
CMPSC 160
Translation of Programming Languages

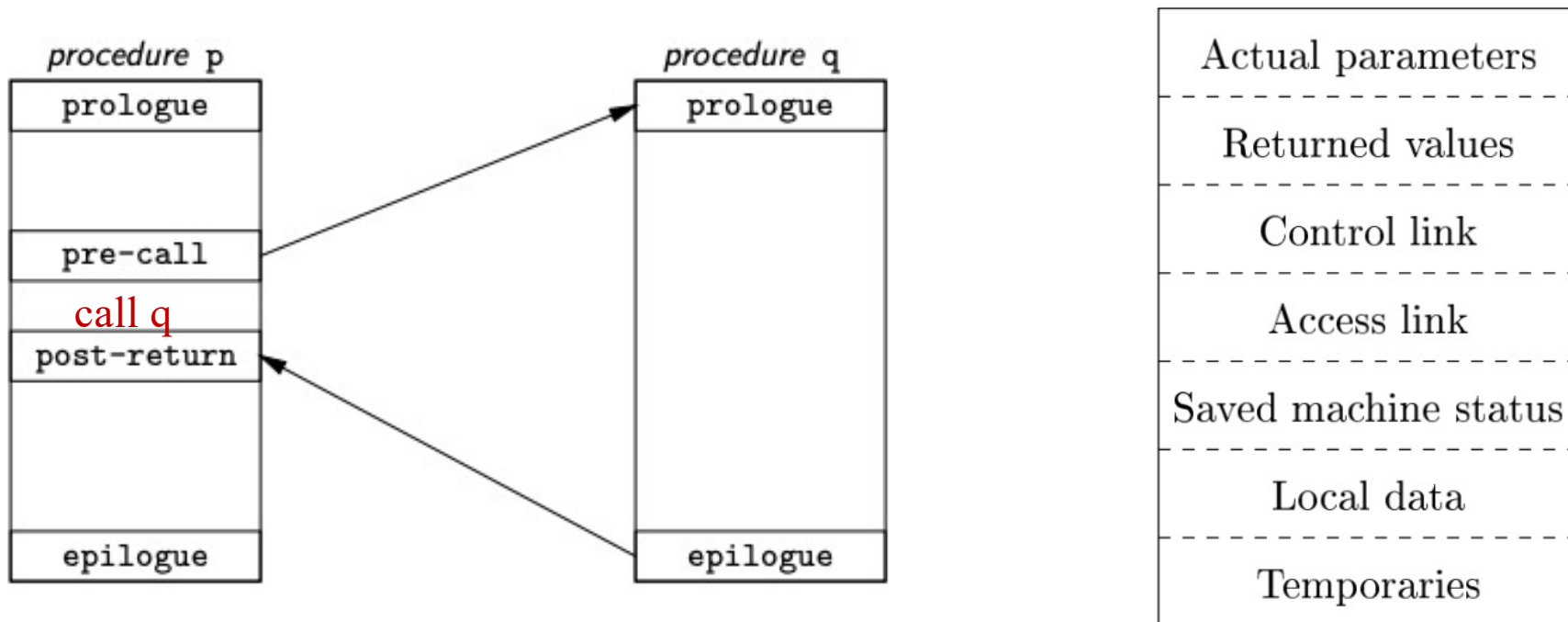
Lecture 13:
Heap Management and Garbage Collection

Recap on Last Lecture

- Stack management for procedures (and methods, classes) implementations.
 - Keys for implementing procedures with stack
 - One copy of code + dynamic copies of data on stack at runtime
 - the relative addresses of local data in the code can be generated at compiler time, so we only need a copy of code.
 - accommodate a single callee functions called by different caller functions.
 - accommodate recursive calls
 - accommodate a functions called by different parameters
 - Data for the procedure is stored on stack, also called **activation record**.
 - Stores the key information for us to restore the execution of the callee (often also filled in by the callee).
 - Also stores the local data and temporizes for the callee (often filled by the caller)
-

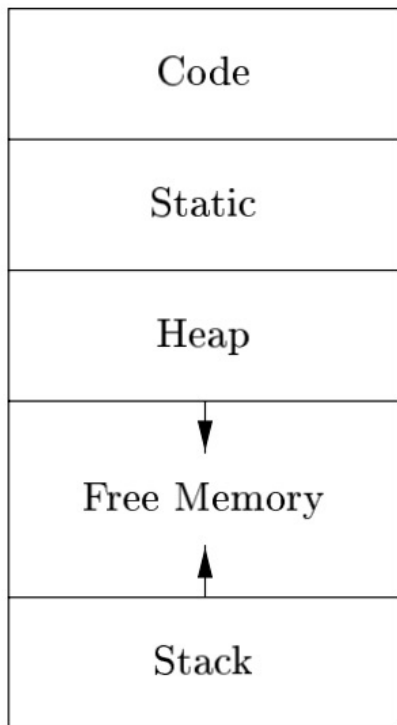
Recap on Last Lecture

- These four (prologue, ...) are the compiler-generated code for manipulating the stack at runtime.



An activation record for a procedure on stack

Another Dynamic memory: Heap

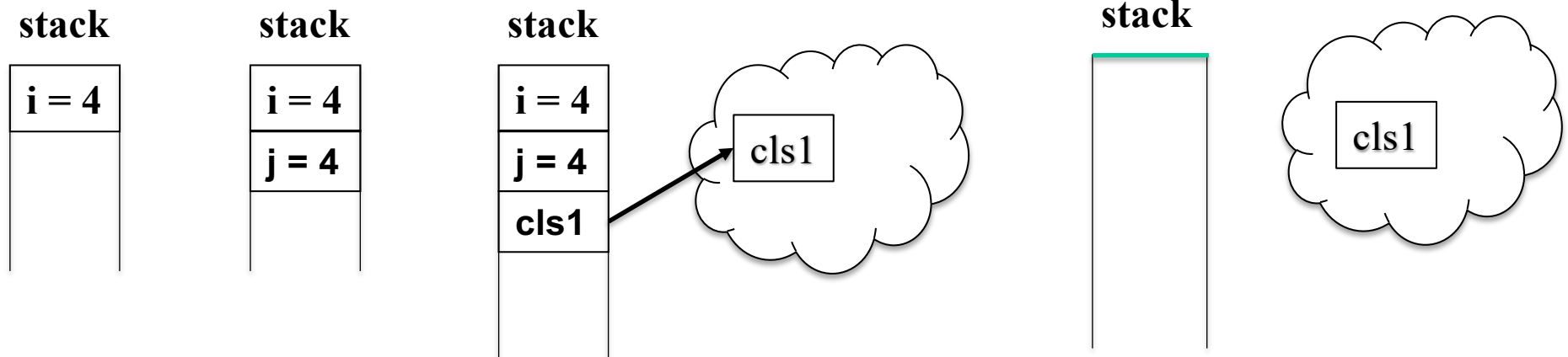


- Heap: The heap is another memory set aside for dynamic allocation. Data on heap have lifetimes that may differ from the life of a procedure call. Memory for heap data is allocated on demand (e.g., ***malloc***, ***new***, etc.) . . .
. . . and released
 - Manually: e.g., using `free`
 - Automatically: e.g., using a garbage collector

Another Dynamic memory: Heap

```
public void Method1()  
{  
    int i = 4;  
    int j = 2;  
    class cls1 = new class();  
}
```

Intermixed example of both
kind of memory allocation
Heap and Stack in *java*.



Stack VS. 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 "**safe**" and "**fast**" as Stack memory allocation was because the data stored in this space is accessible or visible out of a procedure.
 - Performance. Size of Heap-memory is quite larger as compared to the Stack-memory. The processing time (accessing time) of this memory is quite slow as compared to Stack-memory.
 - Safety. Different from stack memory management, no efficient, automatic de-allocation feature is provided. "Safer concerns" raise if not handled well. For example, We receive the corresponding error message if Heap-space is entirely full, for example, `java.lang.OutOfMemoryError` by JVM.
-

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 were 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))
```



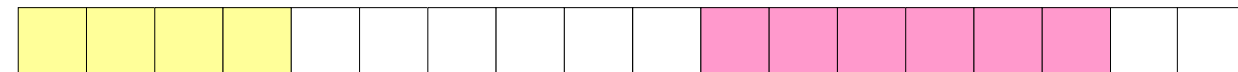
```
p2 = malloc(5*sizeof(int))
```



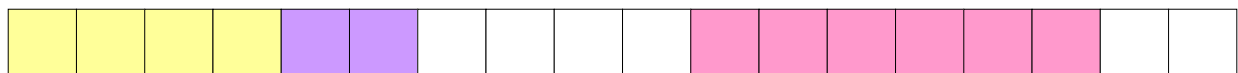
```
p3 = malloc(6*sizeof(int))
```



```
free(p2)
```

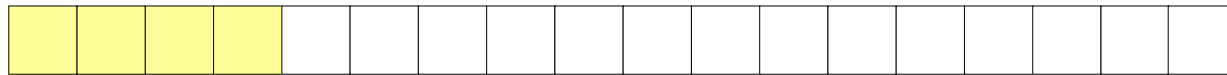


```
p4 = malloc(2*sizeof(int))
```



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 were for chunks of the **different sizes**. (2) There is no good way to predict the **lifetimes** of all allocated objects.

Our focus

- **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 do not return memory to the operating system even if the program's heap usage drops.
-

Manual Memory Deallocation

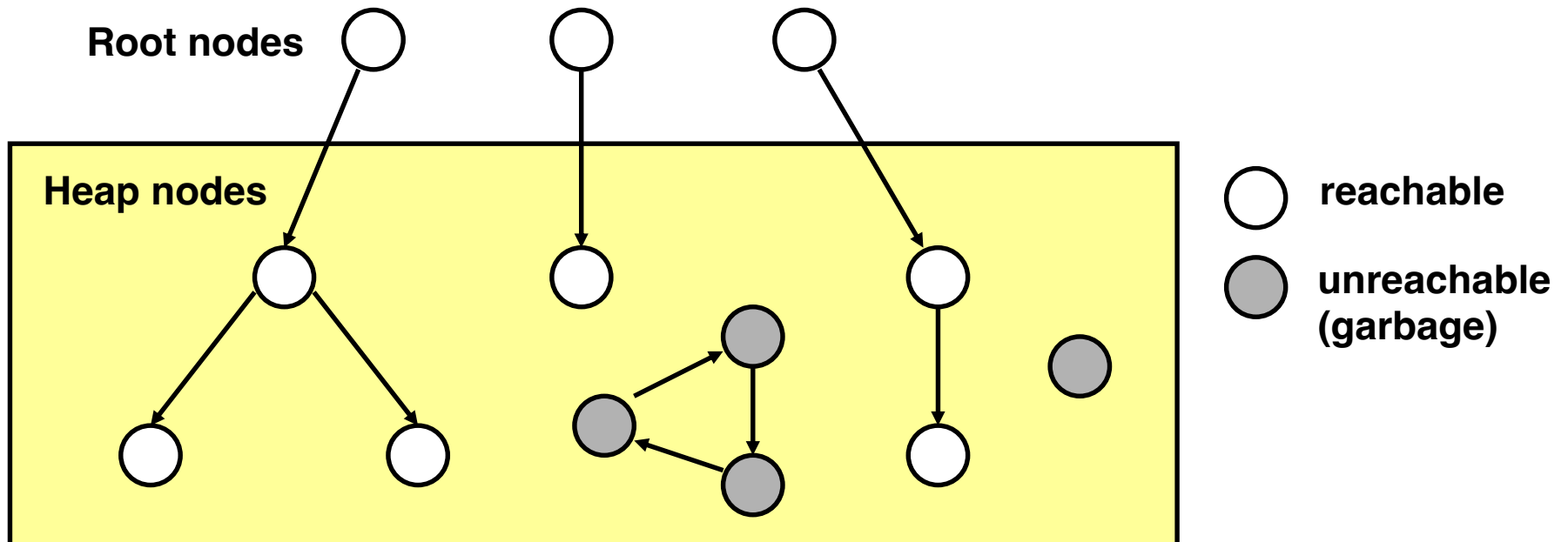
- Programmer has full control over memory
 - . . . with the responsibility to manage it well
 - Premature free's lead to **dangling references** (referencing deleted data)
 - Overly conservative free's lead to **memory leaks** (failing ever to delete data that cannot be referenced)
 - With manual free's it is virtually impossible to ensure that a program is correct and secure.
 - Even with manual memory management, the system maintains **bookkeeping** data and does **non-trivial 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 Java, Perl, ML, Modula-3, Prolog, and 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 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 makes the original problem running much 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)
 - For more information, see Jones and Lin, “Garbage Collection: Algorithms for Automatic Dynamic Memory”, John Wiley & Sons, 1996.
-

Reference Counting

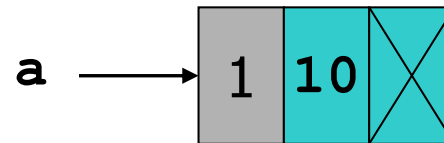
- Reference counting is a conservative technique for detecting garbage.
 - Each object has a **reference count**: the #references made to it. (in-degree of the node in object graph). When the reference count of an object falls to 0, then the object is garbage.
 - When an object is allocated, we initialize its reference count to **0**.
 - Increment reference counts
 - Assignment
 - Parameter Passing (more like explicit assignment)
 - Decrement reference counts
 - New Assignment ($p = q \rightarrow p = r$)
 - Procedure exits. All objects 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 = ...
```

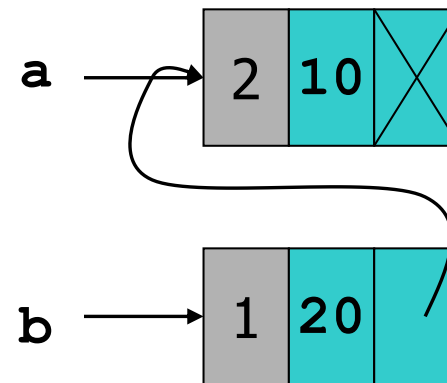
Reference Counting: Example

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



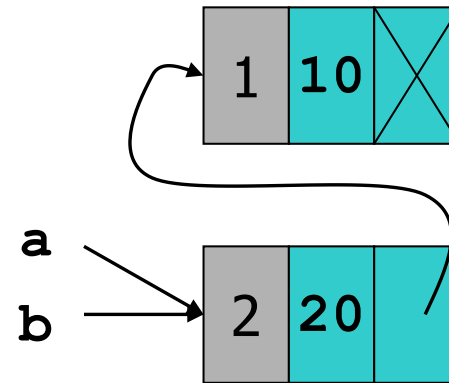
Reference Counting: Example

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



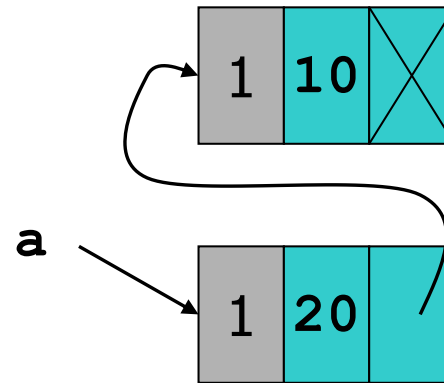
Reference Counting: Example

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



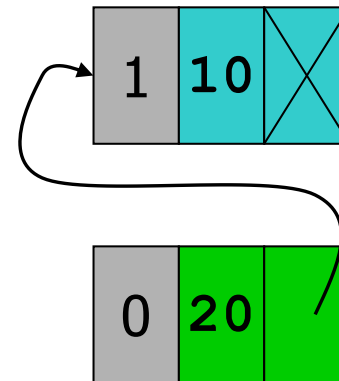
Reference Counting: Example

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



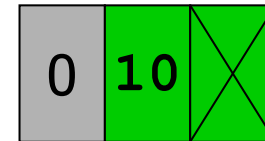
Reference Counting: Example

```
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 = ...
```

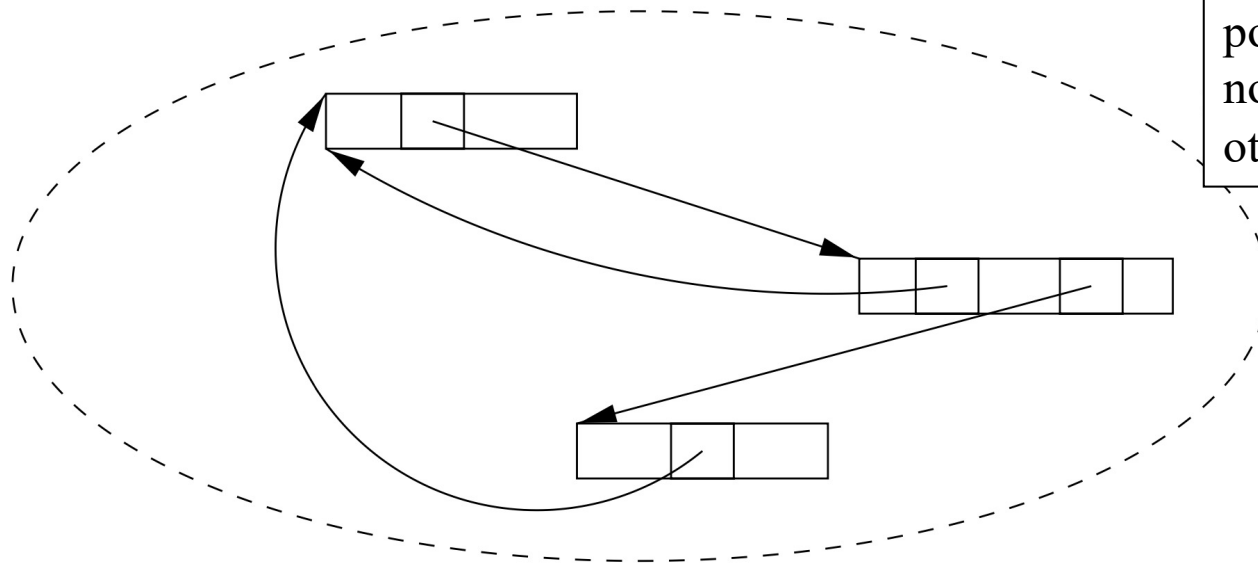


Reference Counting: Example

```
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 is tantamount to 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 soon as it is produced.
- Simple. Needs no elaborate system support. (e.g., used in OS Kernel data structures).

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 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

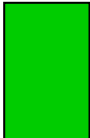

Two-phase collector

- 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.

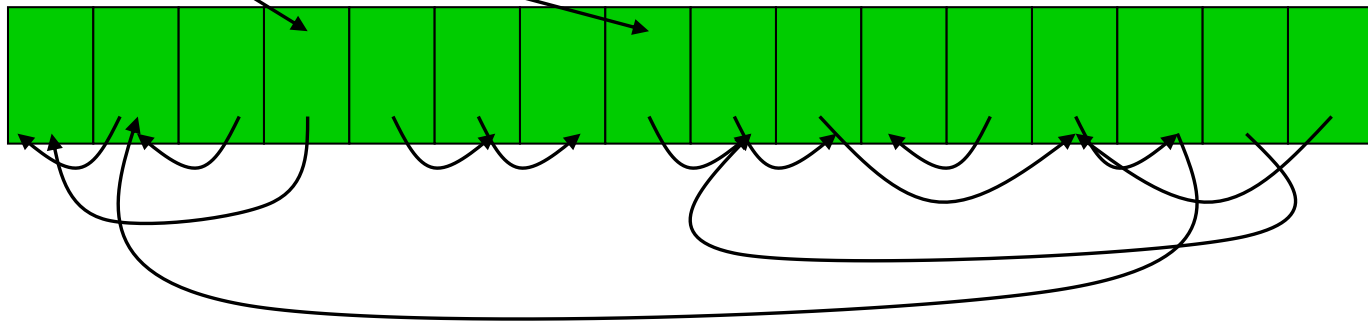
Mark & Sweep: GC Example

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

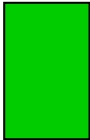

Unmarked=  Marked= 

Root pointers:

Heap:

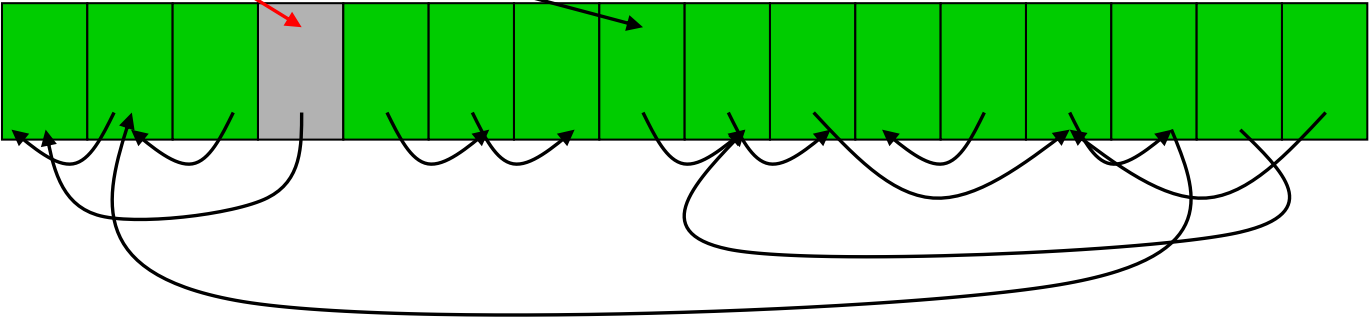


Mark & Sweep: GC Example

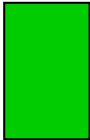

Unmarked=  Marked= 

Root pointers:

Heap:

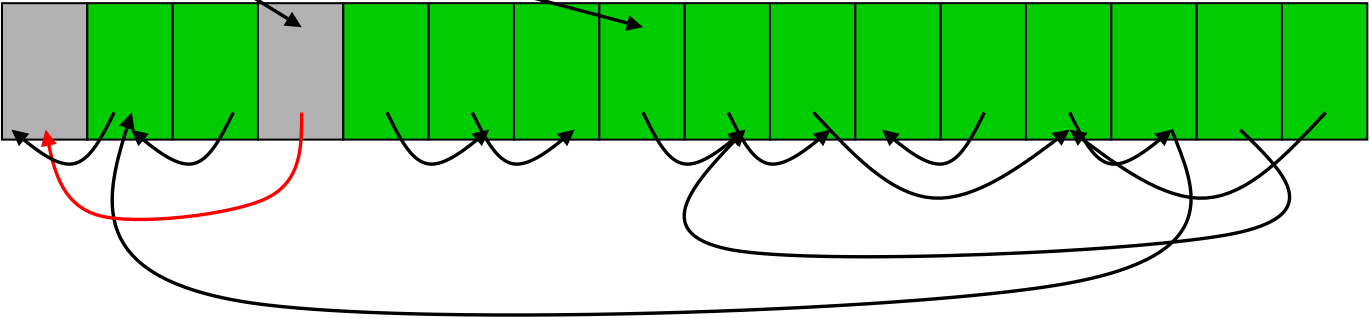


Mark & Sweep: GC Example

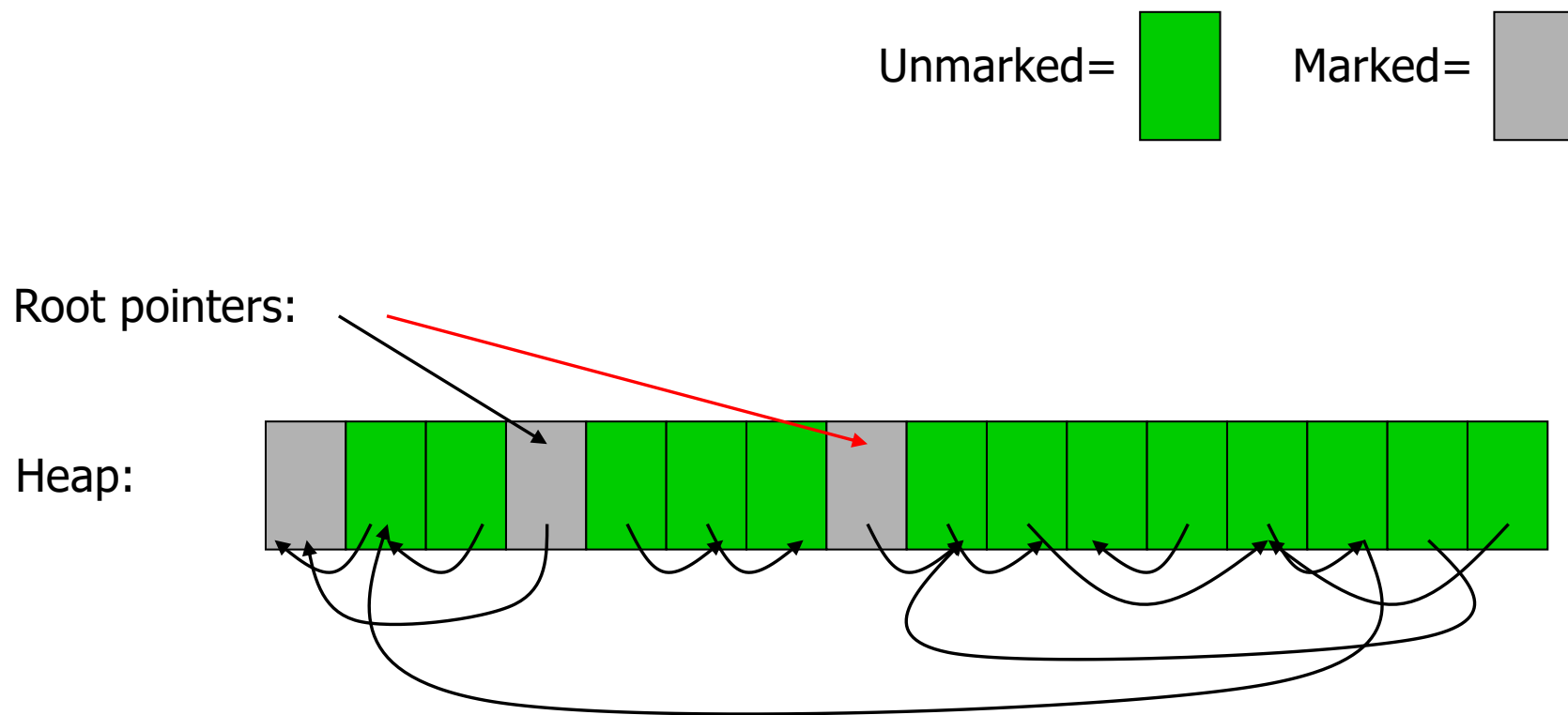
Unmarked=  Marked= 

Root pointers:

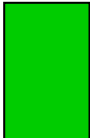

Heap:



Mark & Sweep: GC Example

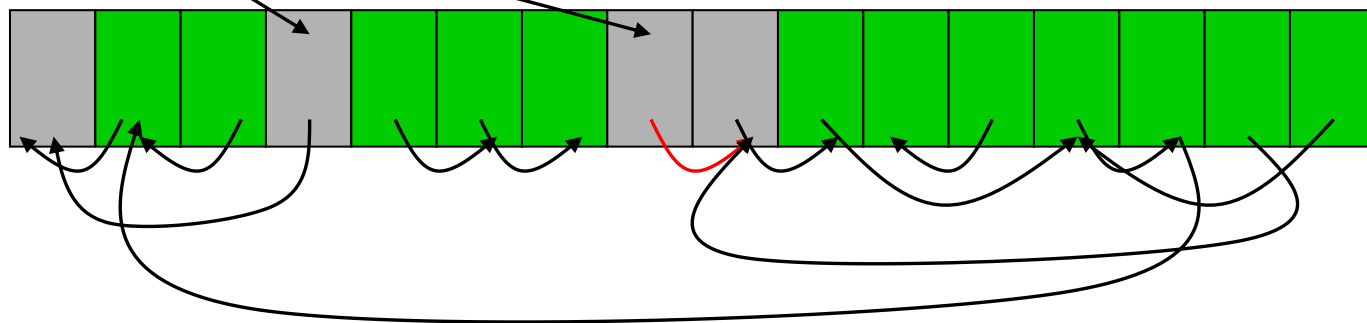


Mark & Sweep: GC Example

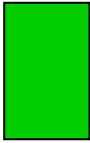

Unmarked=  Marked= 

Root pointers:

Heap:

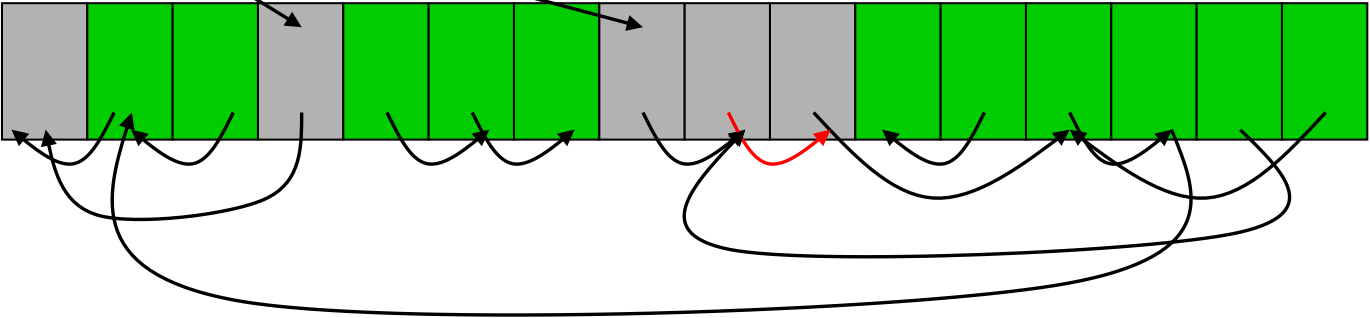


Mark & Sweep: GC Example

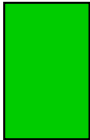

Unmarked=  Marked= 

Root pointers:

Heap:

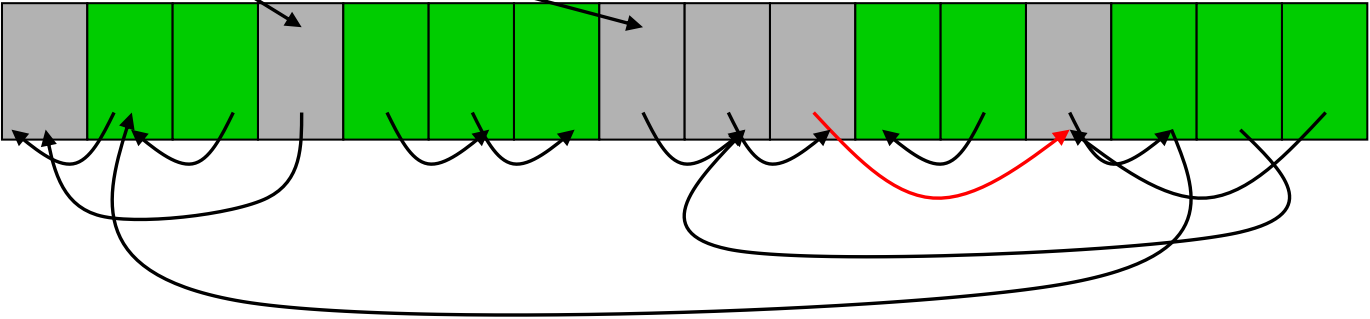


Mark & Sweep: GC Example

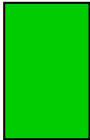

Unmarked=  Marked= 

Root pointers:

Heap:

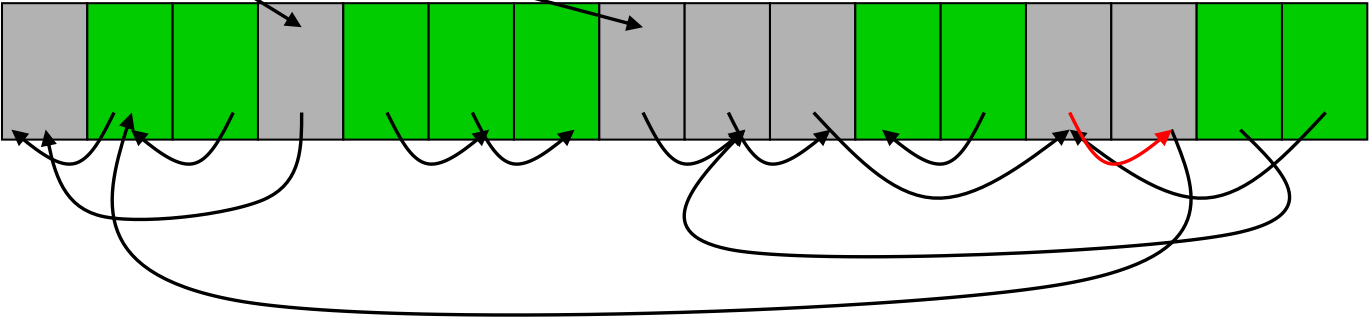


Mark & Sweep: GC Example

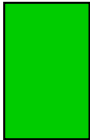

Unmarked=  Marked= 

Root pointers:

Heap:

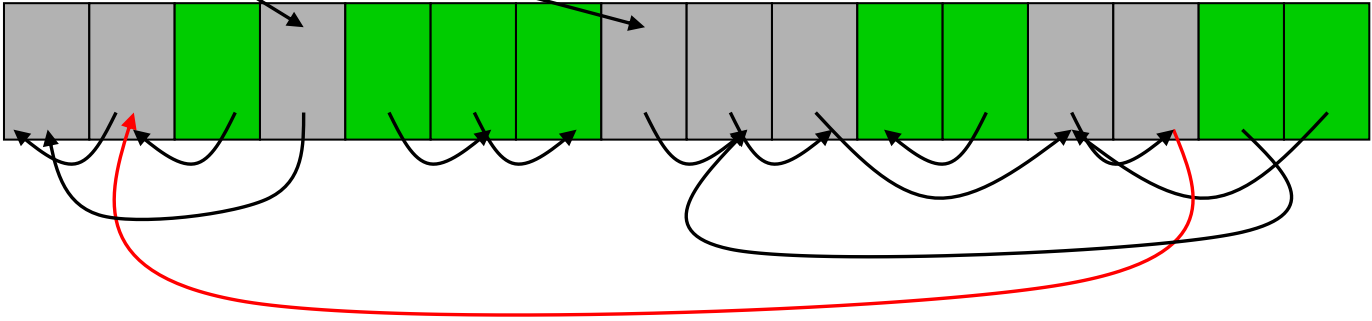


Mark & Sweep: GC Example

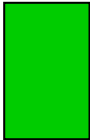

Unmarked=  Marked= 

Root pointers:

Heap:

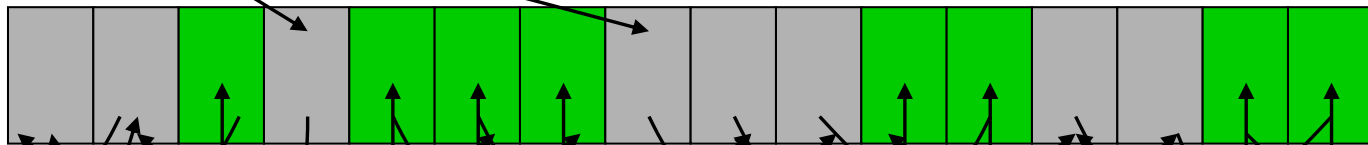


Mark & Sweep: GC Example

Unmarked=  Marked= 

Root pointers:

Heap:



Free list:

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

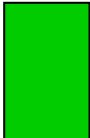

Stop & Copy Garbage Collector

Two-Space Collector (Cheney's Algorithm)

- 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
-

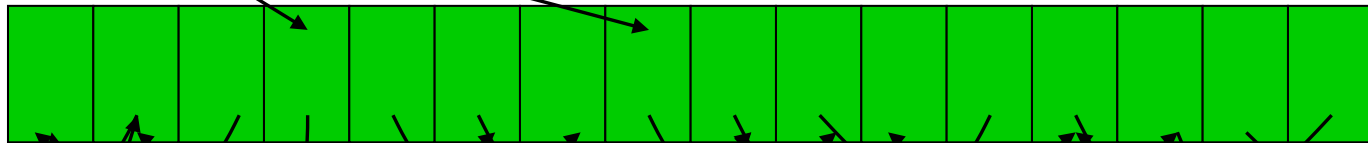
Stop & Copy: GC Example

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

Uncopied=  Copied= 

Root pointers:

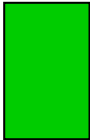

From:



To:

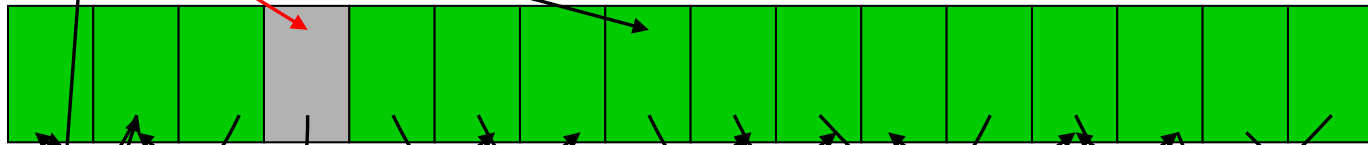


Stop & Copy: GC Example

Uncopied=  Copied= 

Root pointers:

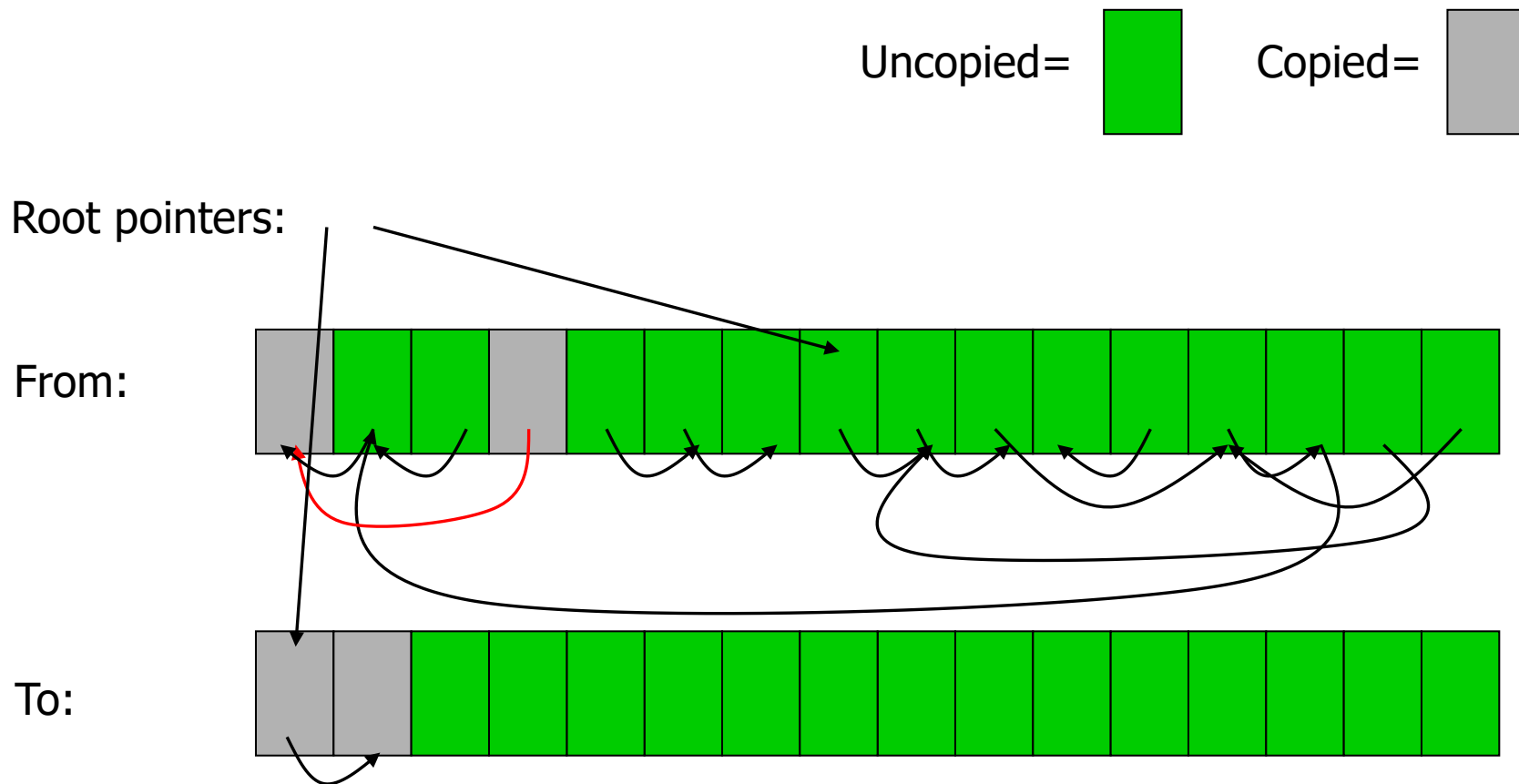
From:



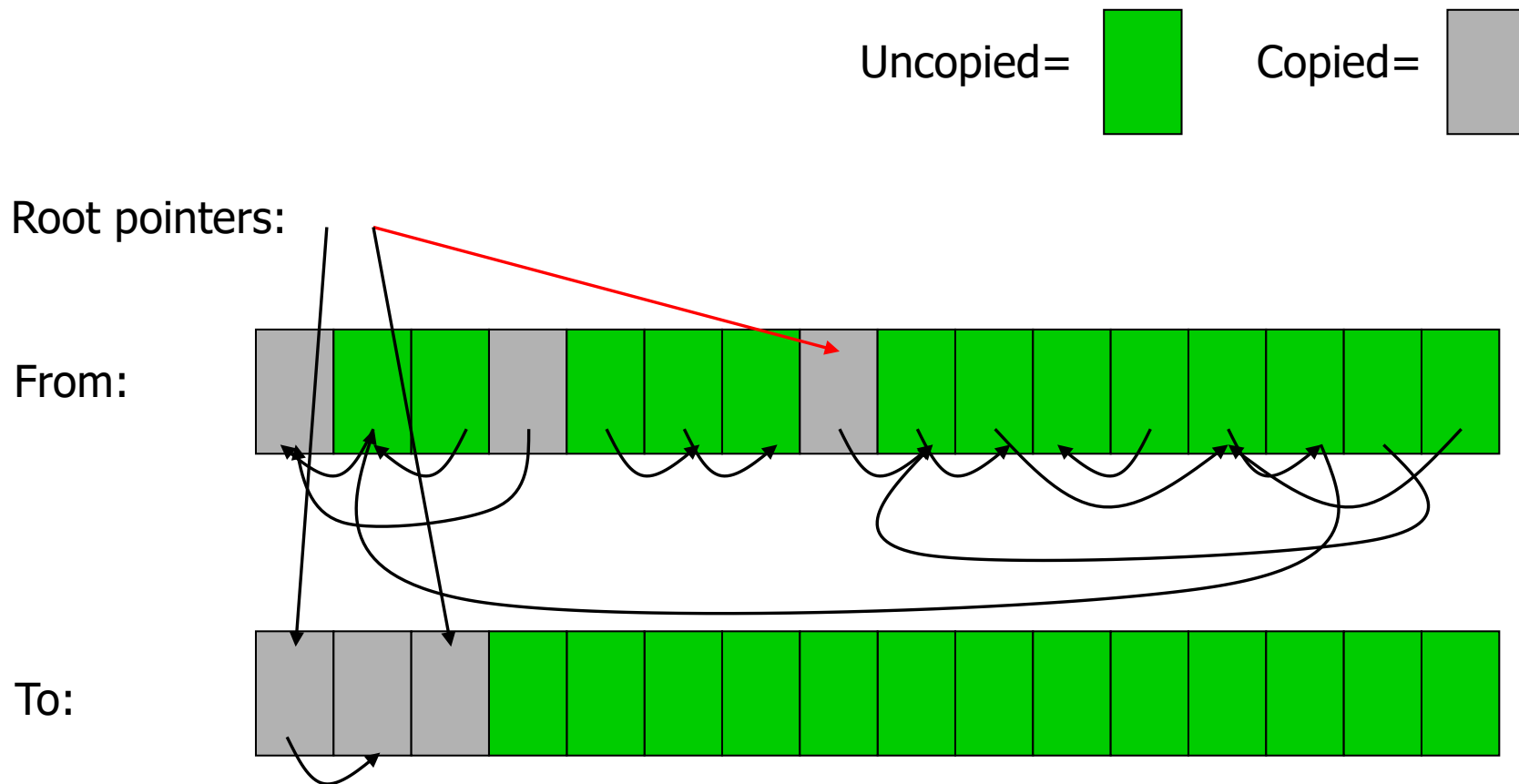
To:



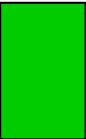

Stop & Copy: GC Example



Stop & Copy: GC Example

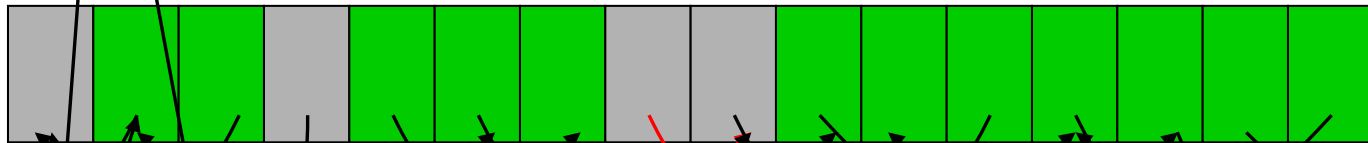


Stop & Copy: GC Example

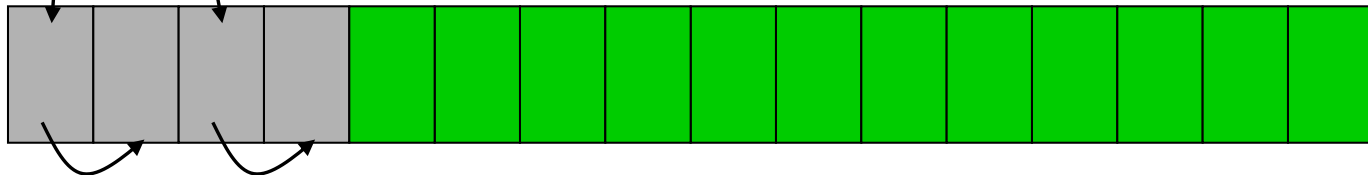
Uncopied=  Copied= 

Root pointers:

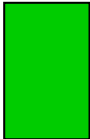

From:



To:

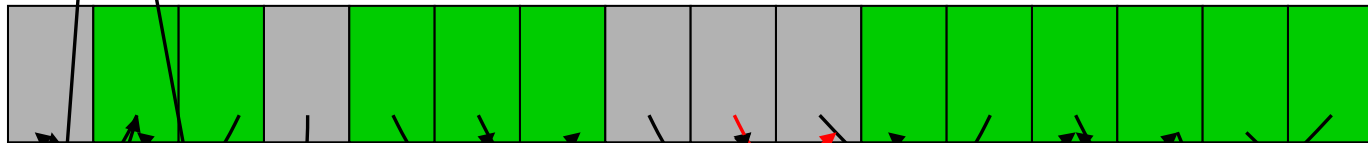


Stop & Copy: GC Example

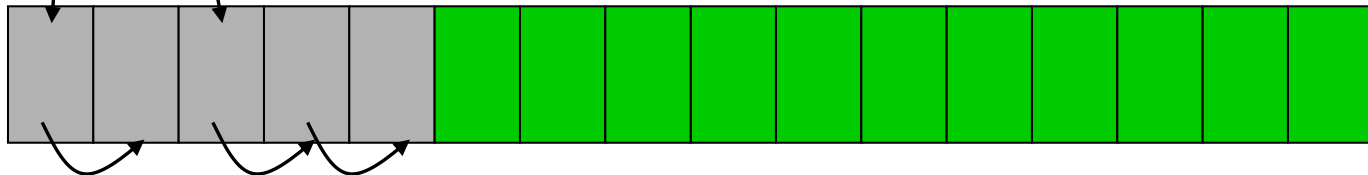
Uncopied=  Copied= 

Root pointers:

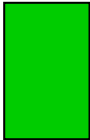

From:



To:

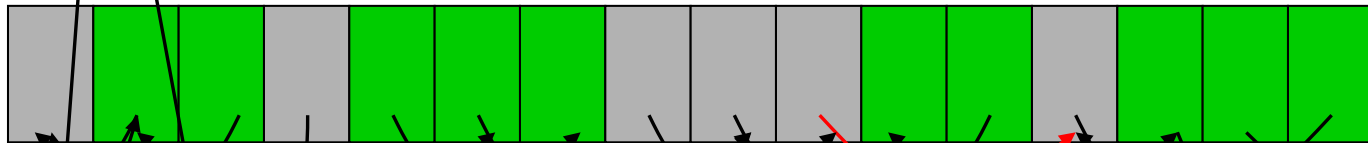


Stop & Copy: GC Example

Uncopied=  Copied= 

Root pointers:

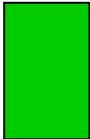

From:



To:

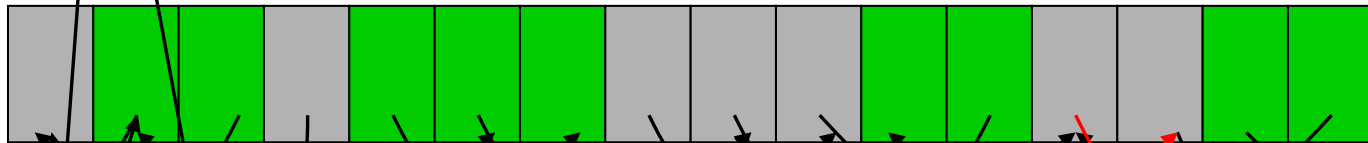


Stop & Copy: GC Example

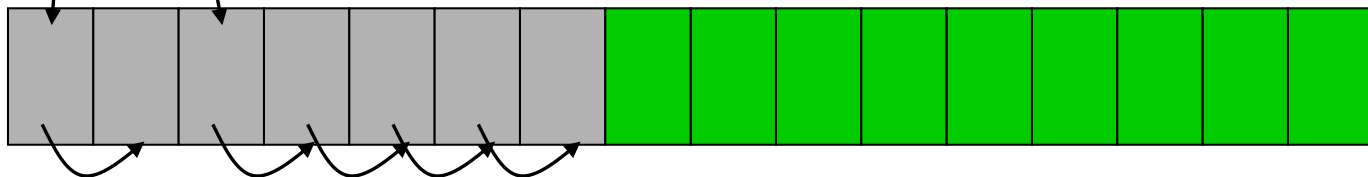
Uncopied=  Copied= 

Root pointers:

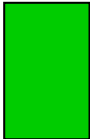

From:



To:

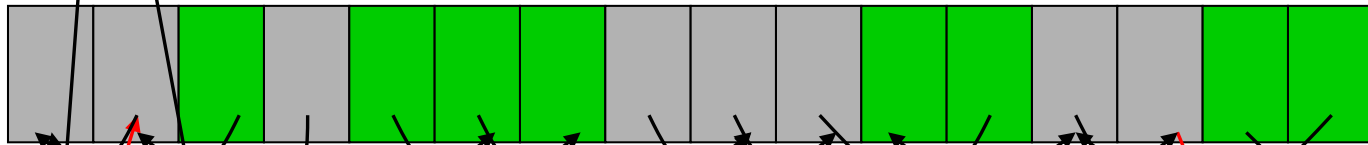


Stop & Copy: GC Example

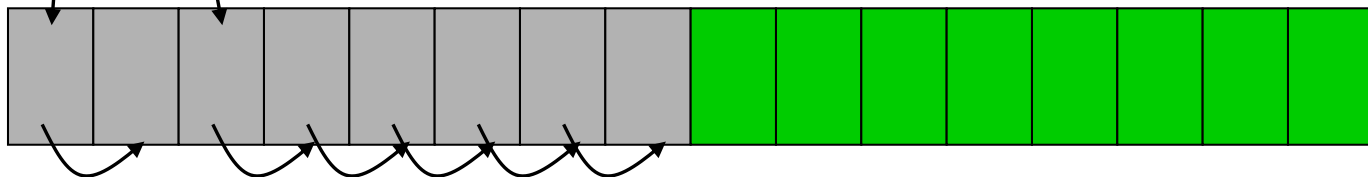
Uncopied=  Copied= 

Root pointers:

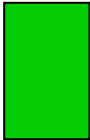

From:



To:

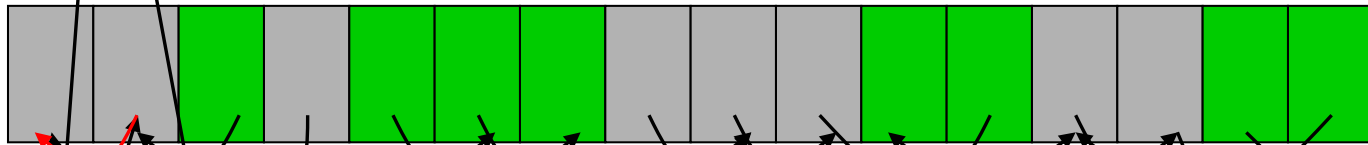


Stop & Copy: GC Example

Uncopied=  Copied= 

Root pointers:

From:



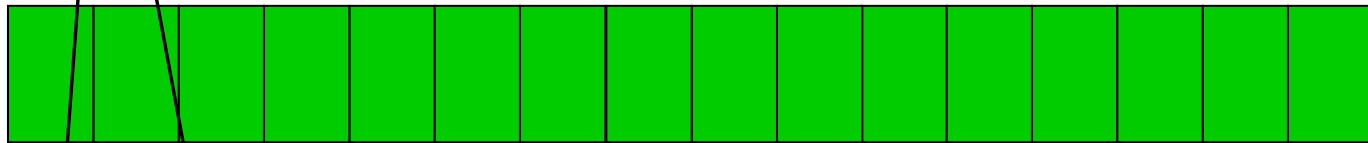
To:



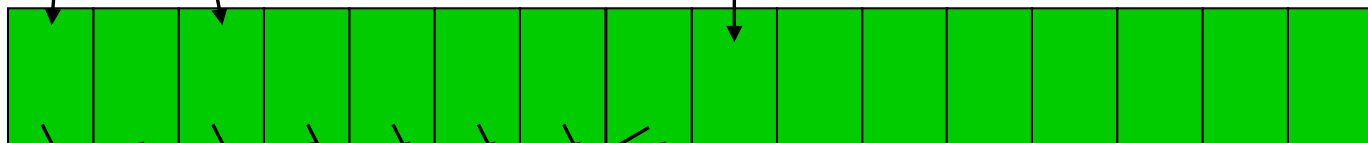
Stop & Copy: GC Example

Root pointers:

To:



From:



Next block to allocate

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
 - Double the basic heap size
-

Other GC Variations

- *Many* variations on three main themes
 - **Concurrent GC**: Garbage collector runs concurrently (e.g., in a separate thread) with the program; the program is not interrupted for collection
 - **Generational GC**: Objects are divided into old and new generations, and new generations are collected more often. It exploits the observation that most objects have a short lifetime.
 - **Conservative GC**: It conservatively assumes that integers and other data can be cast to pointers
 - Combinations of these three main themes are common
 - Java uses both Copying and Mark-and-Sweep within a Generational GC
-