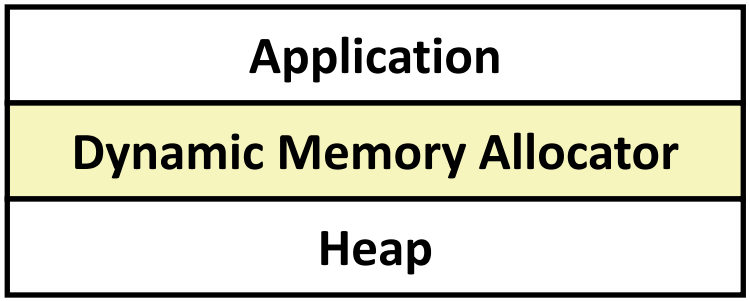

WELCOME
TO
15-213
14-513
18-213
18-613
15-513

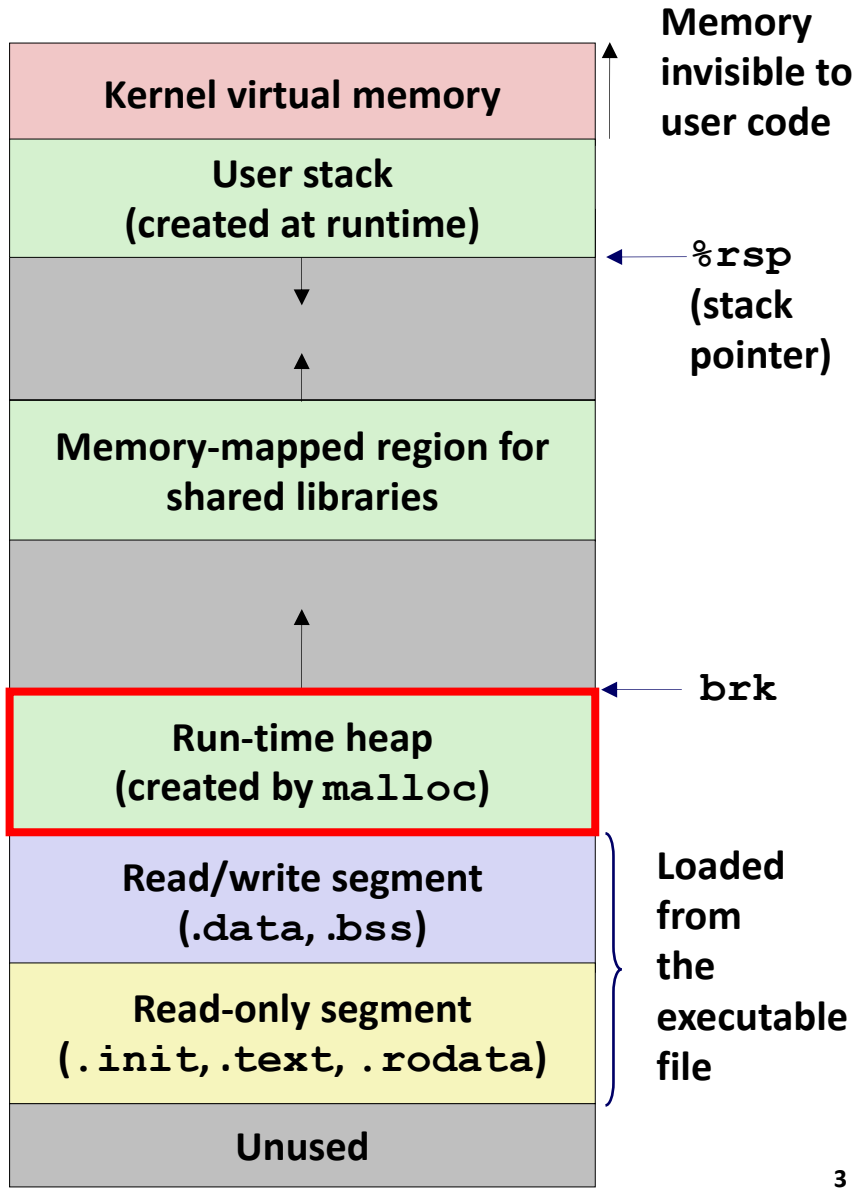
# Dynamic Memory Allocation: Advanced Concepts

18-213/18-613: Introduction to Computer Systems
14th Lecture, February 26, 2025

# Review: Dynamic Memory Allocation

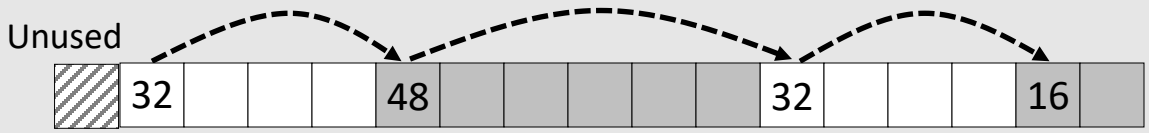



- Programmers use *dynamic memory allocators* (such as `malloc`) to acquire virtual memory (VM) at run time.
  - for data structures whose size is only known at runtime
- Dynamic memory allocators manage an area of process VM known as the *heap*.



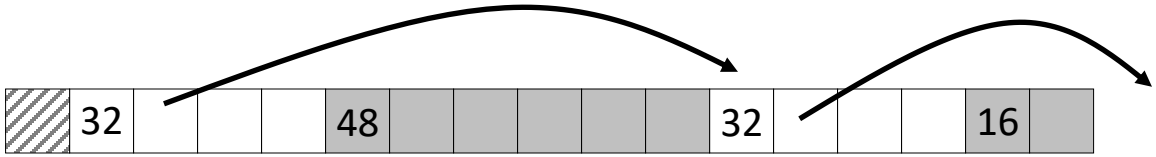
# Review: Keeping Track of Free Blocks

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




Need to tag each block as allocated/free

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



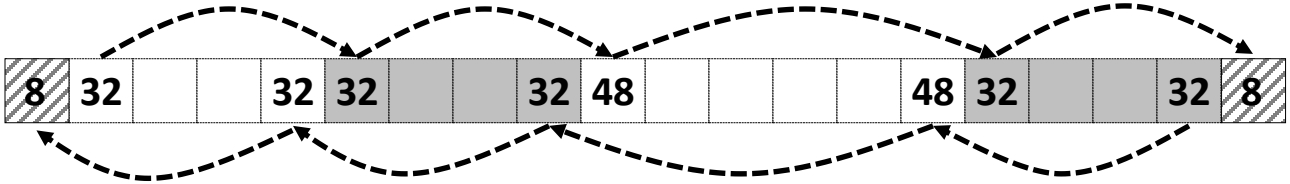

Need space for pointers

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

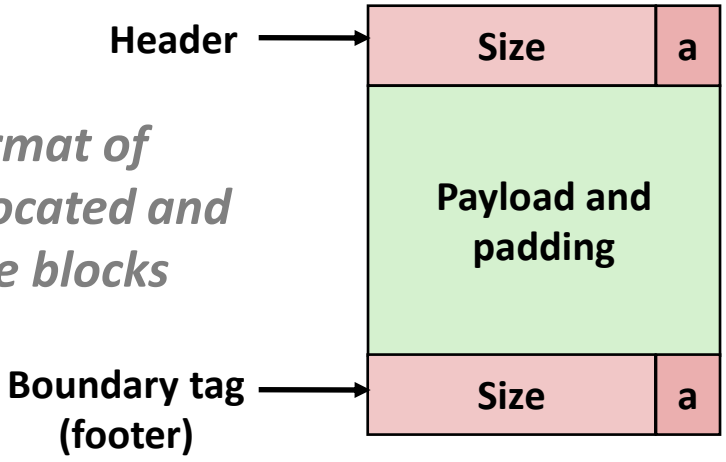
- **Method 4: *Blocks sorted by size***
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

# Review: Boundary Tags for Coalescing

- ***Boundary tags***
  - Replicate size/allocated word at "bottom" (end) of free blocks
  - Allows us to traverse the "list" backwards, but requires extra space
  - Important and general technique!




*Format of allocated and free blocks*




**a = 1: Allocated block**
**a = 0: Free block**

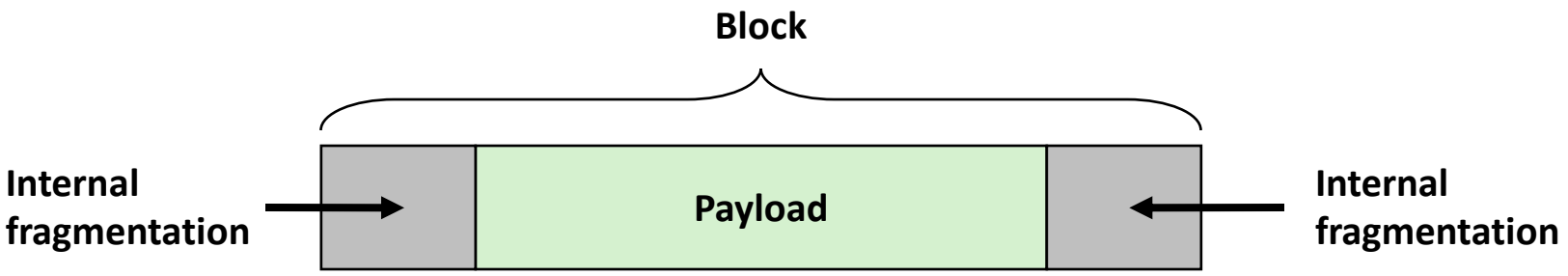
**Size: Total block size**

**Payload: Application data (allocated blocks only)**

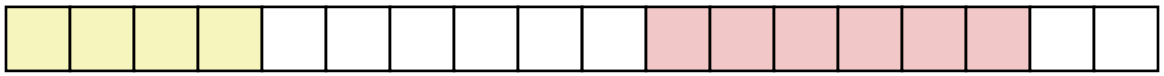
**Disadvantage: Internal fragmentation**

# Review: Internal vs. External Fragmentation

- For a given block, *internal fragmentation* occurs if payload is smaller than block size



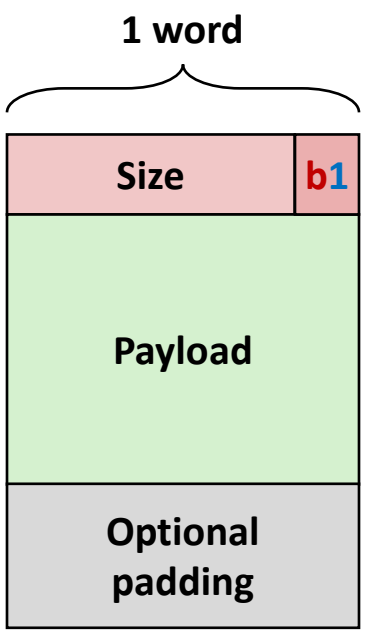

- *External fragmentation* occurs when there is enough aggregate heap memory, but no single free block is large enough



```
p4 = malloc(7*sizeof(size_t))
```

# Review: No Boundary Tag for Allocated Blocks

- Boundary tag needed only for free blocks
- When sizes are multiples of 16, have 4 spare bits

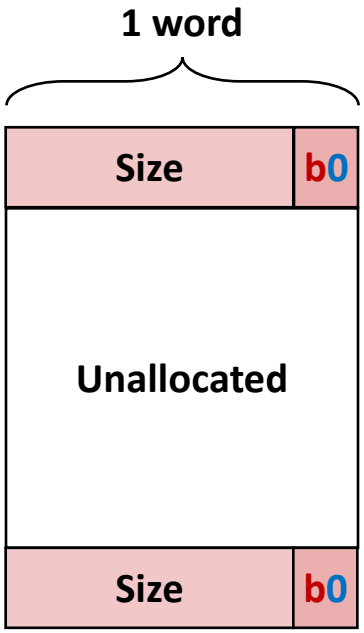



Allocated Block

a = 1: Allocated block
a = 0: Free block
b = 1: Previous block is allocated
b = 0: Previous block is free

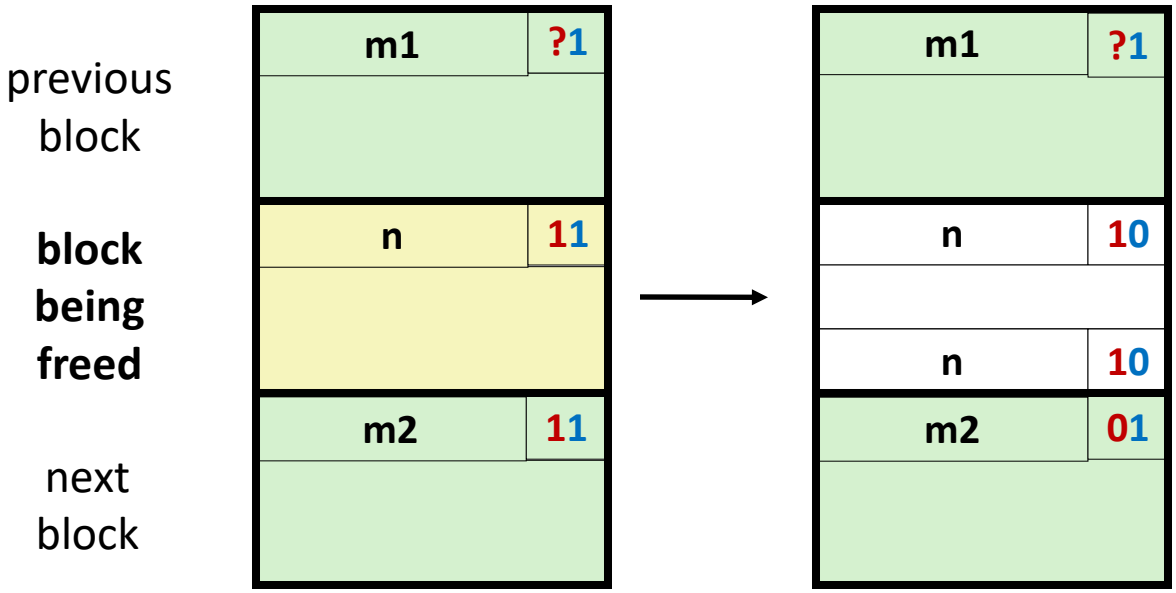
Size: block size

Payload: application data



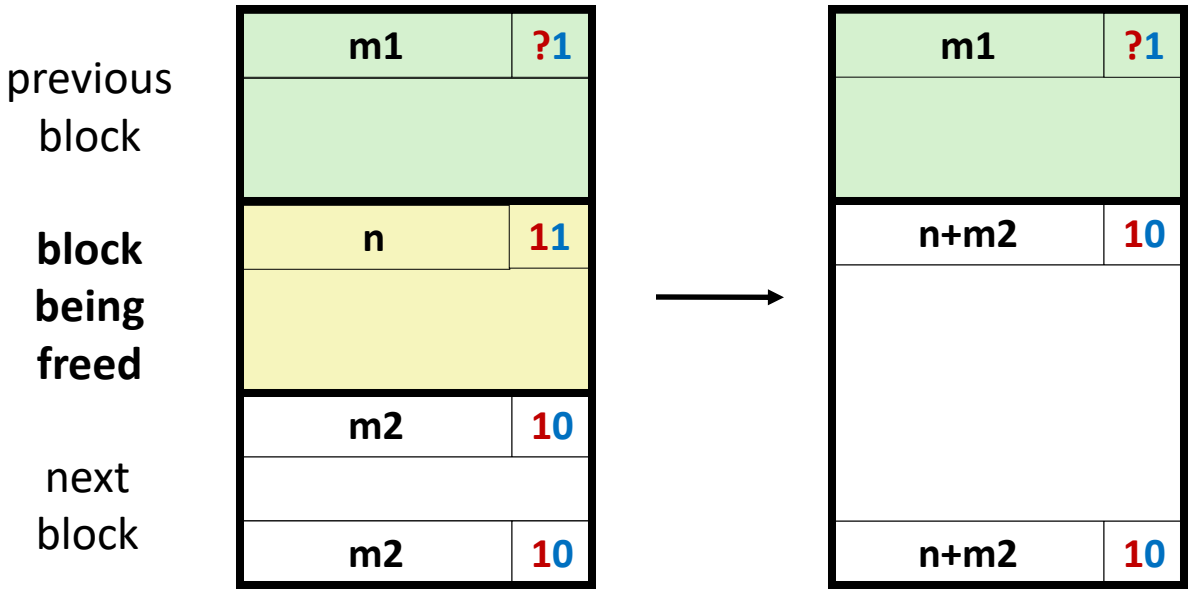

Free Block

# No Boundary Tag for Allocated Blocks (Case 1)



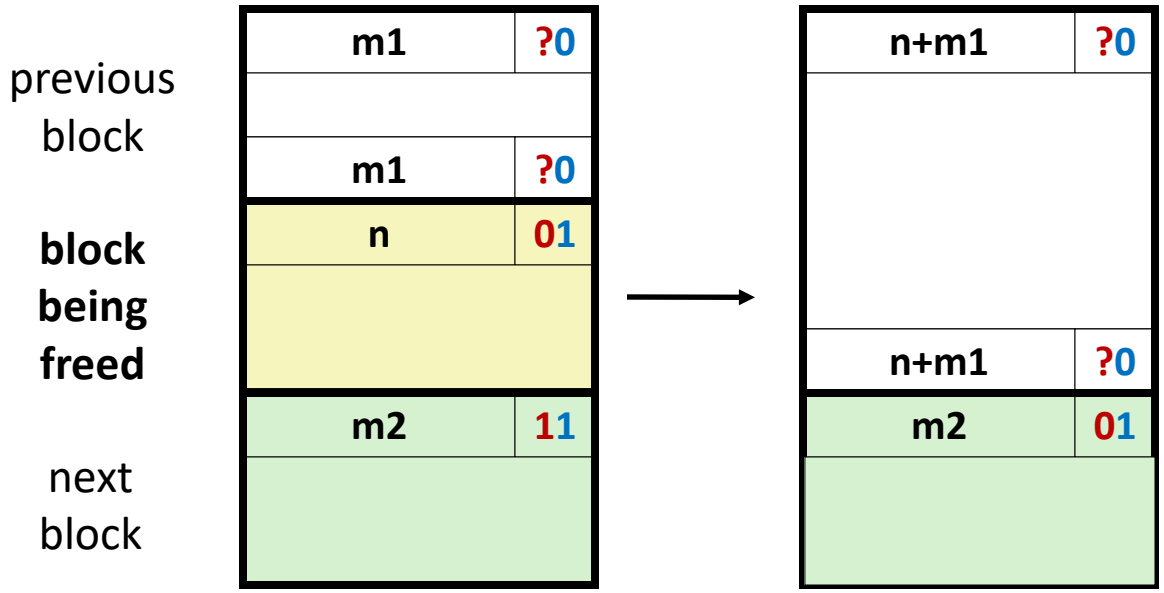

**Header: Use 2 bits (address bits always zero due to alignment):**
**(previous block allocated)<<1 | (current block allocated)**

# No Boundary Tag for Allocated Blocks (Case 2)



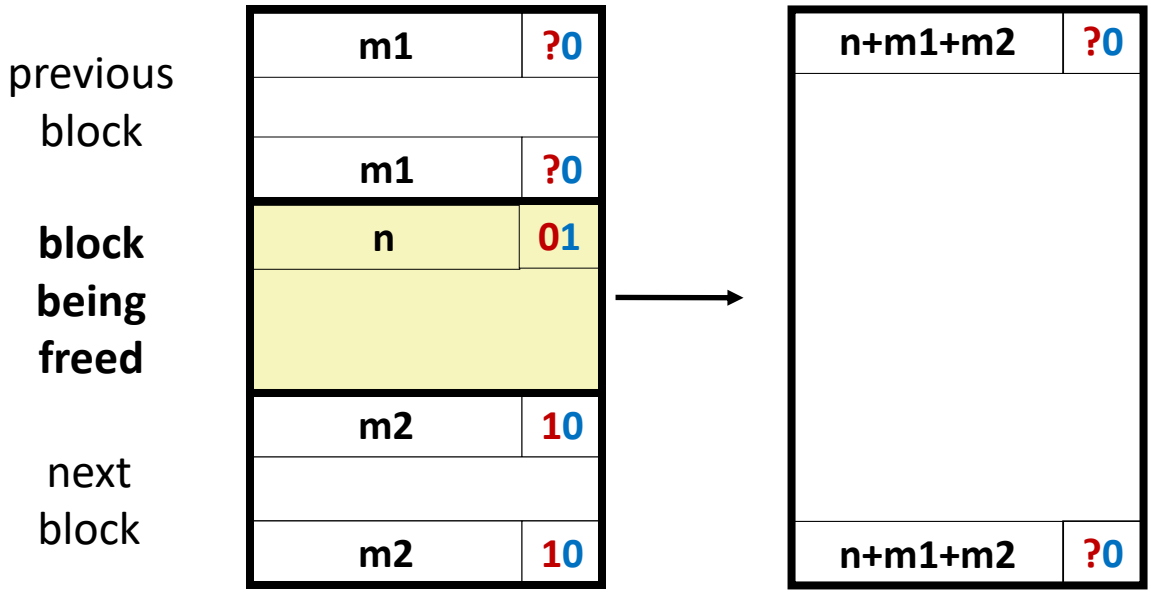

**Header:** Use 2 bits (address bits always zero due to alignment):
(previous block allocated)<<1 | (current block allocated)

# No Boundary Tag for Allocated Blocks (Case 3)




**Header:** Use 2 bits (address bits always zero due to alignment):
(previous block allocated)<<1 | (current block allocated)

# No Boundary Tag for Allocated Blocks (Case 4)




**Header:** Use 2 bits (address bits always zero due to alignment):
(previous block allocated)<<1 | (current block allocated)

# Implicit Lists Summary

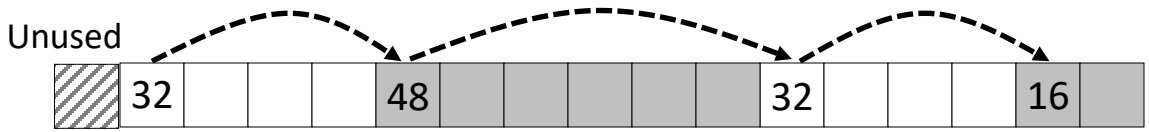
- **Implementation: very simple**
- **Allocate cost:**
  - linear time worst case
- **Free cost:**
  - constant time worst case
  - even with coalescing
- **Memory Overhead:**
  - Depends on placement policy
  - Strategies include first fit, next fit, and best fit
- **Not used in practice for `malloc/free` because of linear-time allocation**
  - used in many special purpose applications
- **However, the concepts of splitting and boundary tag coalescing are general to *all* allocators**

# Today

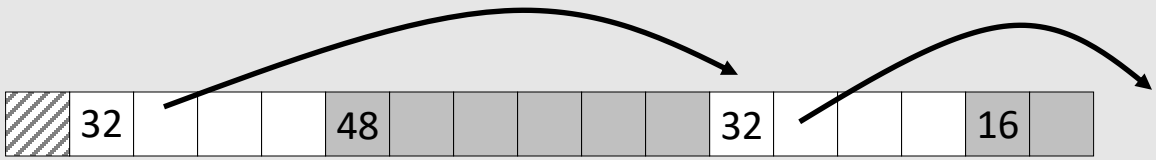
- Implicit free lists (review)
- **Explicit free lists** CSAPP 9.9.13
- **Segregated free lists** CSAPP 9.9.14
- **Memory-related perils and pitfalls** CSAPP 9.11

# Keeping Track of Free Blocks

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



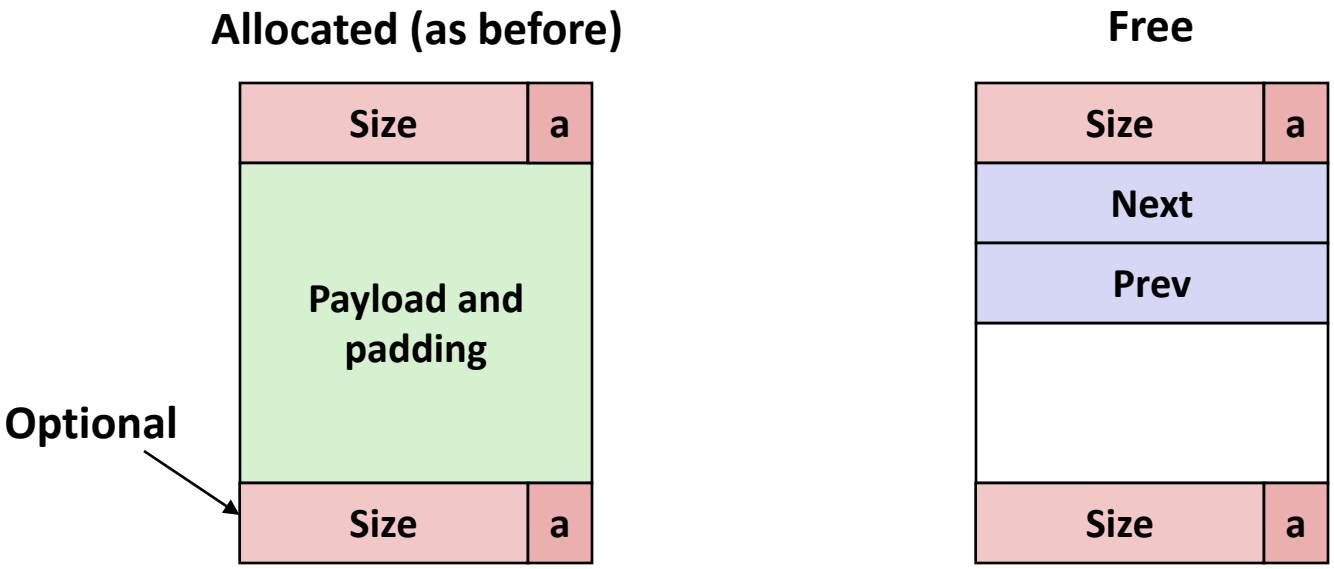

- **Method 2: *Explicit list* among the free blocks using pointers**




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

- **Method 4: *Blocks sorted by size***
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

# Explicit Free Lists

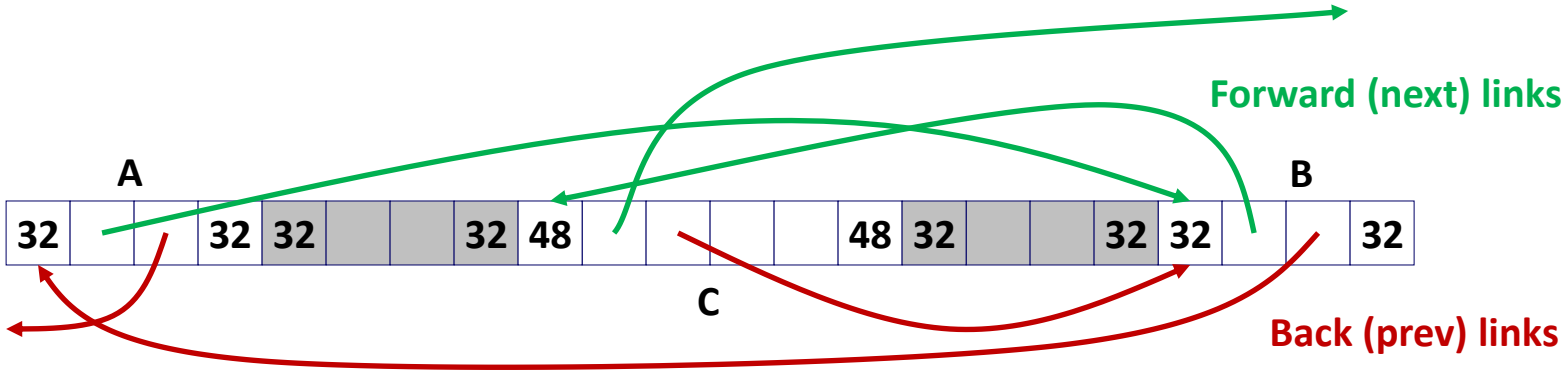



- **Maintain list(s) of *free* blocks, not *all* blocks**
  - Luckily we track only free blocks, so we can use payload area
  - The "next" free block could be anywhere
    - So we need to store forward/back pointers, not just sizes
  - Still need boundary tags for coalescing
    - To find adjacent blocks according to memory order

# Explicit Free Lists

- Logically:

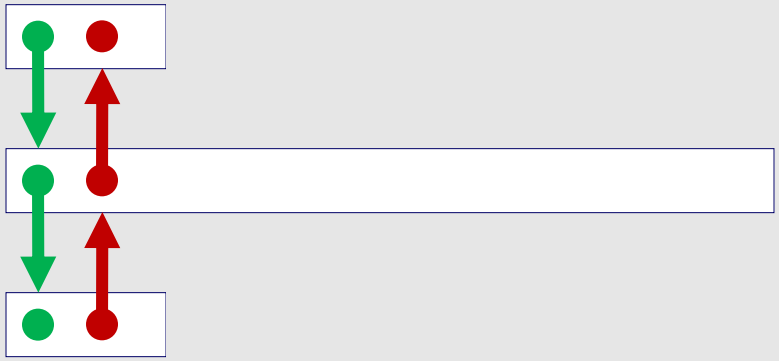

- Physically: blocks can be in any order



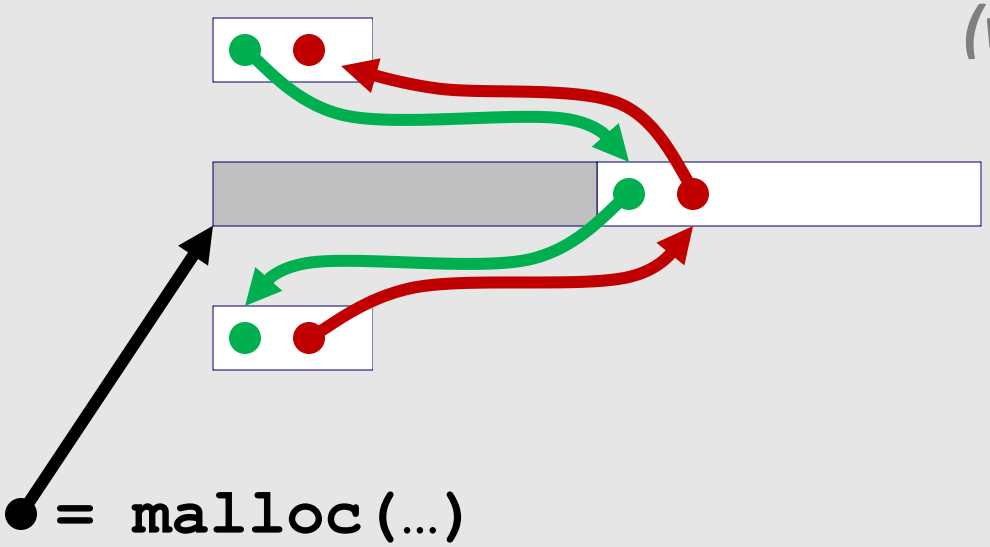
# Allocating From Explicit Free Lists

conceptual graphic

*Before*



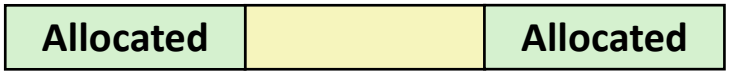
*After* *(with splitting)*



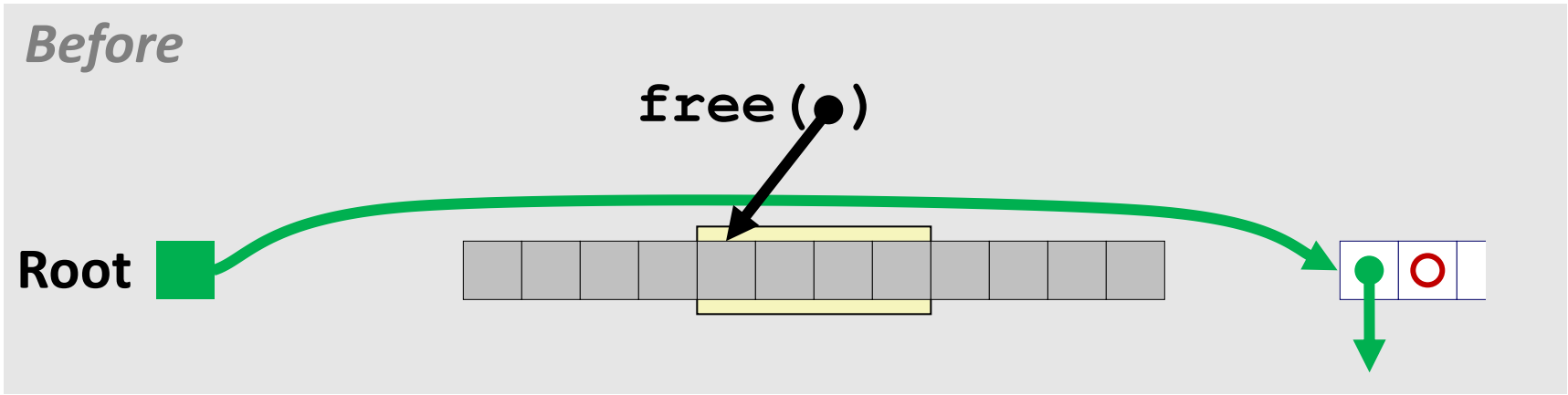
# Freeing With Explicit Free Lists

- ***Insertion policy*: Where in the free list do you put a newly freed block?**
- **Unordered**
  - LIFO (last-in-first-out) policy
    - Insert freed block at the beginning of the free list
  - FIFO (first-in-first-out) policy
    - Insert freed block at the end of the free list
  - ***Pro:*** simple and constant time
  - ***Con:*** studies suggest fragmentation is worse than address ordered
- **Address-ordered policy**
  - Insert freed blocks so that free list blocks are always in address order:
    *addr(prev) < addr(curr) < addr(next)*
  - ***Con:*** requires search
  - ***Pro:*** studies suggest fragmentation is lower than LIFO/FIFO

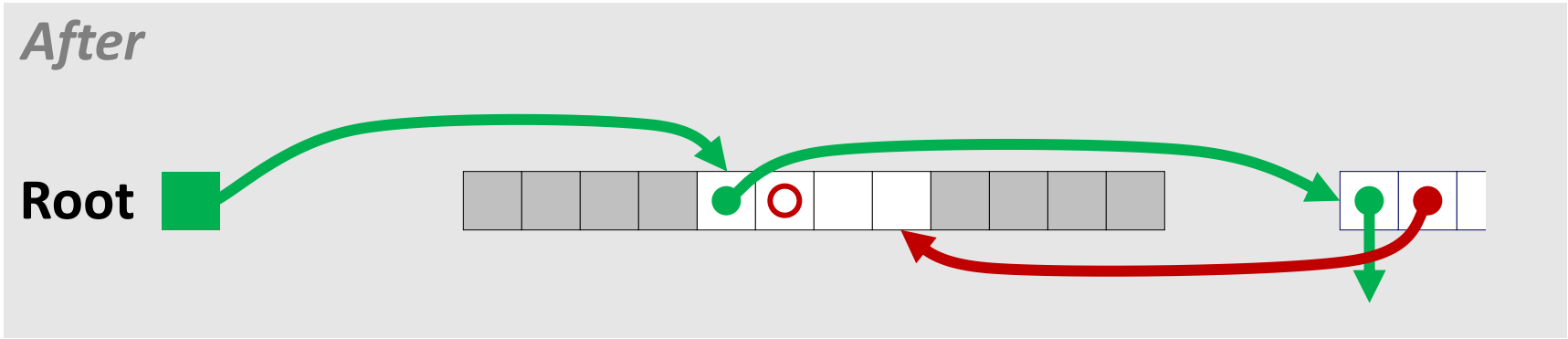
# Freeing With a LIFO Policy (Case 1)



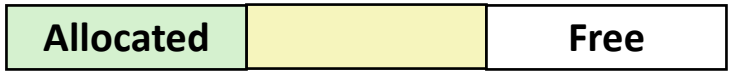

conceptual graphic



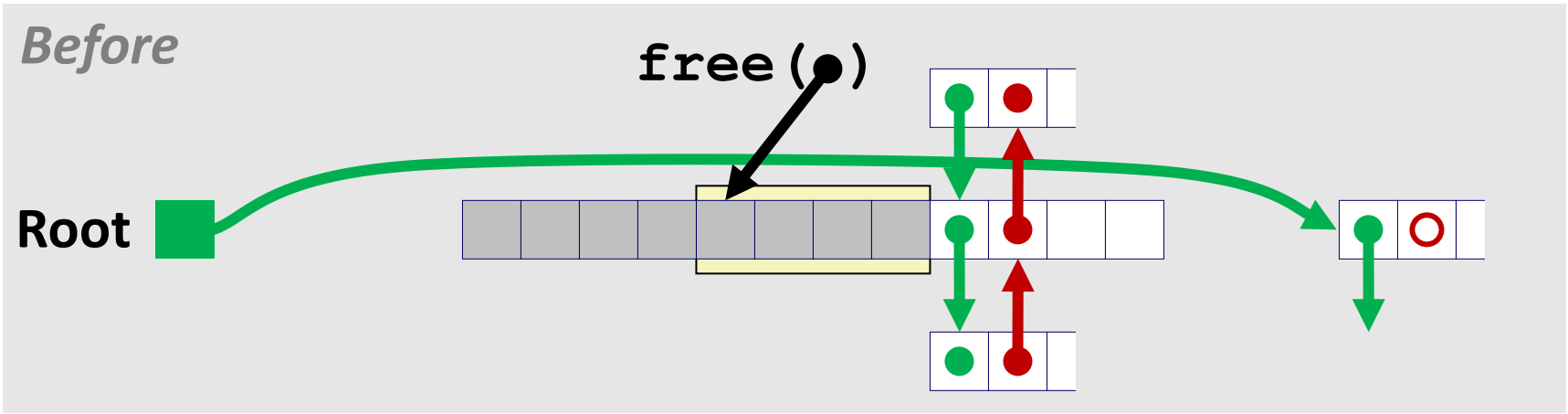

- Insert the freed block at the root of the list



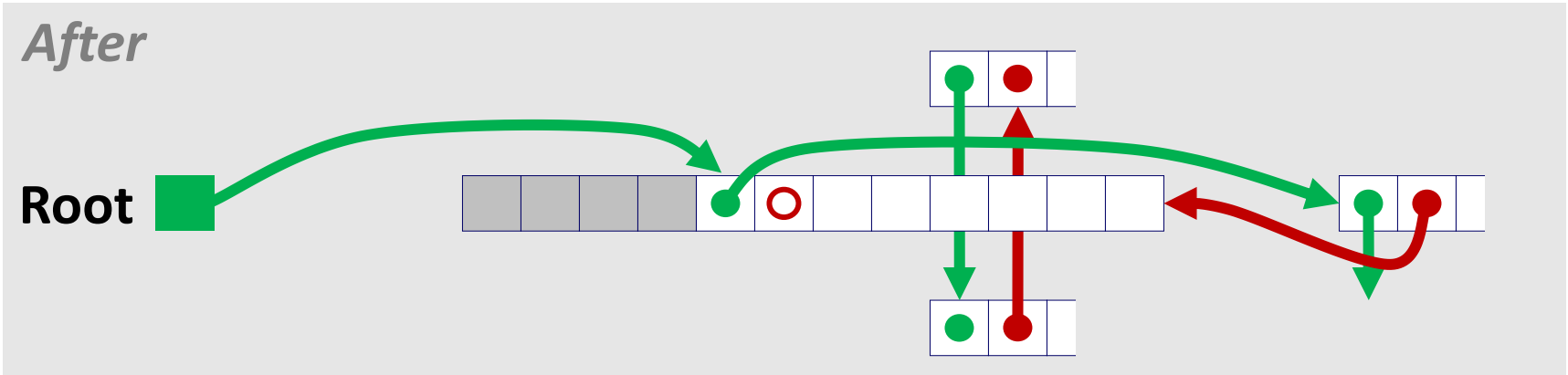
# Freeing With a LIFO Policy (Case 2)



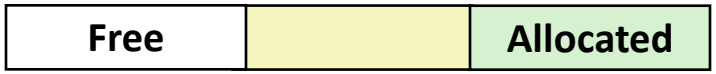

conceptual graphic



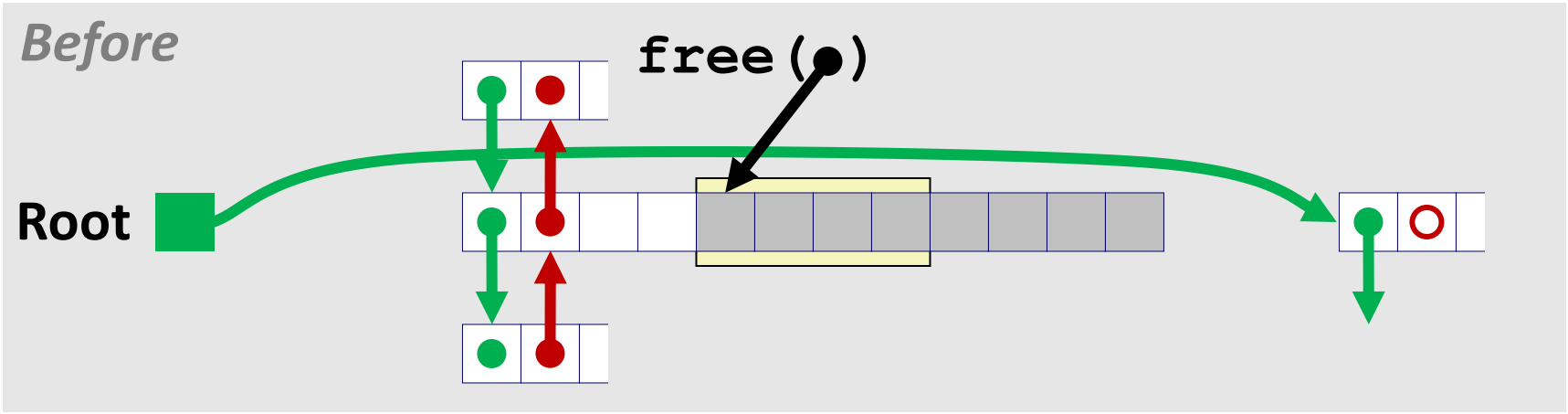

- Splice out adjacent successor block, coalesce both memory blocks, and insert the new block at the root of the list



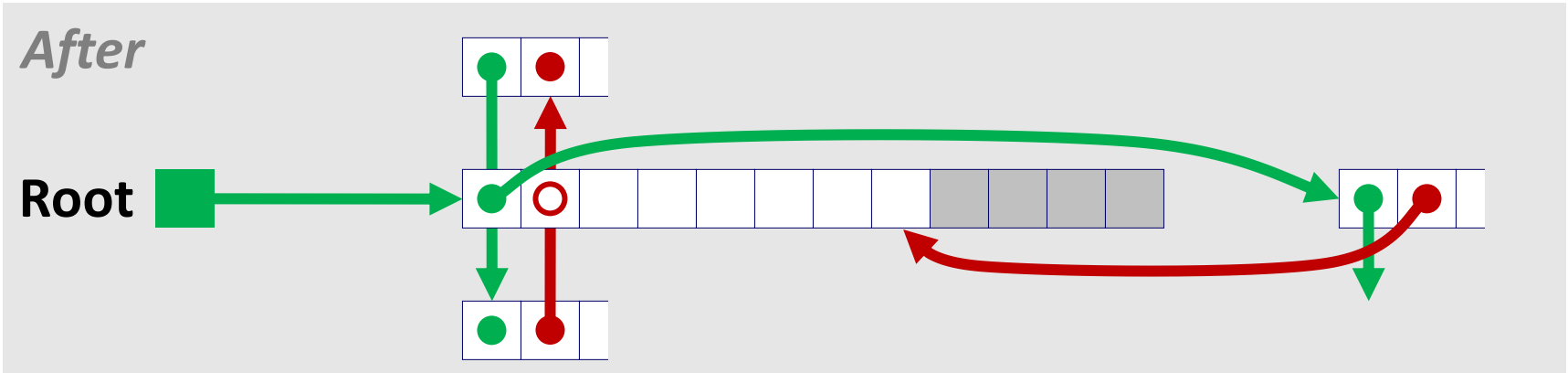
# Freeing With a LIFO Policy (Case 3)



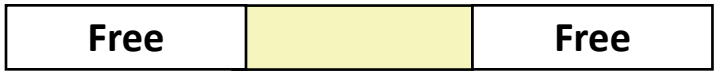

conceptual graphic



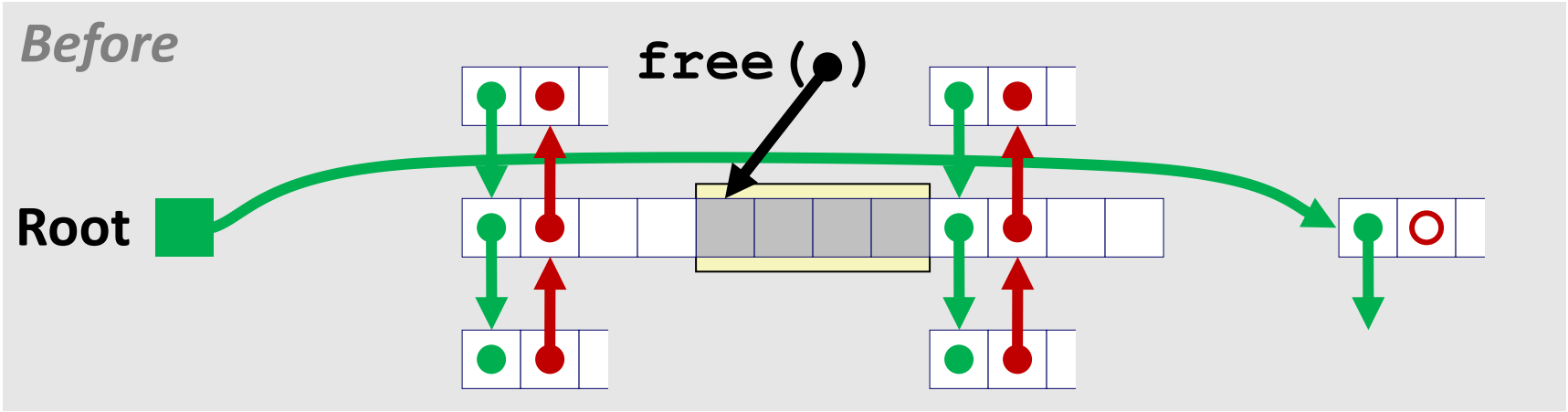

- Splice out adjacent predecessor block, coalesce both memory blocks, and insert the new block at the root of the list



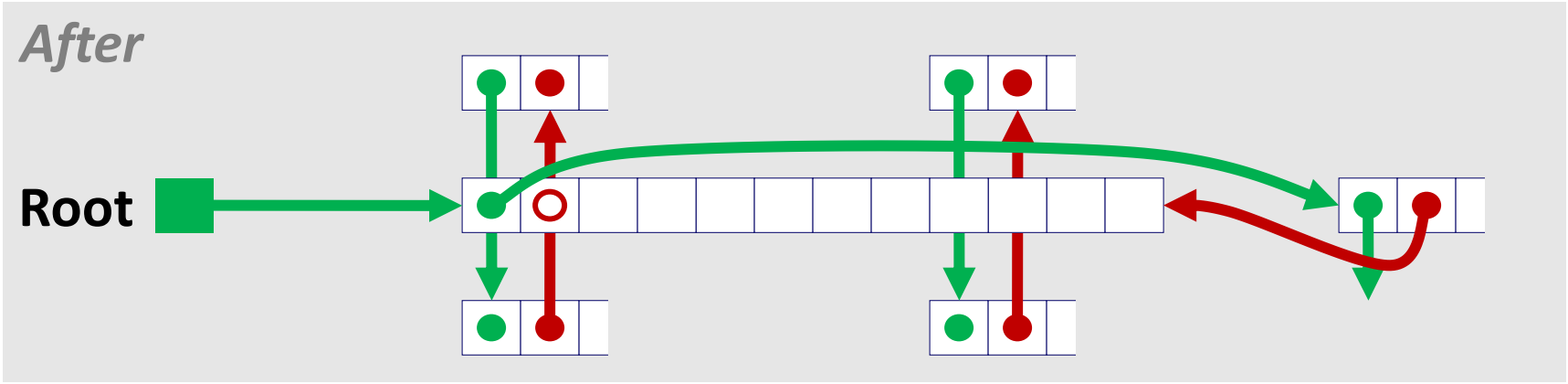
# Freeing With a LIFO Policy (Case 4)



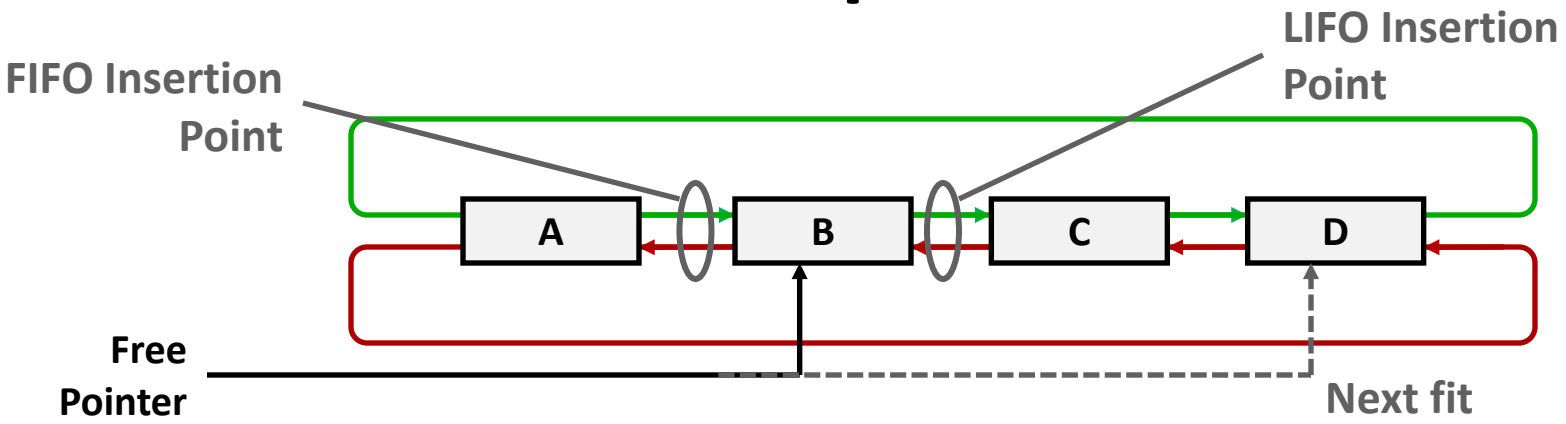

conceptual graphic




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



# Some Advice: An Implementation Trick




- **Use circular, doubly-linked list**
- **Support multiple approaches with single data structure**
- **First-fit vs. next-fit**
  - Either keep free pointer fixed or move as search list
- **LIFO vs. FIFO**
  - Insert as next block (LIFO), or previous block (FIFO)

# Explicit List Summary

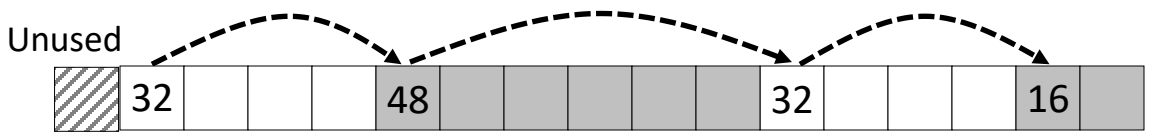
- **Comparison to implicit list:**
  - Allocate is linear time in number of ***free*** blocks instead of ***all*** blocks
    - ***Much faster*** when most of the memory is full
  - Slightly more complicated allocate and free because need to splice blocks in and out of the list
  - Some extra space for the links (2 extra words needed for each block)
    - Does this increase internal fragmentation?

# Today

- Explicit free lists
- **Segregated free lists**
- Memory-related perils and pitfalls

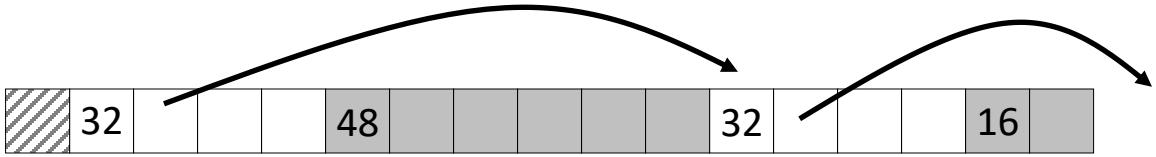
# Keeping Track of Free Blocks

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




Need to tag each block as allocated/free

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



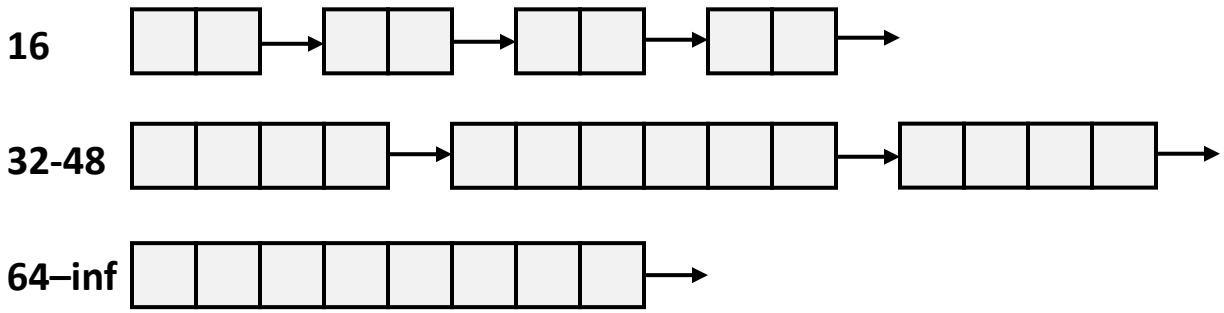

Need space for pointers

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

- **Method 4: *Blocks sorted by size***
  - Can use a balanced 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




- Often have separate classes for each small size
- For larger sizes: One class for each size $[2^i + 1, 2^{i+1}]$

# 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* (i.e., first fit)
  - If an appropriate block is found:
    - Split block and place fragment on appropriate list
    - 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()`**)
  - Allocate block of *n* bytes from this new memory
  - Place remainder as a single free block in appropriate size class.

# Seglist Allocator (cont.)

- **To free a block:**
  - Coalesce and place on appropriate list

- **Advantages of seglist allocators vs. non-seglist allocators (both with first-fit)**
  - Higher throughput
    - log time for power-of-two size classes vs. linear time
  - 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.

# More Info on Allocators

- **D. Knuth, *The Art of Computer Programming,* vol 1, 3rd edition, Addison Wesley, 1997**
  - 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)

# Quiz Time!

Canvas Quiz:  Day 15 – Malloc Advanced

# Today

- Explicit free lists
- Segregated free lists
- **Memory-related perils and pitfalls**

# Memory-Related Perils and Pitfalls

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks

# Dereferencing Bad Pointers

- The classic `scanf` bug

```
int val;

...

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

# Reading Uninitialized Memory

- Assuming that heap data is initialized to zero

```
/* return y = Ax */
int *matvec(int **A, int *x) {
   int *y = 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;
}
```

- Can avoid by using `calloc`

# Overwriting Memory

- Allocating the (possibly) wrong sized object

```
int **p;

p = malloc(N*sizeof(int));

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

- Can you spot the bug?

# Overwriting Memory

- Off-by-one errors

```
char **p;

p = malloc(N*sizeof(int *));

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

```
char *p;

p = malloc(strlen(s));
strcpy(p,s);
```

# Overwriting Memory

- Not checking the max string size

```
char s[8];
int i;

gets(s);  /* reads "123456789" from stdin */
```

- Basis for classic buffer overflow attacks

# Overwriting Memory

- Misunderstanding pointer arithmetic

```
int *search(int *p, int val) {

   while (p && *p != val)
      p += sizeof(int);

   return p;
}
```

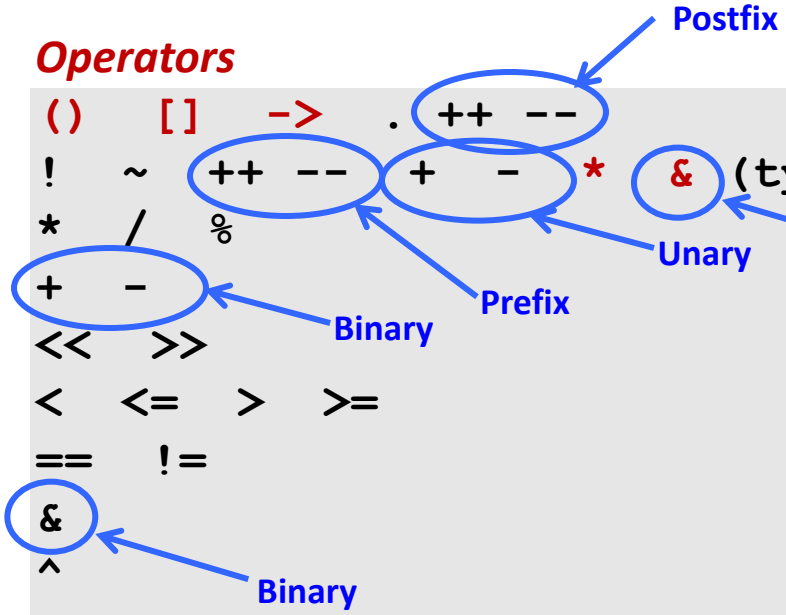
# Overwriting Memory

- Referencing a pointer instead of the object it points to

```
int *BinheapDelete(int **binheap, int *size) {
    int *packet;
    packet = binheap[0];
    binheap[0] = binheap[*size - 1];
    *size--;
    Heapify(binheap, *size, 0);
    return(packet);
}
```

- What gets decremented?
  - (See next slide)

# C operators




| *Operators* | *Associativity* |
|---|---|
| `() [] -> . ++ --` | left to right |
| `! ~ ++ -- + - * & (type) sizeof` | right to left |
| `* / %` | left to right |
| `+ -` | left to right |
| `<< >>` | left to right |
| `< <= > >=` | left to right |
| `== !=` | left to right |
| `&` | left to right |
| `^` | left to right |
| `\|` | left to right |
| `&&` | left to right |
| `\|\|` | left to right |
| `?:` | right to left |
| `= += -= *= /= %= &= ^= != <<= >>=` | right to left |
| `,` | left to right |

- `->`, `()`, and `[]` have high precedence, with `*` and `&` just below
- Unary `+`, `-`, and `*` have higher precedence than binary forms

Source: K&R page 53, updated

# Overwriting Memory

- Referencing a pointer instead of the object it points to

```
int *BinheapDelete(int **binheap, int *size) {
    int *packet;
    packet = binheap[0];
    binheap[0] = binheap[*size - 1];
    *size--;
    Heapify(binheap, *size, 0);
    return(packet);
}
```

- Same effect as
  - `size--;`
- Rewrite as
  - `(*size)--;`

| Operators | Associativity |
|---|---|
| () [] -> . ++ -- | left to right |
| ! ~ ++ -- + - * & (type) sizeof | right to left |
| * / % | left to right |
| + - | left to right |
| << >> | left to right |
| < <= > >= | left to right |
| == != | left to right |
| & | left to right |
| ^ | left to right |
| \| | left to right |
| && | left to right |
| \|\| | left to right |
| ?: | right to left |
| = += -= *= /= %= &= ^= != <<= >>= | right to left |
| , | left to right |

# Referencing Nonexistent Variables

- Forgetting that local variables disappear when a function returns

```
int *foo () {
    int val;

    return &val;
}
```

# Freeing Blocks Multiple Times

- Nasty!

```
x = malloc(N*sizeof(int));
        <manipulate x>
free(x);

y = malloc(M*sizeof(int));
        <manipulate y>
free(x);
```

# Referencing Freed Blocks

- Evil!

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

# Failing to Free Blocks (Memory Leaks)

- Slow, long-term killer!

```
foo() {
    int *x = malloc(N*sizeof(int));
    ...
    return;
}
```

# Failing to Free Blocks (Memory Leaks)

- Freeing only part of a data structure

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

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

# Dealing With Memory Bugs

- **Debugger: `gdb`**
  - Good for finding bad pointer dereferences
  - Hard to detect the other memory bugs
- **Data structure consistency checker**
  - Runs silently, prints message only on error
  - Use as a probe to zero in on error
- **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
- **glibc malloc contains checking code**
  - `setenv MALLOC_CHECK_ 3`

# Supplemental slides

# Implicit Memory Management: Garbage Collection

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

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

- **Common in many dynamic languages:**
  - Python, Ruby, Java, 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

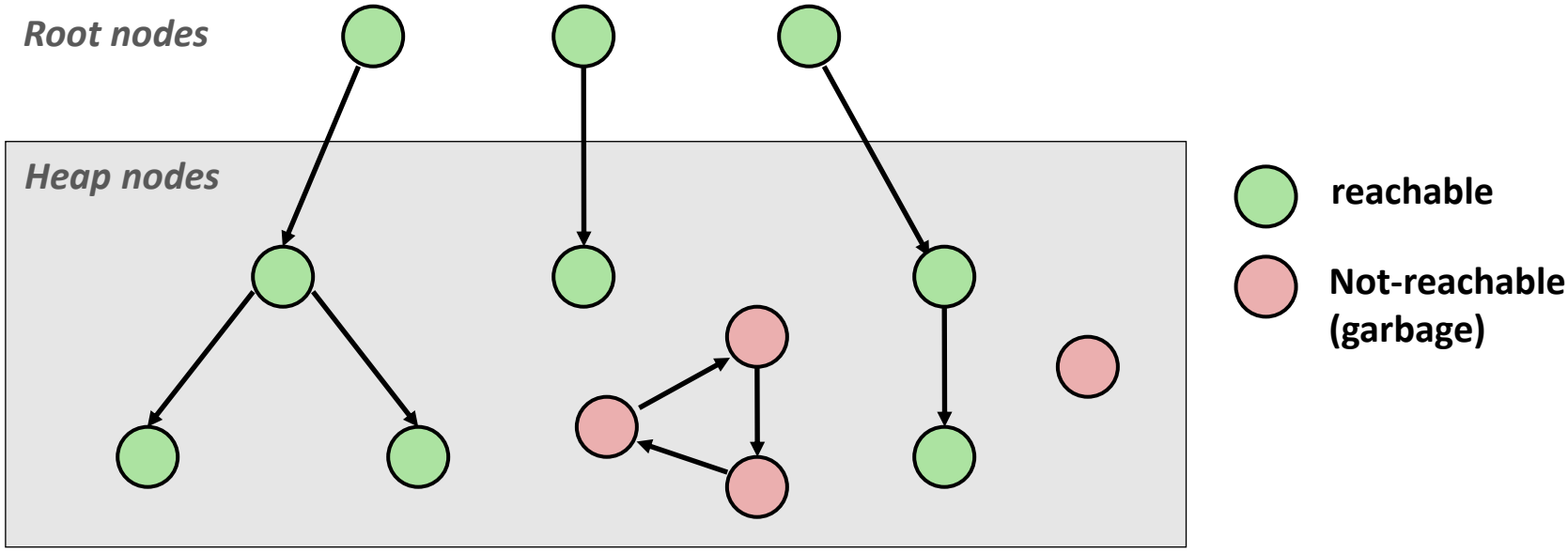
- **Must make certain assumptions about pointers**
  - Memory manager can distinguish pointers from non-pointers
  - All pointers point to the start of a block
  - Cannot hide pointers
    (e.g., by coercing them to an `int`, and then back again)

# 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 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)

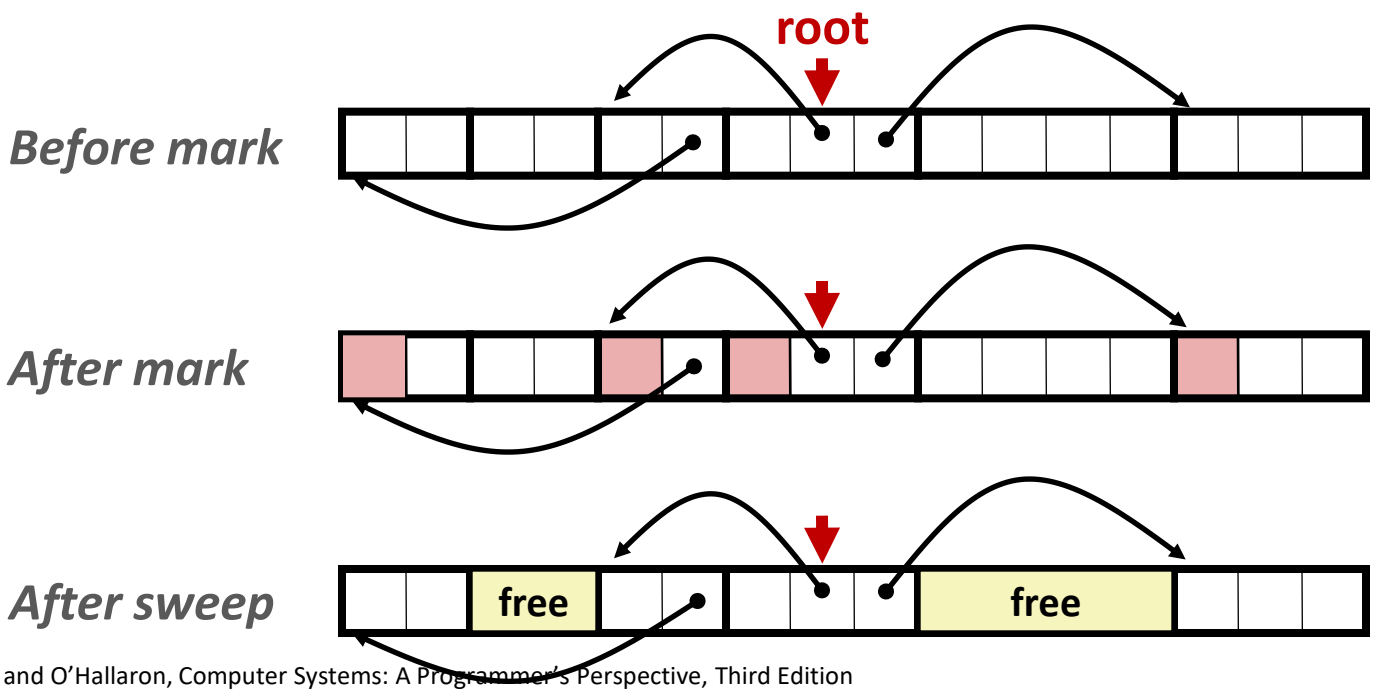



**A node (block) is *reachable* if there is a path from any root to that node.**

**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




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

# Assumptions For a Simple Implementation

- **Application**
  - **`new(n)`:** returns pointer to new block with all locations cleared
  - **`read(b,i)`:** read location **`i`** of block **`b`** into register
  - **`write(b,i,v)`:** write **`v`** into location **`i`** of block **`b`**

- **Each block will have a header word**
  - addressed as **`b[-1]`**, for a block **`b`**
  - Used for different purposes in different collectors

- **Instructions used by the Garbage Collector**
  - **`is_ptr(p)`:** determines whether **`p`** is a pointer
  - **`length(b)`:** returns the length of block **`b`**, not including the header
  - **`get_roots()`:** returns all the roots

# Mark and Sweep Pseudocode

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

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;
   if (markBitSet(p)) return;
   setMarkBit(p);
   for (i=0; i < length(p); i++)
     mark(p[i]);
   return;
}
```

# Mark and Sweep Pseudocode

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

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // if not pointer -> do nothing
   if (markBitSet(p)) return;
   setMarkBit(p);
   for (i=0; i < length(p); i++)
     mark(p[i]);
   return;
}
```

# Mark and Sweep Pseudocode

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

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // if not pointer -> do nothing
   if (markBitSet(p)) return;       // if already marked -> do nothing
   setMarkBit(p);
   for (i=0; i < length(p); i++)
     mark(p[i]);
   return;
}
```

# Mark and Sweep Pseudocode

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

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // if not pointer -> do nothing
   if (markBitSet(p)) return;       // if already marked -> do nothing
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)
     mark(p[i]);
   return;
}
```

# Mark and Sweep Pseudocode

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

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // if not pointer -> do nothing
   if (markBitSet(p)) return;       // if already marked -> do nothing
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // for each word in p's block
     mark(p[i]);
   return;
}
```

# Mark and Sweep Pseudocode

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

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // if not pointer -> do nothing
   if (markBitSet(p)) return;       // if already marked -> do nothing
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // for each word in p's block
     mark(p[i]);                    //  make recursive call
   return;
}
```

# Mark and Sweep Pseudocode

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

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // if not pointer -> do nothing
   if (markBitSet(p)) return;       // if already marked -> do nothing
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // for each word in p's block
     mark(p[i]);                    //  make recursive call
   return;
}
```

## Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {                // for entire heap
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p+1);
}
```

# Mark and Sweep Pseudocode

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

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // if not pointer -> do nothing
   if (markBitSet(p)) return;       // if already marked -> do nothing
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // for each word in p's block
     mark(p[i]);                    //  make recursive call
   return;
}
```

## Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {                // for entire heap
      if markBitSet(p)              // did we reach this block?
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p+1);
}
```

# Mark and Sweep Pseudocode

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

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // if not pointer -> do nothing
   if (markBitSet(p)) return;       // if already marked -> do nothing
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // for each word in p's block
     mark(p[i]);                    //  make recursive call
   return;
}
```

## Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {                // for entire heap
      if markBitSet(p)              // did we reach this block?
         clearMarkBit();            //  yes -> so just clear mark bit
      else if (allocateBitSet(p))
         free(p);
      p += length(p+1);
}
```

# Mark and Sweep Pseudocode

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

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // if not pointer -> do nothing
   if (markBitSet(p)) return;       // if already marked -> do nothing
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // for each word in p's block
     mark(p[i]);                    //  make recursive call
   return;
}
```

## Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {                // for entire heap
      if markBitSet(p)              // did we reach this block?
         clearMarkBit();            //  yes -> so just clear mark bit
      else if (allocateBitSet(p)) // never reached: is it allocated?
         free(p);
      p += length(p+1);
}
```

# Mark and Sweep Pseudocode

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

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // if not pointer -> do nothing
   if (markBitSet(p)) return;       // if already marked -> do nothing
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // for each word in p's block
     mark(p[i]);                    //  make recursive call
   return;
}
```

## Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {                // for entire heap
      if markBitSet(p)              // did we reach this block?
         clearMarkBit();            //  yes -> so just clear mark bit
      else if (allocateBitSet(p)) // never reached: is it allocated?
         free(p);                   //  yes -> its garbage, free it
      p += length(p+1);
}
```

# Mark and Sweep Pseudocode

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

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // if not pointer -> do nothing
   if (markBitSet(p)) return;       // if already marked -> do nothing
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // for each word in p's block
     mark(p[i]);                    //  make recursive call
   return;
}
```

## Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {                // for entire heap
      if markBitSet(p)              // did we reach this block?
         clearMarkBit();            //  yes -> so just clear mark bit
      else if (allocateBitSet(p))   // never reached: is it allocated?
         free(p);                   //  yes -> its garbage, free it
      p += length(p+1);             // goto next block
}
```

# C Pointer Declarations: Test Yourself!

| | |
|---|---|
| `int *p` | p is a pointer to int |
| `int *p[13]` | p is an array[13] of pointer to int |
| `int *(p[13])` | p is an array[13] of pointer to int |
| `int **p` | p is a pointer to a pointer to an int |
| `int (*p)[13]` | p is a pointer to an array[13] of int |
| `int *f()` | f is a function returning a pointer to int |
| `int (*f)()` | f is a pointer to a function returning int |
| `int (*(*x[3])())[5]` | x is an array[3] of pointers to functions returning pointers to array[5] of ints |

Source: K&R Sec 5.12

# C Pointer Declarations: Test Yourself!

| | |
|---|---|
| `int *p` | p is a pointer to int |
| `int *p[13]` | p is an array[13] of pointer to int |
| `int *(p[13])` | p is an array[13] of pointer to int |
| `int **p` | p is a pointer to a pointer to an int |
| `int (*p)[13]` | p is a pointer to an array[13] of int |
| `int *f()` | f is a function returning a pointer to int |
| `int (*f)()` | f is a pointer to a function returning int |
| `int (*(*x[3])())[5]` | x is an array[3] of pointers to functions returning pointers to array[5] of ints |
| `int (*(*f())[13])()` | f is a function returning ptr to an array[13] of pointers to functions returning int |

Source: K&R Sec 5.12

# Parsing: int (*(*f())[13])()

| | |
|---|---|
| `int (*(*f())[13])()` | f |
| `int (*(*f())[13])()` | f is a function |
| `int (*(*f())[13])()` | f is a function that returns a ptr |
| `int (*(*f())[13])()` | f is a function that returns a ptr to an array of 13 |
| `int (*(*f())[13])()` | f is a function that returns a ptr to an array of 13 ptrs |
| `int (*(*f())[13])()` | f is a function that returns a ptr to an array of 13 ptrs to functions returning an int |