

Roadmap

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();
```

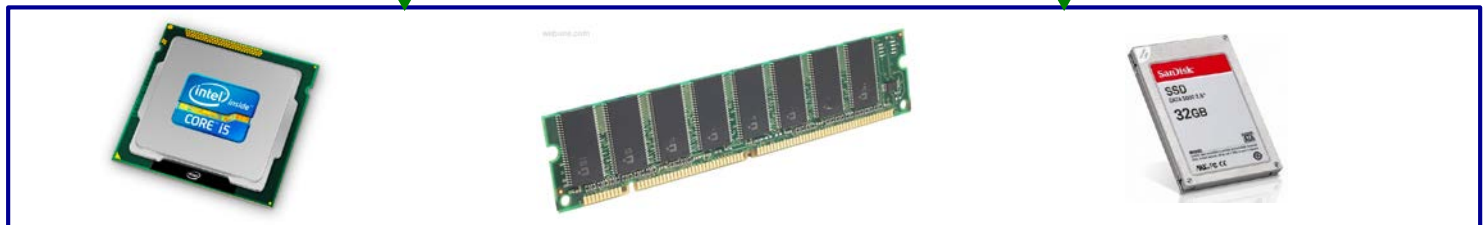
Assembly language:

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

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Computer system:



Memory Allocation

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

OS:



Section 10: Memory Allocation Topics

■ Dynamic memory allocation

- Size/number of data structures may only be known at run time
- Need to allocate space on the heap
- Need to de-allocate (free) unused memory so it can be re-allocated

■ Implementation

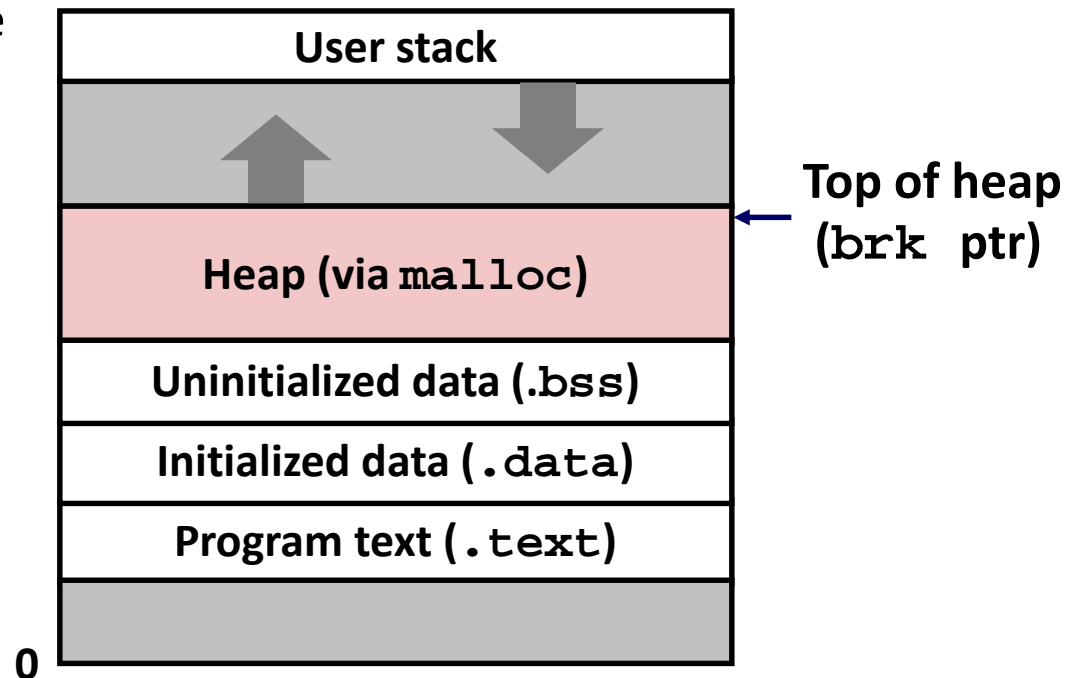
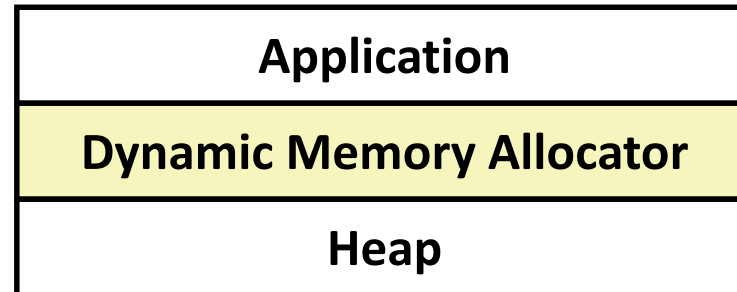
- Implicit free lists
- Explicit free lists – subject of next programming assignment
- Segregated free lists

■ Garbage collection

■ Common memory-related bugs in C programs

Dynamic Memory Allocation

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



Dynamic Memory Allocation

- **Allocator maintains heap as collection of variable sized *blocks*, which are either *allocated* or *free***
 - Allocator requests space in heap region; VM hardware and kernel allocate these pages to the process
 - Application objects are typically smaller than pages, so the allocator manages blocks *within* pages
- **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

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 8-byte boundary
 - If **size == 0**, returns NULL
- Unsuccessful: returns NULL and sets **errno**

```
void free(void *p)
```

- Returns the block pointed at by **p** to pool of available memory
- **p** must come from a previous call to **malloc** or **realloc**

Other functions

- **calloc**: Version of **malloc** that initializes allocated block to zero.
- **realloc**: Changes the size of a previously allocated block.
- **sbrk**: Used internally by allocators to grow or shrink the heap.

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 */
    if ((p = (int *)realloc(p, (n+m) * sizeof(int))) == 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 */
}
```

Section 10: Memory Allocation Topics

■ Dynamic memory allocation

- Size/number of data structures may only be known at run time
- Need to allocate space on the heap
- Need to de-allocate (free) unused memory so it can be re-allocated

■ Implementation

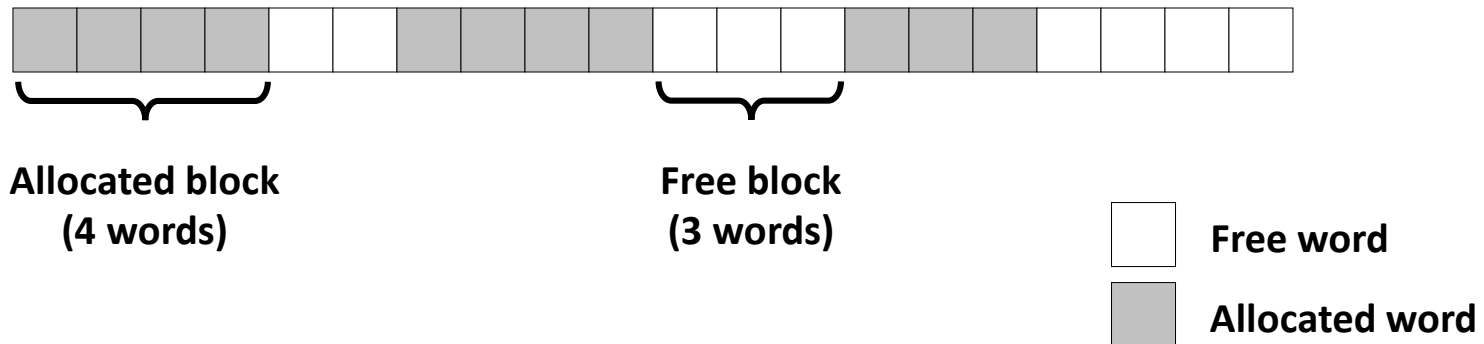
- Implicit free lists
- Explicit free lists – subject of next programming assignment
- Segregated free lists

■ Garbage collection

■ Common memory-related bugs in C programs

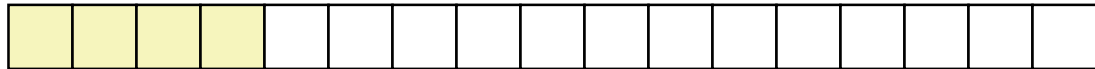
Assumptions Made in This Section

- **Memory is word addressed (each word can hold a pointer)**
 - block size is a multiple of words

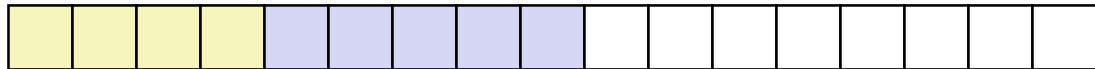


Allocation Example

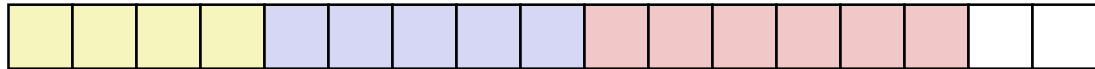
```
p1 = malloc(4)
```



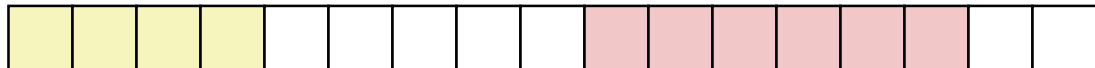
```
p2 = malloc(5)
```



```
p3 = malloc(6)
```



```
free(p2)
```



```
p4 = malloc(2)
```



What information does the allocator need to keep track of?

Constraints

■ Applications

- Can issue arbitrary sequence of `malloc()` and `free()` requests
- `free()` requests must be made only for a previously `malloc()`'d block

■ 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
- Must align blocks so they satisfy all alignment requirements
 - 8 byte alignment for GNU `malloc` (**libc** `malloc`) on Linux boxes
- Can't move the allocated blocks once they are `malloc()`'d
 - *i.e.*, compaction is not allowed. *Why not?*

Performance Goal: Throughput

- Given some sequence of `malloc` and `free` requests:
 - $R_0, R_1, \dots, R_k, \dots, R_{n-1}$
- 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: Peak Memory Utilization

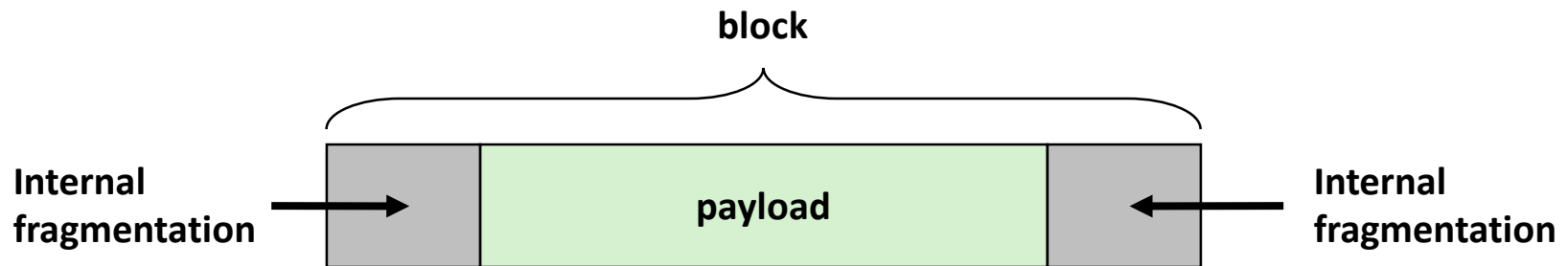
- Given some sequence of `malloc` and `free` requests:
 - $R_0, R_1, \dots, R_k, \dots, R_{n-1}$
- **Def: Aggregate payload P_k**
 - `malloc(p)` results in a block with a **payload** of `p` bytes
 - After request R_k has completed, the **aggregate payload** P_k is the sum of currently allocated payloads
- **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 requests**
 - $U_k = (\max_{i < k} P_i) / H_k$
 - 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*
 - *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?
- **Depends only on the pattern of *previous* requests**
 - thus, easy to measure

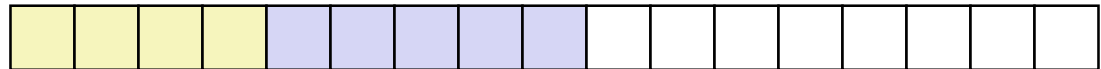
External Fragmentation

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

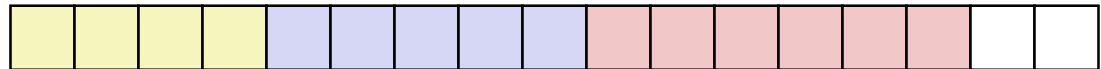
```
p1 = malloc(4)
```



```
p2 = malloc(5)
```



```
p3 = malloc(6)
```



```
free(p2)
```



```
p4 = malloc(6)
```

Oops! (what would happen now?)

- Depends on the pattern of future requests
 - Thus, difficult to measure

Section 10: Memory Allocation Topics

■ ~~Dynamic memory allocation~~

- ~~Size/number of data structures may only be known at run time~~
- ~~Need to allocate space on the heap~~
- ~~Need to de-allocate (free) unused memory so it can be re-allocated~~

■ Implementation

- Implicit free lists
- Explicit free lists – subject of next programming assignment
- Segregated free lists

■ Garbage collection

■ Common memory-related bugs in C programs

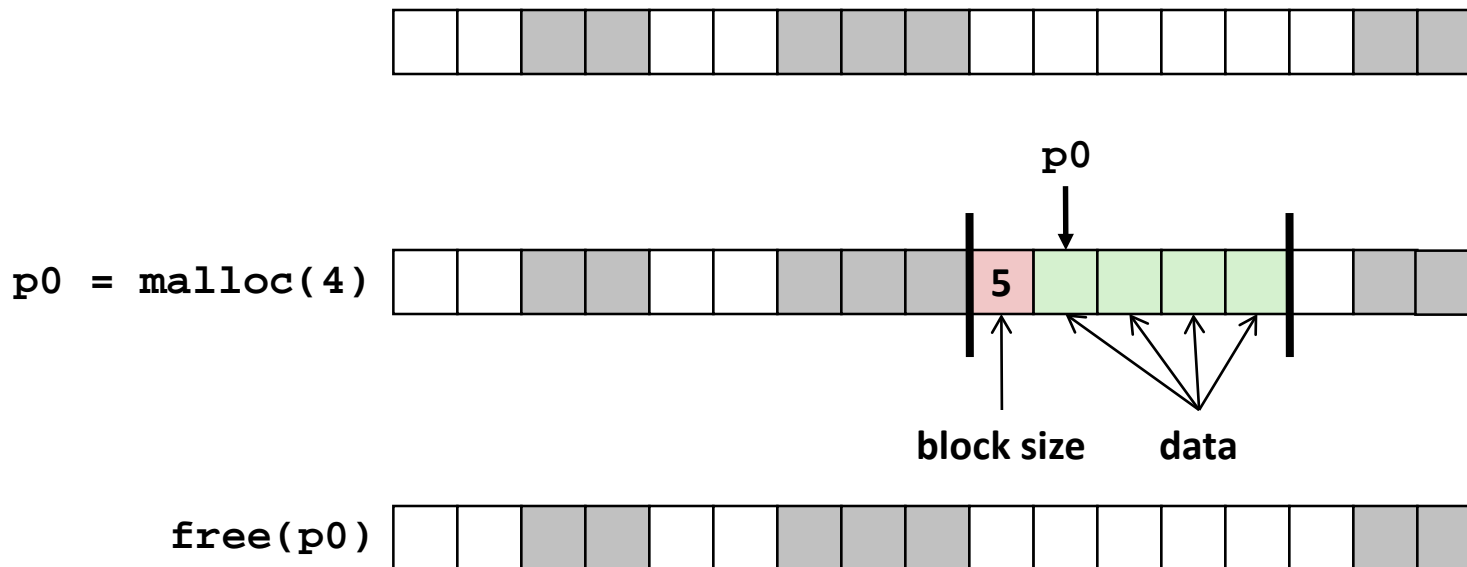
Implementation Issues

- 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 freed block into the heap?

Knowing How Much to Free

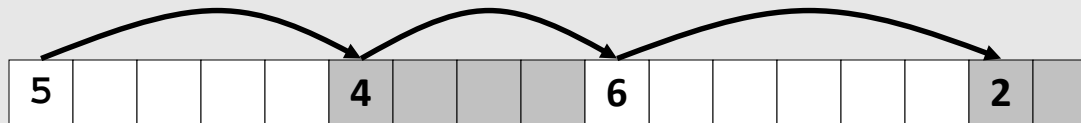
■ Standard method

- 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

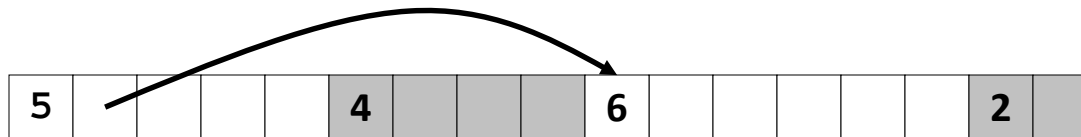


Keeping Track of Free Blocks

- 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 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: size, is-allocated?**
 - Could store this information in two words: wasteful!

- **Standard trick**

- If blocks are aligned, some low-order size bits are always 0
- Instead of storing an always-0 bit, use it as a allocated/free flag
- When reading size, must remember to mask out this bit

e.g. with 8-byte alignment,
sizes look like:

00000000

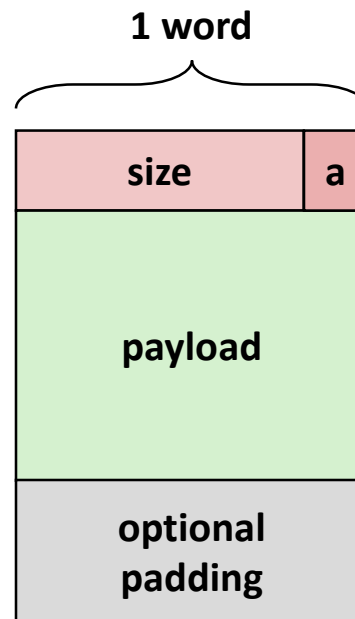
00001000

00010000

00011000

...

*Format of
allocated and
free blocks*



a = 1: allocated block

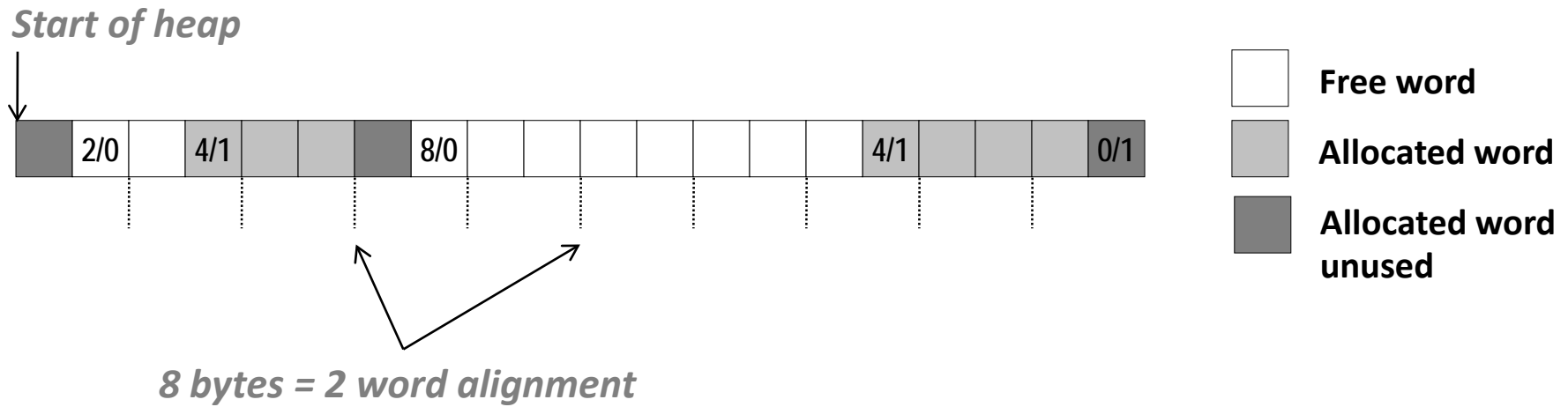
a = 0: free block

size: block size

**payload: application data
(allocated blocks only)**

Implicit Free List Example

Sequence of blocks in heap: 2/0, 4/1, 8/0, 4/1 (size/allocated)



- **8-byte alignment**
 - May require initial unused word
 - Causes some internal fragmentation
- **One word (0/1) to mark end 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 passed end
      ((*p & 1) ||          \\ already allocated
      (*p <= len)))        \\ too small
  p = p + (*p & -2);        \\ goto next block
```

- 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

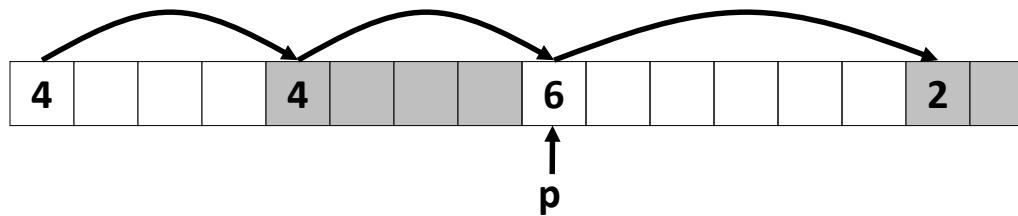
■ *Best fit:*

- Search the list, choose the *best* free block: fits, with fewest bytes left over
- Keeps fragments small—usually helps fragmentation
- Will typically run slower than first-fit

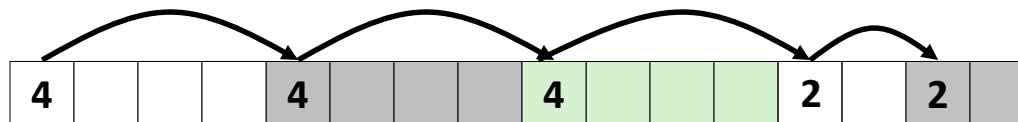
Implicit List: Allocating in Free Block

■ Allocating in a free block: *splitting*

- Since allocated space might be smaller than free space, we might want to split the block



addblock(p, 4)



```
void addblock(ptr p, int len) {
    int newsize = ((len + 1) >> 1) << 1; // round up to even
    int oldsize = *p & -2; // mask out low bit
    *p = newsize | 1; // set new length + allocated
    if (newsize < oldsize)
        *(p+newsized) = oldsize - newsized; // set length in remaining
                                                // part of block
}
```

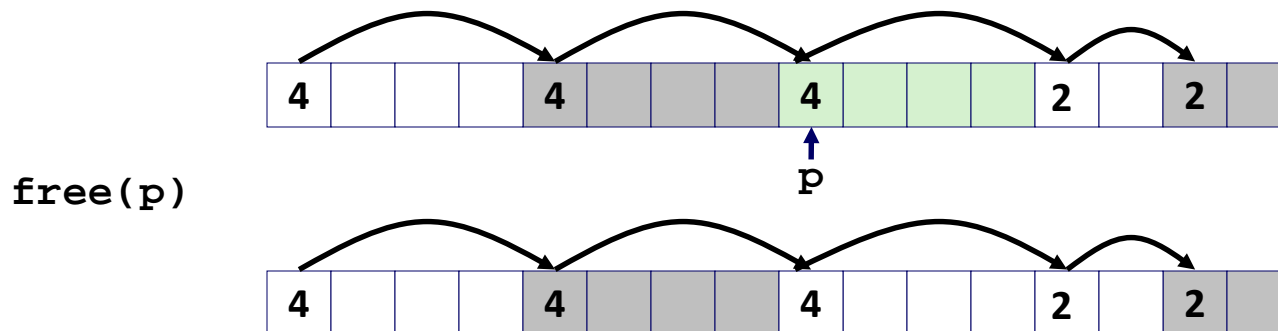
Implicit List: Freeing a Block

■ Simplest implementation:

- Need only clear the “allocated” flag

```
void free_block(ptr p) { *p = *p & -2 }
```

- But can lead to “false fragmentation”

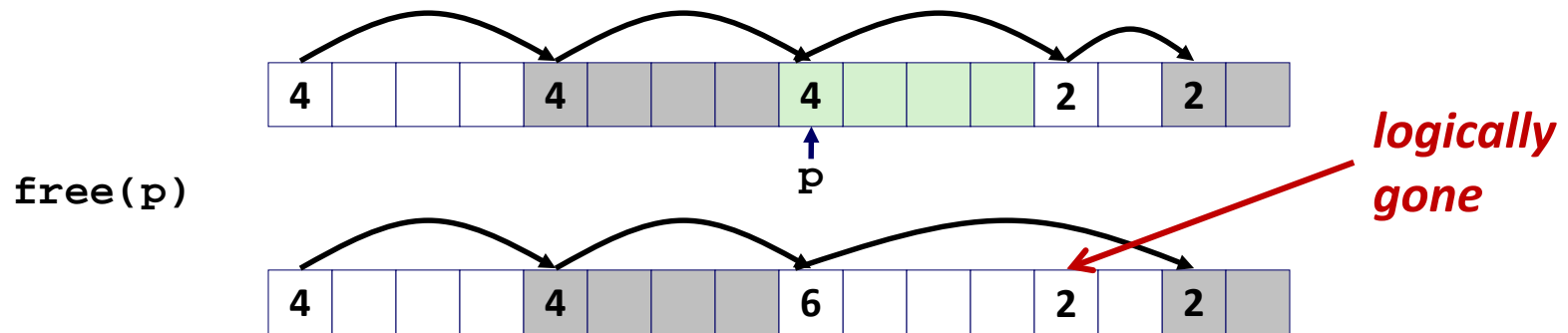


malloc(5) **Oops!**

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



```

void free_block(ptr p) {
    *p = *p & -2;           // clear allocated bit
    next = p + *p;         // find next block
    if ((*next & 1) == 0)
        *p = *p + *next;   // add to this block if
                          // not allocated
}

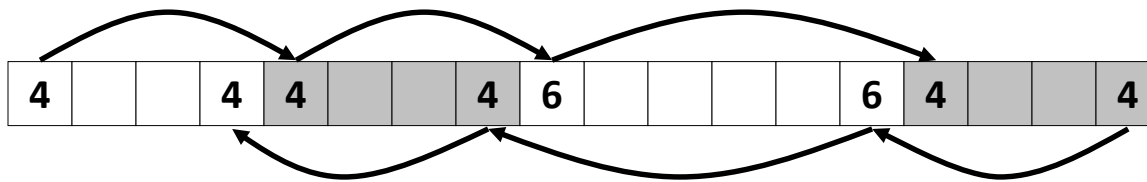
```

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

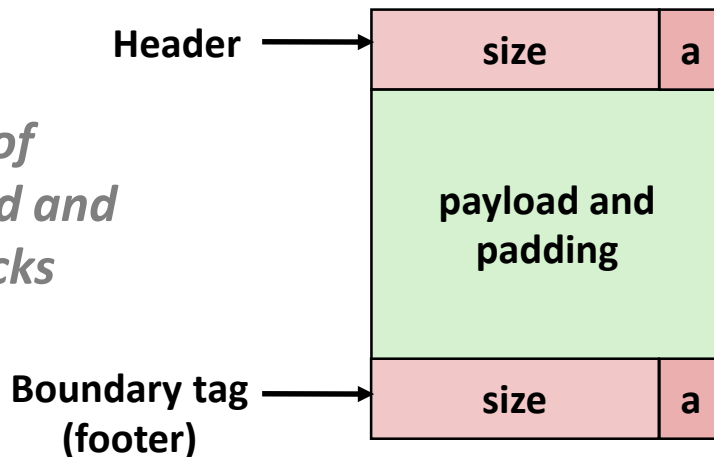
Implicit List: Bidirectional Coalescing

■ *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)**

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
- **The concepts of splitting and boundary tag coalescing are general to *all* allocators**

Section 10: Memory Allocation Topics

■ ~~Dynamic memory allocation~~

- ~~Size/number of data structures may only be known at run time~~
- ~~Need to allocate space on the heap~~
- ~~Need to de-allocate (free) unused memory so it can be re-allocated~~

■ Implementation

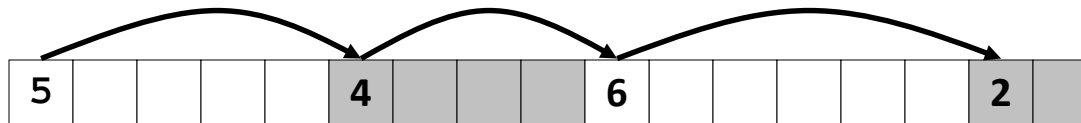
- ~~Implicit free lists~~
- Explicit free lists – subject of next programming assignment
- Segregated free lists

■ Garbage collection

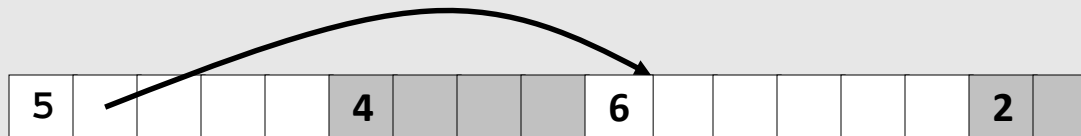
■ Common memory-related bugs in C programs

Keeping Track of Free Blocks

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



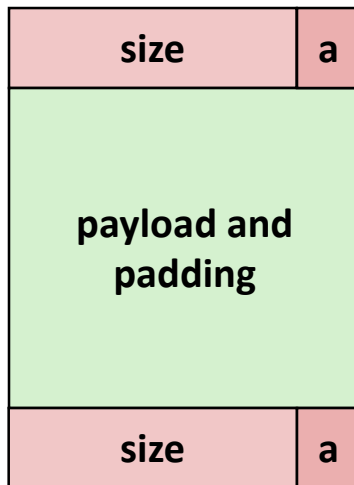
- Method 2: *Explicit free list* among the free blocks using pointers



- 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

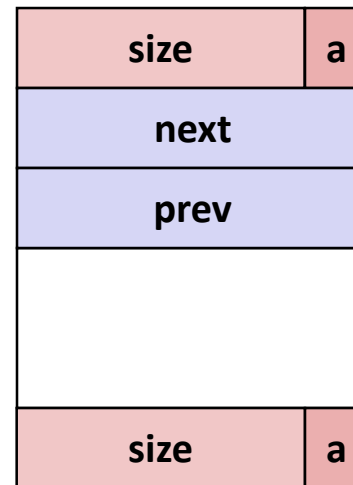
Explicit Free Lists

Allocated block:



(same as implicit free list)

Free block:



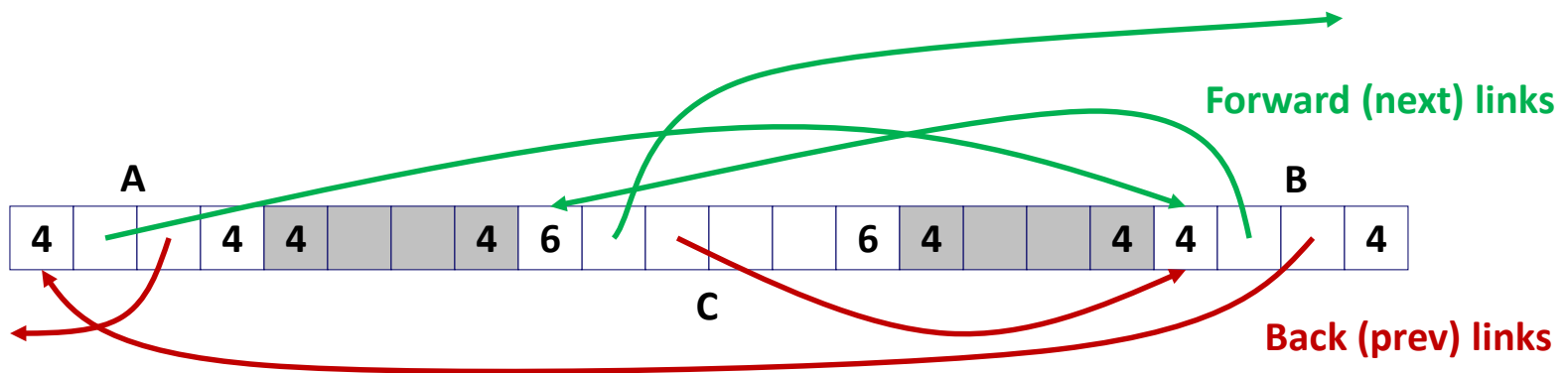
- Maintain list(s) of *free* blocks, rather than implicit list of *all* blocks
 - The “next” free block could be anywhere in the heap
 - So we need to store forward/back pointers, not just sizes
 - Luckily we track only free blocks, so we can use payload area for pointers
 - Still need boundary tags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

Explicit Free Lists

- Logically (doubly-linked lists):



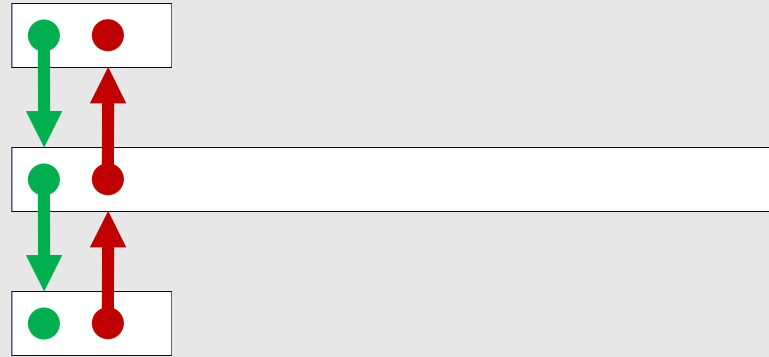
- Physically: blocks can be in any order



Allocating From Explicit Free Lists

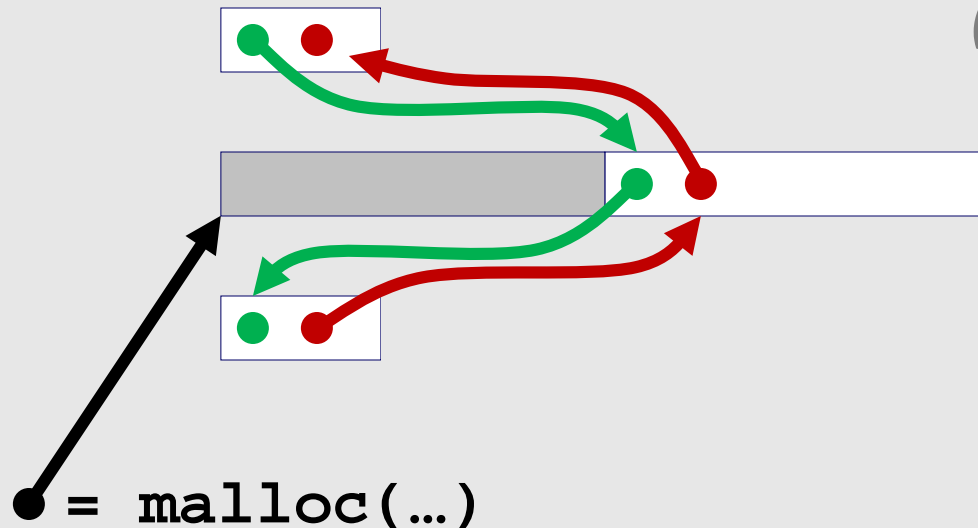
conceptual graphic

Before



After

(with splitting)

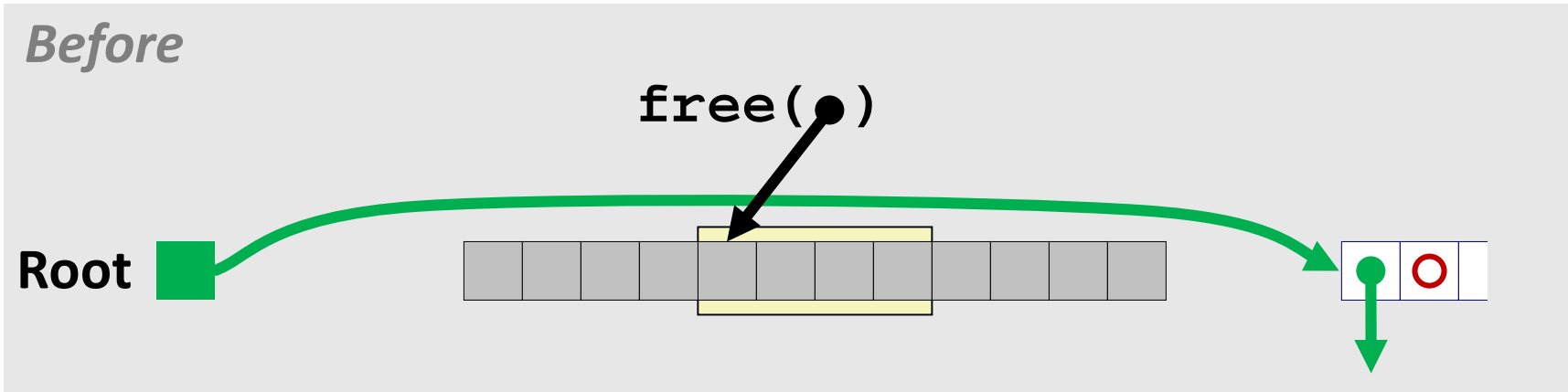


Freeing With Explicit Free Lists

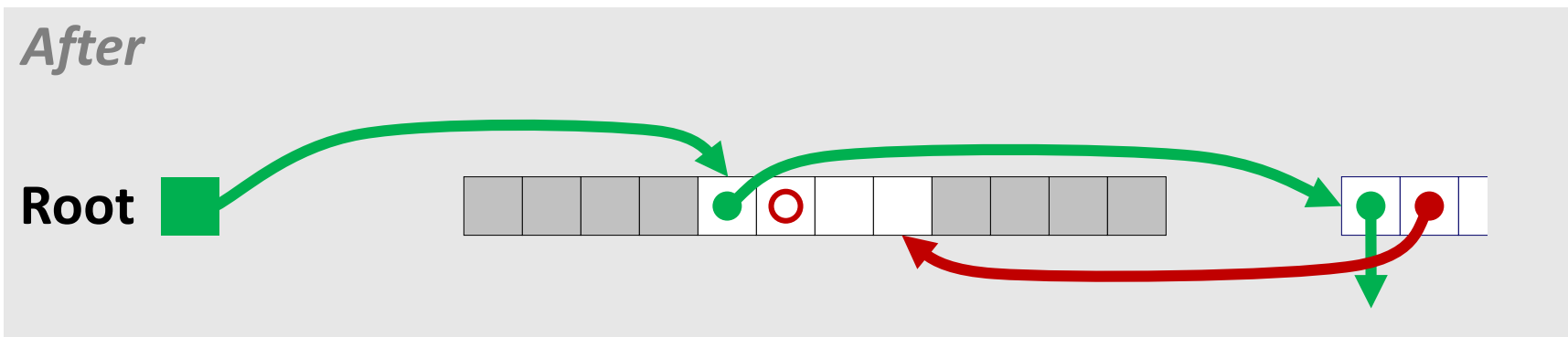
- ***Insertion policy:*** Where in the free list do you put a newly freed block?
 - LIFO (last-in-first-out) policy
 - Insert freed block at the beginning of the free list
 - ***Pro:*** simple and constant time
 - ***Con:*** studies suggest fragmentation is worse than address ordered
 - Address-ordered 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

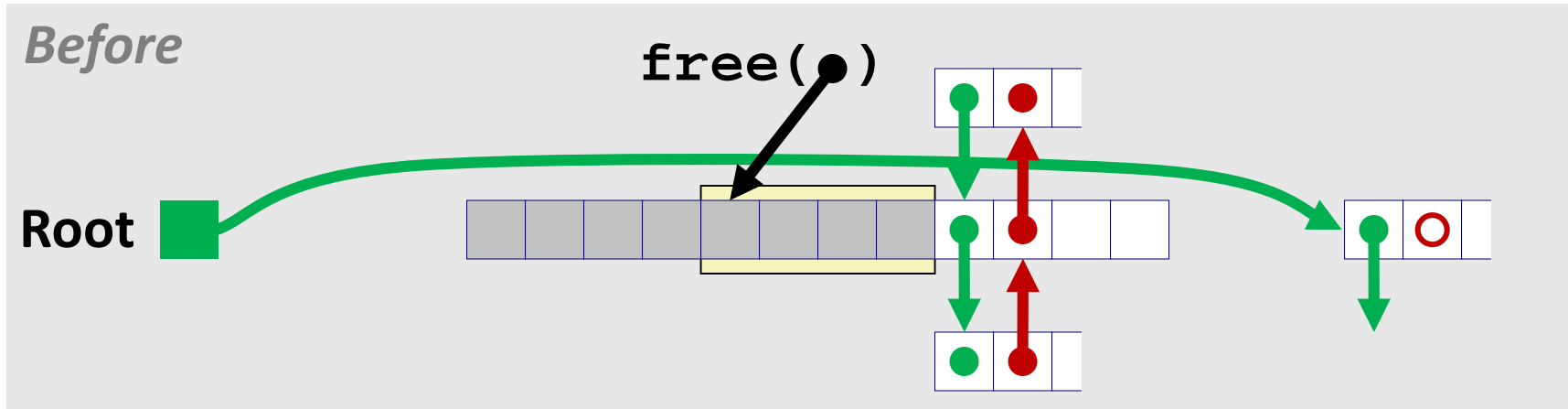


- Insert the freed block at the root of the list

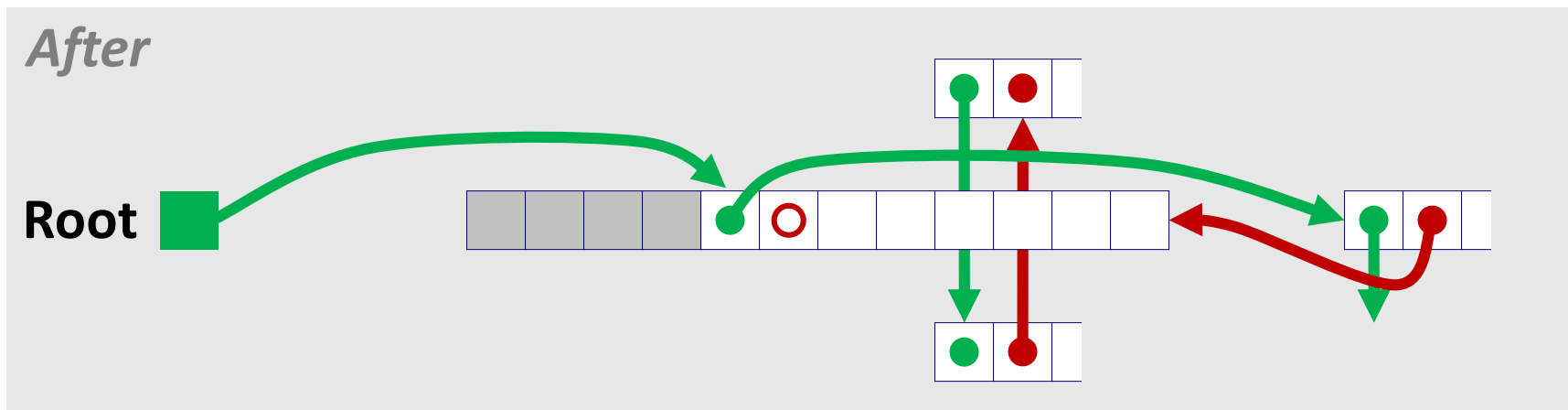


Freeing With a LIFO Policy (Case 3)

conceptual graphic

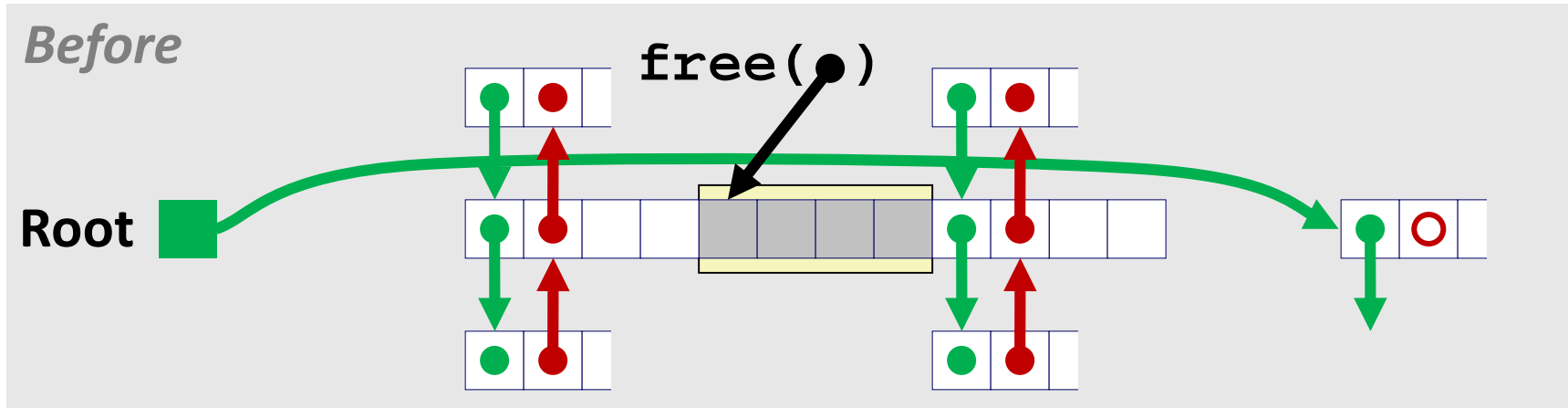


- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list

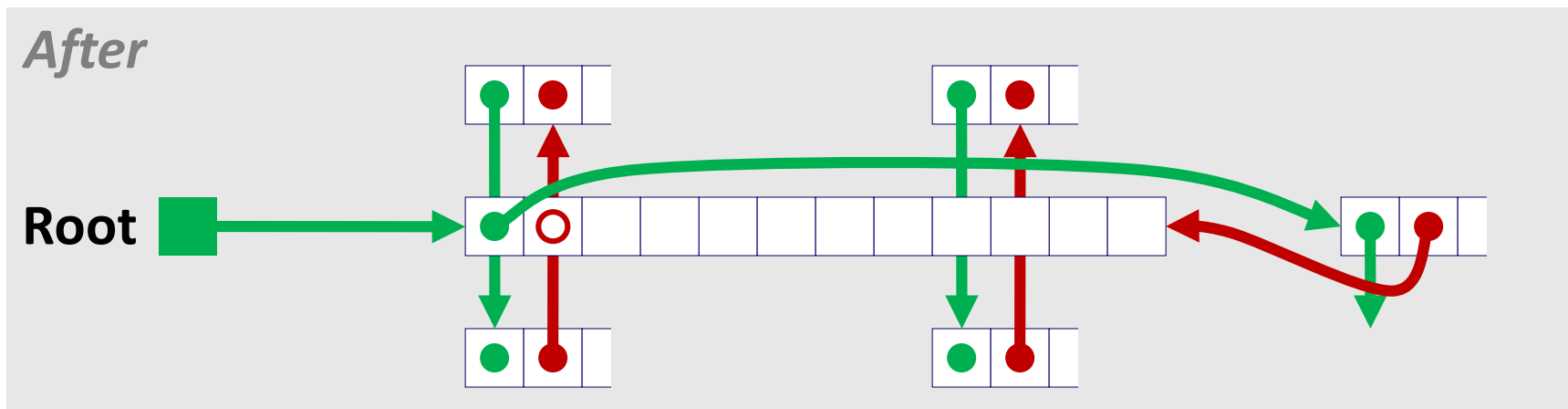


Freeing With a LIFO Policy (Case 4)

conceptual graphic



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



Explicit List Summary

■ Comparison to implicit list:

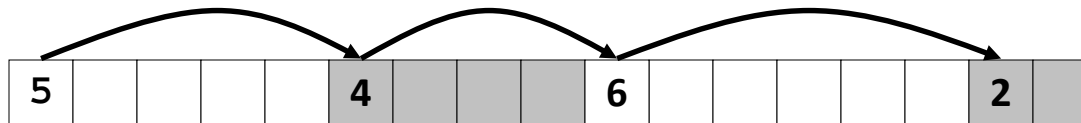
- Allocate is linear time in number of *free* blocks instead of *all* blocks
 - *Much faster* when most of the memory is full
- Slightly more complicated allocate and free since needs to splice blocks in and out of the list
- Some extra space for the links (2 extra words needed for each block)
 - Possibly increases minimum block size, leading to more internal fragmentation

■ Most common use of explicit lists is in conjunction with segregated free lists

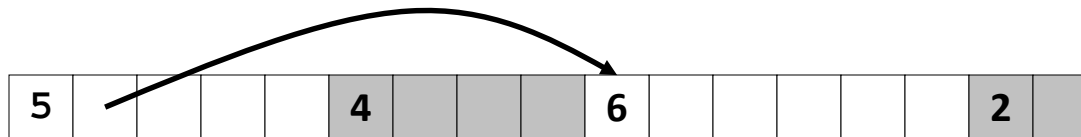
- Keep multiple linked lists of different size classes, or possibly for different types of objects

Keeping Track of Free Blocks

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



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



- Method 3: *Segregated free list*

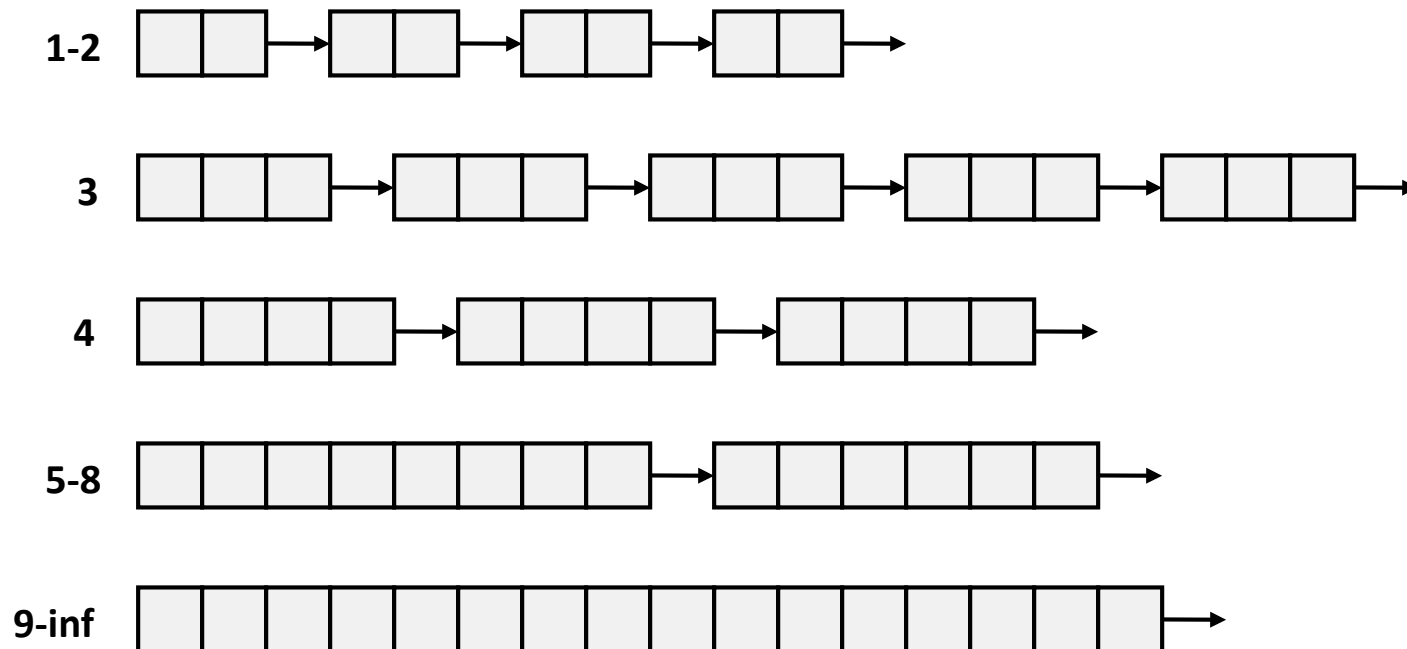
- Different free lists for different size classes

- 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

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 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 largest size class

Seglist Allocator (cont.)

■ 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 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.

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

Section 10: Memory Allocation Topics

■ ~~Dynamic memory allocation~~

- ~~Size/number of data structures may only be known at run time~~
- ~~Need to allocate space on the heap~~
- ~~Need to de-allocate (free) unused memory so it can be re-allocated~~

■ ~~Implementation~~

- ~~Implicit free lists~~
- ~~Explicit free lists — subject of next programming assignment~~
- ~~Segregated free lists~~

■ Garbage collection

■ Common memory-related bugs in C programs

Implicit Memory Allocation: Garbage Collection

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

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

- Common in functional languages, scripting languages, and modern object oriented languages:
 - Lisp, ML, Java, Perl, Mathematica
- Variants (“conservative” garbage collectors) exist for C and C++
 - However, cannot necessarily collect all garbage

Garbage Collection

- **How does the memory allocator know when memory can be freed?**
 - In general, we cannot know what is going to be used in the future since it depends on conditionals
 - But, we can tell that certain blocks cannot be used if there are no pointers to them
- **So the memory allocator needs to know what is a pointer and what is not – how can it do this?**
- **We'll make some assumptions about pointers:**
 - Memory allocator can distinguish pointers from non-pointers
 - All pointers point to the start of a block in the heap
 - Application cannot hide pointers (e.g., by coercing them to an `int`, and then back again)

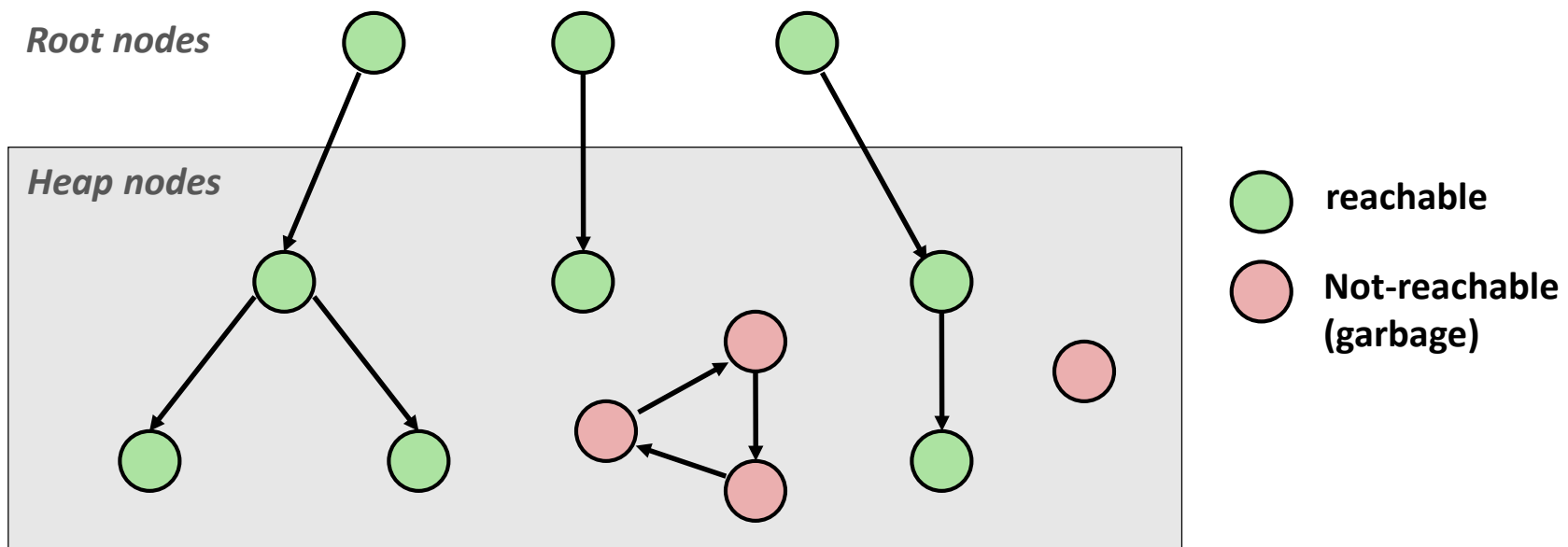
Classical GC Algorithms

- **Mark-and-sweep collection (McCarthy, 1960)**
 - Does not move blocks (unless you also “compact”)
- **Reference counting (Collins, 1960)**
 - Does not move blocks (not discussed)
- **Copying collection (Minsky, 1963)**
 - Moves blocks (not discussed)
- **Generational Collectors (Lieberman and Hewitt, 1983)**
 - Collection based on lifetimes
 - Most allocations become garbage very soon
 - So focus reclamation work on zones of memory recently allocated
- **For more information:**
Jones and Lin, “*Garbage Collection: Algorithms for Automatic Dynamic Memory*”, John Wiley & Sons, 1996.

Memory as a Graph

■ We view memory as a directed graph

- Each allocated heap block is a node in the graph
- Each pointer is an edge in the graph
- Locations not in the heap that contain pointers into the heap are called **root** nodes (e.g. registers, locations on the stack, global variables)

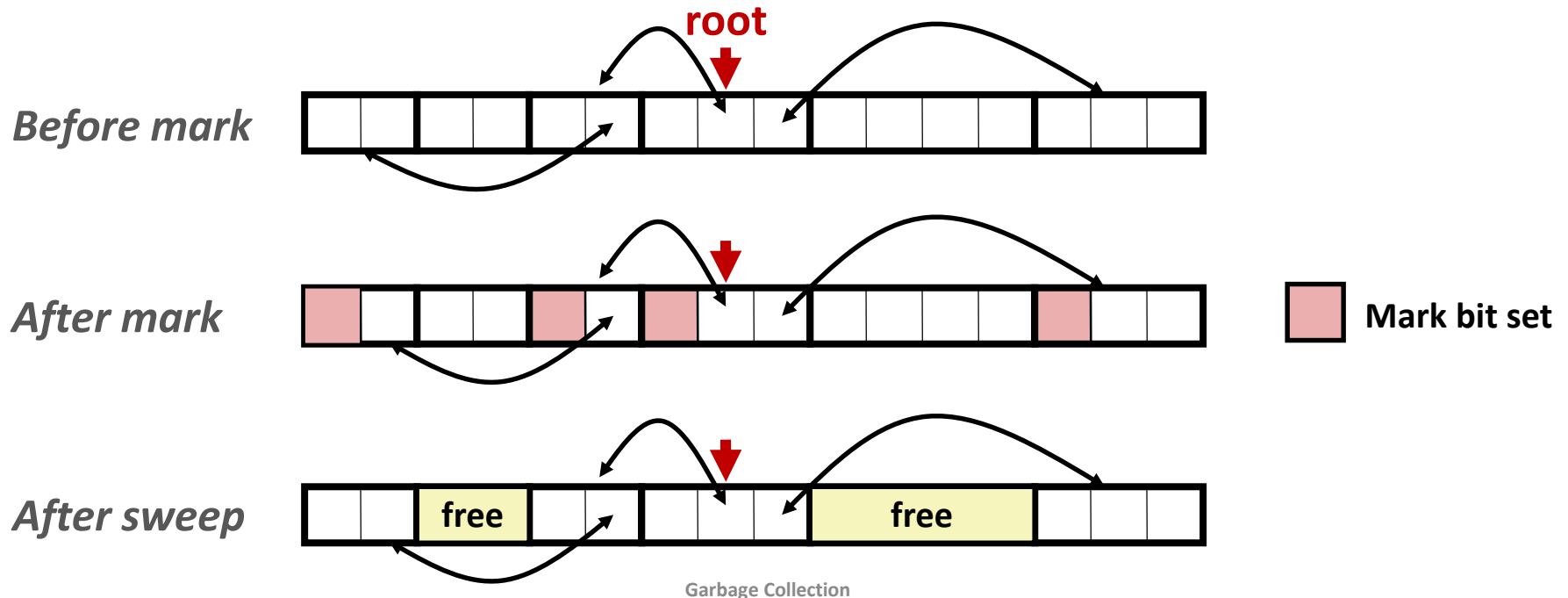


A node (block) is **reachable** if there is a path from any root to that node

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

Mark and Sweep Collecting

- Can build on top of malloc/free package
 - Allocate using malloc until you “run out of space”
- When out of space:
 - Use extra **mark bit** in the head of each block
 - **Mark:** Start at roots and set mark bit on each reachable block
 - **Sweep:** Scan all blocks and free blocks that are not marked



Assumptions For a Simple Implementation

- **Application can use functions such as:**
 - `new(n)` : returns pointer to new block with all locations cleared
 - `read(b, i)` : read location `i` of block `b` into register
 - `b[i]`
 - `write(b, i, v)` : write `v` into location `i` of block `b`
 - `b[i] = v`
- **Each block will have a header word**
 - `b[-1]`
- **Functions used by the garbage collector:**
 - `is_ptr(p)` : determines whether `p` is a pointer to a block
 - `length(p)` : returns length of block pointed to by `p`, not including header
 - `get_roots()` : returns all the roots

Mark and Sweep (cont.)

Mark using depth-first traversal of the memory graph

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

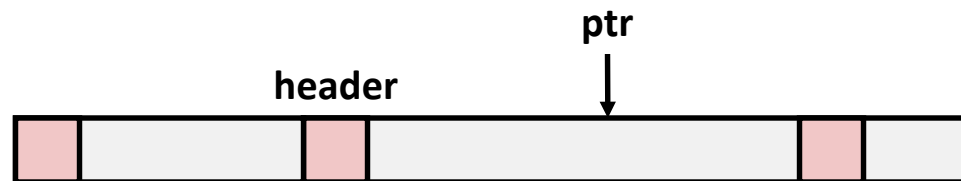
Sweep using lengths to find next block

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

Conservative Mark & Sweep in C

■ Would mark & sweep work in C?

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



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

Section 10: Memory Allocation Topics

■ ~~Dynamic memory allocation~~

- ~~Size/number of data structures may only be known at run time~~
- ~~Need to allocate space on the heap~~
- ~~Need to de-allocate (free) unused memory so it can be re-allocated~~

■ ~~Implementation~~

- ~~Implicit free lists~~
- ~~Explicit free lists – subject of next programming assignment~~
- ~~Segregated free lists~~

■ ~~Garbage collection~~

■ Common memory-related bugs in C programs

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);
```

■ 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

- 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
 - One of your assignments

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 Nonexistent Variables

- Forgetting that local variables disappear when a function returns

```
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>
```

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!

```
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;
};

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