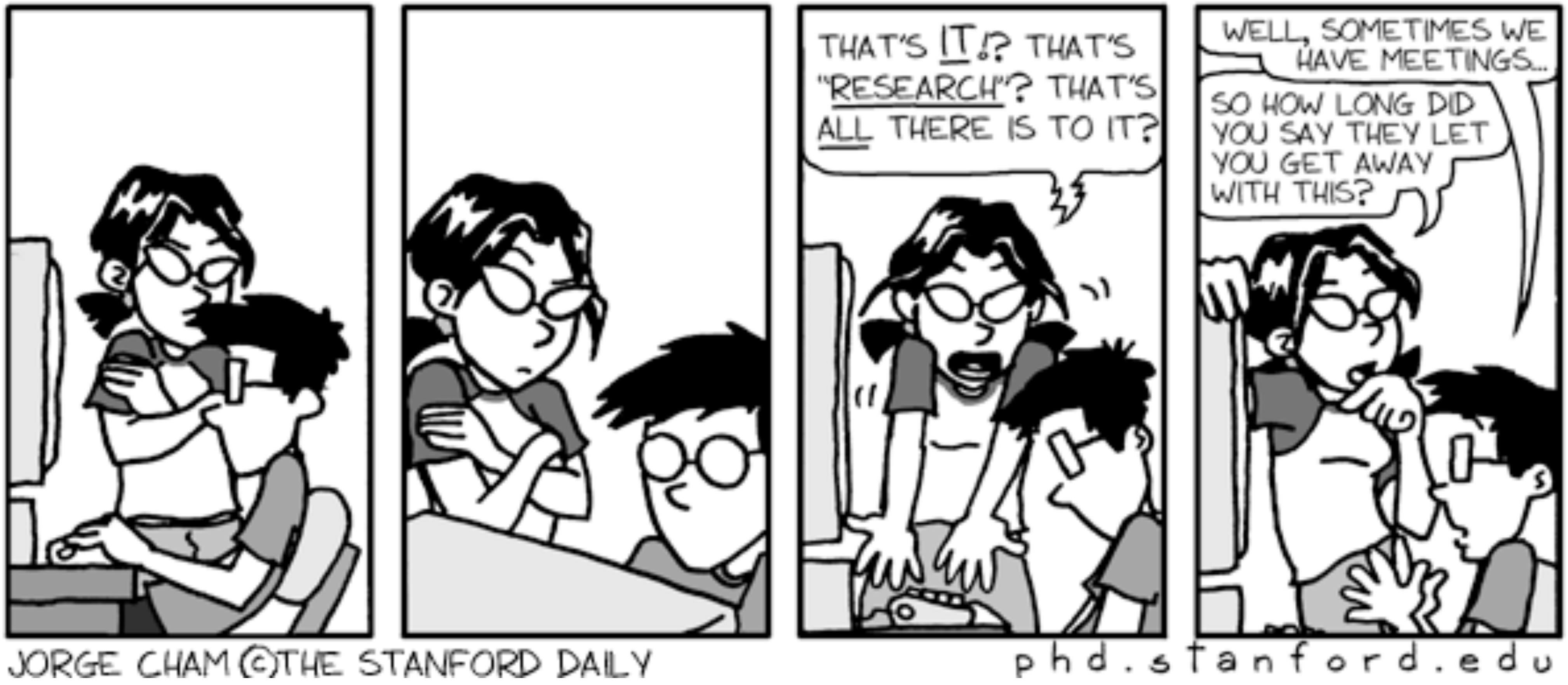
HARVARD
School of Engineering
and Applied Sciences

Dynamic memory allocation

CS61, Lecture 10

Prof. Stephen Chong

October 5, 2010


THAT'S IT⁈ THAT'S "RESEARCH"? THAT'S ALL THERE IS TO IT?
WELL, SOMETIMES WE HAVE MEETINGS...
SO HOW LONG DID YOU SAY THEY LET YOU GET AWAY WITH THIS?
JORGE CHAM ©THE STANFORD DAILY
phd.stanford.edu

Thinking about grad school in Computer Science?

Panel discussion
Thursday October 7th, 6:30pm
Maxwell Dworkin 119

CS faculty and grad students will answer your questions about grad school in Computer Science: Why to apply, how to apply, how to get in, research, reference letters, personal statement, common pitfalls, what to do during your **sophomore** and **junior** years, and more…

Pizza will be served!

Undergraduates at all levels are encouraged to attend.

Announcements

- Lab 3: Finalize groups by 8pm tonight please!
  - Come to the front of class during break if you'd like to find a group
  - Enter your group (containing either one or two people) at http://bit.ly/b8tTuW (link on website)
- Keep an eye on website cs61.seas.harvard.edu for announcements

Wall of Flame

Robert Bowden

Siddarth Chandrasekaran

Michael Chen

Edward Gan

Herman Gudjonson

Svilen Kanev

Yonatan Kogan

Brandon Liu

Lucia Mocz

Robert Nishihara

Mengqi Niu

Carl Pillot

Amy Tai

Joseph Tassarotti

Max Wang

William Weingarten

Shuang Wu

Joy Zhang

Andrew Zhou

Wenchi Zhou

Danny Zhu

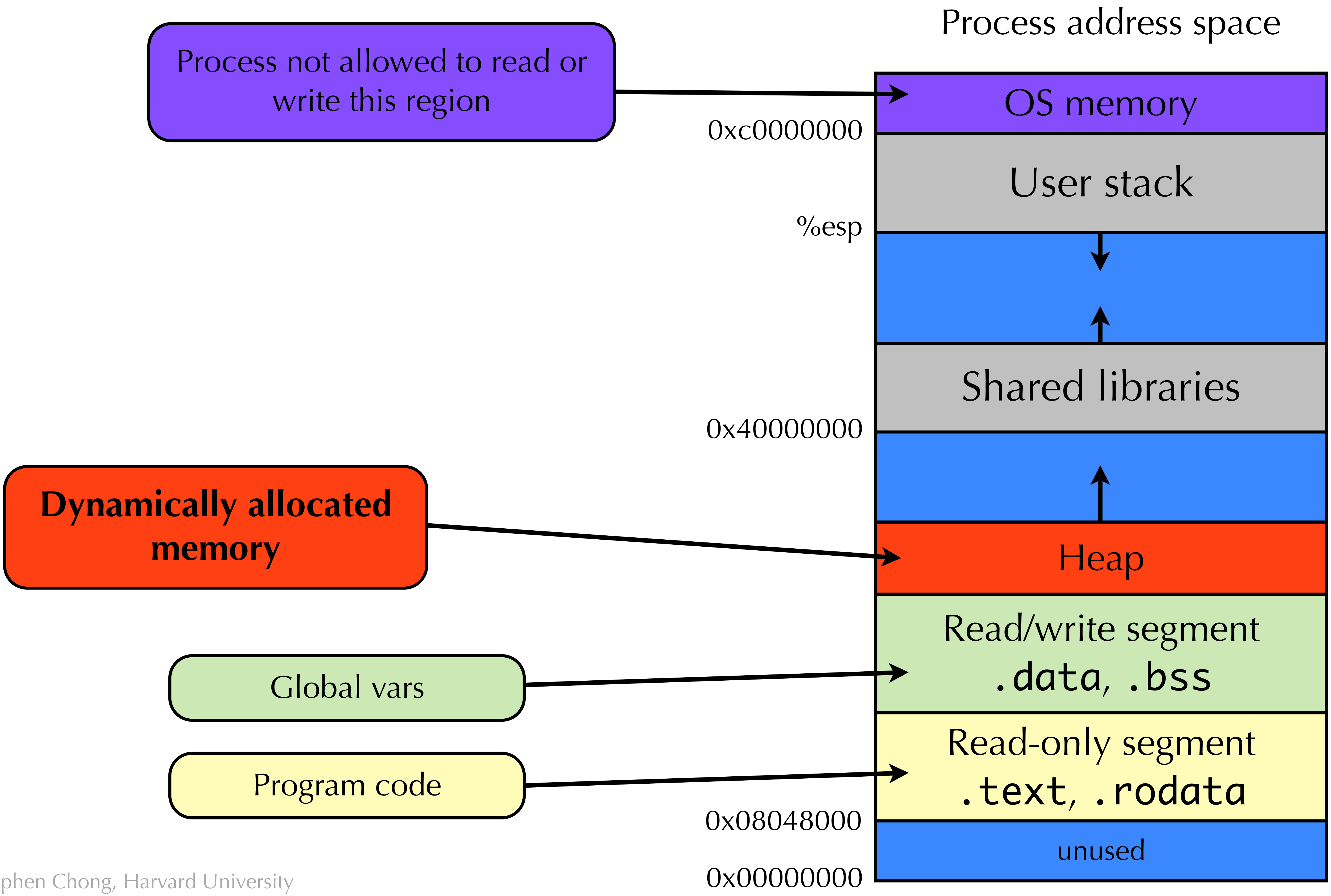
Topics for today

- Dynamic memory allocation
  - Implicit vs. explicit memory management
- Performance goals
- Fragmentation
- Free block list management
  - Implicit free list
  - Explicit free list
  - Segmented lists
  - Tradeoffs

Harsh Reality: Memory Matters!

- Memory is not unlimited!
  - It must be allocated and managed
  - Many applications are memory dominated
    - Especially those based on complex, graph algorithms
- Memory referencing bugs especially pernicious
  - Effects are distant in both time and space
- Memory performance is not uniform
  - Cache and virtual memory effects can greatly affect program performance
  - Adapting program to characteristics of memory system can lead to major speed improvements

A process's view of memory



Process address space

Process not allowed to read or write this region

OS memory

0xc0000000

User stack

%esp

Shared libraries

0x40000000

Dynamically allocated memory

Heap

Global vars

Read/write segment
`.data, .bss`

Program code

Read-only segment
`.text, .rodata`

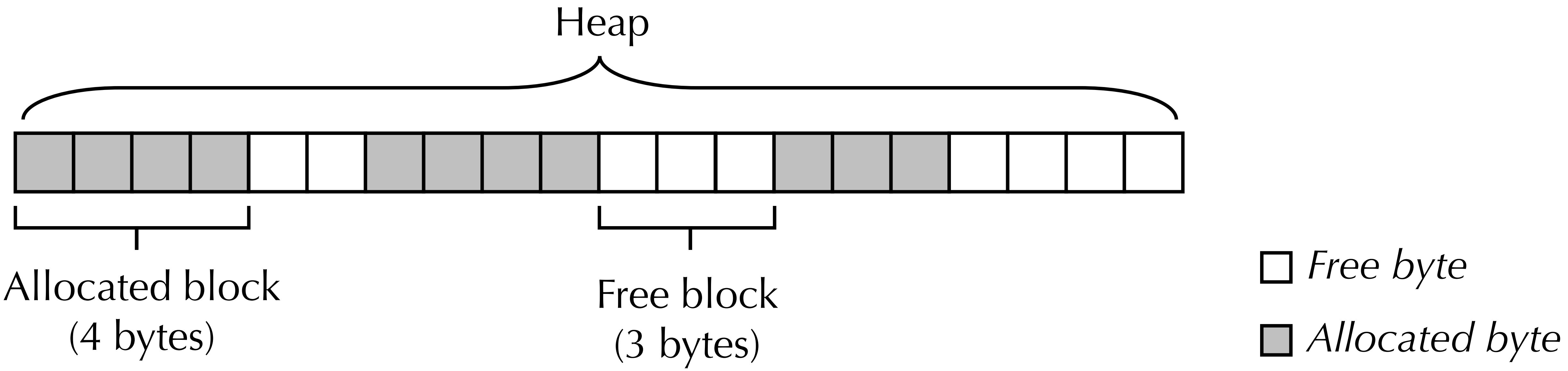
0x08048000

unused

0x00000000

The heap

- The **heap** is the region of a program's memory used for dynamic allocation.
- Program can allocate and free blocks of memory within the heap.



Heap

Allocated block
(4 bytes)

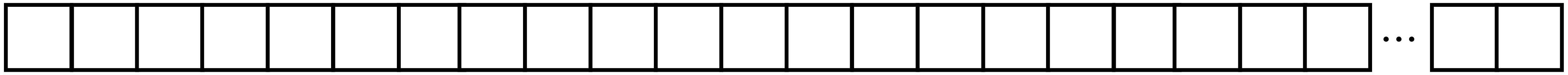
Free block
(3 bytes)

□ *Free byte*

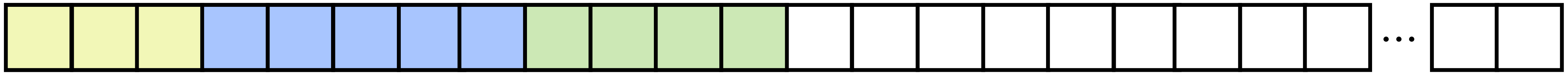
▨ *Allocated byte*

Free blocks, and allocated blocks

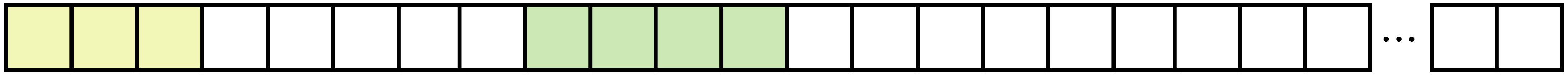
- Heap starts out as a single big “free block” of some fixed size (say a few MB)



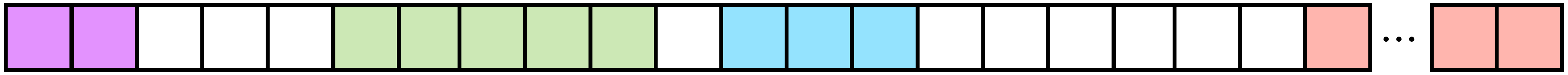
- Program may request memory, which splits up the the free space.



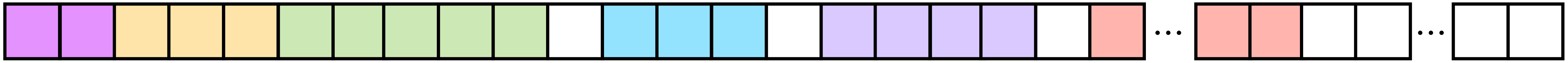
- Program may free up some memory, which increases the free space



- Over time the heap will contain a mixture of free and allocated blocks.



- Heap may need to grow in size (but typically never shrinks)
  - Program can grow the heap if it is too small to handle an allocation request.
  - On UNIX, the `sbrk()` system call is used to expand the size of the heap.



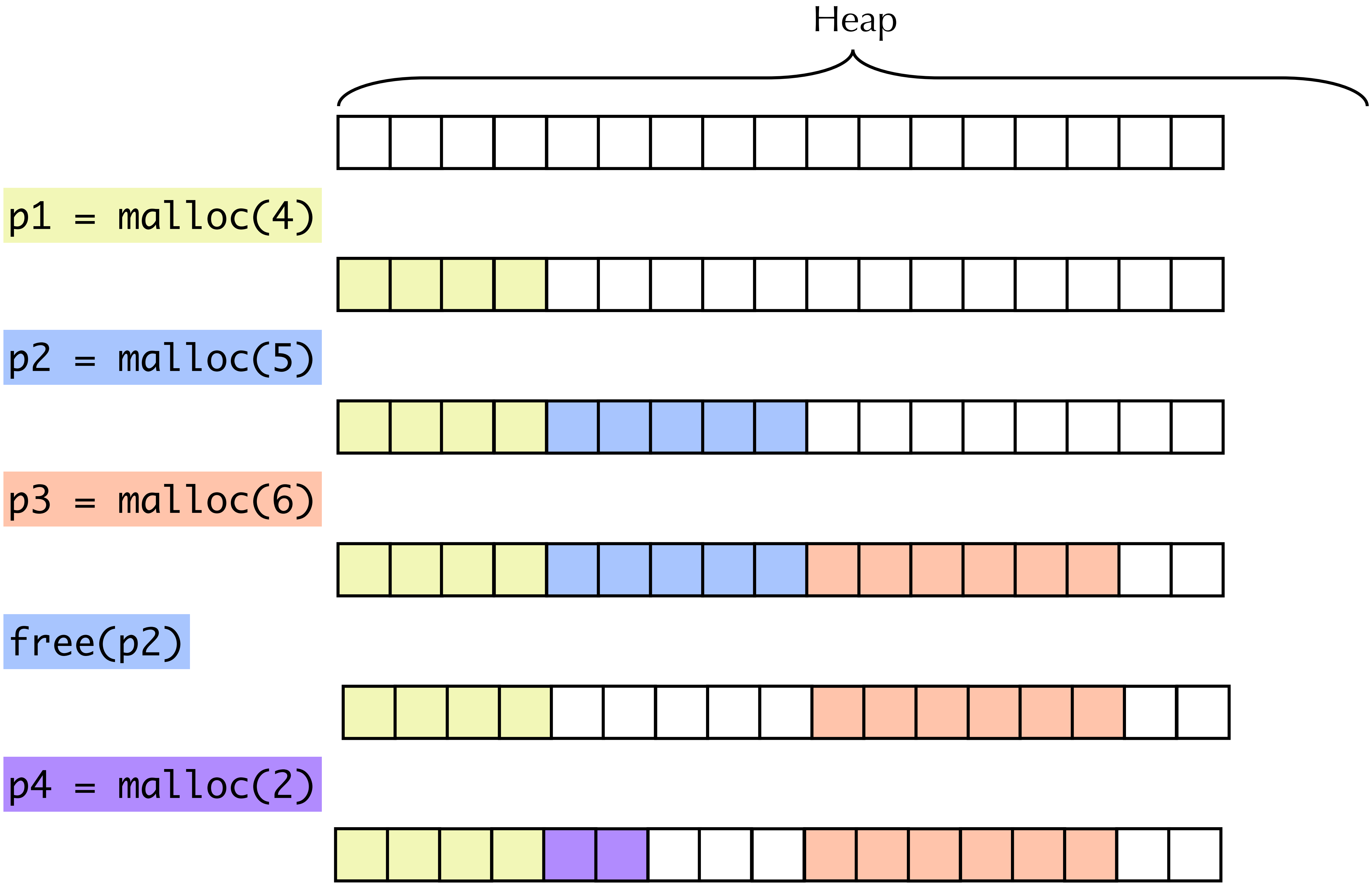
Dynamic Memory Management

- How do we decide when do expand the heap (using `sbrk()`)?
- How do we manage allocating and freeing bytes on the heap?
- There are two broad classes of memory management schemes:
- Explicit memory management
  - Application code responsible for both explicitly **allocating** and **freeing** memory.
  - Example: `malloc()` and `free()`
- Implicit memory management
  - Application code can allocate memory, but **does not explicitly free memory**
  - Rather, rely on **garbage collection** to "clean up" memory objects no longer in use
  - Used in languages like Java, Python, OCaml

Malloc Package

- `#include <stdlib.h>`
- `void *malloc(size_t size)`
  - If successful:
    - Returns a pointer to a memory block of at least `size` bytes
    - If `size == 0`, returns `NULL`
  - If unsuccessful: returns `NULL`.
- `void free(void *p)`
  - Returns the block pointed at by `p` to pool of available memory
  - `p` must come from a previous call to `malloc` or `realloc`.
- `void *realloc(void *p, size_t size)`
  - Changes size of block `p` and returns pointer to new block.
  - Contents of new block unchanged up to min of old and new size.

Allocation Examples



```
p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
p4 = malloc(2)
```

Constraints

- Application code must obey following constraints:
  - Allowed to issue arbitrary sequence of allocation and free requests
  - Free requests must correspond to an allocated block
- Memory management code must obey following constraints:
  - Can't control number or size of requested blocks
  - Must respond immediately to all allocation requests
    - i.e., can't reorder or buffer requests
  - Must allocate blocks from free memory
    - i.e., can only place allocated blocks in free memory
  - Must align blocks so they satisfy all alignment requirements
  - Can't mess around with allocated memory
    - Can only manipulate and modify free memory
    - Can't move the allocated blocks once they are allocated (i.e., compaction is not allowed)

Topics for today

- Dynamic memory allocation
  - Implicit vs. explicit memory management
- Performance goals
- Fragmentation
- Free block list management
  - Implicit free list
  - Explicit free list
  - Segmented lists
  - Tradeoffs

What do we want?

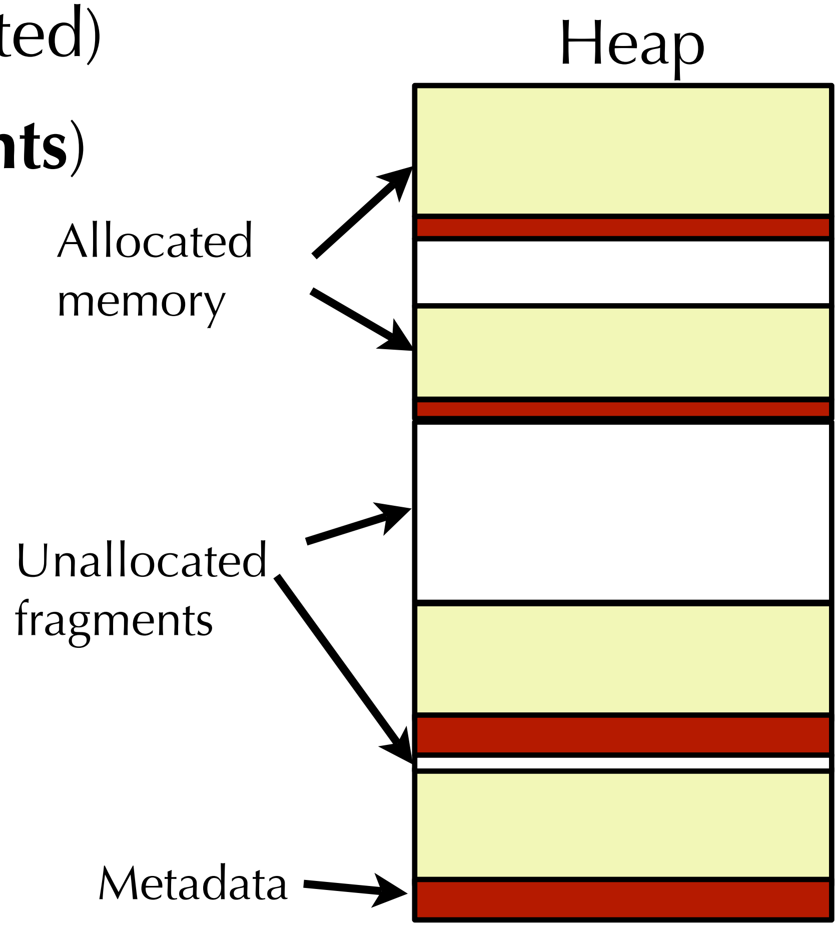
- Want our memory management to be:
  - Fast
    - Minimize overhead of allocation and deallocation operations.
  - Efficient
    - Don’t waste memory space

Performance Goals: Allocation overhead

- Want our memory allocator to be fast!
  - Minimize the overhead of both allocation and deallocation operations.
- One useful metric is **throughput**:
  - Given a series of allocate or free requests
  - Maximize the number of completed requests per unit time
- Example:
  - 5,000 malloc calls and 5,000 free calls in 10 seconds
  - Throughput is 1,000 operations/second.
- Note that a fast allocator may not be efficient in terms of memory utilization.
  - Faster allocators tend to be "sloppier"
  - To do the best job of space utilization, operations must take more time.
  - Trick is to balance these two conflicting goals.

Performance Goals: Memory Utilization

- Allocators rarely do a perfect job of managing memory.
  - Usually there is some "waste" involved in the process.
- Examples of waste...
  - Extra metadata or internal structures used by the allocator itself (example: Keeping track of where free memory is located)
  - Chunks of heap memory that are unallocated (**fragments**)
- We define **memory utilization** as...
  - The **total amount of memory allocated to the application** divided by the **total heap size**
- Ideally, we'd like utilization to be to 100%
  - In practice this is not possible, but want to be as close as possible



Conflicting performance goals

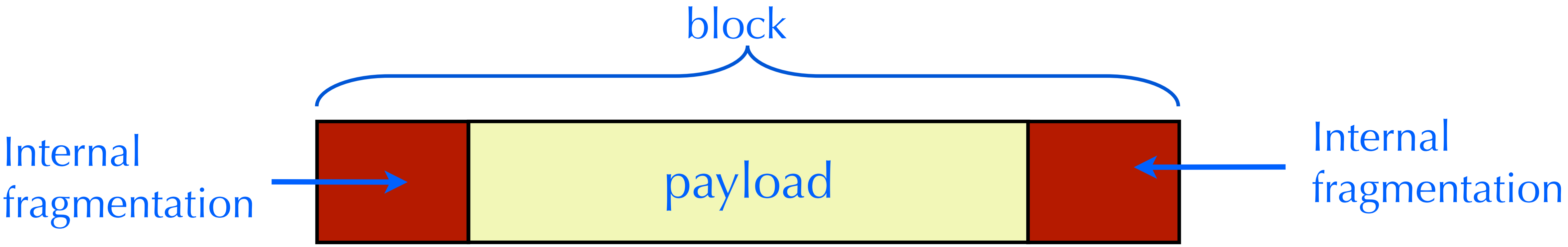
- High throughput and good utilization are difficult to achieve simultaneously.
- A fast allocator may not be efficient in terms of memory utilization.
  - Faster allocators tend to be “sloppier” with their memory usage.
- Likewise, a space-efficient allocator may not be very fast
  - To keep track of memory waste (i.e., tracking fragments), the allocation operations generally take longer to run.
- Trick is to balance these two conflicting goals.

Topics for today

- Dynamic memory allocation
    - Implicit vs. explicit memory management
- Performance goals
- Fragmentation
- Free block list management
    - Implicit free list
    - Explicit free list
    - Segmented lists
    - Tradeoffs
- Alignment

Internal Fragmentation

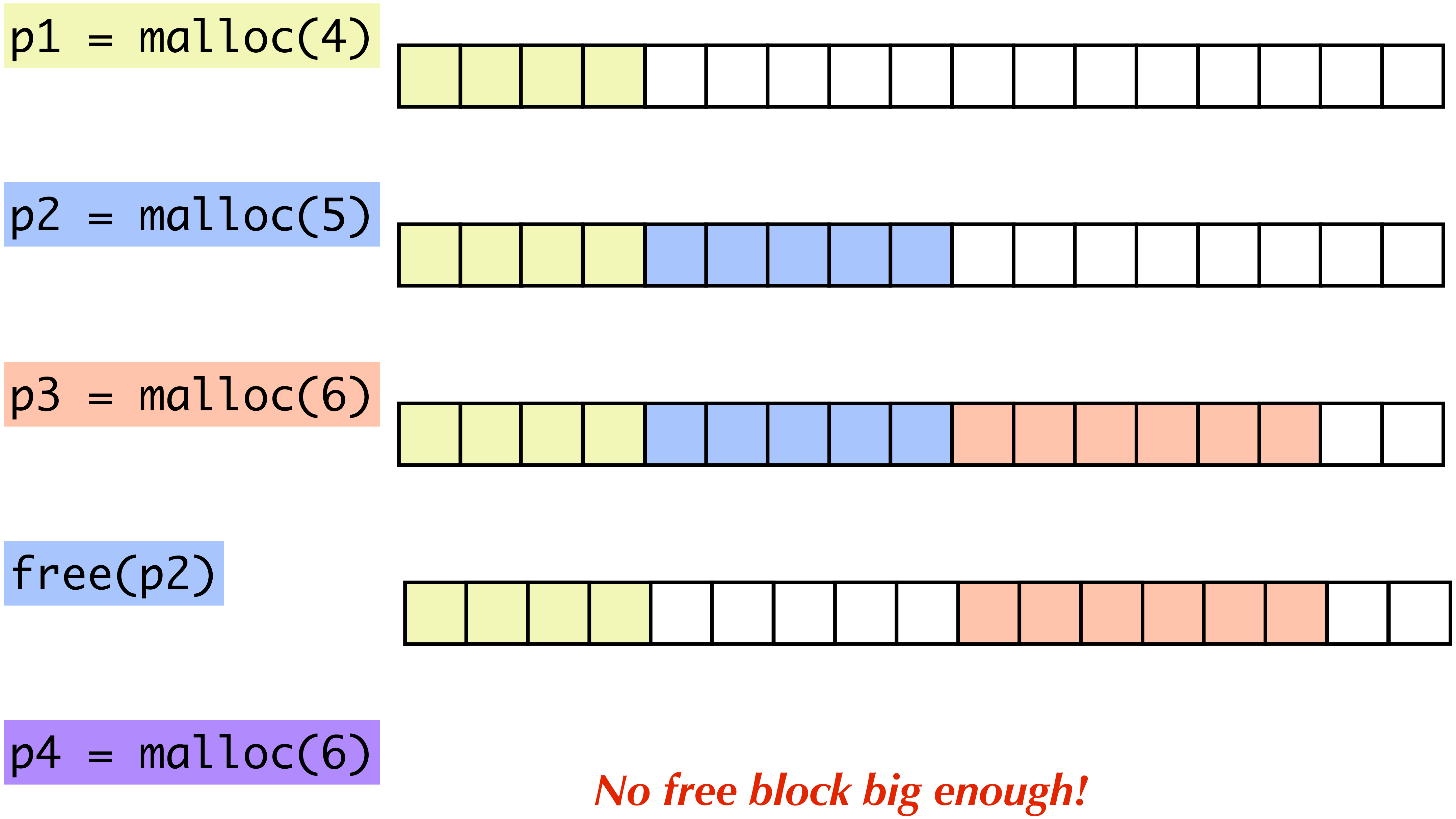
- Poor memory utilization caused by **fragmentation**.
  - Comes in two forms: **internal** and **external** fragmentation
- **Internal fragmentation**
  - Internal fragmentation is the difference between block size and payload size.



  - Caused by overhead of maintaining heap data structures, padding for alignment purposes, or the policy used by the memory allocator
  - Example: Say the allocator always “rounds up” to next highest power of 2 when allocating blocks.
    - So `malloc(1025)` will actually allocate 2048 bytes of heap space!

External Fragmentation

- Occurs when there is enough aggregate heap memory, but no single free block is large enough to satisfy a given request.



```
p1 = malloc(4)
```

```
p2 = malloc(5)
```

```
p3 = malloc(6)
```

```
free(p2)
```

```
p4 = malloc(6)
```

No free block big enough!

Topics for today

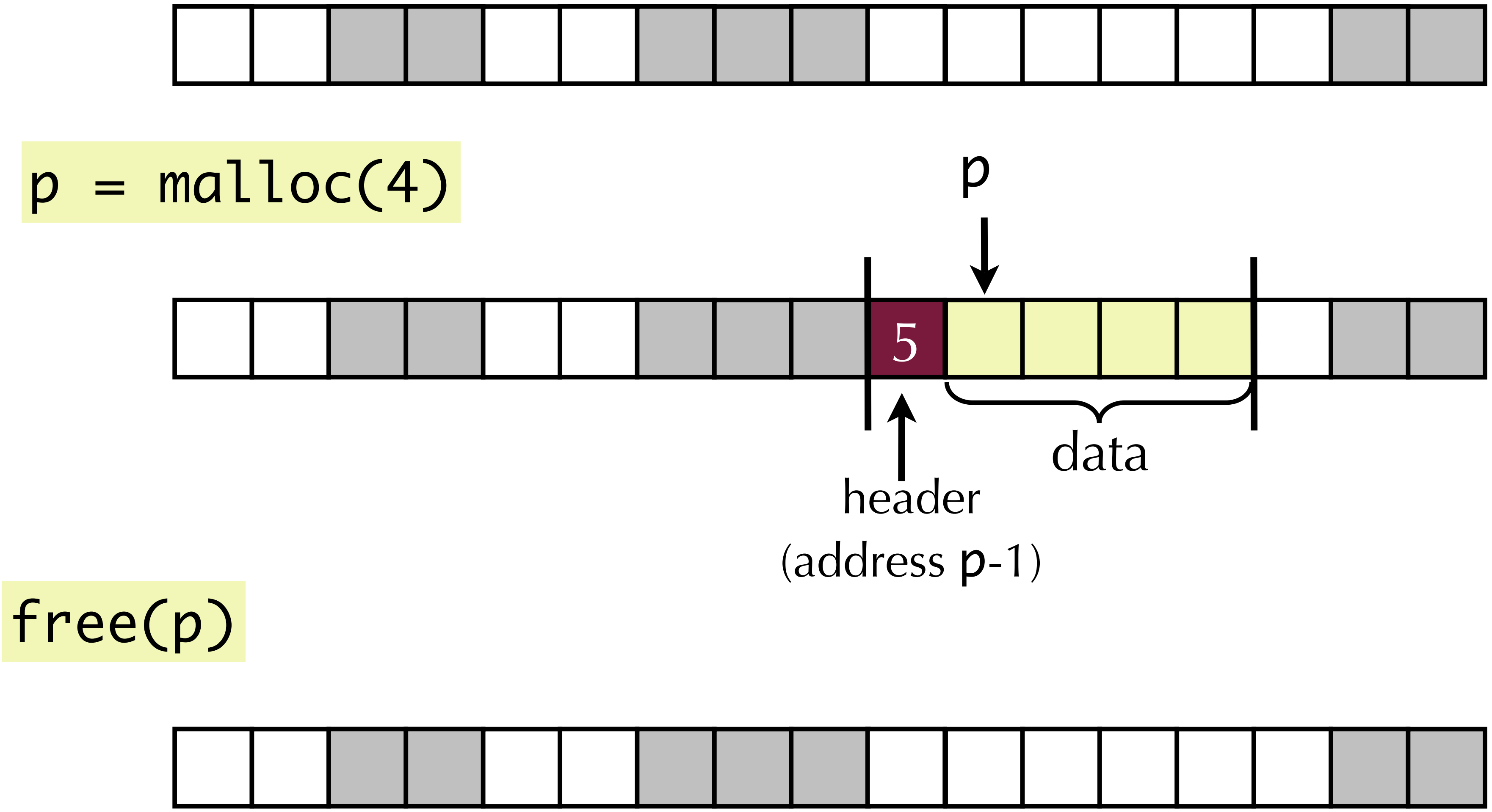
- Dynamic memory allocation
  - Implicit vs. explicit memory management
- Performance goals
- Fragmentation
- Free block list management
  - Implicit free list
  - Explicit free list
  - Segmented lists
  - Tradeoffs

Free block list management

- How do we know how much memory to free just given a pointer?
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a memory block that is smaller than the free block it is placed in?
- How do we pick which free block to use for allocation?

Knowing how much to free

- Standard method
  - Keep the length of a block in a header preceding the block.
  - Requires an extra word for every allocated block



```
p = malloc(4)
```

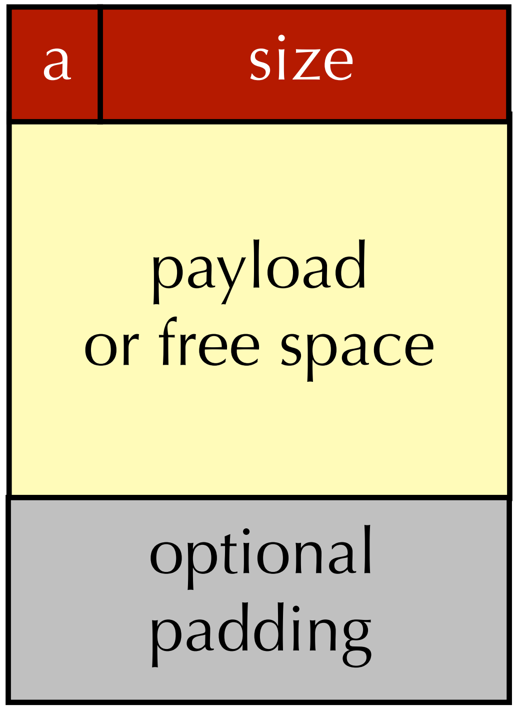
```
free(p)
```

Keeping Track of Free Blocks

- One of the biggest jobs of an allocator is knowing where the free memory is.
  - Affects throughput and utilization
- Many approaches to free block management.
  - Today, we will talk about three techniques
    - Implicit free lists
    - Explicit free lists
    - Segmented free lists
  - There are other approaches

Implicit free list

- Idea: Each block contains a **header** with some extra information.
    - **Allocated bit** indicates whether block is allocated or free.
    - **Size field** indicates entire size of block (including the header)
    - Trick: Allocation bit is just the high-order bit of the size word
- For this lecture, assume header size is 1 byte.
    - Makes later pictures easier to understand.
    - This means the block size is only 7 bits, so max. block size is 127 bytes (2^7-1).
    - Clearly a real implementation would want to use a larger header (e.g., 4 bytes).

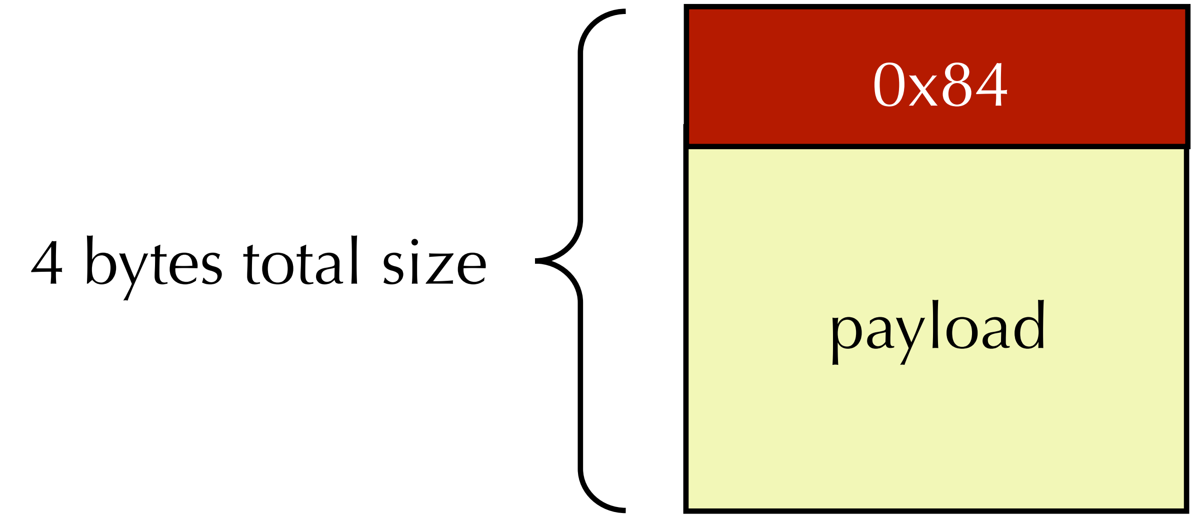


a = 1: block is allocated
a = 0: block is free

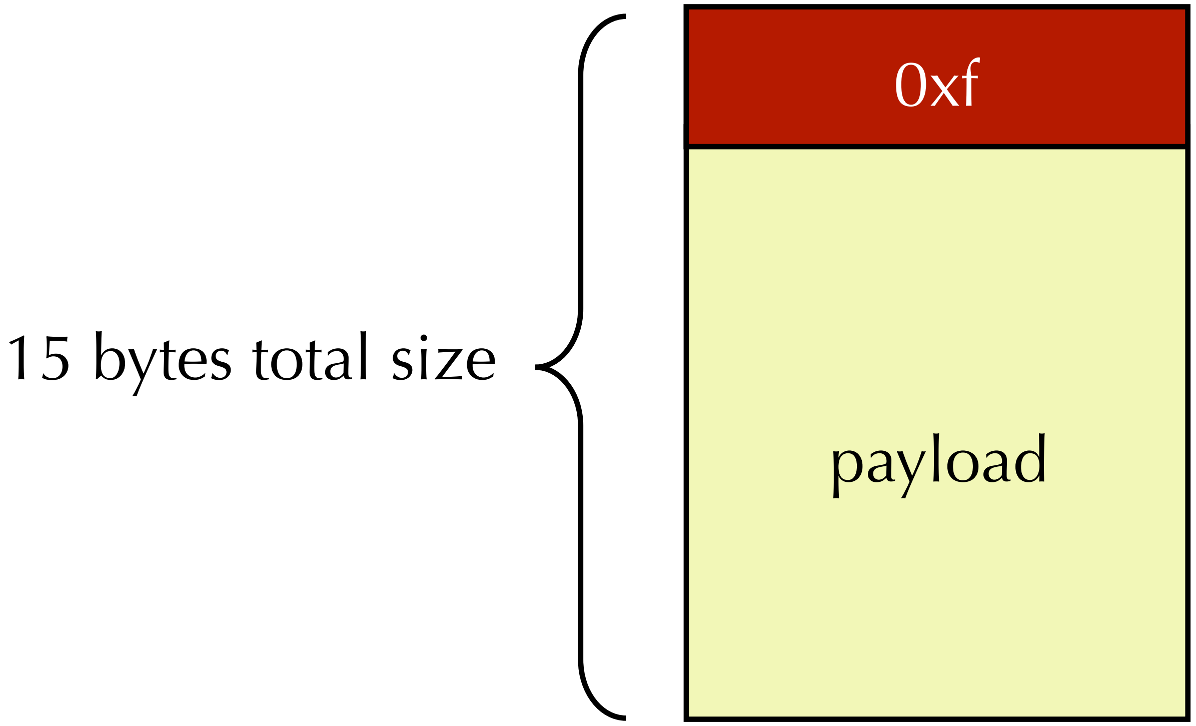
size: block size

payload: application data

Examples

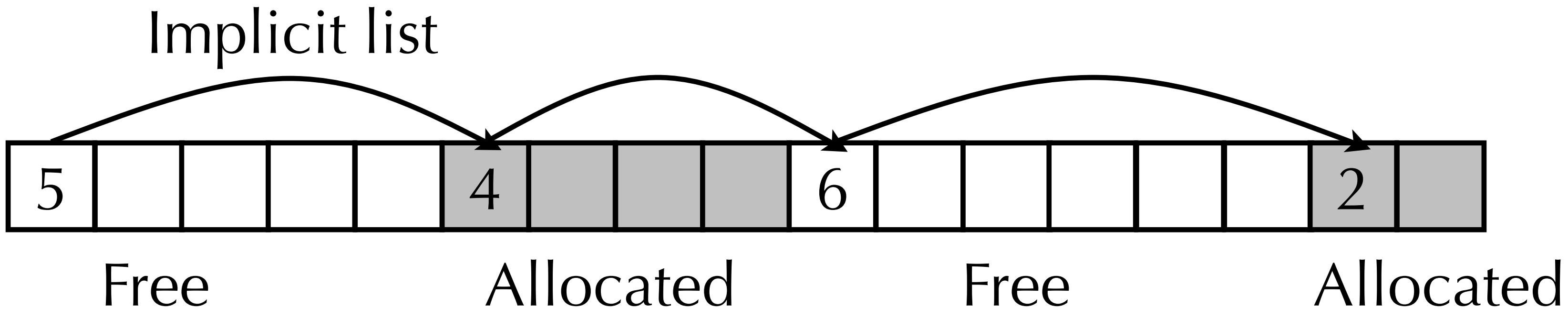


0x84 in binary: 1000 0100
allocated = 1
size = 0x4 = 4 bytes



0xf in binary: 0000 1111
allocated = 0
size = 0xf = 15 bytes

Implicit free list



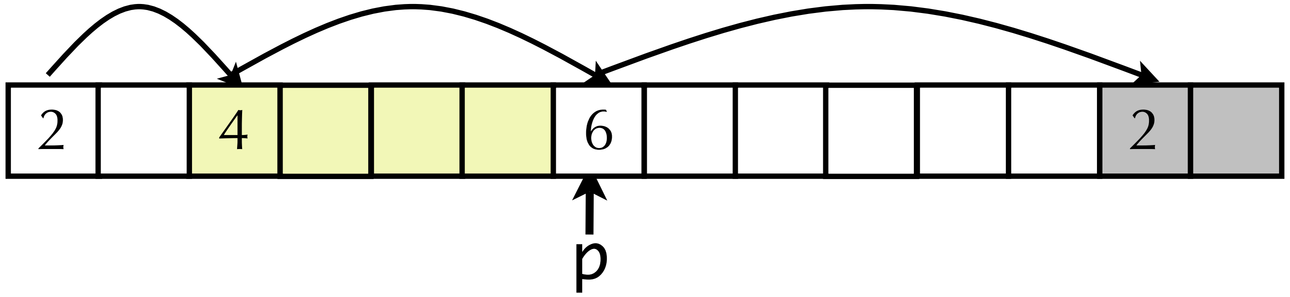
- No **explicit** structure tracking location of free/allocated blocks.
  - Rather, the size word (and allocated bit) in each block form an **implicit** "block list"
- How do we find a free block in the heap?
  - Start scanning from the beginning of the heap.
  - Traverse each block until (a) we find a free block and (b) the block is large enough to handle the request.
  - This is called the **first fit** strategy.

Implicit List: Finding a Free Block

- **First fit** strategy:
  - Search list from beginning, choose first free block that fits
  - Can take linear time in total number of blocks (allocated and free)
  - In practice it can cause "splinters" at beginning of list
- **Next fit** strategy:
  - Like first-fit, but search list from location of end of previous search
  - Research suggests that fragmentation is worse than first-fit
- **Best fit** strategy:
  - Search the list, choose the free block with the closest size that fits
  - Keeps fragments small — usually helps fragmentation
  - Runs slower than first- or next-fit, since the entire list must be searched each time

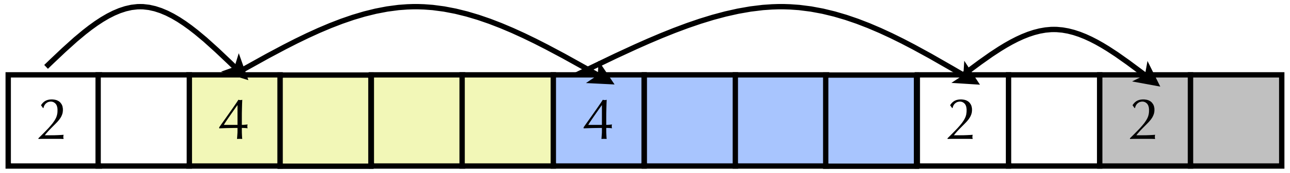
Implicit List: Allocating in Free Block

- Splitting free blocks
  - Since allocated space might be smaller than free space, we may need to **split** the free block that we're allocating within
  - E.g., `malloc(3)`



```
void addblock(ptr_t p, int len) {
  int oldsize = *p;              // Get old size of free block
  *p = len | 0x80;               // Set new size + alloc bit
  if (len < oldsize)
    *(p+len) = oldsize - len;    // Set size of remaining
}                                //   part of free block
```

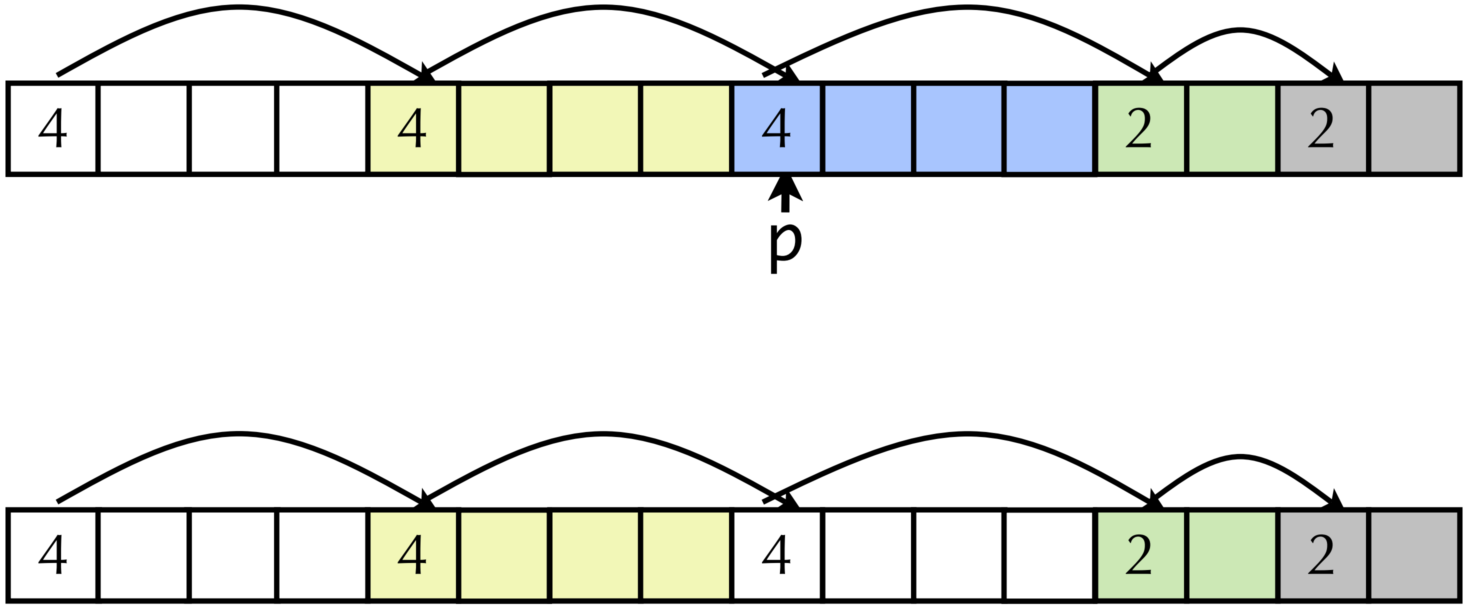
```
addblock(p, 4)
```



Implicit List: Freeing a Block

- Simplest implementation:
  - Simply clear the allocated bit in the header

```
/* Here, p points to the block header. */
/* This sets the high-order bit to 0. */
void free_block(ptr_t p) { *p = *p & ~0x80; }
```

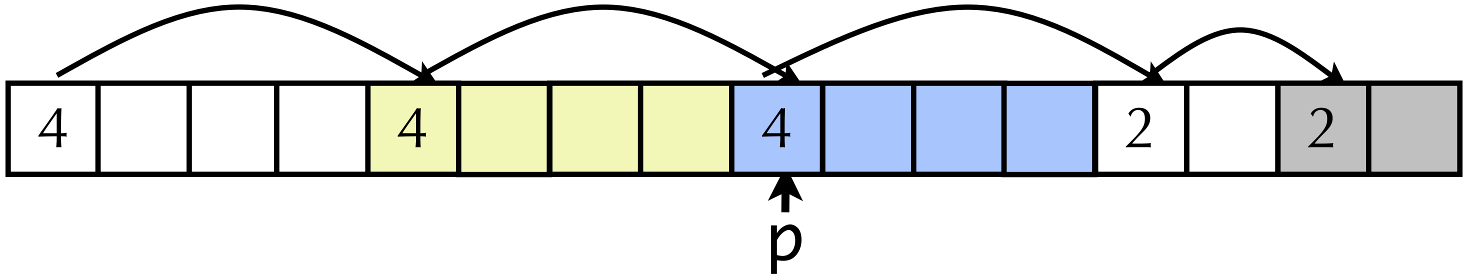


Implicit List: Freeing a Block

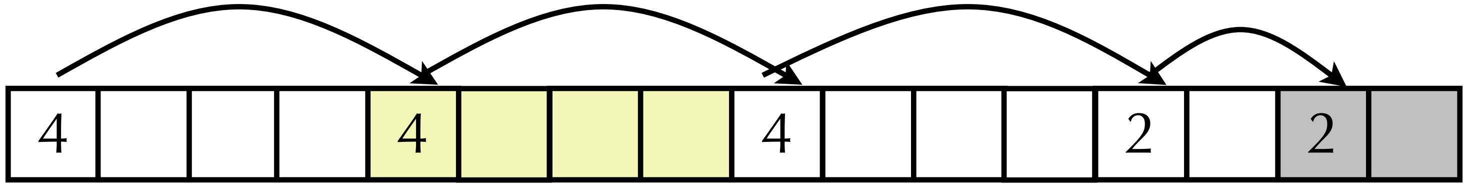
- Simplest implementation:
  - Simply clear the allocated bit in the header

```
/* Here, p points to the block header. */
/* This sets the high-order bit to 0. */
void free_block(ptr_t p) { *p = *p & ~0x80; }
```

  - But can lead to "false fragmentation"



`free(p+1)`



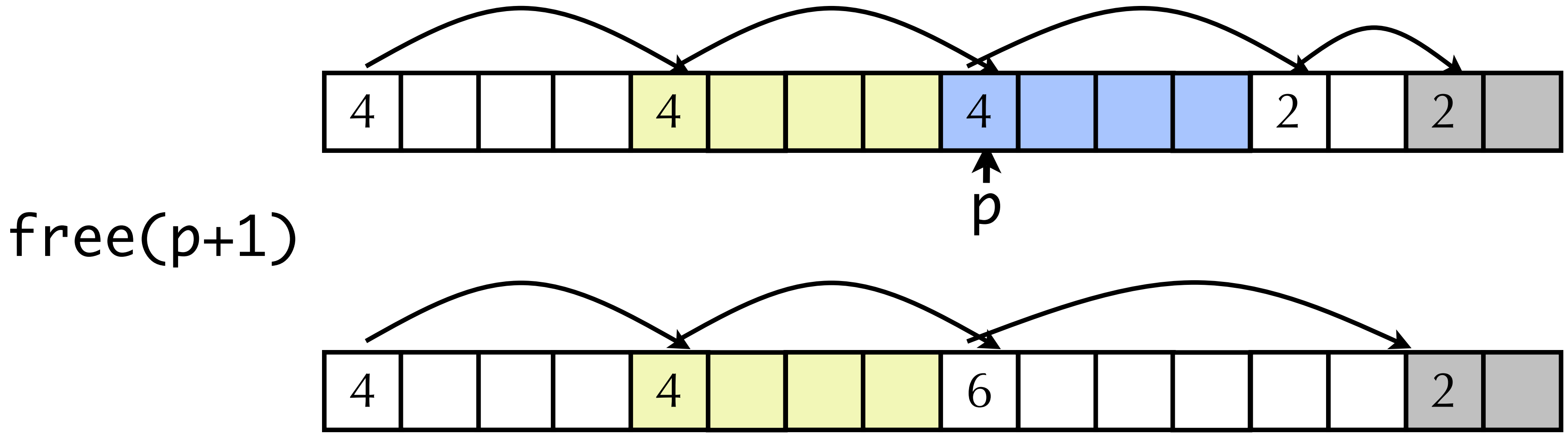
`malloc(5)`

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

Implicit List: Coalescing

- **Coalesce** with adjacent block(s) if they are free

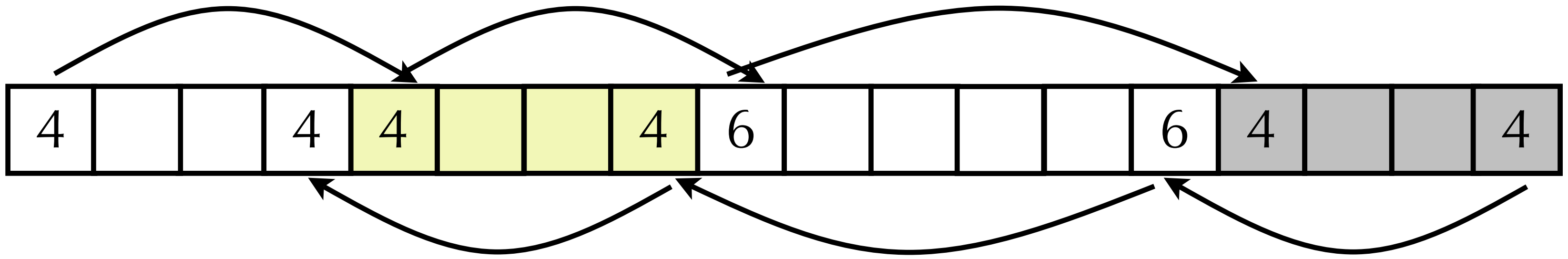
```
void free_block(ptr_t p) {
    *p = *p & ~0x80;       // Clear allocated bit
    next = p + *p;         // Find next block
    if ((*next & 0x80) == 0)
      *p = *p + *next;     // Add to this block if
}                          //    next is also free
```



- This is coalescing with the next free block.
- How would we coalesce with the previous free block?

Implicit List: Bidirectional Coalescing

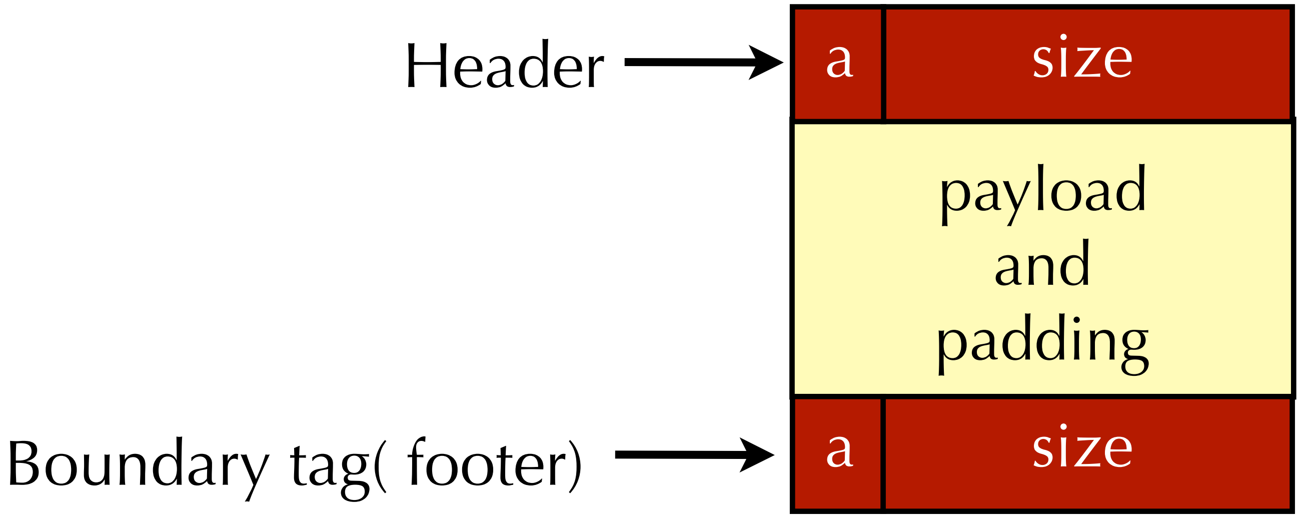
- **Boundary tags** [Knuth73]
  - Also maintain the size/allocated word at end of free blocks (a footer)



"Pointers" in headers

"Pointers" in footers

  - Allows us to traverse the "block list" backwards, but requires extra space


Header

Boundary tag(footer)

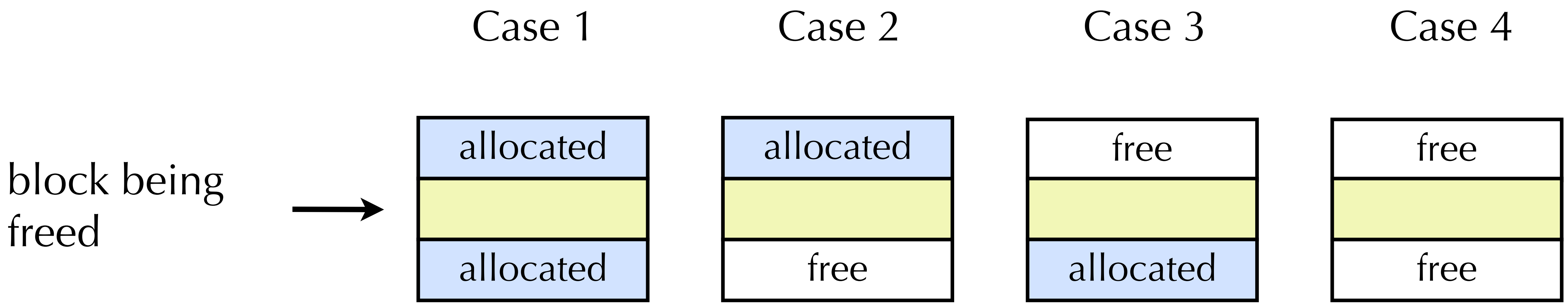
a = 1: block is allocated
a = 0: block is free

size: total block size

payload: application data

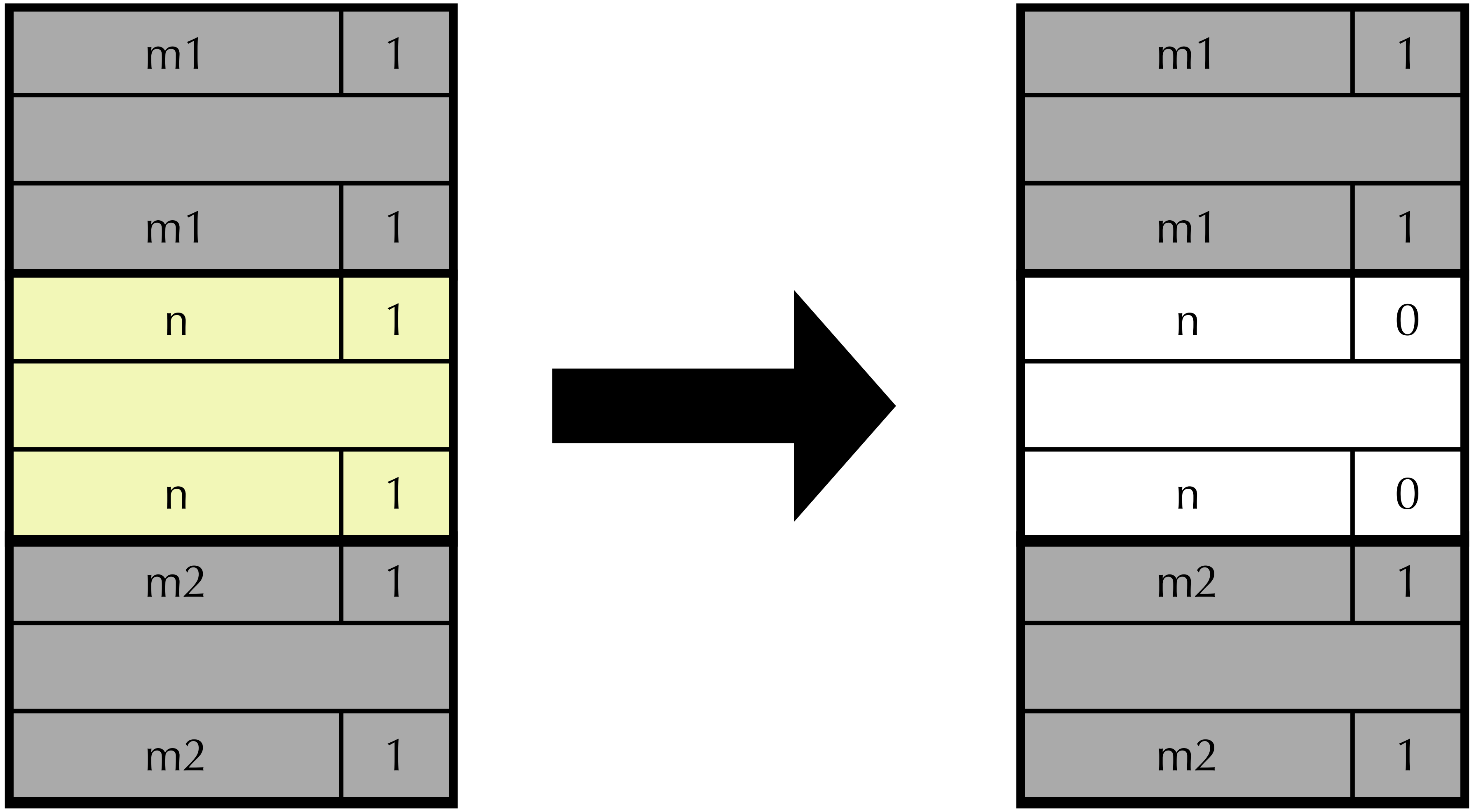
  - Important and general technique!

Constant Time Coalescing

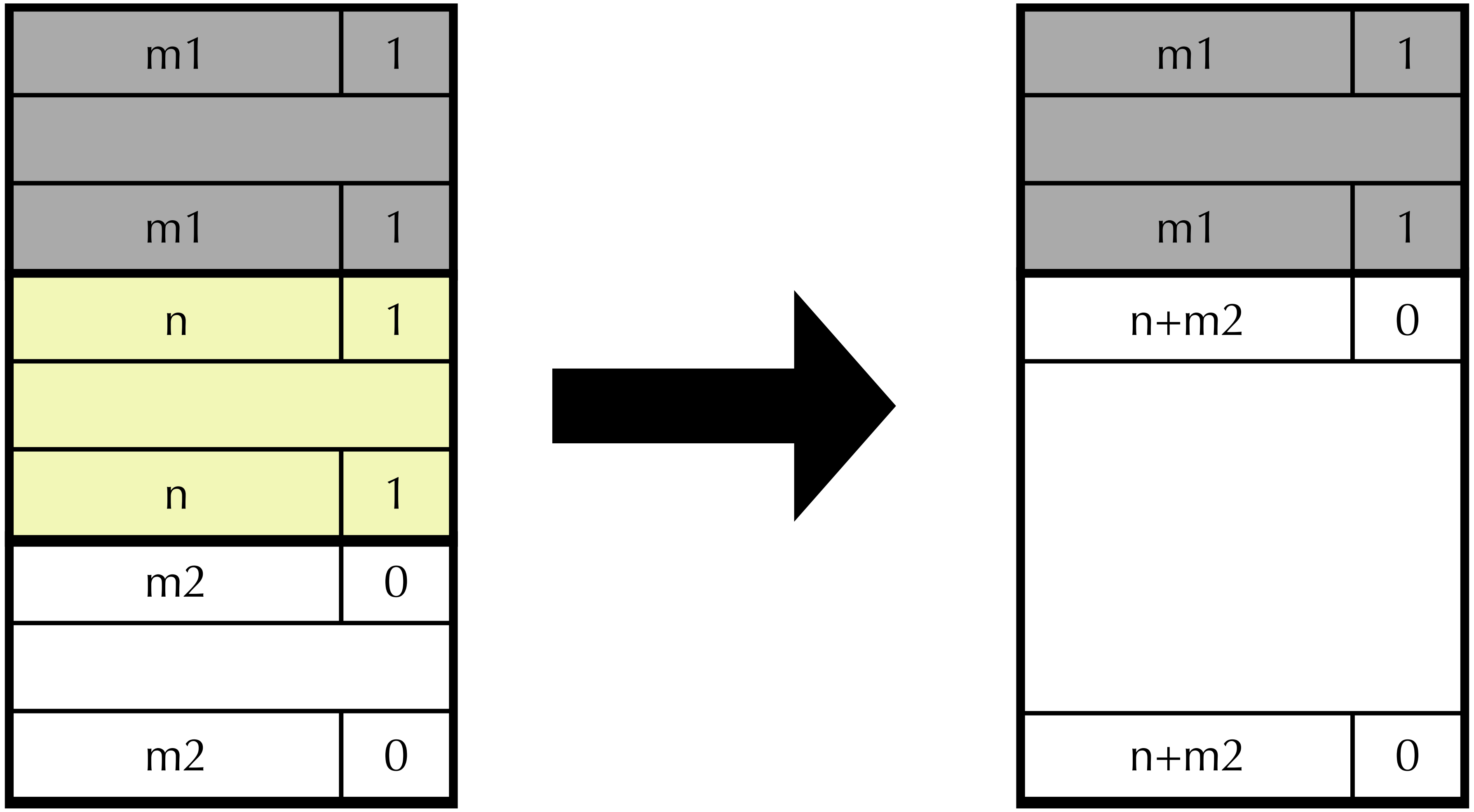


	Case 1	Case 2	Case 3	Case 4
	allocated	allocated	free	free
block being freed →				
	allocated	free	allocated	free

Constant Time Coalescing (Case 1)



Constant Time Coalescing (Case 2)



m1 1

m1 1

n 1

n 1

m2 0

m2 0

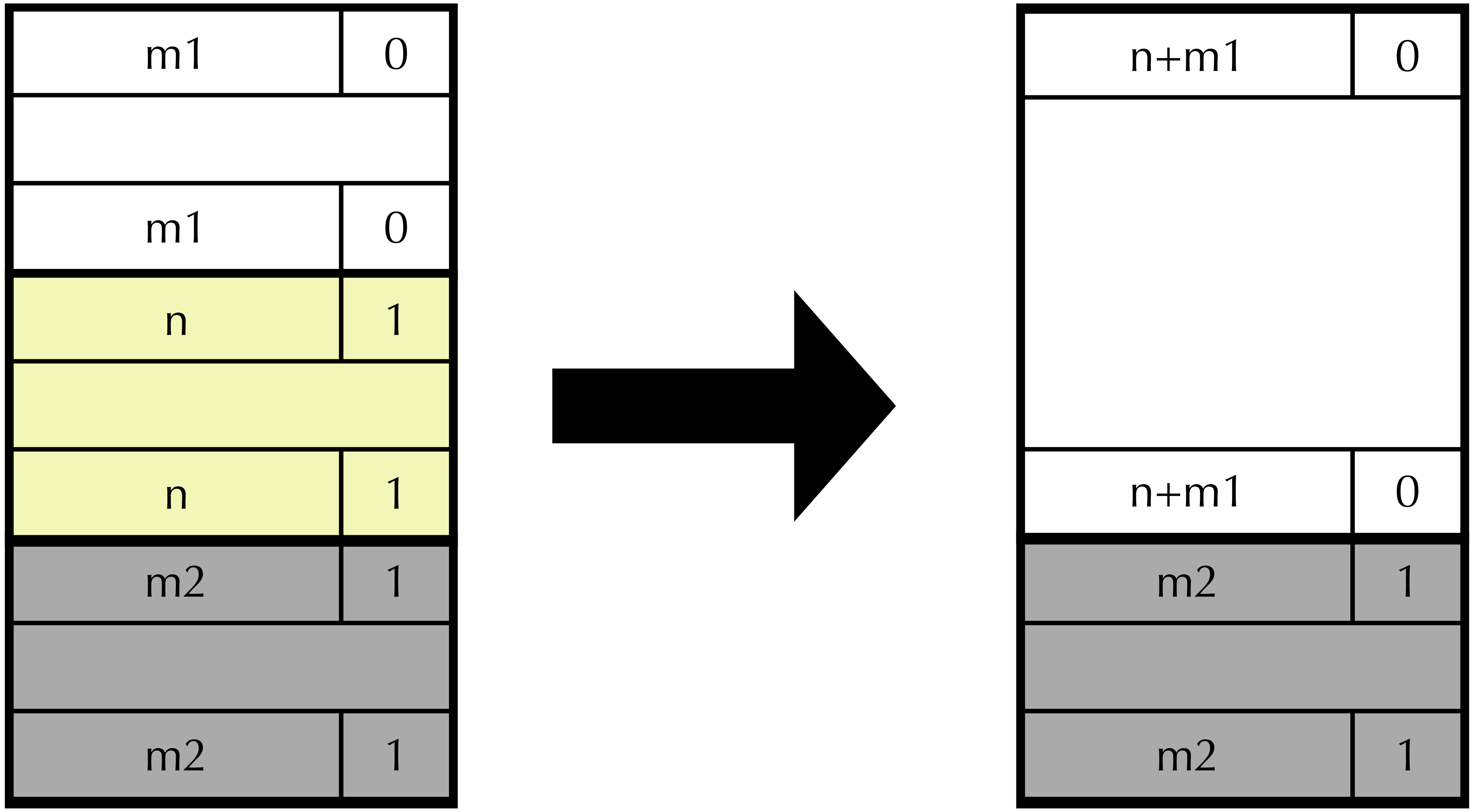
m1 1

m1 1

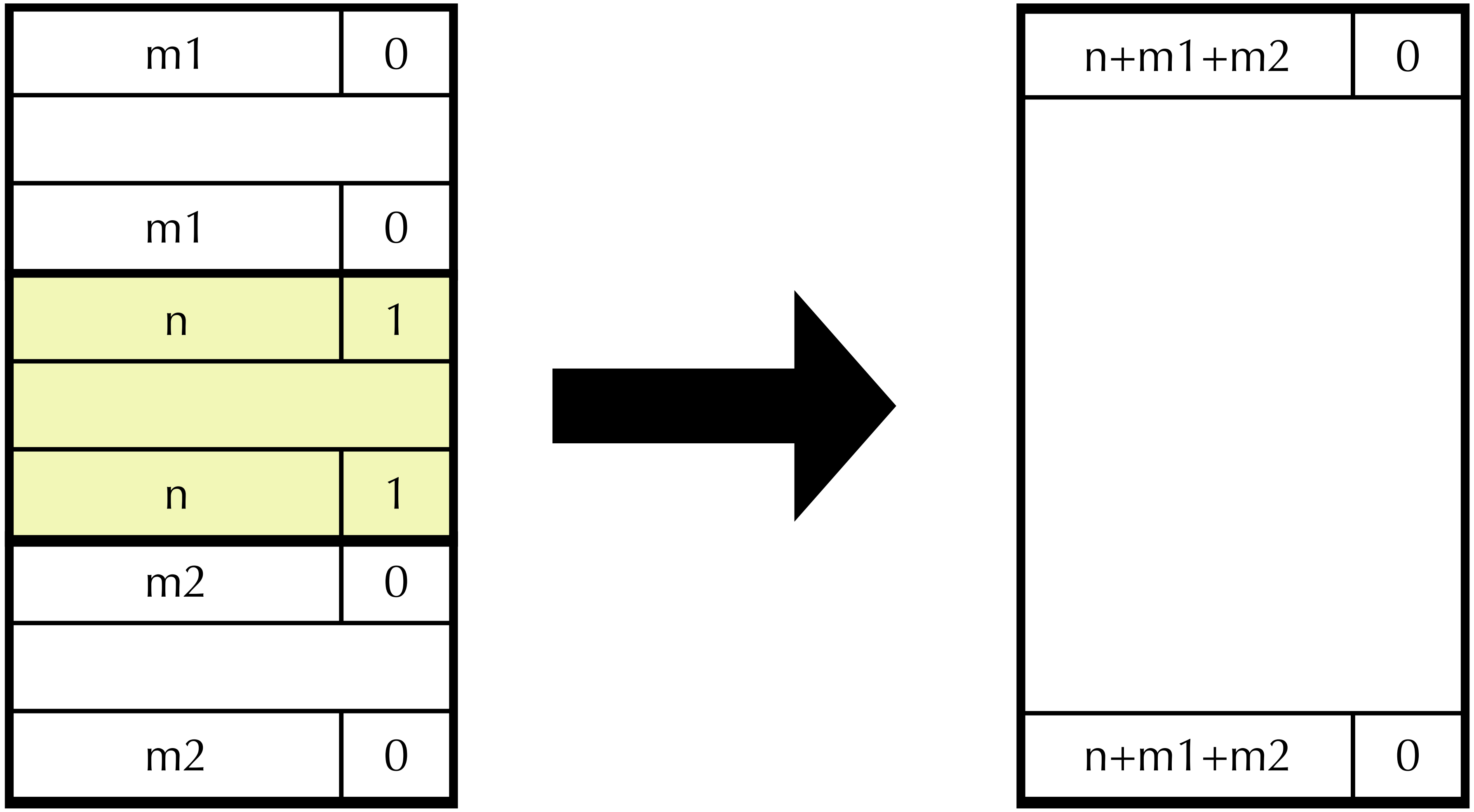
n+m2 0

n+m2 0

Constant Time Coalescing (Case 3)



Constant Time Coalescing (Case 4)



Implicit Lists: Summary

- Implementation: Very simple.
- Allocation cost: Linear time worst case
- Free cost: Constant time, even with coalescing
- Memory usage: Depends on placement policy
    - First fit, next fit, or best fit
- Not used in practice for `malloc/free` because of linear time allocation.

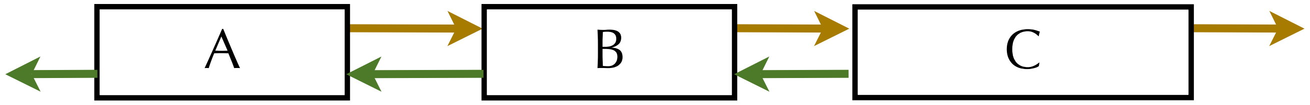
- The concepts of splitting and boundary tag coalescing are general to *all* allocators.

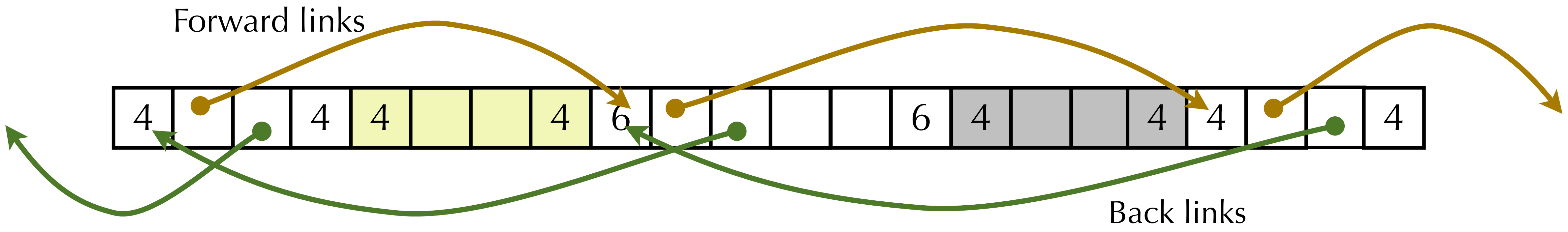
Topics for today

- Dynamic memory allocation
    - Implicit vs. explicit memory management
- Performance goals
- Fragmentation
- Free block list management
    - Implicit free list
    - Explicit free list
    - Segmented lists
    - Tradeoffs

Explicit Free Lists

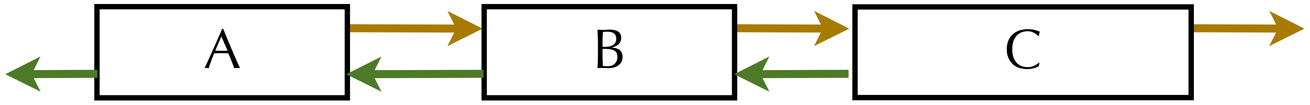
- Use an **explicit** data structure to track the free blocks
  - Just a doubly-linked list.
  - No pointers to or from allocated blocks: Can put the pointers into the payload!
  - Still need boundary tags, in order to perform free block coalescing.

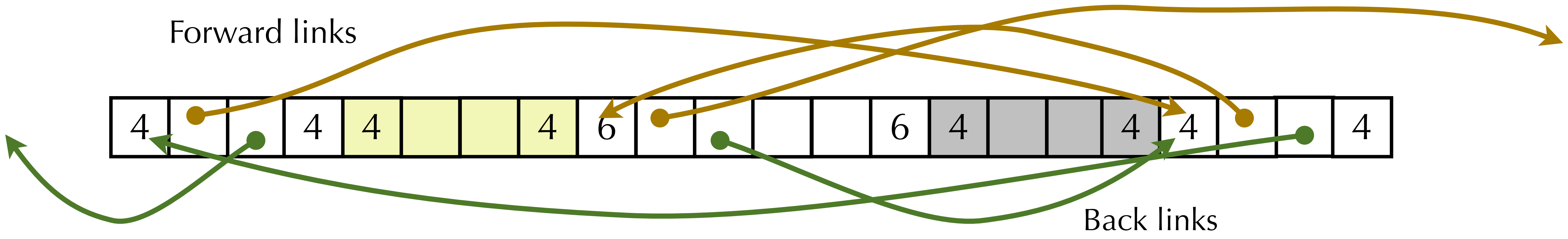




Explicit Free Lists

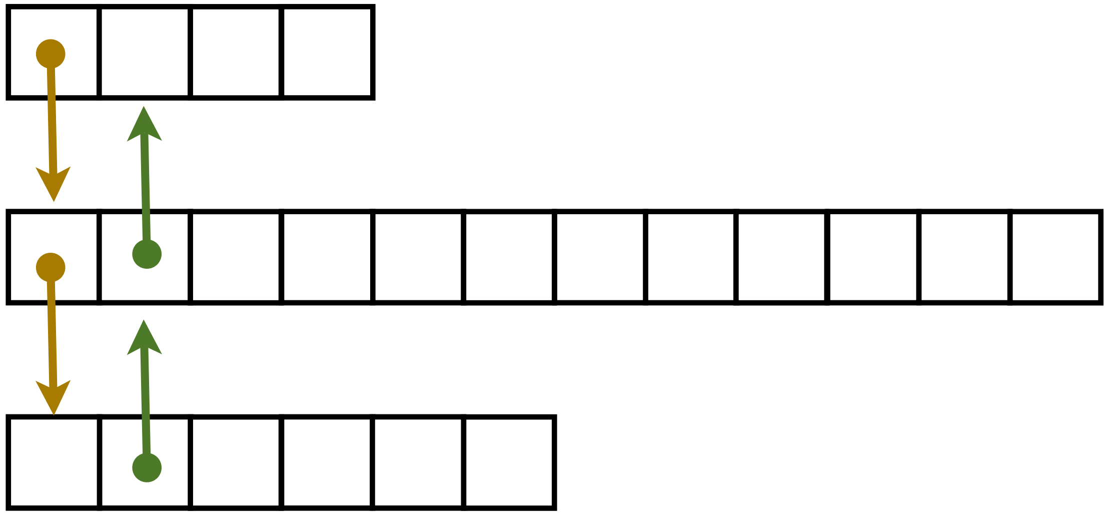
- Note that free blocks need not be linked in the same order
  they appear in memory!
  - Free blocks can be chained together in any order.





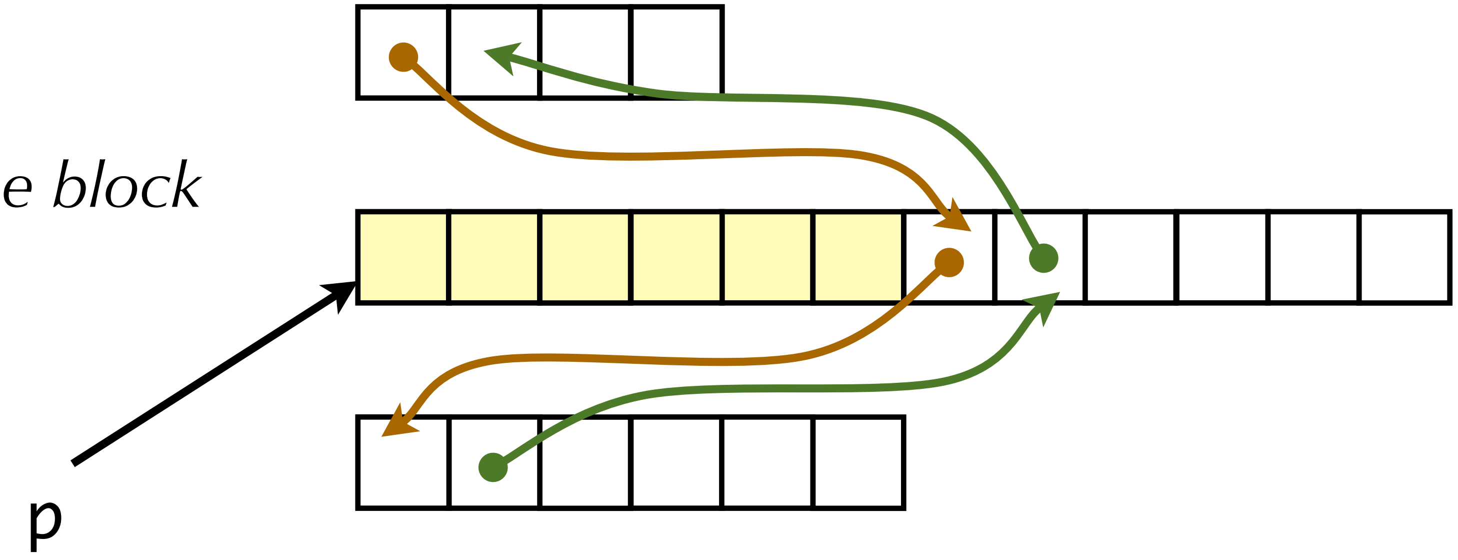
Allocation using an explicit free list

Before:



`p = malloc(size);`

After:



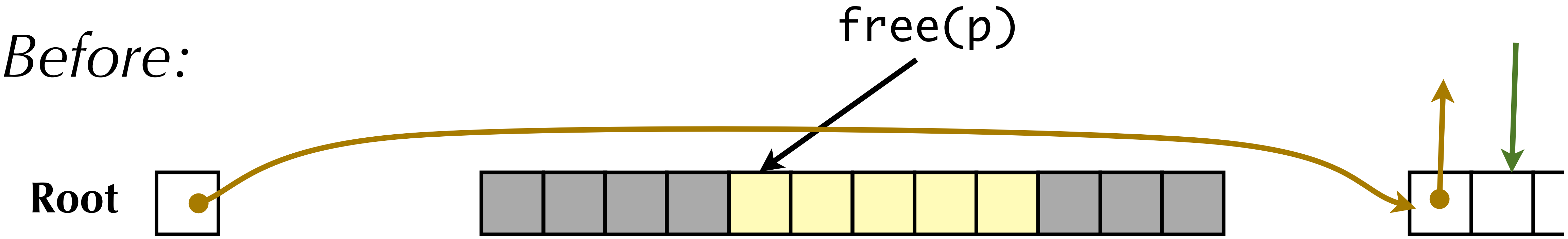
Deallocation with an explicit free list

- Same idea as previously
  - Step 1: Mark block as free
  - Step 2: Coalesce adjacent free blocks
  - Step 3: Insert free block into the free list

- 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
    - Simple and constant time
  - Address-ordered policy
    - Insert freed blocks so that free list blocks are always in address order
    - Con: requires search
    - Pro: studies suggest fragmentation is lower than LIFO

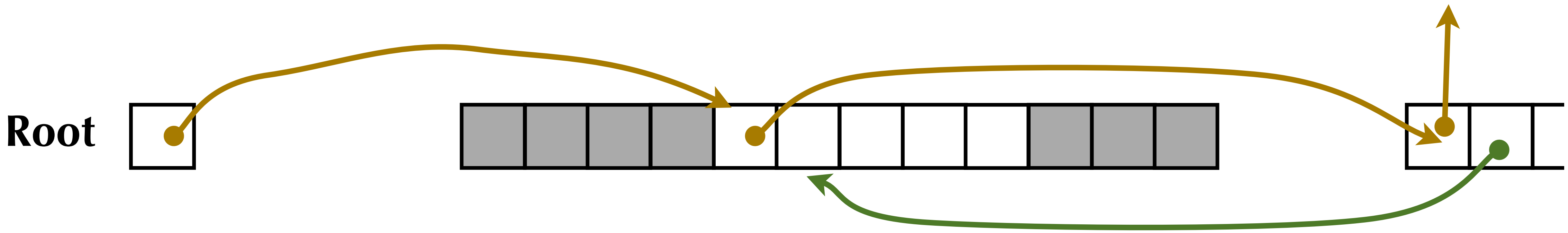
Freeing With a LIFO Policy (Case 1)

Before:

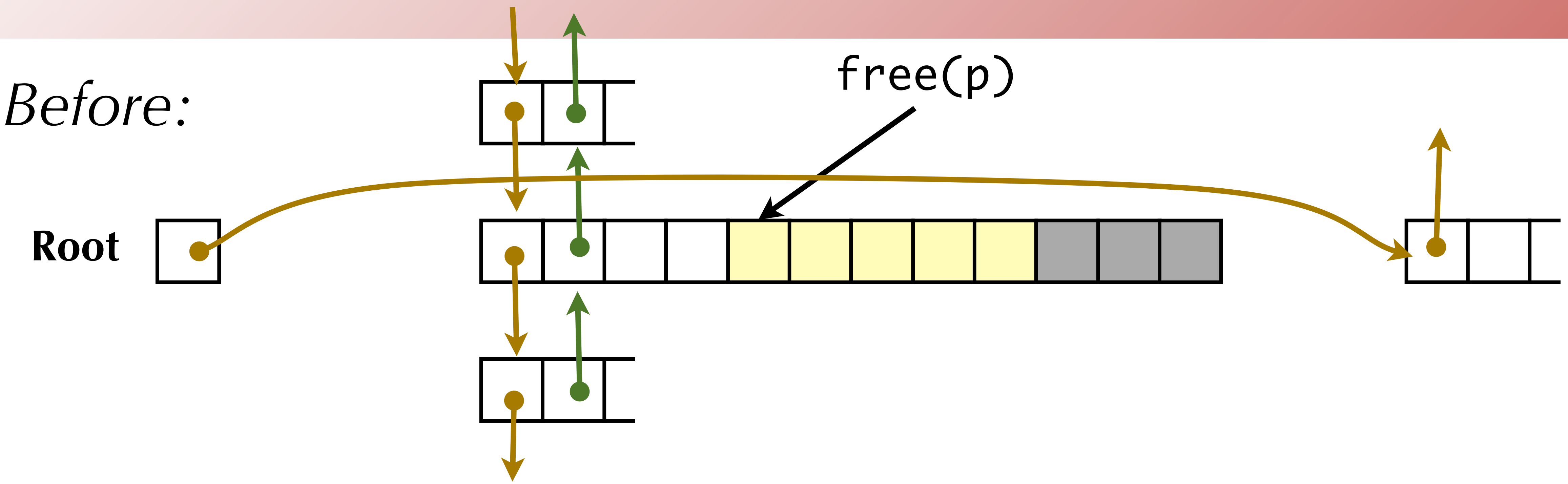


- Insert the freed block at the root of the free block list

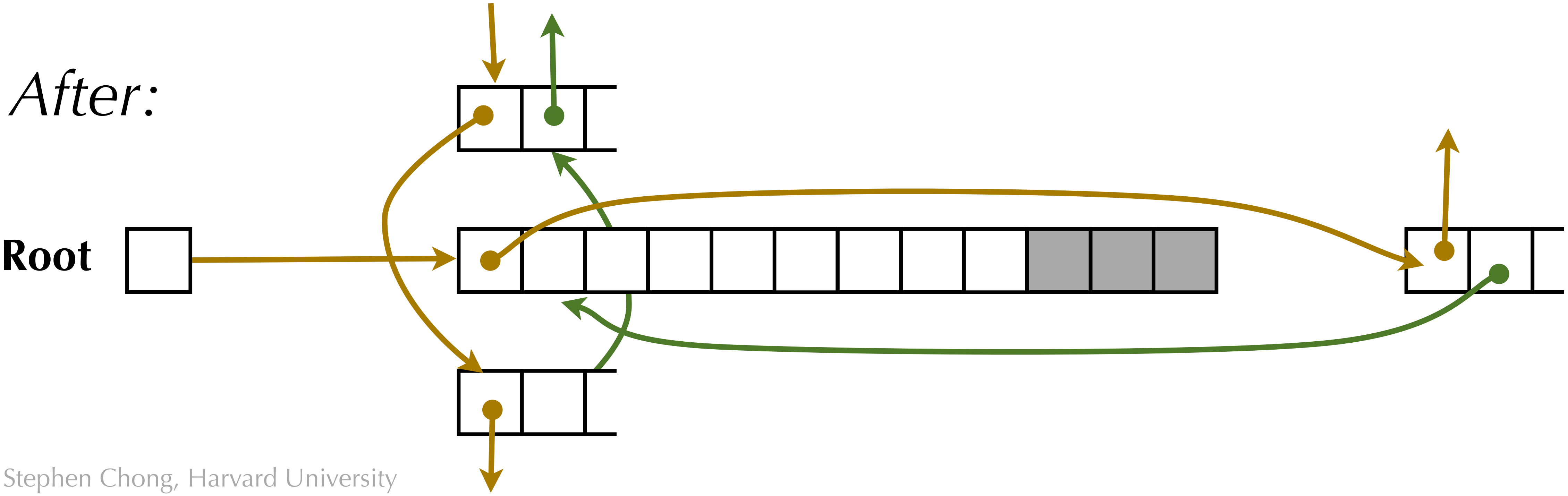
After:



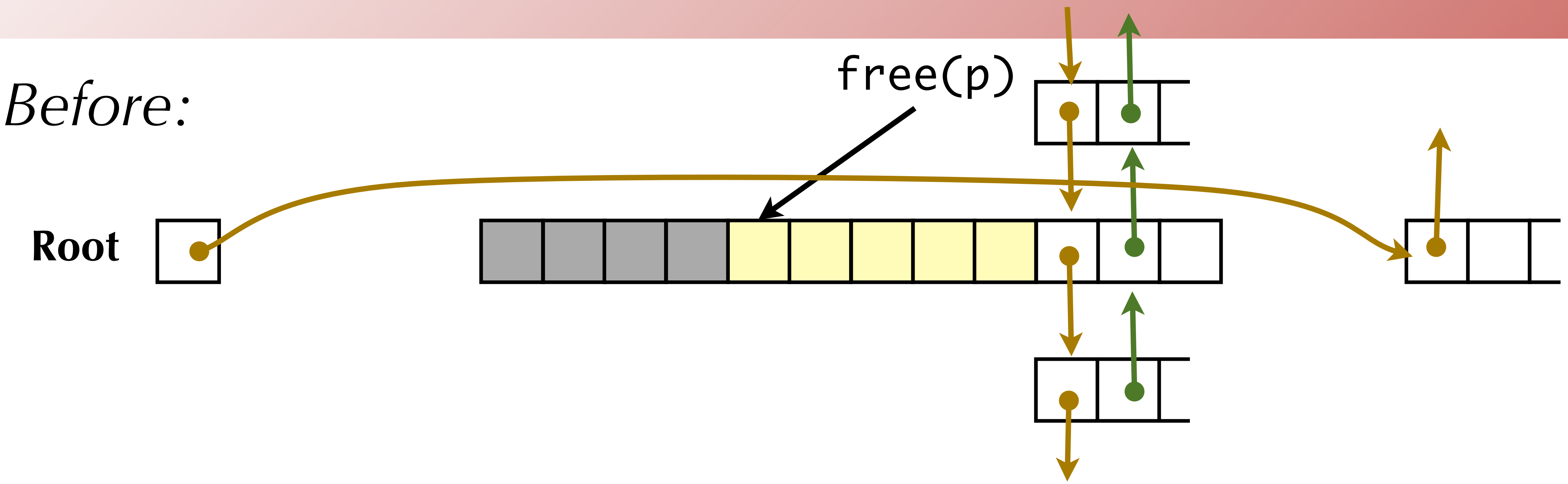
Freeing With a LIFO Policy (Case 2)



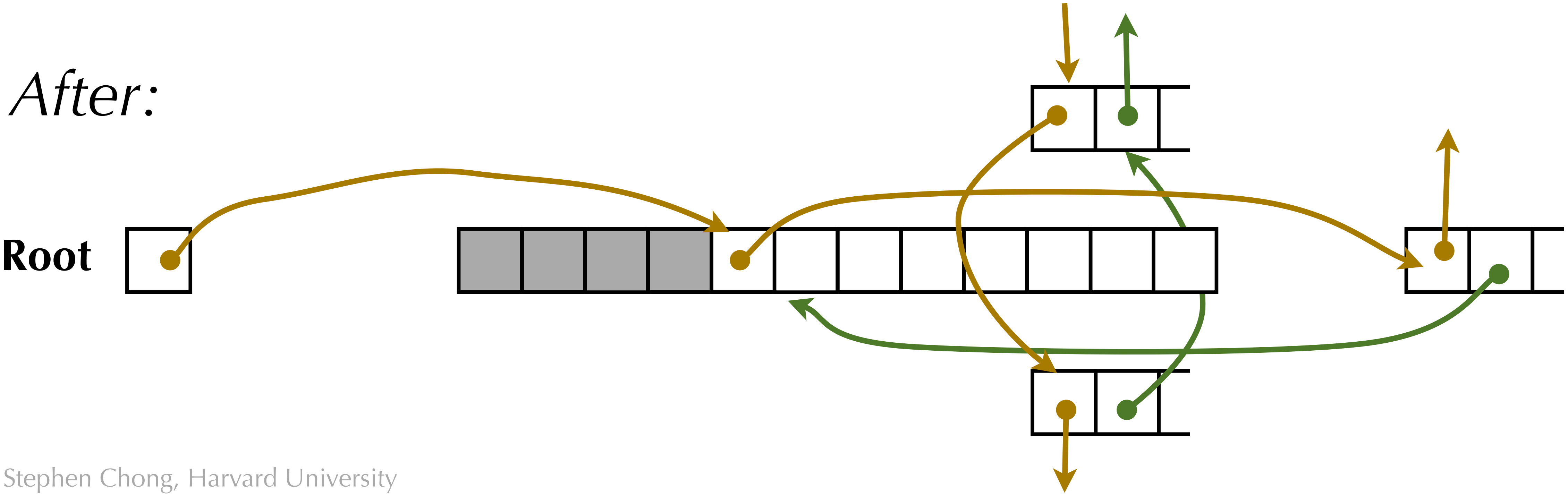
- Splice out predecessor block, coalesce both memory blocks and insert the new block at the root of the list



Freeing With a LIFO Policy (Case 3)

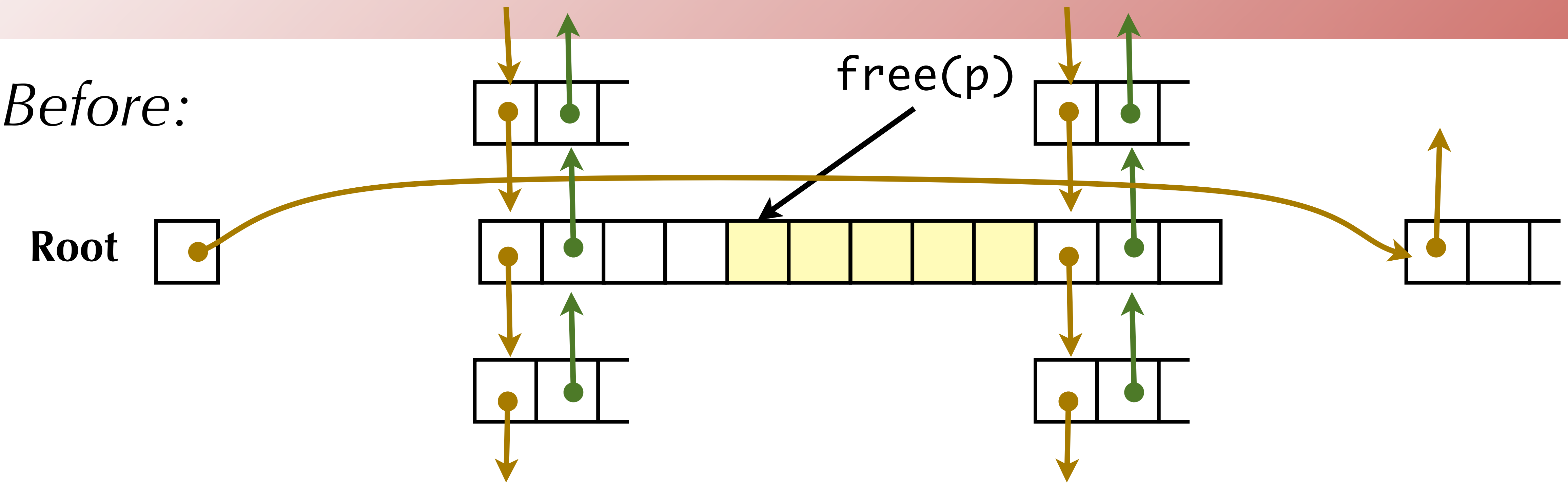


- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list



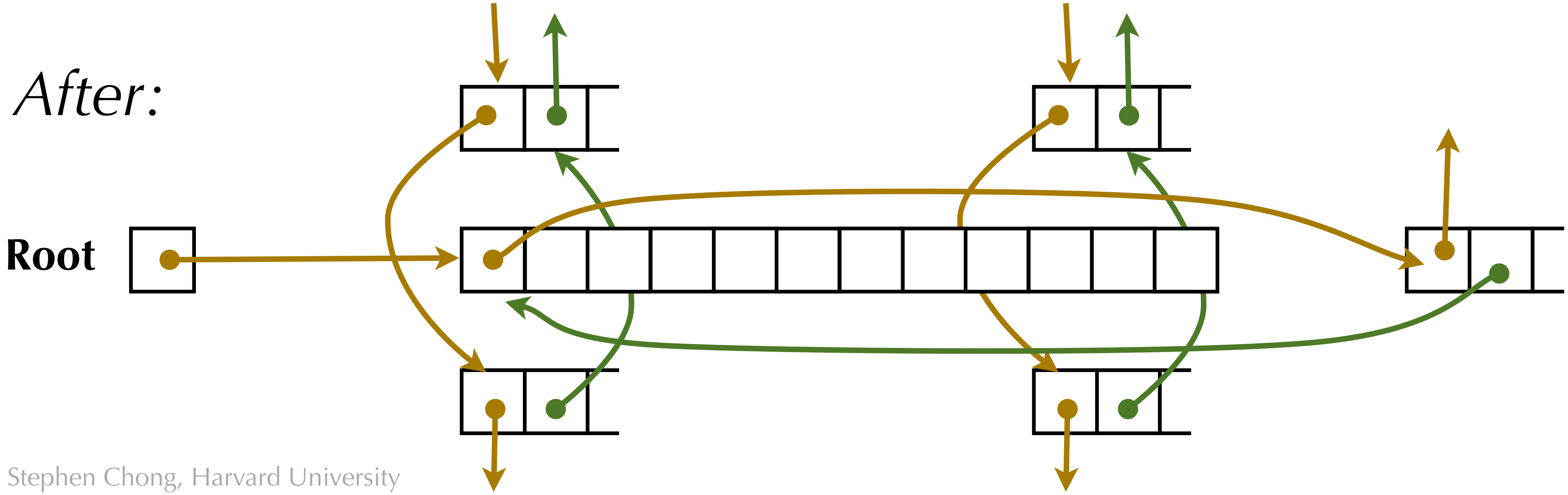
Freeing With a LIFO Policy (Case 4)

Before:



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

After:



Explicit List Summary

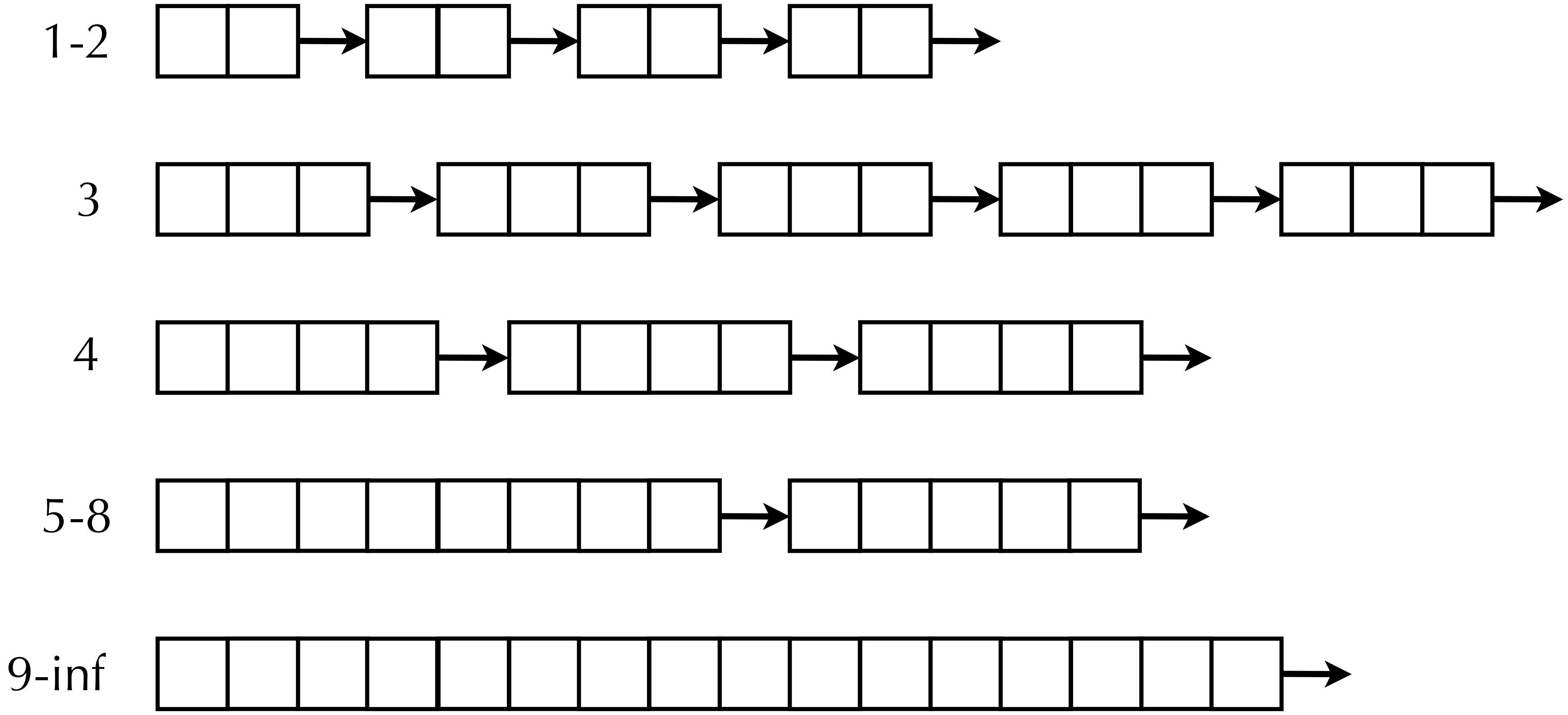
- Comparison to implicit list:
    - Allocation is linear time in number of **free blocks**
    - Implicit list allocation is linear time in the number of **total blocks**
        - Allocation for explicit lists is faster when memory is nearly full!
- Slightly more complicated allocate and free since need to splice blocks in and out of the list
- Need some extra space for the links
    - 2 extra words needed for each free block
    - But these can be stored in the payload, since only needed for free blocks.

Topics for today

- Dynamic memory allocation
  - Implicit vs. explicit memory management
- Performance goals
- Fragmentation
- Free block list management
  - Implicit free list
  - Explicit free list
  - Segmented lists
  - Tradeoffs

Segregated List (seglist) Allocators

- Use a different free list for blocks of different sizes!



  - Often have separate size class for every small size (4,5,6,…)
  - For larger sizes typically have a size class for each power of 2

Seglist Allocator

- To allocate a block of size *n*:
  - Determine correct free list to use
  - Search that free list for block of size $m \geq n$
  - If an appropriate block is found:
    - Split block and place fragment on appropriate list
  - If no block is found, try next larger class
  - Repeat until block is found
- If no free block is found:
  - Request additional heap memory from OS (using `sbrk()` system call)
  - Allocate block of *n* bytes from this new memory
  - Place remainder as a single free block in largest size class.

Freeing with Seglist

- To free a block:
  - Mark block as free
  - Coalesce (if needed)
  - Place free block on appropriate sized list

Seglist advantages

- Advantages of seglist allocators
  - Higher throughput
    - Faster to find appropriate sized block: Look in the right list.
  - Better memory utilization
    - First-fit search of segregated free list approximates a best-fit search of entire heap.
    - Extreme case: Giving each size its own segregated list is equivalent to best-fit.

Topics for today

- Dynamic memory allocation
  - Implicit vs. explicit memory management
- Performance goals
- Fragmentation
- Free block list management
  - Implicit free list
  - Explicit free list
  - Segmented lists
  - Tradeoffs

Allocation Policy Tradeoffs

- Data structure for free lists
  - Implicit lists, explicit lists, segregated lists
  - Other structures possible, e.g.explicit free blocks in binary tree, sorted by size
- Placement policy: First fit, next fit, or best fit
  - Best fit has higher overhead, but less fragmentation.
- Splitting policy: When do we split free blocks?
  - Splitting leads to more internal fragmentation, since each block needs its own header.
- Coalescing policy: When do we coalesce free blocks?
  - **Immediate coalescing**: Coalesce each time free is called
  - **Deferred coalescing**: Improve free performance by deferring coalescing until needed.
    - E.g., While scanning the free list for `malloc()`, or when external fragmentation reaches some threshold.

Topics for next time

- Allocation requirements
- Common memory bugs
- Implicit memory management: Garbage collection

Malloc Example

```
void foo(int n, int m) {
  int i, *p;

  /* allocate a block of n ints */
   p = (int *)malloc(n * sizeof(int));
   if (p == NULL) {
    perror("malloc");
    exit(0);
  }
  for (i=0; i<n; i++) p[i] = i;

  /* add m bytes to end of p block */
  if ((p = (int *) realloc(p, (n+m) * sizeof(int))) == NULL) {
    perror("realloc");
    exit(0);
  }
  for (i=n; i < n+m; i++) p[i] = i;

  /* print new array */
  for (i=0; i<n+m; i++)
    printf("%d\n", p[i]);

  free(p); /* return p to available memory pool */
}
```