Verifying malloc

John Wickerson / Mike Dodds / Matthew Parkinson
University of Cambridge

Explicit Stabilisation

$$\frac{\vdash \{p\}\ c\ \{q\} \qquad p\ \text{stab}\ R \qquad p \rightsquigarrow q \subseteq G \qquad q\ \text{stab}\ R}{R,G \vdash \{\lfloor p \rfloor_R\}\ c\ \{\lceil q \rceil_R\}}\ \text{Basic}$$

$$p \ ::= \ \ldots \ | \ \lfloor p \rfloor_R \ | \ \lceil p \rceil_R$$

$$\lfloor p \rfloor_R = \bigvee \{q \mid q \Rightarrow p \wedge q\ \text{stab}\ R\}$$

$$\lceil p \rceil_R = \bigwedge \{q \mid q \Leftarrow p \wedge q\ \text{stab}\ R\}$$

Natural specifications

The malloc(nb) function allocates nb bytes of memory and returns a pointer to the allocated memory.

The free(ptr) function deallocates the memory allocation pointed to by ptr. If ptr is a NULL pointer, no operation is performed.

emp

x := malloc(n×wordsize)

$\circledast_{0 \leq i < n}\ x{+}i \mapsto \_$

$\circledast_{0 \leq i < n}\ x{+}i \mapsto \_$

free(x)

emp

Malloc rounds up to a whole number of words.
Discount, for now, the possibility that malloc fails.

Natural specifications

The malloc(nb) function allocates nb bytes of memory and returns a pointer to the allocated memory.

emp

x := malloc(n×wordsize)

token(x,n)
$* \circledast_{0 \leq i < n}\ x+i \mapsto \_$

The free(ptr) function deallocates the memory allocation pointed to by ptr. If ptr is a NULL pointer, no operation is performed.

token(x,n)
$* \circledast_{0 \leq i < n}\ x+i \mapsto \_$

free(x)

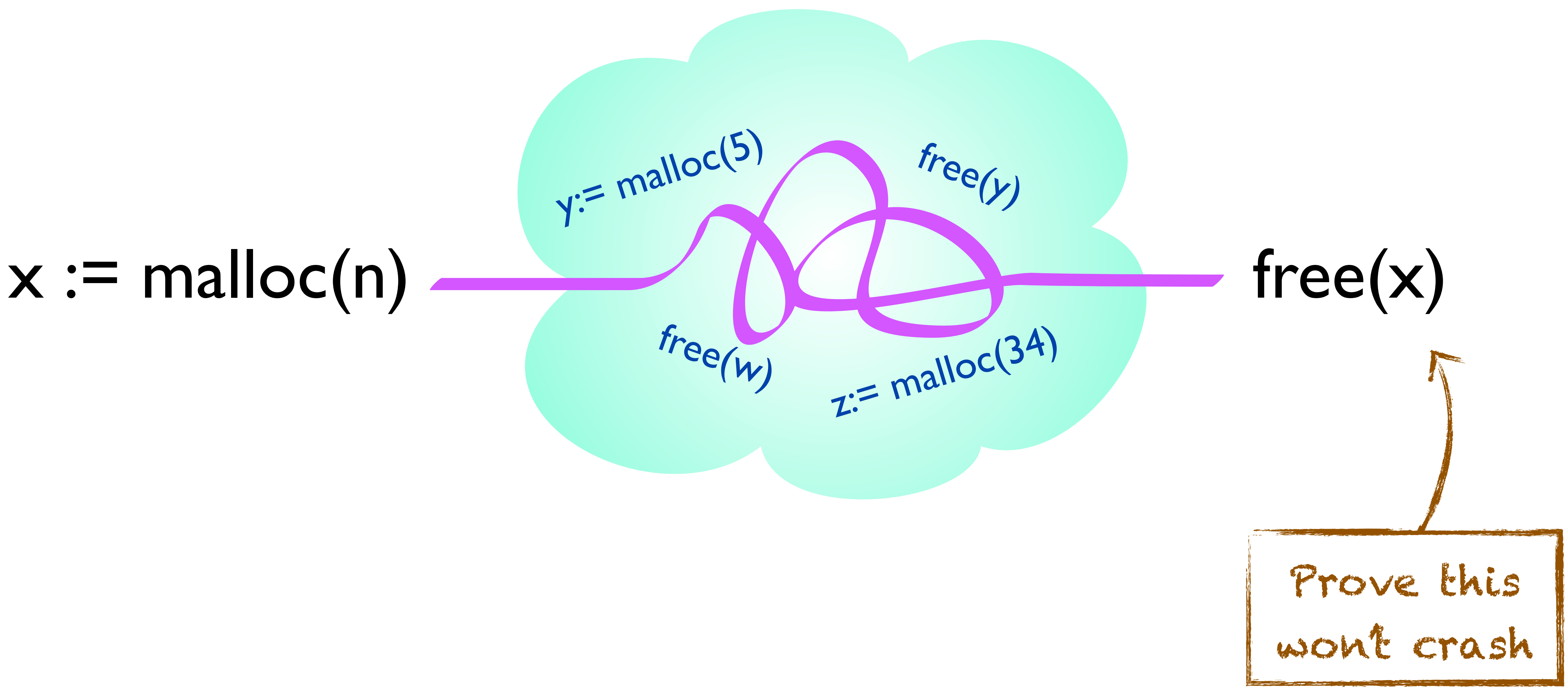
emp

token(x,n) is an abstract spatial predicate, used to prove to free that the block being returned was allocated by malloc.
Abstract ⇒ its definition is out of scope ⇒ it cannot be faked.

Spatial ⇒ cannot be duplicated.

Hopefully we can reuse some part of the existing state as the token, or else we need some auxiliary state.

The crux of the proof



Prove that the call to free(x) won't crash. The information required to prove this has to come from the upstream call to malloc, via a sea of other calls to malloc and free.

Including malloc's internal state

Will arena-with-gap(x,n) still hold here?

arena

x := malloc(n×wordsize)

arena-with-gap(x,n)
$* \text{ token}(x,n)$
$* \circledast_{0\le i<n}\ x{+}i \mapsto \_$

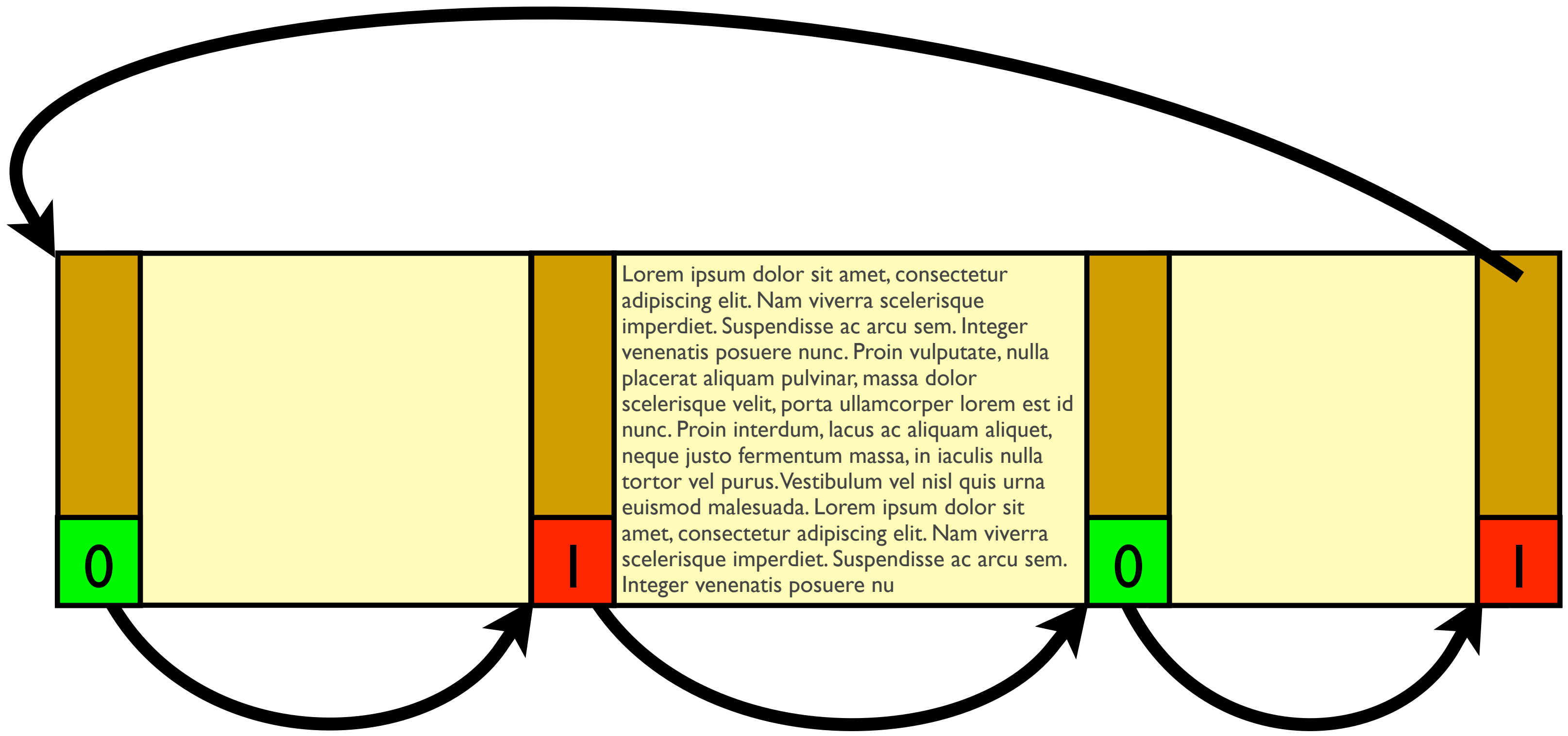
arena-with-gap(x,n)
$* \text{ token}(x,n)$
$* \circledast_{0\le i<n}\ x{+}i \mapsto \_$

free(x)

arena

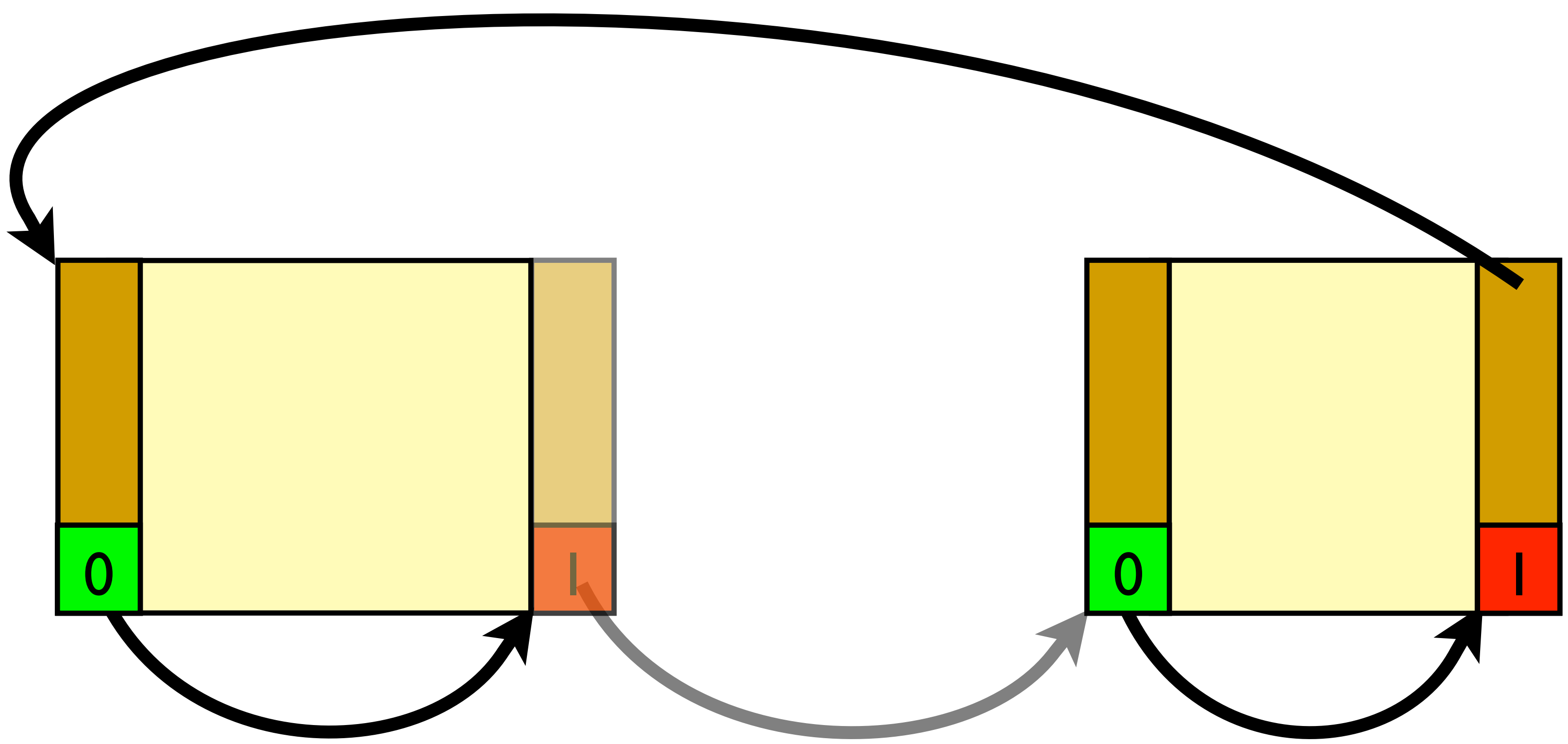
Include malloc's internal state. Shows that the block is not actually created out of thin air.

Version 7 Unix malloc



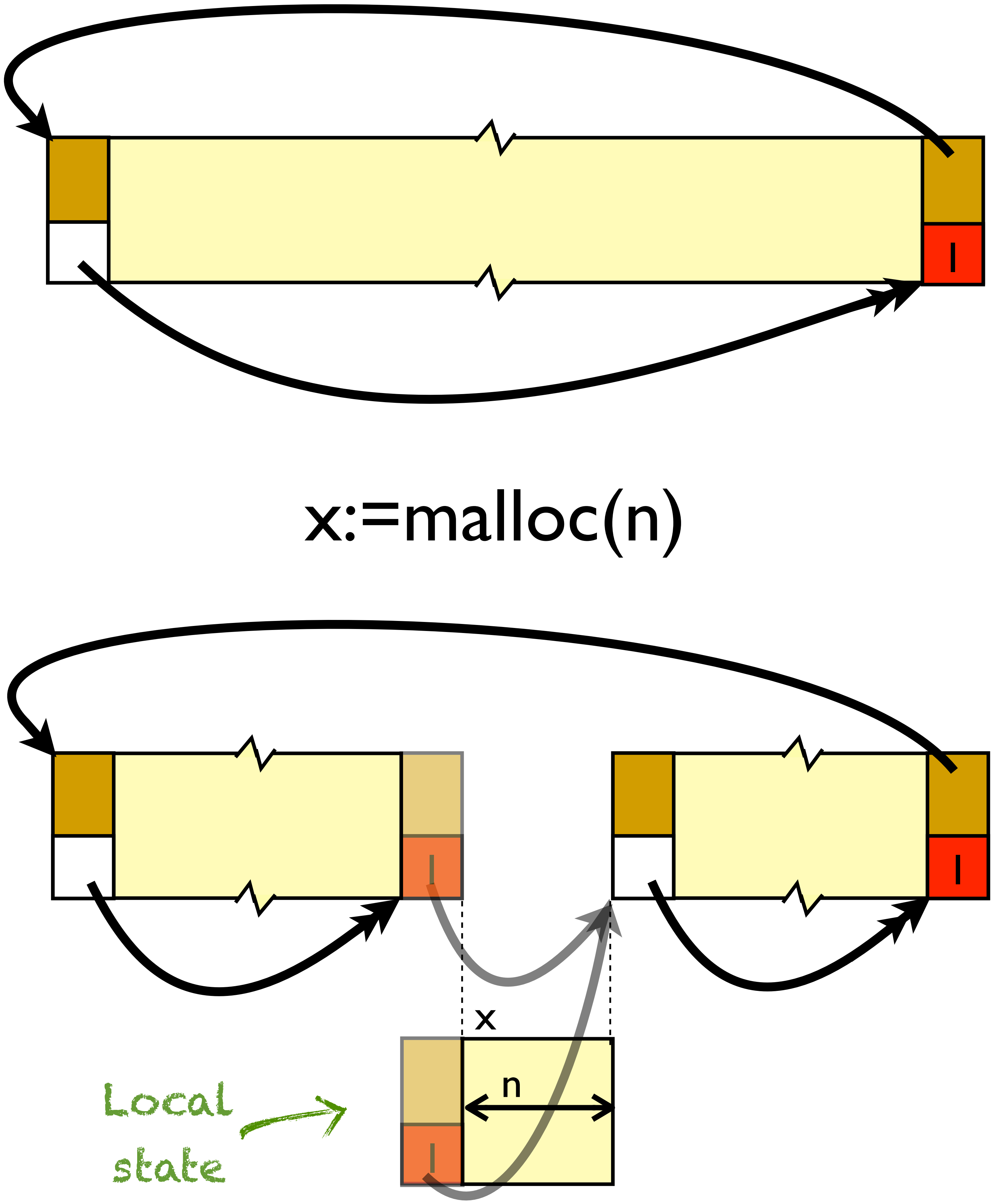
Here's the arena. First-fit strategy. Overhead of one pointer per block (which points to the next block). Blocks are word-aligned, so redundant LSB is used as “busy” flag.

Version 7 Unix malloc



$$\mathrm{token}(x,n) \;\stackrel{\mathrm{def}}{=}\; (x\text{-}1) \overset{0.5}{\mapsto} (x+n)$$

Here’s the internal state. It contains the free blocks, and the linked-list infrastructure. The allocated blocks belong to the respective clients. We use half of the block’s pointer as a token -- half must be kept by malloc so it can continue to traverse the list.



arena

x:=malloc(n)

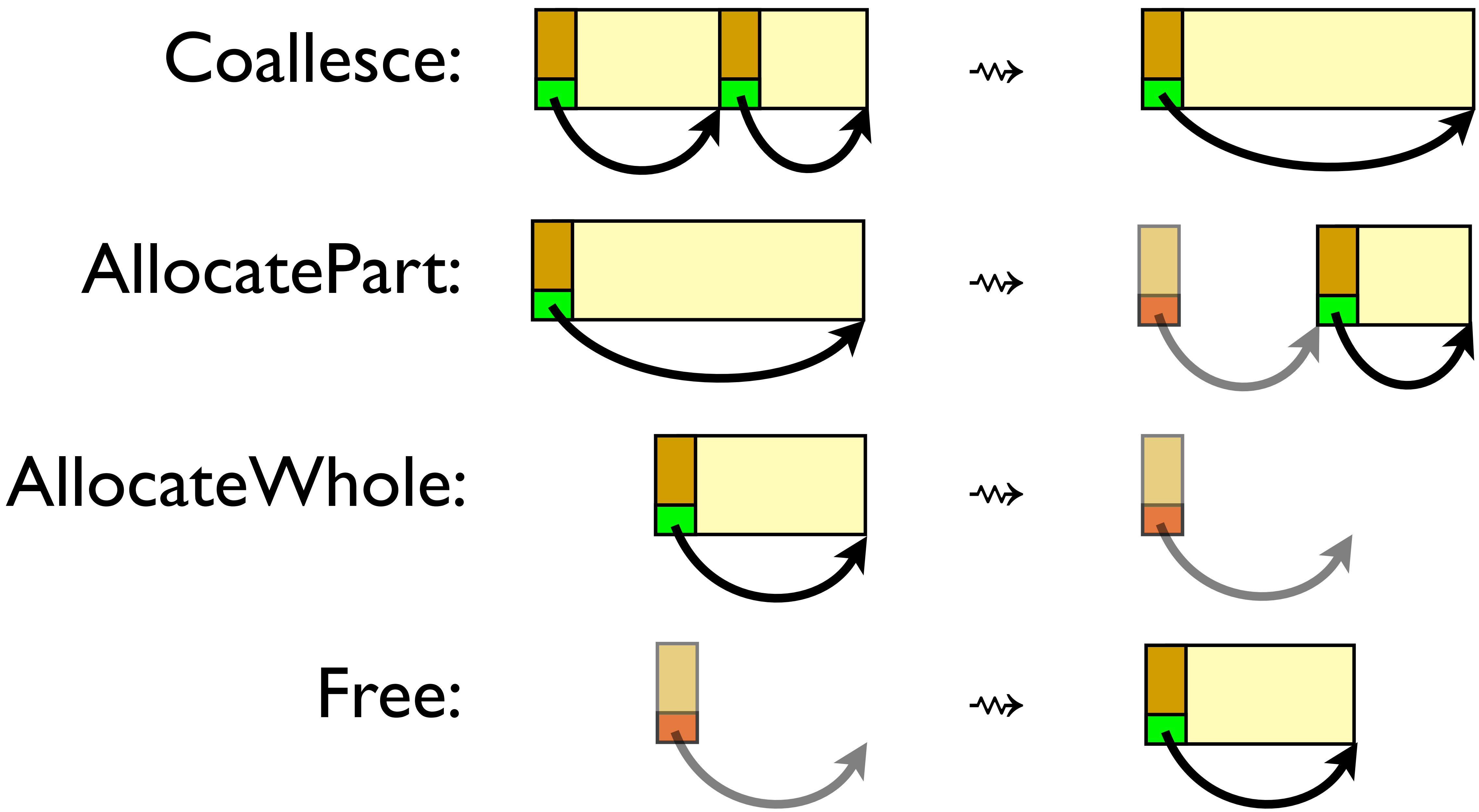
arena-with-gap(x,n)

$* \ \mathrm{token}(x,n)$

$* \ \circledast_{0\leq i<n} \ x+i \mapsto \_$

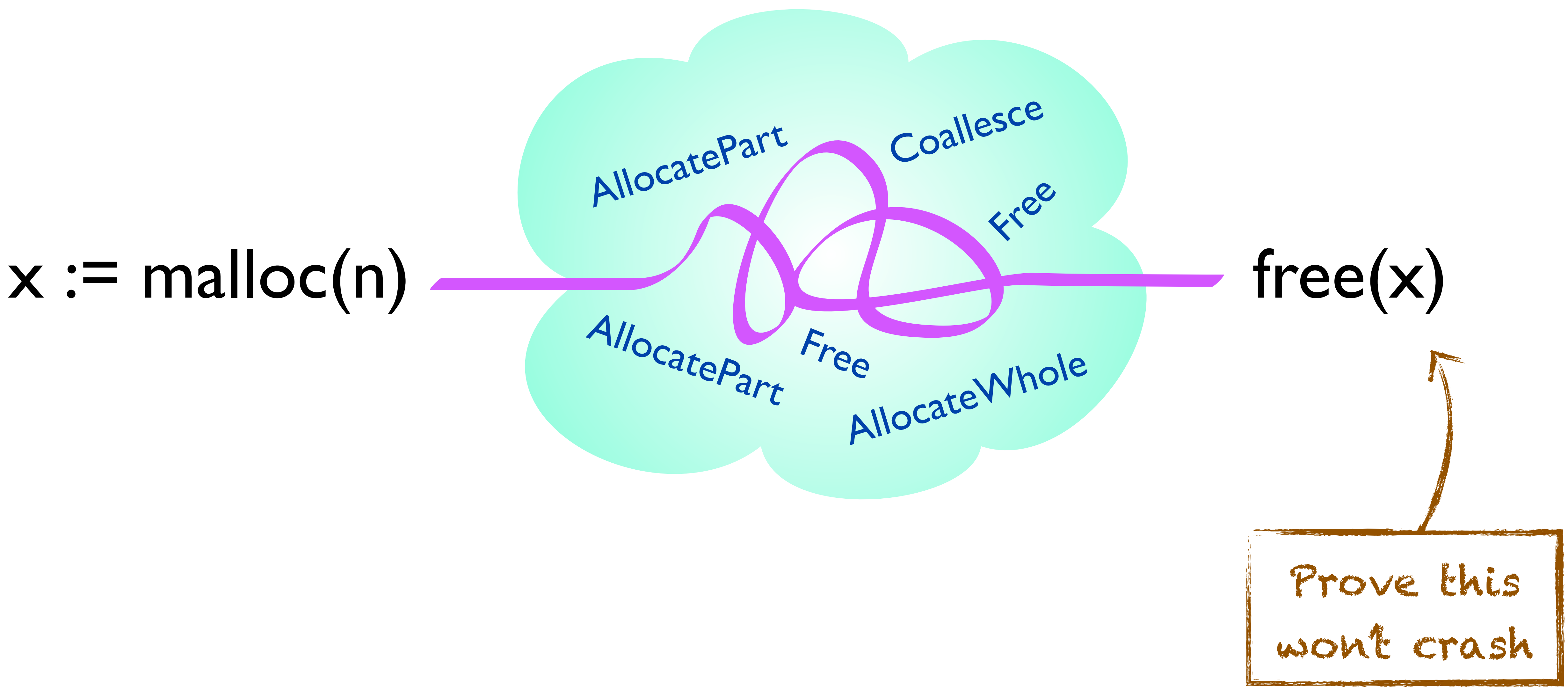
Explain double-headed arrows and zig zags.

Actions



"RGSep style" – but don't go into details of RGSep (nor separation logic). The abstraction is that we have the malloc call, the free call, and inbetween, these actions happen lots of times.

The crux of the proof



Is the arena-with-gap predicate stable under the actions? No – it's not stable under Free. But crucially, the Free action requires the presence of the token in local state, and it can't be present in some other client's local state if it is here in ours!

Stability

arena

x:=malloc(n)

unstable — arena-with-gap(x,n) — stable

$*$ token(x,n)

$* \circledast_{0 \le i < n}\ x+i \mapsto \_$

This is funny because the client's state is immune to interference from other clients, and yet is crucial to the stability of the module's state.

Explicit Stability

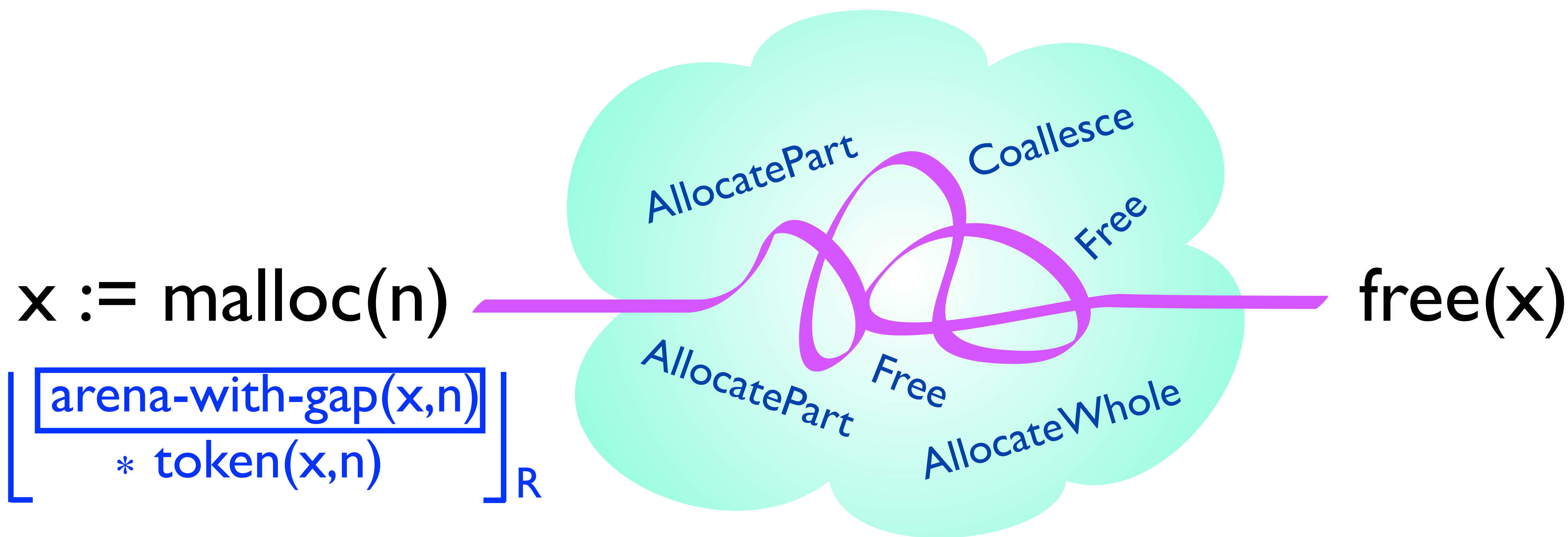
$$\left\lceil \boxed{\text{arena}} \right\rceil_R$$

x:=malloc(n)

$$\left\lfloor \boxed{\text{arena-with-gap}(x,n)} * \text{token}(x,n) \right\rfloor_R * \circledast_{0 \le i < n}\, x+i \mapsto \_$$

The floor and ceiling brackets act as a certificate of stability under a rely R. Thus the arena-with-gap predicate will survive. We don't have to worry about the stability of the block itself, because being local it's immune to interference. By being outside the brackets, it can be freely mutated; it doesn't play a part in the stability argument. But not all local state can be treated so flippantly, indeed the token is crucial to the stability argument. The brackets thus delimit which part of the assertion can be safely touched and which mustn't.

The crux of the proof

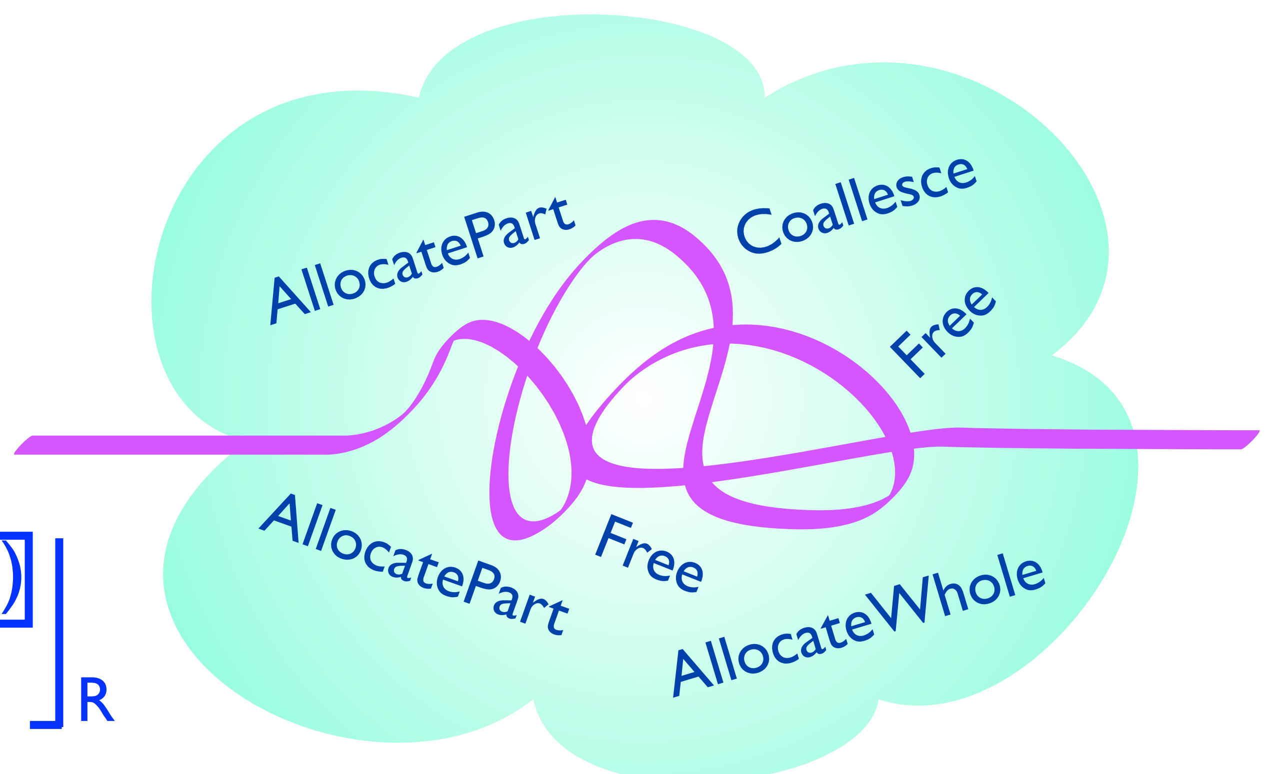


So how can the arena-with-gap predicate reach the call to free? Accompany it with the token, and package them together in stability brackets.

The crux of the proof

x := malloc(n)

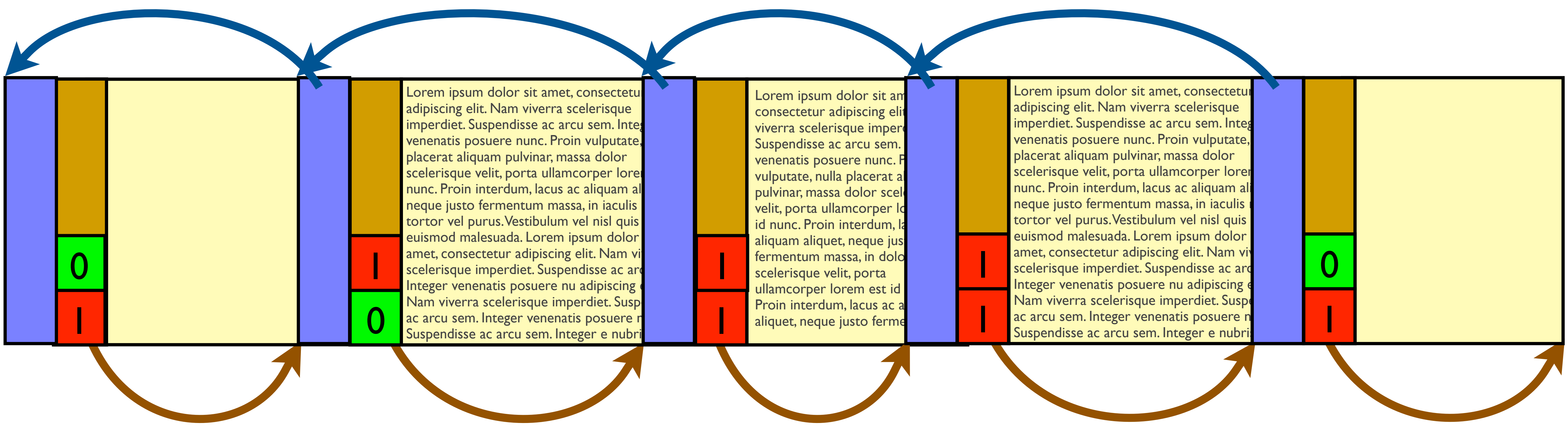
$\left\lfloor \boxed{\text{arena-with-gap}(x,n)} * \text{token}(x,n) \right\rfloor_R$



free(x)

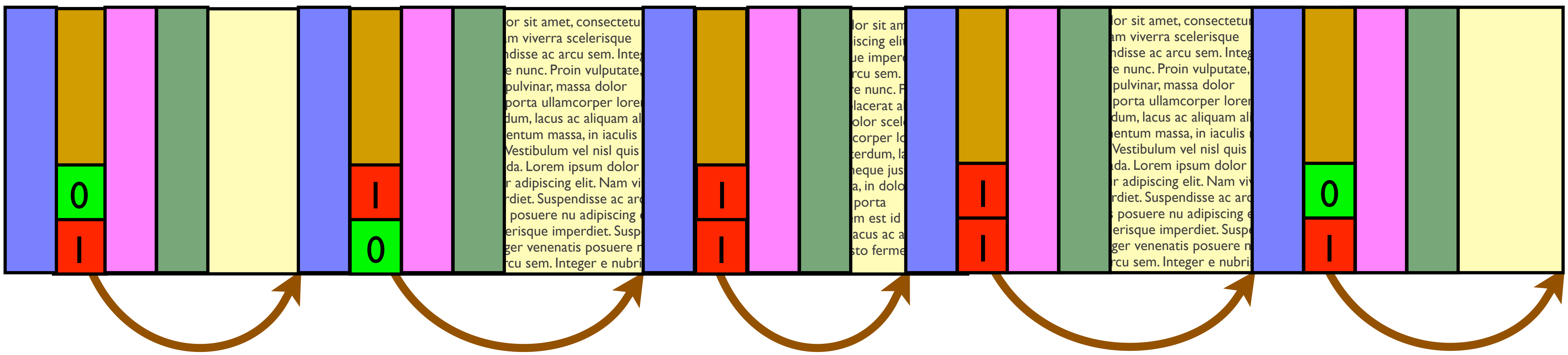
So how can the arena-with-gap predicate reach the call to free? Accompany it with the token, and package them together in stability brackets.

Doug Lea's malloc



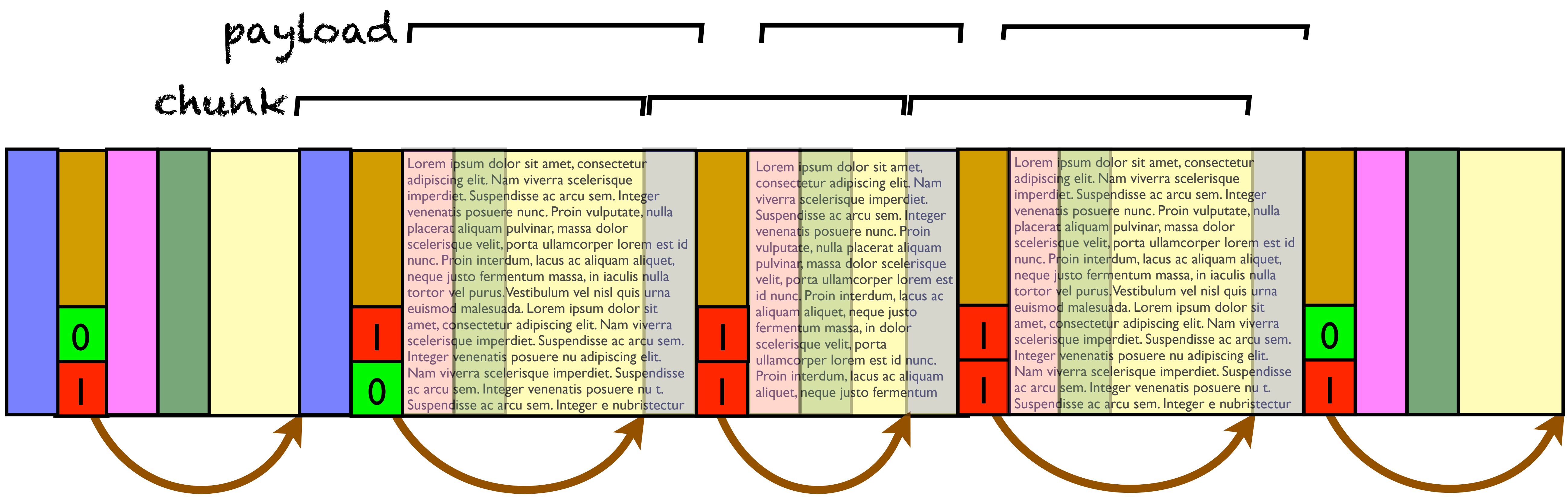
Arena is now doubly-linked. Means that when a block becomes free, we can coallesce with a free block to our left and to our right.

Doug Lea’s malloc



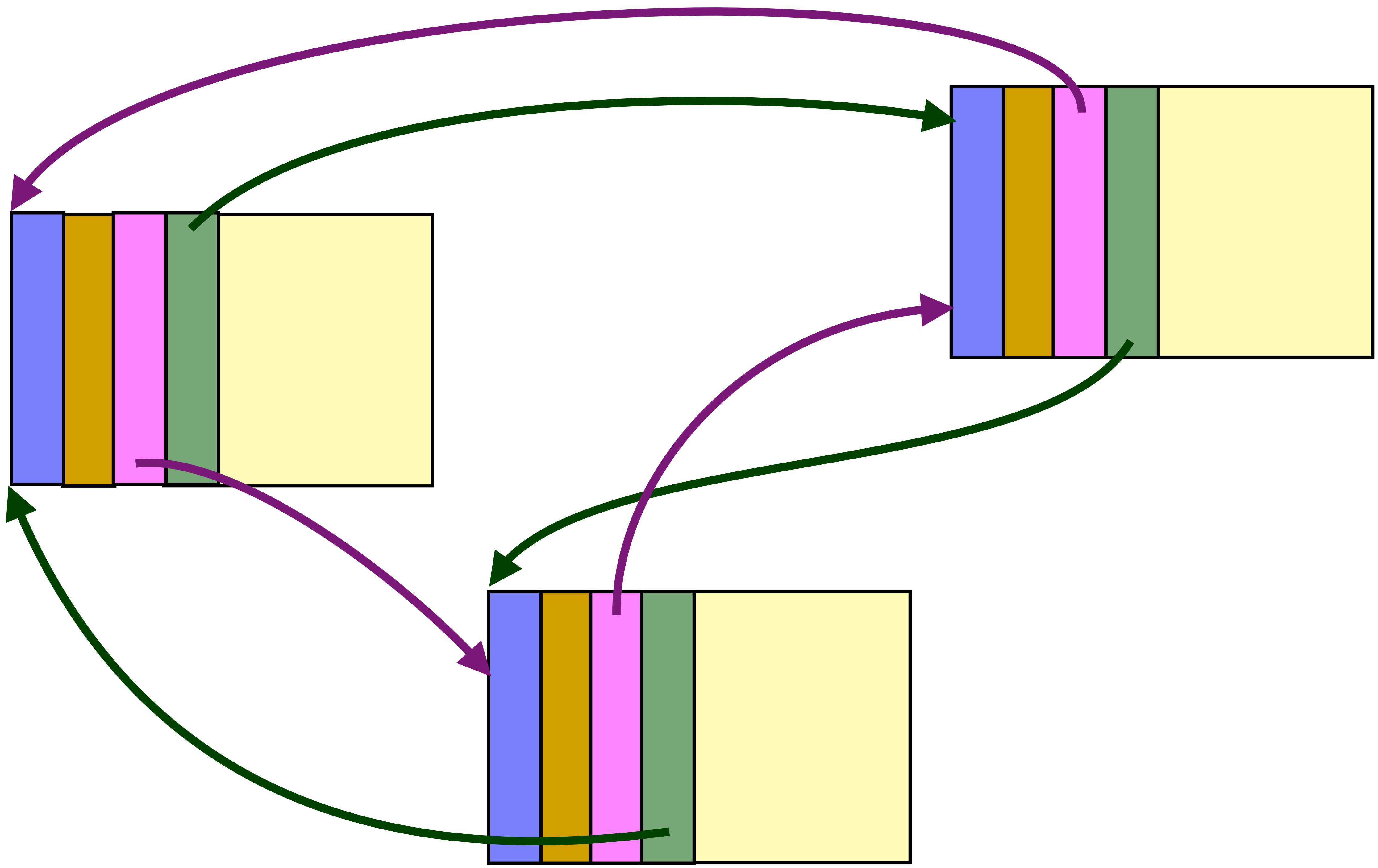
Blocks are also indexed by size (so no linear searching any more!)

Doug Lea's malloc



But to lower the overhead, we let the payload of chunks overwrite some of the fields – even those of the next chunk! The fd and bk fields can be sacrificed, because we are only interested in searching for *free* chunks of the right size. And we only need to follow the prev_foot pointer if PINUSE is not set.

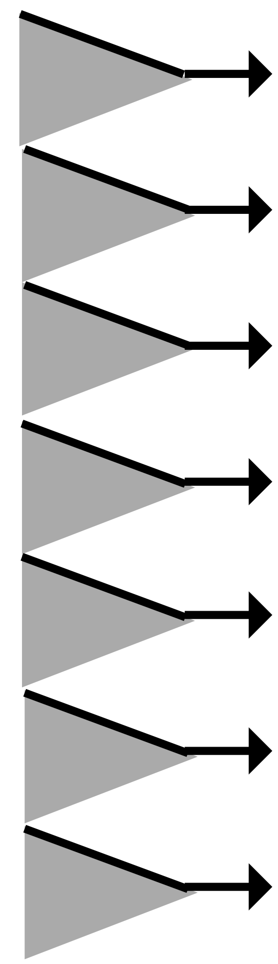
Smallbins



We have an array of 32 smallbins, which are circular doubly-linked lists of free blocks of exactly the same size. Element i has blocks of size 8i bytes (block size is always a multiple of 8 bytes).

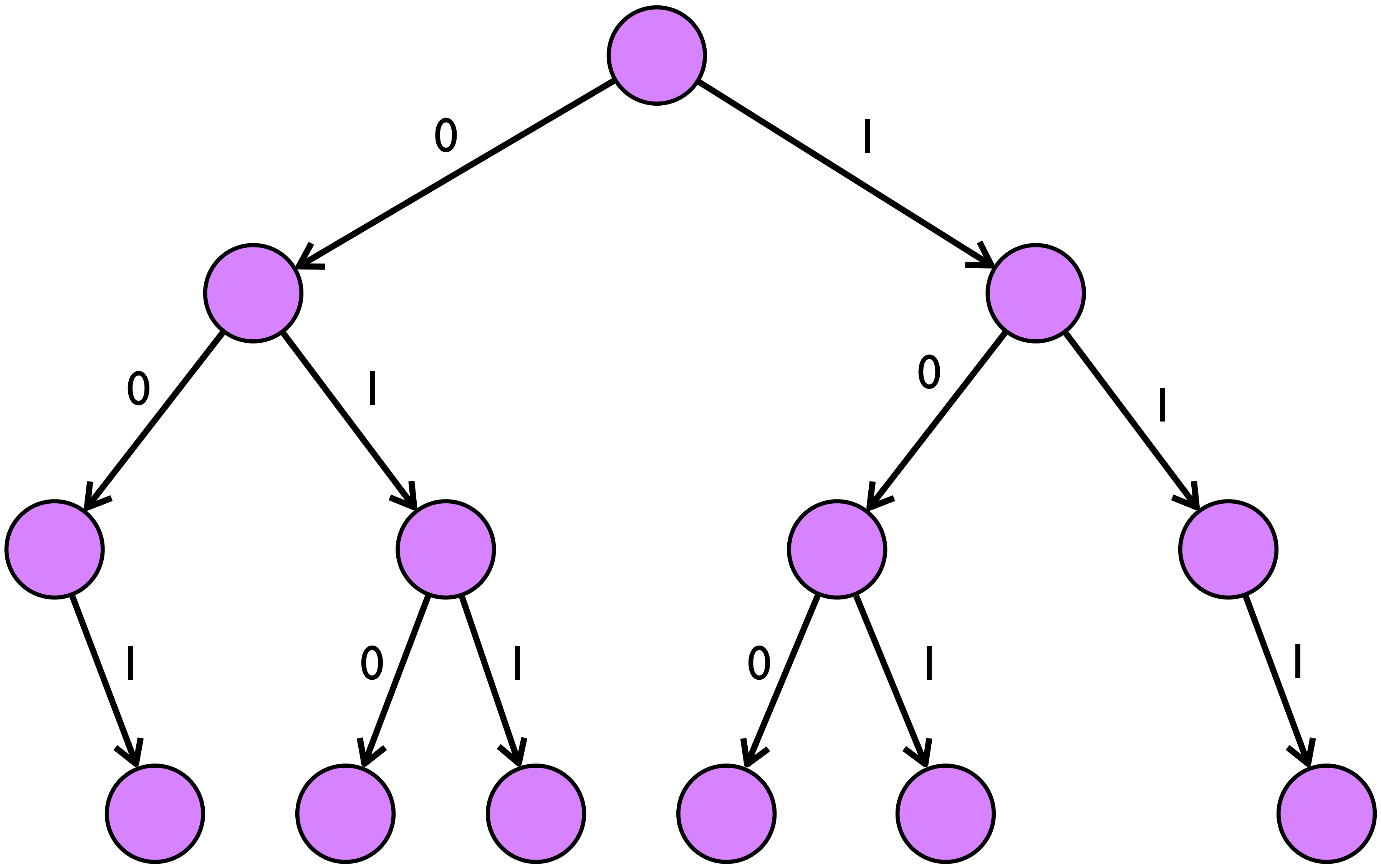
Treebins

		Size (bytes)	Index
100000000	=	256	0
110000000	=	384	1
1000000000	=	512	2
1100000000	=	768	3
10000000000	=	1024	4
11000000000	=	1536	5
100000000000	=	2048	6
110000000000	=	3072	
...		...	...



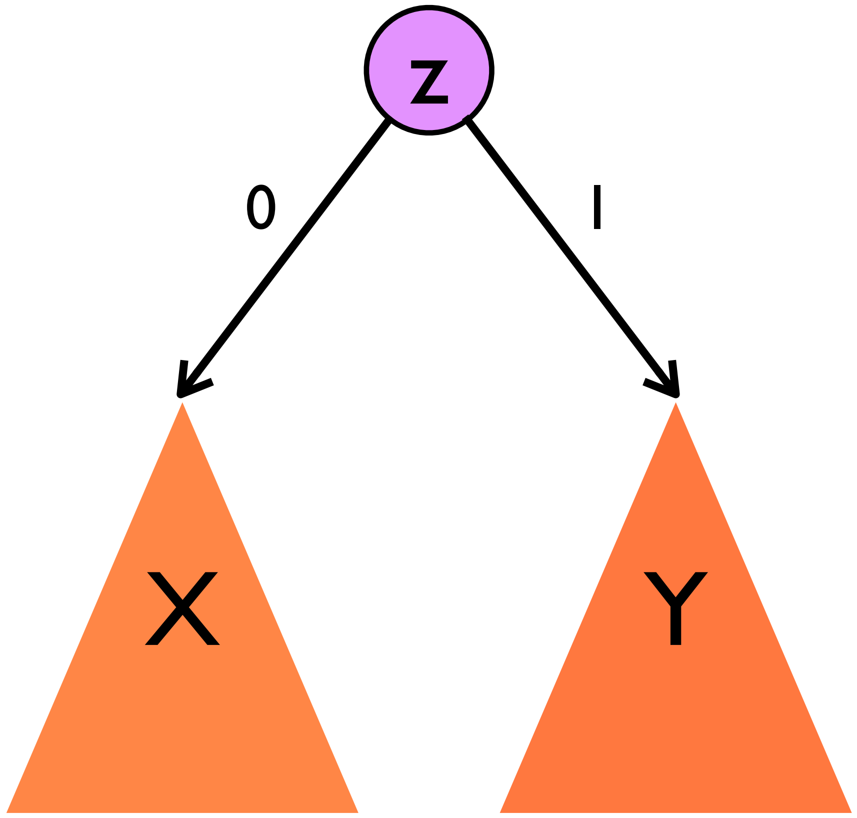
Larger free blocks are put into treebins. Unlike smallbins, treebins store a range of bin sizes, approximately logarithmically-spaced, with two bins per power of 2.

Treebins



Within the treebin, nodes are in a trie structure. Each node is a smallbin, containing all the blocks of that exact size. So every node holds a unique size. When a node becomes empty, a leaf node is moved up to fill the gap. So we never have empty nodes. This means that we can store the entire structure of the trie *within* the payload of the chunks.

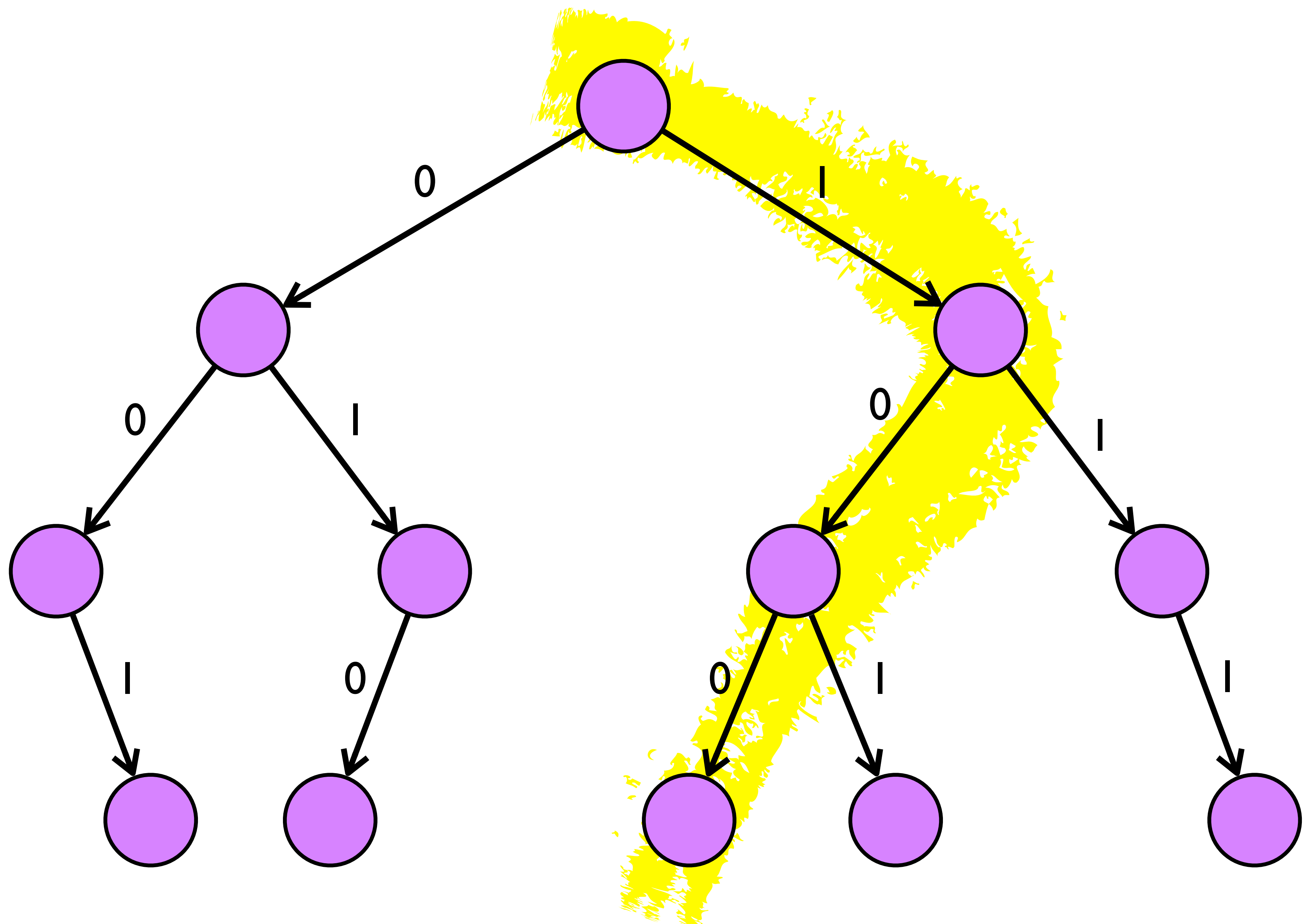
Treebins



$\forall x \in X, y \in Y.\ x < y$

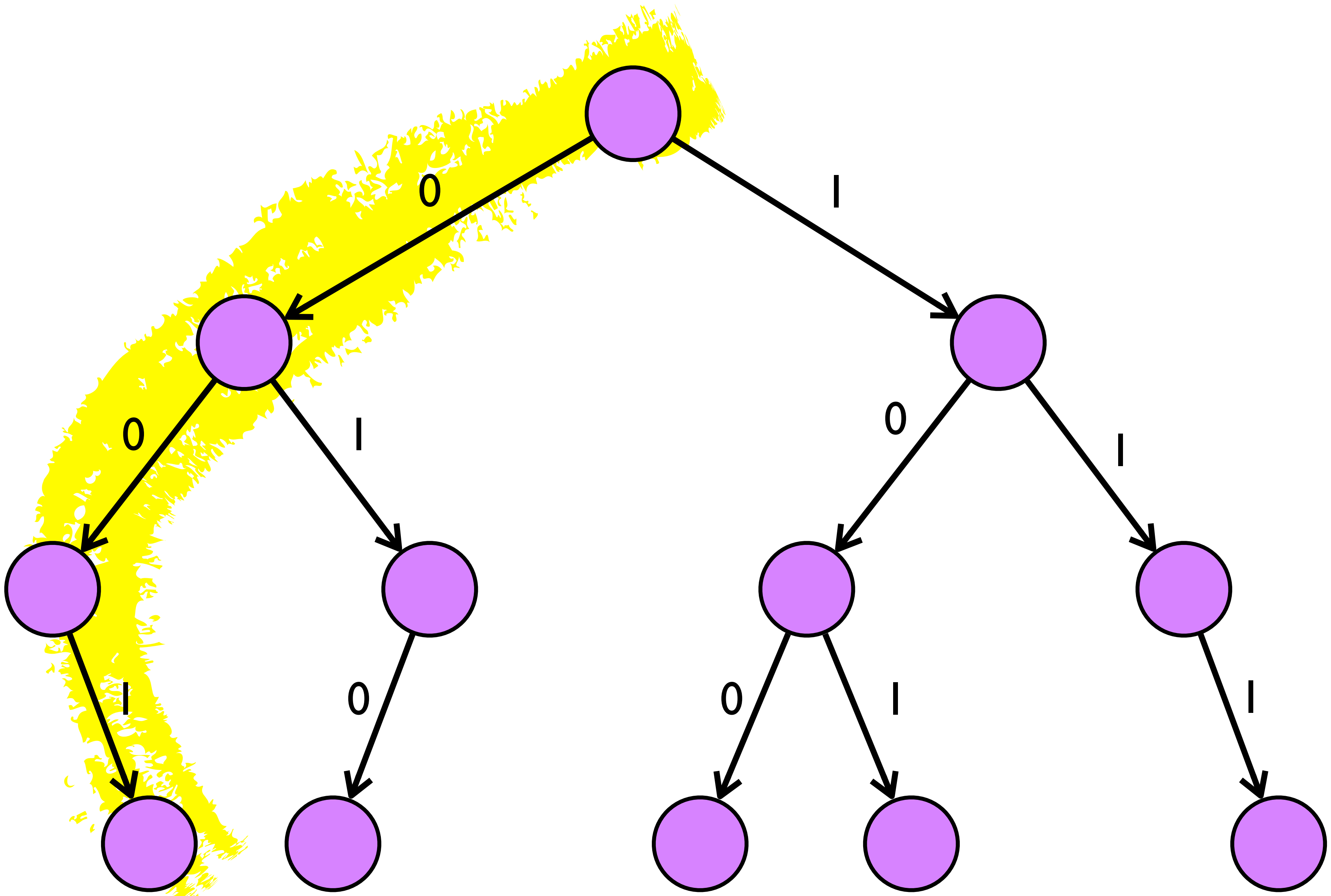
Every left subtree has sizes less than the right subtree, but neither is related to the parent.

Treebins



So to find the chunk of size 100, it will be *somewhere* along the path 1-0-0.

Treebins



To find the smallest chunk in a tree follow the left-most path (going right when necessary). The smallest chunk will be somewhere along that path.

Overlaid structures

```
struct chunk {
  size_t prev_foot;
  size_t head;
  struct chunk* fd;
  struct chunk* bk;
  struct chunk* child[2];
  struct chunk* parent;
  unsigned int index;
}
```

All but "head" are part of the payload. The prev_foot and head fields locate the chunk in the arena. The fd and bk pointers locate the chunk in its smallbin, the child and parent pointers locate it in its treebin (if the chunk is large), and the index identifies which treebin it is in. Note that the overhead is just 4 bytes per chunk (wow!).

So we have lots of overlaid data structures, which means we don't have the natural notion of "separation" that we're used to.