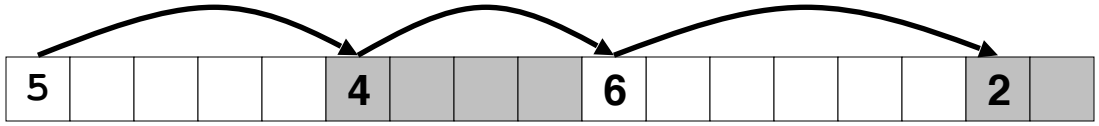
CSC 252: Computer Organization
Spring 2025: Lecture 24

Instructor: Yuhao Zhu

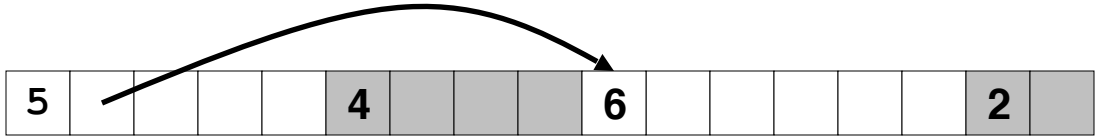
Department of Computer Science
University of Rochester

Keeping Track of Free Blocks

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

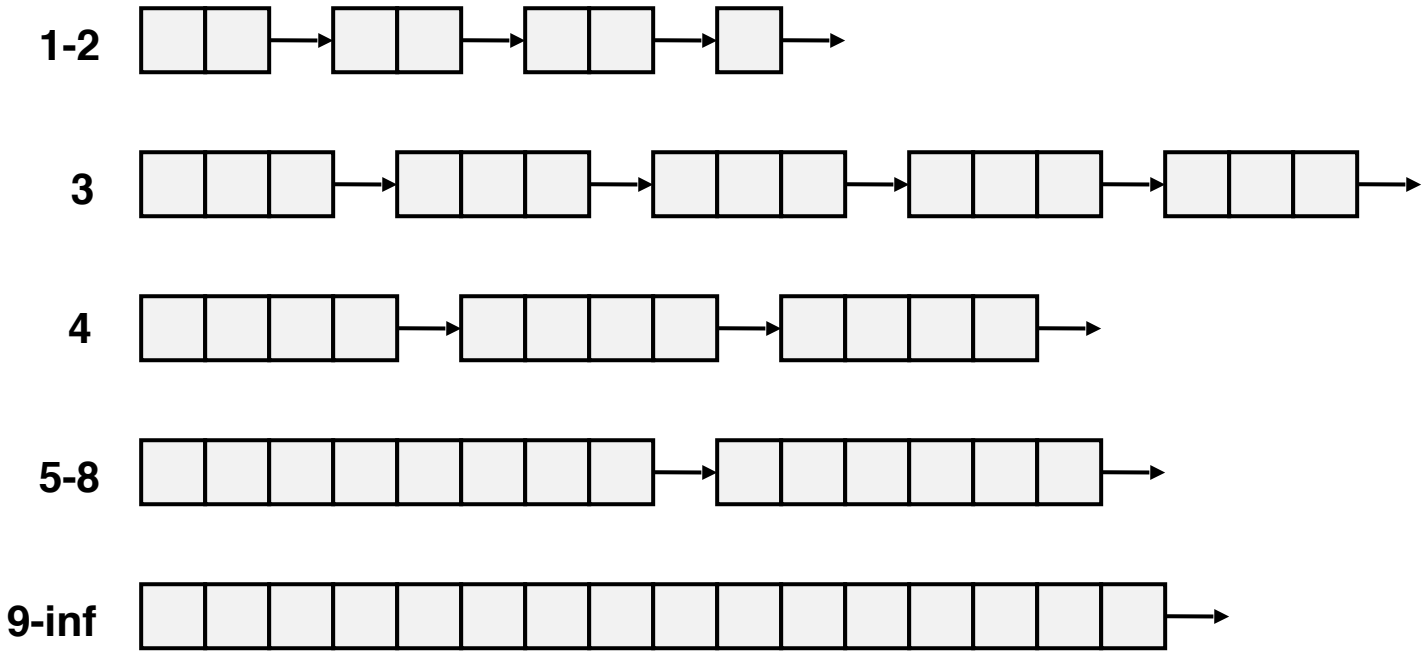


- Method 2: *Explicit list* among the free blocks using pointers



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

Segregated List (Seglist) Allocators



- Each *size class* of blocks has its own free list
- Organize the Seglist
  - Often have separate classes for each small size
  - For larger sizes: One class for each two-power size (why?)

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()`**)
  - Remember heap is in VM, so request heap memory in pages
  - Allocate block of *n* bytes from this new memory
  - Place remainder as a single free block in largest size class.
- To free a block:
  - Coalesce and place on appropriate list

Advantages of Seglist allocators

- Higher throughput
  - Constant time allocation and free for requests that have a dedicated free list (most of the cases)
  - log time for power-of-two size classes (searching the lists)
- 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.

Explicit/Implicit Memory Management

- So far we have been talking about explicitly memory management: programmers explicitly calling malloc/free (C/C++)
- Downside: potential memory leaks

```
void foo() {
   int *p = malloc(128);
   p = malloc(32);
   return; /* both blocks are now garbage */
}
```

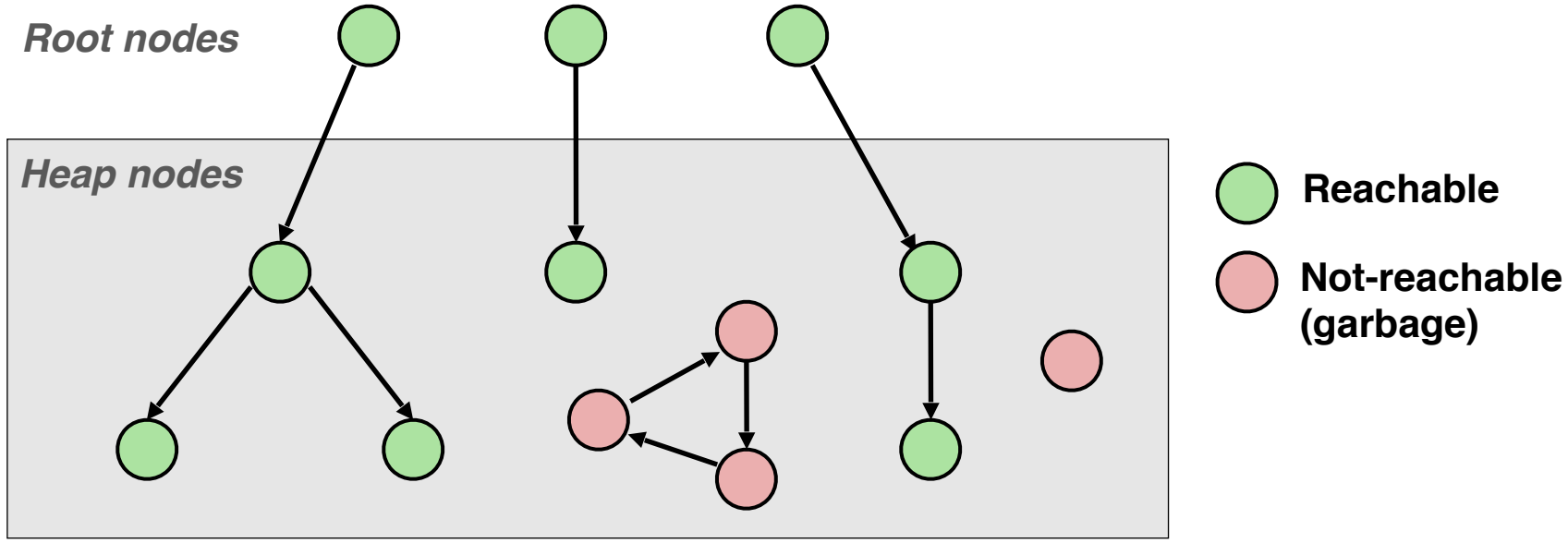
- Alternative: implicit memory management; the programmers never explicitly request/free memory
- Common in many dynamic languages:
  - Python, Ruby, Java, JavaScript, Perl, ML, Lisp, Mathematica
- The key: Garbage collection
  - Automatic reclamation of heap-allocated storage—application never has to free

Garbage Collection

- How does the memory manager know when certain memory blocks can be freed?
  - If a block will never be used in the future. How do we know that?
  - In general we cannot know what is going to be used in the future since it depends on program's future behaviors
  - But we can tell that certain blocks cannot possibly be used ***if there are no pointers to them***
  - Garbage collection is essentially to obtain all **reachable** blocks and discard unreachable blocks.

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)

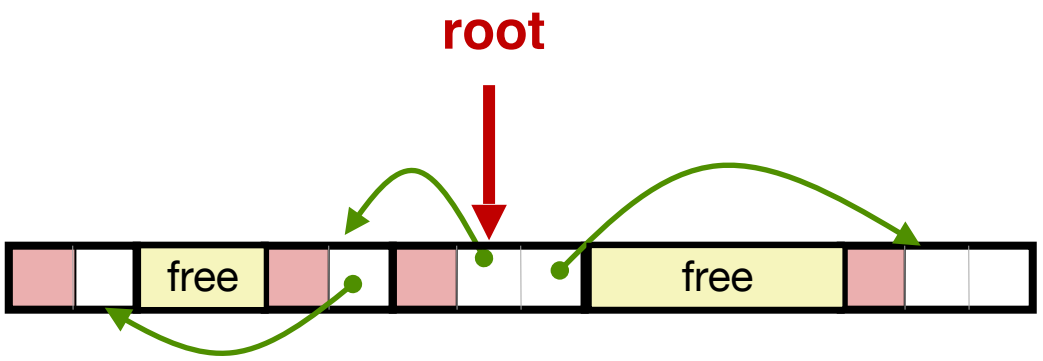


A node (block) is *reachable* if there is a path from any root to that node.

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

Mark and Sweep Collecting

- Idea:
  - Use extra ***mark bit*** in the header to indicate if a block is reachable
  - *Mark:* Start at roots and set mark bit on each reachable block
  - *Sweep:* Scan all blocks and free blocks that are not marked



Note: arrows here denote memory refs, not free list ptrs.

Mark bit set

Mark and Sweep (cont.)

Mark using depth-first traversal of the memory graph

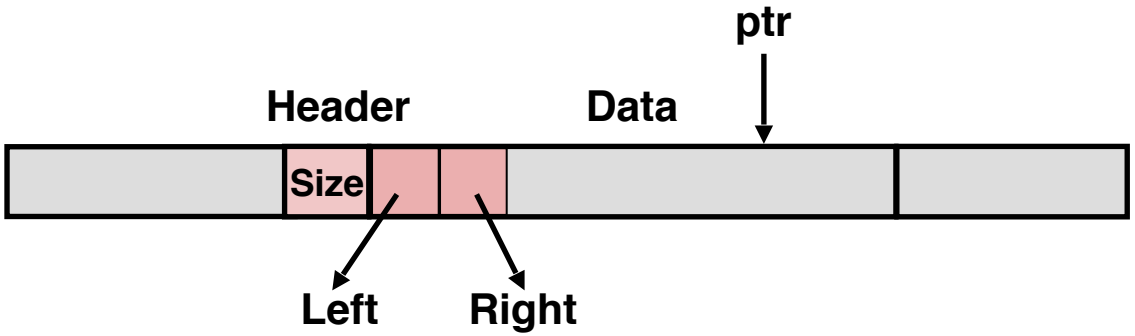
```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // do nothing if not pointer
   if (markBitSet(p)) return;       // check if already marked
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // call mark on all words
     mark(p[i]);                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

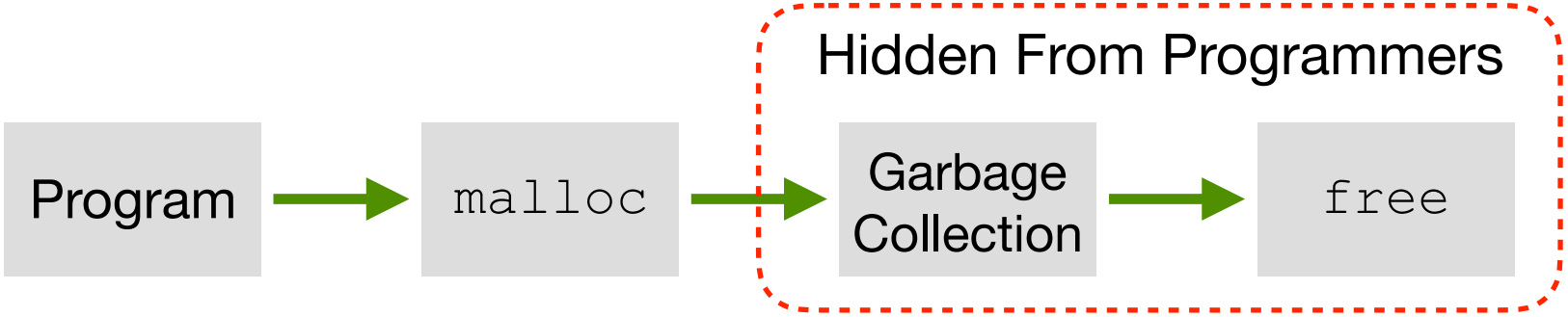
- Garbage Collection in C is tricky.
- How do you know a pointer is a pointer? After all, a pointer is just a 8-byte value. Any consecutive 8 bytes could be disguised as a pointer.
  - Must be conservative. Any 8 bytes whose values fall within the range of the heap must be treated as a pointer.
- C pointers can point to the middle of a block. How do you find the header of a block?
  - Can use a balanced binary tree to keep track of all allocated blocks (key is start-of-block)



Left: smaller addresses
Right: larger addresses

Potential GC Implementations (in C)

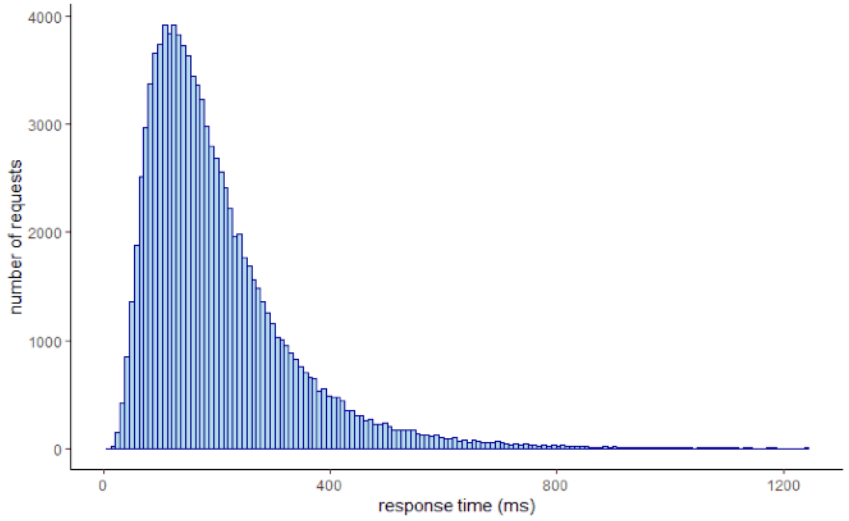
- Can build on top of `malloc`/`free` function
  - Call `malloc` until you run out of space. Then `malloc` will call GC.
  - Stop-the-world GC. When performing GC, the entire program stops. Some calls to `malloc` will take considerably longer than others.



- To minimize main application (called mutator) pause time:
  - Incremental GC: Examine a small portion of heap every GC run
  - Concurrent GC: Run GC service in a separate process/thread

Garbage Collection Implications

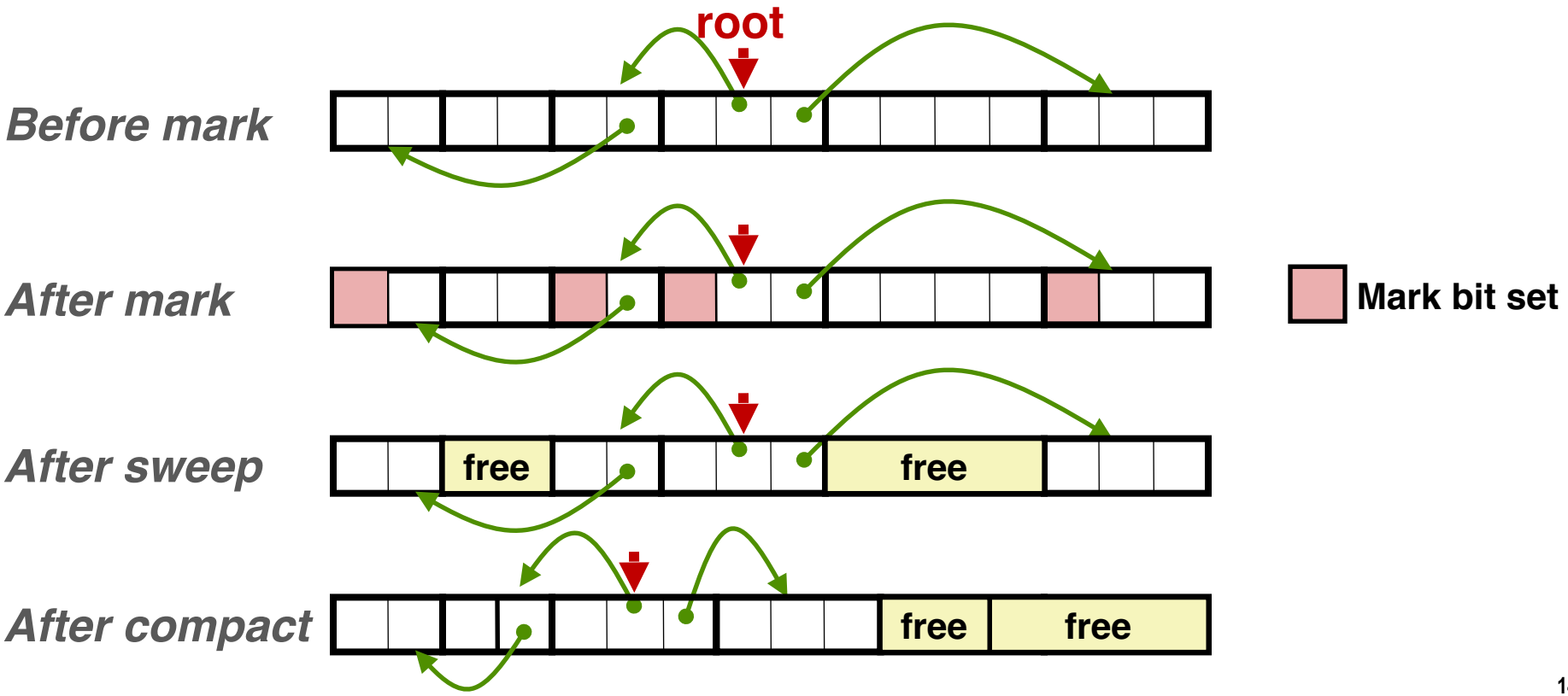
- GC is a great source of performance non-determinisms
  - Generally can't predict when GC will happen
  - Stop-the-world GC makes program periodically unresponsive
  - Concurrent/Incremental GC helps, but still has performance impacts
  - Bad for real-time systems: think of a self-driving car that needs to decide whether to avoid a pedestrian but a GC kicks in…
  - Bad for server/cloud systems: GC is a great source of *tail latency*



https://robertovitillo.com/why-you-should-measure-tail-latencies/

Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
  - After M&S, compact allocated blocks to consecutive memory region.
  - Reduce external fragmentation. Allocation is also easier.



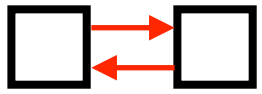
Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
- Mark-copy collection (Minsky, 1963)
  - After mark, copy reachable objects to another region of memory as they are being traversed. Can be done without auxiliary storage.
- Generational Collectors (Lieberman and Hewitt, 1983)
  - Observation: most allocations become garbage very soon ("infant mortality"); others will survive for a long time.
  - Wasteful to scan long-lived objects every collection time
  - Idea: divide heap into two generations, young and old. Allocate into young gen., and promote to old gen. if lived long enough. Collect young gen. more often than old gen.
- Question: Can any of these algorithms be used for GC in C?

Jones and Lin, "Garbage Collection: Algorithms for Automatic Dynamic Memory", 1996.

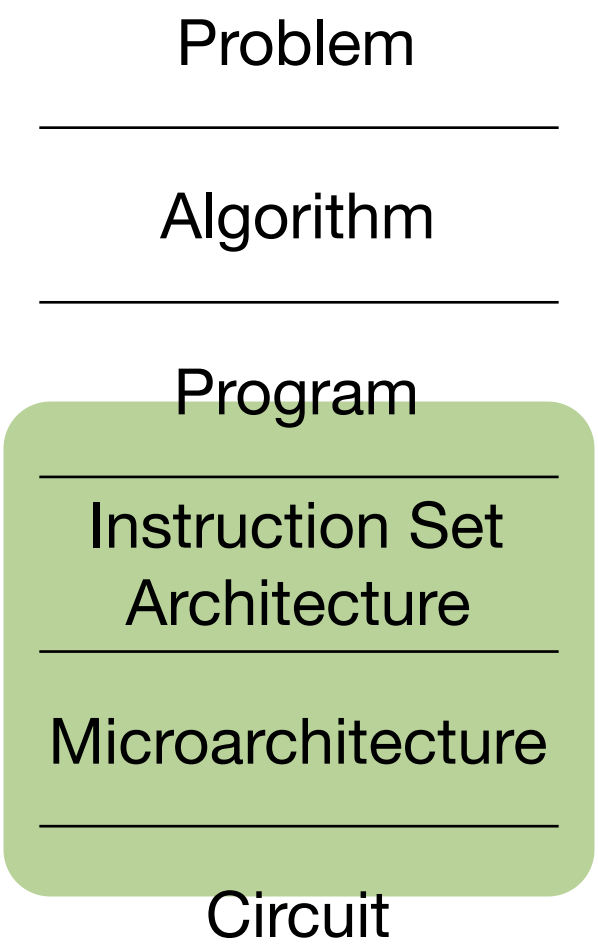
Classical GC Algorithms

- All the GC algorithms described so far are tracing-based
  - Start from the root pointers, trace all the reachable objects
  - Need graph traversal. Different to implement.
- Reference counting (Collins, 1960)
  - Keep a counter for each object
  - Increment the counter if there is a new pointer pointing to the object
  - Decrement the counter if a pointer is taken off the object
  - When the counter reaches zero, collect the object
- Advantages of Reference Counting
  - Simpler to implement
  - Collect garbage objects immediately; generally less long pauses
- Disadvantages of Reference Counting
  - A naive implementation can't deal with self-referencing



- A heterogeneous approach (RC + tracing) is often used

Scope of CSC 252



The Most Important Take Away of 252

- "*There is no magic.*"
- Computer systems are engineering work, not science, so:
  - We know exactly how things work because we build them (unlike natural sciences)
  - Every thing can be derived from first principles. Trust your logical reasoning.

The Second Most Important Take Away of 252

- "*Things don't have to be this way.*"
- As long as you don't violate physics, you can design a computer however you want.
- But every design decision you make usually involves certain trade-offs. Be clear what your design goal is.

The Third Most Important Take Away of 252

- Virtual all computer system design practices follow a small set of basic principles.
- It is these basic principles that are important, not the practices.