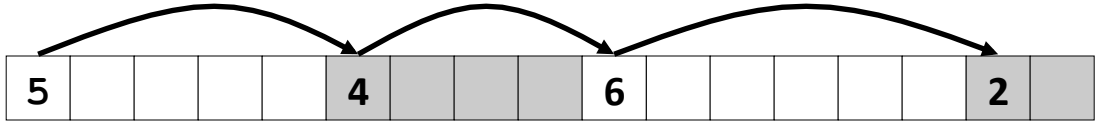
Section 10: Memory Allocation Topics

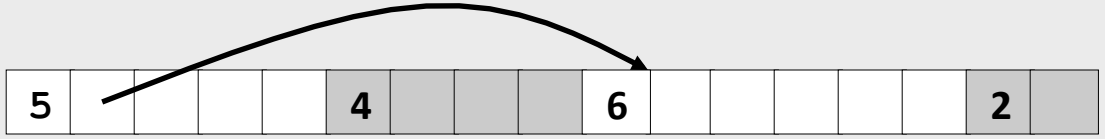
- ~~**Dynamic memory allocation**~~
  - ~~Size/number of data structures may only be known at run time~~
  - ~~Need to allocate space on the heap~~
  - ~~Need to de-allocate (free) unused memory so it can be re-allocated~~
- **Implementation**
  - ~~Implicit free lists~~
  - Explicit free lists – subject of next programming assignment
  - Segregated free lists
- **Garbage collection**
- **Common memory-related bugs in C programs**

Keeping Track of Free Blocks

- Method 1: *Implicit free list* using length—links all blocks



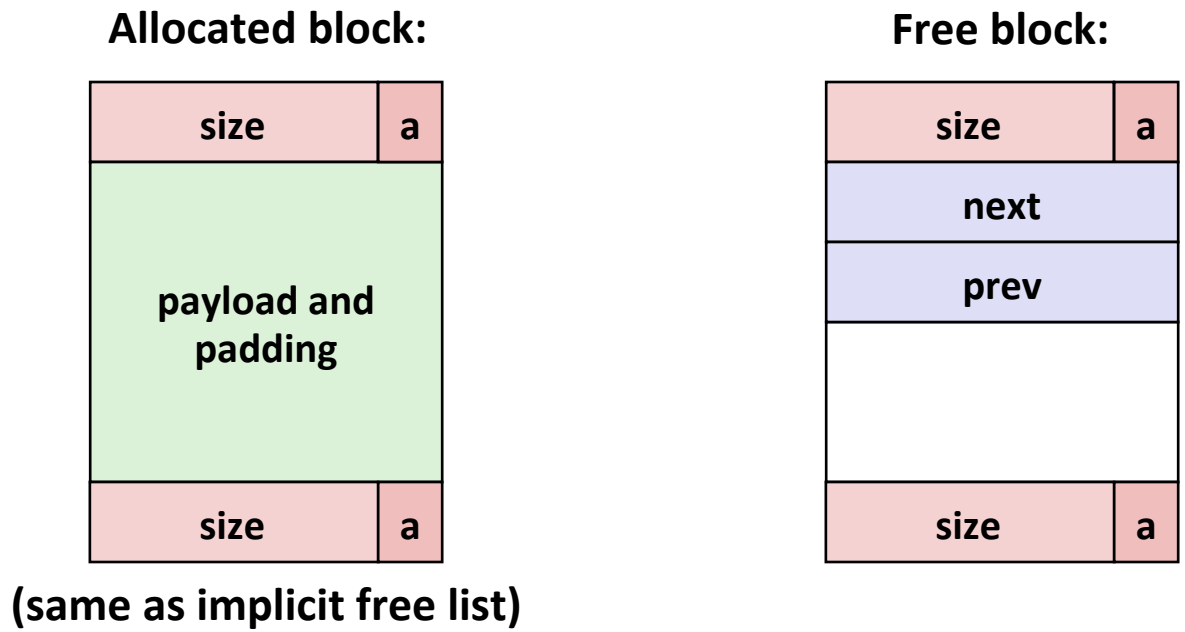
- Method 2: *Explicit free list* among the free blocks using pointers



- Method 3: *Segregated free list*
  - Different free lists for different size classes

- Method 4: *Blocks sorted by size*
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Explicit Free Lists



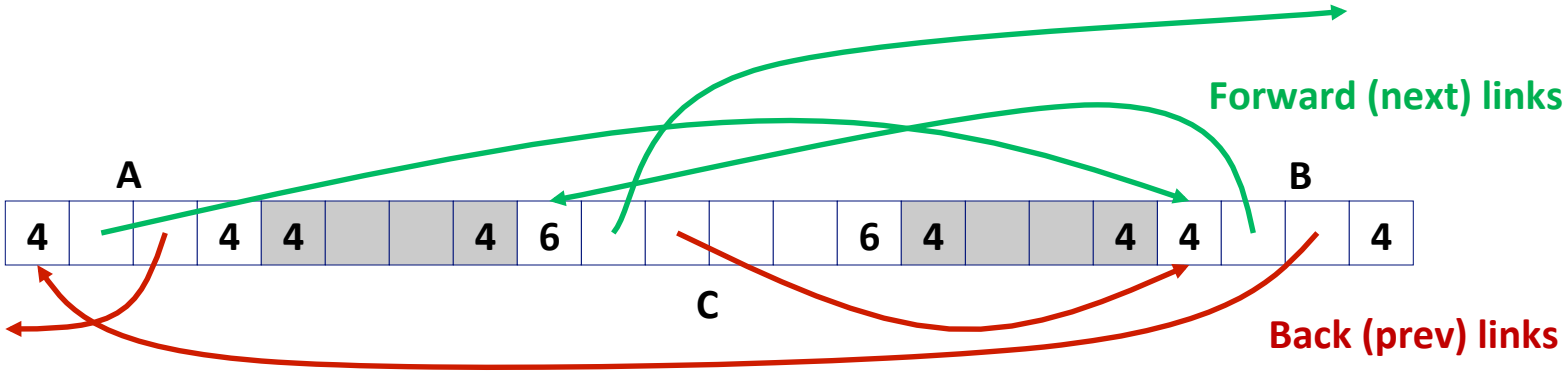
(same as implicit free list)

- **Maintain list(s) of *free* blocks, rather than implicit list of *all* blocks**
  - The "next" free block could be anywhere in the heap
    - So we need to store forward/back pointers, not just sizes
  - Luckily we track only free blocks, so we can use payload area for pointers
  - Still need boundary tags for coalescing

Explicit Free Lists

- Logically (doubly-linked lists):

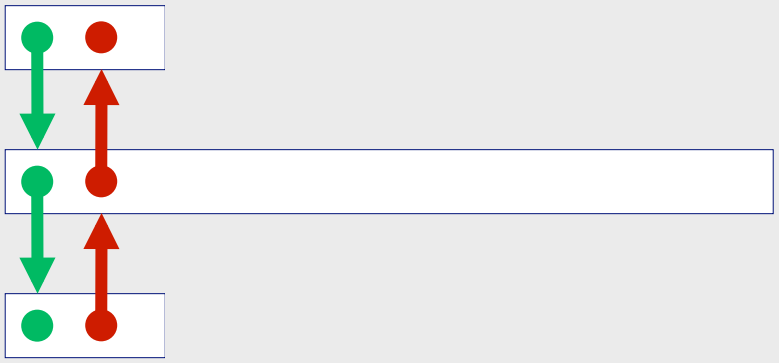
- Physically: blocks can be in any order



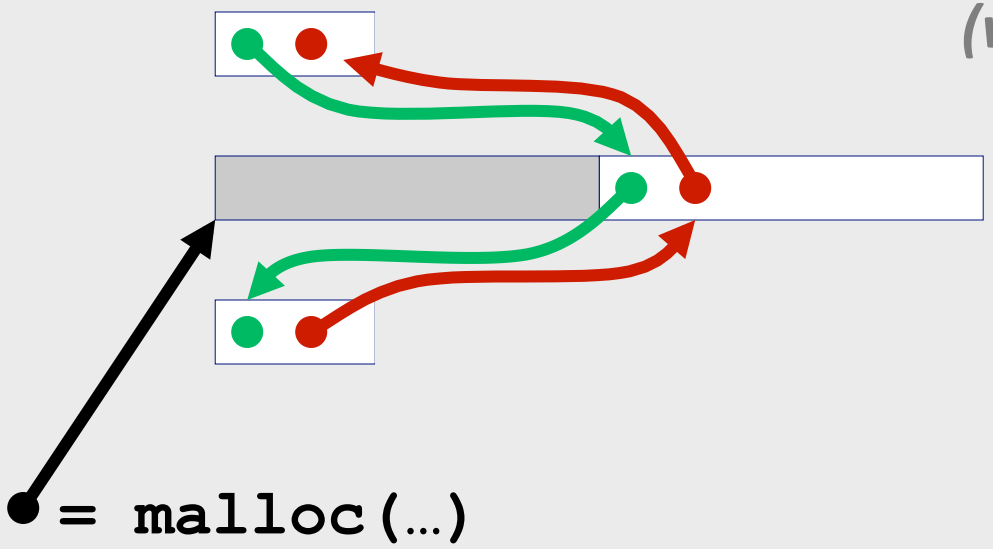
Allocating From Explicit Free Lists

conceptual graphic

Before



After *(with splitting)*



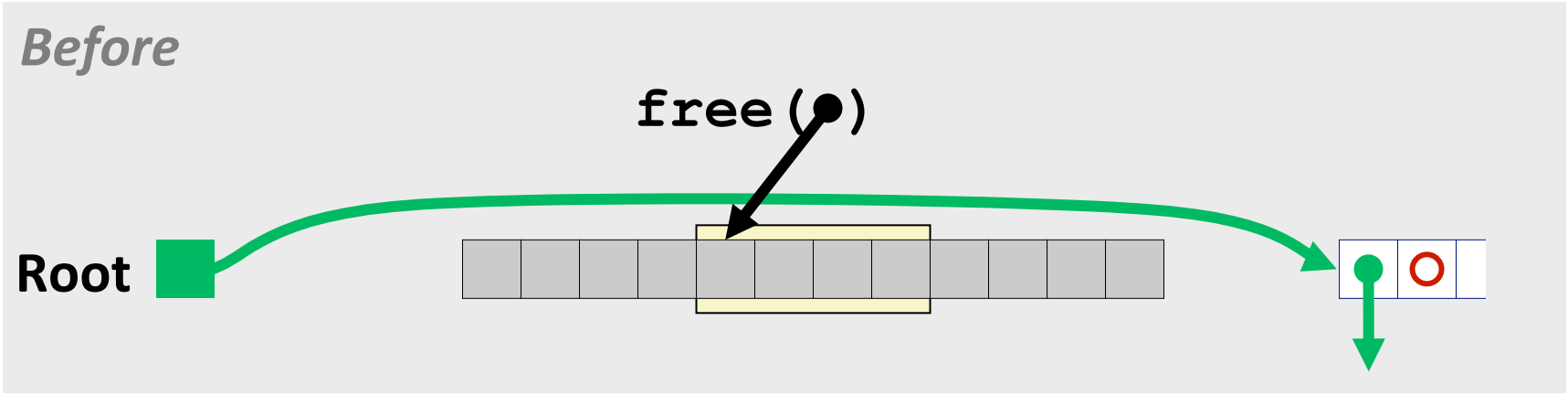
Freeing With Explicit Free Lists

- ***Insertion policy*: Where in the free list do you put a newly freed block?**
  - LIFO (last-in-first-out) policy
    - Insert freed block at the beginning of the free list
    - ***Pro:*** simple and constant time
    - ***Con:*** studies suggest fragmentation is worse than address ordered
  - Address-ordered policy
    - Insert freed blocks so that free list blocks are always in address order:

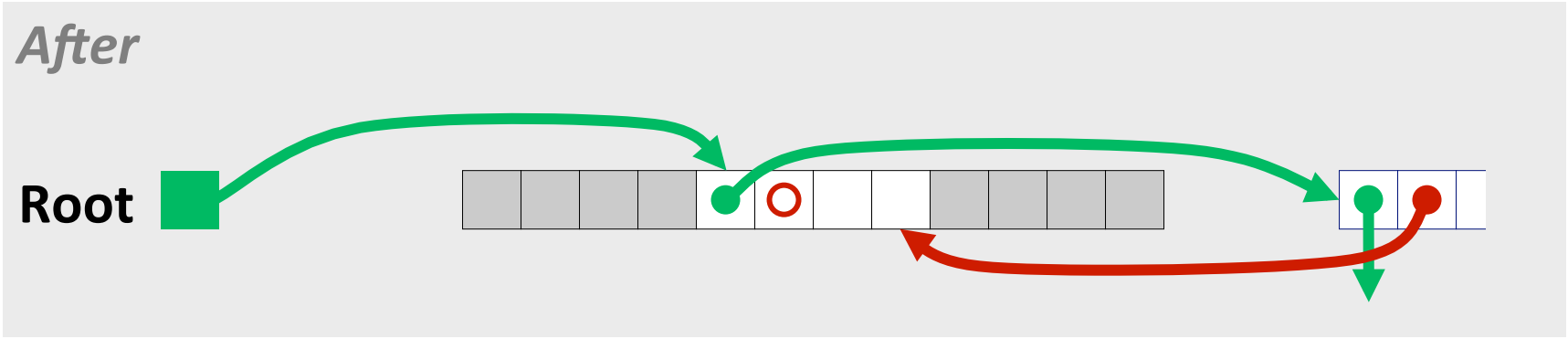
      addr(prev) < addr(curr) < addr(next)
    - ***Con:*** requires linear-time search when blocks are freed
    - ***Pro:*** studies suggest fragmentation is lower than LIFO

Freeing With a LIFO Policy (Case 1)

conceptual graphic

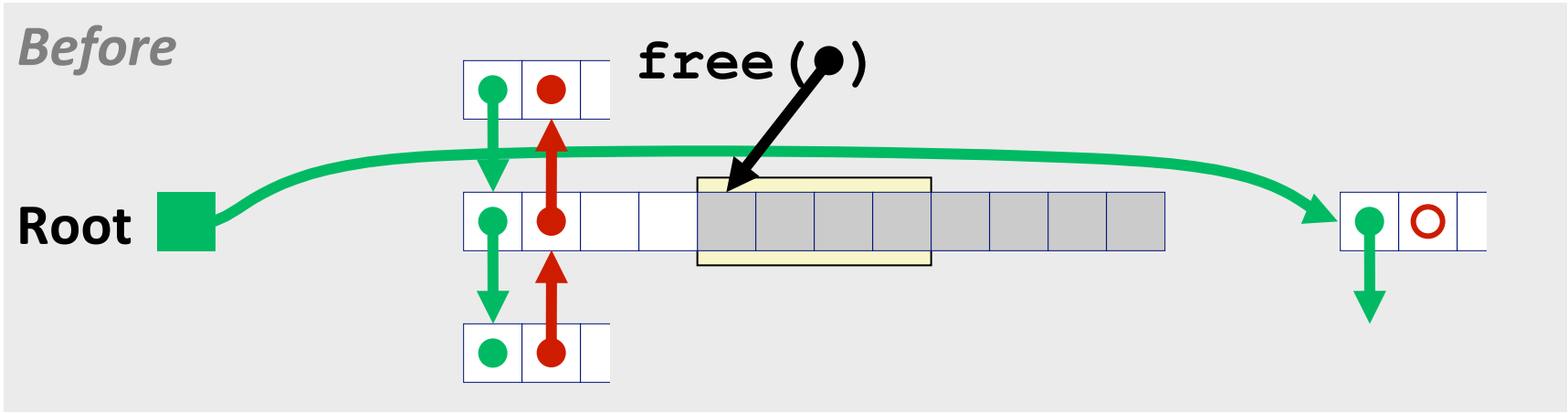


- Insert the freed block at the root of the list

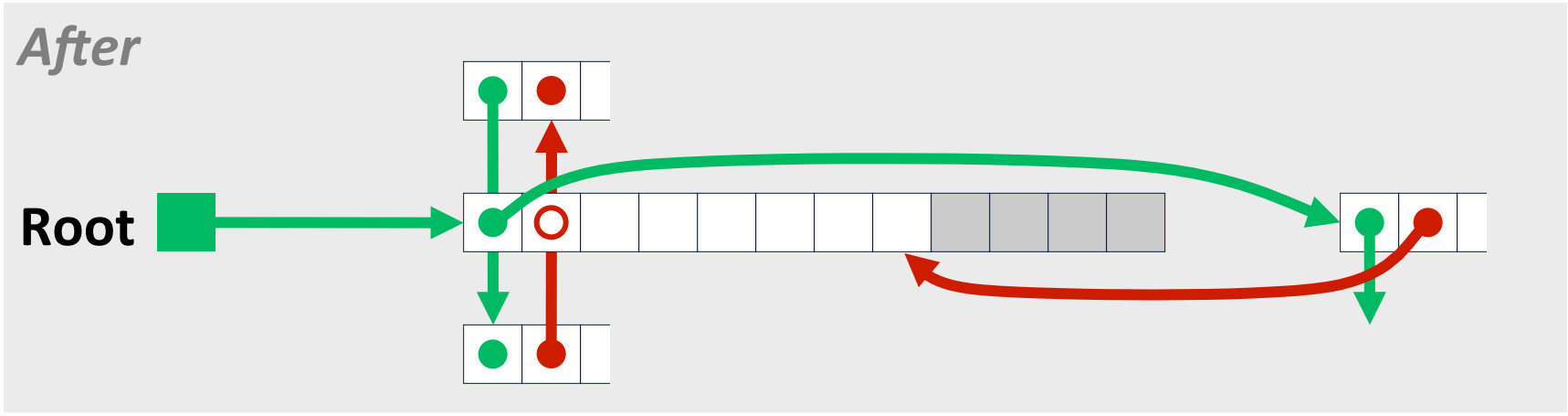


Freeing With a LIFO Policy (Case 2)

conceptual graphic

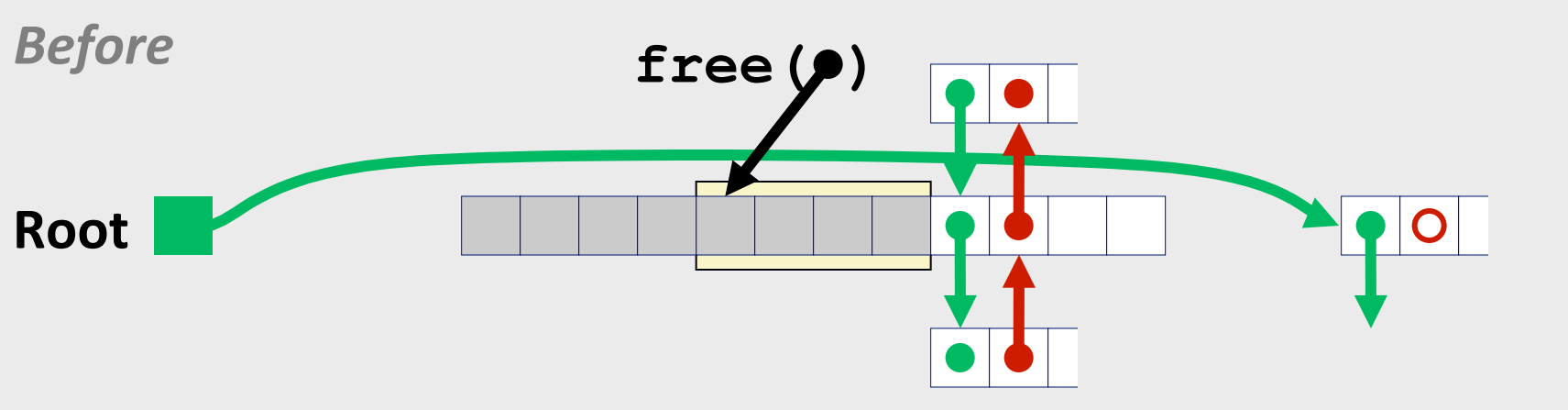


- Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list

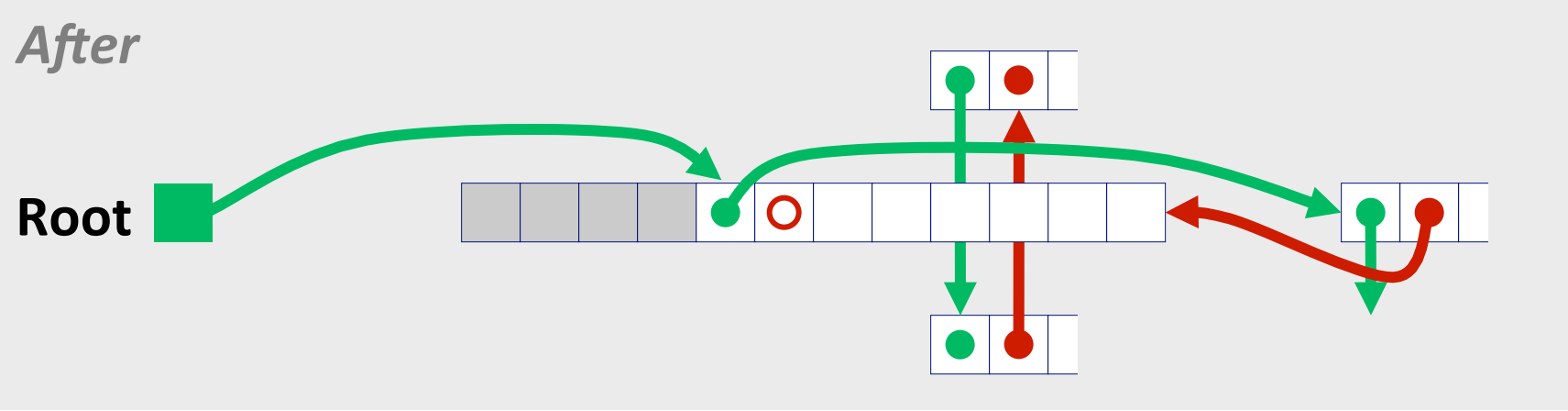


Freeing With a LIFO Policy (Case 3)

conceptual graphic

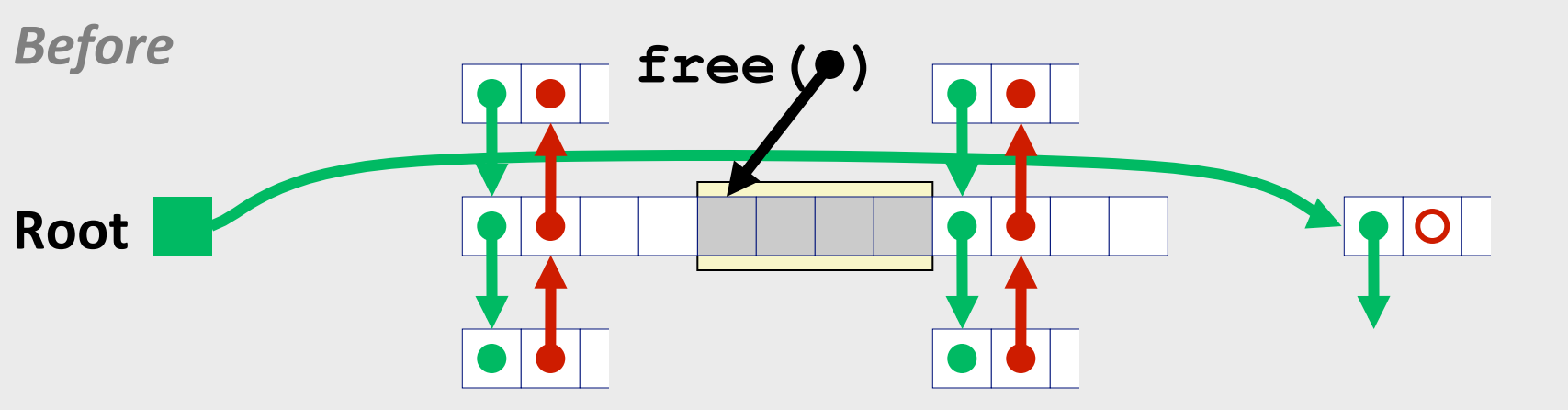


- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list

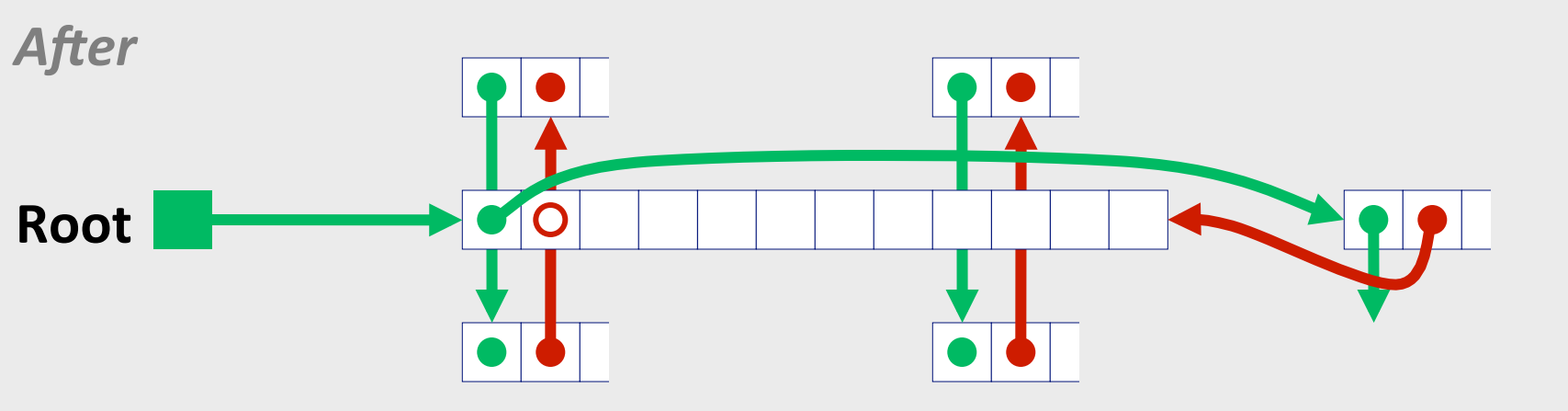


Freeing With a LIFO Policy (Case 4)

conceptual graphic



- Splice out predecessor and successor blocks, coalesce all 3 memory blocks and insert the new block at the root of the list



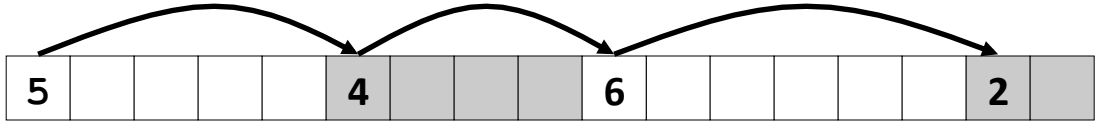
Explicit List Summary

- **Comparison to implicit list:**
  - Allocate is linear time in number of ***free*** blocks instead of ***all*** blocks
    - ***Much faster*** when most of the memory is full
  - Slightly more complicated allocate and free since needs to splice blocks in and out of the list
  - Some extra space for the links (2 extra words needed for each block)
    - Possibly increases minimum block size, leading to more internal fragmentation

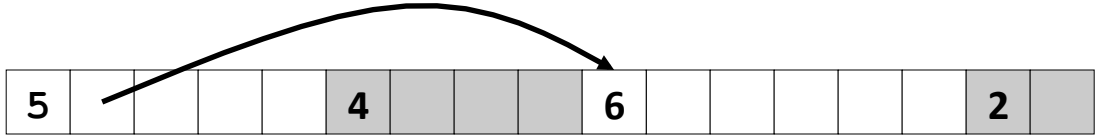
- **Most common use of explicit lists is in conjunction with segregated free lists**
  - Keep multiple linked lists of different size classes, or possibly for different types of objects

Keeping Track of Free Blocks

- Method 1: *Implicit list* using length—links all blocks



- Method 2: *Explicit list* among the free blocks using pointers

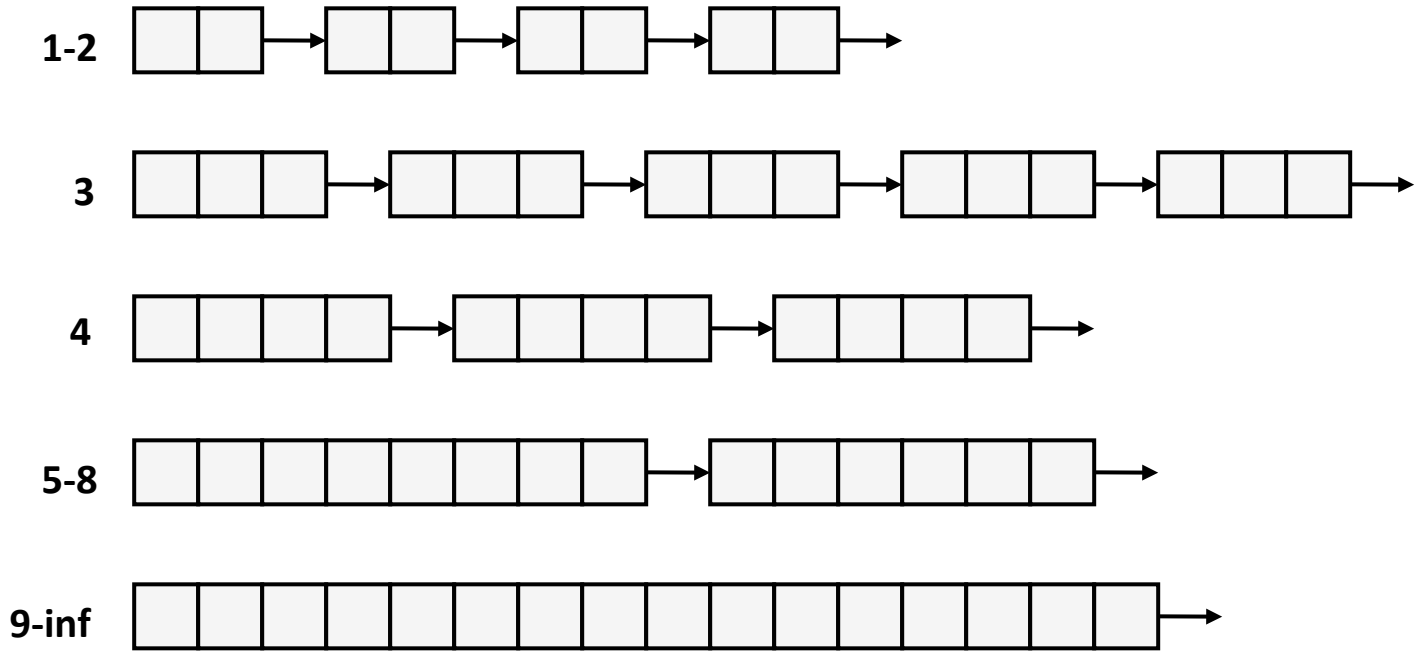


- Method 3: *Segregated free list*
  - Different free lists for different size classes

- Method 4: *Blocks sorted by size*
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Segregated List (Seglist) Allocators

- Each *size class* of blocks has its own free list



- Often have separate classes for each small size
- For larger sizes: One class for each two-power size

Seglist Allocator

- **Given an array of free lists, each one for some size class**

- **To allocate a block of size *n*:**
  - Search appropriate free list for block of size *m* > *n*
  - If an appropriate block is found:
    - Split block and place fragment on appropriate list (optional)
  - If no block is found, try next larger class
  - Repeat until block is found

- **If no block is found:**
  - Request additional heap memory from OS (using **`sbrk()`**)
  - Allocate block of *n* bytes from this new memory
  - Place remainder as a single free block in largest size class

Seglist Allocator (cont.)

- **To free a block:**
  - Coalesce and place on appropriate list (optional)

- **Advantages of seglist allocators**
  - Higher throughput
    - log time for power-of-two size classes
  - Better memory utilization
    - First-fit search of segregated free list approximates a best-fit search of entire heap.
    - Extreme case: Giving each block its own size class is equivalent to best-fit.

Summary of Key Allocator Policies

- **Placement policy:**
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
  - ***Observation***: segregated free lists approximate a best fit placement policy without having to search entire free list
- **Splitting policy:**
  - When do we go ahead and split free blocks?
  - How much internal fragmentation are we willing to tolerate?
- **Coalescing policy:**
  - ***Immediate coalescing:*** coalesce each time `free()` is called
  - ***Deferred coalescing:*** try to improve performance of `free()` by deferring coalescing until needed. Examples:
    - Coalesce as you scan the free list for `malloc()`
    - Coalesce when the amount of external fragmentation reaches some threshold