Memory Allocation II
CSE 351 Winter 2020

Instructor:

Ruth Anderson

Teaching Assistants:

Jonathan Chen
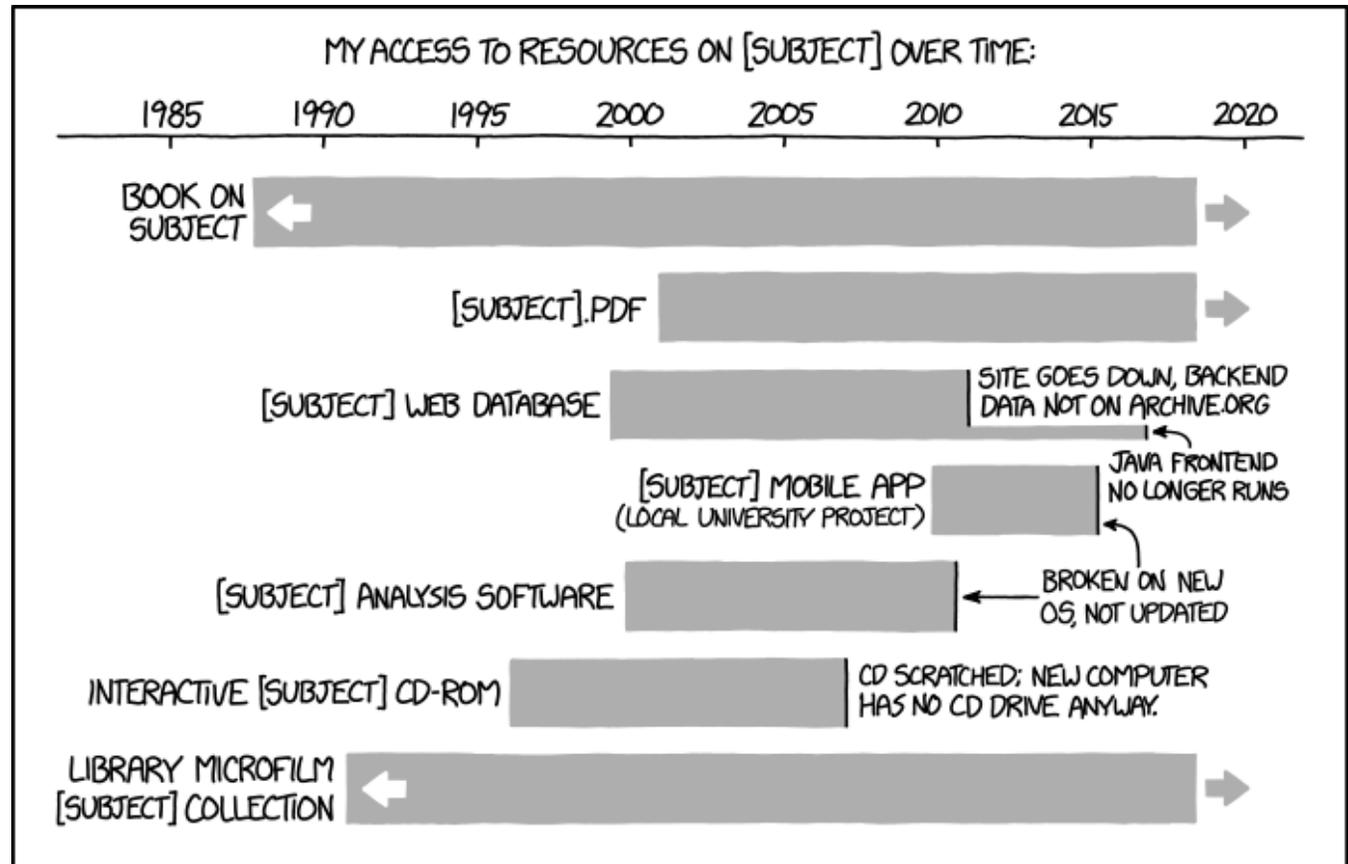Justin Johnson
Porter Jones
Josie Lee
Jeffery Tian
Callum Walker
Eddy (Tianyi) Zhou

http://xkcd.com/1909/

Administrivia

❖ hw21 due Monday (3/09)

❖ hw22 due Wednesday (3/11) – Do EARLY, will help with Lab 5

❖ Lab 5 (on Mem Alloc) due the last day of class (3/13)

▪ Can only use 1 late day (hard deadline Sun 3/15)

▪ Light style grading

❖ **Final Exam:**
Wed, March 18, 2:30-4:20pm in CSE2 G20

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   33   64   33   ← actual header data

Start of heap

16 bytes = 2 word alignment

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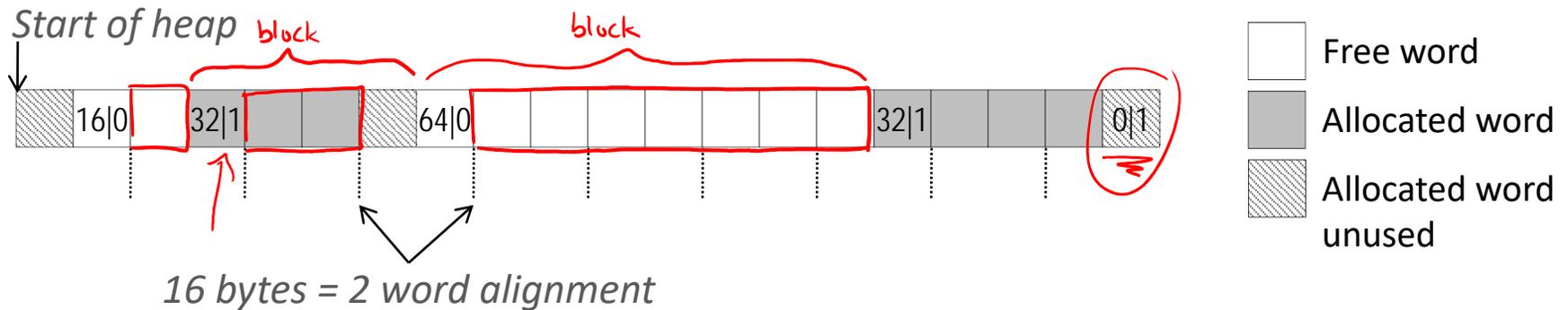
- ❖ 16-byte alignment for *payload*
  - ▪ May require initial padding (internal fragmentation)
  - ▪ Note size: padding is considered part of *previous* block
- ❖ Special one-word marker (0|1) marks end of list
  - ▪ Zero size is distinguishable from all other blocks

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:
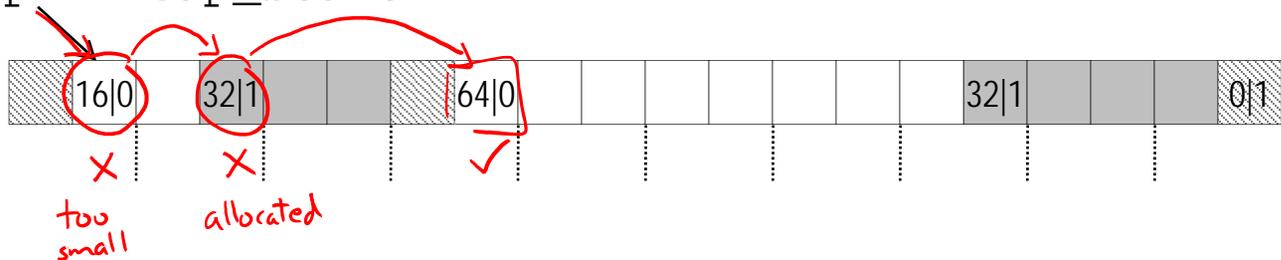
```
p = heap_start;
while ((p < end) &&        // not past end
       ((*p & 1) ||         // already allocated
        (*p <= len))) {    // too small    equivalent to pointer arithmetic with
  p = p + (*p & -2);        // go to next block (UNSCALED +)    char *
}                           // p points to selected block or end
```
                                                        after loop exits

$O(n)$

  ▪ Can take time linear in total number of blocks

  ▪ In practice can cause "splinters" at beginning of list

```
p = heap_start
```


| 16|0 | 32|1 | | | 64|0 | | | | | 32|1 | | | 0|1 |

✗ too small   ✗ allocated   ✓

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

❖ 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)`

not external fragmentation

A B C D

payload
size

space between blocks

	50
	B2 (10)
	30
	B3 (10)
	50
	B1 (50)

Start of heap
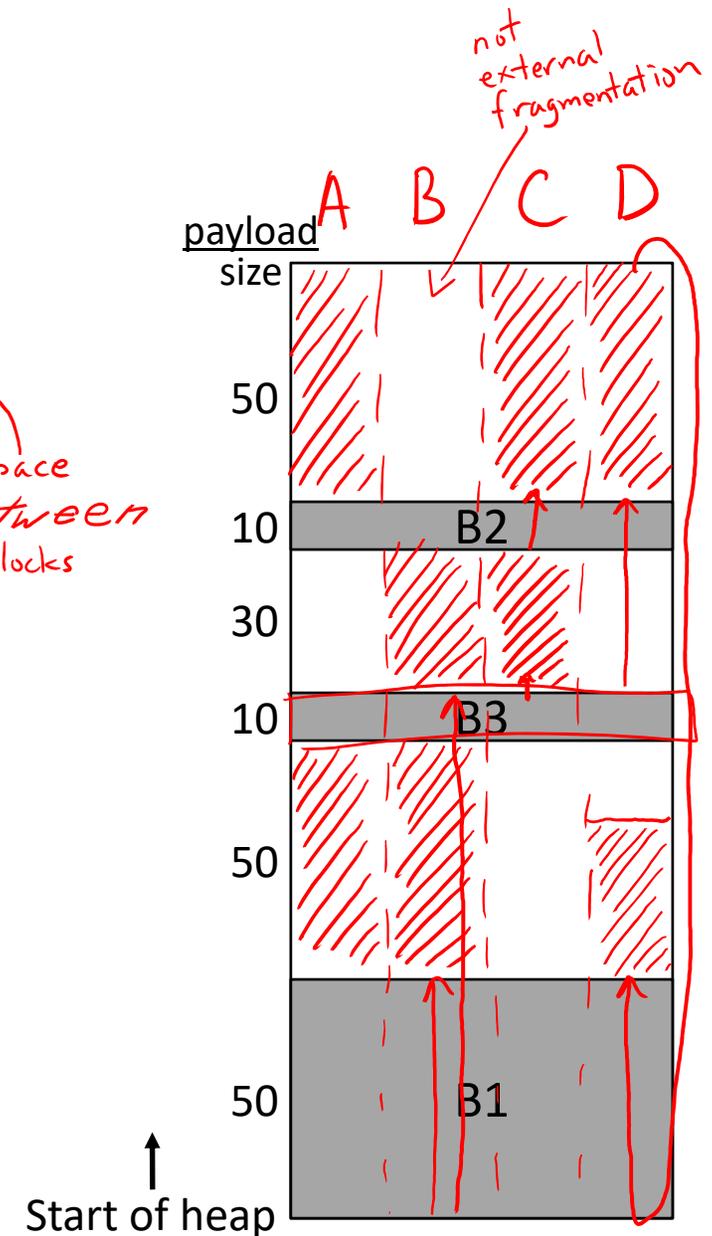
Implicit List: Allocating in a Free Block

new allocated   new free
split

❖ Allocating in a free block: *splitting*
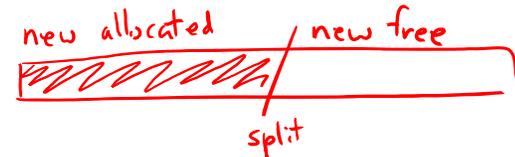
  ▪ Since allocated space might be smaller than free space, we might want to split the block

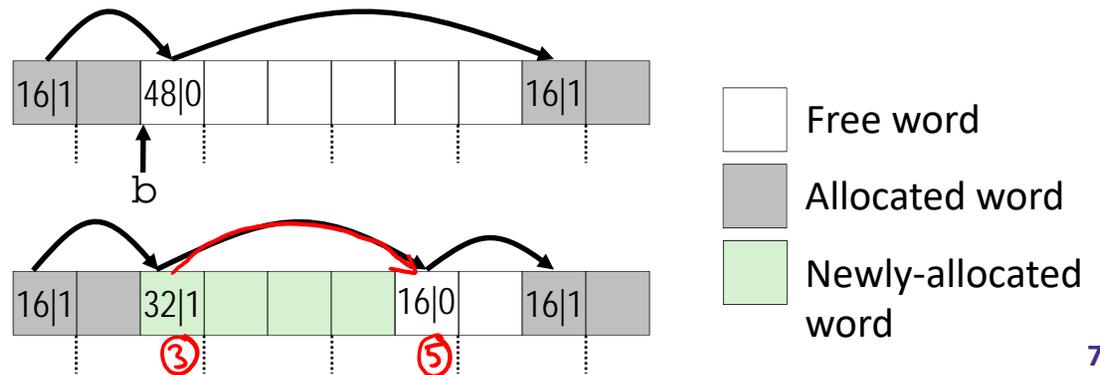Assume `ptr` points to a *free* block and has unscaled pointer arithmetic

```
void split(ptr b, int bytes) {          // bytes = desired block size
①  int newsize = ((bytes+15) >> 4) << 4;  // round up to multiple of 16
②  int oldsize = *b;                       // why not mask out low bit?
③  *b = newsize;                           // initially unallocated
④  if (newsize < oldsize)
⑤    *(b+newsize) = oldsize - newsize;    // set length in remaining
}                                          // part of block (UNSCALED +)
```
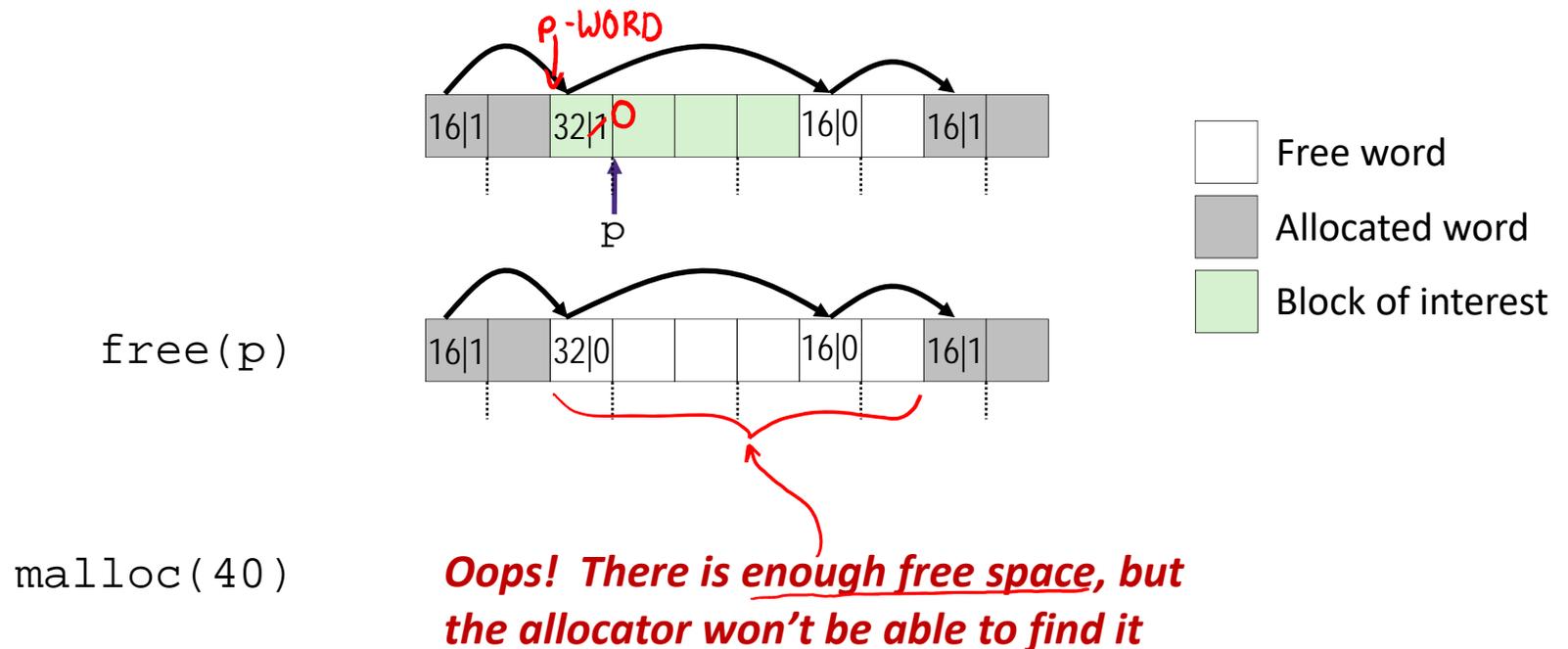
(annotations: 32, 24 above bytes; 48 above oldsize; 48 above oldsize-newsize; 32 above newsize)

```
malloc(24):
   ptr b = find(24+8)
   split(b, 24+8)
   allocate(b)
```
sets a=1

header

| 16|1 | 48|0 | | | | 16|1 | |

b

| 16|1 | 32|1 | | | 16|0 | 16|1 | |

③   ⑤

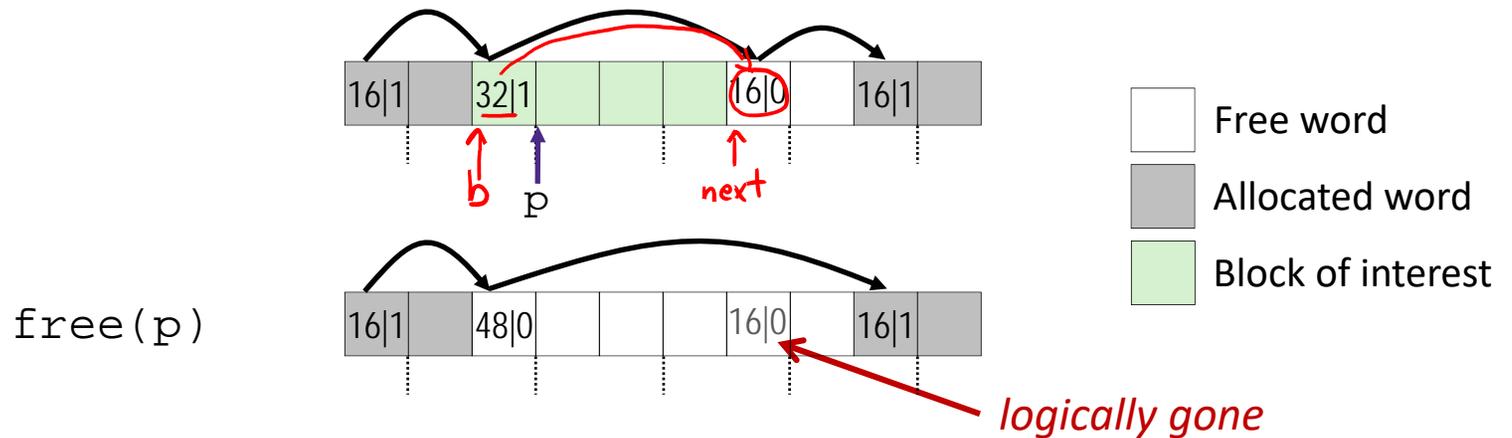| | Free word |
| | Allocated word |
| | Newly-allocated word |

Implicit List: Freeing a Block

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

free(p)

malloc(40)

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

Implicit List: Coalescing with Next

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

Free word	
Allocated word	
Block of interest	

free(p)

logically gone

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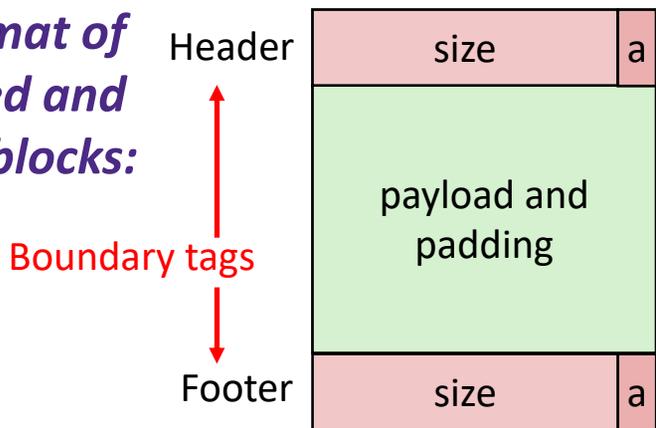
❖ How do we coalesce with the *previous* block? we can't currently

Implicit List: Bidirectional Coalescing

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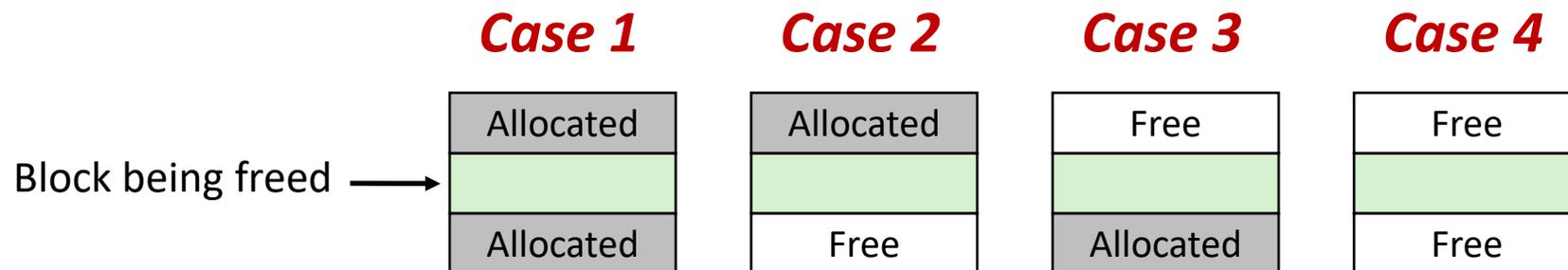- Important and general technique!
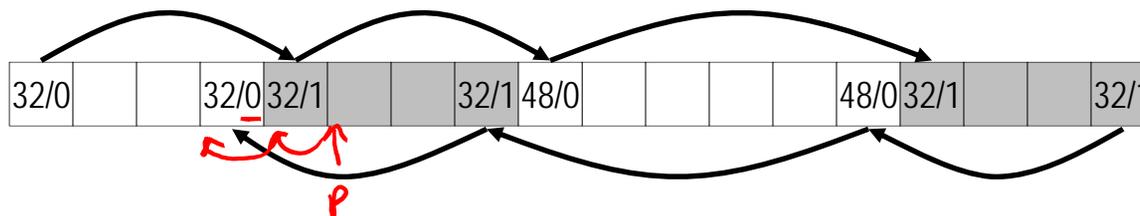
Format of allocated and free blocks:

Header	size	a
	payload and padding	
Footer	size	a

Boundary tags

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

size: block size (in bytes)

payload: application data
(allocated blocks only)
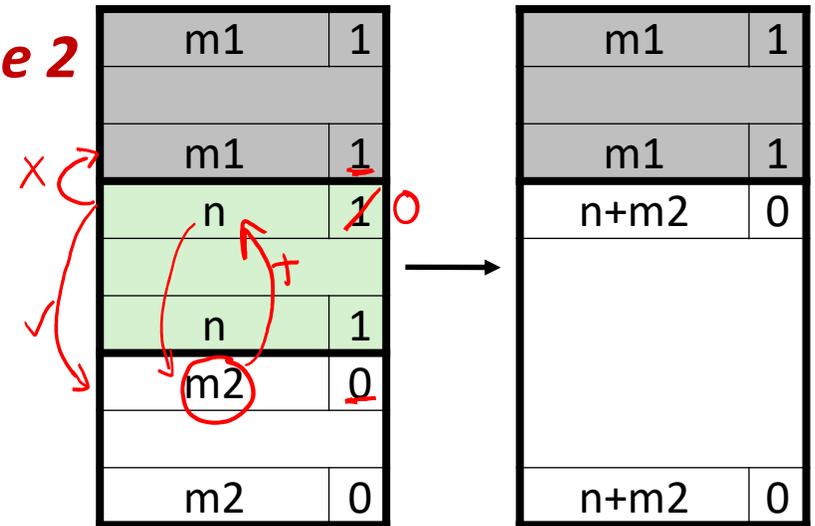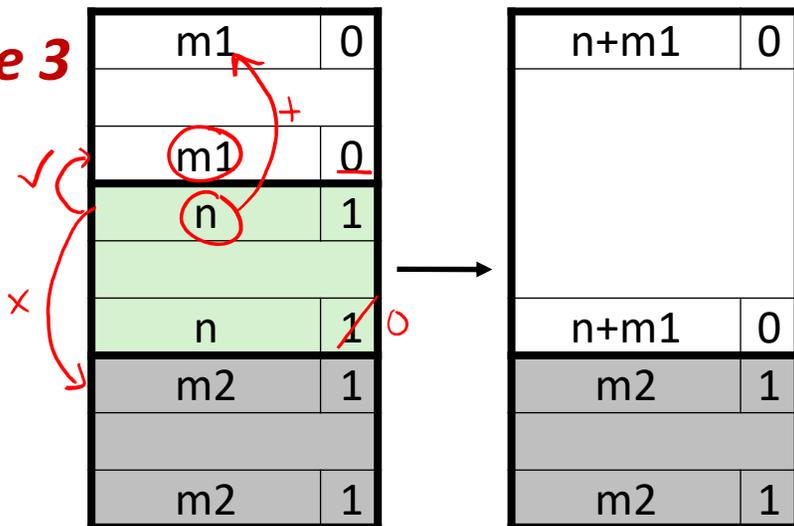
Constant Time Coalescing

	Case 1	**Case 2**	**Case 3**	**Case 4**
	Allocated	Allocated	Free	Free
Block being freed →				
	Allocated	Free	Allocated	Free

Constant Time Coalescing

Case 1

m1	1
m1	1
n	1
n	1
m2	1
m2	1

→

m1	1
m1	1
n	0
n	0
m2	1
m2	1

Case 2

m1	1
m1	1
n	1
n	1
m2	0
m2	0

→

m1	1
m1	1
n+m2	0
n+m2	0

Case 3

m1	0
m1	0
n	1
n	1
m2	1
m2	1

→

n+m1	0
n+m1	0
m2	1
m2	1

Case 4

m1	0
m1	0
n	1
n	1
m2	0
m2	0

→

n+m1+m2	0
n+m1+m2	0
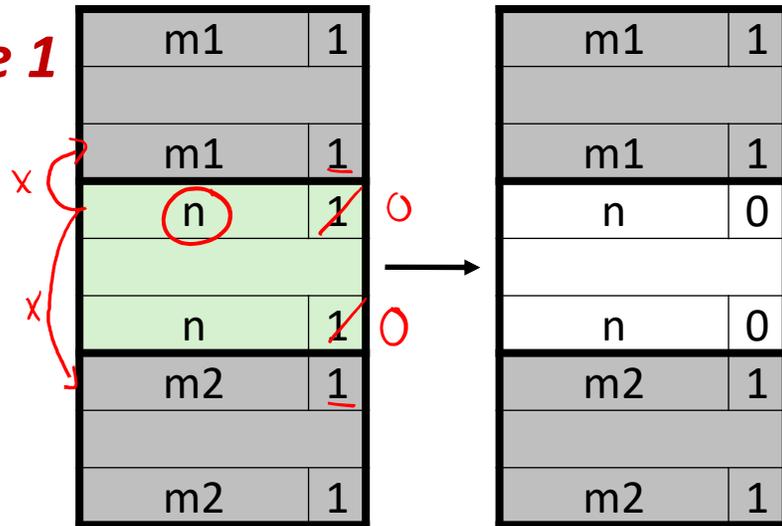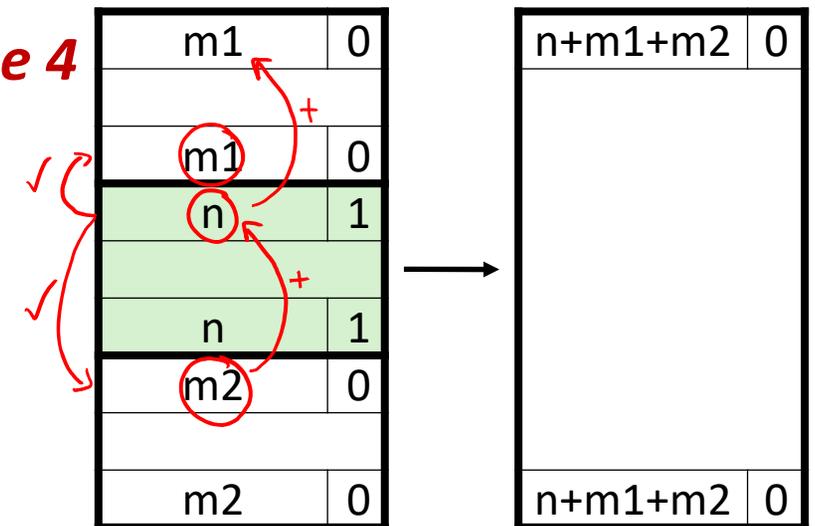
Implicit Free List Review Questions

current
block

32/0   32/0 32/1   32/1 48/0   48/0 32/1   32/1 $n=3$

① ② ③ ④
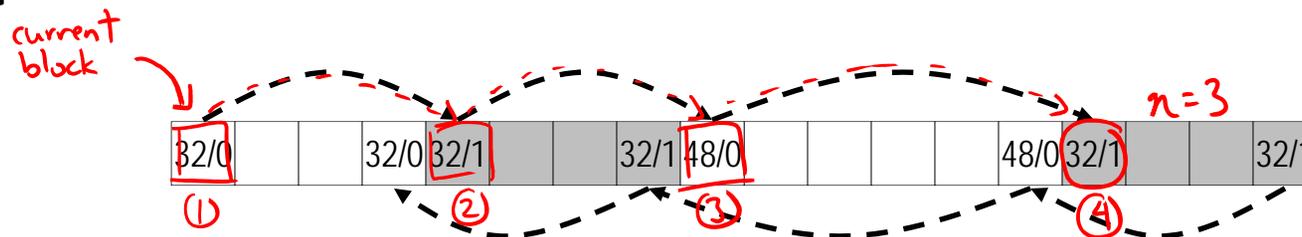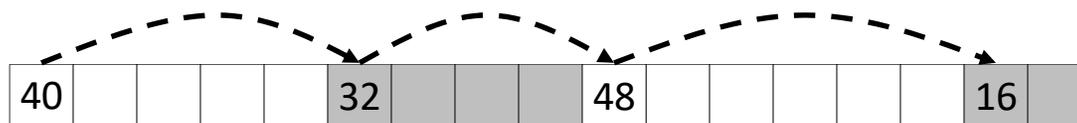
❖ **What is the block header? What do we store and how?**

stores info about block   size of block, is-allocated?
                                        ↳ lowest bit of header

❖ **What are boundary tags and why do we need them?**

header and footer (same info)   So we can traverse list in either direction
                                        (particularly for coalescing)

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

just **1** — adjacent free blocks should 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?**

$n+1$: need to read current block's header as well as header of target block to get the size
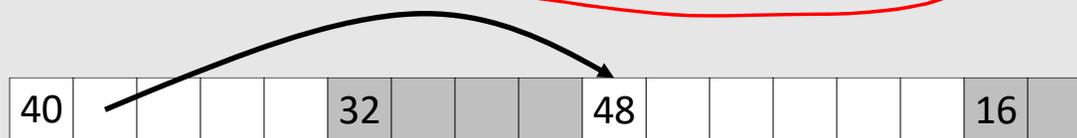
Keeping Track of Free Blocks
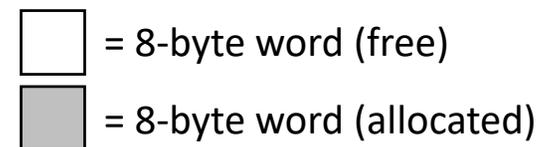
☐ = 8-byte word (free)
▨ = 8-byte word (allocated)

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

| 40 | | | | 32 | | | 48 | | | | 16 | |

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

| 40 | | | | 32 | | | 48 | | | | 16 | |

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:

Free block:


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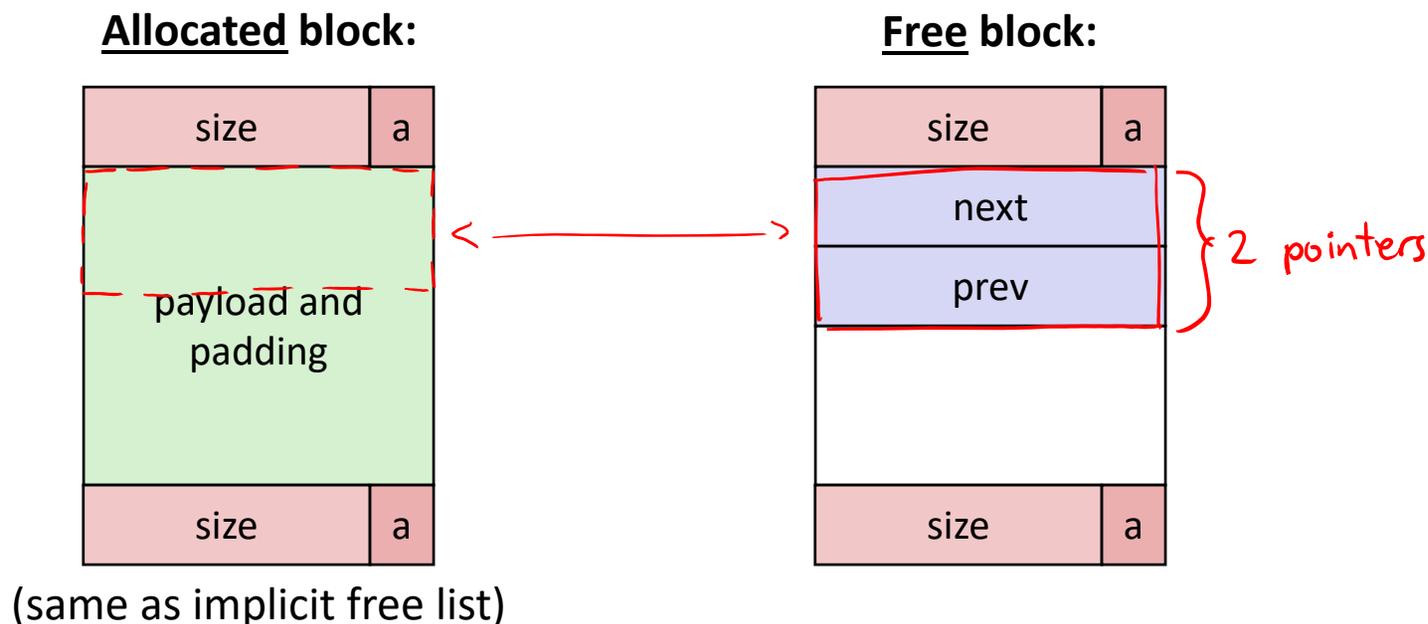
- ❖ Use list(s) of *free* blocks, rather than implicit list of *all* blocks
  - The "next" free block could be anywhere in the heap
    - So we need to store next/previous pointers, not just sizes
  - Since we only track free blocks, so we can use "payload" for pointers
  - Still need boundary tags (header/footer)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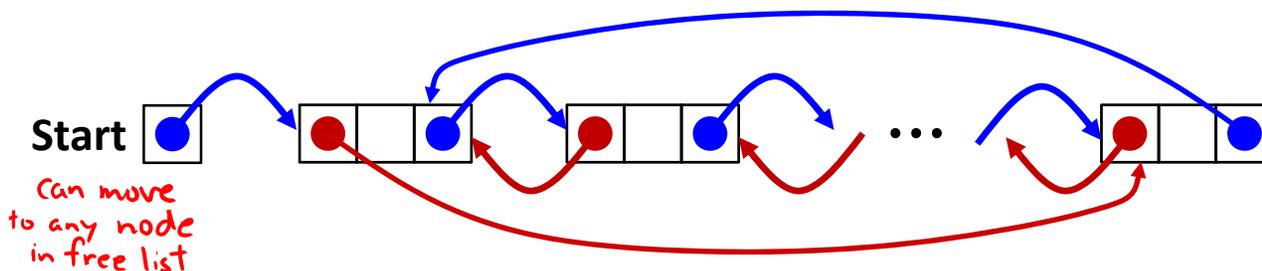
Doubly-Linked Lists

❖ **Linear**

- Needs head/root pointer
- First node prev pointer is `NULL`
- Last node next pointer is `NULL`
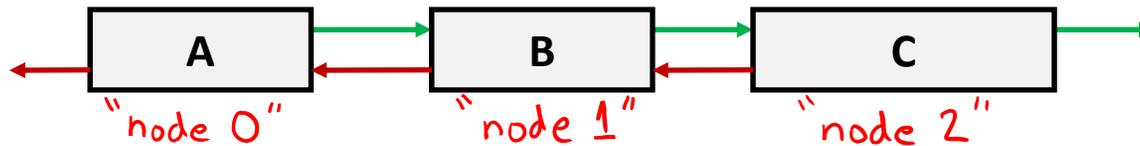- Good for <u>first-fit</u>, best-fit
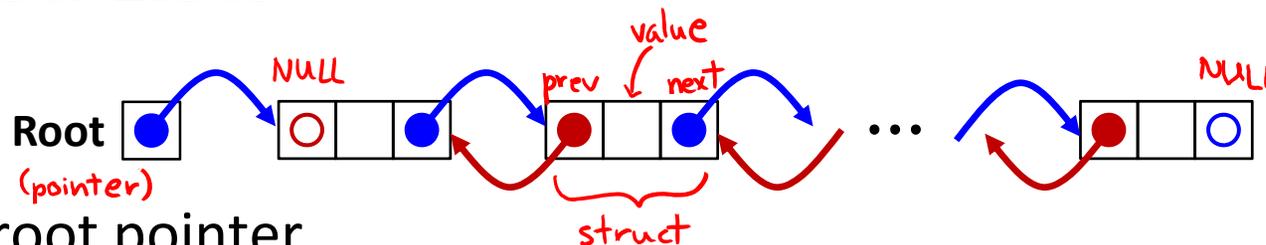
❖ **Circular**

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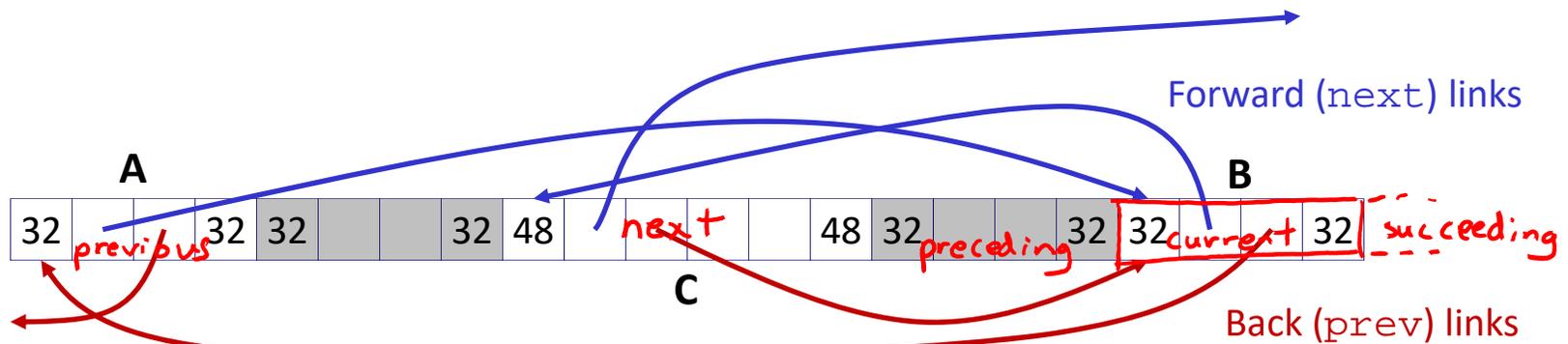- Good for <u>next-fit</u>, best-fit

Explicit Free Lists
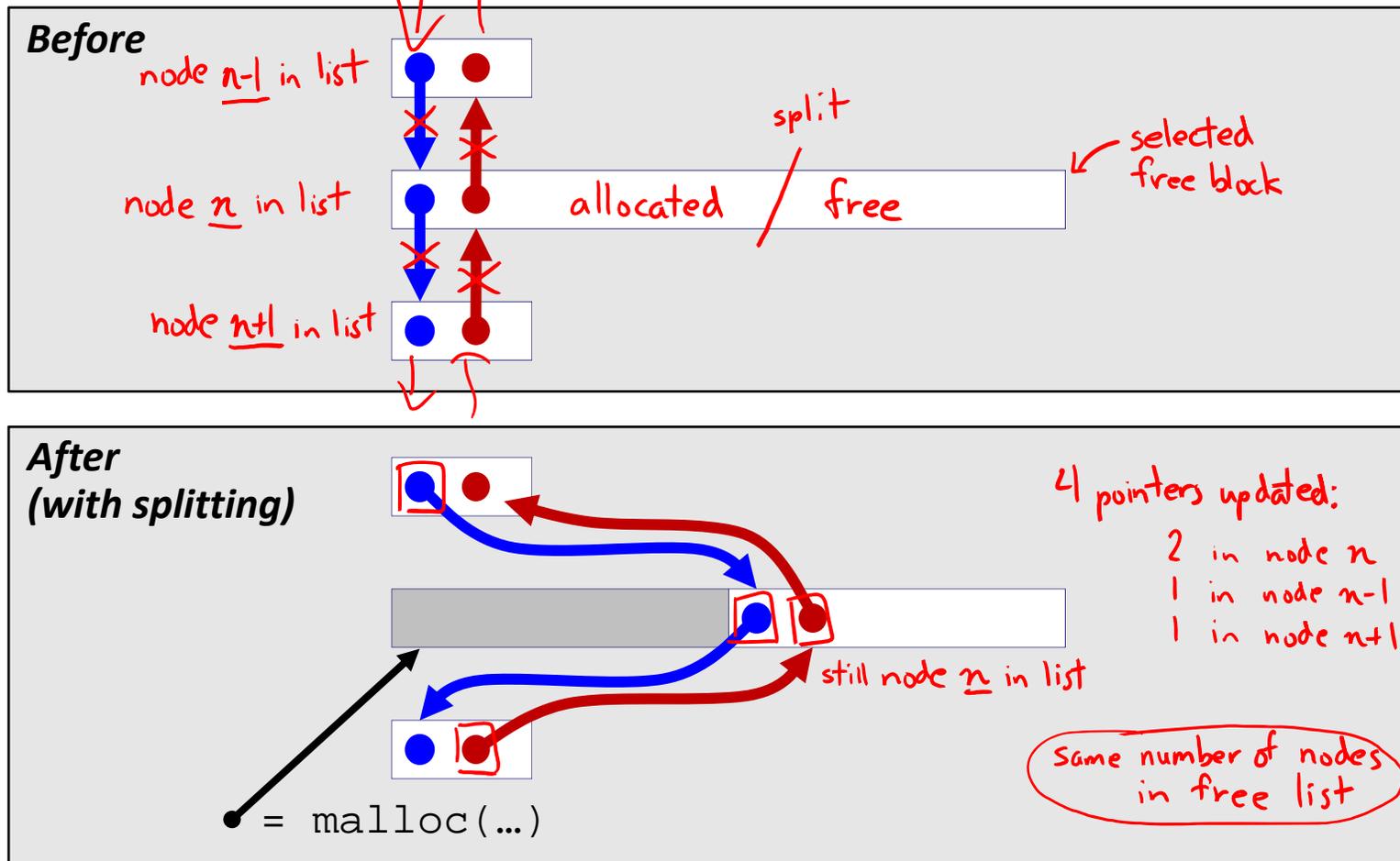
❖ **Logically:** doubly-linked list

"node 0"   "node 1"   "node 2"

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

Forward (next) links

Back (prev) links

next/prev blocks are part of free list
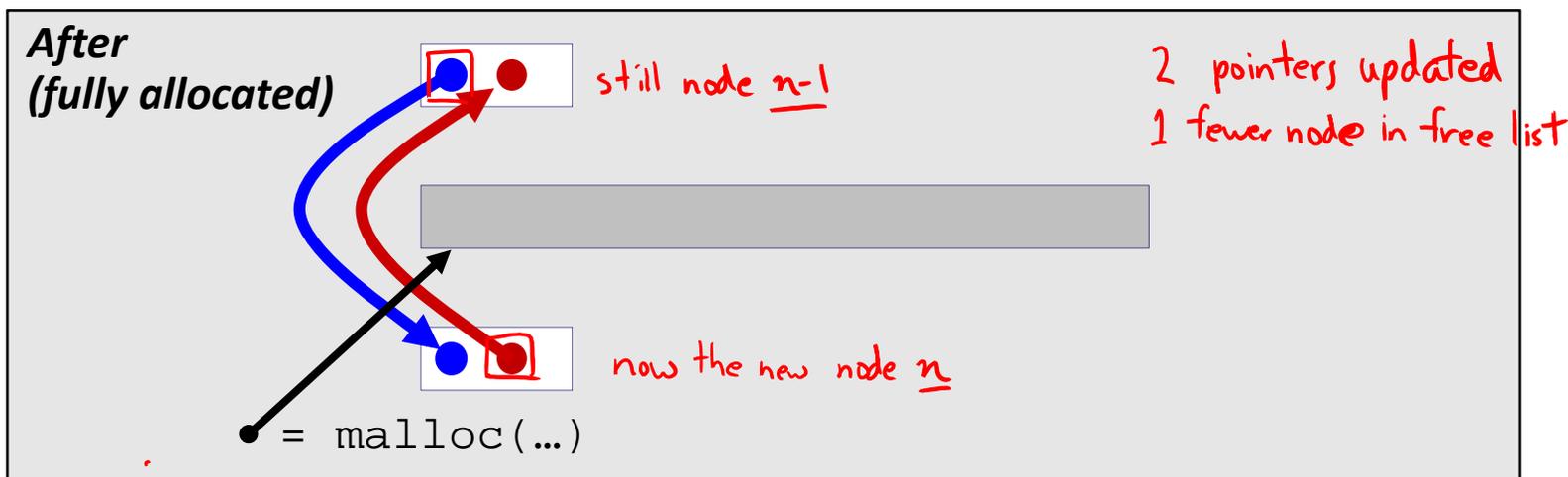preceding/succeeding blocks are physical neighbors

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).

Before

node n-1 in list

node n in list

allocated / free

split

selected free block

node n+1 in list

After (with splitting)

4 pointers updated:
2 in node n
1 in node n-1
1 in node n+1

still node n in list

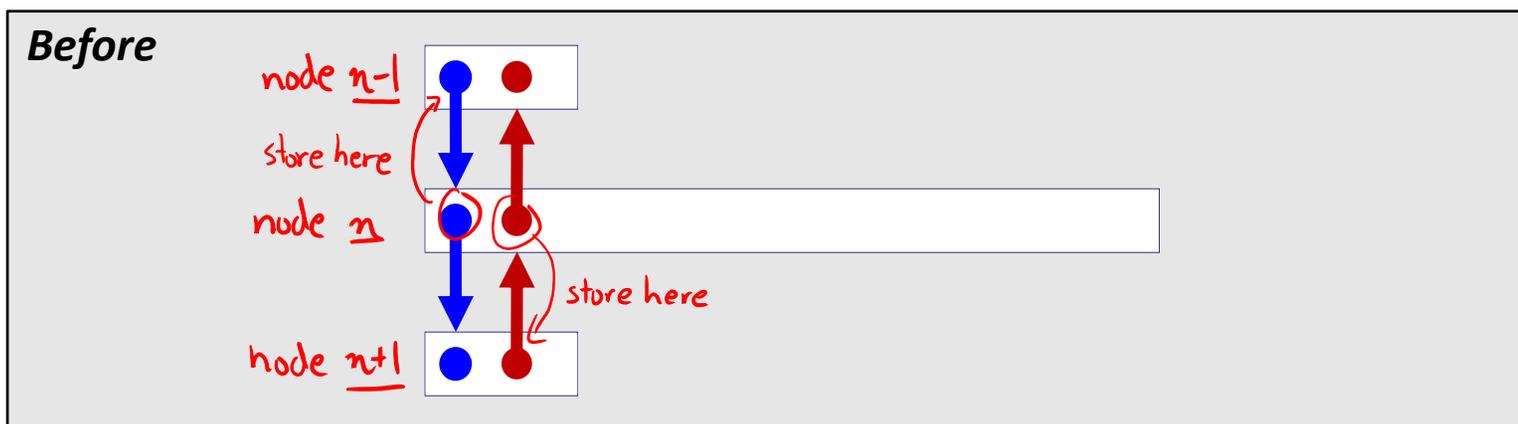Same number of nodes in free list

= malloc(...)

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).

Before

node *n-1*

store here

node *n*

store here

node *n+1*

After (fully allocated)

still node *n-1*

2 pointers updated
1 fewer node in free list

now the new node *n*

= malloc(…)

Freeing With Explicit Free Lists

❖ *Insertion policy:* Where in the free list do you put the newly freed block?
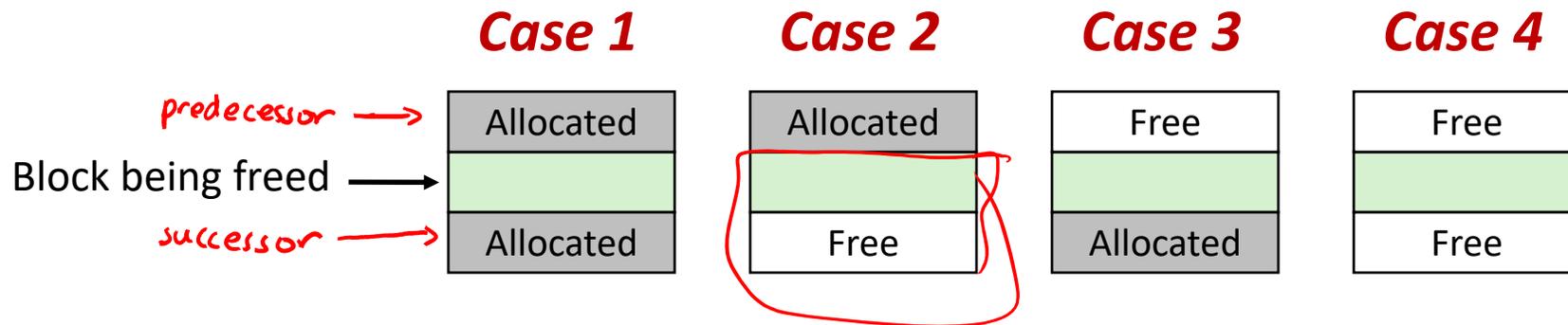
- **LIFO (last-in-first-out) policy**
  - Insert freed block at the beginning (head) of the free list
  - <u>Pro</u>: simple and constant time
  - <u>Con</u>: studies suggest fragmentation is worse than the alternative

- **Address-ordered policy**
  - Insert freed blocks so that free list blocks are always in address order:
    address(previous) < address(current) < address(next)
  - <u>Con</u>: requires linear-time search
  - <u>Pro</u>: studies suggest fragmentation is better than the alternative

Coalescing in Explicit Free Lists

Case 1	*Case 2*	*Case 3*	*Case 4*

predecessor ⟶

Block being freed ⟶

successor ⟶

Case 1	Case 2	Case 3	Case 4
Allocated	Allocated	Free	Free
Allocated	Free	Allocated	Free

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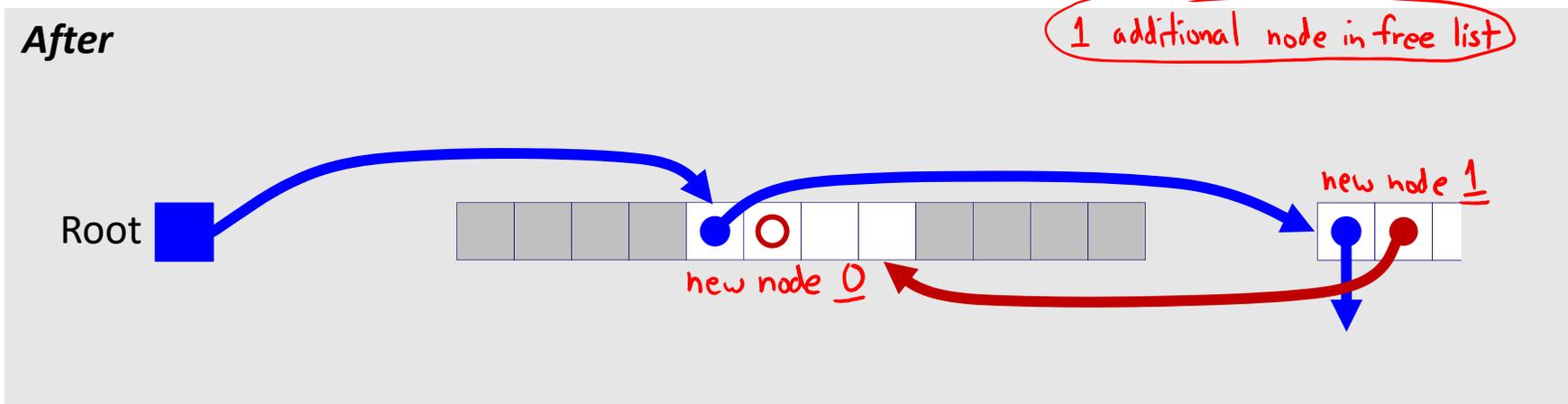
- ❖ How do we tell if a neighboring block if free?

can still use boundary tags (don't need to search free list).   other implementations possible (see Lab5)

21

Freeing with LIFO Policy (Case 1)

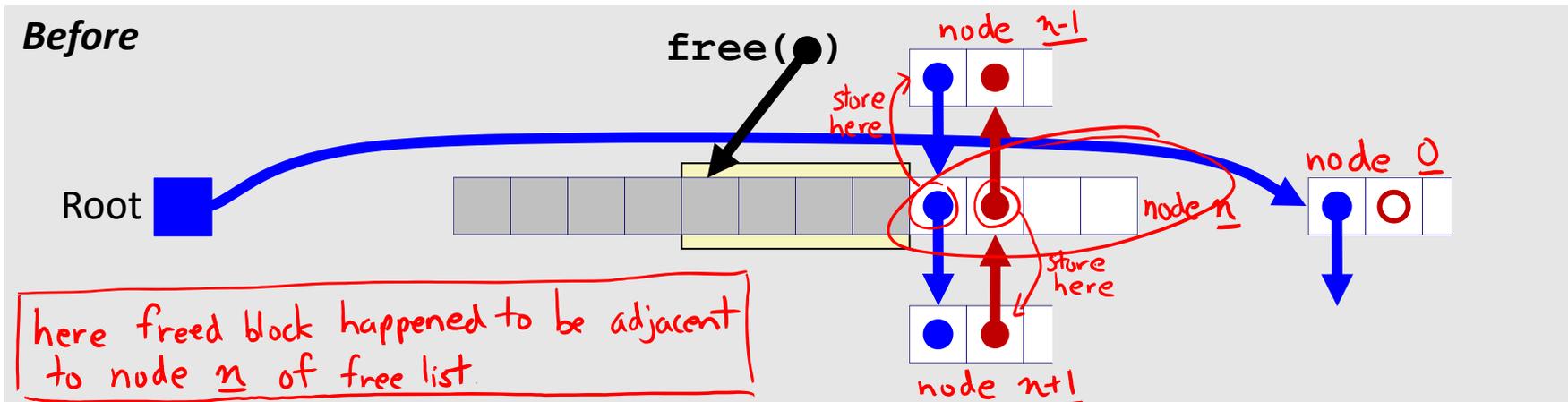Boundary tags not shown, but don't forget about them!

Before

free(●)

Root

node 0
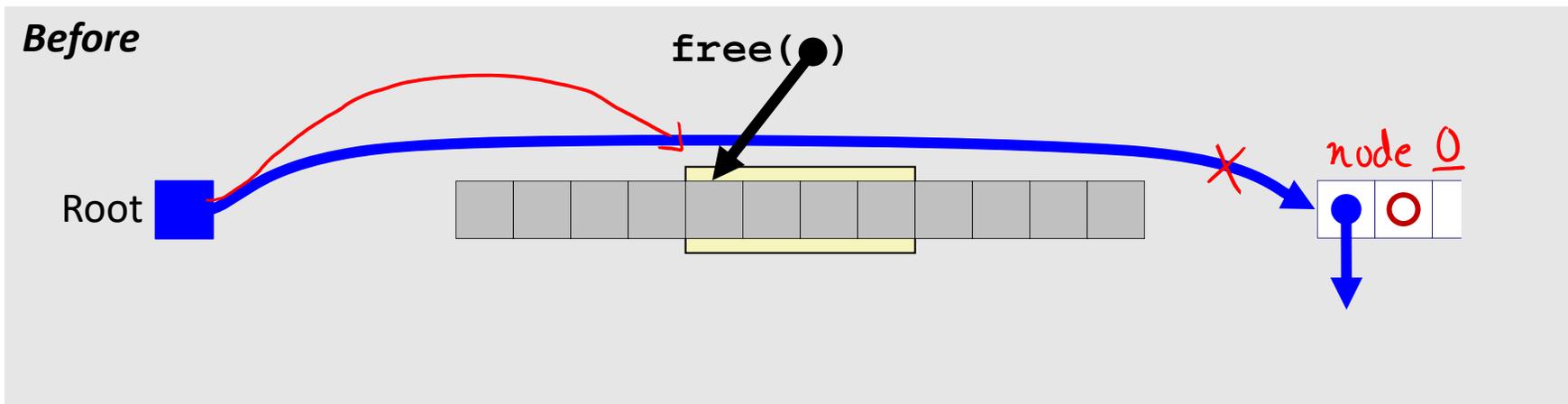
❖ Insert the freed block at the root of the list

After

1 additional node in free list

Root

new node 1

new node 0

Freeing with LIFO Policy (Case **2**)

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

Before

free(●)

node *n-1*

store here

Root

node *0*

node *n*

store here
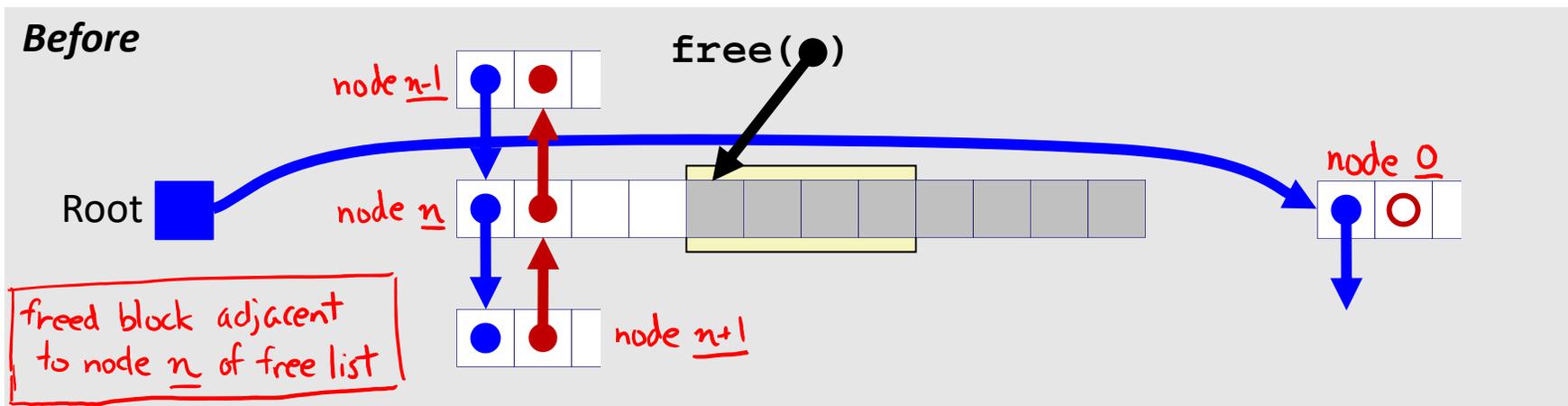
here freed block happened to be adjacent to node *n* of free list.

node *n+1*

❖ 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

After

new node *n*

①

new node *1*

④

③

Root

new node *0*

②

⑤

node *n+1*
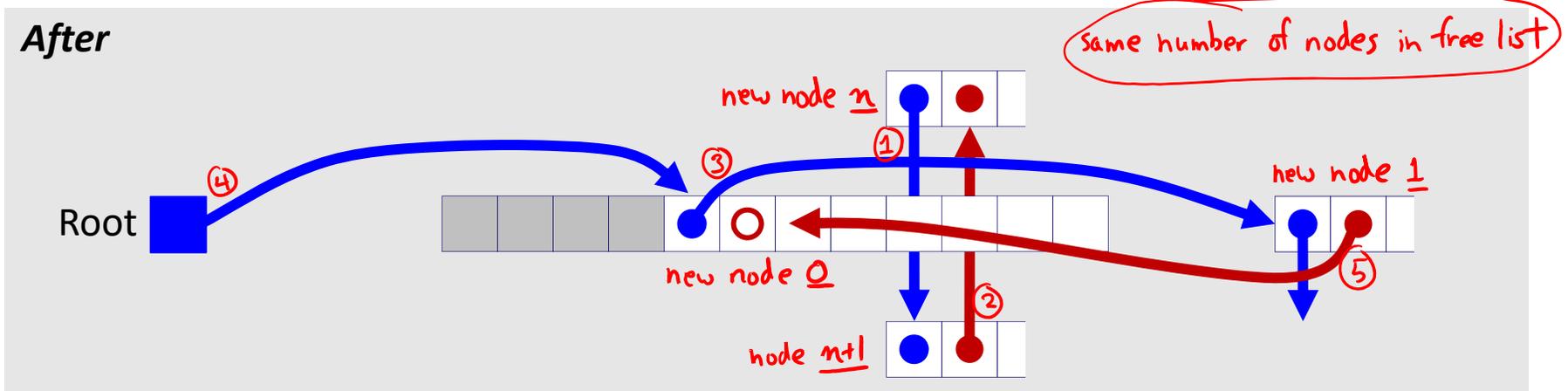
Freeing with LIFO Policy (Case **3**)

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

Before

free(●)

node n-1

Root

node n

node 0

freed block adjacent to node n of free list

node n+1

❖ Splice *predecessor* block out of list, coalesce both memory blocks, and insert the new block at the root of the list
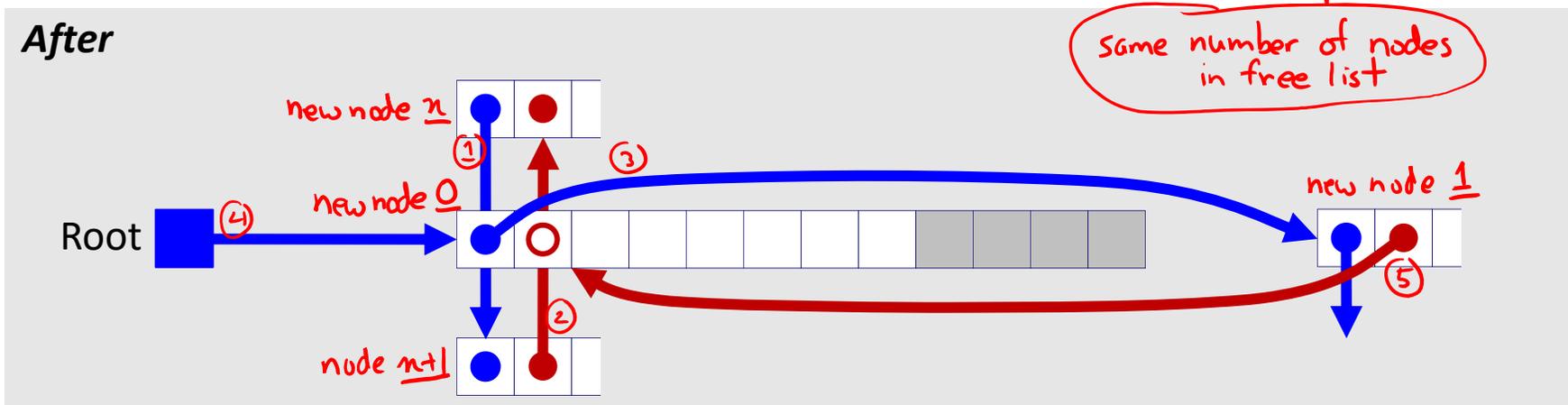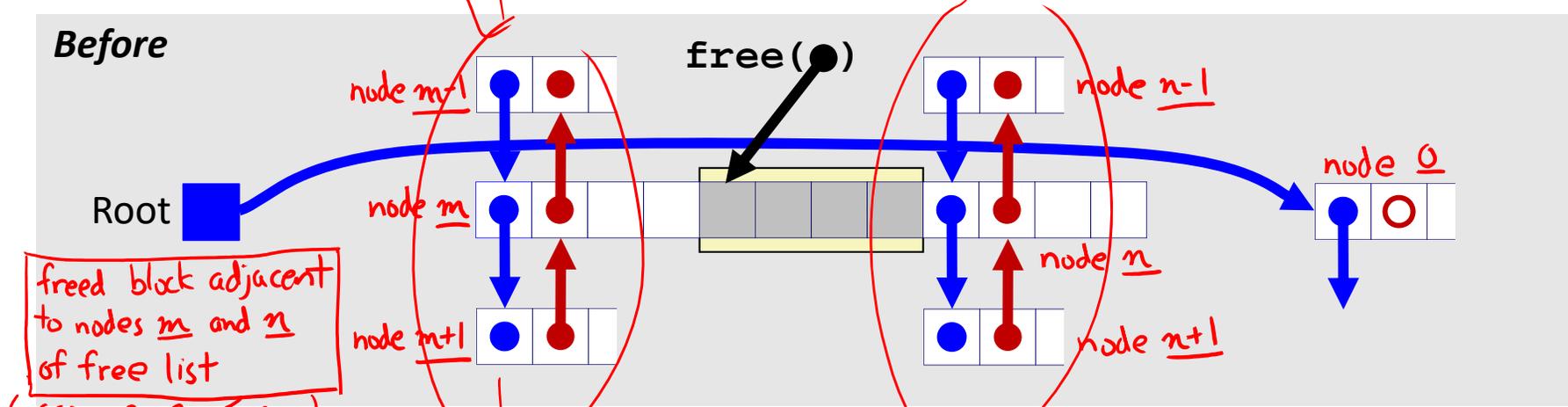
5 pointers updated

Same number of nodes in free list

After

new node n

①

③

new node 0

④

new node 1

Root

②

⑤

node n+1

Freeing with LIFO Policy (Case 4)

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

Before

node m-1

free(●)

node n-1

node 0

Root

node m

node n

freed block adjacent to nodes m and n of free list
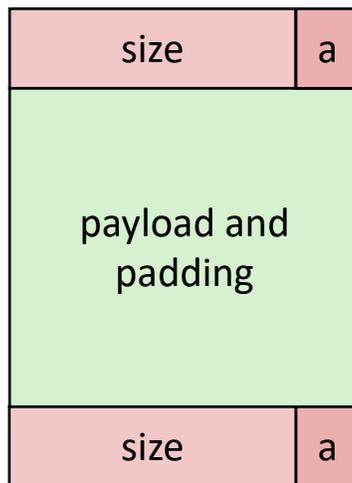
(assume m < n)

node m+1

node n+1

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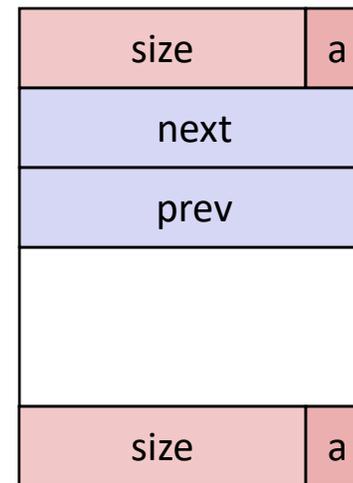
7 pointers updated

1 fewer node in free list

After

new node m

node n-1

① ⑤

③

new node 1

Root ⑥

new node 0

② ④ ⑦

node m+1

new node n
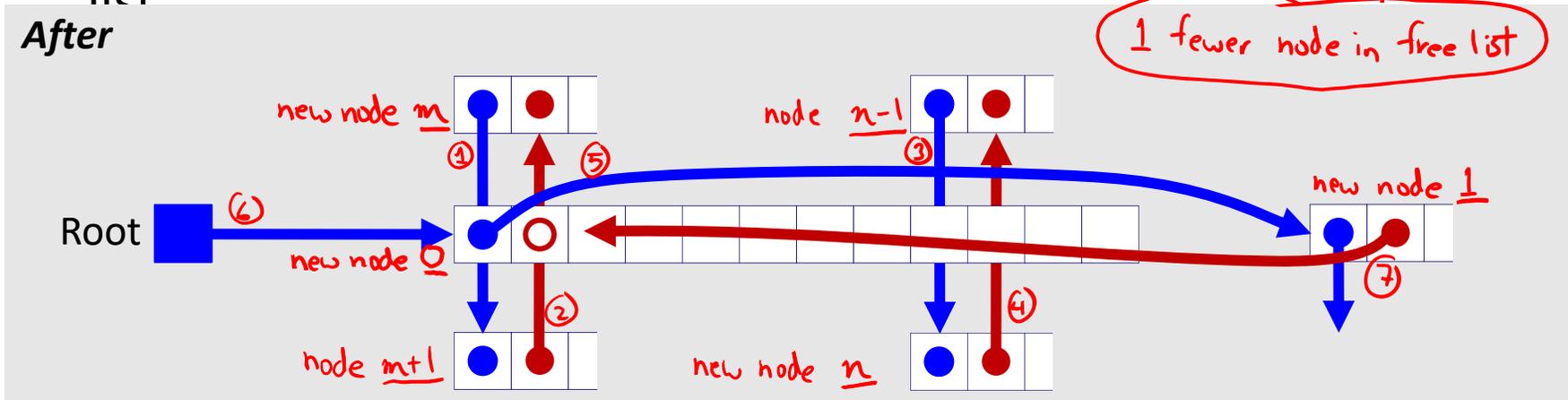
Do we always need the boundary tags?

Allocated block:

size	a
payload and padding	
size	a

(same as implicit free list)

Free block:

| size	a
next	
prev	
size	a

❖ 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
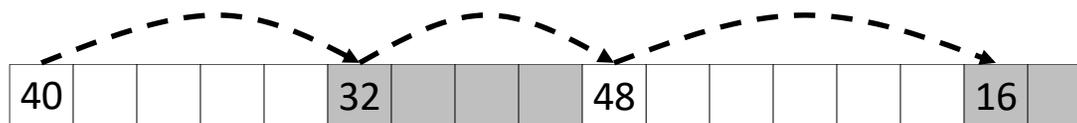
BONUS SLIDES

The following slides are about the **SegList Allocator**, for those curious. You will NOT be expected to know this material.
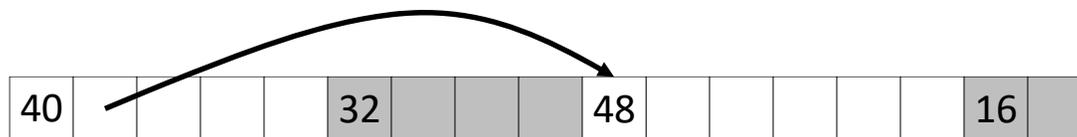
Keeping Track of Free Blocks
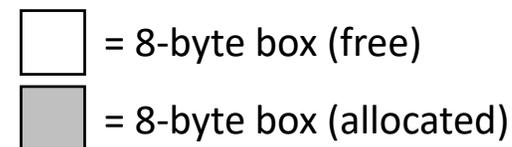
□ = 8-byte box (free)

▨ = 8-byte box (allocated)

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

| 40 | | | | 32 | | | | 48 | | | | | 16 | |

2) *Explicit free list* among <u>only the free blocks</u>, using pointers

| 40 | | | | 32 | | | | 48 | | | | | 16 | |

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 <u>array of free lists</u>

Size class
(in bytes)

16

32

48-64

80-inf
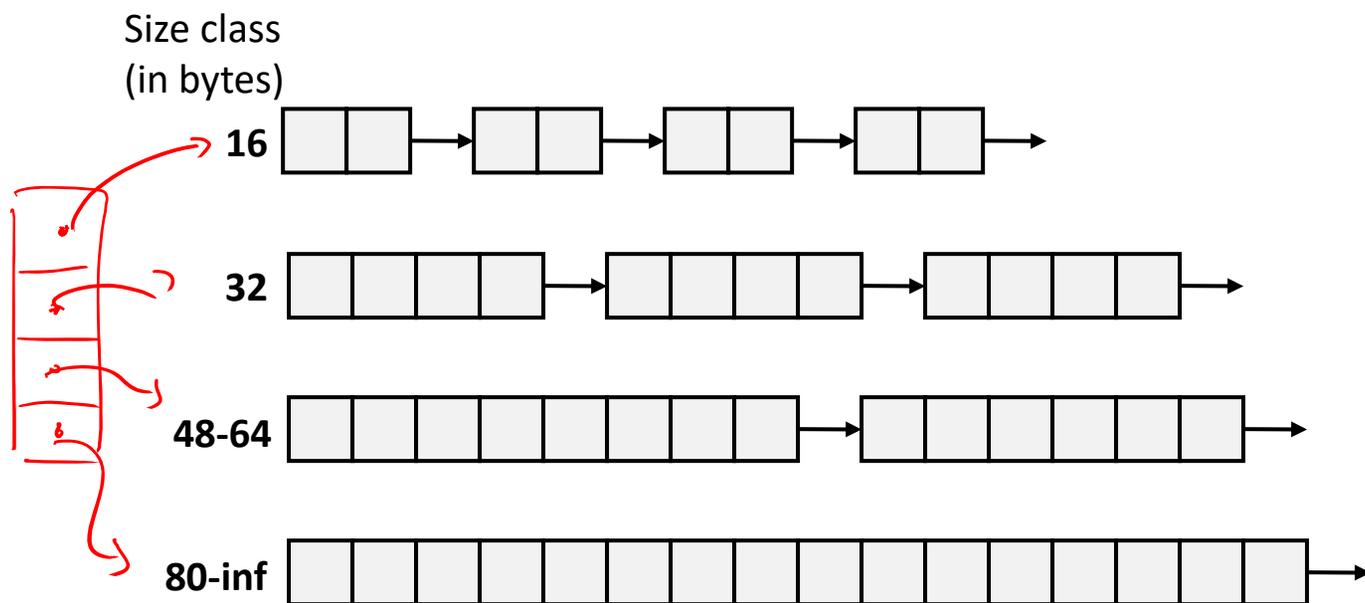
❖ Often have separate classes for each small size

❖ For larger sizes: One class for each two-power size

SegList Allocator

❖ Have an <u>array of free lists</u> for various size classes

❖ To <u>allocate</u> 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 <u>array of free lists</u> for various size classes

❖ To <u>free</u> 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