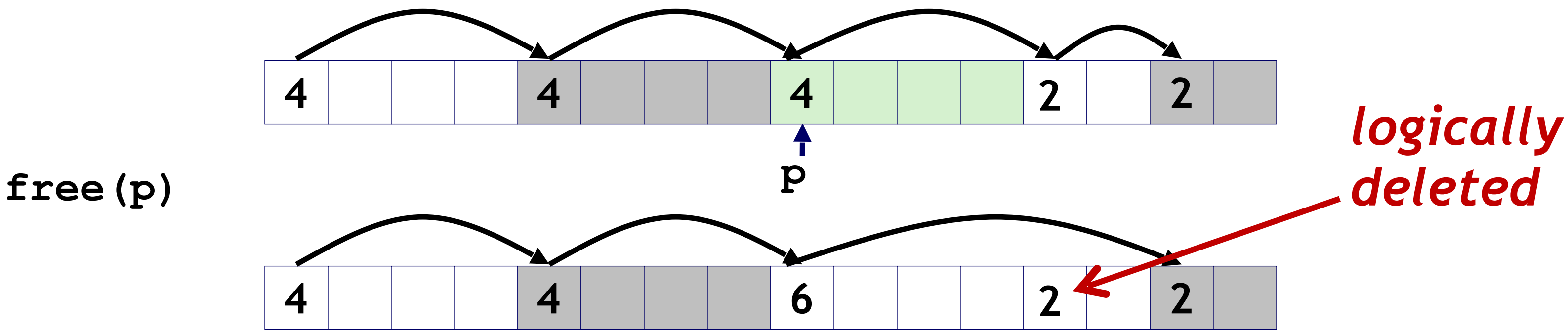
+ Dynamic Memory Allocation *cont't*

Implicit List: Coalescing

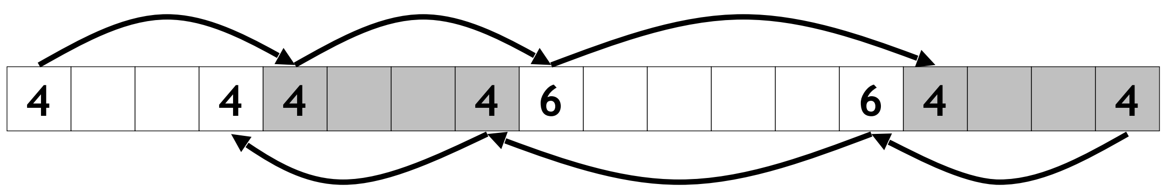
- **Join (coalesce) with next/previous blocks, if they are free**
  - Coalescing with next block



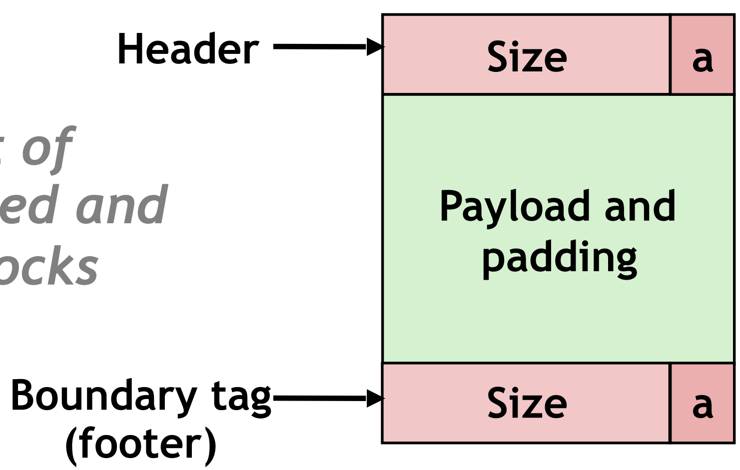
  - But how do we coalesce with previous block?

+ Implicit List: Boundary Tags (footers)

- **Boundary tags [Knuth '73] https://en.wikipedia.org/wiki/Donald_Knuth**
  - Replicate size/allocated word at "bottom" (end) of free blocks
  - Allows us to traverse the "list" backwards, but requires extra space



Format of allocated and free blocks

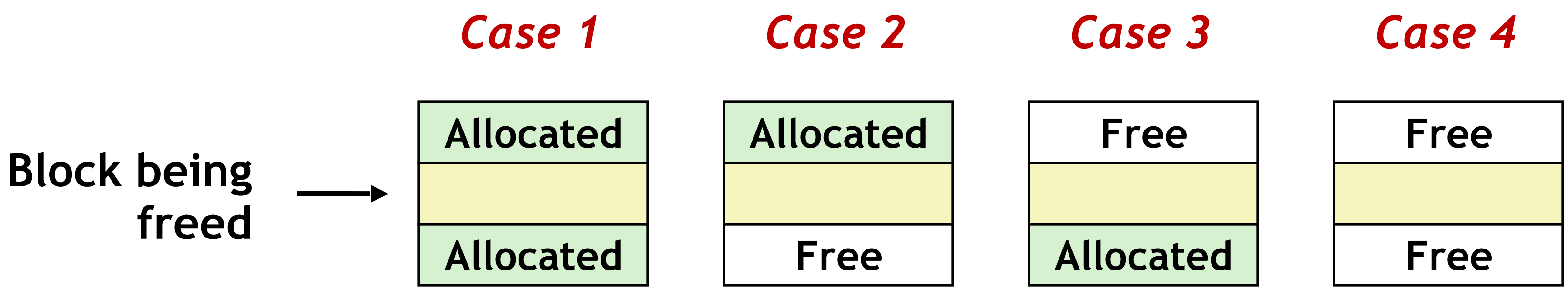


a = 1: Allocated block
a = 0: Free block

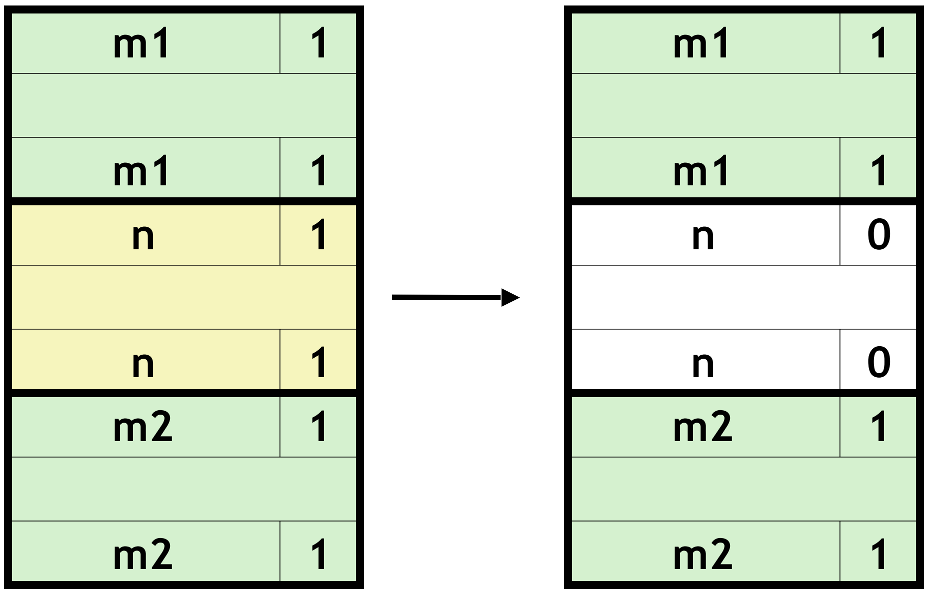
Size: Total block size

Payload: Application data (allocated blocks only)

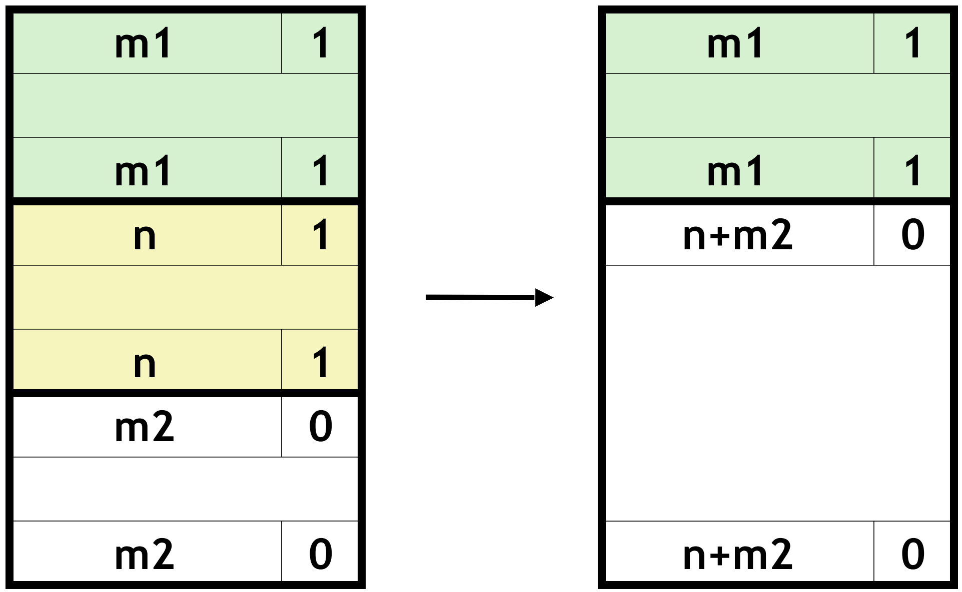
Constant Time Coalescing



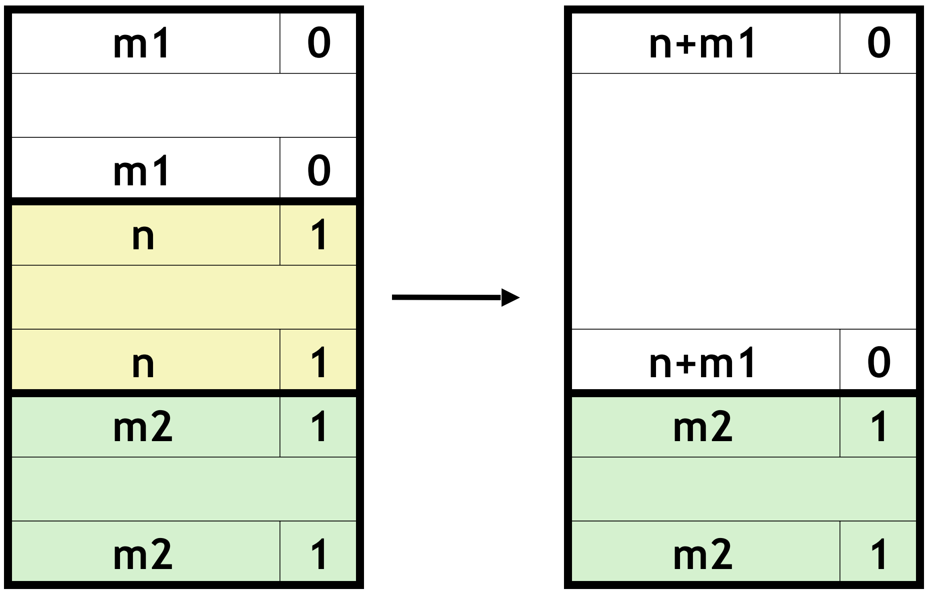
Constant Time Coalescing (Case 1)



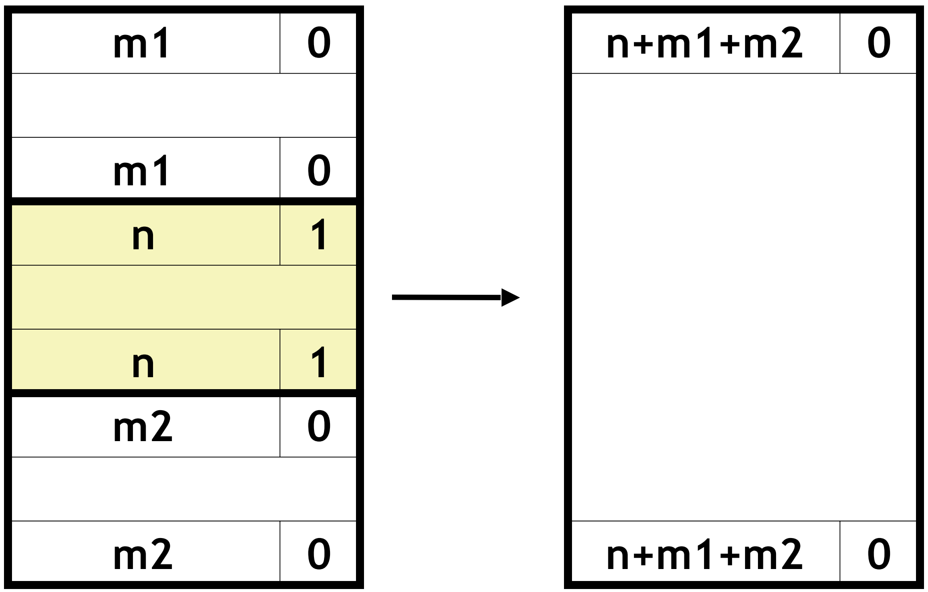
Constant Time Coalescing (Case 2)



Constant Time Coalescing (Case 3)



Constant Time Coalescing (Case 4)



+ Disadvantages of Boundary Tags

- **Internal fragmentation**
  - Again we are trading space for time, utilization for throughput
- **Can it be optimized?**
  - Which blocks need the footer tag?

+ Disadvantages of Boundary Tags

- **Internal fragmentation**
  - Again we are trading space for time, utilization for throughput
- **Can it be optimized?**
  - Which blocks need the footer tag?
  - Only free blocks!
- **So how do we know if the last word in the previous block is a boundary tag or not, after all its just bits back there!**

+ Disadvantages of Boundary Tags

- **Internal fragmentation**
  - Again we are trading space for time, utilization for throughput
- **Can it be optimized?**
  - Which blocks need the footer tag?
  - Only free blocks!
- **So how do we know if the last word in the previous block is a boundary tag or not, after all its just bits back there!**
  - We can use one of those low order bits in the header to indicate the allocation status of the previous block.

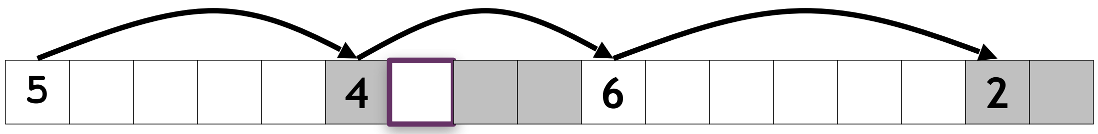
Implicit Lists: Summary

- **Implementation: very simple**
- **Allocate cost:**
  - linear time
- **Free cost:**
  - constant time (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**
- **Concepts of splitting and boundary tag coalescing are general to all allocators**

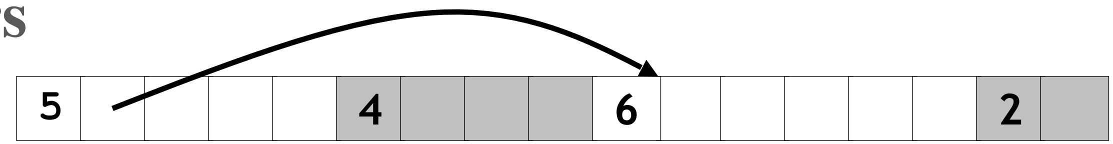
Explicit Free Lists

Keeping Track of Free Blocks

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

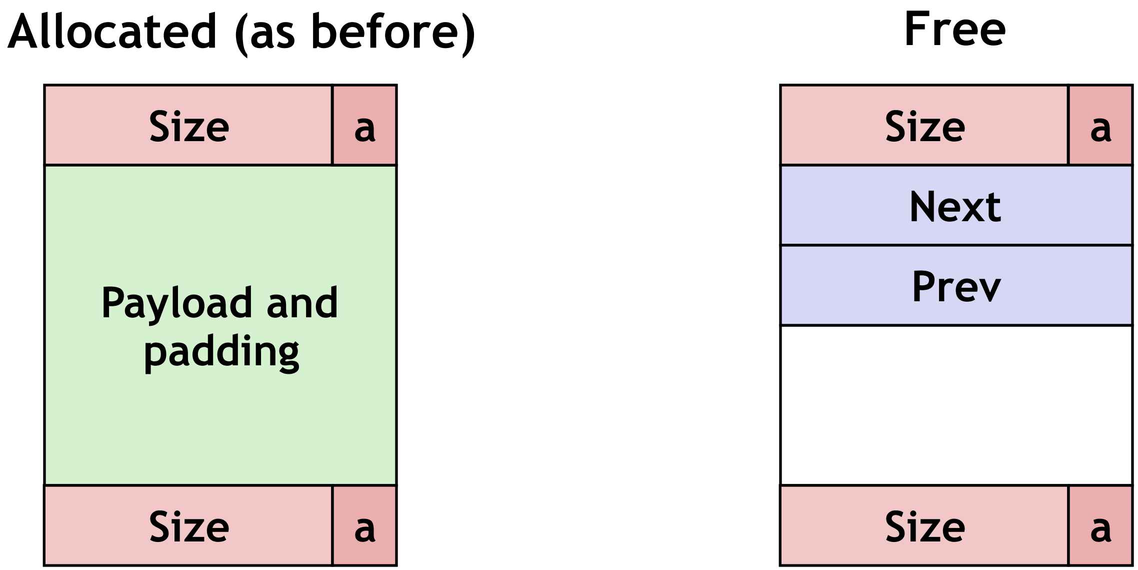


- **Method 2: Explicit free list among the free blocks using pointers**



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

+ Explicit Free Lists

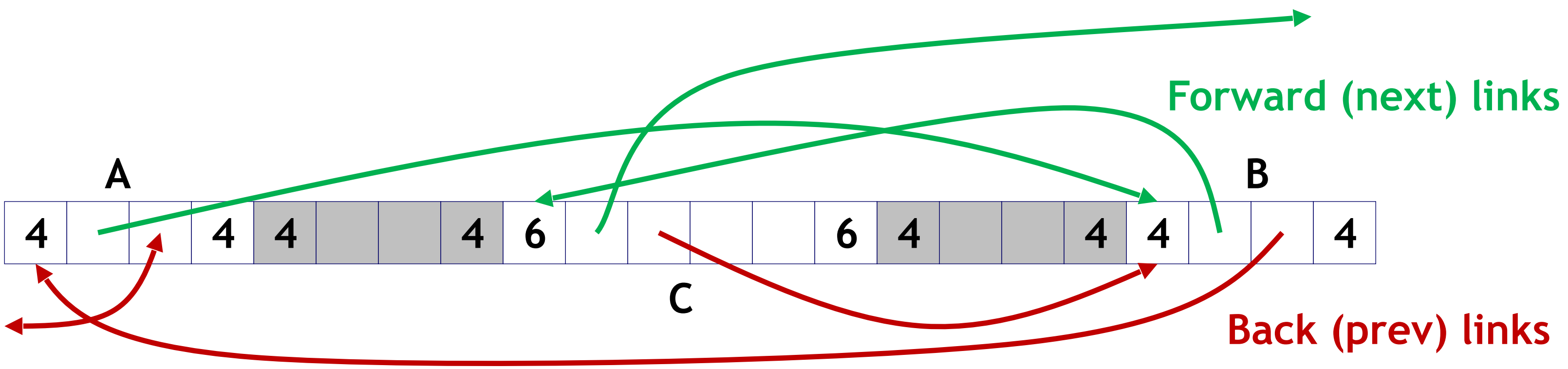


- **Maintain list(s) of *free* blocks, not *all* blocks**
  - The "next" free block could be anywhere
    - So we need to store forward/back pointers, not just sizes
  - Still need boundary tags for coalescing
  - Luckily we track only free blocks, so we can use payload area

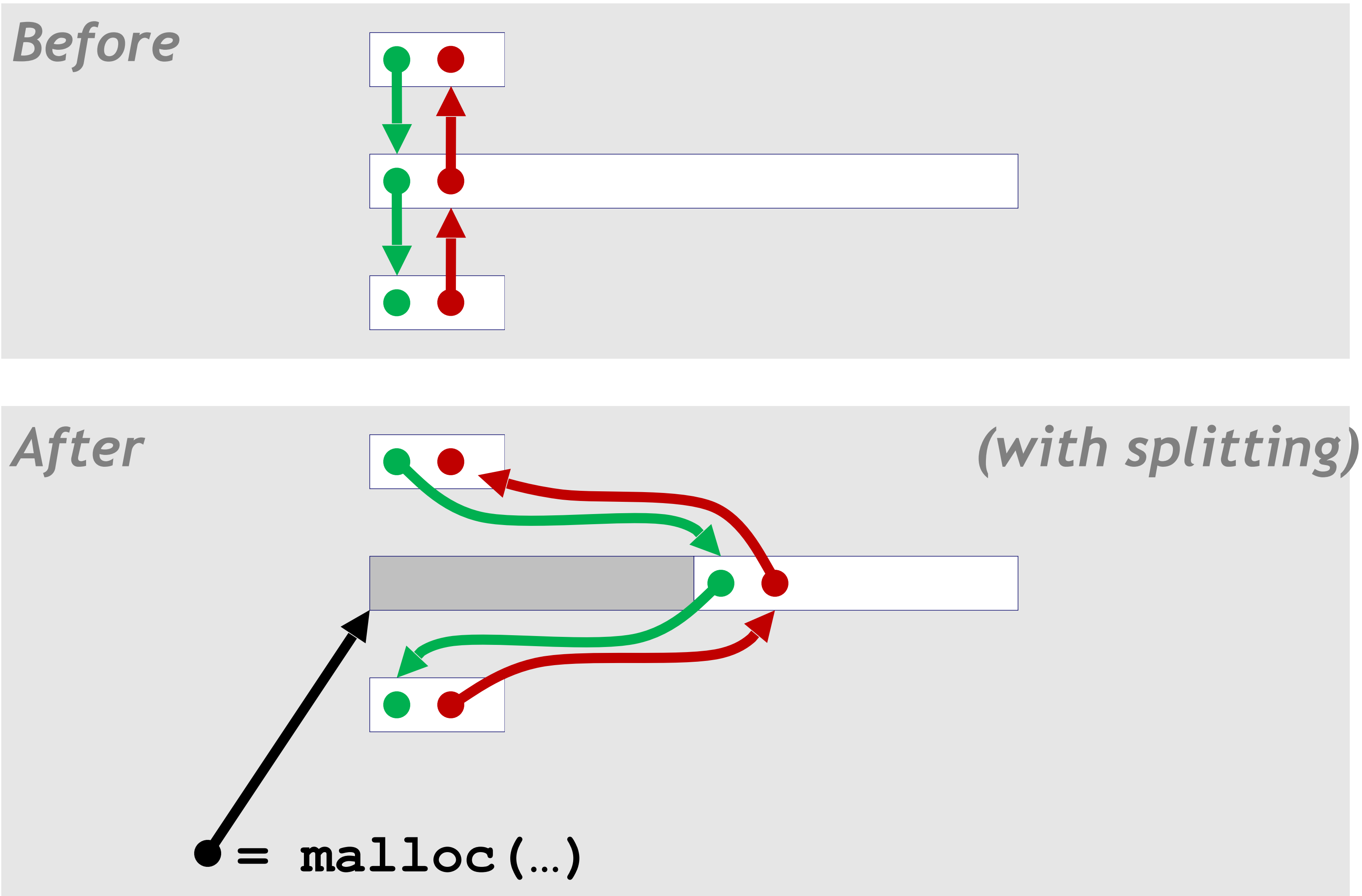
Explicit Free Lists

- **Logically:**

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



+ Allocating From Explicit Free Lists



+ 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

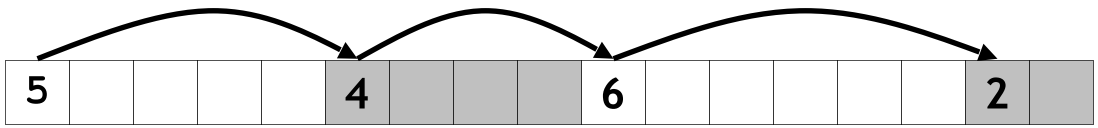
+ Explicit List Summary

- **Comparison to implicit list:**
  - Allocate is linear time in number of *free* blocks instead of all
    - Much faster when good heap utilization is maintained
  - 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 per block)
    - Does this increase internal fragmentation?
- **Most common use of explicit free lists is in context of segregated free lists**

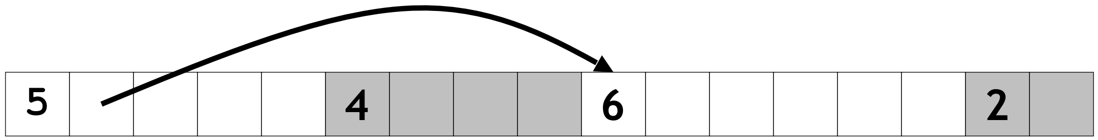
Segregated Free Lists

Keeping Track of Free Blocks

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



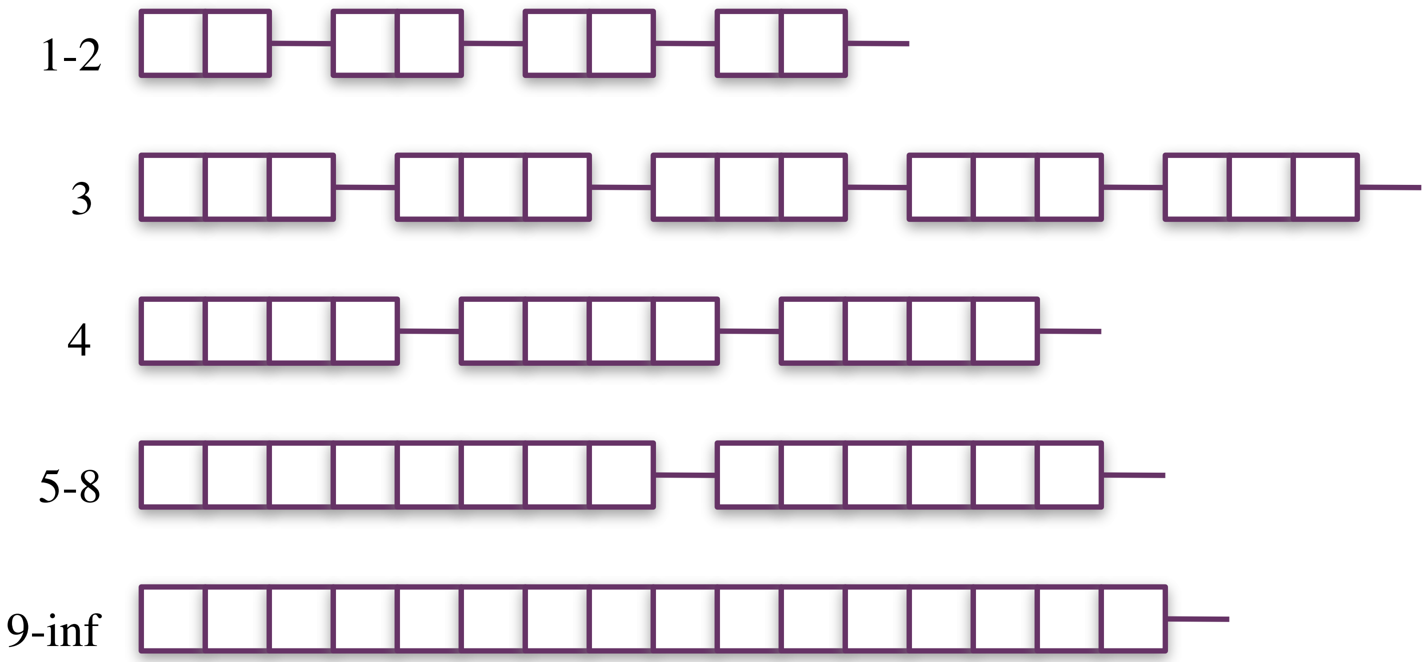
- **Method 2: Explicit list among the free blocks using pointers**



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

Size Classes

- 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

+ Segregated List 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 some size class.

+ Segregated List Allocator *con't*

- **To free a block:**
  - Coalesce and place on appropriate list
    - Common technique is to place it at the end of the list, so constant time.
- **Advantages of segregated list allocators**
  - *Higher throughput*
    - Size class lists are a smaller search space
  - *Better memory utilization*
    - *First-fit* search of segregated free list approximates a *best-fit* search of entire heap.

Garbage Collection

+ Garbage Collection

- **Automatic free'ing of heap-allocated storage no longer in use**

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

- **Common in many languages:**
  - Python, Ruby, Java, Go, ML, Lisp….
- **Variants exist for C and C++**
  - However, pointer semantics make it impossible for it be perfect

```
void main() {
   int* p = malloc(128);
   int* q = p+32;  //Confuses collector
   // other code
   return;
}
```

+ Garbage

- **How does the memory manager know when memory can be freed?**
  - We can tell that certain blocks cannot be used if there are no pointers to them

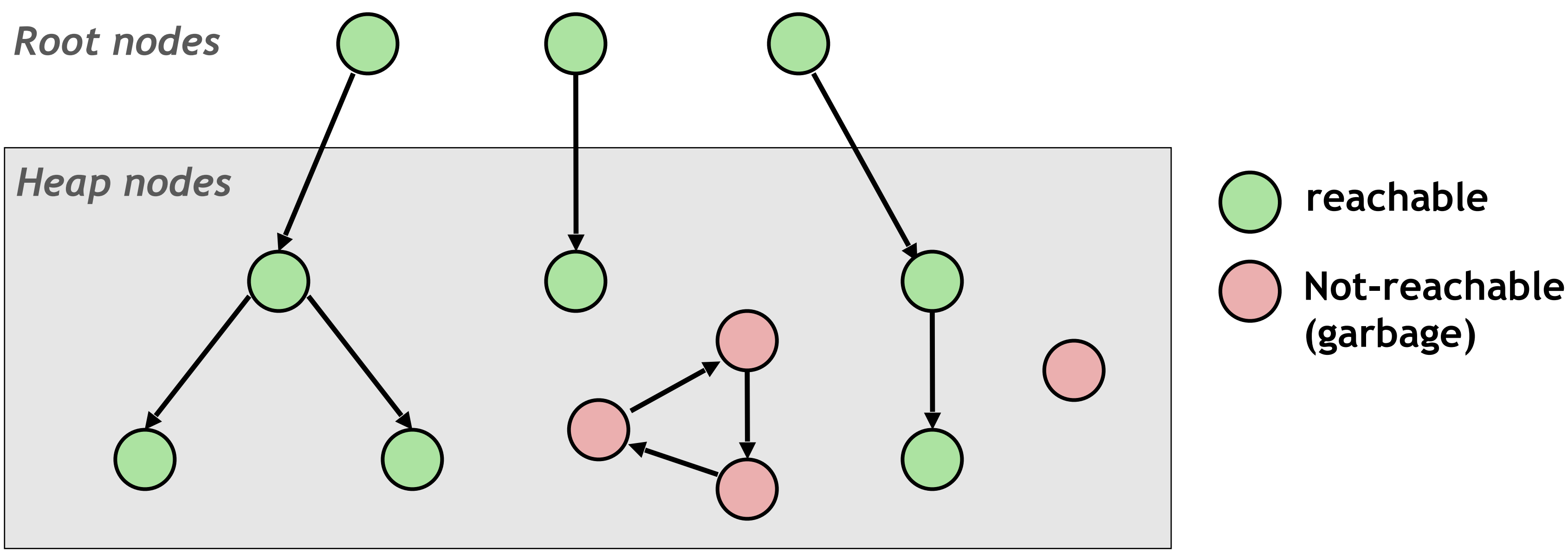
- **Must make certain assumptions about pointers**
  - Memory manager can distinguish pointers from non-pointers
  - All pointers point to the start of a block
  - Cannot hide pointers
    (e.g., by coercing them to an int, and then back again)
  - All these things are true in Java but not in C

+ 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)

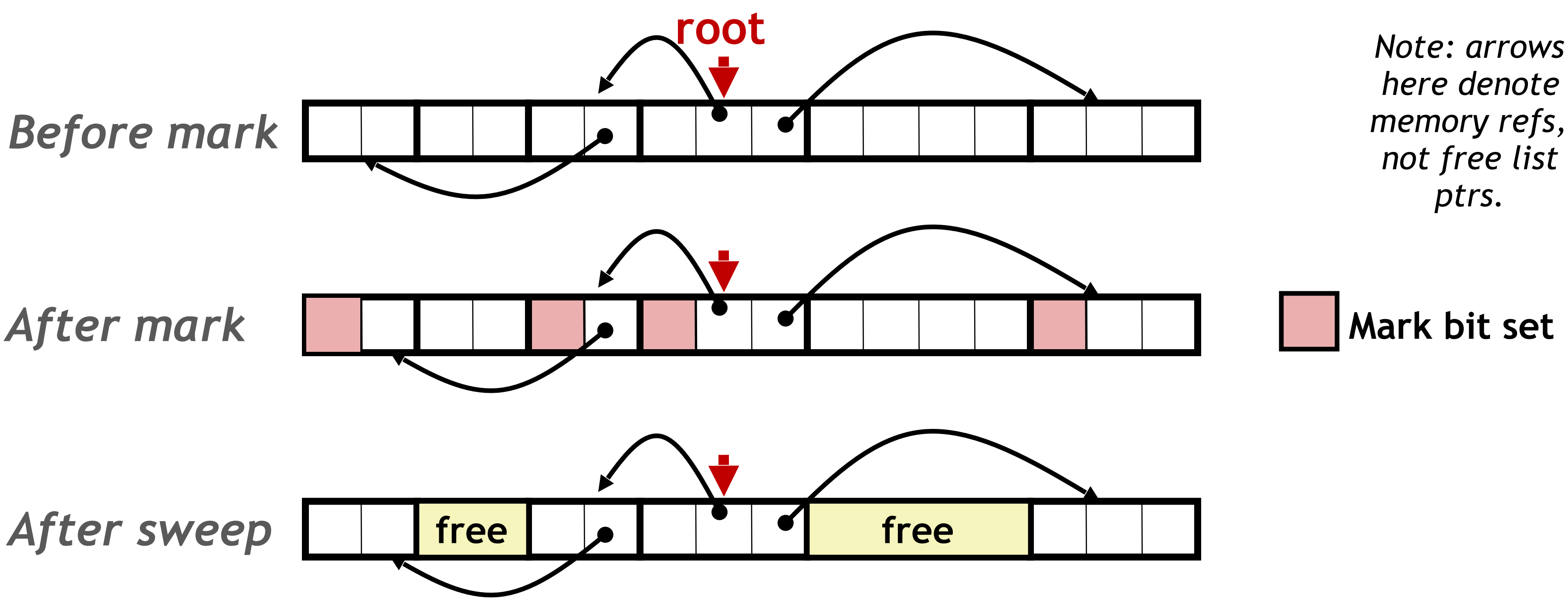


A node (block) is *reachable* if there is a path from any root to that node.

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



+ Mark and Sweep (cont.)

Mark using depth-first traversal of the memory graph

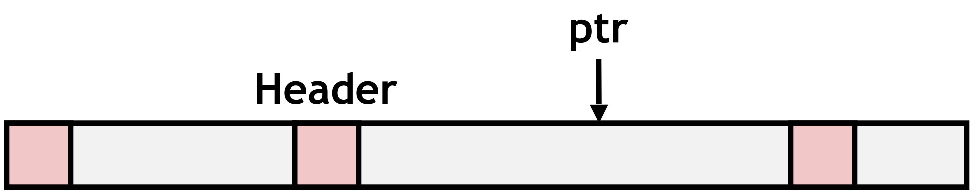
```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // do nothing if not pointer
   if (markBitSet(p)) return;       // check if already marked
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // call mark on all words
     mark(p[i]);                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

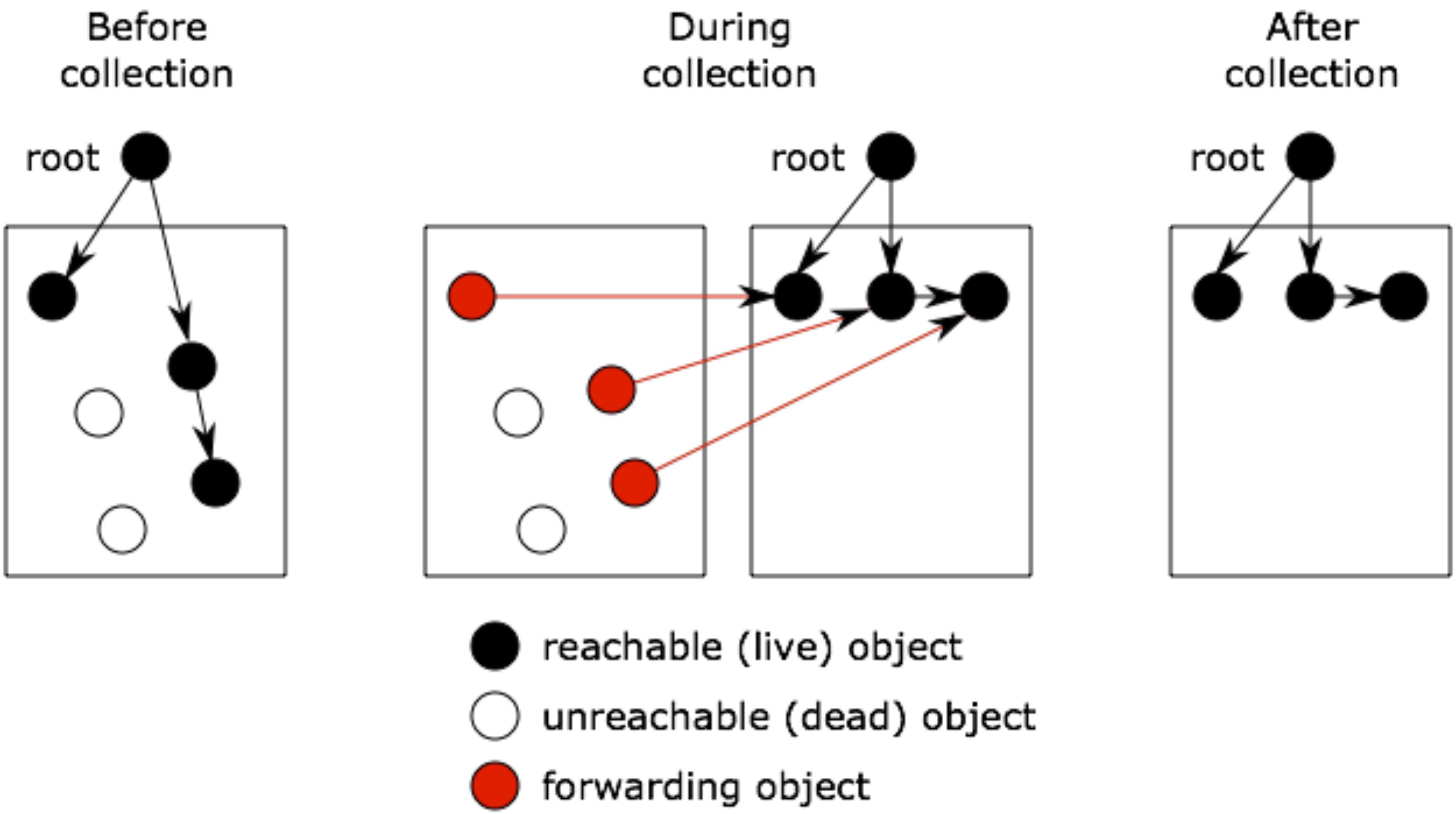


- **So how to find the beginning of the block?**
  - Keep a balance tree where the key is the starting address of every allocation.
  - Search tree and see if some pointer falls within the key+size range of addresses.
  - Could yield false positives! Hence "conservative".

+ True Collection in Java: Copying

- **Java uses a different technique than ‘Mark & Sweep’ called ‘Copying’**
- **The heap is split into two parts: FROM space and TO space**
- **Objects are allocated in the FROM space**
  - When FROM is full, collection begins
- **During traversal, each reachable object is copied to TO space**
  - When traversal is done, all live objects in are in TO space
- **Now the spaces are flipped, FROM becomes TO and vice versa**
  - Everything in FROM gets freed.

+ True Collection in Java: Copying *con't*



+ True Collection in Java: Generational

- **A variant of 'Copying'.**

- ***Infant mortality* or the *generational hypothesis* is the observation that, in most cases, young objects are much more likely to die than old objects. Why?**
  - Objects that live for a long time tend to make up core program data structures and will probably live until the end of the programs life.

- **It turns out that the vast majority of data in typical programs (between 92 and 98 percent according to various studies), *die young*.**
  - Moreover, most variables are short-lived.

+ True Collection in Java: Generational *con't*

- **Generational GCs exploit this 'generational hypothesis'**
- **Instead of just two heaps (FROM and TO), we have several signifying 'generations' of objects.**
  - Younger generations collected more frequently than older generations (because younger generations will have more garbage to collect)
  - When a generation is traversed, live objects are copied to the next-older generation
  - When a generation fills up, it is garbage collected.

+ Urban Performance Legends

- **"Garbage collection will never be as efficient as manual memory management." - Snooty C Programmer**
  - In a way, those statements are right -- automatic memory management is not as fast -- it's often considerably faster.
  - Explicit allocators deal with blocks of memory *one at a time*, whereas the garbage collection approach tends to deal with memory management in large batches, yielding more opportunities for optimization.

+ Urban Performance Legends *con't*

- **Allocation in JVMs was not always so fast -- early JVMs indeed had poor allocation and garbage collection performance.**
  - A lot has happened since the JDK 1.0 days; the introduction of generational collectors in JDK 1.2 has greatly improved performance.
- **As a result, for most objects, the garbage collection cost is -- zero.**
  - This is because a copying collector does not need to visit or copy dead objects, only live ones. So objects that become garbage shortly after allocation contribute no workload to the collection cycle.