Garbage Collection

Today: various garbage collection strategies; basic ideas:

- 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++, asm; like ruby, python, perl, java, racket, ... everything else really)

Reference Counting

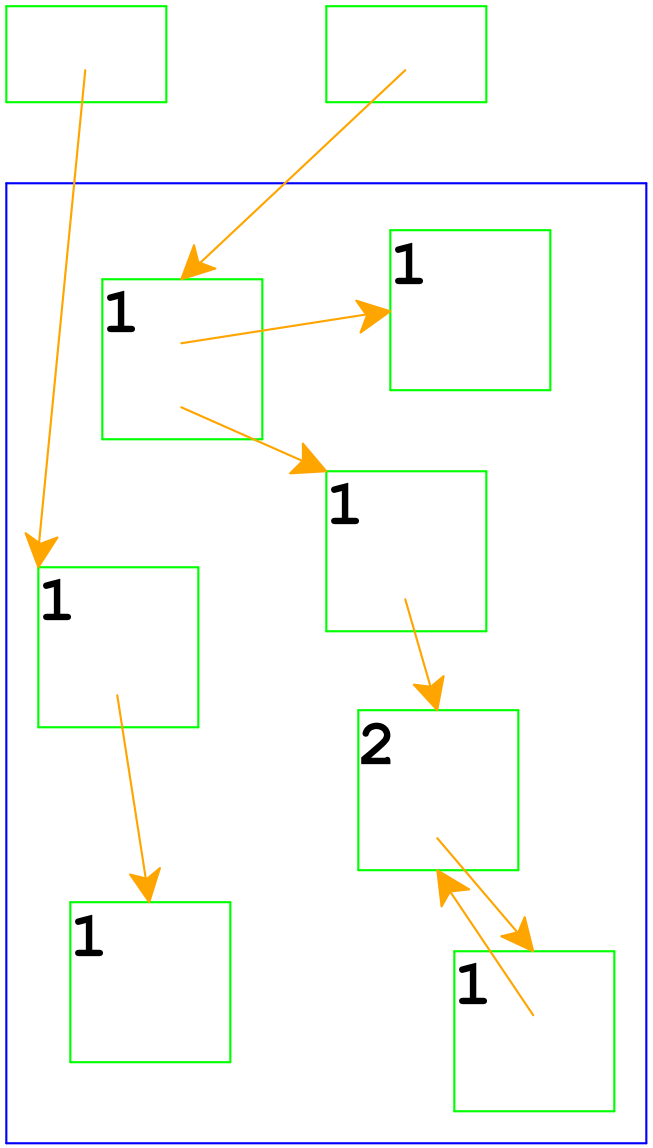
Reference counting: a way to know whether a record has other users

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

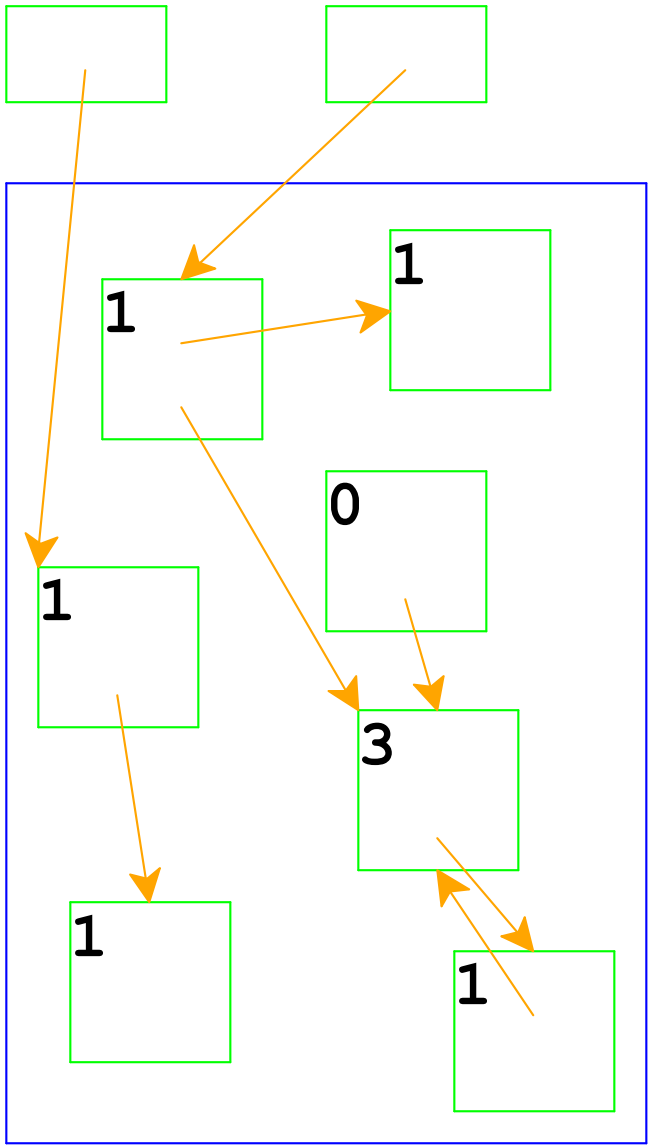
Reference Counting



Top boxes are the roots

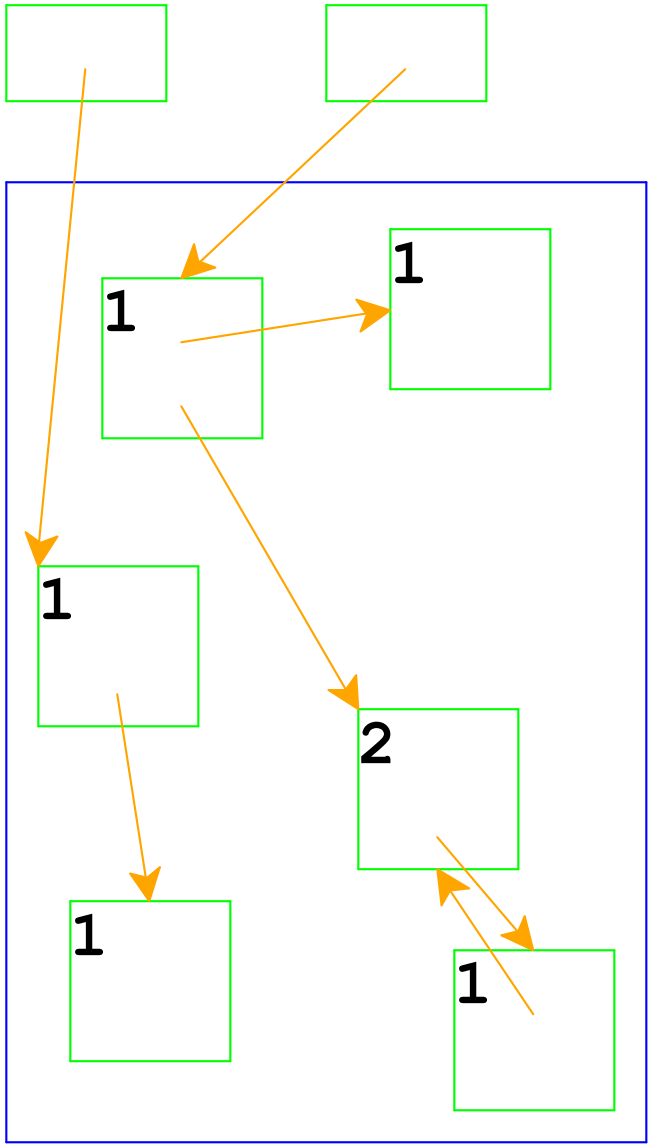
Boxes in the blue area are allocated memory

Reference Counting



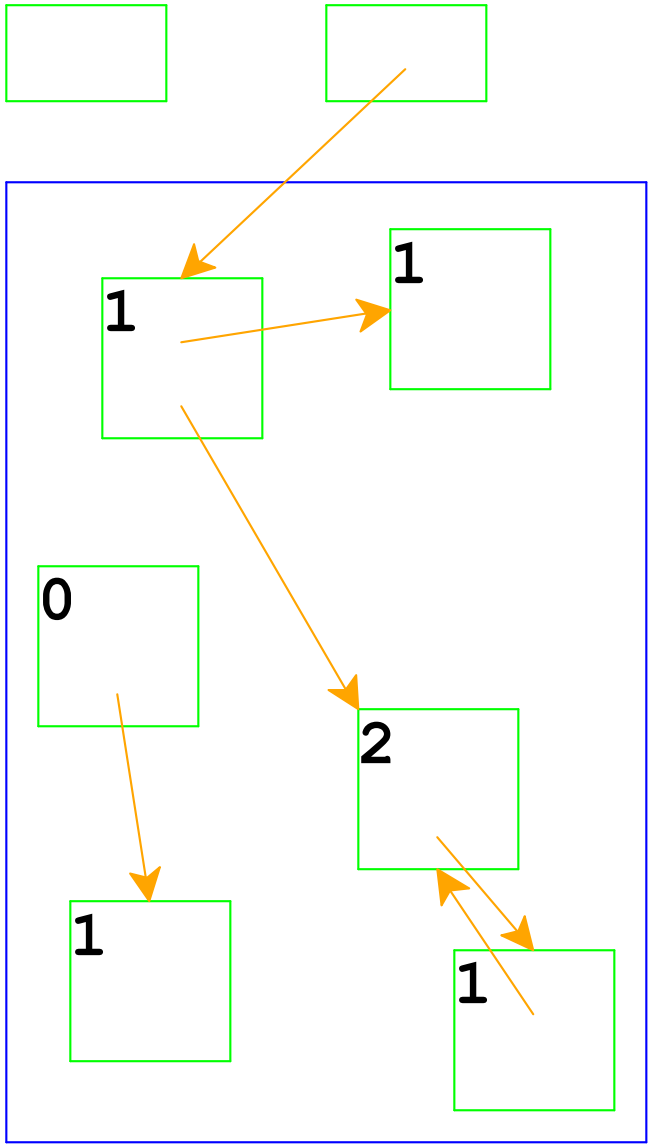
Adjust counts when a pointer is changed...

Reference Counting



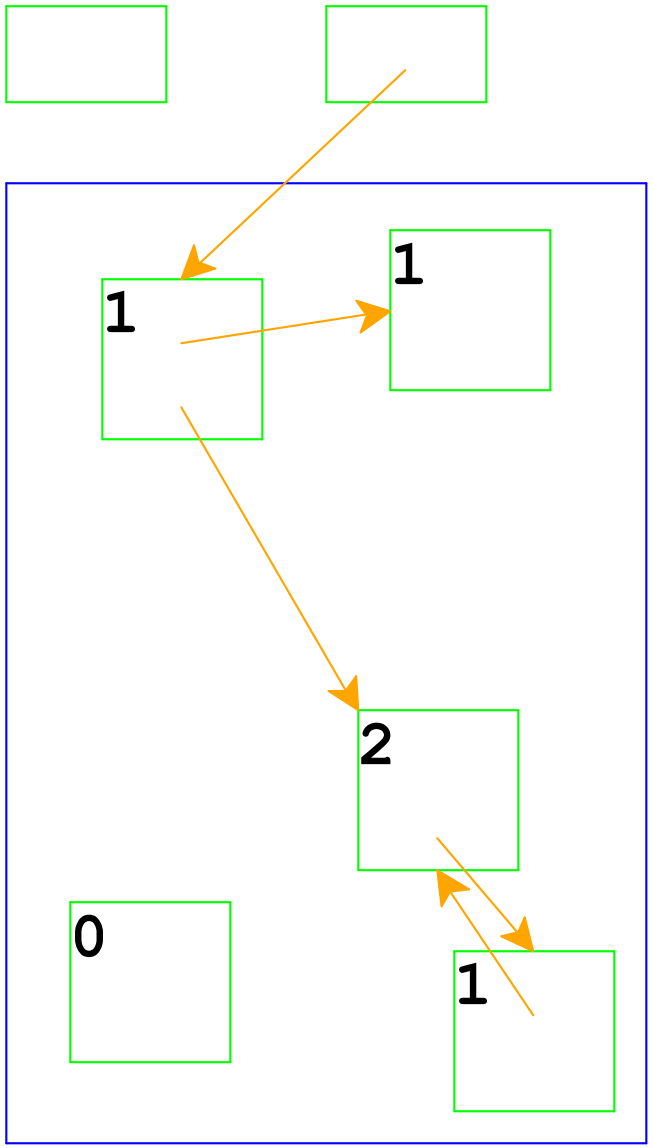
... freeing a record if its count goes to 0

Reference Counting



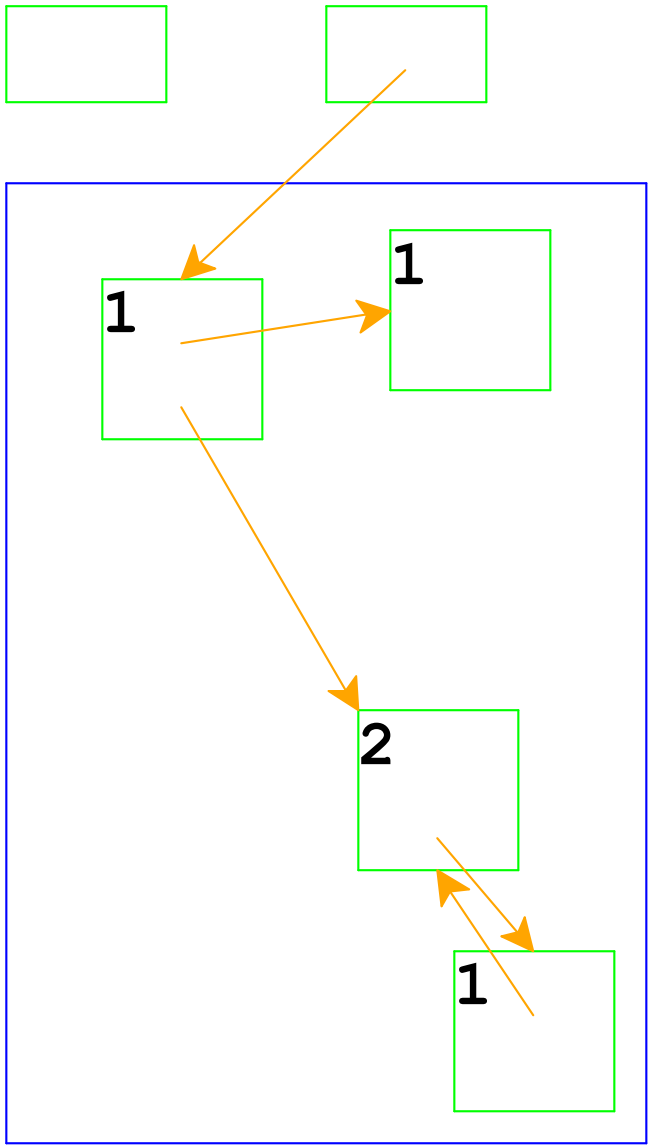
Same if the pointer is a root

Reference Counting



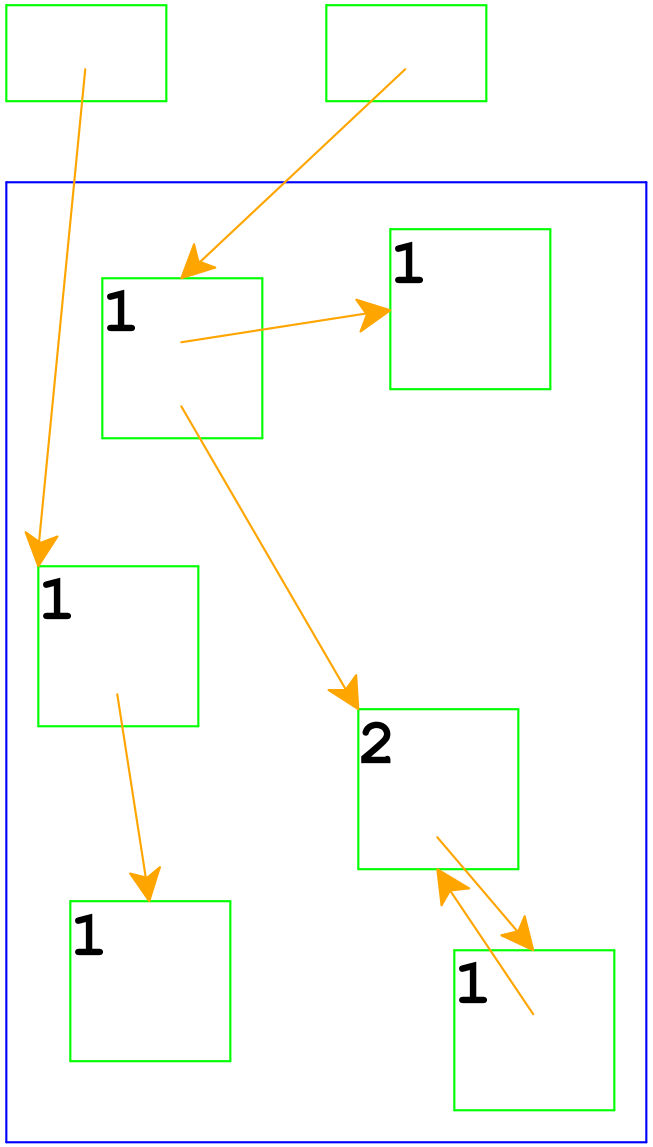
Adjust counts after frees, too...

Reference Counting



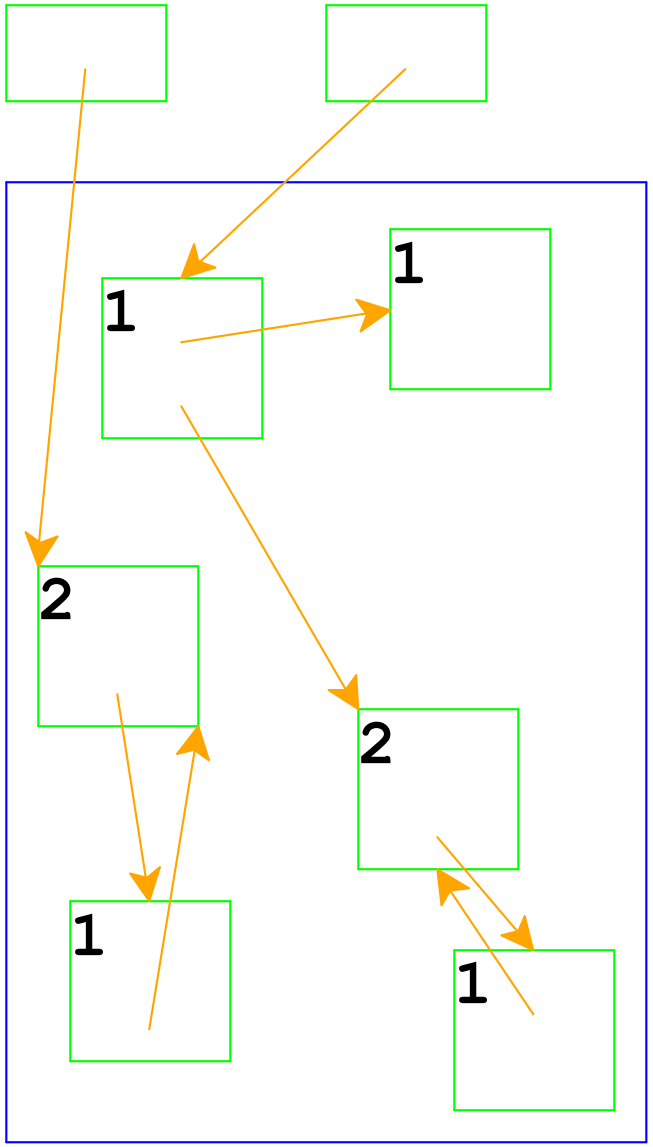
... which can trigger more frees

Reference Counting And Cycles



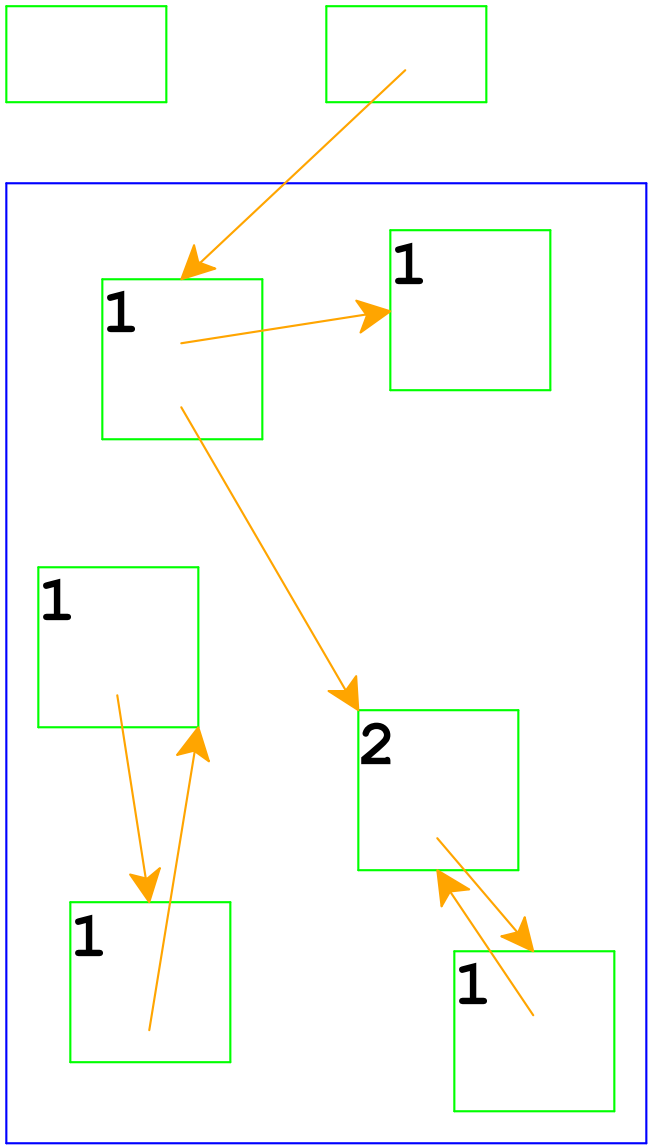
An assignment can create a cycle...

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

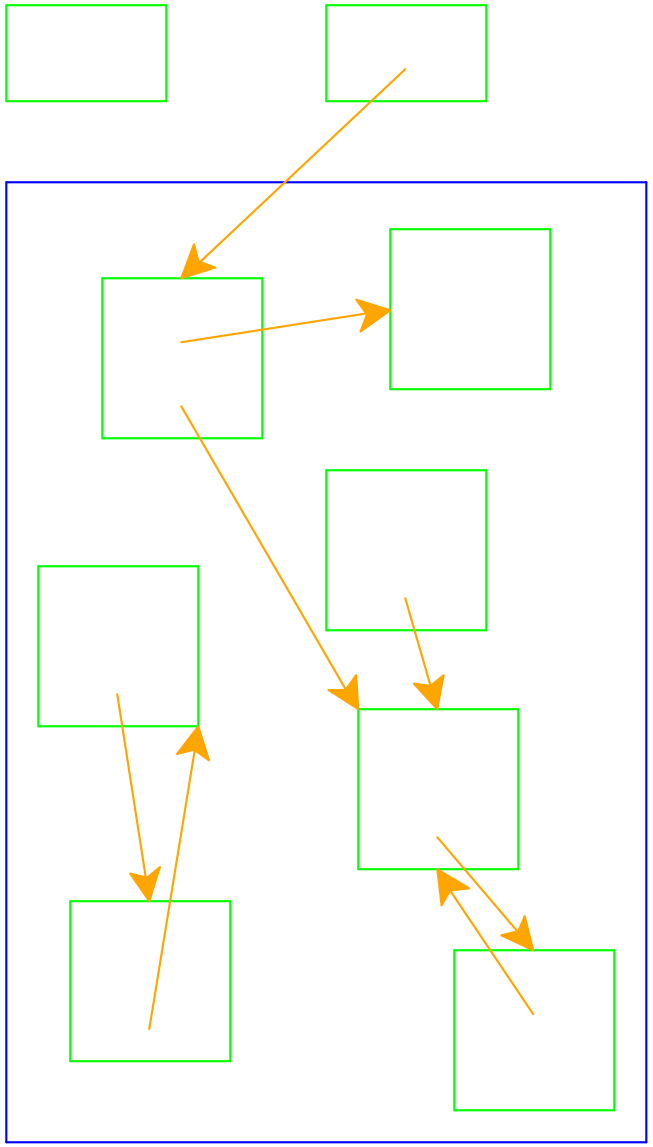
- Cycles
- Maintaining counts wastes time & space
- Need to use free lists to track available memory

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

Mark & Sweep Garbage Collection Algorithm

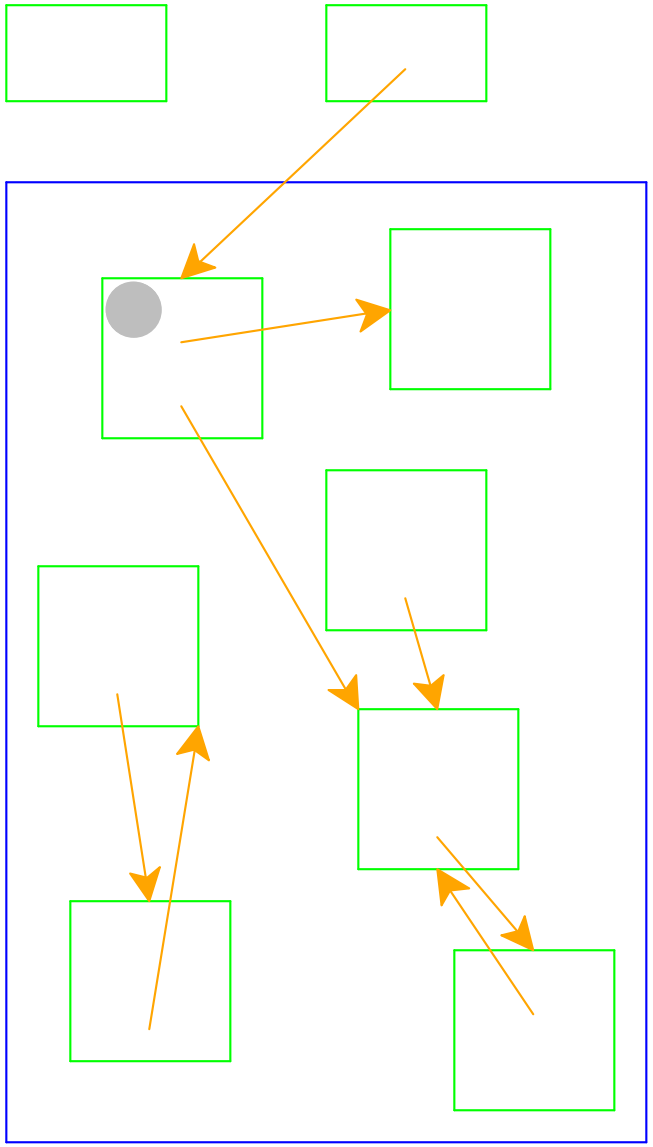
- Color all records ***white***
- Color records referenced by roots ***gray***
- Repeat until there are no gray records:
  - Pick a gray record, *r*
  - For each white record that *r* points to, make it gray
  - Color *r* ***black***
- Deallocate all white records

Mark & Sweep Garbage Collection



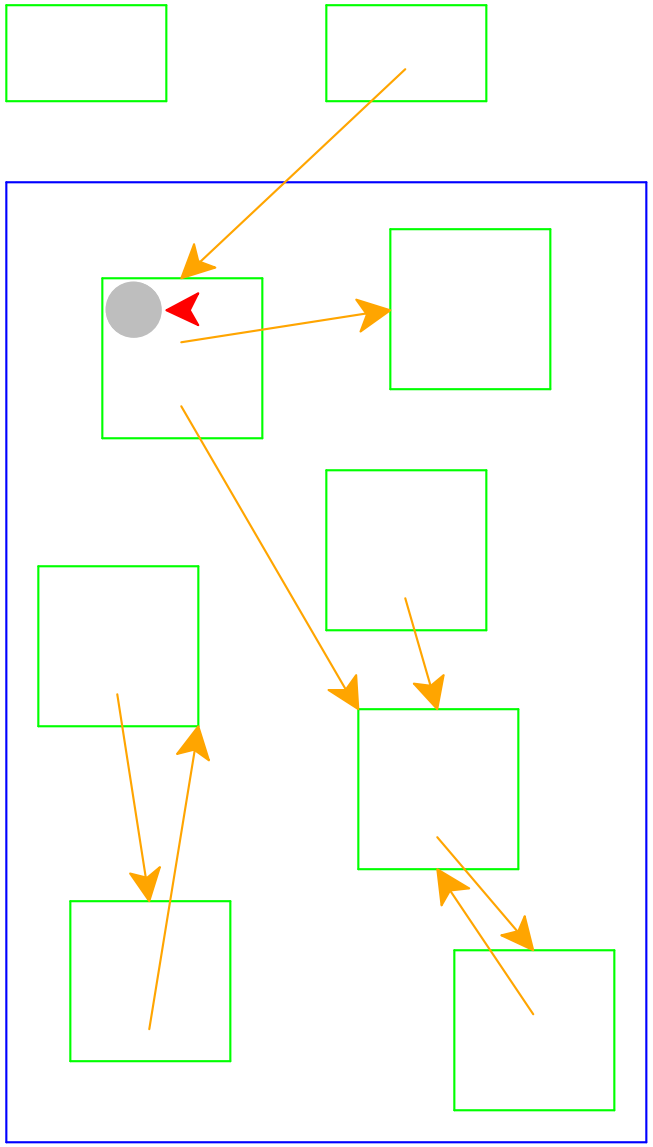
All records are marked white

Mark & Sweep Garbage Collection



Mark records referenced by roots as gray

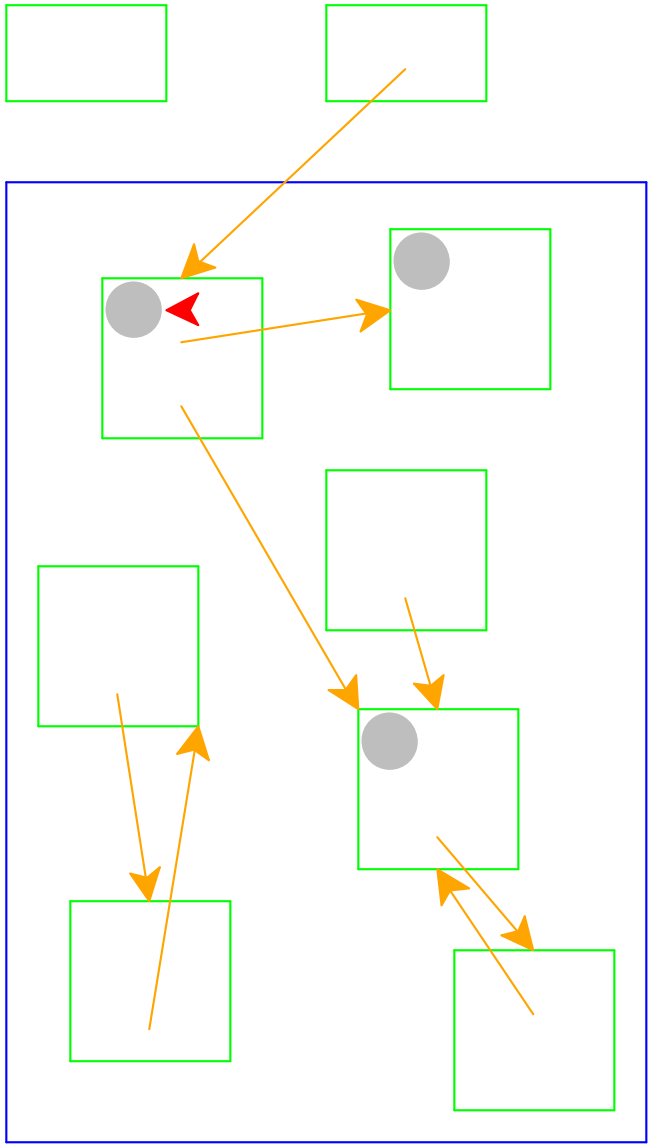
Mark & Sweep Garbage Collection



Need to pick a gray record

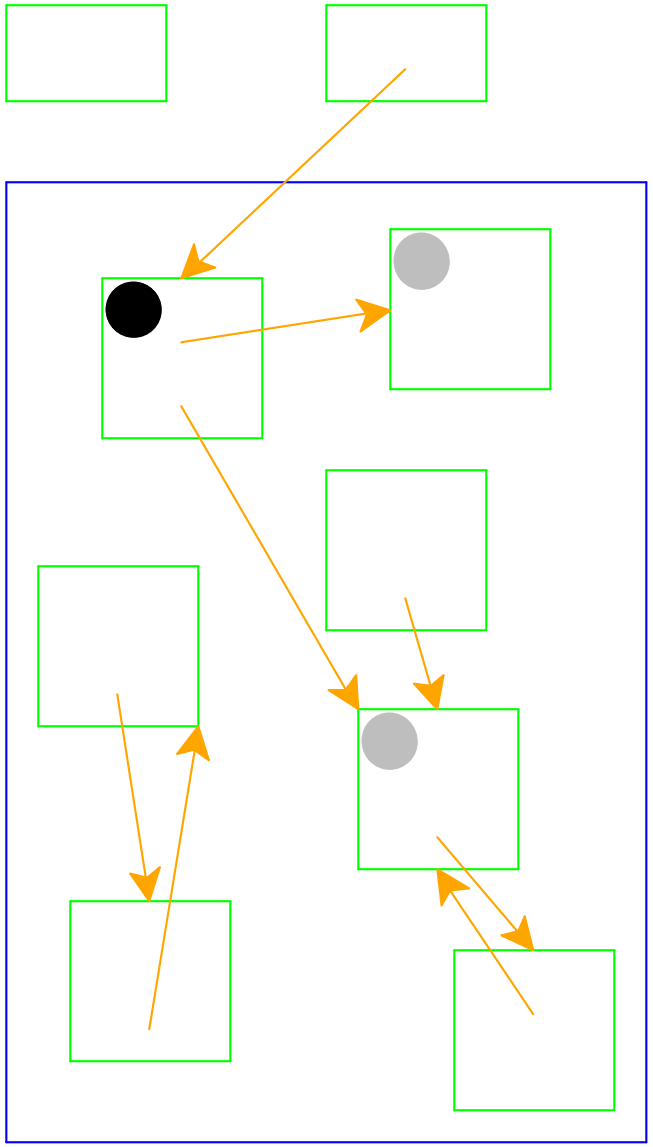
Red arrow indicates the chosen record

Mark & Sweep Garbage Collection



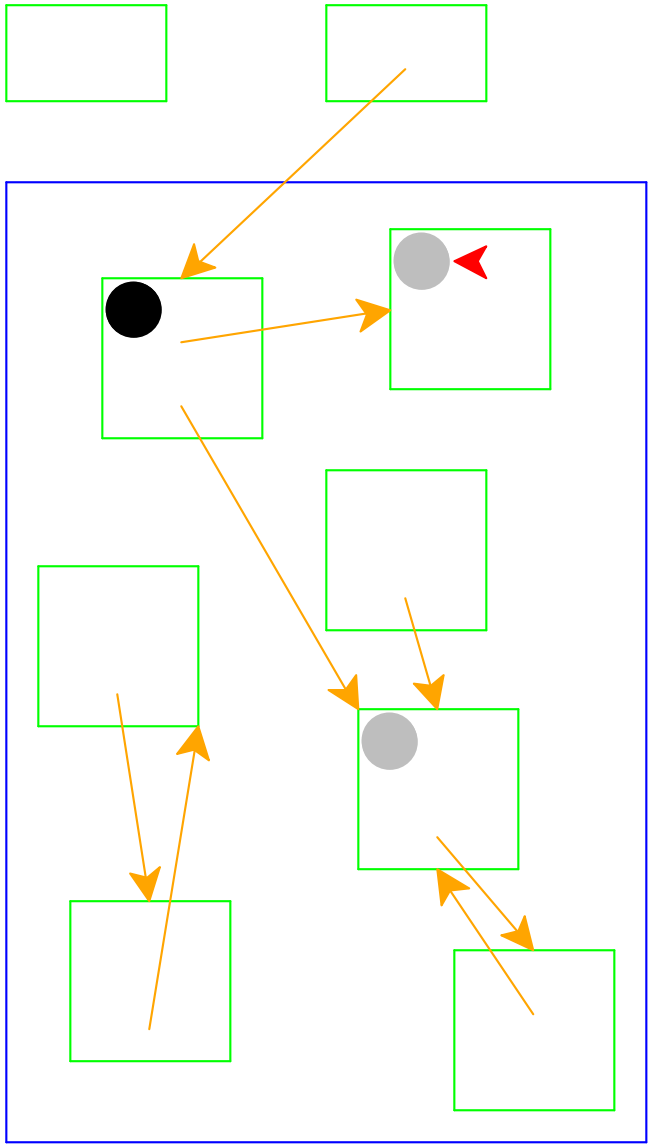
Mark white records referenced by chosen record as gray

Mark & Sweep Garbage Collection



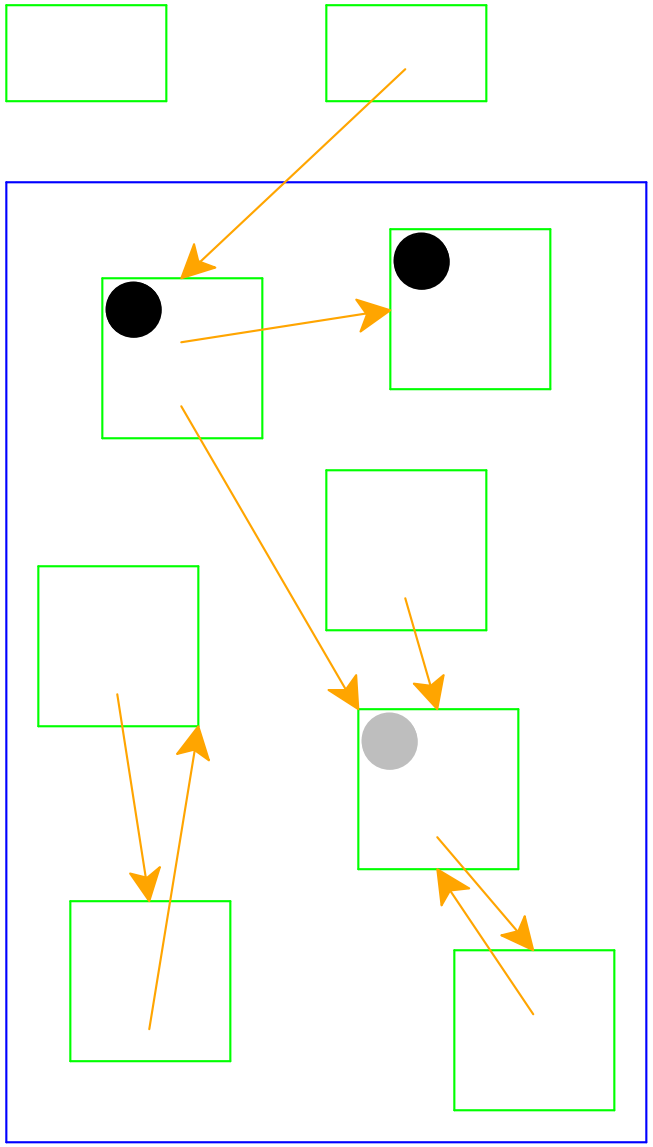
Mark chosen record black

Mark & Sweep Garbage Collection



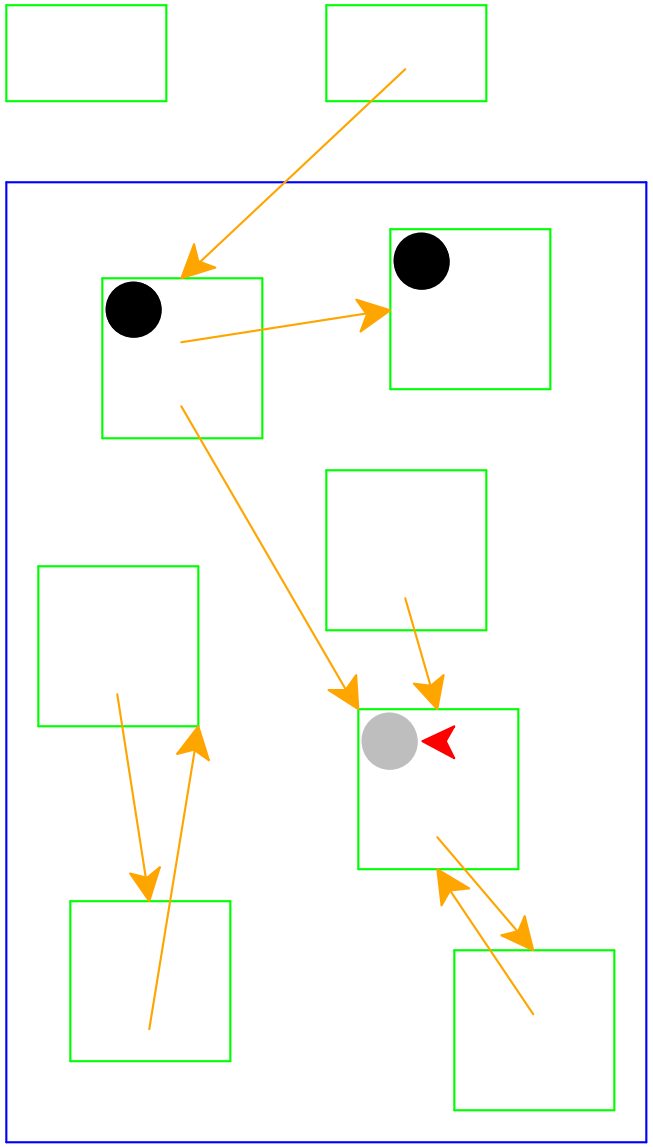
Start again: pick a gray record

Mark & Sweep Garbage Collection



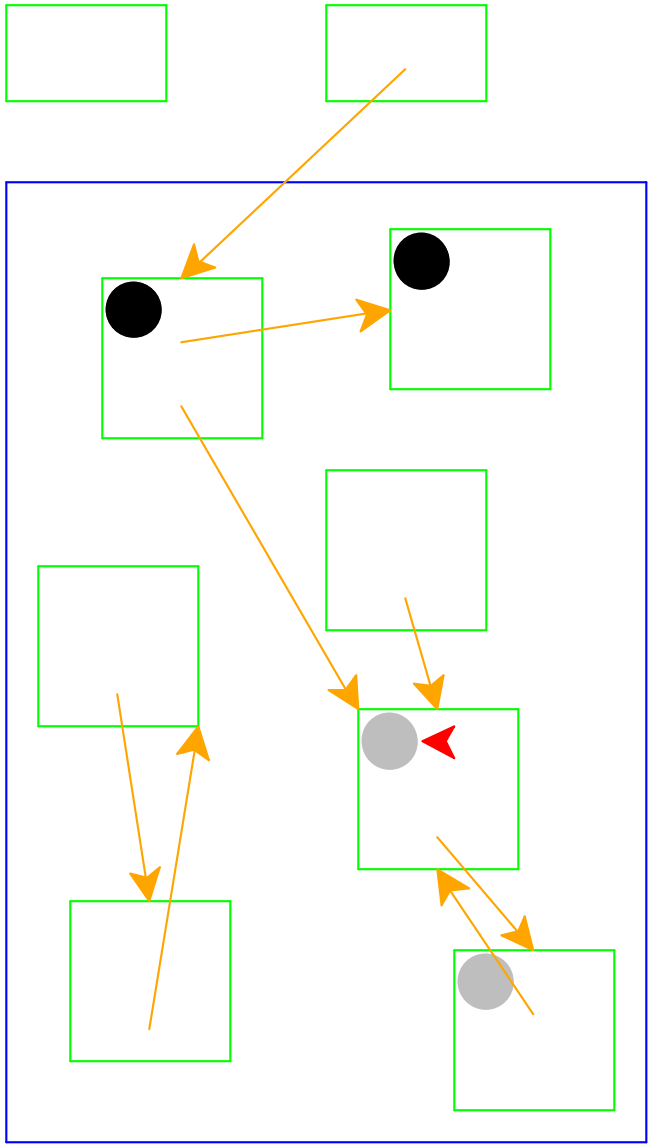
No referenced records; mark black

Mark & Sweep Garbage Collection



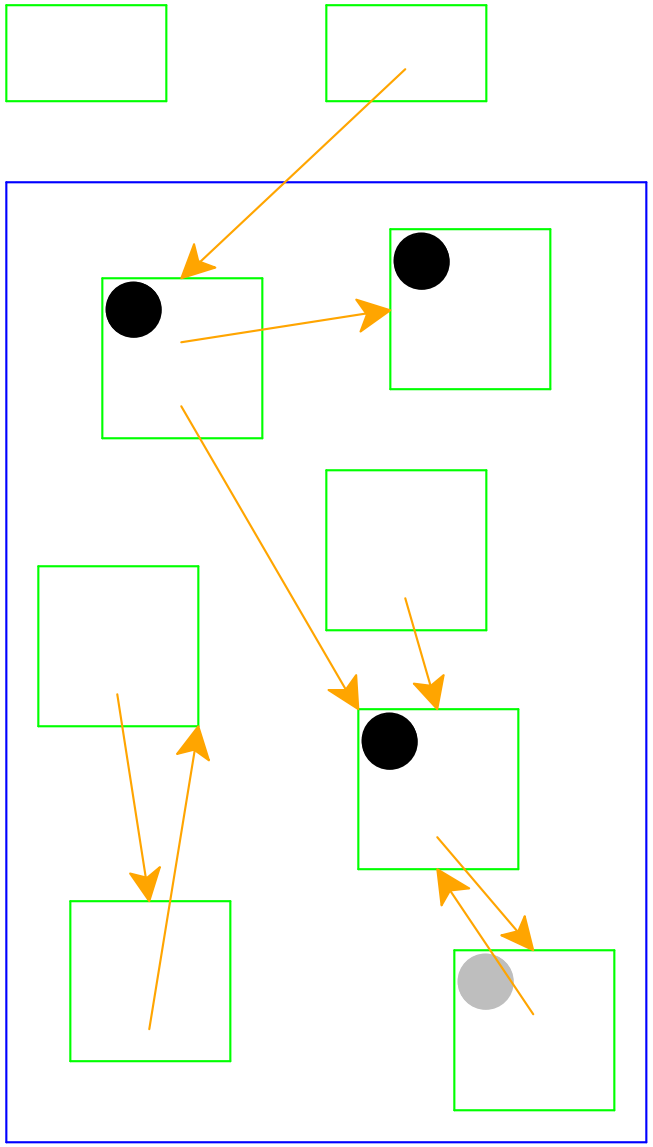
Start again: pick a gray record

Mark & Sweep Garbage Collection



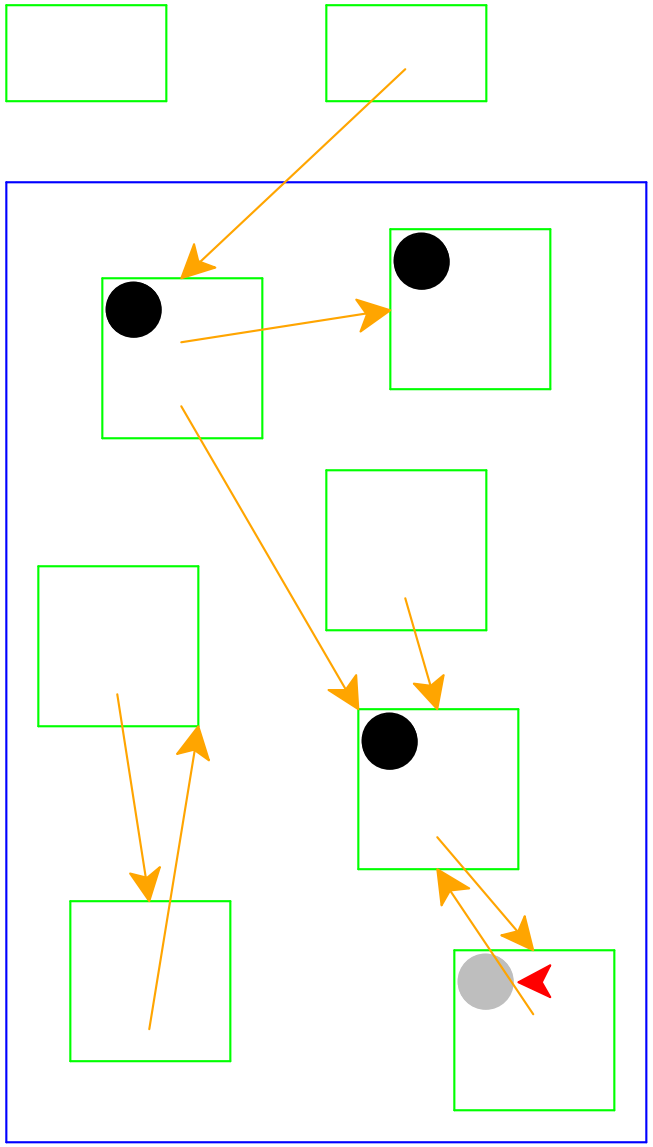
Mark white records referenced by chosen record as gray

Mark & Sweep Garbage Collection



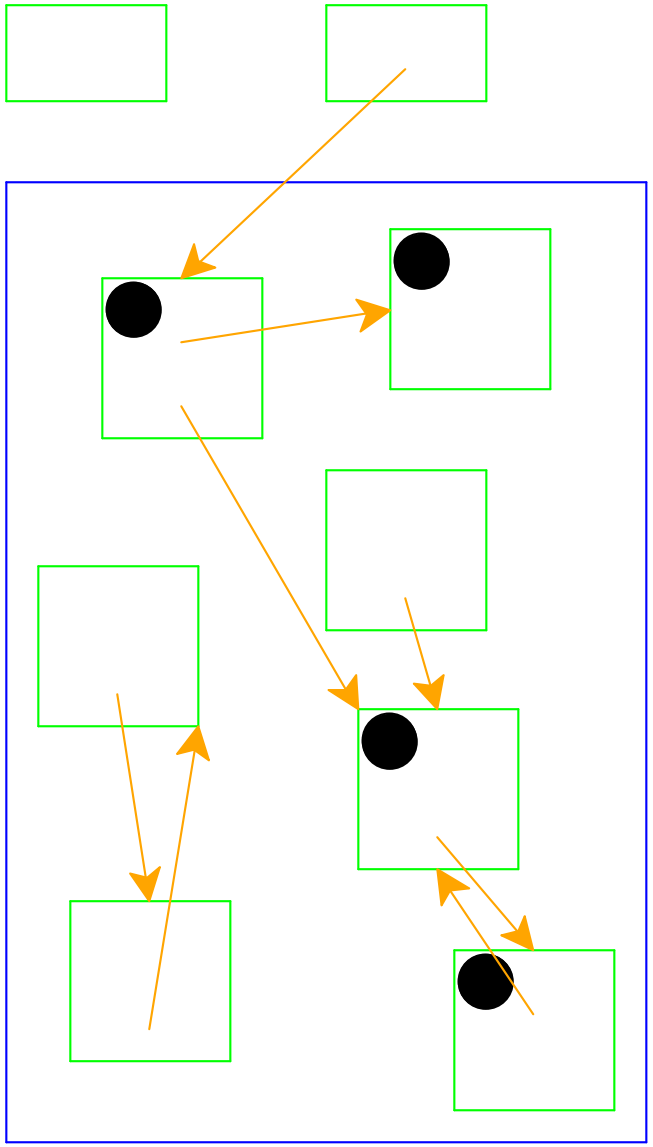
Mark chosen record black

Mark & Sweep Garbage Collection



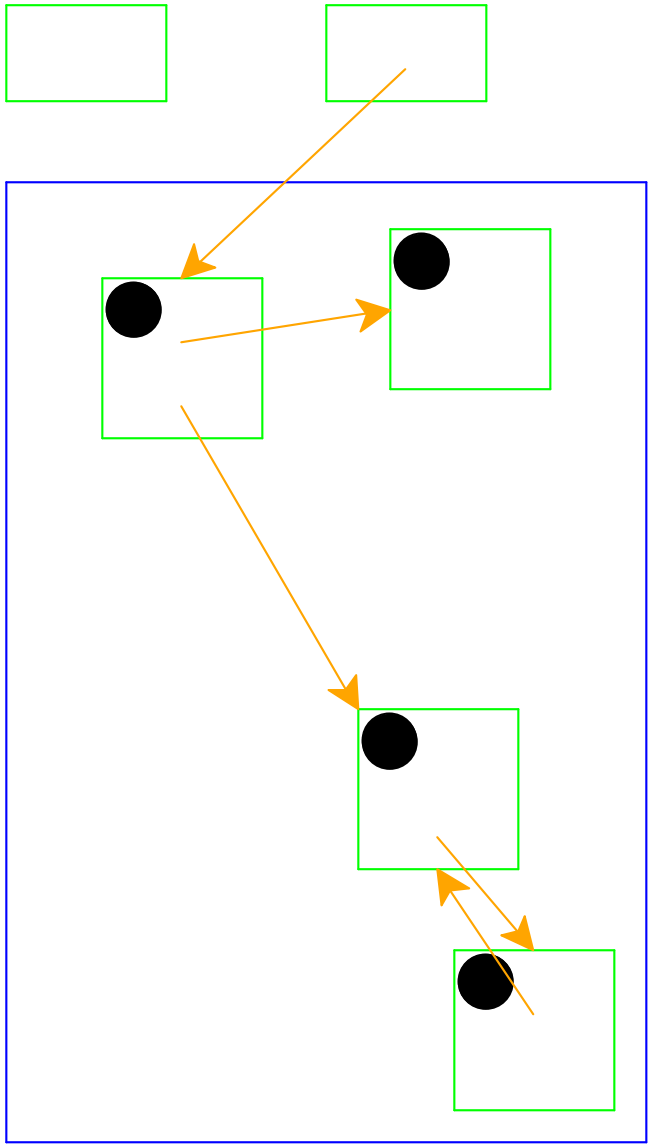
Start again: pick a gray record

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 Problems

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

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

Two-Space Copying Collectors

A ***two-space*** copying collector compacts memory as it collects, making allocation easier.

Allocator:

- Partitions memory into ***to-space*** and ***from-space***
- Allocates only in ***to-space***

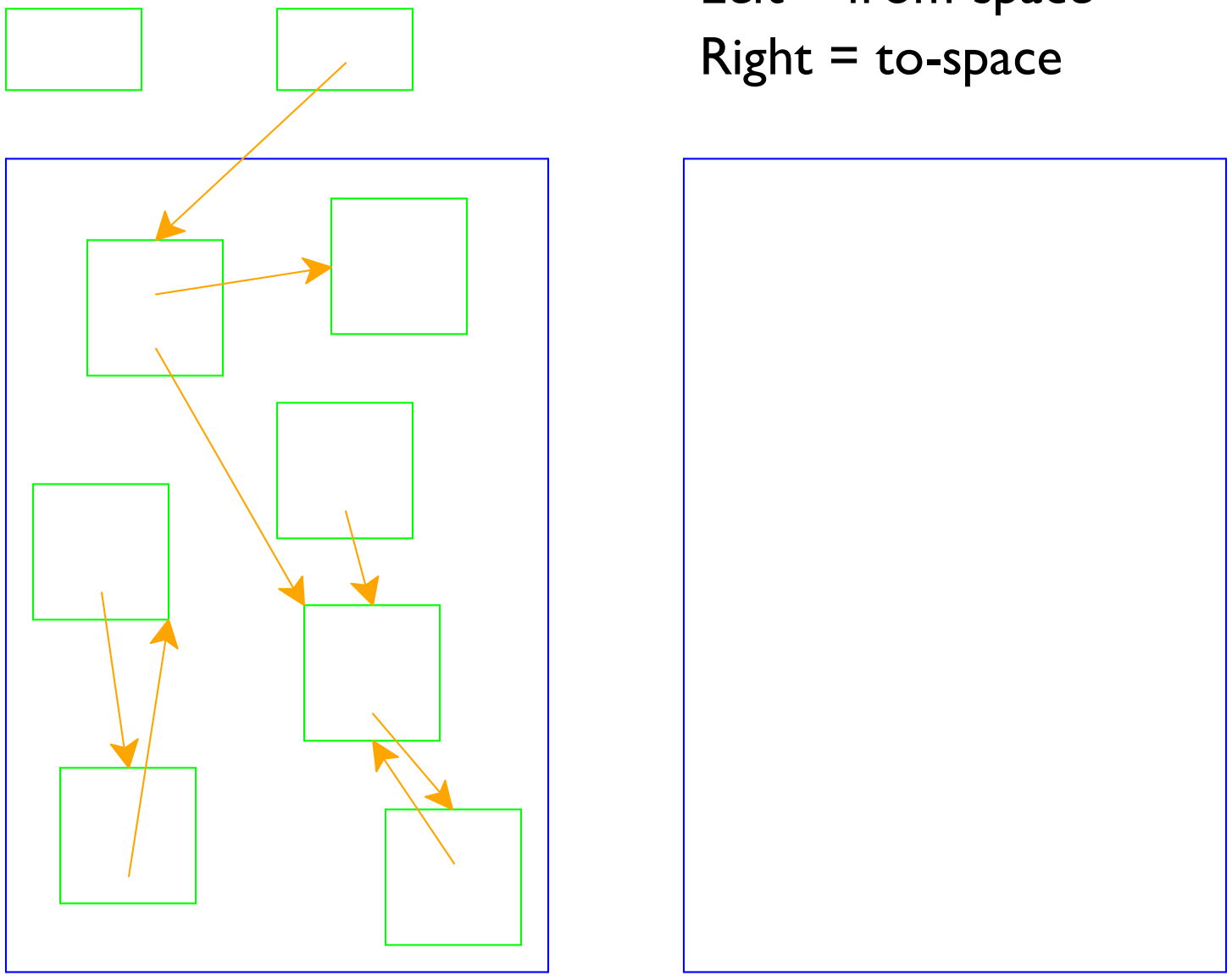
Collector:

- Starts by swapping ***to-space*** and ***from-space***
- Coloring gray ⇒ copy from ***from-space*** to ***to-space***
- Choosing a gray record ⇒ walk once though the new ***to-space***, update pointers

Two-Space Collection

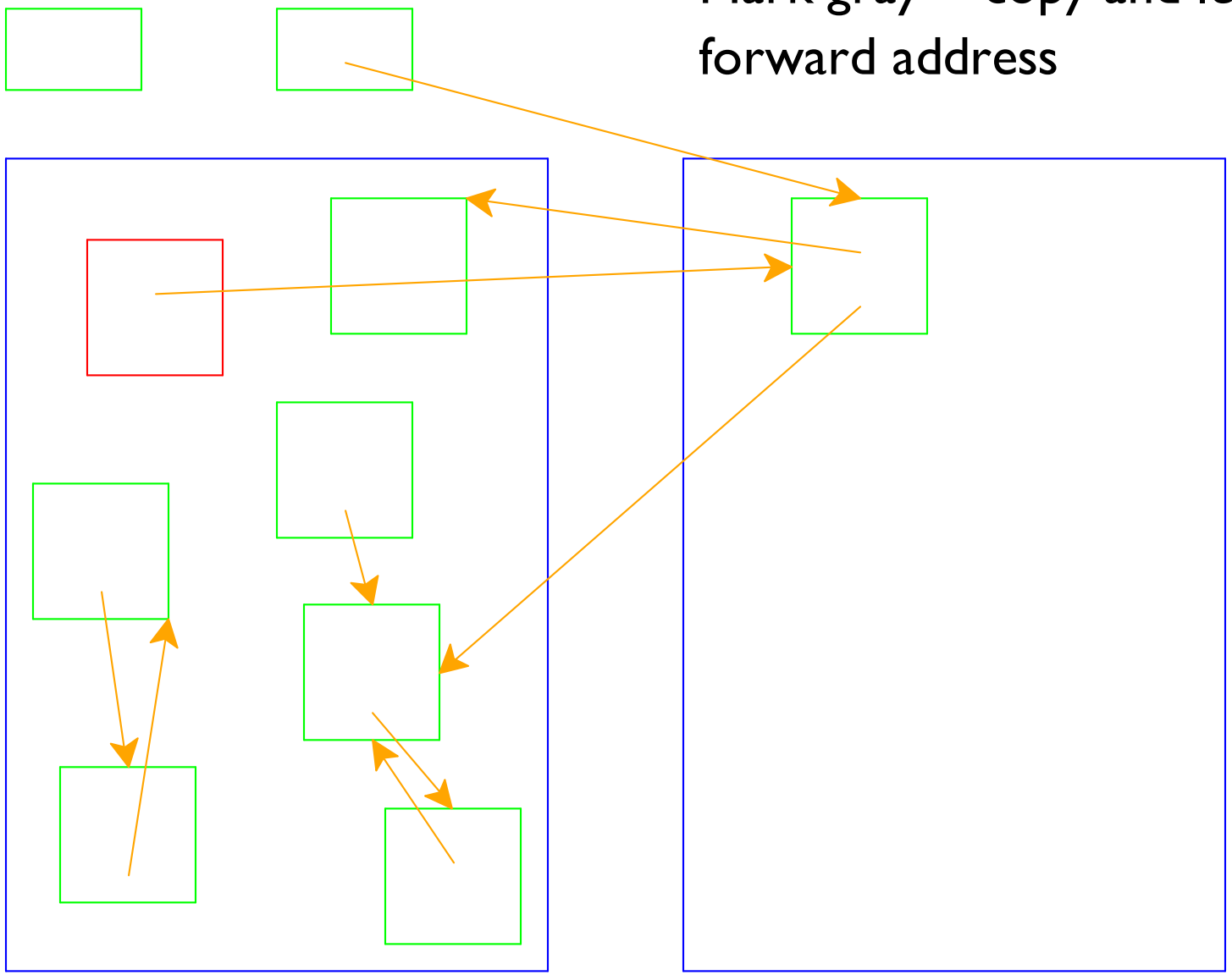
Left = from-space

Right = to-space



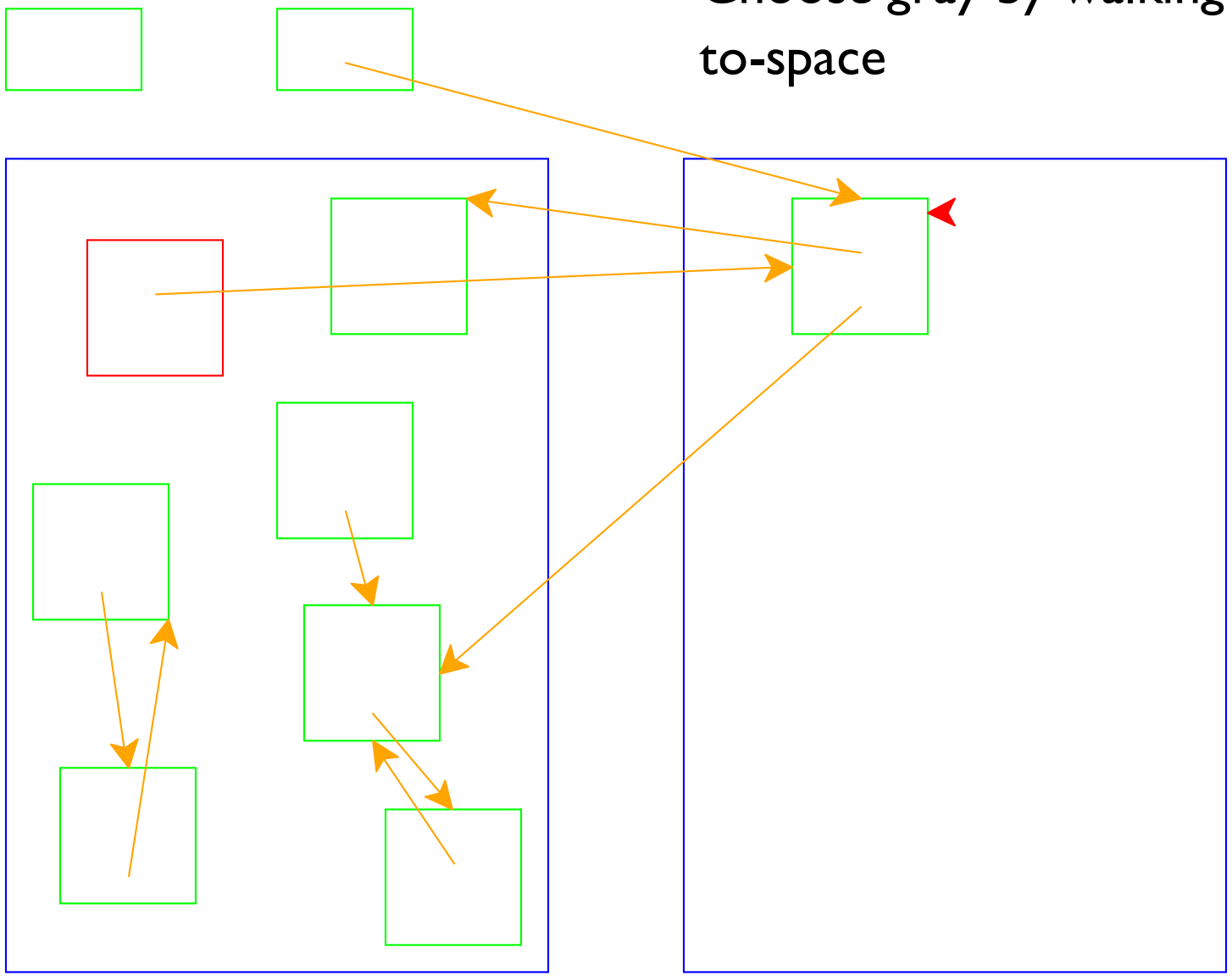
Two-Space Collection

Mark gray = copy and leave forward address



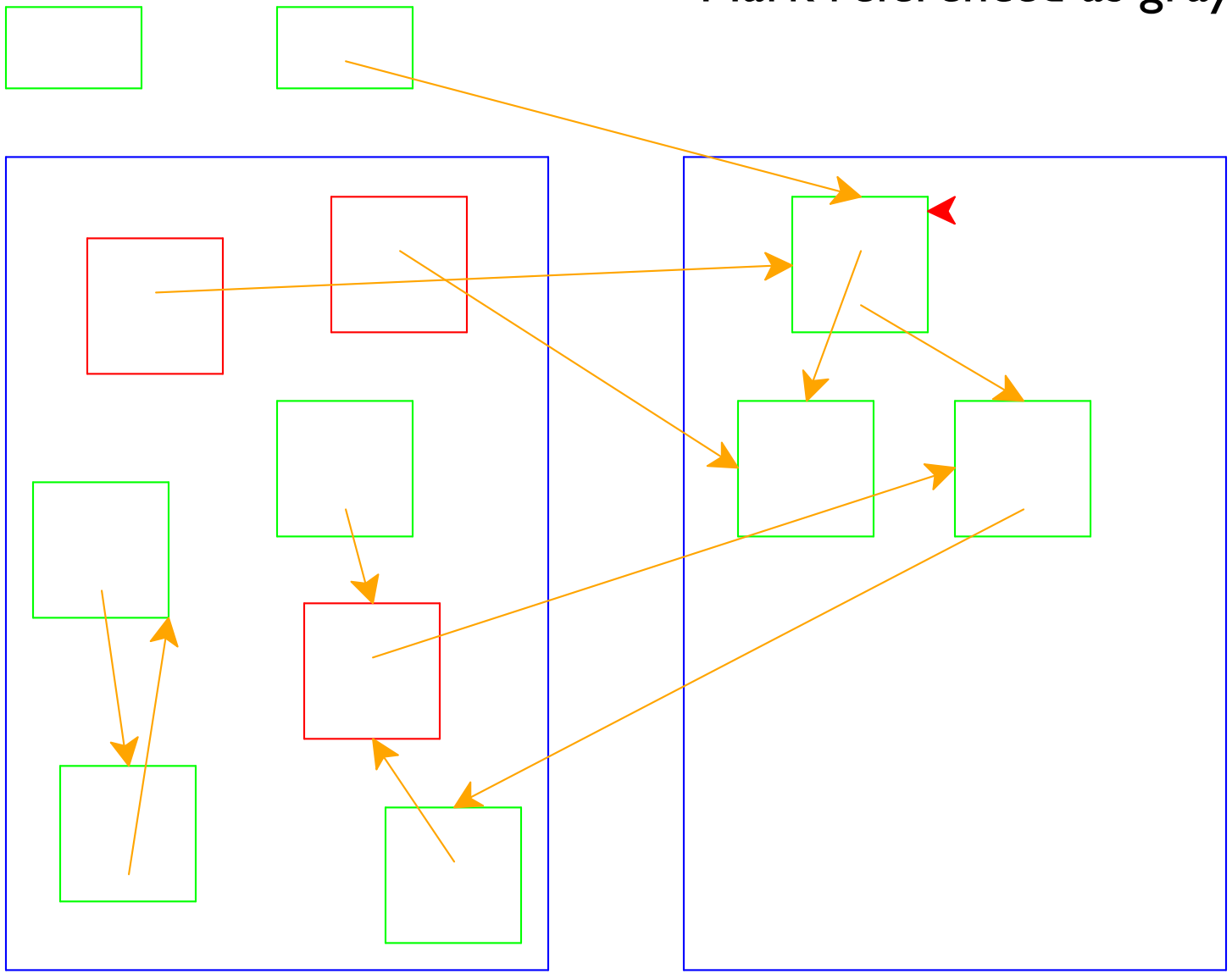
Two-Space Collection

Choose gray by walking through to-space



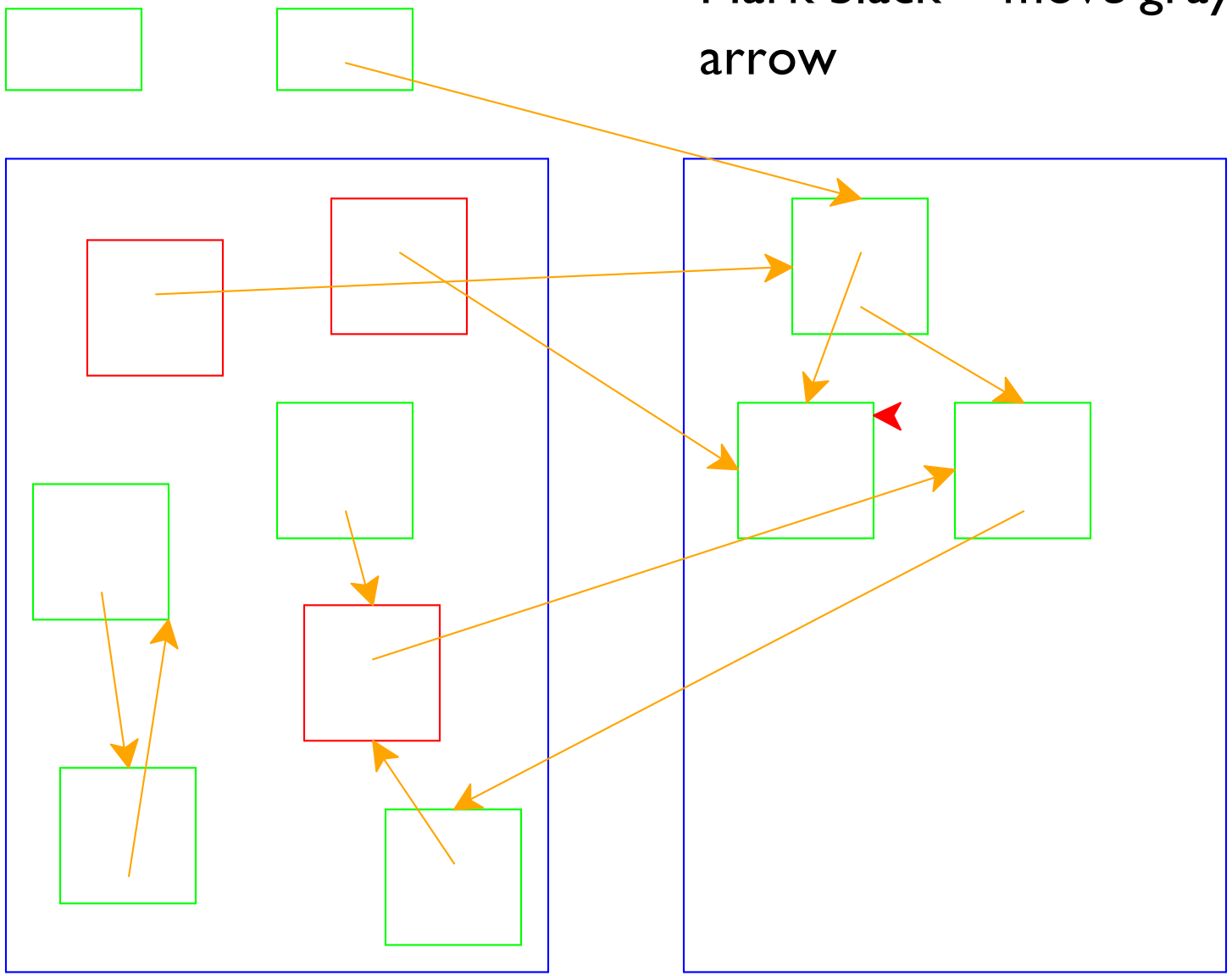
Two-Space Collection

Mark referenced as gray



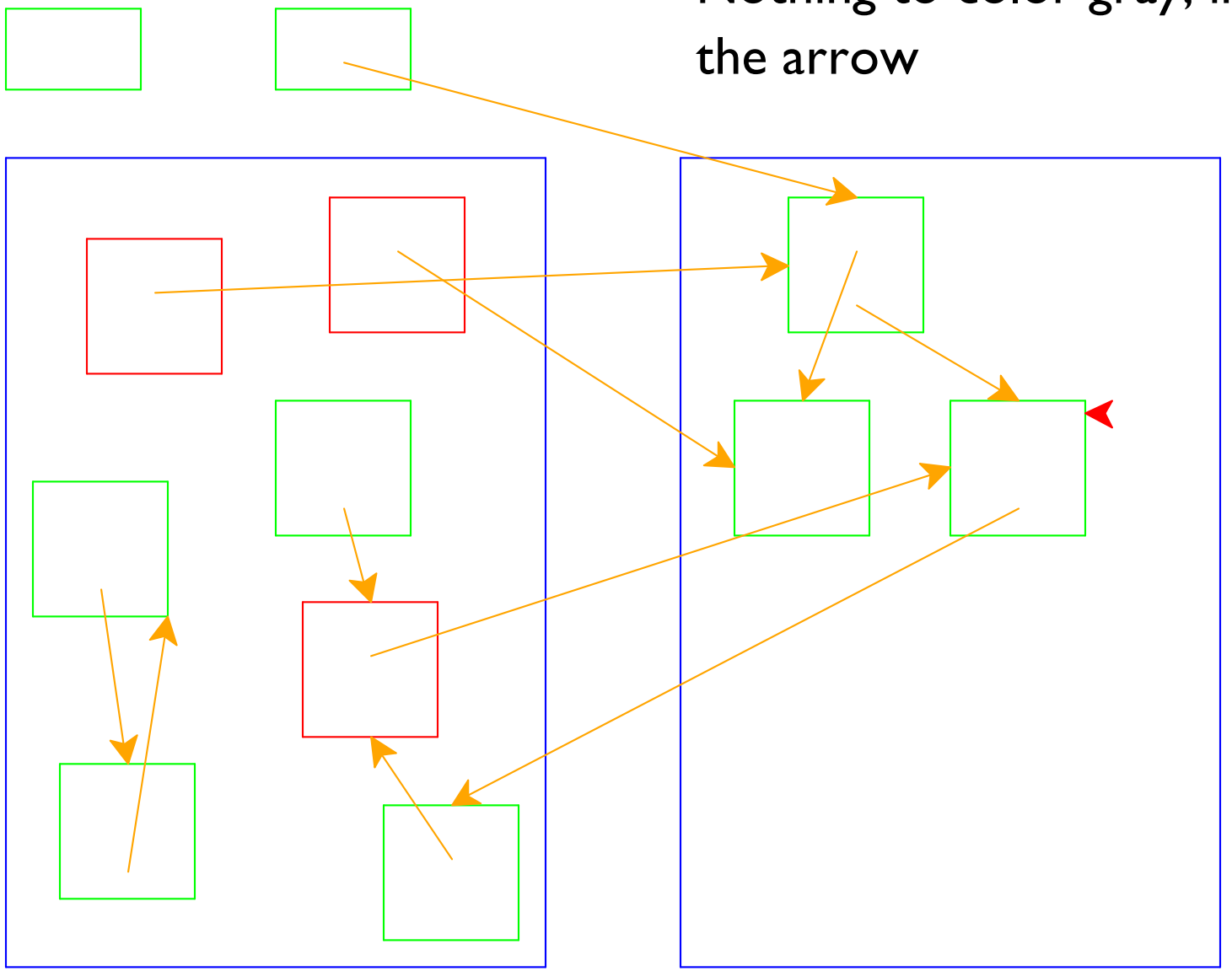
Two-Space Collection

Mark black = move gray-choosing arrow



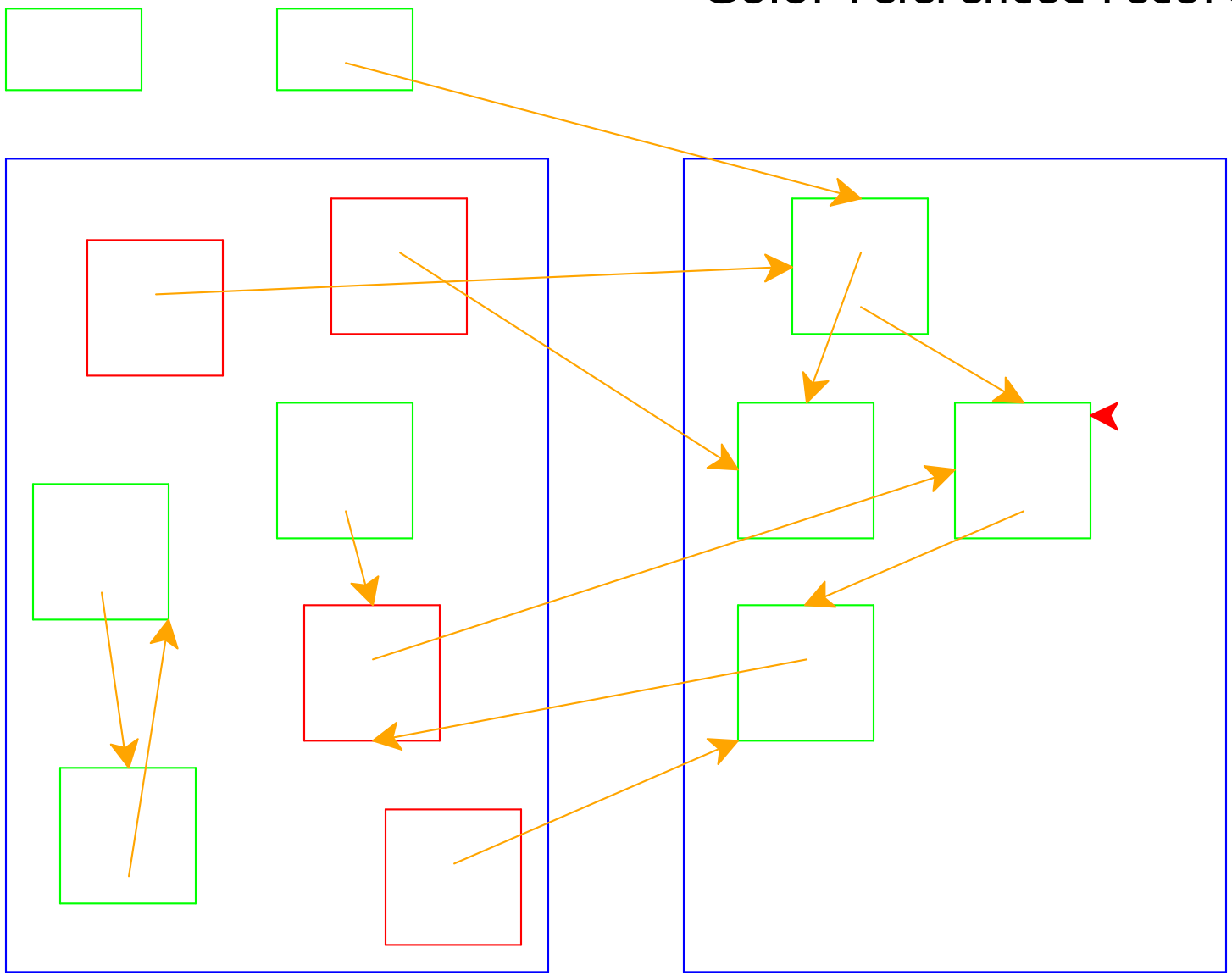
Two-Space Collection

Nothing to color gray; increment the arrow



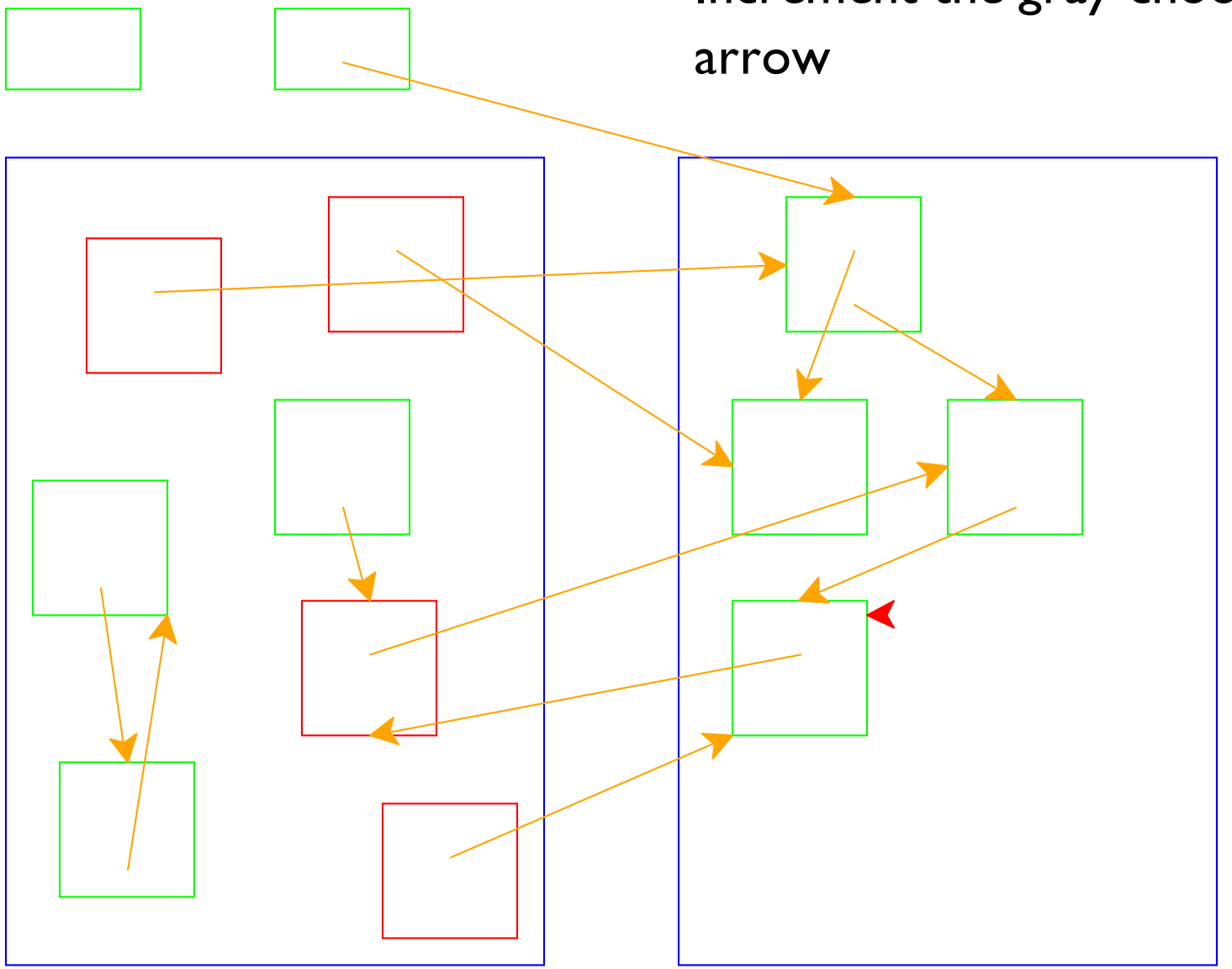
Two-Space Collection

Color referenced record gray



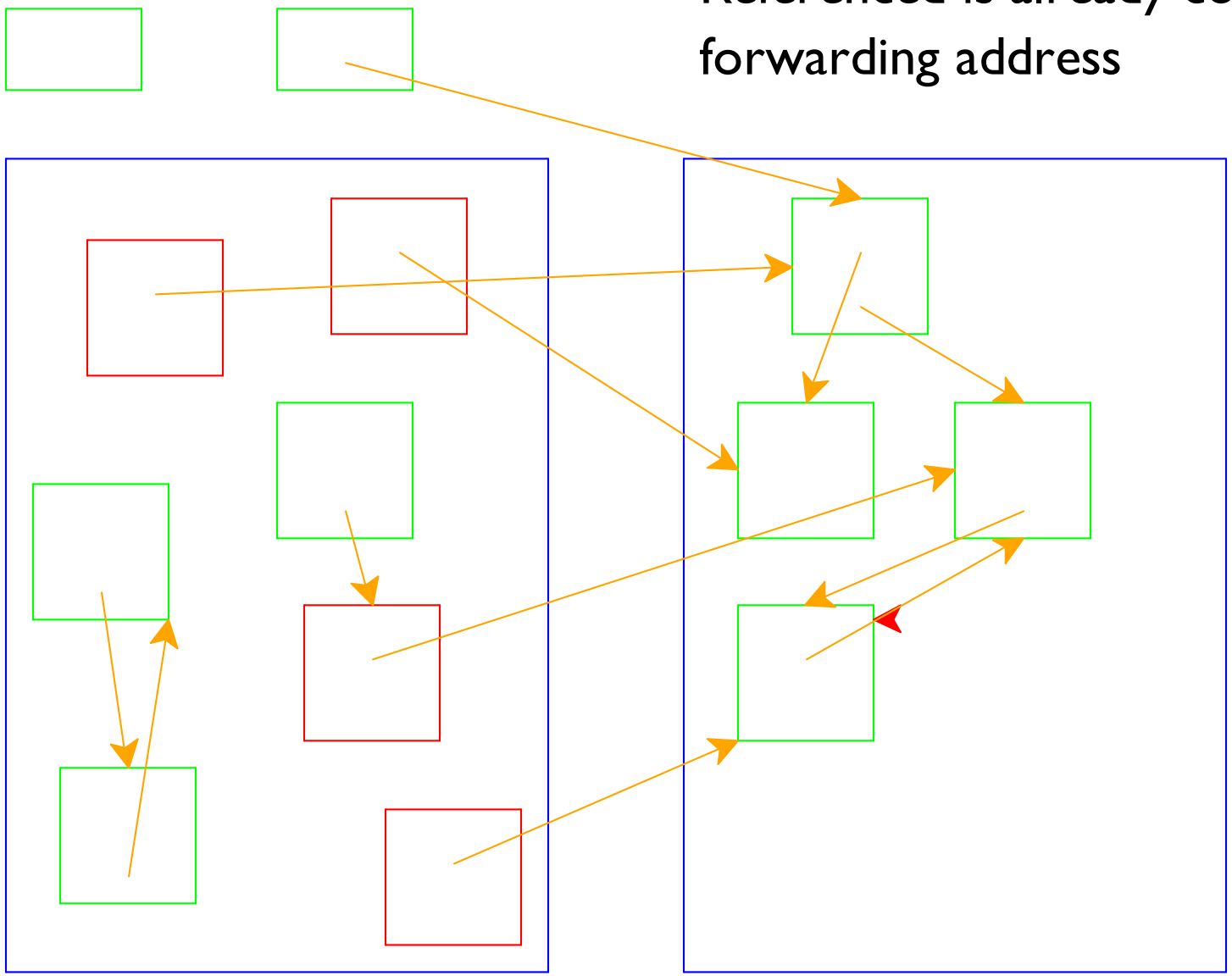
Two-Space Collection

Increment the gray-choosing arrow



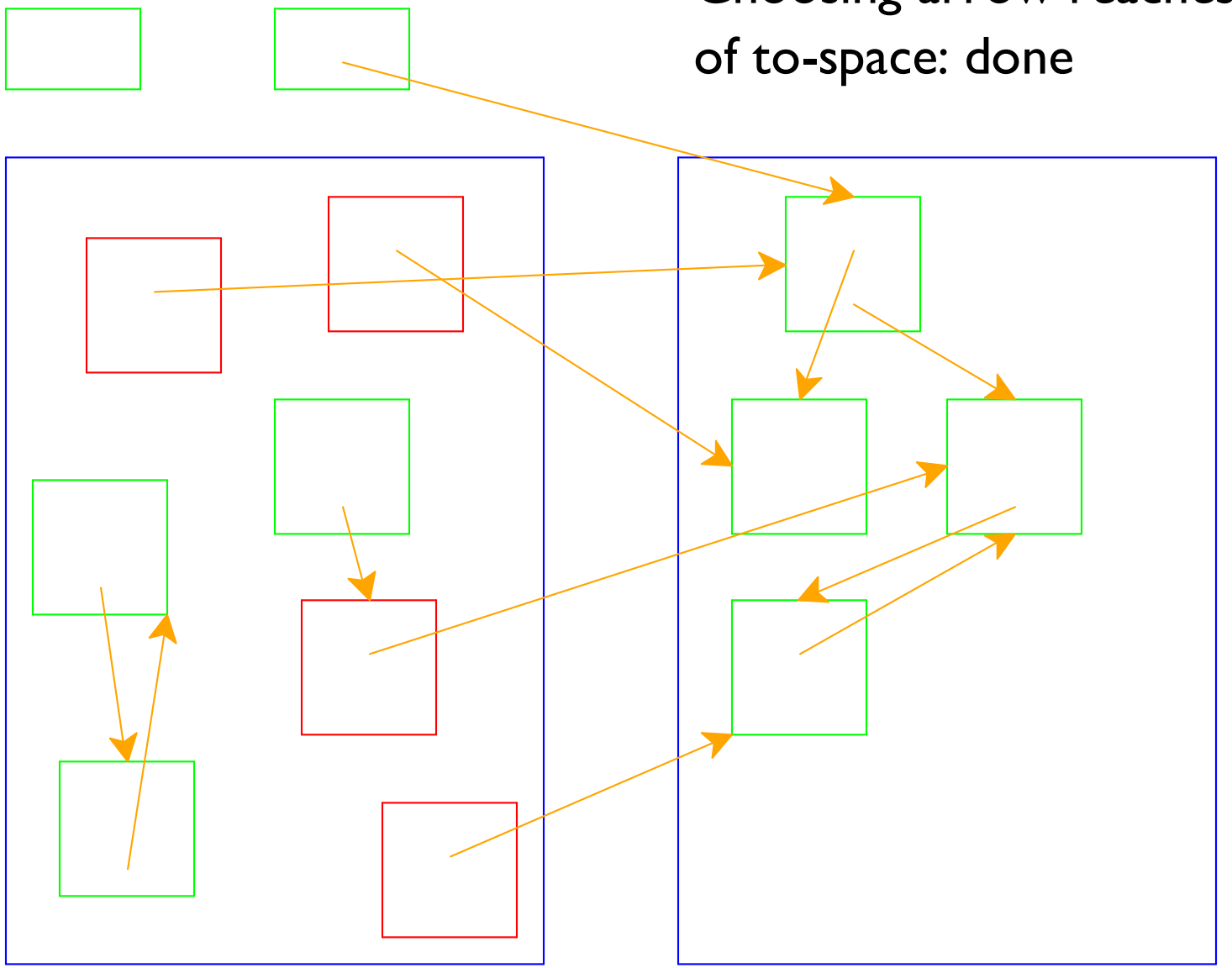
Two-Space Collection

Referenced is already copied, use forwarding address

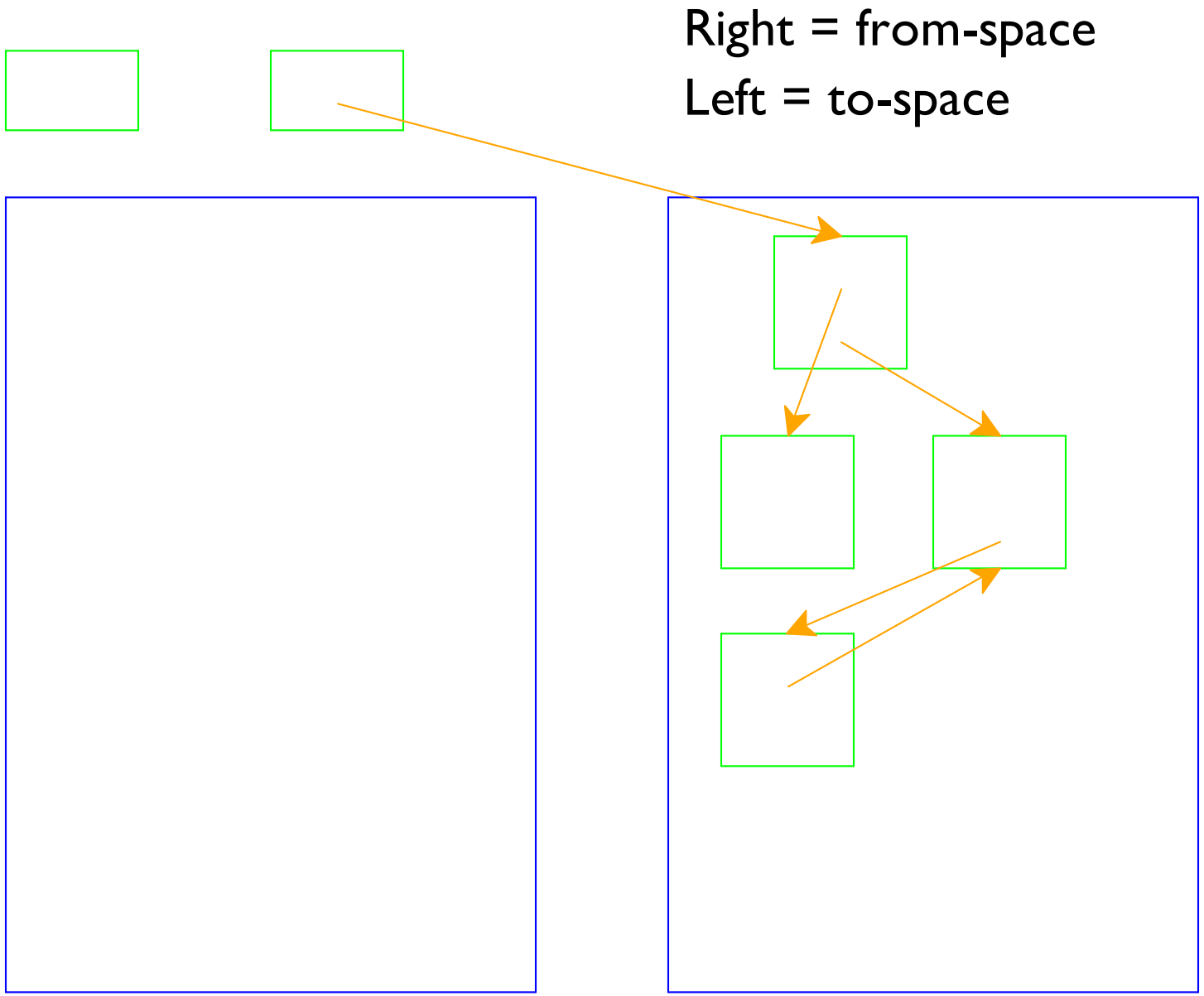


Two-Space Collection

Choosing arrow reaches the end of to-space: done



Two-Space Collection



Two-Space Collection on Vectors

- Everything is a number:
  - Some numbers are immediate integers
  - Some numbers are pointers
- An allocated record in memory starts with a tag, followed by a sequence of pointers and immediate integers
  - The tag describes the shape

Two-Space Collection on Vectors

- Use two pointers into the ***to-space*** to maintain a queue for a breadth-first traversal
- Inc the end pointer to add to the queue, increment the front pointer to remove from the queue; when the pointers come together, terminate

Two-Space Vector Example

- 26-byte memory (13 bytes per space), 2 roots
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

Root 1: 7 Root 2: 0

From: `1 75 2 0 3 2 10 3 2 2 3 1 4`

Two-Space Vector Example

- 26-byte memory (13 bytes per space), 2 roots
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

Root 1: 7 Root 2: 0

From:	1	75	2	0	3	2	10	3	2	2	3	1	4
Addr:	00	01	02	03	04	05	06	07	08	09	10	11	12

Two-Space Vector Example

- 26-byte memory (13 bytes per space), 2 roots
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

Root 1: 7 Root 2: 0

From:	1	75	2	0	3	2	10	3	2	2	3	1	4
Addr:	00	01	02	03	04	05	06	07	08	09	10	11	12
	^		^		^			^			^		

Two-Space Vector Example

- 26-byte memory (13 bytes per space), 2 roots
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

Root 1: 7 Root 2: 0

From:	1	75	2	0	3	2	10	3	2	2	3	1	4
Addr:	00	01	02	03	04	05	06	07	08	09	10	11	12
	^		^		^			^			^		
To:	0	0	0	0	0	0	0	0	0	0	0	0	0
Q:													

Two-Space Vector Example

- 26-byte memory (13 bytes per space), 2 roots
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer
  - Tag 99: forwarding pointer (to to space)

Root 1: 0 Root 2: 0

From:	1	75	2	0	3	2	10	99	0	2	3	1	4
Addr:	00	01	02	03	04	05	06	07	08	09	10	11	12
	^		^		^			^			^		
To:	3	2	2	0	0	0	0	0	0	0	0	0	0
Q:	^			^									

Two-Space Vector Example

- 26-byte memory (13 bytes per space), 2 roots
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer
  - Tag 99: forwarding pointer (to to space)

Root 1: 0 Root 2: 3

```
From:  99  3  2  0  3  2 10 99  0  2  3  1  4
Addr:  00 01 02 03 04 05 06 07 08 09 10 11 12
        ^     ^     ^        ^        ^
To:    3  2  2  1 75  0  0  0  0  0  0  0  0
Q:     ^        ^
```

Two-Space Vector Example

- 26-byte memory (13 bytes per space), 2 roots
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer
  - Tag 99: forwarding pointer (to to space)

Root 1: 0 Root 2: 3

```
From:  99  3 99  5  3  2 10 99  0  2  3  1  4
Addr:  00 01 02 03 04 05 06 07 08 09 10 11 12
        ^     ^     ^        ^        ^
To:     3  2  5  1 75  2  0  0  0  0  0  0  0
Q:             ^        ^
```

Two-Space Vector Example

- 26-byte memory (13 bytes per space), 2 roots
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer
  - Tag 99: forwarding pointer (to to space)

Root 1: 0 Root 2: 3

```
From:  99  3 99  5  3  2 10 99  0  2  3  1  4
Addr:  00 01 02 03 04 05 06 07 08 09 10 11 12
        ^     ^     ^        ^        ^
To:     3  2  5  1 75  2  0  0  0  0  0  0  0
Q:                    ^     ^
```

Two-Space Vector Example

- 26-byte memory (13 bytes per space), 2 roots
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer
  - Tag 99: forwarding pointer (to to space)

Root 1: 0 Root 2: 3

```
From:  99  3 99  5  3  2 10 99  0  2  3  1  4
Addr:  00 01 02 03 04 05 06 07 08 09 10 11 12
        ^     ^     ^        ^        ^
To:     3  2  5  1 75  2  3  0  0  0  0  0  0
Q:                          ^^
```

Further reading

Uniprocessor Garbage Collection Techniques, by Wilson

`ftp://ftp.cs.utexas.edu/pub/garbage/gcsurvey.ps`

Mark and sweep implementation, with linear-time allocator

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

(define (gc:flat? loc)
  (equal? (heap-ref loc) 'flat))

(define (gc:deref loc)
  (cond
    [(equal? (heap-ref loc) 'flat)
     (heap-ref (+ loc 1))]
    [else
     (error 'gc:deref
            "non-flat @ ~s"
            loc)]))
```

Mark and sweep implementation, with linear-time allocator

```
(define (gc:cons? loc)
  (equal? (heap-ref loc) 'pair))

(define (gc:first pr-ptr)
  (if (equal? (heap-ref pr-ptr) 'pair)
      (heap-ref (+ pr-ptr 1))
      (error 'first "non pair @ ~s" pr-ptr)))

(define (gc:rest pr-ptr)
  (if (equal? (heap-ref pr-ptr) 'pair)
      (heap-ref (+ pr-ptr 2))
      (error 'rest "non pair @ ~s" pr-ptr)))
```

Mark and sweep implementation, with linear-time allocator

```
(define (gc:set-first! pr-ptr new)
  (if (equal? (heap-ref pr-ptr) 'pair)
      (heap-set! (+ pr-ptr 1) new)
      (error 'set-first! "non pair")))

(define (gc:set-rest! pr-ptr new)
  (if (equal? (heap-ref pr-ptr) 'pair)
      (heap-set! (+ pr-ptr 2) new)
      (error 'set-first! "non pair")))
```

Mark and sweep implementation, with linear-time allocator

```
(define (gc:alloc-flat fv)
  (let ([ptr (alloc 2
                    (if (procedure? fv)
                        (procedure-roots fv)
                        '())
                    '())])
    (heap-set! ptr 'flat)
    (heap-set! (+ ptr 1) fv)
    ptr))

(define (gc:cons hd tl)
  (let ([ptr (alloc 3 hd tl)])
    (heap-set! ptr 'pair)
    (heap-set! (+ ptr 1) hd)
    (heap-set! (+ ptr 2) tl)
    ptr))
```

Mark and sweep implementation, with linear-time allocator

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

; alloc : number[size] roots roots -> loc
(define (alloc n some-roots more-roots)
  (let ([next (find-free-space 0 n)])
    (cond
      [next next]
      [else
       (collect-garbage some-roots more-roots)
       (let ([next (find-free-space 0 n)])
         (unless next (error 'alloc "no space"))
         next)])))
```

Mark and sweep implementation, with linear-time allocator

```
; find-free-space : loc number -> loc or #f
(define (find-free-space start size)
  (cond
    [(= start (heap-size)) #f]
    [(n-free-blocks? start size) start]
    [else (find-free-space (+ start 1) size)]))

; n-free-blocks? : loc number -> loc or #f
(define (n-free-blocks? start size)
  (cond
    [(= size 0) #t]
    [(= start (heap-size)) #f]
    [else
     (and (eq? 'free (heap-ref start))
          (n-free-blocks? (+ start 1)
                          (- size 1)))]))
```

Mark and sweep implementation, with linear-time allocator

```
; collect-garbage : roots roots -> void
(define (collect-garbage some-roots more-roots)
  (mark-white! 0)
  (traverse/roots (get-root-set))
  (traverse/roots some-roots)
  (traverse/roots more-roots)
  (free-white! 0))
```

Mark and sweep implementation, with linear-time allocator

```
; mark-white! : loc -> void
; marks all records as white, starting with 'i'
; (linear scan of the heap (this linear scan isn't
; really necc but we do it that way for simplicity))
(define (mark-white! i)
  (when (< i (heap-size))
    (case (heap-ref i)
      [(pair) (heap-set! i 'white-pair)
              (mark-white! (+ i 3))]
      [(flat) (heap-set! i 'white-flat)
              (mark-white! (+ i 2))]
      [(free) (mark-white! (+ i 1))]
      [else (error 'mark-white!
                   "unknown tag ~s"
                   (heap-ref i))])))
```

Mark and sweep implementation, with linear-time allocator

```
; free-white : loc -> void
; frees all white records, starting at 'i'
(define (free-white! i)
  (when (< i (heap-size))
    (case (heap-ref i)
      [(pair)        (free-white! (+ i 3))]
      [(flat)        (free-white! (+ i 2))]
      [(white-pair)  (heap-set! i 'free)
                     (heap-set! (+ i 1) 'free)
                     (heap-set! (+ i 2) 'free)
                     (free-white! (+ i 3))]
      [(white-flat)  (heap-set! i 'free)
                     (heap-set! (+ i 1) 'free)
                     (free-white! (+ i 2))]
      [(free)        (free-white! (+ i 1))]
      [else (error 'free-white! "~s" i)])))
```

Mark and sweep implementation, with linear-time allocator

```
; traverse/roots : roots -> void
; traverses the heap, marking
; everything reachable from 'roots'
(define (traverse/roots thing)
  (cond
    [(list? thing)
     (for-each traverse/roots thing)]
    [(root? thing)
     (traverse/loc (read-root thing))]
    [(number? thing)
     (traverse/loc thing)]))
```

Mark and sweep implementation, with linear-time allocator

```
; traverse/loc : loc -> void
; depth first search for live records
(define (traverse/loc loc)
  (case (heap-ref loc)
    [(white-pair)
     (heap-set! loc 'pair)
     (traverse/loc (heap-ref (+ loc 1)))
     (traverse/loc (heap-ref (+ loc 2)))]
    [(white-flat)
     (heap-set! loc 'flat)
     (let ([val (heap-ref (+ loc 1))])
       (when (procedure? val)
         (traverse/roots (procedure-roots val))))]
    [(pair) (void)]
    [(flat) (void)]
    [else (error 'traverse/loc "~s" loc)]))
```