Memory Allocation II

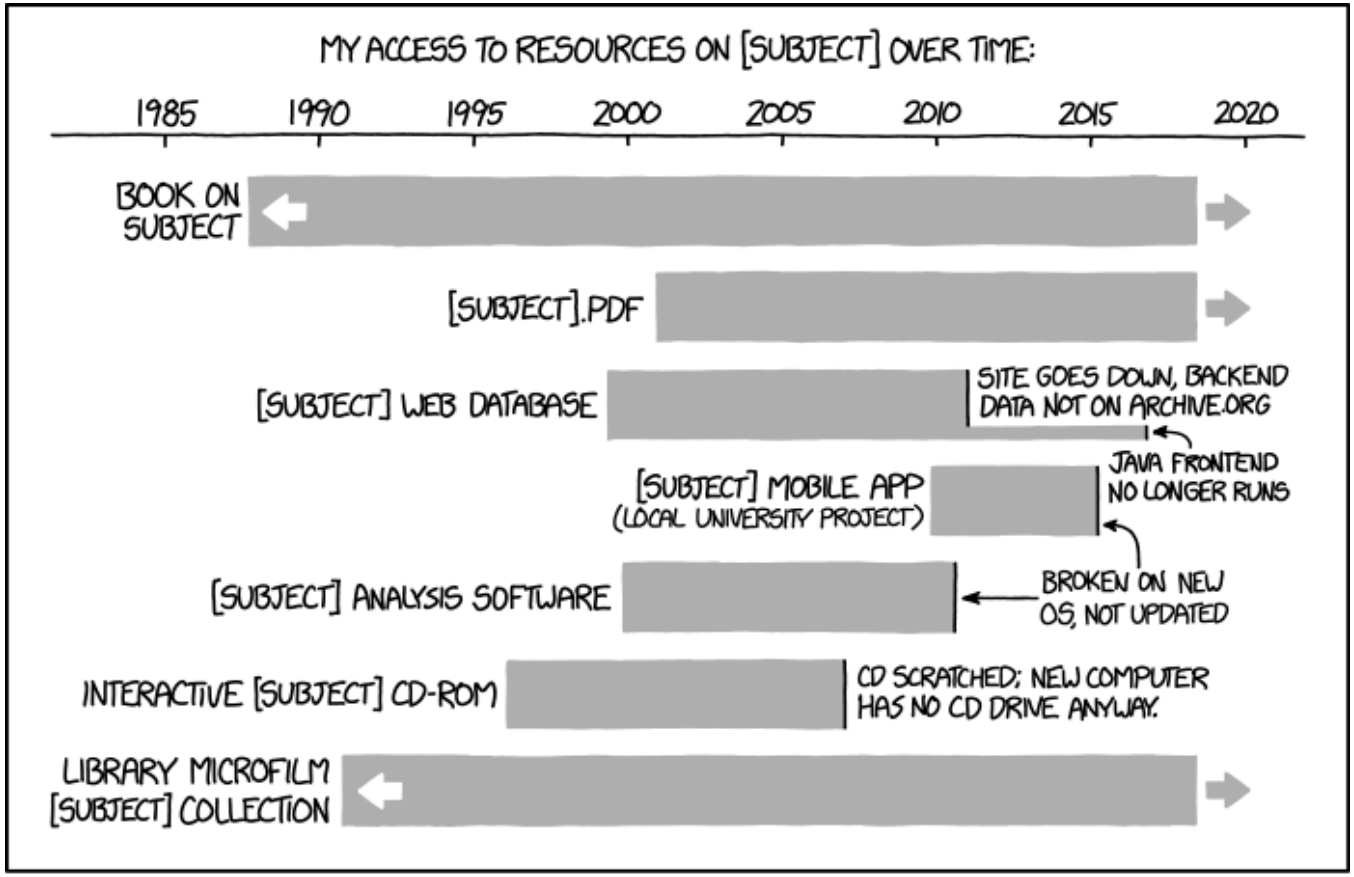
CSE 351 Spring 2019

Instructor:

Ruth Anderson

Teaching Assistants:

Gavin Cai
Jack Eggleston
John Feltrup
Britt Henderson
Richard Jiang
Jack Skalitzky
Sophie Tian
Connie Wang
Sam Wolfson
Casey Xing
Chin Yeoh



http://xkcd.com/1909/

Administrivia

- Homework 5, due Friday (5/31)
  - Processes and Virtual Memory
- Lab 5, released Wed Evening, due Friday (6/7)
  - Memory Allocation
  - Recommended that you watch the Lab 5 helper videos
  - Sunday 6/9 is last day Lab 5 may be submitted (if one late day is used)
- **Final Exam:** Wed, 6/12, 12:30-2:20 pm in KNE 130

Peer Instruction Question

external fragmentation is when there is enough free space in heap to fit next block, its just not contiguous, so you have to allocate more space on the heap anyways

- 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:

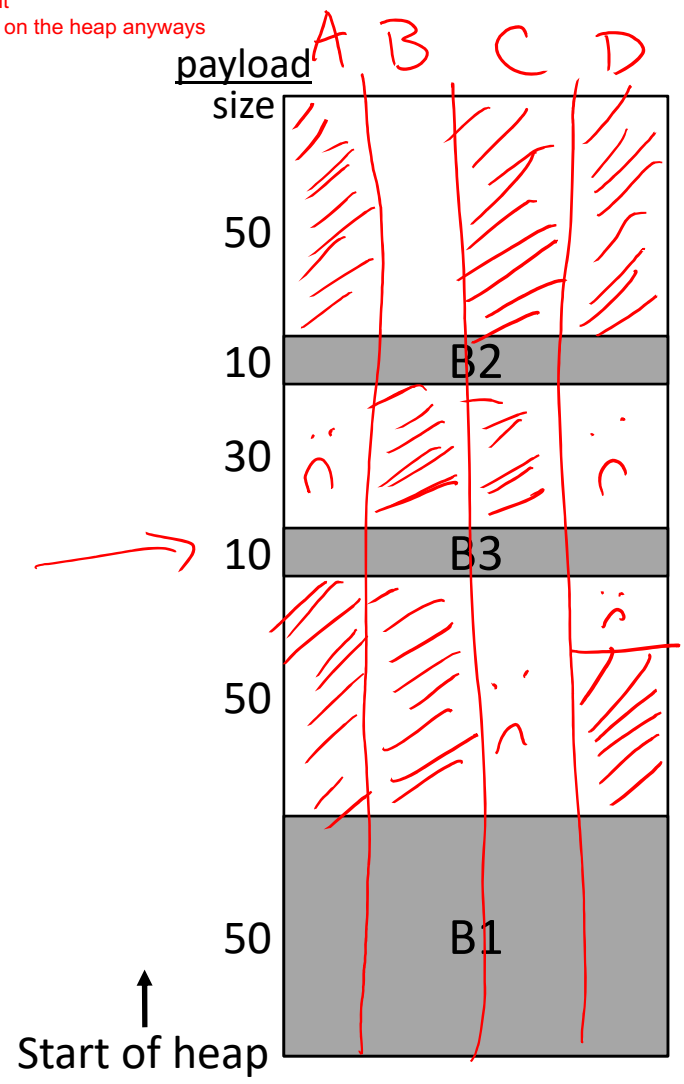
```
malloc(50), malloc(30)
```

(C) Next-fit: start searching heap at B3

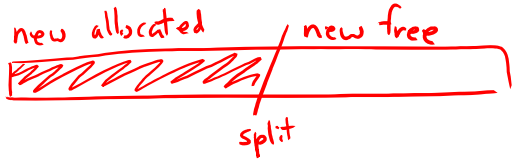
```
malloc(30), malloc(50)
```

(D) Next-fit: start searching heap at B3

```
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

rather than just allocating the entire free block for the given payload, only allocate part of it (if the remainder might be large enough for some other payload in the future.

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+15) >> 4) << 4;           // round up to multiple of 16
  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 +)
```

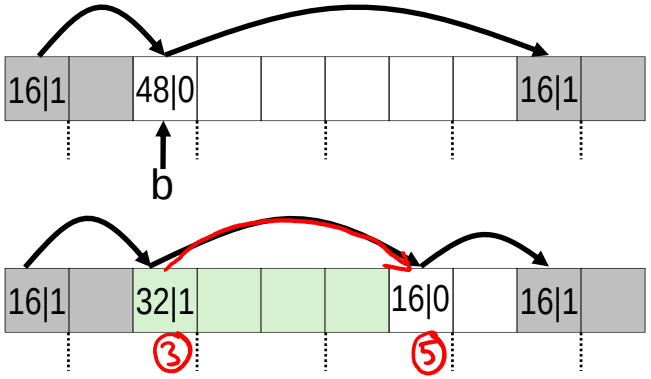
b is a pointer to the block header, so *b reads the size of the block

create the new free block starting at address b+newsize (this line sets the header with info on that new blocks size)

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

header

sets a=1



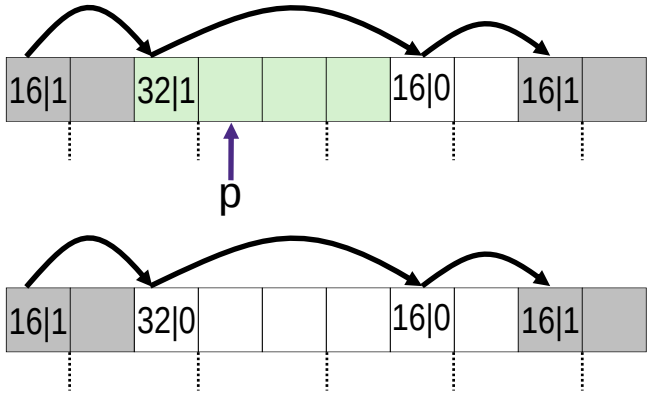
Free word

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"



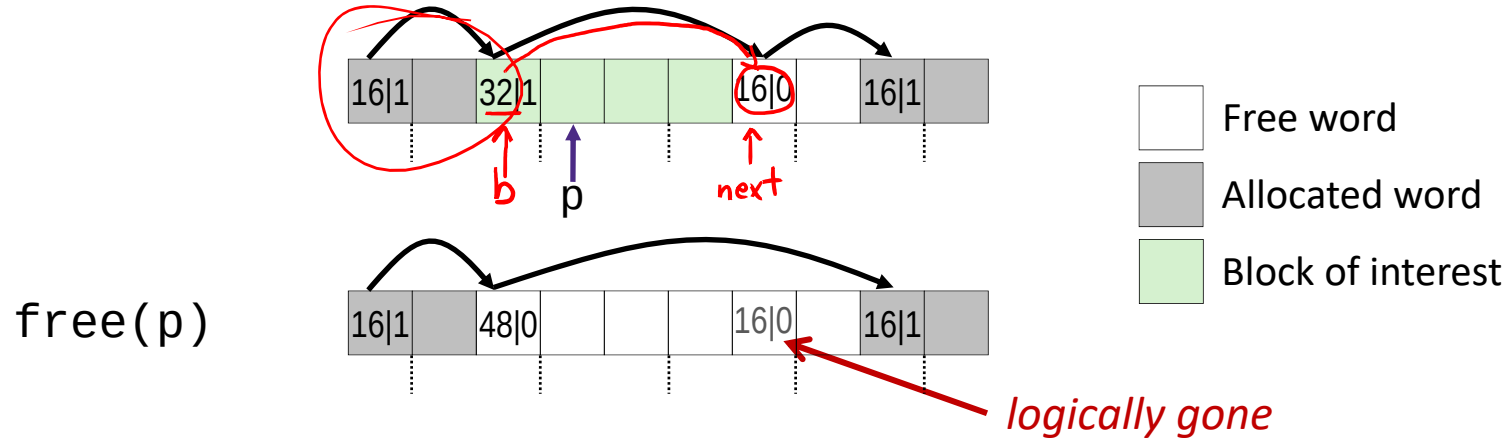
`malloc(40)`

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

here we have two adjacent free blocks, but neither alone is large enough to fit the 40 bytes requested by malloc. Hence it looks like there's not enough room, when actuality we just want to merge these two adjacent free blocks

Implicit List: Coalescing with Next

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




```
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

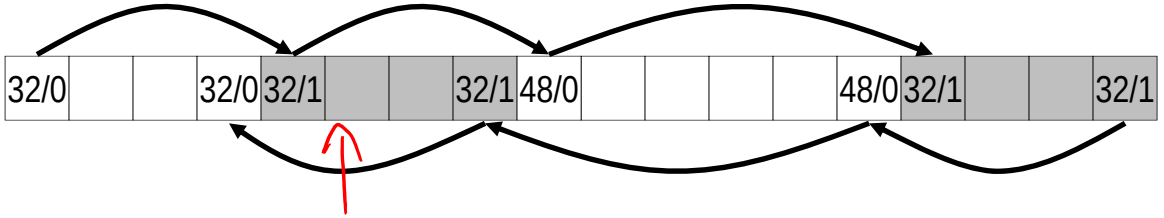
Have no way of telling where the previous block starts (Because we don't know its size) so no way of reading its header. The implicit list only lets us jump forward!

Implicit List: Bidirectional Coalescing

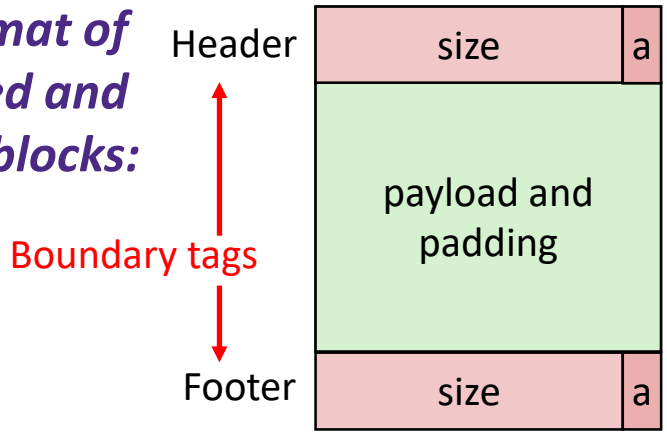
- *Boundary tags* [Knuth73]

e.g. footers as well as headers on our blocks.

  - 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:



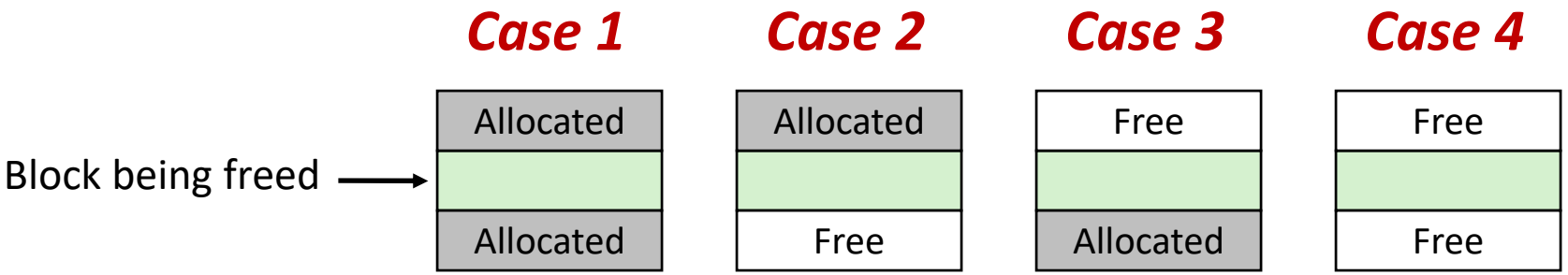
a = 1: allocated block
a = 0: free block

size: block size (in bytes)

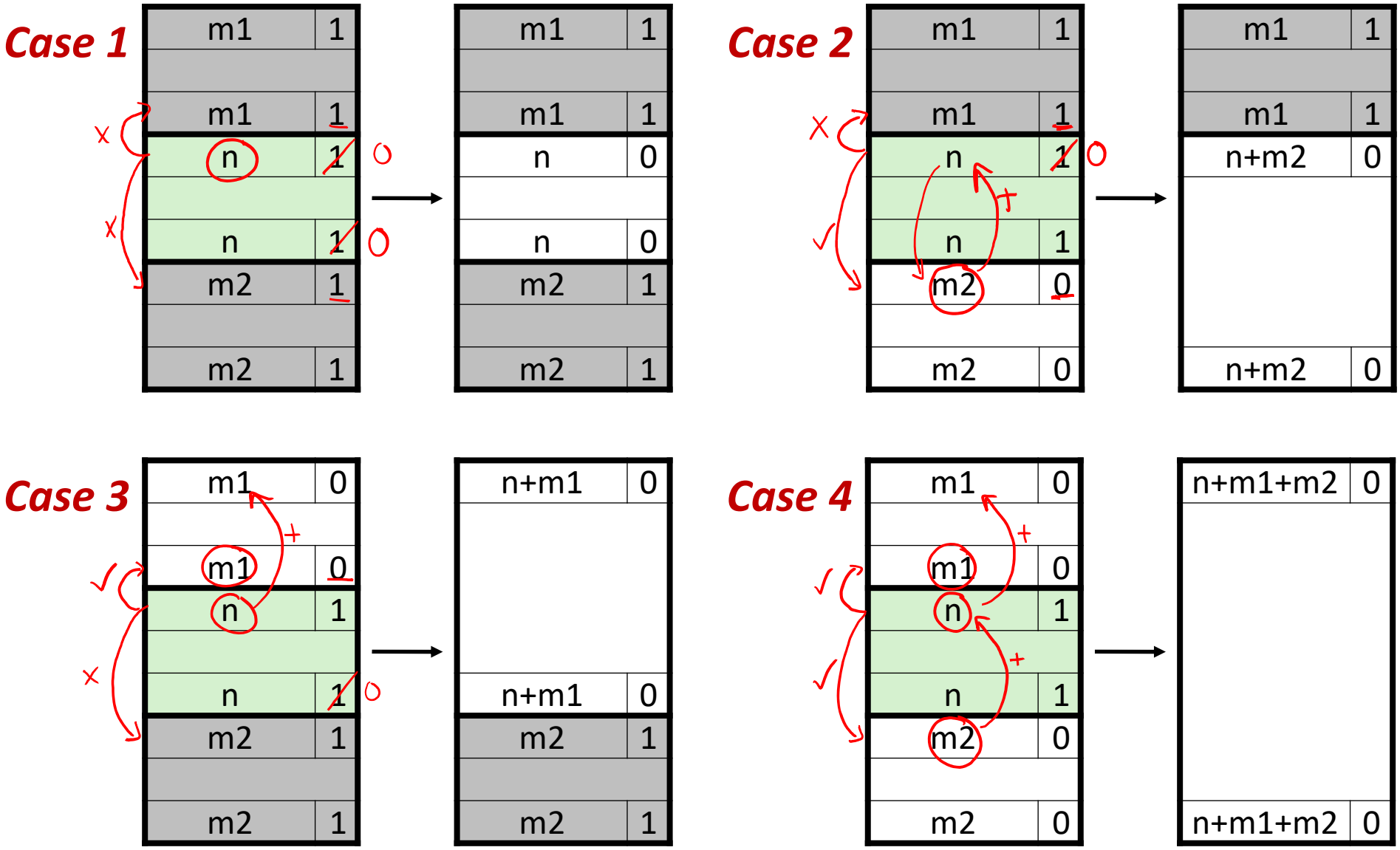
payload: application data (allocated blocks only)

The issue is that if the footer/header are large relative to our payload, the addition of a footer significantly increases our memory uses in the heap (in the case we have lots of small blocks).

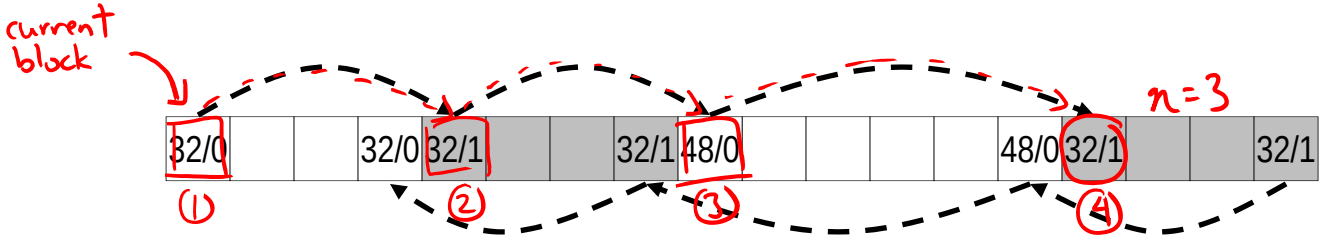
Constant Time Coalescing



Constant Time Coalescing

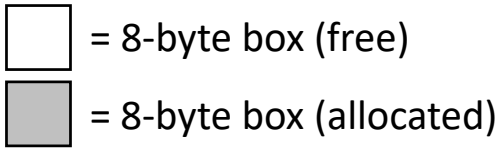


Implicit Free List Review Questions

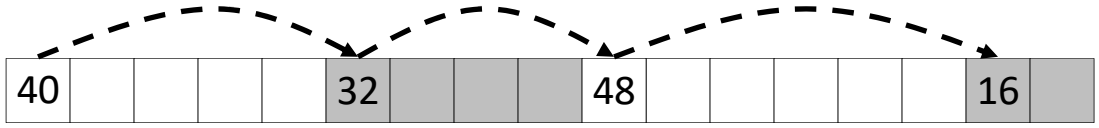


- What is the block header? What do we store and how?
  stores info about block
  size of block, is-allocated?
  ↑ lowest bit of header
- What are boundary tags and why do we need them?
  header and footer (same info)
  so we can traverse list in either direction (particularly for coalescing)
- When 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

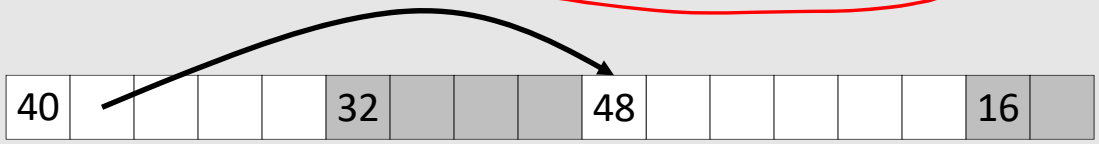


1) *Implicit free list* using length – links all blocks using math
   - No actual pointers, and must check each block if allocated or free



can be slow because we have to traverse allocated blocks as well as free, but simple to implement and does not require any additional data stored in blocks besides header

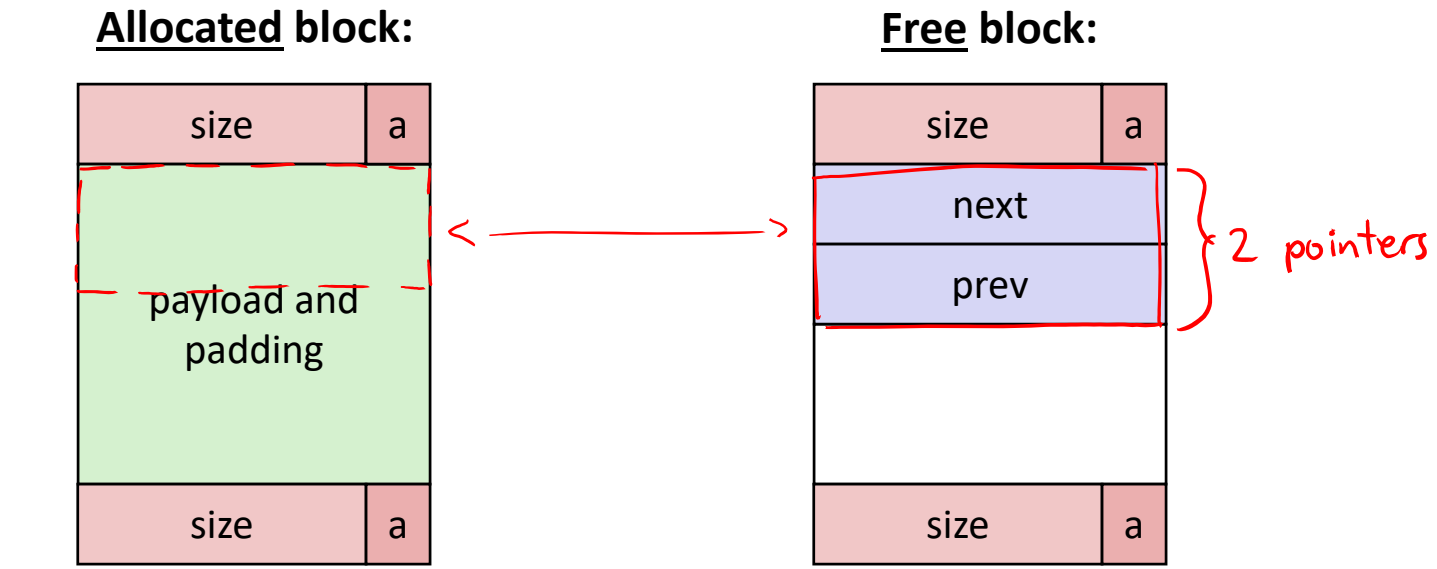
2) *Explicit free list* among only the free blocks, using pointers



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



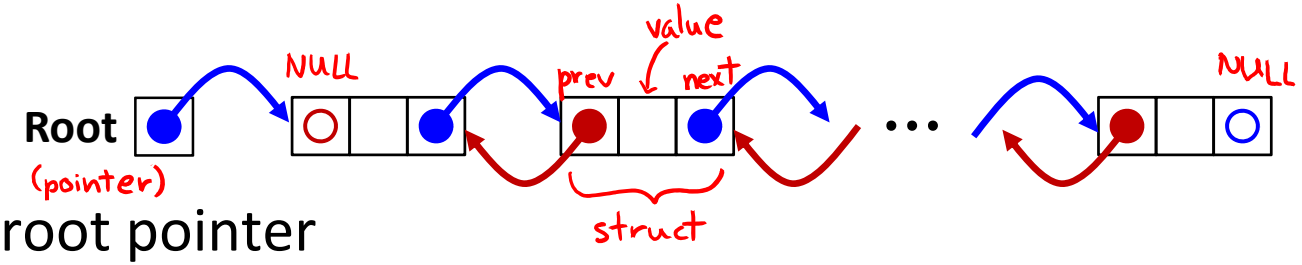
(same as implicit free list)

- 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

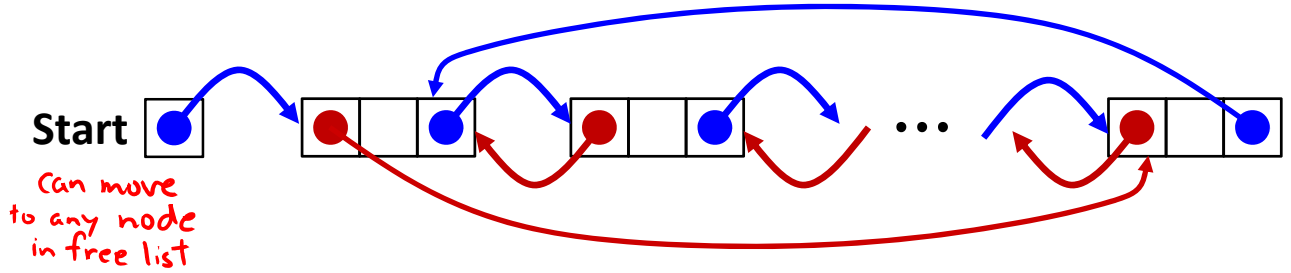
Additional data usage in heap, but allows us to traverse free b faster; trade off

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

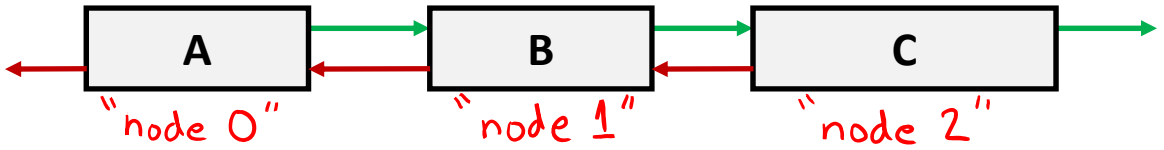


- **Circular**
  - Still have pointer to tell you which node to start with
  - No NULL pointers (term condition is back at starting point)
  - Good for next-fit, best-fit

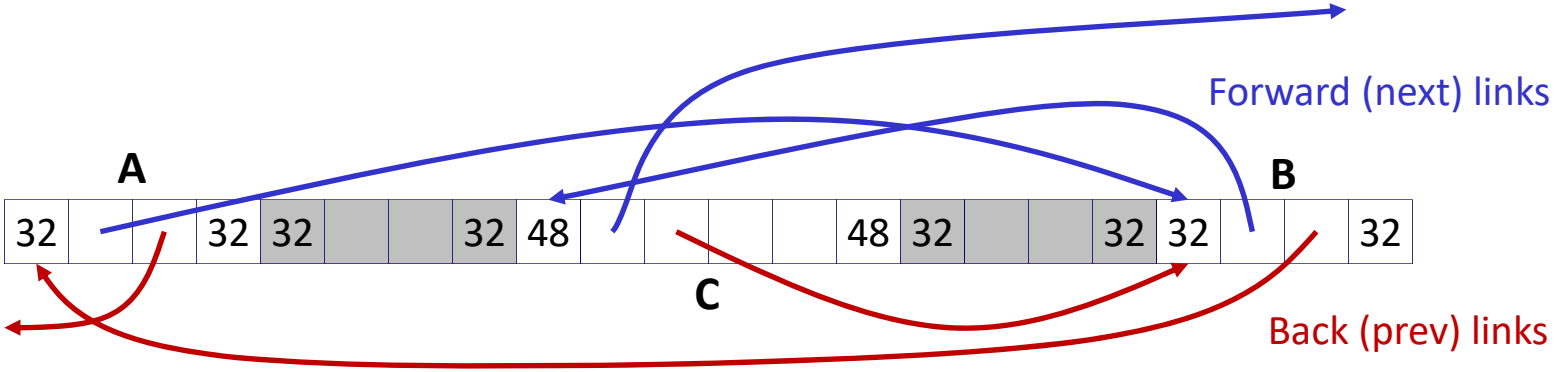
because we might end up having to loop back towards start of free list

Explicit Free Lists

- **Logically:** doubly-linked list

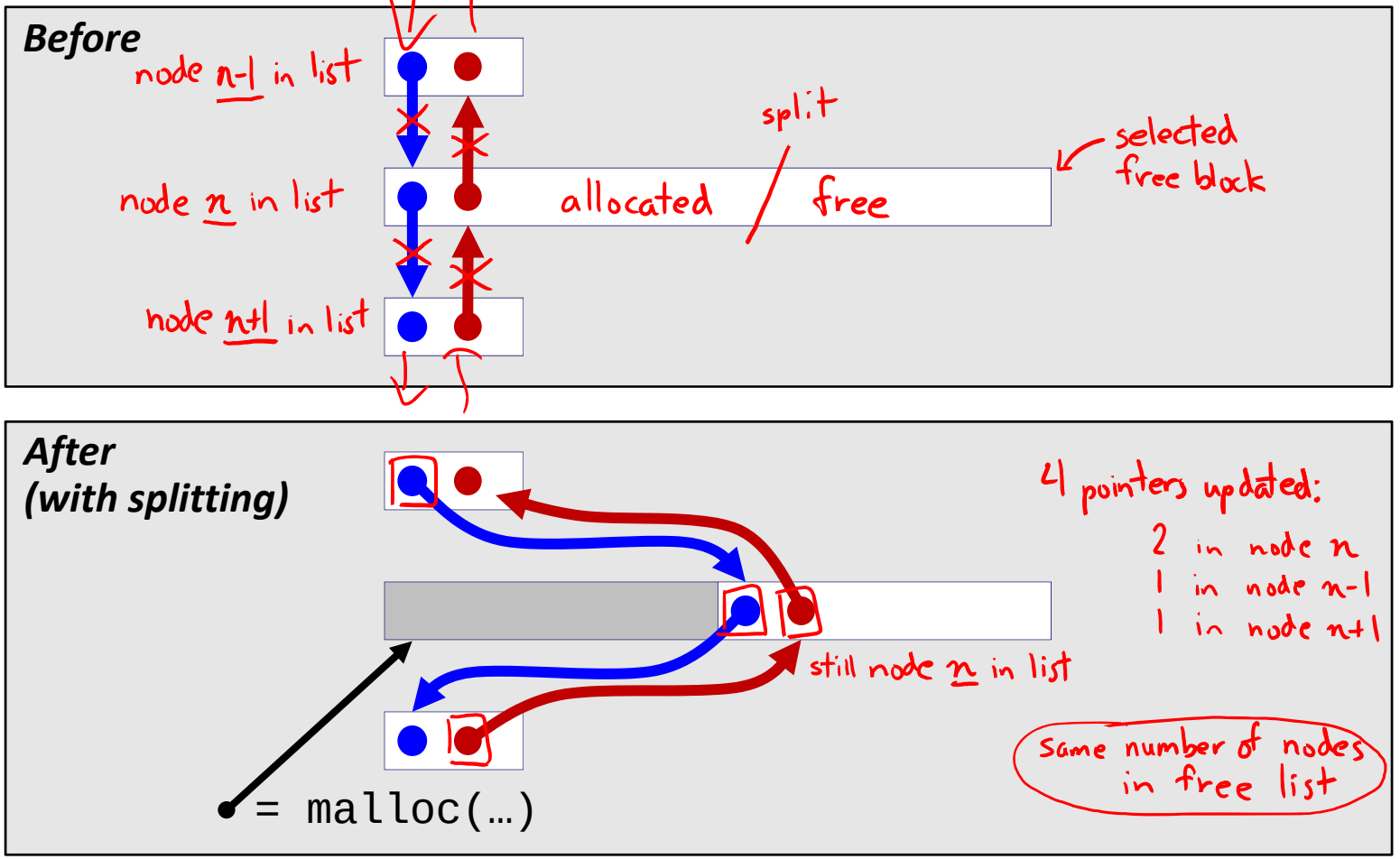


- **Physically:** blocks can be in any order



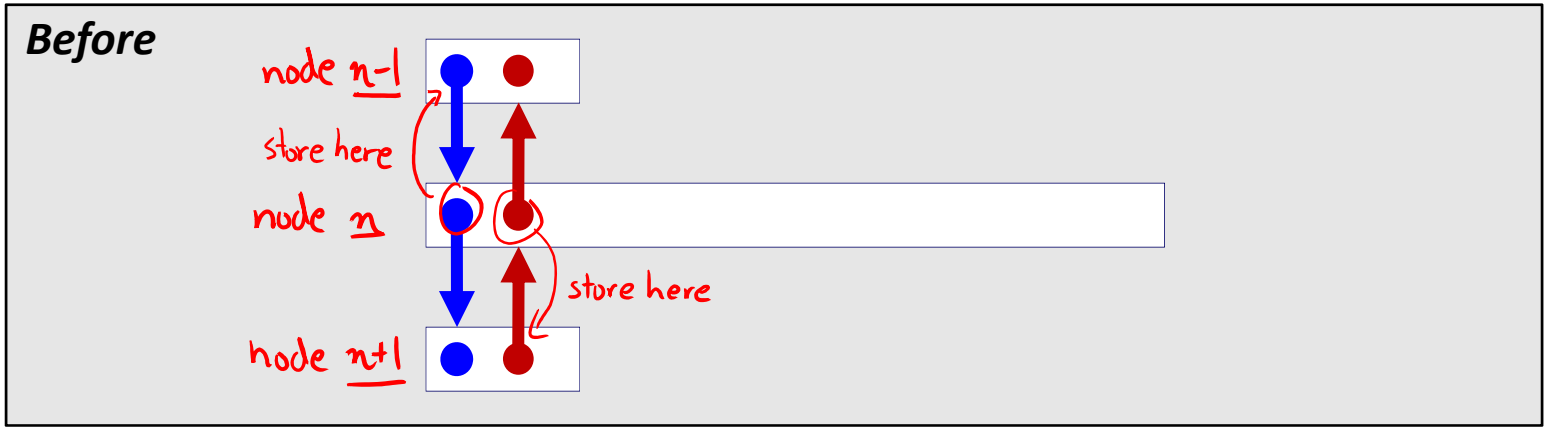
Allocating From Explicit Free Lists

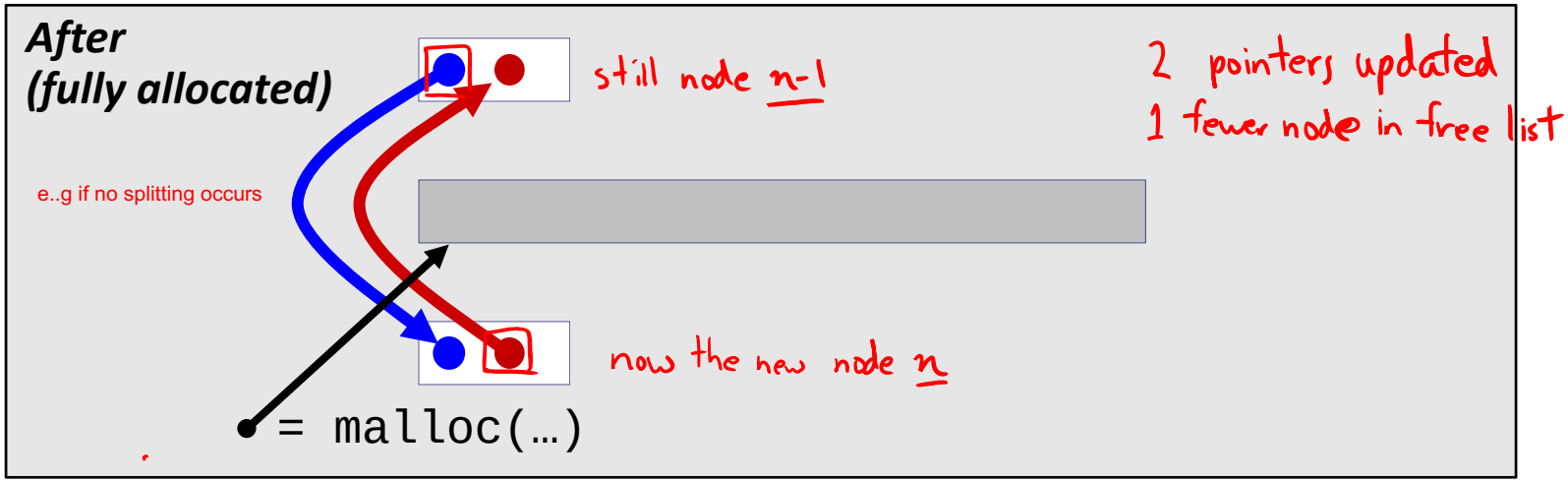
Note: These diagrams are not very specific about *where inside a block* a pointer points. In reality we would always point to one place (e.g. start/header of a block).



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?

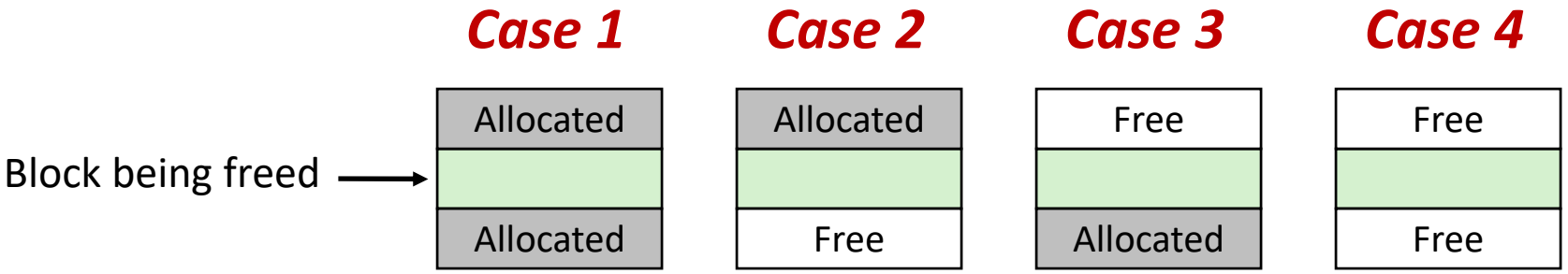
LIFO (last-in-first-out) policy

- 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 extra work to find where to put node.
  - Pro: studies suggest fragmentation is better than the alternative

Coalescing in Explicit Free Lists



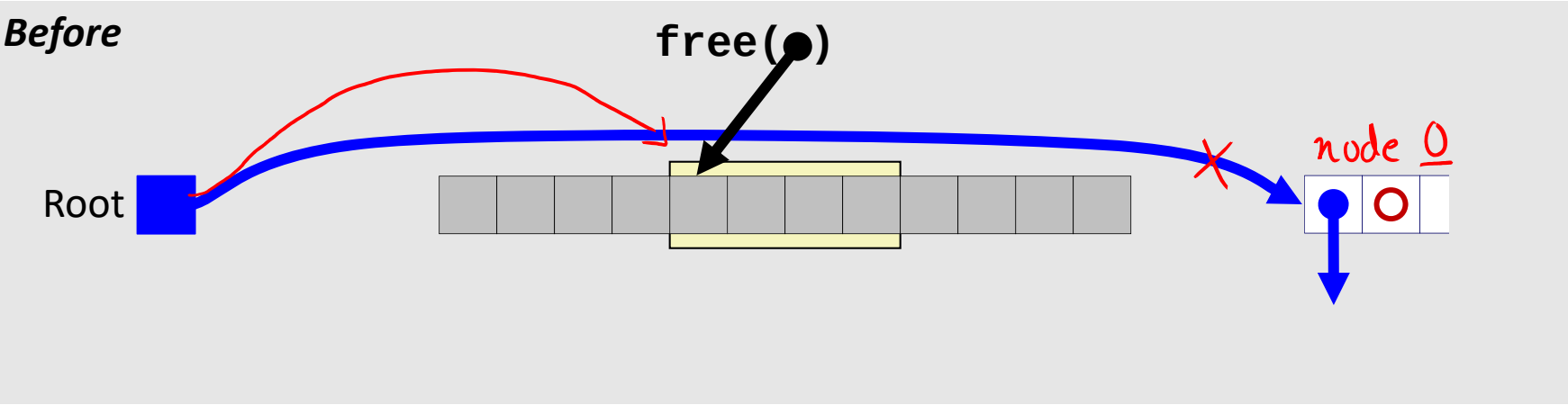
- Neighboring free blocks are *already part of the free list*
  1) Remove old block from free list
  2) Create new, larger coalesced block
  3) Add new block to free list (insertion policy)

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

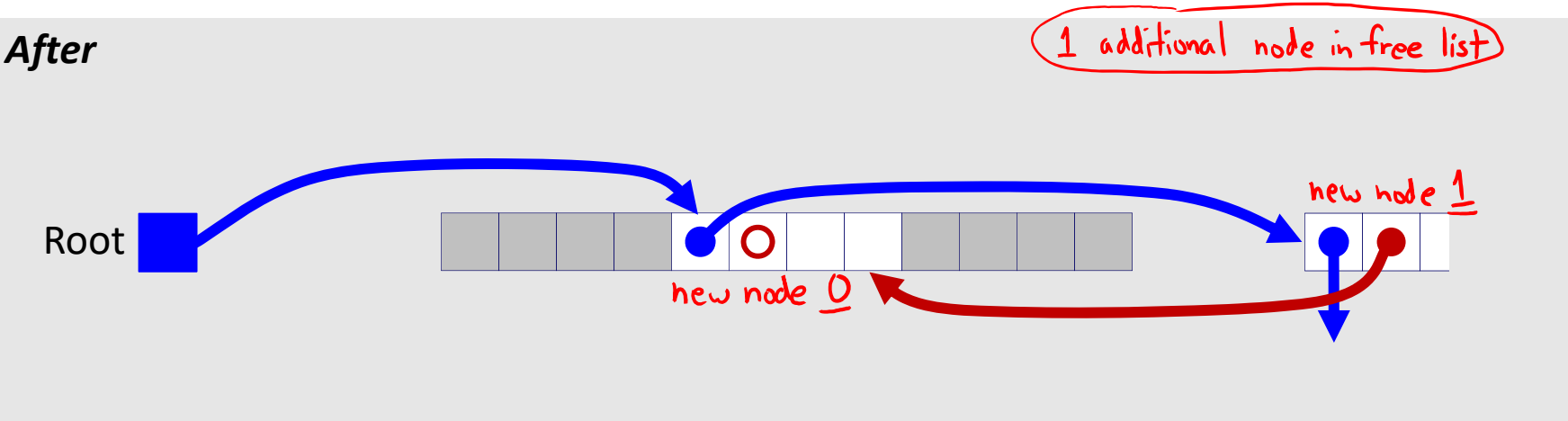
can still use boundary tags (don't need to search free list). other implementations possible (see Lab5)

Freeing with LIFO Policy (Case 1)

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

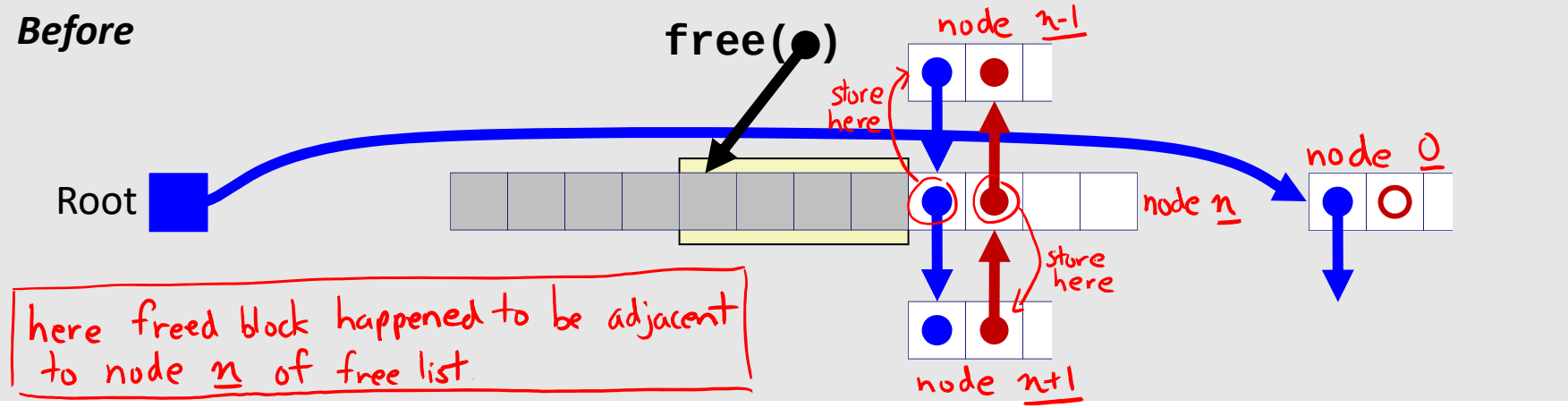


- Insert the freed block at the root of the list

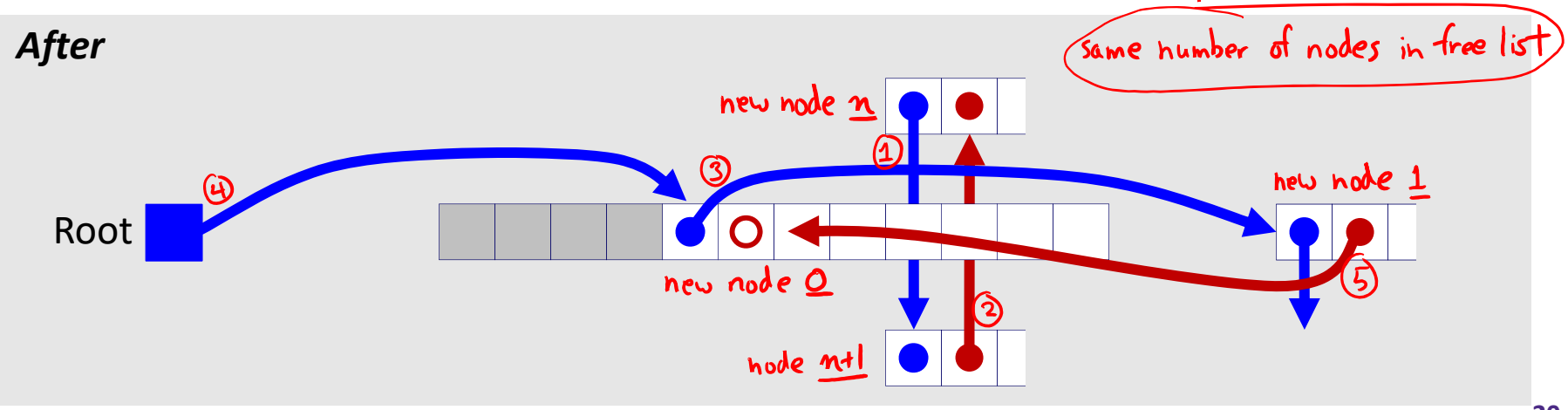


Freeing with LIFO Policy (Case 2)

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

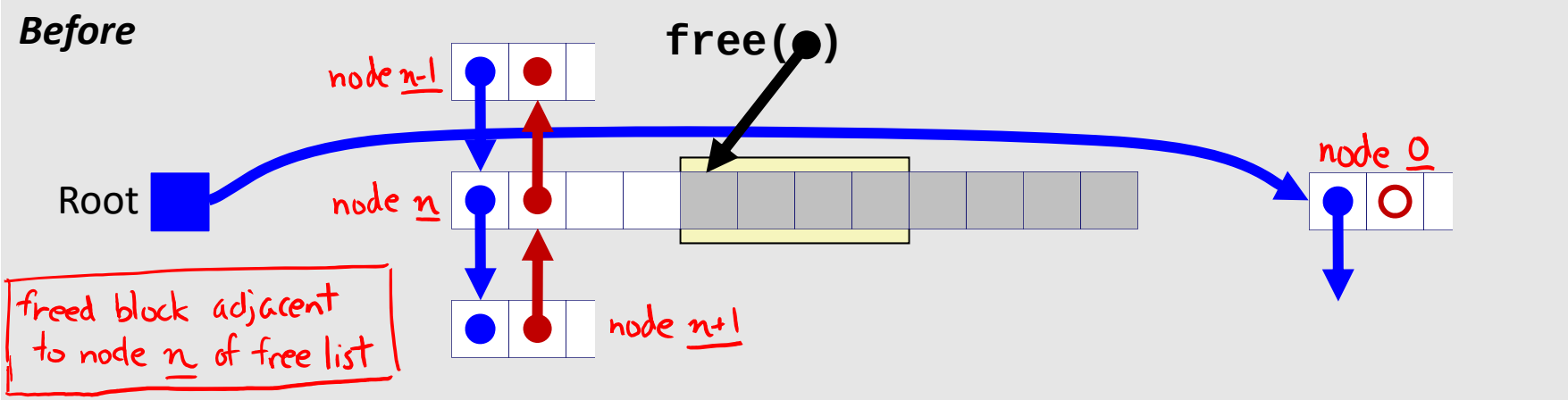


- Splice *successor* block out of list, coalesce both memory blocks, and insert the new block at the root of the list

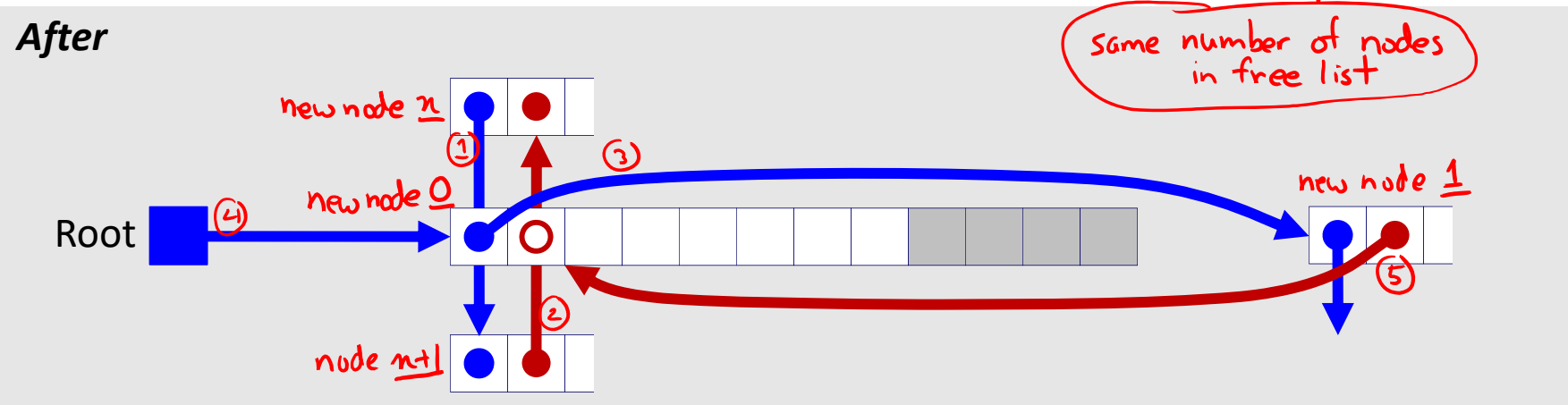


Freeing with LIFO Policy (Case 3)

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

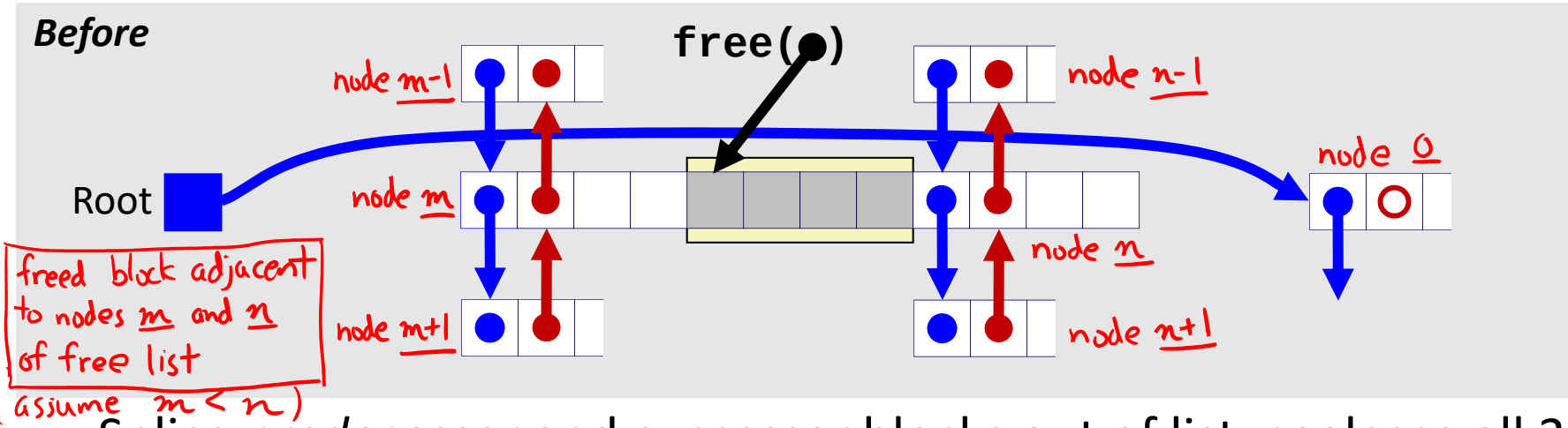


- Splice *predecessor* block out of list, coalesce both memory blocks, and insert the new block at the root of the list

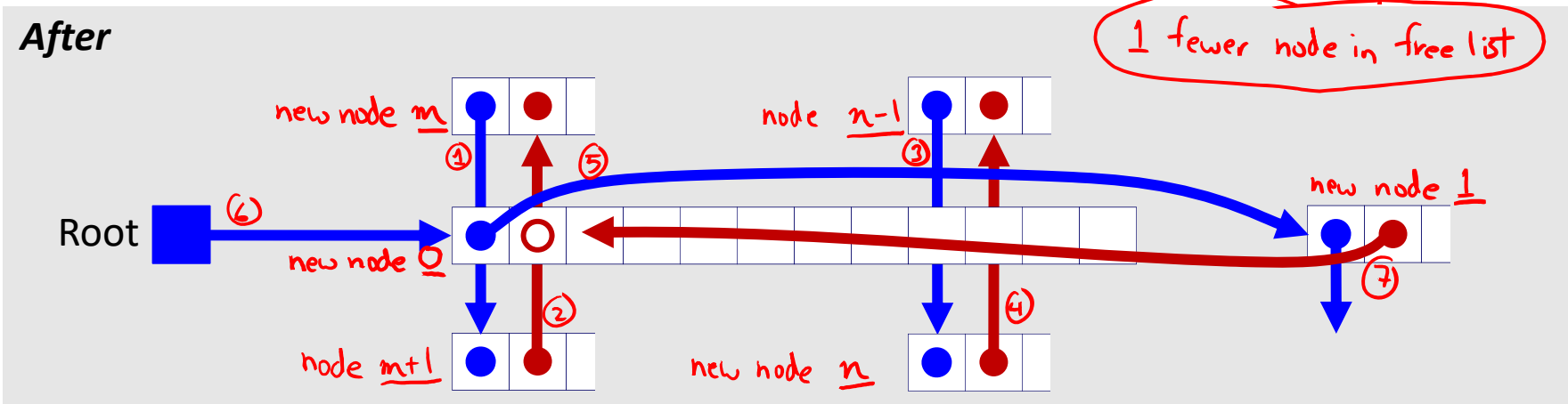


Freeing with LIFO Policy (Case 4)

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

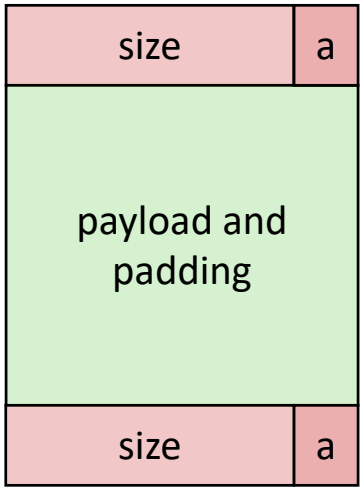


- Splice *predecessor* and *successor* blocks out of list, coalesce all 3 memory blocks, and insert the new block at the root of the list



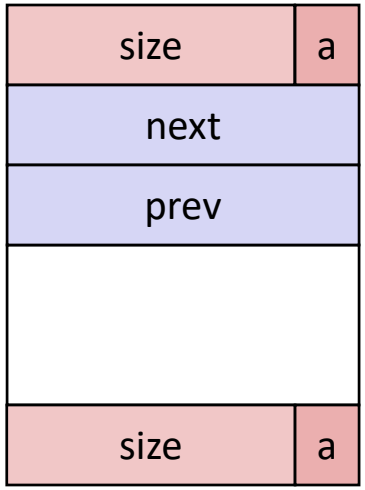
Do we always need the boundary tags?

Allocated block:



(same as implicit free list)

Free block:



- Lab 5 suggests no…

and if we can get away without using them in some cases, it saves us space on the heap!

We would only need the single bit telling us a block is allocated, not the size, in the footer of an allocated block. The footer is really only used for collalescing free blocks

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.

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

Freeing with LIFO Policy (Explicit Free List)

	Predecessor Block	Successor Block	Change in Nodes in Free List	Number of Pointers Updated
Case 1	Allocated	Allocated		
Case 2	Allocated	Free		
Case 3	Free	Allocated		
Case 4	Free	Free		