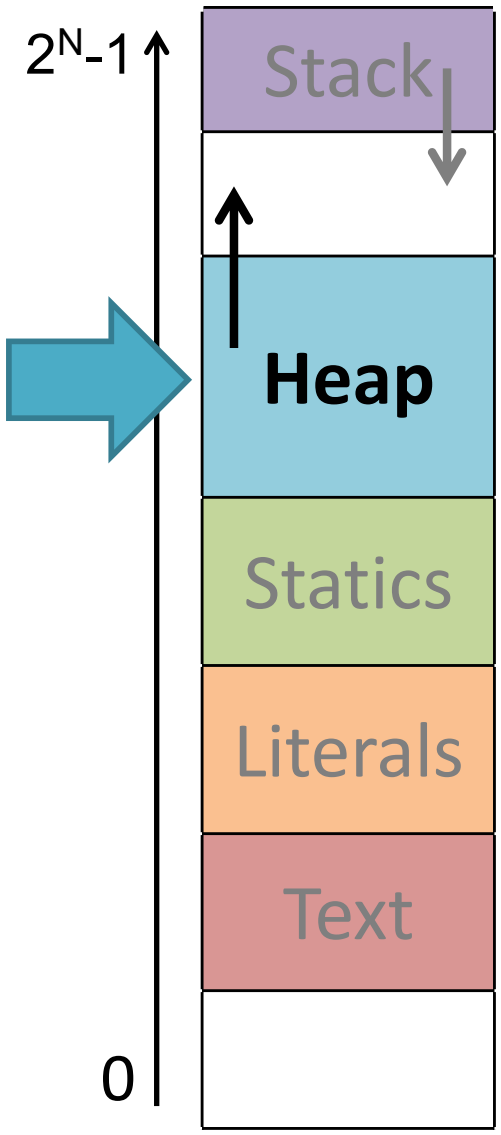
Dynamic Memory Allocation in the Heap

(malloc and free)

Now: Explicit allocators (a.k.a. manual memory management)

Later: implicit allocators (a.k.a. automatic memory management)

Heap Allocation

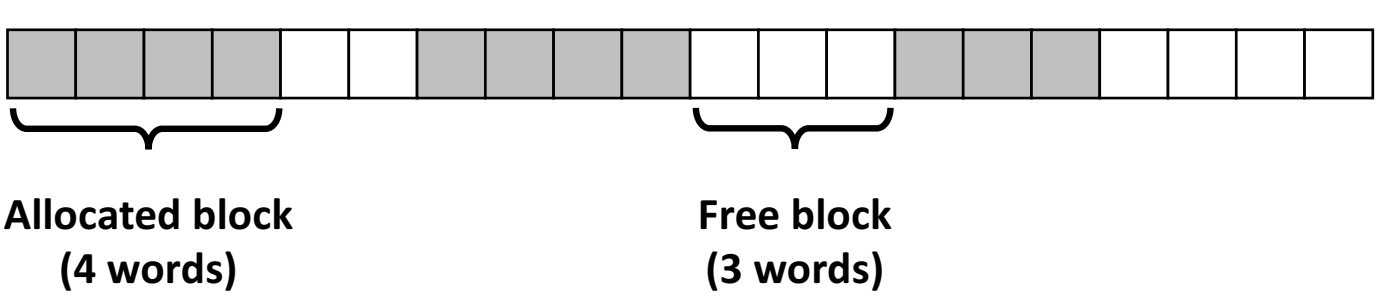


Perm	Contents	Managed by	Initialized
RW	Procedure context	Compiler	Run-time
RW	**Dynamic data structures**	Programmer, **malloc/free,** new/GC	**Run-time**
RW	Global variables/ static data structures	Compiler/ Assembler/Linker	Startup
R	String literals	Compiler/ Assembler/Linker	Startup
X	Instructions	Compiler/ Assembler/Linker	Startup

Allocator Basics

Pages too coarse-grained for allocating individual objects.

Instead: **flexible-sized, word-aligned blocks.**



pointer to newly allocated block of at least that size

number of contiguous bytes required

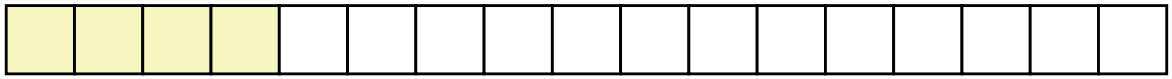
```
void* malloc(size_t size);
```

pointer to allocated block to free

```
void free(void* ptr);
```

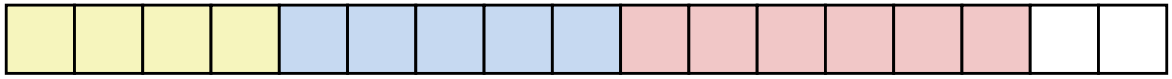
Example (64-bit words)

```
p1 = malloc(32);
```

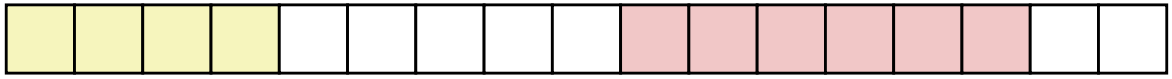


```
p2 = malloc(40);
```

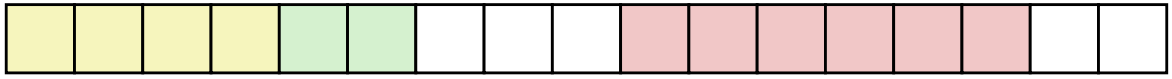
```
p3 = malloc(48);
```



```
free(p2);
```



```
p4 = malloc(16);
```



Allocator Goals: malloc/free

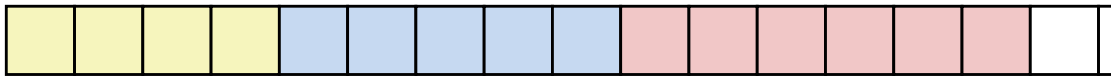
1. Programmer does not decide locations of distinct objects.

Programmer decides: what size, when needed, when no longer needed

2. Fast allocation.

mallocs/second or bytes malloc'd/second

3. High memory utilization.



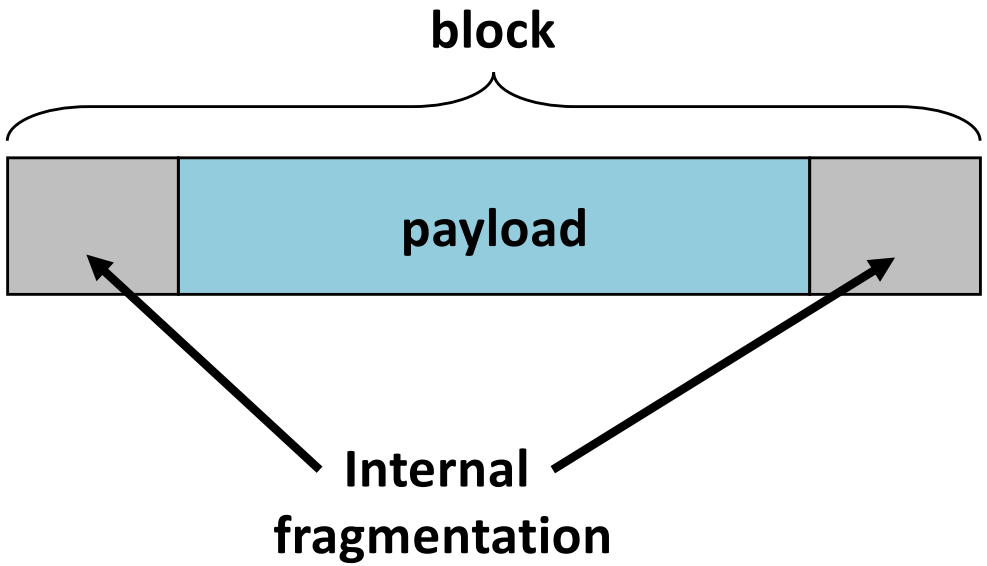
Most of heap contains necessary program data.

Little wasted space.

Enemy: **fragmentation** – unused memory that cannot be allocated.

Internal Fragmentation

payload smaller than block



Causes

- metadata
- alignment
- policy decisions

External Fragmentation (64-bit words)

Total free space large enough,

but no contiguous free block large enough

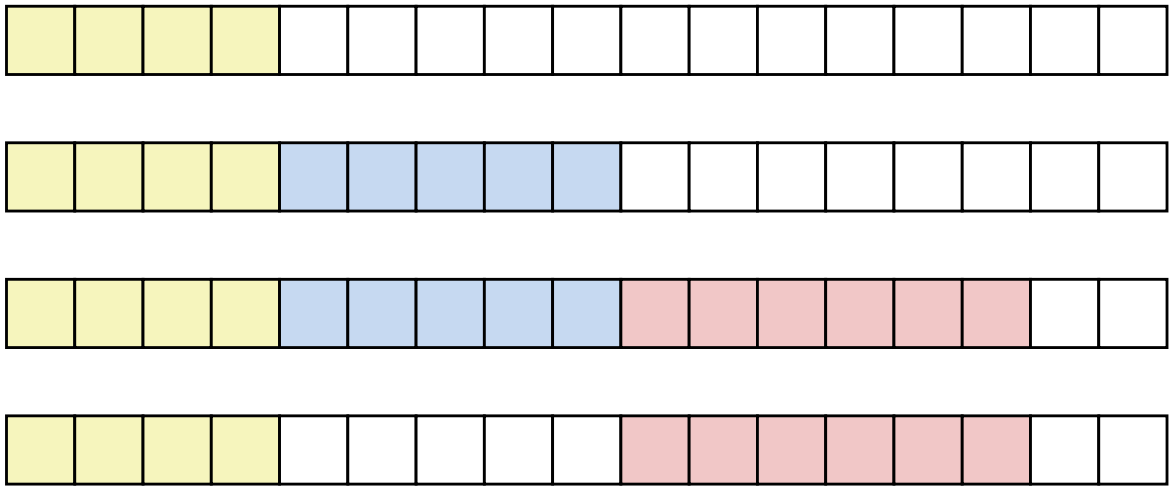
```
p1 = malloc(32);
```

```
p2 = malloc(40);
```

```
p3 = malloc(48);
```

```
free(p2);
```

```
p4 = malloc(48);
```



Depends on the pattern of future requests.

Implementation Issues

1. Determine **how much to free** given just a pointer.

2. **Keep track of free blocks.**

3. **Pick a block to allocate.**

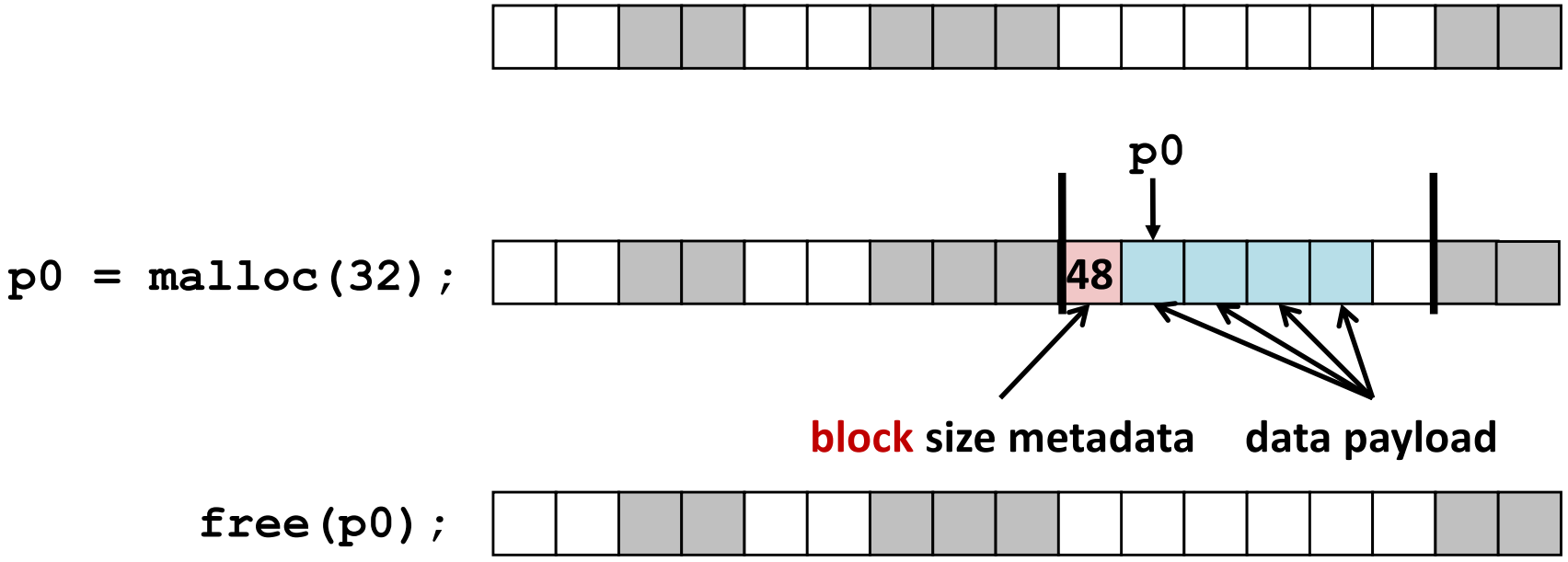
4. Choose what do with **extra space when allocating** a structure that is **smaller than the free block used**.

5. **Make a freed block available for future reuse.**

Knowing How Much to Free

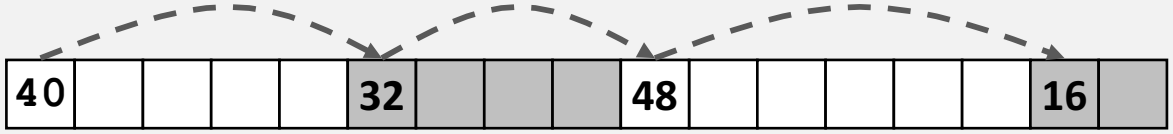
Keep length of block in ***header*** word preceding block

Takes extra space!

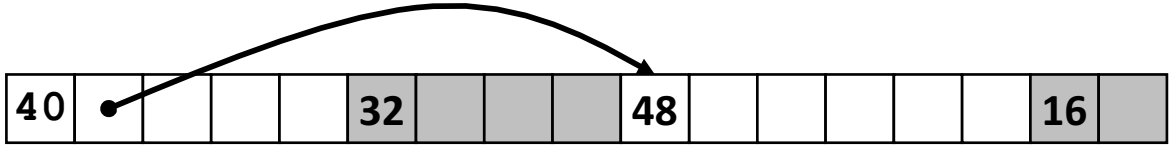


Keeping Track of Free Blocks

Method 1: ***Implicit list*** of all blocks using length



Method 2: ***Explicit list*** of free blocks using pointers



Method 3: *Seglist*

Different free lists for different size blocks

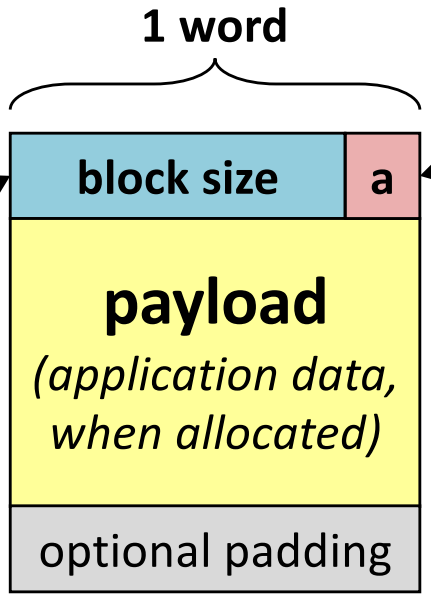
More methods that we will skip…

Implicit Free List: **Block Format**

Block metadata:

1. Block size
2. Allocation status

Store in one header word.

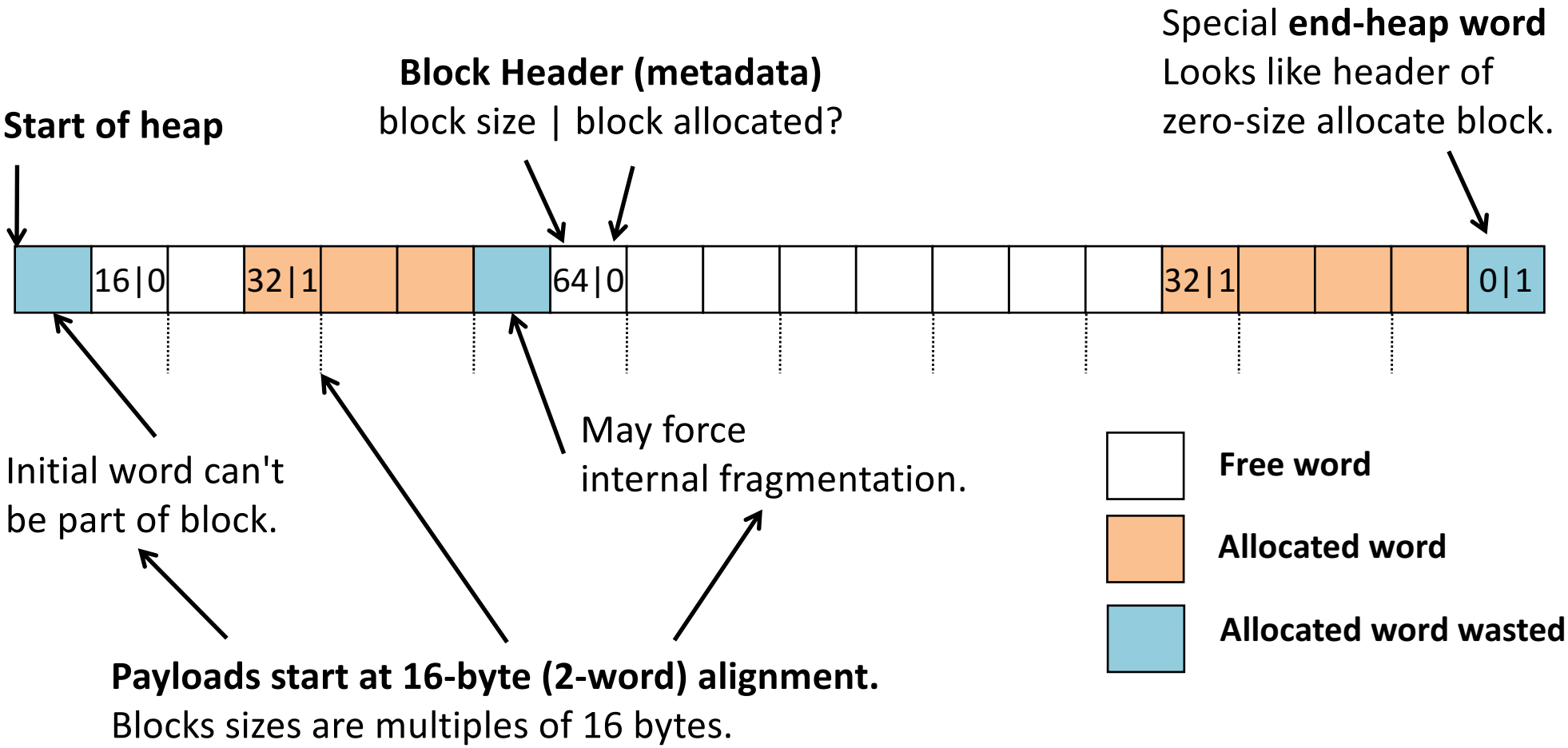


Steal LSB for status flag.
LSB = 1: allocated
LSB = 0: free

16-byte aligned sizes have
4 zeroes in low-order bits

0000**0000**
0001**0000**
0010**0000**
0011**0000**
…

Implicit Free List: **Heap Layout**



Implicit Free List: **Finding a Free Block**

First fit:

Search list from beginning, choose ***first*** free block that fits

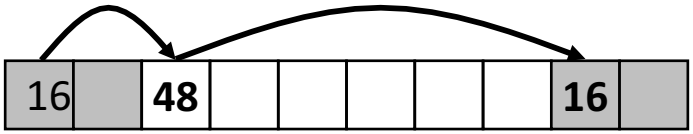
Next fit:

Do first-fit starting where previous search finished

Best fit:

Search the list, choose the ***best*** free block: fits, with fewest bytes left over

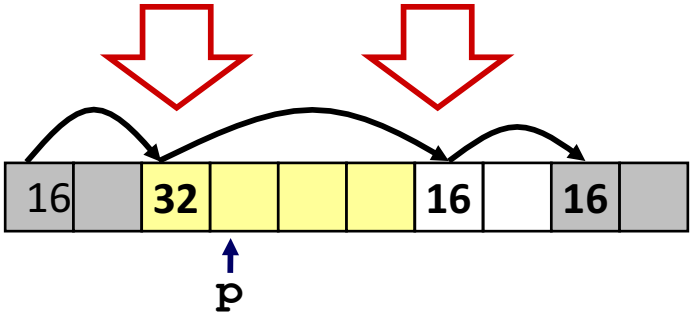
Implicit Free List: **Allocating a Free Block**




```
p = malloc(24);
```

Allocated space **≤** free space.

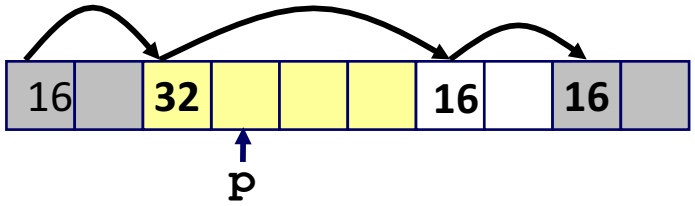
Use it all? Split it up?



Block **Splitting**

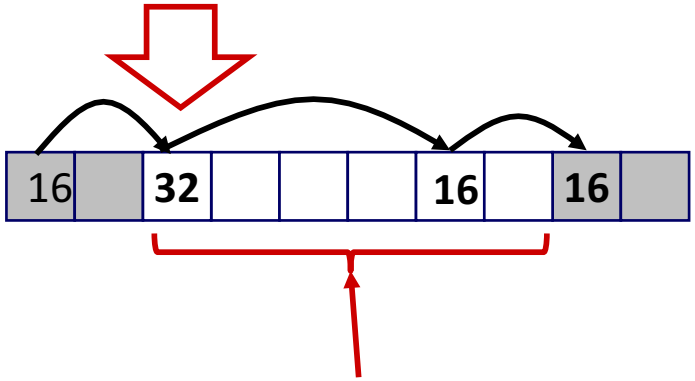
Now showing allocation status flag implicitly with shading.

Implicit Free List: **Freeing a Block**

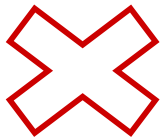


`free(p);`

Clear *allocated* flag.



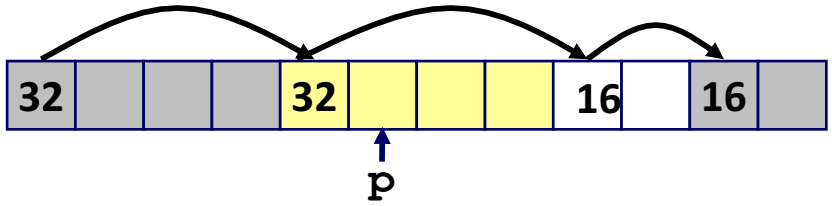
`malloc(40);`



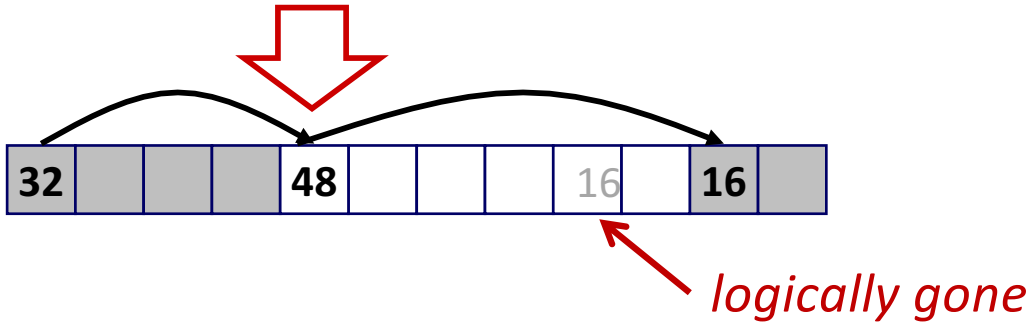
External fragmentation!

Enough space, not one block.

Coalescing Free Blocks



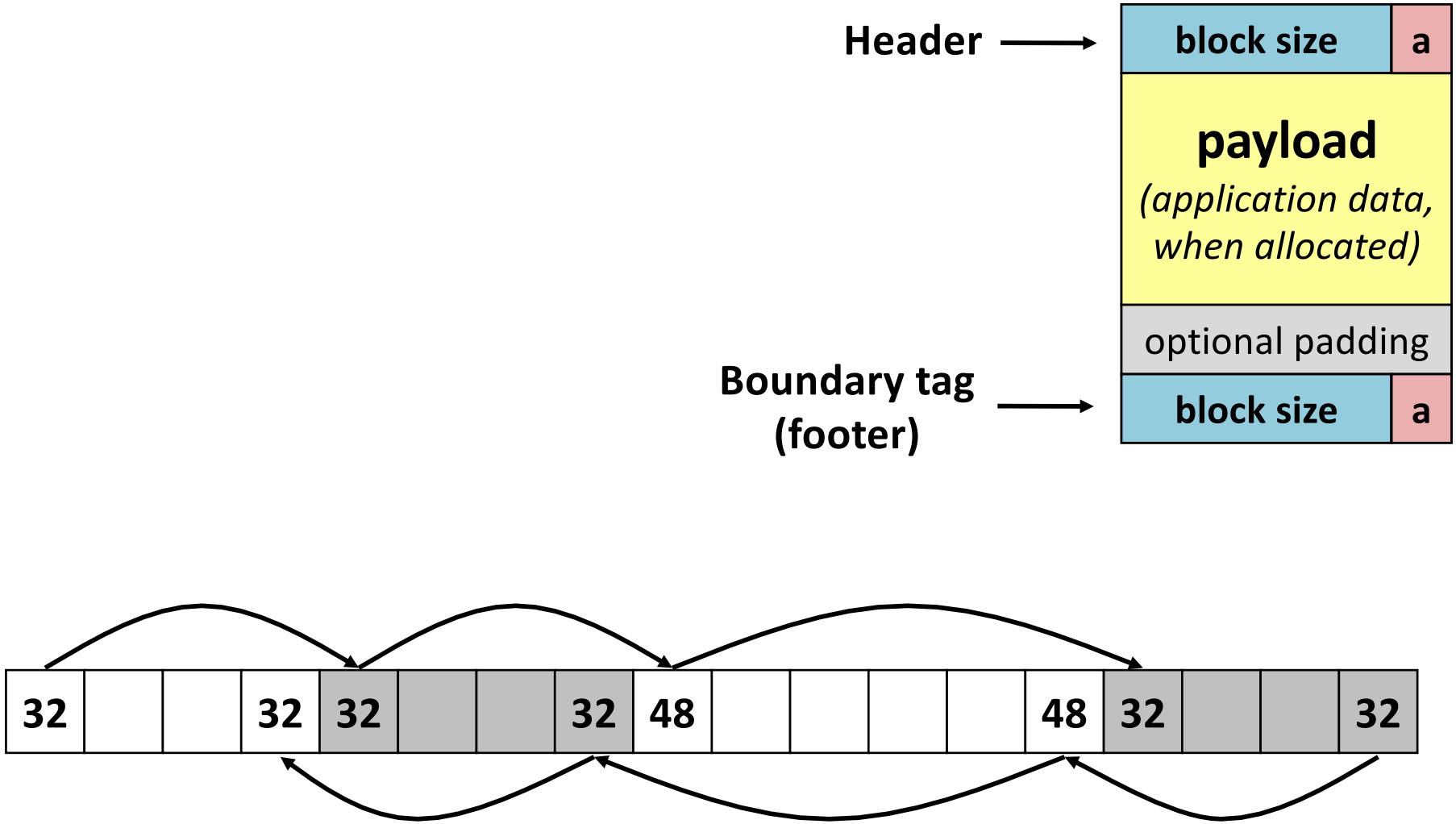
`free(p)` **Coalesce** with following *free* block.



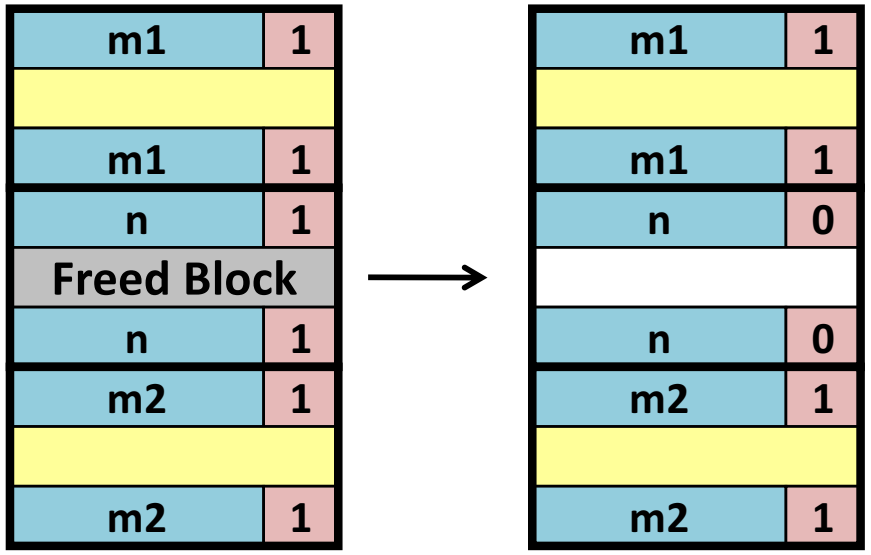
Coalesce with **preceding** *free* block?

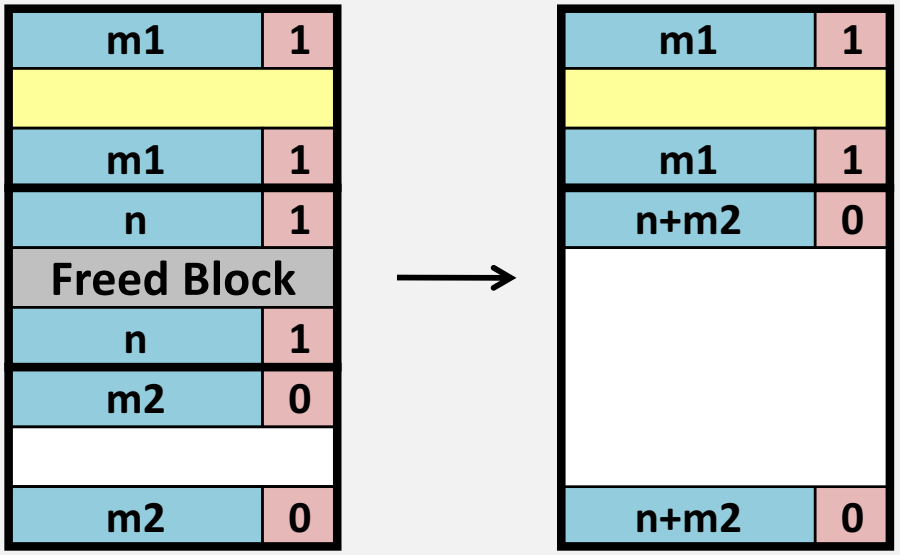
[Knuth73]

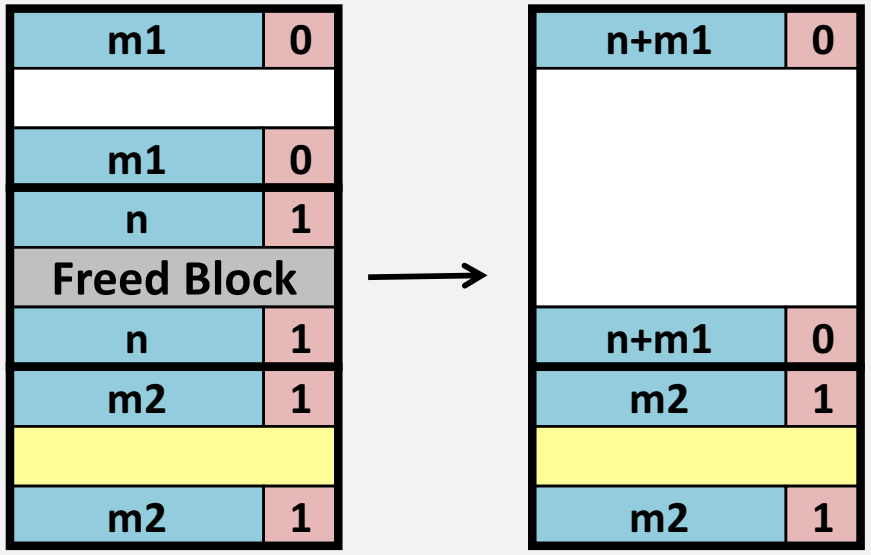
Bidirectional Coalescing: **Boundary Tags**

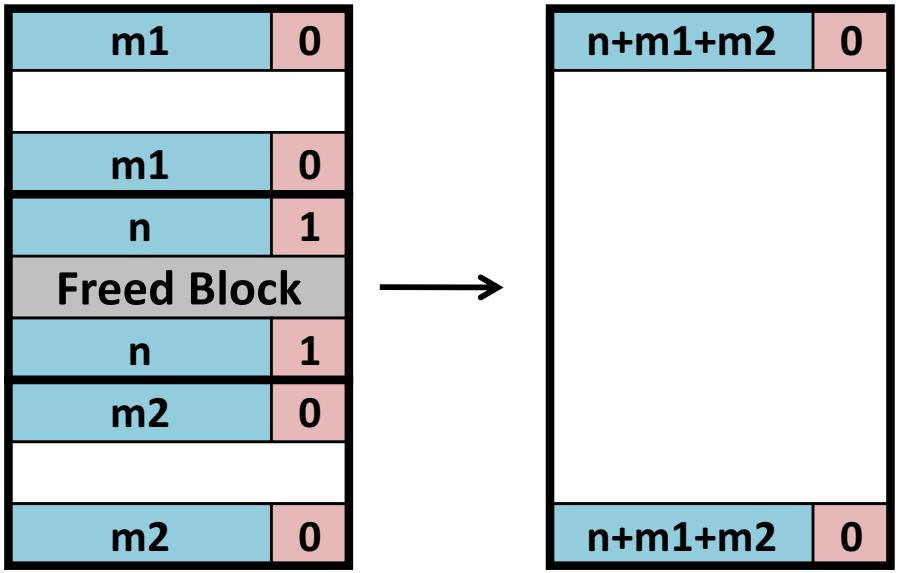


Constant-Time Coalescing: 4 cases









Summary: **Implicit Free Lists**

Implementation: **simple**

Allocate: **O(blocks in heap)**
Free: **O(1)**

Memory utilization: depends on placement policy

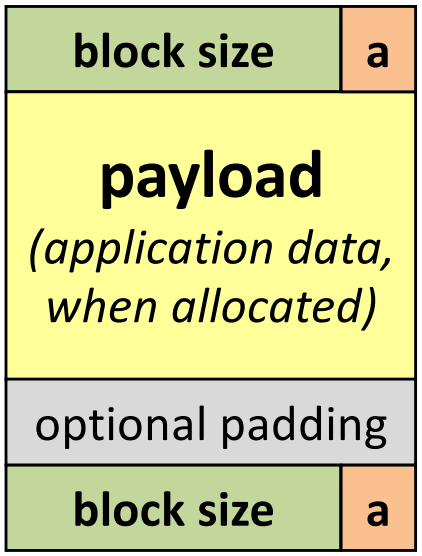
Not widely used in practice

some special purpose applications

Splitting, boundary tags, coalescing are general to ***all*** allocators.

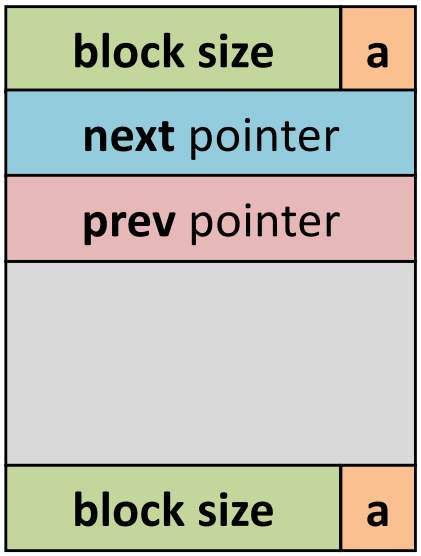
Explicit Free Lists

Allocated block:



(same as implicit free list)

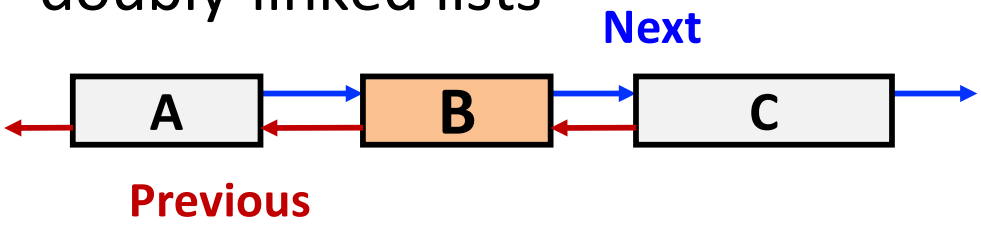
Free block:



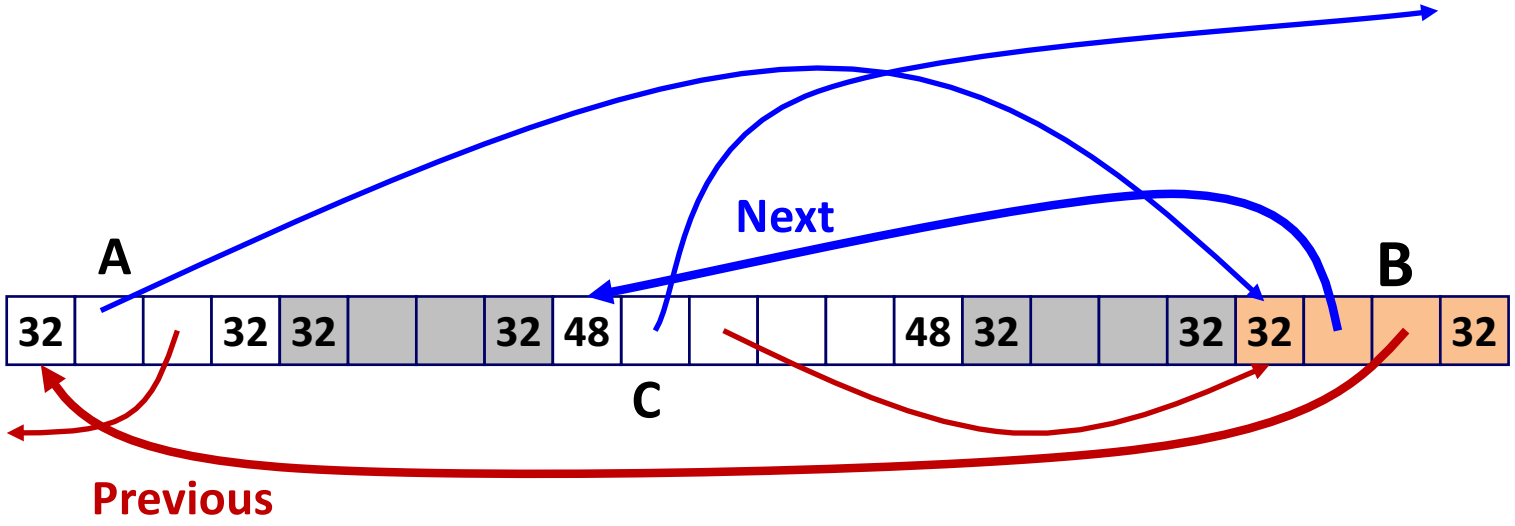
Explicit list of *free* blocks rather than **implicit list of *all* blocks.**

Explicit Free Lists: **List vs. Memory Order**

Abstractly: doubly-linked lists



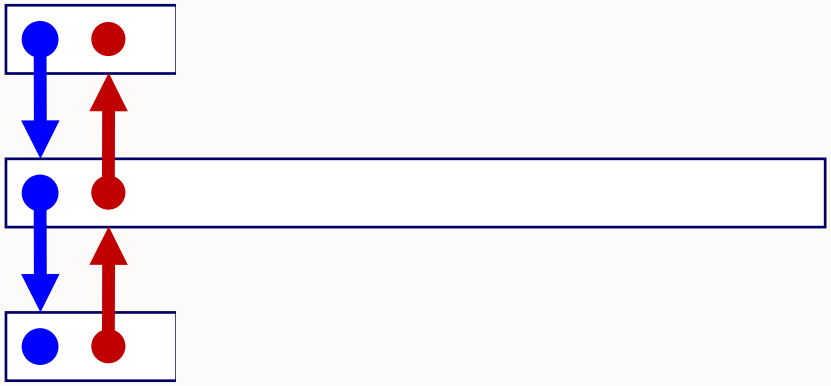
Concretely: free list blocks in any **memory** order



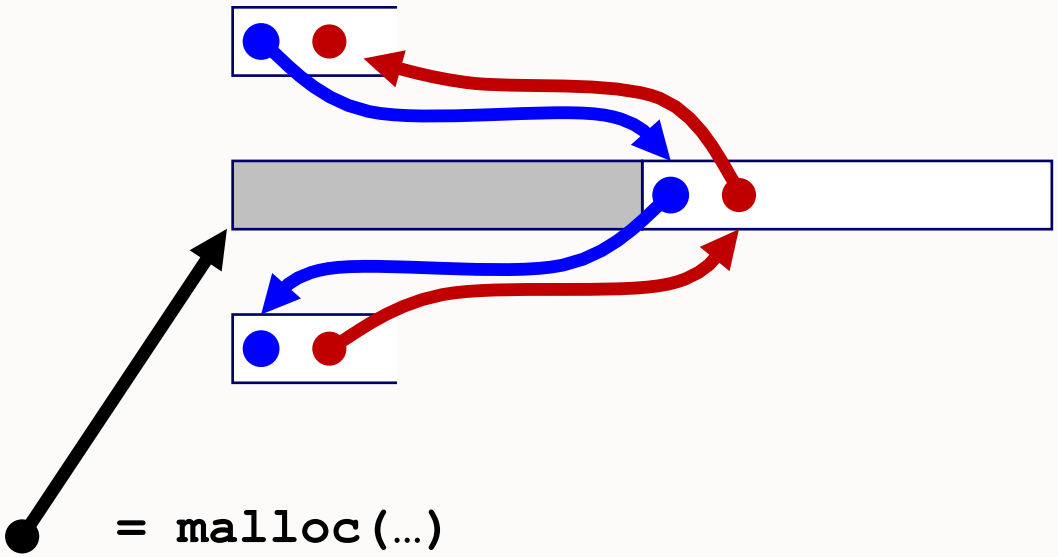
List Order ≠ Memory Order

Explicit Free Lists: **Allocating a Free Block**

Before



After ***(with splitting)***



Explicit Free Lists: **Freeing a Block**

***Insertion policy*:** Where in the free list do you add a freed block?

LIFO (last-in-first-out) policy

Pro: simple and constant time

Con: studies suggest fragmentation is worse than address ordered

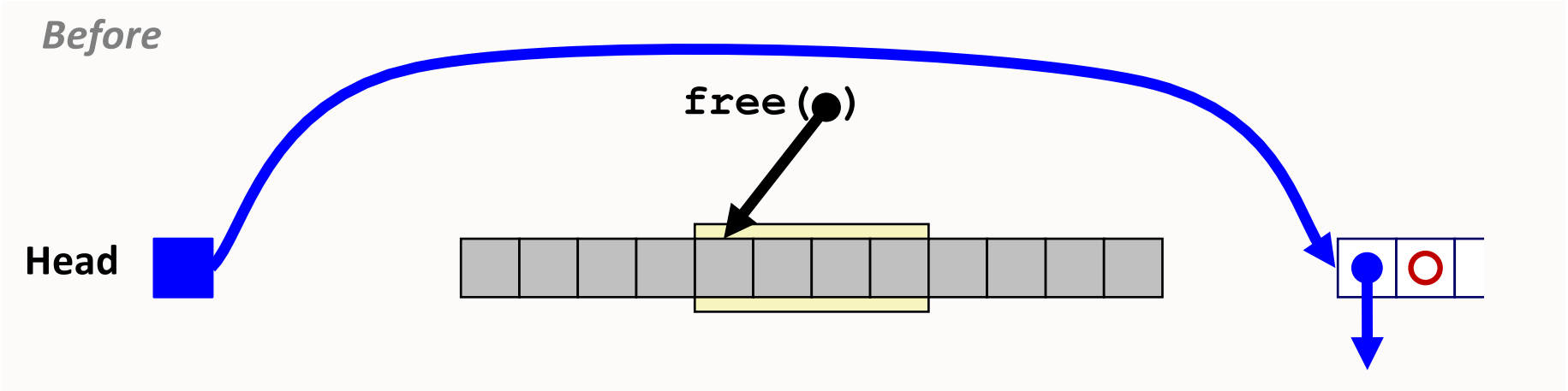
Address-ordered policy

Con: linear-time search to insert freed blocks

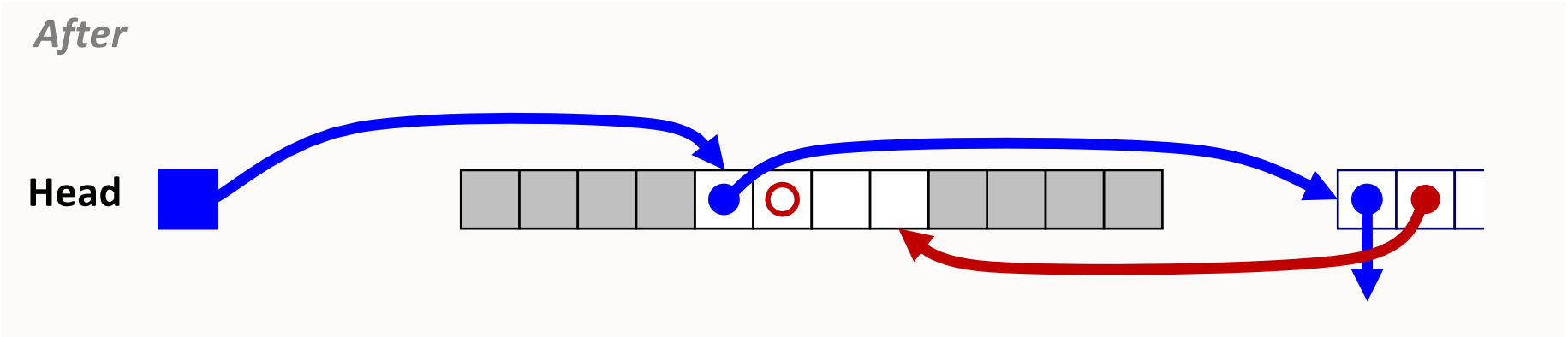
Pro: studies suggest fragmentation is lower than LIFO

LIFO Example: 4 cases of freed block neighbor status.

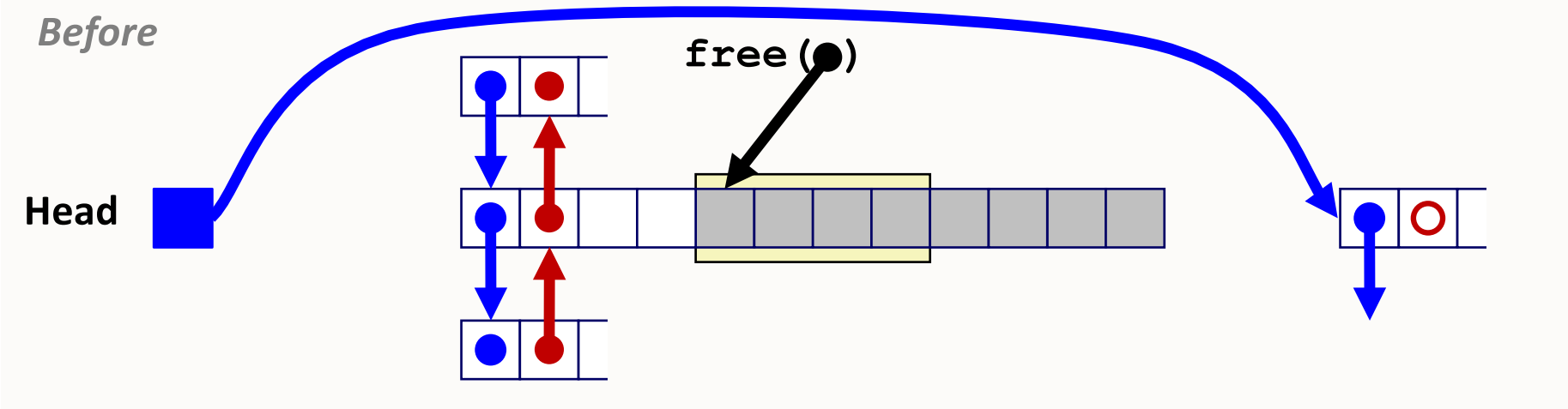
Freeing with LIFO Policy: **between allocated blocks**



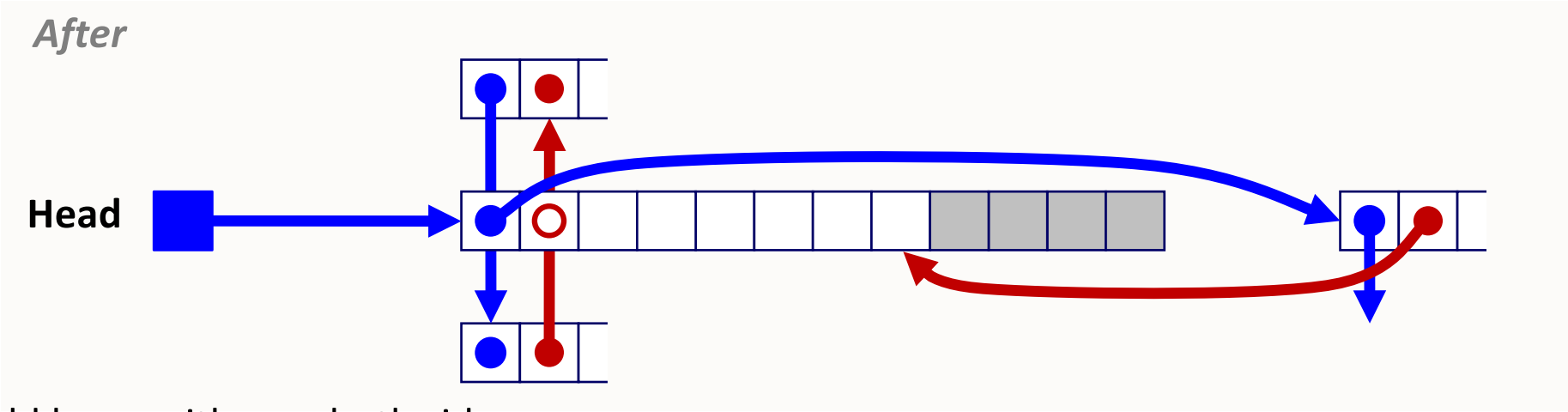
Insert the freed block at head of free list.



Freeing with LIFO Policy:
between free and allocated

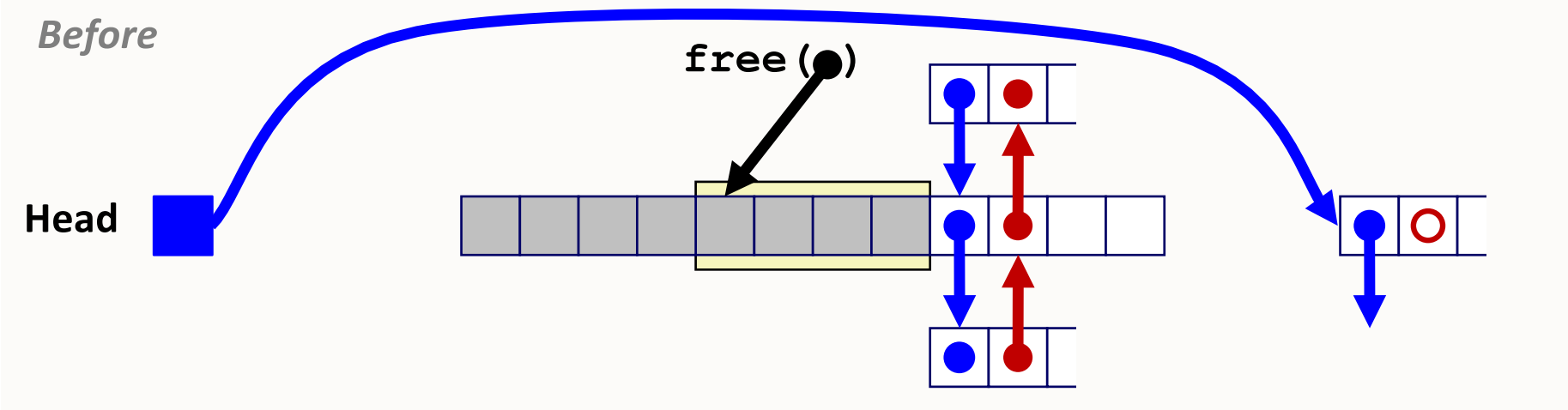


Splice out predecessor block, coalesce both memory blocks, and insert the new block at the head of the free list.

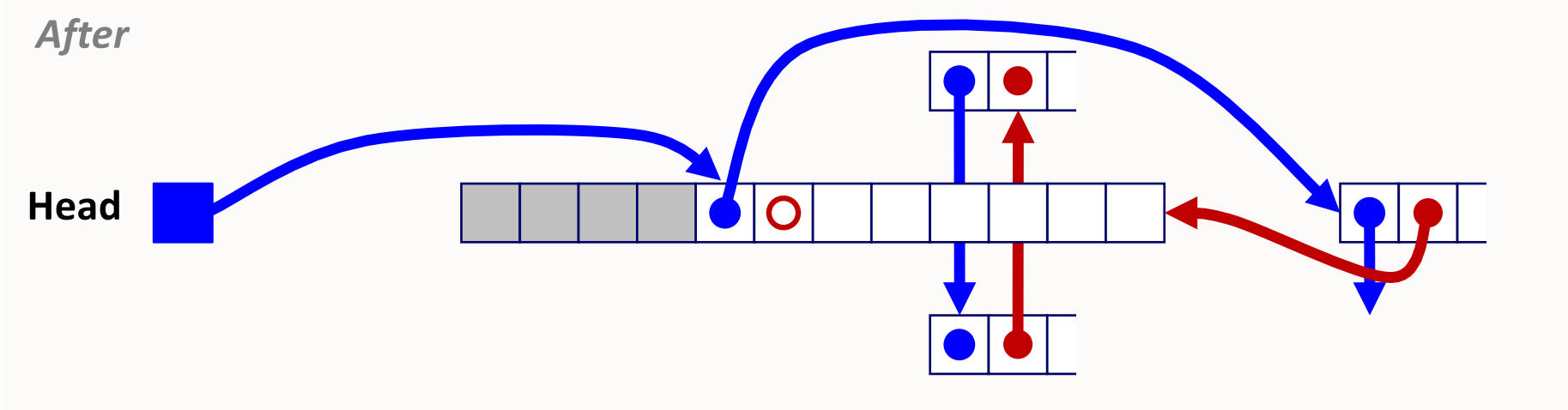


Could be on either or both sides...

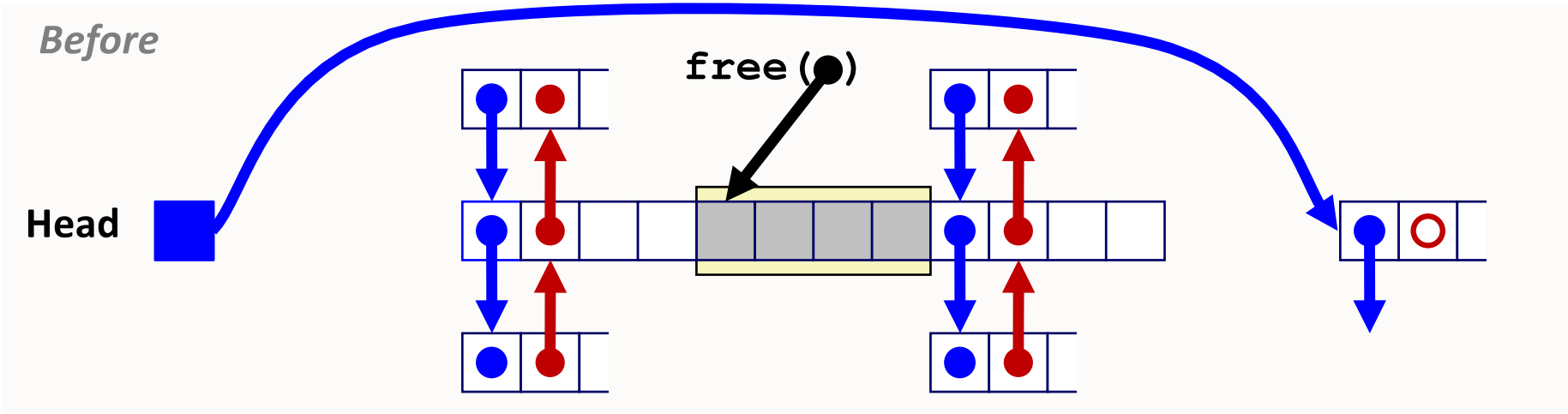
Freeing with LIFO Policy:
between allocated and free



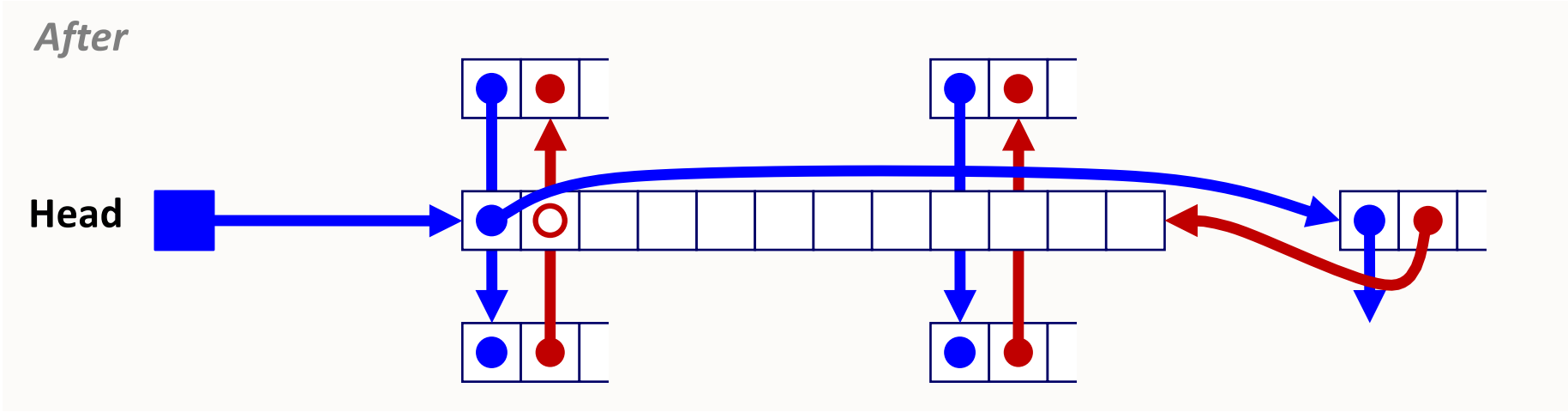
Splice out successor block, coalesce both memory blocks and insert the new block at the head of the free list.



Freeing with LIFO Policy: **between free blocks**



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



Summary: **Explicit Free Lists**

Implementation: **fairly simple**

Allocate:	**O(*free* blocks)**	vs. O(*all* blocks)
Free:	**O(1)**	vs. O(1)

Memory utilization:

depends on placement policy

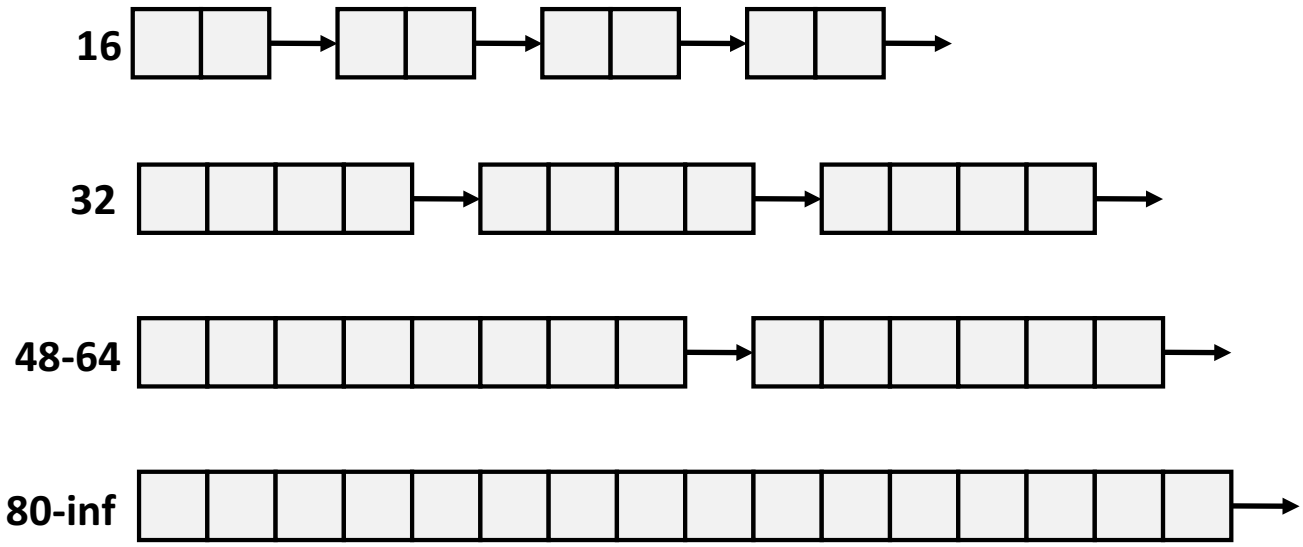
larger minimum block size (next/prev) vs. implicit list

Used widely in practice, often with more optimizations.

Splitting, boundary tags, coalescing are general to ***all*** allocators.

Seglist Allocators

Each *size bracket* has its own free list



Faster best-fit allocation...

Summary: **Allocator Policies**

All policies offer trade-offs in fragmentation and throughput.

Placement policy:

First-fit, next-fit, best-fit, etc.

Seglists approximate best-fit in low time

Splitting policy:

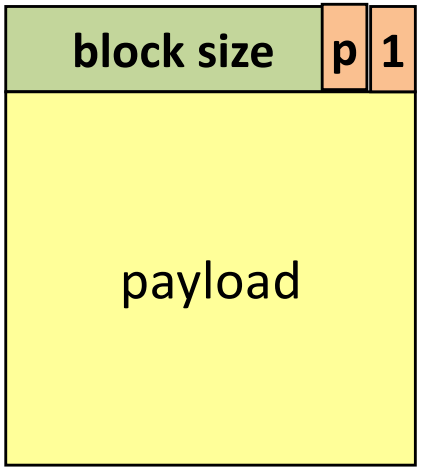
Always? Sometimes? Size bound?

Coalescing policy:

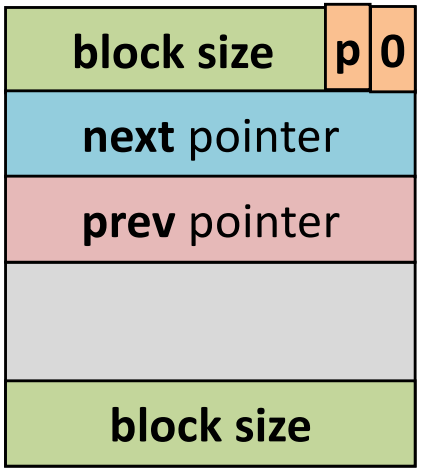
Immediate vs. deferred

Remembrallocator Block Format

Allocated block:



Free block:



Minimum block size?

- Implicit free list
- Explicit free list

Update 2 headers on each malloc/free.