Dynamic Memory Allocation: Implicit and Explicit Free Lists

CSCI 237: Computer Organization
29th Lecture, Apr 30, 2025

Jeannie Albrecht

Administrative Details

- Lab 5 – cache lab
  - Due yesterday/today
- Lab 6 – malloc using explicit lists
  - Starts today/tomorrow
  - Starter code takes time to understand
  - Also helpful to read Ch 9.9
  - Think before you type!
- Glow HW due Friday
  - Pencil and paper!

Last time

- Dynamic Memory Allocation (Ch 9.9)
  - Basic concepts

Recap: Allocator Implementation Issues

1) How do we know how much memory to free given just a pointer?

   Use header that stored size and allocated/unallocated

2) How do we keep track of the free blocks?

3) What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?

4) How do we pick a block to use for allocation -- many might fit?

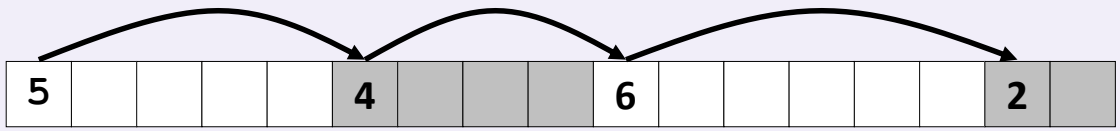
5) How do we reinsert freed block?

Today

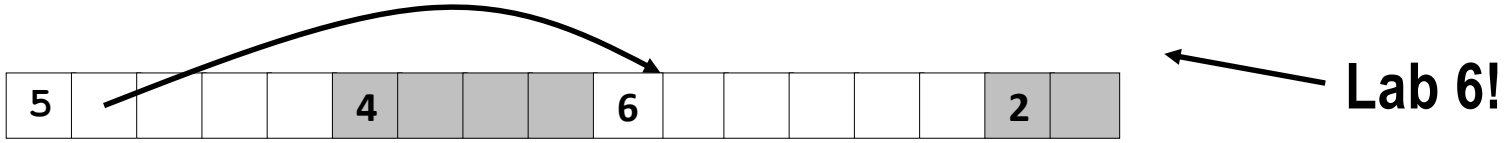
- Dynamic Memory Allocation (Ch 9.9)
  - Implicit free lists
  - Explicit free lists

2) Keeping Track of Free Blocks

- Method 1: *Implicit list* using length—links all blocks



- Method 2: *Explicit free list* among the free blocks using pointers



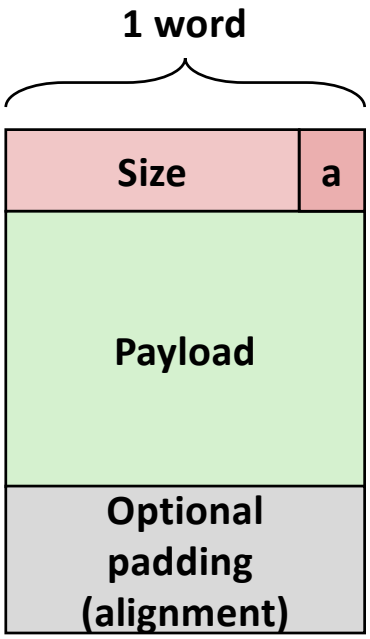
- Method 3: *Segregated free list*
  - Different free lists for different size classes

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

Method 1: Implicit (Free) List

- For each block we need both size and allocation status
  - Could store this information in two header words: wasteful!
- Standard trick to save space
  - When blocks are aligned, some low-order address bits (3 for 8 byte alignment) are always 0 (because everything is an even multiple of 8)
  - Instead of storing an always-0 bit, use it as an allocated/free flag
  - When reading the size, we can "mask out" this bit and ignore it

Format of allocated and free blocks

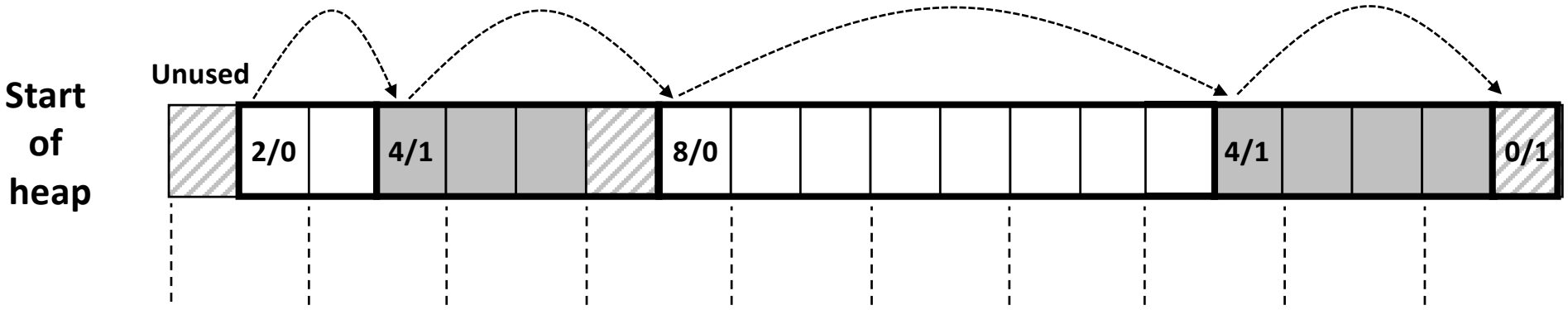


a = 1: Allocated block
a = 0: Free block

Size: block size

Payload: application data (allocated blocks only)

Detailed Implicit List Example



Payloads are double-word aligned

Allocated blocks: shaded
Free blocks: unshaded
Striped blocks: unused/padding/header (*not* payload)
Headers: labeled with "size in words/allocated bit"

Headers are at non-aligned positions
Payloads (aka "actual" data being stored) are aligned

Allocator Implementation Issues

1) How do we know how much memory to free given just a pointer?

2) How do we keep track of the free blocks?

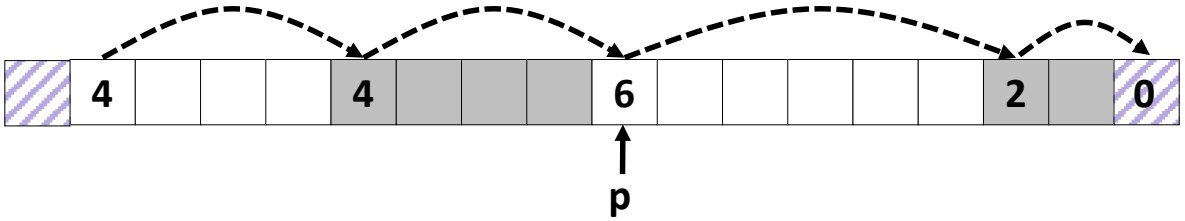
3) What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?

4) How do we pick a block to use for allocation -- many might fit?

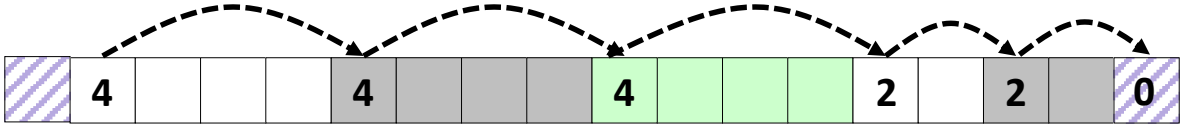
5) How do we reinsert freed block?

3) Implicit List: Allocating in Free Block

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



`add_block(p, 4)`




```
void add_block(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;  // round up to even
  int oldsize = *p & -2;                // mask out low bit
  *p = newsize | 1;                     // set new length + alloc
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;   // set length in remaining
}                                       //   part of block
```

4) Implicit List: Finding a Free Block (9.9.7)

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

```
p = start;
while ((p < end) &&      \\ not passed end
       ((*p & 1) ||      \\ already allocated
       (*p  <= len)))    \\ too small
  p = p + (*p & -2);     \\ goto next block (word addressed)
```

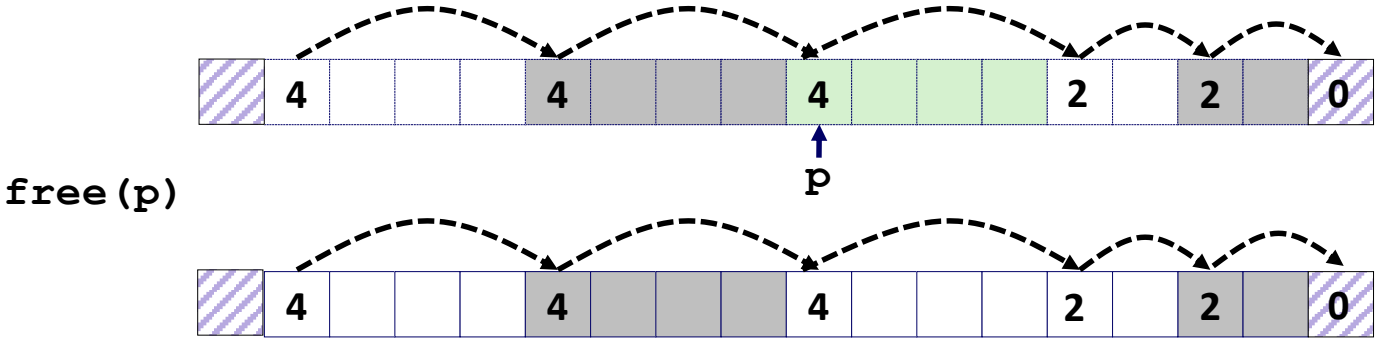
  - Can take linear time in total number of blocks (allocated and free)
  - In practice it can cause "splinters" at beginning of list
- *Next fit:*
  - Like first fit, but search list starting where previous search finished
  - Should often be faster than first fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is surprisingly worse however
- *Best fit:*
  - Search the list, choose the ***best*** free block: fits, with fewest bytes left over
  - Keeps fragments small—usually improves memory utilization
  - Will typically run slower than first fit because we have to search entire list

5) Implicit List: Freeing a Block

- Simplest implementation:
  - Need only clear the "allocated" flag

```
void free_block(ptr p) { *p = *p & -2 }
```

  - But can lead to "false fragmentation"



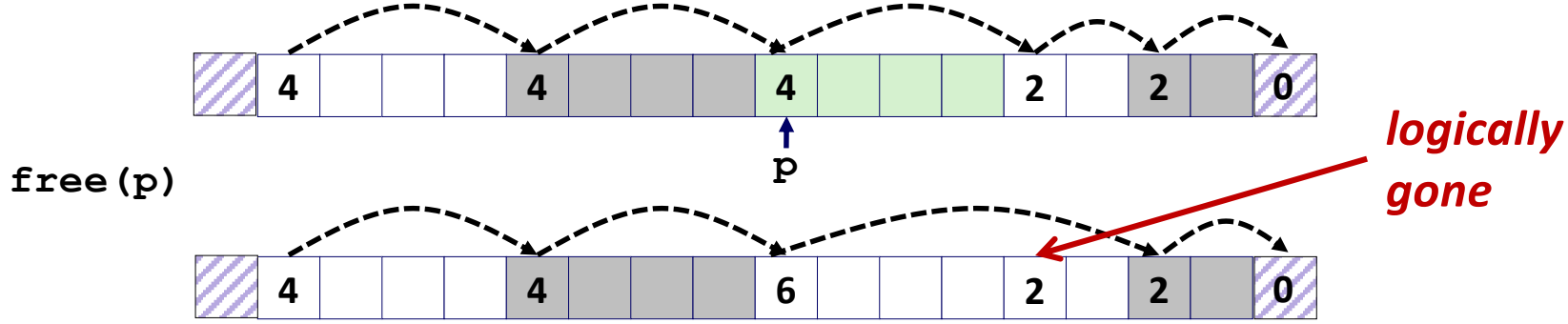

```
malloc(5*SIZ)
```

Oops!

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

Implicit List: Coalescing

- Join *(aka coalesce)* with next/previous blocks, but only if they are free
  - Coalescing with next block

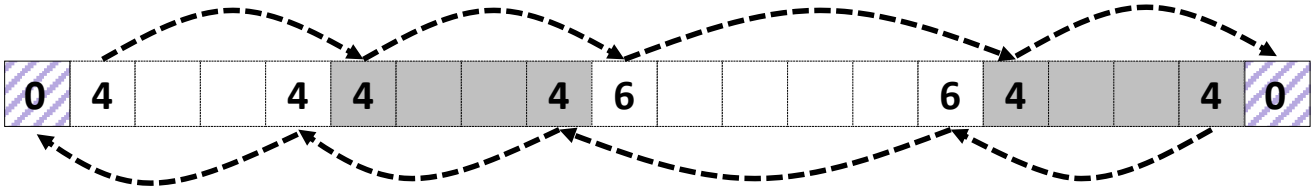



```
void free_block(ptr p) {
    *p = *p & -2;              // clear allocated flag
    next = p + *p;             // find next block
    if ((*next & 1) == 0)
      *p = *p + *next;         // add to this block if
}                              //    not allocated
```

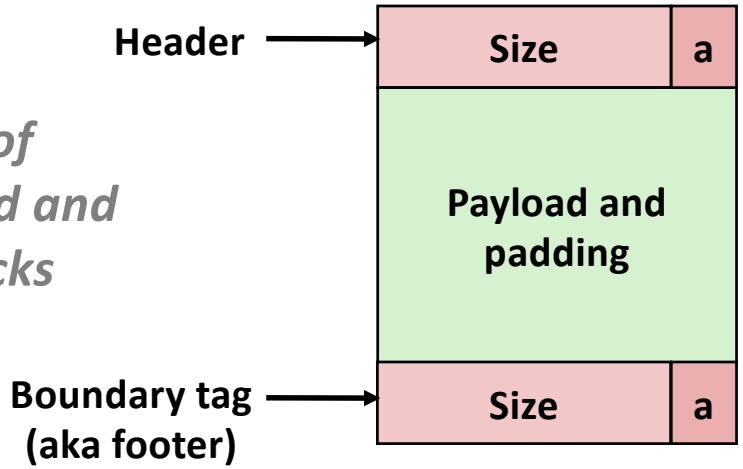
  - But how do we coalesce with *previous* block?

Implicit List: Bidirectional Coalescing

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



Format of allocated and free blocks

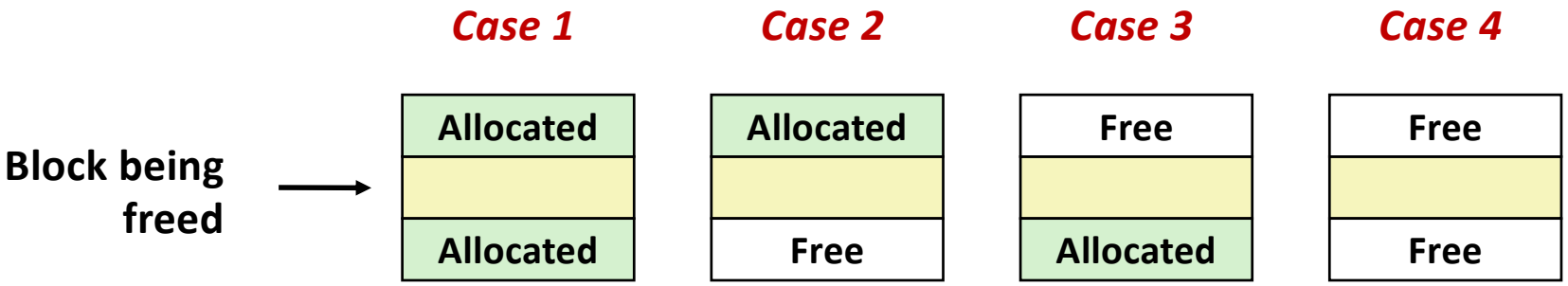


a = 1: Allocated block
a = 0: Free block

Size: Total block size

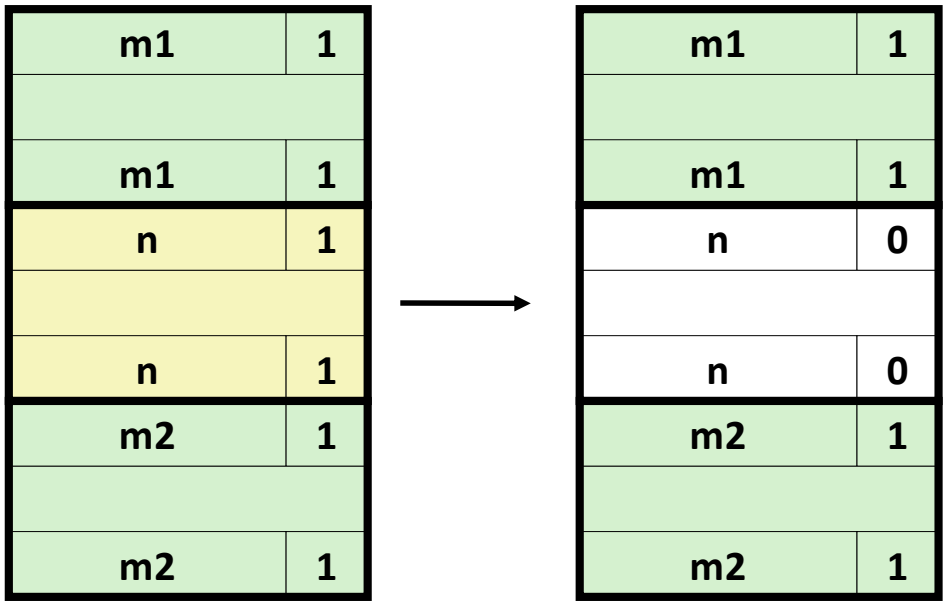
Payload: Application data (allocated blocks only)

Constant Time Coalescing

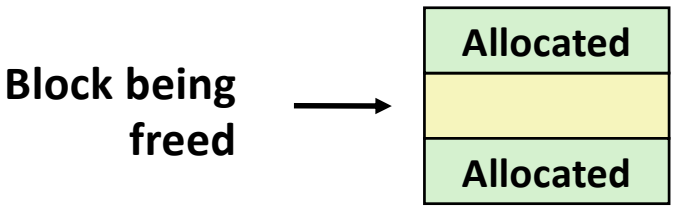


Given a block to free and its two neighbors, there are 4 unique combinations of free/allocated to consider. Let's look at each case individually.

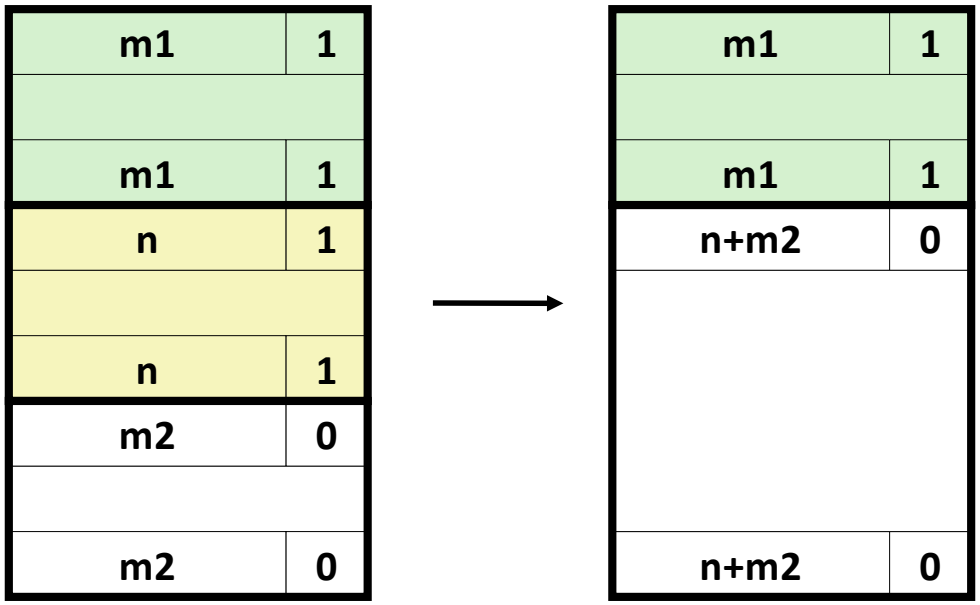
Constant Time Coalescing (Case 1)



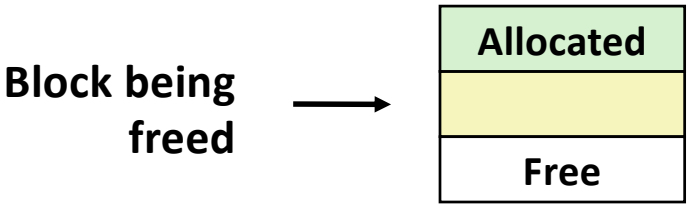
Case 1



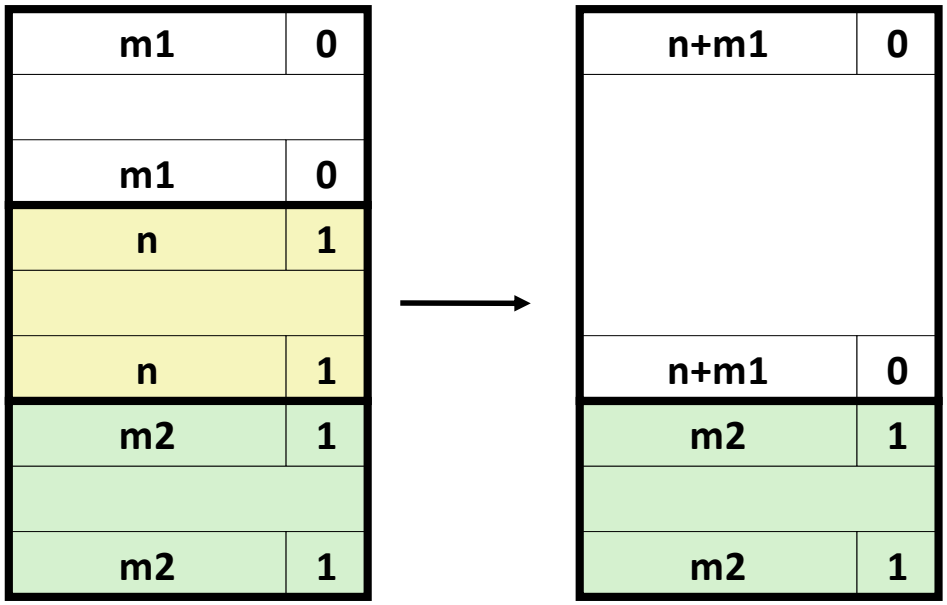
Constant Time Coalescing (Case 2)



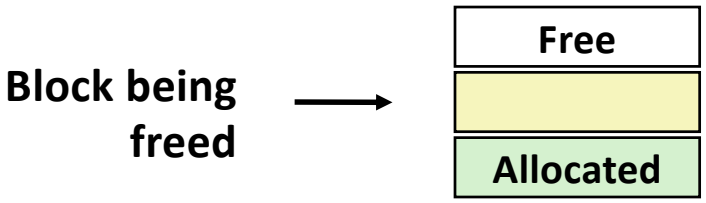
Case 2



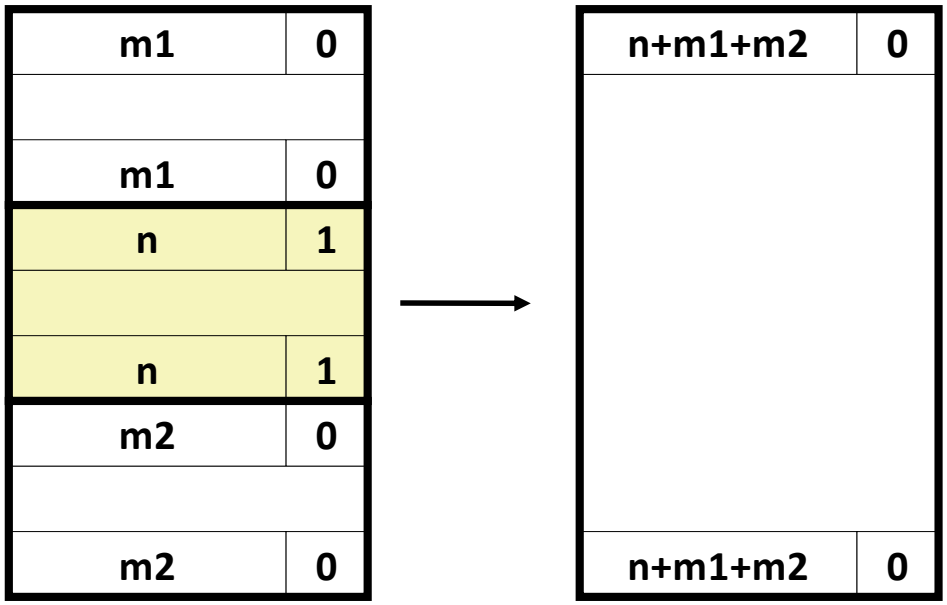
Constant Time Coalescing (Case 3)



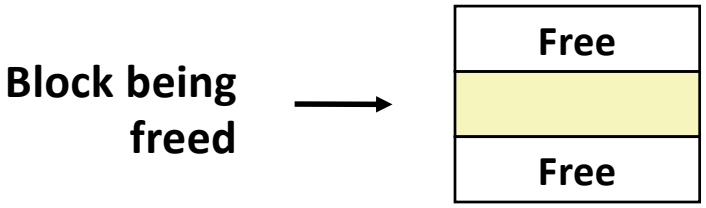
Case 3



Constant Time Coalescing (Case 4)



Case 4



Disadvantages of Boundary Tags

- Internal fragmentation

- Can it be optimized?
  - Which blocks need the footer tag?
  - What does that mean?

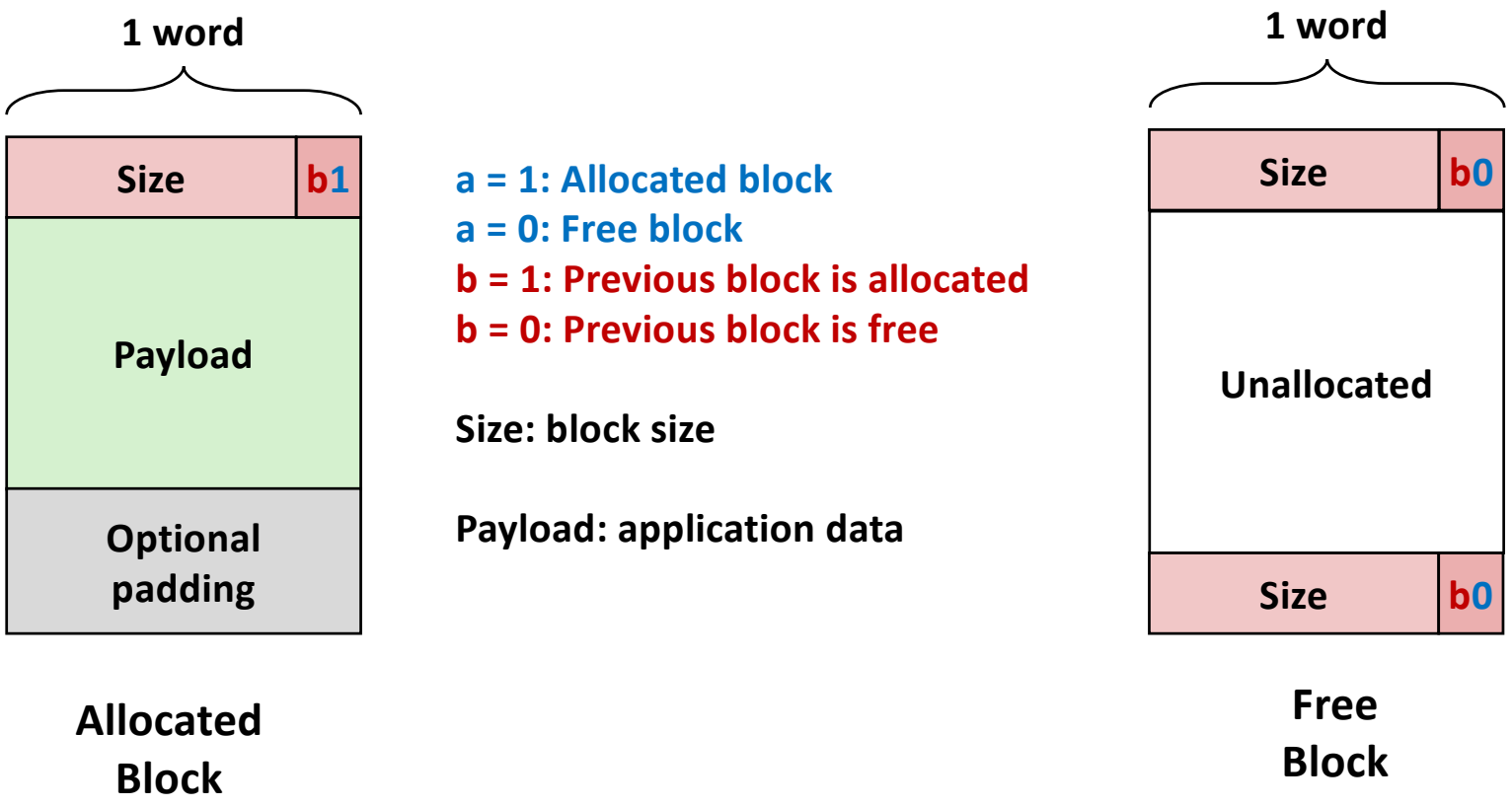
Disadvantages of Boundary Tags

- Internal fragmentation

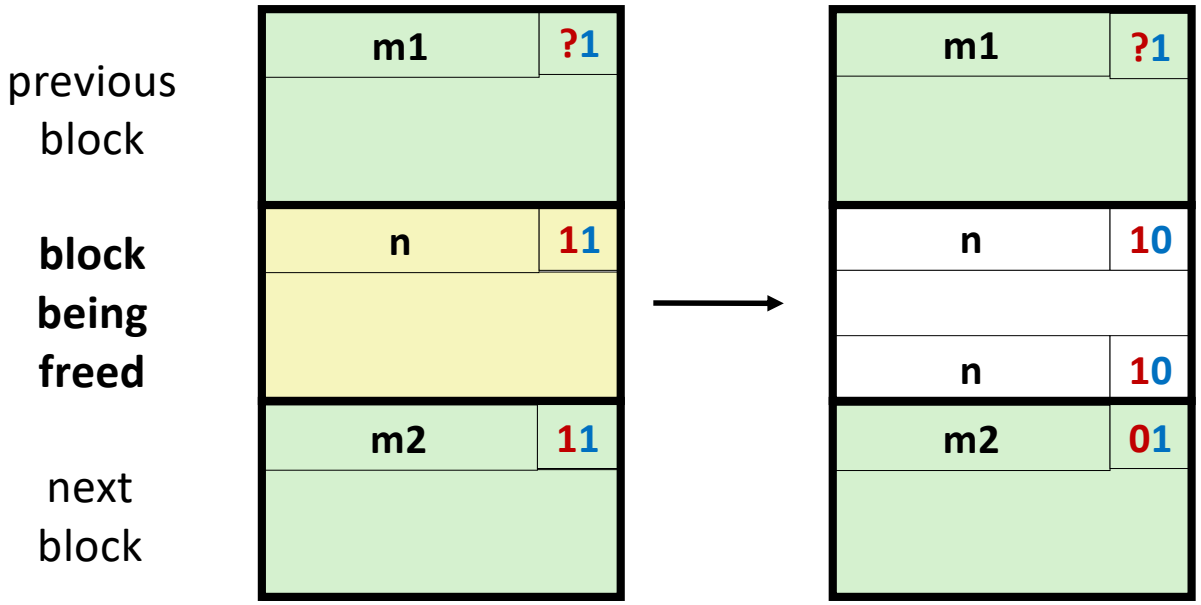
- Can it be optimized?
  - Which blocks need the footer tag? Only free blocks!
  - What does that mean? Can save space!

No Boundary Tag for *Allocated* Blocks

- Boundary tag needed only for **free** blocks
- When sizes are multiples of 4 or more, have 2+ spare bits

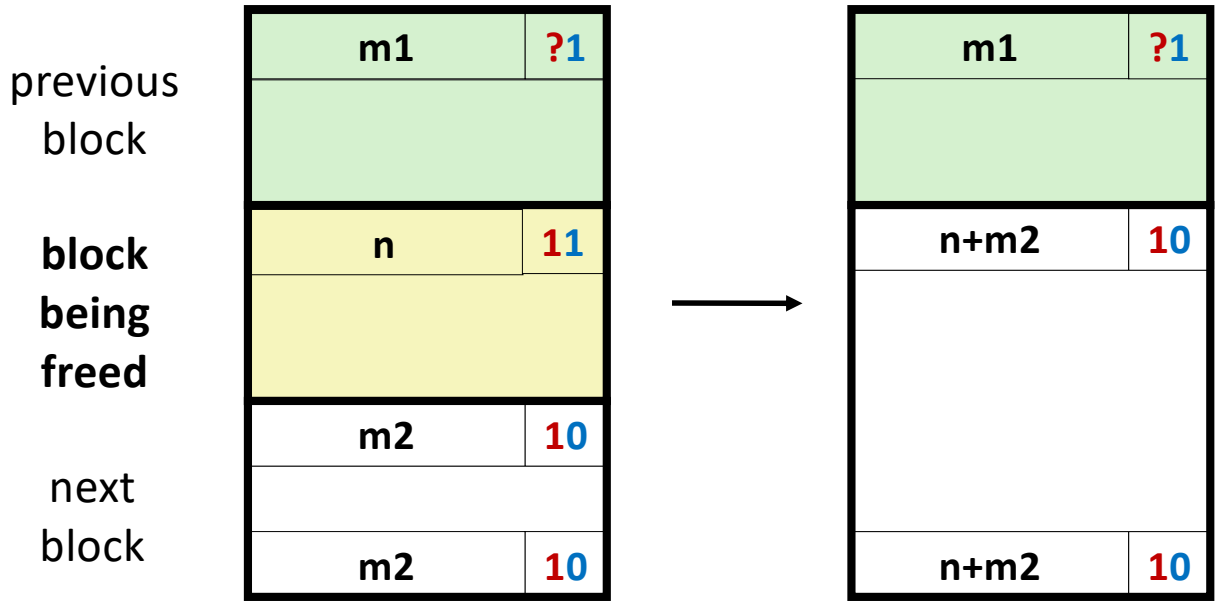


No Boundary Tag for Allocated Blocks (Case 1)



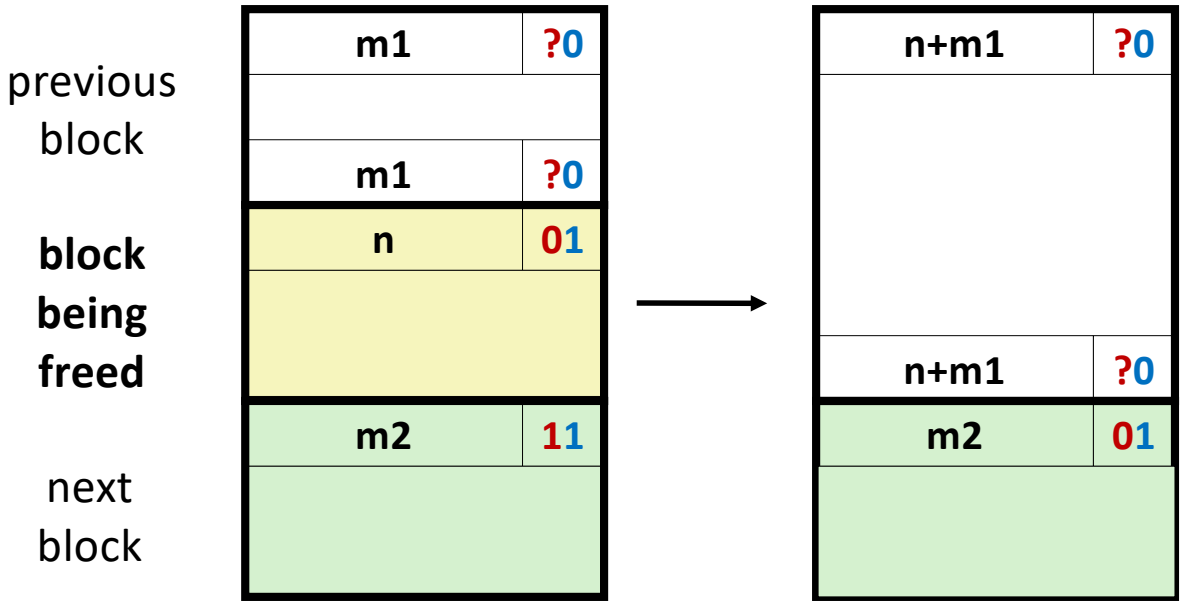
Header: Use 2 bits (always zero due to alignment):
(previous block allocated)<<1 | (current block allocated)

No Boundary Tag for Allocated Blocks (Case 2)



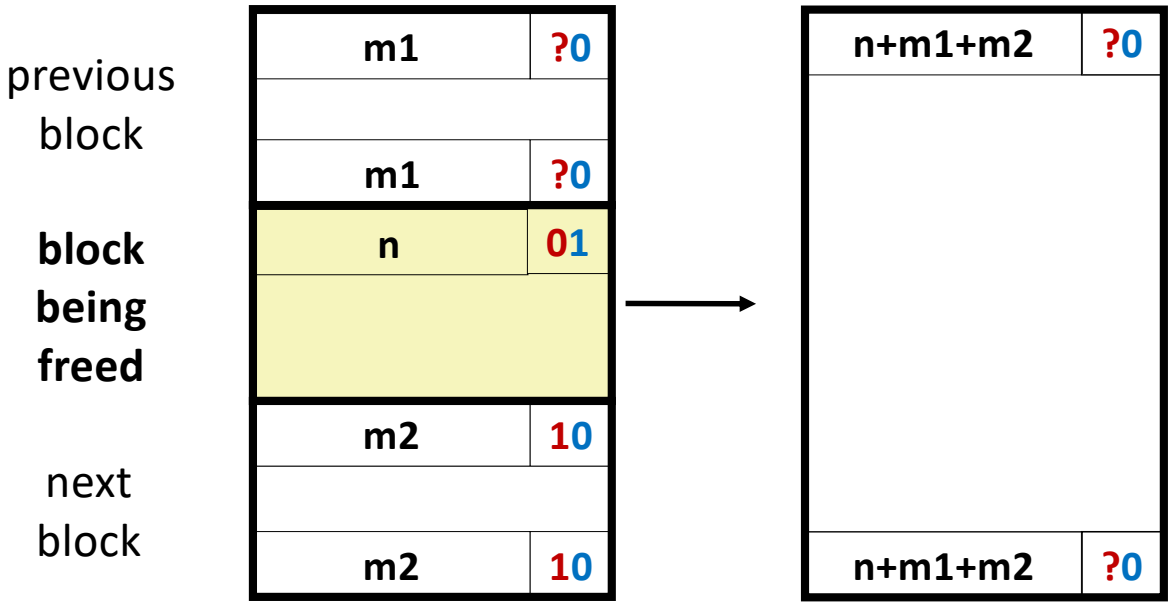
Header: Use 2 bits (always zero due to alignment):
(previous block allocated)<<1 | (current block allocated)

No Boundary Tag for Allocated Blocks (Case 3)



Header: Use 2 bits (always zero due to alignment):
(previous block allocated)<<1 | (current block allocated)

No Boundary Tag for Allocated Blocks (Case 4)



Header: Use 2 bits (always zero due to alignment):
(previous block allocated)<<1 | (current block allocated)

Summary of Key Allocator Policies

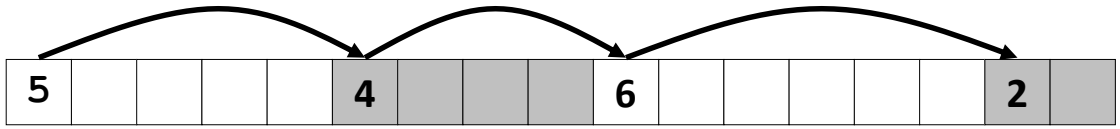
- Placement policy:
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
  - ***Interesting observation*:** segregated free lists (next lecture?) approximate a best fit placement policy without having to search entire free list
- Splitting policy:
  - When do we go ahead and split free blocks?
  - How much internal fragmentation are we willing to tolerate?
- Coalescing policy:
  - ***Immediate coalescing:*** coalesce each time **`free`** is called
  - ***Deferred coalescing:*** try to improve performance of **`free`** by deferring coalescing until needed. Examples:
    - Coalesce as you scan the free list for **`malloc`**
    - Coalesce when the amount of external fragmentation reaches some threshold

Implicit Lists: Summary

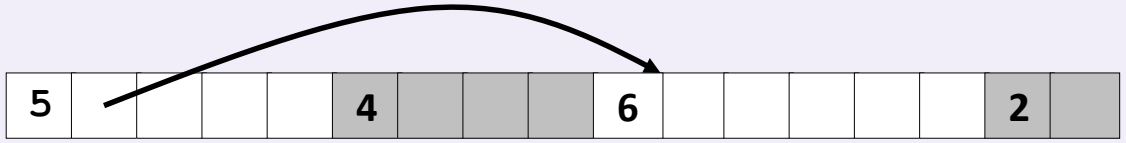
- Implementation: very simple
- Allocate cost:
  - linear time worst case
- Free cost:
  - constant time worst case
  - even with coalescing
- Memory usage:
  - will depend on placement policy
  - first-fit, next-fit or best-fit
- Not used in practice for `malloc/free` because of linear-time allocation
  - Still used in many special purpose applications
- However, the concepts of splitting and boundary tag coalescing are general to *all* allocators (LAB 6!!!!!)

Keeping Track of Free Blocks

- Method 1: *Implicit list* using length—links all blocks



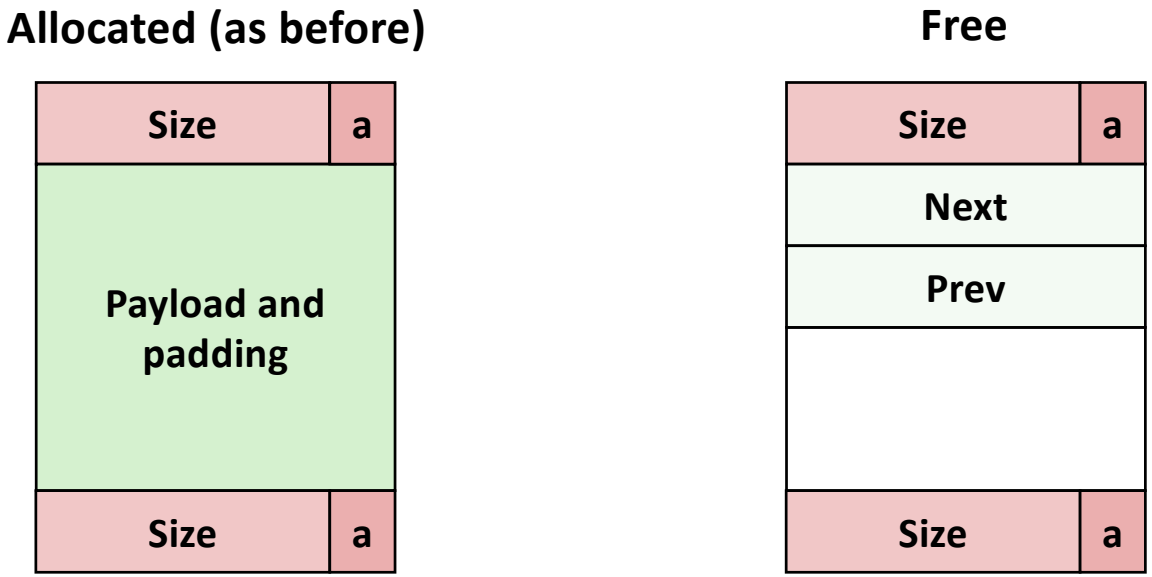
- Method 2: *Explicit free list* among the free blocks using pointers



- Method 3: *Segregated free list*
  - Different free lists for different size classes

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

Explicit Free Lists

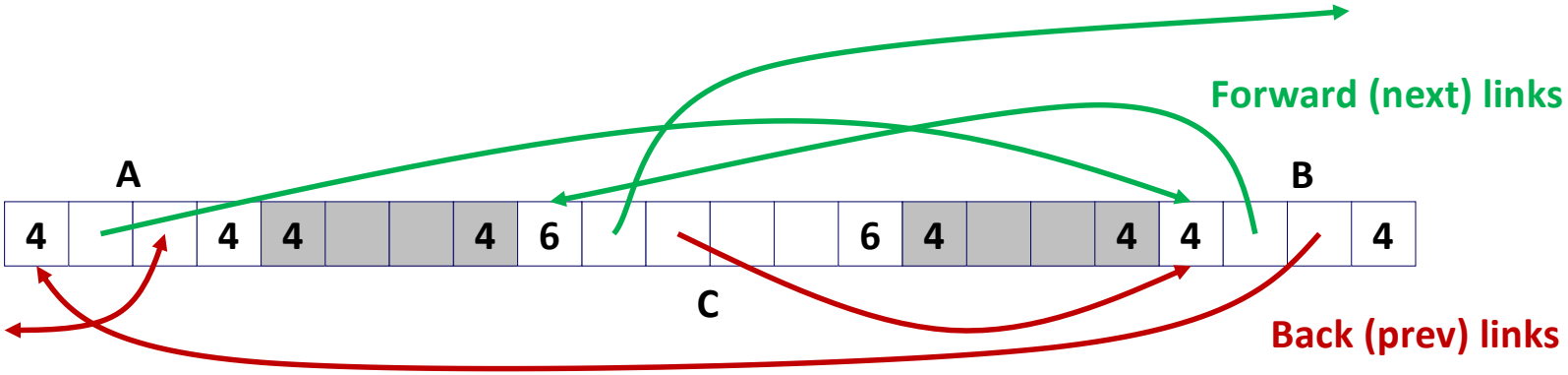


- Maintain list(s) of *free* blocks, not *all* blocks
  - The "next" free block could be anywhere
    - So we need to store forward/back pointers, not just sizes
    - Basically a doubly linked list
  - Still need boundary tags for coalescing
  - Luckily we track only free blocks, so we can use payload area

Explicit Free Lists

- Logically:

- Physically: blocks in free list can be in any order in physical reality

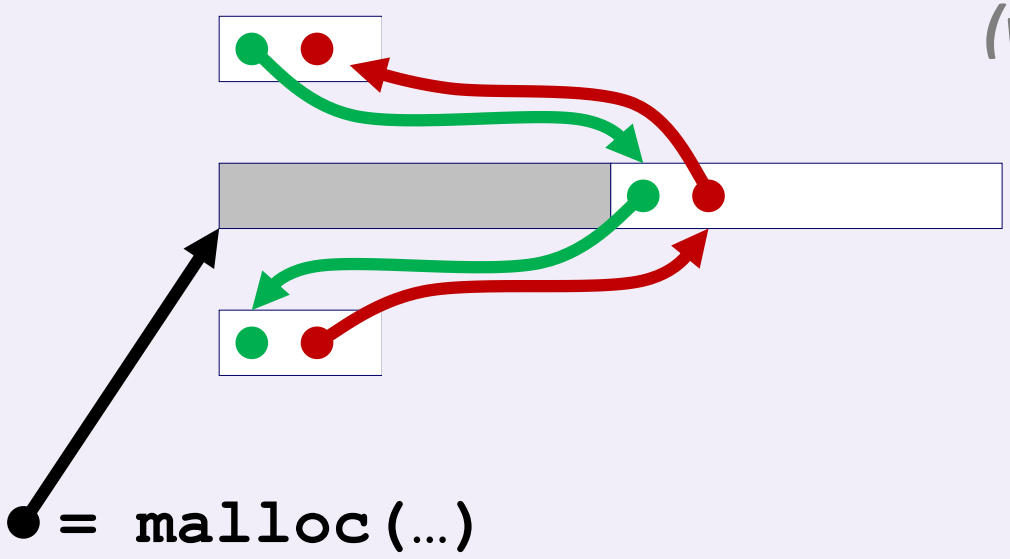


Allocating From Explicit Free Lists

conceptual graphic

Before

After *(with splitting)*

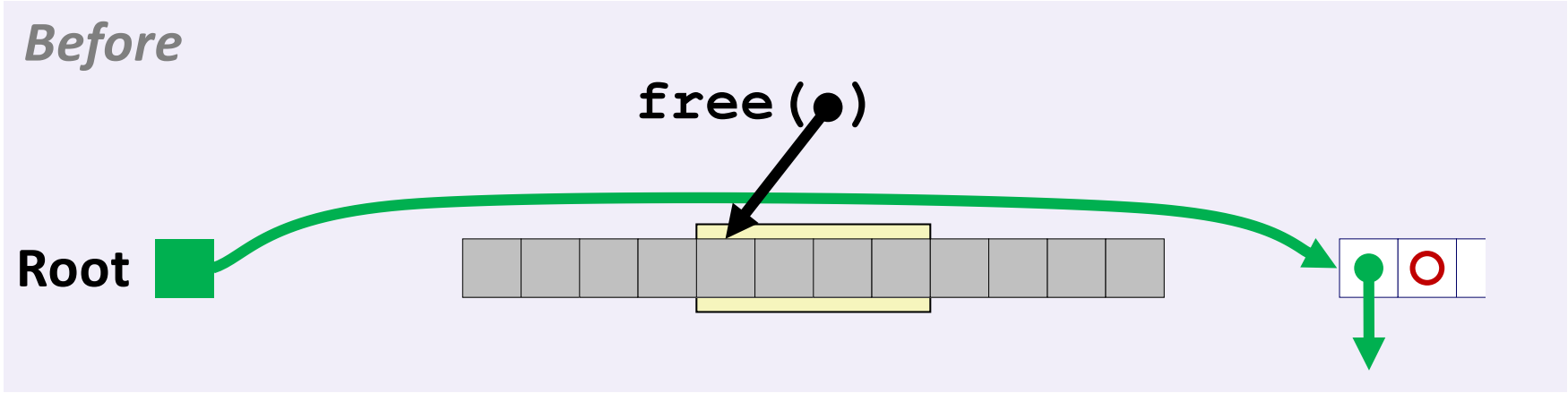


Freeing With Explicit Free Lists

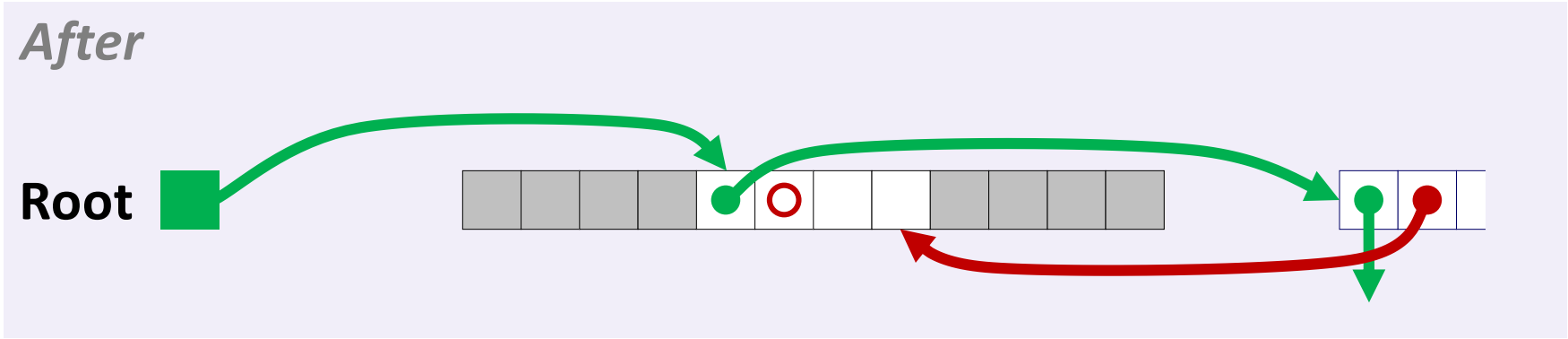
- *Insertion policy*: Where in the free list do we put a newly freed block?
- **LIFO (last-in-first-out) policy**
  - Insert freed block at the beginning of the free list
  - ***Pro:*** simple and constant time
  - ***Con:*** studies suggest fragmentation is worse than address ordered
- **Address-ordered policy**
  - Insert freed blocks so that free list blocks are always in address order:
    addr(prev) < addr(curr) < addr(next)
  - ***Con:*** requires search
  - ***Pro:*** studies suggest fragmentation is lower than LIFO

Freeing With a LIFO Policy (Case 1)

conceptual graphic

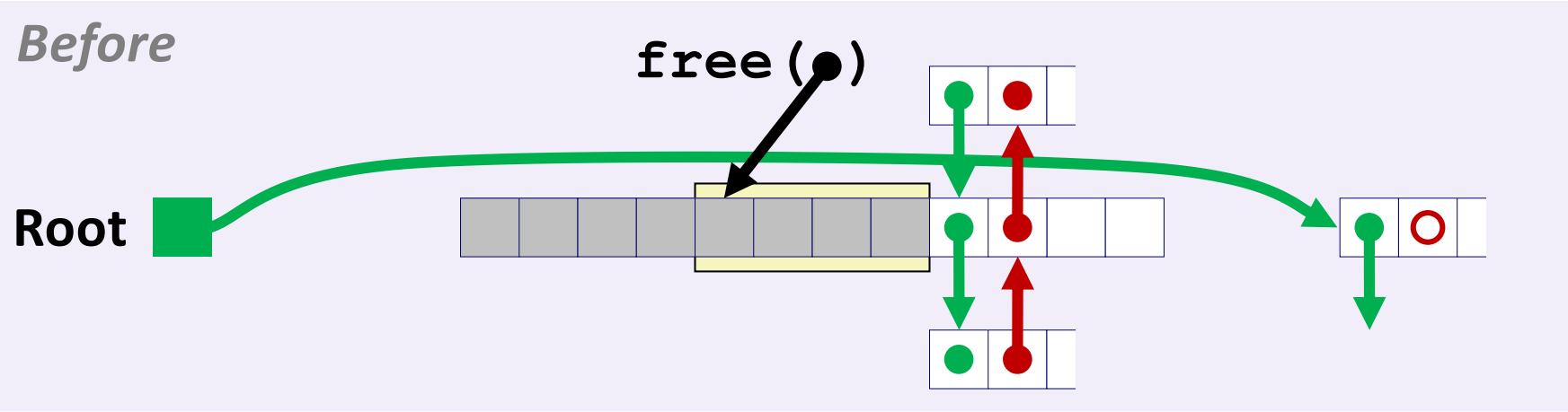


- Insert the freed block at the root (front) of the free list

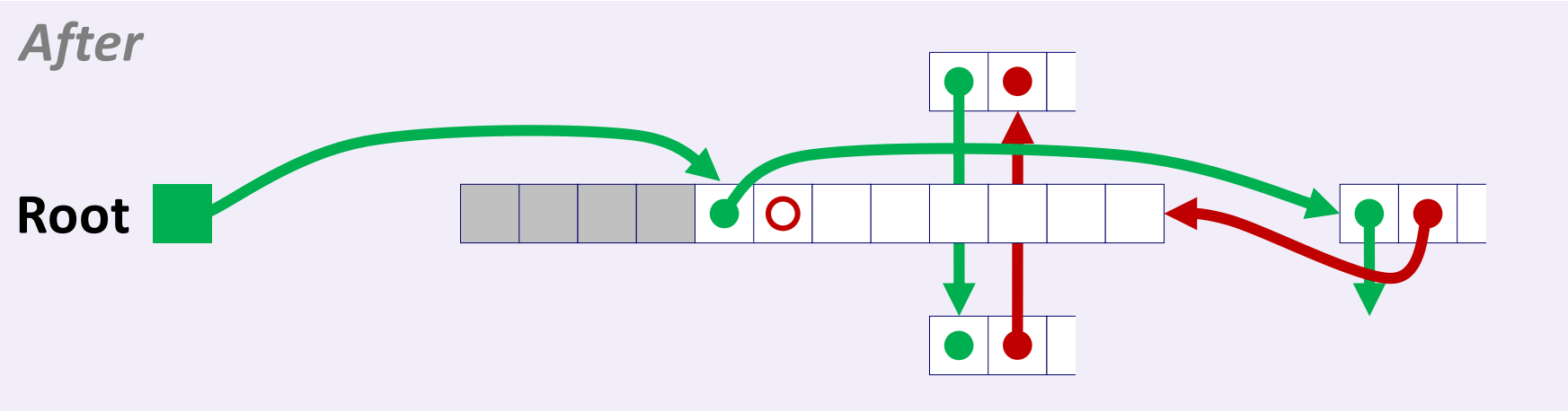


Freeing With a LIFO Policy (Case 2)

conceptual graphic

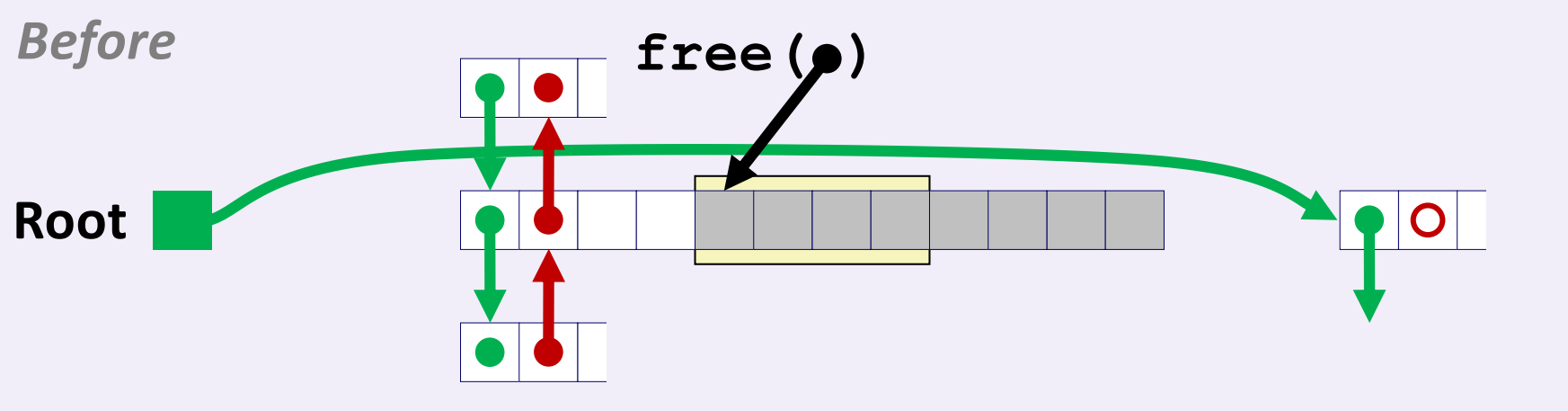


- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list

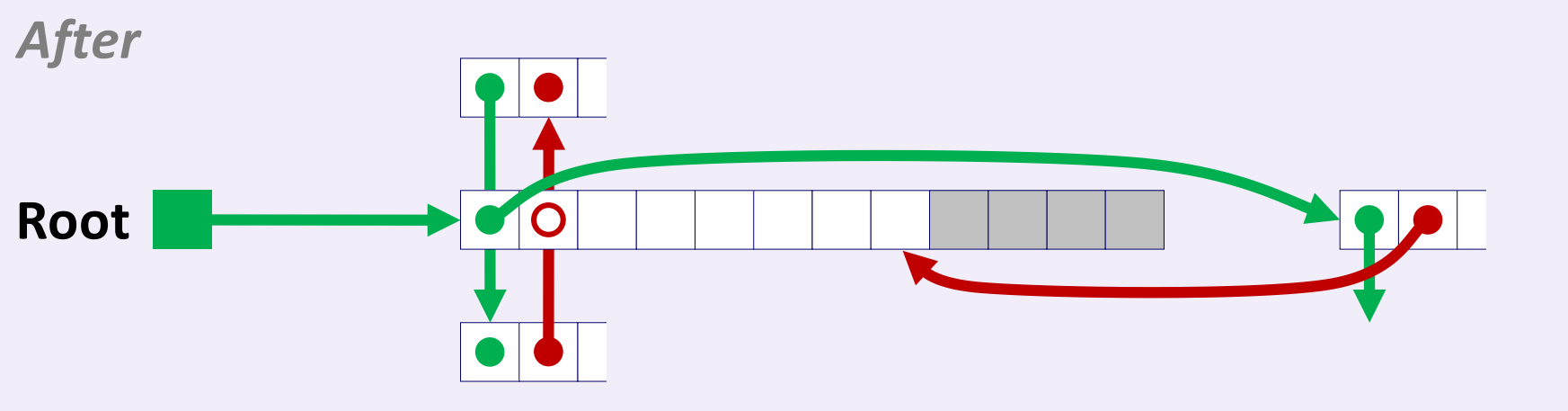


Freeing With a LIFO Policy (Case 3)

conceptual graphic

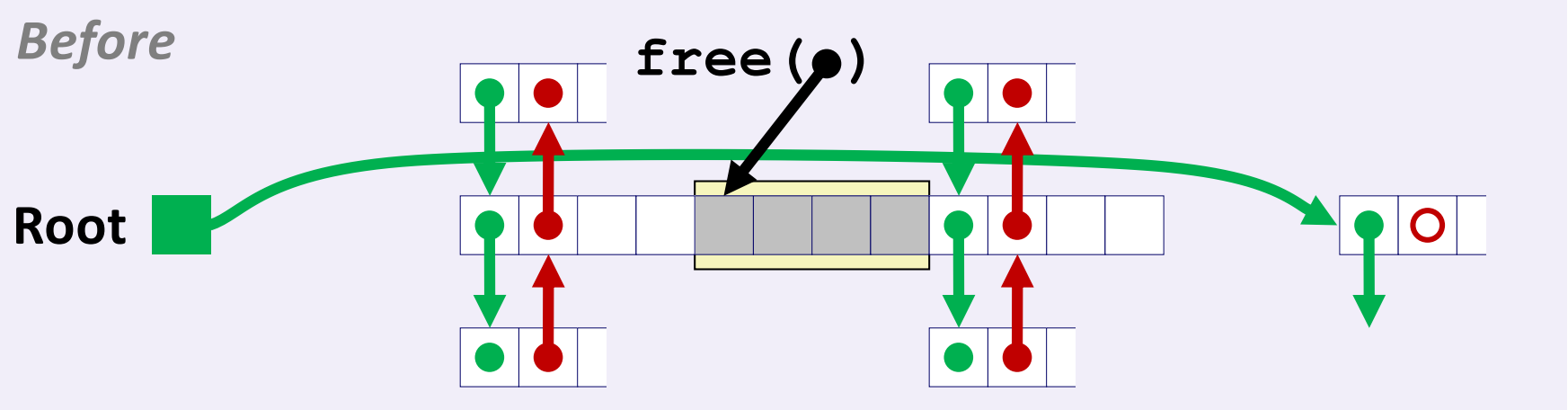


- Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list

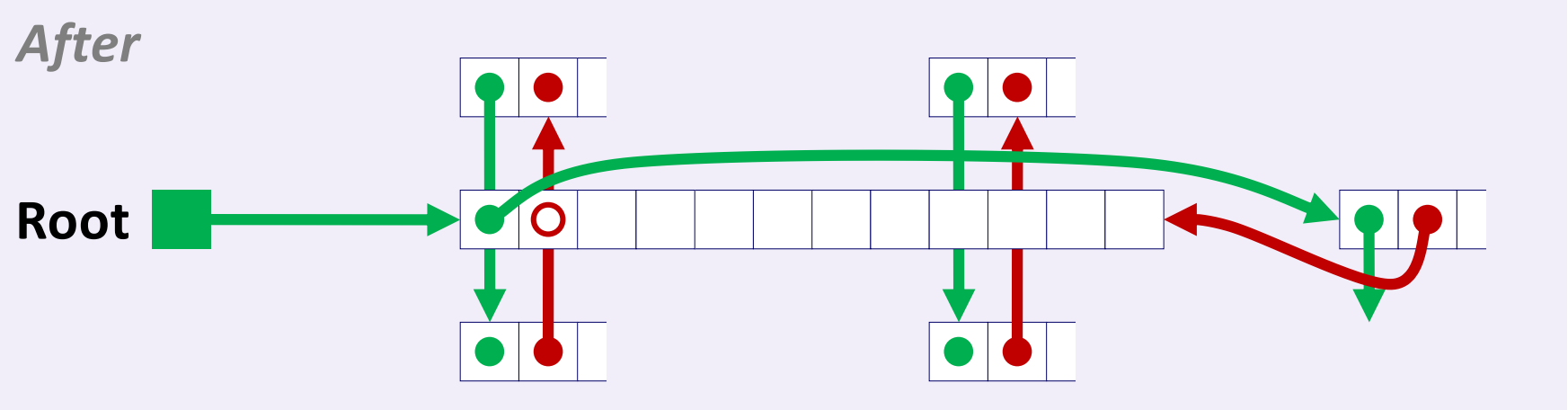


Freeing With a LIFO Policy (Case 4)

conceptual graphic



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



Explicit List Summary

- Comparison to implicit list:
  - Allocate 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 needs to splice blocks in and out of the list
  - Some extra space for the links (2 extra words needed for each block)
    - Does this increase internal fragmentation?

- One of most common uses of linked lists is in conjunction with segregated free lists
  - Keep multiple linked lists of different size classes, or possibly for different types of objects