CSC 252: Computer Organization Spring 2018: Lecture 24

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Action Items:
- **Programming Assignment 6 is out**

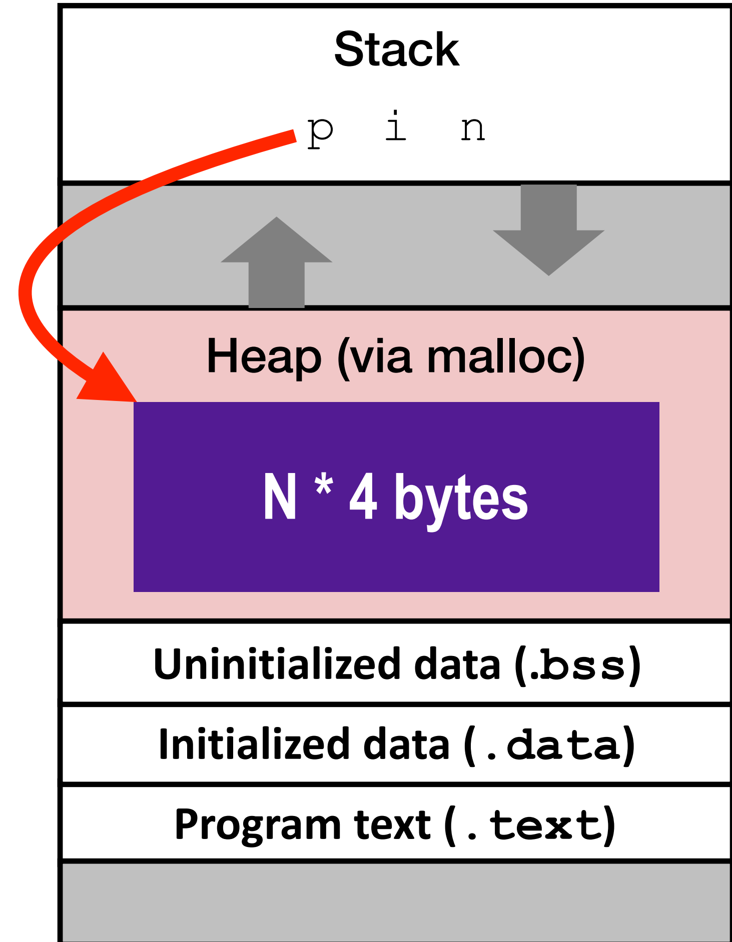
Announcement

- Programming Assignment 6 is out
  - Main assignment: 11:59pm, **Monday, April 30**.
- No office hours today (held on Tuesday)

15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30 **Due**	May 1	2	3	4	5

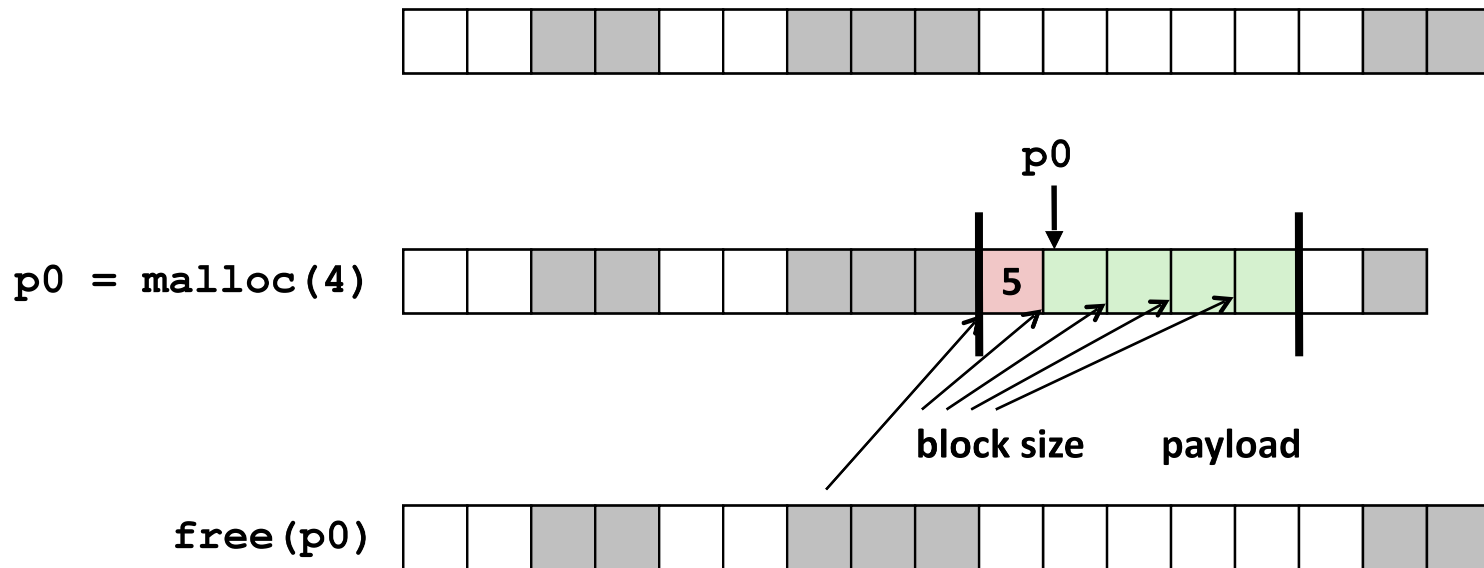
`malloc` Example

```c
#include <stdio.h>
#include <stdlib.h>

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

    /* Allocate a block of n ints */
    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }

    /* Initialize allocated block */
    for (i=0; i<n; i++)
        p[i] = i;


    /* Return allocated block to the heap */
    free(p);
}
```



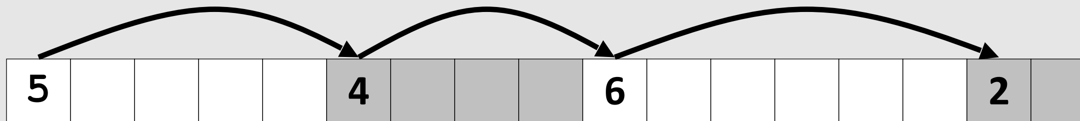
Knowing How Much to Free

- 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

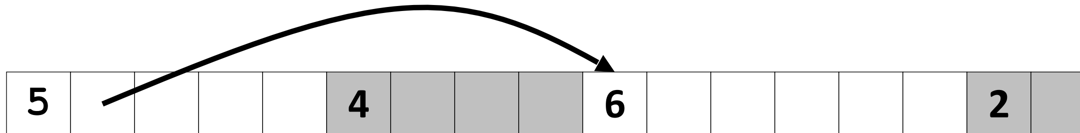


Keeping Track of Free Blocks

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



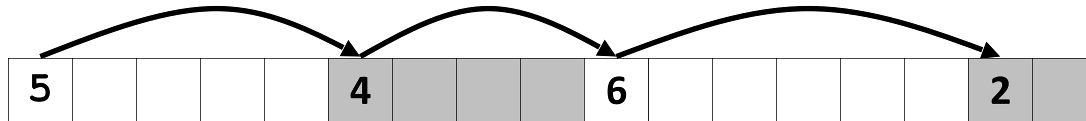
- Method 2: *Explicit list* among the free blocks using pointers



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

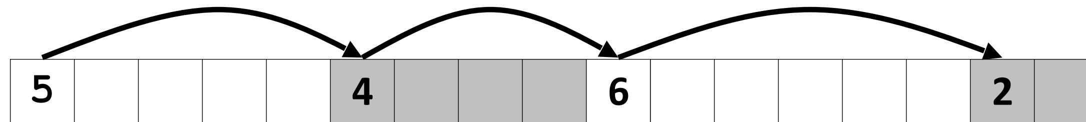
Implicit List

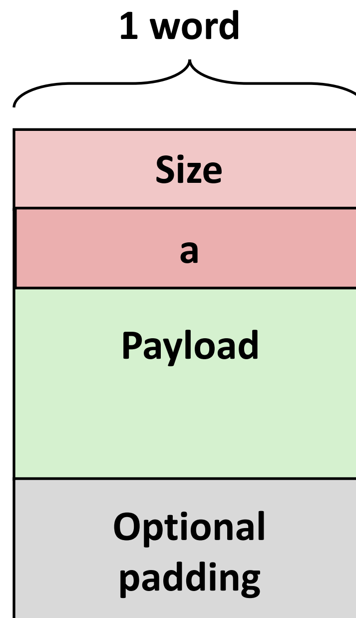
- For each block we need both size and allocation status
  - Could store this information in two words: wasteful!



Implicit List

- For each block we need both size and allocation status
  - Could store this information in two words: wasteful!





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

Size: block size

Payload: application data (allocated blocks only)

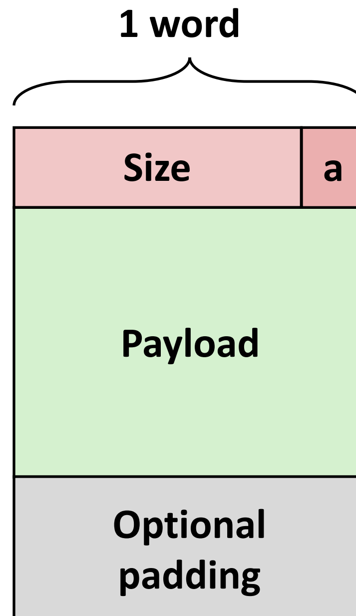
Implicit List

- For each block we need both size and allocation status
  - Could store this information in two words: wasteful!
- Standard trick
  - If blocks are aligned, some low-order address bits are always 0
  - Instead of storing an always-0 bit, use it as a allocated/free flag
  - When reading size word, must mask out this bit

Implicit List

- For each block we need both size and allocation status
  - Could store this information in two words: wasteful!
- Standard trick
  - If blocks are aligned, some low-order address bits are always 0
  - Instead of storing an always-0 bit, use it as a allocated/free flag
  - When reading size word, must mask out this bit

Format of allocated and free blocks

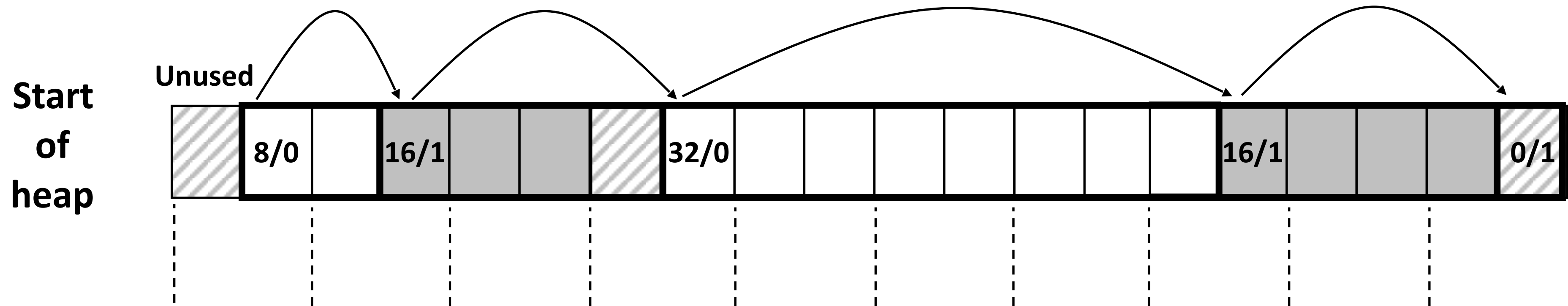


a = 1: Allocated block
a = 0: Free block

Size: block size

Payload: application data (allocated blocks only)

Detailed Implicit Free List Example



Double-word aligned

Allocated blocks: shaded
Free blocks: unshaded
Headers: labeled with size in bytes/allocated bit

Finding a Free Block

- First fit:
  - Search list from beginning, choose first free block that fits
  - Can take linear time in total number of blocks (allocated and free)
  - In practice it can cause “splinters” at beginning of list

Finding a Free Block

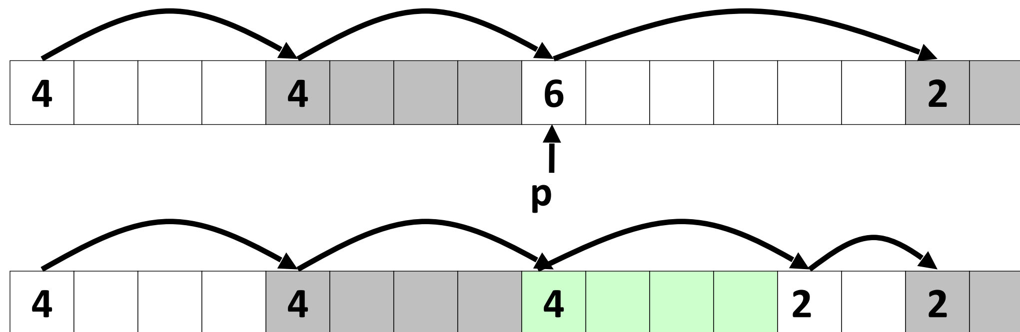
- First fit:
  - Search list from beginning, choose first free block that fits
  - Can take linear time in total number of blocks (allocated and free)
  - In practice it can cause “splinters” at beginning of list
- Next fit:
  - Like first fit, but search list starting where previous search finished
  - Should often be faster than first fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse

Finding a Free Block

- First fit:
  - Search list from beginning, choose first free block that fits
  - Can take linear time 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 improves memory utilization
  - Will typically run slower than first fit

Allocating in Free Block

- Allocated space might be smaller than free space
- We could simply leave the extra space there. Simple to implement but causes internal fragmentation
- Or we could split the block



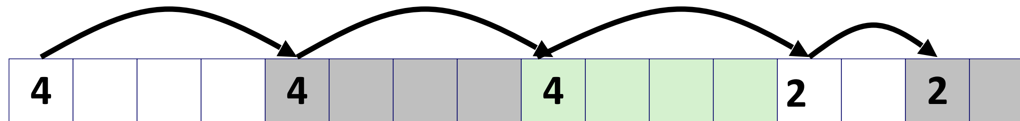

```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;  // round up to even
  int oldsize = *p & -2;                // mask out low bit
  *p = newsize | 1;                     // set new length
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;   // set length in remaining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

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"

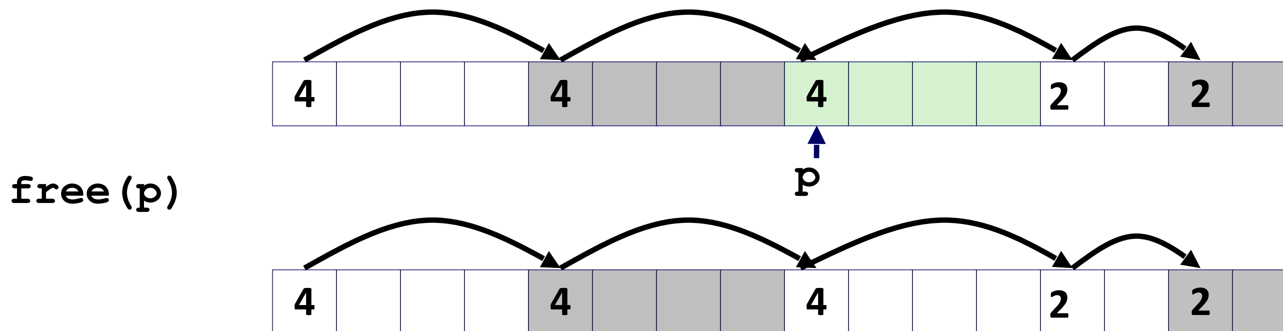


Freeing a Block

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

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

  - But can lead to “false fragmentation”

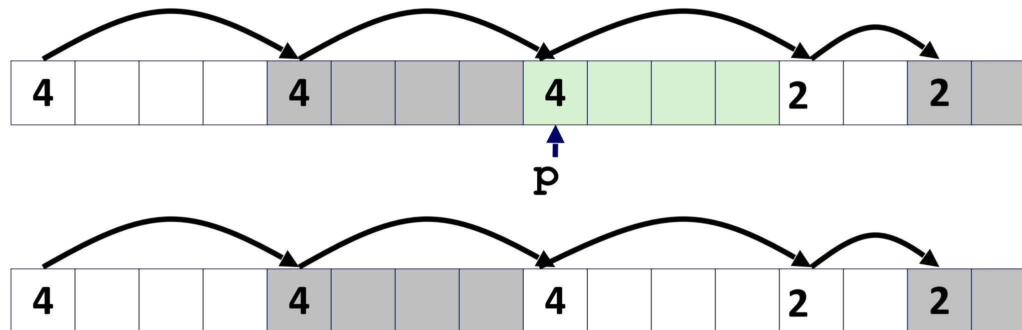


Freeing a Block

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

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

  - But can lead to “false fragmentation”



free(p)

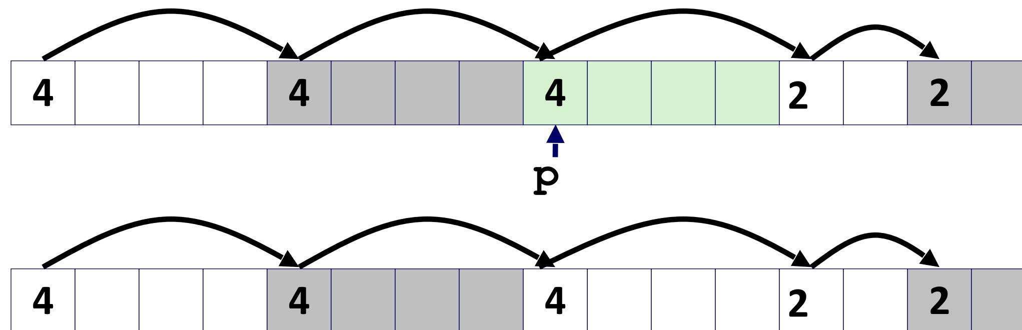
malloc(5) ***Oops!***

Freeing a Block

- Simplest implementation:
  - Need only clear the "allocated" flag

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

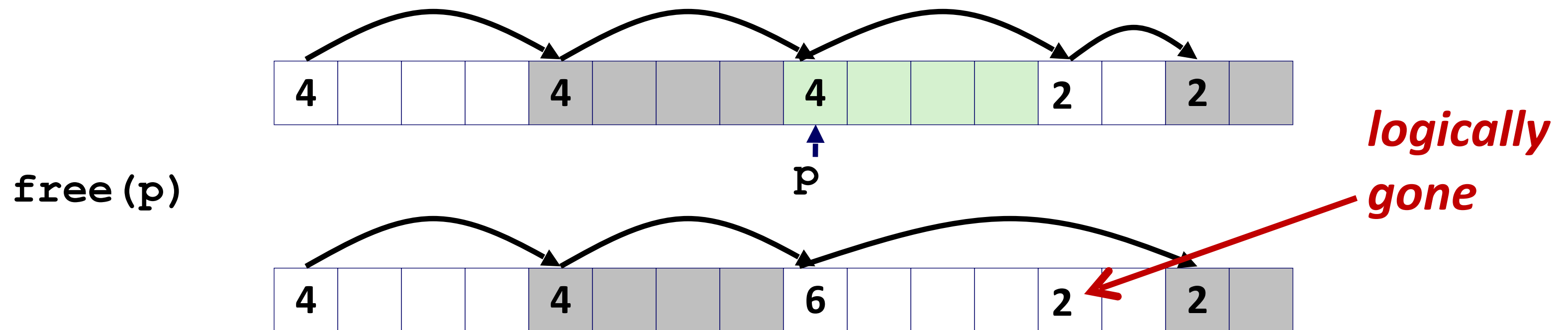
  - But can lead to "false fragmentation"



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

Coalescing

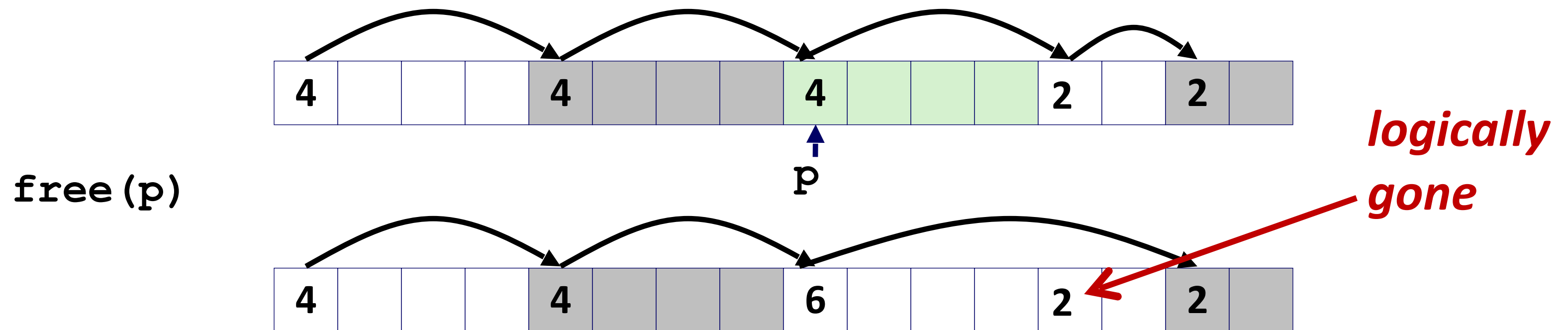
- Join *(coalesce)* with next/previous blocks, if they are free
  - Coalescing with next block




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

Coalescing

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



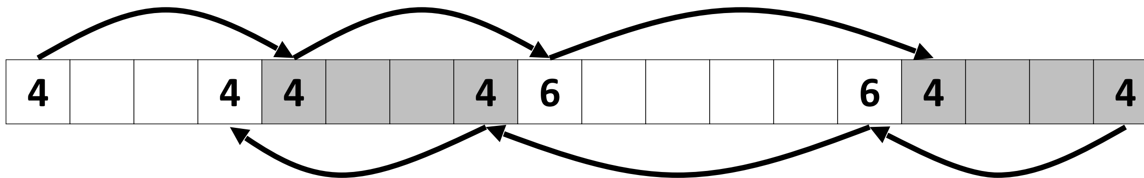

```
void free_block(ptr p) {
    *p = *p & -2;            // clear allocated flag
    next = p + *p;           // find next block
    if ((*next & 1) == 0)
      *p = *p + *next;       // add to this block if
}                            //    not allocated
```

But how do we coalesce with previous block?

- Linear time solution: scans from beginning

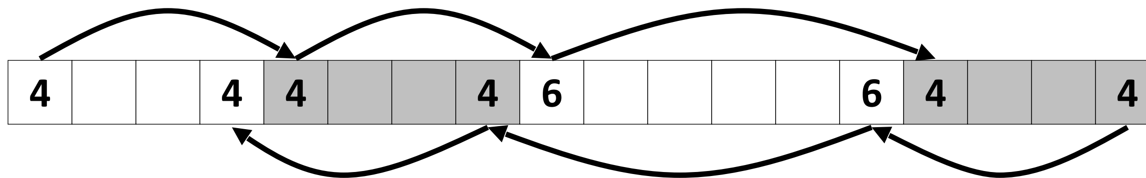
Bidirectional Coalescing (Constant Time)

- *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!

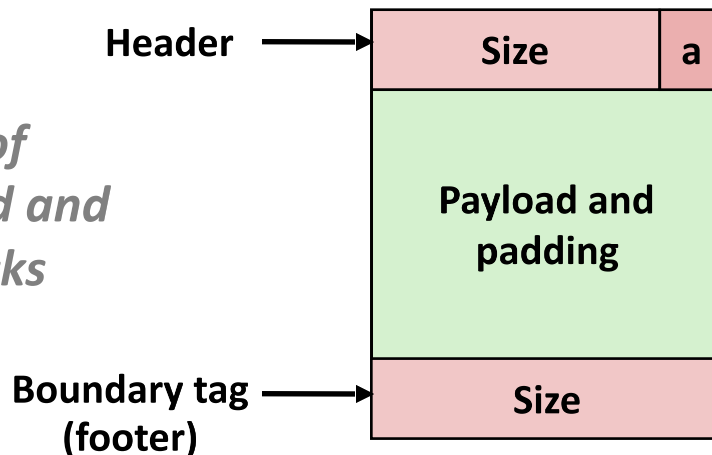


Bidirectional Coalescing (Constant Time)

- *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



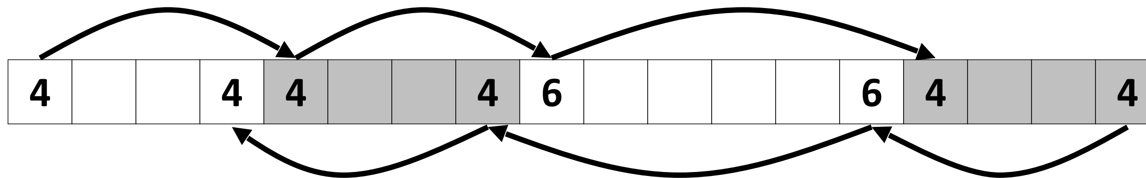
a = 1: Allocated block
a = 0: Free block

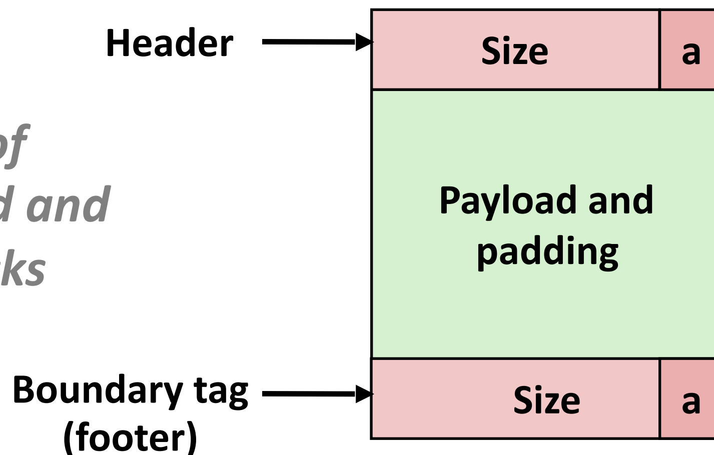
Size: Total block size

Payload: Application data (allocated blocks only)

Bidirectional Coalescing (Constant Time)

- *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

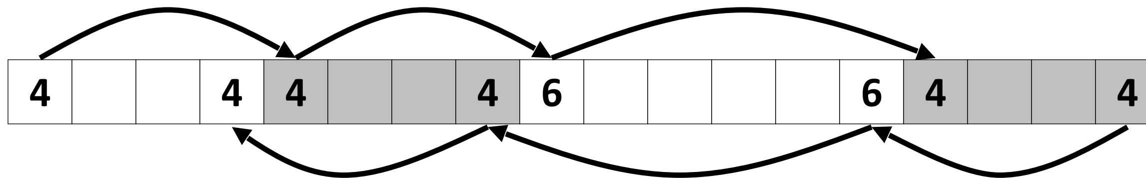
a = 1: Allocated block
a = 0: Free block

Size: Total block size

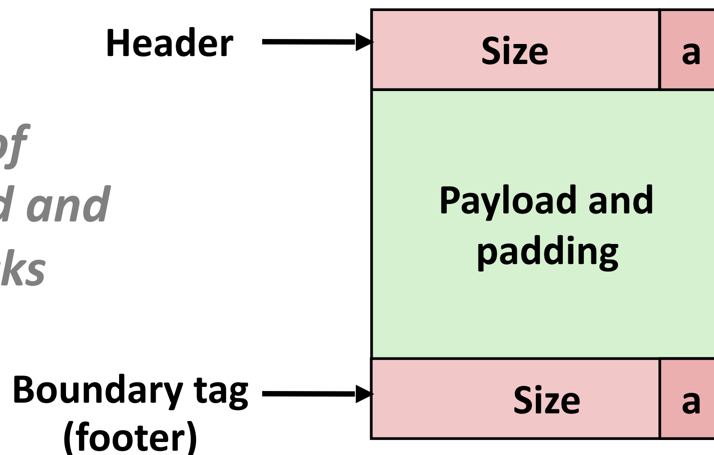
Payload: Application data (allocated blocks only)

Bidirectional Coalescing (Constant Time)

- *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!
- Disadvantages? (Think of small blocks…)



Format of allocated and free blocks



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

Size: Total block size

Payload: Application data (allocated blocks only)

Summary of Key Allocator Policies

- Placement policy:
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation

Summary of Key Allocator Policies

- Placement policy:
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
- Splitting policy:
  - When do we split free blocks?
  - How much internal fragmentation are we willing to tolerate?

Summary of Key Allocator Policies

- Placement policy:
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
- Splitting policy:
  - When do we 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

Implicit Lists: Summary

- Implementation: very simple

Implicit Lists: Summary

- Implementation: very simple
- Allocate cost:
  - linear time worst case
  - Identify free blocks requires scanning *all* the blocks!

Implicit Lists: Summary

- Implementation: very simple
- Allocate cost:
  - linear time worst case
  - Identify free blocks requires scanning *all* the blocks!
- Free cost:
  - constant time worst case

Implicit Lists: Summary

- Implementation: very simple
- Allocate cost:
  - linear time worst case
  - Identify free blocks requires scanning *all* the blocks!
- Free cost:
  - constant time worst case
- Memory usage:
  - Will depend on placement policy
  - First-fit, next-fit, or best-fit

Implicit Lists: Summary

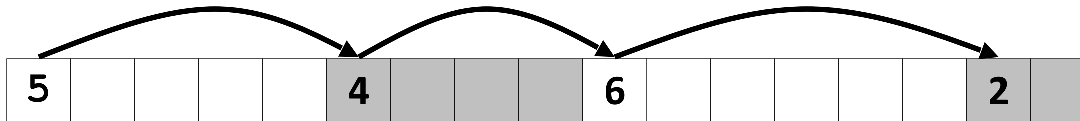
- Implementation: very simple
- Allocate cost:
  - linear time worst case
  - Identify free blocks requires scanning *all* the blocks!
- Free cost:
  - constant time worst case
- Memory usage:
  - Will depend on placement policy
  - First-fit, next-fit, or best-fit
- Not used in practice because of linear-time allocation
  - used in many special purpose applications

Implicit Lists: Summary

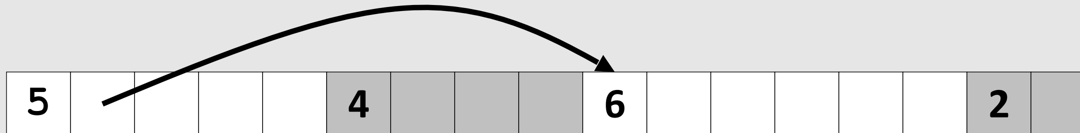
- Implementation: very simple
- Allocate cost:
  - linear time worst case
  - Identify free blocks requires scanning *all* the blocks!
- Free cost:
  - constant time worst case
- Memory usage:
  - Will depend on placement policy
  - First-fit, next-fit, or best-fit
- Not used in practice because of linear-time allocation
  - used in many special purpose applications
- However, the concepts of splitting and boundary tag coalescing are general to *all* allocators

Keeping Track of Free Blocks

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



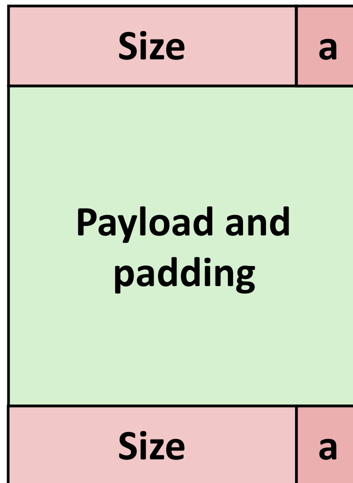
- Method 2: *Explicit list* among the free blocks using pointers



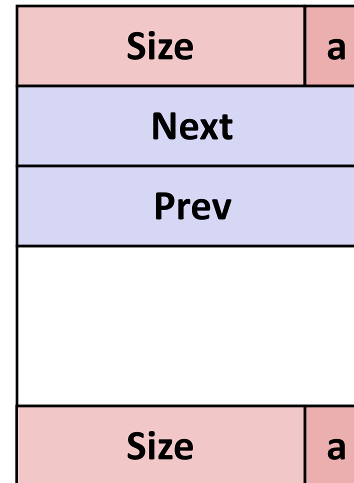
- Method 3: *Segregated free list*
  - Different free lists for different size classes

Explicit Free Lists

Allocated (as before)



Free

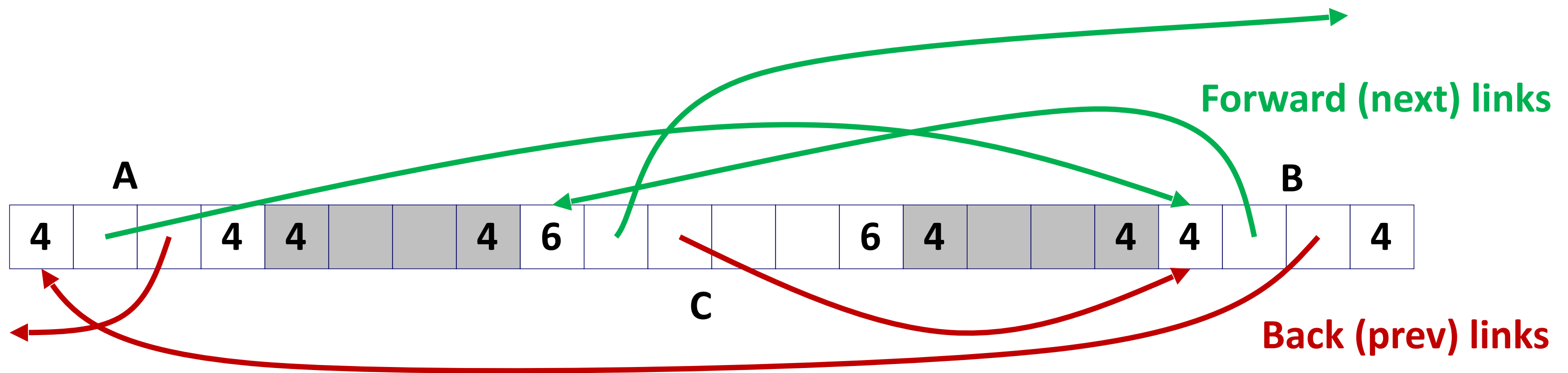


- Maintain list(s) of *free* blocks, not *all* blocks
  - The "next" free block could be anywhere
    - So we need to store forward/back pointers, not just sizes
  - Still need boundary tags for coalescing
  - Luckily we track only free blocks, so we can use payload area

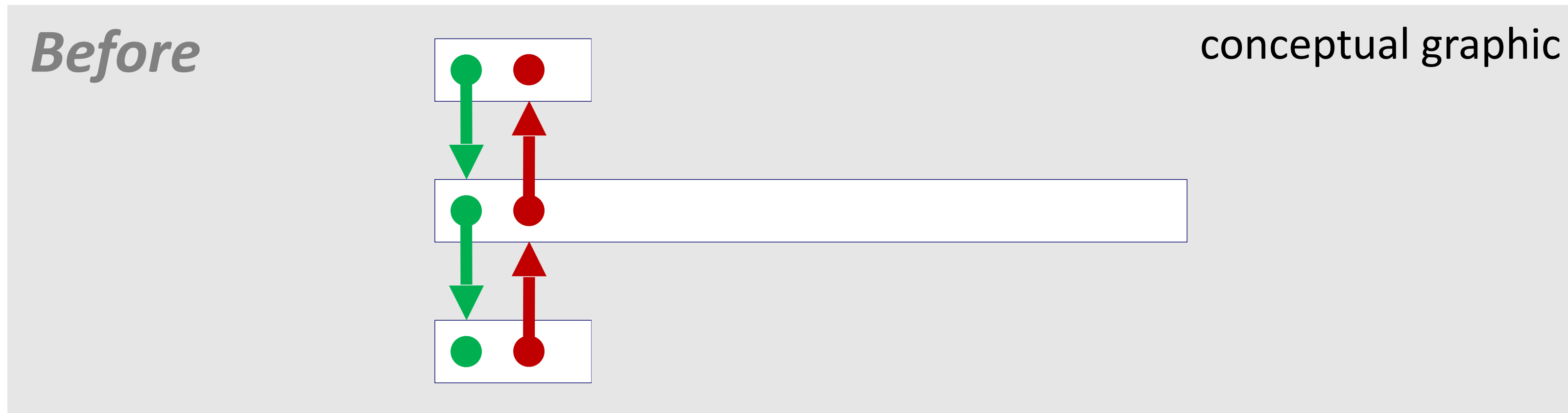
Explicit Free Lists

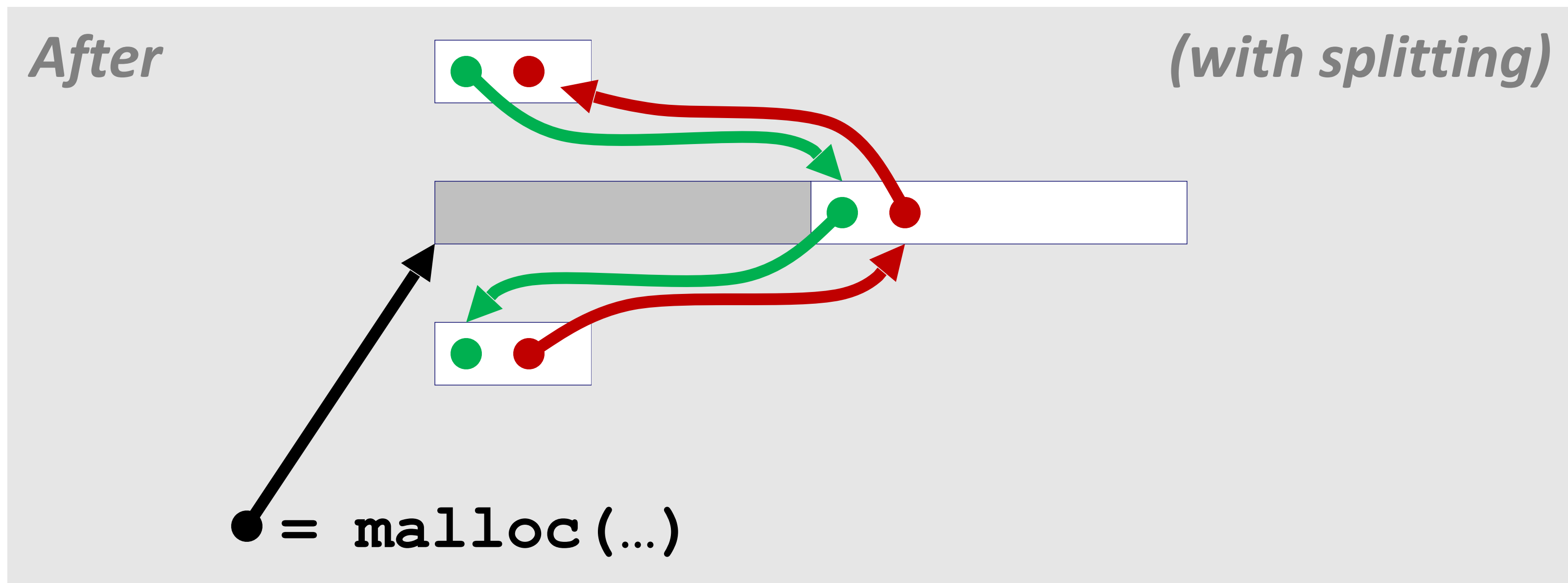
- Logically:

- Physically: blocks can be in any order



Allocating From Explicit Free Lists





Freeing With Explicit Free Lists

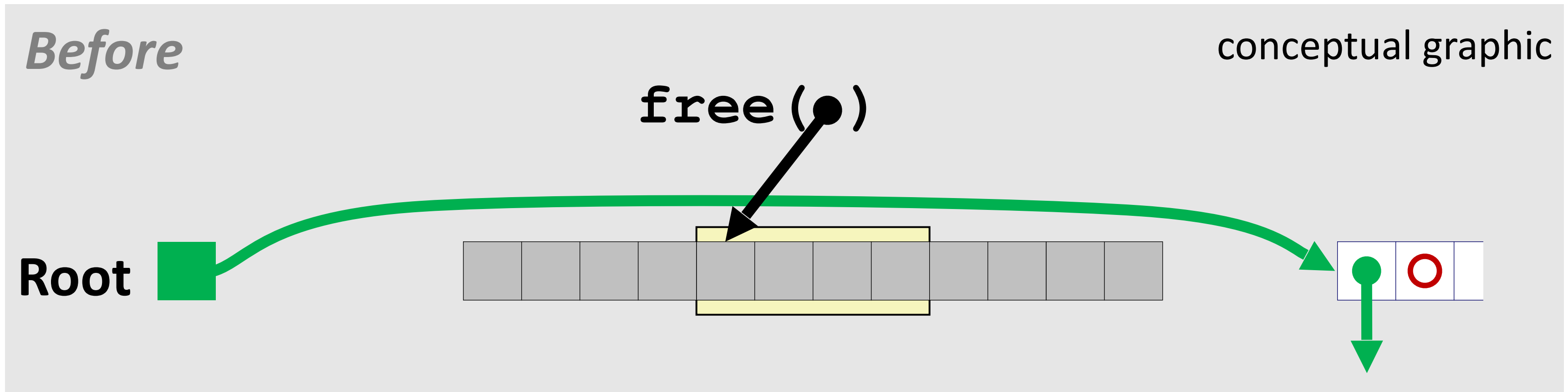
- *Insertion policy*: Where in the free list do you put a newly freed block?
- LIFO (last-in-first-out) policy
  - Insert freed block at the beginning of the free list
  - *Pro:* simple and constant time
  - *Con:* studies suggest fragmentation is worse than address ordered

- Address-ordered policy
  - Insert freed blocks so that free list blocks are always in address order:

    addr(prev) < addr(curr) < addr(next)
  - ***Con:*** requires search
  - ***Pro:*** studies suggest fragmentation is lower than LIFO

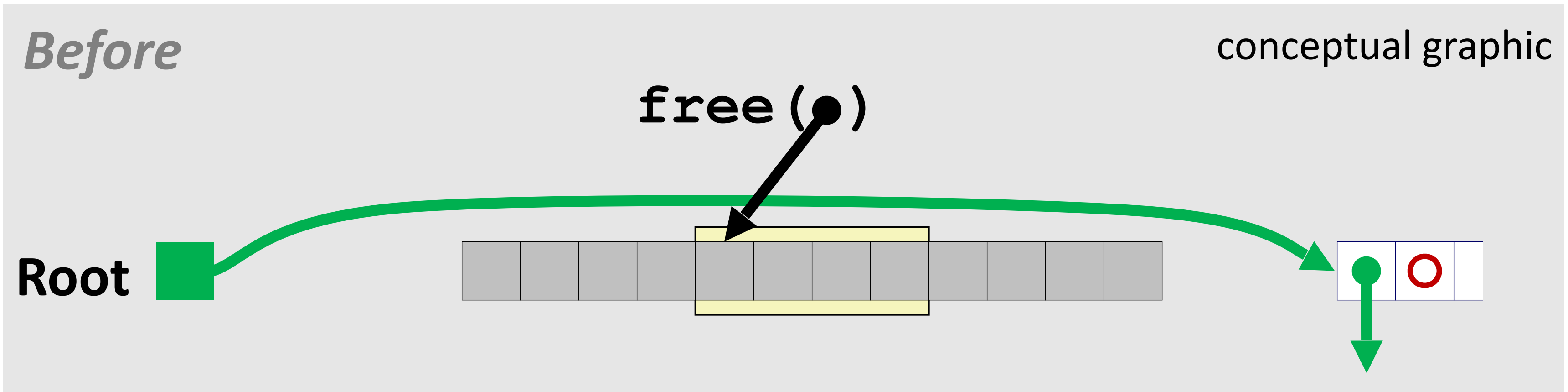
Freeing With a LIFO Policy (Case 1)

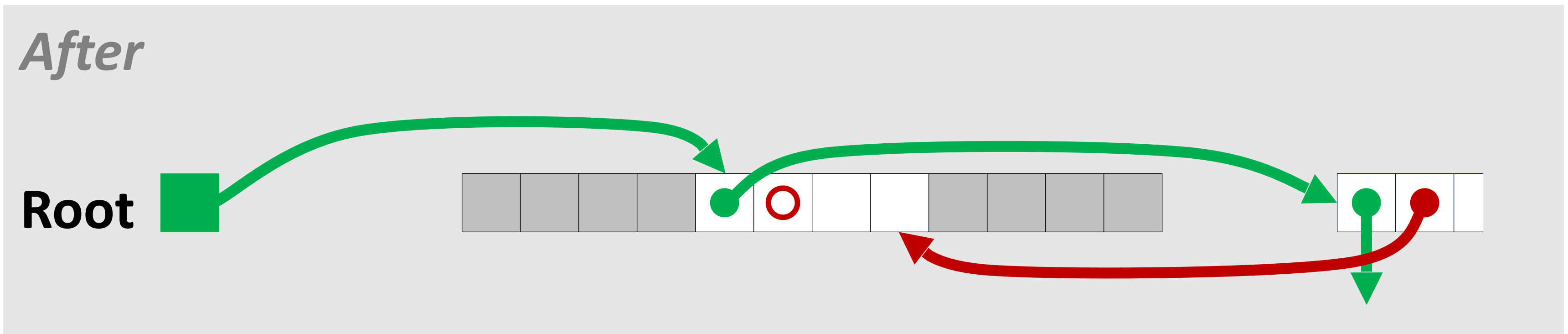
- Insert the freed block at the root of the list



Freeing With a LIFO Policy (Case 1)

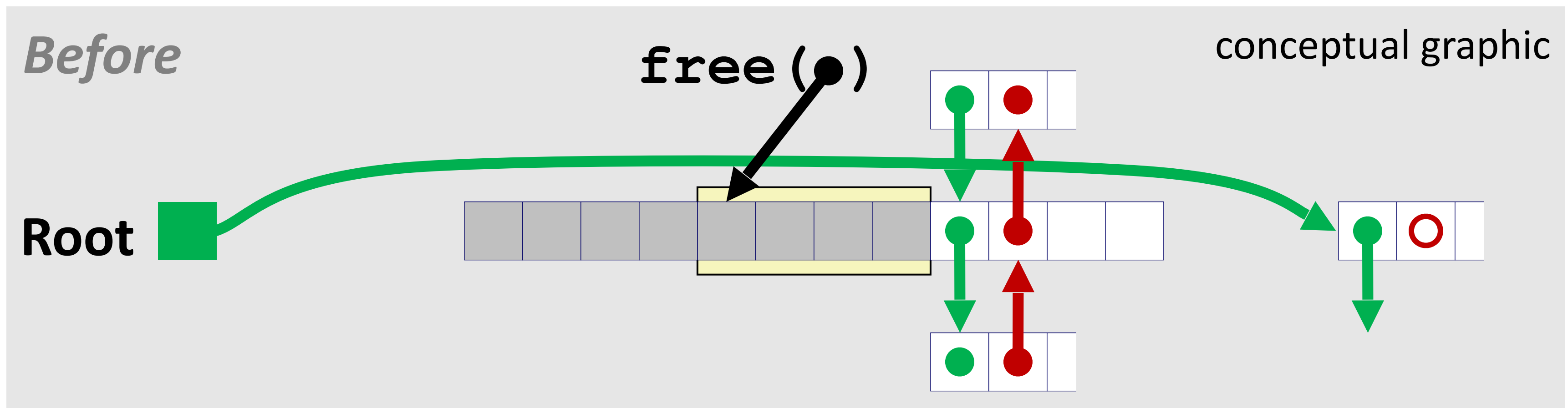
- Insert the freed block at the root of the list





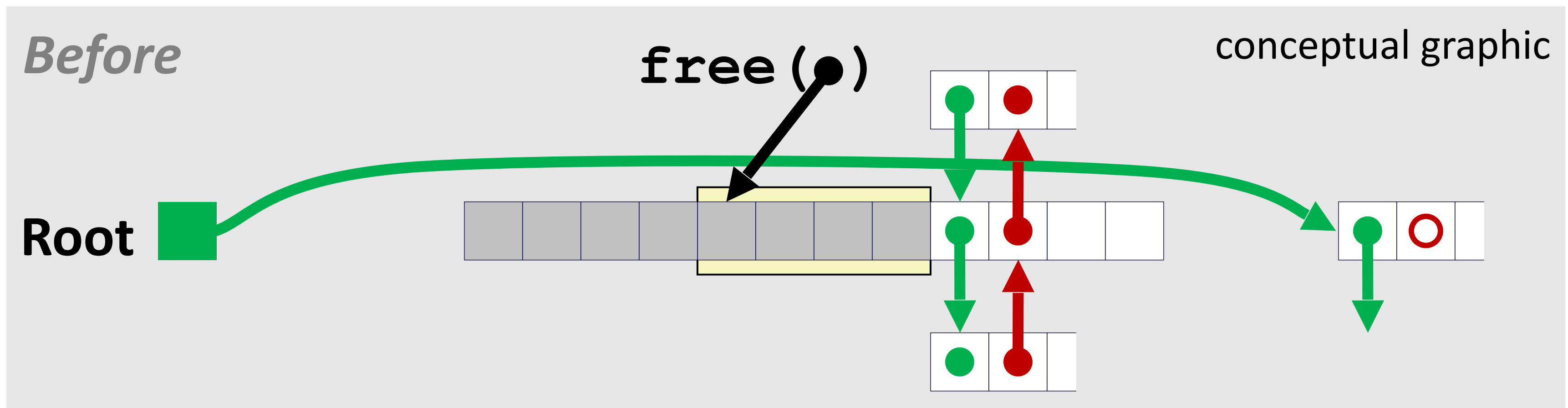
Freeing With a LIFO Policy (Case 2)

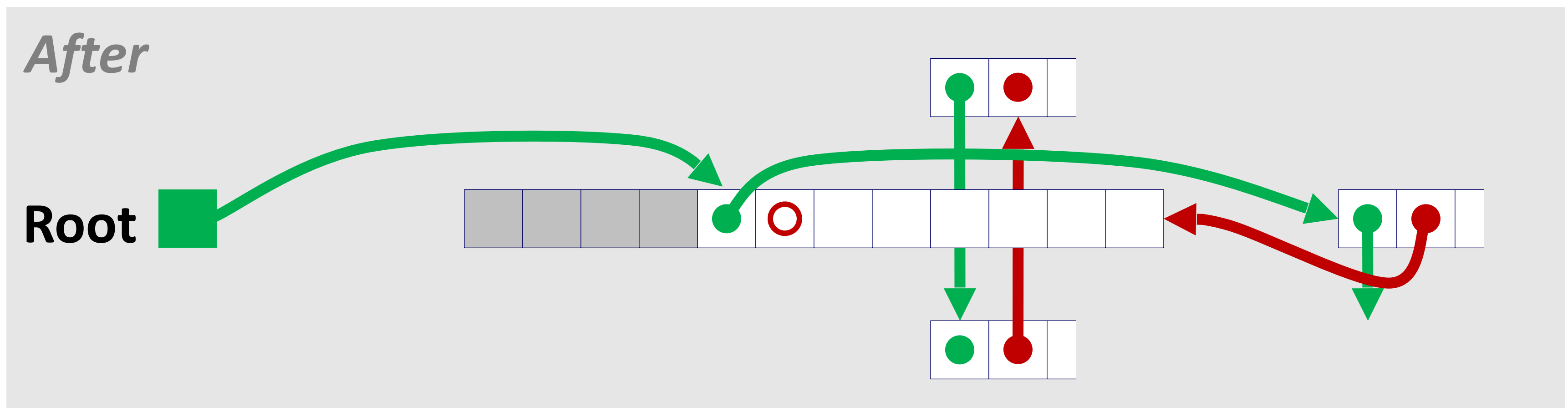
- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list



Freeing With a LIFO Policy (Case 2)

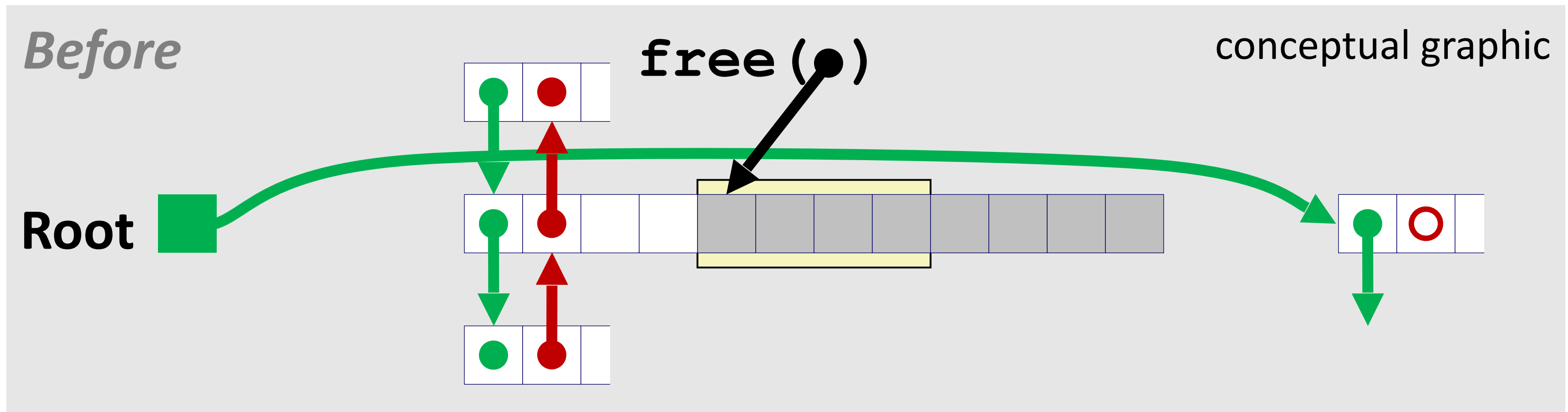
- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list





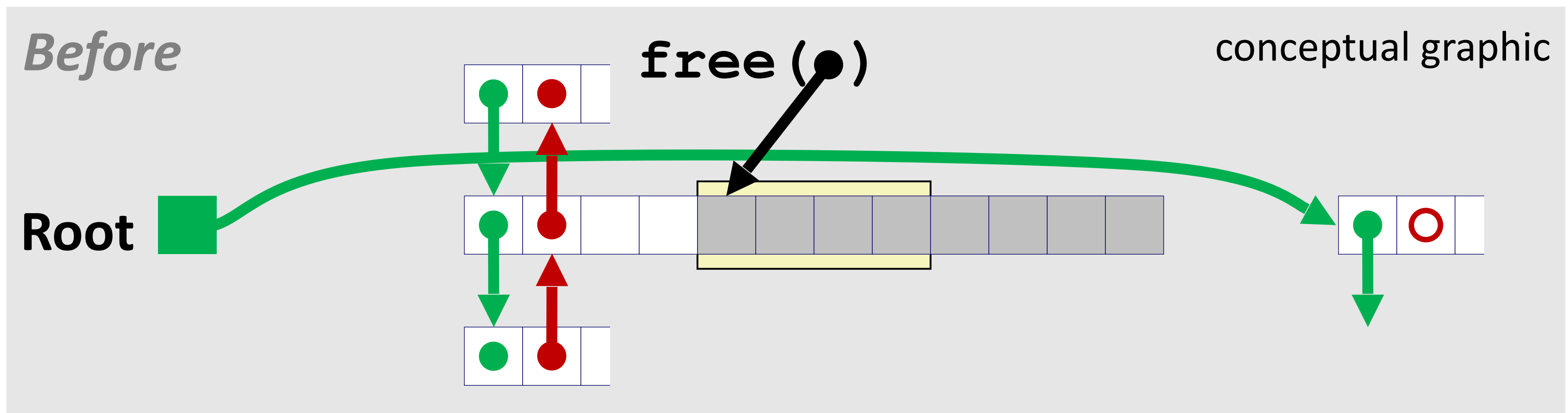
Freeing With a LIFO Policy (Case 3)

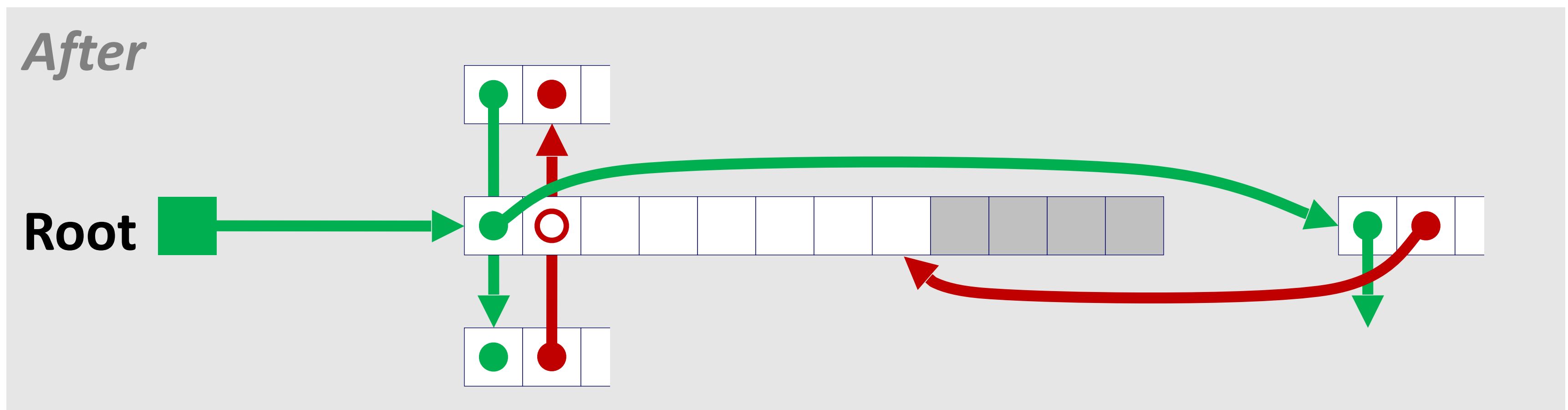
- Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list



Freeing With a LIFO Policy (Case 3)

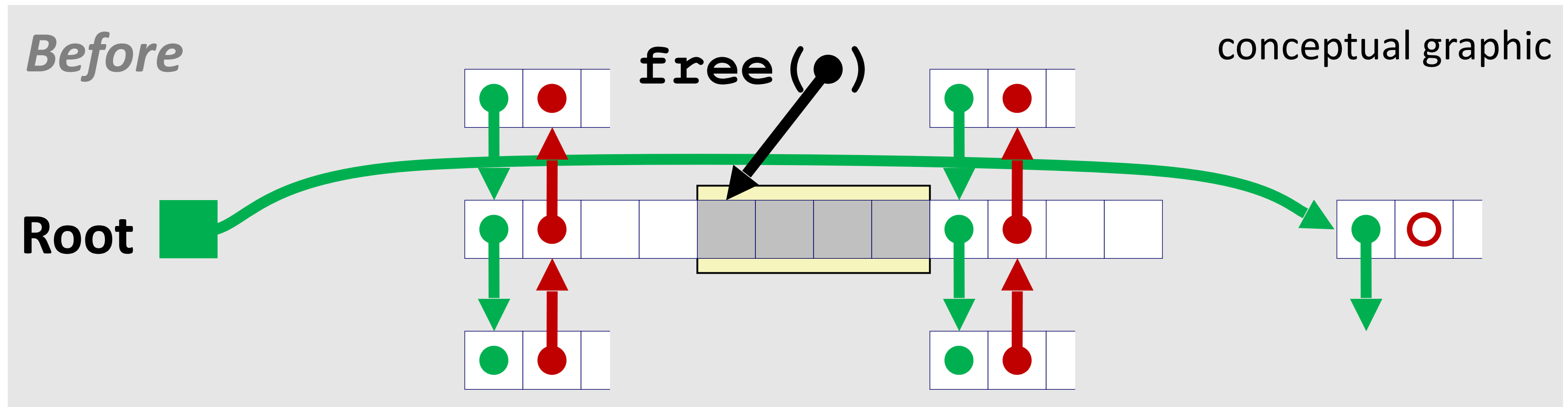
- Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list





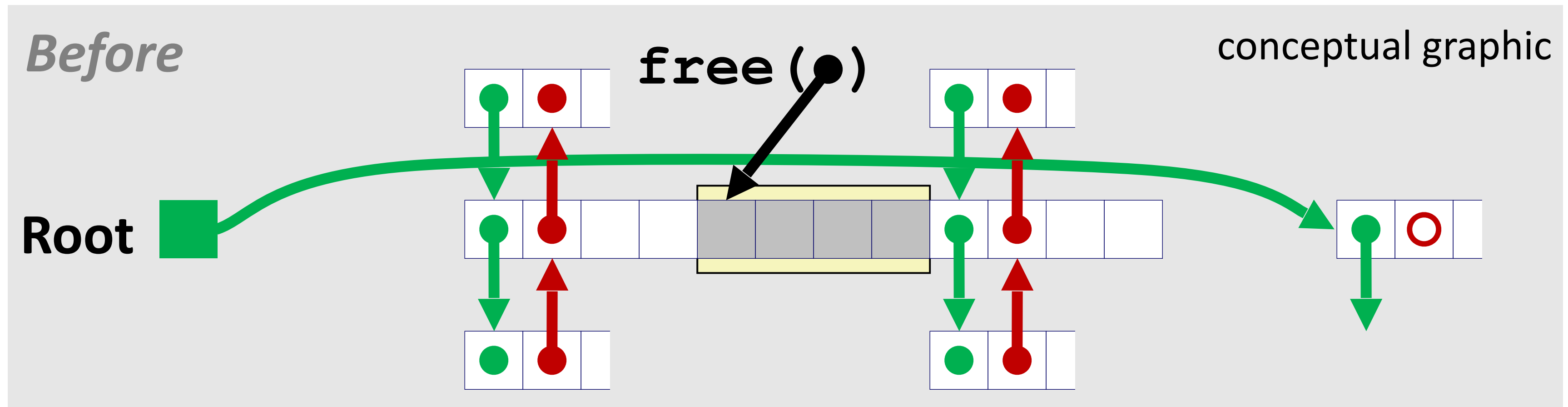
Freeing With a LIFO Policy (Case 4)

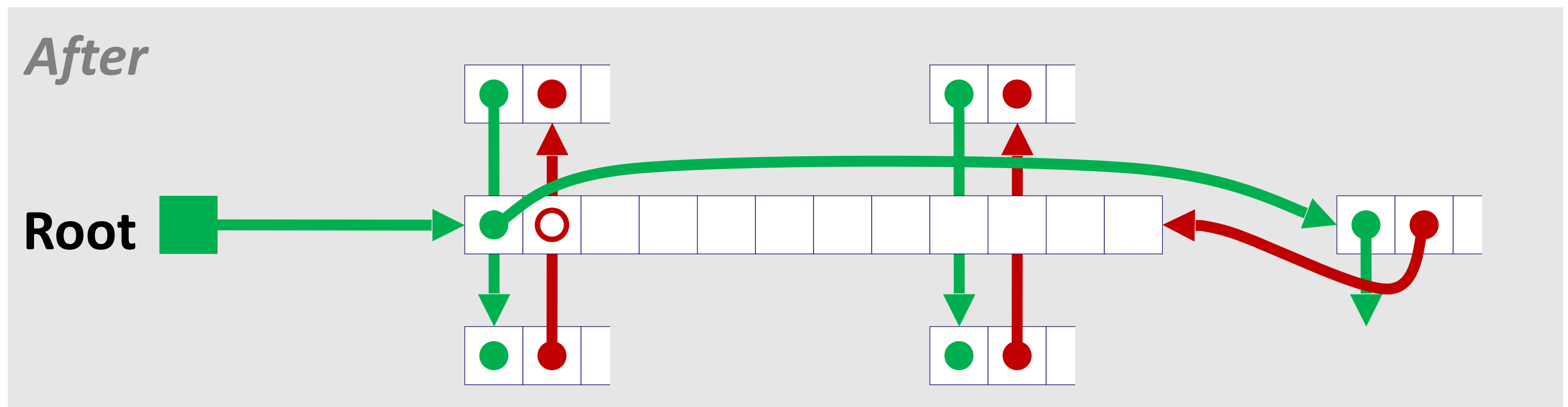
- Splice out predecessor and successor blocks, coalesce all 3 memory blocks and insert the new block at the root of the list



Freeing With a LIFO Policy (Case 4)

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



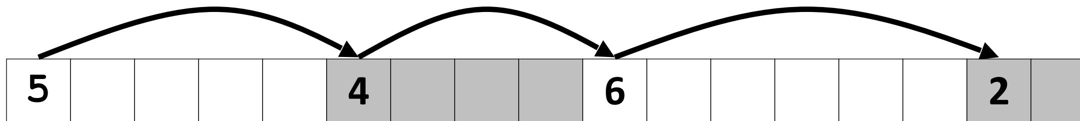


Explicit List Summary

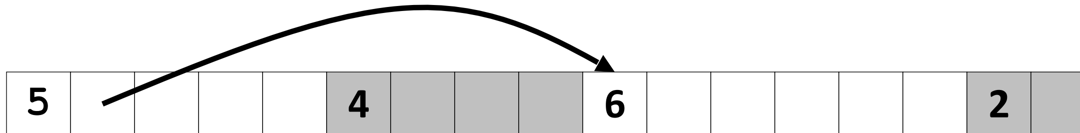
- Comparison to implicit list:
  - Allocate is linear time in number of free blocks instead of all blocks. Much faster when most of the memory is full.
  - Slightly more complicated allocate and free since needs to splice blocks in and out of the list
  - Some extra space for the links (2 extra words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

Keeping Track of Free Blocks

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



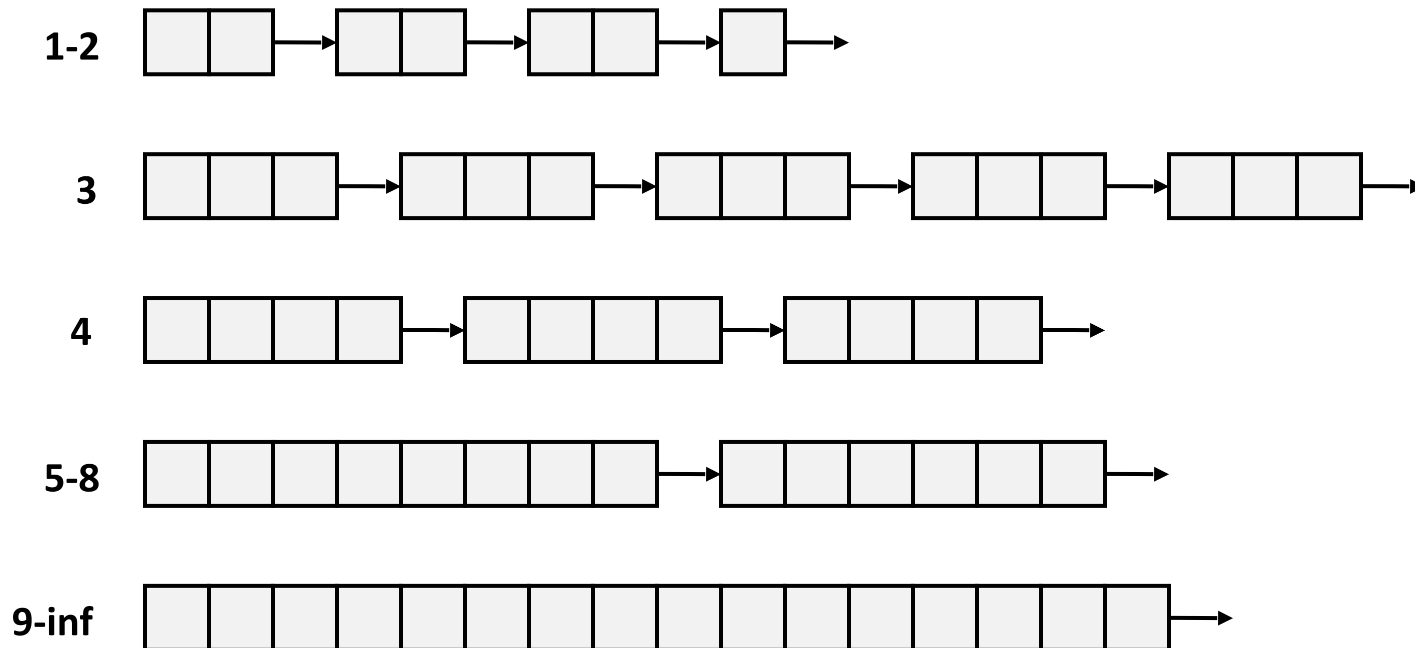
- Method 2: *Explicit list* among the free blocks using pointers



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

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

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

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()`**)
  - Remember heap is in VM, so request heap memory in pages
  - Allocate block of *n* bytes from this new memory
  - Place remainder as a single free block in largest size class.

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()`**)
  - Remember heap is in VM, so request heap memory in pages
  - Allocate block of *n* bytes from this new memory
  - Place remainder as a single free block in largest size class.
- To free a block:
  - Coalesce and place on appropriate list

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.

Implicit Memory Management: Garbage Collection

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

```
void foo() {
   int *p = malloc(128);
   p = malloc(32);
   return; /* both blocks are now garbage */
}
```

- Common in many dynamic languages:
  - Python, Ruby, Java, JavaScript, Perl, ML, Lisp, Mathematica

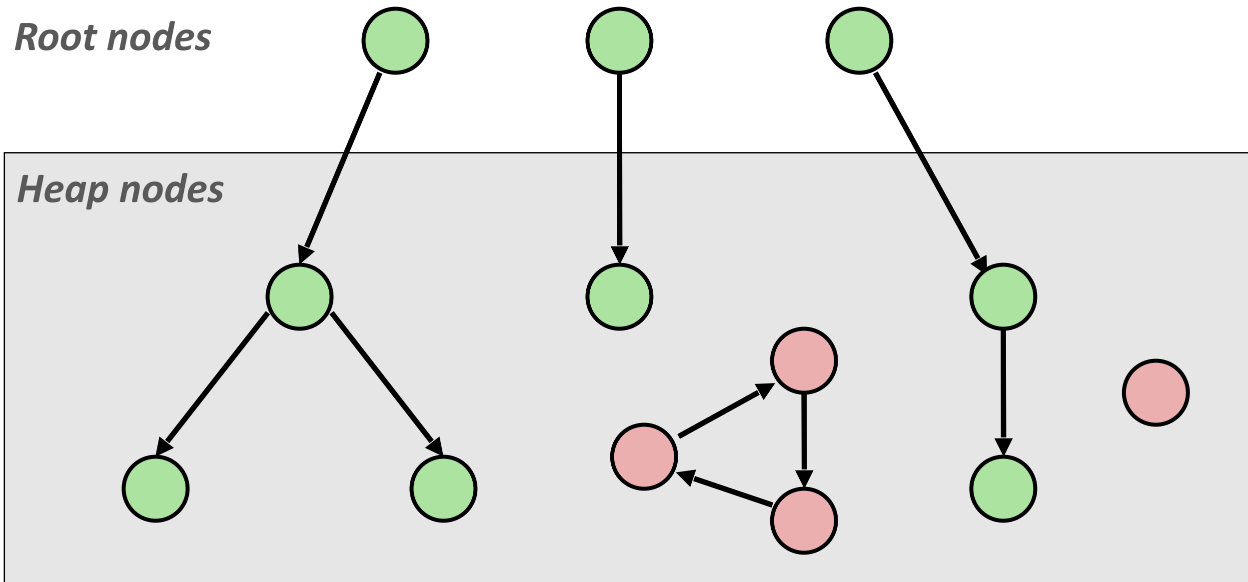
- Variants (“conservative” garbage collectors) exist for C and C++
  - However, cannot necessarily collect all garbage

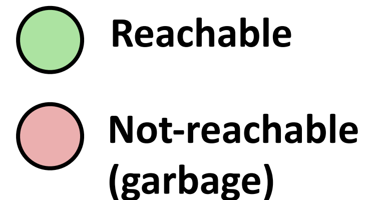
Garbage Collection

- How does the memory manager know when certain memory blocks can be freed?
  - In general we cannot know what is going to be used in the future since it depends on program's future behaviors
  - But we can tell that certain blocks cannot possibly be used if *there are no pointers to them*
  - Garbage collection is essentially to obtain all **reachable** blocks and discard unreachable blocks.

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)



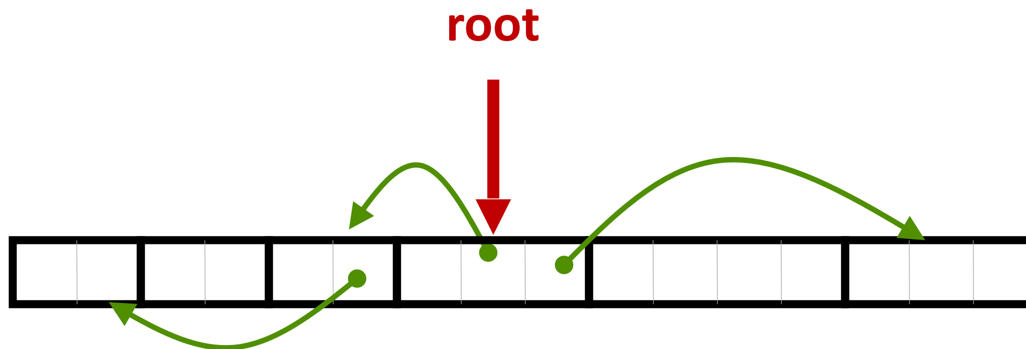


A node (block) is *reachable* if there is a path from any root to that node.

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

Mark and Sweep Collecting

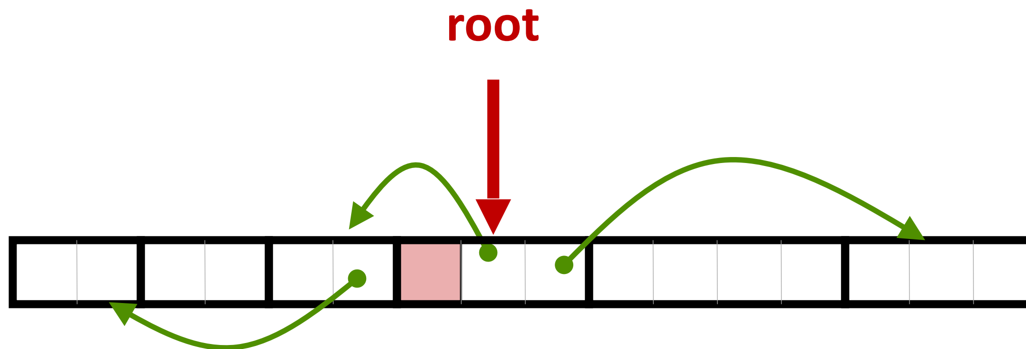
- Idea:
  - Use extra ***mark bit*** in the head to indicate if a block is reachable
  - *Mark:* Start at roots and set mark bit on each reachable block
  - *Sweep:* Scan all blocks and free blocks that are not marked



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

Mark and Sweep Collecting

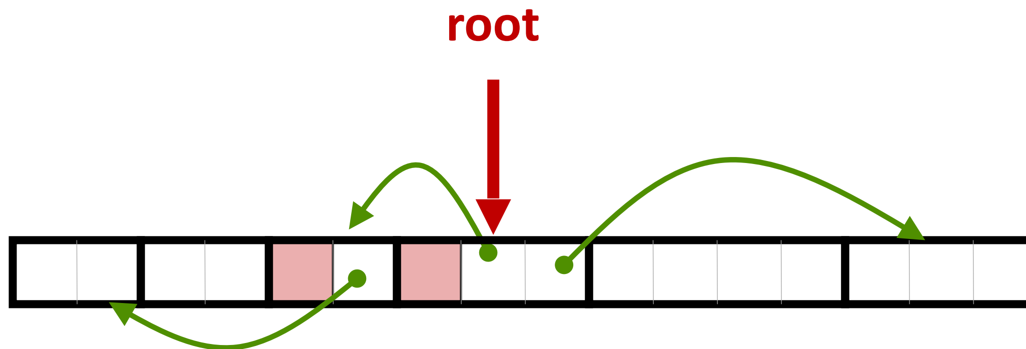
- Idea:
  - Use extra ***mark bit*** in the head to indicate if a block is reachable
  - *Mark:* Start at roots and set mark bit on each reachable block
  - *Sweep:* Scan all blocks and free blocks that are not marked



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

Mark and Sweep Collecting

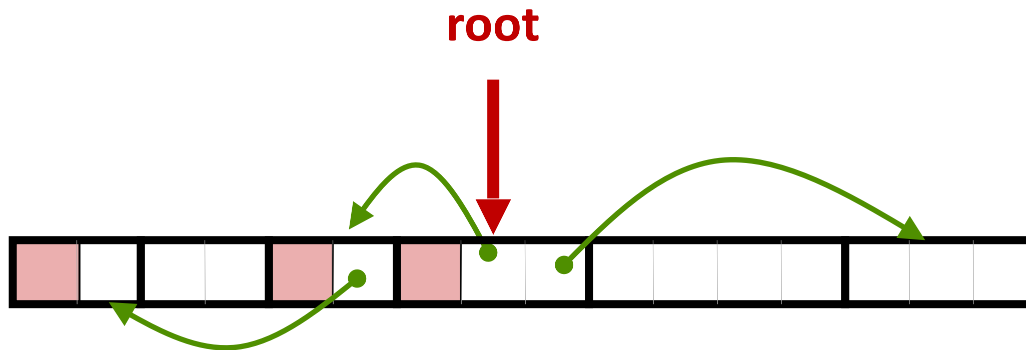
- Idea:
  - Use extra ***mark bit*** in the head to indicate if a block is reachable
  - *Mark:* Start at roots and set mark bit on each reachable block
  - *Sweep:* Scan all blocks and free blocks that are not marked



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

Mark and Sweep Collecting

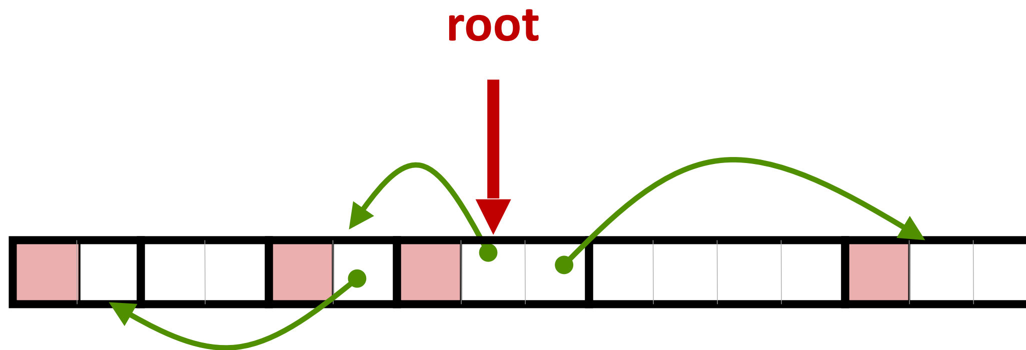
- Idea:
  - Use extra ***mark bit*** in the head to indicate if a block is reachable
  - *Mark:* Start at roots and set mark bit on each reachable block
  - *Sweep:* Scan all blocks and free blocks that are not marked



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

Mark and Sweep Collecting

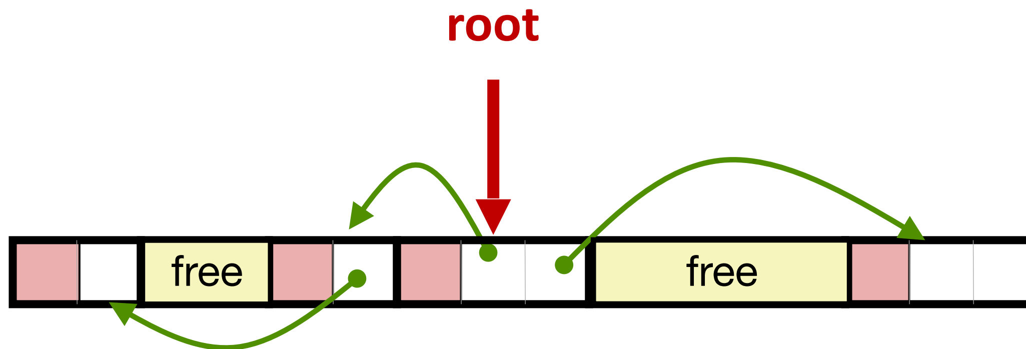
- Idea:
  - Use extra ***mark bit*** in the head to indicate if a block is reachable
  - *Mark:* Start at roots and set mark bit on each reachable block
  - *Sweep:* Scan all blocks and free blocks that are not marked



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

Mark and Sweep Collecting

- Idea:
  - Use extra ***mark bit*** in the head to indicate if a block is reachable
  - *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.

Conservative Mark & Sweep in C

- Garbage Collection in C is tricky.

Conservative Mark & Sweep in C

- Garbage Collection in C is tricky.
- How do you know a pointer is a pointer? After all, a pointer is just a 8-byte value. Any consecutive 8 bytes could be disguised as a pointer.

Conservative Mark & Sweep in C

- Garbage Collection in C is tricky.
- How do you know a pointer is a pointer? After all, a pointer is just a 8-byte value. Any consecutive 8 bytes could be disguised as a pointer.
  - Must be conservative. Any 8 bytes that happen to have the value of some address in the heap must be treated as a pointer.

Conservative Mark & Sweep in C

- Garbage Collection in C is tricky.
- How do you know a pointer is a pointer? After all, a pointer is just a 8-byte value. Any consecutive 8 bytes could be disguised as a pointer.
  - Must be conservative. Any 8 bytes that happen to have the value of some address in the heap must be treated as a pointer.
- C pointers can point to the middle of a block. How do you find the header of a block?

Conservative Mark & Sweep in C

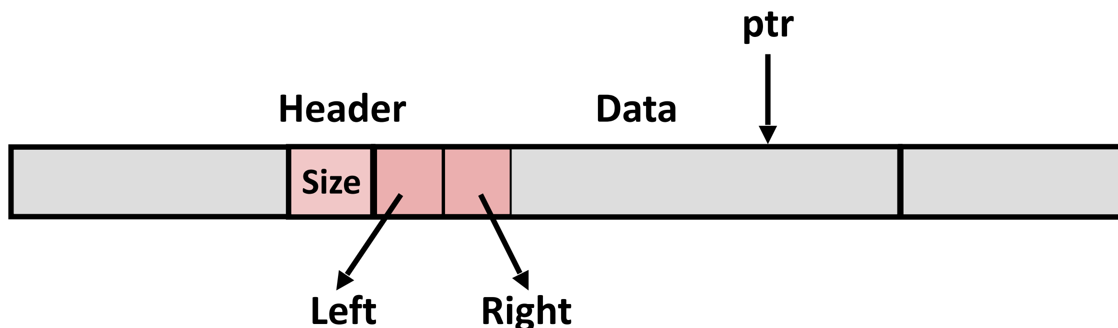
- Garbage Collection in C is tricky.
- How do you know a pointer is a pointer? After all, a pointer is just a 8-byte value. Any consecutive 8 bytes could be disguised as a pointer.
  - Must be conservative. Any 8 bytes that happen to have the value of some address in the heap must be treated as a pointer.
- C pointers can point to the middle of a block. How do you find the header of a block?

Conservative Mark & Sweep in C

- Garbage Collection in C is tricky.
- How do you know a pointer is a pointer? After all, a pointer is just a 8-byte value. Any consecutive 8 bytes could be disguised as a pointer.
  - Must be conservative. Any 8 bytes that happen to have the value of some address in the heap must be treated as a pointer.
- C pointers can point to the middle of a block. How do you find the header of a block?
  - Can use a balanced binary tree to keep track of all allocated blocks (key is start-of-block)

Conservative Mark & Sweep in C

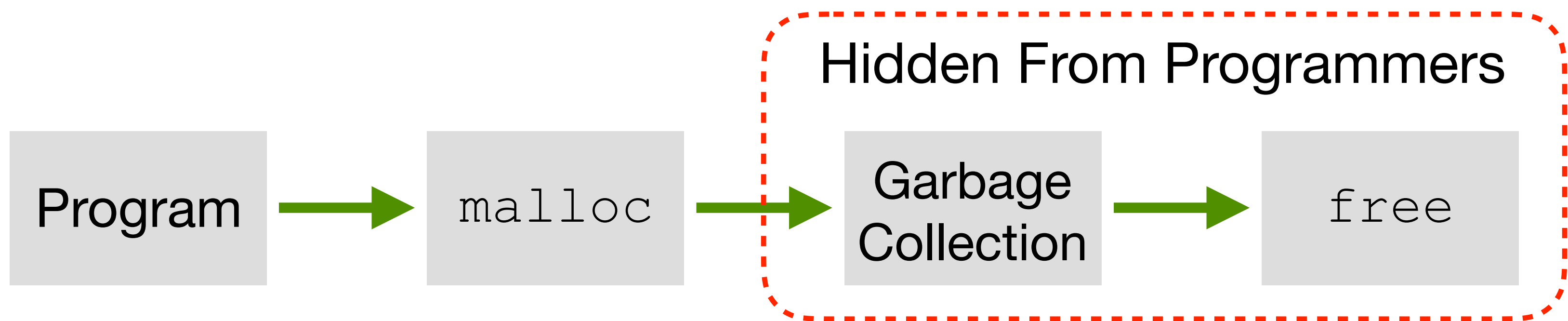
- Garbage Collection in C is tricky.
- How do you know a pointer is a pointer? After all, a pointer is just a 8-byte value. Any consecutive 8 bytes could be disguised as a pointer.
  - Must be conservative. Any 8 bytes that happen to have the value of some address in the heap must be treated as a pointer.
- C pointers can point to the middle of a block. How do you find the header of a block?
  - Can use a balanced binary tree to keep track of all allocated blocks (key is start-of-block)



Left: smaller addresses
Right: larger addresses

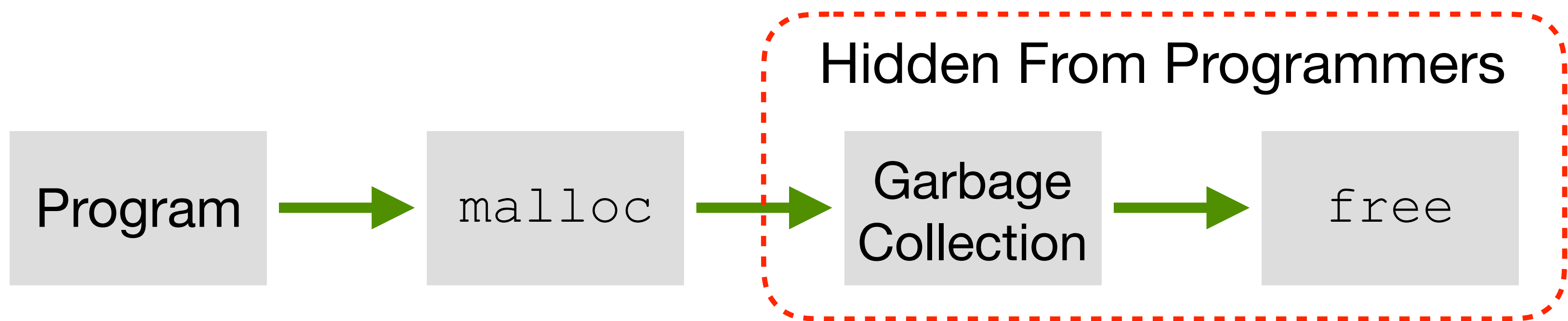
Potential GC Implementations (in C)

- Can build on top of `malloc`/`free` function
  - Call `malloc` until you run out of space. Then `malloc` will call GC.
  - Stop-the-world GC. When performing GC, the entire program stops. Some calls to `malloc` will take considerably longer than others.



Potential GC Implementations (in C)

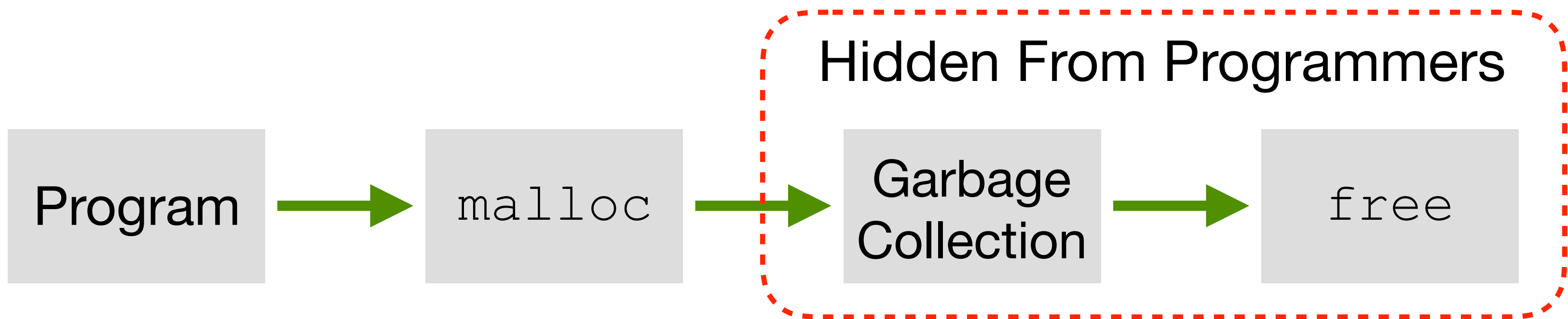
- Can build on top of `malloc`/`free` function
  - Call `malloc` until you run out of space. Then `malloc` will call GC.
  - Stop-the-world GC. When performing GC, the entire program stops. Some calls to `malloc` will take considerably longer than others.



- To minimize main application (called mutator) pause time:

Potential GC Implementations (in C)

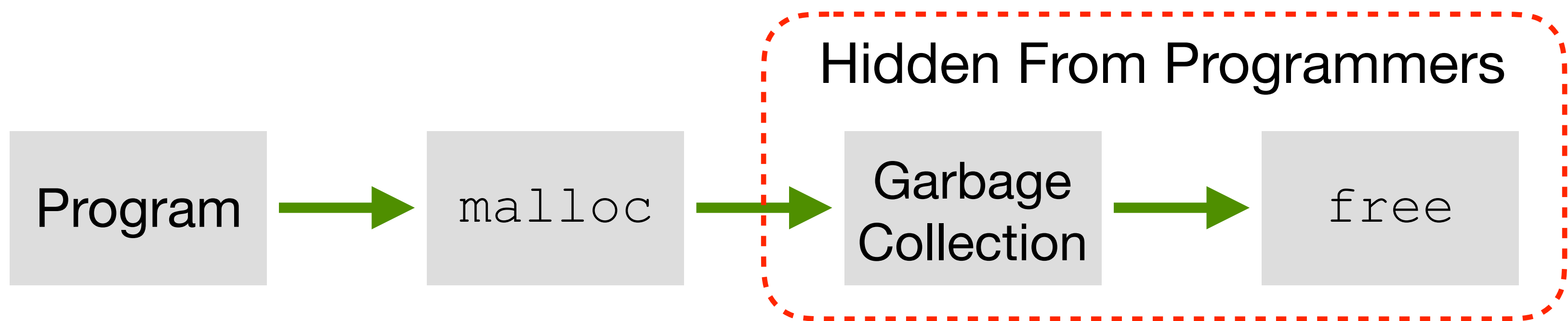
- Can build on top of `malloc`/`free` function
  - Call `malloc` until you run out of space. Then `malloc` will call GC.
  - Stop-the-world GC. When performing GC, the entire program stops. Some calls to `malloc` will take considerably longer than others.



- To minimize main application (called mutator) pause time:
  - Incremental GC: Examine a small portion of heap every GC run

Potential GC Implementations (in C)

- Can build on top of `malloc`/`free` function
  - Call `malloc` until you run out of space. Then `malloc` will call GC.
  - Stop-the-world GC. When performing GC, the entire program stops. Some calls to `malloc` will take considerably longer than others.



- To minimize main application (called mutator) pause time:
  - Incremental GC: Examine a small portion of heap every GC run
  - Concurrent GC: Run GC service in a separate process/thread

Garbage Collection Implications

- GC is a great source of performance non-determinism
  - Generally can't predict when GC will happen

Garbage Collection Implications

- GC is a great source of performance non-determinism
  - Generally can’t predict when GC will happen
  - Stop-the-world GC makes program periodically unresponsive

Garbage Collection Implications

- GC is a great source of performance non-determinism
  - Generally can't predict when GC will happen
  - Stop-the-world GC makes program periodically unresponsive
  - Concurrent/Incremental GC helps, but still has performance impacts

Garbage Collection Implications

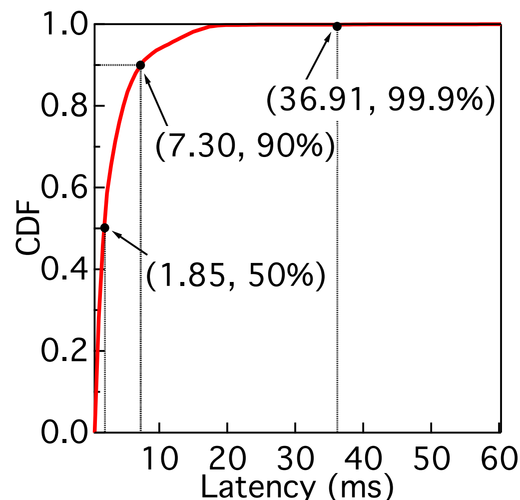
- GC is a great source of performance non-determinism
  - Generally can't predict when GC will happen
  - Stop-the-world GC makes program periodically unresponsive
  - Concurrent/Incremental GC helps, but still has performance impacts
  - Bad for real-time systems: think of a self-driving car that needs to decide whether to avoid a pedestrian but a GC kicks in…

Garbage Collection Implications

- GC is a great source of performance non-determinism
  - Generally can't predict when GC will happen
  - Stop-the-world GC makes program periodically unresponsive
  - Concurrent/Incremental GC helps, but still has performance impacts
  - Bad for real-time systems: think of a self-driving car that needs to decide whether to avoid a pedestrian but a GC kicks in…
  - Bad for server/cloud systems: GC is a great source of *tail latency*

Garbage Collection Implications

- GC is a great source of performance non-determinism
  - Generally can't predict when GC will happen
  - Stop-the-world GC makes program periodically unresponsive
  - Concurrent/Incremental GC helps, but still has performance impacts
  - Bad for real-time systems: think of a self-driving car that needs to decide whether to avoid a pedestrian but a GC kicks in…
  - Bad for server/cloud systems: GC is a great source of *tail latency*



Energy-Efficient Boosting Techniques for Tail Latency in Asynchronous Event-driven Web Services

Classical GC Algorithms

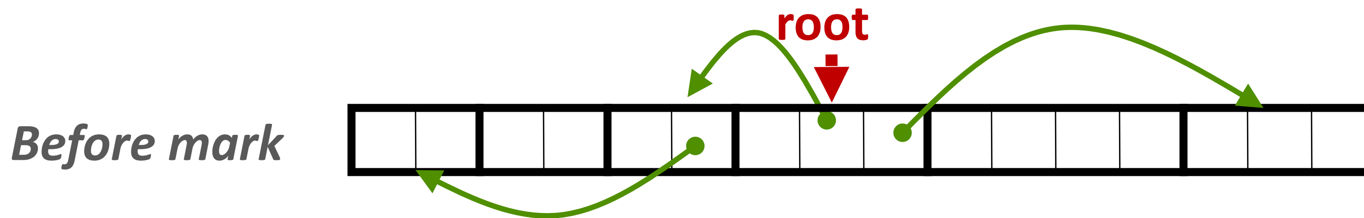
- Mark-and-sweep collection (McCarthy, 1960)

Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
  - After M&S, compact allocated blocks to consecutive memory region.
  - Reduce external fragmentation. Allocation is also easier.

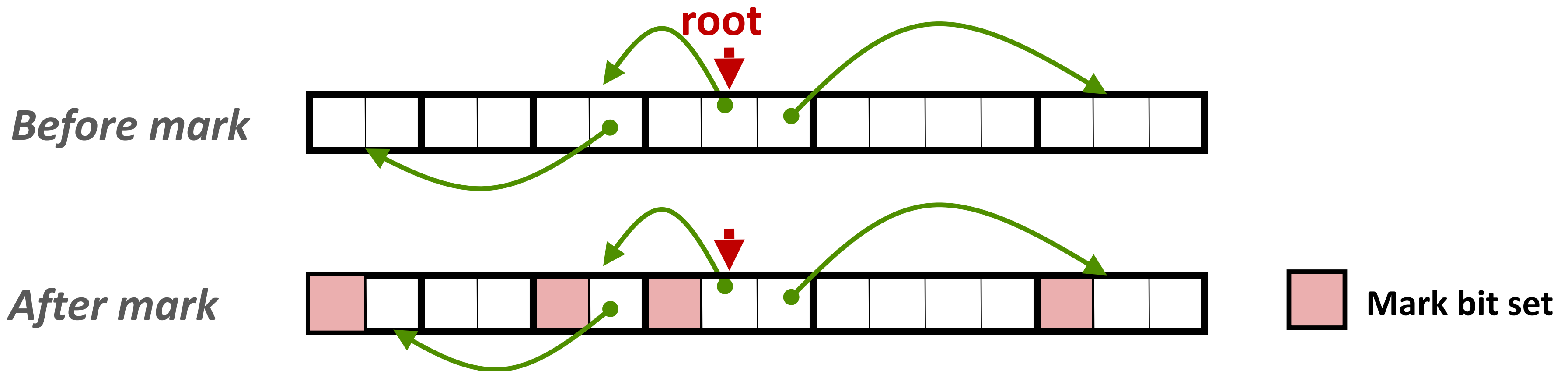
Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
    - After M&S, compact allocated blocks to consecutive memory region.
    - Reduce external fragmentation. Allocation is also easier.



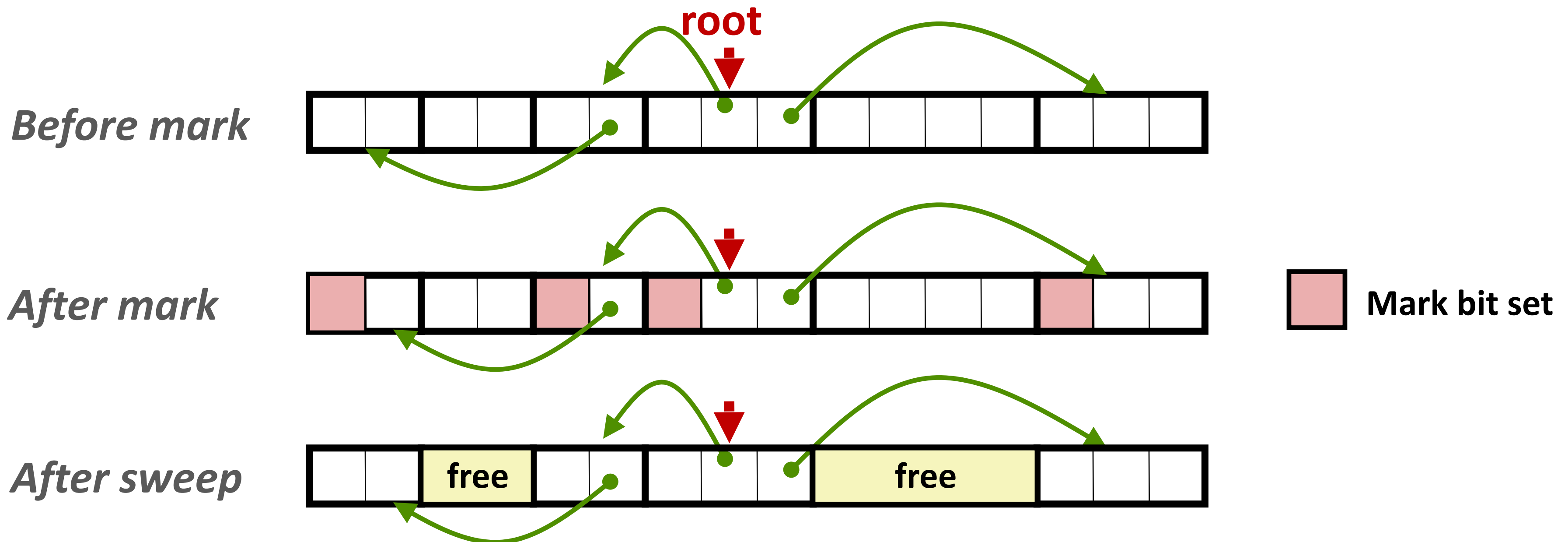
Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
  - After M&S, compact allocated blocks to consecutive memory region.
  - Reduce external fragmentation. Allocation is also easier.



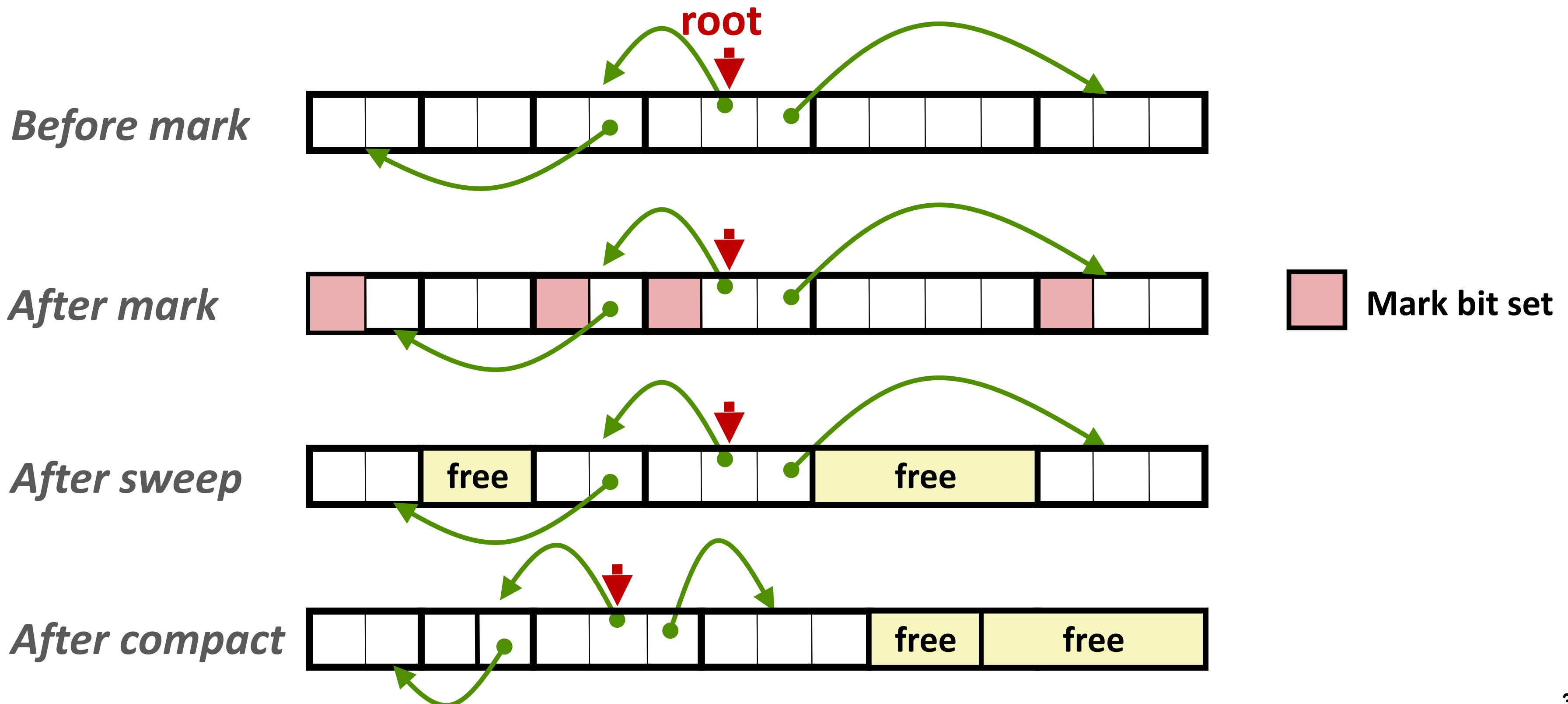
Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
  - After M&S, compact allocated blocks to consecutive memory region.
  - Reduce external fragmentation. Allocation is also easier.



Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
  - After M&S, compact allocated blocks to consecutive memory region.
  - Reduce external fragmentation. Allocation is also easier.



Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)

Jones and Lin, “Garbage Collection: Algorithms for Automatic Dynamic Memory”, 1996.

Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
- Mark-copy collection (Minsky, 1963)
    - After mark, copy reachable objects to another region of memory as they are being traversed. Can be done without auxiliary storage.

Jones and Lin, "Garbage Collection: Algorithms for Automatic Dynamic Memory", 1996.

Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
- Mark-copy collection (Minsky, 1963)
  - After mark, copy reachable objects to another region of memory as they are being traversed. Can be done without auxiliary storage.
- Generational Collectors (Lieberman and Hewitt, 1983)
  - Observation: most allocations become garbage very soon (infant mortality); those survive will always survive.
  - Wasteful to scan long-lived objects every collection time
  - Idea: divide heap into two generations, young and old. Allocate into young gen., and promote to old gen. if lived long enough. Collect young gen. more often than old gen.

Jones and Lin, “Garbage Collection: Algorithms for Automatic Dynamic Memory”, 1996.

Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
- Mark-copy collection (Minsky, 1963)
  - After mark, copy reachable objects to another region of memory as they are being traversed. Can be done without auxiliary storage.
- Generational Collectors (Lieberman and Hewitt, 1983)
  - Observation: most allocations become garbage very soon (infant mortality); those survive will always survive.
  - Wasteful to scan long-lived objects every collection time
  - Idea: divide heap into two generations, young and old. Allocate into young gen., and promote to old gen. if lived long enough. Collect young gen. more often than old gen.
- Question: Can all these algorithms be used for GC in C?

Jones and Lin, “Garbage Collection: Algorithms for Automatic Dynamic Memory”, 1996.

Classical GC Algorithms

- All the GC algorithms described so far are tracing-based
  - Start from the root pointers, trace all the reachable objects
  - Need graph traversal. Different to implement.

Classical GC Algorithms

- All the GC algorithms described so far are tracing-based
  - Start from the root pointers, trace all the reachable objects
  - Need graph traversal. Different to implement.
- Reference counting (Collins, 1960)
  - Keep a counter for each object
  - Increment the counter if there is a new pointer pointing to the object
  - Decrement the counter if a pointer is taken off the object
  - When the counter reaches zero, collect the object

Classical GC Algorithms

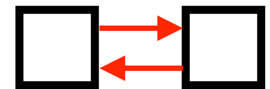
- All the GC algorithms described so far are tracing-based
  - Start from the root pointers, trace all the reachable objects
  - Need graph traversal. Different to implement.
- Reference counting (Collins, 1960)
  - Keep a counter for each object
  - Increment the counter if there is a new pointer pointing to the object
  - Decrement the counter if a pointer is taken off the object
  - When the counter reaches zero, collect the object
- Advantages of Reference Counting
  - Simpler to implement
  - Collect garbage objects immediately; generally less long pauses

Classical GC Algorithms

- All the GC algorithms described so far are tracing-based
  - Start from the root pointers, trace all the reachable objects
  - Need graph traversal. Different to implement.
- Reference counting (Collins, 1960)
  - Keep a counter for each object
  - Increment the counter if there is a new pointer pointing to the object
  - Decrement the counter if a pointer is taken off the object
  - When the counter reaches zero, collect the object
- Advantages of Reference Counting
  - Simpler to implement
  - Collect garbage objects immediately; generally less long pauses
- Disadvantages of Reference Counting
  - A naive implementation can't deal with self-referencing

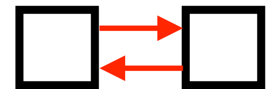
Classical GC Algorithms

- All the GC algorithms described so far are tracing-based
  - Start from the root pointers, trace all the reachable objects
  - Need graph traversal. Different to implement.
- Reference counting (Collins, 1960)
  - Keep a counter for each object
  - Increment the counter if there is a new pointer pointing to the object
  - Decrement the counter if a pointer is taken off the object
  - When the counter reaches zero, collect the object
- Advantages of Reference Counting
  - Simpler to implement
  - Collect garbage objects immediately; generally less long pauses
- Disadvantages of Reference Counting
  - A naive implementation can't deal with self-referencing



Classical GC Algorithms

- All the GC algorithms described so far are tracing-based
  - Start from the root pointers, trace all the reachable objects
  - Need graph traversal. Different to implement.
- Reference counting (Collins, 1960)
  - Keep a counter for each object
  - Increment the counter if there is a new pointer pointing to the object
  - Decrement the counter if a pointer is taken off the object
  - When the counter reaches zero, collect the object
- Advantages of Reference Counting
  - Simpler to implement
  - Collect garbage objects immediately; generally less long pauses
- Disadvantages of Reference Counting
  - A naive implementation can't deal with self-referencing



- A heterogeneous approach (RC + tracing) is often used