

# Dynamic Memory Allocation

---

**Alan L. Cox**  
**alc@rice.edu**

**Some slides adapted from CMU 15.213 slides**

# Objectives

---

**Be able to analyze a memory allocator's performance**

- ♦ **Memory usage efficiency (fragmentation)**
- ♦ **Speed of allocation and deallocation operations**
- ♦ **Locality of allocations**
- ♦ **Robustness**

**Be able to implement your own efficient memory allocator (Malloc Project)**

**Be able to analyze the advantages and disadvantages of different garbage collector designs**

# Harsh Reality: Memory Matters

---

## Memory is not unbounded

- ♦ It must be allocated and managed
- ♦ Many applications are memory dominated
  - E.g., applications based on complex graph algorithms

## Memory referencing bugs especially pernicious

- ♦ Effects are distant in both time and space

## Memory performance is not uniform

- ♦ Cache and virtual memory effects can greatly affect program performance
- ♦ Adapting program to characteristics of memory system can lead to major speed improvements

# Memory Allocation

---

## Static size, static allocation

- ♦ Global variables
- ♦ Linker allocates final virtual addresses
- ♦ Executable stores these allocated addresses

## Static size, dynamic allocation

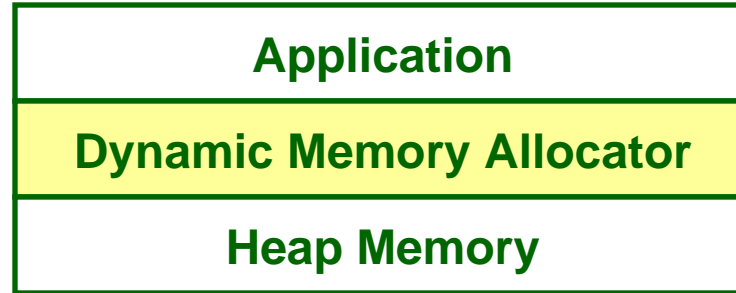
- ♦ Local variables
- ♦ Compiler directs stack allocation
- ♦ Stack pointer offsets stored directly in the code

## Dynamic size, dynamic allocation

- ♦ Programmer controlled
- ♦ Allocated in the heap - how?

# Dynamic Memory Allocation

---



## Explicit vs. implicit memory allocator

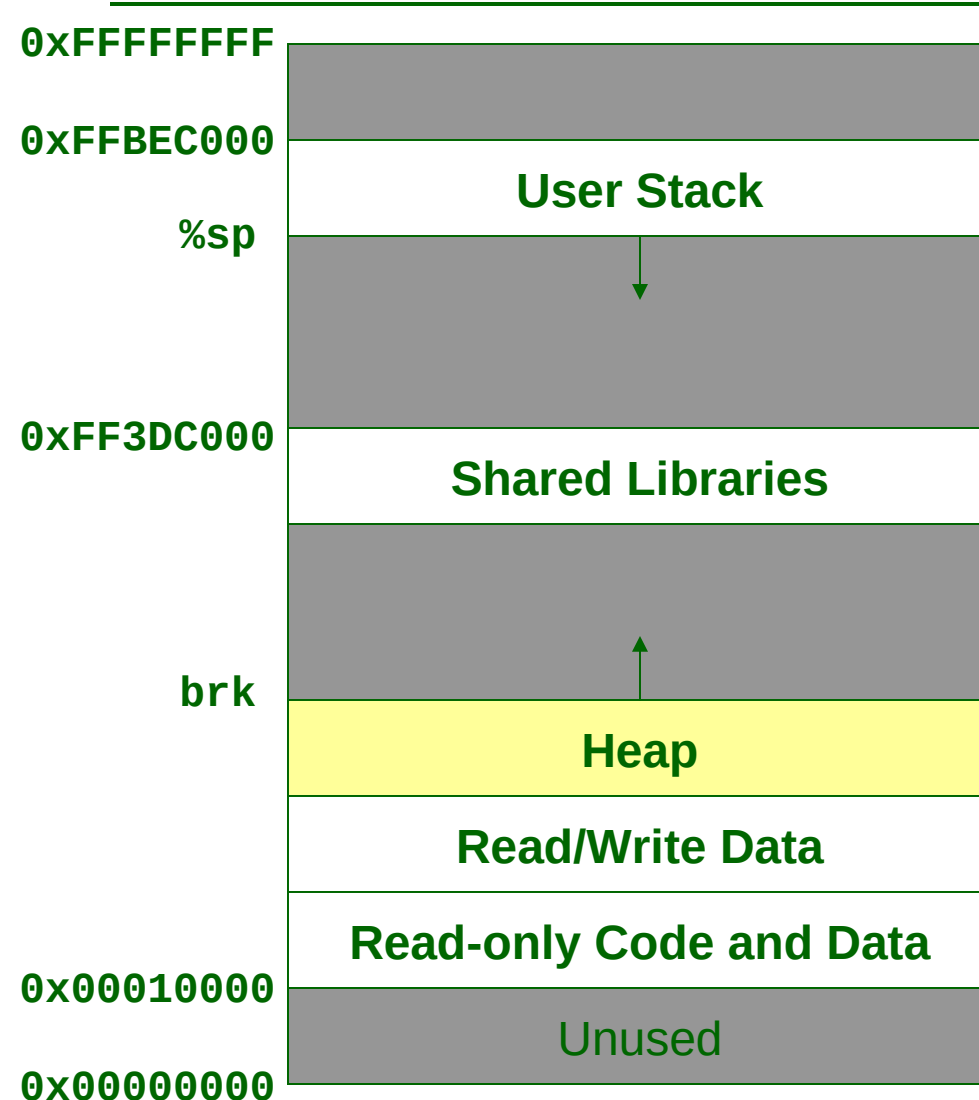
- ♦ **Explicit:** application allocates and frees space
  - e.g., malloc and free in C
- ♦ **Implicit:** application allocates, but does not free space
  - e.g., garbage collection in Java or Python

## Allocation

- ♦ In both cases the memory allocator provides an abstraction of memory as a set of blocks
- ♦ Doles out free memory blocks to application

**We will first discuss simple explicit memory allocation**

# Process Memory Image



`void *sbrk(int incr)`

- ♦ **Used by allocators to request additional memory from the OS**
- ♦ **brk initially set to the end of the data section**
- ♦ **Calls to sbrk increment brk by incr bytes (new virtual memory pages are demand-zeroed)**
- ♦ **incr can be negative to reduce the heap size**

# Malloc Package

---

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

```
void *malloc(size_t size)
```

- ♦ **If successful:**

- Returns a pointer to a memory block of at least size bytes, (typically) aligned to 8-byte boundary
- If size == 0, returns NULL

- ♦ **If unsuccessful: returns NULL (0) and sets errno**

```
void free(void *ptr)
```

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

```
void *realloc(void *ptr, size_t size)
```

- ♦ **Changes size of block pointed at by ptr and returns pointer to new block**
- ♦ **Contents of new block unchanged up to the minimum of the old and new sizes**

# malloc Example

---

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

    /* Allocate a block of n ints. */
    if ((p = malloc(n * sizeof(int))) == NULL) {
        perror("malloc");
        exit(1);
    }
    for (i = 0; i < n; i++)
        p[i] = i;
    /* Add m bytes to end of p block. */
    if ((p = realloc(p, (n + m) * sizeof(int))) == NULL) {
        perror("realloc");
        exit(1);
    }
    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]);
    /* Return p to available memory pool. */
    free(p);
}
```

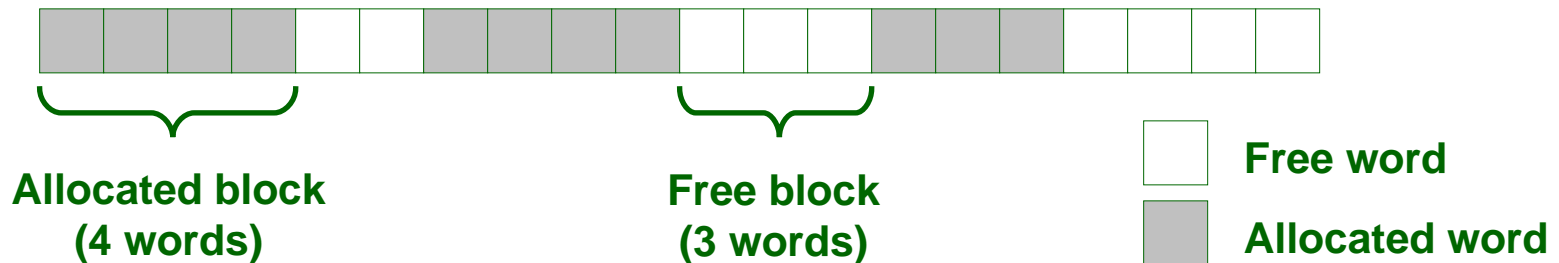


# Assumptions

---

## Conventions used in these lectures

- ♦ Memory is word addressed
- ♦ “Boxes” in figures represent a word
- ♦ Each word can hold an integer or a pointer



# Allocation Examples

---

`p1 = malloc(4*sizeof(int))`



`p2 = malloc(5*sizeof(int))`



`p3 = malloc(6*sizeof(int))`



`free(p2)`



`p4 = malloc(2*sizeof(int))`



# Constraints

---

## Applications:

- ♦ Can issue arbitrary sequence of malloc and free requests
- ♦ Free requests must correspond to an allocated block

## Allocators

- ♦ Can't control number or size of allocated blocks
- ♦ Must respond immediately to all allocation requests
  - i.e., can't reorder or buffer requests
- ♦ Must allocate blocks from free memory
  - i.e., can only place allocated blocks in free memory
- ♦ Must align blocks so they satisfy all alignment requirements
  - 8 byte alignment for libc malloc on many systems
- ♦ Can only manipulate and modify free memory
- ♦ Can't move the allocated blocks once they are allocated
  - i.e., compaction is not allowed

# Goals of Good malloc/free

---

## Primary goals

- ♦ **Good time performance for malloc and free**
  - Ideally should take constant time (not always possible)
  - Should certainly not take linear time in the number of blocks
- ♦ **Good space utilization**
  - User allocated structures should use most of the heap
  - Want to minimize “fragmentation”

## Some other goals

- ♦ **Good locality properties**
  - Structures allocated close in time should be close in space
  - “Similar” objects should be allocated close in space
- ♦ **Robust**
  - Can check that `free(p1)` is on a valid allocated object `p1`
  - Can check that memory references are to allocated space

# Maximizing Throughput

---

**Given some sequence of malloc and free requests:**

- ♦  $R_0, R_1, \dots, R_k, \dots, R_{n-1}$

**Want to 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

# Maximizing 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 is denoted by  $H_k$**

- ♦ Assume that  $H_k$  is monotonically increasing

**Def: Peak memory utilization:**

- ♦ After  $k$  requests, peak memory utilization is:
  - $U_k = ( \max_{i < k} P_i ) / H_k$

# Internal Fragmentation

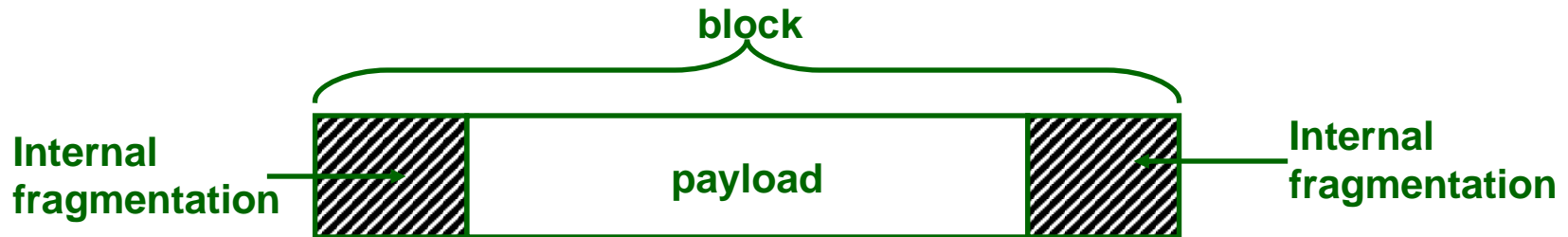
---

## Poor memory utilization caused by fragmentation

- ♦ Comes in two forms: internal and external fragmentation

## Internal fragmentation

- ♦ For some block, internal fragmentation is the difference between the block size and the payload size



- ♦ Caused by overhead of maintaining heap data structures, padding for alignment purposes, or explicit policy decisions (e.g., not to split the block)
- ♦ Depends only on the pattern of *previous* requests, and thus is easy to measure

# External Fragmentation

---

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

```
p1 = malloc(4*sizeof(int))
```



```
p2 = malloc(5*sizeof(int))
```



```
p3 = malloc(6*sizeof(int))
```



```
free(p2)
```



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

oops!

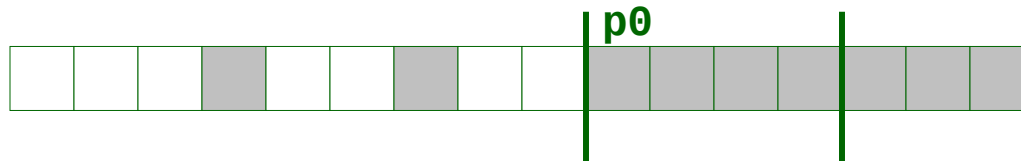
**External fragmentation depends on the pattern of future requests, and thus is difficult to measure**



# Implementation Issues

---

- ♦ How do we know how much memory to free just given a pointer?
- ♦ How do we keep track of the free blocks?
- ♦ 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?
- ♦ How do we reinsert a freed block?



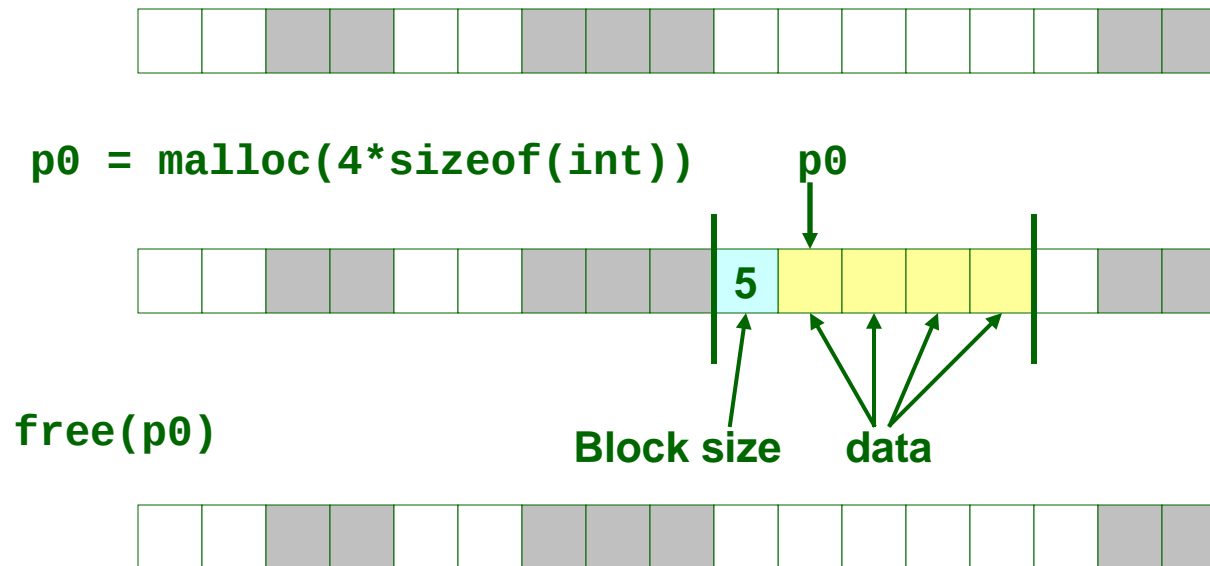
`free(p0)`

`p1 = malloc(1)`

# 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 lengths - links all blocks



## Method 2: Explicit list among the free blocks using pointers within the free blocks



## Method 3: Segregated free list

- ♦ Different free lists for different size classes

## Method 4: Blocks sorted by size

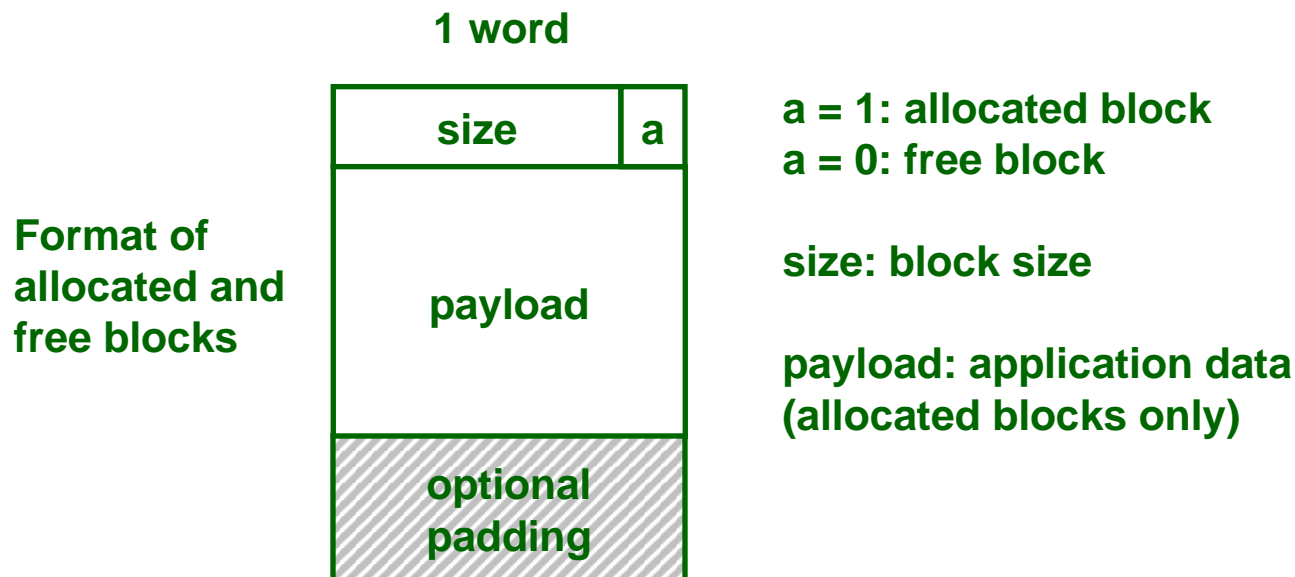
- ♦ Can use a balanced tree (e.g., Red-Black tree) with pointers within each free block, and the length used as a key

# Method 1: Implicit List

---

## Need to identify whether each block is free or allocated

- ♦ Can use extra bit
- ♦ Bit can be put in the same word as the size if block sizes are always multiples of two (mask out low order bit when reading size)



# Implicit List: Finding a Free Block

---

## First fit:

- ♦ Search list from beginning, choose first free block that fits

```
p = start;
while ((p < end) &&      \\ not past end
      ((*p & 1) ||      \\ already allocated
      (*p <= len)))    \\ too small
  p = NEXT_BLK(p);
```

- ♦ Can take linear time in total number of blocks (allocated/free)
- ♦ Can cause “splinters” (small free blocks) at beginning of list

## Next fit:

- ♦ Like first-fit, but search list from end of previous search
- ♦ Research suggests that fragmentation is worse

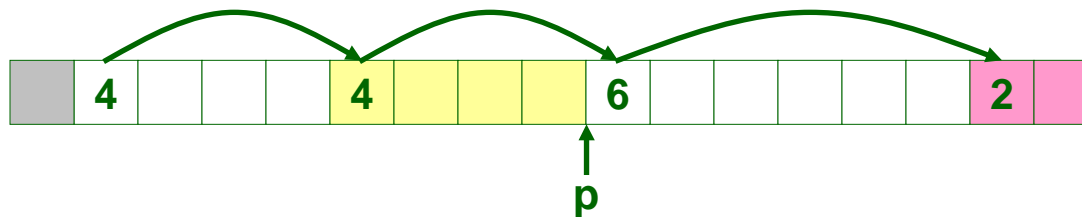
## Best fit:

- ♦ Choose the free block with the closest size that fits (requires complete search of the list)
- ♦ Keeps fragments small - usually helps fragmentation
- ♦ Will typically run slower than first-fit

# Implicit List: Allocating in Free Block

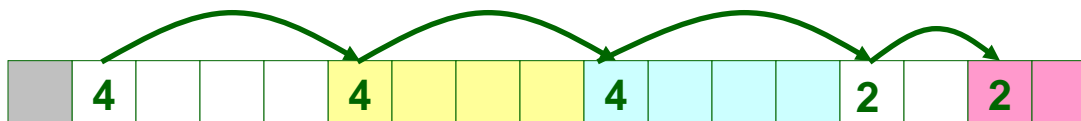
## Allocating in a free block - splitting

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



```
void addblock(ptr p, int len) {  
    int newsize = ((len + 1) >> 1) << 1; // add 1 and round up  
    int oldsize = *p & ~0x1; // mask out low bit  
    *p = newsize | 0x1; // set new length  
    if (newsize < oldsize)  
        *(p+newsize) = oldsize - newsize; // set length in remaining  
}
```

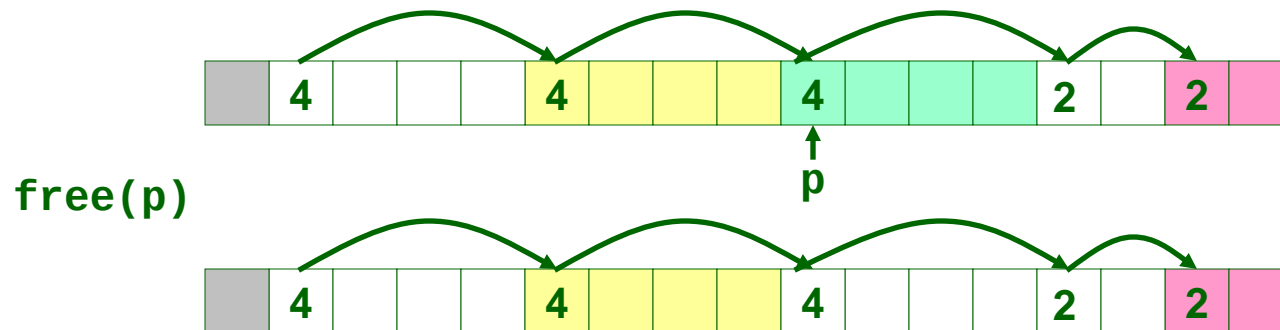
addblock(p, 4)



# Implicit List: Freeing a Block

## Simplest implementation:

- ♦ Only need to clear allocated flag
  - `void free_block(ptr p) { *p = *p & ~0x1 }`
- ♦ But can lead to “false fragmentation”



`malloc(5*sizeof(int))`

Oops!

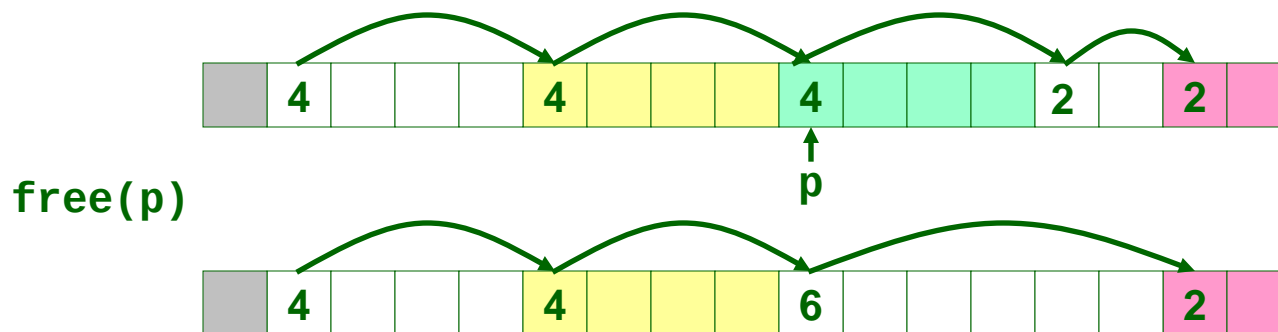
- ♦ There is enough free space, but the allocator won't be able to find it!

# Implicit List: Coalescing

**Join (coalesce) with next and/or previous block if they are free**

- ♦ **Coalescing with next block**

```
void free_block(ptr p) {  
    *p = *p & ~0x1;           // clear allocated flag  
    next = p + *p;             // find next block  
    if ((*next & 0x1) == 0)  
        *p = *p + *next;      // add to this block if  
                                // not allocated  
}
```



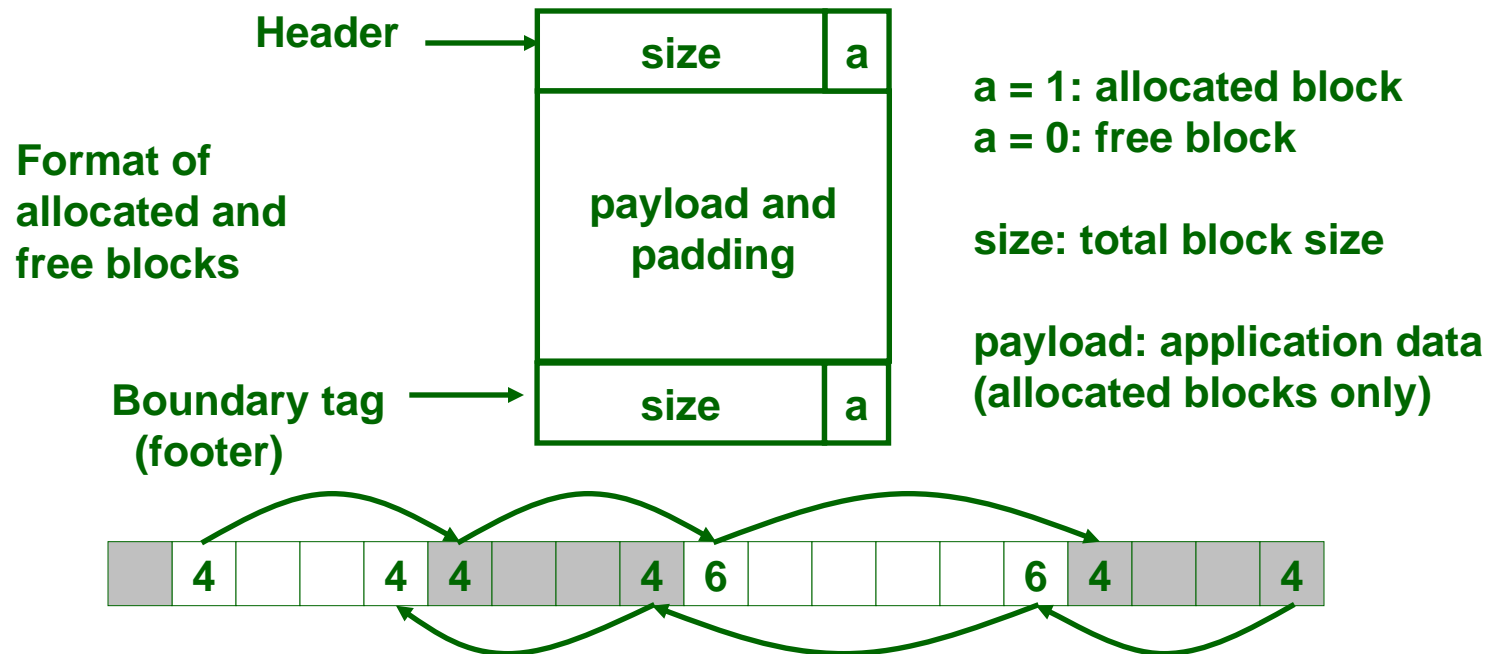
- ♦ **But how do we coalesce with previous block?**



# Implicit List: Bidirectional Coalescing

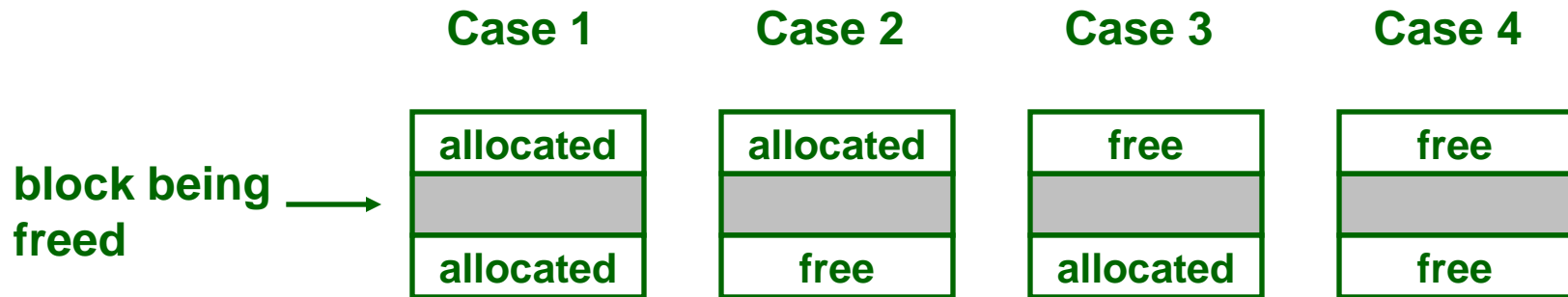
## Boundary tags [Knuth73]

- ♦ Replicate header word at end of block
- ♦ Allows us to traverse the “list” backwards, but requires extra space
- ♦ Important and general technique!



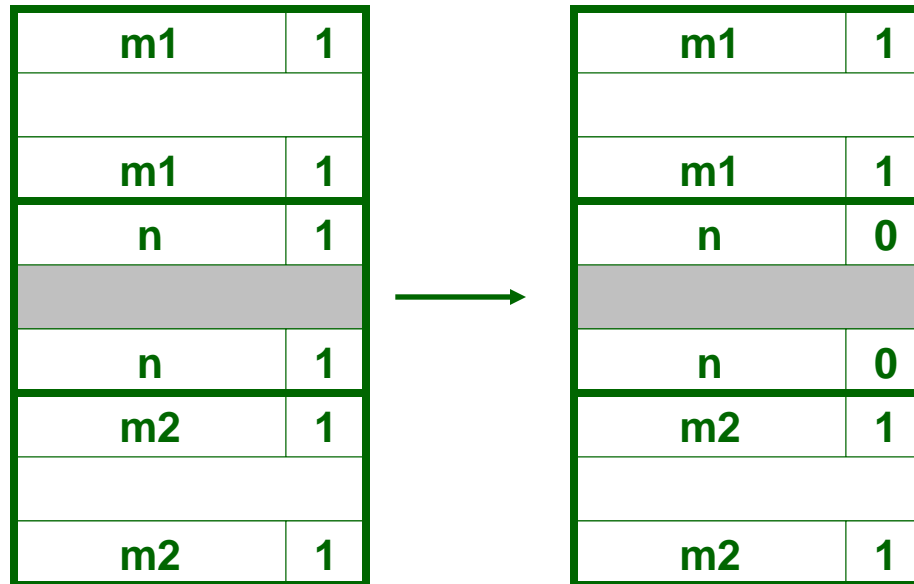
# Constant Time Coalescing

---



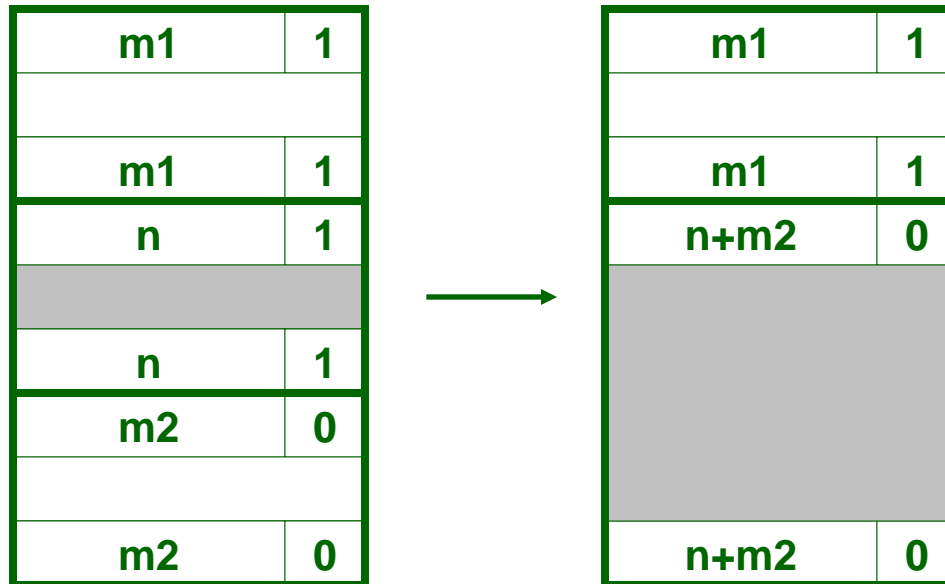
# Constant Time Coalescing (Case 1)

---



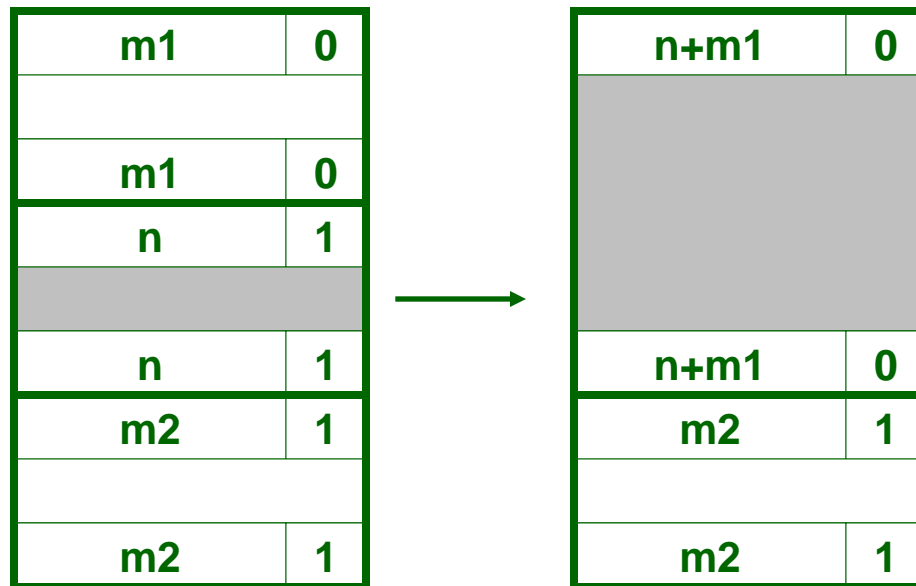
# Constant Time Coalescing (Case 2)

---



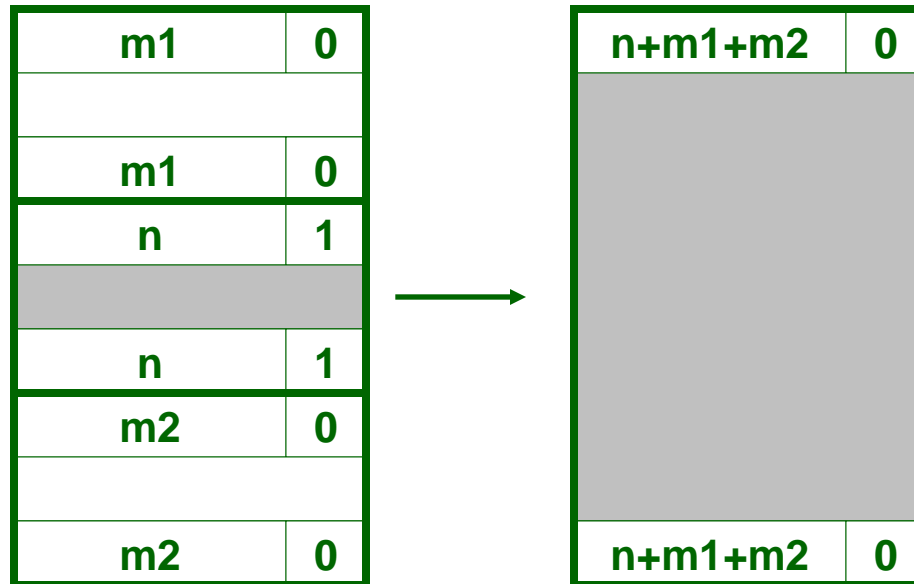
# Constant Time Coalescing (Case 3)

---



# Constant Time Coalescing (Case 4)

---



# 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 go ahead and split free blocks?
- ♦ How much internal fragmentation are we willing to tolerate?

## Coalescing policy:

- ♦ Immediate coalescing: coalesce adjacent blocks each time free is called
- ♦ Deferred coalescing: try to improve performance of free by deferring coalescing until needed. e.g.,
  - 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**

**Allocate: linear time worst case**

**Free: constant time worst case - even with coalescing**

**Memory usage: will depend on placement policy**

- ♦ **First fit, next fit or best fit**

**Not used in practice for malloc/free because of linear time allocate**

- ♦ **Used in many special purpose applications**

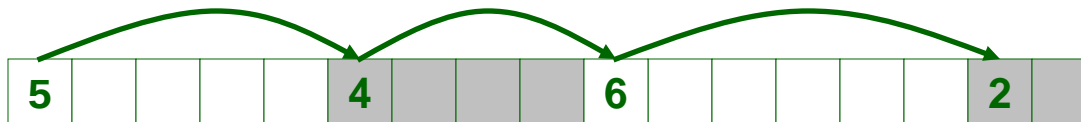
**However, the concepts of splitting and boundary tag coalescing are general to all allocators**



# Keeping Track of Free Blocks

---

## Method 1: Implicit list using lengths - links all blocks



## Method 2: Explicit list among the free blocks using pointers within the free blocks



## Method 3: Segregated free list

- ♦ Different free lists for different size classes

## Method 4: Blocks sorted by size

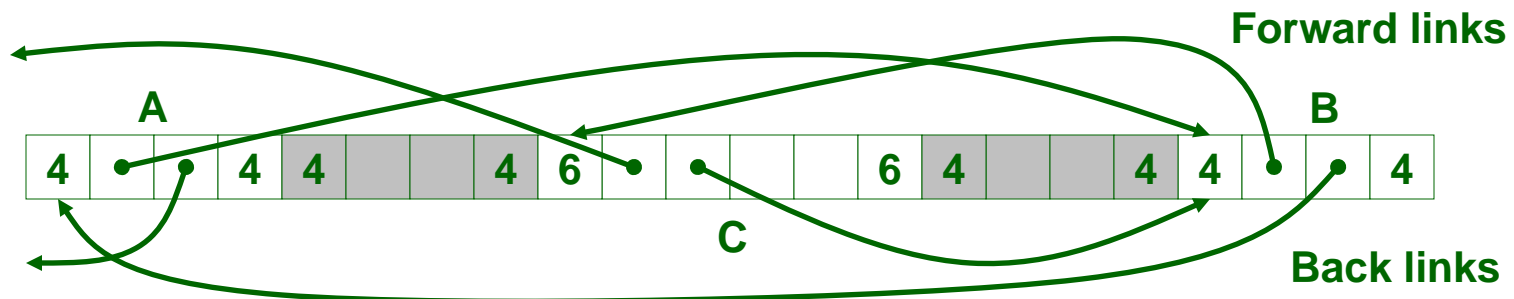
- ♦ Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

# Explicit Free Lists



## Use data space for link pointers

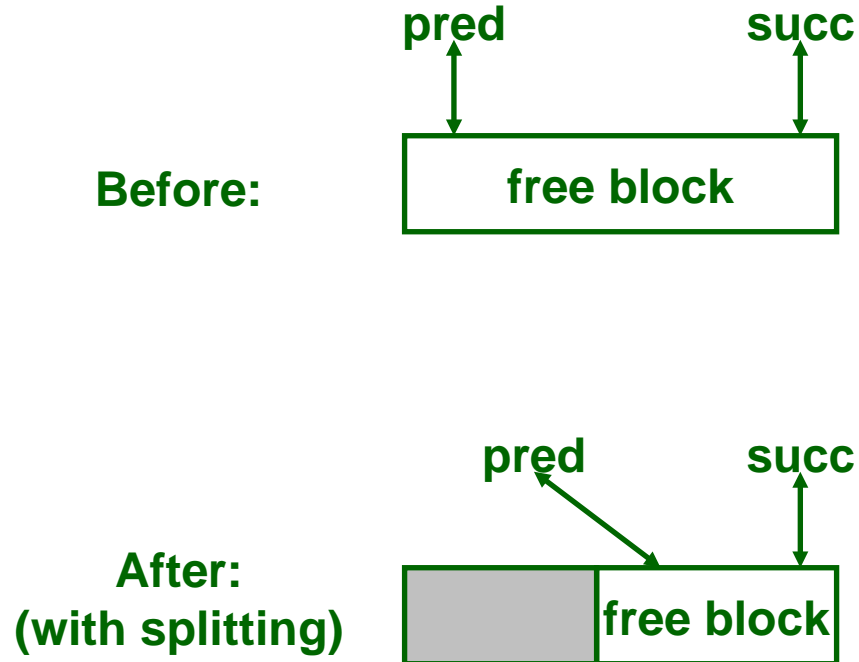
- ♦ Typically doubly linked
- ♦ Still need boundary tags for coalescing



- ♦ It is important to realize that links are not necessarily in the same order as the blocks

# 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

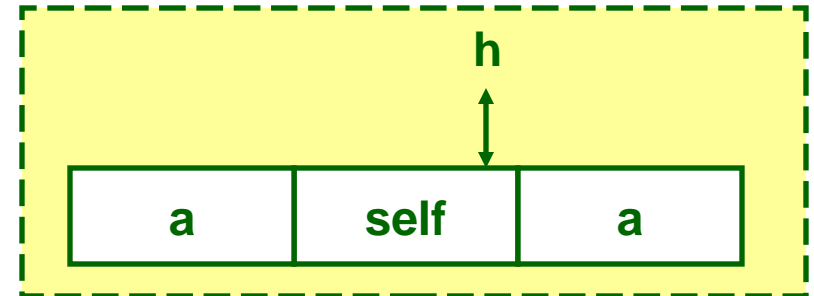
### ♦ Address-ordered policy

- Insert freed blocks so that free list blocks are always in address order
  - i.e.  $\text{addr}(\text{pred}) < \text{addr}(\text{curr}) < \text{addr}(\text{succ})$
- Con: requires search
- Pro: studies suggest fragmentation is better than LIFO

# Freeing With a LIFO Policy

## Case 1: a-a-a

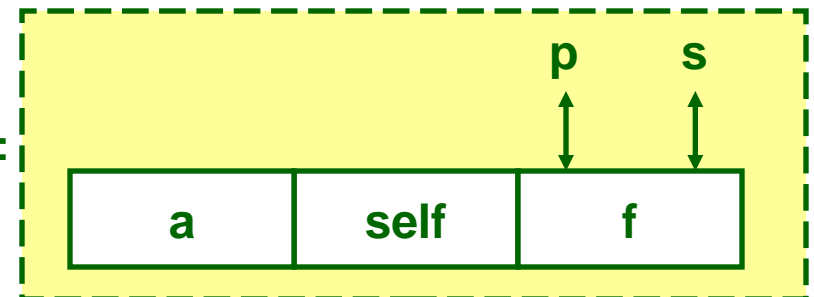
- ♦ Insert self at beginning of free list



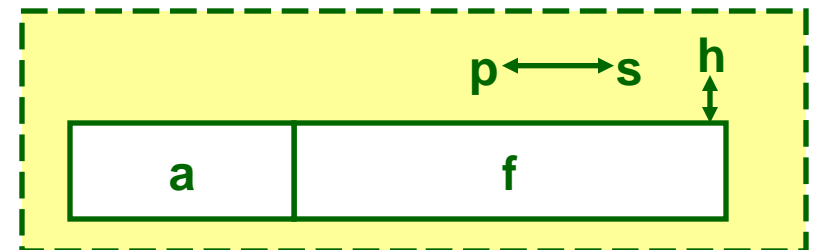
## Case 2: a-a-f

- ♦ Splice out next, coalesce self and next, and add to beginning of free list

before:



after:



# Freeing With a LIFO Policy (cont)

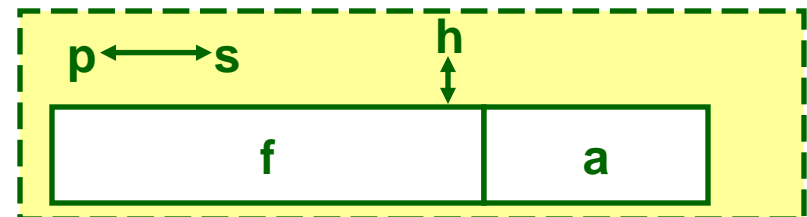
## Case 3: f-a-a

- ♦ Splice out prev, coalesce with self, and add to beginning of free list

before:



after:



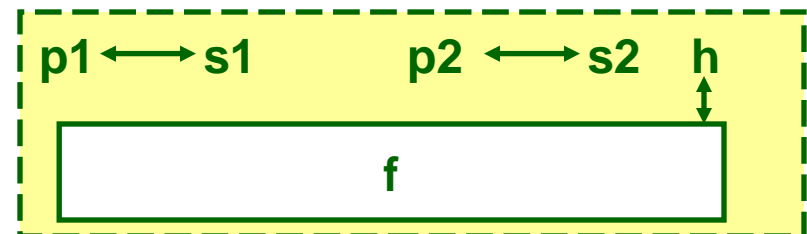
## Case 4: f-a-f

- ♦ Splice out prev and next, coalesce with self, and add to beginning of list

before:



after:



# Explicit List Summary

---

## Comparison to implicit list:

- ♦ Allocate is linear time in number of free blocks instead of total blocks - much faster allocates 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)

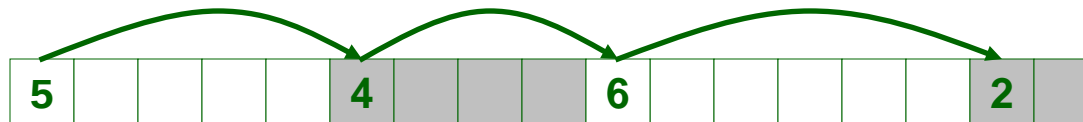
## Main use of linked lists is in conjunction with segregated free lists

- ♦ 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 lengths - links all blocks



## Method 2: Explicit list among the free blocks using pointers within the free blocks



## Method 3: Segregated free list

- ♦ Different free lists for different size classes

## Method 4: Blocks sorted by size

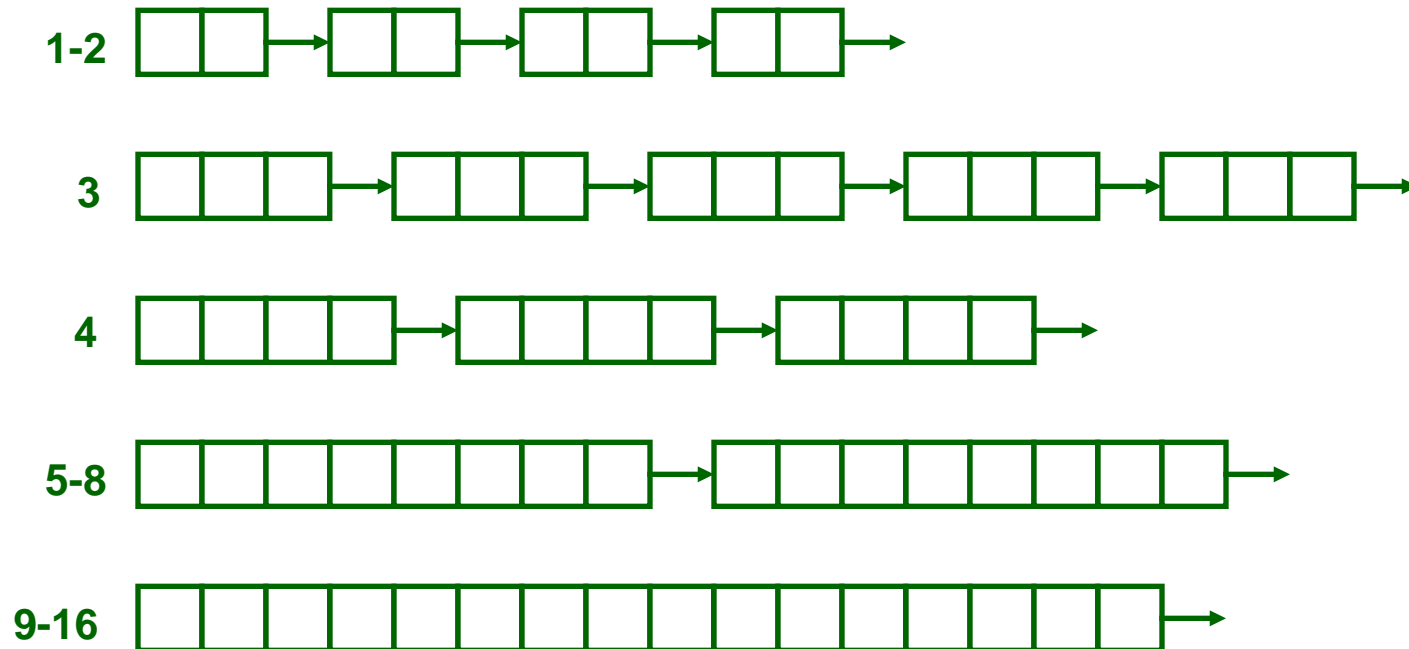
- ♦ Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key



# Segregated Storage

---

**Each size class has its own collection of blocks**



- ♦ **Often separate classes for every small size (2,3,4,...)**
- ♦ **Larger sizes typically grouped into powers of 2**

# Simple Segregated Storage

---

**Separate heap and free list for each size class**

**No splitting**

**To allocate a block of size n:**

- ♦ **If free list for size n is not empty,**
  - Allocate first block on list (list can be implicit or explicit)
- ♦ **If free list is empty,**
  - Get a new page
  - Create new free list from all blocks in page
  - Allocate first block on list
- ♦ **Constant time**

**To free a block:**

- ♦ **Add to free list**
- ♦ **If page is empty, could return the page for use by another size**

**Tradeoffs:**

- ♦ **Fast, but can fragment badly**
- ♦ **Interesting observation: approximates a best fit placement policy without having the search entire free list**

# Segregated Fits

---

**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

**To free a block:**

- ♦ Coalesce and place on appropriate list (optional)

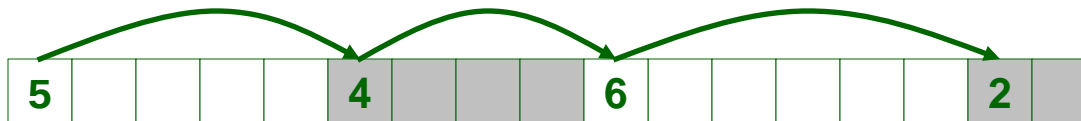
**Tradeoffs**

- ♦ Faster search than sequential fits (i.e., log time for power of two size classes)
- ♦ Controls fragmentation of simple segregated storage
- ♦ Coalescing can increase search times
  - Deferred coalescing can help

# Keeping Track of Free Blocks

---

## Method 1: Implicit list using lengths - links all blocks



## Method 2: Explicit list among the free blocks using pointers within the free blocks



## 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

# Spatial Locality

---

**Most techniques give little control over spatial locality**

- ♦ **Sequentially-allocated blocks not necessarily adjacent**
- ♦ **Similarly-sized blocks (e.g., for same data type) not necessarily adjacent**

**Would like a series of similar-sized allocations and deallocations to reuse same blocks**

- ♦ **Splitting & coalescing tend to reduce locality**

**Of techniques seen, which best for spatial locality?**

**Simple segregated lists  
Each page only has similar-sized blocks**

# Spatial Locality: Regions

---

**One technique to improve spatial locality**

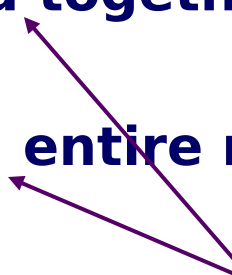
**Dynamically divide heap into mini-heaps**

- ♦ **Programmer-determined**

**Allocate data within appropriate region**

- ♦ **Data that is logically used together**
- ♦ **Increase locality**
- ♦ **Can quickly deallocate an entire region at once**

**Changes API  
malloc() and free()  
must take a region  
as an argument**



# For More Info on Allocators

---

**D. Knuth, “The Art of Computer Programming, Second Edition”, Addison Wesley, 1973**

- ♦ **The classic reference on dynamic storage allocation**

**Wilson et al, “Dynamic Storage Allocation: A Survey and Critical Review”, Proc. 1995 Int’l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.**

- ♦ **Comprehensive survey**
- ♦ **Available from CS:APP student site ([csapp.cs.cmu.edu](http://csapp.cs.cmu.edu))**

# Implementation Summary

---

## Many options:

- ♦ **Data structures for keeping track of free blocks**
- ♦ **Block choice policy**
- ♦ **Splitting & coalescing policies**

## No clear best option

- ♦ **Many tradeoffs**
- ♦ **Some behaviors not well understood by anyone**
- ♦ **Depends on “typical” program’s pattern of allocation and deallocation**



# Explicit Memory Allocation/Deallocation

---

- + Usually low time- and space-overhead**
- Challenging to use correctly by programmers**
  - Lead to crashes, memory leaks, etc.**

# Implicit Memory Deallocation

---

- + Programmers don't need to free data explicitly, easy to use**
- + Some implementations could achieve better spatial locality and less fragmentation in the hands of your average programmers**
- Price to pay: depends on implementation**

**But HOW could a memory manager know when to deallocate data without instruction from programmer?**

# Implicit Memory Management: Garbage Collection

---

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

```
void foo() {  
    int *p = 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++**

- ♦ **Cannot collect all garbage**

# Garbage Collection

---

## How does the memory manager know when memory can be freed?

- ♦ In general we cannot know what is going to be used in the future since it depends on conditionals
- ♦ But we can tell that certain blocks cannot be used if there are no pointers to them

## Need to make certain assumptions about pointers

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

# Classical GC algorithms

---

## Reference counting (Collins, 1960)

- ♦ Does not move blocks

## Mark and sweep collection (McCarthy, 1960)

- ♦ Does not move blocks (unless you also “compact”)

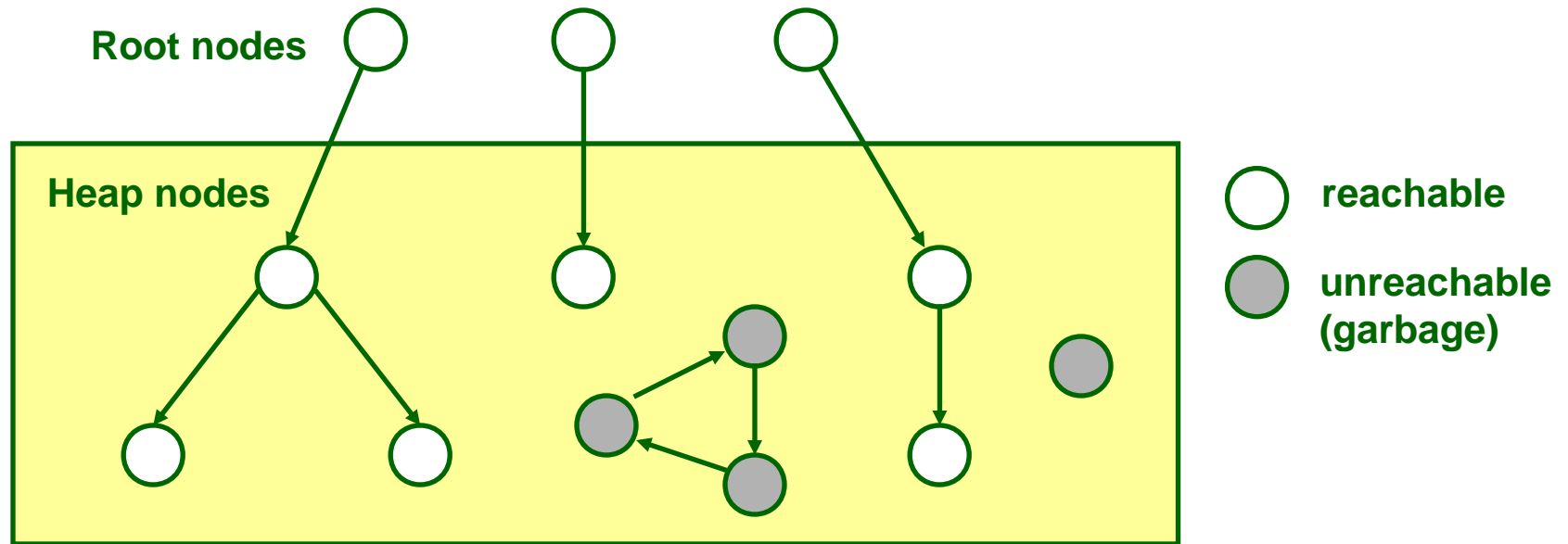
## Copying collection (Minsky, 1963)

- ♦ Moves blocks (compacts memory)

For more information, see Jones and Lin,  
“Garbage Collection: Algorithms for  
Automatic Dynamic Memory”, John Wiley &  
Sons, 1996.

# Memory as a Graph

- ♦ Each data block is a node in the graph
- ♦ Each pointer is an edge in the graph
- ♦ Root nodes: locations not in the heap that contain pointers into the heap (e.g. registers, locations on the stack, global variables)



# Reference Counting

---

## Overall idea

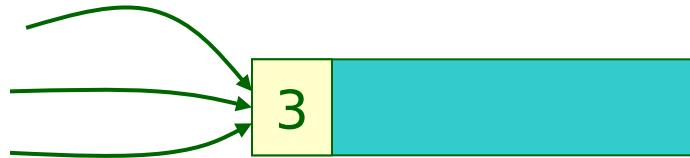
- ♦ **Maintain a free list of unallocated blocks**
- ♦ **Maintain a count of the number of references to each allocated block**
- ♦ **To allocate, grab a sufficiently large block from the free list**
- ♦ **When a count goes to zero, deallocate it**

# Reference Counting: More Details

---

**Each allocated block keeps a count of references to the block**

- ♦ Reachable  $\square$  count is positive
- ♦ Compiler inserts counter increments and decrements as necessary
- ♦ Deallocate when count goes to zero



**Typically built on top of an explicit deallocation memory manager**

- ♦ All the same implementation decisions as before
- ♦ E.g., splitting & coalescing



# Reference Counting: Example

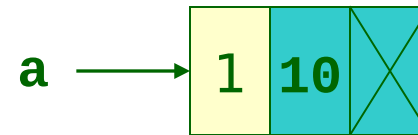
---

```
a = cons(10, empty)
b = cons(20, a)
a = b
b = ...
a = ...
```

# Reference Counting: Example

---

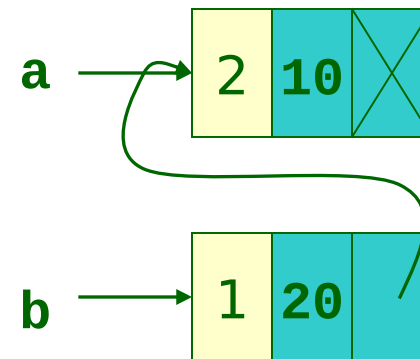
```
a = cons(10, empty)  
b = cons(20, a)  
a = b  
b = ...  
a = ...
```



# Reference Counting: Example

---

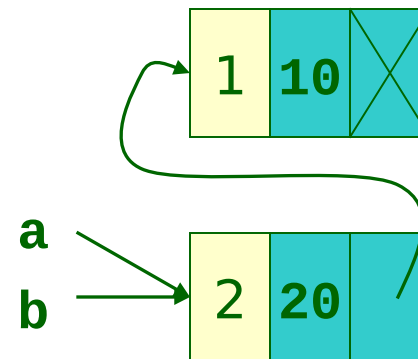
```
a = cons(10, empty)
b = cons(20, a)
a = b
b = ...
a = ...
```



# Reference Counting: Example

---

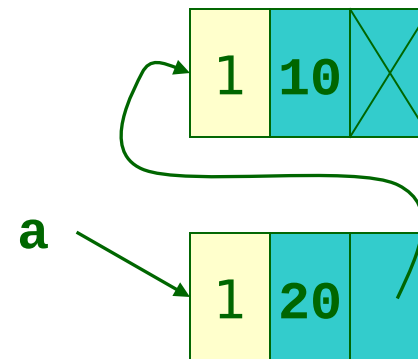
```
a = cons(10, empty)
b = cons(20, a)
a = b
b = ...
a = ...
```



# Reference Counting: Example

---

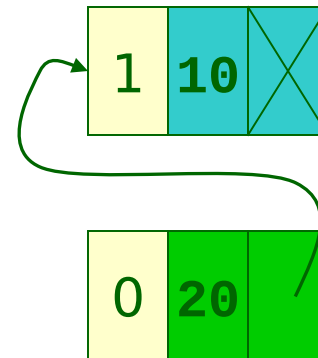
```
a = cons(10, empty)
b = cons(20, a)
a = b
b = ...
a = ...
```



# Reference Counting: Example

---

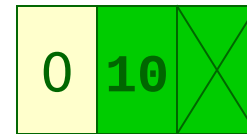
```
a = cons(10, empty)
b = cons(20, a)
a = b
b = ...
a = ...
```



# Reference Counting: Example

---

```
a = cons(10, empty)
b = cons(20, a)
a = b
b = ...
a = ...
```



# Reference Counting: Example

---

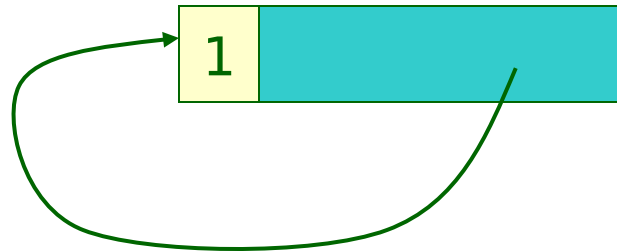
```
a = cons(10, empty)
b = cons(20, a)
a = b
b = ...
a = ...
```



# Reference Counting: Problem

---

? What's the problem? ?



**No other pointer to this data, so can't refer to it**  
**Count not zero, so never deallocated**  
**Following does NOT hold: Count is positive  $\square$  reachable**

**Can occur with any cycle**

# Reference Counting: Summary

---

## Disadvantages:

- ♦ **Managing & testing counts is generally expensive**
  - Can optimize
- ♦ **Doesn't work with cycles!**
  - Approach can be modified to work, with difficulty

## Advantage:

- ♦ **Simple**
  - Easily adapted, e.g., for parallel or distributed GC

## Useful when cycles can't happen

- ♦ **E.g., UNIX hard links**

# GC Without Reference Counts

---

**If don't have counts, how to deallocate?**

**Determine reachability by traversing pointer graph directly**

- ♦ **Stop user's computation periodically to compute reachability**
- ♦ **Deallocate anything unreachable**

# Mark & Sweep

---

## Overall idea

- ♦ **Maintain a free list of unallocated blocks**
- ♦ **To allocate, grab a sufficiently large block from free list**
- ♦ **When no such block exists, GC**
  - **Should find blocks & put them on free list**

# Mark & Sweep: GC

---

## Follow all pointers, marking all reachable data

- ♦ Use depth-first search
- ♦ Data must be tagged with info about its type, so GC knows its size and can identify pointers
- ♦ Each piece of data must have a mark bit
  - Can alternate meaning of mark bit on each GC to avoid erasing mark bits


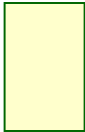
## Sweep over all heap, putting all unmarked data into a free list

- ♦ Again, same implementation issues for the free list

# Mark & Sweep: GC Example

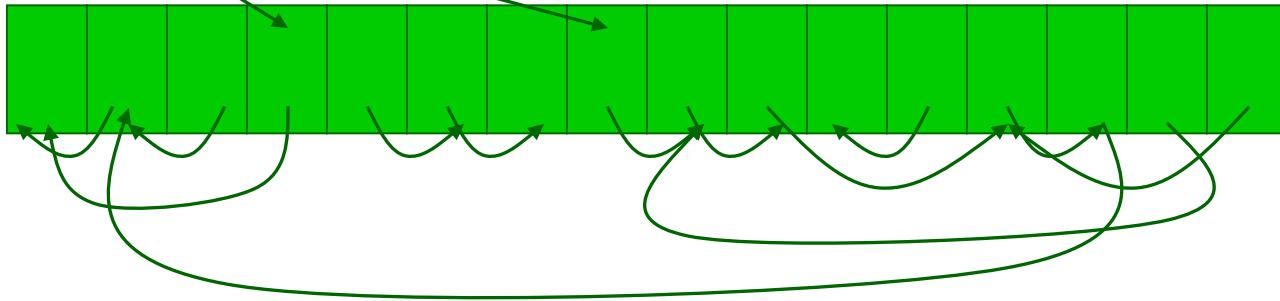
---

Assume fixed-sized, single-pointer data blocks, for simplicity.

Unmarked=  Marked= 

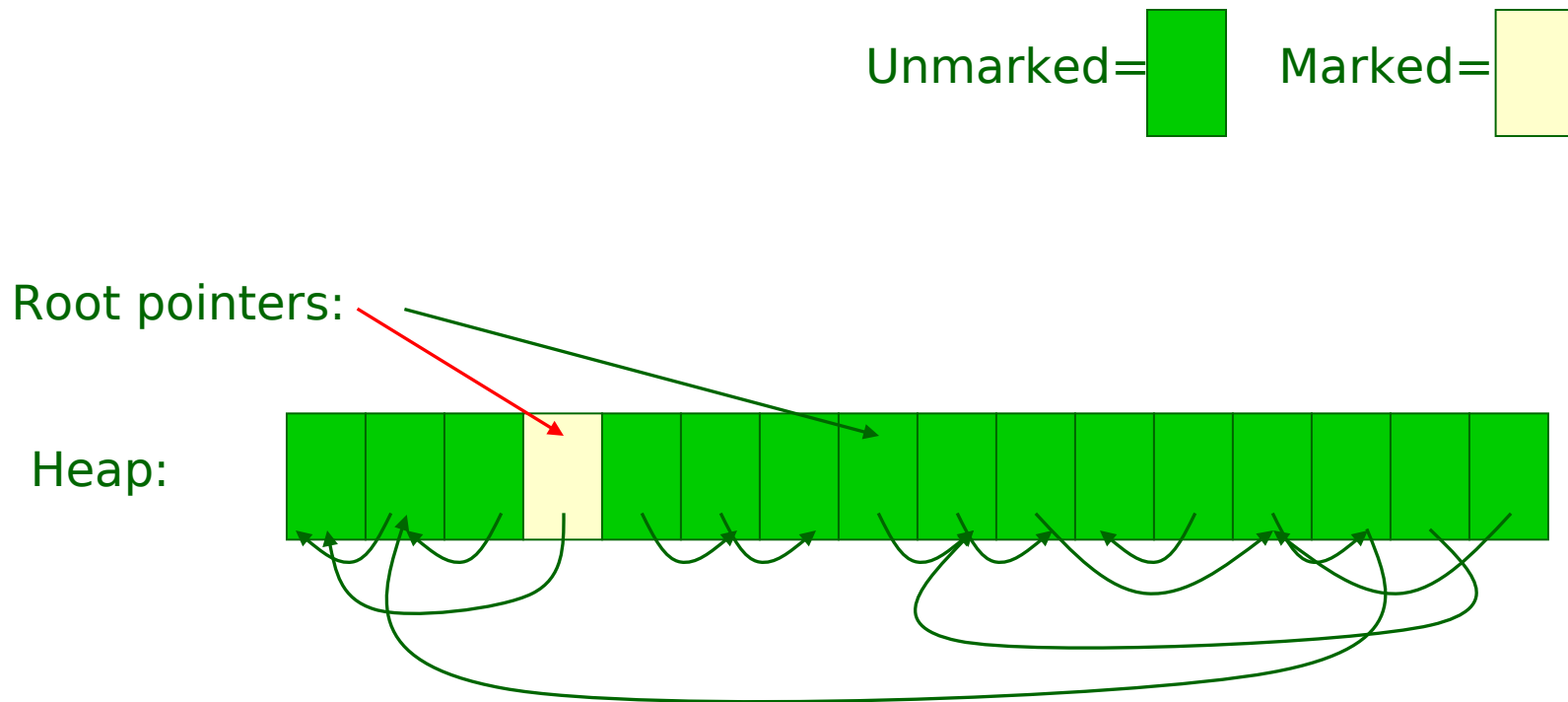
Root pointers:

Heap:



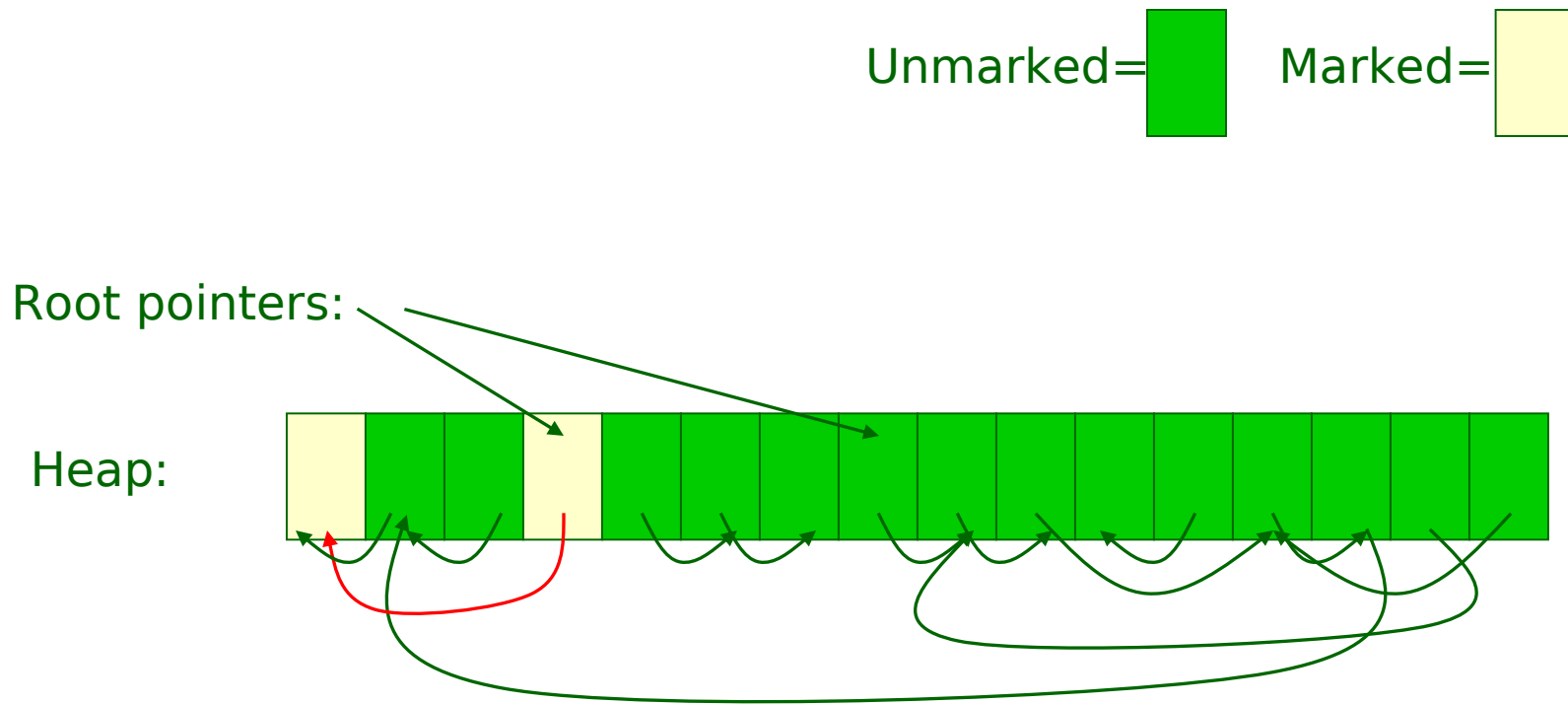
# Mark & Sweep: GC Example

---



# Mark & Sweep: GC Example

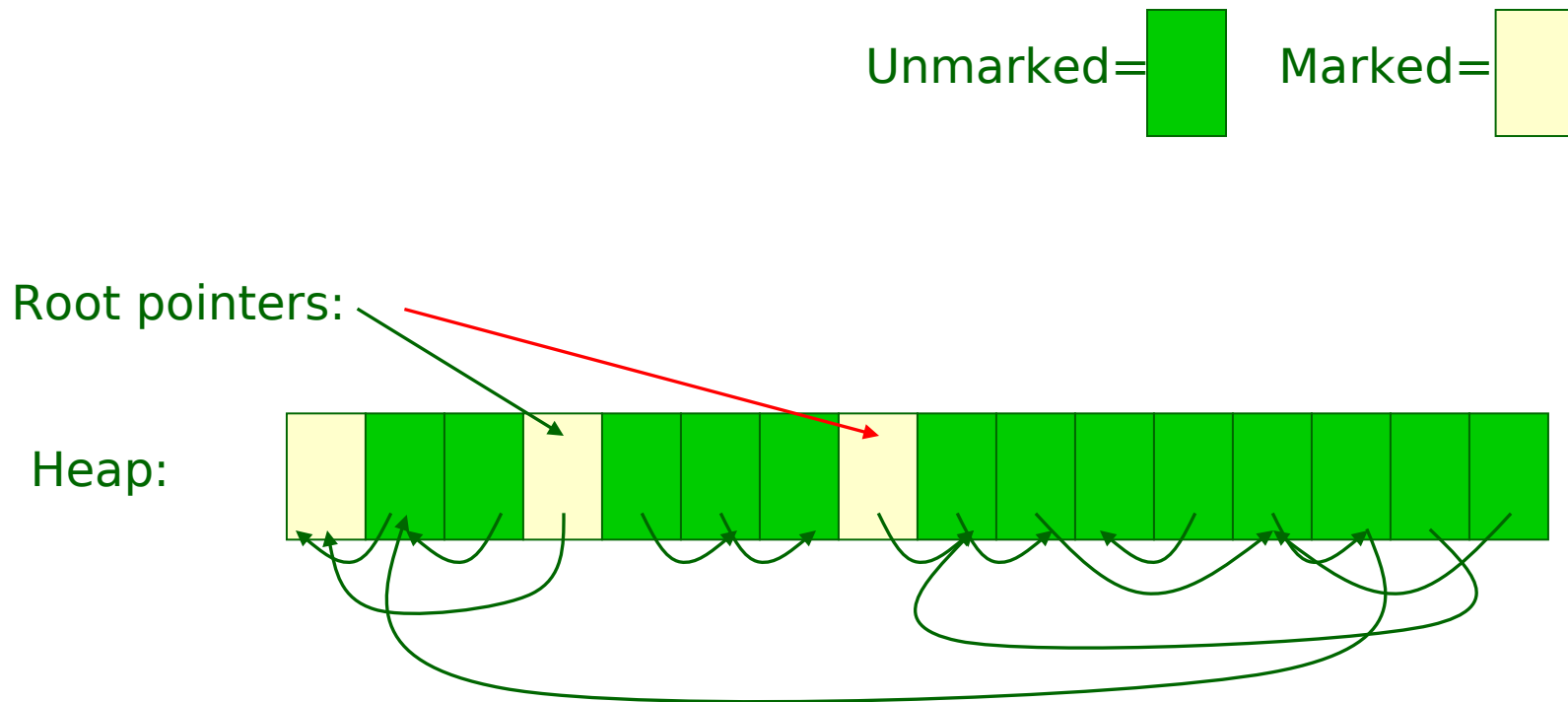
---






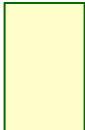
# Mark & Sweep: GC Example

---



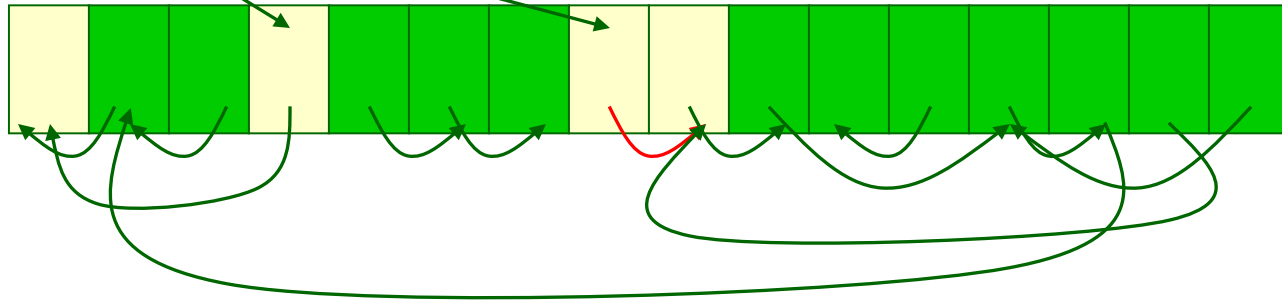
# Mark & Sweep: GC Example

---

Unmarked=  Marked= 

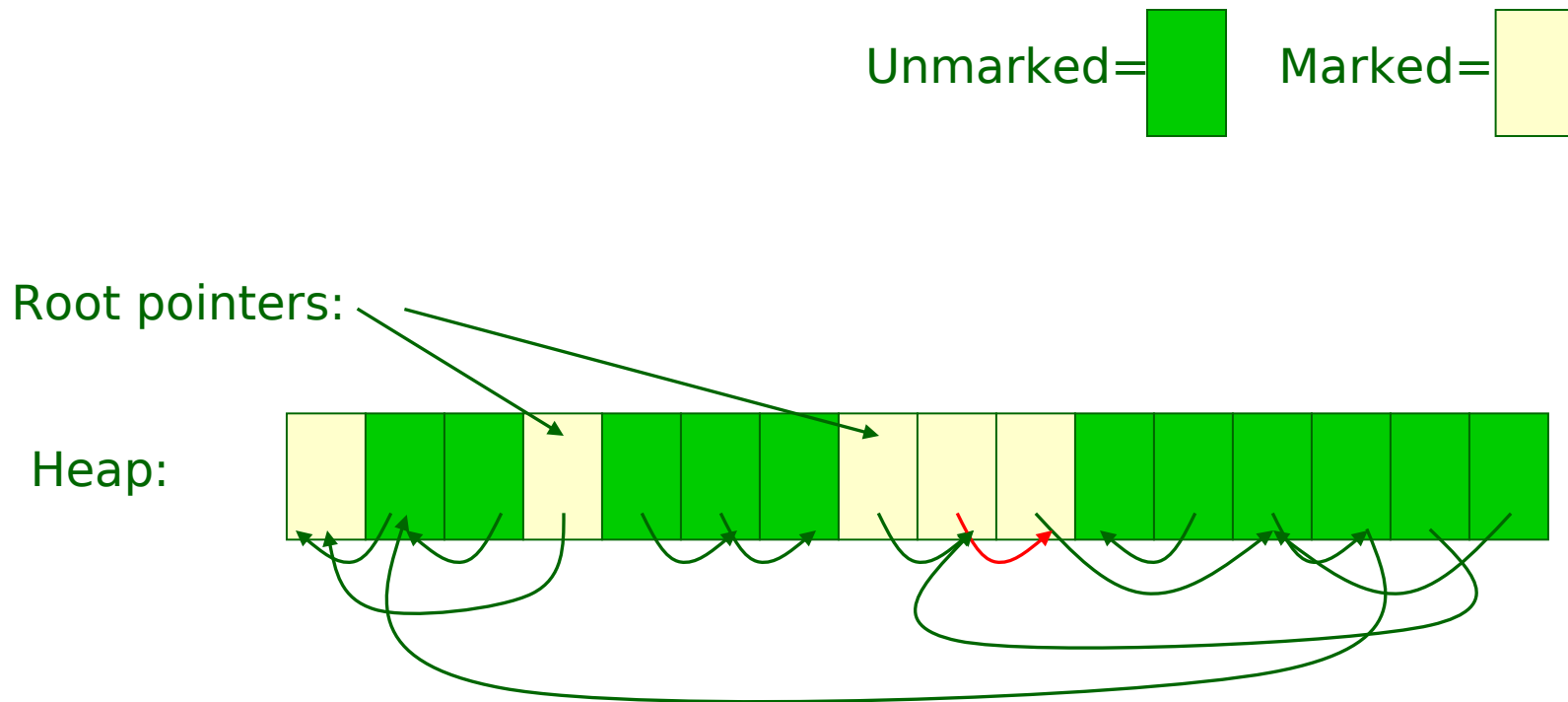
Root pointers:

Heap:



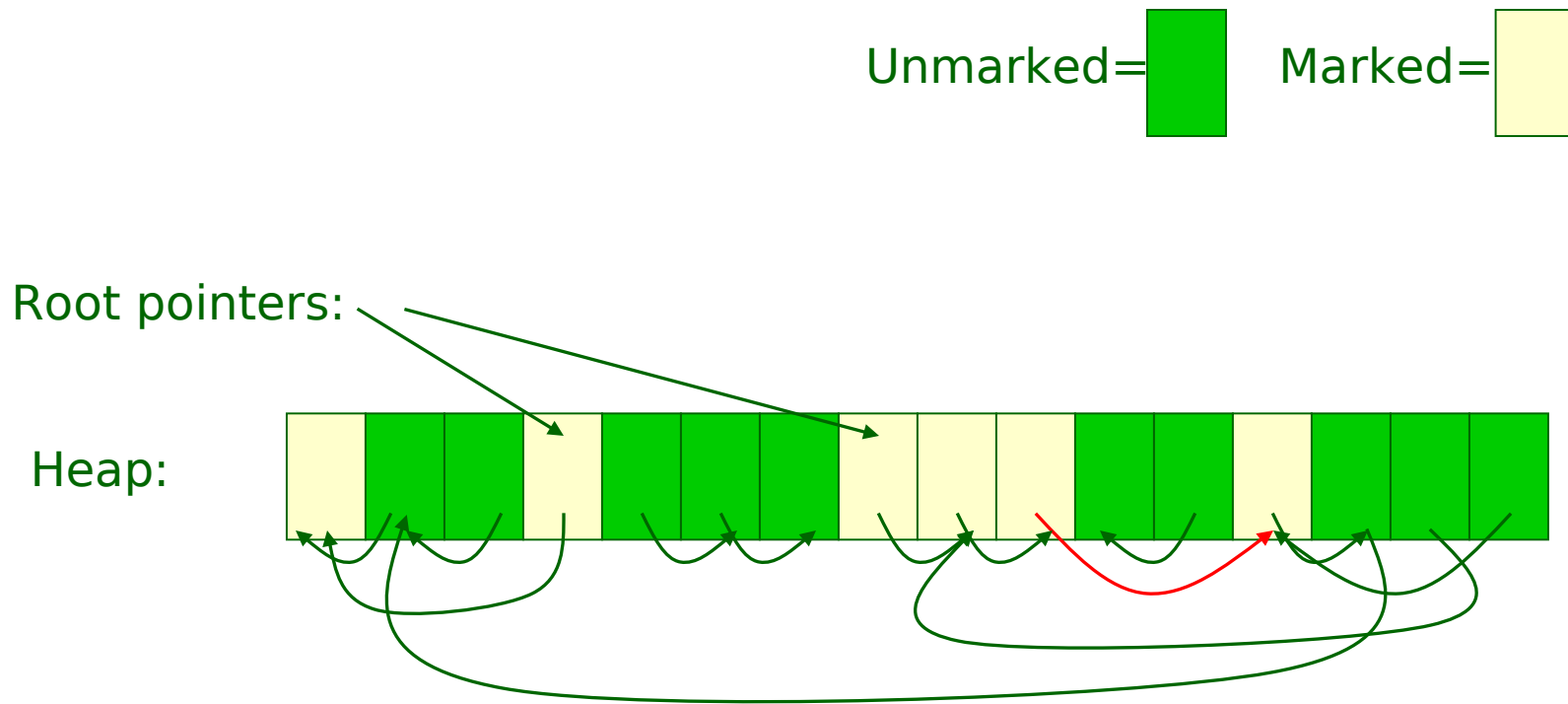
# Mark & Sweep: GC Example

---



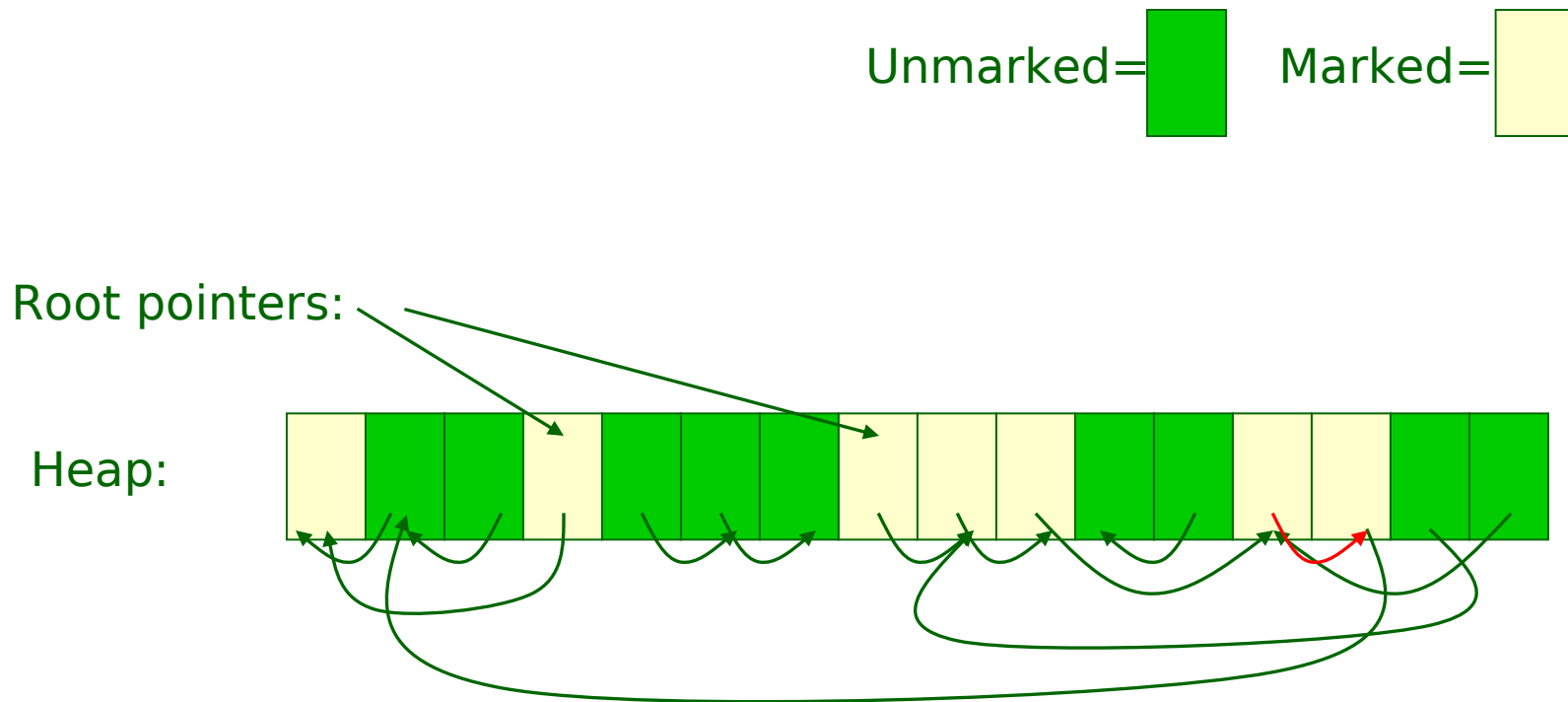
# Mark & Sweep: GC Example

---





# Mark & Sweep: GC Example

---

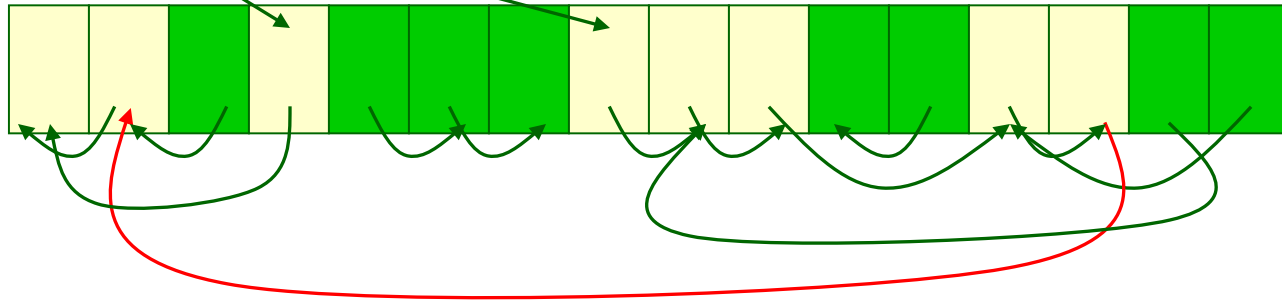


# Mark & Sweep: GC Example

Unmarked=       Marked= 

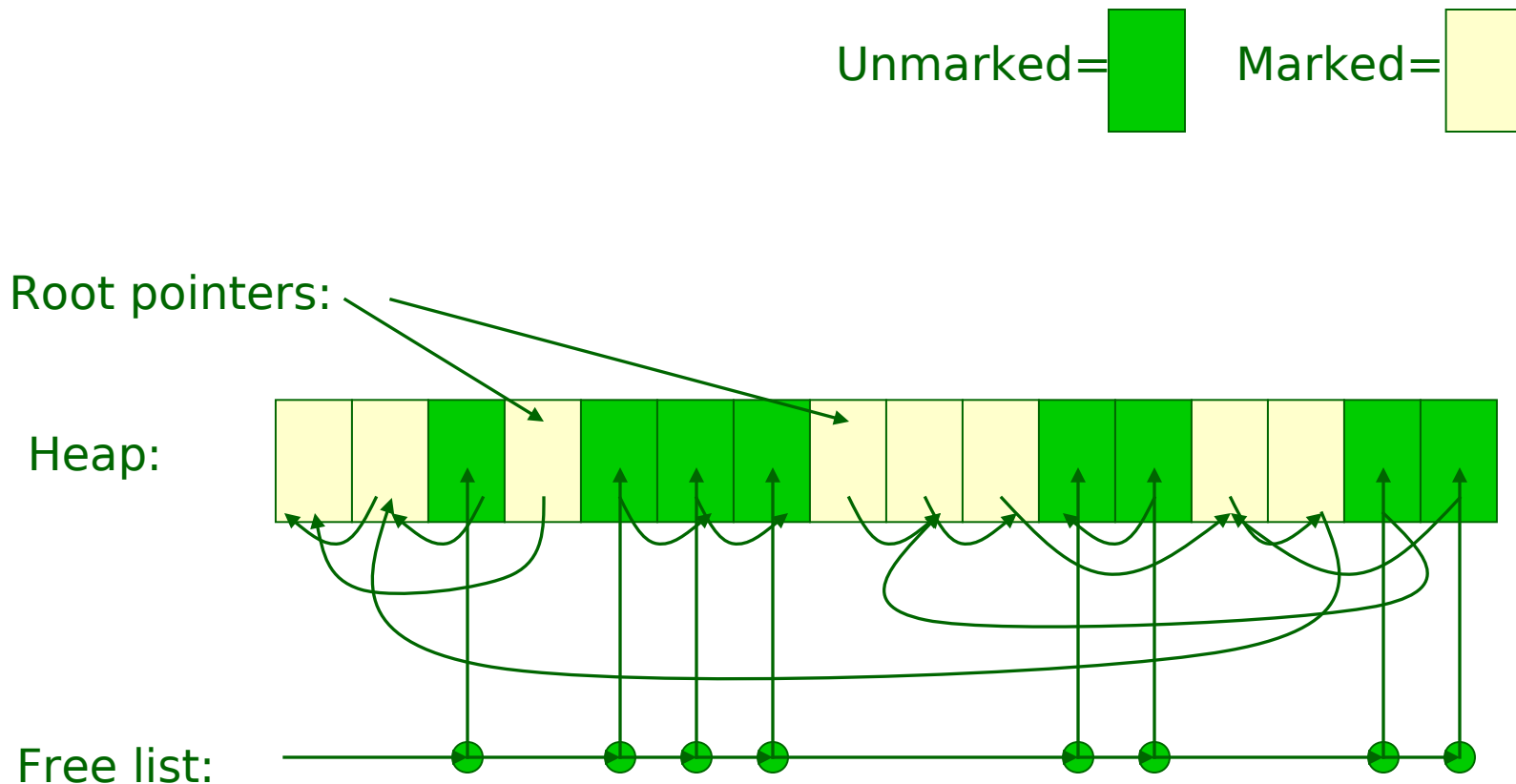
## Root pointers:

## Heap:



# Mark & Sweep: GC Example

---



# Mark & Sweep: Summary

---

## Advantages:

- ♦ No space overhead of reference counts
- ♦ No time overhead of reference counts
- ♦ Handles cycles

## Disadvantage:

- ♦ Noticeable pauses for GC



# Stop & Copy

---

## Overall idea:

- ♦ Maintain *From* and *To* spaces in heap
- ♦ To allocate, get sequentially next block in *From* space
  - No free list!
- ♦ When *From* space full, GC into *To* space
  - Swap *From* & *To* names

# Stop & Copy: GC

---

**Follow all From-space pointers, copying all reachable data into To-space**

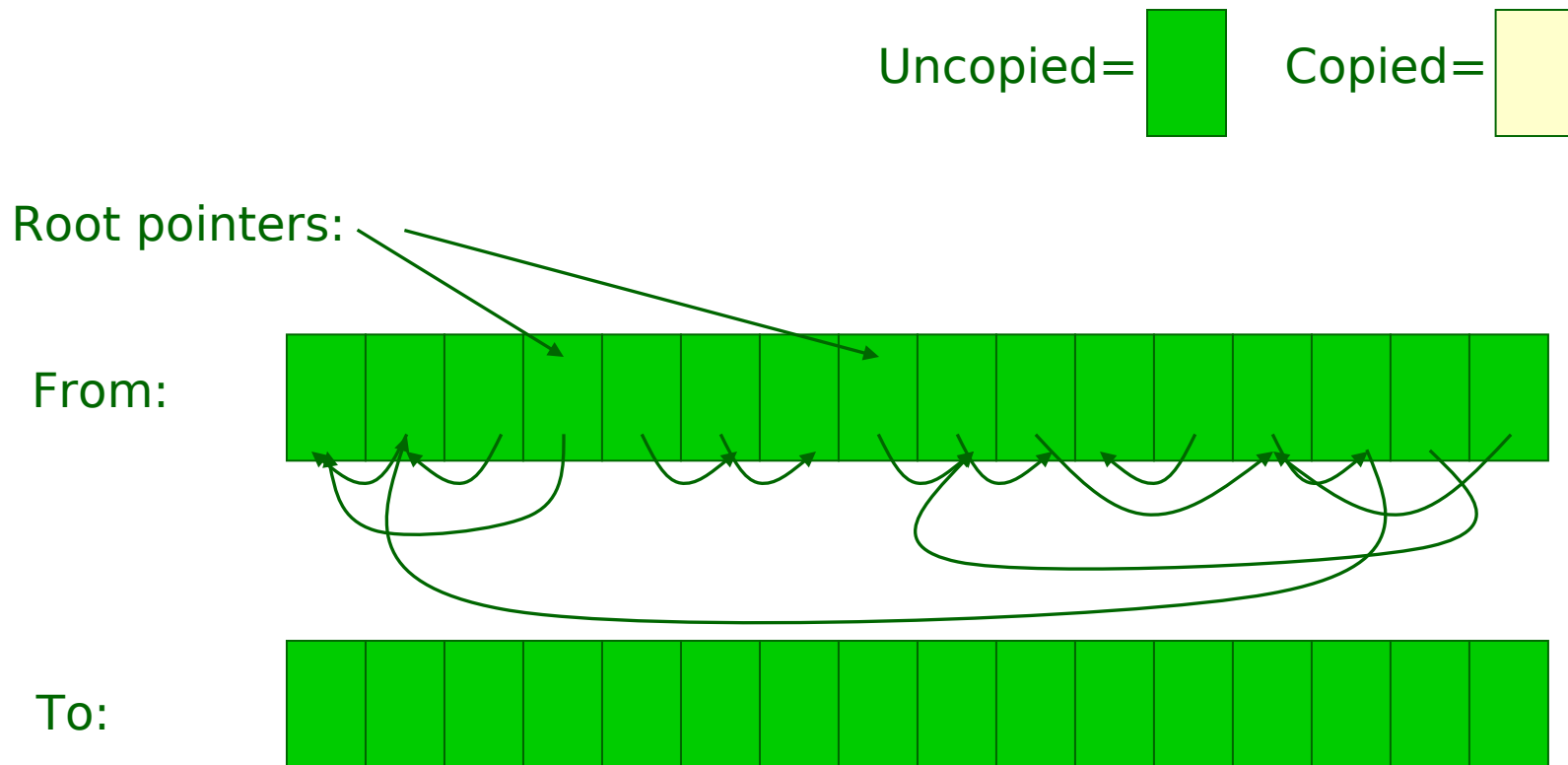
- ♦ **Use depth-first search**
- ♦ **Data must be tagged with info about its type, so GC knows its size and can identify pointers**

**Swap From-space and To-space names**

# Stop & Copy: GC Example

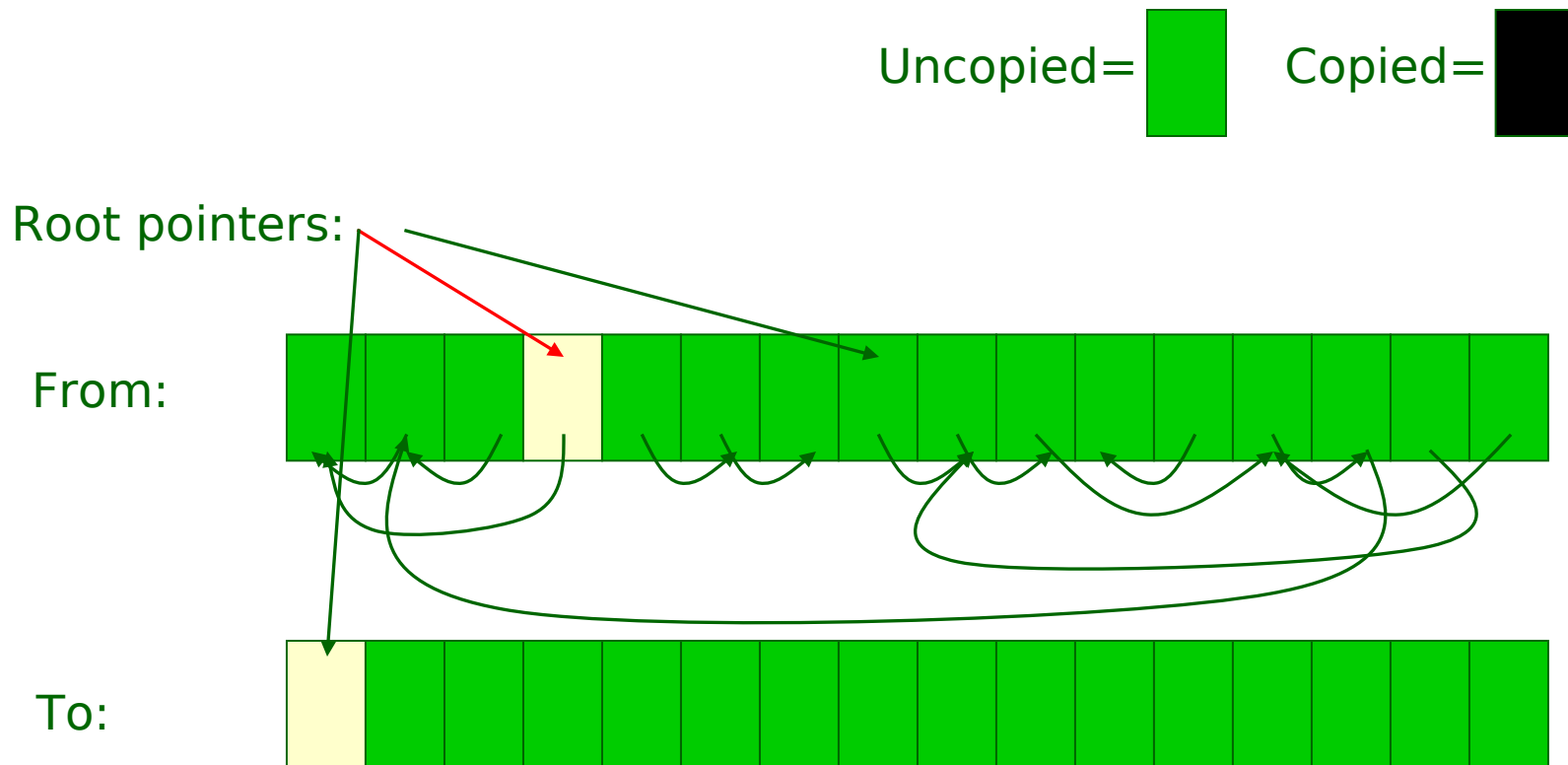
---

Assume fixed-sized, single-pointer data blocks, for simplicity.



# Stop & Copy: GC Example

---

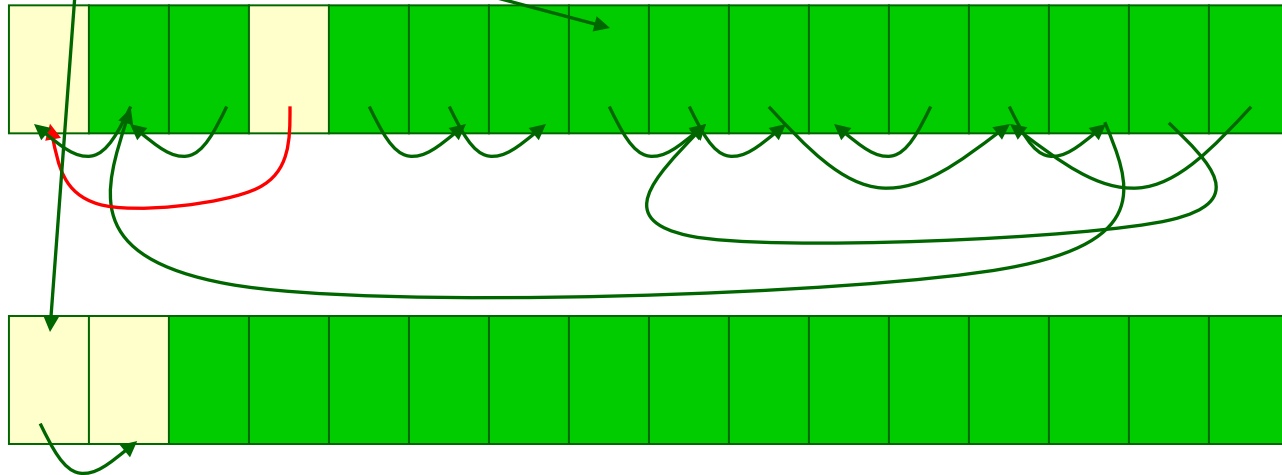


Uncopied=

## Root pointers:

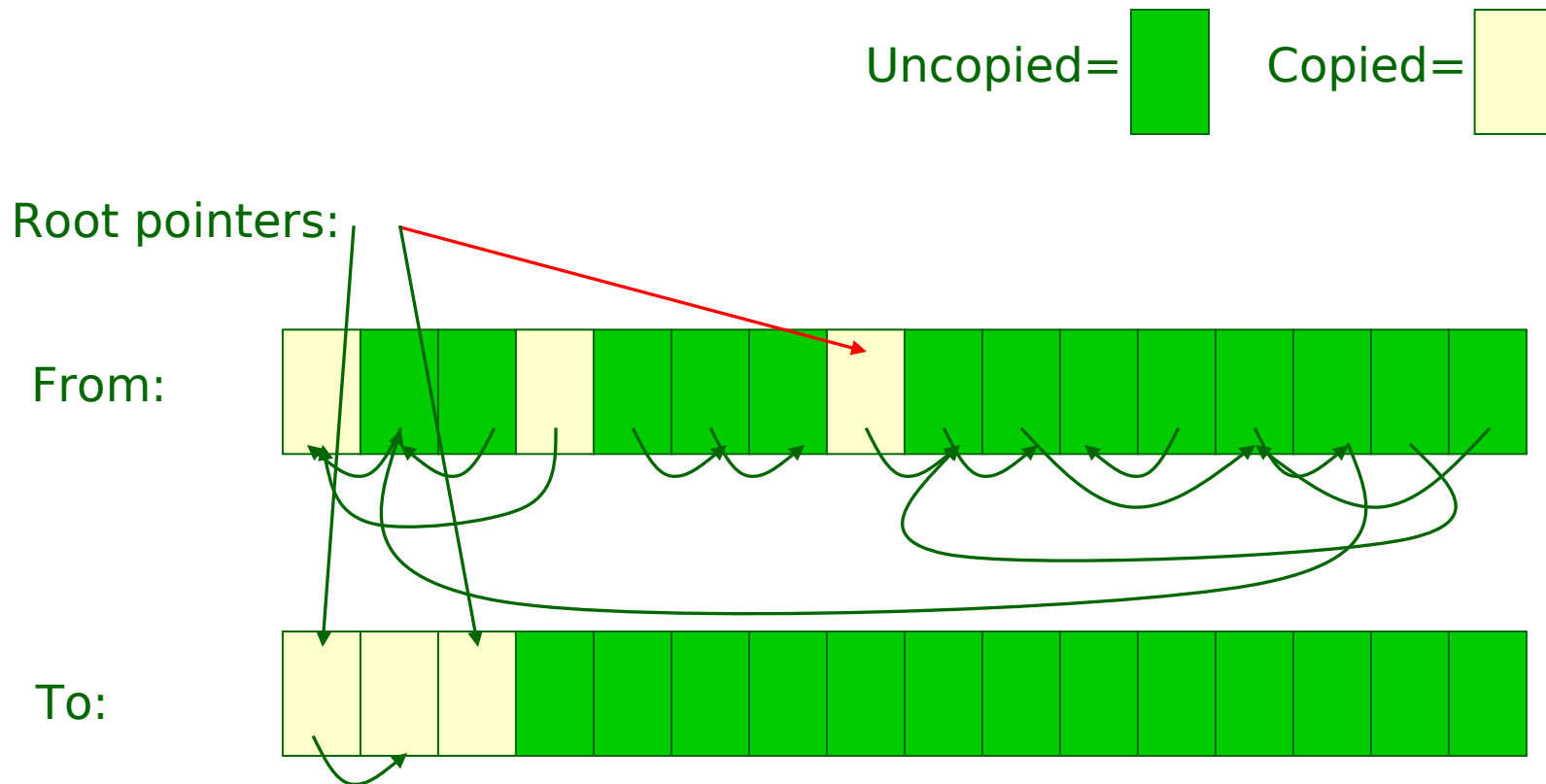
From:

To:



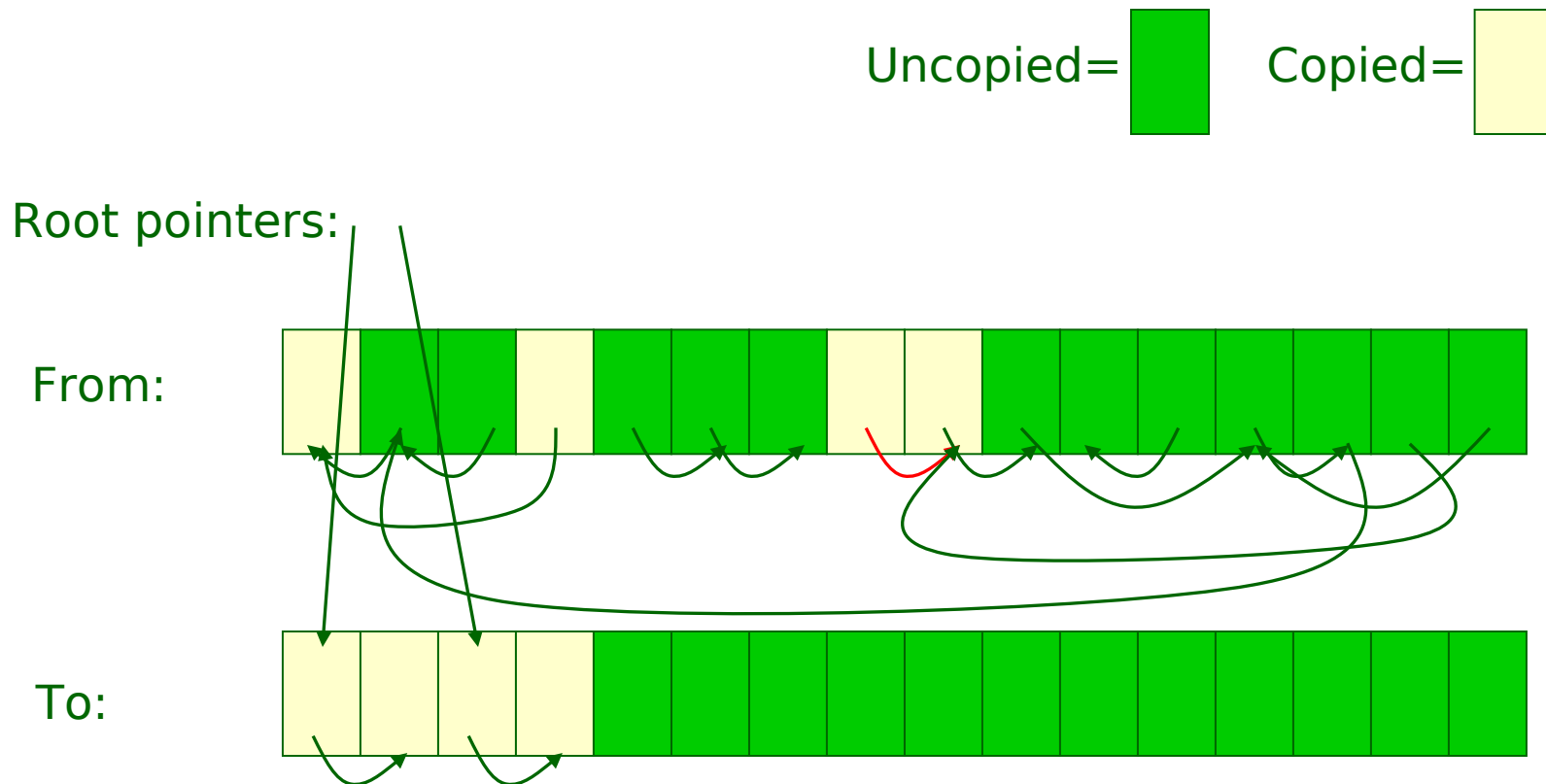
# Stop & Copy: GC Example

---



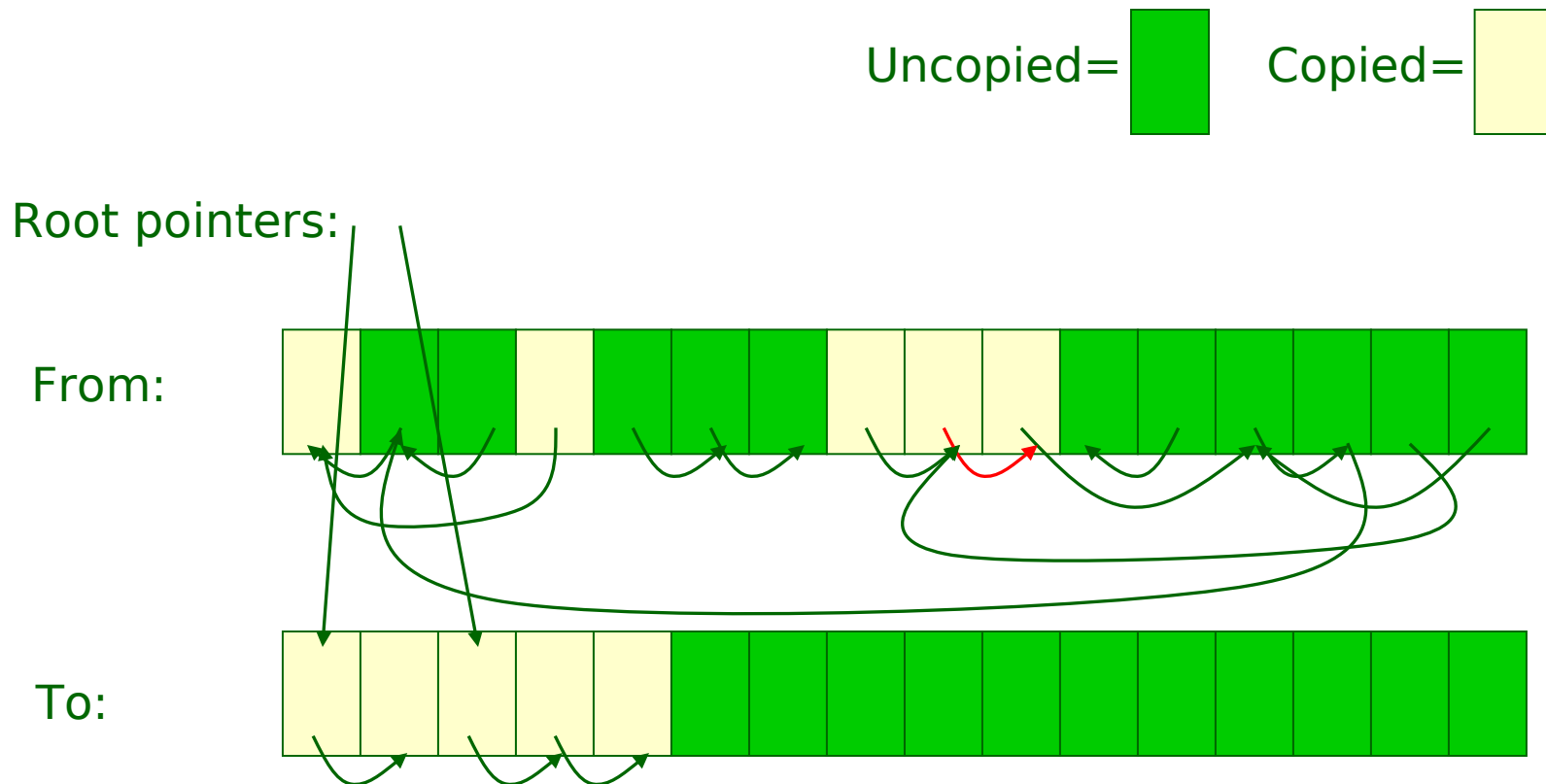
# Stop & Copy: GC Example

---



# Stop & Copy: GC Example

---





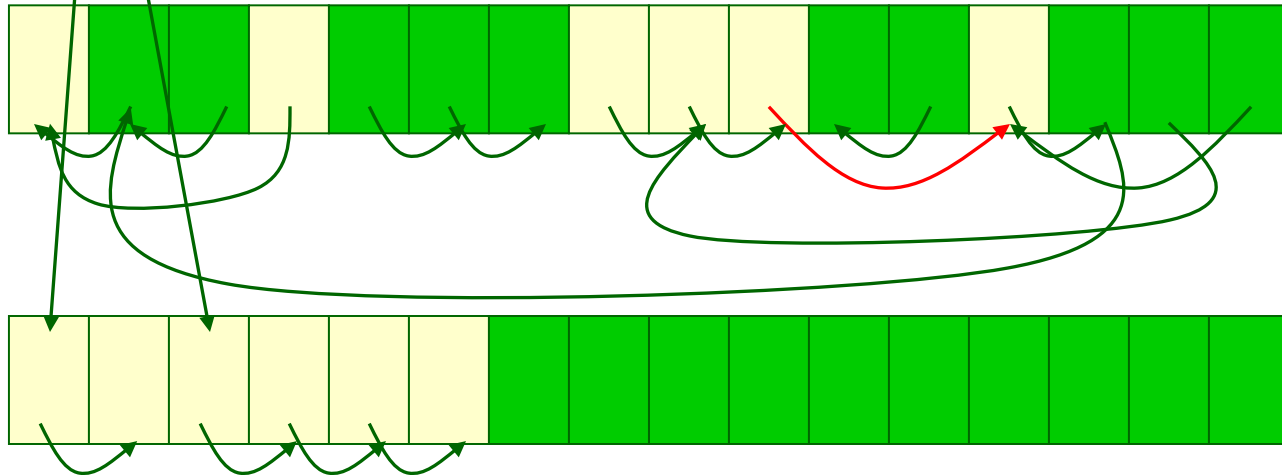
Uncopied=

Copied=

## Root pointers:

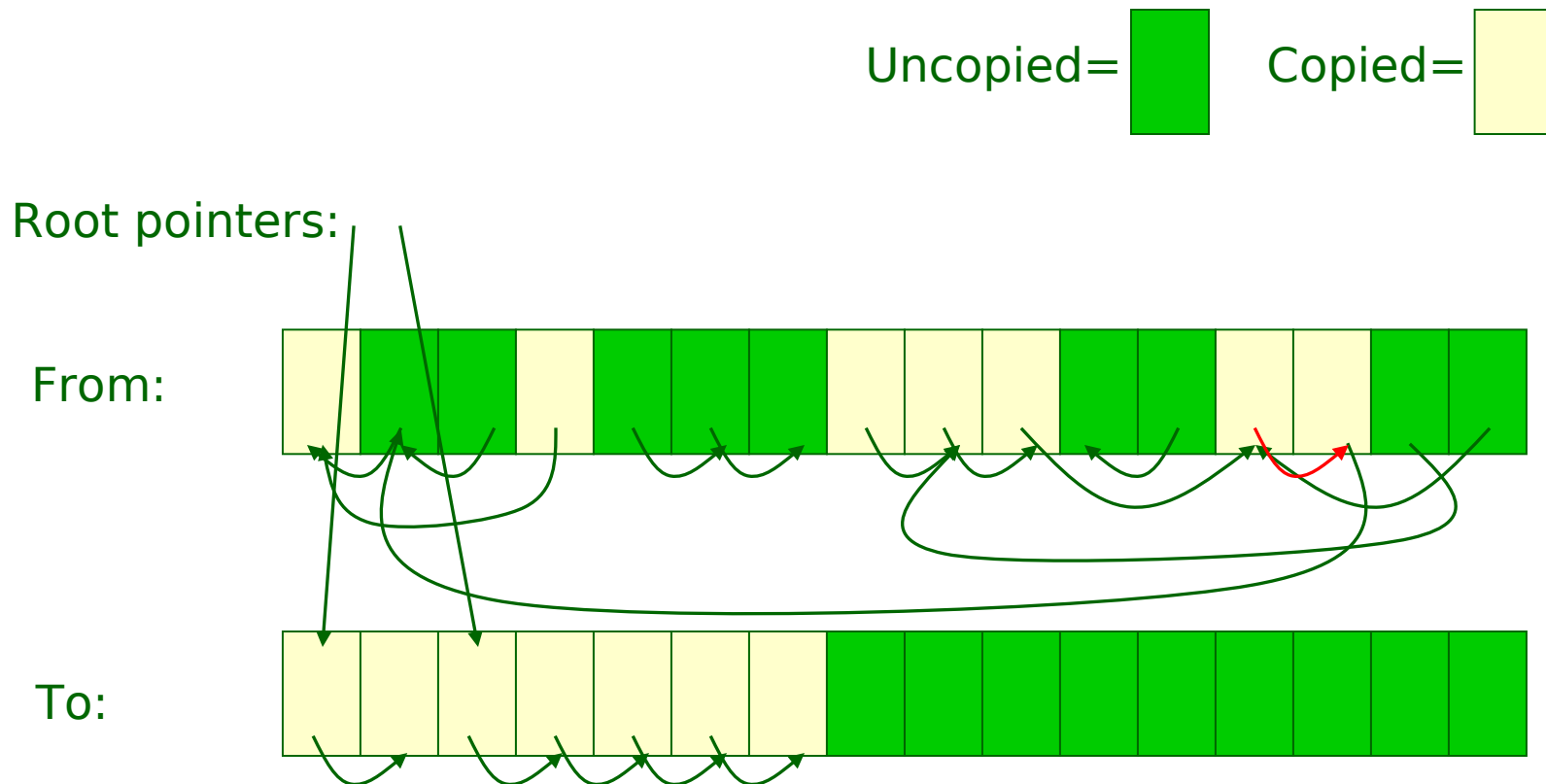
From:

To:



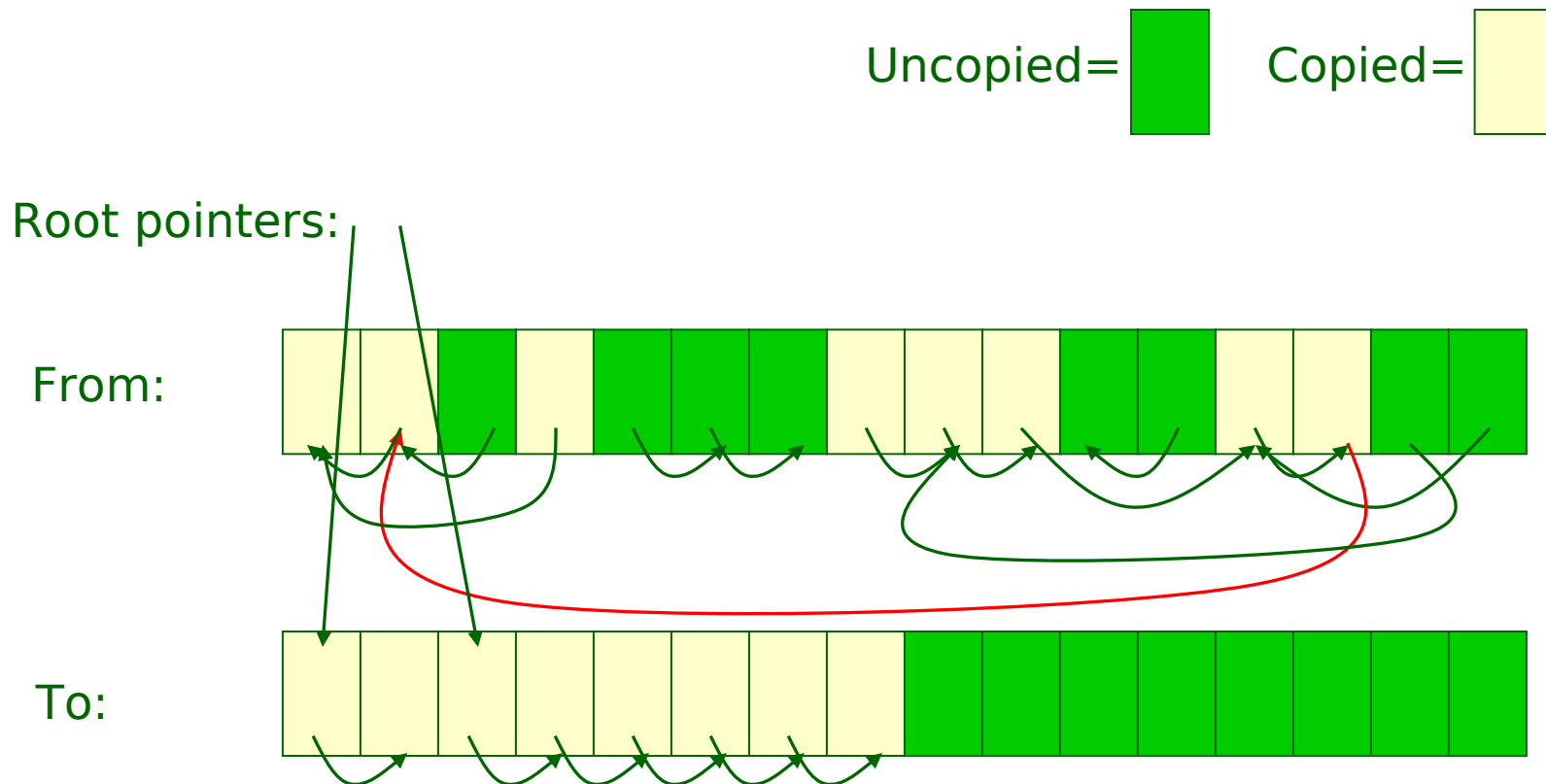
# Stop & Copy: GC Example

---



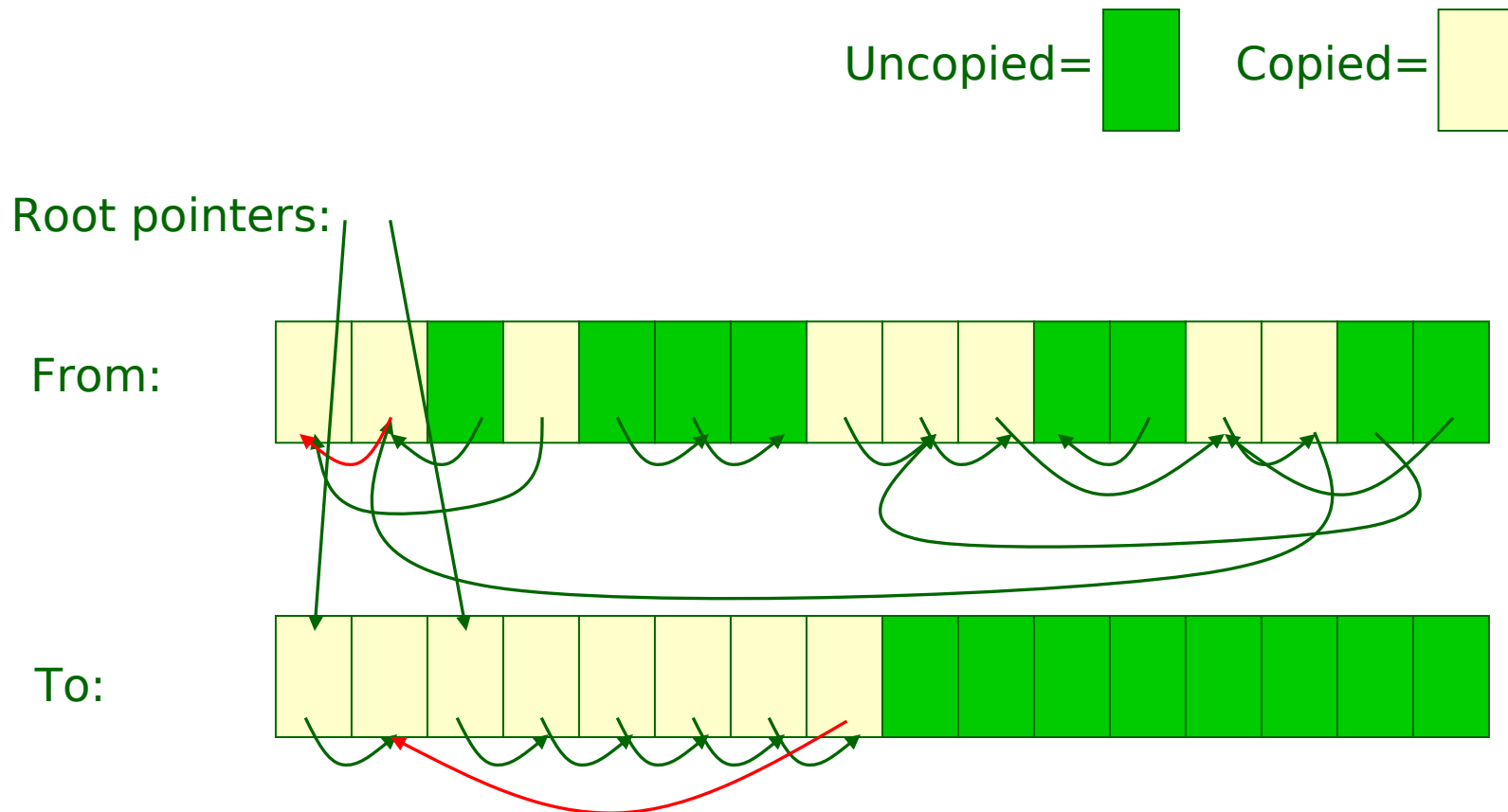
# Stop & Copy: GC Example

---



# Stop & Copy: GC Example

---



# Stop & Copy: GC Example

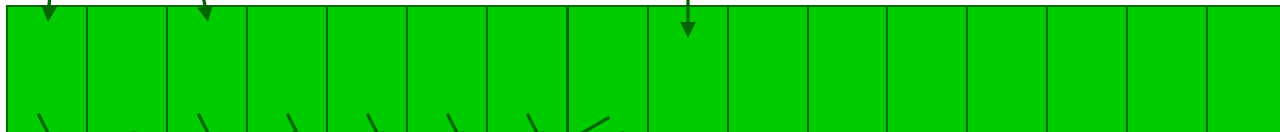
---

Root pointers:

To:



From:



Next block to allocate

# Stop & Copy

---

## Advantages:

- ♦ Only one pass over data
- ♦ Only touches reachable data
- ♦ Little space overhead per data item
- ♦ Very simple allocation
- ♦ “Compacts” data
- ♦ Handles cycles

## Disadvantages:

- ♦ Noticeable pauses for GC
- ♦ Double the basic heap size

# Compaction

---

**Moving allocated data into contiguous memory**

**Eliminates fragmentation**

**Tends to increase spatial locality**

**Must be able to reassociate data & locations**

- ♦ **Not possible if pointers in source language**

# GC Variations

---

***Many variations on these three main themes***



# Conservative GC

---

## Goal

- ♦ **Allow GC in C-like languages**

**Usually a variation on Mark & Sweep**

**Must conservatively assume that integers and other data can be cast to pointers**

- ♦ **Compile-time analysis to see when this is definitely not the case**
- ♦ **Code style heavily influences effectiveness**

# GC vs. malloc/free Summary

---

**Safety is not programmer-dependent**

**Compaction generally improves locality**

**Higher or lower time overhead**

- ♦ **Generally less predictable time overhead**

**Generally higher space overhead**

# Next Time

---

## Virtual Memory