Memory Allocation II

CSE 351 Winter 2018

Instructor:

Mark Wyse

Teaching Assistants:

Kevin Bi

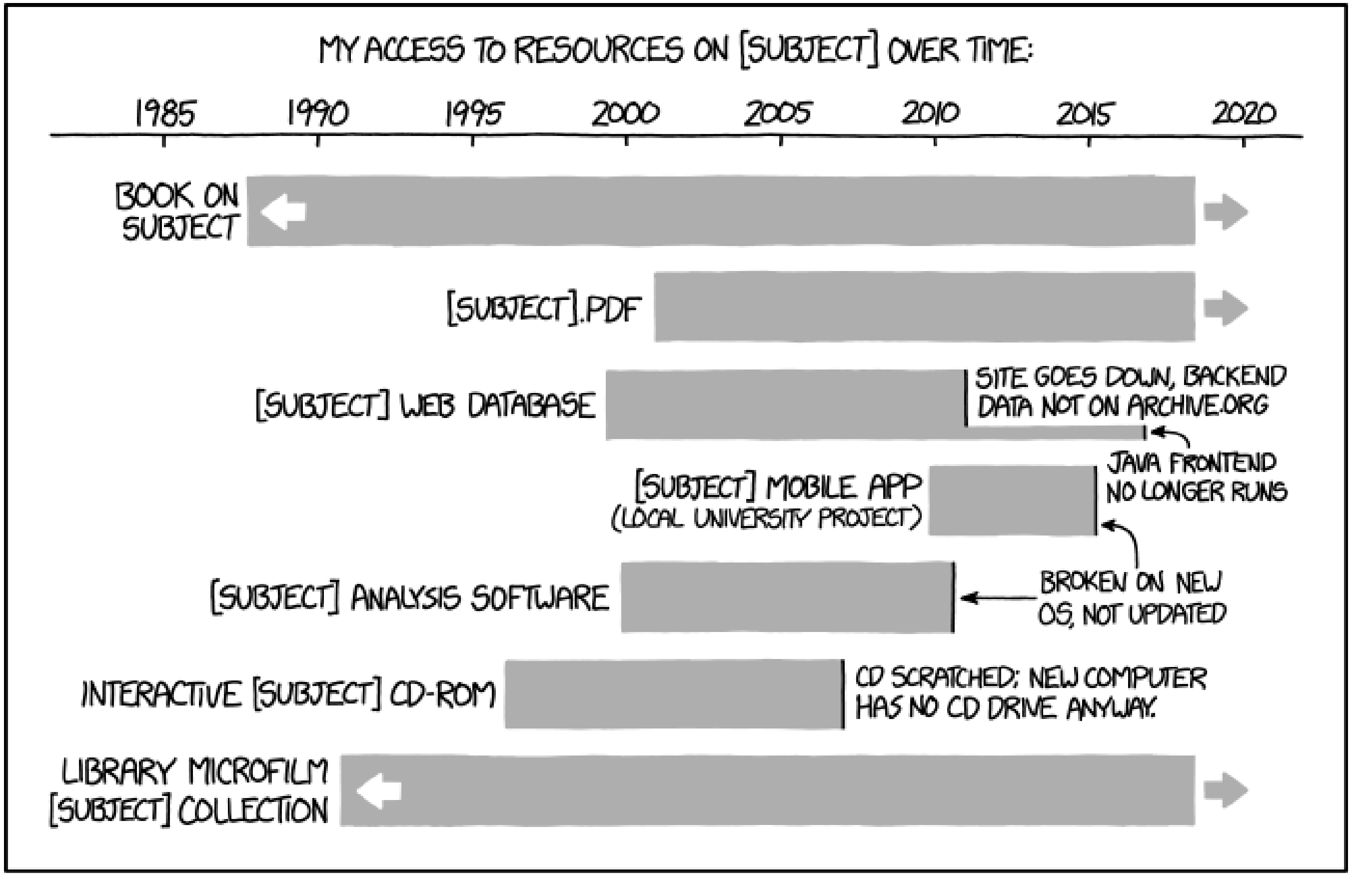
Parker DeWilde

Emily Furst

Sarah House

Waylon Huang

Vinny Palaniappan



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

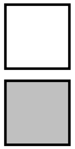
http://xkcd.com/1909/

Administrative

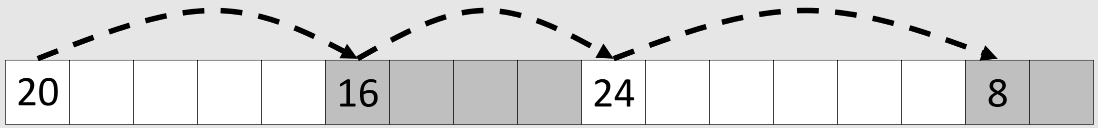
- Homework 5 due Wednesday (3/7)
- Lab 5 due Saturday (3/10)
  - Recommended that you watch the Lab 5 helper videos

- **Final Exam:** Wed, March 14 @ 2:30pm in KNE 110
  - Mult. Choice, Short Answer, True/False - everything
  - **VM** – see practice questions at end of VM II lecture
  - **Caching**
  - **Arrays and Structs**
  - Processes
  - Dynamic Memory Allocation

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

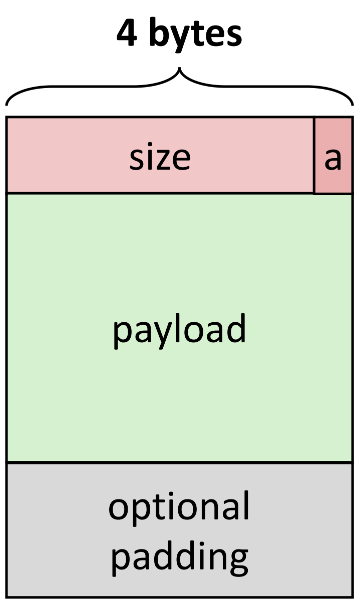


Implicit Free Lists

- For each block we need: `size`, **is-allocated?**
  - Could store using two boxes, but wasteful
- Standard trick
  - If blocks are aligned, some low-order bits of `size` are always 0
  - Use lowest bit as an allocated/free flag (fine as long as aligning to K>1)
  - When reading `size`, must remember to mask out this bit!

e.g. with 8-byte alignment, possible values for size:
00001000 = 8 bytes
00010000 = 16 bytes
00011000 = 24 bytes
. . .

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)

If `x` is first box (header):

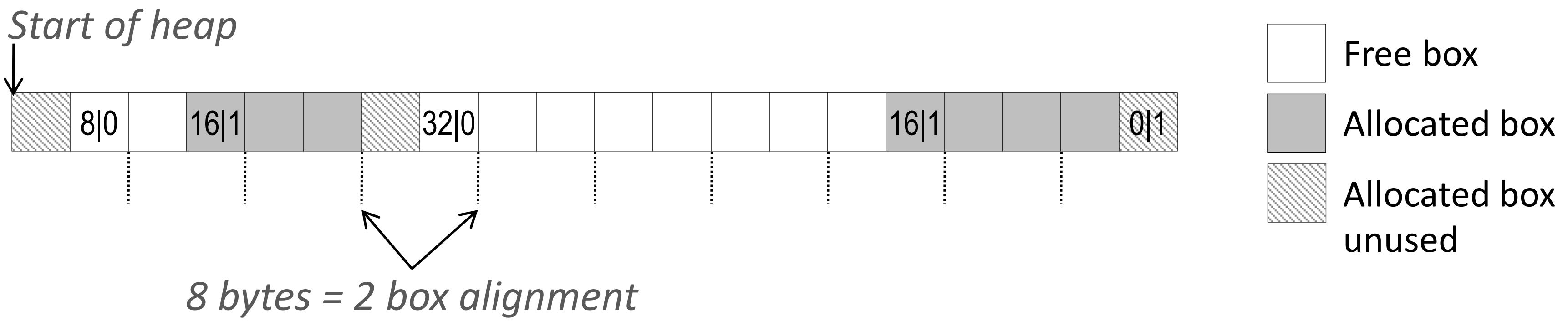
```
x = size | a;

a = x & 1;

size = x & ~1;
```

Implicit Free List Example

- Each block begins with header (size in bytes and allocated bit)
- Sequence of blocks in heap (`size|allocated`):
  8|0, 16|1, 32|0, 16|1



- 8-byte alignment for *payload*
  - May require initial padding (internal fragmentation)
  - Note `size`: padding is considered part of *previous* block
- Special one-box marker (0|1) marks end of list
  - Zero `size` is distinguishable from all other blocks

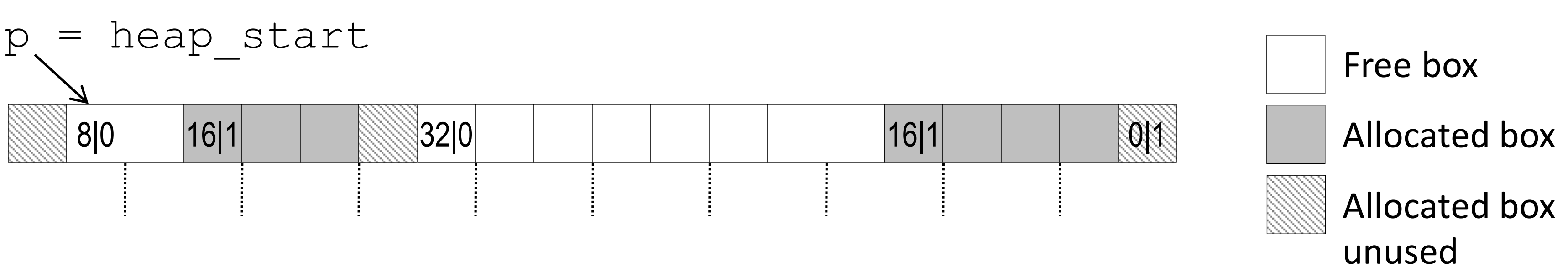
Implicit List: Finding a Free Block

> (*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
  p = p + (*p & -2);       // go to next block (UNSCALED +)
}                          // p points to selected block or end
```

  - Can take time linear in total number of blocks
  - 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

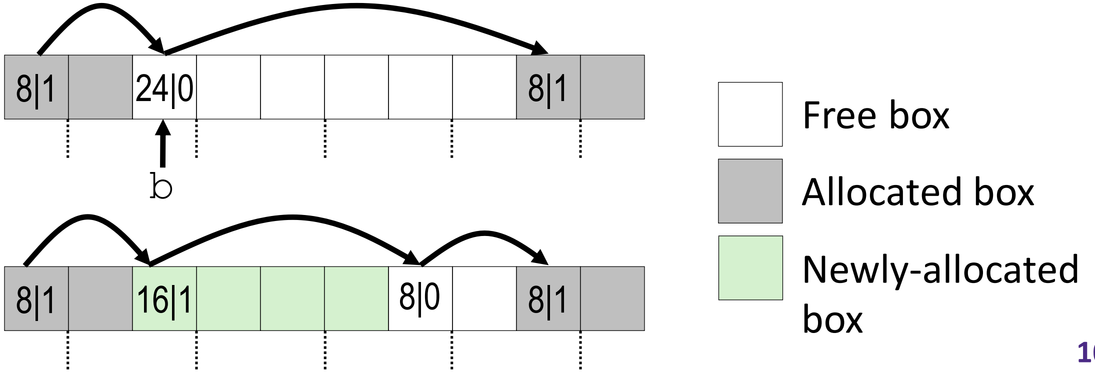
Implicit List: Allocating in a Free Block

- 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

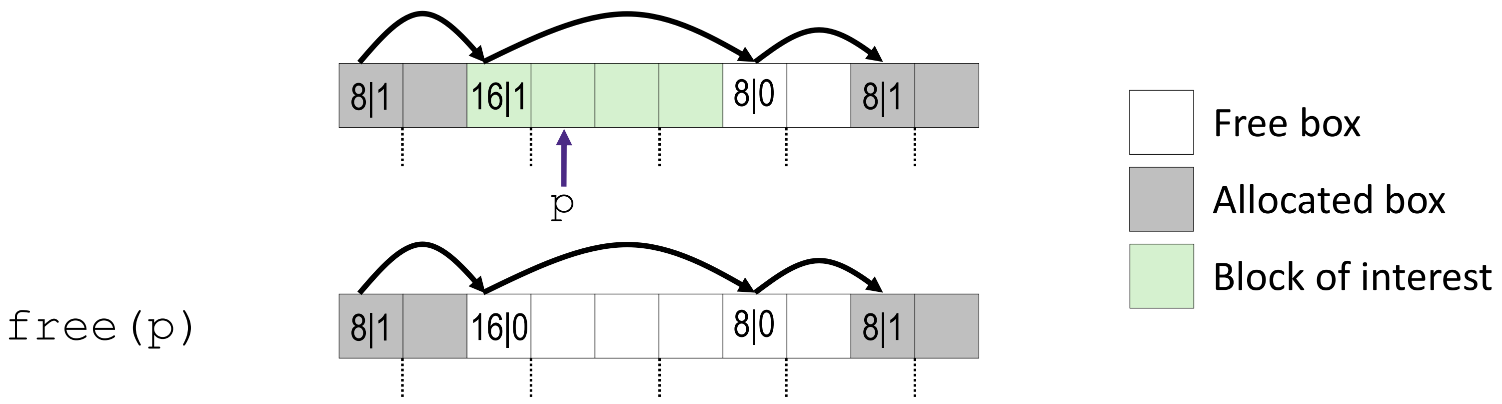
```
void split(ptr b, int bytes) {                 // bytes = desired block size
  int newsize = ((bytes+7) >> 3) << 3;         // round up to multiple of 8
  int oldsize = *b;                            // why not mask out low bit?
  *b = newsize;                                // initially unallocated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

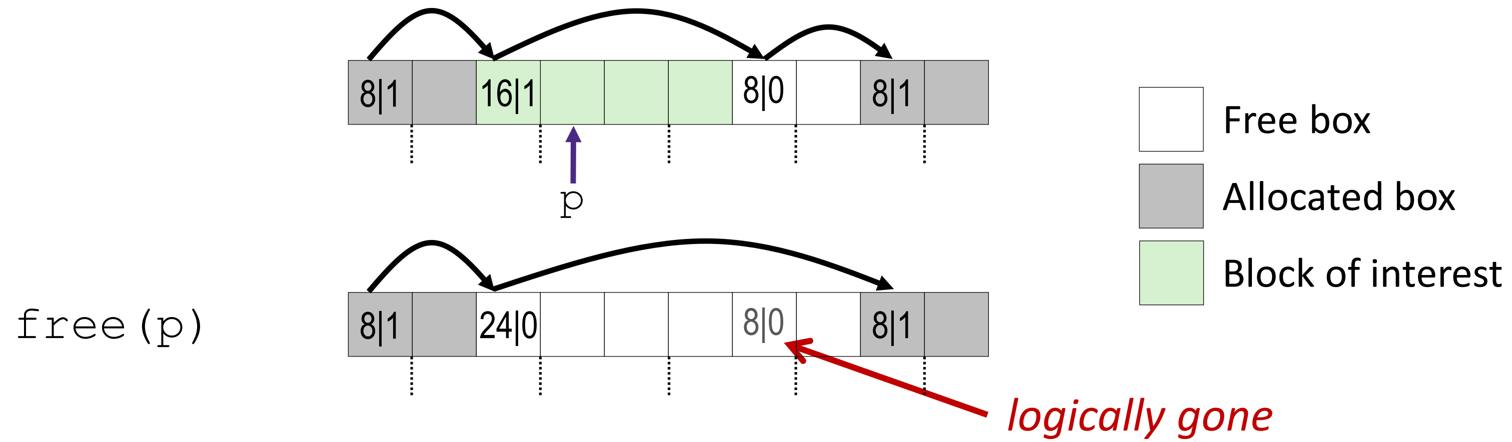
- Simplest implementation just clears "allocated" flag
  - `void free(ptr p) {*(p-BOX) &= -2;}`
  - But can lead to "false fragmentation"



`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

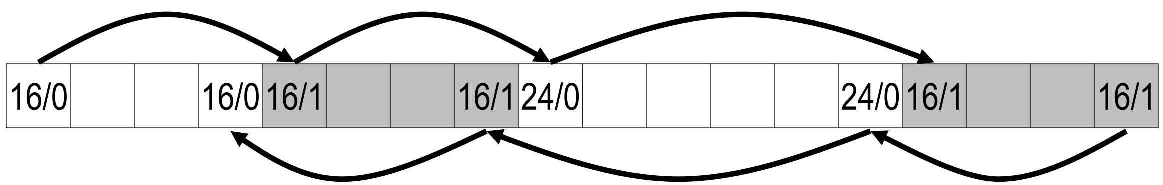



```
void free(ptr p) {            // p points to payload
    ptr b = p - BOX;          // b points to block header
    *b &= -2;                 // clear allocated bit
    ptr next = b + *b;        // find next block (UNSCALED +)
    if ((*next & 1) == 0)     // if next block is not allocated,
     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

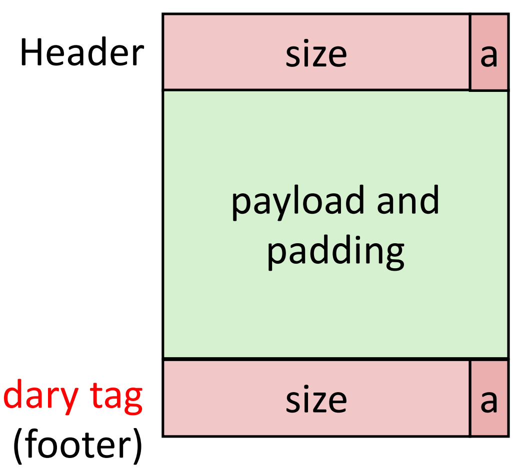
- How do we coalesce with the *previous* block?

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:

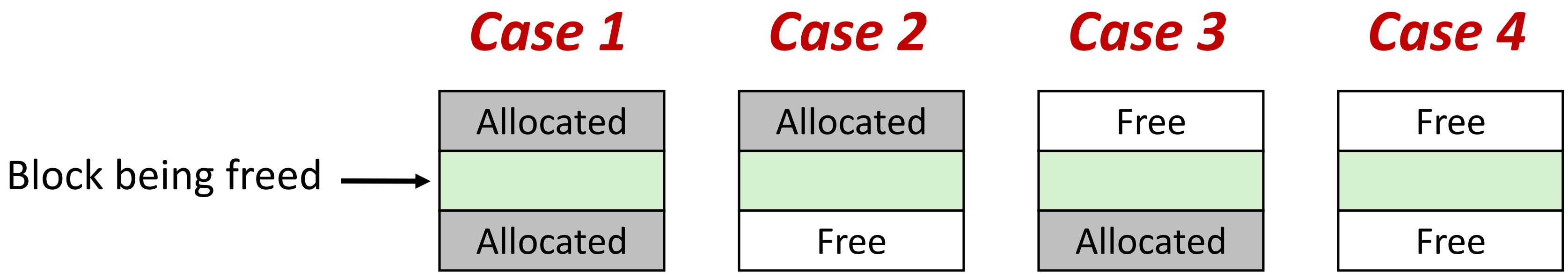


a = 1: allocated block
a = 0: free block

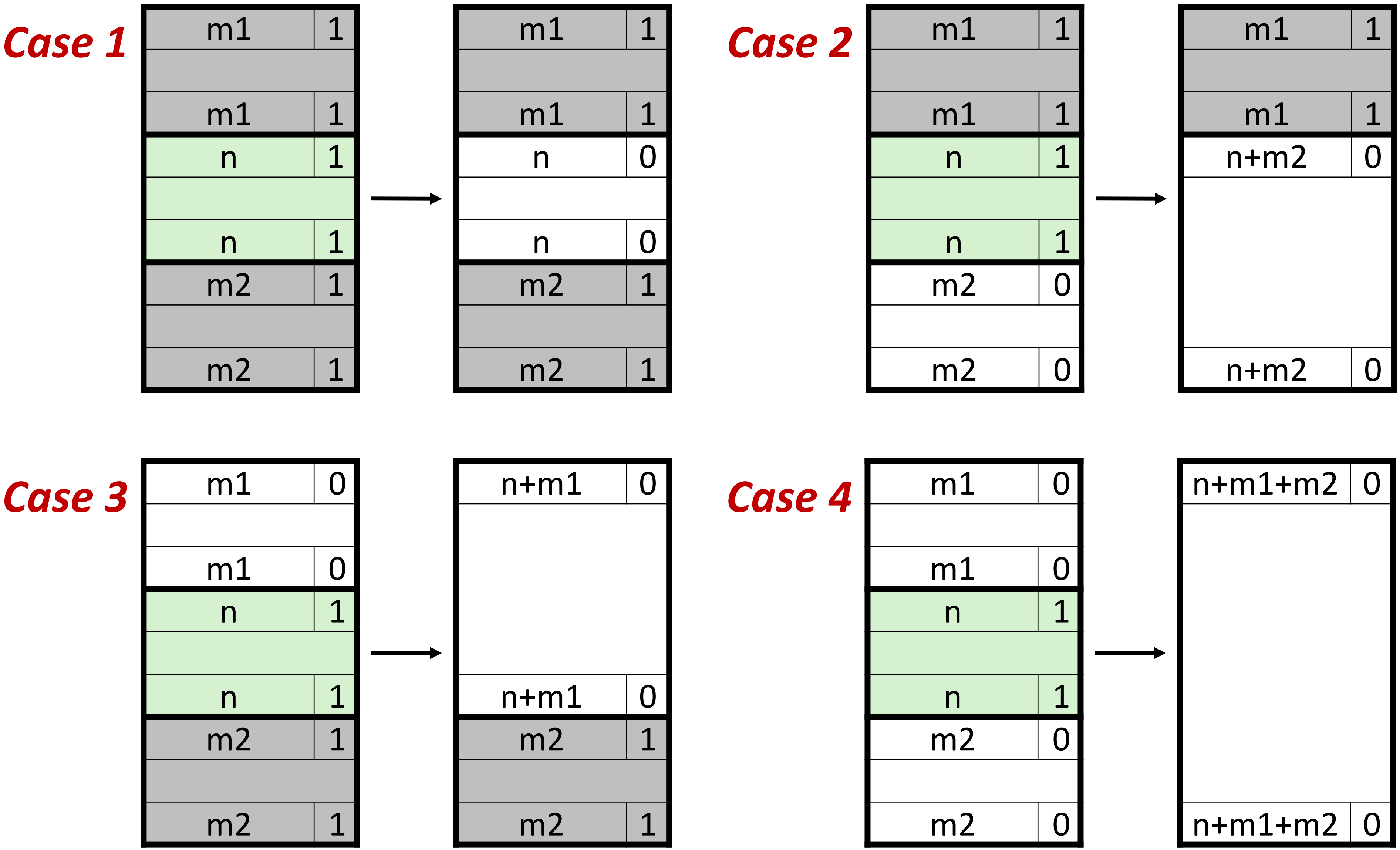
size: block size (in bytes)

payload: application data (allocated blocks only)

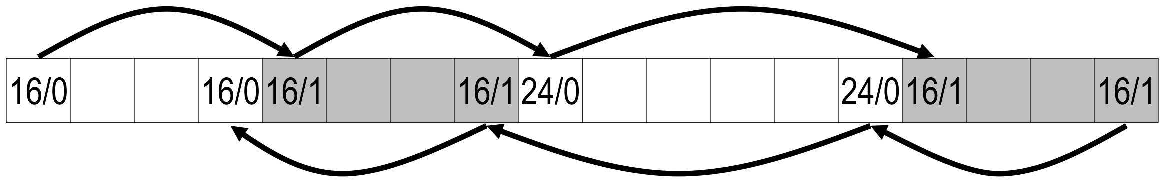
Constant Time Coalescing



Constant Time Coalescing

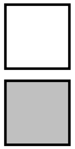


Implicit Free List Review Questions

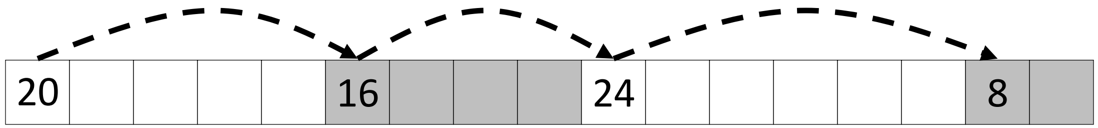


- What is the block header? What do we store and how?
- What are boundary tags and why do we need them?
- When we coalesce free blocks, how many neighboring blocks do we need to check on either side? Why is this?
- If I want to check the size of the n-th block forward from the current block, how many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

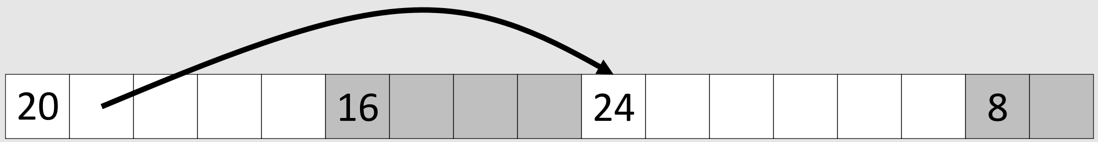
Keeping Track of Free Blocks


= 4-byte box (free)
= 4-byte box (allocated)

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



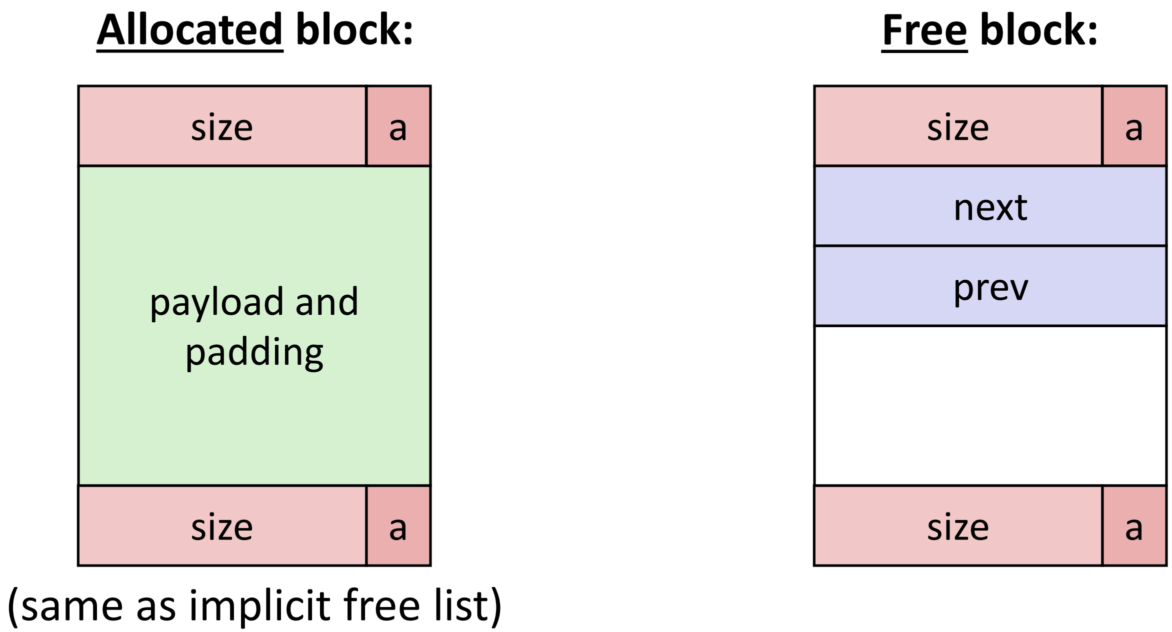
2) *Explicit free list* among only the free blocks, using pointers



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

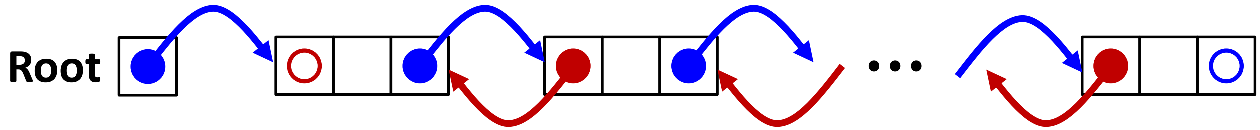
Explicit Free Lists



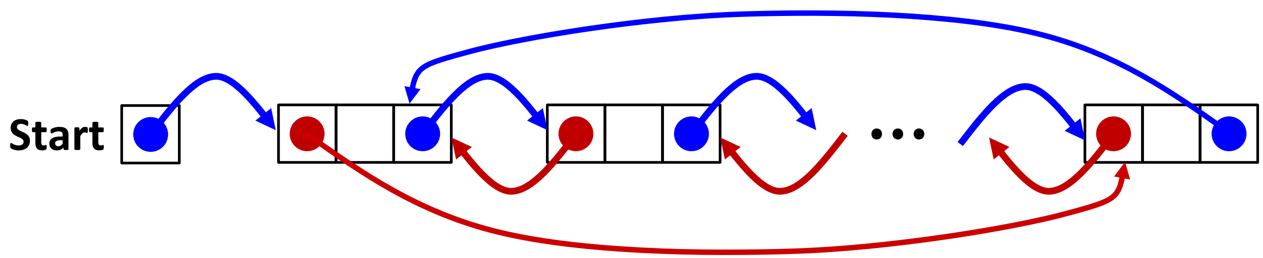
- Use list(s) of *free* blocks, rather than implicit list of *all* blocks
  - The "next" free block could be anywhere in the heap
    - So we need to store next/previous pointers, not just sizes
  - Since we only track free blocks, so we can use "payload" for pointers
  - Still need boundary tags (header/footer)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

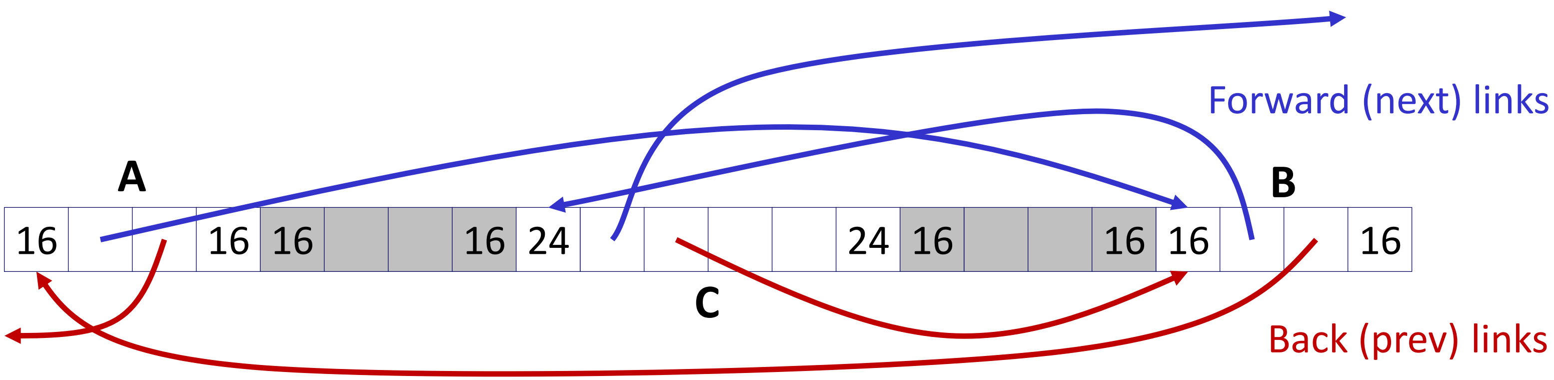


- **Circular**
  - Still have pointer to tell you which node to start with
  - No NULL pointers (term condition is back at starting point)
  - Good for next-fit, best-fit

Explicit Free Lists

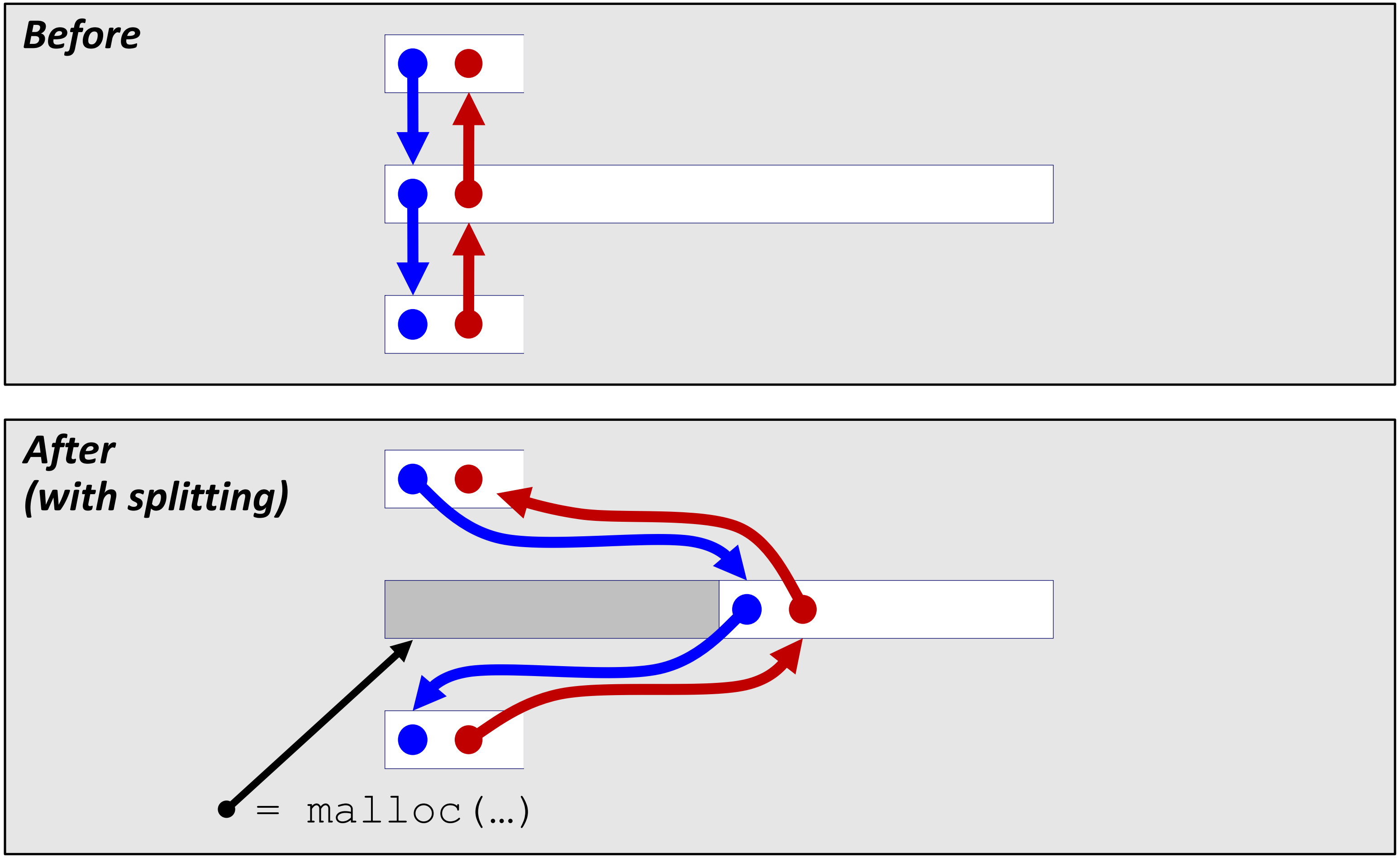
- **Logically:** doubly-linked list

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



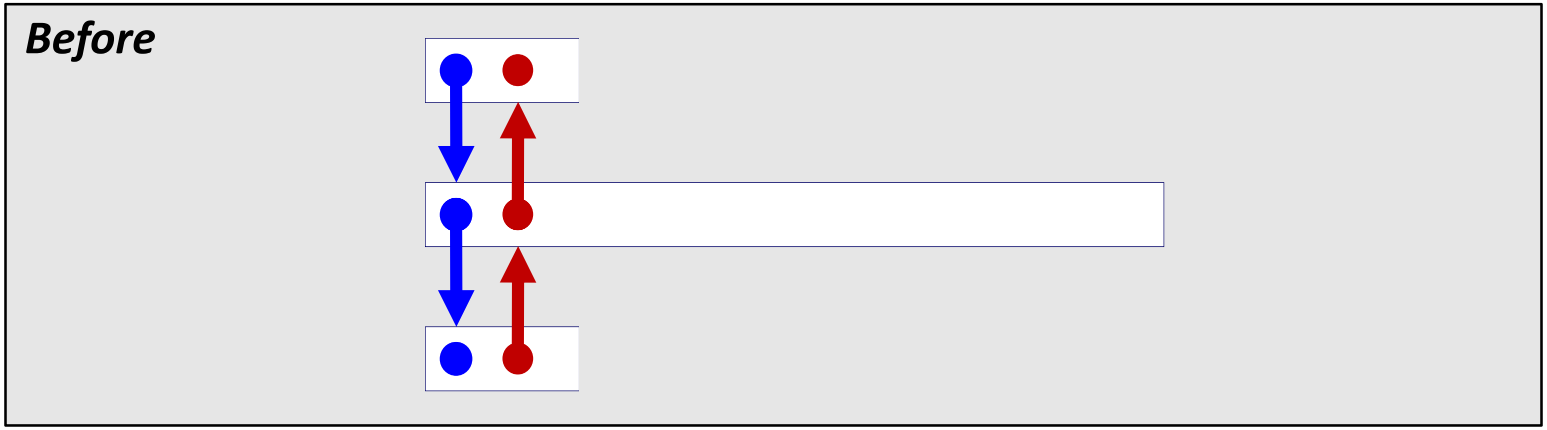
Allocating From Explicit Free Lists

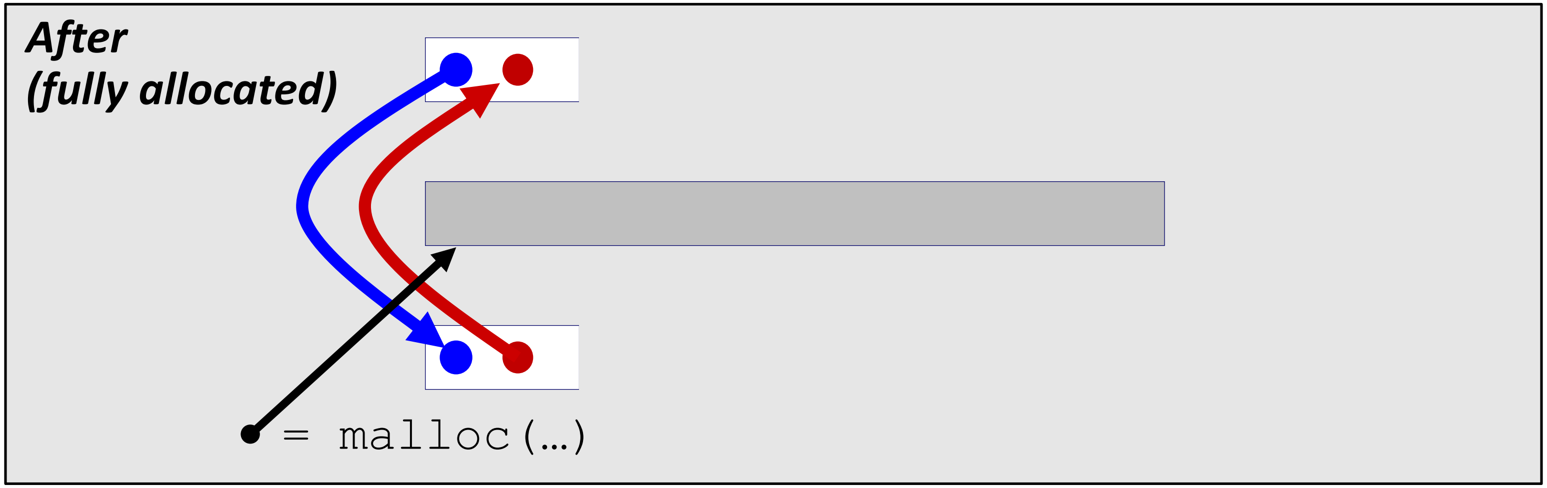
Note: These diagrams are not very specific about *where inside a block* a pointer points. In reality we would always point to one place (e.g. start/header of a block).



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).



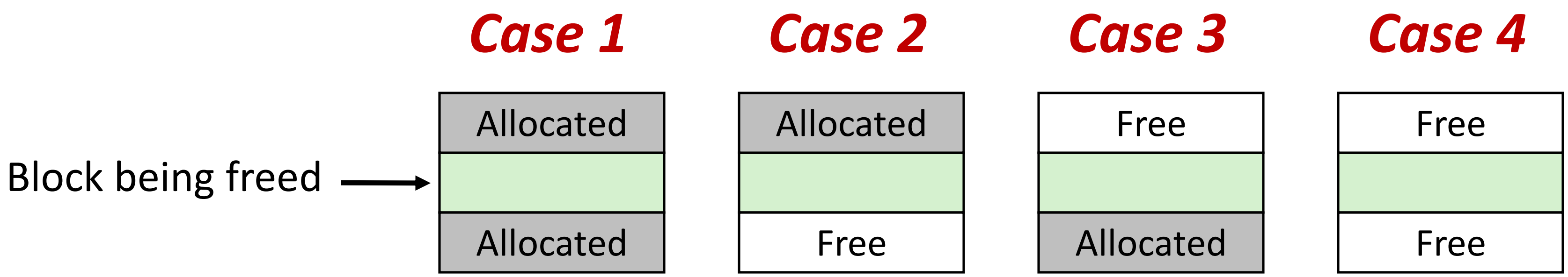


Freeing With Explicit Free Lists

- *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:

      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

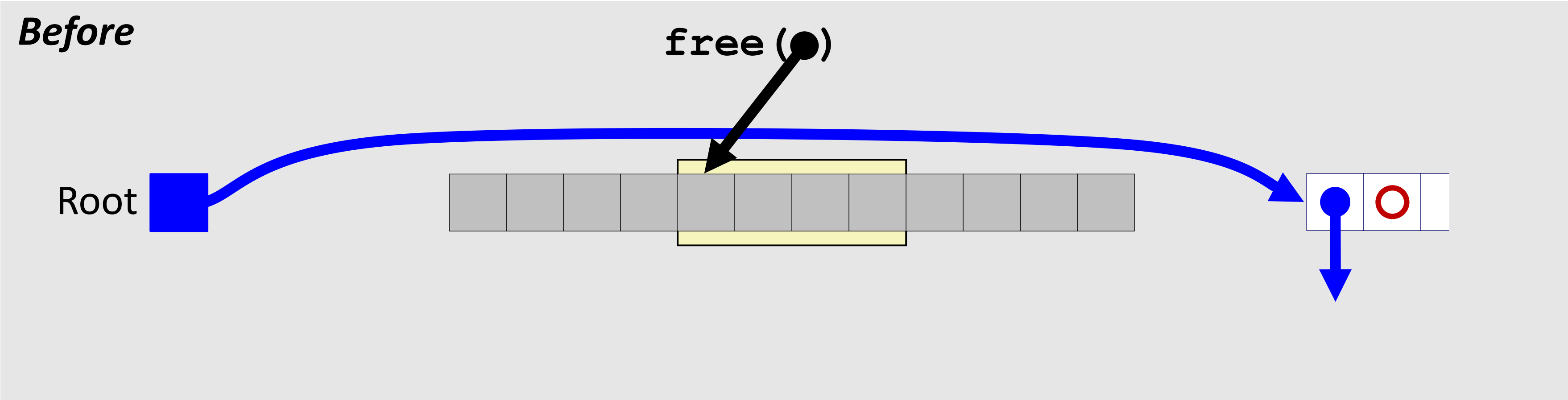


- Neighboring free blocks are *already part of the free list*
  1) Remove old block from free list
  2) Create new, larger coalesced block
  3) Add new block to free list (insertion policy)

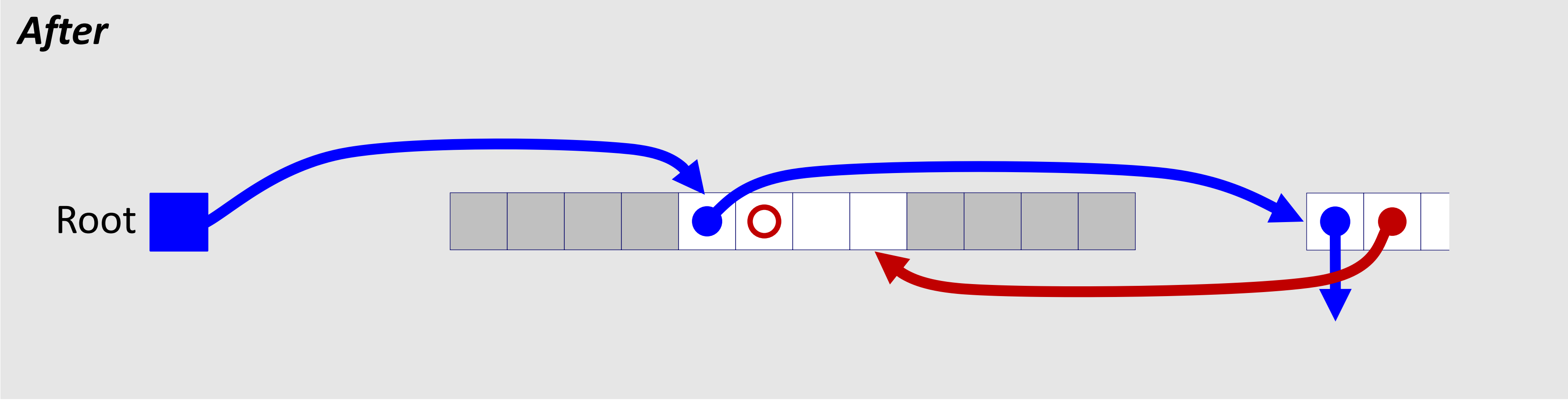
- How do we tell if a neighboring block if free?

Freeing with LIFO Policy (Case 1)

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

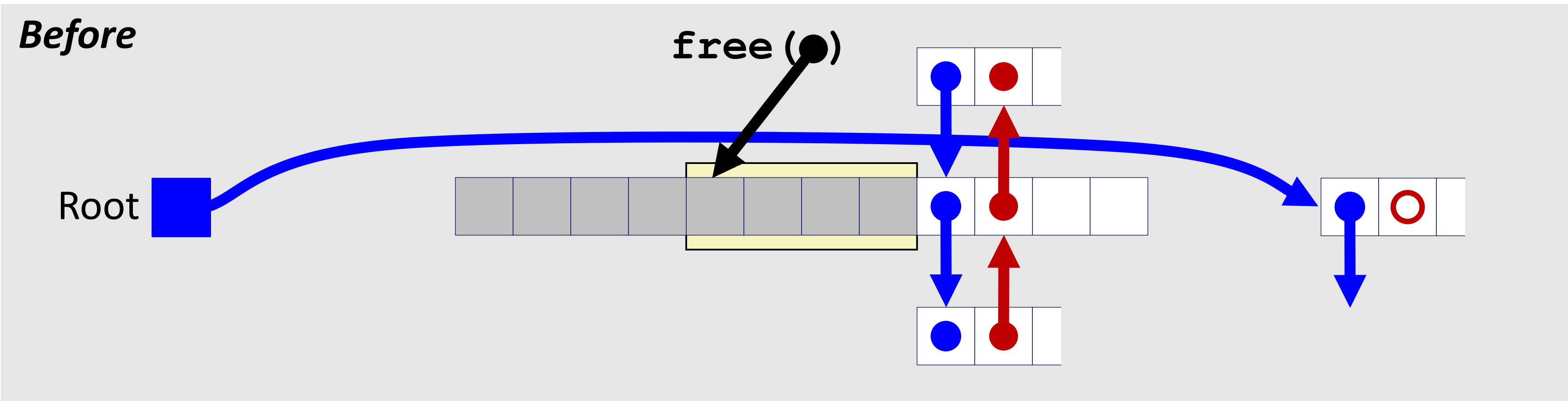


- Insert the freed block at the root of the list

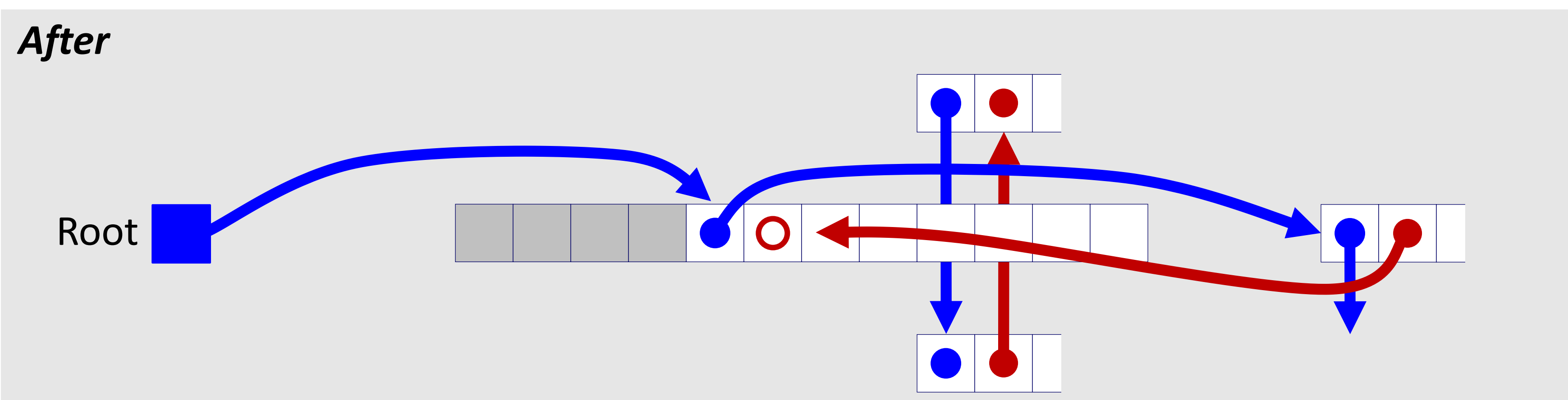


Freeing with LIFO Policy (Case 2)

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

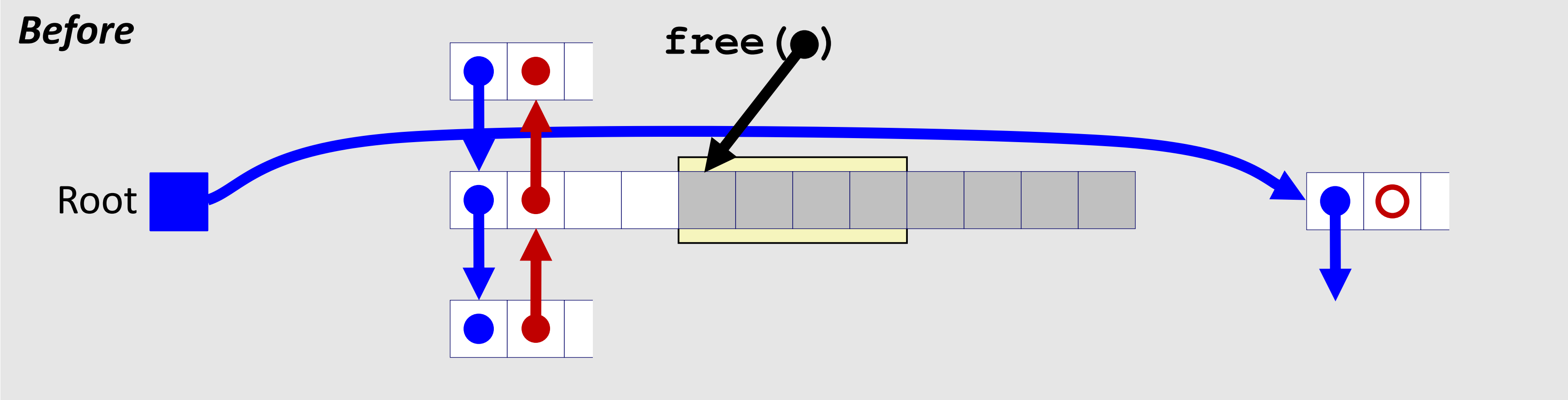


- Splice *successor* block out of list, coalesce both memory blocks, and insert the new block at the root of the list

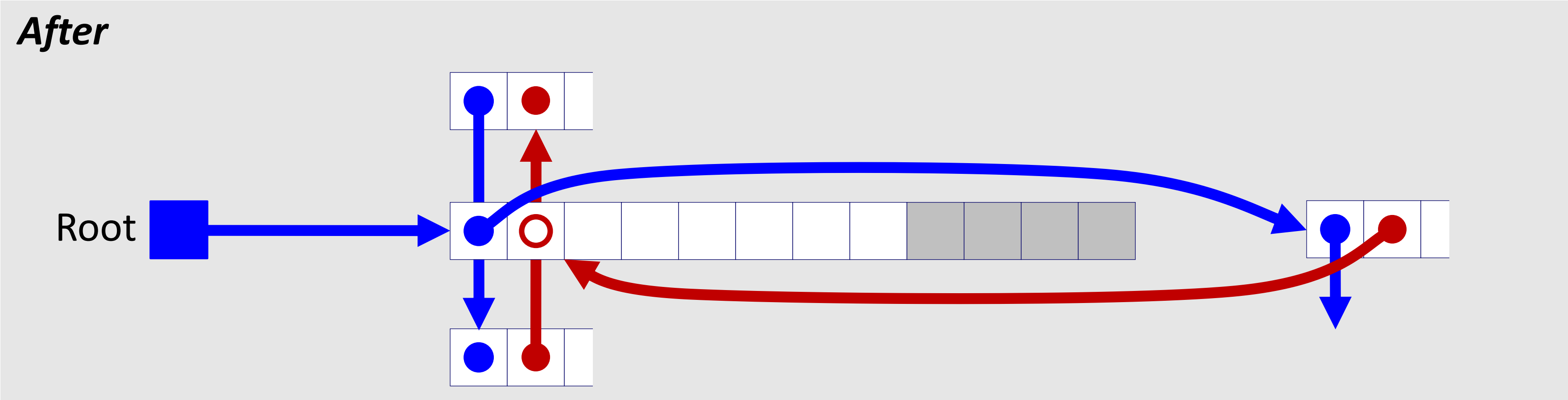


Freeing with LIFO Policy (Case 3)

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

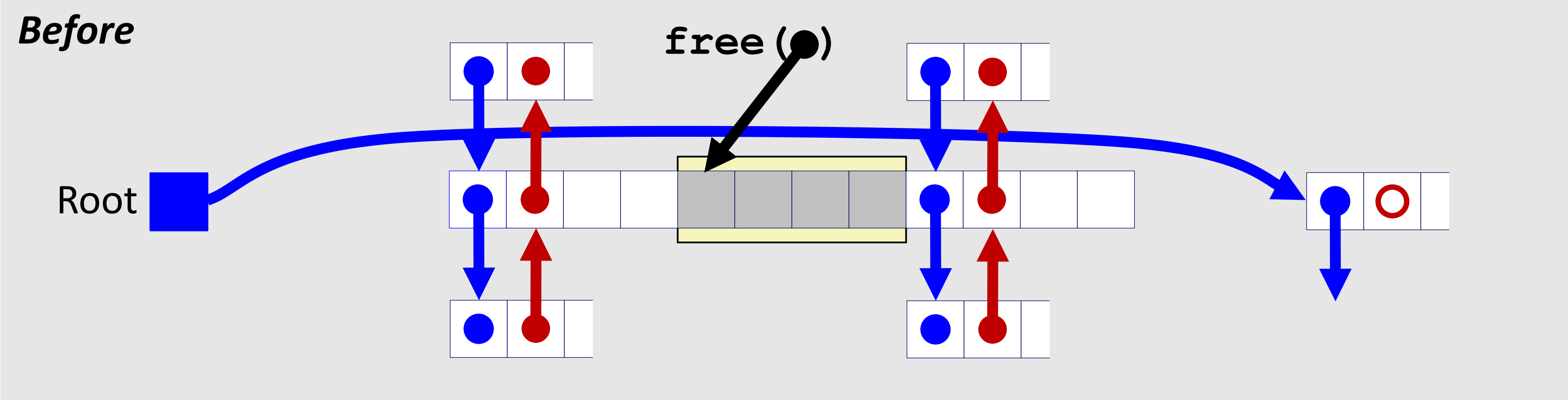


- Splice *predecessor* block out of list, coalesce both memory blocks, and insert the new block at the root of the list

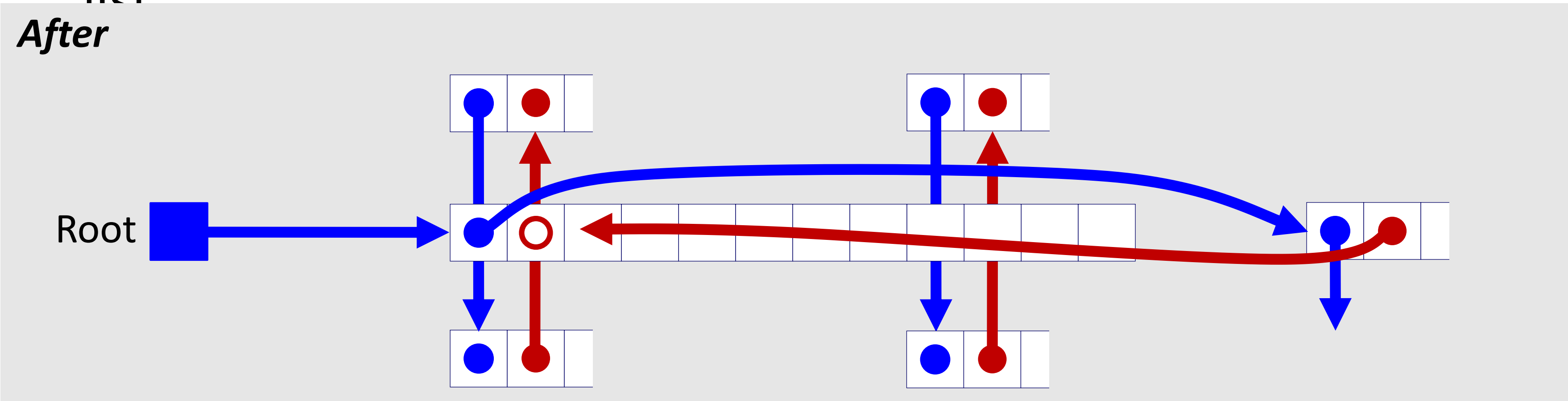


Freeing with LIFO Policy (Case 4)

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



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