Dynamic Memory II

15-213/18-243: Introduction to Computer Systems

19th Lecture, 30 March 2010

Instructors:

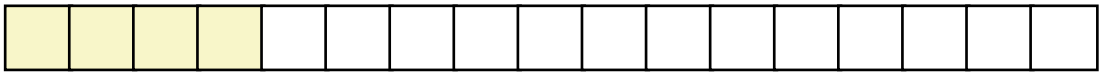
Bill Nace and Gregory Kesden

Exam 2: 6 April

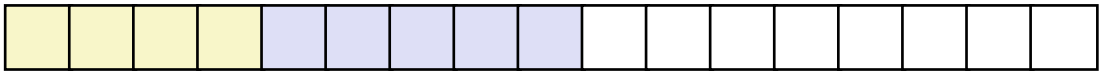
- **At your assigned lecture time**
- **Closed book, closed notes, closed friends, open mind**
  - We will provide reference material (if necessary)
- **Covers:**
  - Lectures 10 - 19
    - Optimizations
    - Exceptions
    - Signals
    - Virtual Memory
    - Dynamic Memory
  - Labs
    - Tshlab
    - Malloclab (through checkpoint 1)

Last Time: Dynamic Memory Allocation

```
p1 = malloc(4)
```



```
p2 = malloc(5)
```



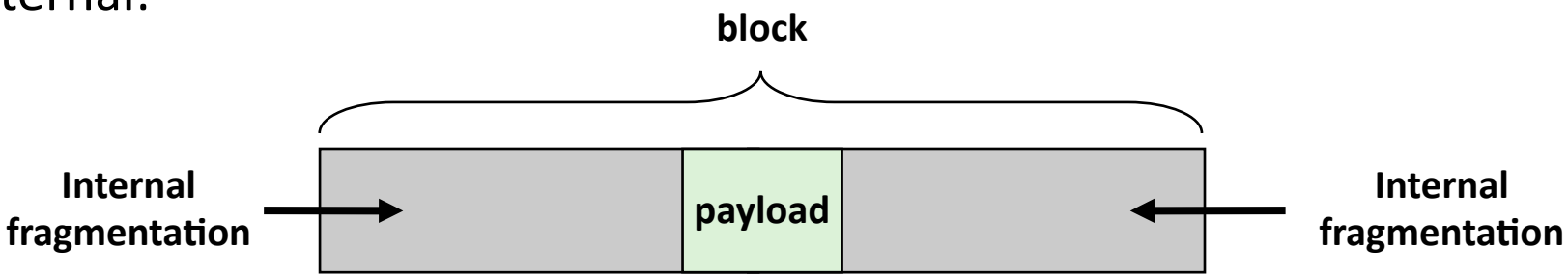
```
p3 = malloc(6)
```

```
free(p2)
```

```
p4 = malloc(2)
```

Last Time: Fragmentation

- Internal:



- External:

```
p4 = malloc(6)
```

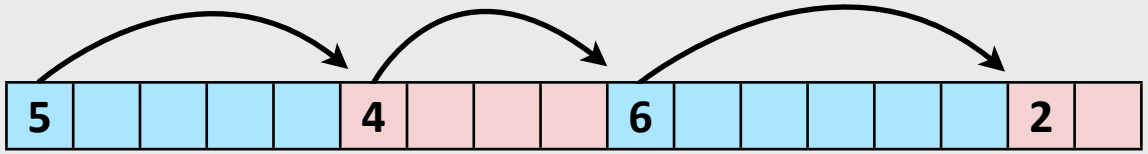
Oops! (what would happen now?)

Today

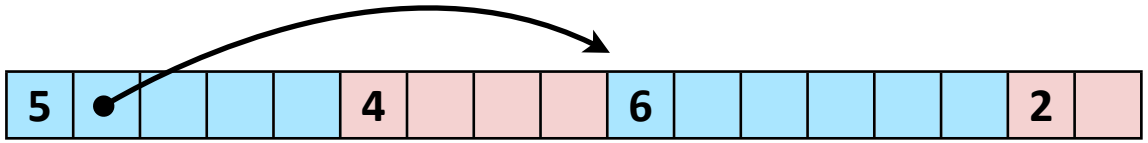
- **Dynamic memory allocation:**
  - Implicit free lists (review)
  - Explicit free lists
  - Segregated free lists
- **Garbage collection**

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

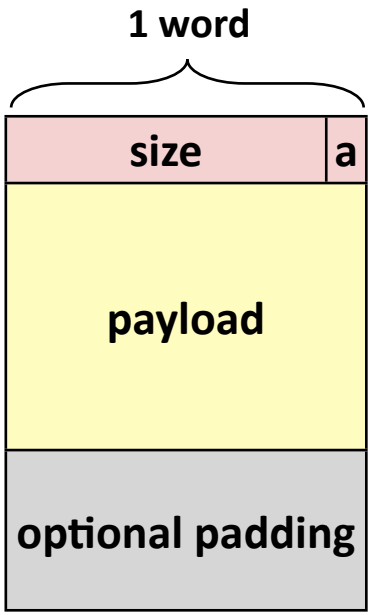


- Method 3: *Segregated free list*
  - Different free lists for different size classes
- Method 4: *Blocks sorted by size*
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Implicit List

- **For each block we need: length, is-allocated?**
  - Could store this information in two words: wasteful!
- **Standard trick**
  - If blocks are aligned, some low-order address bits are always 0
  - Instead of storing an always-0 bit, use it as a allocated/free flag
  - When reading size word, must mask out this bit

Format of allocated and free blocks



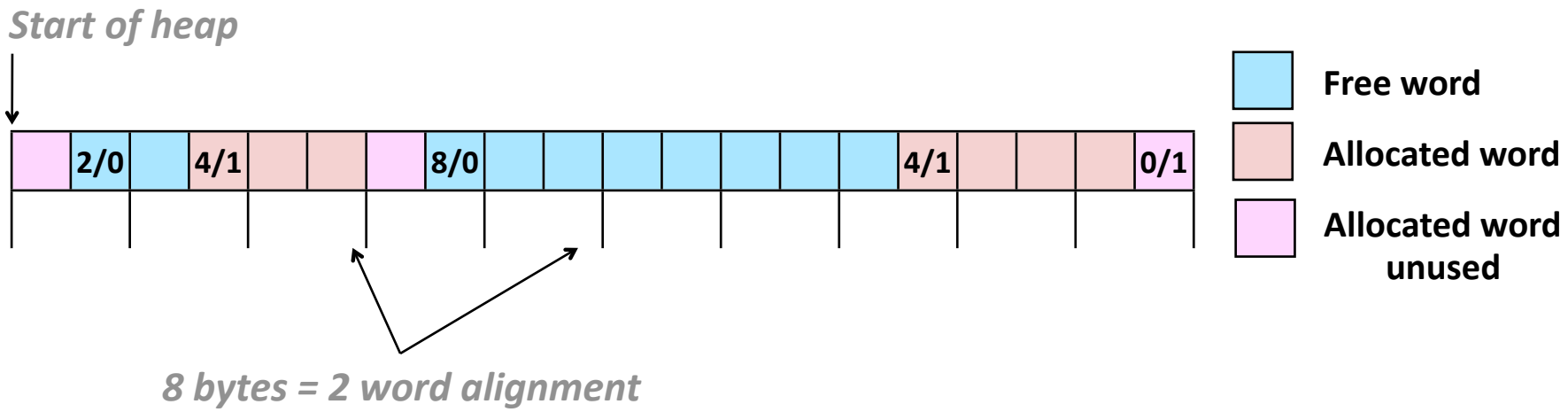
a = 1: allocated block
a = 0: free block

size: block size

payload: application data (allocated blocks only)

Example

Assumptions: word addressed memory, 2 word alignment



- **8-byte alignment**
  - May require initial unused word
  - Causes some internal fragmentation
- **One word (0/1) to mark end of list**
- **Here: block size in words for simplicity**

Implicit List: Finding a Free Block

- ***First fit:***
  - Search list from beginning, choose ***first*** free block that fits: ***(Cost?)***

```
p = start;
while ((p < end) &&      \\ not passed end
       ((*p & 1) ||      \\ already allocated
       (*p <= len)))     \\ too small
  p = p + (*p & -2);     \\ goto next block (word addressed)
```

  - Can take linear time in total number of blocks (allocated and free)
  - In practice it can cause "splinters" at beginning of list
- ***Next fit:***
  - Like first-fit, but search list starting where previous search finished
  - Should often be faster than first-fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse
- ***Best fit:***
  - Search the list, choose the ***best*** free block: fits, with fewest bytes left over
  - Keeps fragments small—usually helps fragmentation
  - Will typically run slower than first-fit

Summary of Key Allocator Policies

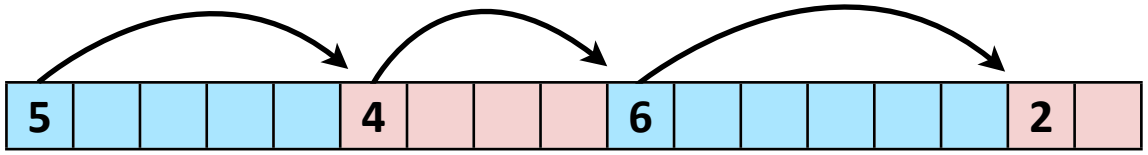
- **Placement policy:**
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
  - ***Interesting observation:*** segregated free lists (next lecture) approximate a best fit placement policy without having to search entire free list
- **Splitting policy:**
  - 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

Implicit Lists: Summary

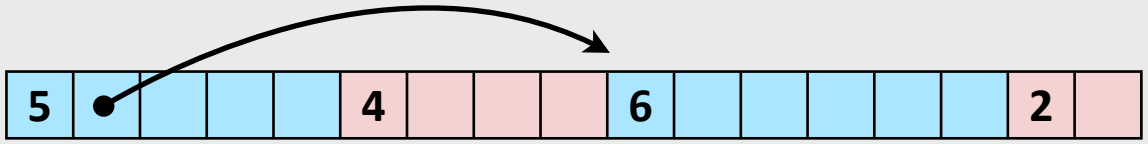
- **Implementation: very simple**
- **Allocate cost: linear time worst case**
- **Free cost: constant time worst case**
  - even with coalescing
- **Memory usage:**
  - will depend on placement policy
  - First-fit, next-fit or best-fit
- **Not used in practice for `malloc()/free()` because of linear-time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

Keeping Track of Free Blocks

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

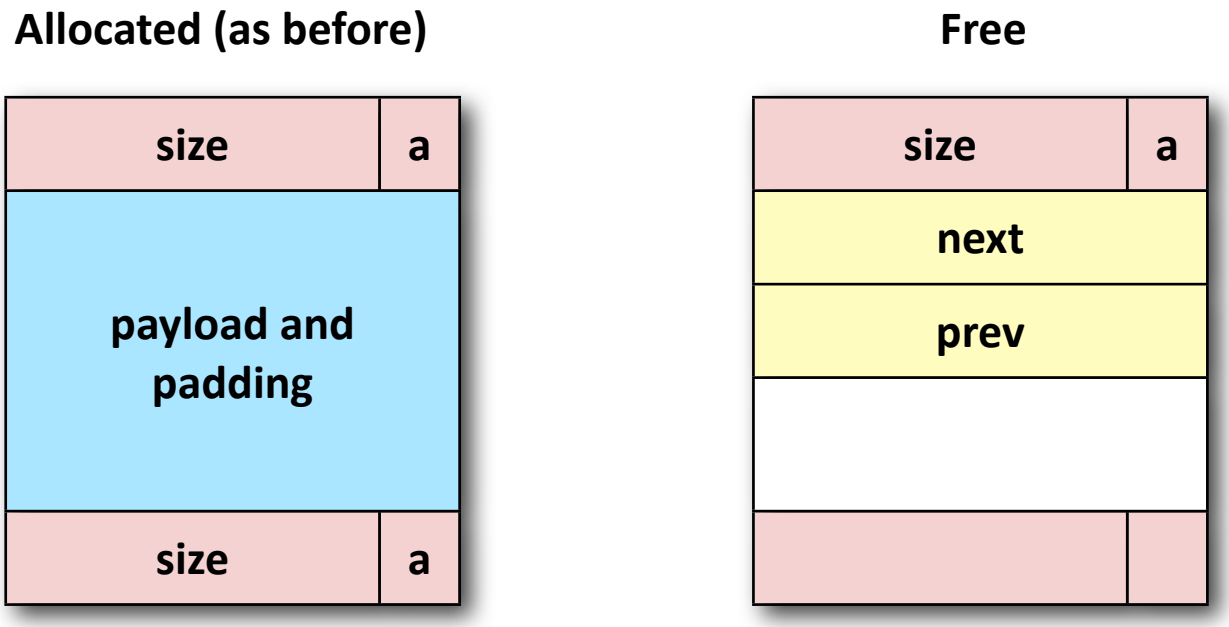


- Method 2: *Explicit free list* among the free blocks using pointers



- 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

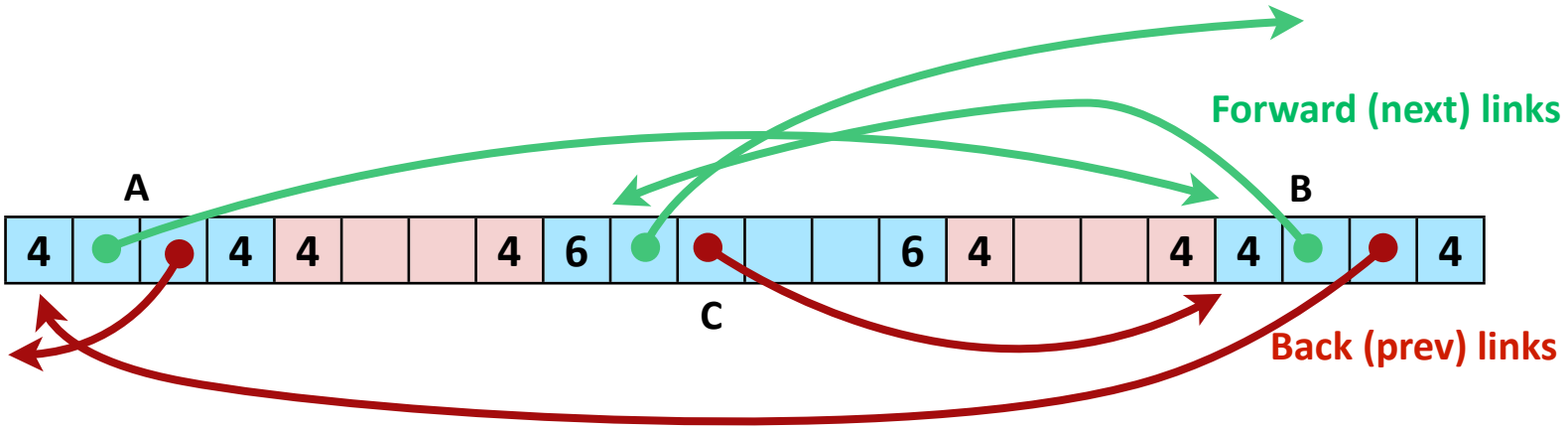


- **Maintain list(s) of *free* blocks, not *all* blocks**
  - The "next" free block could be anywhere
    - So we need to store forward/back pointers, not just sizes
  - Still need boundary tags for coalescing
  - Luckily we track only free blocks, so we can use payload area

Explicit Free Lists

- Logically:

- Physically: blocks can be in any order

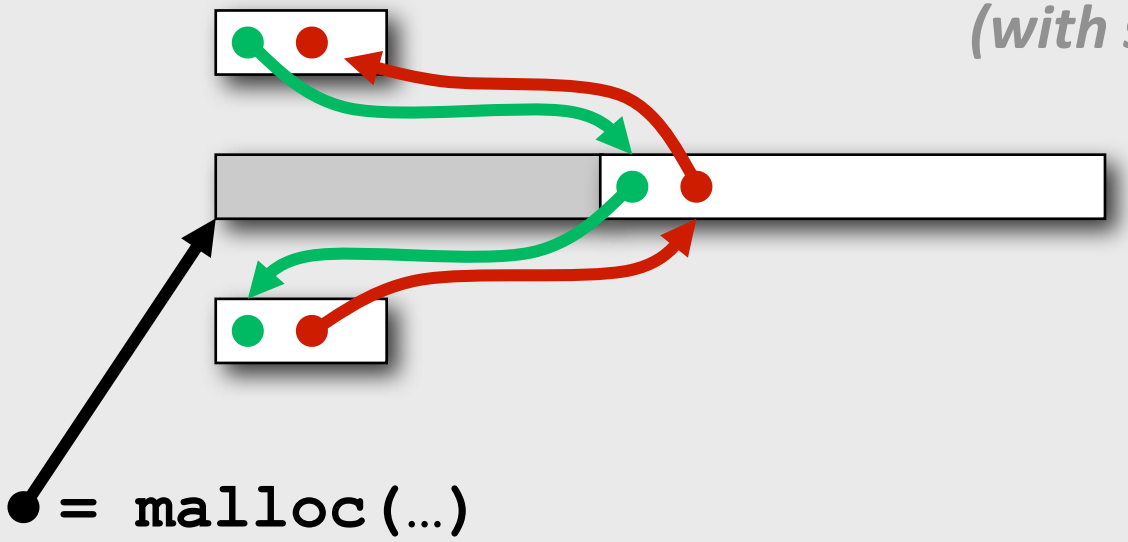


Allocating From Explicit Free Lists

conceptual graphic

Before

After *(with splitting)*

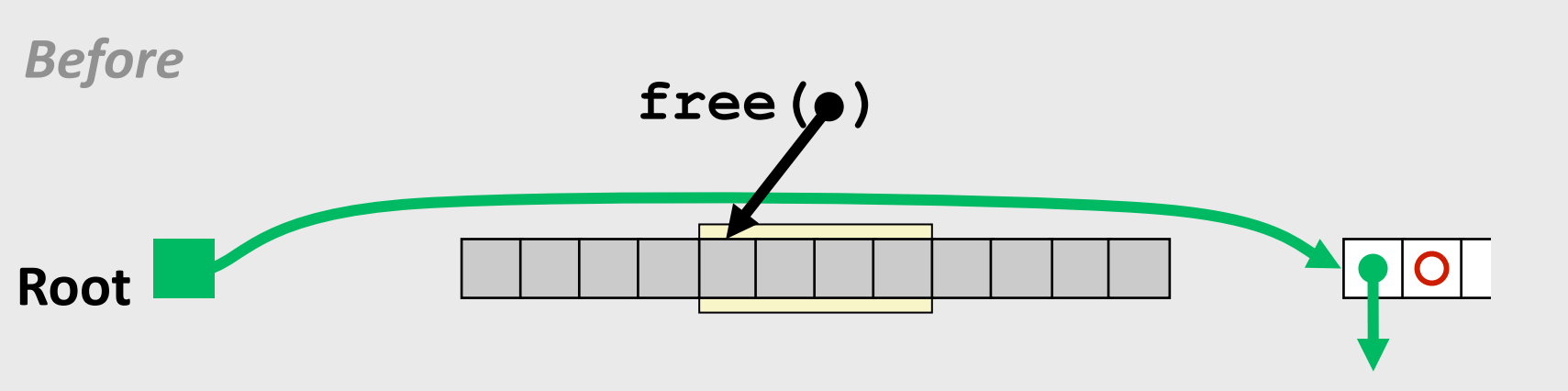


Freeing With Explicit Free Lists

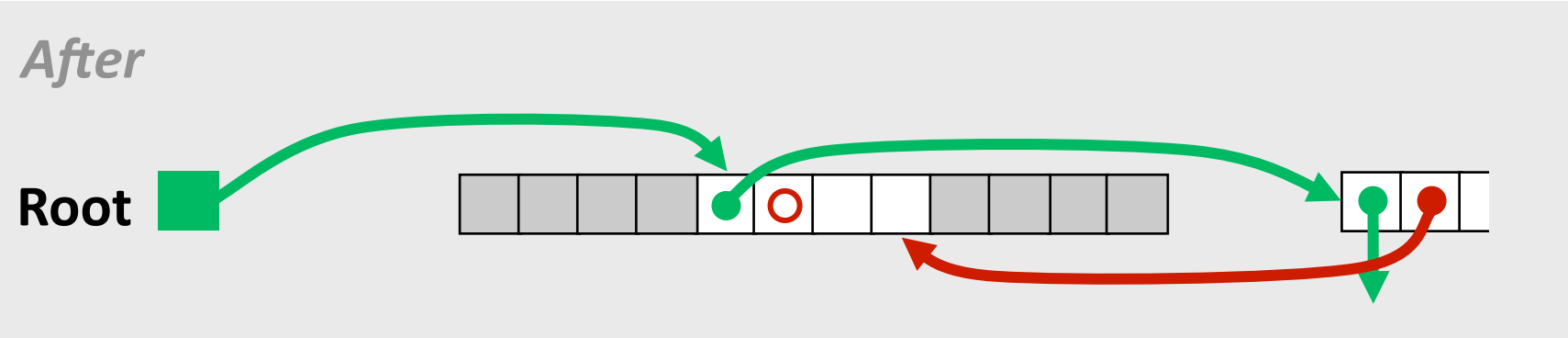
- ***Insertion policy:*** **Where in the free list do you put a newly freed block?**
  - LIFO (last-in-first-out) policy
    - Insert freed block at the beginning of the free list
    - **Pro:** simple and constant time
    - **Con:** studies suggest fragmentation is worse than address ordered
  - Address-ordered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

Freeing With a LIFO Policy (Case 1)

conceptual graphic

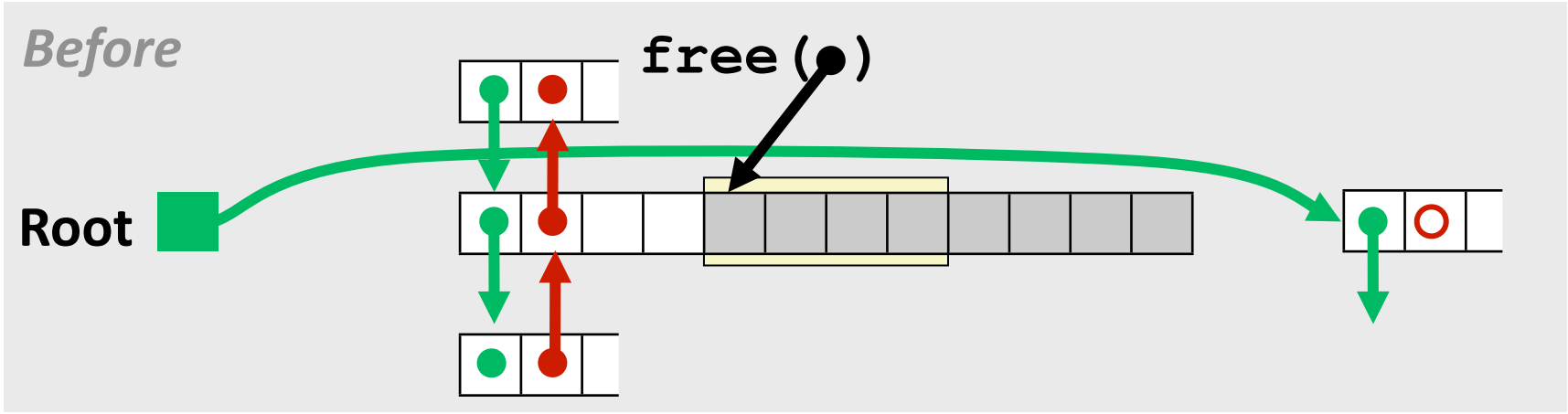


- Insert the freed block at the root of the list

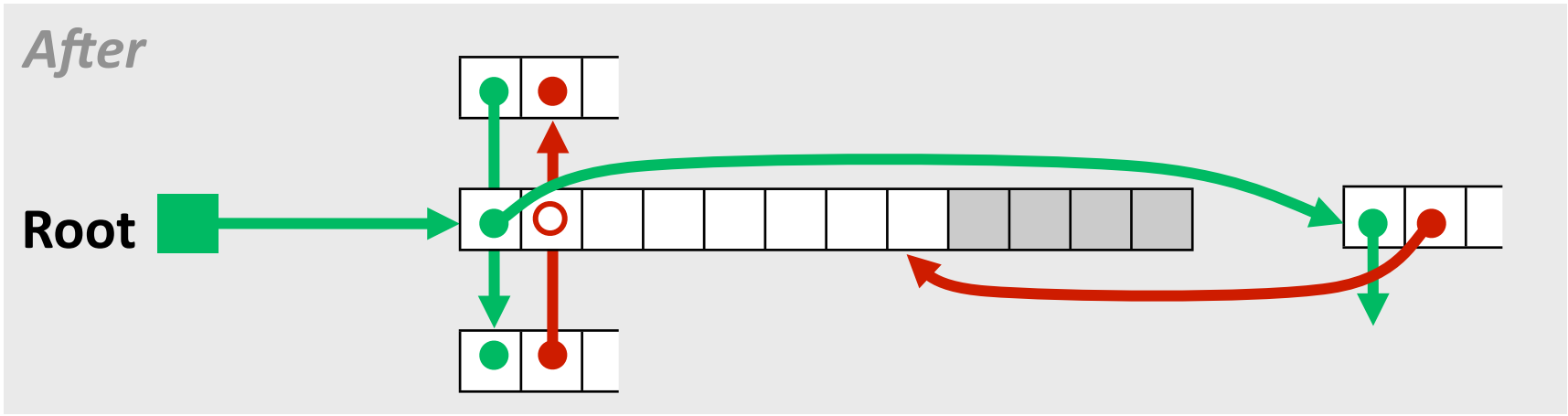


Freeing With a LIFO Policy (Case 2)

conceptual graphic

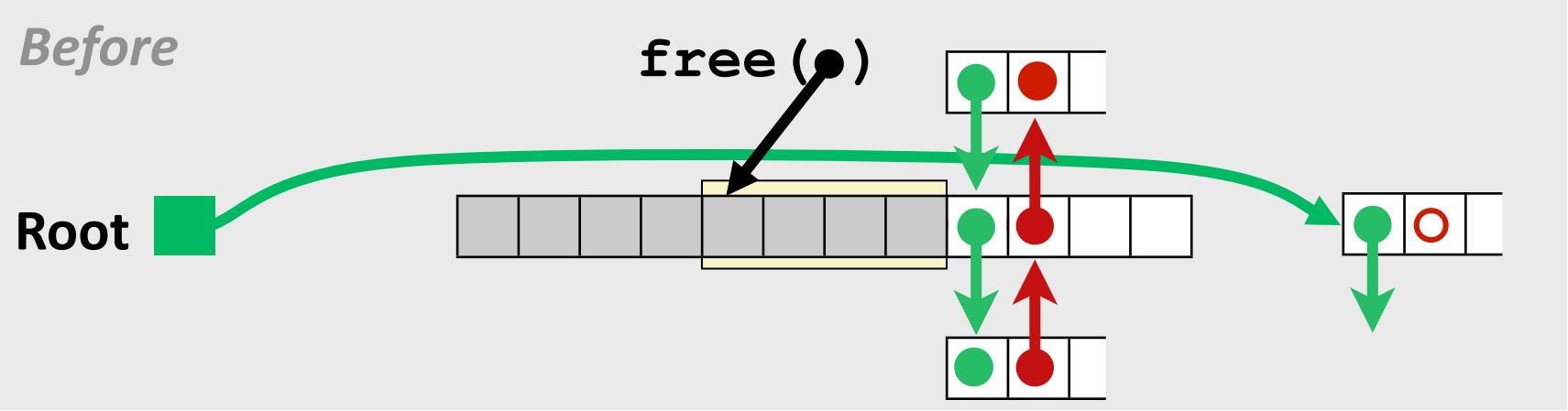


- Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list

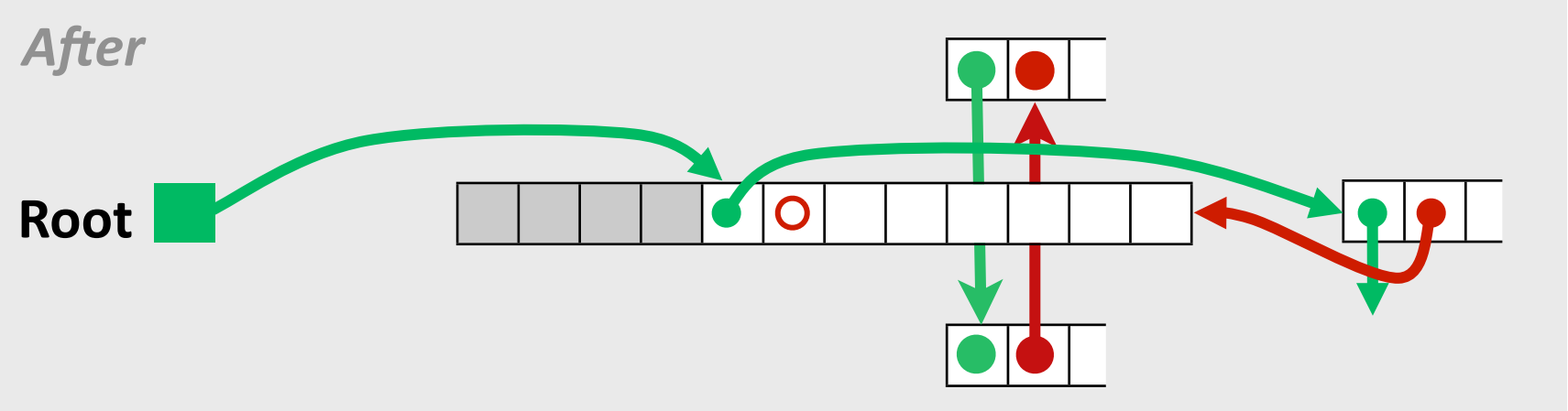


Freeing With a LIFO Policy (Case 3)

conceptual graphic

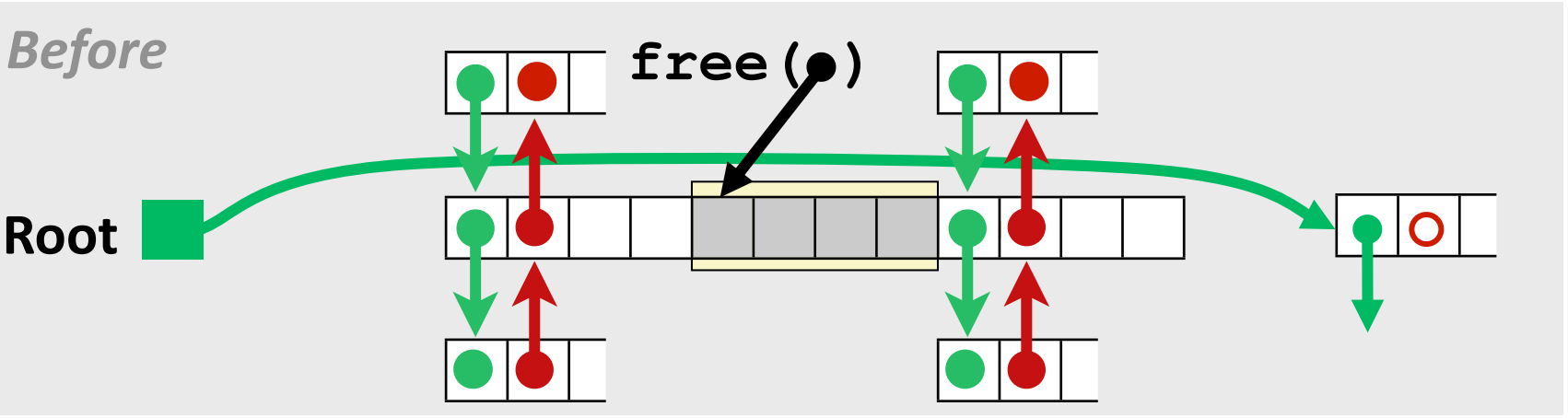


- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list

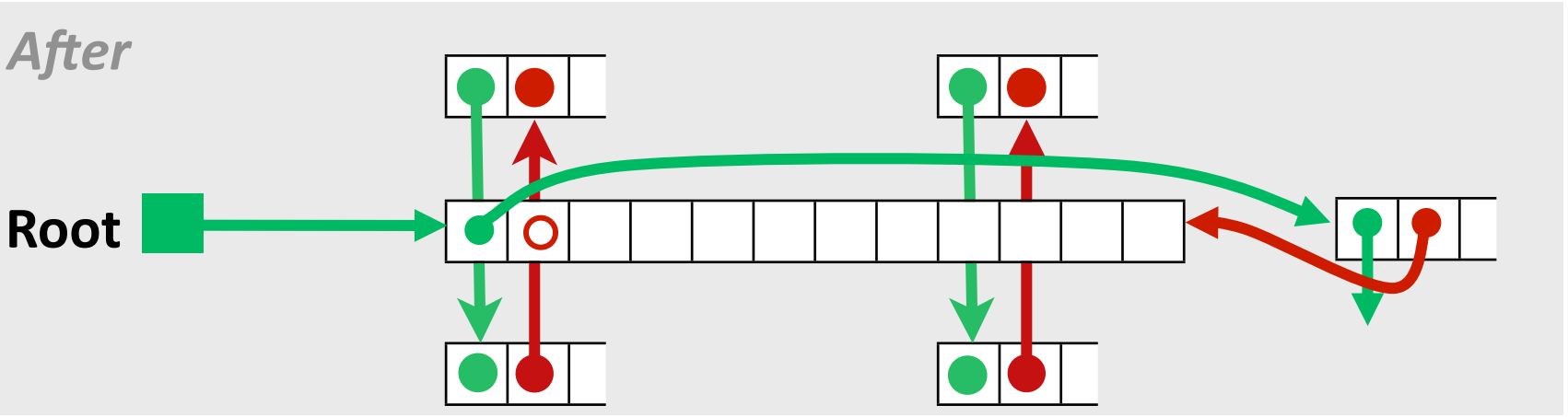


Freeing With a LIFO Policy (Case 4)

conceptual graphic



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



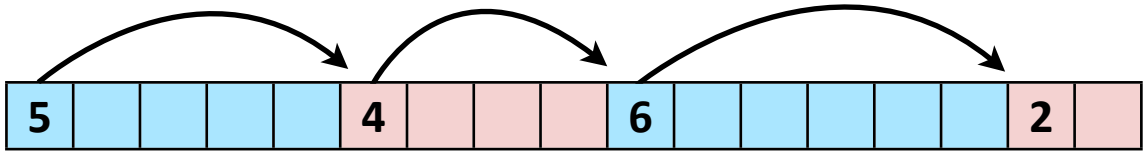
Explicit List Summary

- **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)
    - Does this increase internal fragmentation?

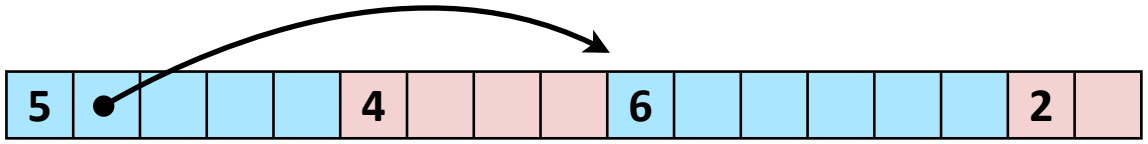
- **Most common use of linked lists is in conjunction with segregated free lists**
  - Keep multiple linked lists of different size classes, or possibly for different types of objects

Keeping Track of Free Blocks

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



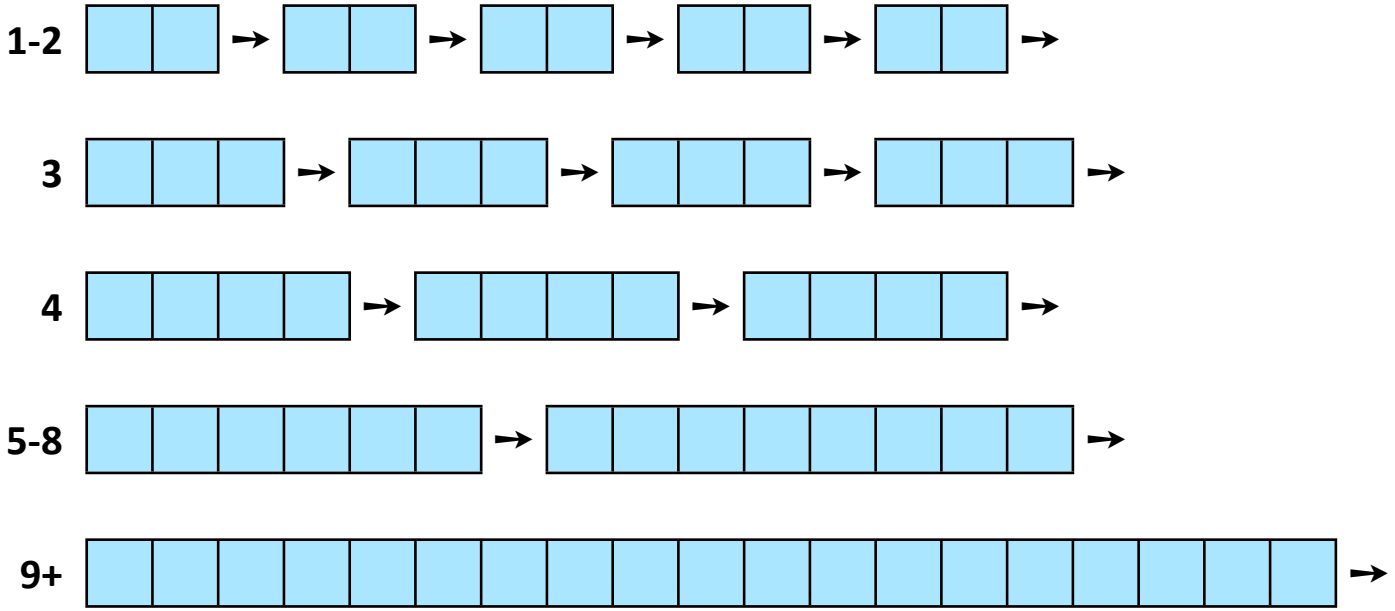
- Method 2: *Explicit free list* among the free blocks using pointers



- 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

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 power-of-two 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 on appropriate list

Seglist Allocator (continued)

- **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

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)

Today

- Dynamic memory allocation:
  - Implicit free lists
  - Explicit free lists
  - Segregated free lists
- **Garbage collection**

Implicit Memory Management: Garbage Collection

- ***Garbage collection:*** **automatic reclamation of heap-allocated storage—application never has to free**

```
void foo() {
    int *p = malloc(128);
    return; /* p block is now garbage */
}
```

- **Common in functional languages, scripting languages, and modern object oriented languages:**
  - Lisp, ML, Java, Ruby, Mathematica
- **Variants ("conservative" garbage collectors) exist for C and C++**
  - However, cannot necessarily collect all garbage

Garbage Collection

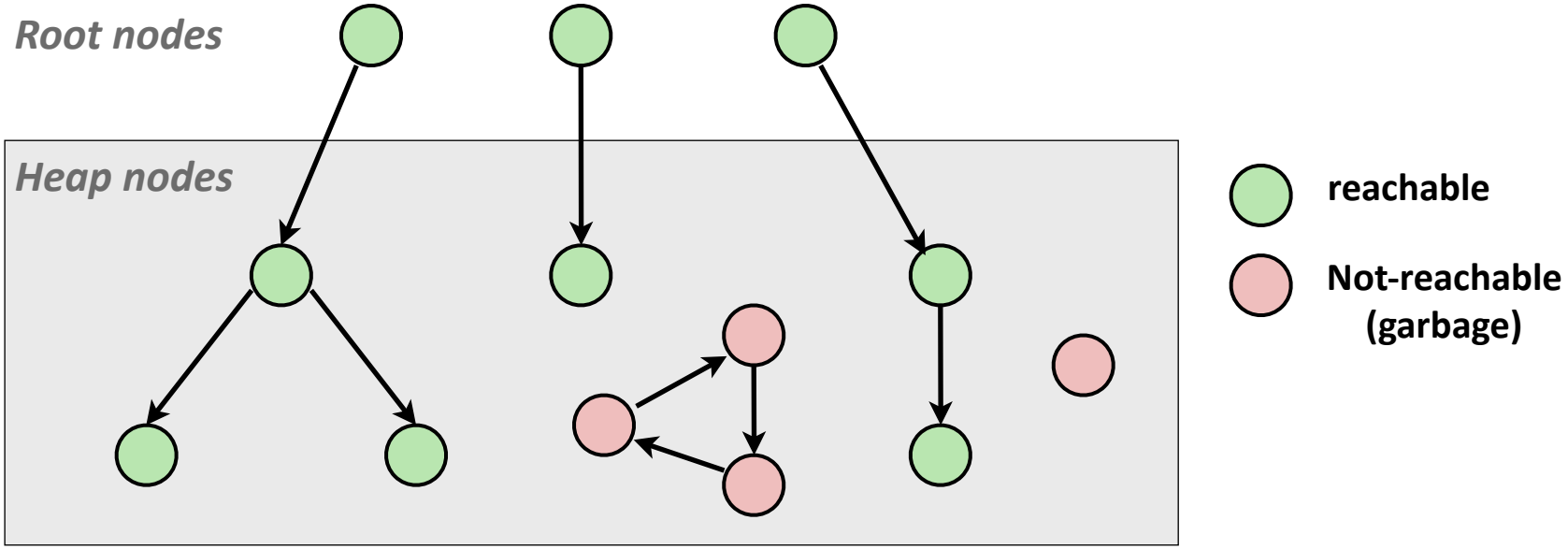
- **How does the memory manager know when a block can be freed?**
  - In general we cannot know what is going to be used in the future since it depends on conditionals
  - But we can tell that certain blocks cannot be used if there are no pointers to them
    - No longer any way to access the contents
- **Must make certain assumptions about pointers**
  - Memory manager can distinguish pointers from non-pointers
  - All pointers point to the start of a block
  - Cannot hide pointers
    (e.g., by coercing them to an int, and then back again)

Classical GC Algorithms

- **Mark-and-sweep collection (McCarthy, 1960)**
  - Does not move blocks (unless you also "compact")
- **Reference counting (Collins, 1960)**
  - Does not move blocks (not discussed)
- **Copying collection (Minsky, 1963)**
  - Moves blocks (not discussed)
- **Generational Collectors (Lieberman and Hewitt, 1983)**
  - Collection based on lifetimes
    - Most allocations become garbage very soon
    - So focus reclamation work on zones of memory recently allocated
- **For more information:**
  Jones and Lin, "Garbage Collection: Algorithms for Automatic Dynamic Memory", John Wiley & Sons, 1996

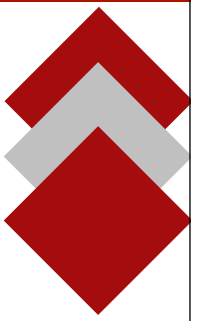
Memory as a Graph

- **We view memory as a directed graph**
  - Each block is a node in the graph
  - Each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called ***root*** nodes (e.g. registers, locations on the stack, global variables)



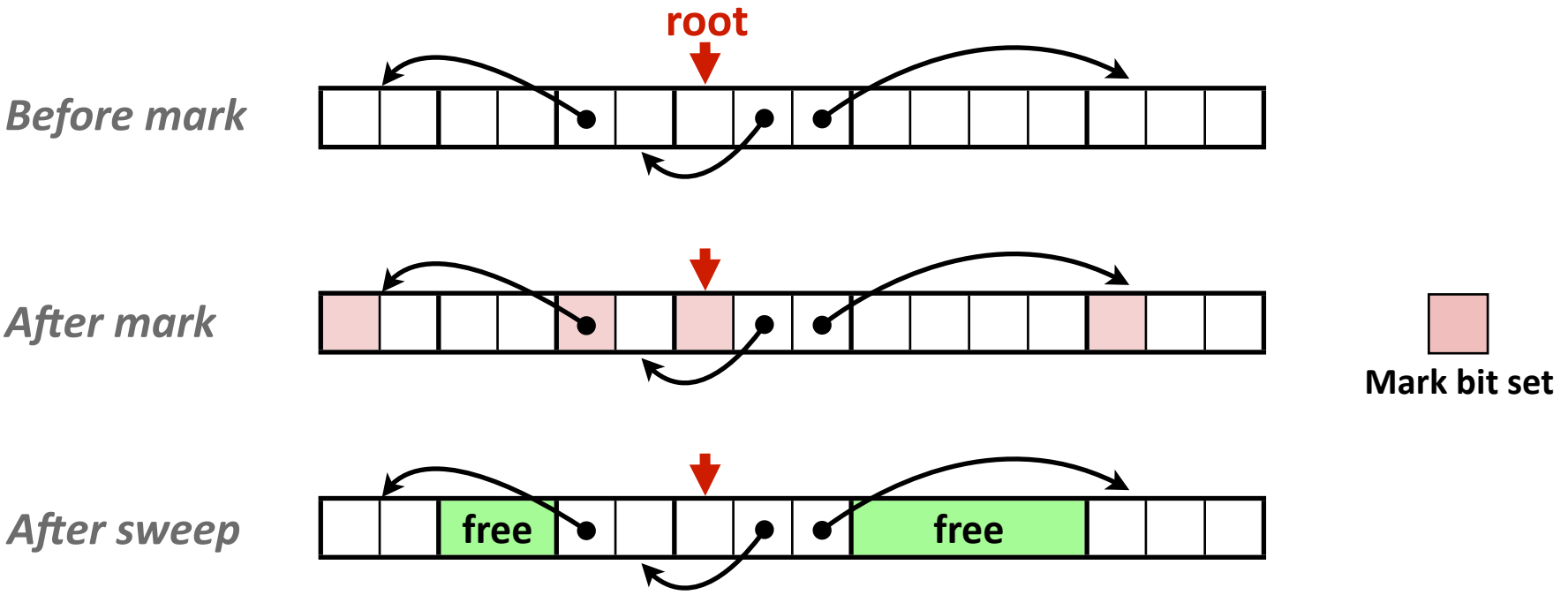
A node (block) is ***reachable*** if there is a path from any root to that node

Non-reachable nodes are ***garbage*** (cannot be needed by the application)



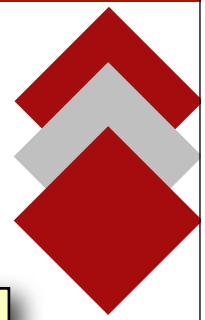
Mark and Sweep Collecting

- **Can build on top of malloc/free package**
  - Allocate using malloc until you "run out of space"
- **When out of space:**
  - Use extra ***mark bit*** in the head of each block
  - **Mark:** Start at roots and set mark bit on each reachable block
  - **Sweep:** Scan all blocks and free blocks that are not marked



Assumptions For a Simple Implementation

- **Application**
  - **`new(n)`:** returns pointer to new block with all locations cleared
  - **`read(b,i)`:** read location **`i`** of block **`b`** into register
  - **`write(b,i,v)`:** write v into location **`i`** of block **`b`**
- **Each block will have a header word**
  - addressed as **`b[-1]`**, for a block **`b`**
  - Used for different purposes in different collectors
- **Instructions used by the Garbage Collector**
  - **`is_ptr(p)`:** determines whether **`p`** is a pointer
  - **`length(b)`:** returns the length of block **`b`**, not including the header
  - **`get_roots()`:** returns all the roots



Mark and Sweep Implementation

Mark using depth-first traversal of the memory graph

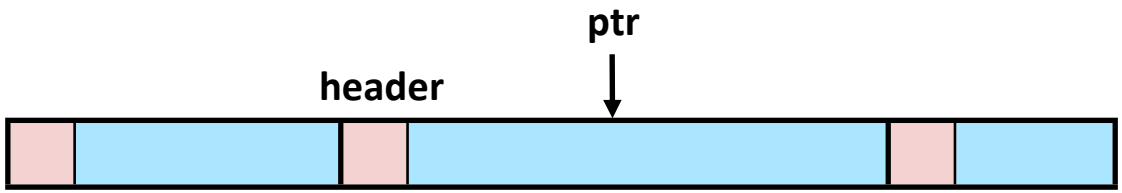
```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // do nothing if not pointer
   if (markBitSet(p)) return;       // check if already marked
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // call mark on all words
     mark(p[i]);                    //   in the block
   return;
}
```

Sweep using lengths to find next block

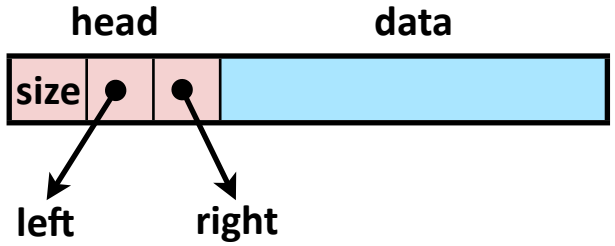
```
ptr sweep(ptr p, ptr end) {
   while (p < end) {
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p);
}
```

Conservative Mark & Sweep in C

- **A "conservative garbage collector" for C programs**
  - `is_ptr()` determines if a word is a pointer by checking if it points to an allocated block of memory
  - But, in C pointers can point to the middle of a block



- **So how to find the beginning of the block?**
  - Can use a balanced binary tree to keep track of all allocated blocks (key is start-of-block)
  - Balanced-tree pointers can be stored in header (use two additional words)



Left: smaller addresses
Right: larger addresses

Summary

- Dynamic Memory Allocation (2 of 2)
- Garbage Collection

- Next Time: System-Level I/O
- Unix I/O
- File sharing
- I/O redirection
- Standard I/O