Memory Allocation
and
Recycling
Parallel, Concurrent, Real-time

Resources

- Reference:
  Garbage collection : algorithms for automatic dynamic memory management, by Richard Jones, Rafael Lins
  Wiley, 1996

- Richard Jones' GC Page:
  http://www.cs.kent.ac.uk/people/staff/rej/gc.html

- Glossary: http://www.memorymanagement.org/glossary

- FAQ: http://www.iecc.com/gclist/GC-faq.html

- Repository: ftp://ftp.cs.utexas.edu/pub/garbage/

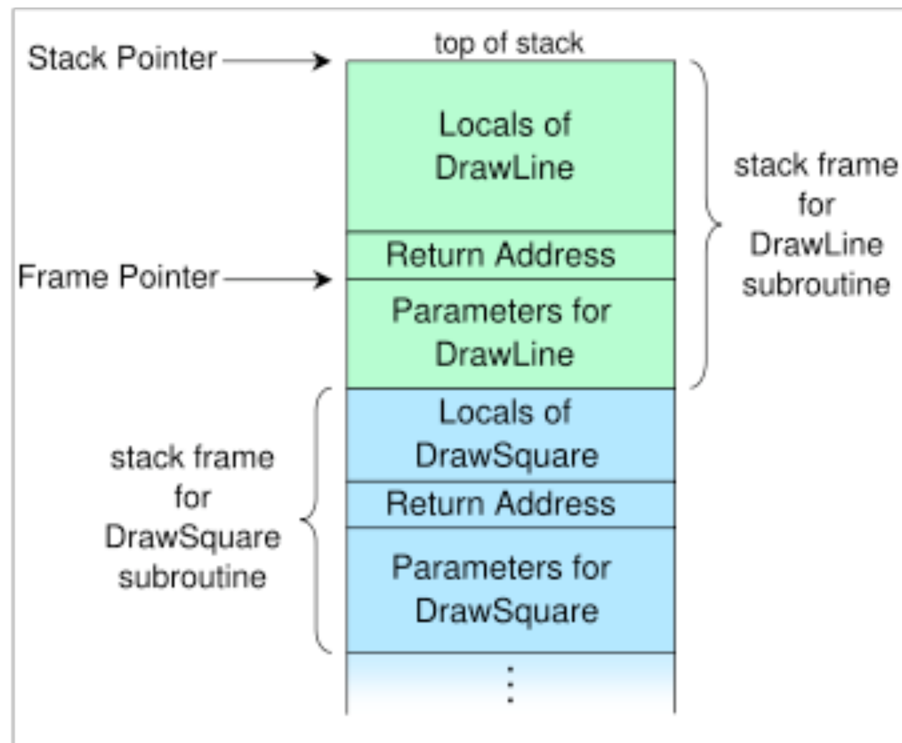
- Various presentations (David F. Bacon collection):
  http://domino.watson.ibm.com/comm/research_people.nsf/pages/bacon.presentations.html

Ways Memory is Used

- **Code**
- **Static variables**:
  remain allocated throughout execution
- **Stack variables** (aka "automatic" variables):
  e.g. arguments and local variables
  of nested functions
  These **cease to exist** after the
  function returns.
- **Dynamic variables:**
  instance variables of objects created
  during execution

Stack-Based Allocation

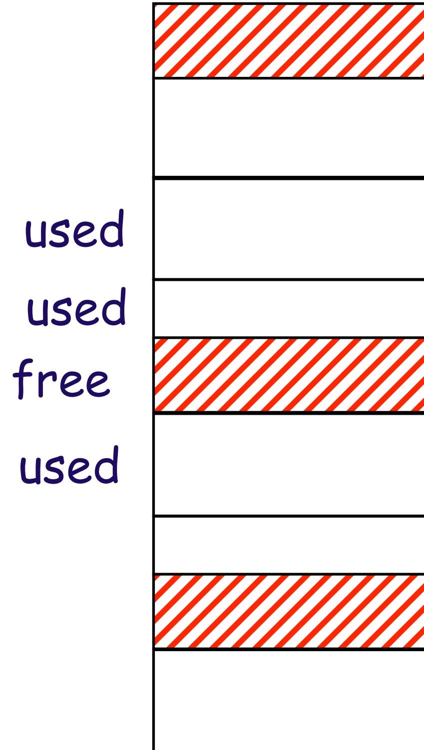
Low-overhead, but confining



From http://en.wikipedia.org/wiki/Call_stack

Heap-Based Allocation

More flexibility, but more overhead



Space overheads:

Each block stores a size.

Free blocks also store a pointer to the “next” free block.

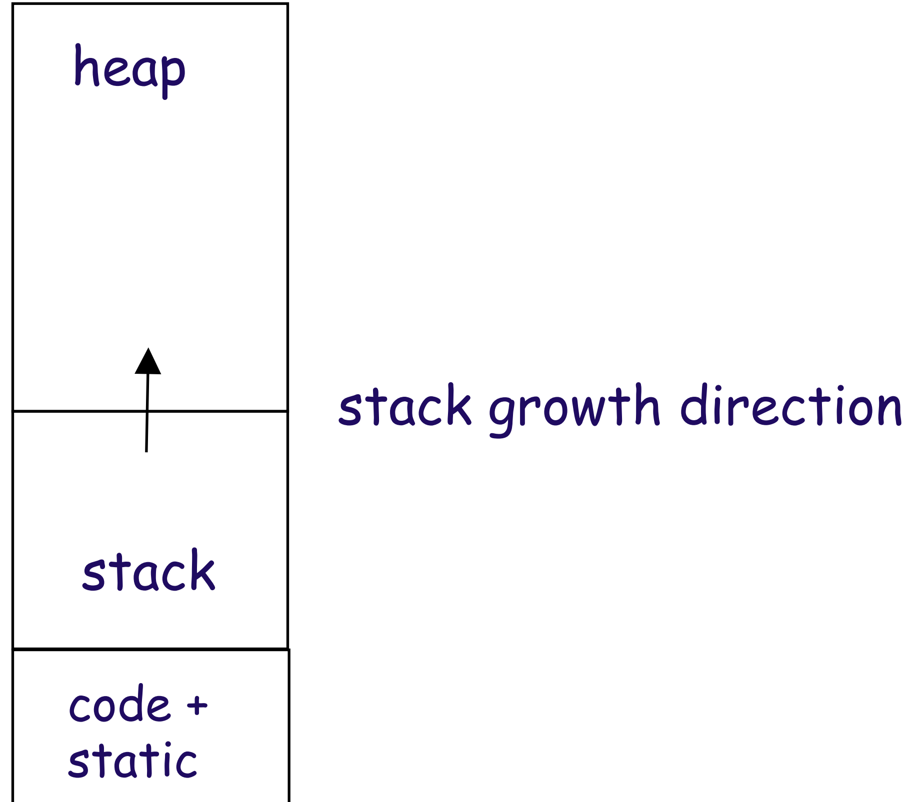
Time overheads:

Must *search* for an adequate free block.

Must *sub-divide* free blocks that are bigger than requirement.

Must coalesce blocks as they become freed.

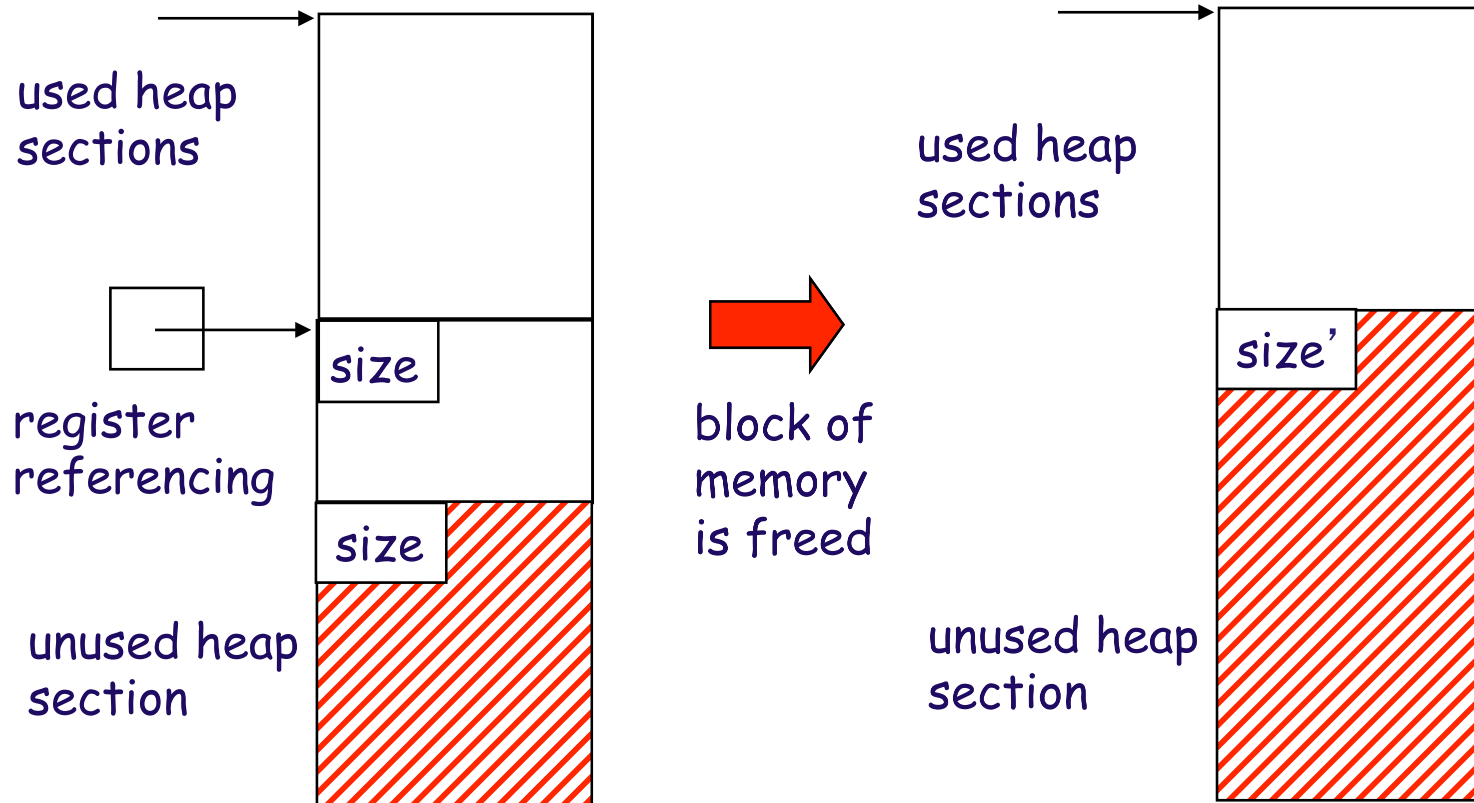
Memory is Usually Pre-Divided



Cactus Stacks

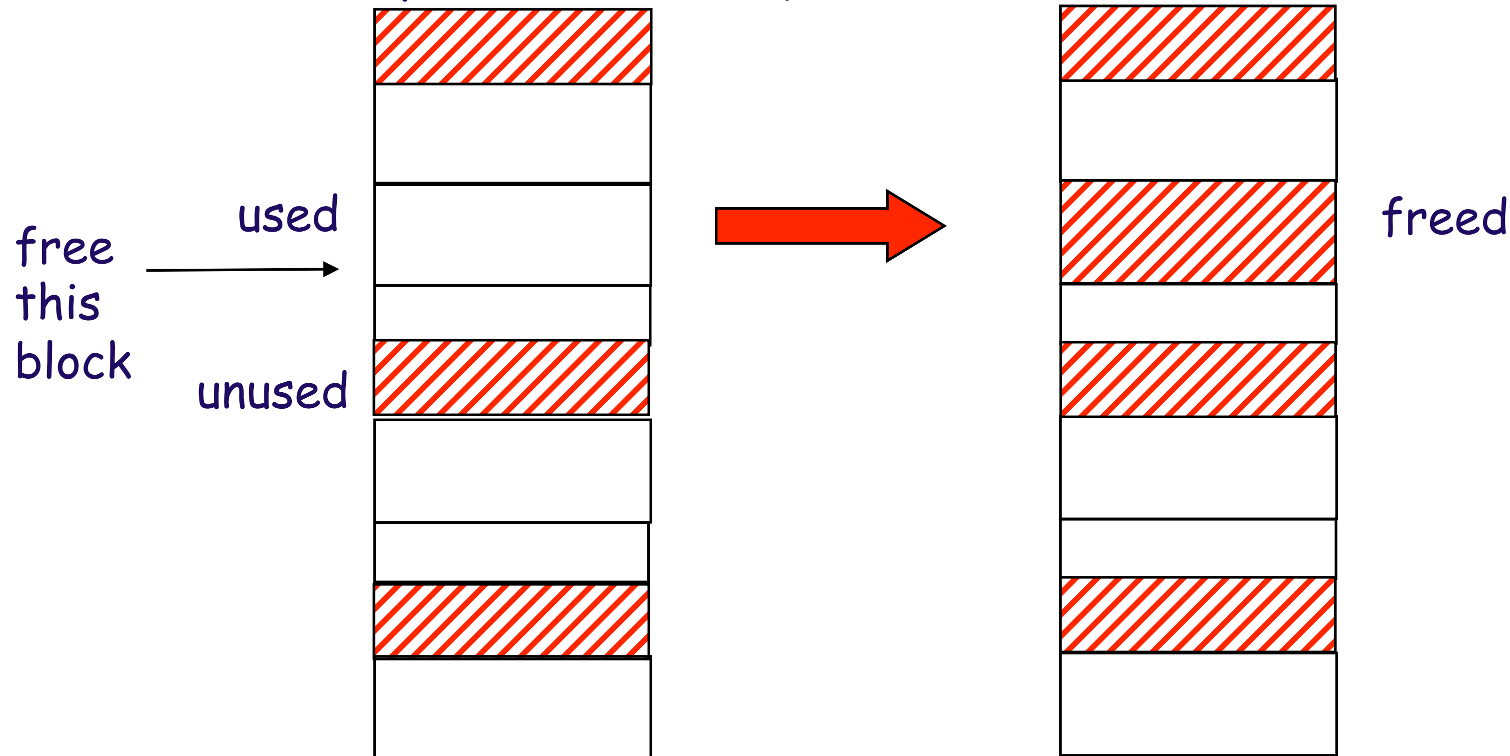
- Stacks for multithreading
- Common trunk can be shared among threads
- Branches separate
- Cilk++ is example

Heap Recycling Aspect



The Bigger Picture

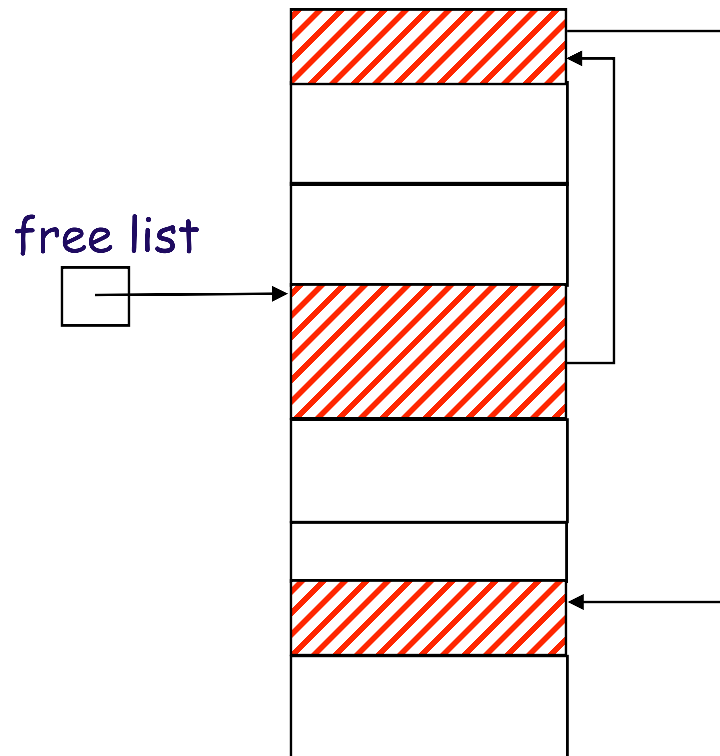
Memory becomes "fragmented" because blocks are not necessarily freed in the same order as allocated.



There are never **adjacent** unused blocks. They are always coalesced into one. (Why?)

Maintaining the “Free List”

Each unused block has a **size** field and a pointer to the **next** unused block.



Heap Issues

- Fragmentation ("checkerboarding")
  - Why is this an issue?
- Allocation Policy:
  - First-fit
  - Best-fit
  - . . .

Parallel Heap Issues

- Several threads may want to allocate or deallocate at the same time.
- They need to be coordinated.

Real-time Heap Issue

- Search for an appropriate block, and reclamation, may not have predictable execution times.

"Bump Pointer" Alternative to Free List

- By using frequent **compaction**, all free memory can be kept in one contiguous area.
- Then there is no need for a free list.
- Just keep a pointer to the first and last free locations.
- "Bump" the first pointer by the amount of the request.

Approaches to Recycling Heap Memory

- Don't-do-it approach
- Programmer-burden approach
- Automatic approaches
  - Reference-Counting
  - Garbage collection
    - Mark-Sweep
    - Copying
    - Generational
    - others

Consequences of Getting It Wrong

- Free space too early:
  Disastrous consequences – Values overwrite each other, producing nonsensical results. Dangling pointers can be left: active memory pointing to freed memory.

- Free space too late (e.g. never):
  Space leak – Program may grind to a halt prematurely, since all memory is used up.

Why is the problem hard?

- Usually pointers are one-directional.
- So it is not trivial to tell **how many active pointers** point to a given object, a requirement for determining whether an object is garbage.

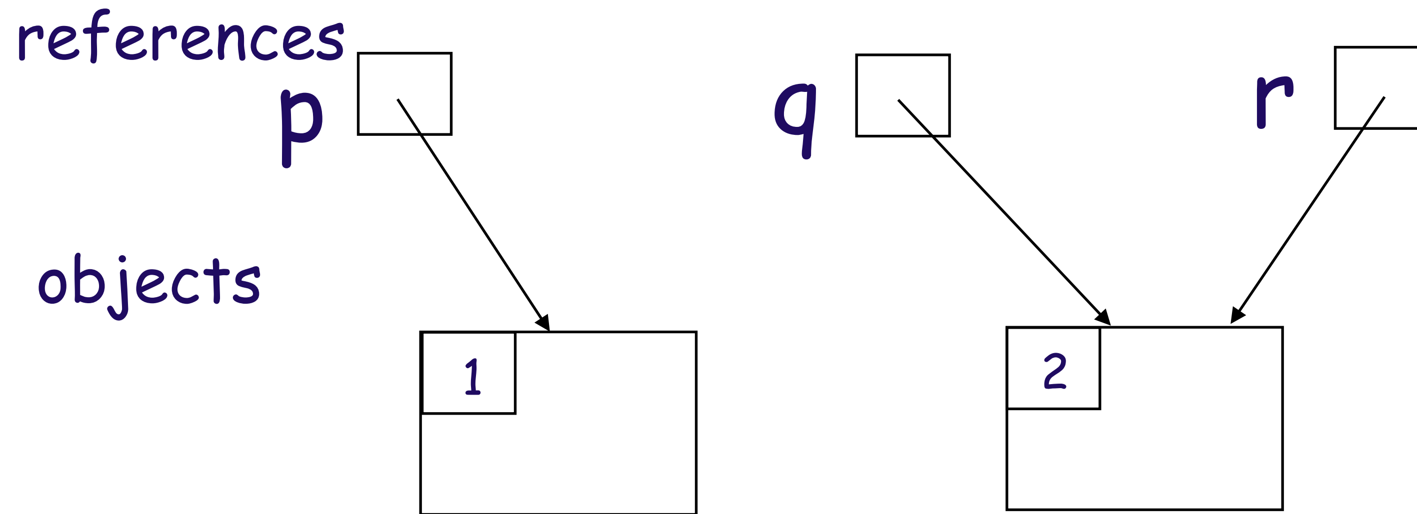
Reference Counting

- Each object has a reference count, not normally shown.
- An **invariant** is maintained:

  Reference count =

  # of references pointing to this object

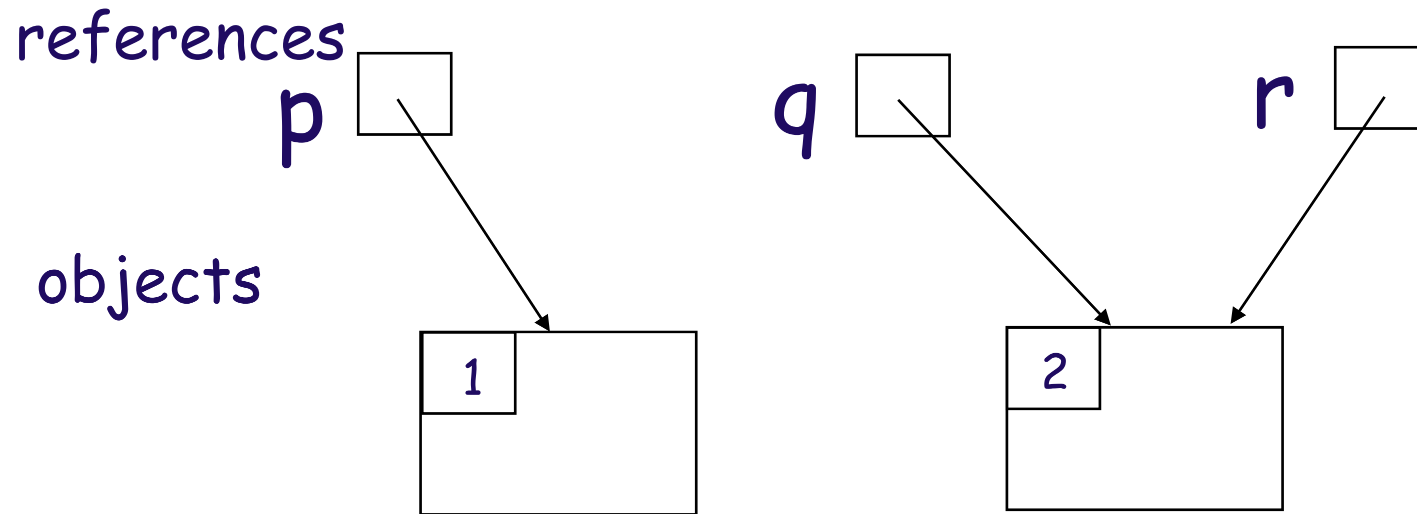
Reference Counting



What happens when we execute:

q = p;

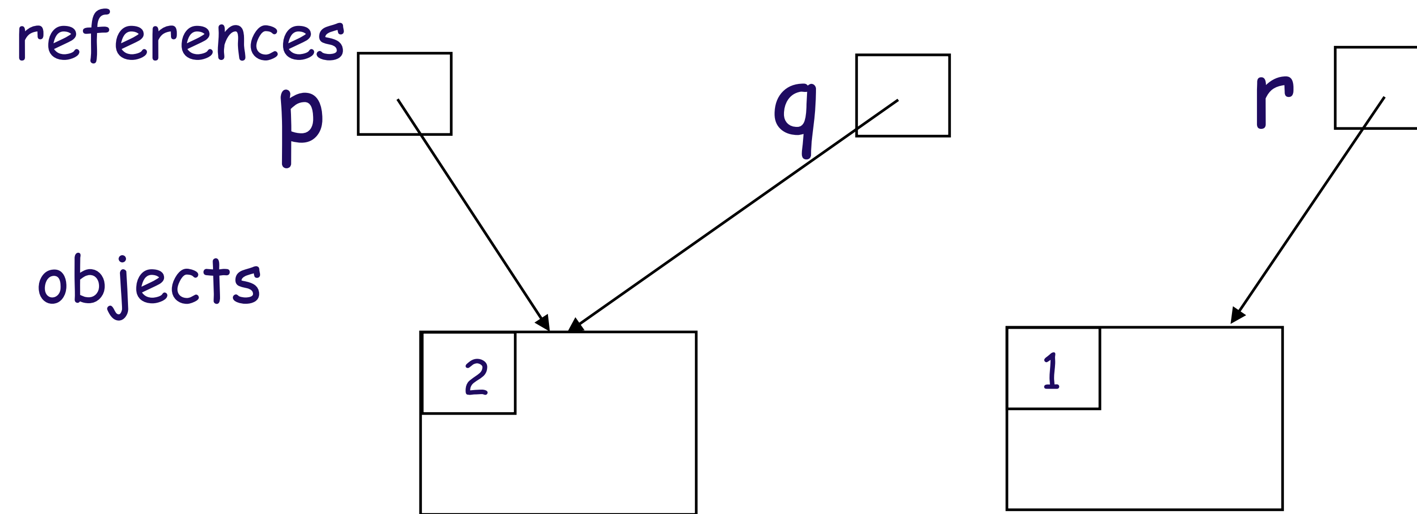
Reference Counting



What happens when we execute:

q = p; // "make q point to where p points"

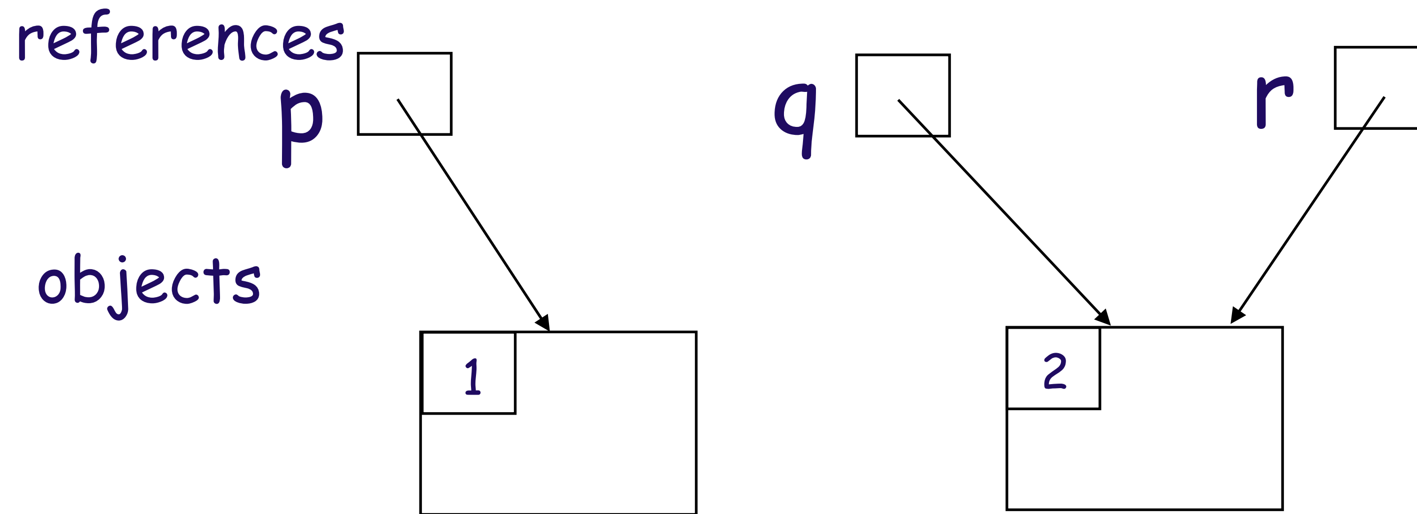
Reference Counting



What happens when we execute:

q = p; // "make q point to where p points"

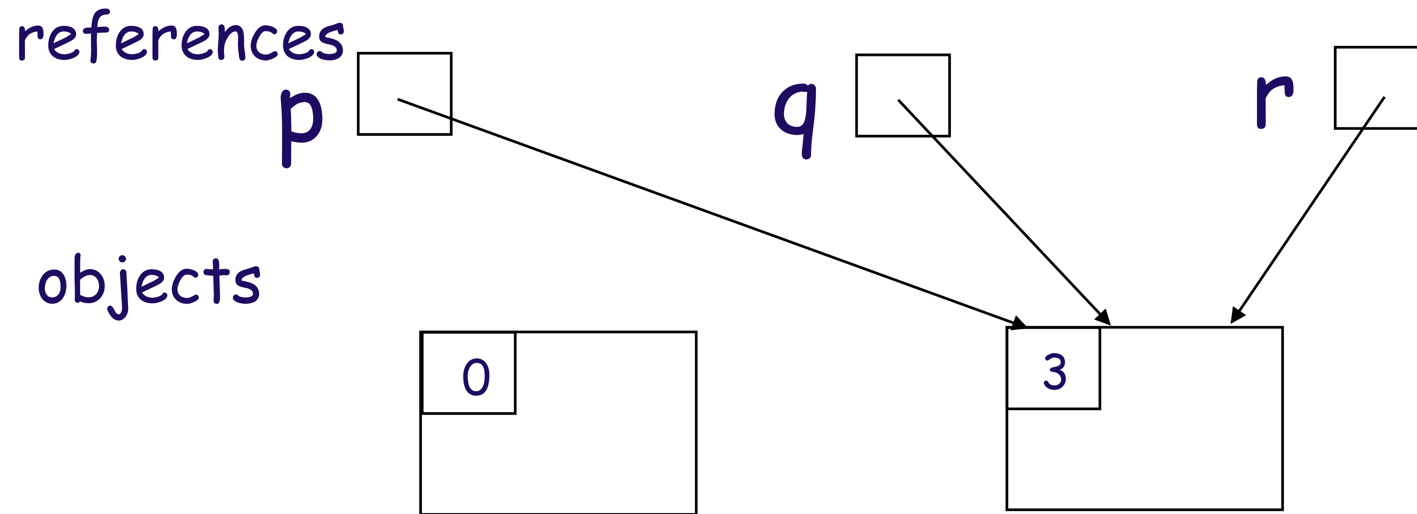
Reference Counting



What happens when we execute:

p = q;

Reference Counting

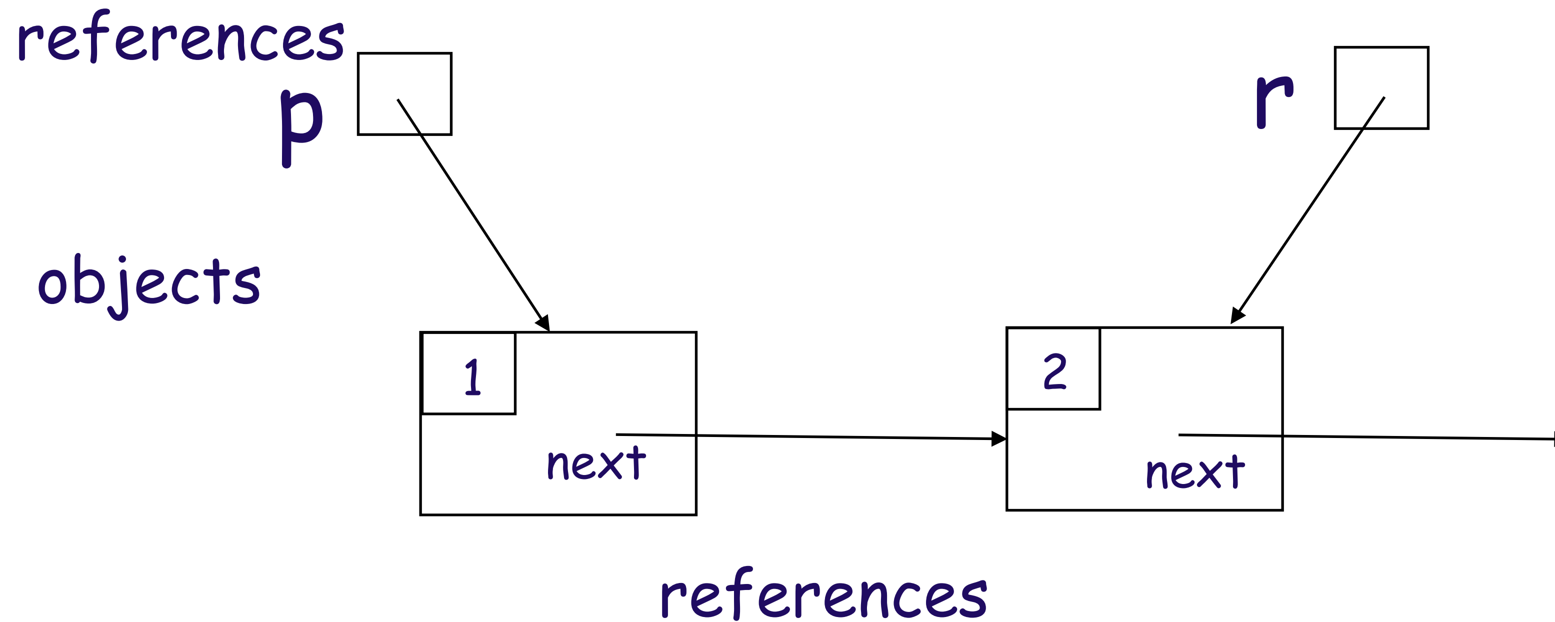


What happens when we execute:

p = q;

The block to which p formerly pointed is **reclaimable**.

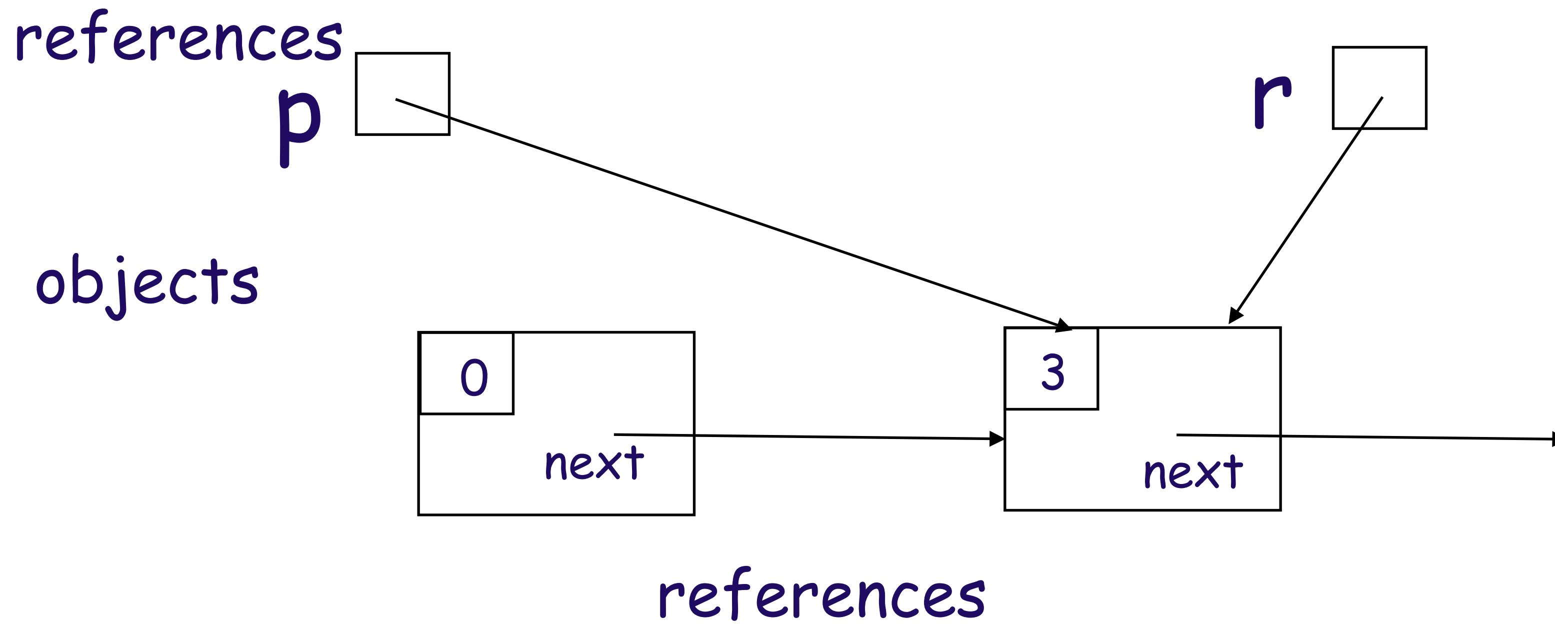
Linked List



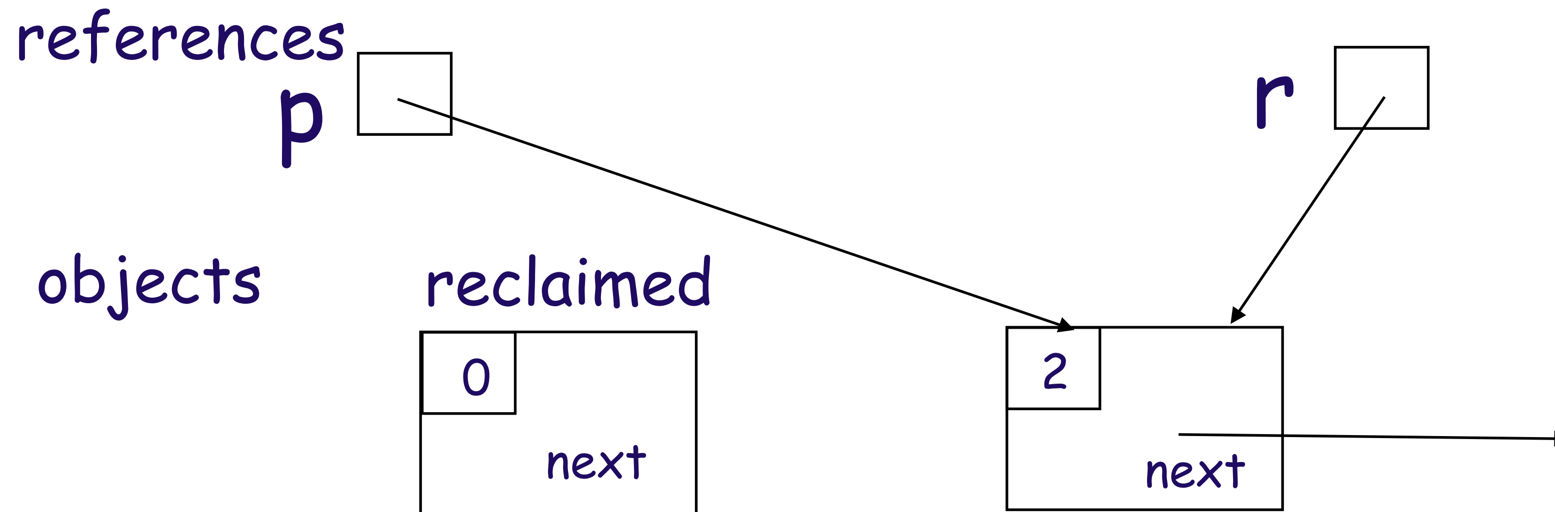
What happens when we execute:

p = p.next;

Linked List



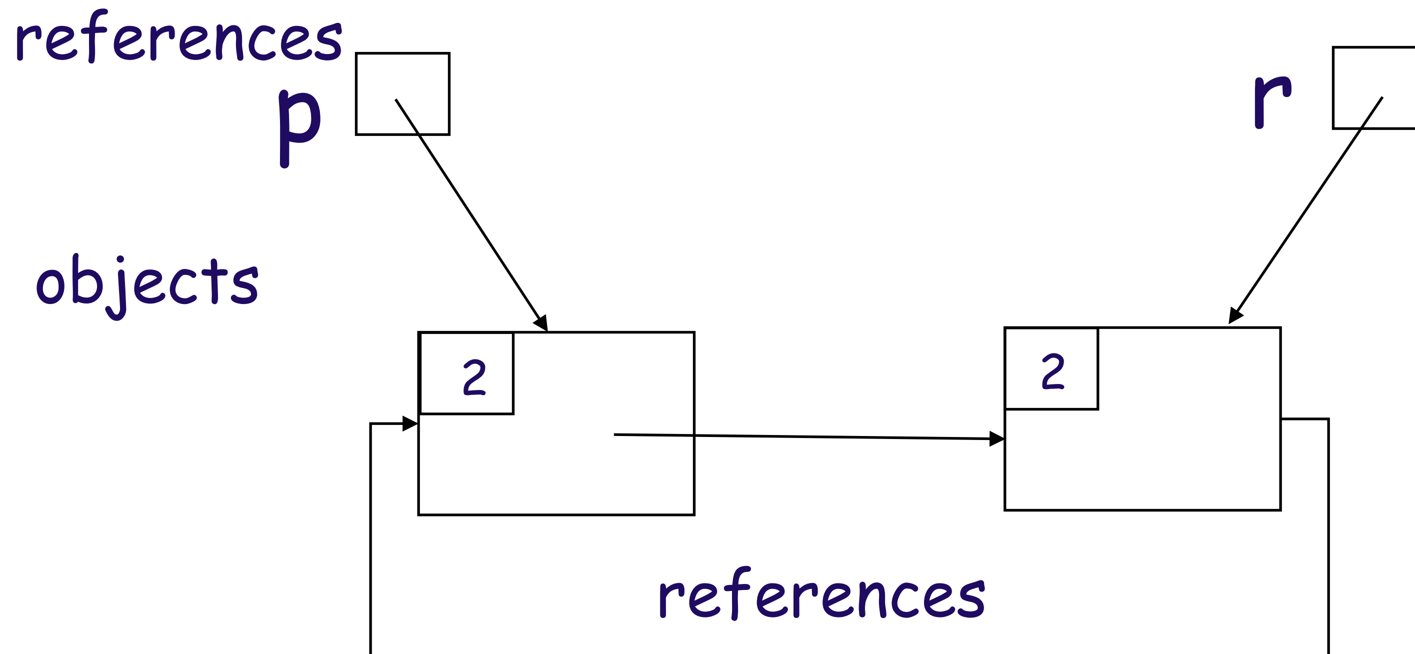
Linked List



What happens when we execute:

p = p.next;

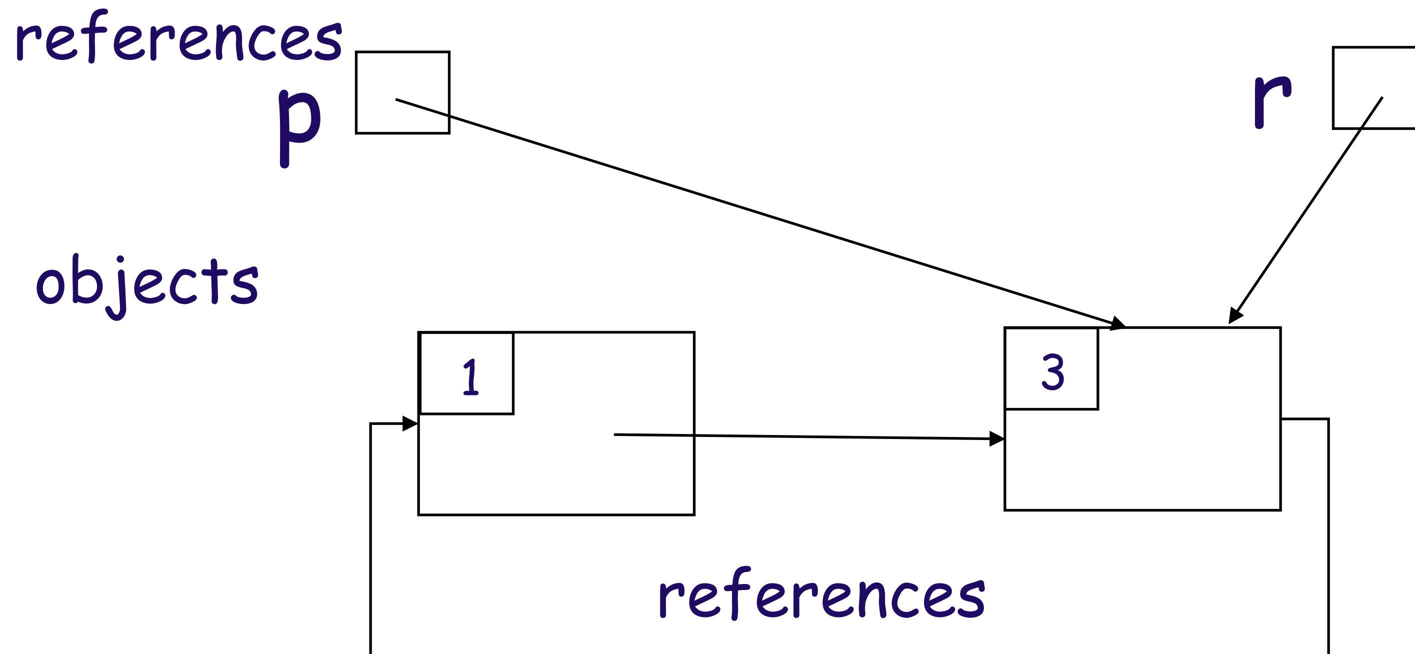
Circular Linked List



What happens when we execute:

p = r;

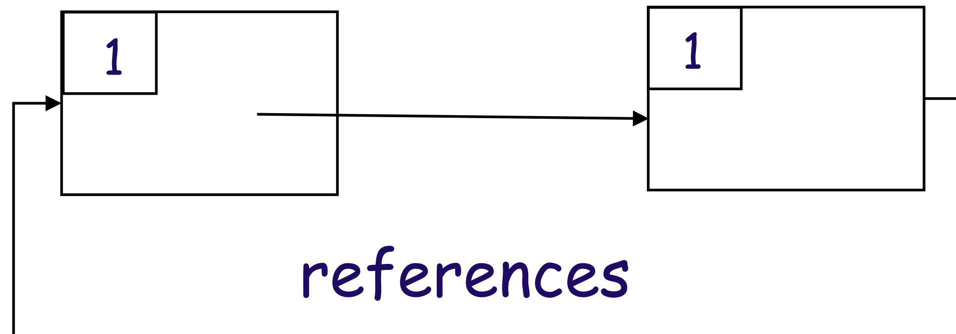
Circular Linked List



What happens when we execute:

```
p = null;
r = null;
```

Circular Linked List



Oops!

Other Reference Counting Problems

- Adds overhead to each operation.
- For parallel execution, entails additional locking or barrier synchronization on the reference count field.

Reference Count Research

- It may be possible to coarsen the grain of reference counts.
- There is recent work in this area.
- The last word probably has not been written.

Garbage Collection (GC)

- Garbage collection waits until memory is scarce, then determines what is accessible. The non-accessible objects can be recycled as free memory.
- Garbage collection does not suffer from the cyclic problem of reference counting.
- GC was invented by John McCarthy in conjunction with early Lisp implementation.

Graph Trace

- GC is essentially a **graph-search**, or more exactly, **graph-trace** problem.
- Determine what is accessible from one or more "roots" of a directed graph.
- The **complement** of accessible is inaccessible, i.e. garbage.

Graph Trace?

- Depth-First
- Breadth-First
- How could it matter?

Some GC Techniques

- Mark/Sweep
- Copying
- Generational
- Combinations
- Hundreds of Variations
- Thousands of papers written on this topic.

Mark/Sweep Garbage Collection

- Do a search (say depth-first) from the roots of the "memory graph".
- Mark any reachable nodes as you go.
- (By making a pass over *all* nodes **linearly** through memory) Sweep up any unmarked nodes into the free list.
- (This all supposes that node entities are clearly identifiable. It requires that memory be maintained appropriately.)

Memory Overhead Comparison

- Reference counting:
  - One **integer** per node
- Mark/sweep:
  - One **bit** per node

Time Overhead Comparison

- Reference counting:
  - A small tax on every operation
- Mark/sweep:
  - A big tax when memory runs out

Copying Garbage Collection

- Divide the memory into two **half-spaces**, say A and B.
- Allocate from one half-space (A or B) at any given time.
- Assuming allocating from A now. When it comes time to collect garbage, perform a depth-first search, **copying** each used record from A to B. Then switch to allocating from B.

Copying Advantages

- Trivial to **compact** as you copy. Relative locations do not get maintained.
- Compacting results in a single large unused chunk of memory on each collection, making it easier to respond to requests.
- Caution: Cannot rely on any absolute addresses, because memory is generally **relocated**.
- Need to leave “forwarding addresses” to accomplish relocation efficiently.

Copying Collector

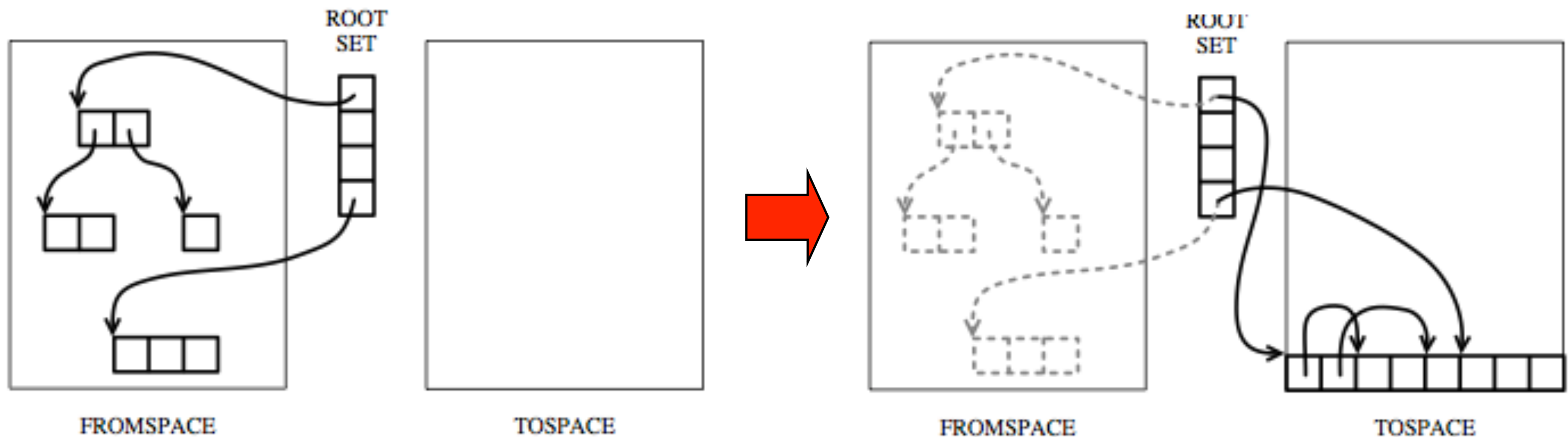


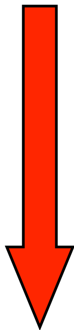
Figure 3: A simple semispace garbage collector before garbage collection.

Figure 4: Semispace collector after garbage collection.

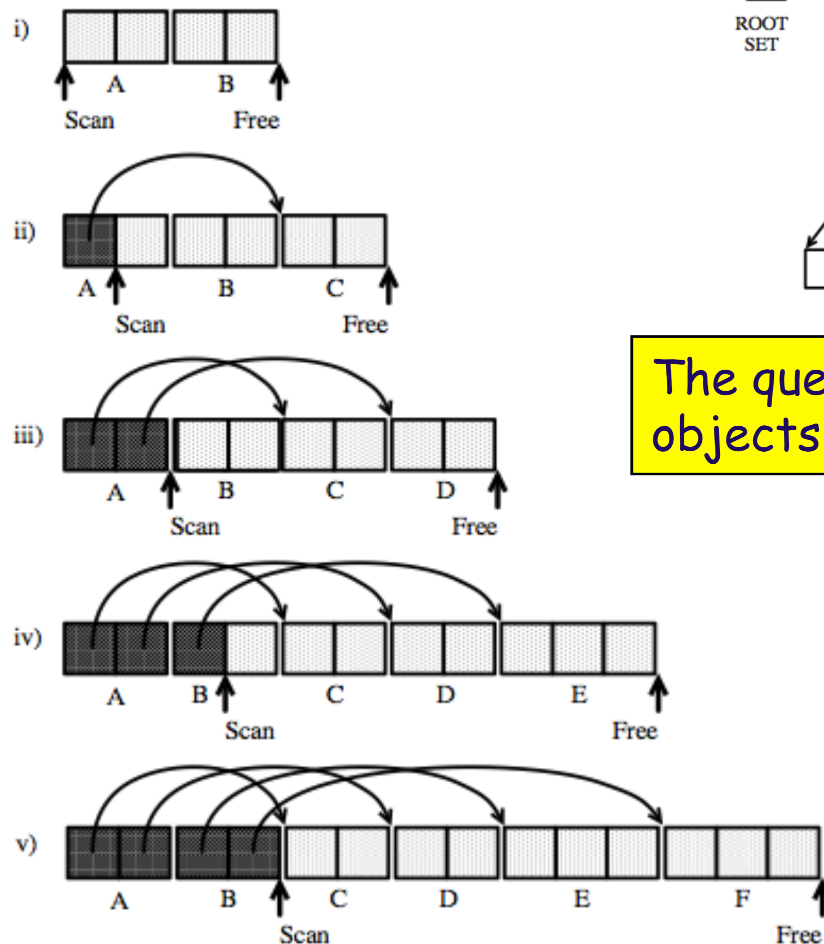
From Paul R. Wilson: Uniprocessor Garbage Collection Techniques

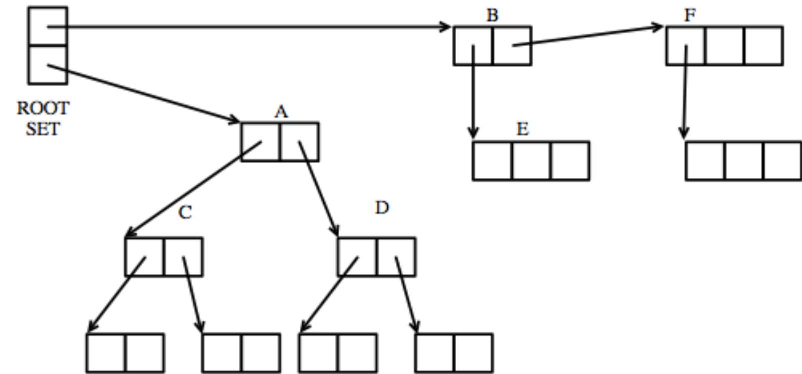
Example: Cheney's Method
= Breadth-First Copying

Steps



Queue





The queue is built of the objects themselves.

From Paul R. Wilson: Uniprocessor Garbage Collection Techniques

Copying GC Disadvantages?

Generational Garbage Collection

- This is one of many heuristics used to reduce overhead in GC.
- It was introduced by Lieberman and Hewitt in 1983 (CACM), and inspired by Baker's real-time collector (described later).
- It can be observed that some nodes are **ephemeral** (temporary and quickly become garbage), while others have great **longevity**.
- Thus devise a way to do a **quick partial** collection to pick up the ephemeral nodes, reserving a full GC until more desperate.

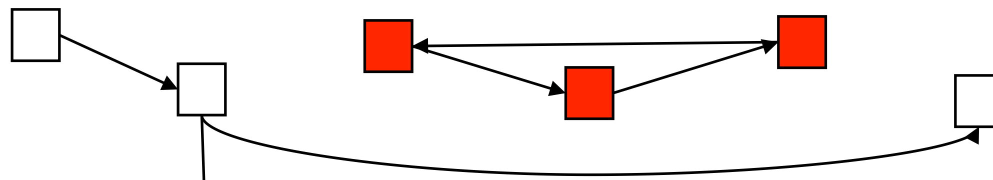
Generational Garbage Collection

- The extension of the dichotomy ephemeral vs. long-lived is achieved by assigning nodes to **generations**.
- A node in the **youngest** generation is usually the most likely to become garbage.
- References can freely point from a younger to an older generation, but in the other direction only by special consideration.
- If a node survives collection at one generation, it is **promoted** to the next older generation.
- A generation is collected only if there is not enough memory freed by collecting younger generations.

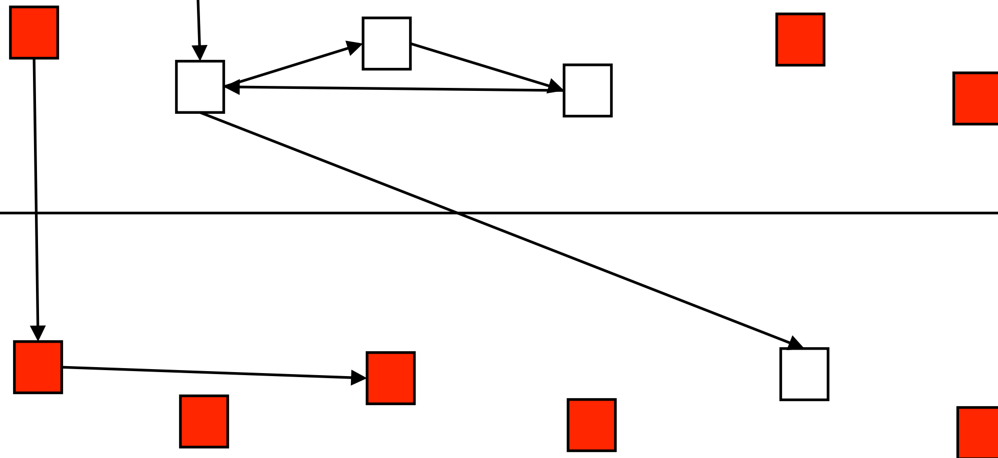
Generational Garbage Collection

younger

garbage



Any pointers this direction have to be treated as roots.



older

Concurrency: Modes of Interaction

- Several threads that don't interact on the surface can implicitly interact through the garbage collector.
- Garbage collection could take place while the threads are running.
- Concurrency could be used to speed-up the garbage collection itself.

Concurrency

- It is intended that both marking and collecting proceed concurrently with the main computation.
- Of course, collecting can only take place after marking is complete.

Sequential vs. Concurrent Distinctions

- Typically sequential collection is **STW** ("stop the world") – the main computation stops until collection is done. So all garbage can be expected to be collected when the collector stops.

Sequential vs. Concurrent Distinctions

- In concurrent collection, new garbage can be created while the collector is operating.
- We thus can't expect a given collection to find all garbage. (There will be some "floating garbage".)
- Newly-created garbage will need to wait until the next collection cycle.

Guy Steele's
Multiprocessing Compactifying Garbage Collector

- Possibly the first paper to address the problem in a multiprocessing context.

1975 ACM Student Award
Paper: First Place

Multiprocessing Compactifying Garbage Collection

Guy L. Steele Jr.
Harvard University

Communications of the ACM
September 1975
Volume 18
Number 9

State of the Art in 1975

As an example, an average MACSYMA [2] job running interactively in a MacLISP on a Digital Equipment Corp. PDP-10 computer [4] contains approximately 50,000 to 70,000 36-bit words of list data, and a garbage collection typically takes 1500 to 3000 msec of run time (about two to five times that much real time under daytime time-sharing loads).

If the garbage collection technique is to be retained, then the only way to avoid this suspension of operations is to introduce parallelism; that is, to garbage collect while list operations are going on. (Knuth credits this idea to M. Minsky [10, exercise 2.3.5–12].)

Steele

- Adds an extra “flag” bit along with the mark bit.
- The flag bit indicates whether or not the object has already be relocated in the “to-space”.
- The algorithms are presented in great detail (six 2-column CACM pages).
- Synchronization is expressed using semaphores.

Table I. Meanings of Mark and Flag Bits

Mark bit Flag bit	*false* *false*	*false* *true*	*true* *false*	*true* *true*
Mark phase	Cell not yet seen by mark and trace routine.	(Does not occur during mark phase.)	Cell seen by mark and trace routine. Cell is therefore accessible.	Cell on freelist. Should not be seen by mark and trace routine.
Relocate phase	Discarded cell. May be used to relocate an accessible object into if necessary.	Relocated cell. First pointer component indicates new location.	Accessible cell. May be relocated into new place if necessary.	Cell on freelist. Ignored by relocate phase.
Update phase	Discarded cell. Ignored by update phase.	Relocated cell. Ignored by update phase.	Accessible cell. Pointer components may need to be normalized.	Cell on freelist. Ignored by update phase.
Reclaim phase	Discarded cell. May be returned to freelist.	Relocated cell, now discarded. May be returned to freelist.	Accessible cell. Ignored by reclaim phase.	Cell on freelist. Ignored by reclaim phase.

Operating Systems

R.S. Gaines
Editor

On-the-Fly Garbage Collection: An Exercise in Cooperation

Edsger W. Dijkstra
Burroughs Corporation

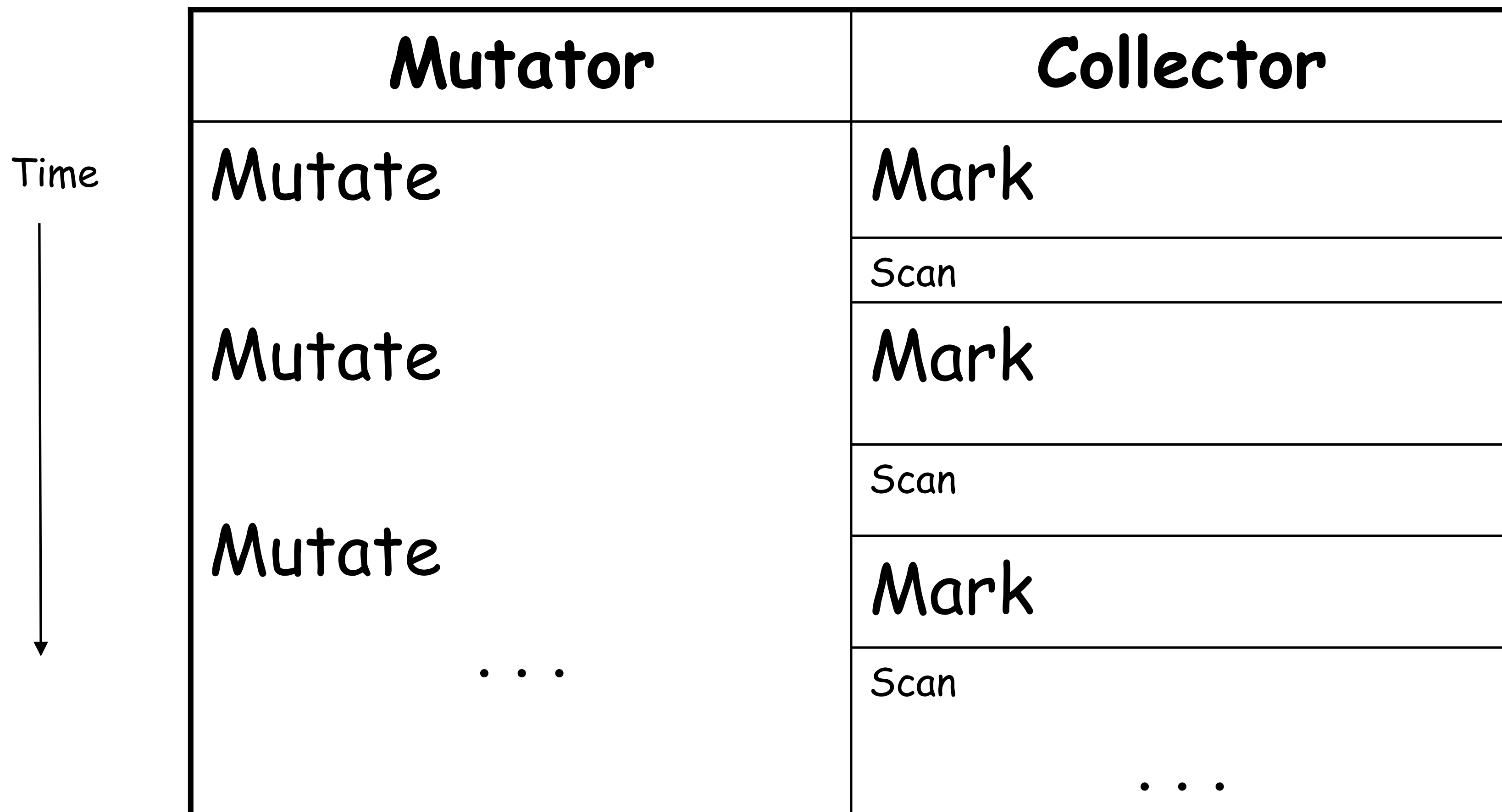
Leslie Lamport
SRI International

A.J. Martin, C.S. Scholten, and
E.F.M. Steffens
Philips Research Laboratories

Dijkstra, Lamport, et al. On-the-Fly Garbage Collection

- May have been the first to introduce "**tri-color**" marking, to be described.
- A ***collector*** is operating **concurrently** with a ***mutator*** (object program) which creates garbage.
- A mark/scan mode of operation is used.

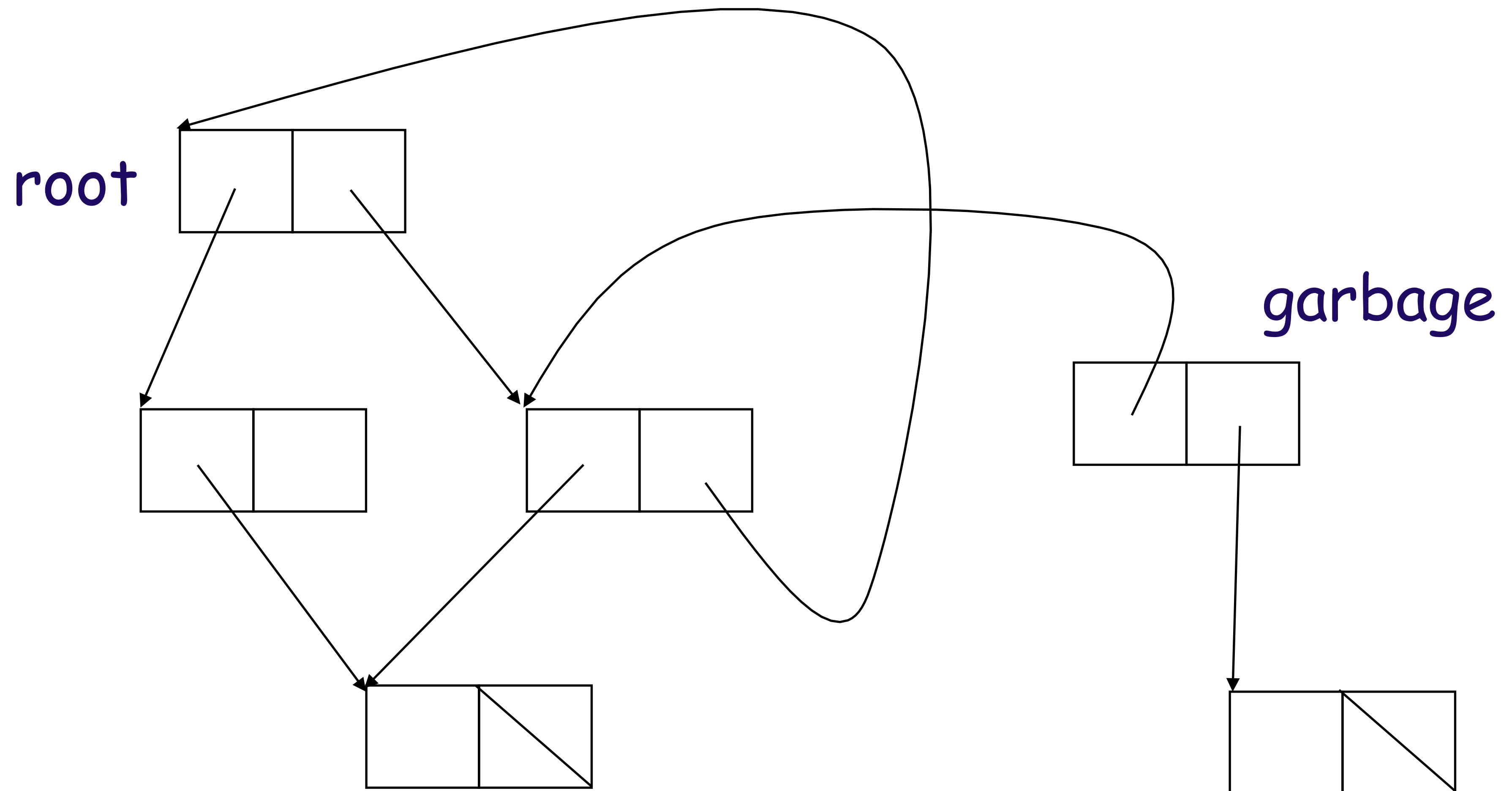
On-the-Fly Collection Timing



Focus: Correctness proof

It has hardly been our purpose to contribute specifically to the art of garbage collection, and consequently no practical significance is claimed for our solution. For that reason we felt justified in tackling a specific form of the garbage collection problem as it presents itself in the traditional implementation environment of pure Lisp. We are aware of the fact that we have left out of consideration several aspects of the garbage collection problem that are important from other points of view

Lisp-Like Data Structures Assumed



Reachable Nodes

- The root is reachable.
- If there is a connection from A to B and A is reachable, then B is reachable.
- Potentially reachable nodes are also called "live".
- Unreachable nodes are garbage.
- Nodes that are not reachable, but not recognized as such are called "floating garbage".

Terminology

- **Mutator**: thread acting on behalf of the user program
- **Collector**: garbage collector thread (includes marking and scanning)

How is concurrent collection and mutation possible?

- It is only a necessary condition that any node thought to be garbage actually is.
- It is ok if a node is thought to be non-garbage, but actually is garbage
- as long as it can be collected on a later collection cycle.

Mutator Operations Assumed

- **Redirecting** an arrow from a reachable node to:

  (1) one that is already reachable, or

  (2) one that is not yet reachable.

Operations Assumed

- **Adding** an arrow from a reachable node to:

  (3) one that is already reachable, or

  (4) one that is not yet reachable.

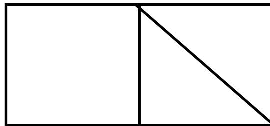
Operations Assumed

- (5) **Removing** an arrow from a reachable node.

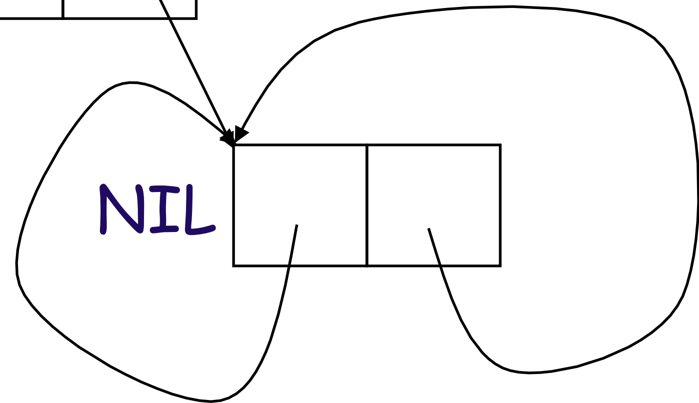
Reducing Operation Set

- Turn the null reference into an actual node, NIL.
- Make NIL's pointer fields point to itself.
- Nodes are initialized with all of their pointers pointing to NIL.

NIL Hack



becomes



Using NIL

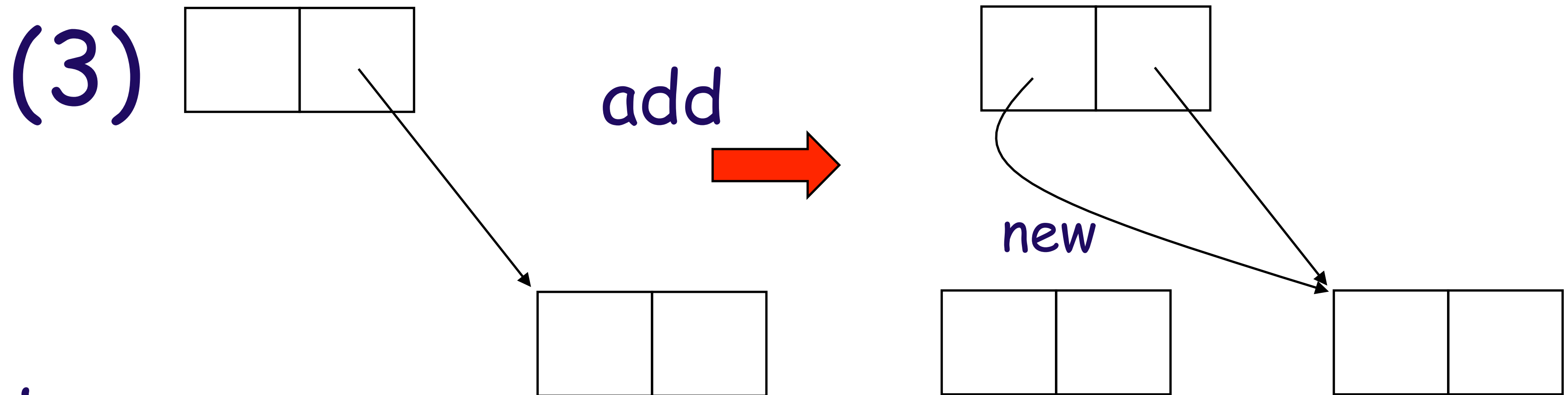
- Now (3): **Adding** an arrow from a reachable node to one that is already reachable.

becomes an instance of

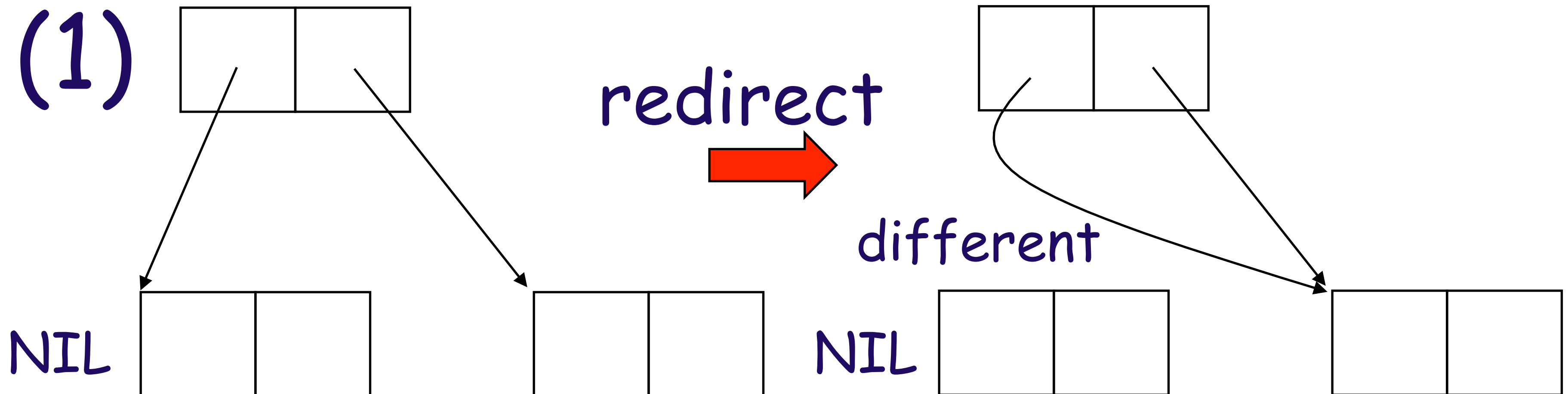
(1) **Redirecting** an arrow from a reachable node to one that is already reachable.

Similarly (4) becomes an instance of (2).

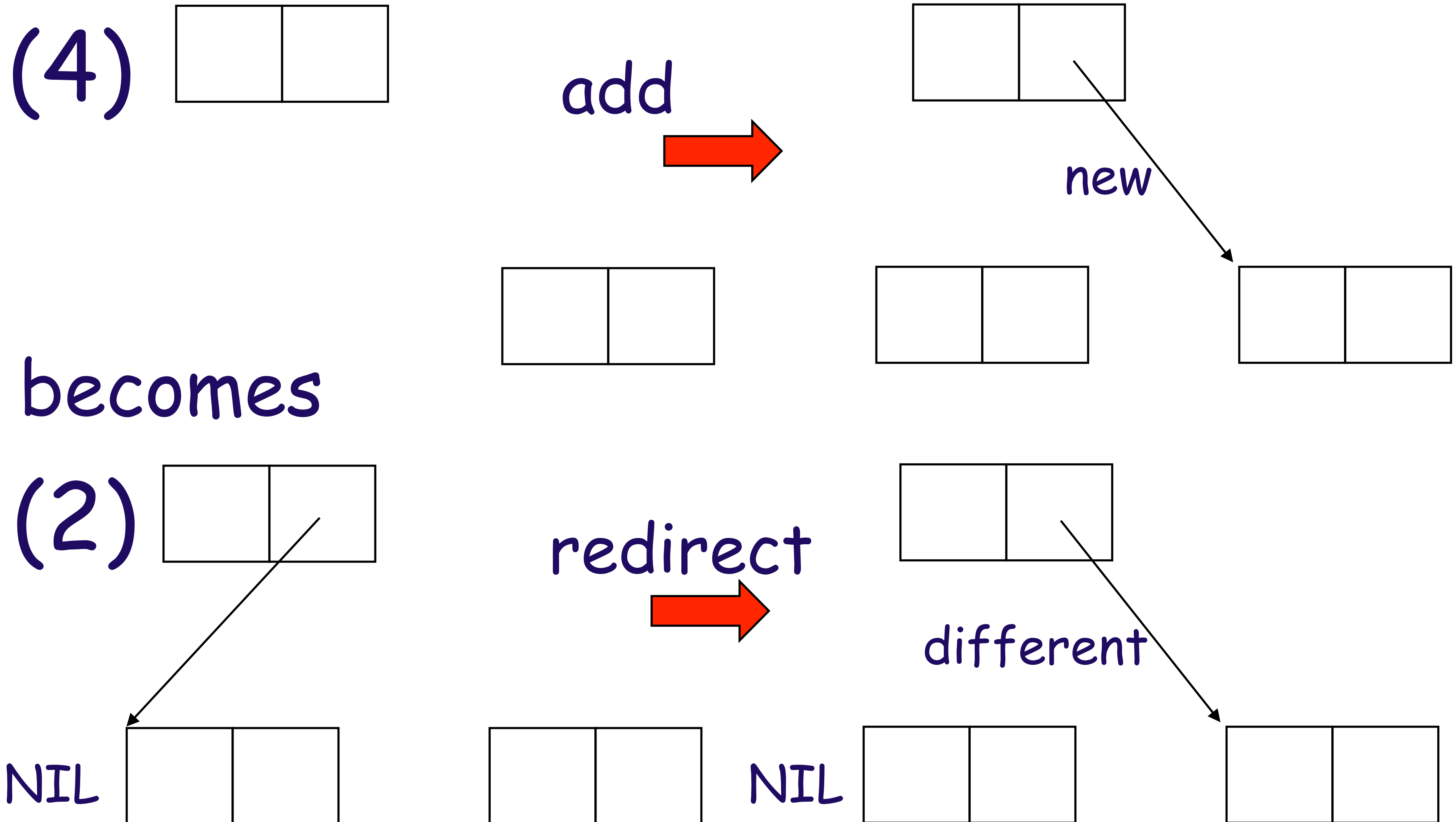
(3) becomes (1)



becomes



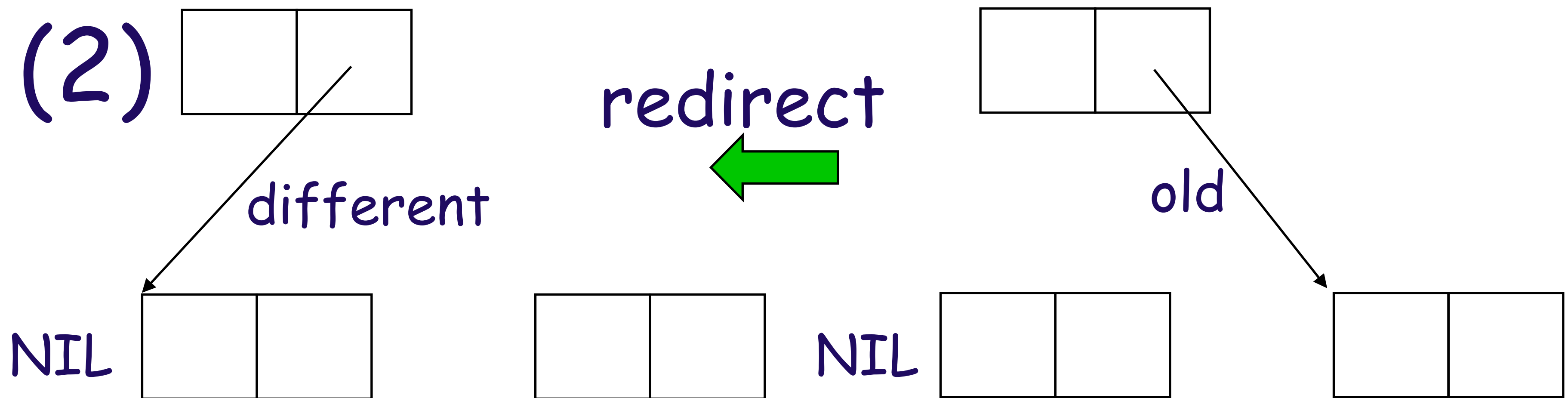
(4) becomes (2)



Reducing all to (1)

- (5) becomes (2): Just reverse the arrows in the previous diagram.
- Consider the free list to be a segregated part of the main structure.
- Then (2) becomes (1).

(5) becomes (2)



Marking Color Scheme

- At the start of marking, all nodes are reset to color "white".
- A node is colored "black" when it has been "processed".
- At the end, all nodes are either white or black.
- White nodes are not reachable, thus can be reclaimed.

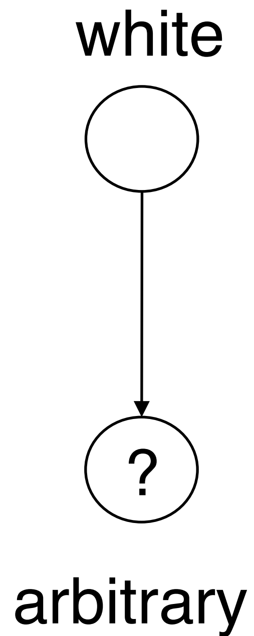
Invariant Used for Establishing Correctness

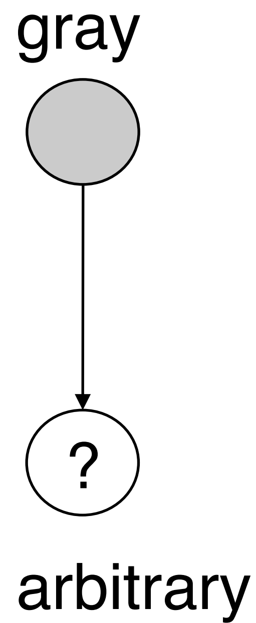
- A black node cannot point to a white node.
- Sensible, because if a reachable node points to something, the latter is also reachable.
- While desirable, some collection algorithms may violate this invariant, but compensate in a different way.

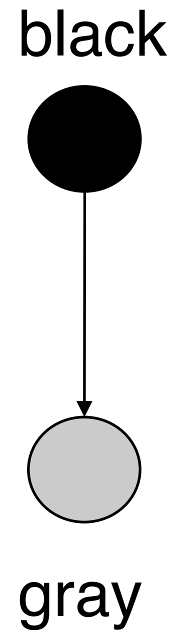
Need Another Color

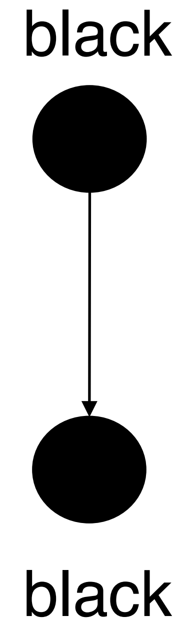
- To avoid violating the invariant during marking, one other color "gray" is used.
- Gray says the node is reachable, but its immediate descendants have not all been marked.
- Black says the node is reachable, and its immediate descendants have been marked "at least gray".
- Black can point to gray or black.
- Gray can point to anything.

Allowable Configurations









Collection Algorithm Summary

- Color all nodes white.
- Color the roots gray.
- **while** there is at least one gray node:
  - Mark the children of gray nodes at least gray.
  - Mark the node black.
- The nodes that are white can now be collected.

Implied Grayness

- Gray does not necessarily entail an added bit.
- Gray could be identified as the set of nodes being on a marking stack or queue for example.
- Then a marked node would be interpreted as gray if on the stack, or black if not.

Example

In Cheney's algorithm:

Grayness is implied to be the elements between the forward and back pointer (i.e. the elements "on the queue")

More Important Invariant

To avoid collecting nodes that are not garbage:

> For each reachable node n,
> there is *at least one* path
> from some gray node to n.

"Variant" Function Used for Establishing Termination

- At the start of marking, all nodes are white.
- During the marking process, nodes can only get darker (white to gray or black, or gray to black), not lighter.
- A gray node will eventually become black (when its descendants all are).
- Thus the **variant** is the number of white nodes + the number of gray nodes.
- The variant decreases monotonically, and being non-zero is a requirement for continuing.

"Variant" Function Used for Establishing Termination

- Color all nodes white.
- Color the roots gray.
- (The variant now has its initial value.)
- **while** there is at least one gray node:
  - (The variant is positive.)
  - Mark the children of gray nodes at least gray.
  - (The variant does not increase.)
  - Mark the node black.
  - (The variant decreases.)
- The nodes that are white can now be reused.
- (The variant is 0.)

Mutator/Collector Cooperation

- The mutator's activity is interleaved with the collector's.
- There needs to be some form of **cooperation** to prevent nodes from being incorrectly identified as garbage.
- Cooperation could take the form of synchronization, although more typically, more economical "**barriers**" are used.

Effect of Non-Cooperation, Invariant Violation

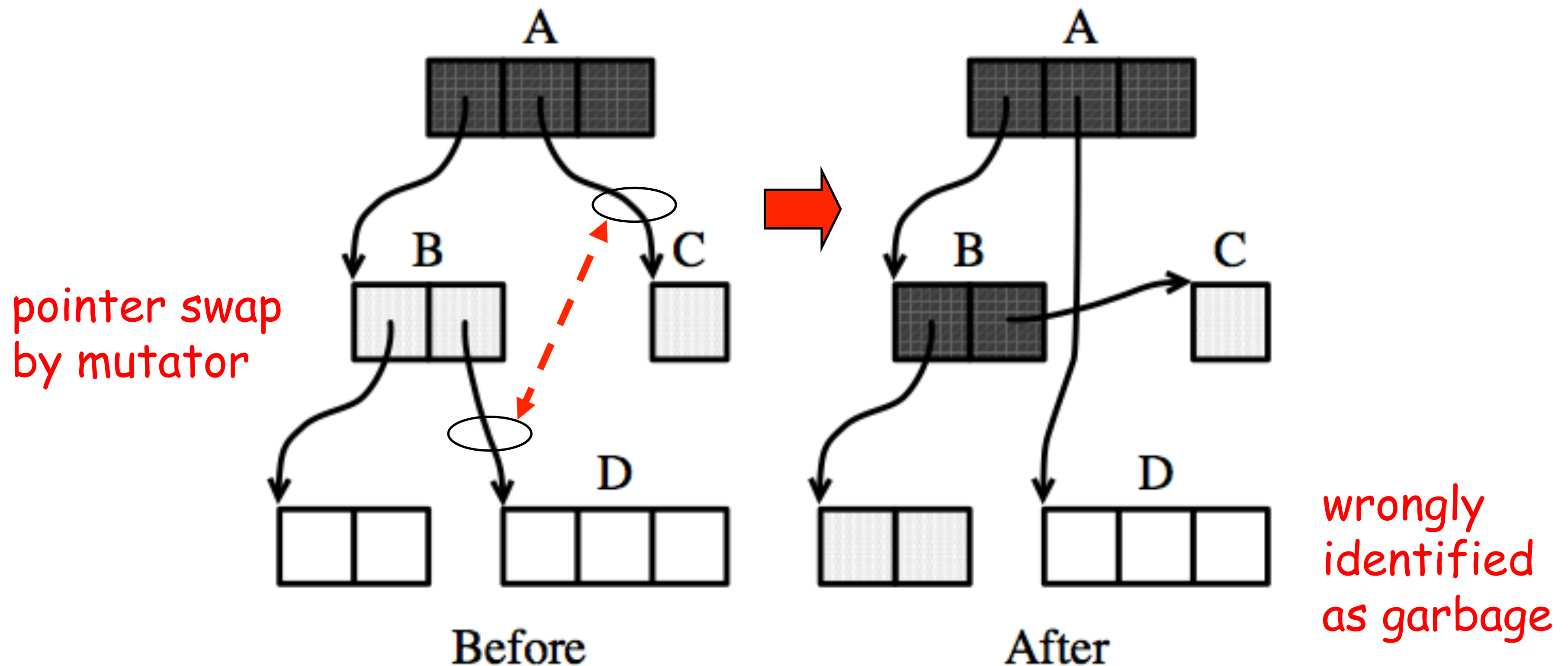
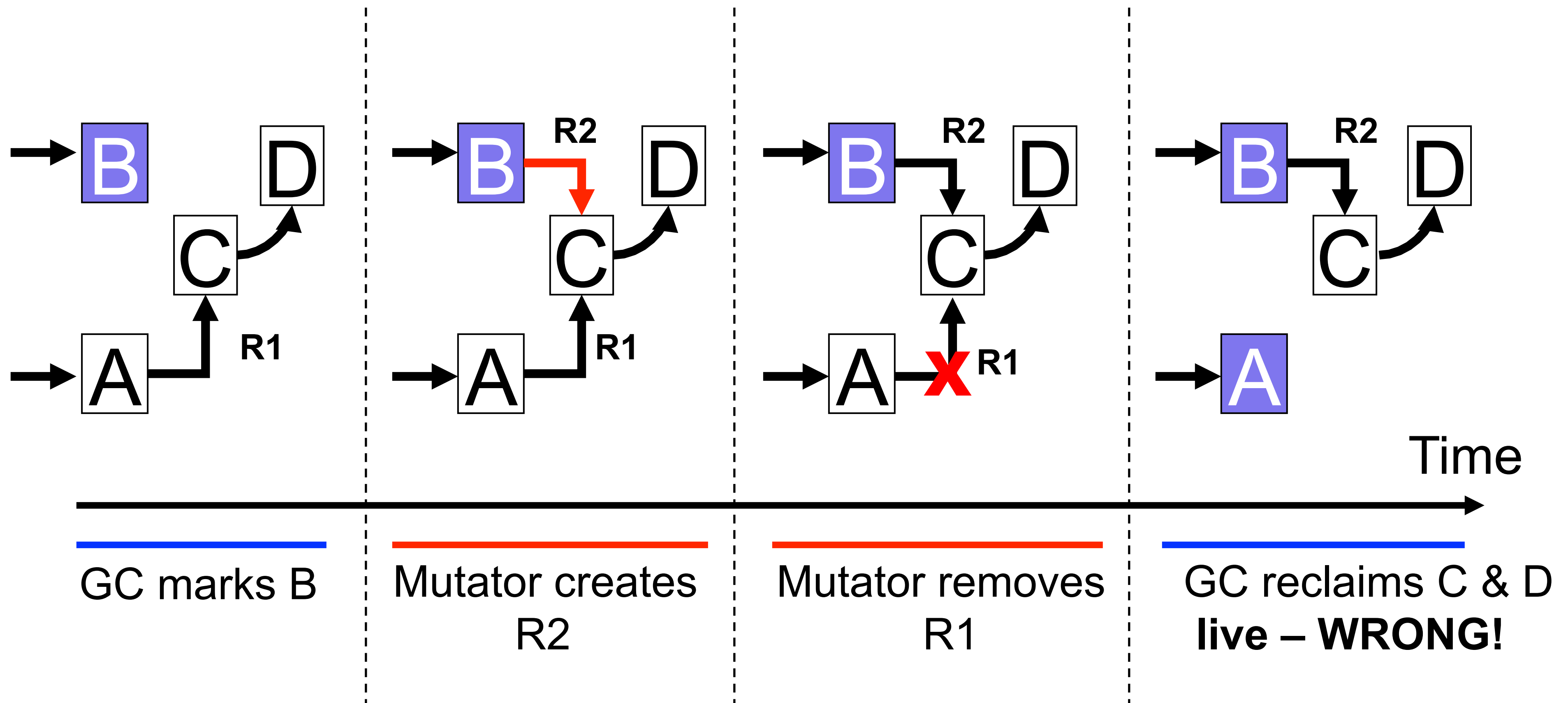


Figure 7: A violation of the coloring invariant.

From Paul R. Wilson: Uniprocessor Garbage Collection Techniques

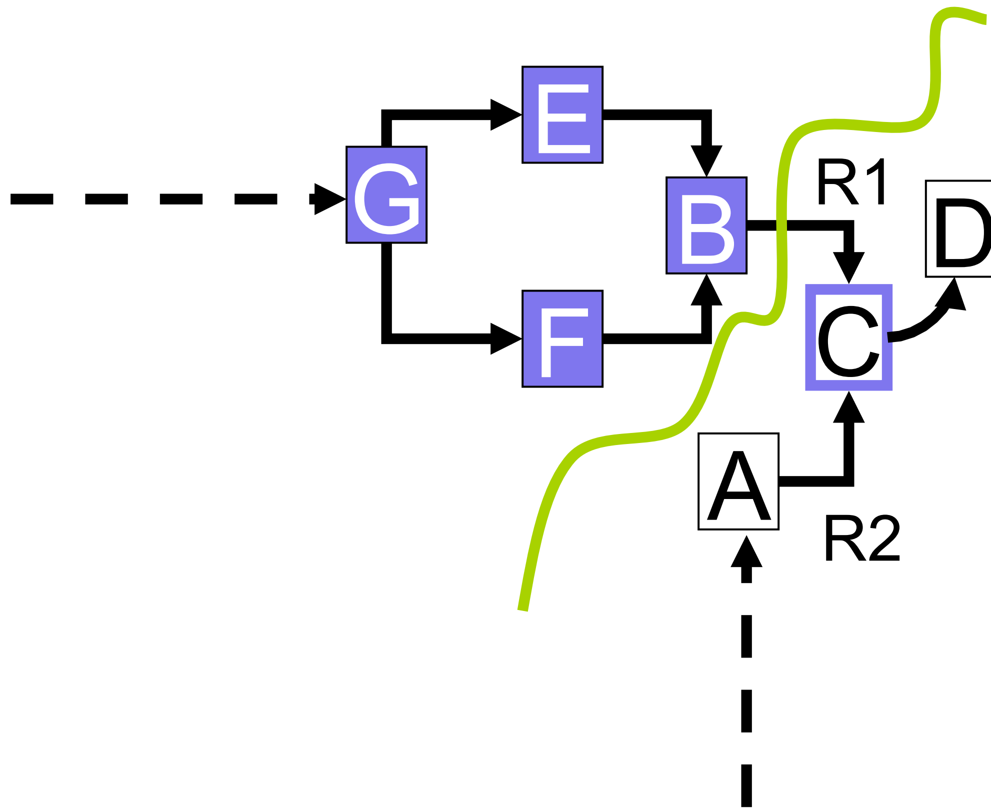
Issue in More Detail



source http://domino.watson.ibm.com/comm/research_people.nsf/pages/bacon.presentations.html/$FILE/Vechev05DerivationTalk.ppt

Marking Wavefront View

"Gray" nodes are those just ahead of a wavefront.

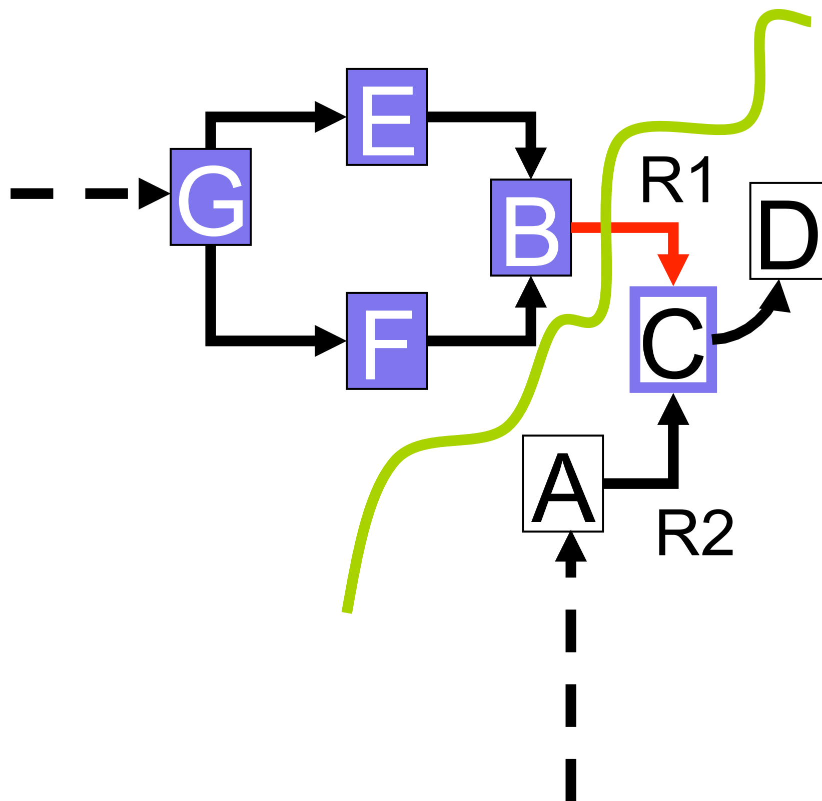


source http://domino.watson.ibm.com/comm/research_people.nsf/pages/bacon.presentations.html/$FILE/Vechev05DerivationTalk.ppt

2 Ways to Resolve Issue

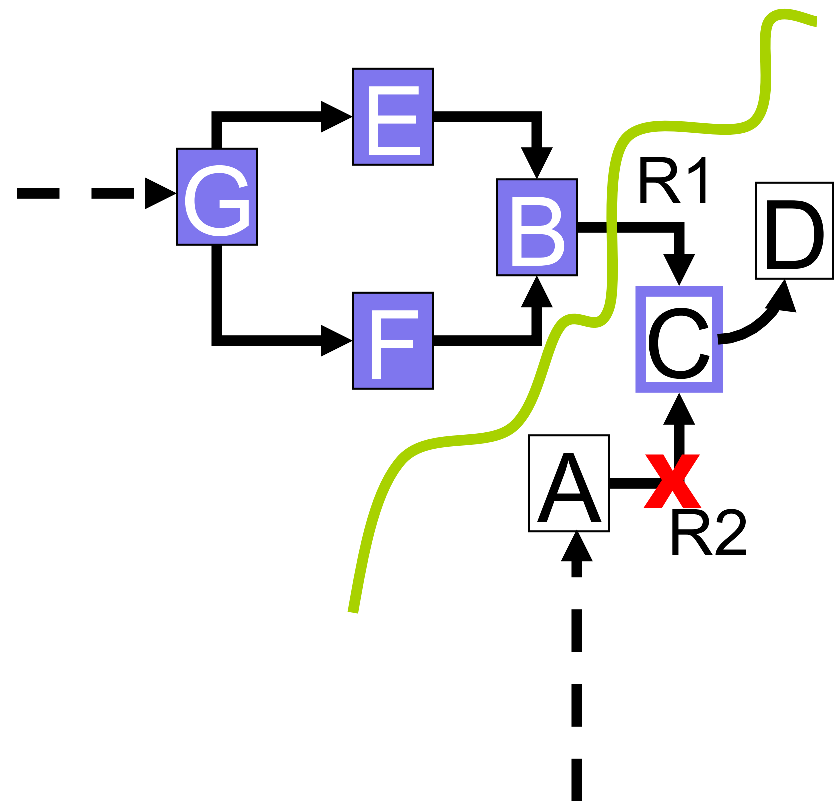
At Pointer Installation

Remember Crossing Pointer R1



At Pointer Deletion

Remember R2



source http://domino.watson.ibm.com/comm/research_people.nsf/pages/bacon.presentations.html/$FILE/Vechev05DerivationTalk.ppt

Dijkstra et al. used the first way

- When the mutator finds a pointer to a node, the node is immediately made gray.
- Said to be a form of "write barrier".

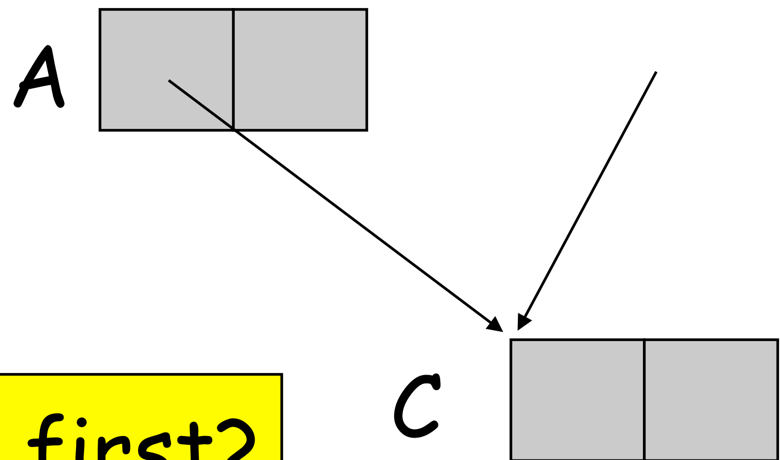
Write-Barrier Implementation for Dijkstra GC (Jones and Lins)

- Define

```
shade(p):
        if white(p)
                color(p) = gray
```

- Making *A* point to *C* is

```
update(A, C):
        *A = C;
        shade(C)
```



Or should shade come first?

Jones and Lins (after Dijkstra, et al.)

- Why shade(C) should come **after** *A = C:
- Suppose instead we had:
  - Define

```
shade(p):
        if white(p)
                color(p) = gray
```

  - Making A point to C

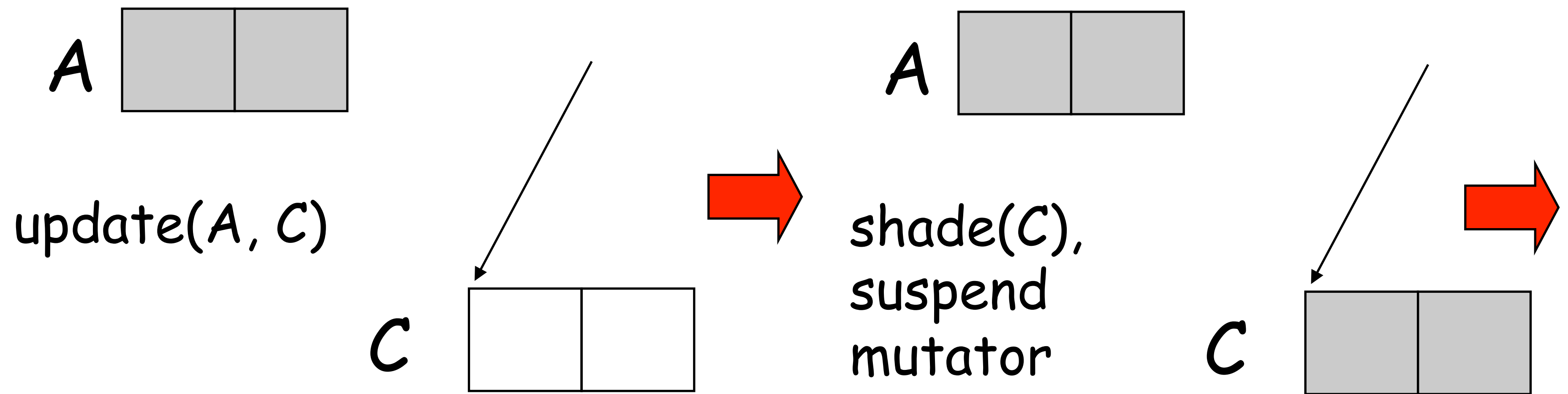
```
update(A, C):
        shade(C)
        *A = C
```

This order may cause non-garbage to be collected!

Example

- Suppose the mutator does update(*A*, *C*) and is **suspended between** shade(*C*) and **A* = *C*.
- Suppose the collector **completes the current cycle**, then starts another cycle (resetting all nodes to white).
- When the collector reaches *A* (which has no children yet), it colors it black.
- Suppose the mutator now continues, writing the pointer to *C* in **A*.
- Now *C* will be collected, wrongly.

Diagram for Shading First



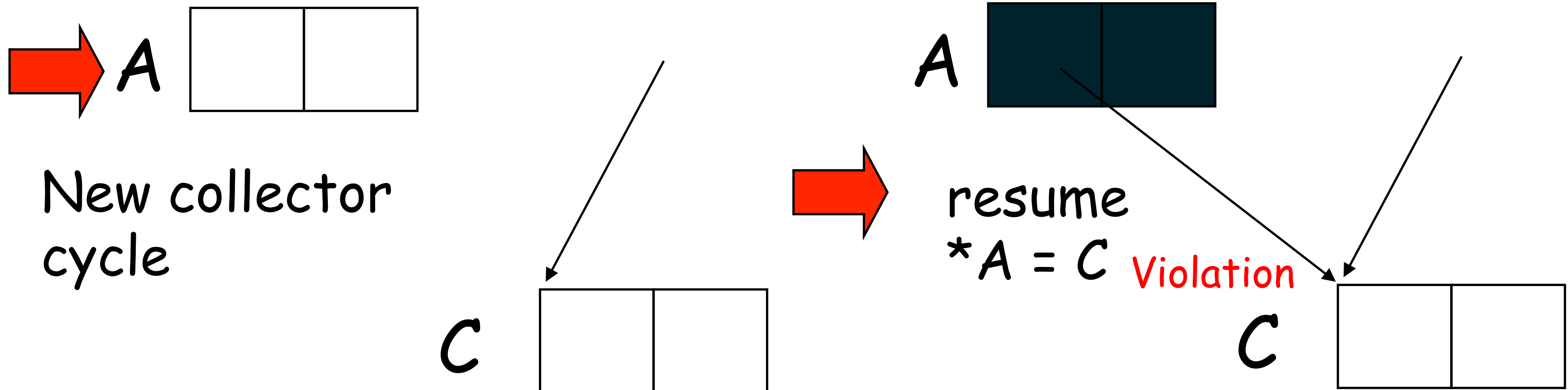
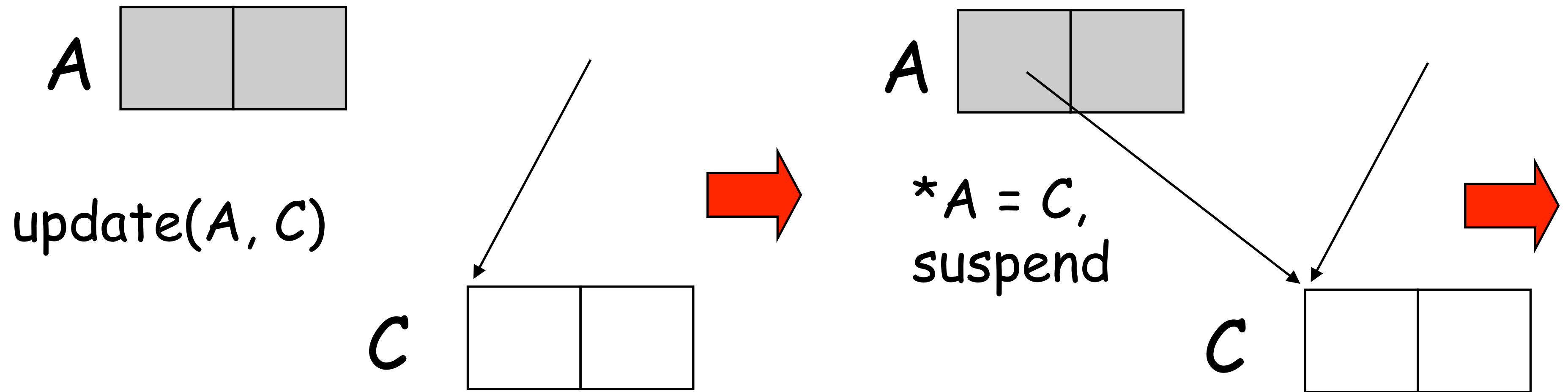
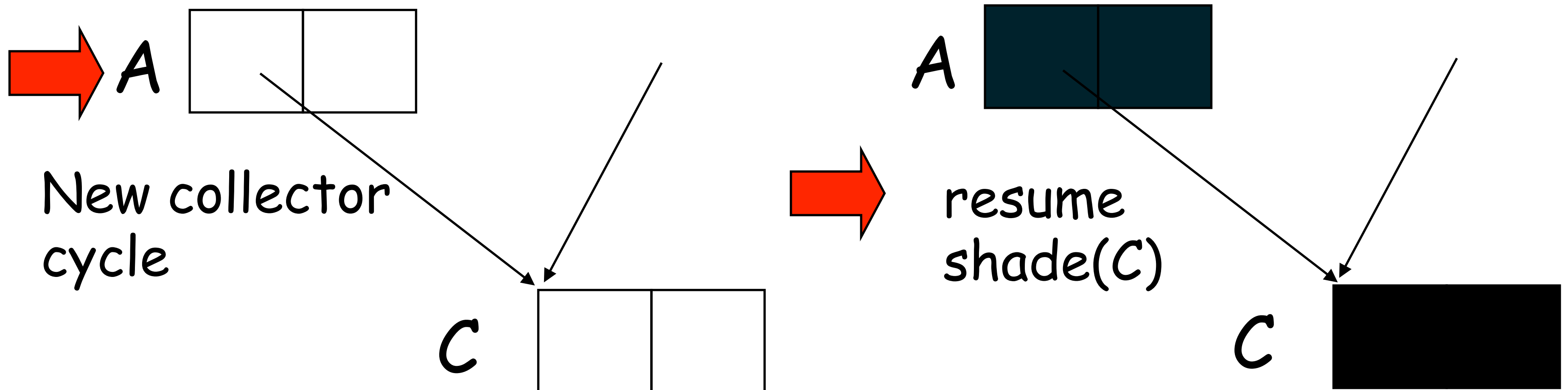


Diagram for Shading Second





Comments by Dijkstra, et al.

Note 1. Disregarding P1, the problem of node C from the counterexample at the beginning of this section could also have been solved by having the mutator shade the old target instead of the new one. This, however, would lead to a solution in which garbage created during a marking phase is guaranteed not to be collected during the next appending phase. Hence, we rejected this solution in accordance with the last sentence of Section 1. □

“Correctness Conditions” by Dijkstra, et al.

The mutator and the collector must cooperate in such a fashion that the following two correctness criteria are satisfied.

CC1: Every garbage node is eventually appended to the free list. More precisely, every garbage node present at the beginning of an appending phase will have been appended by the end of the next appending phase.

CC2: Appending a garbage node to the free list is the collector’s only modification of (the shape of) the data structure.

Specification of GC

- Do those conditions provide a complete specification of a concurrent garbage collector?
- If not, how would we specify it?

Dijkstra et al. "Coarse-Grain" Algorithm

P1: "no edge points from a black node to a white one."

```
marking phase:
begin {there are no black nodes}
   "shade all roots" {P1 and there are no white roots};
   i:= 0; k:= M;
marking cycle:
   do k > 0 → {P1 and there are no white roots}
      ⟨c:= color of node nr. i⟩;
      if c = gray → k:= M;
         C1: ⟨shade the successors of node nr. i and make node nr. i
              black⟩
      [] c ≠ gray → k:= k − 1
      fi;
      i:= (i + 1) mod M
   od
end {P1 and there are no white roots and no gray nodes, hence—as is
   easily seen—all white nodes are garbage};
```

Dijkstra et al. "Coarse-Grain" Algorithm

```
appending phase:
begin i:= 0;
   appending cycle:
      do i < M → {all nodes with a number < i are nonblack; all nodes
            with a number ≥ i are nongray, and are garbage, if white}
         ⟨c:= color of node nr. i⟩;
         if c = white → ⟨append node nr. i to the free list⟩
         [] c = black → ⟨make node nr. i white⟩
         fi;
         i:= i + 1
      od {there are no black nodes}
end
```

Proof of the Coarse-Grain Solution

- An extended informal argument is given in the paper to show the two correctness conditions (CC1 and CC2) are satisfied.

Problem with Coarse-Grain

- It repeatedly resets the marking ($k := M$), which is impractical.

An attempt at finer grain introduced the shading-order bug described earlier.

ourselves—contained the following bug, discovered by N. Stenning and M. Woodger [5].

Consider the following sequence of events:

1. Prior to introducing an edge from node *A* to node *B*, the mutator shades node *B* (and goes to sleep)
2. The collector goes through a complete marking phase, followed by an appending phase (node *B* is now white, i.e. the mutator's shading has been undone! We further note that there is no garbage)
3. The collector goes through part of the next marking phase (and then goes to sleep), during which it so happens that node *A* is made black and node *B* is left white
4. The mutator (wakes up and) introduces without making garbage the edge from *A* to *B* (P1 is now violated)
5. The mutator removes all other ingoing edges of *B*—the absence of garbage makes this possible—and goes to sleep again (node *B* is now only reachable via the edge from *A*)
6. The collector completes its marking phase (node B has remained white)
7. The collector goes through its appending phase, during which the reachable node *B* is erroneously appended to the free list.

Revising the Coarse-Grain Solution

This ill-fated effort convinced us that in the finer-grained solution we were heading for, total absence of an edge from a black node to a white one was a stronger relation than we could maintain. However, it still seemed reasonable to retain the notion of "gray" as "semi-marked," more precisely, as representing an unfulfilled marking obligation. This meant that we could use the same collector. However, we had to find a different coarse-grained mutator that we could use as a stepping stone to our ultimate fine-grained solution.

Fine-Grained Mutator

M2: ⟨shade the target of the previously redirected edge, and redirect an outgoing edge of a reachable node towards a reachable node⟩.

For our fine-grained mutator, M2 is split up into the following succession of atomic operations:

M2.1: ⟨shade the target of the previously redirected edge⟩;

M2.2: ⟨redirect an outgoing edge of a reachable node towards a reachable node⟩

Fine-Grained Collector

C1: ⟨shade the successors of node nr. i and make node nr. i black⟩

In the collector, we break open C1 as the sequence of five atomic operations ($m1$ and $m2$ being local variables of the collector):

C1.1: ⟨$m1$:= number of the left-hand successor of node nr. i⟩;
C1.2: ⟨shade node nr. $m1$⟩;
C1.3: ⟨$m2$:= number of the right-hand successor of node nr. i⟩;
C1.4: ⟨shade node nr. $m2$⟩;
C1.5: ⟨make node nr. i black⟩.

Fine-Grained Collector Re-abstracted

C1: ⟨shade the successors of node nr. i and make node nr. i black⟩

We first observe that the collector's action of shading a node commutes with any number of mutator actions M2.1 and M2.2; without loss of generality we can, therefore, continue our discussion as if the four atomic operations C1.1 through C1.4 were replaced by a succession of the following two atomic operations:

C1.1a: ⟨shade the left-hand successor of node nr. i⟩;
C1.3a: ⟨shade the right-hand successor of node nr. i⟩.

Detail Omitted Here

- A series of similar simplifications of both code and invariants follows.
- See the glossary of the paper for these.
- By the time we are done with the informal arguments, it's really not clear what we have.

Alternate: Read barrier

- A read-barrier solution is to never let the mutator hold a pointer to a white node; the node is **shaded before** it gets the pointer.

vs.

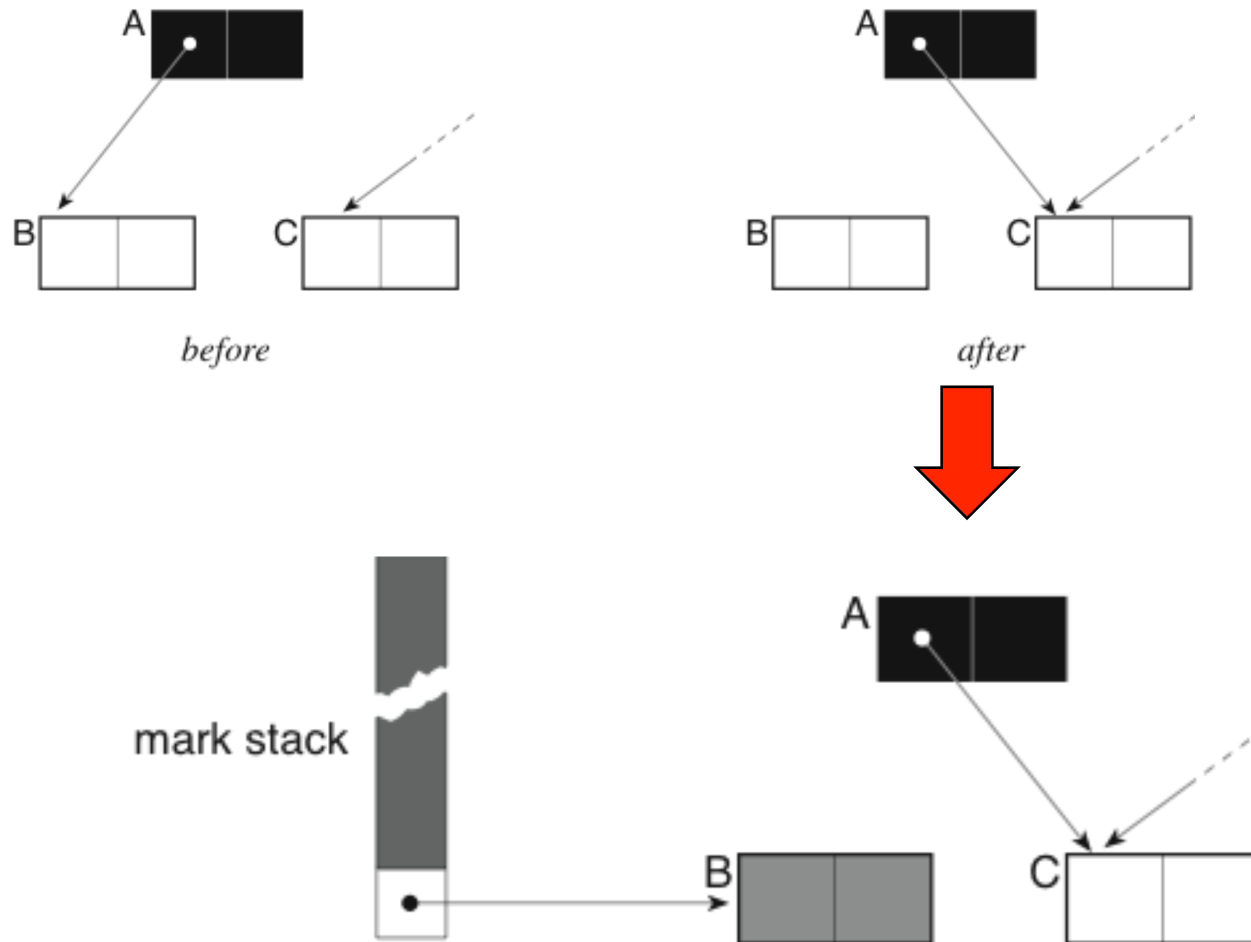
- write-barrier: Shade the node when a pointer to it is first placed.

- The read-barrier approach is used in the Steele paper.

Yuasa's "Snapshot" Method

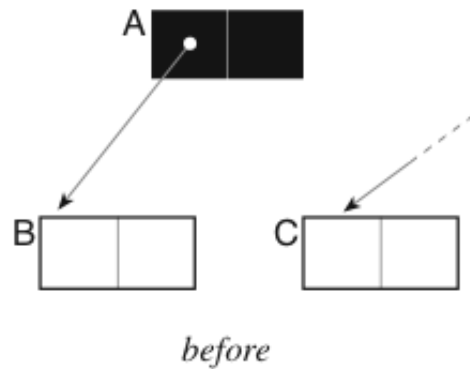
- Called "snapshot" because it records situation before a pointer is modified.
- Write-barrier traps pointer updates during marking.
- Shades old white pointer "gray" implicitly by setting a single mark bit and pushing a reference to the old cell onto a marking stack.
- Conservative: Can't collect an object in the same cycle that it becomes garbage.

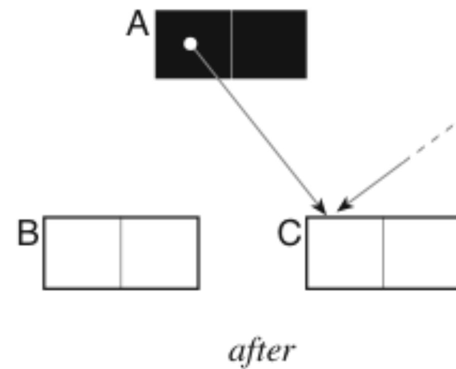
Yuasa's "Snapshot" Method

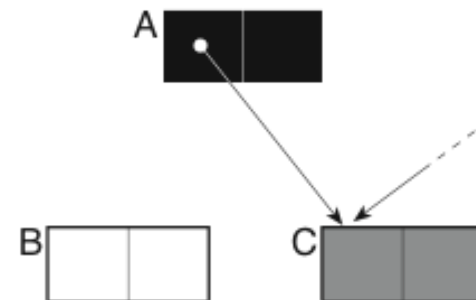


Figures from Jones & Lins

Compare Dijkstra, et al. Method
(mutator shades cell when pointer set to it)



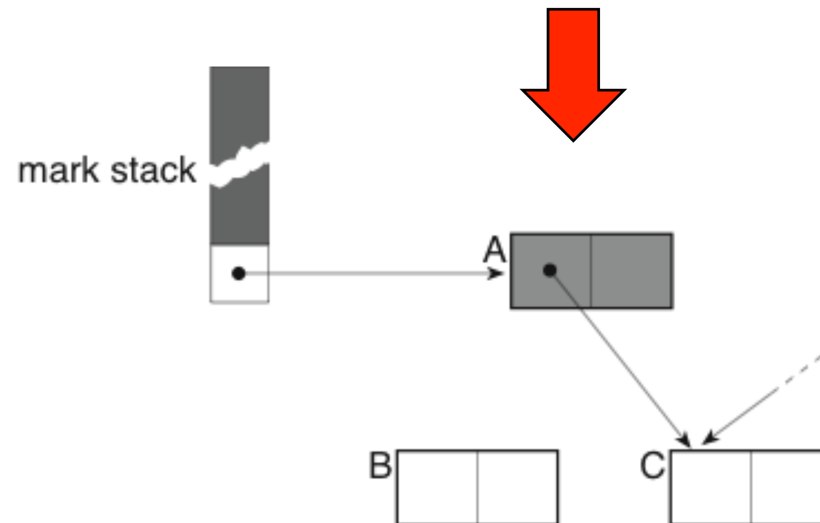




Figures from Jones & Lins

Compare Steele's Method
(retreat the wavefront when setting pointer)

This method may cause A to be revisited, but ends up with less floating garbage at the end of a cycle.



Figures from Jones & Lins

The tri-color idea stuck.

- Following the Dijkstra et al paper, the tri-coloration idea continues to be used on all sorts of algorithms, to mean something similar.
- The “gray” idea is even applied in explaining earlier methods, even when gray is not an explicit color.
- Sometimes more colors are added.

Baker's Real-Time GC

Communications of the ACM **21**, 4 (April 1978), 280-294.

List Processing in Real Time on a Serial Computer

Henry G. Baker, Jr.
Massachusetts Institute of Technology

Baker's Real-Time GC

- Extends Cheney's compacting method.
- Allocate new objects in ***tospace***.
- Copy any accessed object in ***fromspace*** to ***tospace*** when it is accessed, leaving a forwarding address. Also, revise the pointer to point to the new location.
- ***Scavenging*** is used to complete the ***evacuation*** of ***fromspace***.
- Rather than being done all at once, scavenging is done ***a few steps at a time***, interleaved with the memory allocation operation. This is the **real-time** aspect.

Scavenging

- ***Scavenging*** is a process that linearly scans the evacuation area of ***tospace***; if a component of an object points to ***fromspace***, the ***fromspace*** object is evacuated to ***tospace*** (appended to the evacuation area). Like the mark phase of traditional garbage collectors, scavenging touches all accessible objects. It does so in breadth-first order and does not require a stack.

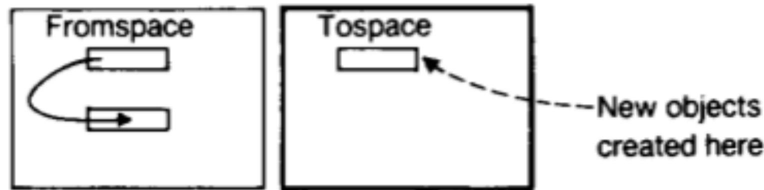
- The *scavenger* process can be interleaved with object creation, evacuating a few ***fromspace*** objects to ***tospace*** every time an object is created.

- Since only a small amount of work must be done whenever an object is created or parts of an object are accessed, the garbage collection operates in real time.

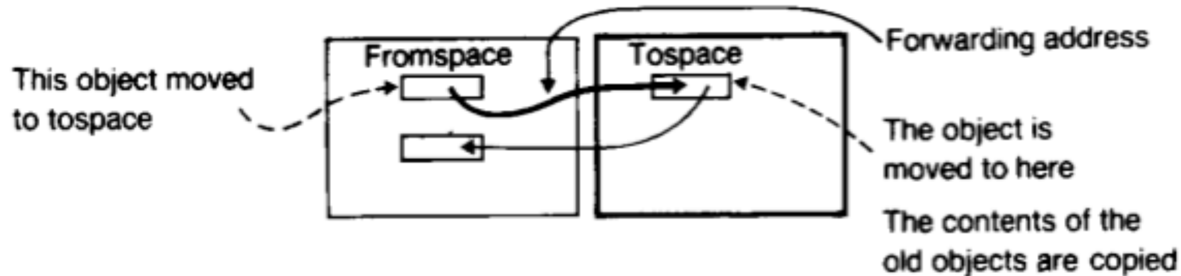
source: Lieberman and Hewitt, 1983 CACM

Baker's Real-Time GC

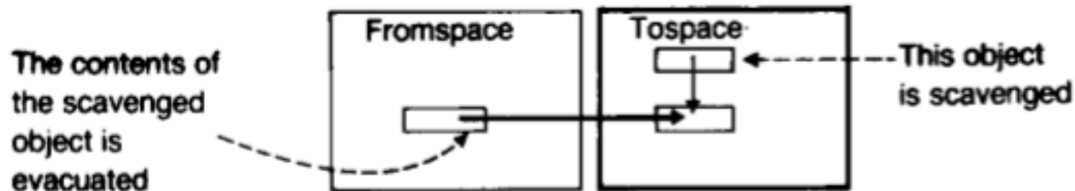
(figure from Lieberman and Hewitt, 1983 CACM)



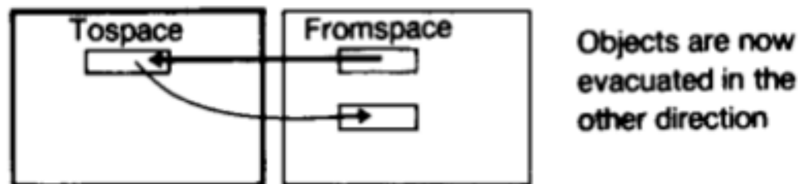
Memory is divided into fromspace and tospace



Evacuating an object moves it from fromspace to tospace

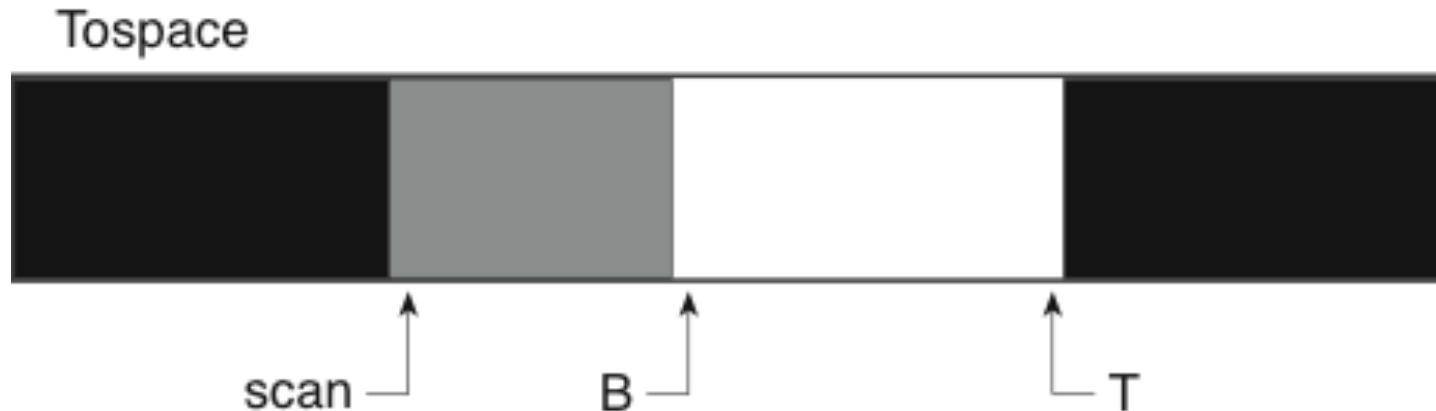


Scavenging an object removes pointers to fromspace



After a flip, fromspace and tospace are exchanged

Baker ToSpace in 3 Colors



- The scavenging region is between scan and B.
- Copying is done at B.
- Allocation is done at T.

Figure source:
Jones & Lins,
Garbage Collection,
Wiley, 1996

A Baker Disadvantage

- No cell can be reclaimed until the cycle *following* its death.

Use of Baker in Multiprocessing

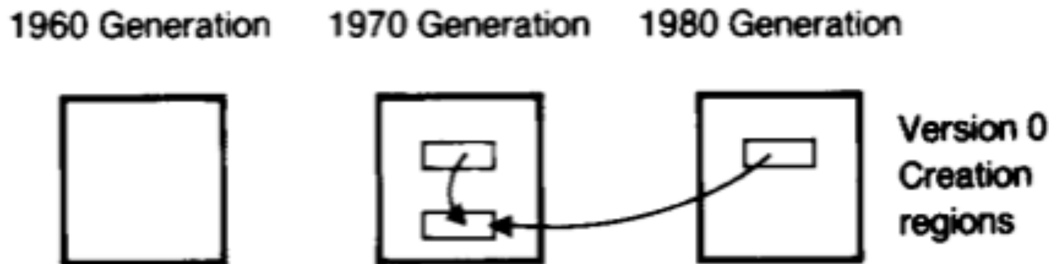
- Concert Multilisp (Robert Halstead) used a variant of Baker.

Lieberman & Hewitt: Generational Real-Time

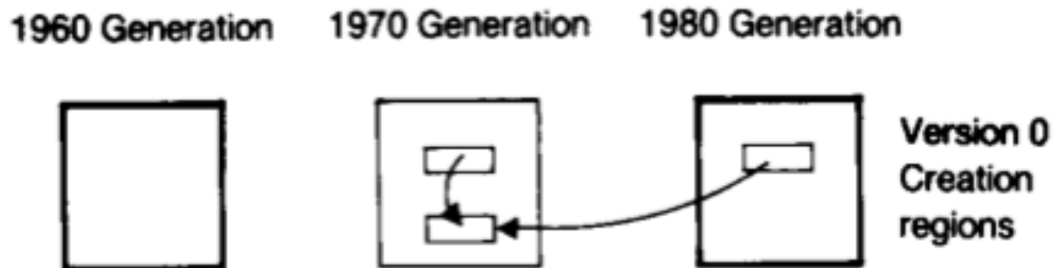
- Extends Baker space dichotomy to multiple regions (generations).
- A generation can be **condemned**, meaning that all objects are going to be evacuated from it.
- Once the condemned generation is completely evacuated, the space is recycled.

Lieberman and Hewitt Real-Time GC

(figure from Lieberman and Hewitt, 1983 CACM)



Memory is allocated in small regions
Regions are tagged with generation and version numbers

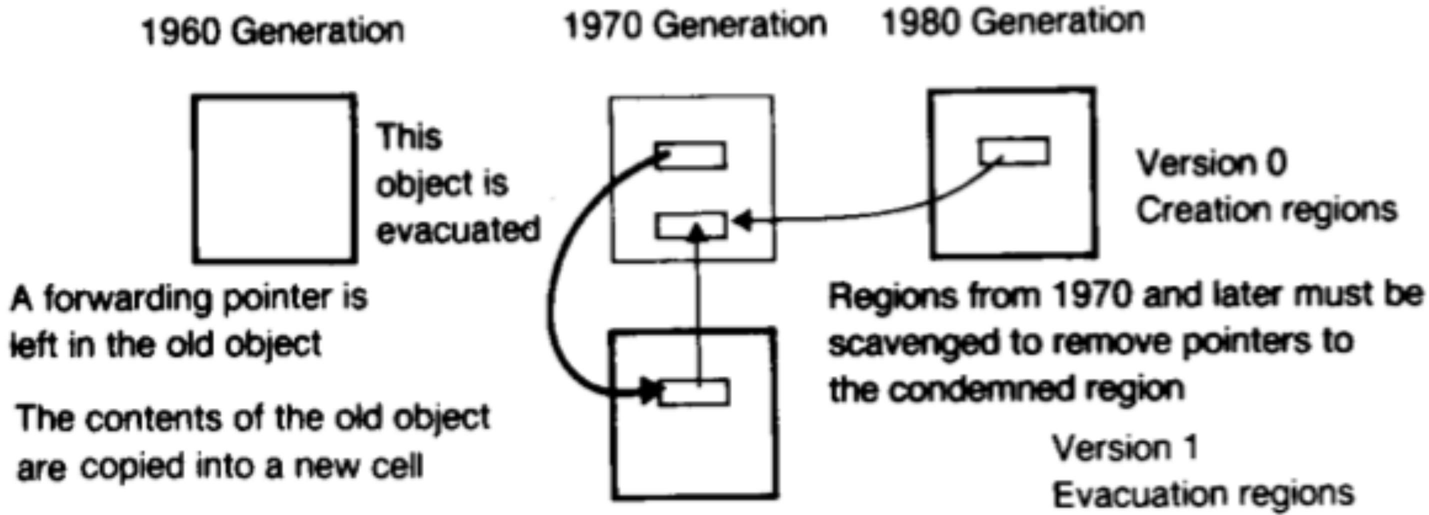


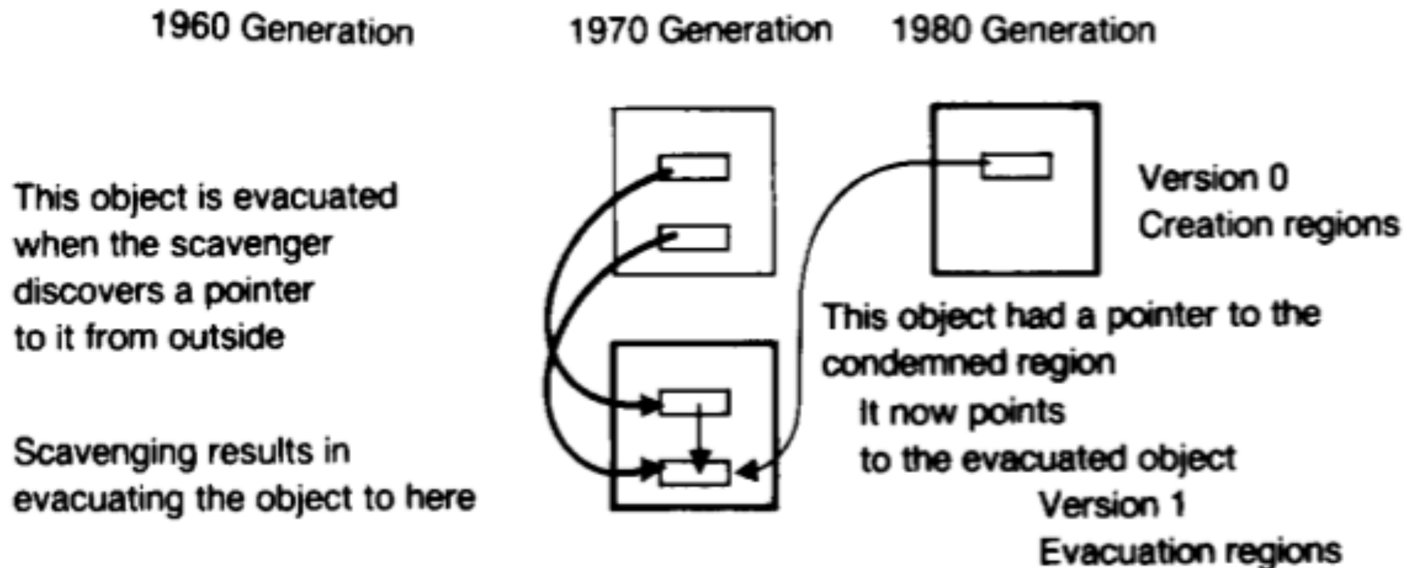
This region is condemned

All accessible objects from the condemned region will be moved to here

Version 1
Evacuation
regions

Garbage collecting a region is initiated by condemning it
Accessible objects from the condemned region
will be evacuated to a new region





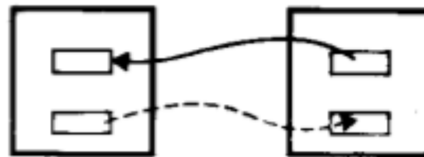
Evacuation and Scavenging in Lieberman and Hewitt Real-Time GC

(figure from Lieberman and Hewitt, 1983 CACM)

Pointers "Forward in Time"

- *cons* operations create objects that contain pointers to pre-existing objects , thus they point "backward in time".
- rplaca & rplacd operations may cause pointers "forward in time", since they modify an existing objects to have a pointer to *any* object.
- The latter pointers complicate generational collection.

1960 Generation 1970 Generation

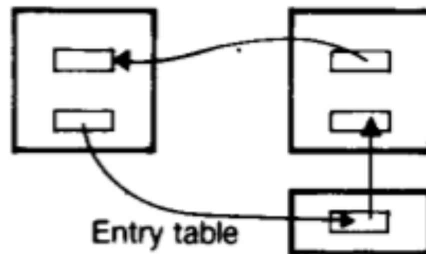


Pointers may point back any numbers of generations

Pointers may not point directly from older to younger objects. This pointer would not be allowed.

An object in 1960 may not point directly to an object in 1970

1960 Generation 1970 Generation

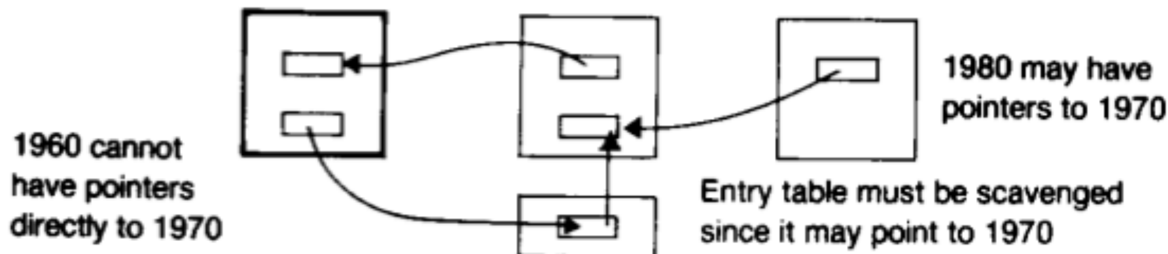


Entry table

All pointers to 1970 from earlier generations go indirectly through this entry table

Pointers forward in time go indirectly through entry tables

1960 Generation 1970 Generation 1980 Generation



1960 cannot have pointers directly to 1970

1980 may have pointers to 1970

Entry table must be scavenged since it may point to 1970

When 1970 is condemned, 1980 must be scavenged, but not 1960

Lieberman and Hewitt solve the forward-in-time problem by means of **Entry Tables.**

(figure from Lieberman and Hewitt, 1983 CACM)

Reclaiming Entry Tables

“How do we recover storage in the entry table when a pointer from an older to a younger object becomes inaccessible? Since we expect there to be a relatively small number of forward pointers, efficiency of storage management for entry tables is not as critical an issue as it is for objects. There are several alternatives, and here we present a suggestion of Lucassen's [21]: if we record the name of the region of the originating object with each entry in the entry table, we have a means of detecting inaccessible pointers in the entry table. When the system completes garbage collection and scavenging for a region, it is known that all objects in the region are inaccessible, and the system records the region in a list. When looking at entry tables, any cell created for an object in an inaccessible region is known to be inaccessible. This requires that region names are unique, which is not hard to assure, and also that entries are not shared, since every forward pointer gets its own entry.”

Lieberman & Hewitt

- Refer to the original paper for more details and insights:

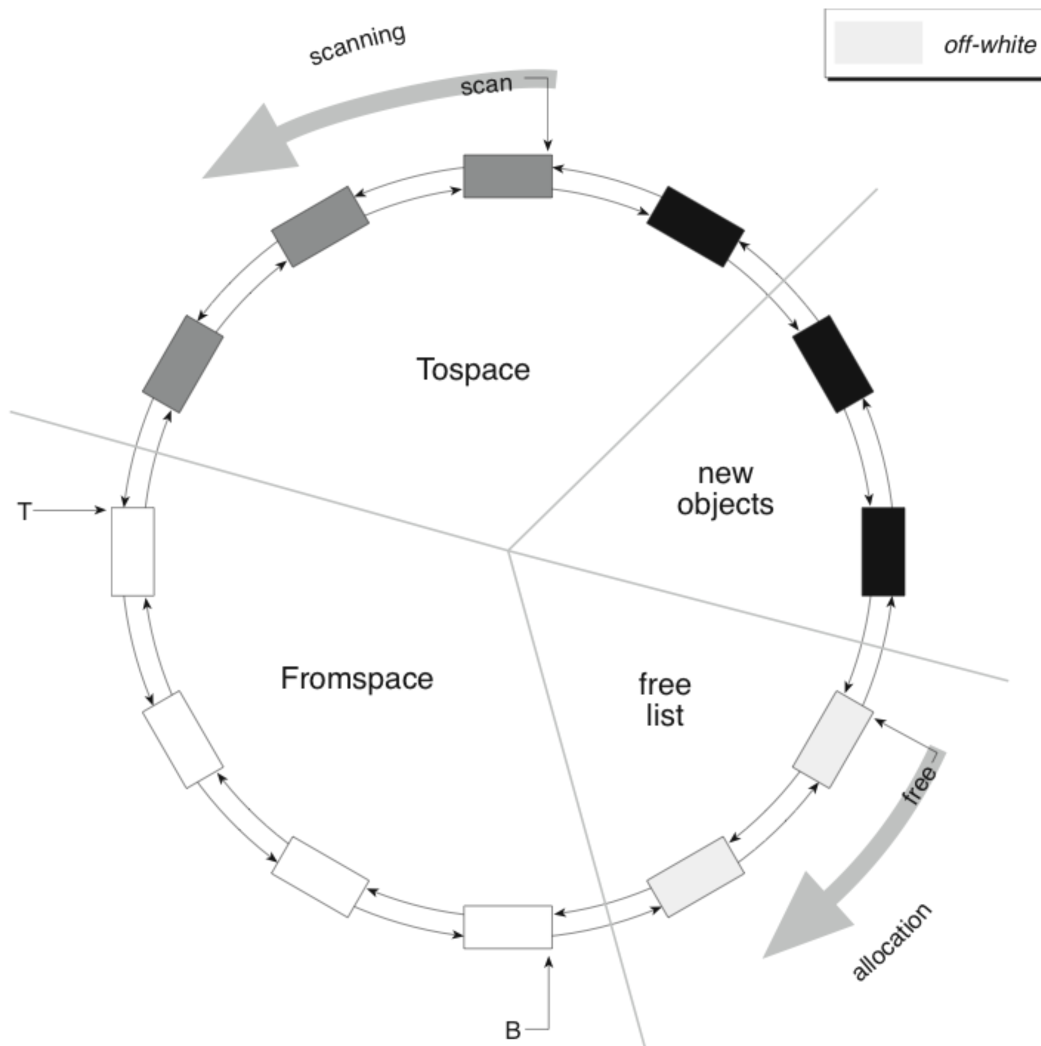
  A Real-Time Garbage Collector Based on the Lifetimes of Objects

  CACM, June 1983

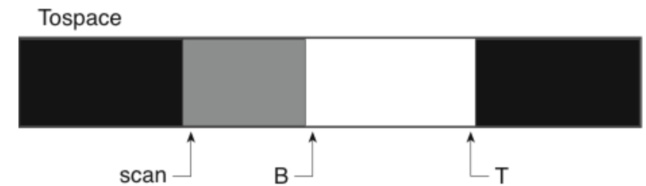
Baker's Treadmill Algorithm

- Non-moving collector
- Objects have a doubly-linked **infrastructure** (in addition to pointers in the objects proper).
- One more color: **off-white** = on free list.

Baker's Treadmill Algorithm



compare original Baker algorithm



Source:
Jones & Lins,
Garbage Collection,
Wiley, 1996

Baker's Treadmill Algorithm

- Scanning on the treadmill:
  - If scanned pointer refers to black or gray object, no action.
  - If it points to white, then the object is **unlinked** and moved to the gray section (at either end).
  - This is the only "color change".
- When B meets T, spaces are flipped.

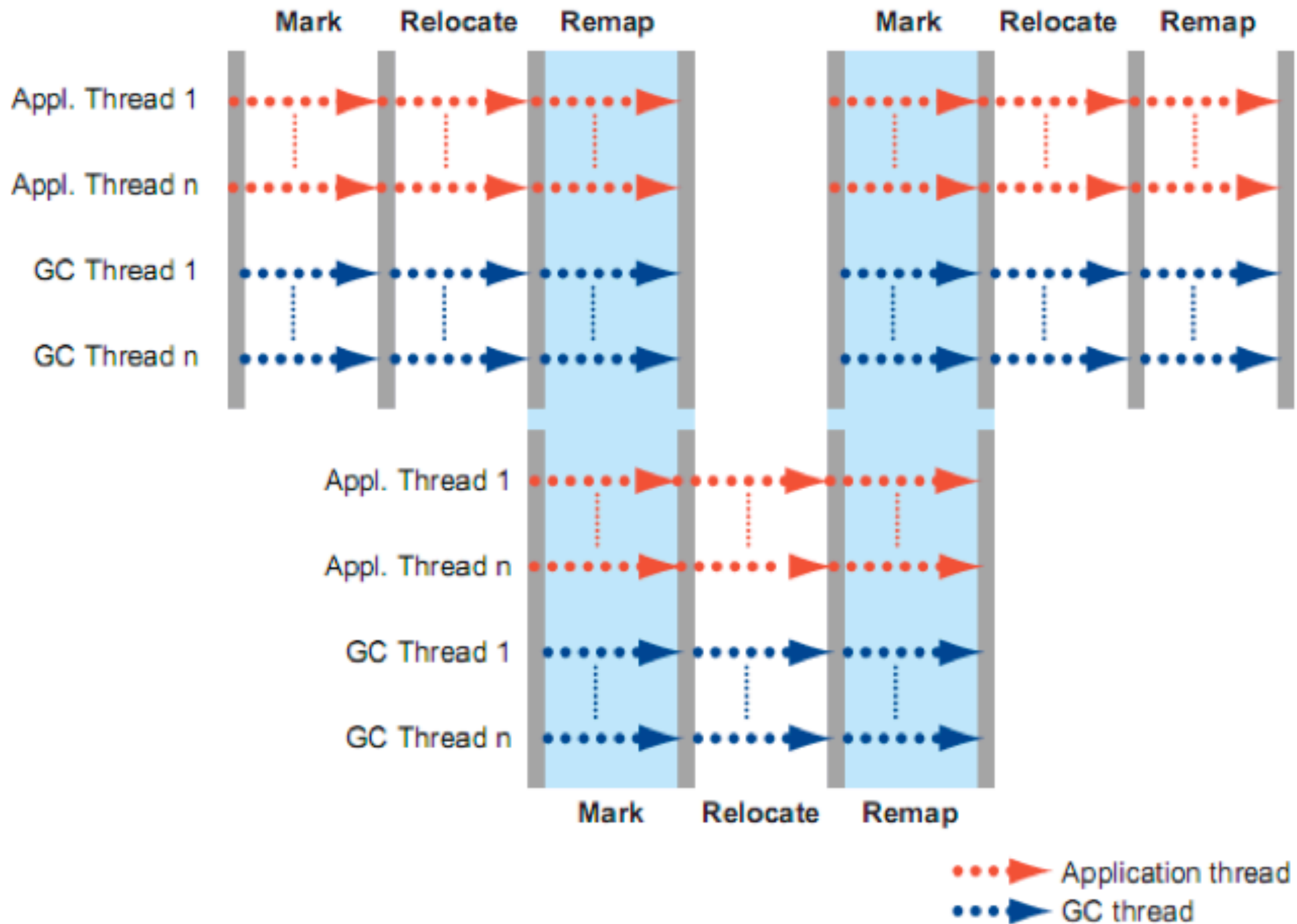
Other GC Considerations

- Interaction / Exploitation of Virtual Memory and Paging
- Fault Tolerance
- Distributed Systems Approaches

Azul "Pauseless" Concurrent Garbage Collector

- http://www.azulsystems.com/zing/pgc
  (Caution: Some marketese ahead)
- Designed for the Zing Java VM
- Phases:
  - **Mark**: Periodically refresh mark bits.
  - **Relocate**: Using most recent mark bits, relocate and compact data on sparsely-populated pages.
  - **Remap**: Update pointers to relocated objects.

Azul GC



Azul Marking

- Hardware emulated read barriers intercept attempts by the application to use pointers that have not yet been seen by the garbage collector.
- The barrier logs such pointers to assure the collector visits them, tags the pointers as “seen by the collector”, and continues the application’s execution without a pause.

Azul Marking

- The mark phase marks all live objects in the Java heap concurrently with execution of application threads.
- In addition to marking all live objects, the mark phase maintains a count of the **total live memory** in each memory page (each page is currently 1MB). **This information is later used by the relocation phase to select pages for relocation and compaction**.
- The algorithm tracks the object references that have been traversed by using an architecturally **reserved bit in each 64 bit object reference**. This bit, called the "not marked through" (NMT) bit, designates an object reference as either "marked through" or "not marked through".
- The marking algorithm recursively loops through a working list of object references and finds all objects reachable from the list. As it traverses the list, it marks all reachable objects as "live", and marks each traversed object reference as having been "marked through".

Use of NMT
(Not-Marked-Through) Bit

- The Zing JVM includes a read barrier instruction contained in hardware emulation software that is aware of the NMT bit's function, and **ensures application threads will never see an object reference that was not visited by the marker**.

- The NMT bit is tested by the read barrier instruction that all application threads use when reading Java references from the memory heap. If the NMT bit does not match an expected value of "marked through" for the current GC cycle, the processor generates a fast GC trap.

- The trap code **corrects the cause** of the trap by queuing the reference to the marker's work list, setting the NMT bit to "marked through", and correcting the source memory locationfrom which the object reference was loaded, to ensure that it too includes the "marked through" indication.

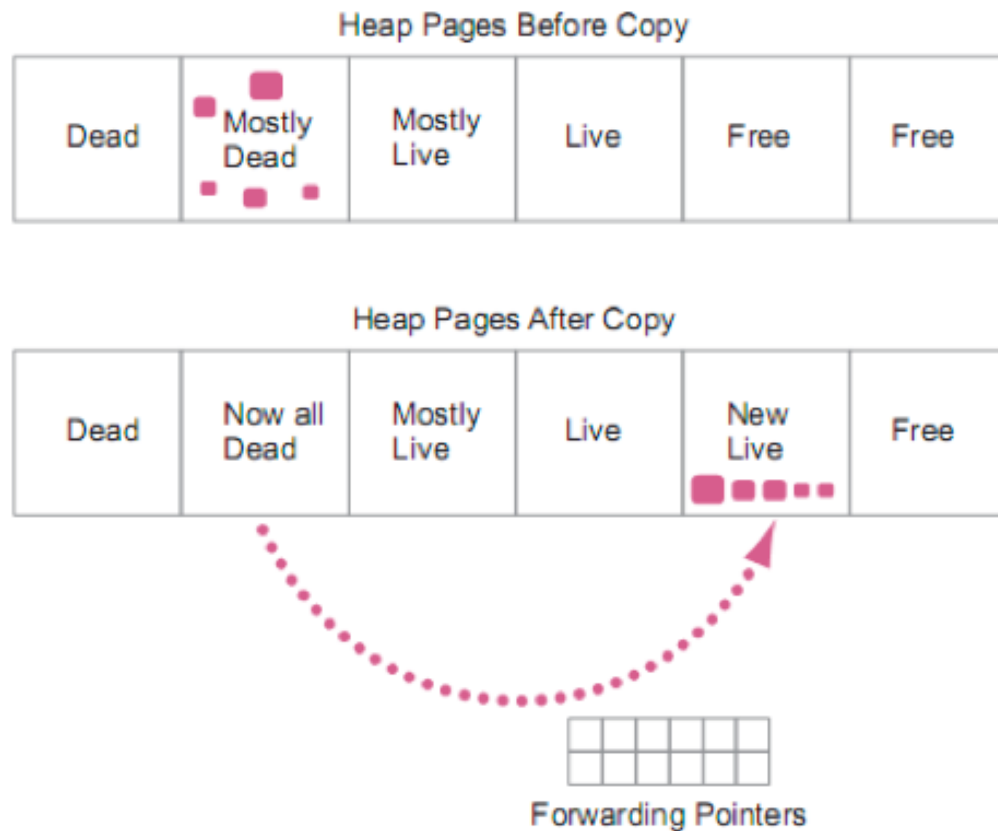
Use of NMT
(Not-Marked-Through) Bit

- Following the trap, the machine returns to normal application thread execution, with the application observing only the safely "marked through" reference.
- By using the read barrier and fast trapping mechanism, the marker is assured of safe marking in a single pass, eliminating the possibility of the application threads causing any live references to escape its reach.
- By correcting the cause of the trap in the source memory location (possible only with a read barrier that **intercepts the source address)**, the GC trap has a "self-healing" effect since the same object references will not re-trigger additional GC traps. This ensures a finite and predictable amount of work in a mark phase.

Azul Relocation Phase

- The relocate phase reclaims Java heap space occupied by dead objects while compacting the heap.
- The relocation phase selects memory pages that contain mostly dead objects, reclaims the memory space occupied by them, and relocates any live objects to newly allocated compacted areas.
- As live objects are relocated from sparse pages, their physical memory resources can be immediately reclaimed and used for new allocations.
- The algorithm **requires only one empty memory page** to compact the entire heap.

Azul Relocation



Azul Relocation

- The GC protects "from" pages, and uses the LVB to support **lazy remapping** by triggering on access to references on "from" pages.
- The GC relocates any live objects to newly allocated "to" pages.
- The algorithm maintains **forwarding pointers** outside of the "from" pages.
- Physical memory is immediately reclaimed for use in further compaction or allocation, but virtual "from" space cannot be recycled until all references to allocated objects are remapped

Azul Generational Aspect

- Central to the algorithm is a **Loaded Value Barrier** (LVB).
- Every Java reference is verified as "sane" when loaded. (A "sane" reference has the correct behavior and state.)
- Any "non-sane" references are fixed in a self-healing barrier.
- References that have not yet been marked through, or that point to relocated objects, are caught.
- This allows a guaranteed single pass concurrent marker and the ability to lazily and concurrently remap references.

Pauseless Aspect

- The Azul garbage collection algorithm actually requires zero pauses in applications.
- The JVM implementation itself includes brief pauses at phase transitions. These pauses are well below the threshold that would be noticeable to users and are not affected by application scale or data set size.
- The Azul algorithm completely decouples pause time from memory heap size, allowing applications to scale and use large amounts of heap memory without impacting performance or user response time.

Azul Concurrent Relocation

- Azul uses hardware emulated barriers to intercept any attempts by the application to use stale pointers referring to old locations of relocated objects.
- It then fixes those stale object pointers as they are encountered such that they point to the correct location without pausing the application.

Products to Watch

- Azul garbage collector
- Metronome garbage collector (IBM)