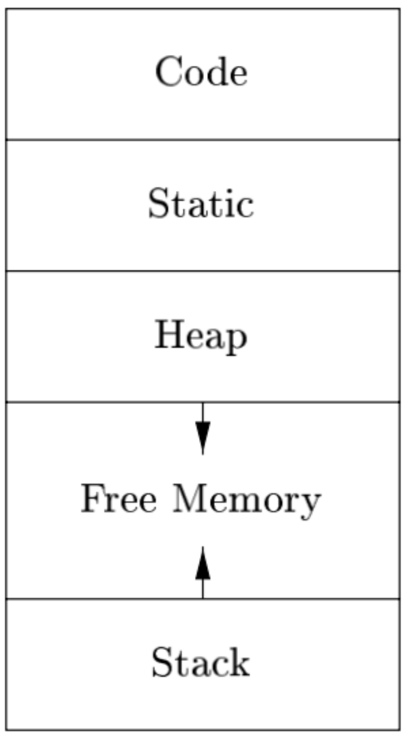
Computer Science 160
Translation of Programming Languages

Instructor: Christopher Kruegel

Memory Management

Overall: Runtime Memory

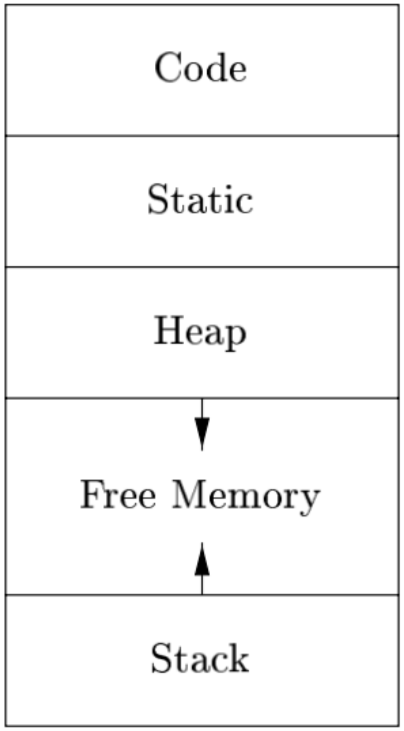
Typical subdivision of run-time memory into code and data areas



- Compiler writer: The executing target program runs in its own continuous logical address space in which each program value has a location
- The operating system then maps the logical addresses into physical addresses, which are usually spread throughout memory

Static Storage Allocation

We say that a storage allocation decision is **static** if it is made by the compiler looking only at the text of the program



Static allocation

- Code: generated target code is fixed at compile time, so the compiler can place the executable target code in a statically determined **Code**, area usually in the low end of memory

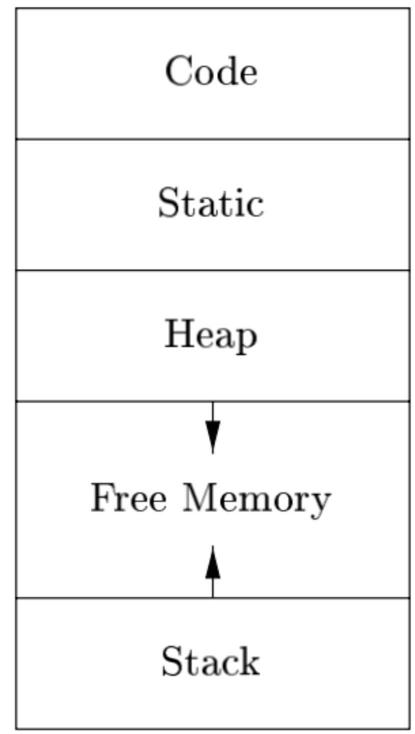
- Static data, such as globals. These data objects can be placed in another statically determined area called **Static.**
  - Benefits: the addresses of these objects can be compiled into the target executable.

Dynamic Storage Allocation

For dynamic space, whose size can change as the program executes

- Stack
  - centering around procedures (activation records)

- Heap
  - dynamic but not local: data that may outlive the call to the procedure that created it is usually allocated on a “heap” of reusable storage



Examples: Stack and Heap Memory (Both at Runtime)


```
int main()
{
   // All these variables get memory
   // allocated on stack
   int a;
   int b[10];
   int n = 20;
   int c[n];
}
```

```
int main()
{
   // This memory for 10 integers
   // is allocated on heap.
   int *ptr  = new int[10];
}
```

Recall: Stack Used By Procedures

- When a procedure is called, a block is reserved on the top of the **stack** for local variables and some bookkeeping data
- When that procedure returns, the block becomes unused and can be used the next time a function is called
- The stack is always reserved in a **LIFO (last in, first out)** order; the most recently reserved block is always the next block to be freed
- This makes it simple to keep track of the stack; **freeing a block from the stack is nothing more than adjusting one pointer**

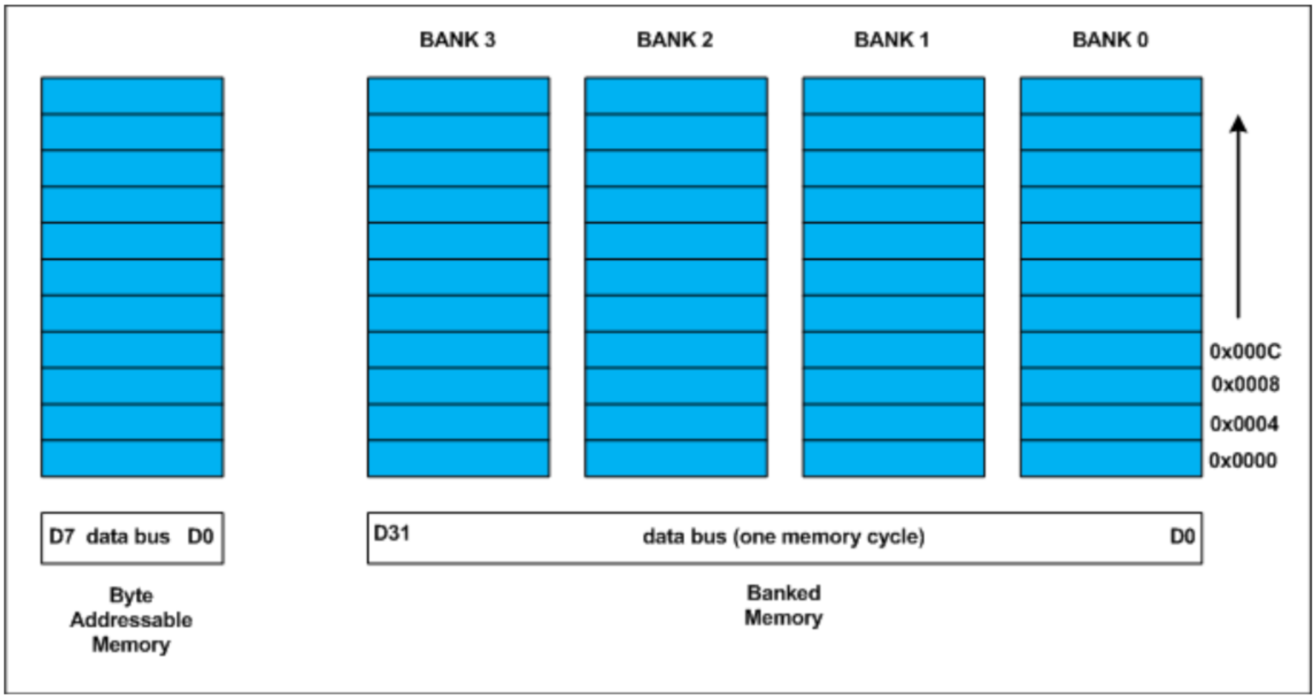
Address of Local Variables

How does the compiler represent memory location for a specific instance of variable *x* for a procedure?

- Name is translated into a ***static coordinate**: < level, offset >*
  - *"level"* is lexical scoping level
  - *"offset"* is *unique* within that scope
  - *"offset"* is assigned at compile time and it is used to generate code that executes at run-time
- Static distance coordinate is used to generate addresses
  - For each lexical scope level, we generate a *base address*
  - *offset* gives the location of a variable relative to that base address

Memory Alignment and Padding

- The storage layout for data is influenced by the addressing constraints of the target machine



- On many machines, instructions to add integers may expect integers to be aligned that is placed at an address divisible by 4

Question Time


```c
// structure A
typedef struct structa_tag
{
    char        c;
    short int   s;
} structa_t;
```

```c
// structure B
typedef struct structb_tag
{
    short int   s;
    char        c;
    int         i;
} structb_t;
```

```c
// structure C
typedef struct structc_tag
{
    char        c;
    double      d;
    int         s;
} structc_t;
```

```c
// structure D
typedef struct structd_tag
{
    double      d;
    int         s;
    char        c;
} structd_t;
```

```c
int main()
{
    printf("sizeof(structa_t) = %lu\n", sizeof(structa_t));
    printf("sizeof(structb_t) = %lu\n", sizeof(structb_t));
    printf("sizeof(structc_t) = %lu\n", sizeof(structc_t));
    printf("sizeof(structd_t) = %lu\n", sizeof(structd_t));

    return 0;
}
```

What does this print out?

```
sizeof(structa_t) = 4
sizeof(structb_t) = 8
sizeof(structc_t) = 24
sizeof(structd_t) = 16
```

Stack versus Heap

Stack

Heap

- Stack memory is associated with the stack data structure, which follows a LIFO pattern for memory allocation and deallocation.
- But the name **heap** has nothing to do with the **heap** data structure. It is called heap because it is a pile of memory space available to programmers to allocate a block at any time and free it at any time.
- This makes it much more complex to keep track of which parts of the heap are allocated or free at any given time; there are many custom heap allocators available to tune heap performance for different usage patterns.

Heap Memory

- Heap memory allocation isn't as easy as stack memory allocation because the data stored in this space is accessible or visible out of a procedure.

- Different from stack memory management, no efficient, automatic de-allocation feature is provided

Key Memory Manager Functions

Memory Manager

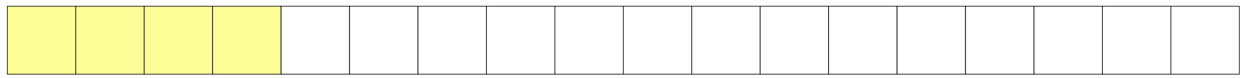
- The subsystem that allocates and deallocates space within the heap.

  - **Allocation:** A chunk of contiguous heap memory of the requested size when a request is issued. If not enough space, then increasing the heap storage space by getting consecutive bytes of virtual memory.
  - We also assume that (1) Allocation requests are for chunks of the different sizes. (2) There is no good way to predict the lifetimes of all allocated objects.

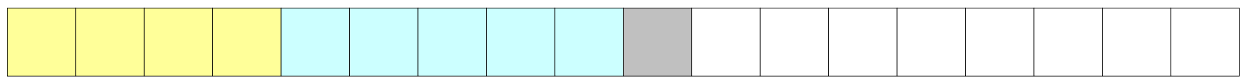
  - **Deallocation: …**

Allocation Examples: Internal Fragmentation

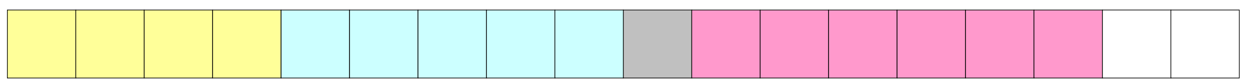

```
p1 = malloc(4*sizeof(int))
```



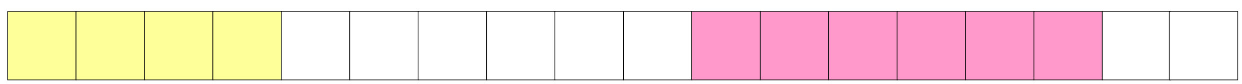
```
p2 = malloc(5*sizeof(int))
```



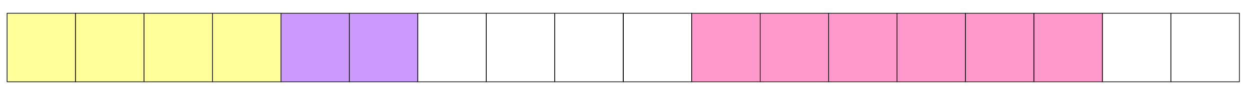
```
p3 = malloc(6*sizeof(int))
```



```
free(p2)
```

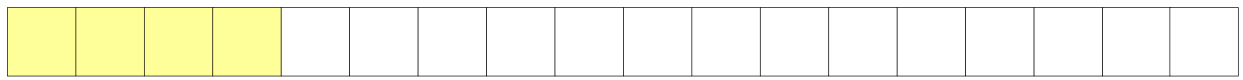


```
p4 = malloc(2*sizeof(int))
```

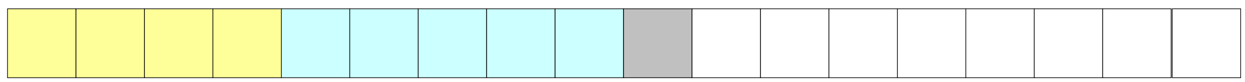


Allocation Examples: External Fragmentation

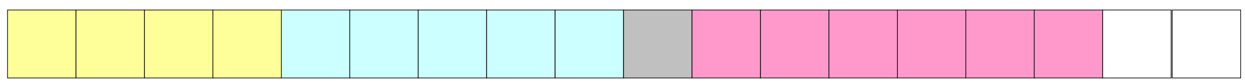

```
p1 = malloc(4*sizeof(int))
```



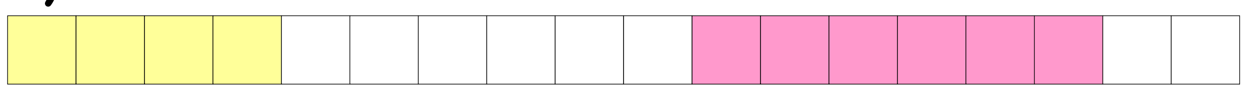
```
p2 = malloc(5*sizeof(int))
```



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



```
free(p2)
```



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

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

Key Memory Manager Functions

Memory Manager

- The subsystem that allocates and deallocates space within the heap.

  - **Allocation:** A chunk of contiguous heap memory of the requested size when a request is issued. If not enough space, then increasing the heap storage space by getting consecutive bytes of virtual memory.
  - We also assume that (1) Allocation requests are for chunks of the different sizes. (2) There is no good way to predict the lifetimes of all allocated objects.

  - **Deallocation:** It will return deallocated space to the pool of free space so it can reuse the space to satisfy other allocation requests. **NOTE:** it typically does not return memory to the operating system even if the program's heap usage drops.

Manual Memory Deallocation

- Programmer has full control over memory

  . . . with the responsibility to manage it well

- Premature free's lead to **dangling references** (referencing deleted data)
- Overly conservative free's lead to **memory leaks** (failing ever to delete data that cannot be referenced)
- With manual free's, it is difficult to ensure that a program is correct and secure

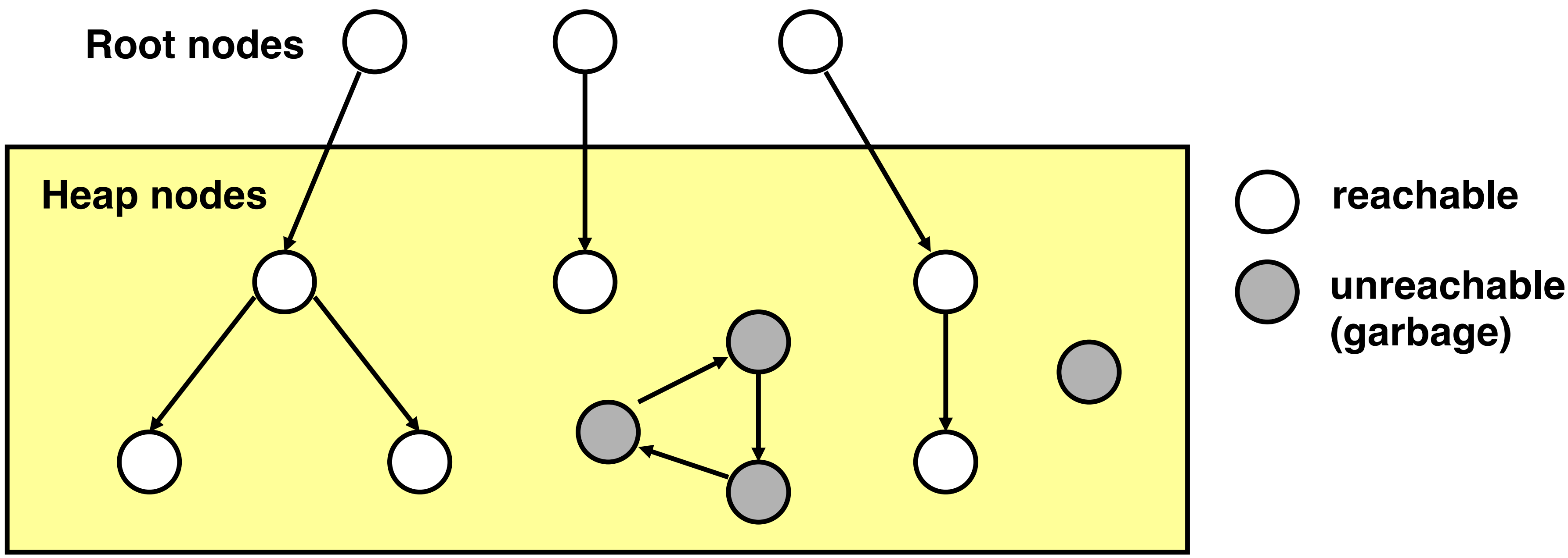
- Even with manual memory management, the system maintains bookkeeping data and does nontrival memory-related processing (e.g., search for appropriate chunk to allocate, avoid fragmentation, etc.)

Garbage Collector

- Data that cannot be **referenced** is generally known as garbage
- Many high-level programming languages remove the burden of manual memory management from the programmer by offering automatic garbage collection, which deallocates **unreachable** data
- Garbage collection dates back to the initial implementation of Lisp in 1958.
- Other significant languages that offer garbage collection include Python, Prolog, Smalltalk, …

Garbage Data: 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)



Performance Metrics

- Many different approaches, but there is not one clearly best garbage collection algorithm.

- Key metrics
  - Overall Execution Time. It is at runtime, taking part of our program execution time.

  - Pause Time. It could cause programs to pause suddenly. A maximum pause time shall be guaranteed, especially for those real-time applications that require certain computations to be completed within a time limit.

  - Program Locality. It also controls the placement of data and thus influences the data locality. A “great” garbage collector could make the original problem running slower

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)

Reference Couting

- Reference counting is a conservative technique for detecting garbage
- Each object has a **reference count**: the #references made to it (in-degree of the node in object graph). When the reference count of an object falls to 0, then the object is garbage (and, hence, collected)

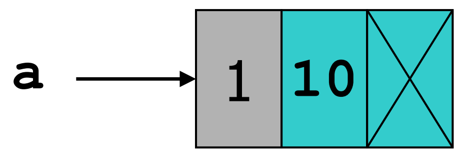
- When an object is allocated, we initialize its reference count to **0.**
- Increment reference counts
  - Assignment
  - Parameter Passing (more like explicit assignment)
- Decrement reference counts
  - New Assignment (p = q → p = r)
  - Procedure exits. All objected referred to by its local variables shall have their counts decremented. If local variables hold references to the same object, that object's count must be decremented once for each such reference.

Reference Counting: Example


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

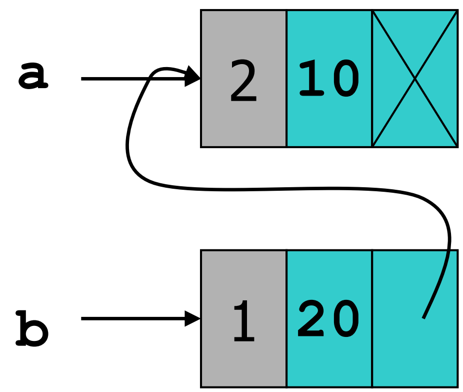
Reference Counting: Example


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



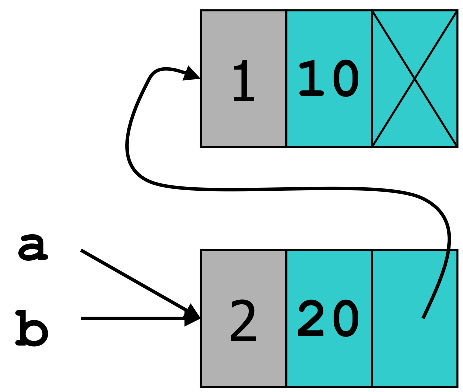
Reference Counting: Example


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



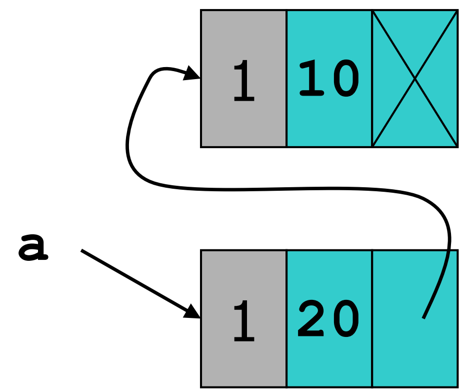
Reference Counting: Example


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



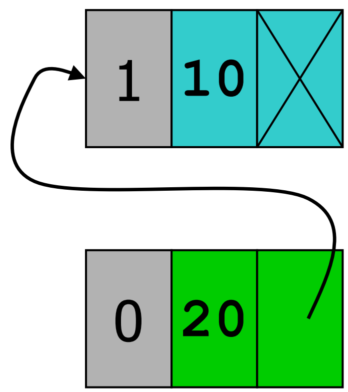
Reference Counting: Example


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



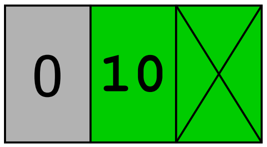
Reference Counting: Example


```
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 = ...
```

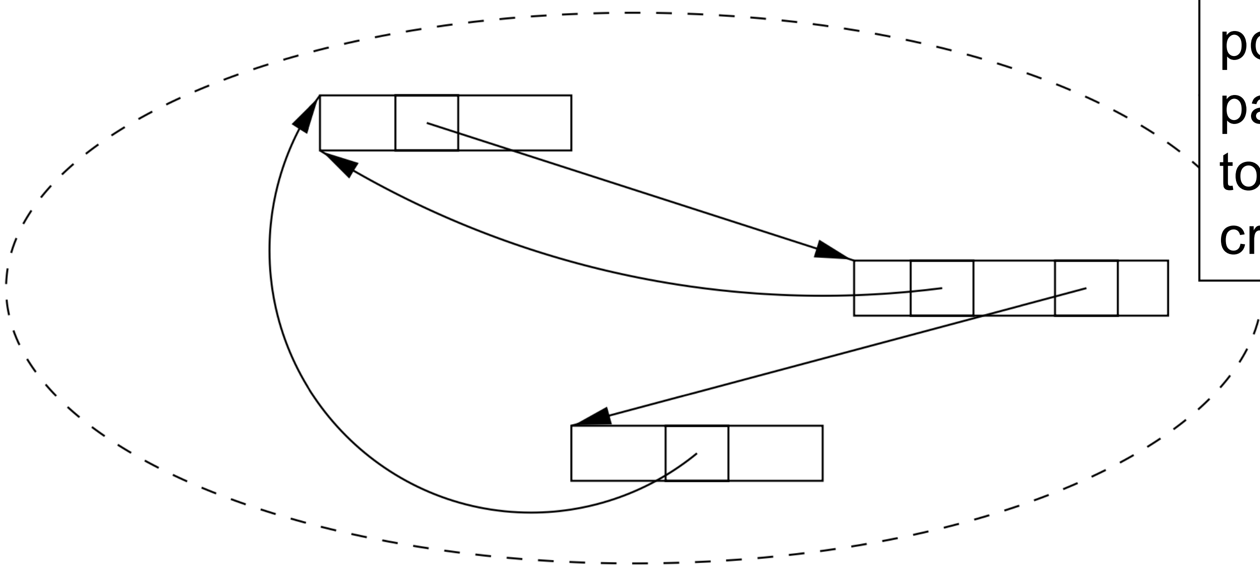


Reference Counting: Example


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

An Unreachable, Cyclic Data Structure

- Three objects with references among them, but no references from anywhere else.



data structures often point back to their parent nodes or point to each other as cross references

- If none of them is part of the root set, then they are all garbage, but their reference counts are each greater than 0.
- Such a situation constitutes a memory leak if we use reference counting for garbage collection.

Reference Counting: Summary

Advantages

- Does not create long pauses
- Memory efficient, because it finds garbage as so on as it is produced
- Simple

Disadvantages

- Has high overheads which is proportional to the amount of computation in the program and not just to the number of objects in the system. It indeed imposes an overhead on every operation that stores a pointer, e.g., a single move operation p = q will need manipulation of two counts.
- Cyclic structures cannot be detected as garbage

GC Without Reference Counts

- If we 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-and-Sweep Collector

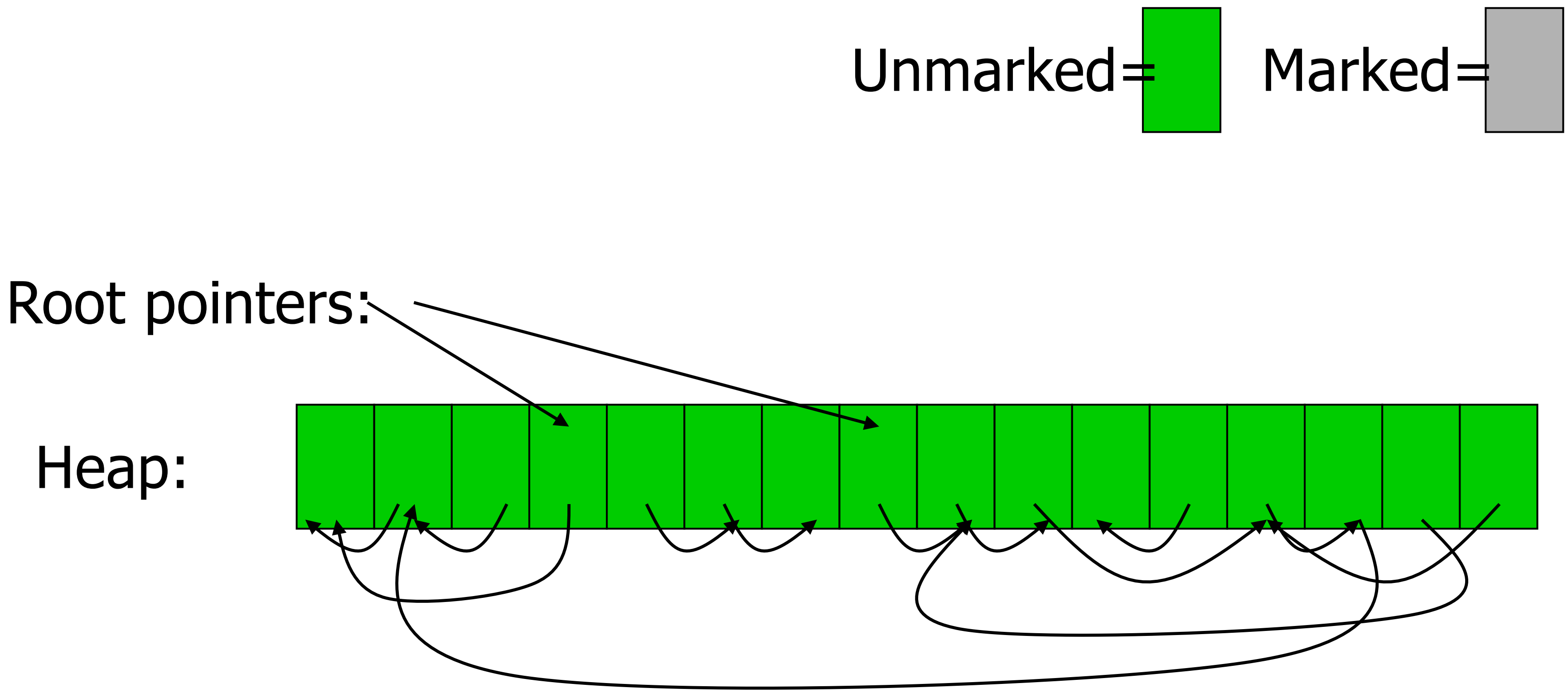
Two-phase collector

- Mark Phase: Does a depth-first traversal of the object graph, starting from the roots
  Marks all objects visited (note reachable nodes represent live data)
- Sweep Phase: Does a sweep over the entire heap, adding any unmarked node to the free list, and removing marks from nodes (preparing for next round)

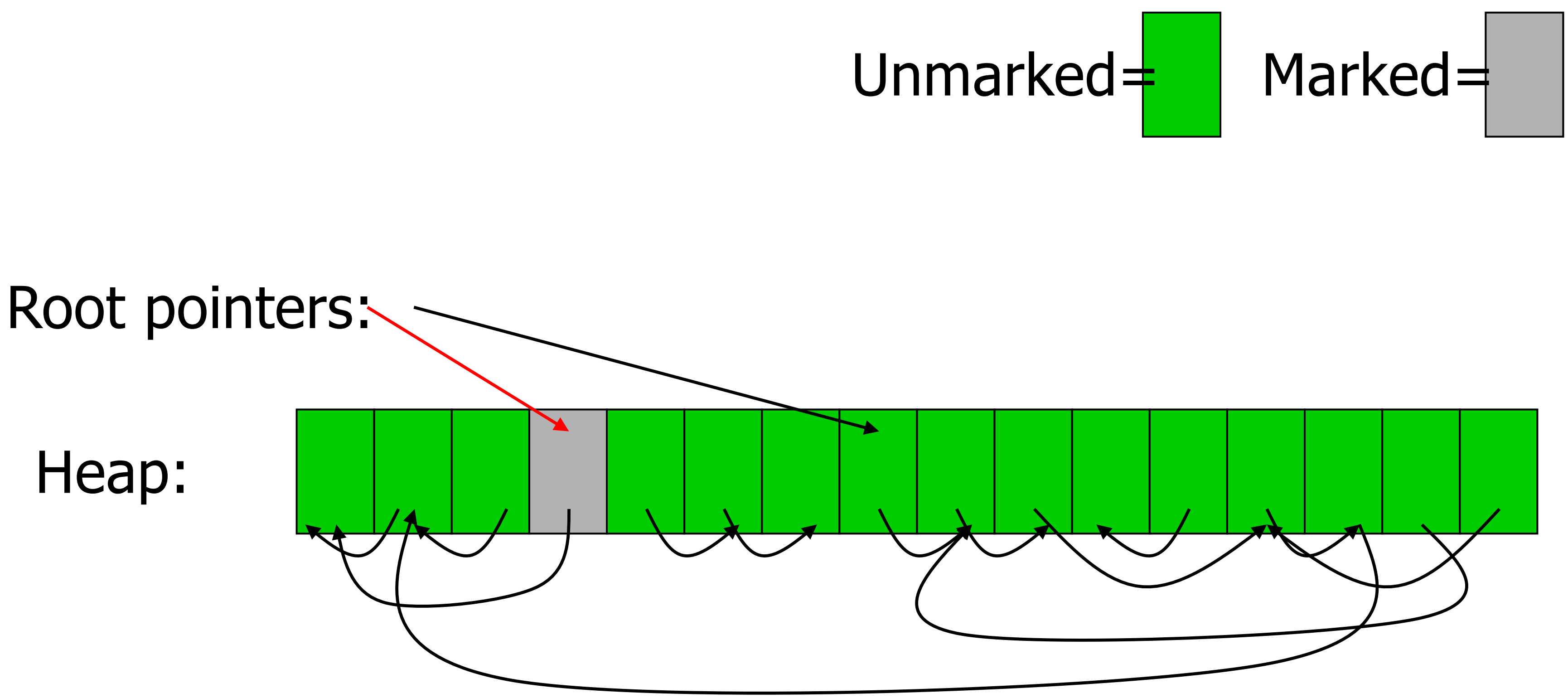
Needs extra bookkeeping space in each object for storing the marks

Mark & Sweep: GC Example

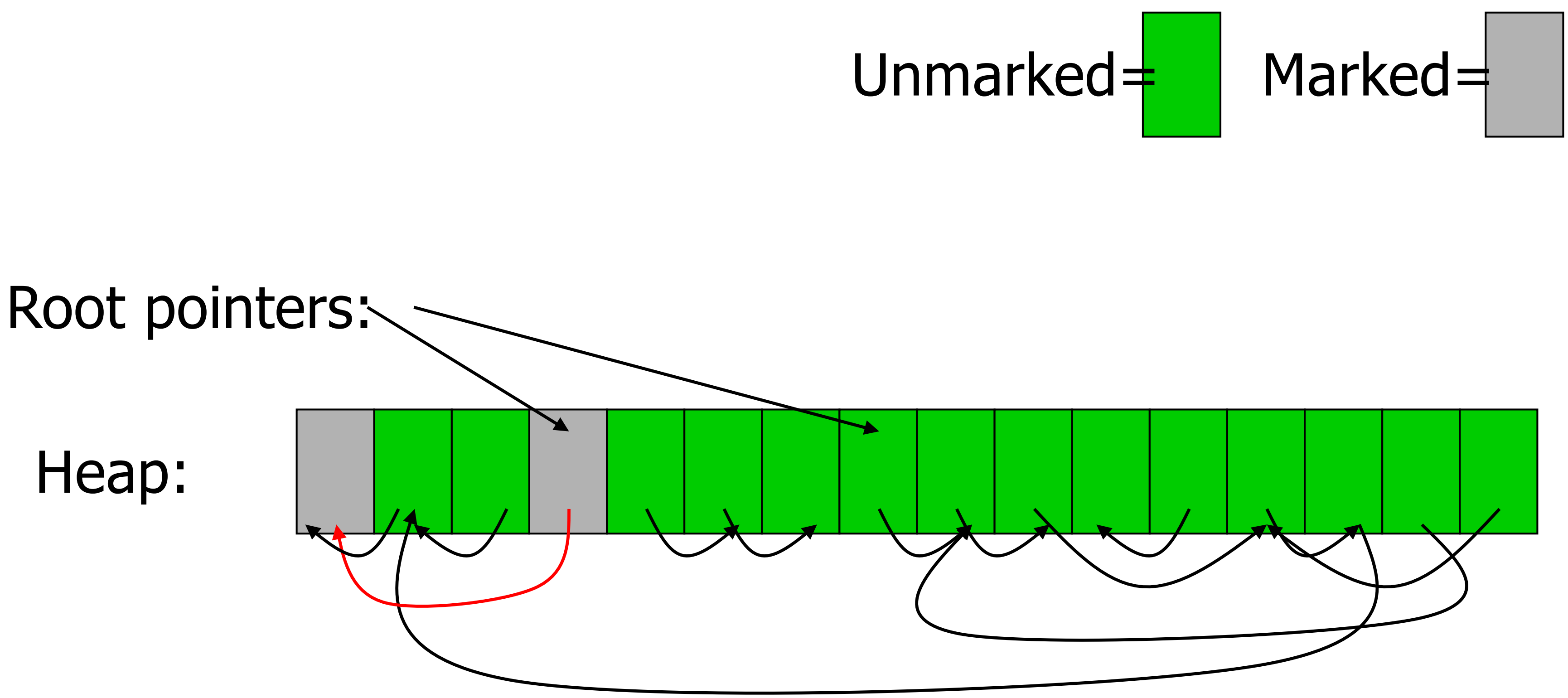
Assume fixed-sized, single-pointer data blocks, for simplicity.



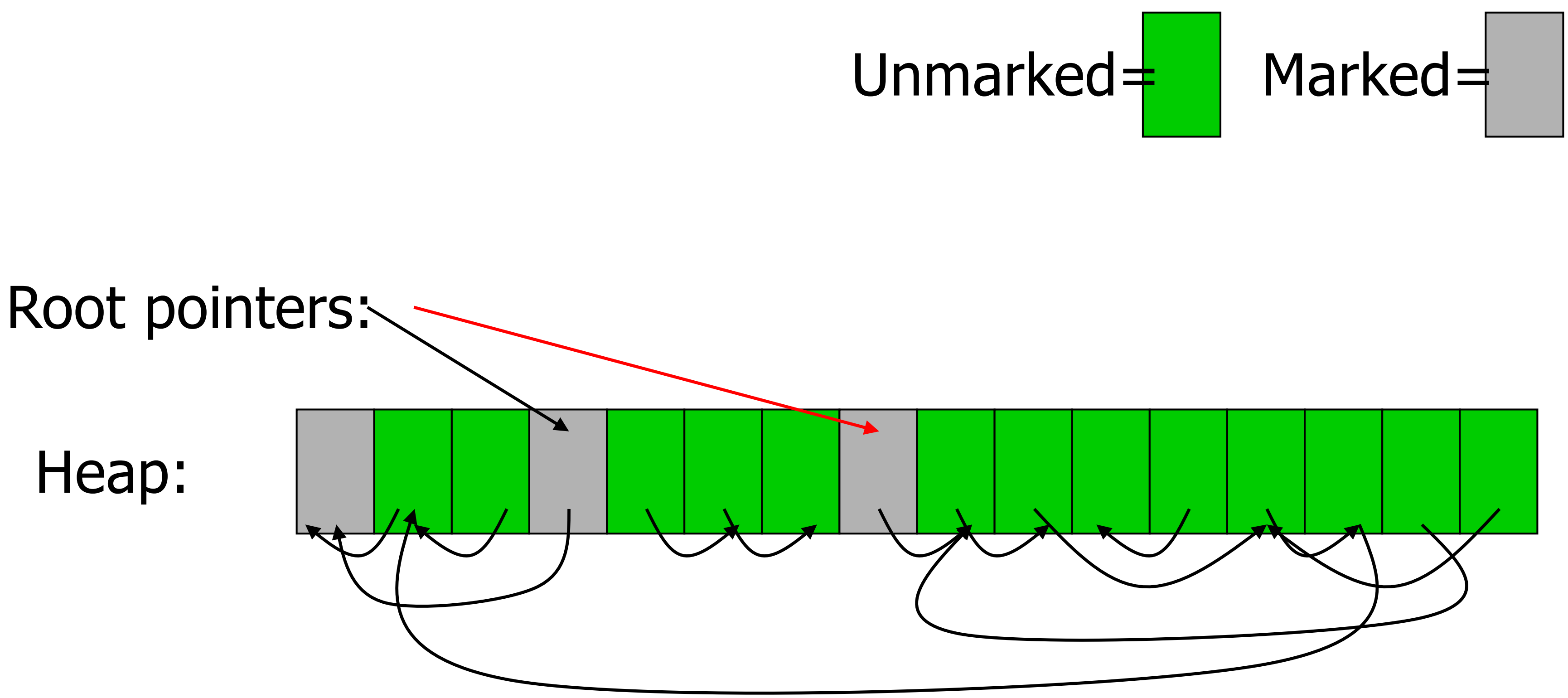
Mark & Sweep: GC Example



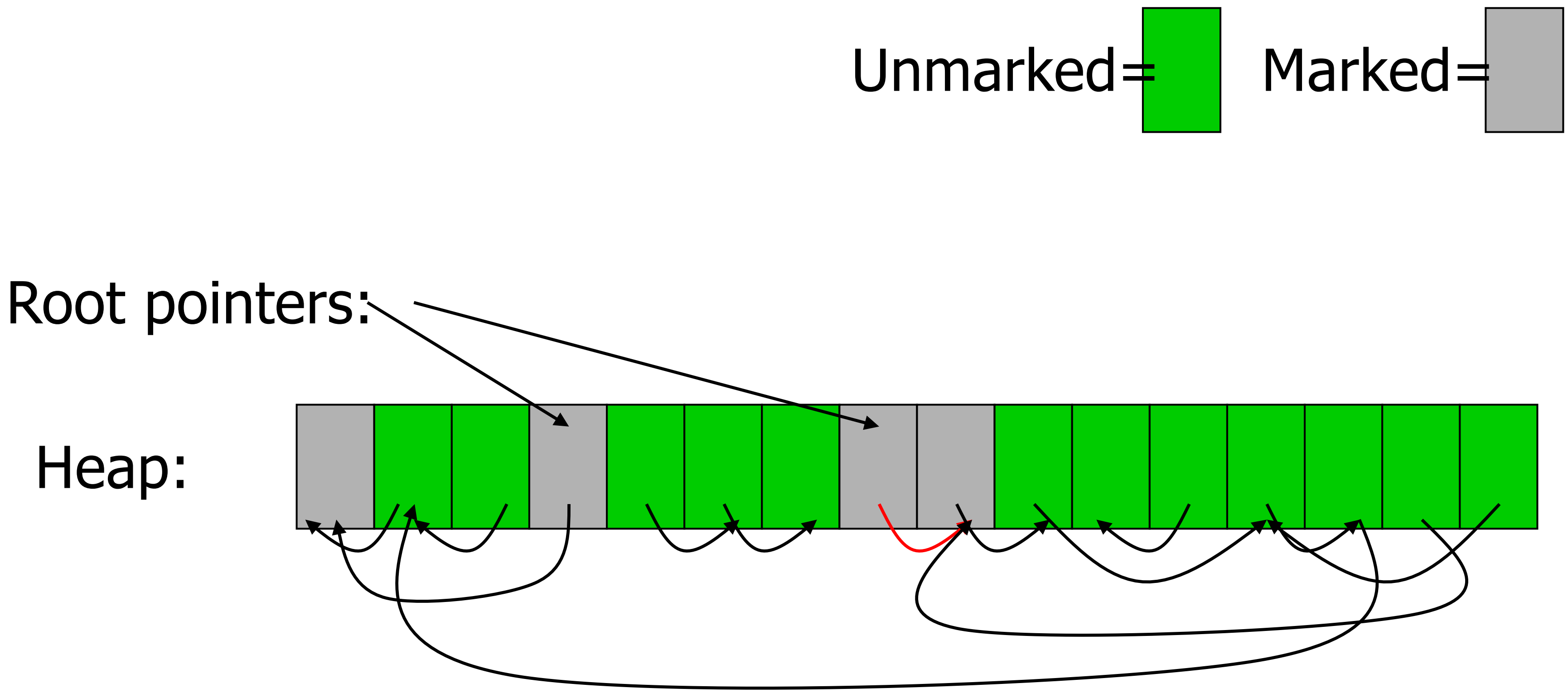
Mark & Sweep: GC Example



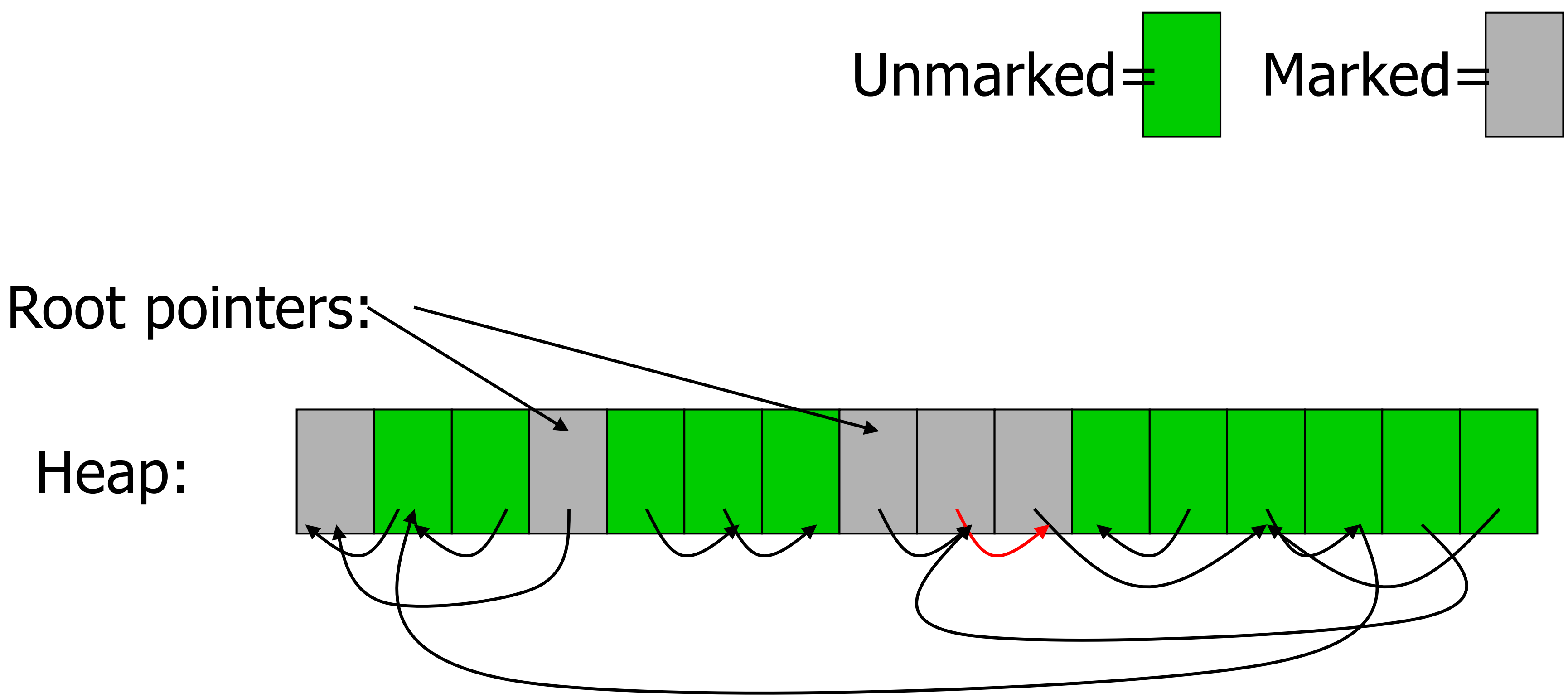
Mark & Sweep: GC Example



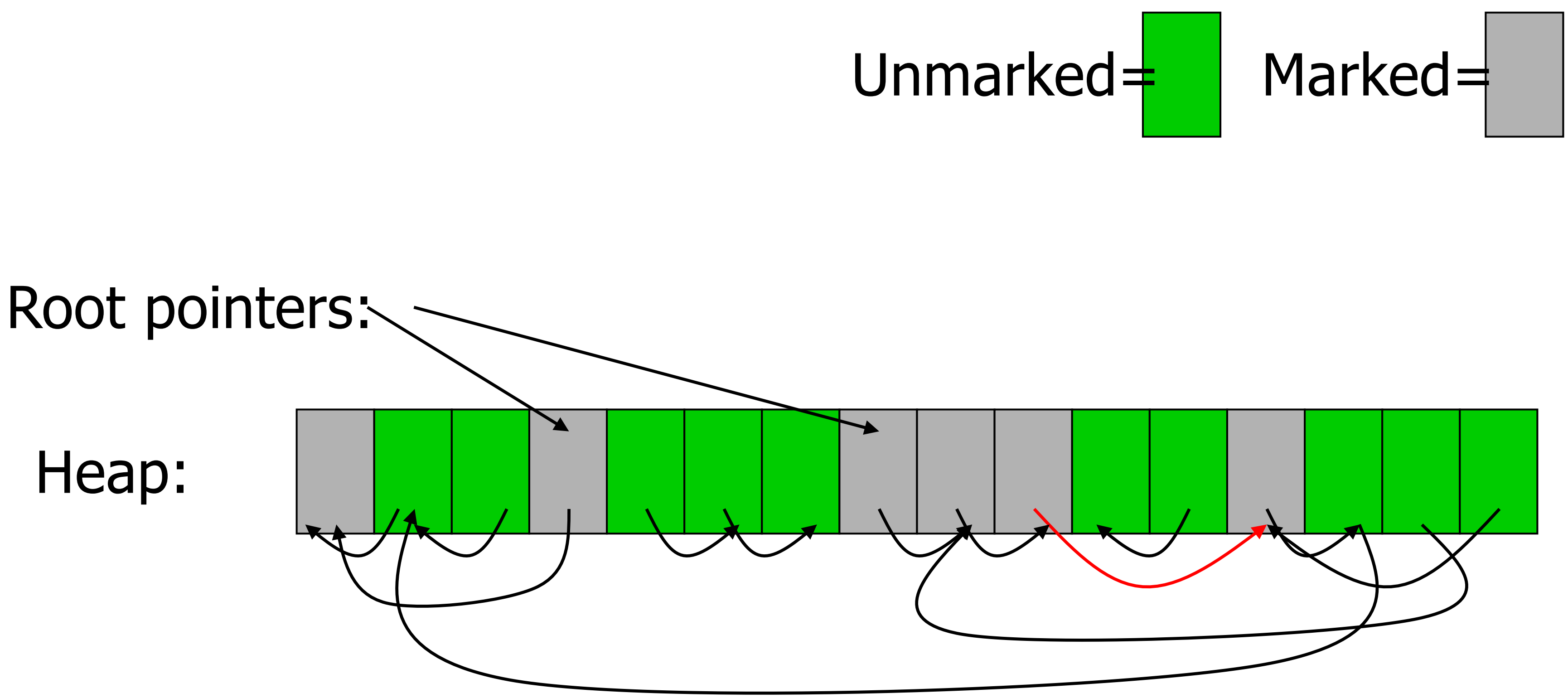
Mark & Sweep: GC Example



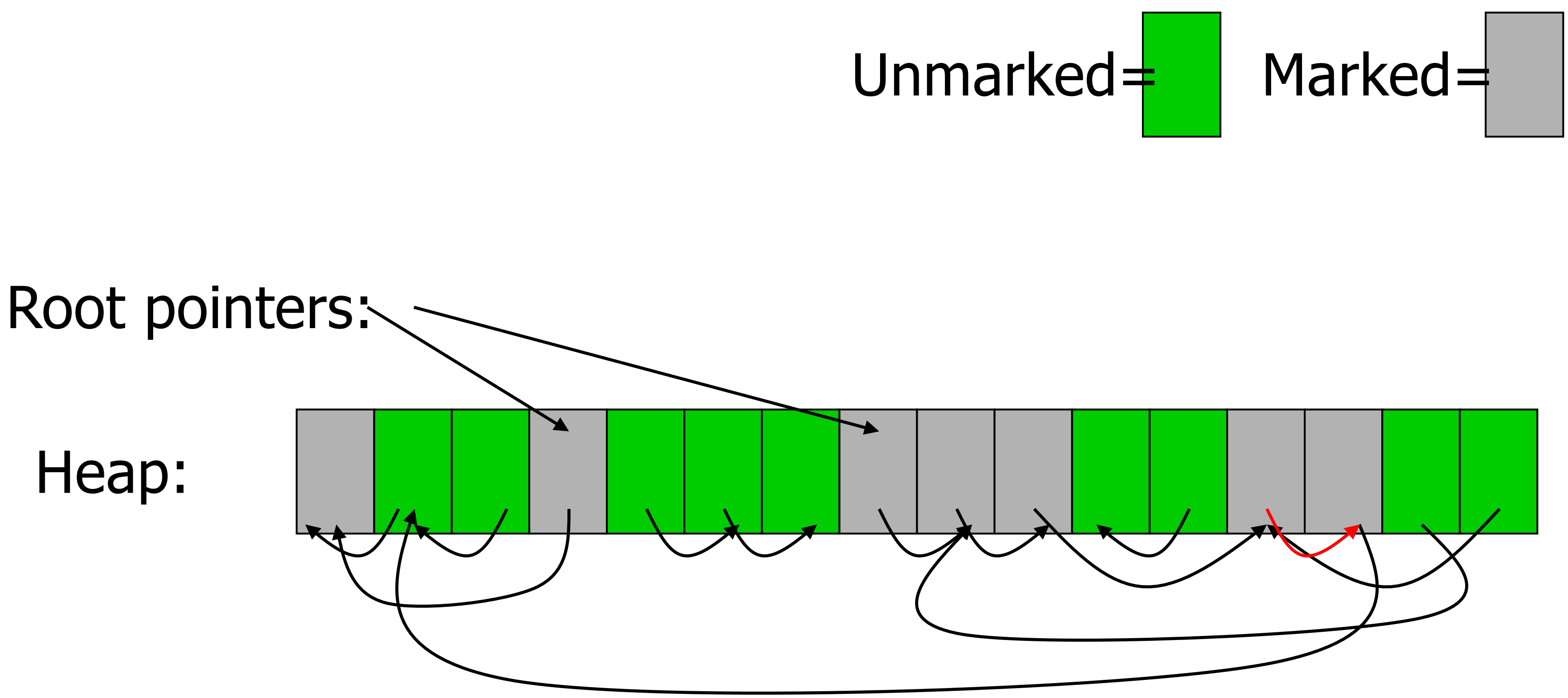
Mark & Sweep: GC Example



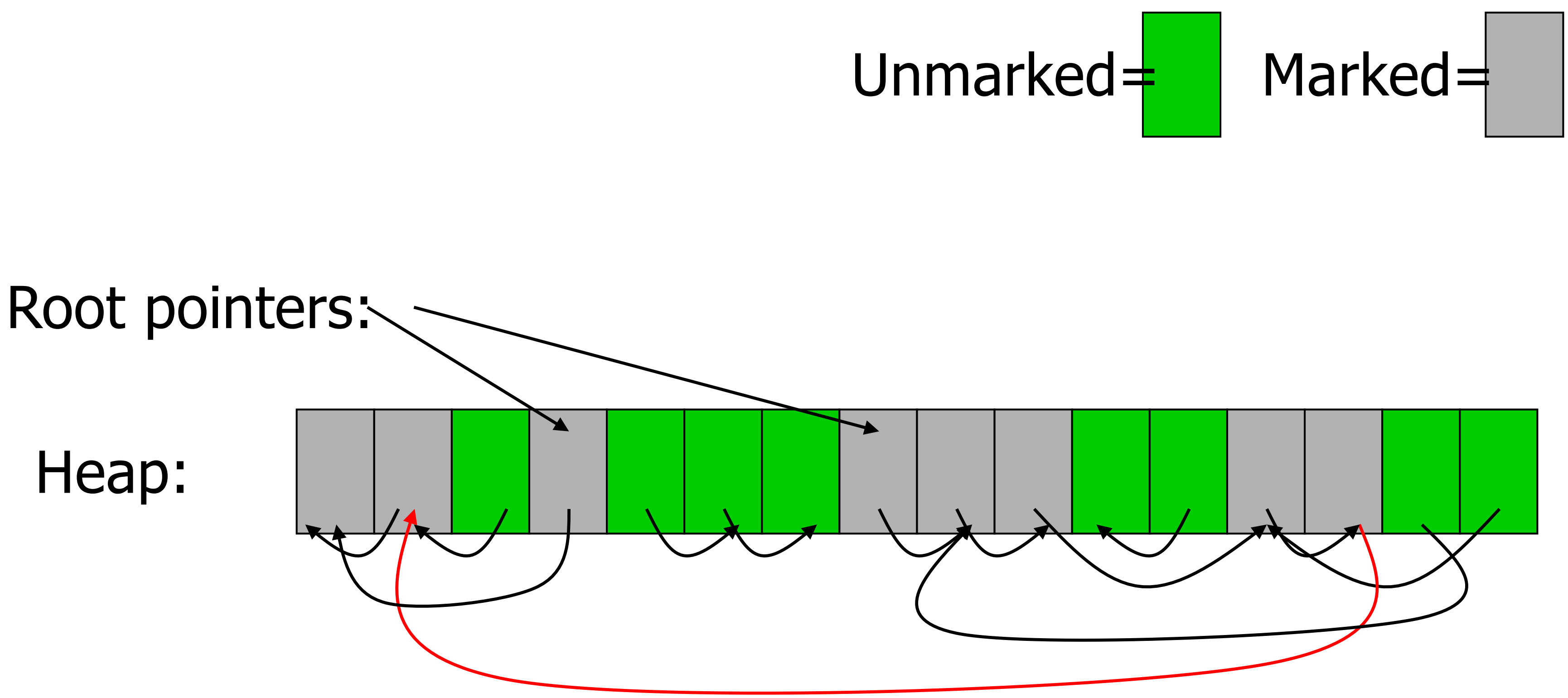
Mark & Sweep: GC Example



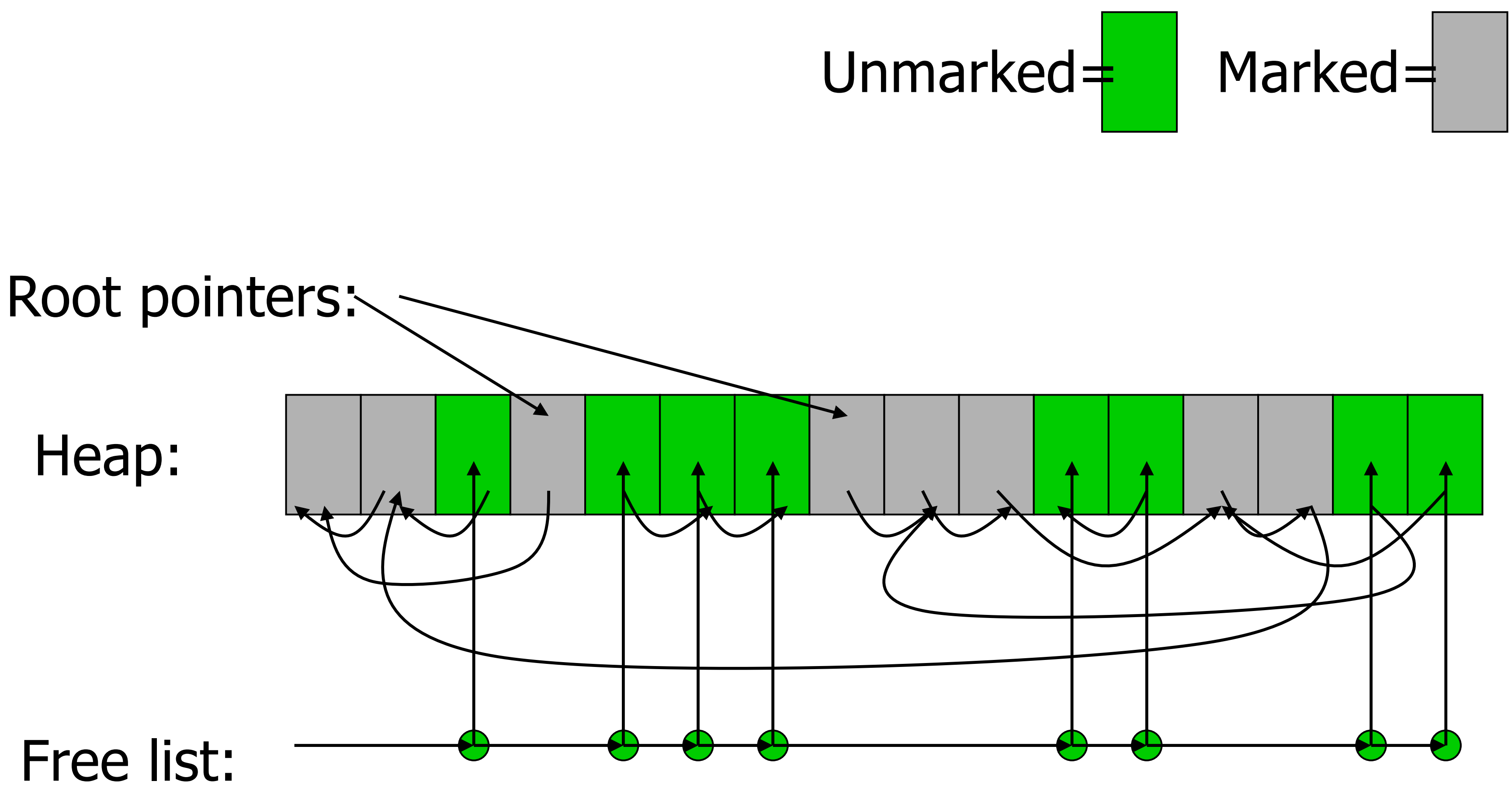
Mark & Sweep: GC Example



Mark & Sweep: GC Example



Mark & Sweep: GC Example



Mark & Sweep: Summary

- Advantages
  - No space overhead for reference counts
  - No time overhead for reference counts
  - Handles cycles

- Disadvantage
  - Cost of collection is proportional to the entire heap size (since sweep traverses the whole heap).
  - Noticeable pauses for GC

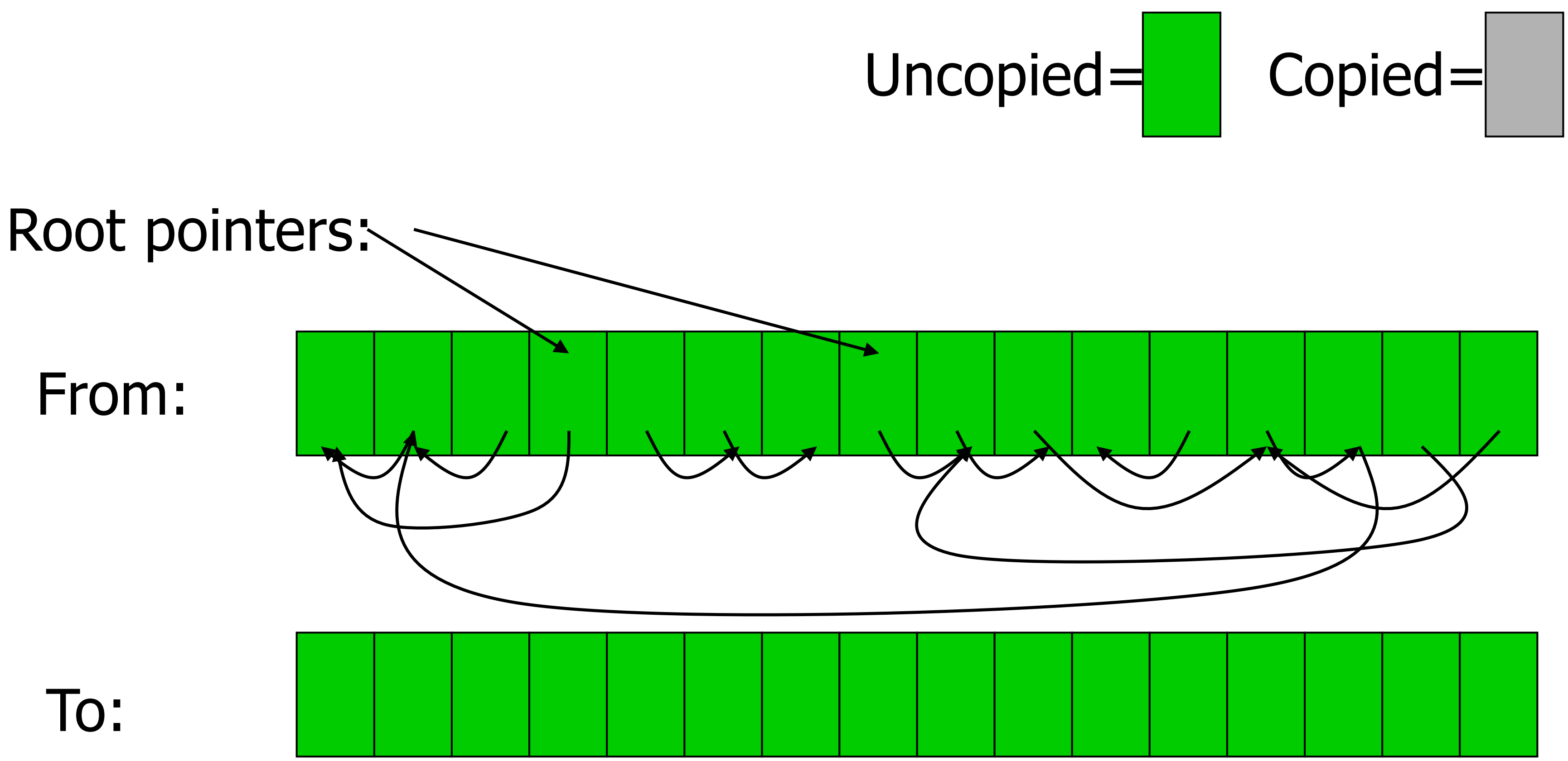
Stop & Copy Garbage Collector

Two-Space Collector

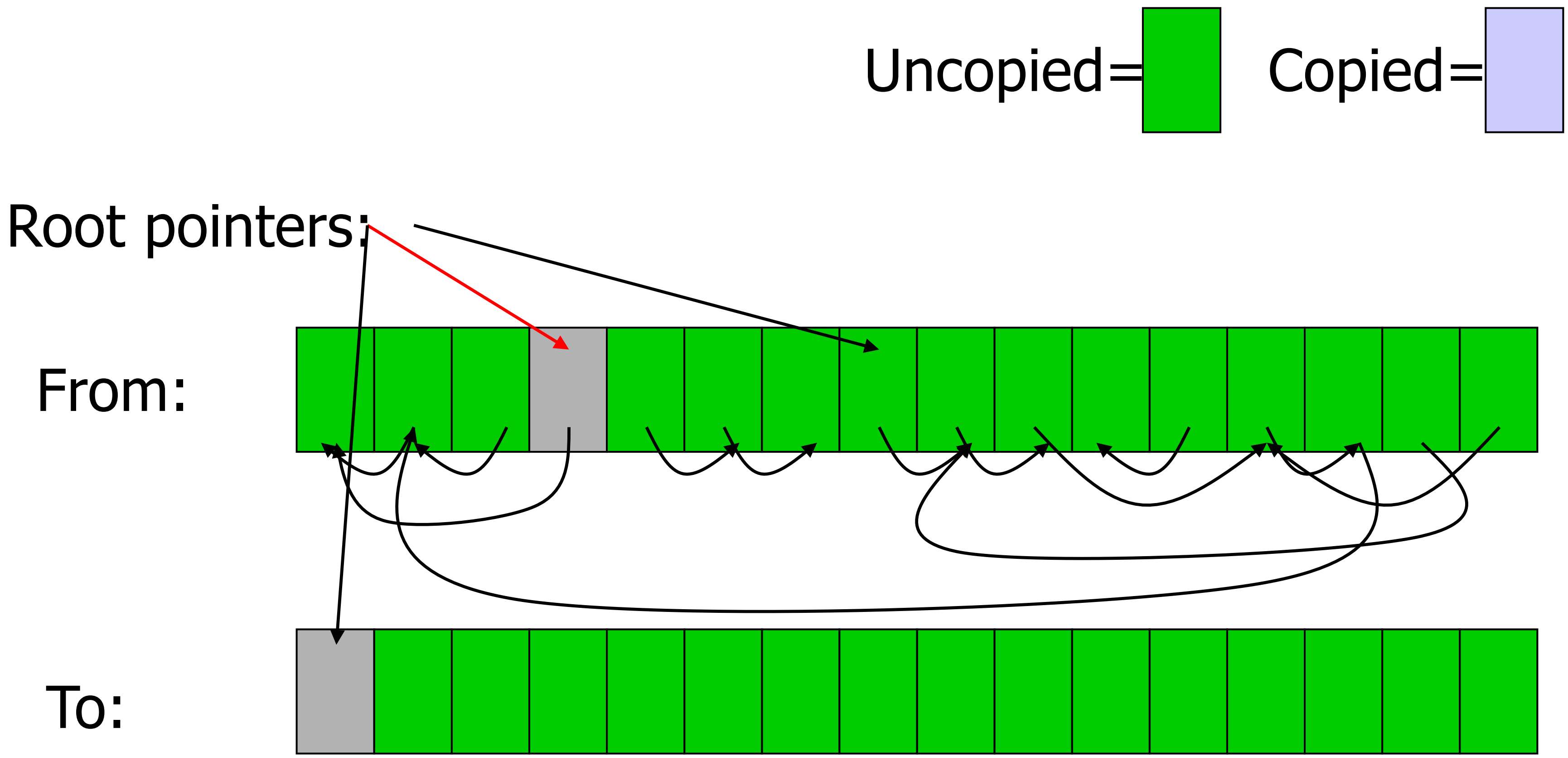
- Heap is divided into two spaces
  - ***From*** Space: The currently active heap
  - ***To*** Space: Space to which objects will be copied (currently inactive)
- Objects reached are copied from the ***From*** Space to ***To*** Space
- References to copied objects are modified during the traversal
- ***From*** and ***To*** spaces are swapped at the end of copying

Stop & Copy: GC Example

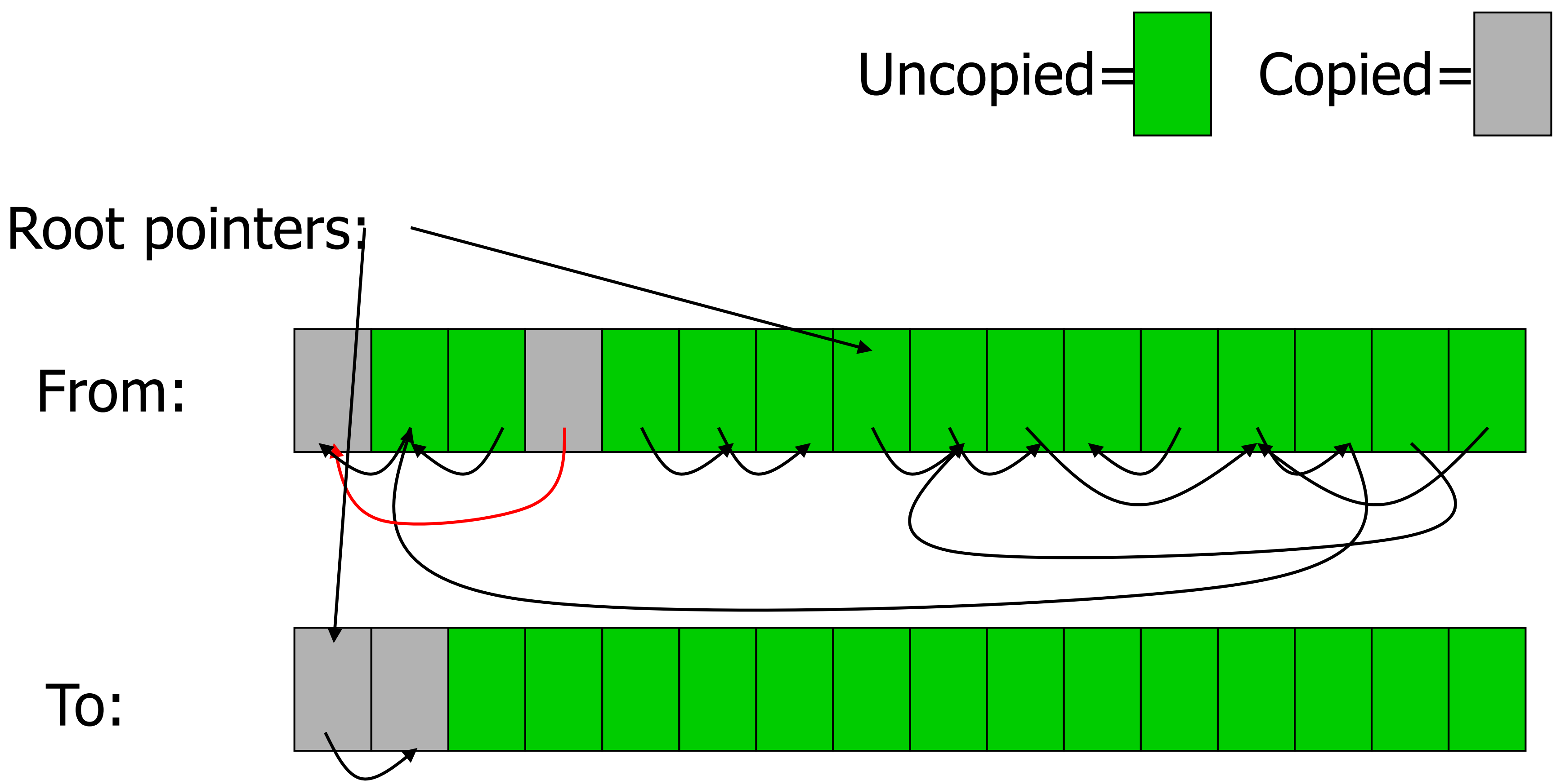
Assume fixed-sized, single-pointer data blocks, for simplicity.



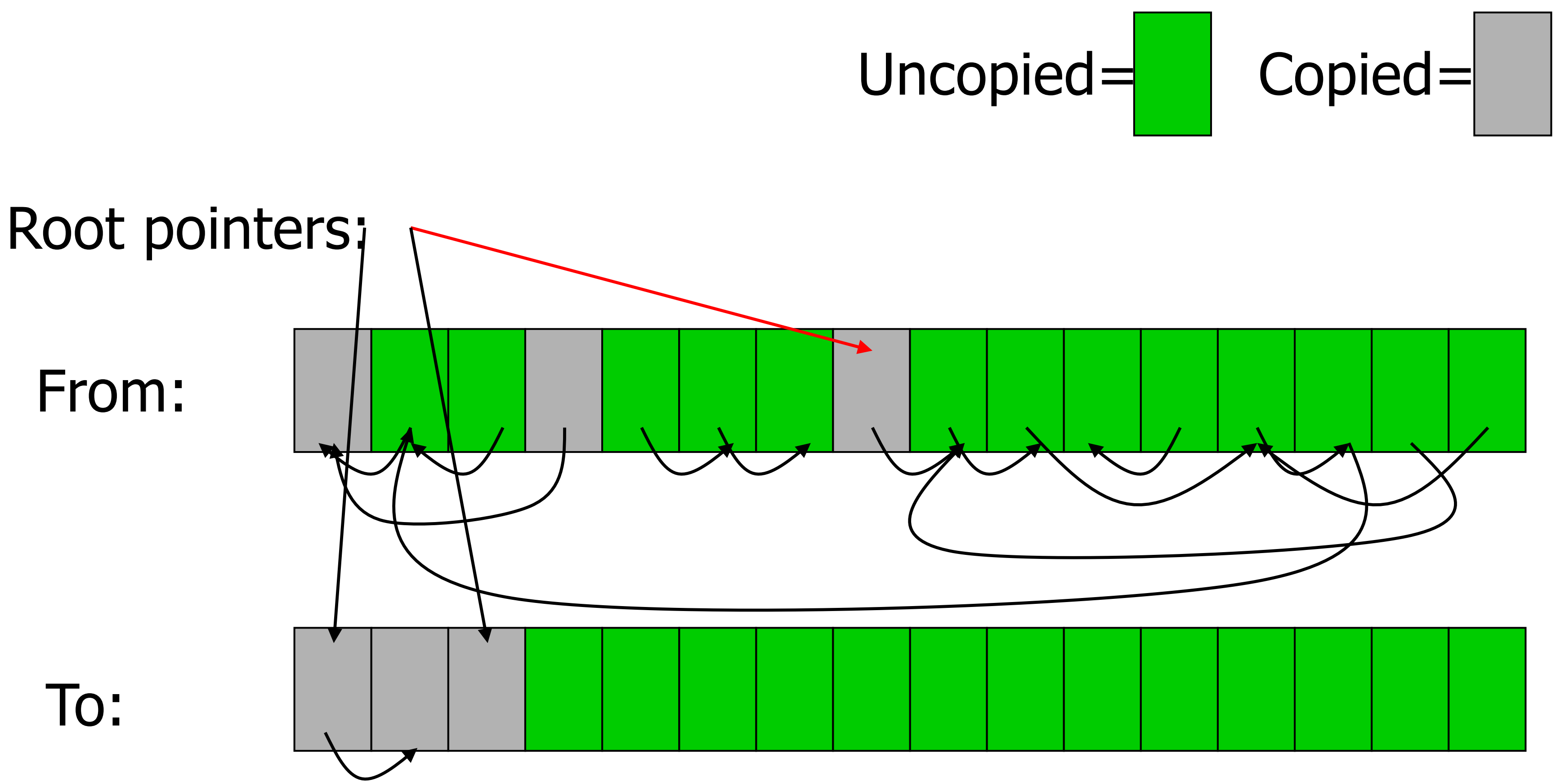
Stop & Copy: GC Example



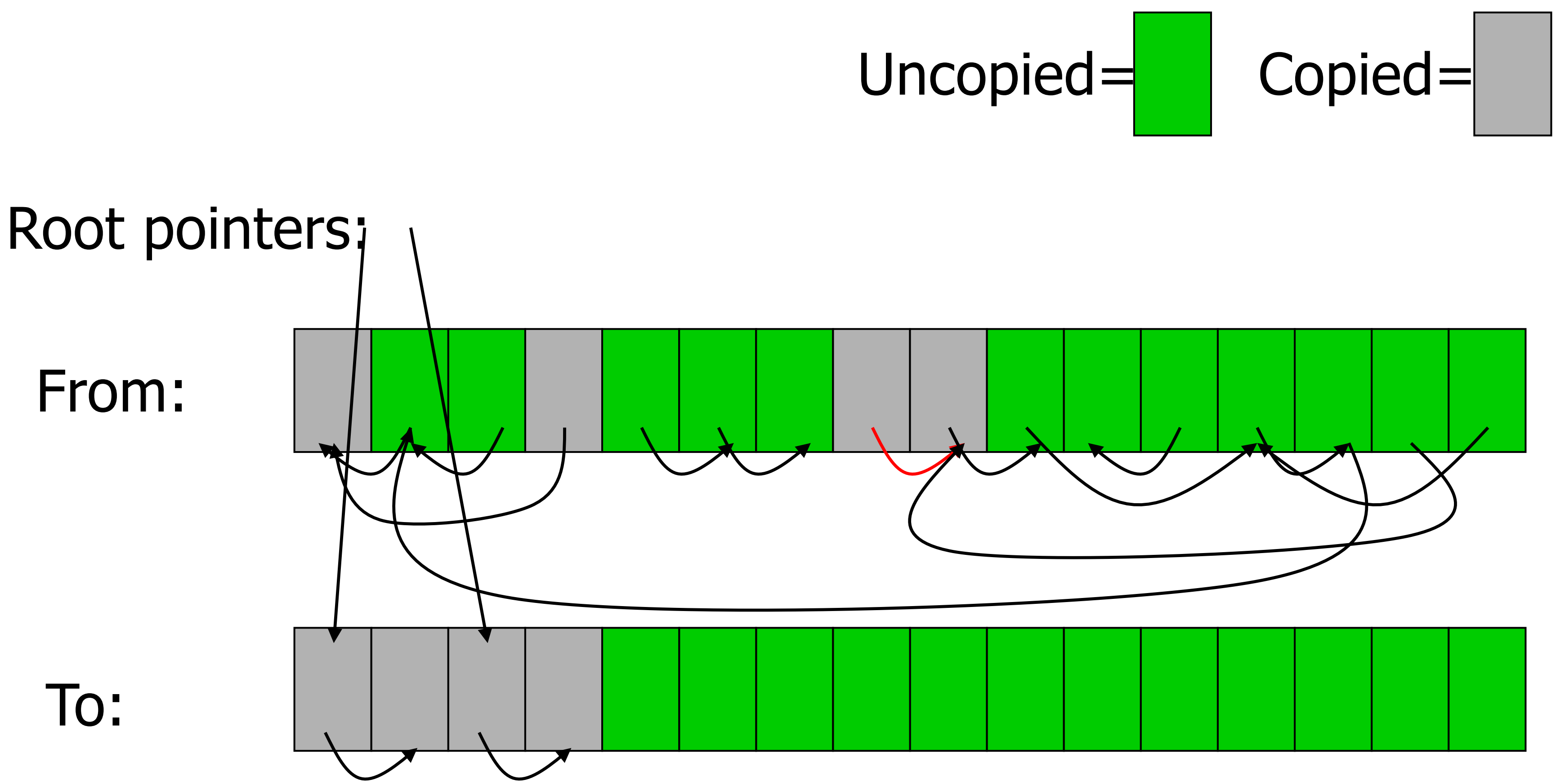
Stop & Copy: GC Example



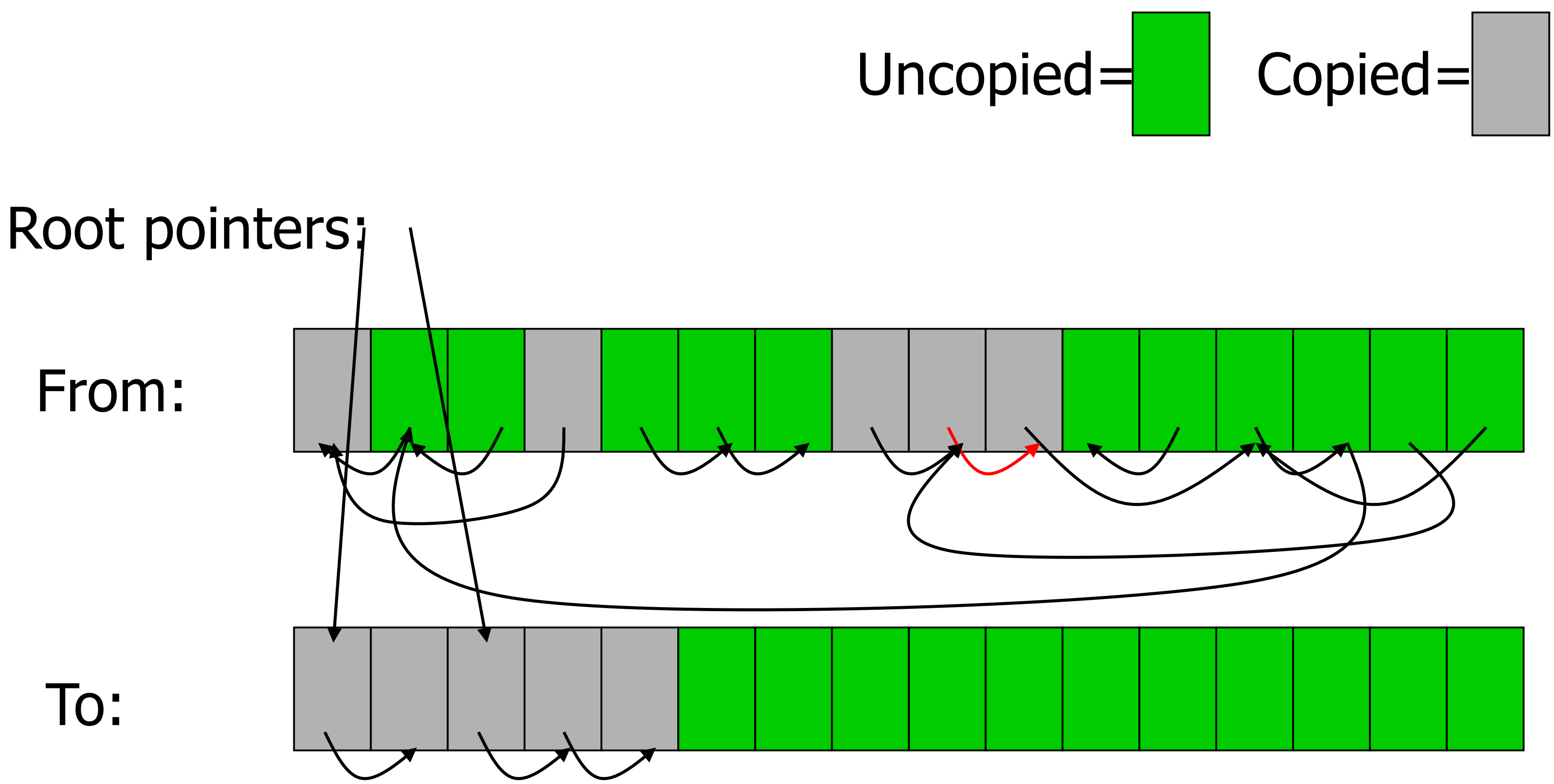
Stop & Copy: GC Example



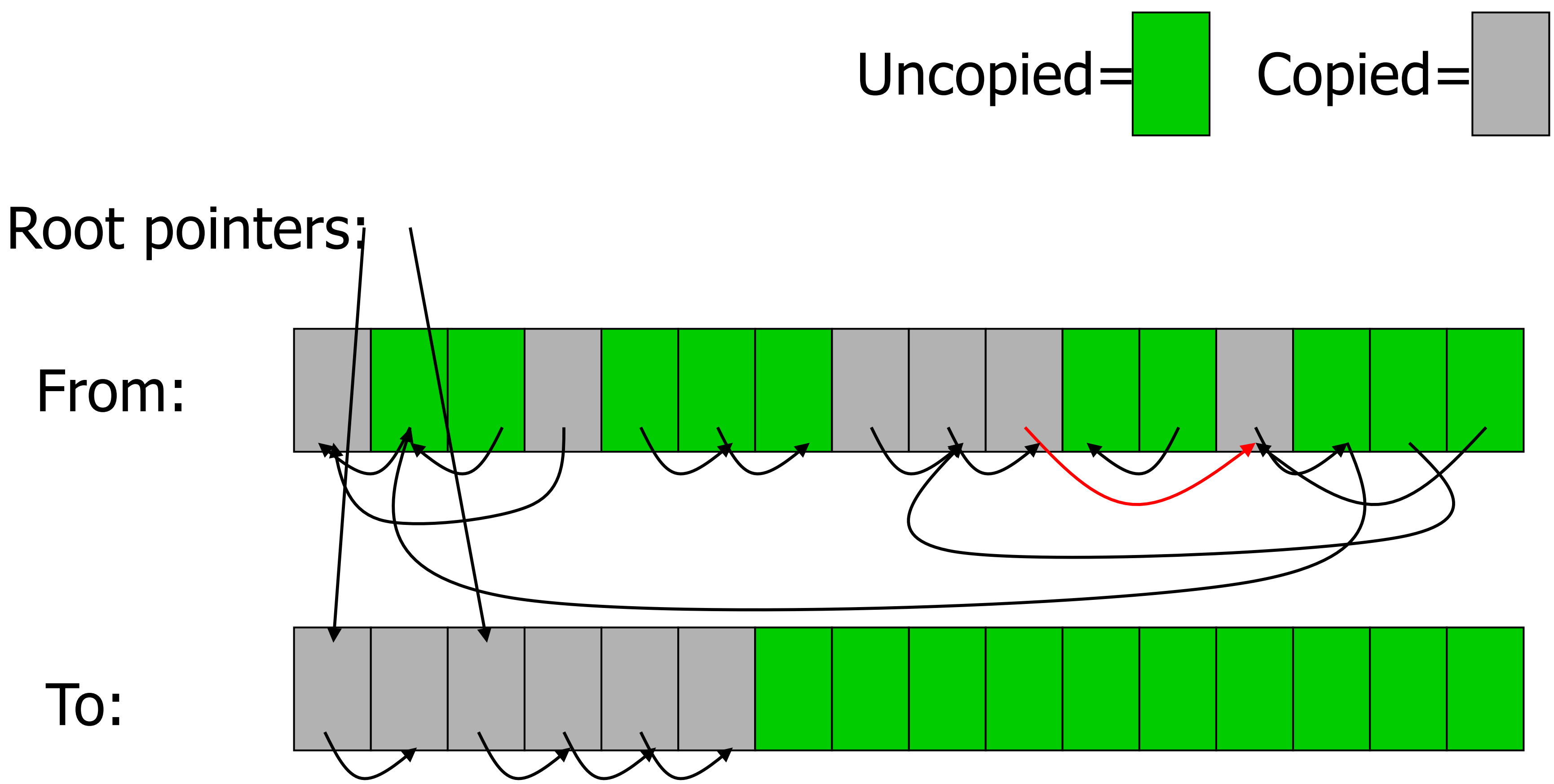
Stop & Copy: GC Example



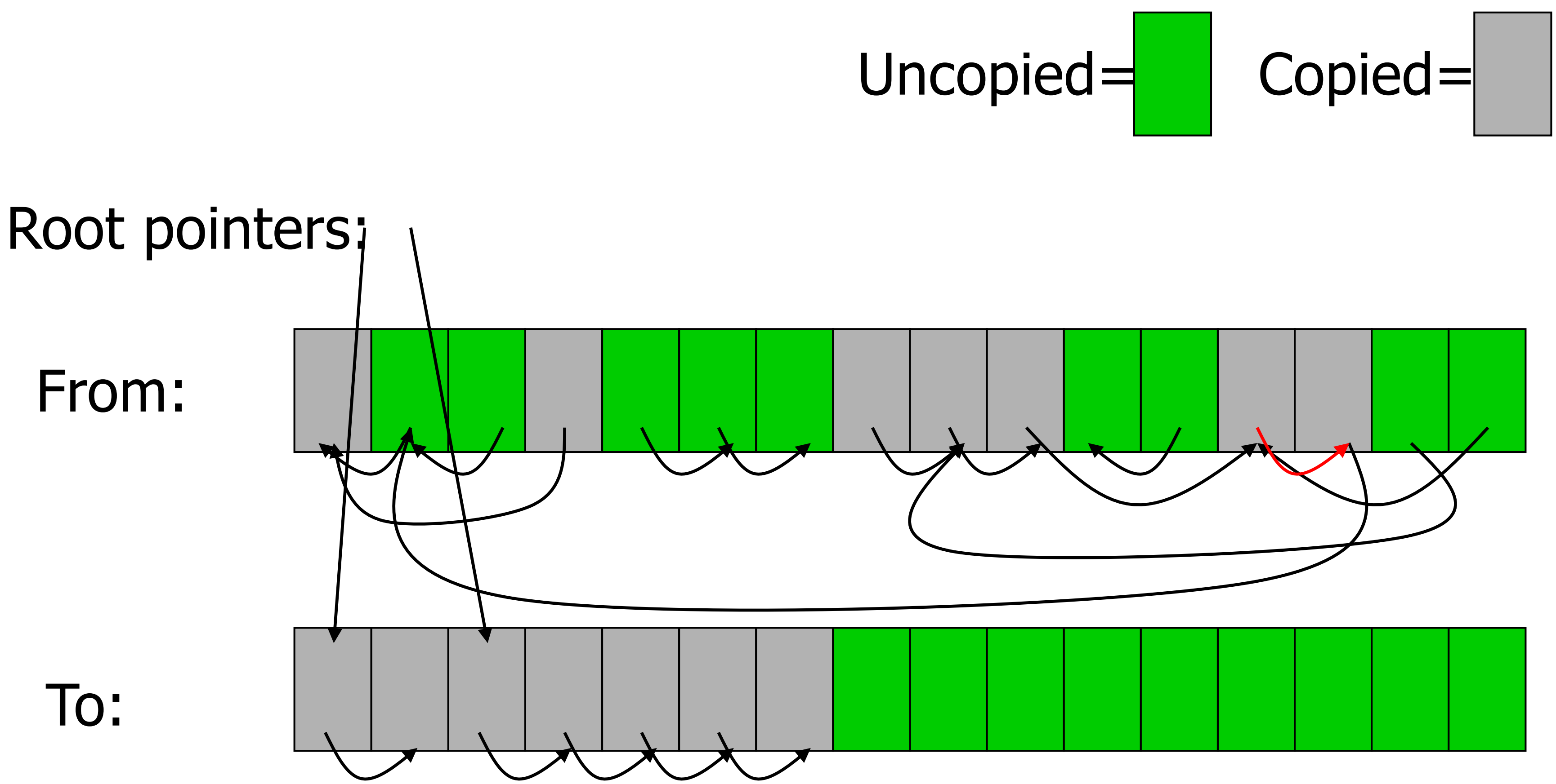
Stop & Copy: GC Example



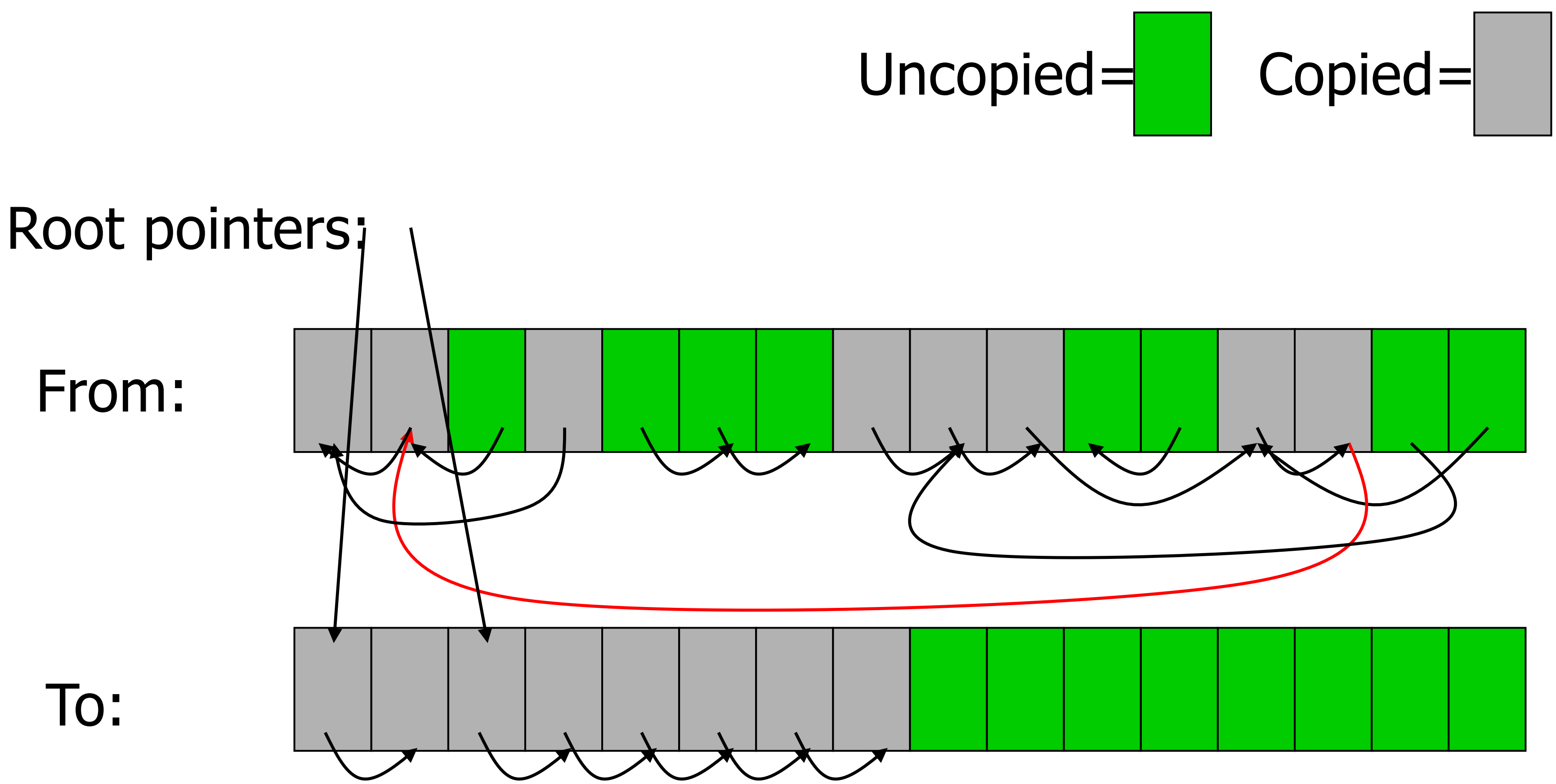
Stop & Copy: GC Example



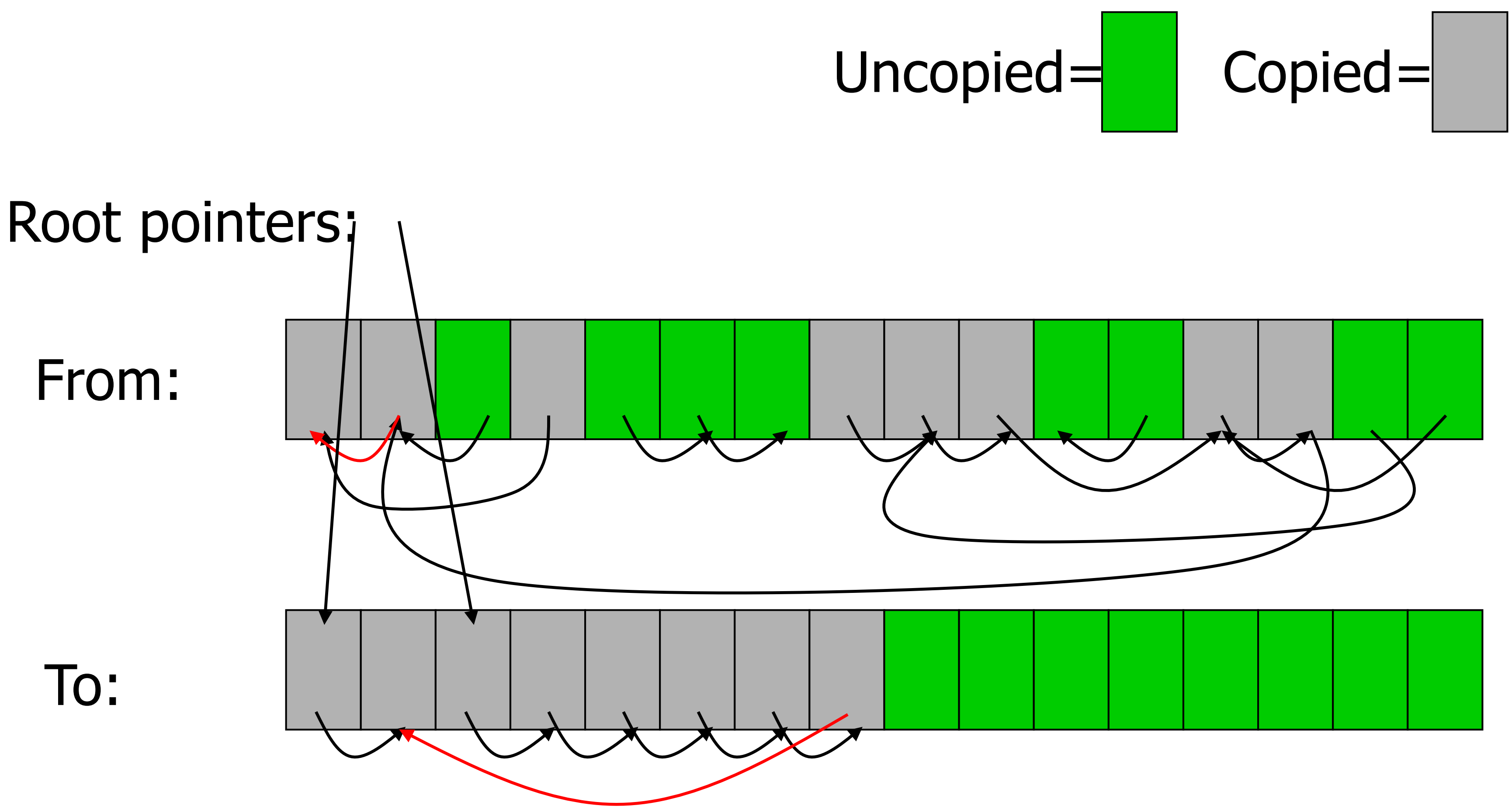
Stop & Copy: GC Example



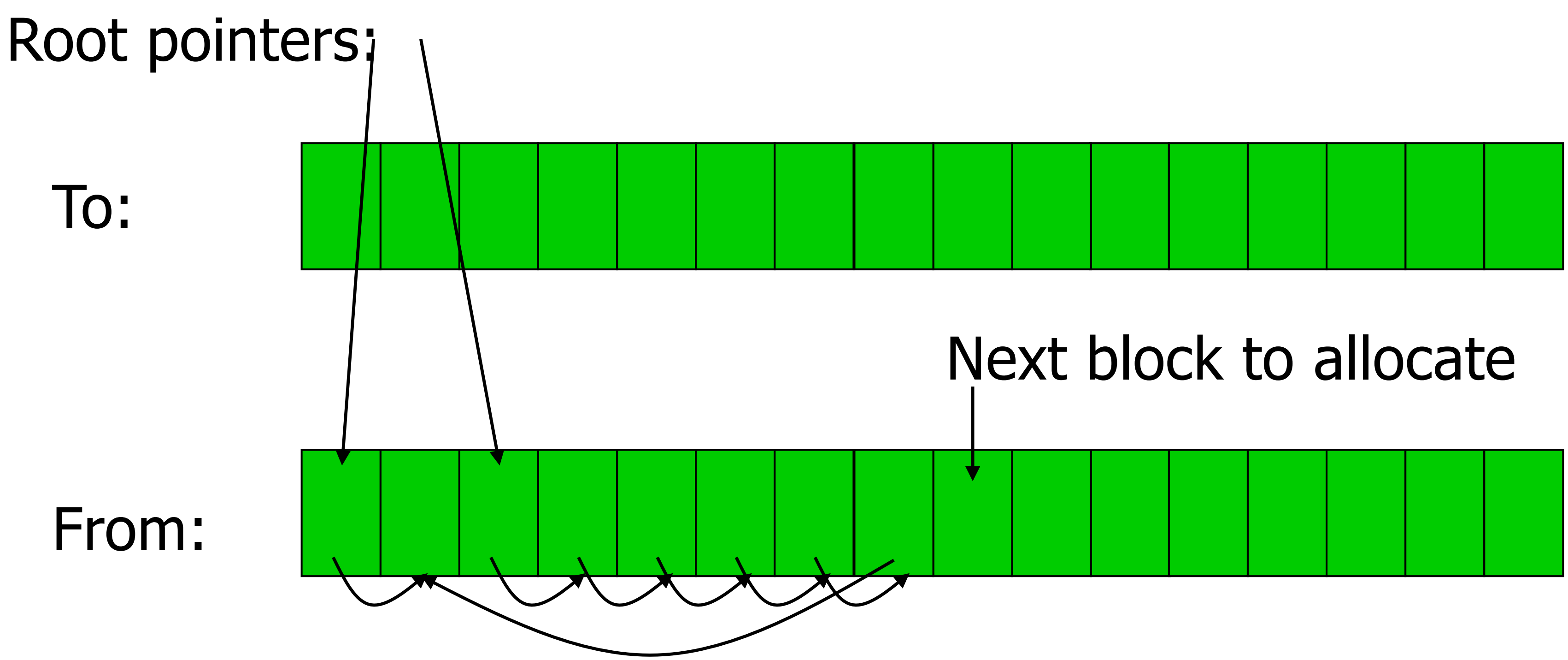
Stop & Copy: GC Example



Stop & Copy: GC Example



Stop & Copy: GC Example



Stop & Copy GC

- Needs more heap space than is currently used, but
  - Memory is compacted during copy, and hence no fragmentation
- Cost of collection is proportional to size of live objects in heap (unreachable objects are not touched).
- Objects that survive a collection may get copied repeatedly, which is expensive.
- Often used as a part of a **generational garbage collector**

Stop & Copy GC

- Advantages
  - Handles cycles
  - "Compacts" data, tends to increase spatial locality
  - Very simple allocation
- Disadvantages
  - Noticeable pauses for GC
  - Doubles the basic heap size