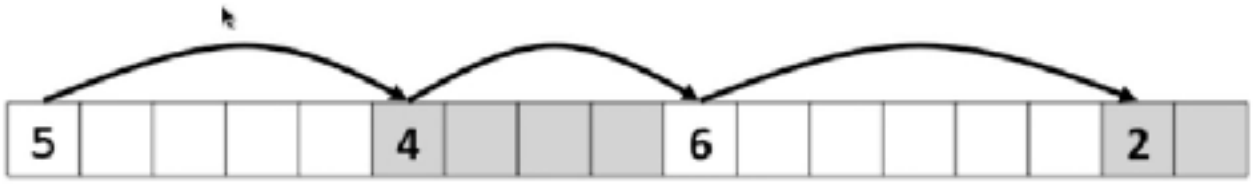
# Dynamic Memory Allocators *con't:* Explicit Free Lists

# Keeping Track of Free Blocks

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

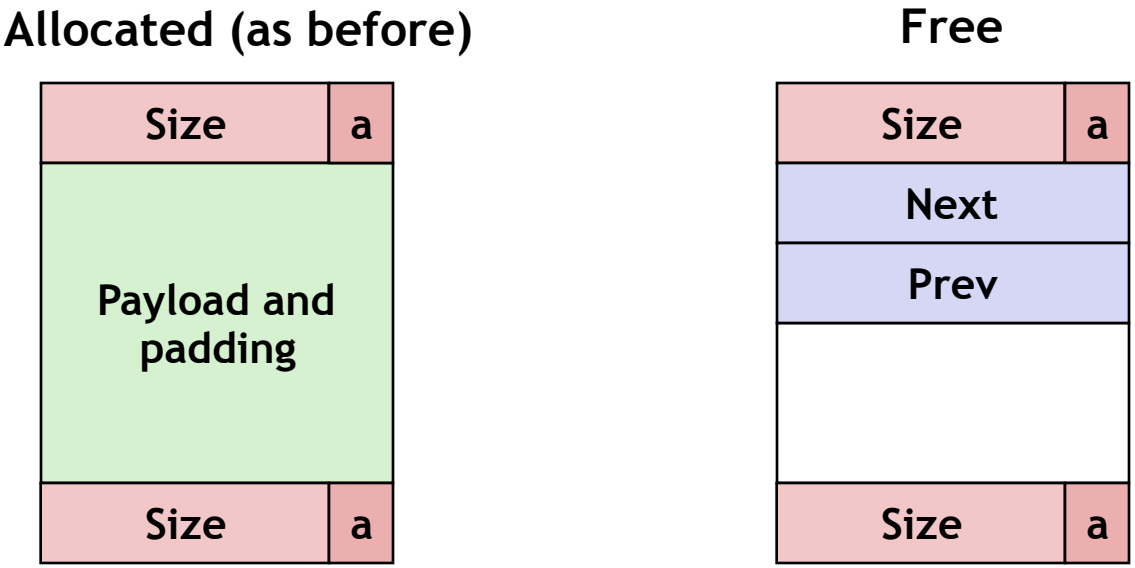



- **<u>Method 2</u>: Explicit list among the free blocks using pointers**


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

- *(1 covered last class)*

# Explicit Free Lists

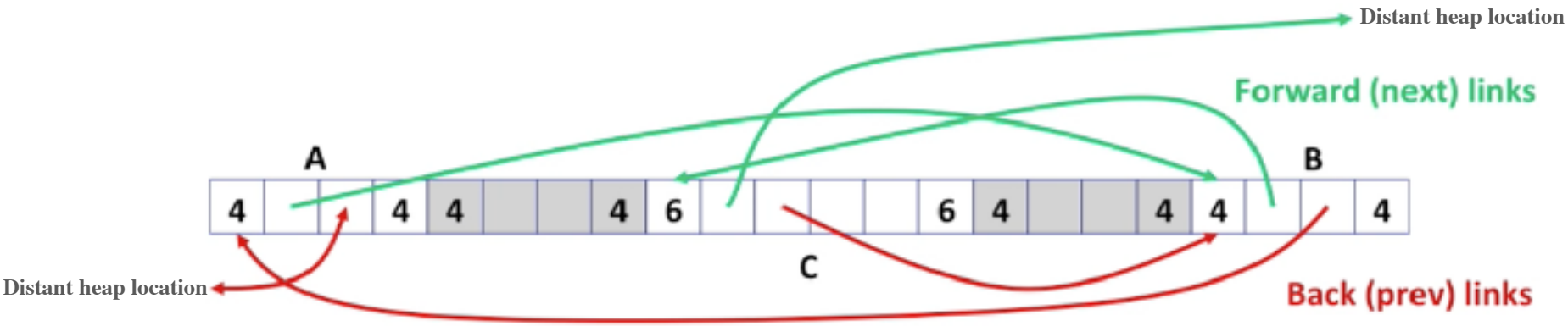



- **Maintain list(s) of *free* blocks, not *all* blocks**
  - The "next" free block could be anywhere
    - So we need to store forward/back pointers, not just sizes
  - Still need boundary tags for coalescing
  - Luckily we track only free blocks, so we can use payload area

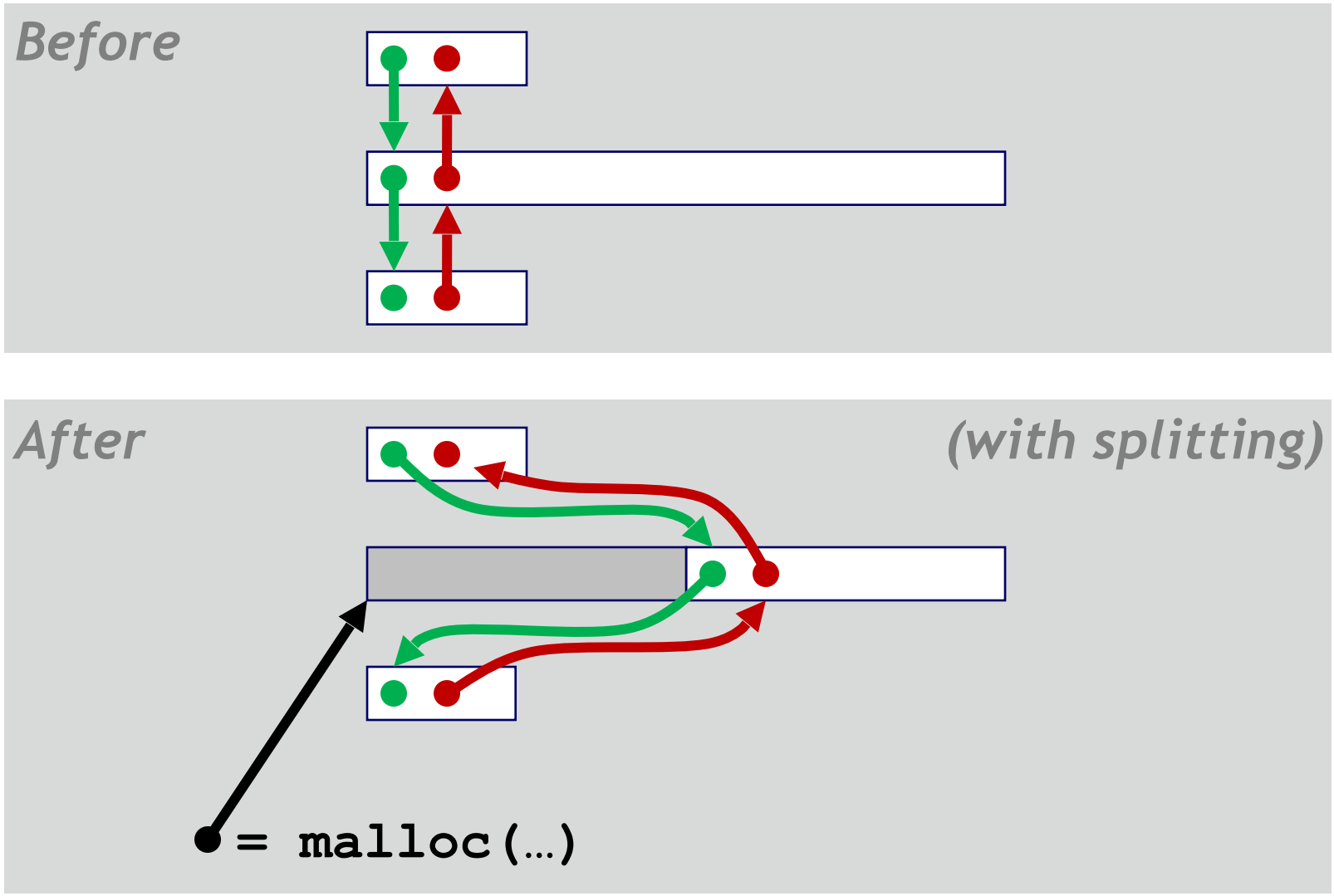
# Physical vs Logical Block Adjacency

- **Logically:**

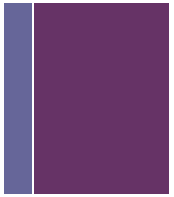

- **Physically: blocks can be in any order**



# Allocating With Explicit Free Lists



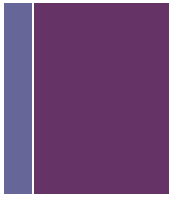
# Freeing With Explicit Free Lists



- **Insertion policy:** Where in the free list do you put a newly freed block? (Remember the free list is a doubly-linked list)
- **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



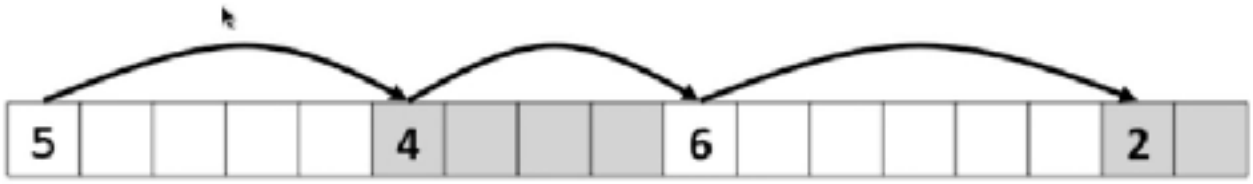
- **Comparison to implicit list:**
  - *Allocate is linear time* in number of *free* blocks instead of all
    - Much faster when good heap utilization is maintained
  - *Slightly more complicated* allocate and free
  - Some *extra space required* for pointers in free blocks
    - Does this increase fragmentation?
      - *How do pointers affect minimum block size?*
      - *What effect might that have?*
- **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**

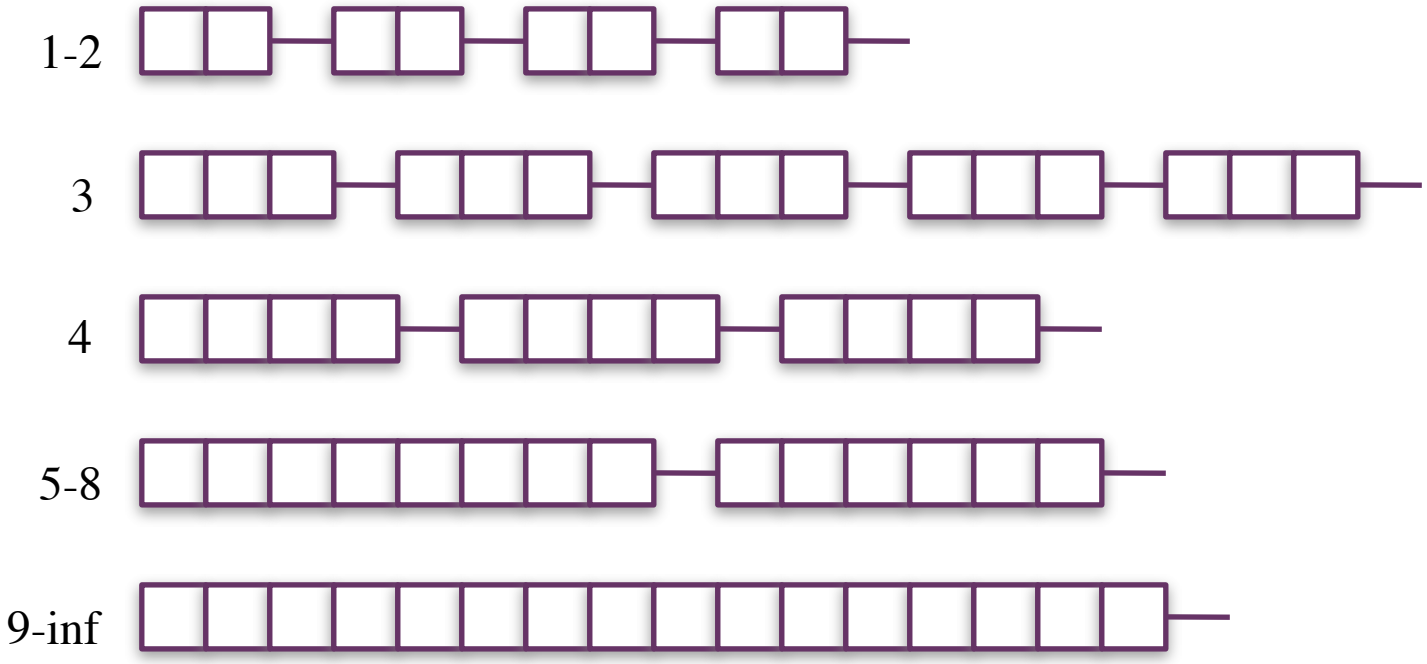



- **<u>Method 2</u>: Explicit list among the free blocks using pointers**


- **<u>Method 3</u>: Segregated free list**
  - Multiple explicit free lists for different *size classes*

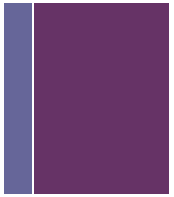
# Size Classes

- **Each *size class* of blocks has its own free list**



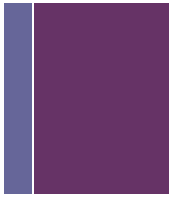

- **Often have separate classes for each small size**
- **For larger sizes: One class for each two-power size**

# 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
  - 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 prepend to front 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 comes close to a 'best-fit' search of entire heap

# Implicit Allocators (Garbage Collection)

# Garbage Collection

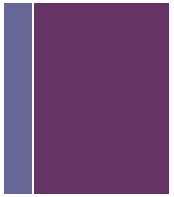
- **Automatic free'ing of heap-allocated storage no longer 'live'**

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

- **Common in many languages:**
  - Python, Ruby, Java, Go, ML, Lisp….
- **"Conservative" solutions exist for C and C++**
  - Pointer semantics make it impossible for it be perfect

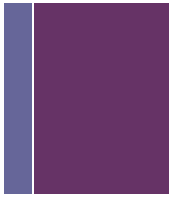
```
int* main() {
   int* p = malloc(128);
   int* q = p+8;
   // other code
   return p;  // Is q garbage?
}
```

# 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
- **Some properties of pointers must be true…**
  - 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 back again)
- **The above assumptions are true in Java (and others), not in C**

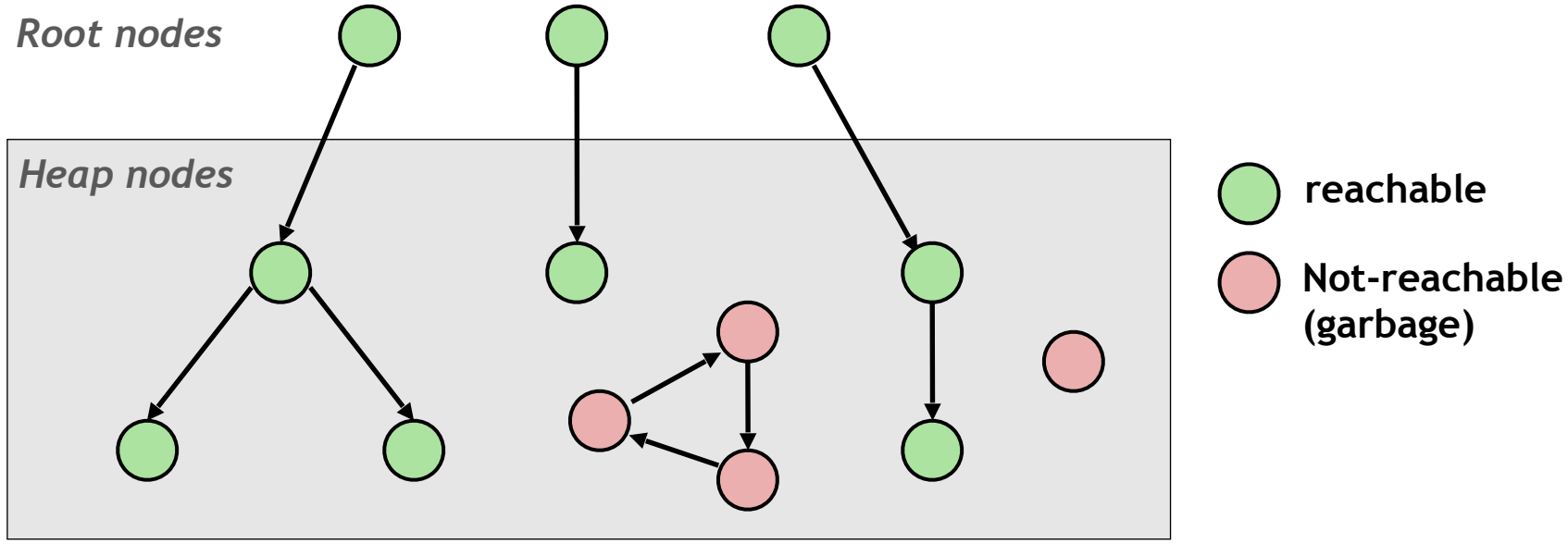
# Classical GC Algorithms



- **Reference counting (Collins, 1960)** (not discussed)
- **Mark-and-sweep collection (McCarthy, 1960)**
- **Copying collection (Minsky, 1963)**
- **Generational Collectors (Lieberman and Hewitt, 1983)**

# 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)

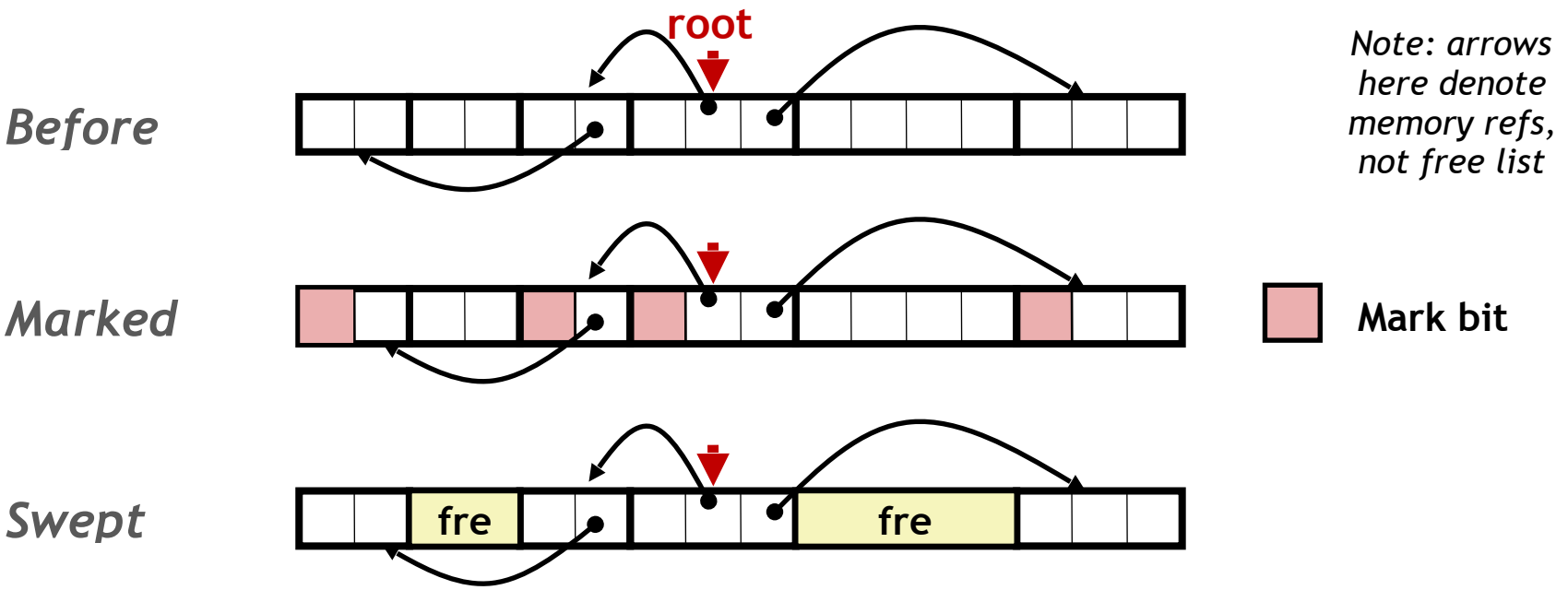



**A node (block) is *reachable* if there is a path from any root to that node.**

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

# Mark and Sweep Collecting

- **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



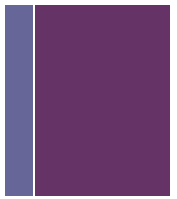

- Can be build on top of malloc/free (with a caveat!)

# “Conservative” Collection in C

- **Mark phase determines if a word is reachable by checking if it points to an allocated block of memory, however….**
  - C keeps no metadata about a word, its just bits!
  - C pointers can point to the *middle of a block*

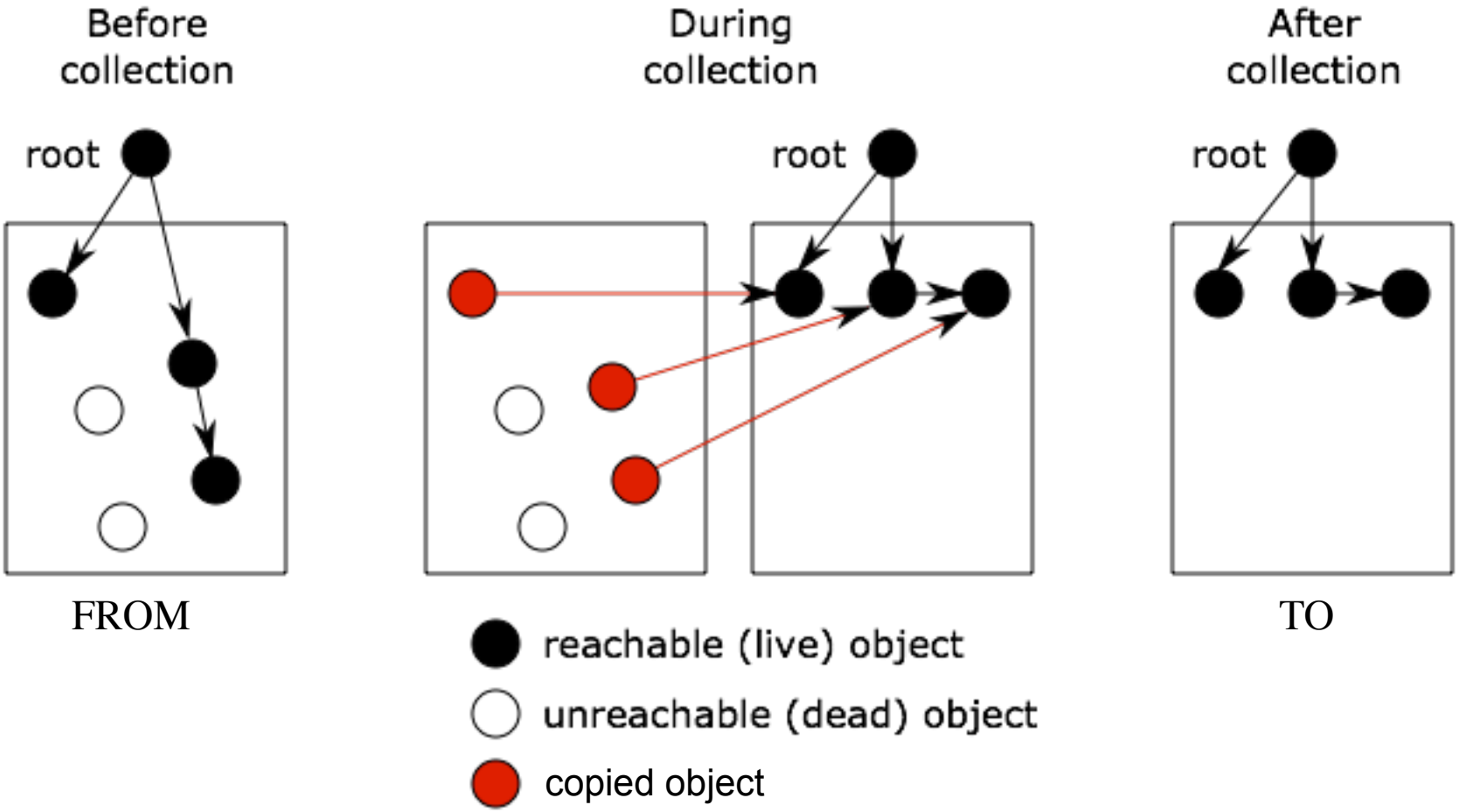

- **So how do we do determine reachability of such pointers?**
  - Additional tree data structure of allocated blocks introduced
  - Tree represents start and end addresses of all blocks
  - Collector a*ssumes all words are pointers*, searches tree for block whose address range contains that value *interpreted as a pointer*
  - Could yield false positives! Hence “conservative”.

# True Collection in Java: Copying

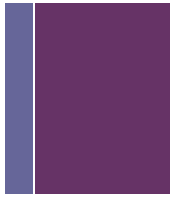


- **Java uses a technique different traditional 'Mark & Sweep' called 'Copying'**
- **The heap is split into two parts: FROM space and TO space**
- **Objects are allocated in the FROM space**
  - When FROM is full, collection begins by traversing from roots
- **During traversal, each reachable object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

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



# True Collection in Java: Generational

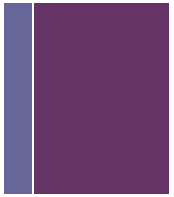


- **A variant of 'Copying'**
- ***Infant mortality* or the *generational hypothesis* is the observation that, in most cases, young objects are much more likely to die than old objects. Why?**
  - Objects that live for a long time tend to make up core program data structures and will probably live until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

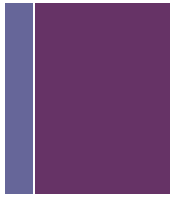
- **Generational GCs exploit this 'generational hypothesis'**
- **Instead of just two heaps (FROM and TO), we have several signifying 'generations' of objects.**
  - Younger generations collected more frequently than older generations (because younger generations will have more garbage to collect)
  - When a generation is traversed, live objects are copied to the next-older generation
  - When a generation fills up, it is garbage collected.
  - "Eden" space reserved for things that will (probably) never be 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 -- implicit 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*



- **Early garbage collectors indeed had poor garbage collection performance.**
  - A lot has happened in the last couple decades. The introduction of generational collectors and other improvements 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.