CS 240
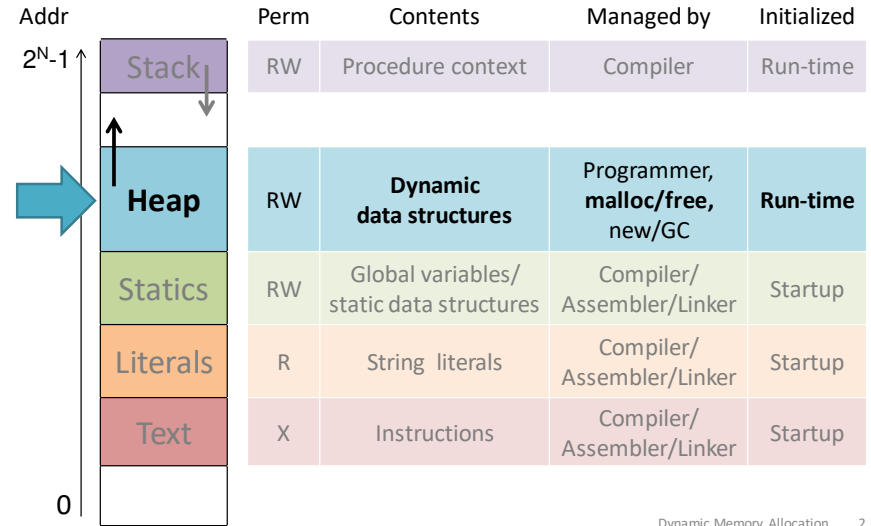Foundations of Computer Systems

WELLESLEY

Dynamic Memory Allocation in the Heap

Explicit allocators

Manual memory management

C: implementing malloc and free

Heap Allocation

Addr		Perm	Contents	Managed by	Initialized
2^N-1 ↑	Stack ↓	RW	Procedure context	Compiler	Run-time
→	**Heap**	RW	**Dynamic data structures**	**Programmer, malloc/free, new/GC**	**Run-time**
	Statics	RW	Global variables/ static data structures	Compiler/ Assembler/Linker	Startup
	Literals	R	String literals	Compiler/ Assembler/Linker	Startup
	Text	X	Instructions	Compiler/ Assembler/Linker	Startup
0					

Allocator basics

Pages too coarse-grained for allocating individual objects.
Instead: flexible-sized, word-aligned blocks.

☐ **Free word**
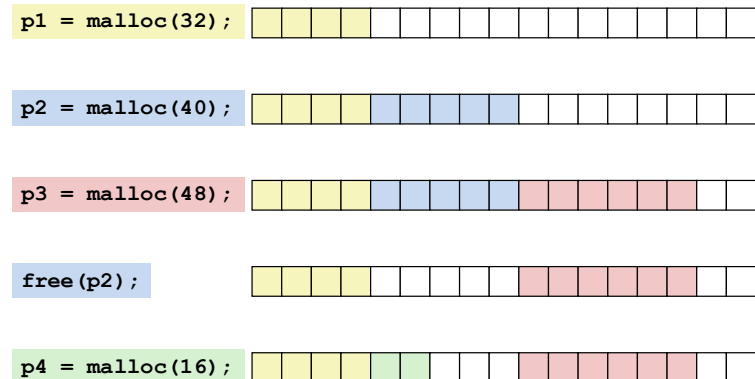▨ **Allocated word**

Allocated block (4 words)   **Free block (3 words)**
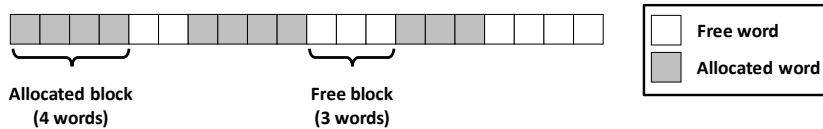
pointer to newly allocated block of at least that size

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);
```
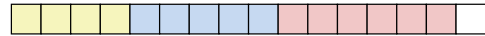
Allocator goals: malloc/free

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

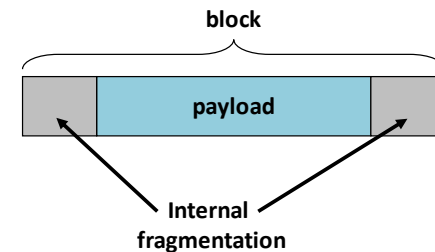**3. High memory utilization.**

  Most of heap contains necessary program data.
  Little wasted space.

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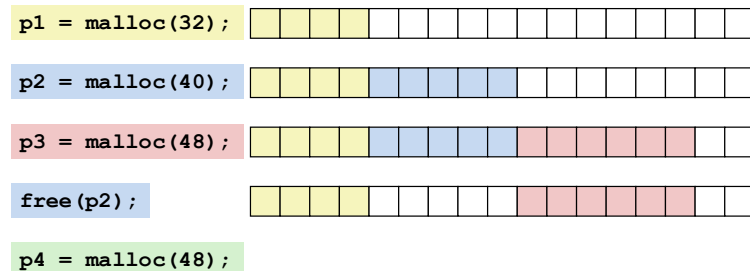
Internal fragmentation

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!

p0

p0 = malloc(32);

48

block size metadata   **data payload**

free(p0);
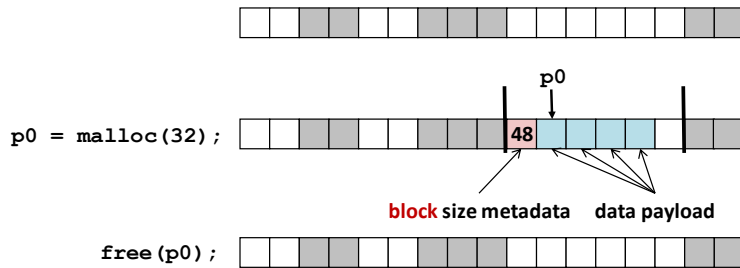
Keeping track of free blocks

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

| 40 | | | 32 | | | 48 | | | | 16 | |

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

| 40 | | | 32 | | | 48 | | | | 16 | |

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
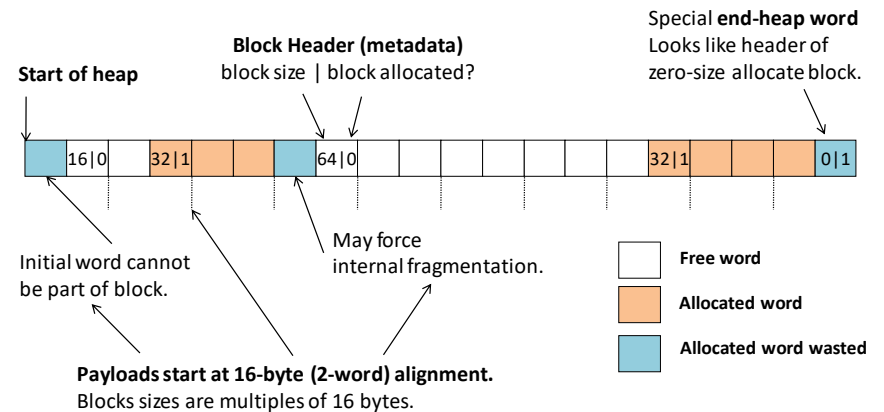2. Allocation status

Store in one header word.

1 word

block size | a

payload
*(application data,
when allocated)*

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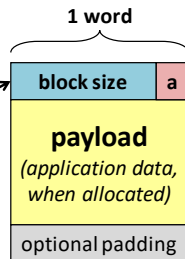
Steal LSB for status flag.
LSB = 1: allocated
LSB = 0: free

16-byte aligned sizes have
4 zeroes in low-order bits
00000**0000**
00010**0000**
00100**0000**
00110**0000**
…

Implicit free list: **heap layout**

Block Header (metadata)
block size | block allocated?

Special **end-heap word**
Looks like header of
zero-size allocate block.

Start of heap

| 16|0 | | 32|1 | | | 64|0 | | | | | | 32|1 | | | 0|1 |

May force
internal fragmentation.

Initial word cannot
be part of block.

Payloads start at 16-byte (2-word) alignment.
Blocks sizes are multiples of 16 bytes.

☐ **Free word**

🟧 **Allocated word**

🟦 **Allocated word wasted**

Implicit free list: **finding a free block**

First fit:

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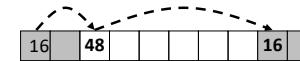
Next fit:

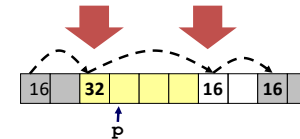Do first-fit starting where previous search finished

Best fit:

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

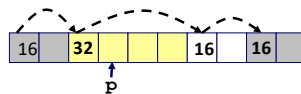# Implicit free list: **allocating a free block**

`p = malloc(24);`

Allocated space $\leq$ free space.
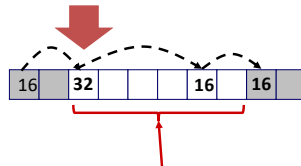Use it all? Split it up?

Block **Splitting**

Now showing allocation status flag implicitly with shading.

Implicit free list: **freeing an allocated block**
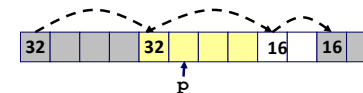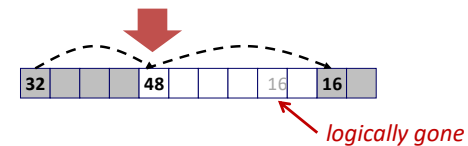

`free(p);` **Clear *allocated* flag.**

`malloc(40);` ✖ **External fragmentation!**
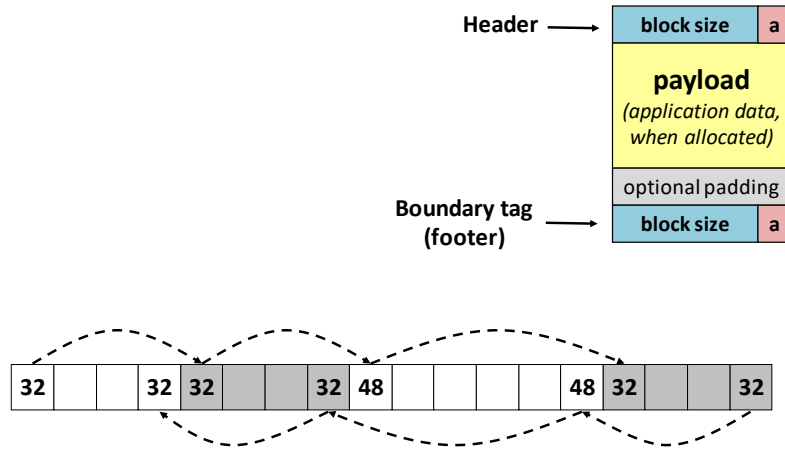Enough space, not one block.

Coalescing free blocks
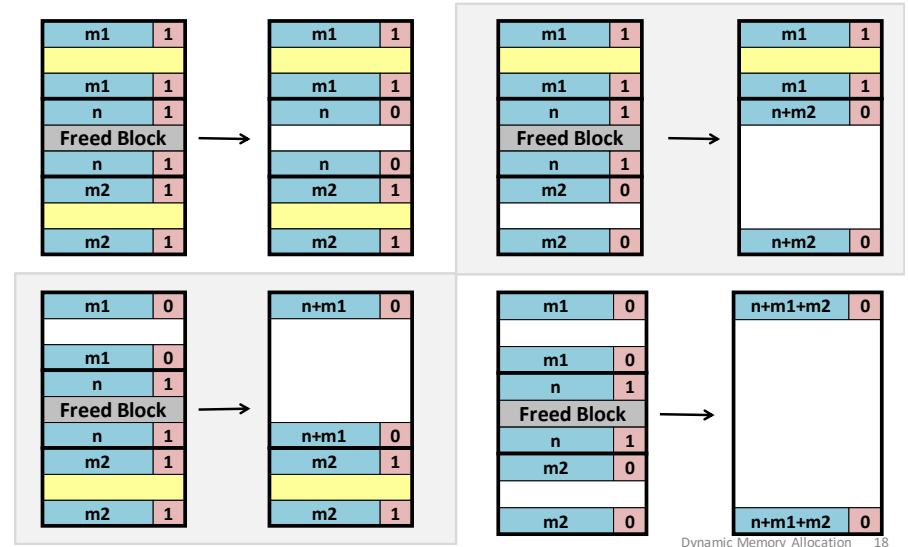

`free(p)` **Coalesce** with following *free* block.

logically gone

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

Bidirectional coalescing: boundary tags

Header →

| block size | a |

payload
(application data, when allocated)

optional padding

Boundary tag (footer) →

| block size | a |

| 32 | | | 32 | 32 | | | 32 | 48 | | | | | 48 | 32 | | | 32 |

Constant-time O(1) coalescing: 4 cases

Summary: **implicit free lists**

Implementation: simple

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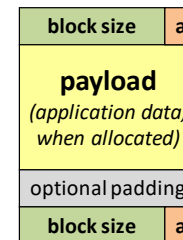**Free**:   O(1)

Memory utilization: depends on placement policy

Not widely used in practice
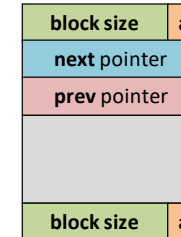some special purpose applications

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

Explicit free list: block format

Allocated block:

| block size | a |

payload
(application data, when allocated)

optional padding

| block size | a |

(same as implicit free list)

Free block:

| block size | a |
| **next** pointer |
| **prev** pointer |
| |
| block size | a |

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

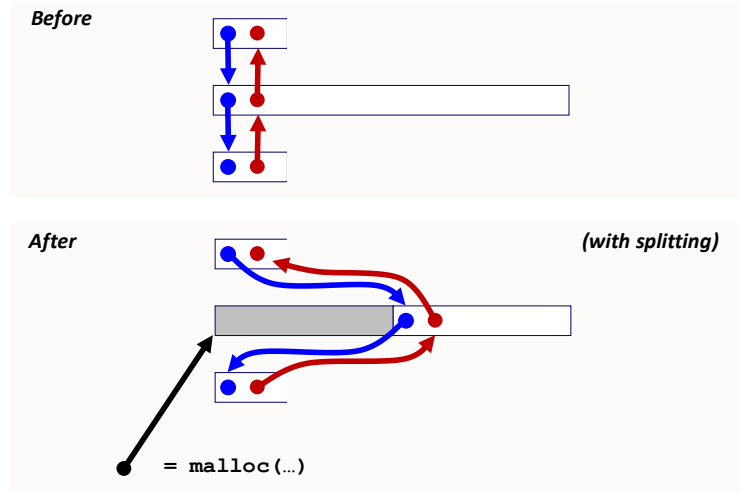# Explicit free list: **list vs. memory order**

Abstractly: doubly-linked lists
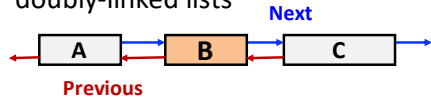
Concretely: free list blocks in any memory order
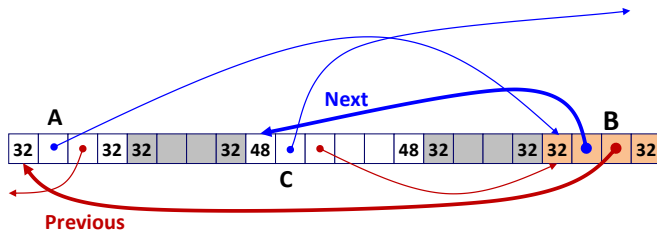
List Order ≠ Memory Order

Explicit free list: **allocating a free block**

Before

After   *(with splitting)*

= malloc(…)

Explicit free list: **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
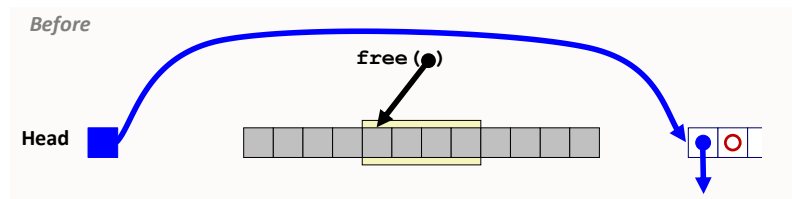
Address-ordered policy
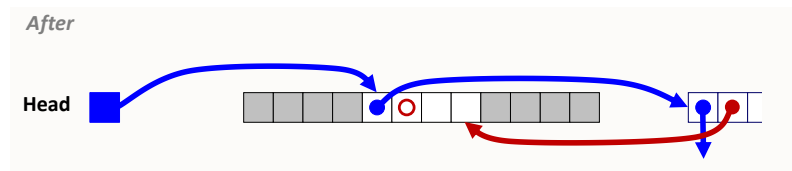 *Con:* linear-time search to insert freed blocks
  Pro: studies suggest fragmentation is lower than LIFO

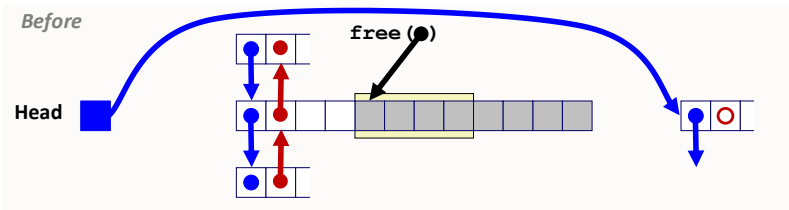LIFO Example: 4 cases of freed block neighbor status.
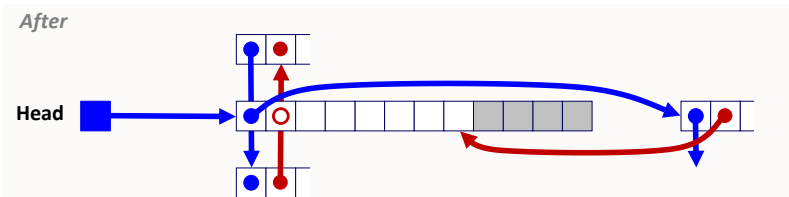
Freeing with LIFO policy: **between allocated blocks**

Before

free(●)

Head

Insert the freed block at head of free list.

After

Head

Freeing with LIFO policy:
between free and allocated
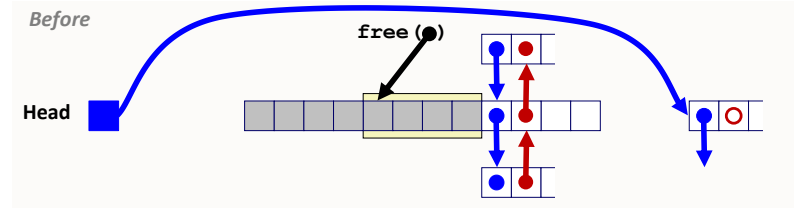

Before

Head

free(●)

After

Head

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

Before

Head

free(●)

After

Head

Splice out successor block, coalesce both memory blocks and insert the new block at the head of the free list.
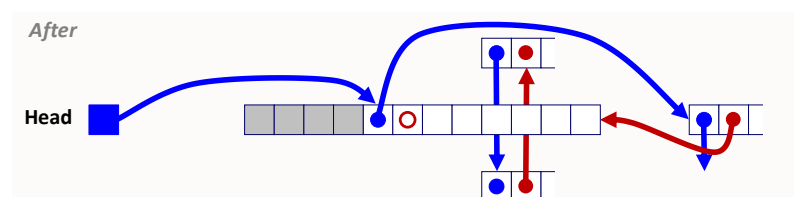
Freeing with LIFO policy:
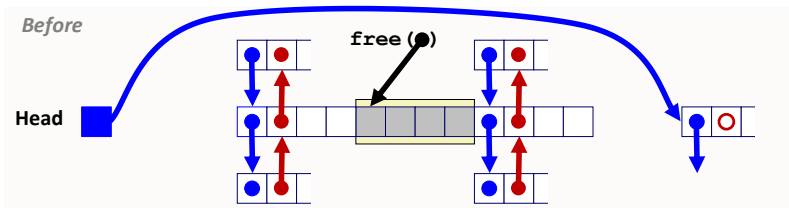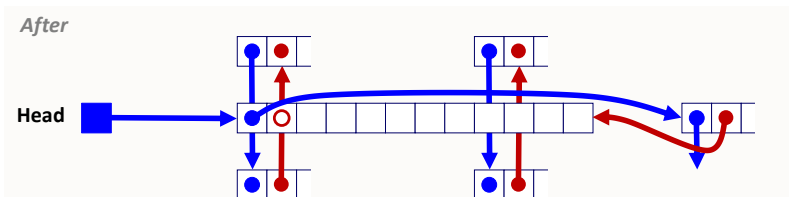## **between free blocks**

Before

Head

free(●)

After

Head

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:
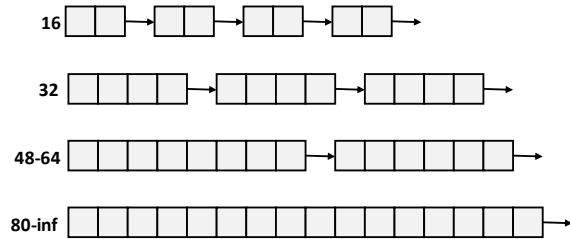   depends on placement policy
   larger minimum block size (next/prev) vs. implicit list

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

16 ▢▢ → ▢▢ → ▢▢ → ▢▢ →

32 ▢▢▢ → ▢▢▢ → ▢▢▢ →

48-64 ▢▢▢▢▢▢ → ▢▢▢▢▢ →

80-inf ▢▢▢▢▢▢▢▢▢▢▢▢ →

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

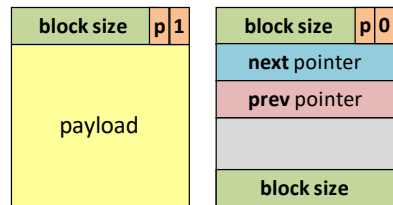**Splitting policy:**
Always? Sometimes? Size bound?

Coalescing policy:
Immediate vs. deferred

Improved block format

Allocated block: Free block:

| block size | p | 1 |

payload

| block size | p | 0 |
| **next** pointer |
| **prev** pointer |

| block size |

Minimum block size?
- Implicit free list
- Explicit free list

Update headers of 2 blocks on each malloc/free.

payload

| block size | 1 | 0 |
| **next** pointer |
| **prev** pointer |

| block size |
| block size | 0 | 1 |

payload

| block size | 1 | 1 |

payload