

# Memory Allocation II

CSE 351 Spring 2019

## Instructor:

Ruth Anderson

## Teaching Assistants:

Gavin Cai

Jack Eggleston

John Feltrup

Britt Henderson

Richard Jiang

Jack Skalizky

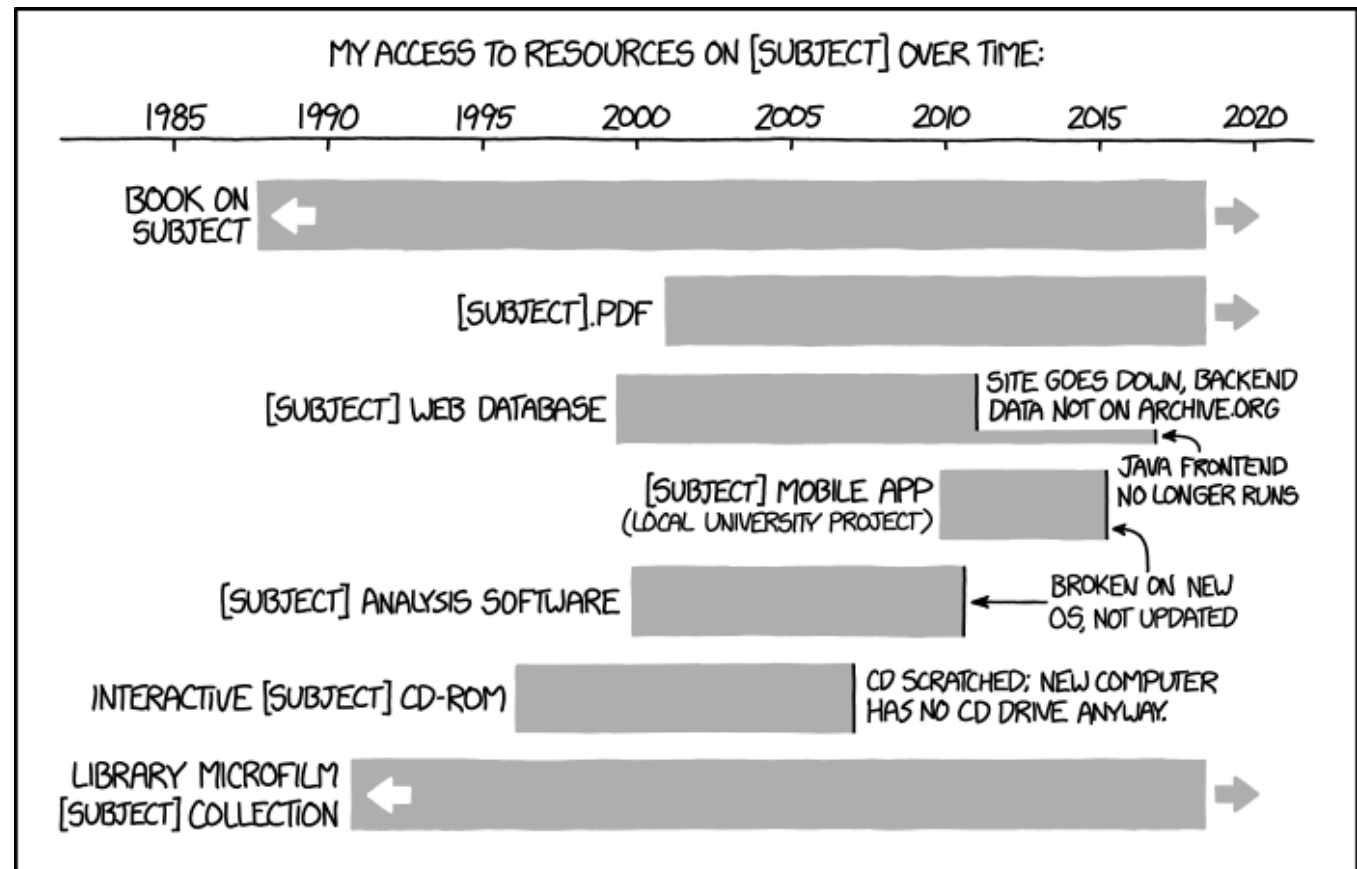
Sophie Tian

Connie Wang

Sam Wolfson

Casey Xing

Chin Yeoh



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

<http://xkcd.com/1909/>

# Administrivia

- ❖ Homework 5, due Friday (5/31)
  - Processes and Virtual Memory
- ❖ Lab 5, released Wed Evening, due Friday (6/7)
  - Memory Allocation
  - Recommended that you watch the Lab 5 helper videos
  - Sunday 6/9 is last day Lab 5 may be submitted (if one late day is used)
- ❖ **Final Exam:** Wed, 6/12, 12:30-2:20 pm in KNE 130

# Peer Instruction Question

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

- <http://pollev.com/rea>

(A) Best-fit:

`malloc(50), malloc(50)`

(B) First-fit:

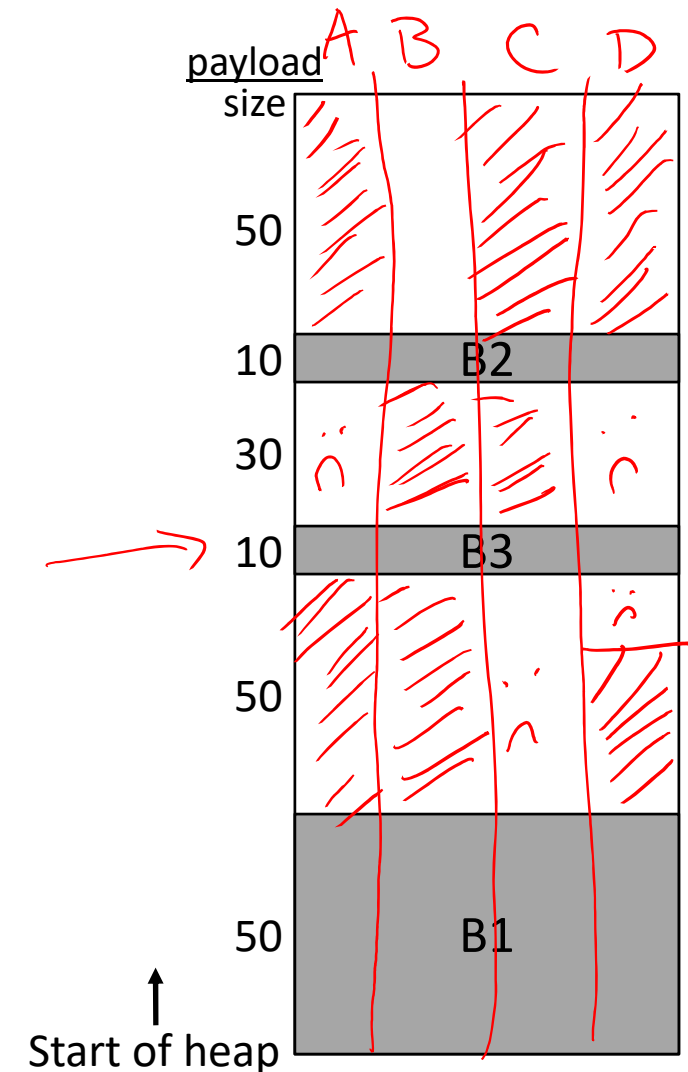
`malloc(50), malloc(30)`

(C) Next-fit:

`malloc(30), malloc(50)`

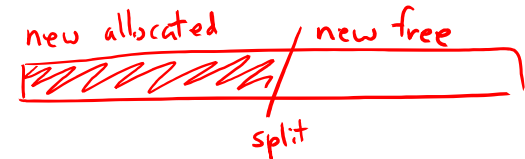
(D) Next-fit:

`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

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

```

void split(ptr b, int bytes) {
  ① int newsize = ((bytes+15) >> 4) << 4; // bytes = desired block size
  ② int oldsize = *b; // round up to multiple of 16
  ③ *b = newsize; // why not mask out low bit?
  ④ if (newsize < oldsize) // initially unallocated
  ⑤ *(b+newsize) = oldsize - newsize; // set length in remaining
  } // part of block (UNSCALED +)

```

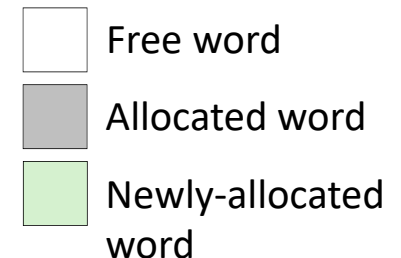
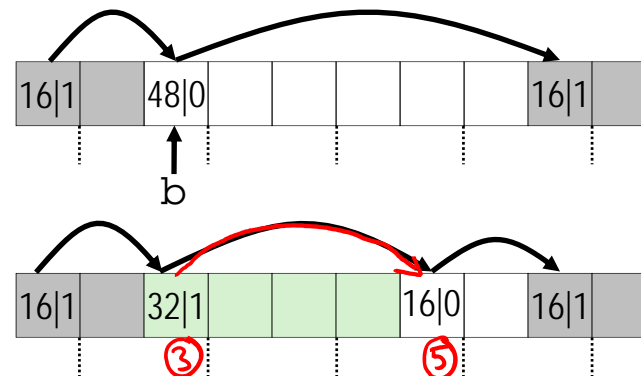
```

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

```

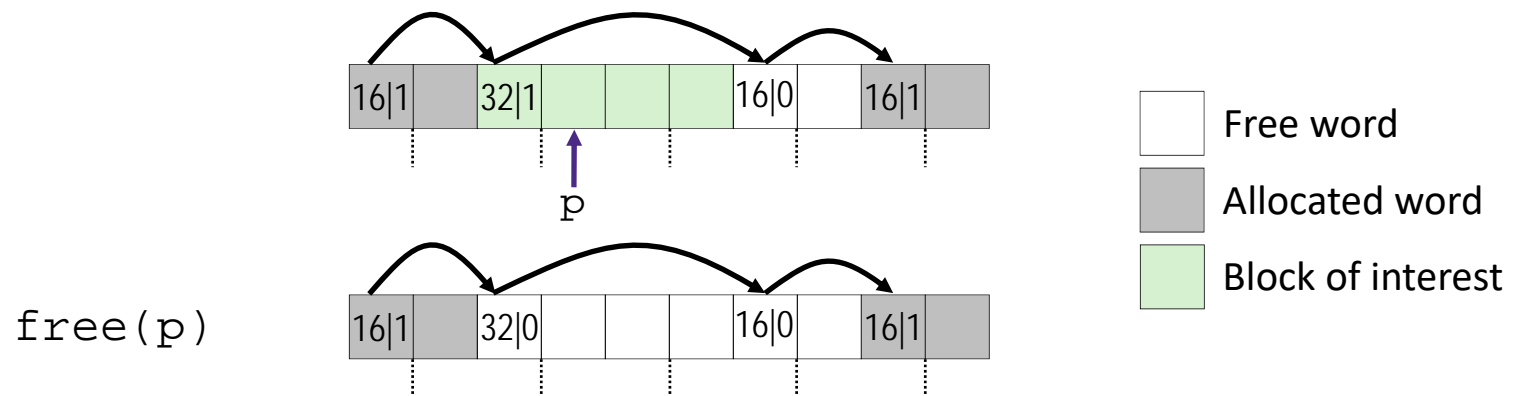
↑ sets  $a=1$

header



# Implicit List: Freeing a Block

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

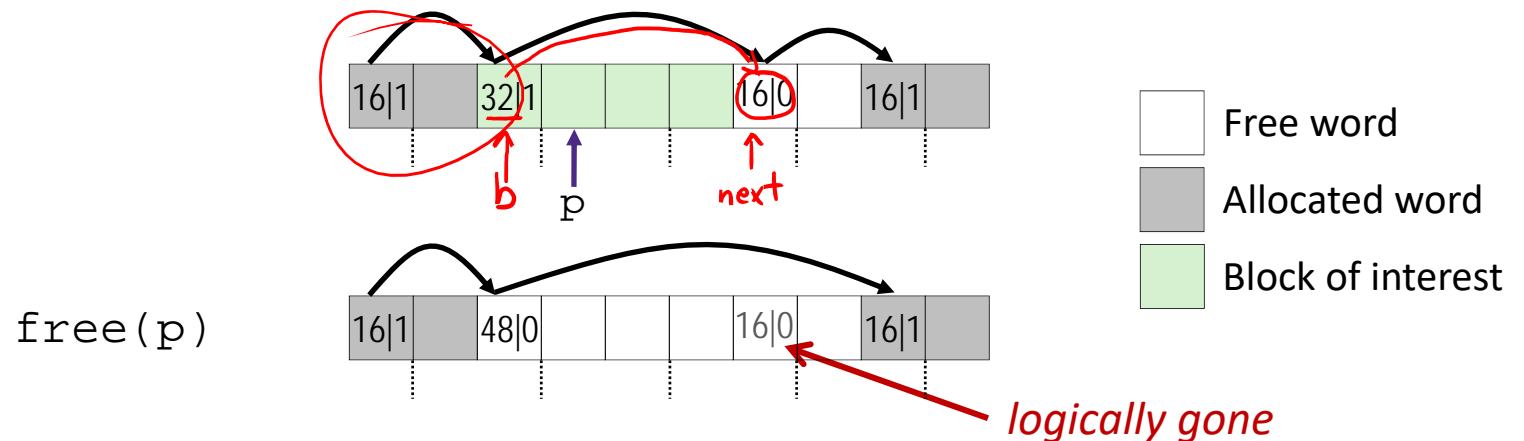


`malloc(40)`

*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) {
    ptr b = p - 8;           // p points to payload
    *b &= -2;                // b points to block header
    ptr next = b + 32;        // clear allocated bit
    if ((*next & 1) == 0)    // find next block (UNSCALED +)
        *b += *next;         // if next block is not allocated,
                             // add its size to this block
}

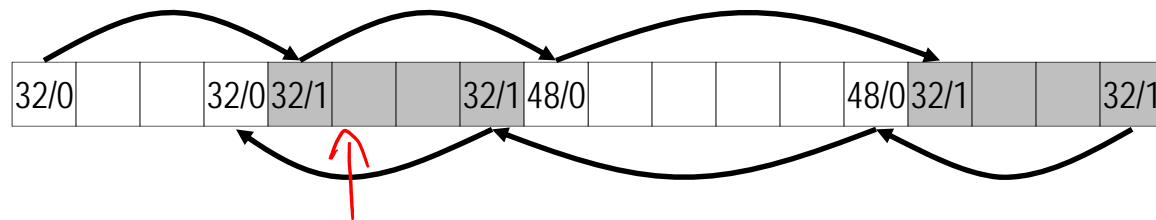
```

- ❖ How do we coalesce with the *previous* block? *we can't currently*

# 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!

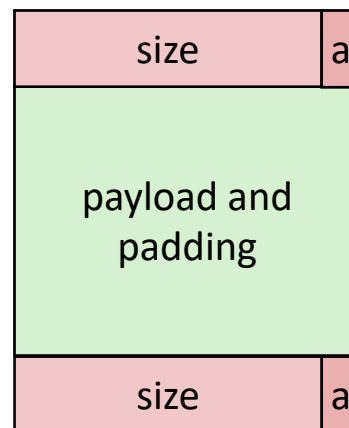


*Format of  
allocated and  
free blocks:*

Boundary tags

Header

Footer



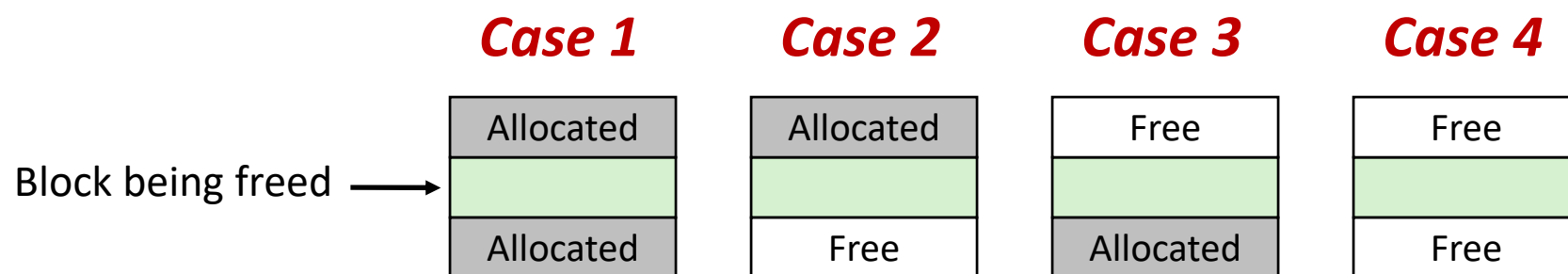
**a = 1:** allocated block

**a = 0:** free block

**size:** block size (in bytes)

**payload:** application data  
(allocated blocks only)

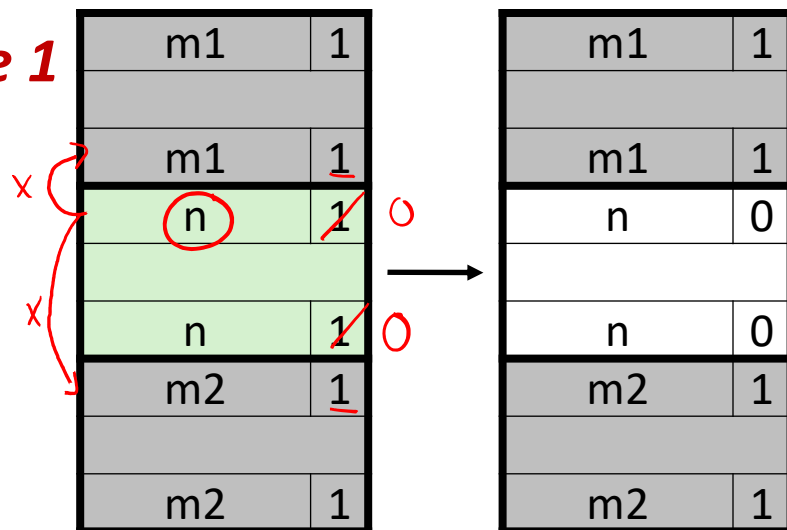
# Constant Time Coalescing



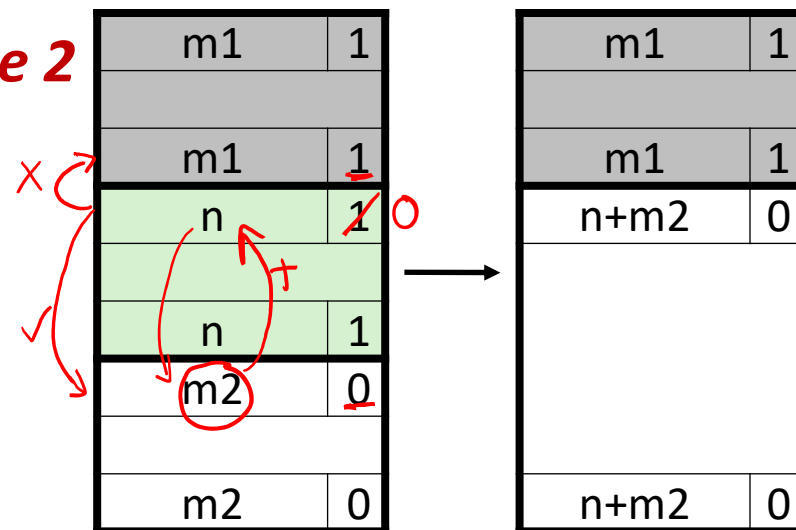


# Constant Time Coalescing

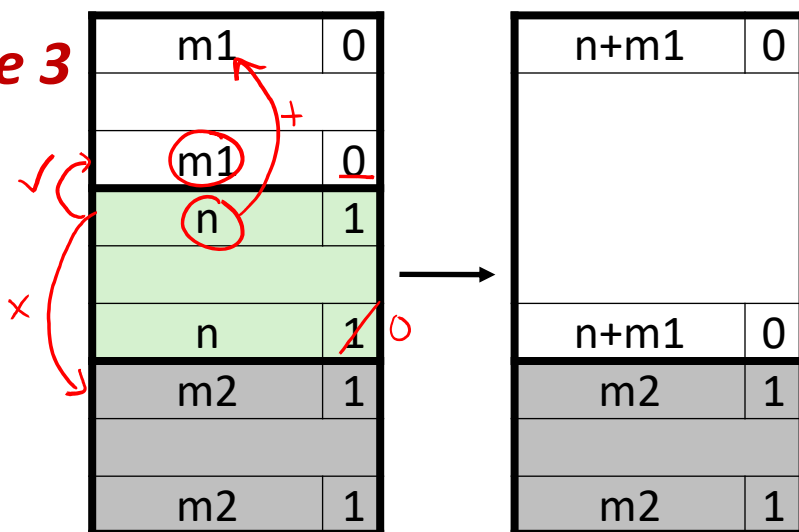
**Case 1**



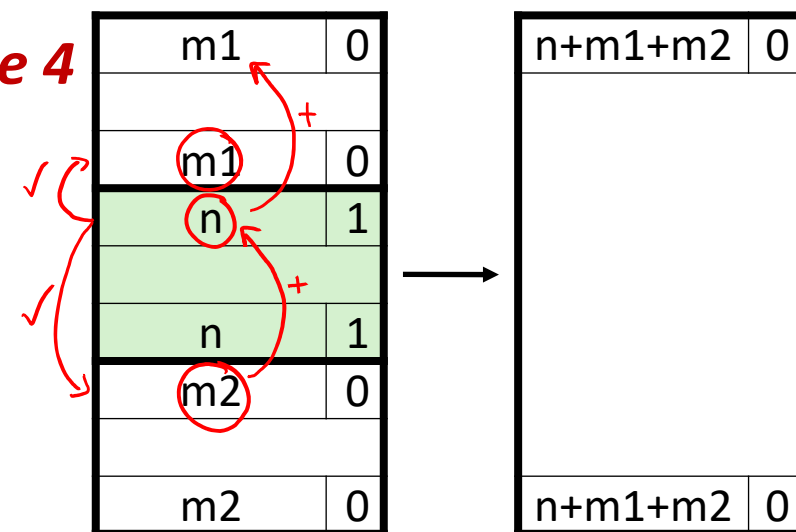
**Case 2**

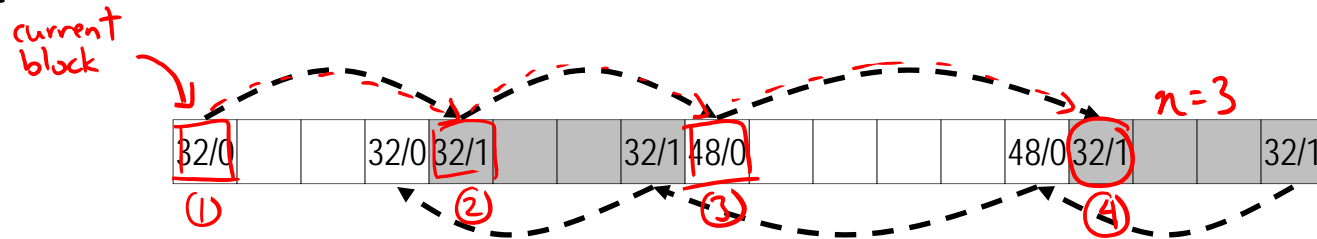


**Case 3**

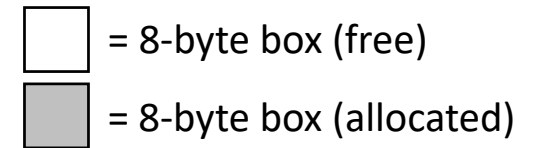


**Case 4**



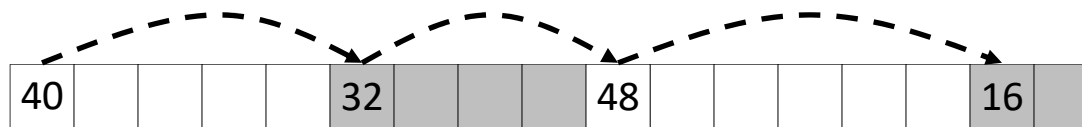


# Keeping Track of Free Blocks

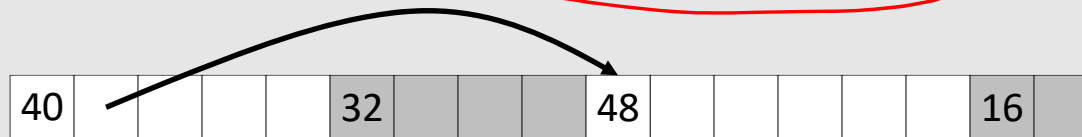


## 1) *Implicit free list* using length – links all blocks using math

- No actual pointers, and must check each block if allocated or free



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



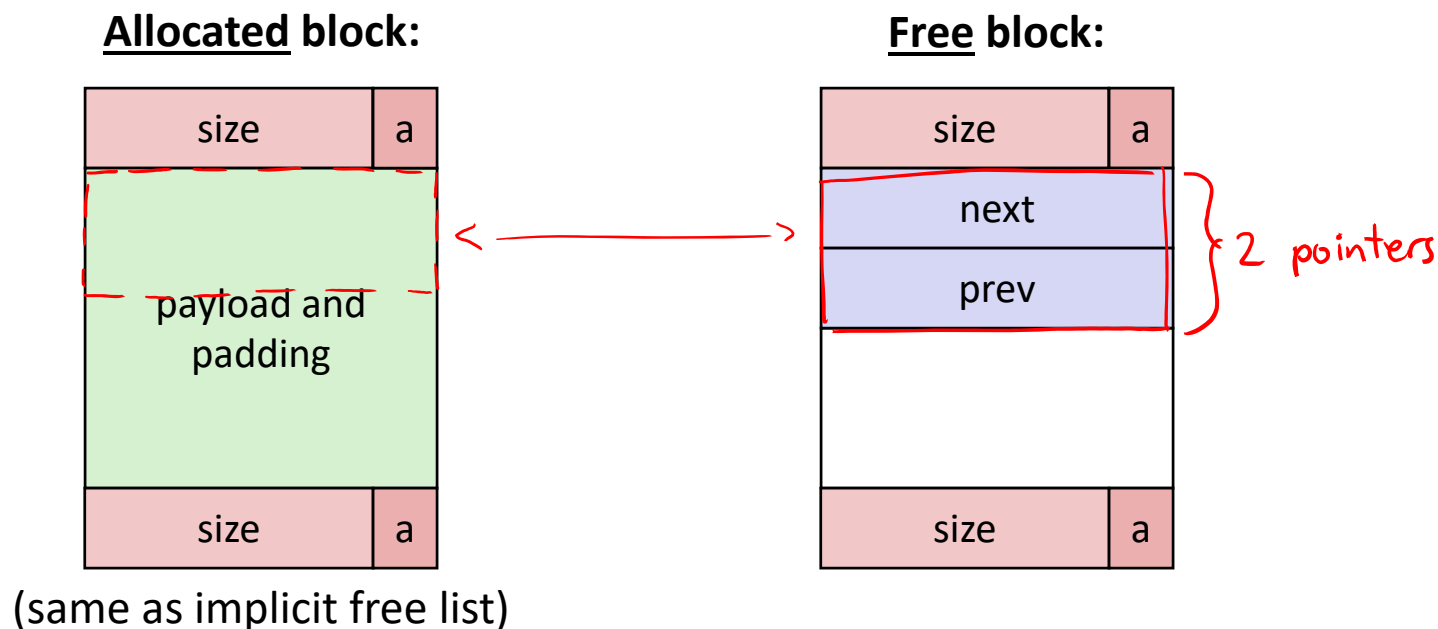
## 3) *Segregated free list*

- 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

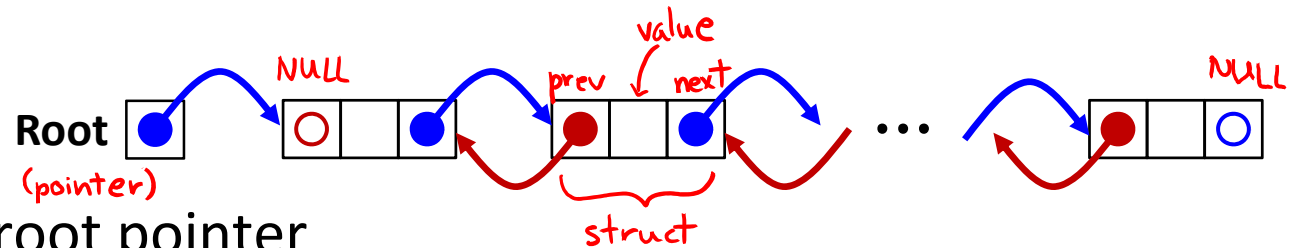


- ❖ Use list(s) of *free* blocks, rather than implicit list of *all* blocks
  - The “next” free block could be anywhere in the heap
    - So we need to store next/previous pointers, not just sizes
  - Since we only track free blocks, so we can use “payload” for pointers
  - Still need boundary tags (header/footer)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

# Doubly-Linked Lists

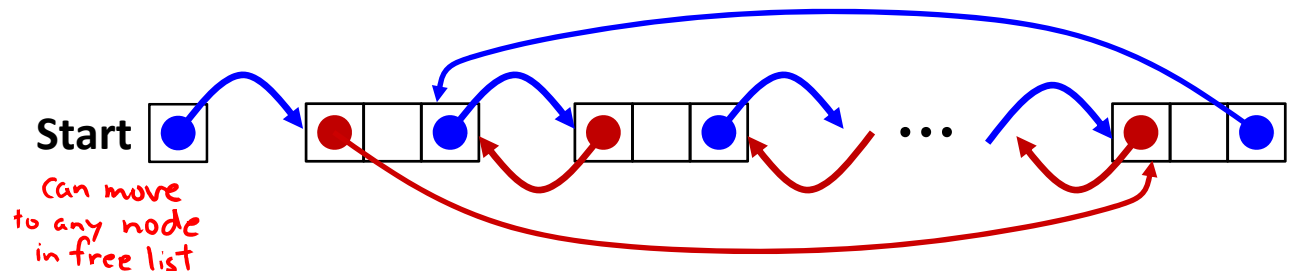
## ❖ Linear

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



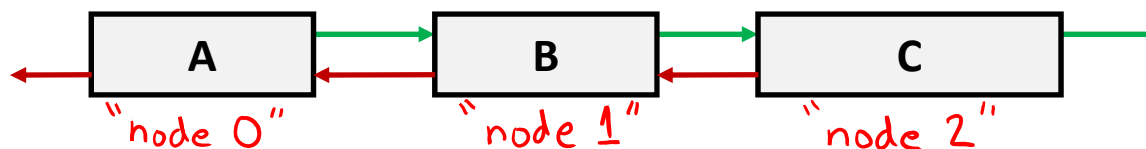
## ❖ Circular

- 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

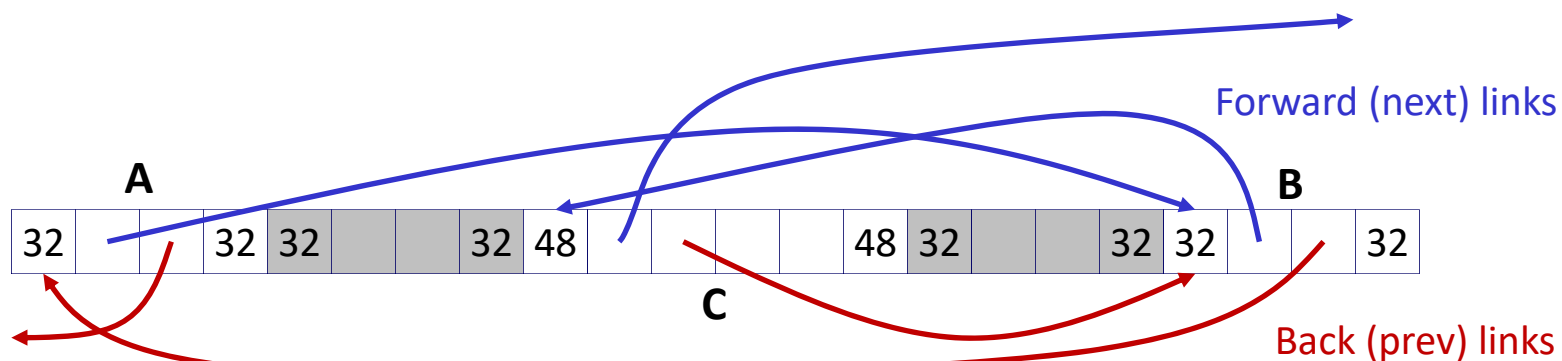


# Explicit Free Lists

- ❖ **Logically:** doubly-linked list

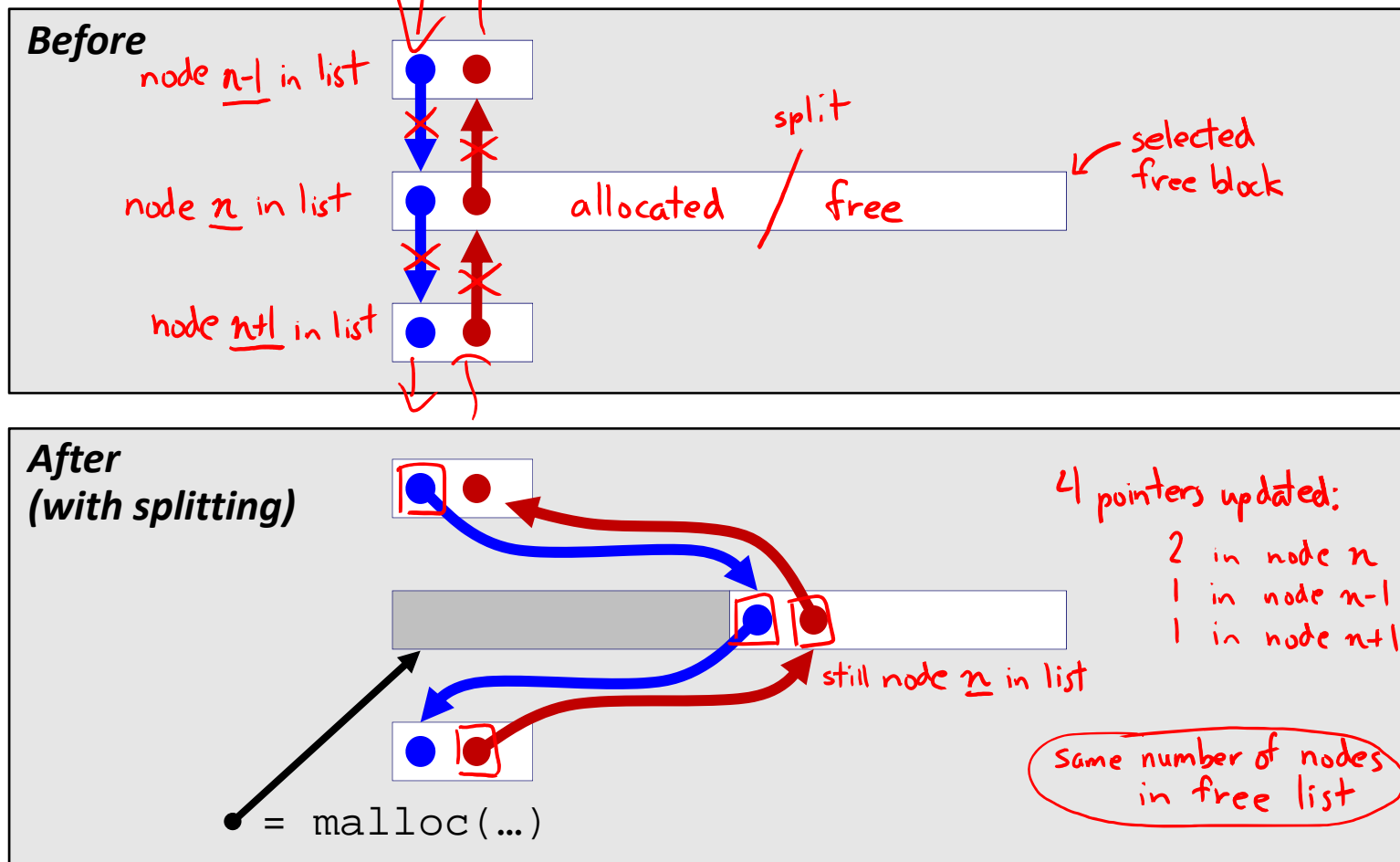


- ❖ **Physically:** blocks can be in any order



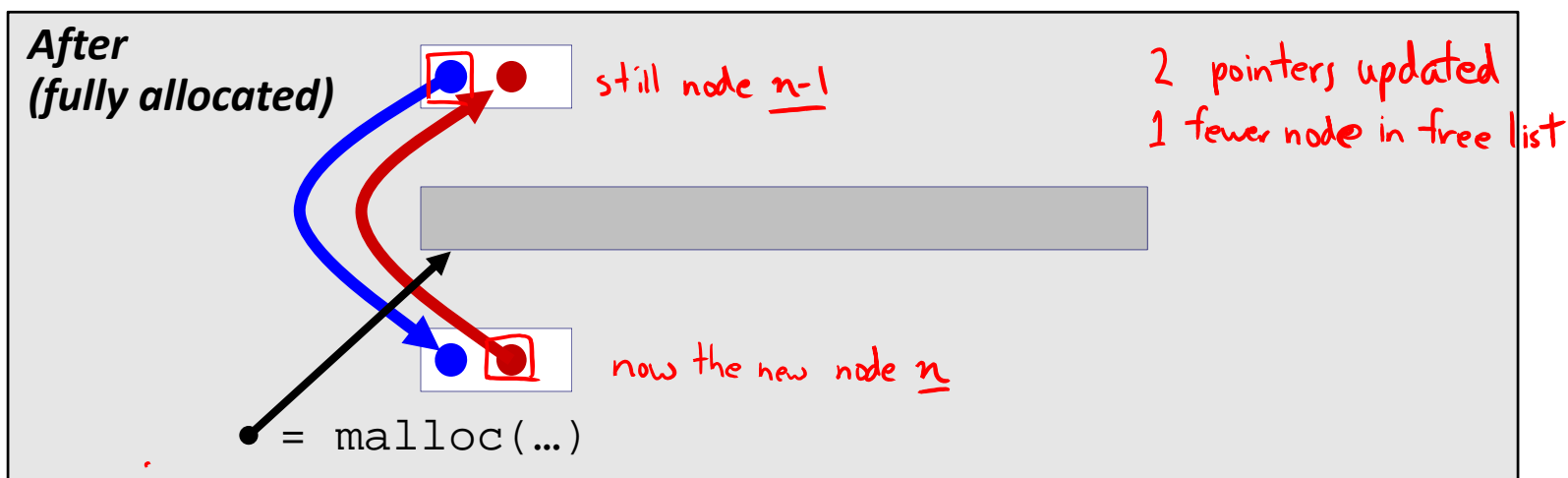
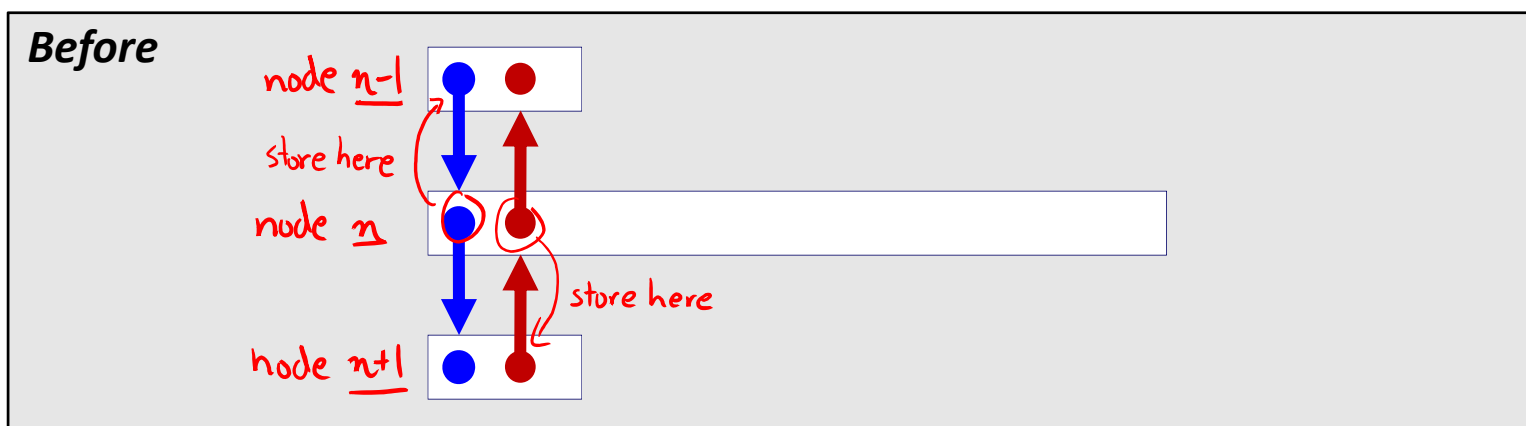
# Allocating From Explicit Free Lists

**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?

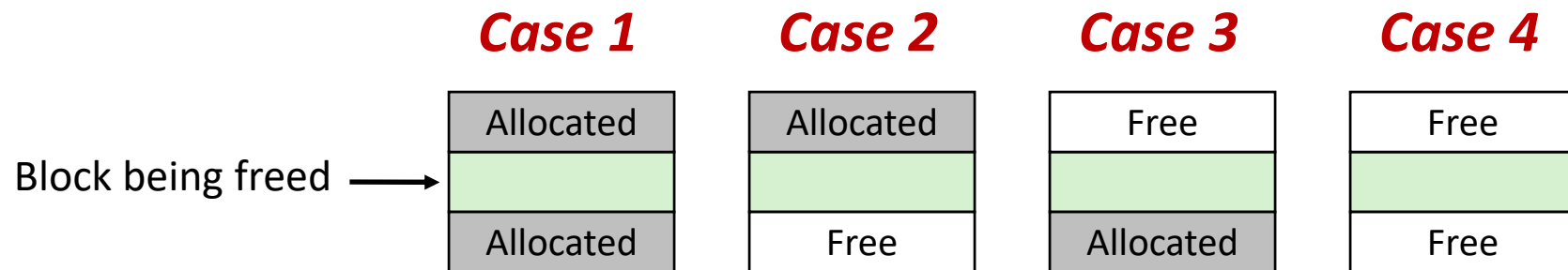
- ★ **LIFO (last-in-first-out) policy**

- 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:  
$$\text{address}(\text{previous}) < \text{address}(\text{current}) < \text{address}(\text{next})$$
- 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*

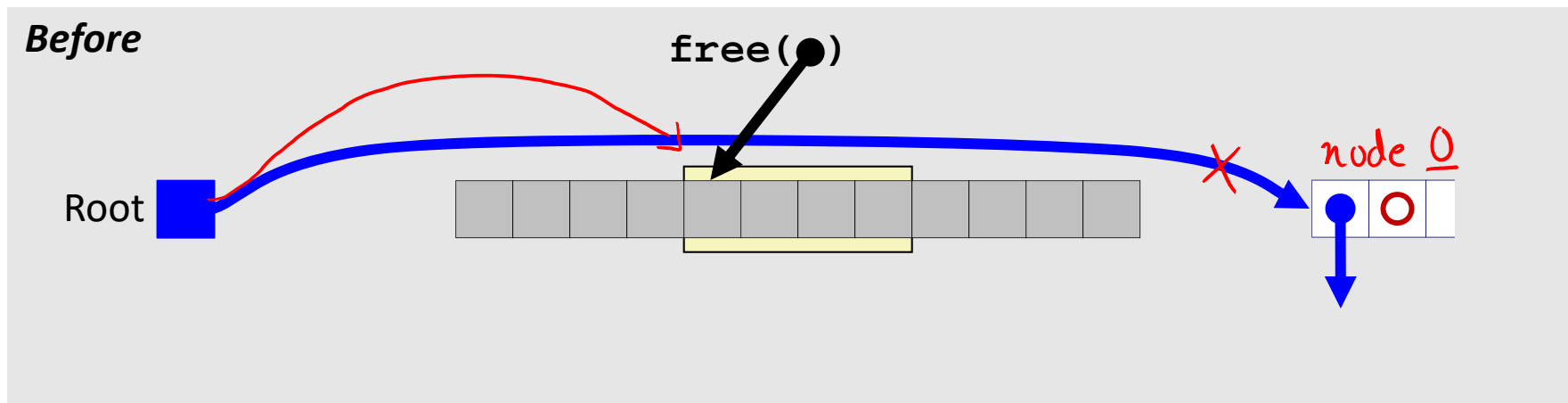
- 1) Remove old block from free list
- 2) Create new, larger coalesced block
- 3) Add new block to free list (insertion policy)

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

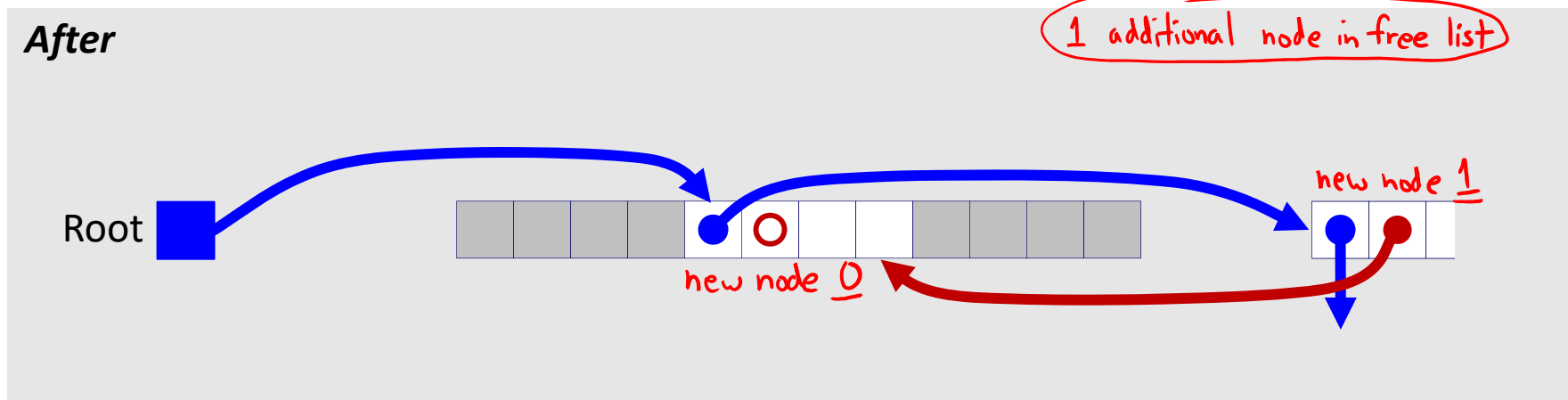
can still use boundary tags (don't need to search free list). other implementations possible (see Lab 5)

# Freeing with LIFO Policy (Case 1)

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

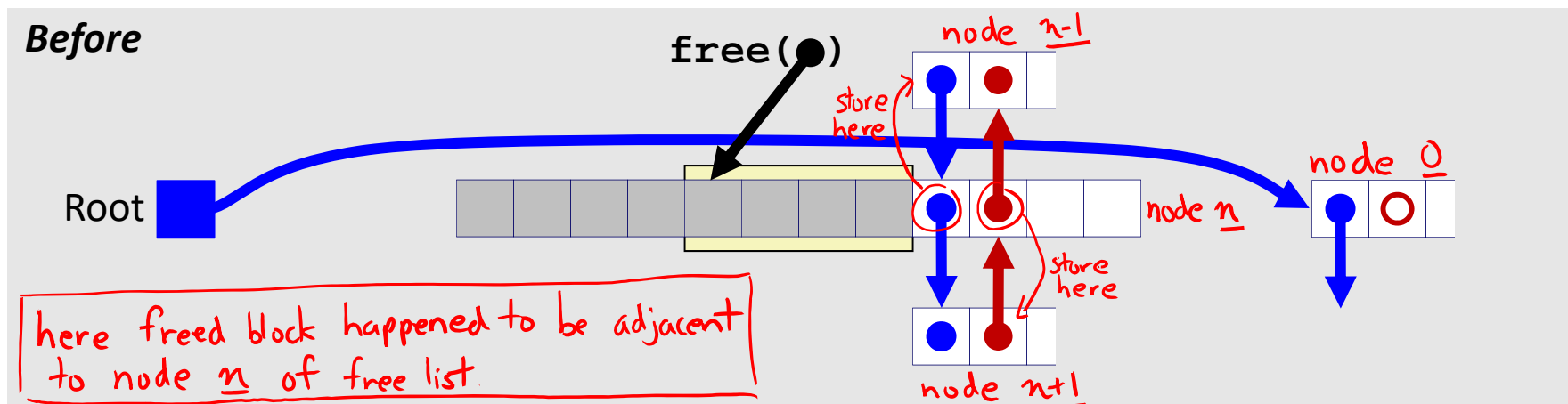


- ❖ Insert the freed block at the root of the list

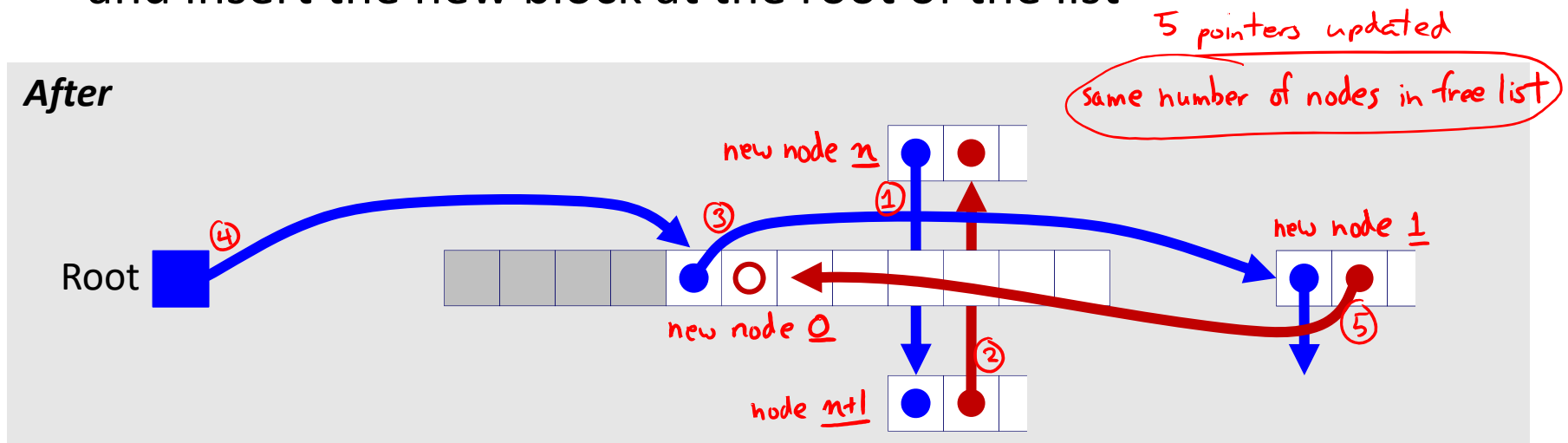


# Freeing with LIFO Policy (Case 2)

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

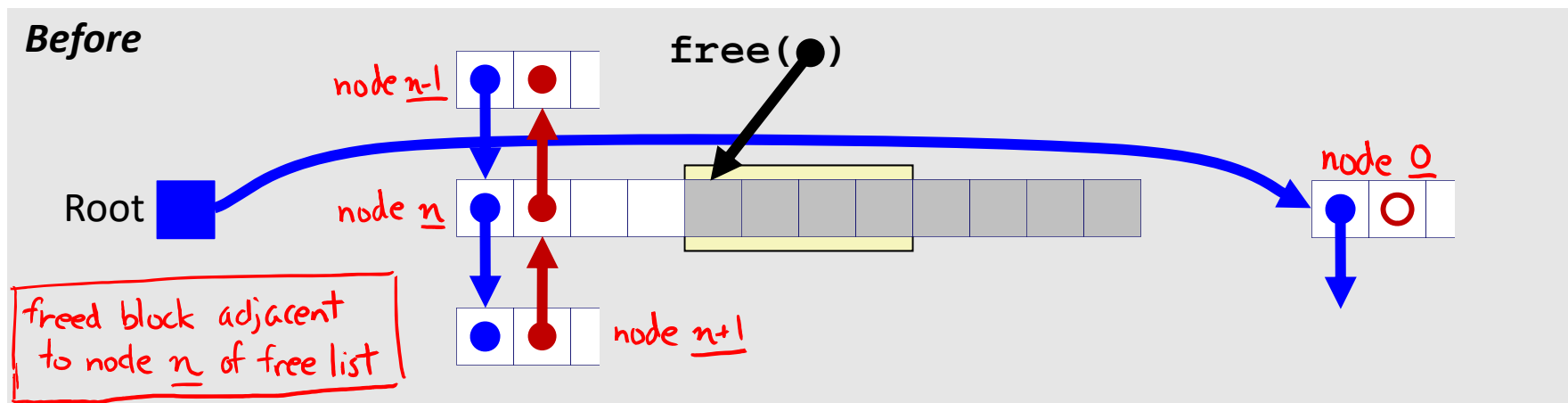


- ❖ Splice successor block out of list, coalesce both memory blocks, and insert the new block at the root of the list

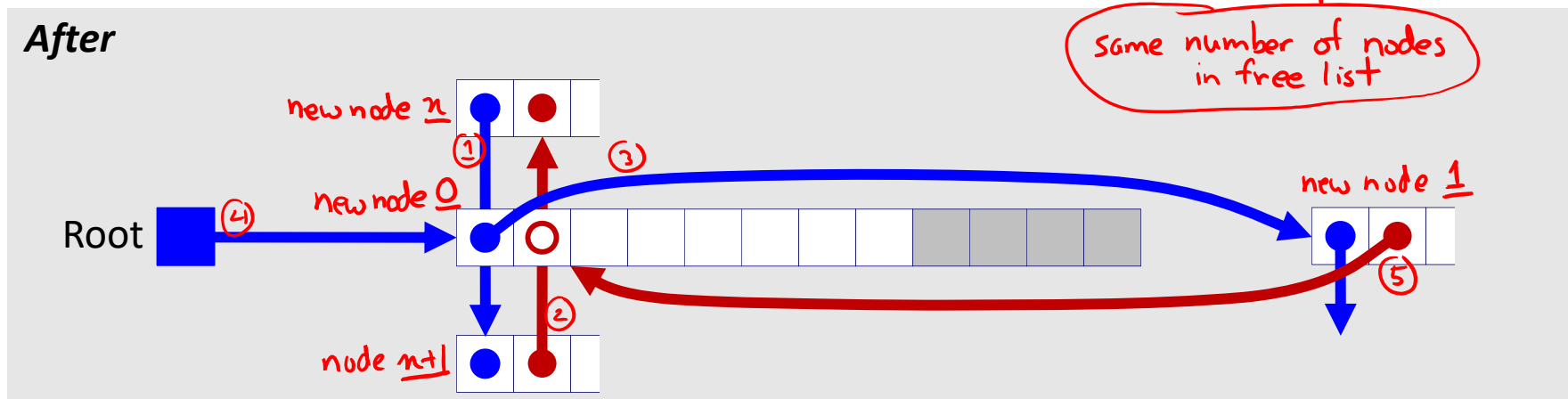


# Freeing with LIFO Policy (Case 3)

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

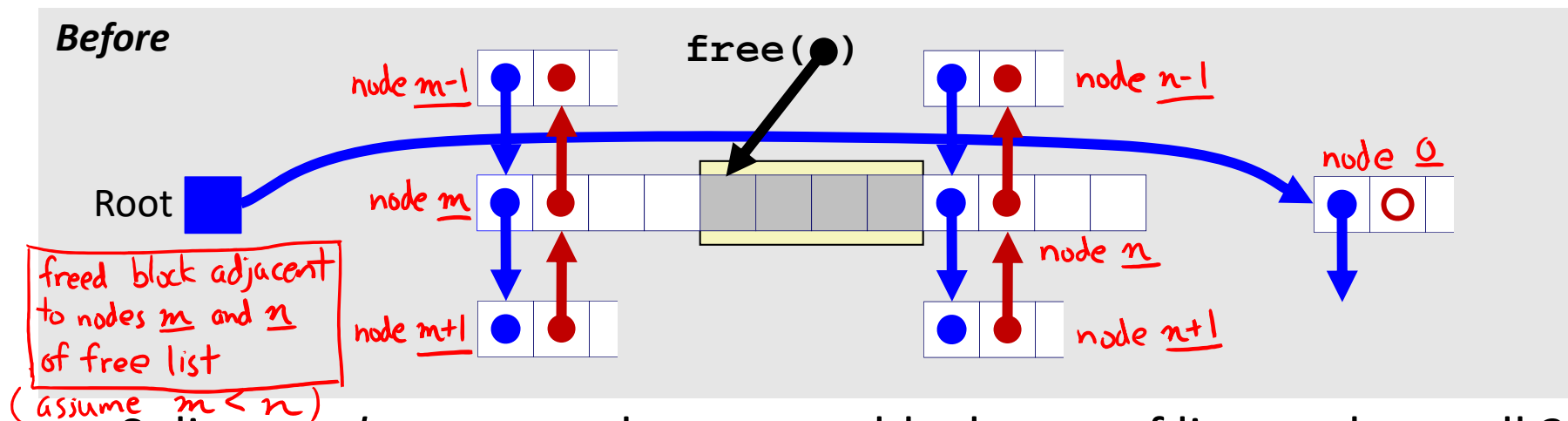


- ❖ Splice predecessor block out of list, coalesce both memory blocks, and insert the new block at the root of the list

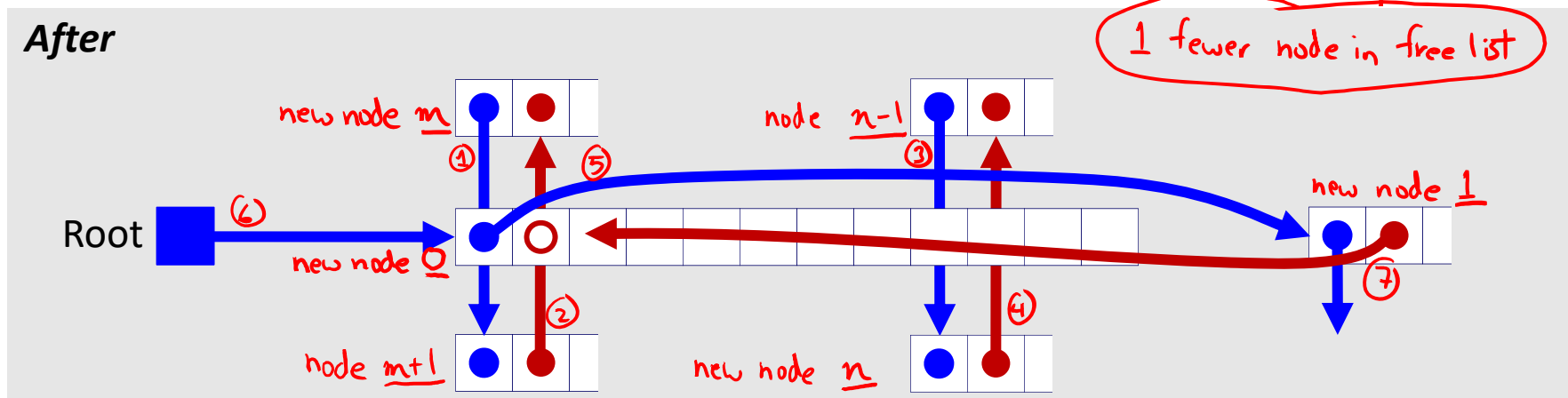


# Freeing with LIFO Policy (Case 4)

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

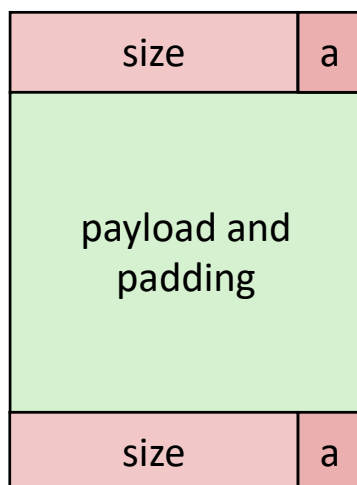


❖ Splice predecessor and successor blocks out of list, coalesce all 3 memory blocks, and insert the new block at the root of the list



# 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.

# 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		