

# Memory Allocation II

CSE 351 Autumn 2017

## Instructor:

Justin Hsia

## Teaching Assistants:

Lucas Wotton

Michael Zhang

Parker DeWilde

Ryan Wong

Sam Gehman

Sam Wolfson

Savanna Yee

Vinny Palaniappan

# Administrivia

- ❖ Homework 5 due Friday (12/1)
- ❖ Lab 5 due 12/8
  
- ❖ **Final Exam:** Wed, Dec. 13 @ 12:30pm in KNE 120
  - Same seating chart as Midterm
  - Review Session: Mon, Dec. 11 @ 5:00pm in EEB 105
  - Cumulative (midterm clobber policy applies)
  - You get TWO double-sided handwritten 8.5×11" cheat sheets
    - Recommended that you reuse or remake your midterm cheat sheet

# Peer Instruction Question

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

- <http://PollEv.com/justinh>

(A) Best-fit:

`malloc(50), malloc(50)`

(B) First-fit:

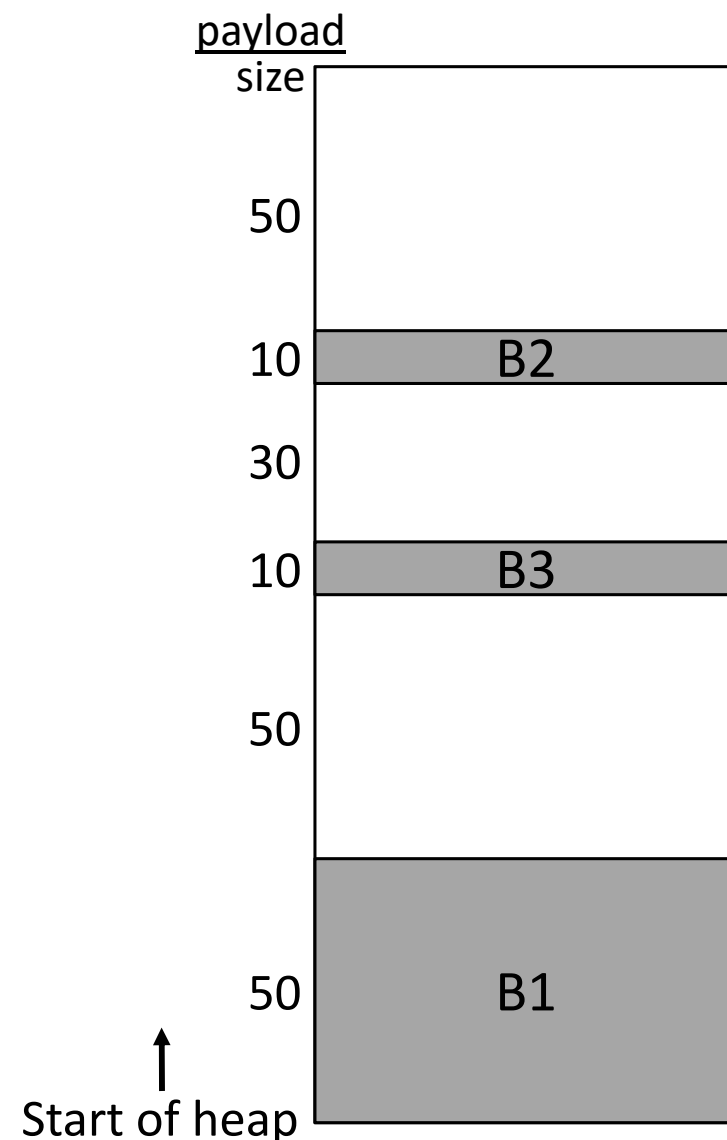
`malloc(50), malloc(30)`

(C) Next-fit:

`malloc(30), malloc(50)`

(D) Next-fit:

`malloc(50), malloc(30)`



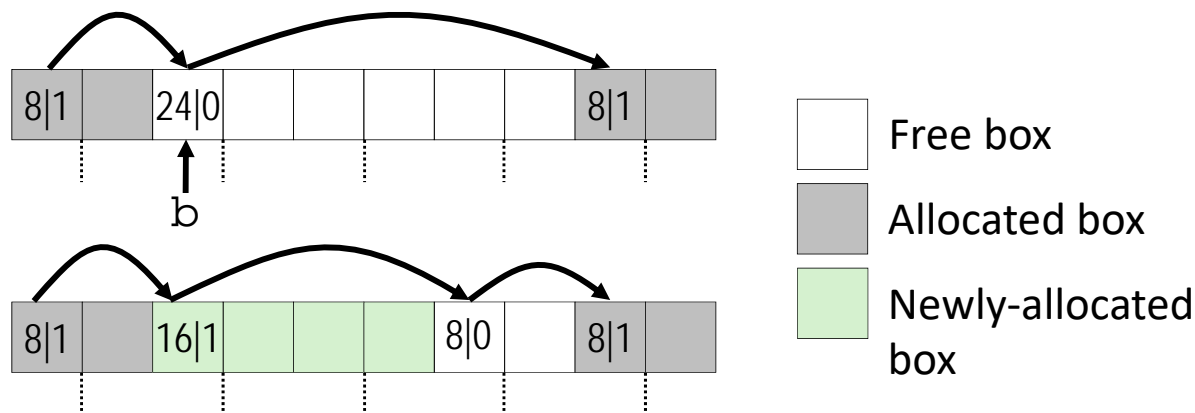
# Implicit List: Allocating in a Free Block

- ❖ 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

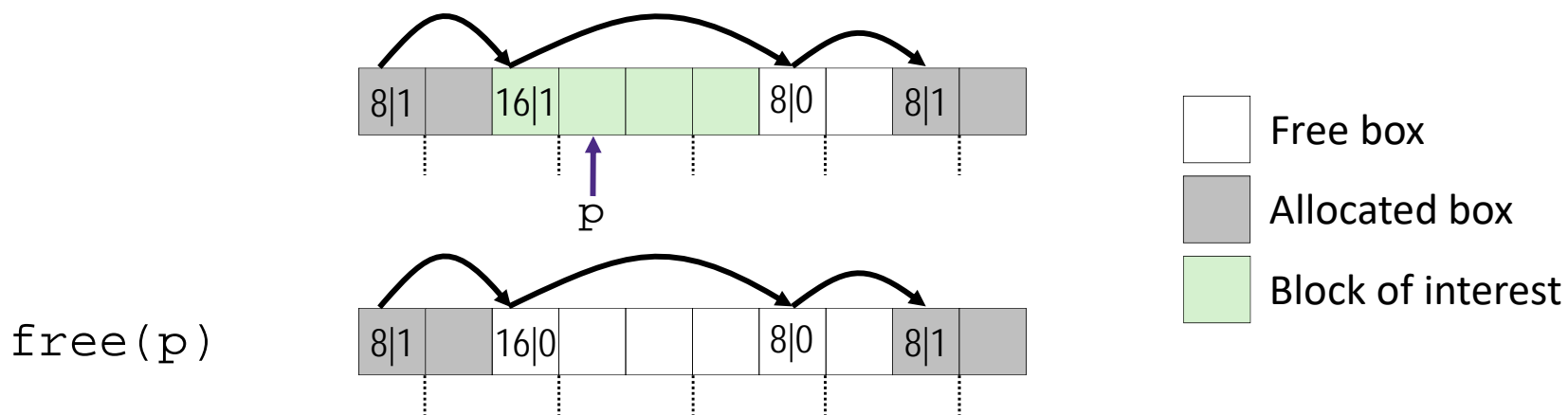
```
void split(ptr b, int bytes) { // bytes = desired block size
    int newsize = ((bytes+7) >> 3) << 3; // round up to multiple of 8
    int oldsize = *b; // why not mask out low bit?
    *b = newsize; // initially unallocated
    if (newsize < oldsize)
        *(b+newsize) = oldsize - newsize; // set length in remaining
} // part of block (UNSCALED +)
```

```
malloc(12):
ptr b = find(12+4)
split(b, 12+4)
allocate(b)
```



# Implicit List: Freeing a Block

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

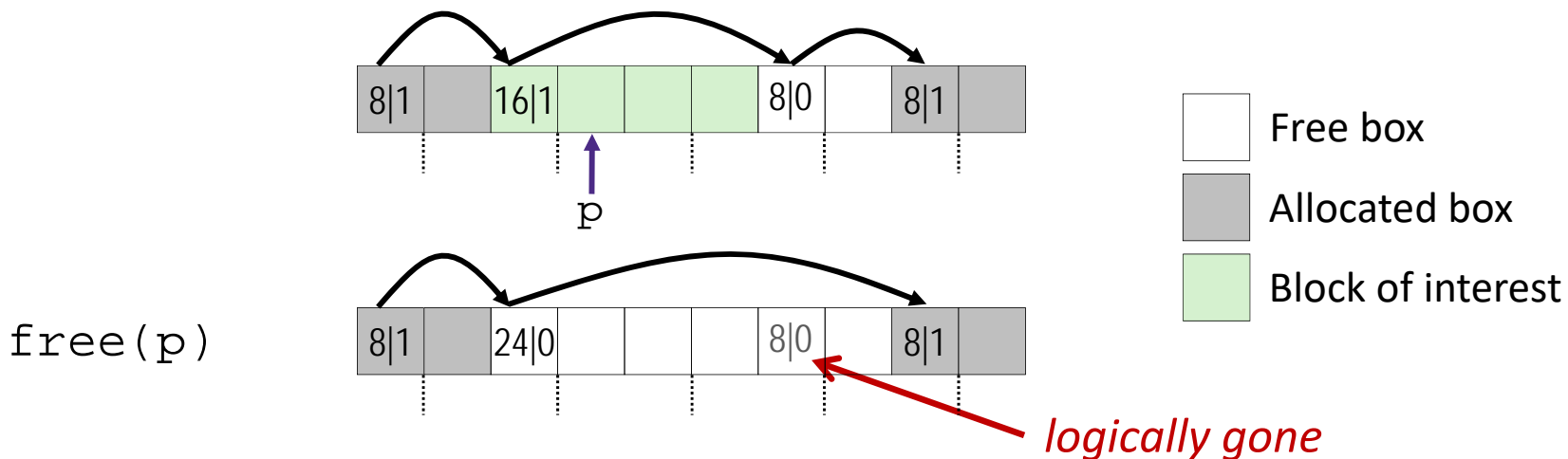


`malloc(20)`

**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



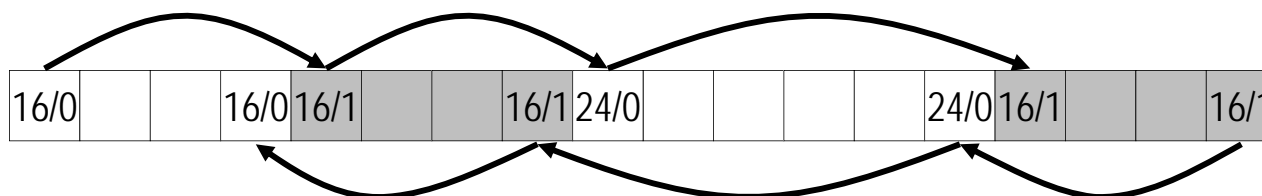
```

void free(ptr p) { // p points to data
    ptr b = p - BOX; // b points to block
    *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
}
    
```

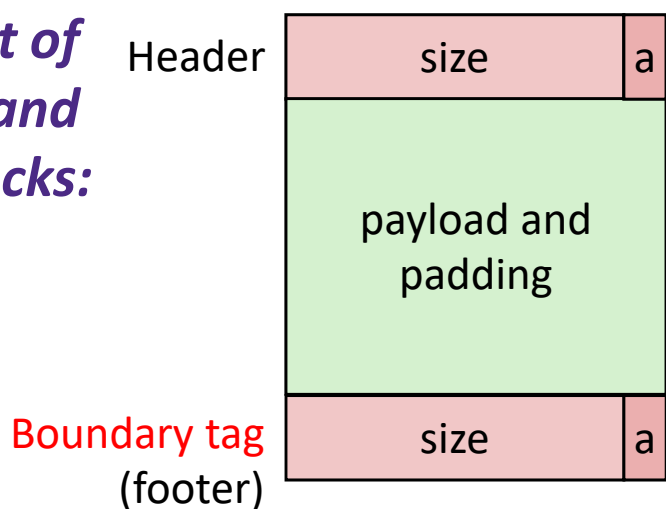
- ❖ How do we coalesce with the *previous* block?

# 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:*

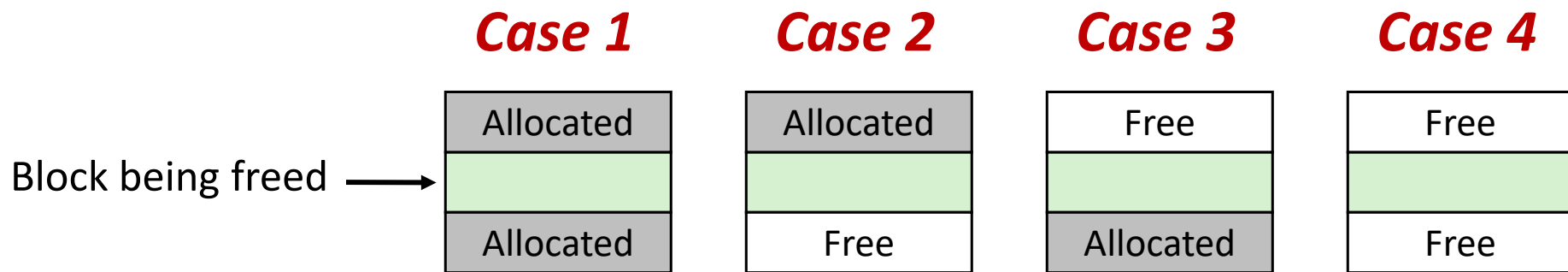


**a = 1:** allocated block  
**a = 0:** free block

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

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

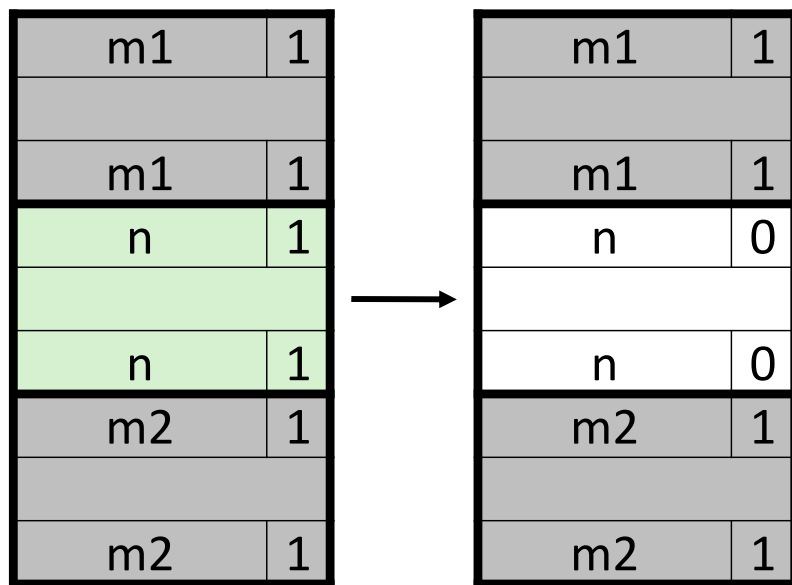
# Constant Time Coalescing



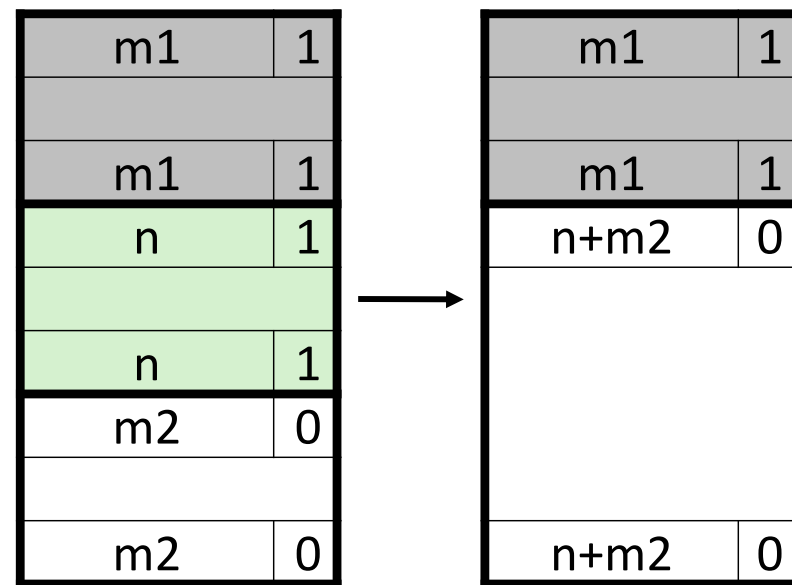


# Constant Time Coalescing

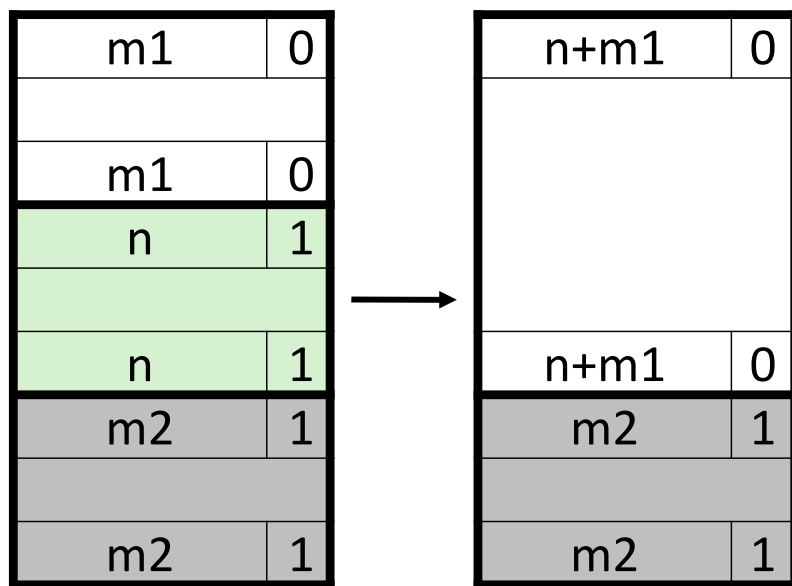
**Case 1**



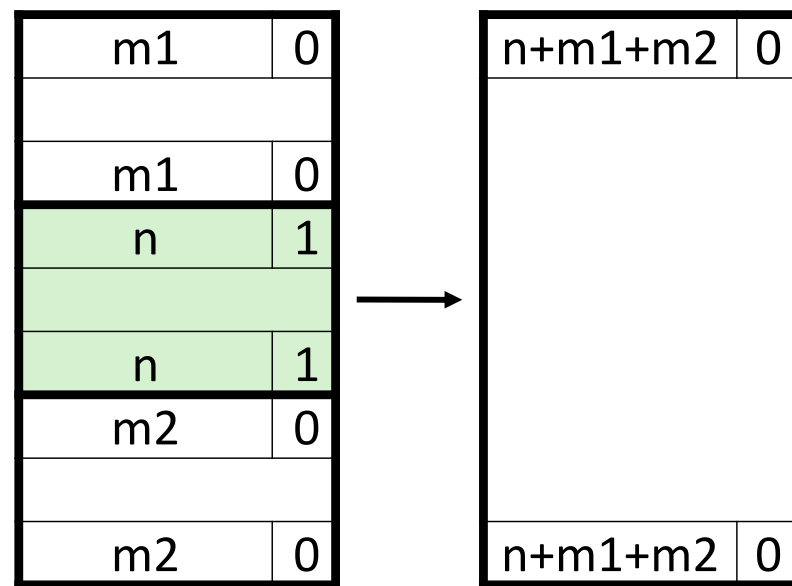
**Case 2**



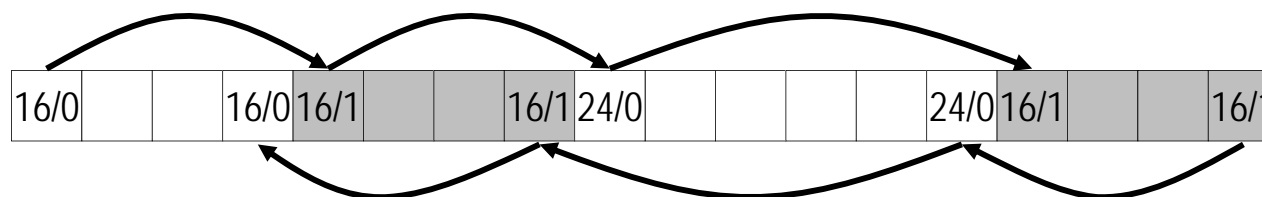
**Case 3**



**Case 4**

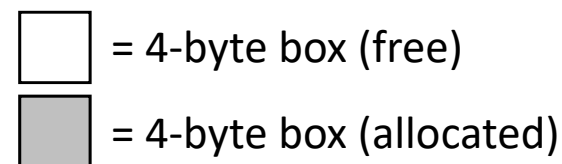


# Implicit Free List Review Questions



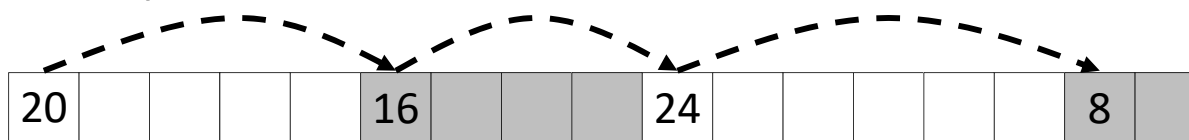
- ❖ What is the block header? What do we store and how?
- ❖ What are boundary tags and why do we need them?
- ❖ When we coalesce free blocks, how many neighboring blocks do we need to check on either side? Why is this?
- ❖ If I want to check the size of the  $n$ -th block forward from the current block, how many memory accesses do I make?

# Keeping Track of Free Blocks

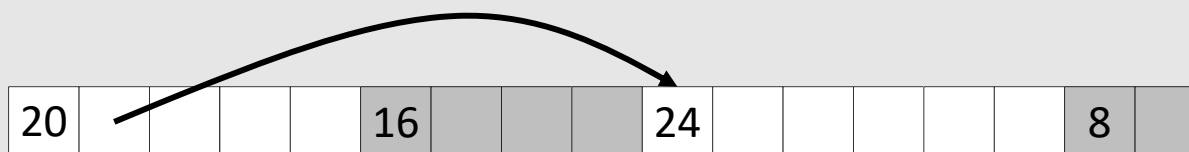


1) *Implicit free list* using length – links all blocks using math

- 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*

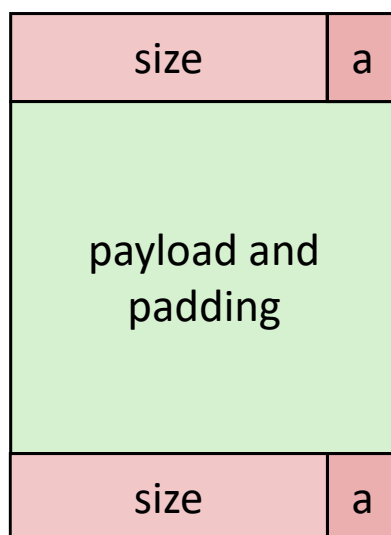
- Different free lists for different size “classes”

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

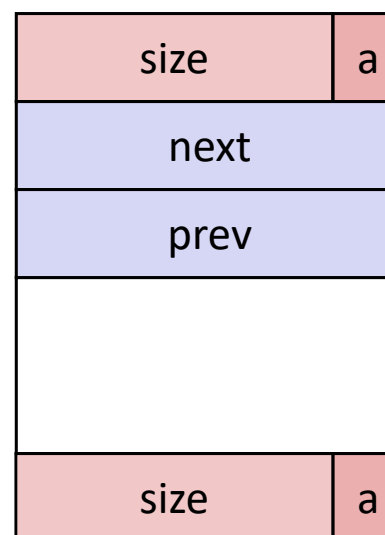
# Explicit Free Lists

Allocated block:



(same as implicit free list)

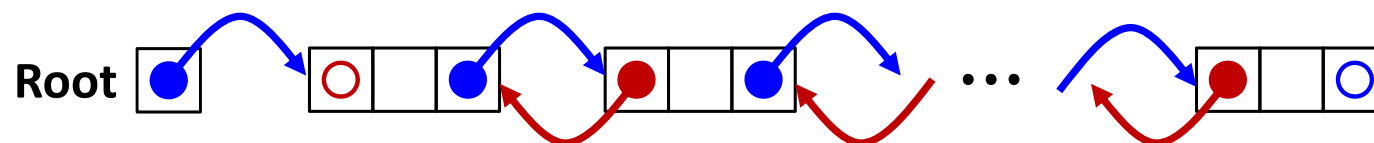
Free block:



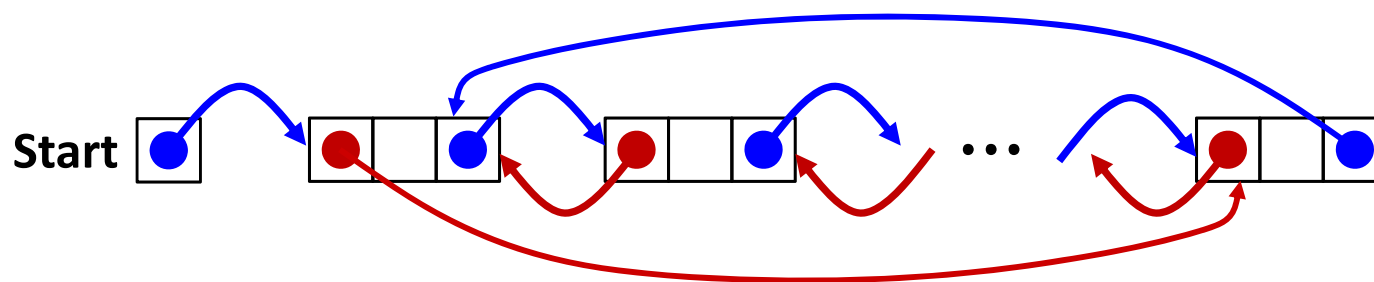
- ❖ Use list(s) of *free* blocks, rather than implicit list of *all* blocks
  - The “next” free block could be anywhere in the heap
    - So we need to store next/previous pointers, not just sizes
  - Since we only track free blocks, so we can use “payload” for pointers
  - Still need boundary tags (header/footer)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

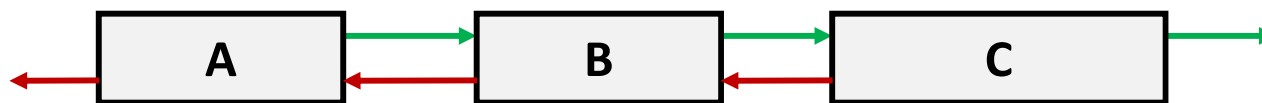


## ❖ Circular

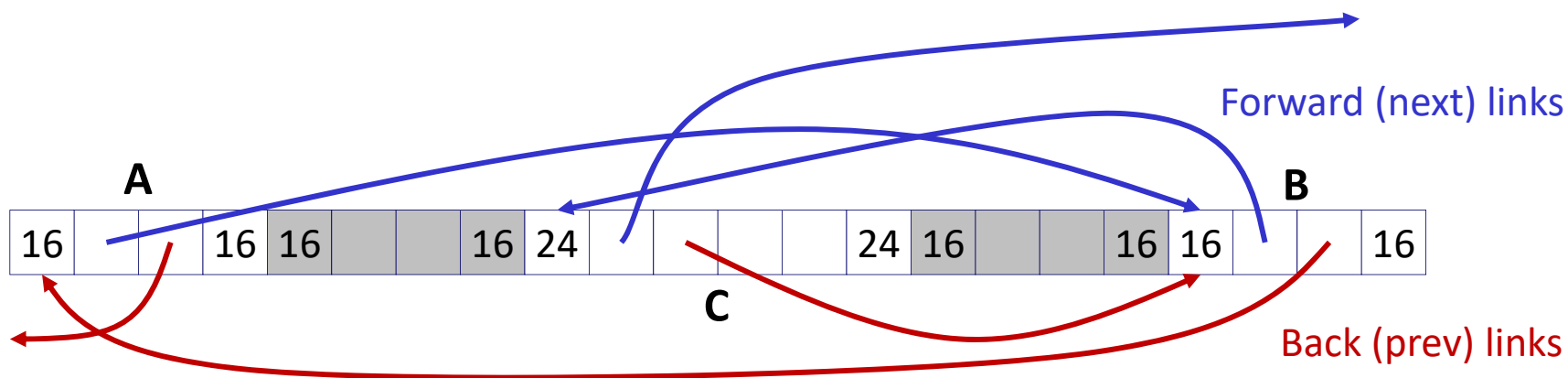
- Still have pointer to tell you which node to start with
- No NULL pointers (term condition is back at starting point)
- Good for next-fit, best-fit

# Explicit Free Lists

❖ **Logically:** doubly-linked list

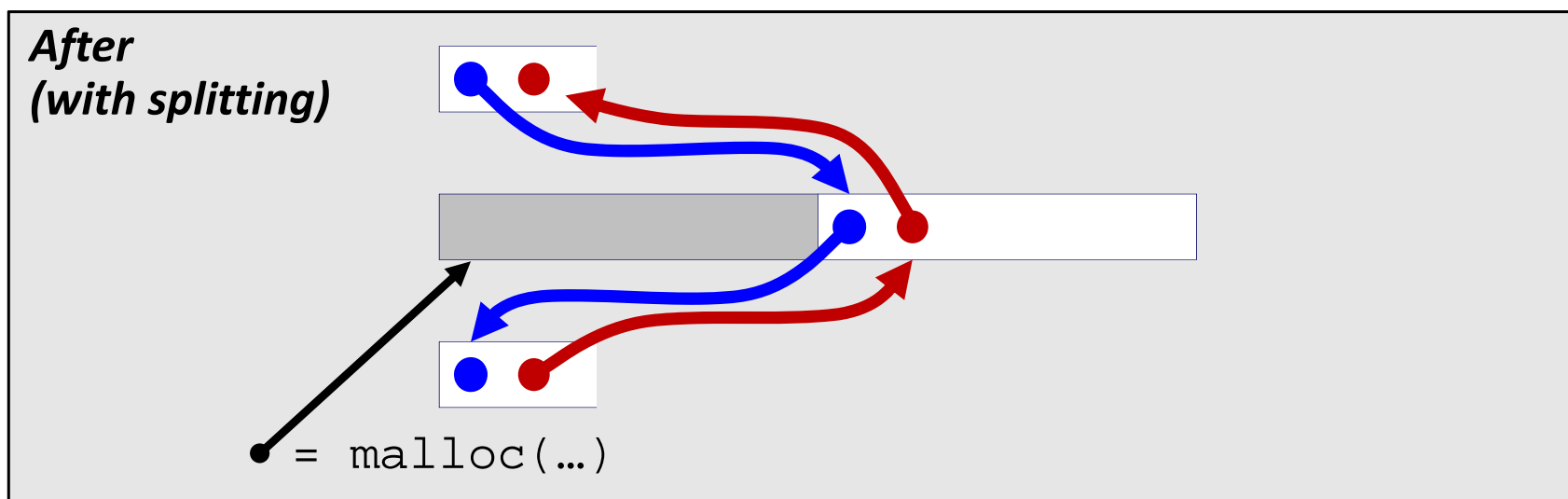
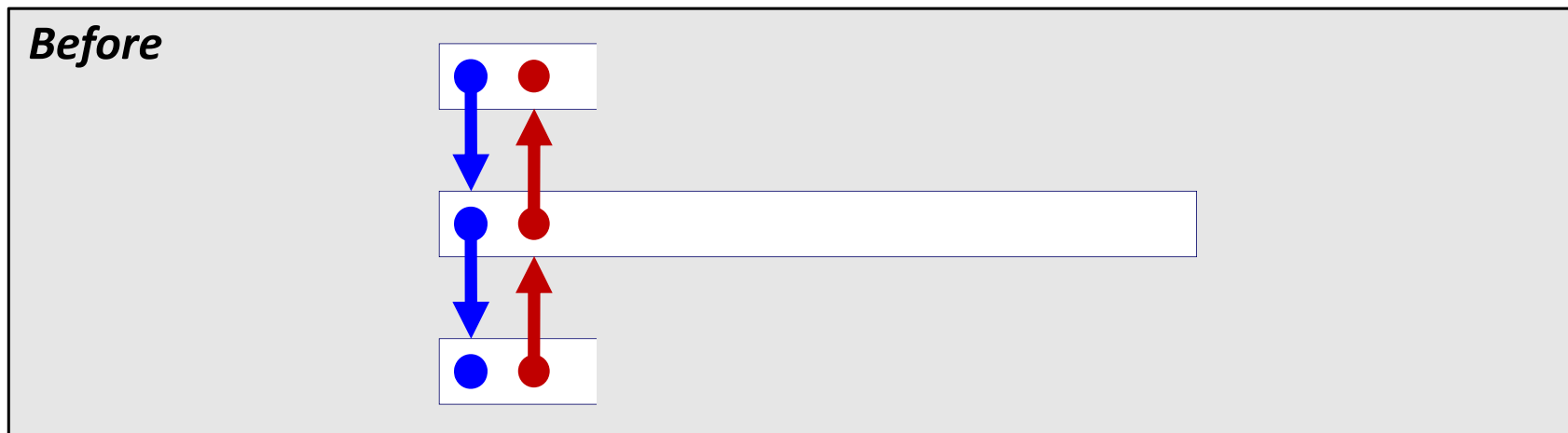


❖ **Physically:** blocks can be in any order



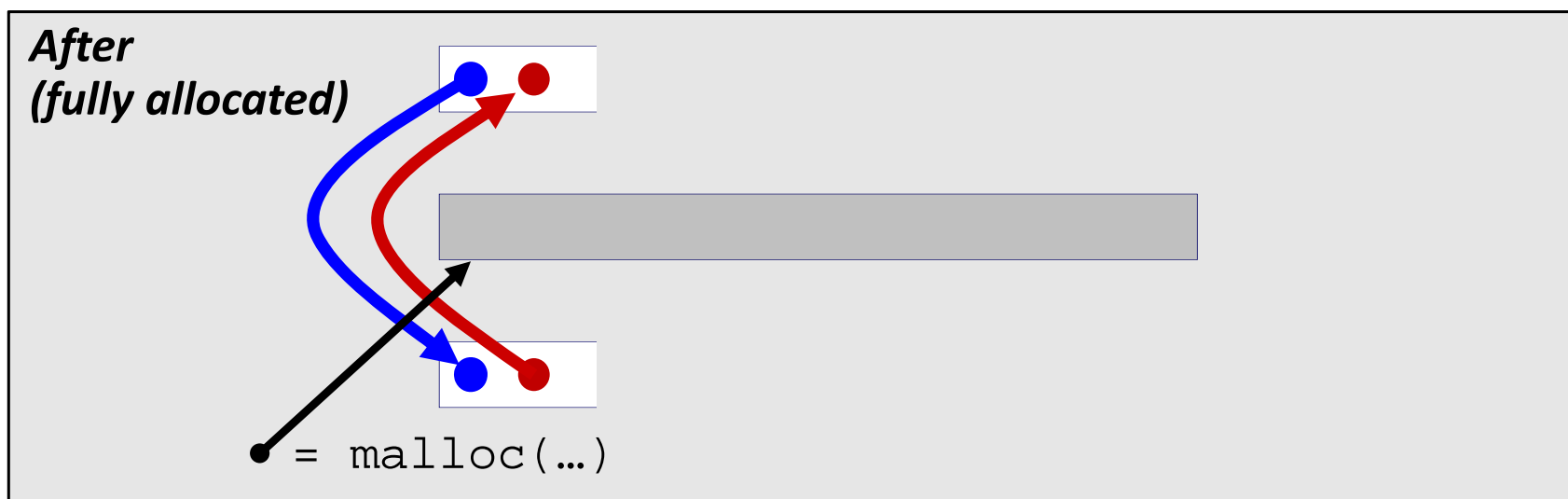
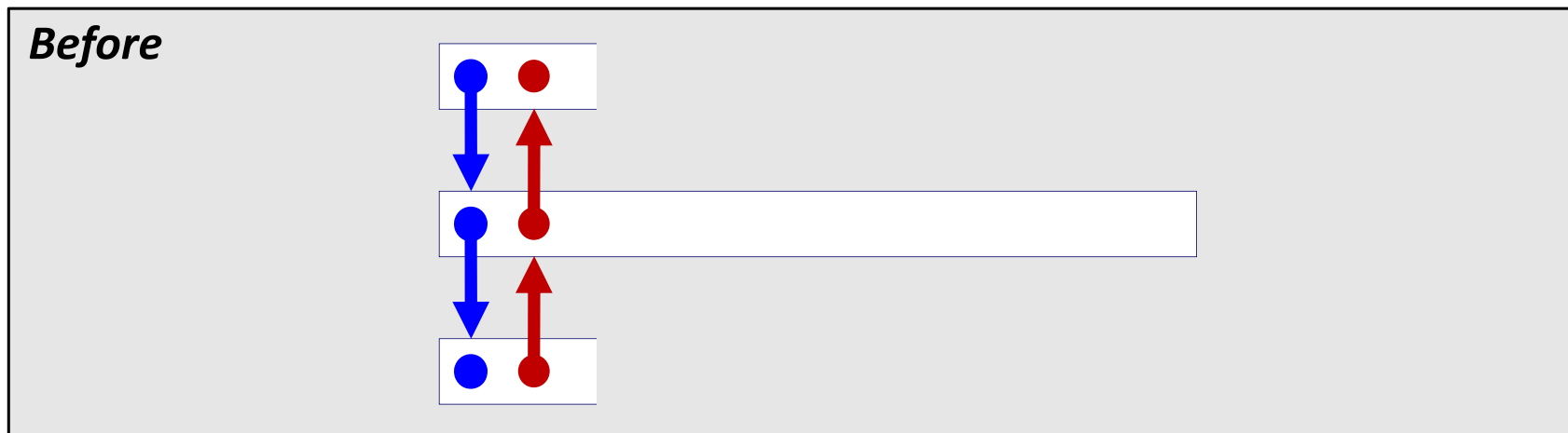
# Allocating From Explicit Free Lists

**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).

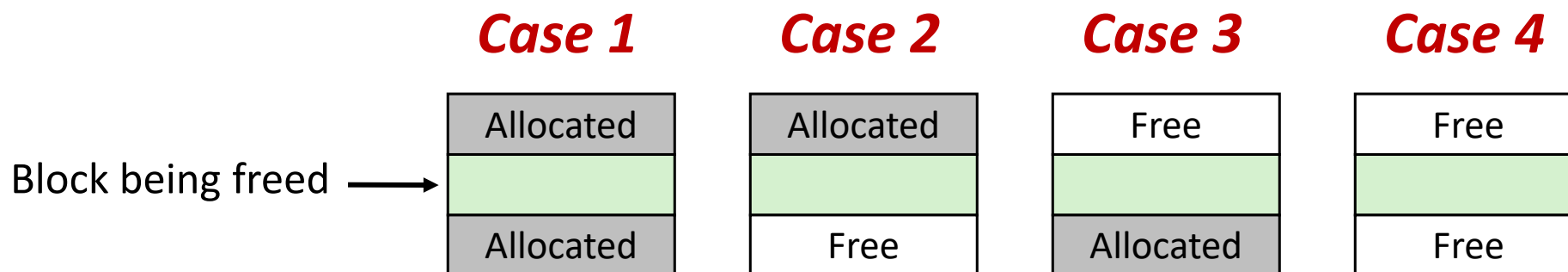




# Freeing With Explicit Free Lists

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

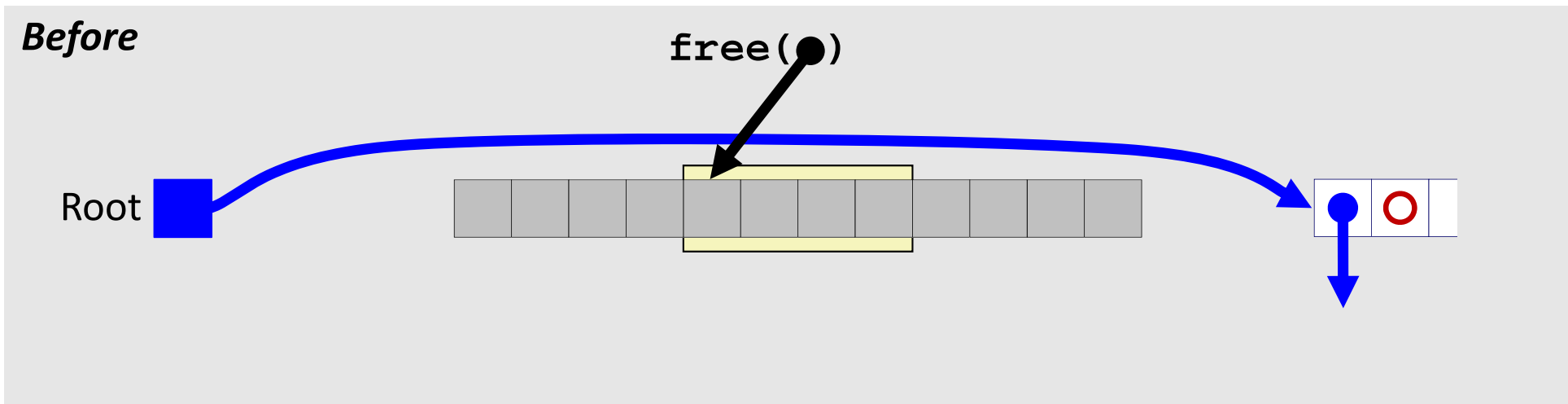
# Coalescing in Explicit Free Lists



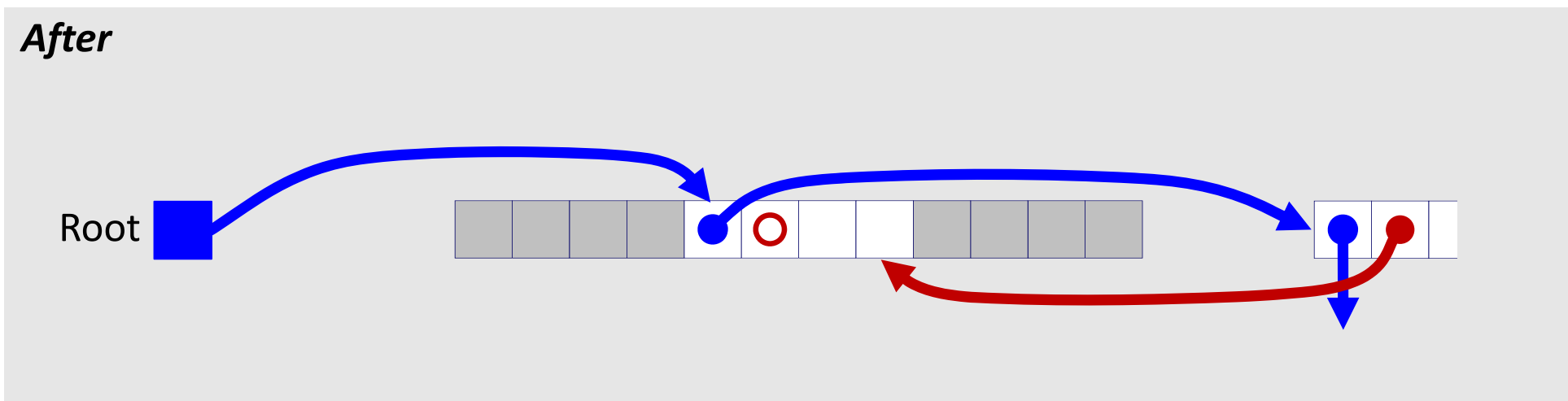
- ❖ Neighboring free blocks are *already part of the free list*
  - 1) Remove old block from free list
  - 2) Create new, larger coalesced block
  - 3) Add new block to free list (insertion policy)
- ❖ How do we tell if a neighboring block is free?

# Freeing with LIFO Policy (Case 1)

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

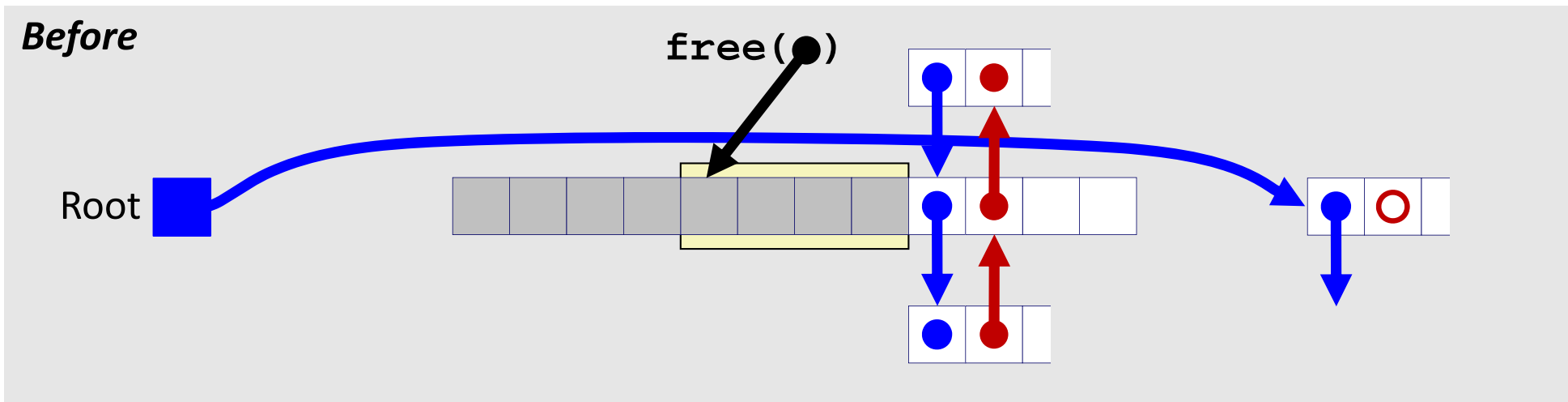


❖ Insert the freed block at the root of the list

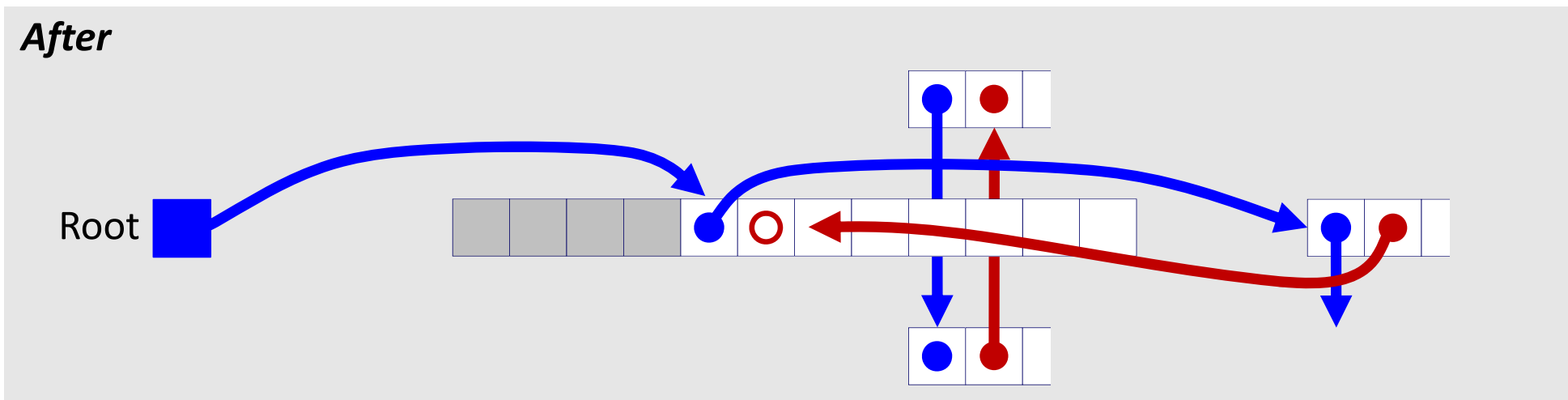


# Freeing with LIFO Policy (Case 2)

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

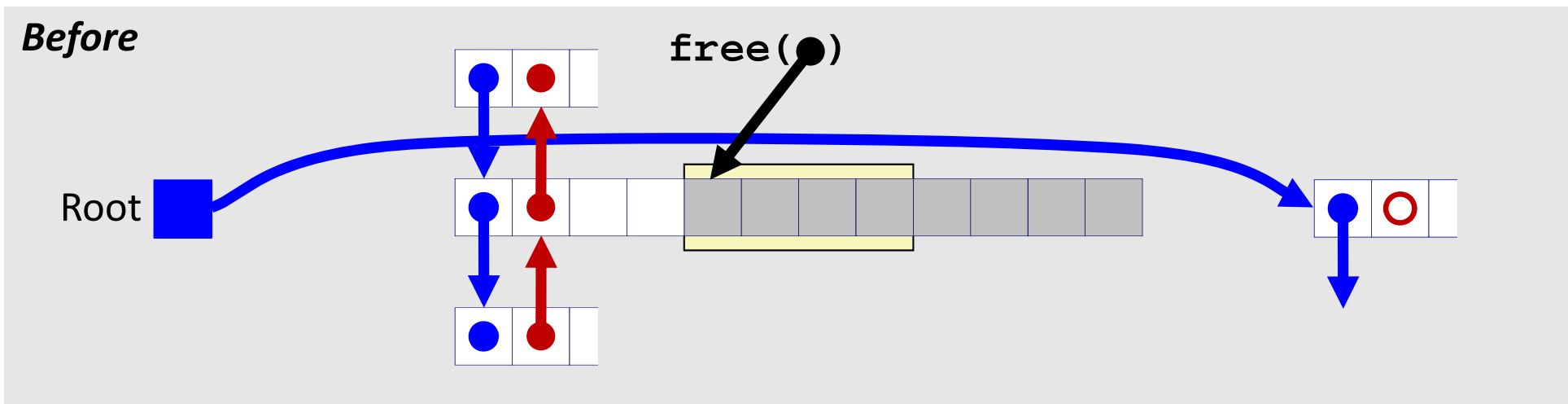


- ❖ Splice successor block out of list, coalesce both memory blocks, and insert the new block at the root of the list

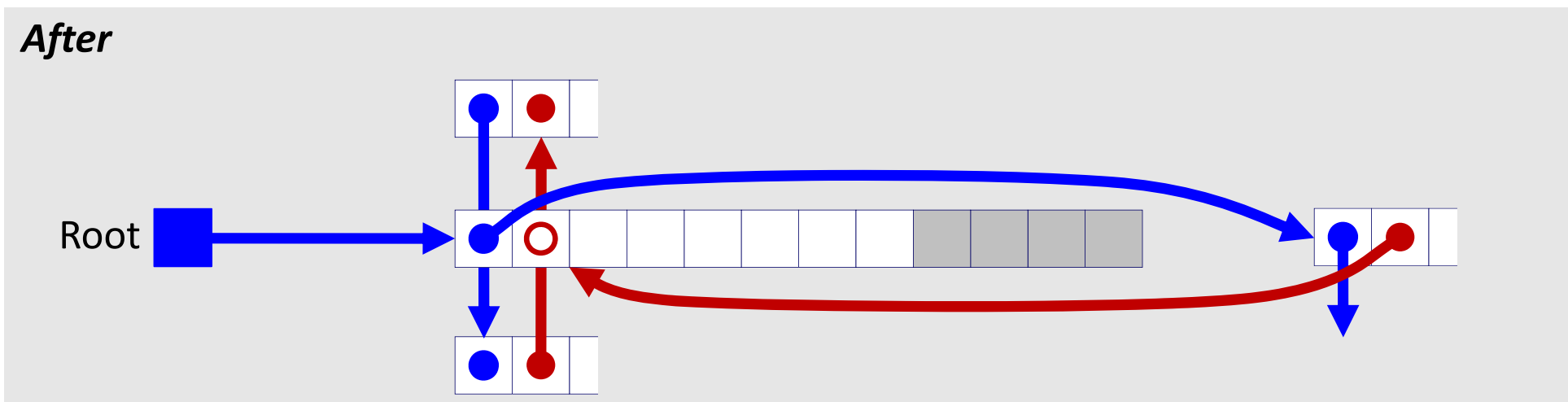


# Freeing with LIFO Policy (Case 3)

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

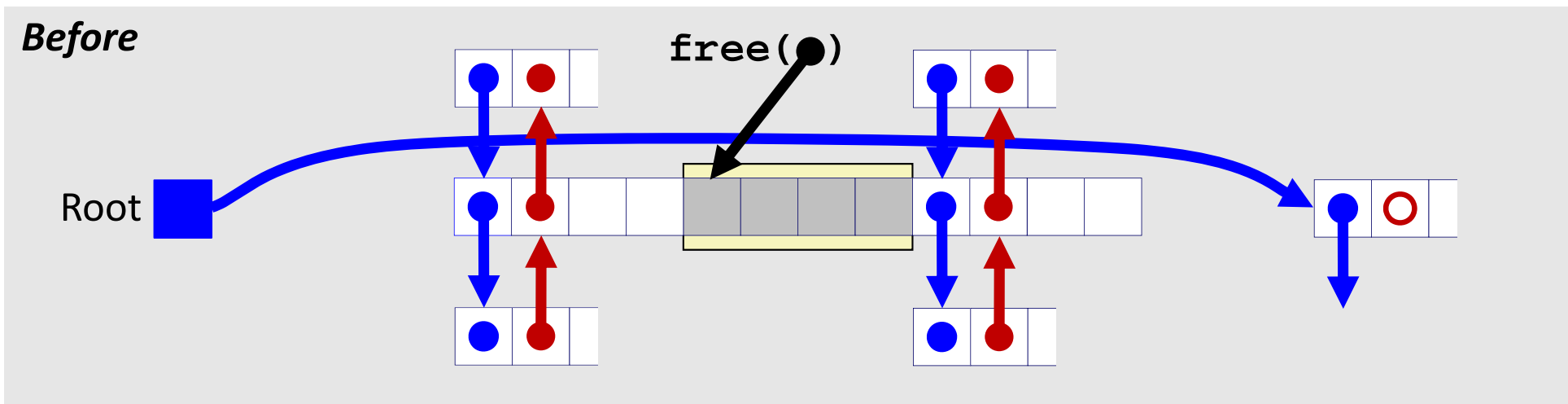


- ❖ Splice predecessor block out of list, coalesce both memory blocks, and insert the new block at the root of the list

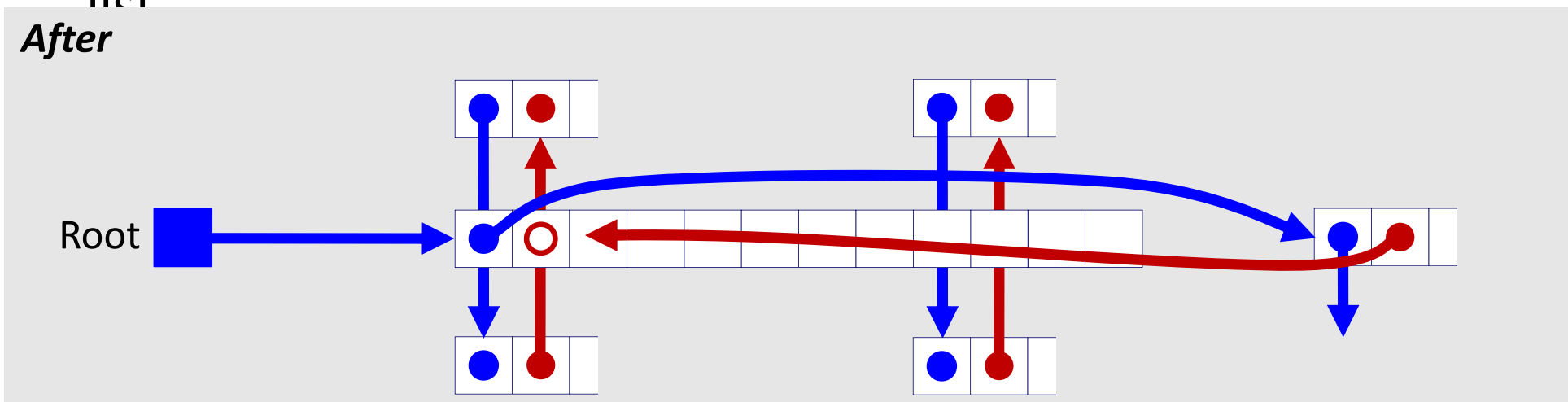


# Freeing with LIFO Policy (Case 4)

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

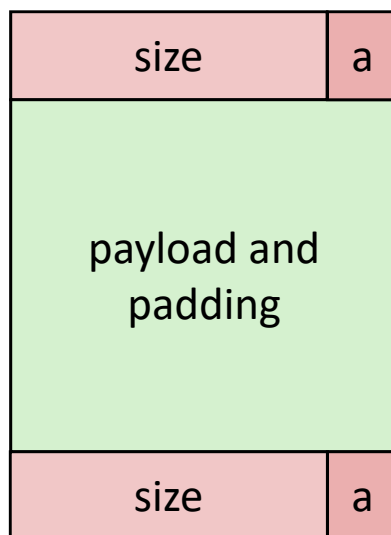


- ❖ Splice predecessor and successor blocks out of list, coalesce all 3 memory blocks, and insert the new block at the root of the list



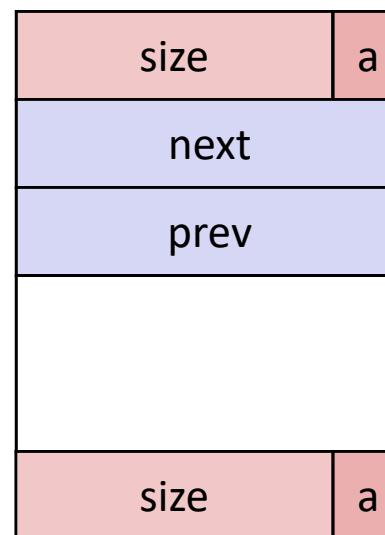
# Do we always need the boundary tags?

Allocated block:



(same as implicit free list)

Free block:



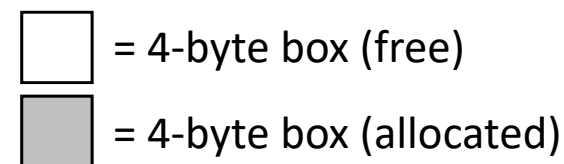
❖ Lab 5 suggests no...

# 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

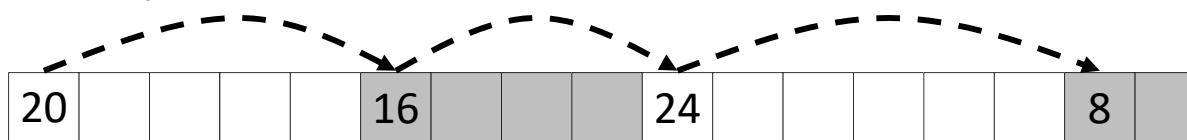


# Keeping Track of Free Blocks

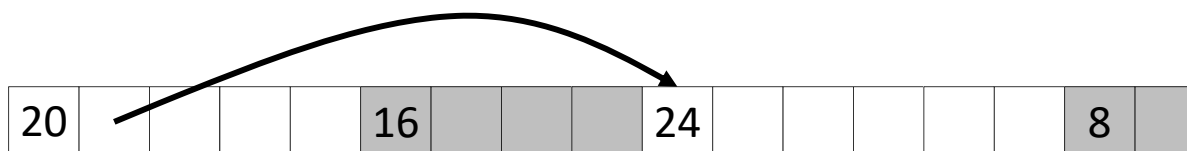


1) *Implicit free list* using length – links all blocks using math

- 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”

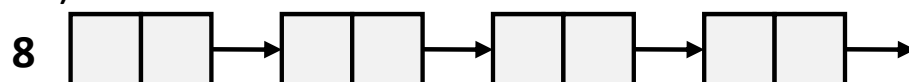
4) *Blocks sorted by size*

- 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

Size class  
(in bytes)



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

# Allocation Policy Tradeoffs

- ❖ Data structure of blocks on lists
  - Implicit (free/allocated), explicit (free), segregated (many free lists) – others possible!
- ❖ Placement policy: first-fit, next-fit, best-fit
  - Throughput vs. amount of fragmentation
- ❖ When do we split free blocks?
  - How much internal fragmentation are we willing to tolerate?
- ❖ When do we coalesce free blocks?
  - **Immediate coalescing:** Every time `free` is called
  - **Deferred coalescing:** Defer coalescing until needed
    - e.g. when scanning free list for `malloc` or when external fragmentation reaches some threshold

# More Info on Allocators

- ❖ D. Knuth, “*The Art of Computer Programming*”, 2<sup>nd</sup> 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](http://csapp.cs.cmu.edu))