

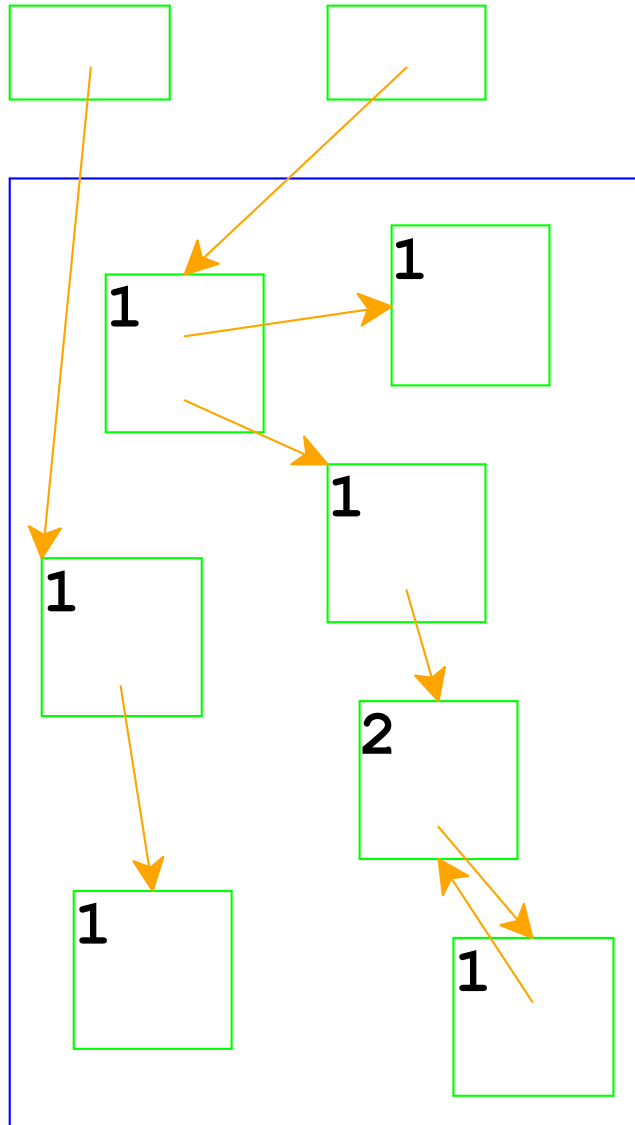
Garbage Collection Techniques

Reference Counting

Reference counting: a way to know whether a record has other users

- Attach a count to every record, starting at 0
- When installing a pointer to a record increment its count
- When replacing a pointer to a record, decrement its count
- When a count reaches 0, decrement counts for other records referenced by the record, then free it

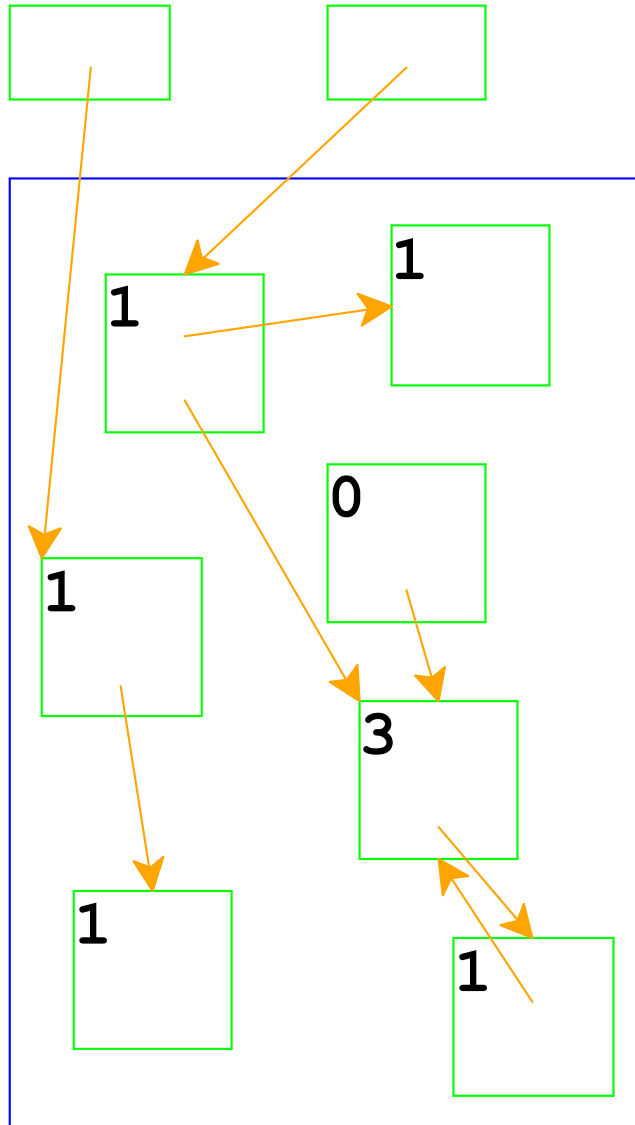
Reference Counting



Top boxes are the roots

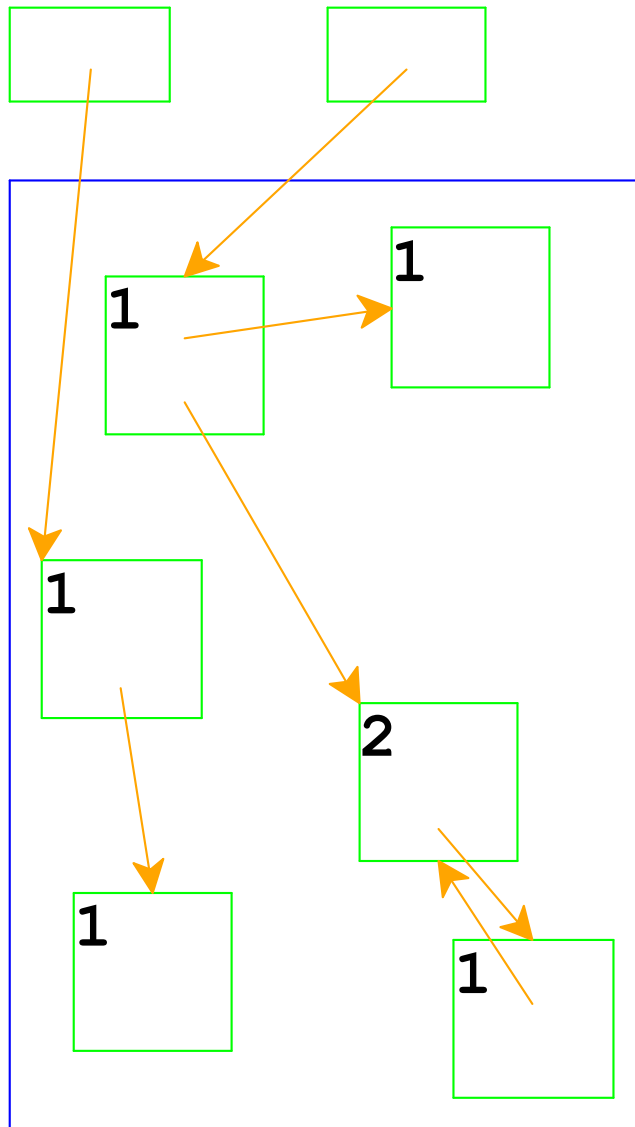
Boxes in the blue area are allocated memory

Reference Counting



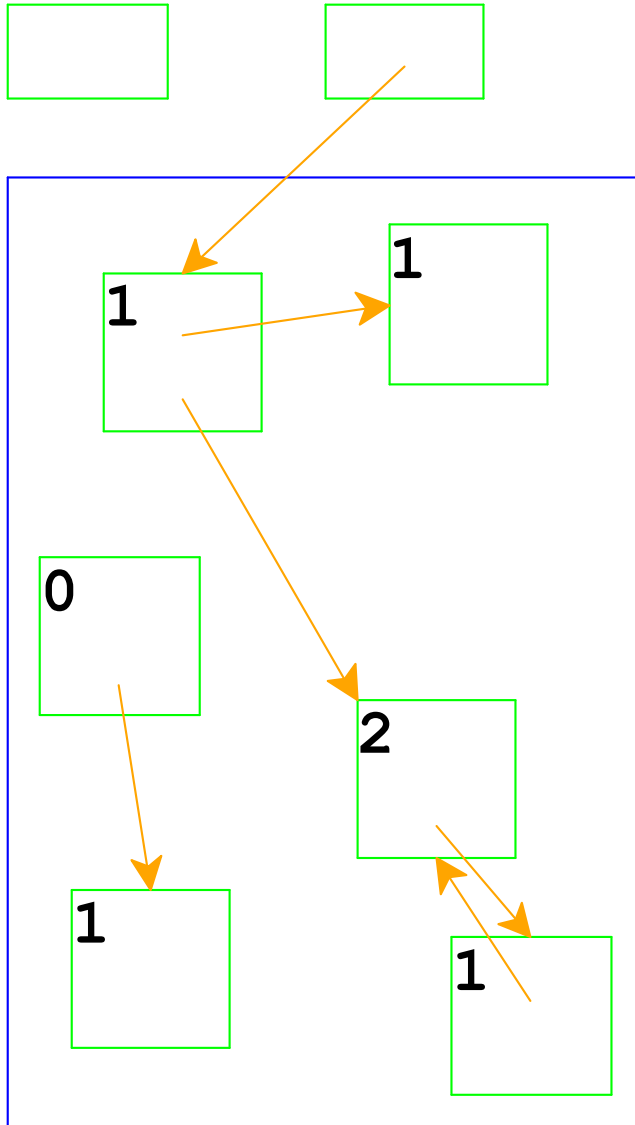
Adjust counts when a pointer is changed...

Reference Counting



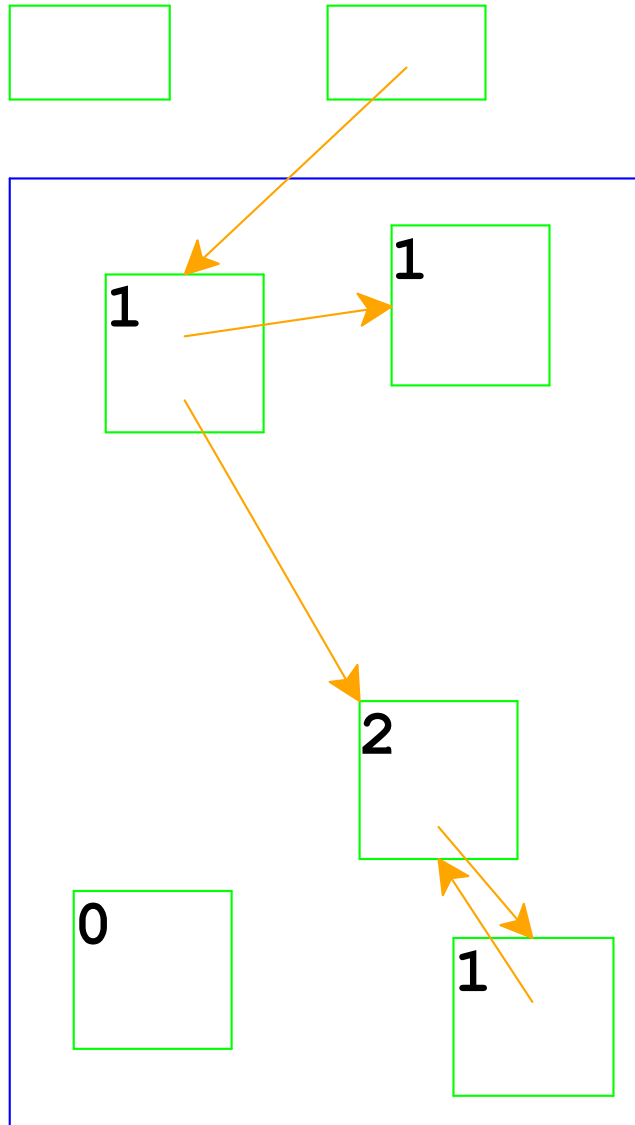
... freeing a record if its count goes to 0

Reference Counting



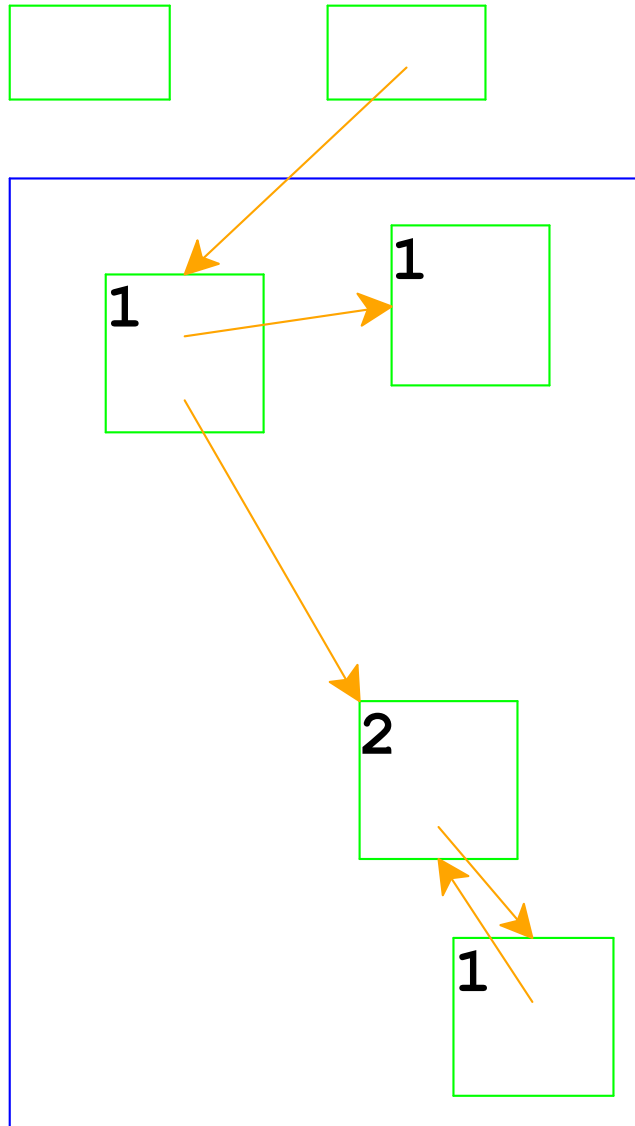
Same if the pointer is a root

Reference Counting



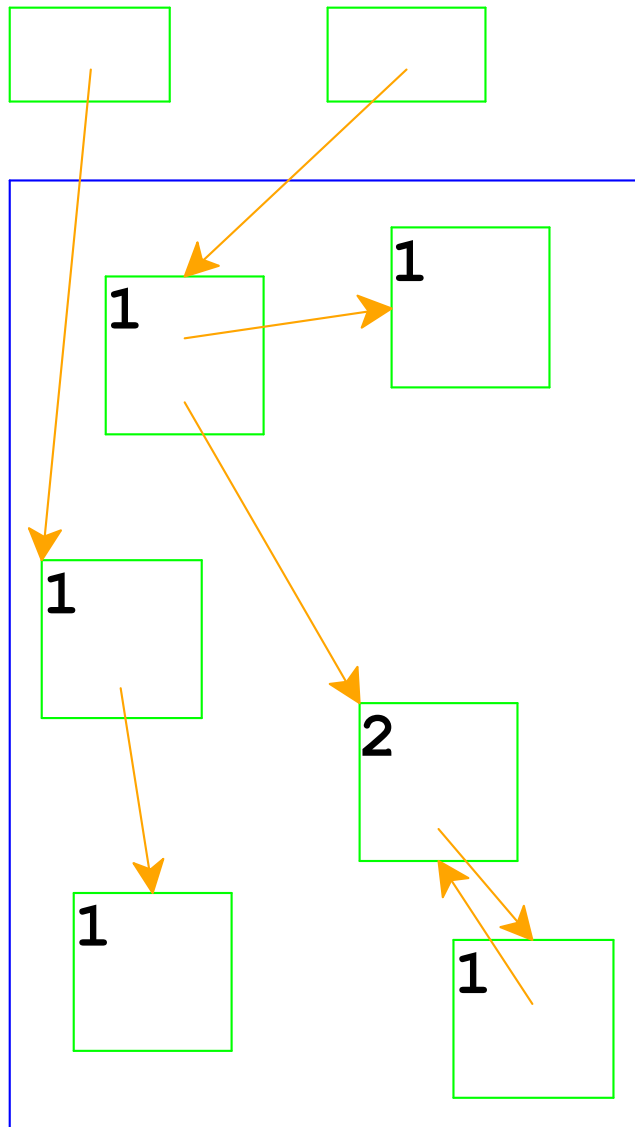
Adjust counts after frees, too...

Reference Counting



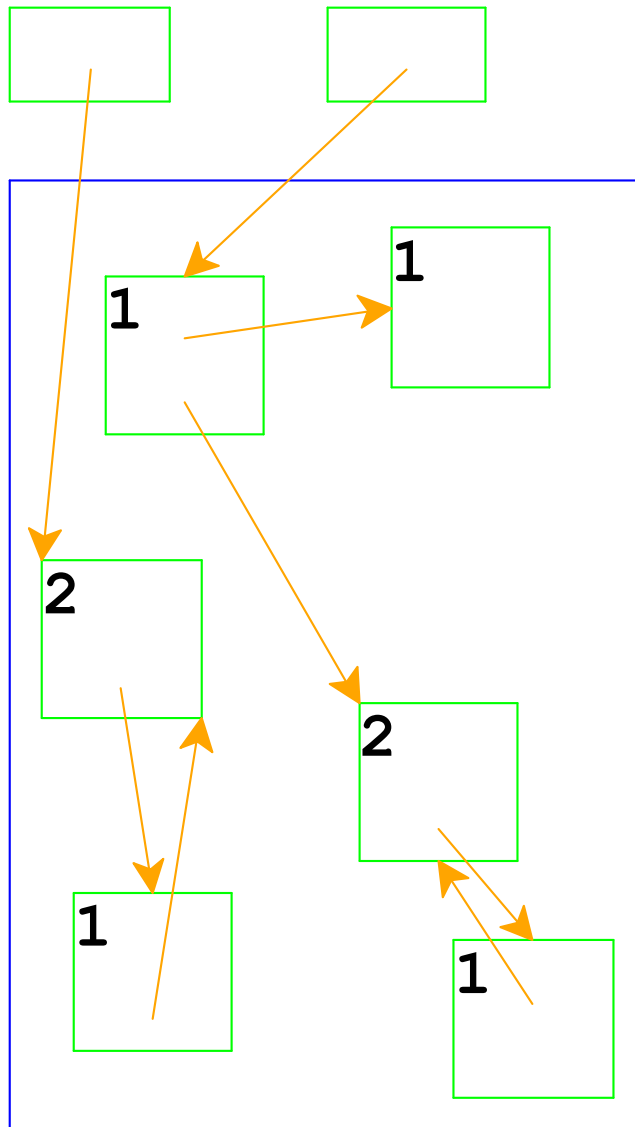
... which can trigger more frees

Reference Counting And Cycles



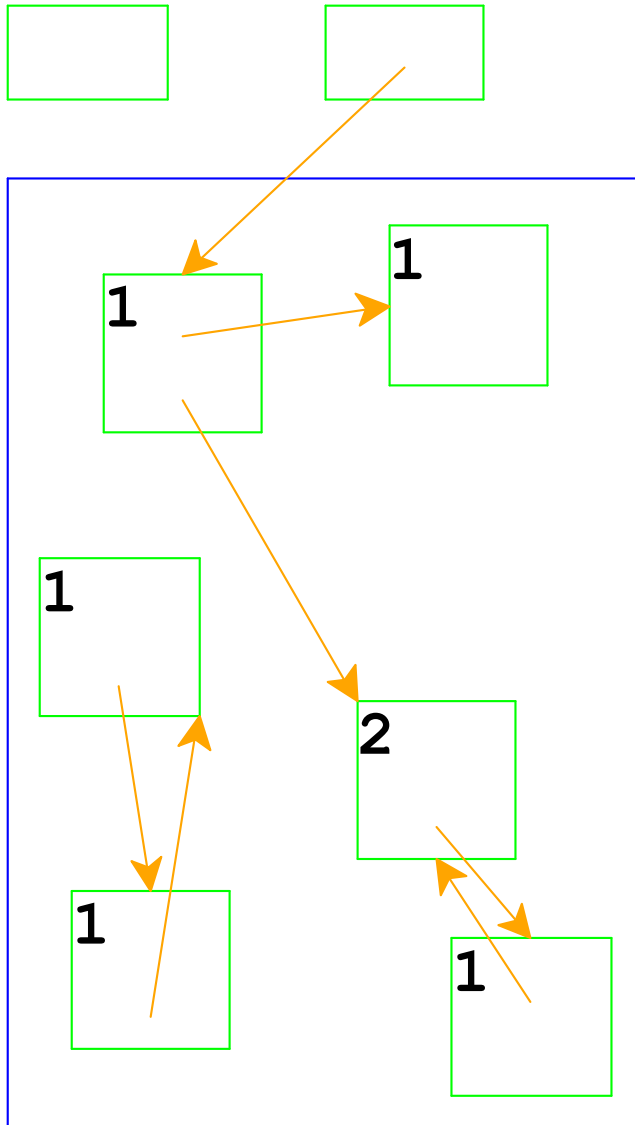
An assignment can create a cycle...

Reference Counting And Cycles



Adding a reference increments a count

Reference Counting And Cycles



Lower-left records are inaccessible, but not deallocated

In general, cycles break reference counting

Reference counting problems

- Cycles don't get collected
- Maintaining counts wastes time & space
- Need to use free lists to track available memory
- Bad locality (and fragmentation!)

Reference counting

But there are times when this is a good choice:

- Objects freed right away (may avoid large pauses)
- Interop with non-GC languages (but see the Boehm collector)

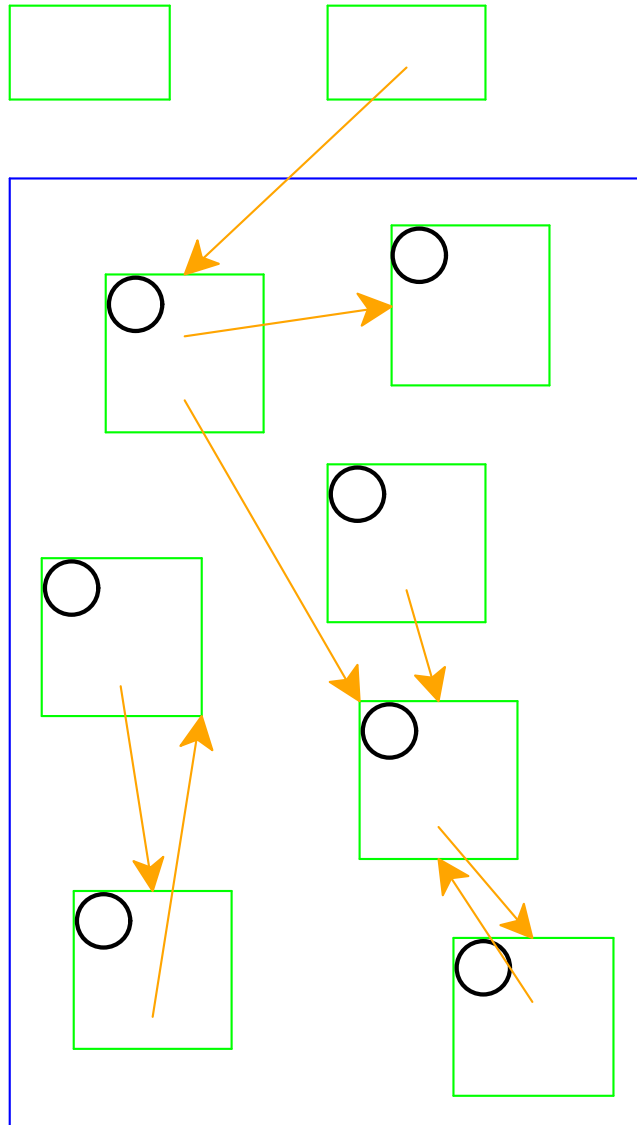
Garbage Collection

- Allocate until we run out of space; then try to free stuff
- **Invariant:** only the PL implementation (runtime system) knows about pointers so we can tag everything and find all reachable data
- Unlike C, C++, assembly;
- Like Python, Java, JavaScript, Racket, ... everything else really

Mark & Sweep Garbage Collection Algorithm

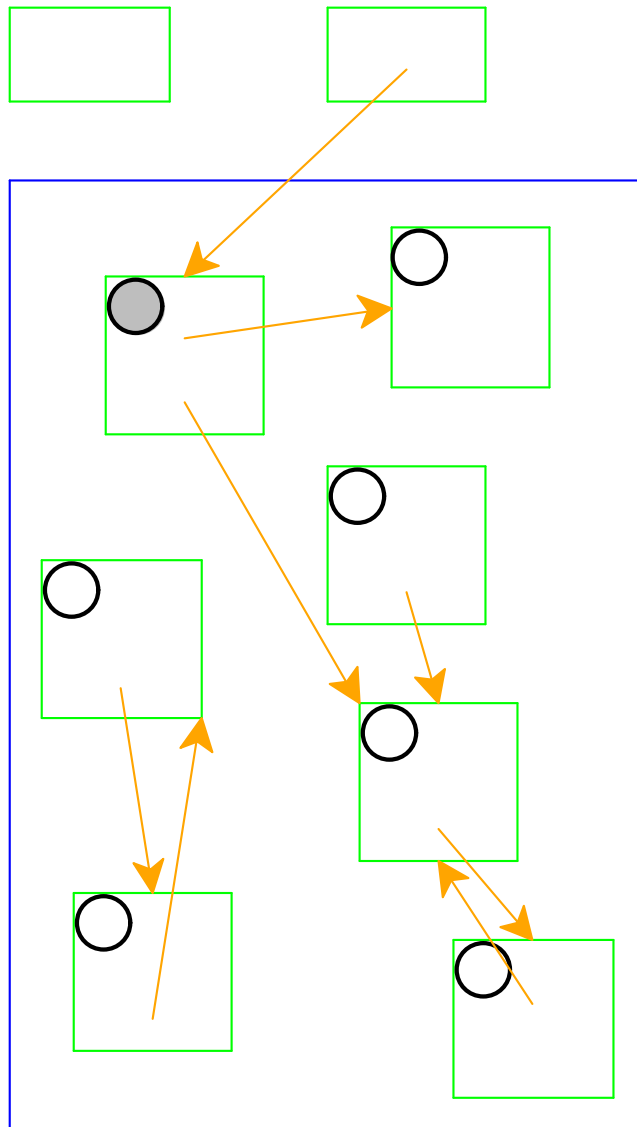
- Color all records **white**
- Color records referenced by roots **gray**
- Repeat until there are no gray records:
 - Pick a gray record, r
 - For each white record that r points to, make it gray
 - Color r **black**
- Deallocate all white records

Mark & Sweep Garbage Collection



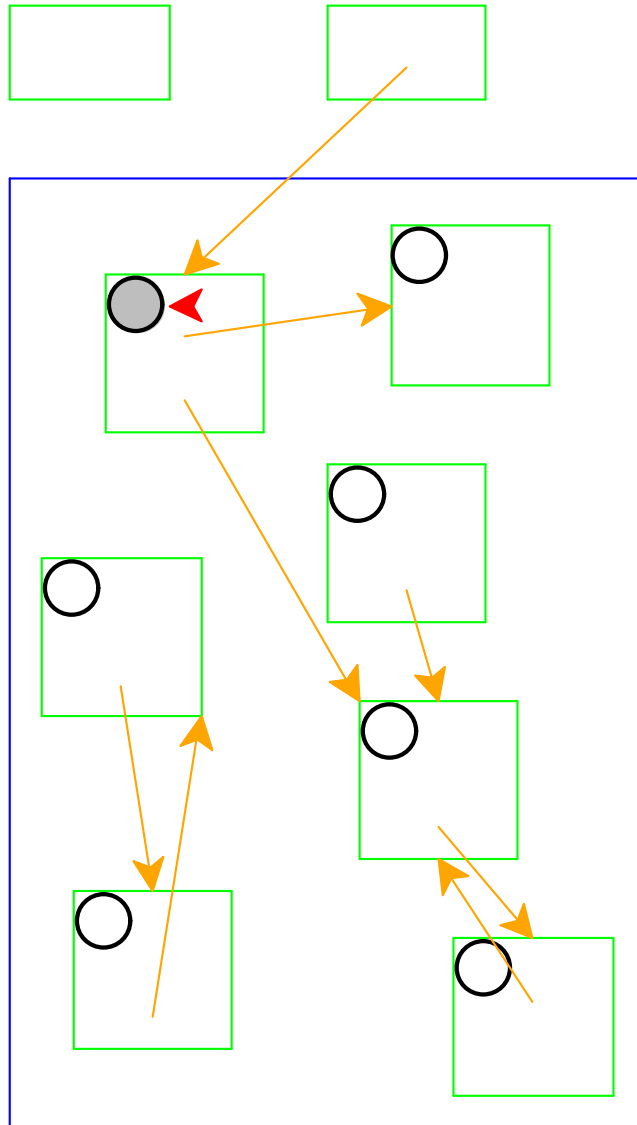
All records are marked white

Mark & Sweep Garbage Collection



Mark records referenced by roots
as gray

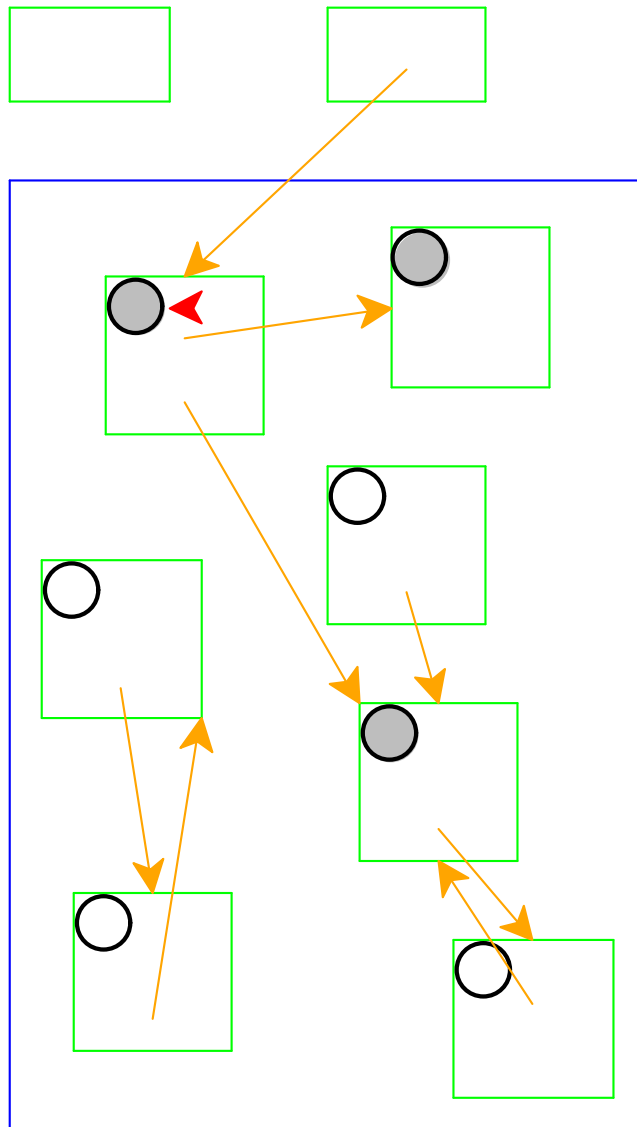
Mark & Sweep Garbage Collection



Need to pick a gray record

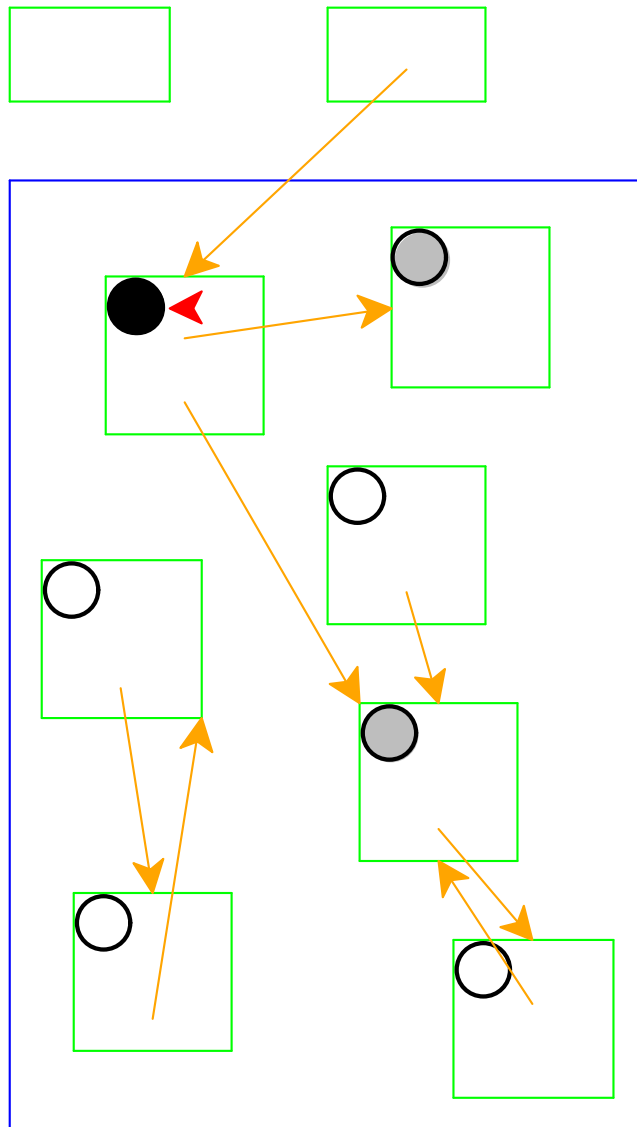
Red arrow indicates the chosen record

Mark & Sweep Garbage Collection



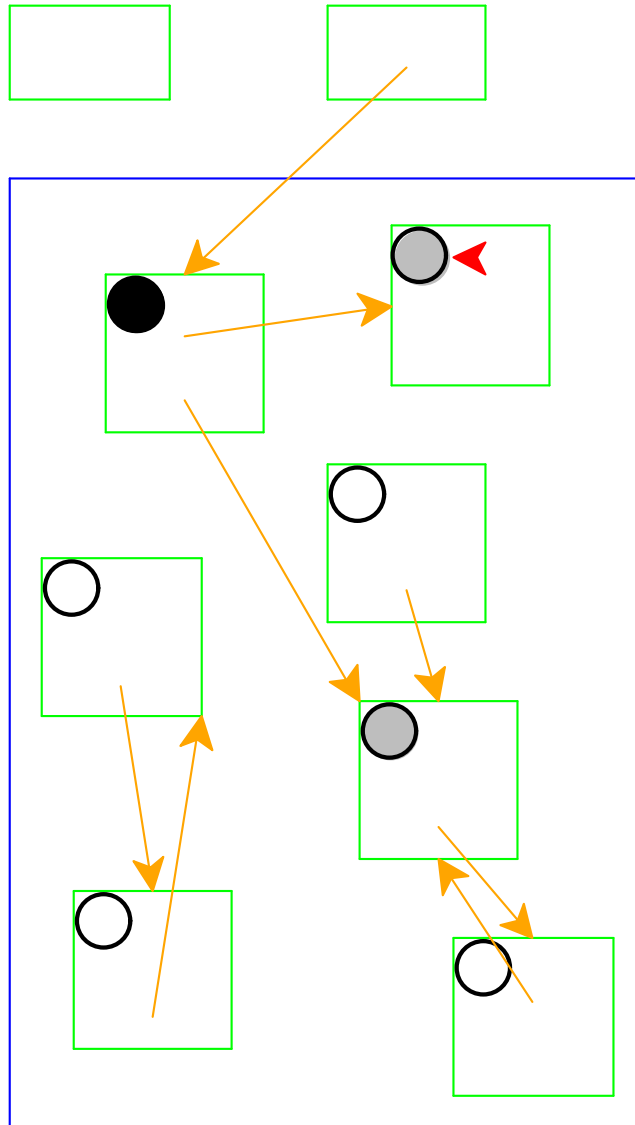
Mark white records referenced by chosen record as gray

Mark & Sweep Garbage Collection



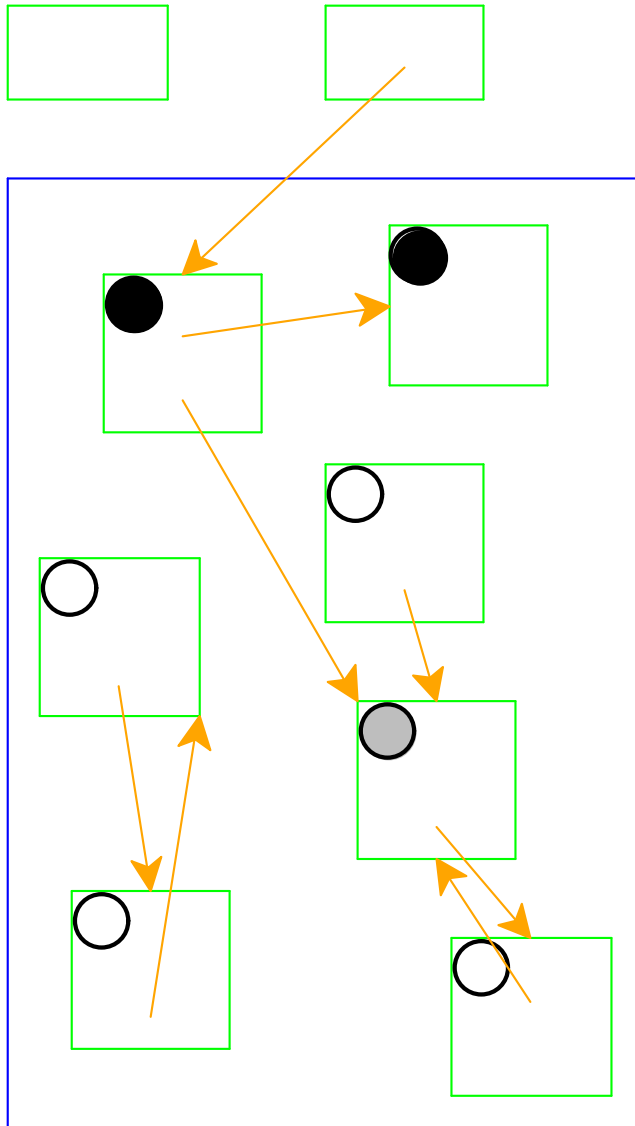
Mark chosen record black

Mark & Sweep Garbage Collection



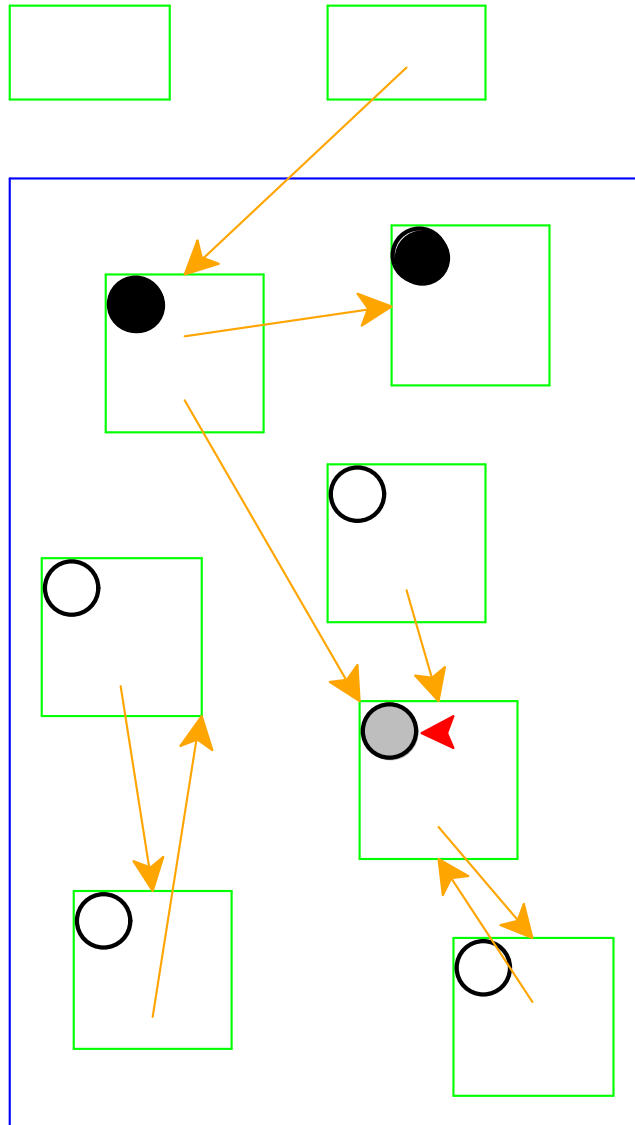
Start again: pick a gray record

Mark & Sweep Garbage Collection



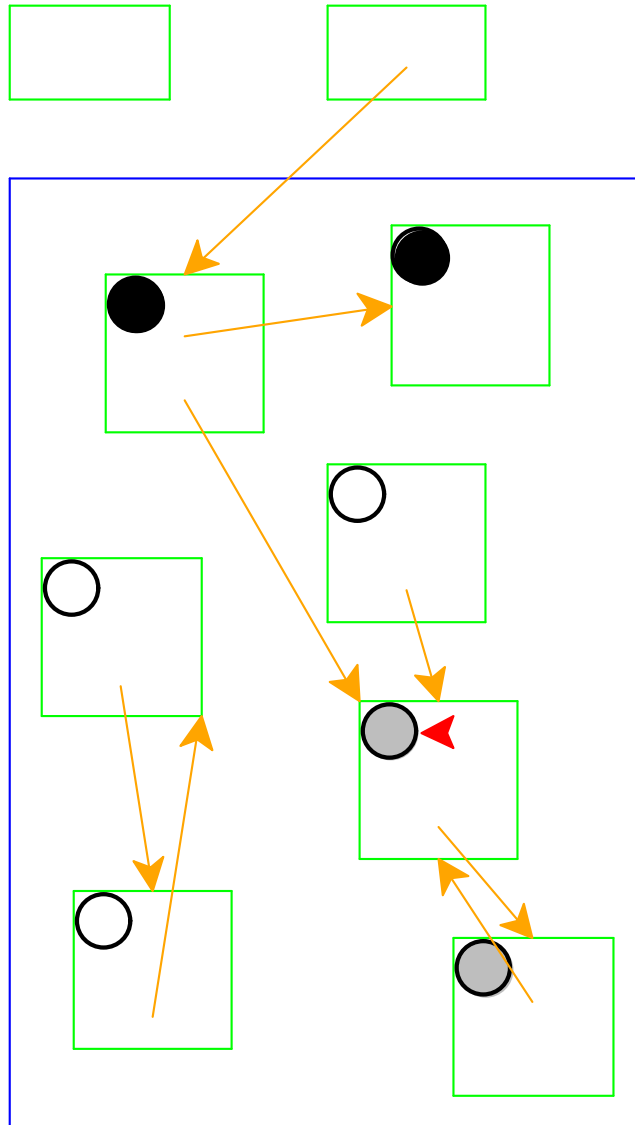
No referenced records; mark black

Mark & Sweep Garbage Collection



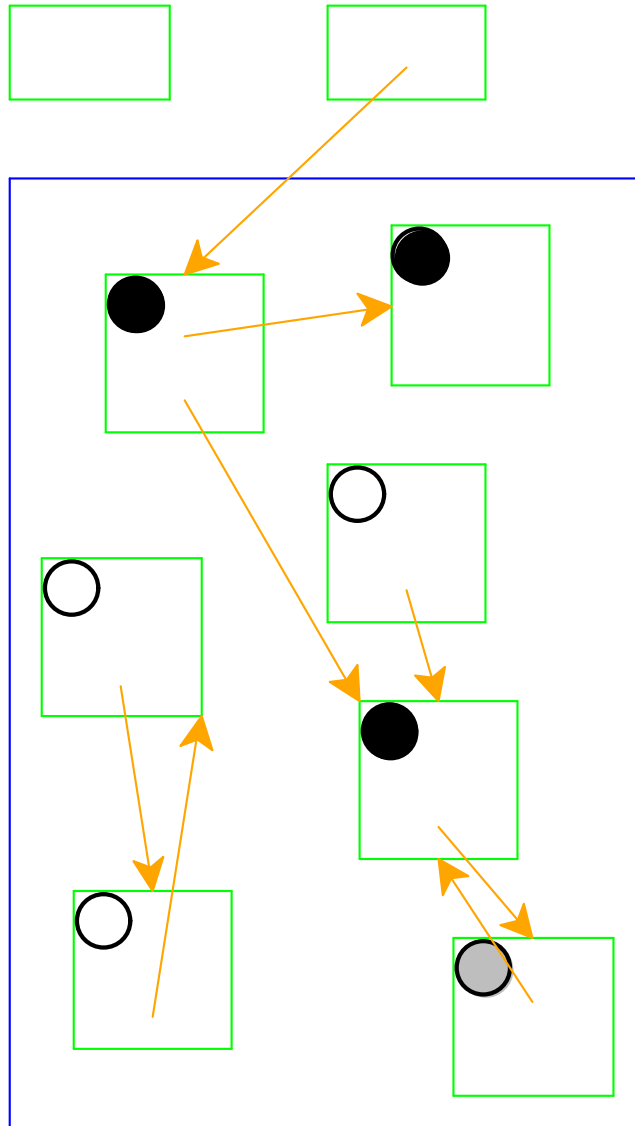
Start again: pick a gray record

Mark & Sweep Garbage Collection



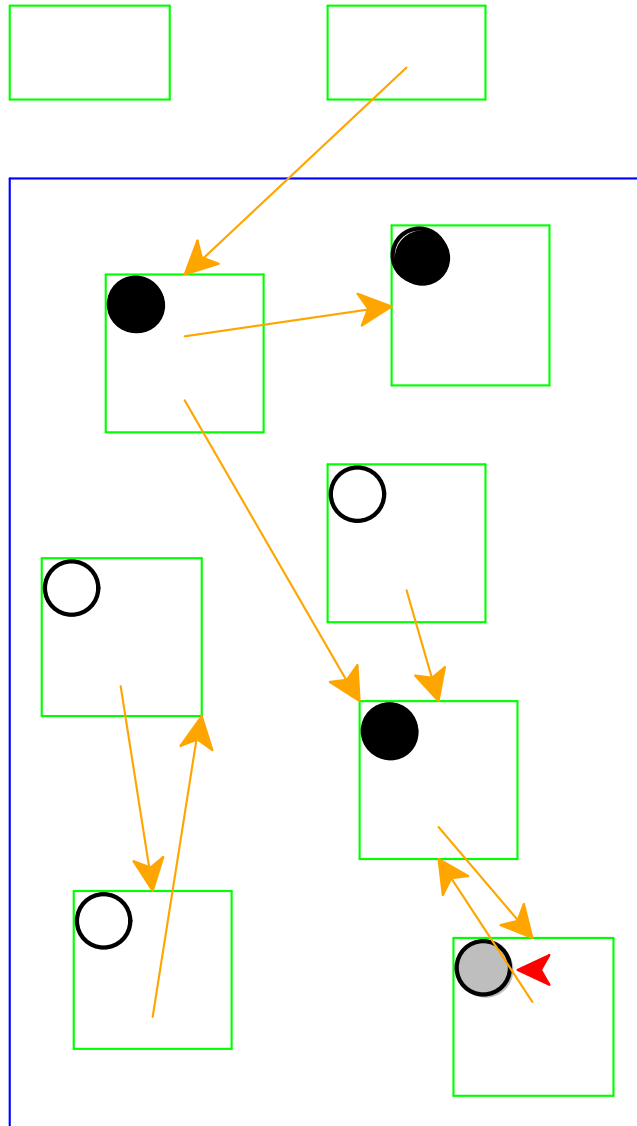
Mark white records referenced by chosen record as gray

Mark & Sweep Garbage Collection



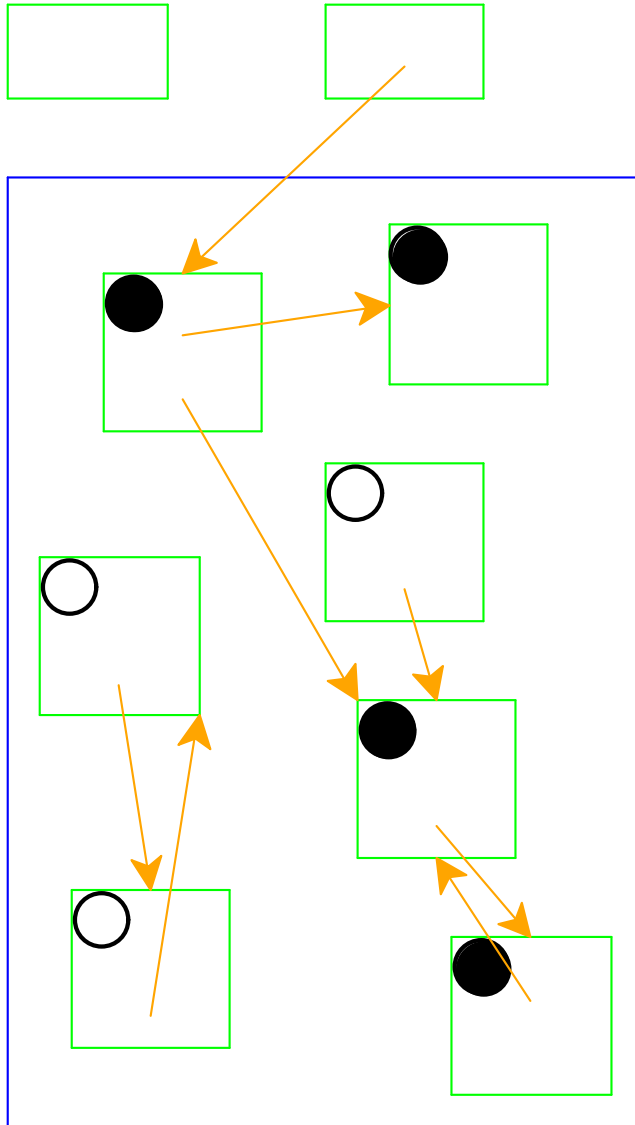
Mark chosen record black

Mark & Sweep Garbage Collection



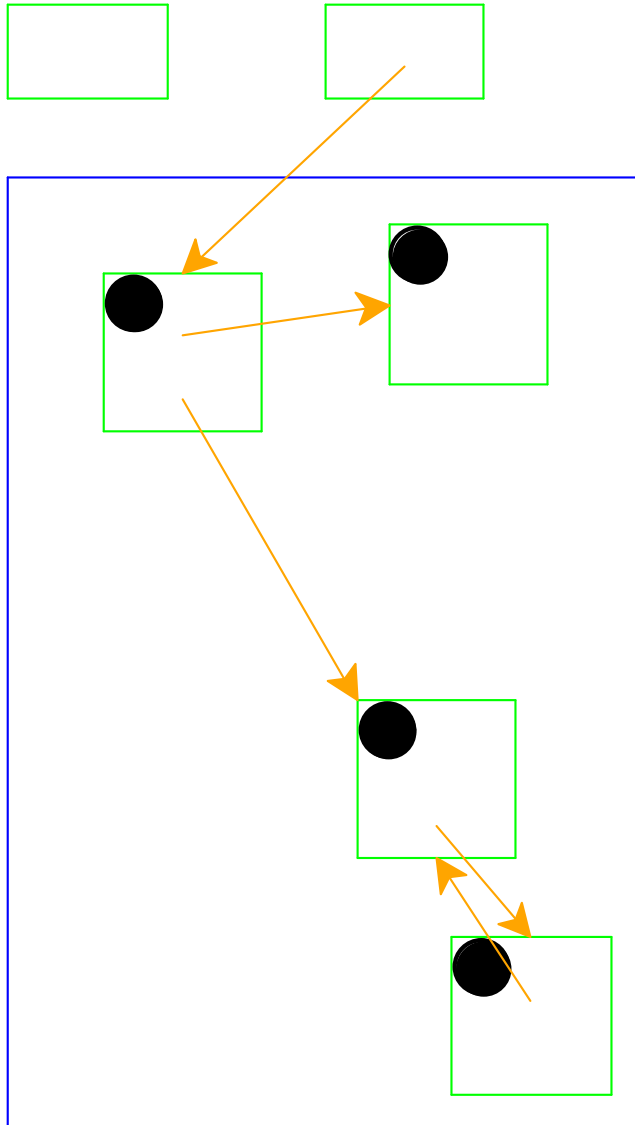
Start again: pick a gray record

Mark & Sweep Garbage Collection



No referenced white records;
mark black

Mark & Sweep Garbage Collection



No more gray records; deallocate white records

Cycles **do not** break garbage collection

Mark & Sweep Garbage Collection

- To turn this into code, we'll use tags to indicate both color and kind of data
- Unqualified tags will denote the black color
- E.g., `'flat` vs `'white-flat` vs `'gray-flat`

Mark and sweep implementation, with linear-time allocator

```
; init-allocator : -> void?
(define (init-allocator)
  (for ([i (in-range 0 (heap-size))])
    (heap-set! i 'free)))

; gc:flat? : location? -> boolean?
(define (gc:flat? addr)
  (equal? (heap-ref addr) 'flat))

; gc:deref location? -> heap-value?
(define (gc:deref addr)
  (unless (gc:flat? addr)
    (error 'gc:flat? "not a flat @ ~a" addr))
  (heap-ref (+ addr 1)))
```

Mark and sweep implementation, with linear-time allocator

```
; gc:cons? : location? -> boolean?
(define (gc:cons? addr)
  (equal? (heap-ref addr) 'cons))

; gc:first : location? -> location?
(define (gc:first addr)
  (unless (gc:cons? addr)
    (error 'gc:first "not a cons @ ~a" addr))
  (heap-ref (+ addr 1)))

; gc:rest : location? -> location?
(define (gc:rest addr)
  (unless (gc:cons? addr)
    (error 'gc:rest "not a cons @ ~a" addr))
  (heap-ref (+ addr 2)))
```

Mark and sweep implementation, with linear-time allocator

```
; gc:set-first! : location? location? -> void?
(define (gc:set-first! addr v)
  (unless (gc:cons? addr)
    (error 'gc:set-first! "not a cons @ ~a" addr))
  (heap-set! (+ addr 1) v))

; gc:set-rest! : location? location? -> void
(define (gc:set-rest! addr v)
  (unless (gc:cons? addr)
    (error 'gc:set-rest! "not a cons @ ~a" addr))
  (heap-set! (+ addr 2) v))
```


Mark and sweep implementation, with linear-time allocator

```
; gc:closure? : location? -> boolean?
(define (gc:closure? addr)
  (equal? (heap-ref addr) 'clos))

; gc:closure-code-ptr : location? -> heap-value?
(define (gc:closure-code-ptr addr)
  (unless (gc:closure? addr)
    (error 'gc:closure-code-ptr "not a closure @ ~a" addr))
  (heap-ref (+ addr 1)))

; gc:closure-env-ref : location? integer? -> location?
(define (gc:closure-env-ref addr i)
  (unless (gc:closure? addr)
    (error 'gc:closure-env-ref "not a closure @ ~a" addr))
  (heap-ref (+ addr 3 i)))
```

Mark and sweep implementation, with linear-time allocator

```
; gc:alloc-flat : heap-value? -> location?  
(define (gc:alloc-flat v)  
  (define address (alloc 2 #f #f))  
  (heap-set! address 'flat)  
  (heap-set! (+ 1 address) v)  
  address)
```

```
; gc:cons : root? root? -> location?  
(define (gc:cons v1 v2)  
  (define address (alloc 3 v1 v2))  
  (heap-set! address 'cons)  
  (heap-set! (+ address 1) (read-root v1))  
  (heap-set! (+ address 2) (read-root v2))  
  address)
```

Mark and sweep implementation, with linear-time allocator

```
; gc:closure : heap-value? (vectorof location?)  
;           -> location?  
(define (gc:closure code-ptr free-variables)  
  (define address  
    (alloc (+ 3 (length free-variables))  
           free-variables #f))  
  (heap-set! address 'clos)  
  (heap-set! (+ address 1) code-ptr)  
  (heap-set! (+ address 2) (length free-variables))  
  (for ([i (in-range 0 (length free-variables))]  
        [f (in-list free-variables)]))  
    (heap-set! (+ address 3 i) (read-root f)))  
  address)
```

Mark and sweep implementation, with linear-time allocator

```
; a roots is either:  
; - root?  
; - location?  
; - (listof roots?)
```

```
; alloc : number? roots? roots? -> location?  
(define (alloc n roots1 roots2)  
  (define a (find-free-space 0 n))  
  (cond [a  
        a]  
        [else  
         (collect-garbage roots1 roots2)  
         (define a (find-free-space 0 n))  
         (unless a  
           (error 'alloc "out of memory"))  
         a])))
```

Mark and sweep implementation, with linear-time allocator

```
; find-free-space : location? number?
;                  -> (or/c location? #f)
(define (find-free-space start n)
  (cond [(= start (heap-size))
        #f]
        [else
         (case (heap-ref start)
             [(flat) (find-free-space (+ start 2) n)]
             [(cons) (find-free-space (+ start 3) n)]
             [(clos) (find-free-space
                      (+ start 3 (heap-ref (+ start 2)))] n)]
             [(free) (if (n-free-blocks? start n)
                         start
                         (find-free-space (+ start 1) n))]
             [else (error 'find-free-space
                          "unexpected tag ~a" start)]]))])
```

Mark and sweep implementation, with linear-time allocator

```
; n-free-blocks? : location? integer? -> boolean?  
(define (n-free-blocks? start n)  
  (cond [(= n 0) #t]  
        [(= start (heap-size)) #f]  
        [else (and (equal? (heap-ref start) 'free)  
                    (n-free-blocks?  
                      (+ start 1) (- n 1))))]))
```

Mark and sweep implementation, with linear-time allocator

Note: this allocator is ridiculously inefficient, in more ways than one.

A proper mark-and-sweep garbage collector would use a free list, as with reference counting (or C's malloc, for that matter).

All your favorite tricks from the malloc lab apply.

Mark and sweep implementation, with linear-time allocator

```
; collect-garbage : roots? roots? -> void?  
(define (collect-garbage roots1 roots2)  
  (validate-heap)  
  (mark-white!)  
  (traverse/roots (get-root-set))  
  (traverse/roots roots1)  
  (traverse/roots roots2)  
  (free-white!)  
  (validate-heap))
```


Mark and sweep implementation, with linear-time allocator

```
; validate-heap : -> void?
(define (validate-heap)
  (define (valid-pointer? p)
    (unless (< p (heap-size))
      (error 'validate-heap "pointer out of bounds ~a" p))
    (unless (member (heap-ref p) '(flat cons clos))
      (error 'validate-heap "pointer to non-tag ~a" p)))
  (let loop ([i 0])
    (when (< i (heap-size))
      (case (heap-ref i)
        [(flat) (loop (+ i 2))]
        [(cons)
         (valid-pointer? (heap-ref (+ i 1)))
         (valid-pointer? (heap-ref (+ i 2)))
         (loop (+ i 3))]
        [(clos)
         (for ([j (in-range 0 (heap-ref (+ i 2)))]])
           (valid-pointer? (heap-ref (+ i 3 j))))
         (loop (+ i 3 (heap-ref (+ i 2))))]
        [(free) (loop (+ i 1))]
        [else (error 'validate-heap "unexpected tag @ ~a" i)]))))))
```

Mark and sweep implementation, with linear-time allocator

```
; mark-white! : -> void?
(define (mark-white!)
  (let loop ([i 0])
    (when (< i (heap-size))
      (case (heap-ref i)
        [(cons)
         (heap-set! i 'white-cons)
         (loop (+ i 3))]
        [(flat)
         (heap-set! i 'white-flat)
         (loop (+ i 2))]
        [(clos)
         (heap-set! i 'white-clos)
         (loop (+ i 3 (heap-ref (+ i 2)))))]
        [(free)
         (loop (+ i 1))]
        [else (error 'mark-white!
                     "unexpected tag @ ~a" i)]))))
```

Mark and sweep implementation, with linear-time allocator

```
; free-white! : -> void?
(define (free-white!)
  (let loop ([i 0])
    (when (< i (heap-size))
      (case (heap-ref i)
        [(cons)      (loop (+ i 3))]
        [(flat)     (loop (+ i 2))]
        [(clos)     (loop (+ i 3 (heap-ref (+ i 2))))]
        [(free)     (loop (+ i 1))]
        [(white-flat) (heap-set! i 'free)
                    (heap-set! (+ i 1) 'free)
                    (loop (+ i 2))]
        [(white-cons) (heap-set! i 'free)
                       (heap-set! (+ i 1) 'free)
                       (heap-set! (+ i 2) 'free)
                       (loop (+ i 3))]
        [(white-clos) (heap-set! i 'free)
                       (heap-set! (+ i 1) 'free)
                       (define size (heap-ref (+ i 2)))
                       (for ([x (in-range 0 size)])
                         (heap-set! (+ i 3 x) 'free))
                       (heap-set! (+ i 2) 'free)
                       (loop (+ i 3 size))]
        [(free)     (loop (+ i 1))]
        [else (error 'free-white! "unexpected tag @ ~a" i)]))))
```

Mark and sweep implementation, with linear-time allocator

```
; traverse/roots : roots? -> void?
(define (traverse/roots roots)
  (cond [(list? roots)
        (for-each traverse/roots roots)]
        [(root? roots)
        (traverse/loc (read-root roots))]
        [(false? roots)
        (void)]
        [else
        (error 'traverse/roots
              "unexpected roots: ~a" roots)]))
```

Mark and sweep implementation, with linear-time allocator

```
; traverse/loc : location? -> void?
(define (traverse/loc loc)
  (case (heap-ref loc)
    [(flat gray-flat) (void)]
    [(cons gray-cons) (void)]
    [(clos gray-clos) (void)]
    [(white-flat)
     ; can skip gray
     (heap-set! loc 'flat)]
    [(white-cons)
     (heap-set! loc 'gray-cons)
     (traverse/loc (heap-ref (+ loc 1)))
     (traverse/loc (heap-ref (+ loc 2)))
     (heap-set! loc 'cons)]
    [(white-clos)
     (heap-set! loc 'gray-clos)
     (for ([i (in-range 0 (heap-ref (+ loc 2)))]])
       (traverse/loc (heap-ref (+ loc i 3))))
     (heap-set! loc 'clos)]
    [else (error 'traverse/loc "unexpected tag @ ~a" loc)]))
```

Mark & Sweep Problems

- Cost of collection proportional to (entire) heap
- Bad locality (and fragmentation!)
- Need to use free lists to track available memory

(But there are times when this is a good choice)

- Our M&S also has a terrible allocator on top of that
 - Can do better with free lists, etc., see malloc lab