CMSC 330: Organization of Programming Languages

Garbage Collection

Memory Attributes

- Memory to store data in programming languages has several attributes
  - Persistence (or lifetime)
    - How long the memory exists
  - Allocation
    - When the memory is available for use
  - Recovery
    - When the system recovers the memory for reuse

Memory Attributes (cont.)

- Most programming languages are concerned with some subset of the following 4 memory classes
  1. Fixed (or static) memory
  2. Automatic memory
  3. Programmer allocated memory
  4. Persistent memory

Memory Classes

- Static memory – Usually a fixed address in memory
  - Persistence – Lifetime of execution of program
  - Allocation – By compiler for entire execution
  - Recovery – By system when program terminates
- Automatic memory – Usually on a stack
  - Persistence – Lifetime of method using that data
  - Allocation – When method is invoked
  - Recovery – When method terminates

Memory Classes (cont.)

- Allocated memory – Usually memory on a heap
  - Persistence – As long as memory is needed
  - Allocation – Explicitly by programmer
  - Recovery – Either by programmer or automatically (when possible and depends upon language)

Memory Classes (cont.)

- Persistent memory – Usually the file system
  - Persistence – Multiple execution of a program
    - E.g., files or databases
  - Allocation – By program or user
    - Often outside of program execution
  - Recovery – When data no longer needed
  - Dealing with persistent memory → databases
    - CMSC 424

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 Mistakes

- 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! */
}
```

- May try to 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

- Don't allocate memory on the heap
  - Often impractical
  - Leads to confusing code (e.g., alloca())
- Never free memory
  - OS will reclaim process's memory anyway at exit
  - Memory is cheap; who cares about a little leak?
  - LISP model – System halts program and reclaims unused memory when there is no more available
- Use a garbage collector
  - E.g., conservative Boehm-Weiser collector for C

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
    - Uses a garbage collector to reclaim memory

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

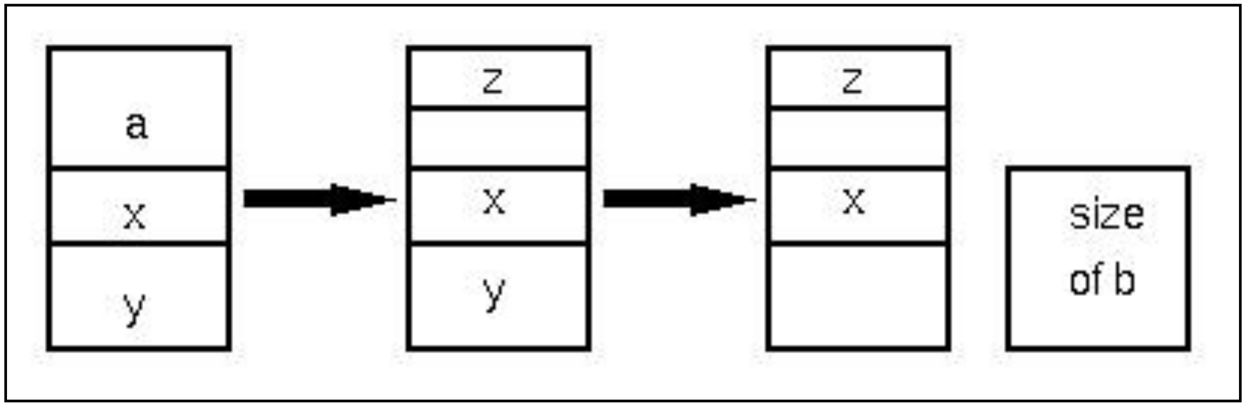
- Local variables live on the stack
  - Allocated at method invocation time
  - Deallocated when method returns

- Other data lives on the heap
  - Memory is allocated with new
  - But never explicitly deallocated
    - Java uses automatic memory management

Fragmentation

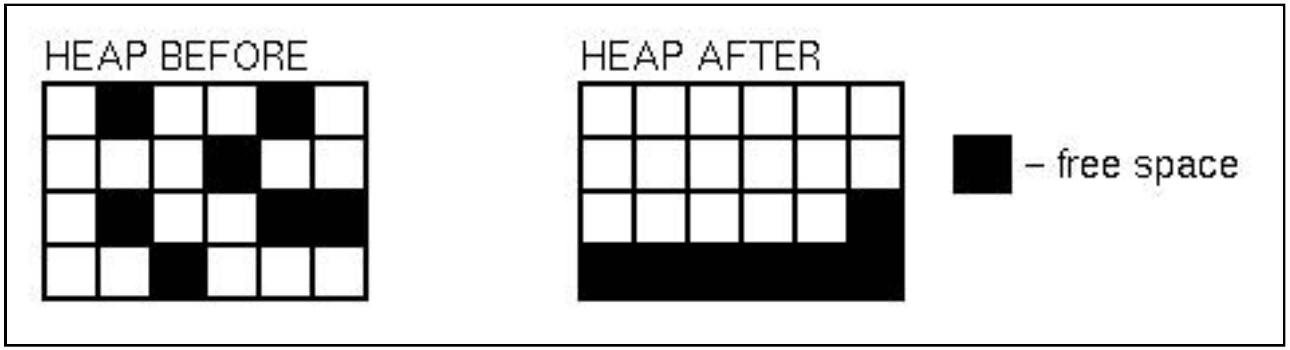
- Another memory management problem
- Example sequence of calls

  allocate(a);
  allocate(x);
  allocate(y);
  free(a);
  allocate(z);
  free(y);
  allocate(b);
  ⇒ Not enough contiguous space for b



Garbage Collection Goal

- Process to reclaim memory
  - Also solve fragmentation



- Algorithm: You can do garbage collection and memory compaction if you know where every pointer is in a program. If you move the allocated storage, simply change the pointer to it.
- This is true in LISP, OCAML, Java, Prolog
- Not true in C, C++, Pascal, Ada

Garbage Collection (GC)

- 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"
- Idea
  - Can reuse memory from dead objects (recycling!)
- Goals
  - Reduce memory leaks
  - Make dangling pointers impossible

Many GC Techniques

- 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 an 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

Reachability

- An object is reachable if it can be accessed by 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

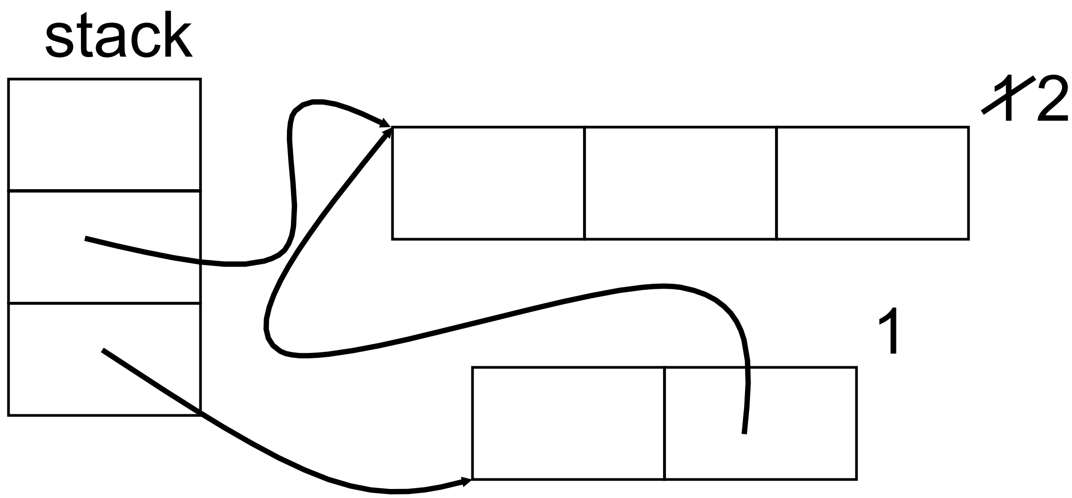
Roots

- 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 pointer chasing

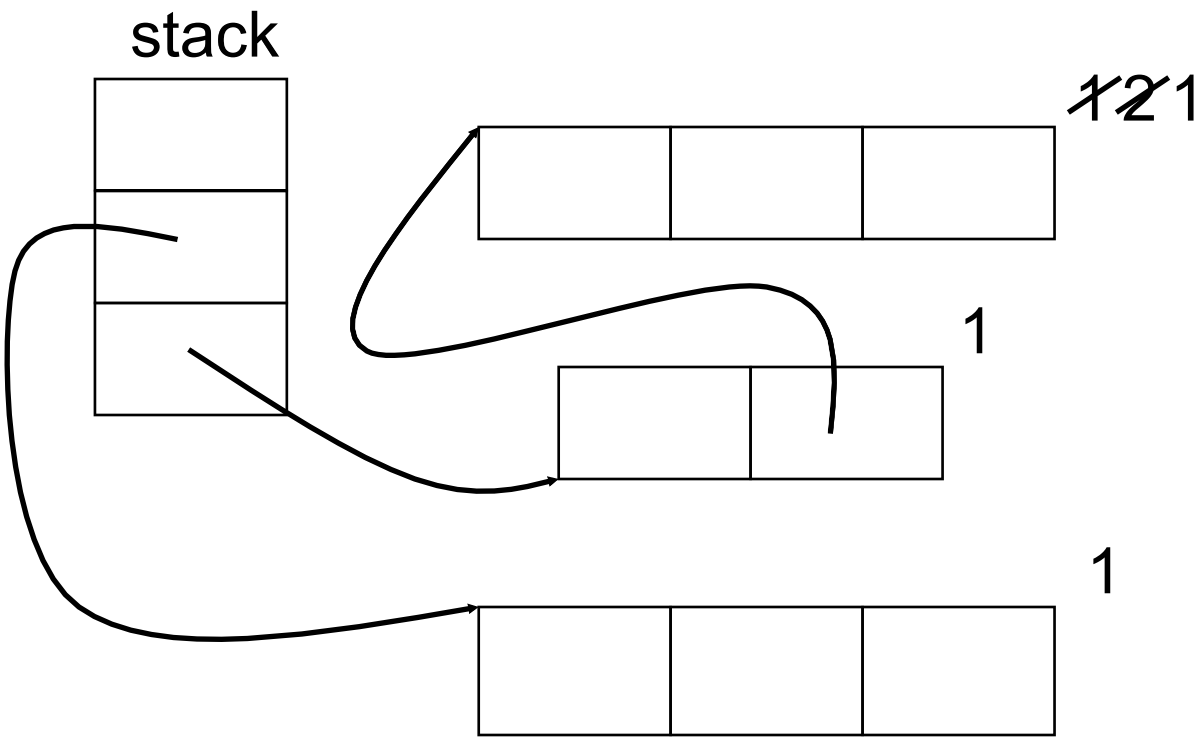
Reference Counting

- Old technique (1960)
- Each object has count of number of pointers to it from other objects and from the stack
  - When count reaches 0, object can be deallocated
- Counts tracked by either compiler or manually
- To find pointers, need to know layout of objects
  - In particular, need to distinguish pointers from ints
- Method works mostly for reclaiming memory
  - Doesn't handle fragmentation problem

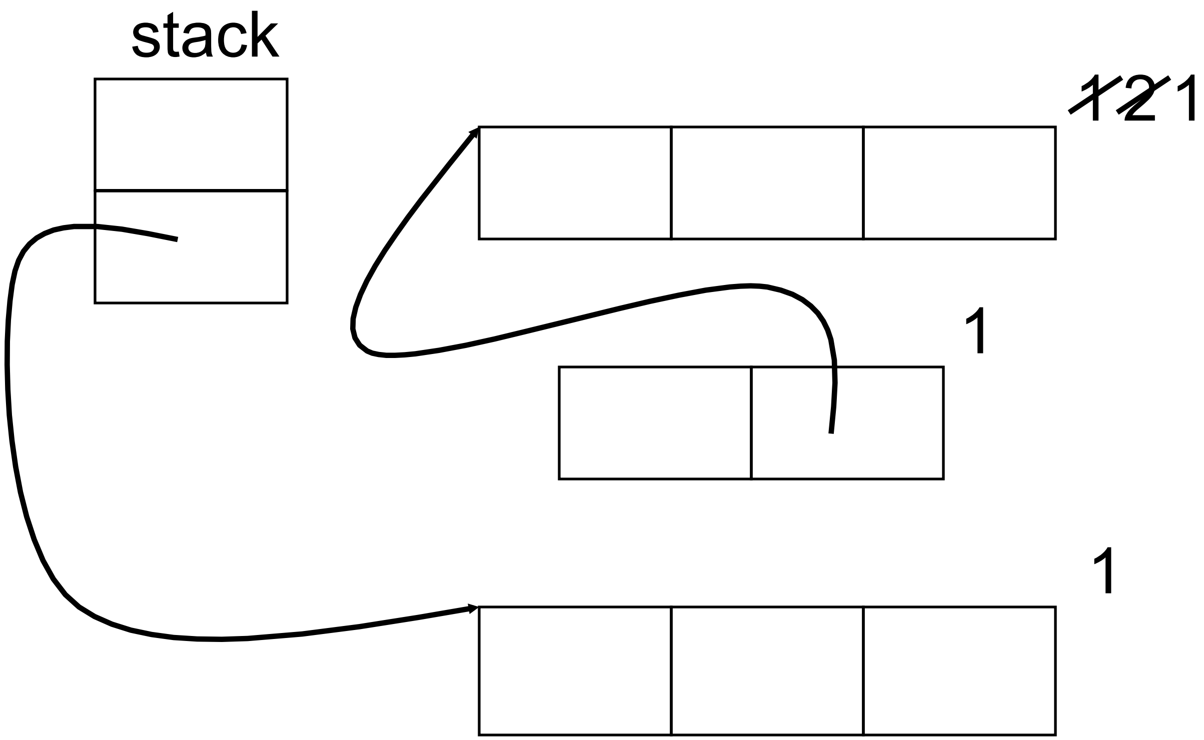
Reference Counting Example



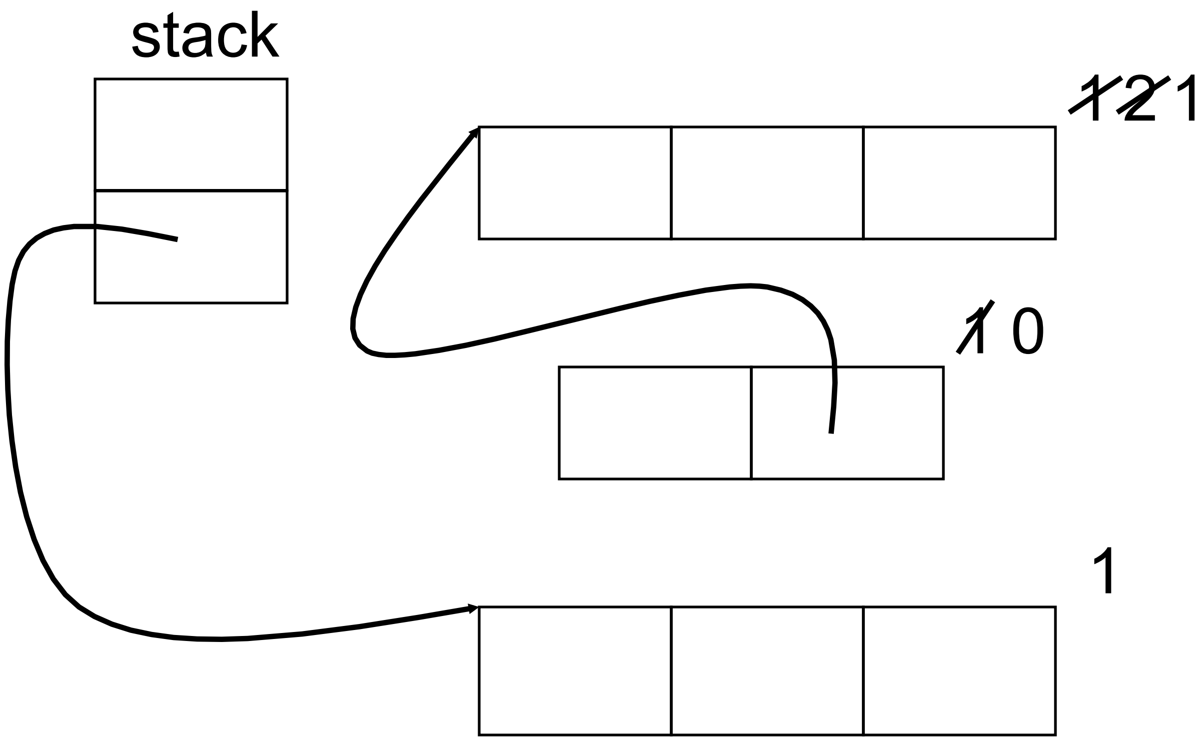
Reference Counting Example (cont.)



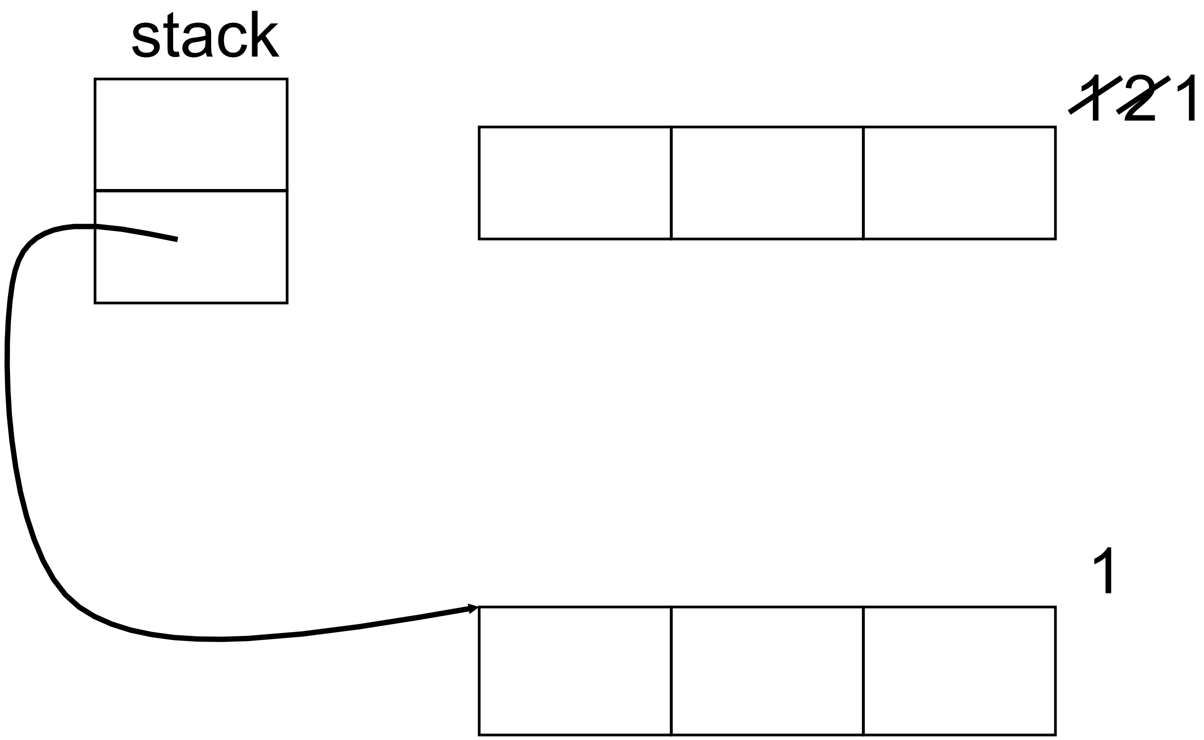
Reference Counting Example (cont.)



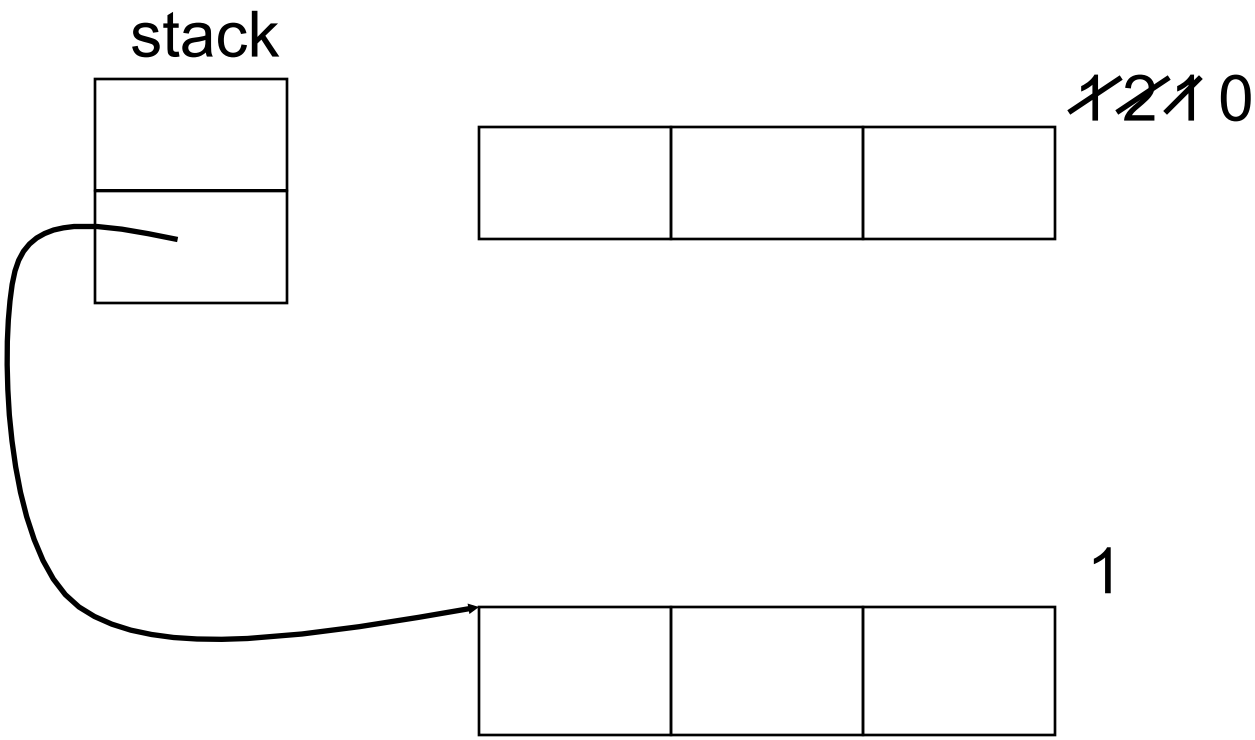
Reference Counting Example (cont.)



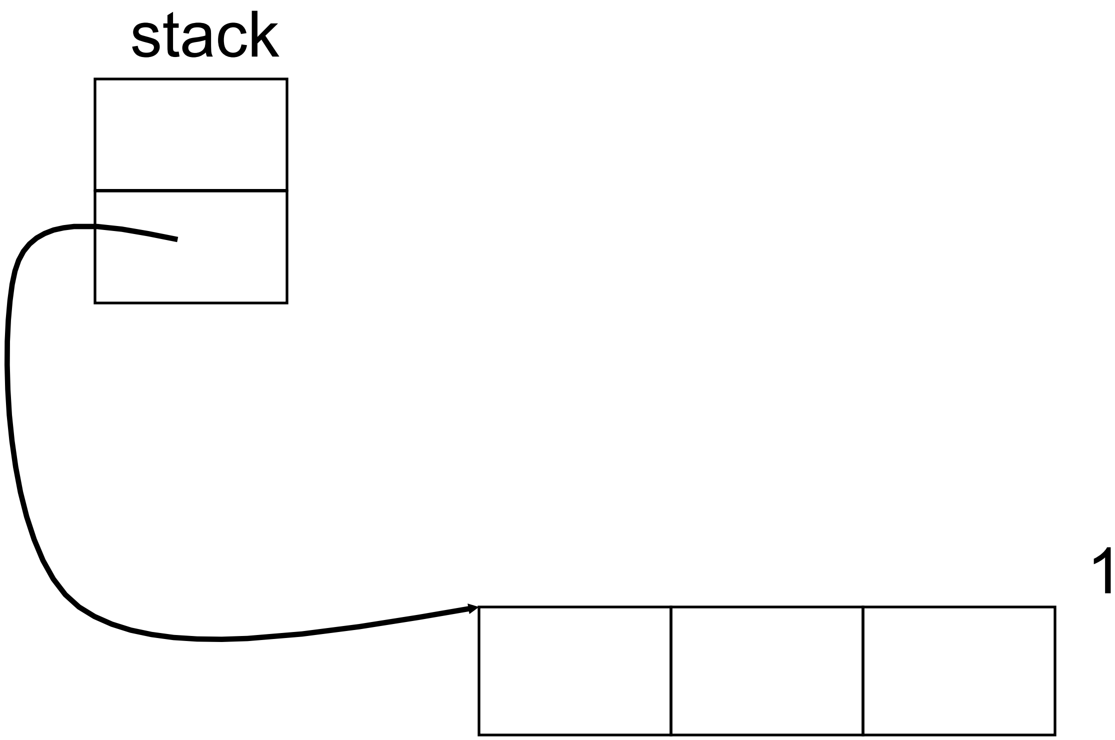
Reference Counting Example (cont.)



Reference Counting Example (cont.)

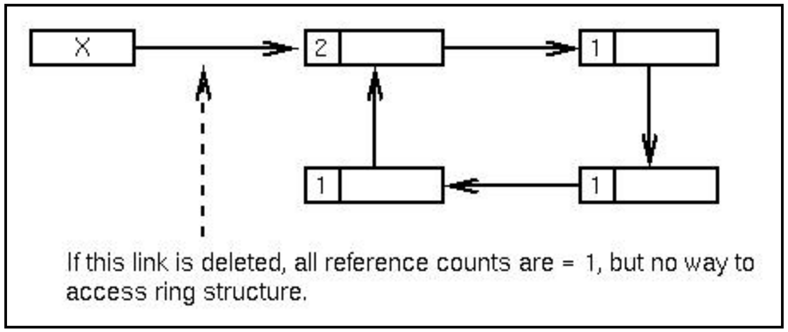


Reference Counting Example (cont.)



Reference Counting Tradeoffs

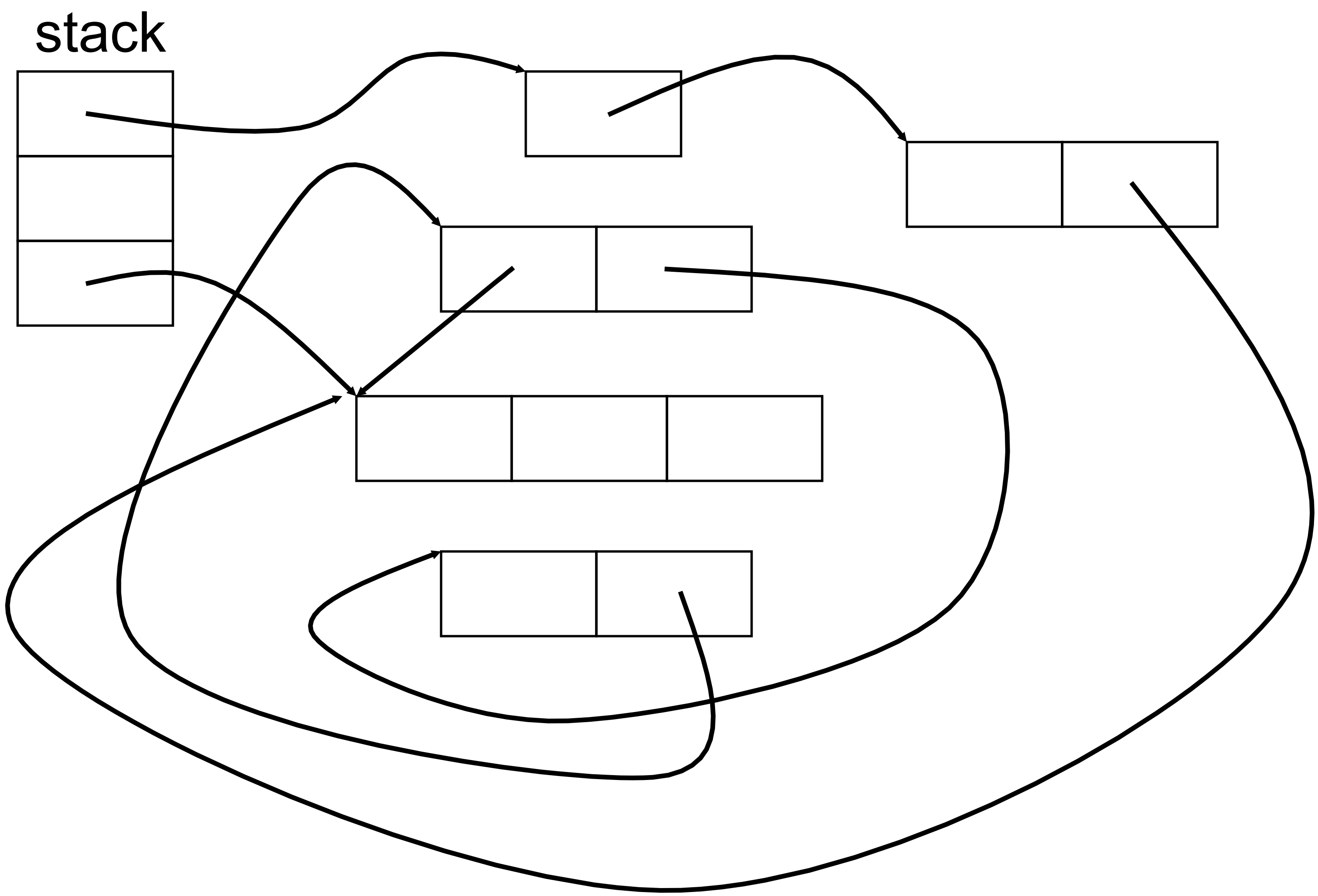
- Advantage
  - Incremental technique
    - Generally small, constant amount of work per memory write
    - With more effort, can even bound running time
- Disadvantages
  - Cascading decrements can be expensive
  - Requires extra storage for reference counts
  - Can't collect cycles, since counts never go to 0



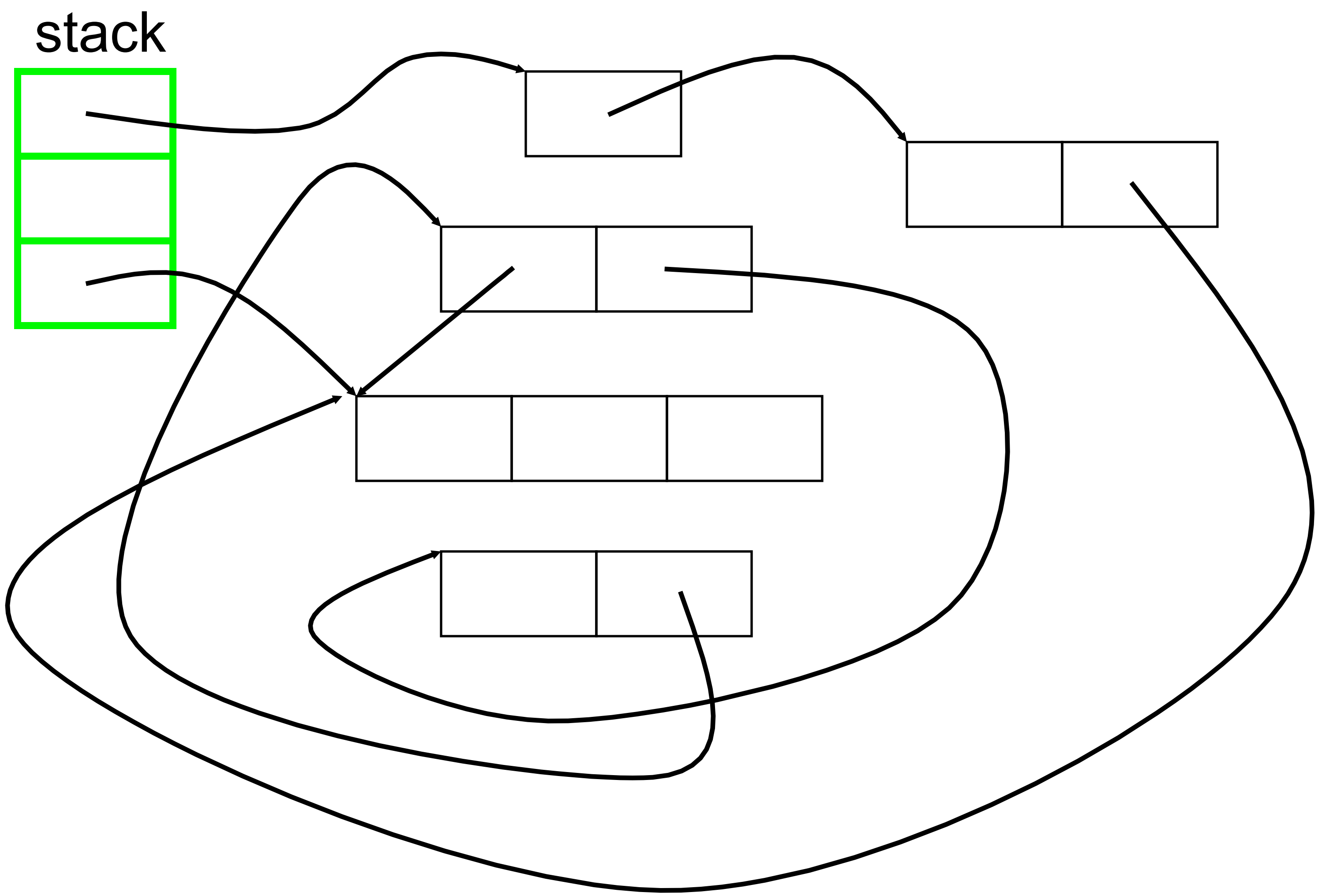
Mark and Sweep GC

- Idea
  - Only objects reachable from stack can be live
- Every so often, stop the world and do GC
  - Mark all objects on stack as live
  - Mark object reachable from live object as live
    - Repeat until no more reachable objects
  - Deallocate any non-reachable objects
- This is a tracing garbage collector
  - Does not handle fragmentation problem

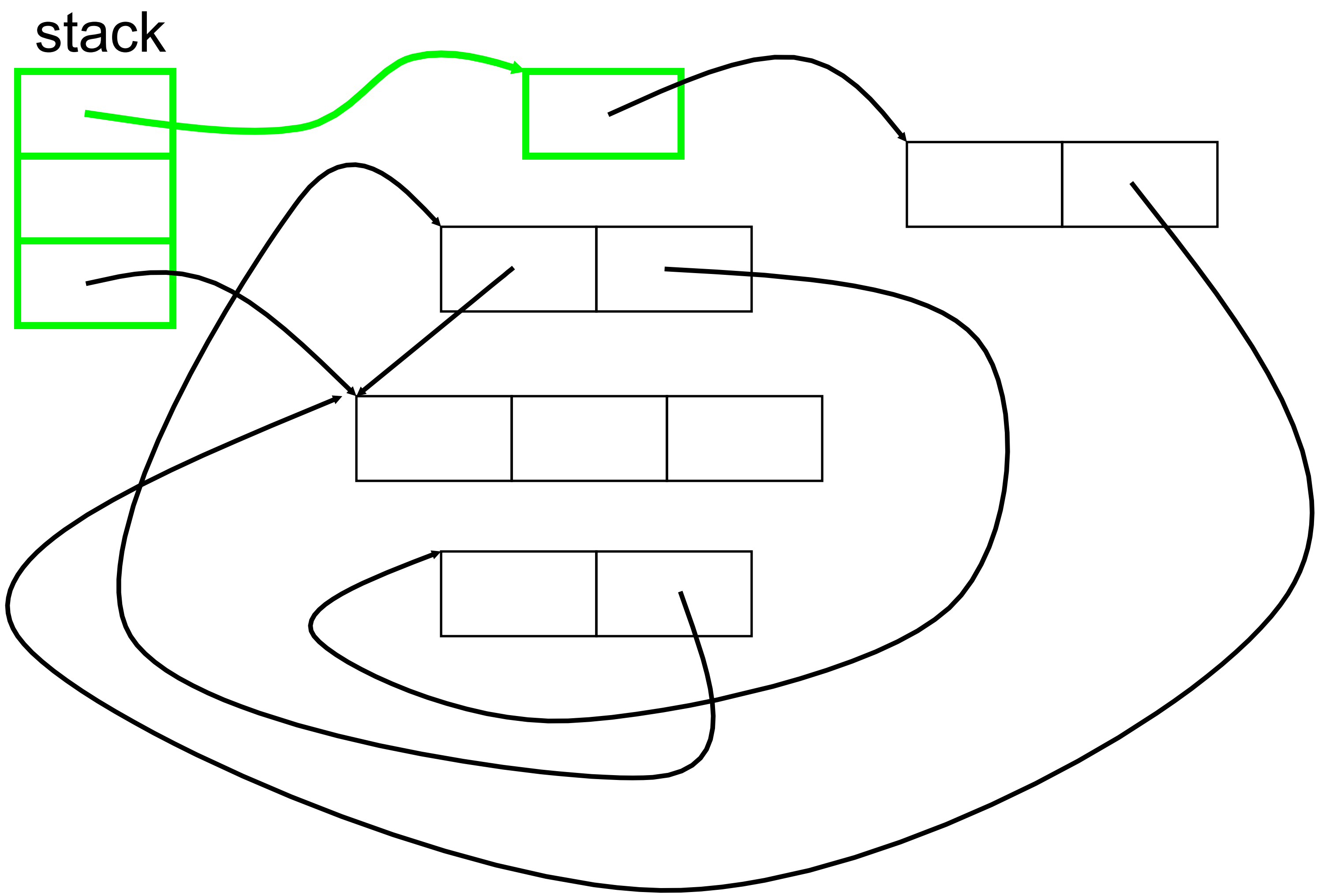
Mark and Sweep Example



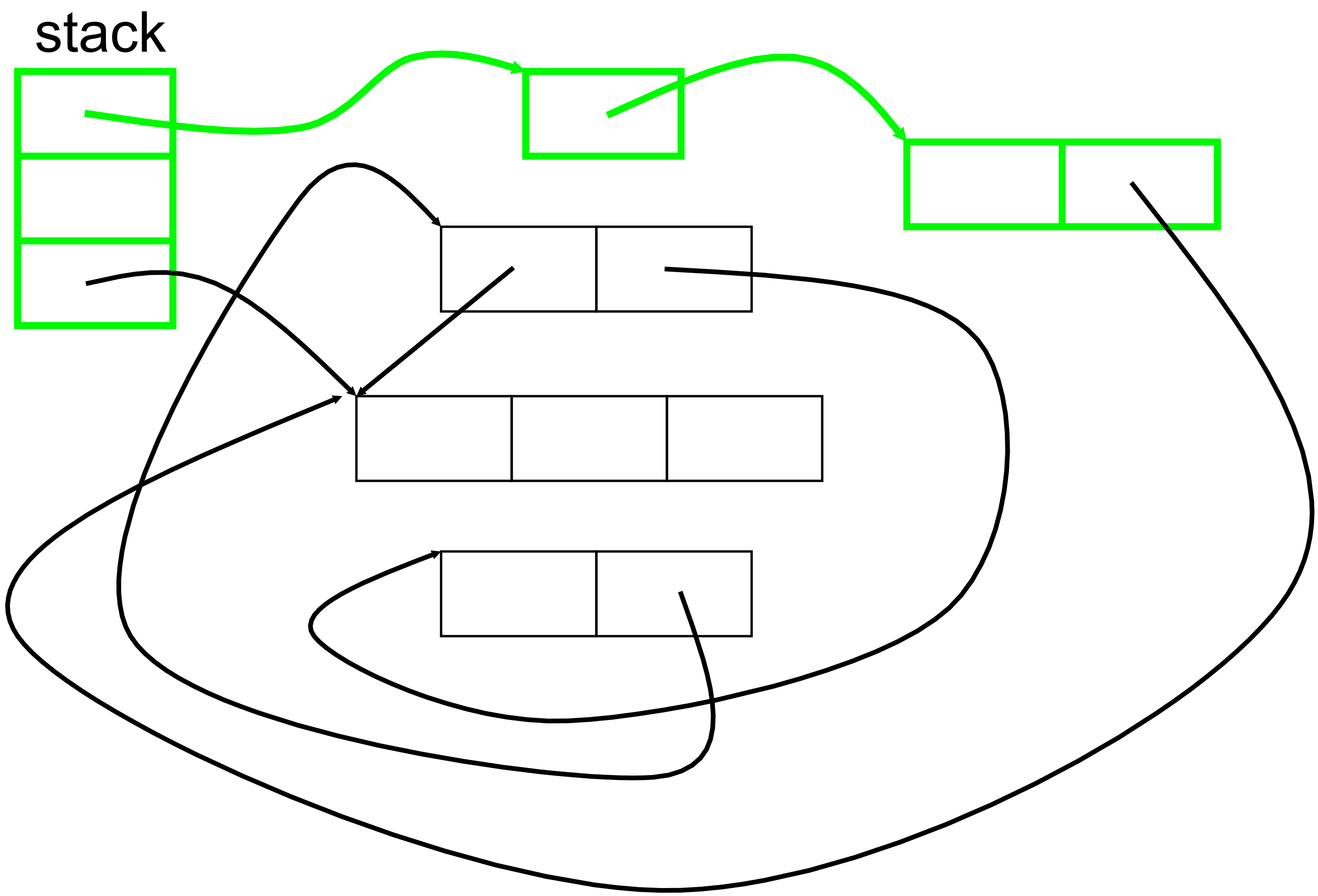
Mark and Sweep Example (cont.)



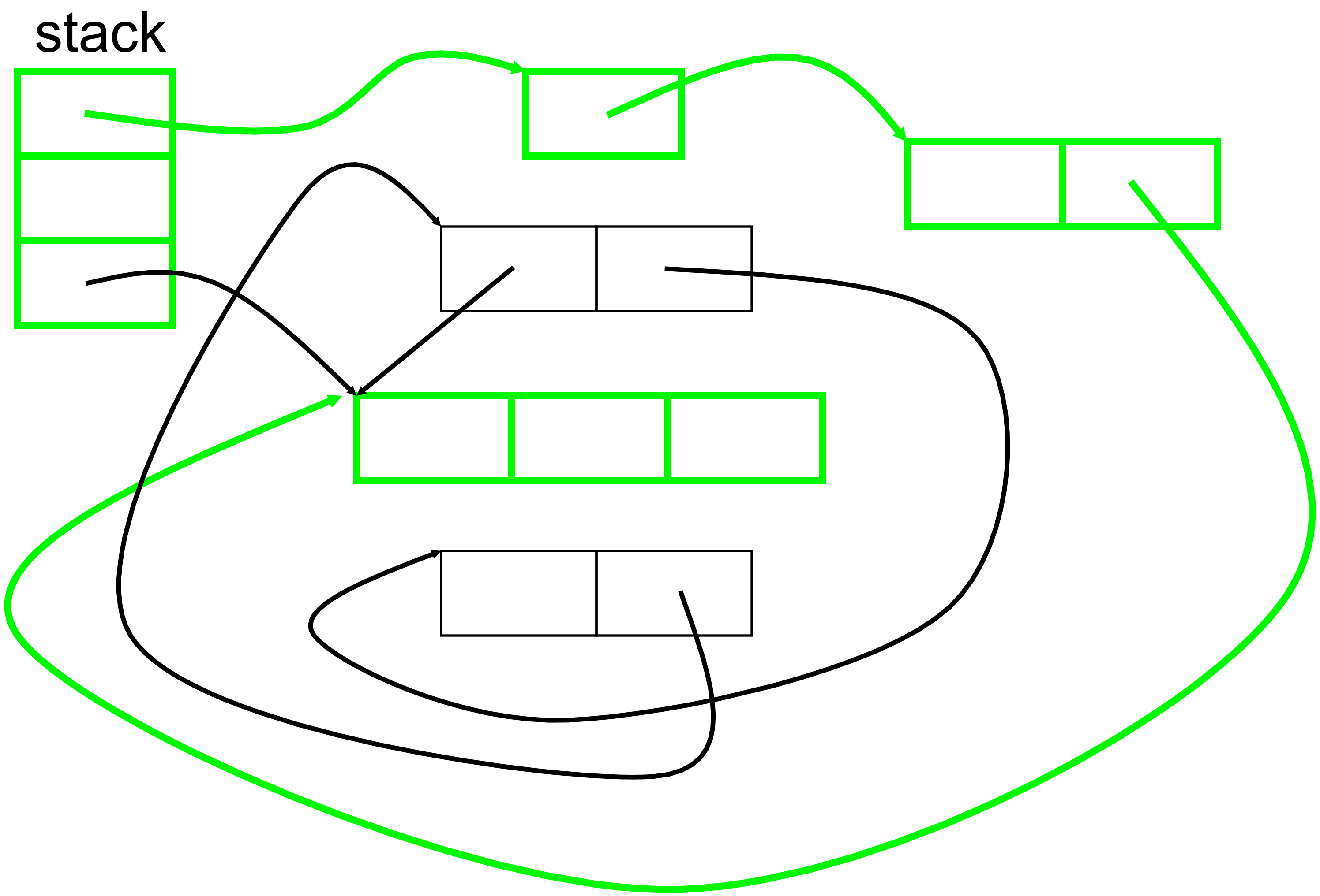
Mark and Sweep Example (cont.)



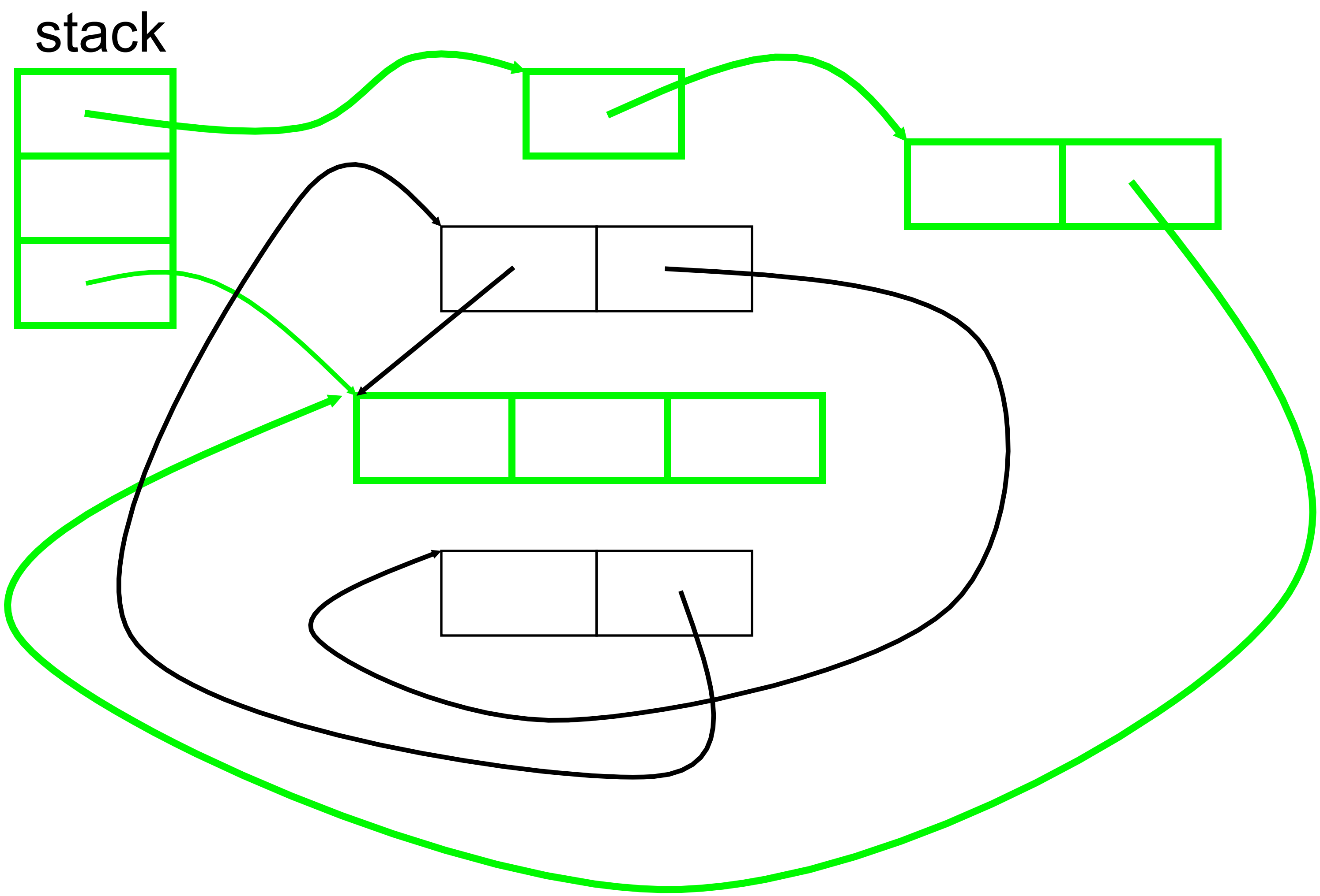
Mark and Sweep Example (cont.)



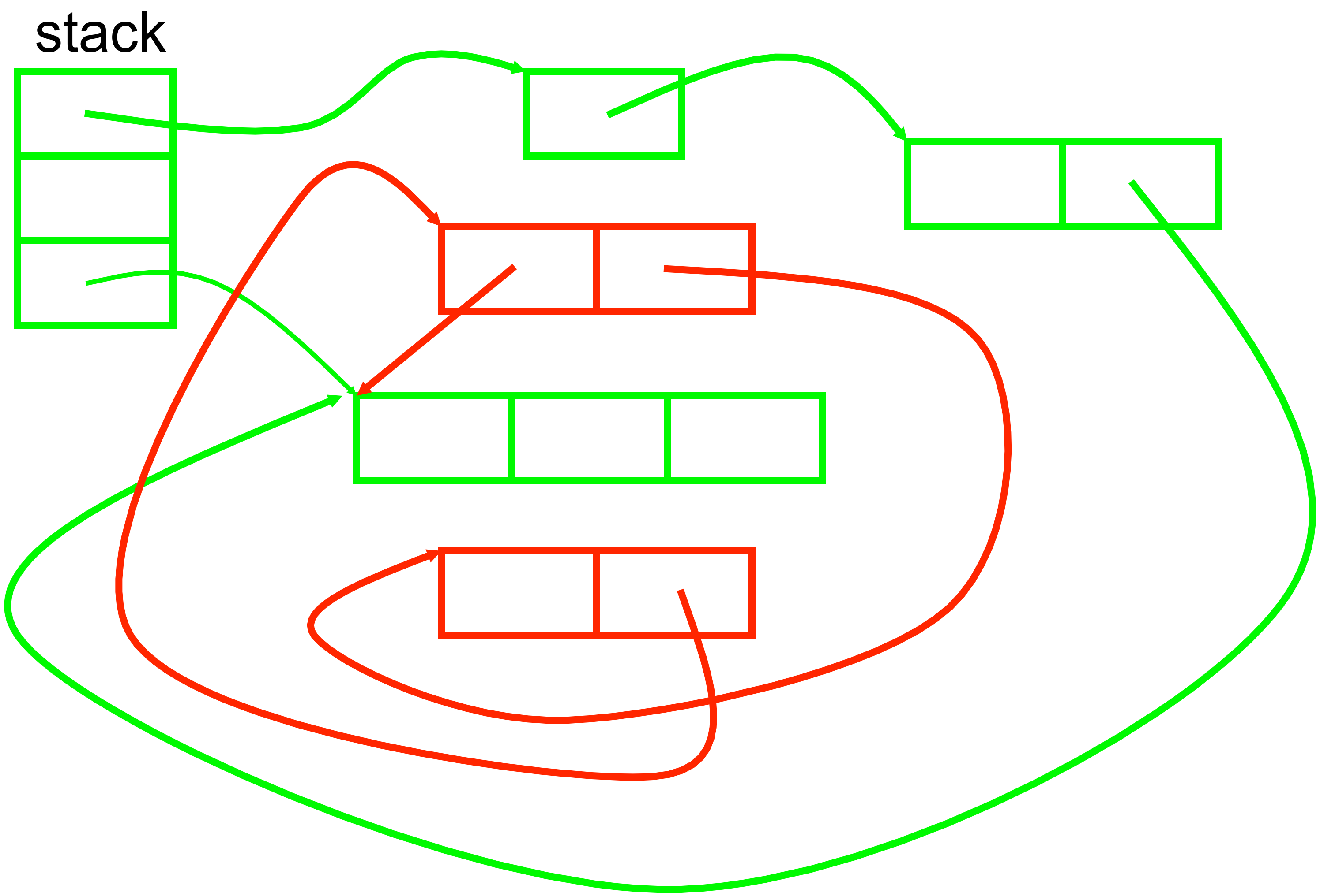
Mark and Sweep Example (cont.)



Mark and Sweep Example (cont.)



Mark and Sweep Example (cont.)



Mark and Sweep Tradeoffs

- Advantages
  - No problem with cycles
  - Memory writes have no cost

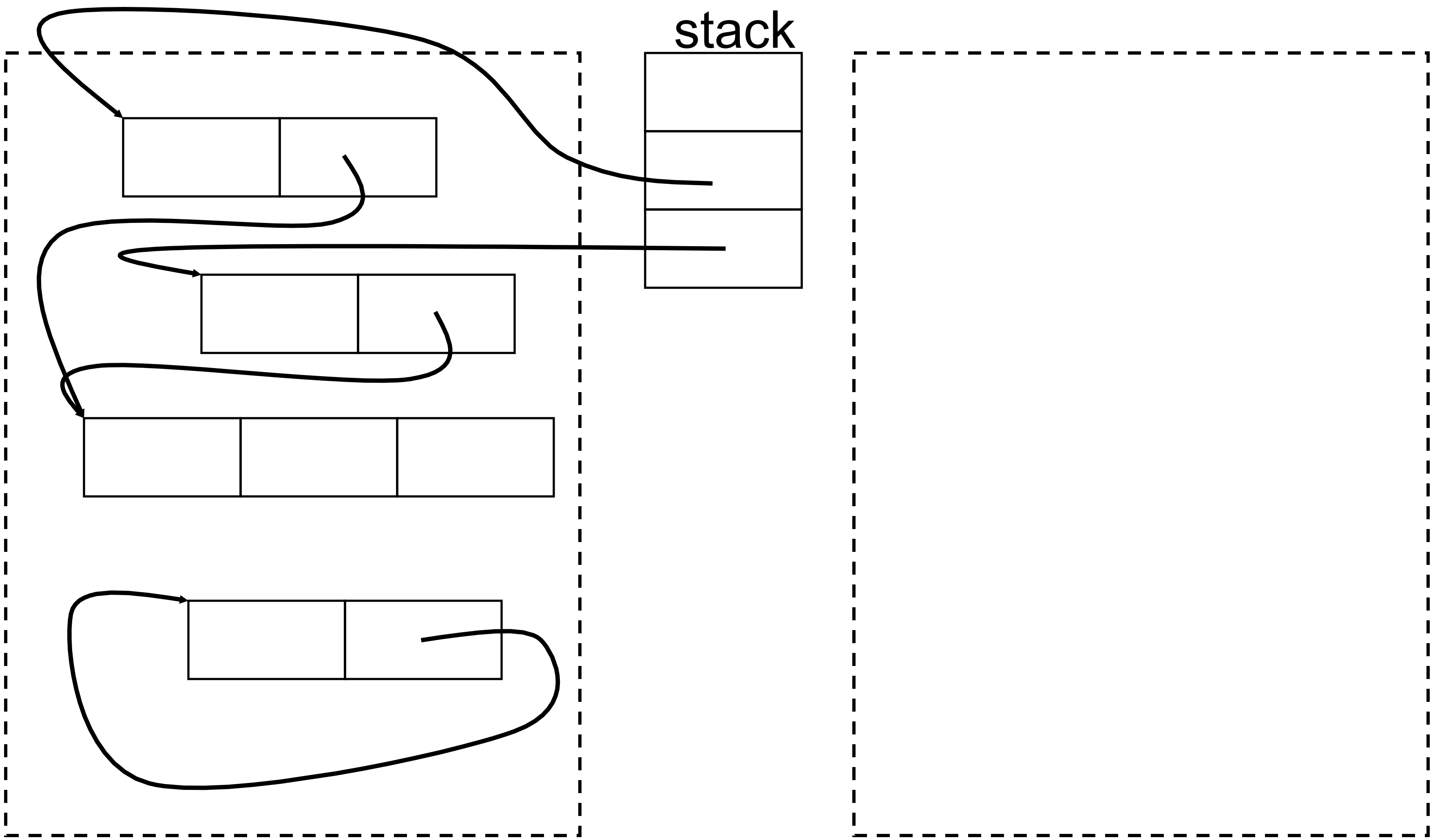
Mark and Sweep Tradeoffs (cont.)

- 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
  - Not appropriate for real-time applications
    - Bad if your car's braking system performs GC while you are trying to stop at a busy intersection

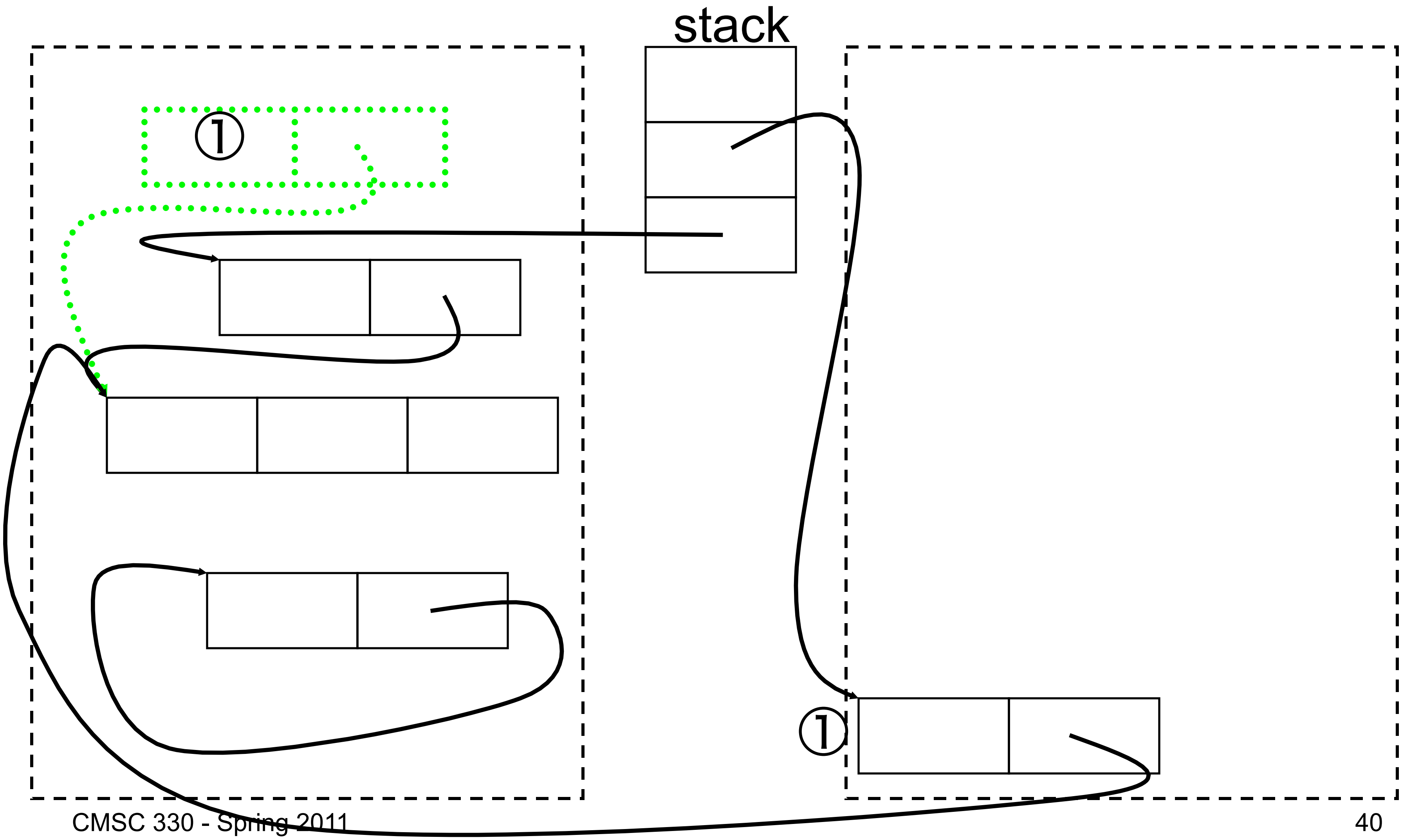
Stop and Copy GC

- 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 stack
    2. Copy live data into other semispace
    3. Declare everything in current semispace dead
    4. Switch to other semispace

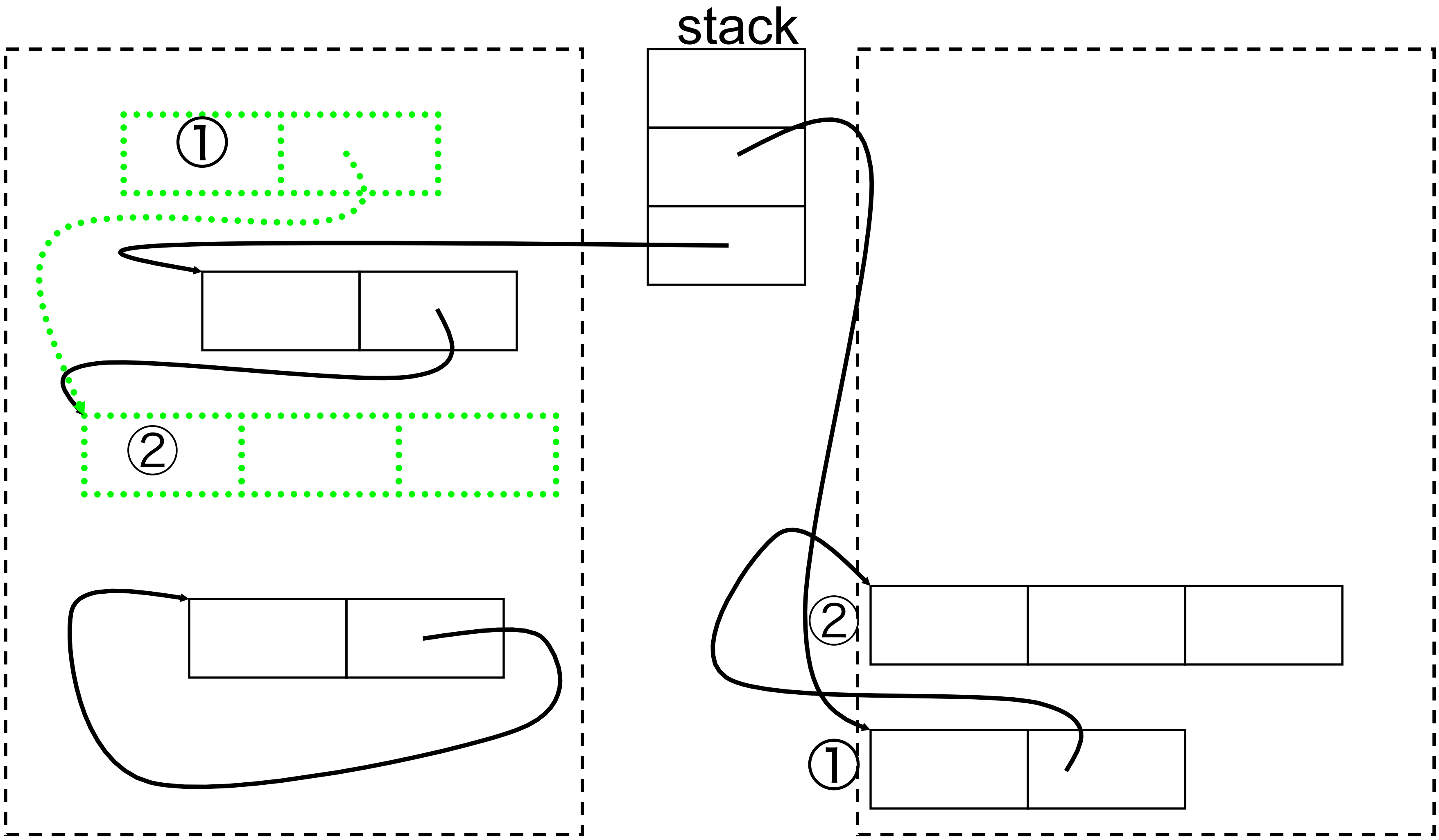
Stop and Copy Example



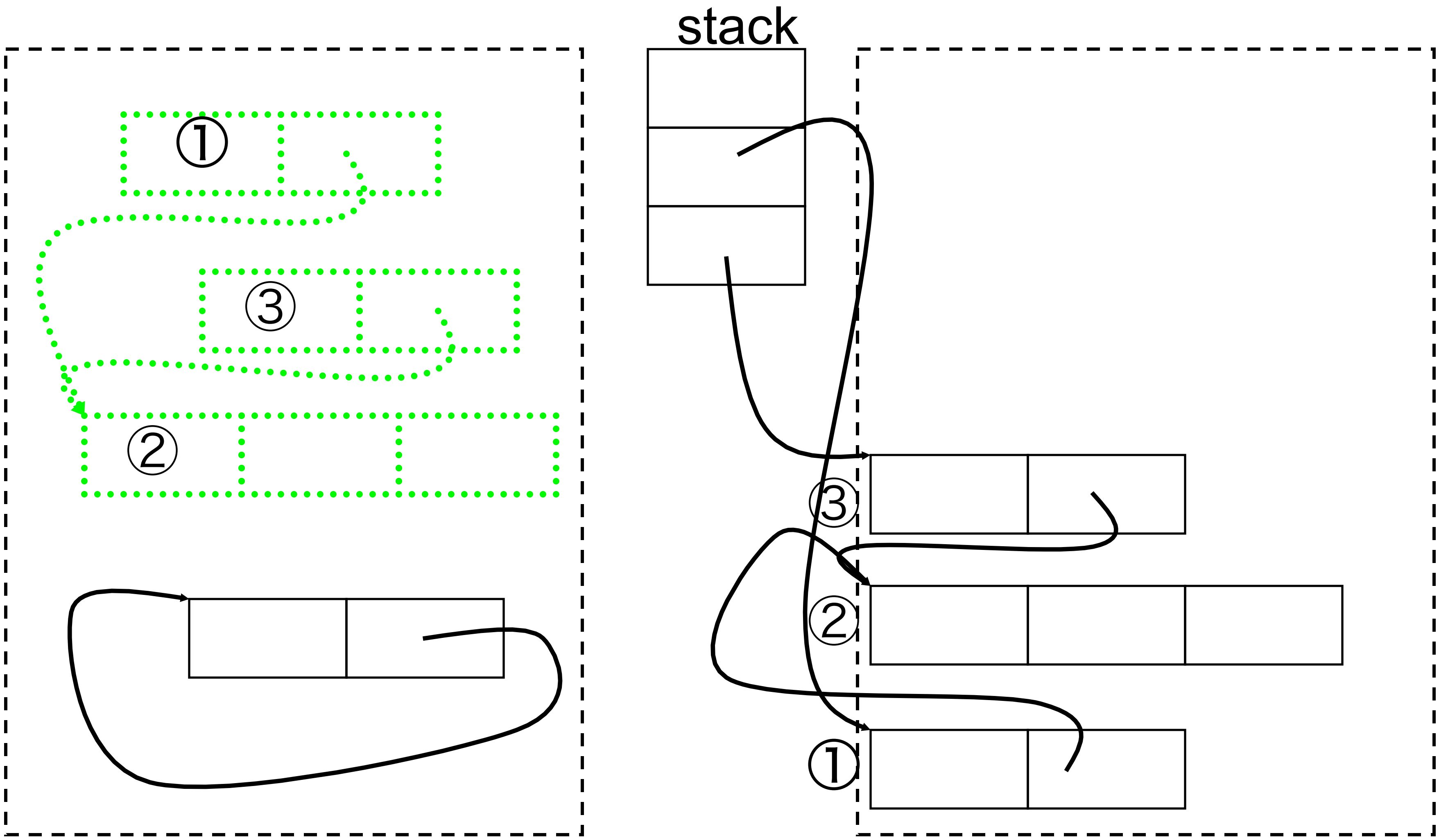
Stop and Copy Example (cont.)



Stop and Copy Example (cont.)



Stop and Copy Example (cont.)

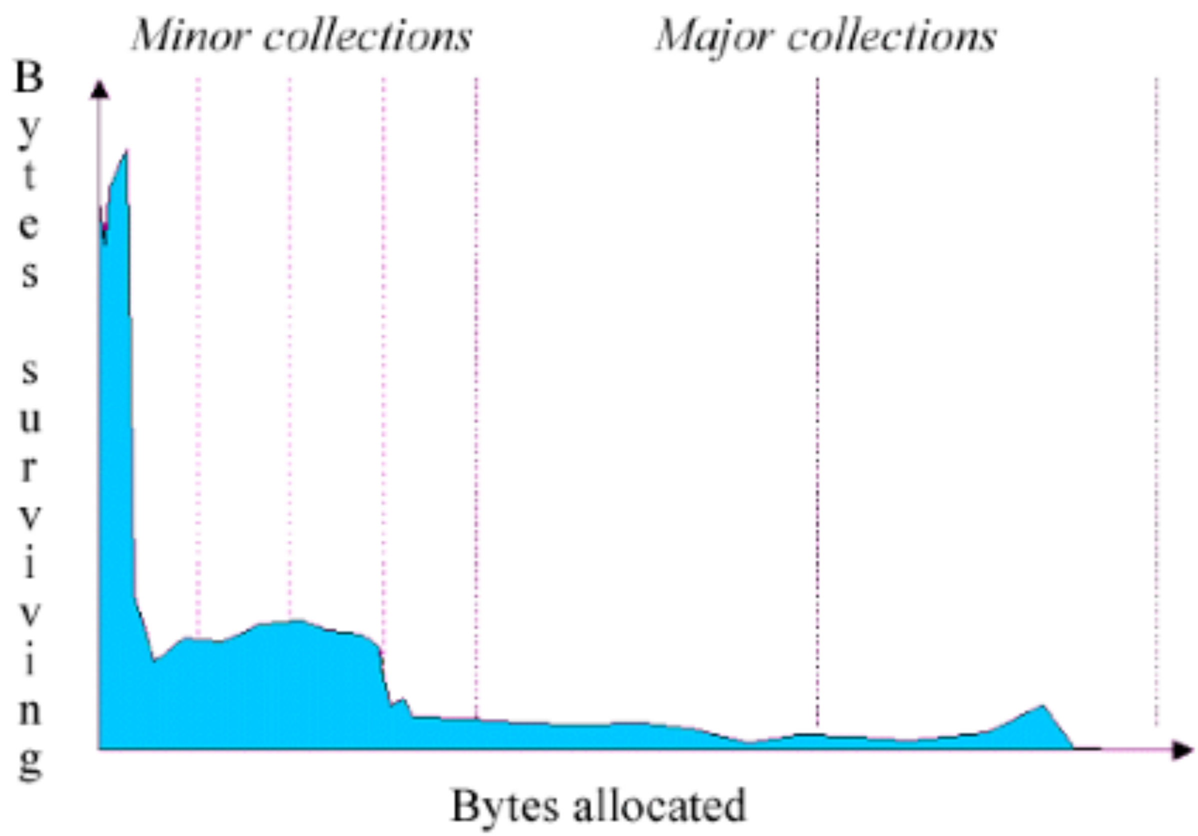


Stop and Copy Tradeoffs

- Advantages
  - Only touches live data
  - No fragmentation (automatically compacts)
    - Will probably increase locality
- Disadvantages
  - Requires twice the memory space
  - Like mark and sweep, need to "stop the world"
    - Program must stop running to let garbage collector move around data in the heap

The Generational Principle

More objects live ⇑



"Young objects die quickly; old objects keep living"

Object lifetime increases ⇒

Generational Collection

- Long lived objects get copied over and over
  - Idea: Have more than one semispace, divide into generations
    - Older generations collected less often
    - Objects that survive many collections get pushed into older generations
    - Need to track pointers from old to young generations to use as roots for young generation collection
- One popular setup
  - Generational stop and copy

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
- Questions
  - Why does using a copy collector for the youngest generation make sense?
  - What apps will be penalized by this setup?

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() {
    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