The Hardware/Software Interface

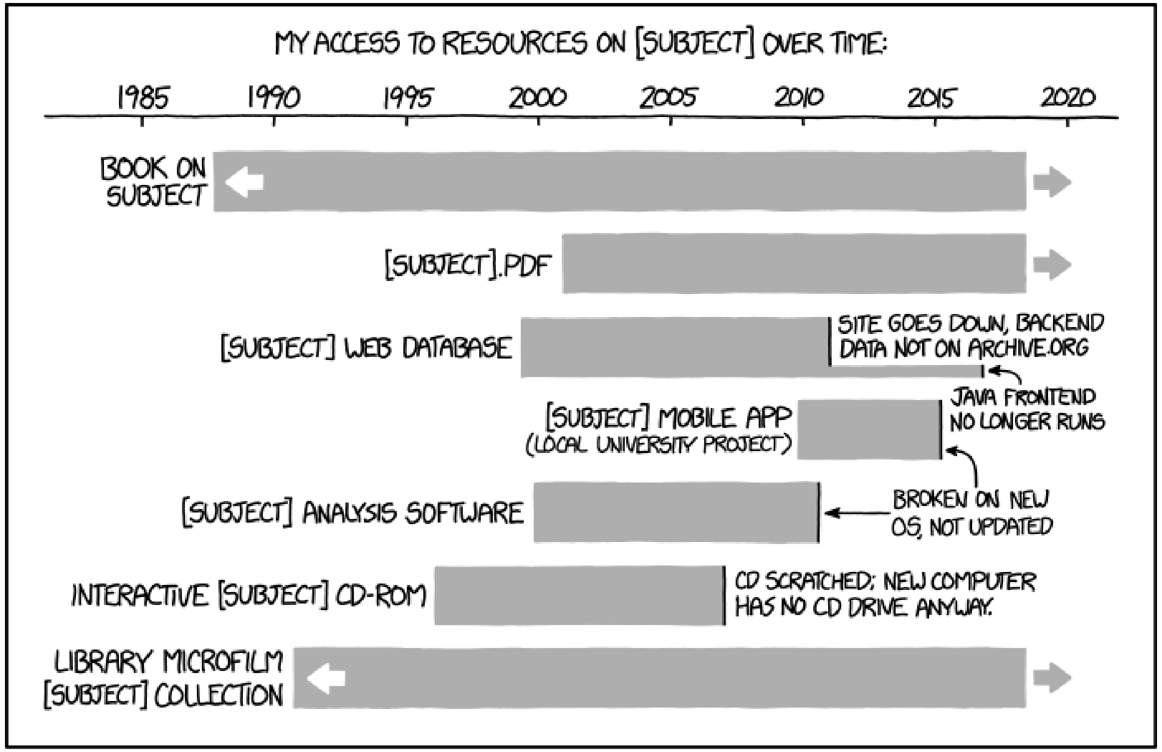
Memory Allocation II

Instructors:

Justin Hsia, Amber Hu

Teaching Assistants:

Anthony Mangus, Divya Ramu, Grace Zhou, Jessie Sun, Jiuyang Lyu, Kanishka Singh, Kurt Gu, Liander Rainbolt, Mendel Carroll, Ming Yan, Naama Amiel, Pollux Chen, Rose Maresh, Soham Bhosale, Violet Monserate



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

http://xkcd.com/1909/

Relevant Course Information

- HW19 due tonight, HW20 due Monday (11/17)

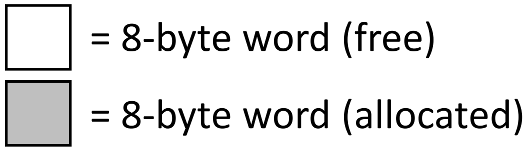
- Lab 4 due next Friday (11/21), closes Monday (11/24)
  - The Cache Simulator is especially helpful for visualizing and debugging Part 1
  - Cache images and more abstract thinking more helpful for Part 2
    - Trace files can be used for nitty-gritty debugging
  - Make sure you run the variable check for Part 2 to check for rules compliance

- Lab 5 (Memory Allocator) will be released on Monday (11/17), due last Thursday (12/4)
  - Implement part of `malloc` and `free` – maintain heap blocks and free list
  - The most C programming you will do in this class (still not a ton)

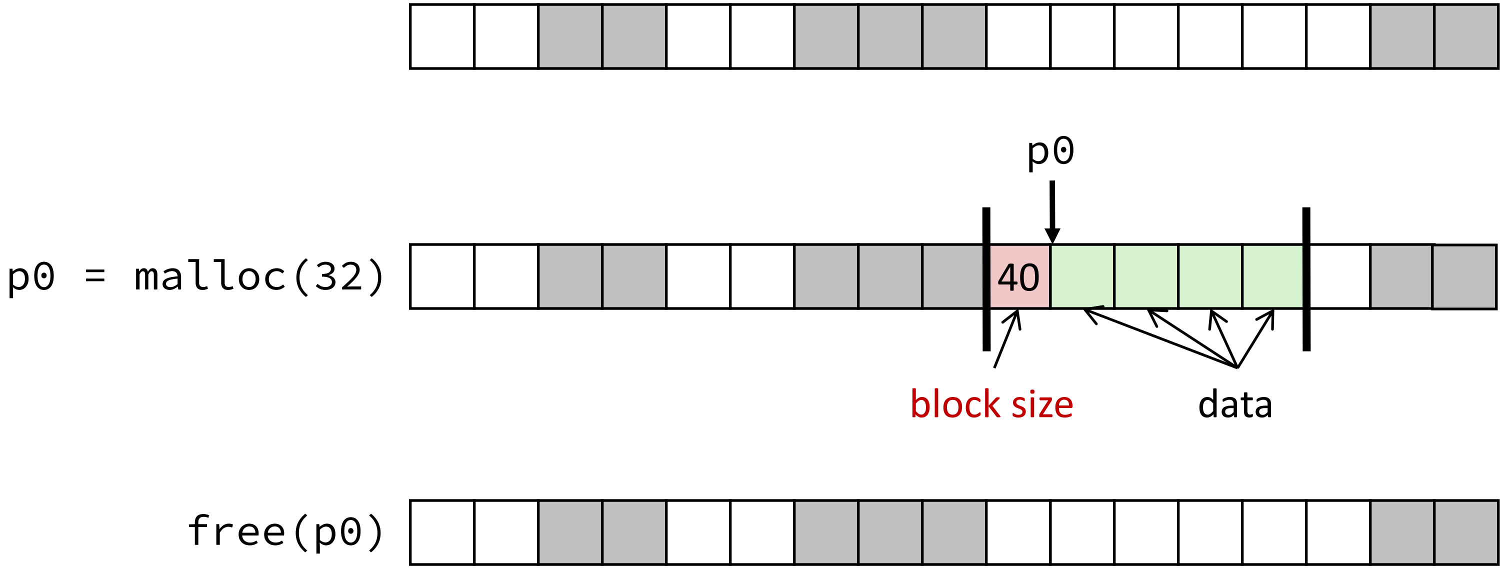
Lecture Outline (0/3)

- **Heap Implementation Basics (from Lecture 20)**
- Allocation and Splitting
- Deallocation and Coalescing
- Explicit Free Lists

Knowing How Much to Free



- Standard method: keep the length of a block in the word preceding the data called the ***header field*** or ***header***
  - Requires an extra word for every allocated block

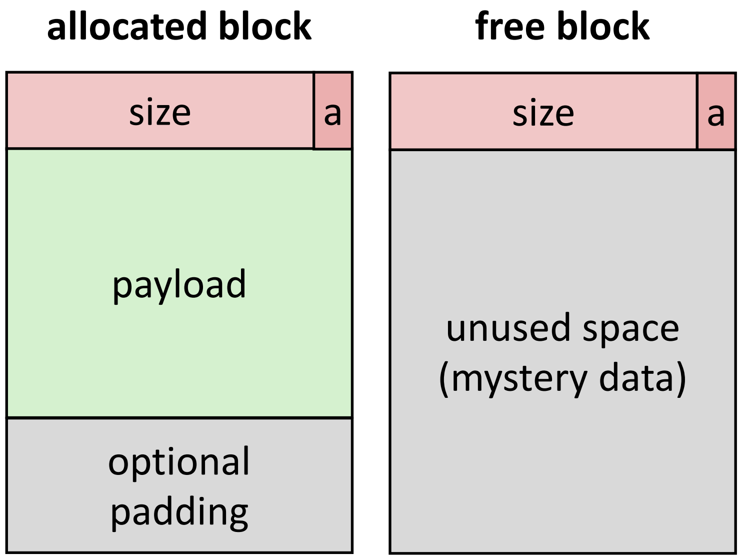


Header Information

- For each block we need: **`size`**, **is-allocated?**
- Standard trick: Use lowest bit as an allocated/free flag
  - If blocks are aligned (K>1), some low-order bits of `size` are always 0:

    e.g., with 8-byte alignment:
    00001000 = 8 bytes
    00010000 = 16 bytes
    00011000 = 24 bytes
  - When reading `size`, must remember to mask out this bit!

Format of heap blocks:



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

size: block size (in bytes)

payload: application data (allocated blocks only)

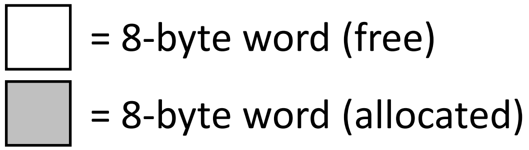
If x is first word (header):

```
x = size | a;

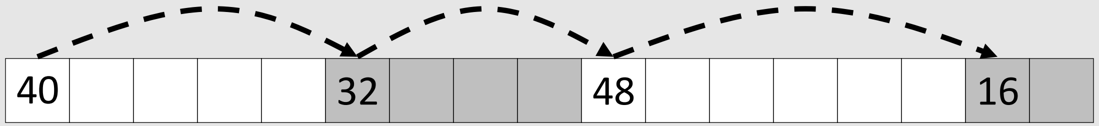
a = x & 1;

size = x & ~1;
```

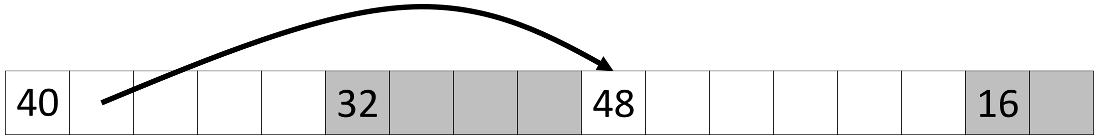
Keeping Track of Free Blocks



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



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



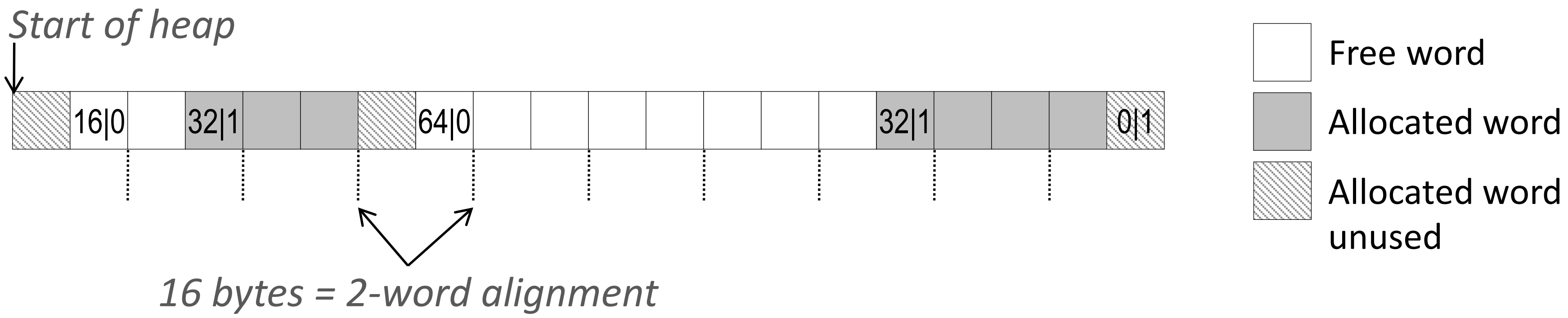
3) *Segregated free list*
   - Different free lists for different size "classes"

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

Implicit Free List Example

= 8-byte word

- Each block begins with header (size in bytes and is-allocated? bit)
- Heap blocks (`size`|is-allocated?): 16|0, 32|1, 64|0, 32|1



- 16-byte alignment for (1) *heap block size* and (2) *payload address*
  - Padding for size is considered part of *previous* heap block (internal fragmentation)
  - May require initial padding at start of heap
- Special one-word marker (0|1) marks end of list
  - Zero `size` is distinguishable from all other blocks

Lecture Outline (1/3)

- **Allocation and Splitting**
- Deallocation and Coalescing
- Explicit Free Lists

Fulfilling an Allocation Request (Review)

1) Compute the necessary block size
2) Search for a suitable free block using the allocator's *allocation strategy*
   - If found, continue
   - If not found, return `NULL`
3) Compare the necessary block size against the size of the chosen block
   - If equal, allocate the block
   - If not, *split* off the excess into a new free block before allocating the block
4) Return the address of the beginning of the payload

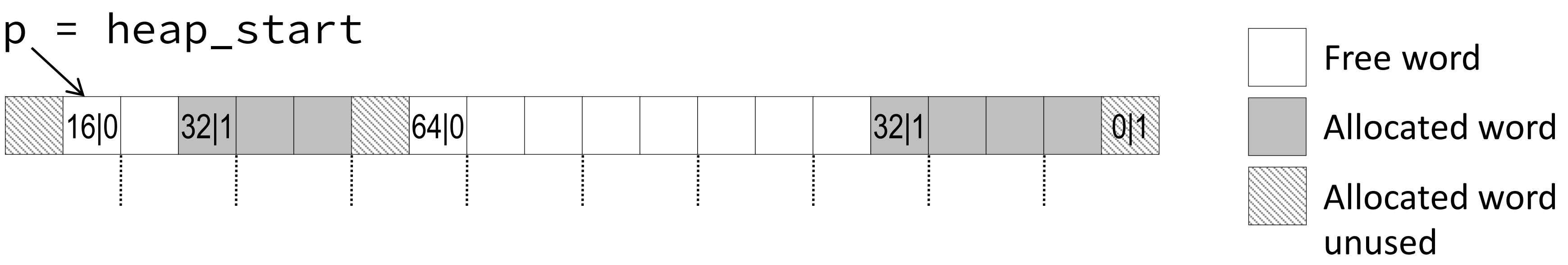
Necessary Block Size (Review)

- For `malloc(`n`)`:
  - Payload size: n
  - Metadata: (for now) size of the header (h)
  - Padding: add to $(n + h)$ to reach nearest multiple of alignment

- Example code for alignment requirement of 16:
  - ```
    size = ((n+h+15) >> 4) << 4;   // round up to multiple of 16
    ```

- ***Minimum block size***: Smallest possible block that we can allocate
  - Determined by max requirements for allocated and free blocks in implementation
  - Another way to think about this is what size is allocated on `malloc(1)`
    ```

Allocation Strategies: First Fit (Review)



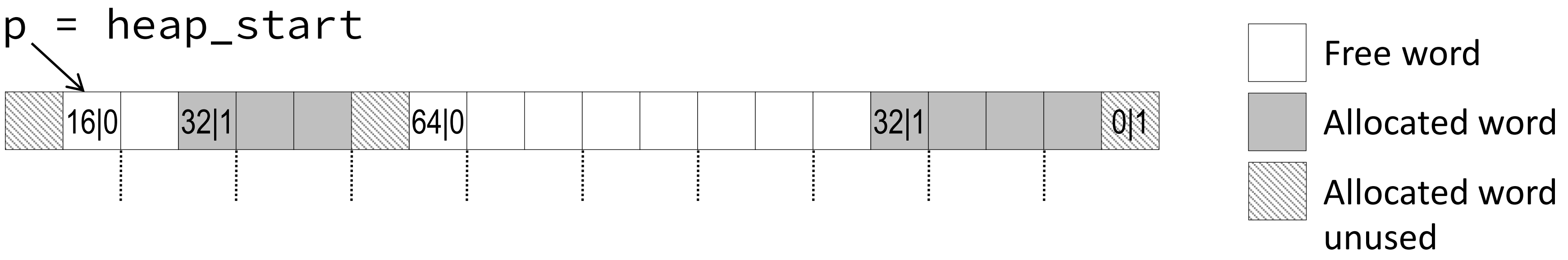
- ***First fit:*** Search the free list from the beginning & choose first free block that fits
  - Can take time linear in total number of blocks
  - In practice, can cause "splinters" at beginning of list
  - Pseudocode for implicit free list:

```
p = heap_start;
while ((p < end) &&         // not past end
       ((*p & 1) ||         // already allocated
        (*p <= len))) {     // too small
  p = p + (*p & ~1);        // go to next block (UNSCALED +)
}                           // p points to selected block or end
```

Header operations:

`*p` fetch block header x
`*p & 1` extracts `is-alloc?`
`*p & ~1` extracts `size`

Allocation Strategies: Next fit, Best fit (Review)



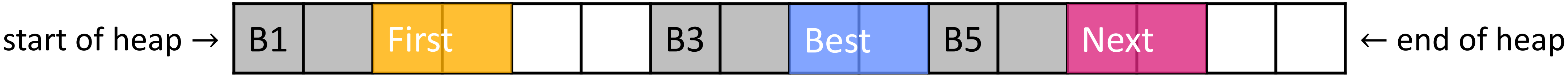
- ***Next fit:*** Search the free list starting where previous search finished (wrap around at end of heap) & choose first free block that fits
  - Should often be faster than first-fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse
- ***Best fit:*** Search the entire free list & choose the best free block (*i.e.*, large enough with fewest bytes left over)
  - Keeps fragments small – usually helps fragmentation
  - Usually worse throughput

Allocation Strategy Example (Review)

= 8-byte word (free)

= 8-byte word (allocated)

- The current state of the heap is shown below:
  - Headers are 8 bytes
  - Block B5 was the last fulfilled request (one possible sequence of requests is: allocate B1, allocate B2, allocate B3, allocate B4, allocate B5, free B4, free B2)



- Calling `malloc(8)`:
  - *First fit*: Allocate after B1, since we search from the start of the heap
  - *Next fit*: Allocate after B5, since we search from the end of B5
  - *Best fit*: Allocate after B3, since the available space is the closest fit to the requested payload size

Free Block Conversion

- If selected free block is the same size as the chosen block, convert to allocated
- If selected free block is larger than necessary block size, then need to ***split*** to create a new, smaller leftover free block first
  - Pseudocode:

```
void split(ptr fb, int bsize) {      // bsize = necessary block size
  int oldsize = *fb;                 // why not mask out low bit?
  *fb = bsize | 1;                   // resize and allocate
  if (bsize < oldsize)
    *(fb+bsize) = oldsize - bsize;   // set length in remaining
}                                    //   part of block (UNSCALED +)
```

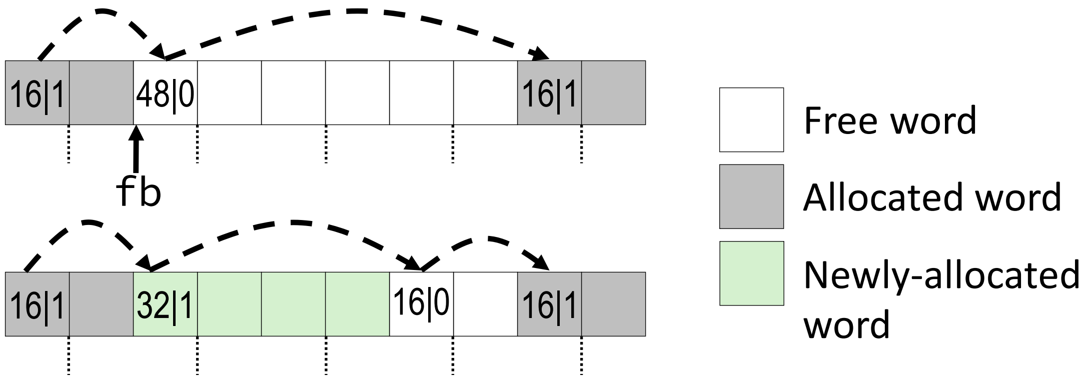
  - Exception: What if leftover space after splitting < minimum block size?

Allocation Request Example (Review)

1) Compute the necessary block size
2) Search for a suitable free block
3) Allocate the chosen block, splitting as necessary
4) Return payload address

```
void* malloc(size_t n) {
  int bsize = ((n+WORD+15)>>4)<<4;
  ptr fb = find(bsize);
  split(fb, bsize);
  return (void*)(fb+WORD);
}
```

malloc(24):

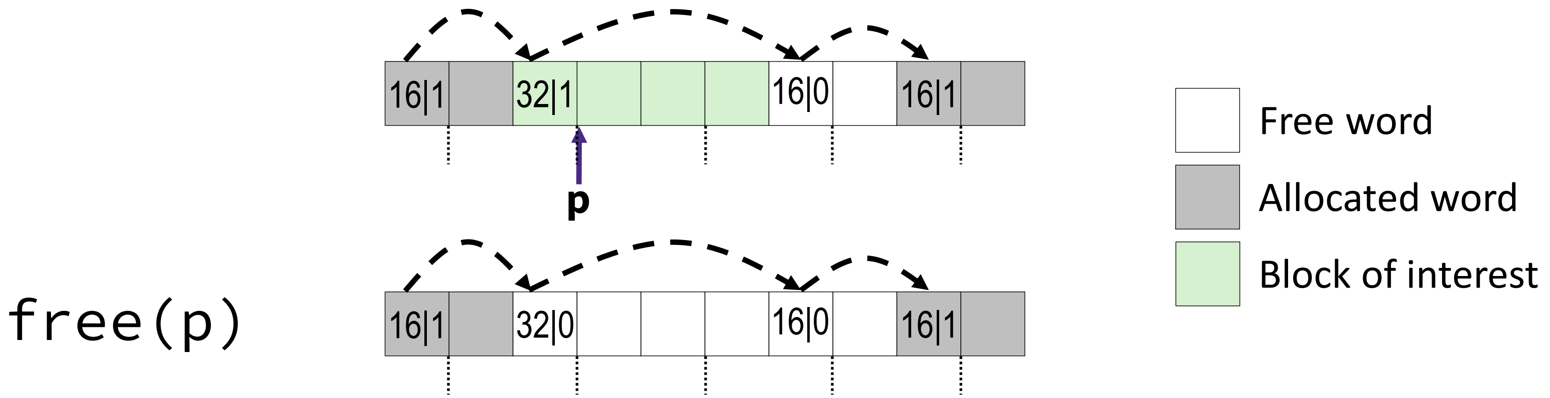


Lecture Outline (2/3)

- Allocation and Splitting
- **Deallocation and Coalescing**
- Explicit Free Lists

Deallocating a Block (Review)

- Simplest implementation just clears "is-allocated?" flag
  - `void free(ptr p) {*(p-WORD) &= ~1;}   // UNSCALED –`
  - But can lead to "false fragmentation":

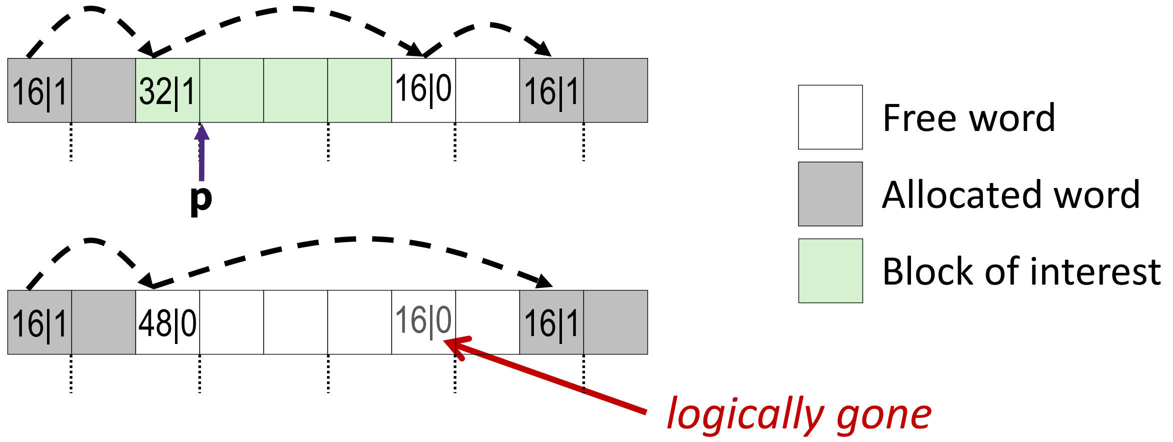


`malloc(40)`

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

Coalescing with Following Block (Review)

- Join (*coalesce*) with following block if also free:

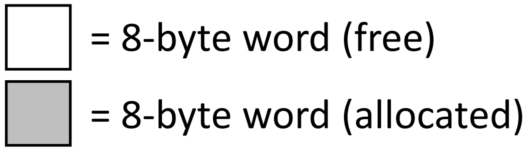


free(p)

  - Pseudocode:

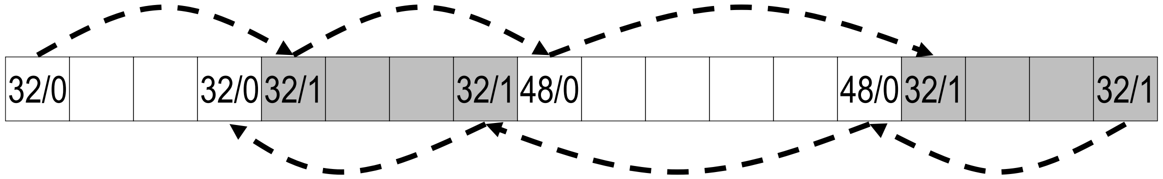
```
void free(ptr p) {        // p points to payload
  ptr hd = p – WORD;      // hd points to header
  *hd &= ~1;              // clear allocated bit
  ptr next = hd + *hd;    // find next block (UNSCALED +)
  if (!(*next & 1))       // if following block is not allocated,
    *hd += *next;         // add its size to this block
}
```

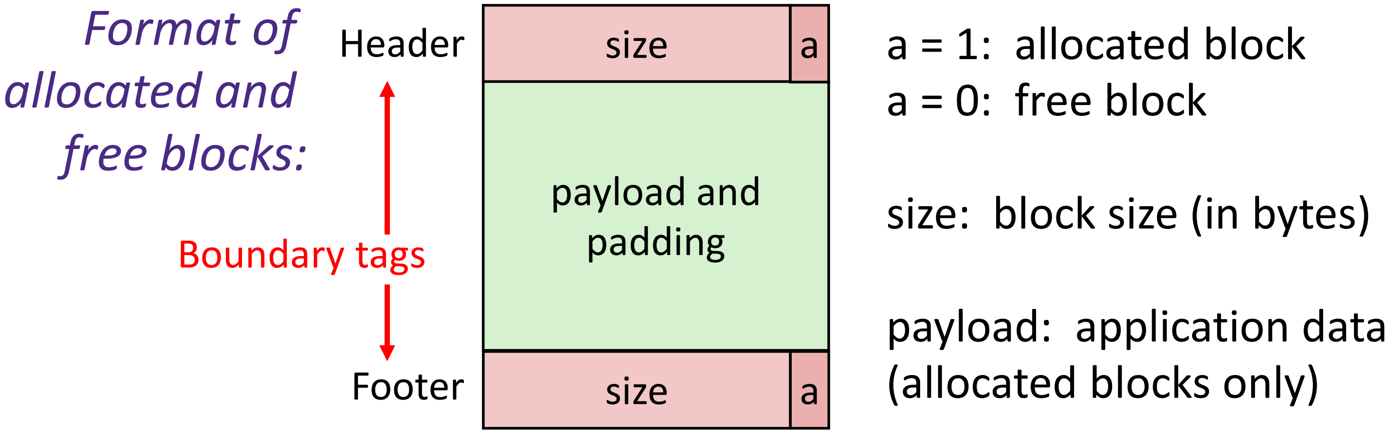
- How do we coalesce with the *preceding* block?



Bidirectional Coalescing (Review)

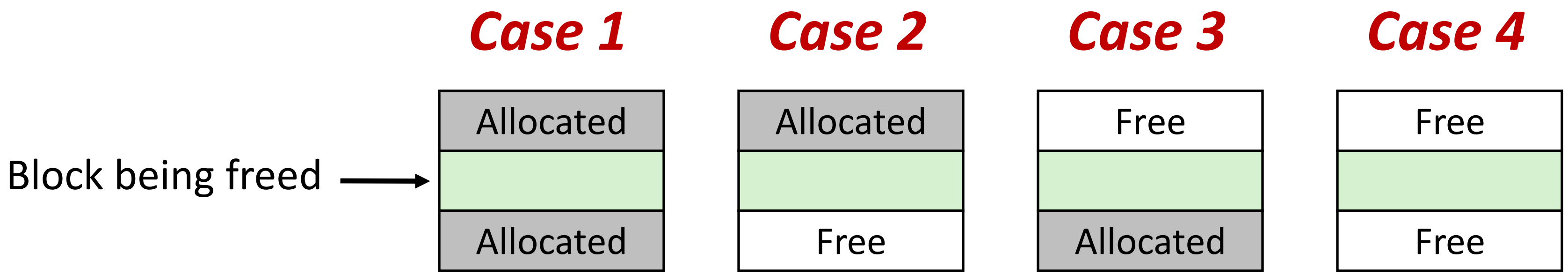
- *Boundary tags* [Knuth73]
  - Replicate header at end of heap blocks
  - Allows us to traverse backwards, but requires extra space



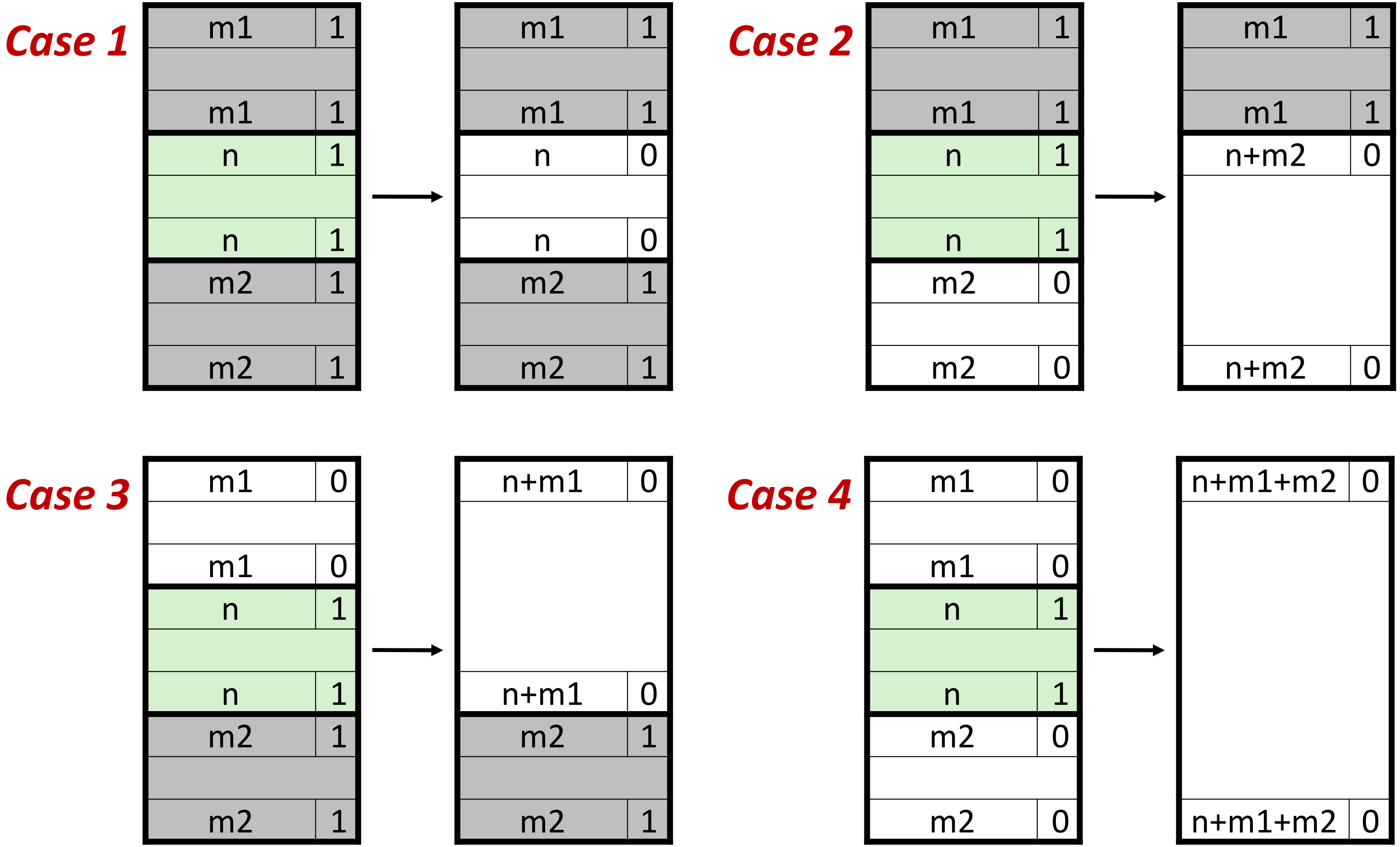


- ⚠️ Everything from "Allocation and Splitting" should be reconsidered with footers!

Constant Time Coalescing: Cases (Review)



Constant Time Coalescing: Updates (Review)



Lecture Outline (3/3)

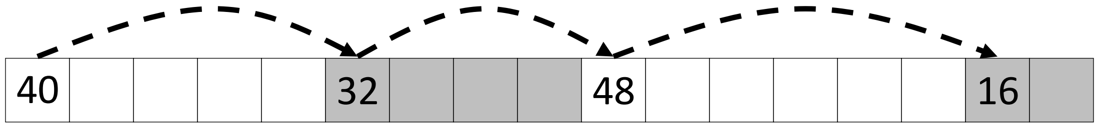
- Allocation and Splitting
- Deallocation and Coalescing
- **Explicit Free Lists**

Keeping Track of Free Blocks

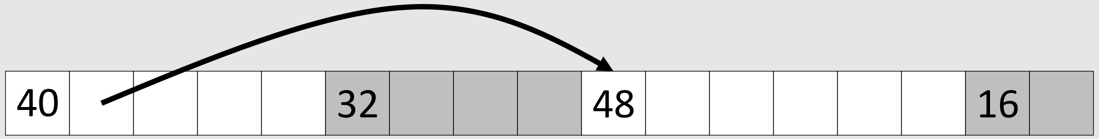
= 8-byte word (free)

= 8-byte word (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

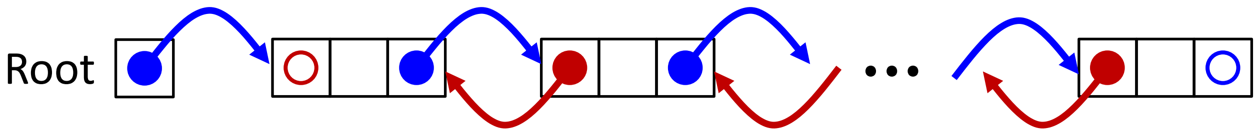


3) *Segregated free list*
   - Different free lists for different size "classes"

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

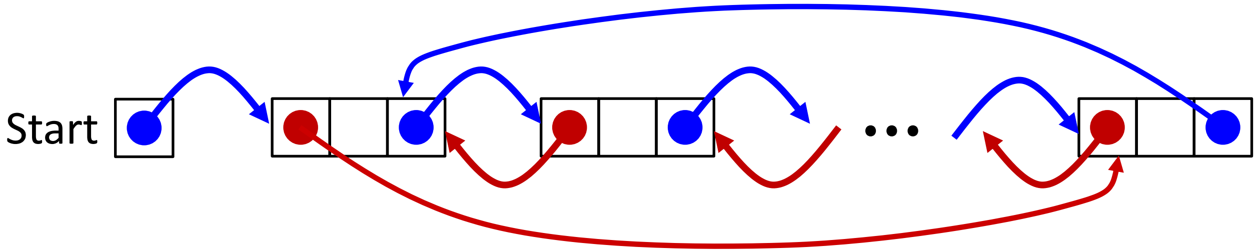
Recall: Doubly-Linked Lists

- **Linear**



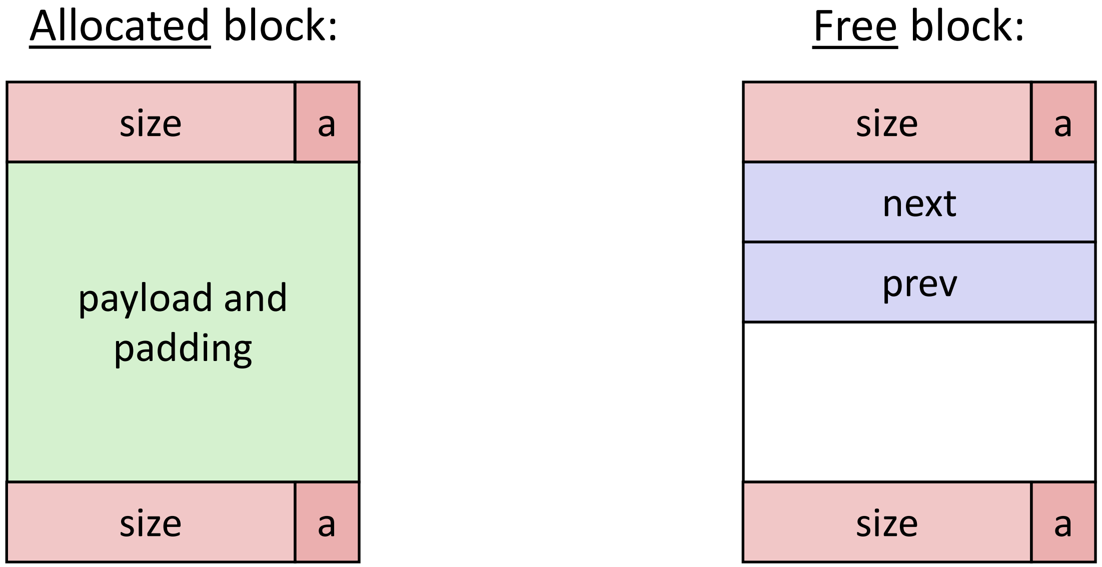
  - Needs head/root pointer
  - First node prev pointer is NULL
  - Last node next pointer is NULL
  - Good for first-fit, best-fit

- **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 List Blocks (Review)



(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
    - These free list pointers affect the minimum block size!
  - Still need boundary tags (header/footer) for coalescing

Polling Question

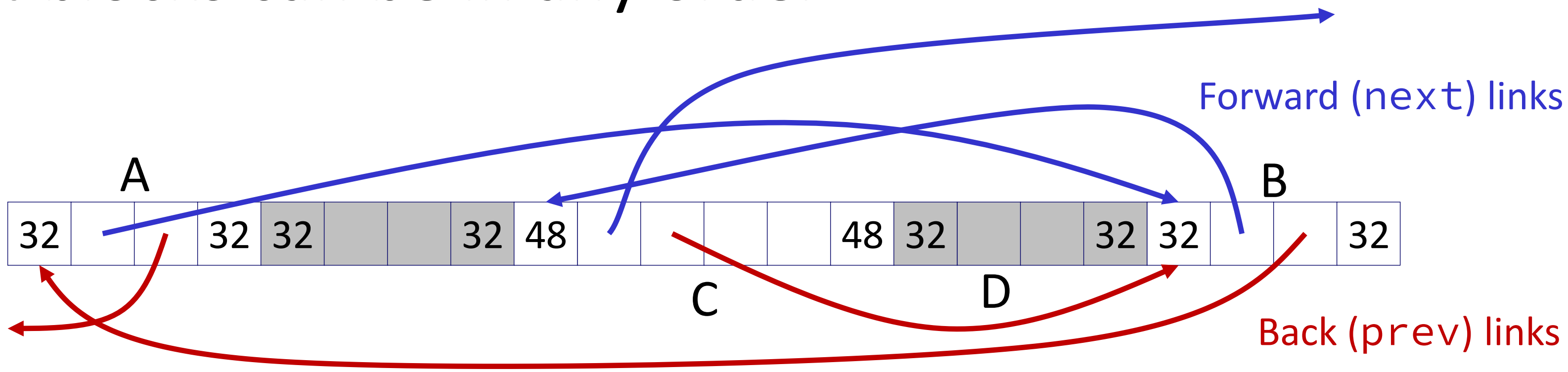
- Determine the minimum block size (MBS) for the given memory allocators, assuming:
  - Allocated blocks must have a payload size ≥ 1 B
  - Boundary tags (headers and footers) are 8 B each

Free List Type	Alloc. Block Boundaries	Free Block Boundaries	Alignment	Alloc. Block MBS	Free Block MBS	MBS
Implicit	header & footer	header & footer	8-byte			
Explicit	header	header & footer	8-byte			
Explicit	header & footer	header & footer	16-byte			

Explicit Free List Organization

- **Logically:** doubly-linked list

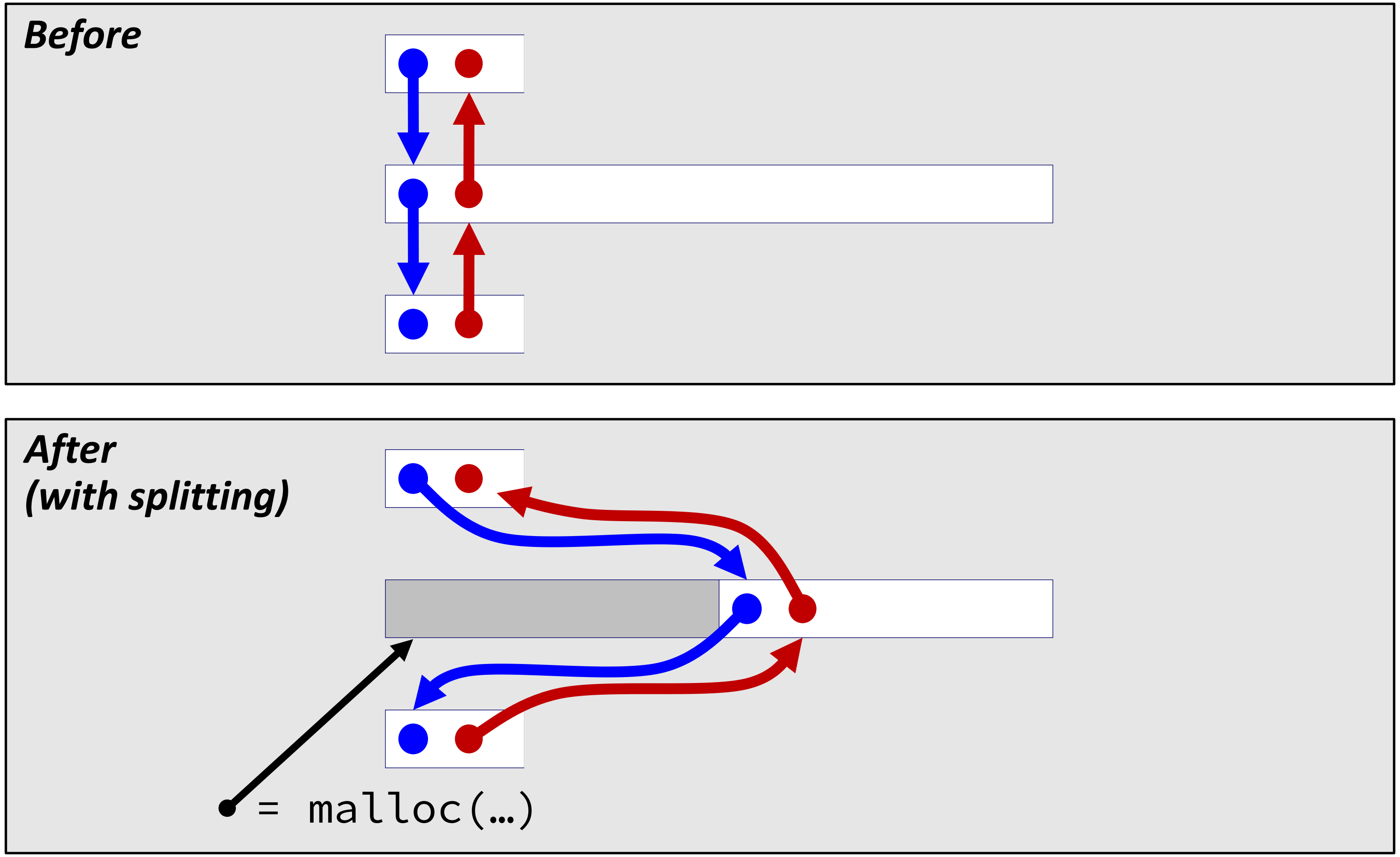
- **Physically:** blocks can be in any order



- **Terminology:**
  - "previous" and "next" blocks are part of the free list
  - "preceding" and "following" blocks are physical neighbors

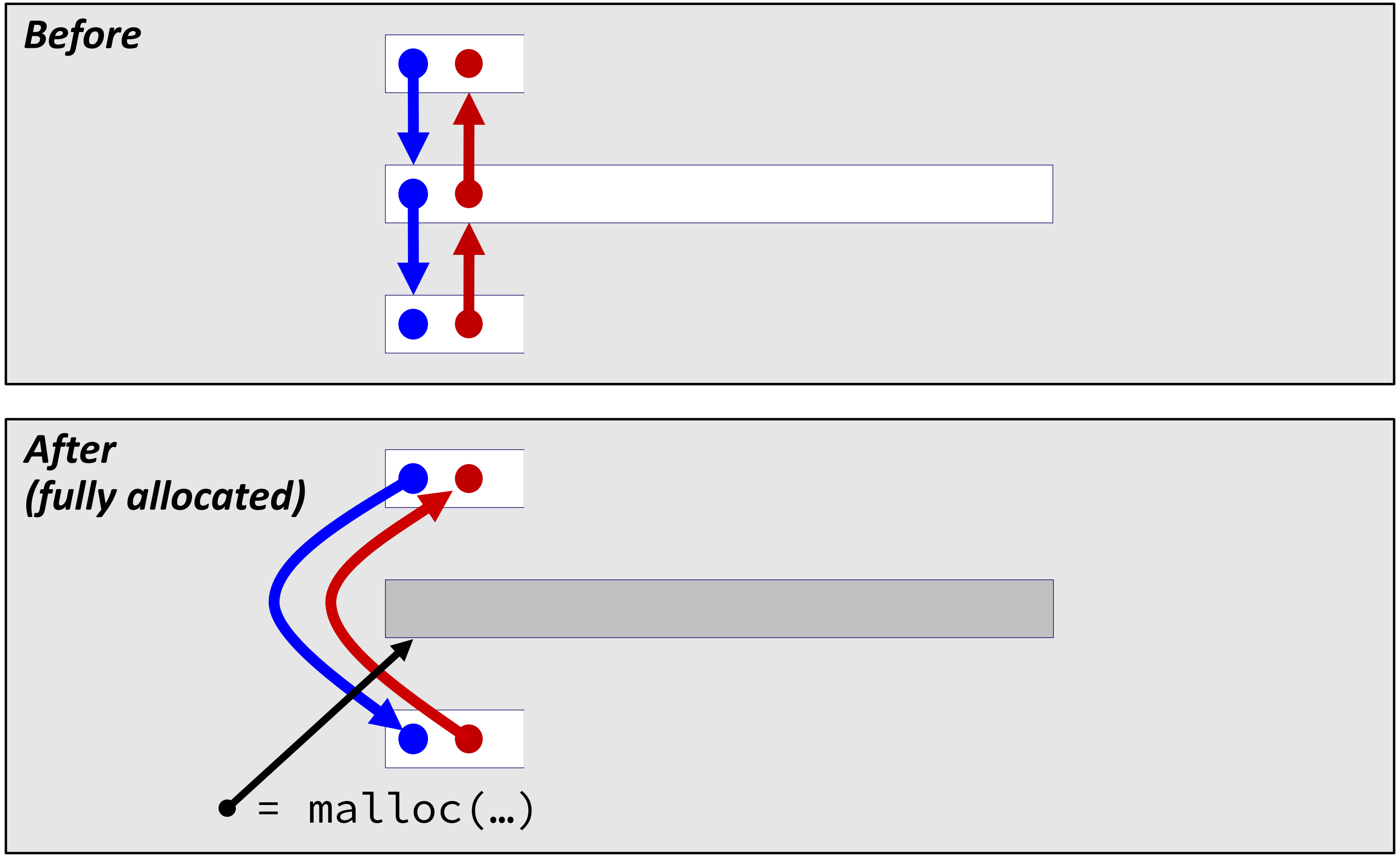
Allocating From Explicit Free Lists: Splitting

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: Full Allocation

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).

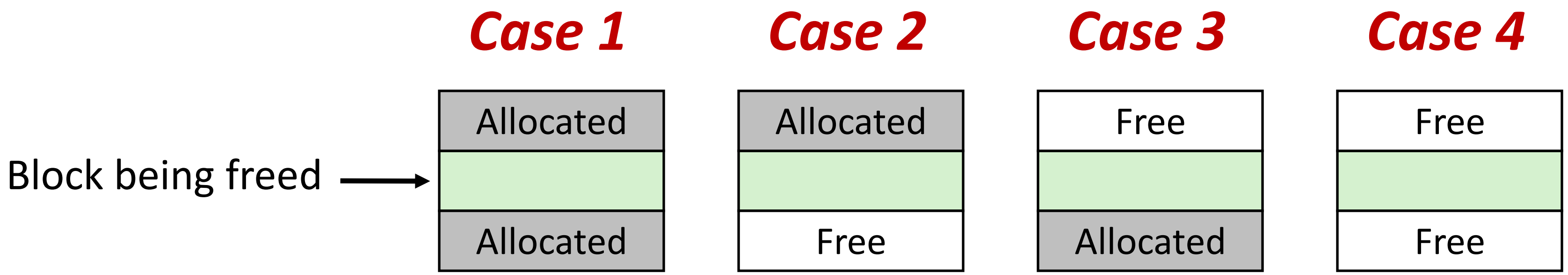


Deallocating With Explicit Free Lists

- *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

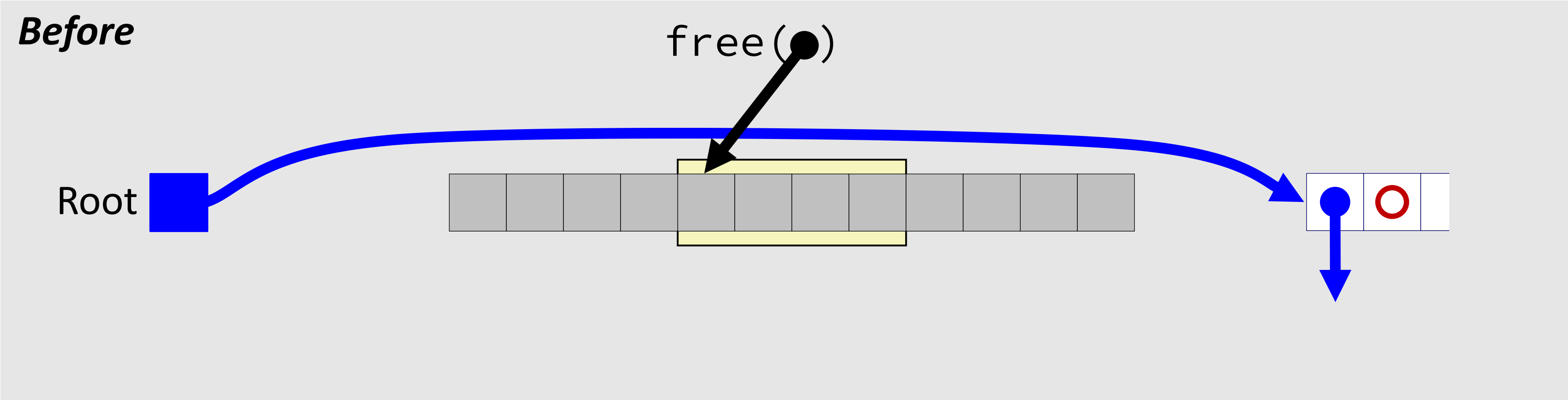


- Neighboring free blocks are *already part of the free list*
  1) Remove old block from free list
  2) Create new, larger coalesced block
  3) Add new block to free list (insertion policy)

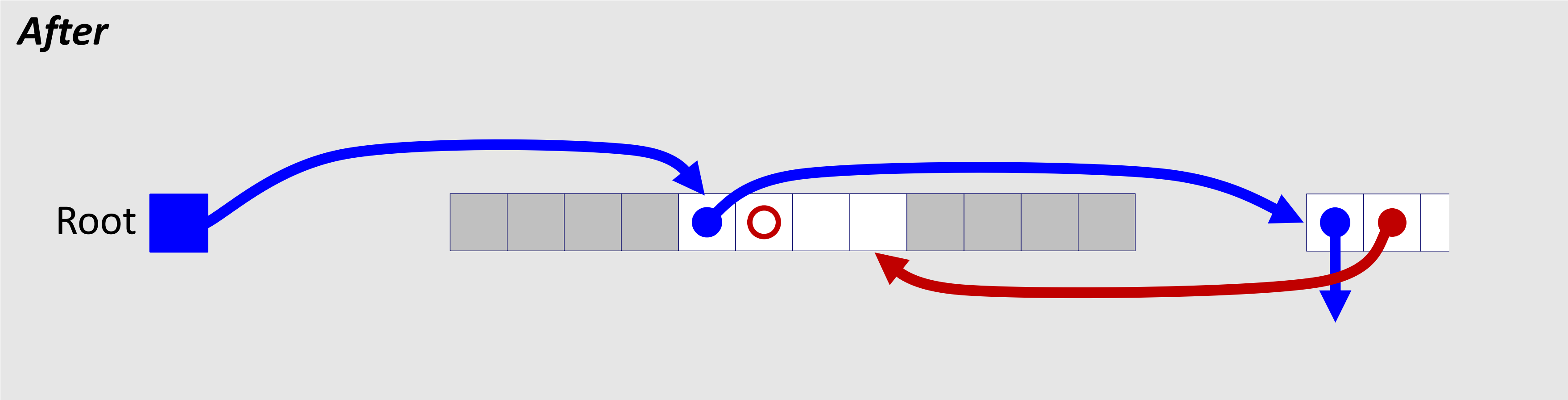
- How do we tell if a neighboring block is free?

Freeing with LIFO Policy (Case 1)

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

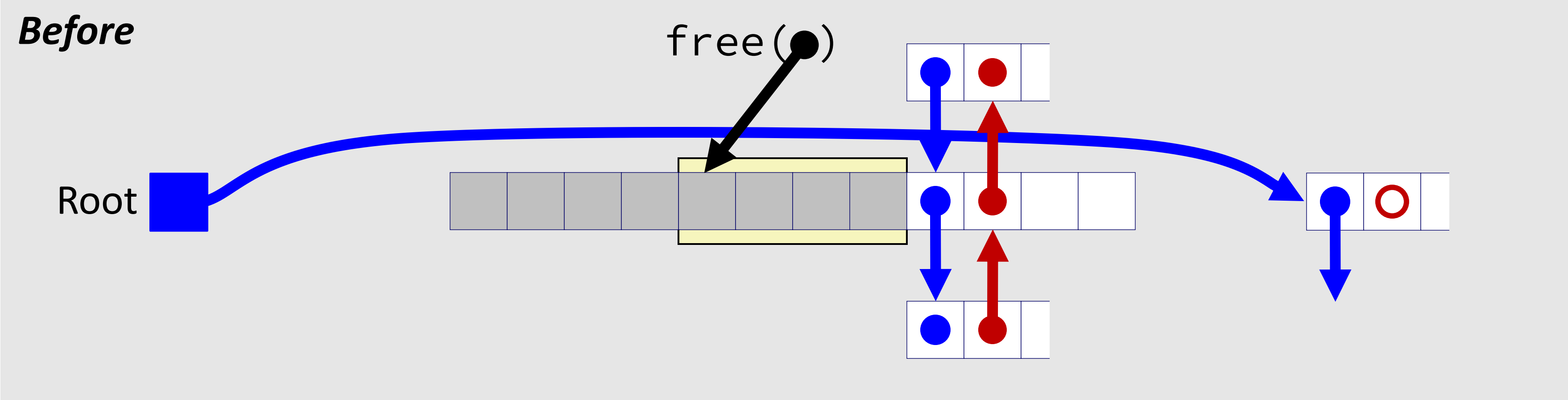


- Insert the freed block at the root of the list

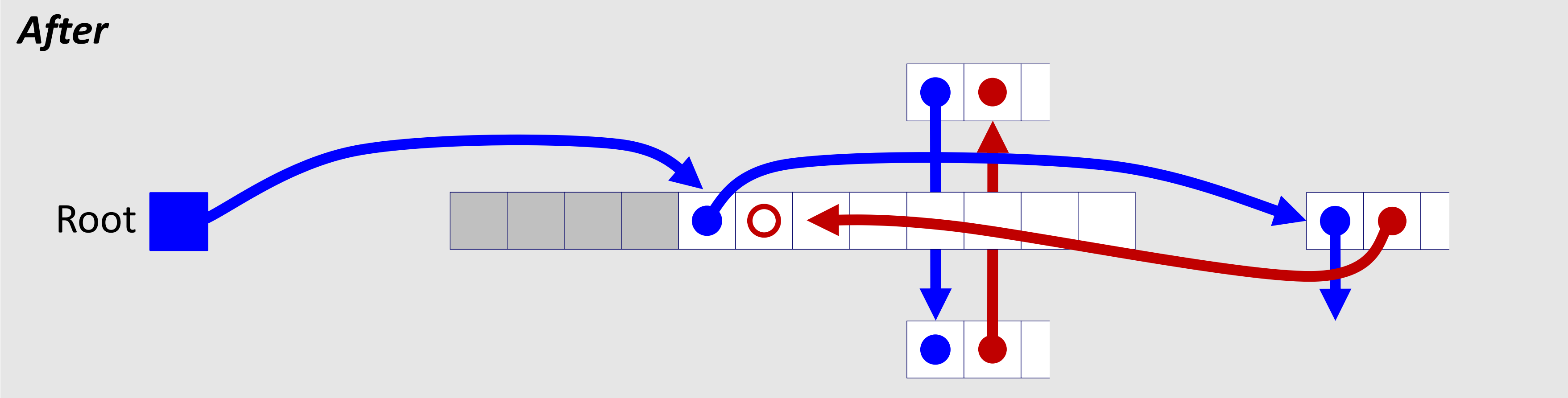


Freeing with LIFO Policy (Case 2)

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

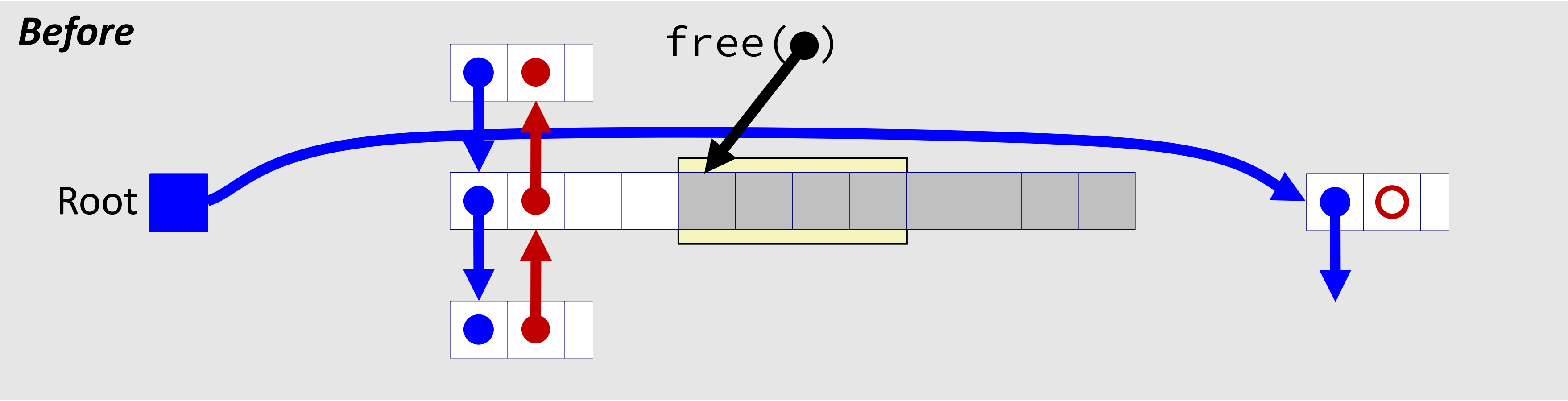


- Splice *following* block out of list, coalesce both memory blocks, and insert the new block at the root of the list

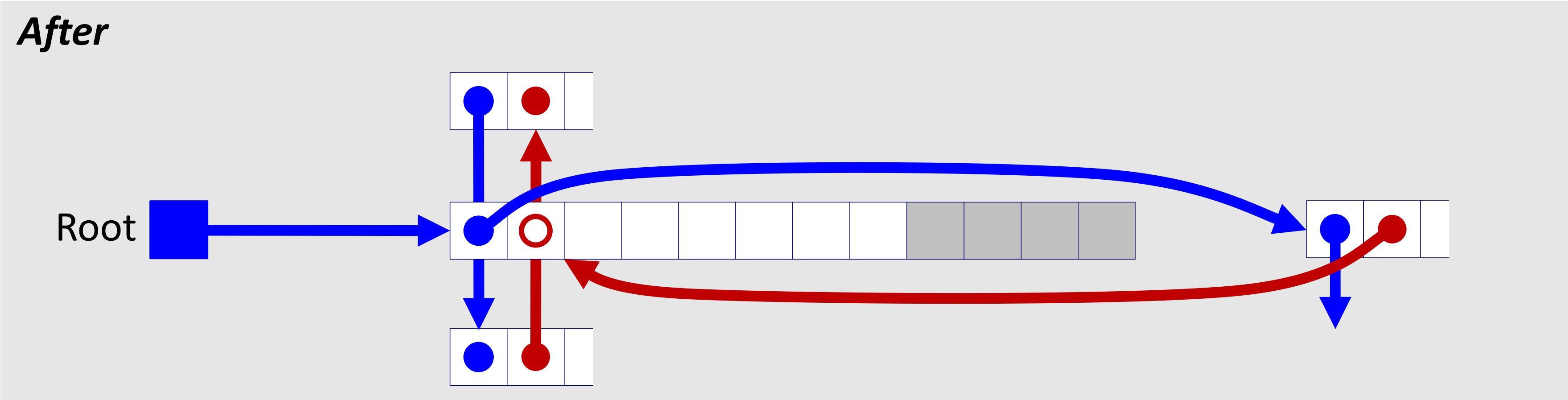


Freeing with LIFO Policy (Case 3)

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

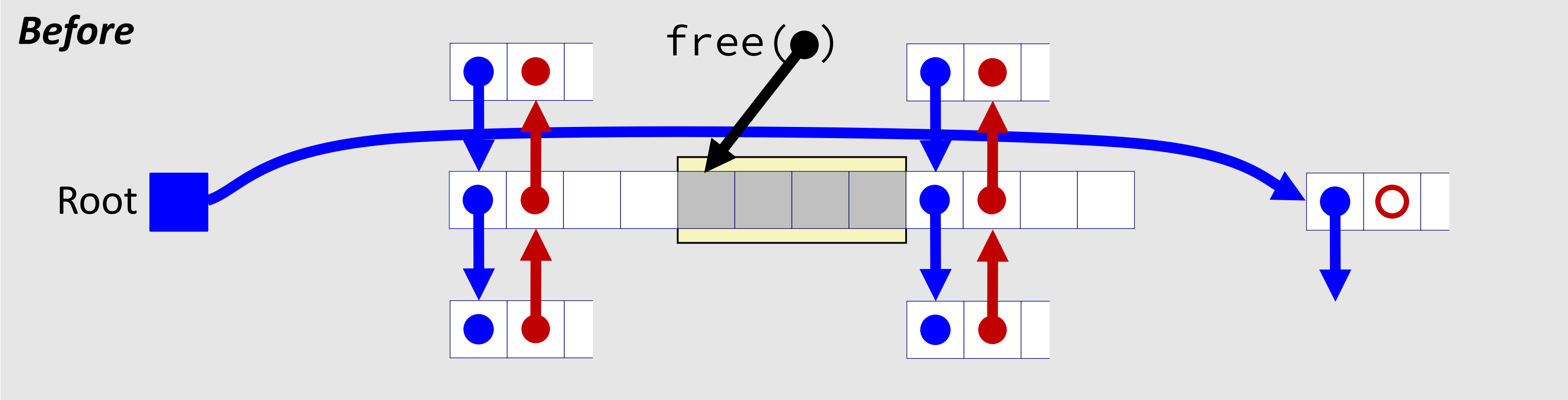


- Splice *preceding* block out of list, coalesce both memory blocks, and insert the new block at the root of the list

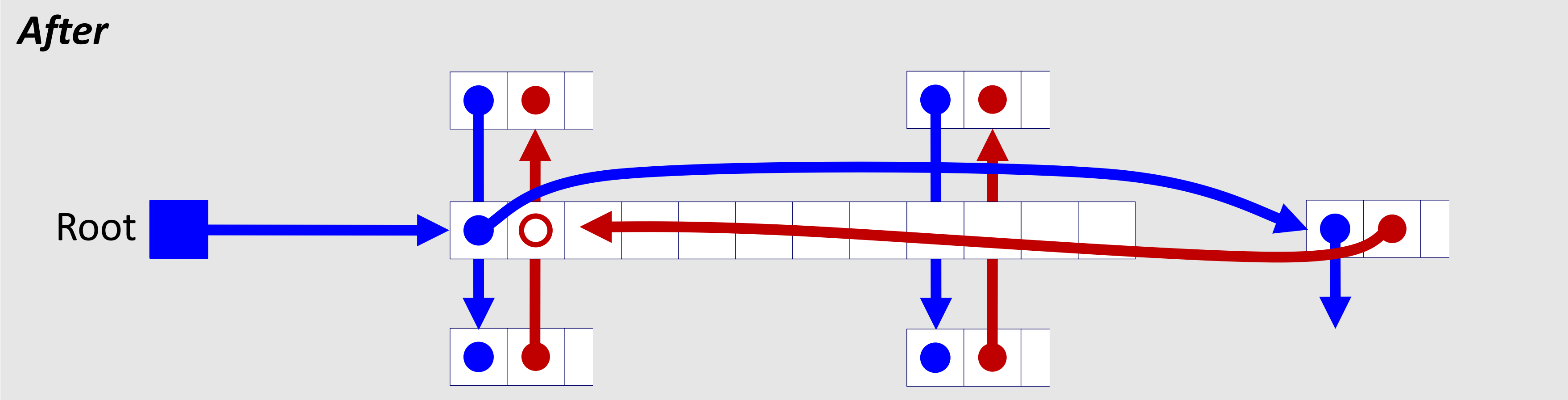


Freeing with LIFO Policy (Case 4)

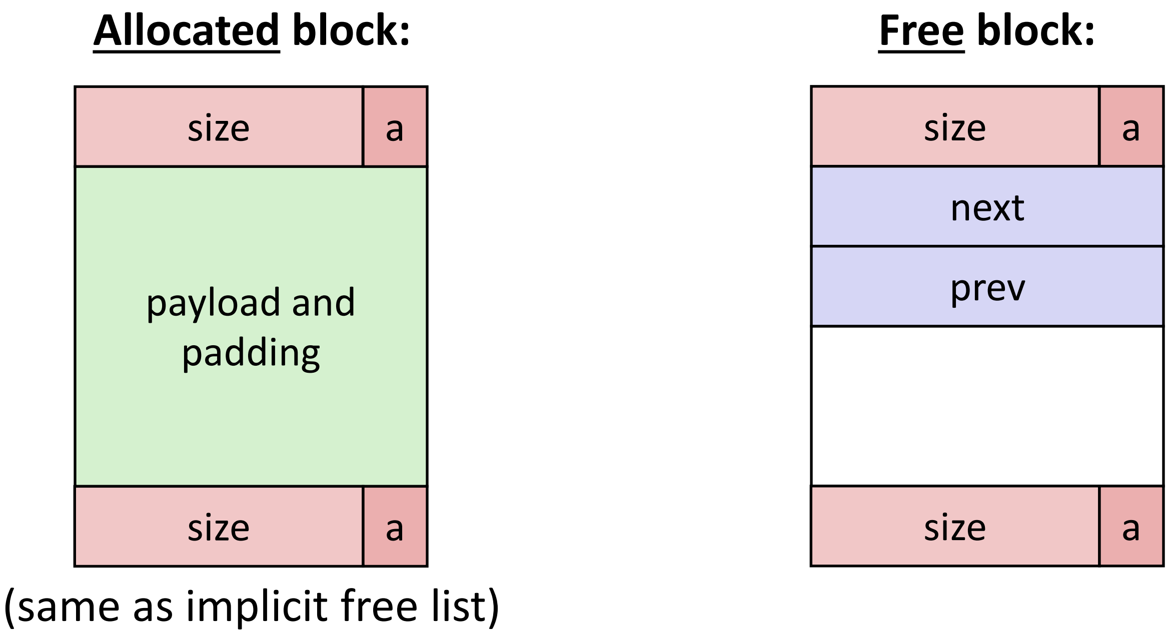
Boundary tags not shown, but don't forget about them!



- Splice *preceding* and *following* 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?

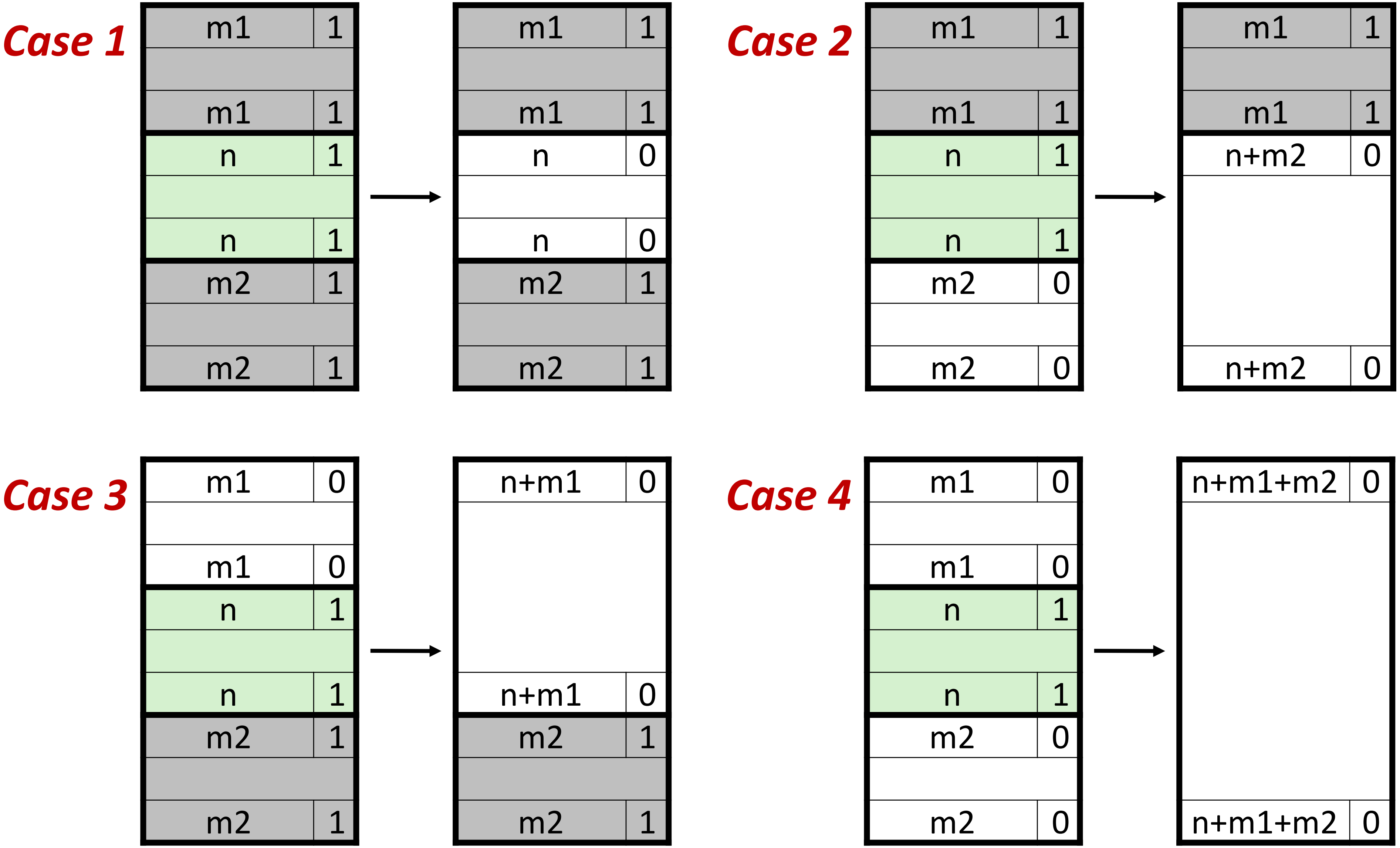


- Lab 5 suggests no…

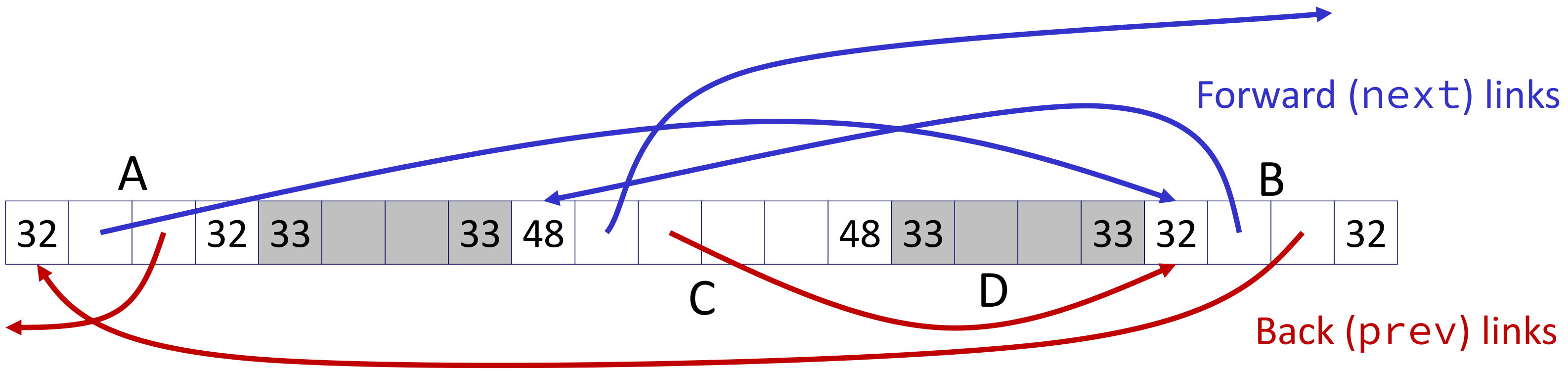
Summary: Fulfilling an Allocation Request

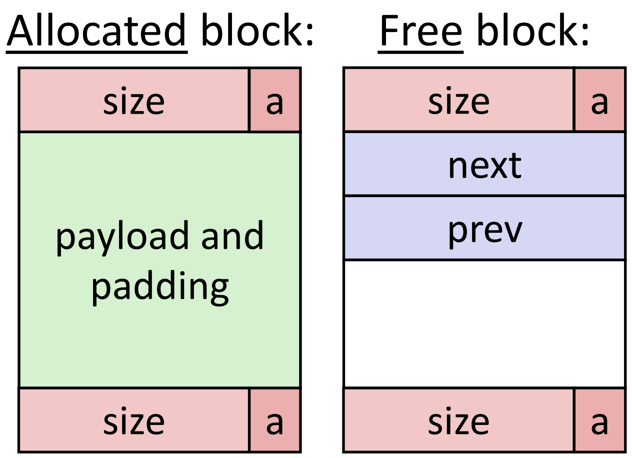
1) Compute the necessary block size
2) Search for a suitable free block using the allocator's *allocation strategy*
   - If found, continue
   - If not found, return `NULL`
3) Compare the necessary block size against the size of the chosen block
   - If equal, allocate the block
   - If not, *split* off the excess into a new free block before allocating the block
4) Return the address of the beginning of the payload

Summary: Constant Time Coalescing



Summary: 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

BONUS SLIDES

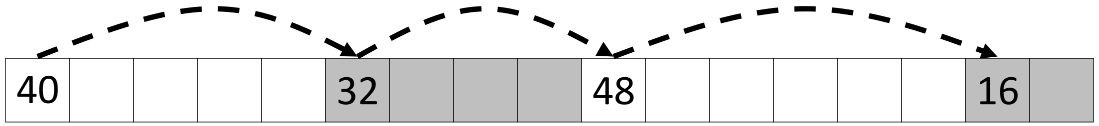
The following slides are about the **SegList Allocator**, for those curious. You will NOT be expected to know this material.

Keeping Track of Free Blocks

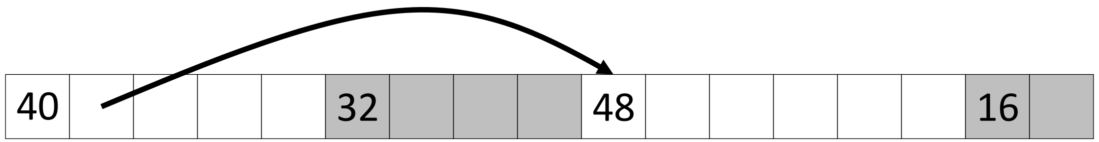
= 8-byte word (free)

= 8-byte word (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

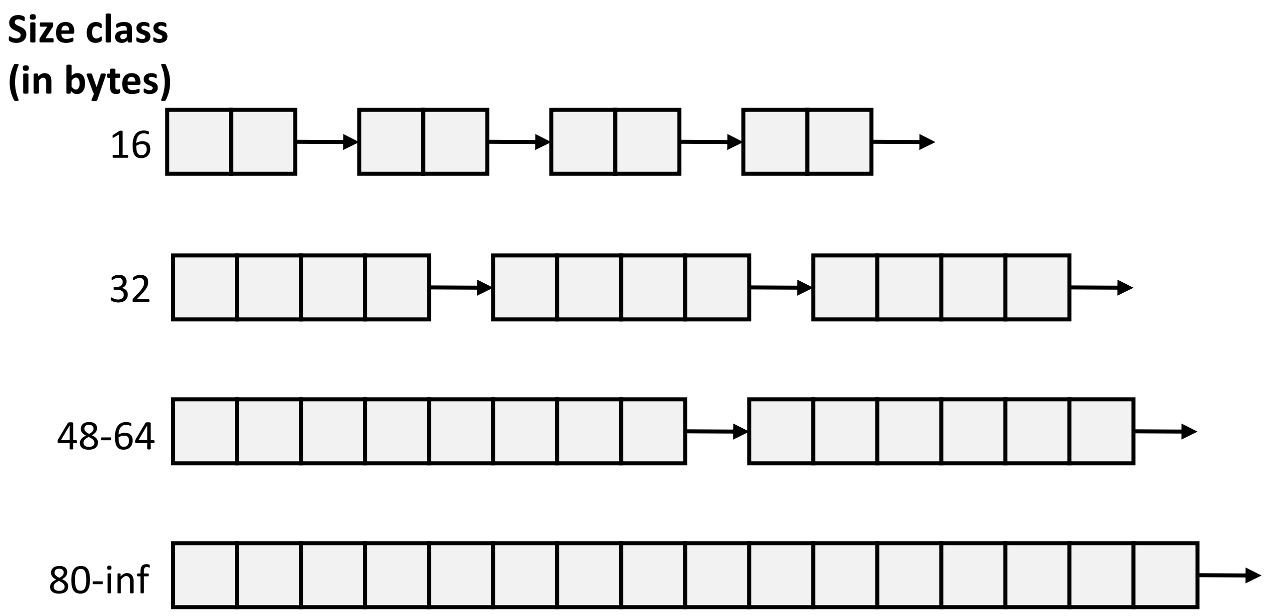


3) *Segregated free list*
   - Different free lists for different size "classes"

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

Segregated List (SegList) Allocators

- Each *size class* of blocks has its own free list
- Organized as an array of free lists



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

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