointers within each free block, and the length used as a key

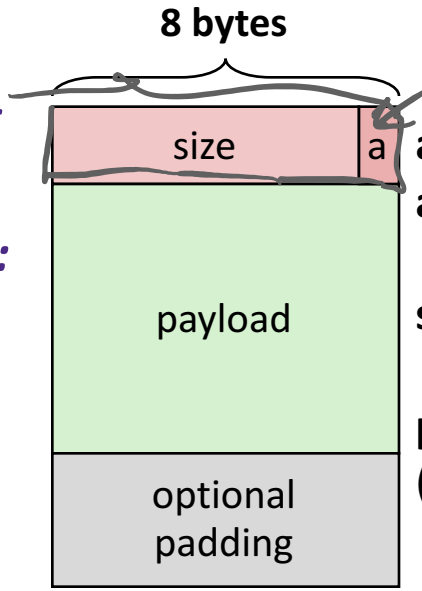
Implicit Free Lists

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):

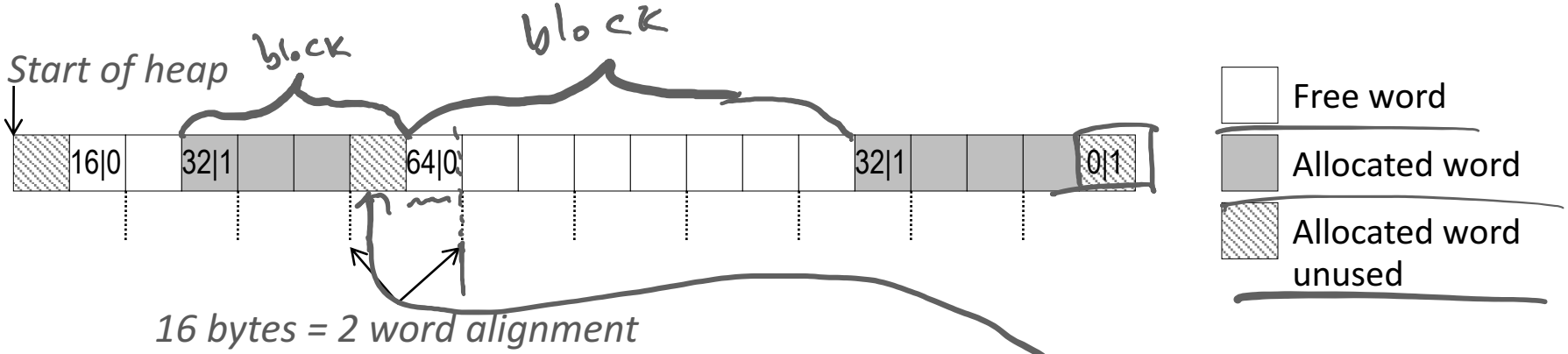
```
x = size | a;

a = x & 1;

size = x & ~1;
```

Implicit Free List Example

- Each block begins with header (size in bytes and allocated bit)
- Sequence of blocks in heap (`size|allocated`):
  16|0, 32|1, 64|0, 32|1



- 16-byte alignment for *payload*
  - May require initial padding (internal fragmentation)
  - Note `size`: padding is considered part of *previous* block
- Special one-word marker (0|1) marks end of list
  - Zero `size` is distinguishable from all other blocks

Implicit List: Finding a Free Block

(*p) gets the block *header*
(*p & 1) extracts the `allocated` bit
(*p & -2) extracts the `size`

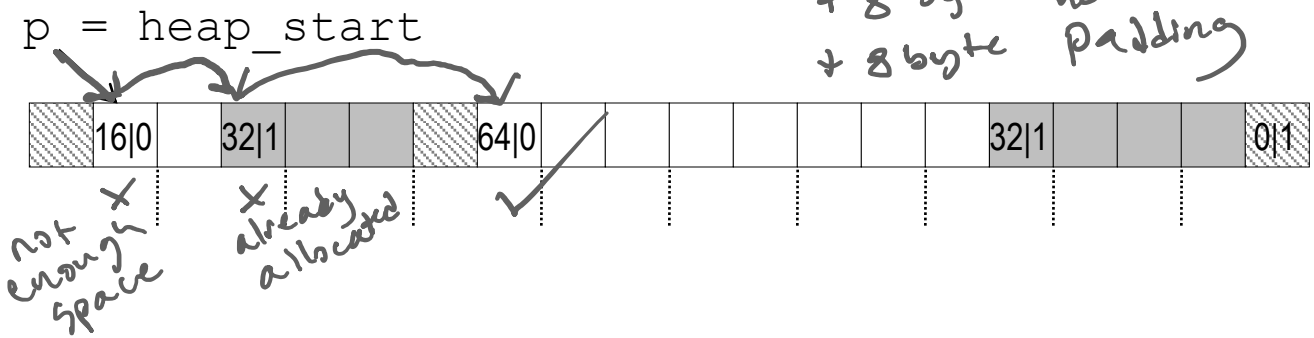
- *First fit*
  - **Search list from beginning**, choose first free block that fits:

```
p = heap_start;
while ((p < end) &&       // not past end
       ((*p & 1) ||       // already allocated
        (*p <= len))) {   // too small
  p = p + (*p & -2);      // go to next block (UNSCALED +)
}                         // p points to selected block or end
```

equivalent to pointer arithmetic w/ char*

  - Can take time linear O(n) in total number of blocks
  - In practice can cause "splinters" at beginning of list

malloc(16) need 32 bytes = 16 byte payload + 8 byte header + 8 byte padding



Free word
Allocated word
Allocated word unused

Implicit List: Finding a Free Block

- *Next fit*
  - Like first-fit, but **search list starting where previous search finished**
  - Should often be faster than first-fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse
- *Best fit*
  - Search the list, choose the ***best*** free block: large enough AND with fewest bytes left over
  - Keeps fragments small—usually helps fragmentation
  - Usually worse throughput

Polling Question [Alloc II]

- Which allocation strategy and requests remove *external* fragmentation in this Heap? B3 was the last fulfilled request.
  - http://pollev.com/pbjones

(A) Best-fit:

`malloc(50),malloc(50)`

(B) First-fit:

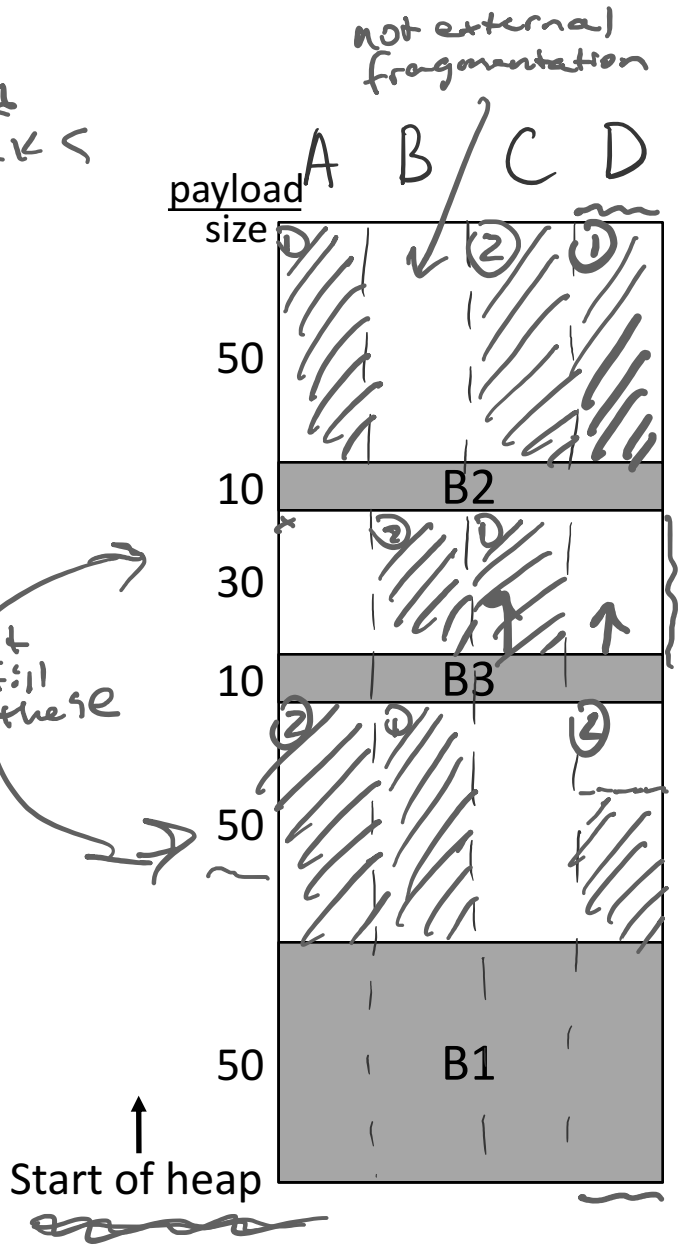
`malloc(50),malloc(30)`

(C) Next-fit:

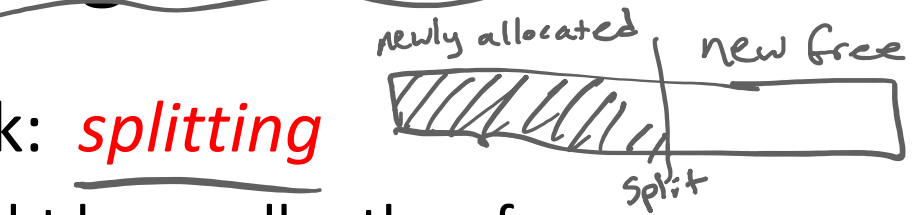
`malloc(30),malloc(50)`

(D) Next-fit:

`malloc(50),malloc(30)`



Implicit List: Allocating in a Free Block

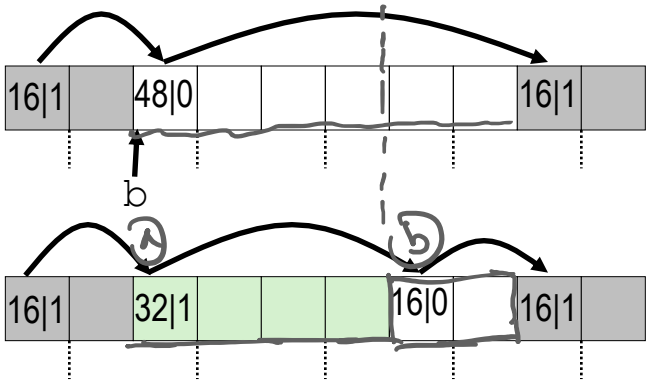


- Allocating in a free block: *splitting*
  - Since allocated space might be smaller than free space, we might want to split the block if the leftover space is big enough

Assume `ptr` points to a *free* block and has unscaled pointer arithmetic

```
void split(ptr b, int bytes) {                  // bytes = desired block size
  int newsize = ((bytes+15) >> 4) << 4;         // round up to multiple of 16
  int oldsize = *b;                             // why not mask out low bit?
  *b = newsize;                                 // initially unallocated
  if (newsize < oldsize)
    *(b+newsize) = oldsize - newsize;           // set length in remaining
}                                               // part of block (UNSCALED +)
```

```
malloc(24):
  ptr b = find(24+8)
  split(b, 24+8)
  allocate(b)
```



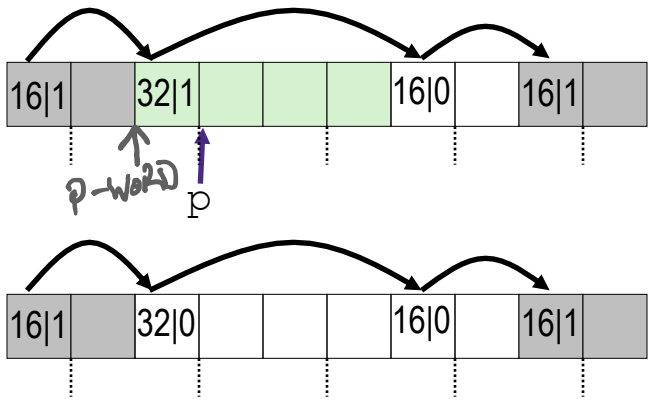
Free word

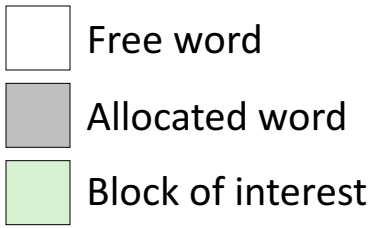
Allocated word

Newly-allocated word

Implicit List: Freeing a Block

- Simplest implementation just clears "allocated" flag
  - `void free(ptr p) {*(p-WORD) &= -2;}`
  - But can lead to "false fragmentation"



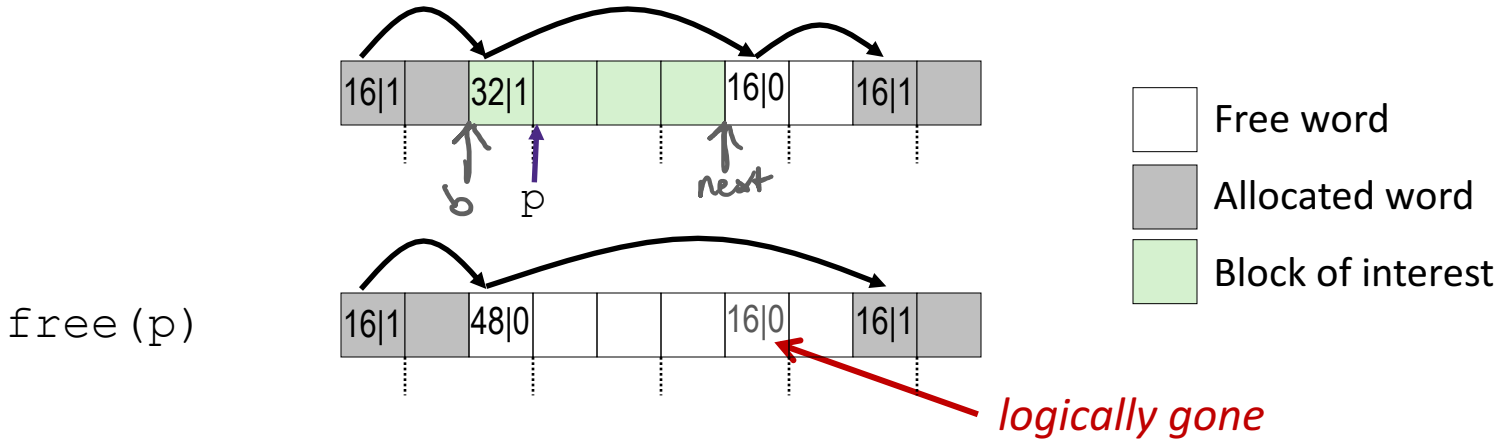


`malloc(40)`

Oops! There is enough free space, but the allocator won't be able to find it

Implicit List: Coalescing with Next

- Join (*coalesce*) with next block if also free



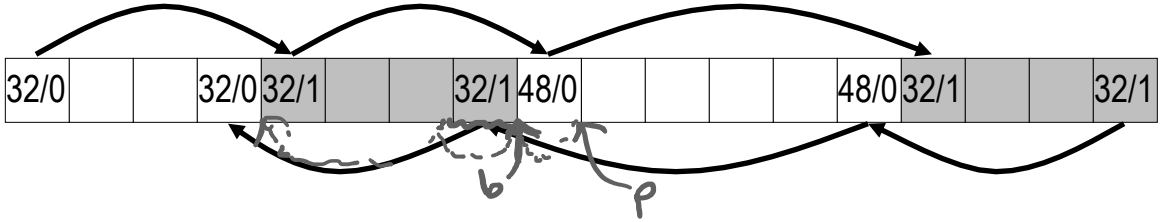

```
void free(ptr p) {          // p points to payload
    ptr b = p - WORD;        // b points to block header
    *b &= -2;                // clear allocated bit
    ptr next = b + *b;       // find next block (UNSCALED +)
    if ((*next & 1) == 0)    // if next block is not allocated,
      *b += *next;           // add its size to this block
}
```

- How do we coalesce with the *previous* block?

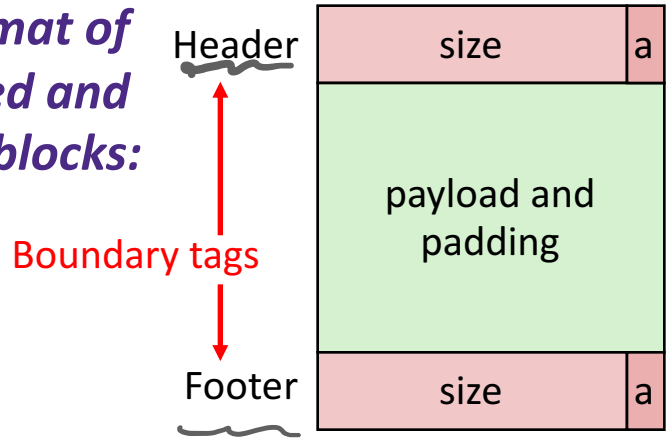
we can't !! need more info

Implicit List: Bidirectional Coalescing

- *Boundary tags* [Knuth73]
  - Replicate header at "bottom" (end) of free blocks
  - Allows us to traverse backwards, but requires extra space
  - Important and general technique!



Format of allocated and free blocks:

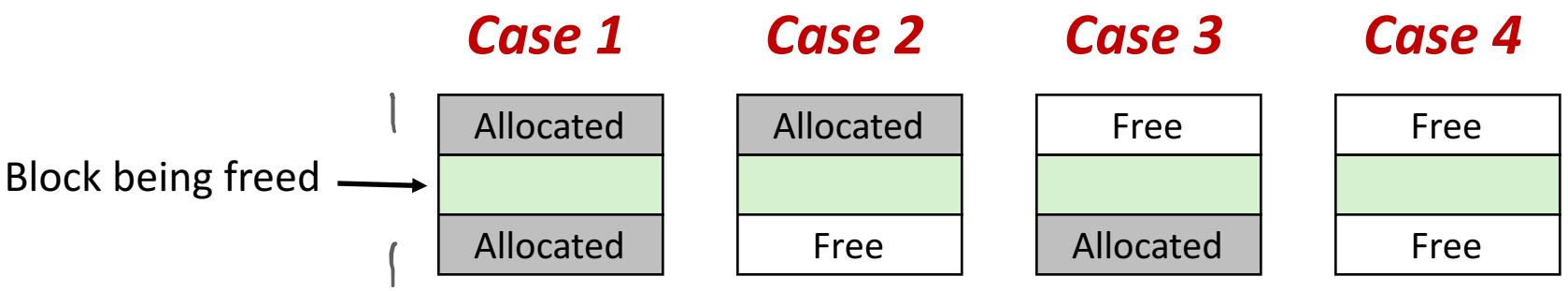


a = 1: allocated block
a = 0: free block

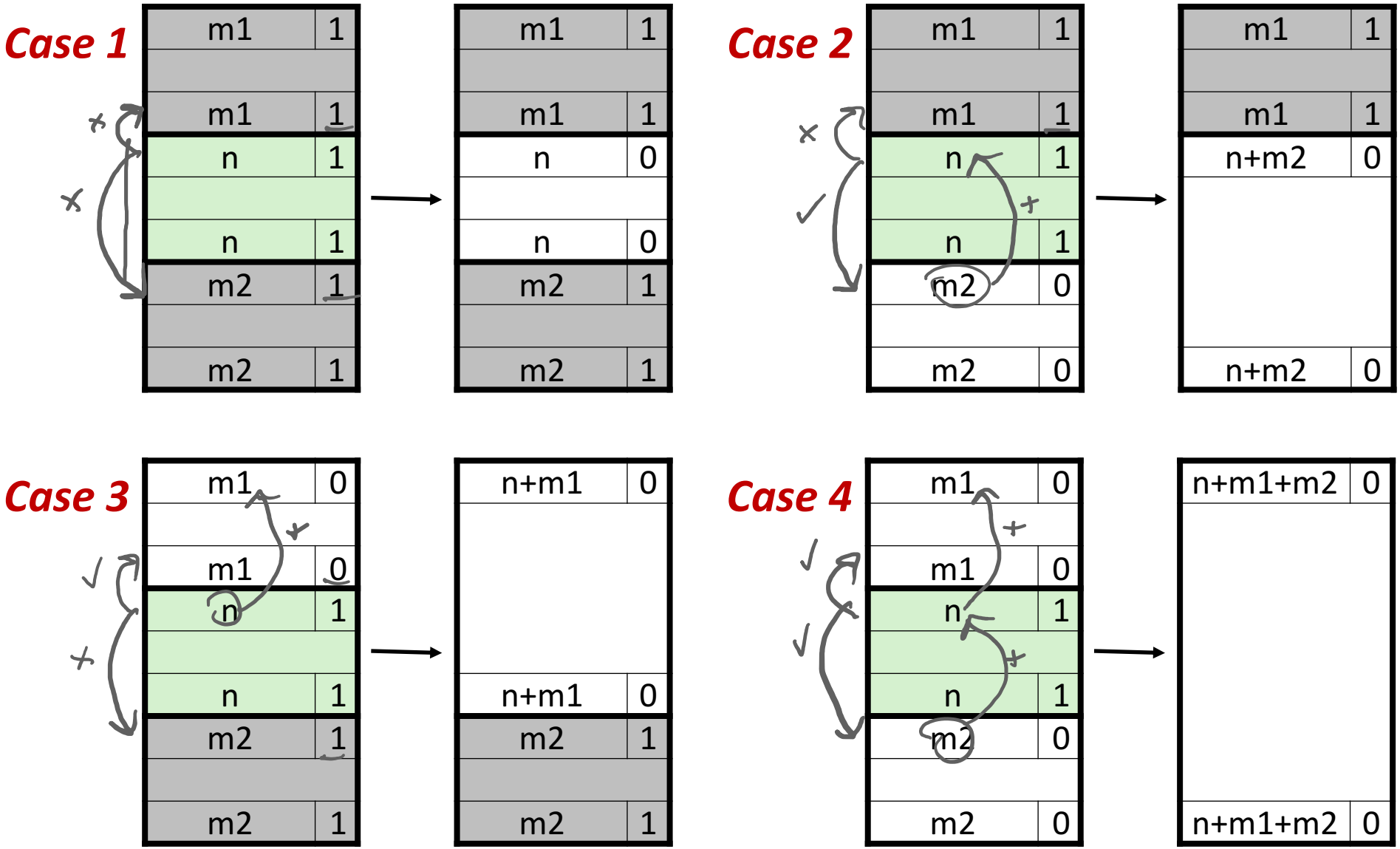
size: block size (in bytes)

payload: application data (allocated blocks only)

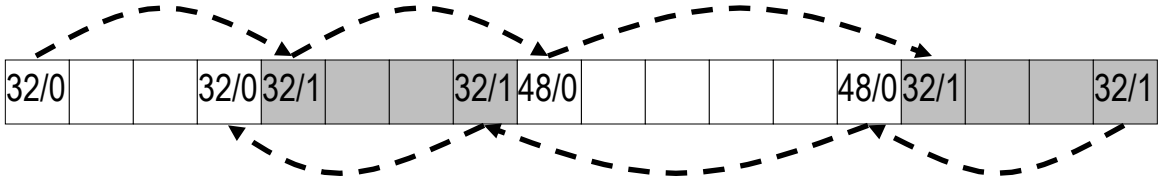
Constant Time Coalescing



Constant Time Coalescing



Implicit Free List Review Questions



- What is the block header? What do we store and how?

  stores info about block: size and isallocated?

  header = size | is_allocated

- What are boundary tags and why do we need them?

  header and footer

  traverse the heap both forwards and backwards (helpful w/coalescing)

- When we coalesce free blocks, how many neighboring blocks do we need to check on either side? Why is this?

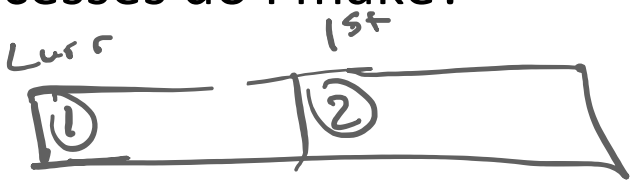
  one on each side: adjacent free blocks have already been coalesced

- If I want to check the size of the n-th block forward from the current block, how many memory accesses do I make?

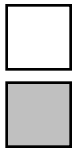
  n+1 accesses

  to get to block in question

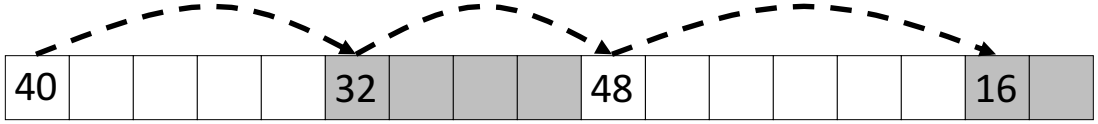
  to look at the size of block



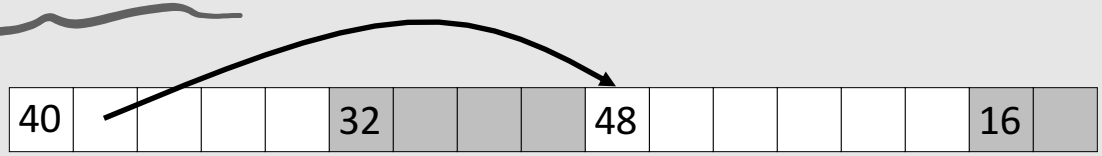
Keeping Track of Free Blocks



1) *Implicit free list* using length – links all blocks using math
   - No actual pointers, and must check each block if allocated or free



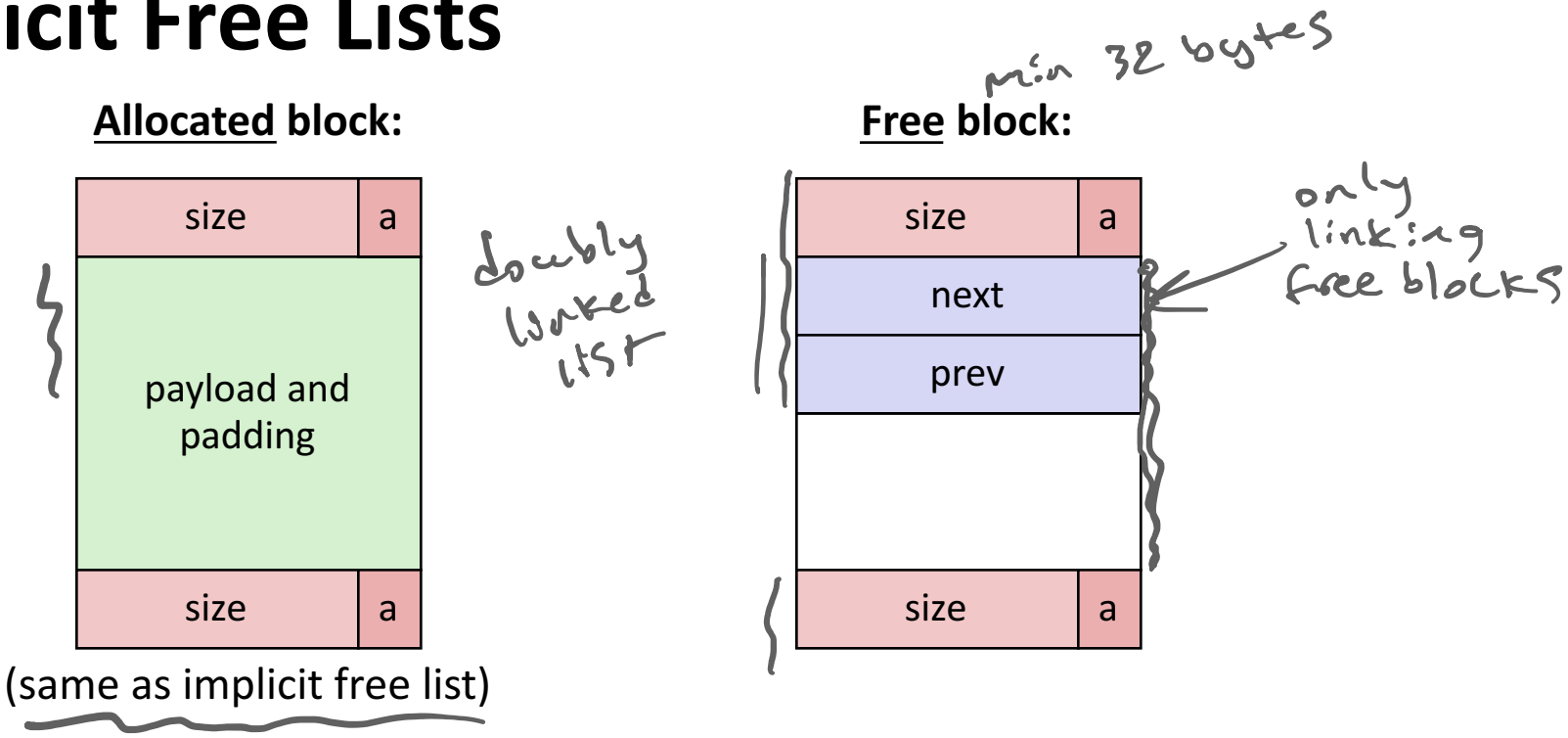
2) *Explicit free list* among only the free blocks, using pointers



3) *Segregated free list*
   - Different free lists for different size "classes"

4) *Blocks sorted by size*
   - Can use a balanced binary tree (*e.g.* red-black tree) with pointers within each free block, and the length used as a key

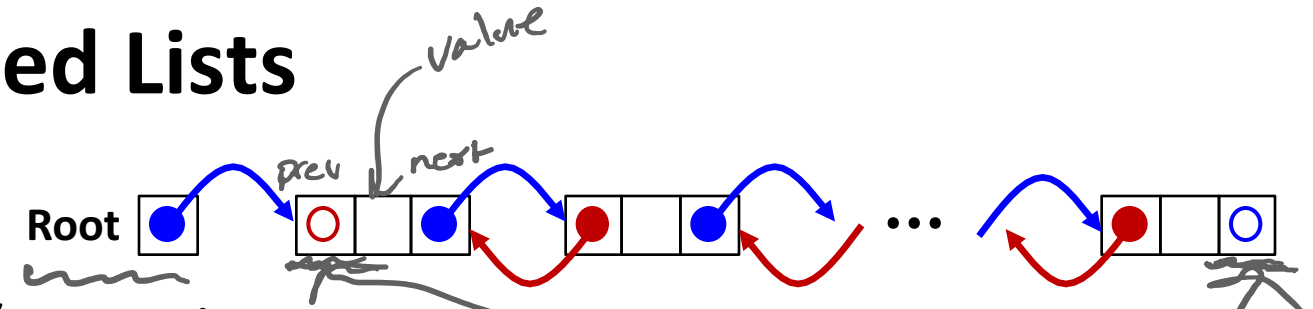
Explicit Free Lists



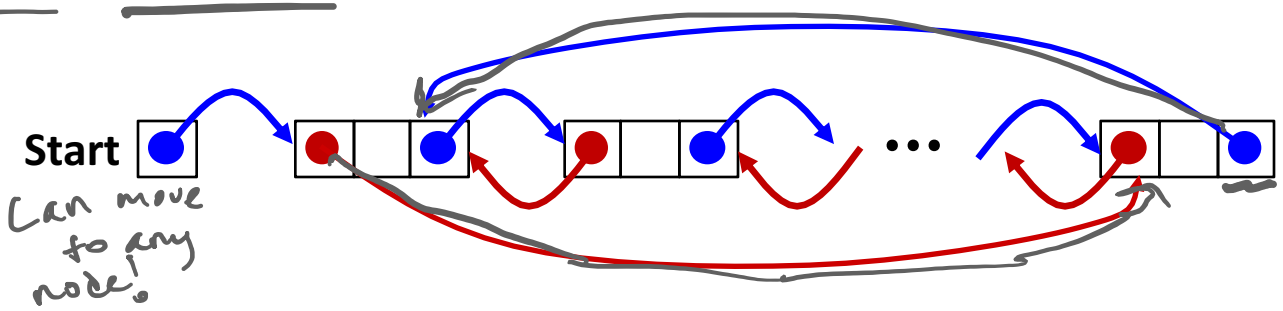
- Use list(s) of *free* blocks, rather than implicit list of *all* blocks
  - The "next" free block could be anywhere in the heap
    - So we need to store next/previous pointers, not just sizes
  - Since we only track free blocks, so we can use "payload" for pointers
  - Still need boundary tags (header/footer)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

Doubly-Linked Lists

- **Linear**
  - Needs head/root pointer
  - First node prev pointer is `NULL`
  - Last node next pointer is `NULL`
  - Good for first-fit, best-fit

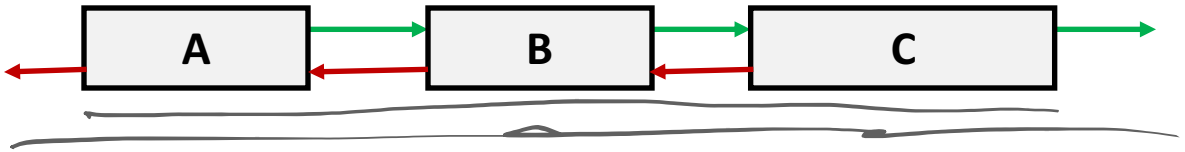


- **Circular**
  - Still have pointer to tell you which node to start with
  - No `NULL` pointers (term condition is back at starting point)
  - Good for next-fit, best-fit

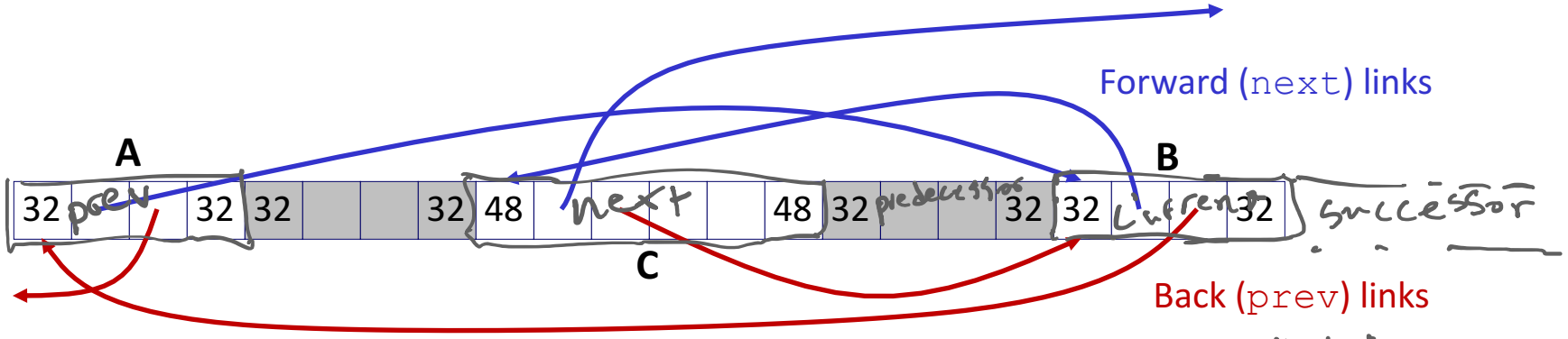


Explicit Free Lists

- **Logically:** doubly-linked list



- **Physically:** blocks can be in any order

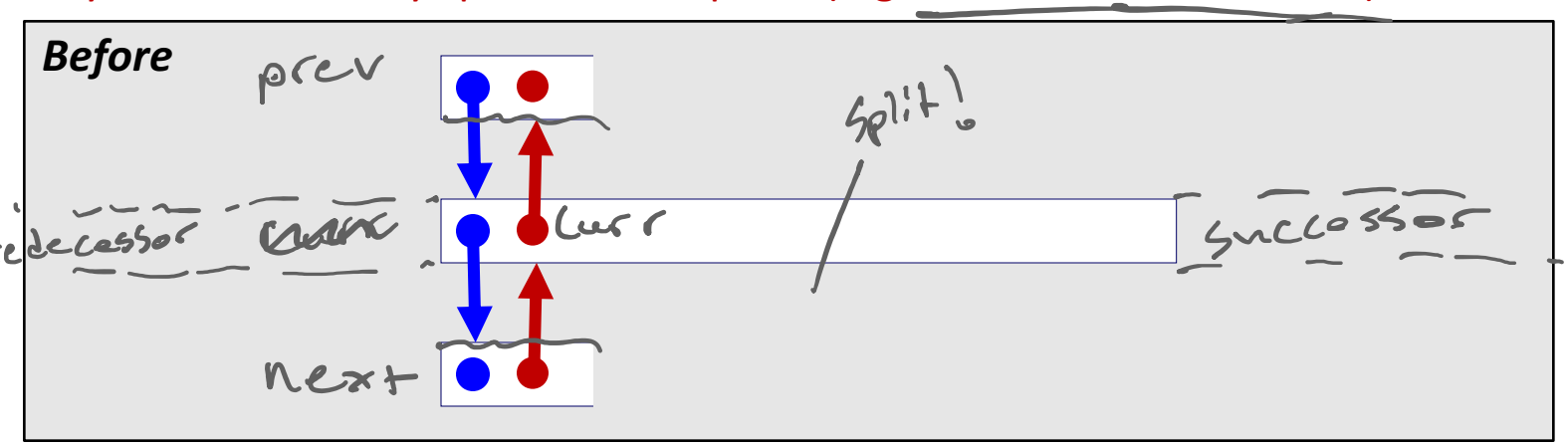


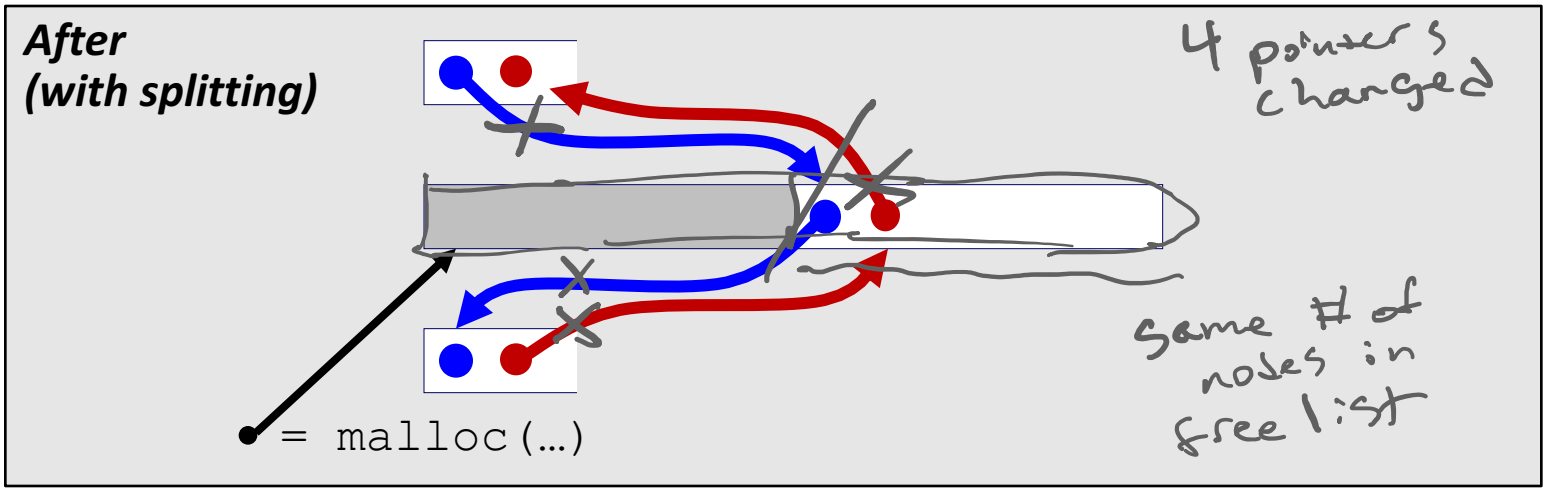
next/prev: next and prev in linked list

predecessor/successor: blocks physically next to our current block

Allocating From Explicit Free Lists

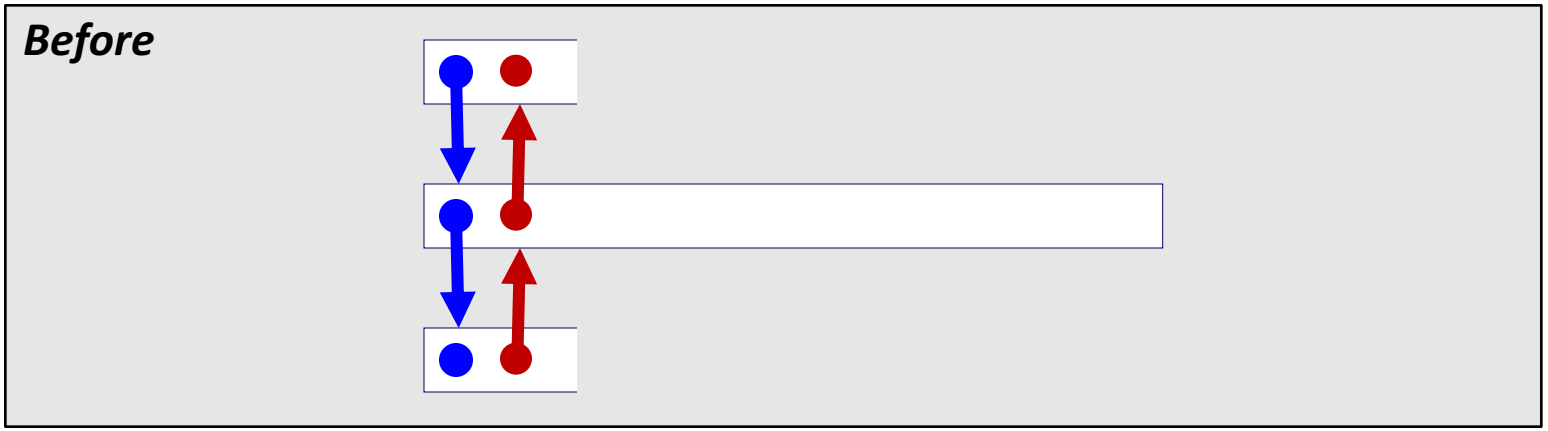
Note: These diagrams are not very specific about *where inside a block* a pointer points. In reality we would always point to one place (e.g. start/header of a block).

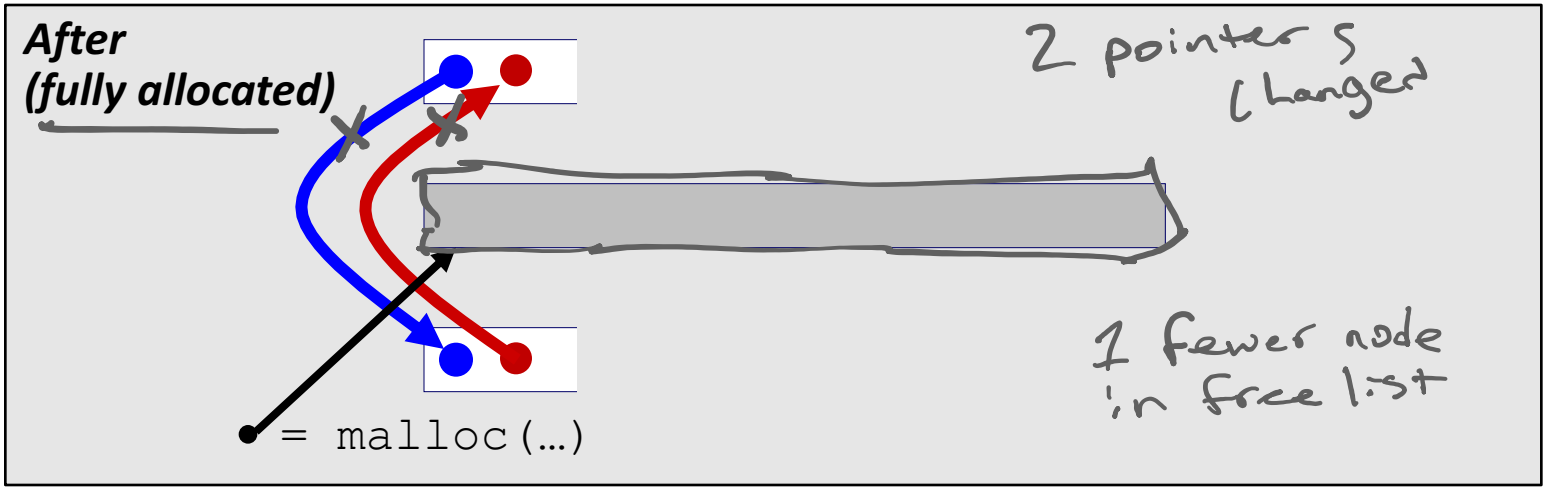




Allocating From Explicit Free Lists

Note: These diagrams are not very specific about *where inside a block* a pointer points. In reality we would always point to one place (e.g. start/header of a block).



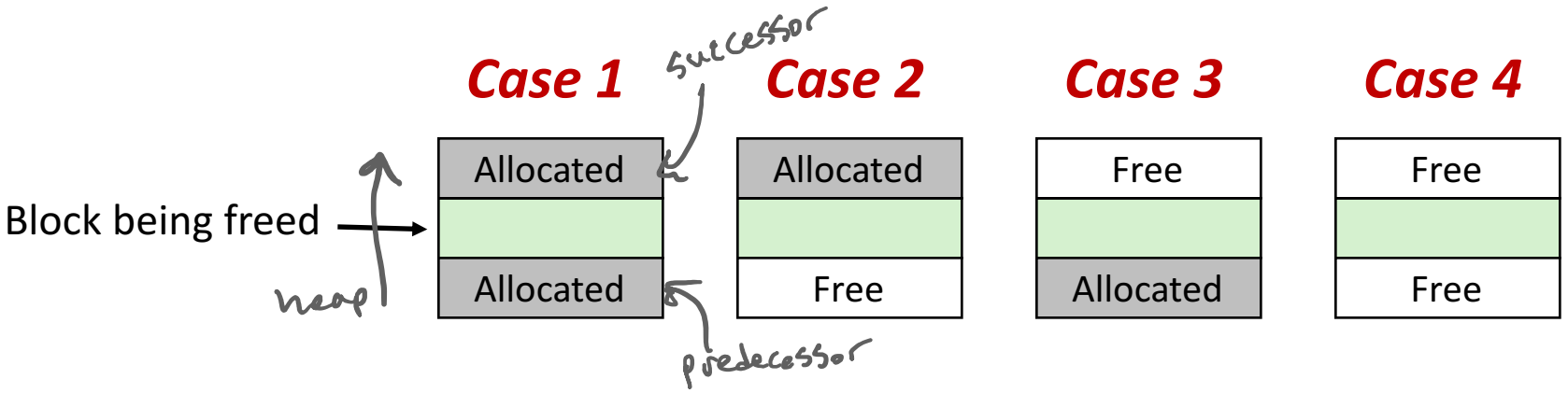


Freeing With Explicit Free Lists

- *Insertion policy:* Where in the free list do you put the newly freed block?
  - **LIFO (last-in-first-out) policy**
    - Insert freed block at the beginning (head) of the free list
    - Pro: simple and constant time
    - Con: studies suggest fragmentation is worse than the alternative
  - **Address-ordered policy**
    - Insert freed blocks so that free list blocks are always in address order:

      address(previous) < address(current) < address(next)
    - Con: requires linear-time search
    - Pro: studies suggest fragmentation is better than the alternative

Coalescing in Explicit Free Lists



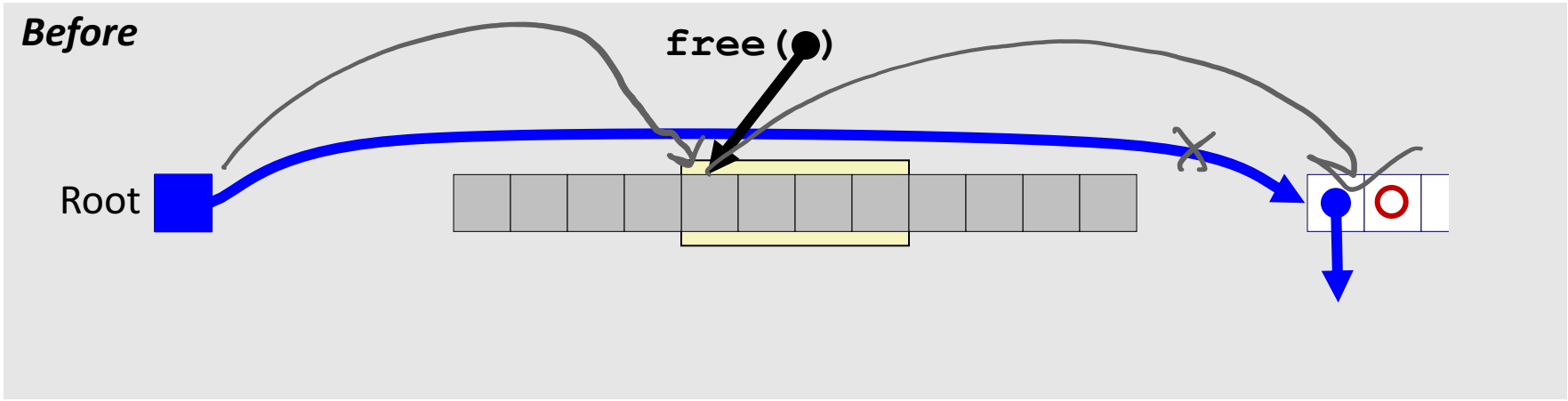
- Neighboring free blocks are *already part of the free list*
  1) Remove old block from free list
  2) Create new, larger coalesced block
  3) Add new block to free list (insertion policy)

- How do we tell if a neighboring block if free?

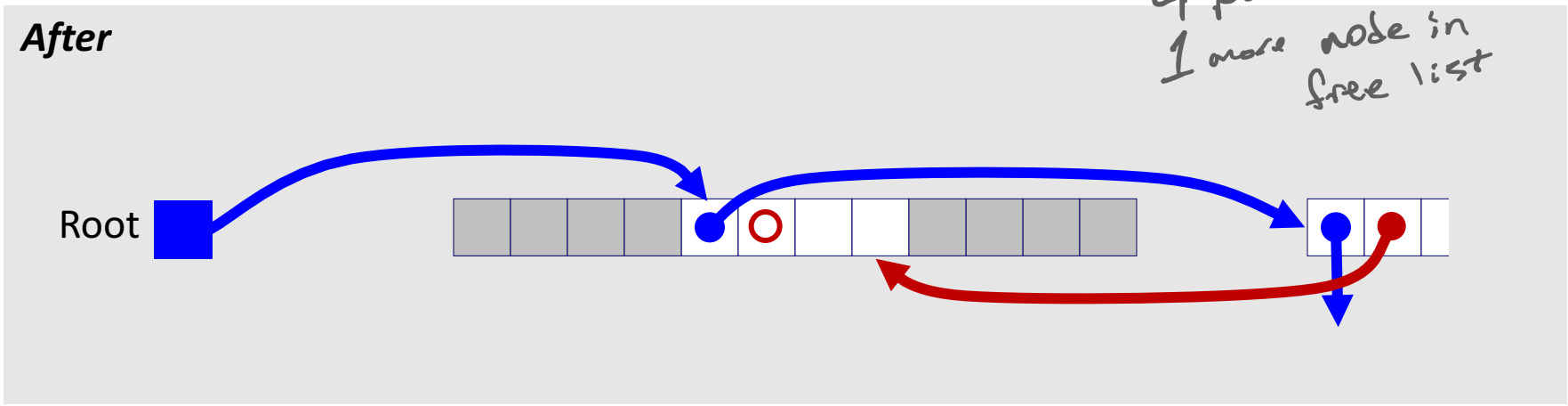
Can still use boundary tags!

Freeing with LIFO Policy (Case 1)

Boundary tags not shown, but don't forget about them!

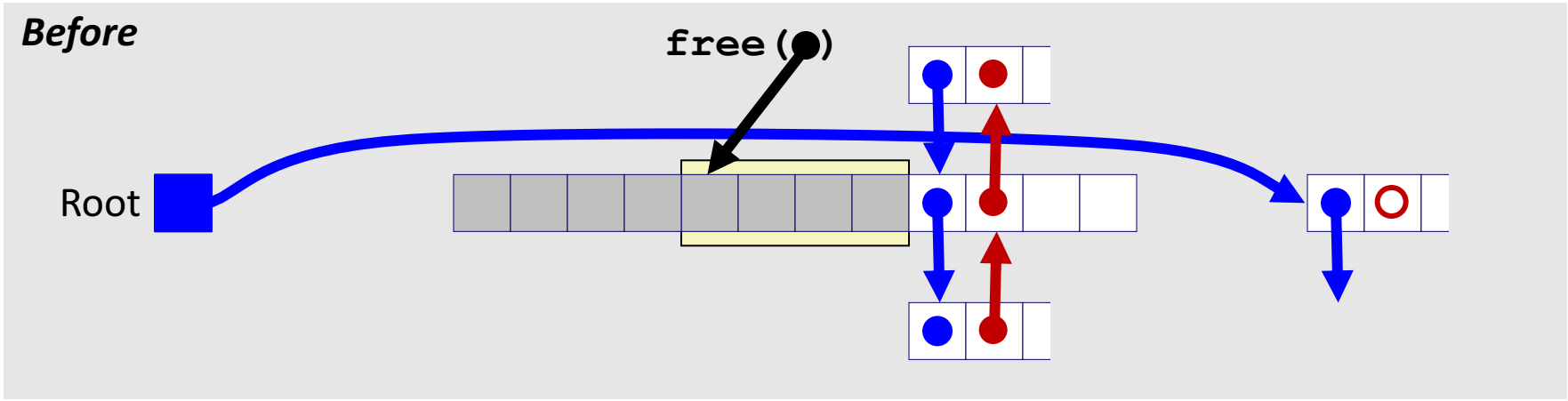


- Insert the freed block at the root of the list



Freeing with LIFO Policy (Case 2)

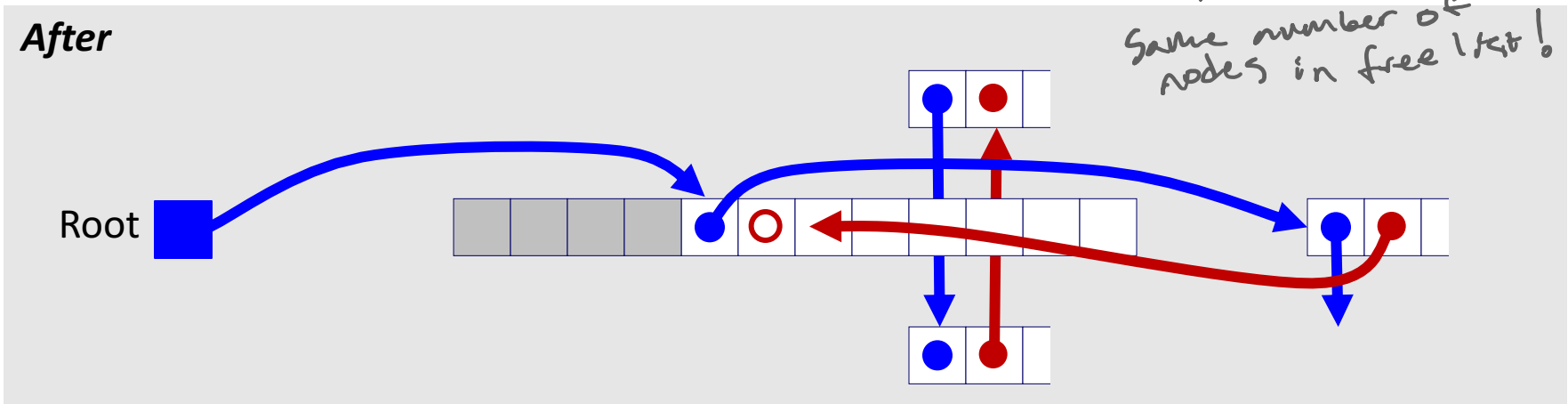
Boundary tags not shown, but don't forget about them!



- Splice *successor* block out of list, coalesce both memory blocks, and insert the new block at the root of the list

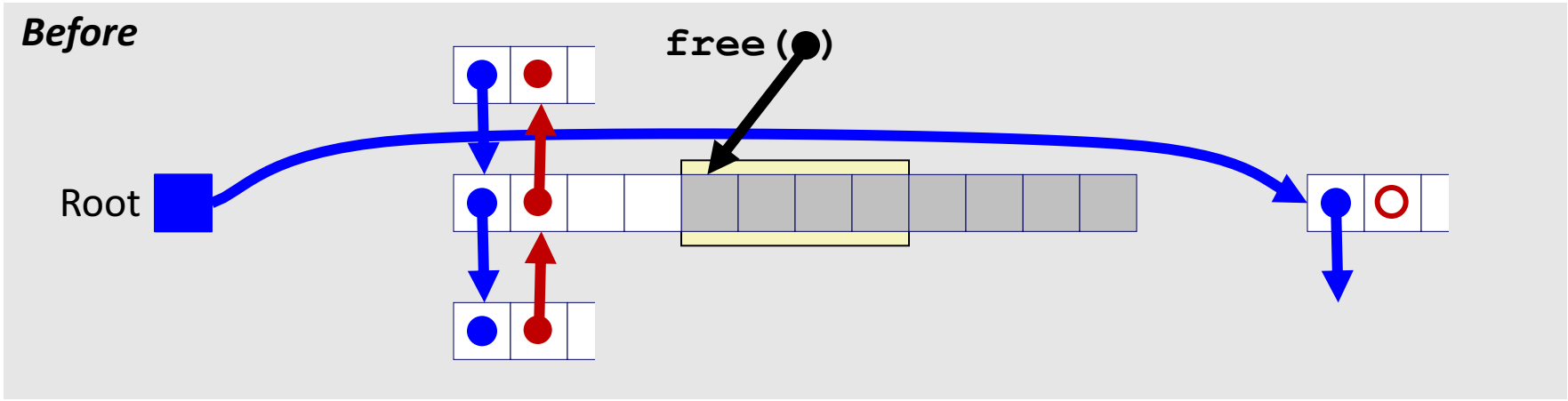
5 pointers updated

Same number of nodes in free list!

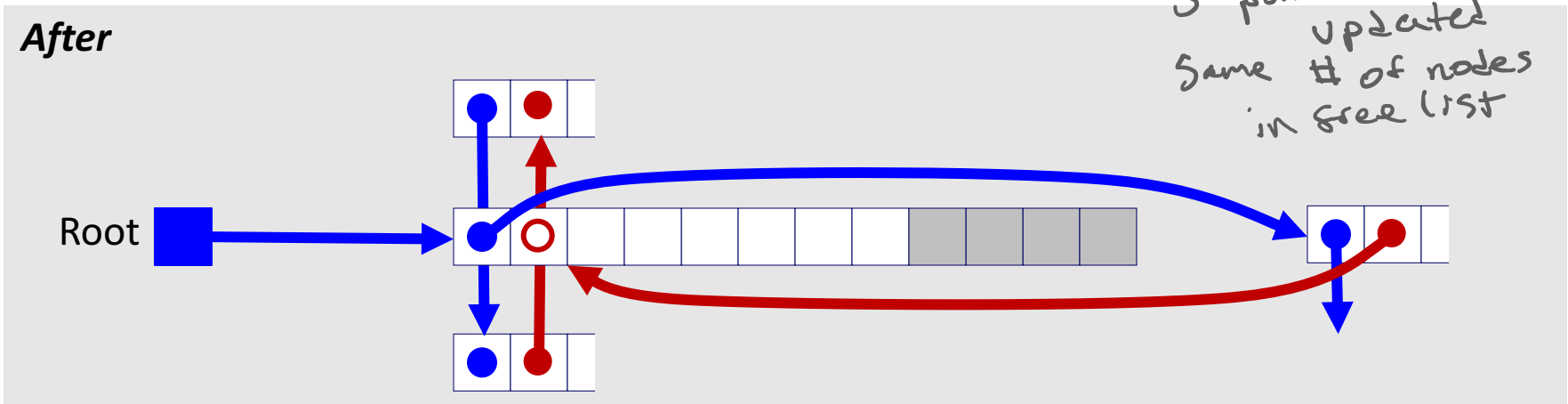


Freeing with LIFO Policy (Case 3)

Boundary tags not shown, but don't forget about them!

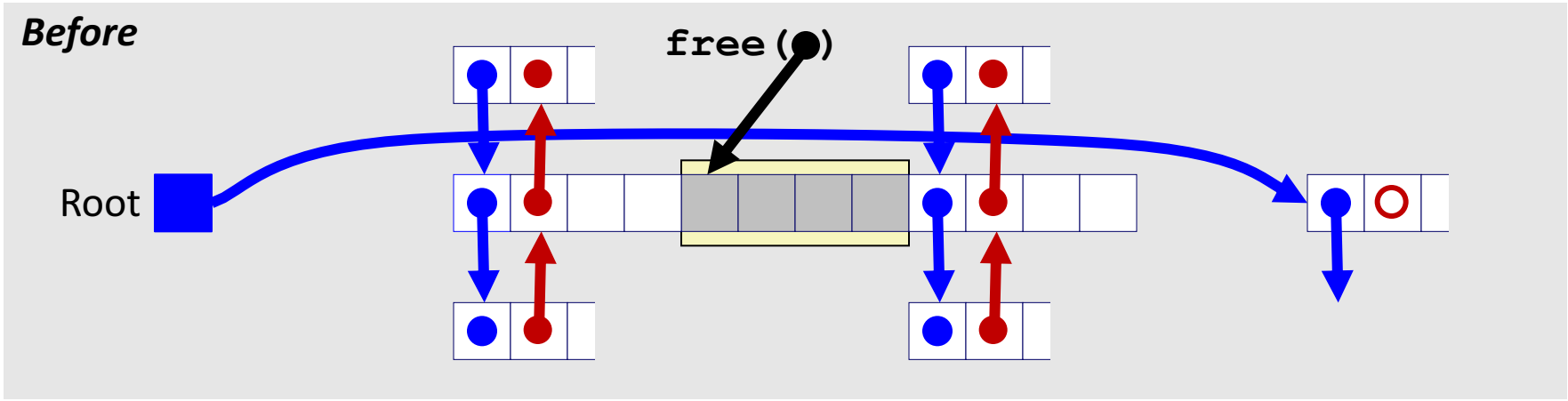


- Splice *predecessor* block out of list, coalesce both memory blocks, and insert the new block at the root of the list



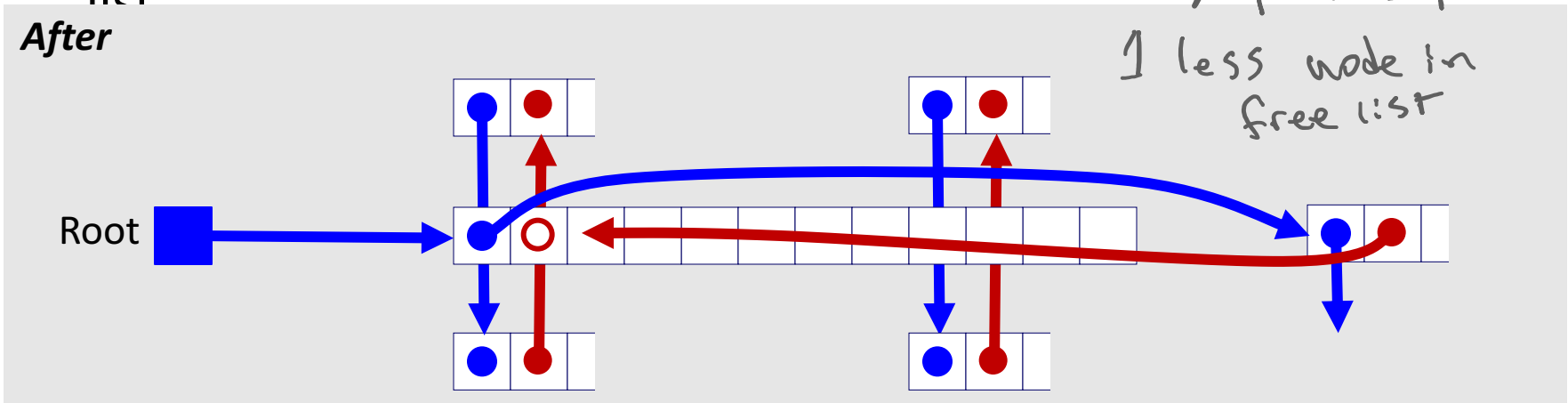
Freeing with LIFO Policy (Case 4)

Boundary tags not shown, but don't forget about them!



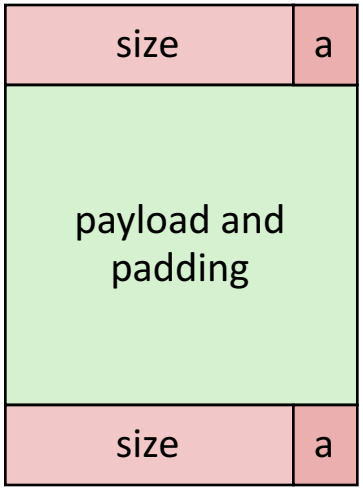
- Splice *predecessor* and *successor* blocks out of list, coalesce all 3 memory blocks, and insert the new block at the root of the list

7 pointers updated



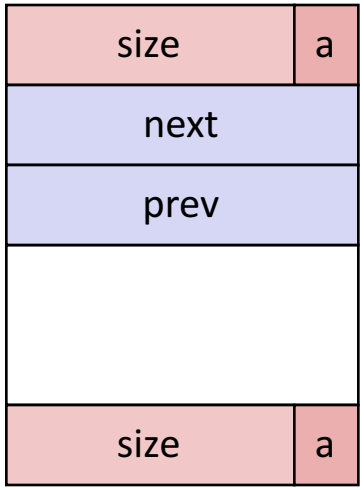
Do we always need the boundary tags?

Allocated block:



(same as implicit free list)

Free block:



- Lab 5 suggests no…

Explicit List Summary

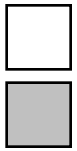
- Comparison with implicit list:
  - Block allocation is linear time in number of ***free*** blocks instead of ***all*** blocks
    - ***Much faster*** when most of the memory is full
  - Slightly more complicated allocate and free since we need to splice blocks in and out of the list
  - Some extra space for the links (2 extra pointers needed for each free block)
    - Increases minimum block size, leading to more internal fragmentation

- Most common use of explicit lists is in conjunction with *segregated free lists*
  - Keep multiple linked lists of different size classes, or possibly for different types of objects

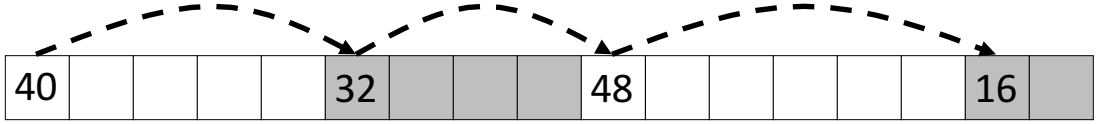
BONUS SLIDES

The following slides are about the **SegList Allocator**, for those curious. You will NOT be expected to know this material.

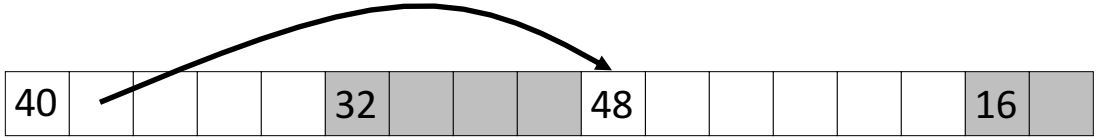
Keeping Track of Free Blocks



1) *Implicit free list* using length – links all blocks using math
   - No actual pointers, and must check each block if allocated or free



2) *Explicit free list* among only the free blocks, using pointers

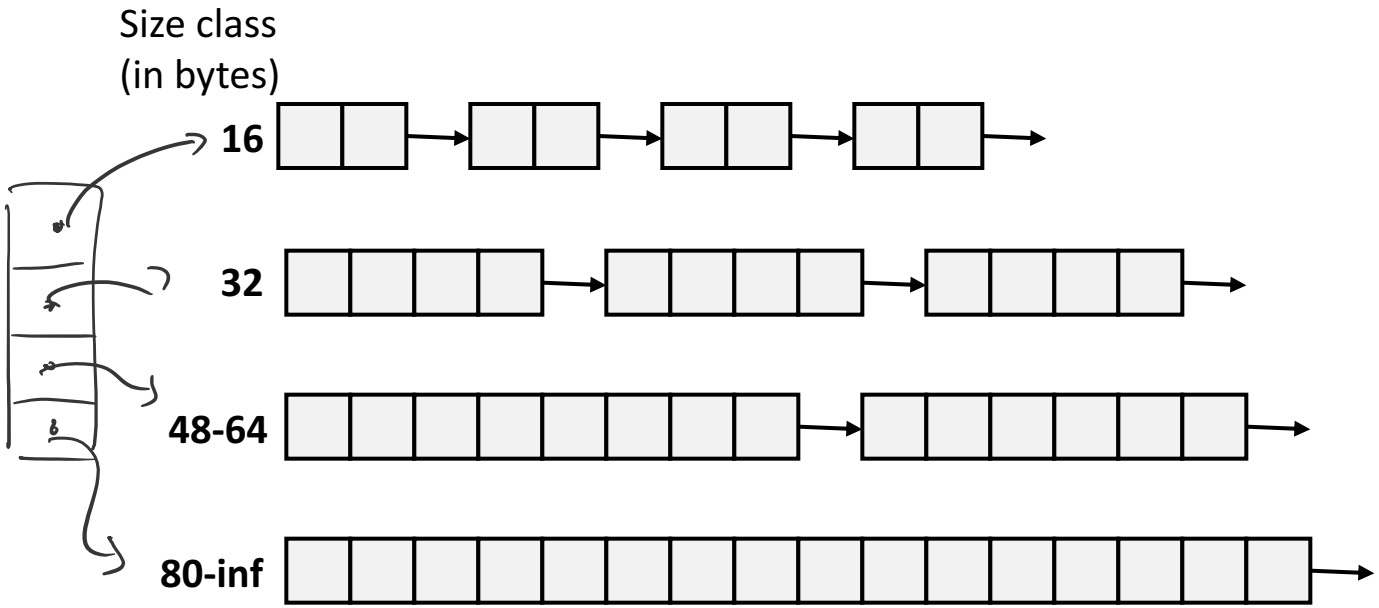


3) *Segregated free list*
   - Different free lists for different size "classes"

4) *Blocks sorted by size*
   - Can use a balanced binary tree (e.g. red-black tree) with pointers within each free block, and the length used as a key

Segregated List (SegList) Allocators

- Each *size class* of blocks has its own free list
- Organized as an array of free lists



- Often have separate classes for each small size
- For larger sizes: One class for each two-power size

SegList Allocator

- Have an array of free lists for various size classes

- To allocate a block of size n:
  - Search appropriate free list for block of size $m \geq n$
  - If an appropriate block is found:
    - [Optional] Split block and place free fragment on appropriate list
  - If no block is found, try the next larger class
    - Repeat until block is found
- If no block is found:
  - Request additional heap memory from OS (using `sbrk`)
  - Place remainder of additional heap memory as a single free block in appropriate size class

SegList Allocator

- Have an array of free lists for various size classes

- To free a block:
  - Mark block as free
  - Coalesce (if needed)
  - Place on appropriate class list

SegList Advantages

- Higher throughput
  - Search is log time for power-of-two size classes
- Better memory utilization
  - First-fit search of seglist approximates a best-fit search of entire heap
  - *Extreme case:* Giving every block its own size class is no worse than best-fit search of an explicit list
  - Don't need to use space for block size for the fixed-size classes

Freeing with LIFO Policy (Explicit Free List)

	Predecessor Block	Successor Block	Change in Nodes in Free List	Number of Pointers Updated
Case 1	Allocated	Allocated		
Case 2	Allocated	Free		
Case 3	Free	Allocated		
Case 4	Free	Free		