

CMSC 330: Organization of Programming Languages

Memory Management and Garbage Collection

Memory Attributes

- ▶ Memory to store data in programming languages has the following lifecycle
 - Allocation
 - When the memory is allocated to the program
 - Lifetime
 - How long allocated memory is used by the program
 - Recovery
 - When the system recovers the memory for reuse
- ▶ The **allocator** is the system feature that performs allocation and recovery

Memory Attributes (cont.)

- ▶ Most programming languages are concerned with some subset of the following 4 memory classes
 1. Static (or fixed) memory
 2. Automatic (LIFO) memory
 3. Dynamically allocated memory
 4. Persistent memory

Memory Classes

- ▶ Static memory – Usually at a fixed address
 - Lifetime – The execution of program
 - Allocation – For entire execution
 - Recovery – By system when program terminates
 - Allocator – Compiler
- ▶ Automatic (LIFO) memory – Usually on a stack
 - Lifetime – Activation of method using that data
 - Allocation – When method is invoked
 - Recovery – When method terminates
 - Allocator – Typically compiler, sometimes programmer

Memory Classes (cont.)

- ▶ Dynamic memory – Addresses allocated on demand in an area called the **heap**
 - Lifetime – As long as memory is needed
 - Allocation – Explicitly by programmer, or implicitly by compiler
 - Recovery – Either by programmer or automatically (when possible and depends upon language)
 - Allocator – Manages free/available space in heap

Memory Management in C

- ▶ Local variables live on the stack
 - Allocated at function invocation time
 - Deallocated when function returns
 - Storage space reused after function returns
- ▶ Space on the heap allocated with `malloc()`
 - Must be explicitly freed with `free()`
 - Called **explicit** or **manual** memory management
 - Deletions must be done by the user

Memory Management Errors

- ▶ May forget to free memory (**memory leak**)

```
{ int *x = (int *) malloc(sizeof(int)); }
```

- ▶ May retain ptr to freed memory (**dangling pointer**)

```
{ int *x = ...malloc();  
  free(x);  
  *x = 5; /* oops! */  
}
```

```
/* also: */  
int* foo(){  
    int x;  
    return &x; /*bad!*/  
}
```

- ▶ May free something twice

```
{ int *x = ...malloc(); free(x); free(x); }
```

- This may corrupt the memory management data structures
 - E.g., the memory allocator maintains a **free list** of space on the heap that's available

Ways to Avoid Mistakes in C

- ▶ Don't allocate memory on the heap
 - Could lead to confusing code
- ▶ Never free memory
 - OS will reclaim process's memory anyway at exit
 - Memory is cheap; who cares about a little leak?
- ▶ But: Both of the above two may be impractical
- ▶ Can avoid all three problems by using **automatic memory management**
 - Key technology to ensure type safety
 - Does not prevent all leaks, as we will see

Fragmentation

- ▶ Another memory management problem
- ▶ Example sequence of calls

allocate(a);

allocate(x);

allocate(y);

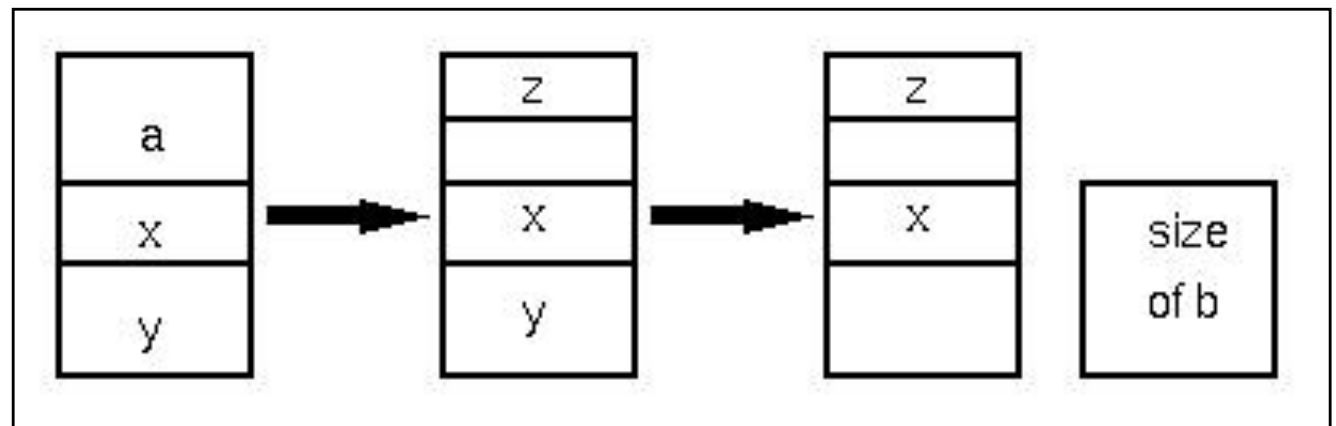
free(a);

allocate(z);

free(y);

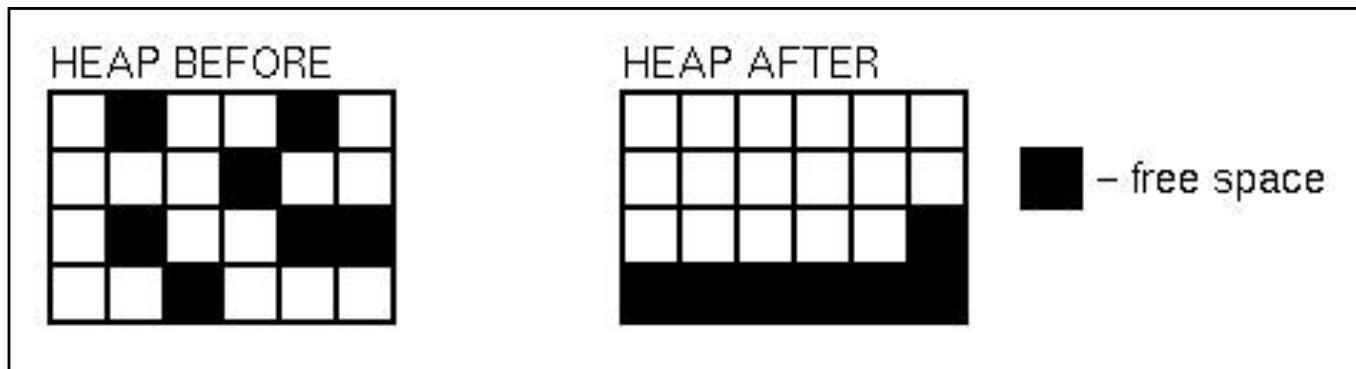
allocate(b);

⇒ Not enough contiguous space for b



Automatic memory management

- ▶ Primary goal: automatically reclaim dynamic memory
 - Secondary goal: also avoid fragmentation



- ▶ **Insight:** You can do reclamation **and** avoid fragmentation if you can identify every pointer in a program
 - You can move the allocated storage, then redirect pointers to it
 - Compact it, to avoid fragmentation
 - Compiler ensures perfect knowledge LISP, OCAML, Java, Prolog but not in C, C++, Pascal, Ada

Strategy

- ▶ At any point during execution, can divide the objects in the heap into two classes
 - Live objects will be used later
 - Dead objects will never be used again
 - They are “garbage”
- ▶ Thus we need garbage collection (GC) algorithms that can
 1. Distinguish live from dead objects
 2. Reclaim the dead objects and retain the live ones

Determining Liveness

- ▶ In most languages we can't know for sure which objects are really live or dead
 - Undecidable, like solving the halting problem
- ▶ Thus we need to make a **safe** approximation
 - OK if we decide something is live when it's not
 - But we'd better not deallocate an object that will be used later on

Liveness by Reachability

- ▶ An object is **reachable** if it can be accessed by dereferencing (“chasing”) pointers from live data
- ▶ Safe policy: delete **unreachable** objects
 - An unreachable object can never be accessed again by the program
 - The object is definitely garbage
 - A reachable object may be accessed in the future
 - The object could be garbage but will be retained anyway
 - Could lead to memory leaks

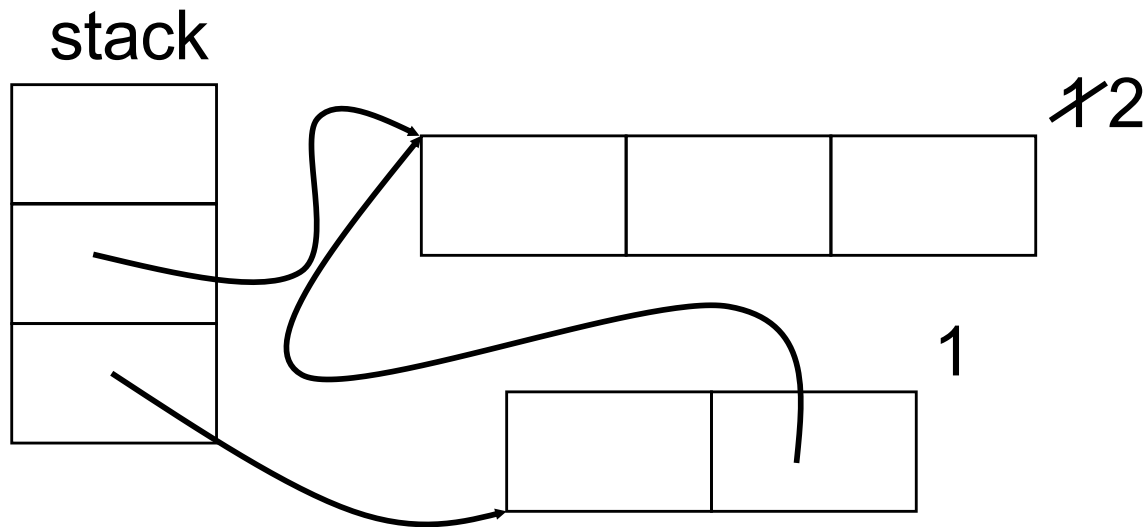
Roots

- ▶ At a given program point, we define **liveness** as being data reachable from the **root set**
 - Global variables
 - What are these in Java? Ruby? OCaml?
 - Local variables of all live method activations
 - I.e., the stack
- ▶ At the machine level
 - Also consider the register set
 - Usually stores local or global variables
- ▶ Next
 - Techniques for determining reachability

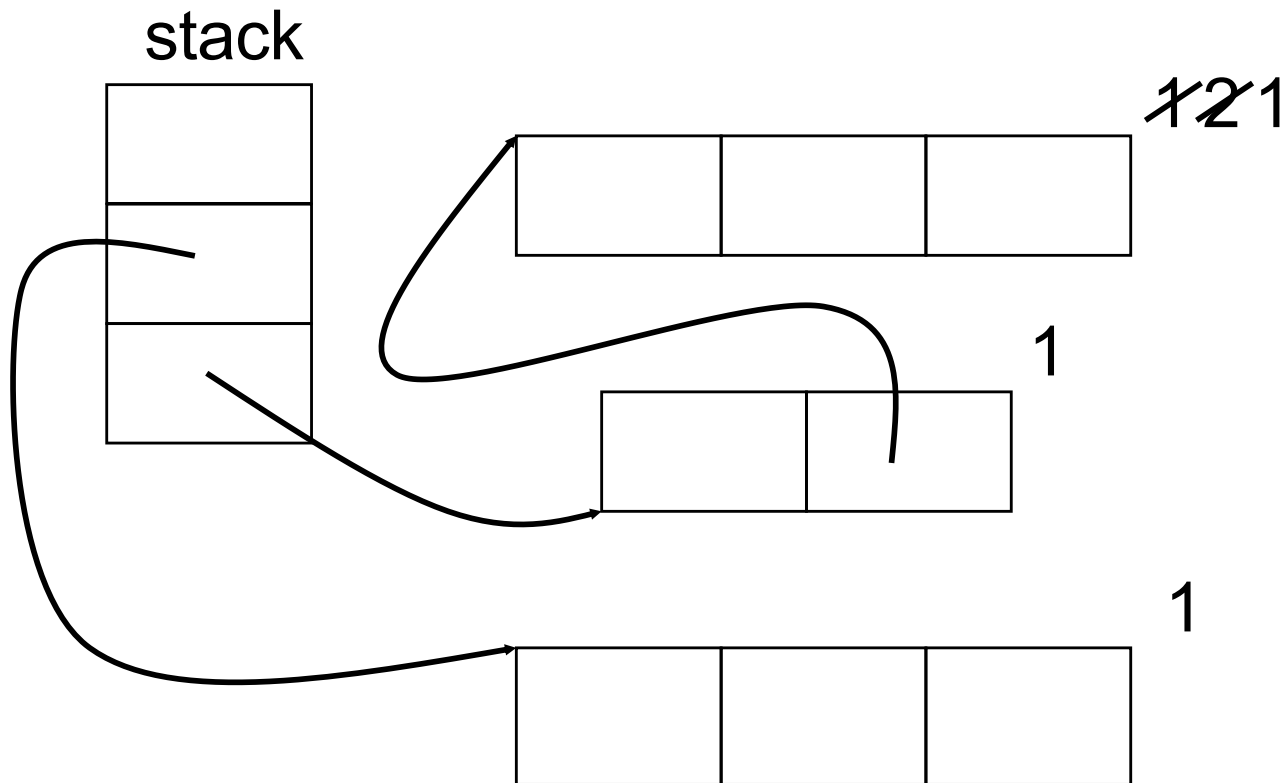
Reference Counting

- ▶ **Idea:** Each object has count of number of pointers to it from the roots or other objects
 - When count reaches 0, object is unreachable
- ▶ Count tracking code may be manual or automatic
- ▶ In regular use
 - C++ and **Rust** (smart pointers), Cocoa (manual), Python (automatic)
- ▶ Method doesn't address fragmentation problem
- ▶ Invented by Collins in 1960
 - A method for overlapping and erasure of lists.
Communications of the ACM, December 1960

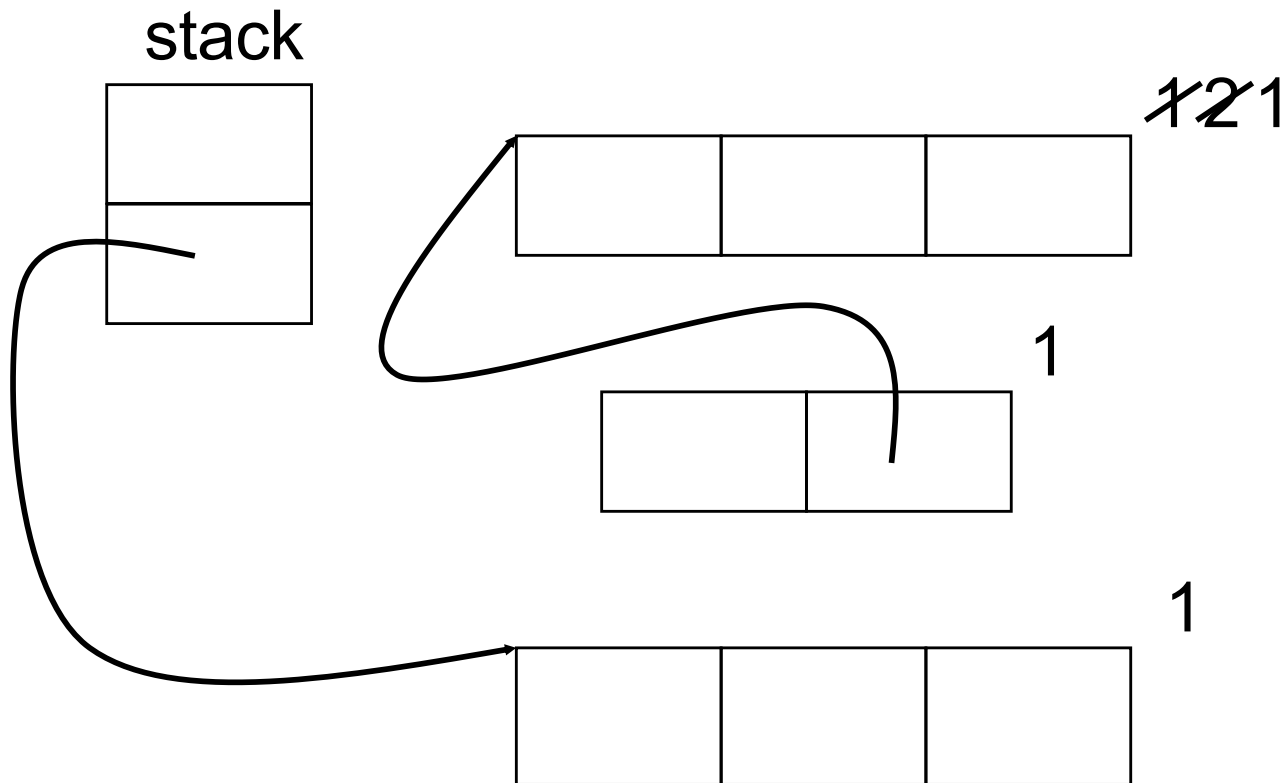
Reference Counting Example



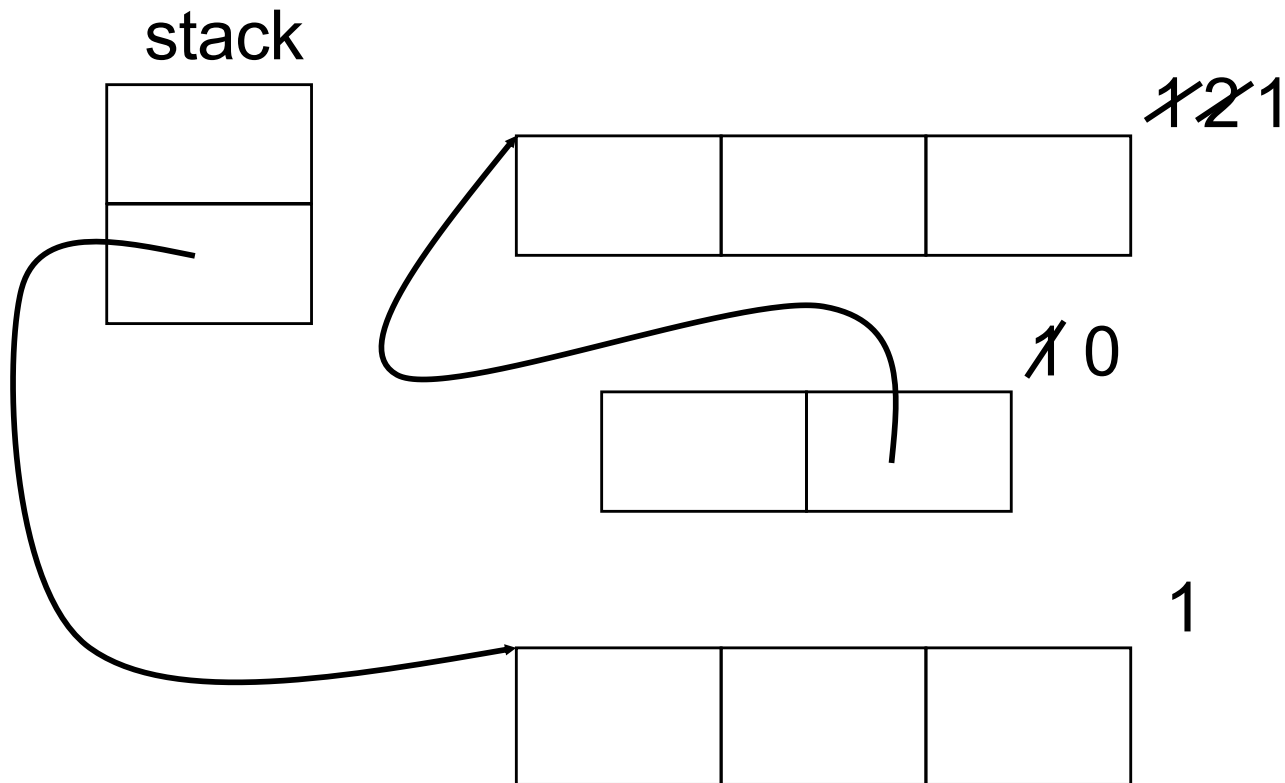
Reference Counting Example (cont.)



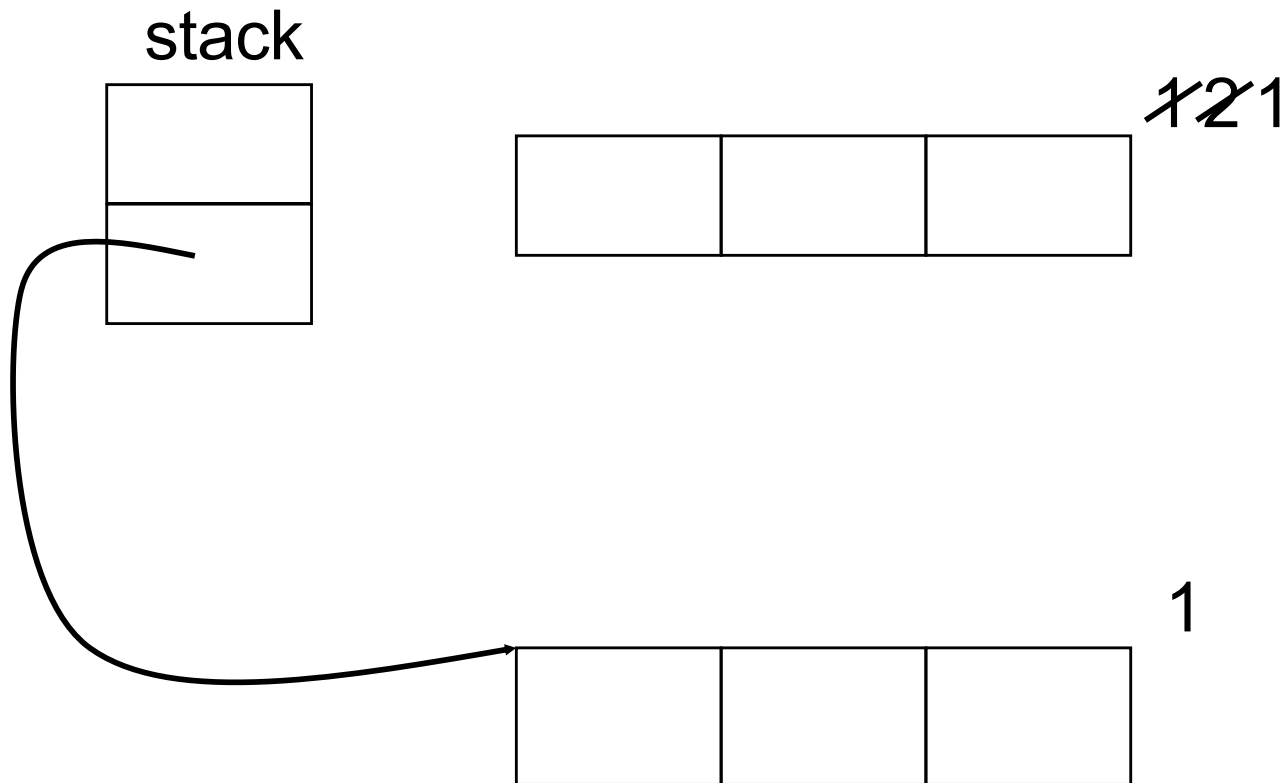
Reference Counting Example (cont.)



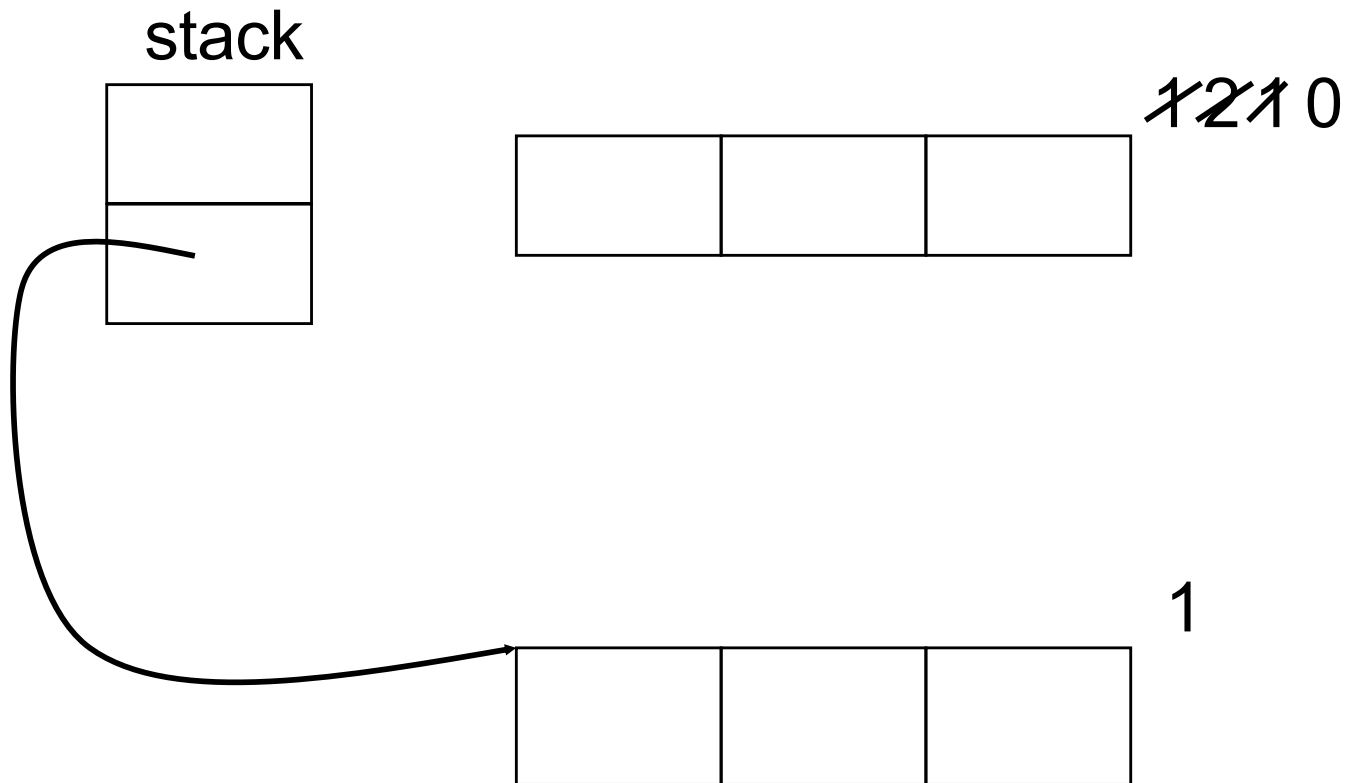
Reference Counting Example (cont.)



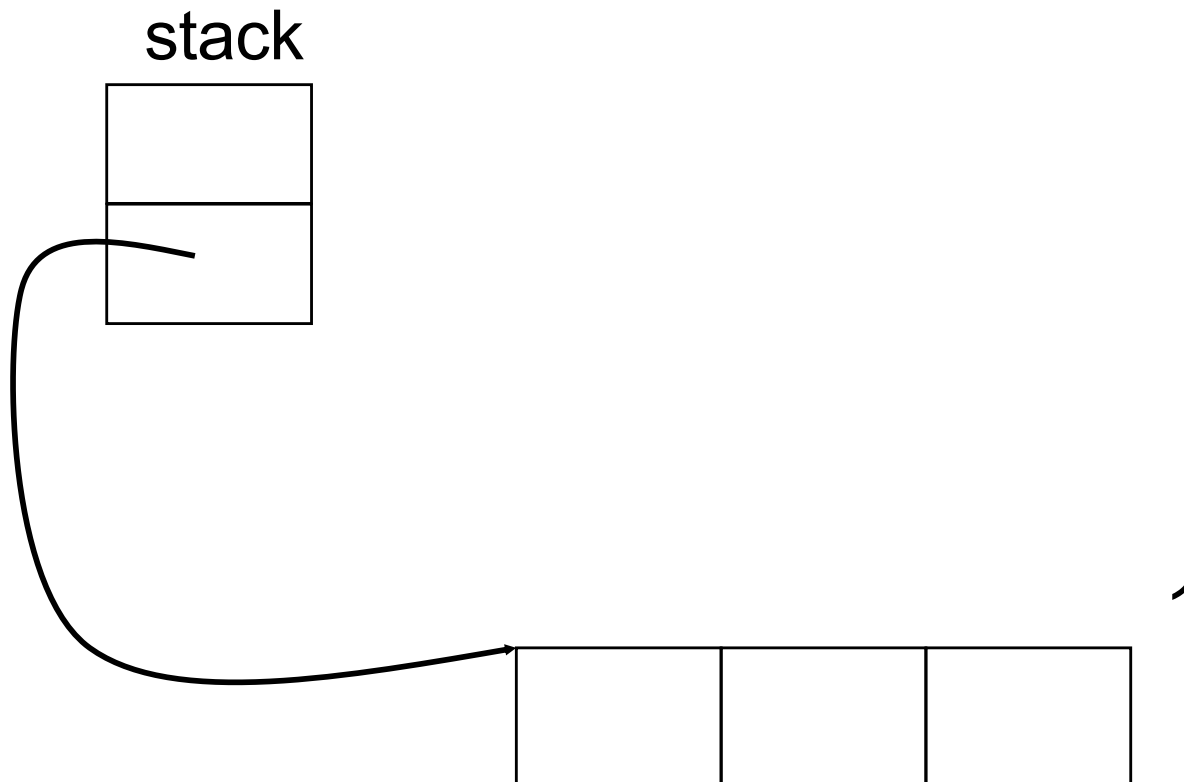
Reference Counting Example (cont.)



Reference Counting Example (cont.)



Reference Counting Example (cont.)



Rust Rc Example

```
use std::rc::Rc;
fn main() {
    let s = String::from("hello");
    let r1 = Rc::new(&s);
    {
        let r2 = Rc::clone(&r1);
        println!("r1 = {}", Rc::strong_count(&r1));
        println!("r2 = {}", Rc::strong_count(&r2));
    }
    // r2 is out of scope
    println!("r1 = {}", Rc::strong_count(&r1));
}
```

Output:

```
r1 = 2
r2 = 2
r1 = 1
```

Reference Counting Tradeoffs

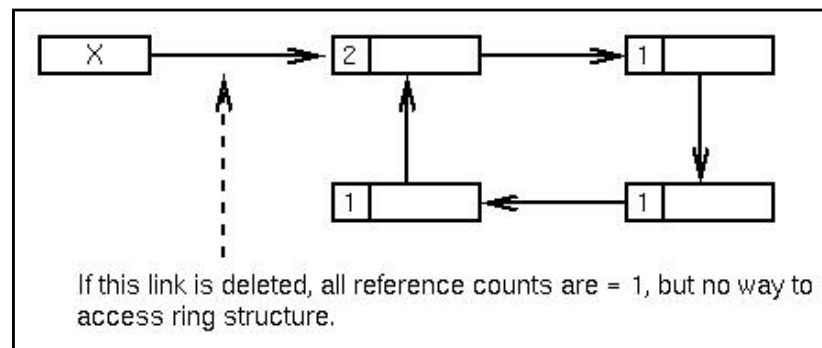
► Advantage

- Incremental technique

- Generally small, constant amount of work per memory write
- With more effort, can even bound running time

► Disadvantages

- Cascading decrements can be expensive
- Requires extra storage for reference counts
- Need other means to collect cycles, for which counts never go to 0



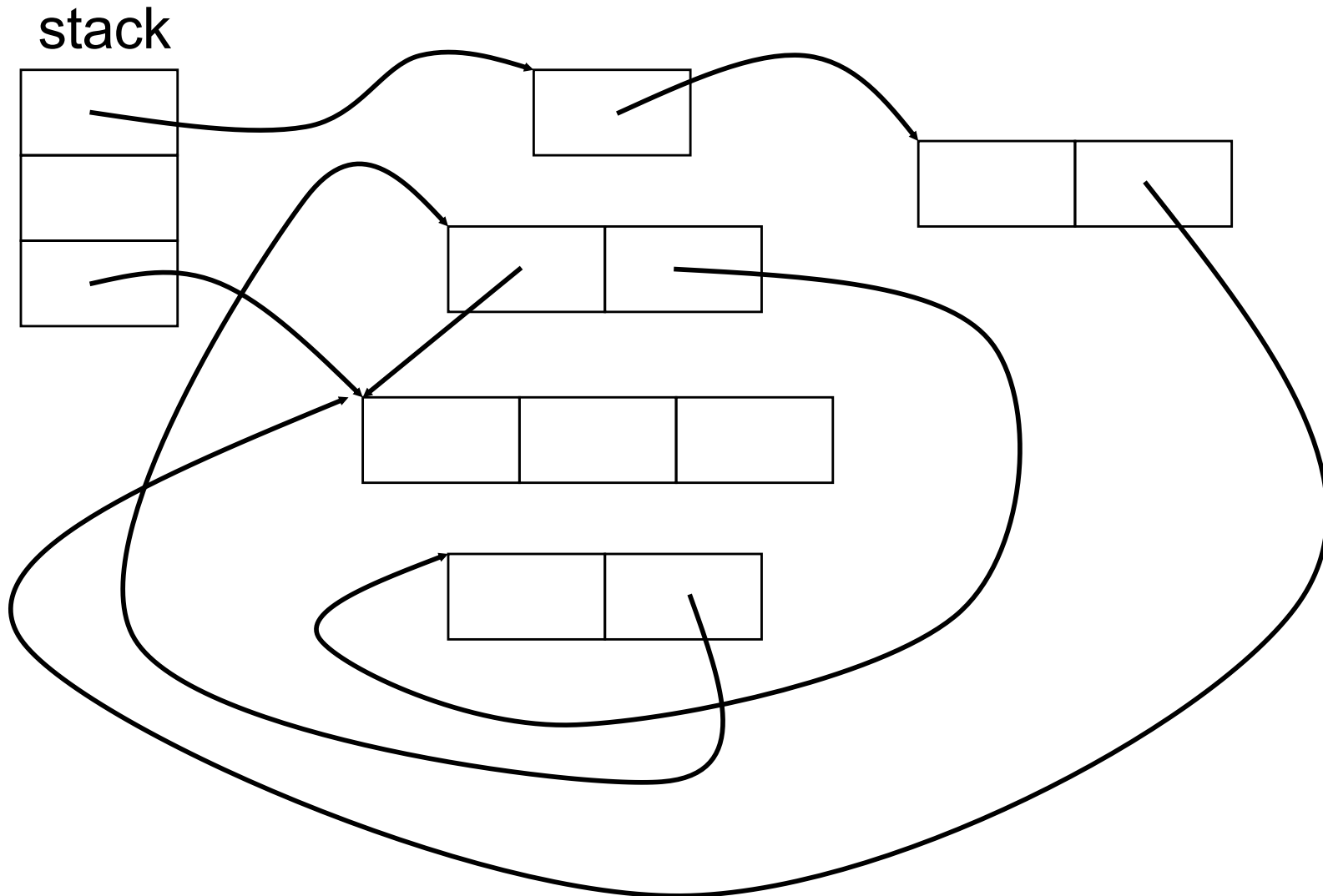
Tracing Garbage Collection

- ▶ **Idea:** Determine reachability as needed, rather than by stored counts, incrementally
- ▶ Every so often, stop the world and
 - Follow pointers from live objects (starting at roots) to expand the live object set
 - Repeat until no more reachable objects
 - Deallocate any non-reachable objects
- ▶ Two main variants of tracing GC
 - Mark/sweep (McCarthy 1960) and stop-and-copy (Cheney 1970)

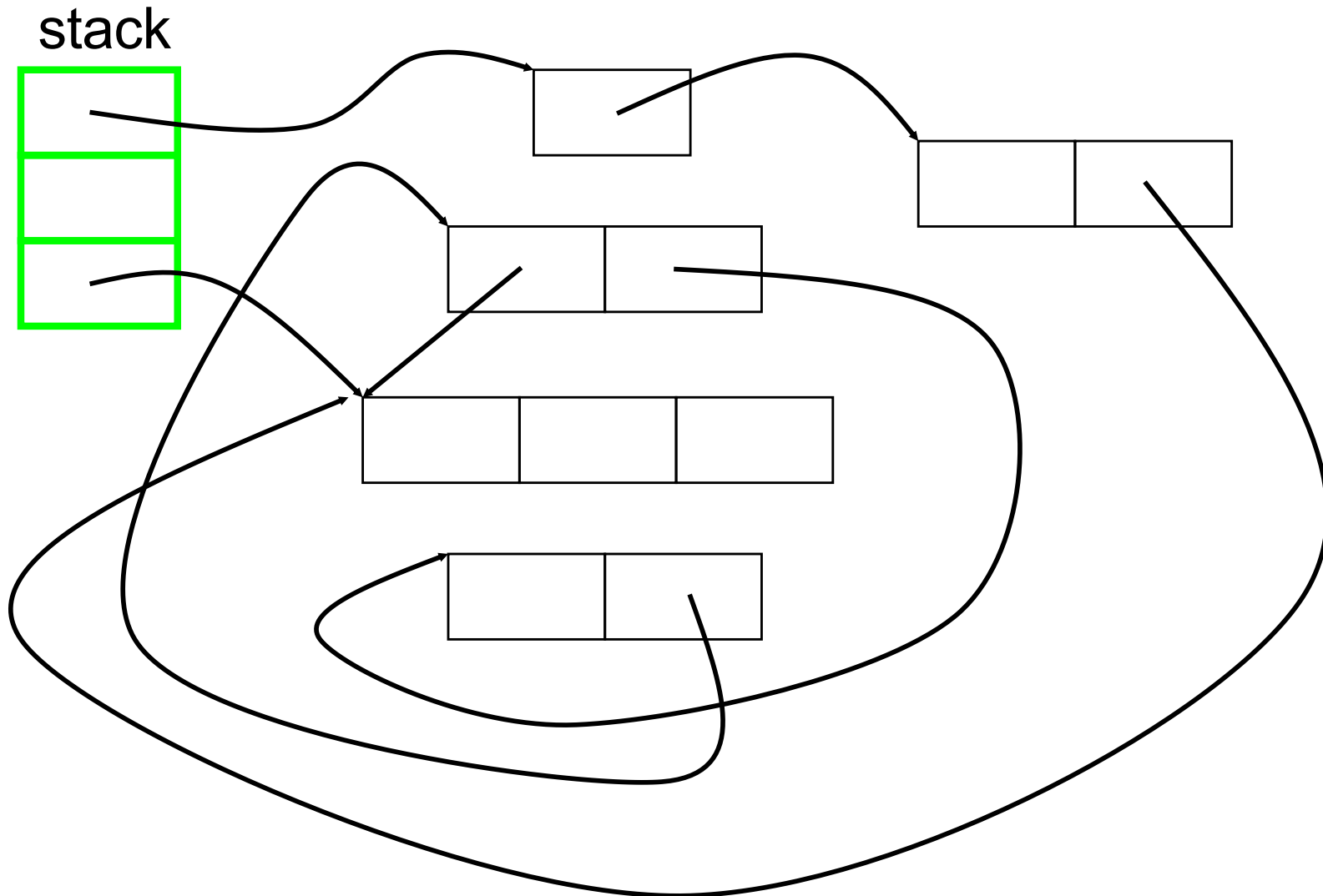
Mark and Sweep GC

- ▶ Two phases
 - Mark phase: trace the heap and mark all reachable objects
 - Sweep phase: go through the entire heap and reclaim all unmarked objects

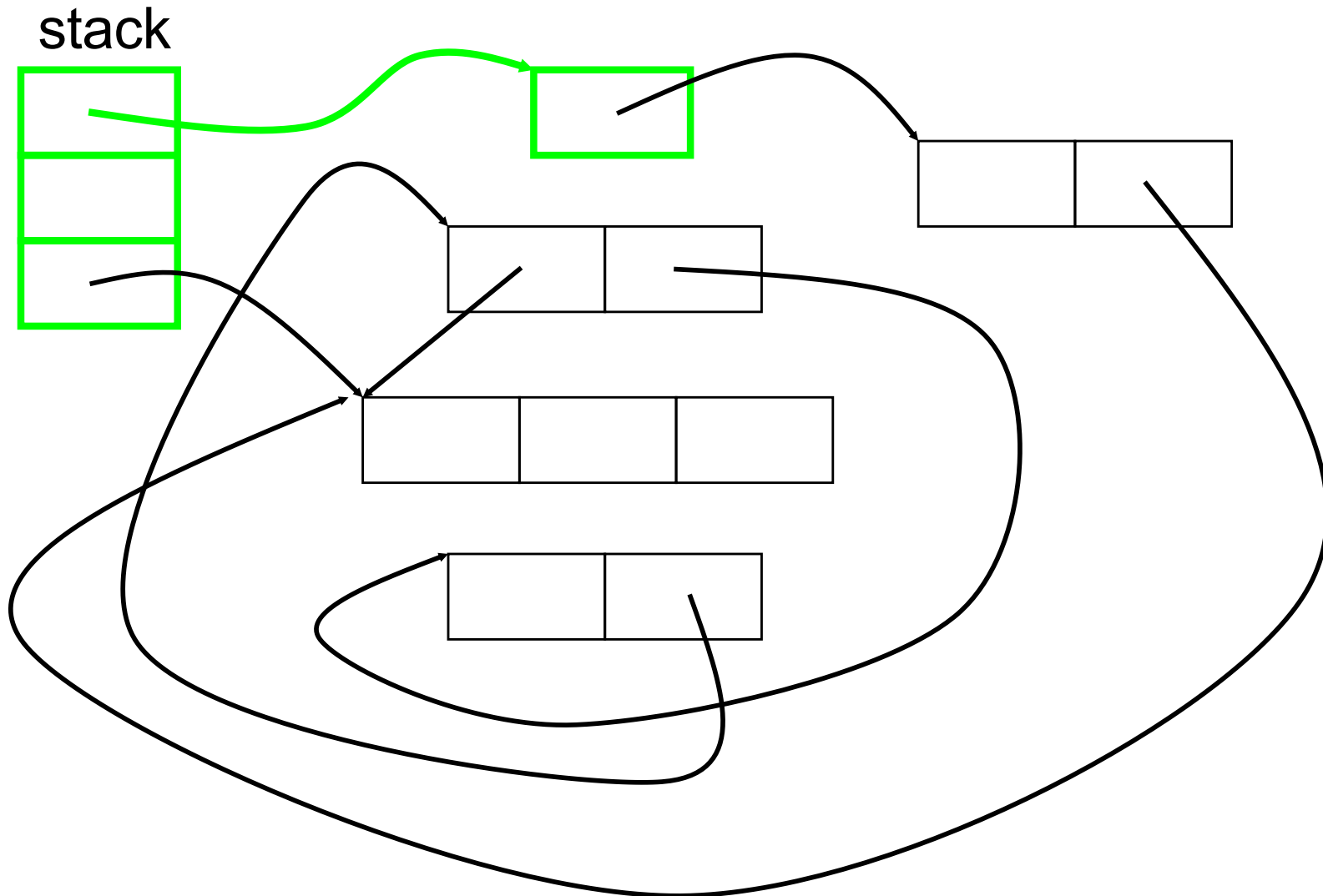
Mark and Sweep Example



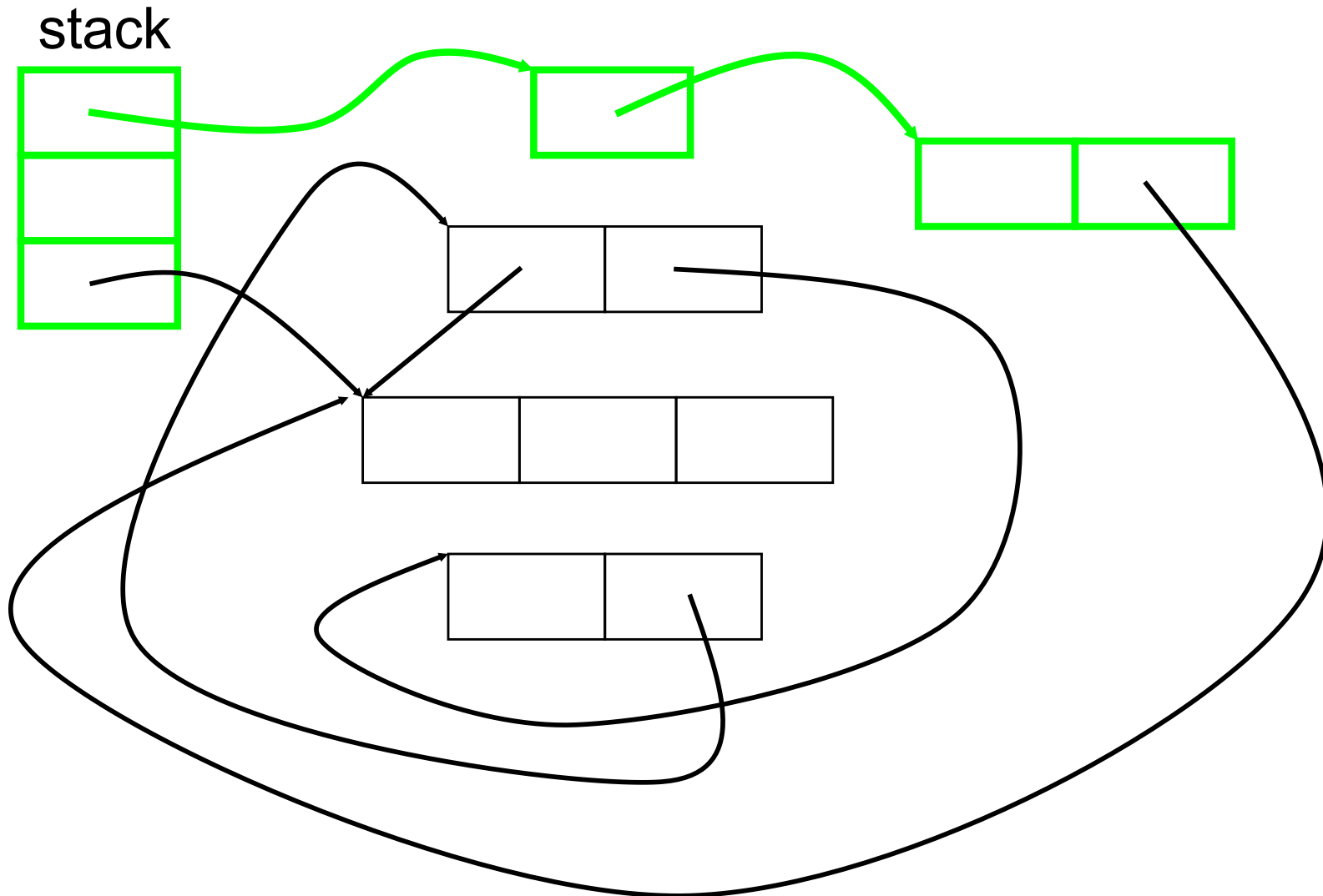
Mark and Sweep Example (cont.)



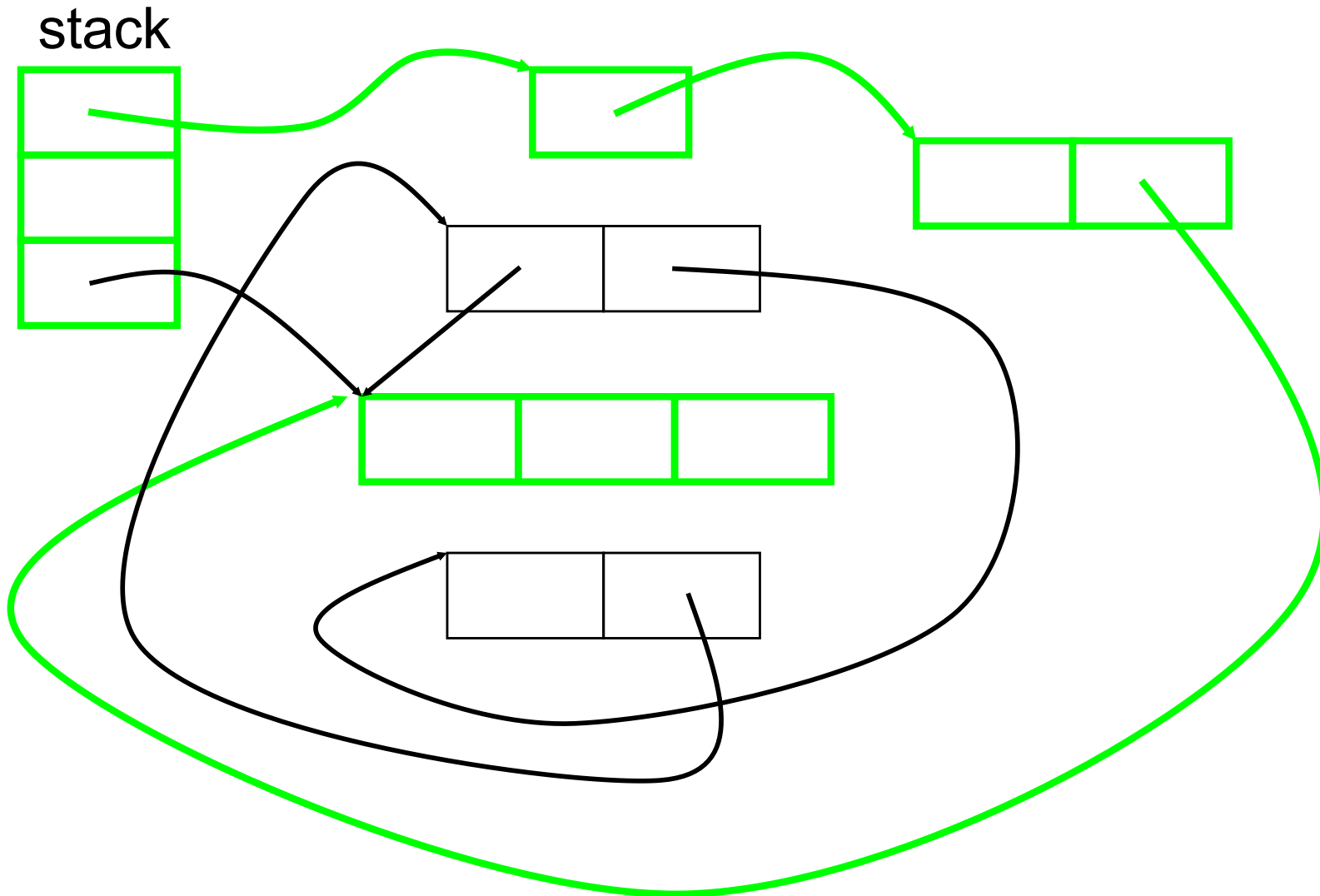
Mark and Sweep Example (cont.)



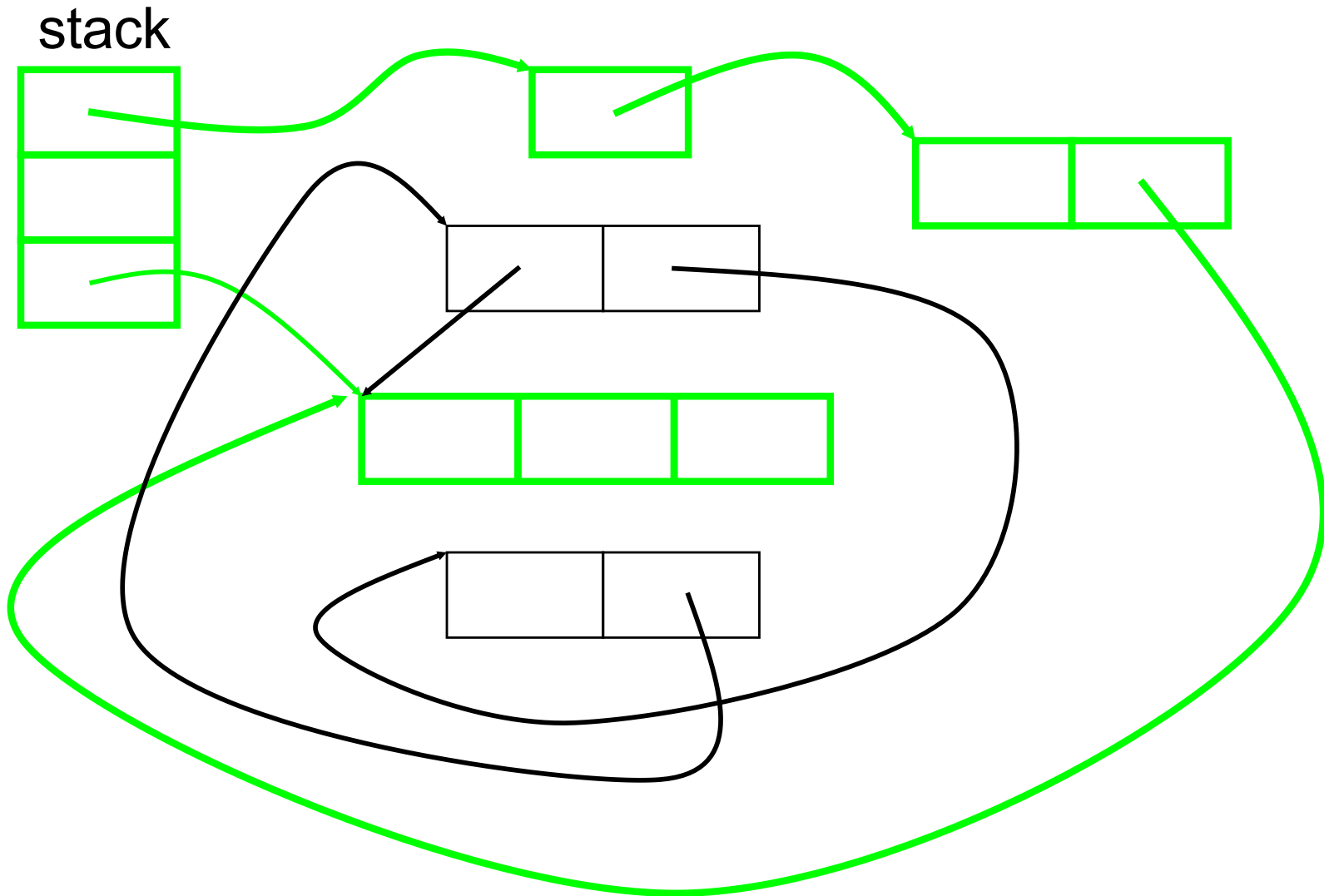
Mark and Sweep Example (cont.)



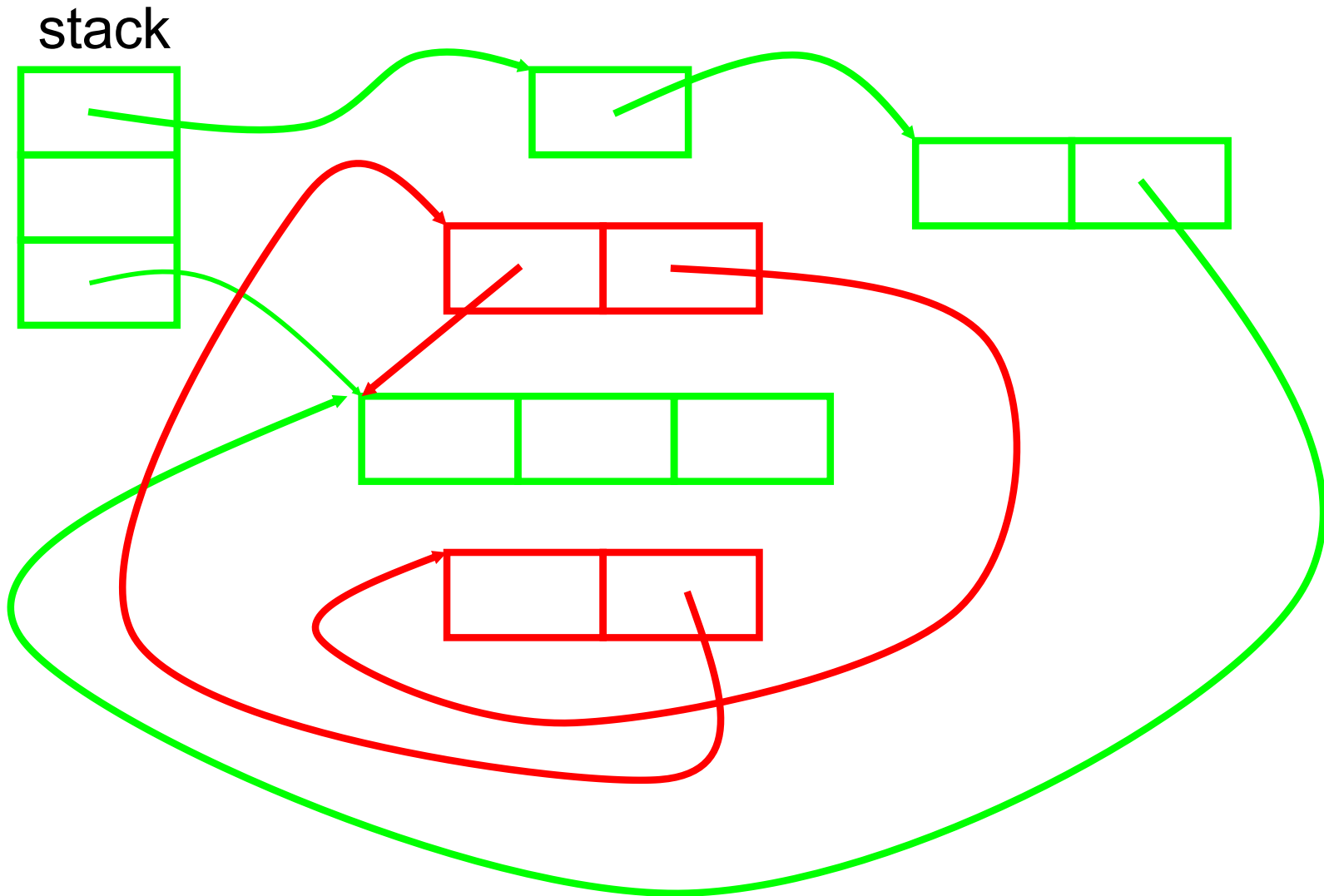
Mark and Sweep Example (cont.)



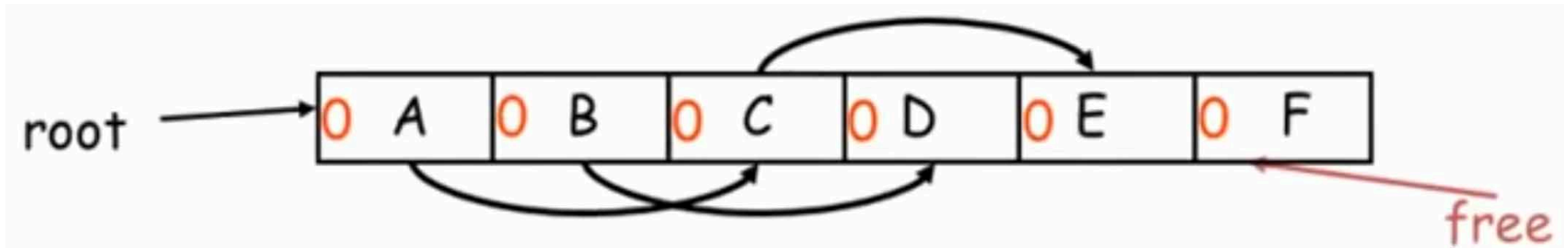
Mark and Sweep Example (cont.)



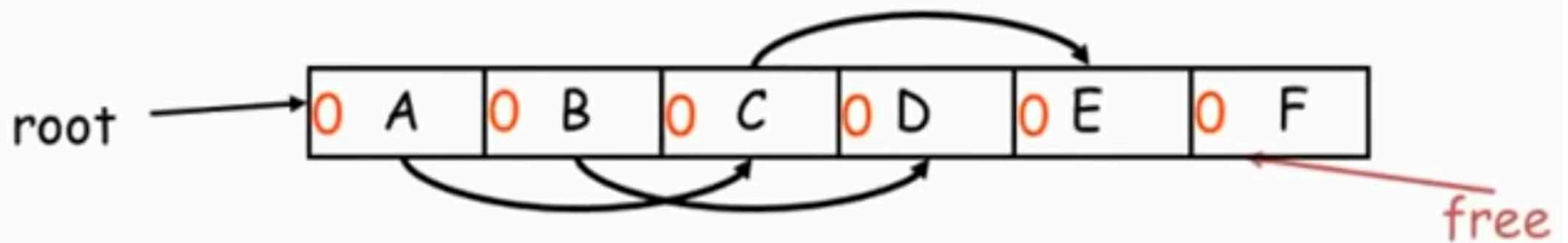
Mark and Sweep Example (cont.)



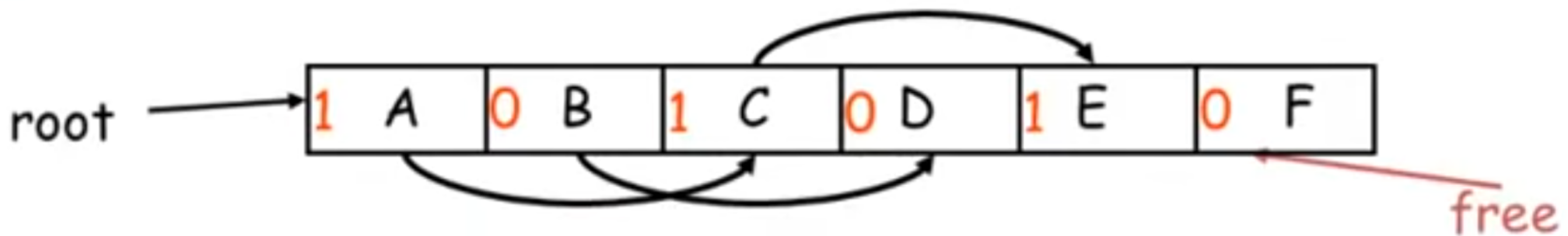
Mark and Sweep Example 2



Mark and Sweep Example 2

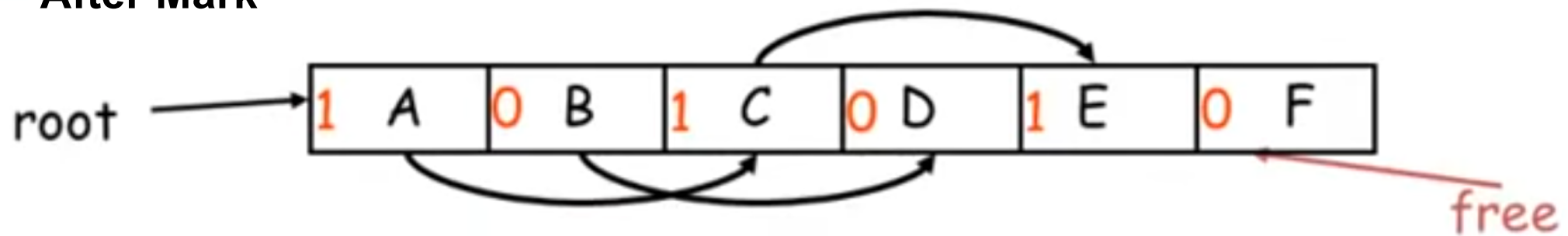


After Mark

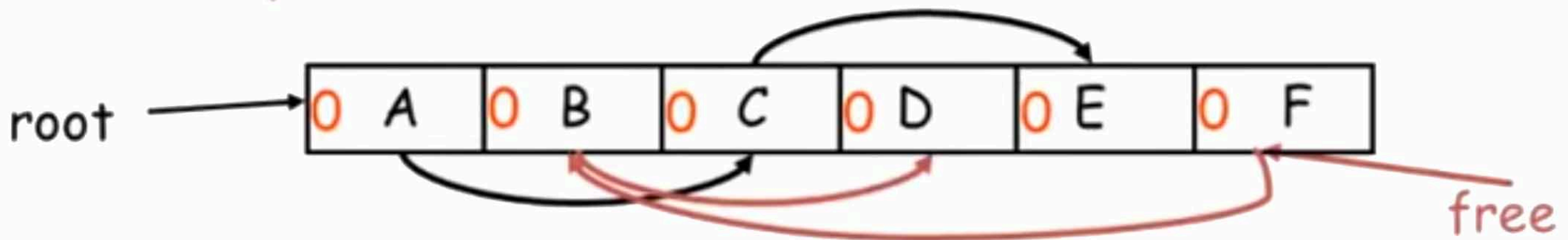


Mark and Sweep Example 2

After Mark



After Sweep



Mark and Sweep Advantages

- ▶ No problem with cycles
- ▶ Non-moving
 - Live objects stay where they are
 - Makes **conservative** GC possible
 - Used when identification of pointer vs. non-pointer uncertain
 - More later

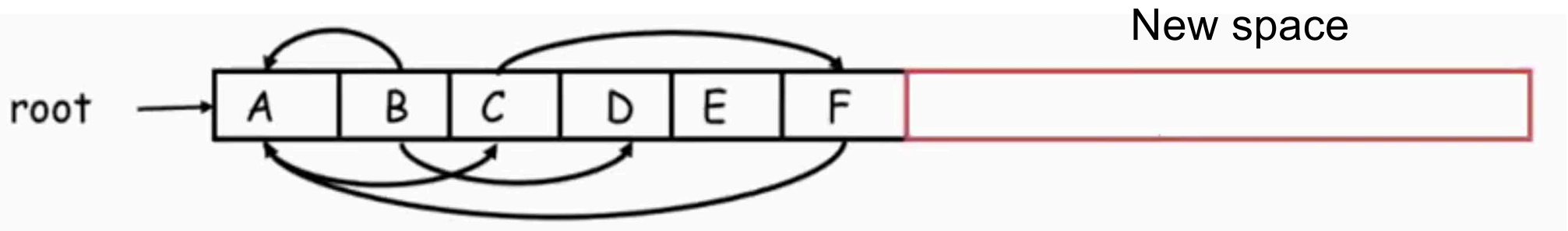
Mark and Sweep Disadvantages

- ▶ Fragmentation
 - Available space broken up into many small pieces
 - Thus many mark-and-sweep systems may also have a compaction phase (like defragmenting your disk)
- ▶ Cost proportional to heap size
 - Sweep phase needs to traverse whole heap – it touches dead memory to put it back on to the free list

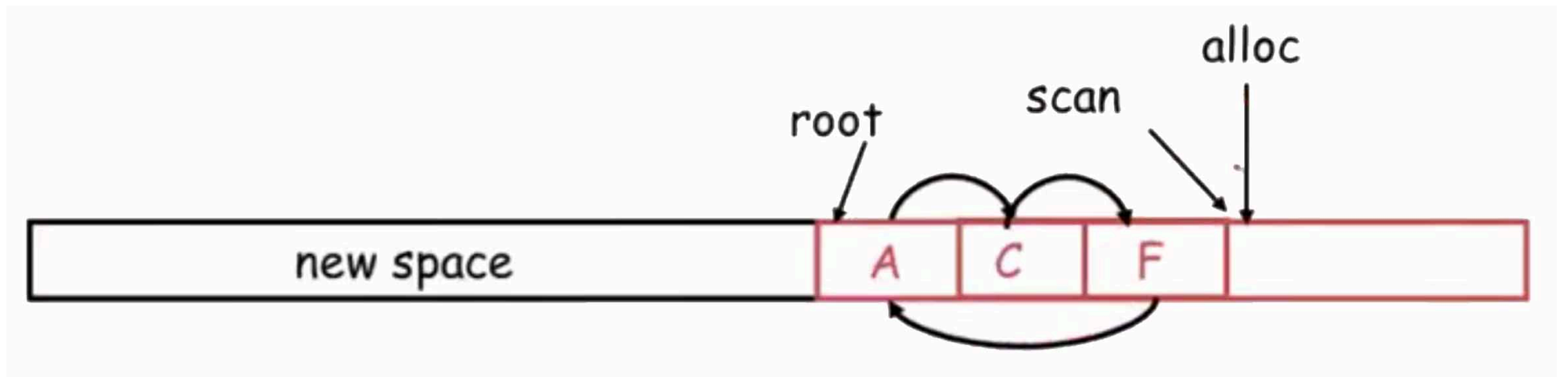
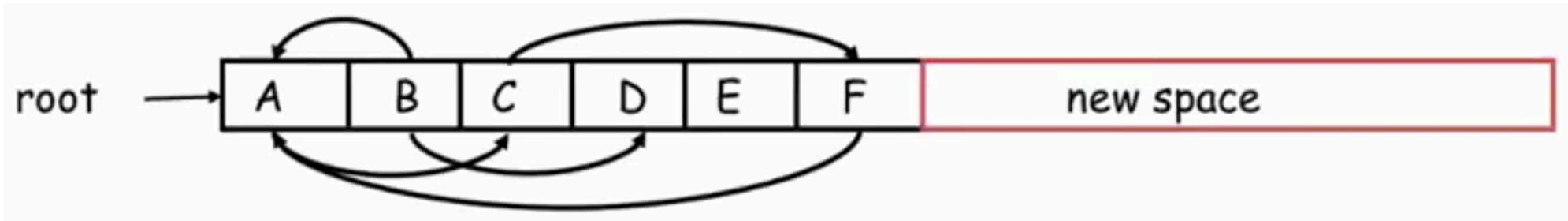
Copying GC

- ▶ Like mark and sweep, but only touches live objects
 - Divide heap into two equal parts (**semispaces**)
 - Only one semispace active at a time
 - At GC time, flip semispaces
 1. Trace the live data starting from the roots
 2. Copy live data into other semispace
 3. Declare everything in current semispace dead
 4. Switch to other semispace

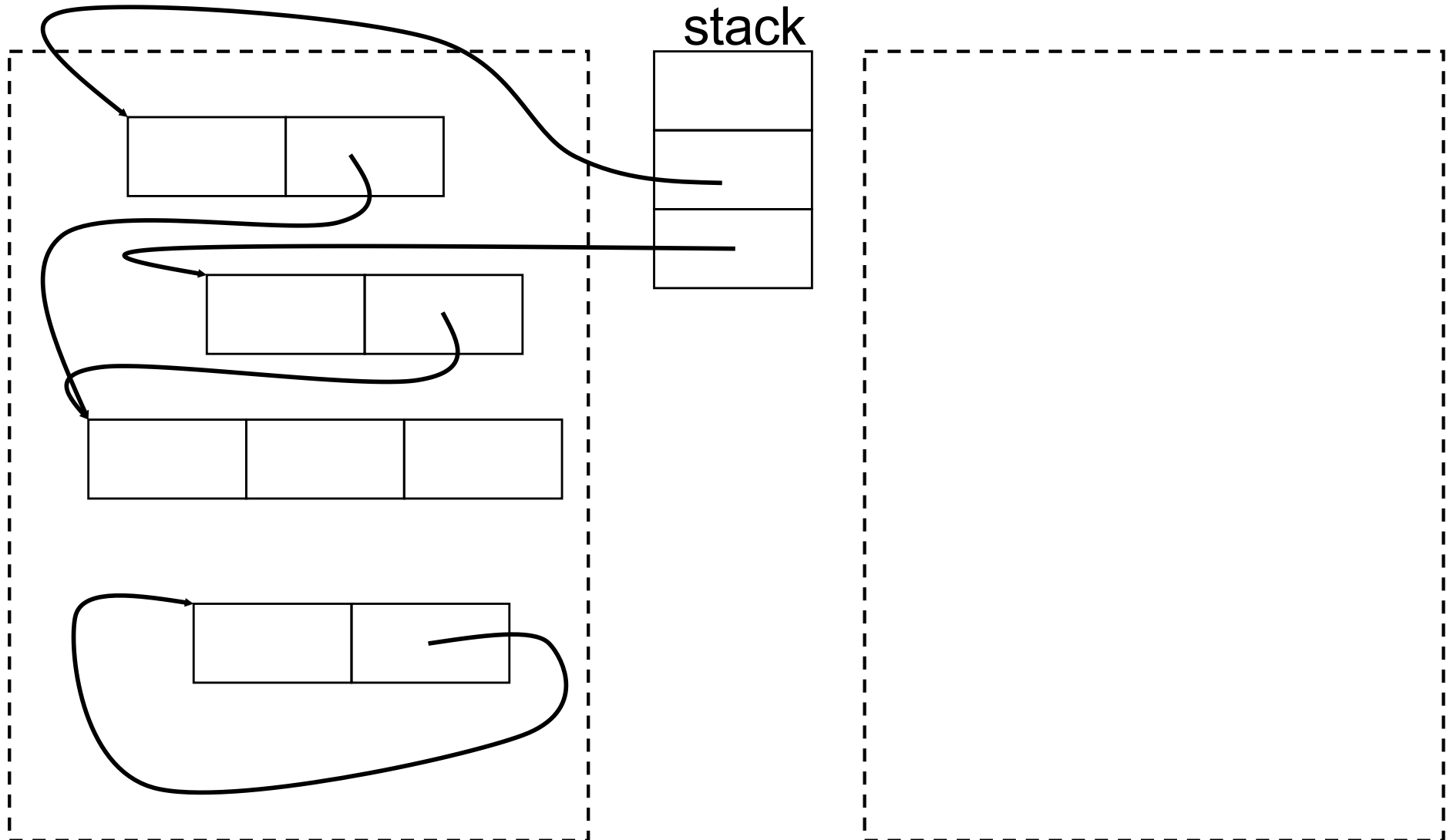
Copying GC Example



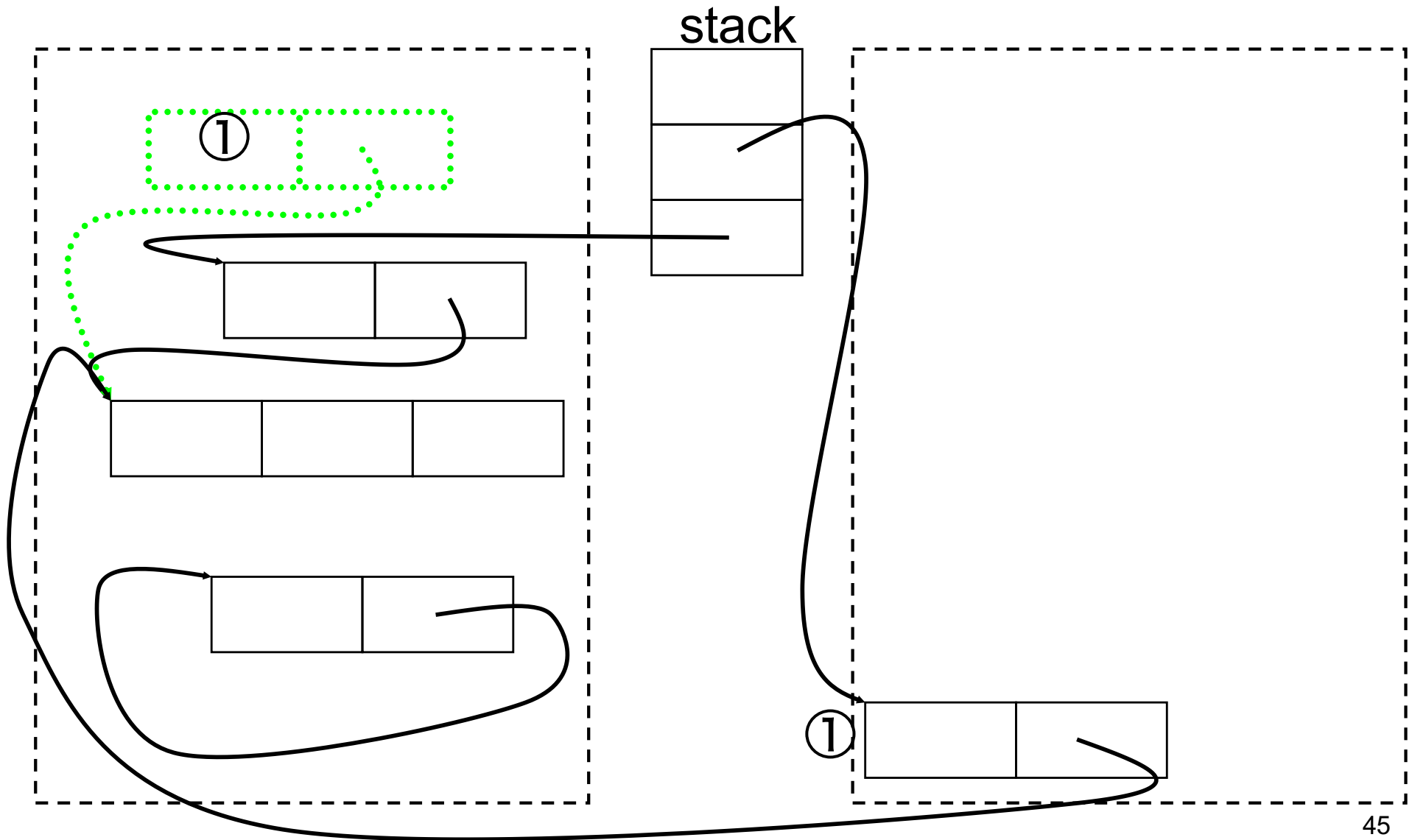
Copying GC Example



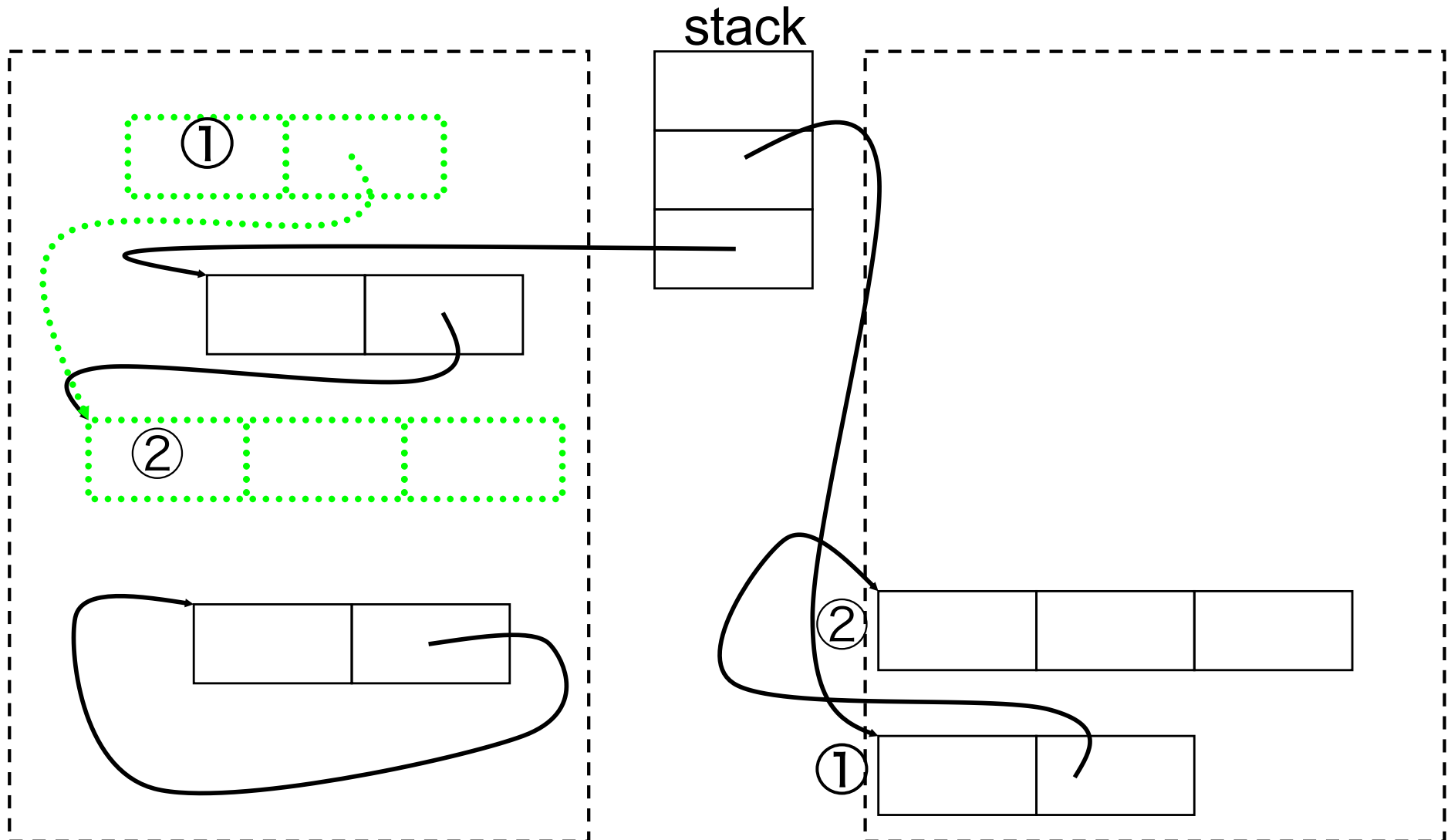
Copying GC Example



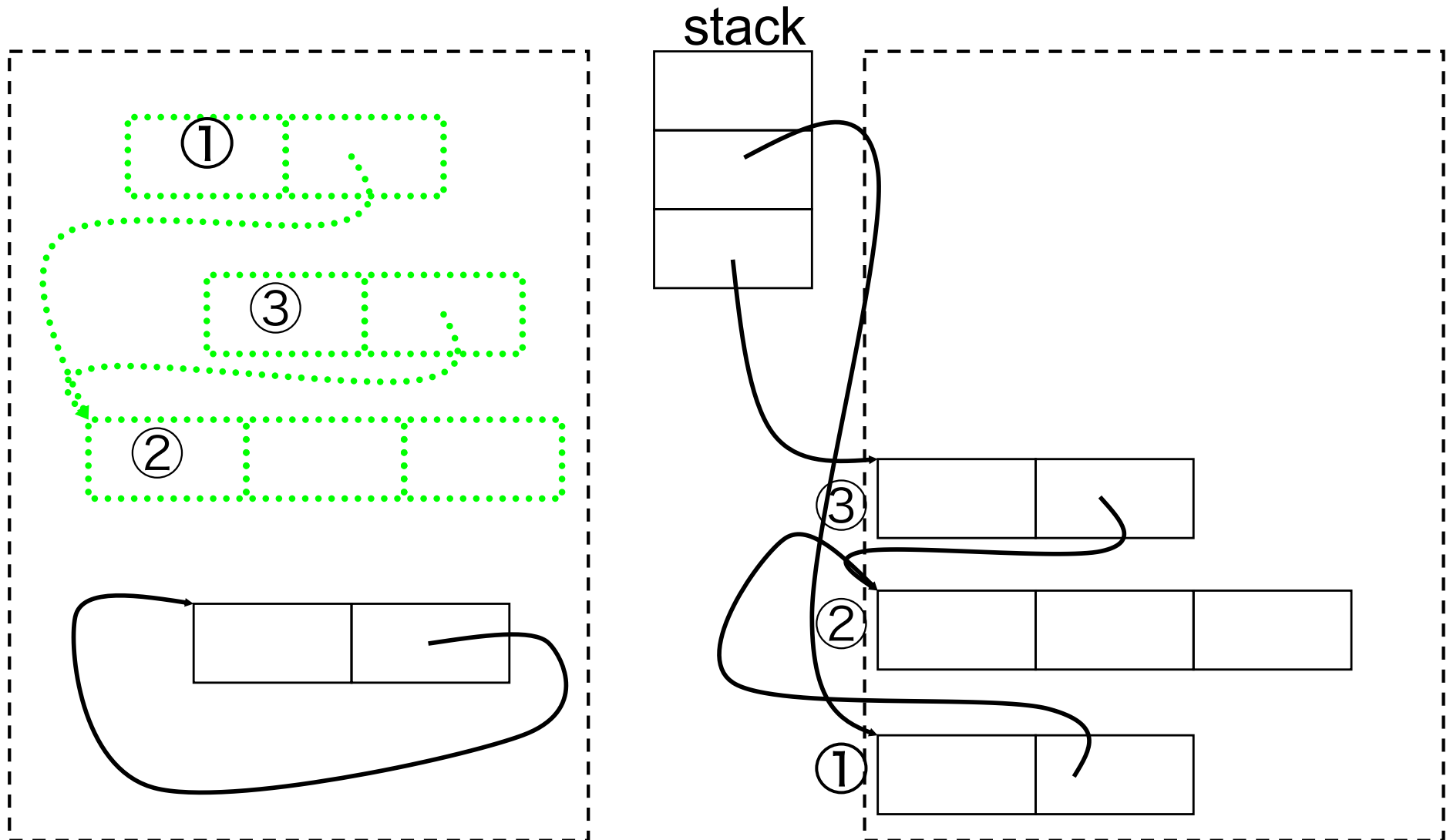
Copying GC Example (cont.)



Copying GC Example (cont.)



Copying GC Example (cont.)



Copying GC Tradeoffs

► Advantages

- Only touches live data
- No fragmentation (automatically compacts)
 - Will probably increase locality

► Disadvantages

- Requires twice the memory space

Quiz 1

Which garbage collection implementation requires more storage?

A. Mark and Sweep

B. Copying GC

Quiz 1

Which garbage collection implementation requires more storage?

A. Mark and Sweep

B. Copying GC

Quiz 2

Which compacts the heap to prevent fragmentation?

- A. Mark and Sweep
- B. Reference Counting
- C. Copying GC

Quiz 2

Which compacts the heap to prevent fragmentation?

- A. Mark and Sweep
- B. Reference Counting
- C. Copying GC**

Quiz 3

The computational cost of Copying GC is proportional to the heap size

- A. True
- B. False

Quiz 3

The computational cost of Copying GC is proportional to the heap size

A. True

B. False

Quiz 4

Which of the following happens most frequently?

- A. Reference Count Updating
- B. Mark and Sweep checking for dead memory
- C. Copying GC copying live data

Quiz 4

Which of the following happens most frequently?

A. Reference Count Updating

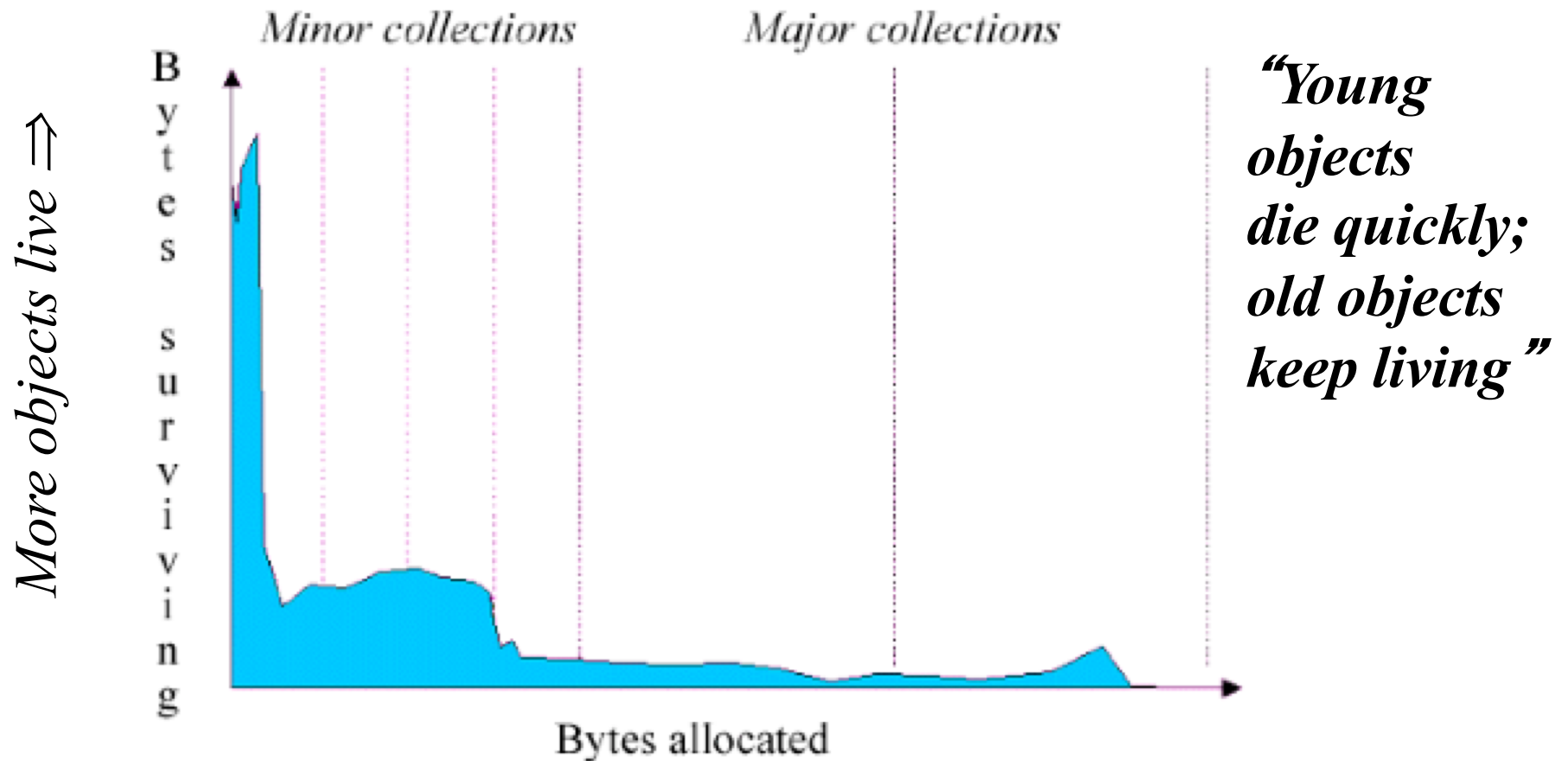
B. Mark and Sweep checking for dead memory

C. Copying GC copying live data

Stop the World: Potentially Long Pause

- ▶ Both of the previous algorithms “stop the world” by prohibiting program execution during GC
 - Ensures that previously processed memory is not changed or accessed, creating inconsistency
- ▶ But the execution pause could be too long
 - Bad if your car’s braking system performs GC while you are trying to stop at a busy intersection!
- ▶ How can we reduce the pause time of GC?
 - Don’t collect the whole heap at once (incremental)

The Generational Principle



Object lifetime increases \Rightarrow

Generational Collection

- ▶ Long lived objects visited multiple times
 - Idea: Have more than one heap region, divide into generations
 - Older generations collected less often
 - Objects that survive many collections get promoted into older generations
 - Need to track pointers from old to young generations to use as roots for young generation collection
 - Tracking one in the remembered set
- ▶ One popular setup: Generational, copying GC

Conservative Garbage Collection (for C)

- ▶ For C, we cannot be sure which elements of an object are pointers
 - Because of incomplete type information, the use of unsafe casts, etc.
- ▶ Idea: suppose it is a pointer if it looks like one
 - Most pointers are within a certain address range, they are word aligned, etc.
 - May retain memory spuriously
- ▶ Different styles of conservative collector
 - Mark-sweep: important that objects not moved
 - Mostly-copying: can move objects you are sure of

Memory Management in Ruby

- ▶ Local variables live on the stack
 - Storage reclaimed when method returns
- ▶ Objects live on the heap
 - Created with calls to `Class.new`
- ▶ Objects never explicitly freed
 - Ruby uses automatic memory management

Memory Management in OCaml

- ▶ Local variables live on the stack
- ▶ Tuples, closures, and constructed types live on the heap
 - `let x = (3, 4) (* heap-allocated *)`
 - `let f x y = x + y in f 3`
(* result heap-allocated *)
 - `type 'a t = None | Some of 'a`
 - `None` (* not on the heap—just a primitive *)
 - `Some 37` (* heap-allocated *)
- ▶ Garbage collection reclaims memory

Memory Management in Java

- ▶ Local variables live on the stack
 - Allocated at method invocation time
 - Deallocated when method returns
- ▶ Other data lives on the heap
 - Memory is allocated with `new`
 - But never explicitly deallocated
 - Java uses automatic memory management

Java HotSpot SDK 1.4.2 Collector

- ▶ Multi-generational, hybrid collector
 - Young generation
 - Stop and copy collector
 - Tenured generation
 - Mark and sweep collector
 - Permanent generation
 - No collection

More Issues in GC (cont.)

- ▶ Stopping the world is a big hit
 - Unpredictable performance
 - Bad for real-time systems
 - Need to stop all threads
 - Without a much more sophisticated GC
- ▶ One-size-fits-all solution
 - Sometimes, GC just gets in the way
 - But correctness comes first

What Does GC Mean to You?

- ▶ Ideally, nothing
 - GC should make programming easier
 - GC should not affect performance (much)
- ▶ Usually bad idea to manage memory yourself
 - Using object pools, free lists, object recycling, etc...
 - GC implementations have been heavily tuned
 - May be more efficient than explicit deallocation
- ▶ If GC becomes a problem, hard to solve
 - You can set parameters of the GC
 - You can modify your program

Increasing Memory Performance

- ▶ Don't allocate as much memory
 - Less work for your application
 - Less work for the garbage collector

- ▶ Don't hold on to references
 - Null out pointers in data structures
 - Example

```
Object a = new Object;
```

```
...use a...
```

```
a = null;           // when a is no longer needed
```

Find the Memory Leak

```
class Stack {  
    private Object[] stack;  
    private int index;  
    public Stack(int size) {  
        stack = new Object[size];  
    }  
    public void push(Object o) {  
        stack[index++] = o;  
    }  
    public void pop() {  
        return stack[index--];  
    }  
}
```

From Hagggar, Garbage Collection and the Java Platform Memory Model

Find the Memory Leak

```
class Stack {  
    private Object[] stack;  
    private int index;  
    public Stack(int size) {  
        stack = new Object[size];  
    }  
    public void push(Object o) {  
        stack[index++] = o;  
    }  
    public void pop() {  
        stack[index] = null; // null out ptr  
        return stack[index--];  
    }  
}
```

From Haggar, Garbage Collection and the Java Platform Memory Model

Answer: pop() leaves item on stack array; storage not reclaimed