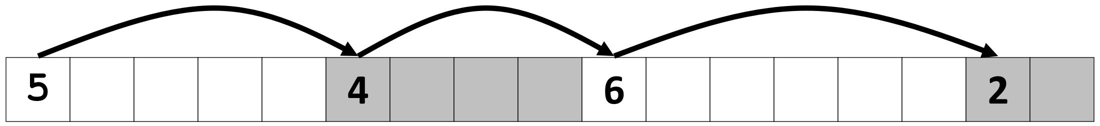
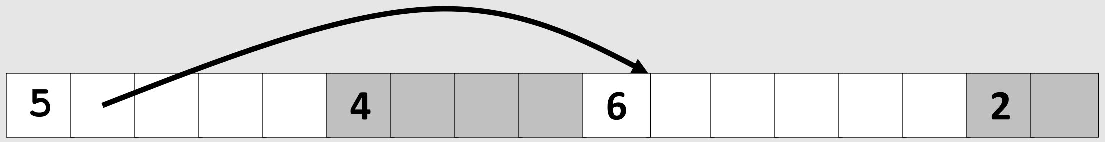
Virtual Memory: *malloc, method 2: explicit lists*

Keeping Track of Free Blocks

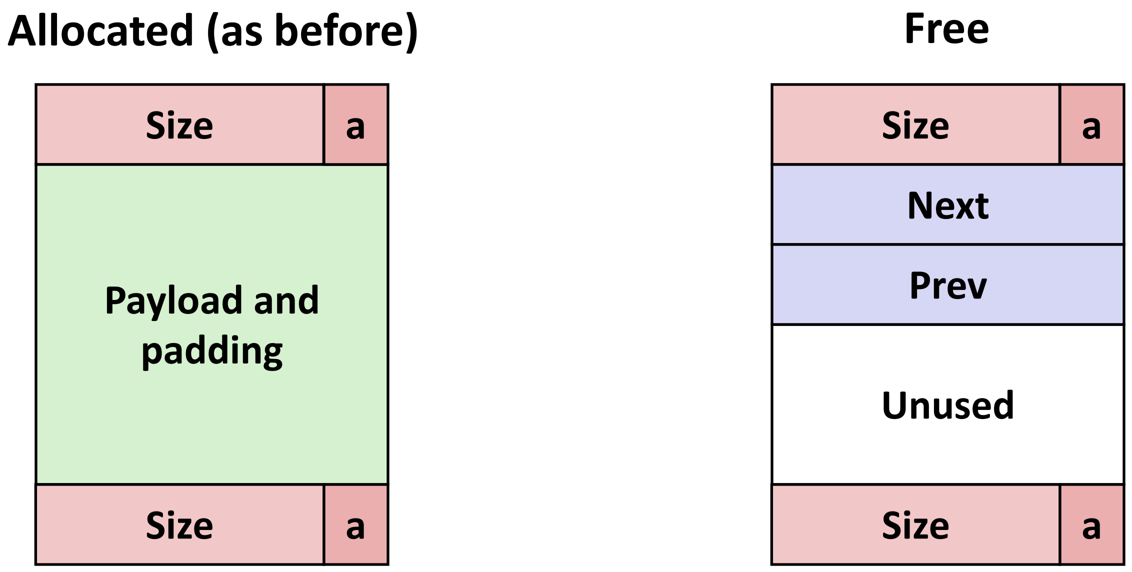
- Method 1: *Implicit free list* using length—links all blocks



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



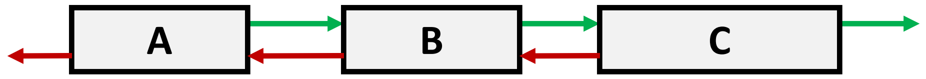
Explicit Free Lists



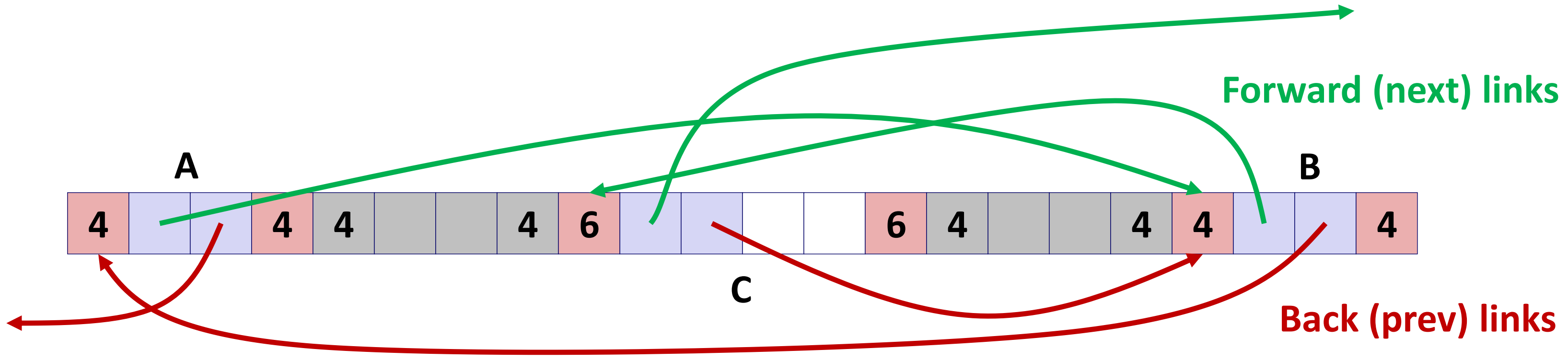
- **Maintain list(s) of *free* blocks, not *all* blocks**
  - The “next” free block could be anywhere
    - So we need to store forward/back pointers, not just sizes
  - Still need boundary tags for coalescing
  - Luckily we track only free blocks, **so we can use payload area**

Explicit Free Lists

- **Logically:**

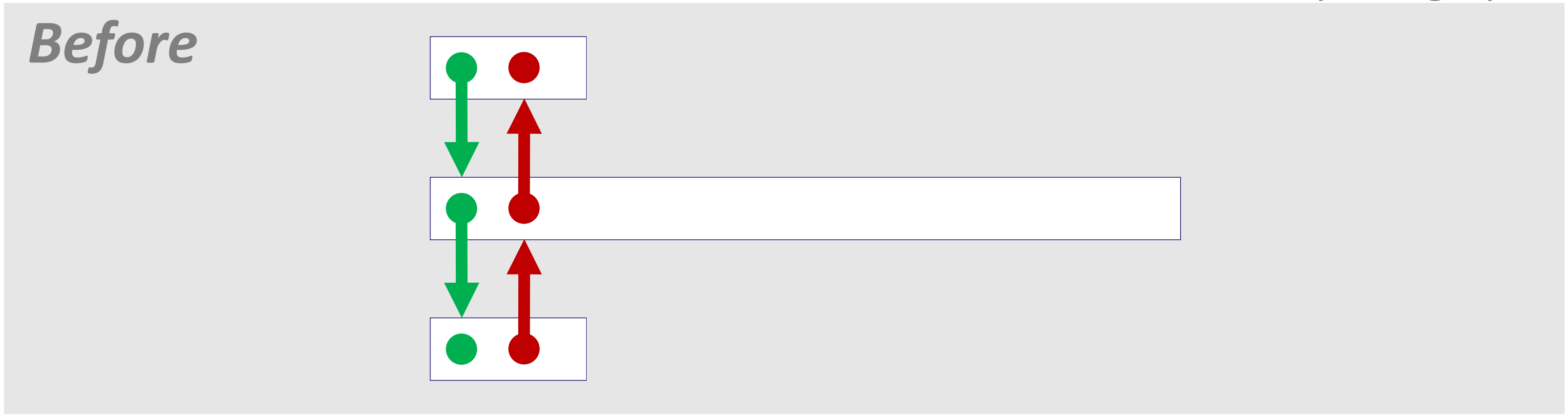


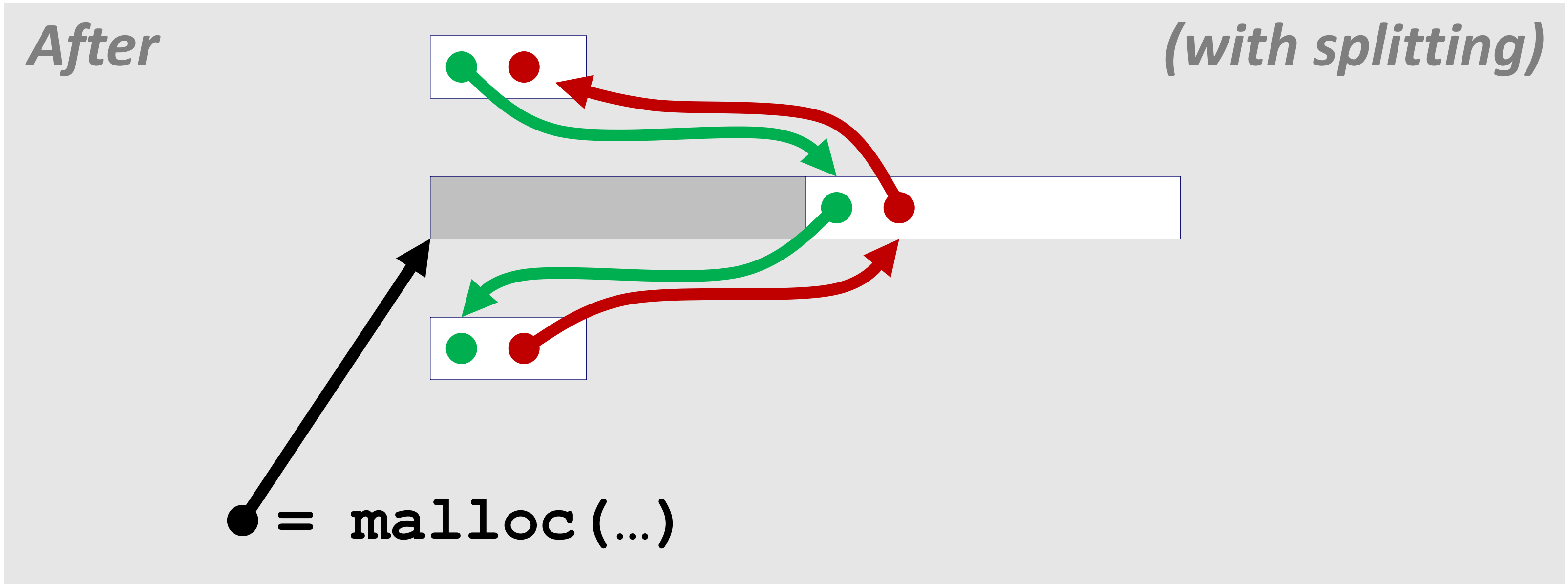
- **Physically: blocks can be in any order**



Allocating From Explicit Free Lists

conceptual graphic





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:

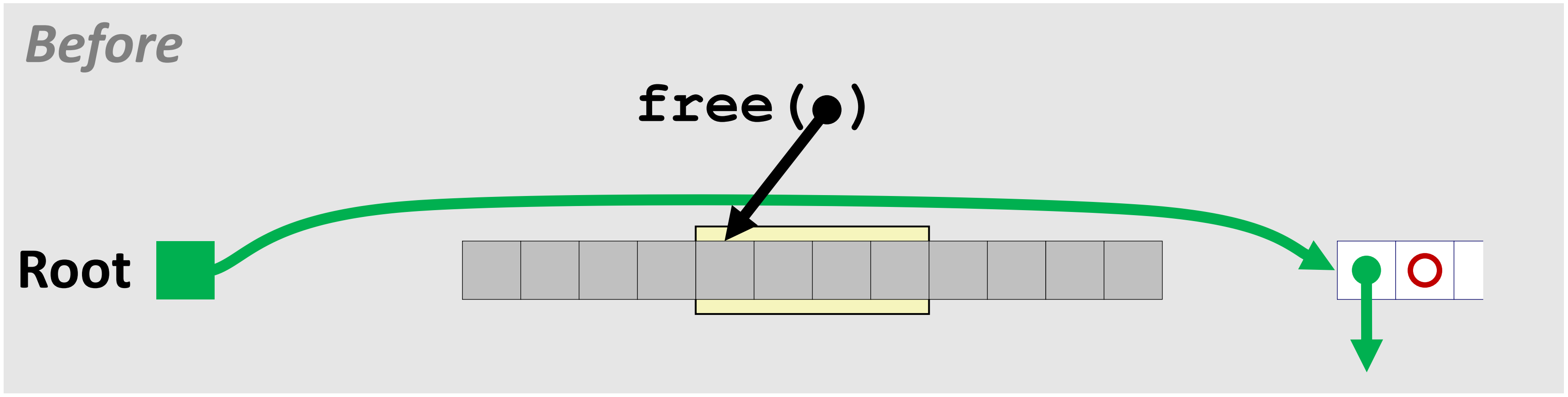
    addr(prev) < addr(curr) < addr(next)
  - ***Con:*** requires search
  - ***Pro:*** studies suggest fragmentation is lower than LIFO

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 search
  - *Pro:* studies suggest fragmentation is lower than LIFO

Freeing With a LIFO Policy (Case 1)

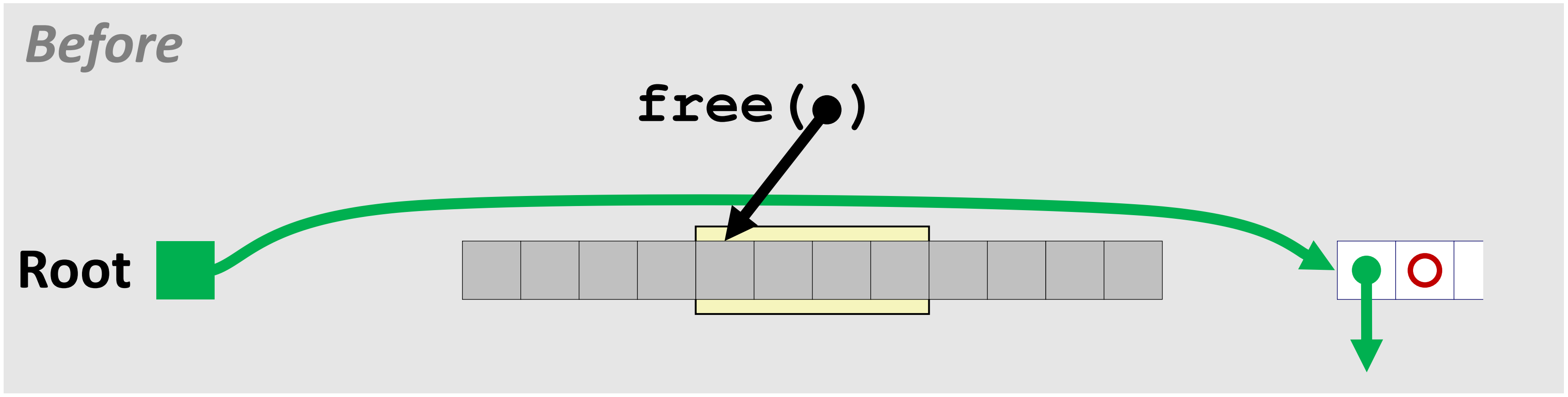
conceptual graphic



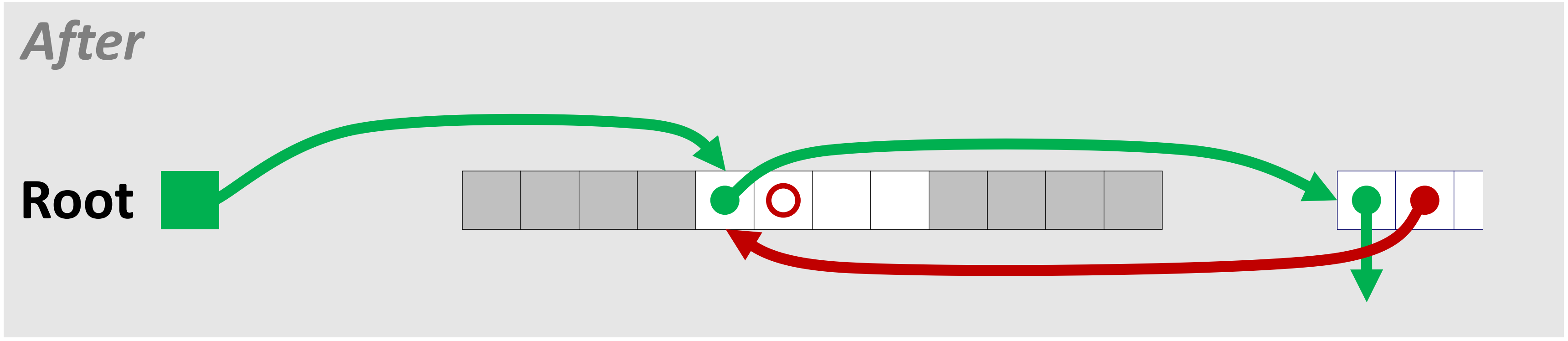
- Insert the freed block at the root of the list

Freeing With a LIFO Policy (Case 1)

conceptual graphic

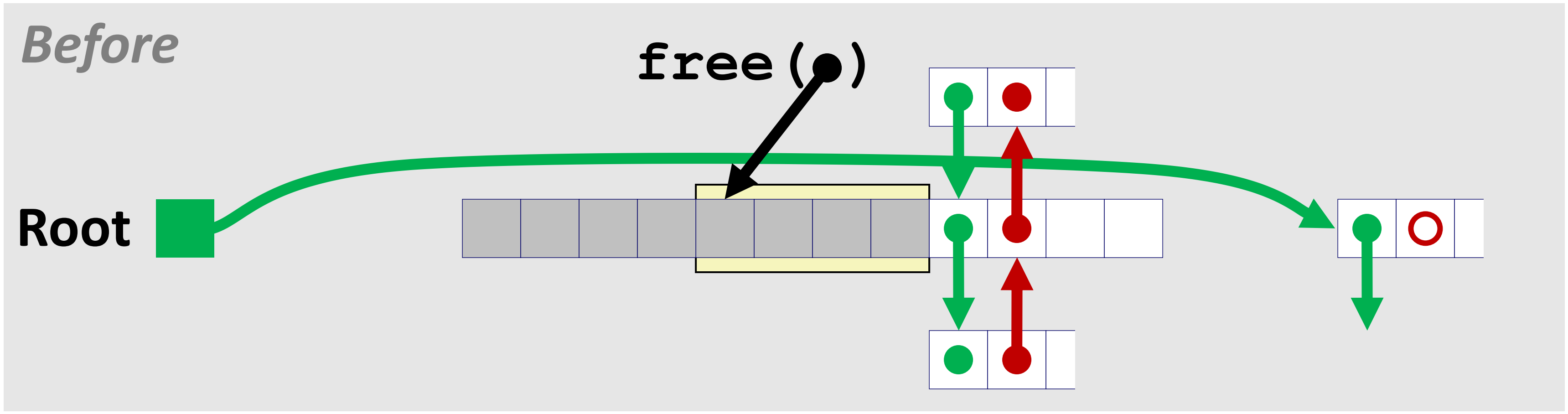


- Insert the freed block at the root of the list



Freeing With a LIFO Policy (Case 2)

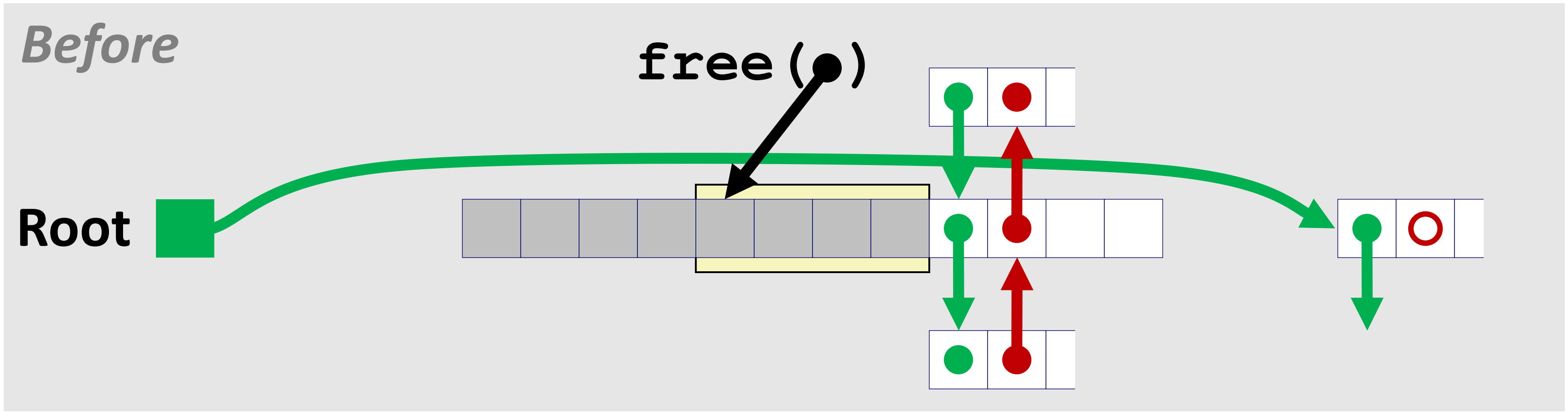
conceptual graphic



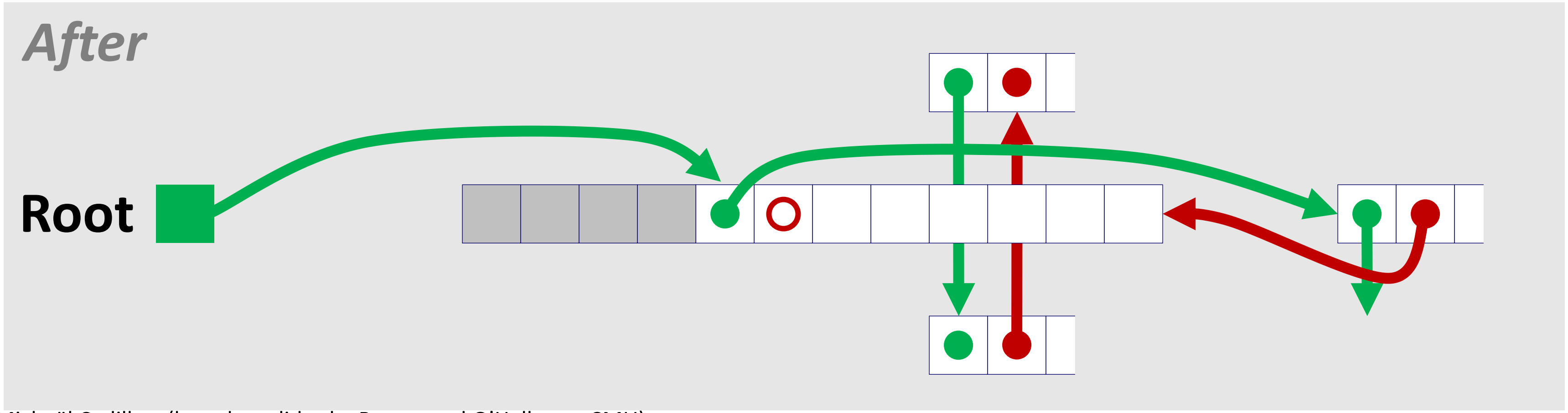
- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list

Freeing With a LIFO Policy (Case 2)

conceptual graphic

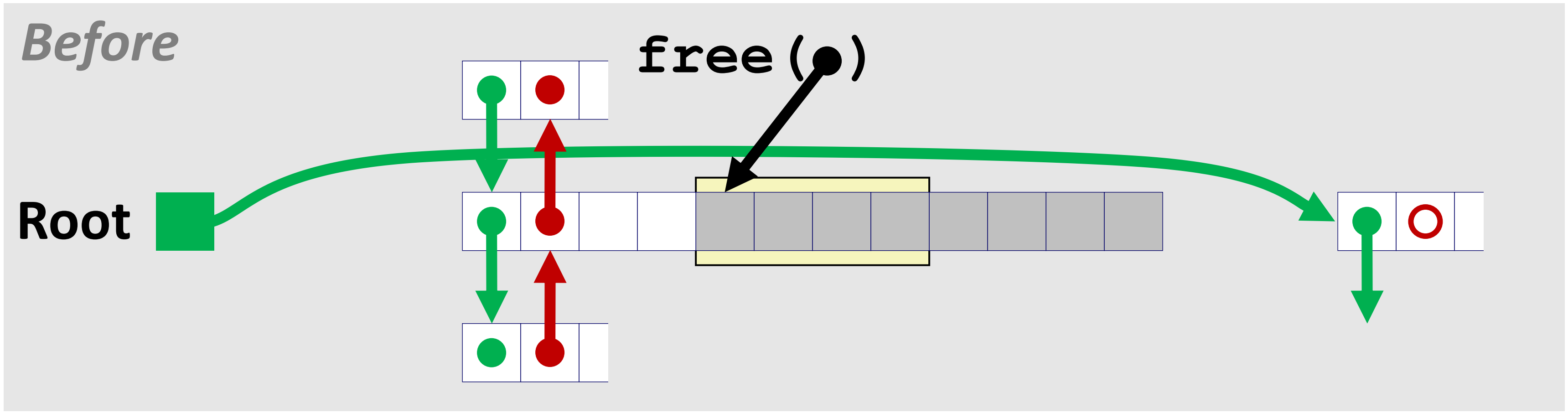


- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list



Freeing With a LIFO Policy (Case 3)

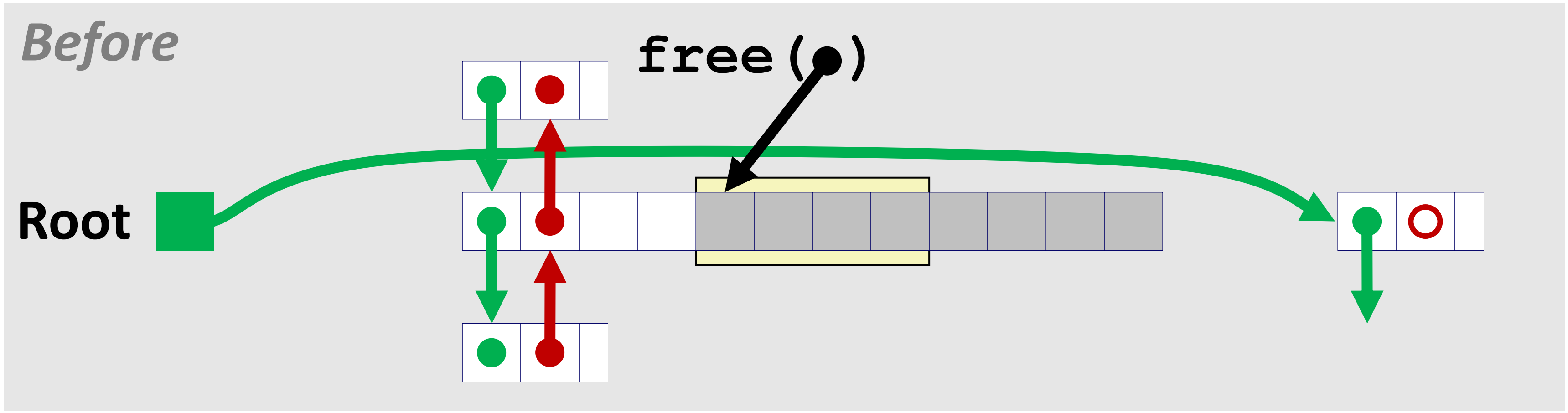
conceptual graphic



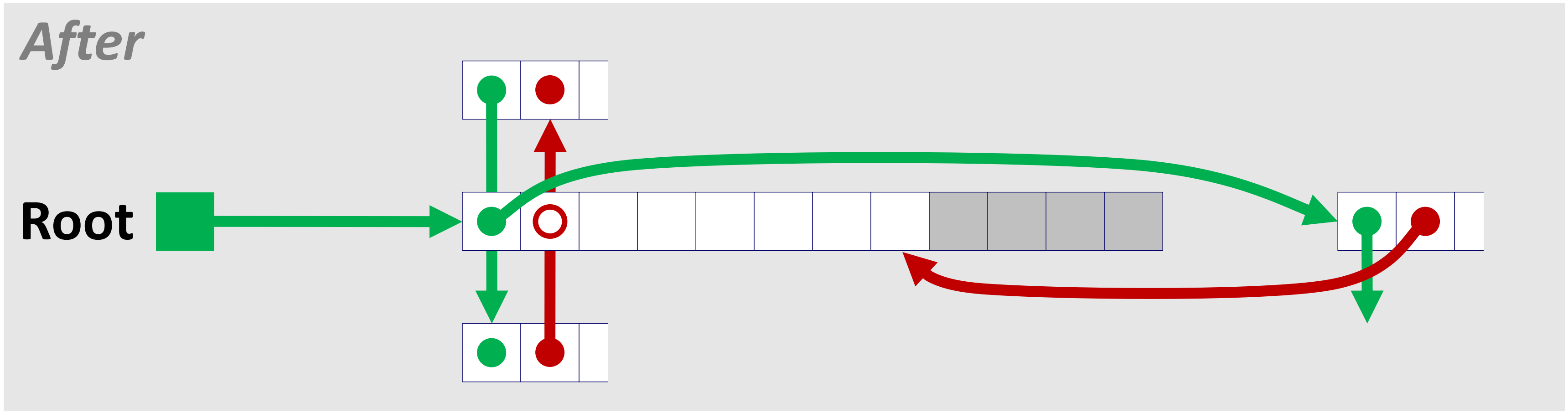
- Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list

Freeing With a LIFO Policy (Case 3)

conceptual graphic

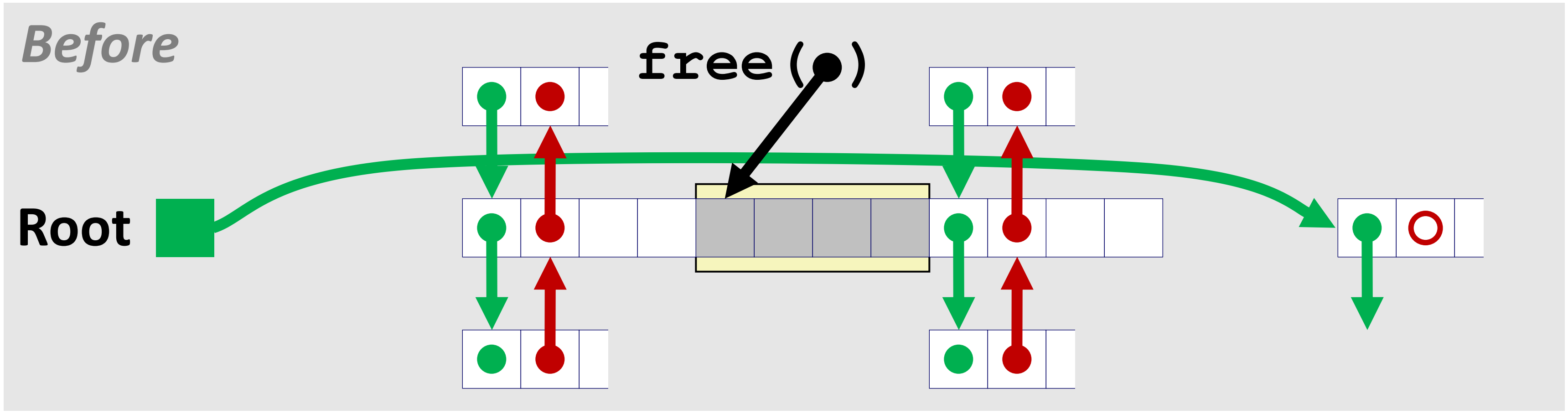


- Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list



Freeing With a LIFO Policy (Case 4)

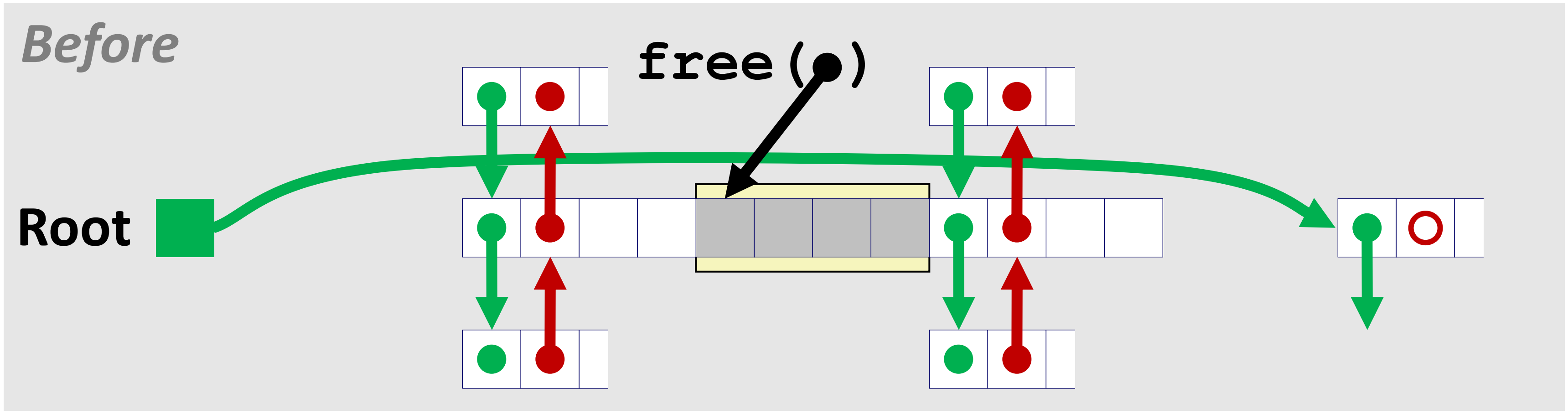
conceptual graphic



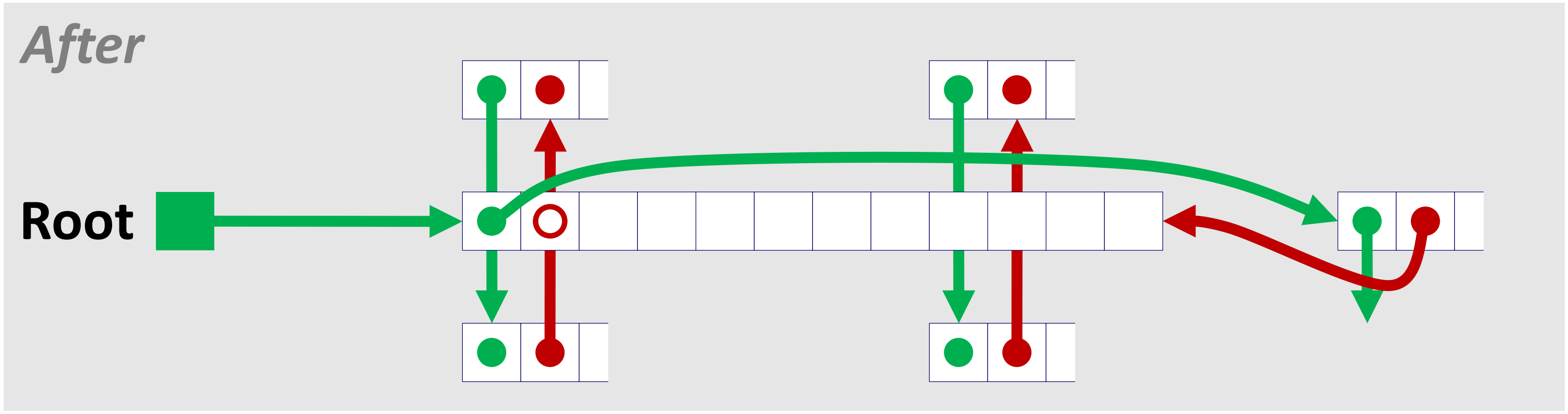
- Splice out predecessor and successor blocks, coalesce all 3 memory blocks and insert the new block at the root of the list

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)
    - Does this increase internal fragmentation?

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