

Memory Allocation III

CSE 351 Winter 2019

Instructors:

Max Willsey

Luis Ceze

Teaching Assistants:

Britt Henderson

Lukas Joswiak

Josie Lee

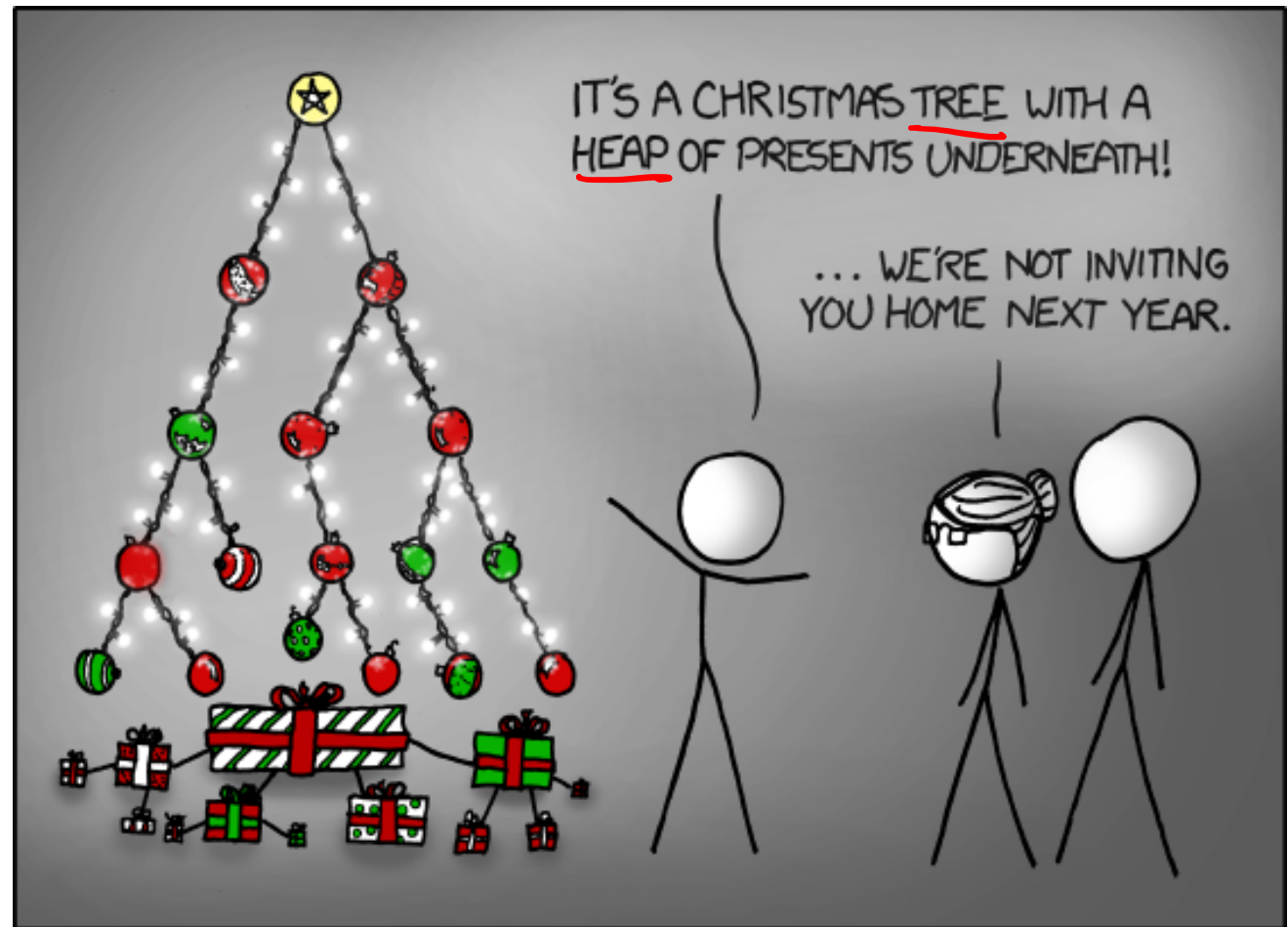
Wei Lin

Daniel Snitkovsky

Luis Vega

Kory Watson

Ivy Yu



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

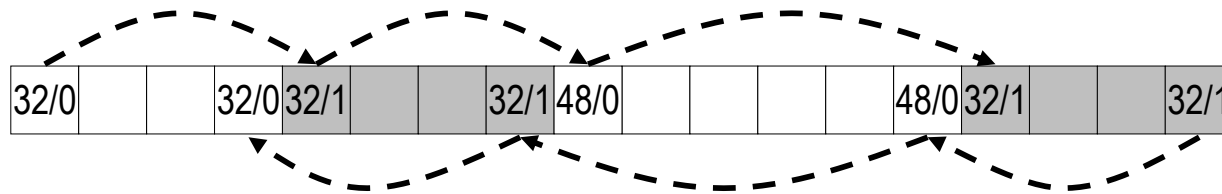
Administrivia

- ❖ Homework 5 due Wednesday, March 13
- ❖ Lab 5 due Friday, March 15

- ❖ **Final Exam:** Tue, March 19, 8:30-10:20am in KNE 130
 - Review in next week's section

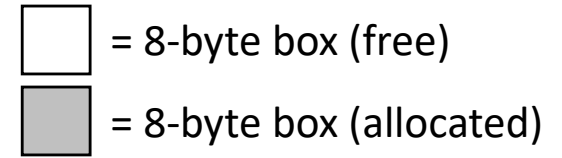
- ❖ Course feedback

Implicit Free List Review Questions



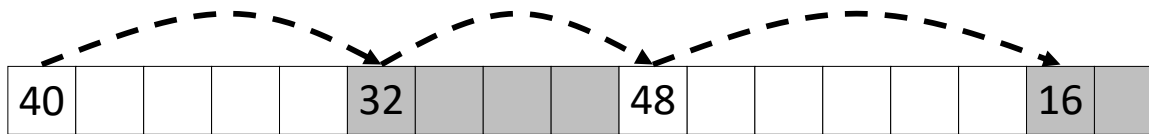
- ❖ What is the block header? What do we store and how?
- ❖ What are boundary tags and why do we need them?
- ❖ When we coalesce free blocks, how many neighboring blocks do we need to check on either side? Why is this?
- ❖ If I want to check the size of the n -th block forward from the current block, how many memory accesses do I make?

Keeping Track of Free Blocks

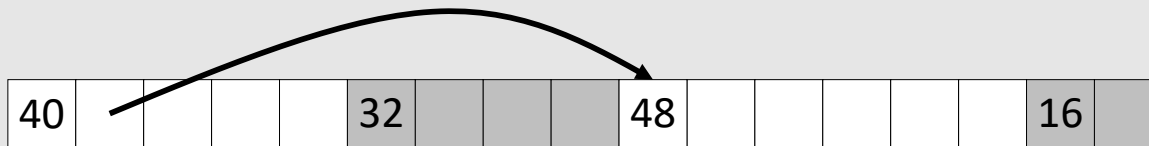


1) *Implicit free list* using length – links all blocks using math

- No actual pointers, and must check each block if allocated or free



2) *Explicit free list* among only the free blocks, using pointers



3) *Segregated free list*

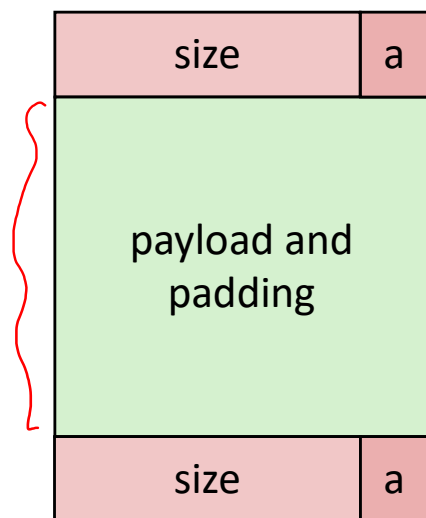
- Different free lists for different size “classes”

4) *Blocks sorted by size*

- Can use a balanced binary tree (e.g. red-black tree) with pointers within each free block, and the length used as a key

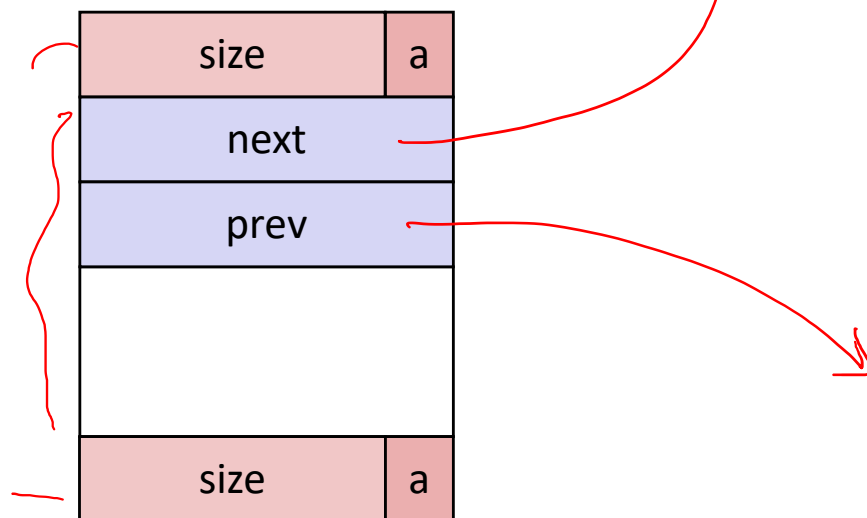
Explicit Free Lists

Allocated block:



(same as implicit free list)

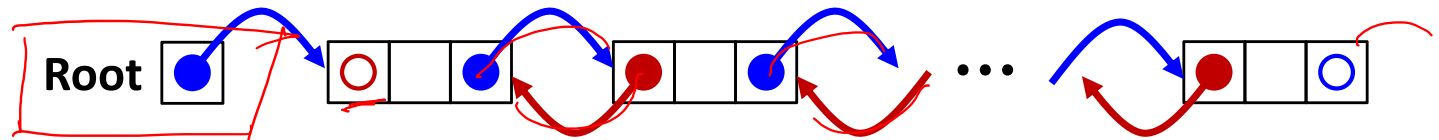
Free block:



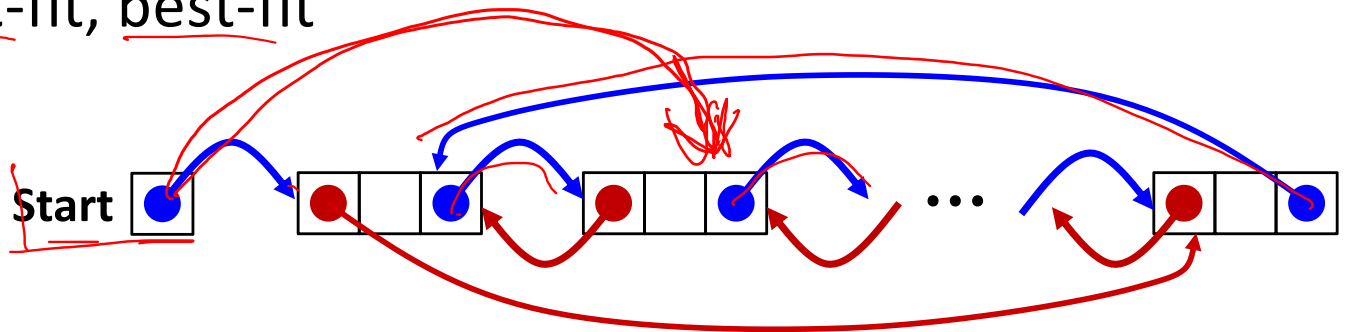
- ❖ Use list(s) of *free* blocks, rather than implicit list of *all* blocks
 - The “next” free block could be anywhere in the heap
 - So we need to store next/previous pointers, not just sizes
 - Since we only track free blocks, so we can use “payload” for pointers
 - Still need boundary tags (header/footer) for coalescing

Doubly-Linked Lists

❖ Linear



- Needs head/root pointer
- First node prev pointer is NULL
- Last node next pointer is NULL
- Good for first-fit, best-fit



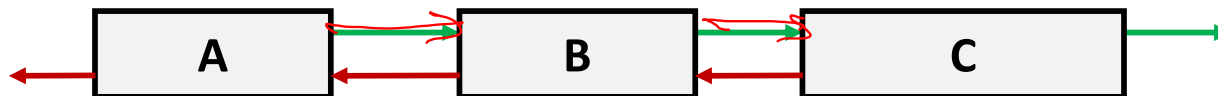
❖ Circular

- Still have pointer to tell you which node to start with
- No NULL pointers (term condition is back at starting point)
- Good for next-fit, best-fit

Explicit Free Lists

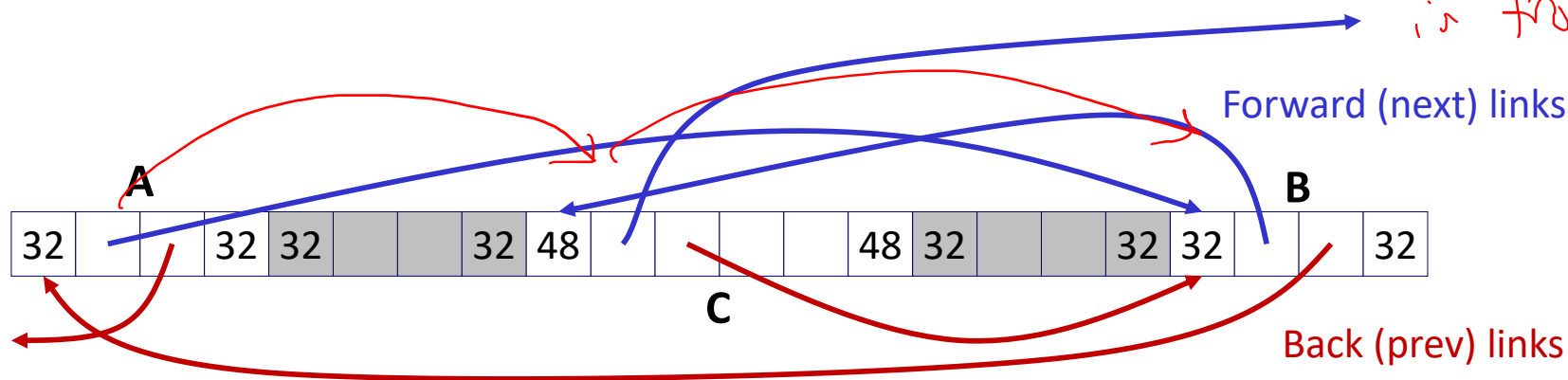
next prev

- ❖ **Logically:** doubly-linked list



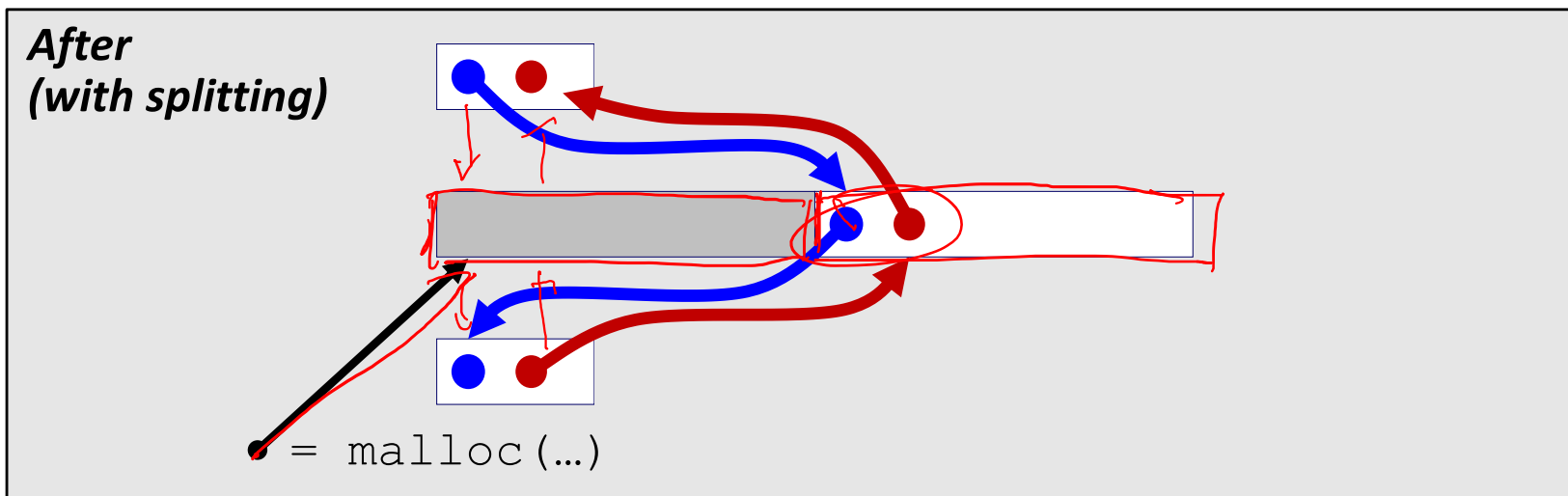
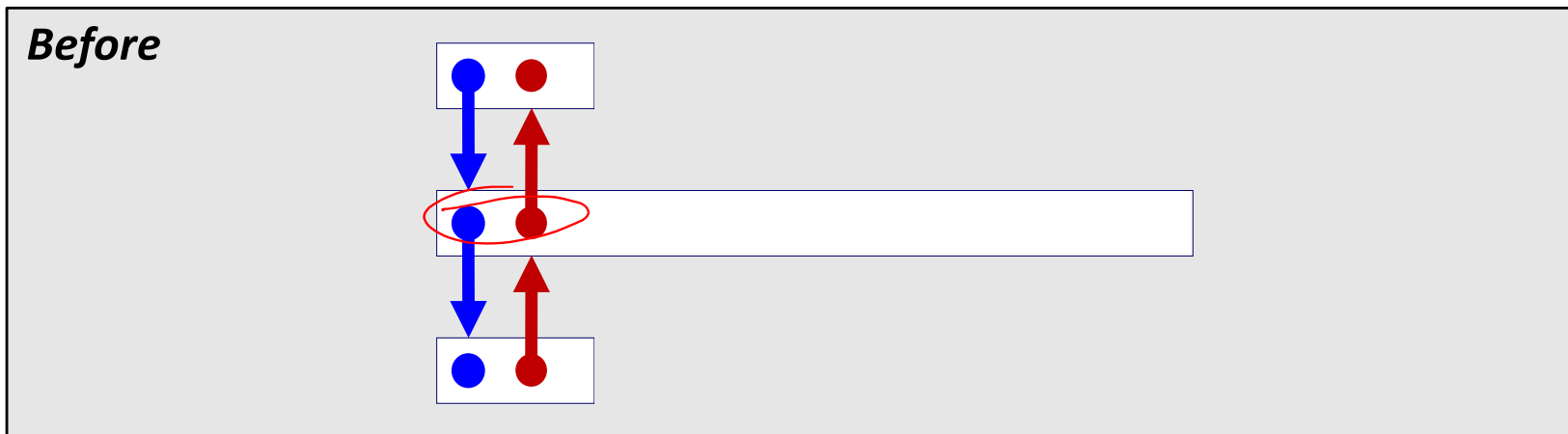
- ❖ **Physically:** blocks can be in any order

beard is front



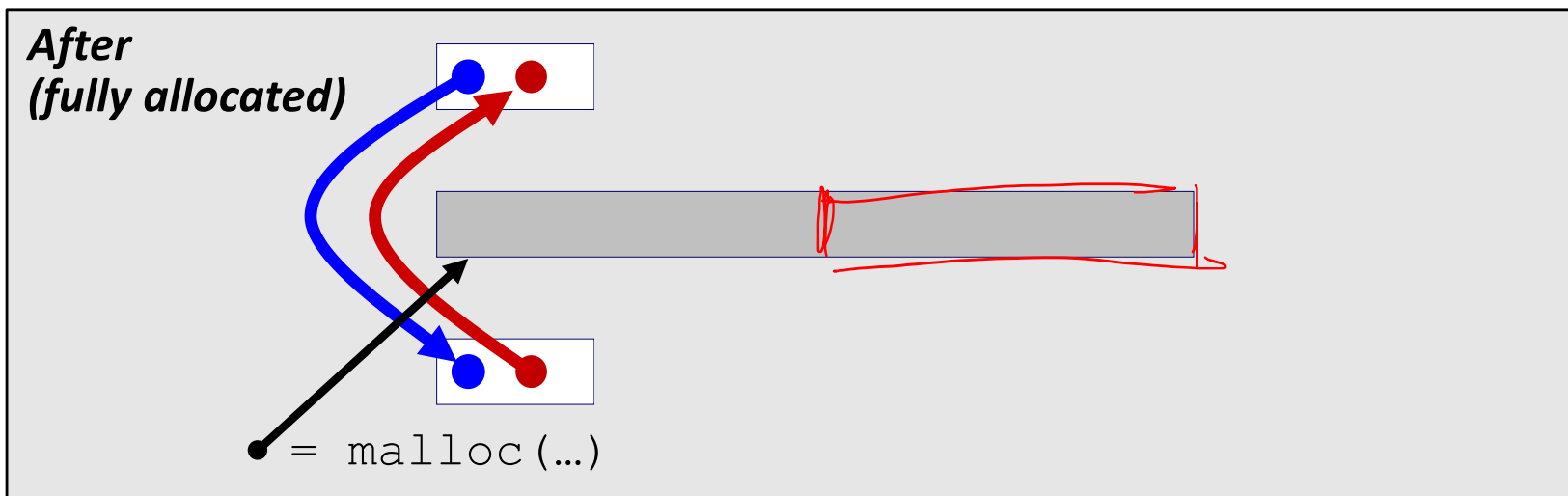
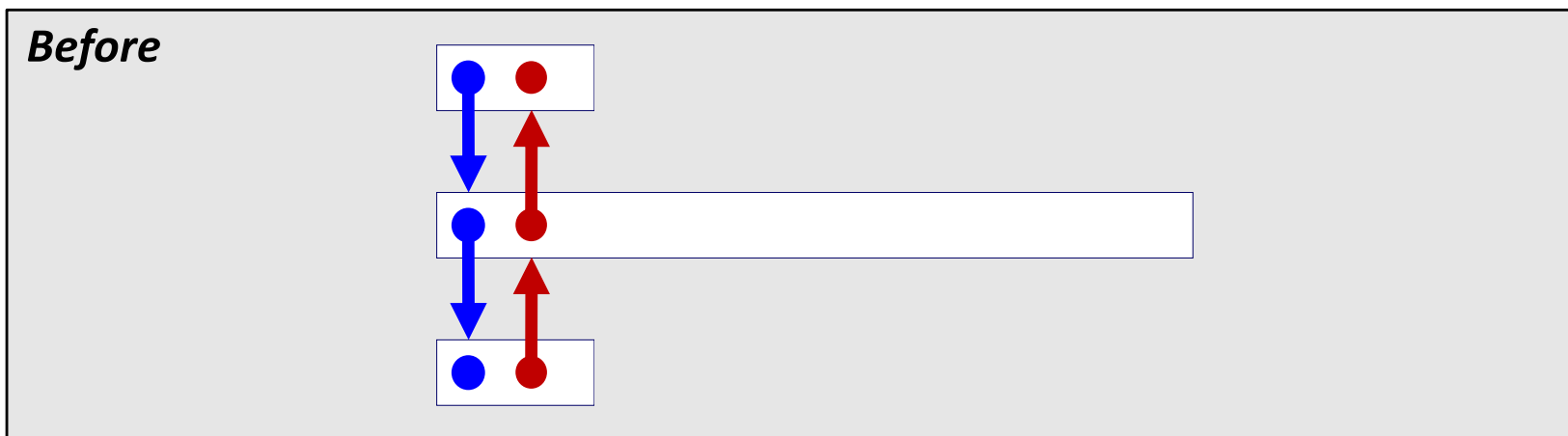
Allocating From Explicit Free Lists

Note: These diagrams are not very specific about where inside a block a pointer points. In reality we would always point to one place (e.g. start/header of a block).



Allocating From Explicit Free Lists

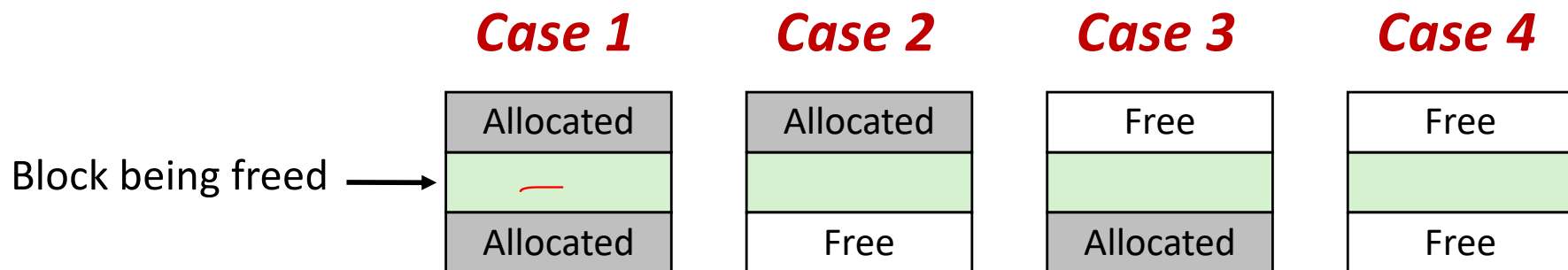
Note: These diagrams are not very specific about where inside a block a pointer points. In reality we would always point to one place (e.g. start/header of a block).



Freeing With Explicit Free Lists

- ❖ *Insertion policy*: Where in the free list do you put the newly freed block?
 - **LIFO (last-in-first-out) policy**
 - Insert freed block at the beginning (head) of the free list
 - Pro: simple and constant time
 - Con: studies suggest fragmentation is worse than the alternative
 - **Address-ordered policy**
 - Insert freed blocks so that free list blocks are always in address order:
 $address(previous) < address(current) < address(next)$
 - Con: requires linear-time search
 - Pro: studies suggest fragmentation is better than the alternative

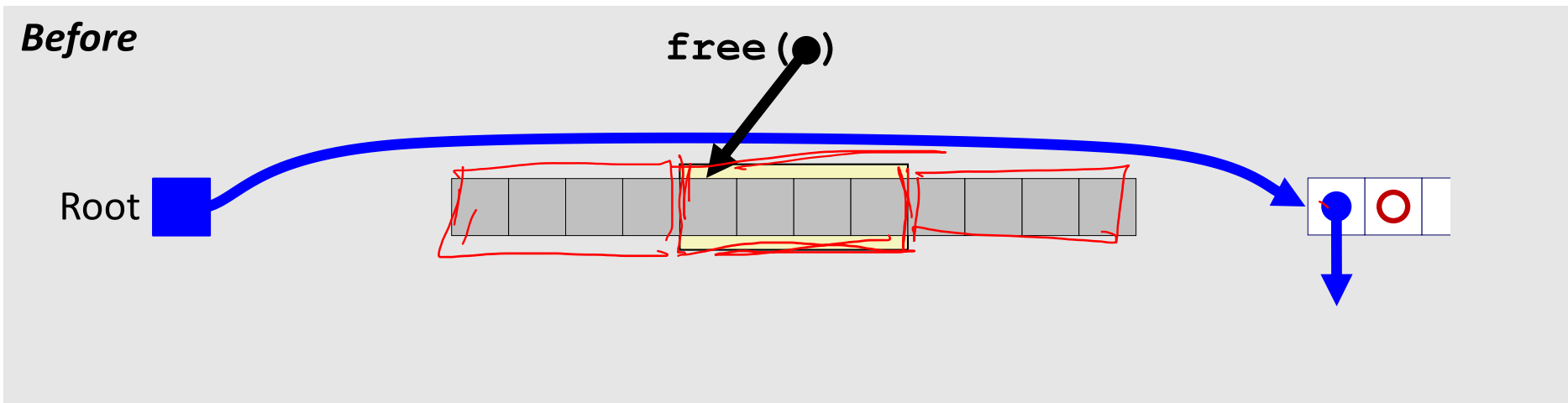
Coalescing in Explicit Free Lists



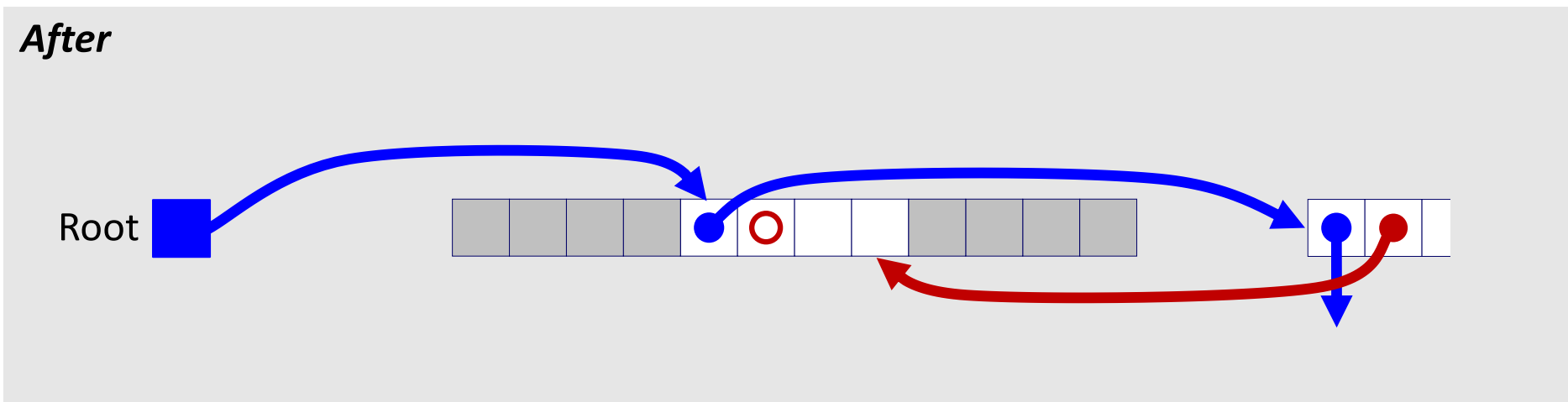
- ❖ Neighboring free blocks are *already part of the free list*
 - 1) Remove old block from free list
 - 2) Create new, larger coalesced block
 - 3) Add new block to free list (insertion policy)
- ❖ How do we tell if a neighboring block is free?

Freeing with LIFO Policy (Case 1)

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

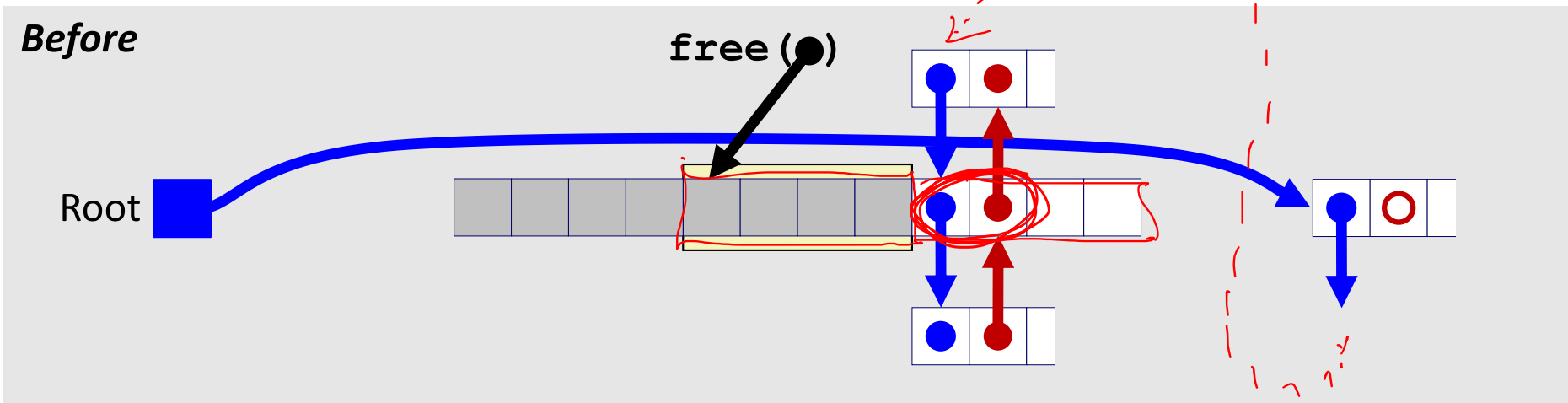


❖ Insert the freed block at the root of the list

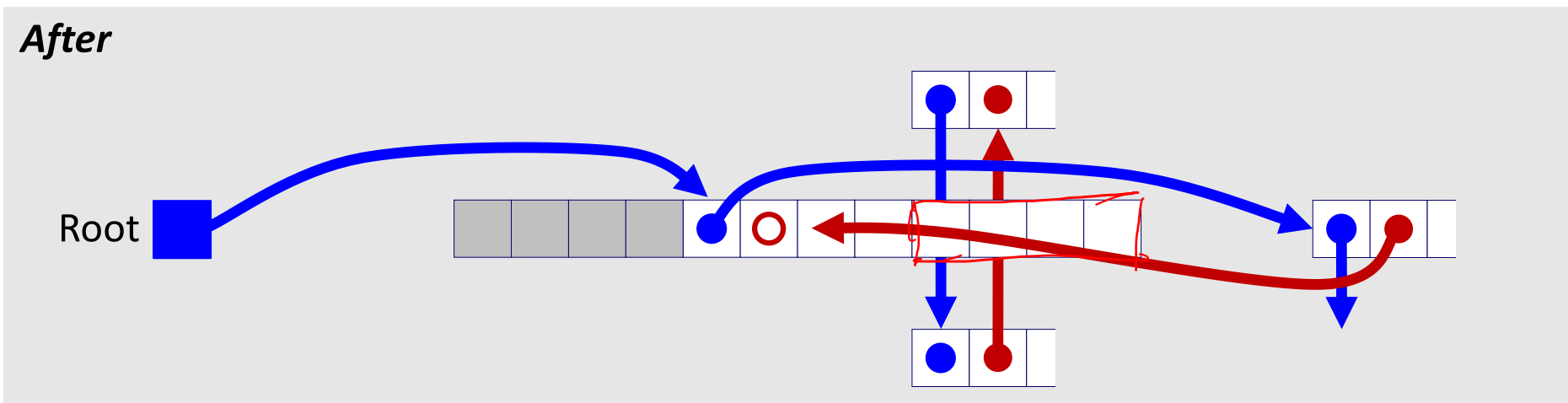


Freeing with LIFO Policy (Case 2)

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

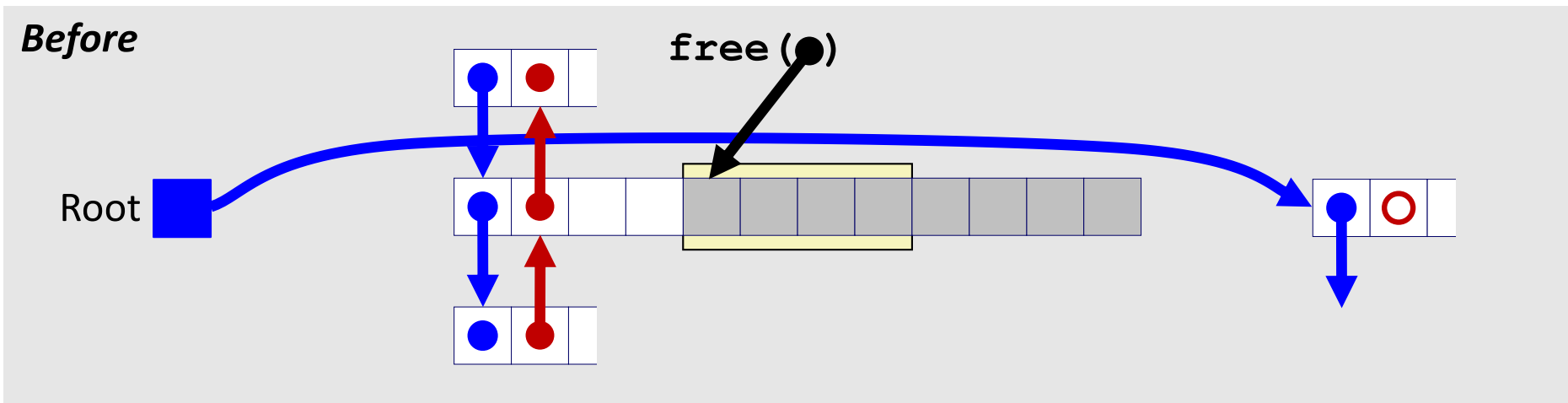


- ❖ Splice successor block out of list, coalesce both memory blocks, and insert the new block at the root of the list

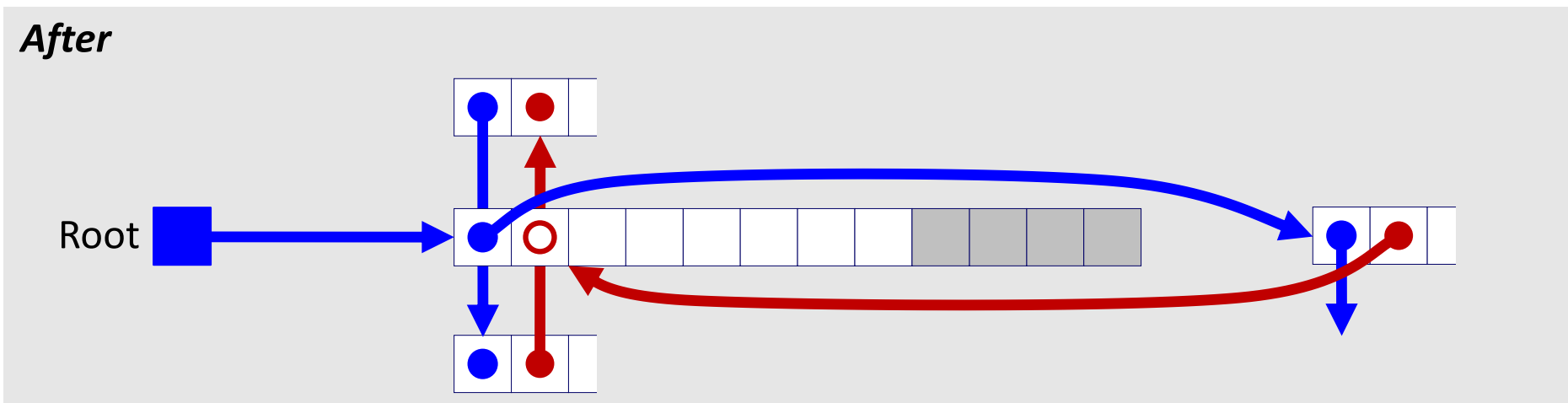


Freeing with LIFO Policy (Case 3)

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

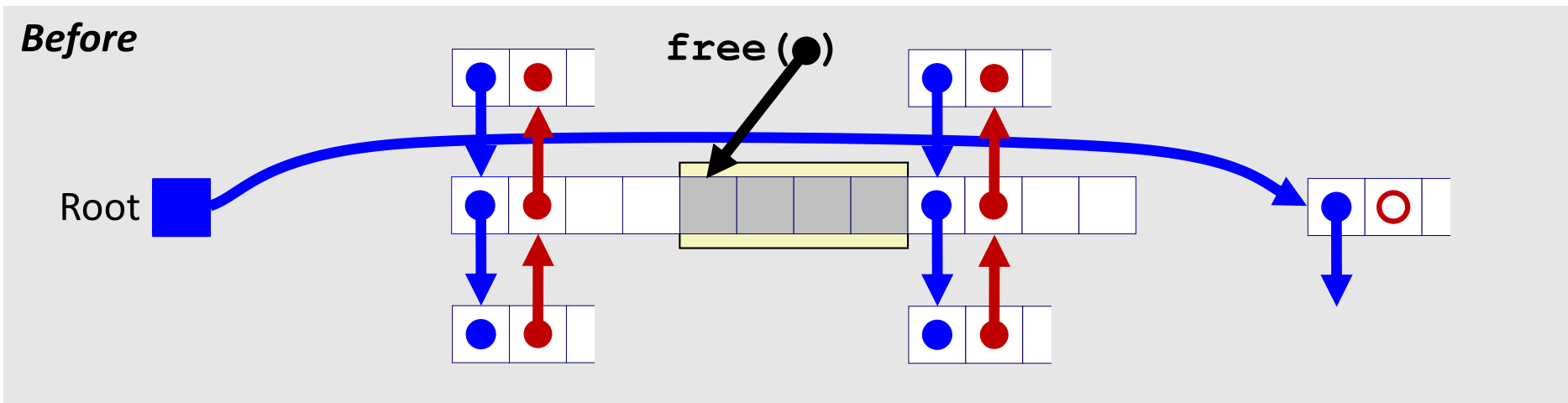


- ❖ Splice predecessor block out of list, coalesce both memory blocks, and insert the new block at the root of the list

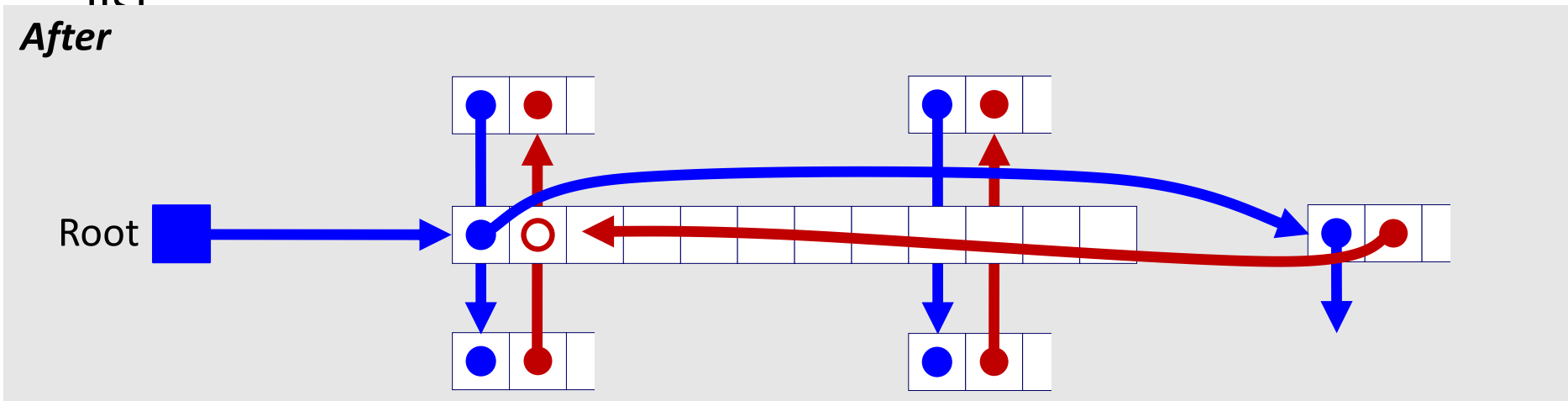


Freeing with LIFO Policy (Case 4)

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

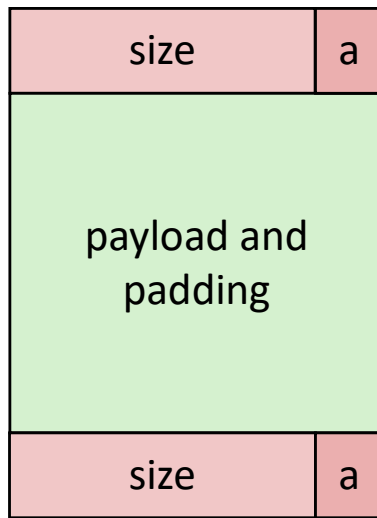


- ❖ Splice predecessor and successor blocks out of list, coalesce all 3 memory blocks, and insert the new block at the root of the list



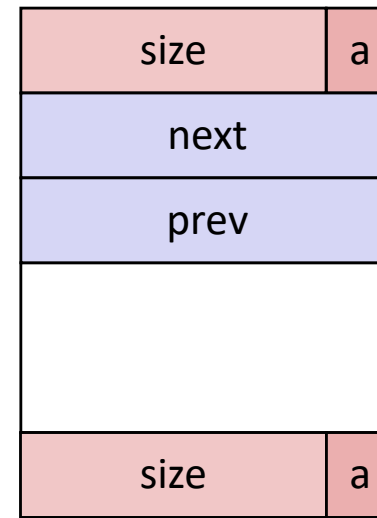
Do we always need the boundary tags?

Allocated block:



(same as implicit free list)

Free block:



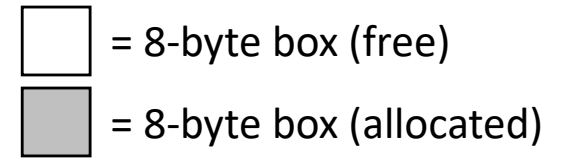
❖ Lab 5 suggests no...

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

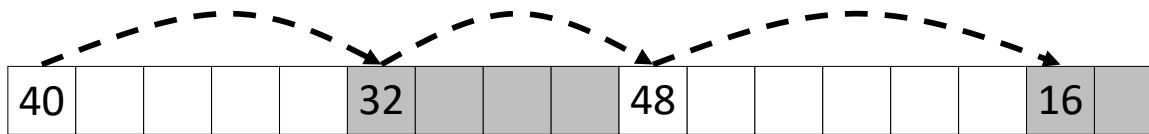
- ❖ Most common use of explicit lists is in conjunction with segregated free lists
 - Keep multiple linked lists of different size classes, or possibly for different types of objects

Keeping Track of Free Blocks

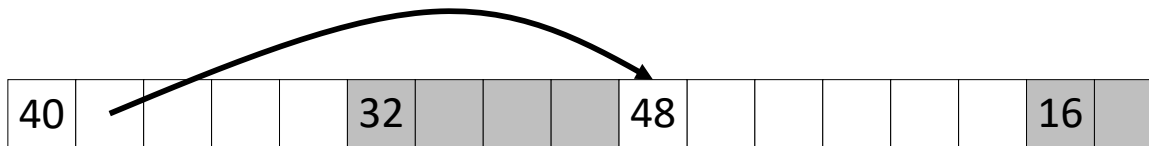


1) *Implicit free list* using length – links all blocks using math

- No actual pointers, and must check each block if allocated or free



2) *Explicit free list* among only the free blocks, using pointers



3) *Segregated free list*

- Different free lists for different size “classes”

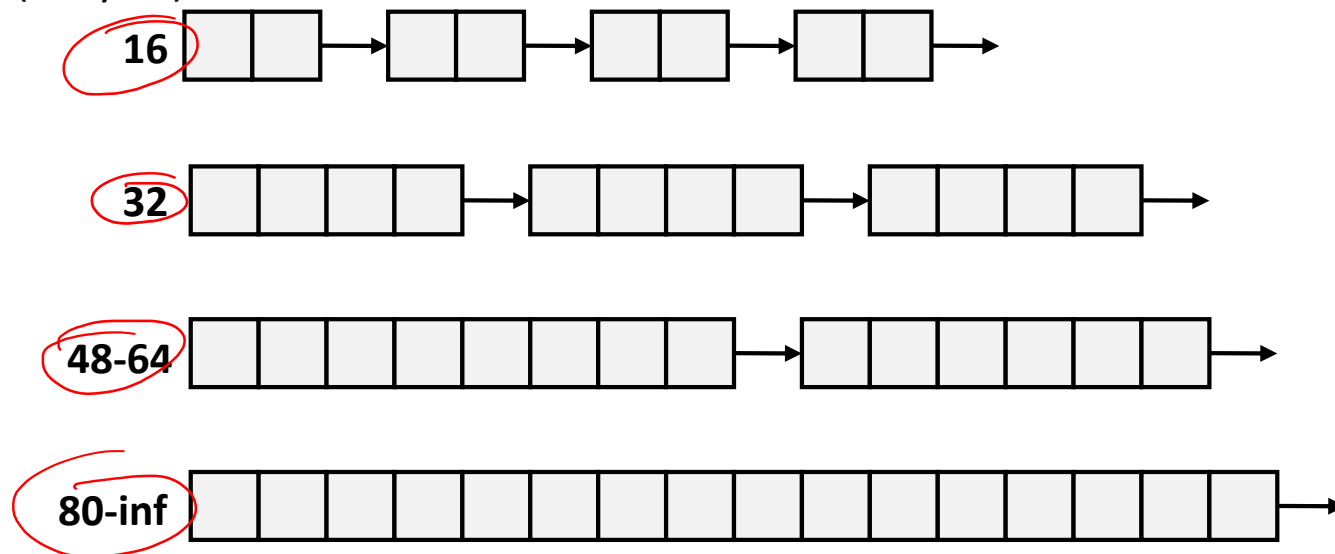
4) *Blocks sorted by size*

- Can use a balanced binary tree (e.g. red-black tree) with pointers within each free block, and the length used as a key

Segregated List (SegList) Allocators

- ❖ Each *size class* of blocks has its own free list
- ❖ Organized as an array of free lists

Size class
(in bytes)



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

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*”, 2nd 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)

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**

Wouldn't it be nice...

- ❖ If we never had to free memory?
- ❖ Do you free objects in Java?
 - Python, Javascript, Ruby, Go, etc...
 - 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

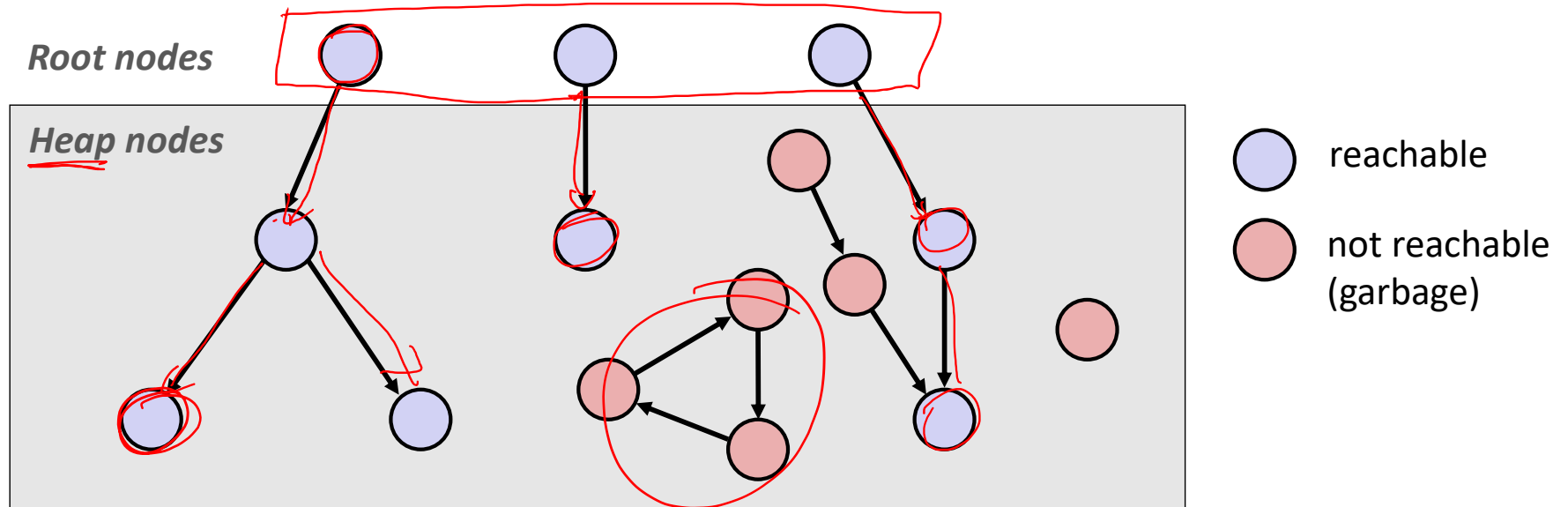
Garbage Collection

- ❖ 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

↑
heap

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

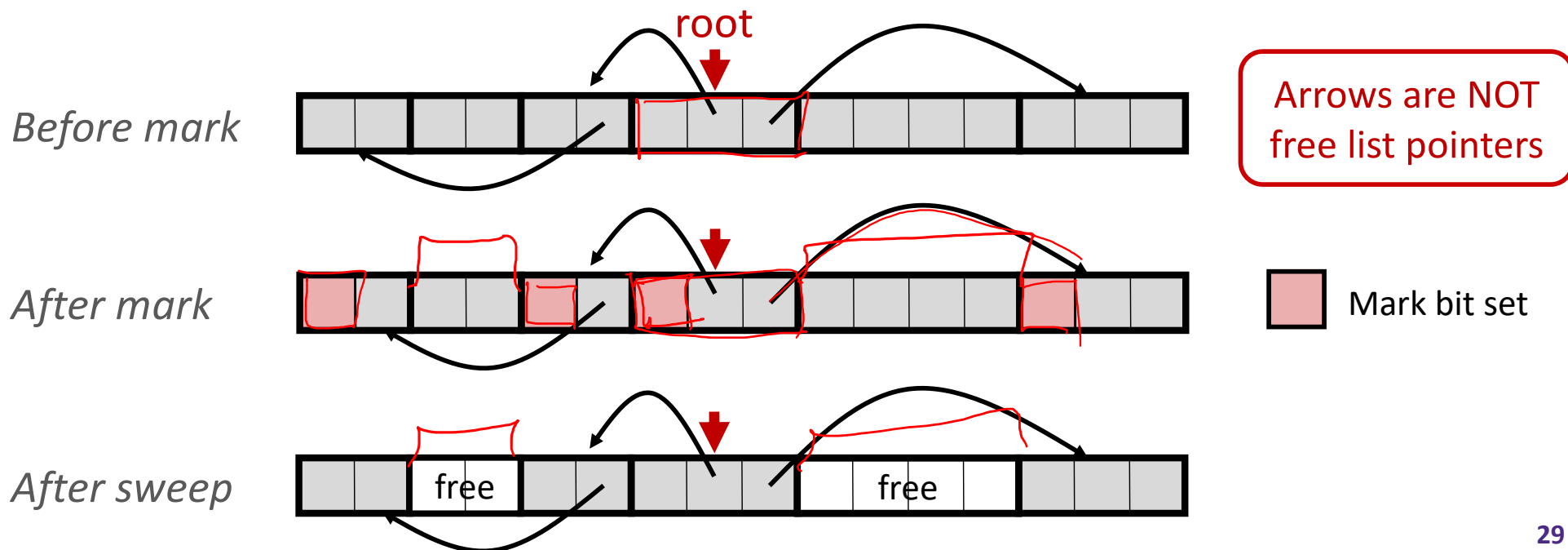
- ❖ Dynamic memory allocator can free blocks if there are no pointers to them
- ❖ How can it know what is a pointer and what is not?
- ❖ We'll make some *assumptions* about pointers:
 - Memory allocator can distinguish pointers from non-pointers
 - All pointers point to the start of a block in the heap
 - 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:
 - $b = \text{new}(n)$ 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 $b[-1]$)
- ❖ Computer Functions used by the garbage collector:
 - $\text{is_ptr}(p)$ determines whether p is a pointer to a block
 - $\text{length}(p)$ returns length of block pointed to by p , not including header
 - $\text{get_roots}()$ returns all the roots

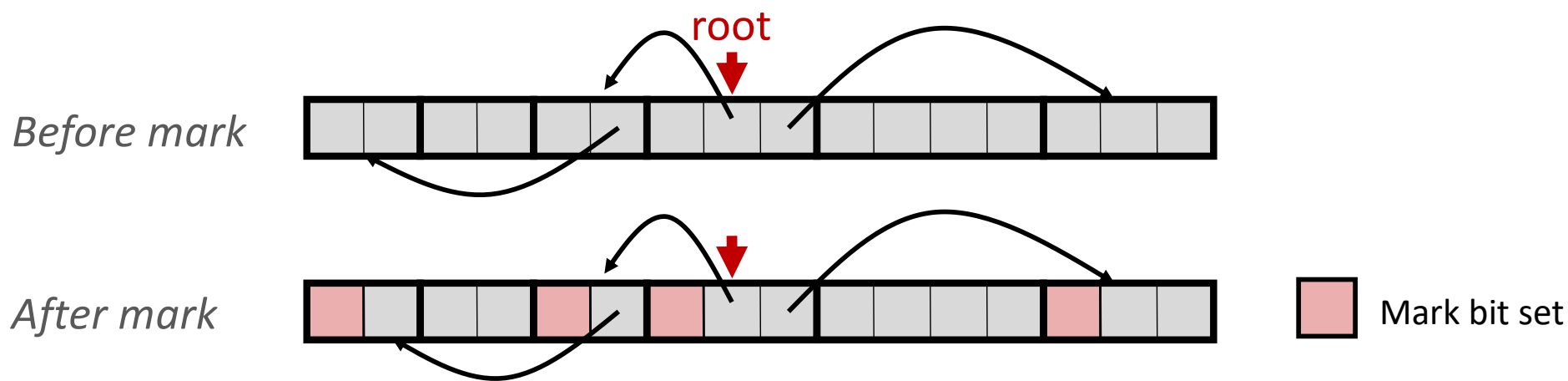
Non-testable
Material

Mark

- ❖ Mark using depth-first traversal of the memory graph

```

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



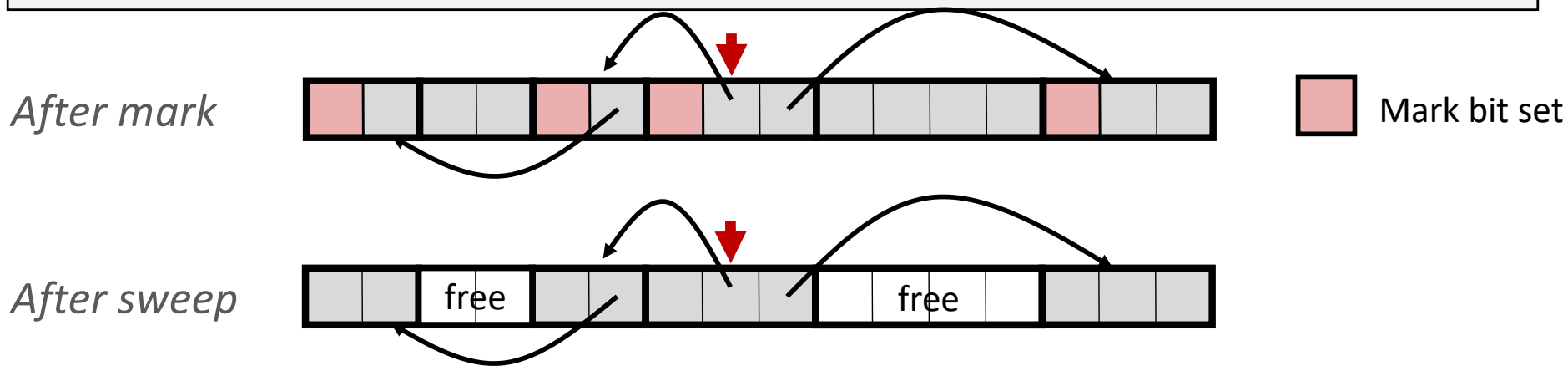
Non-testable
Material

Sweep

❖ Sweep using sizes in headers

```

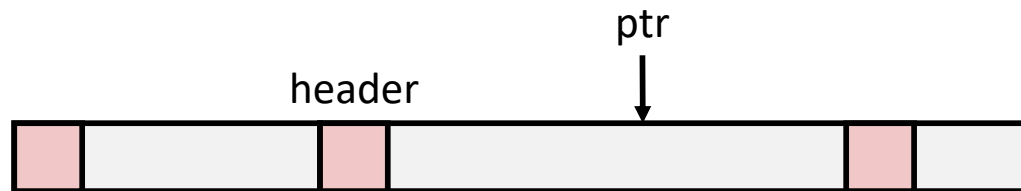
ptr sweep(ptr p, ptr end) { // ptrs to start & end of heap
  while (p < end) { // while not at end of heap
    if (markBitSet(p)) // check if block is marked
      clearMarkBit(p); // if so, reset mark bit
    else if (allocateBitSet(p)) // if not marked, but allocated
      free(p); // free the block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



Conservative Mark & Sweep in C

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

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

Find That Bug!

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

Error
Type:

Prog stop
Possible?

Fix:

Find That Bug!

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

Error
Type:

Prog stop
Possible?

Fix:

Find That Bug!

```
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!

```
/* 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!

❖ The classic scanf bug

- `int scanf(const char *format)`

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

Error
Type:

Prog stop
Possible?

Fix:

Find That Bug!

```
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!

```
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!

```
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:

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.)

- ❖ 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 ...?

- ❖ 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

