<Heap Allocation>

7. static vs. Local variables.

local variable { stored on the call stack. (push & pop)
lifetime: function: called → return
params, return address, local vars are pushed by compiler

static variable { stored at a fixed memory location (heap)
assigned at compile time by linker (loc is known)
encoded in the executable
lifetime: ('static) the complete runtime of the program
can be referenced from local variables
cons: read-only by default → should be encap in a Mutex type
(a single &mut at any point)
⇒ avoid data race

2. Dynamic Memory: Heap

- heap supports dynamic memory allocation at runtime through [allocate] & [deallocate].

common errors { ① memory leak: forget to deallocate: (excessive mem consumption)
② use-after-free: can be exploited to exe arbitrary code
③ double-free: can lead to use-after-free.

⇓

Garbage Collection: the program is regularly paused & scanned for (Java, Python ...) unused heap variables ⇒ auto deallocate

↕ (assign an abstract lifetime to each reference)

Rust: Ownership - check the correctness of dynamic memory operations at compile time

{ alloc: Box::new(...) ⇒ no garbage collection at runtime & fine-grained control
dealloc: Drop trait (goes out of range) ⇒ no performance overhead.

↑ ↓

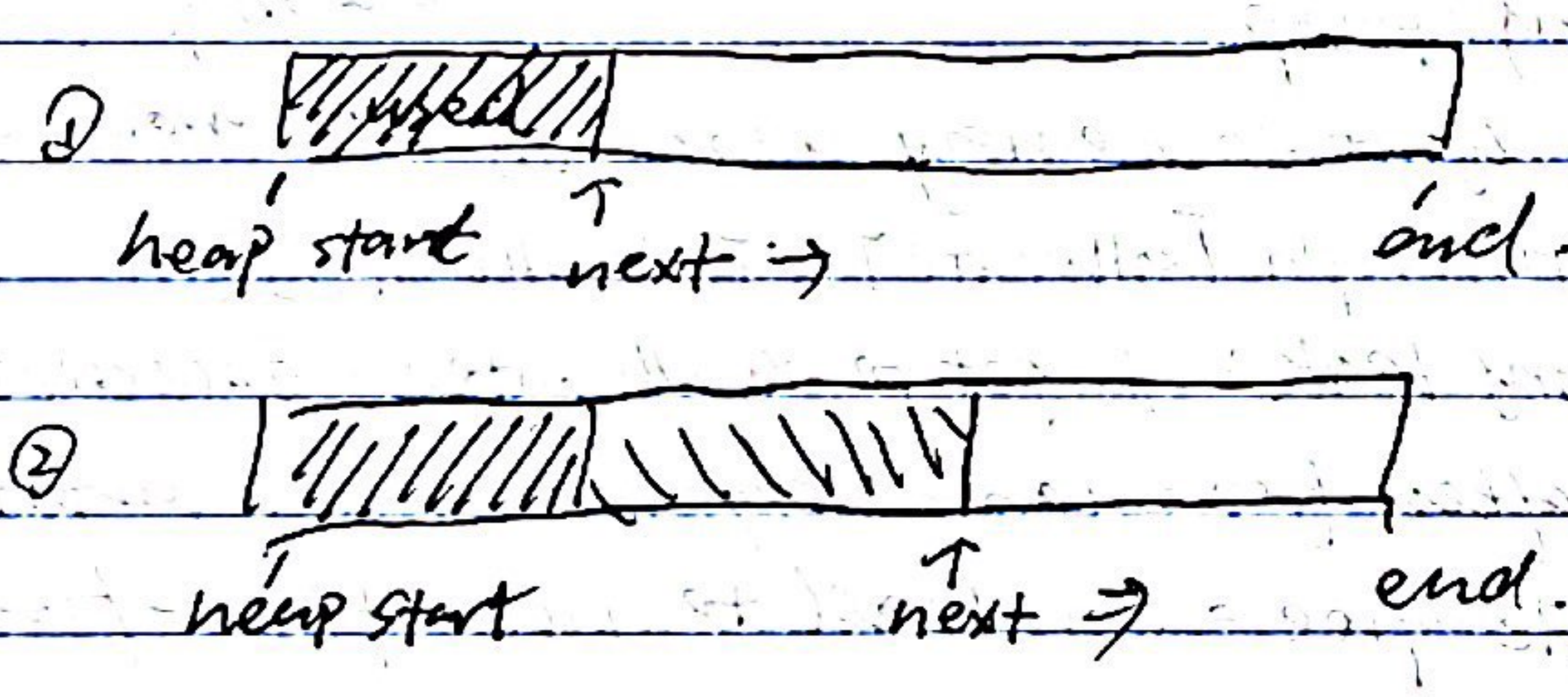
Resource acquisition is initialization (RAII) Rust ensures memory safety & thread safety

<Allocator Design>.

goals:
- ① to manage the available heap memory:
  - ⇒ return unused memory on 'alloc'
  - keep track of memory freed by 'dealloc' so that it can be reused.
- ② effectively utilize the available memory & keep fragmentation low.
  - work well for concurrent apps & scale to any # processors.
  - optimize the memory layout w.r.t. CPU caches to ↑ cache locality & avoid false sharing.

1. Bump Allocator (stack allocator)
   - allocates memory linearly & only keeps track of # allocated bytes & # allocations.

- Design idea:



linearly allocate memory by increasing [next]

It is often implemented with an allocation counter:
- +1 on each 'alloc' call
- −1 on each 'dealloc' call

when [counter] == 0, it means that all allocations on the heap have been deallocated.

⇒ [next] will be reset to the [heap-start].

⇒ bump allocator can only free all memory at once!

- pros: very fast - can be optimized to just a few assembly instrs.
- cons: it can only reuse deallocated memory after all allocations have been freed.

  ⇓

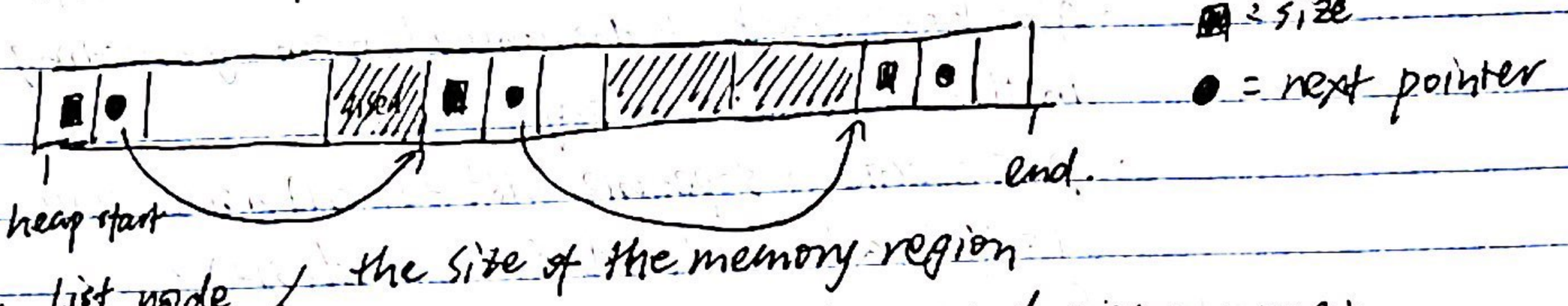
  [test_case]: many_boxes_long_lived.

① only splits the heap into smaller blocks
but never merges them back.
(ext fragmentation)

con { ② worse performance

2. Linked List Allocator

- constructing a single linked list in the freed memory.
to store the information about freed region.



each list node (free list) { the size of the memory region
or pointer to the next unused memory region.

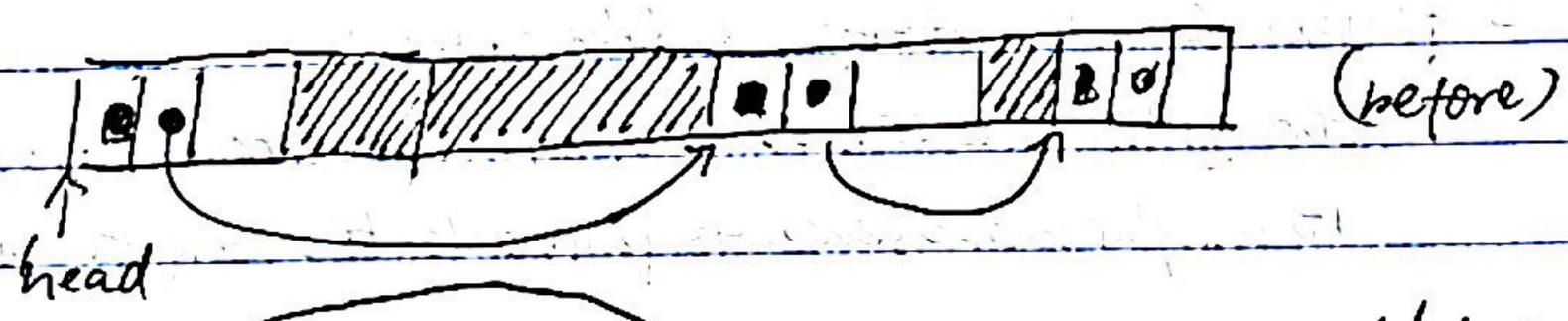
2.1. Implementation of [add_free_region]

takes the `addr` & `size` of a mem region
adds it to the front of the list.

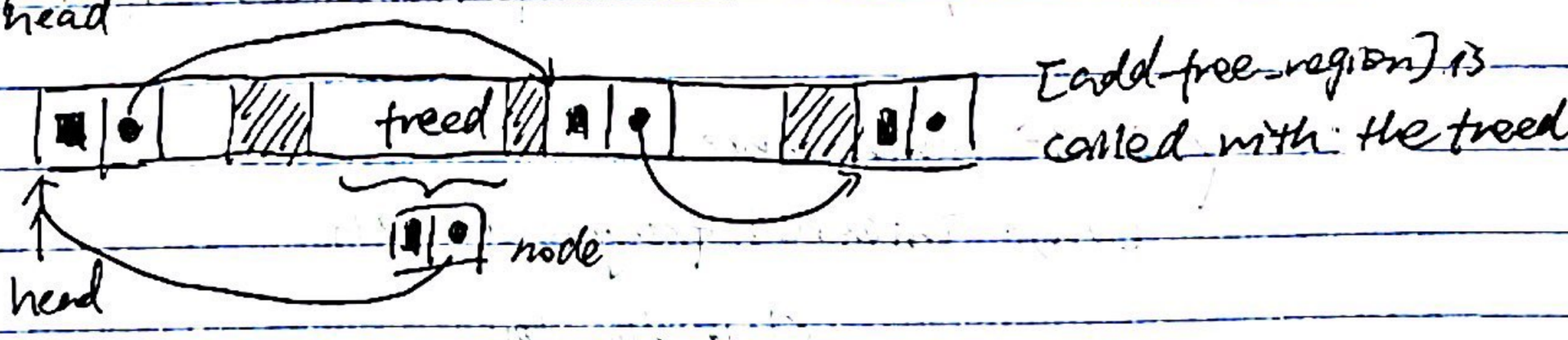
① checks that the given region has the necessary size & alignment
for storing a `ListNode`.

② creates the node & inserts it into the head of the list

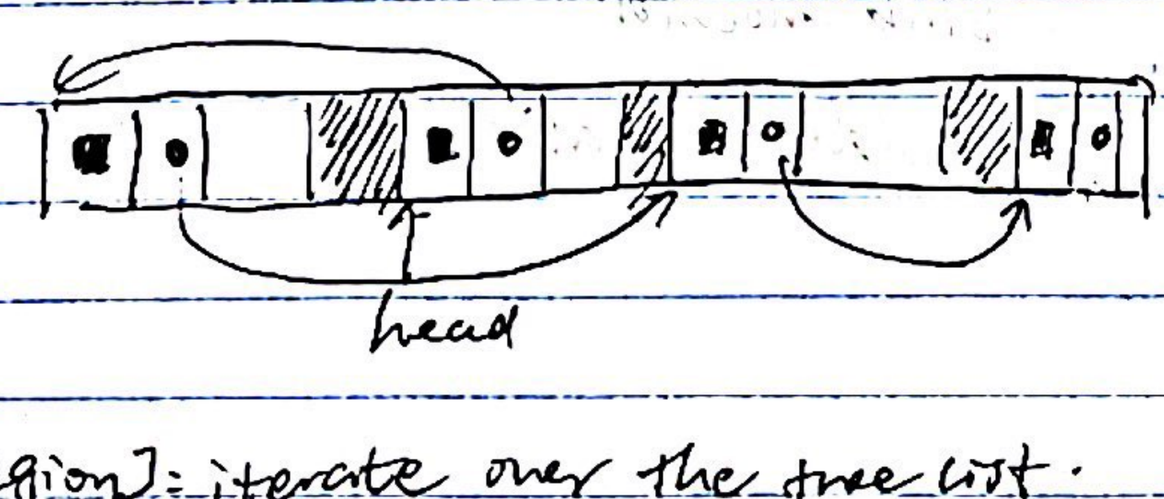
2.0.)



2.1).



2.2).



2.2. [find_region]: iterate over the free list.
to find suitable region.

2.3. [alloc_from_region] = given a `region`, `size`, `align`

① calculate the start & end address of a potential allocation
(alloc_start = align_up(region.start, align))

② calculate the excessive size ⇒ { >0 && ≥ sizeof<ListNode>
{ ==0

③ return the `alloc_start`

3. Fixed-Size Block Allocator.

- idea: Instead of allocating exactly as much memory as required,
  we define a small # of block sizes & round up each allocation to the next block size
  - Also keep track of the unused mem by creating a linked list in the unused memory.
  - But create a seperate list for each size class.
    (e.g. head-16, head-64, head-512)
- alloc: ① choose the smallest mem block that satisfies the request. (e.g. 16 for 12)
  from available blocks
  ② retrieve the head pointer for the list (e.g. head-16)
  ③ remove the first block & return it
- dealloc: ① round up the freed allocation size to the next block size.
  ② retrieve the head pointer for the list
  ③ add the freed block to the front of the list & update head ptr.
- create new blocks: (2 ways) — allocate new block from fallback allocator
  — split a larger block from a different list.
- fallback allocator:
  Large allocations (>2KB) are often rare, especially in OS kernels.
  It might make sense to fallback to a different allocator for these alloc.

- pros: performance
  cons: internal fragmentation
- Variations: Slab allocator
  Buddy allocator