Memory Allocation II

CSE 351 Spring 2024

Instructor:

Elba Garza

Teaching Assistants:

Ellis Haker
Adithi Raghavan
Aman Mohammed
Brenden Page
Celestine Buendia
Chloe Fong
Claire Wang
Hamsa Shankar
Maggie Jiang
Malak Zaki
Naama Amiel
Nikolas McNamee
Shananda Dokka
Stephen Ying
Will Robertson

Playlist: CSE 351 24Sp Lecture Tunes!

Announcements, Reminders

- HW17/18 due tonight!
  - HW19 due Monday (13 May)
  - HW20 due Wednesday (15 May)
  - HW21 due Friday (17 May)
- Lab 4 due May 17th
  - Lab 5 will release same day!
  - Given Lab 5 is due May 31st, use any late days left on Lab 4!
- Looking ahead: Guest lectures on May 15th and 17th

Reading Review

- Terminology:
  - Allocation strategies: first fit, next fit, best fit
  - Allocating a block: splitting, minimum block size
  - Freeing a block: coalescing
  - Boundary tags: header and footer
  - Explicit free list

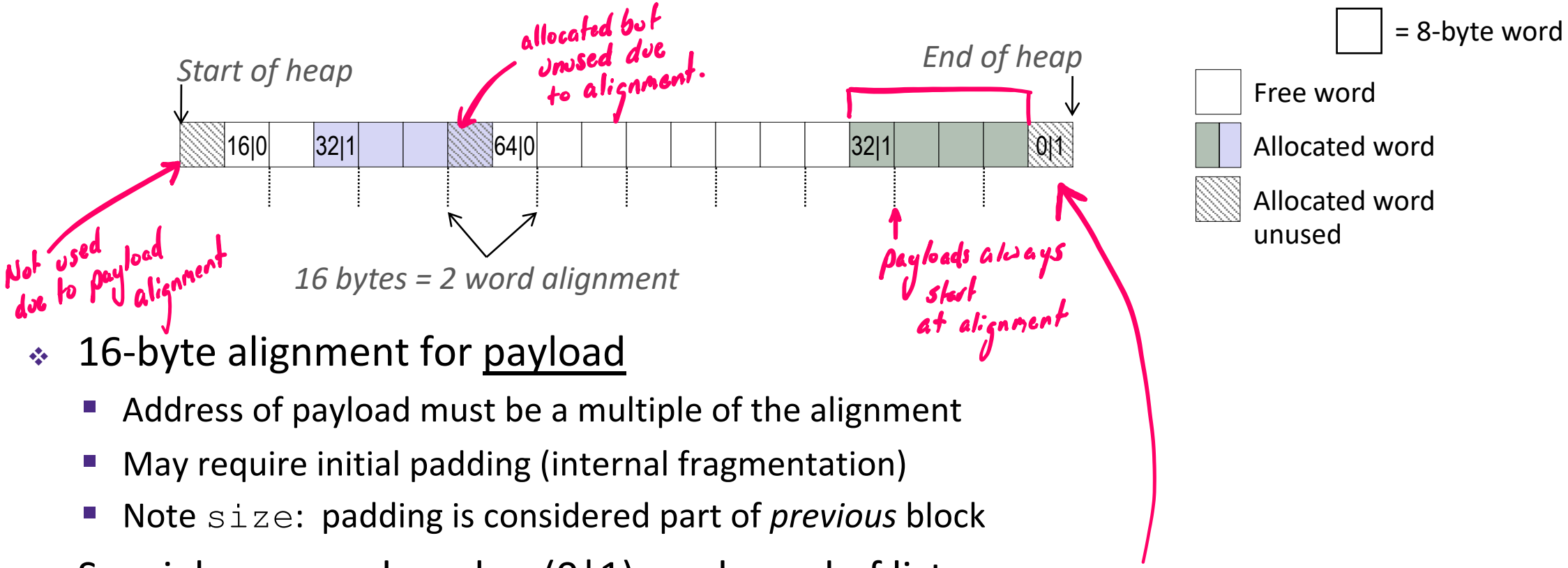
Header Questions

- How many “flags” can we fit in our header if our allocator uses 16-byte alignment?

- If we placed a new “flag” in the second least significant bit, write out a C expression that will extract this new flag from the `header`!

Implicit Free List Example

- Each block begins with header containing 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
  - Address of payload must be a multiple of the alignment
  - 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 (external fragmentation)

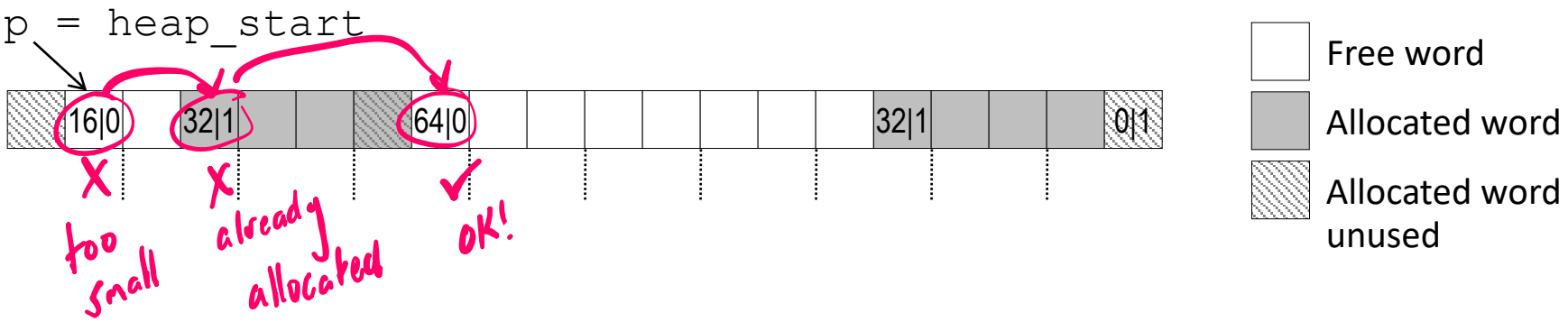
Implicit List: Finding a Free Block

(*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 of heap
       ((*p & 1) ||      // checked allocated bit & it's taken :(
        (*p <= len))) {  // block is too small :(
  p = p + (*p & -2);     // Thus, go to next block (UNSCALED +)
}                        // p points to selected block or end
```

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



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, because being picky means timing is worse

Polling Question

- Which allocation strategy and requests remove external fragmentation in this Heap? Note, B3 was the last fulfilled request.

(A) Best-fit:
`malloc(50), malloc(50)`

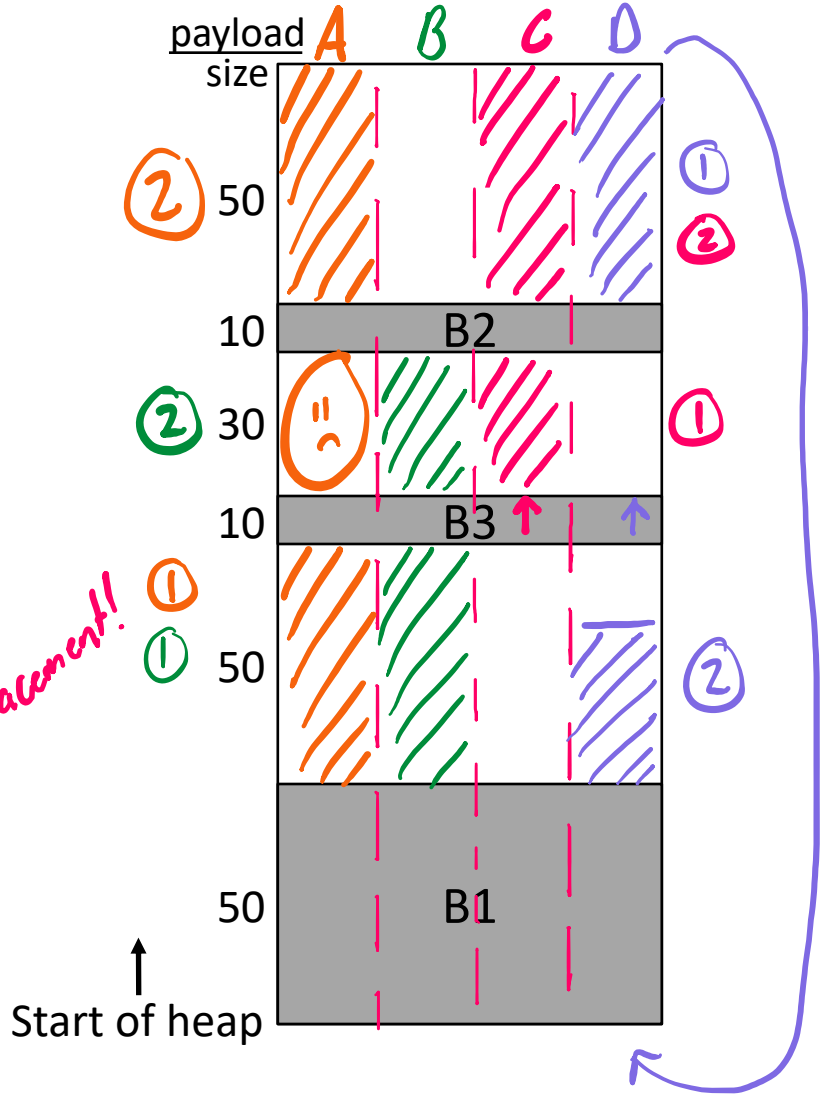
(B) First-fit:
`malloc(50), malloc(30)`

(C) Next-fit:
`malloc(30), malloc(50)`

(D) Next-fit:
`malloc(50), malloc(30)`

i.e. space between blocks

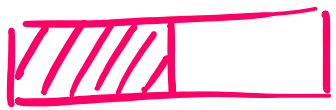
* Remember, start from B3's placement!



Implicit List: Allocating in a Free Block

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

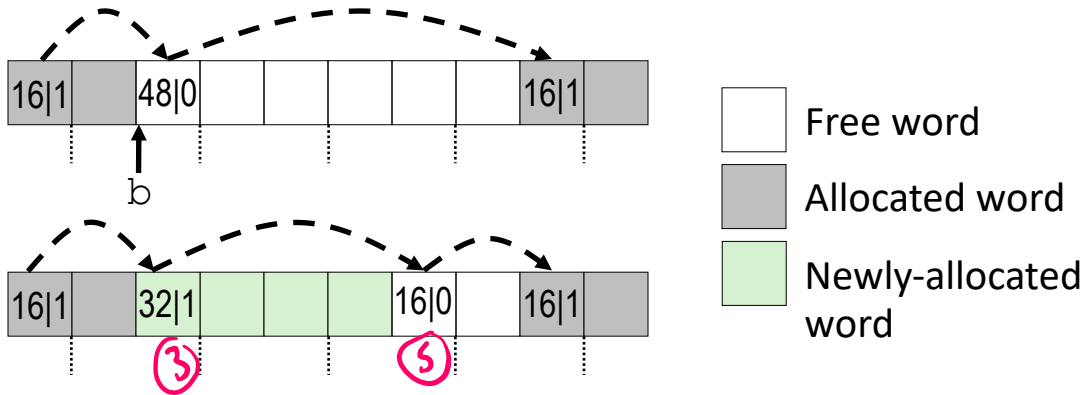
- Allocating in a free block: **splitting**



  - Since allocated space might be smaller than free space, we might want to split the block

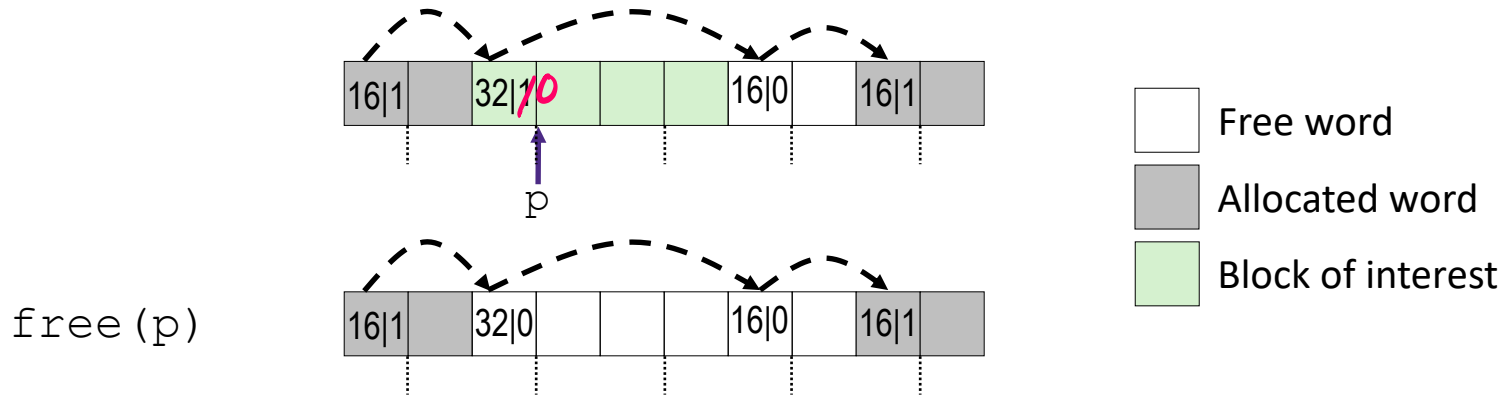
```
void split(ptr b, int bytes) {                  // bytes = desired block size
  int newsize = ((bytes+15) >> 4) << 4;         // round up to multiple of 16
  int oldsize = *b;                             // why not mask out low bit?
  *b = newsize;                                 // initially unallocated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



Implicit List: Freeing a Block

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



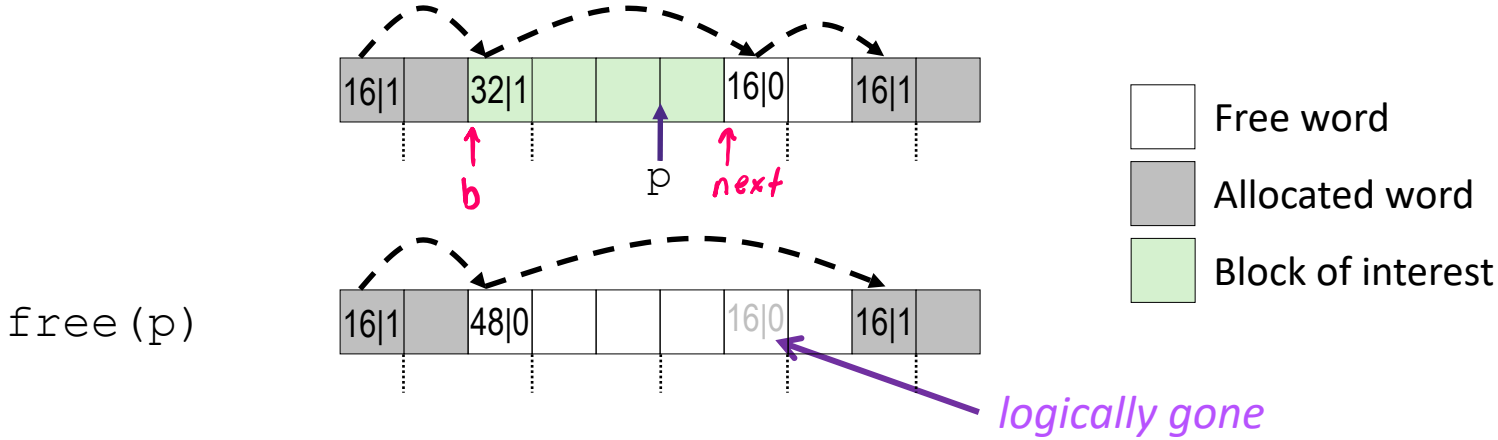
`malloc(40)`

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

It'll see 32 + 16 as separate!

Implicit List: Coalescing with Next

- Join *(i.e. coalesce)* with the next block if it's also free

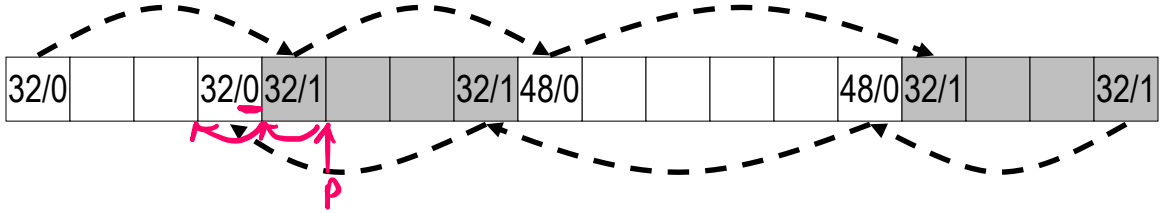


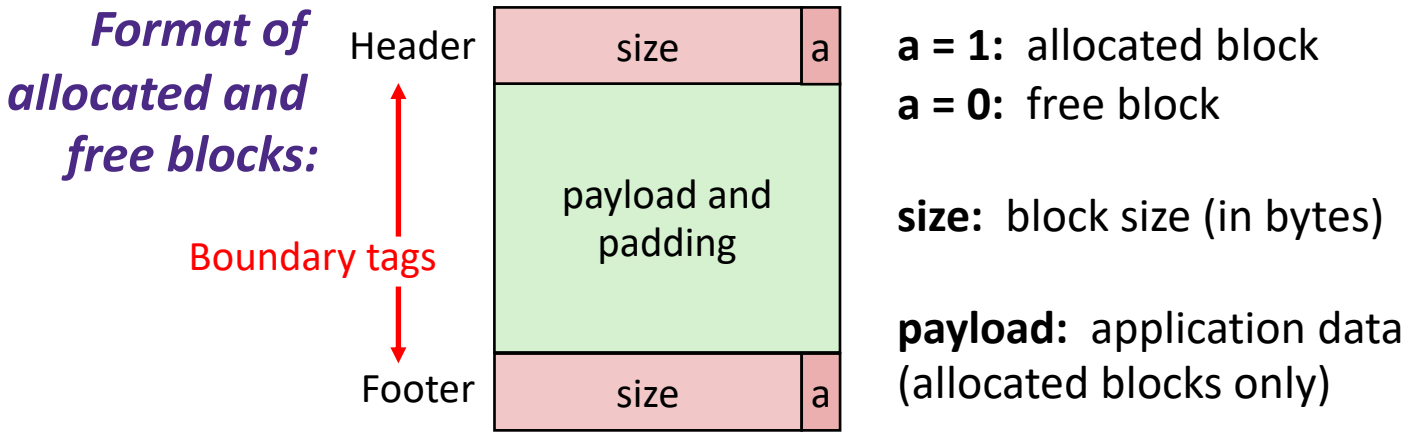

```
void free(ptr p) {          // p points to payload
    ptr b = p - WORD;       // b points to block header
    *b &= -2;               // clear allocated bit
    ptr next = b + *b;      // find next block (UNSCALED +)
    if ((*next & 1) == 0)   // if next block is not allocated,
      *b += *next;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

- How do we coalesce with a <u>preceding</u> block, though?

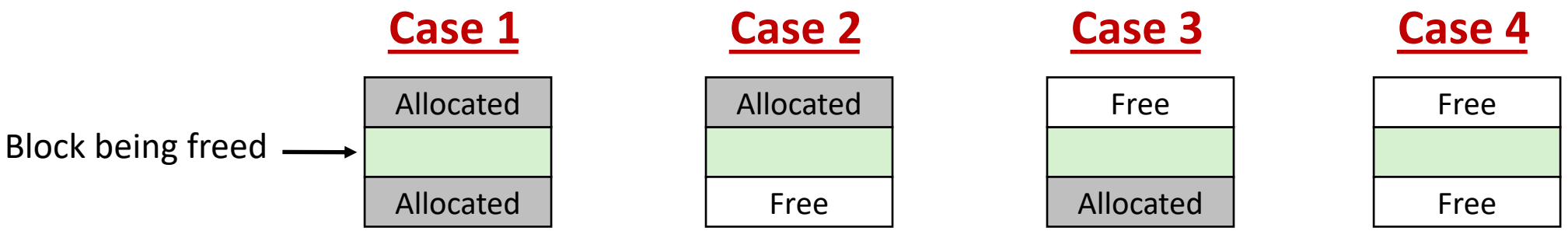
Implicit List: Bidirectional Coalescing

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



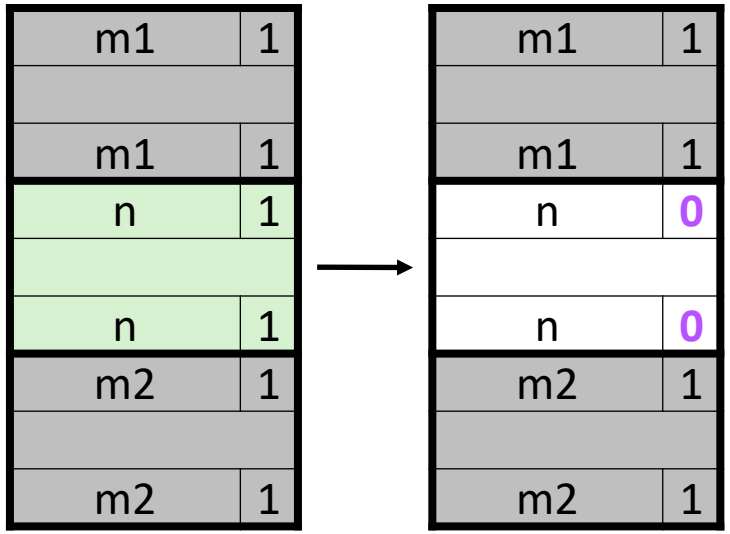


Constant Time Coalescing

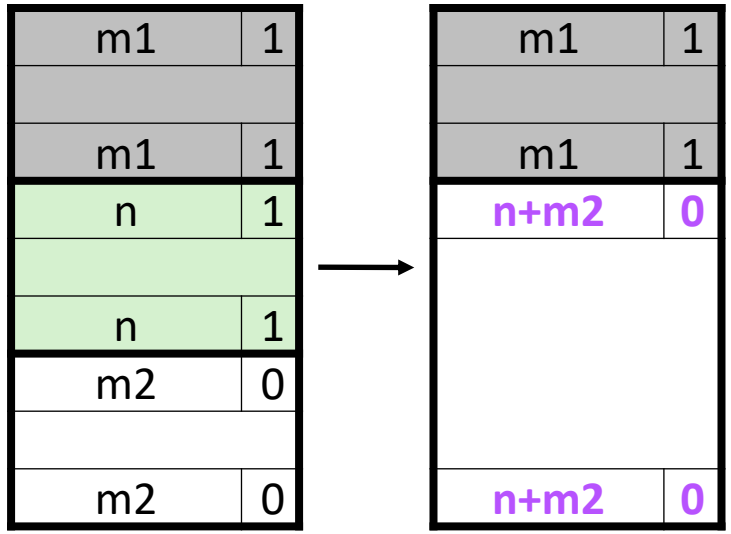


Constant Time Coalescing

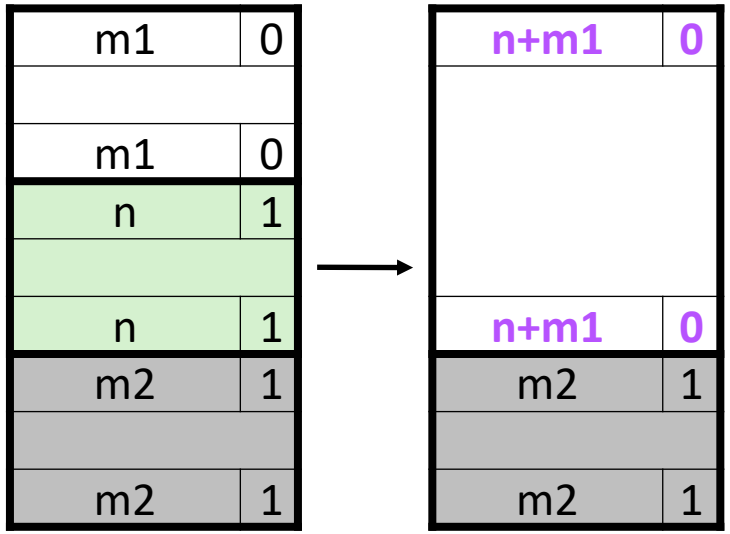
Case 1



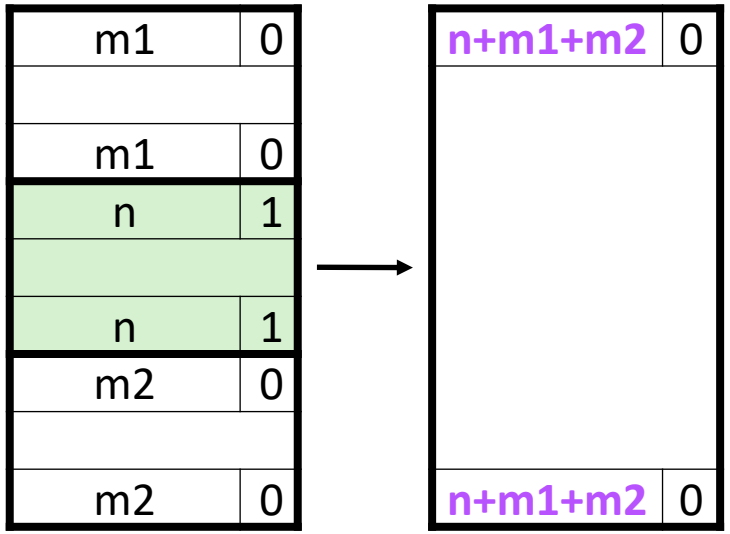
Case 2



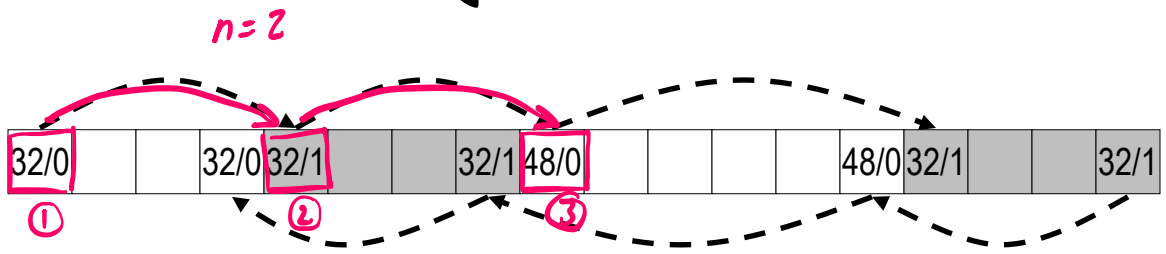
Case 3



Case 4



Implicit Free List Review Questions



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

  Stores info about block size of block & whether it's allocated (in LSB of size)

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

  header & footer (same data!) so we can traverse list in either direction (for coalescing)

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

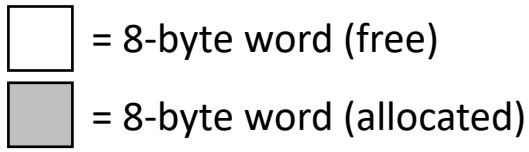
  two in general, so one on each size! (assuming all others have been coalesced correctly)

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

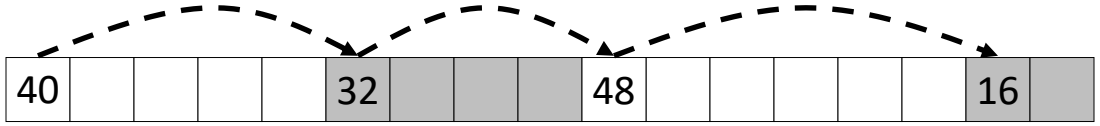
  for n blocks ⟶ n+1

  (see diagram)

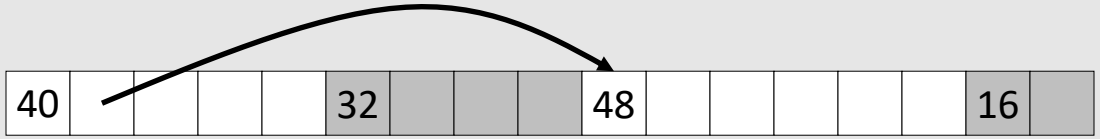
Keeping Track of Free Blocks



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



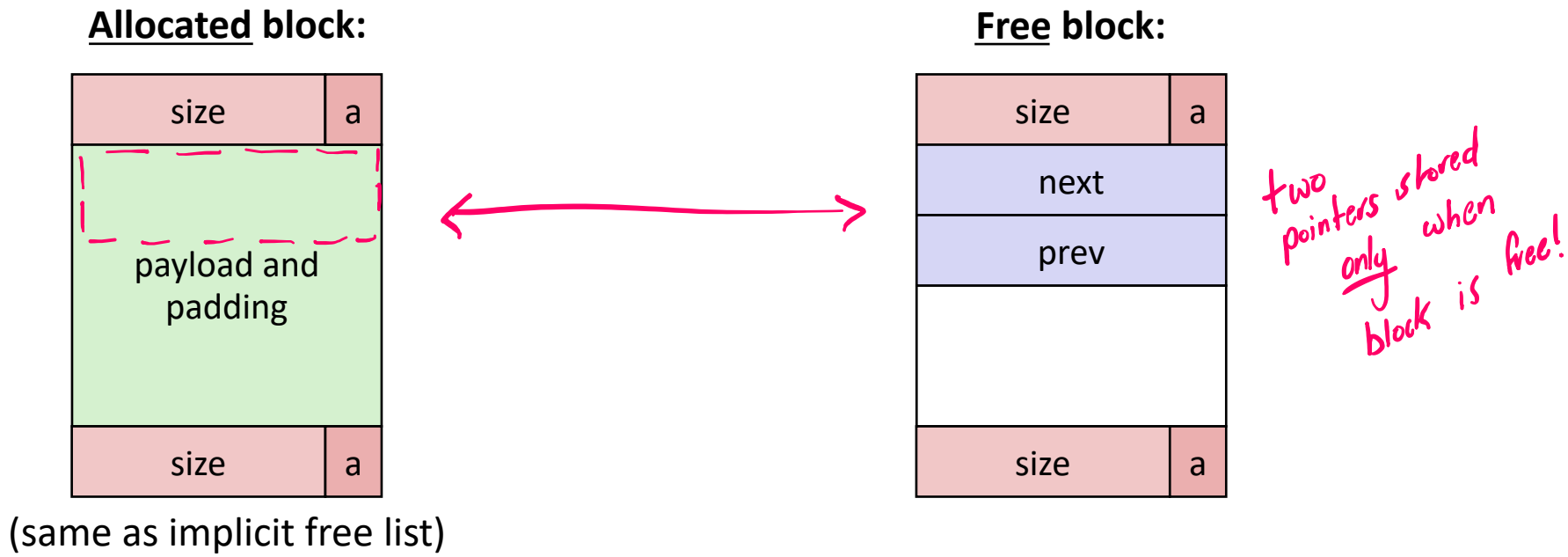
2) *Explicit free list* among only the free blocks, using pointers



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

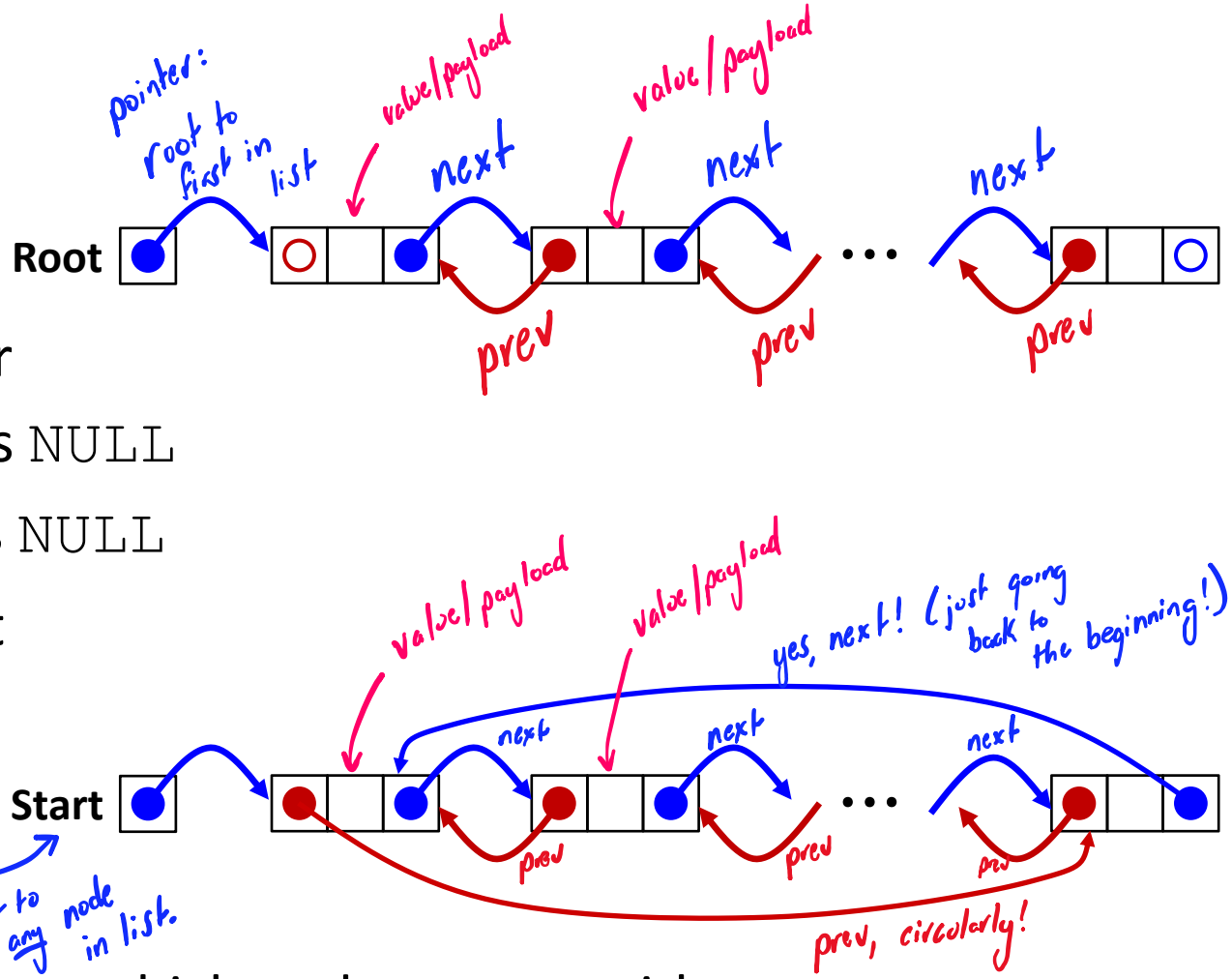
4) *Blocks sorted by size*
   - Can use a balanced binary tree (*e.g.*, red-black tree) with pointers within each free block, and the length used as a key

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 with pointers when a block is free, we can use the payload "space" for pointers
    - In Lab 5, it'll be a bit different. All info: size, allocated bit, pointers are stored in a `struct`
  - 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

especially

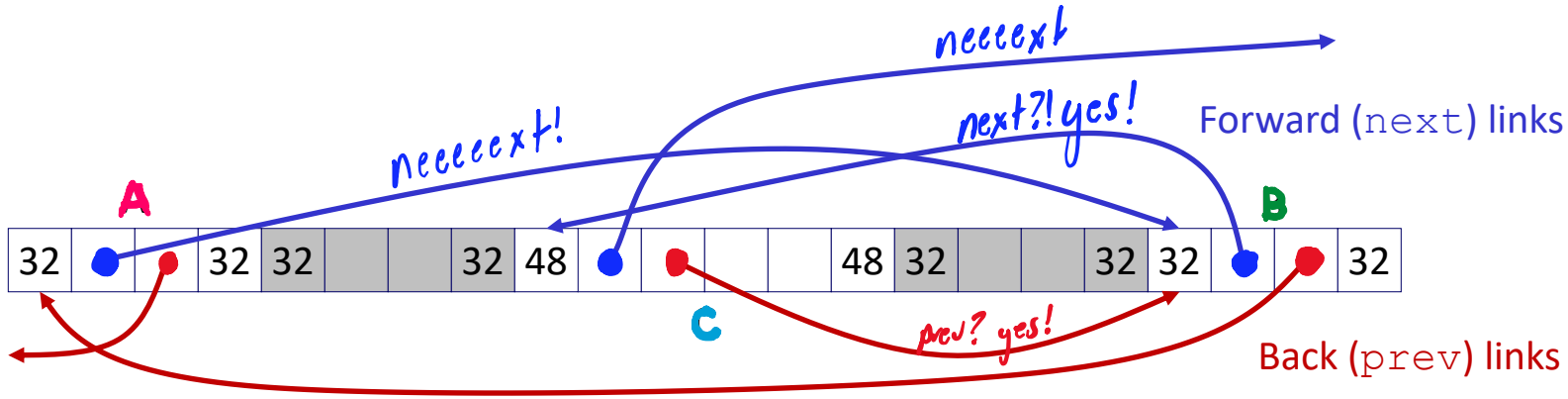
- **Circular**
  - Still have pointer to tell you which node to start with
  - No `NULL` pointers (term condition is back at starting point)
  - Good for next-fit, best-fit

could theoretically point to any node in list.

Explicit Free Lists

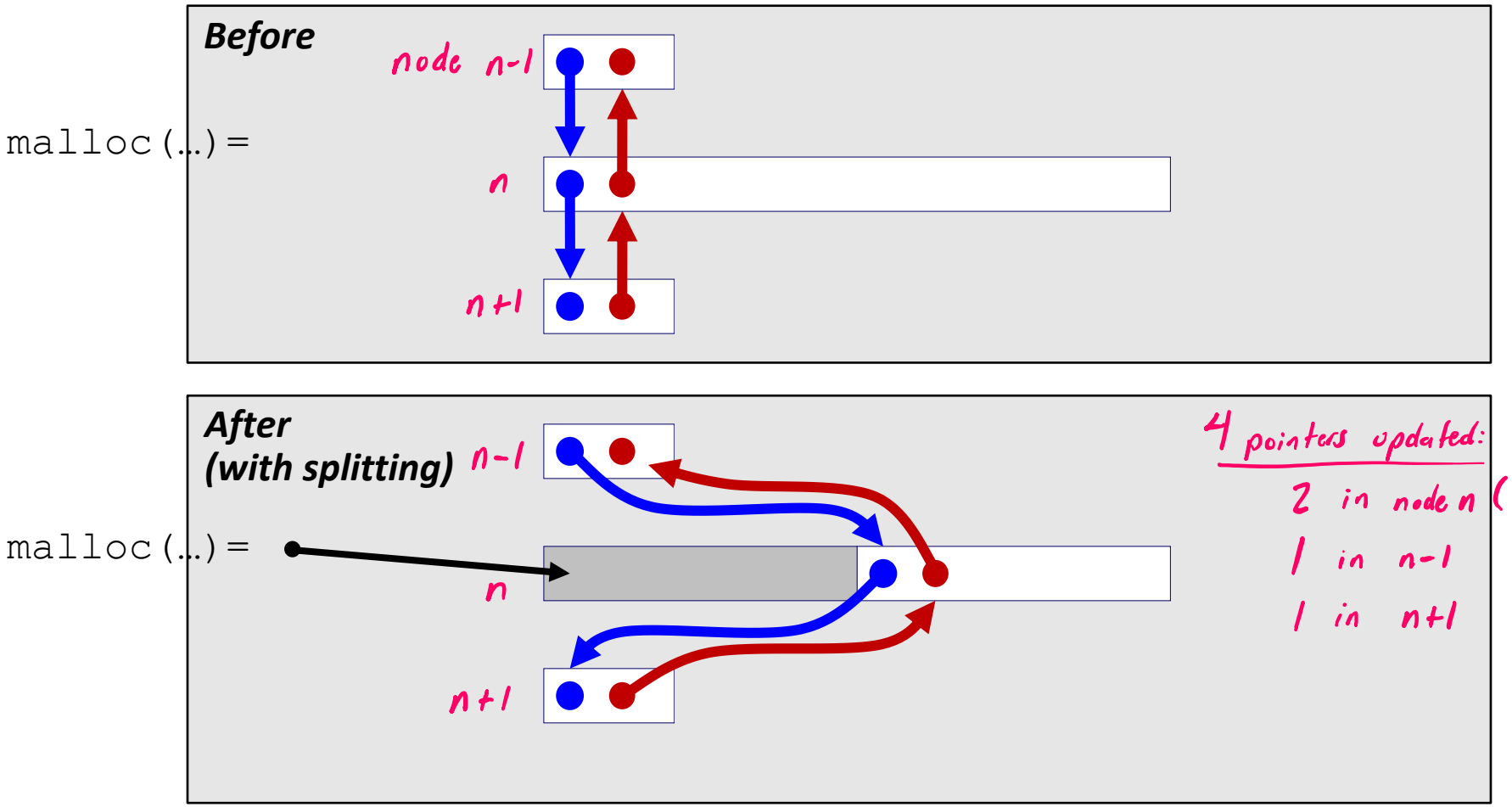
- **Logically:** doubly-linked list

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



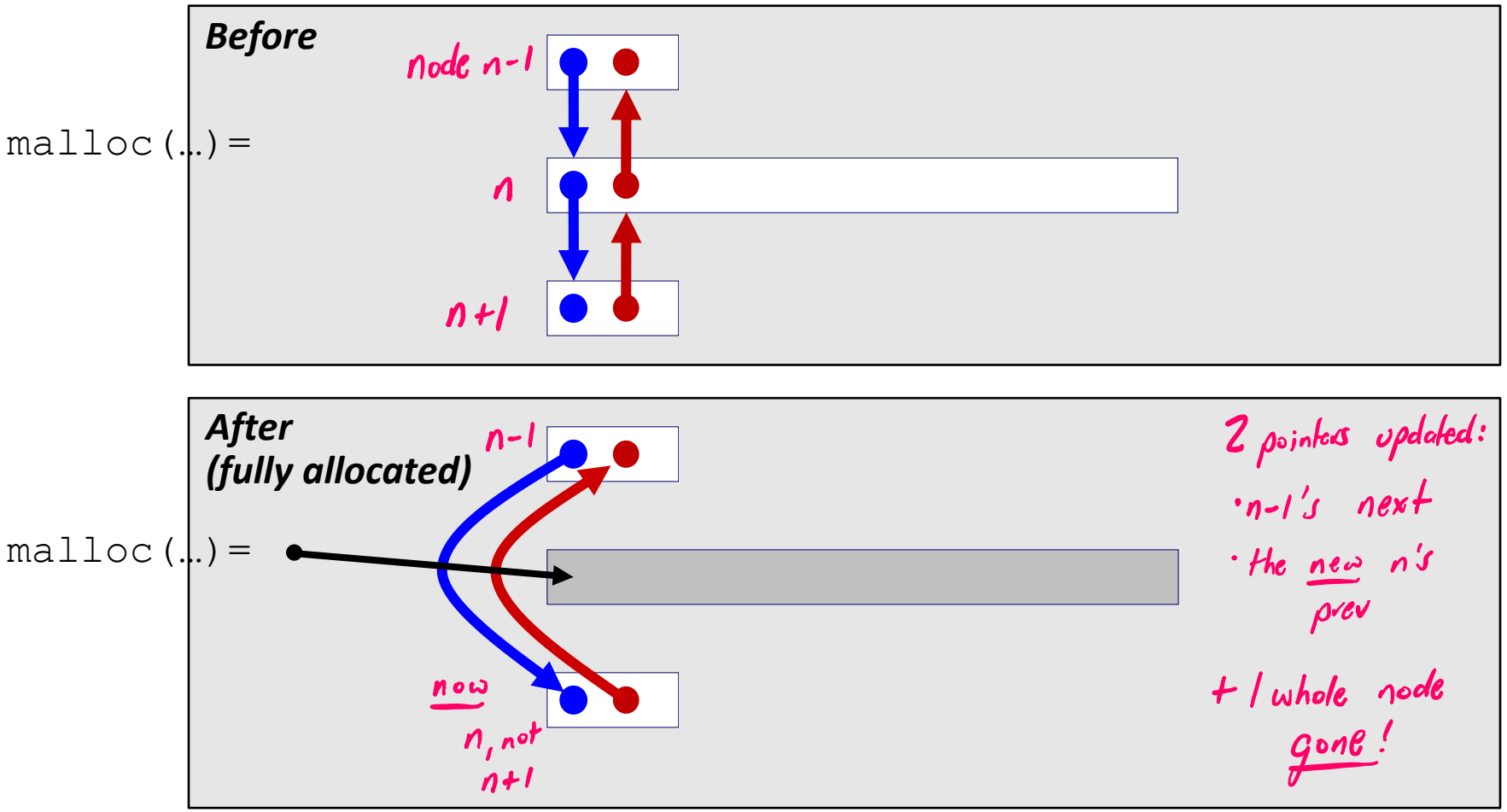
Allocating From Explicit Free Lists Splitting Version

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
Full Allocation Version

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).



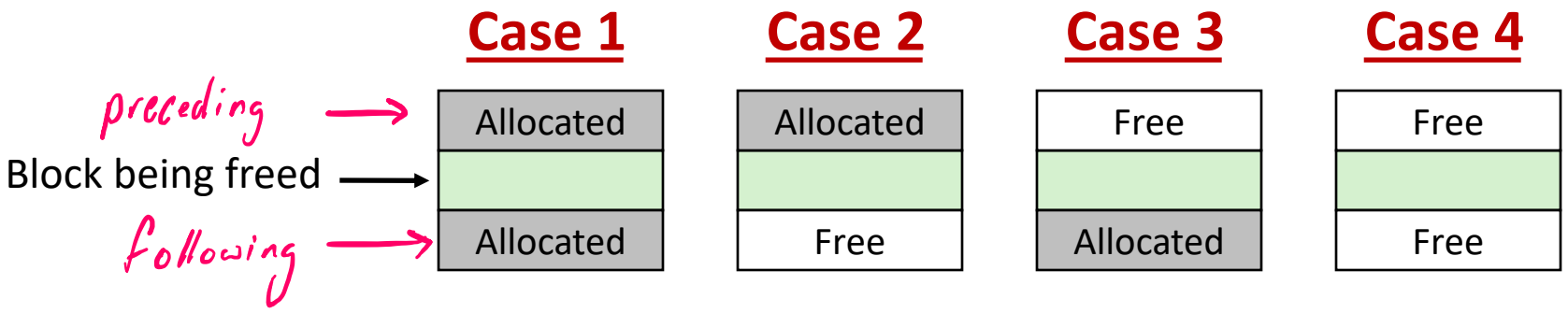
One whole fewer node in the list!

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)
    - Pro: studies suggest fragmentation is better than the alternative
    - Con: requires linear-time search

Coalescing in Explicit Free Lists



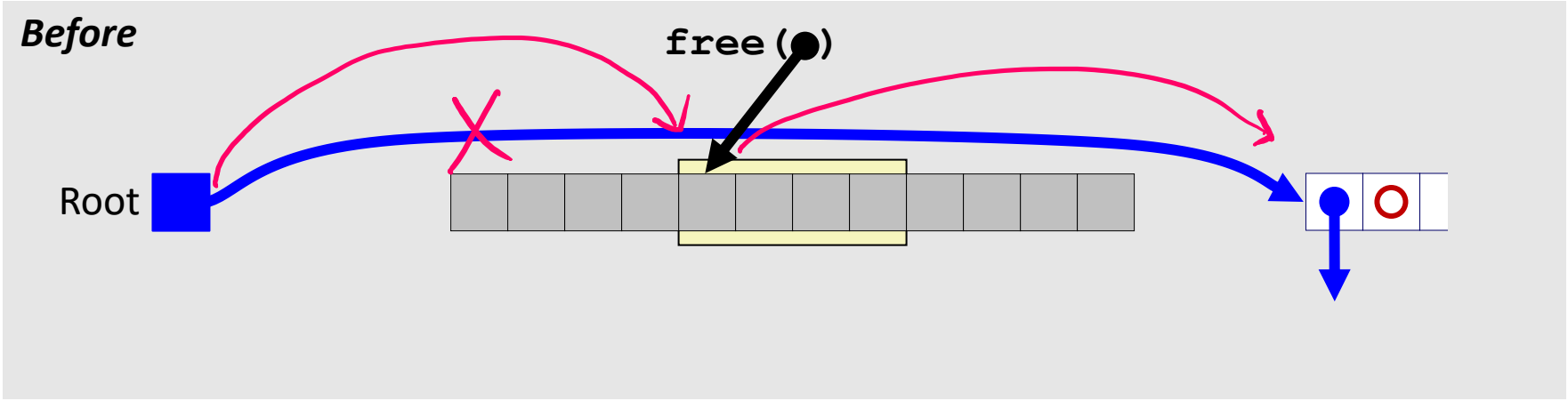
- Neighboring free blocks are already part of the free list
  1) Remove old block from free list
  2) Create new, larger coalesced block
  3) Add new block to free list (insertion policy)

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

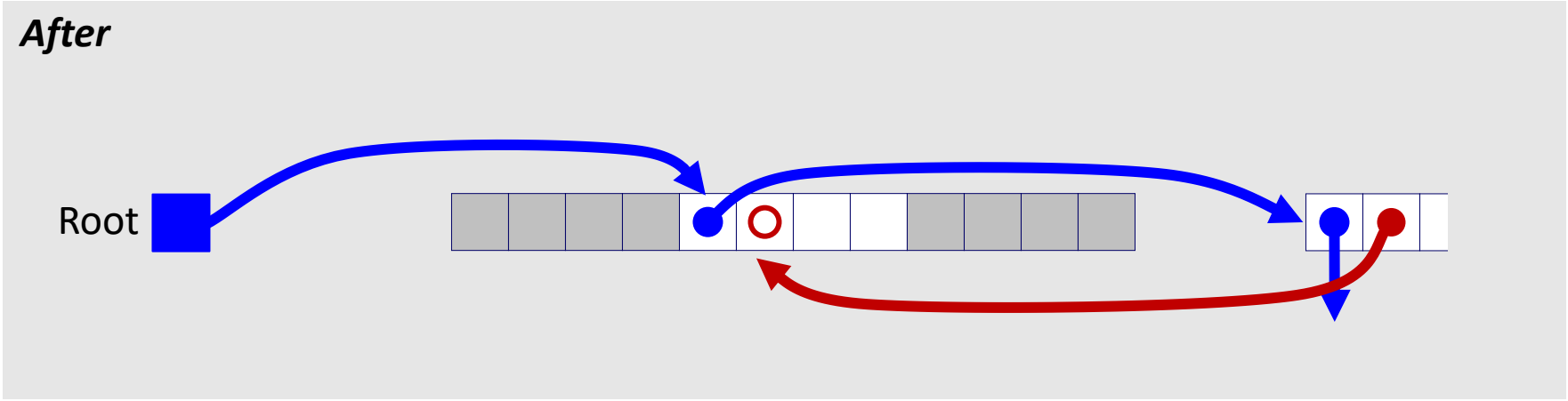
Can search list... but can also use boundary tags!

Freeing with LIFO Policy (Case 1)

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

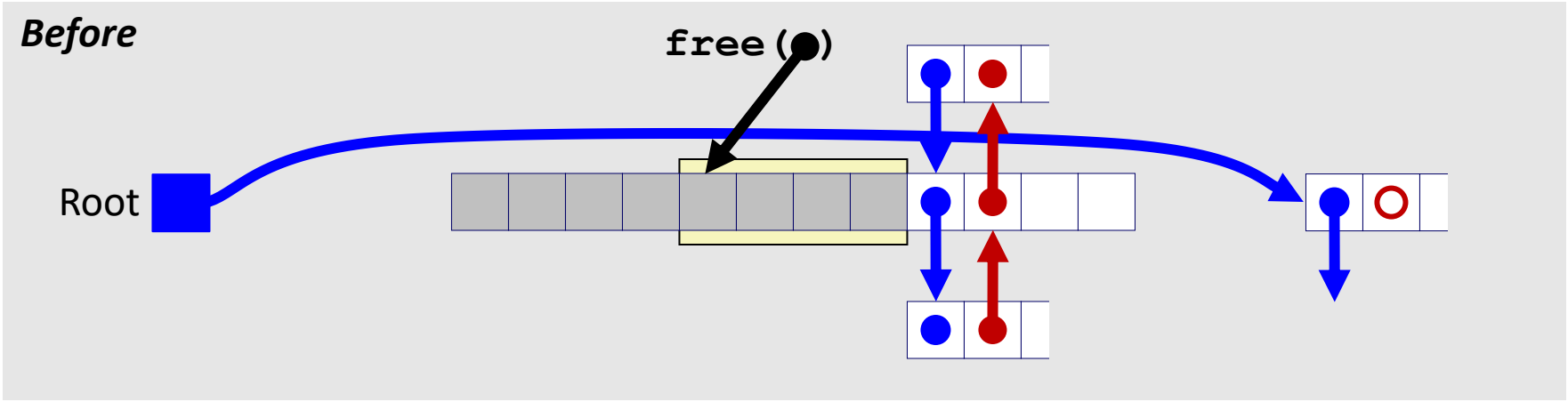


- Insert the freed block at the root of the list

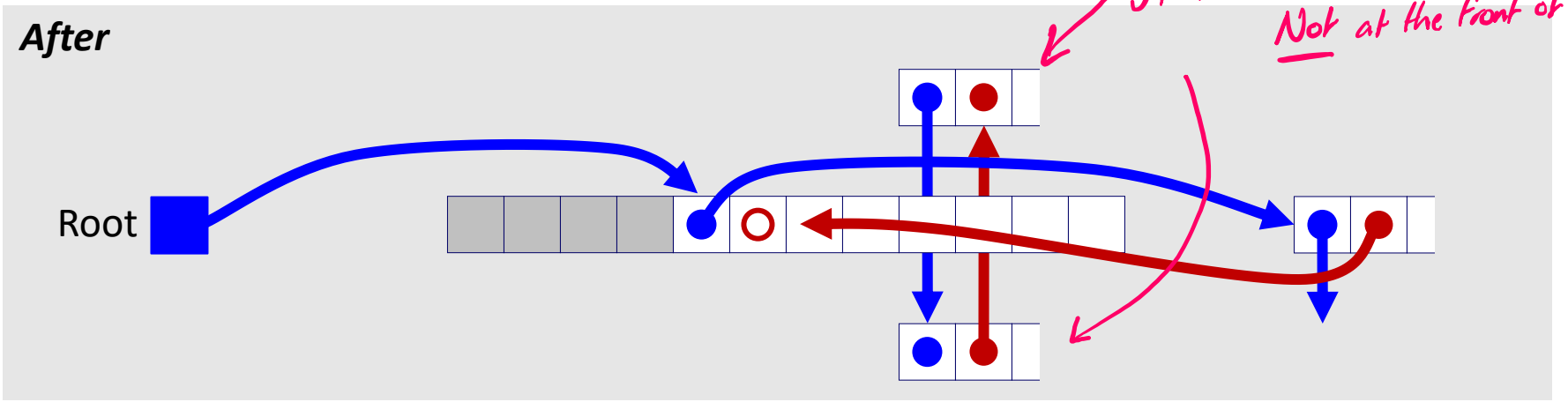


Freeing with LIFO Policy (Case 2)

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

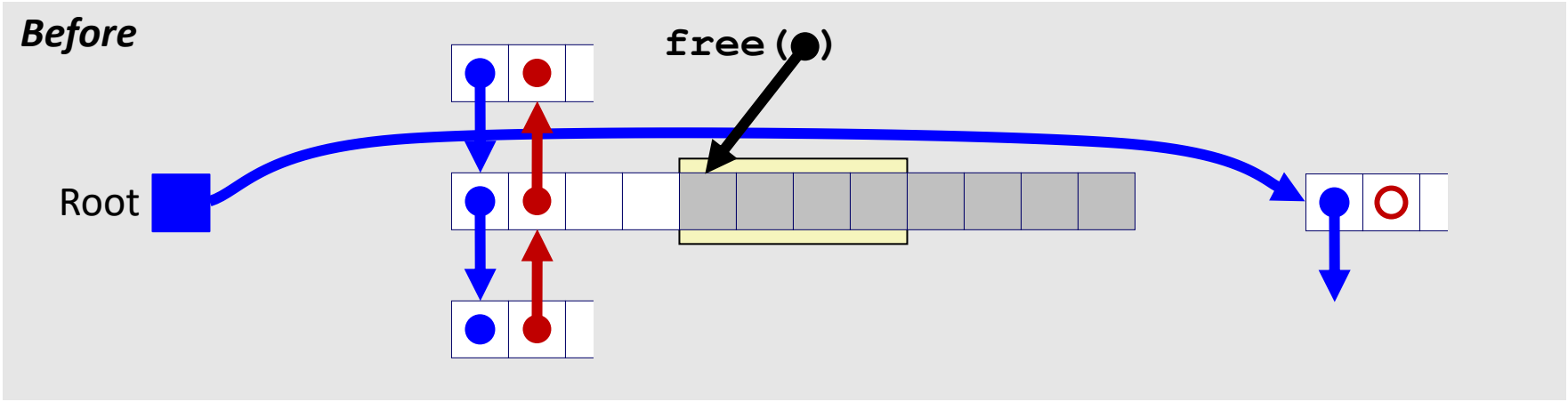


- Splice *following* block out of list, coalesce both memory blocks, and insert the new block at the root of the list

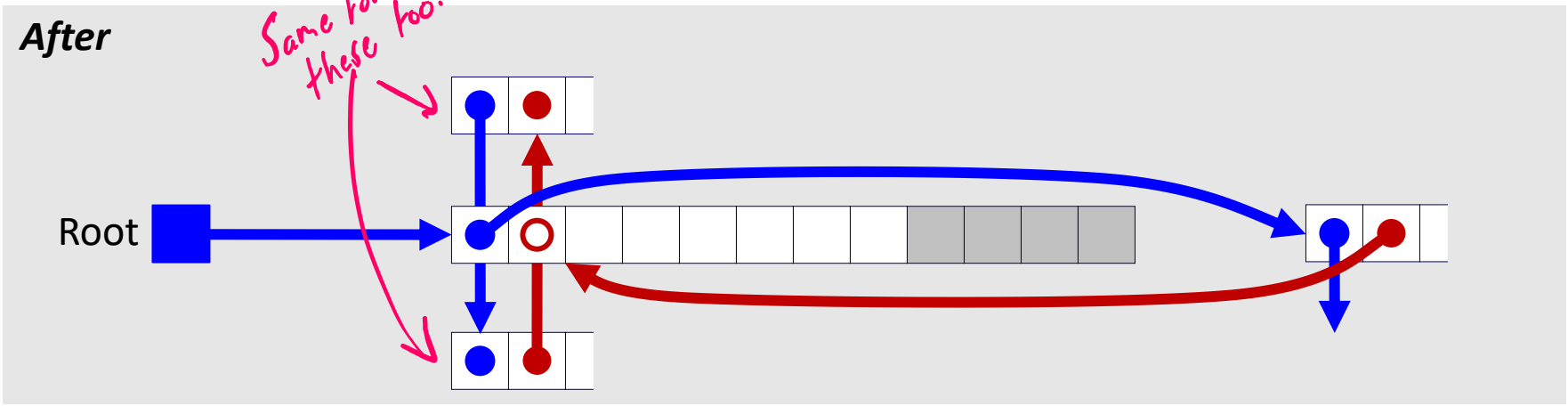


Freeing with LIFO Policy (Case 3)

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

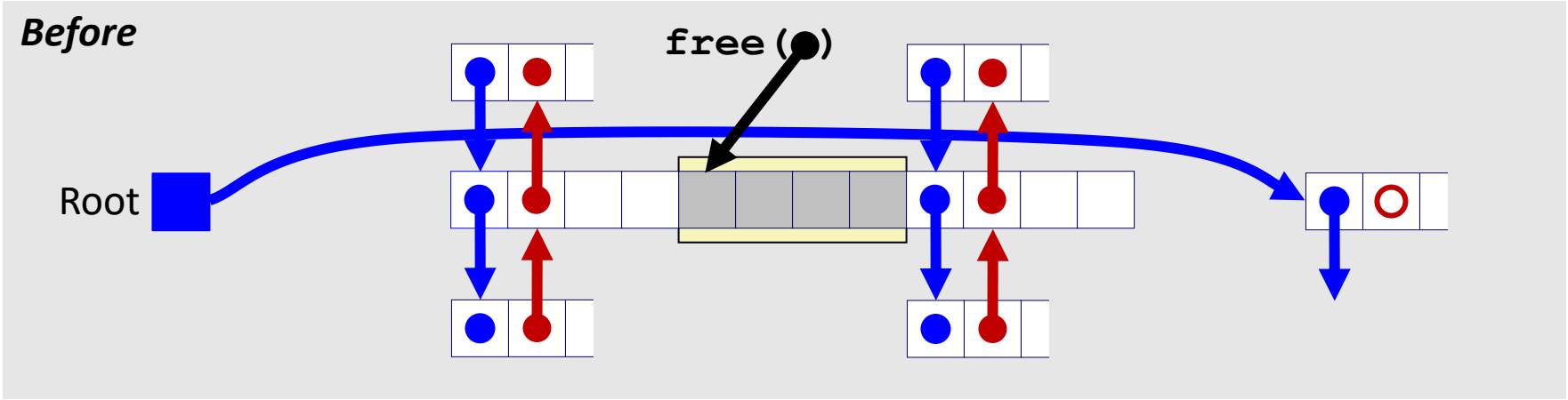


- Splice *preceding* block out of list, coalesce both memory blocks, and insert the new block at the root of the list

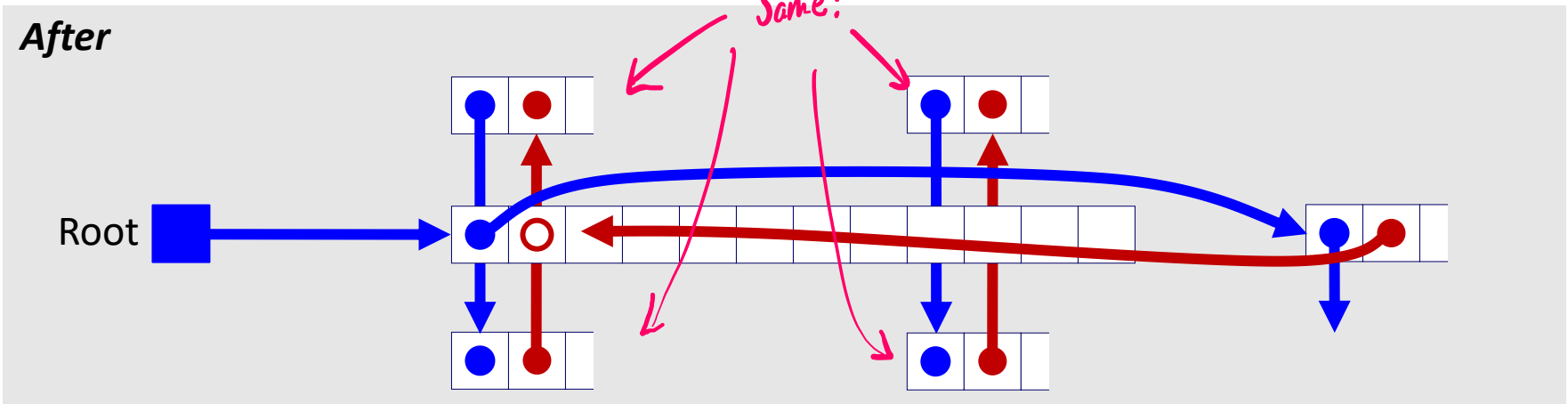


Freeing with LIFO Policy (Case 4)

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

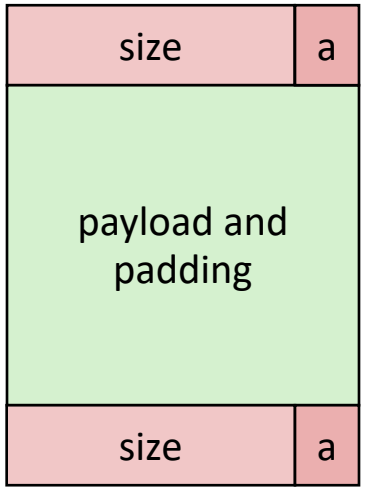


- Splice *preceding* and *following* blocks out of list, coalesce all 3 memory blocks, and insert the new block at the root of the list



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 (list is small!)
  - 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 → only when free!
- 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