Roadmap

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 8 Mac

Computer system:

Memory Allocation Topics

- **Dynamic memory allocation**
  - Size/number/lifetime 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
- **Explicit Allocation Implementation**
  - Implicit free lists
  - Explicit free lists – subject of next programming assignment
  - Segregated free lists
- **Implicit Deallocation: Garbage collection**
- **Common memory-related bugs in C programs**

ALL THE DATA!!!

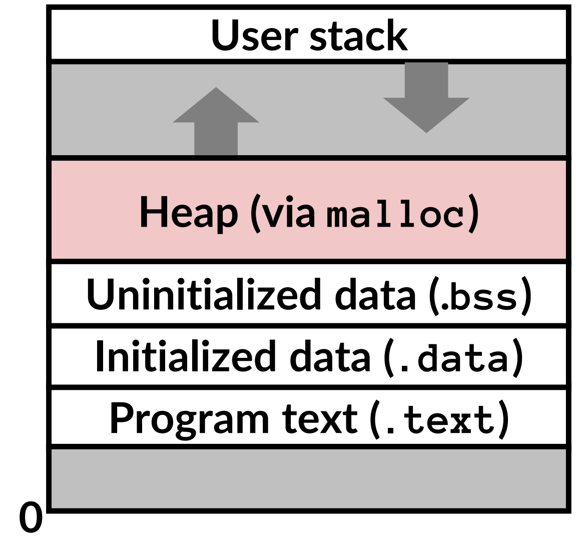
Multiple ways so far of storing data:

```
int array[1024];

void foo(int n) {
  int tmp;
  int local_array[n];

  int* dyn =
    (int*)malloc(n * sizeof(int));

}
```

- **Static global data**
  - ***Fixed size*** at compile-time
  - Entire ***lifetime of the program*** (loaded from executable)
  - Portion is read-only (e.g. string literals)
- **Stack-allocated data**
  - Local / temporary variables, *can* be dynamically sized (in some versions of C)
  - ***Known lifetime*** (deallocated on **`return`**)
- **Dynamic (heap) data**
  - Size known only at runtime (based on user-input, etc)
  - Lifetime known only at runtime (long-lived data structures)

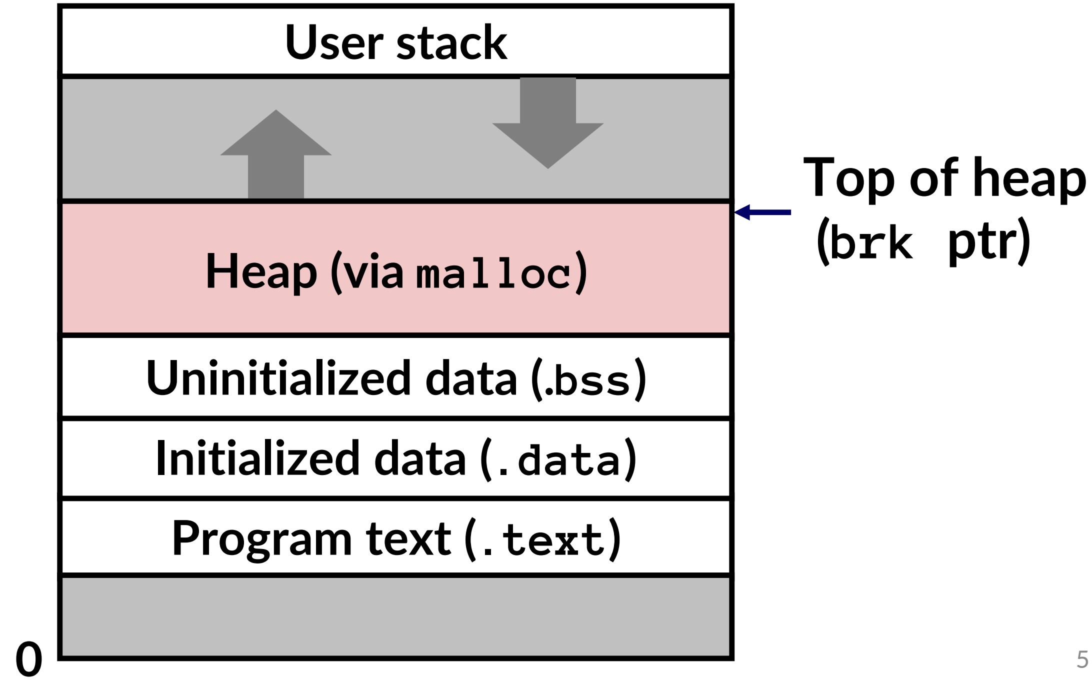
Dynamic Memory Allocation

- **Programmers use *dynamic memory allocators* (such as `malloc`) to acquire virtual memory at run time**
  - For data structures whose size (or lifetime) is known only at runtime
- **Types of allocators**
  - *Explicit allocator*: application allocates and frees space
    - E.g. **`malloc`** and **`free`** in C
  - *Implicit allocator:* application allocates, but does not free space
    - E.g. garbage collection in Java, ML, and Lisp
- **Dynamic memory allocators manage an area of a process' virtual memory known as the *heap***



Dynamic Memory Allocation

- **Allocator organizes the heap as a collection of variable-sized *blocks*, which are either *allocated* or *free***
  - Allocator requests pages in heap region; virtual memory hardware and OS kernel allocate these pages to the process
  - Application objects are typically smaller than pages, so the allocator manages blocks *within* pages
    - (Larger objects handled too; ignored here)



The `malloc` Package

`#include <stdlib.h>`

`void* malloc(size_t size)`

- Successful:
  - Returns a pointer to a memory block of at least **`size`** bytes (typically) aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
  - If **`size == 0`**, returns NULL
- Unsuccessful: returns NULL and sets **`errno`**

`void free(void* p)`

- Gives back the block pointed at by **`p`** to pool of available memory
- **`p`** must come from a previous call to **`malloc`** (or similar, see below)

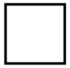
Other functions

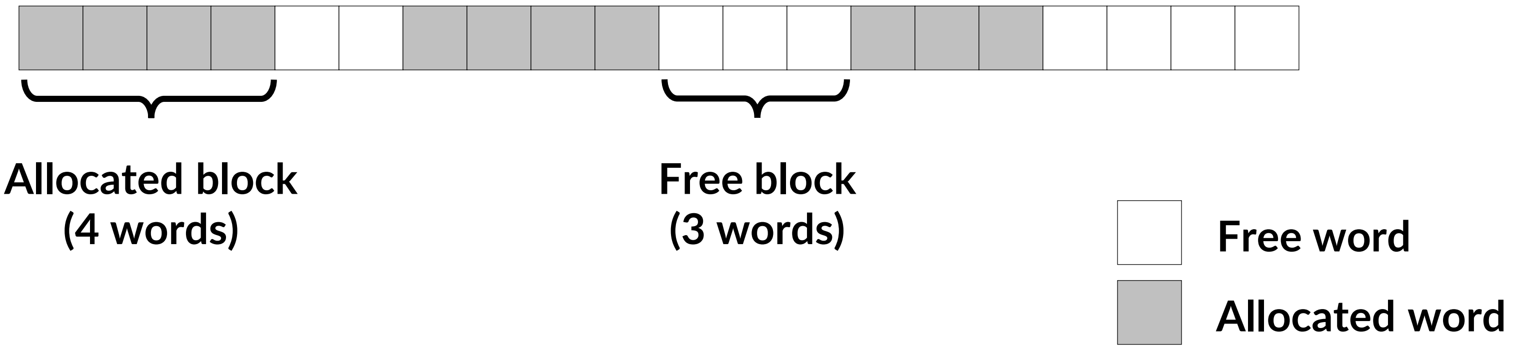
- **`calloc`**: Version of **`malloc`** that "zeros out" allocated block
- **`realloc`**: Changes the size of a previously allocated block (if possible)
  (warning: realloc works differently on BSD/OSX than in our version of Linux for this course)
- **`sbrk`**: Used internally by allocators to grow or shrink the heap
  - historical naming from before virtual memory was common

Malloc Example

```
void foo(int n, int m) {
  int i, *p;

  /* allocate a block of n ints */
  p = (int *)malloc(n * sizeof(int));
  if (p == NULL) {
    perror("malloc");
    exit(0);
  }
  for (i=0; i<n; i++)
    p[i] = i;

  /* add space for m ints to end of p block */
  p = (int *)realloc(p, (n+m) * sizeof(int));
  if (p == NULL) {
    perror("realloc");
    exit(0);
  }
  for (i=n; i < n+m; i++)
    p[i] = i;

  /* print new array */
  for (i=0; i<n+m; i++)
    printf("%d\n", p[i]);

  free(p); /* return p to available memory pool */
}
```

Assumptions made in: these slides, book, videos

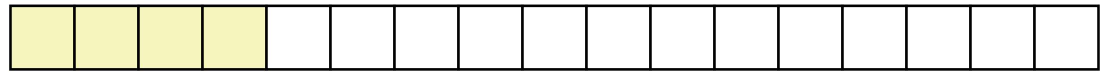
- **Memory is drawn divided into *words***
  - Each *word* can hold an `int` (32 bits/4 bytes)
  - Allocations will be in sizes that are a multiple of words, i.e. multiples of 4 bytes
  - In pictures in slides, book, videos :  = one word, 4 bytes



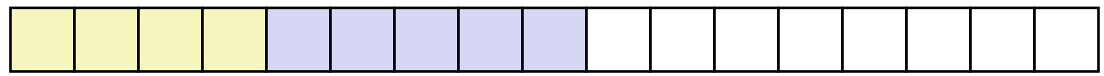
Allocation Example

= 4 byte word

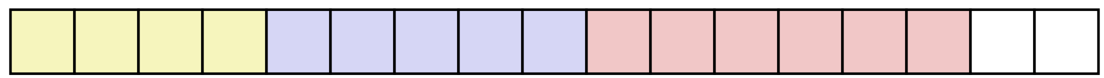
`p1 = malloc(16)`



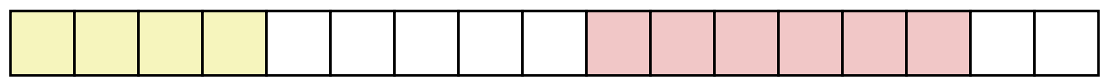
`p2 = malloc(20)`



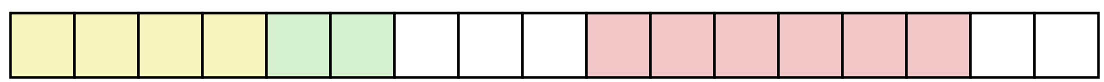
`p3 = malloc(24)`



`free(p2)`



`p4 = malloc(8)`



Constraints (interface/contract)

- **Applications**
  - Can issue arbitrary sequence of `malloc` and `free` requests
  - Must never access memory not currently allocated (else "who knows")
  - Must never free memory not currently allocated (else "who knows")
    - Also must only use **`free`** with previously malloc'ed (or calloc'ed etc.) blocks (not, e.g., stack data) (else "who knows")
- **Allocators**
  - Can't control number or size of allocated blocks
  - Must respond immediately to `malloc` requests
    - *i.e.*, can't reorder or buffer requests
  - Must allocate blocks from free memory
    - *i.e.*, blocks can't overlap, *why not?*
  - Must align blocks so they satisfy all alignment requirements
  - Can't move the allocated blocks
    - *i.e.*, compaction is not allowed. *Why not?*

Performance Goal #1: Throughput

- **Given some sequence of `malloc` and `free` requests:**
  - $R_0, R_1, ..., R_k, ... , R_{n-1}$
- **Goals: maximize throughput and peak memory utilization**
  - These goals are often conflicting
- **Throughput:**
  - Number of completed requests per unit time
  - Example:
    - 5,000 `malloc` calls and 5,000 `free` calls in 10 seconds
    - Throughput is 1,000 operations/second

Performance Goal #2: Peak Memory Utilization

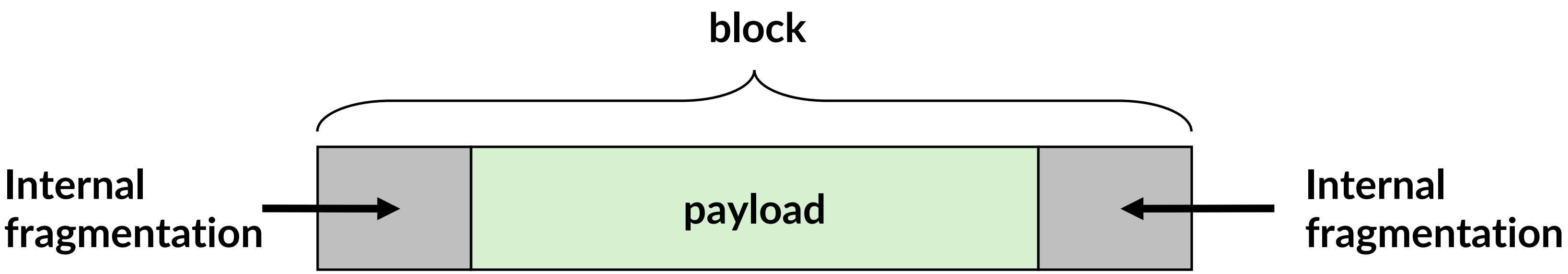
- **Given some sequence of `malloc` and `free` requests:**
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$
- ***Def: Aggregate payload P_k***
  - `malloc(p)` results in a block with a ***payload*** of `p` bytes
  - After request R_k has completed, the ***aggregate payload*** P_k is the sum of currently allocated payloads
- ***Def: Current heap size = H_k***
  - Assume H_k is monotonically nondecreasing
    - Allocator can increase size of heap using `sbrk`
- ***Def: Peak memory utilization after k + 1 requests***
  - $U_k = (\max_{i \le k} P_i) \ / \ H_k$
  - Goal: maximize utilization for a sequence of requests.
  - *Why is this hard? And what happens to throughput?*

Fragmentation

- Poor memory utilization is caused by *fragmentation*
- Sections of memory are not used to store anything useful, but cannot satisfy allocation requests
- *internal* fragmentation
- *external* fragmentation

Internal Fragmentation

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



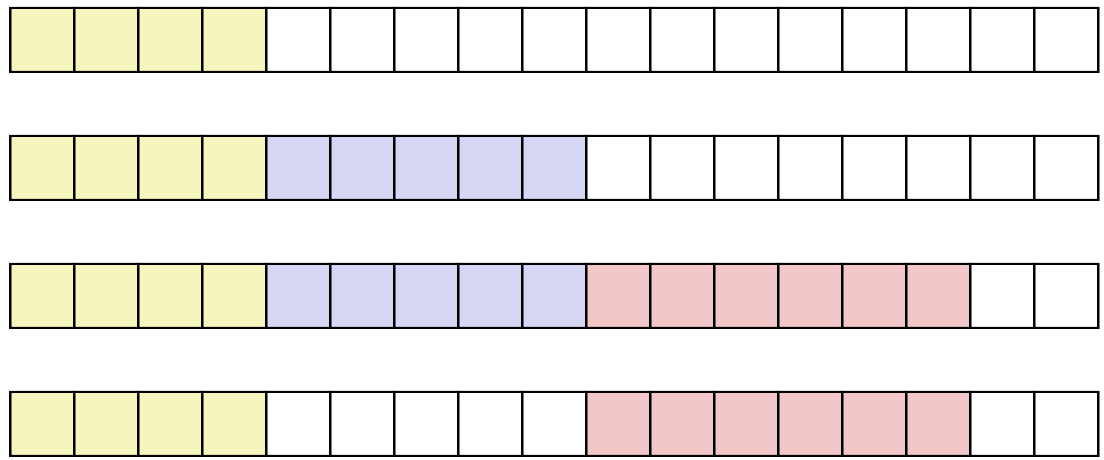
- Caused by
  - overhead of maintaining heap data structures (inside block, outside payload)
  - padding for alignment purposes
  - explicit policy decisions (e.g., to return a big block to satisfy a small request)
    why would anyone do that?

External Fragmentation

= 4 byte word

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

```
p1 = malloc(16)
p2 = malloc(20)
p3 = malloc(24)
free(p2)
p4 = malloc(24)
```



Bummer! (what would happen now?)

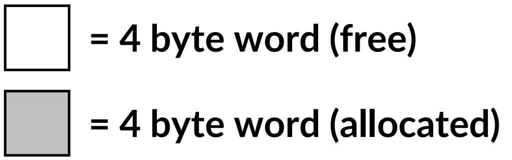
- **Don't know what requests will come in the future…**
  - Thus, difficult (er, impossible) to know where to best place things

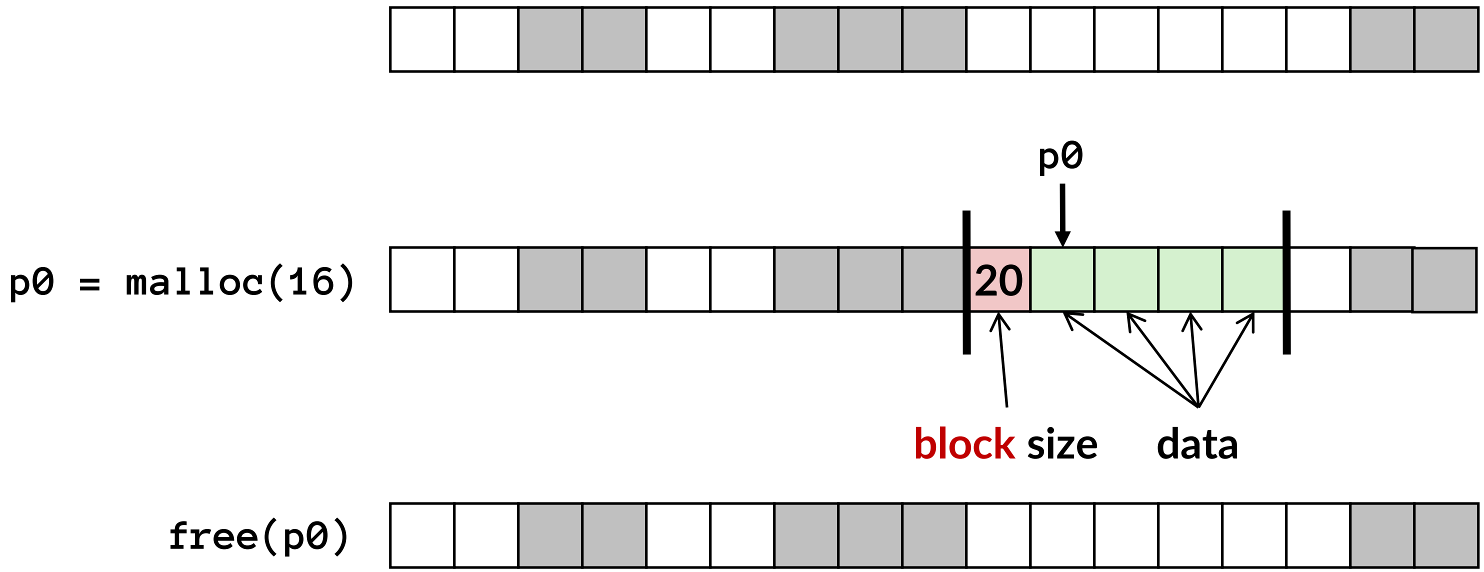
Implementation Issues

- How do we know how much memory to free given just a pointer?
- How do we keep track of the free blocks?
- How do we pick a block to use for allocation (when many might fit)?
- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
- How do we reinsert a freed block into the heap?

Knowing How Much to Free

- **Standard method**
  - Keep the length of a block in the word preceding the block
    - This word is often called the ***header field*** or ***header***
  - Requires an extra word for every allocated block



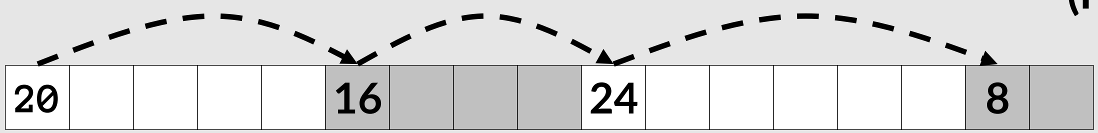


Keeping Track of Free Blocks

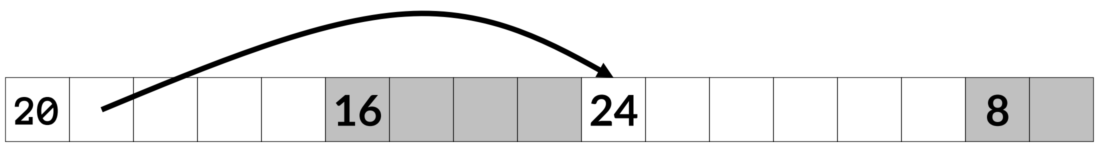
= 4 byte word (free)

= 4 byte word (allocated)

- Method 1: ***Implicit free list*** **using length— links <u>all</u> blocks using math (no actual pointers)**



- **Method 2: *Explicit free list* among <u>only the free blocks</u>, using pointers**



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

- **Method 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

Implicit Free Lists

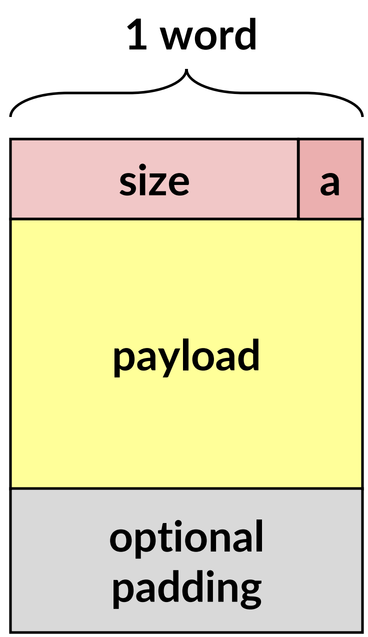
- **For each block we need two pieces of info:**
  - size? is-allocated?
  - Could store this information in two words: wasteful!
- **Standard trick**
  - If blocks are aligned some low-order bits are always 0
  - Instead of storing a bit that will always be 0, use it as a allocated/free flag
  - When reading size, must remember to mask out this bit

e.g. with 8-byte alignment, possible values for size:
00001000 = 8 bytes
00010000 = 16 bytes
00011000 = 24 bytes
...

```
p = malloc(_)
```

```
x = size | allocated
```

```
allocated = x & 1
size = x & 0b11…10
```



Format of allocated and free blocks

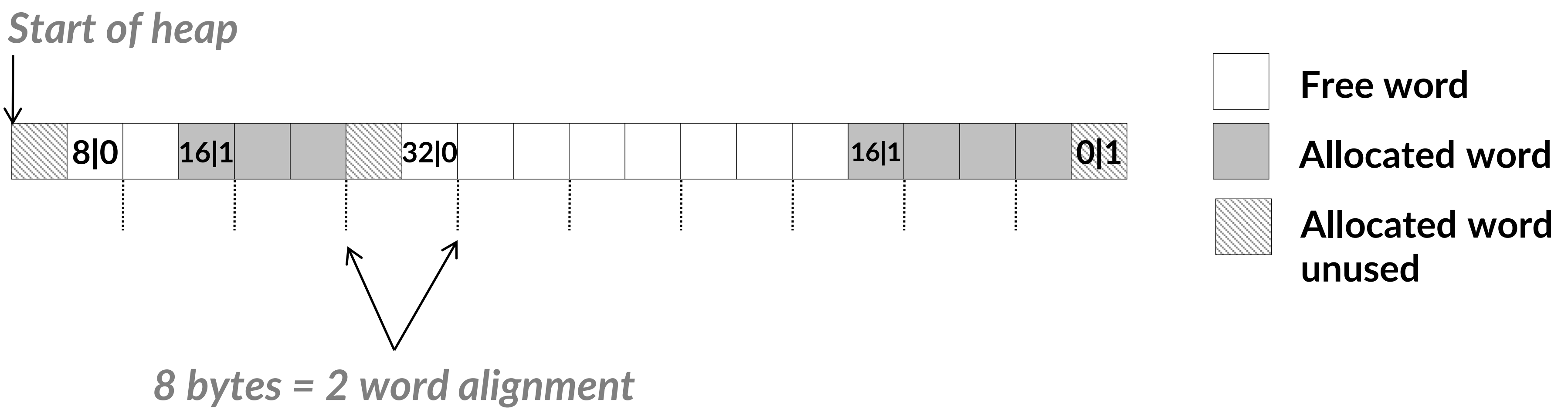
a = 1: allocated block
a = 0: free block

size: block size

payload: application data (allocated blocks only)

Implicit Free List Example (words = 32-bits)

Each block begins with a header that contains its size in bytes/allocated bit.
Sequence of blocks in heap (size|allocated): 8|0, 16|1, 32|0, 16|1



- **8-byte alignment**
  - May require initial unused word
  - Causes some internal fragmentation
- **Special one-word marker (0|1) marks end of list**
  - zero size is distinguishable from all real sizes

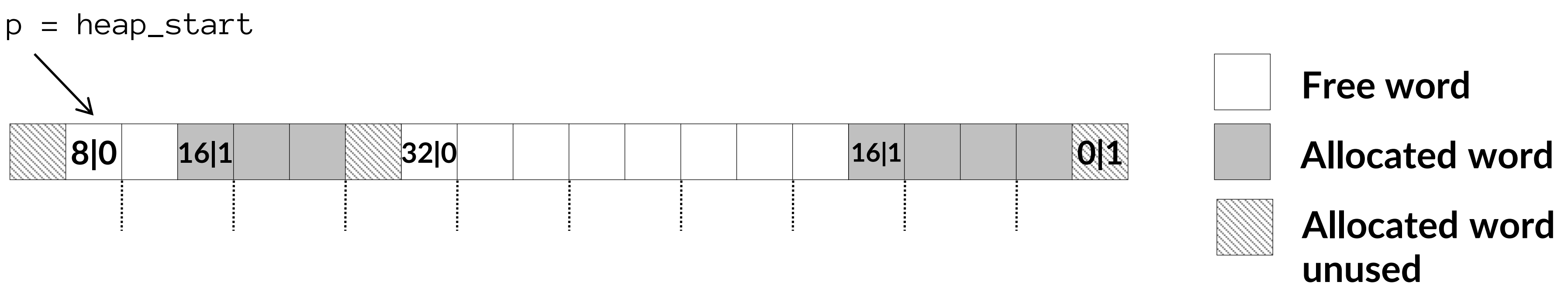
Implicit List: Finding a Free Block

*p gets the block *header*
*p & 1 extracts the allocated bit
*p & -2 masks the allocated bit, gets just the size

- ***First fit:***
  - **Search list from beginning**, choose ***first*** free block that fits:

```
p = heap_start;
while ((p < end) &&       // not past end
       ((*p & 1) ||       // already allocated
        (*p <= len))) {  // too small
  p = p + (*p & -2);      // go to next block (UNSCALED +)
}                         // p points to selected block or end
```

  - Can take time linear in total number of blocks (allocated and free)
  - In practice it can cause “splinters” at beginning of list



Implicit List: Finding a Free Block

*p gets the block *header*
*p & 1 extracts the allocated bit
*p & -2 masks the allocated bit, gets just the size

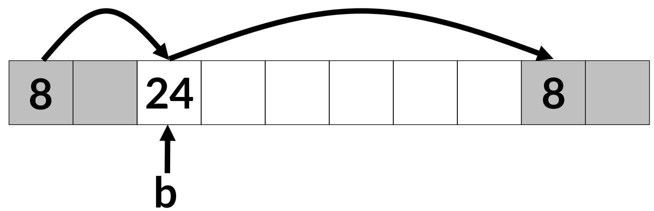
- ***First fit:***
  - **Search list from beginning**, choose ***first*** free block that fits:

```
p = heap_start;
while ((p < end) &&      // not past end
       ((*p & 1) ||      // already allocated
        (*p <= len))) {  // too small
  p = p + (*p & -2);     // go to next block (UNSCALED +)
}                        // p points to selected block or end
```

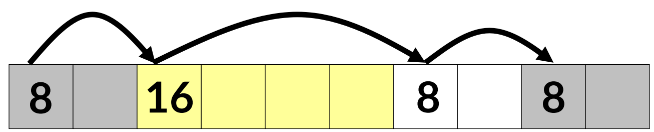
  - Can take time linear in total number of blocks (allocated and free)
  - In practice it can cause "splinters" at beginning of list
- ***Next fit:***
  - Like first-fit, but **search list starting where previous search finished**
  - Should often be faster than first-fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse
- ***Best fit:***
  - Search the list, choose the ***best*** free block: fits, with fewest bytes left over
  - Keeps fragments small—usually helps fragmentation
  - Usually worse throughput

Implicit List: Allocating in a Free Block

- Allocating in a free block: *splitting*
  - Since allocated space might be smaller than free space, we might want to split the block




```
malloc(n = 12):
  ptr b = find(12+4)
  split(b, 12+4)
```



assume ptr points to word and has unscaled pointer arithmetic

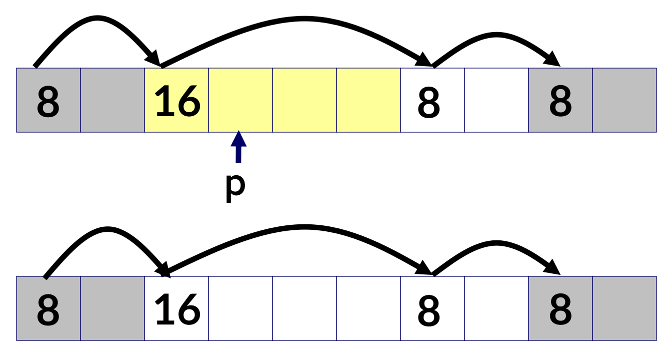
```
void split(ptr b, int bytes) {                  // bytes = desired block size
  int newsize = ((bytes + 7) >> 3) << 3;        // round up to multiple of 8
  int oldsize = *b;                             // why not mask out low bit?
  *b = newsize;                                 // initially unallocated
  if (newsize < oldsize)
    *(b+newsize) = oldsize - newsize;           // set length in remaining
}                                               // part of block (UNSCALED +)
```

Implicit List: Freeing a Block

- **Simplest implementation:**
  - Need only clear the “allocated” flag

    `void free(ptr p) { ptr b = p – WORD; *b = *b & -2 }`
  - But can lead to “false fragmentation”

`free(p)`



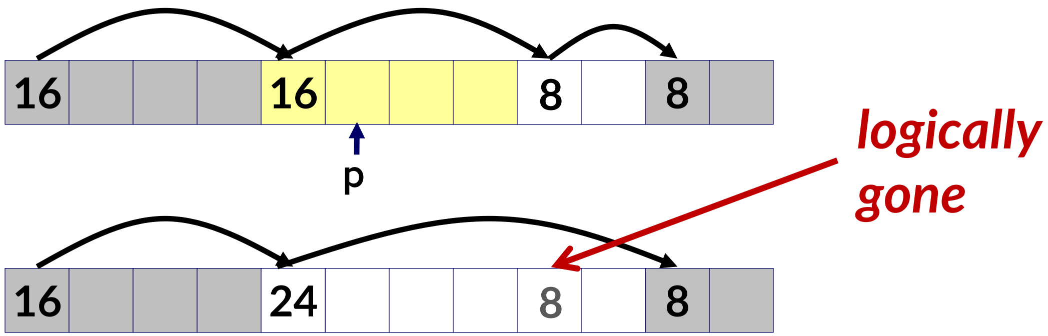
`malloc(20)` ***Oops!***

There is enough free space, but the allocator won’t be able to find it

Implicit List: Coalescing

- **Join *(coalesce)* with next/previous blocks, if they are free**
  - Coalescing with next block

`free(p)`

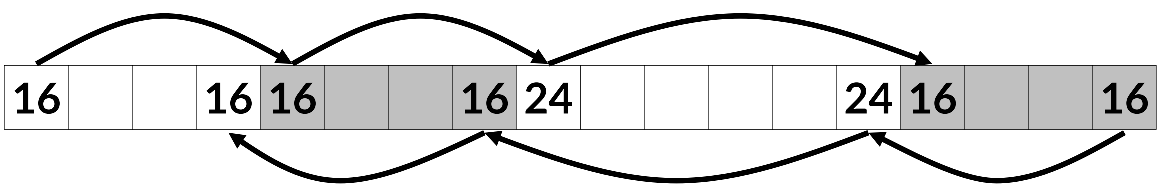



```
void free(ptr p) {           // p points to data
    ptr b = p – WORD;        // b points to block
    *b = *b & -2;            // clear allocated bit
    ptr next = b + *b;       // find next block (UNSCALED +)
    if ((*next & 1) == 0)    // if next block is not allocated,
      *b = *b + *next;       // add its size to this block
}
```

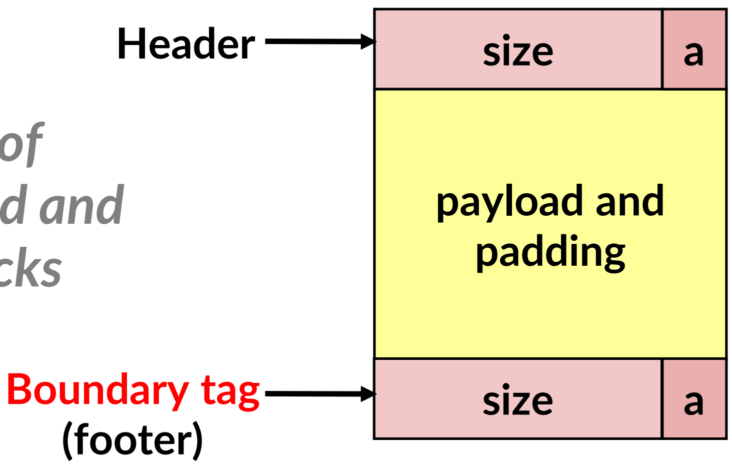
- But how do we coalesce with the *previous* block?

Implicit List: Bidirectional Coalescing

- ***Boundary tags*** [Knuth73]
  - 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

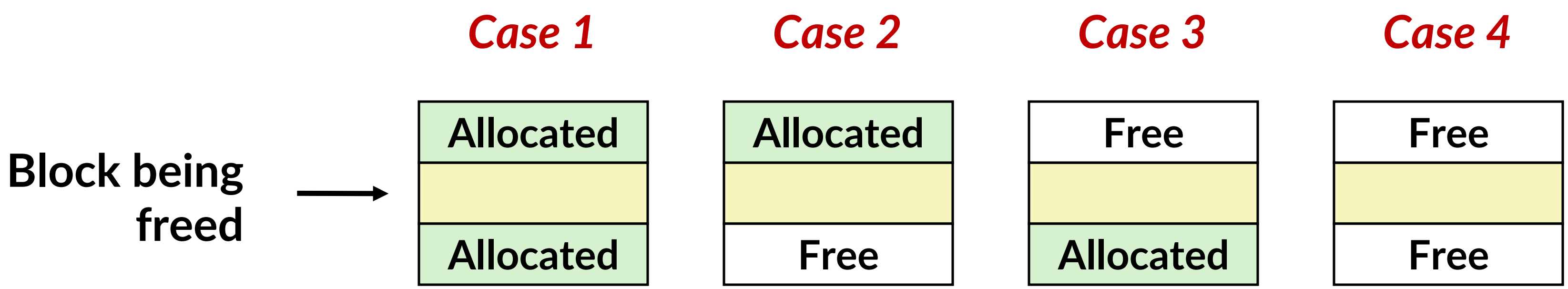


a = 1: allocated block
a = 0: free block

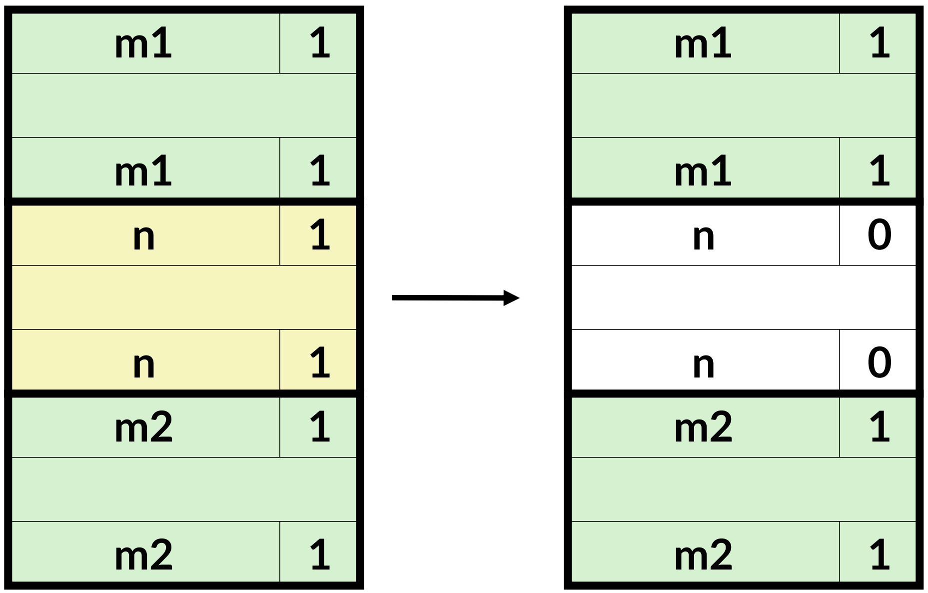
size: total block size

payload: application data (allocated blocks only)

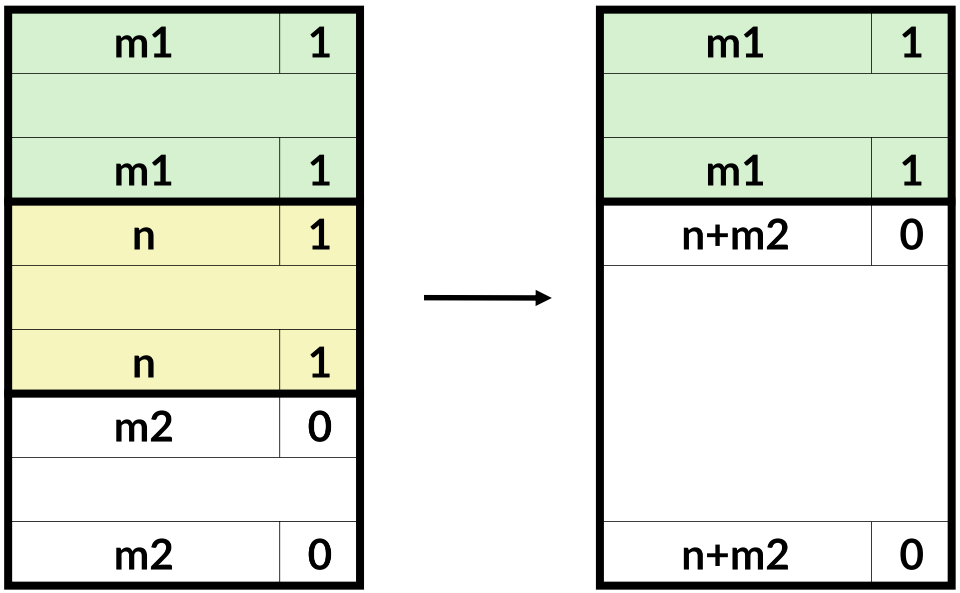
Constant Time Coalescing



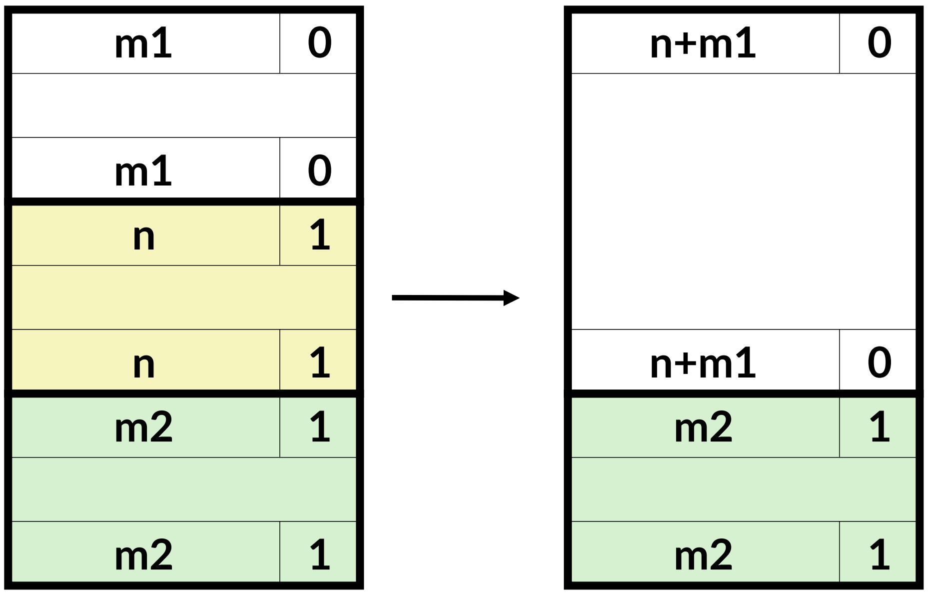
Constant Time Coalescing (Case 1)



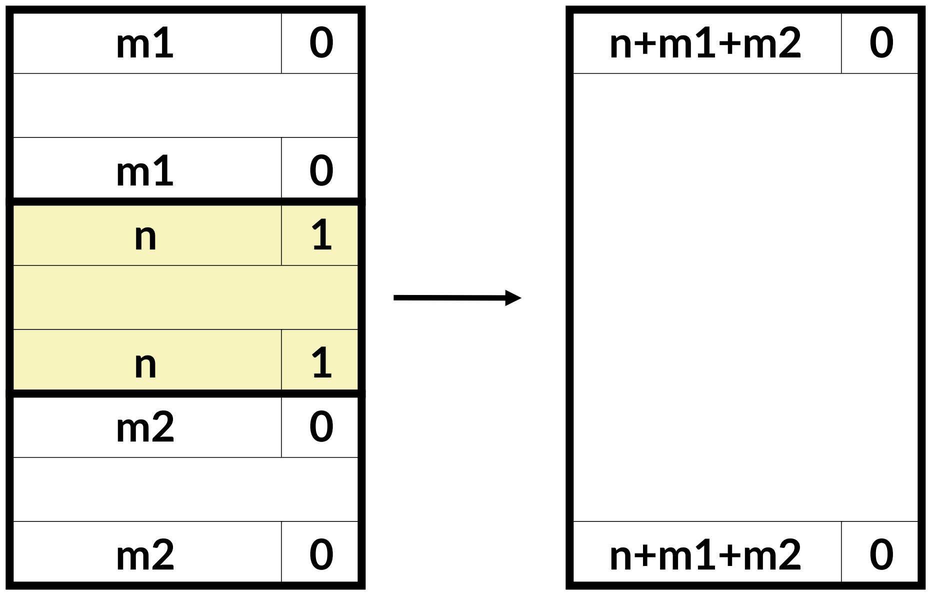
Constant Time Coalescing (Case 2)



Constant Time Coalescing (Case 3)



Constant Time Coalescing (Case 4)



Implicit Free Lists: Summary

- **Implementation: very simple**
- **Allocate cost:**
  - linear time (in total number of heap blocks) worst case
- **Free cost:**
  - constant time worst case
  - even with coalescing
- **Memory utilization:**
  - will depend on placement policy
  - First-fit, next-fit, or best-fit

- **Not used in practice for `malloc/free` because of linear-time allocation**
  - used in some special purpose applications

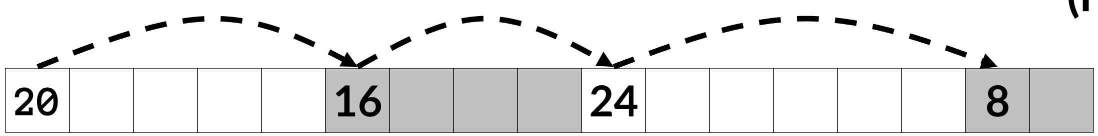
- ***Concepts* of splitting and boundary tag coalescing are general to *all (?)* allocators**

Keeping Track of Free Blocks

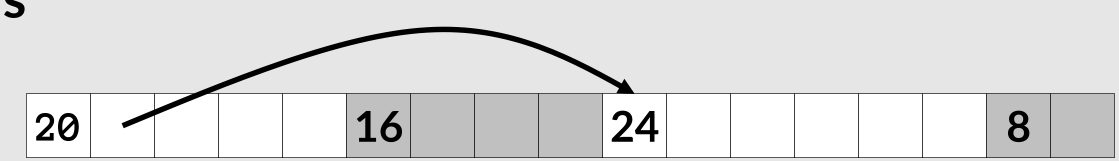
□ = 4 byte word (free)

■ = 4 byte word (allocated)

- Method 1: *Implicit free list* using length— links <u>all</u> blocks using math (no actual pointers)



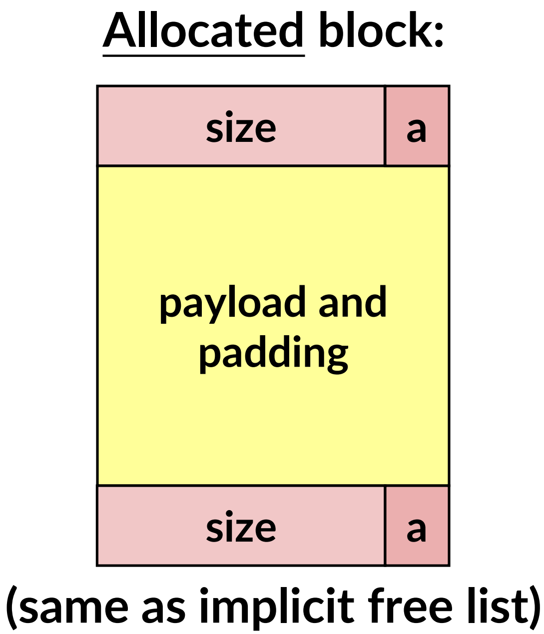
- Method 2: *Explicit free list* among <u>only the free blocks</u>, using pointers



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

- Method 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



- **Maintain 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 forward/back pointers, not just sizes
  - Luckily we track only free blocks, so we can use payload area for pointers
  - Still need boundary tags for coalescing

Explicit Free Lists

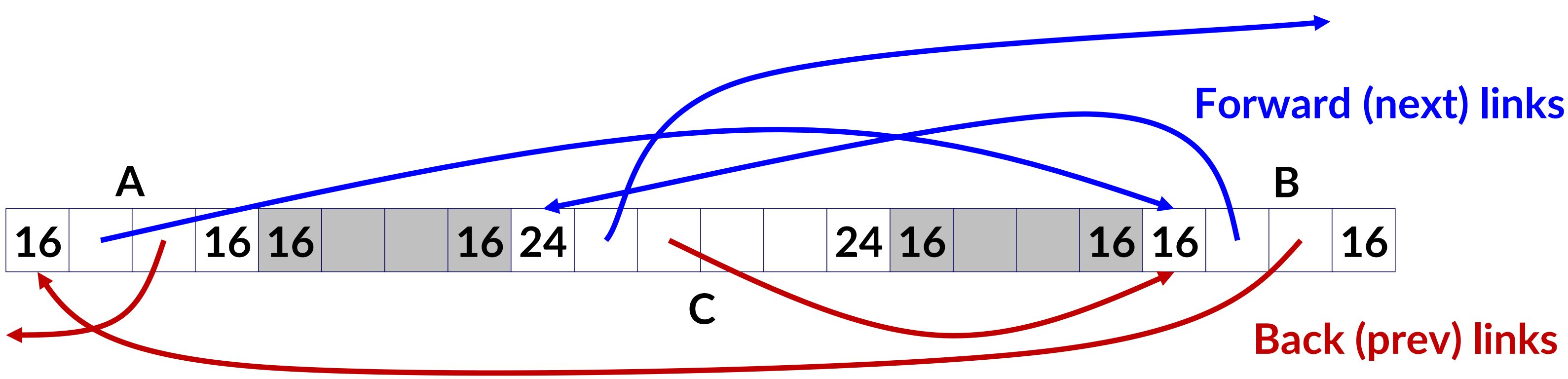
- **Logically (doubly-linked lists):**

- **Physically?**

Explicit Free Lists

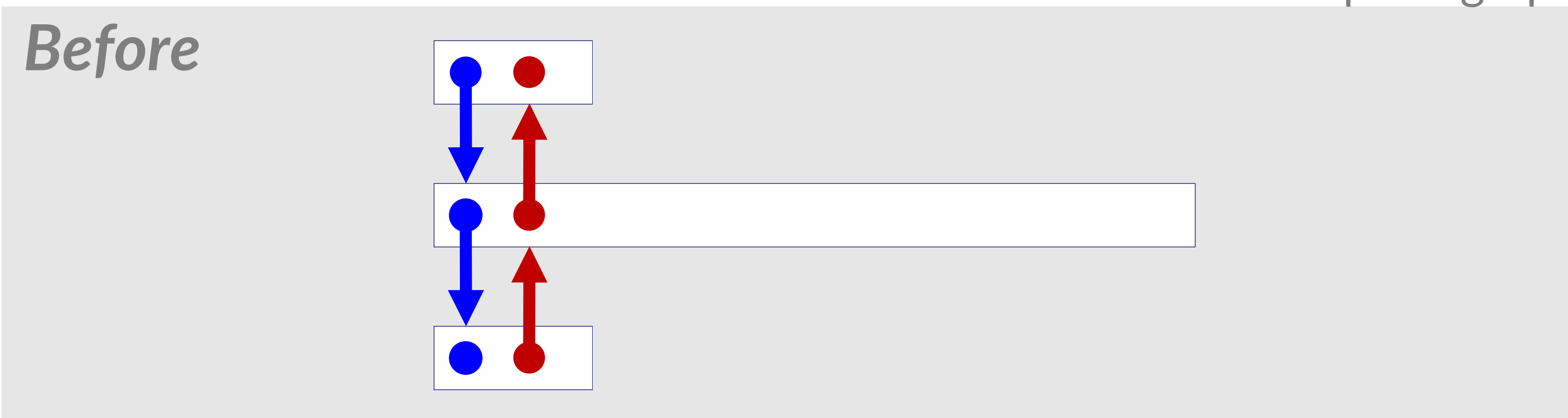
- Logically (doubly-linked lists):

- Physically: blocks can be in any order



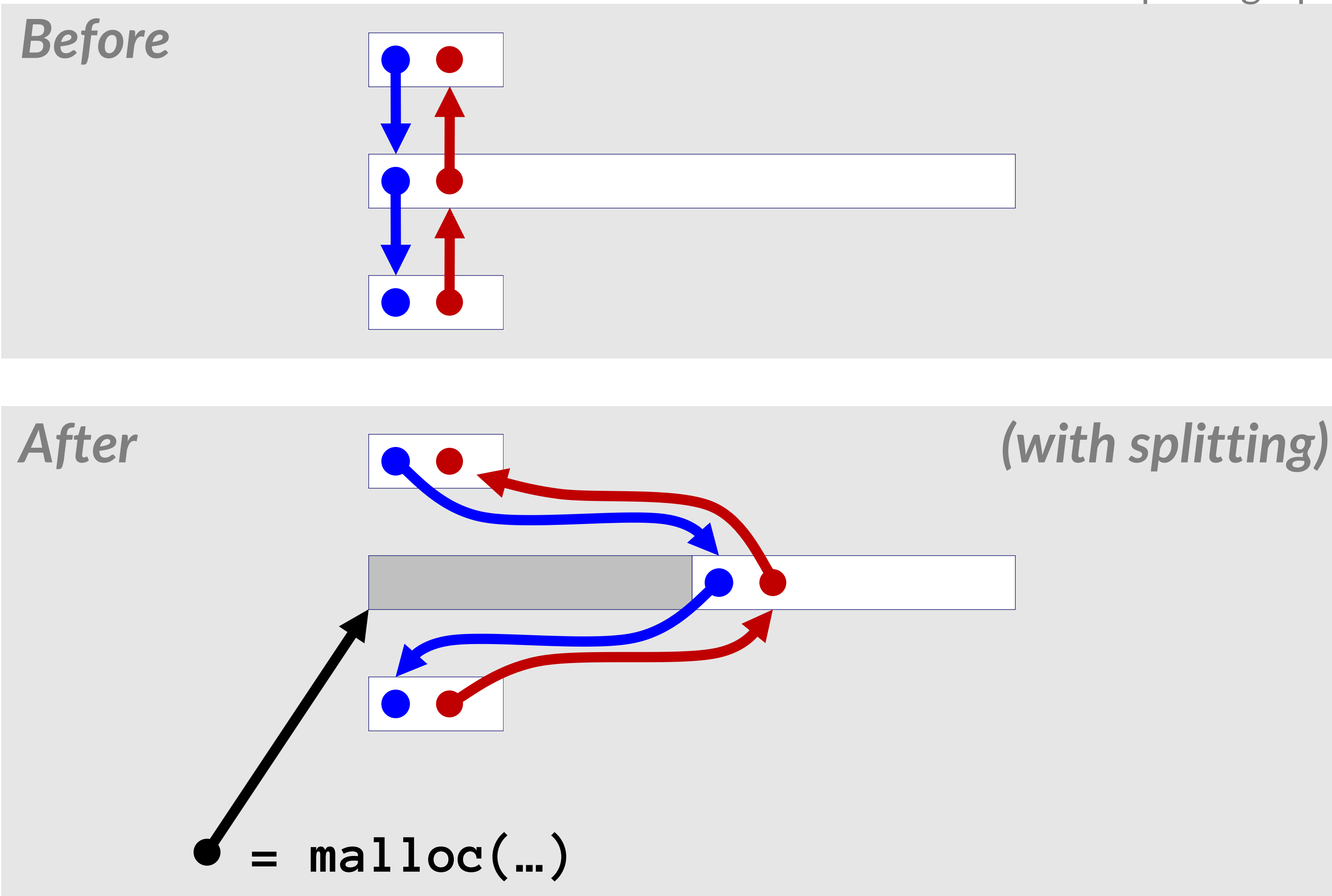
Allocating From Explicit Free Lists

conceptual graphic



Allocating From Explicit Free Lists

conceptual graphic



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

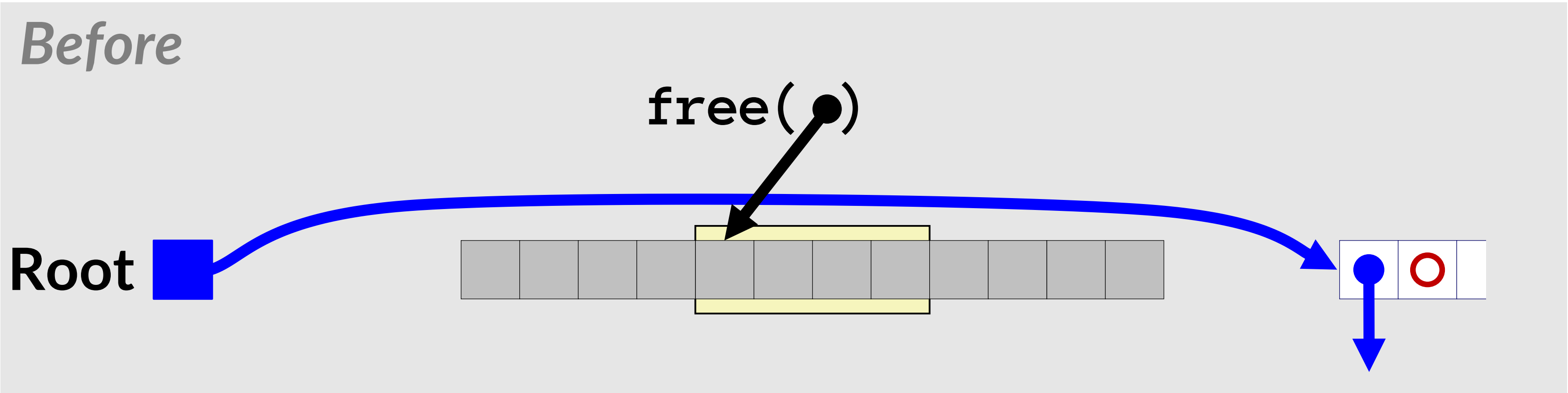
- *Insertion policy*: Where in the free list do you put a newly freed block?

Freeing With Explicit Free Lists

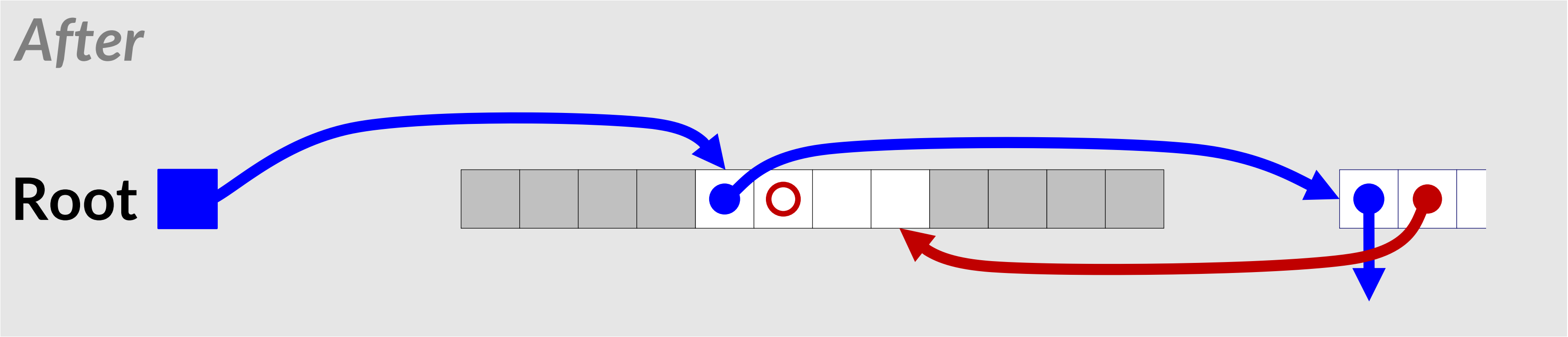
- ***Insertion policy*: Where in the free list do you put a newly freed block?**
  - LIFO (last-in-first-out) policy
    - Insert freed block at the beginning of the free list
    - ***Pro:*** simple and constant time
    - ***Con:*** studies suggest fragmentation is worse than address ordered
  - Address-ordered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

Freeing With a LIFO Policy (Case 1)

conceptual graphic

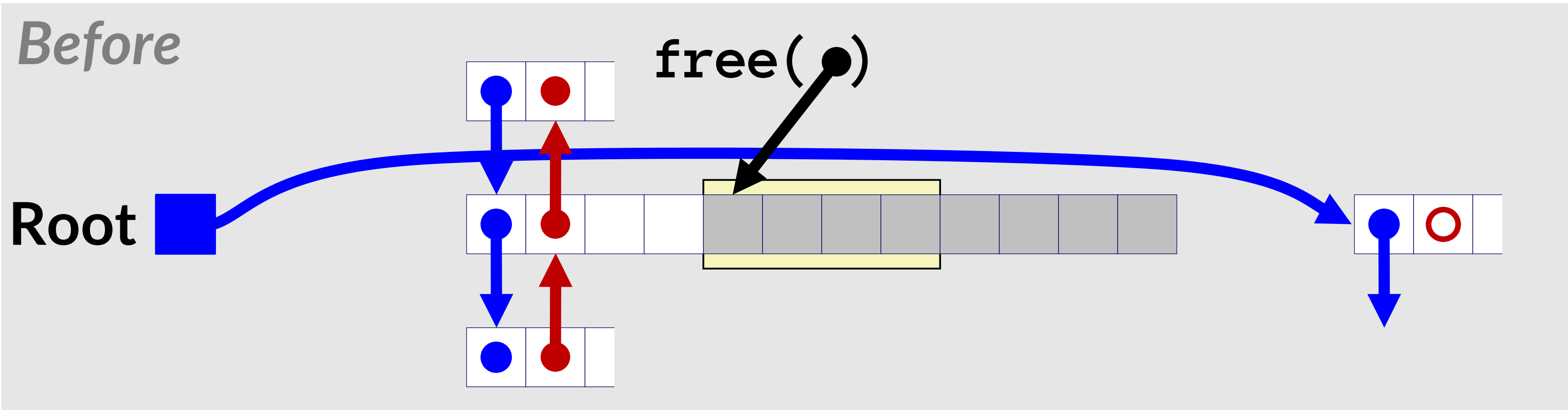


- Insert the freed block at the root of the list

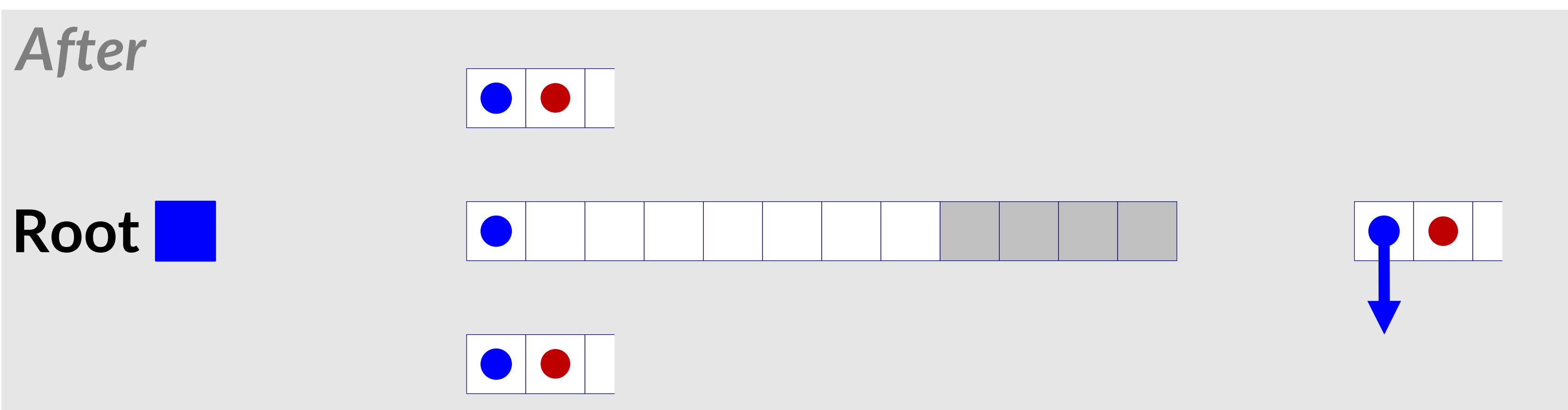


Freeing With a LIFO Policy (Case 2)

conceptual graphic

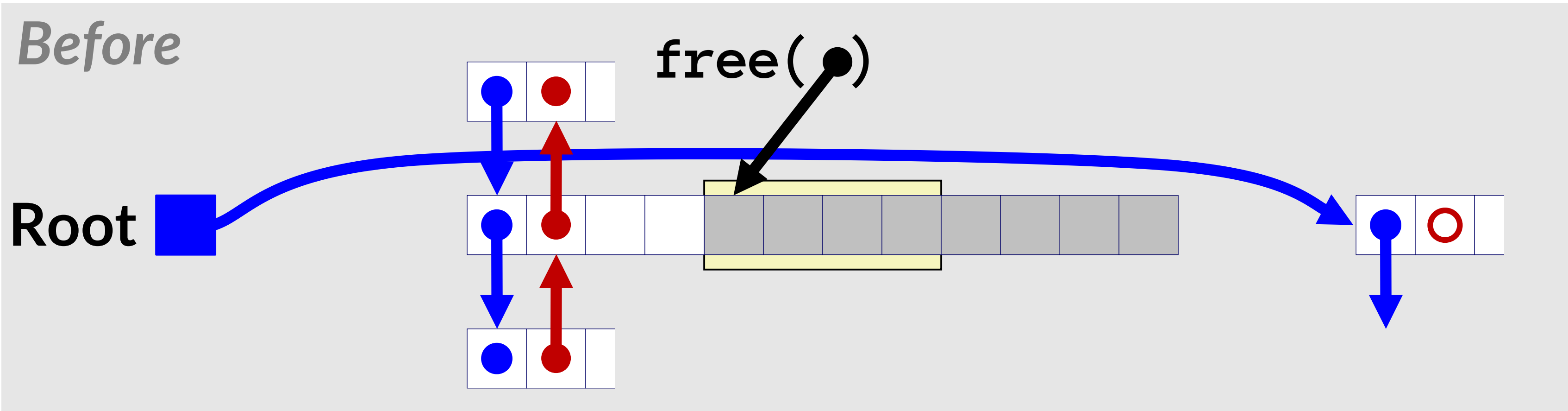


- Splice ***predecessor*** block out of list, coalesce both memory blocks, and insert the new block at the root of the list

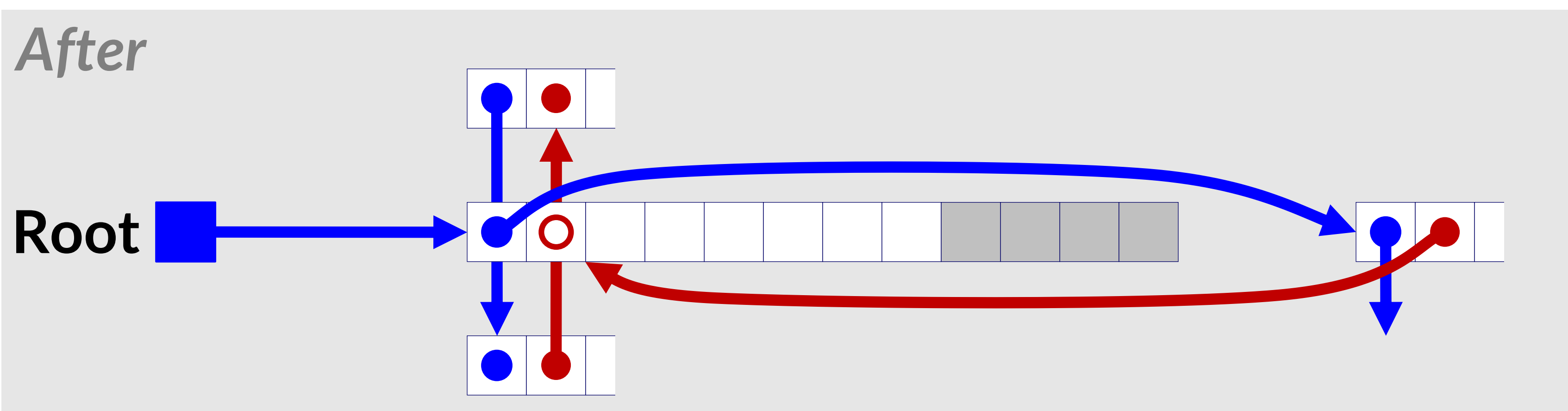


Freeing With a LIFO Policy (Case 2)

conceptual graphic

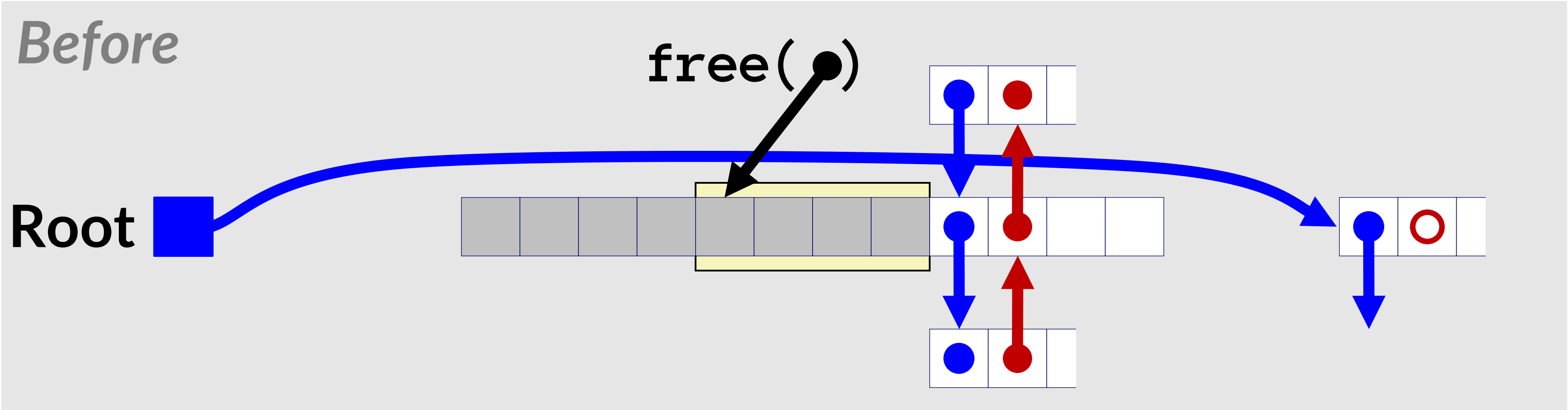


- **Splice *predecessor* block out of list, coalesce both memory blocks, and insert the new block at the root of the list**

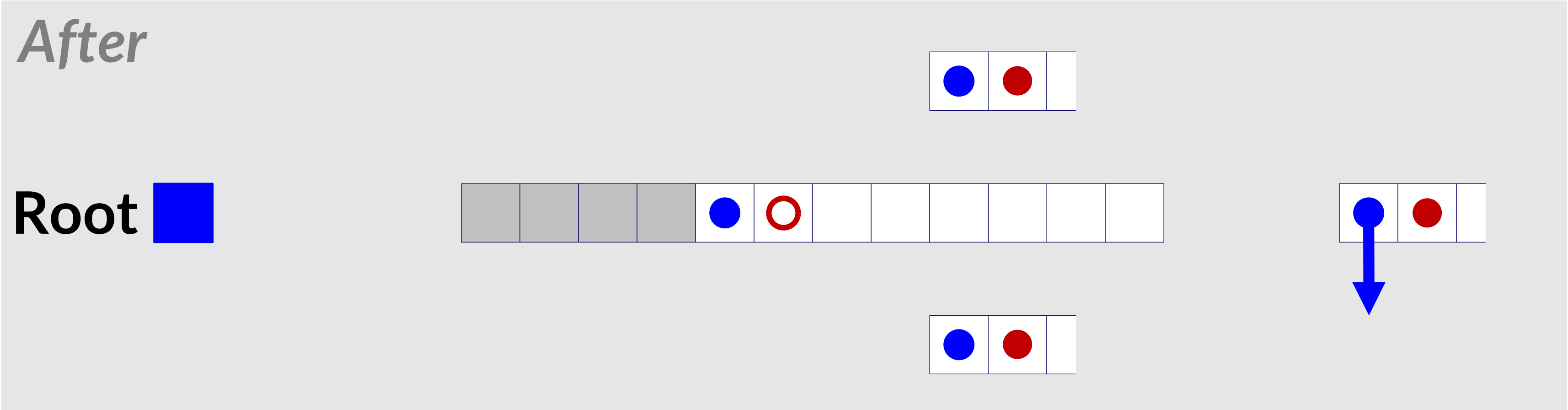


Freeing With a LIFO Policy (Case 3)

conceptual graphic

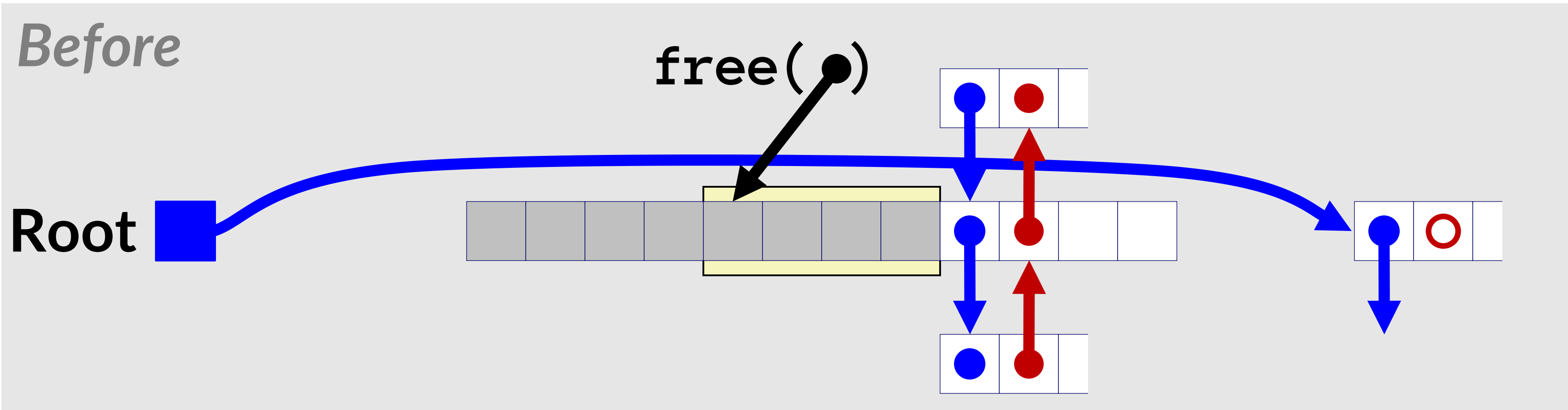


- Splice ***successor*** block out of list, coalesce both memory blocks and insert the new block at the root of the list

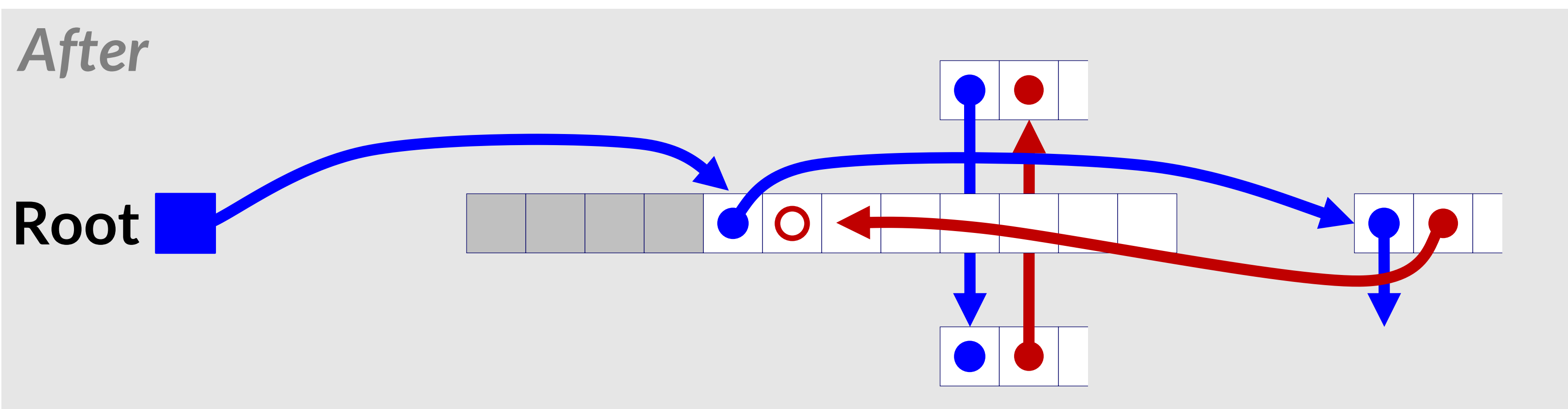


Freeing With a LIFO Policy (Case 3)

conceptual graphic

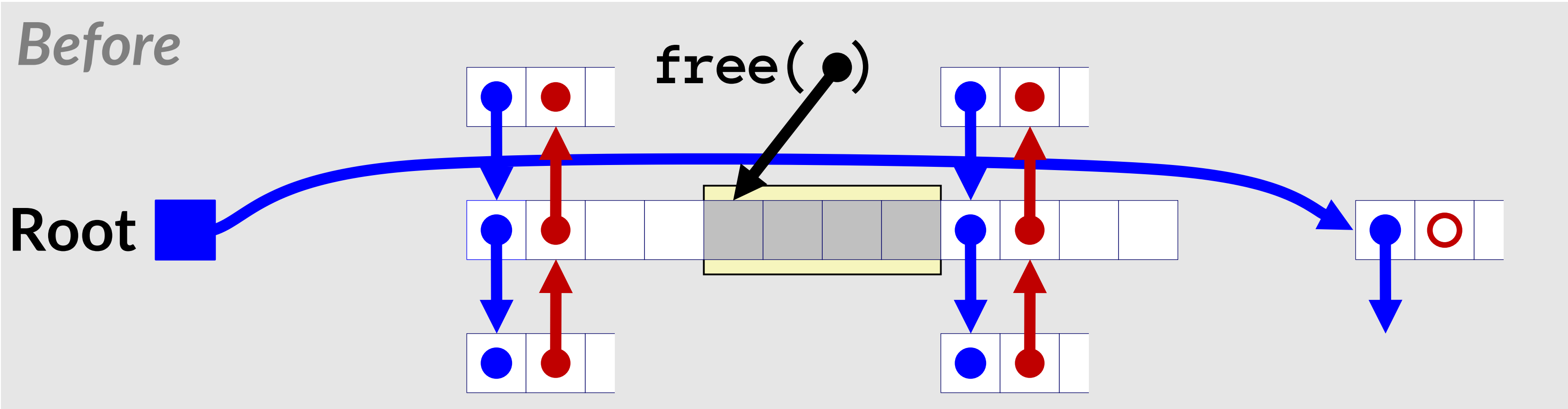


- Splice *successor* block out of list, coalesce both memory blocks and insert the new block at the root of the list

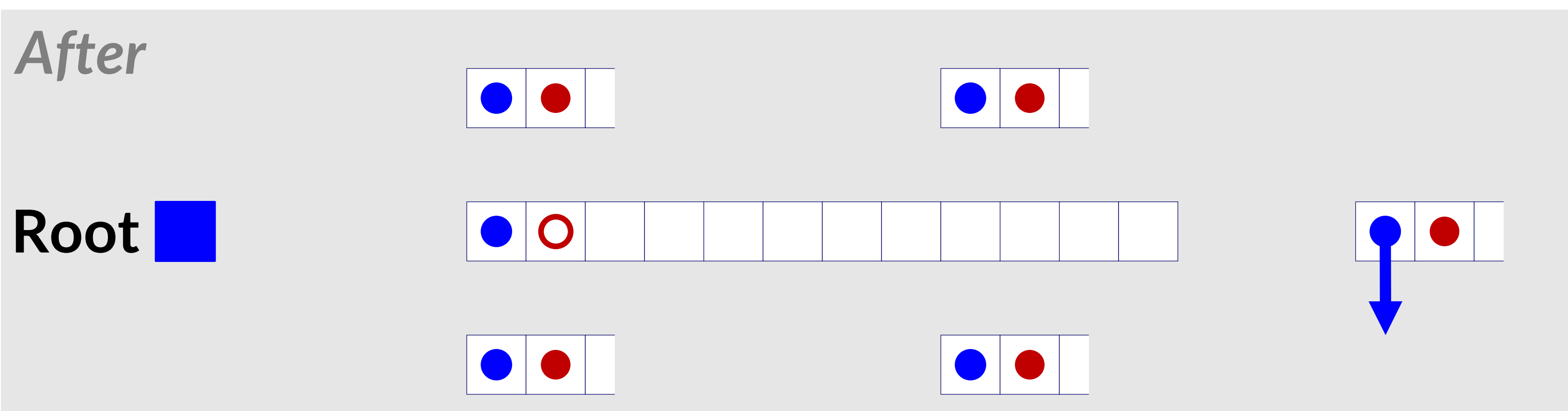


Freeing With a LIFO Policy (Case 4)

conceptual graphic

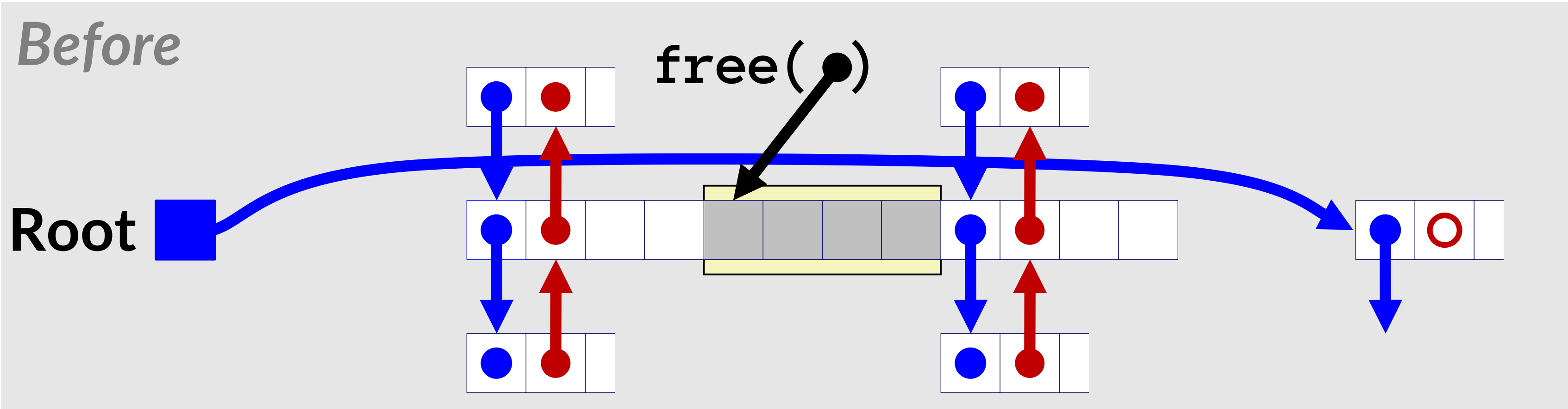


- Splice ***predecessor*** and ***successor*** blocks out of list, coalesce all 3 memory blocks and insert the new block at the root of the list

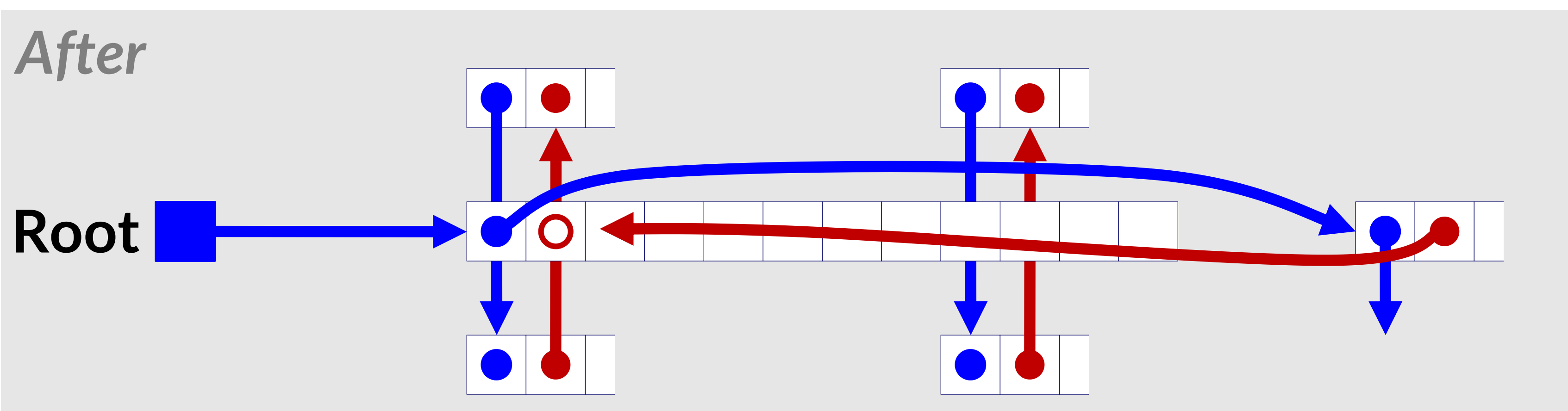


Freeing With a LIFO Policy (Case 4)

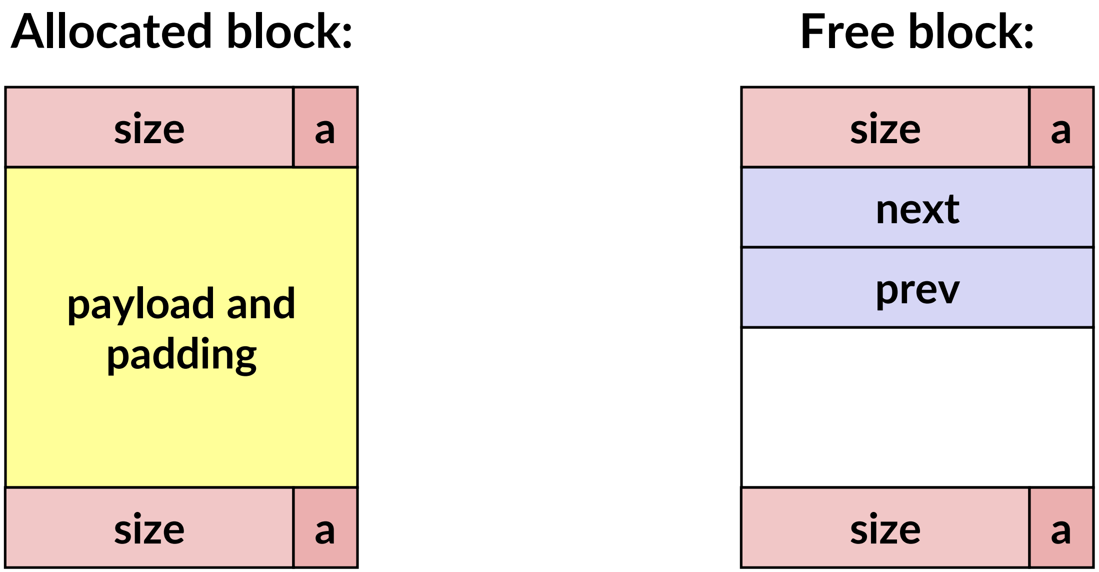
conceptual graphic



- 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 tag?



- Lab 5 suggests no…

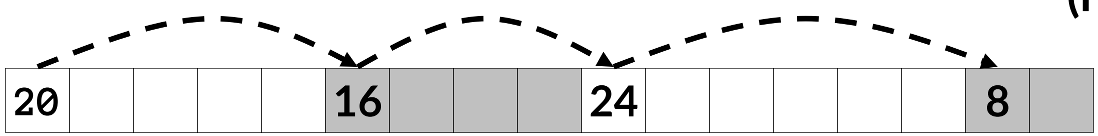
Explicit List Summary

- **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 since needs to splice blocks in and out of the list
  - Some extra space for the links (2 extra words needed for each block)
    - Possibly increases minimum block size, leading to more internal fragmentation

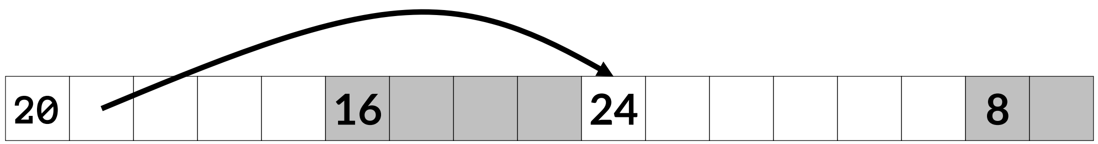
- **Most common use of explicit lists is in conjunction with *segregated free lists***
  - Keep multiple linked lists of different size classes, or possibly for different types of objects

Keeping Track of Free Blocks

- Method 1: *Implicit free list* using length— links <u>all</u> blocks using math (no actual pointers)



- Method 2: *Explicit free list* among <u>only the free blocks</u>, using pointers

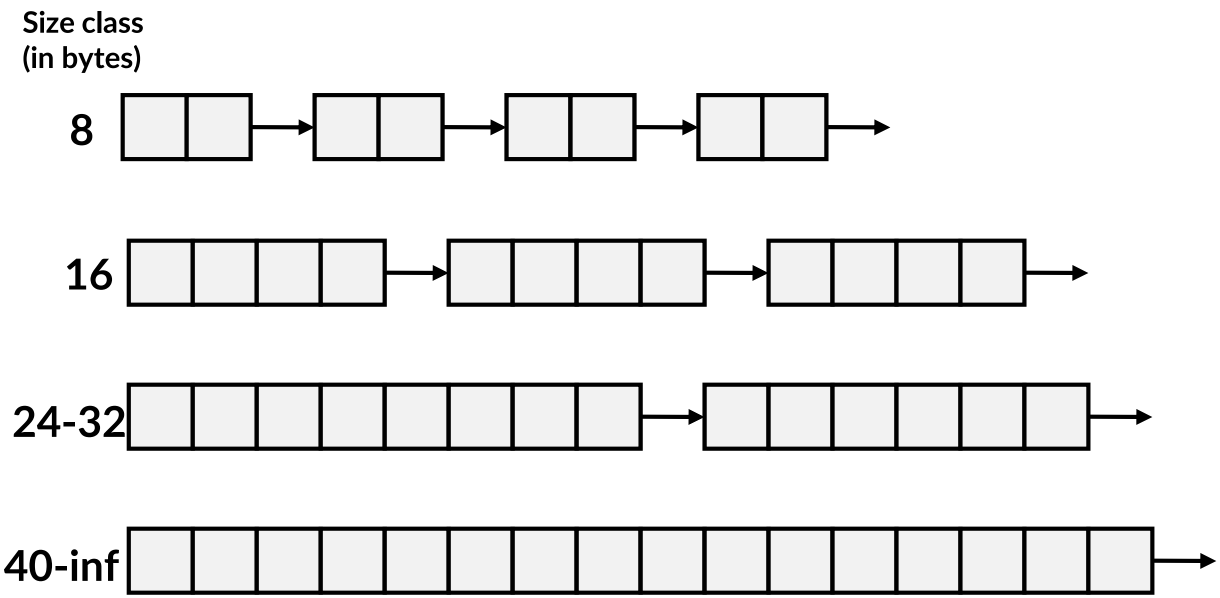


- Method 3: *Segregated free list*
  - Different free lists for different size classes

- Method 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



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

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`**)
  - Allocate block of *n* bytes from this new memory
  - Place remainder as a single free block in appropriate size class

Seglist Allocator

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

- **Advantages of seglist allocators**
  - Higher throughput
    - log time for power-of-two size classes
  - Better memory utilization
    - First-fit search of seglist approximates a best-fit search of entire heap
    - Extreme case: Giving each block its own size class is equivalent to best-fit
    - Don't need to use space for block size if it's a fixed-size list

Summary of Key Allocator Policies

- **Placement policy:**
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
  - *Observation:* segregated free lists approximate a best fit placement policy without having to search entire free list

- **Splitting policy:**
  - When do we go ahead and split free blocks?
  - How much internal fragmentation are we willing to tolerate?

- **Coalescing policy:**
  - *Immediate coalescing:* coalesce each time `free` is called
  - *Deferred coalescing:* try to improve performance of `free` by deferring coalescing until needed. Examples:
    - Coalesce as you scan the free list for `malloc`
    - Coalesce when the amount of 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)

Wouldn't it be nice…

- **If we never had to free memory?**
- **Do you free objects in Java?**

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

Garbage Collection

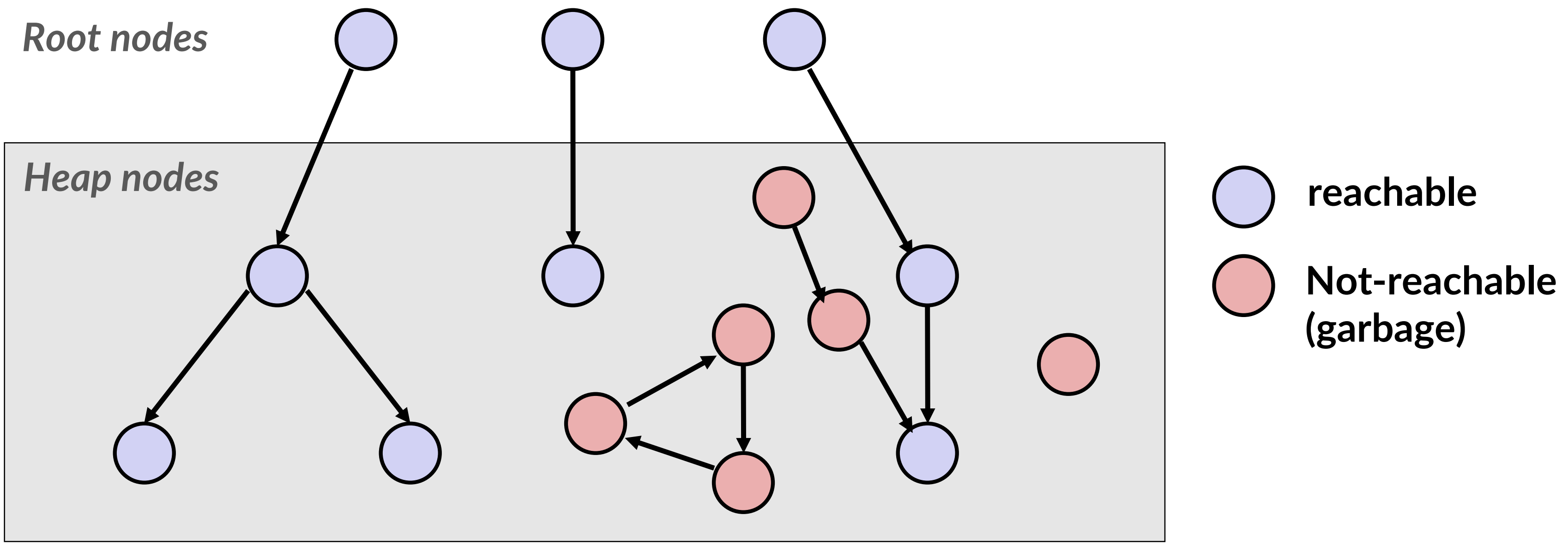
- **How does the memory allocator know when memory can be freed?**
  - In general, we cannot know what is going to be used in the future since it depends on conditionals
  - But, we can tell that certain blocks cannot be used if they are *unreachable* (via pointers starting at registers/stack/globals)

- **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)

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

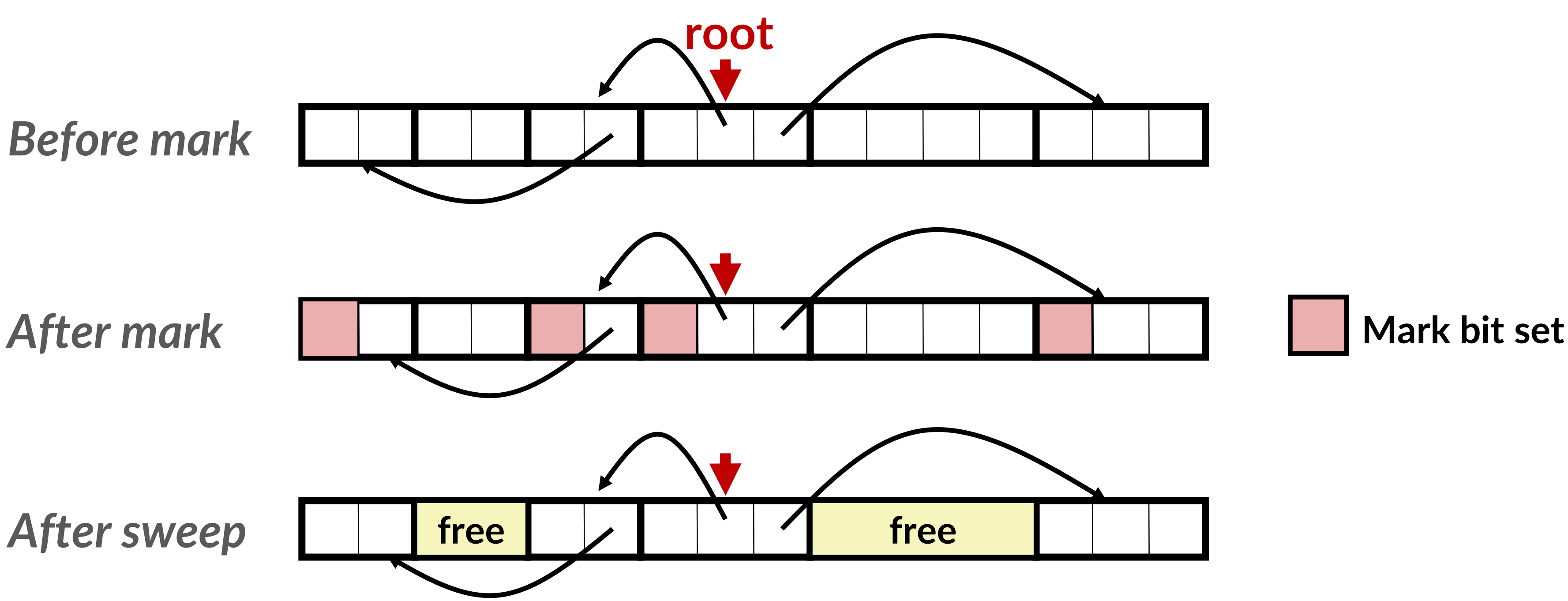
Non-reachable nodes are *garbage* (cannot be needed by the application)

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 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 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**
  - `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

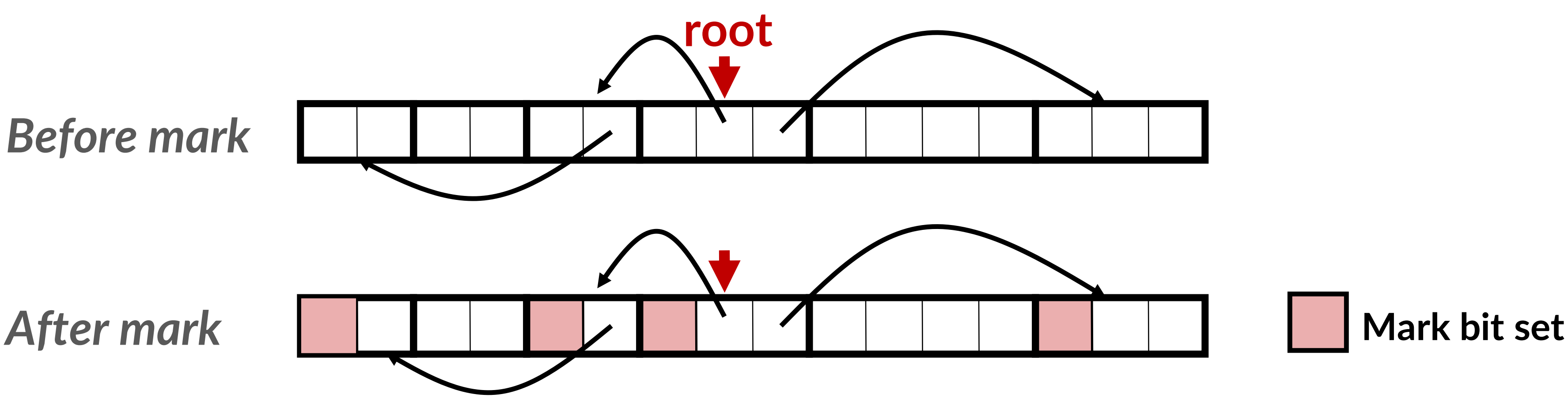
Extra Material

Mark

Extra Material

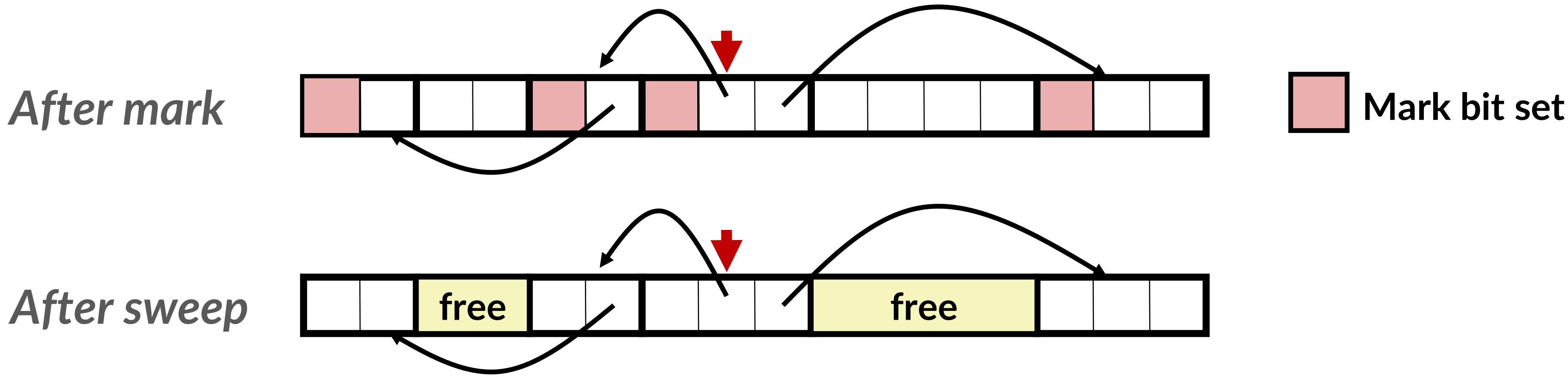
Mark using depth-first traversal of the memory graph

```
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;
}
```



Sweep

Extra Material



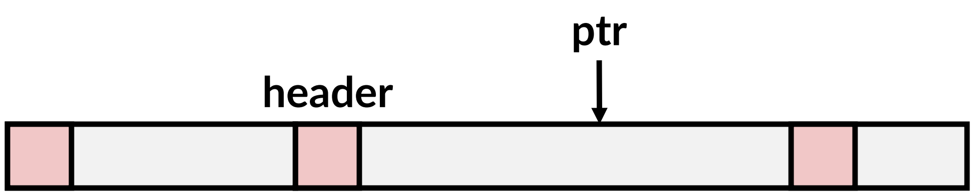
Sweep using lengths to find next block

```
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(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

Extra Material

- **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
  - 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

!!!

A. Failing to Free Blocks
B. Misunderstanding pointer arithmetic
C. Off by one error
D. Freeing blocks multiple times
E. Referencing a pointer instead of the object it points to
F. Not checking the max string size
G. Interpreting something that is not a ptr as a ptr
H. Accessing Freed Blocks
I. Referencing nonexistent variables
J. Allocating the (possibly) wrong sized object
K. Reading uninitialized memory

Dereferencing Bad Pointers

- The classic `scanf` bug

```
int val;

...

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

- **Will cause `scanf` to interpret contents of `val` as an address!**
  - **Best case:** program terminates immediately due to segmentation fault
  - **Worst case:** contents of `val` correspond to some valid read/write area of virtual memory, causing `scanf` to overwrite that memory, with disastrous and baffling consequences much later in program execution

Reading Uninitialized Memory

- **Wrongly assuming that heap data is initialized to zero**

```
/* 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;
}
```

Overwriting Memory

- **Allocating the (possibly) wrong sized object**

```
int **p;

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

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

Overwriting Memory

- **Off-by-one error**

```
int **p;

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

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

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**
  - Lab 3

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* getPacket(int** packets, int* size) {
   int* packet;
   packet = packets[0];
   packets[0] = packets[*size - 1];
   *size--;    // what is happening here?
   reorderPackets(packets, *size);
   return packet;
}
```

- **'--' and '*' operators have same precedence and associate from right-to-left, so -- happens first!**

Referencing Stack Variables Too Late

- **Forgetting that local variables disappear when a function returns (call-stack space reused by subsequent calls)**

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

   return &val;
}
```

Freeing Blocks Multiple Times

- **Nasty!**

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

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

Freeing Blocks Multiple Times

- Nasty!

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

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

- What does the free list look like?

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

Referencing Freed Blocks

- **Evil!**

```
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]++;
```

Failing to Free Blocks (Memory Leaks)

- **Slow, silent, long-term killer!**

```
void foo() {
   int* x = (int*)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;
};

void foo() {
   struct list *head =
      (struct list *)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

- **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