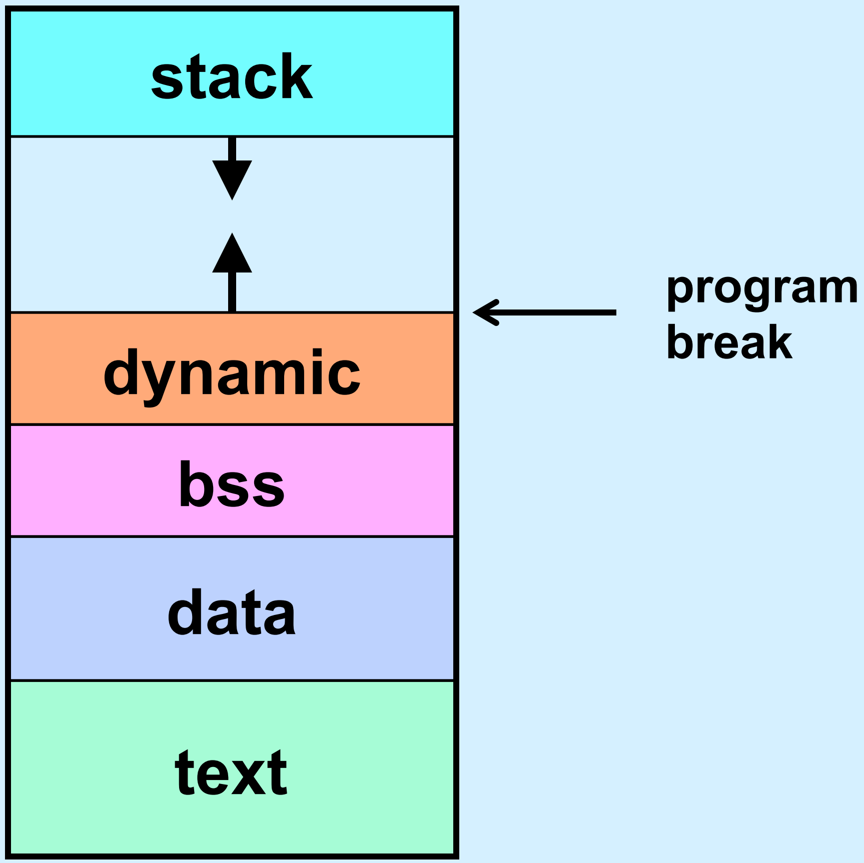
CS 33

Storage Allocation

The Unix Address Space



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

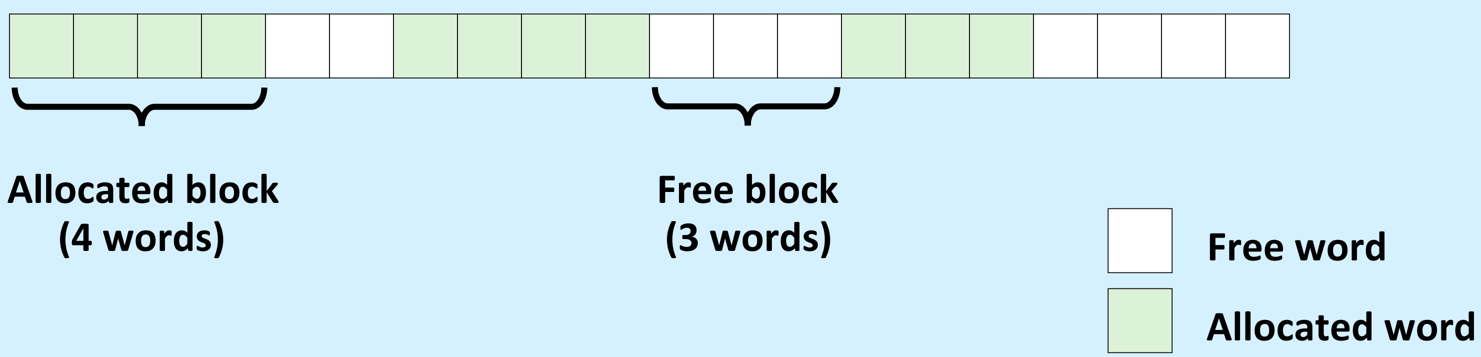
- **Strategy**
  - get a "chunk" of memory from the OS using *sbrk*
    - create pool of available storage, aka the "heap"
  - *malloc*, *calloc*, *realloc*, and *free* use this storage if possible
    - they manage the heap
  - if not possible, get more storage from OS
    - heap is made larger (by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

Dynamic Memory Allocation

- Allocator maintains heap as collection of variable sized ***blocks***, which are either ***allocated*** or ***free***
- Types of allocators
  - ***explicit allocator***: application allocates and frees space
    - » e.g., `malloc` and `free` in C
  - ***implicit allocator:*** application allocates, but does not free space
    - » e.g. garbage collection in Java, ML, and Racket

Assumptions Made in This Lecture

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



Allocation Example

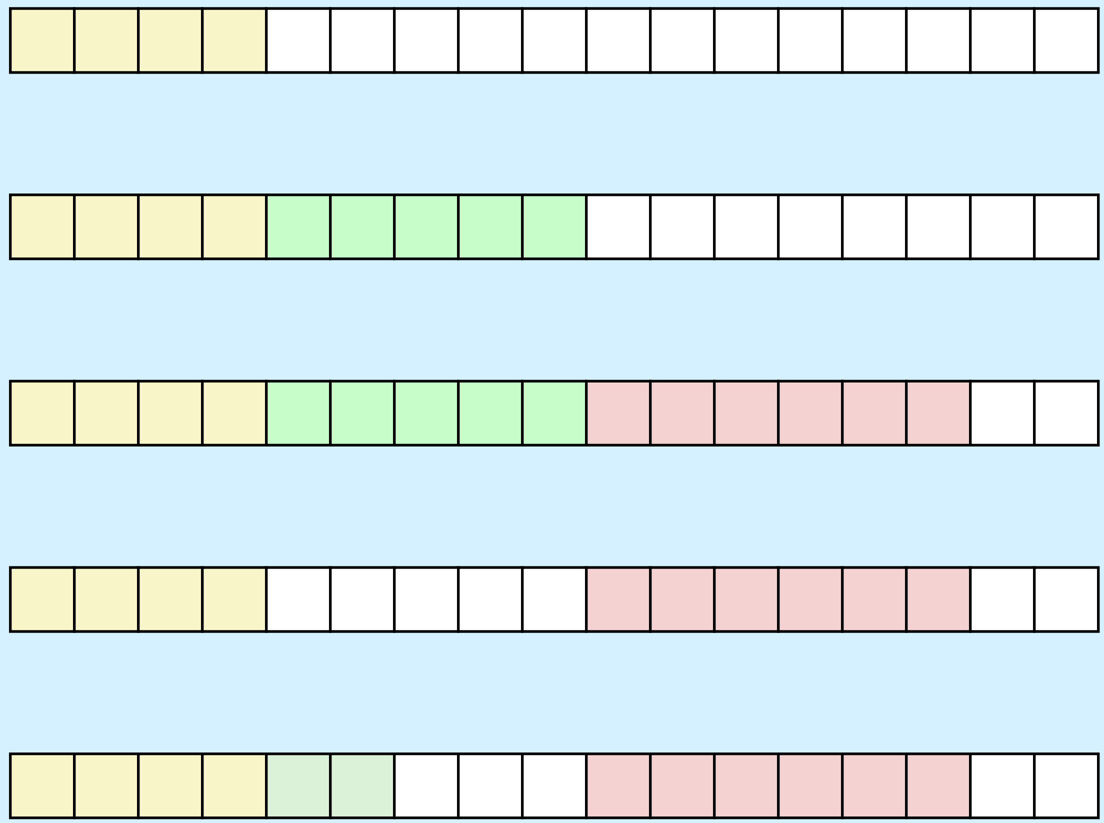
`p1 = malloc(4)`

`p2 = malloc(5)`

`p3 = malloc(6)`

`free(p2)`

`p4 = malloc(2)`

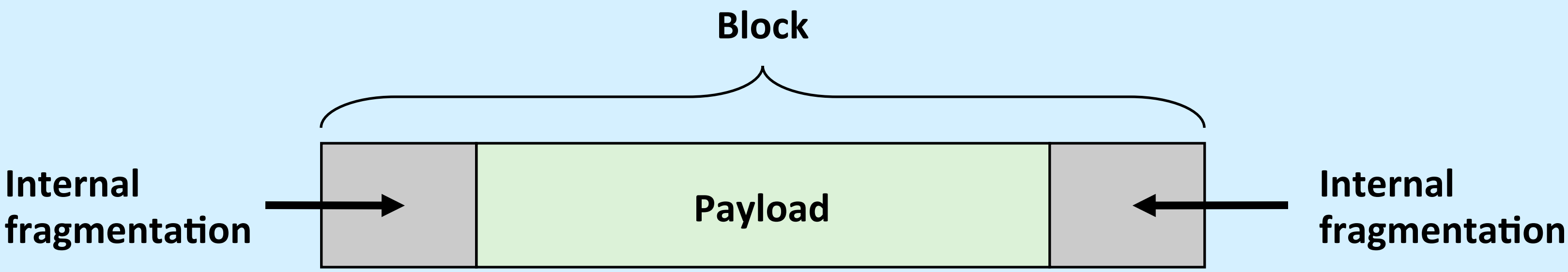


Constraints

- **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

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

External Fragmentation

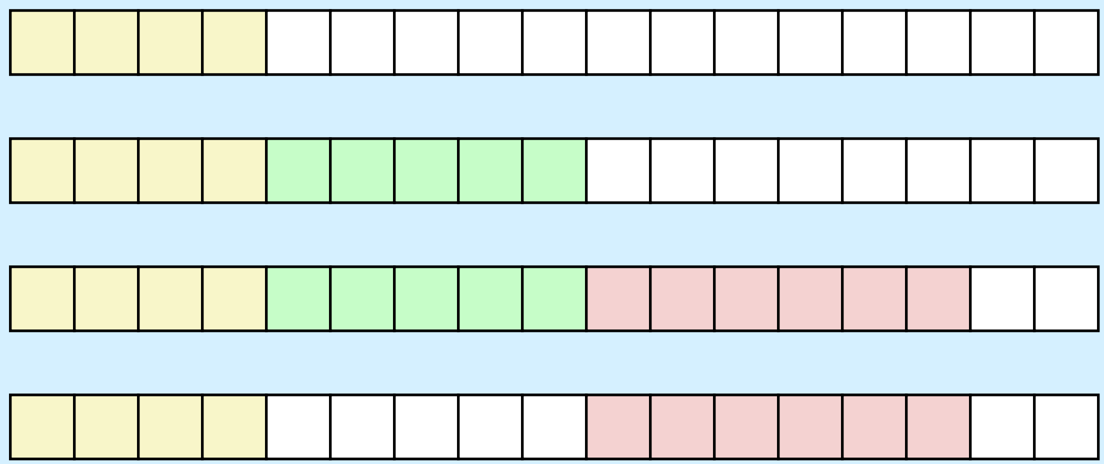
- **Occurs when there is enough aggregate heap memory, but no single free block is large enough**

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

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

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

```
free(p2)
```



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

Oops! (what would happen now?)

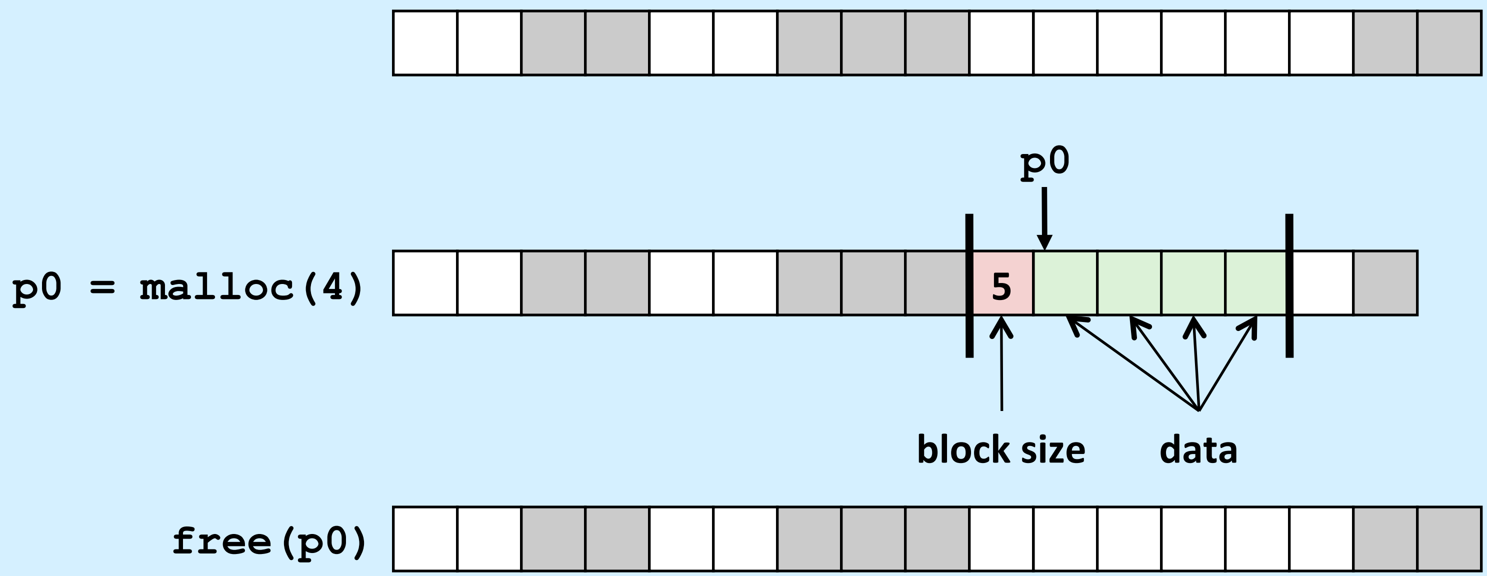
- **Depends on the pattern of future requests**
  - **thus, difficult to measure**

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?

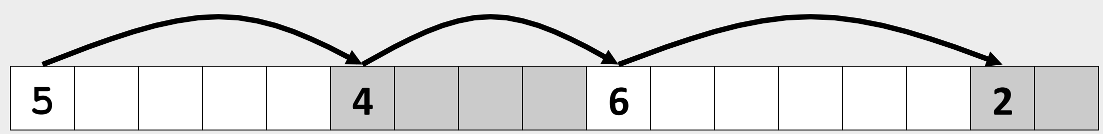
Knowing How Much to Free

- **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

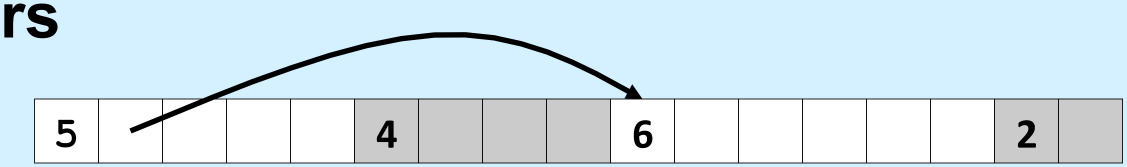


Keeping Track of Free Blocks

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



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

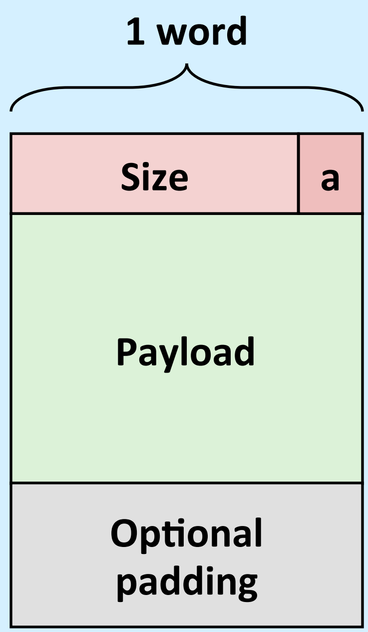


- **Method 3: *Segregated free list***
  - **different free lists for different size classes**
- **Method 4: *Blocks sorted by size***
  - **can use a balanced tree (e.g. red-black tree) with pointers within each free block, and the length used as a key**

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

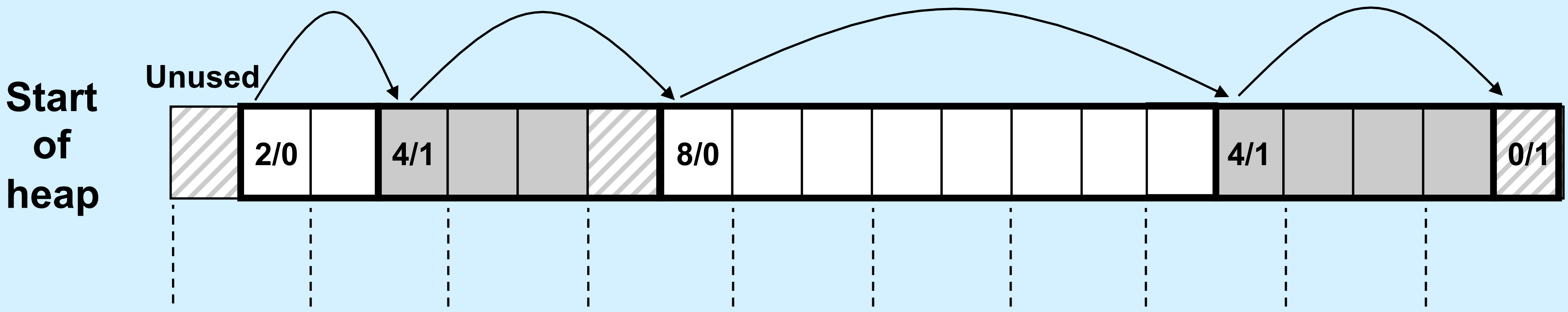


a = 1: Allocated block
a = 0: Free block

Size: block size

Payload: application data
(allocated blocks only)

Detailed Implicit Free-List Example



Double-word aligned

Allocated blocks: shaded
Free blocks: unshaded
Headers: labeled with size in bytes/allocated bit

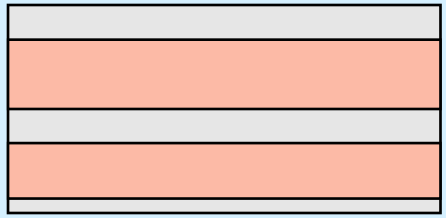
Implicit List: Finding a Free Block

- ***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

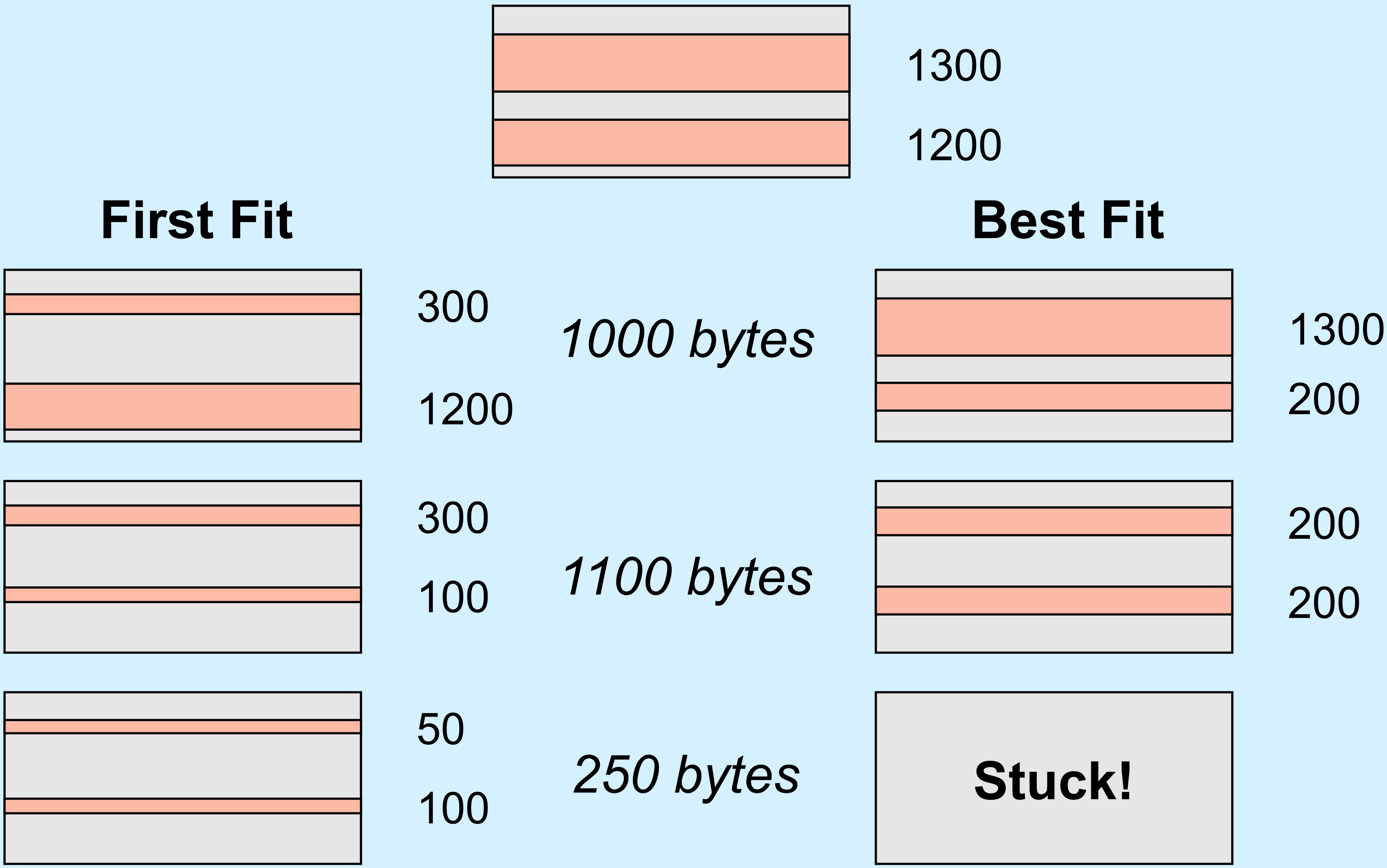
Quiz 1



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

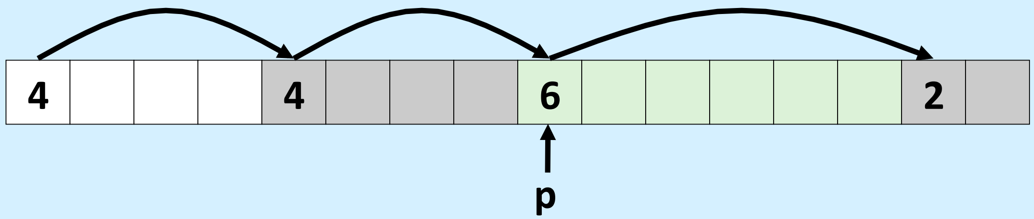
Allocation



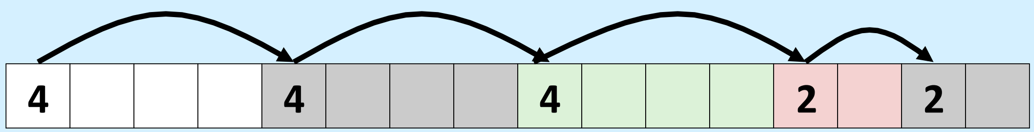
1300
1200
First Fit
Best Fit
300
1200
1000 bytes
1300
200
300
100
1100 bytes
200
200
50
100
250 bytes
Stuck!

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)`



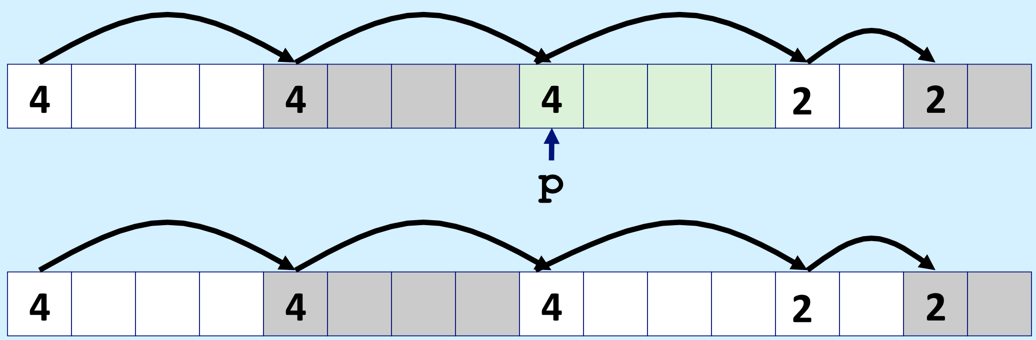

```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;  // round up to even
  int oldsize = *p & -2;                // mask out low bit
  *p = newsize | 1;                     // set new length
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;   // set length in remaining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

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"



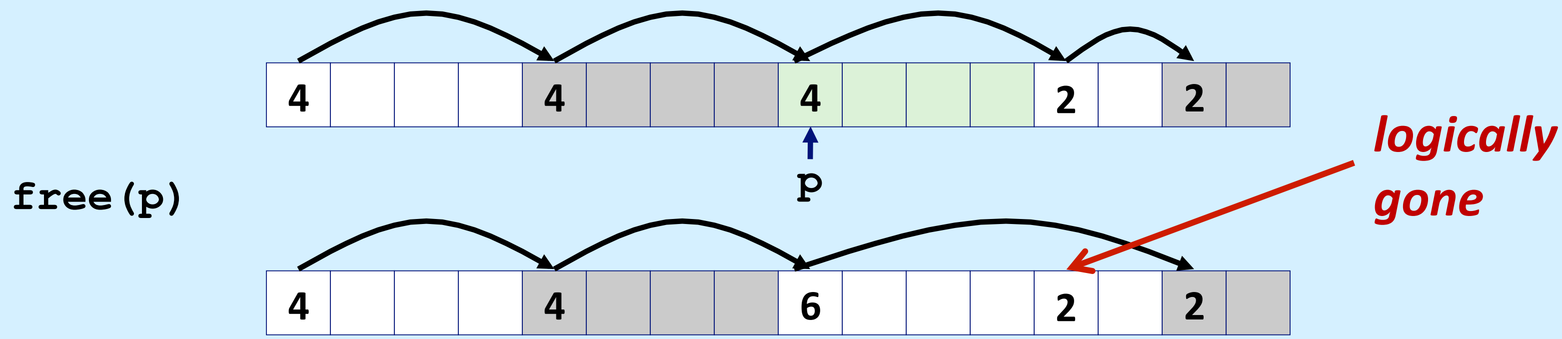
`free(p)`

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

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

Implicit List: Coalescing

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

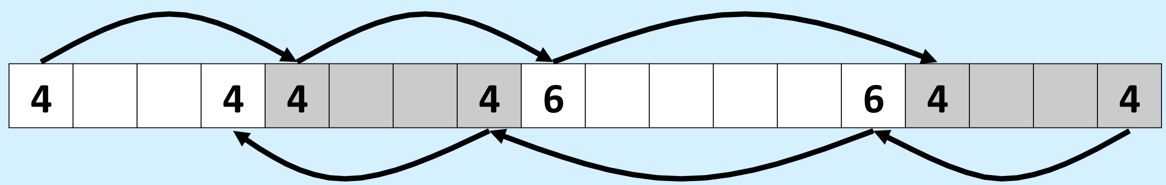


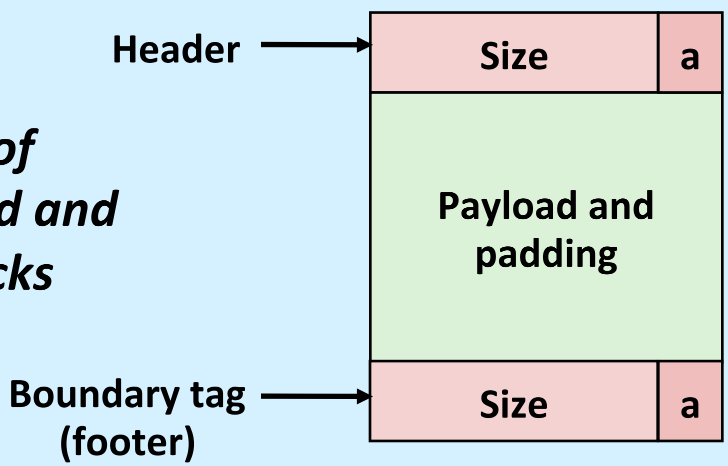

```
void free_block(ptr p) {
    *p = *p & -2;            // clear allocated flag
    next = p + *p;           // find next block
    if ((*next & 1) == 0)
      *p = *p + *next;       // add to this block if
}                            //    not allocated
```

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

Implicit List: Bidirectional Coalescing

- ***Boundary tags*** [Knuth73]
  - replicate size/allocated word at “bottom” (end) of free blocks
  - allows us to traverse the “list” backwards, but requires extra space
  - important and general technique!





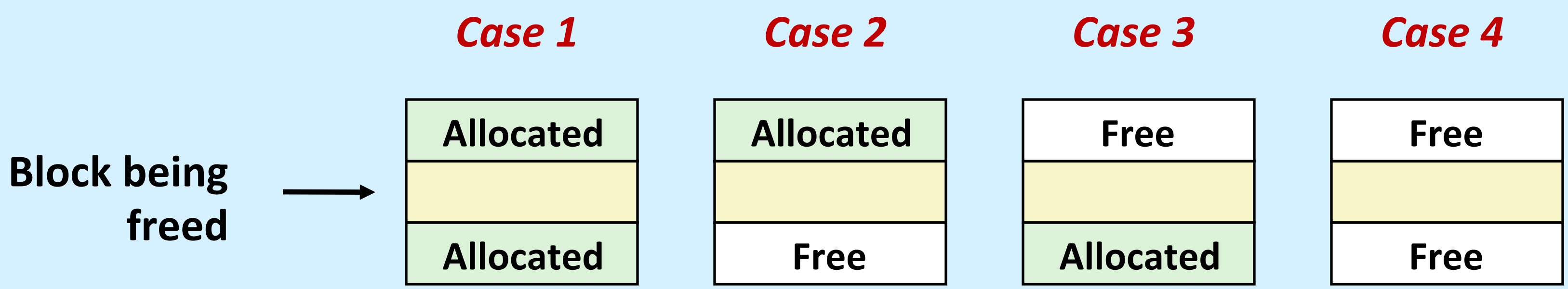
Format of allocated and free blocks

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

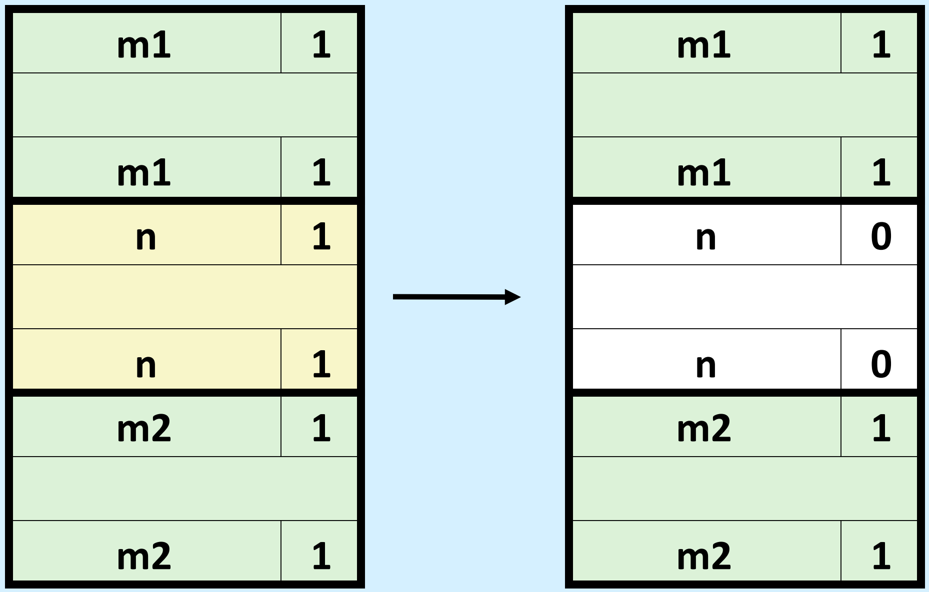
Size: Total block size

Payload: Application data (allocated blocks only)

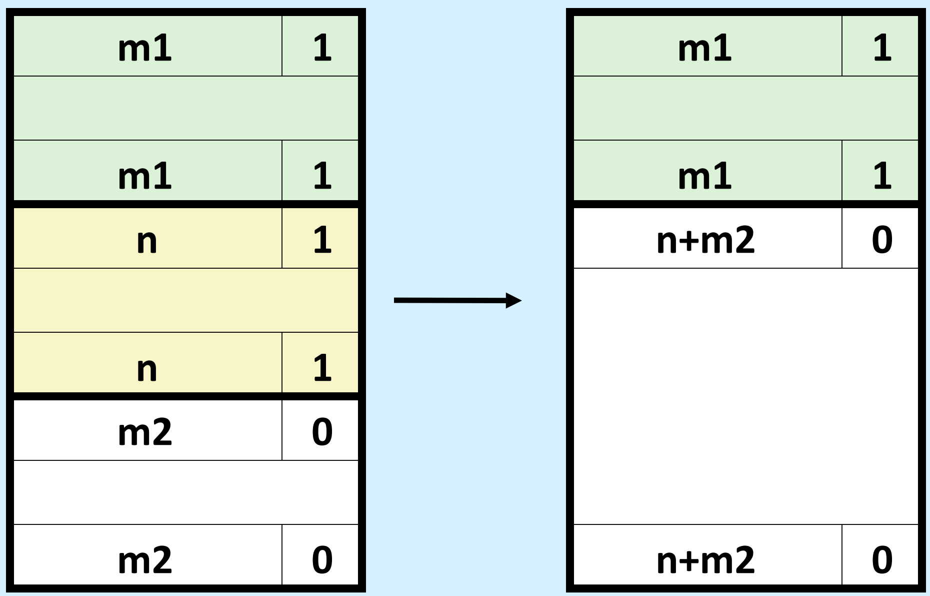
Constant Time Coalescing



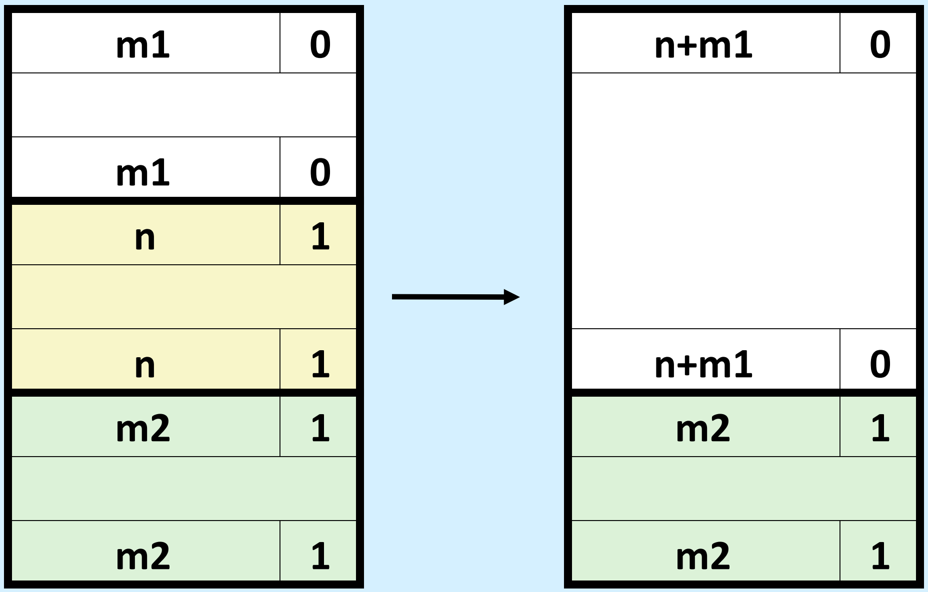
Constant Time Coalescing (Case 1)



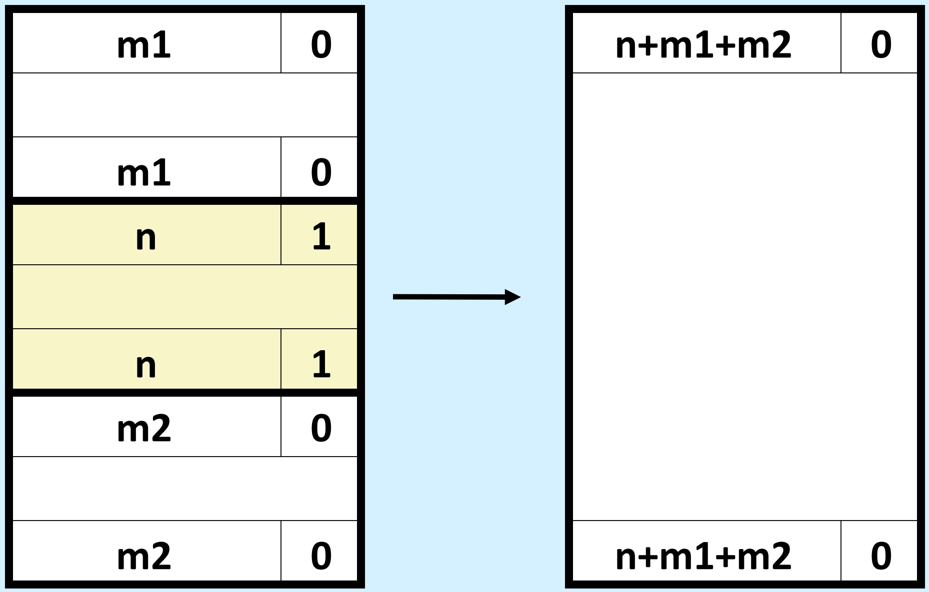
Constant Time Coalescing (Case 2)



Constant Time Coalescing (Case 3)



Constant Time Coalescing (Case 4)

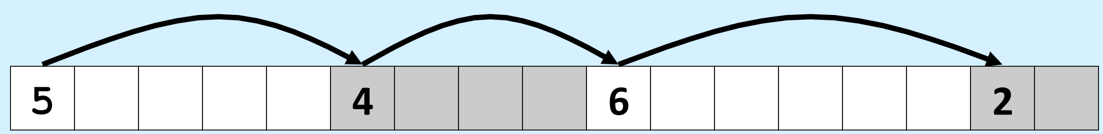


Implicit Lists: Summary

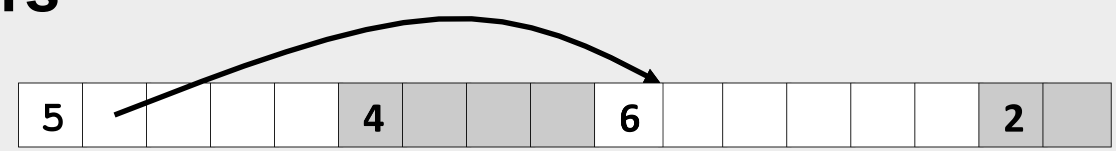
- Implementation: very simple
- Allocate cost:
  - linear time worst case
- Free cost:
  - constant time worst case
  - even with coalescing
- Memory usage:
  - will depend on placement policy
  - first-fit, next-fit or best-fit
- Not used in practice for `malloc/free` because of linear-time allocation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

Keeping Track of Free Blocks

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

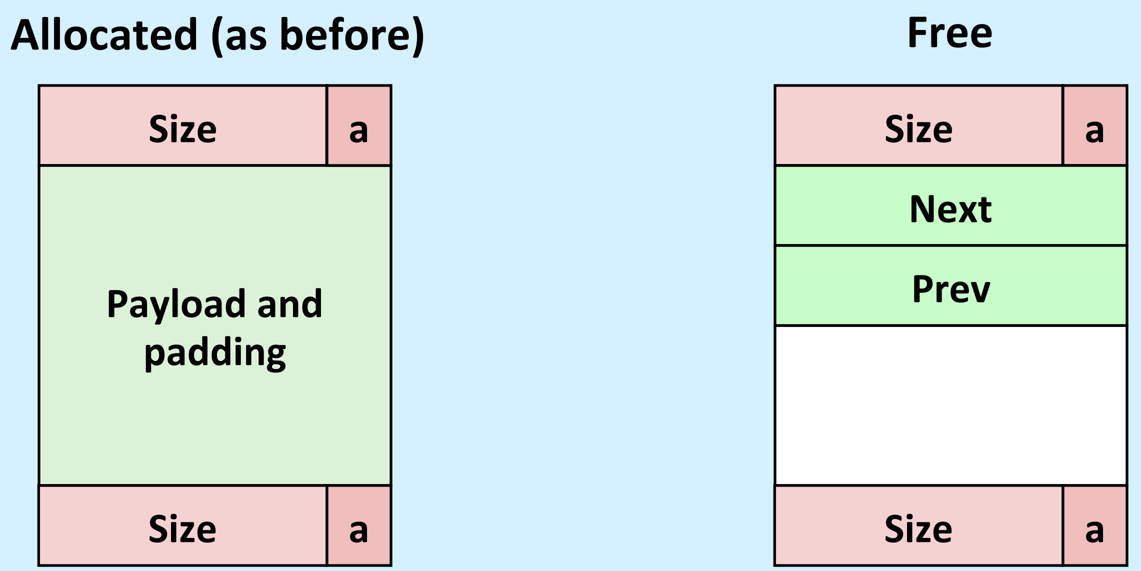


- Method 2: ***explicit free list*** among the free blocks using pointers



- 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

Explicit Free Lists

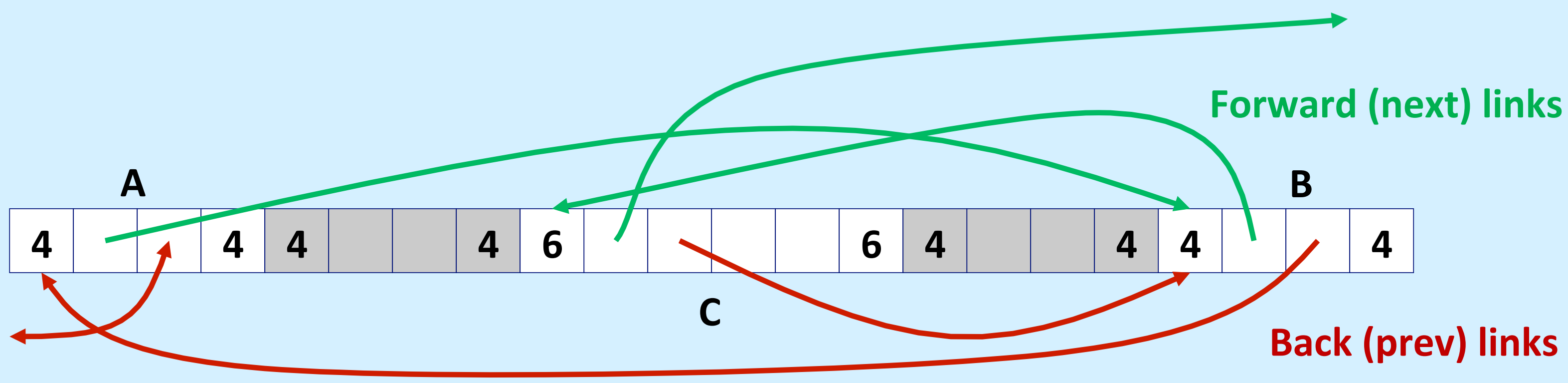


- Maintain list(s) of *free* blocks, not *all* blocks
  - the "next" free block could be anywhere
    - so we need to store forward/back pointers, not just sizes
    - luckily we track only free blocks, so we can use payload area
  - still need boundary tags for coalescing

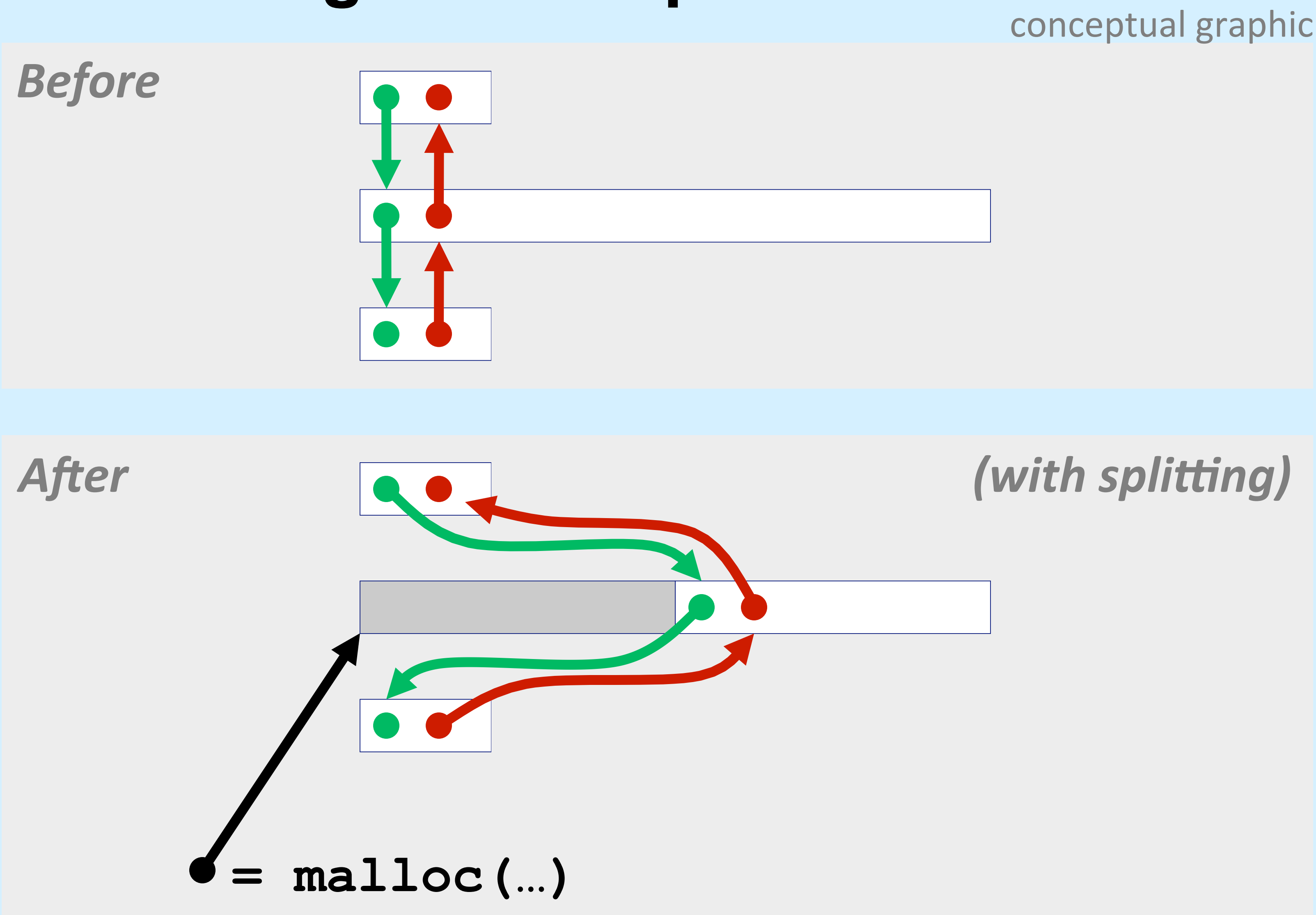
Explicit Free Lists

- **Logically:**

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



Allocating From Explicit Free Lists



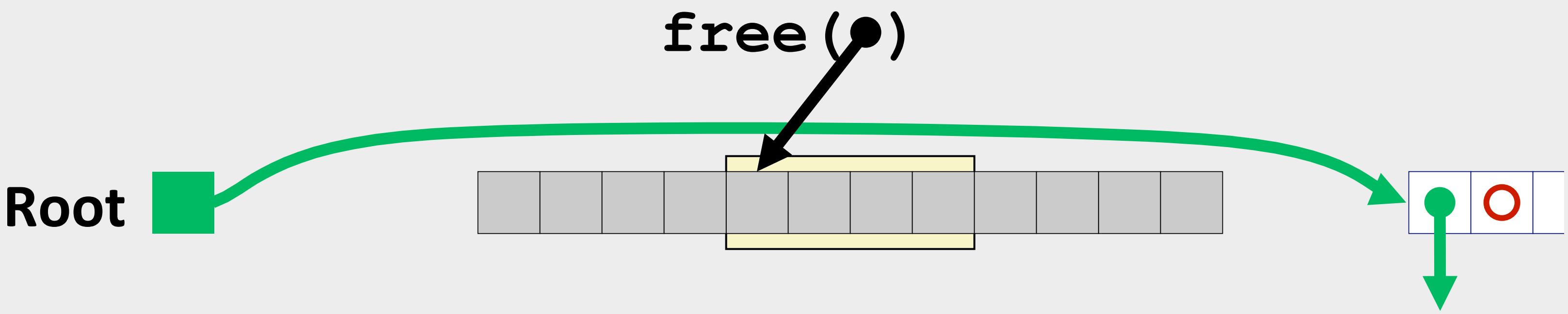
Freeing With Explicit Free Lists

- ***Insertion policy***: where in the free list do you put a newly freed block?
  - LIFO (last-in-first-out) policy
    - » insert freed block at the beginning of the free list
    - » *pro:* simple and constant time
    - » *con:* studies suggest fragmentation is worse than address ordered
  - address-ordered policy
    - » Insert freed blocks so that free list blocks are always in address order:

      addr(prev) < addr(curr) < addr(next)
    - » *con:* requires search
    - » *pro:* studies suggest fragmentation is lower than LIFO

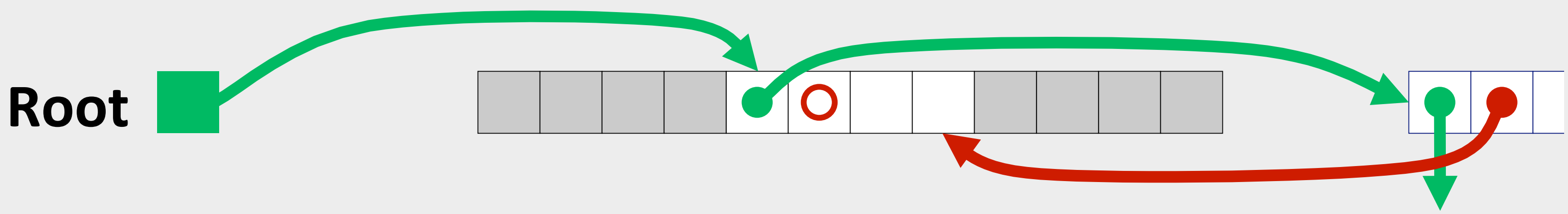
Freeing With a LIFO Policy (Case 1)

Before



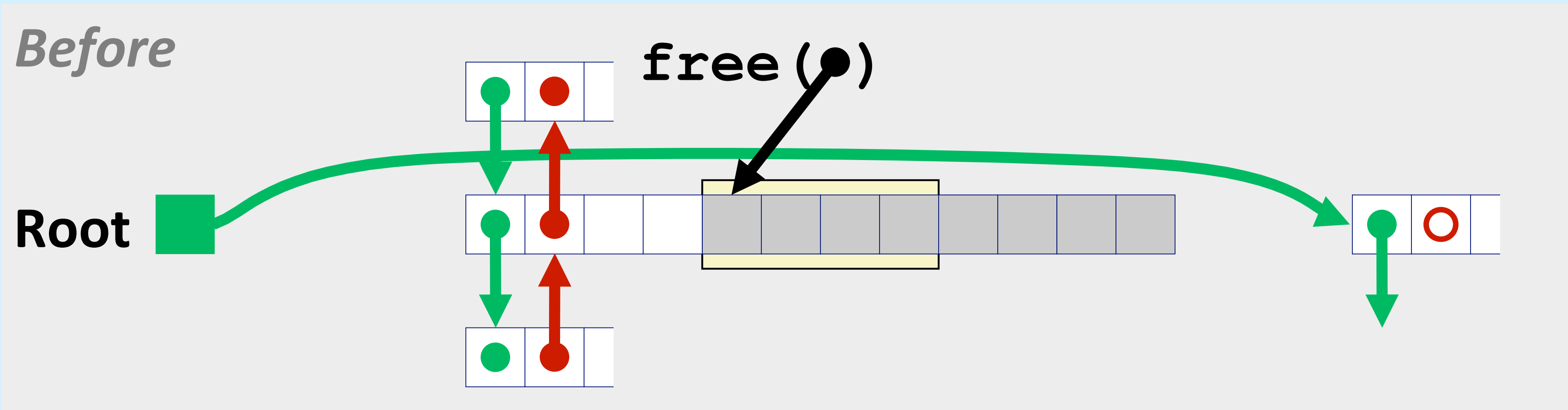
- Insert the freed block at the root of the list

After

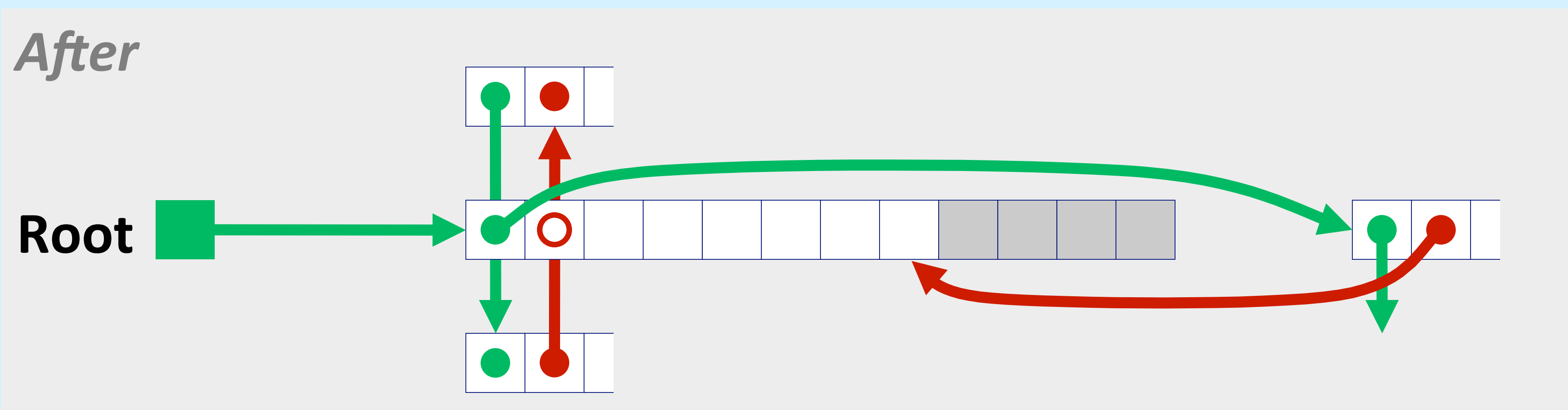


Freeing With a LIFO Policy (Case 2)

conceptual graphic

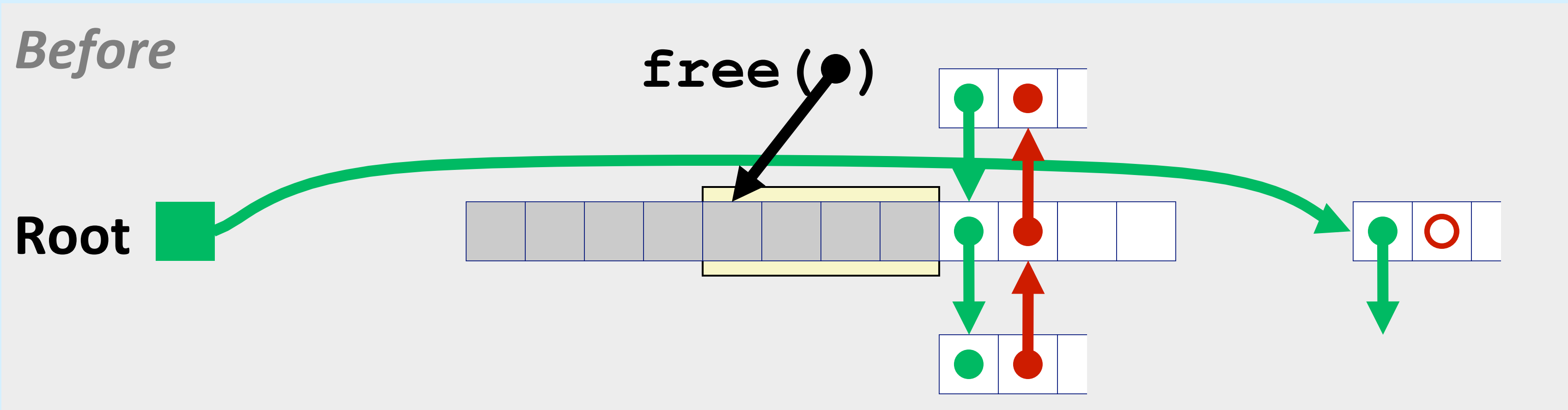


- **Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list**

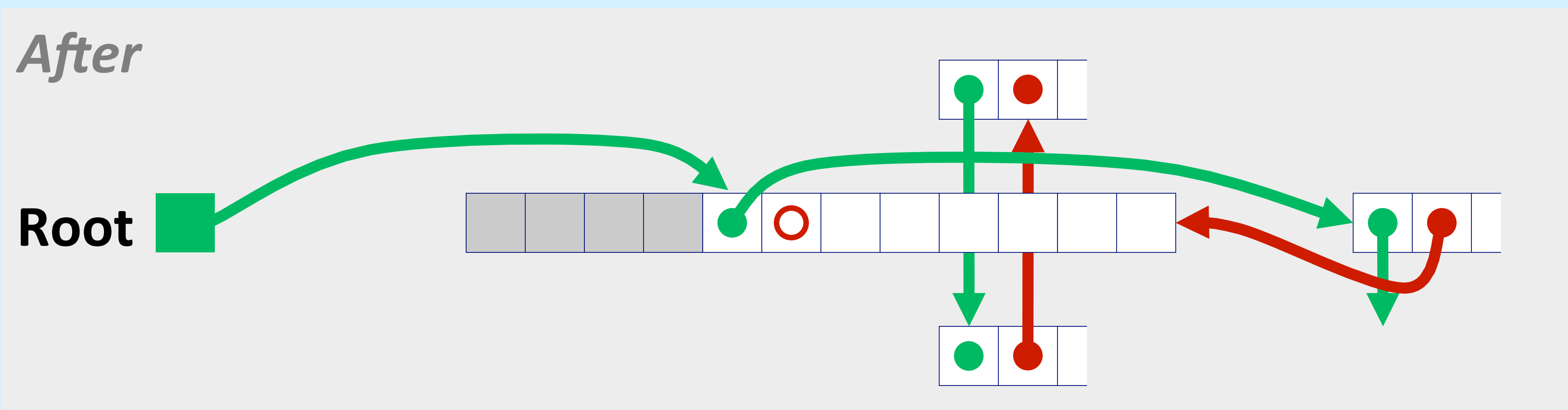


Freeing With a LIFO Policy (Case 3)

conceptual graphic

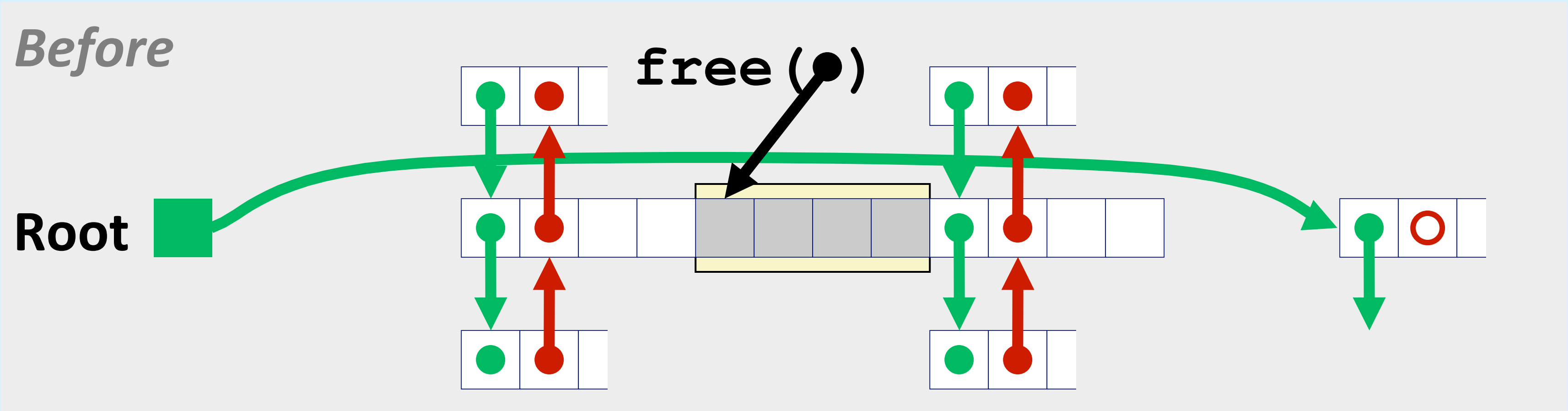


- **Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list**

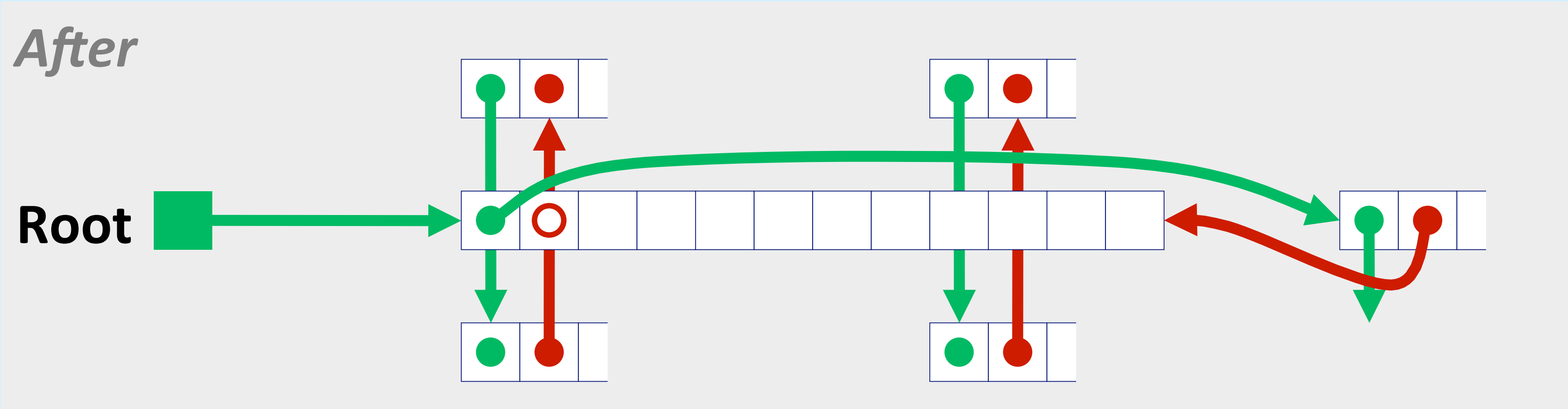


Freeing With a LIFO Policy (Case 4)

conceptual graphic



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



Explicit List Summary

- Comparison to implicit list:
  - allocate 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 needs to splice blocks in and out of the list
  - some extra space for the links (2 extra words needed for each block)
- Most common use of linked lists is in conjunction with segregated free lists
  - keep multiple linked lists of different size classes, or possibly for different types of objects

Quiz 2

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

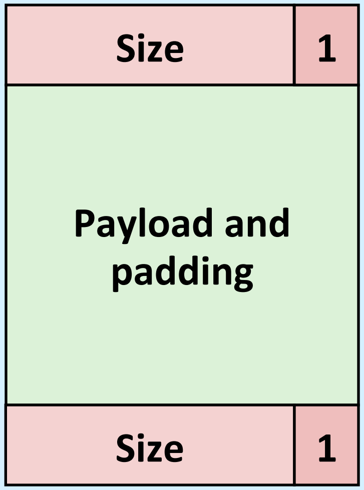
We are running an application with modest memory demands. Which allocation strategy is likely to result in better performance (in terms of time) for the application:

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;
```

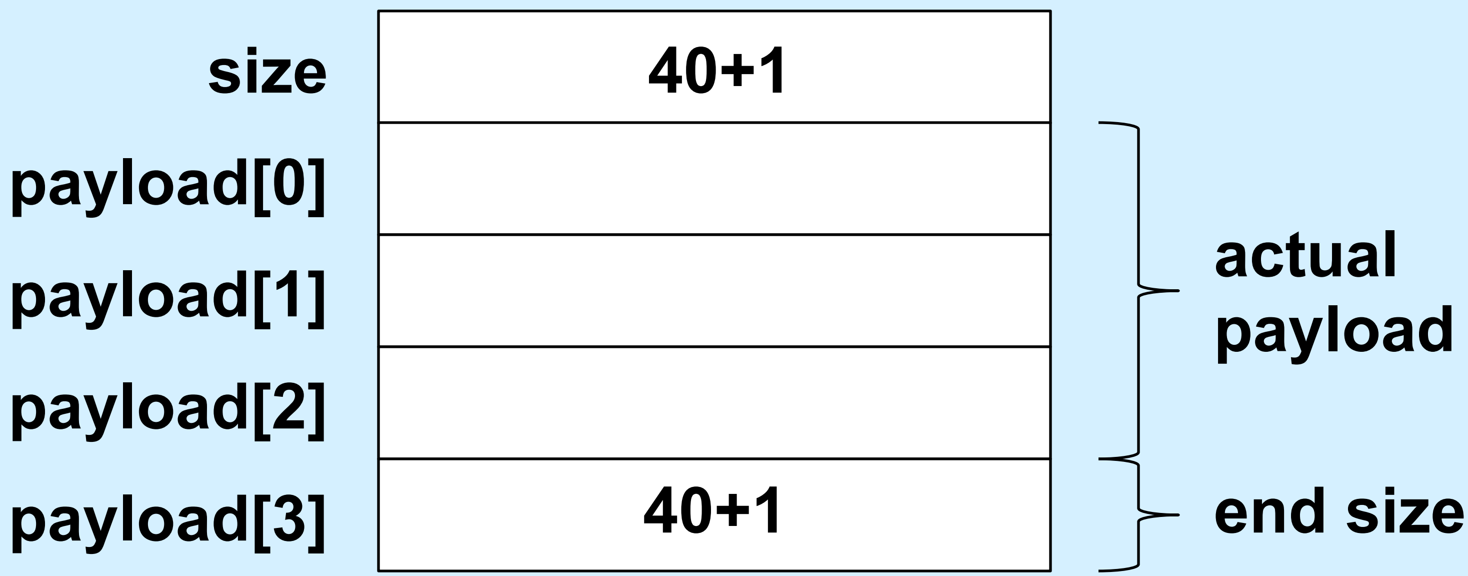
```
typedef struct free_block {
  long size;
  struct free_block *next;
  struct free_block *prev;
  long filler[size/8 - 4];
  long end_size;
} free_block_t;
```

Overcoming C

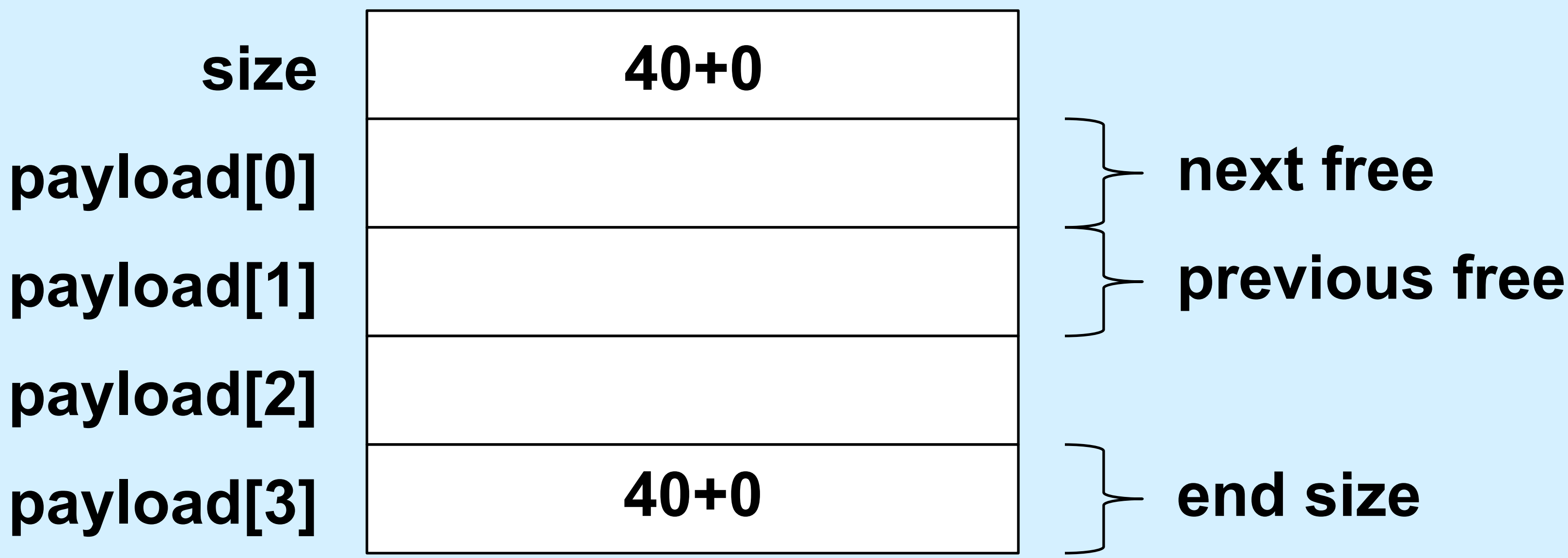
- **Think objects**
  - a block is an object
    - » opaque to the outside world
  - define accessor functions to get and set its contents

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


Allocated Block

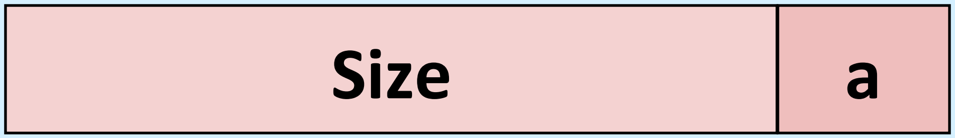


Free Block



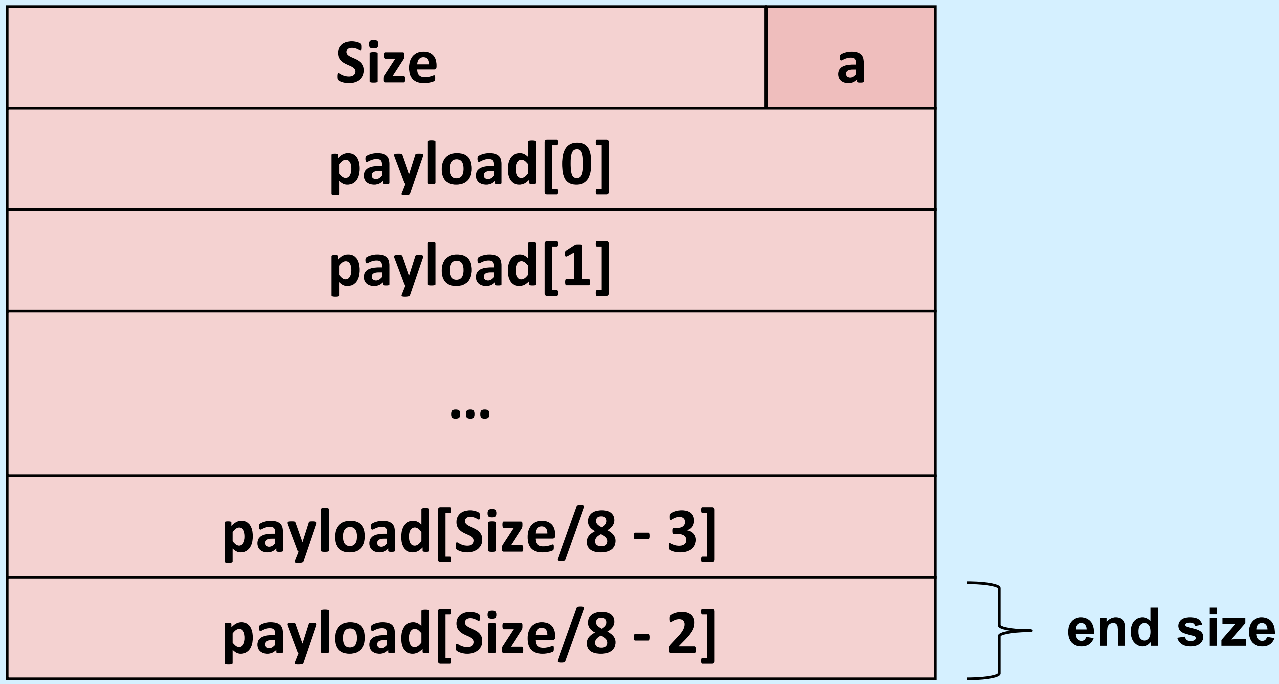
- In general, end size is at *payload[size/8 – 2]*

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

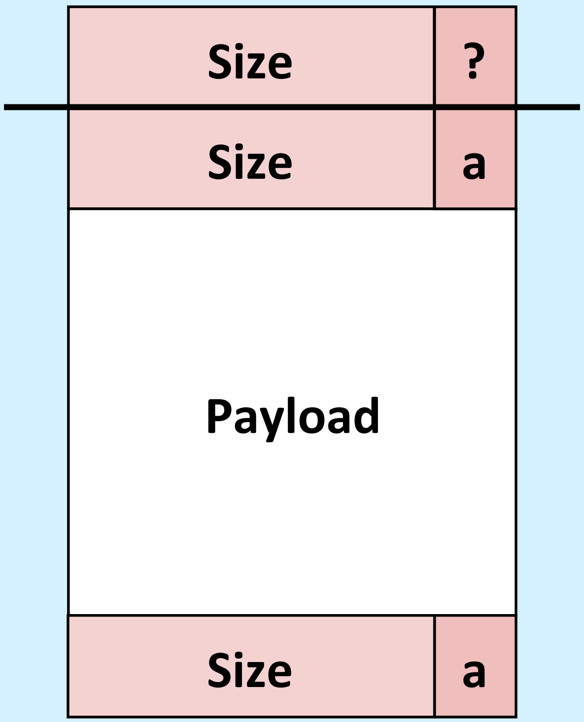



```
size_t *block_end_tag(block_t *b) {
  return &b->payload[b->size/8 - 2];
}
```

Setting the Size

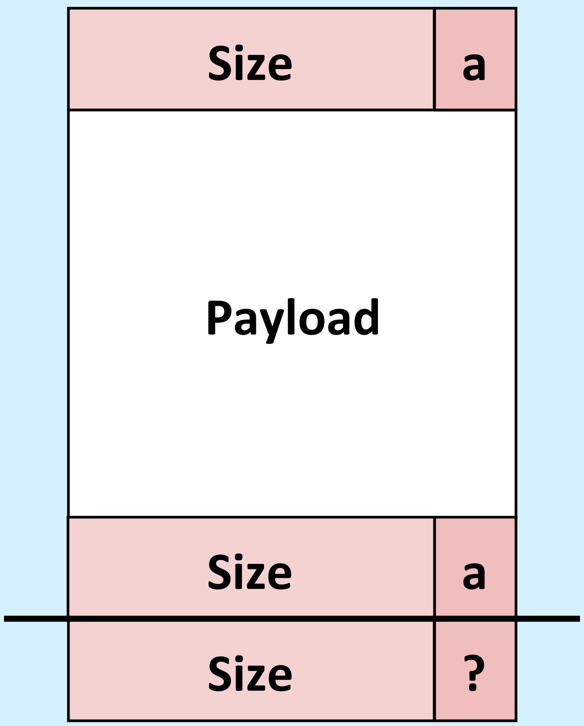
```
void block_setsize(block_t *b, size_t size) {
  assert(!(size & 7));          // multiple of 8
  size |= block_allocated(b);   // preserve alloc bit
  b->size = size;
  *block_end_tag(b) = size;
}

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?



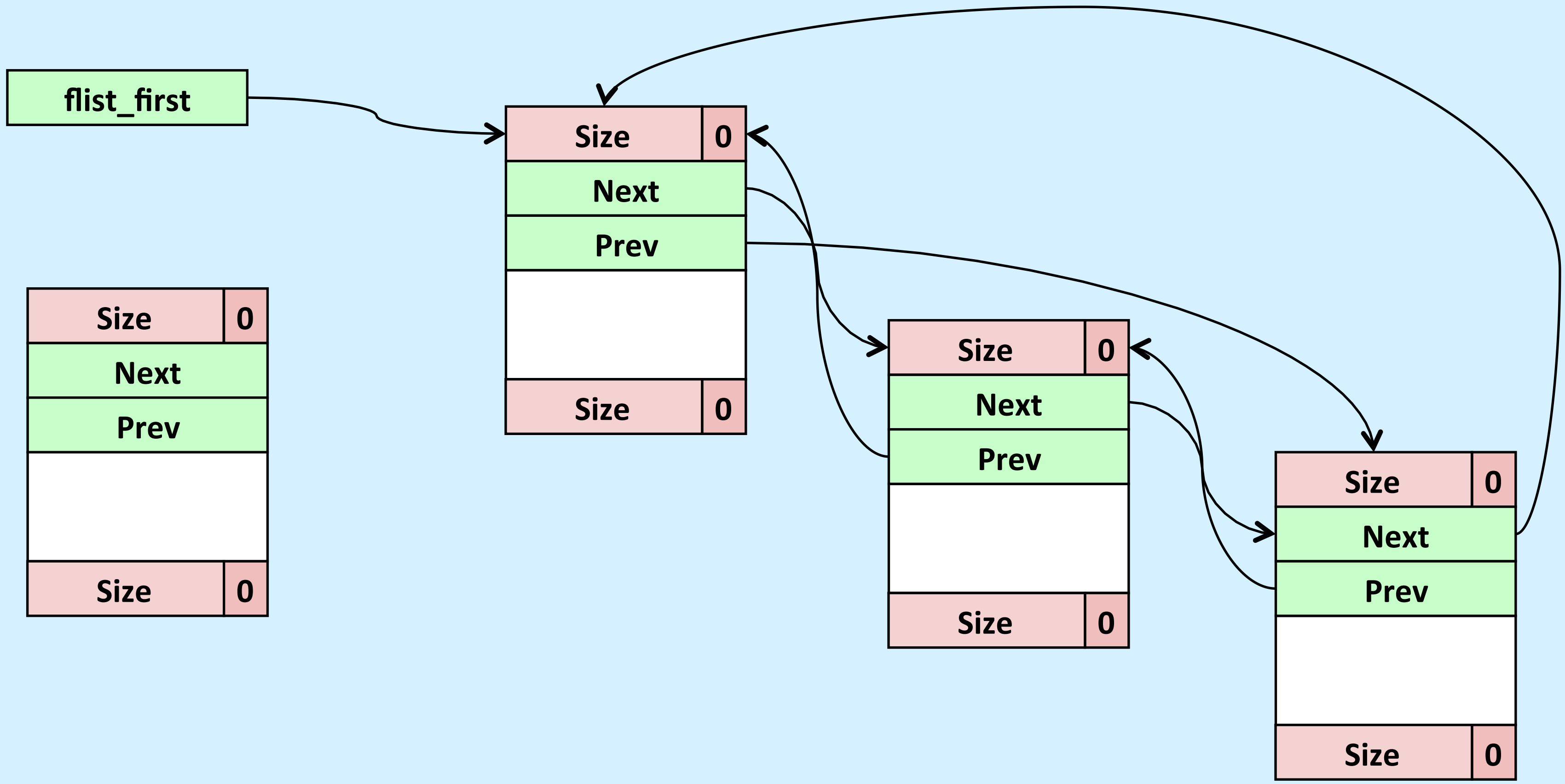

```
size_t block_prev_allocated(
    block_t *b) {
  return b->payload[-2] & 1;
}
```


Is Next Adjacent Block Free?

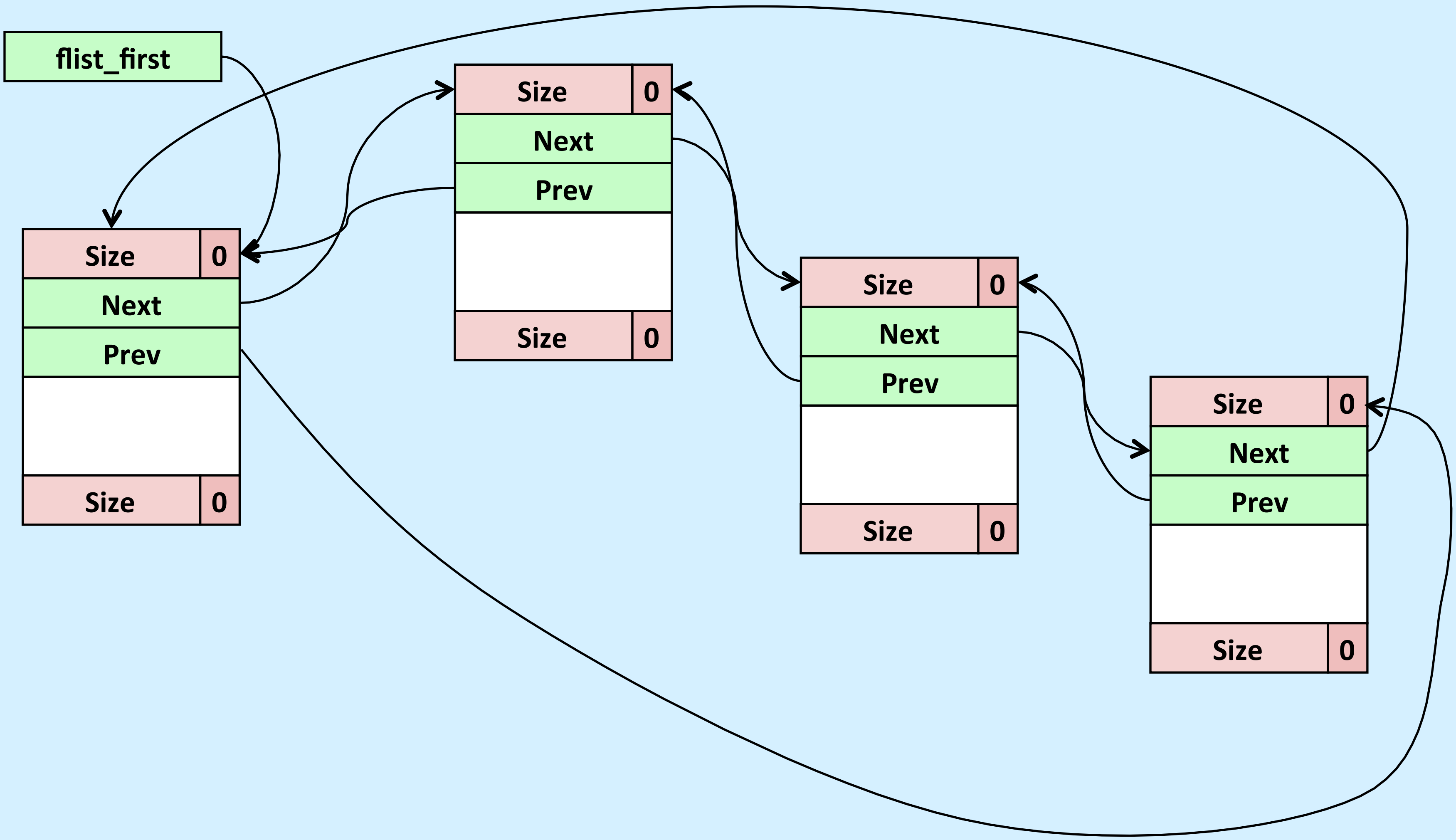



```
block_t *block_next(
    block_t *b) {
  return (block_t *)
    ((char *)b + block_size(b));
}

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