

# Memory Allocation III

CSE 351 Autumn 2020

## Instructor:

Justin Hsia

## Teaching Assistants:

Aman Mohammed

Ami Oka

Callum Walker

Cosmo Wang

Hang Do

Jim Limprasert

Joy Dang

Julia Wang

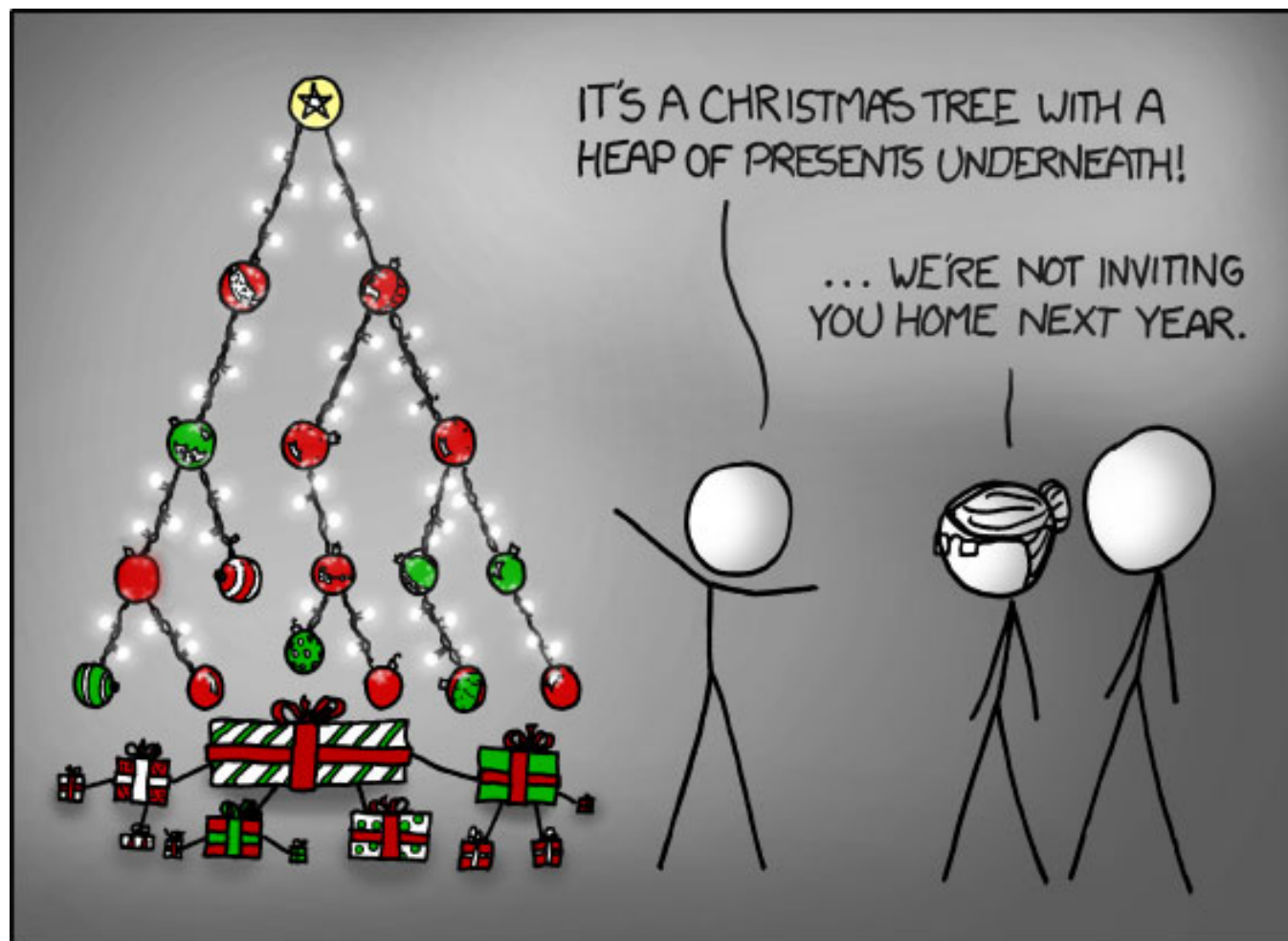
Kaelin Laundry

Kyrie Dowling

Mariam Mayanja

Shawn Stanley

Yan Zhe Ong



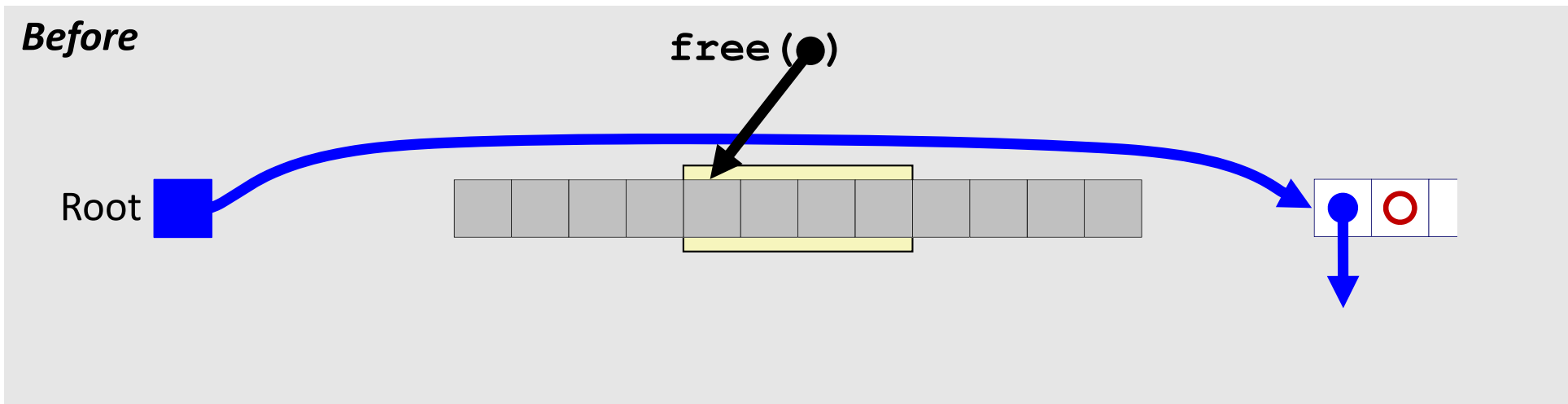
<https://xkcd.com/835/>

# Administrivia

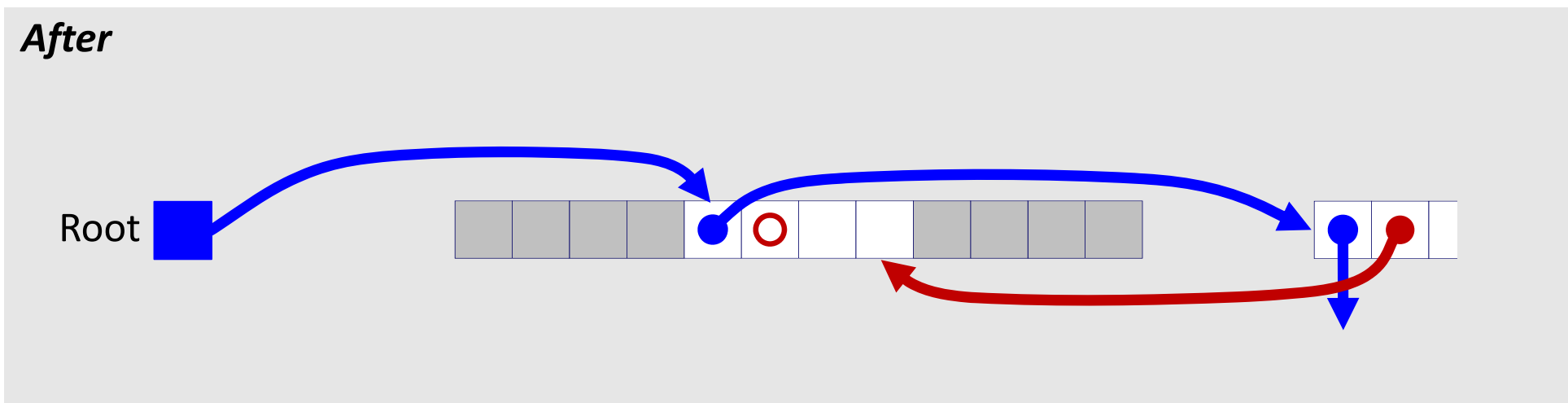
- ❖ hw25 due Wednesday (12/9)
- ❖ Lab 5 due next Friday (12/11)
  - Recommended that you watch the Lab 5 helper videos
- ❖ **Final Exam:** Group (12/11-13), Individual (12/16-17)
  - Sign your group up – matching will start late tonight
  - Final review section on 12/10, no lecture on 12/11
  - Review Session: Wed, 12/9, 5:30 – 7:30 pm on Zoom
    - More info to be released on Ed Discussion

# Freeing with LIFO Policy (Case 1)

Boundary tags not shown, but don't forget about them!

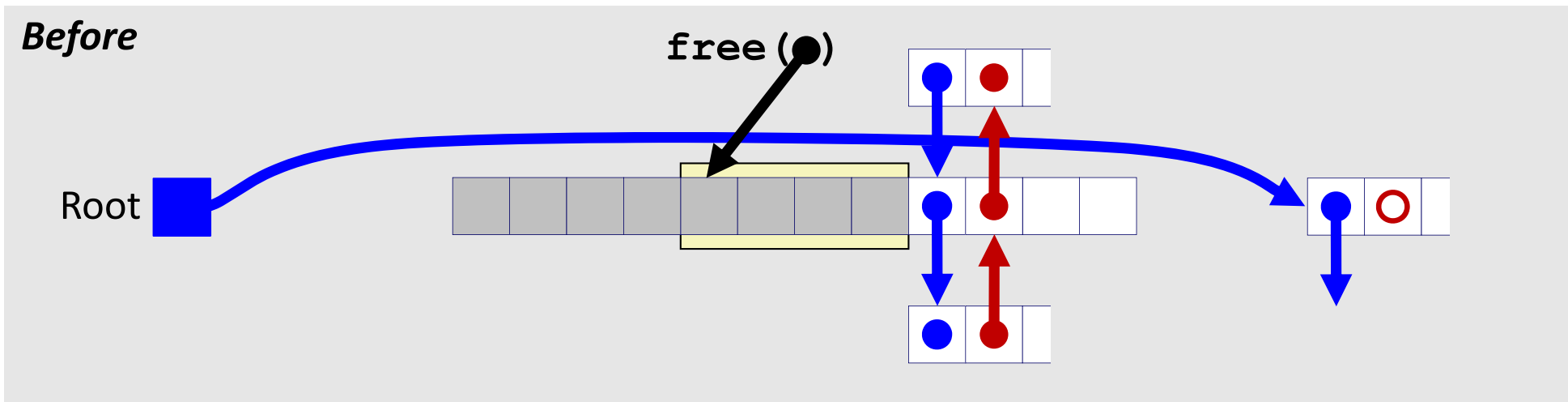


❖ Insert the freed block at the root of the list

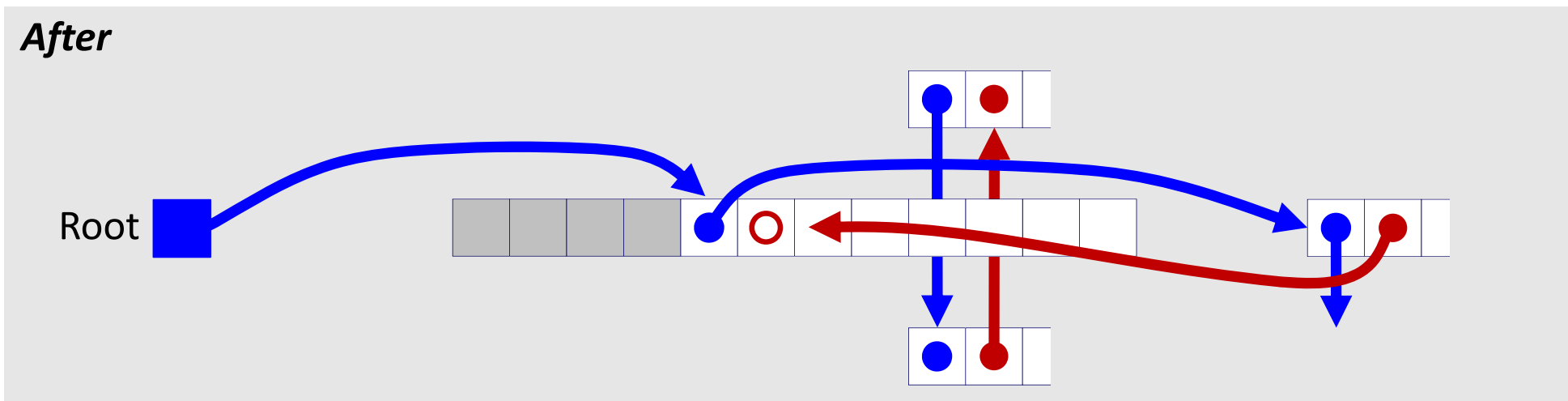


# Freeing with LIFO Policy (Case 2)

Boundary tags not shown, but don't forget about them!

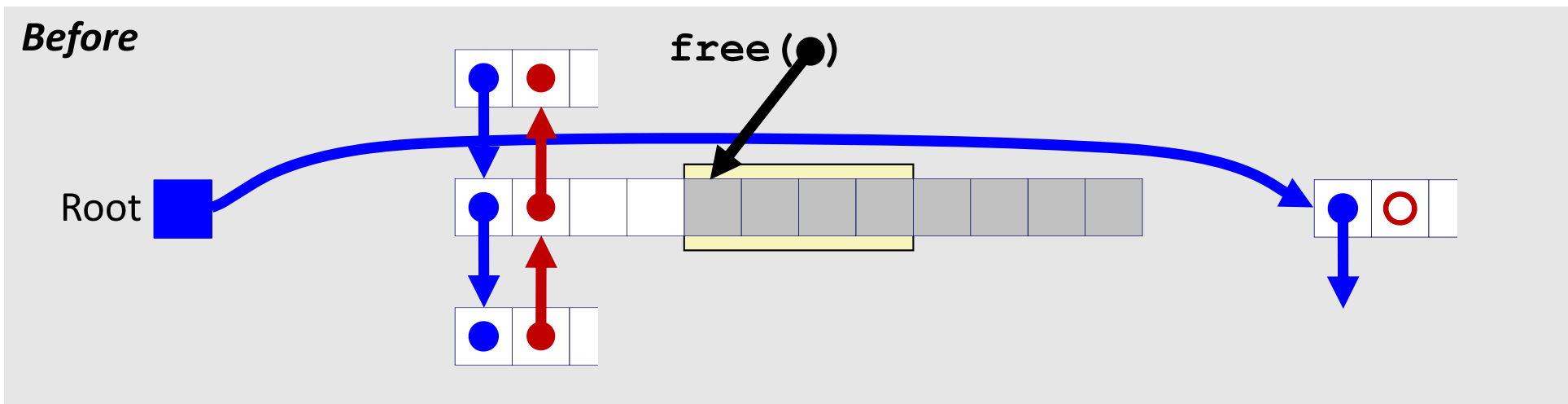


- ❖ Splice successor block out of list, coalesce both memory blocks, and insert the new block at the root of the list

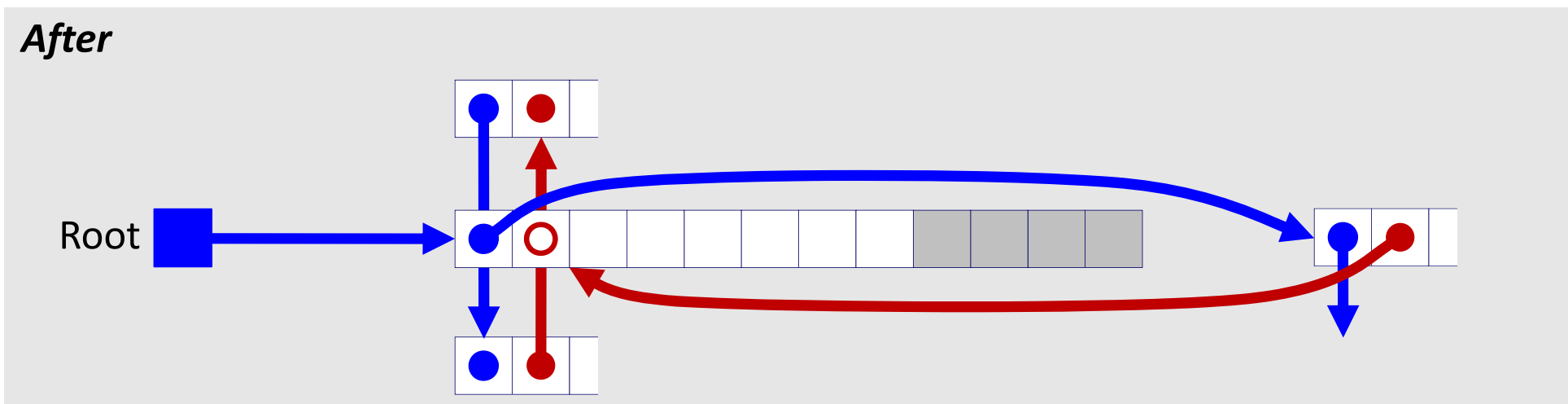


# Freeing with LIFO Policy (Case 3)

Boundary tags not shown, but don't forget about them!

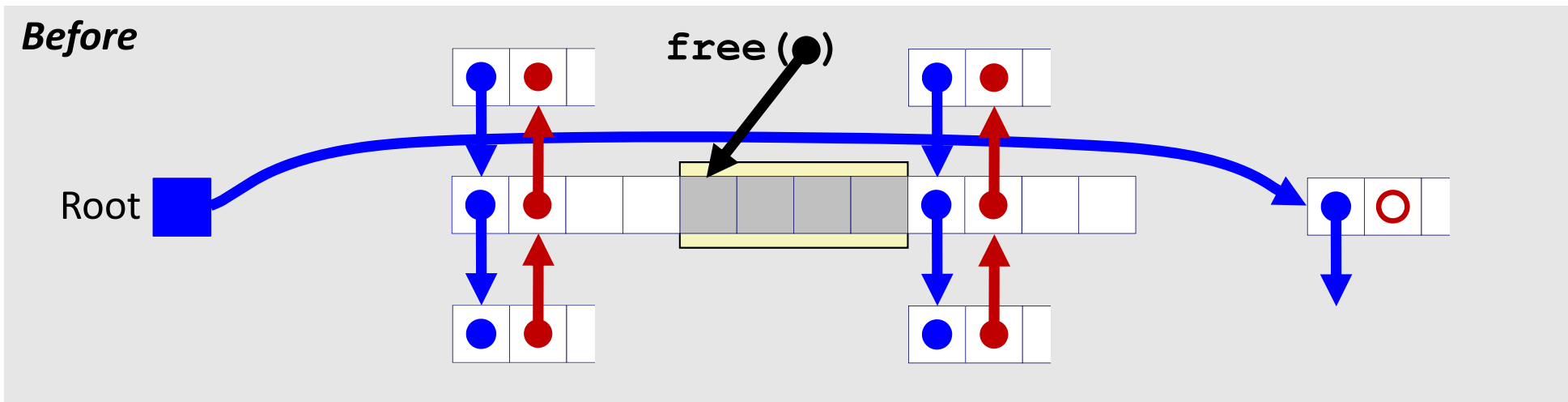


- ❖ Splice predecessor block out of list, coalesce both memory blocks, and insert the new block at the root of the list

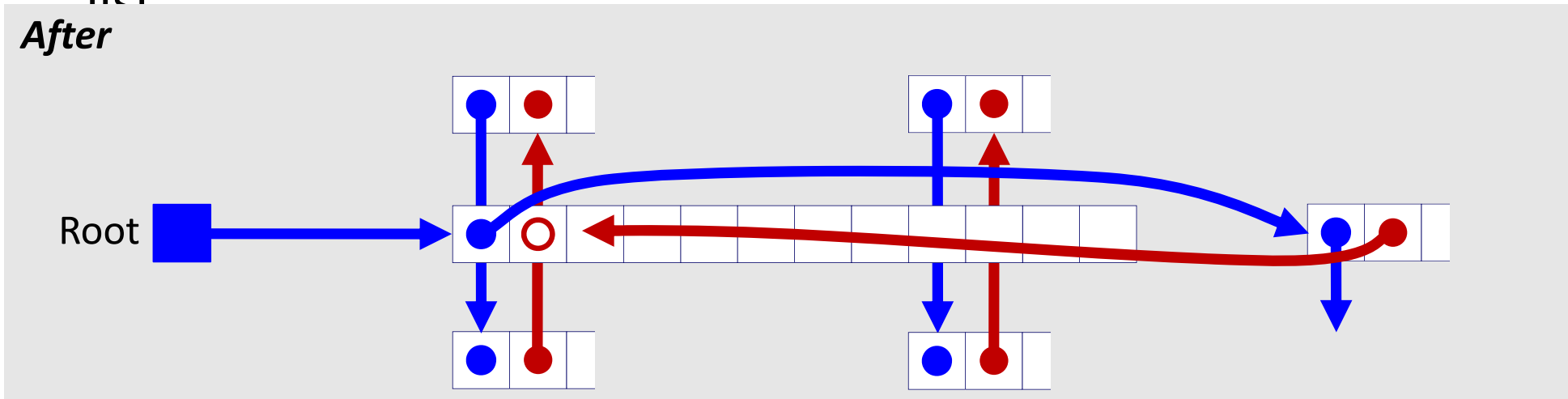


# Freeing with LIFO Policy (Case 4)

Boundary tags not shown, but don't forget about them!



- ❖ Splice predecessor and successor blocks out of list, coalesce all 3 memory blocks, and insert the new block at the root of the list



# Lab 5 Hints

- ❖ Struct pointers can be used to access field values, even if no struct instances have been created – just reinterpreting the data in memory
- ❖ **Pay attention to is boundary tag data**
  - Size value + 2 tag bits – when do these need to be updated and do they have the correct values?
  - The `examine_heap` function follows the implicit free list searching algorithm – don't take its output as “truth”
- ❖ Learn to use and interpret the trace files for testing!!!
- ❖ A special heap block marks the end of the heap

# Explicit List Summary

- ❖ Comparison with implicit list:
  - Block allocation is linear time in number of *free* blocks instead of *all* blocks
    - *Much faster* when most of the memory is full
  - Slightly more complicated allocate and free since we need to splice blocks in and out of the list
  - Some extra space for the links (2 extra pointers needed for each free block)
    - Increases minimum block size, leading to more internal fragmentation
  
- ❖ Most common use of explicit lists is in conjunction with *segregated free lists*
  - Keep multiple linked lists of different size classes, or possibly for different types of objects



# Allocation Policy Tradeoffs

- ❖ Data structure of blocks on lists
  - Implicit (free/allocated), explicit (free), segregated (many free lists) – others possible!
- ❖ Placement policy: first-fit, next-fit, best-fit
  - Throughput vs. amount of fragmentation
- ❖ When do we split free blocks?
  - How much internal fragmentation are we willing to tolerate?
- ❖ When do we coalesce free blocks?
  - **Immediate coalescing:** Every time `free` is called
  - **Deferred coalescing:** Defer coalescing until needed
    - *e.g.*, when scanning free list for `malloc` or when external fragmentation reaches some threshold

# More Info on Allocators

- ❖ D. Knuth, “*The Art of Computer Programming*”, 2<sup>nd</sup> edition, Addison Wesley, 1973
  - The classic reference on dynamic storage allocation
- ❖ Wilson et al, “*Dynamic Storage Allocation: A Survey and Critical Review*”, Proc. 1995 Int’l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.
  - Comprehensive survey
  - Available from CS:APP student site ([csapp.cs.cmu.edu](http://csapp.cs.cmu.edu))

# Memory Allocation

- ❖ Dynamic memory allocation
  - Introduction and goals
  - Allocation and deallocation (free)
  - Fragmentation
- ❖ Explicit allocation implementation
  - Implicit free lists
  - Explicit free lists (Lab 5)
  - Segregated free lists
- ❖ **Implicit deallocation: garbage collection**
- ❖ **Common memory-related bugs in C**

# Reading Review

- ❖ Terminology:
  - Garbage collection: mark-and-sweep
  - Memory-related issues in C
- ❖ Questions from the Reading?

# Wouldn't it be nice...

- ❖ If we never had to free memory?
- ❖ Do you free objects in Java?
  - Reminder: *implicit* allocator

# Garbage Collection (GC)

## (Automatic Memory Management)

- ❖ *Garbage collection*: automatic reclamation of heap-allocated storage – application never explicitly frees memory

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

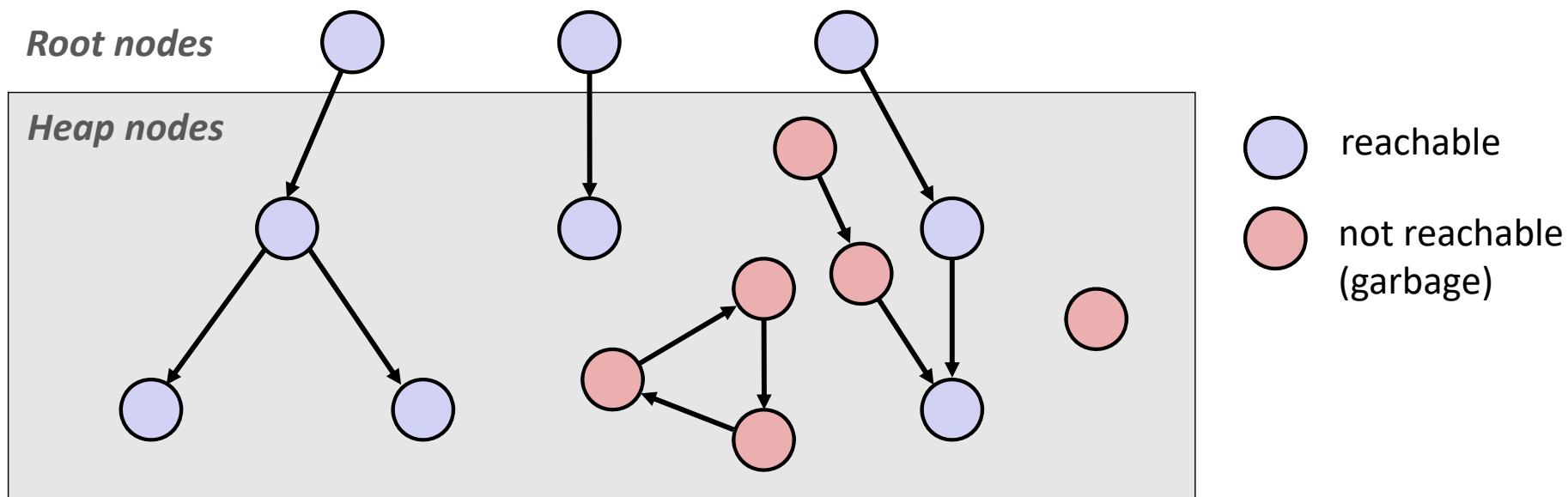
- ❖ Common in implementations of functional languages, scripting languages, and modern object oriented languages:
  - Lisp, Racket, Erlang, ML, Haskell, Scala, Java, C#, Perl, Ruby, Python, Lua, JavaScript, Dart, Mathematica, MATLAB, many more...
- ❖ Variants (“conservative” garbage collectors) exist 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 they are *unreachable* (via pointers in registers/stack/globals)
- ❖ Memory allocator needs to know what is a pointer and what is not – how can it do this?
  - Sometimes with help from the compiler

# 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, stack locations, 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)



# Garbage Collection

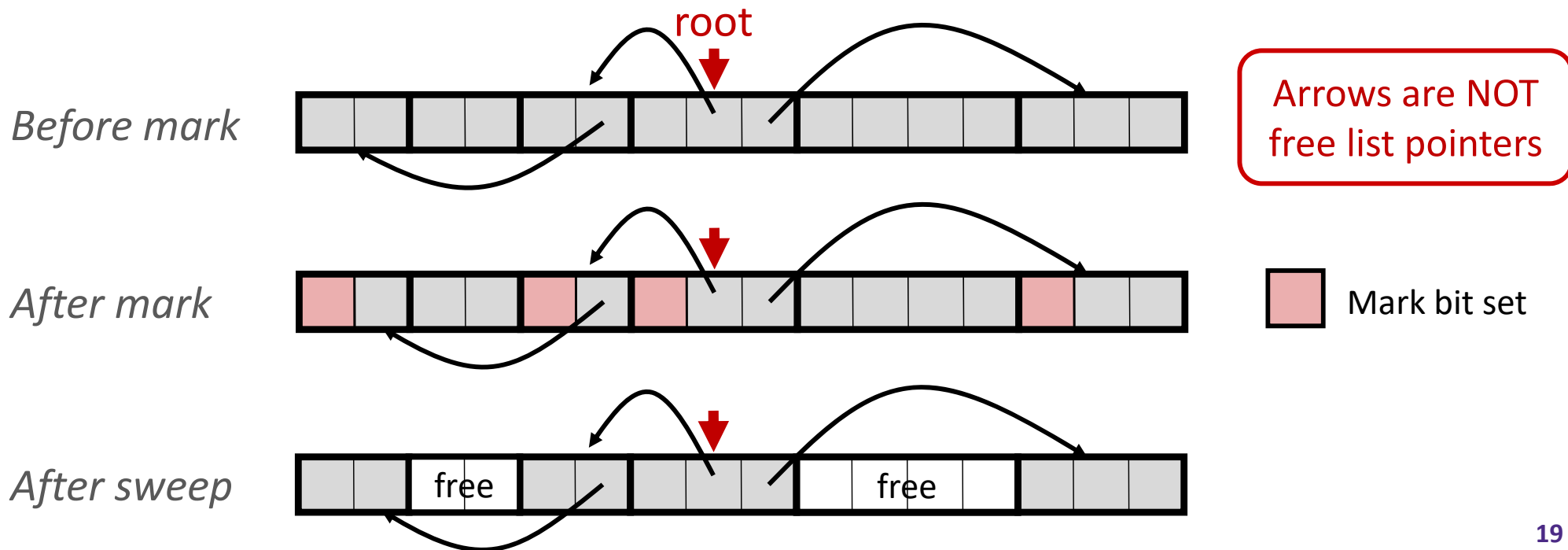
- ❖ Dynamic memory allocator can free blocks if there are no pointers to them
- ❖ How can it know what is a pointer and what is not?
- ❖ We'll make some *assumptions* about pointers:
  - Memory allocator can distinguish pointers from non-pointers
  - All pointers point to the start of a block in the heap
  - Application cannot hide pointers  
(*e.g.*, by coercing them to a `long`, 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)
  - Most allocations become garbage very soon, so focus reclamation work on zones of memory recently allocated.
- ❖ For more information:
  - Jones, Hosking, and Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*, CRC Press, 2012.
  - Jones and Lin, *Garbage Collection: Algorithms for Automatic Dynamic Memory*, John Wiley & Sons, 1996.

# 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 header 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

Non-testable  
Material

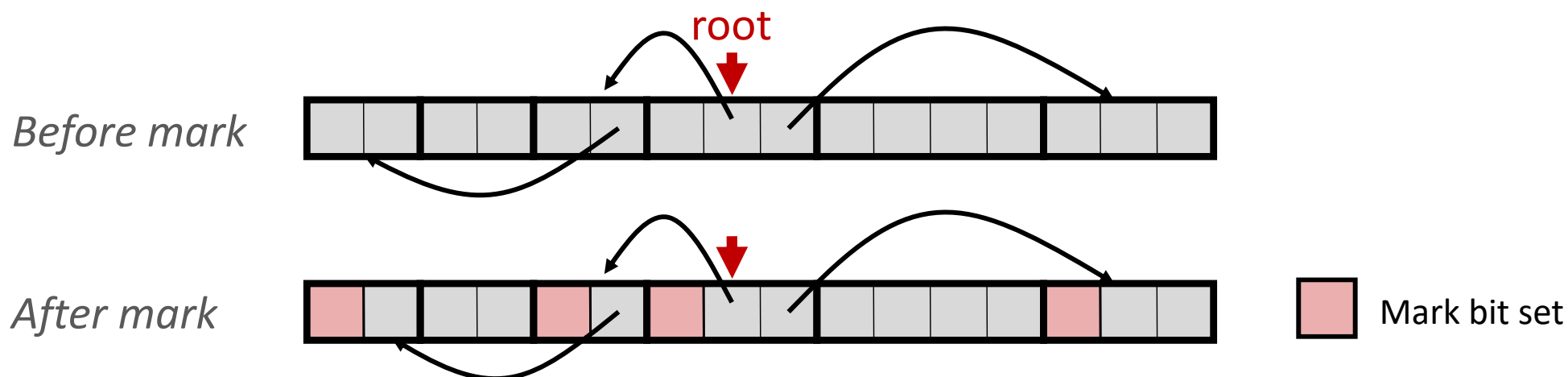
- ❖ Application can use functions to allocate memory:
  - `b=new(n)` returns pointer, `b`, to new block with all locations cleared
  - `b[i]` read location `i` of block `b` into register
  - `b[i]=v` write `v` into location `i` of block `b`
- ❖ Each block will have a header word (accessed at `b[-1]`)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

Non-testable  
Material

- ❖ 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;
}
```



Non-testable  
Material

# Sweep

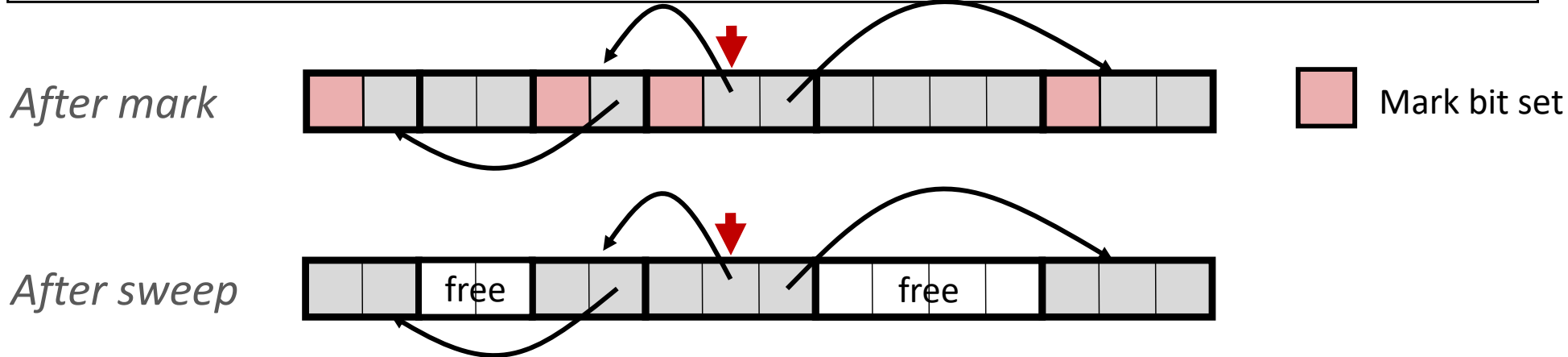
## ❖ Sweep using sizes in headers

```

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

```

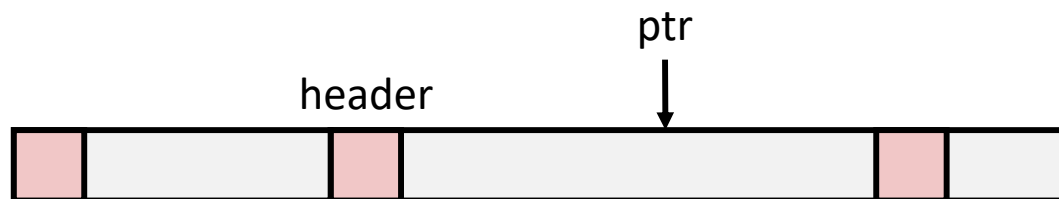
*// ptrs to start & end of heap*  
*// while not at end of heap*  
*// check if block is marked*  
*// if so, reset mark bit*  
*// if not marked, but allocated*  
*// free the block*  
*// adjust pointer to next block*



# Conservative Mark & Sweep in C

Non-testable  
Material

- ❖ Would mark & sweep work in C?
  - `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 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
- In Java, all pointers (*i.e.*, references) point to the starting address of an object structure – the start of an allocated block

# Memory-Related Perils and Pitfalls in C

	Slide	Program stop possible?	Fixes:
<b>A)</b> Dereferencing a non-pointer			
<b>B)</b> Freed block – access again			
<b>C)</b> Freed block – free again			
<b>D)</b> Memory leak – failing to free memory			
<b>E)</b> No bounds checking			
<b>F)</b> Reading uninitialized memory			
<b>G)</b> Referencing nonexistent variable			
<b>H)</b> Wrong allocation size			



# Find That Bug! (Slide 25)

```
char s[8];  
int i;  
  
gets(s);  /* reads "123456789" from stdin */
```

Error  
Type:

Prog stop  
Possible?

Fix:

# Find That Bug! (Slide 26)

```
int* foo() {  
    int val = 0;  
  
    return &val;  
}
```

Error  
Type:

Prog stop  
Possible?

Fix:

# Find That Bug! (Slide 27)

```
int** p;  
  
p = (int**)malloc( N * sizeof(int) );  
  
for (int i = 0; i < N; i++) {  
    p[i] = (int*)malloc( M * sizeof(int) );  
}
```

- N and M defined elsewhere (#define)

Error  
Type:

Prog stop  
Possible?

Fix:

# Find That Bug! (Slide 28)

```
/* return y = Ax */
int* matvec(int** A, int* x) {
    int* y = (int*)malloc( N*sizeof(int) );
    int i, j;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            y[i] += A[i][j] * x[j];

    return y;
}
```

- A is NxN matrix, x is N-sized vector (so product is vector of size N)
- N defined elsewhere (#define)

Error  
Type:

Prog stop  
Possible?

Fix:

# Find That Bug! (Slide 29)

## ❖ The classic scanf bug

- `int scanf(const char *format)`

```
int val;  
...  
scanf("%d", val);
```

Error  
Type:

Prog stop  
Possible?

Fix:

# Find That Bug! (Slide 30)

```
x = (int*)malloc( N * sizeof(int) );  
    // manipulate x  
free(x);  
  
...  
  
y = (int*)malloc( M * sizeof(int) );  
    // manipulate y  
free(x);
```

Error  
Type:

Prog stop  
Possible?

Fix:

# Find That Bug! (Slide 31)

```
x = (int*)malloc( N * sizeof(int) );  
    // manipulate x  
free(x);  
  
    ...  
  
y = (int*)malloc( M * sizeof(int) );  
for (i=0; i<M; i++)  
    y[i] = x[i]++;
```

Error  
Type:

Prog stop  
Possible?

Fix:

# Find That Bug! (Slide 32)

```
typedef struct L {
    int val;
    struct L *next;
} list;

void foo() {
    list *head = (list *) malloc( sizeof(list) );
    head->val = 0;
    head->next = NULL;
    // create and manipulate the rest of the list
    ...
    free(head);
    return;
}
```

Error  
Type:

Prog stop  
Possible?

Fix:



Non-testable  
Material

# Dealing With Memory Bugs

- ❖ Conventional debugger (`gdb`)
  - Good for finding bad pointer dereferences
  - Hard to detect the other memory bugs
- ❖ Debugging `malloc` (UToronto CSRI `malloc`)
  - Wrapper around conventional `malloc`
  - Detects memory bugs at `malloc` and `free` boundaries
    - Memory overwrites that corrupt heap structures
    - Some instances of freeing blocks multiple times
    - Memory leaks
  - Cannot detect all memory bugs
    - Overwrites into the middle of allocated blocks
    - Freeing block twice that has been reallocated in the interim
    - Referencing freed blocks

# Dealing With Memory Bugs (cont.)

Non-testable  
Material

- ❖ Some `malloc` implementations contain checking code
  - Linux glibc malloc: `setenv MALLOC_CHECK_ 2`
  - FreeBSD: `setenv MALLOC_OPTIONS AJR`
- ❖ Binary translator: `valgrind` (Linux), Purify
  - Powerful debugging and analysis technique
  - Rewrites text section of executable object file
  - Can detect all errors as debugging `malloc`
  - Can also check each individual reference at runtime
    - Bad pointers
    - Overwriting
    - Referencing outside of allocated block

# What about Java or ML or Python or ...?

Non-testable  
Material

- ❖ In *memory-safe languages*, most of these bugs are impossible
  - Cannot perform arbitrary pointer manipulation
  - Cannot get around the type system
  - Array bounds checking, null pointer checking
  - Automatic memory management
- ❖ But one of the bugs we saw earlier is possible. Which one?

# Memory Leaks with GC

- ❖ Not because of forgotten `free` — we have GC!
- ❖ Unneeded “leftover” roots keep objects reachable
- ❖ *Sometimes* nullifying a variable is not needed for correctness but is for performance
- ❖ Example: Don't leave big data structures you're done with in a static field

