

CSC 252: Computer Organization

Spring 2023: Lecture 23

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

malloc Example

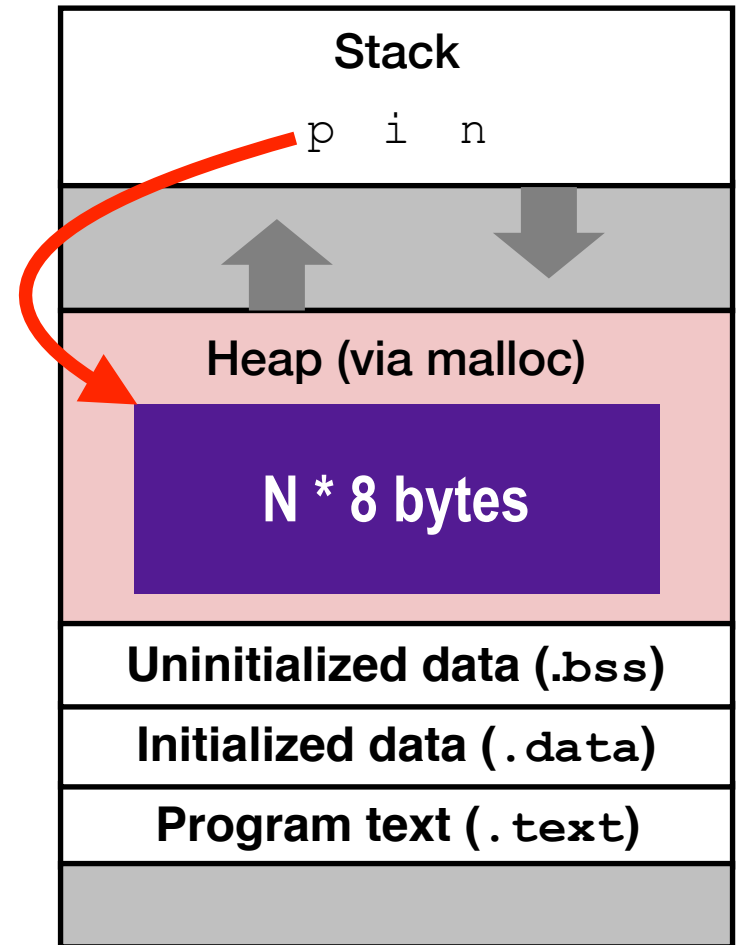
```
#include <stdio.h>
#include <stdlib.h>

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);
}
```



Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

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

    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) exit(0);

    for (i=0; i<n; i++)
        p[i] = i;

    return p;
}

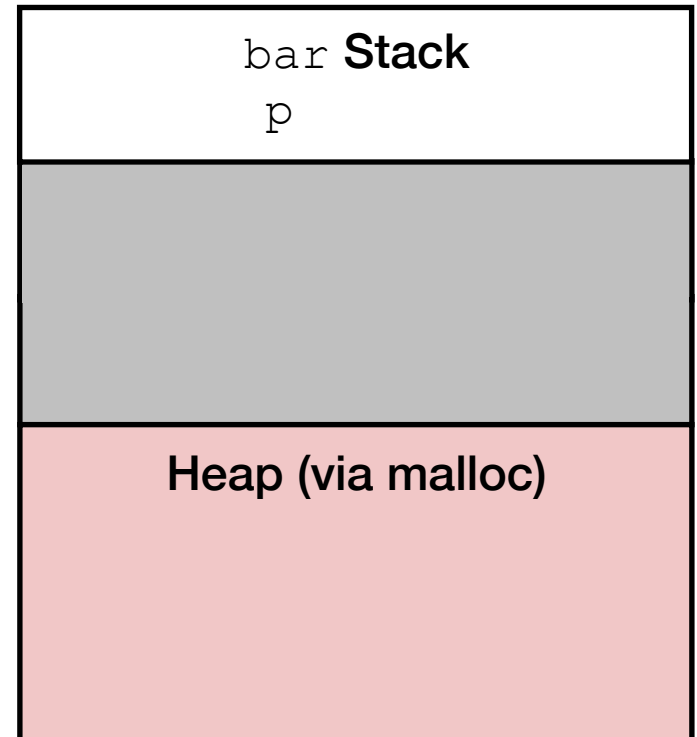
void bar() {
    int *p = foo(5);

    printf("%d\n", p[0]);
}
```

Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

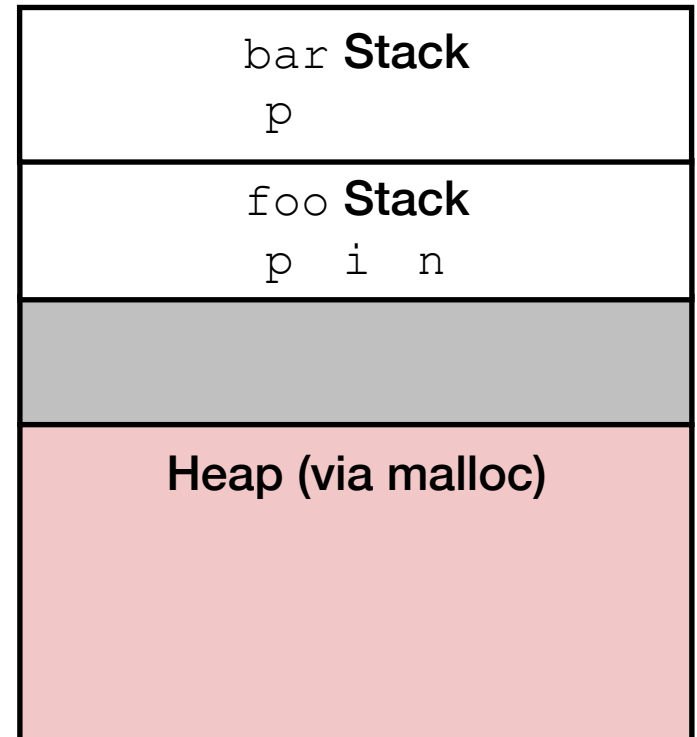
```
int* foo(int n) {  
    int i, *p;  
  
    p = (int *) malloc(n * sizeof(int));  
    if (p == NULL) exit(0);  
  
    for (i=0; i<n; i++)  
        p[i] = i;  
  
    return p;  
}  
  
void bar() {  
    int *p = foo(5);  
  
    printf("%d\n", p[0]);  
}
```



Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

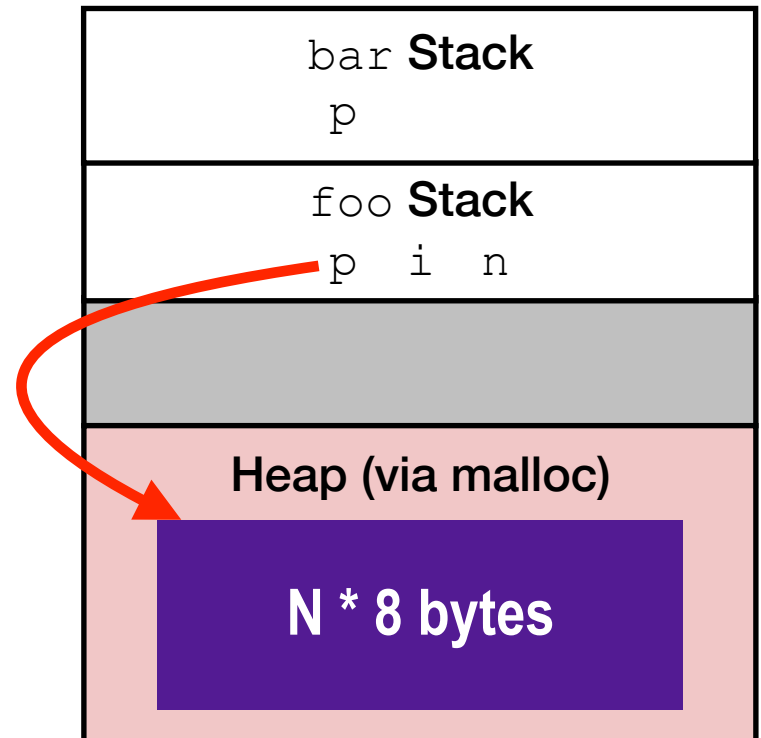
```
int* foo(int n) {  
    int i, *p;  
  
    p = (int *) malloc(n * sizeof(int));  
    if (p == NULL) exit(0);  
  
    for (i=0; i<n; i++)  
        p[i] = i;  
  
    return p;  
}  
  
void bar() {  
    int *p = foo(5);  
  
    printf("%d\n", p[0]);  
}
```



Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

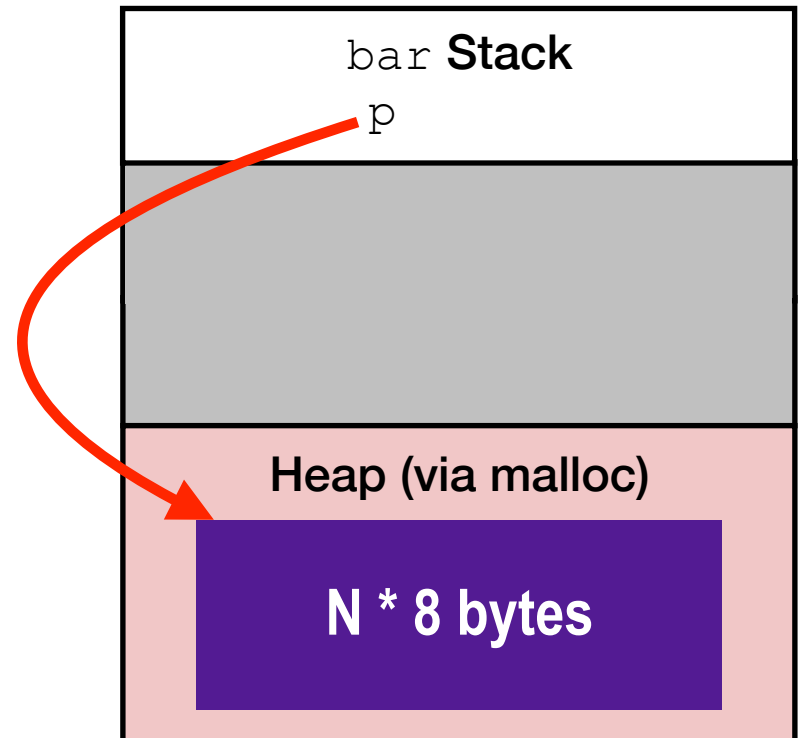
```
int* foo(int n) {  
    int i, *p;  
  
    p = (int *) malloc(n * sizeof(int));  
    if (p == NULL) exit(0);  
  
    for (i=0; i<n; i++)  
        p[i] = i;  
  
    return p;  
}  
  
void bar() {  
    int *p = foo(5);  
  
    printf("%d\n", p[0]);  
}
```



Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

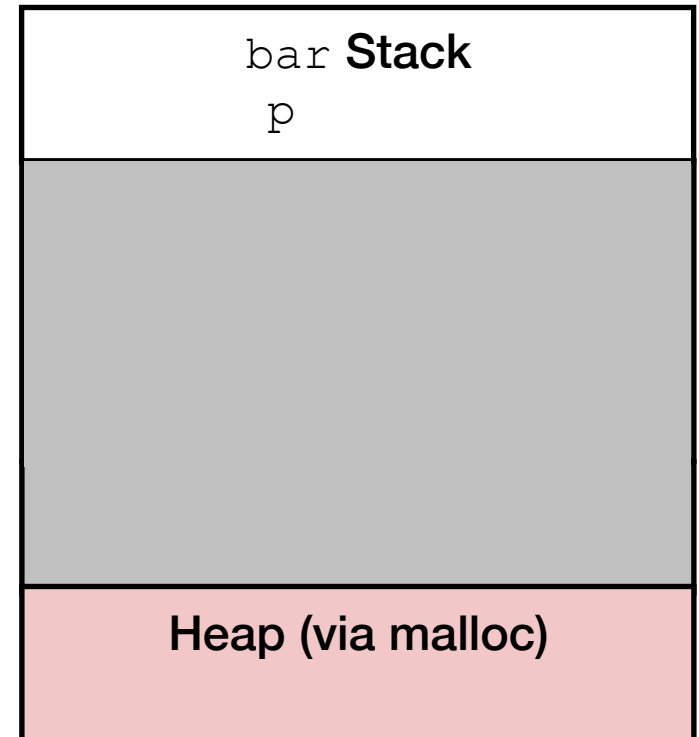
```
int* foo(int n) {  
    int i, *p;  
  
    p = (int *) malloc(n * sizeof(int));  
    if (p == NULL) exit(0);  
  
    for (i=0; i<n; i++)  
        p[i] = i;  
  
    return p;  
}  
  
void bar() {  
    int *p = foo(5);  
  
    printf("%d\n", p[0]);  
}
```



Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

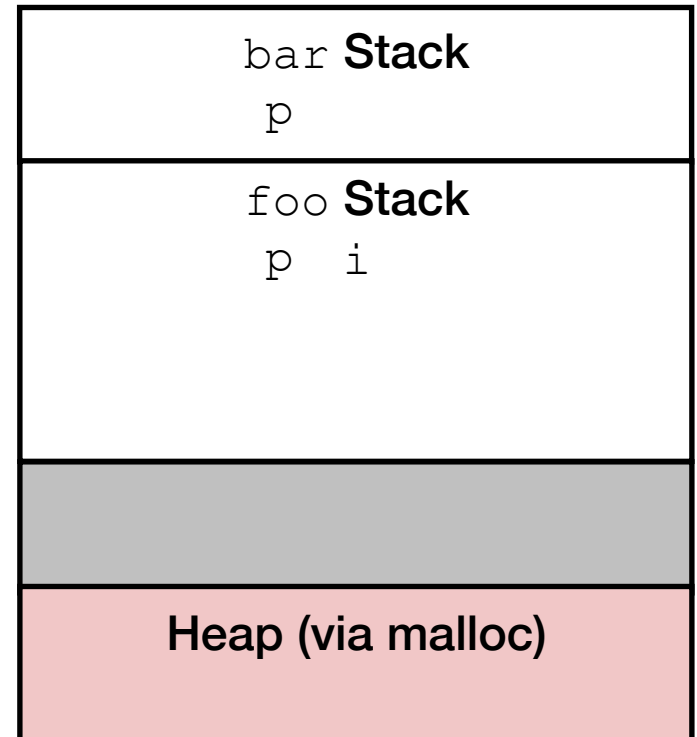
```
int* foo() {  
    int i;  
    int p[5];  
  
    for (i=0; i<5; i++)  
        p[i] = i;  
  
    return p;  
}  
  
void bar() {  
    int *p = foo();  
  
    printf("%d\n", p[0]);  
}
```



Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

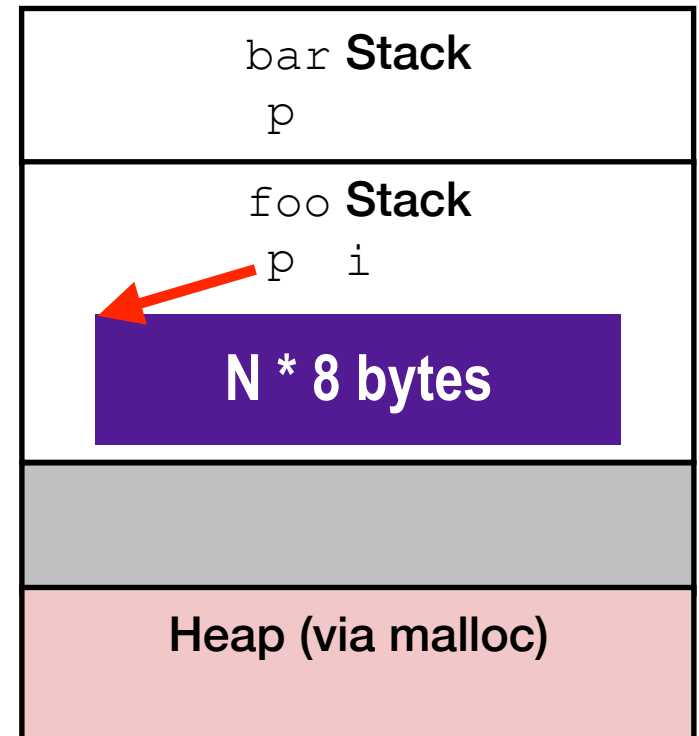
```
int* foo() {  
    int i;  
    int p[5];  
  
    for (i=0; i<5; i++)  
        p[i] = i;  
  
    return p;  
}  
  
void bar() {  
    int *p = foo();  
  
    printf("%d\n", p[0]);  
}
```



Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

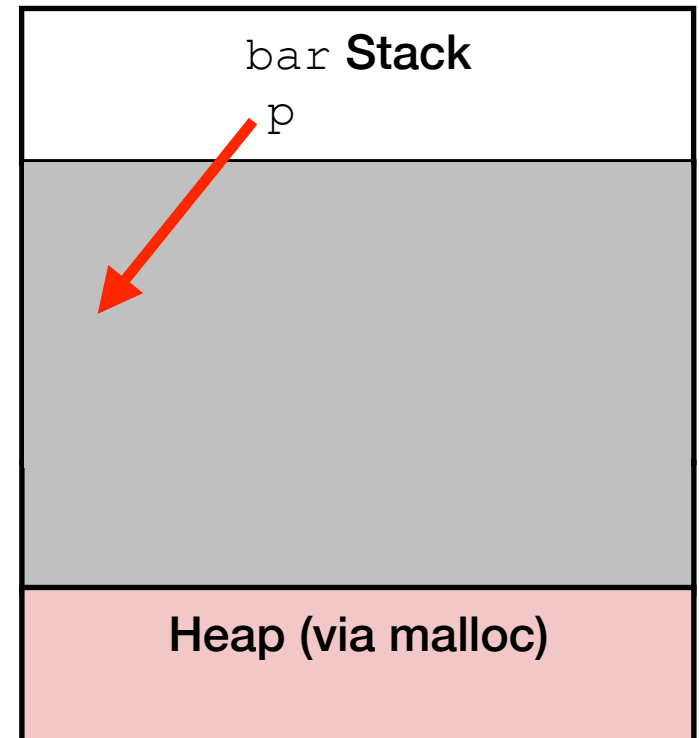
```
int* foo() {  
    int i;  
    int p[5];  
  
    for (i=0; i<5; i++)  
        p[i] = i;  
  
    return p;  
}  
  
void bar() {  
    int *p = foo();  
  
    printf("%d\n", p[0]);  
}
```



Why Do We Need Dynamic Allocation?

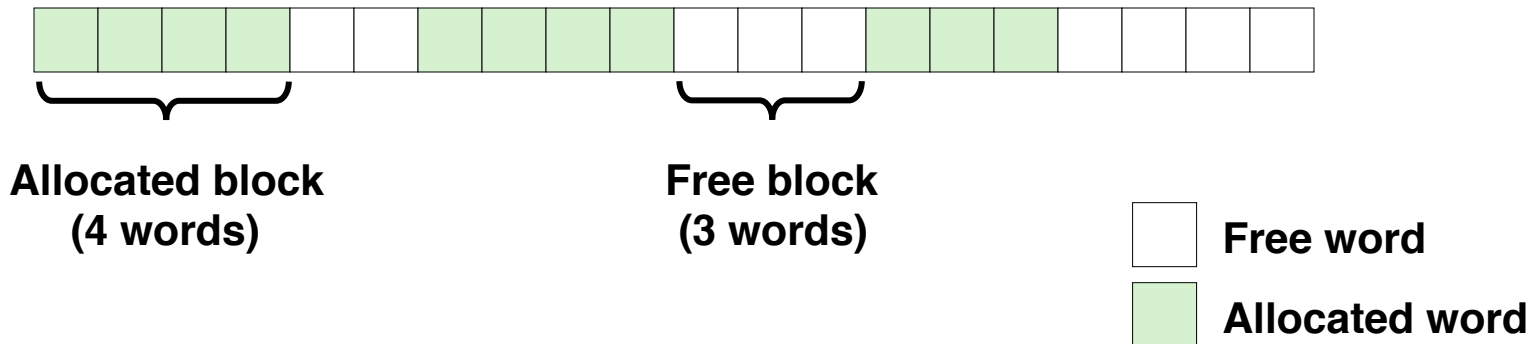
- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

```
int* foo() {  
    int i;  
    int p[5];  
  
    for (i=0; i<5; i++)  
        p[i] = i;  
  
    return p;  
}  
  
void bar() {  
    int *p = foo();  
  
    printf("%d\n", p[0]);  
}
```



Dynamic Memory Allocation

- Allocator maintains heap as collection of variable sized *blocks/chunks*, which are either *allocated* or *free*
- Blocks that are no longer used should be free-ed to save space



- Assumptions Made in This Lecture
 - Memory is word addressed
 - Words are int-sized

Dynamic Memory Allocation

- Types of allocators
 - *Explicit allocator*: application (i.e., programmer) 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, JavaScript, Python, etc...
- Will discuss simple explicit memory allocation today

Allocation Example

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



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

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

```
free(p2)
```

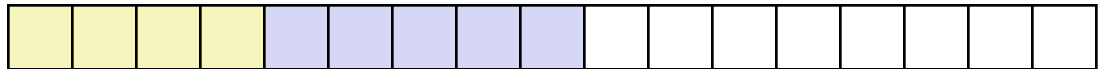
```
p4 = malloc(2)
```

Allocation Example

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



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



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

```
free(p2)
```

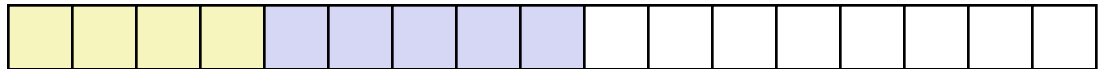
```
p4 = malloc(2)
```


Allocation Example

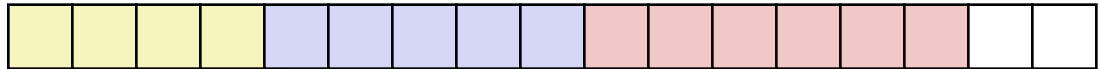
```
p1 = malloc(4)
```



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



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



```
free(p2)
```

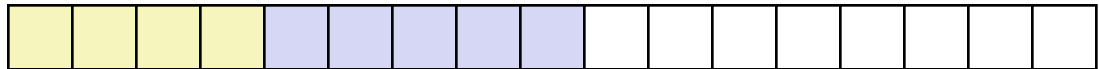
```
p4 = malloc(2)
```

Allocation Example

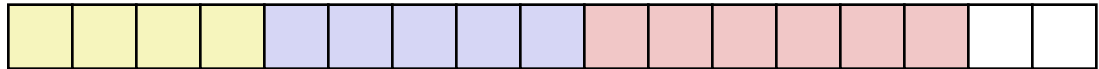
`p1 = malloc(4)`



`p2 = malloc(5)`



`p3 = malloc(6)`



`free(p2)`



`p4 = malloc(2)`

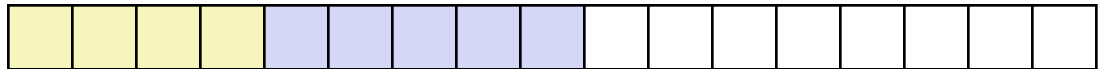


Allocation Example

`p1 = malloc(4)`



`p2 = malloc(5)`



`p3 = malloc(6)`



`free(p2)`



`p4 = malloc(2)`



Constraints

- Applications

Constraints

- Applications
 - Can issue arbitrary sequence of **malloc** and **free** requests

Constraints

- Applications
 - Can issue arbitrary sequence of **malloc** and **free** requests
 - `free` request must be to a `malloc`'d block

Constraints

- Applications
 - Can issue arbitrary sequence of **malloc** and **free** requests
 - `free` request must be to a `malloc`'d block
- Allocators

Constraints

- Applications
 - Can issue arbitrary sequence of **malloc** and **free** requests
 - `free` request must be to a `malloc`'d block
- Allocators
 - Can't control number or size of allocated blocks

Constraints

- Applications
 - Can issue arbitrary sequence of **malloc** and **free** requests
 - `free` request must be to a `malloc`'d block
- Allocators
 - Can't control number or size of allocated blocks
 - Must respond immediately to **malloc** requests

Constraints

- Applications
 - Can issue arbitrary sequence of `malloc` and `free` requests
 - `free` request must be to a `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

Constraints

- Applications
 - Can issue arbitrary sequence of `malloc` and `free` requests
 - `free` request must be to a `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

Constraints

- Applications
 - Can issue arbitrary sequence of **malloc** and **free** requests
 - `free` request must be to a `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.*, can place allocated blocks only in free memory

Constraints

- Applications
 - Can issue arbitrary sequence of `malloc` and `free` requests
 - `free` request must be to a `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.*, can place allocated blocks only in free memory
 - Must align blocks so they satisfy all alignment requirements

Constraints

- Applications
 - Can issue arbitrary sequence of **malloc** and **free** requests
 - **free** request must be to a **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.*, can place allocated blocks only in free memory
 - Must align blocks so they satisfy all alignment requirements
 - 8-byte (x86) or 16-byte (x86-64) alignment on Linux boxes

Constraints

- Applications
 - Can issue arbitrary sequence of **malloc** and **free** requests
 - `free` request must be to a `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.*, can place allocated blocks only in free memory
 - Must align blocks so they satisfy all alignment requirements
 - 8-byte (x86) or 16-byte (x86-64) alignment on Linux boxes
 - Can manipulate and modify only free memory

Constraints

- Applications
 - Can issue arbitrary sequence of **malloc** and **free** requests
 - **free** request must be to a **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.*, can place allocated blocks only in free memory
 - Must align blocks so they satisfy all alignment requirements
 - 8-byte (x86) or 16-byte (x86-64) alignment on Linux boxes
 - Can manipulate and modify only free memory
 - Can't move the allocated blocks once they are **malloc**'d

Constraints

- Applications
 - Can issue arbitrary sequence of `malloc` and `free` requests
 - `free` request must be to a `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.*, can place allocated blocks only in free memory
 - Must align blocks so they satisfy all alignment requirements
 - 8-byte (x86) or 16-byte (x86-64) alignment on Linux boxes
 - Can manipulate and modify only free memory
 - Can't move the allocated blocks once they are `malloc`'d
 - *i.e.*, compaction is not allowed; more on this later. Why?

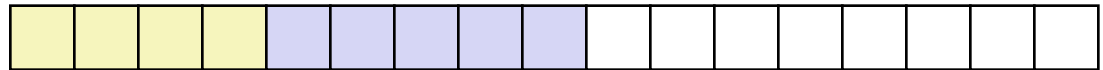
External Fragmentation

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

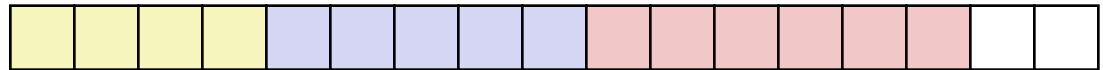
```
p1 = malloc(4)
```



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



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



```
free(p2)
```



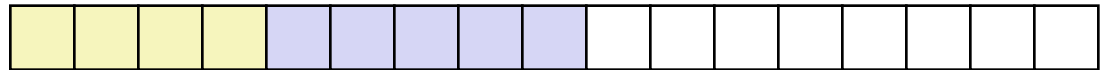
External Fragmentation

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

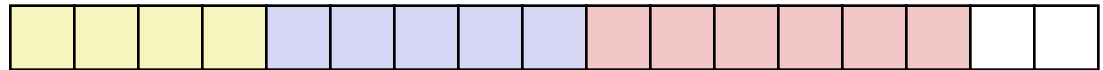
```
p1 = malloc(4)
```



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



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



```
free(p2)
```



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

Oops! (what would happen now?)

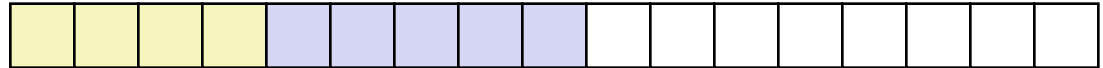
External Fragmentation

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

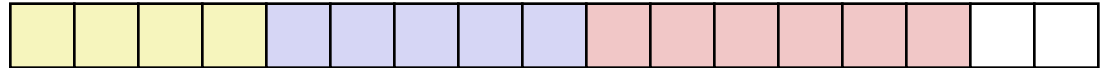
```
p1 = malloc(4)
```



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



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



```
free(p2)
```



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

Oops! (what would happen now?)

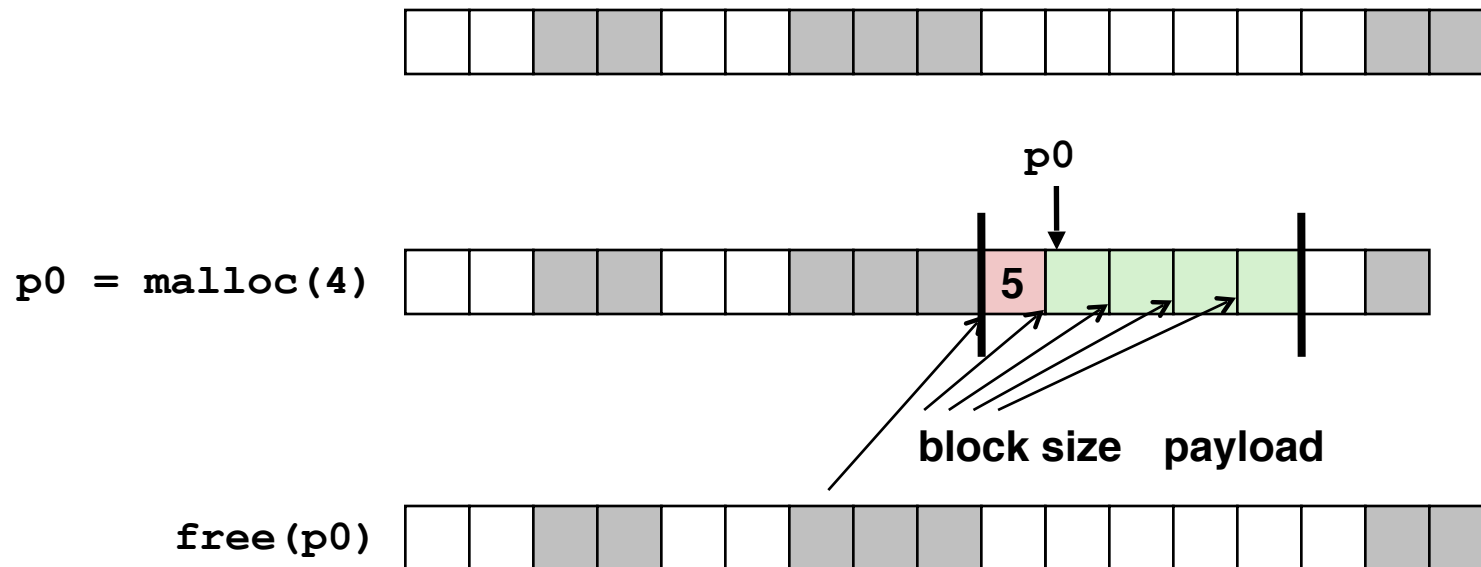
- Depends on the pattern of future requests

Key Issues in Dynamic Memory Allocation

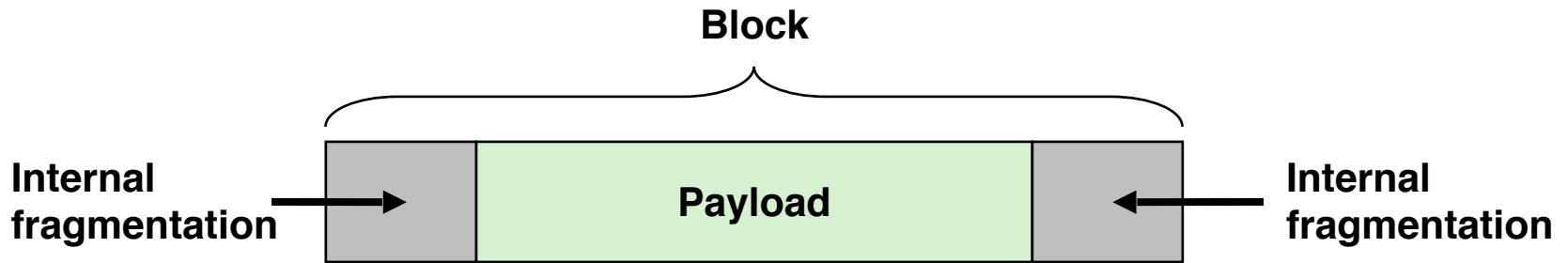
- Free:
 - How do we know how much memory to free given just a pointer?
 - How do we keep track of the free blocks?
 - How do we reinsert freed block?
- Allocation:
 - What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
 - How do we pick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

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

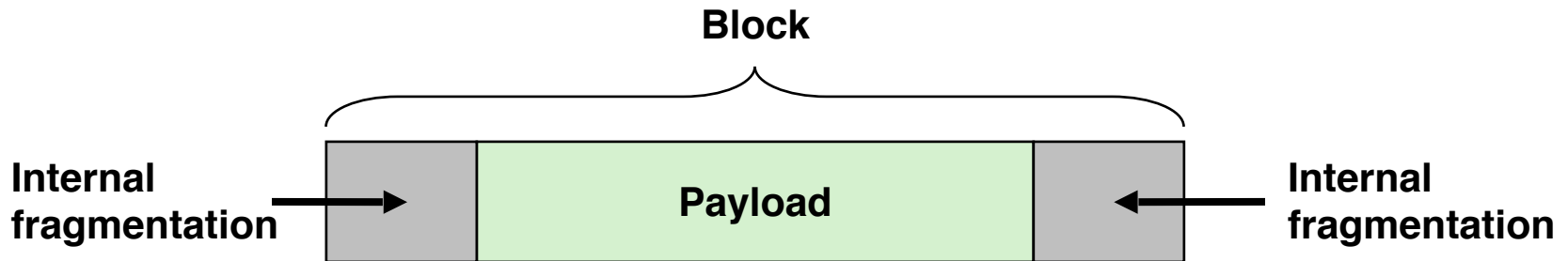


Internal Fragmentation



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

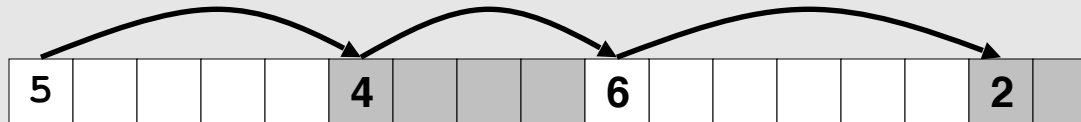
Internal Fragmentation



- For a given block, *internal fragmentation* occurs if payload is smaller than block size
- Caused by
 - Overhead of maintaining heap data structures
 - Padding for alignment purposes
 - Explicit policy decisions (e.g., to return a big block to satisfy a small request)

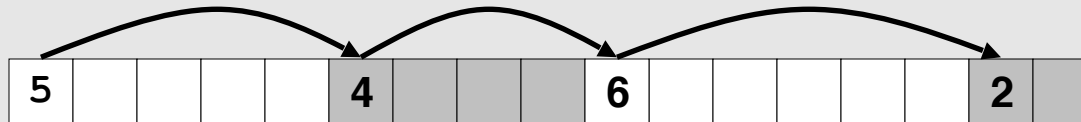
Keeping Track of Free Blocks

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



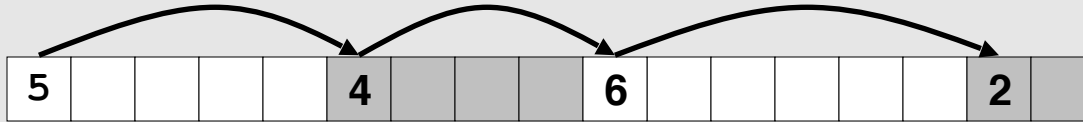
Keeping Track of Free Blocks

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

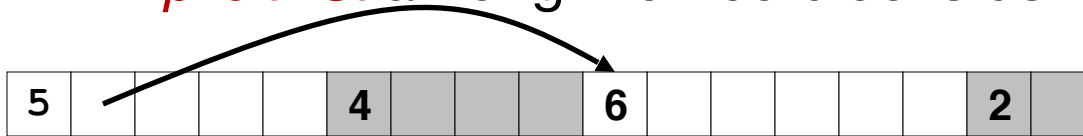


Keeping Track of Free Blocks

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

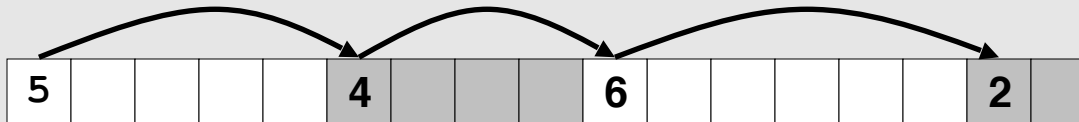


- Method 2: *Explicit list* among the free blocks using pointers

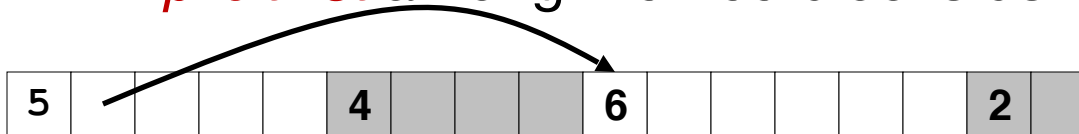


Keeping Track of Free Blocks

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



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



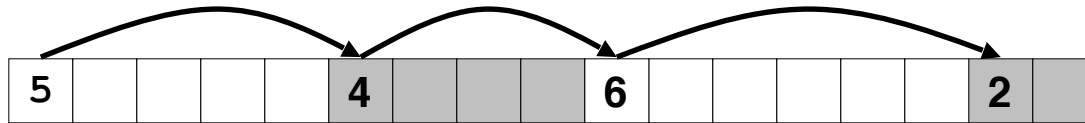
- Method 3: *Segregated free list*
 - Different free lists for different size classes

Today

- Memory mapping
- Dynamic memory allocation
 - Basic concepts
 - Implicit free lists

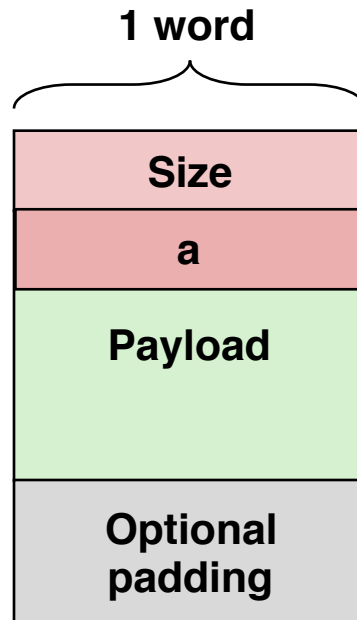
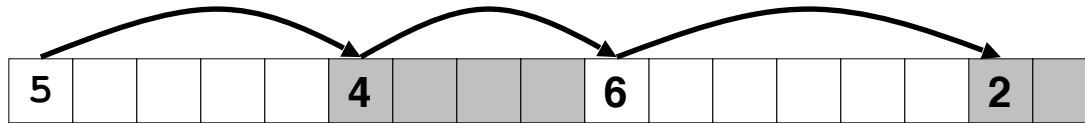
Implicit List

- 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)**

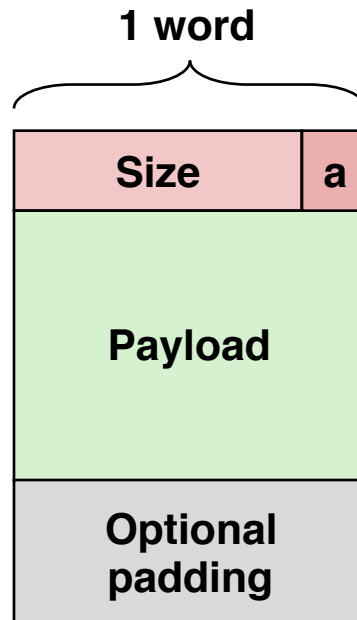
Implicit List

- 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*



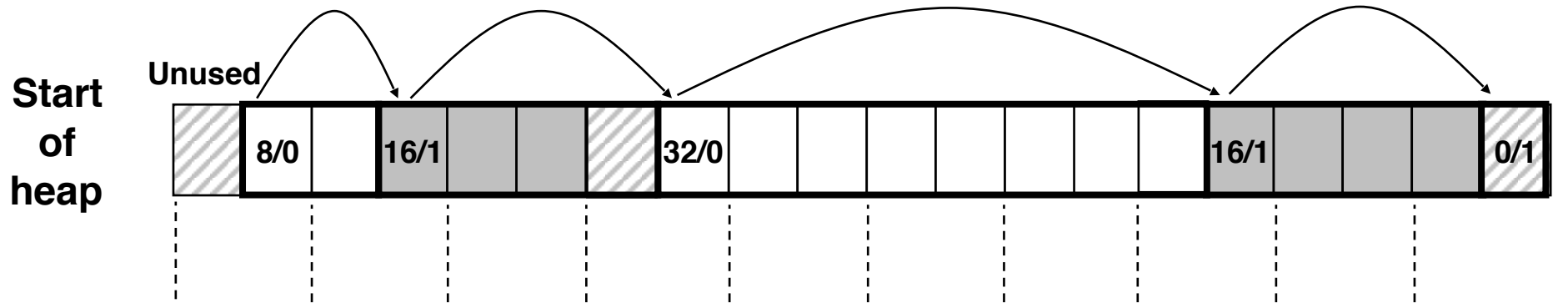
a = 1: Allocated block

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

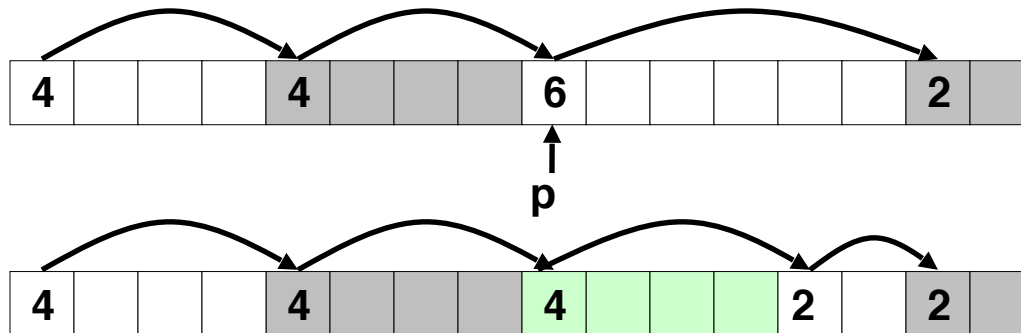
- **First fit:**
 - Search list from beginning, choose **first** free block that fits
 - Can take linear time in total number of blocks (allocated and free)
 - In practice it can cause “splinters” at beginning of list
- **Next fit:**
 - Like first fit, but search list starting where previous search finished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



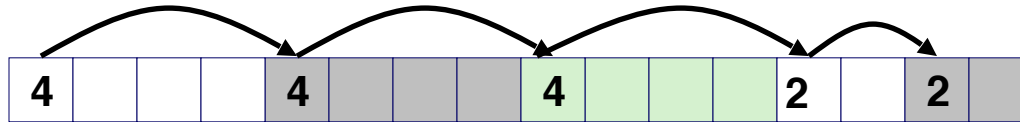
```
void addblock(ptr p, int len) {  
    int newsize = ((len + 1) >> 1) << 1; // round up to even  
    int oldsize = *p & -2;                // mask out low bit  
    *p = newsize | 1;                      // set new length  
    if (newsize < oldsize)  
        *(p+newsize) = oldsize - newsize; //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



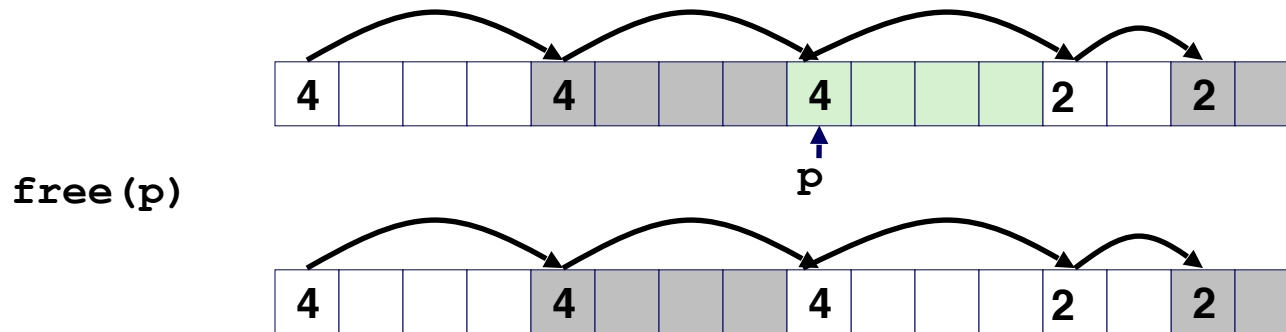
Freeing a Block

- Simplest implementation:

- Need only clear the “allocated” flag

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

- But can lead to “false fragmentation”



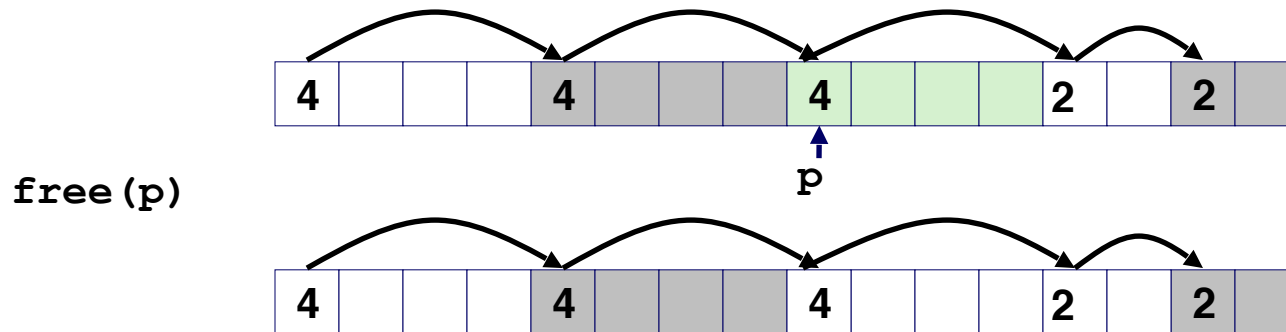
Freeing a Block

- Simplest implementation:

- Need only clear the “allocated” flag

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

- But can lead to “false fragmentation”



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

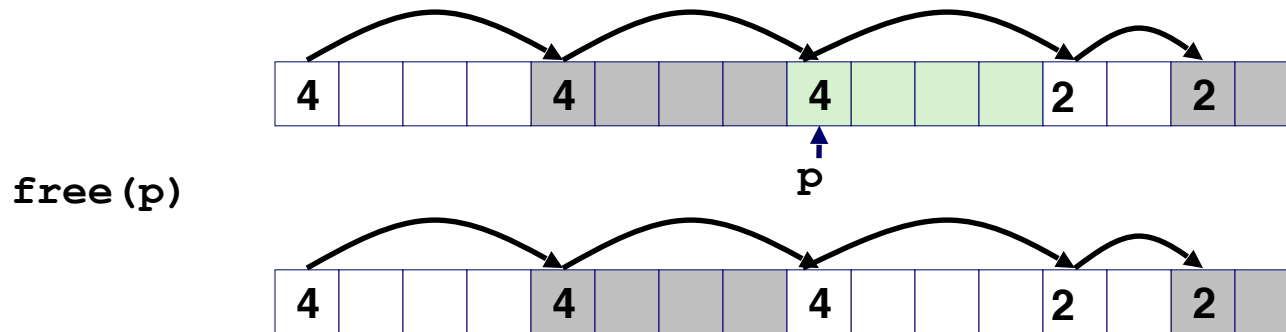
Freeing a Block

- Simplest implementation:

- Need only clear the “allocated” flag

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

- But can lead to “false fragmentation”

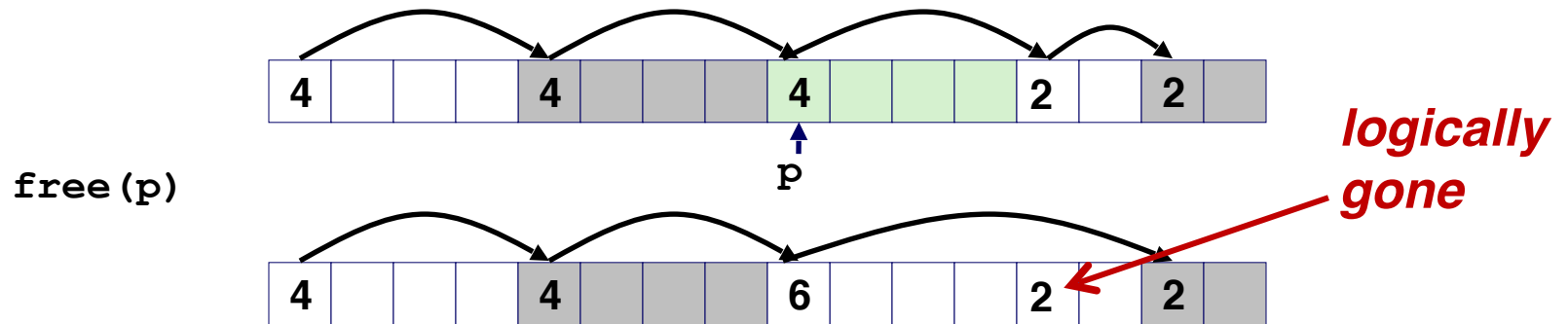


`malloc(5)` ***Oops!***

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

Coalescing

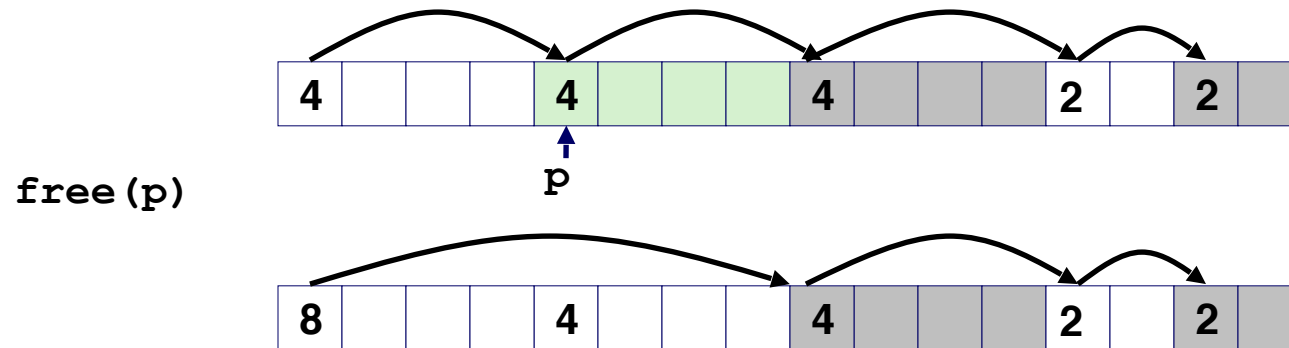
- Join (*coalesce*) with next/previous blocks, if they are free
 - Coalescing with next block



```
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  
}
```

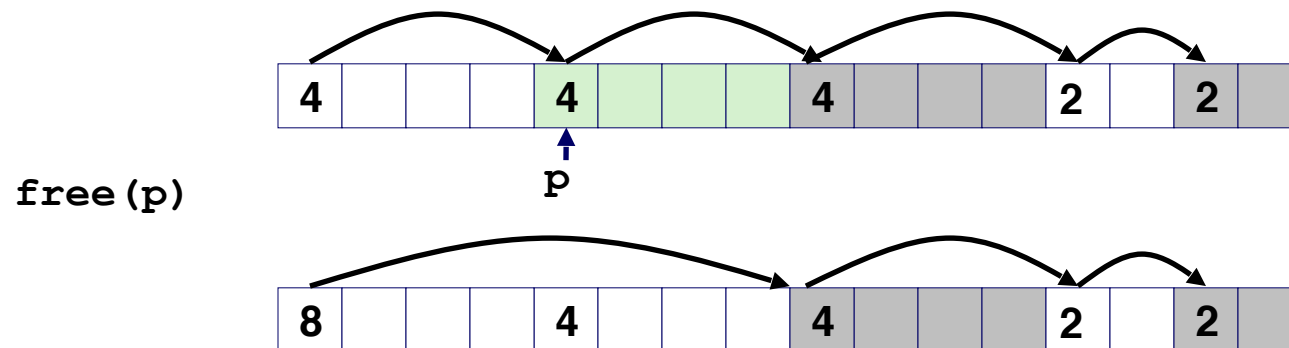
Coalescing

- How about now?



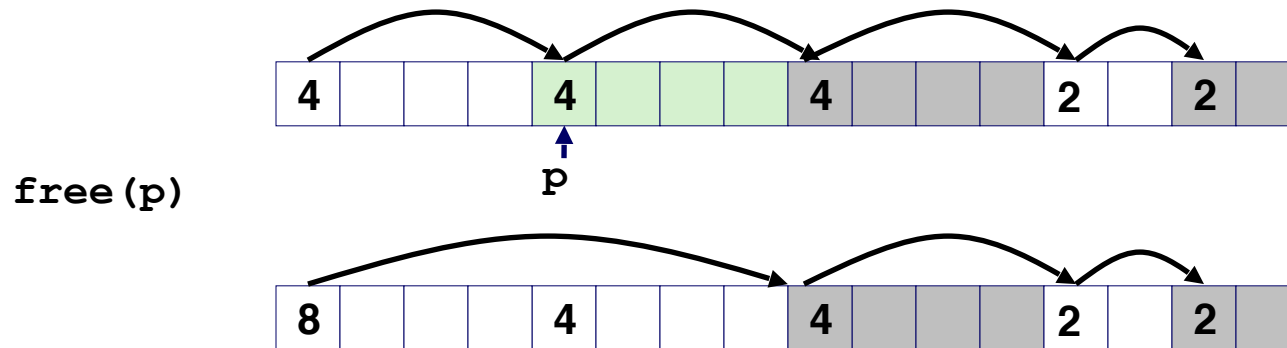
Coalescing

- How about now?
- How do we coalesce with previous block?



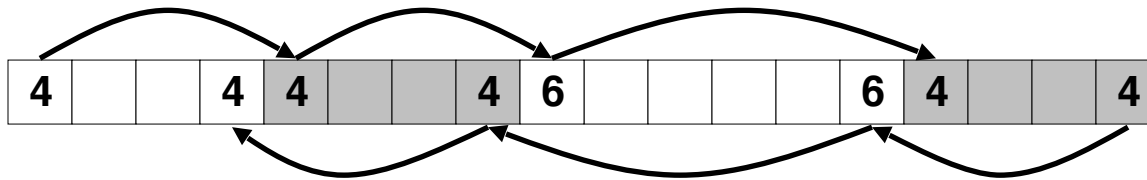
Coalescing

- How about now?
- How do we coalesce with previous block?
 - Linear time solution: scans from beginning



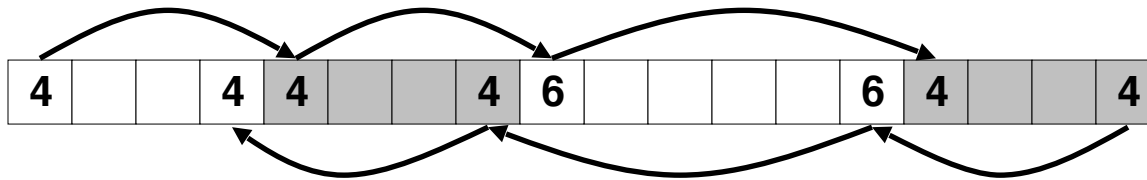
Bidirectional Coalescing (Constant Time)

- *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!

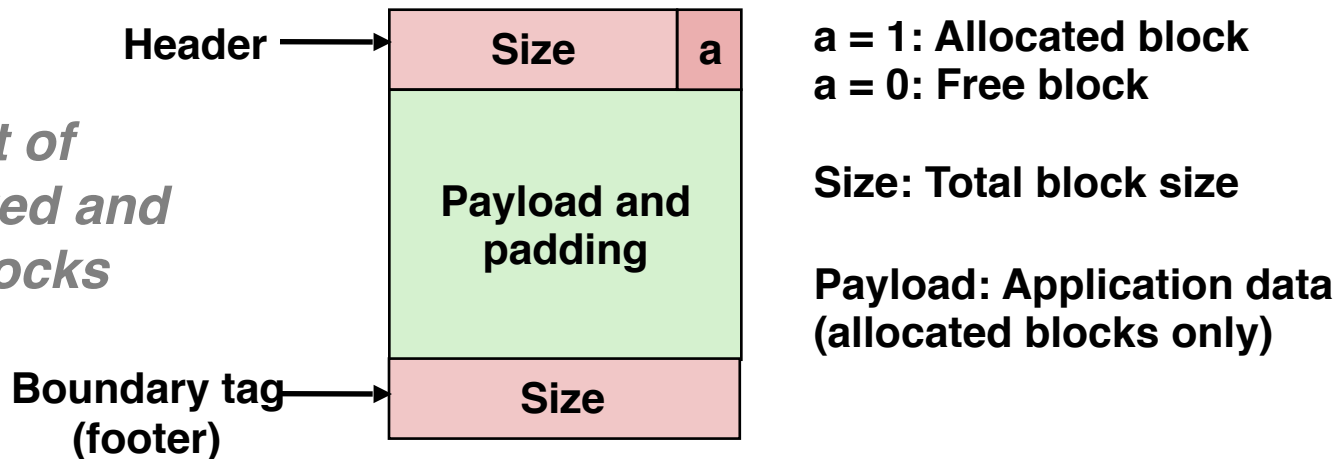


Bidirectional Coalescing (Constant Time)

- *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!

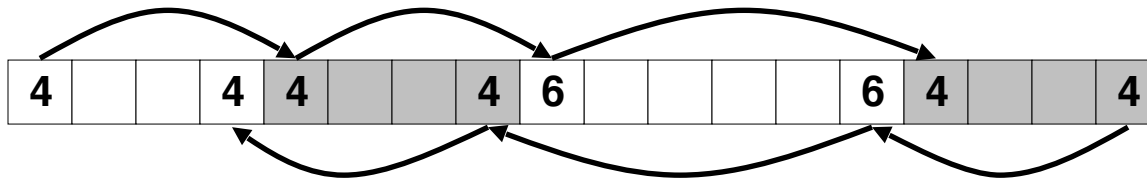


*Format of
allocated and
free blocks*

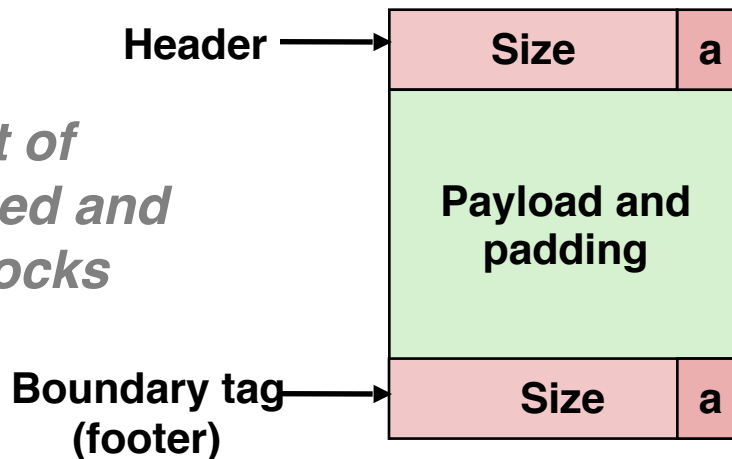


Bidirectional Coalescing (Constant Time)

- *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*



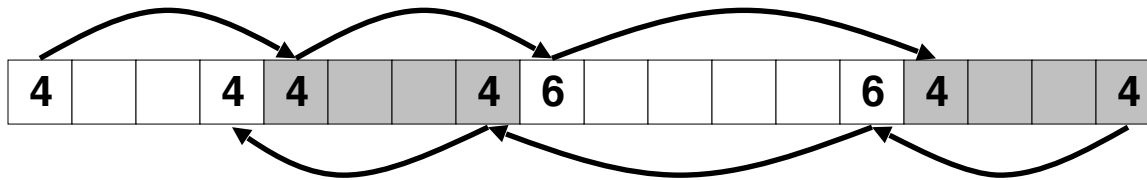
a = 1: Allocated block
a = 0: Free block

Size: Total block size

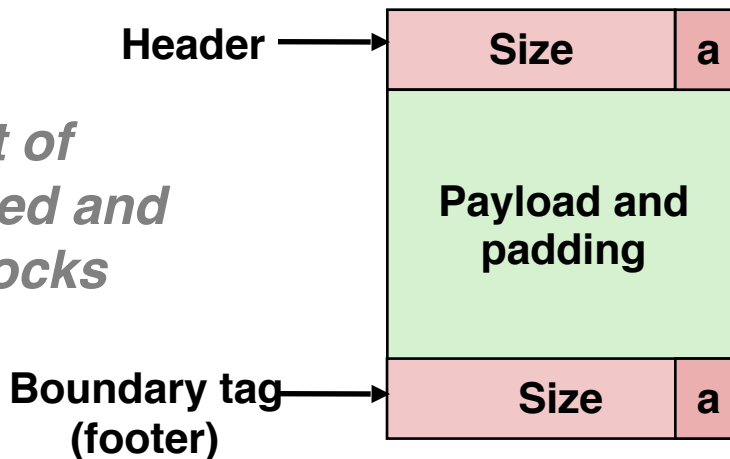
**Payload: Application data
(allocated blocks only)**

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

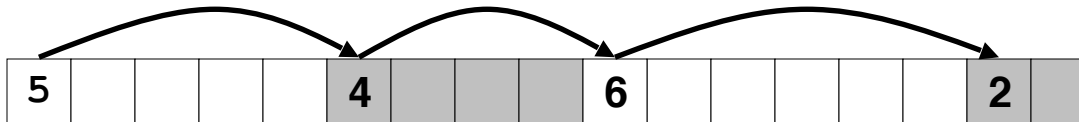
- Implementation: very simple
- Allocate cost:
 - **linear** time worst case
 - Identify free blocks requires scanning **all** the blocks!
- Free cost:
 - **constant** time worst case
- Memory usage:
 - Will depend on placement policy
 - First-fit, next-fit, or best-fit
- Not used in practice because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

Implicit Lists: Summary

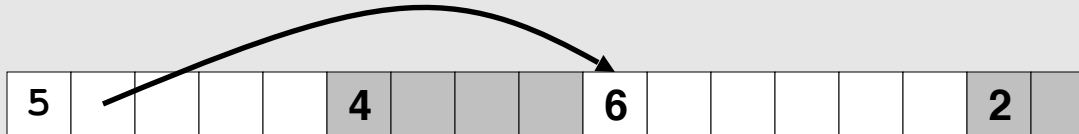
- Implementation: very simple
- Allocate cost:
 - **linear** time worst case
 - Identify free blocks requires scanning **all** the blocks!
- Free cost:
 - **constant** time worst case
- Memory usage:
 - Will depend on placement policy
 - First-fit, next-fit, or best-fit
- Not used in practice because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



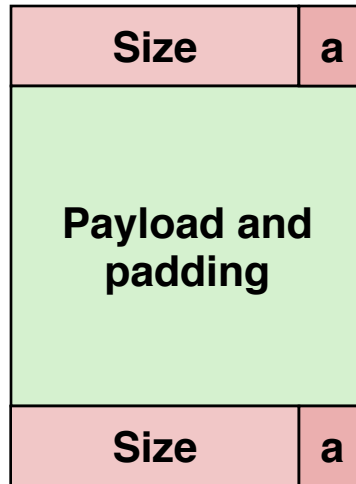
- Method 2: *Explicit list* among the free blocks using pointers



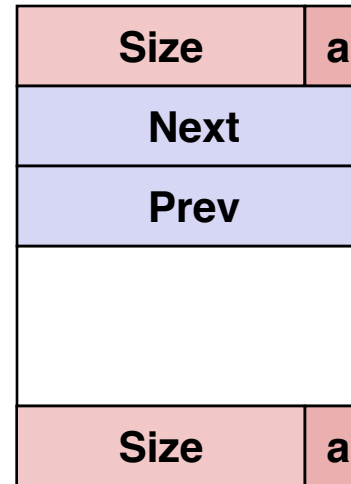
- Method 3: *Segregated free list*
 - Different free lists for different size classes

Explicit Free Lists

Allocated block
(same as before)



Free block



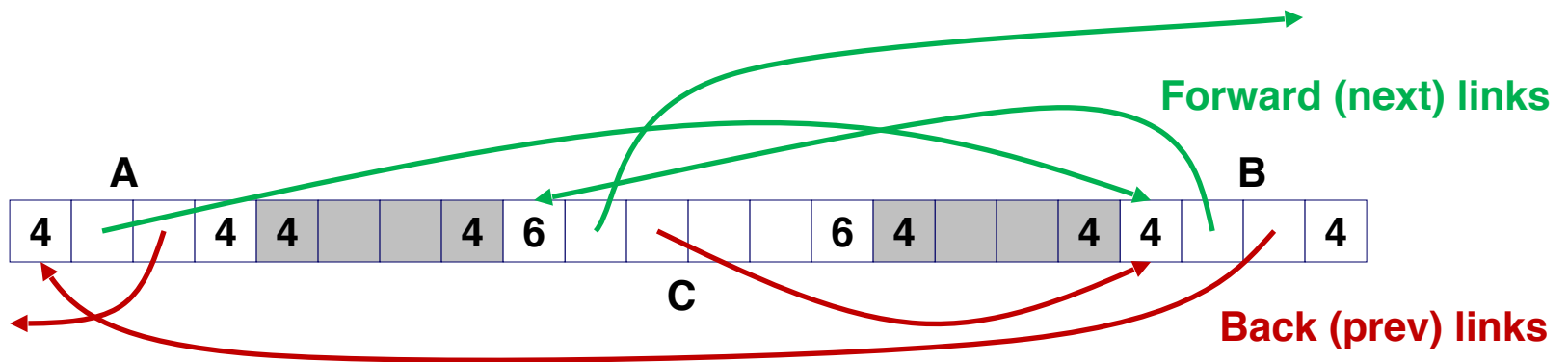
- Maintain list(s) of *free* blocks, not *all* blocks
 - The “next” free block could be anywhere
 - So we need to store forward/back pointers, not just sizes
 - These pointers exist only in free blocks, occupying the would-be payload area, so not really an overhead.
 - Still need boundary tags for coalescing.

Explicit Free Lists

- Logically:

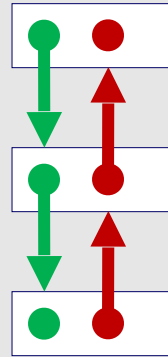


- Physically: blocks can be in any order



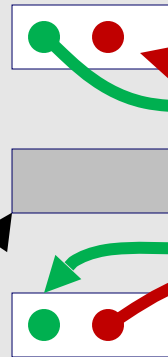
Allocating From Explicit Free Lists

Before



conceptual graphic

After



(with splitting)

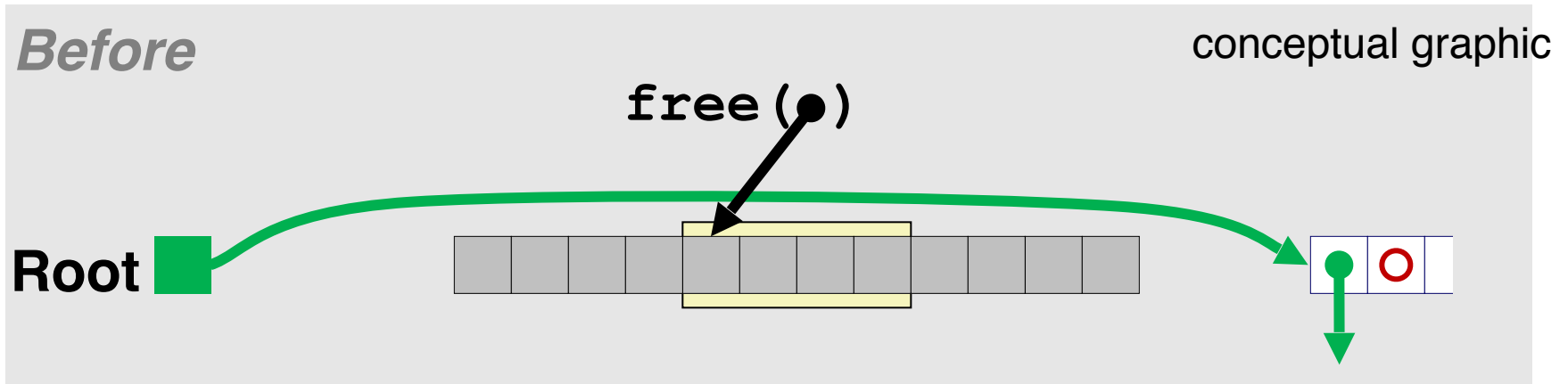
● = malloc(...)

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

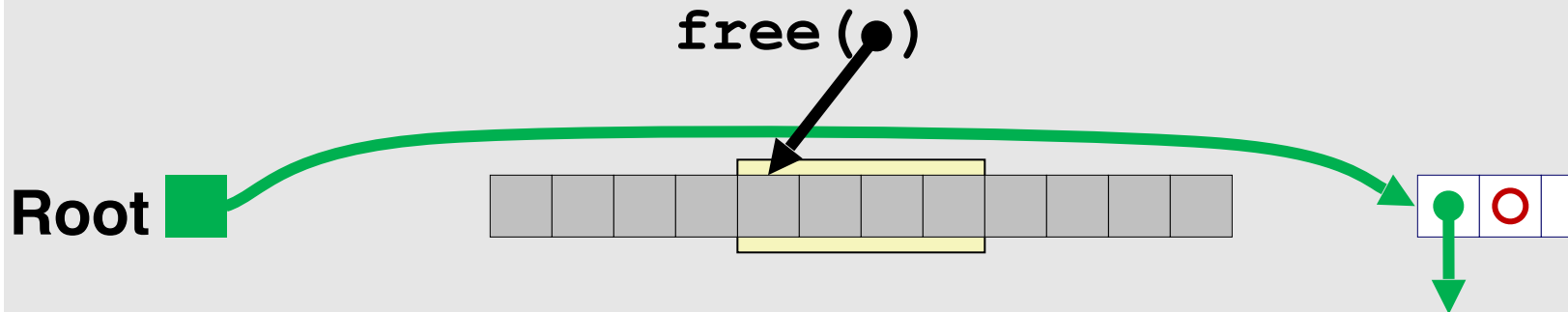


Freeing With a LIFO Policy (Case 1)

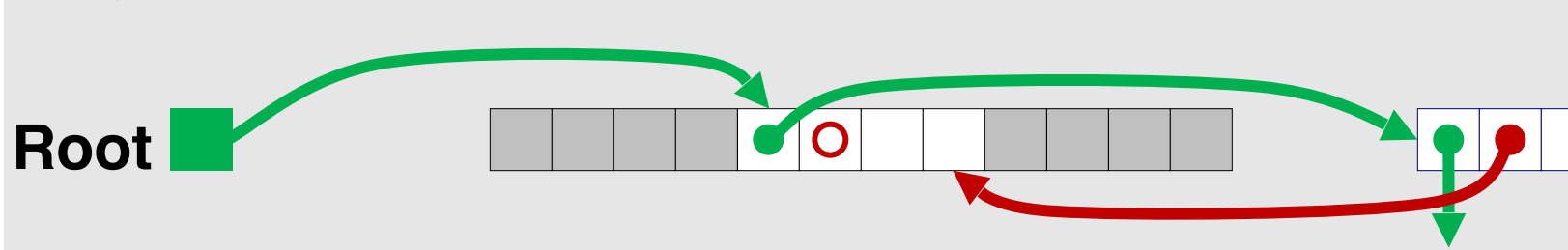
- Insert the freed block at the root of the list

Before

conceptual graphic

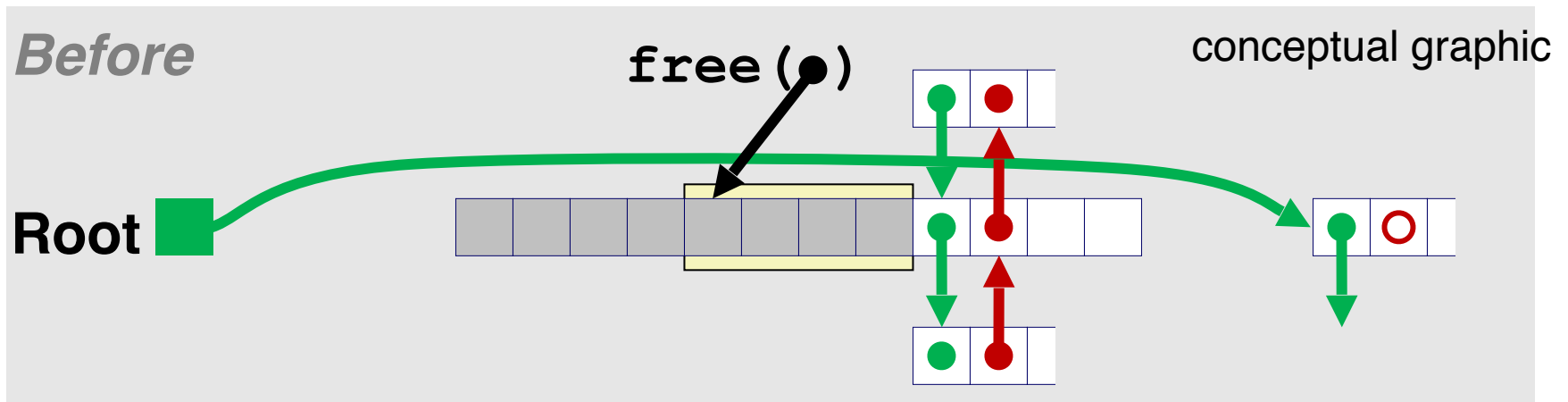


After



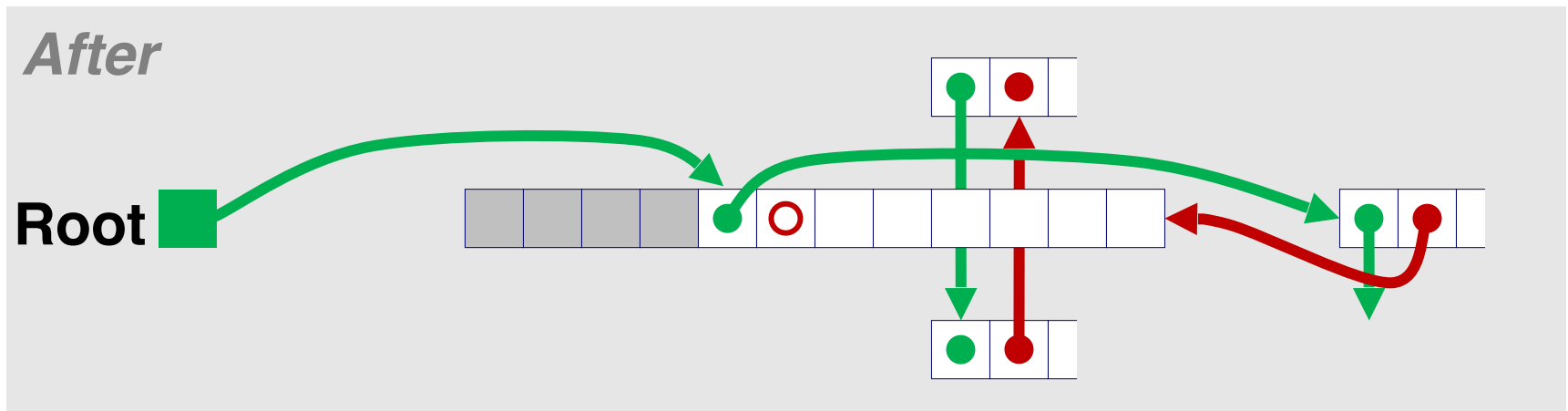
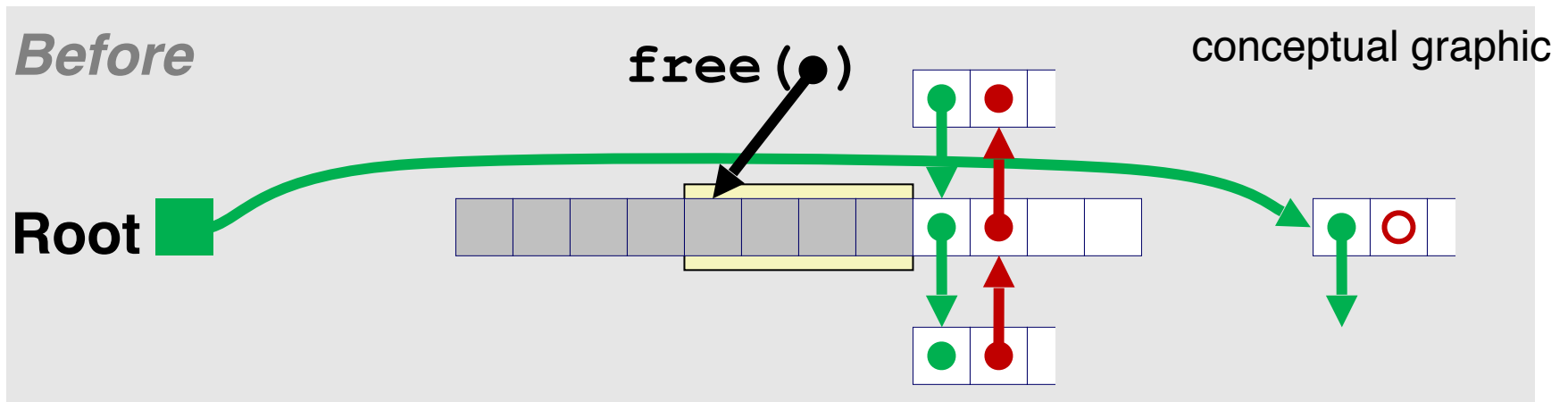
Freeing With a LIFO Policy (Case 2)

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



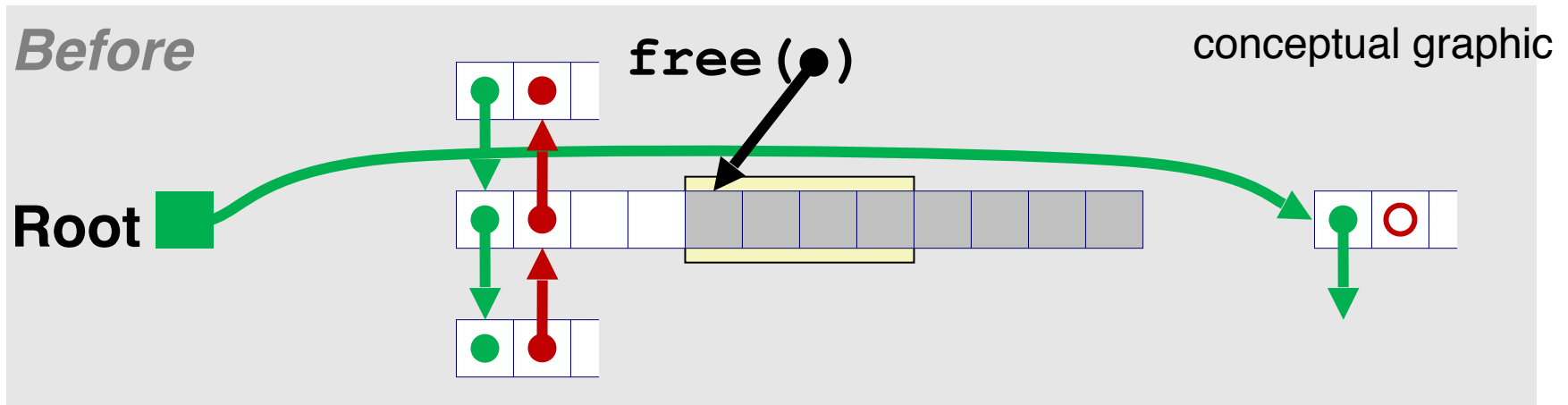
Freeing With a LIFO Policy (Case 2)

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



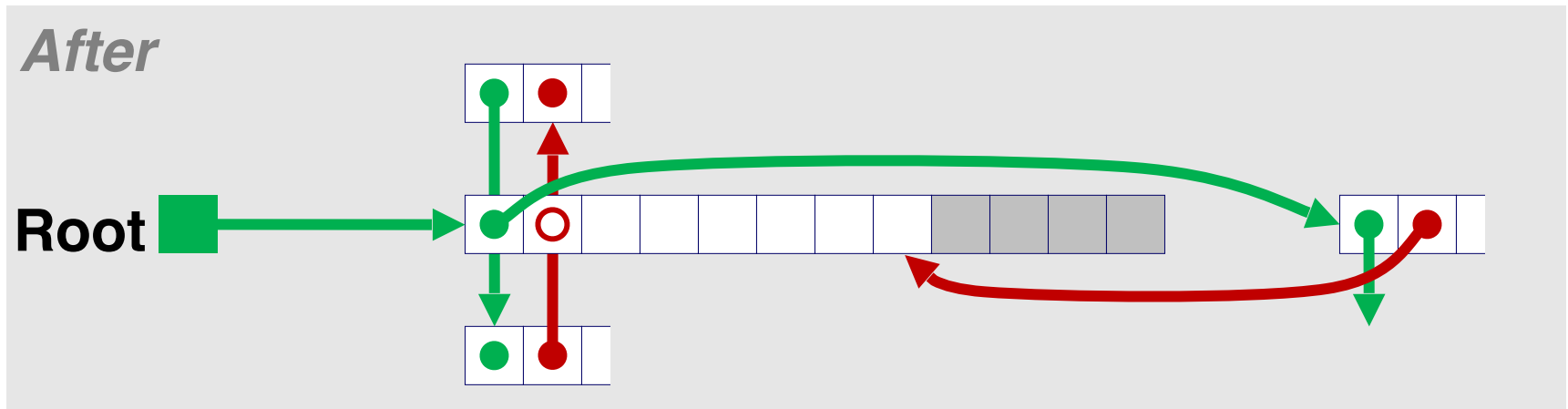
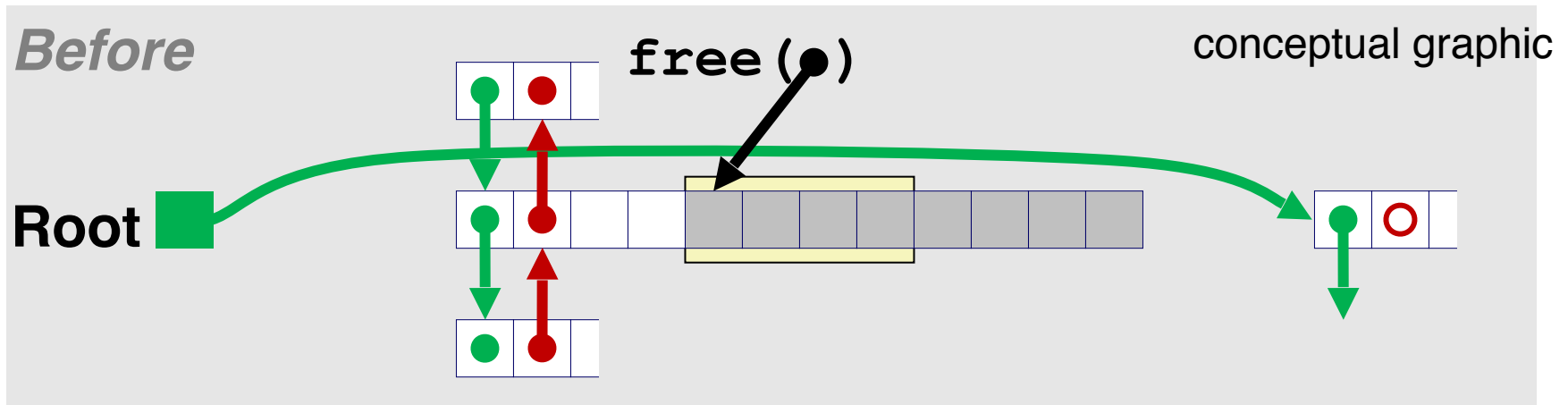
Freeing With a LIFO Policy (Case 3)

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



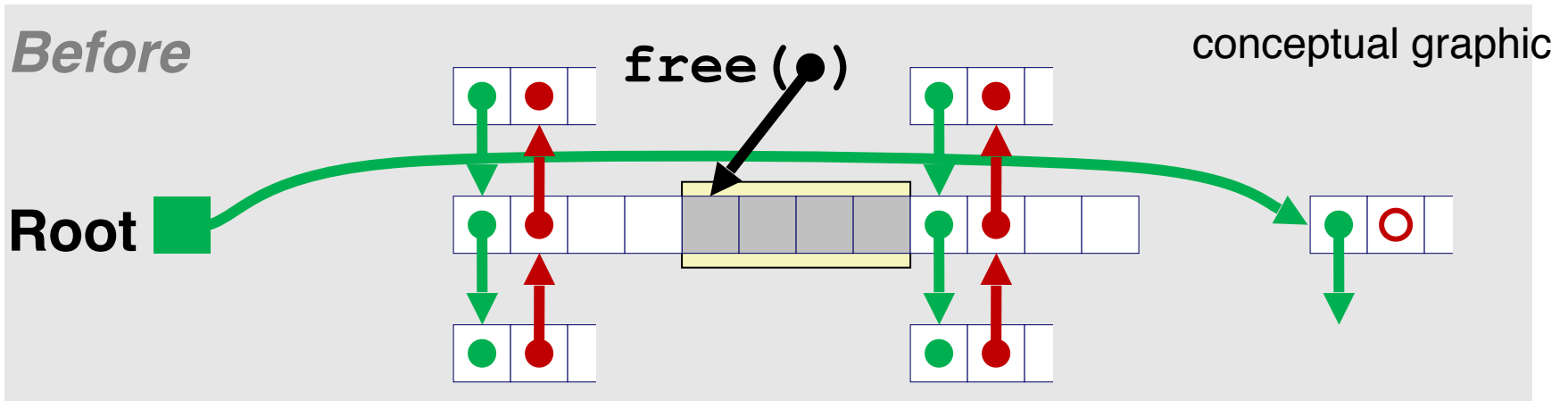
Freeing With a LIFO Policy (Case 3)

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



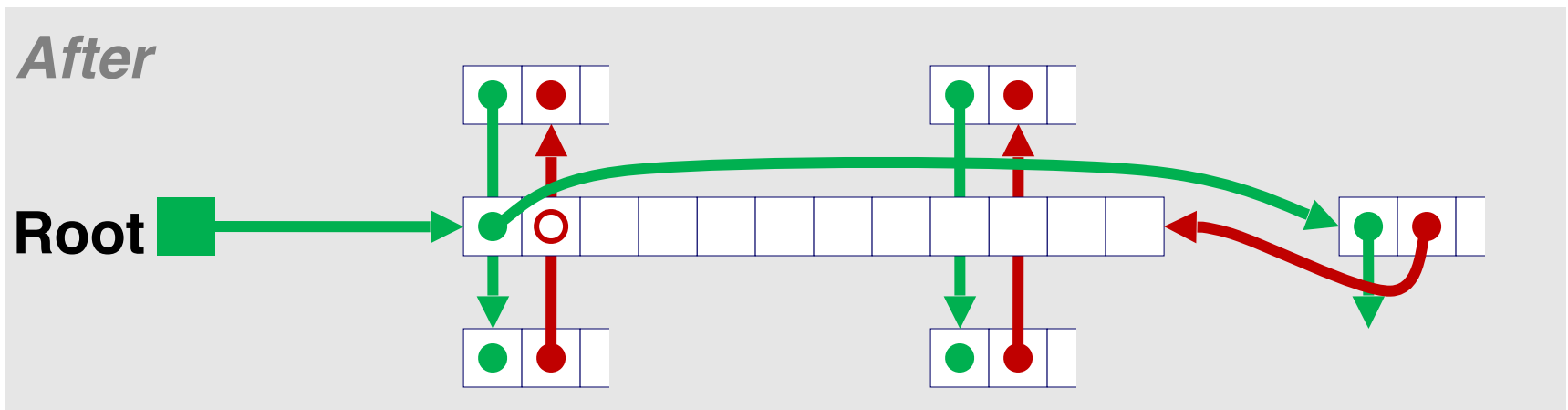
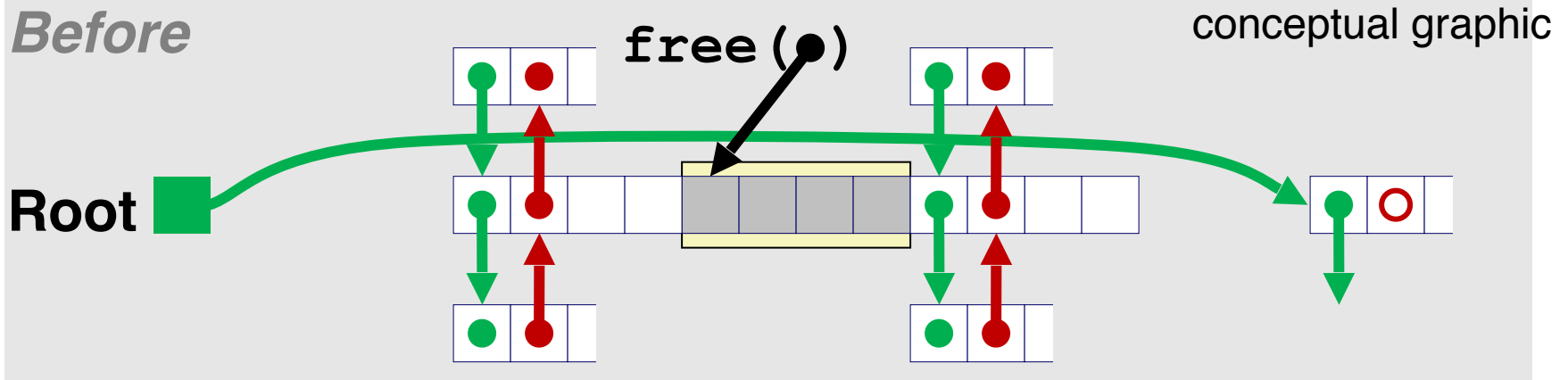
Freeing With a LIFO Policy (Case 4)

- 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

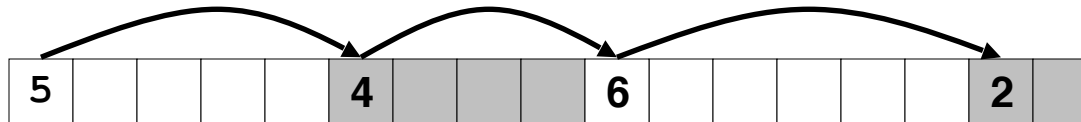


Explicit List Summary

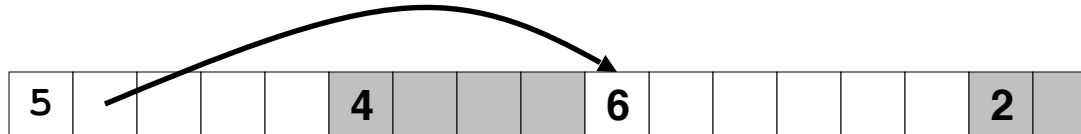
- Comparison to implicit list:
 - Allocate is linear time in number of **free** blocks instead of **all** blocks. Much faster when most of the memory is full.
 - Slightly more complicated allocate and free since needs to splice blocks in and out of the list
 - Some extra space for the links in free blocks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

Keeping Track of Free Blocks

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

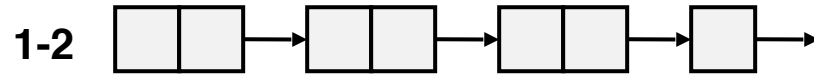


- Method 2: *Explicit list* among the free blocks using pointers



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

Segregated List (Seglist) Allocators



- Each *size class* of blocks has its own free list
- Organize the Seglist
 - Often have separate classes for each small size
 - For larger sizes: One class for each two-power size (why?)

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 $m \geq n$
 - 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
 - Constant time allocation and free for requests that have a dedicated free list (most of the cases)
 - log time for power-of-two size classes (searching the lists)
- 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.