Today - how could talloc have been better?

- Talloc does garbage collection!

A gc is supposed to free up memory that's not being used anymore. (C doesn't have one)

↙ take time ↙ and memory

unavoidable, but maybe worth it

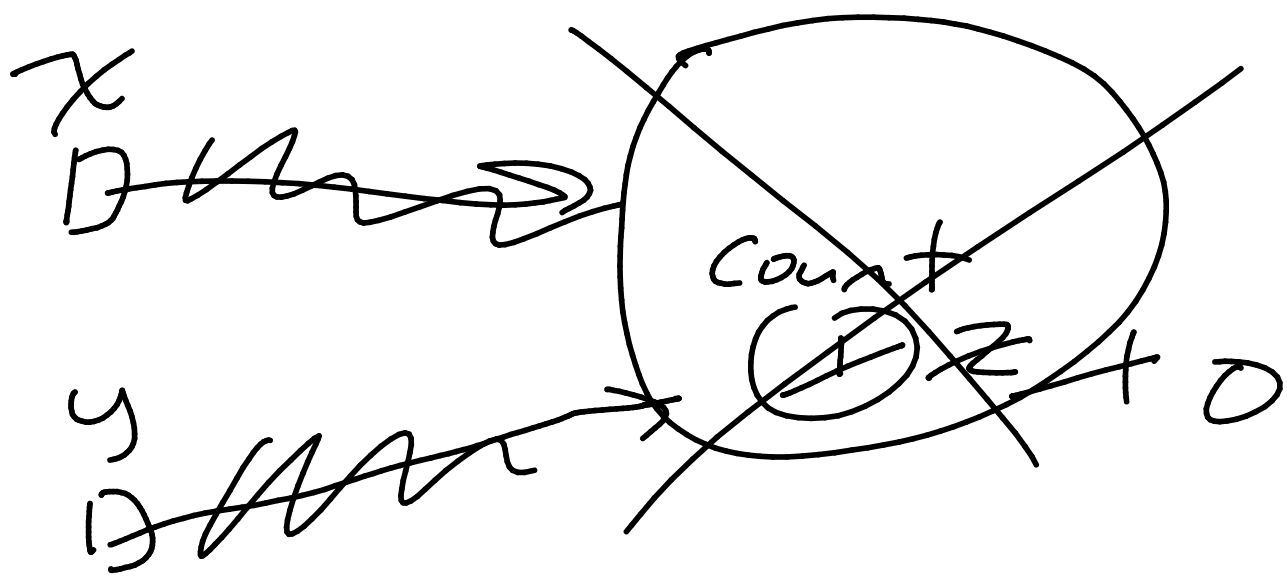
How is it done better?

① Straightforward, good in some ways, and somewhat common, but also doesn't quite work

Reference Counting

Every object [SchemeVal] has a count of the number of pointers referring to it.

When you assign a new one, increment
... move one away, decrement
when count is 0, you free it



y = x

y = null

x = null

What's good?

Easy

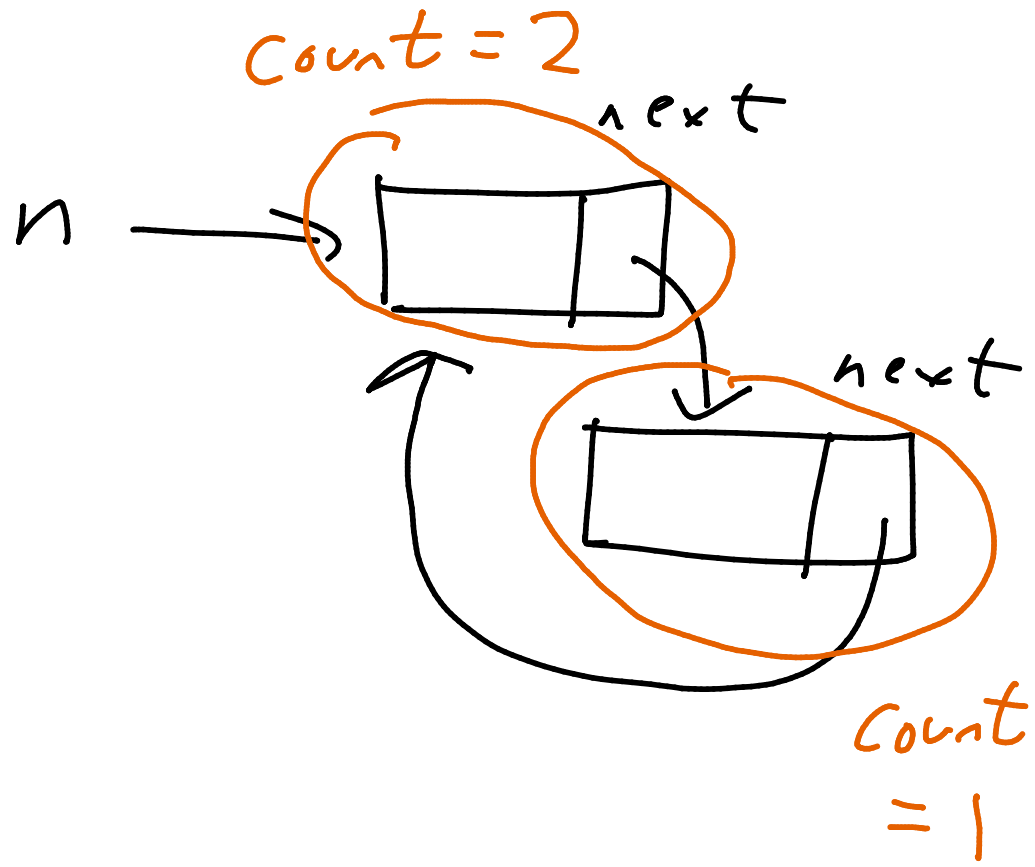
Incremental - frees memory as soon as it becomes free

Relatively fast

What's the problem?

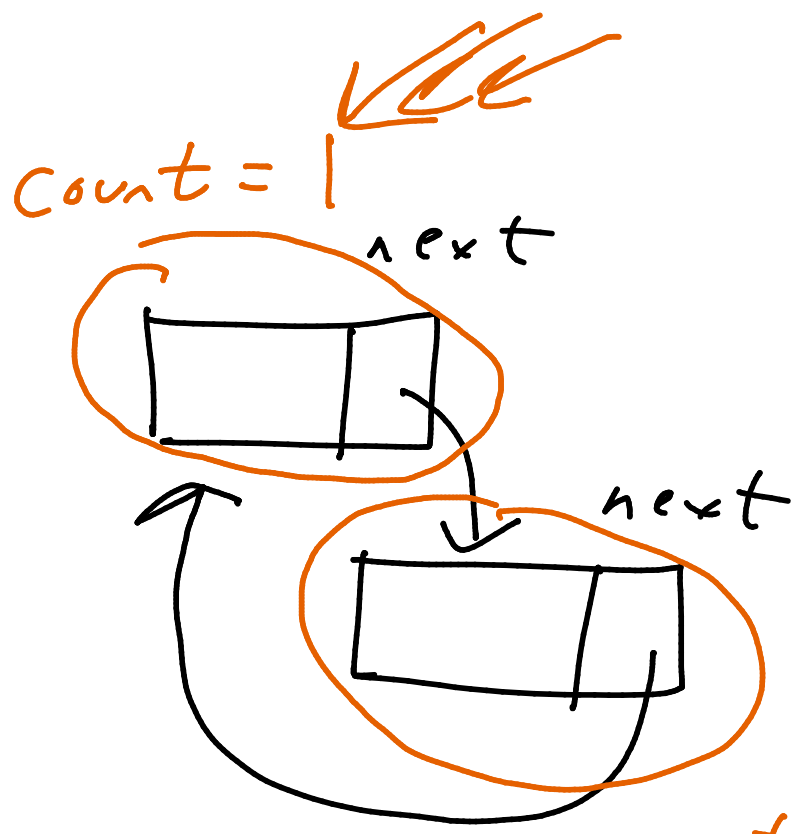
- circular references

```
n = Node()
n.next = Node()
n.next.next = n
```



n = null

Counts are 1, so nothing gets freed - but it's all garbage!



How do you handle this?

① You don't, ugh

② You fit some kind of ugly cycle detection algorithm on top of this

C Python (python.org)
does this. Ref counting +
occasional cycle detecting

---

② (actually does work)

Mark-and-sweep

- occurs "every so often"

follow all pointers that you have access to (in your function call stack, e.g. by all variables that are live)

- mark those objects as alive
- recursively, mark all the objects that they point to, and so on.

When all done marking, free all the unmarked objects.

Accuracy?

Time?

Memory needed?

Other?

Where did I oversimplify that would require effort / time / memory / etc?

---

* How do you find the decd objects?

When are you doing this?

Tradeoff of time spent vs free memory

Typically

- time intervals
- memory gets "close" to full

How do you find all the pointers?

- you need some kind of clear object defns and known pointer types within?

How do you find all current variables?

In our Scheme implementation, look in current frame, its parent, grandparent, etc.

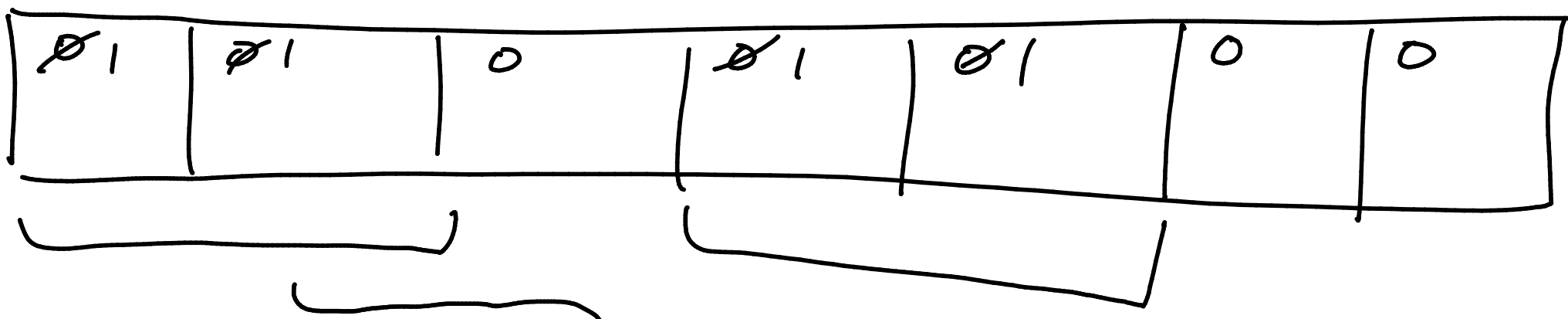
[follow everything recursively]

This does correctly handle circular references.

* How find dead objects?

The heap is broken up into small fixed-size units, which are combined to make bigger units if you need them for your objects.

- there's a space in each unit indicating if it was allocated when the object was constructed



get me a int

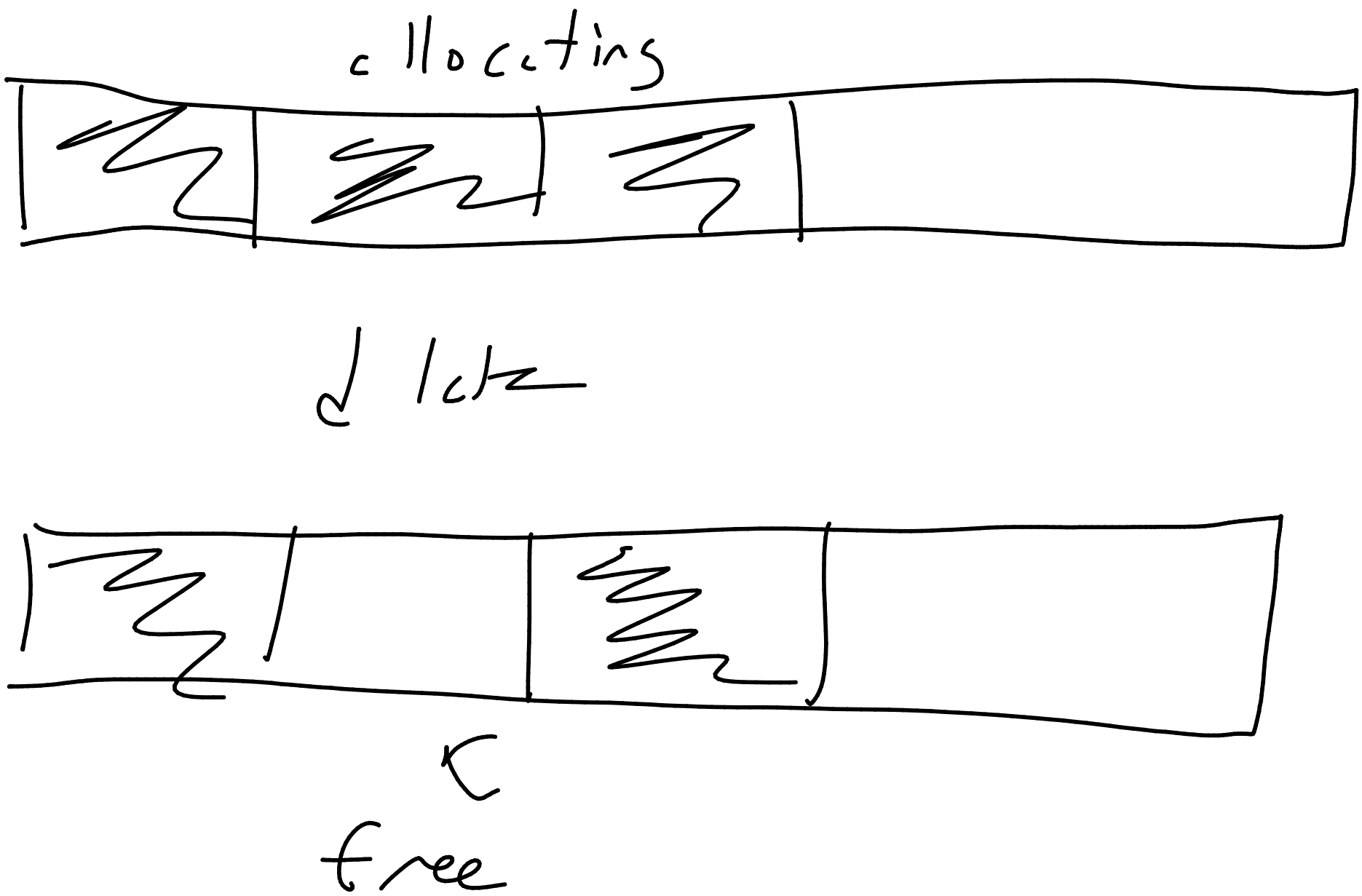
set me a double

this is not marking from gc, this is just an indication if it had ever been allocated

---

To find dead objects, loop over entire heap. If it had been allocated at some point and is not marked, free it.

One more big issue is that this approach leaves memory fragmated

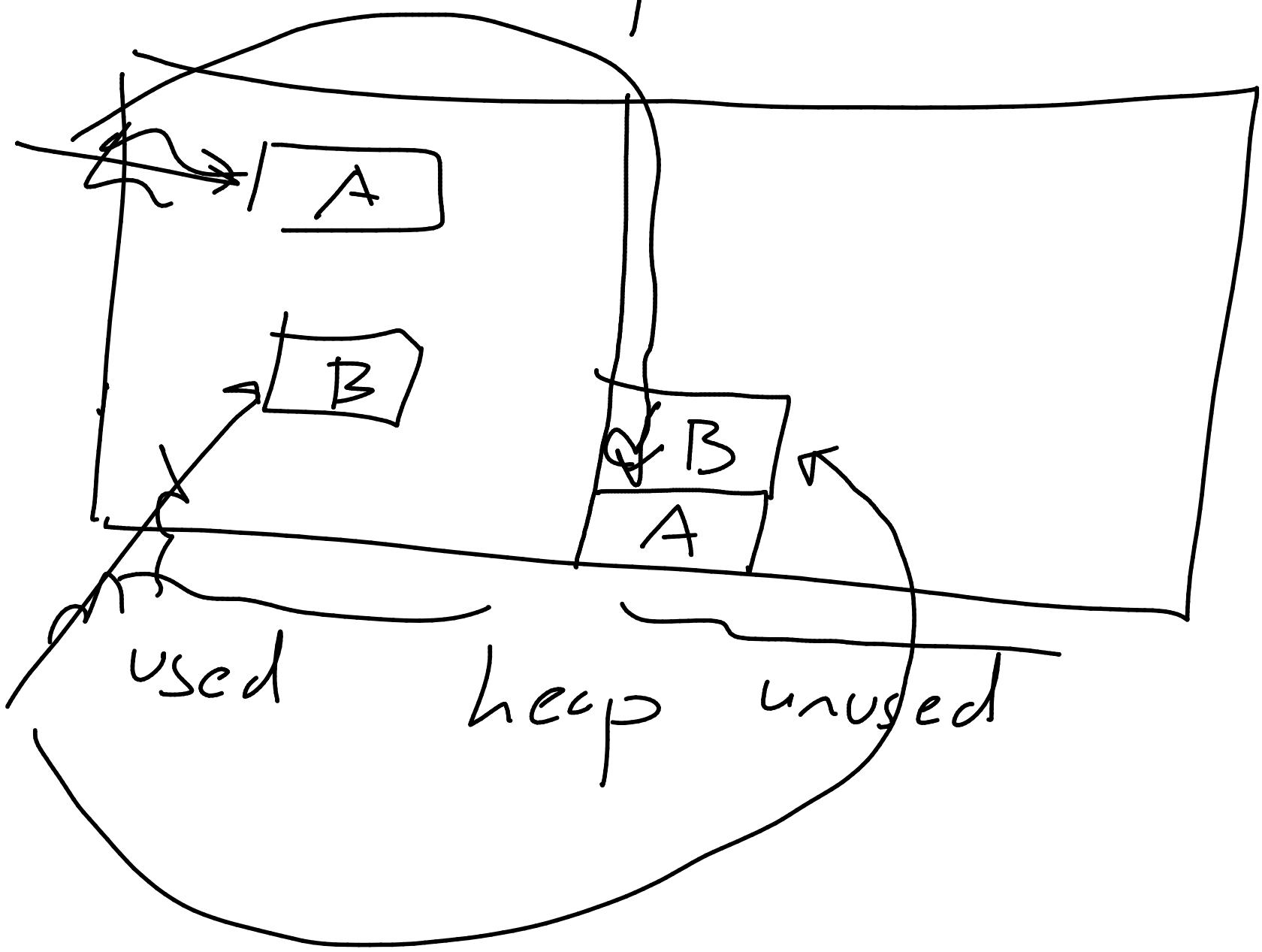


Our free memory is now broken into smaller chunks (fragmented)
- limits the size of future objects
- and makes it slower to find memory big enough for your object

[would be just as bad if we were freeing manually]

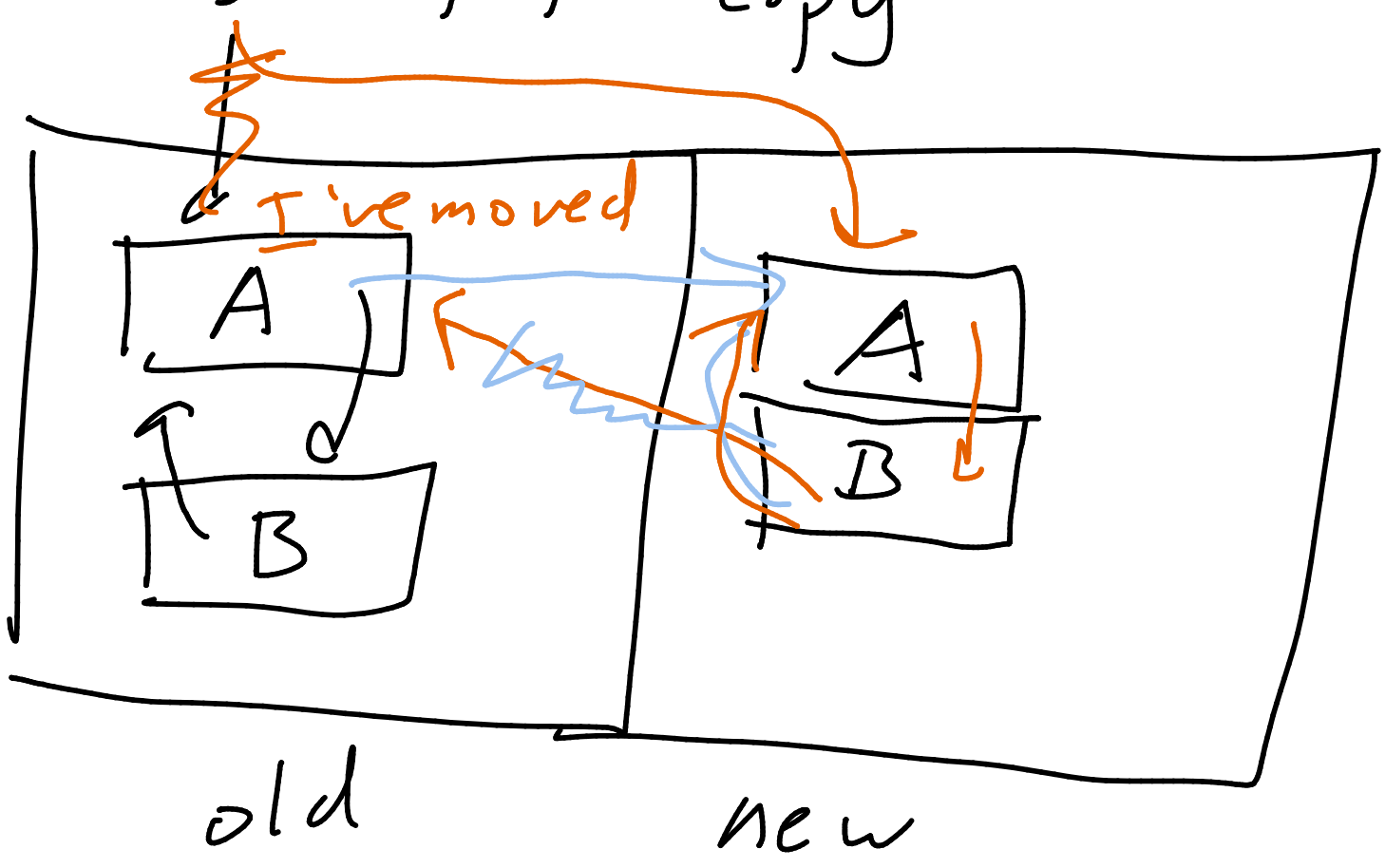
(3) Copying (stop-and-copy)

- similar to mark and sweep
- difference: when you get to a live object, you copy it to an alternative heap



- copy each object to the unused heap, reset ptrs, and when done, the old heap is now the unused one.

Whenever you copy an object from the old heap to the new, you write in the old location a "forwarding address" so that any other objects pointing there can update their ptrs if they try to copy



This works great, it results in defragmented memory, and uses <u>twice as much</u> memory to pull off.

When done, old heap is just completely done until you reverse roles next time

Takes a little more than mark/sweep because it has to copy. (??)
But it doesn't have to free individual chunks like mark/sweep does, so takes less?

One hybrid approach that is common (Java VM does this) is <u>generational</u> gc

studies have shown that young objects are most likely to die

↳ Use one gc technique more frequently only on young objects