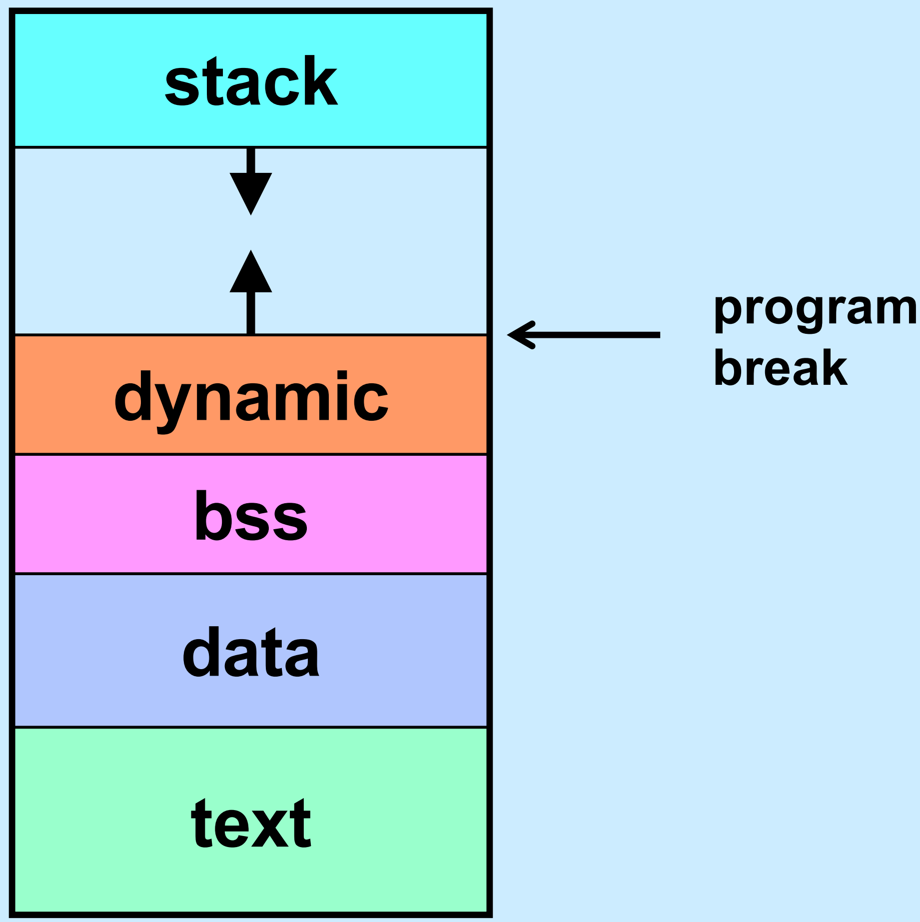
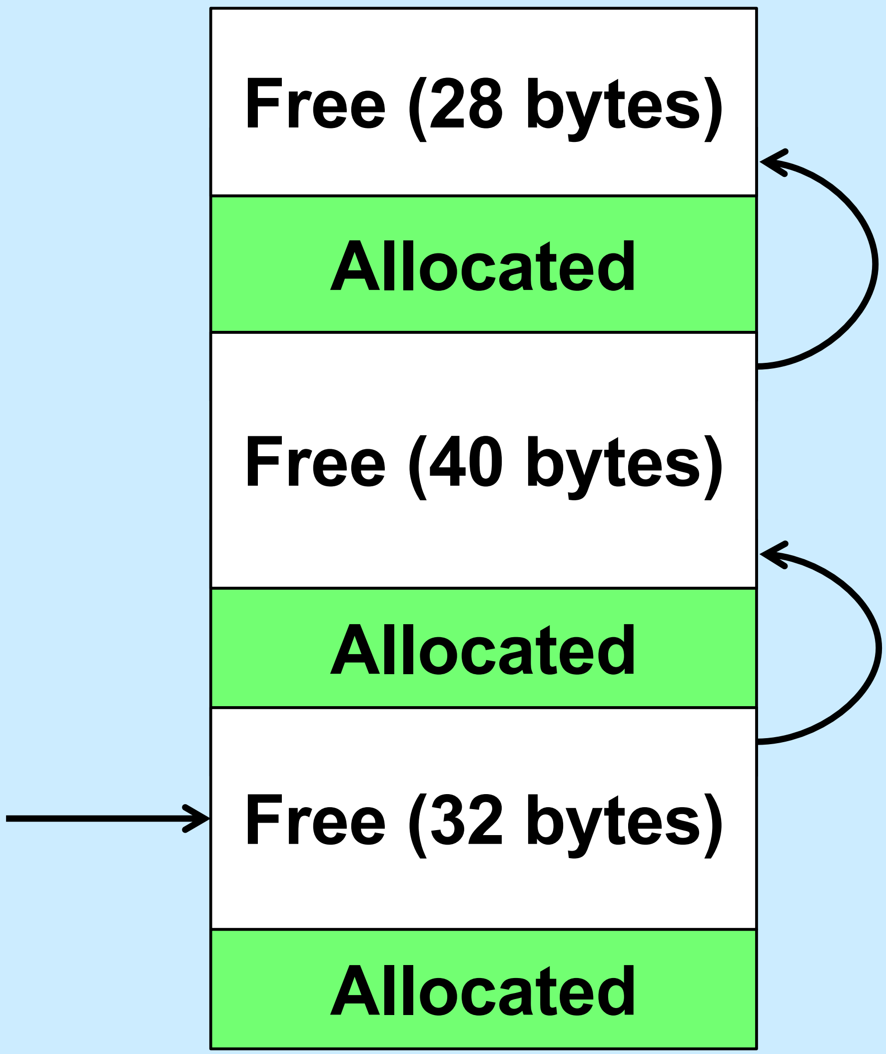
CS 33

Storage Allocation

The Unix Address Space



Finding the Right Free Block



`malloc(24)`

- Search strategies
  - first fit
  - best fit

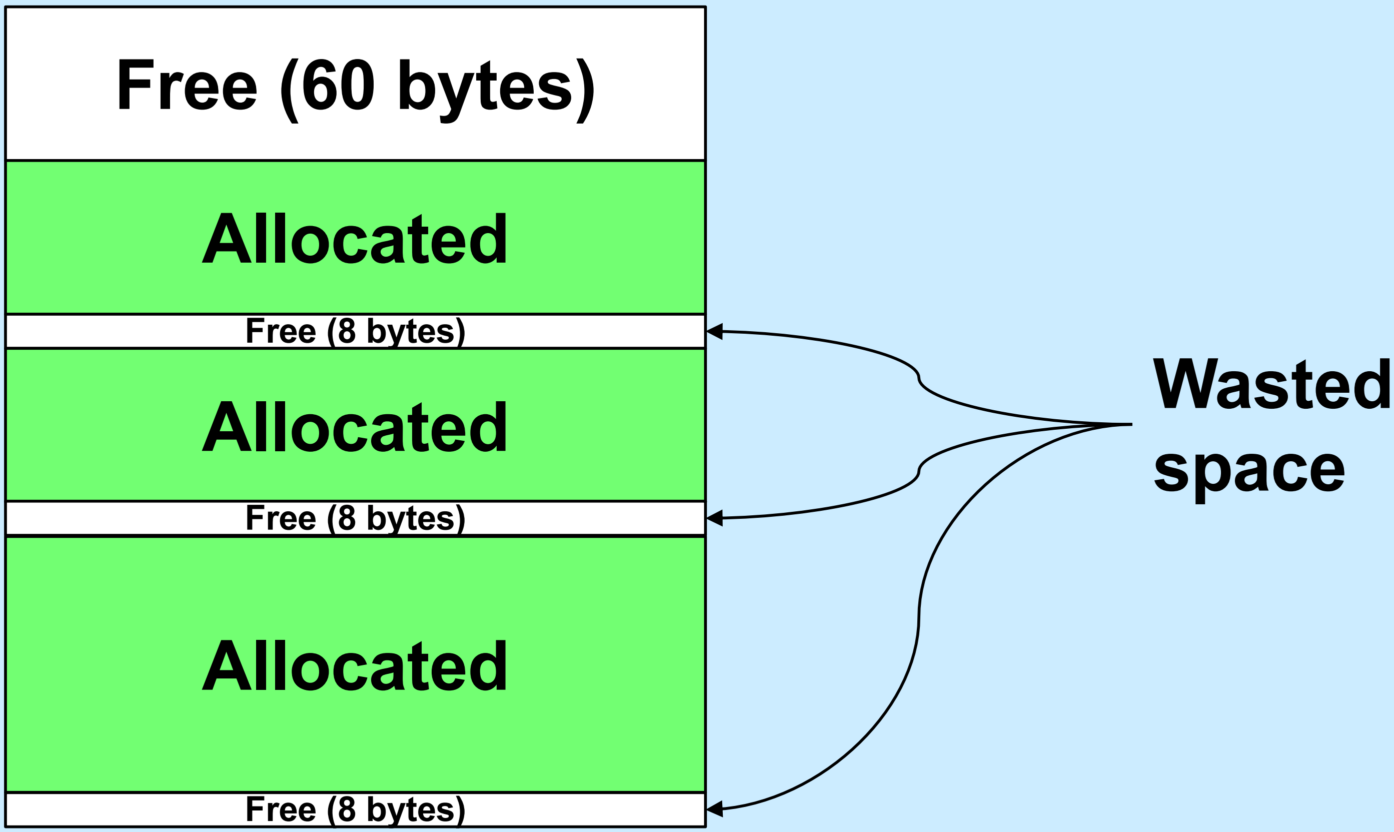
Some Observations

- **Best fit**
  - perhaps leaves behind chunks that are too small to be of use
  - requires linear time (in size of free list) for malloc
- **First fit**
  - small chunks congregate at beginning of free list
  - upper bound of linear time for malloc, but often much less

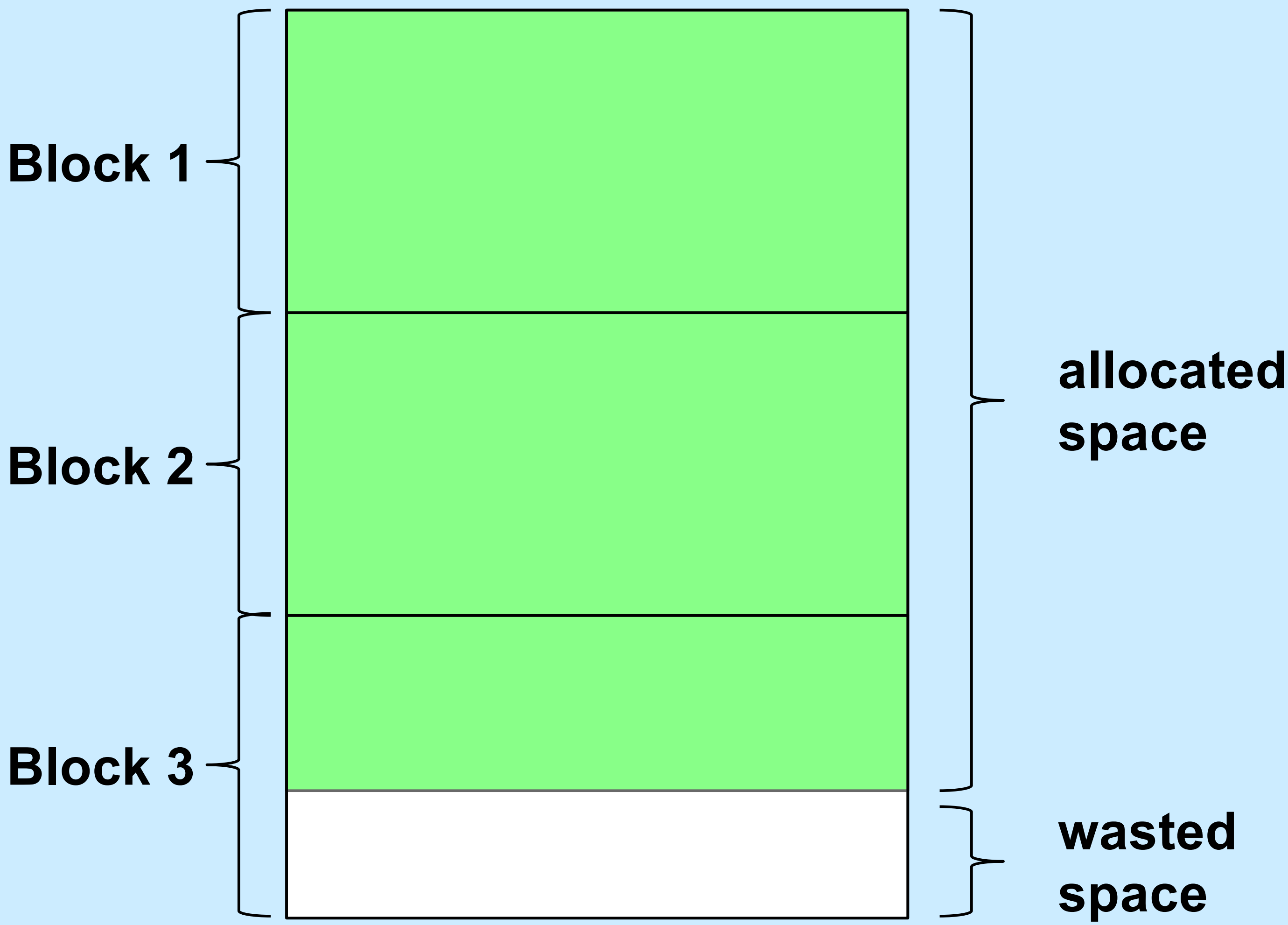
Fragmentation

- Fragmentation refers to the wastage of memory due to our allocation policy
- Two sorts
  - external fragmentation
  - internal fragmentation

External Fragmentation



Internal Fragmentation



Variations

- **Next fit**
  - like first fit, but the next search starts where the previous ended
- **Worst fit**
  - always allocate from largest free block
    - perhaps reduces the number of “too small” blocks
- **Free-list insertion**
  - LIFO
    - easy to do
    - O(1)
  - ordered insertion
    - O(n)

Quiz 1

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) first-fit with LIFO insertion

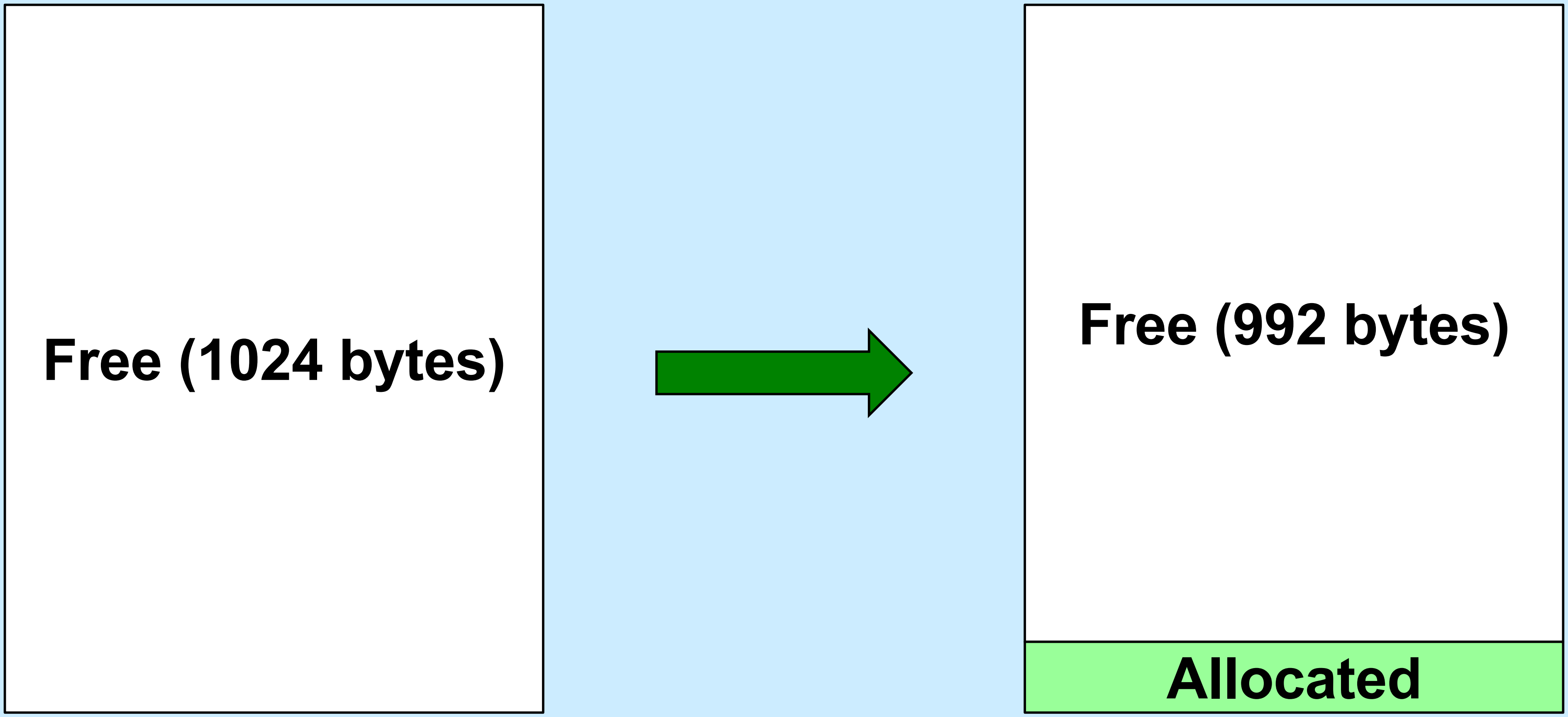
b) first-fit with ordered insertion

c) best-fit

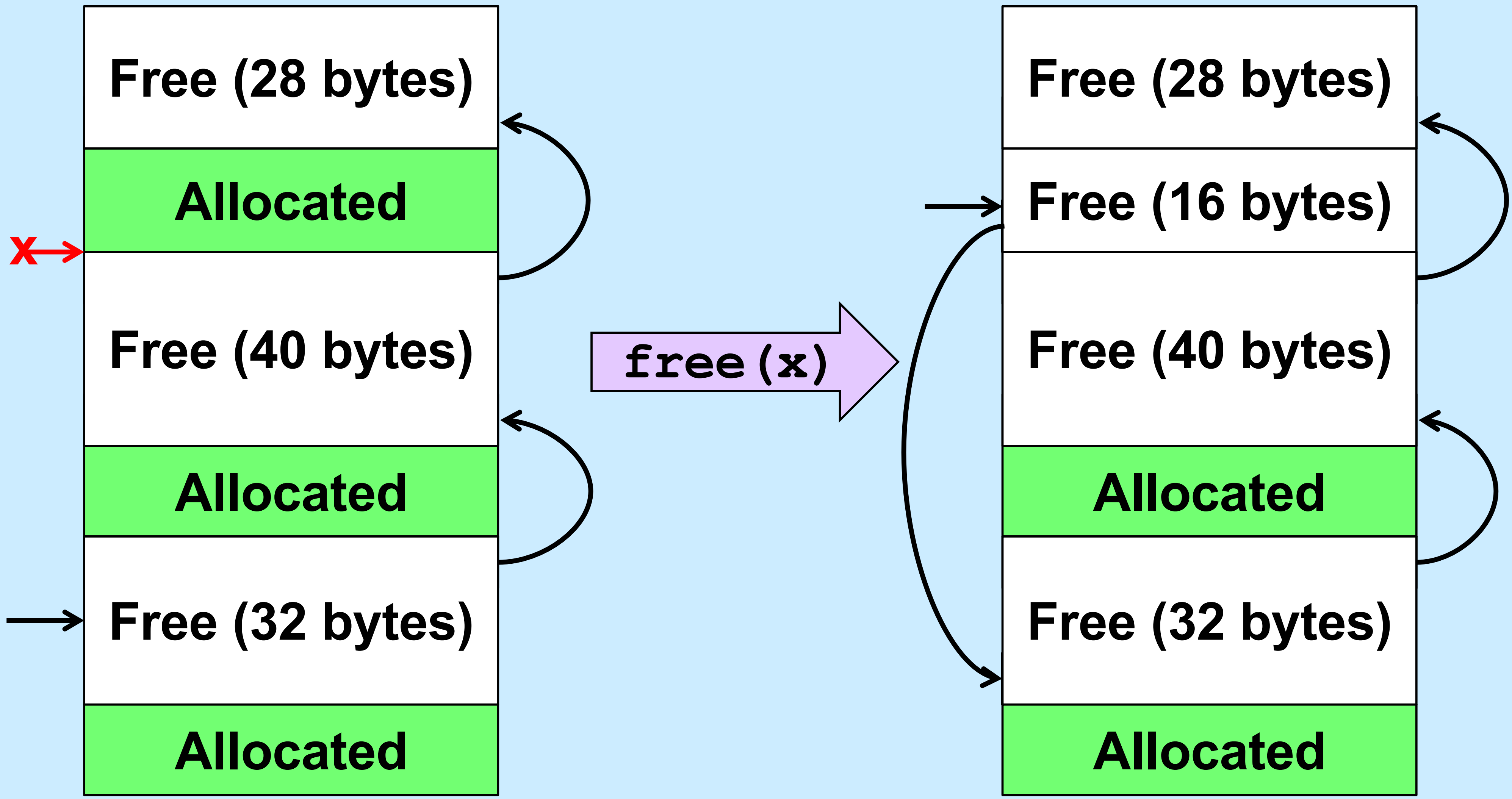
A Problem

- A malloc request is for a block of 32 bytes
- The block found on the free list is 1024 bytes long
- Should malloc return a pointer to the entire 1024-byte block?

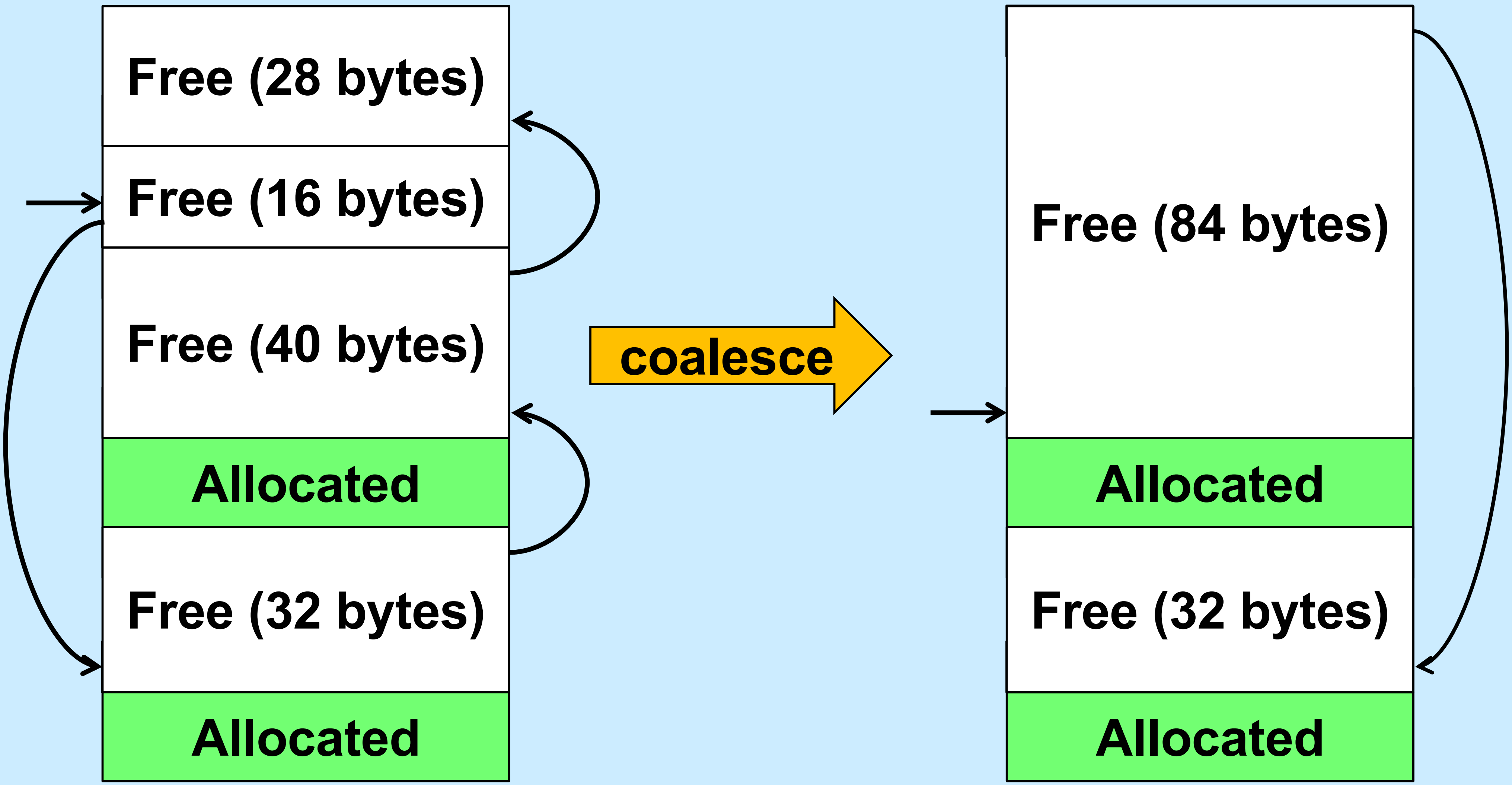
Splitting



Another Problem



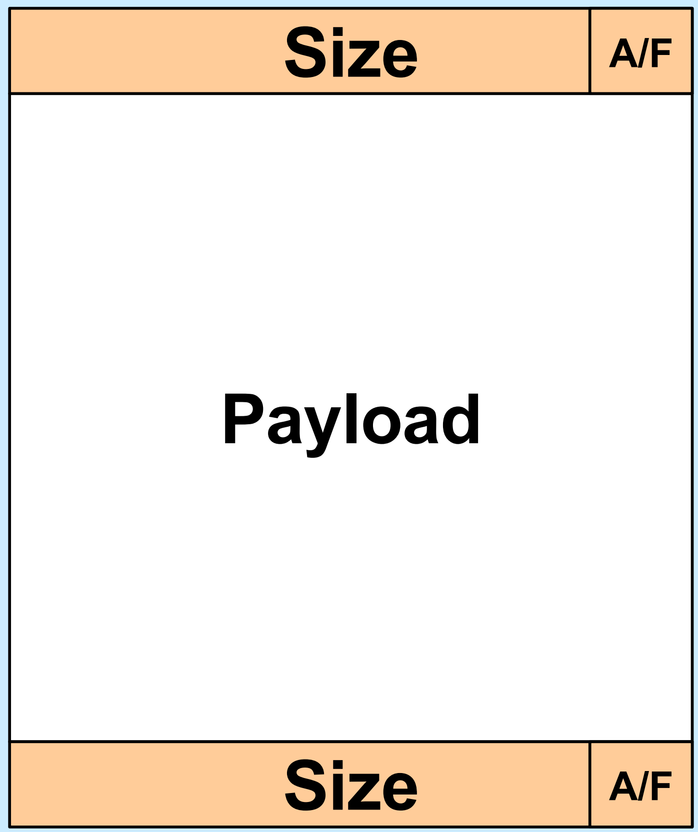
Coalescing


Free (28 bytes)
Free (16 bytes)
Free (40 bytes)
Allocated
Free (32 bytes)
Allocated
coalesce
Free (84 bytes)
Allocated
Free (32 bytes)
Allocated

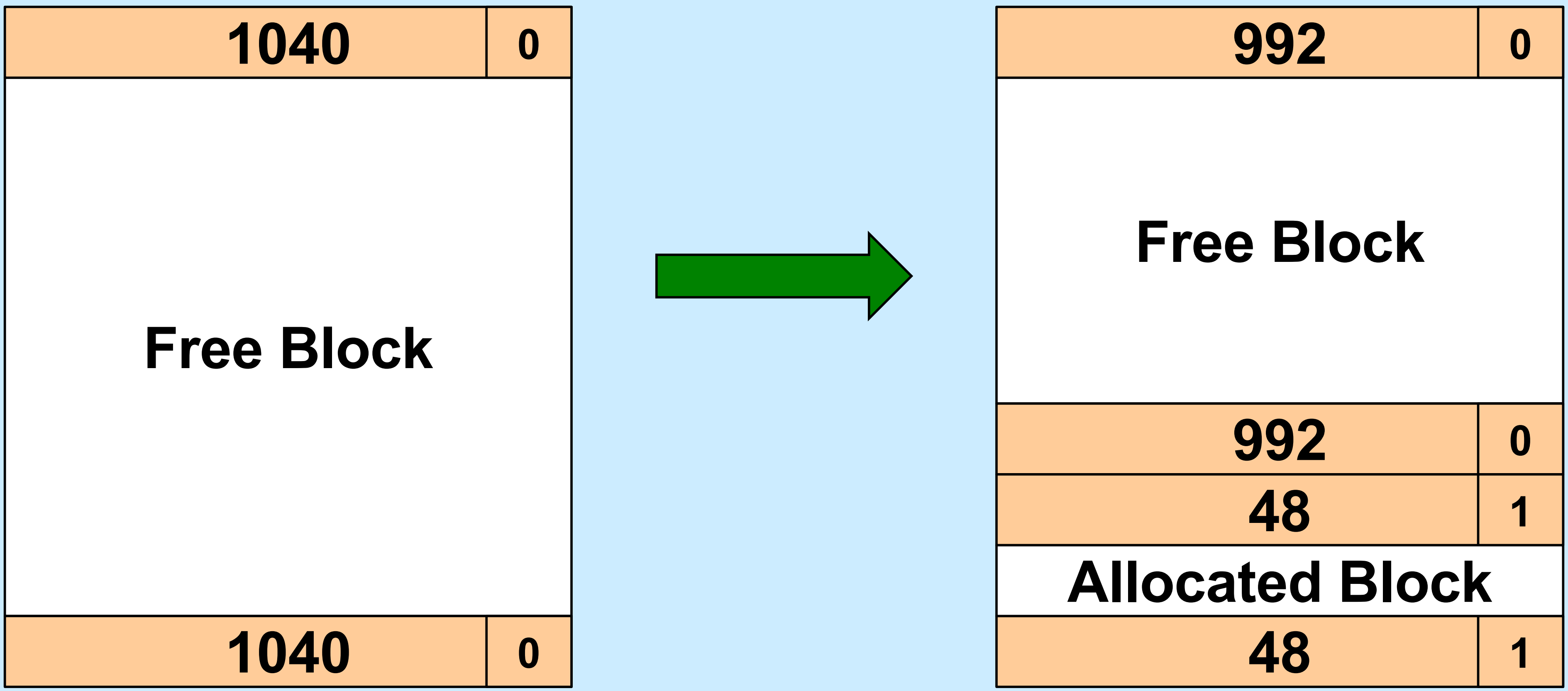
Data Structure Requirements

- All blocks
  - we need to know how big they are
    - » when free is called, it must be known how much to free
    - » when looking at a free block in malloc, we need to know its size
  - we need to know which they are: free or allocated
    - » needed for coalescing
- Free blocks
  - they need to be linked into the free list

Solution: Boundary Tags



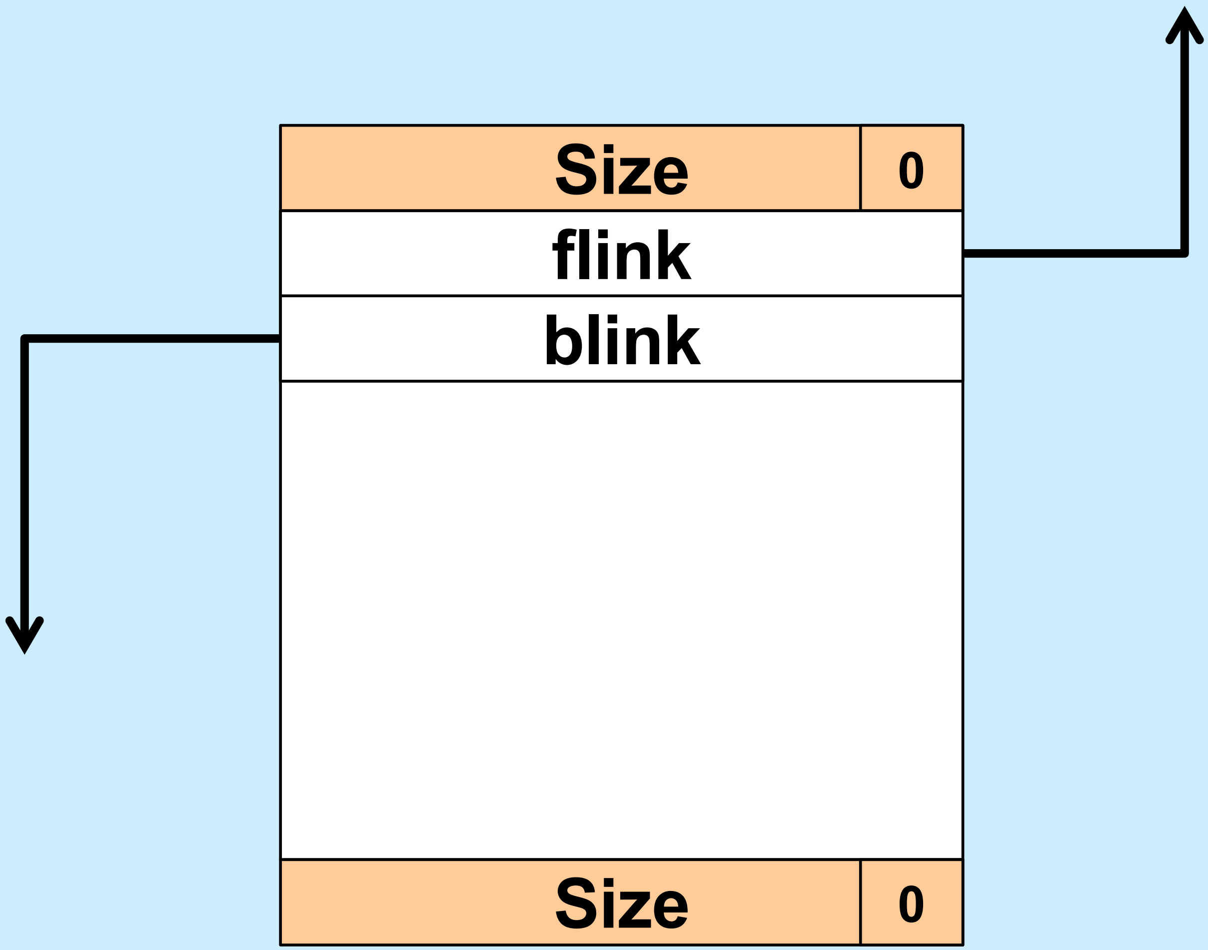
Splitting a Block



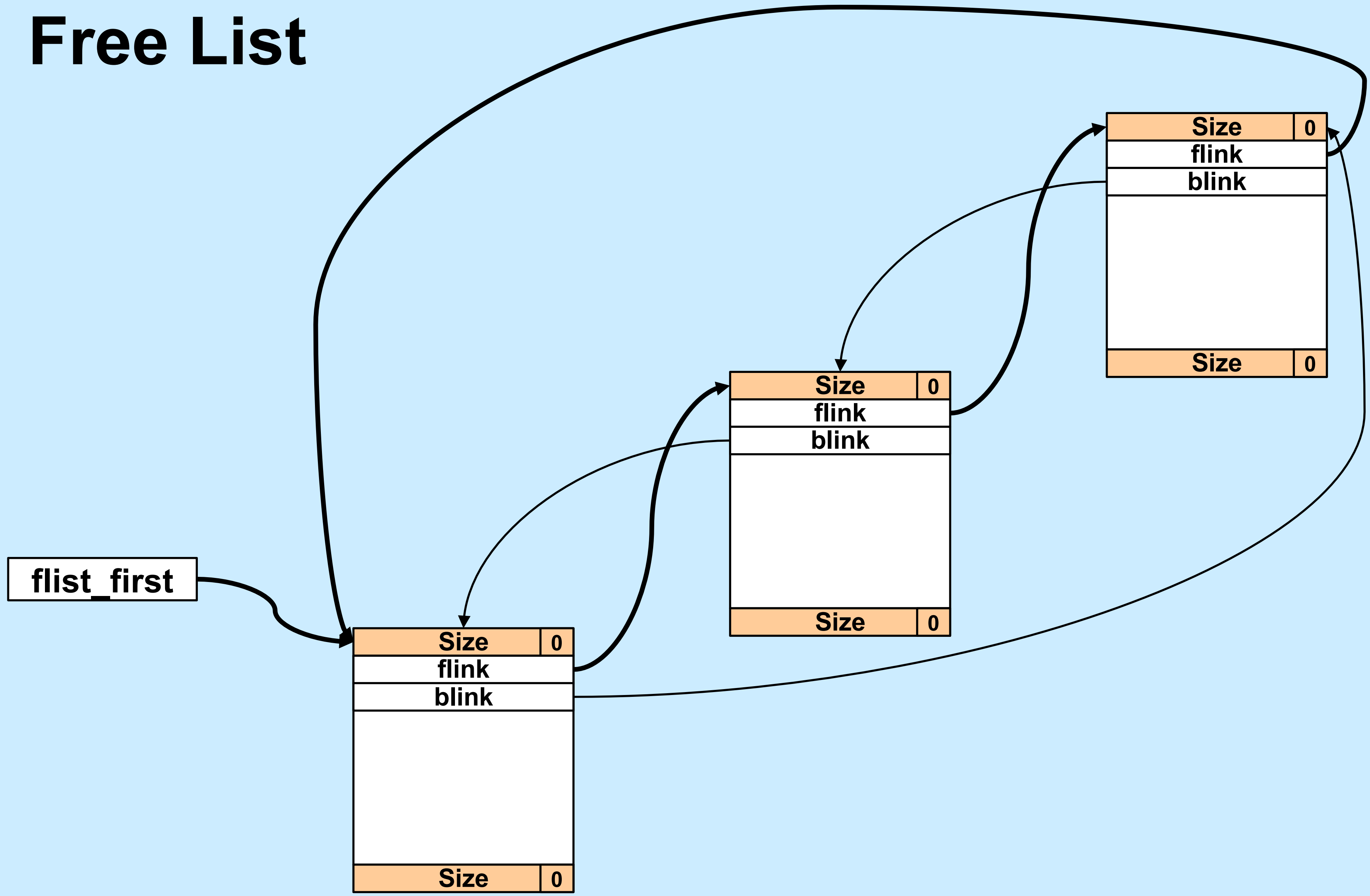
Representing the Free List

- We need a pointer to the first element
  – *flist_first*
- We need to traverse the list from beginning to end
  – required by malloc
- We need to merge adjacent blocks
  – this may require removing a block from the free list, then reinserting it (as part of a coalesced block)
- Links may be put in the free block's payload area
  – not needed for allocated blocks!

Free Block Representation



Free List



Size
0
flink
blink
Size
0
Size
0
flink
blink
Size
0
flist_first
Size
0
flink
blink
Size
0

Quiz 2

Why is the free list doubly linked?

a) we don’t really need it to be doubly linked for malloc and free, but it may be necessary for some future operations

b) to facilitate sorting the free list

c) so we can traverse it in both directions

d) so that, given a pointer to an arbitrary free block, we can easily remove the block from the list

Quiz 3

Why is the free list circular?

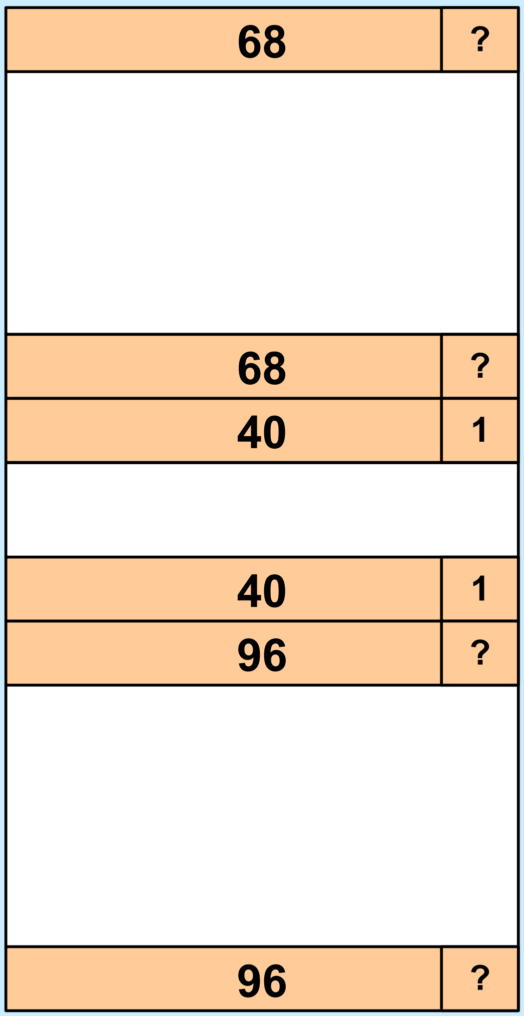
a) to facilitate implementing the next-fit search strategy

b) so that we don't have to special-case the the handling of the first and last list elements

c) both of the above

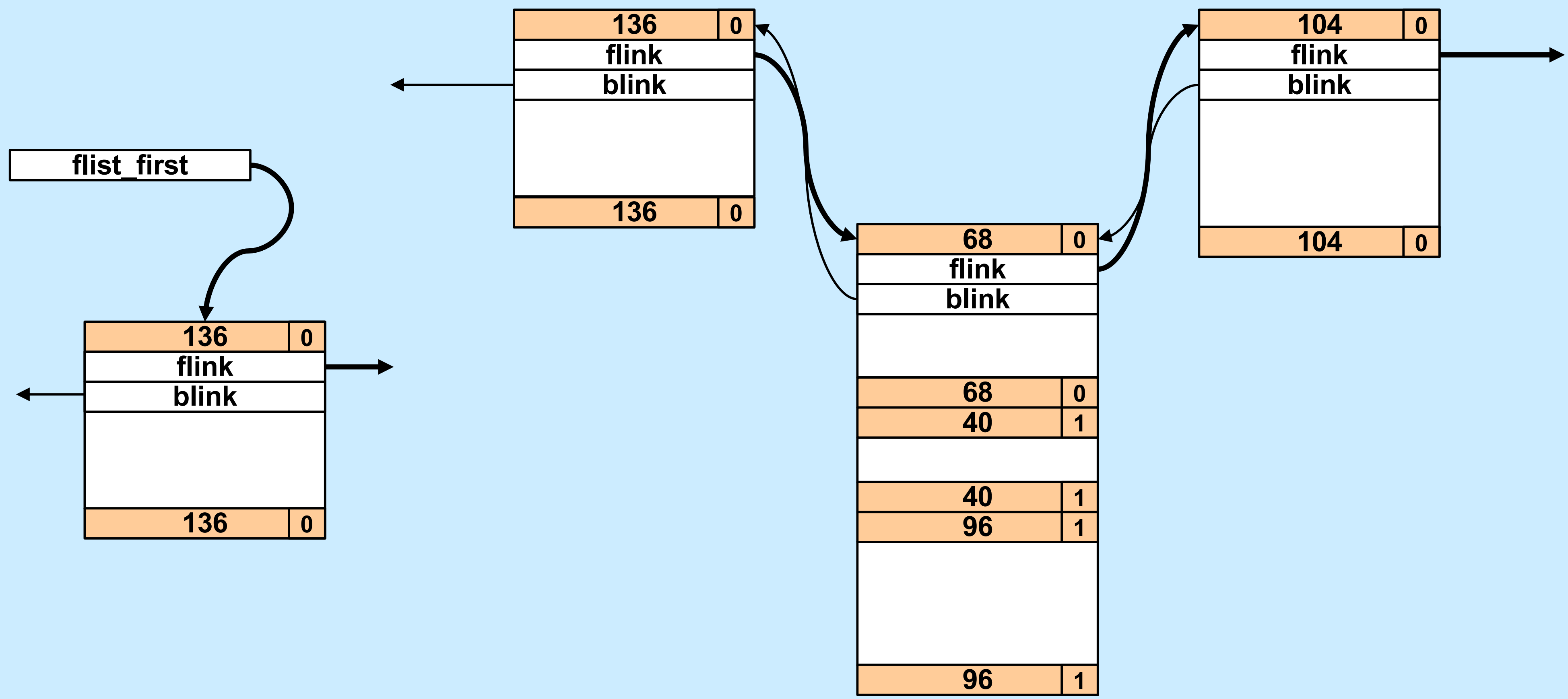
d) none of the above

Coalescing Revisited

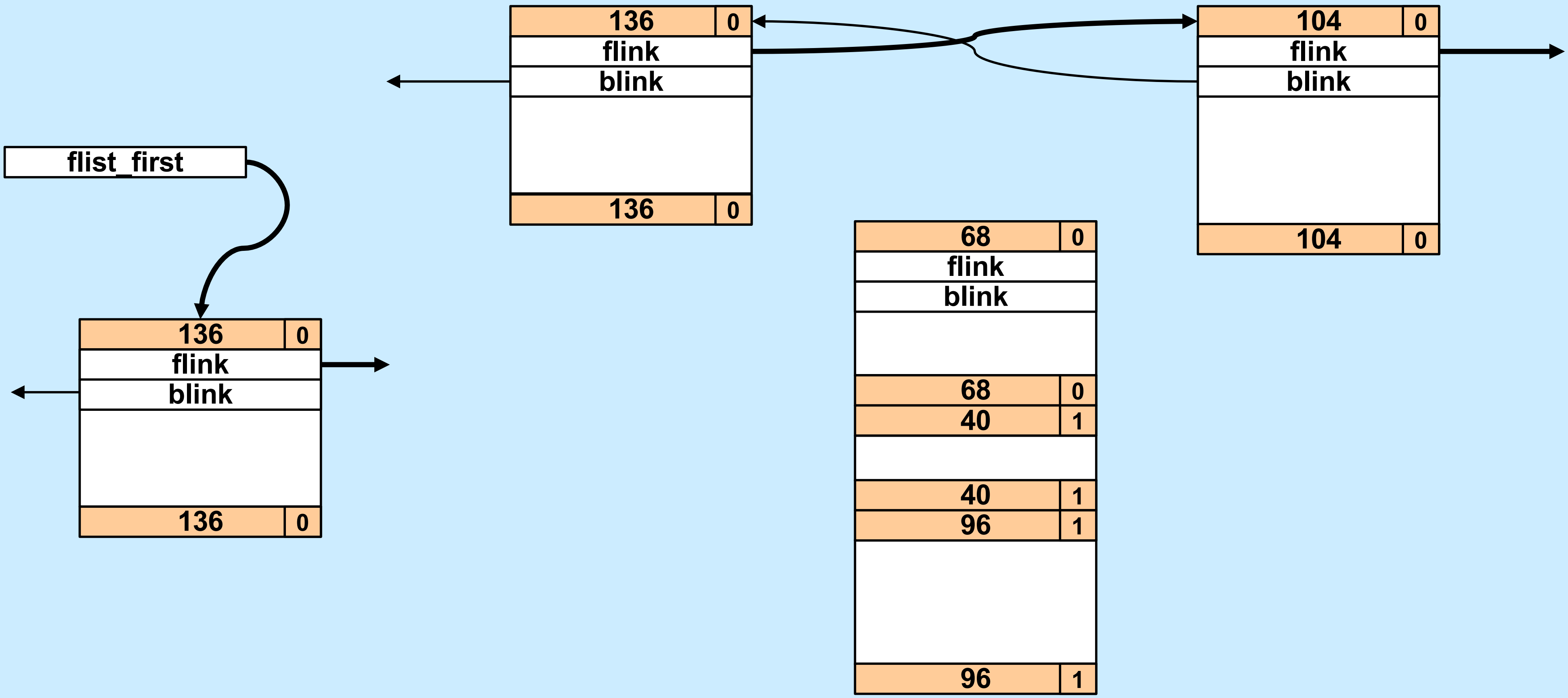


- **We are freeing a block**
  - **is the previous block free?**
  - **is the next block free?**
  - **are both free?**

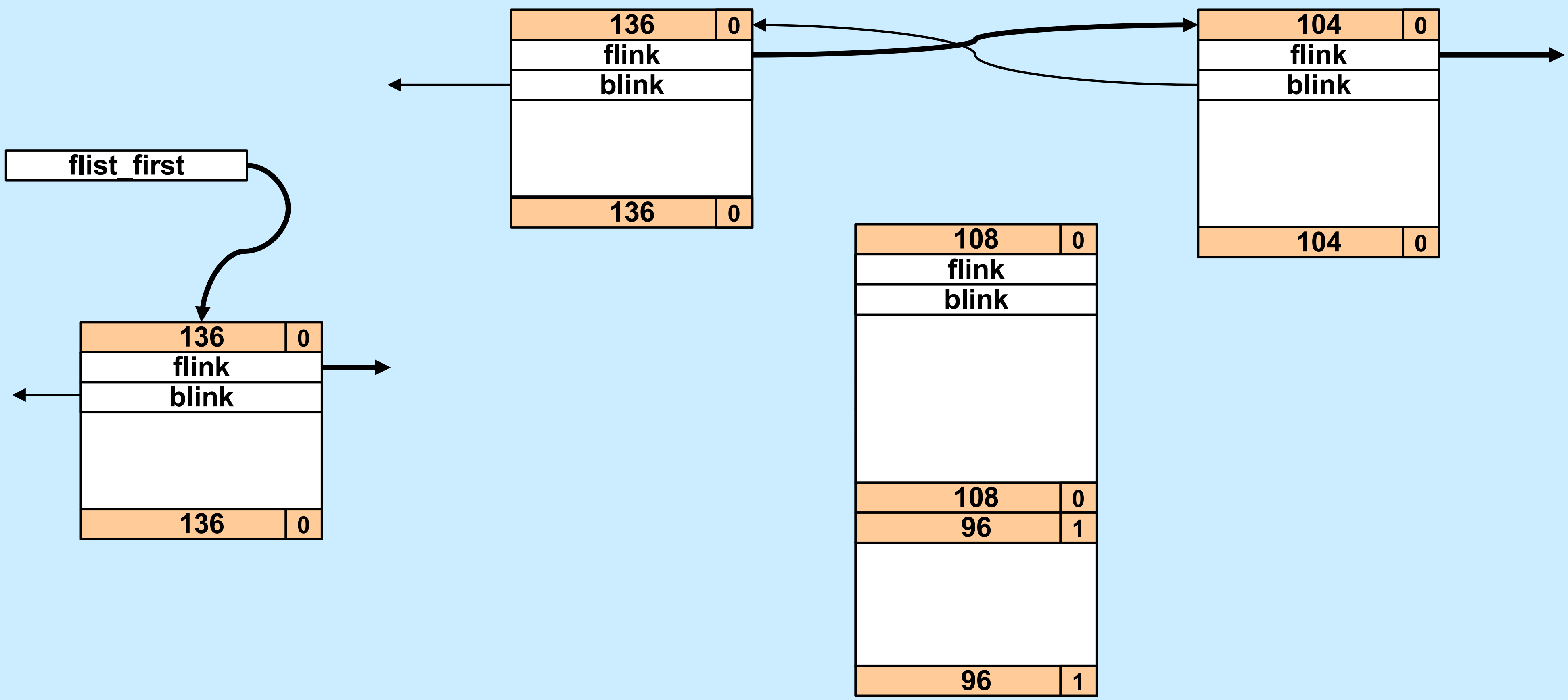
Coalescing: Previous Free (1)



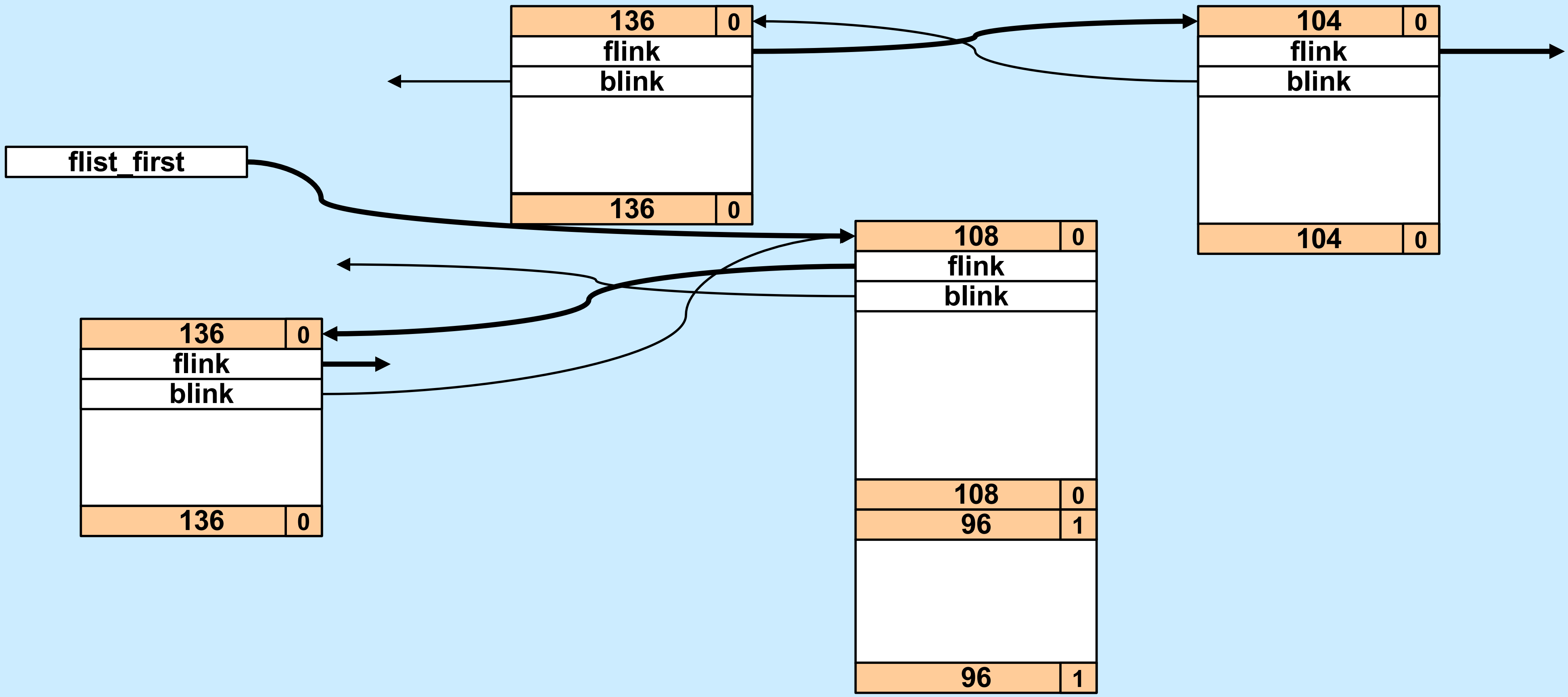
Coalescing: Previous Free (2)



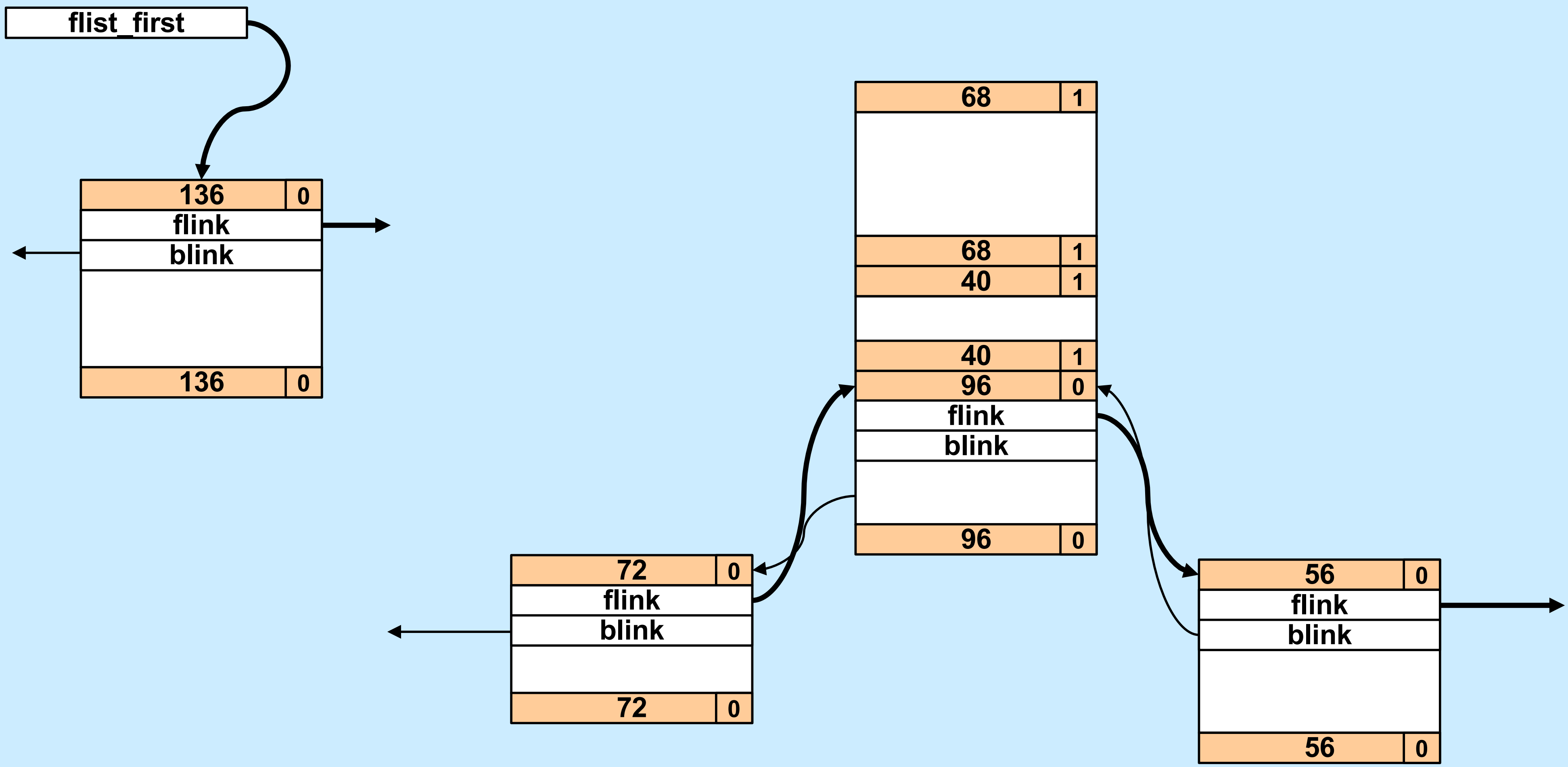
Coalescing: Previous Free (3)



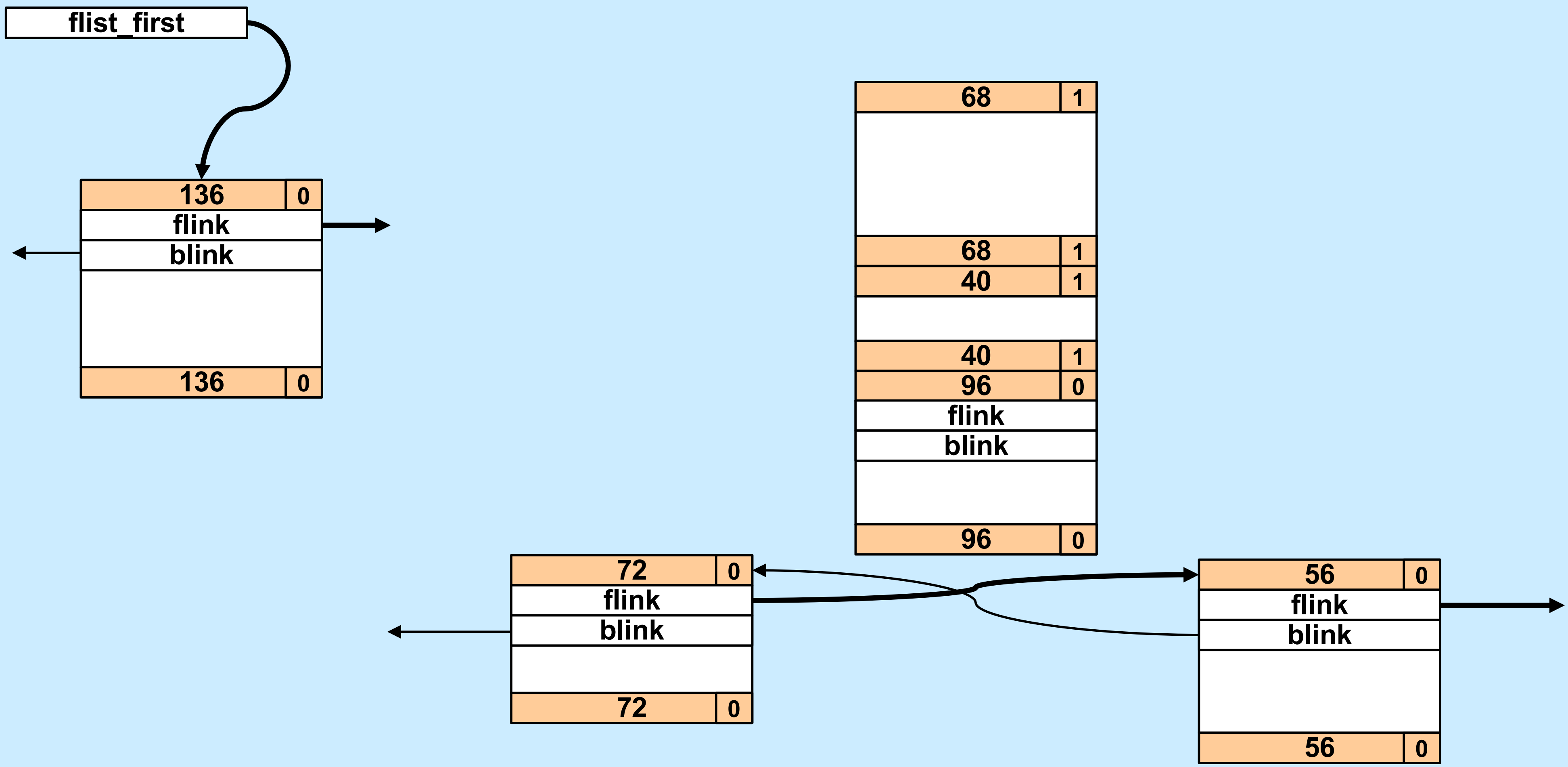
Coalescing: Previous Free (4)



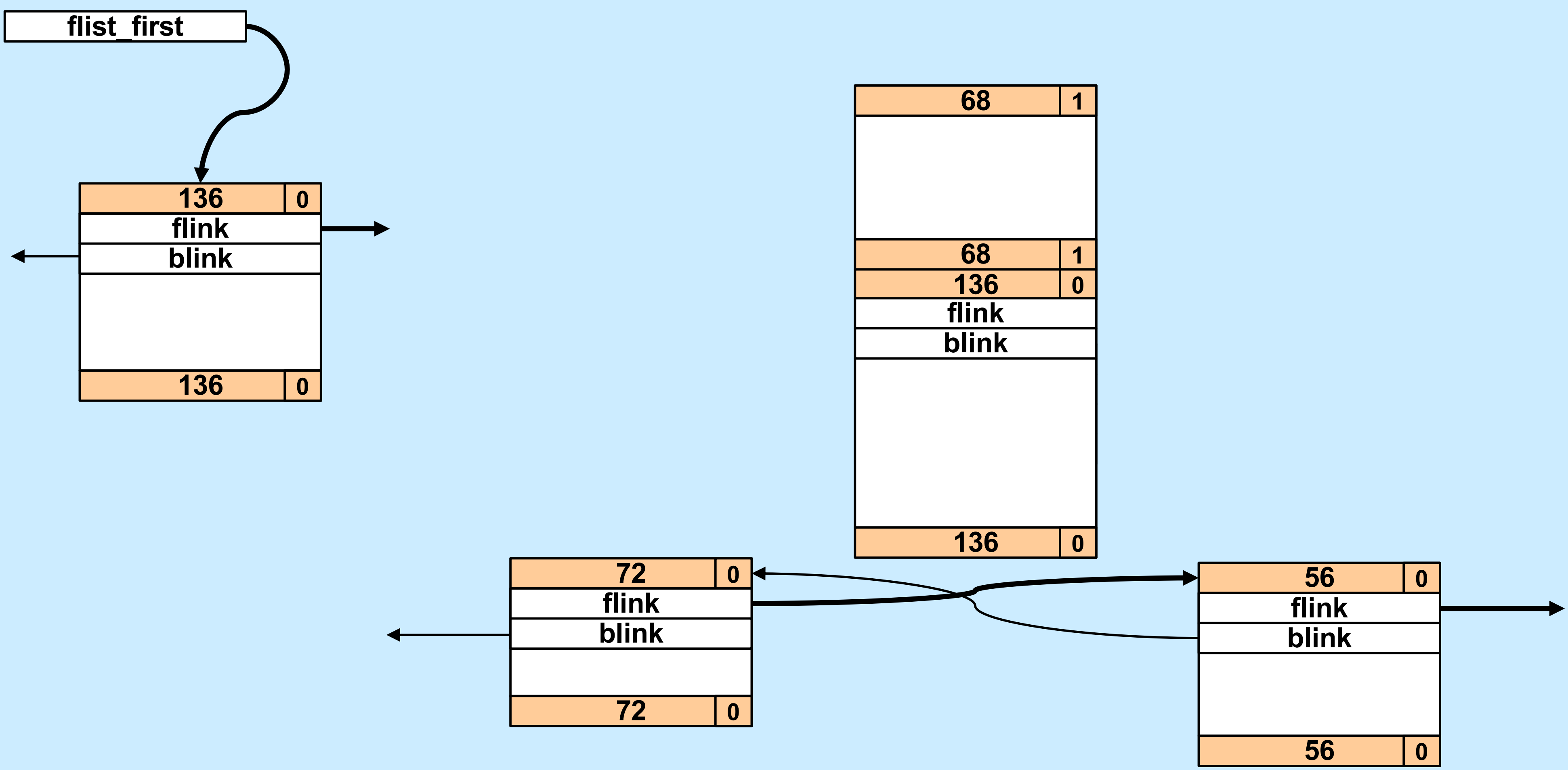
Coalescing: Next Free (1)



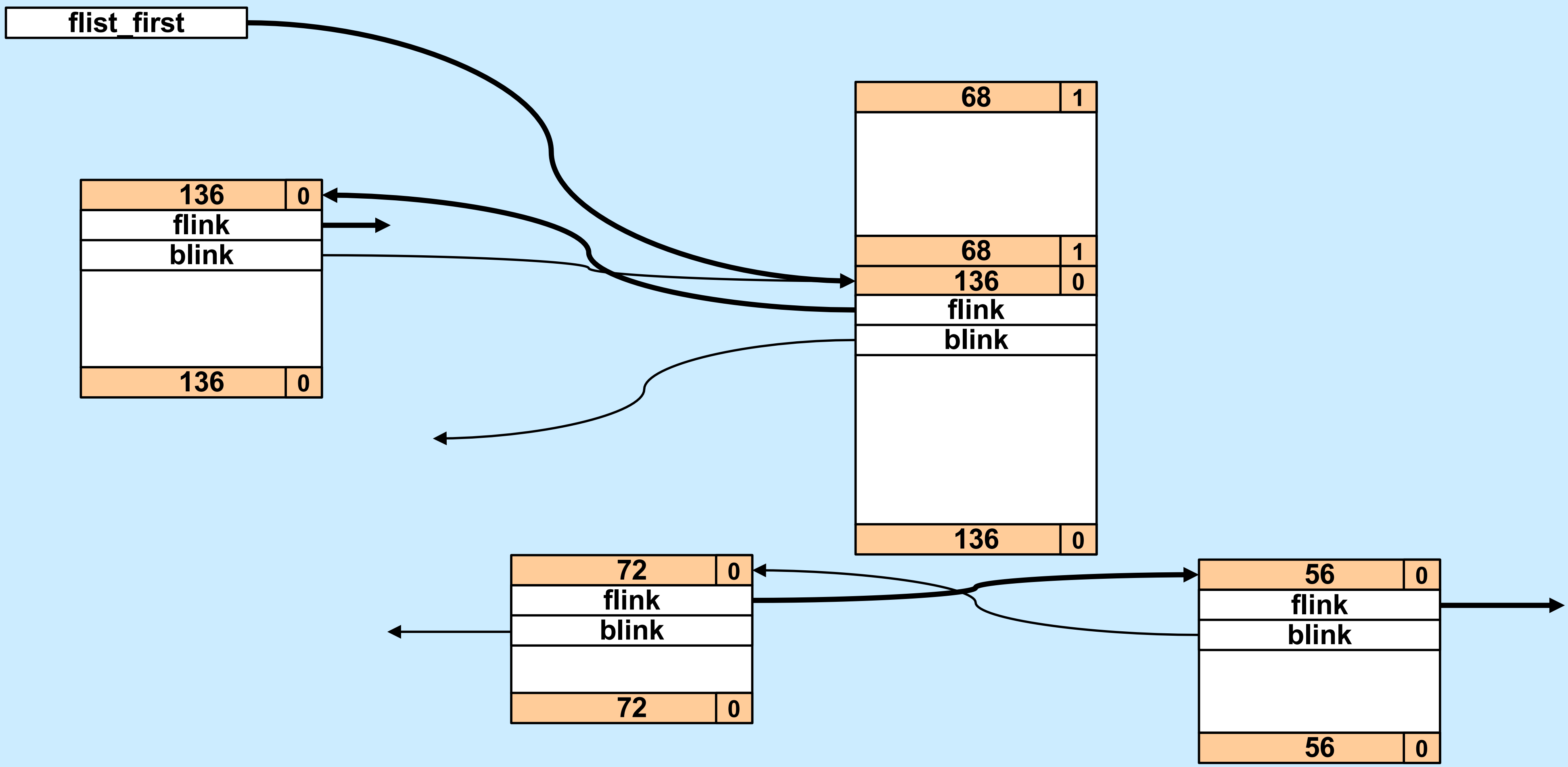
Coalescing: Next Free (2)



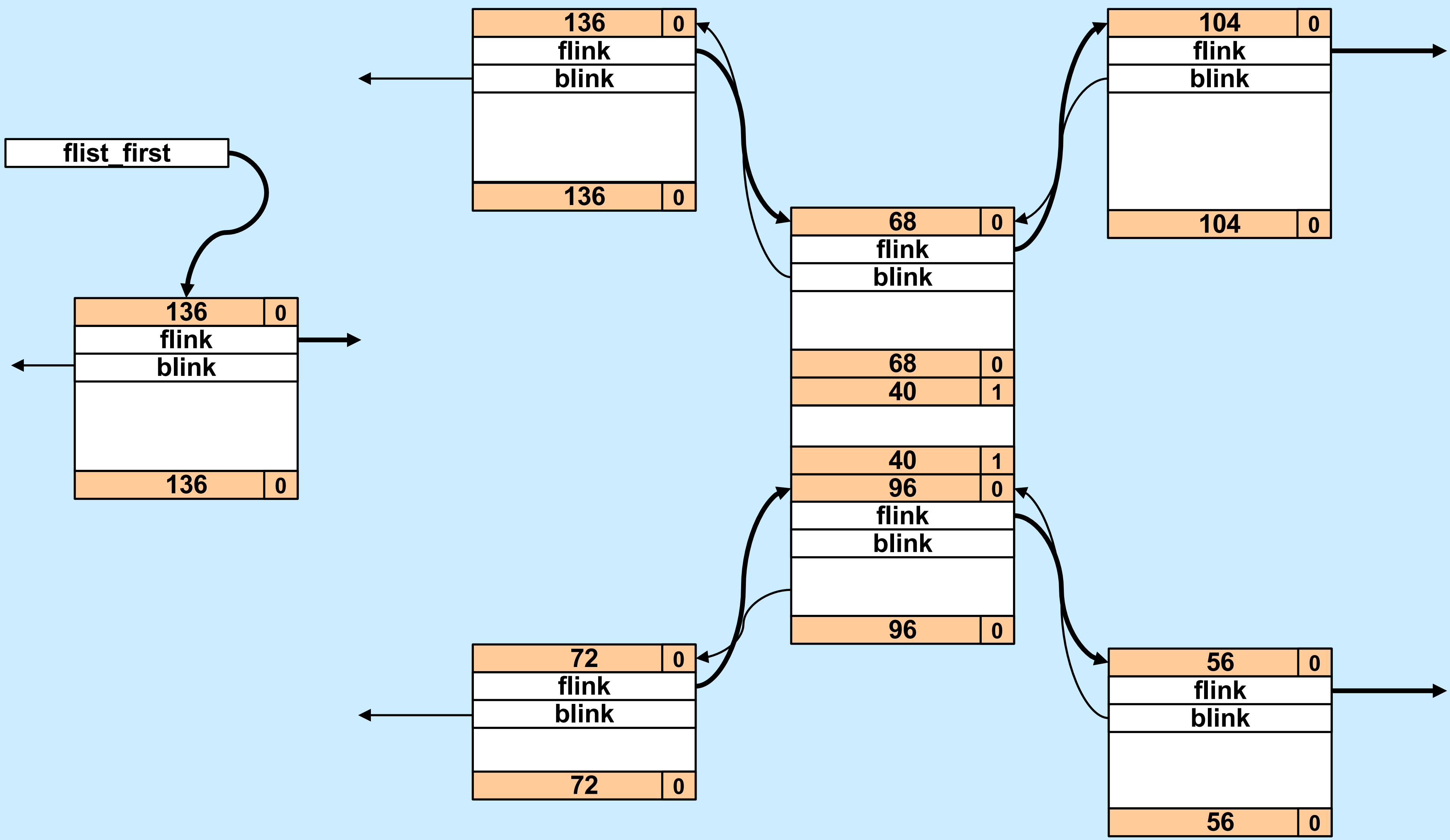
Coalescing: Next Free (3)



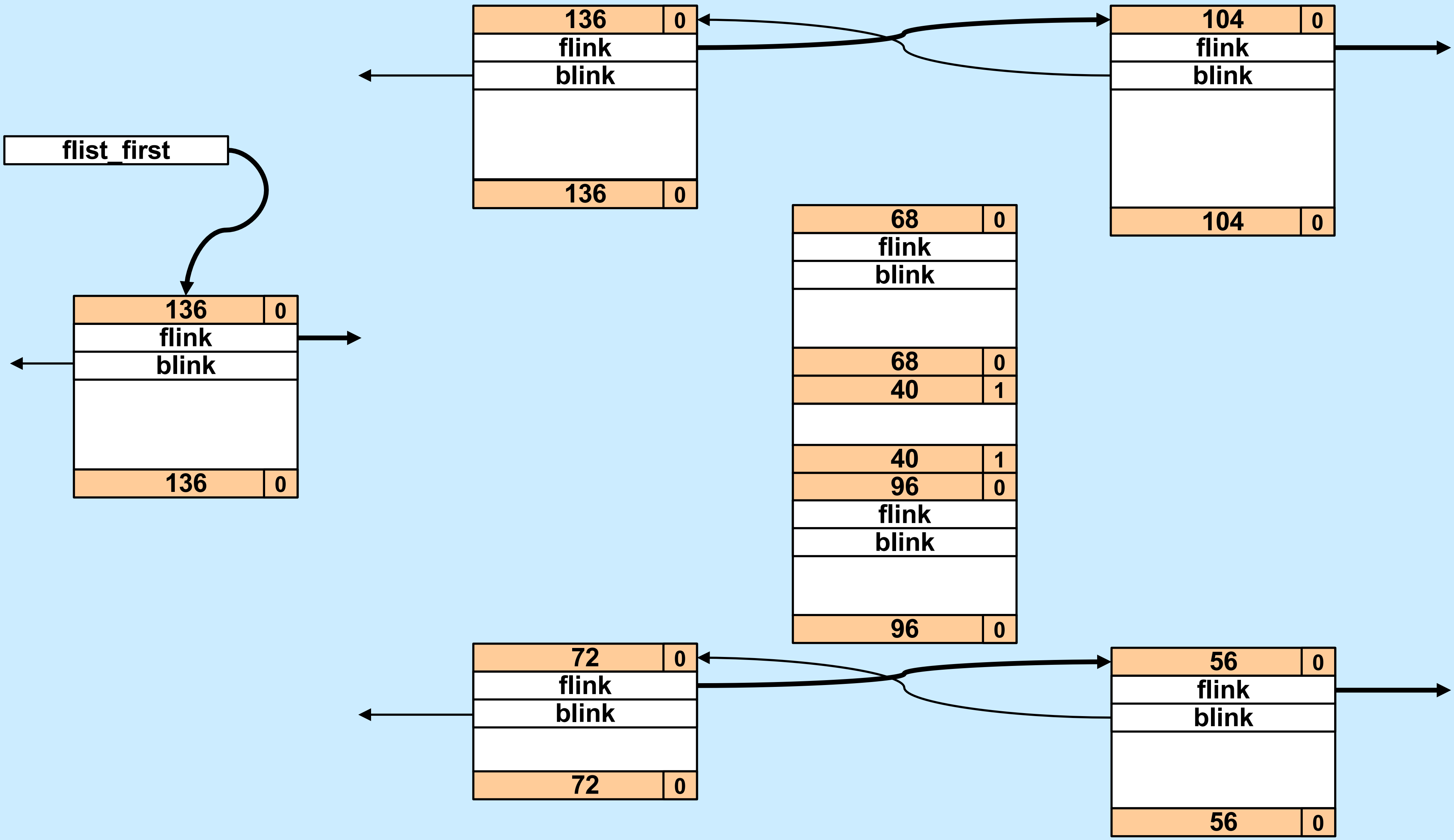
Coalescing: Next Free (4)



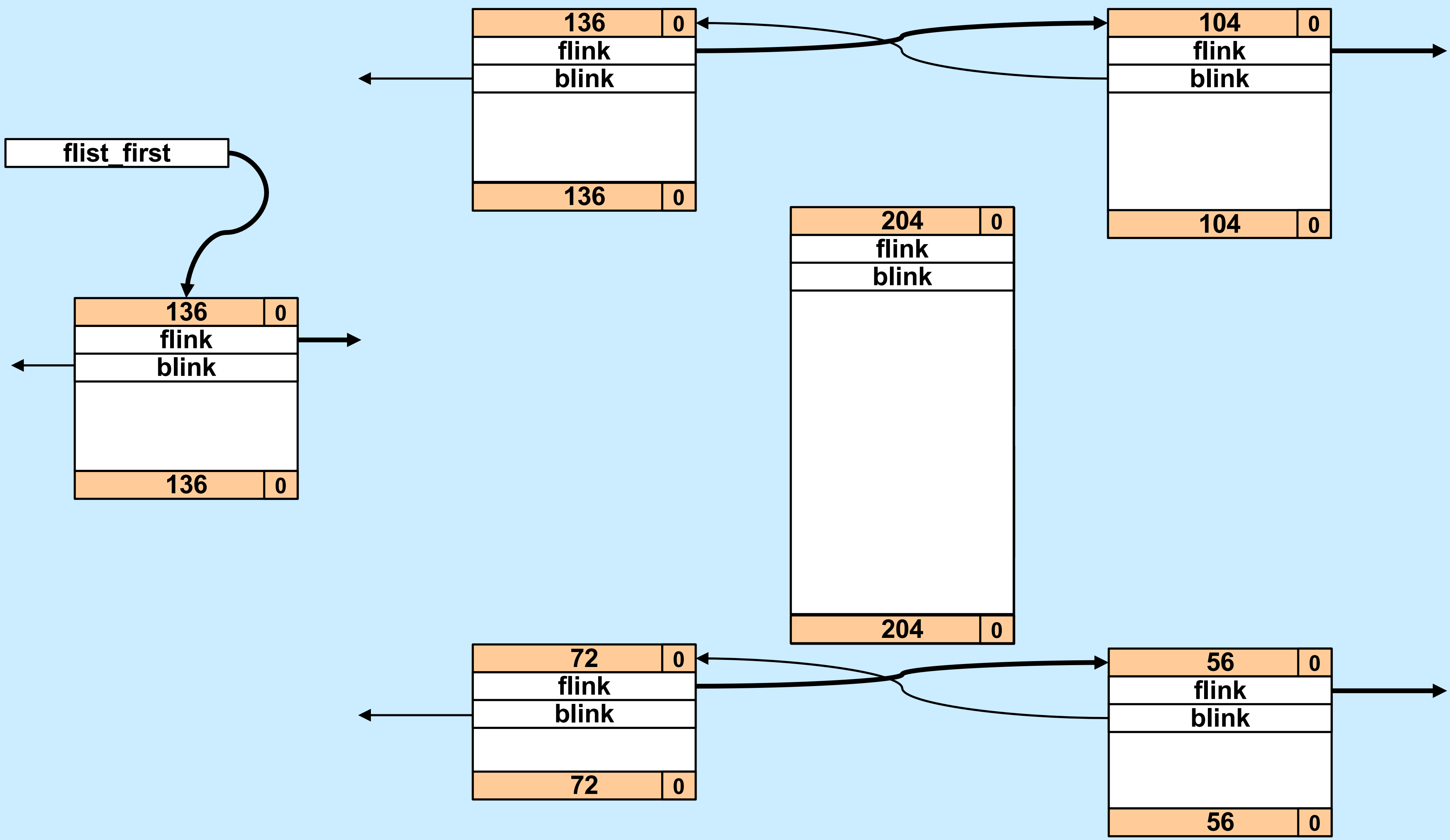
Coalescing: Both Free (1)



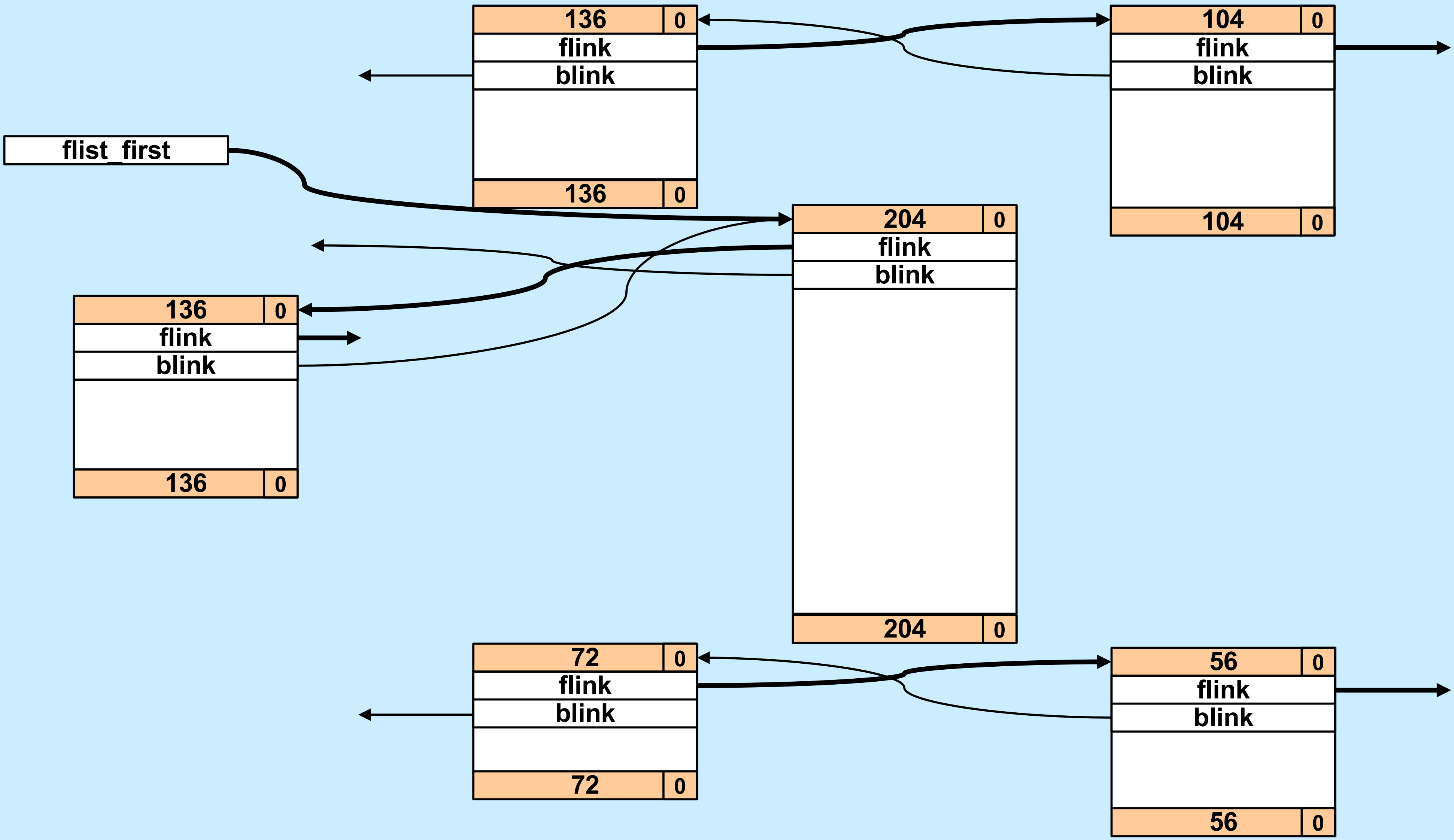
Coalescing: Both Free (2)



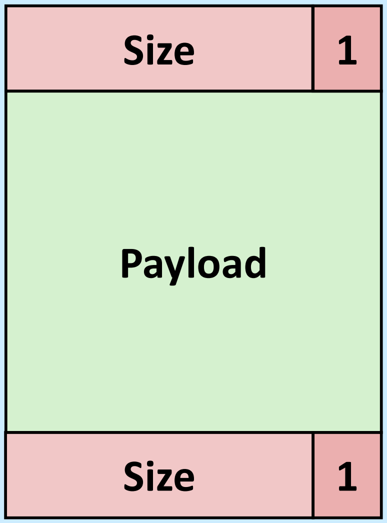
Coalescing: Both Free (3)



Coalescing: Both Free (4)



C vs. Storage Allocation




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

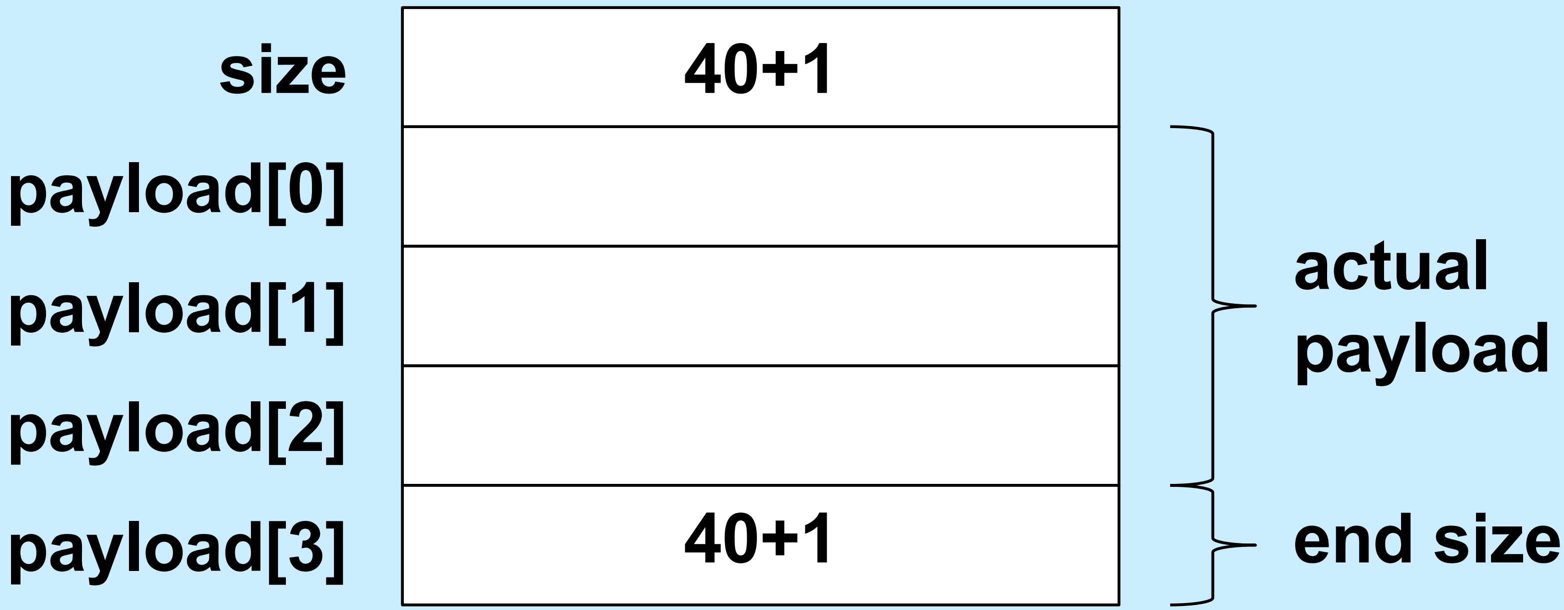
```
typedef struct free_block {
  long size;
  struct free_block *flink;
  struct free_block *blink;
  long filler[size/8 - 4];
  long end_size;
} free_block_t;
```

Overcoming C

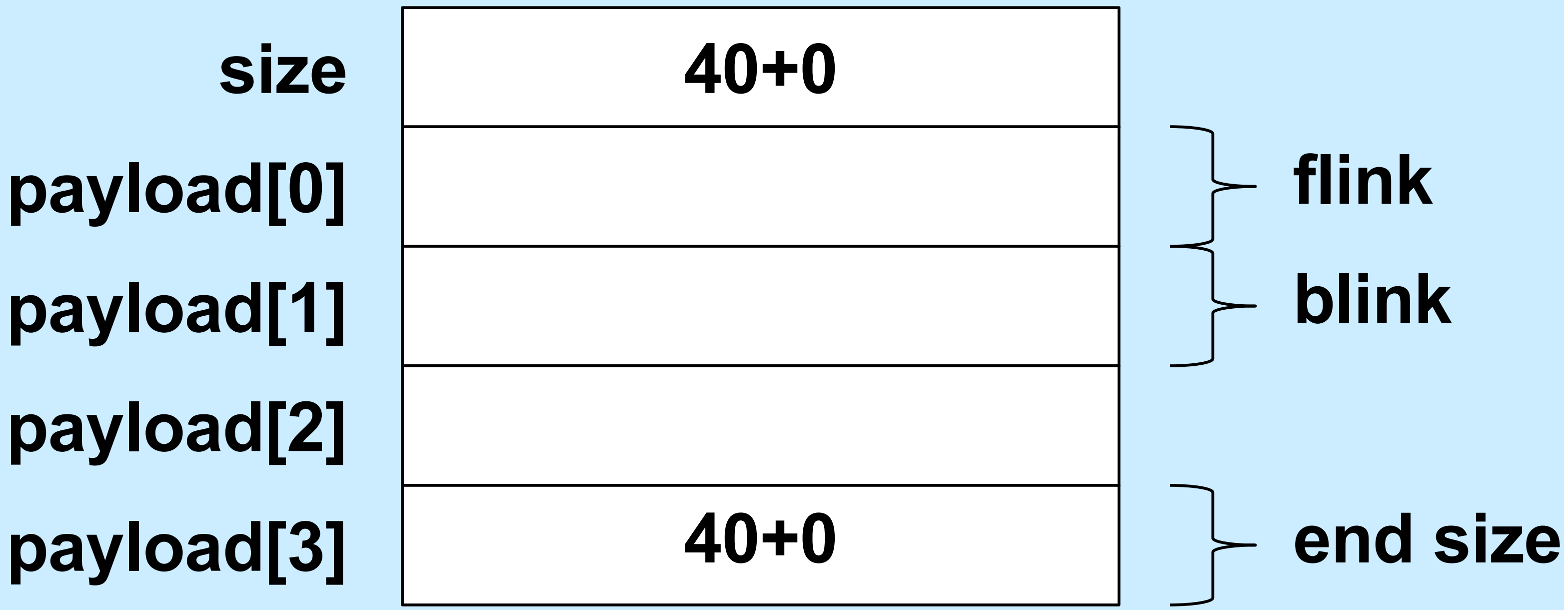
- **Think objects**
  - a block is an object
    - » opaque to the outside world
  - define accessor functions to get and set its contents

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

Allocated Block

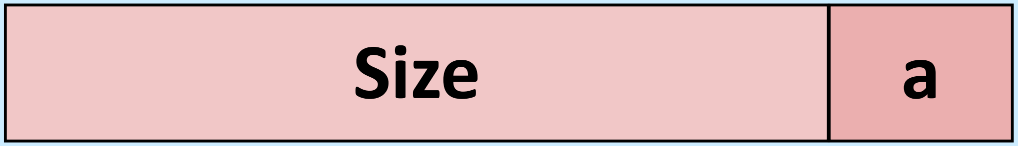


Free Block



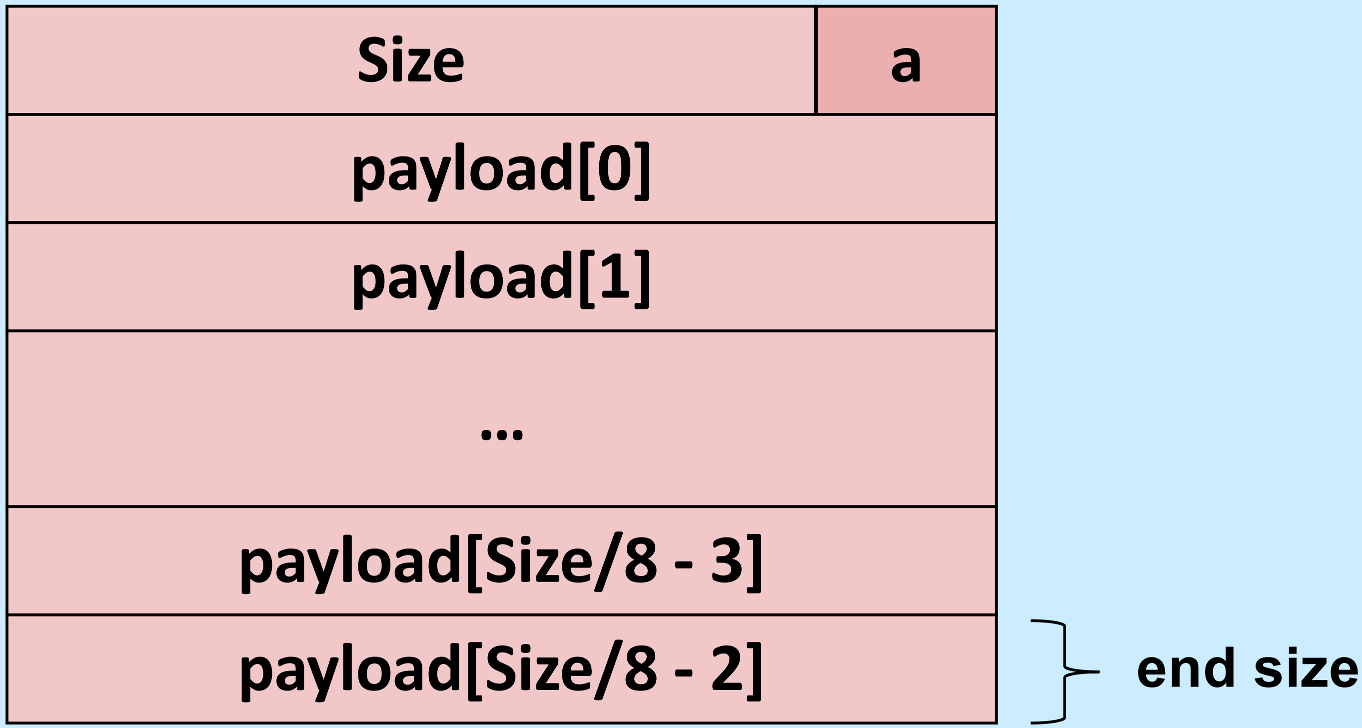
- In general, end size is at *payload[size/8 – 2]*

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

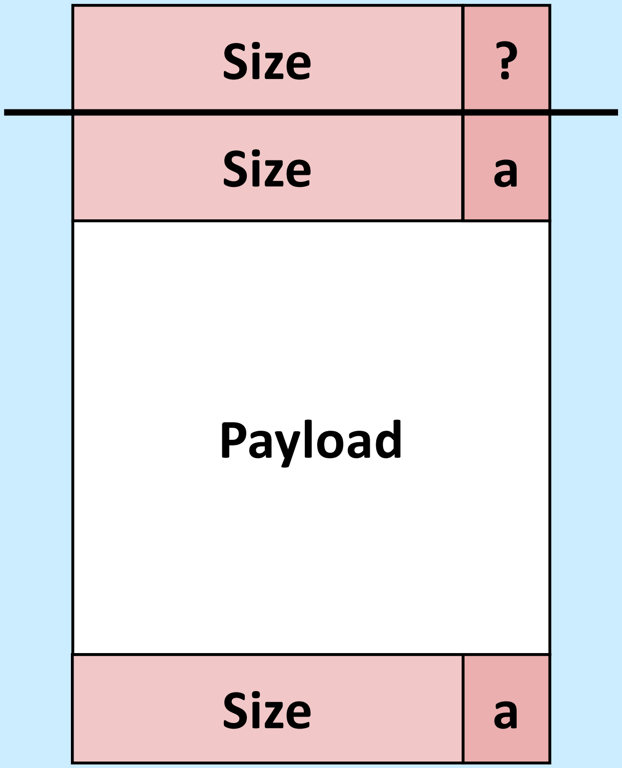



```
size_t *block_end_tag(block_t *b) {
  return &b->payload[b->size/8 - 2];
}
```


Setting the Size

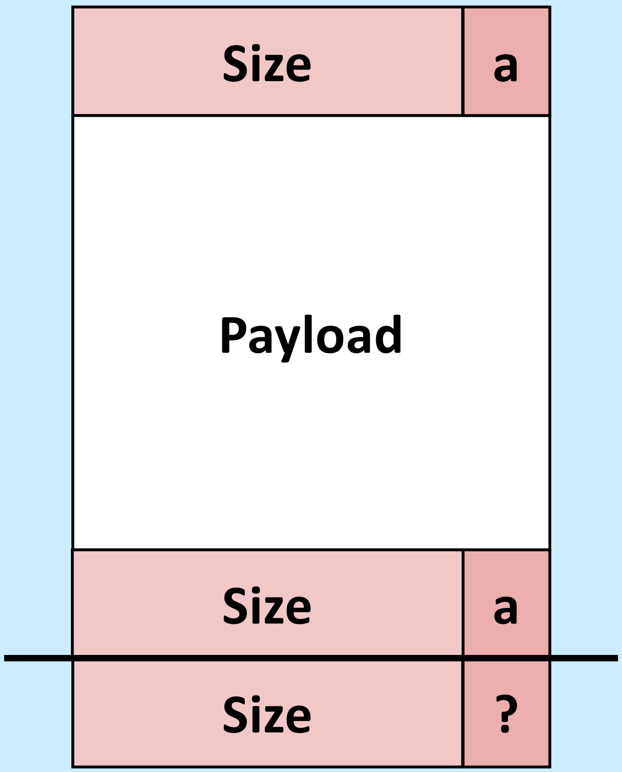
```
void block_set_size(block_t *b, size_t size) {
  assert(!(size & 7));           // multiple of 8
  size |= block_allocated(b);   // preserve alloc bit
  b->size = size;
  *block_end_tag(b) = size;
}

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?



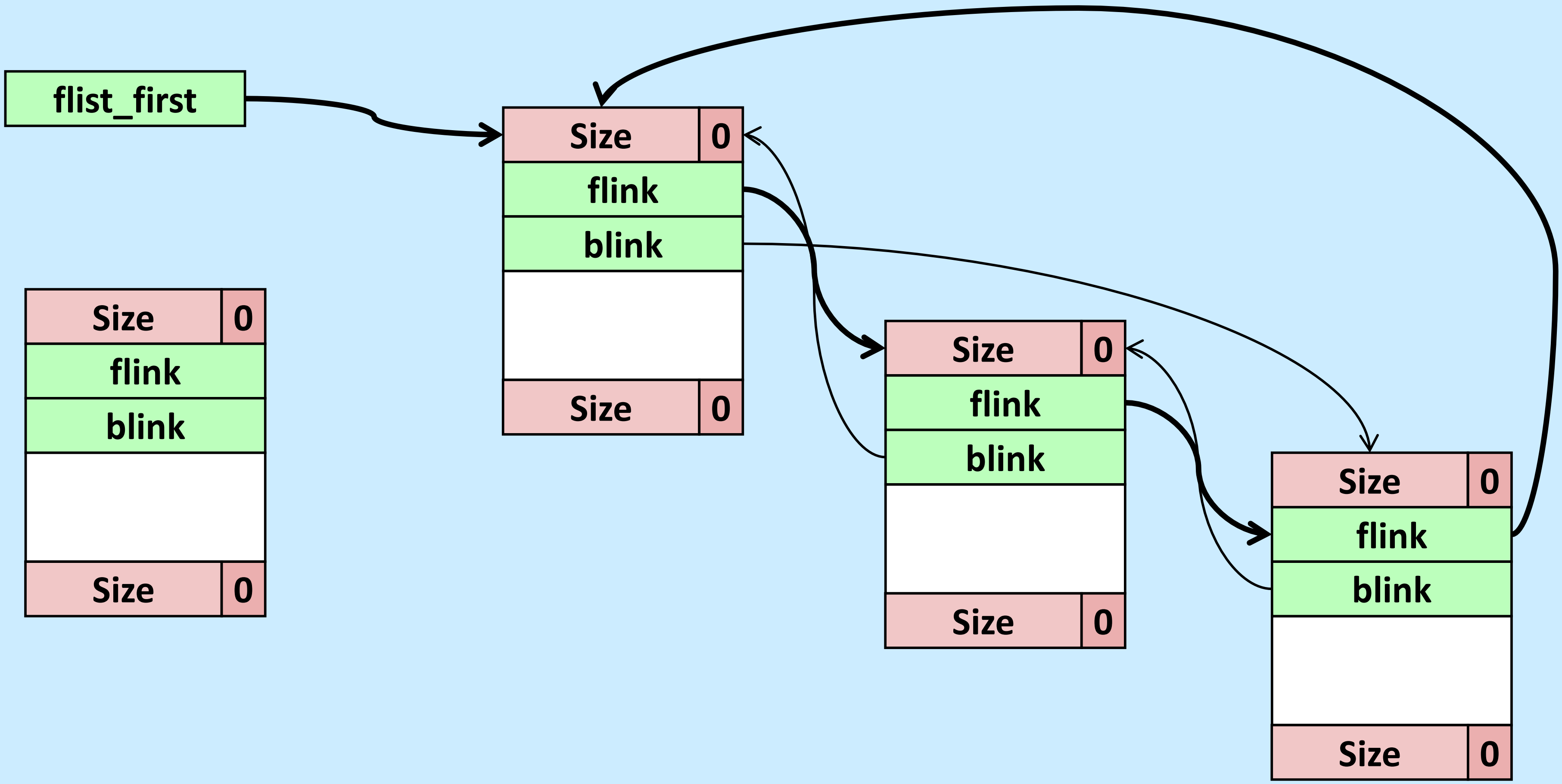

```
size_t block_prev_allocated(
    block_t *b) {
  return b->payload[-2] & 1;
}
```


Is Next Adjacent Block Free?

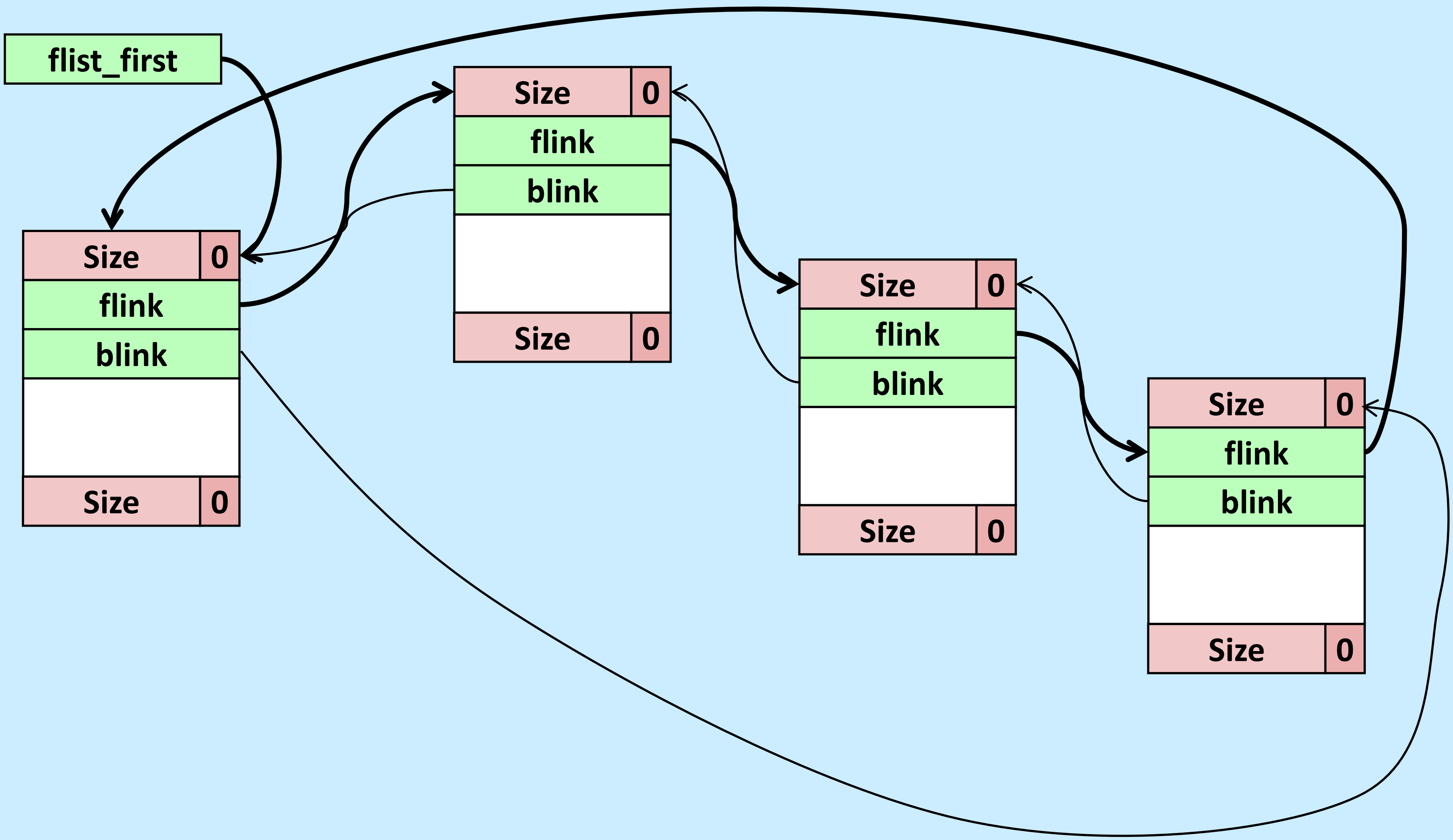



```
block_t *block_next(
    block_t *b) {
  return (block_t *)
    ((char *)b + block_size(b));
}

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

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

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

void block_set_blink(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_blink(flist_first);
    block_set_flink(fb, flist_first);
    block_set_blink(fb, last);
    block_set_flink(last, fb);
    block_set_blink(flist_first, fb);
  } else {
    block_set_flink(fb, fb);
    block_set_blink(fb, fb);
  }
  flist_first = fb;
}
```


Performance

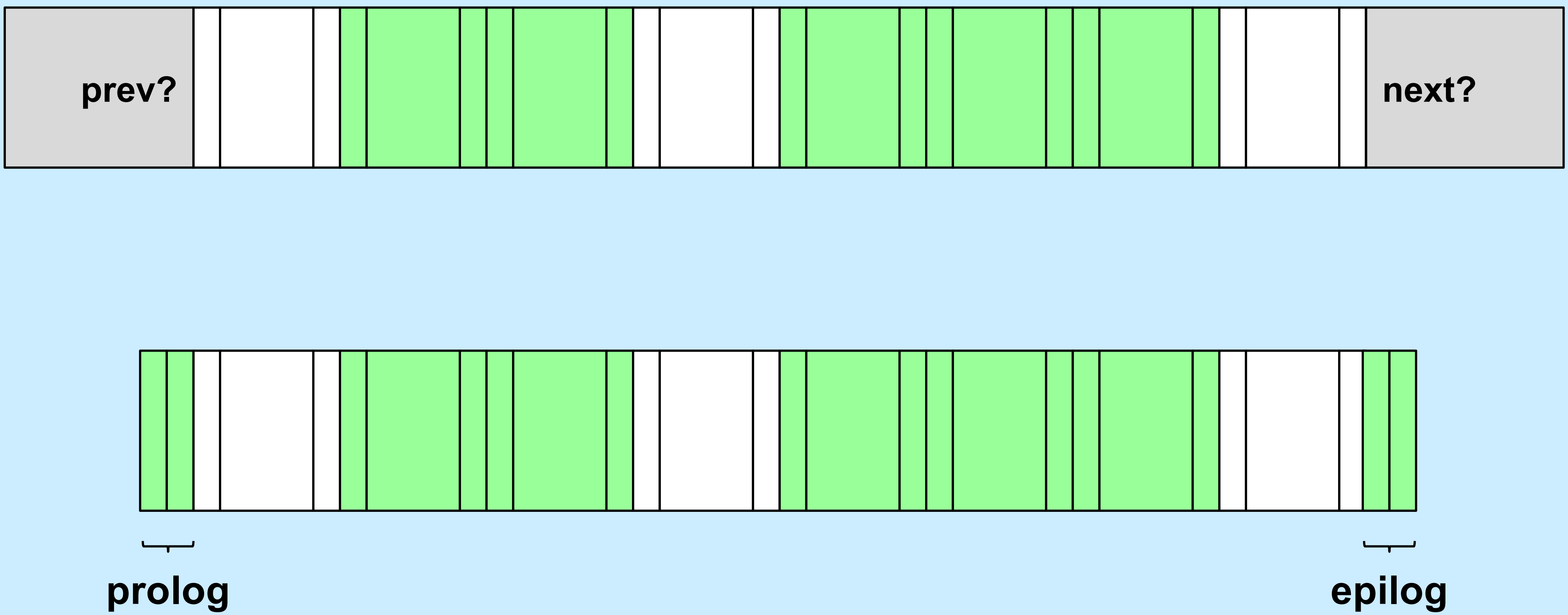
- Won't all the calls to the accessor functions slow things down a lot?
  - yes — not just a lot, but tons
- Why not use macros (#define) instead?
  - the textbook does this
  - it makes the code impossible to debug
    - » gdb shows only the name of the macro, not its body
- What to do????

Inline Functions

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

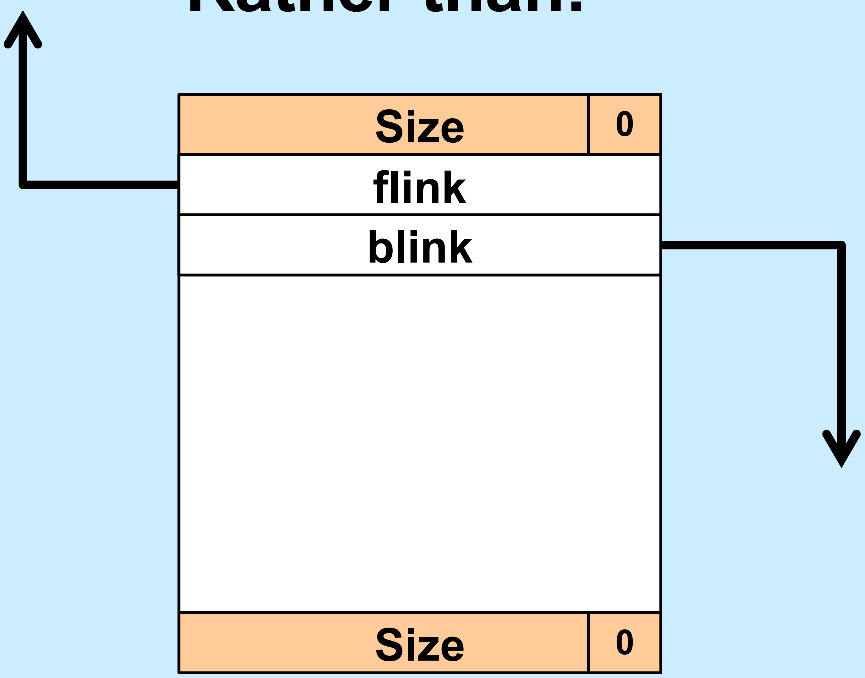
- when debugging (–O0), the code is implemented as a normal function
  - » easy to debug with gdb
- when optimized (–O1, –O2), calls to the function are replaced with the body of the function
  - » no function-call overhead

Prolog and Epilog



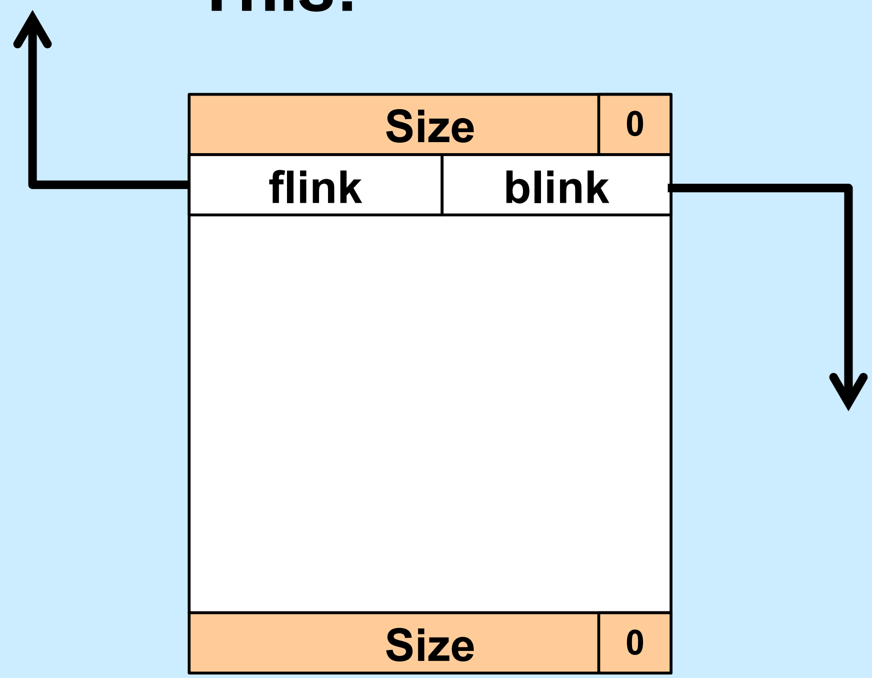
Compressed Pointers

- Rather than:



64-bit
pointers

- This:



32-bit
offsets

Freelist with Offsets