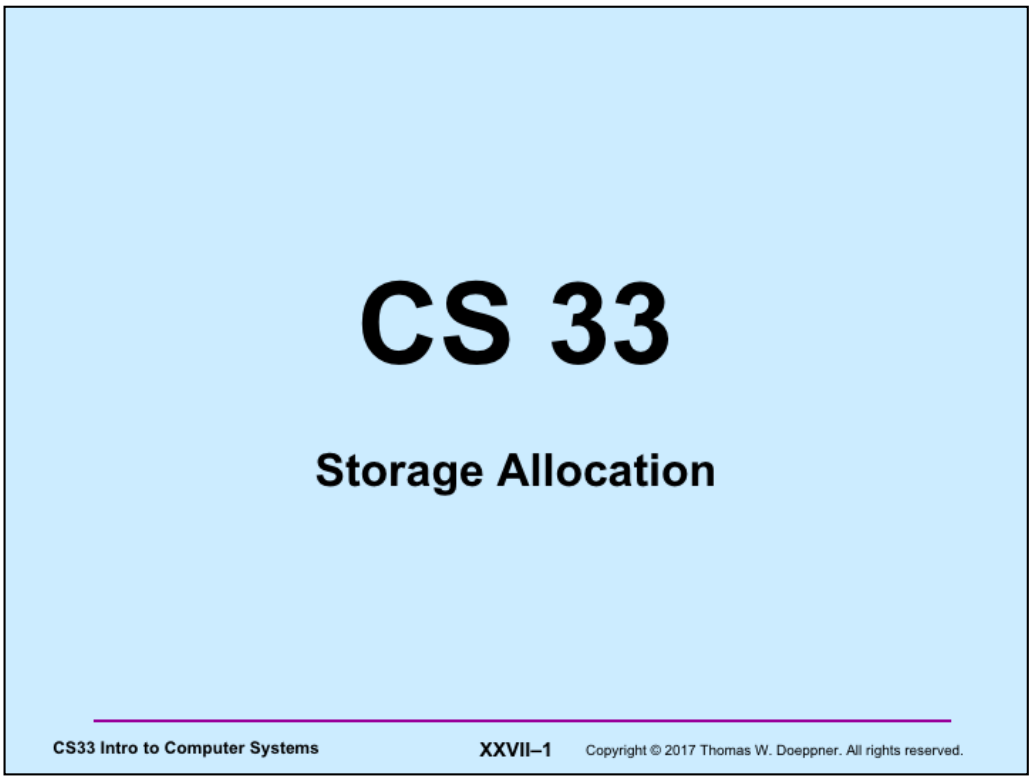
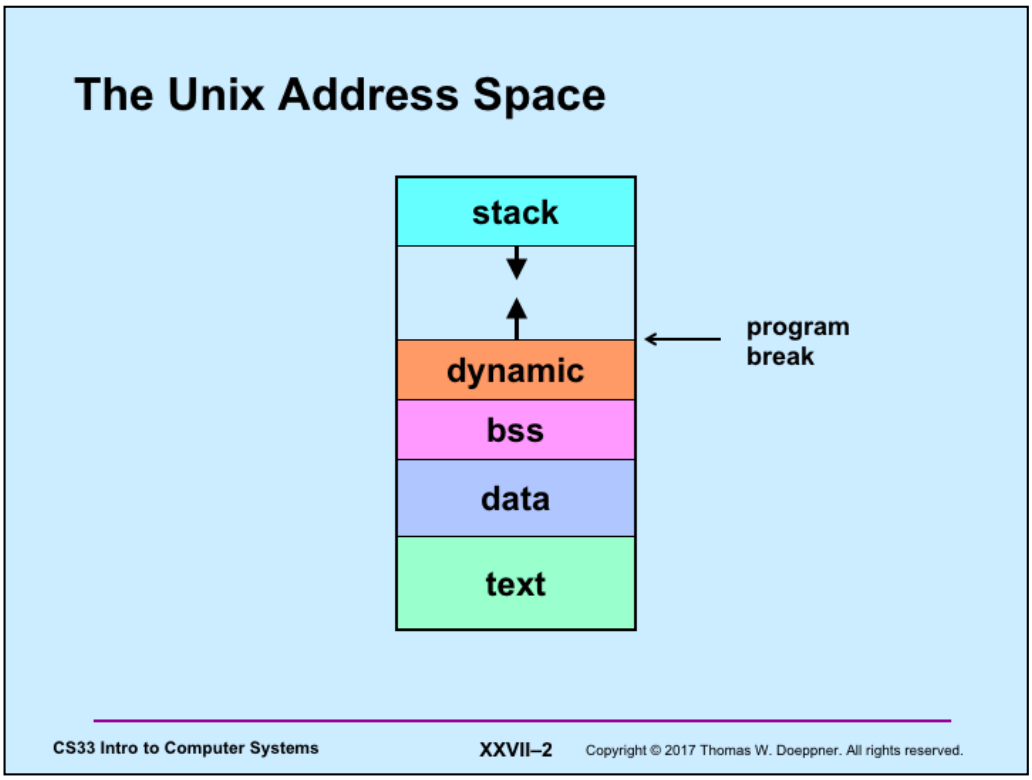


Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective," 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in Fall 2010. These slides are indicated "Supplied by CMU" in the notes section of the slides.



The program break is the upper limit of the currently allocated dynamic region.

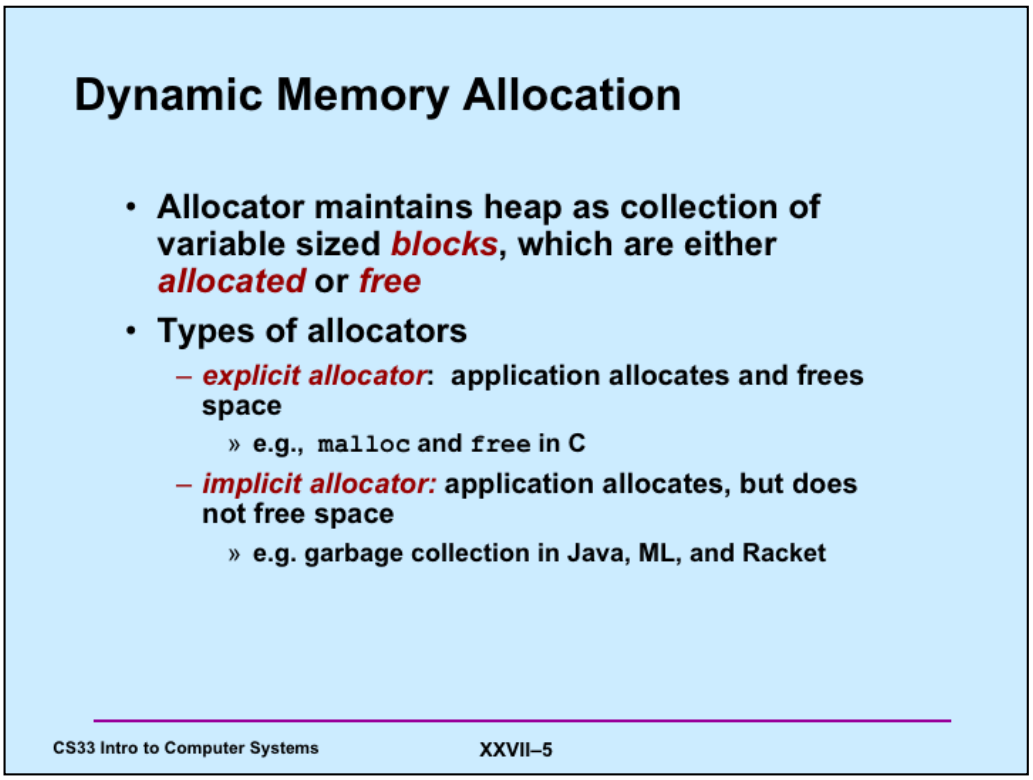
sbrk System Call

```
void *sbrk(intptr_t increment)
```

- moves the program break by an amount equal to *increment*
- returns the previous program break
- *intptr_t* is typedef'd to be a *long*

Managing Dynamic Storage

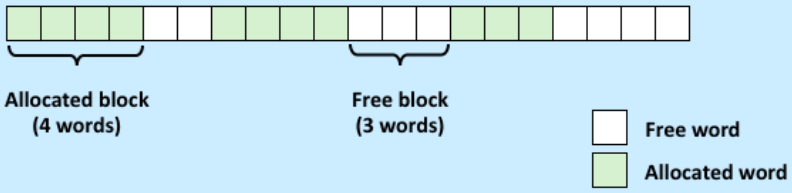
- Strategy
  - get a "chunk" of memory from the OS using *sbrk*
    - » create pool of available storage, aka the "heap"
  - *malloc*, *calloc*, *realloc*, and *free* use this storage if possible
    - » they manage the heap
  - if not possible, get more storage from OS
    - » heap is made larger (by calling sbrk)
- Important note:
  - when process terminates, all storage is given back to the system
    - » all memory-related sins are forgotten!



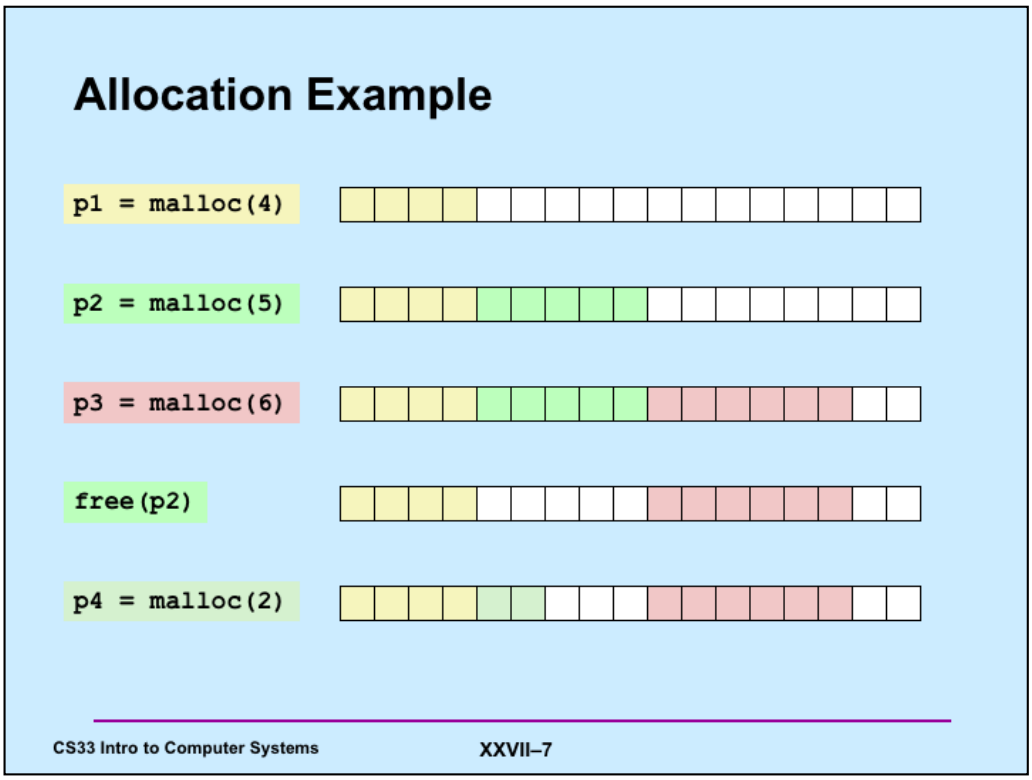
Supplied by CMU.

Assumptions Made in This Lecture

- Memory is word addressed (each word can hold a pointer)



Supplied by CMU.



Supplied by CMU.

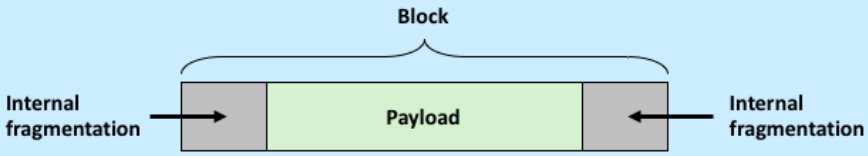
Constraints

- Applications
  - can issue arbitrary sequence of `malloc` and `free` requests
  - `free` request must be to a malloc'd block
- Allocators
  - can't control number or size of allocated blocks
  - must respond immediately to `malloc` 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
    - » 8-byte alignment for GNU `malloc` (`libc malloc`) on Linux
  - can manipulate and modify only free memory
  - can't move the allocated blocks once they are `malloc`'d
    - » i.e., compaction is not allowed

Supplied by CMU.

Internal Fragmentation

- For a given block, *internal fragmentation* occurs if payload is smaller than block size



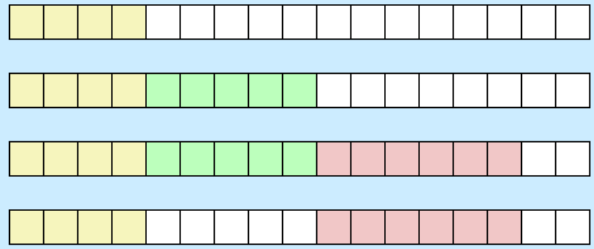
- Caused by
  - overhead of maintaining heap data structures
  - padding for alignment purposes
  - explicit policy decisions (e.g., to return a big block to satisfy a small request)
- Depends only on the pattern of *previous* requests
  - thus, easy to measure

Supplied by CMU.

External Fragmentation

- Occurs when there is enough aggregate heap memory, but no single free block is large enough

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



Oops! (what would happen now?)

- Depends on the pattern of future requests
  - thus, difficult to measure

Supplied by CMU.

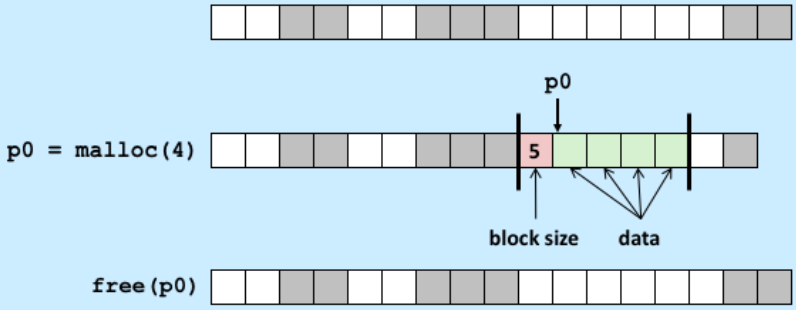
Implementation Issues

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

Supplied by CMU.

Knowing How Much to Free

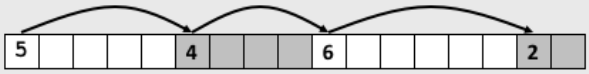
- **Standard method**
  - keep the length of a block in the word preceding the block
    - » this word is often called the *header field* or *header*
  - requires an extra word for every allocated block



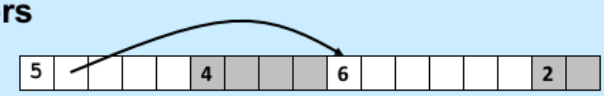
Supplied by CMU.

Keeping Track of Free Blocks

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



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



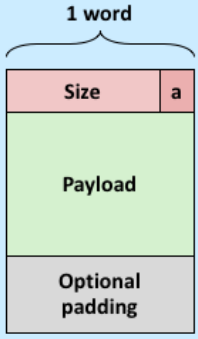
- Method 3: *Segregated free list*
  - different free lists for different size classes
- Method 4: *Blocks sorted by size*
  - can use a balanced tree (e.g. red-black tree) with pointers within each free block, and the length used as a key

Supplied by CMU.

Method 1: Implicit List

- For each block we need both size and allocation status
  - could store this information in two words: wasteful!
- Standard trick
  - if blocks are aligned, some low-order address bits are always 0
  - instead of storing an always-0 bit, use it as a allocated/free flag
  - when reading size word, mask out this bit

Format of allocated and free blocks

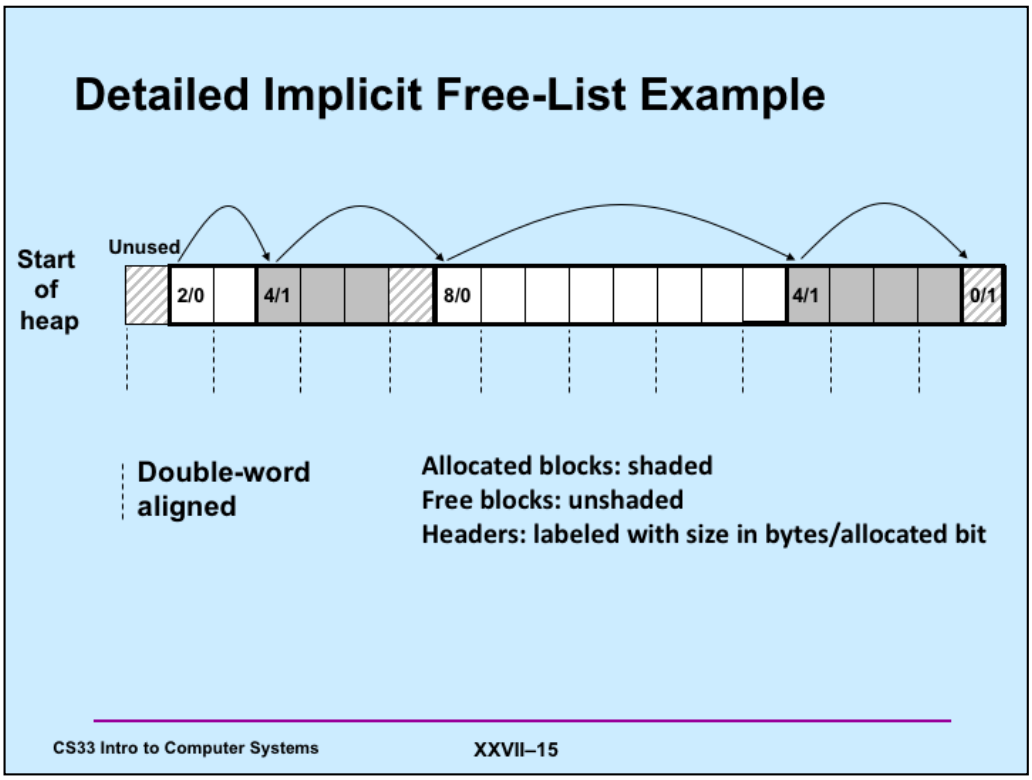


a = 1: Allocated block
a = 0: Free block

Size: block size

Payload: application data (allocated blocks only)

Supplied by CMU.



Supplied by CMU.

Implicit List: Finding a Free Block

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

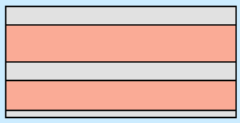
```
p = start;
while ((p < end) &&      // not past end
       ((*p & 1) ||      // already allocated
       (*p <= len)))     // too small
  p = p + (*p & -2);     // goto next block (word addressed)
```

  - can take linear time in total number of blocks (allocated and free)
  - in practice it can cause "splinters" at beginning of list
- ***Next fit:***
  - like first fit, but search list starting where previous search finished
  - should often be faster than first fit: avoids re-scanning unhelpful blocks
  - some research suggests that fragmentation is worse
- ***Best fit:***
  - search the list, choose the ***best*** free block: fits, with fewest bytes left over
  - keeps fragments small—usually helps fragmentation
  - will typically run slower than first fit

Supplied by CMU.

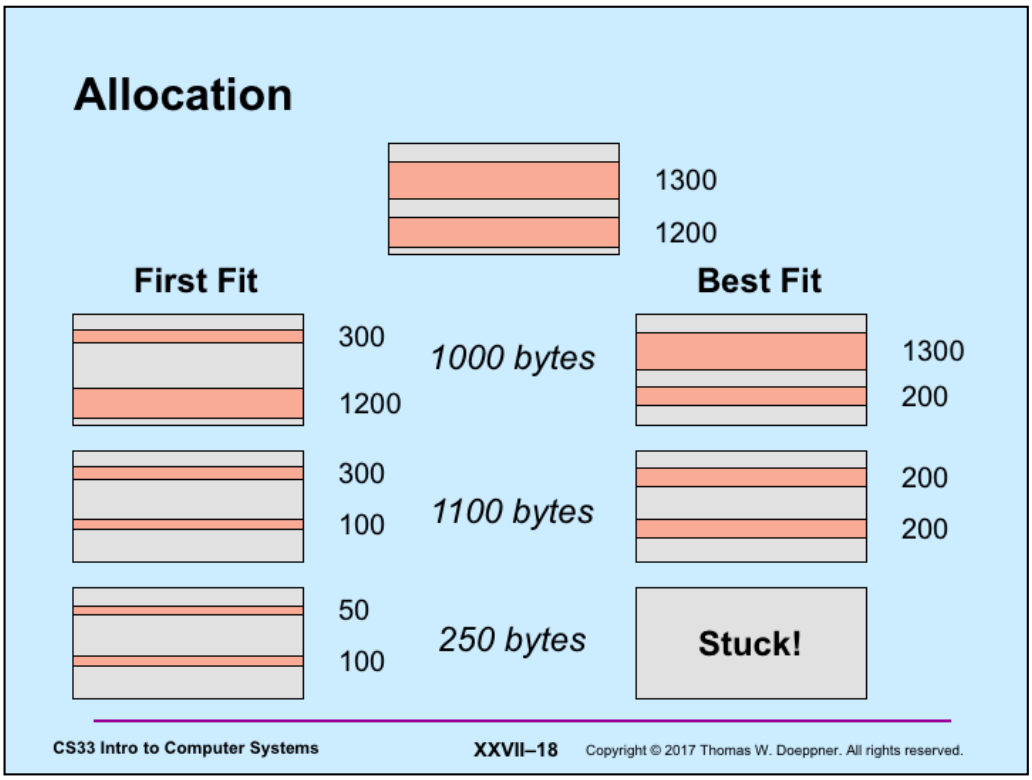
Note that the expression "(*p & -2) computes the bitwise *and* of *p and 0xfffffffffffffffe – i.e., it sets the low-order bit of *p to 0, thus masking off the allocated flag.

Quiz 1



We have two free blocks of memory, of sizes 1300 and 1200 (appearing in that order). There are three successive requests to *malloc* for allocations of 1000, 1100, and 250 bytes. Which approach does best? (Hint: one of the two fails the last request.)

a) **first fit**
b) **best fit**



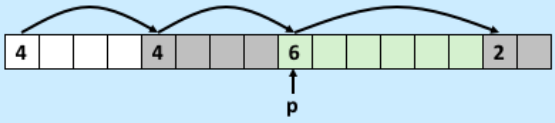
Consider the situation in which we have one large pool of memory from which we will allocate (and to which we will liberate) variable-sized pieces of memory. Assume that we are currently in the situation shown at the top of the picture: two unallocated areas of memory are left in the pool — one of size 1300 bytes, the other of size 1200 bytes. We wish to process a series of allocation requests, and will try out two different algorithms. The first is known as *first fit* — an allocation request is taken from the first area of memory that is large enough to satisfy the request. The second is known as *best fit*—the request is taken from the smallest area of memory that is large enough to satisfy the request. On the principle that whatever requires the most work must work the best, one might think that best fit would be the algorithm of choice.

The picture illustrates a case in which first fit behaves better than best fit. We first allocate 1000 bytes. Under the first-fit approach (shown on the left side), this allocation is taken from the topmost region of free memory, leaving behind a region of 300 bytes of still unallocated memory. With the best-fit approach (shown on the right side), this allocation is taken from the bottommost region of free memory, leaving behind a region of 200 bytes of still-unallocated memory. The next allocation is for 1100 bytes. Under first fit, we now have two regions of 300 bytes and 100 bytes. Under best fit, we have two regions of 200 bytes. Finally, there is an allocation of 250 bytes. Under first fit this leaves behind two regions of 50 bytes and 100 bytes, but the allocation cannot be handled under best fit — neither remaining region is large enough.

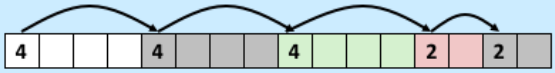
This example comes from the classic book, *The Art of Computer Programming, Vol. 1, Fundamental Algorithms*, by Donald Knuth.

Implicit List: Allocating in Free Block

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



addblock(p, 4)




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

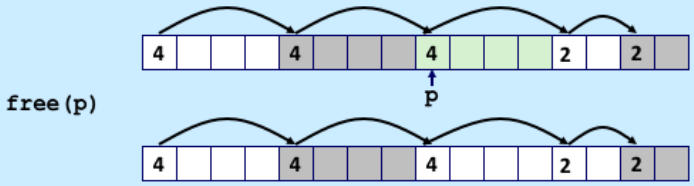

Supplied by CMU.

Implicit List: Freeing a Block

- **Simplest implementation:**
  - **need only clear the "allocated" flag**

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

  - **but can lead to "false fragmentation"**



`free(p)`

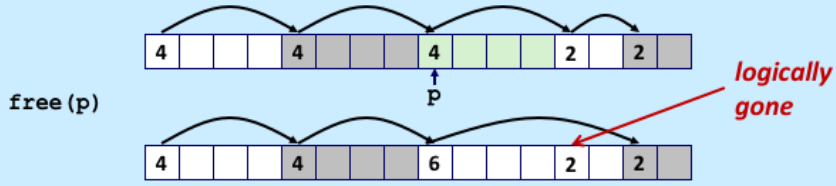
`malloc(5)` ***Oops!***

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

Supplied by CMU.

Implicit List: Coalescing

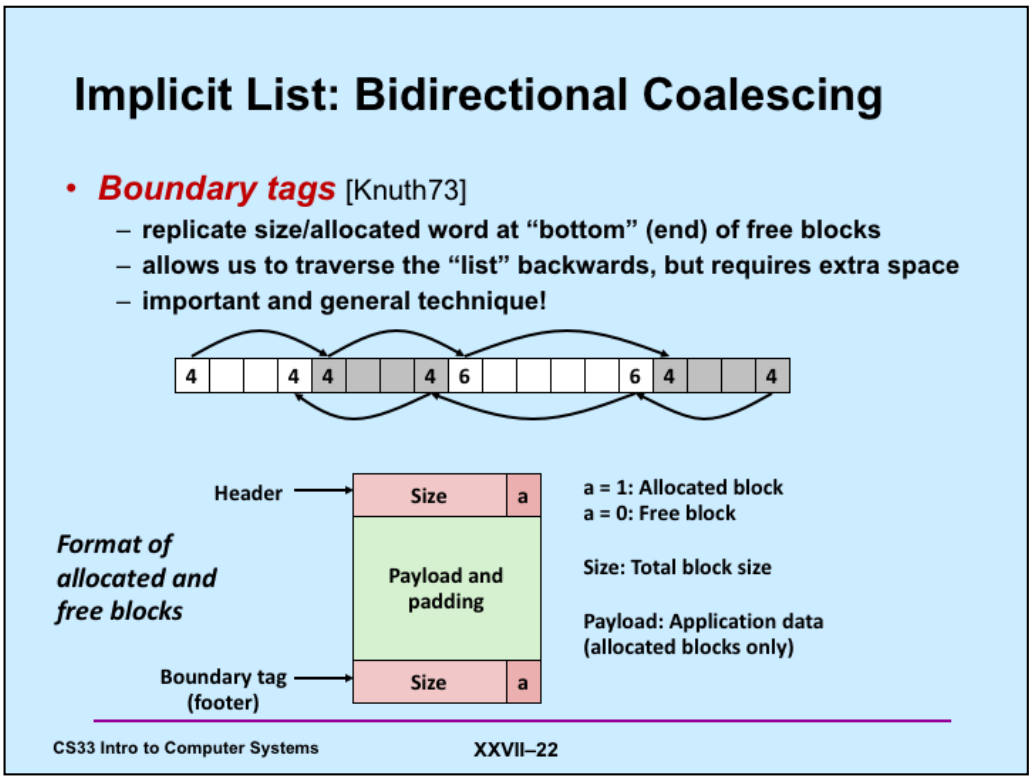
- Join *(coalesce)* with next/previous blocks, if they are free
  - coalescing with next block




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

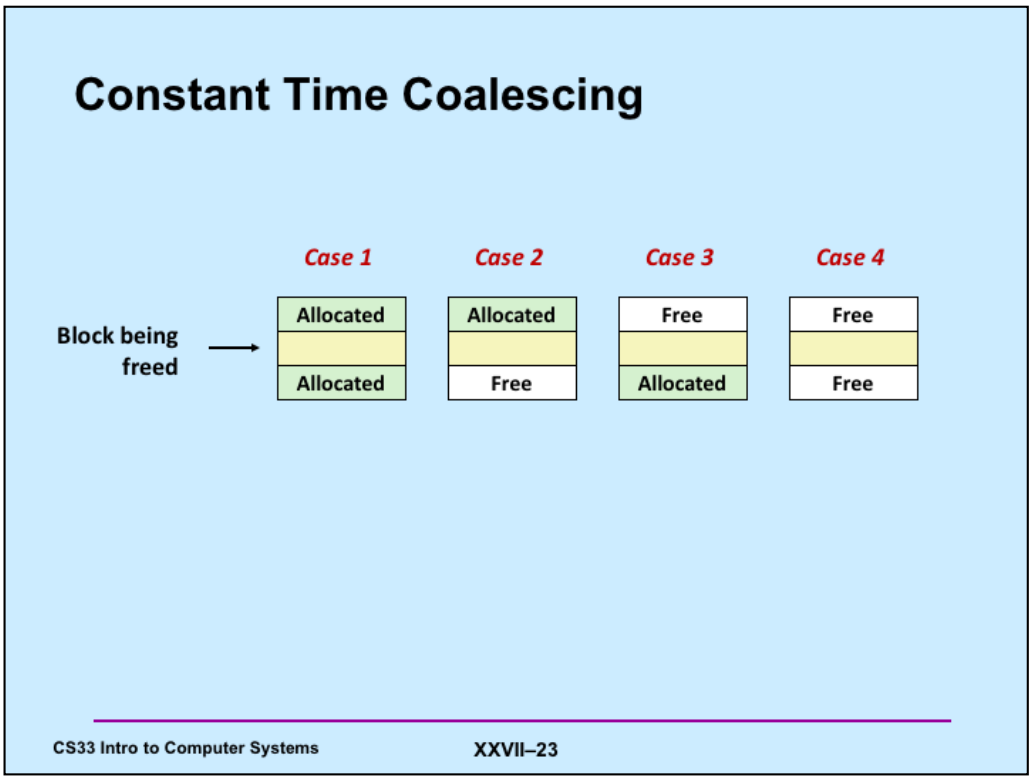
  - but how do we coalesce with *previous* block?

Supplied by CMU.

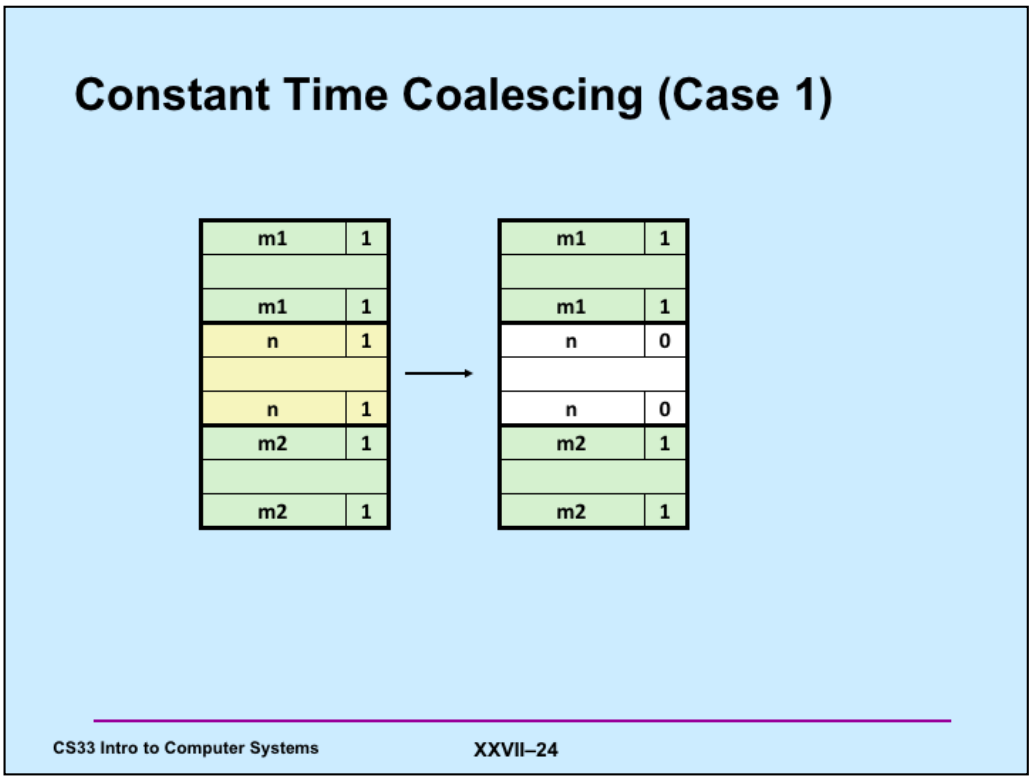


Supplied by CMU.

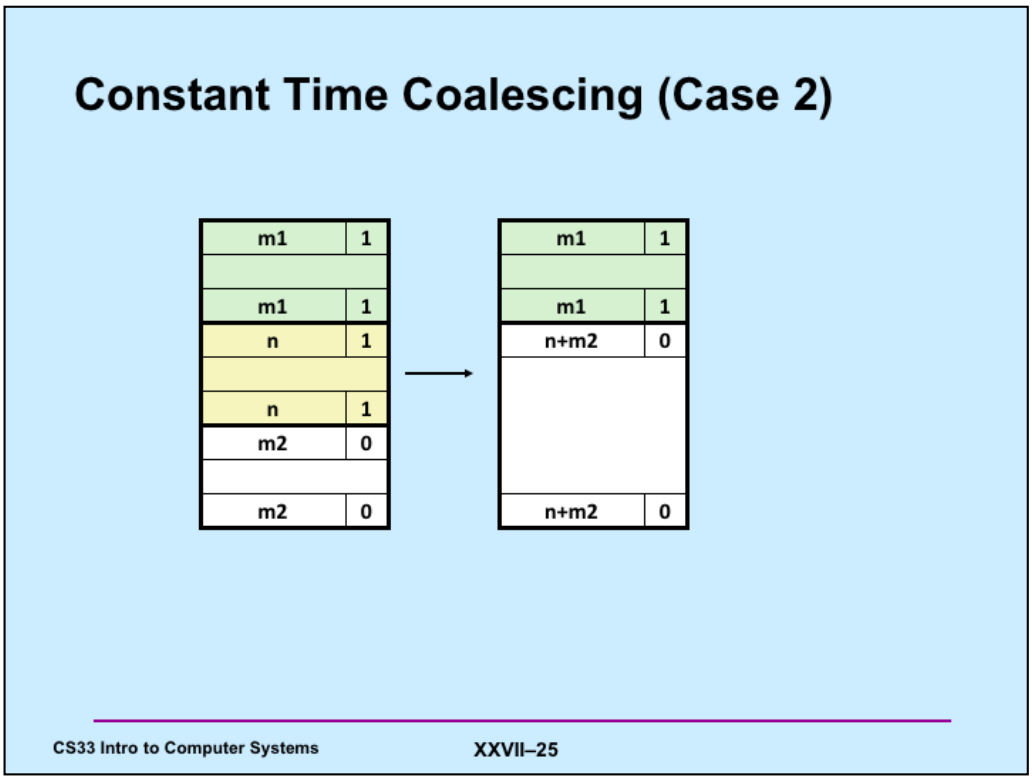
Donald Knuth is perhaps the most famous computer scientist (among computer scientists) and had been writing and continually revising the multi-volume "The Art of Computer Programming" since the 1960s. The "boundary tag" technique was described early on in the work.



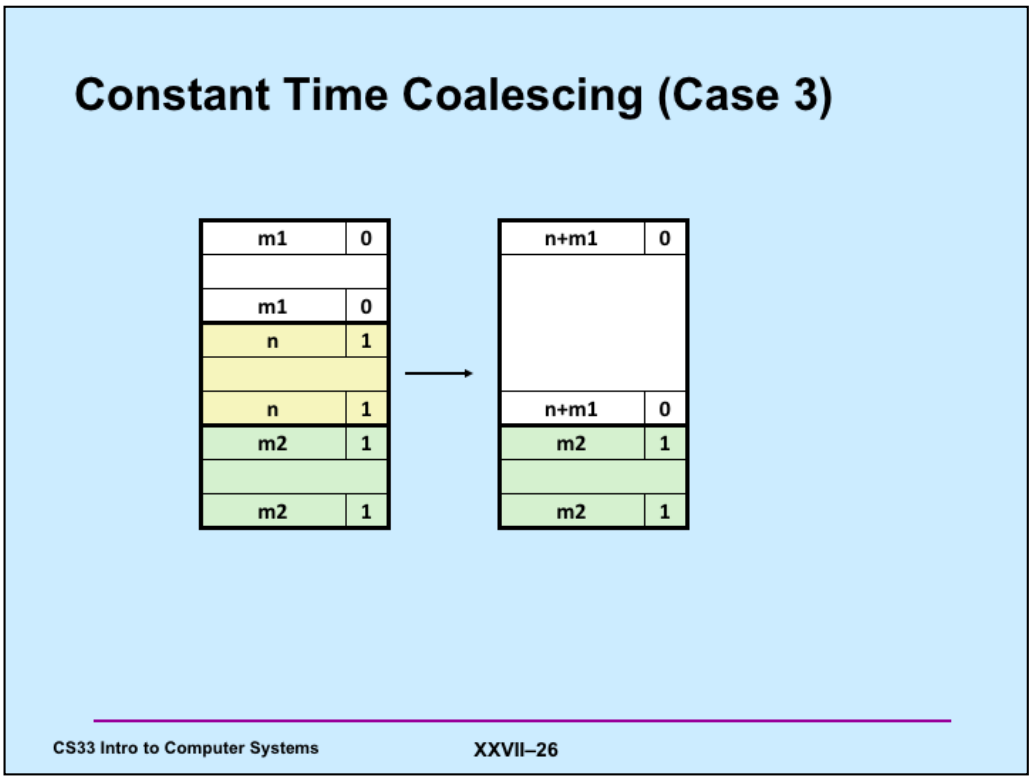
Supplied by CMU.



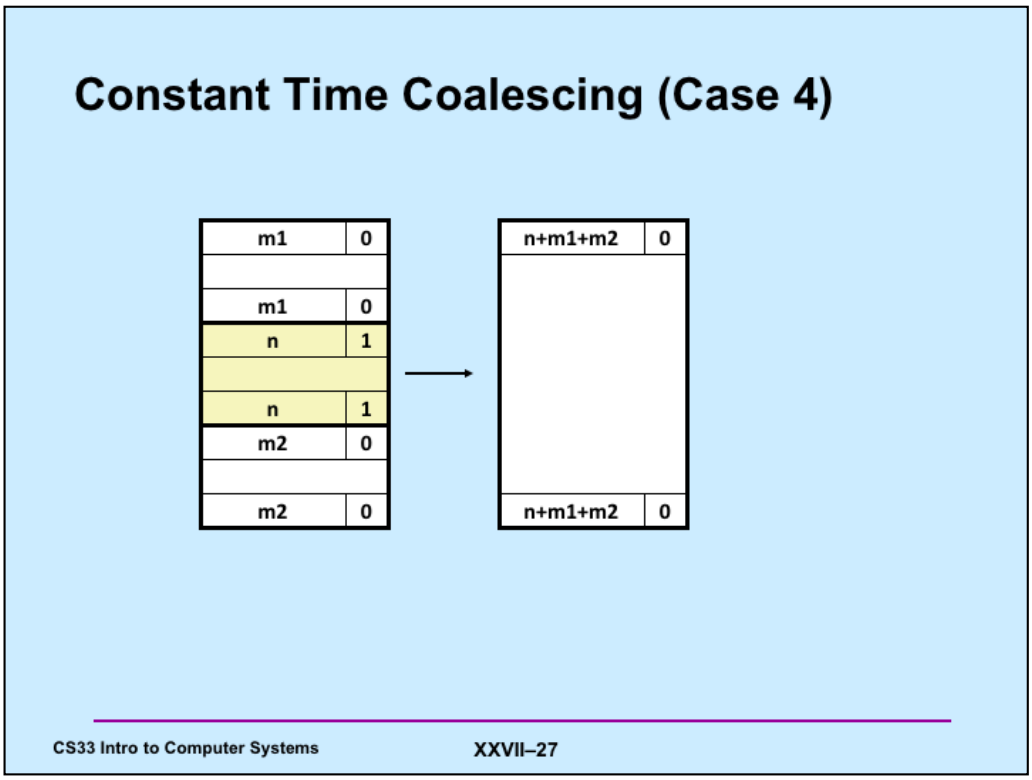
Supplied by CMU.



Supplied by CMU.



Supplied by CMU.



Supplied by CMU.

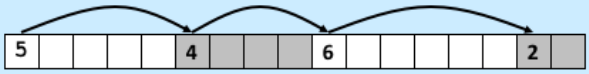
Implicit Lists: Summary

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

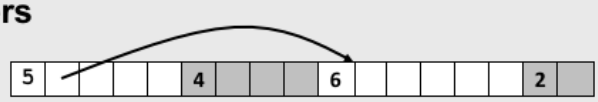
Supplied by CMU.

Keeping Track of Free Blocks

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



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



- Method 3: *segregated free list*
  - different free lists for different size classes
- Method 4: *blocks sorted by size*
  - can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Supplied by CMU.

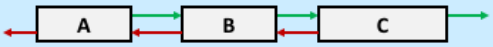
Explicit Free Lists

- Maintain list(s) of *free* blocks, not *all* blocks
  - the "next" free block could be anywhere
    - » so we need to store forward/back pointers, not just sizes
    - » luckily we track only free blocks, so we can use payload area
  - still need boundary tags for coalescing

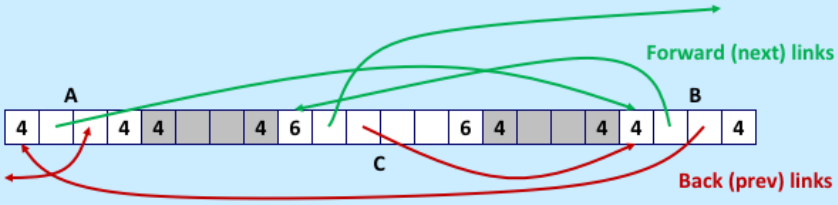
Supplied by CMU.

Explicit Free Lists

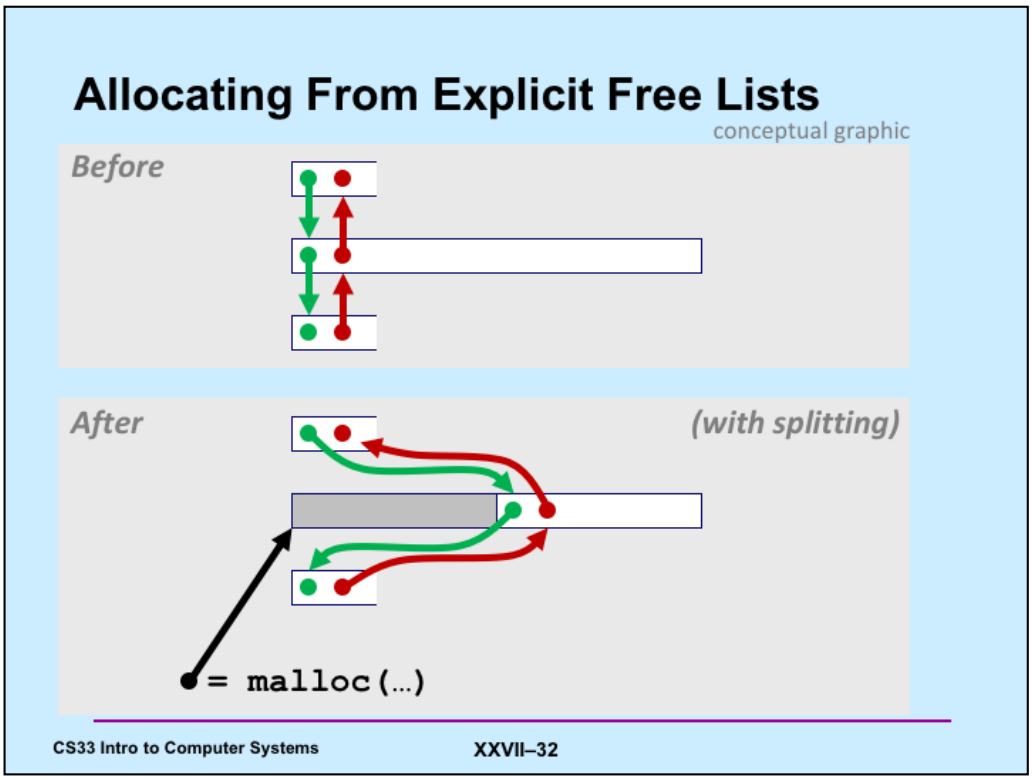
- Logically:



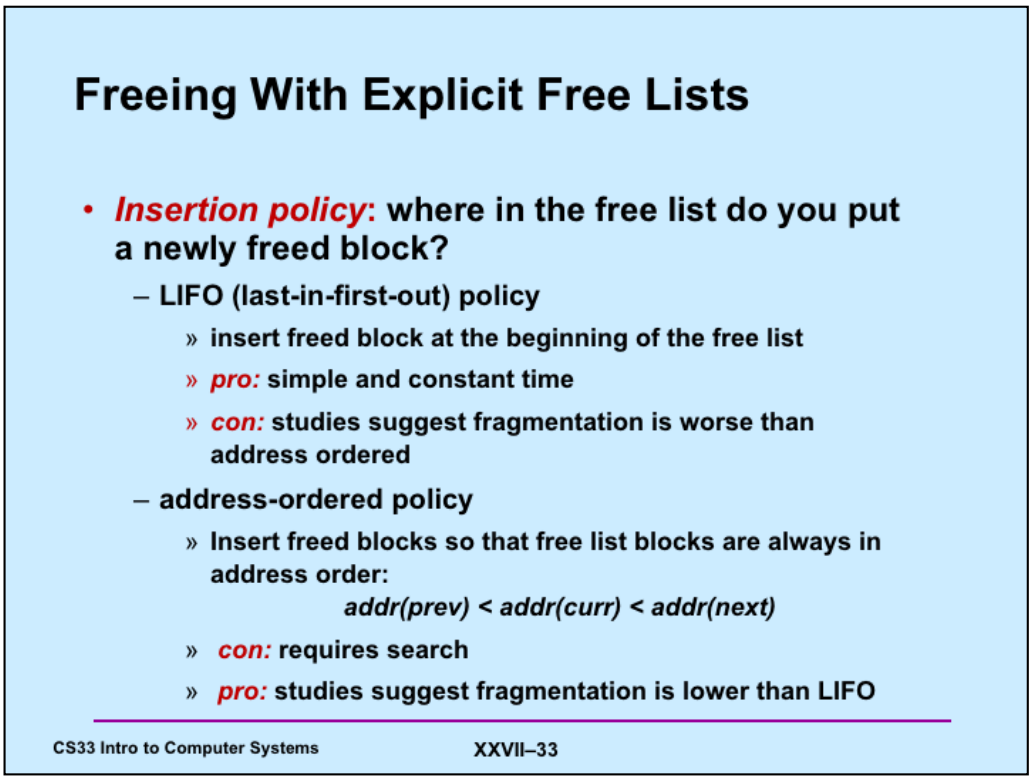
- Physically: blocks can be in any order



Supplied by CMU.

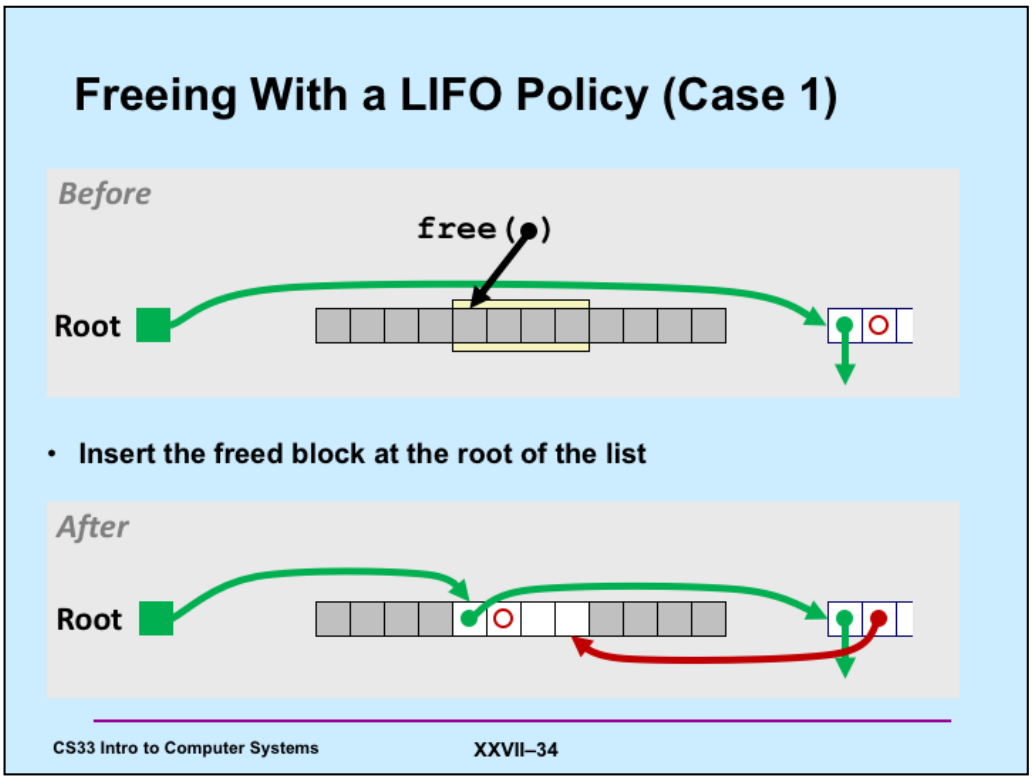


Supplied by CMU.

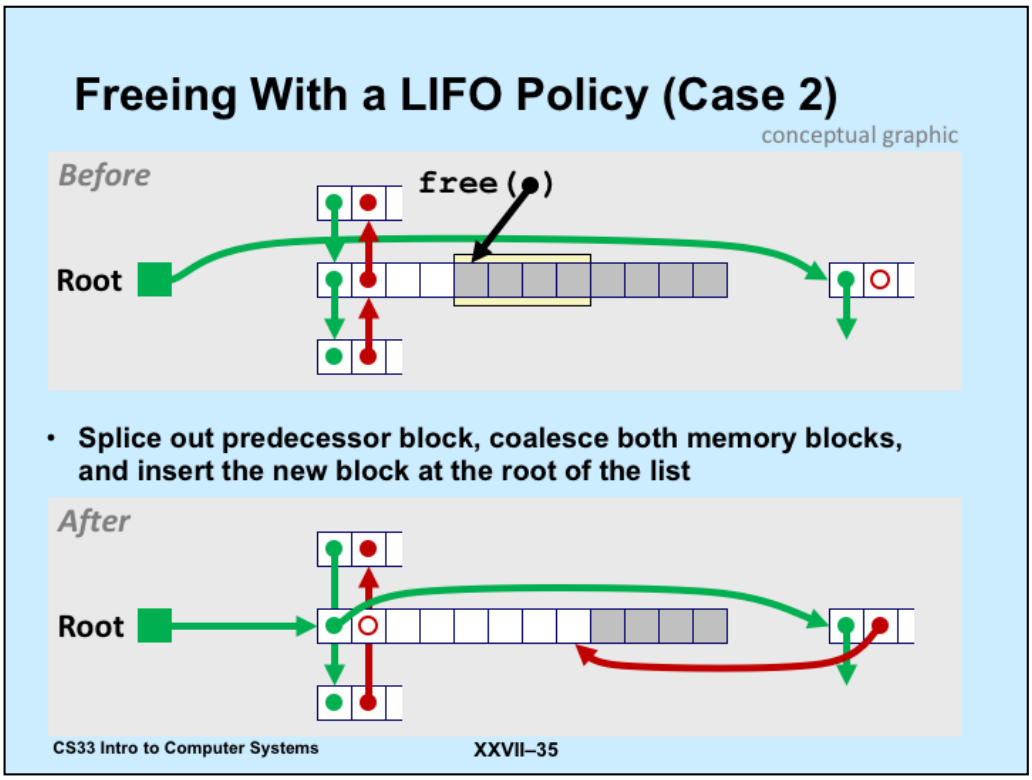


Supplied by CMU.

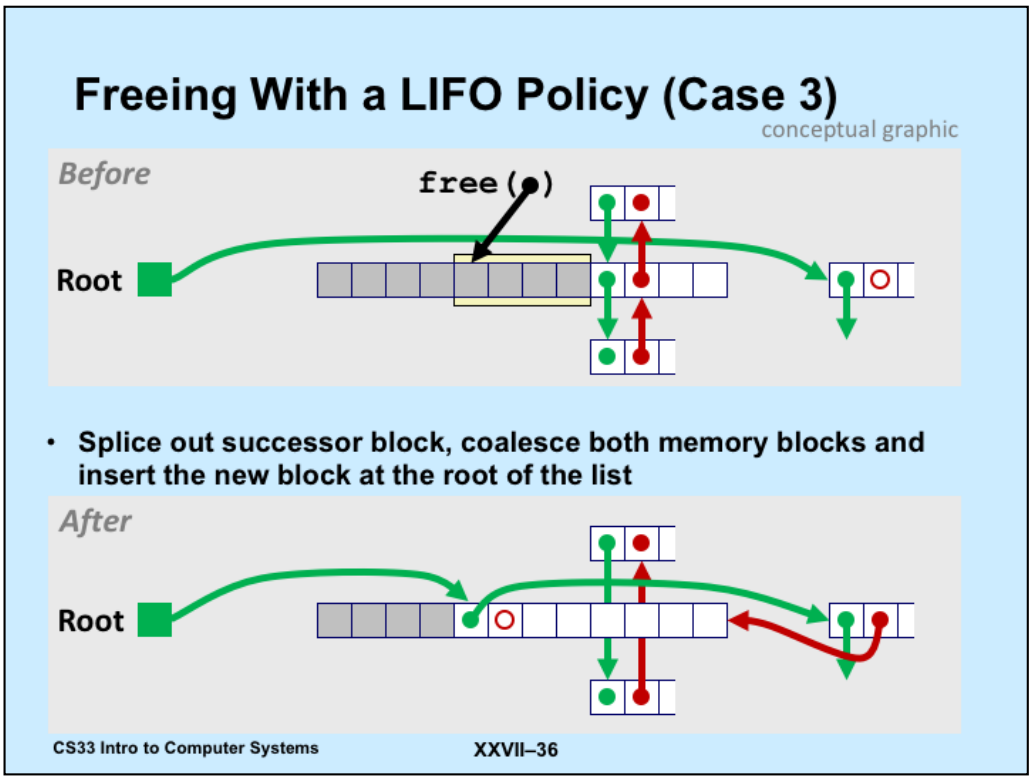
Assume that allocation is first-fit. The claim is that "studies suggest" that fragmentation from first-fit applied to an address-ordered policy is almost as low as in best-fit.



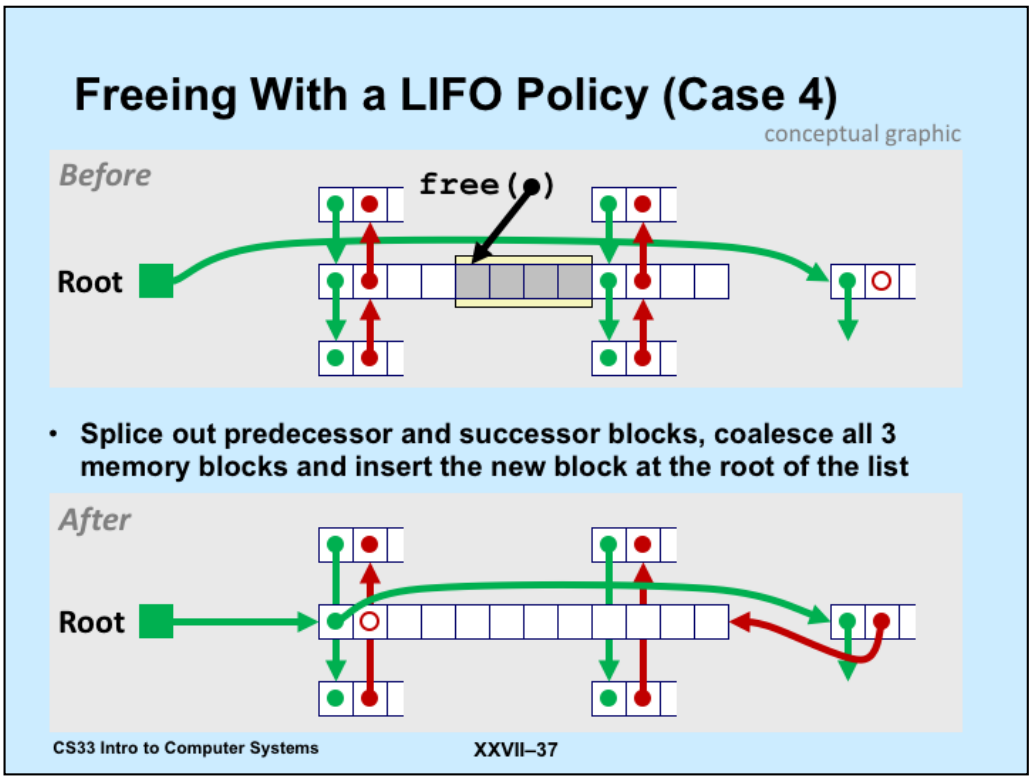
Supplied by CMU.



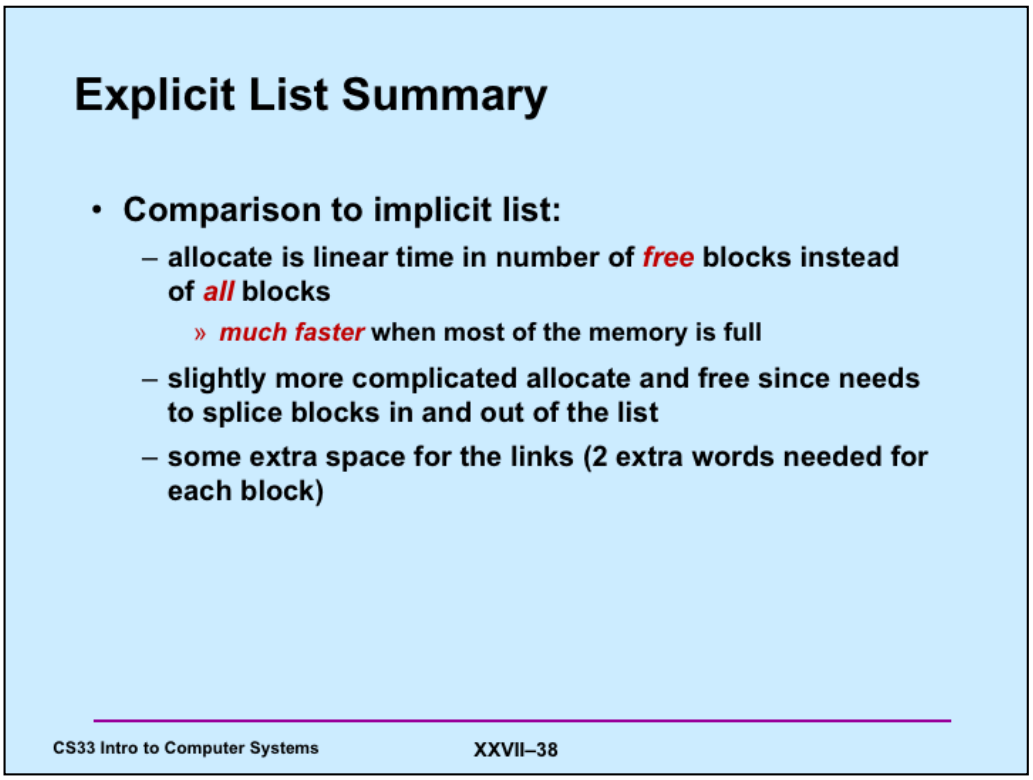
Supplied by CMU.



Supplied by CMU.



Supplied by CMU.



Supplied by CMU.

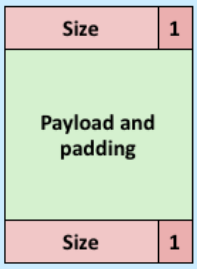
Quiz 2

Assume that best-fit results in less external fragmentation than first-fit.

We are running an application with modest memory demands. Which allocation strategy is likely to result in better performance (in terms of time) for the application:

a) best-fit
b) first-fit with LIFO insertion
c) first-fit with ordered insertion

C vs. Storage Allocation




```
typedef struct block {
  long size;
  long payload[size/8 - 2];
  long end_size;
} block_t;
```

```
typedef struct free_block {
  long size;
  struct free_block *next;
  struct free_block *prev;
  long filler[size/8 - 4];
  long end_size;
} free_block_t;
```

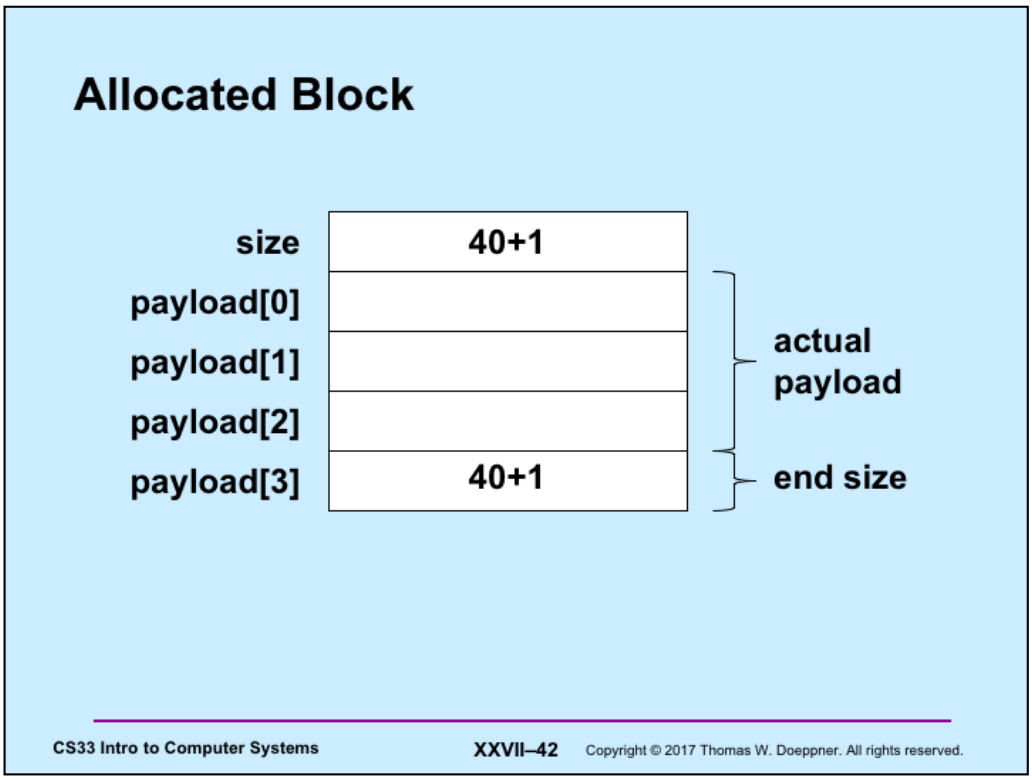

What we might like to be able to do in C is expressed on the slide. Unfortunately, C does not allow such variable-sized arrays. Another concern is the allocated flag, which we'd like to be included in the size fields.

Overcoming C

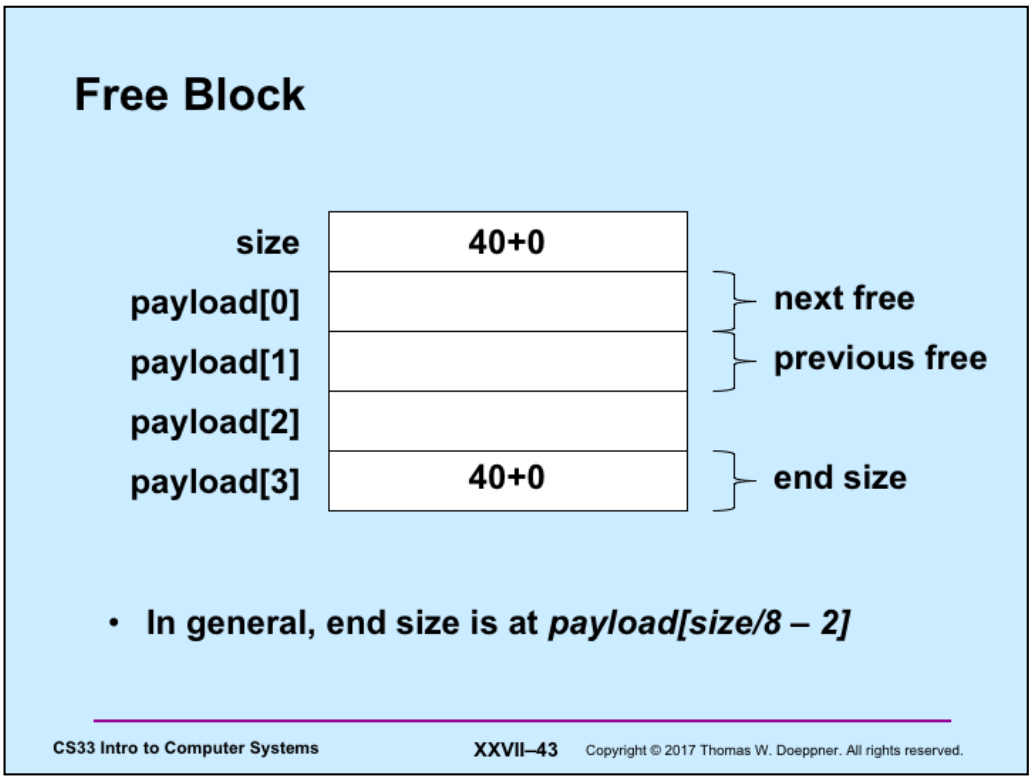
- Think objects
  - a block is an object
    - » opaque to the outside world
  - define accessor functions to get and set its contents

```
typedef struct block {
  size_t size;
  size_t payload[0];
} block_t;
```


Putting a zero for the dimension of payload is a way of saying that we do not know a priori how big payload will be, so we give it an (arbitrary) size of 0.

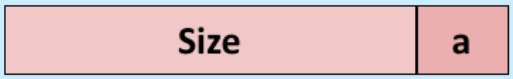


In this example we have an allocated block of 40 bytes. Its size and end size fields have their low-order bits set to one to indicate that the block is allocated.



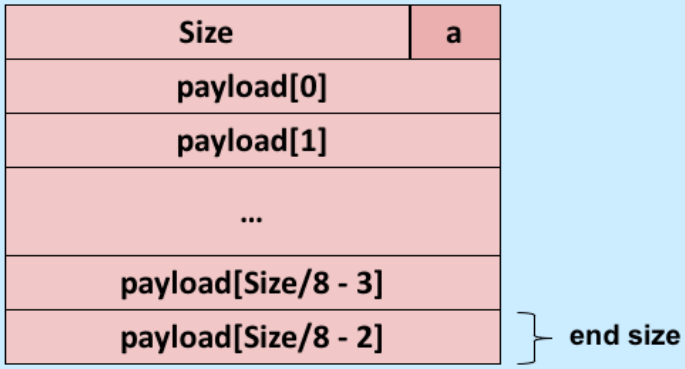
For a free block, the size fields contain the exact size of the block: the allocated bits are zeroes. The first two elements of payload are the next and previous pointers, respectively.

Overloading Size




```
size_t block_allocated(block_t *b) {
  return b->size & 1;
}

size_t block_size(block_t *b) {
  return b->size & -2;
}
```


End Size

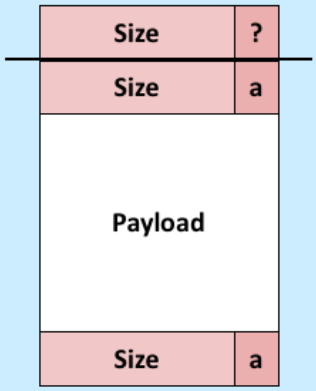



```
size_t *block_end_tag(block_t *b) {
  return &b->payload[b->size/8 - 2];
}
```


Setting the Size

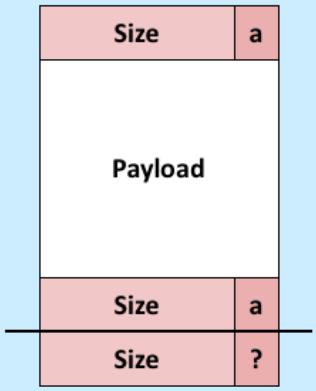
```
void block_setsize(block_t *b, size_t size) {
  assert(!(size & 7));          // multiple of 8
  size |= block_allocated(b);   // preserve alloc bit
  b->size = size;
  *block_end_tag(b) = size;
}

void block_set_allocated(block_t *b, size_t a) {
  assert((a == 0) || (a == 1));
  if (a) {
    b->size |= 1;
    *block_end_tag(b) |= 1;
  } else {
    b->size &= -2;
    *block_end_tag(b) &= -2;
  }
}
```


Is Previous Adjacent Block Free?



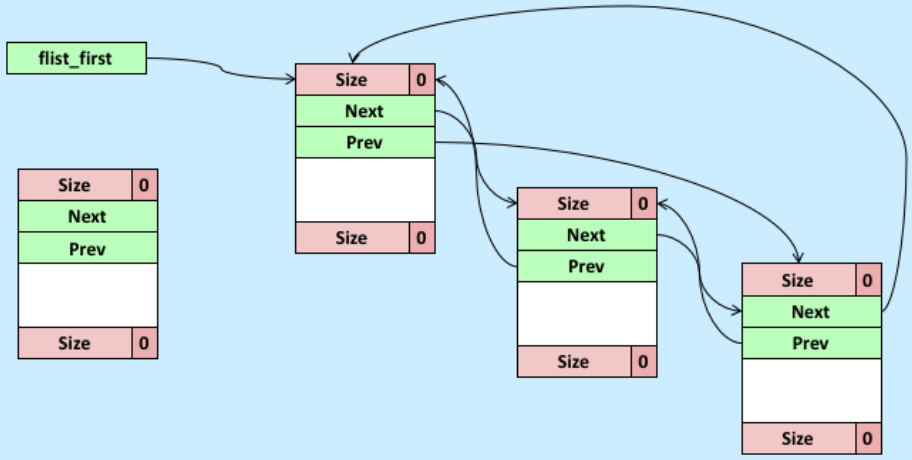

```
size_t block_prev_allocated(
    block_t *b) {
  return b->payload[-2] & 1;
}
```


Is Next Adjacent Block Free?

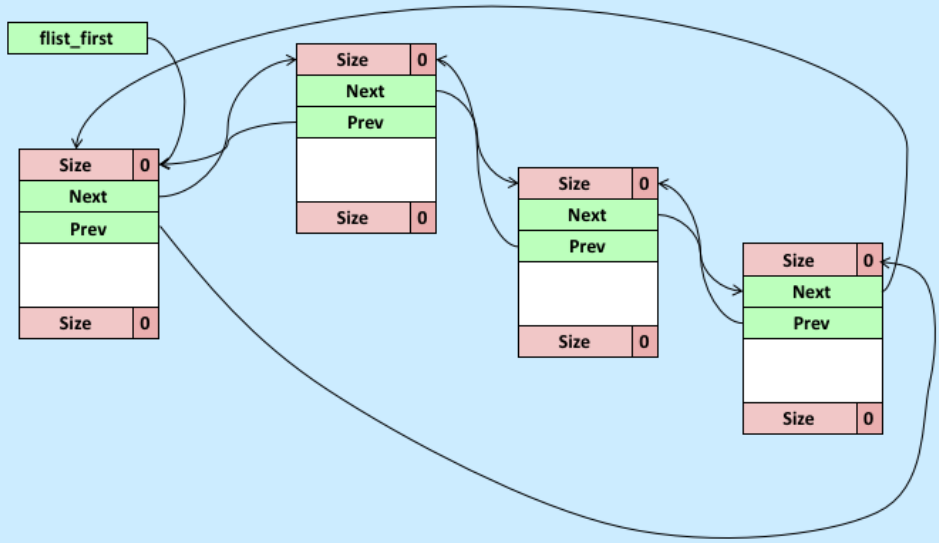



```
block_t *block_next(
    block_t *b) {
  return (block_t *)
    ((char *)b + block_size(b));
}

size_t block_next_allocated(
    block_t *b) {
  return block_allocated(
    block_next(b));
}
```


Adding a Block to the Free List (1)



Adding a Block to the Free List (2)



Accessing the Object

```
block_t *block_next_free(block_t *b) {
  return (block_t *)b->payload[0];
}

void block_set_next_free(block_t *b, block_t *next) {
  b->payload[0] = (size_t)next;
}

block_t *block_prev_free(block_t *b) {
  return (block_t *)b->payload[1];
}

void block_set_prev_free(block_t *b, block_t *next) {
  b->payload[1] = (size_t)next;
}
```


Insertion Code

```
void insert_free_block(block_t *fb) {
  assert(!block_allocated(fb));
  if (flist_first != NULL) {
    block_t *last =
      block_prev_free(flist_first);
    block_set_next_free(fb, flist_first);
    block_set_prev_free(fb, last);
    block_set_next_free(last, fb);
    block_set_prev_free(flist_first, fb);
  } else {
    block_set_next_free(fb, fb);
    block_set_prev_free(fb, fb);
  }
  flist_first = fb;
}
```


Performance

- Won't all the calls to the accessor functions slow things down a lot?
  - yes — not just a lot, but tons
- Why not use macros (#define) instead?
  - the textbook does this
  - it makes the code impossible to debug
    - » gdb shows only the name of the macro, not its body
- What to do????

Inline functions

```
static inline size_t block_size(
    block_t *b) {
  return b->size & -2;
}
```

- when debugging (–O0), the code is implemented as a normal function
  - » easy to debug with gdb
- when optimized (–O1, –O2), calls to the function are replaced with the body of the function
  - » no function-call overhead