CSE251: System Programming

20. Dynamic Memory Allocation (4)

Seongil Wi

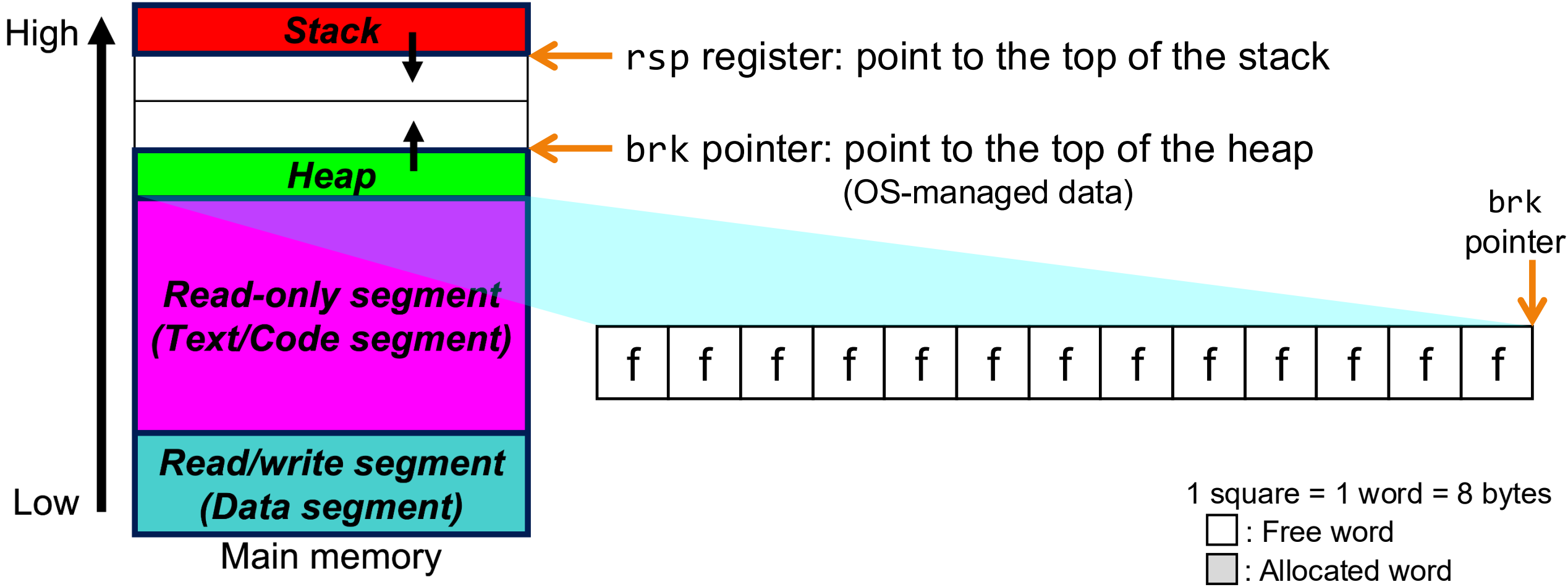
Department of Computer Science and Engineering

Final Exam

- 6/18 (Thu.), Class Time
- The scope includes all topics covered in this semester

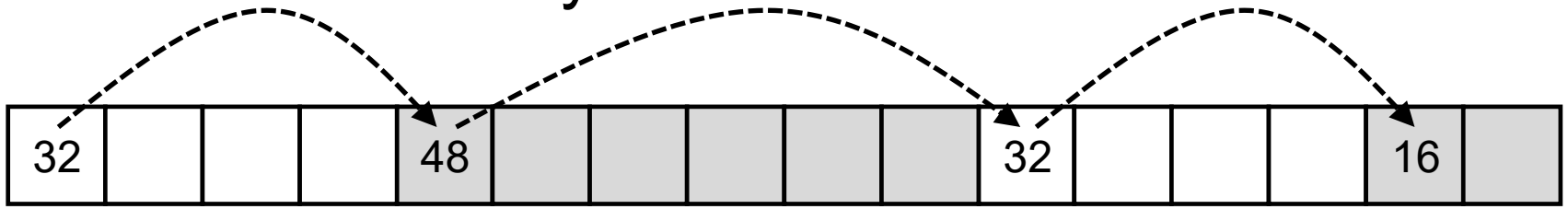
Review: Heap Allocator

- Maintains heap as collection of variable sized ***blocks***, which are either ***allocated*** or ***free***



Review: Implicit Lists

- Method #1: ***Implicit list*** using length. It links all blocks, regardless of whether they are allocated or free



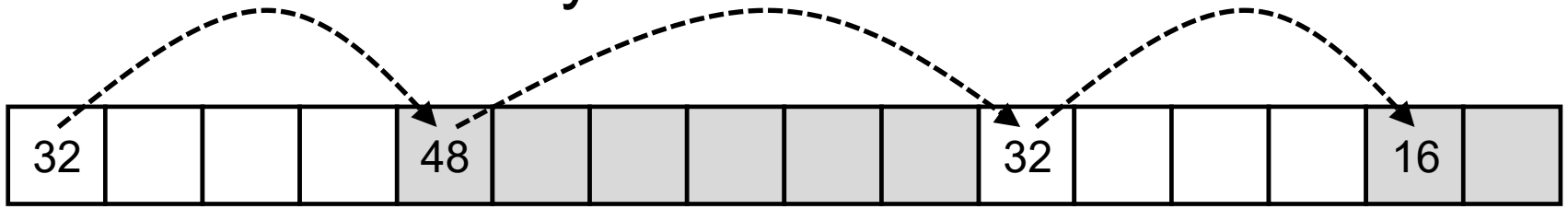
Review: Implicit Lists

- Implementation: very simple
- **Allocate cost: linear time worst case**
- Free cost: constant time worst case
  - even with coalescing
- Memory overhead
  - will depend on placement policy
  - First-fit, next-fit or best-fit
- Not used in practice for `malloc`/`free` **because of linear-time allocation**
- However, the concepts of splitting and boundary tag coalescing are general to *all* allocators

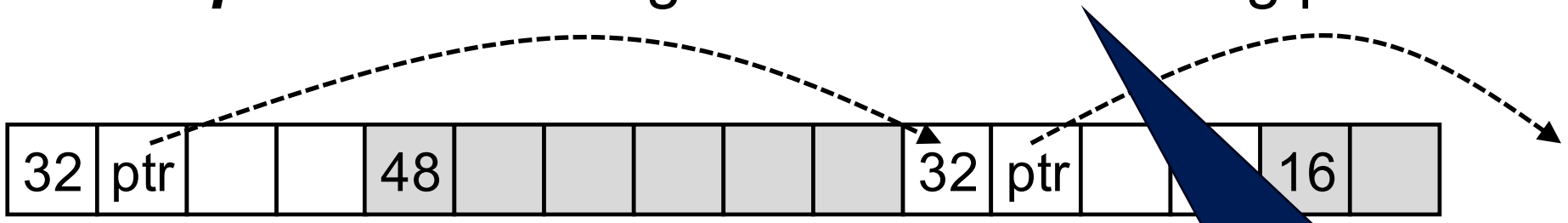
Keeping Track of Free Blocks

1 square = 1 word = 8 bytes
: Free word
: Allocated word

- Method #1: ***Implicit list*** using length. It links all blocks, regardless of whether they are allocated or free



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



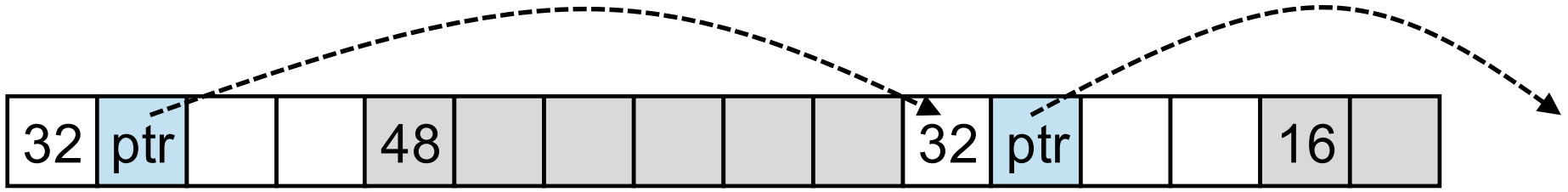
- Method #3: ***Segregated free list***
  - Different free lists for different size classes

Today's topic

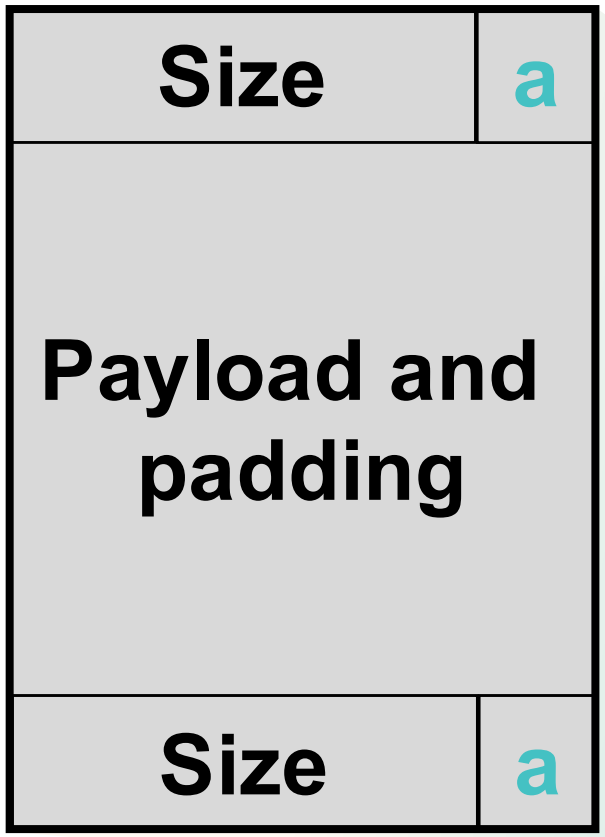
Explicit List

Explicit Lists

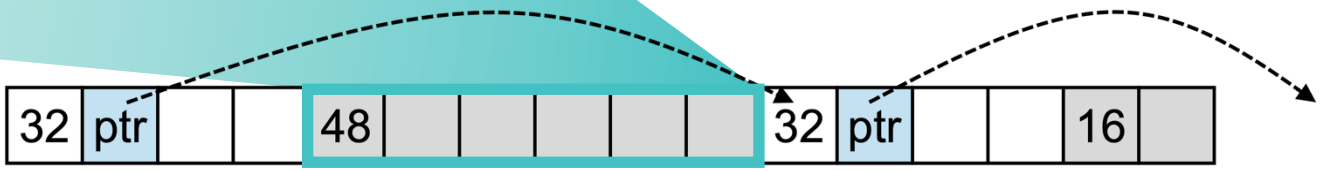
- Maintain list(s) of ***free*** blocks, not all blocks



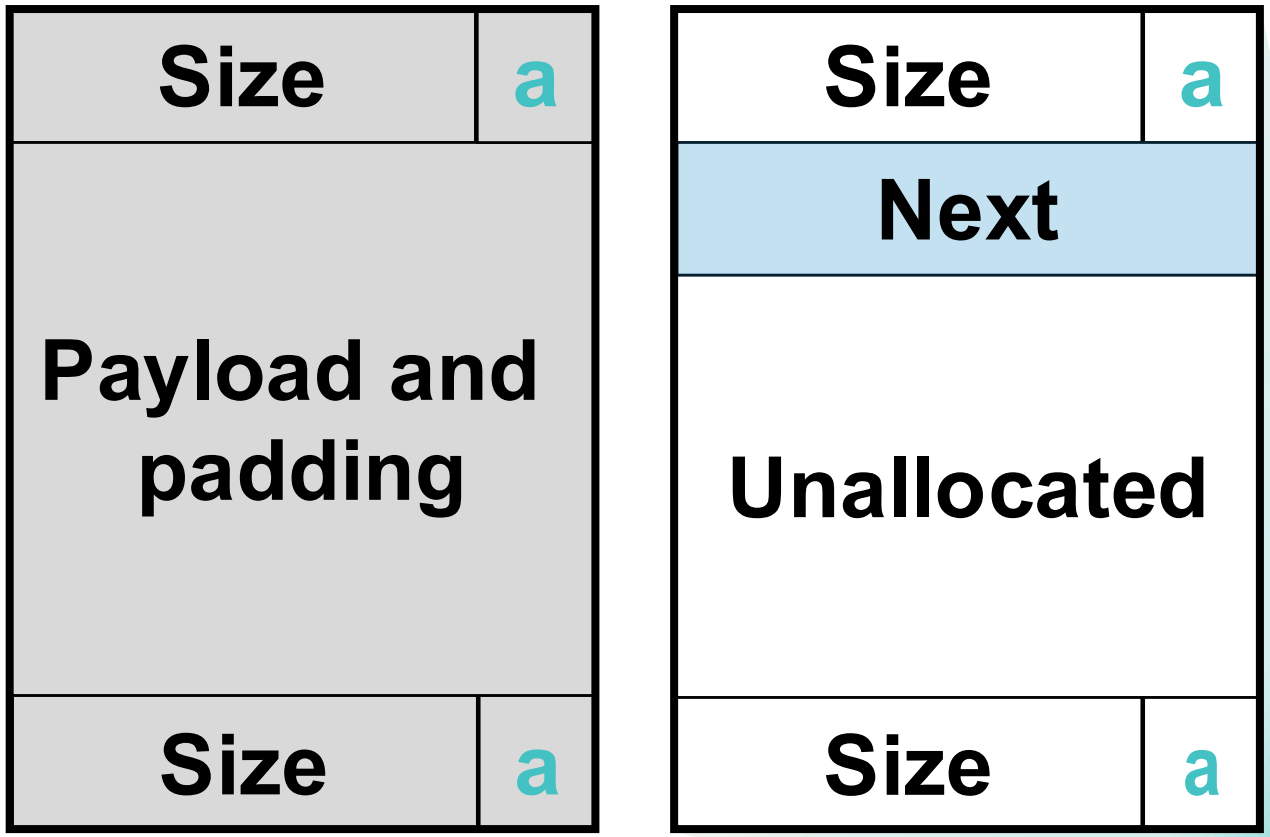
Explicit Lists

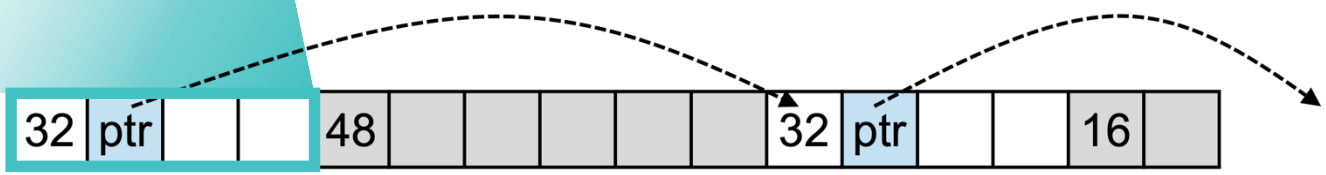


Allocated block

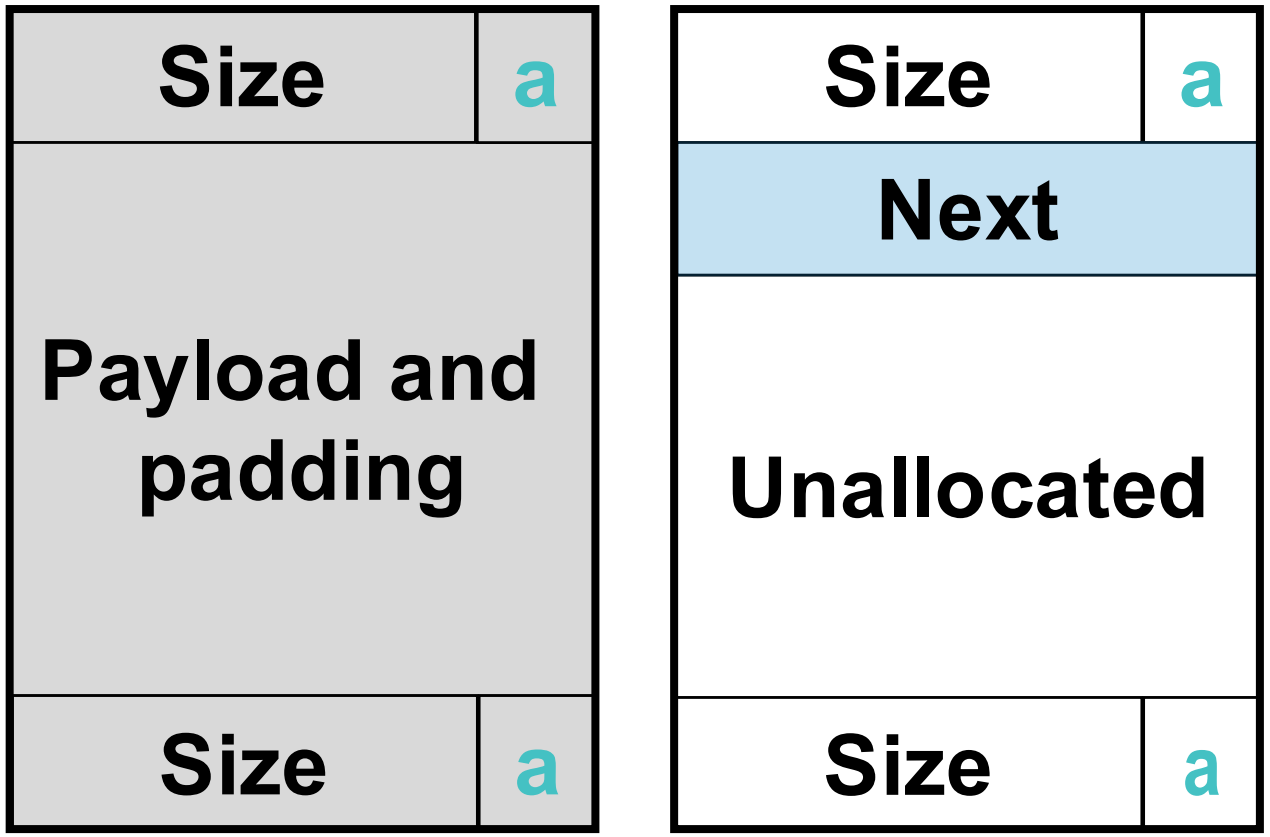


Explicit Lists (Not Final Version)

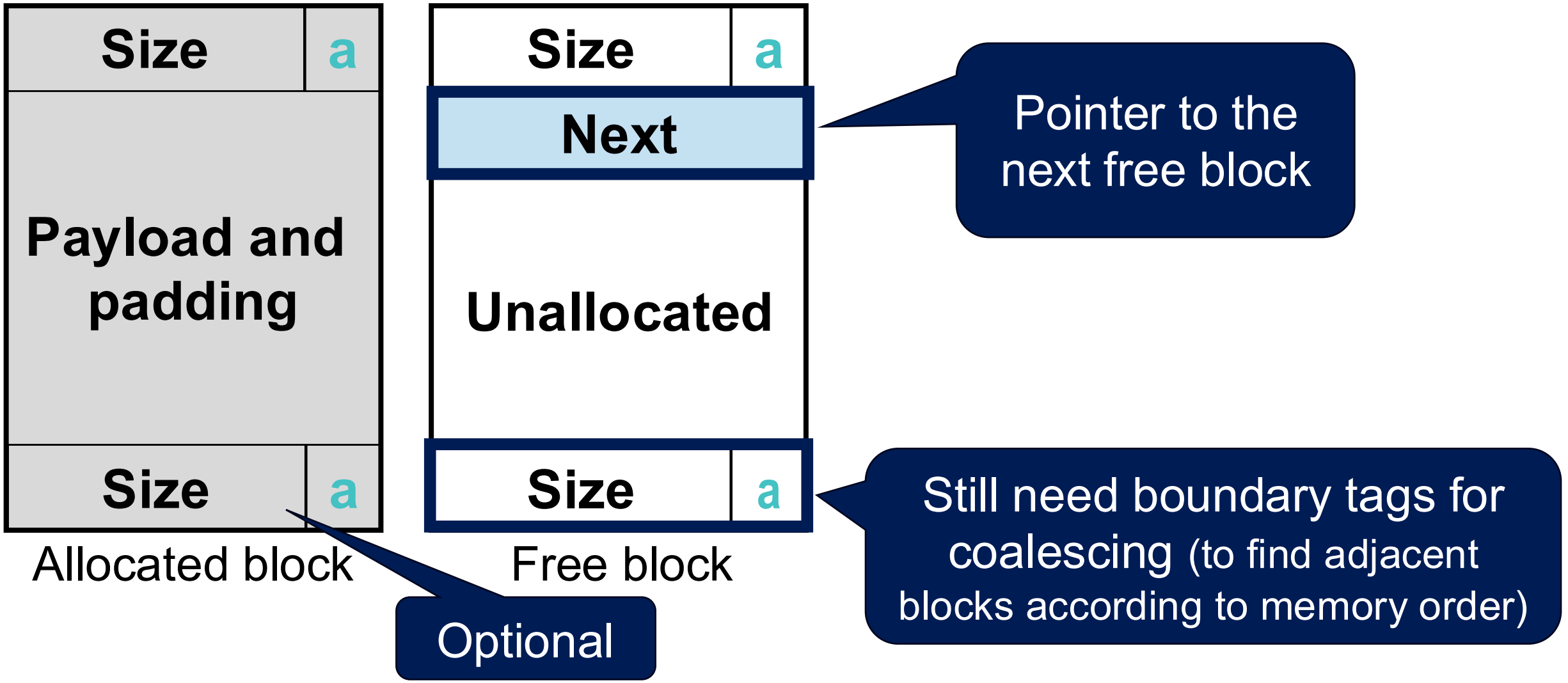




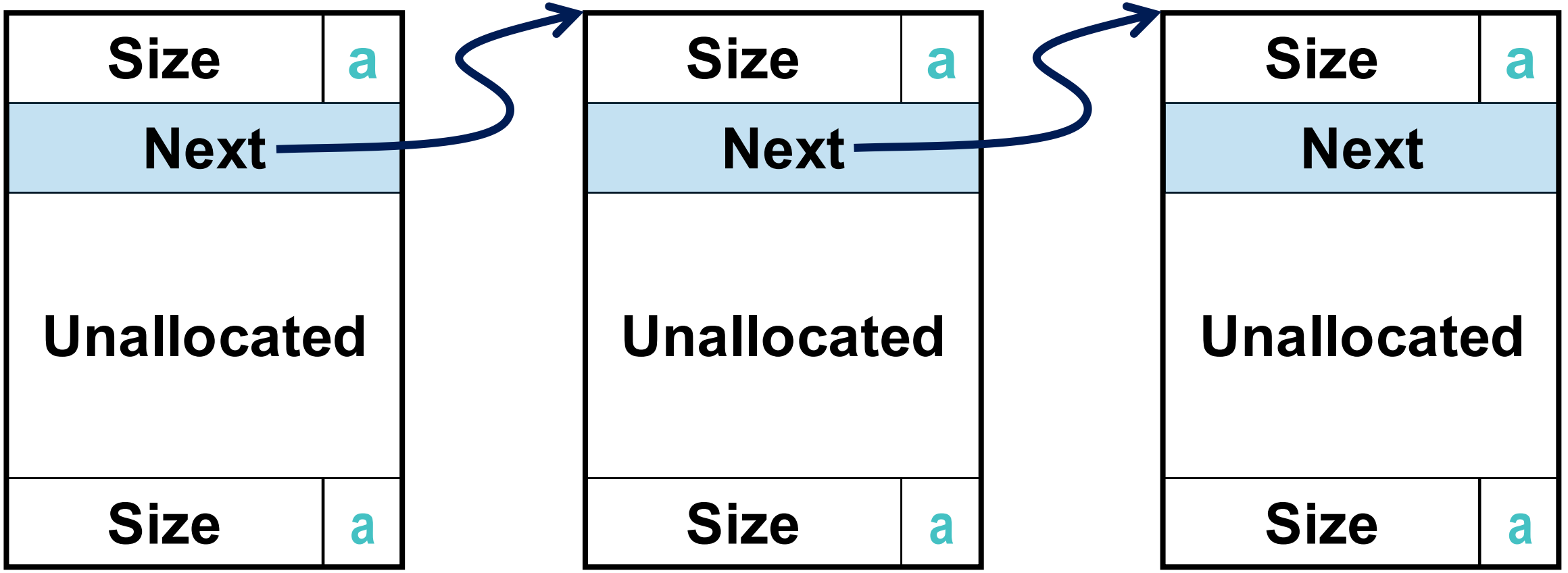
Explicit Lists (Not Final Version)



Explicit Lists (Not Final Version)

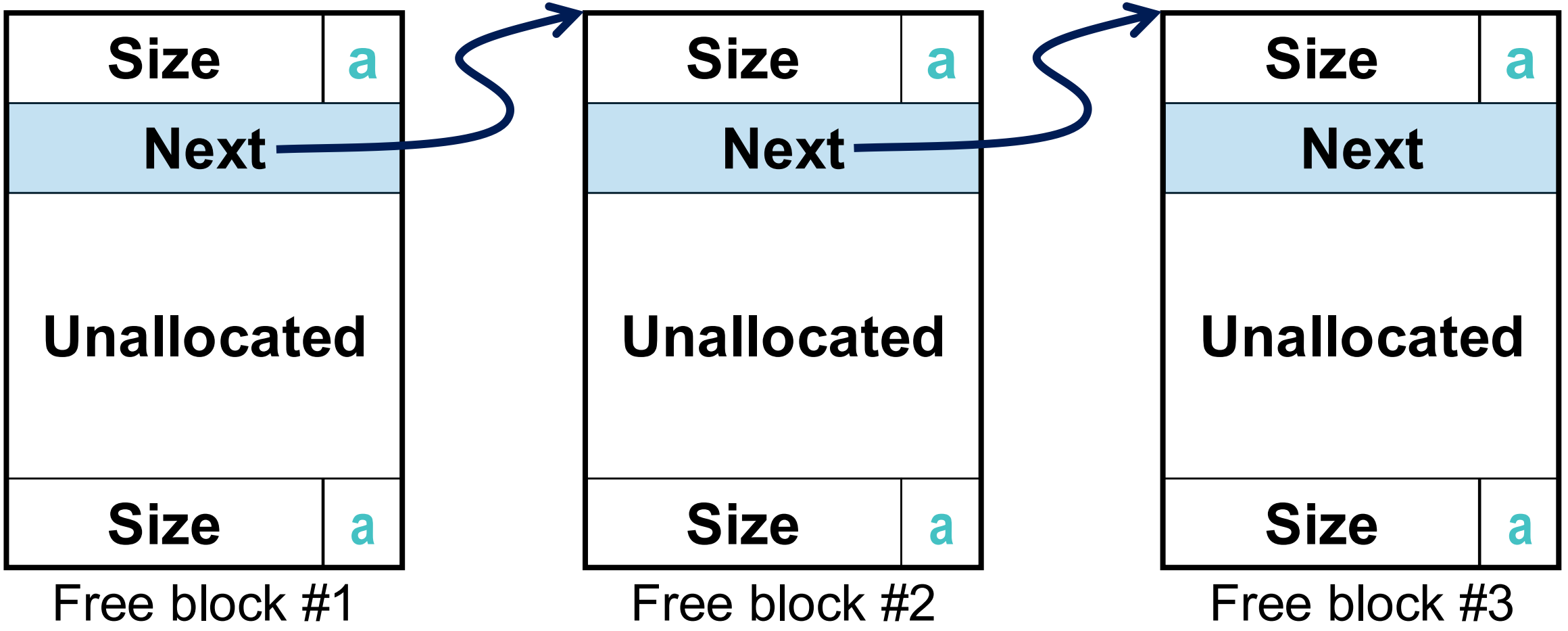


Explicit Lists (Not Final Version)



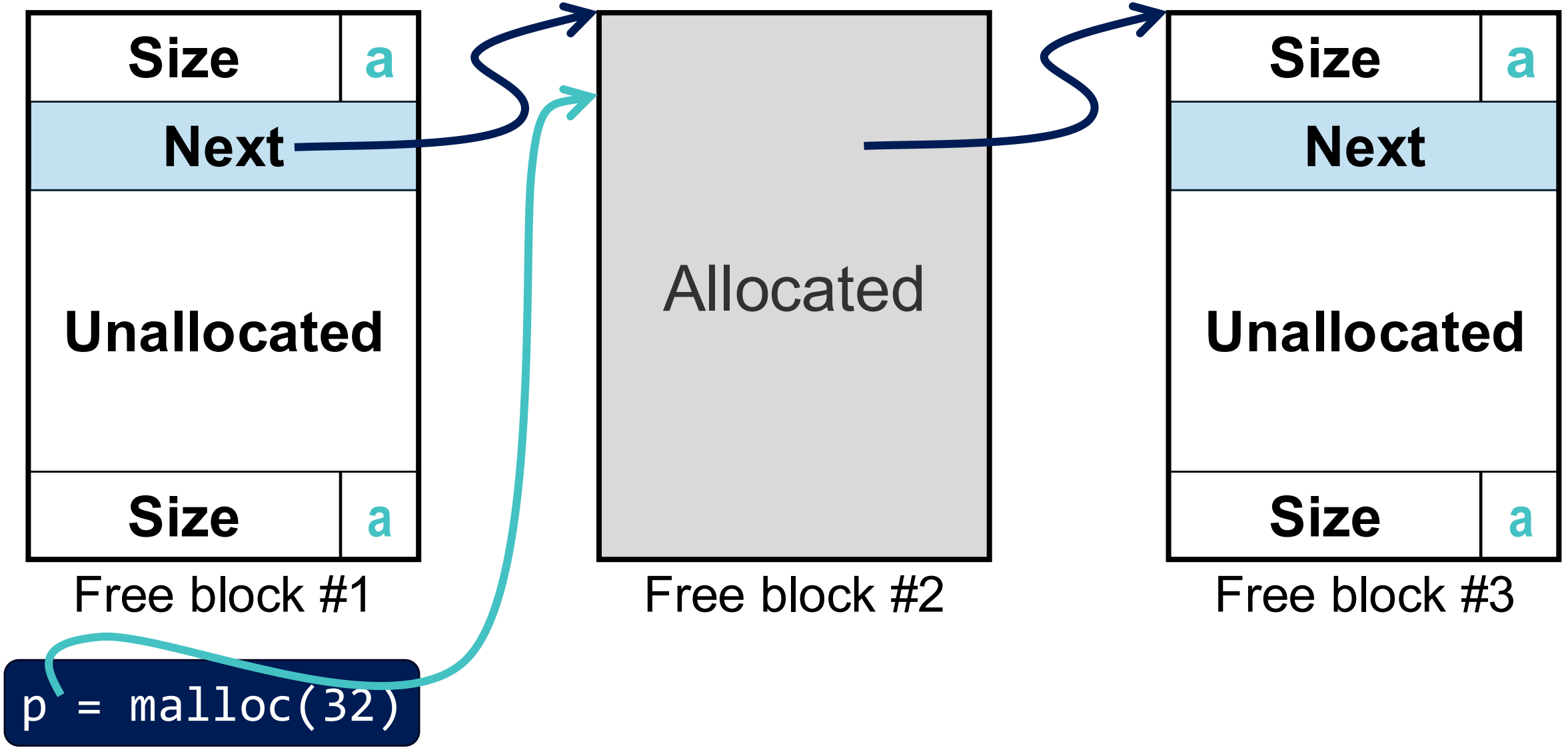
Any problems?

Limitations of the Singly Linked List

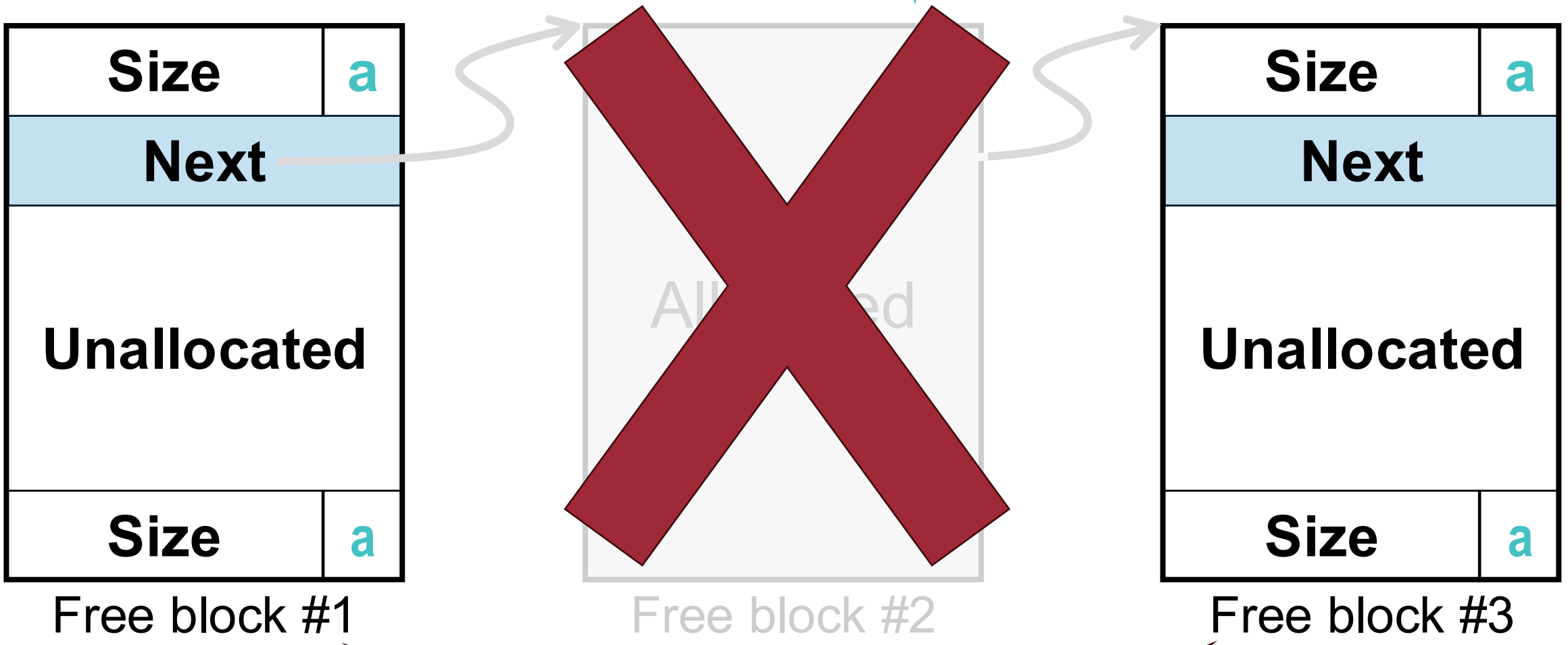



```
p = malloc(32)
```

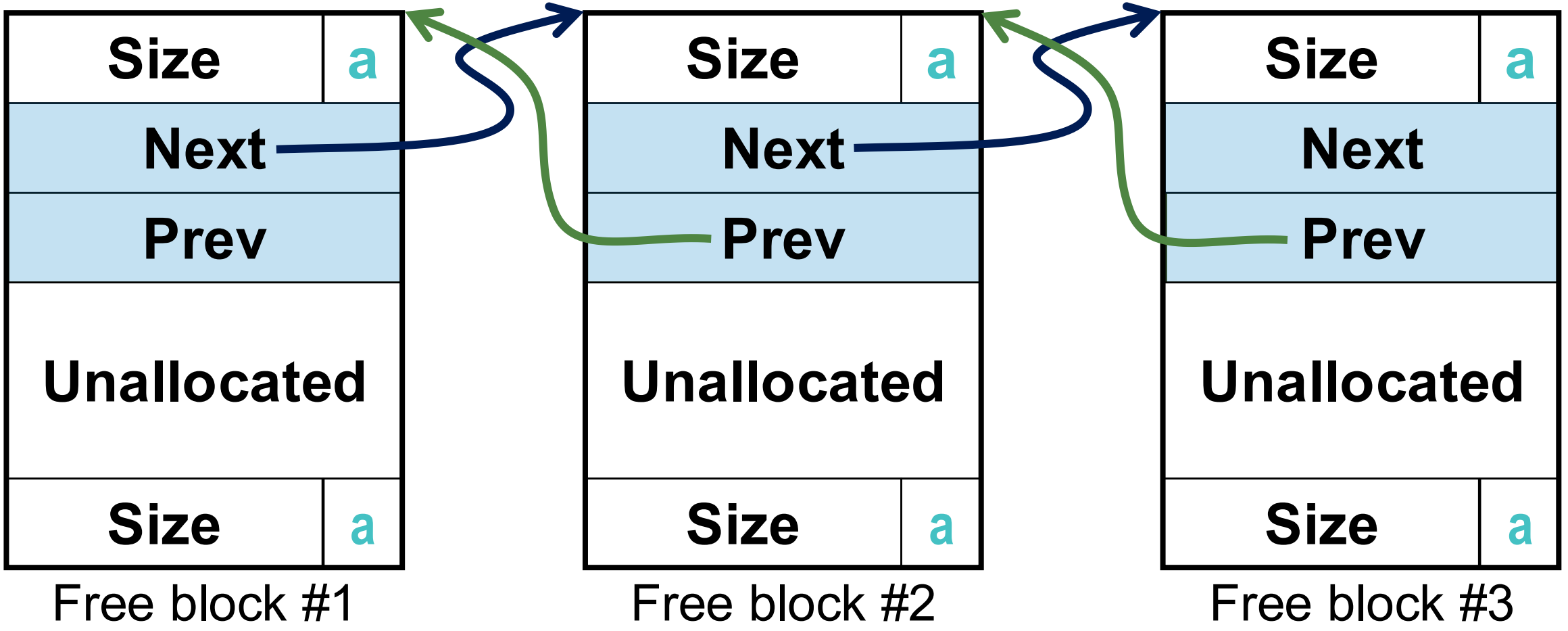
Limitations of the Singly Linked List



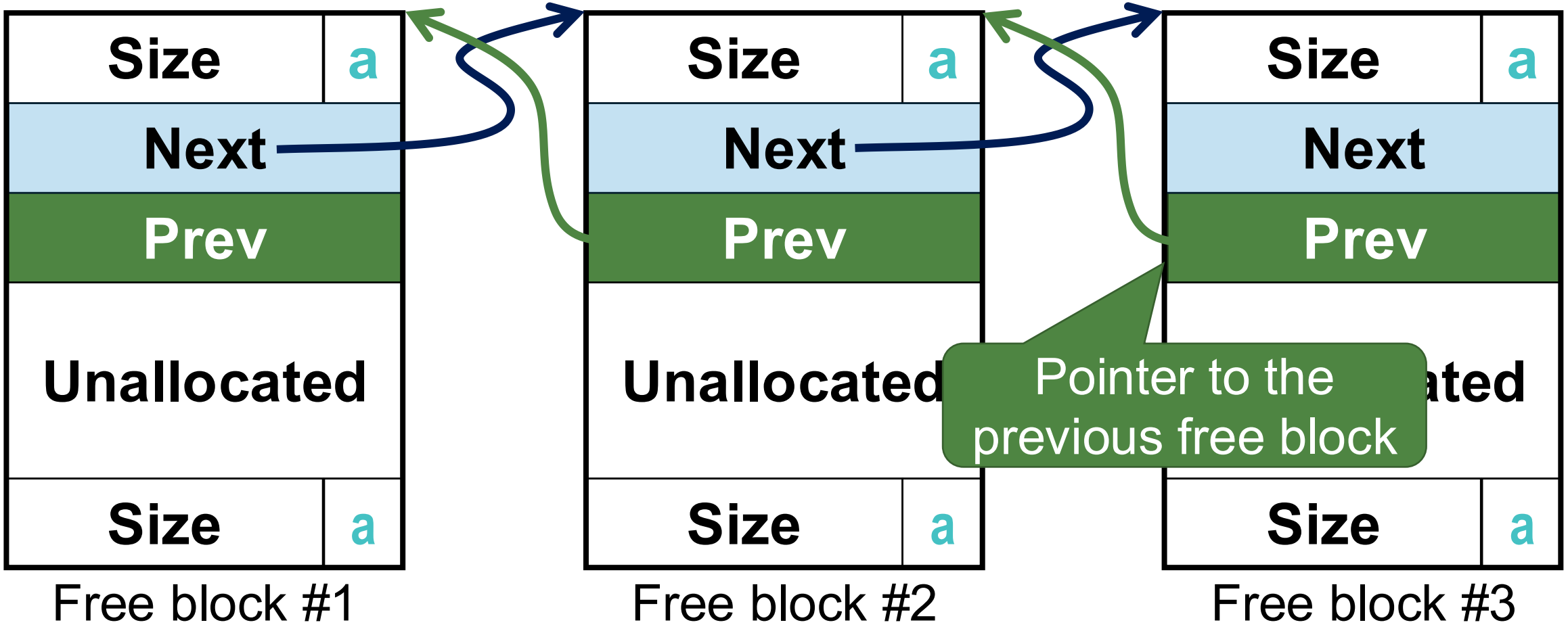
Limitations of the Singly Linked List



Solution: Doubly Linked List



Solution: Doubly Linked List

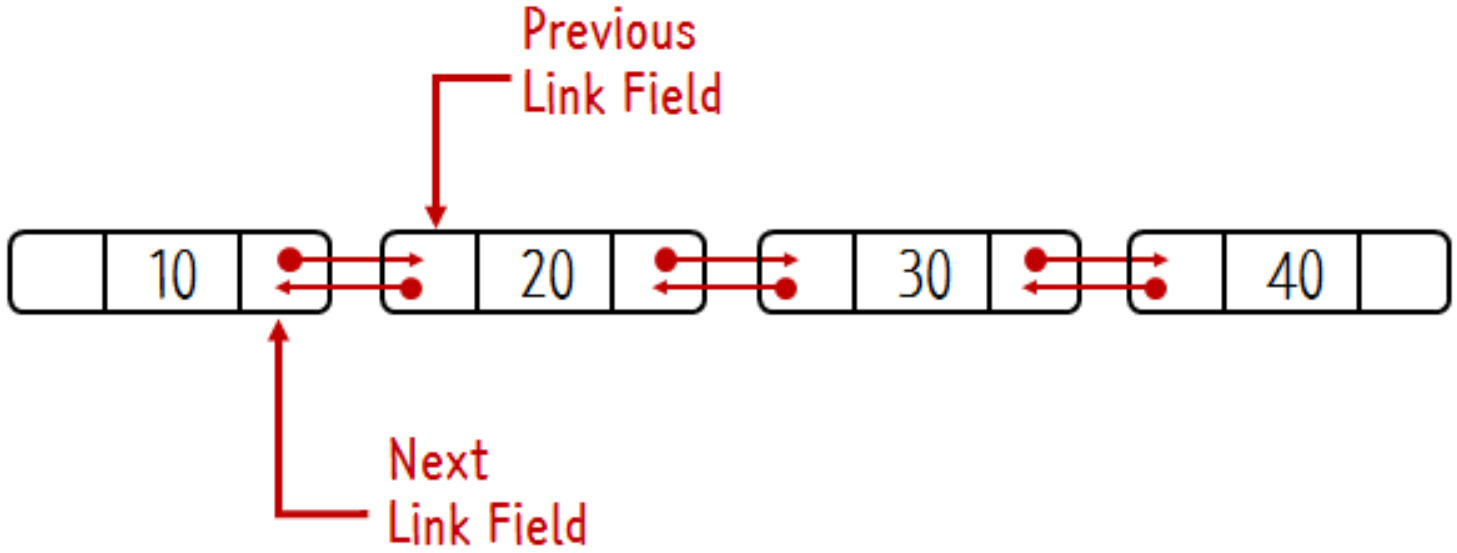


Ref) Doubly Linked List

- Each node contains pointers to **both its previous and next nodes**
  - Allows for efficient traversal of the list in both directions
  - Allows for quick and easy insertion and deletion of nodes

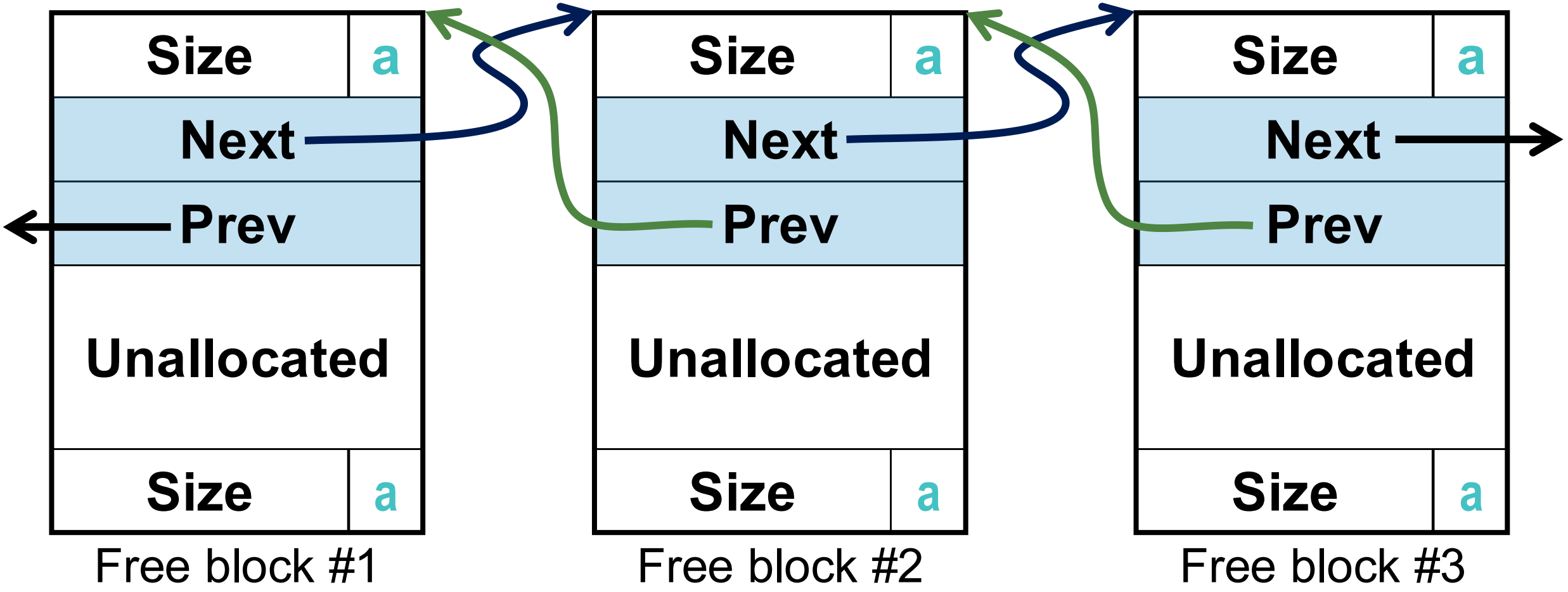
- It will be covered in more detail in CSE221 (Data Structures)

```
typedef struct Block {
    int data;
    struct Block* next;
    struct Block* prev;
} Block;
```



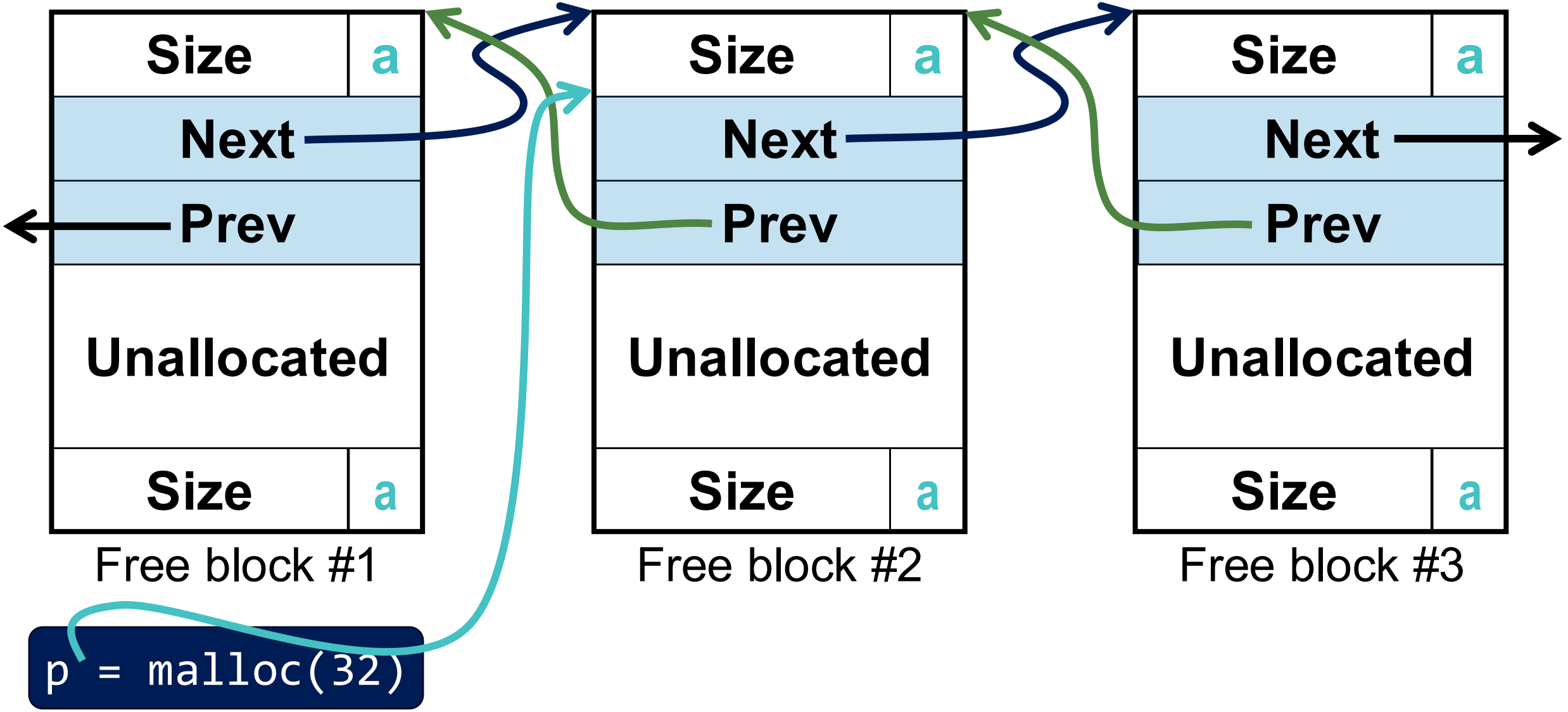
Solution: Doubly Linked List

```
typedef struct Block {
    int data;
    struct Block* next;
    struct Block* prev;
} Block;
```



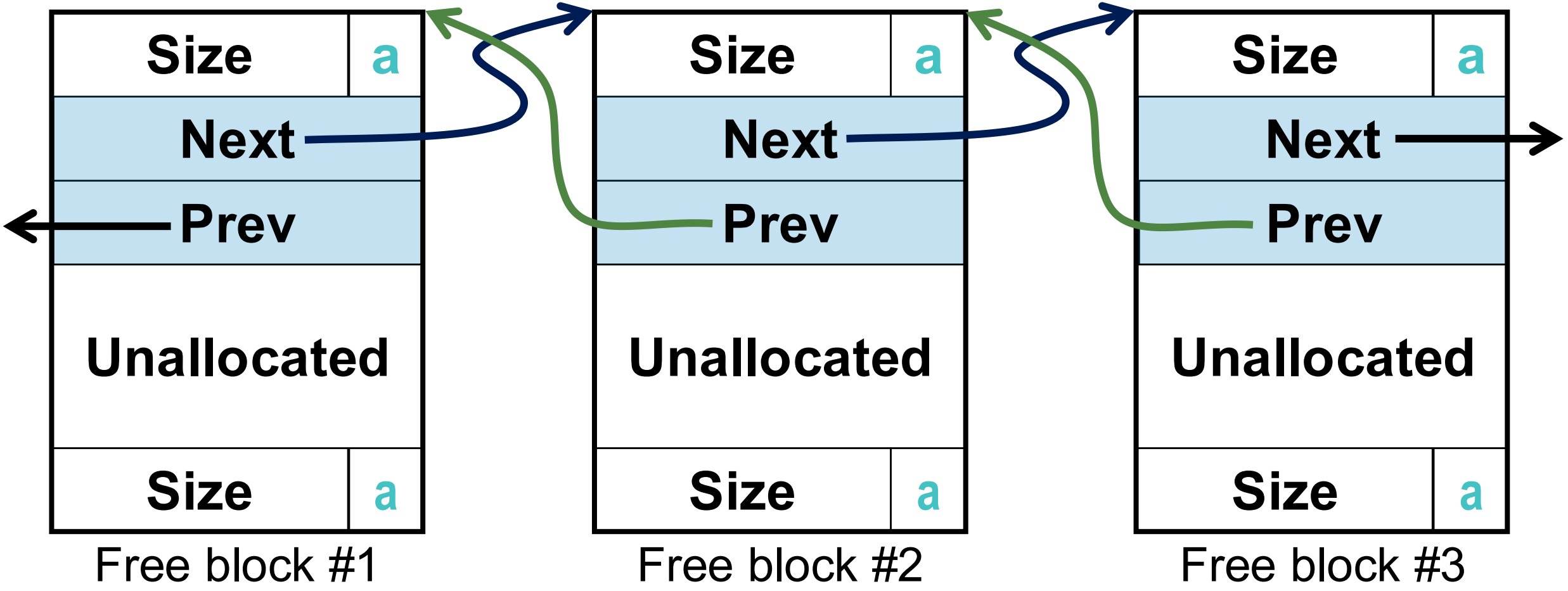
Solution: Doubly Linked List

```
typedef struct Block {
    int data;
    struct Block* next;
    struct Block* prev;
} Block;
```



Solution: Doubly Linked List

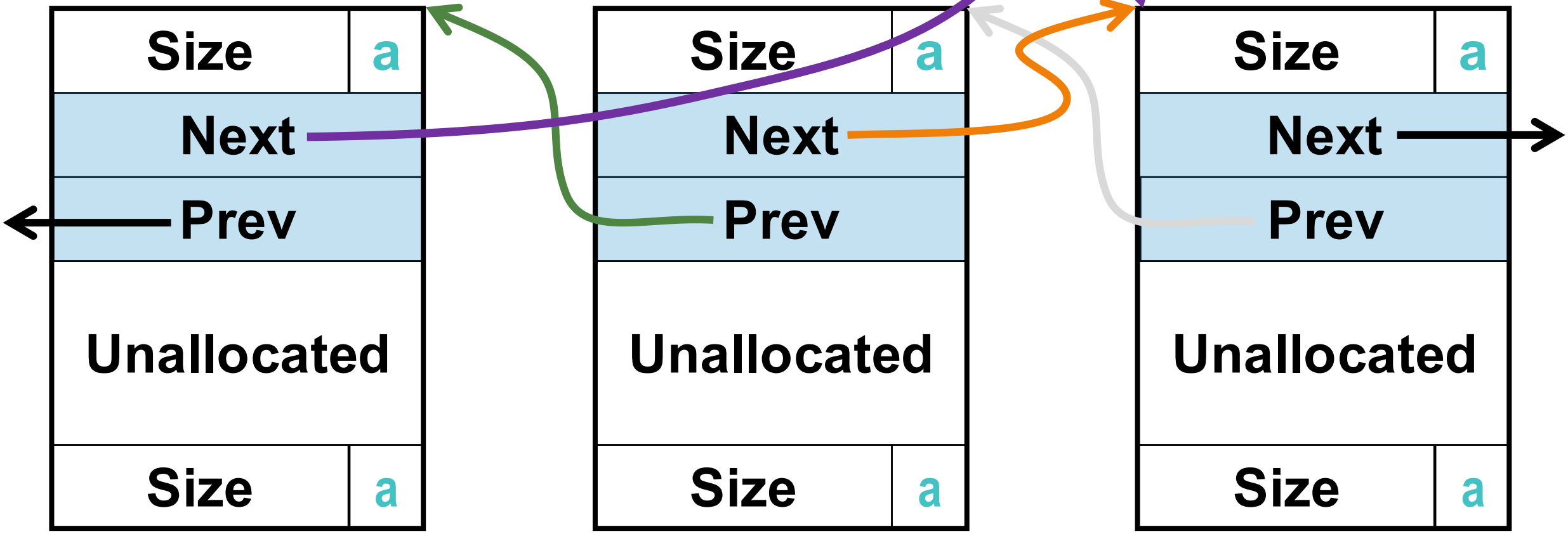
```
typedef struct Block {
    int data;
    struct Block* next;
    struct Block* prev;
} Block;
```




```
removed->prev->next = removed->next;
removed->next->prev = removed->prev;
```

Solution: Doubly Linked List

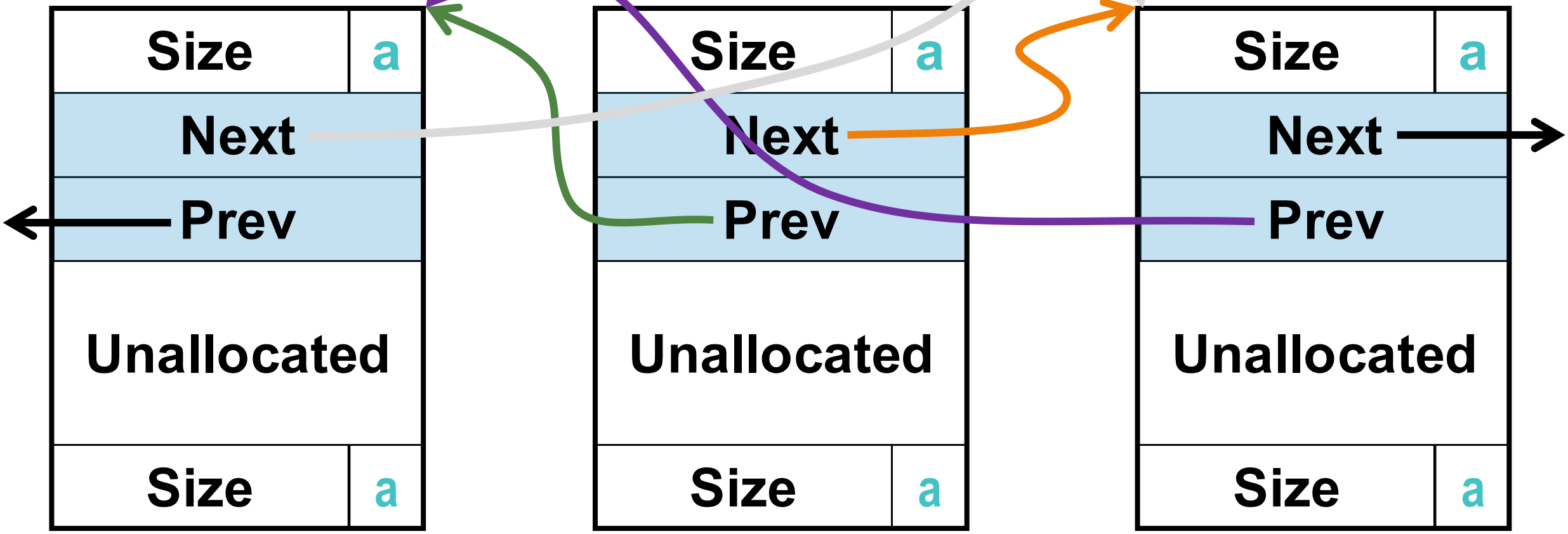
```
typedef struct Block {
    int data;
    struct Block* next;
    struct Block* prev;
} Block;
```




```
removed->prev->next = removed->next;
removed->next->prev = removed->prev;
```

Solution: Doubly Linked List

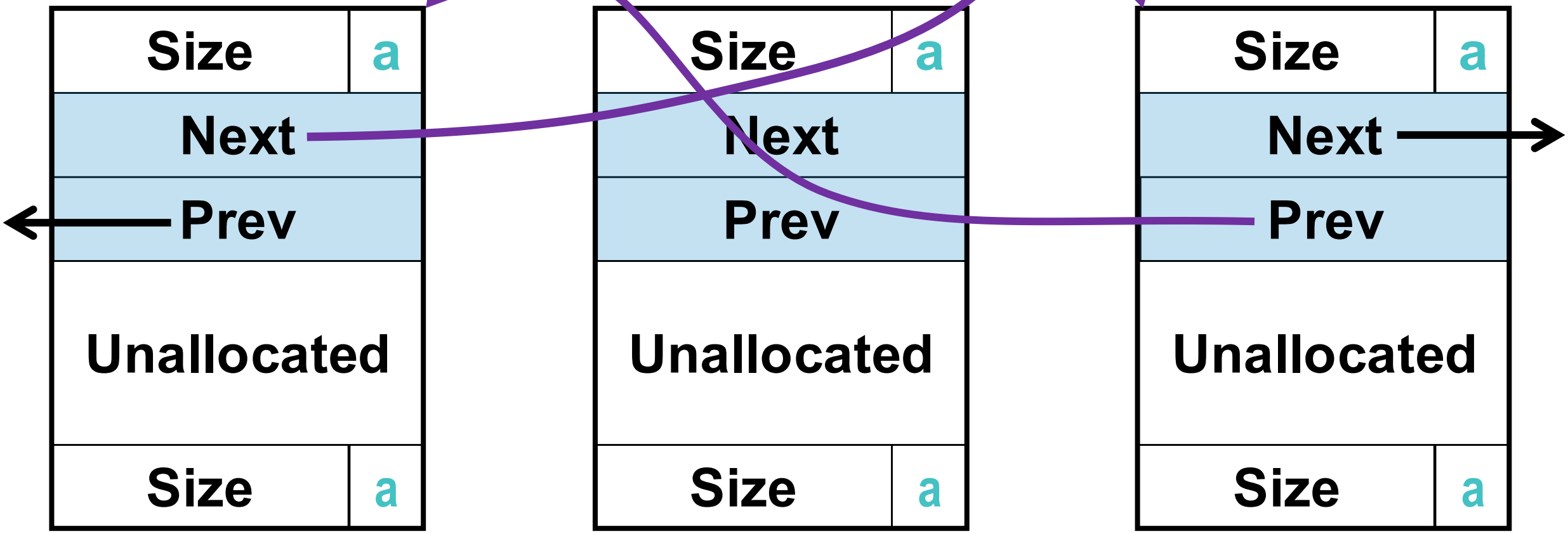
```
typedef struct Block {
    int data;
    struct Block* next;
    struct Block* prev;
} Block;
```




```
removed->prev->next = removed->next;
removed->next->prev = removed->prev;
```

Solution: Doubly Linked List

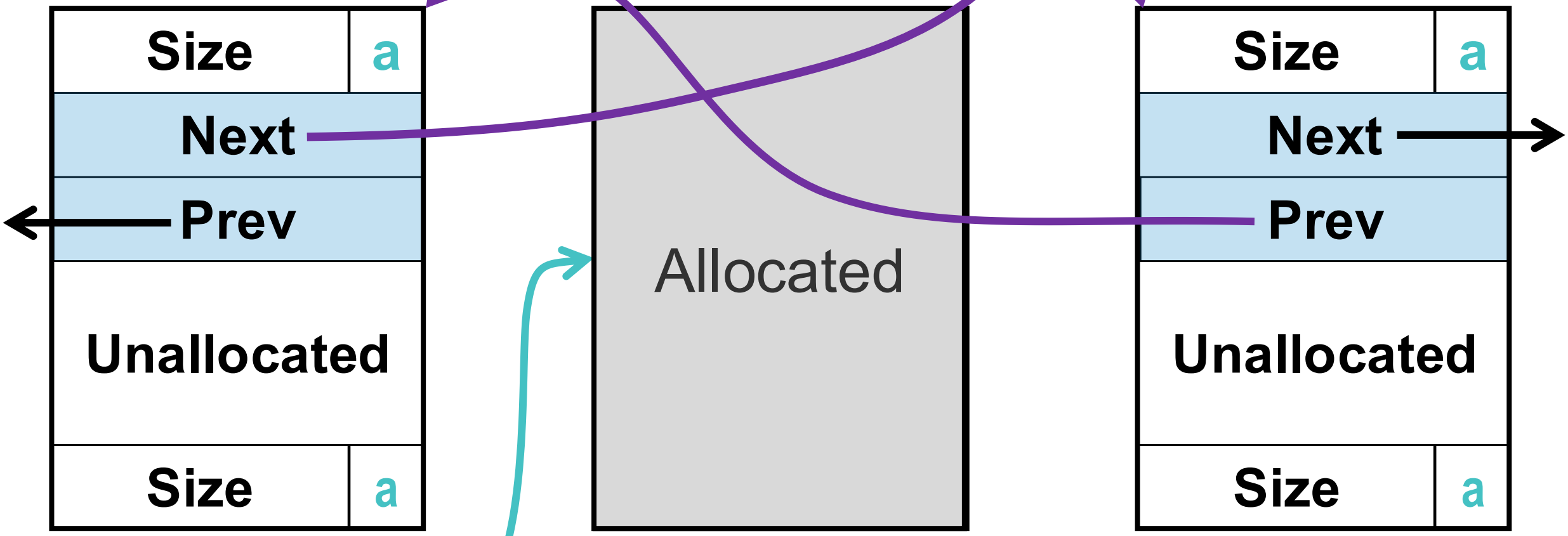
```
typedef struct Block {
    int data;
    struct Block* next;
    struct Block* prev;
} Block;
```




```
removed->prev->next = removed->next;
removed->next->prev = removed->prev;
```

Solution: Doubly Linked List

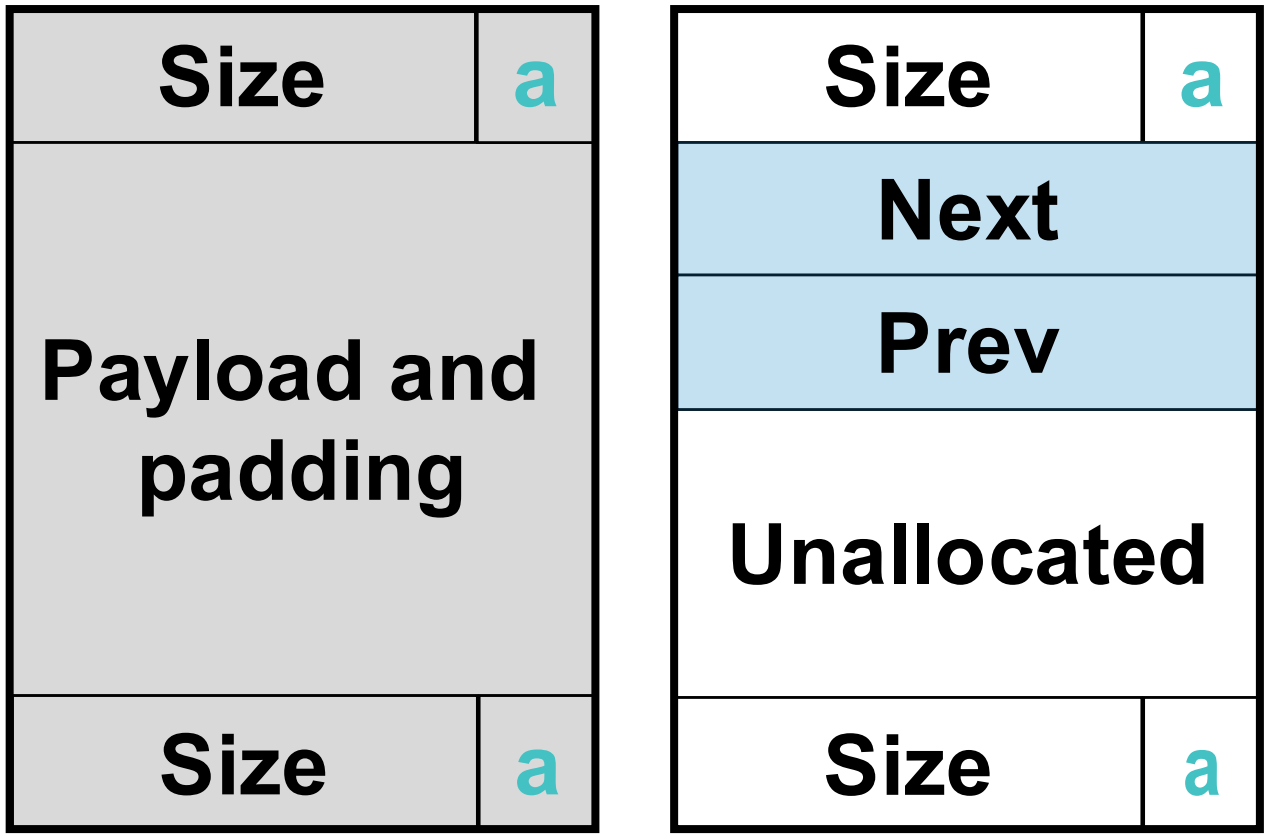
```
typedef struct Block {
    int data;
    struct Block* next;
    struct Block* prev;
} Block;
```




```
p = malloc(32)
```

Node deletion can be done in *constant time* without traversing the entire list, which would take linear time!

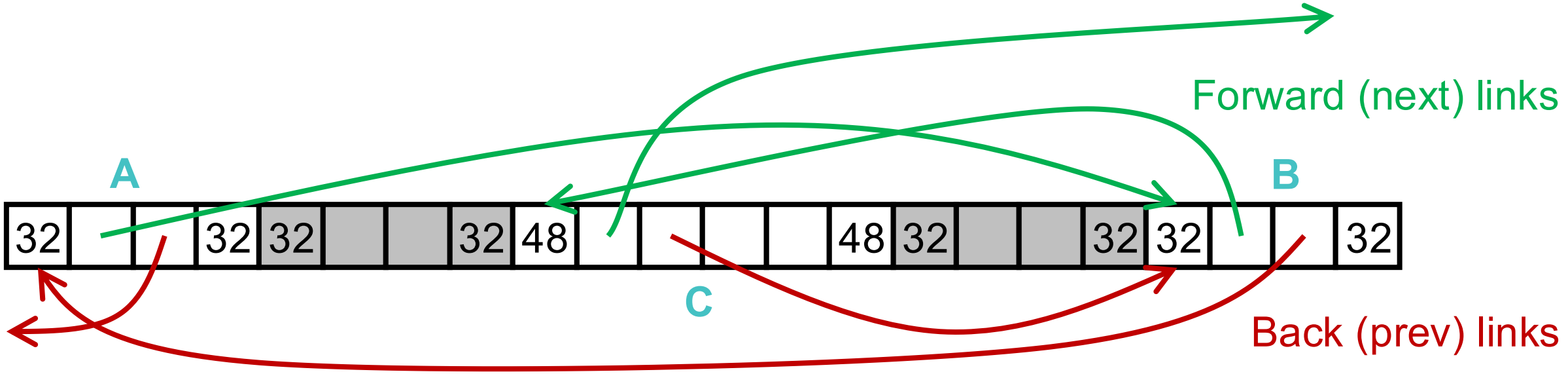
Explicit Lists (Final Version)



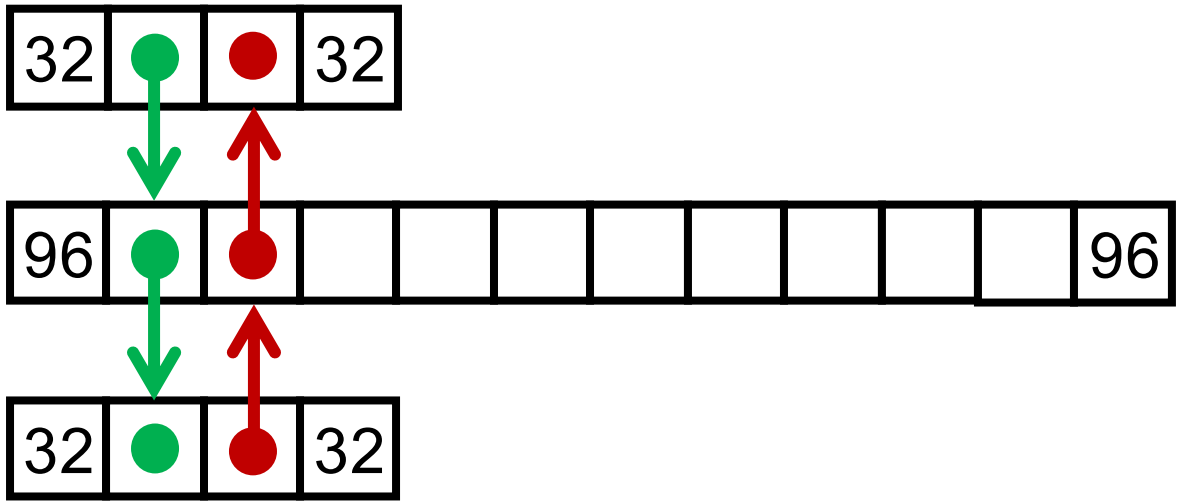
Overview: Explicit Lists

- Logically:

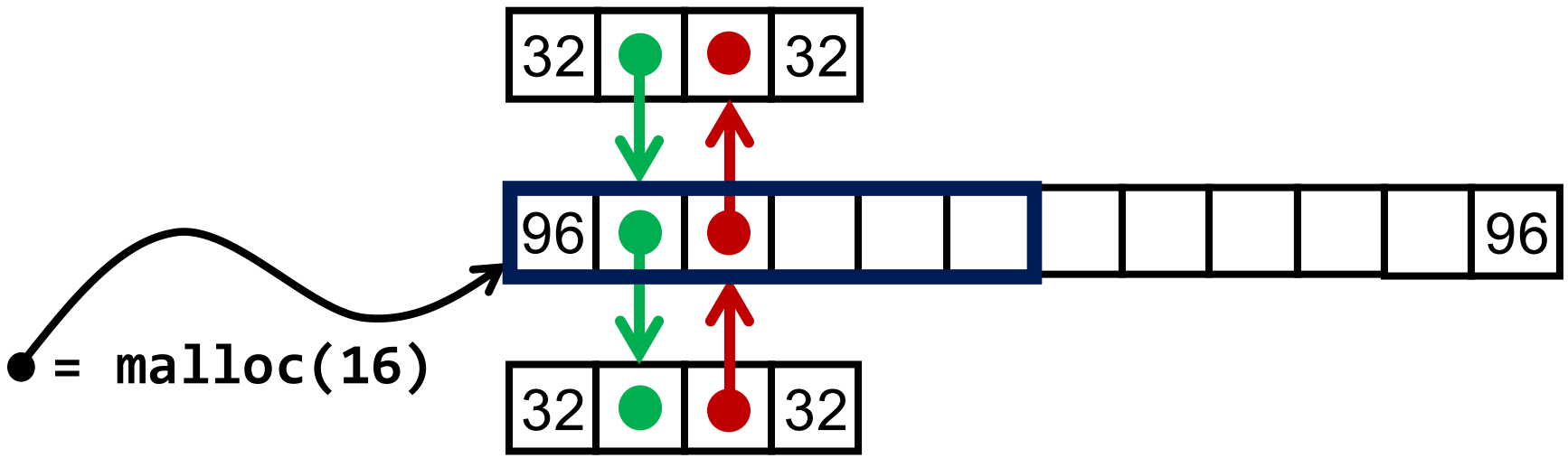
- Physically: blocks can be in any order



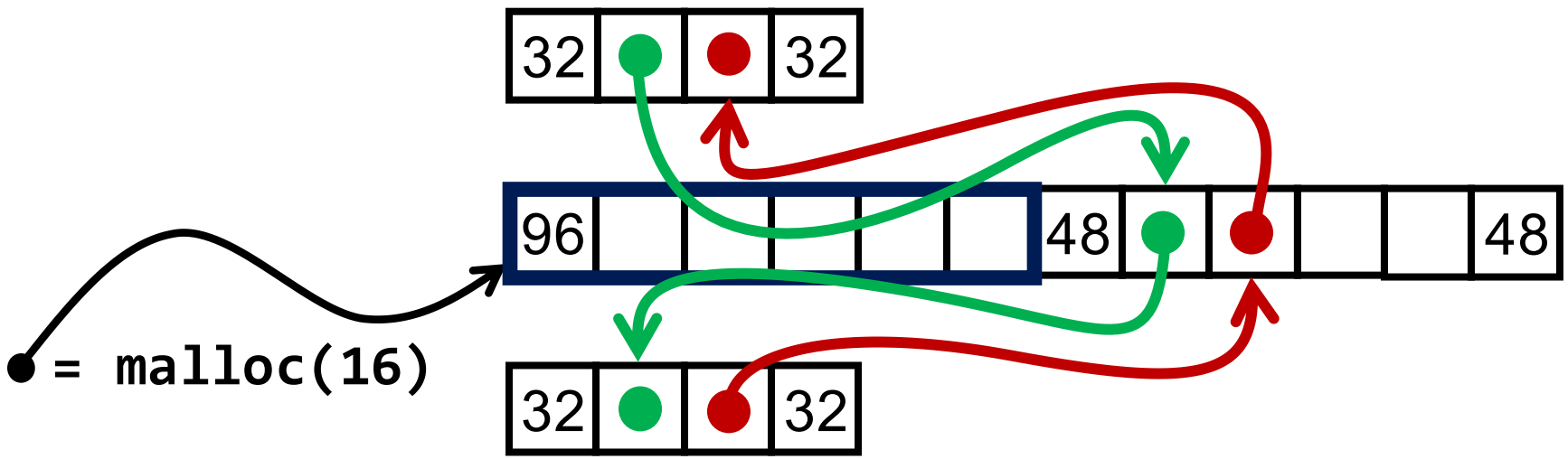
Allocation (+Splitting) from Explicit Lists



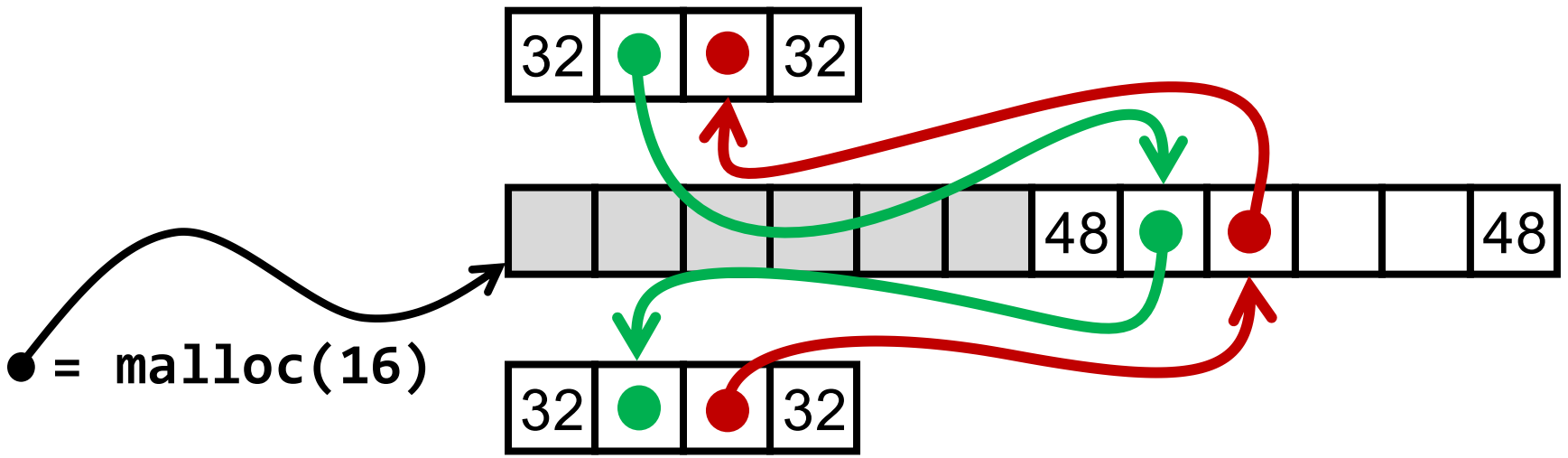
Allocation (+Splitting) from Explicit Lists



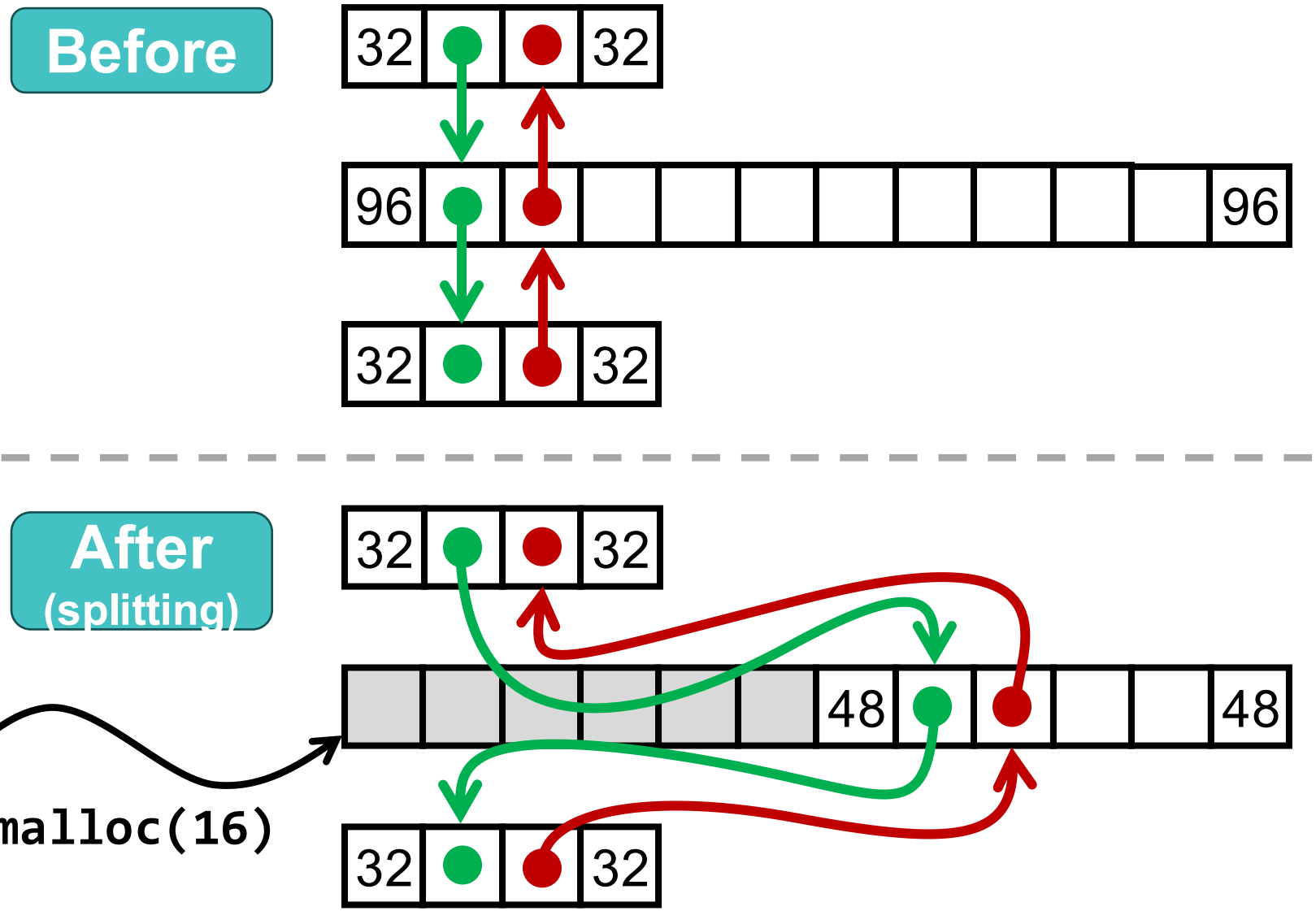
Allocation (+Splitting) from Explicit Lists



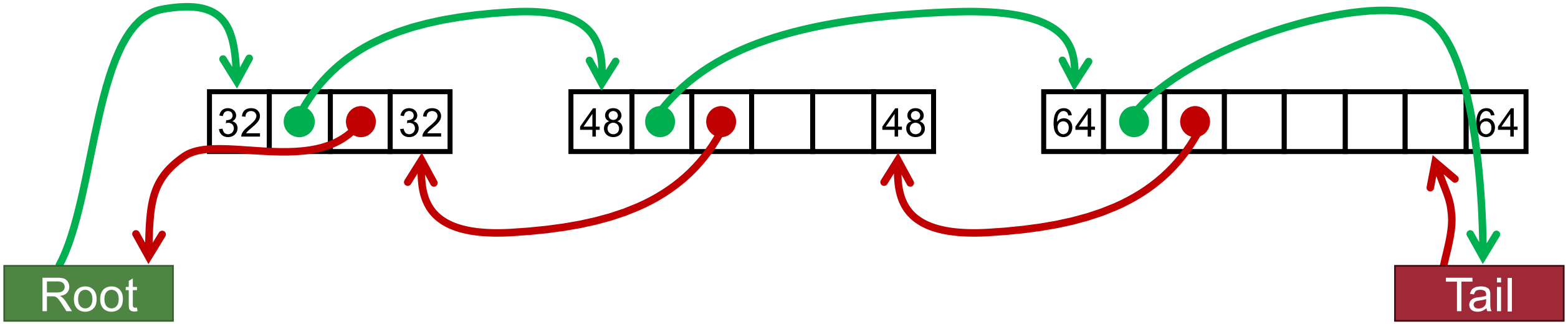
Allocation (+Splitting) from Explicit Lists



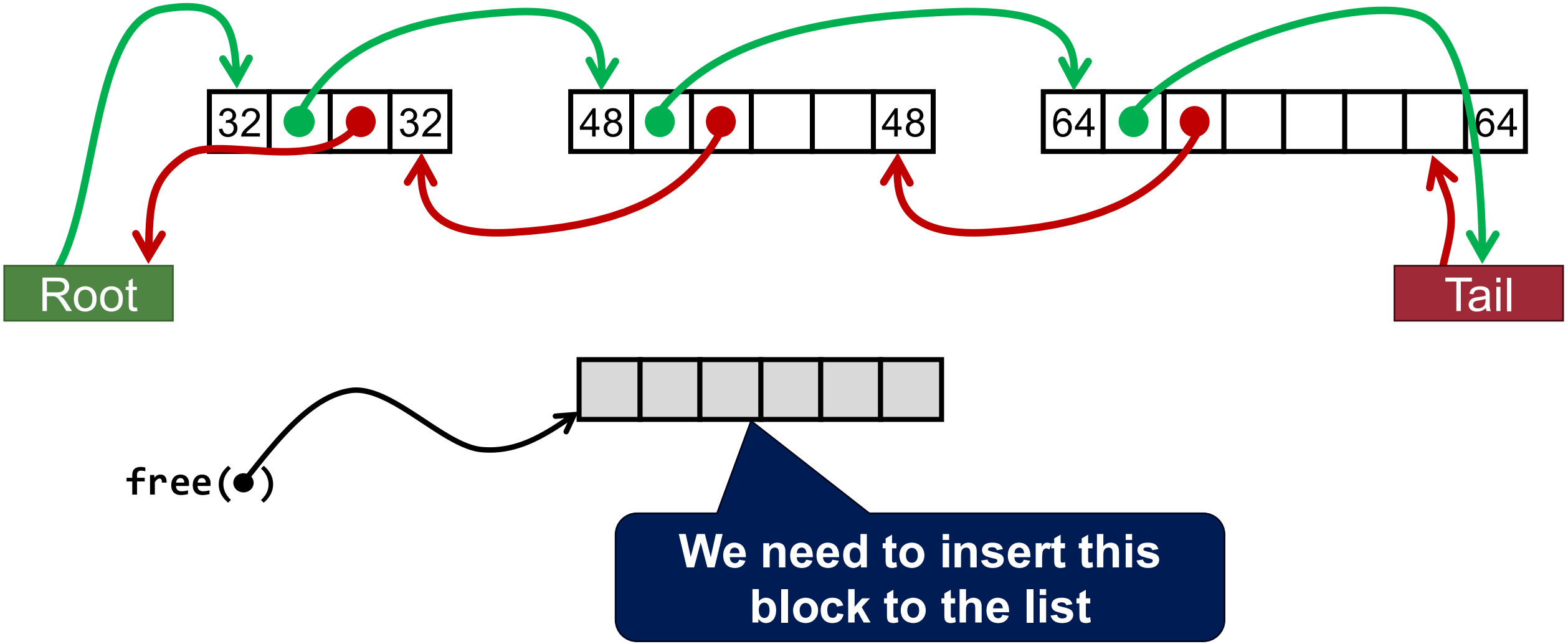
Allocation (+Splitting) Summary



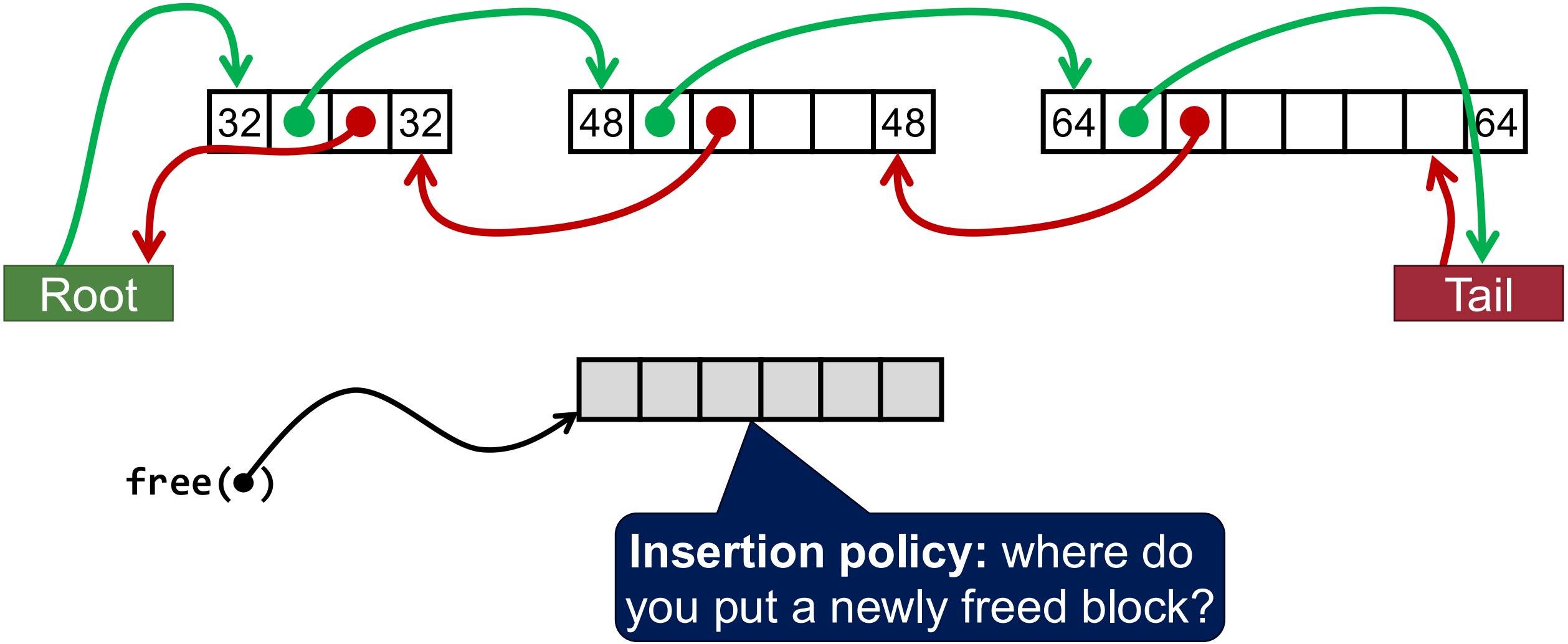
How About Free?



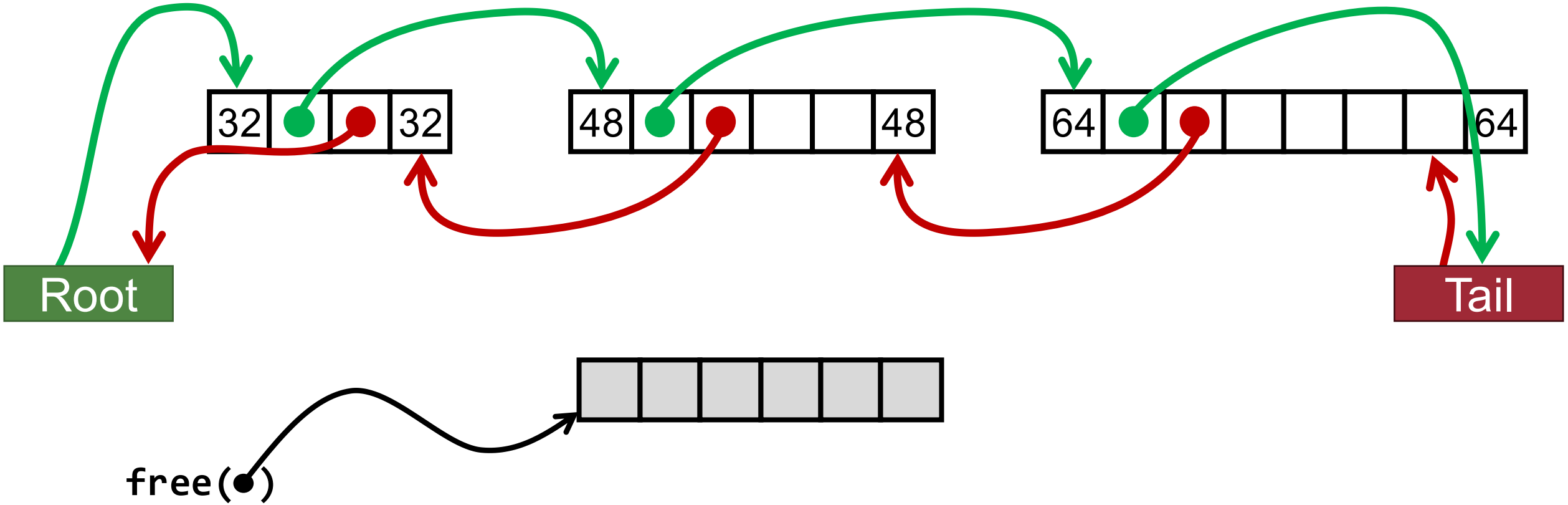
How About Free?



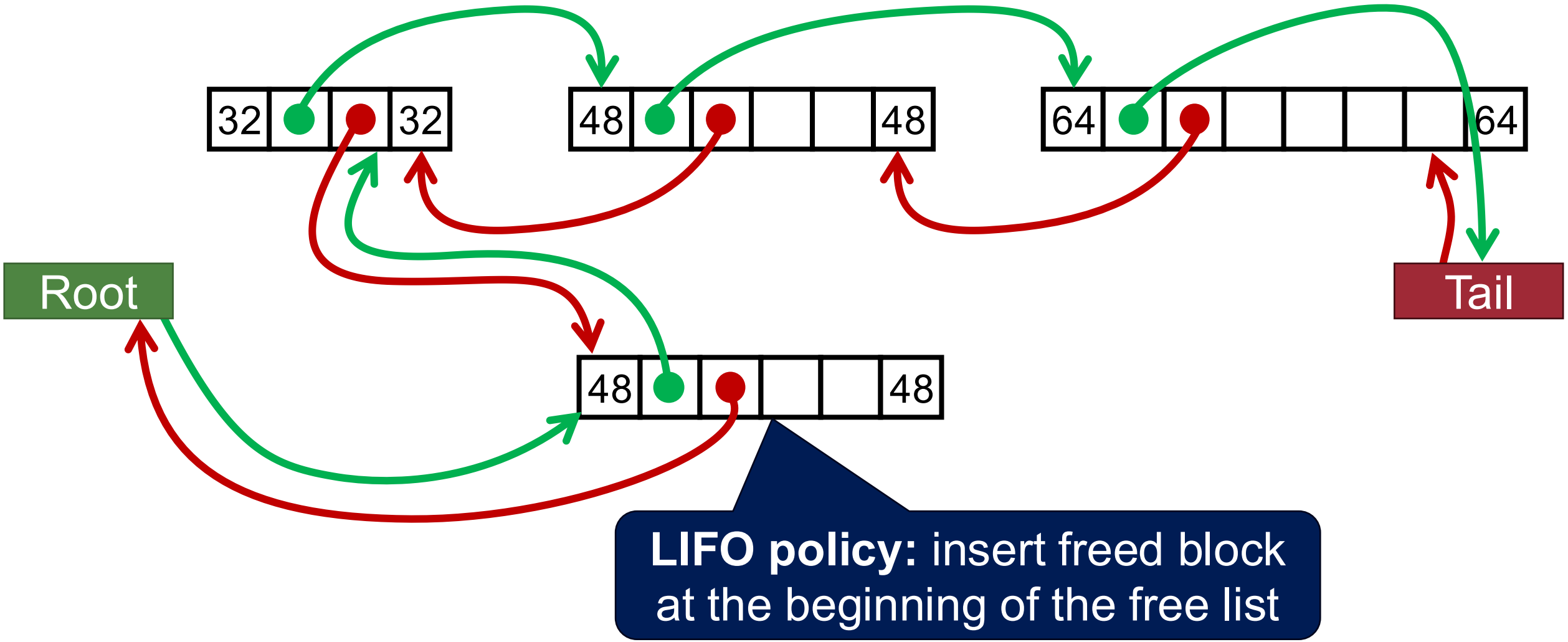
Insertion Policy



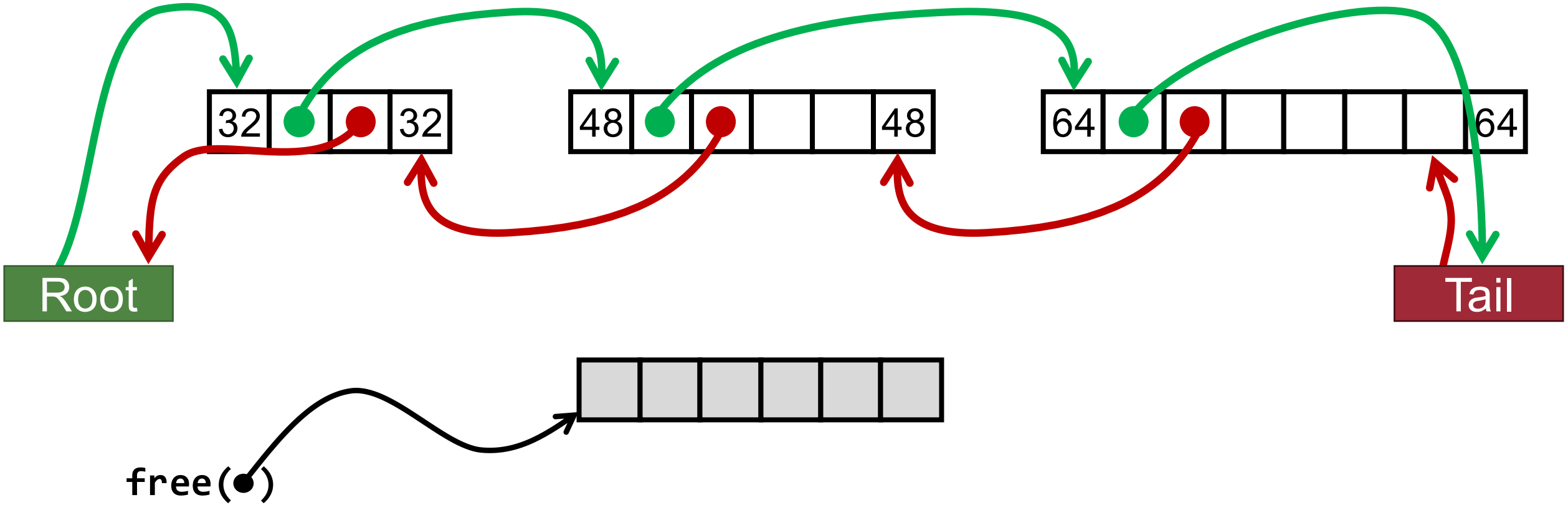
1. LIFO (Last-In-First-Out) Policy



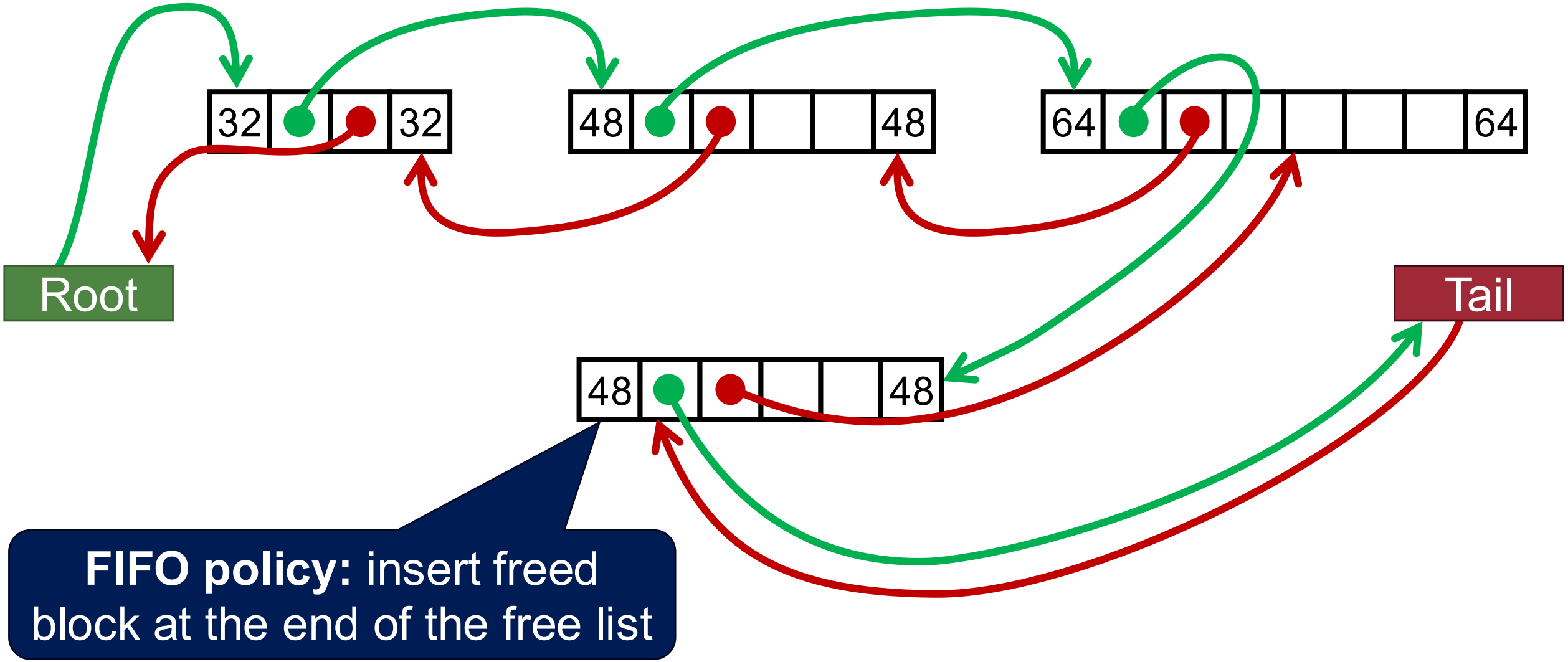
1. LIFO (Last-In-First-Out) Policy



2. FIFO (First-In-First-Out) Policy



2. FIFO (First-In-First-Out) Policy



Insertion Policy Summary

- **Unordered**
  - **LIFO policy:** Insert freed block at the beginning of the free list
  - **FIFO policy:** Insert freed block at the end of the free list
  - ***Pro:*** simple and constant time
  - ***Con:*** studies suggest fragmentation is worse than address ordered

Insertion Policy Summary

- **Unordered**
  - **LIFO policy:** Insert freed block at the beginning of the free list
  - **FIFO policy:** Insert freed block at the end 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:

$$addr(prev) < addr(curr) < addr(next)$$

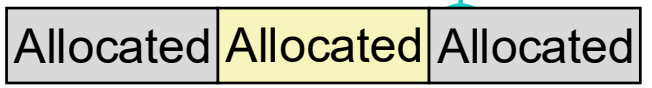
  - ***Con:*** requires search
  - ***Pro:*** studies suggest fragmentation is lower than LIFO/FIFO

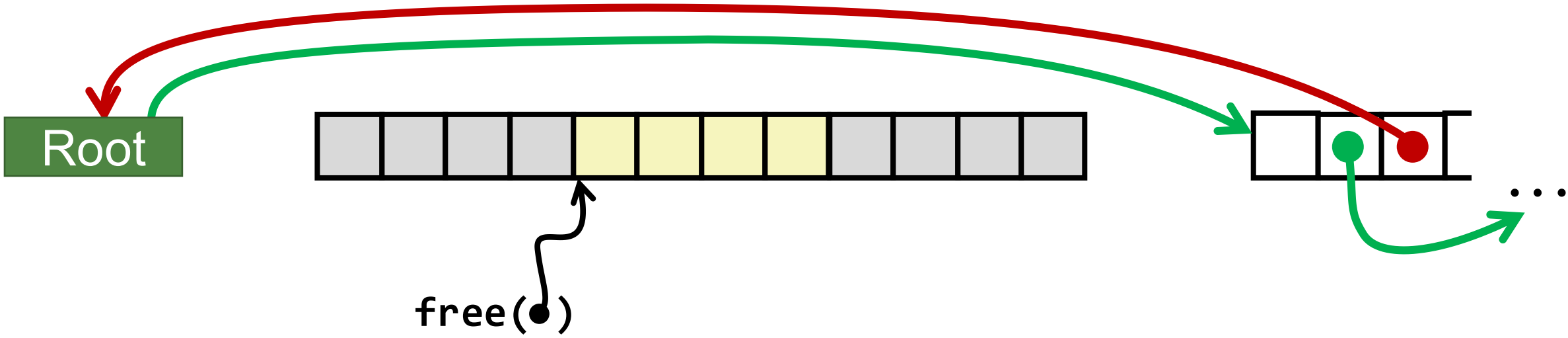
Free (+Coalescing) from Explicit Lists

Allocated block being freed

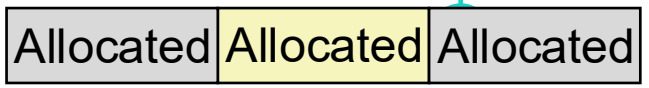
Case 1	Case 2	Case 3	Case 4
Allocated	Allocated	Free	Free
Allocated	Allocated	Allocated	Allocated
Allocated	Free	Allocated	Free

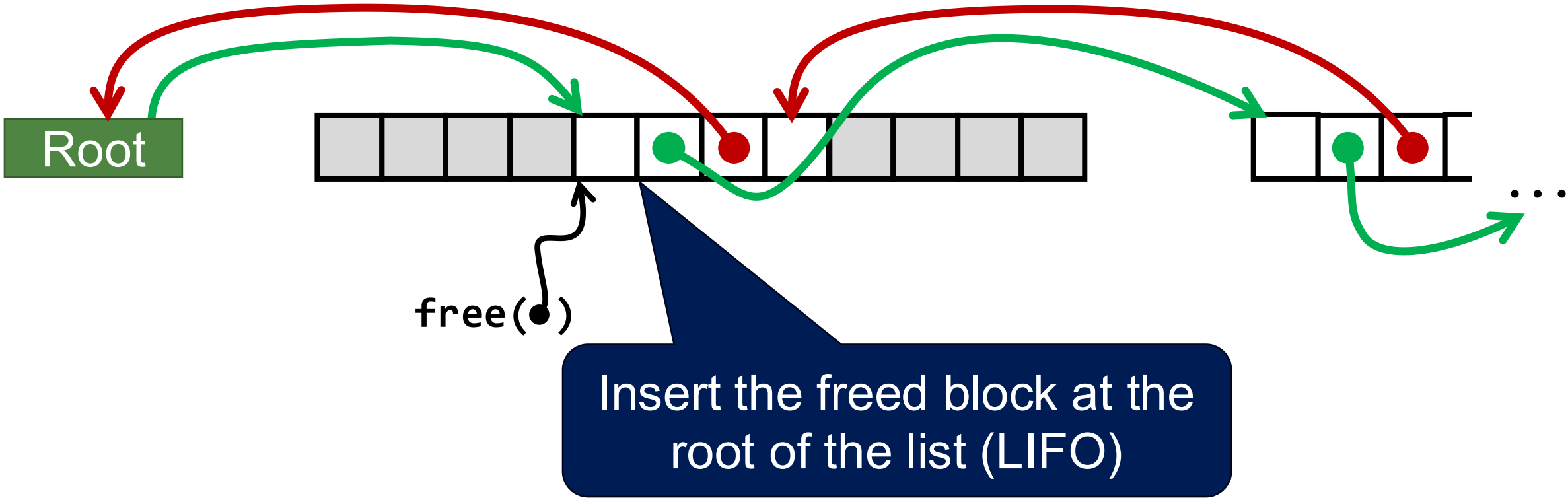
Freeing with LIFO Policy (Case 1)



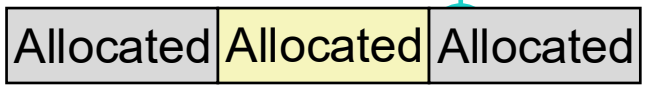


Freeing with LIFO Policy (Case 1)

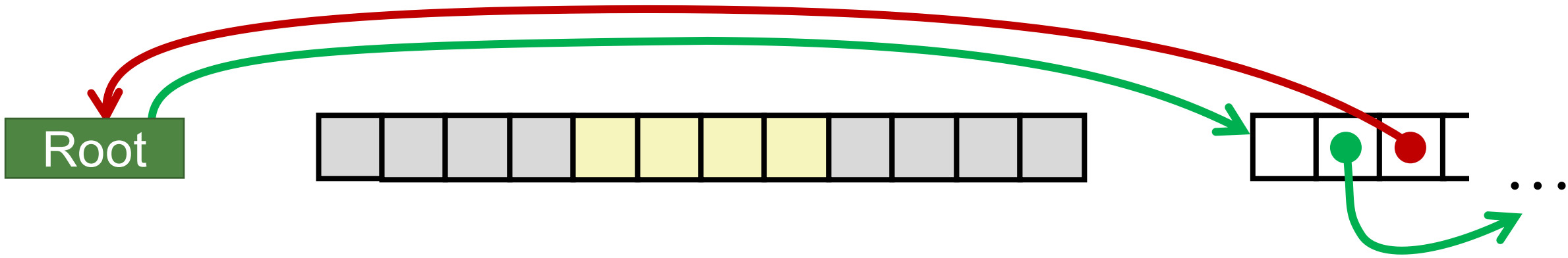




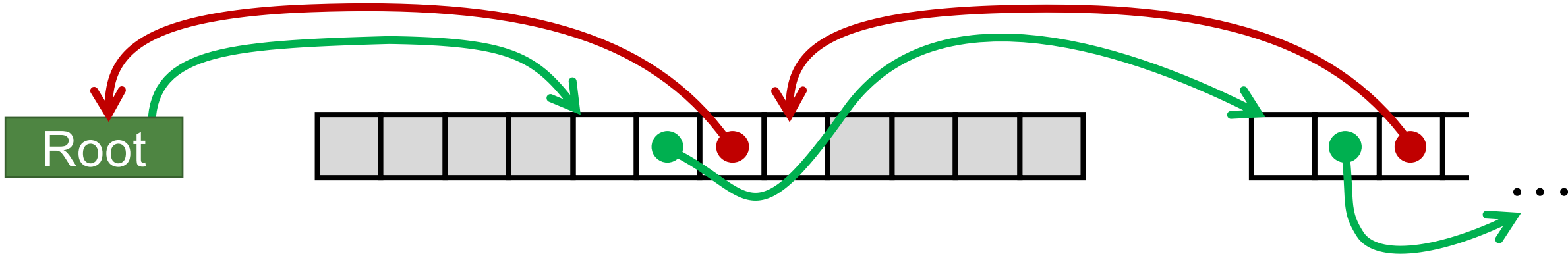
Case 1 Summary



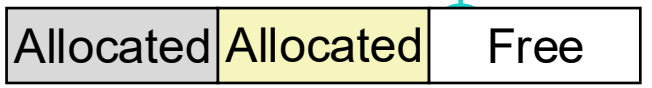
Before

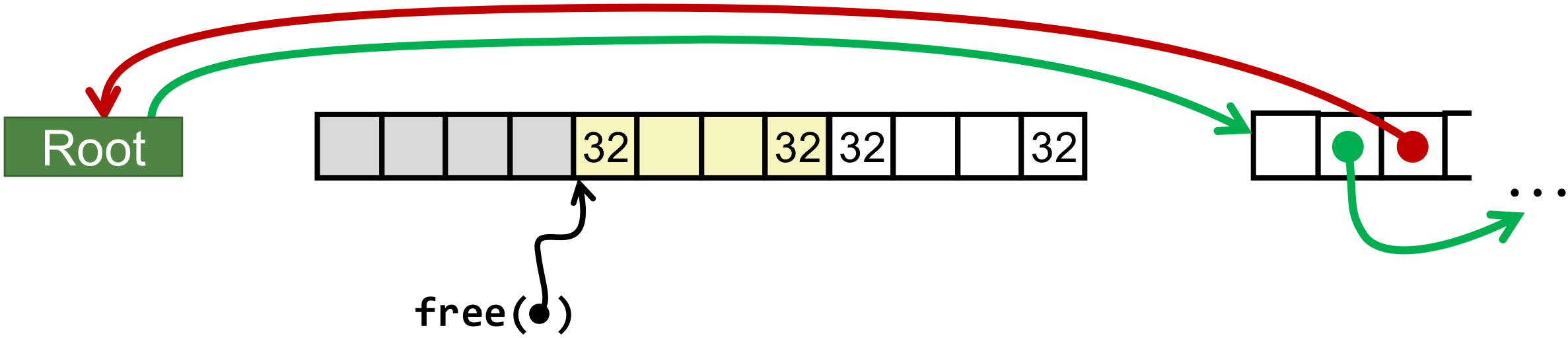


After Insert the freed block at the root of the list (LIFO)

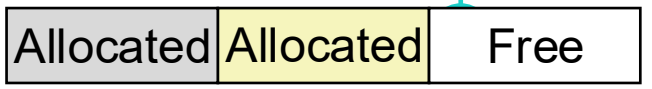


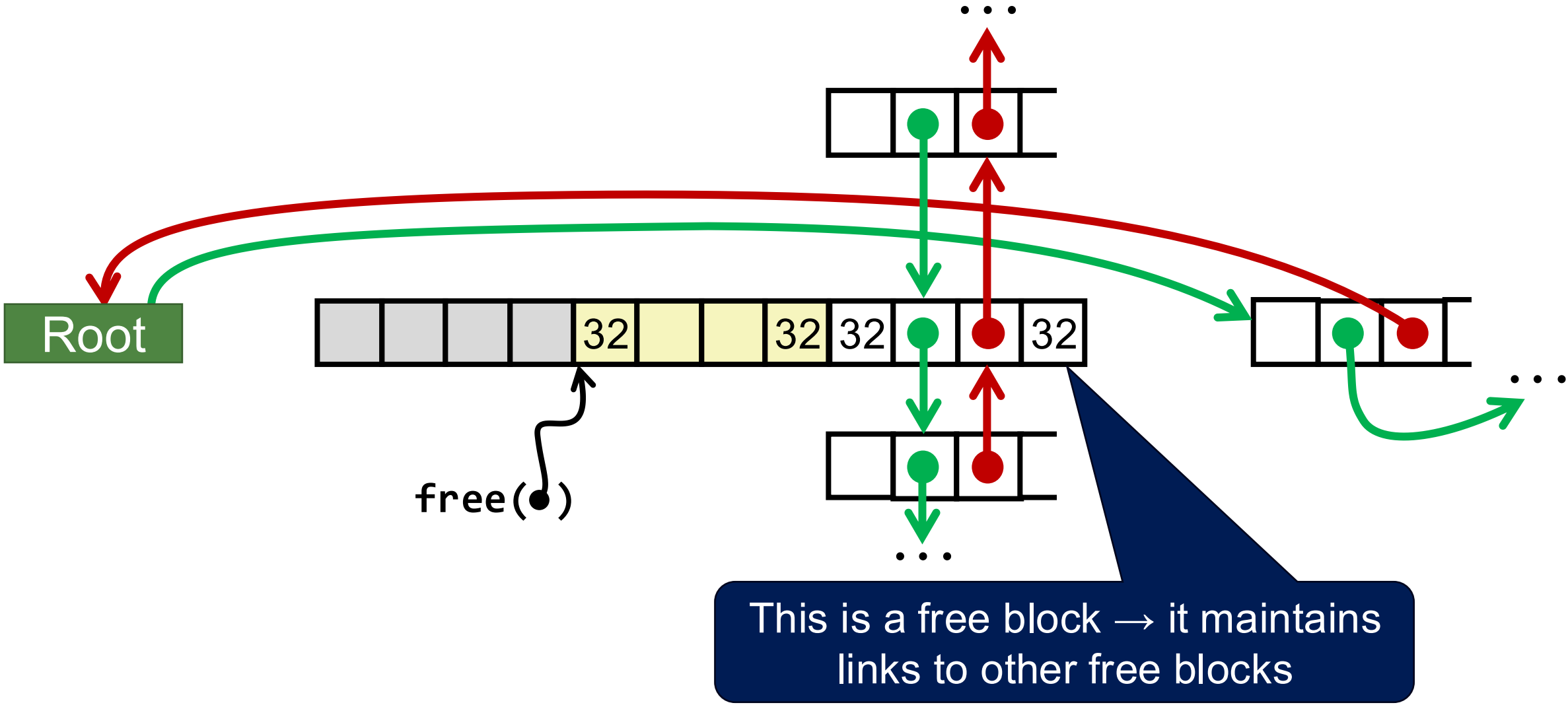
Freeing with LIFO Policy (Case 2)



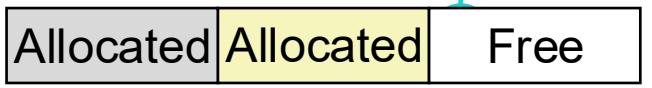


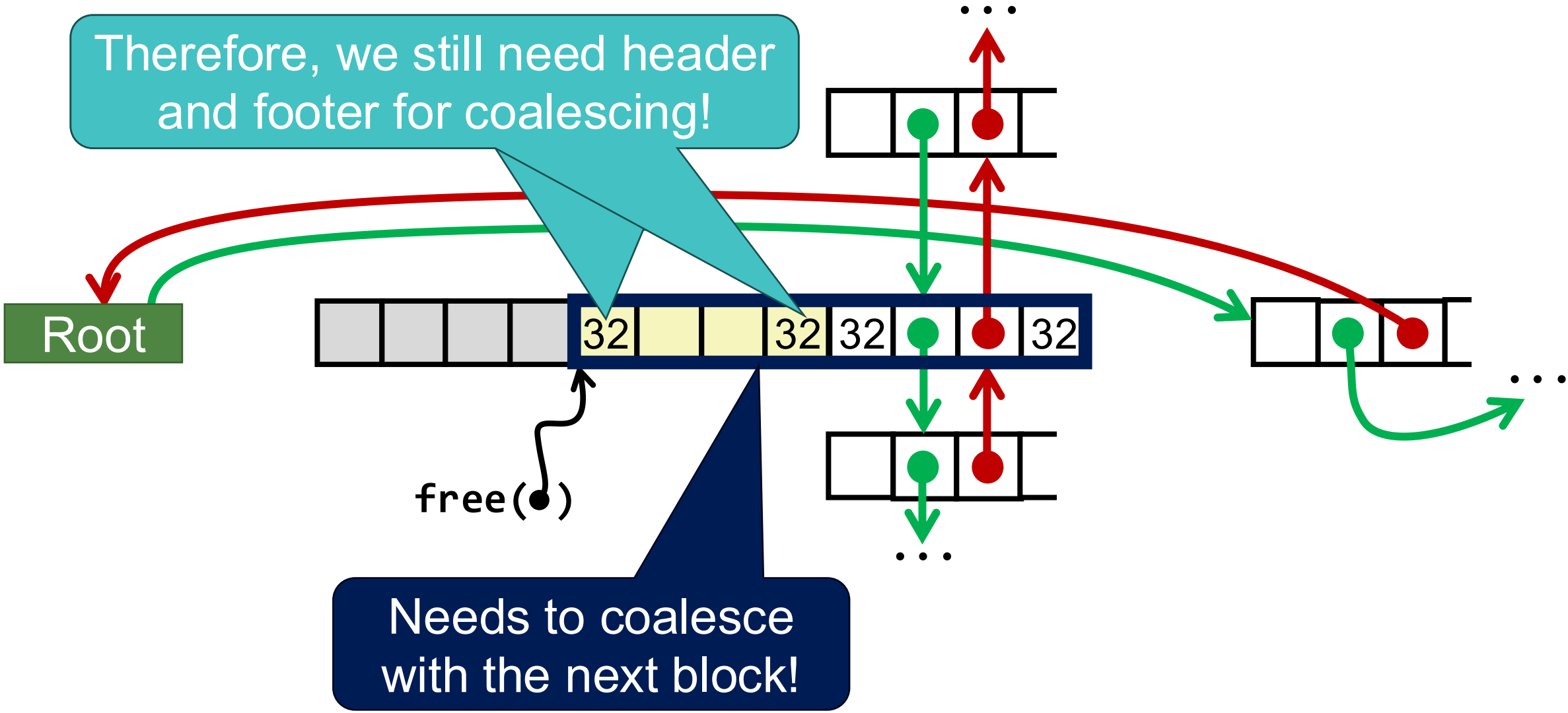
Freeing with LIFO Policy (Case 2)



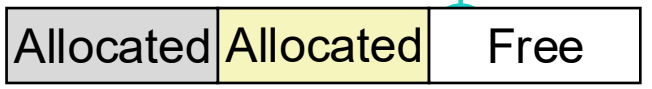


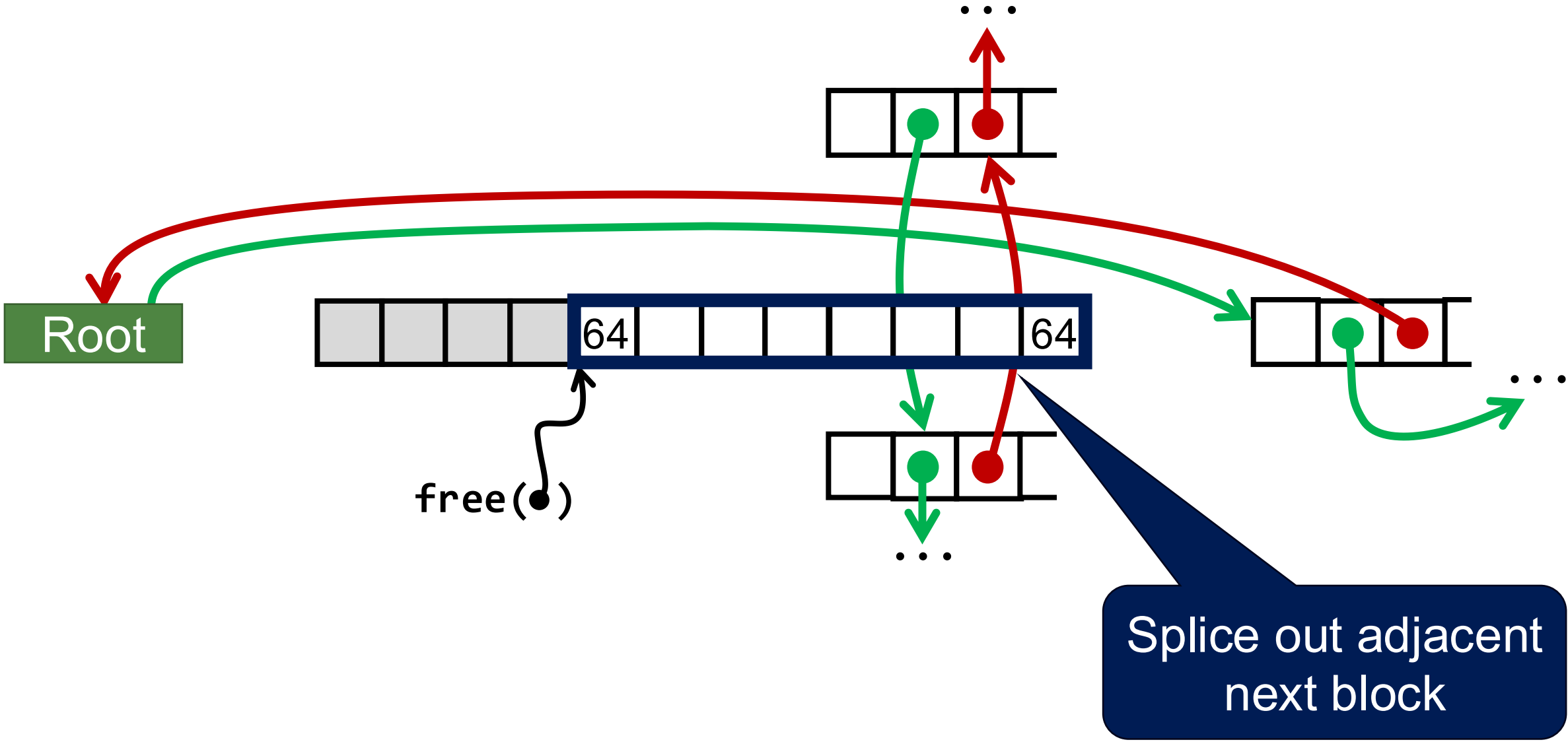
Freeing with LIFO Policy (Case 2)



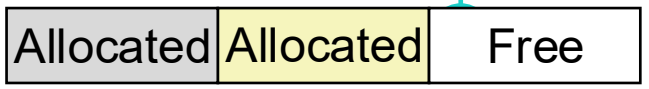


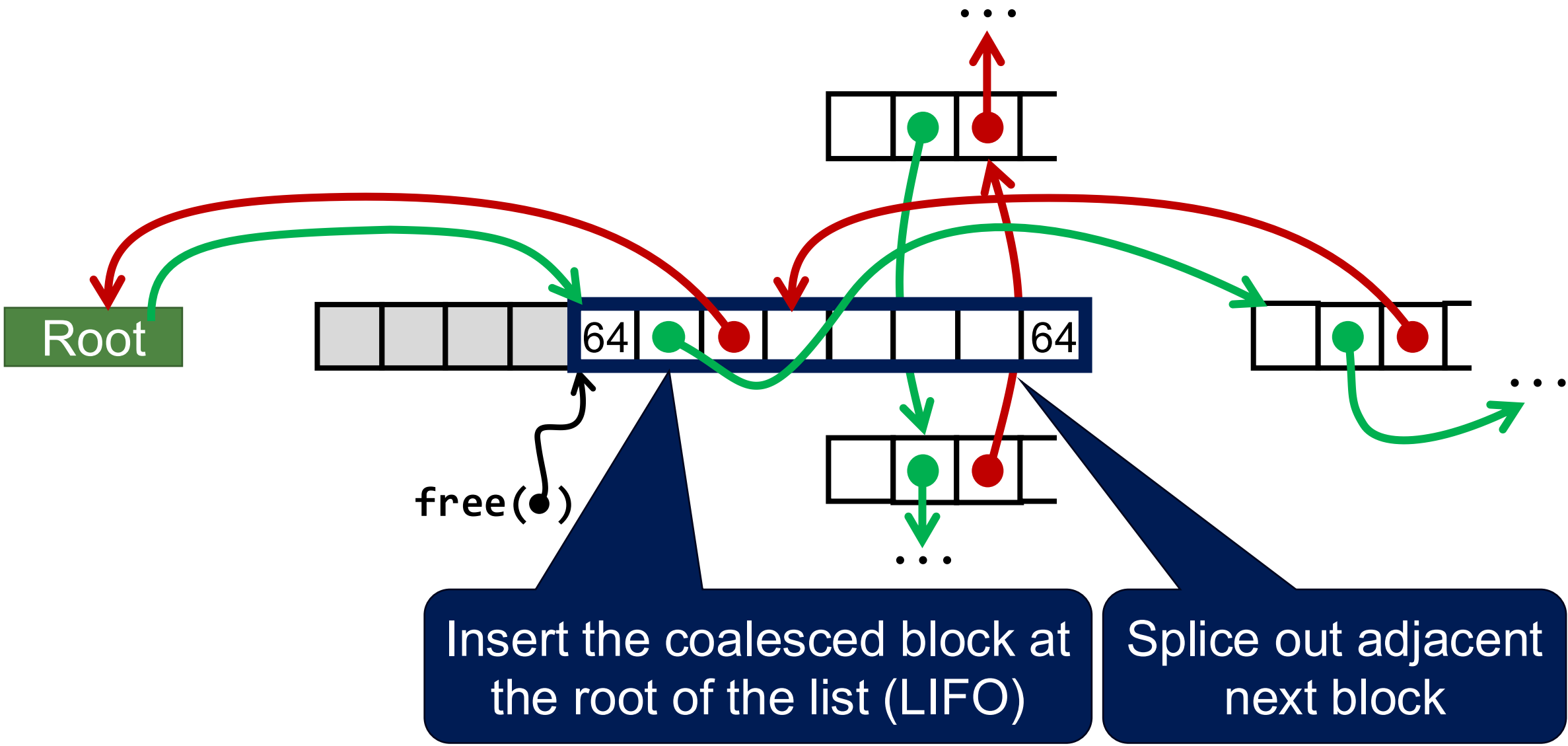
Freeing with LIFO Policy (Case 2)



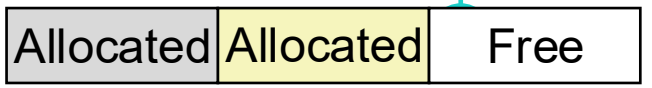


Freeing with LIFO Policy (Case 2)

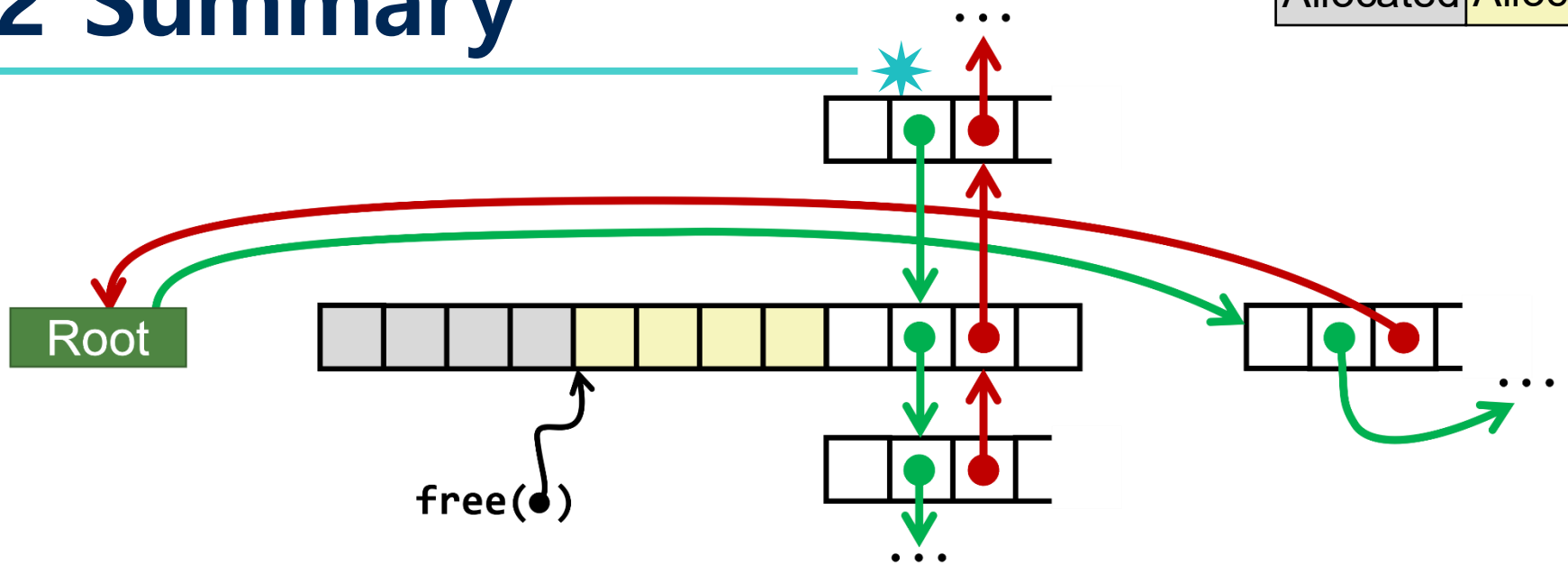




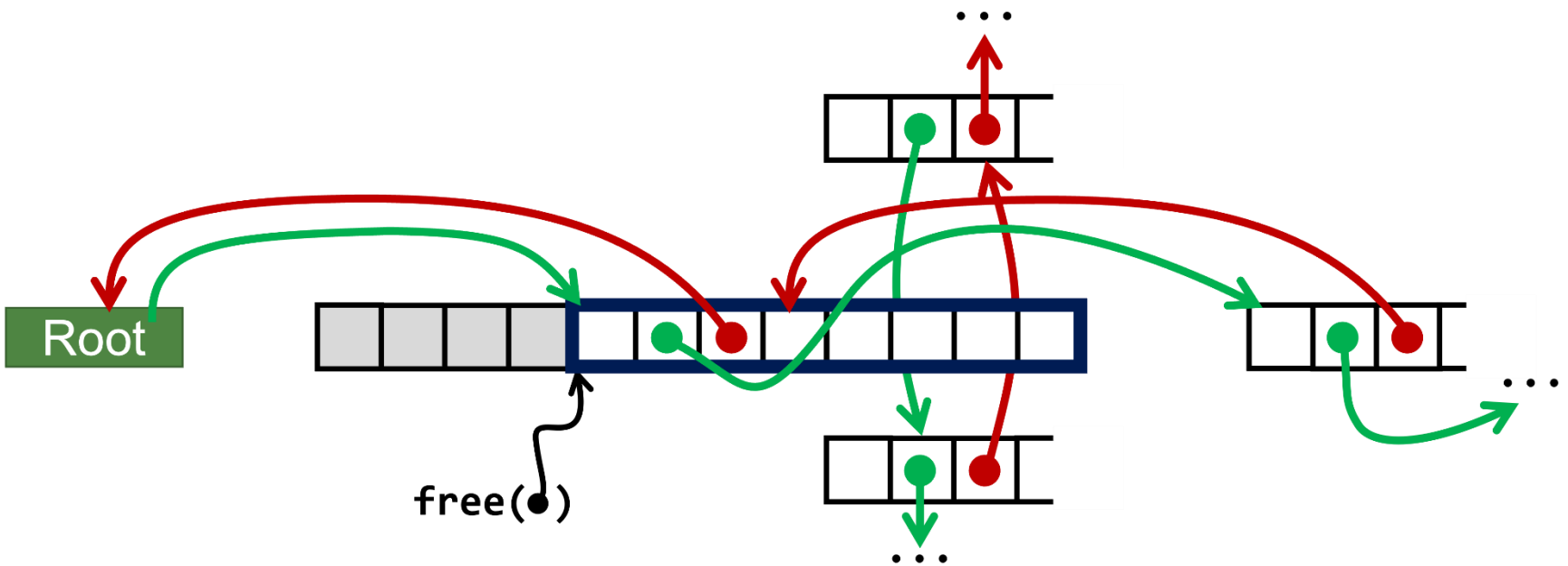
Case 2 Summary



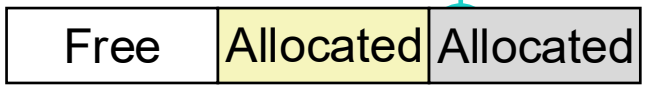
Before

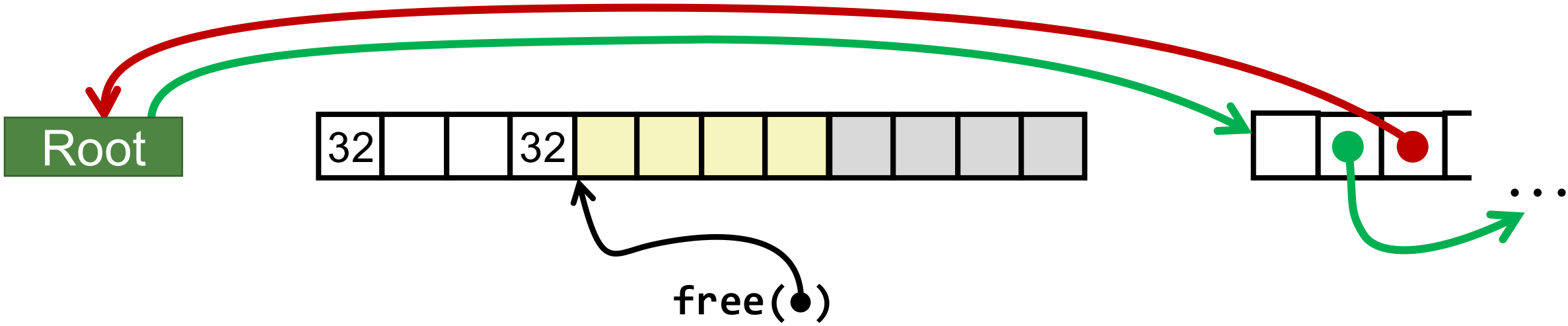


After
(with coalescing)

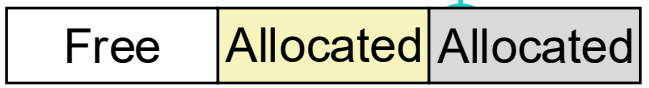


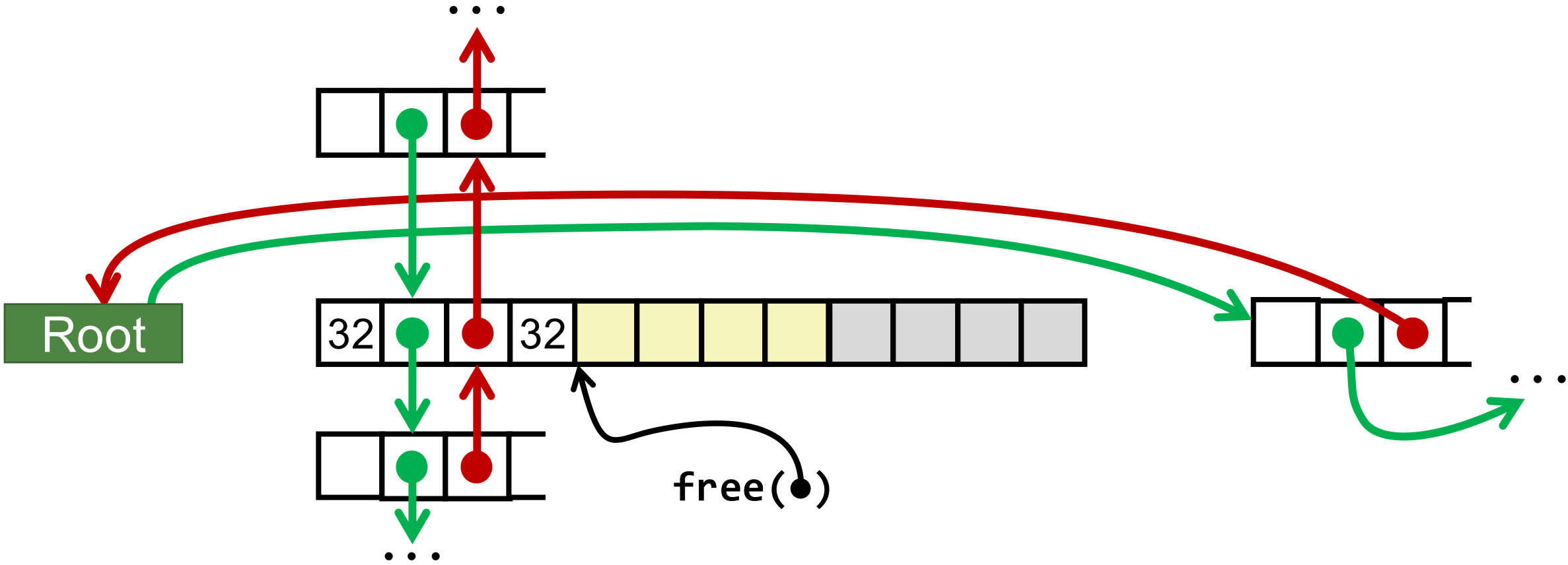
Freeing with LIFO Policy (Case 3)



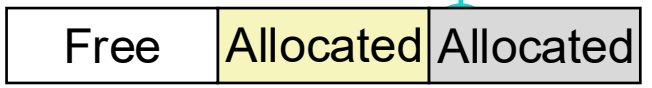


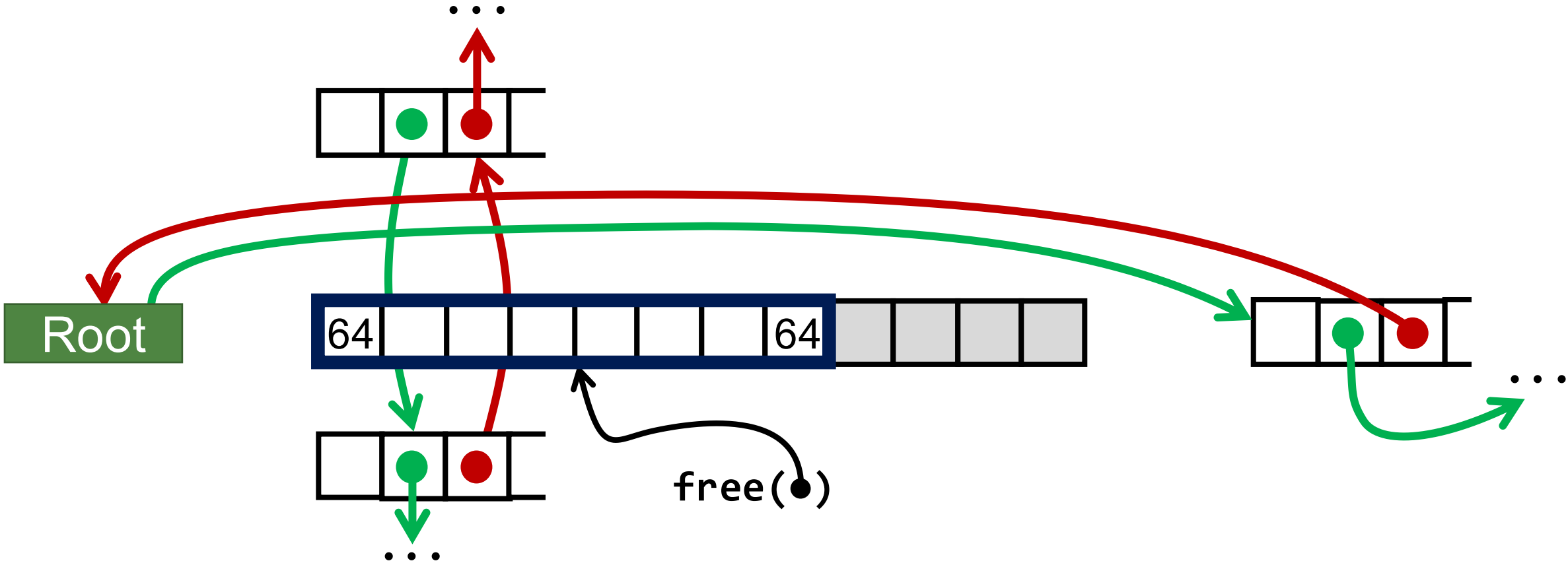
Freeing with LIFO Policy (Case 3)



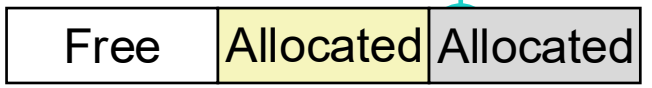


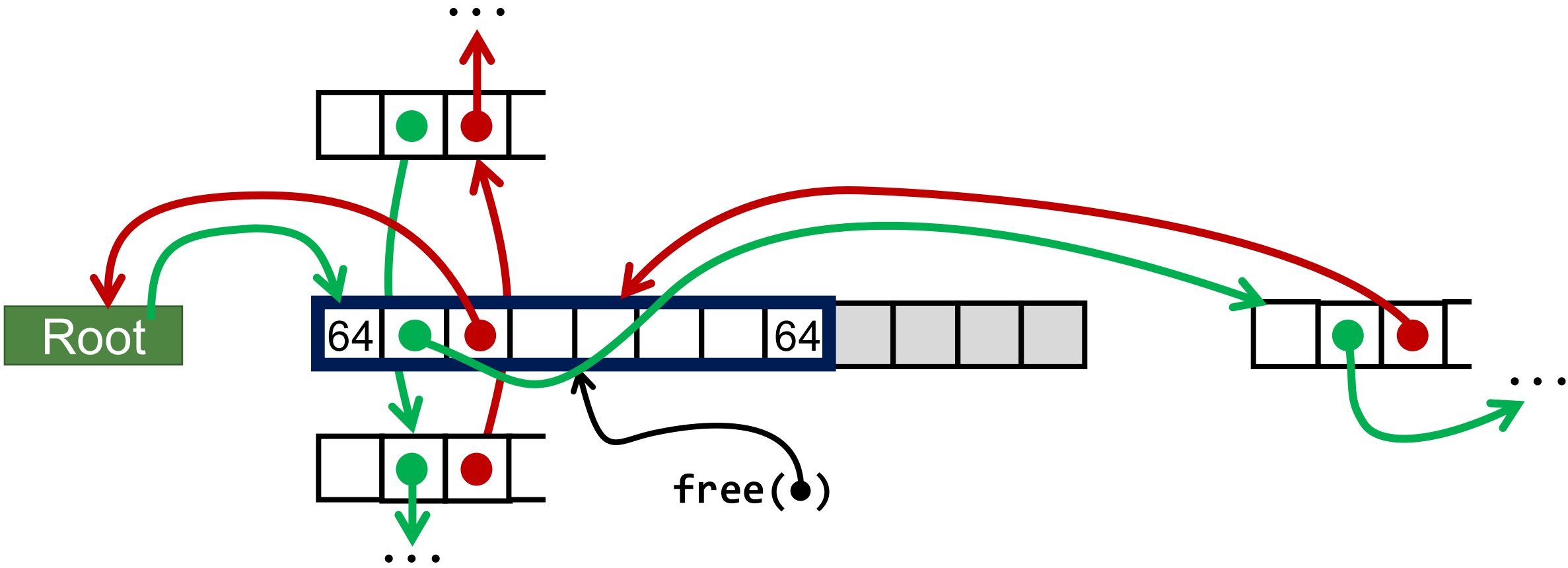
Freeing with LIFO Policy (Case 3)





Freeing with LIFO Policy (Case 3)

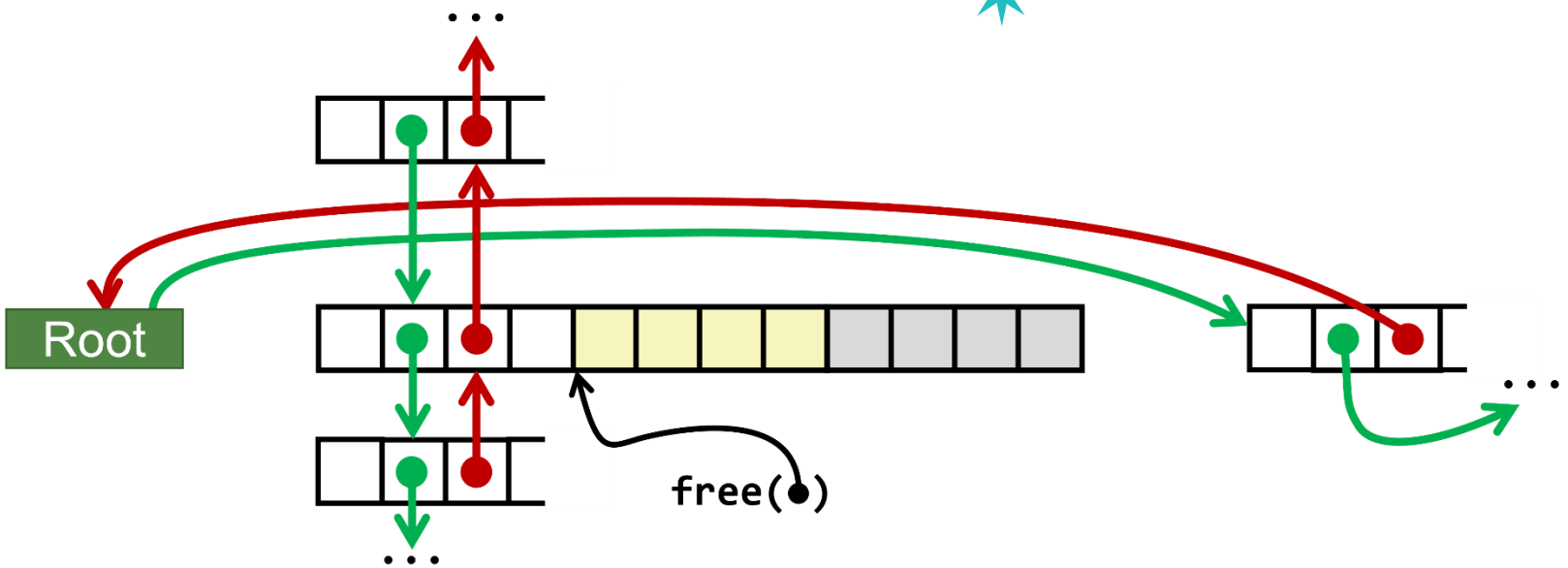




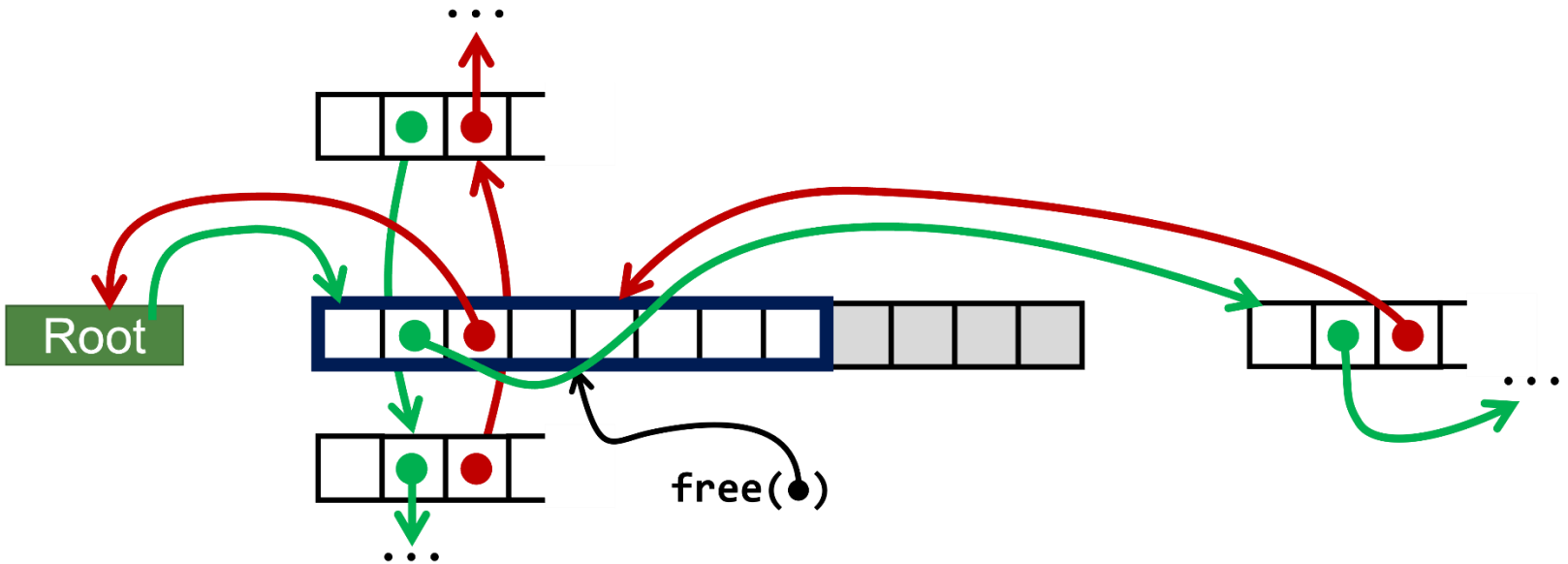
Case 3 Summary

Free	Allocated	Allocated

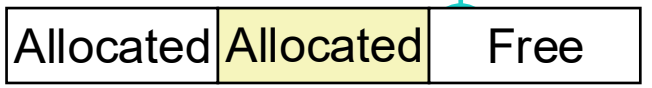
Before

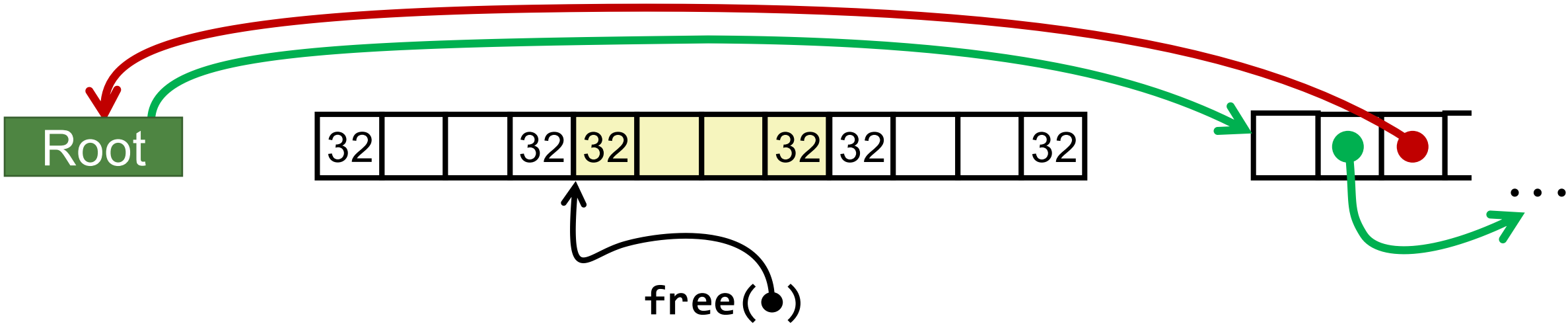


After
(with coalescing)

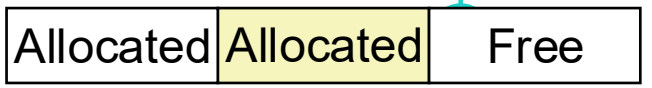


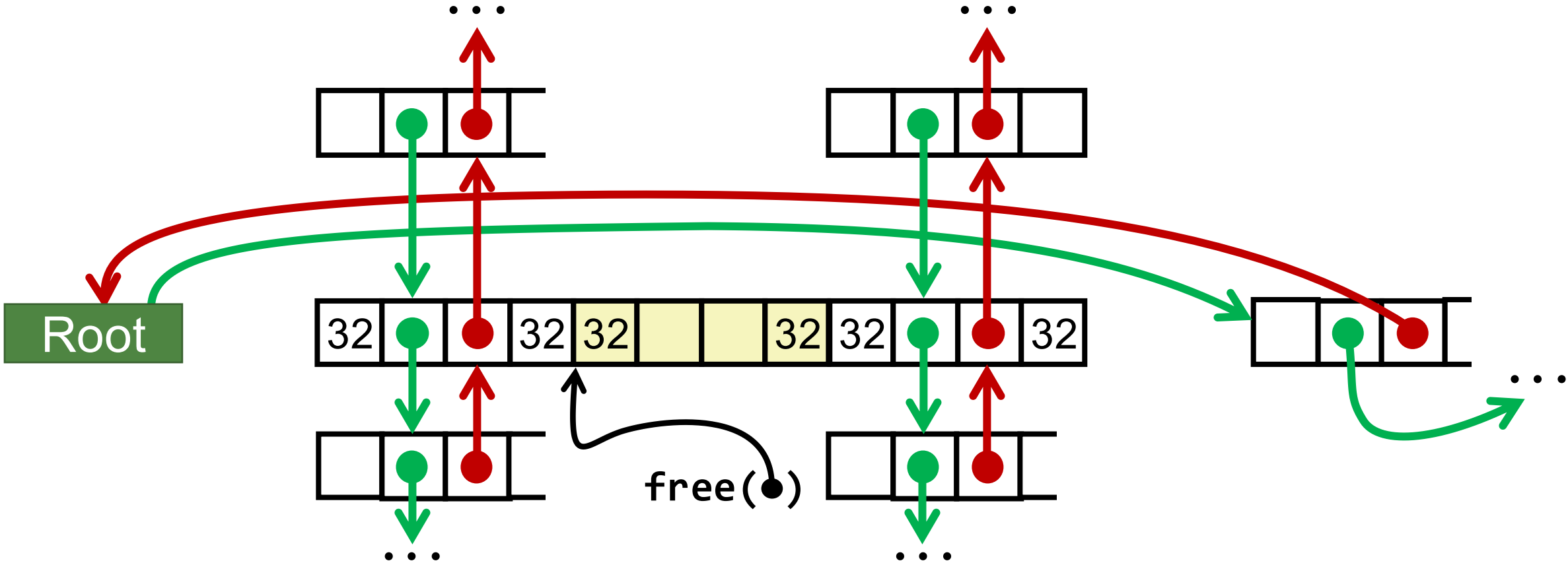
Freeing with LIFO Policy (Case 4)



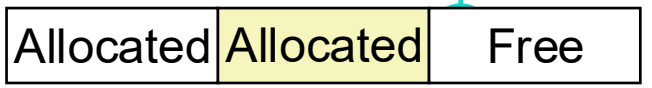


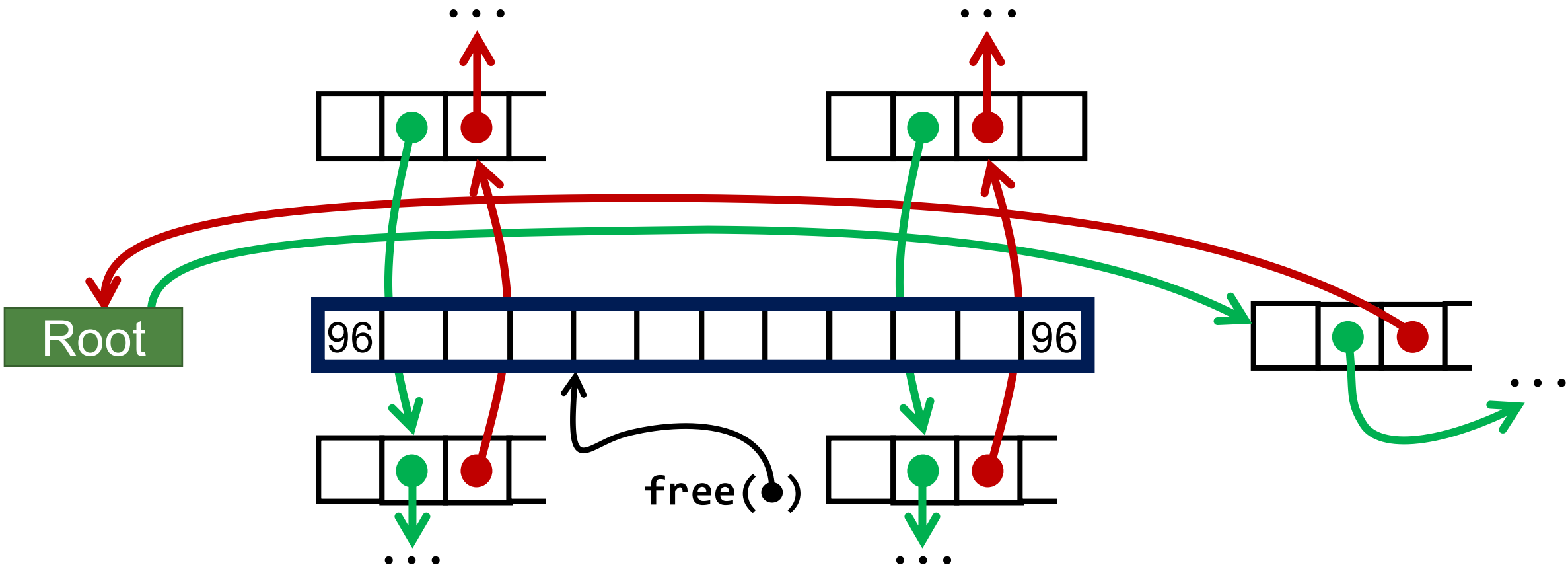
Freeing with LIFO Policy (Case 4)



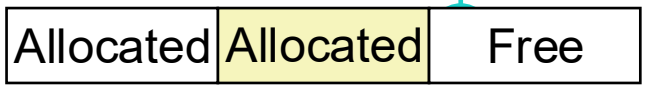


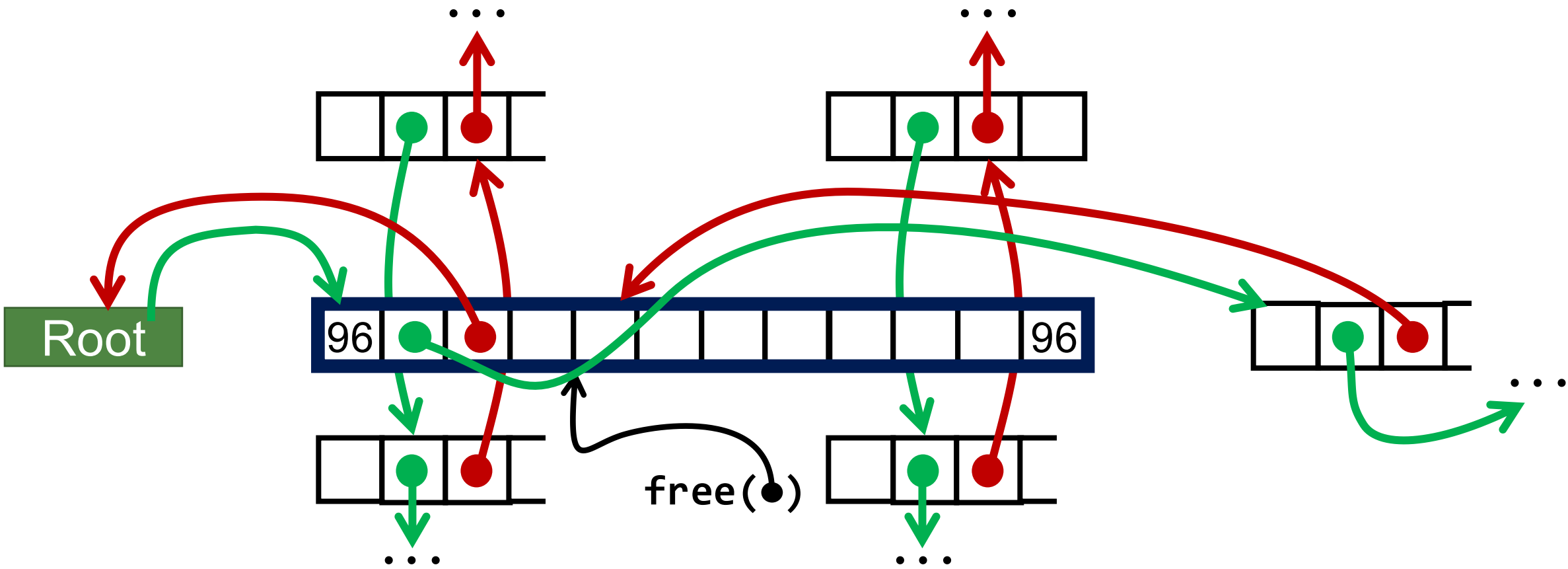
Freeing with LIFO Policy (Case 4)





Freeing with LIFO Policy (Case 4)

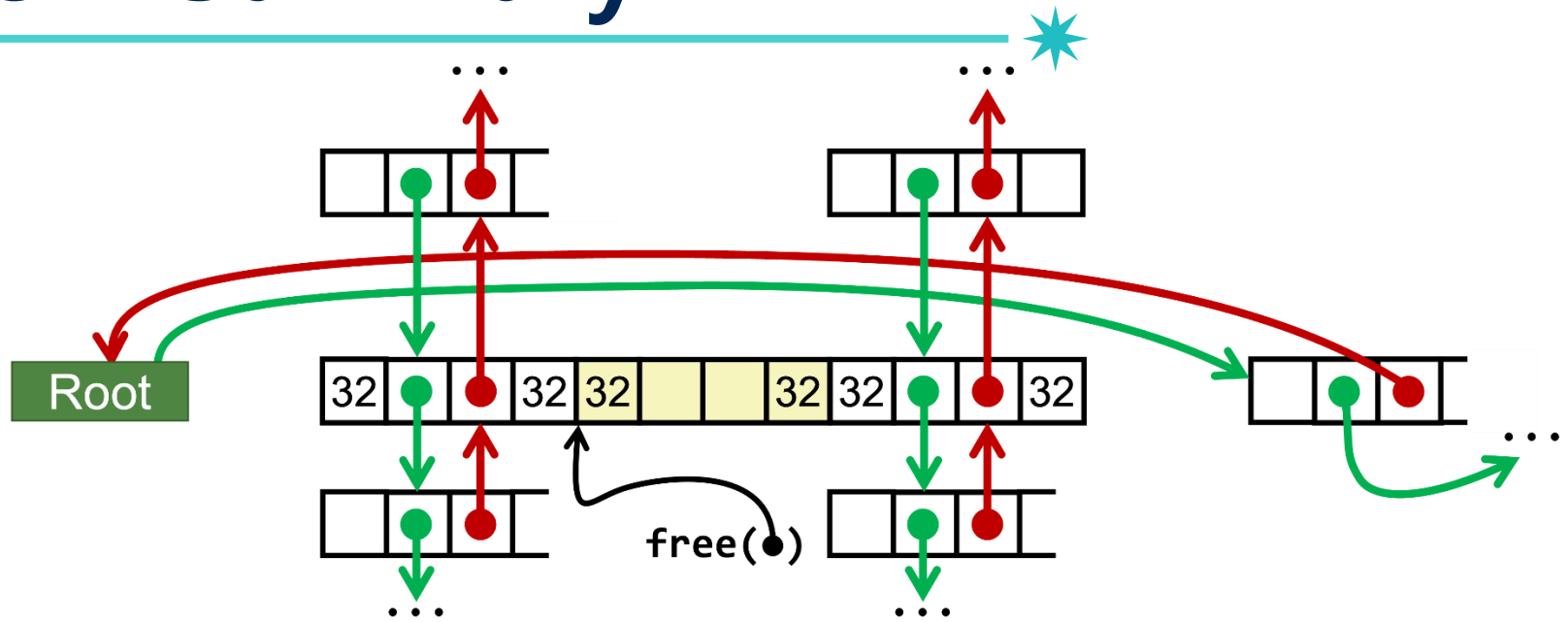




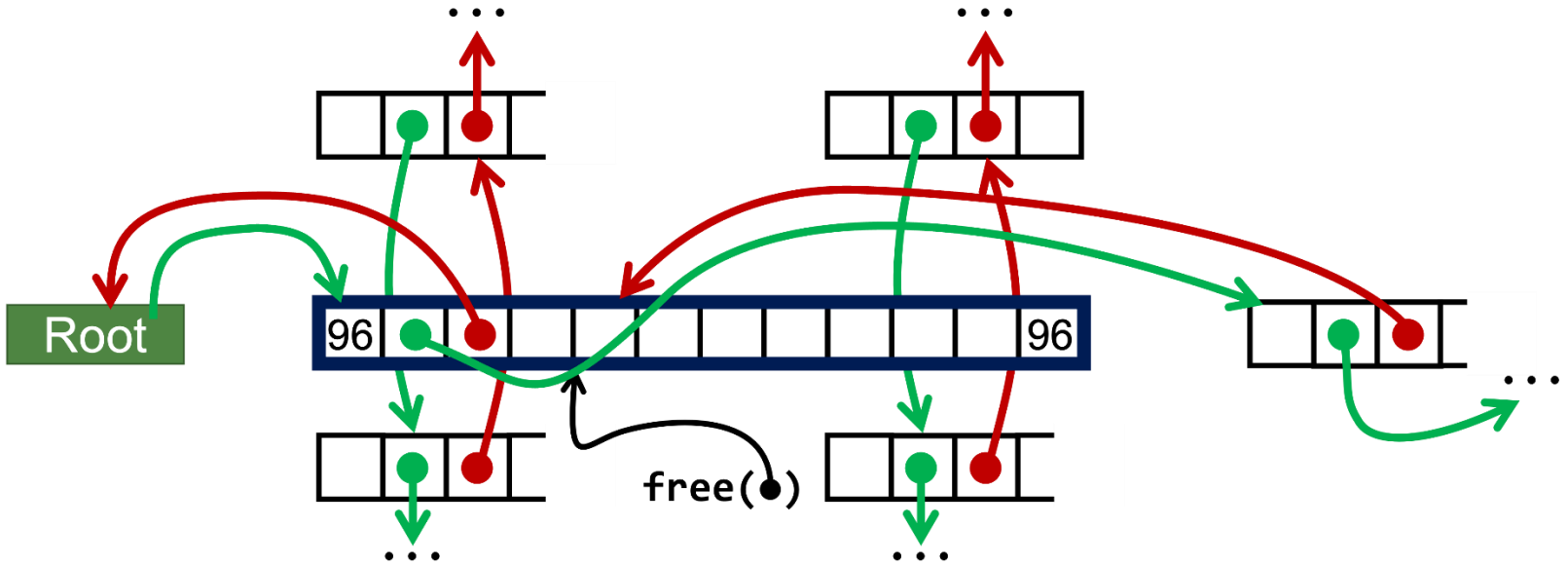
Case 4 Summary

Allocated	Allocated	Free

Before



After (with coalescing)

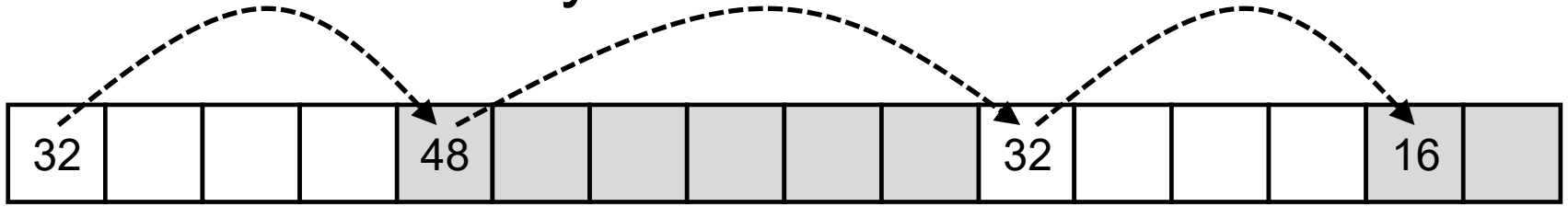


Explicit List Summary

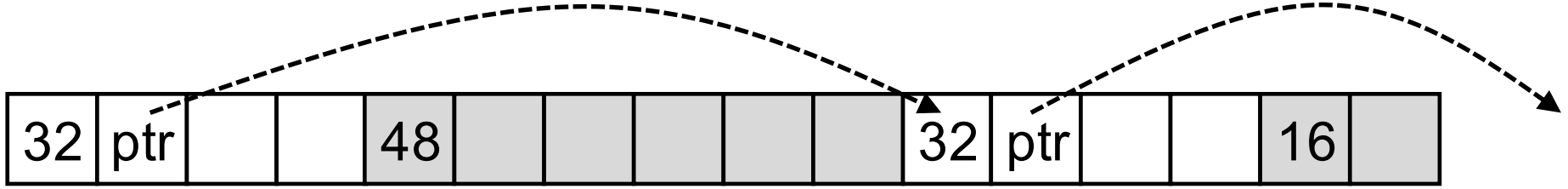
- **Comparison to implicit list:**
  - Allocate is linear time in number of ***free*** blocks instead of ***all*** blocks
    - ***Much faster*** when most of the memory is full
  - Slightly more complicated allocation and free
    - Need to splice blocks in and out of the list
  - Some extra space for the links (two extra words for each block)

Limitations of Implicit and Explicit Lists

- Method #1: ***Implicit list*** using length. It links all blocks, regardless of whether they are allocated or free



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



To find a suitable free block, the allocator must traverse the free list.

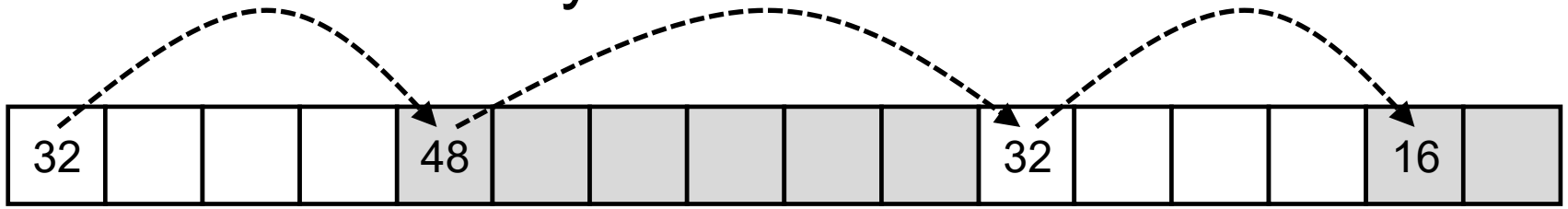
Recall: Placement Policy

- **First fit**
  - Search list from beginning, choose ***first*** free block that fits
- **Next fit**
  - Like first fit, but search list starting where previous search finished
- **Best fit**
  - Search the list, choose the ***best*** free block: fits, with fewest bytes left over

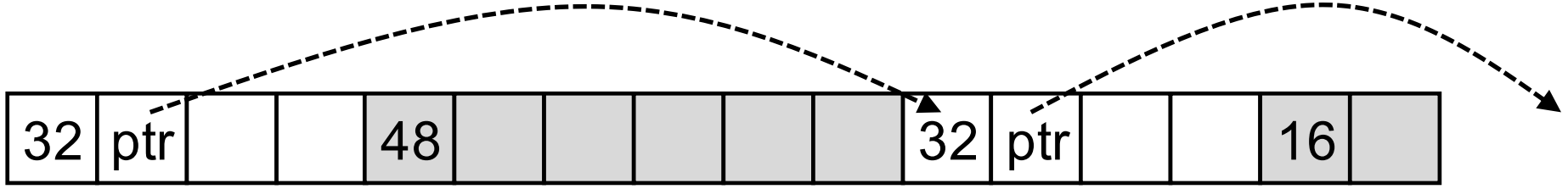
To find a suitable free block, the allocator must traverse the free list.

Limitations of Implicit and Explicit Lists

- Method #1: ***Implicit list*** using length. It links all blocks, regardless of whether they are allocated or free



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



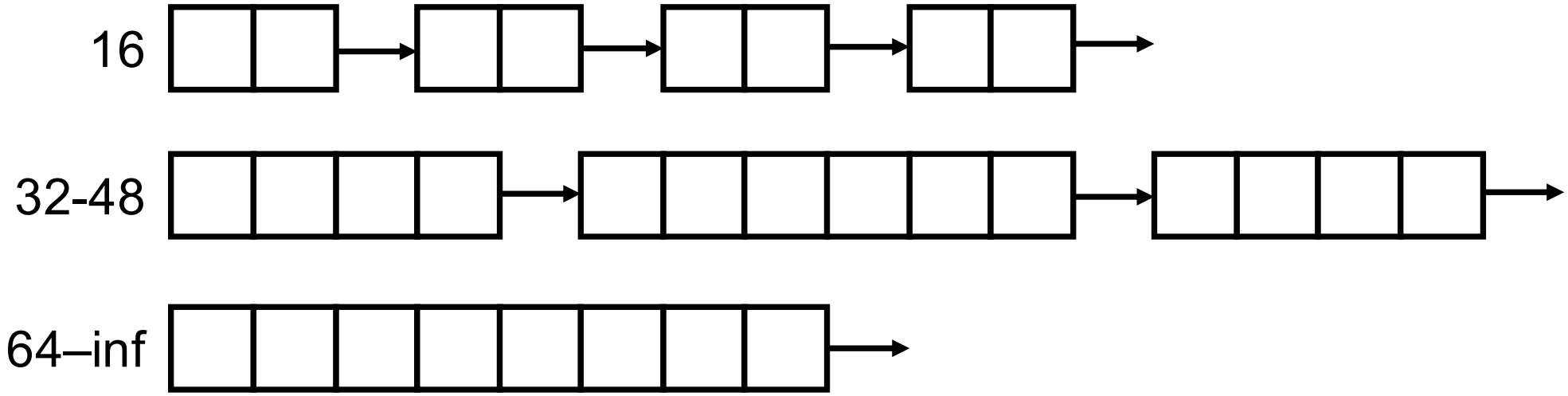
How to resolve this problem?

To find a suitable free block, the allocator must traverse the free list.

Segregated Free List

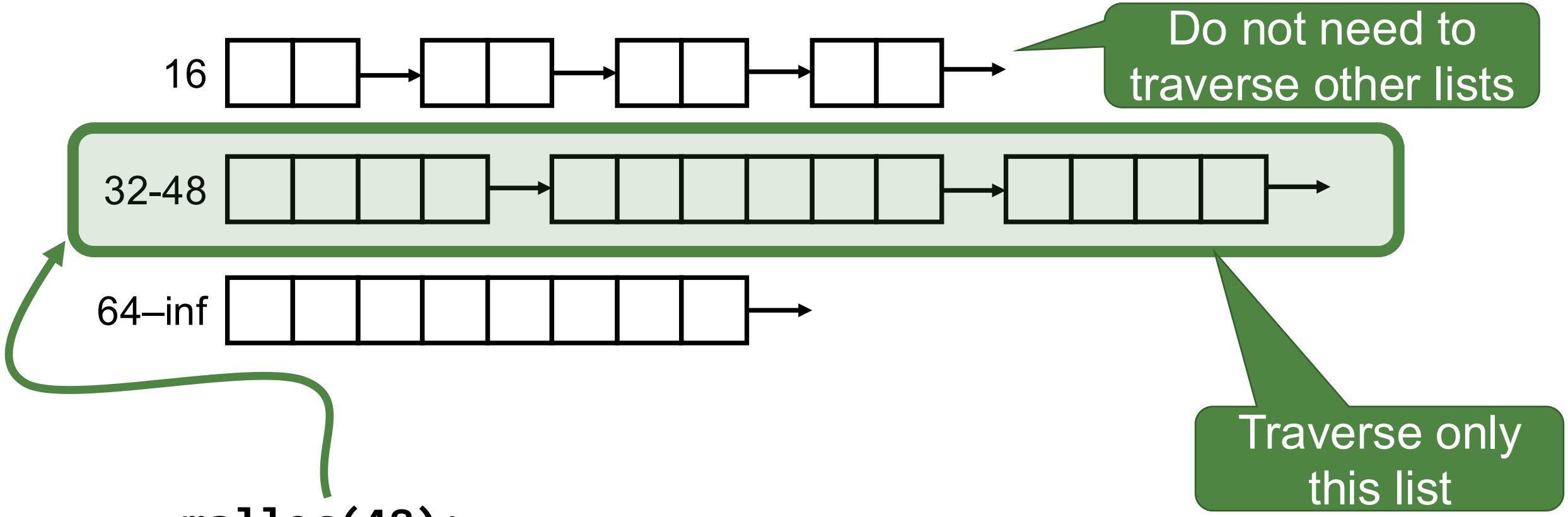
Segregated List (Seglist) Allocators

- Have several free lists, one for each ***size class*** of blocks



Segregated List (Seglist) Allocators

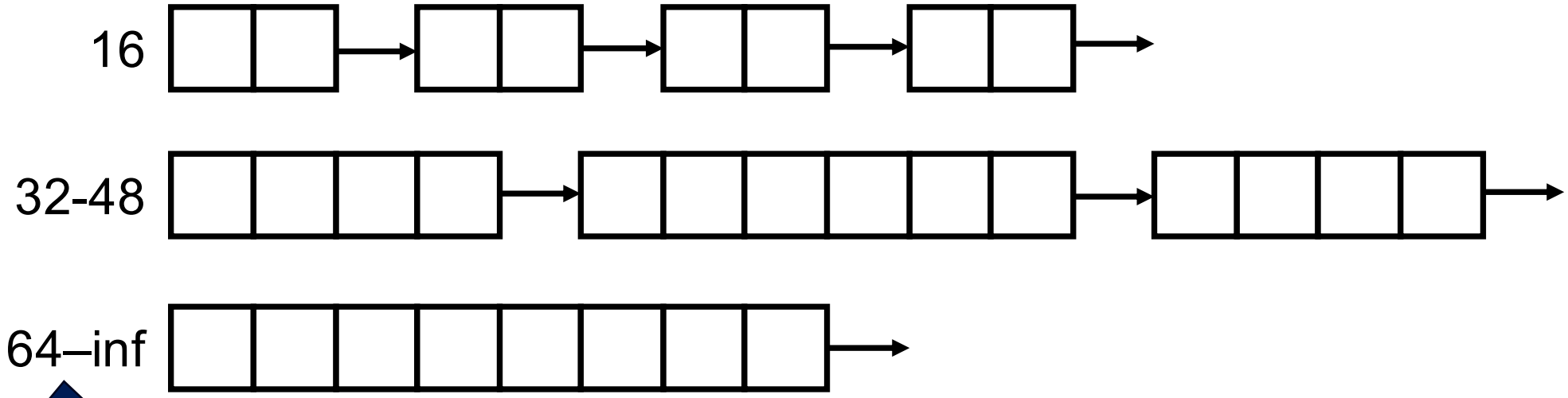
- Have several free lists, one for each ***size class*** of blocks



Higher throughput and better memory utilization

Segregated List (Seglist) Allocators

- Have several free lists, one for each ***size class*** of blocks

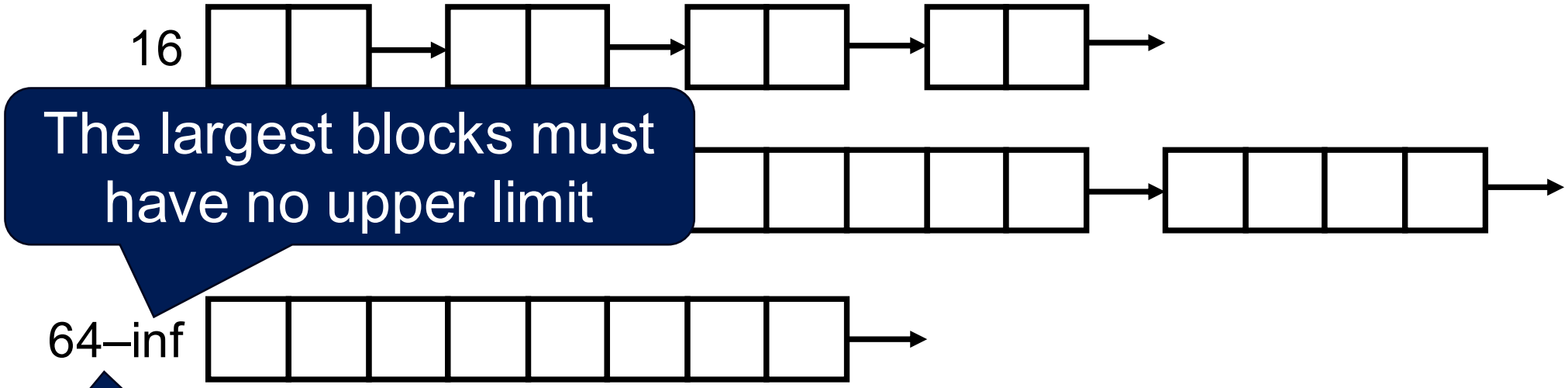


The distribution of size classes (design choice) is important

- Can have major impact on both utilization and throughput
- Common choices include:
  - One class for each small size (16, 32, 48, 64)
  - At some point switch to powers of two: $[2^i+1, 2^{i+1}]$

Segregated List (Seglist) Allocators

- Have several free lists, one for each ***size class*** of blocks



- Can have major impact on both utilization and throughput
- Common choices include:
  - One class for each small size (16, 32, 48, 64)
  - At some point switch to powers of two: $[2^i+1, 2^{i+1}]$

Segregated List (Seglist) Allocators

- Have several free lists, one for each ***size class*** of blocks

- **To allocate a block of size *n*:**
  - Search appropriate free list for block of size $m \geq n$ (i.e., first fit)
  - If an appropriate block is found:
    - Split block and place fragment on appropriate list
    - 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 appropriate size class

Segregated List (Seglist) Allocators

- Have several free lists, one for each ***size class*** of blocks

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

Segregated List (Seglist) Allocators

- **Advantages of seglist allocators vs. non-seglist allocators (both with first-fit)**
  - Higher throughput
    - log time for power-of-two size classes vs. linear time
  - 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

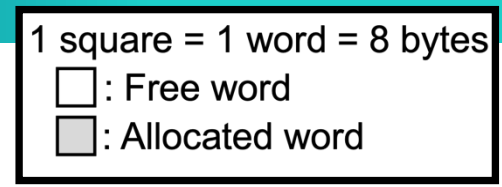
More Information on Allocators

- **D. Knuth, *The Art of Computer Programming,* vol 1, 3rd edition, Addison Wesley, 1997**
  - The classic reference on dynamic storage allocation

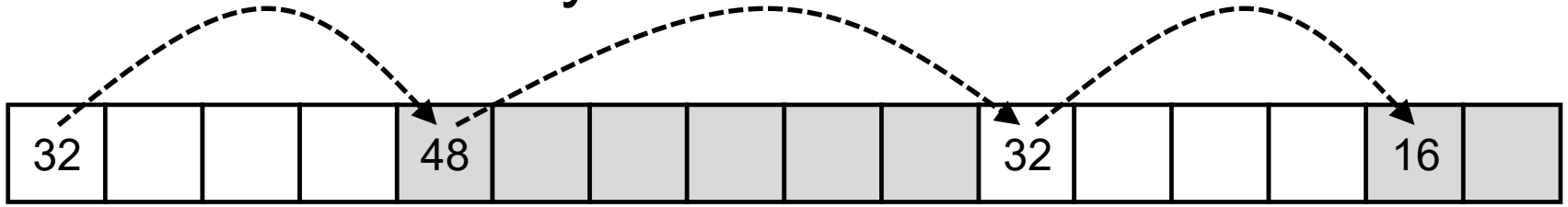
- **Wilson et al, "*Dynamic Storage Allocation: A Survey and Critical Review*", Proc. 1995 Int'l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.**
  - Comprehensive survey

- **Railing, et al, "Implementing Malloc: Students and Systems Programming", SIGCSE'18, Feb 2018.**

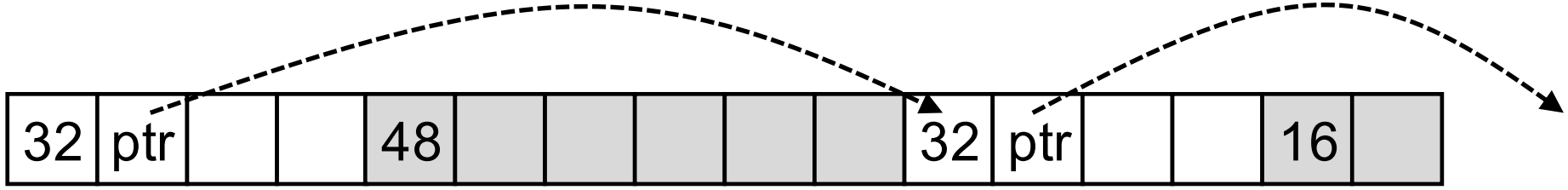
Summary



- Method #1: ***Implicit list*** using length. It links all blocks, regardless of whether they are allocated or free



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



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

Question?