Lecture 16: Dynamic Memory (cont'd)

CS 105 Spring 2026

Review: DM Allocation Goals

- Provide memory (in heap) to a running program (allocate)
- Recycle memory when done (free)

- High **throughput:** number of requests completed per time unit
  - Make allocator efficient
  - Example: if your allocator processes 5,000 `malloc` calls and 5,000 `free` calls in 10 seconds then throughput is 1,000 operations/second

- High **memory utilization:** fraction of heap memory allocated
  - Minimize fragmentation

Review: Challenges

- Goal: maximize throughput and peak memory utilization

- Implementation Challenges:
  - How do we know how much memory to free given just a pointer?
  - How do we keep track of the free blocks?
  - How do we pick a block to use for allocation?
  - What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
  - How do we reinsert a freed block?

Review: Summary of Key Allocator Policies

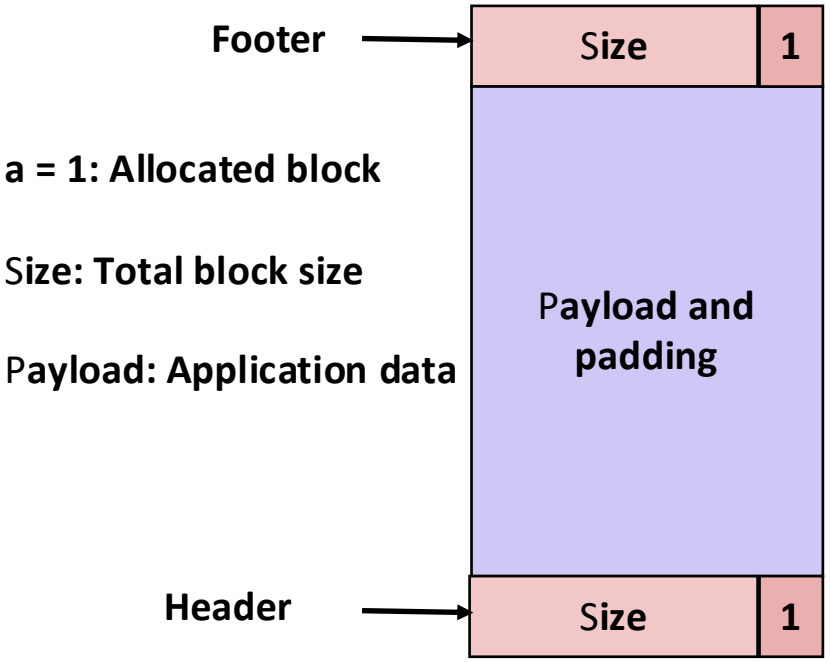
- Storage policy:
  - what data structure will you use to keep track of the free blocks?
- Placement policy:
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
  - segregated free lists approximate a best fit placement policy without having to search entire 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 threshold

Memory-Related Perils and Pitfalls

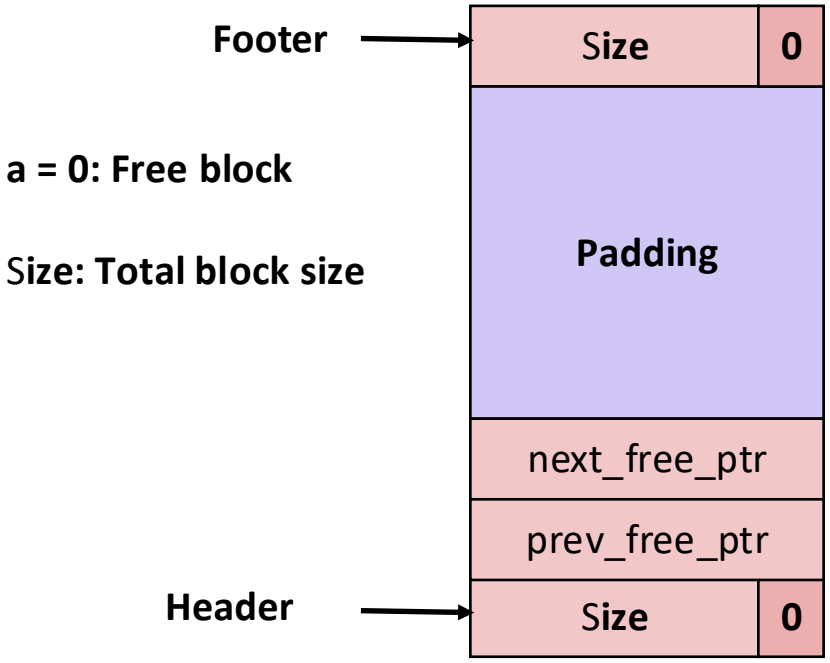
- Dereferencing bad pointers (Correctness)
- Reading uninitialized memory (Correctness)
- Overwriting memory (Security)
- Overreading memory (Security)
- Referencing freed blocks (Security)
- Freeing blocks multiple times (Security)
- Failing to free blocks (Performance)

Review: Block Format

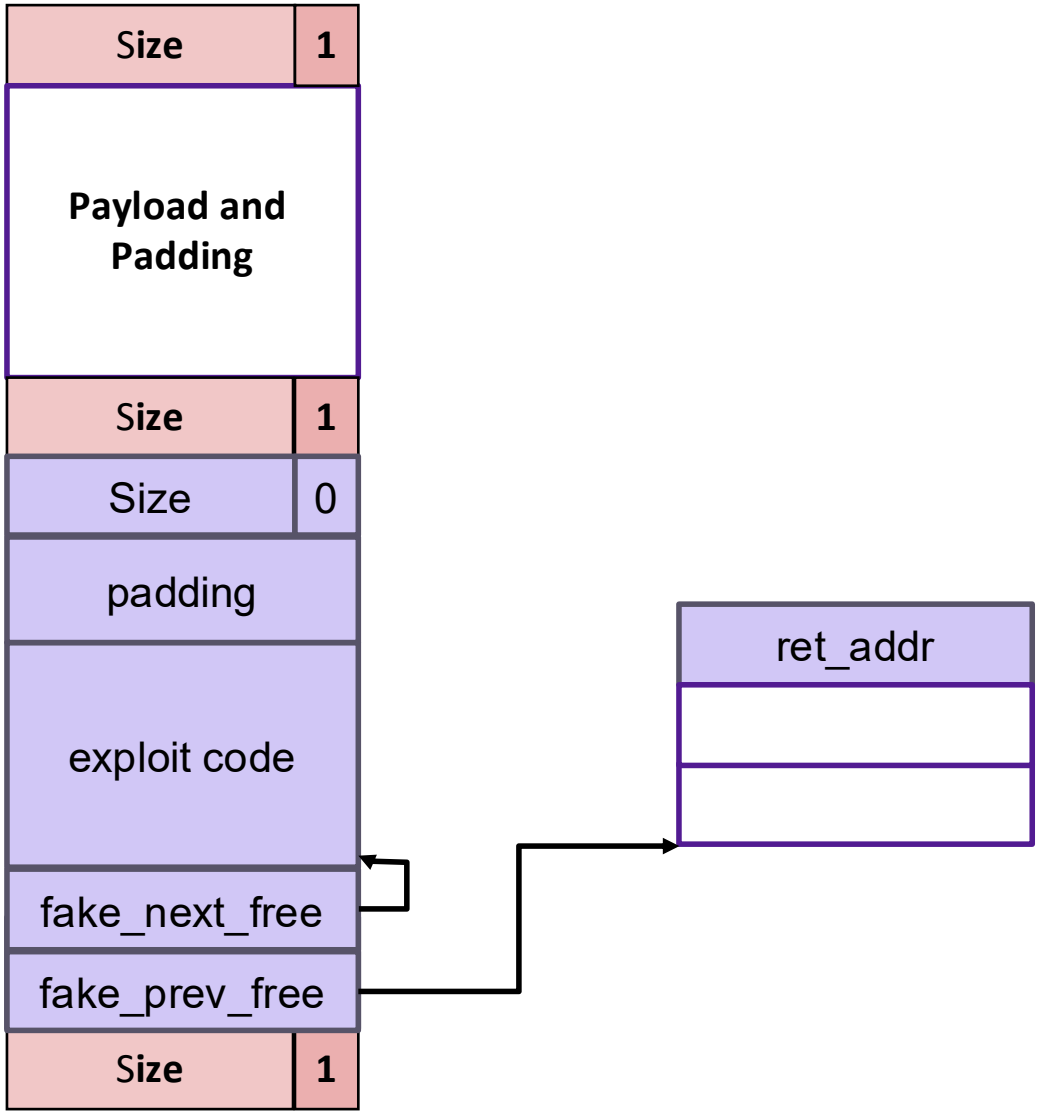
Allocated Blocks



Free Blocks

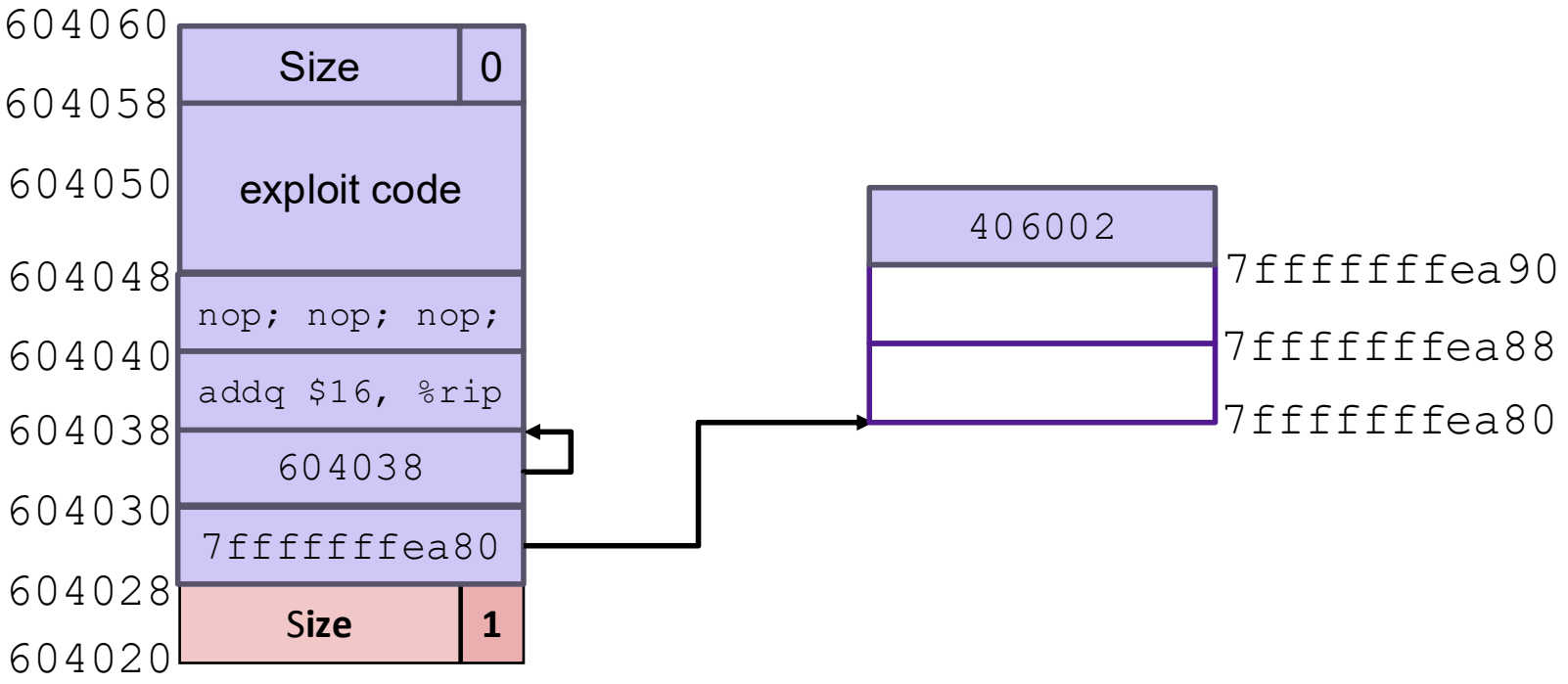


Example: Heap Smashing



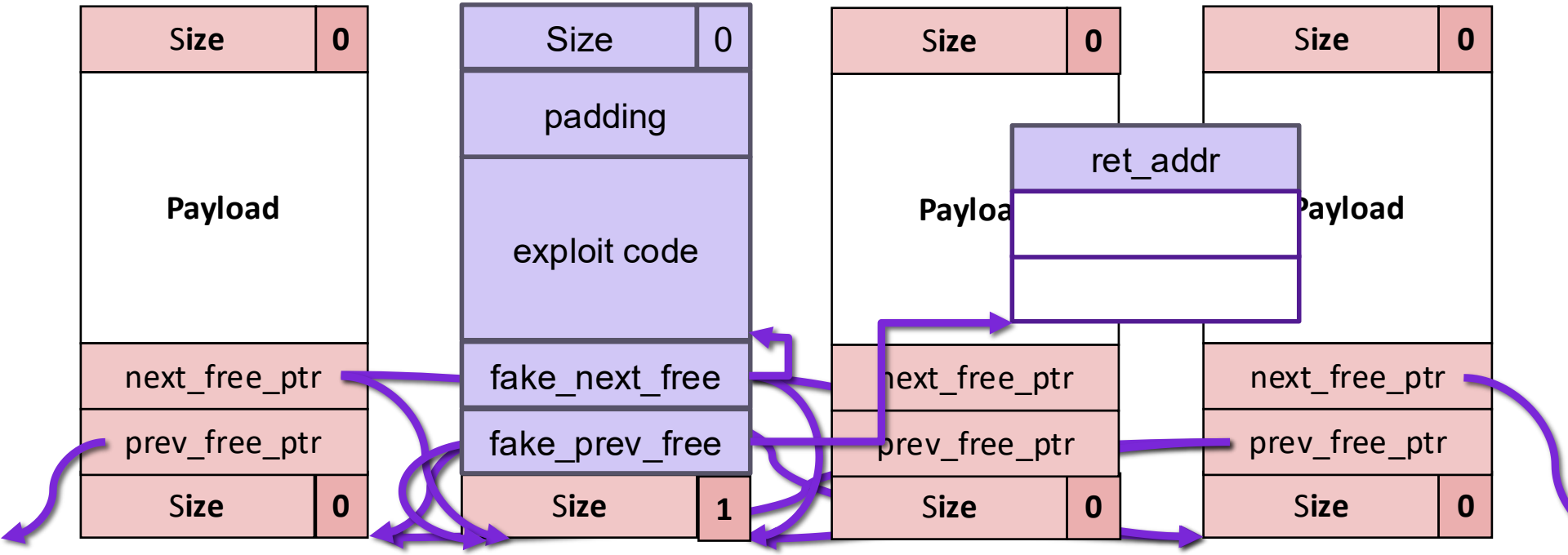
Exercise: Heap Smashing

- What would happen when the block after this one gets freed? (Assume that the coalesced block needs to be moved into a new linked list.)



Example: Double Free Vulnerability

```
free(ptr)
free(ptr)
p = malloc(n)
// write to block
// free block after target
```



Tools for Dealing With Memory Bugs

- Debugger: **`gdb`**
  - Good for finding bad pointer dereferences
  - Hard to detect the other memory bugs
- Heap consistency checker (e.g., **`mcheck`**)
  - Usually run silently, printing message only on error
  - Can be used to detect overreads, double-free
  - glibc malloc contains checking code
    - **`setenv MALLOC_CHECK_ 3`**
- Binary translator: **`valgrind`**
  - Powerful debugging and analysis technique
  - Rewrites text section of executable object file
  - Checks each individual reference at runtime
    - Bad pointers, overwrites, refs outside of allocated block

But Memory Bugs Persist…

WhatsApp

Memory-Related Perils and Pitfalls

- Dereferencing bad pointers (Correctness)
- Reading uninitialized memory (Correctness)
- Overwriting memory (Security)
- Overreading memory (Security)
- Referencing freed blocks (Security)
- Freeing blocks multiple times (Security)
- Failing to free blocks (Performance)

Implicit Allocators: 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 many dynamic languages:
  - Python, Java, Ruby, Perl, ML, Lisp, Mathematica

- Variants ("conservative" garbage collectors) exist for C and C++
  - However, cannot necessarily collect all garbage

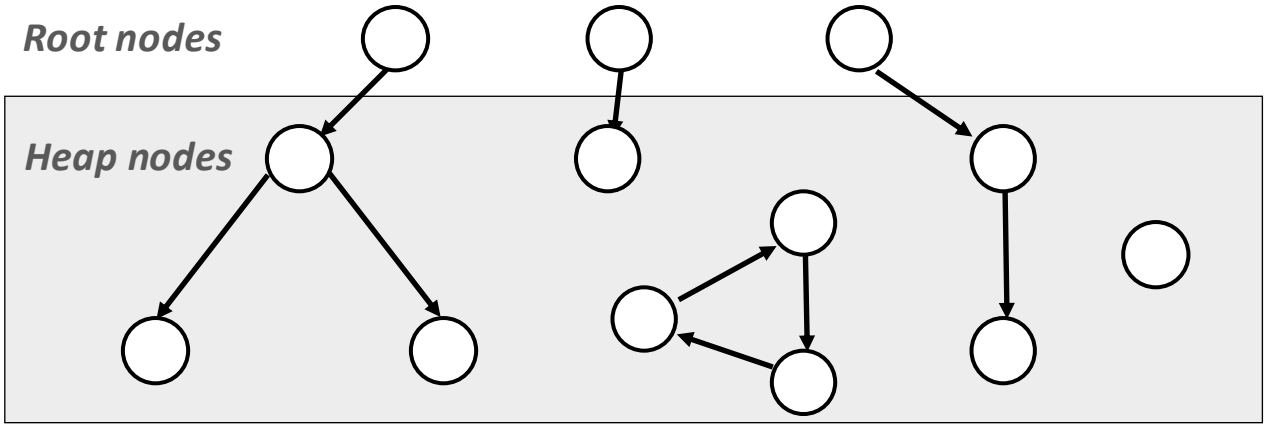
Garbage Collection

- How does the memory manager know when memory 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

- 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 `long`, and then back again)

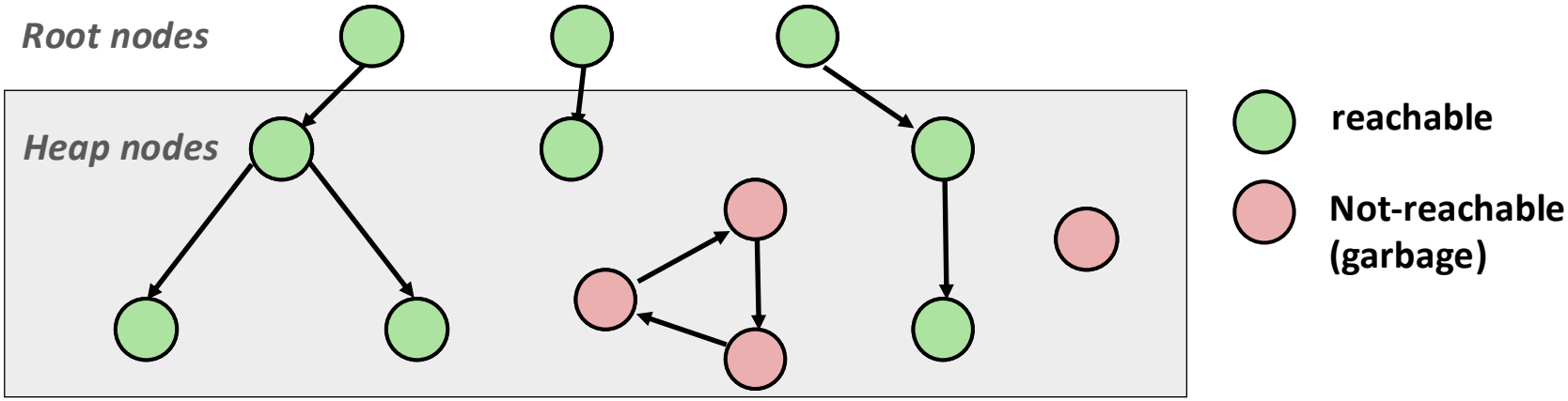
Memory as a Graph

- We view memory as a directed graph
  - Each block is a node in the graph (called a **heap node**)
  - Extra **root nodes** correspond to locations not in the heap that contain pointers into the heap
    - registers, local stack variables, or global variables
  - Each pointer is an edge in the graph



Memory as a Graph

- A node n is reachable if there exists a directed path from some root node to n
- Heap nodes that are not reachable are garbage
  - they can never again be used by the application
  - they should be freed ("garbage collected")



Garbage Collection

- The role of a garbage collector is

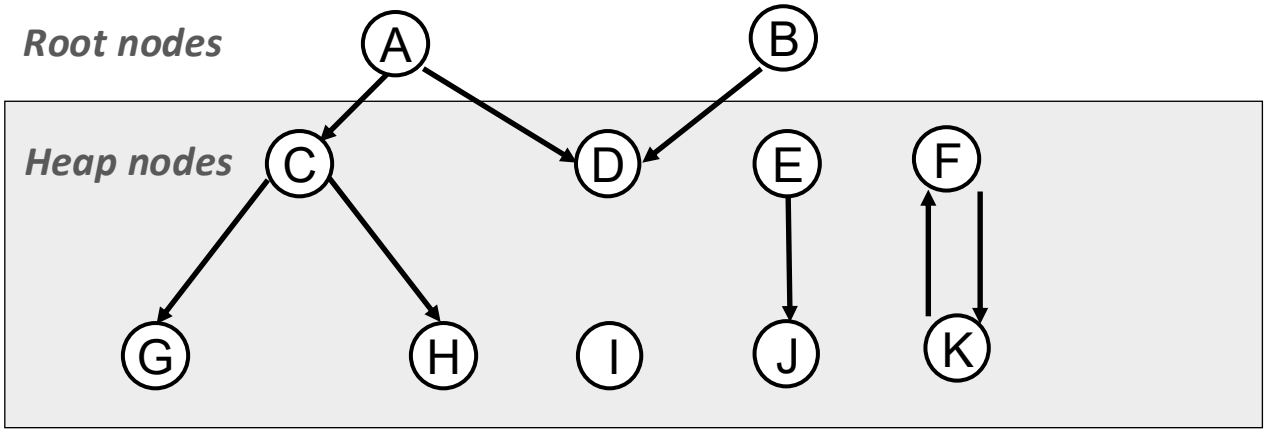
1. to maintain some representation of the reachability graph
2. to reclaim the unreachable nodes by freeing them
   - this can happen periodically or collector can run in parallel with application)

Languages that maintain tight control over how applications create and use pointers (e.g., Java, Python, OCaml) can maintain an exact representation of the graph

Garbage collectors for languages like C/C++ will be conservative

Exercise: Garbage Collection

- Consider the following graph representation of memory. Which nodes correspond to blocks that should be freed by the garbage collector?



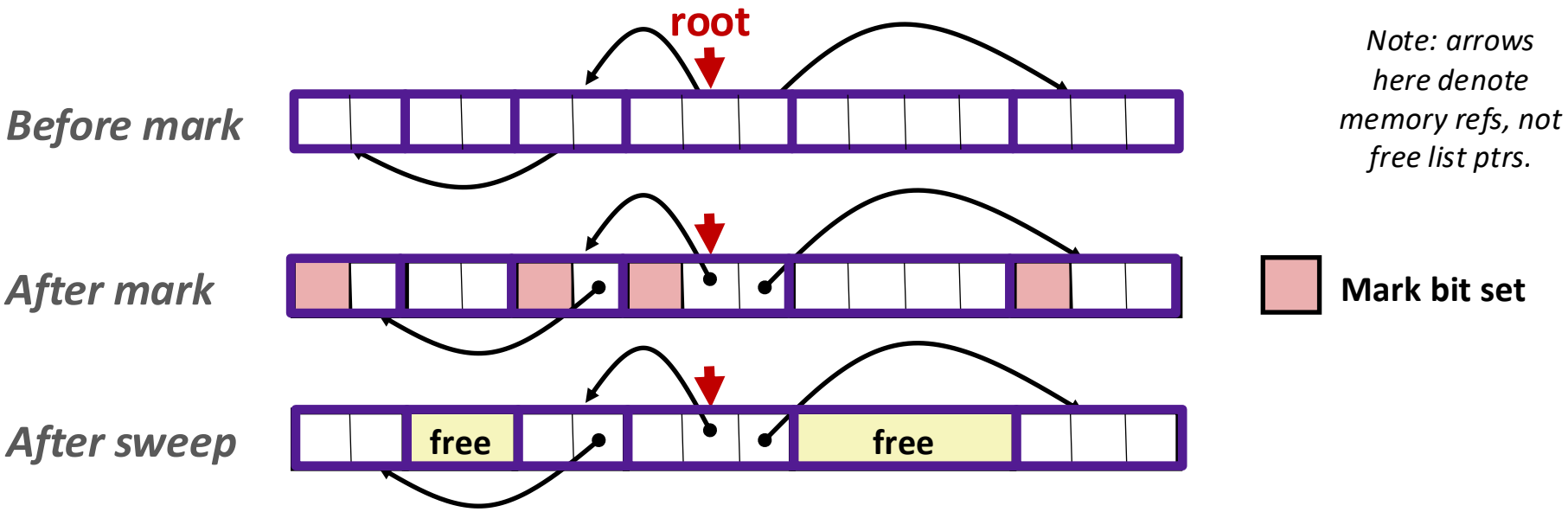
Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
  - Does not move blocks (unless you also “compact”)
- Reference counting (Collins, 1960)
  - Does not move blocks
- Copying collection (Minsky, 1963)
  - Moves blocks
- 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

Mark and Sweep Collector

- Each block header has an extra **mark bit**
  - can use one of the spare low-order bits

- Two phase protocol
  - **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 Collector

Mark using depth-first traversal of the memory graph

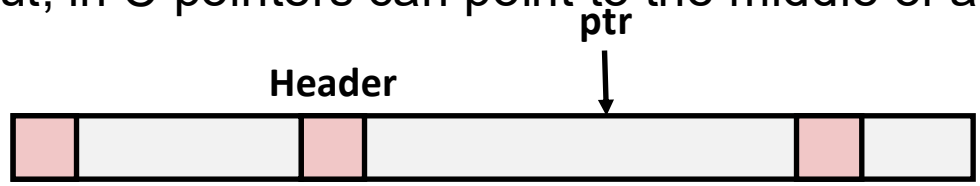
```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // do nothing if not pointer
   if (markBitSet(p)) return;       // check if already marked
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // call mark on all words
     mark(p[i]);                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

Sweep using lengths to find next block

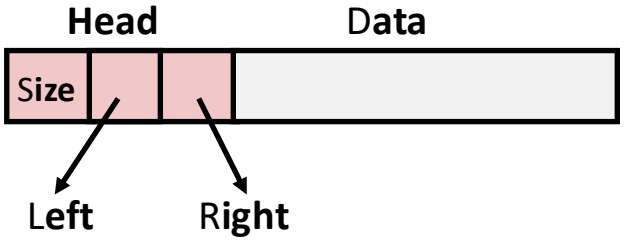
```
ptr sweep(ptr p, ptr end) {
   while (p < end) {
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p);
}
```

Conservative Mark & Sweep in C

- A “conservative garbage collector” for C programs
  - build on top of malloc/free package
  - allocate using `malloc` until you “run out of space”
  - **`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