Section 10: Memory Allocation Topics

- ~~**Dynamic memory allocation**~~
  - ~~Size/number of data structures may only be known at run time~~
  - ~~Need to allocate space on the heap~~
  - ~~Need to de-allocate (free) unused memory so it can be re-allocated~~
- ~~**Implementation**~~
  - ~~Implicit free lists~~
  - ~~Explicit free lists – subject of next programming assignment~~
  - ~~Segregated free lists~~
- **Garbage collection**
- **Common memory-related bugs in C programs**

Implicit Memory Allocation: Garbage Collection

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

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

- **Common in functional languages, scripting languages, and modern object oriented languages:**
  - Lisp, ML, Java, Perl, Mathematica

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

Garbage Collection

- **How does the memory allocator 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 <u>no pointers to them</u>

- **So the memory allocator needs to know what is a pointer and what is not – how can it do this?**

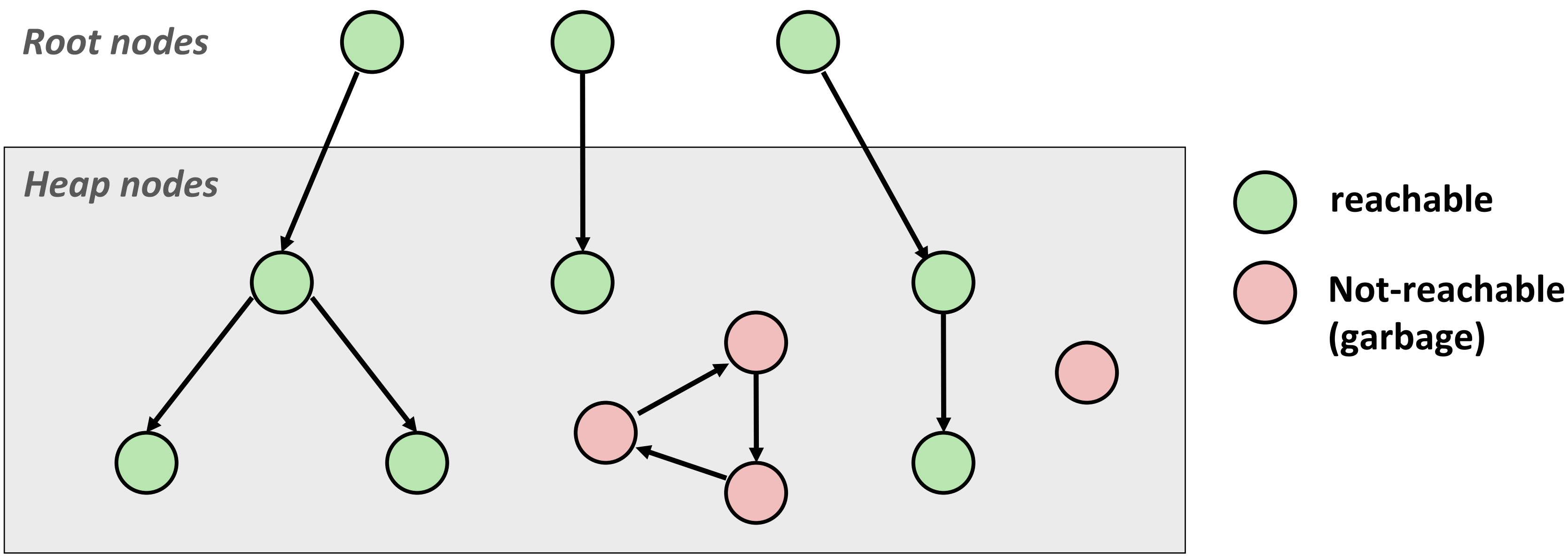
- **We'll make some assumptions about pointers:**
  - Memory allocator can distinguish pointers from non-pointers
  - All pointers point to the start of a block in the heap
  - Application cannot hide pointers
    (e.g., by coercing them to an `int`, and then back again)

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 allocated heap 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)

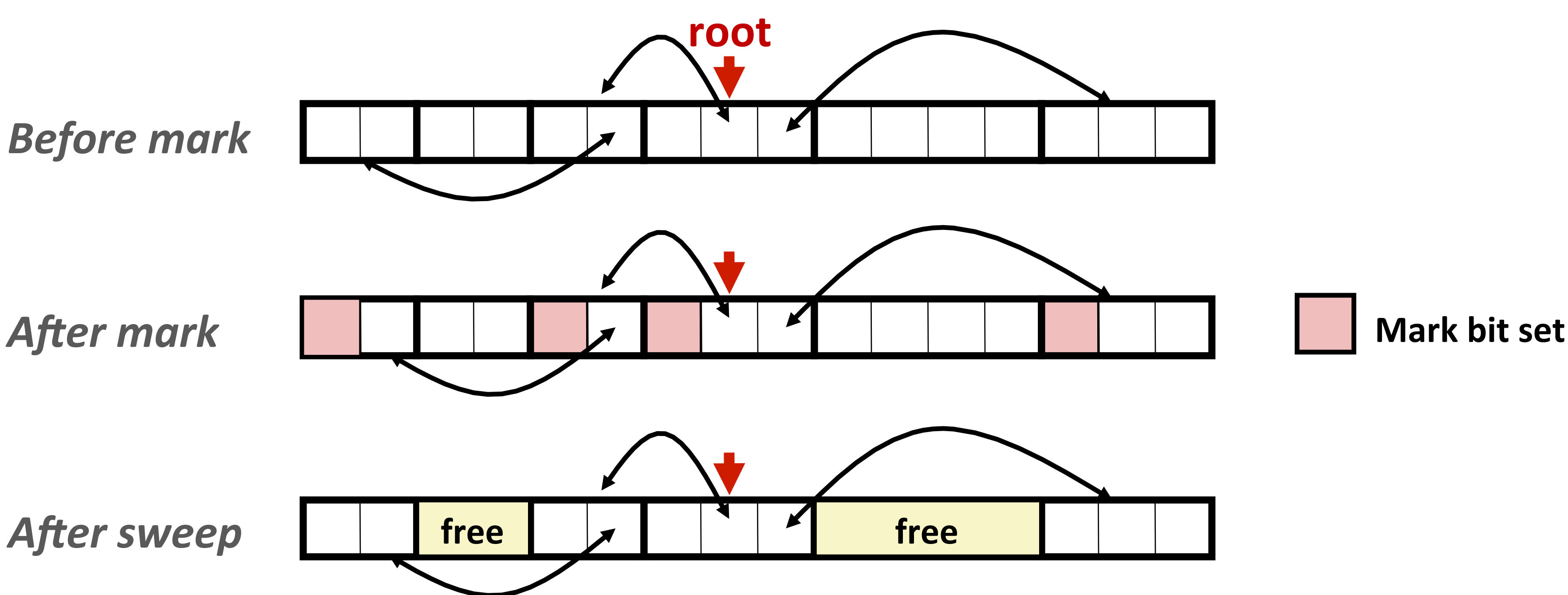


A node (block) is *reachable* if there is a path from any root to that node

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 can use functions such as:**
  - **`new(n)`** : returns pointer to new block with all locations cleared
  - **`read(b,i)`** : read location **`i`** of block **`b`** into register
    - **`b[i]`**
  - **`write(b,i,v)`** : write **`v`** into location **`i`** of block **`b`**
    - **`b[i] = v`**
- **Each block will have a header word**
    - **`b[-1]`**

- **Functions used by the garbage collector:**
  - **`is_ptr(p)`** : determines whether **`p`** is a pointer to a block
  - **`length(p)`** : returns length of block pointed to by **`p`**, not including header
  - **`get_roots()`** : returns all the roots

Mark and Sweep (cont.)

Mark using depth-first traversal of the memory graph

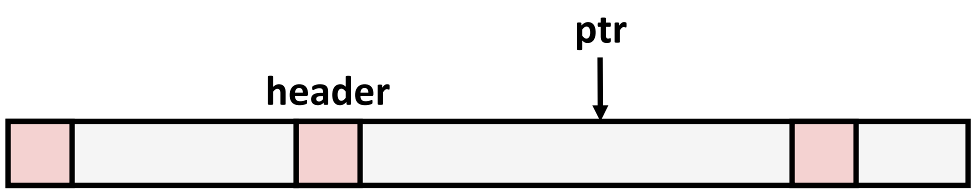
```
ptr mark(ptr p) {                     // p: some word in a heap block
   if (!is_ptr(p)) return;            // do nothing if not pointer
   if (markBitSet(p)) return;         // check if already marked
   setMarkBit(p);                     // set the mark bit
   for (i=0; i < length(p); i++)    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

Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {           // ptrs to start & end of heap
   while (p < end) {                  // while not at end of heap
      if markBitSet(p)                // check if block is marked
         clearMarkBit();              // if so, reset mark bit
      else if (allocateBitSet(p))     // if not marked, but allocated
         free(p);                     // free the block
      p += length(p);                 // adjust pointer to next block
}
```

Conservative Mark & Sweep in C

- **Would mark & sweep work in C?**
  - **`is_ptr()`** (previous slide) determines if a word is a pointer by checking if it points to an allocated block of memory
  - But in C, pointers can point into the *middle* of allocated blocks (not so in Java)
    - Makes it tricky to find all allocated blocks in mark phase



  - There are ways to solve/avoid this problem in C, but the resulting garbage collector is *conservative:*
    - Every reachable node correctly identified as reachable, but some unreachable nodes might be incorrectly marked as reachable