Copying
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*** (in order, c.f. non-collecting)

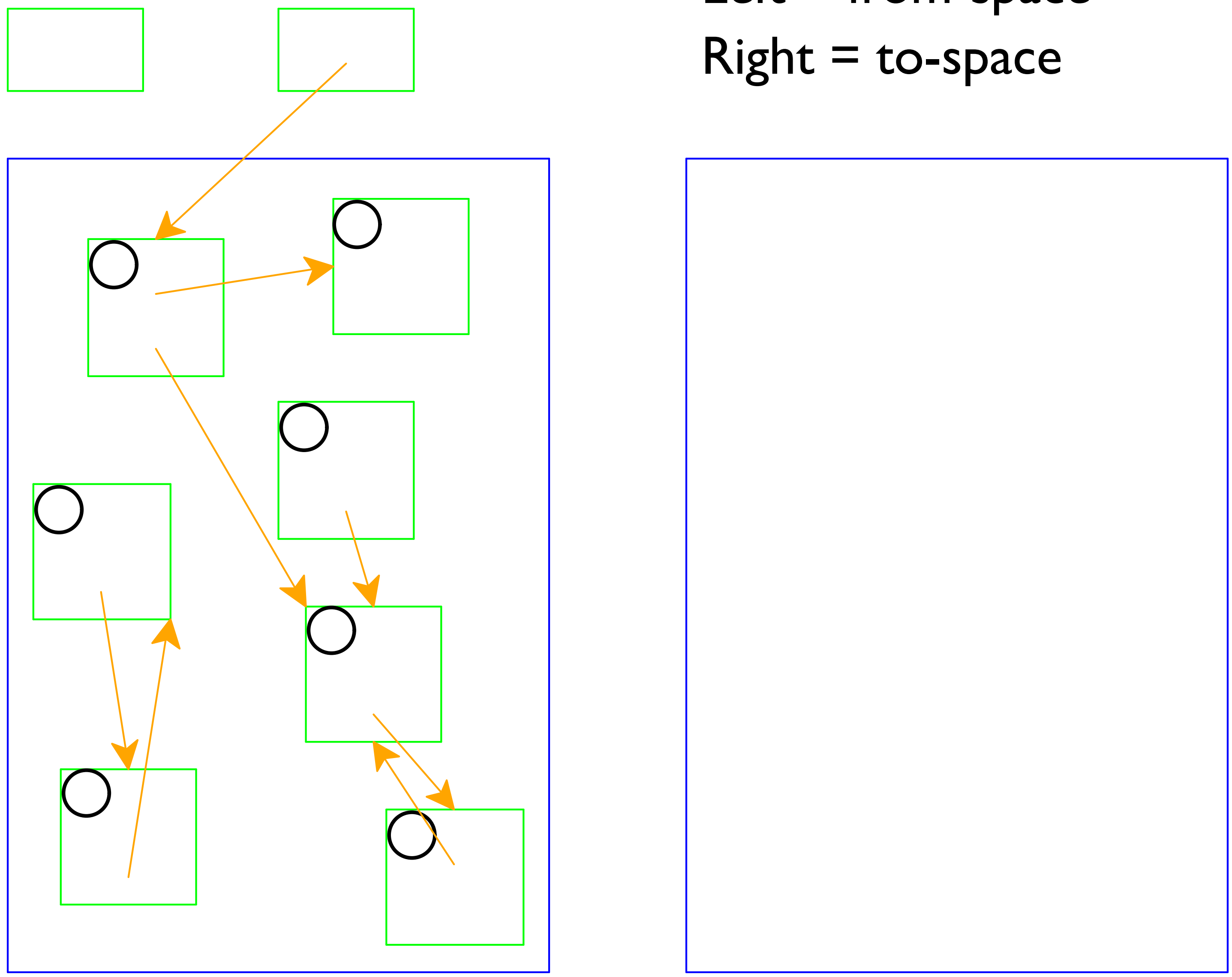
Collector:

- Starts by swapping ***to-space*** and ***from-space***
- Coloring gray ⇒ copy from ***from-space*** to ***to-space***
- Choosing gray records ⇒ go through the new ***to-space***, update pointers

Two-Space Collection

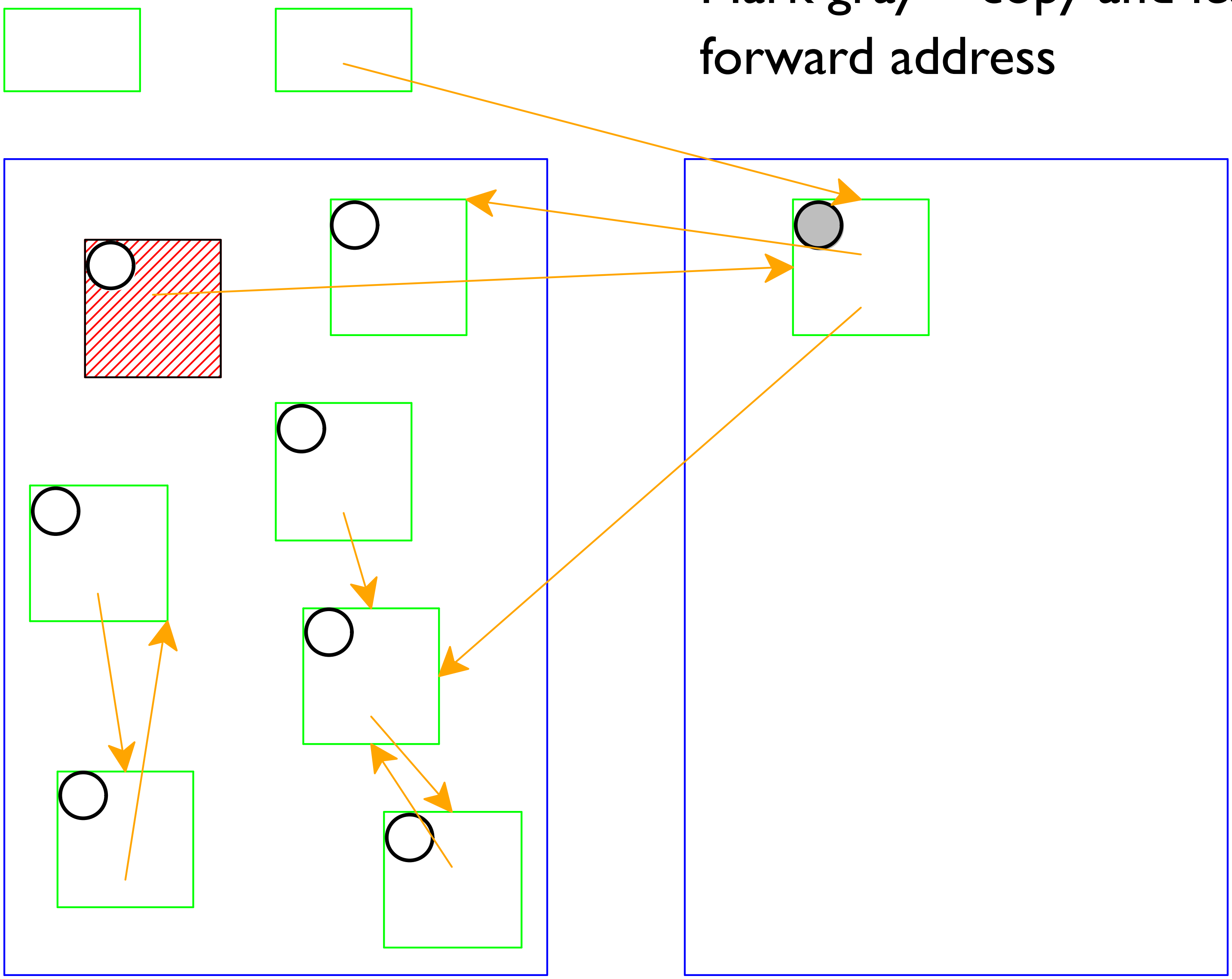
Left = from-space

Right = to-space



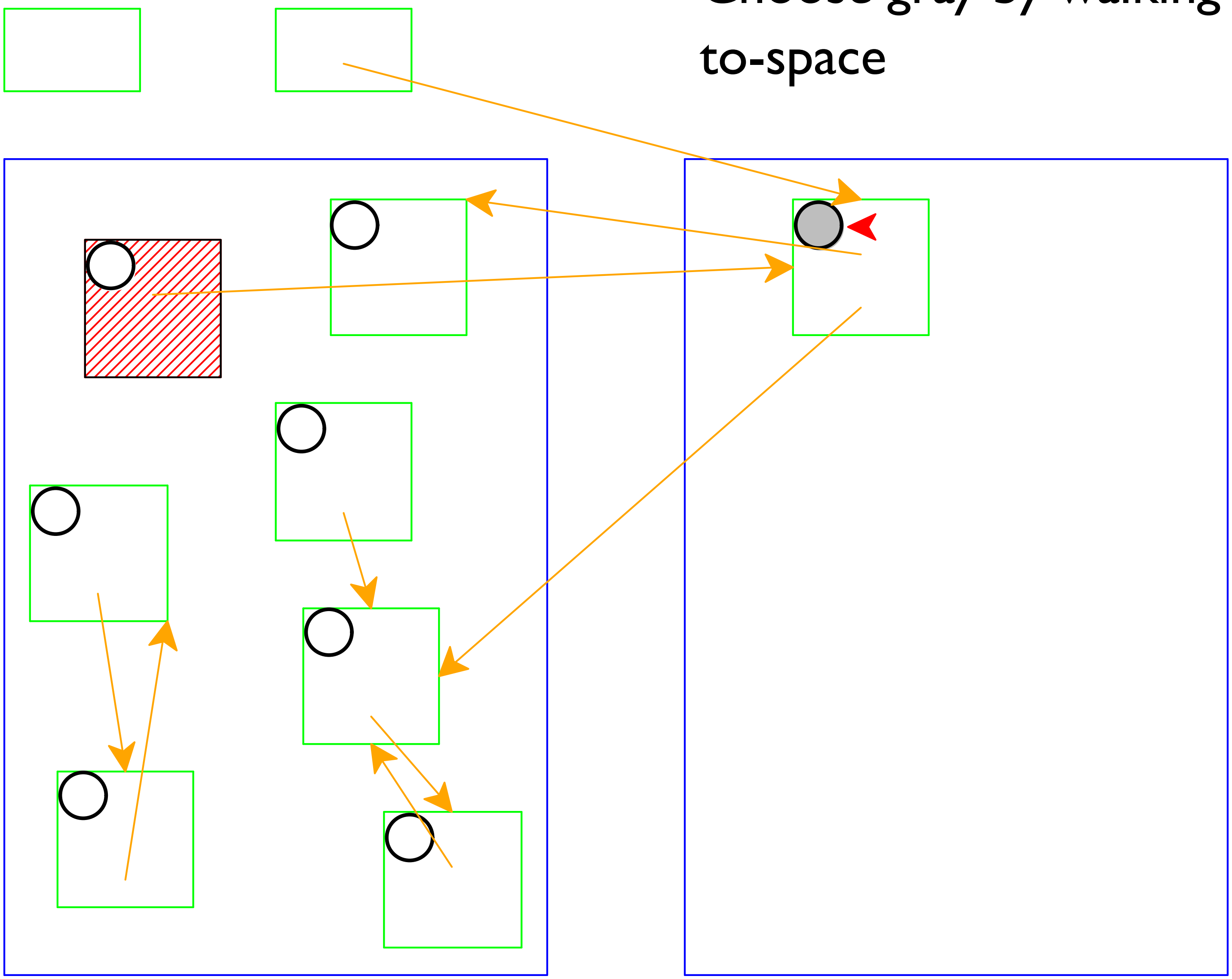
Two-Space Collection

Mark gray = copy and leave forward address



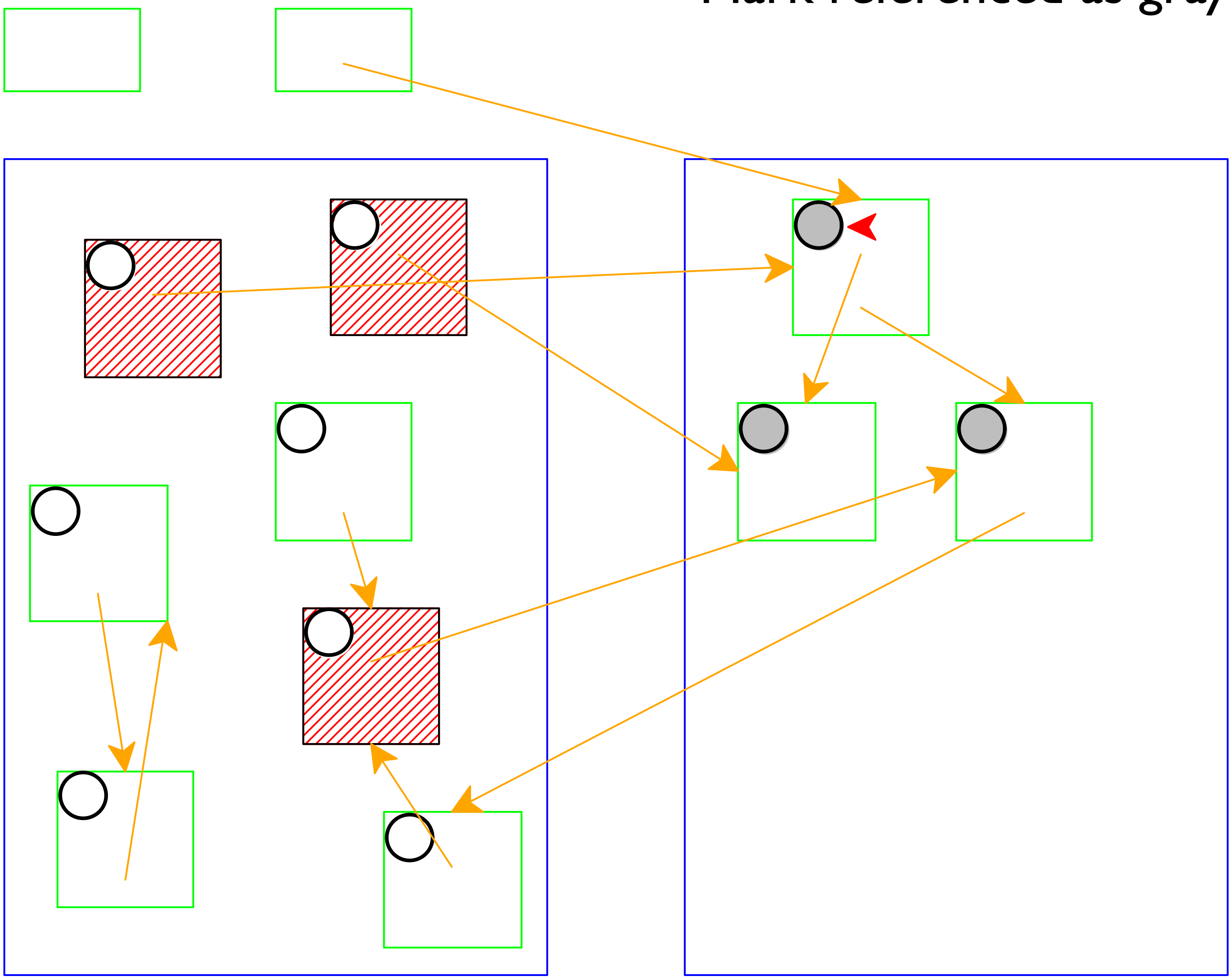
Two-Space Collection

Choose gray by walking through to-space



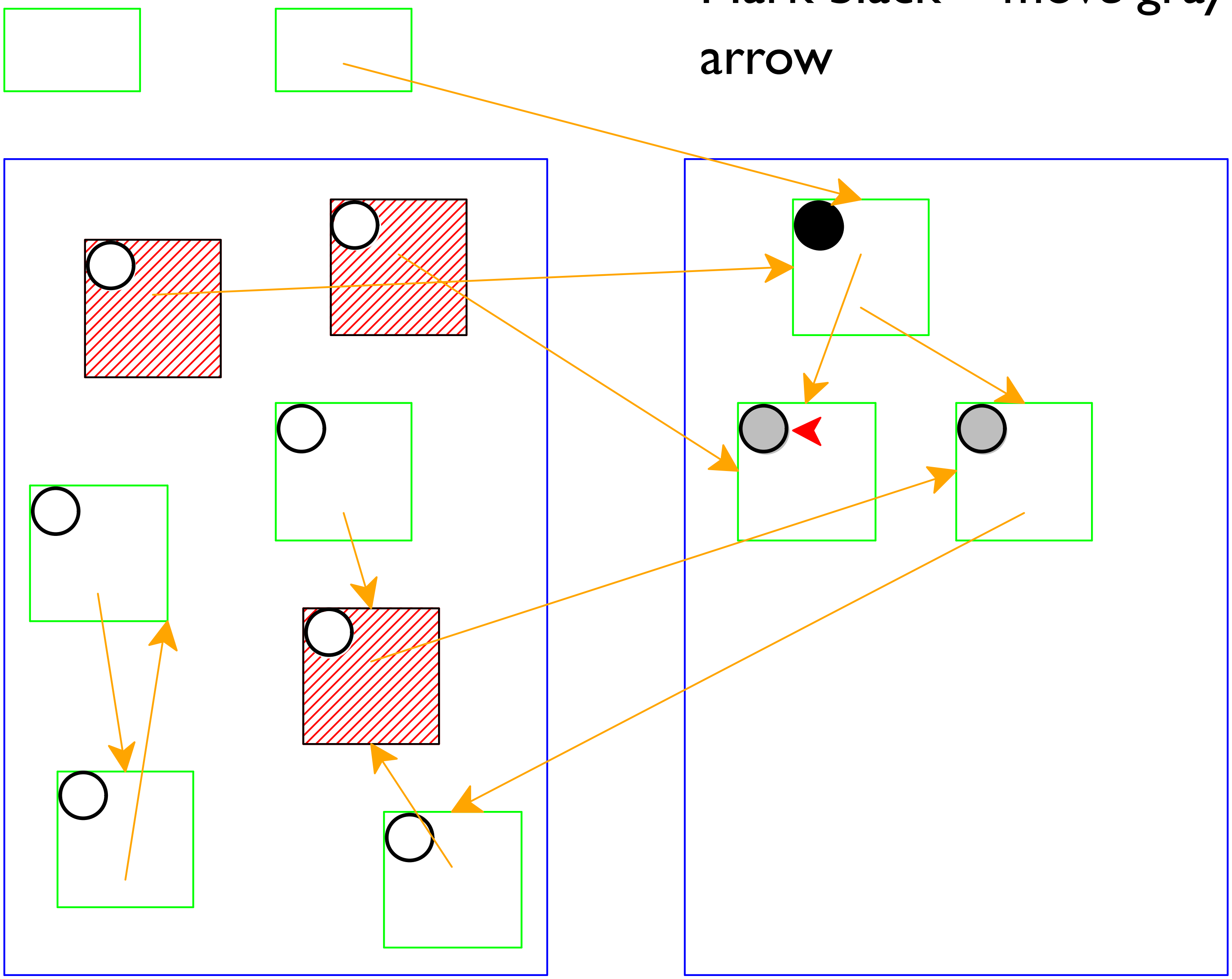
Two-Space Collection

Mark referenced as gray



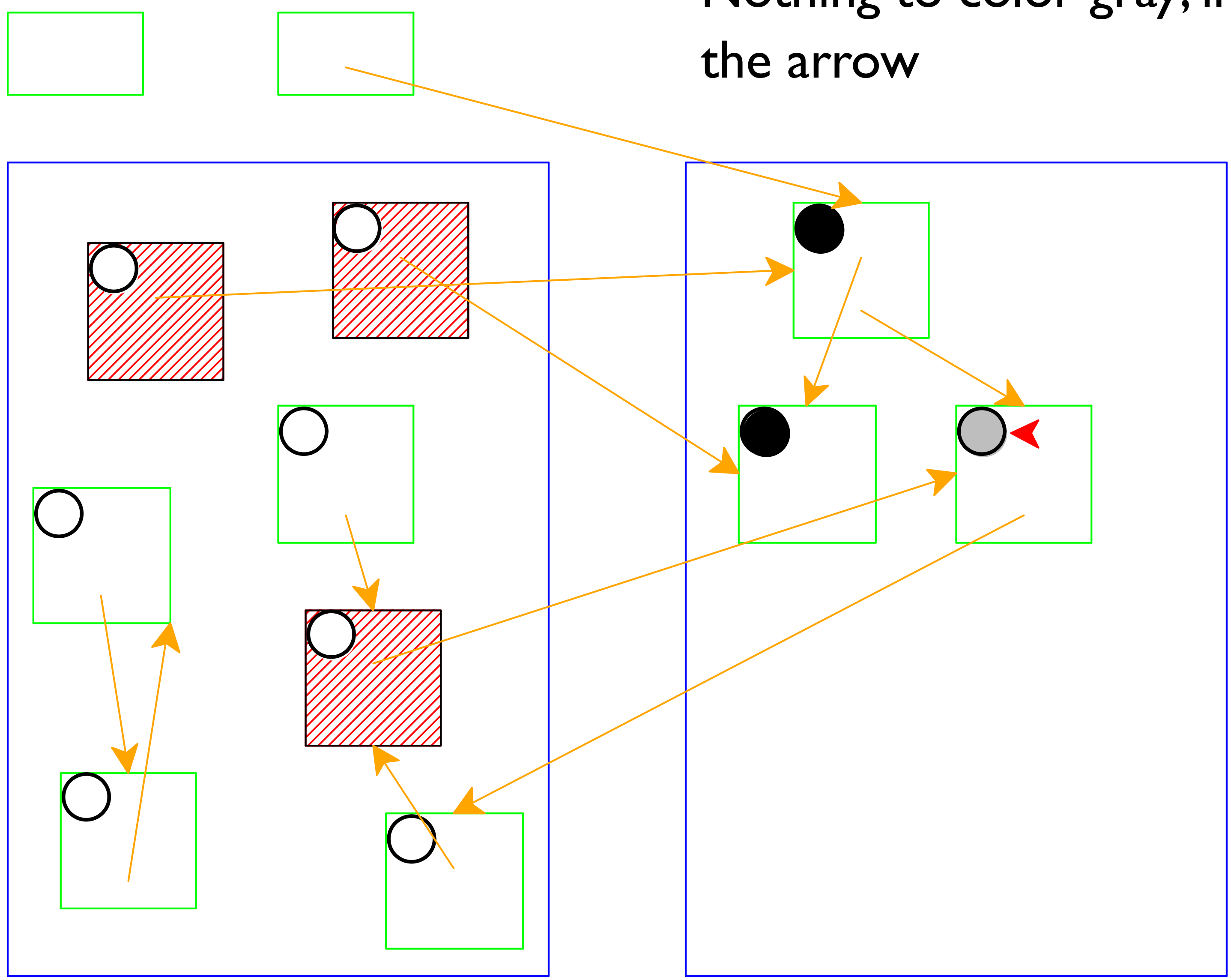
Two-Space Collection

Mark black = move gray-choosing arrow



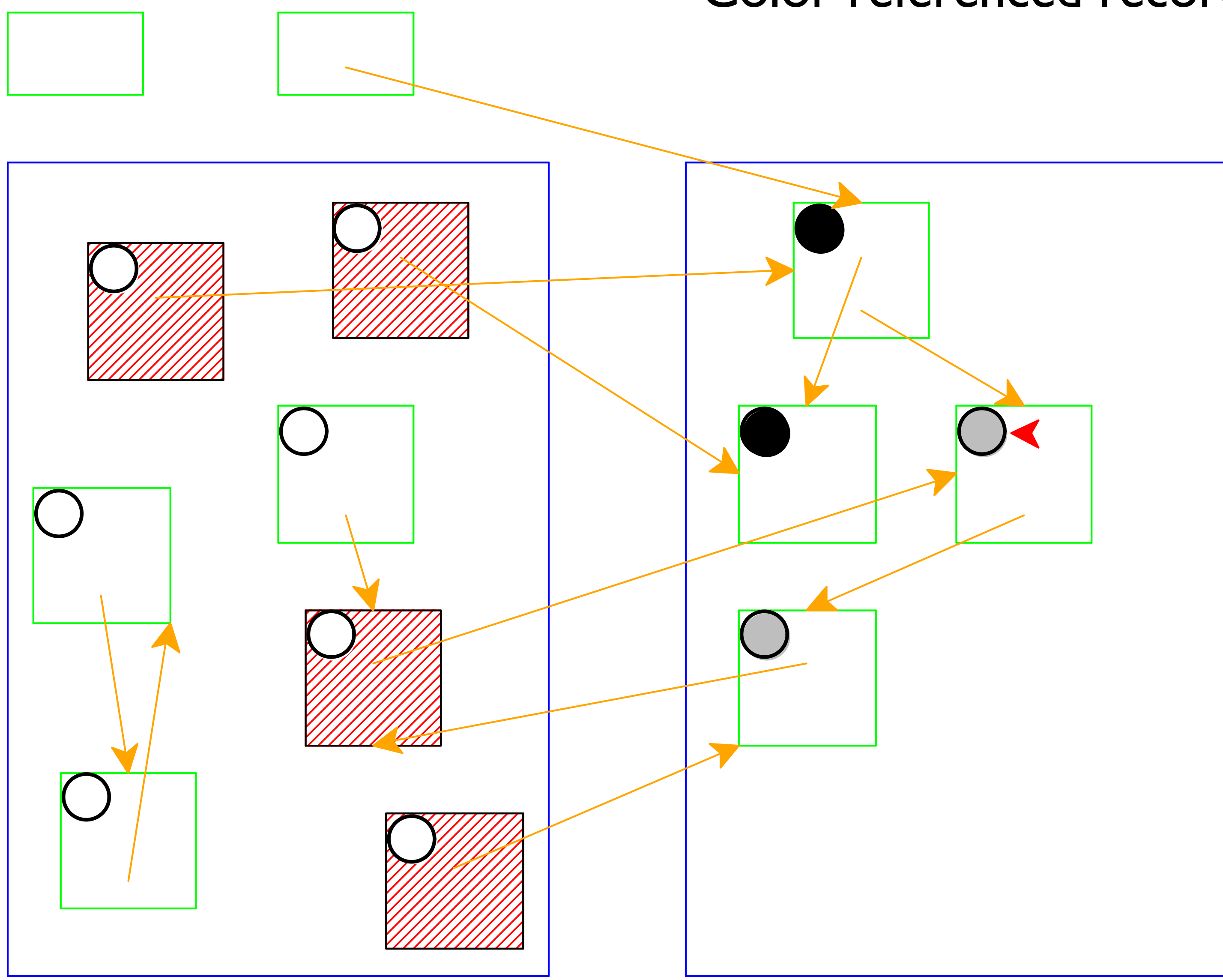
Two-Space Collection

Nothing to color gray; increment the arrow



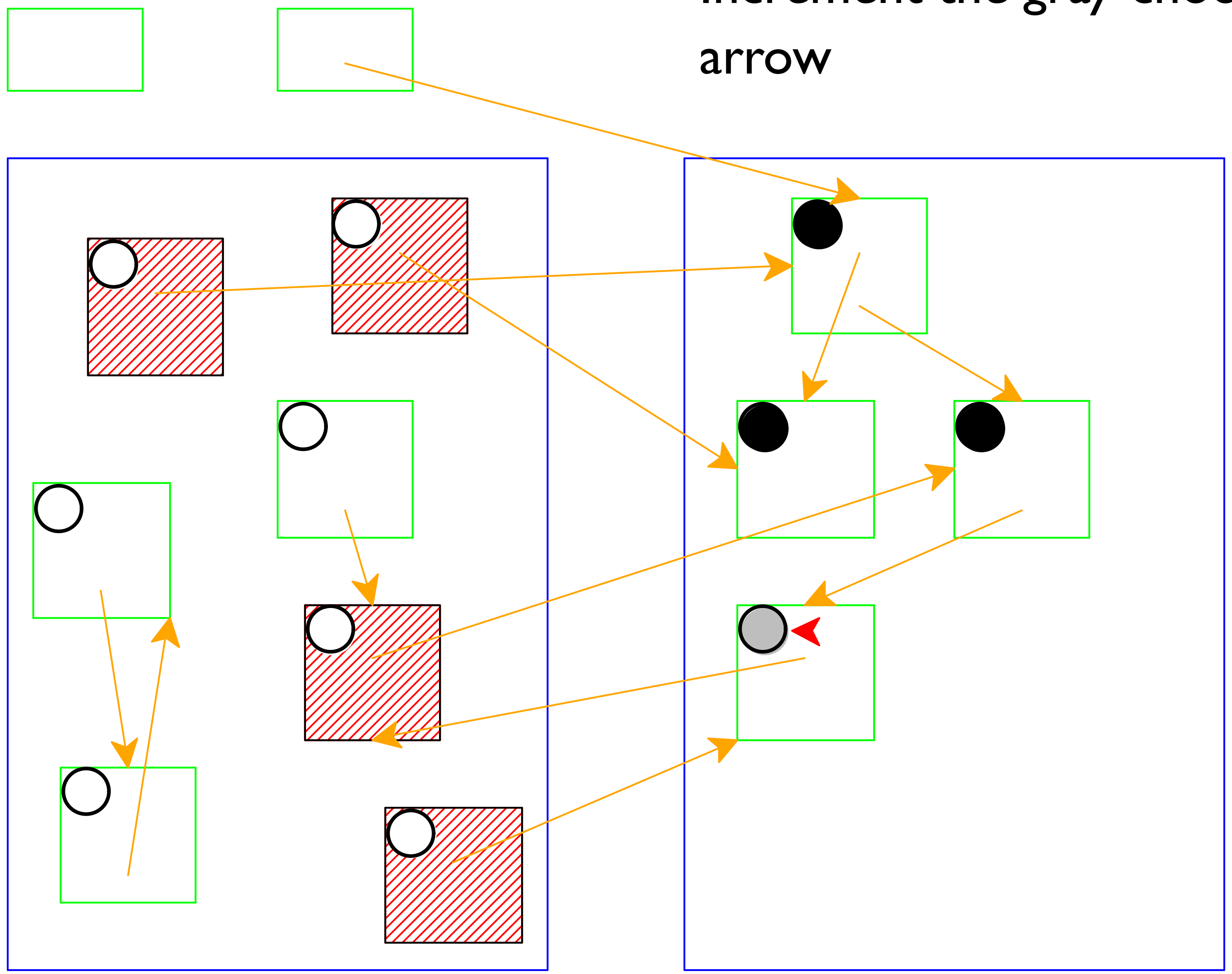
Two-Space Collection

Color referenced record gray



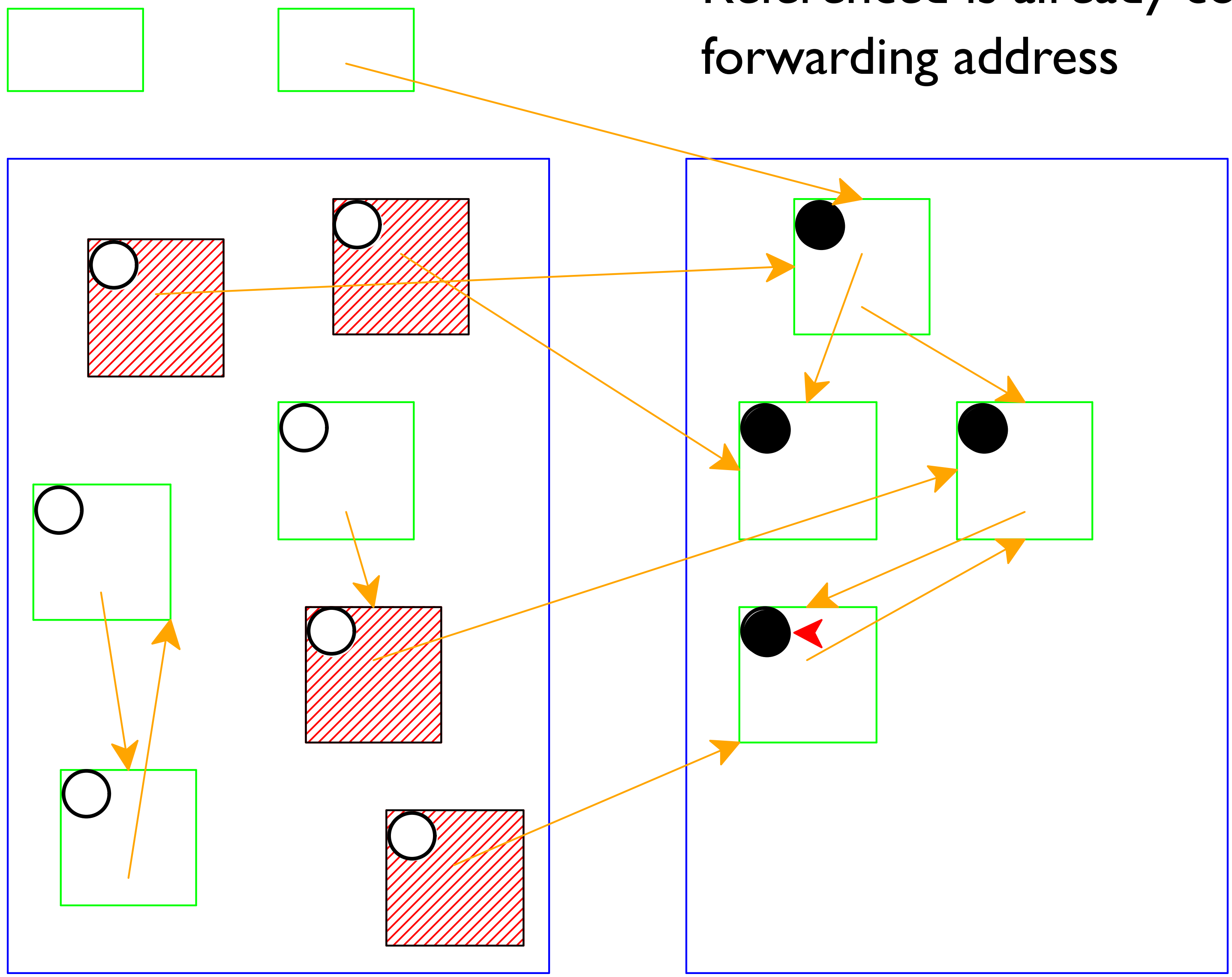
Two-Space Collection

Increment the gray-choosing arrow



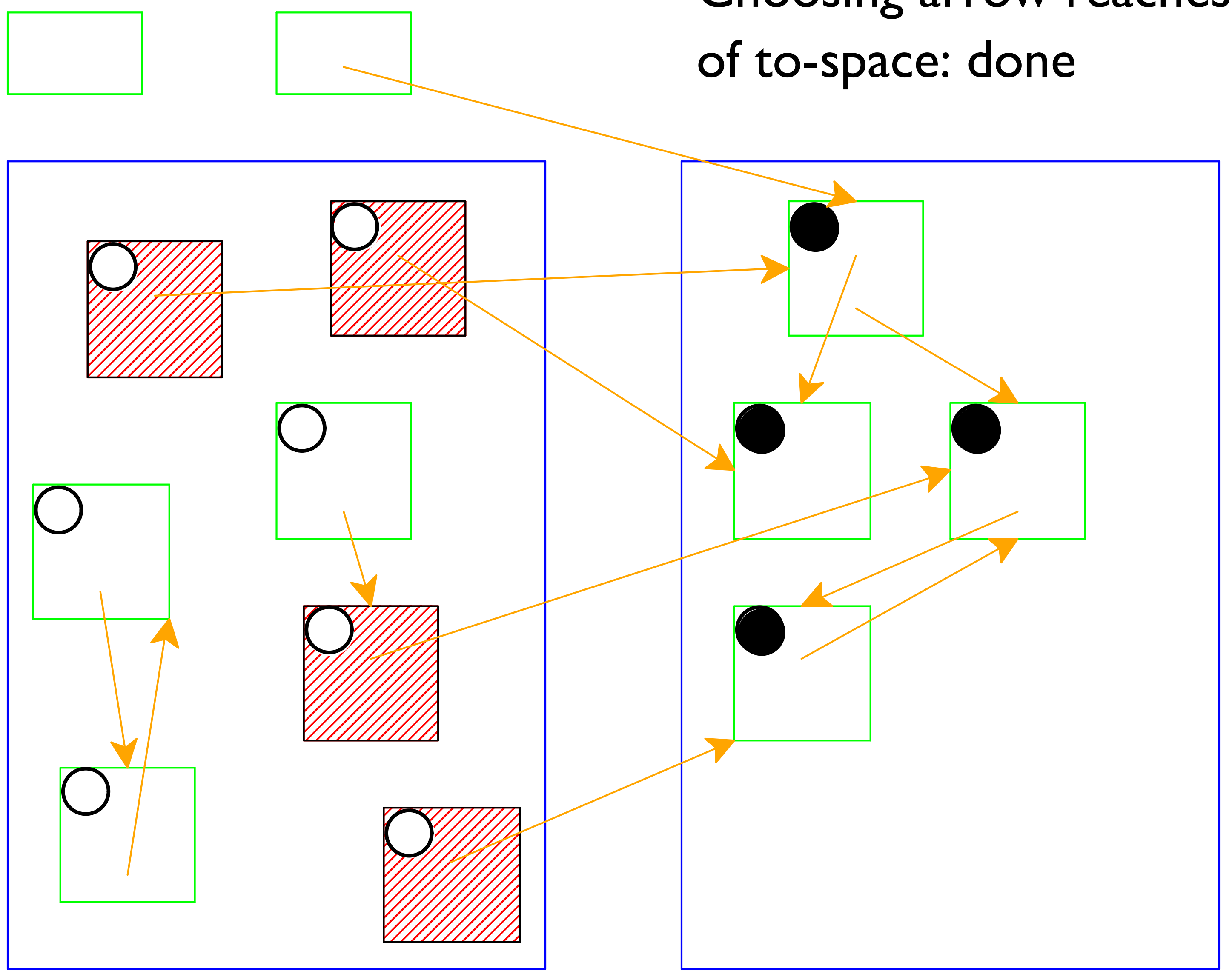
Two-Space Collection

Referenced is already copied, use forwarding address



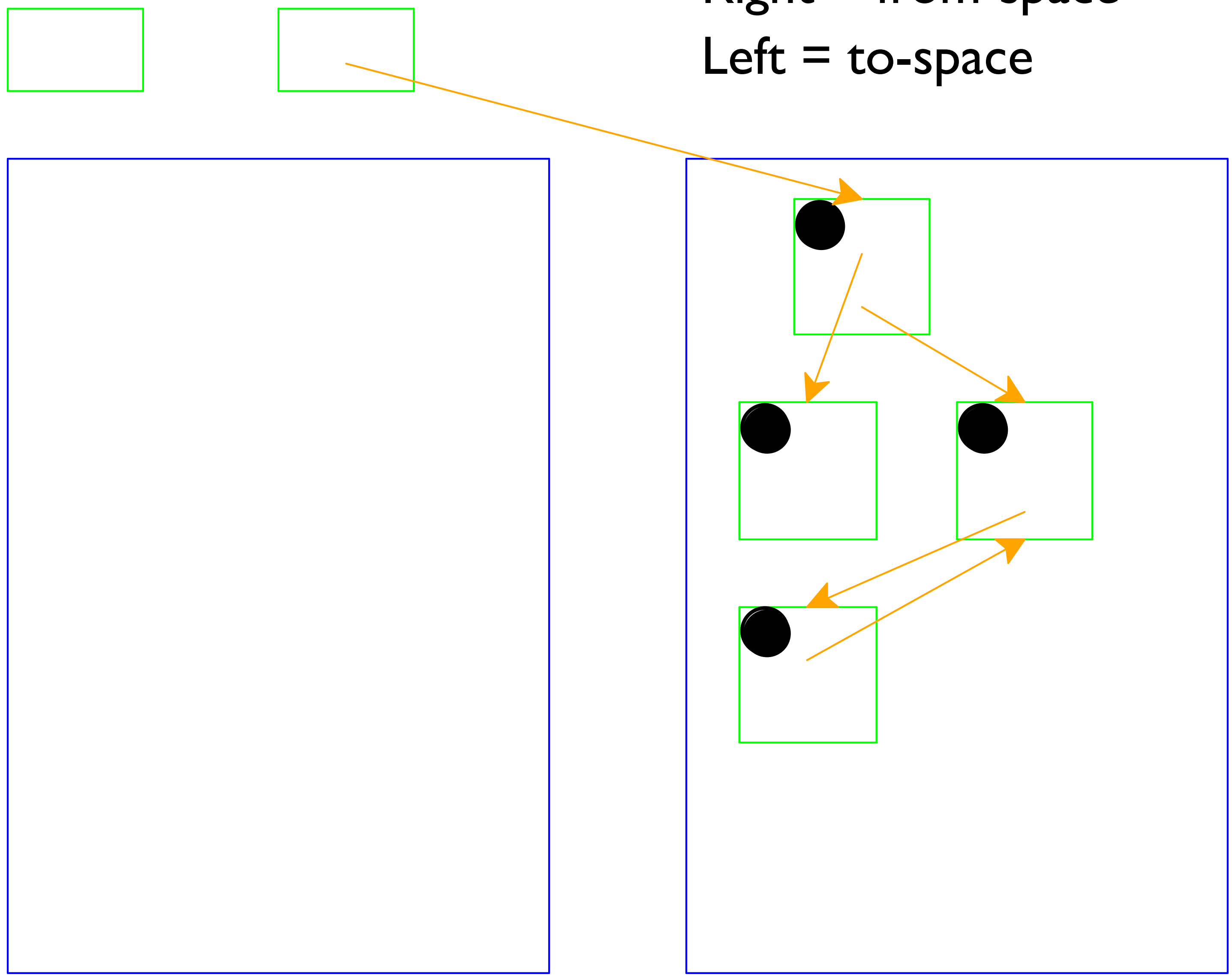
Two-Space Collection

Choosing arrow reaches the end of to-space: done



Two-Space Collection

Right = from-space
Left = to-space



Two-Space Collection

- Cool diagrams, bro
- But what does that look like for an actual heap?
- Like, say, in **`plai/gc2`**?
- So let's go through a more concrete example
- But the actual **`plai/gc2`** implementation is your job for HW8

The Setup

- Each object in memory starts with a tag
  - Just like in **`plai/gc2`**
- Tags tell us how to interpret the heap cells that follow
  - How many cells are part of the object?
  - Which cells hold pointers?
  - Which cells hold flat data?
  - Just like in **`plai/gc2`**

The Setup

- The kinds of objects we'll be dealing with are simplified variants of the ones in **`plai/gc2`**
- Flat data will be integers only, to keep things simple
- Tag **`i`**: one integer
  - Simpler variant of **`'flat`**
- Tag **`b`**: one pointer
  - Simpler variant of **`'cons`** (like a box)
- Tag **`c`**: one integer, then one pointer
  - Simpler variant of **`'clos`**
- Tag **`f`**: forwarding pointer (one pointer)

The Strategy

- Traverse the heap, starting at the roots, using breadth-first search
    - In contrast, mark-and-sweep uses depth-first
- Visiting a node = marking it gray
    - = copying from the from-space to the to-space
    - + leaving a forwarding pointer behind in the from-space

The Strategy

- Maintain a queue of the gray nodes in the to-space
  - Marking a node gray → adding it to the queue
  - Taking a node out of the queue → marking it black
- Use that queue to keep track of the BFS
- **Invariant:**
  - objects in the queue have pointers to the from-space;
  - objects outside the queue (black) have pointers to the to-space
- Represent the queue as two pointers into the to-space
  - Increment the end pointer when enqueuing
  - Increment the front pointer when dequeuing
  - When the two pointers come together, queue is empty
    - I.e., we're done

Two-Space Collection Example

- 26-cell memory (13 cells per space), 2 roots
  - Tag i: one integer
  - Tag b: one pointer
  - Tag c: one integer, then one pointer

Root 1: `7` Root 2: `0`

From: `i 75 b 0 c 2 10 c 2 2 c 1 4`

Two-Space Collection Example

- 26-cell memory (13 cells per space), 2 roots
  - Tag i: one integer
  - Tag b: one pointer
  - Tag c: one integer, then one pointer

Root 1: 7 Root 2: 0

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

Two-Space Collection Example

- 26-cell memory (13 cells per space), 2 roots
  - Tag i: one integer
  - Tag b: one pointer
  - Tag c: one integer, then one pointer

Root 1: 7　　　　Root 2: 0

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

Two-Space Collection Example

- 26-cell memory (13 cells per space), 2 roots
  - Tag i: one integer
  - Tag b: one pointer
  - Tag c: one integer, then one pointer

Root 1: 7 Root 2: 0

From:	i	75	b	0	c	2	10	c	2	2	c	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:	^^												
Addr:	13	14	15	16	17	18	19	20	21	22	23	24	25

Two-Space Collection Example

- 26-cell memory (13 cells per space), 2 roots
  - Tag i: one integer
  - Tag b: one pointer
  - Tag c: one integer, then one pointer
  - Tag f: forwarding pointer (to to-space)

Root 1: 13 Root 2: 0

From:	i	75	b	0	c	2	10	f	13	2	c	1	4
Addr:	00	01	02	03	04	05	06	07	08	09	10	11	12
	^		^		^			^			^		
To:	c	2	2	0	0	0	0	0	0	0	0	0	0
Q:	^			^									
Addr:	13	14	15	16	17	18	19	20	21	22	23	24	25

Two-Space Collection Example

- 26-cell memory (13 cells per space), 2 roots
  - Tag i: one integer
  - Tag b: one pointer
  - Tag c: one integer, then one pointer
  - Tag f: forwarding pointer (to to-space)

Root 1: 13 Root 2: 16

From:	f	16	b	0	c	2	10	f	13	2	c	1	4
Addr:	00	01	02	03	04	05	06	07	08	09	10	11	12
	^		^		^			^			^		
To:	c	2	2	i	75	0	0	0	0	0	0	0	0
Q:	^					^							
Addr:	13	14	15	16	17	18	19	20	21	22	23	24	25

Two-Space Collection Example

- 26-cell memory (13 cells per space), 2 roots
  - Tag i: one integer
  - Tag b: one pointer
  - Tag c: one integer, then one pointer
  - Tag f: forwarding pointer (to to-space)

Root 1: 13 Root 2: 16

From:	f	16	f	18	c	2	10	f	13	2	c	1	4
Addr:	00	01	02	03	04	05	06	07	08	09	10	11	12
	^		^		^			^			^		
To:	c	2	18	i	75	b	0	0	0	0	0	0	0
Q:				^				^					
Addr:	13	14	15	16	17	18	19	20	21	22	23	24	25

Two-Space Collection Example

- 26-cell memory (13 cells per space), 2 roots
  - Tag i: one integer
  - Tag b: one pointer
  - Tag c: one integer, then one pointer
  - Tag f: forwarding pointer (to to-space)

Root 1: 13 Root 2: 16

From:	f	16	f	18	c	2	10	f	13	2	c	1	4
Addr:	00	01	02	03	04	05	06	07	08	09	10	11	12
	^		^		^			^			^		
To:	c	2	18	i	75	b	0	0	0	0	0	0	0
Q:						^		^					
Addr:	13	14	15	16	17	18	19	20	21	22	23	24	25

Two-Space Collection Example

- 26-cell memory (13 cells per space), 2 roots
  - Tag i: one integer
  - Tag b: one pointer
  - Tag c: one integer, then one pointer
  - Tag f: forwarding pointer (to to-space)

Root 1: 13 Root 2: 16

From:	f	16	f	18	c	2	10	f	13	2	c	1	4
Addr:	00	01	02	03	04	05	06	07	08	09	10	11	12
	^		^		^			^			^		
To:	c	2	18	i	75	b	16	0	0	0	0	0	0
Q:								^^					
Addr:	13	14	15	16	17	18	19	20	21	22	23	24	25

Two-Space Collection Example

- 26-cell memory (13 cells per space), 2 roots
  - Tag i: one integer
  - Tag b: one pointer
  - Tag c: one integer, then one pointer
  - Tag f: forwarding pointer (to to-space)

Root 1: 13 Root 2: 16

From:	f	16	f	18	c	2	10	f	13	2	c	1	4
Addr:	00	01	02	03	04	05	06	07	08	09	10	11	12
	^		^		^			^			^		
To:	c	2	18	i	75	b	16	0	0	0	0	0	0
Q:								^	next alloc. here				
Addr:	13	14	15	16	17	18	19	20	21	22	23	24	25

Two-Space Pros and Cons

- Doesn't suffer from fragmentation
- Time cost proportional to live data (not garbage!)
- Allocation is simple, just bump a pointer
- Collection doesn't require much state (handful of pointers, no stack)

- Only half the heap is in use at any time
    - Not a big deal when combined with generational collection
- Still "stop the world"

Tips for Debugging Homework 8

You may need to do a lot of debugging, and it may be painful.

- Write your heap checker first.
- Make the heap smaller to trigger GC more often.
- To stress-test your GC when debugging, GC on every allocation (not just when you run out of space).
- Pause to look at the heap when necessary (i.e., call `read`).
- Make sure you're not forgetting any roots.

Further reading

- GC first appeared circa 1958 (original LISP)
- Went mainstream with Java in the 90s
- Tremedous amount of work: new techniques, improvements, etc.
- Still an active research area to this day

Good reference: Uniprocessor Garbage Collection Techniques, by Wilson

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