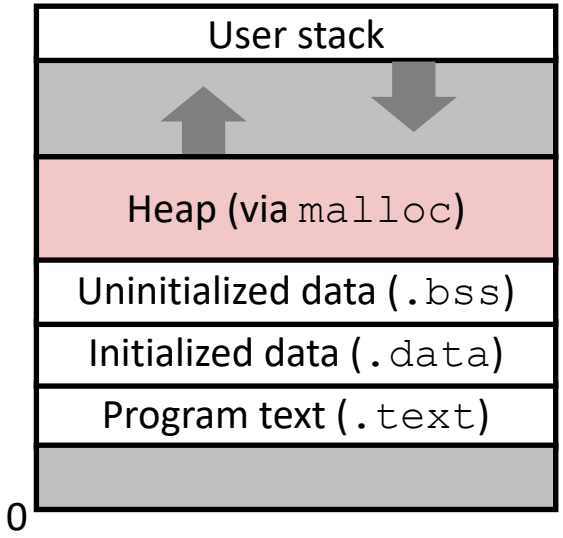
Dynamic Memory Allocation

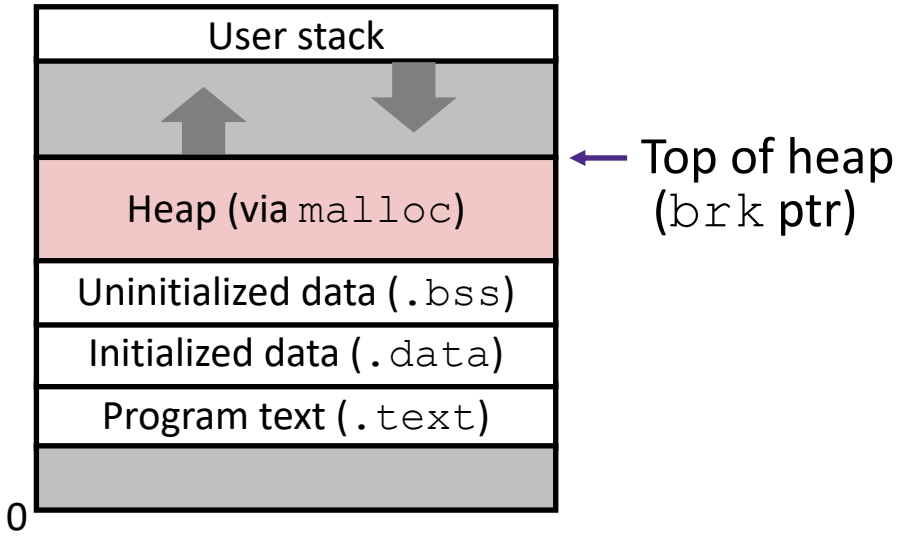
- Programmers use dynamic memory allocators to acquire virtual memory at run time
  - For data structures whose size (or lifetime) is known only at runtime
  - Manage the heap of a process' virtual memory:



- Types of allocators
  - ***Explicit* allocator:** programmer allocates and frees space
    - Example: `malloc` and `free` in C
  - ***Implicit* allocator:** programmer only allocates space (no free)
    - Example: garbage collection in Java, Caml, and Lisp

Dynamic Memory Allocation

- Allocator organizes heap as a collection of variable-sized *blocks*, which are either *allocated* or *free*
  - Allocator requests pages in the heap region; virtual memory hardware and OS kernel allocate these pages to the process
  - Application objects are typically smaller than pages, so the allocator manages blocks *within* pages
    - (Larger objects handled too; ignored here)



Memory Allocation Example in C

```
void foo(int n, int m) {
  int i, *p;
  p = (int*) malloc(n*sizeof(int)); /* allocate block of n ints */
  if (p == NULL) {                  /* check for allocation error */
    perror("malloc");
    exit(0);
  }
  for (i=0; i<n; i++)               /* initialize int array */
    p[i] = i;
                                    /* add space for m ints to end of p block */
  p = (int*) realloc(p,(n+m)*sizeof(int));
  if (p == NULL) {                  /* check for allocation error */
    perror("realloc");
    exit(0);
  }
  for (i=n; i < n+m; i++)           /* initialize new spaces */
    p[i] = i;
  for (i=0; i<n+m; i++)             /* print new array */
    printf("%d\n", p[i]);
  free(p);                          /* free p */
}
```

Performance Goals

- **Goals:** Given some sequence of `malloc` and `free` requests $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$, maximize throughput and peak memory utilization
  - These goals are often conflicting

1) Throughput

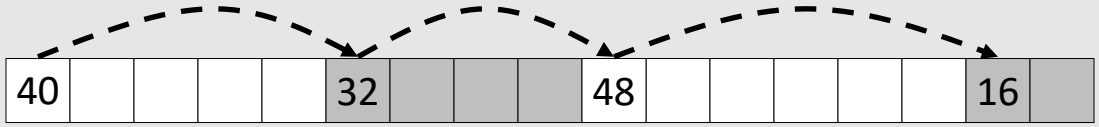
- Number of completed requests per unit time
- Example:
  - If 5,000 `malloc` calls and 5,000 `free` calls completed in 10 seconds, then throughput is 1,000 operations/second

Keeping Track of Free Blocks

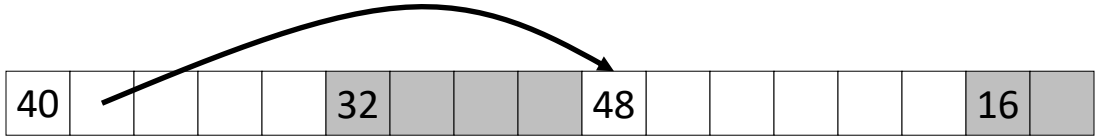

= 8-byte word (free)


= 8-byte word (allocated)

1) *Implicit free list* using length – links all blocks using math
   - No actual pointers, and must check each block if allocated or free



2) *Explicit free list* among only the free blocks, using pointers



3) *Segregated free list*
   - Different free lists for different size "classes"

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

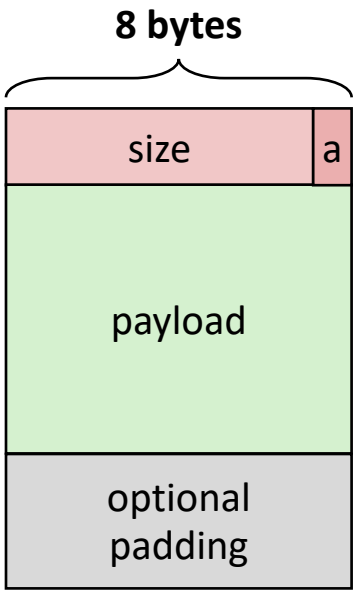
Implicit Free Lists

e.g. with 8-byte alignment, possible values for size:

00001000 = 8 bytes
00010000 = 16 bytes
00011000 = 24 bytes
...

- For each block we need: **`size`, is-allocated?**
  - Could store using two words, but wasteful
- Standard trick
  - If blocks are aligned, some low-order bits of `size` are always 0
  - Use lowest bit as an allocated/free flag (fine as long as aligning to K>1)
  - When reading `size`, must remember to mask out this bit!

Format of allocated and free blocks:



a = 1: allocated block
a = 0: free block

size: block size (in bytes)

payload: application data (allocated blocks only)

If `x` is first word (header):

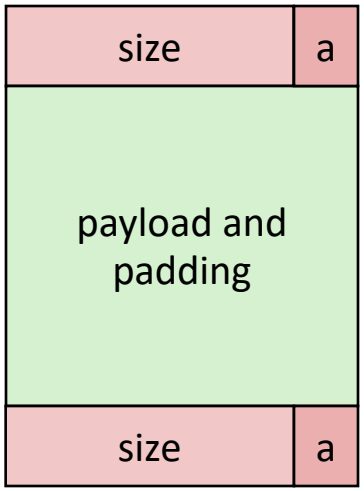
```
x = size | a;

a = x & 1;

size = x & ~1;
```

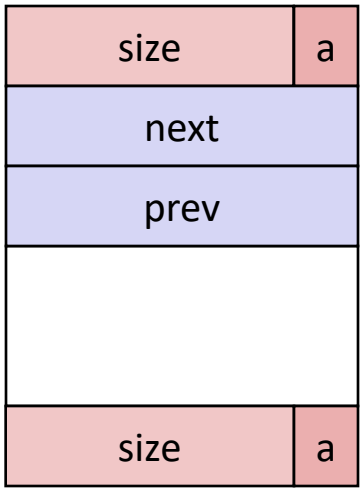
Explicit Free Lists

Allocated block:



(same as implicit free list)

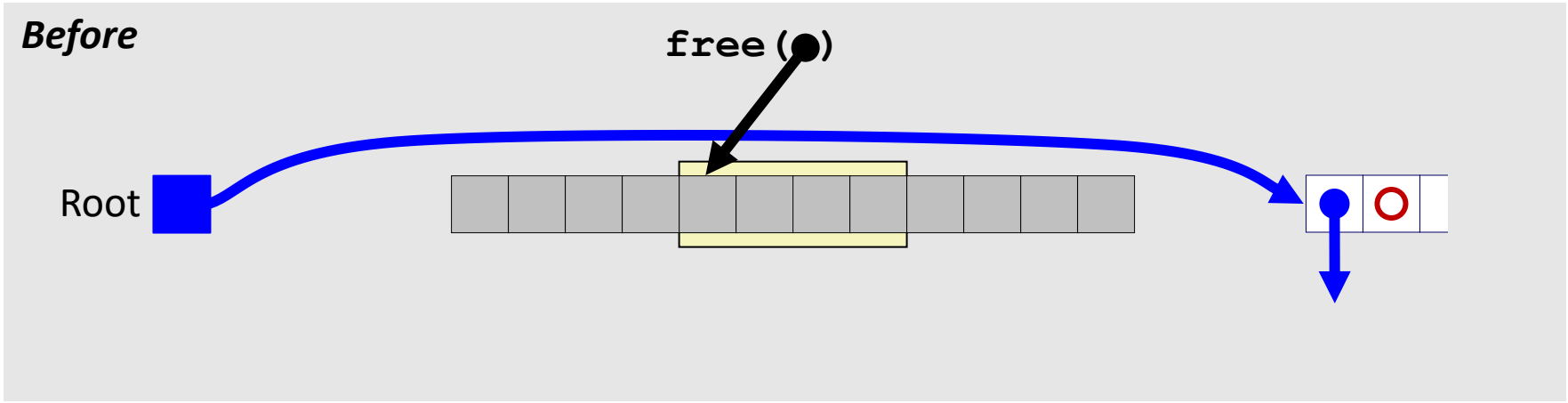
Free block:



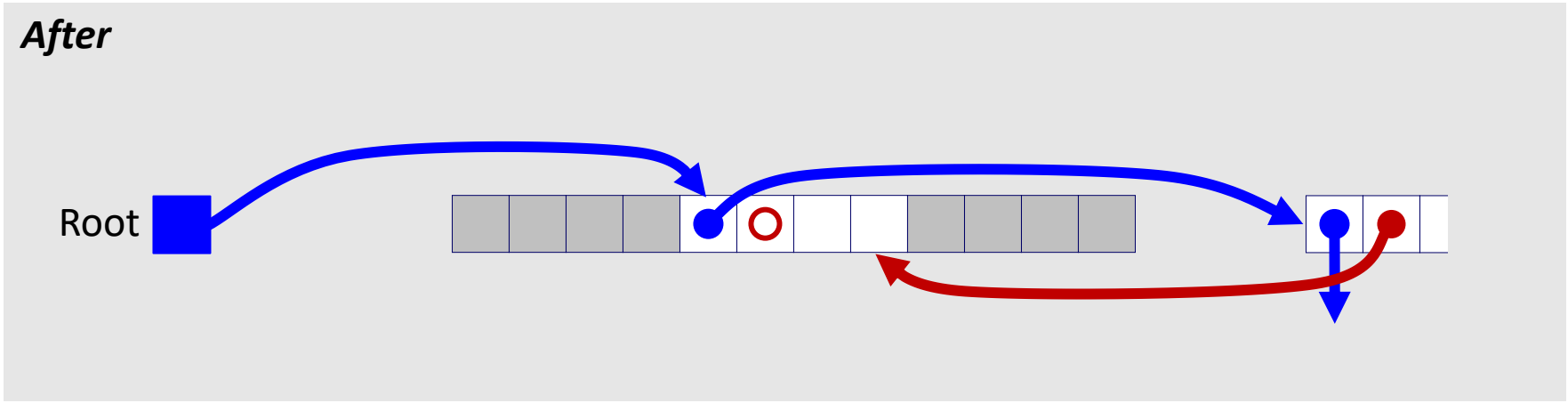
- Use list(s) of *free* blocks, rather than implicit list of *all* blocks
  - The "next" free block could be anywhere in the heap
    - So we need to store next/previous pointers, not just sizes
  - Since we only track free blocks, so we can use "payload" for pointers
  - Still need boundary tags (header/footer)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

Freeing with LIFO Policy (Case 1)

Boundary tags not shown, but don't forget about them!

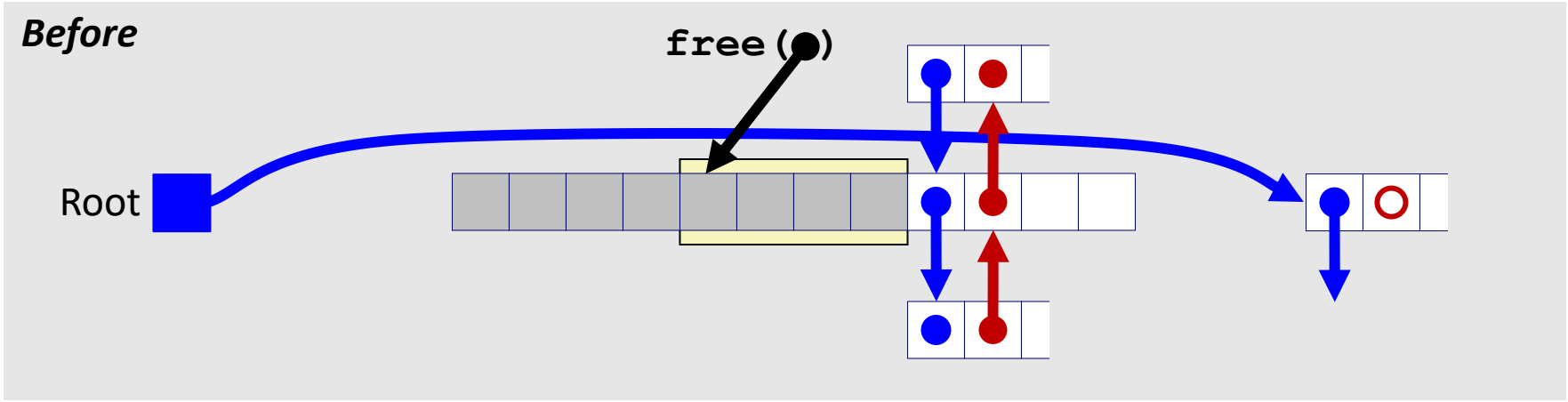


- Insert the freed block at the root of the list

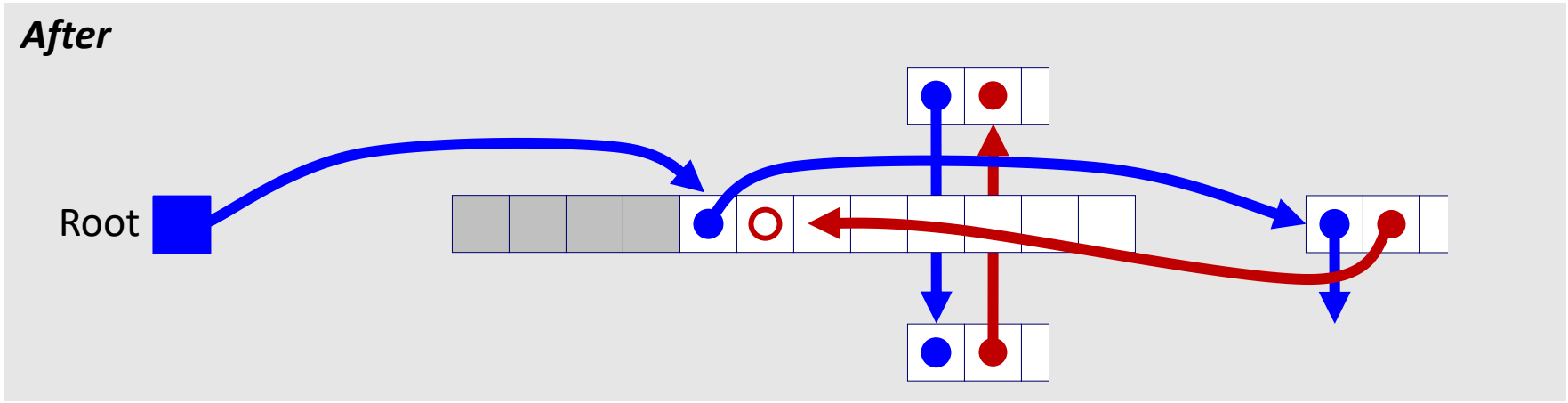


Freeing with LIFO Policy (Case 2)

Boundary tags not shown, but don't forget about them!

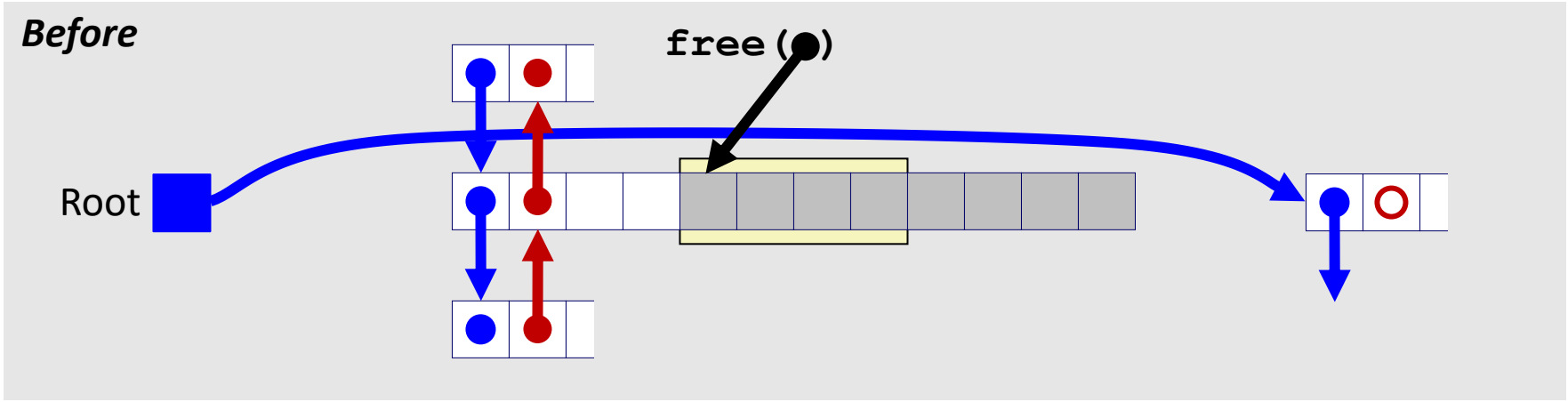


- Splice *successor* block out of list, coalesce both memory blocks, and insert the new block at the root of the list

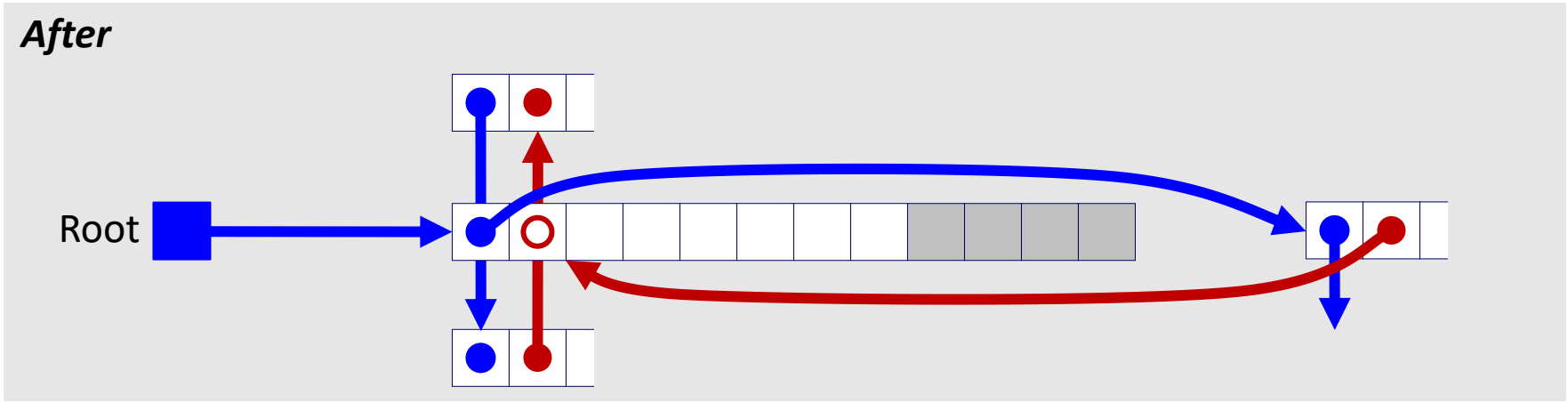


Freeing with LIFO Policy (Case 3)

Boundary tags not shown, but don't forget about them!

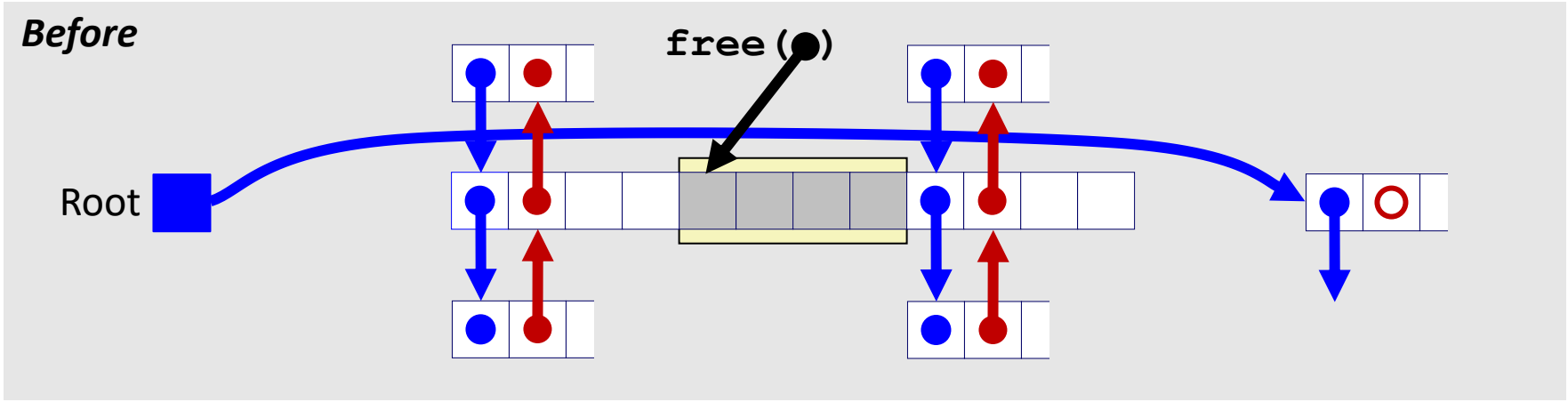


- Splice *predecessor* block out of list, coalesce both memory blocks, and insert the new block at the root of the list

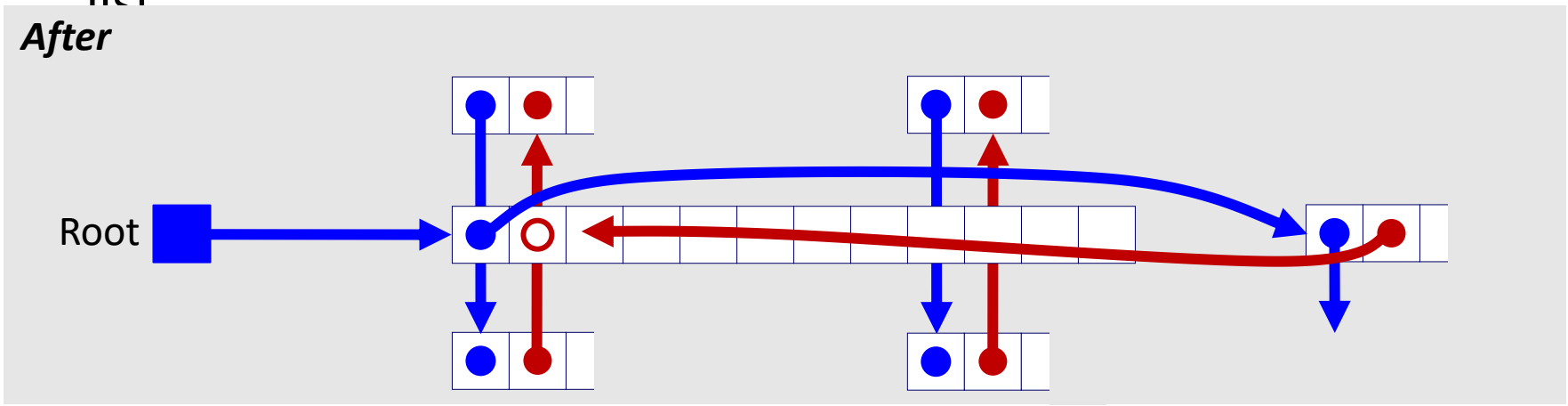


Freeing with LIFO Policy (Case 4)

Boundary tags not shown, but don't forget about them!



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



Explicit List Summary

- Comparison with implicit list:
  - Block allocation 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 we need to splice blocks in and out of the list
  - Some extra space for the links (2 extra pointers needed for each free block)
    - 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

Allocation Policy Tradeoffs

- Data structure of blocks on lists
  - Implicit (free/allocated), explicit (free), segregated (many free lists) – others possible!
- Placement policy: first-fit, next-fit, best-fit
  - Throughput vs. amount of fragmentation
- When do we split free blocks?
  - How much internal fragmentation are we willing to tolerate?
- When do we coalesce free blocks?
  - **Immediate coalescing:** Every time `free` is called
  - **Deferred coalescing:** Defer coalescing until needed
    - e.g. when scanning free list for `malloc` or when external fragmentation reaches some threshold

More Info on Allocators

- D. Knuth, "*The Art of Computer Programming*", 2nd edition, Addison Wesley, 1973
  - 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
  - Available from CS:APP student site (csapp.cs.cmu.edu)