Memory as a vector of numbers

- 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

Memory as a vector of numbers (example)

- 13-byte memory
    - Tag 1: one integer
    - Tag 2: one pointer
    - Tag 3: one integer, then one pointer
- Example Memory State
    - 1 75 2 0 3 2 10 3 2 2 3 1 4

Memory as a vector of numbers (example)

- 13-byte memory
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer
- Example Memory State
  - Root 1: 7
  - Root 2: 0
  - [1 75] 2 0 3 2 10 [3 2 2] 3 1 4

Memory as a vector of numbers (example)

- Tag: 1: integer, 2: pointer, 3 : (integer,pointer)
- Root: 1->0, 2->7
- [1 75] 2 0 3 2 10 [3 2 2] 3 1 4

Memory as a vector of numbers (example)

- Tag: 1: integer, 2: pointer, 3 : (integer,pointer)
- Root: 1->0, 2->7
- [1 75] 2 0 3 2 10 [3 2 2] 3 1 4
- [1 75] [2 0] 3 2 10 [3 2 2] 3 1 4

Memory as a vector of numbers (example)

- Tag: 1: integer, 2: pointer, 3 : (integer,pointer)
- Root: 1->4, 2->7
- 1 75 2 0 [3 2 10] [3 2 2] 3 1 4

Memory as a vector of numbers (example)

- Tag: 1: integer, 2: pointer, 3 : (integer,pointer)
- Root: 1->4, 2->7
- 1 75 2 0 [3 2 10] [3 2 2] 3 1 4
- 1 75 [2 0] [3 2 10] [3 2 2] [3 1 4]

Memory as a vector of numbers (example)

- Tag: 1: integer, 2: pointer, 3 : (integer,pointer)
- Root: 1->4, 2->7
- 1 75 2 0 [3 2 10] [3 2 2] 3 1 4
- 1 75 [2 0] [3 2 10] [3 2 2] [3 1 4]
- [1 75][2 0] [3 2 10] [3 2 2] [3 1 4]

A simple allocator

```
(define (malloc1 tag a)
  (begin
    (vector-set! memory ptr tag)
    (vector-set! memory (+ ptr 1) a)
    (incptr 2)))
```

A simple allocator

```
(define (malloc2 tag a b)
  (begin
    (vector-set! memory ptr tag)
    (vector-set! memory (+ ptr 1) a)
    (vector-set! memory (+ ptr 2) b)
    (incptr 3)))
```

A simple allocator

```
(define (malloc3 tag a b c)
  (begin
    (vector-set! memory ptr tag)
    (vector-set! memory (+ ptr 1) a)
    (vector-set! memory (+ ptr 2) b)
    (vector-set! memory (+ ptr 3) c)
    (incptr 4)))
```

A simple allocator

```
(define (code-malloc1 tag a)
  (begin
    (vector-set! code-memory code-ptr tag)
    (vector-set! code-memory (+ code-ptr 1) a)
    (code-incptr 2)))
```

A simple allocator

```
(reset!)
(test (malloc2 9 3 4) 0)
(test (ref 0 1) 3)
(test (ref 0 2) 4)
```

Compilation is allocation

```
(define (compile a-fae ds)
 (type-case FAE a-fae
  [(Num n) (code-malloc1 8 n)]
  [(Add l r) (code-malloc2 9 (compile l ds) (compile r ds))]
  [(Id name) ....]
  [(Fun param body-expr) ....]
  [(Call fun-expr arg-expr) ....]
  [(Fun param body-expr) ....]
  [(Call fun-expr arg-expr) ....]
  [(If0 test-expr then-expr else-expr) ....]))
```

Compilation is allocation

```
(define (compile a-fae ds)
  (type-case FAE a-fae
    [(Num n) (code-malloc1 8 n)]
    [(Add l r) (code-malloc2 9 (compile l ds) (compile r ds))]
    [(Id name) (code-malloc1 11 (locate name ds))]
    [(Fun param body-expr) ....]
    [(Call fun-expr arg-expr) ....]
    [(Fun param body-expr) ....]
    [(Call fun-expr arg-expr) ....]
    [(If0 test-expr then-expr else-expr) ....]))
```

Compilation is allocation

```
(define (compile a-fae ds)
  (type-case FAE a-fae
    [(Num n) (code-malloc1 8 n)]
    [(Add l r) (code-malloc2 9 (compile l ds) (compile r ds))]
    [(Id name) (code-malloc1 11 (locate name ds))]
    [(Fun param body-expr)
     (code-malloc1 12 (compile body-expr (aCSub param ds)))]
    [(Call fun-expr arg-expr) ....]
    [(Fun param body-expr) ....]
    [(Call fun-expr arg-expr) ....]
    [(If0 test-expr then-expr else-expr) ....]))
```

Interpretation needs allocation too

```
(define (interp)
  (case (code-ref fae-reg 0)
    [(8) ; num
     (begin
       (set! v-reg (malloc1 15 (code-ref fae-reg 1)))
       (continue))]
    [(9) ; add
         ....]
    [(11) ; id
          ....]
    [(12) ; fun
          ....]
    [(13) ; app
          ....]
    [(14) ; if0
          ....]))
```

Interpretation needs allocation too

```
(define (interp)
  (case (code-ref fae-reg 0)
    [(8) ; num
     (begin
       (set! v-reg (malloc1 15 (code-ref fae-reg 1)))
       (continue))]
    [(9) ; add
     (begin
       (set! k-reg (malloc3 1
                            (code-ref fae-reg 2)
                            ds-reg
                            k-reg))
       (set! fae-reg (code-ref fae-reg 1))
       (interp))]
    [(11) ; id
          ....]
```

Interpretation needs allocation too

```
(define (interp)
  (case (code-ref fae-reg 0)
    [(8) ; num
         ....]
    [(9) ; add
         ....]
    [(11) ; id
          ....]
    [(12) ; fun
          (begin
            (set! v-reg (malloc2 16 (code-ref fae-reg 1) ds-reg))
            (continue))]
    [(13) ; app
          ....]
    [(14) ; if0
          ....]))
```

Deallocation

Where does **free** go?

```
; continue : -> void
(define (continue)
  ...
  [(2) ; doAddK
   (begin
     (set! v-reg (num+ (ref k-reg 1) v-reg))
     (free k-reg) ; ???
     (set! k-reg (ref k-reg 2))
     (continue))]
  ...
  [(6) ; doCallK
   (begin
     (set! fae-reg (ref (ref k-reg 1) 1))
     (set! ds-reg (malloc2 17
                           v-reg
                           (ref (ref k-reg 1) 2)))
     (set! k-reg (ref k-reg 2))
     (free fun-val) ; ???
     (interp))]
```

Deallocation

```
[(2) ; doAddK
 (begin
   (set! v-reg (num+ (ref k-reg 1) v-reg))
   (free k-reg) ; ???
   (set! k-reg (ref k-reg 2))
   (continue))]
```

- For simple cases, freeing local storage right after use is fine, which is why most languages use a stack

Deallocation

```
[(6) ; doCallK
 (begin
   (set! fae-reg (ref (ref k-reg 1) 1))
   (set! ds-reg (malloc2 17
                         v-reg
                         (ref (ref k-reg 1) 2)))
   (set! k-reg (ref k-reg 2))
   (free fun-val) ; ???
   (interp))]
```

- This free is *not* ok, because the closure might be kept in a substitution somewhere
- Need to free only if no one else is using it...

Reference Counting

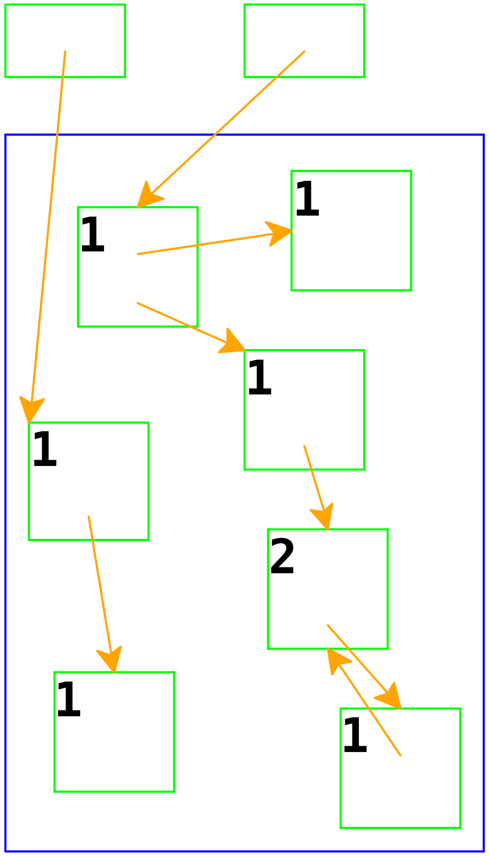
Reference counting: a way to know whether a record has other users

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 (into a register or another record), increment its count
- When replacing a pointer to a record, decrement its count
- When a count is decremented to 0, decrement counts for other records referenced by the record, then free it

Reference Counting

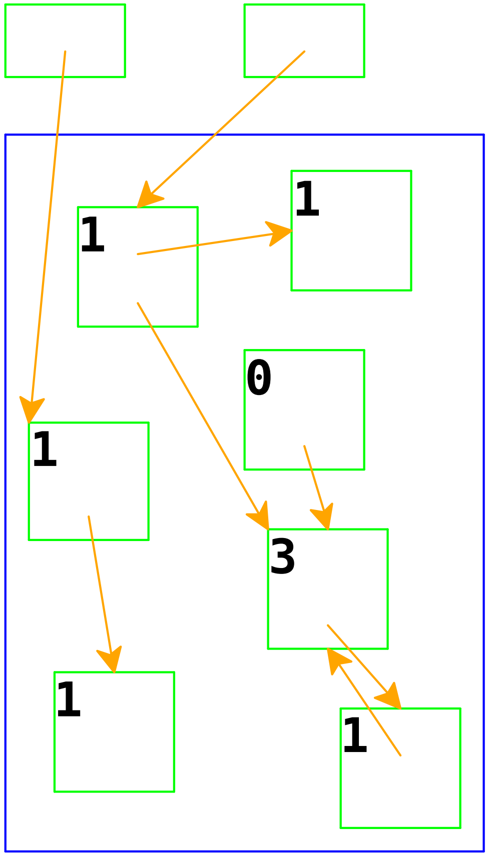


Top boxes are the registers (roots)

`fae-reg`, **`k-reg`**, etc.

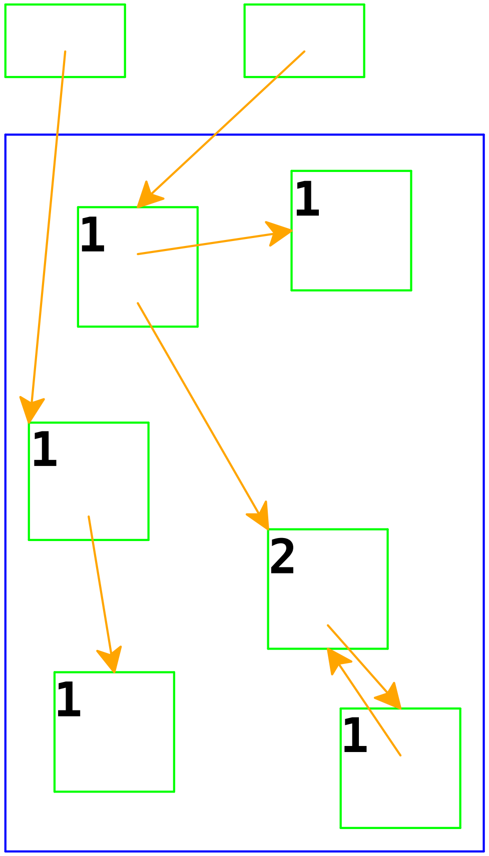
Boxes in the blue area are allocated with **`malloc`**

Reference Counting



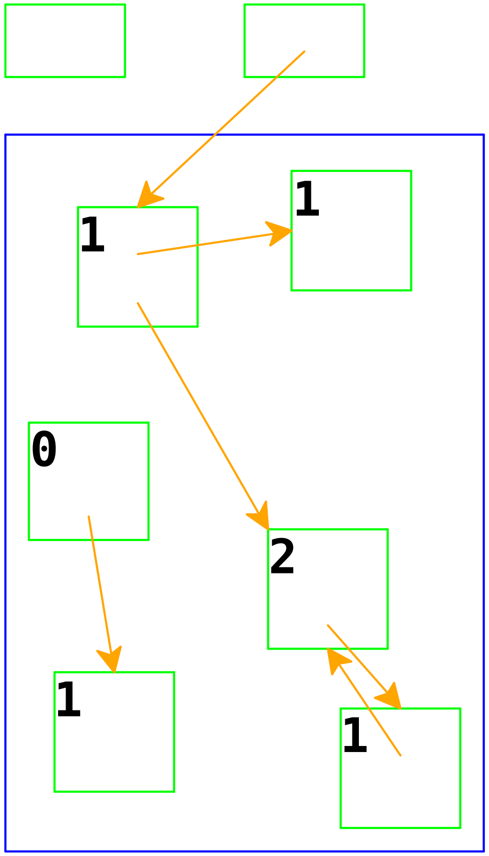
Adjust counts when a pointer is changed...

Reference Counting



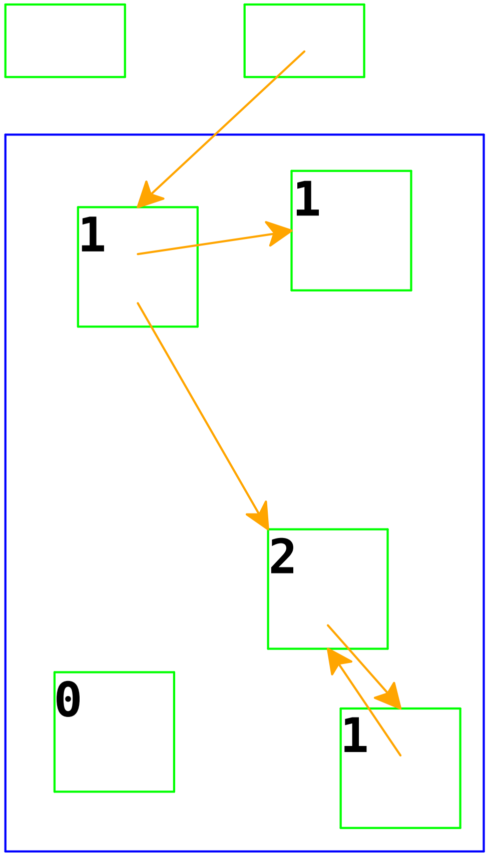
... freeing a record if its count goes to 0

Reference Counting



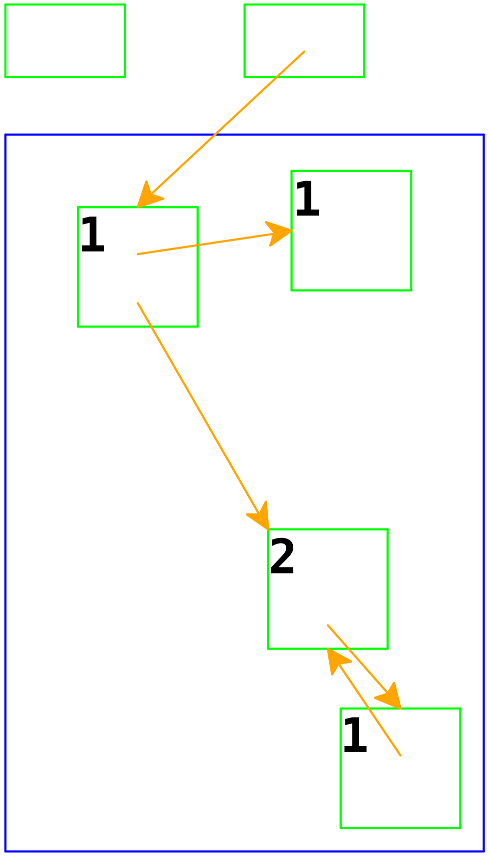
Same if the pointer is in a register

Reference Counting



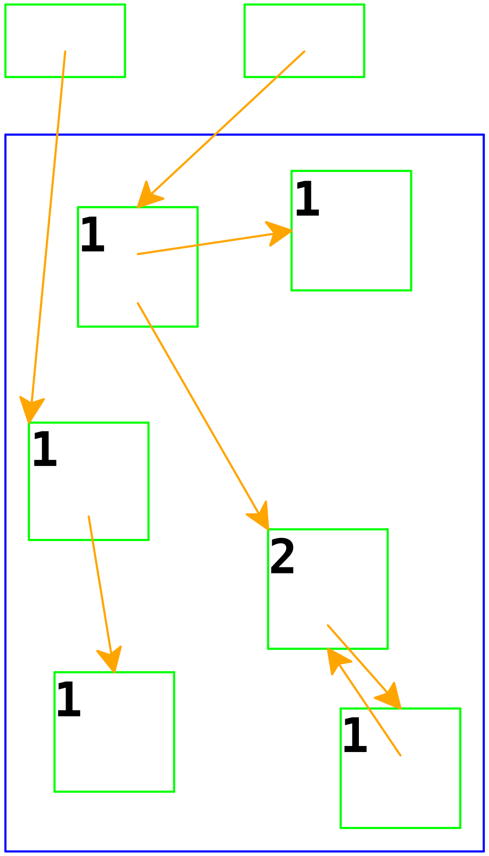
Adjust counts after frees, too...

Reference Counting



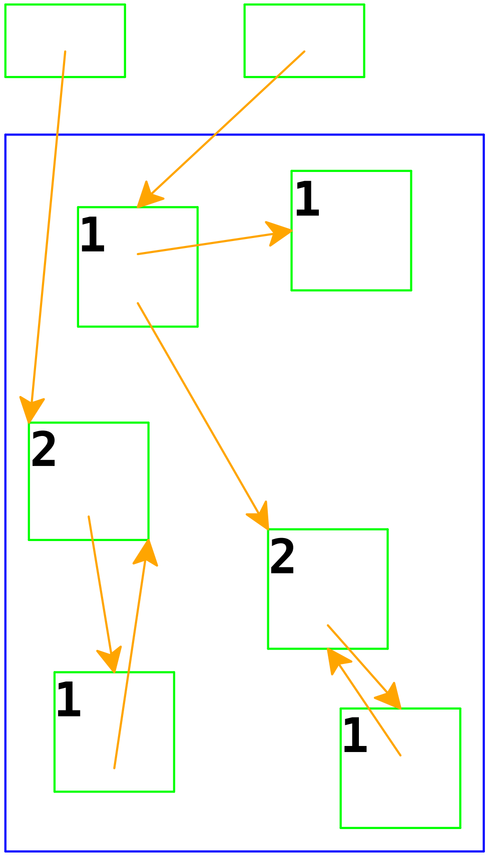
... which can trigger more frees

Reference Counting And Cycles



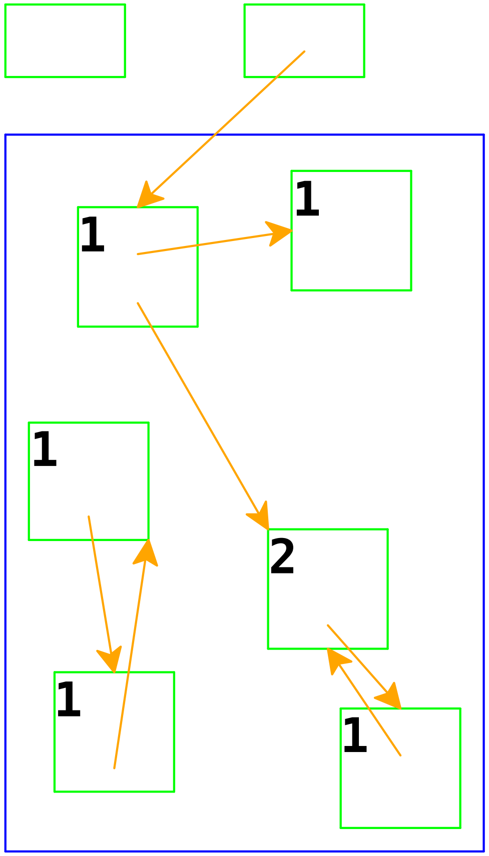
An assignment can create a cycle...

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

Garbage Collection

Garbage collection: a way to know whether a record is *accessible*

Garbage Collection

Garbage collection: a way to know whether a record is *accessible*

- A record referenced by a register is ***live***
- A record referenced by a live record is also live
- A program can only possibly use live records, because there is no way to get to other records

Garbage Collection

Garbage collection: a way to know whether a record is *accessible*

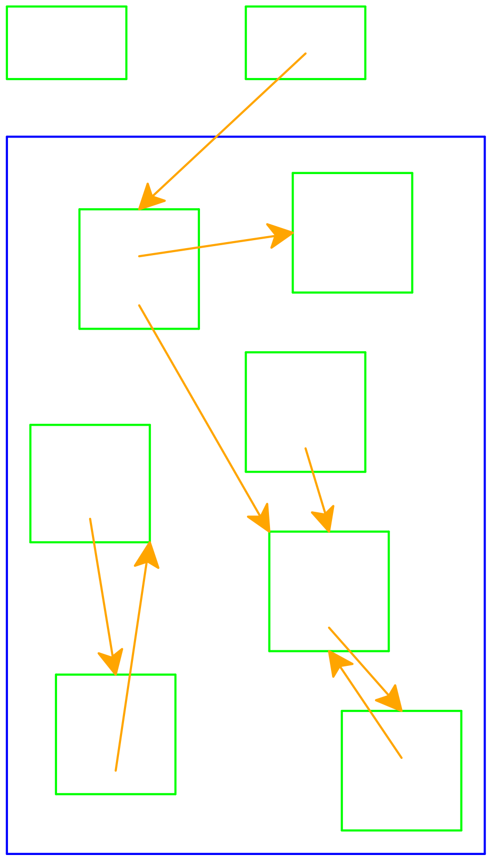
- A record referenced by a register is ***live***
- A record referenced by a live record is also live
- A program can only possibly use live records, because there is no way to get to other records

- A garbage collector frees all records that are not live
- Allocate until we run out of memory, then run a garbage collector to get more space

Garbage Collection Algorithm

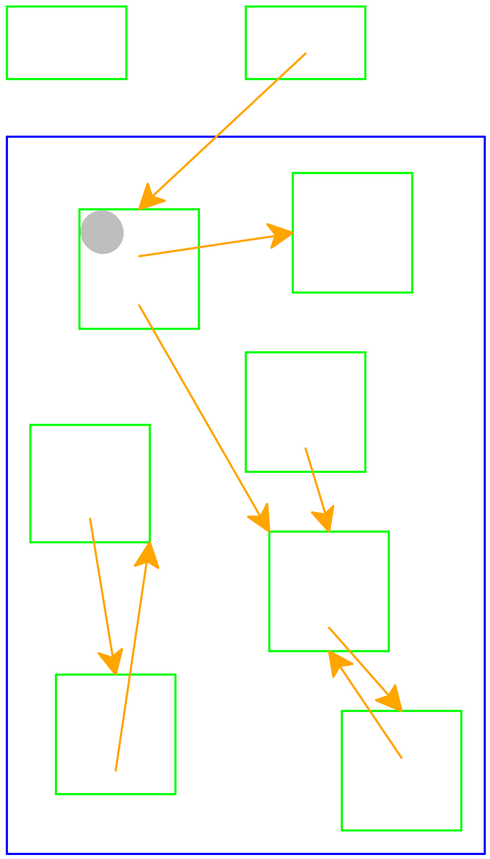
- Color all records ***white***
- Color records referenced by registers ***gray***
- Repeat until there are no gray records:
    - Pick a gray record, *r*
    - For each white record that *r* points to, make it gray
    - Color *r* ***black***
- Deallocate all white records

Garbage Collection



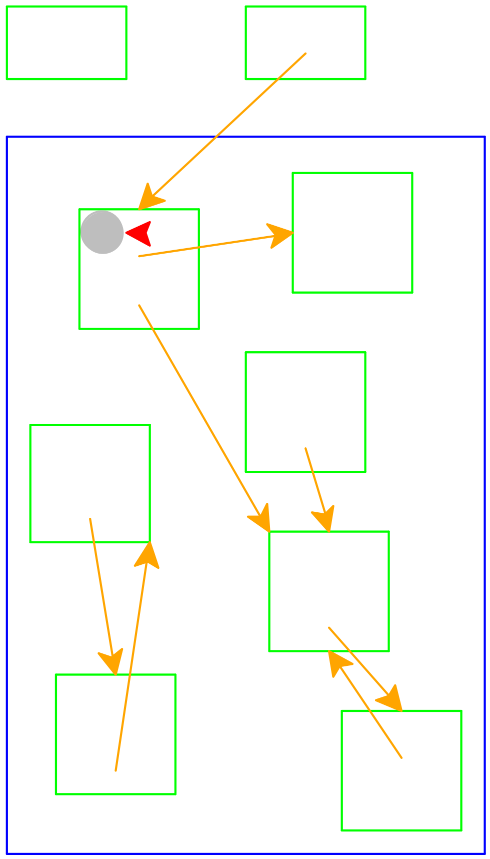
All records are marked white

Garbage Collection



Mark records referenced by registers as gray

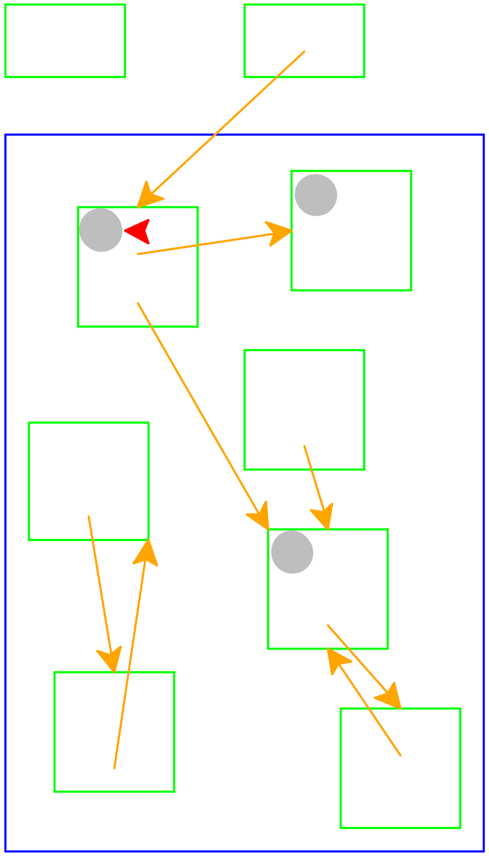
Garbage Collection



Need to pick a gray record

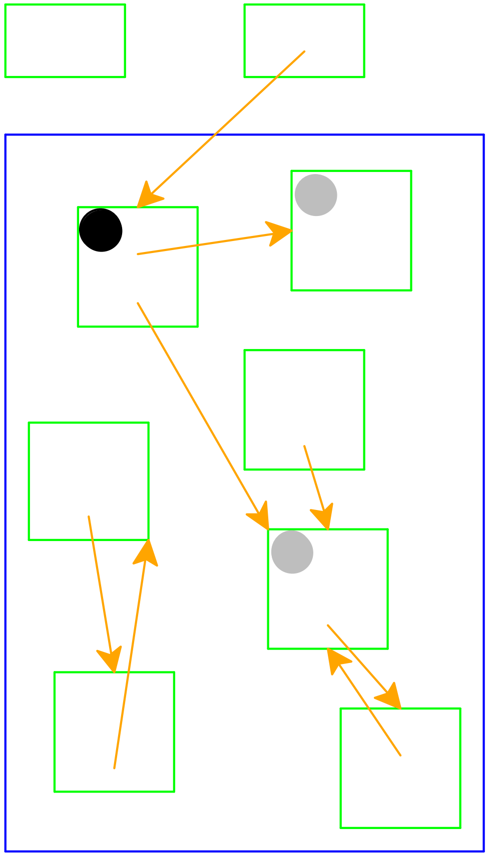
Red arrow indicates the chosen record

Garbage Collection



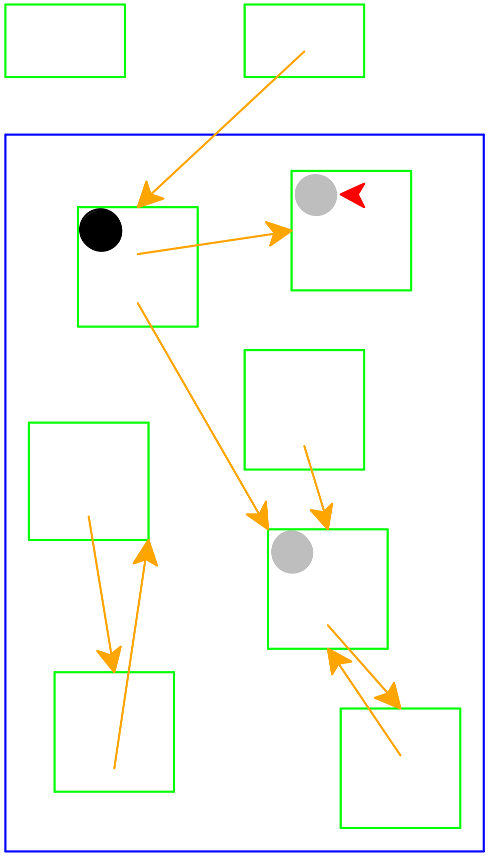
Mark white records referenced by chosen record as gray

Garbage Collection



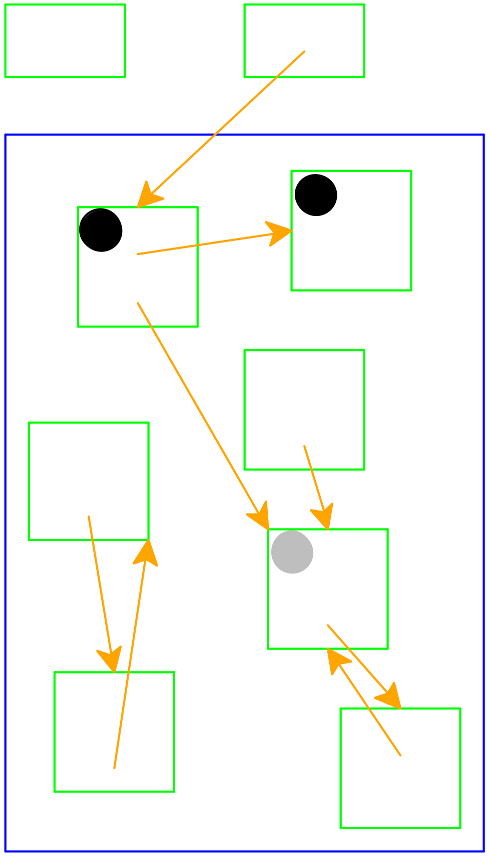
Mark chosen record black

Garbage Collection



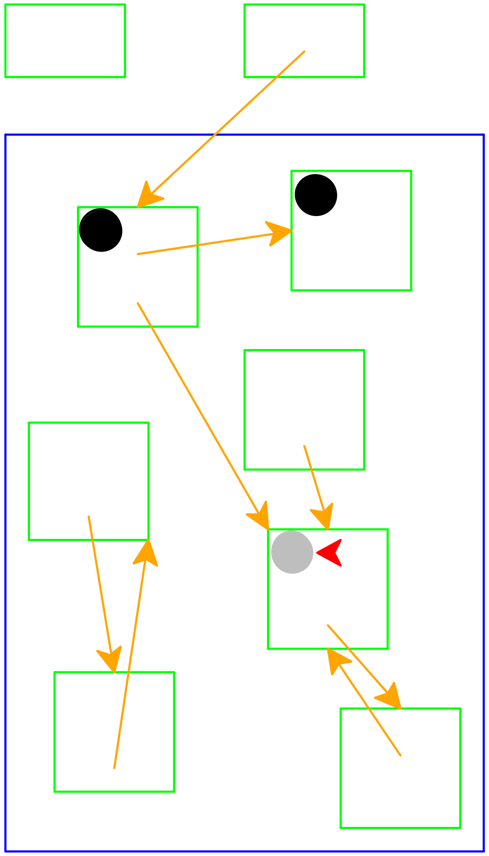
Start again: pick a gray record

Garbage Collection



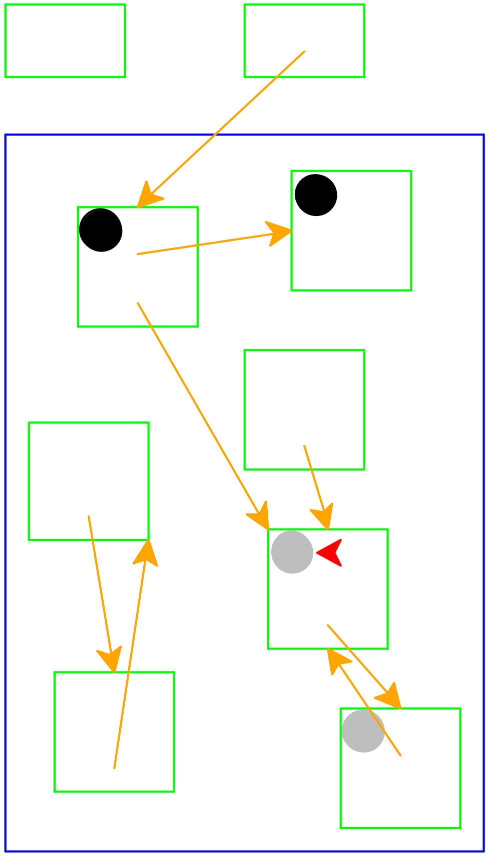
No referenced records; mark black

Garbage Collection



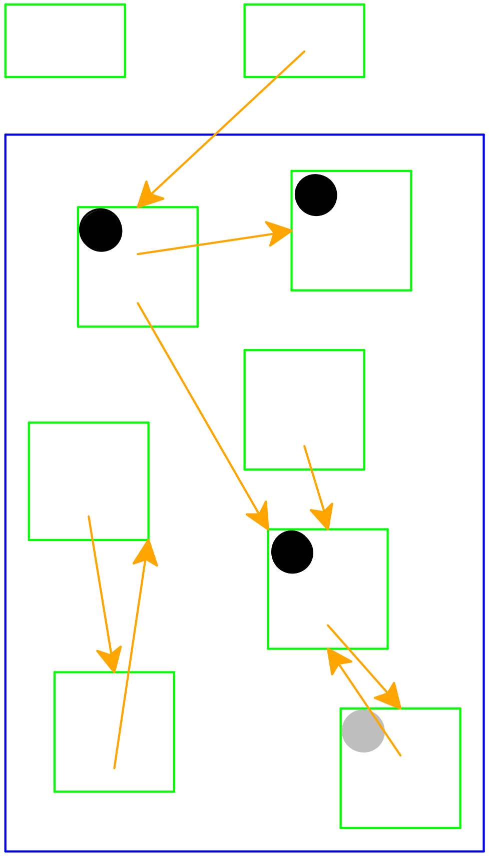
Start again: pick a gray record

Garbage Collection



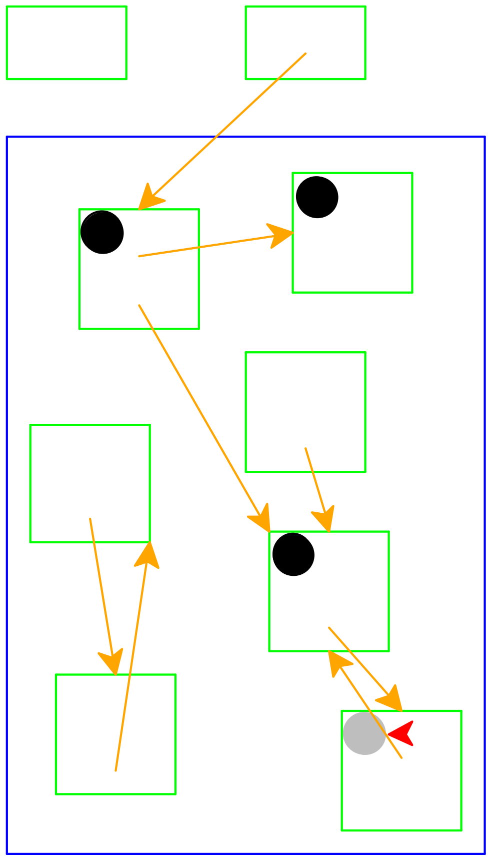
Mark white records referenced by chosen record as gray

Garbage Collection



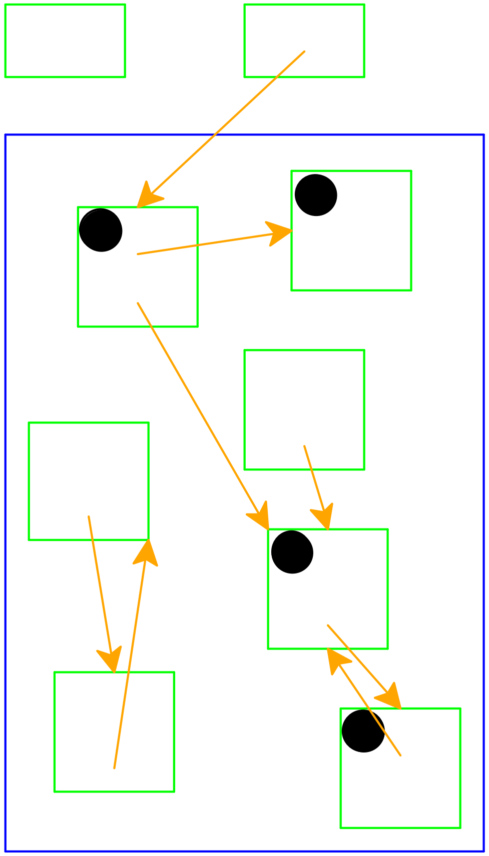
Mark chosen record black

Garbage Collection



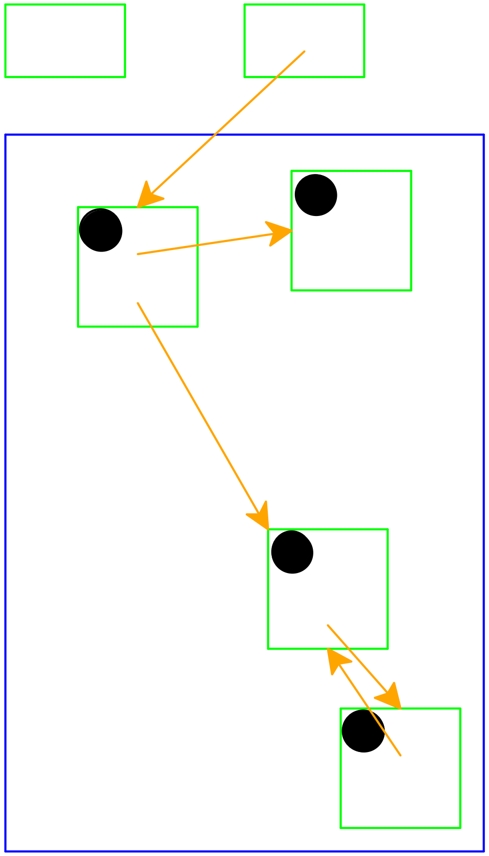
Start again: pick a gray record

Garbage Collection



No referenced white records; mark black

Garbage Collection



No more gray records; deallocate white records

Cycles ***do not*** break garbage collection

Two-Space Copying Collectors

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

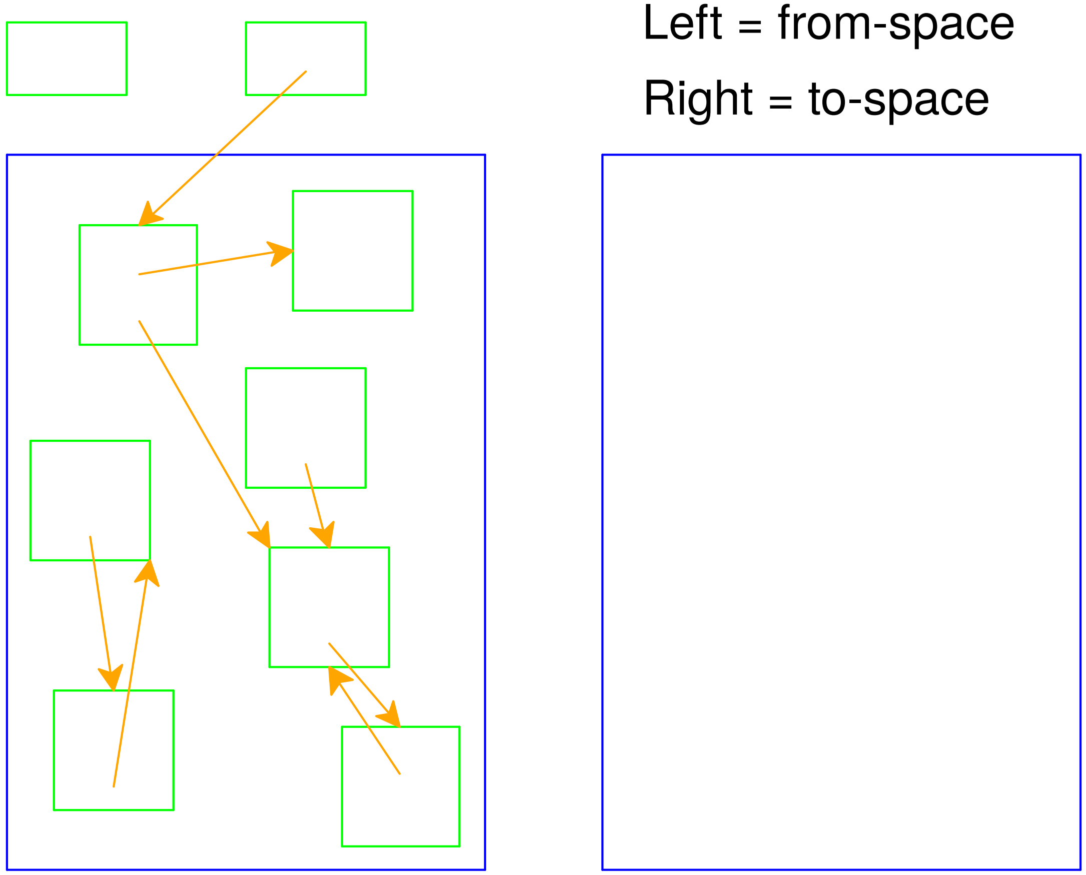
Allocator:

- 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

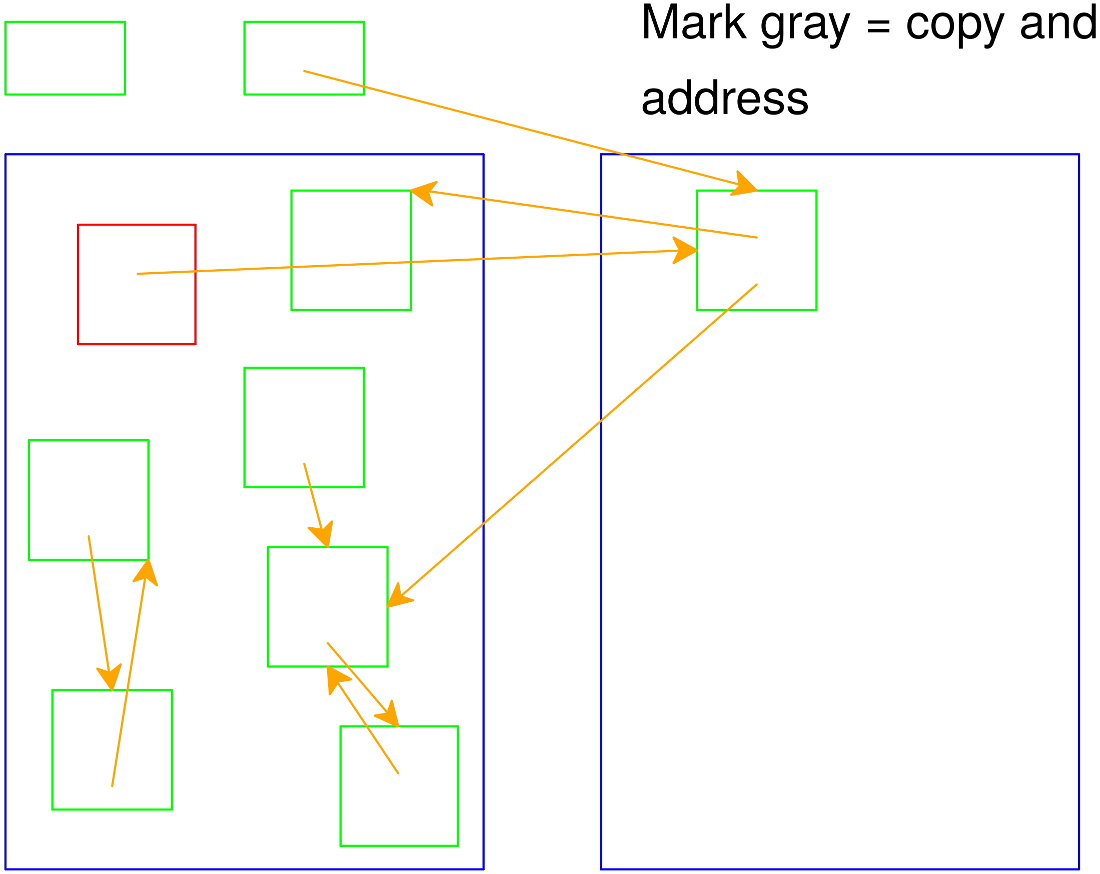


Left = from-space

Right = to-space

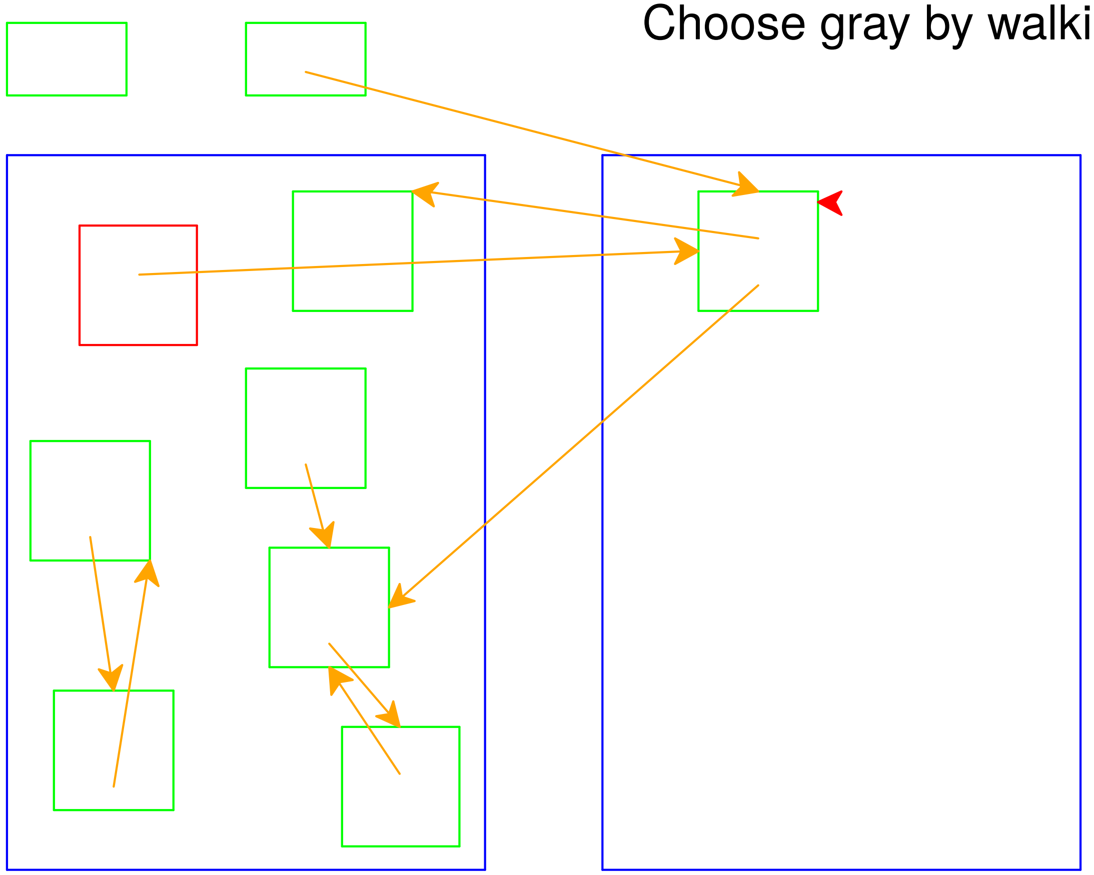
Two-Space Collection

Mark gray = copy and leave forward address

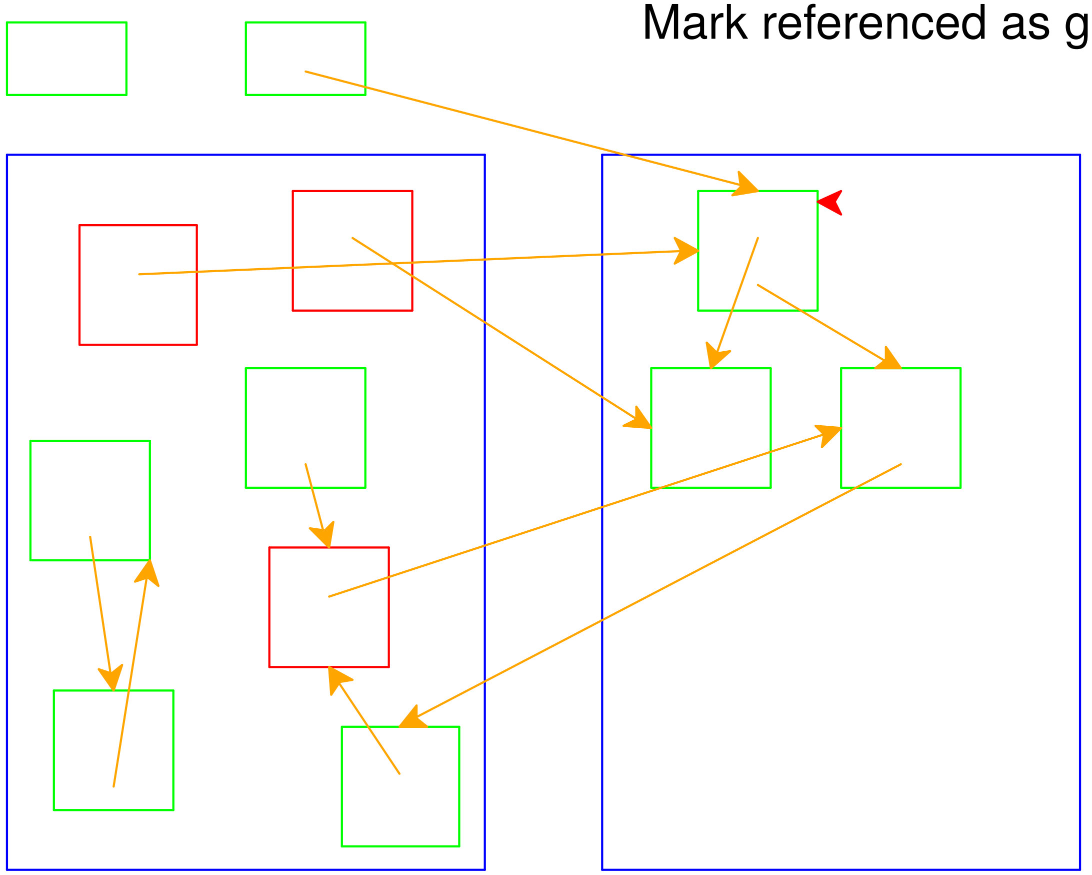


Two-Space Collection

Choose gray by walking through to-space

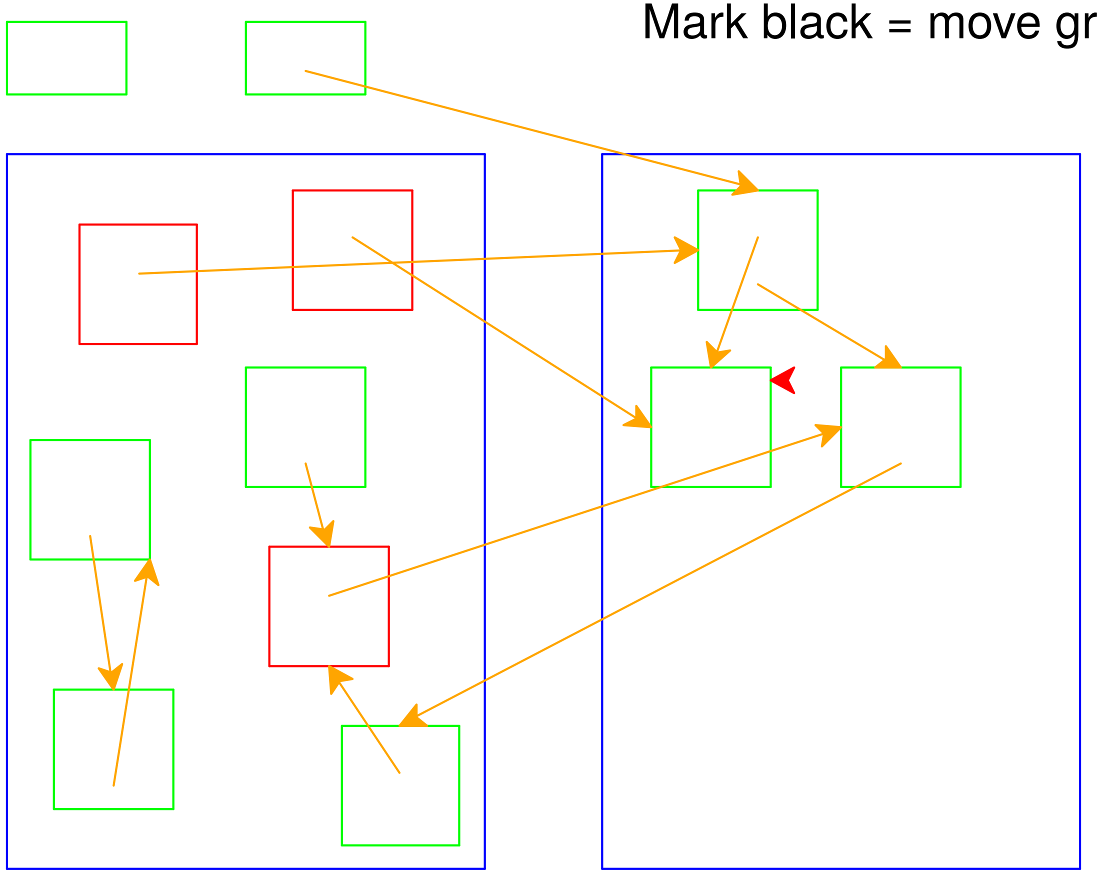


Two-Space Collection



Mark referenced as gray

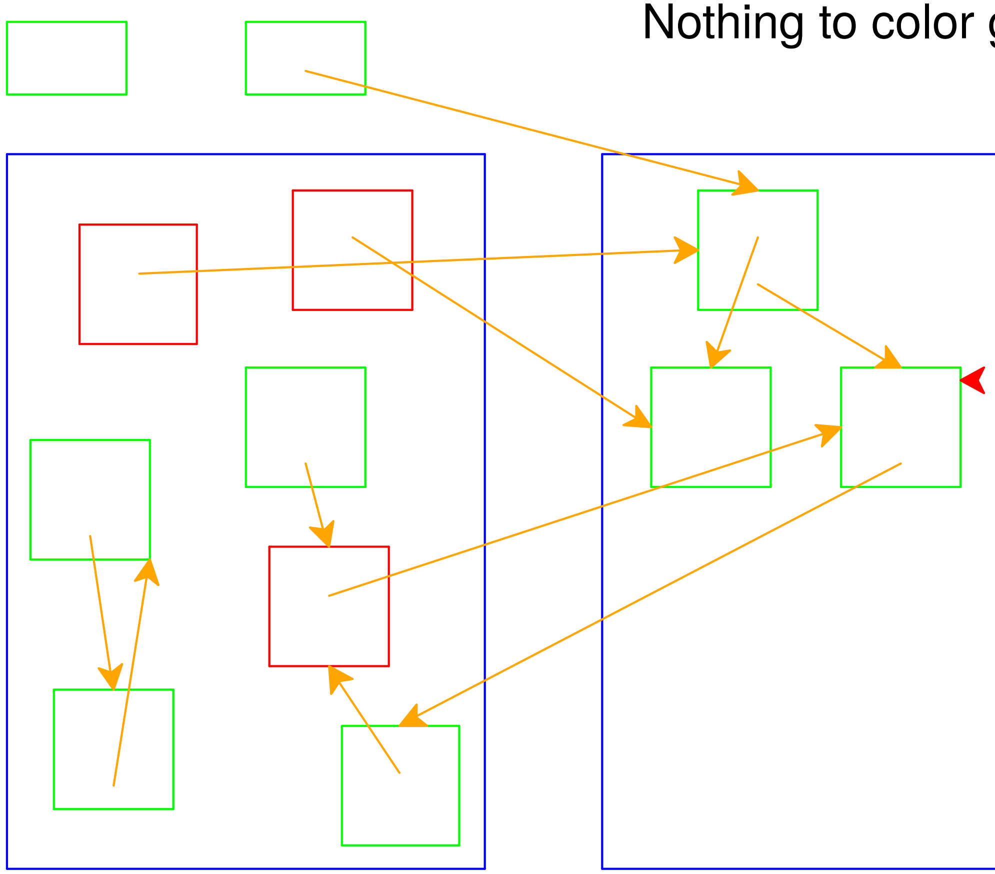
Two-Space Collection



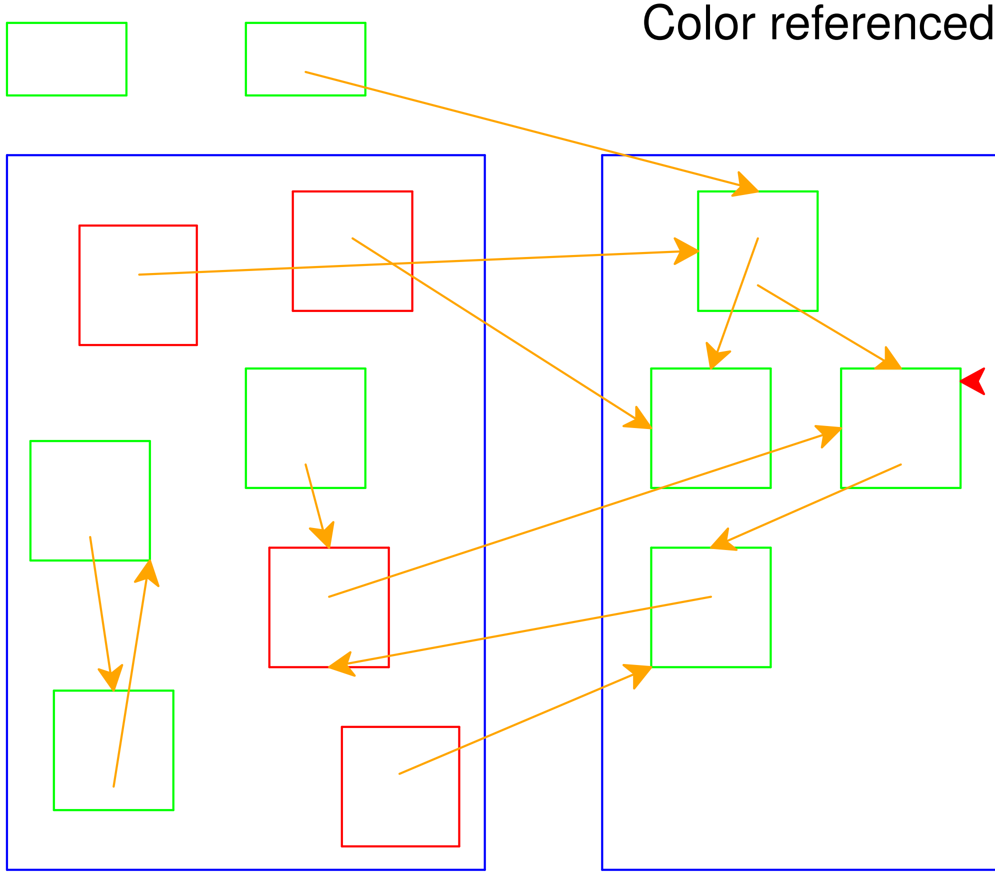
Mark black = move gray-choosing arrow

Two-Space Collection

Nothing to color gray; increment the arrow



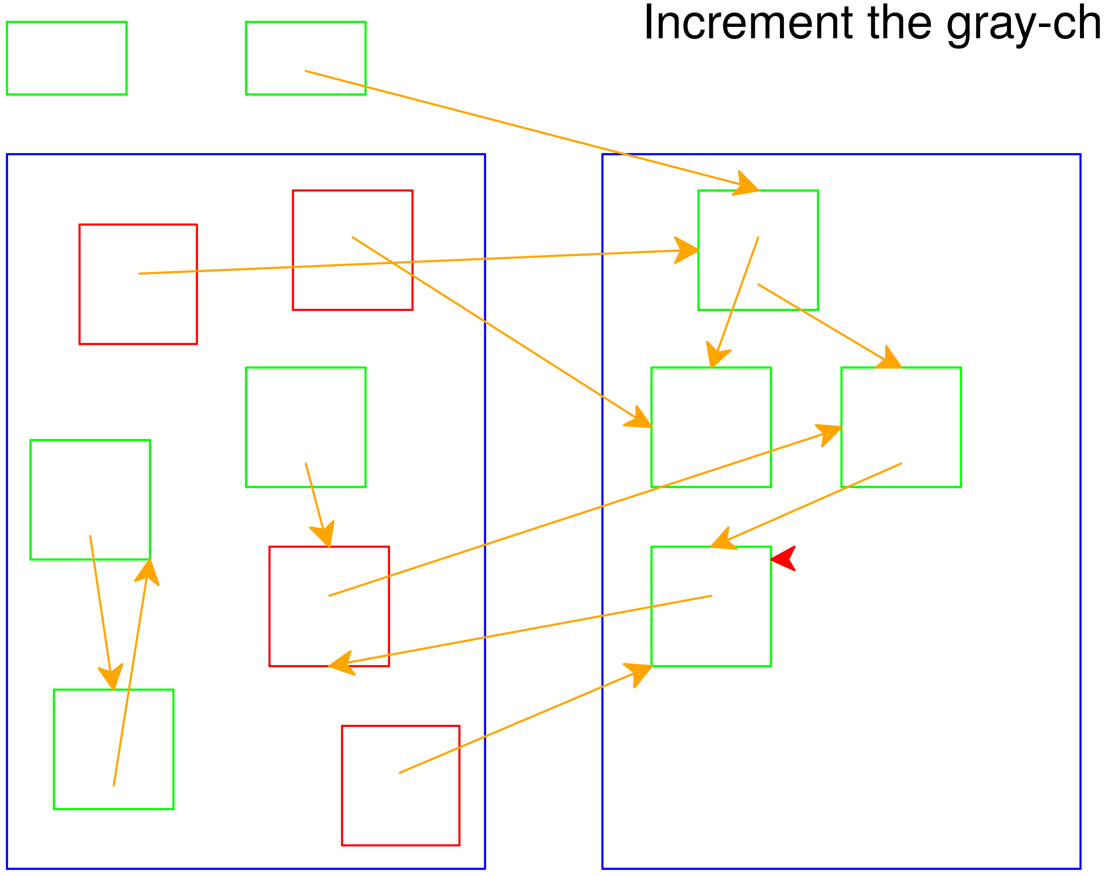
Two-Space Collection



Color referenced record gray

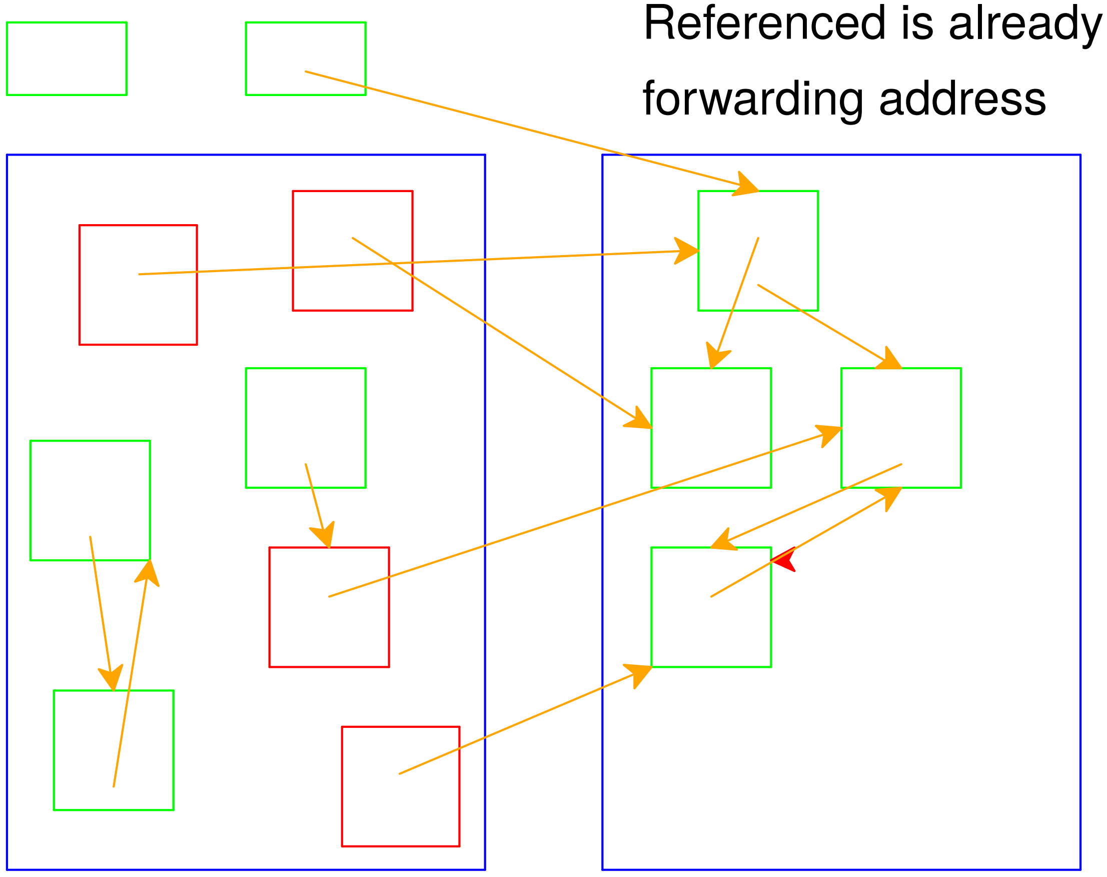
Two-Space Collection

Increment the gray-choosing arrow



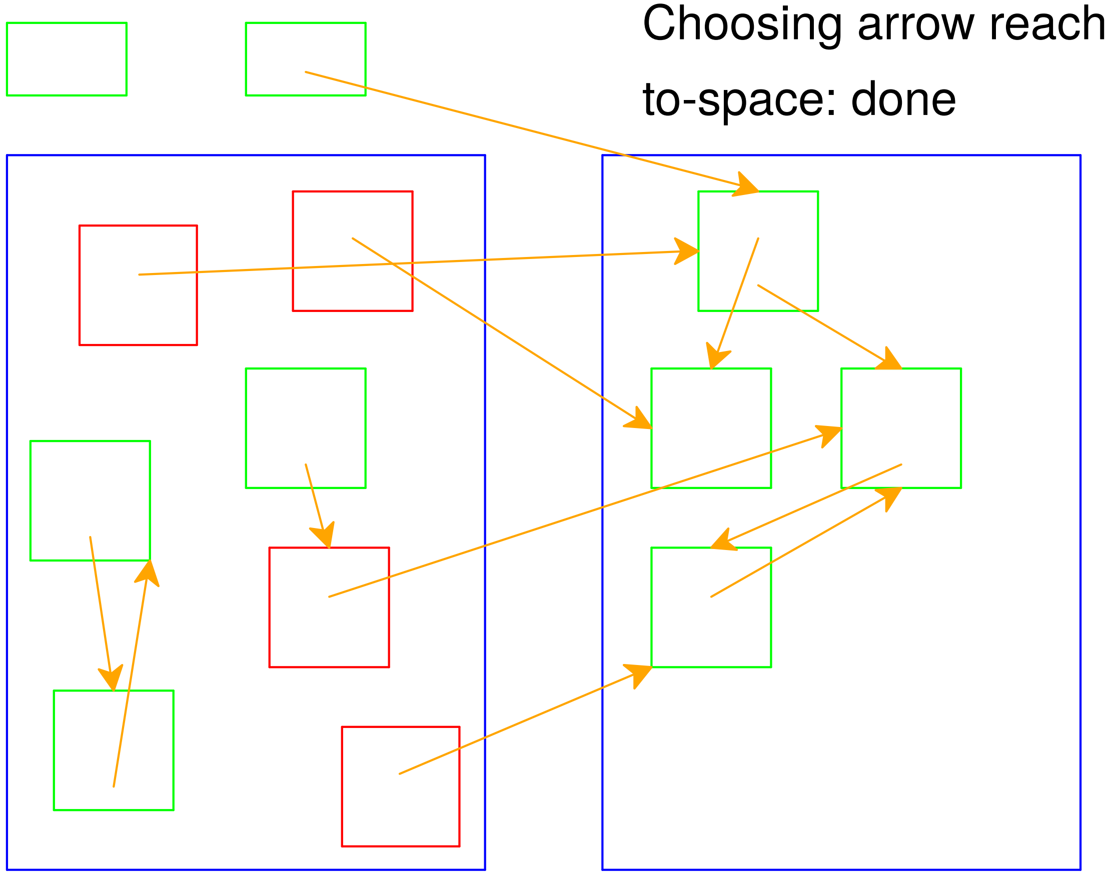
Two-Space Collection

Referenced is already copied, use forwarding address

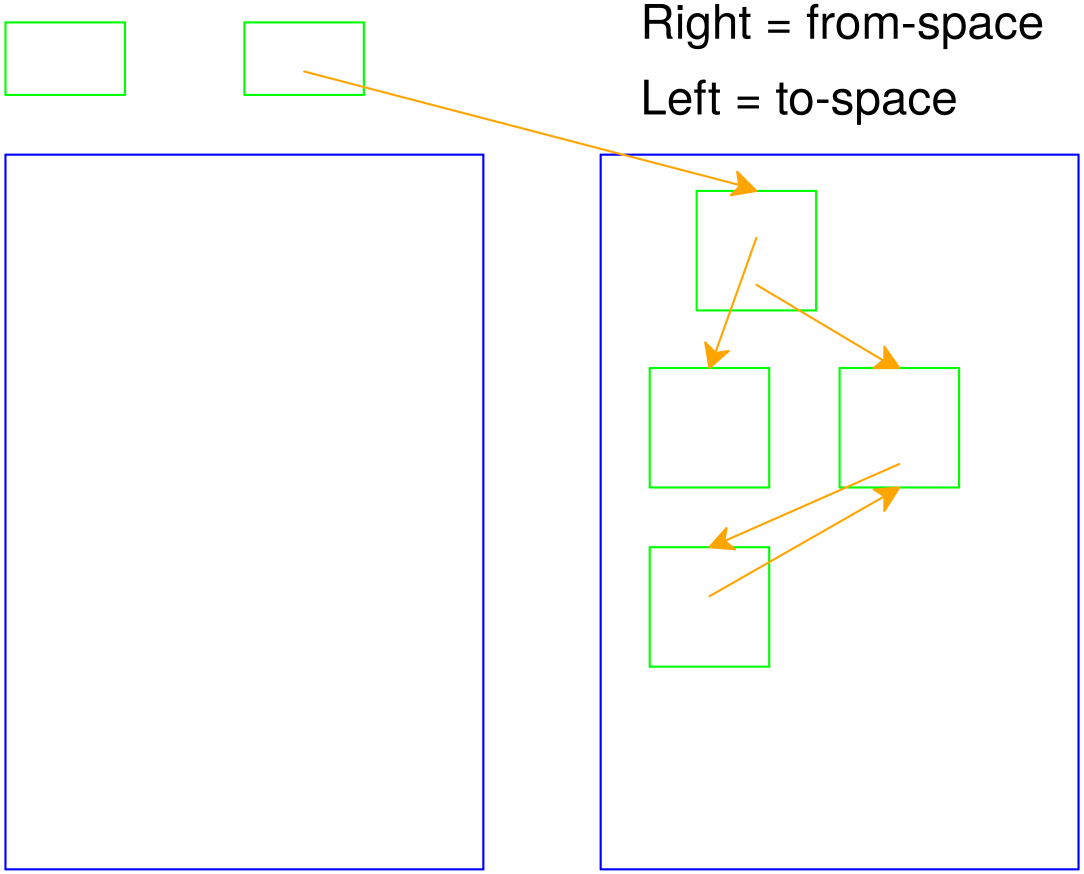


Two-Space Collection

Choosing arrow reaches the end of to-space: done



Two-Space Collection



Right = from-space

Left = to-space

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 Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

Register 1: **7** Register 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 for each space), 2 registers
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

Register 1: **7** Register 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 for each space), 2 registers
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

Register 1: **7**   Register 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 for each space), 2 registers
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

Register 1: **7**   Register 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
        ^
```

Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

Register 1: 0 Register 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
	^												

Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

```
        Register 1: 0        Register 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
        ^
```

Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

Register 1: 0 Register 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
				^									

Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

Register 1: 0   Register 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
                        ^
```

Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

Register 1: 0 Register 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
								^					