Smalltalk Implementation:
Memory Management and Garbage Collection

Prof. Harry Porter
Portland State University

The Object Manager

A separate section of the VM.
- Encapsulates all memory management.
- Includes the garbage collector.

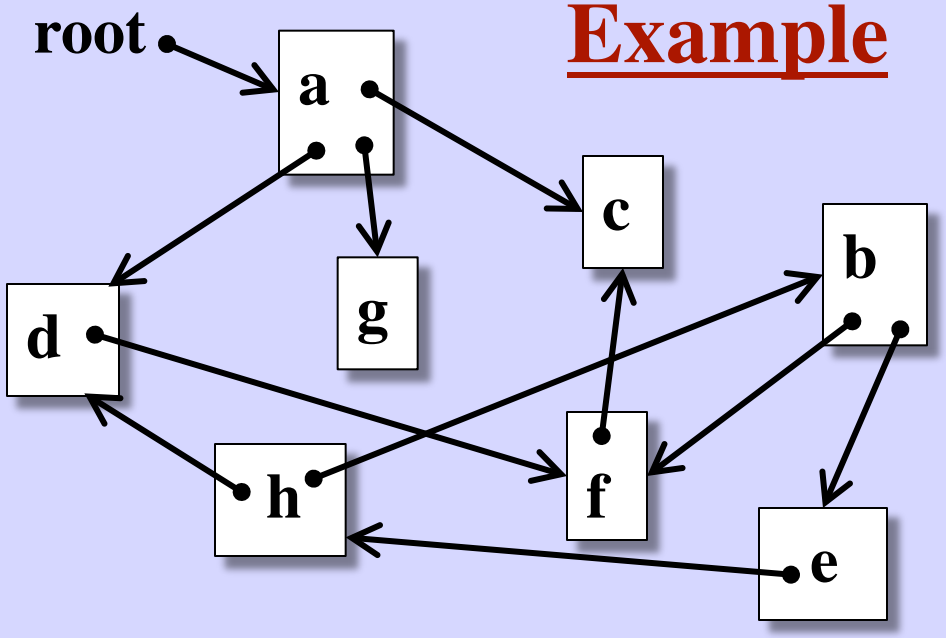
Interface from rest of VM:
- Called to allocate new space, new objects
- May impose constraints on
  - pointer dereferencing
    - (i.e., chasing pointers, fetching OOPs from object memory)
  - pointer stores
    - (i.e., copying an OOP into a variable)

Garbage Collector…
- Called implicitly when allocating new objects
  - No more free space? Run the garbage collector.
    - Try again
    - Still not enough space? *Crash!*
  - May by called periodically to "keep on top of the problem"

Object Manager Interface

- Create a new object
- Retrieve an object's field
- Update an object's field
- Get an object's size
- Get an object's class pointer
- Support "become:" operation
- Enumerate objects… "allInstancesDo:"

Example



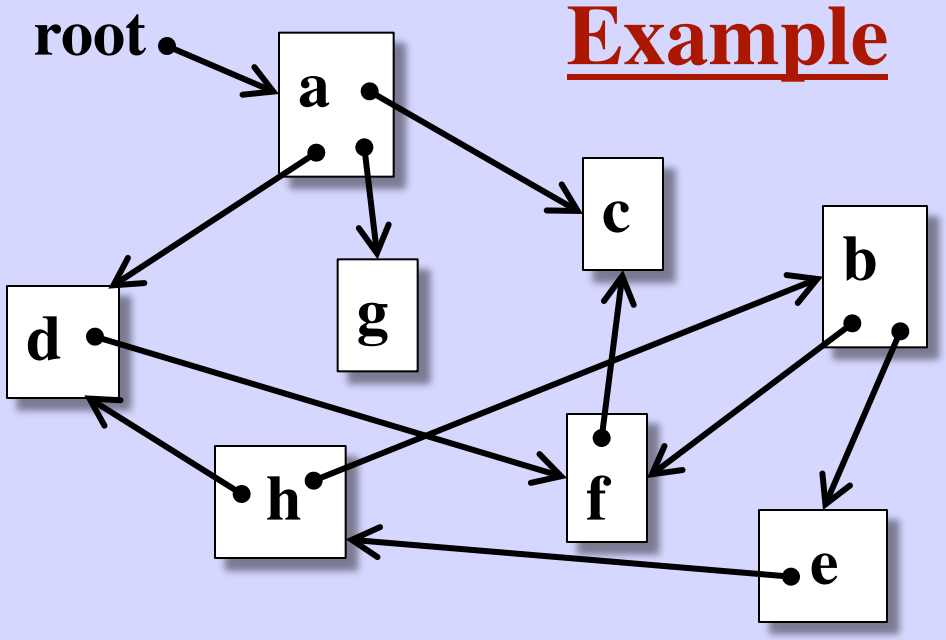
The "root" object

Defines what is reachable

May be several root pointers

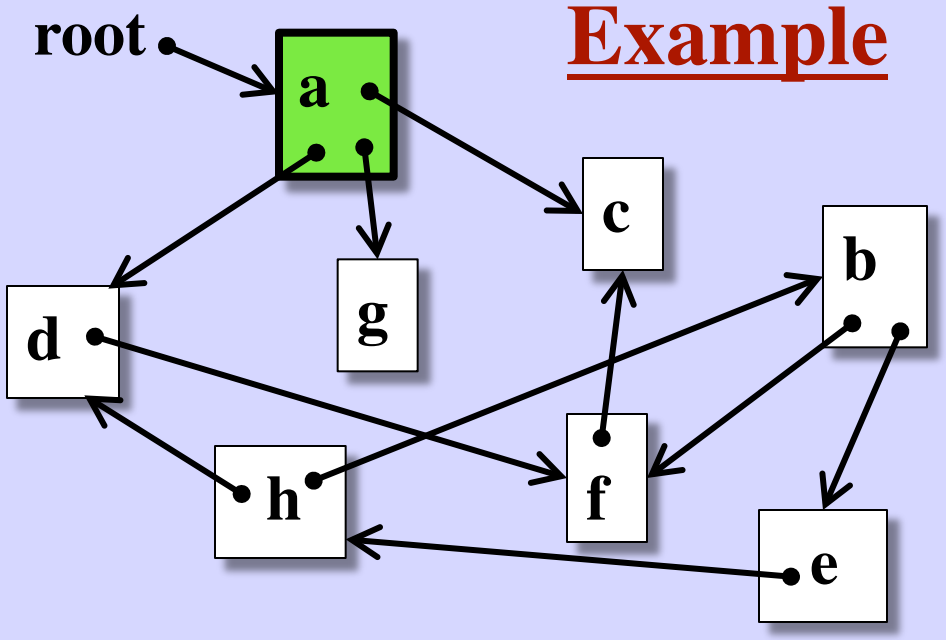
- From the calling stack
- Registers, etc.

Example



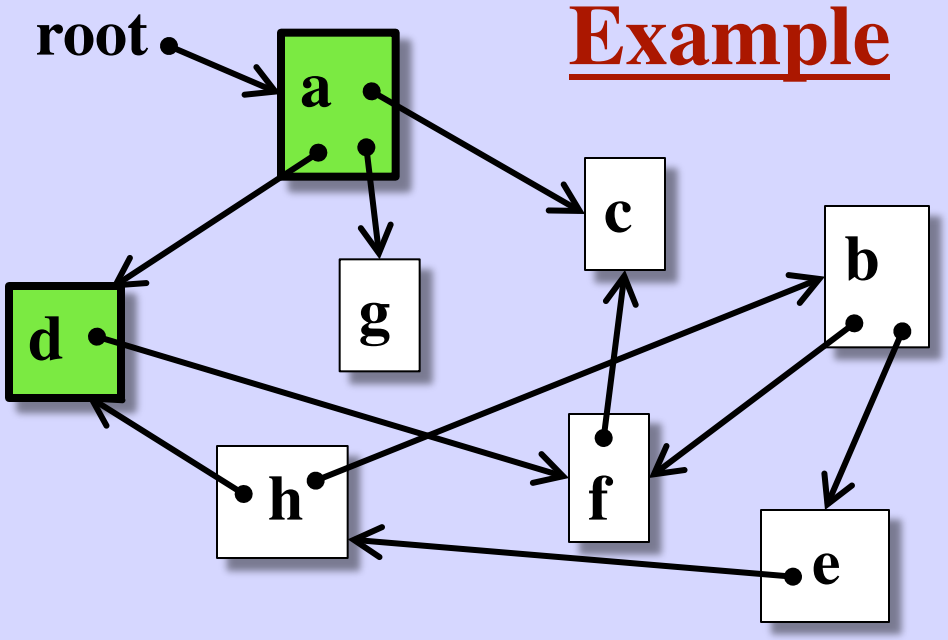
Mark every reachable object starting with the root object(s).

Example



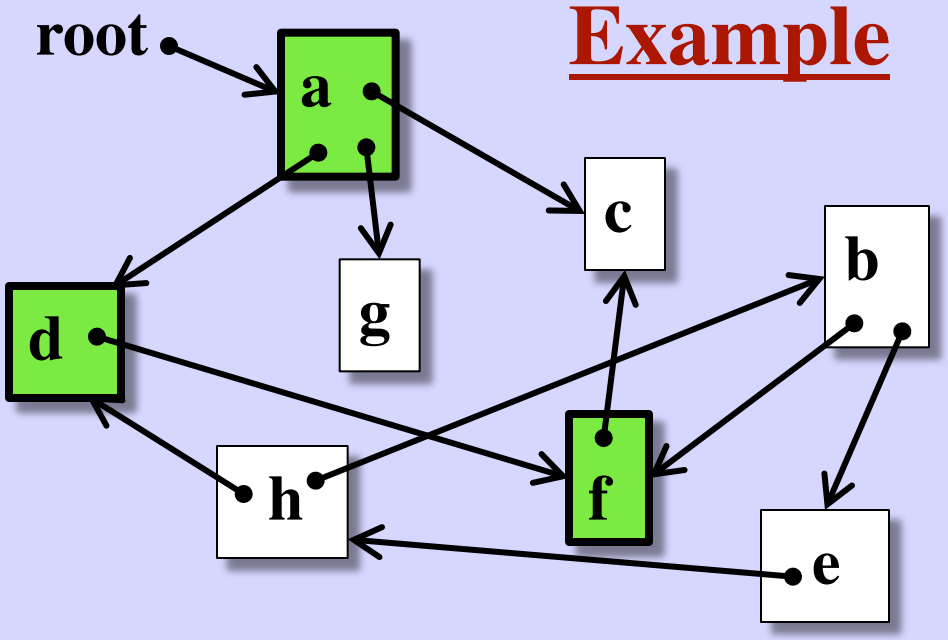
Mark every reachable object starting with the root object(s).

Example



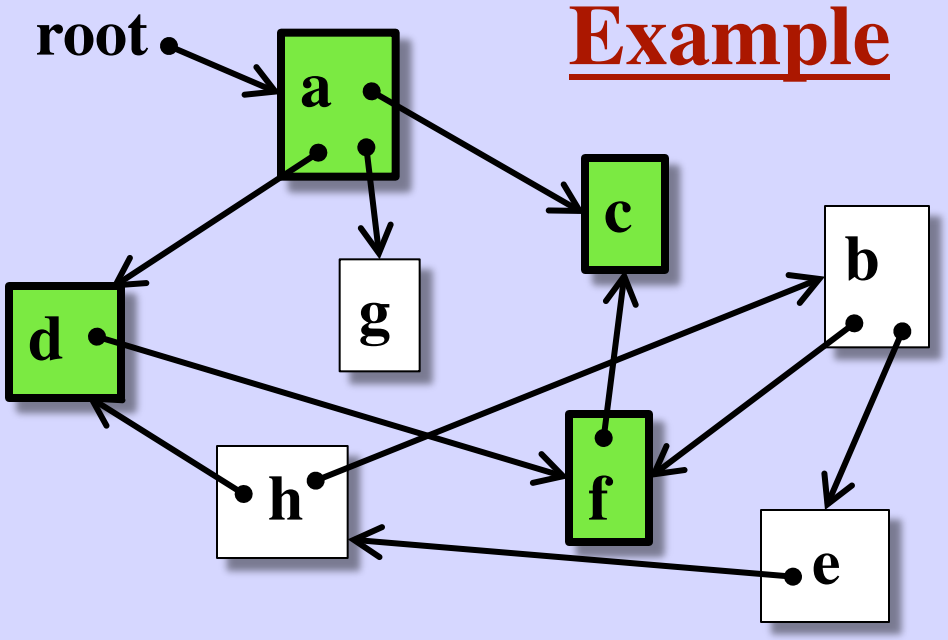
Mark every reachable object starting with the root object(s).

Example



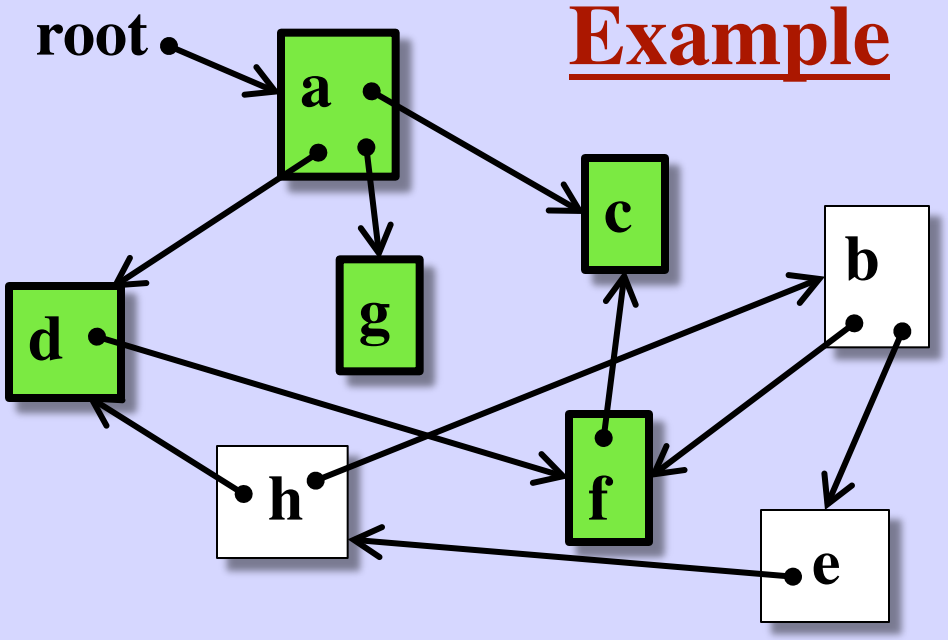
Mark every reachable object starting with the root object(s).

Example



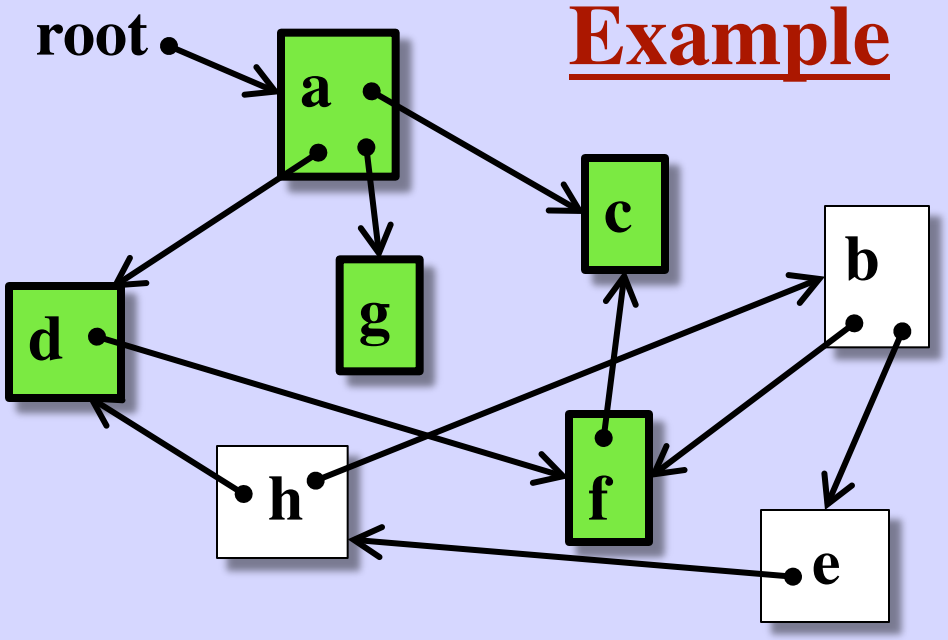
Mark every reachable object starting with the root object(s).

Example



Mark every reachable object starting with the root object(s).

Example

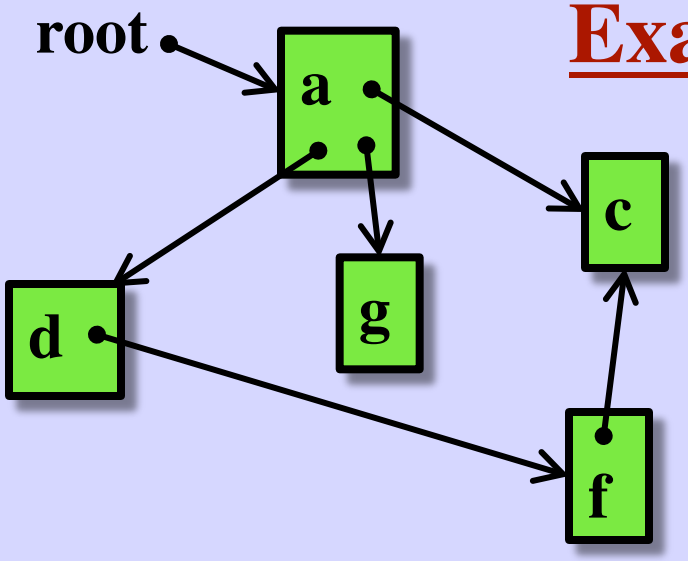


Everything else is garbage

Delete the garbage

"reclaim the memory space"

Example

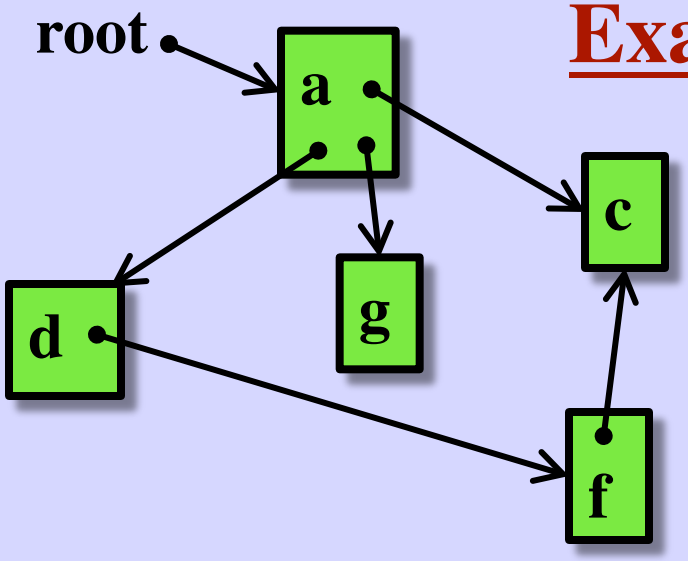


Everything else is garbage

Delete the garbage

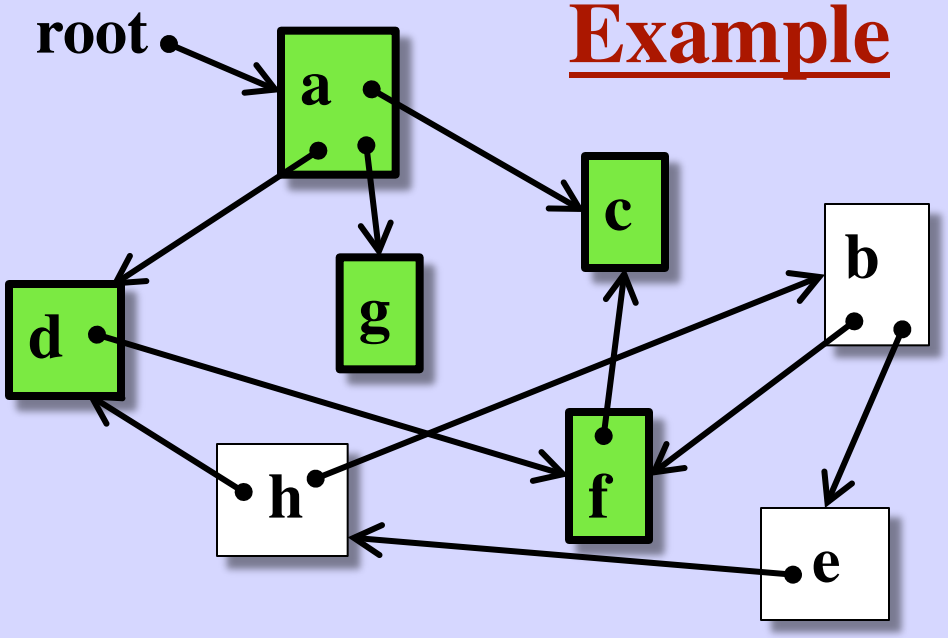
"reclaim the memory space"

Example



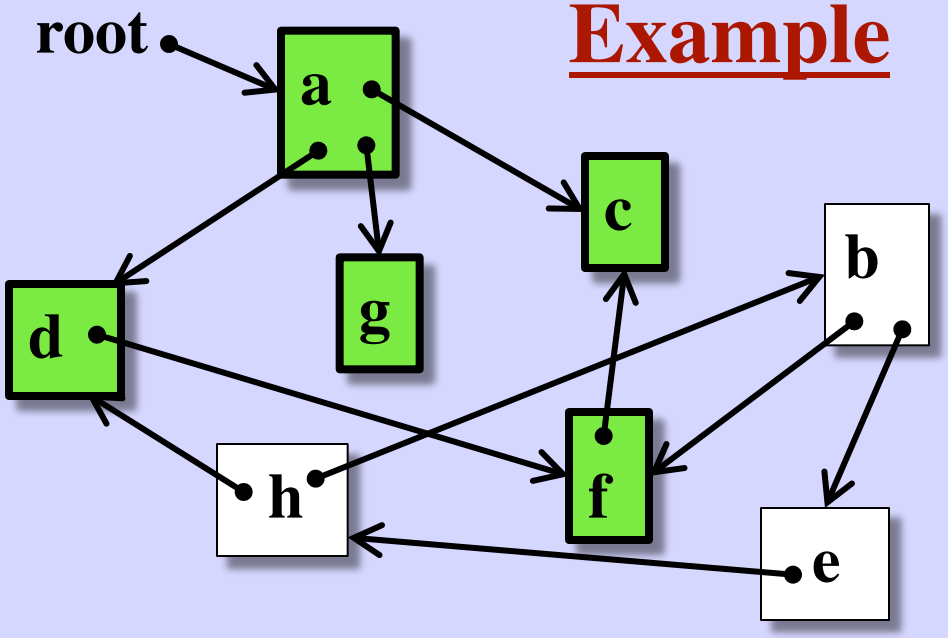
Step 2: Compact the memory.

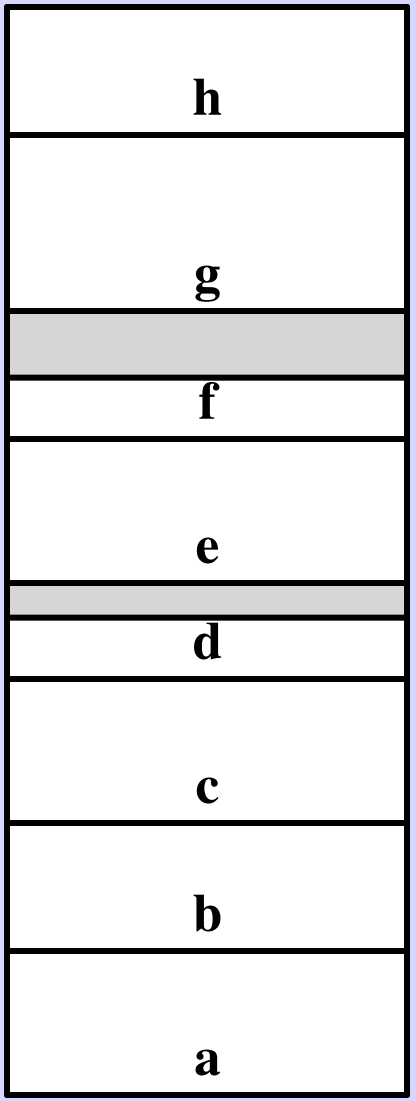
Example



Step 2: Compact the memory.

Example

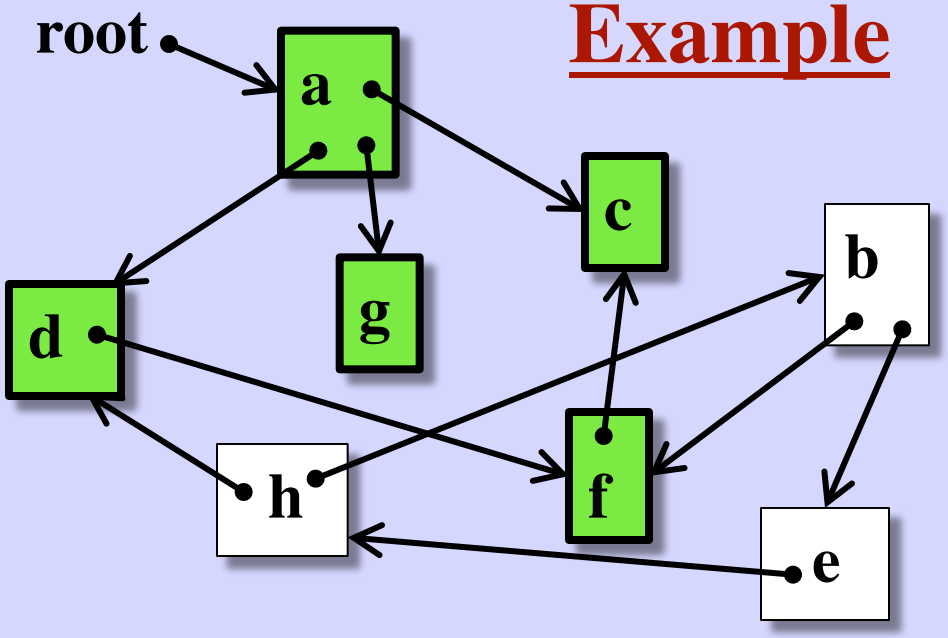


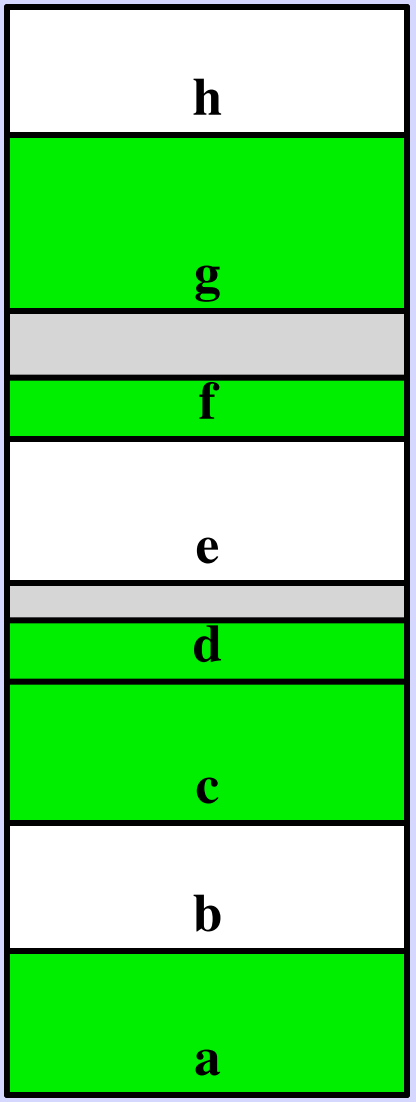


Memory

Step 2: Compact the memory.

Example

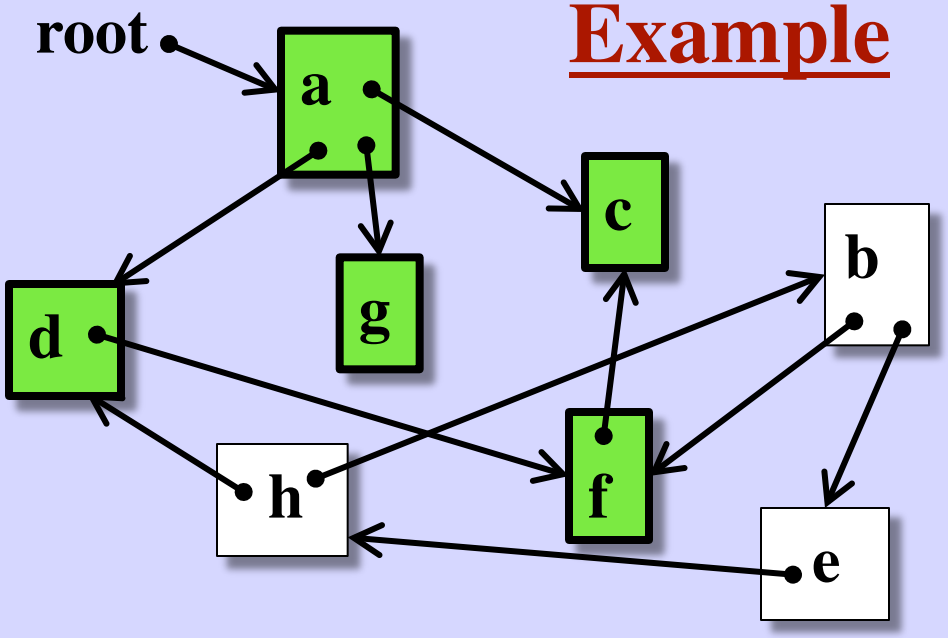




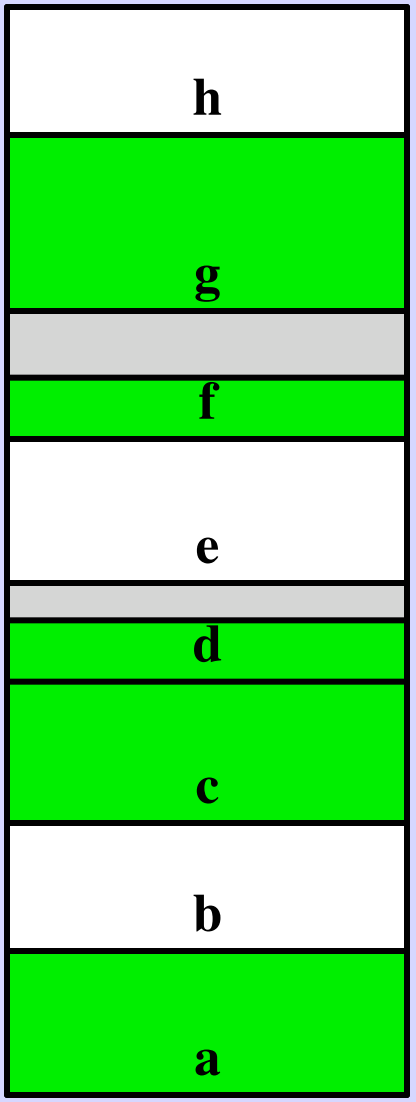
Memory

Step 2: Compact the memory.

Example

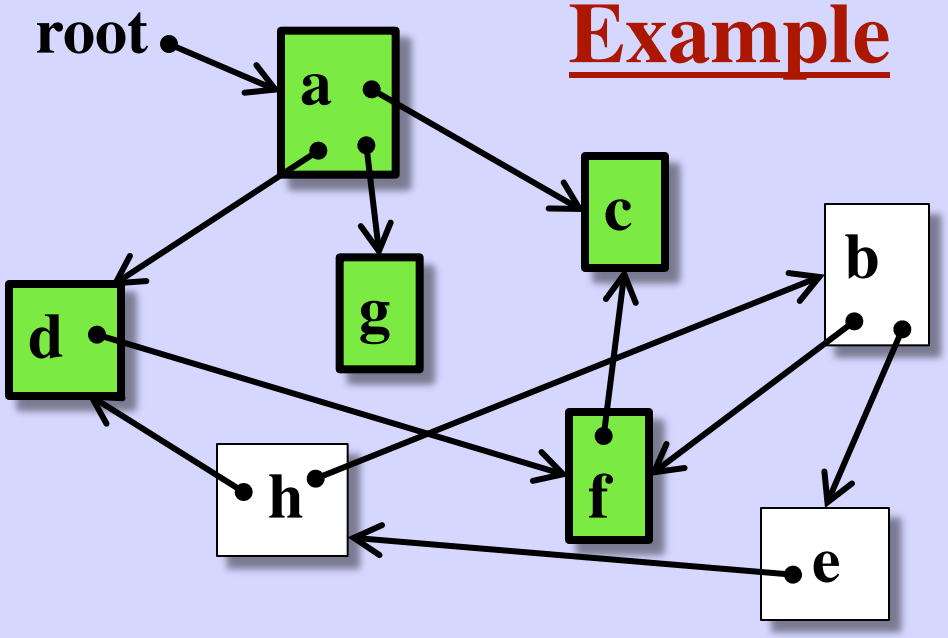


Step 2: Compact the memory.

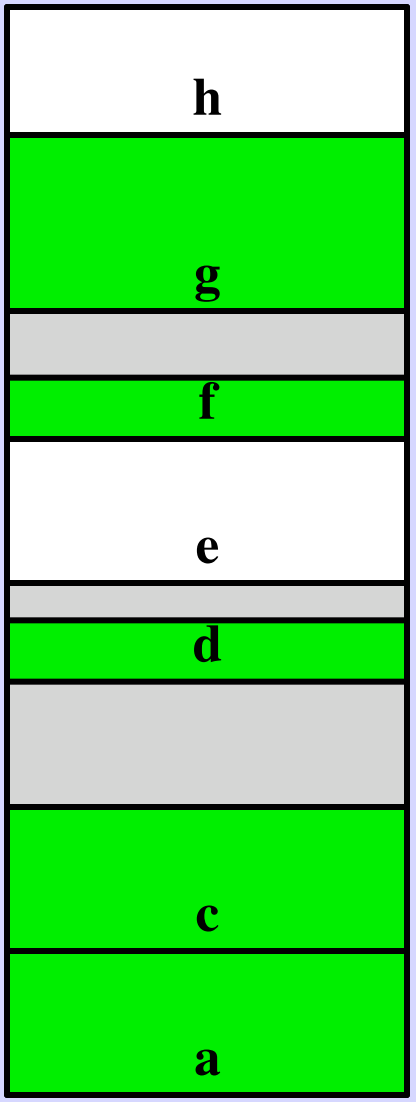


Memory

Example

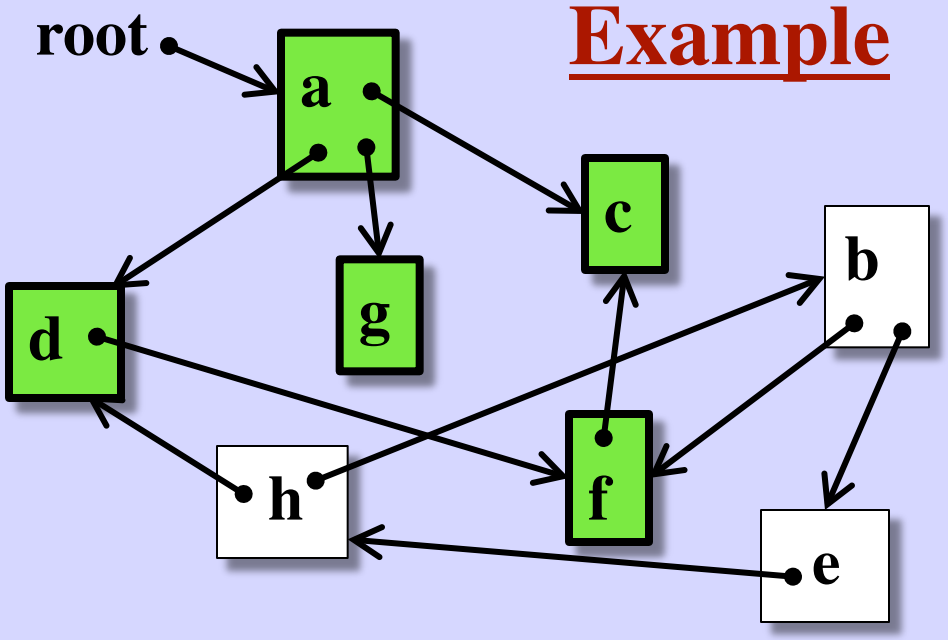


Step 2: Compact the memory.

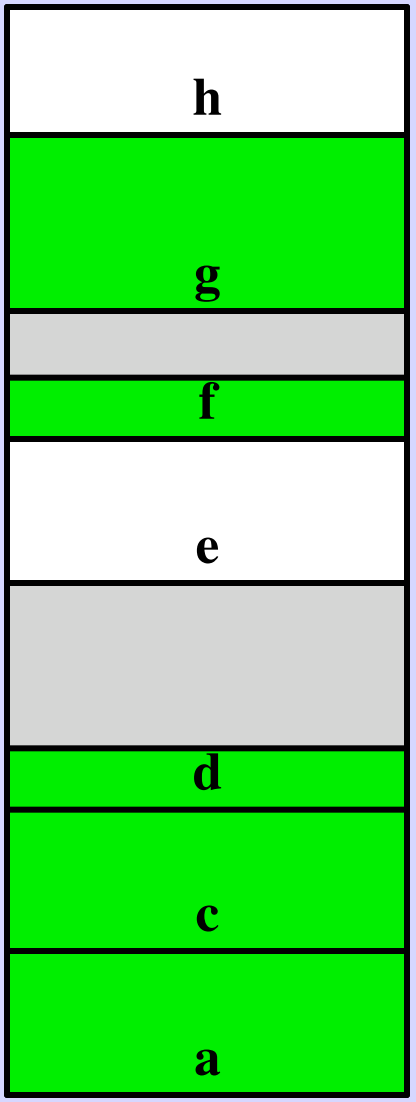


Memory

Example

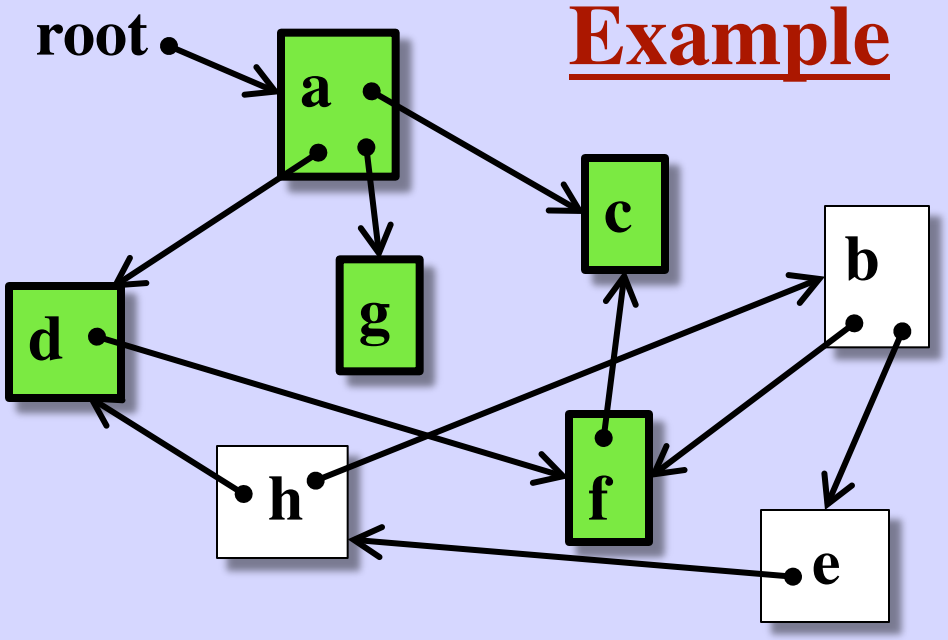


Step 2: Compact the memory.

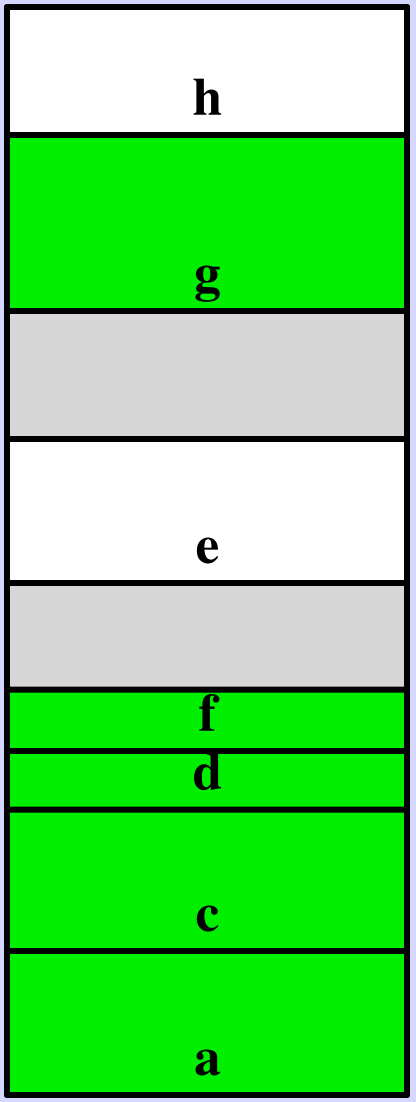


Memory

Example

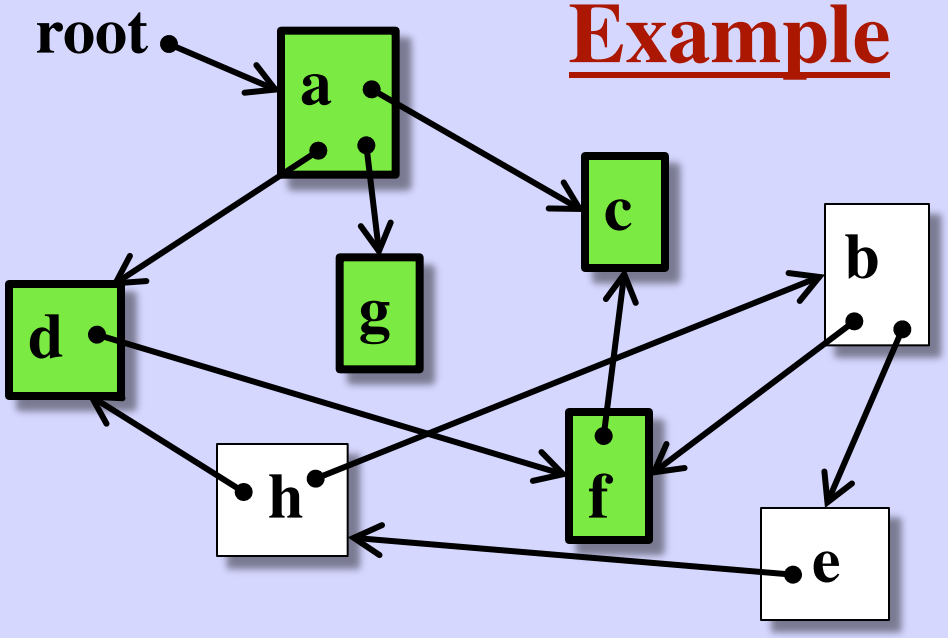


Step 2: Compact the memory.

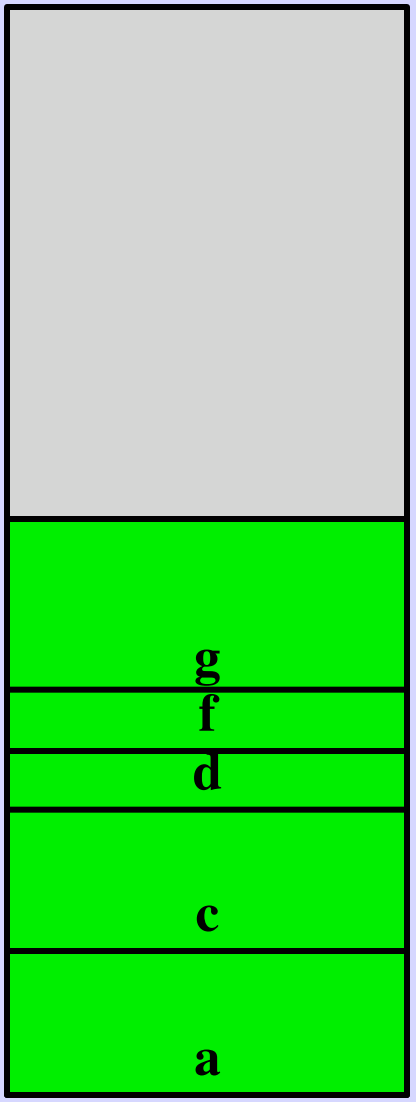


Memory

Example



Step 2: Compact the memory.



Memory

Just Use Virtual Memory???

Idea: Avoid G.C. and just use virtual memory

Page objects out to disk.

Worry about collecting later (perhaps at night?)

Smalltalk Statistics:

Average size of new objects: 20 bytes

Minimum object size: 4 bytes

Object allocation rate:

1 object per 80 bytecodes executed

(= 1/4 bytes allocated per bytecodes executed)

The Numbers:

Execution rate: 4,000,000 bytecodes/sec

Disk Size: 10 Gbyte

Result: Disk fills up in 80 minutes

(And how long to collect 10 Gbyte *on disk*?)

Conclusion: We cannot ignore G.C.

Major Garbage Collection Algorithms

- Mark-Sweep
  Simple
- Baker's Semi-Space Algorithm
  Good intro. to Generation Scavenging
- Generation Scavenging (David Ungar)
  Fast
  Widespread use
- Reference Counting
  No longer used in Smalltalk

Ongoing research:
Performance tuning, variations, …

Mark-Sweep Garbage Collection

Associate a single bit with each object

The "mark" bit

Part of the object's header

Initially, all "mark" bits are clear

- **Phase 1:**

  Set the "mark" bit for every reachable object

- **Phase 2:**

  Compact the object space

  (and clear the "mark" bit for next time)

  Will move objects.

  Need to adjust all pointers.

Mark-Sweep Garbage Collection

How to set the "mark" bit?

Option 1: **A recursive algorithm**

But this requires a stack (and memory is full!)

Option 2:

Option 3:

Mark-Sweep Garbage Collection

How to set the "mark" bit?

Option 1: **A recursive algorithm**

But this requires a stack (and memory is full!)

Option 2:

```
REPEAT
  LOOP through all objects
    IF the object's mark is set THEN
      LOOP through the object's fields
        Set the mark bit of all objects it points to
      ENDLOOP
    ENDIF
  ENDLOOP
UNTIL no more changes
```

Repeated loops through memory? SLOW!

Option 3:

Mark-Sweep Garbage Collection

How to set the "mark" bit?

Option 1: **A recursive algorithm**

But this requires a stack (and memory is full!)

Option 2:

```
REPEAT
  LOOP through all objects
    IF the object's mark is set THEN
      LOOP through the object's fields
        Set the mark bit of all objects it points to
      ENDLOOP
    ENDIF
  ENDLOOP
UNTIL no more changes
```

Repeated loops through memory? SLOW!

Option 3:

Keep a "to-do list".

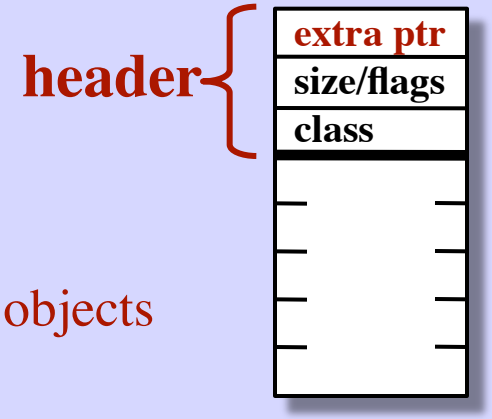
Mark-Sweep Garbage Collection

Desired Algorithm:

When we mark an object, push it on a stack.
Repeat: Pop next object off of stack
Mark all reachable objects
… until stack is empty

Unfortunately:

The stack may be arbitrarily deep.
No extra memory when the G.C. is running!

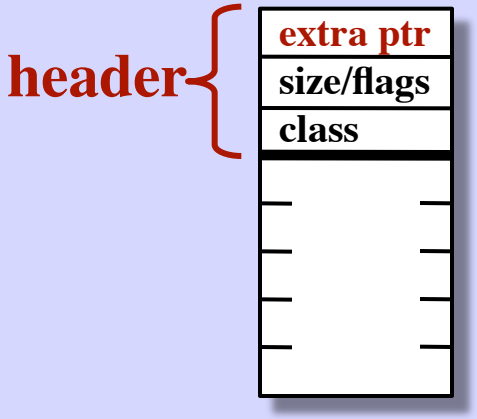


Solution:

Allocate one extra word per object.
Use this "extra" pointer to maintain a linked list of objects
(the stack)
When an object is found to be reachable...
Set its "mark" bit
Add it to the linked list

Mark-Sweep Garbage Collection

```
Mark root object
Add root object to the linked list
LOOP
   Remove an element from the list
   Look at each of its fields...
   FOR EVERY object it points to
      IF it is not already marked THEN
         Mark it
         Add it to the list
      ENDIF
   ENDFOR
UNTIL list is empty
```



Mark-Sweep Garbage Collection

Advantages:

- Will identify all true garbage
- Very little space overhead
- Simple → Easy to program

Disadvantages:

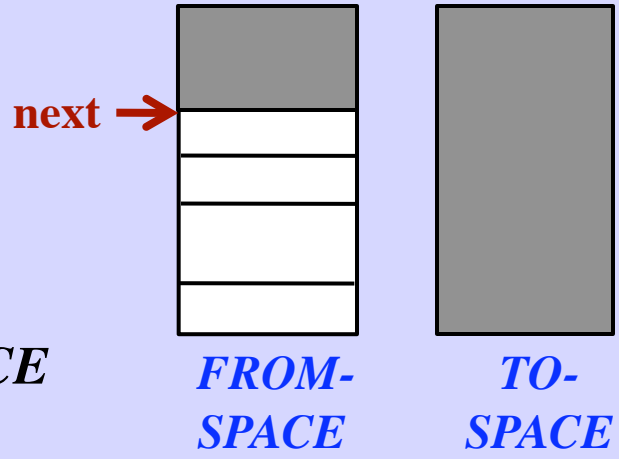
- The marking phase can be slow!
  - Must look at every field
    (in every non-garbage object)
  - Must check the "tag" bit
    OOP → follow the pointer
    SmallInteger → ignore
- Causes lengthy interruptions (periodically)
  Annoying for interactive applications

Baker's Semi-Space Algorithm

Memory is divided into 2 (equal-sized) spaces

FROM-SPACE

TO-SPACE



Normal Operation:

- All objects are in ***FROM-SPACE***
- ***TO-SPACE*** is unused
- New objects are allocated in ***FROM-SPACE*** (typically like a stack)

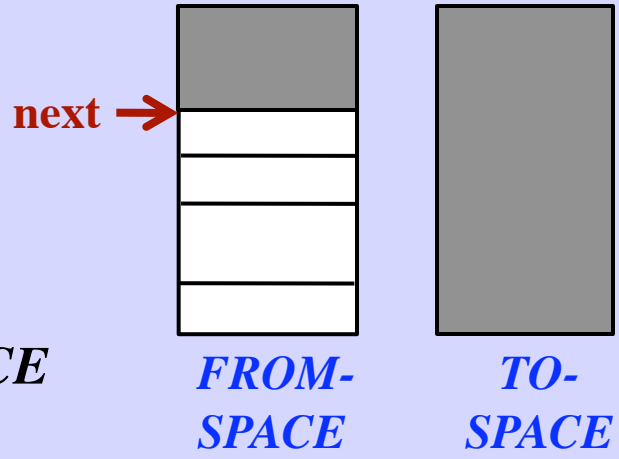
When ***FROM-SPACE*** is exhausted…

Baker's Semi-Space Algorithm

Memory is divided into 2 (equal-sized) spaces

FROM-SPACE

TO-SPACE



Normal Operation:

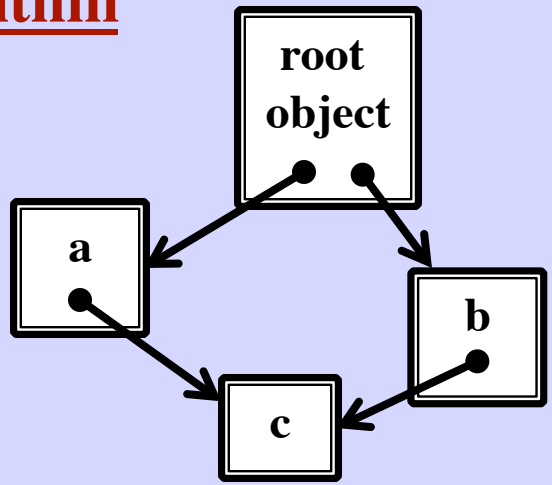
- All objects are in ***FROM-SPACE***
- ***TO-SPACE*** is unused
- New objects are allocated in ***FROM-SPACE***
  (typically like a stack)

When ***FROM-SPACE*** is exhausted…

- Copy the root object to ***TO-SPACE***
- Copy all reachable objects
  from the ***FROM-SPACE***
  to the ***TO-SPACE***
- All the garbage objects are left behind in ***FROM-SPACE***
- Abandon ***FROM-SPACE*** and continue processing in ***TO-SPACE***

Baker's Semi-Space Algorithm

During normal operation

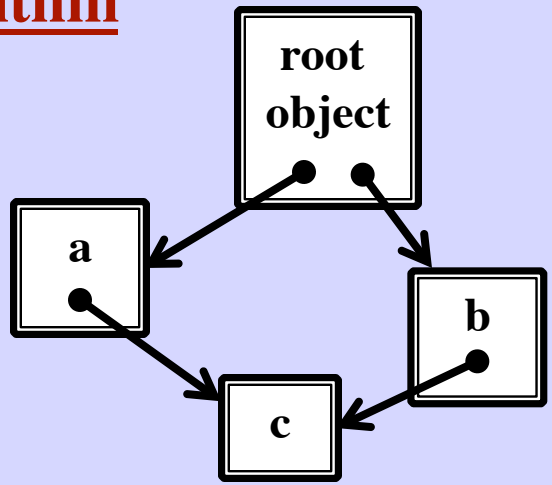


Baker's Semi-Space Algorithm

During normal operation

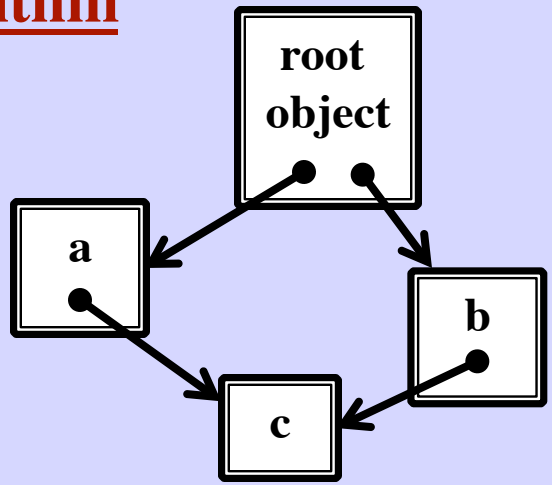
Use one pointer in FROM-SPACE

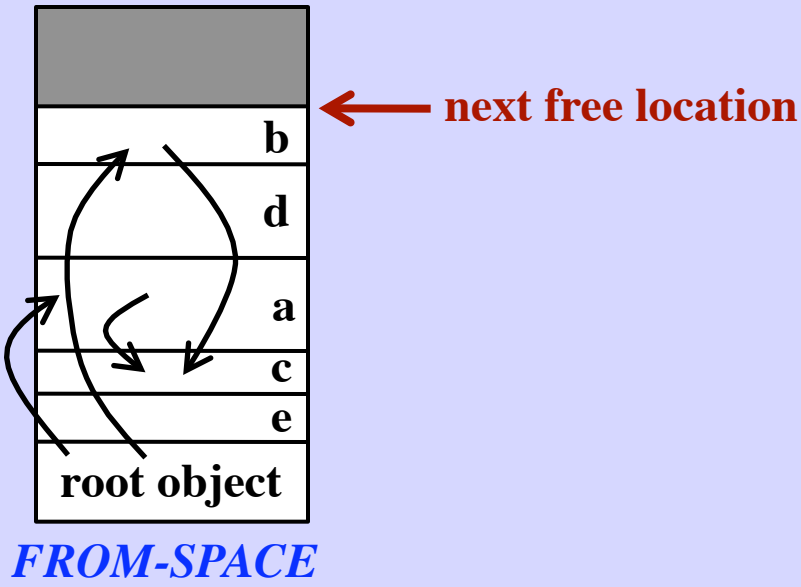
next-free-location



Baker's Semi-Space Algorithm

During normal operation
Use one pointer in FROM-SPACE
next-free-location



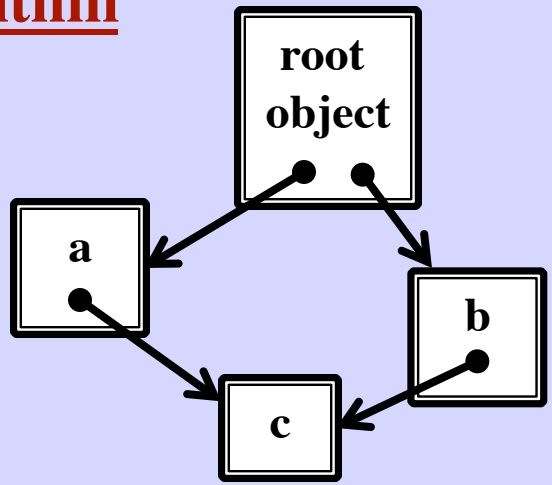


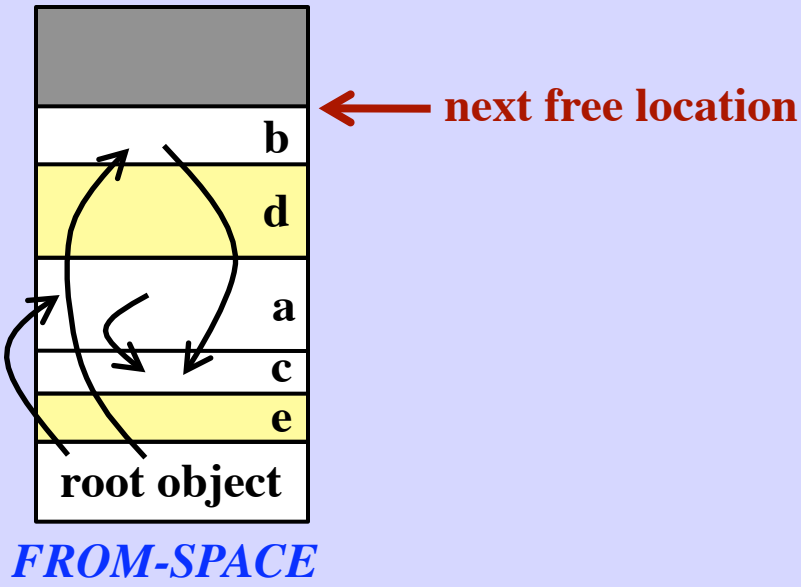
Baker's Semi-Space Algorithm

During normal operation

Use one pointer in FROM-SPACE

next-free-location



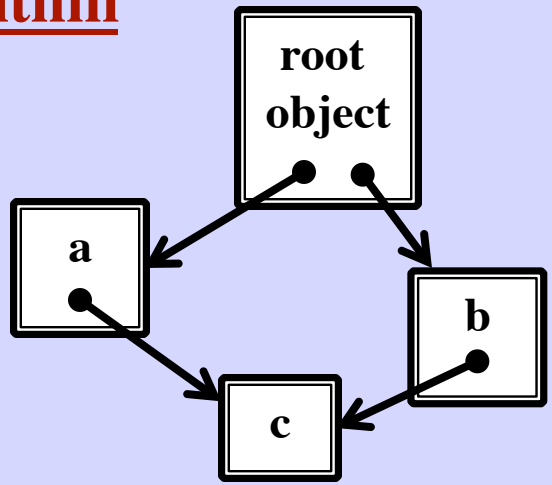


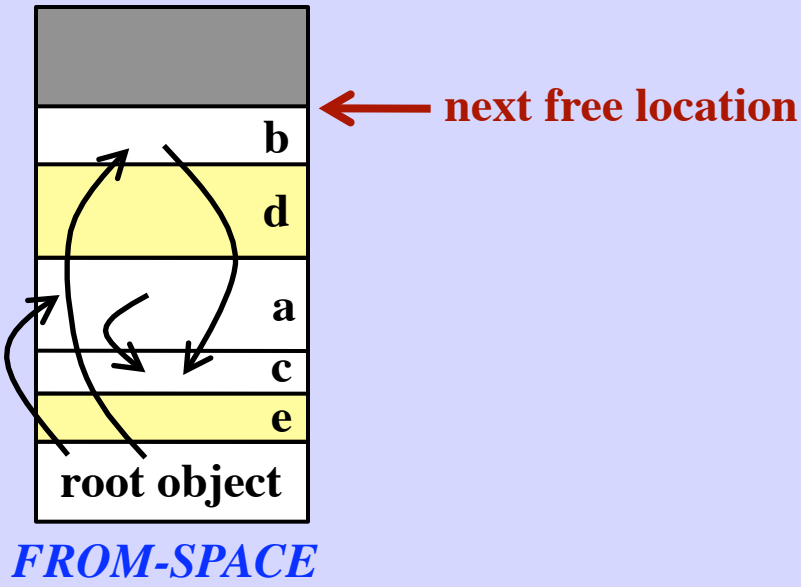
Baker's Semi-Space Algorithm

During garbage collection…

Copy all reachable objects to *TO-SPACE*

First copy the root object.



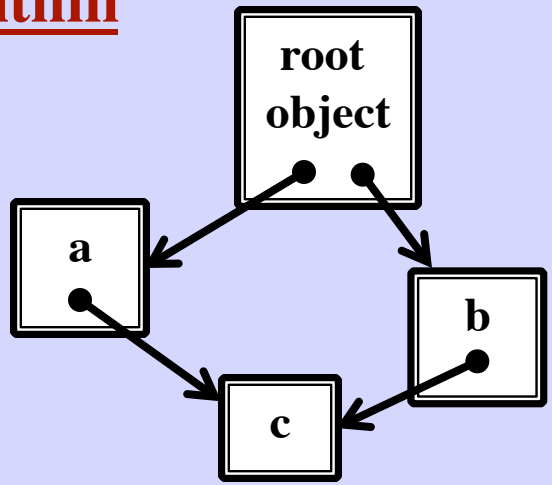


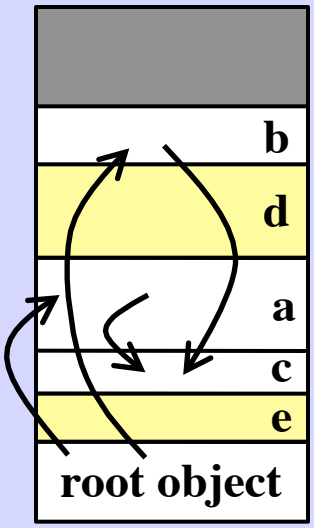
Baker's Semi-Space Algorithm

During garbage collection…

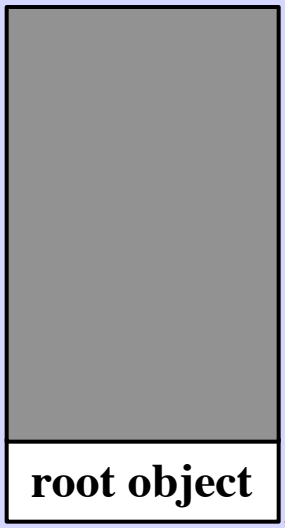
Copy all reachable objects to *TO-SPACE*

First copy the root object.





FROM-SPACE



TO-SPACE

next free location

next unscanned location

Baker's Semi-Space Algorithm

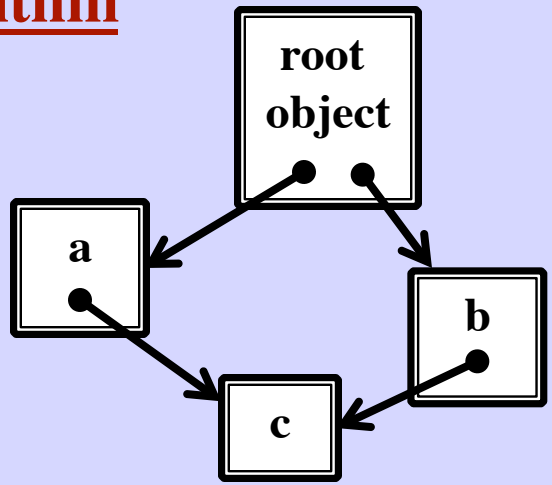
During garbage collection…

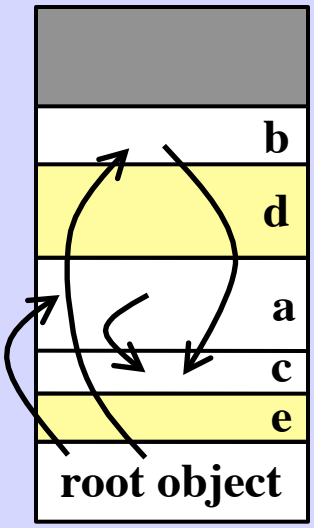
Copy all reachable objects to *TO-SPACE*

First copy the root object.

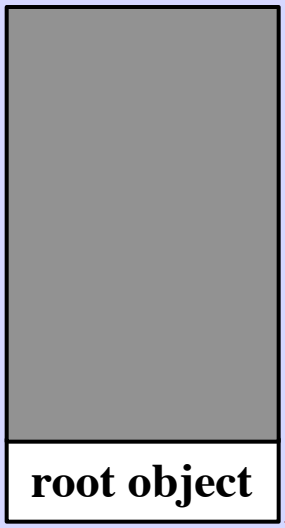
Then scan the next object

and copy the objects it points to.





FROM-SPACE



TO-SPACE

next free location

next unscanned location

Baker's Semi-Space Algorithm

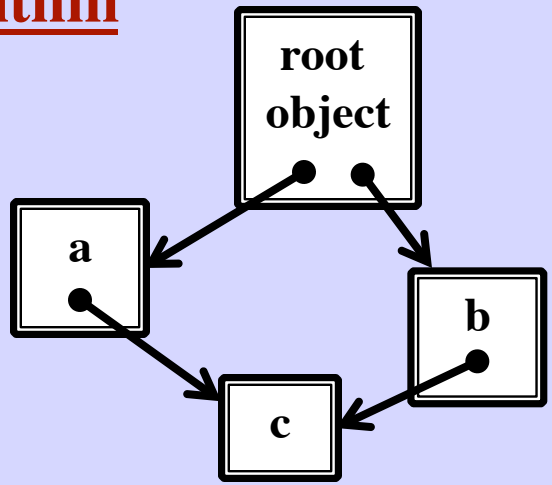
During garbage collection…

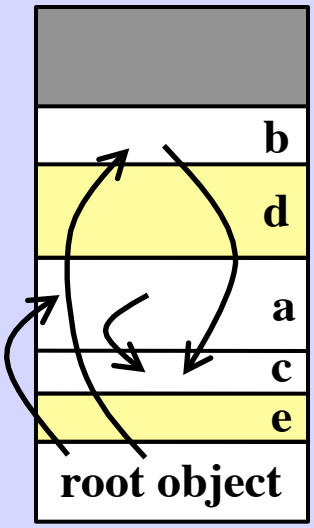
Copy all reachable objects to *TO-SPACE*

First copy the root object.

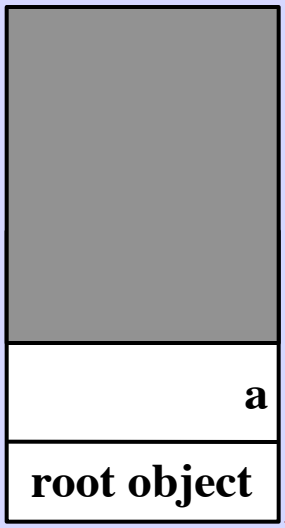
Then scan the next object

and copy the objects it points to.





FROM-SPACE



TO-SPACE

next free location

next unscanned location

Baker's Semi-Space Algorithm

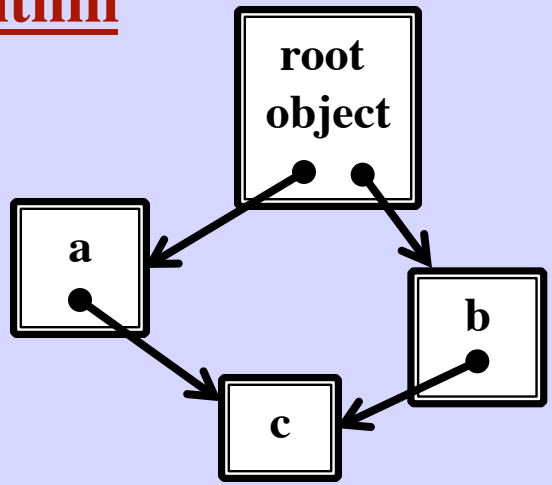
During garbage collection…

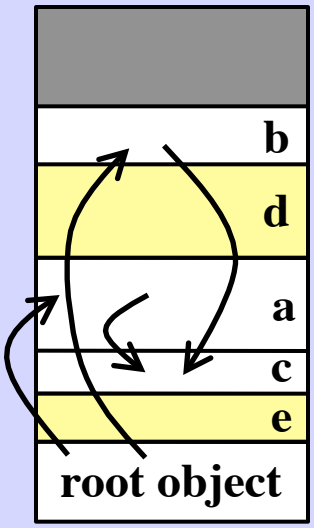
Copy all reachable objects to *TO-SPACE*

First copy the root object.

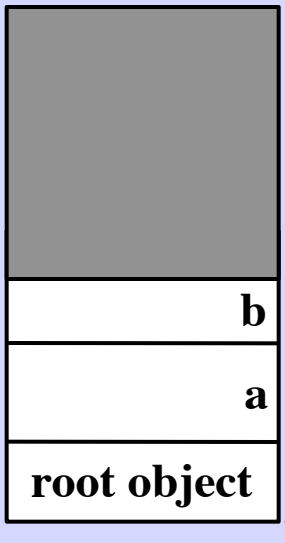
Then scan the next object

and copy the objects it points to.





FROM-SPACE



next free location

next unscanned location

TO-SPACE

Baker's Semi-Space Algorithm

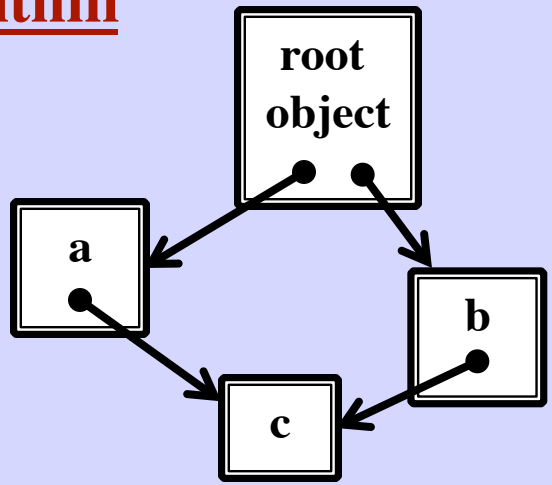
During garbage collection…

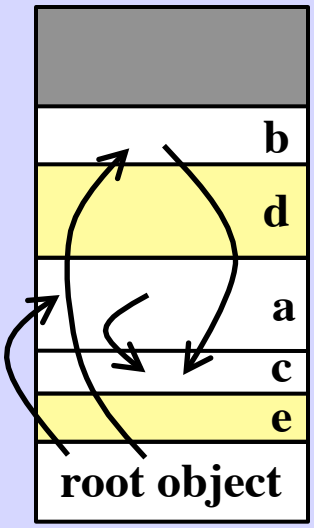
Copy all reachable objects to *TO-SPACE*

First copy the root object.

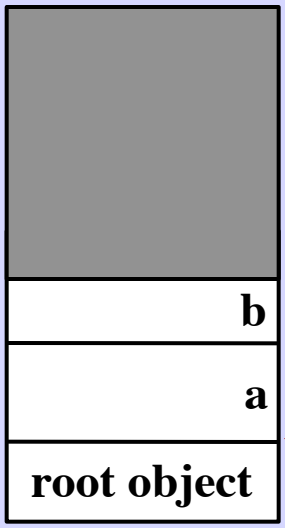
Then scan the next object

and copy the objects it points to.





FROM-SPACE



next free location

next unscanned location

TO-SPACE

Baker's Semi-Space Algorithm

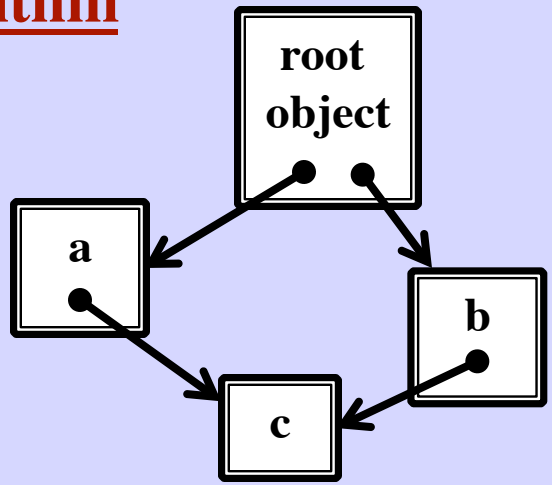
During garbage collection…

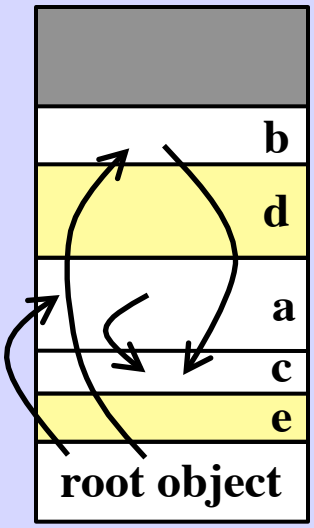
Copy all reachable objects to *TO-SPACE*

First copy the root object.

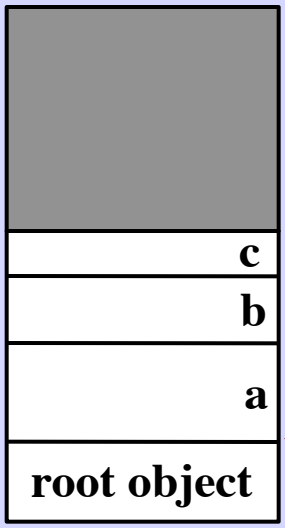
Then scan the next object

and copy the objects it points to.





FROM-SPACE



next free location

next unscanned location

TO-SPACE

Baker's Semi-Space Algorithm

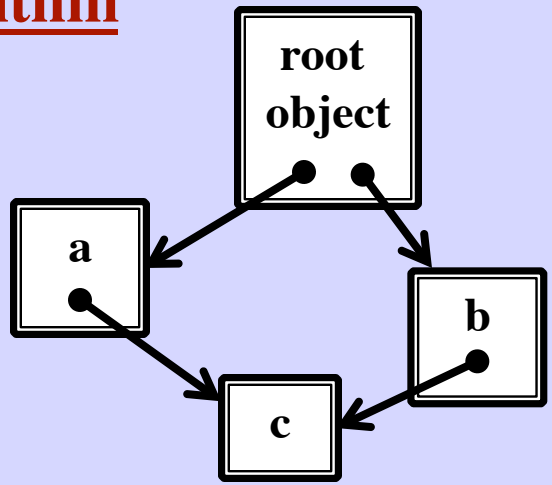
During garbage collection…

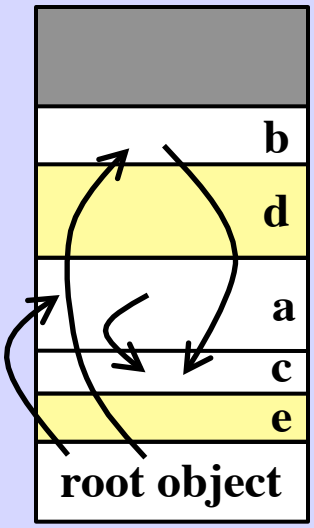
Copy all reachable objects to *TO-SPACE*

First copy the root object.

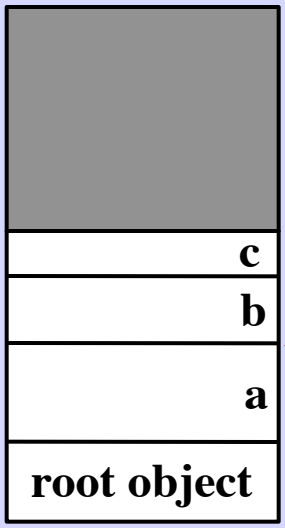
Then scan the next object

and copy the objects it points to.





FROM-SPACE



next free location

next unscanned location

TO-SPACE

Baker's Semi-Space Algorithm

During garbage collection…

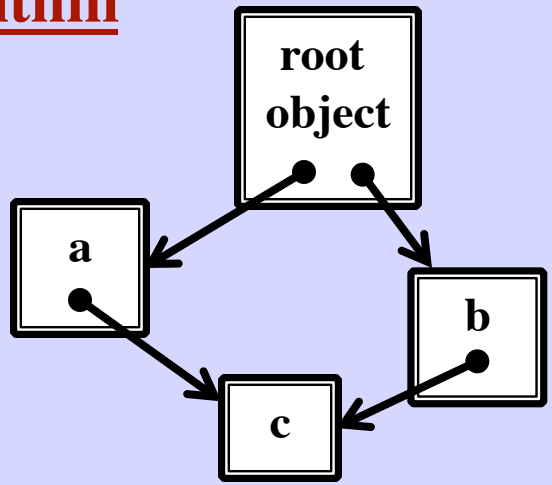
Copy all reachable objects to *TO-SPACE*

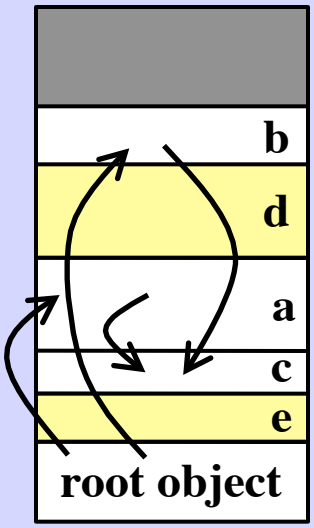
First copy the root object.

Then scan the next object

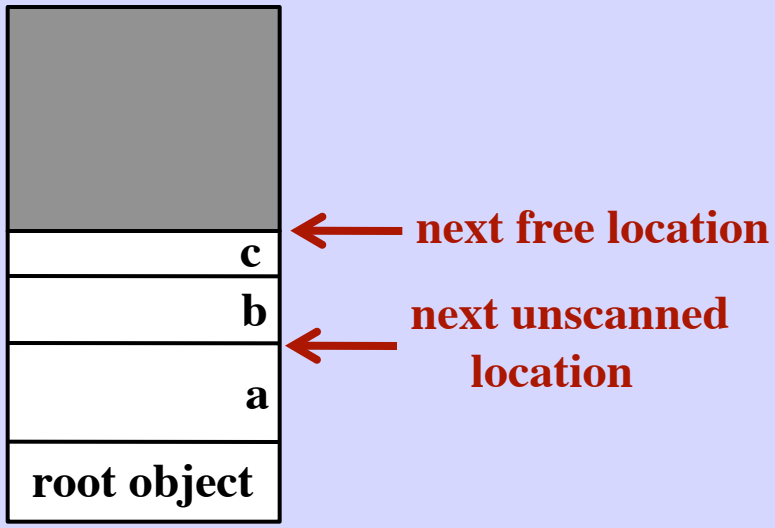
and copy the objects it points to.

Until the pointers meet.





FROM-SPACE



TO-SPACE

Baker's Semi-Space Algorithm

During garbage collection…

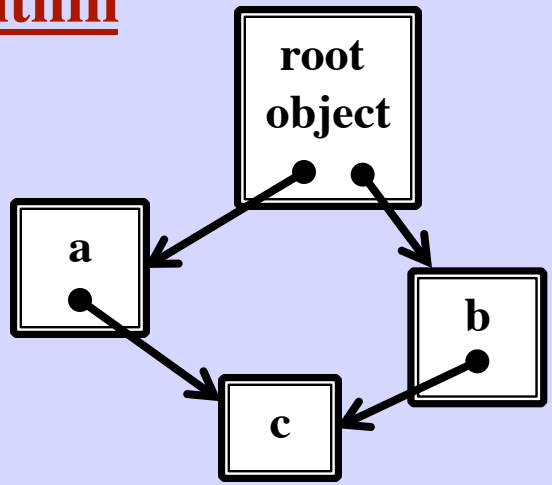
Copy all reachable objects to *TO-SPACE*

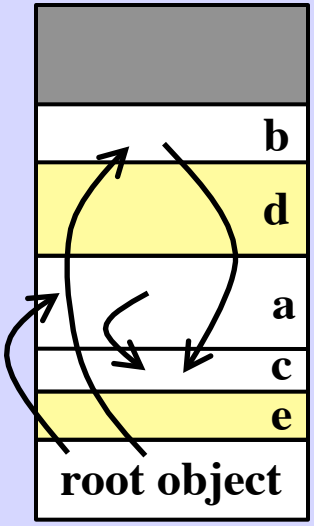
First copy the root object.

Then scan the next object

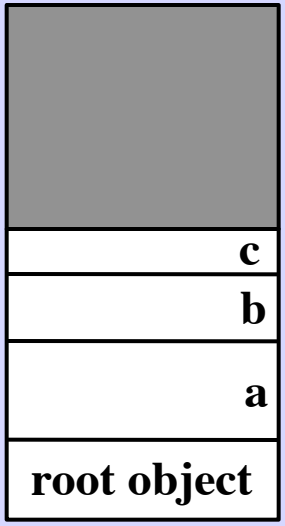
and copy the objects it points to.

Until the pointers meet.





FROM-SPACE



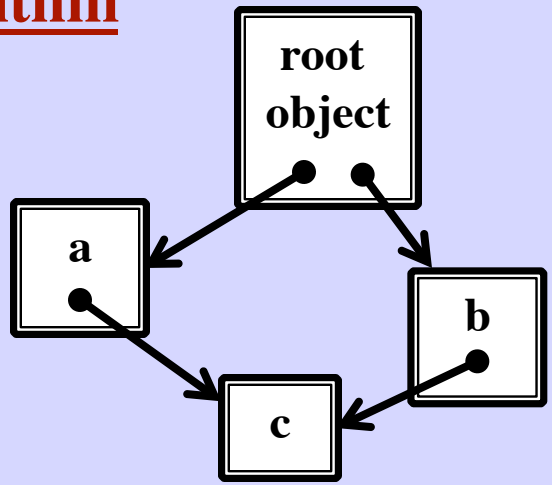
next free location

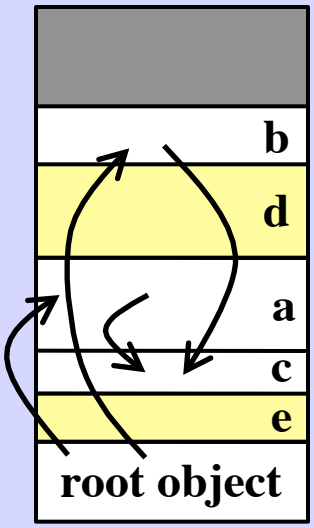
TO-SPACE

Baker's Semi-Space Algorithm

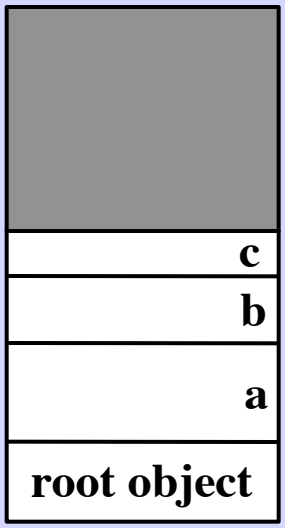
During garbage collection…

- Copy all reachable objects to *TO-SPACE*
- First copy the root object.
- Then scan the next object
  - and copy the objects it points to.
- Until the pointers meet.
- Then swap spaces.





FROM-SPACE



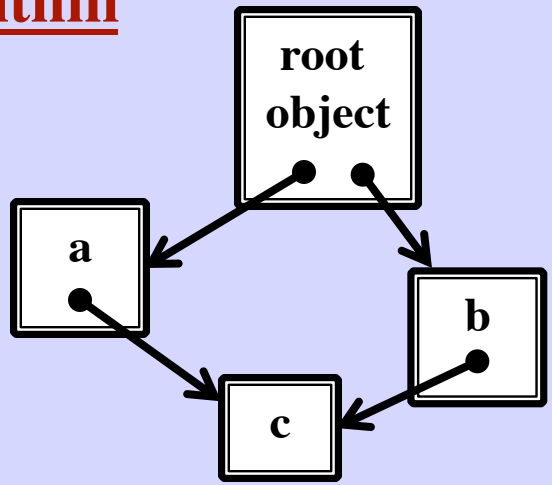
next free location

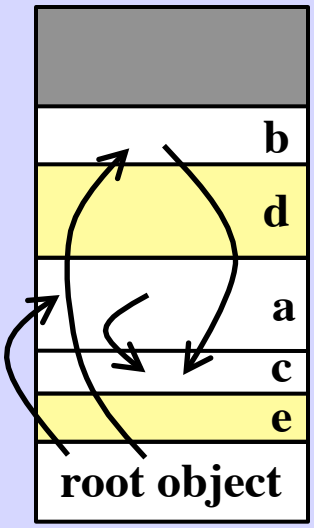
TO-SPACE

Baker's Semi-Space Algorithm

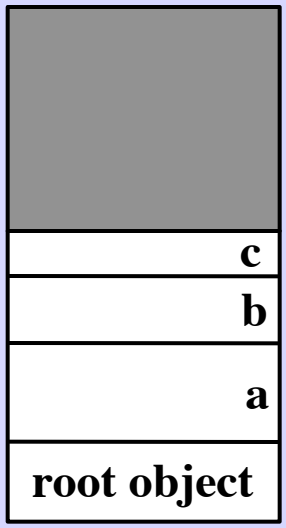
During garbage collection…

- Copy all reachable objects to *TO-SPACE*
- First copy the root object.
- Then scan the next object
  - and copy the objects it points to.
- Until the pointers meet.
- Then swap spaces.





TO-SPACE



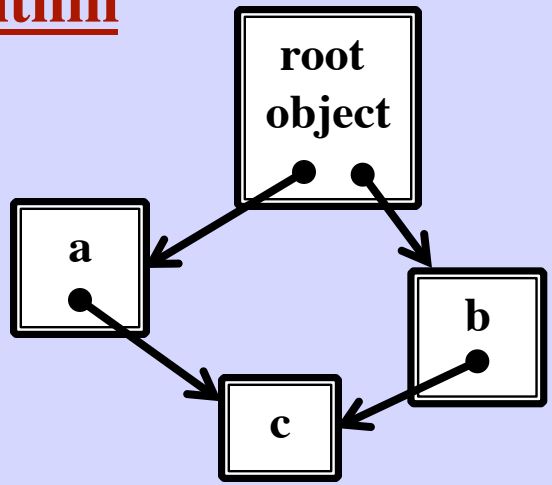
next free location

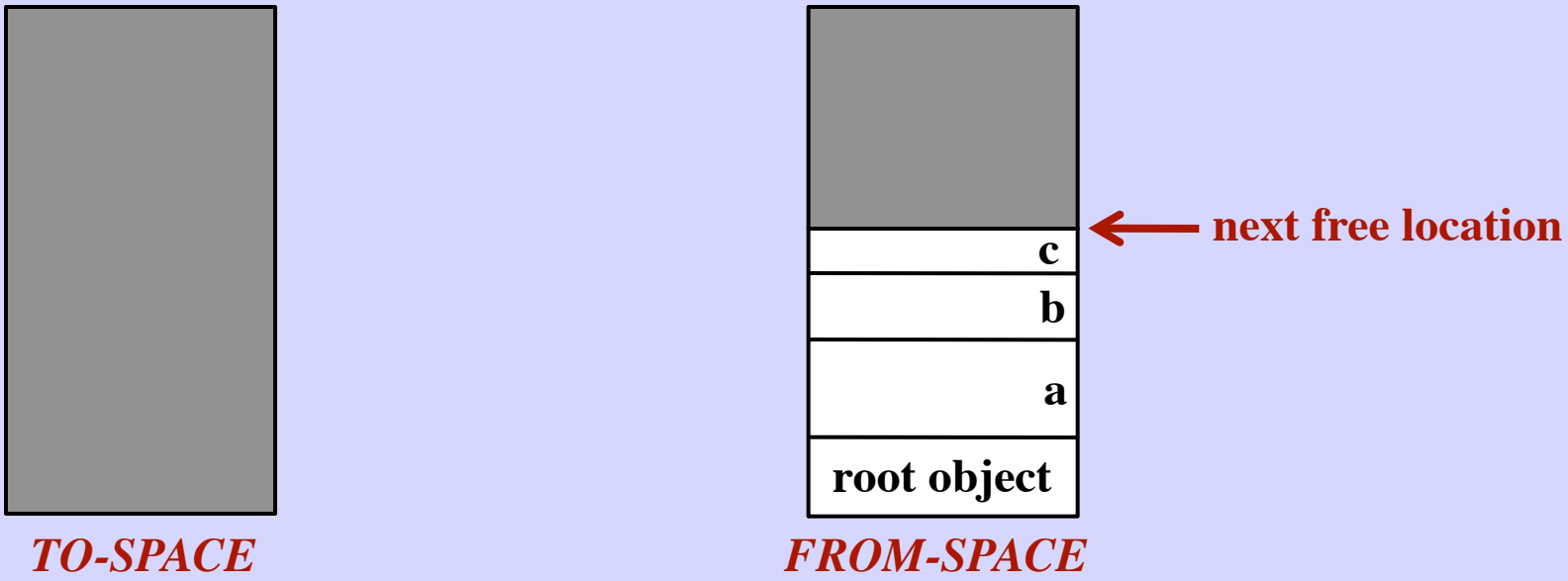
FROM-SPACE

Baker's Semi-Space Algorithm

During garbage collection…

- Copy all reachable objects to *TO-SPACE*
- First copy the root object.
- Then scan the next object
  - and copy the objects it points to.
- Until the pointers meet.
- Then swap spaces.





Baker's Semi-Space Algorithm

Details:

We also need to update all the pointers in the objects.

Whenever we copy an object…

Leave a *"forwarding pointer"* behind in the old object.

Point to the copy in TO-SPACE.

Storage overhead?

OK to overwrite other fields (e.g., size, class)

Will need one bit per object

0 = object not copied (yet)

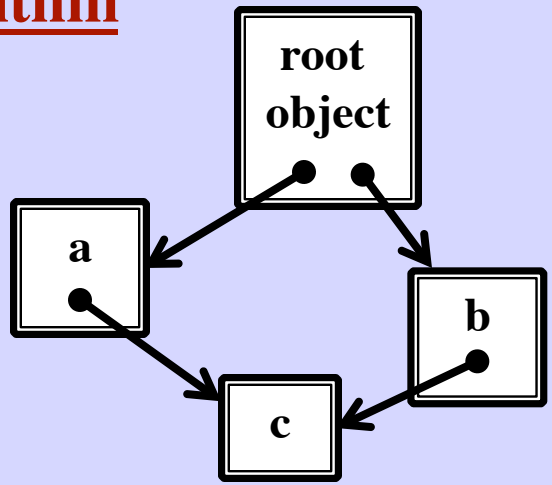
1 = object moved; use forwarding pointer

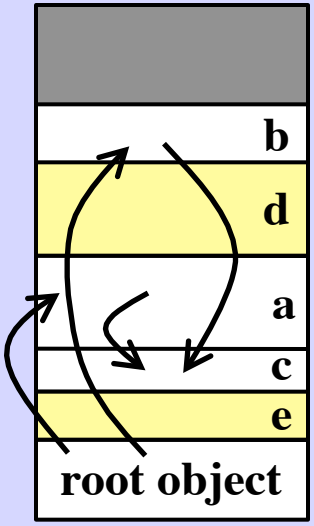
Baker's Semi-Space Algorithm

Will show forwarding pointers this time:

0

1 ---------→





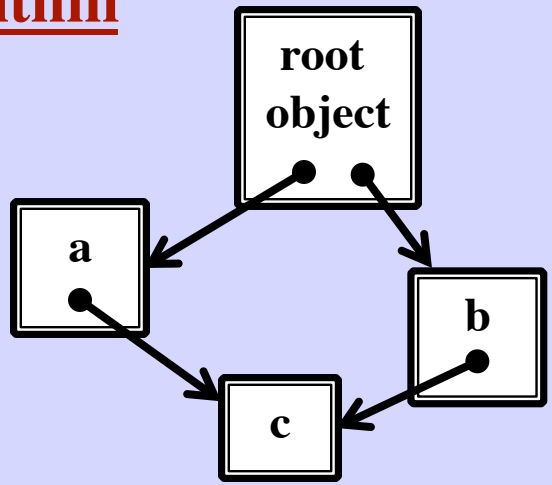
FROM-SPACE

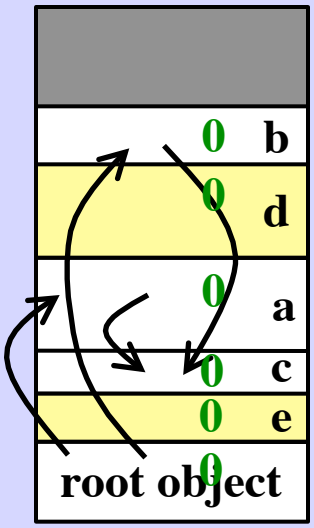
Baker's Semi-Space Algorithm

Will show forwarding pointers this time:

0

1 ---------→

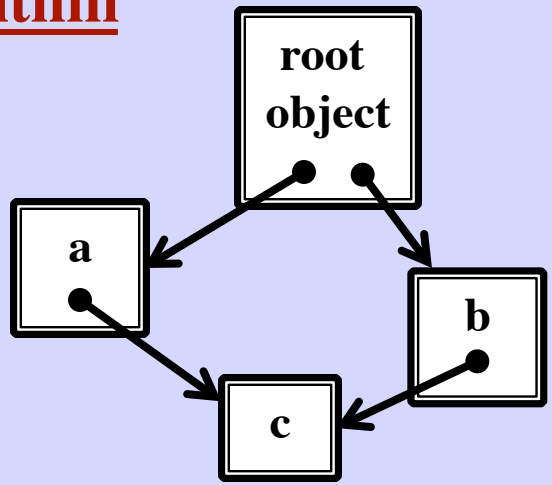


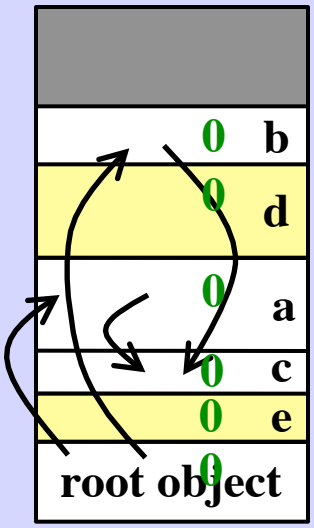


FROM-SPACE

Baker's Semi-Space Algorithm

First copy the root object.

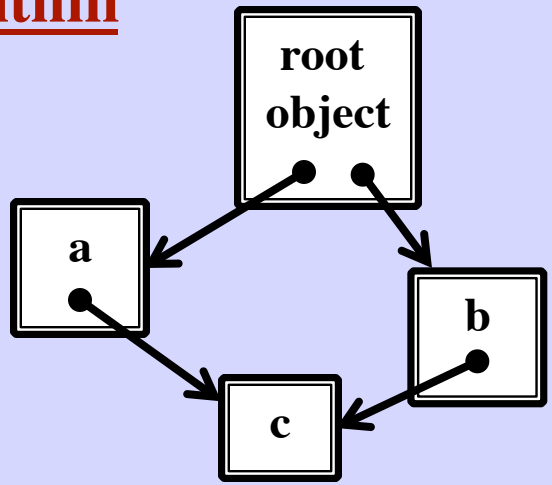


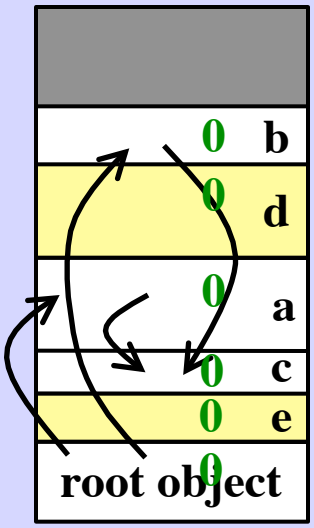


FROM-SPACE

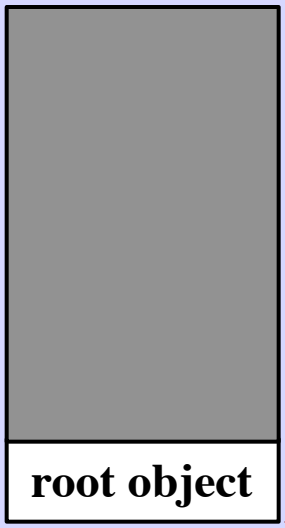
Baker's Semi-Space Algorithm

First copy the root object.





FROM-SPACE



TO-SPACE

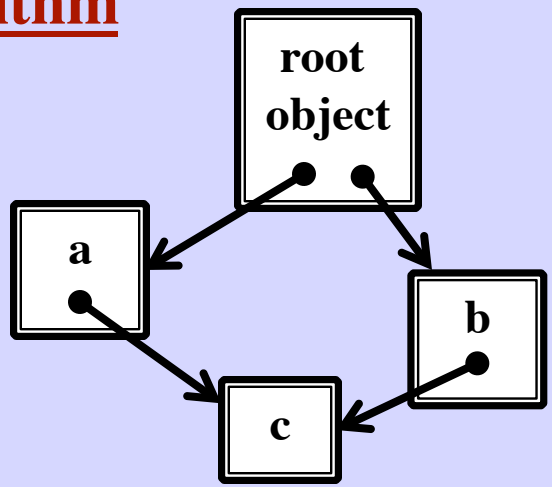
next free location

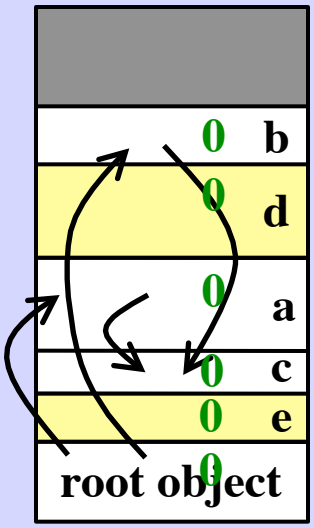
next unscanned location

Baker's Semi-Space Algorithm

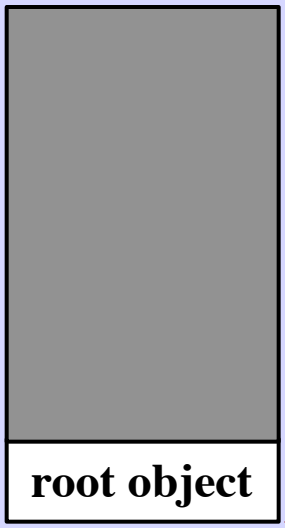
First copy the root object.

Mark it and leave a fowarding pointer





FROM-SPACE



TO-SPACE

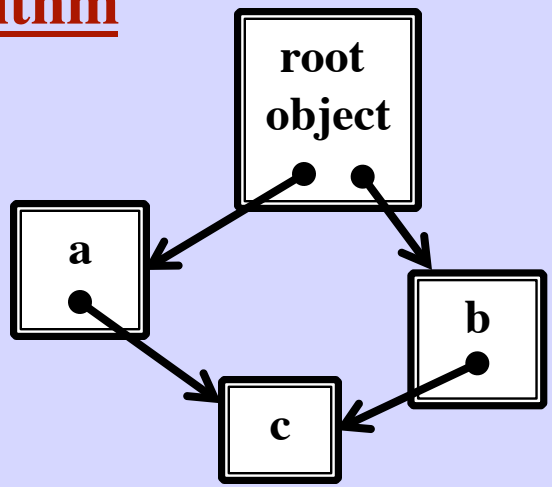
next free location

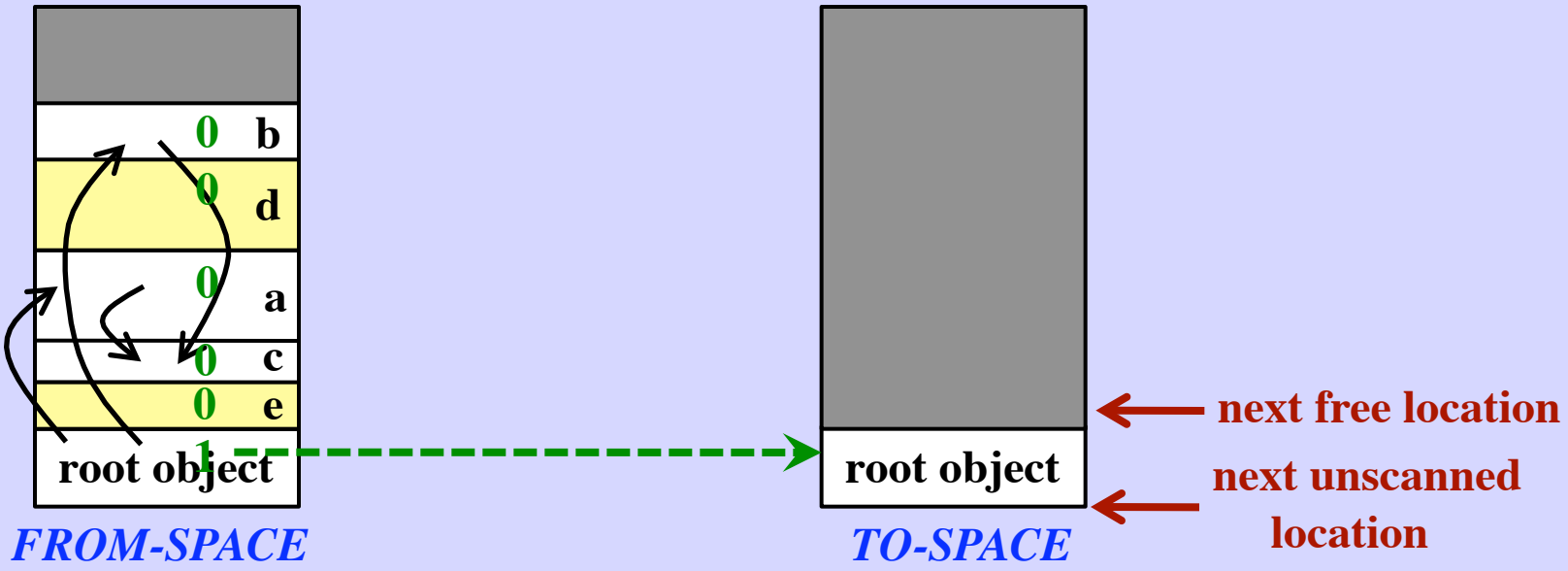
next unscanned location

Baker's Semi-Space Algorithm

First copy the root object.

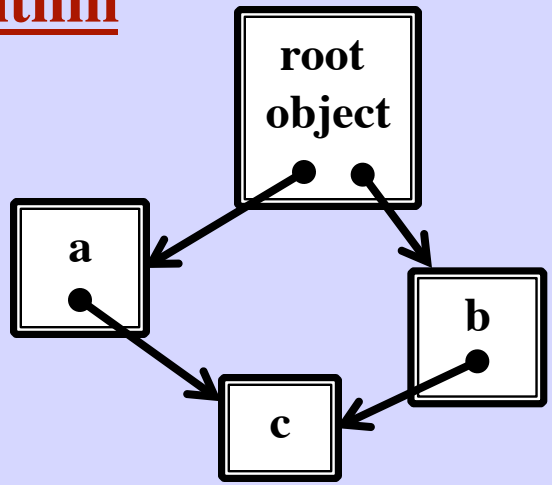
Mark it and leave a fowarding pointer

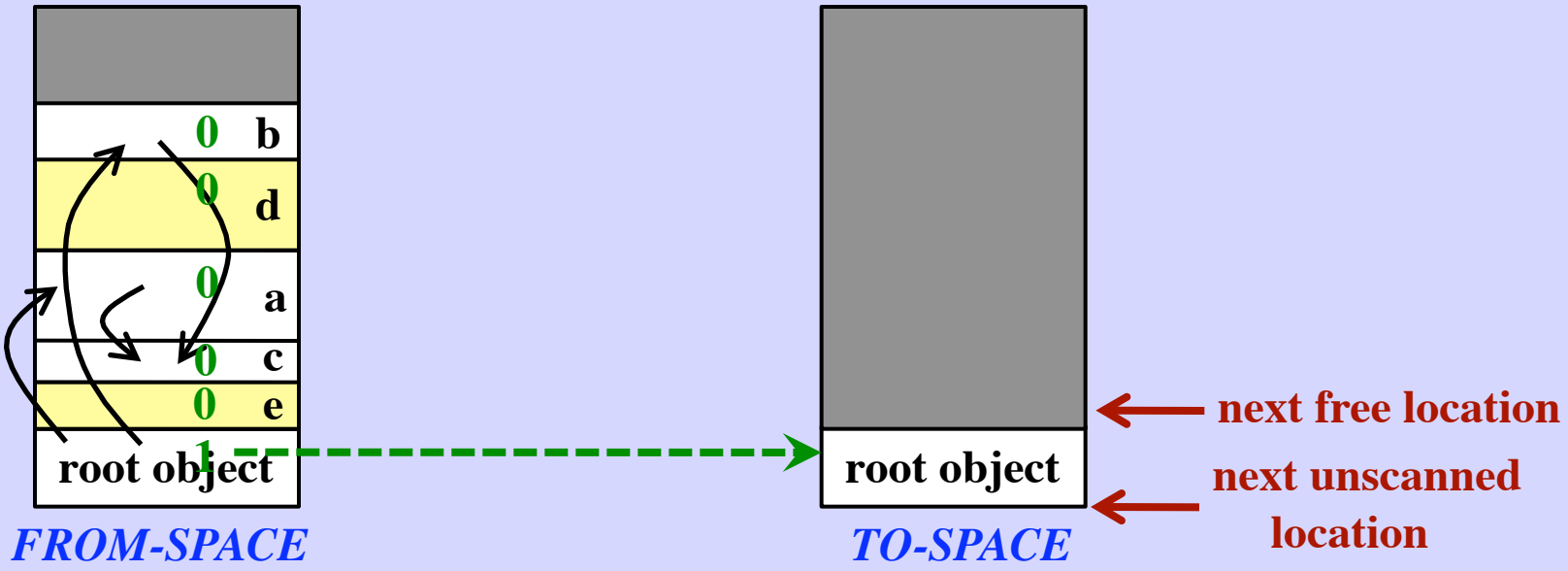




Baker's Semi-Space Algorithm

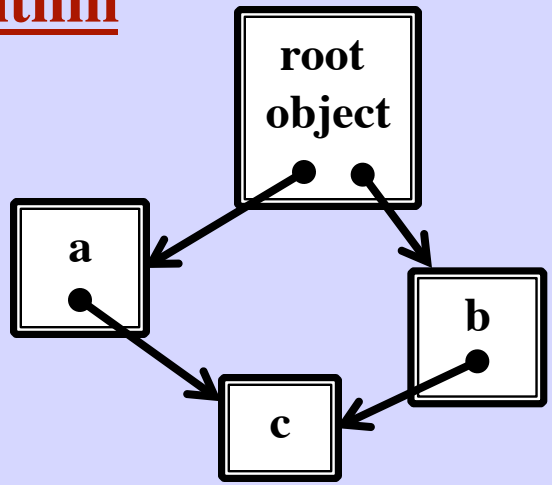
The object contains pointers

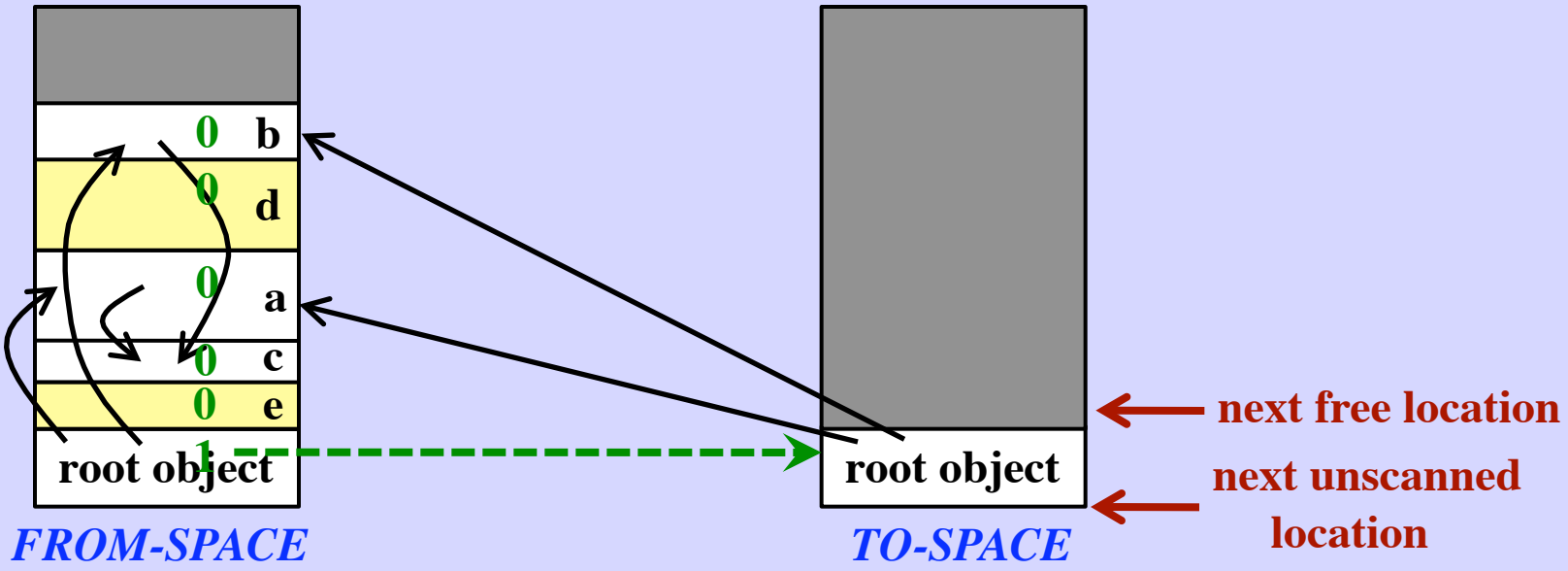




Baker's Semi-Space Algorithm

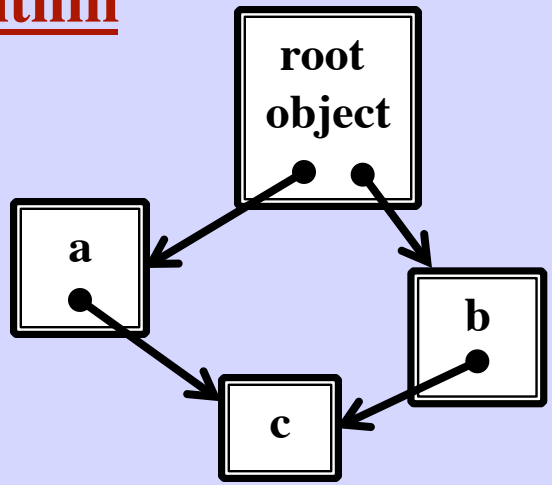
The object contains pointers

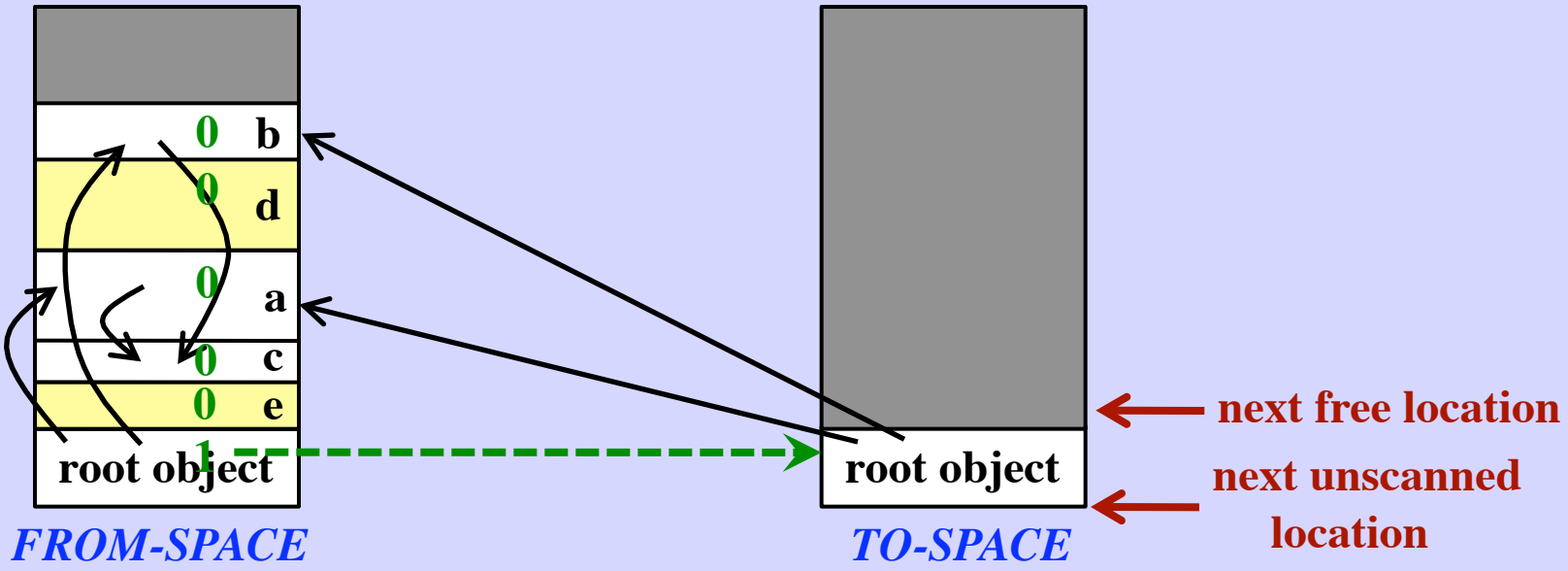




Baker's Semi-Space Algorithm

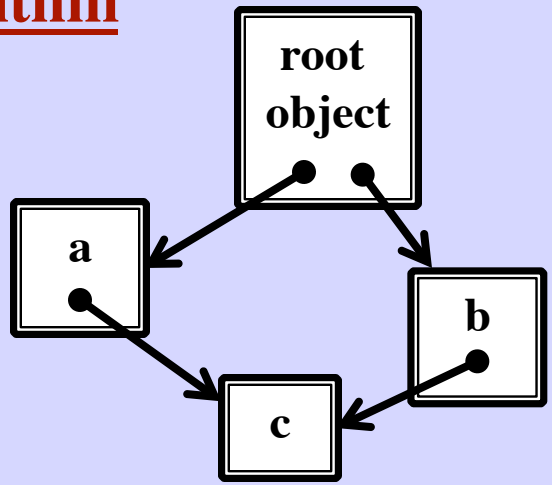
Scan the next object, looking for pointers
into *FROM-SPACE*
Copy these objects.
Leave behind forwarding pointers.
Update the pointers in this object.

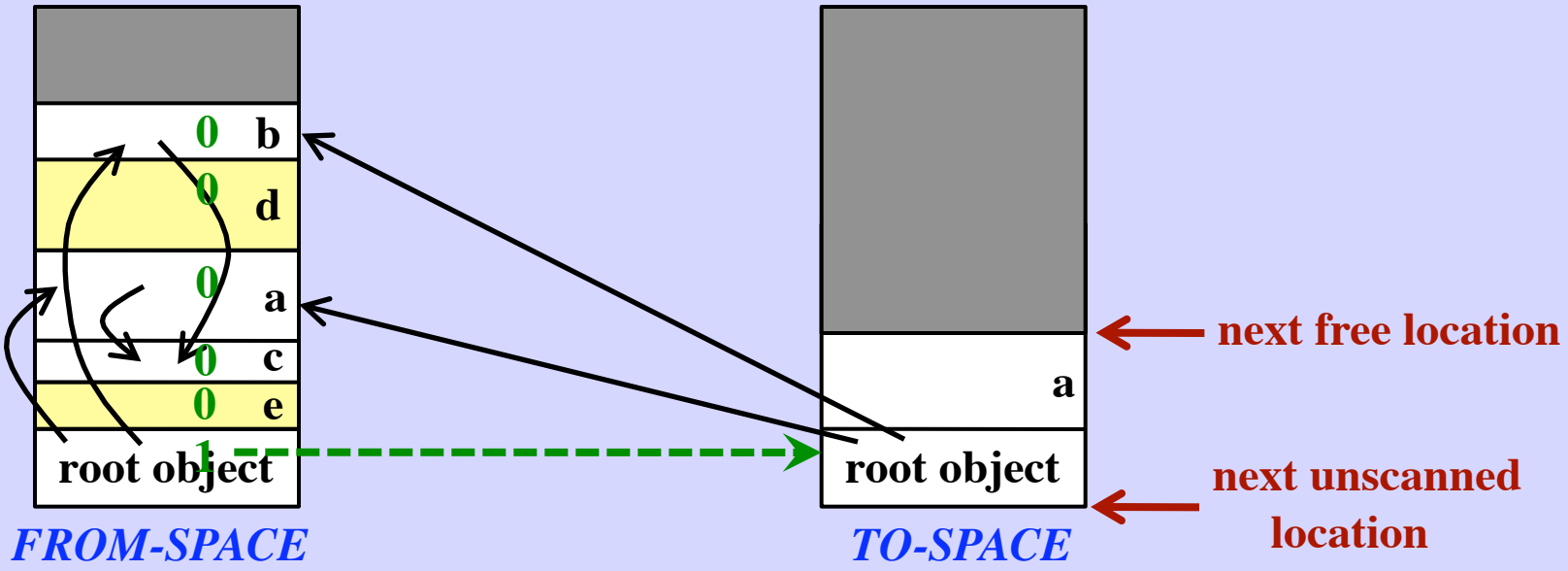




Baker's Semi-Space Algorithm

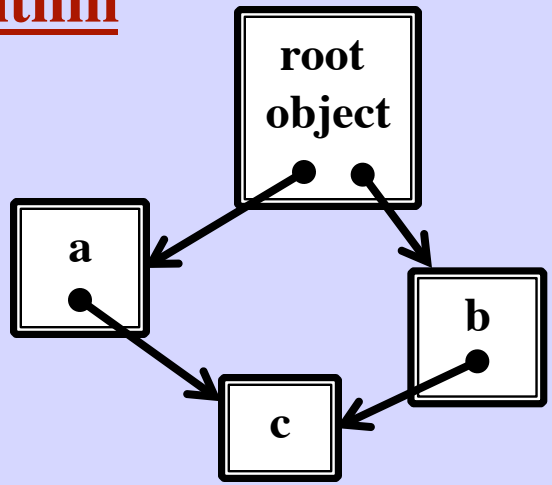
Scan the next object, looking for pointers
into *FROM-SPACE*
Copy these objects.
Leave behind forwarding pointers.
Update the pointers in this object.

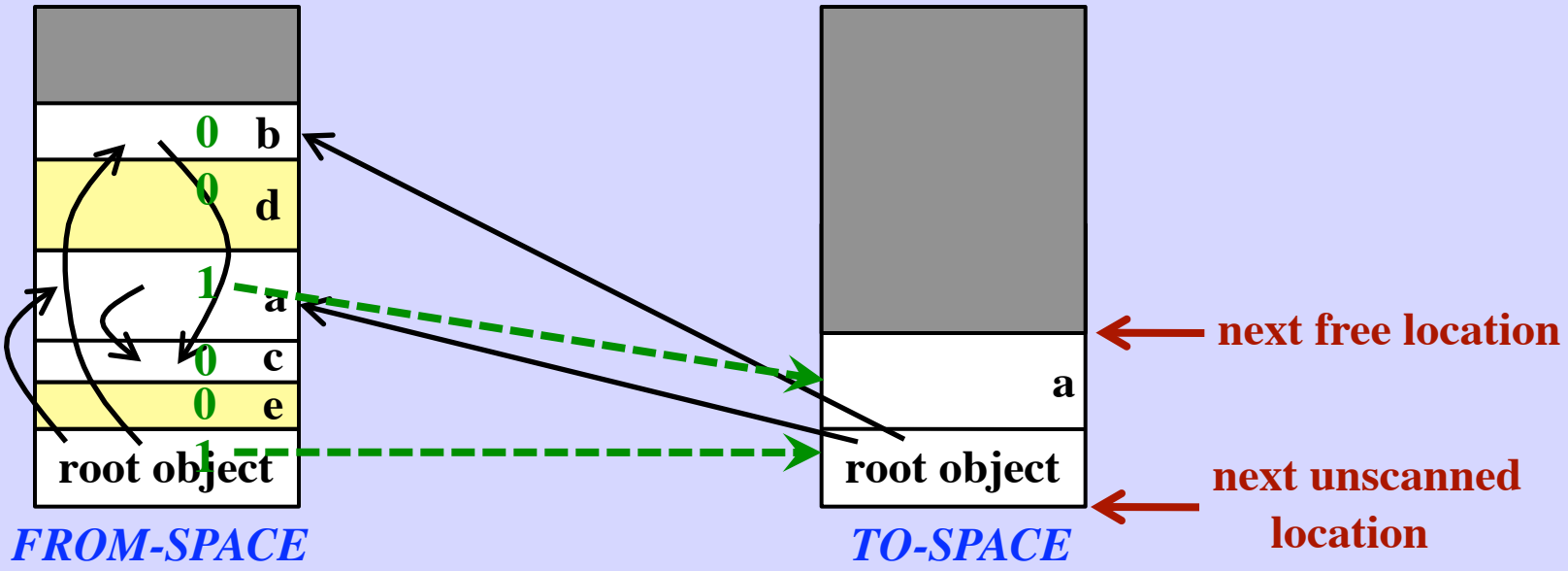




Baker's Semi-Space Algorithm

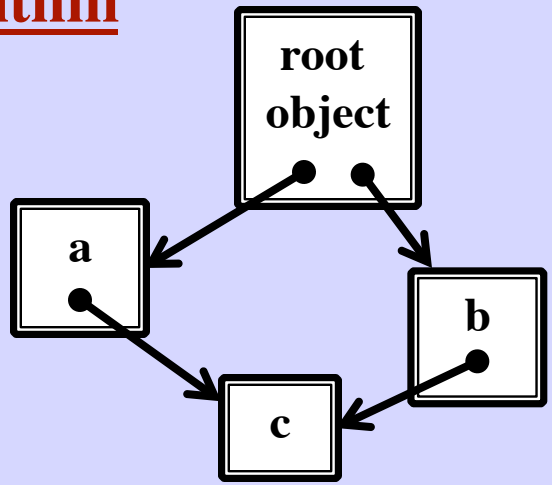
Scan the next object, looking for pointers
into *FROM-SPACE*
Copy these objects.
Leave behind forwarding pointers.
Update the pointers in this object.

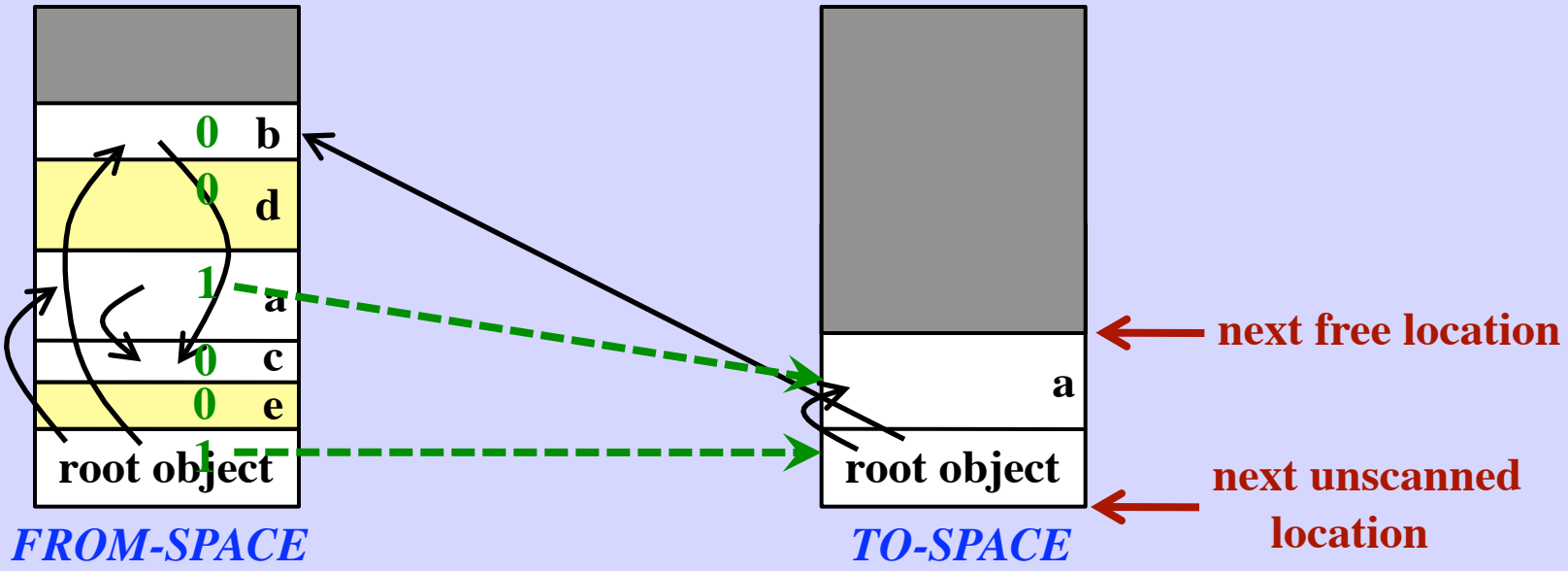




Baker's Semi-Space Algorithm

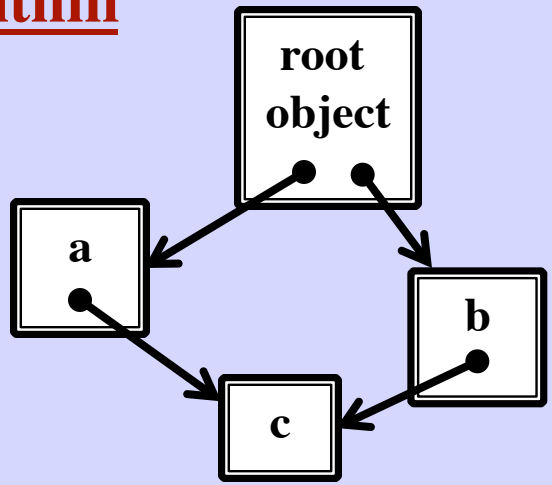
Scan the next object, looking for pointers
into *FROM-SPACE*
Copy these objects.
Leave behind forwarding pointers.
Update the pointers in this object.

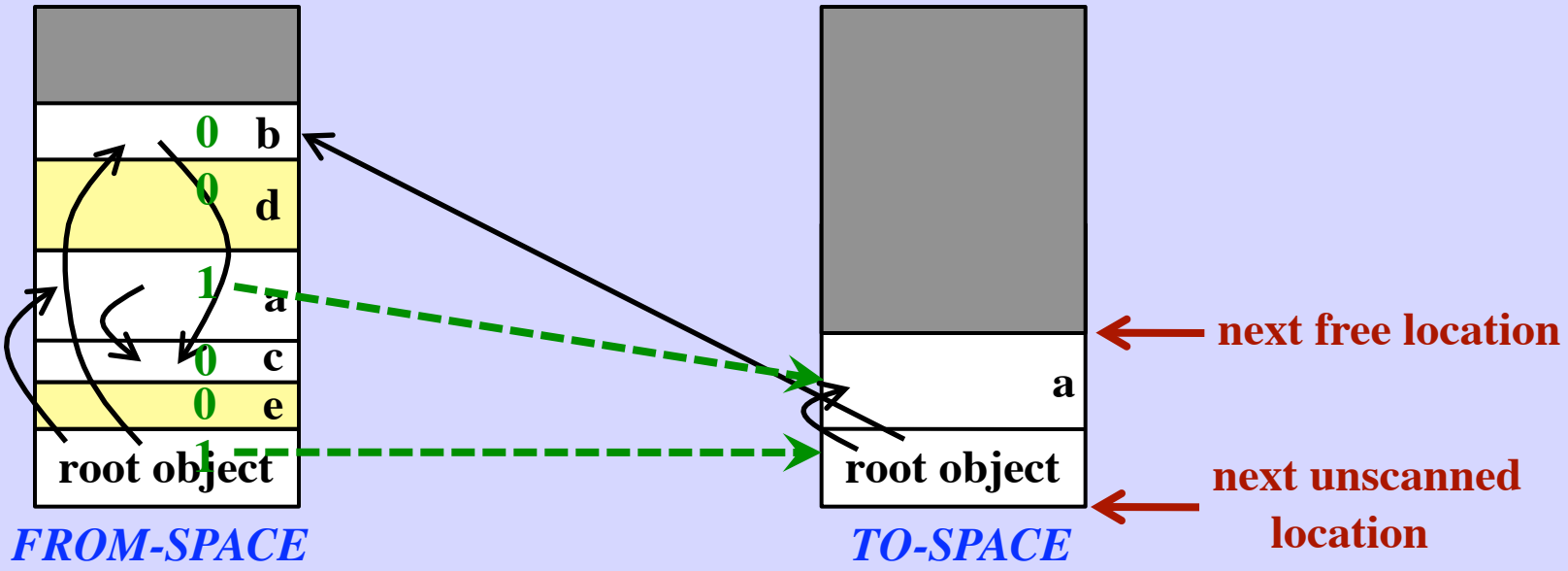




Baker's Semi-Space Algorithm

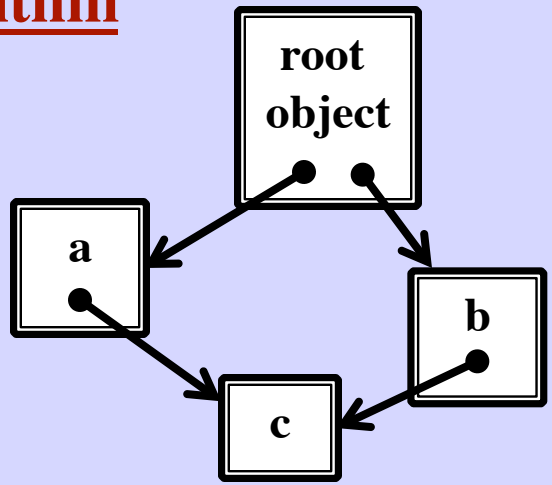
Scan the next object, looking for pointers
into *FROM-SPACE*
Copy these objects.
Leave behind forwarding pointers.
Update the pointers in this object.
(Note: the copied objects contain pointers
into *FROM-SPACE*.)

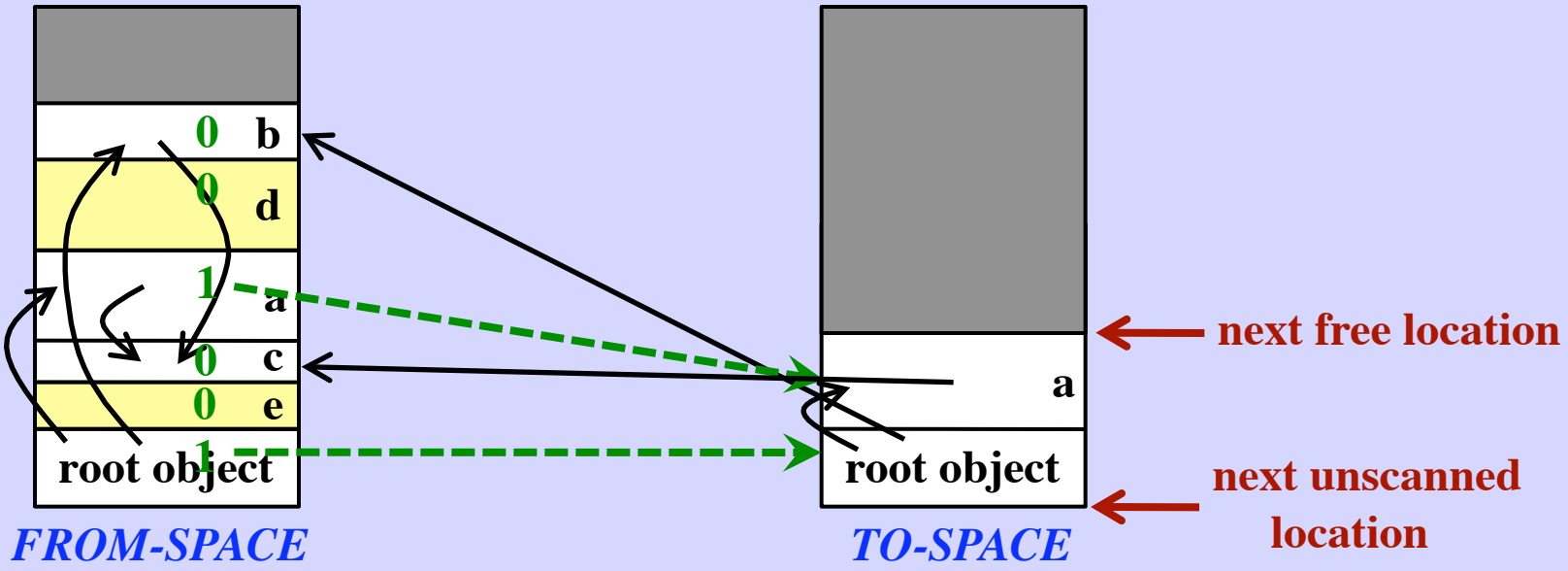




Baker's Semi-Space Algorithm

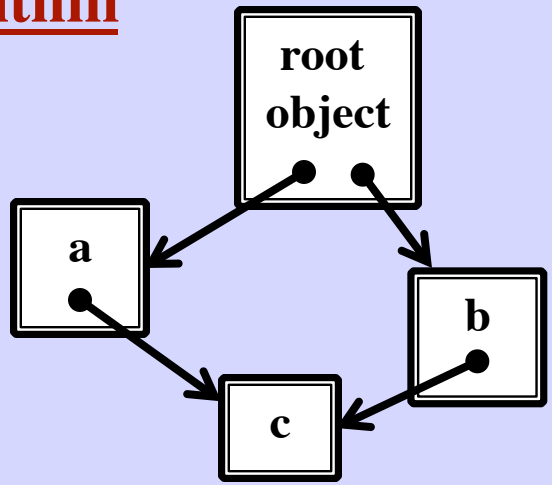
Scan the next object, looking for pointers
into *FROM-SPACE*
Copy these objects.
Leave behind forwarding pointers.
Update the pointers in this object.
(Note: the copied objects contain pointers
into *FROM-SPACE*.)

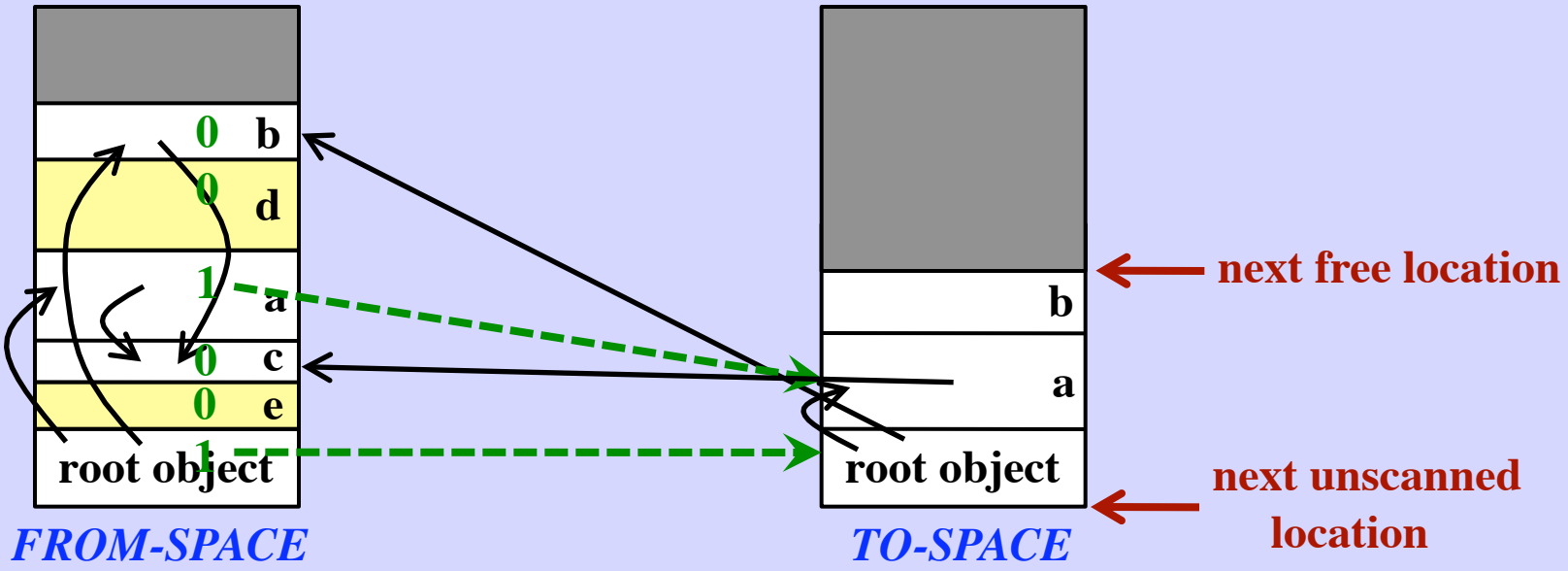




Baker's Semi-Space Algorithm

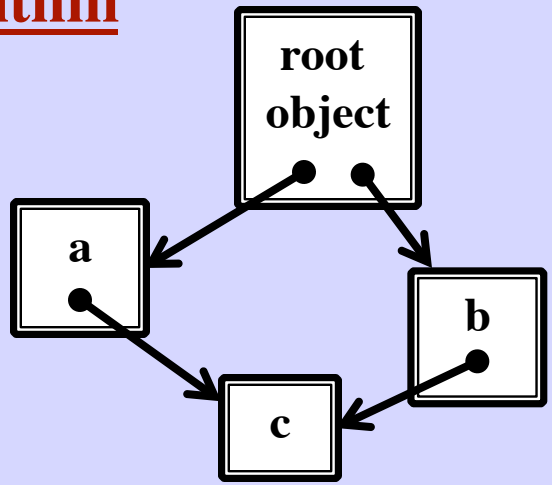
Scan the next object, looking for pointers
into *FROM-SPACE*
Copy these objects.
Leave behind forwarding pointers.
Update the pointers in this object.
(Note: the copied objects contain pointers
into *FROM-SPACE*.)

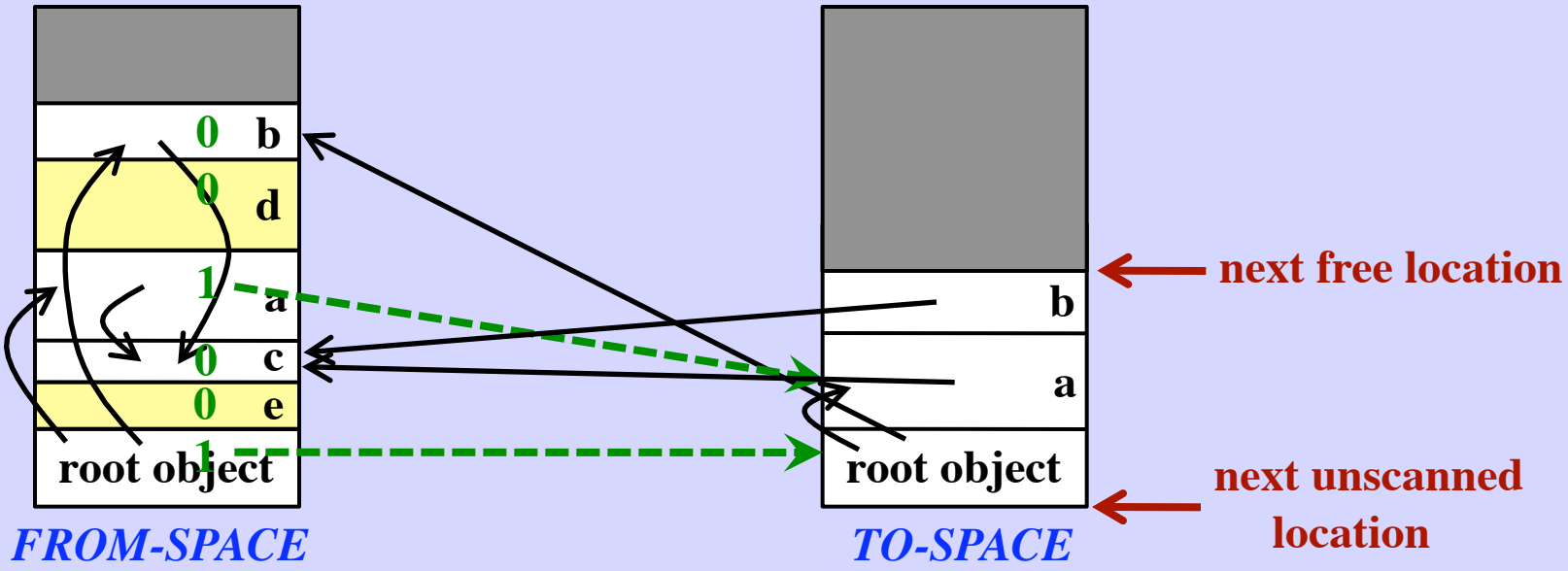




Baker's Semi-Space Algorithm

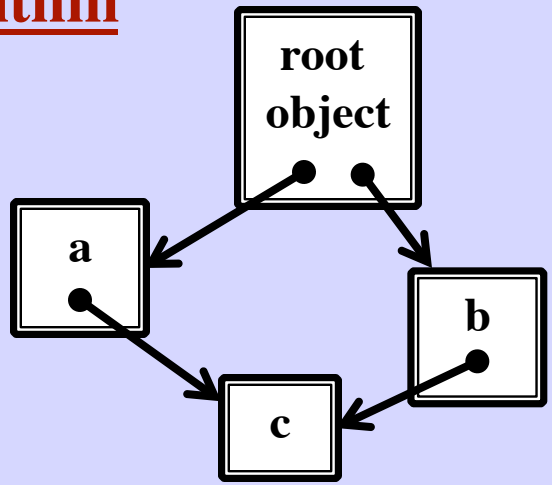
Scan the next object, looking for pointers
into *FROM-SPACE*
Copy these objects.
Leave behind forwarding pointers.
Update the pointers in this object.
(Note: the copied objects contain pointers
into *FROM-SPACE*.)

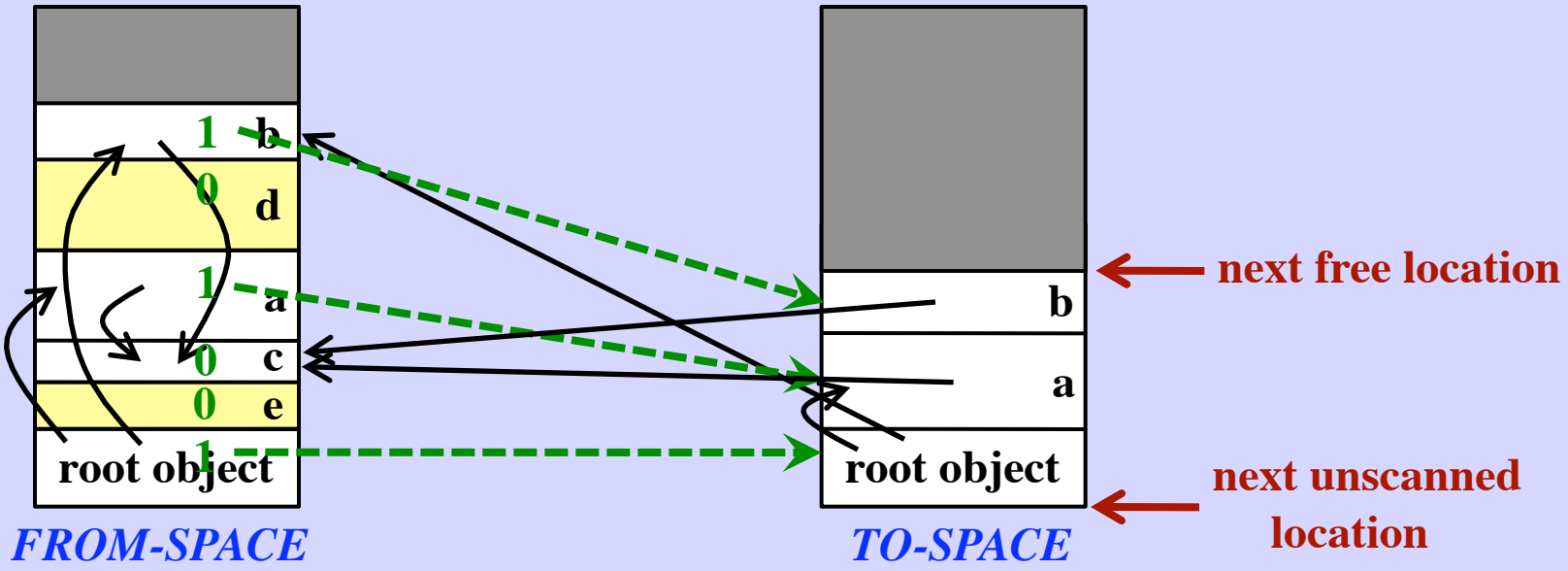




Baker's Semi-Space Algorithm

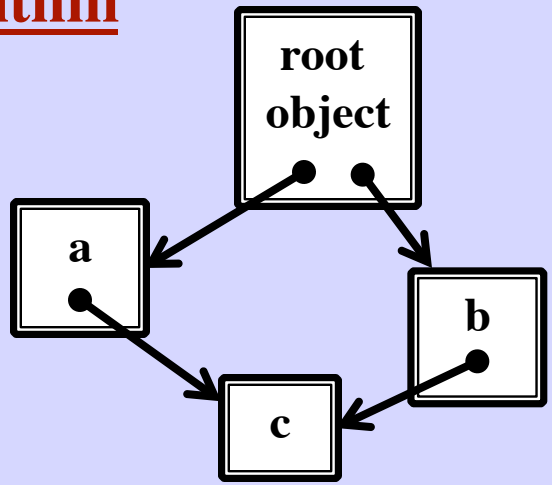
Scan the next object, looking for pointers into *FROM-SPACE*
Copy these objects.
Leave behind forwarding pointers.
Update the pointers in this object.
(Note: the copied objects contain pointers into *FROM-SPACE*.)

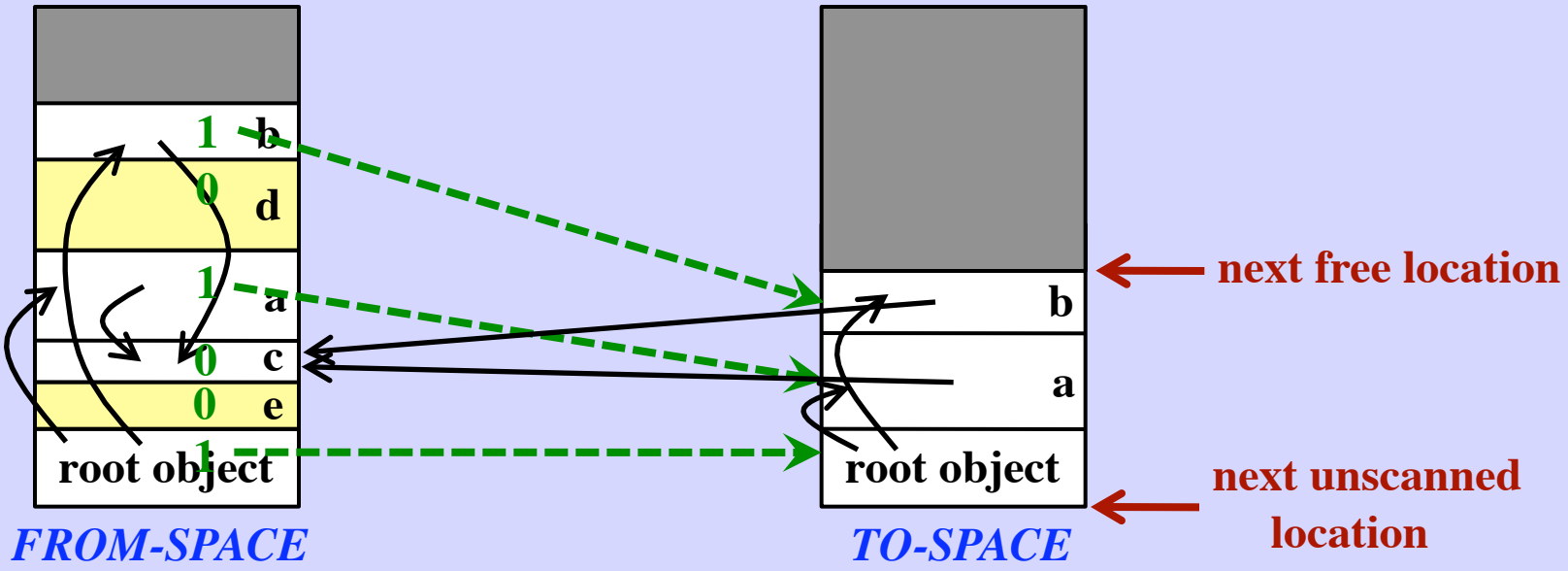




Baker's Semi-Space Algorithm

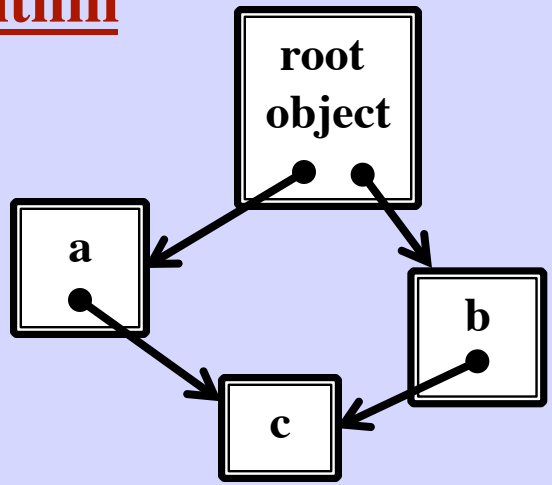
Scan the next object, looking for pointers
into *FROM-SPACE*
Copy these objects.
Leave behind forwarding pointers.
Update the pointers in this object.
(Note: the copied objects contain pointers
into *FROM-SPACE*.)

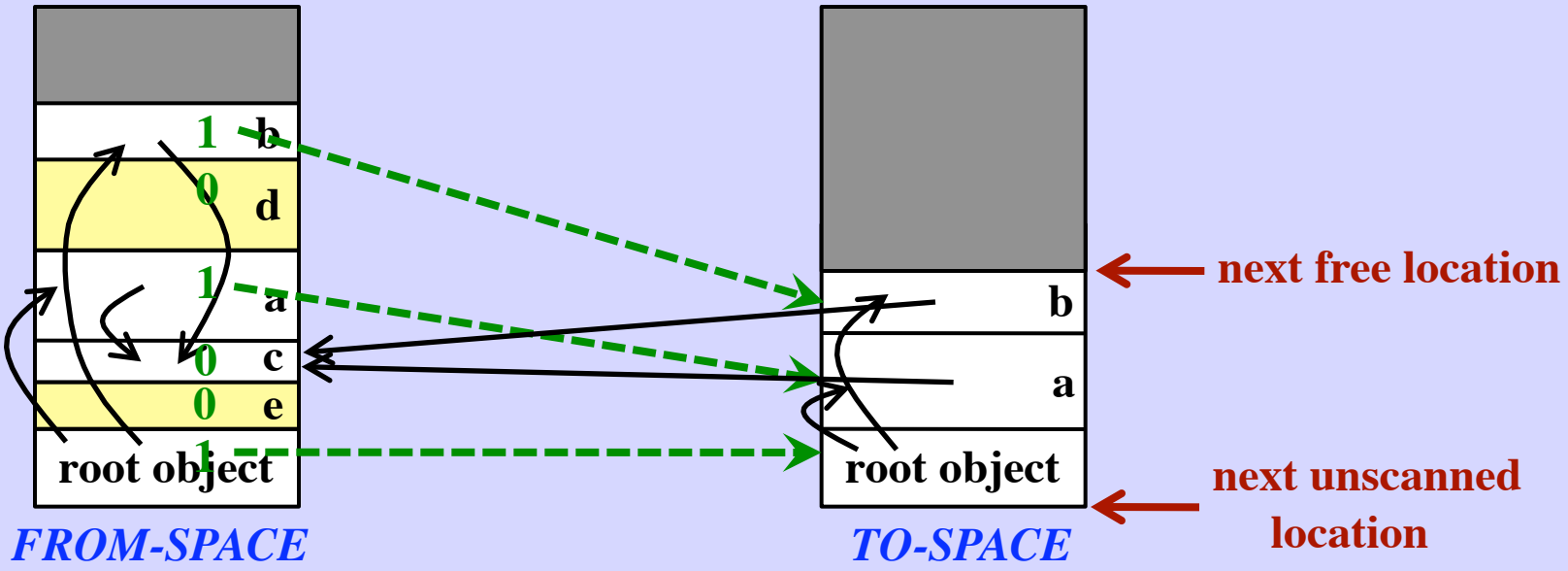




Baker's Semi-Space Algorithm

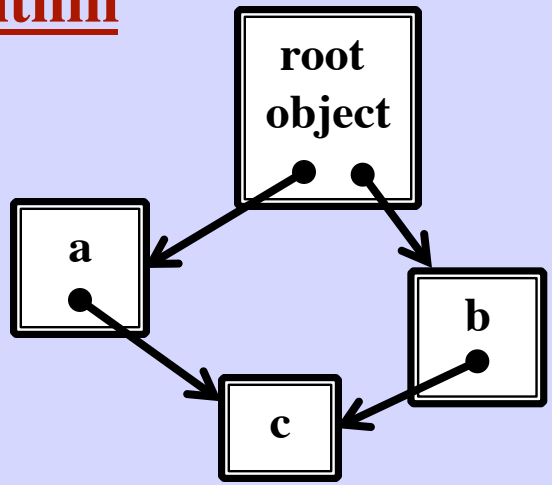
Now we are done with this object.
Move on to next object.

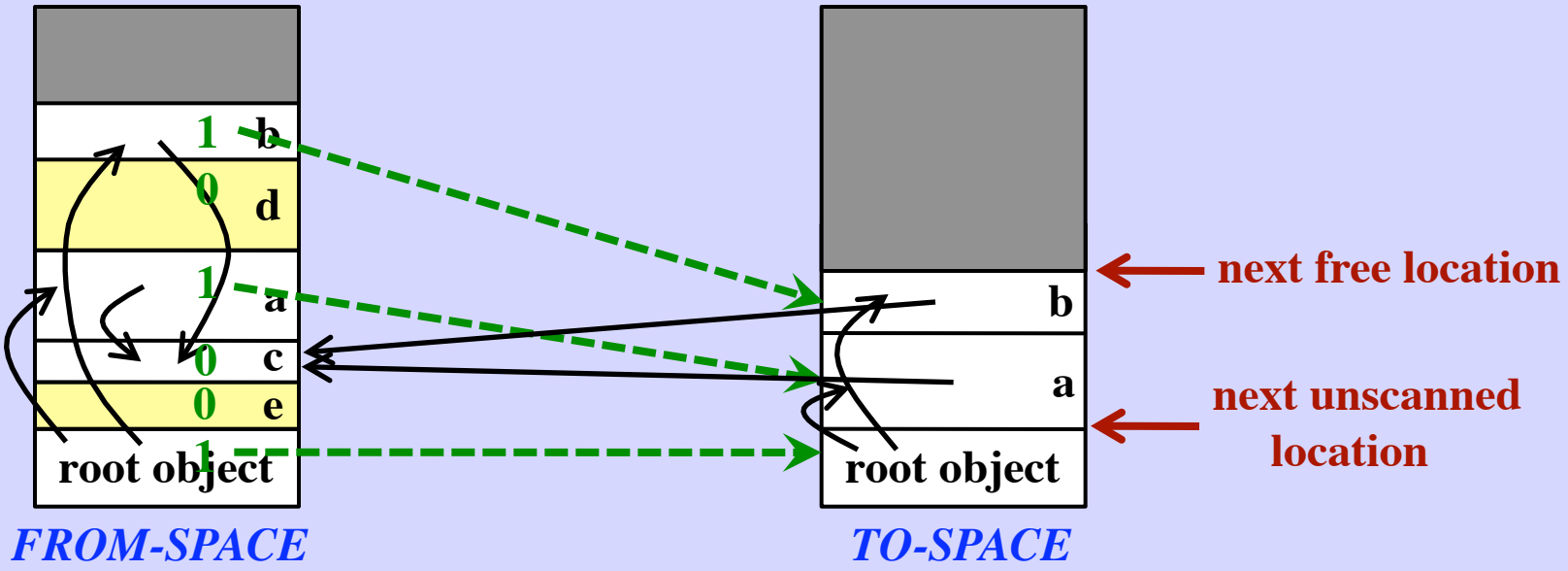




Baker's Semi-Space Algorithm

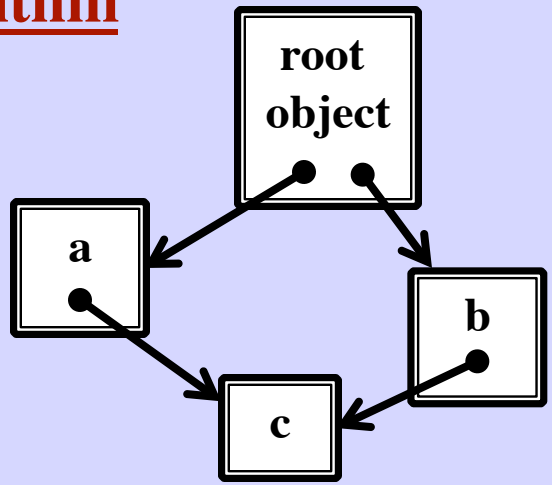
Now we are done with this object.
Move on to next object.

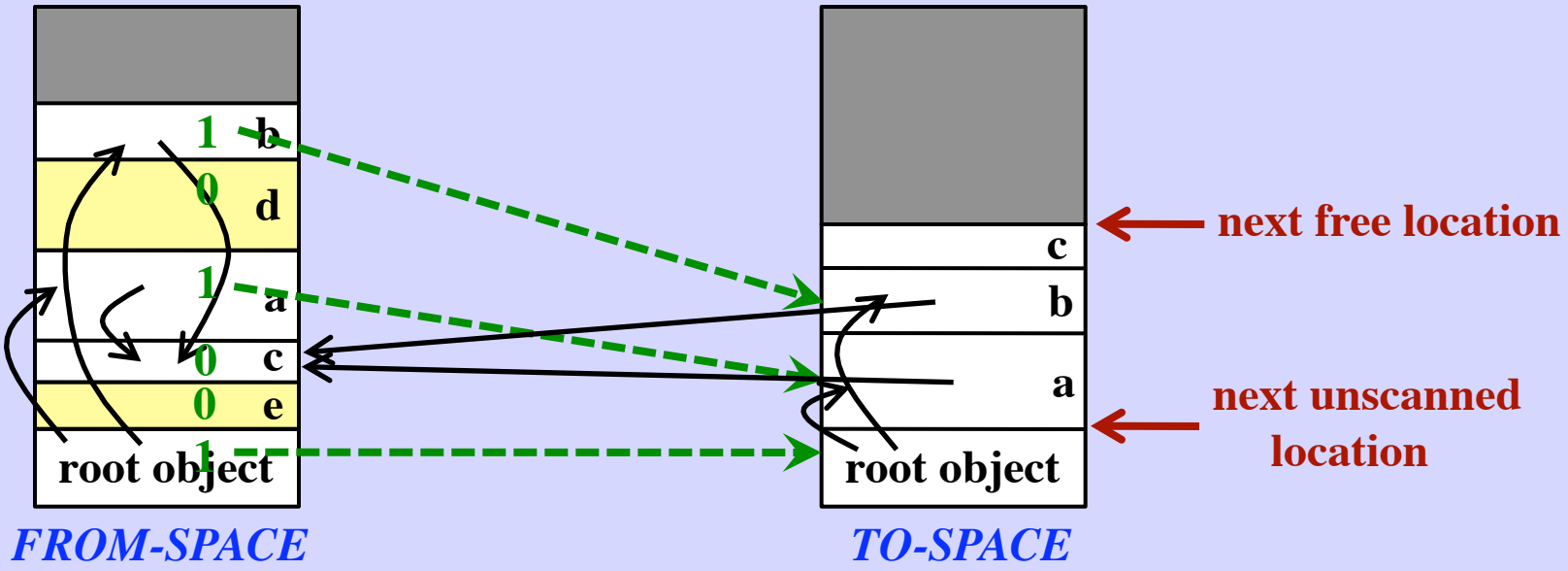




Baker's Semi-Space Algorithm

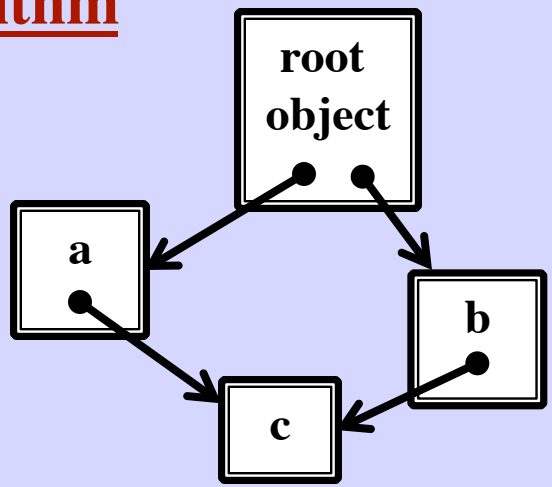
Now we are done with this object.
Move on to next object.

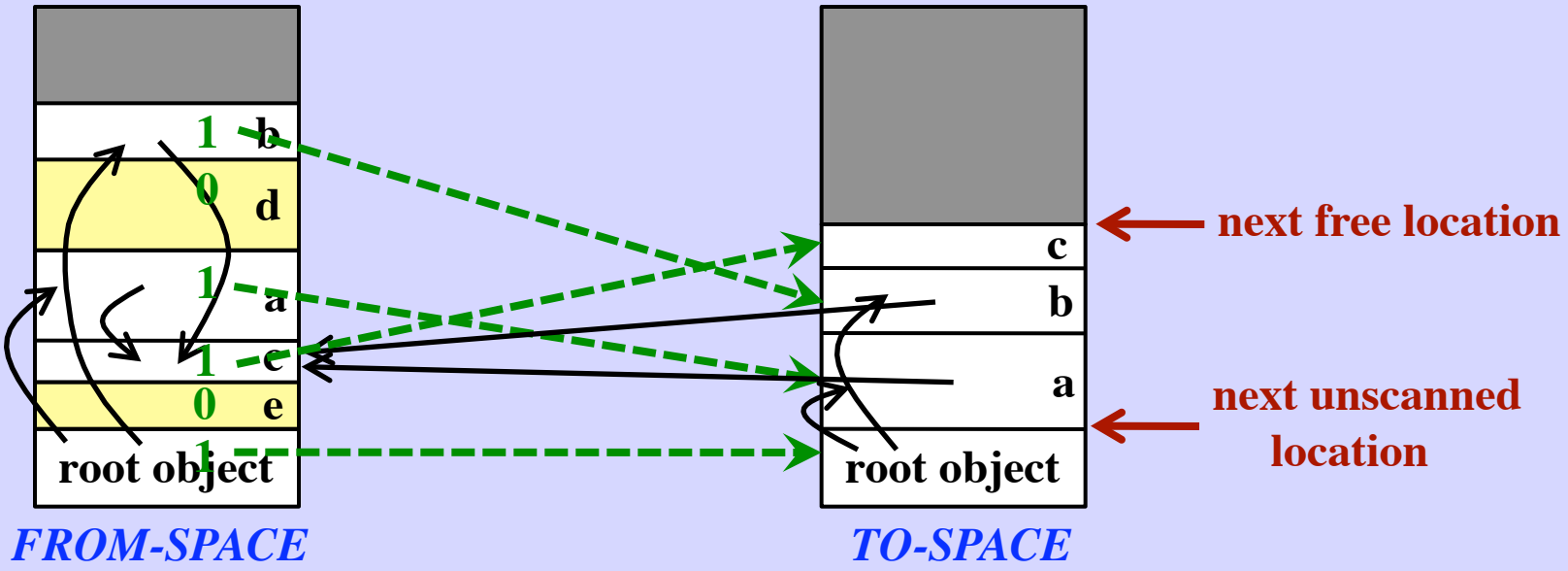




Baker's Semi-Space Algorithm

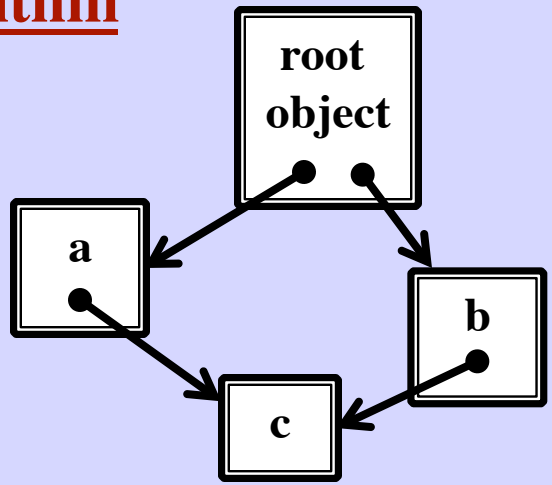
Now we are done with this object.
Move on to next object.

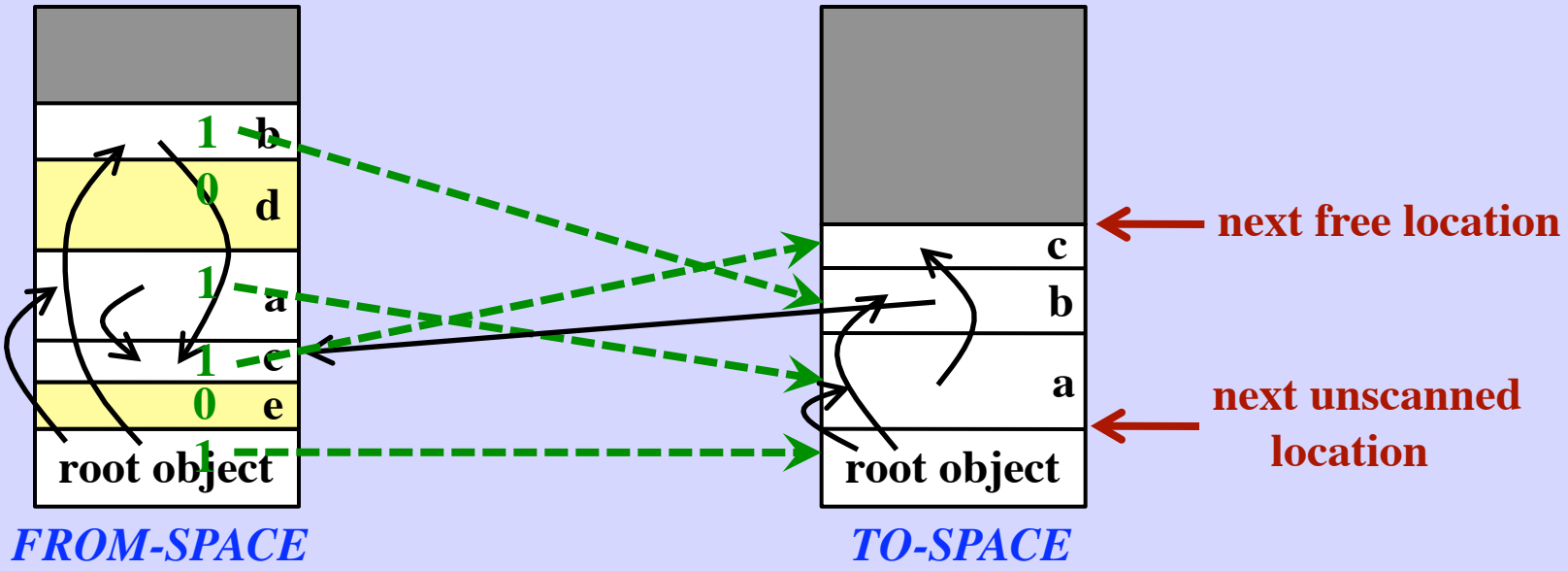




Baker's Semi-Space Algorithm

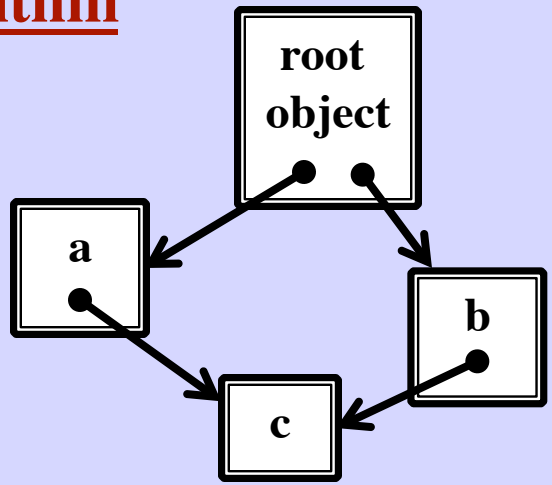
Now we are done with this object.
Move on to next object.

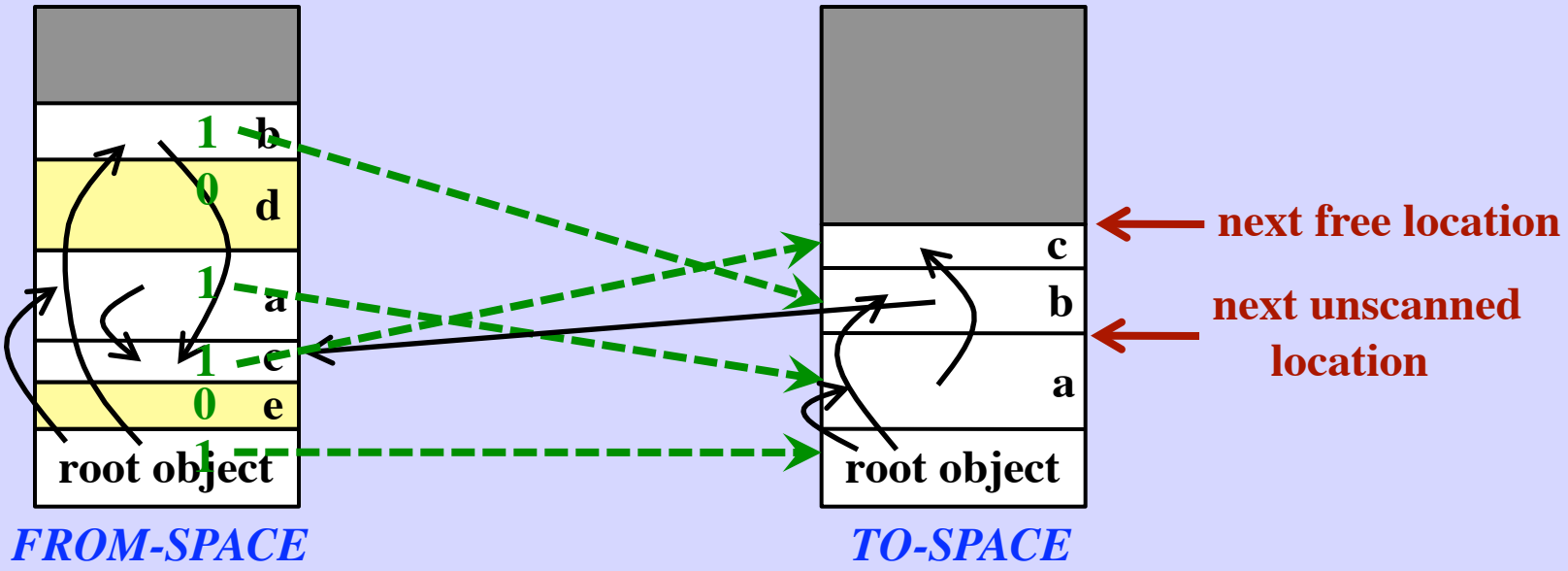




Baker's Semi-Space Algorithm

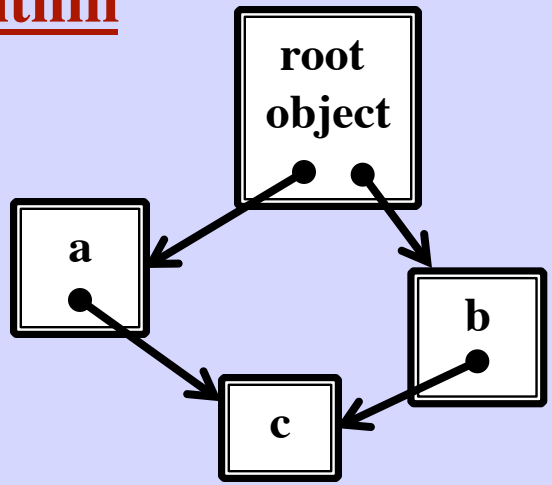
Now we are done with this object.
Move on to next object.

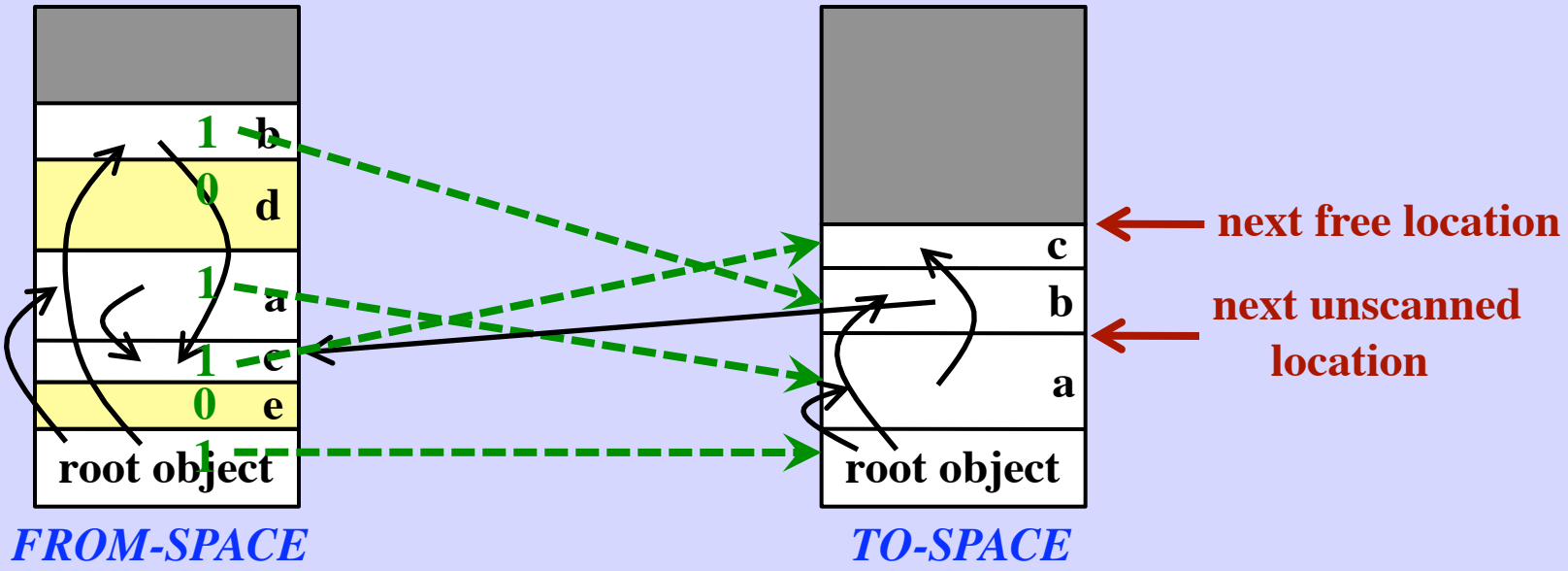




Baker's Semi-Space Algorithm

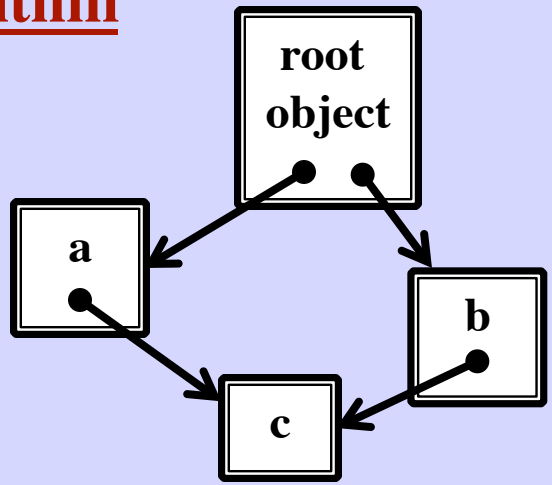
"b" contains a pointer into FROM-SPACE
But that object is marked with 1.
It has already been copied.
Just update the pointer.

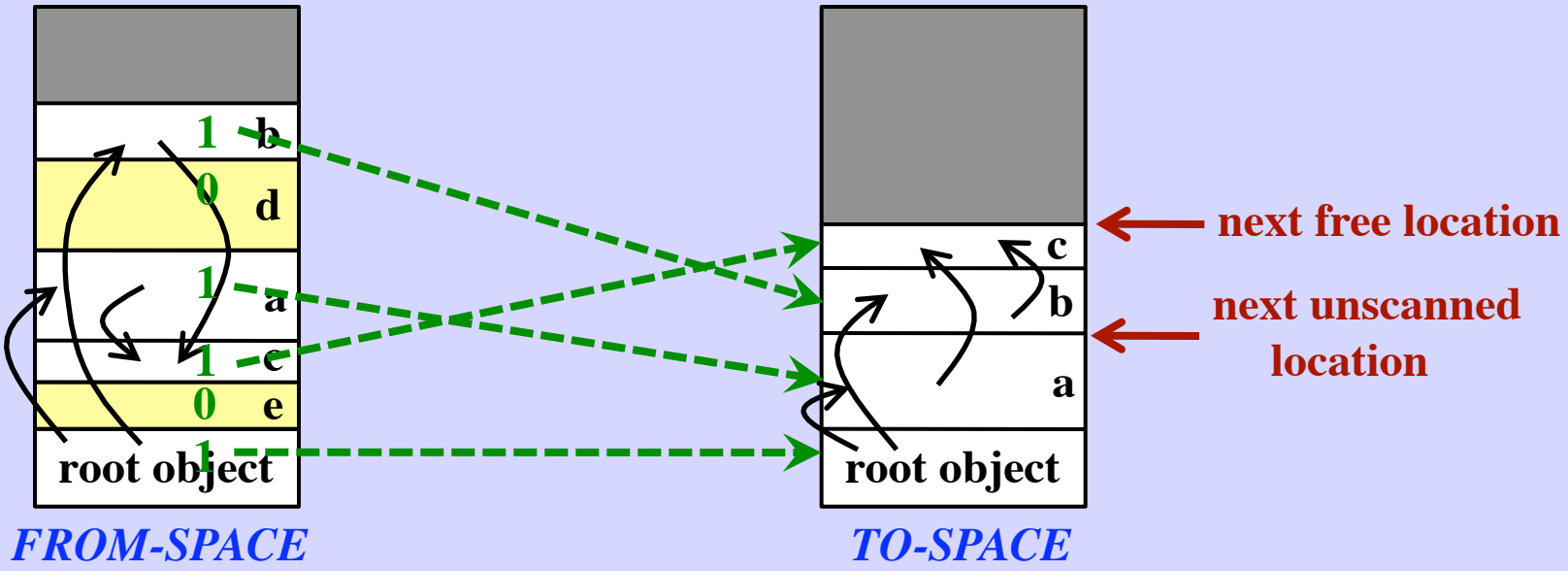




Baker's Semi-Space Algorithm

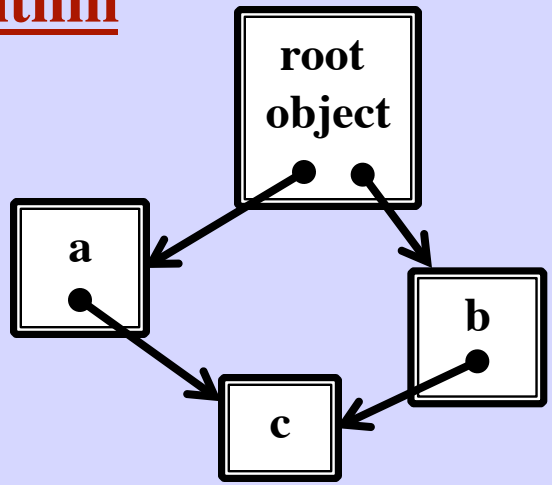
"b" contains a pointer into FROM-SPACE
But that object is marked with 1.
It has already been copied.
Just update the pointer.

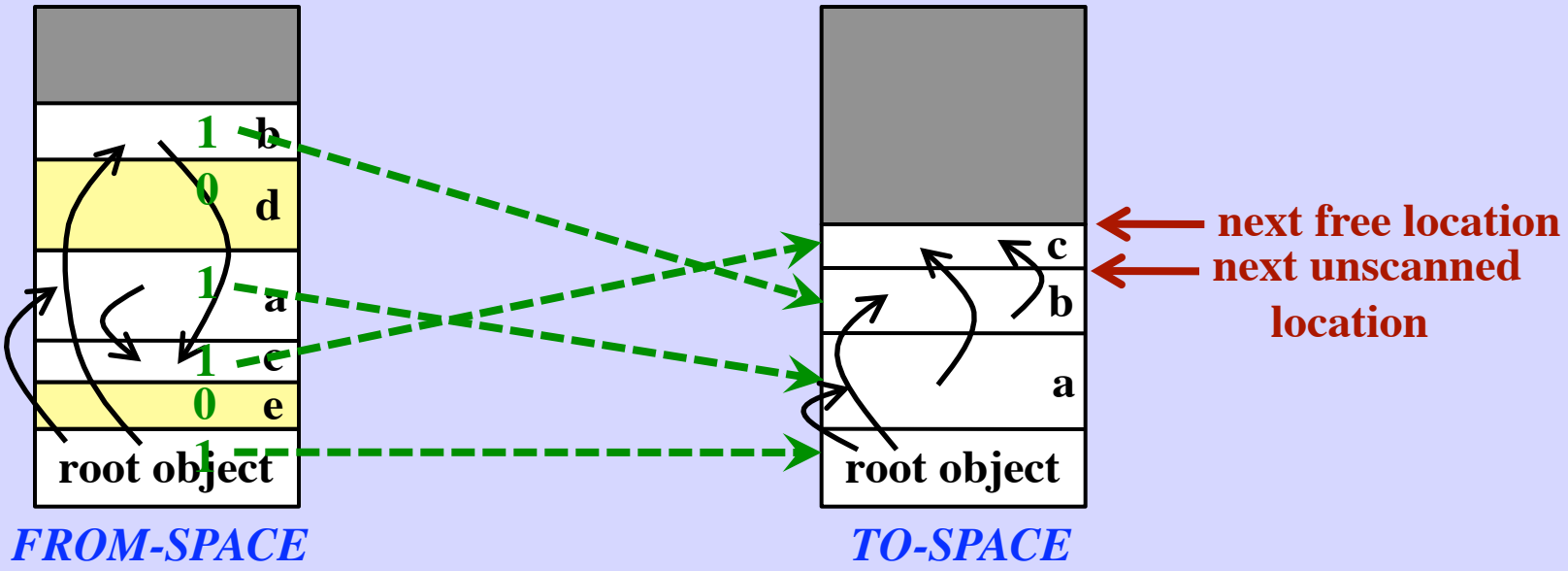




Baker's Semi-Space Algorithm

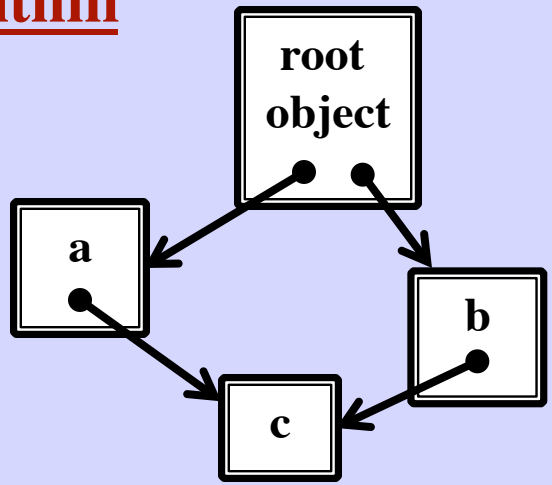
"b" contains a pointer into FROM-SPACE
But that object is marked with 1.
It has already been copied.
Just update the pointer.

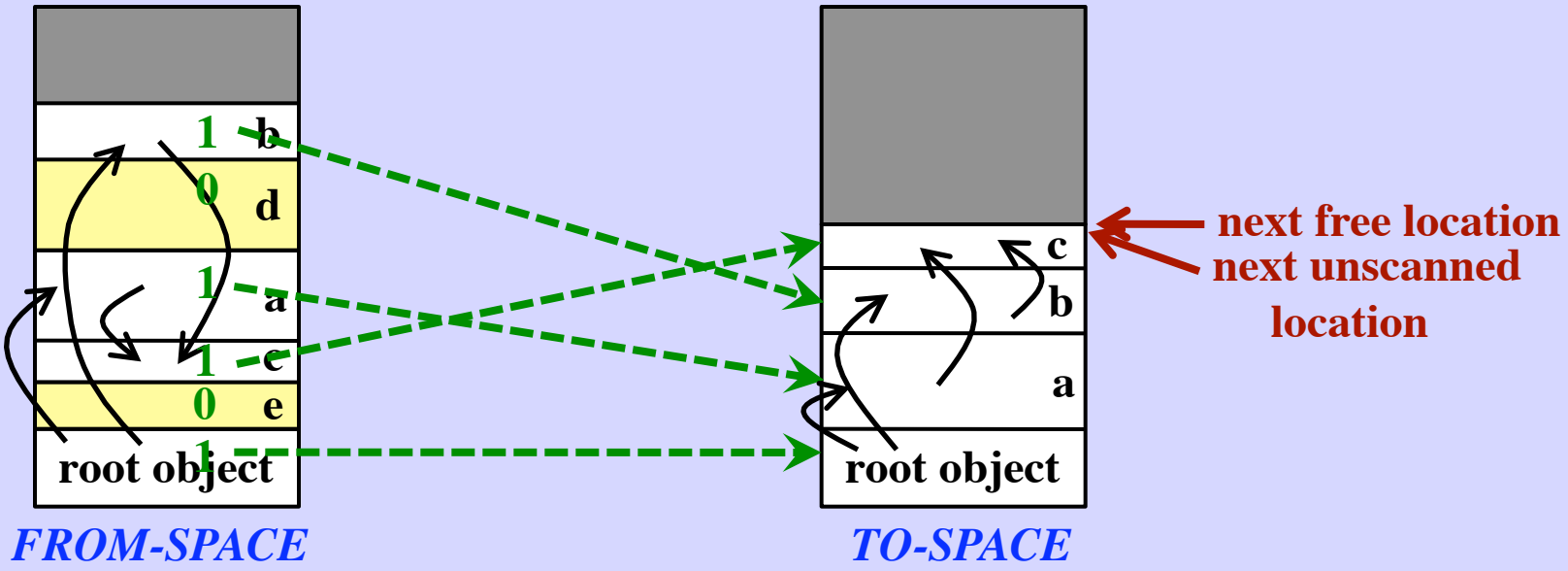




Baker's Semi-Space Algorithm

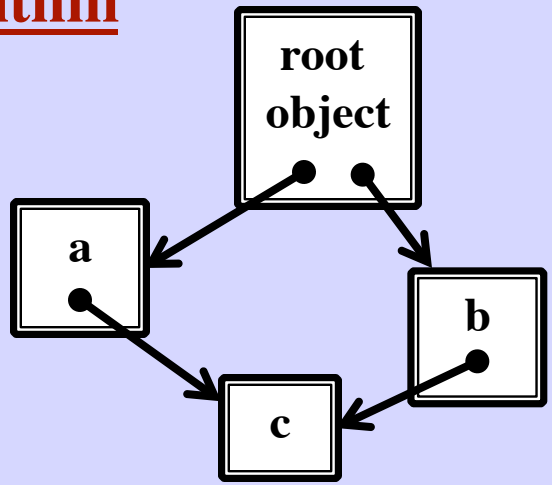
"b" contains a pointer into FROM-SPACE
But that object is marked with 1.
It has already been copied.
Just update the pointer.

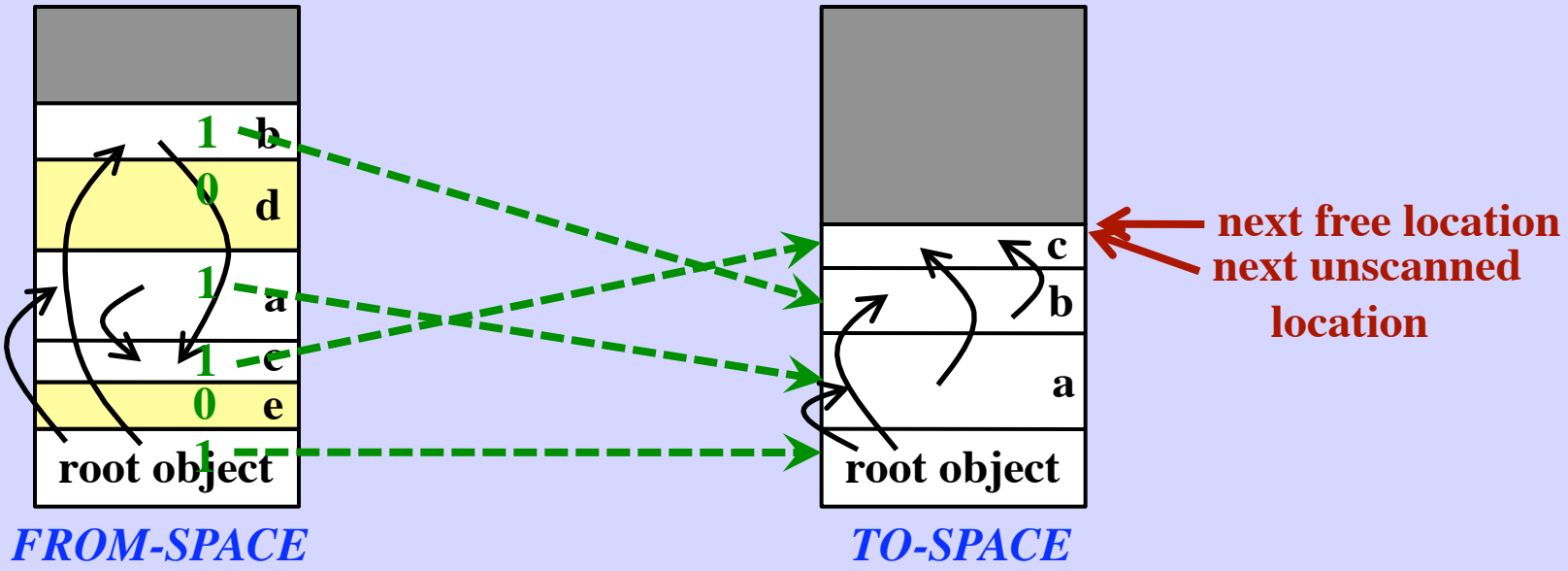




Baker's Semi-Space Algorithm

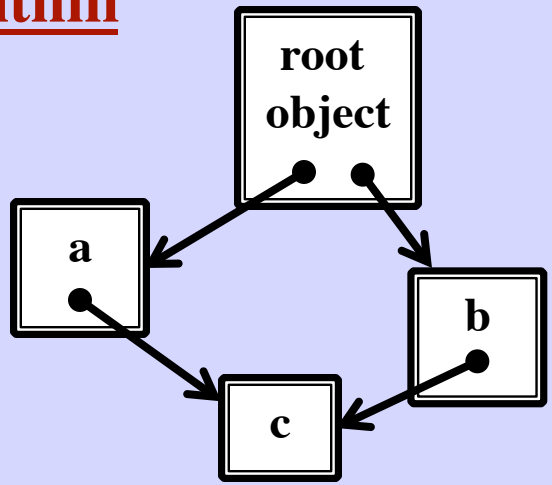
When they meet, we are done.
Continue processing.
…using the TO-SPACE for new objects.

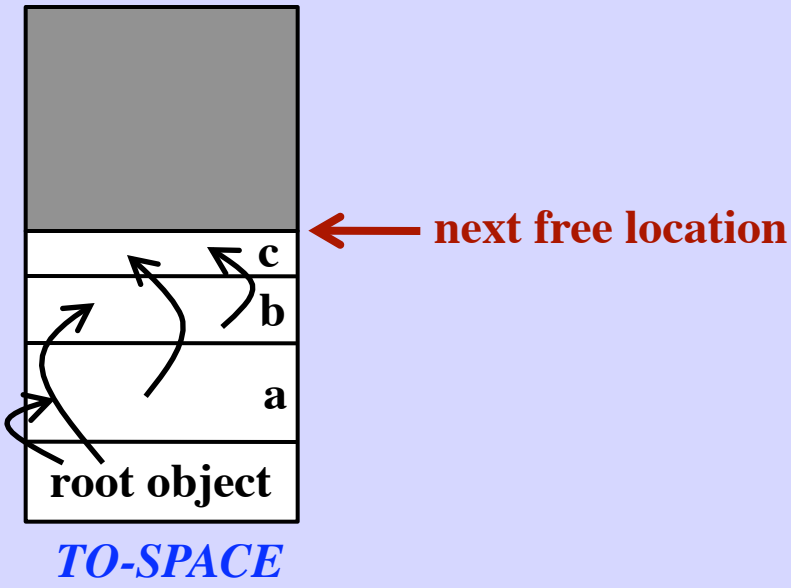




Baker's Semi-Space Algorithm

When they meet, we are done.
Continue processing.
…using the TO-SPACE for new objects.





Baker's Semi-Space Algorithm

Advantages:

No time wasted with dead objects.
Running time proportional to live objects.
Increases locality of reference in TO-SPACE.
(Objects are placed near objects that point to them)

Disadvantages:

Wastes 50% of memory
Exhibits horrible behavior when there are lots of live objects.
i.e., right before memory fills up!

Real-Time Applications:

Goal: eliminate the long copy phase!
Modification:
Every time a new object is allocated…
Do a little collecting.
Whenever a pointer is dereferenced…
Check for a forwarding pointer.

Ballard's Observations

- *Most objects are small.*
  - ≈ 0-5 fields
  - ≈ 0-20 bytes
- A few objects are very large.
  - Examples: bitmaps, also large character strings
    - ≈ 128 Kbytes
  - Do not contain OOPs (except for class ptr)
- *Large objects tend to persist* (through several collections).
- *Short-lived objects tend to be small.*
  - Example: Activation Records

The Semi-Space Algorithm wastes a lot of time on these big objects, copying them back and forth.

Idea:

Put these large objects in a separate memory region.
Collect them less often.
... using a different algorithm (e.g., Mark-Sweep)

Generation Scavenging

"Young objects die young and old objects continue to live."

– David Ungar

Generation Scavenging

"Young objects die young and old objects continue to live."

– David Ungar

Idea: Divide memory into two regions.

Generation Scavenging

"Young objects die young and old objects continue to live."

– David Ungar

Idea: Divide memory into two regions.

A large region holds...

Objects that have been around for a while

A smaller region holds...

Recently allocated objects

Generation Scavenging

"Young objects die young and old objects continue to live."

– David Ungar

Idea: Divide memory into two regions.

A large region holds...

- Objects that have been around for a while
- The "tenured" generation
- Collected less frequently

A smaller region holds...

- Recently allocated objects

Generation Scavenging

"Young objects die young and old objects continue to live."

– David Ungar

Idea: Divide memory into two regions.

A large region holds...

- Objects that have been around for a while
- The "tenured" generation
- Collected less frequently

A smaller region holds...

- Recently allocated objects
- The "new" generation
- Collected frequently
- Most of the garbage objects will be here
  - Most of the garbage will get collected

Generation Scavenging

"Young objects die young and old objects continue to live."
– David Ungar

Idea: Divide memory into two regions.

A large region holds...

Objects that have been around for a while
The "tenured" generation
Collected less frequently

A smaller region holds...

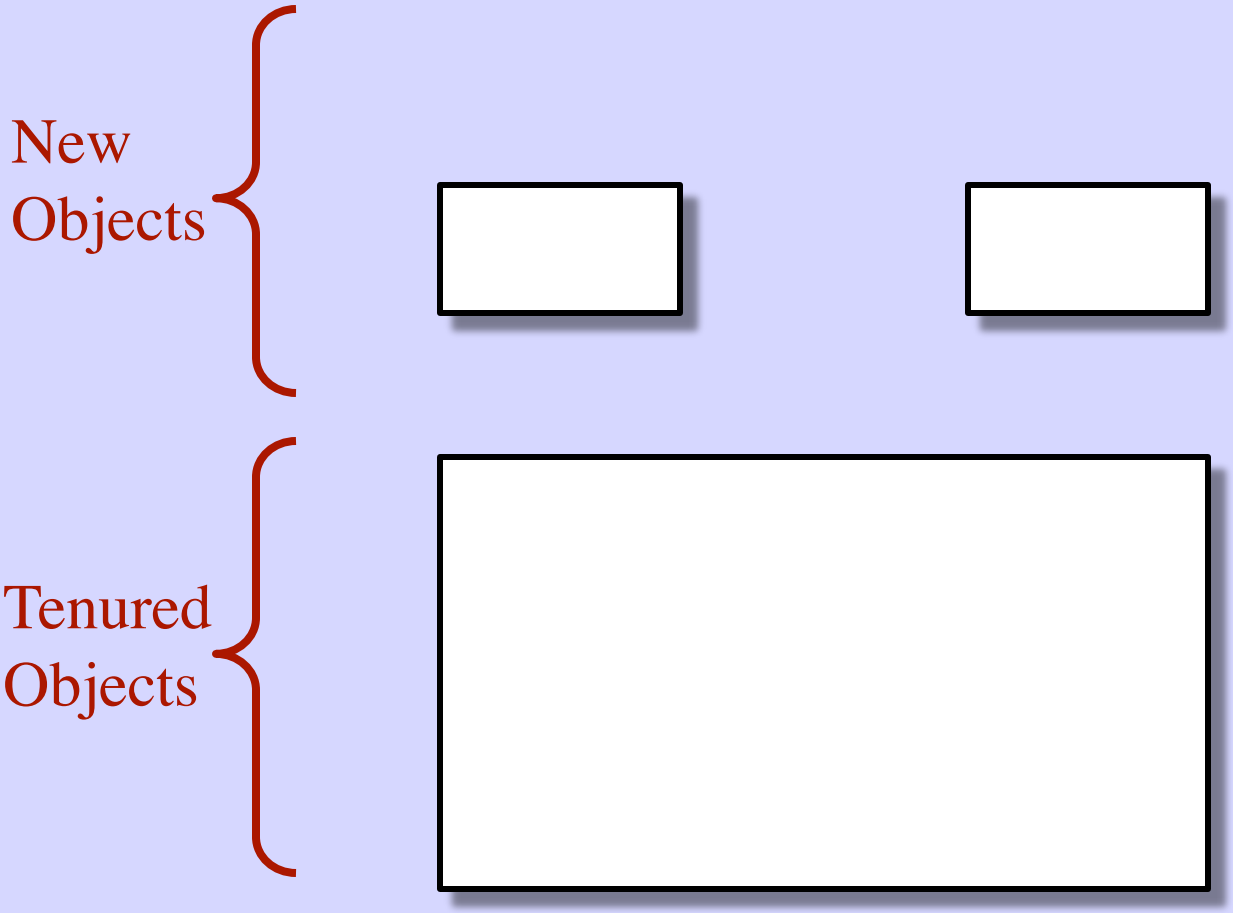
Recently allocated objects
The "new" generation
Collected frequently
Most of the garbage objects will be here
Most of the garbage will get collected

After a new object has survived several collections,
move it to the tenured region.

Generation Scavenging

The Basic Approach

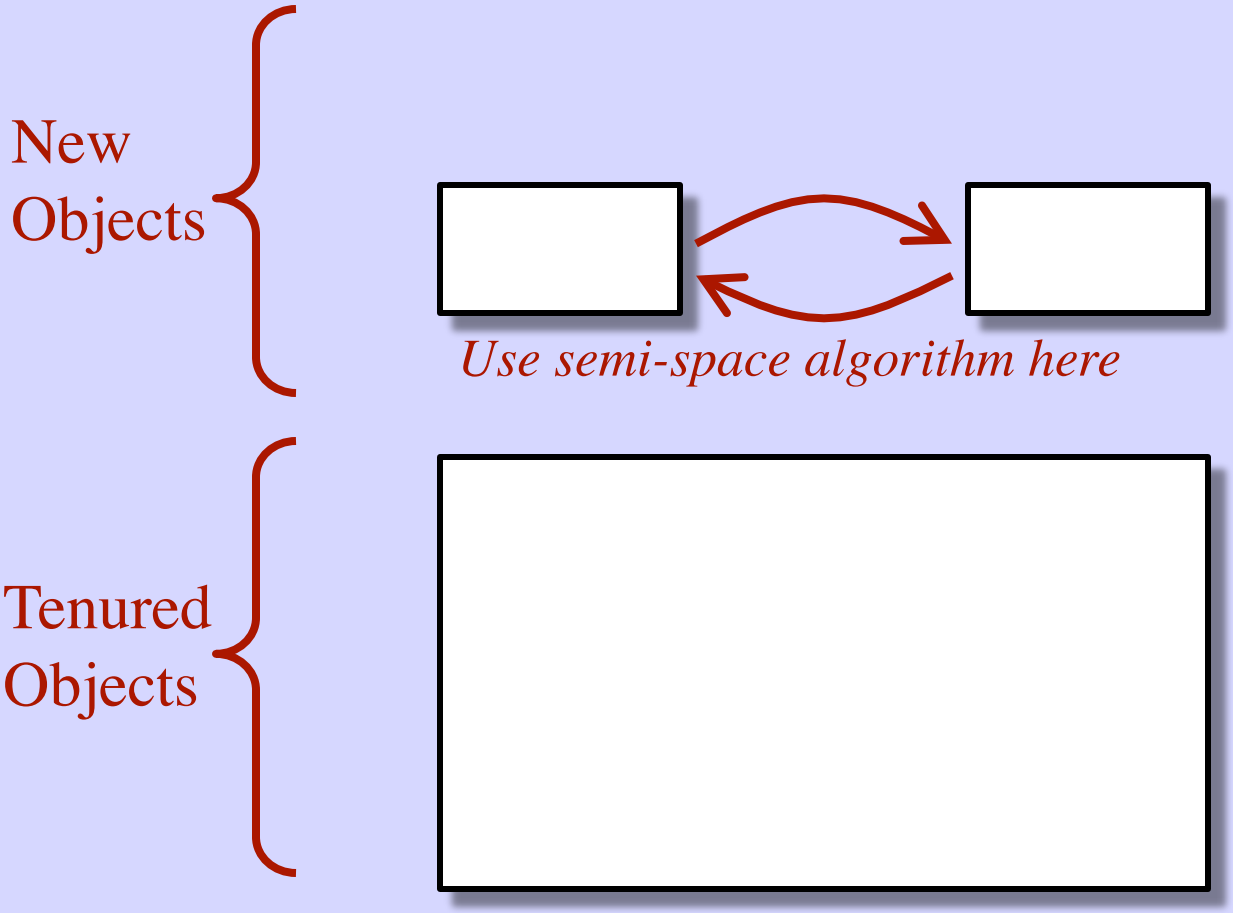
Divide memory into several regions.



Generation Scavenging

The Basic Approach

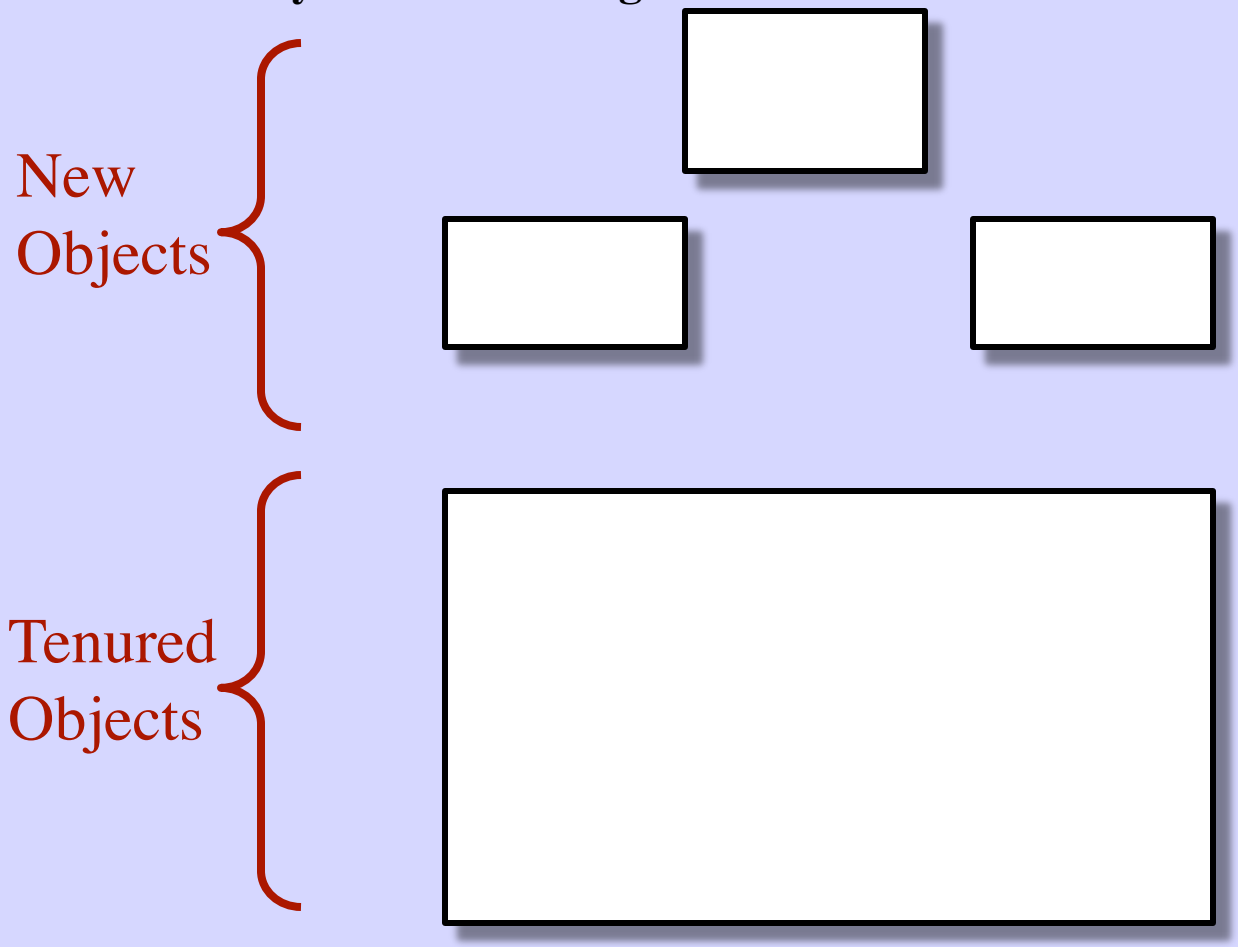
Divide memory into several regions.



Generation Scavenging

The Basic Approach

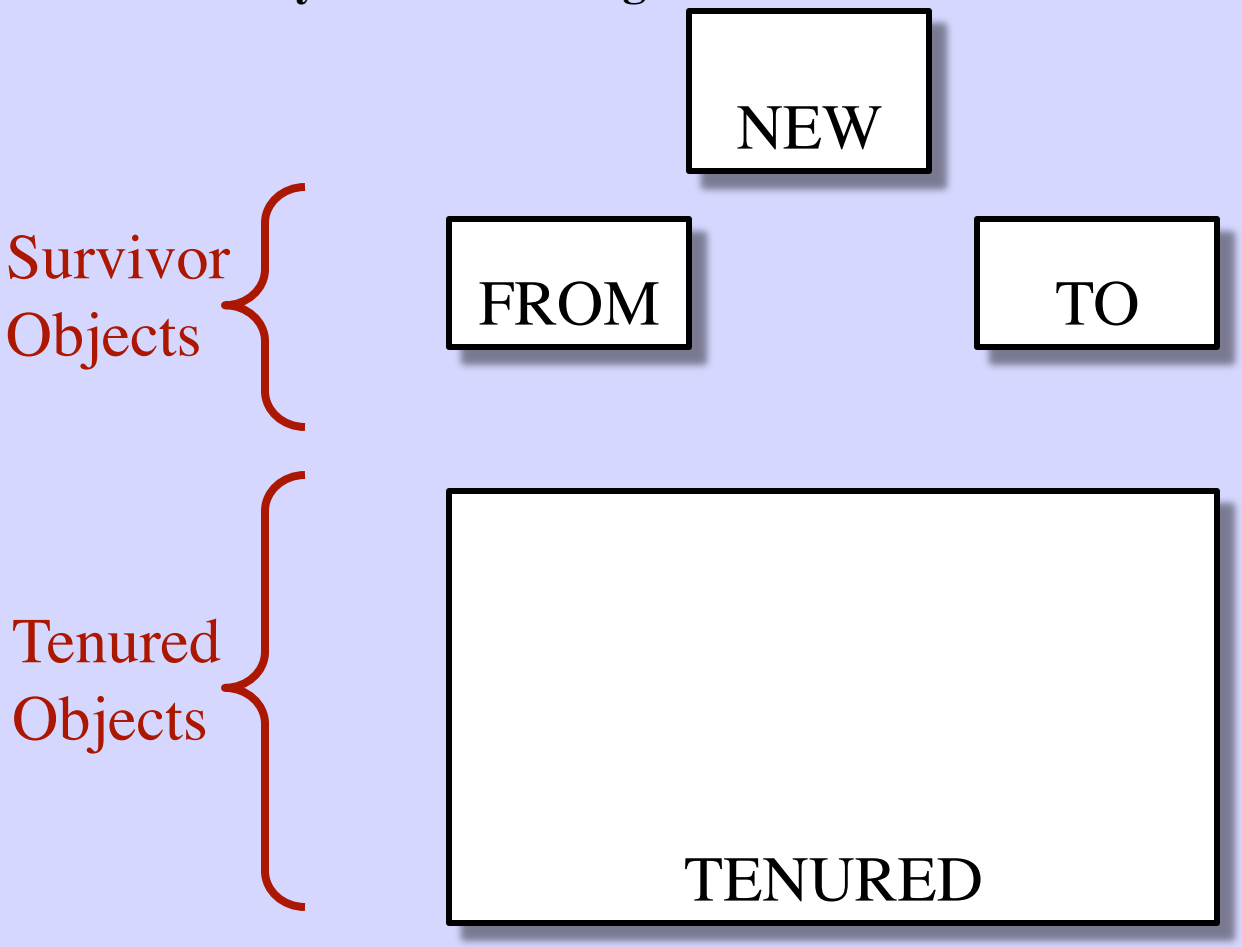
Divide memory into several regions.



Generation Scavenging

The Basic Approach

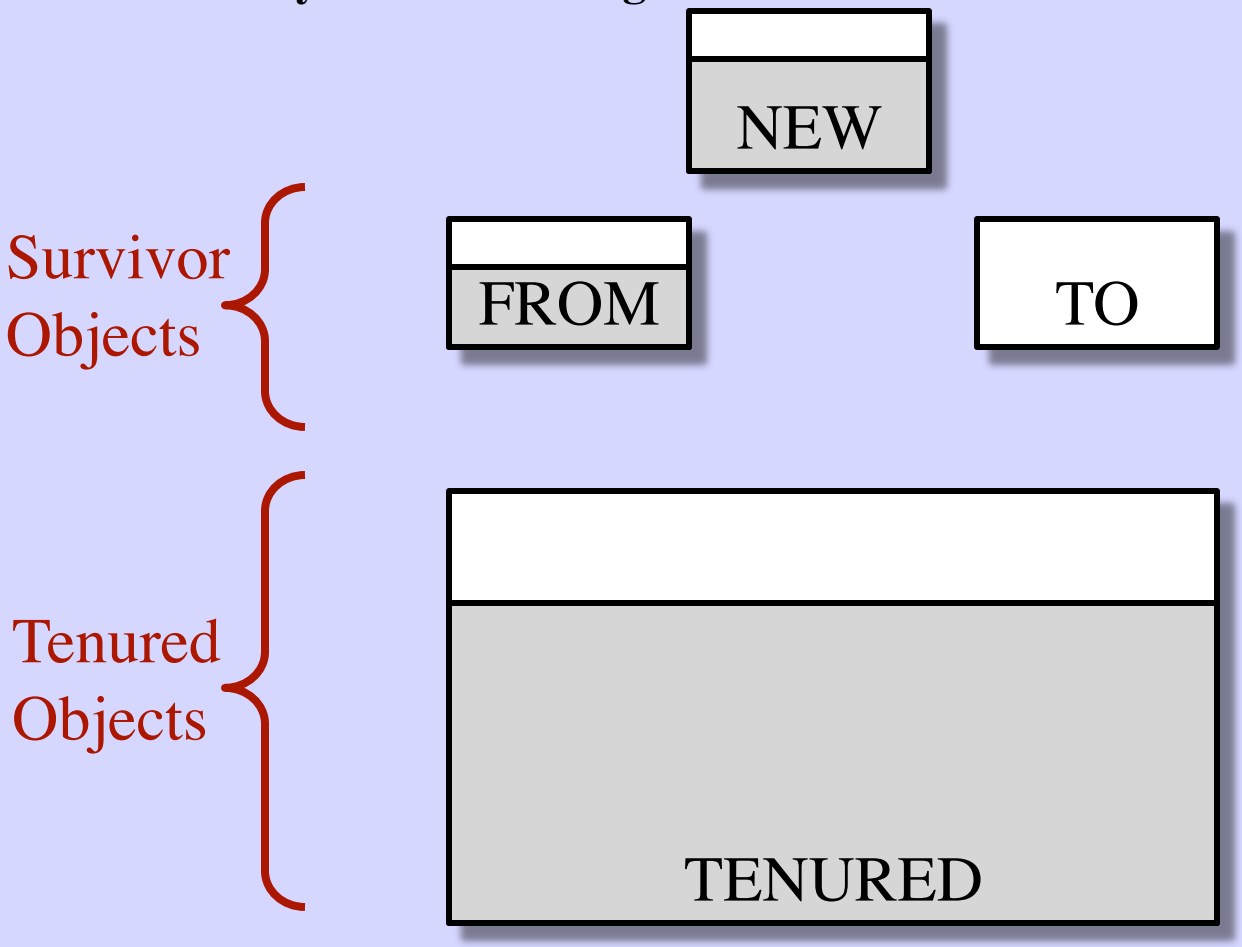
Divide memory into several regions.



Generation Scavenging

The Basic Approach

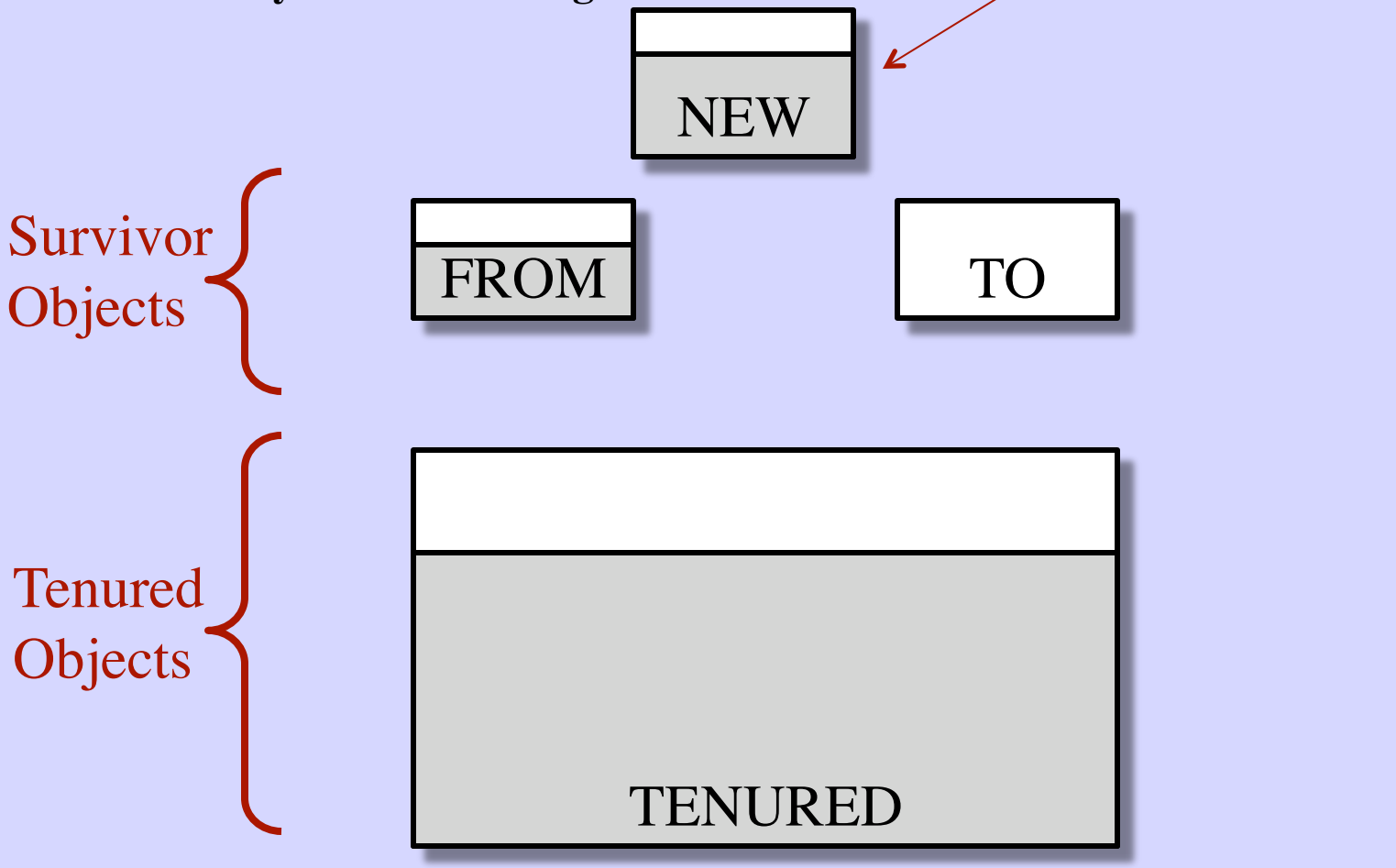
Divide memory into several regions.



Generation Scavenging

The Basic Approach

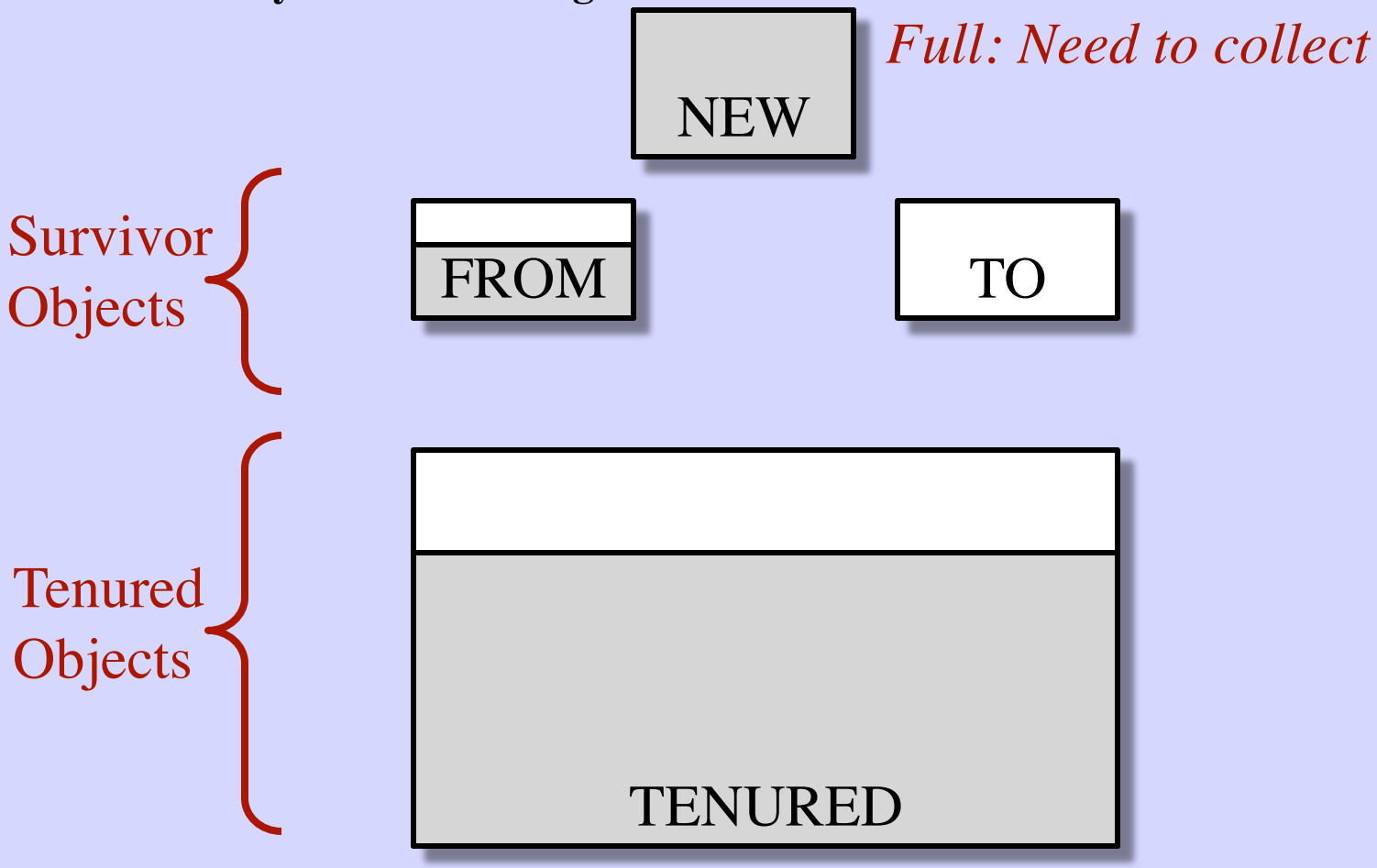
Divide memory into several regions.



Generation Scavenging

The Basic Approach

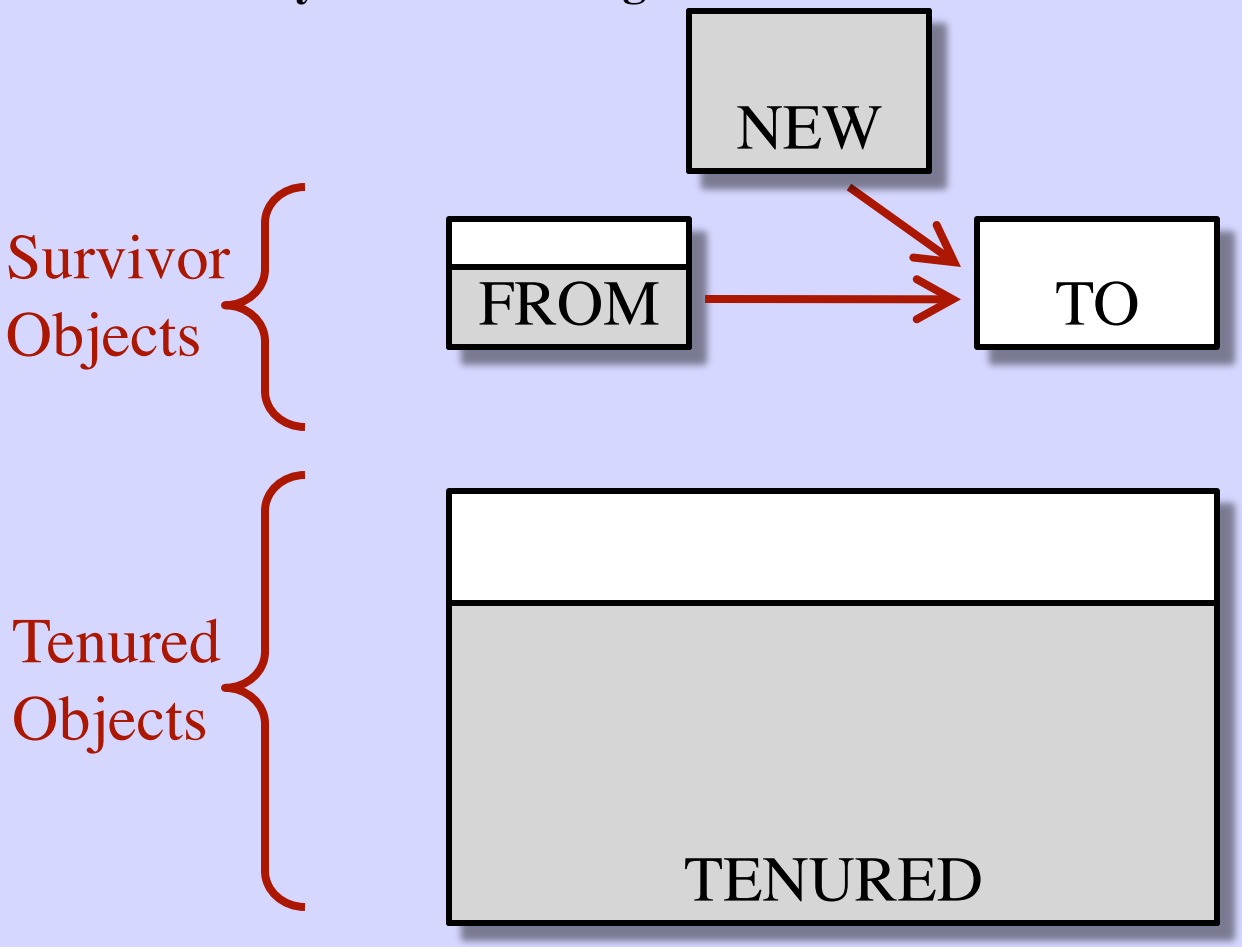
Divide memory into several regions.



Generation Scavenging

The Basic Approach

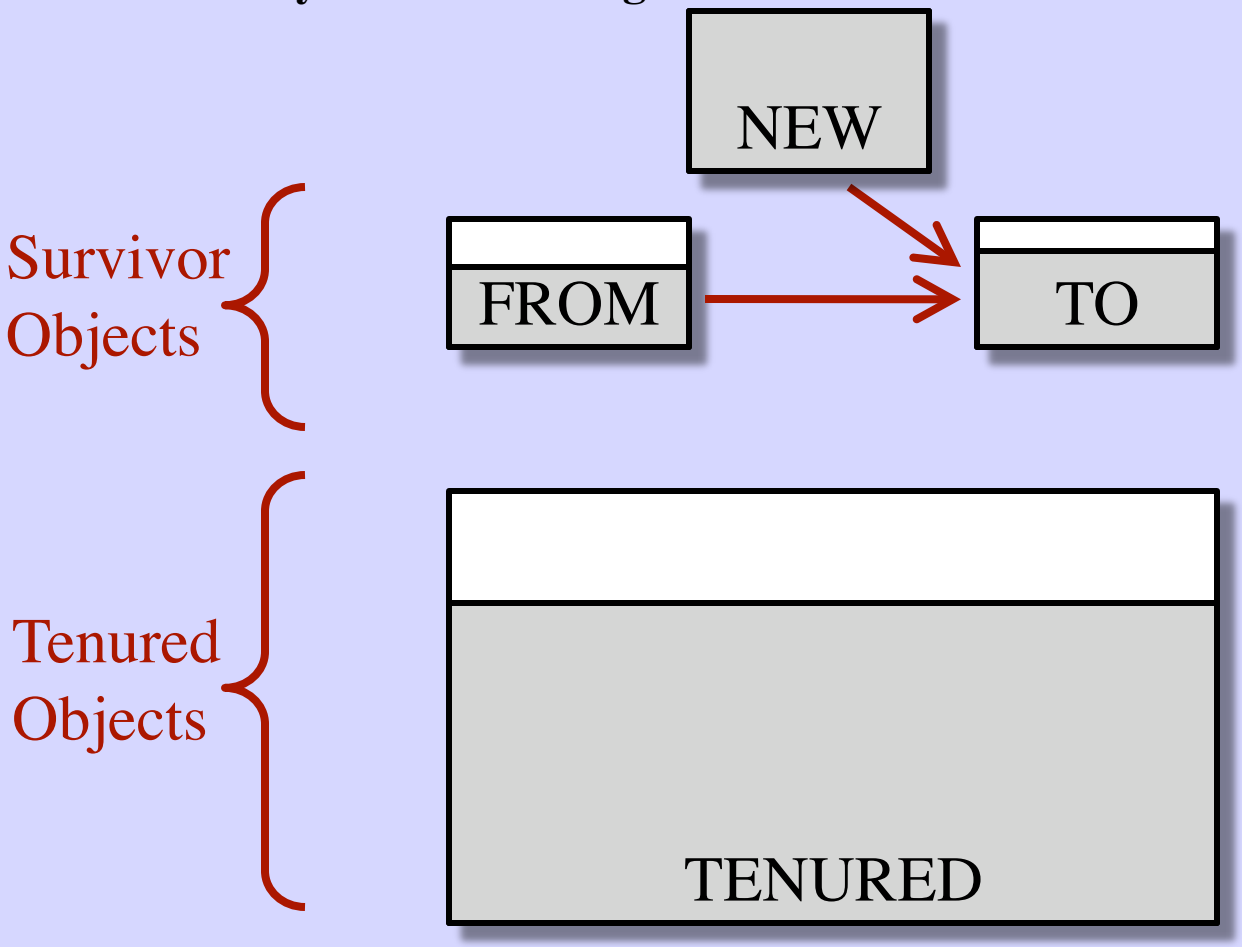
Divide memory into several regions.



Generation Scavenging

The Basic Approach

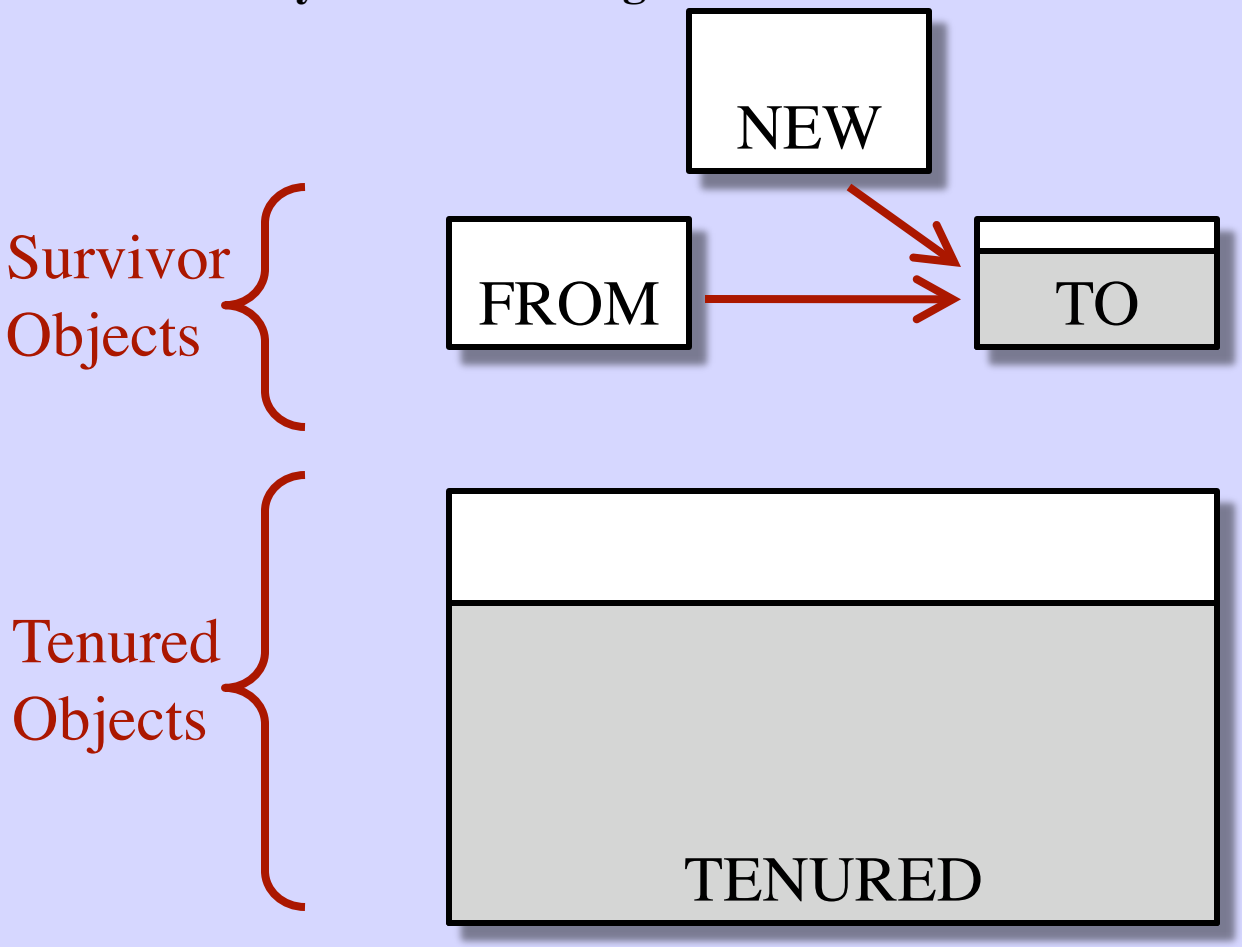
Divide memory into several regions.



Generation Scavenging

The Basic Approach

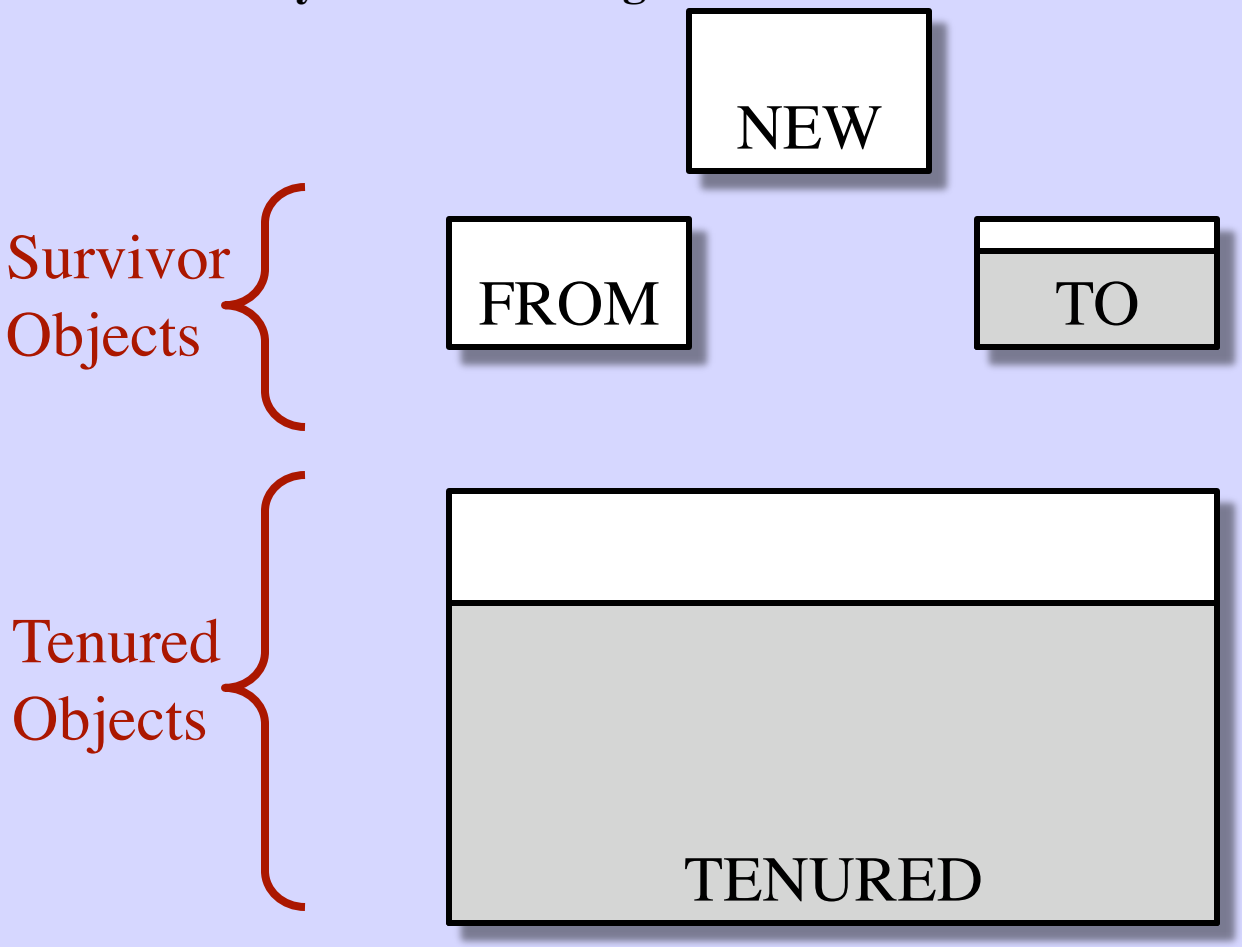
Divide memory into several regions.



Generation Scavenging

The Basic Approach

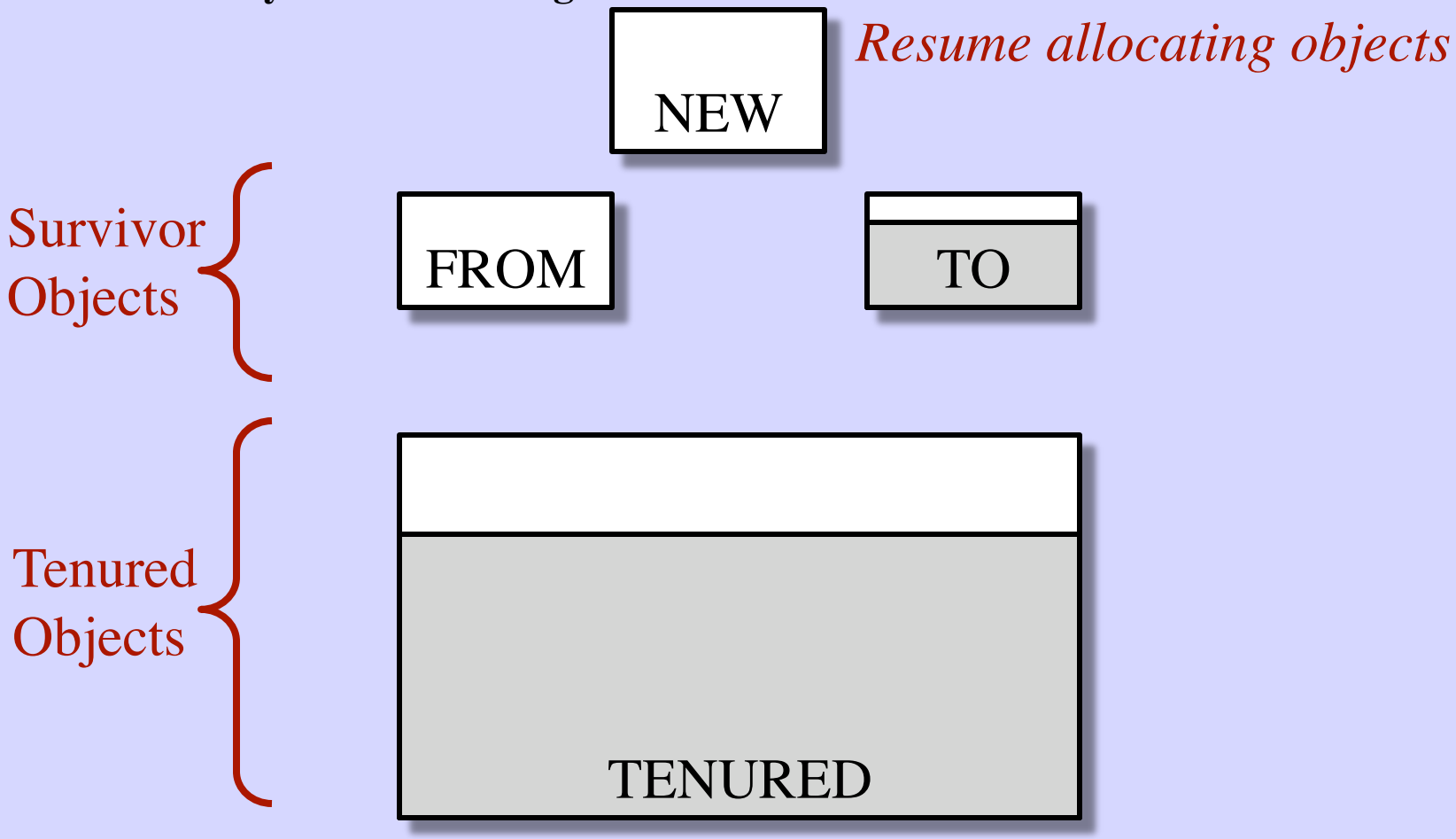
Divide memory into several regions.



Generation Scavenging

The Basic Approach

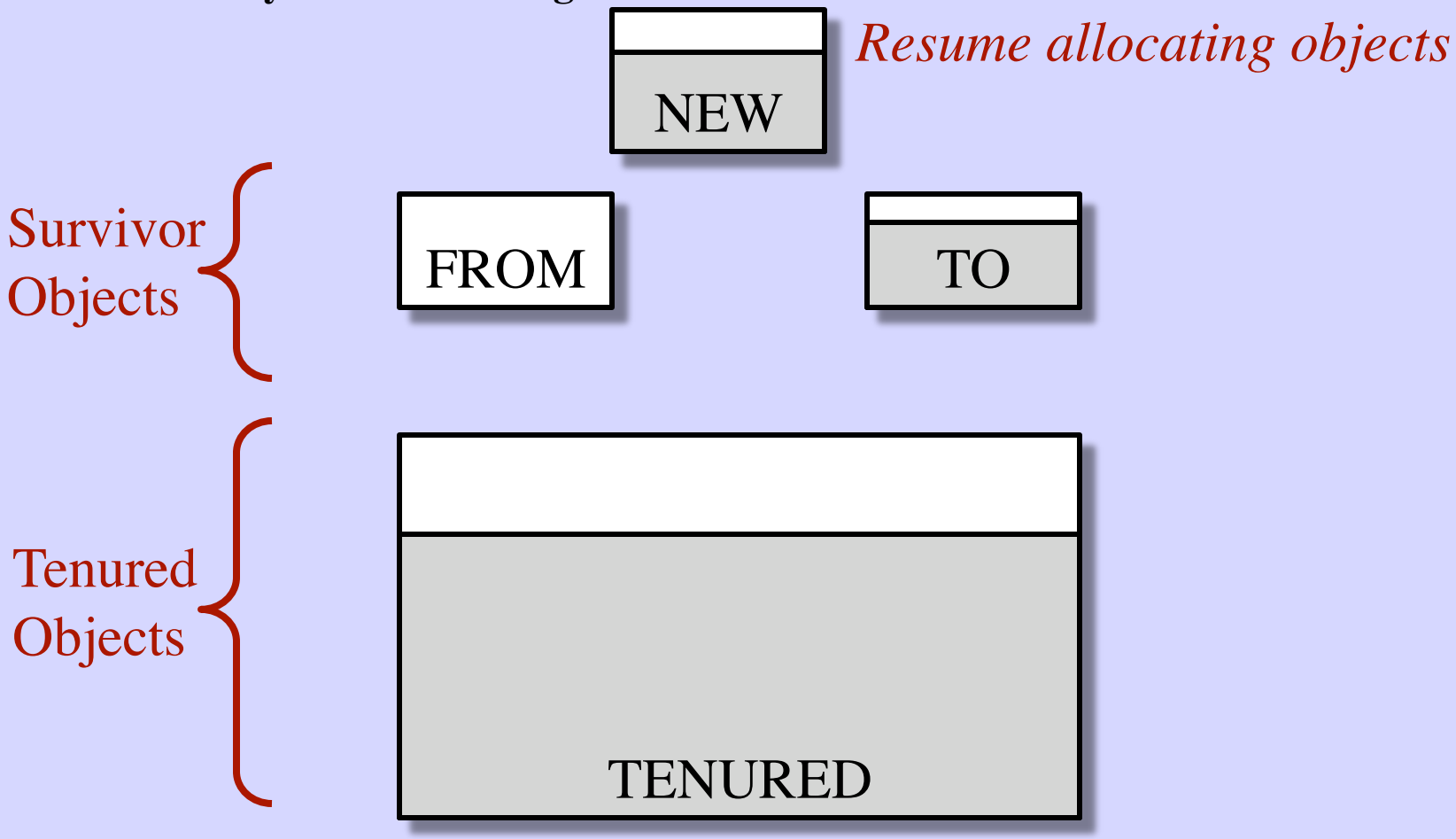
Divide memory into several regions.



Generation Scavenging

The Basic Approach

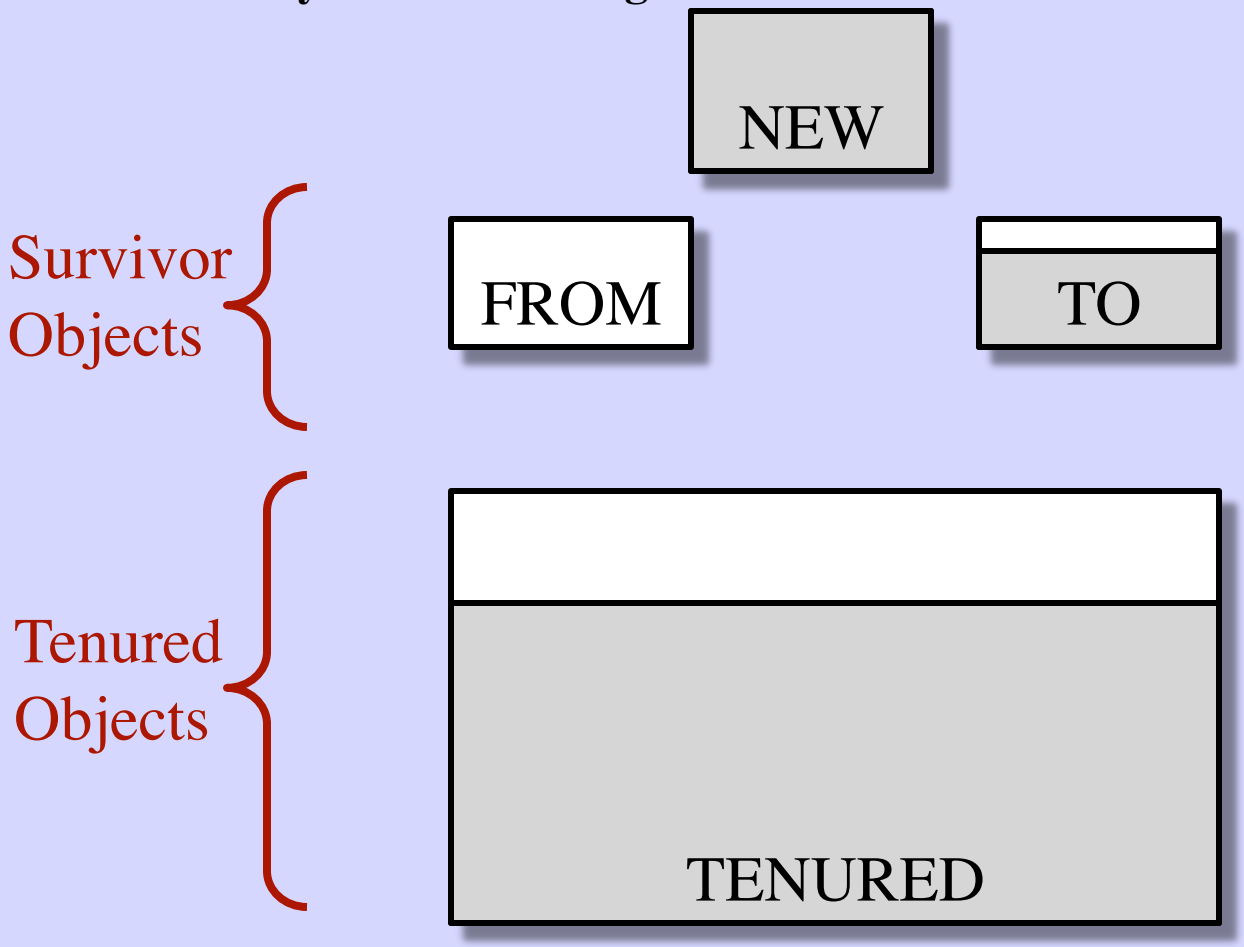
Divide memory into several regions.



Generation Scavenging

The Basic Approach

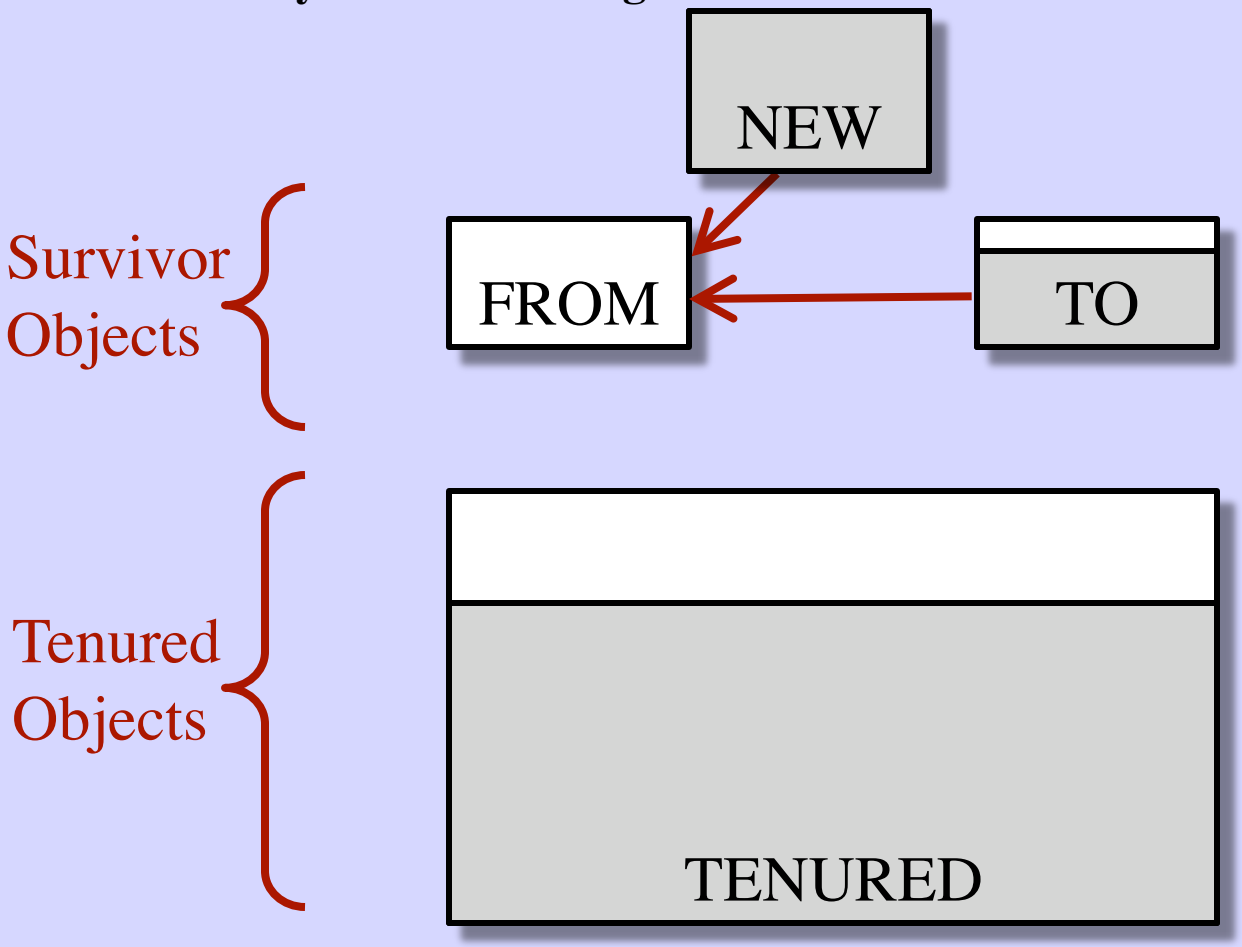
Divide memory into several regions.

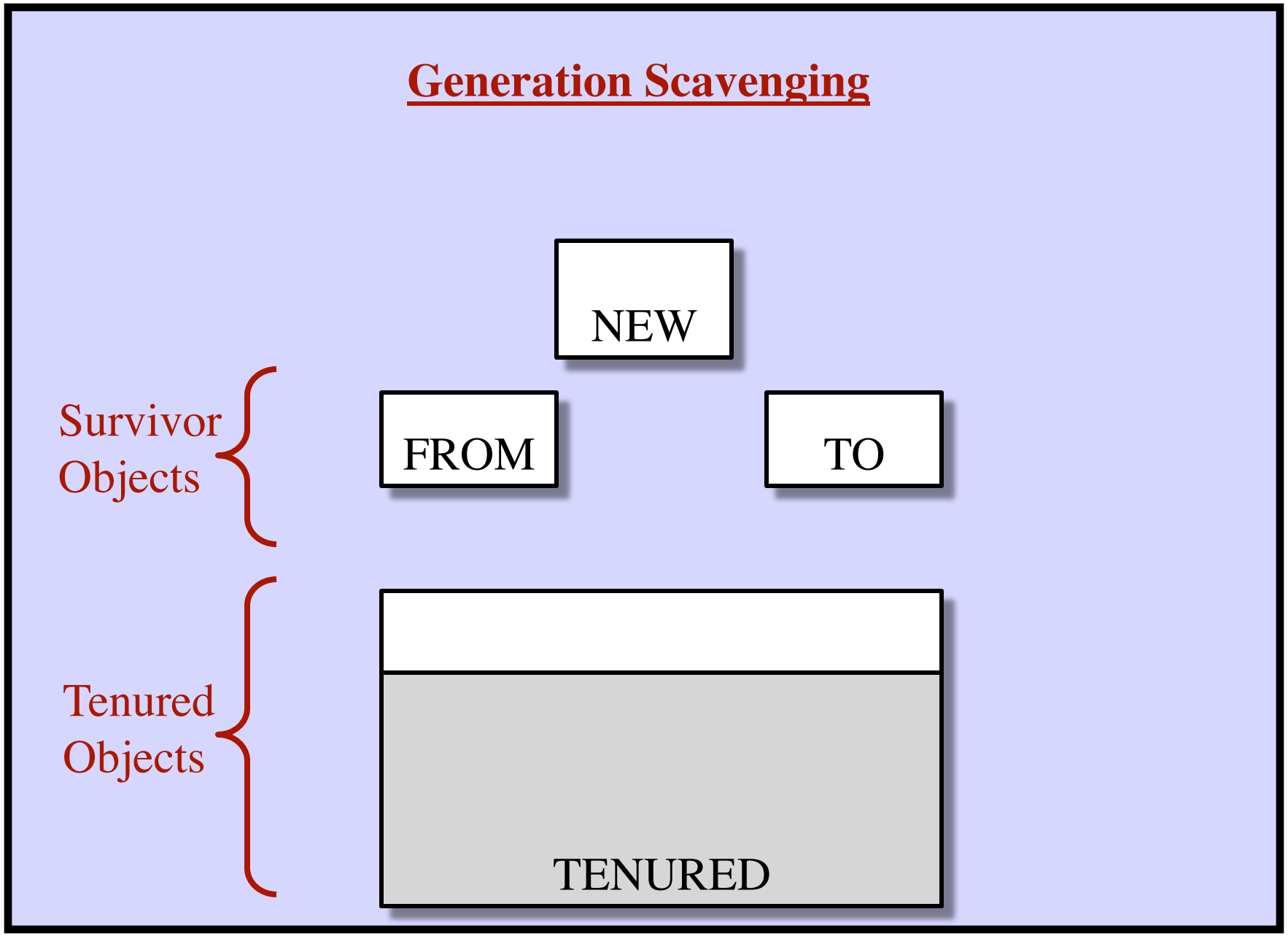


Generation Scavenging

The Basic Approach

Divide memory into several regions.



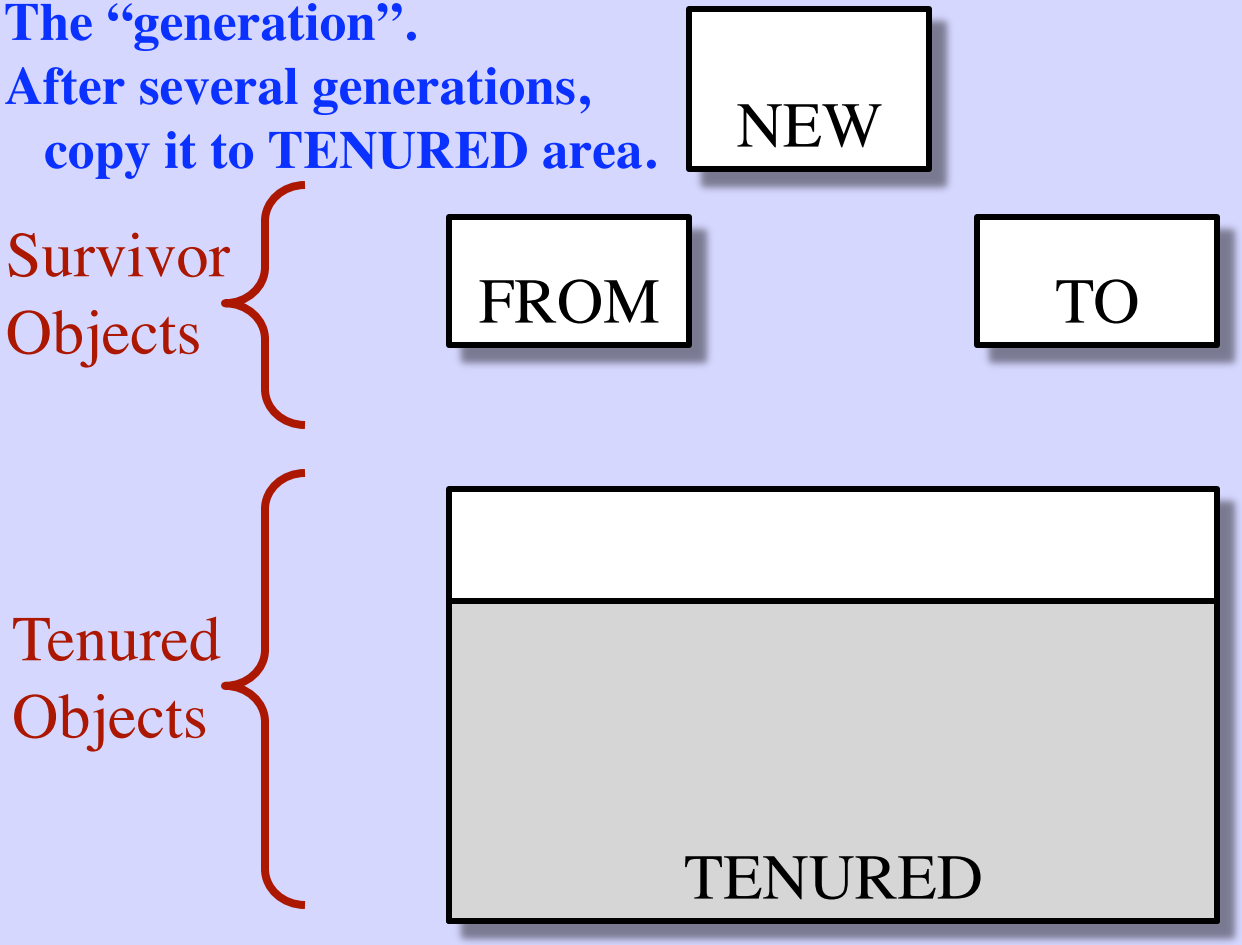

Generation Scavenging
NEW
Survivor Objects
FROM
TO
Tenured Objects
TENURED

Generation Scavenging

For each object, keep a count of how many times it has been copied.

The "generation".

After several generations, copy it to TENURED area.

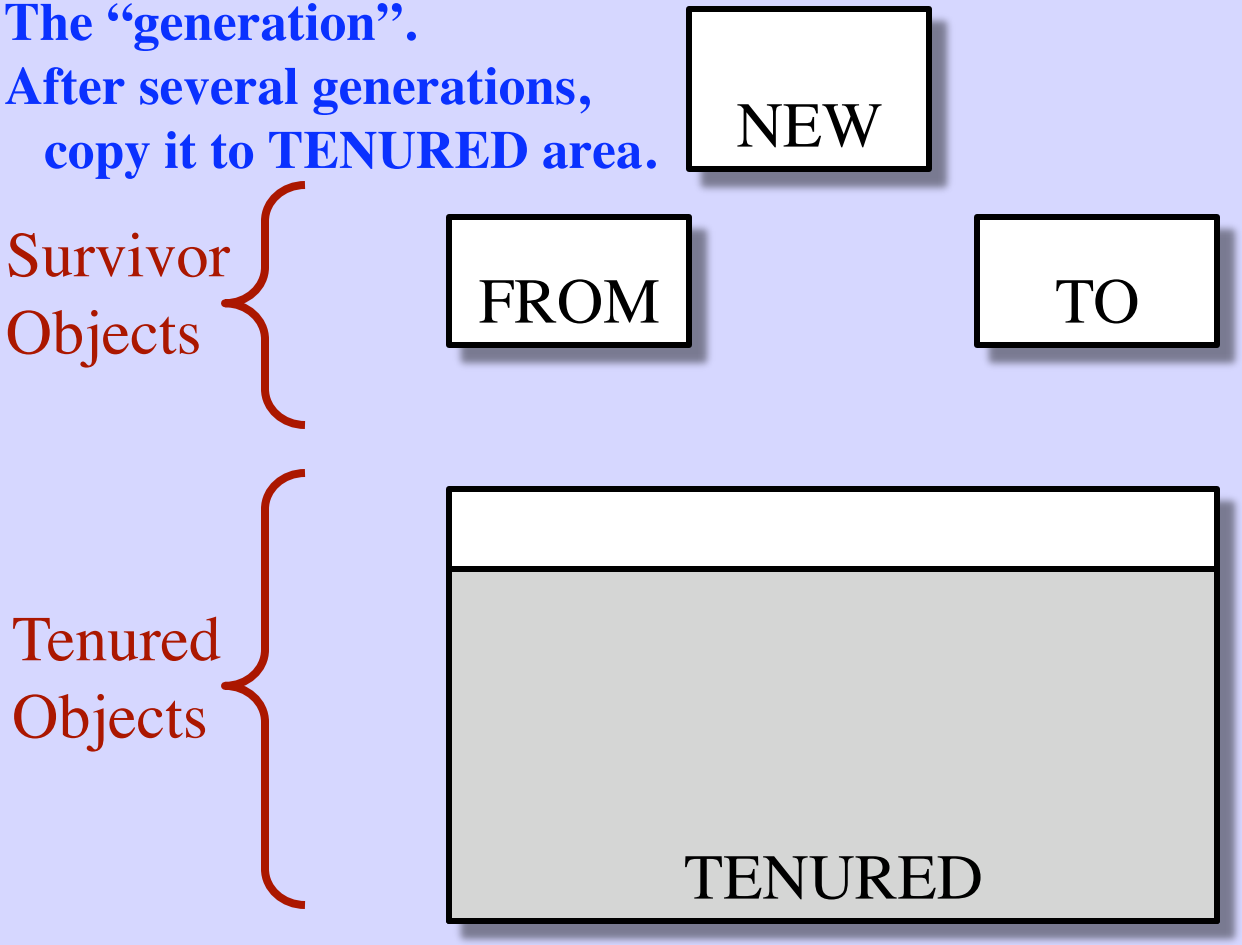


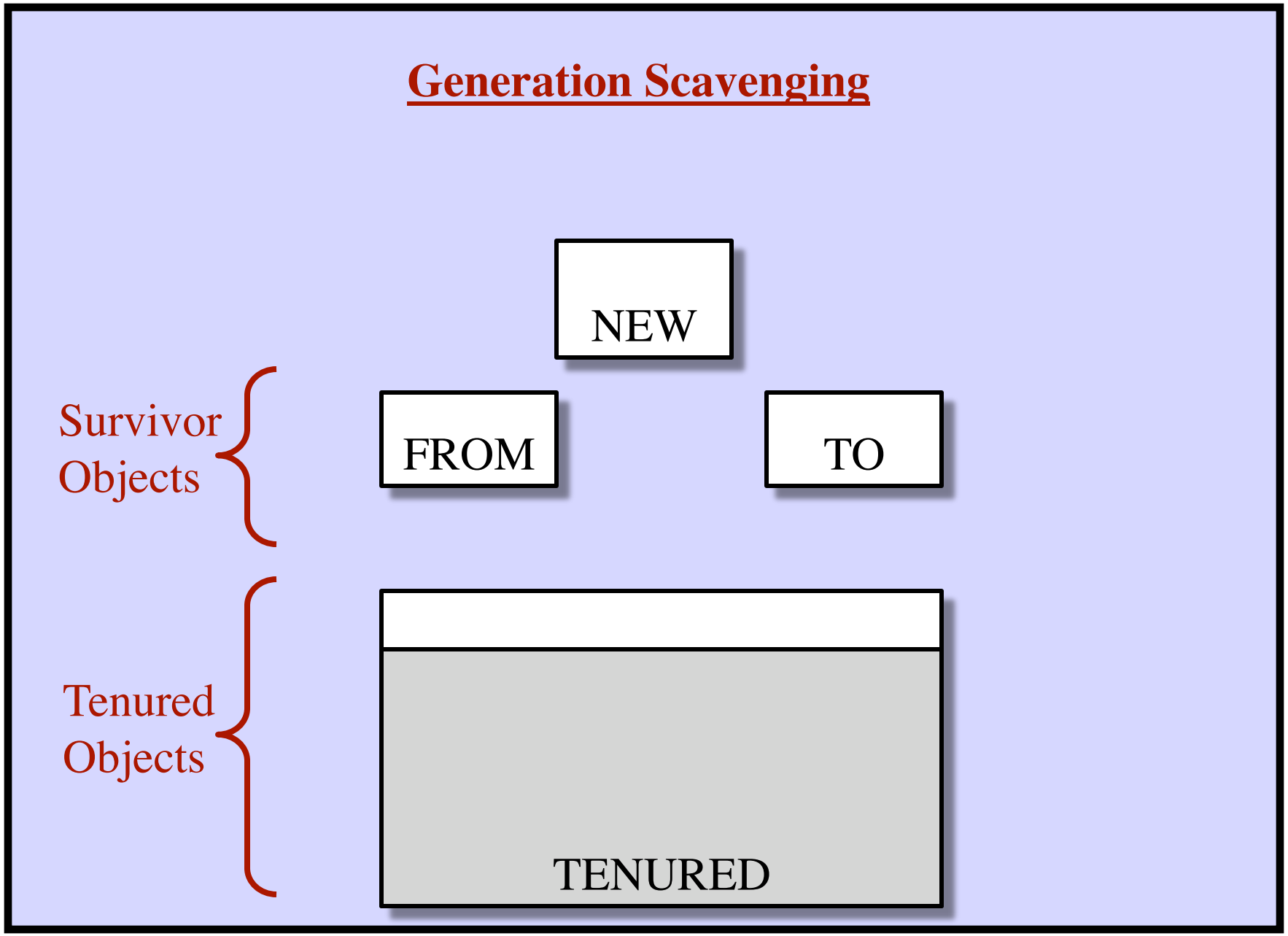
Generation Scavenging

For each object, keep a count of how many times it has been copied.

The "generation".

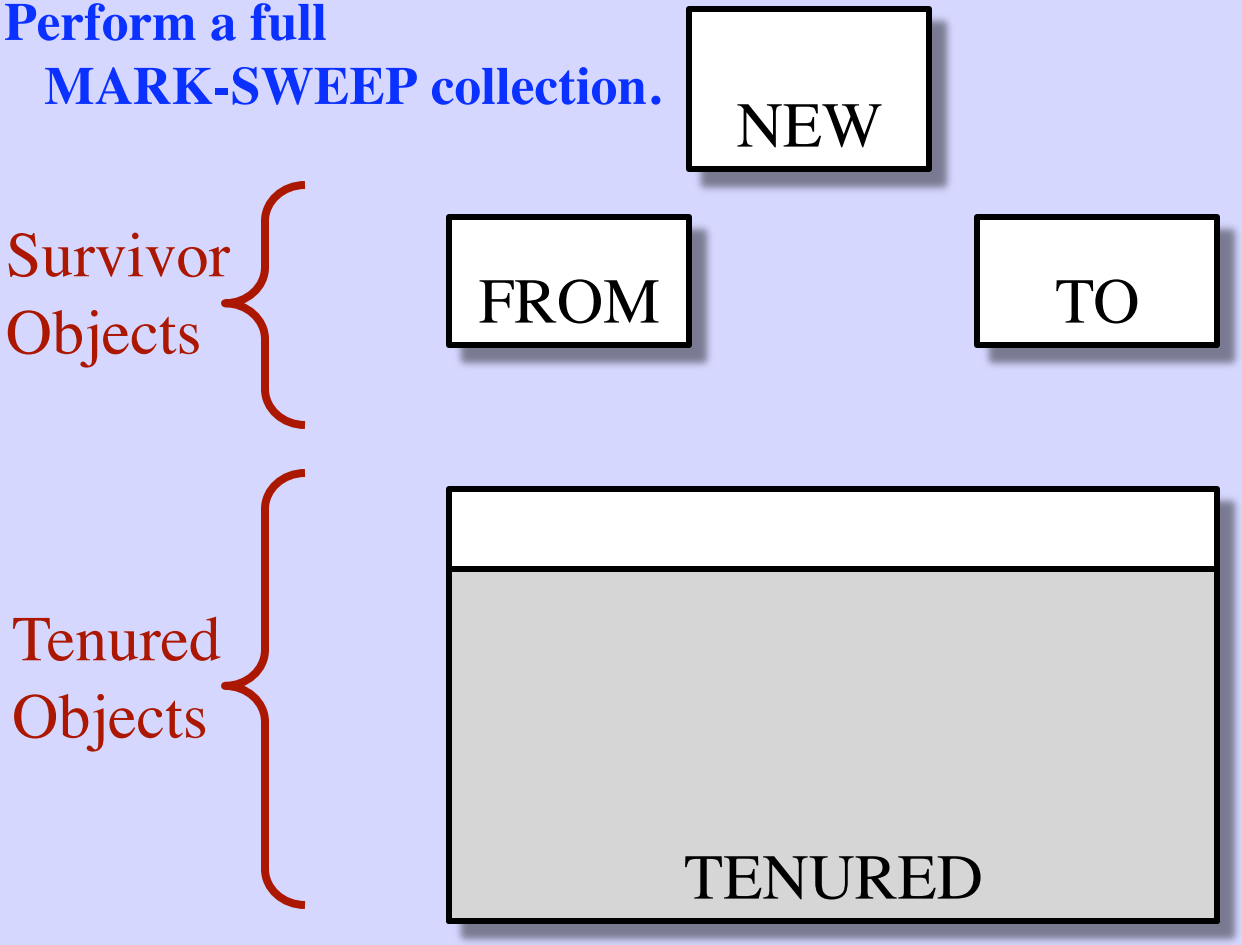
After several generations, copy it to TENURED area.



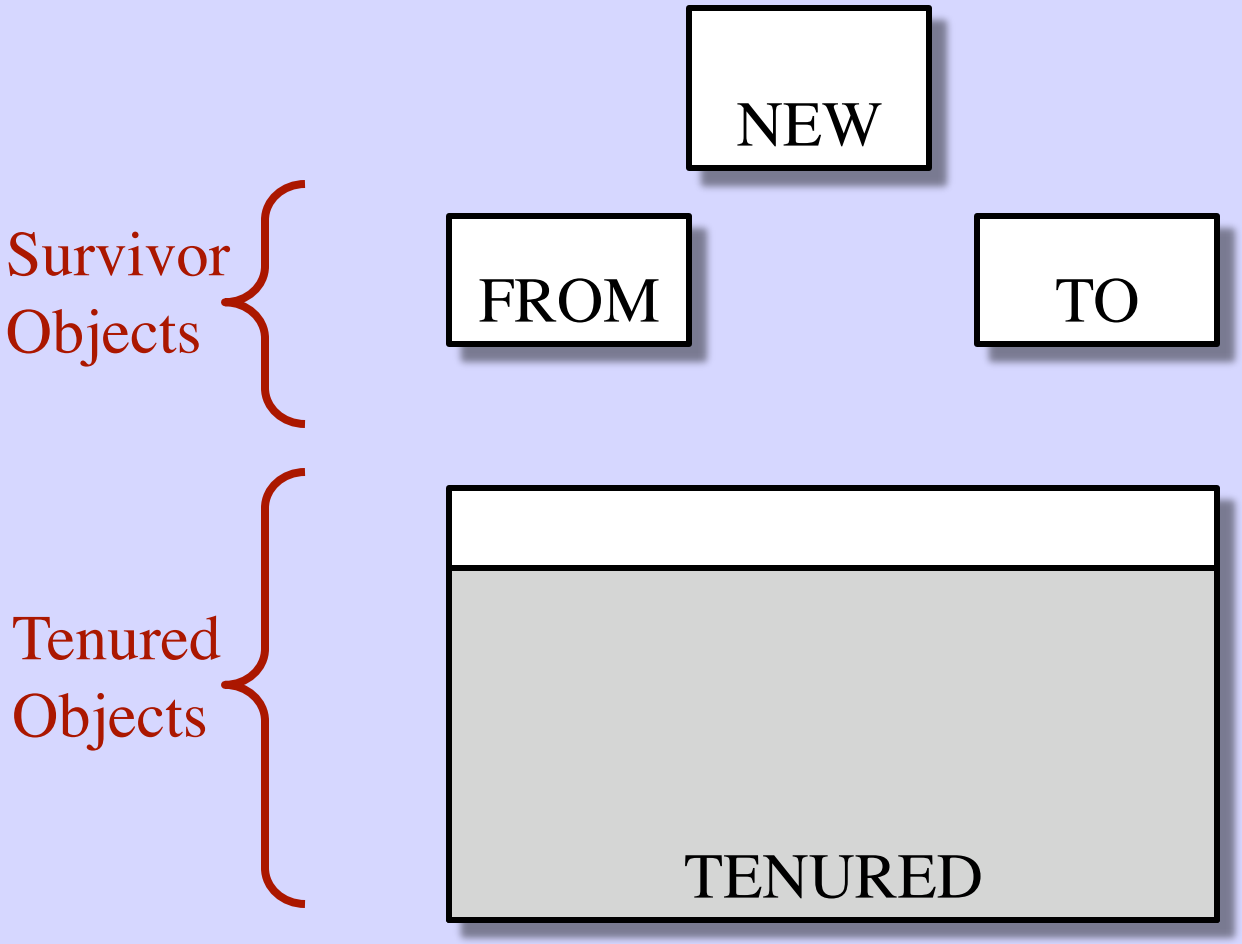

Generation Scavenging
NEW
FROM
TO
Survivor Objects
Tenured Objects
TENURED

Generation Scavenging

Once tenured, the object will be ignored.
When the TENURED area fills up…
Perform a full
MARK-SWEEP collection.



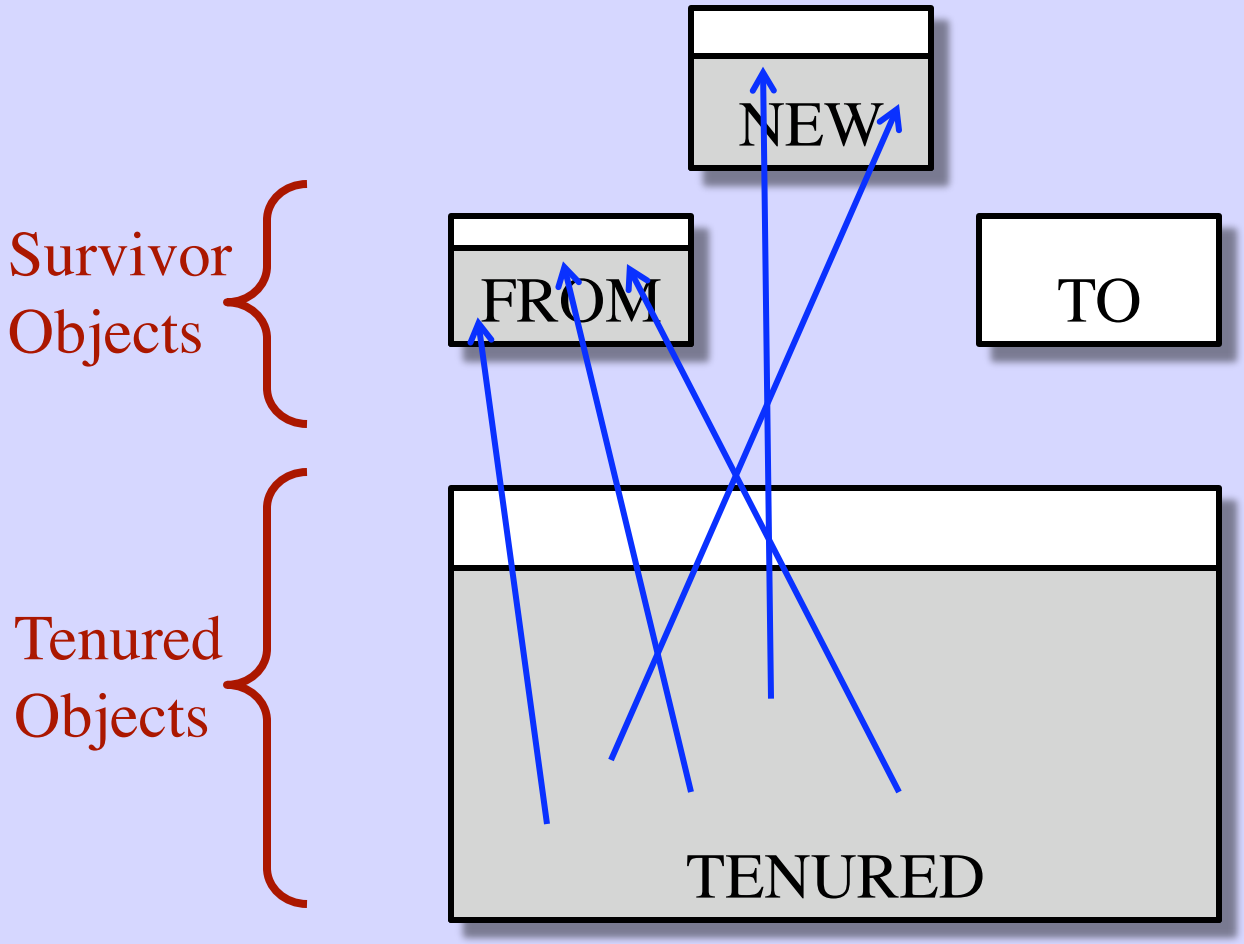
Generation Scavenging



Generation Scavenging

Complication:

Tenured objects may point to newer objects.



Generation Scavenging: Policy Issues

How big to make each space?

An object is moved into the TENURED area after it survives K collections.

What value for K?

The system cannot run during GC.
GC will cause a short pause.
(e.g., 1 msec)

Is it better to collect more frequently than necessary?
The collections will be faster.
The pauses will be shorter.

When to schedule GC?

The "become:" Operation

Exchange the identities of 2 objects

Example: A collection needs to grow itself.

Example: Adding an instance variable to a class.

Must go through all existing instances and "grow" them.

The "become:" Operation

Exchange the identities of 2 objects

Example: A collection needs to grow itself.

Example: Adding an instance variable to a class.

Must go through all existing instances and "grow" them.

Implementation:

Easy with an "object table"

With direct pointers:

Need to scan all objects and change all pointers!

The "become:" Operation

Exchange the identities of 2 objects

Example: A collection needs to grow itself.

Example: Adding an instance variable to a class.

Must go through all existing instances and "grow" them.

Implementation:

Easy with an "object table"

With direct pointers:

Need to scan all objects and change all pointers!

Solution:

- Re-write many classes to avoid using "become:"

  Make indirection explicit.
- The primitive is available to walk through memory.

  Check (and possibly update) every pointer in memory.
- To save time, the primitive can do several at once

```
(A B C) elementsForwardIdentityTo: (X Y Z)
```

Squeak Object Format

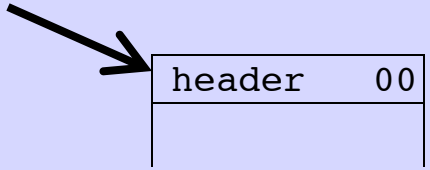
What goes into an object's header?

- Size in bytes (up to 24 bits, max object size = 16 Mbytes)
- Class of object (32 bit pointer)
- Hash code (12 bits)
- Format of object (4 bits)
  - contains pointer/raw bits
  - contains indexable fields or not
  - data is byte / word addressable
  - object is a CompiledMethod
- Bits used by garbage collector

Object Format

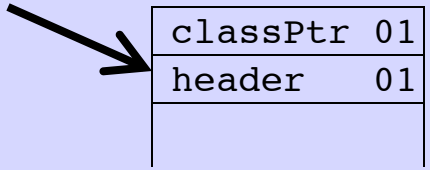
Idea: *Encode more common values in fewer bits.*

Option 1:
- size = 0 .. 64 words (6 bits)
- class = 0 .. 32 (5 bits)



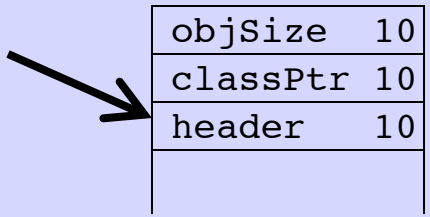
82%

Option 2:
- size = 0 .. 64 words (6 bits)
- any class



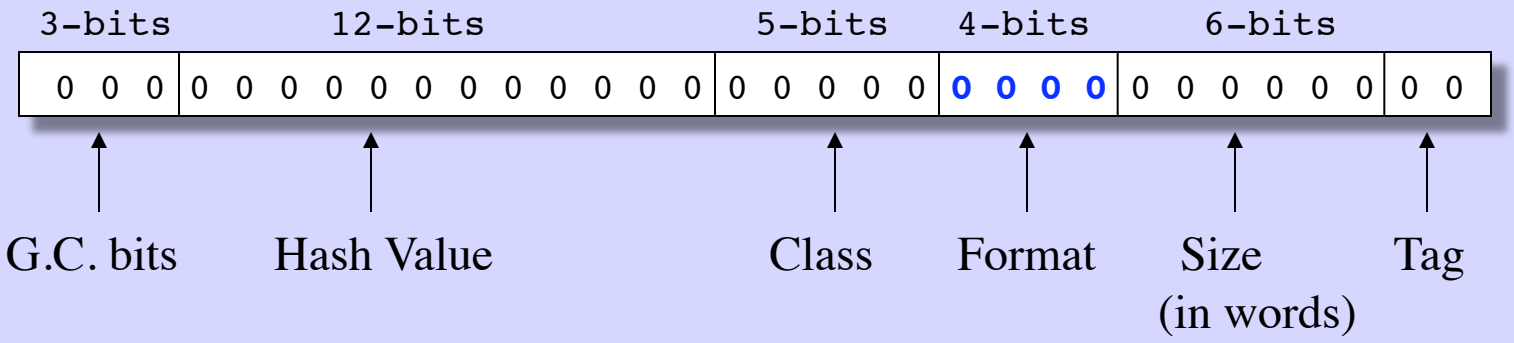
17%

Option 3:
- Most general format



1%

Header Word



Format: This object contains…

- **0000** - no fields at all
- **0001** - fixed pointer fields only (a normal object)
- **0010** - indexed pointer fields
- **0011** - both fixed fields and indexed pointer fields
- **0100** - (unused)
- **0101** - (unused)
- **0110** - indexed word data, but no pointer fields
- **0111** - (unused)
- **10xx** - indexed byte fields, but no pointer fields (xx = rest of size in bytes)
- **11xx** - a compiled method (xx = rest of size in bytes)

Generation Scavenging:

Additional detail.
Ignore these slides.

Generation Scavenging: Concepts

New Objects

Allocated recently; likely to become garbage soon
Must collect them quickly

Survivor Objects

These objects have survived a few collections
There is a probability they may live for a very long time

Tenured Objects

The oldest objects.
They have been around so long we assume they will never die.
(Considered to be "permanent")
Don't bother trying to collect them at all.
GS will occasionally give objects "tenure"
Some tenured objects may become unused / unreachable.
GS will not identify them as garbage.
Must collect tenured objects offline
Use Mark-Sweep occasionally
… when generation scavenging finally fails

Generation Scavenging: Memory Regions

- Tenured Area --
  Contains the permanent objects
  These objects act as the "roots" of reachability

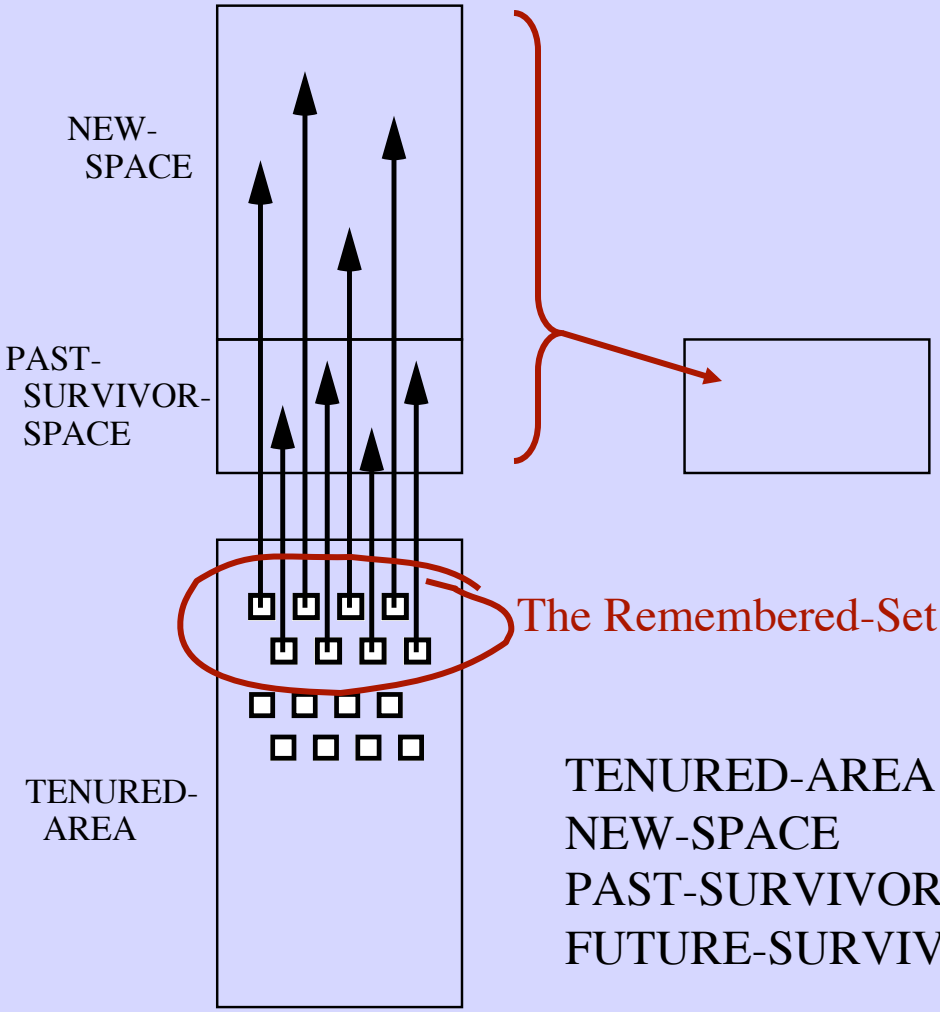
  The "Remembered Set":
  Tenured objects which point to non-tenured objects

- New Space
  Allocate new objects here
  If objects survive the first collection,
  move them into Past Survivor Space

- Past Survivor Space
  These objects have survived several collections
  After K collections, move them into Tenured Area

- Future Survivor Space
  Used only during GS collection

Generation Scavenging: Memory Regions



The Remembered-Set

TENURED-AREA	980 Kbytes
NEW-SPACE	140 Kbytes
PAST-SURVIVOR-SPACE	28 Kbytes
FUTURE-SURVIVOR-SPACE	28 Kbytes

Generation Scavenging: Algorithm

When NEW-SPACE fills up, stop and collect.

The "root" objects in NEW-SPACE, PAST-SURVIVOR-SPACE?
- Every object pointed to by…
  - Objects in the Remembered-Set
  - The interpreter registers, activation-record stack, etc.

Copy all root objects into FUTURE-SURVIVOR-SPACE.

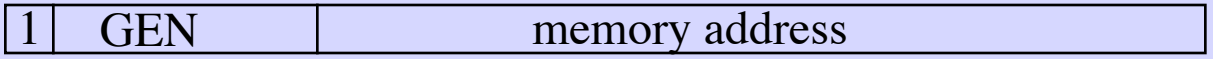
Pull all reachable objects over (as in Baker's Algorithm):
- Scan all pointers in the FUTURE-SURVIVOR-SPACE.
  - For every referenced object
    - (in NEW-SPACE or PAST-SURVIVOR-SPACE)
      - Copy into FUTURE-SURVIVOR-SPACE

Switch the PAST- and FUTURE-SURVIVOR-SPACES.
- Resume Processing.

Generation Scavenging: Algorithm

Do not need to copy FUTURE-SURVIVOR-SPACE
back to PAST-SURVIVOR-SPACE.
(We can update the Remembered-Set as we scan it for root objects.)

Must keep the Remembered-Set up to date.
Every time we store a pointer in the TENURED-AREA
We may need to update the Remembered-Set
Each pointer looks like this



When we overwrite a pointer with a different GEN, update Remembered-Set.

Tenuring Policy

Problem: When to promote an object into the TENURED-AREA?

Associate an "age" with each untenured object.

Increment it whenever the object is copied during G.C.

After the object survives K collections,

Move it into the TENURED-AREA

Problem: Premature Tenuring:

An object is promoted and then dies relatively soon!

Solution:

- Generalize to multiple generations
- Keep track of how old each object is
- At certain ages (2 seconds, 10 seconds, 1 minute, 1 day, …)

  Promote objects to the next older generation
- Scavenge younger generations more frequently.

Squeak: Garbage Collection

Uses both:

- Generation Scavenging for most collections (0.5 msec)
- Mark-Sweep, when Gen Scavenging fails (75 msec)

Mark-Sweep Algorithm

Will perform compaction in place.

To compact all objects: Must redirect all pointers.

Need space for forwarding pointers

But no object table!

Solution:

"Relocation Entries"

Contains info about where an object is being moved to

Pre-allocate an array of 1000 relocation entries.

Can always move at least 1000 objects.

Put at top of heap; if more space available, use it too for additional entries.

Make multiple passes if not enough room for relocation entries (rare).

Squeak: Garbage Collection

Generation Scavenging

G.S. looks at only NEW and SURVIVING objects
… Not the TENURED (old) objects
Copies them into NEW-SURVIVOR space
(Compacting these objects immediately)
Not too many of them --> can be done quickly.

When to perform G.S.?
When memory fills up --> bigger delay
Do it more often!
Keep a counter. Increment whenever an object is allocated.
When counter reaches threshhold, then do G.S.
Smaller delays, but more often (good)

When to grant tenure?
When the number of survivors reaches a threshhold, tenure them all.
(Just move the boundary up --> fast)

Eventually, we must do a full (mark-sweep) collection and compaction.

Comparison of G.C. Algorithms

	CPU overhead	pause time (sec)	interval between pauses (sec)
ref. counting	15-20%	1.3	60-1200
deferred ref. counting	11%	1.3	60-1200
Mark-Sweep	25-40%	4.5	74
Ballard's Algorithm	7%	---	---
Generation Scavenging	1.5-2.5%	.38	30

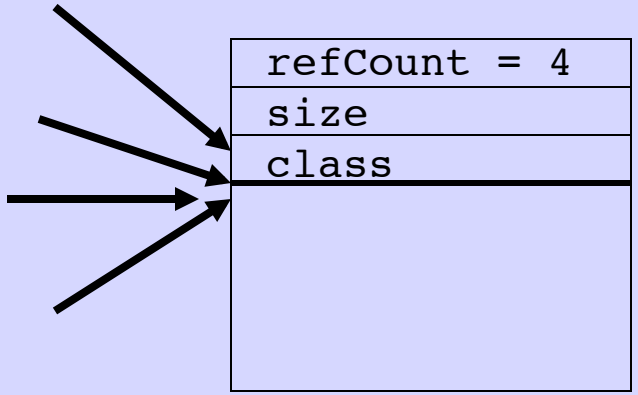
Reference Counting

Not widely used.
Ignore these slides.

Reference Counting

- For each object, store…
  - A count of "incoming pointers"

- Two operations:
  - INCREMENT the reference count
  - DECREMENT the reference count
  - Called by the bytecode interpreter
    - … every time a field is modified!



- When this count goes to zero…
  - The object is garbage.

- Maintain a list of unused garbage objects.
  - When the count goes to zero…
    - Add this object to the free list.
  - To allocate a new object, check the free list first.

- Periodically compact objects

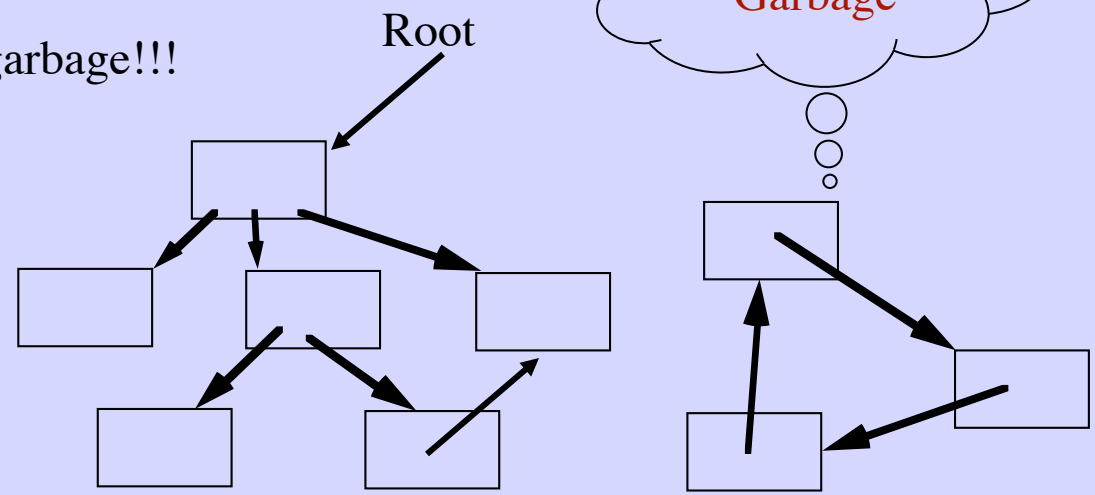
Reference Counting

Advantages:

- The work is spread out over time.
- Good for real-time/interactive systems.
  No long pauses.

Disadvantages:

- Will not identify all garbage!!!
  Cyclic objects.



  Must combine with another G.C. algorithm
  (Usually mark-sweep)
- Count field is of limited size
  Overflow? Sticks on the largest number

Reference Counting - Optimization

Deferred Reference Counting - The Deutsch-Bobrow Algorithm

"An efficient Incremental Automatic Garbage Collection Algorithm," by L.P. Deutsch and D.G. Bobrow, CACM 19:9, p. 522-526, Sept. 1976.

Observations:

- Fields in activation records (e.g., local variables) change rapidly.
- Activation records have short lifetimes.
  ARs are created & destroyed frequently.
- Garbage collection occurs much less frequently.

Optimization:

- Don't modify reference counts every time
  a local variable is modified.
- Thus, reference counts do not include pointers
  from activation record stack.
- The activation record stack will be a second "reachability root"

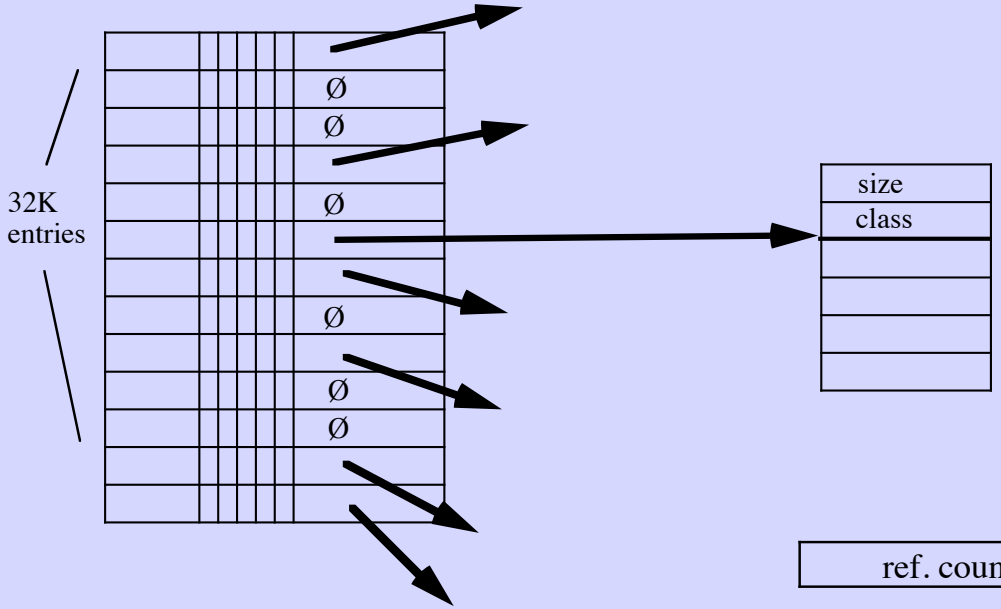
Incremental Reference Counting

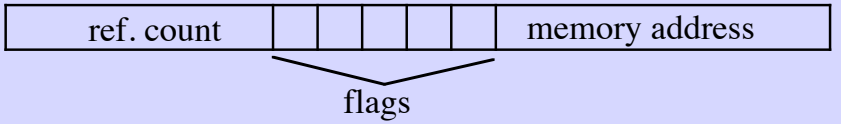
- During normal operation, whenever a reference count goes to zero...
  We can't put it on the list of free objects.
  So add it to a special list: The "Zero Count Table"
- When we run out of memory…
  - Run thru the stack of activation records
    For every pointer we find on the stack…
    Increment the reference count of the object pointed to.
  - Run through the Zero Count Table.
    If the count is still zero…
    The object is unreachable --> Add to free list
- Cleanup: Run thru the stack of activation records again.
  For every pointer we find on the stack…
  Decrement the reference count of the object pointed to.
  If zero, add back to the Zero Count Table
  Resume normal operation.
- Note: nothing is freed until the collector is run (although it may run faster).

Object Table

No longer used in Smalltalk.
Ignore these slides.

The Object Table (for 16-bit implementation)

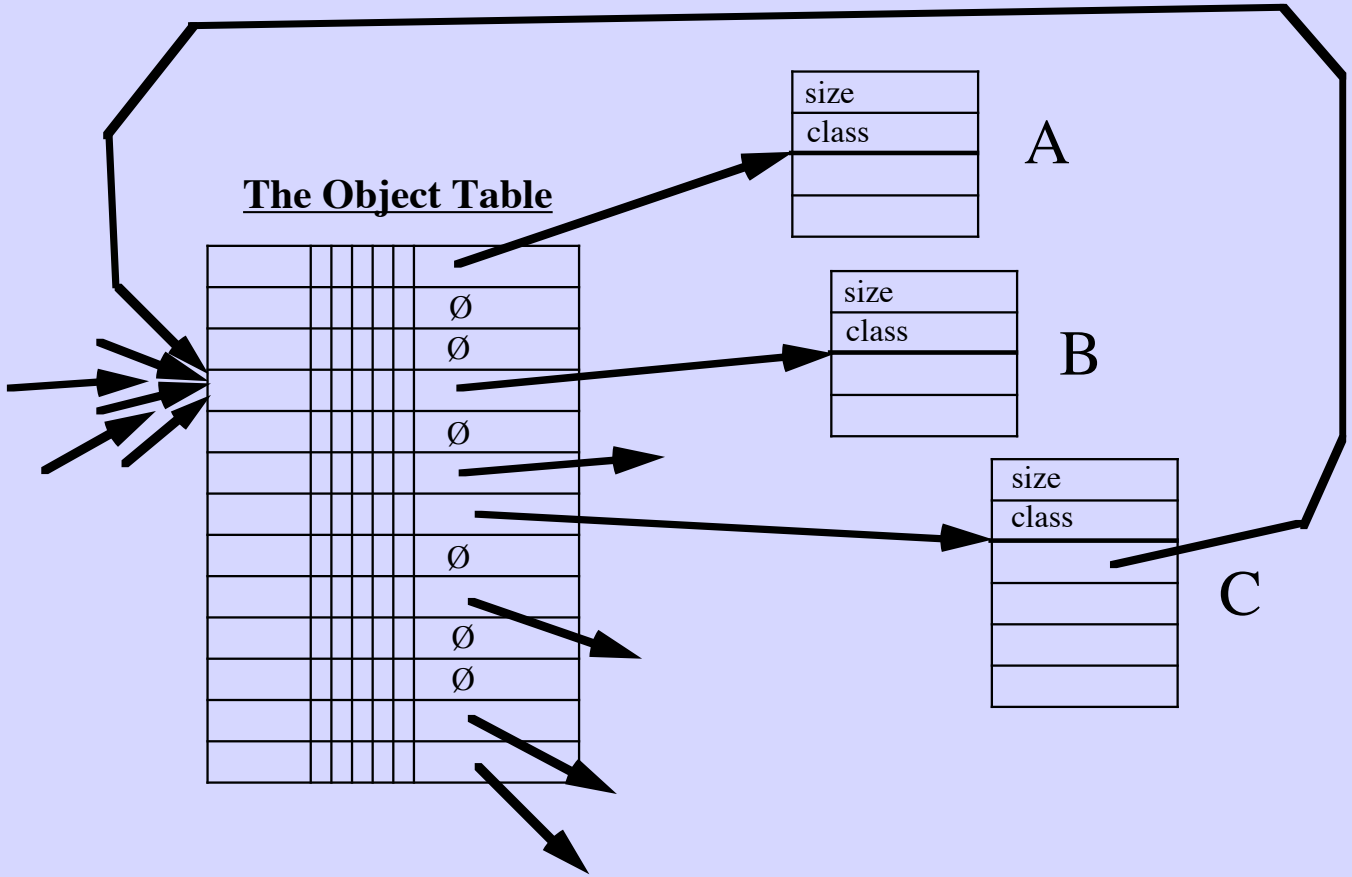




Flags:

- Free table entry
- Used by garbage collection algorithm
- Object format:
  - OOPs, SmallIntegers only
  - ByteArray
  - WordArray

Each OOP points to a ObjectTable entry.
Every pointer is indirect.
Benefit? Easy to move an object



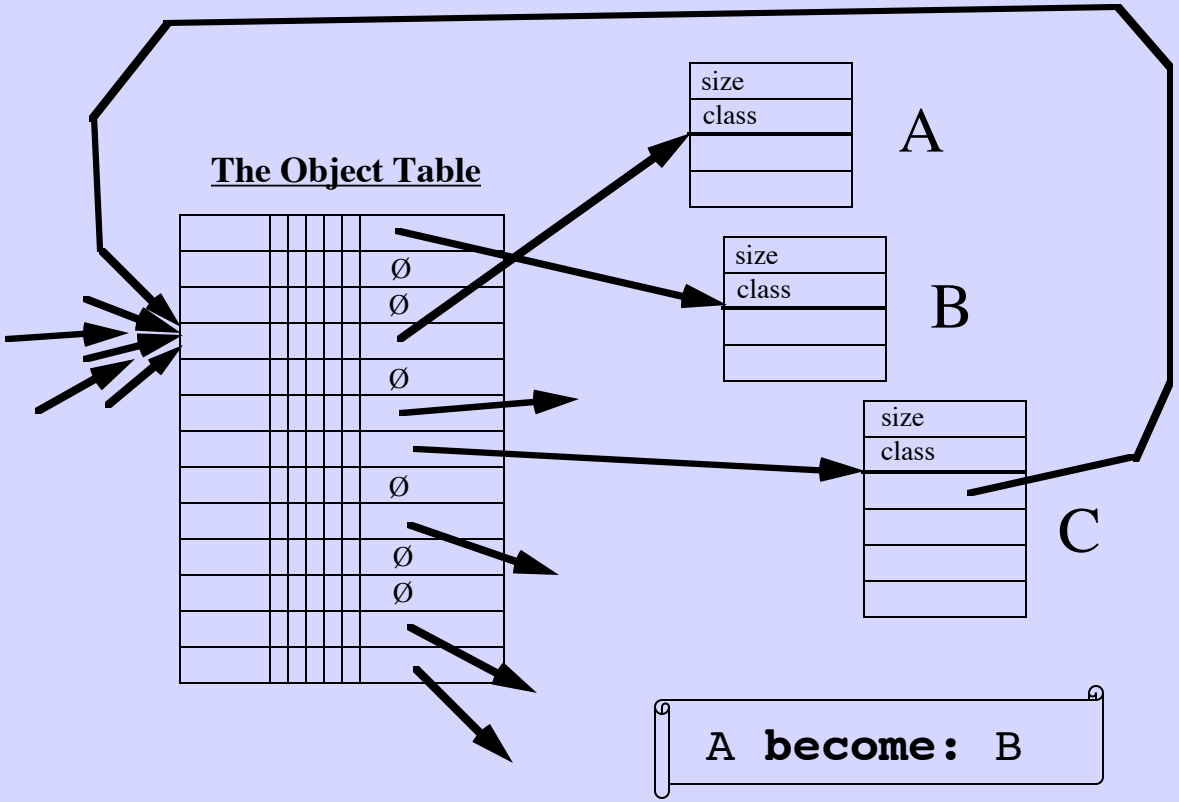
The "become:" Operation

Used to "grow" objects

Examples: OrderedCollection, Dictionary, …

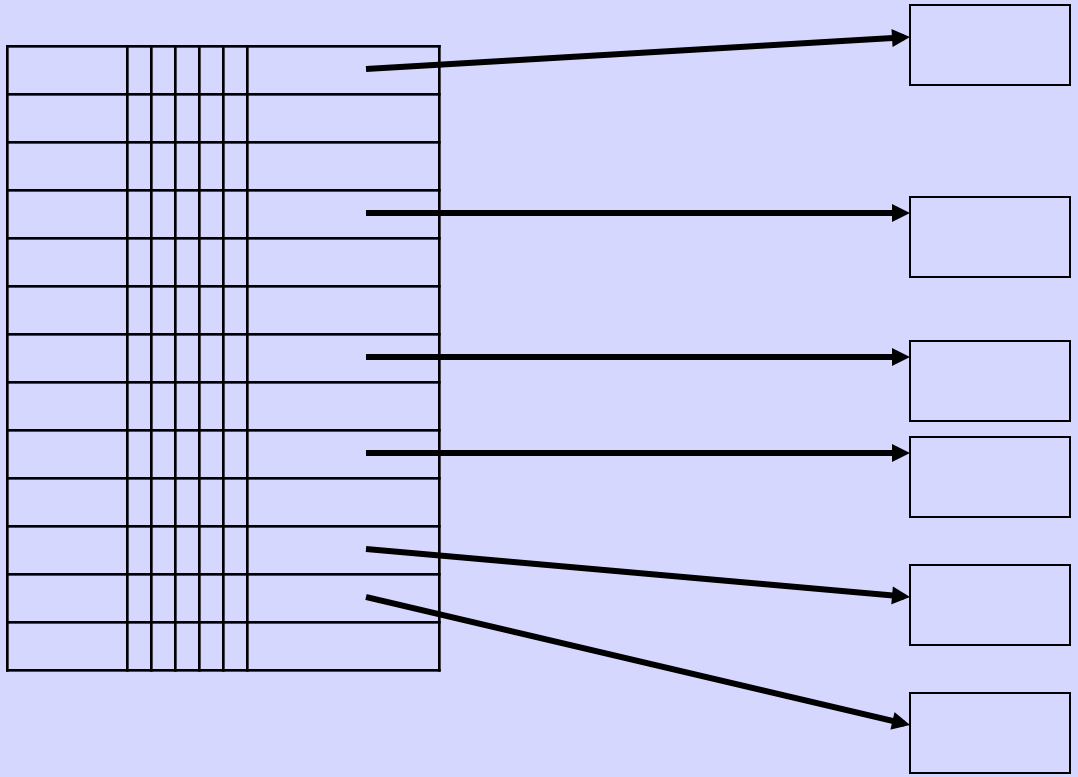
Implementation:

Swap the object table entries



Unused Object Table Entries

Keep in a linked list



Unused Object Table Entries

Keep in a linked list