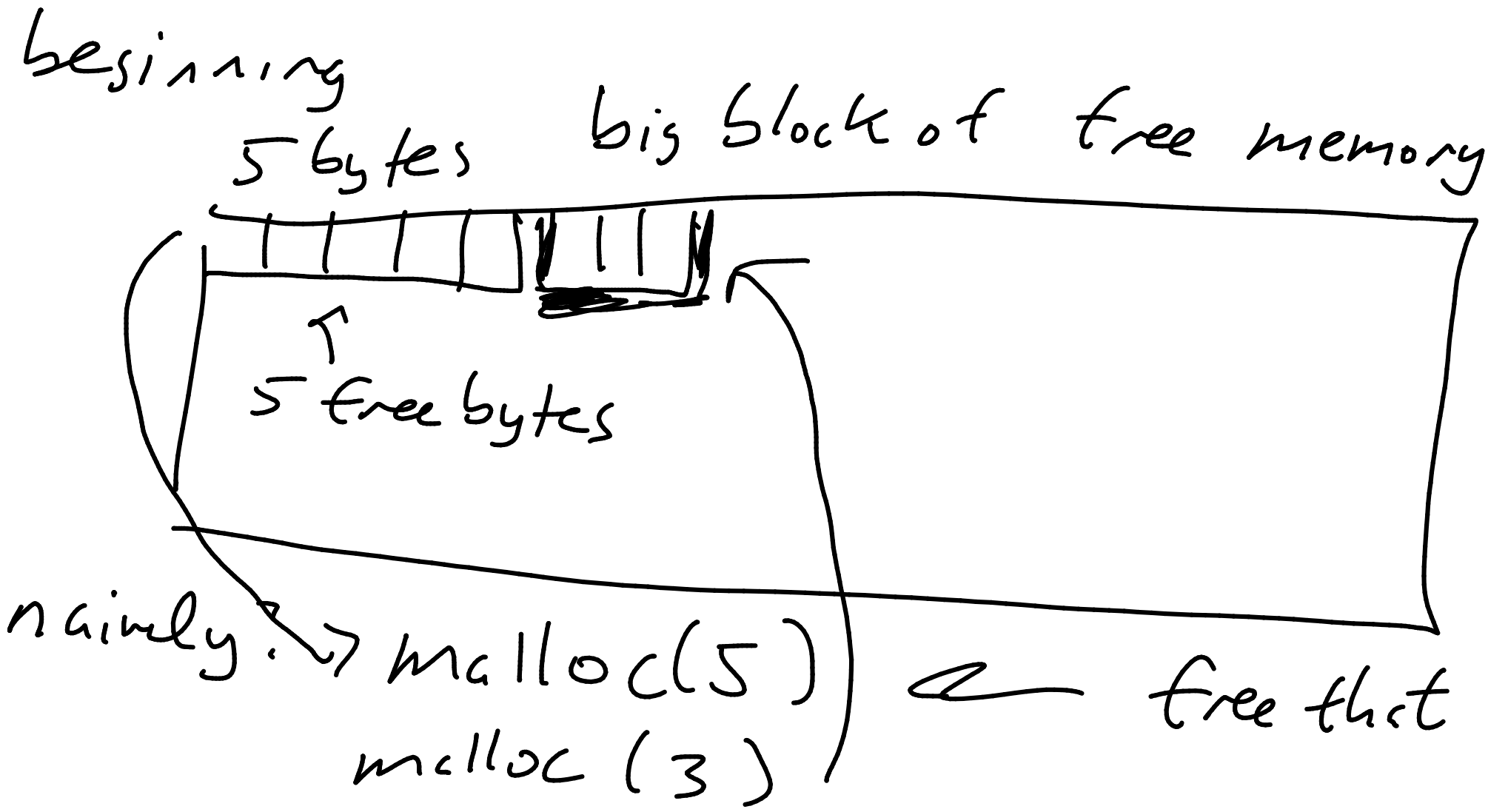
Reminders: today - declare if potatically
self-scheduling

Worksheet on memory allocation
- 5-min-ish
- wl groups review it and remember your initial answers

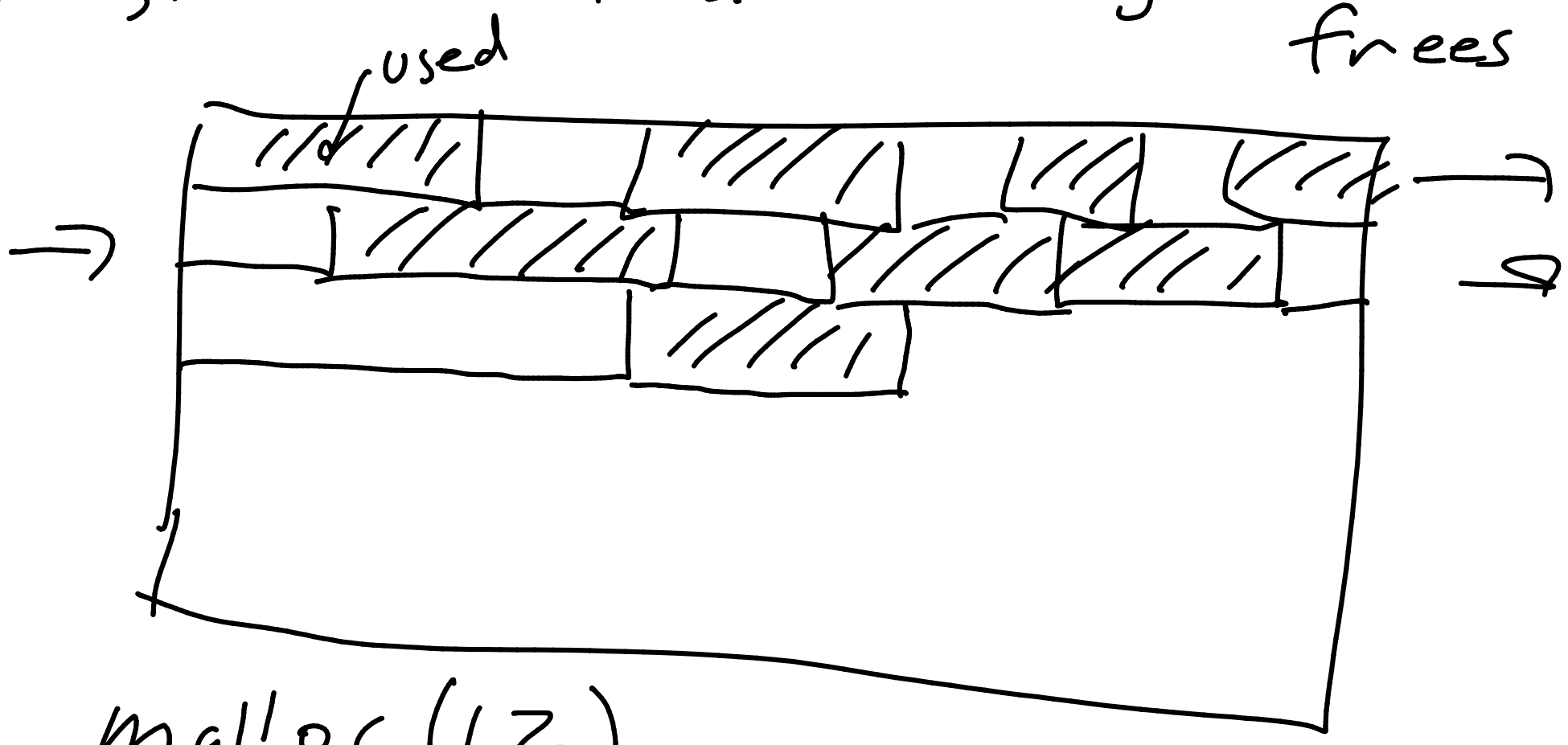
How does malloc / memory allocation in general work? How to do it efficiatly



malloc(7) - search through all empty regions to see if big enough - slow if not smart

(binary search is a no go because data is not sorted)

Fragmentation: after many mallocs and frees



malloc(12)

① do I pick the first spot I find that's at least 12 bytes?

OR

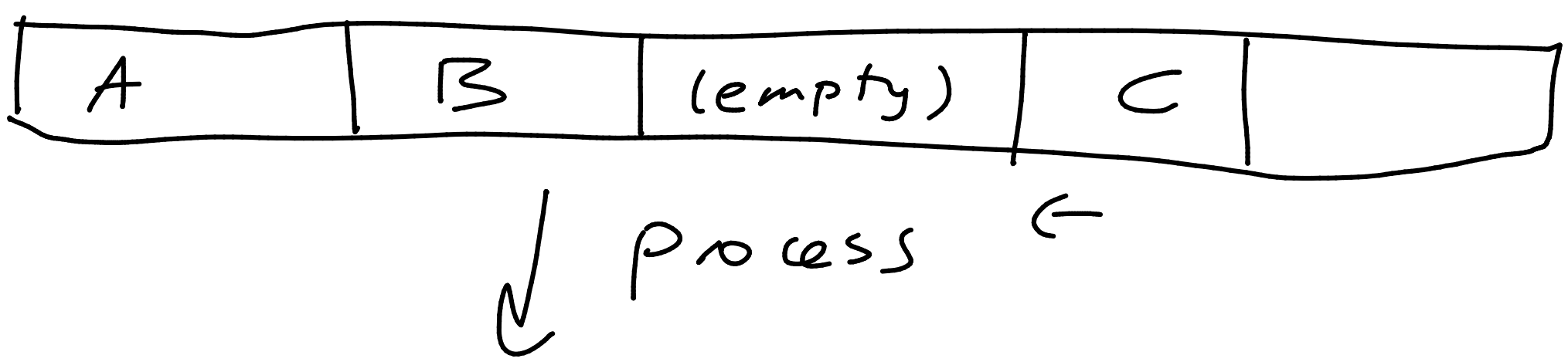
② do I pick the smallest one I can find that's at least 12

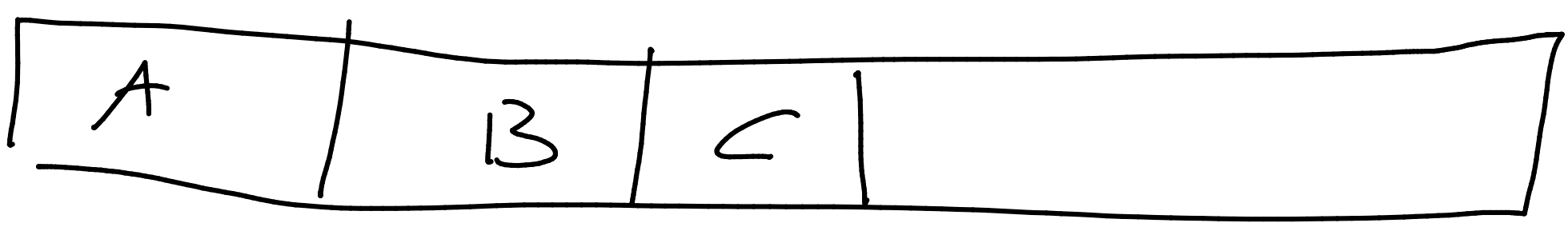
① "first fit" - lots of memory left behind, which someone else could use later, but over time these really small and separated

② "best fit" - slow (if you're not smart) because need to search everything

---

Can we defrag memory?





changes pointer values?

is that ok?

in C, for example

```
int *x = malloc(5);
printf("%p\n", x);
printf("%p\n", x);
printf("%i\n", *x);
```

↓ different values! (if defrag'ed in between)

when you defrag, you have to change all the variable values

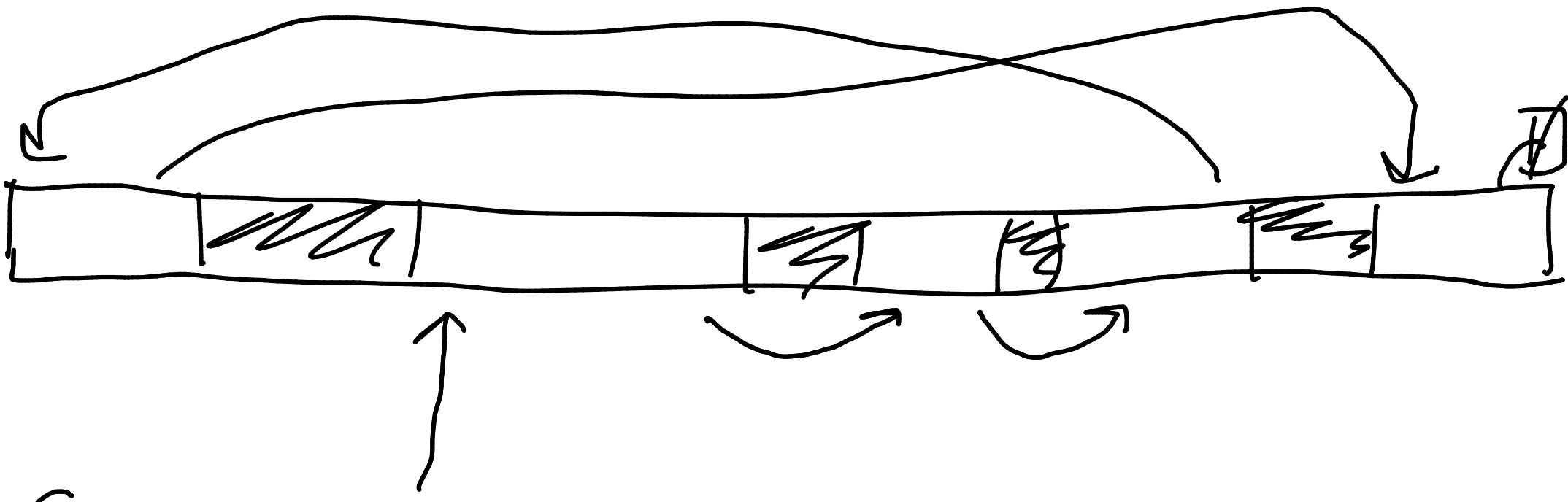
-- double --- messy ---

This is not double/allocable in any language that exposes ptrs in any meaningful way, like C does

---

If you can't defrag memory how can you find / free / recombine efficiently?

Lots of approaches!
If you did nothing: basic idea is you keep a list (a linked list) of free memory.



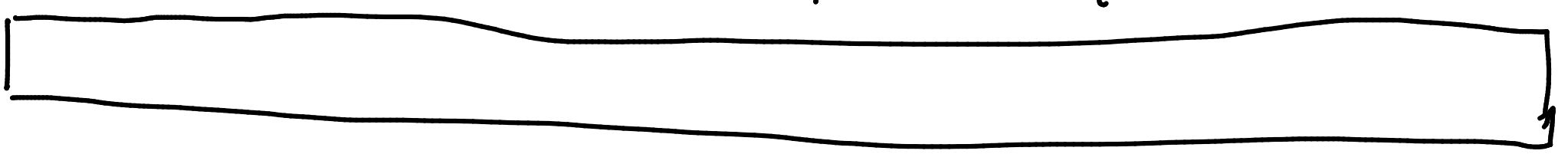
free memory
This is still just one linked list, which is slow to search for best fit.

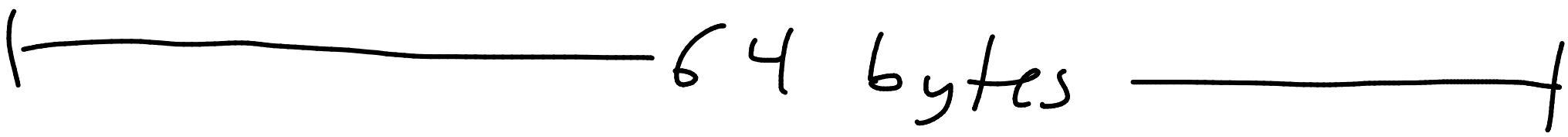
Instead use lots of lists, that represent various sizes of free memory

"segregated lists"

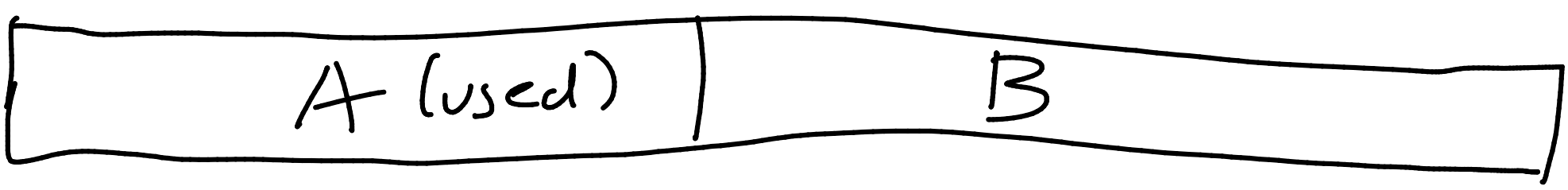
Buddy system

- only break up memory in sizes of Powes of 2





malloc(32)

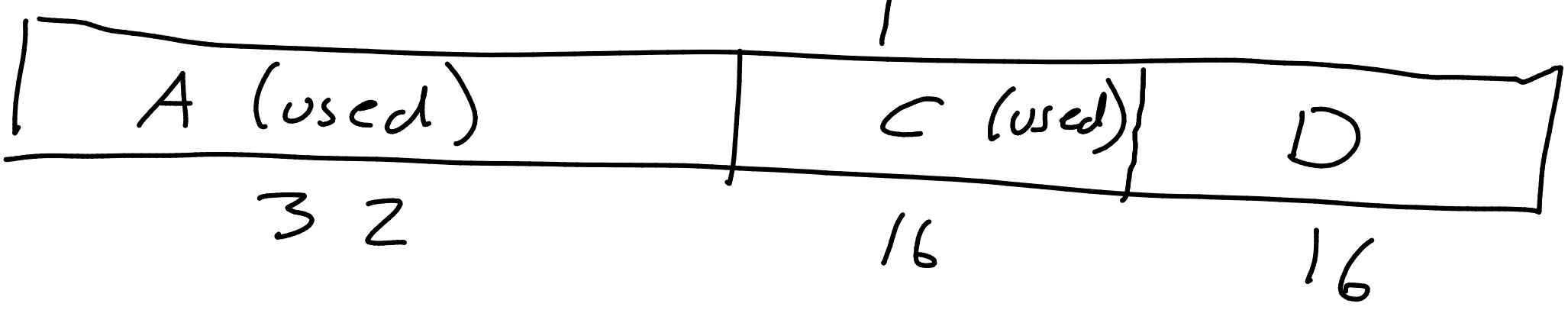


free lists

32: B

malloc(15) ← returned

Keep slicing memory until I have a piece of size that's a power of 2, but as small as possible



free lists

32: ——

16: D

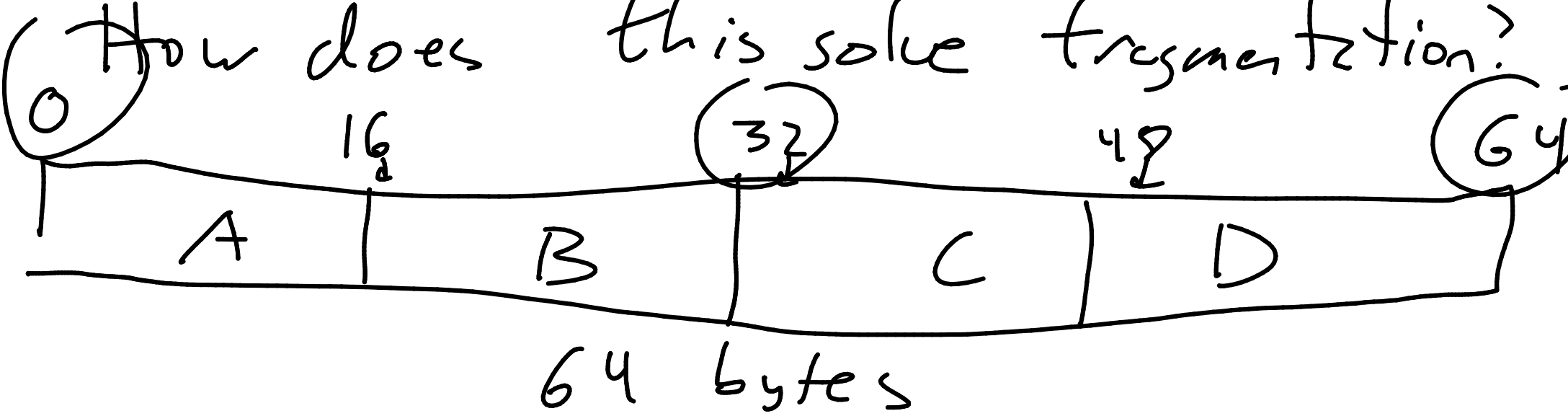
free(A) → A is unused, goes back on the lists

lists:

32: A

16: D

How does this solve fragmentation?



malloc(16) → A

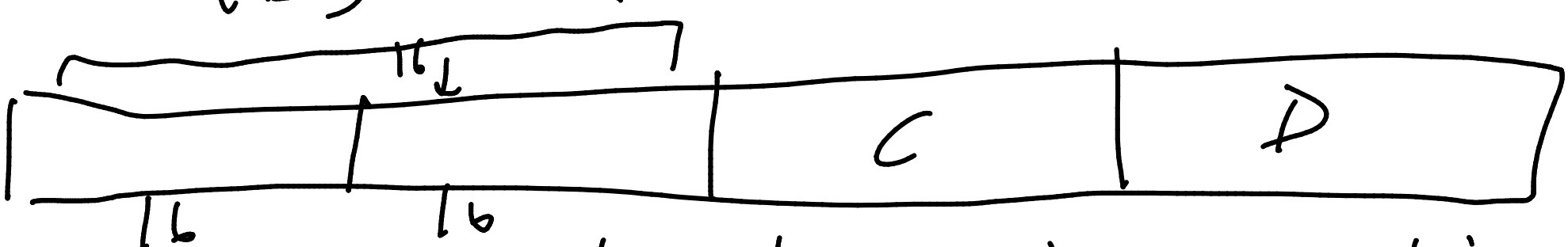
malloc(16) → B

malloc(16) → C

malloc(16) → D

free(A)

free(B)



How does malloc know it can combine old A and B back together again?

Whenever anything is freed, see if its old buddy (the one is split from is also free, and if so, recombine)

When I free B, how do I know if its old buddy was to its left or its right?

B starts at 16

Chunks of size 32 must start at memory locs that are multiples of size 32

When I free B, look at mem address: if multiple of 32, it is a "left buddy", otherwise, it is a "right buddy"

But if memory you want to malloc is not a pow of 2, end up wasting memory

---

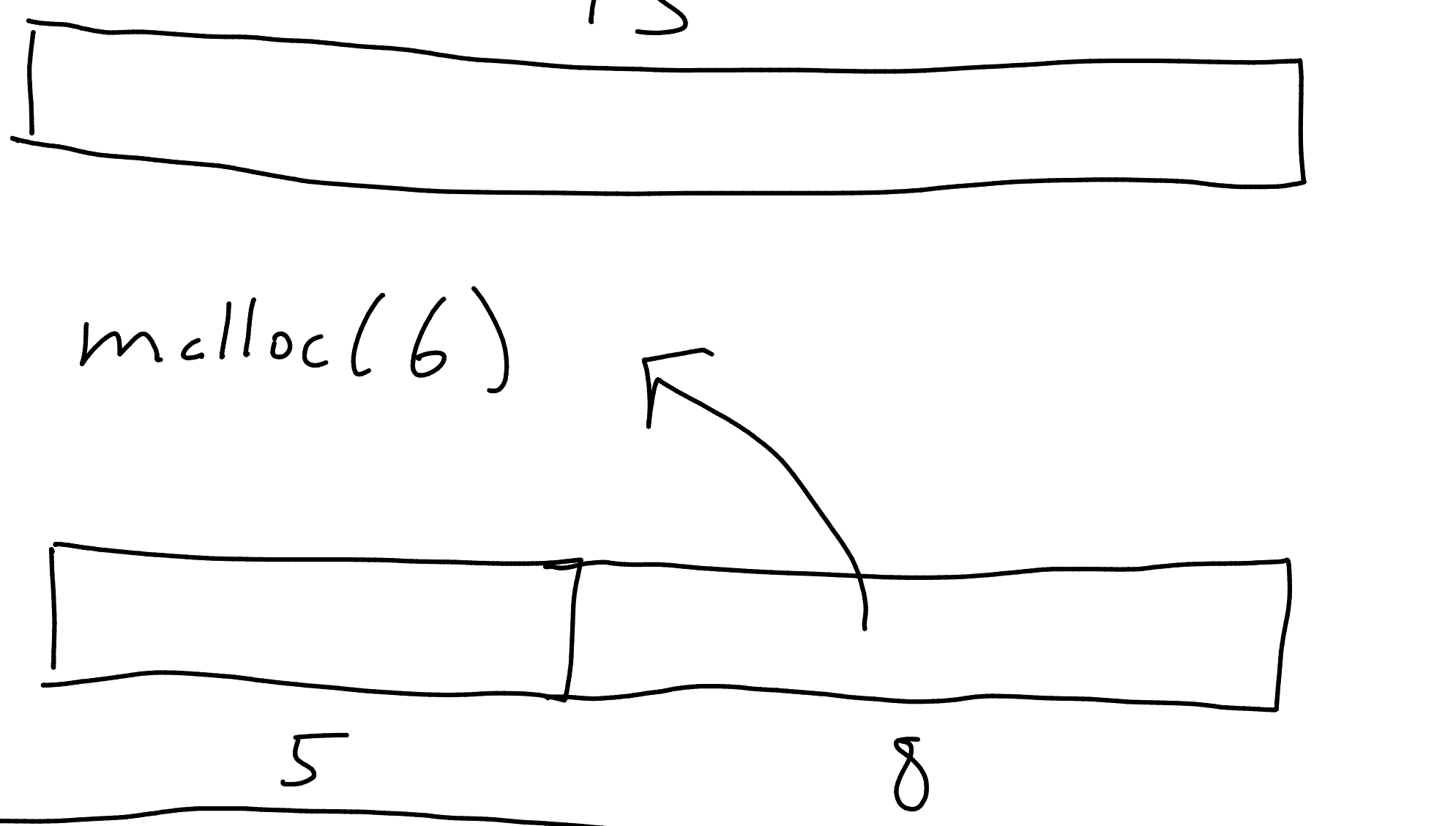
Can we improve?

- Fibonacci numbers can help!
- seq where each number is the sum of previous two

1, 1, 2, 3, 5, 8, 13, ...

grows more slowly than pows of 2

1 2 4 8 16 32 64



Garbage collection

How could we have improved talloc?

- talloc remembers all of it, but never lets it go until you call tfree.

A good garbage collector would free memory not in use anymore

Approach: reference counting
(easy)

Every time an object gets pointed to,
increase a count
When you move a ptr away,
decrease count
When count is 0, free it

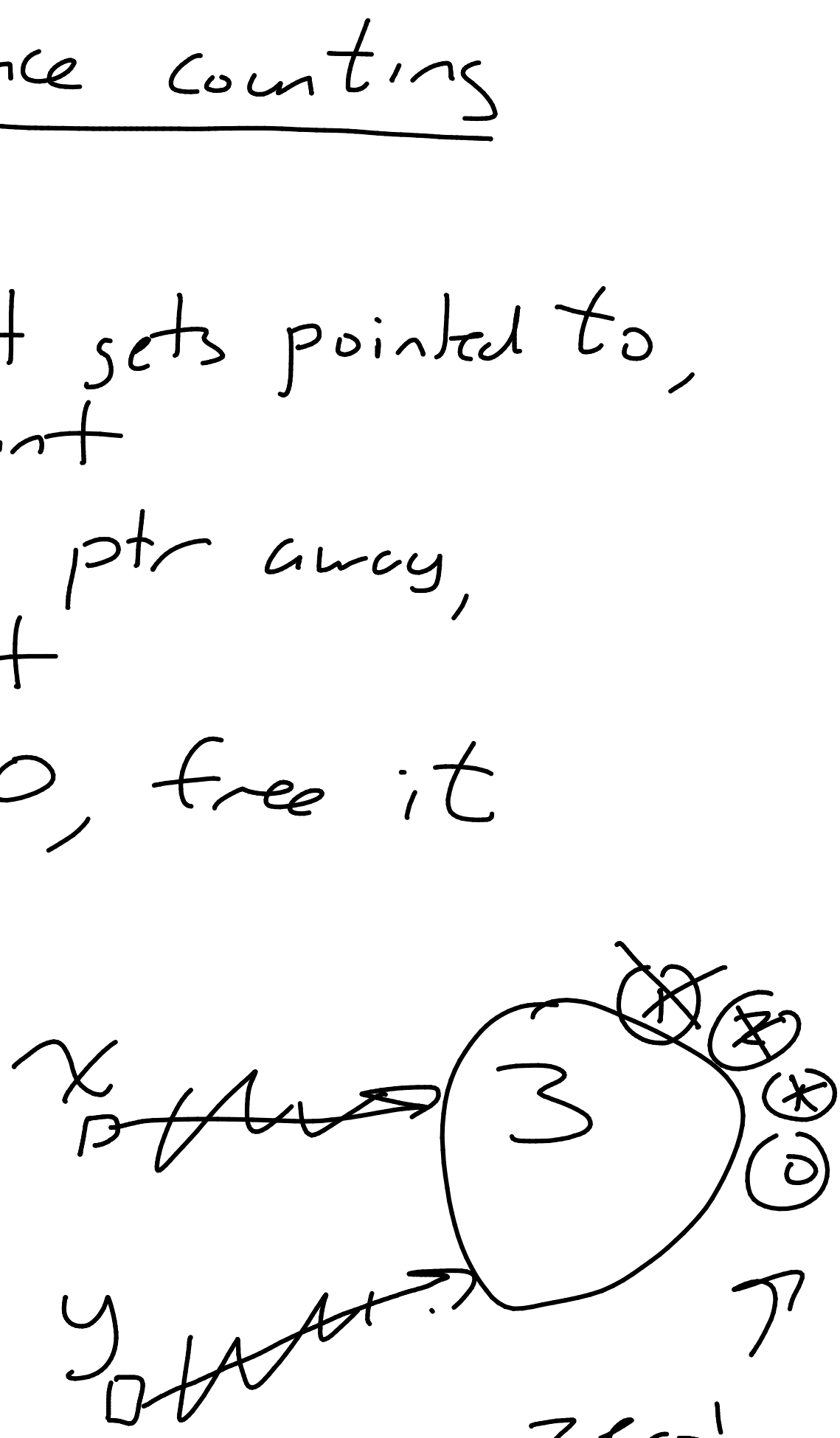
Python

$x = 3$

$y = x$

$y = None$

$x = None$



Advantages:
- easy to explain and to implement
- works incrementally
  - memory is freed right after you don't need it anymore

Amazing (except that it's broken)

pseudocode

Node

value

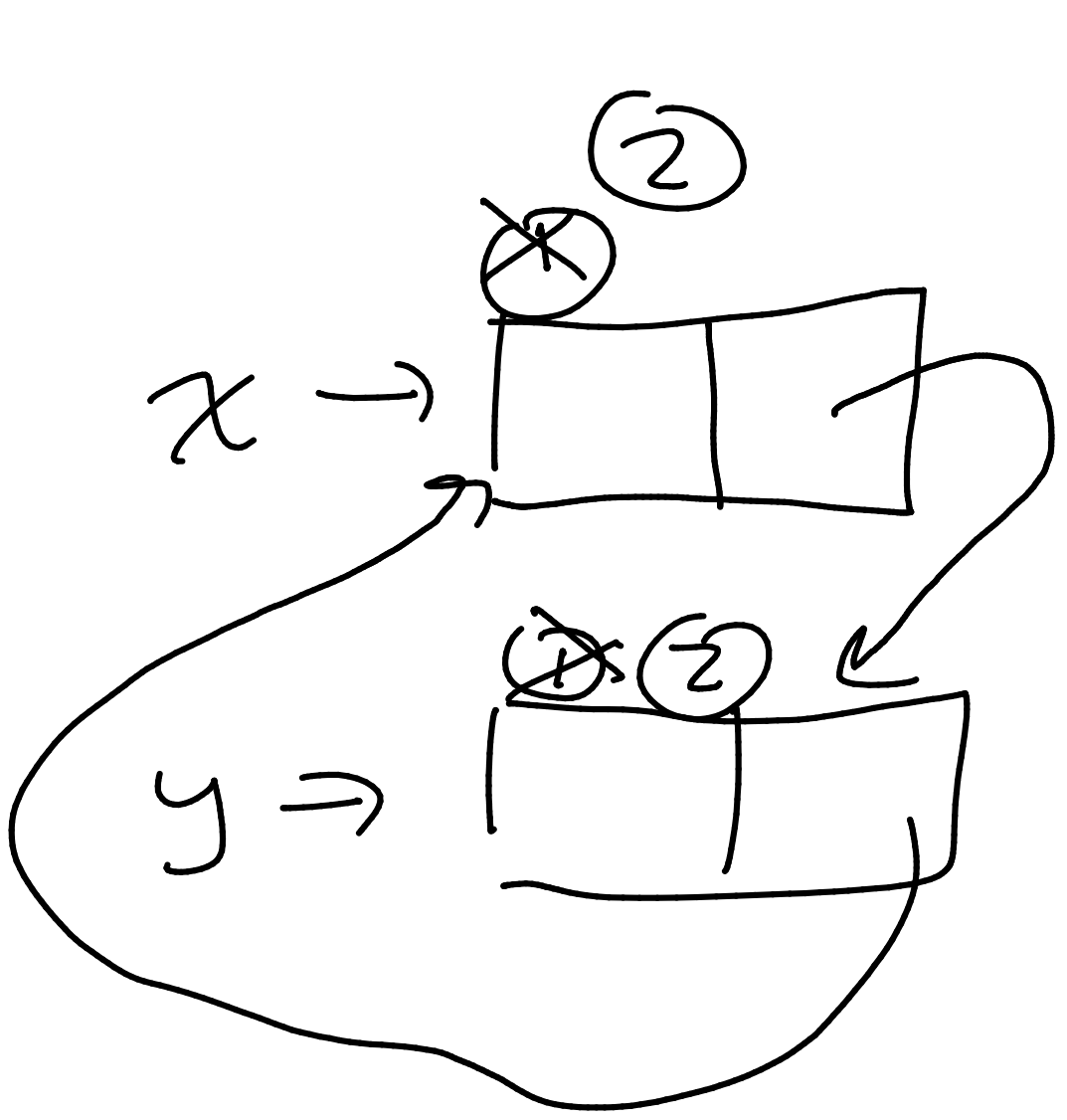
next

x = Node()

y = Node()

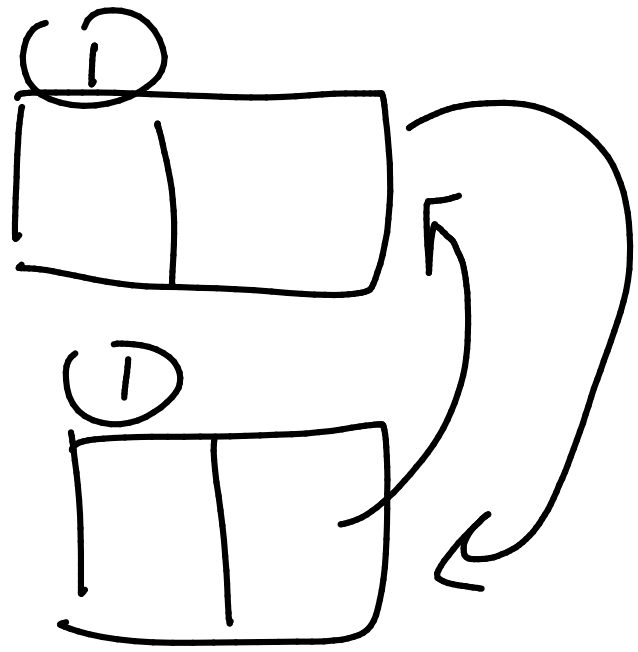
x.next = y

y.next = x



x = None

y = None



This fails if have circular references.