Bounding Pause Times in a Regional Garbage Collector

Felix S Klock II

Thesis Advisor: Will Clinger

What is Garbage Collection?

- Automated reclamation of unreachable storage
- (Tracing) Garbage collection
- *Mutator*: Main application apart from collector

Say: “Tracing GC finds connected component of the directed object graph that includes the program registers (that is, the roots)”

[[(Alternative techniques, but probably shouldn’t mention them explicitly: reference counting; static region+effect systems)]]

Thesis

- Our regional garbage collector has *provable worst case* bounds on pause times, space usage, and mutator utilization, and it also achieves high throughput if provided a spare concurrent task.
- (and there's an additional bonus!)

We've made a new design, that we call "regional GC". My thesis is...
BONUS: designed to be adopted in existing runtime systems; compiler implementors and low-level library writers do not need to know more about the collector than they already do.

Outline

- Review of garbage collection & existing technology
- Essential structure
  - Problem (plus solution)
  - Ensuring completeness
- Worst case bounds
- Empirical results

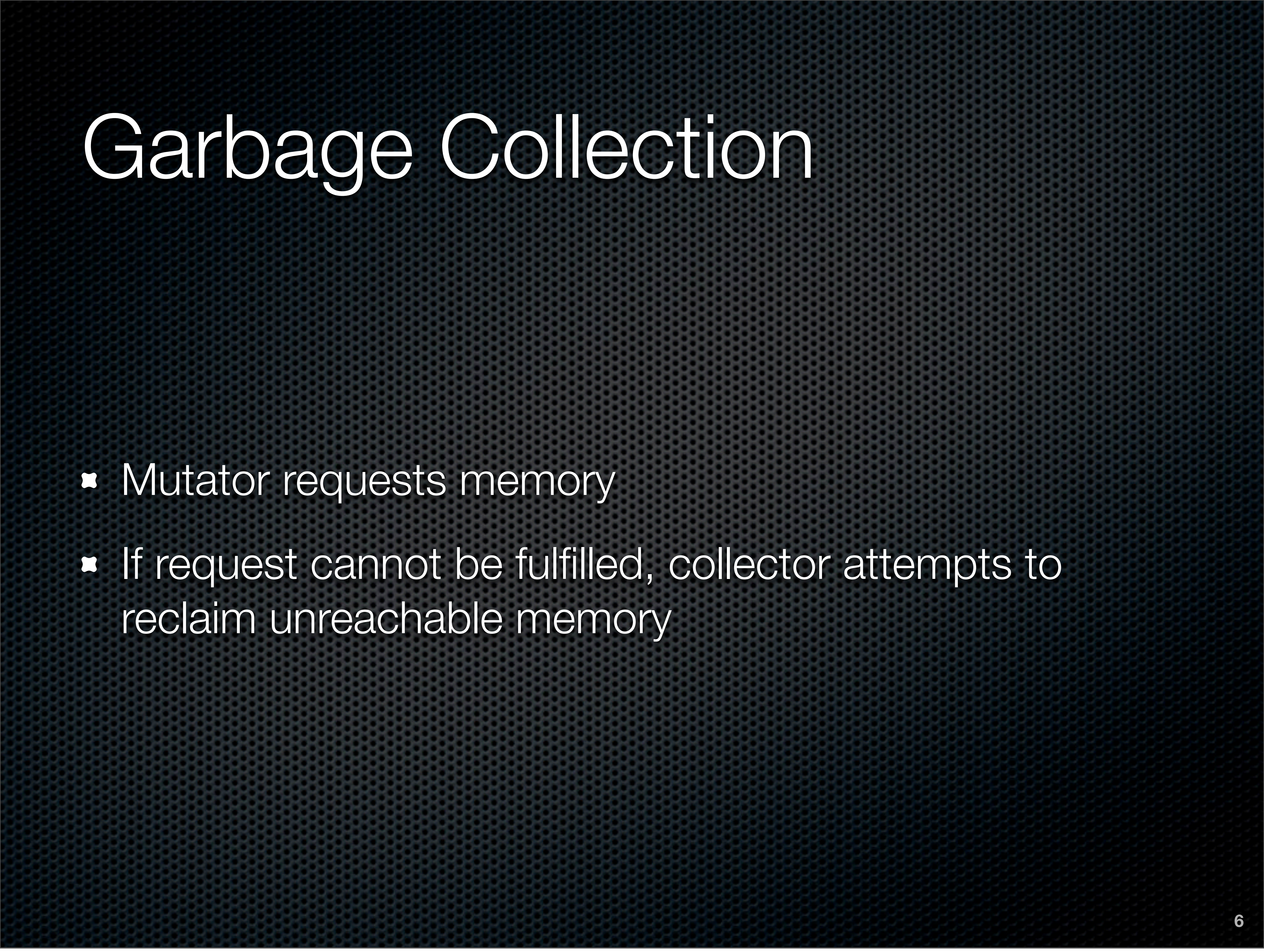
I will be comparing current tech against the regional GC
“Essential Structure” of Regional Collector
Completeness means GC eventually reclaims all unreachable storage.

Why Garbage Collect?

- Reduces programming effort
- No dangling pointers
- Simplifies component interfaces

- Do not want to program in C or C++; would prefer ...

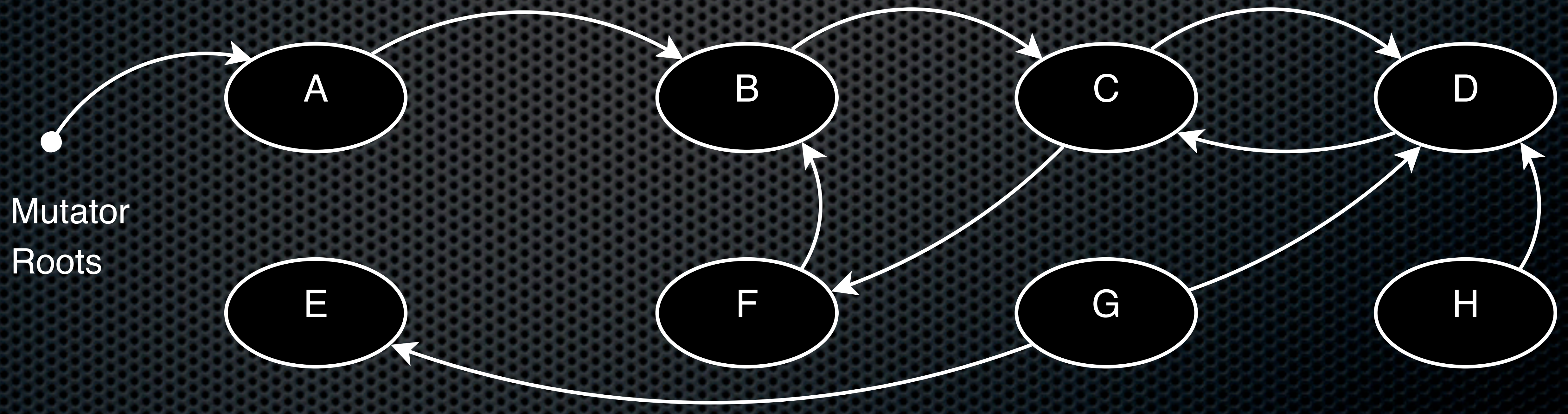
You would prefer AT LEAST to pgm in Java or C# or ...
[[of course, if “you” would prefer to program C, C++, Forth, or ASM for critical applications, then “you” might not want to stick around for the rest of the talk.]]



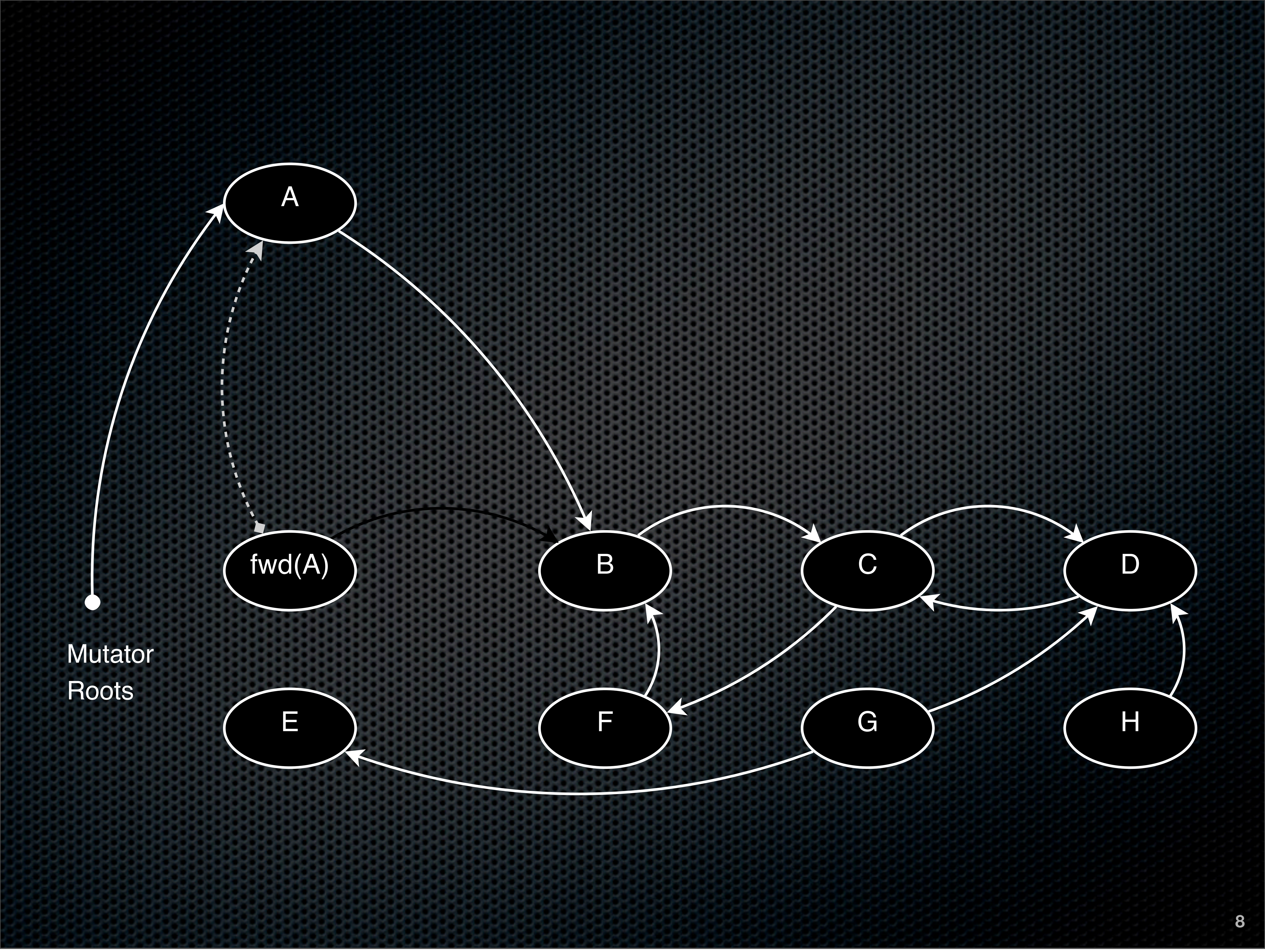
Garbage Collection

- Mutator requests memory
- If request cannot be fulfilled, collector attempts to reclaim unreachable memory

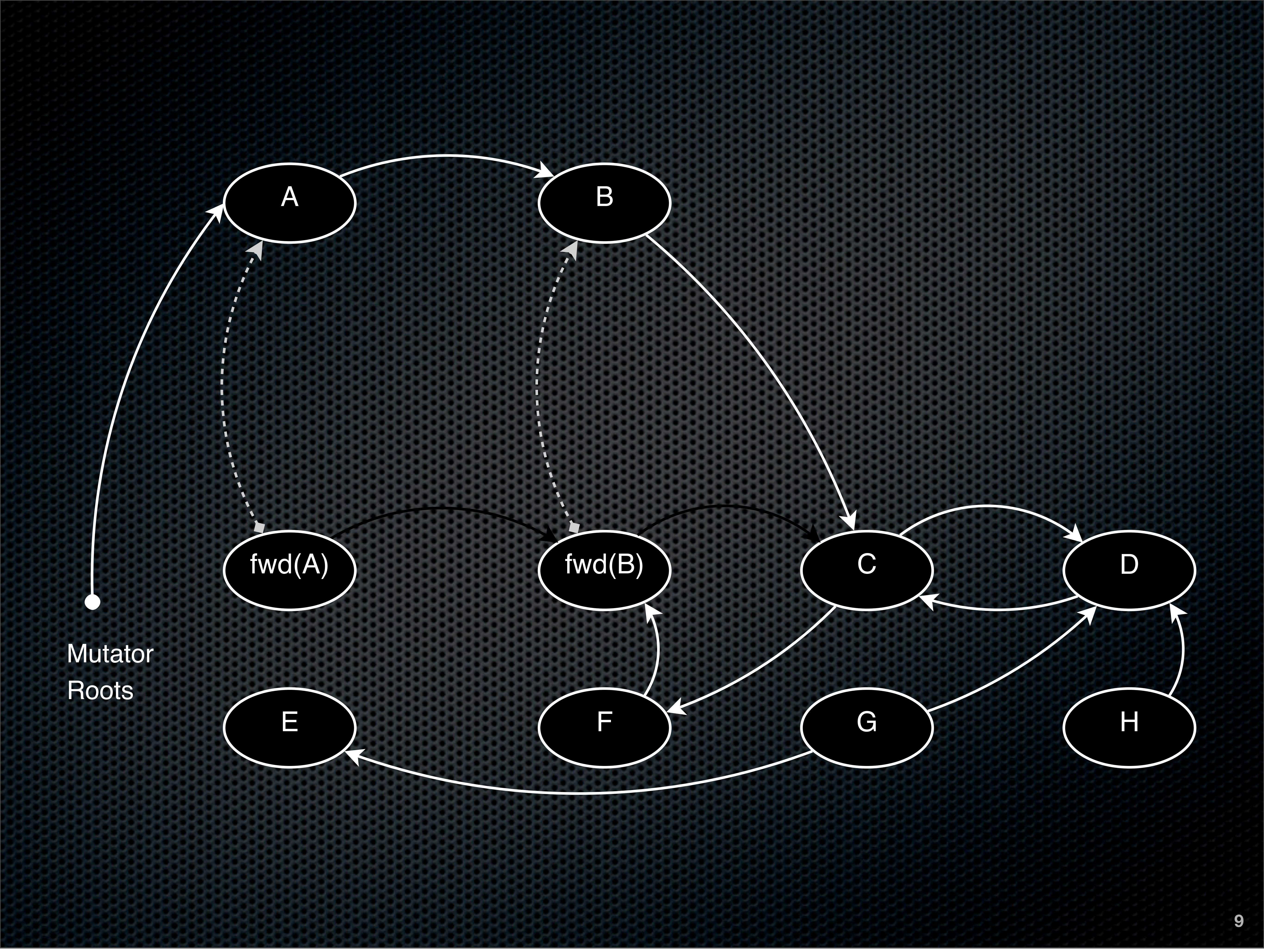
[[because of (1.)conservative GC where mutator might hide pointer data from GC and, (2.)some languages do offer primitives that might expose object addresses (e.g. for hash codes); the point is that in most cases the language enforces insensitivity]]



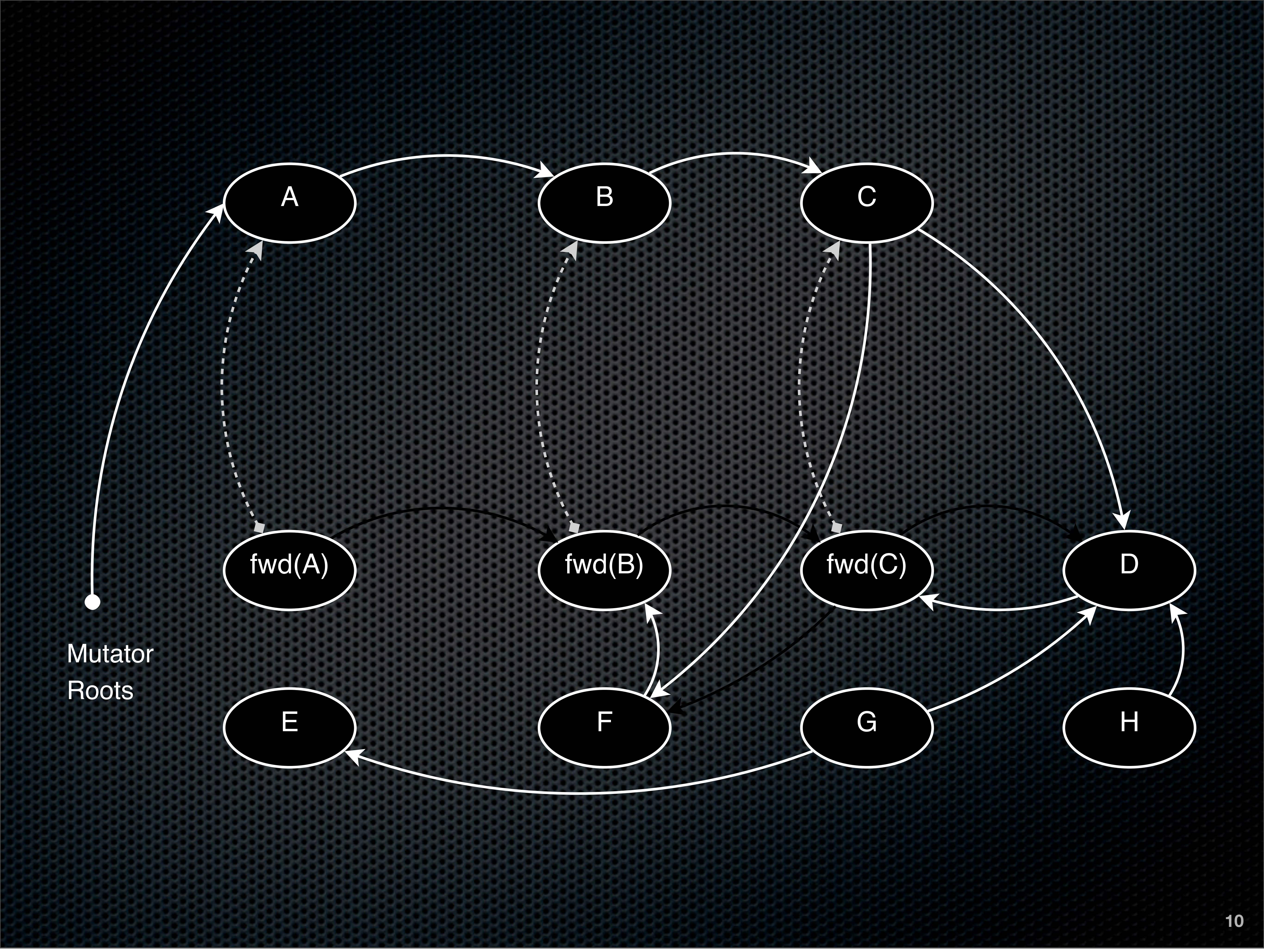
[[this is a quick demo of copying gc just to level the playing field]]
start by scanning roots and copying their reachable objects



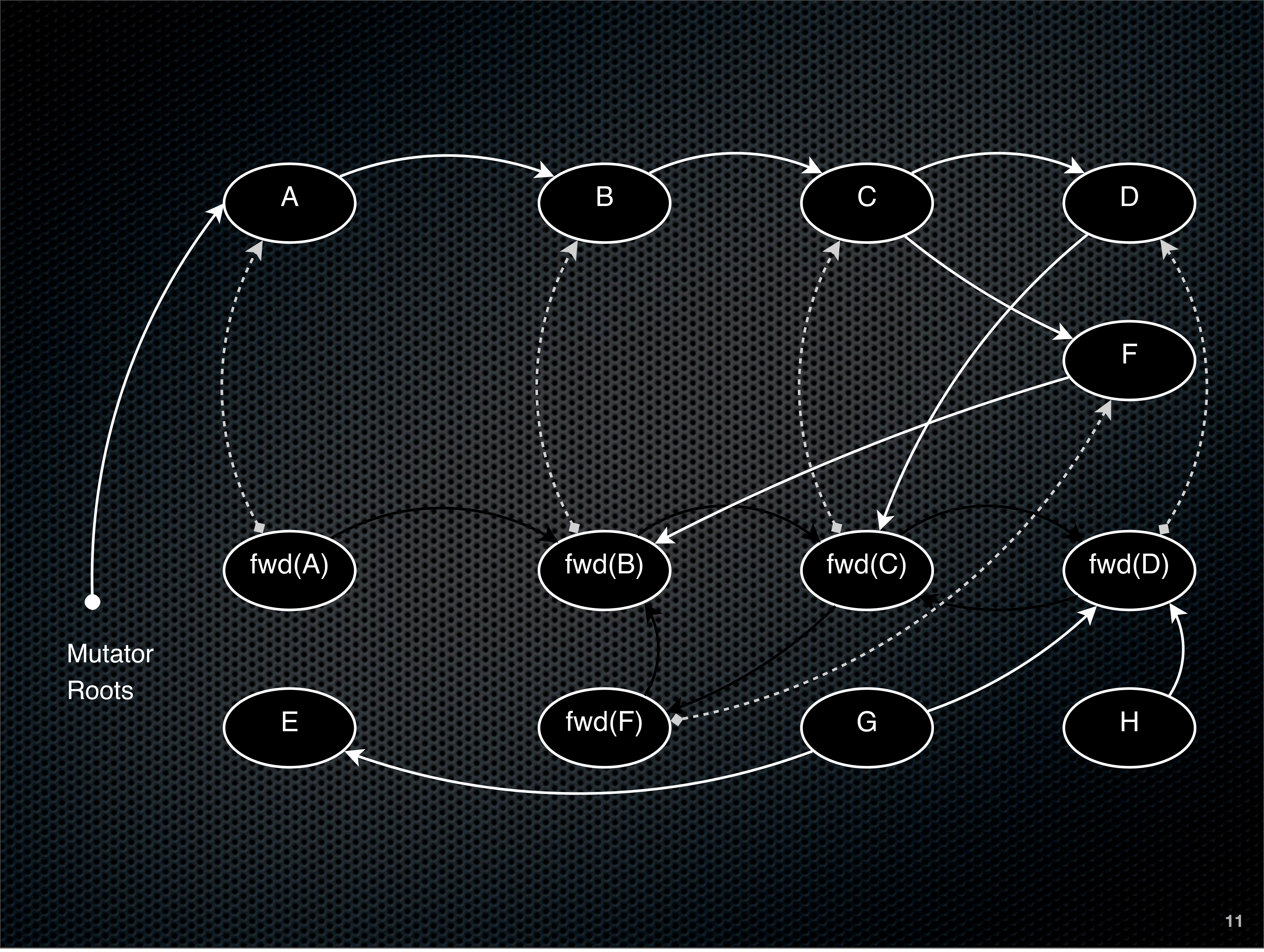
scanning roots causes migration of A into to-space
(here to-space is on top, from-space is on bottom). Next we scan to-space, which means scanning A



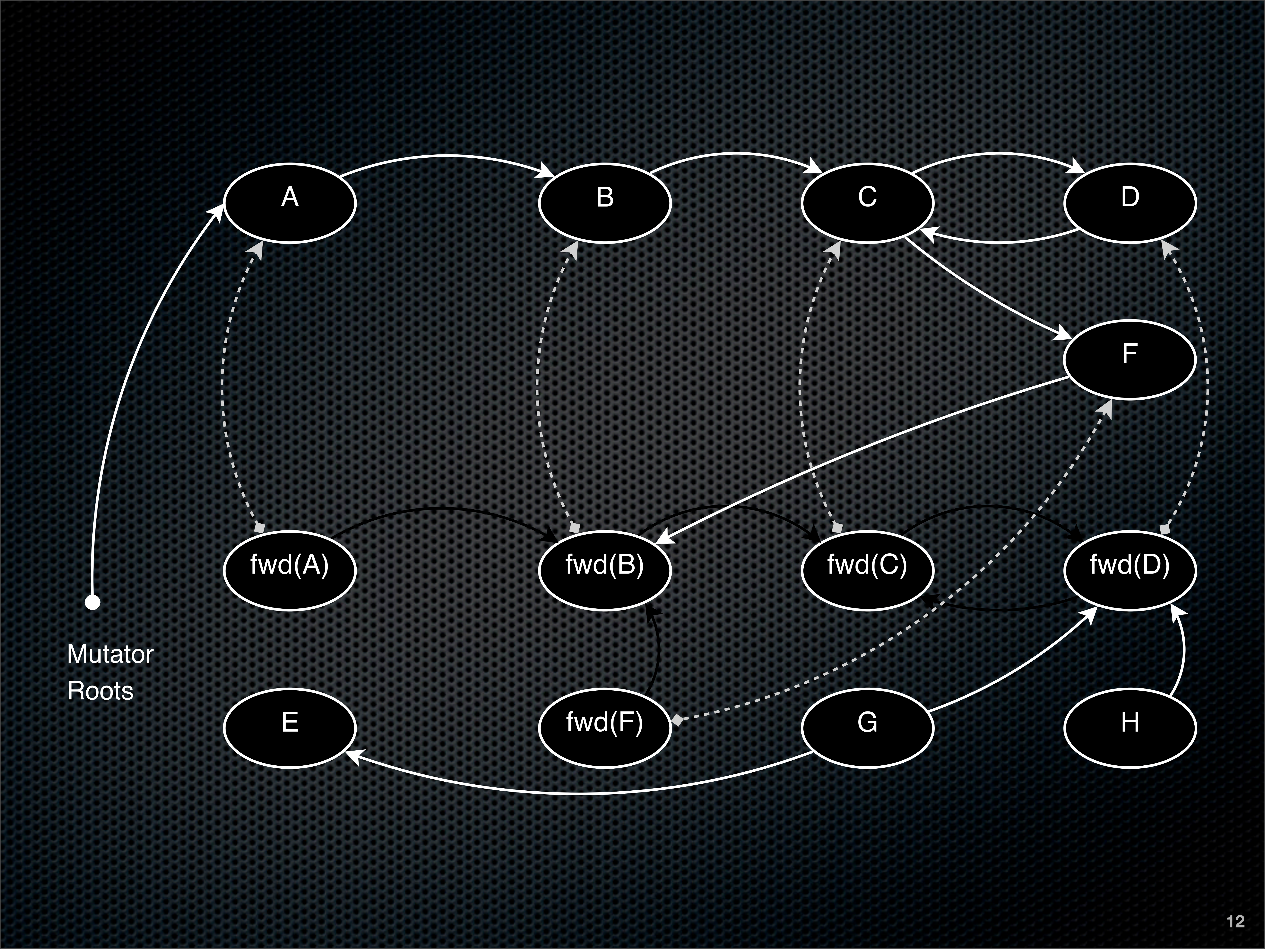
scanning A in to-space migrates B, and we'll scan it next



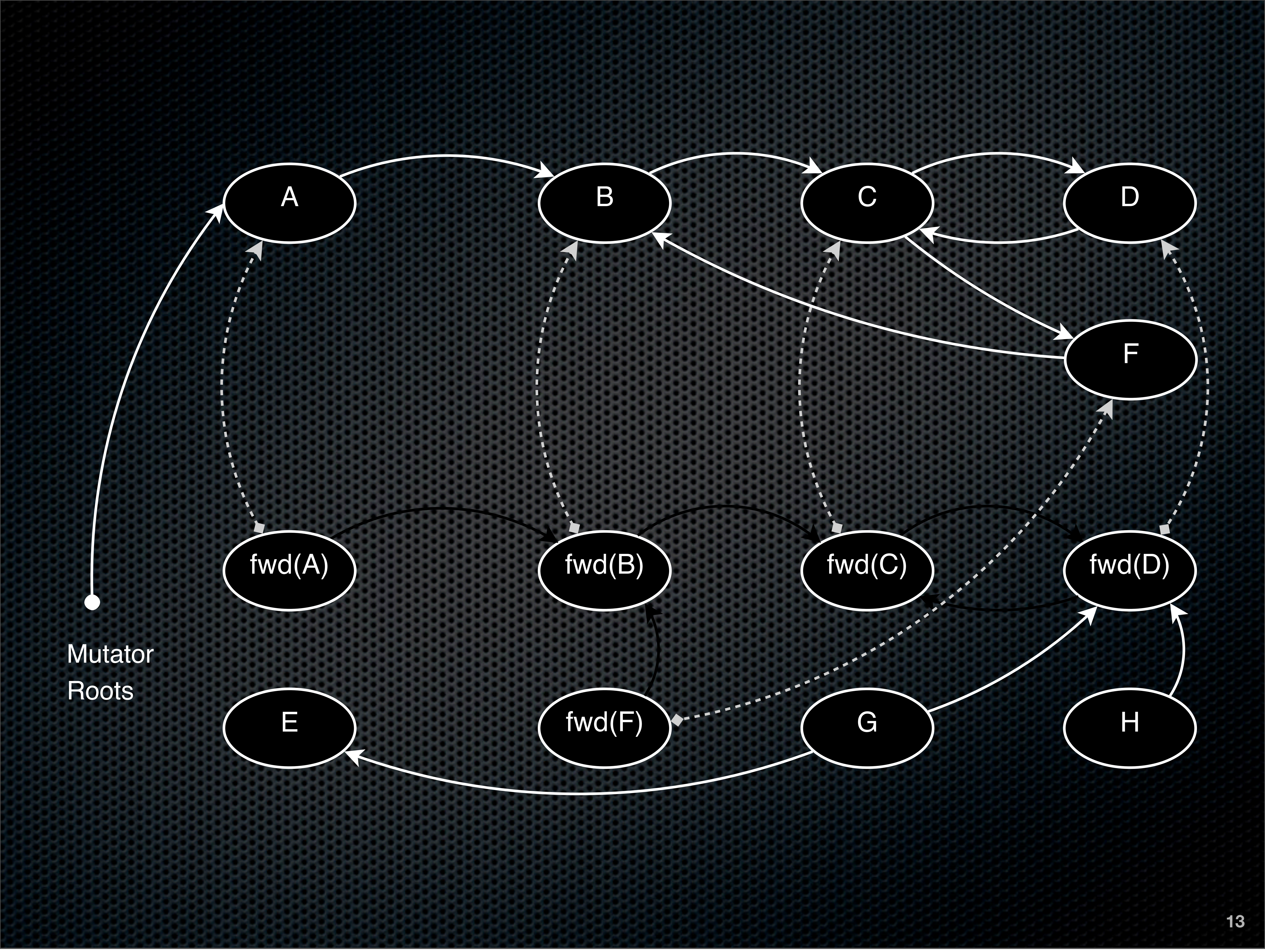
now scanning B migrates C



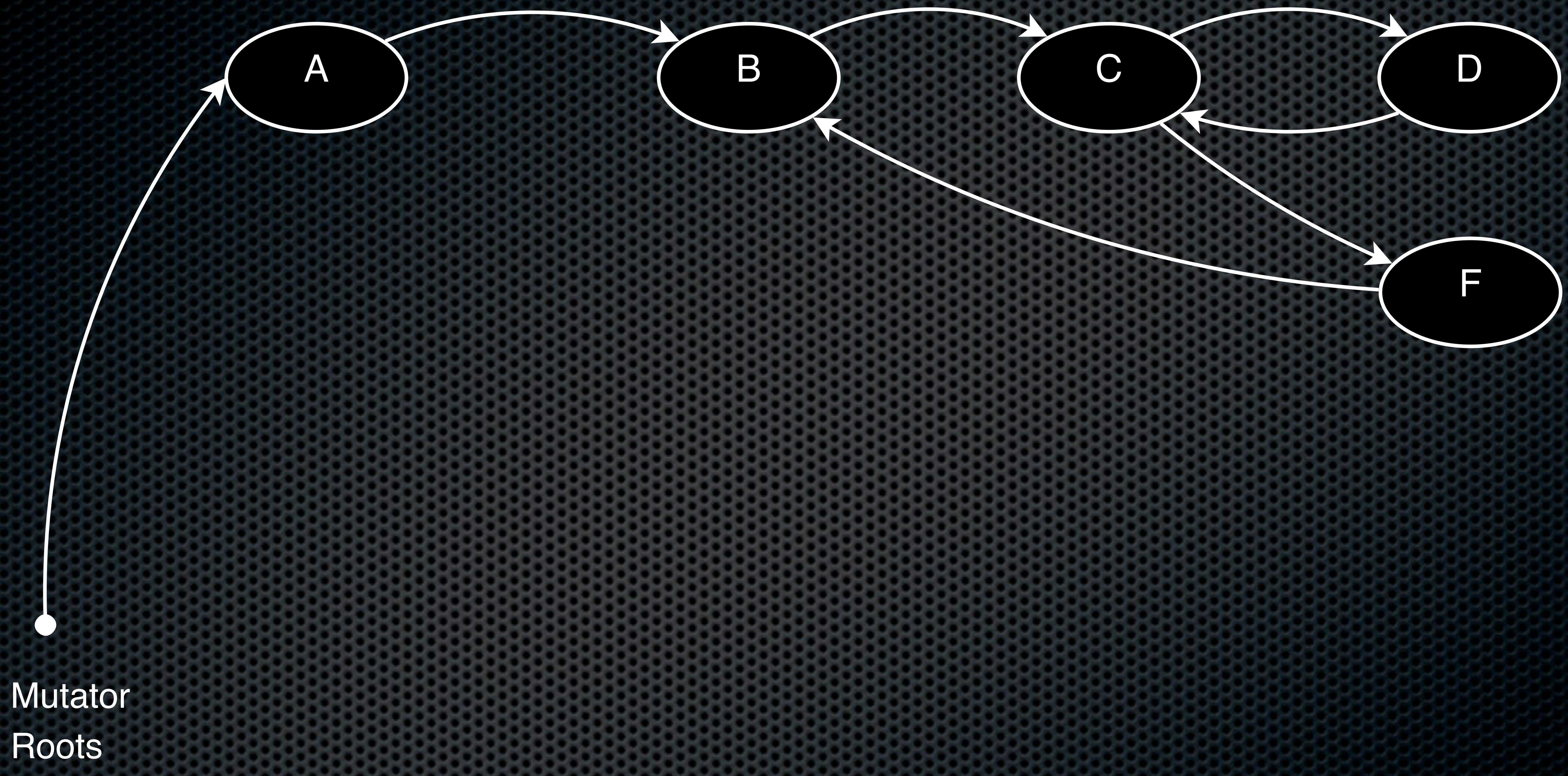
scanning C migrates both D and F
we'll happen to scan D first after copying both objects.



scanning D updates its reference to C



and scanning F updates its reference to B.
All of to-space has been scanned; entirety of from-space can be reclaimed.



leaving us with just the reachable objects from the original graph

Garbage Collection: Standard Objections

- Requires extra memory
- Increases execution time
- Constrains mutator implementation strategy
- Introduces long pauses
  - Disrupts interaction with user

Much of this reduces to “automated processes (1.)introduce new obligations to support automation, and (2.)make it harder to predict system behavior”
Also, the first two objections are moot; (1.)memory leaks use even more memory, and (2.)maintaining metadata to guide manual mgmt adds time overhead.

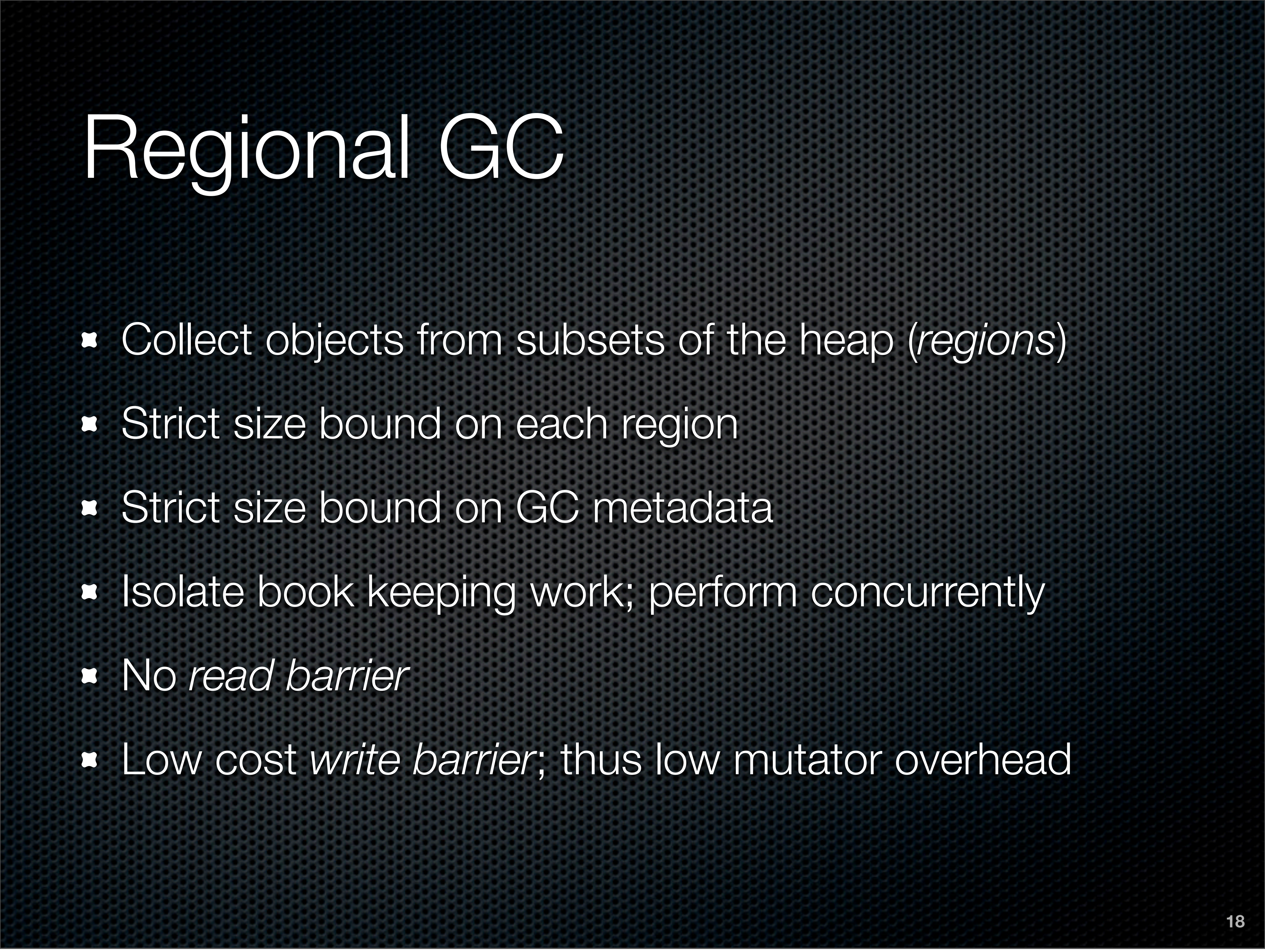
Bounding Pause Times

- My work: eliminate long pauses
- 64-bit address space: larger memories, longer pauses; problem only getting worse
- Total memory usage, overall throughput, and complexity of GC invariants also matter

Say: “We already see pause times on the order of seconds with the memory accessible on 32-bit systems; the problem is only going to get worse as we get more addressable memory on 64-bit machines.”
I am not getting rid of the pauses entirely. I am just introducing strict bounds on how long they are allowed to be. The bounds I am trying to achieve are on the order of <100ms, which is not good enough for most hard real-time systems, but is fine for many classes of applications.
On the last note, I am just making it explicit that I am addressing the three issues w.r.t. other collection technology, not explicitly mem mgmt.

Us and Them

Our invention!



Regional GC

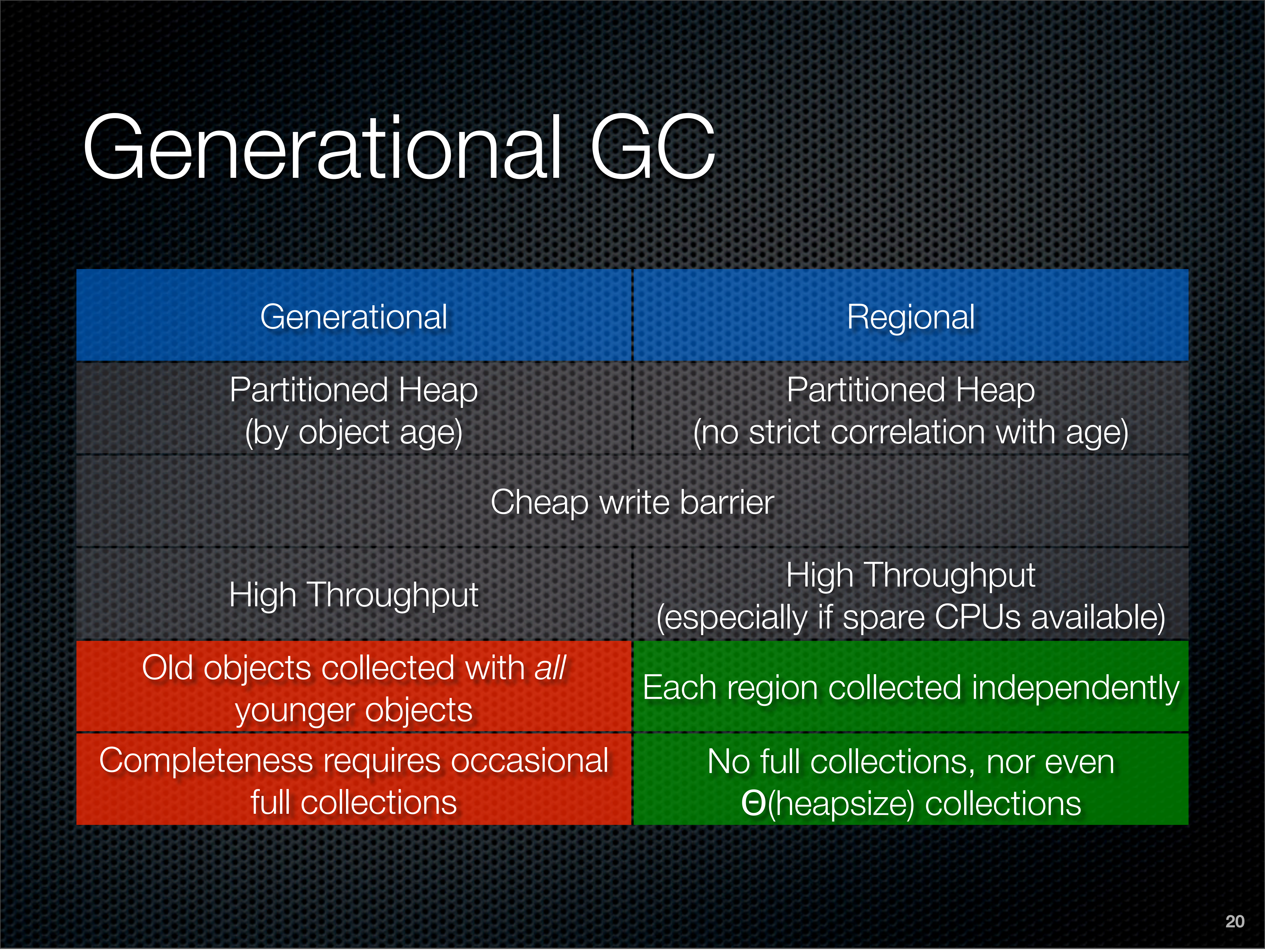
- Collect objects from subsets of the heap (*regions*)
- Strict size bound on each region
- Strict size bound on GC metadata
- Isolate book keeping work; perform concurrently
- No *read barrier*
- Low cost *write barrier*; thus low mutator overhead

Say “MY INVENTION”
Generally, write barrier is used to maintain collector invariants in presence of mutator actions.

Current Technology

- Generational GC
- { Incremental, Concurrent, Real-Time } GC
- Garbage-First GC
- (none of the above are my work)

I am putting Incremental/Concurrent/Real-Time GC into the same category because they share similar attributes that contrast them against the Regional GC.
I am mentioning Garbage-First collector explicitly because the Regional collector draws a lot of inspiration from it, and therefore I need to explicitly point out the novelties in the Regional collector.



Generational GC

Generational	Regional
Partitioned Heap (by object age)	Partitioned Heap (no strict correlation with age)
Cheap write barrier	
High Throughput	High Throughput (especially if spare CPUs available)
Old objects collected with *all* younger objects	Each region collected independently
Completeness requires occasional full collections	No full collections, nor even Θ(heapsize) collections

MY WORK IS IN RIGHT COLUMN
1. “Age” is quoted because there are some varying notions of age
2. Collecting newly allocated more often is a great *initial* heuristic (weak generation hypothesis); does not scale (strong gen. hypothesis does not hold); GC implementors ignore generational effect at peril.
3. Need to track old-to-young references for two reasons:
(a.) To ensure that reachable young objects are not reclaimed by GC
(b.) To update the old object with the new address for the migrated young one

[[The generational write barrier is cheaper than the regional one.]]
[[The generational remembered sets will occupy less space than the regional one, at least by a constant factor]]

Incremental, Concurrent, Real-Time GC

Incremental, Concurrent, Real-Time	Regional
All collection work interleaved/concurrent with mutator	Book-keeping work concurrent with mutator
Never pauses for time proportional to heap size	
Complex, expensive {read, write} barriers	No read barrier; cheap write barrier
Low overall throughput	High overall throughput
Good MMU at fine grain (conjectured)	Good MMU at coarse grain (provably)

Explanation of MMU: choose a fixed grain of time, then determine the minimum execution time the mutator gets within that grain over the course of entire computation.
When explaining the MMU row, point out classes of applications where this is and is not appropriate (missile, med. devices, vs video games or office applications)

Garbage-First GC

Garbage-First	Regional
Partitioned heap; cheap write barrier	
Good performance on typical programs	
Searches for garbage-rich regions	Treats regions uniformly
Soft fine grain pause time bounds	Hard coarse grain pause time bounds
Concurrent marking ensures completeness	
Worst case *quadratic* space usage	Worst case *linear* space usage

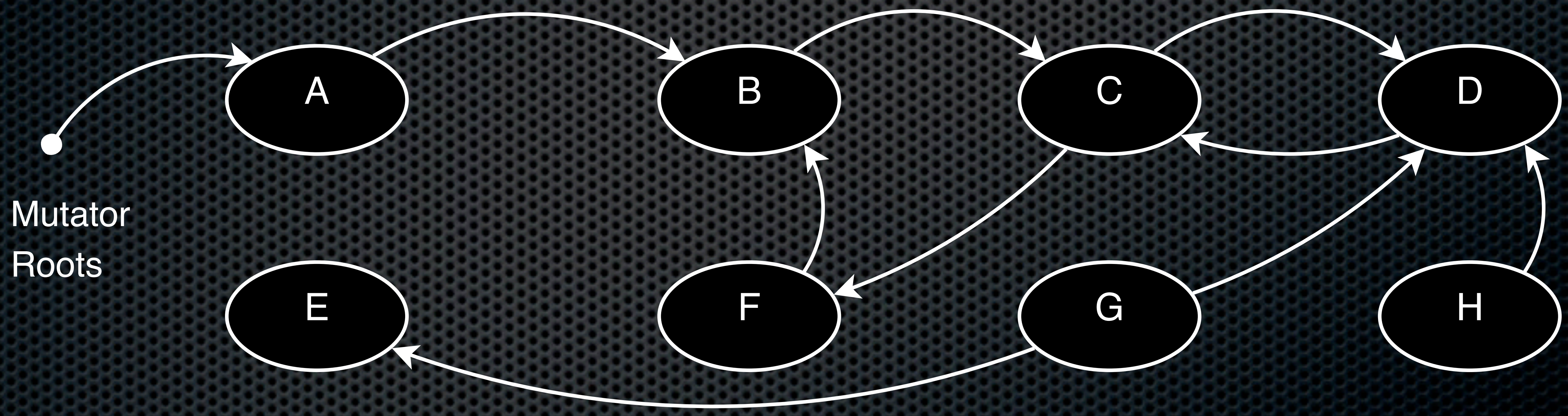
Say "*Heuristic* search for gbg-rich regions" for GF
[[Note: Garbage-First was also a *parallel* collector; it would distribute the collection work across multiple processors, which is part of why it chose its points-into remset structure.]]

Regional Collection

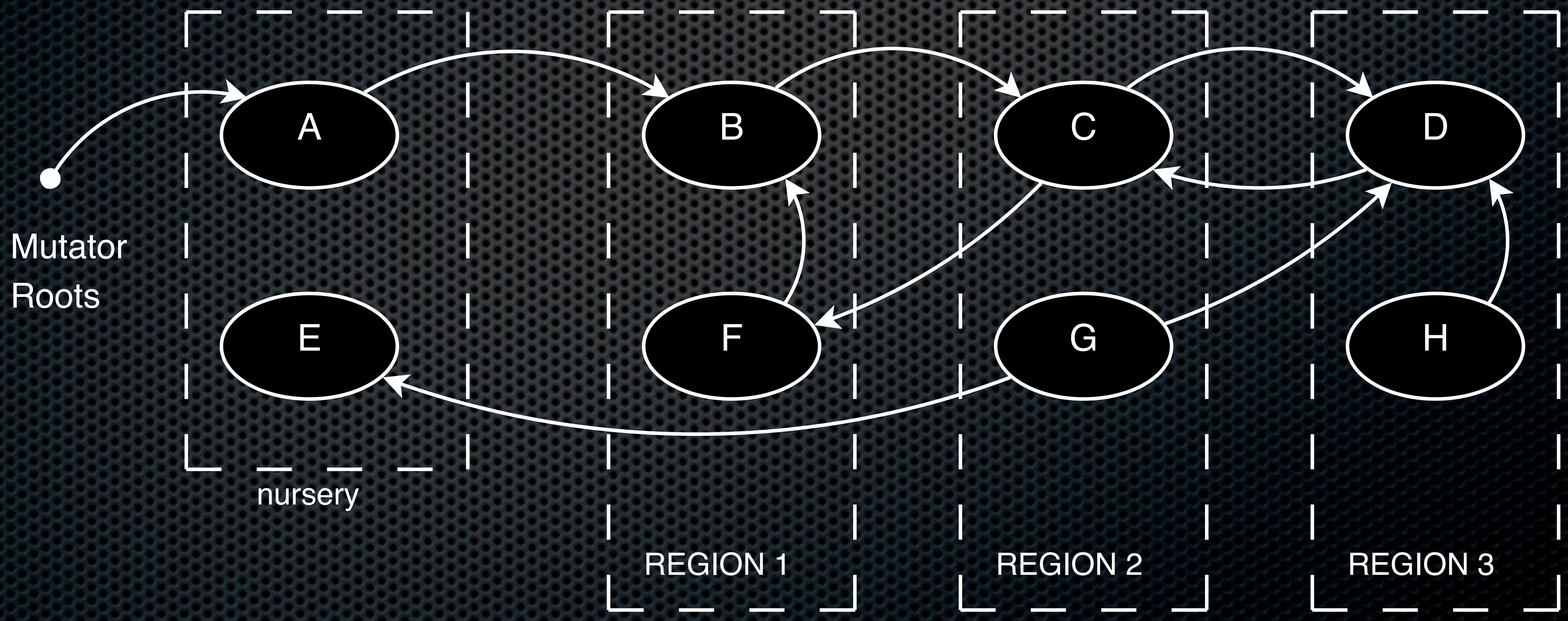
Regional GC:
Heap Structure

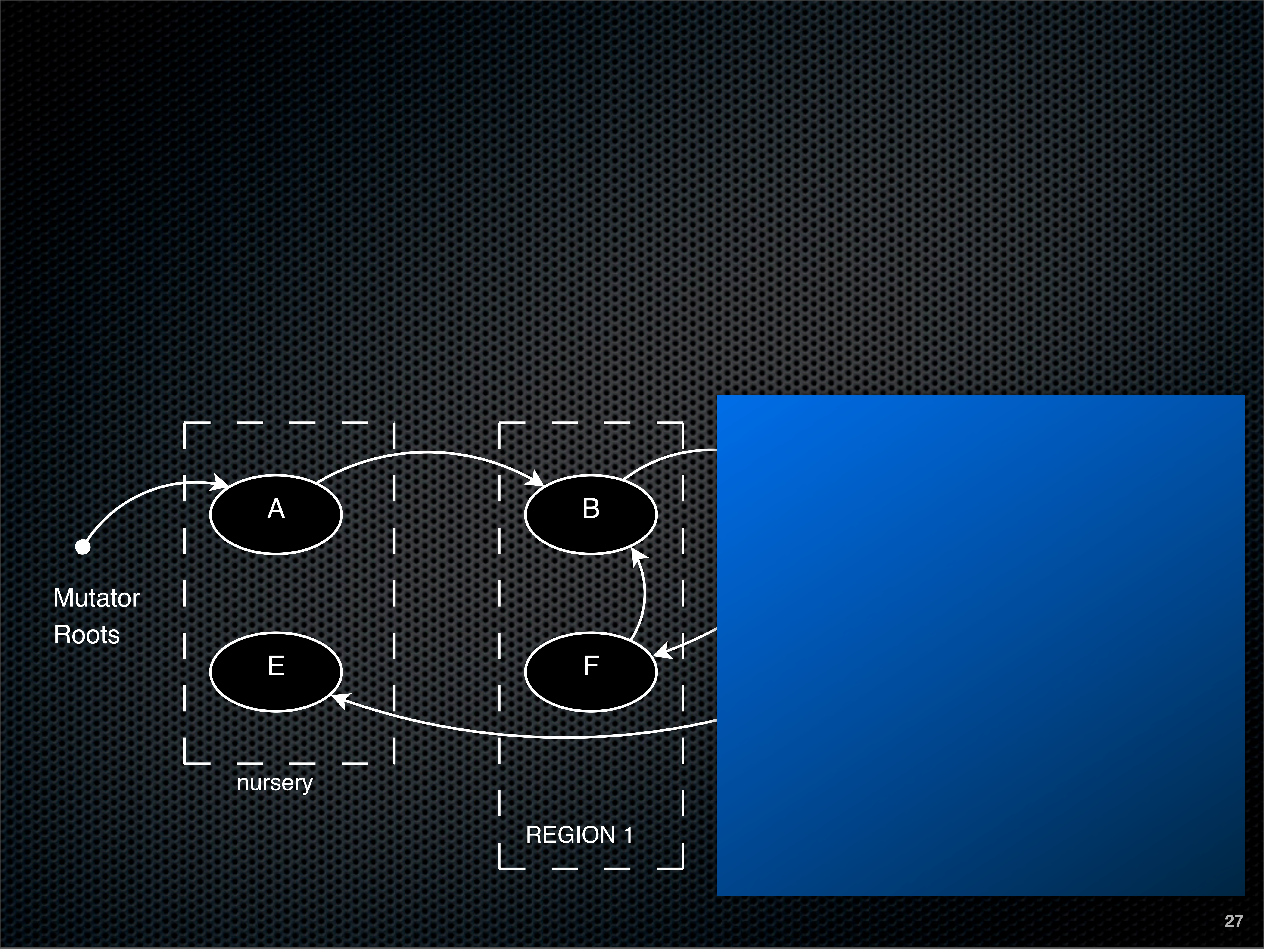
- Heap (N words) partitioned into regions of fixed capacity (R words)
- Thus N/R is total number of regions
- Minor collection: collect nursery only
- Major collection: collect some region *and* nursery together

Define nursery *vocally*: say its where the young objects live
Assumption: the size of the nursery is significantly less than R; we've been using a 1 MB nursery and 5 MB regions
Note: Major GC migrates Y+R words; thus all migrated objects may not fit into R. The collection policy must address this in some manner. Currently using "reserve regions" to resolve this, but the long term approach will a more sophisticated policy. The point is that the collector may migrate objects from region to region; the mutator cannot do so (and should be ignorant of object migration).

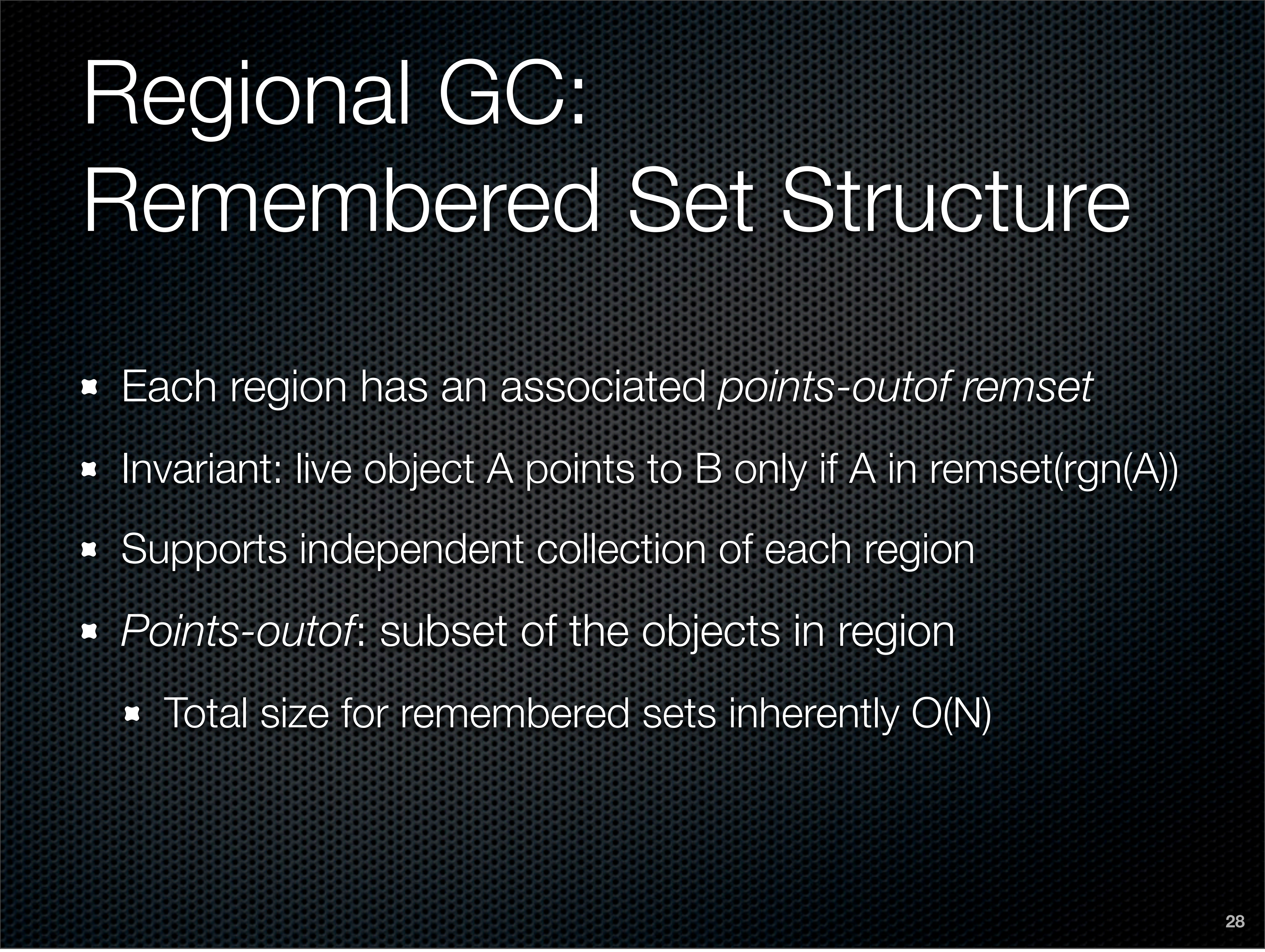


Lets partition the object graph


Mutator
Roots
A
B
C
D
E
F
G
H
nursery
REGION 1
REGION 2
REGION 3



Ask: How are we going to do this? We need to collect the unreachable objects, but we don't know which of the incoming pointers are from reachable objects.



Regional GC: Remembered Set Structure

- Each region has an associated *points-outof remset*
- Invariant: live object A points to B only if A in remset(rgn(A))
- Supports independent collection of each region
- *Points-outof*: subset of the objects in region
  - Total size for remembered sets inherently O(N)

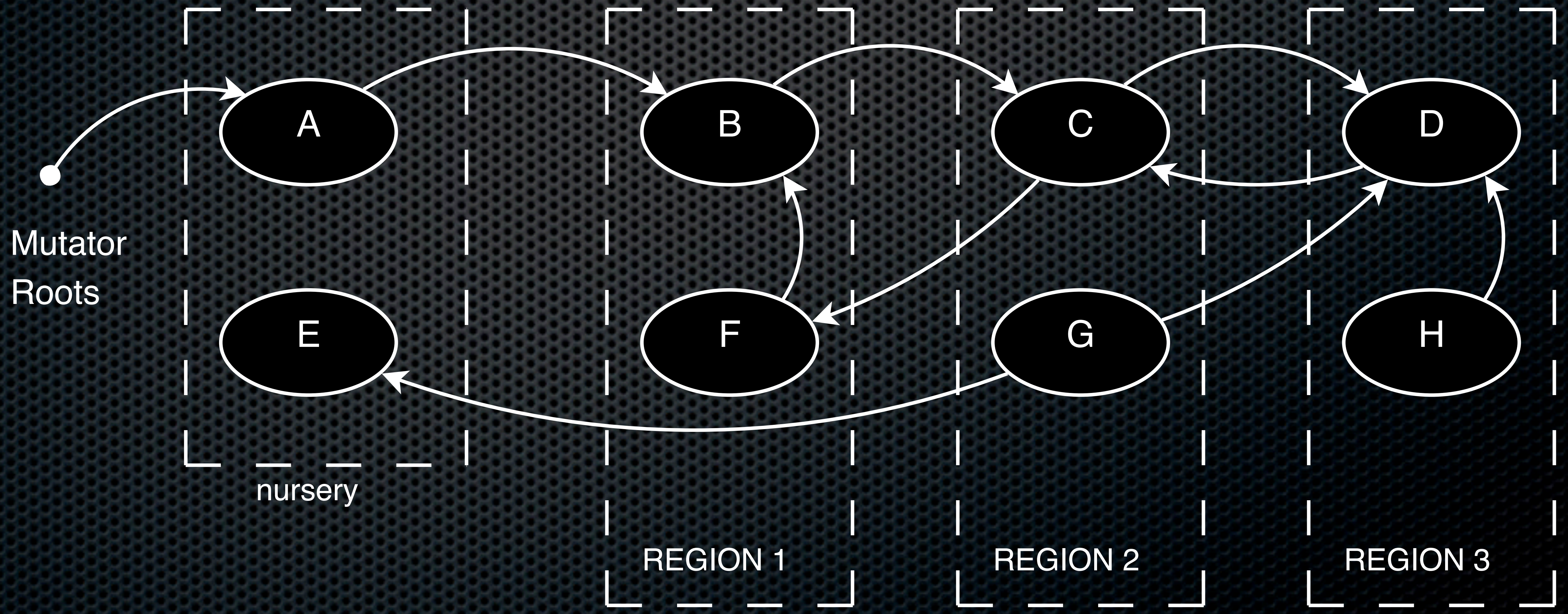
Say: “Tracks any object with a region-crossing reference.”
Say: “This is a slight generalization of a remembered set in a generational collector.”
Invariant only applies when A and B are [1] in different regions, and [2] A not in nursery.
(Note the implication is one-way, which means that the remsets are *imprecise*. There will be a quiz on this later.)

Regional GC: Remembered Set Structure

- Maintenance is standard
- Mutator logs introduction of region-crossing references
  - Write-barrier stores objects holding references into log
  - Periodically fold log into remembered sets
- Collector must update remembered sets

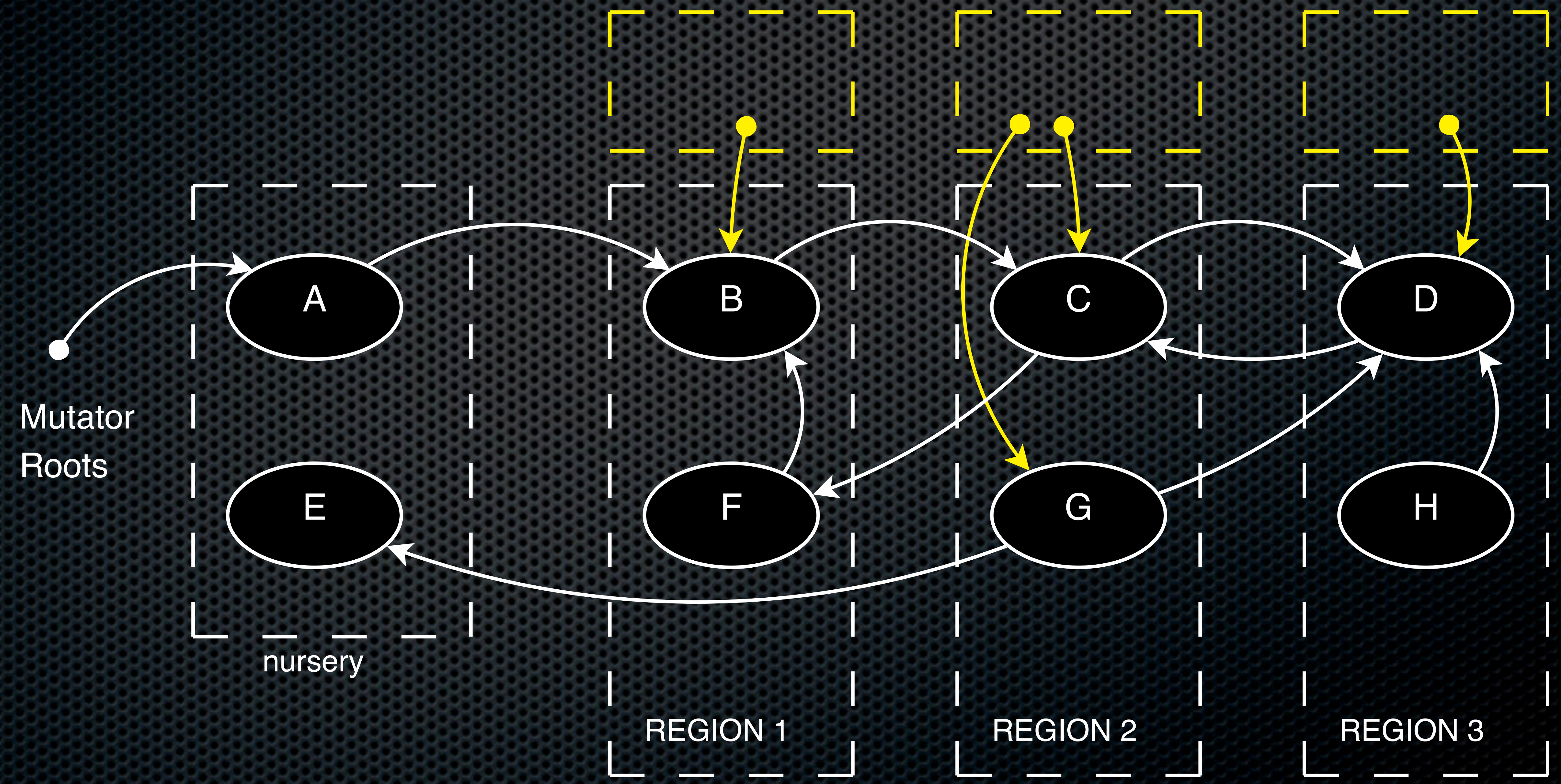
When introducing GC meta-data structure, the questions I ask are: what obligations does it impose on the mutator, and what obligations does it impose on the collector.
The mutator requirements are certainly not novel. The collector requirements are not novel either, but they were new to me. (Point out that the updating must happen during both minor and major gc's)
[[Important: mutator is responsible for logging objects when region-crossing pointers are introduced; other parts of the collector will make use of that.]]

Live object A points to B only if A in remset(rgn(A))



RE: the invariant, A and B are in different regions, and are not in the nursery

Live object A points to B only if A in remset(rgn(A))



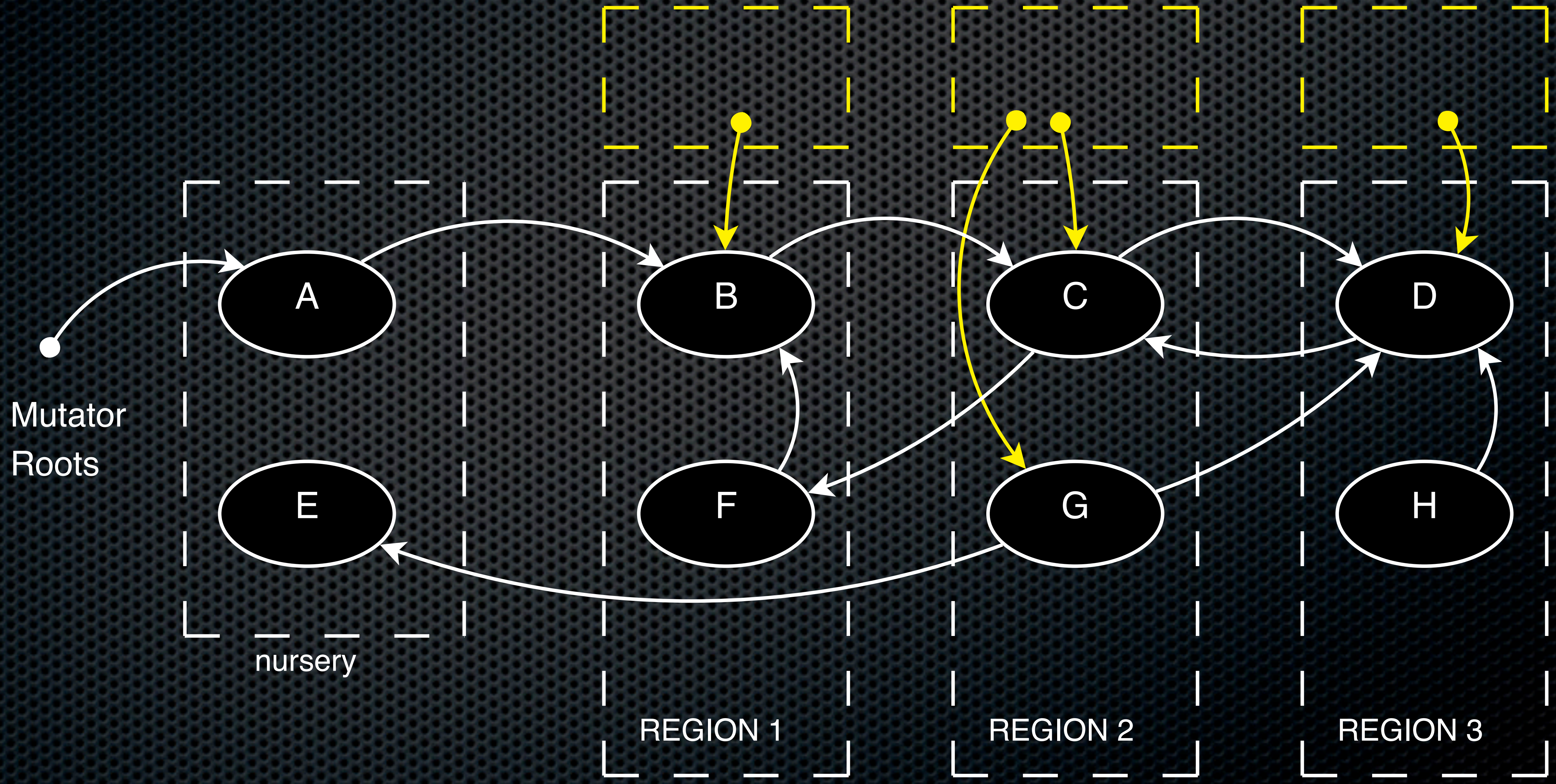
Note that G is unreachable, so this is not the smallest remembered set allowed by the invariant. (But it is the remembered set that will tend to develop.)
Point out that if mutator added references from F to one of C or D, then it would need to add F to Region 1's remset. (Don't say F to A; keep discussion regional)
Point out that if mutator changed B's outgoing reference to point at F, then we'd have further imprecision.
"So, now we can collect any region we like independently from the others. This raises a question: ..."

Which region to collect next?

- Vast range of workable policies
- We choose round-robin
- Round-robin behaves like renewal-older-first
  [Clinger and Hansen, PLDI ’97]
  [Clinger and Rojas, SCP ’06]

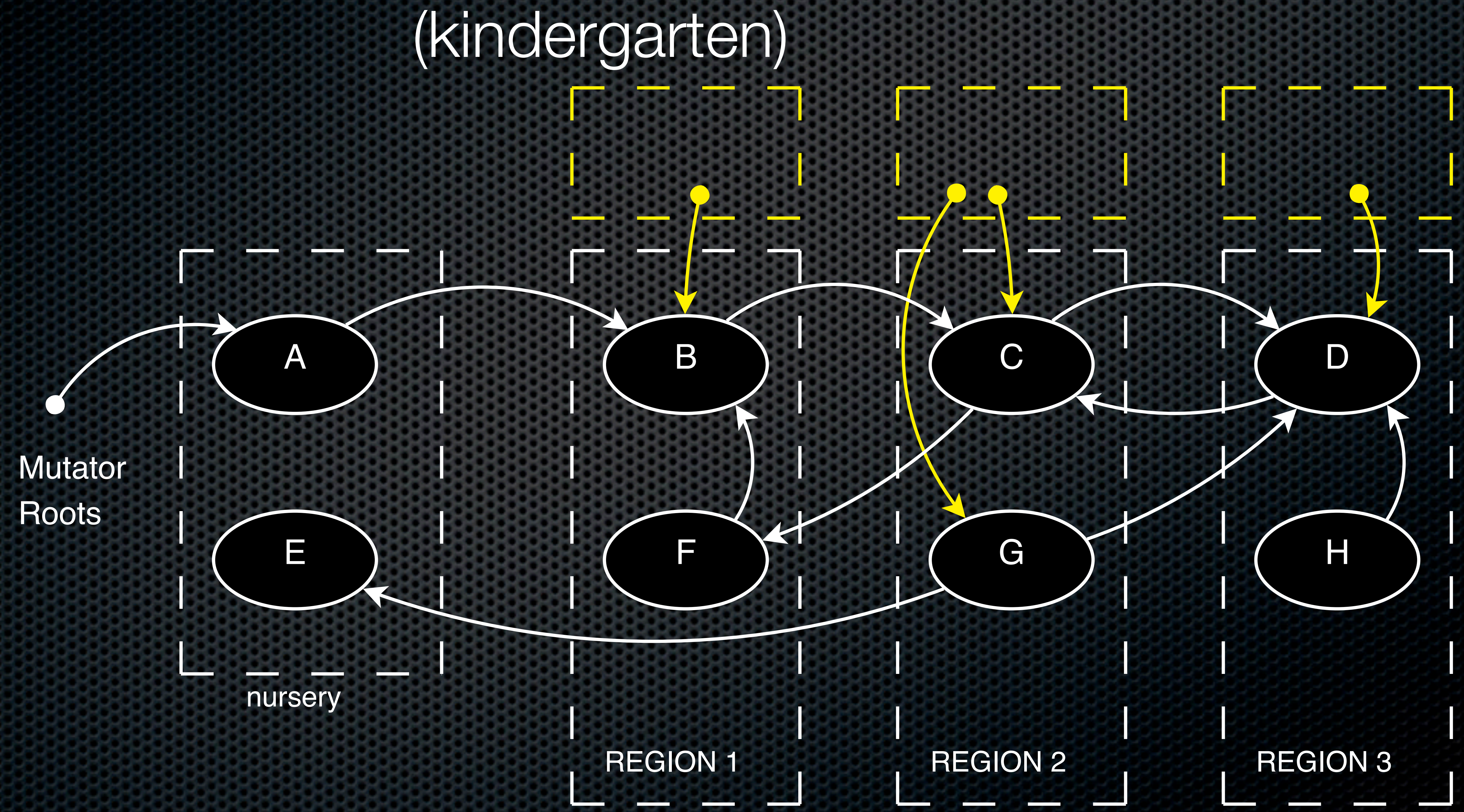
[[Say: “It does not matter much”, but only if I believe this...]]
What I really mean is “Any policy that dictates which region to collect reasonably far ahead of time will do.” Last-minute selection a la Garbage-First might not be appropriate.
Say “If you know what ROF is, then this acts like that. If you don’t, don’t worry about it; its not a crucial aspect of this work.”

Round Robin Collection



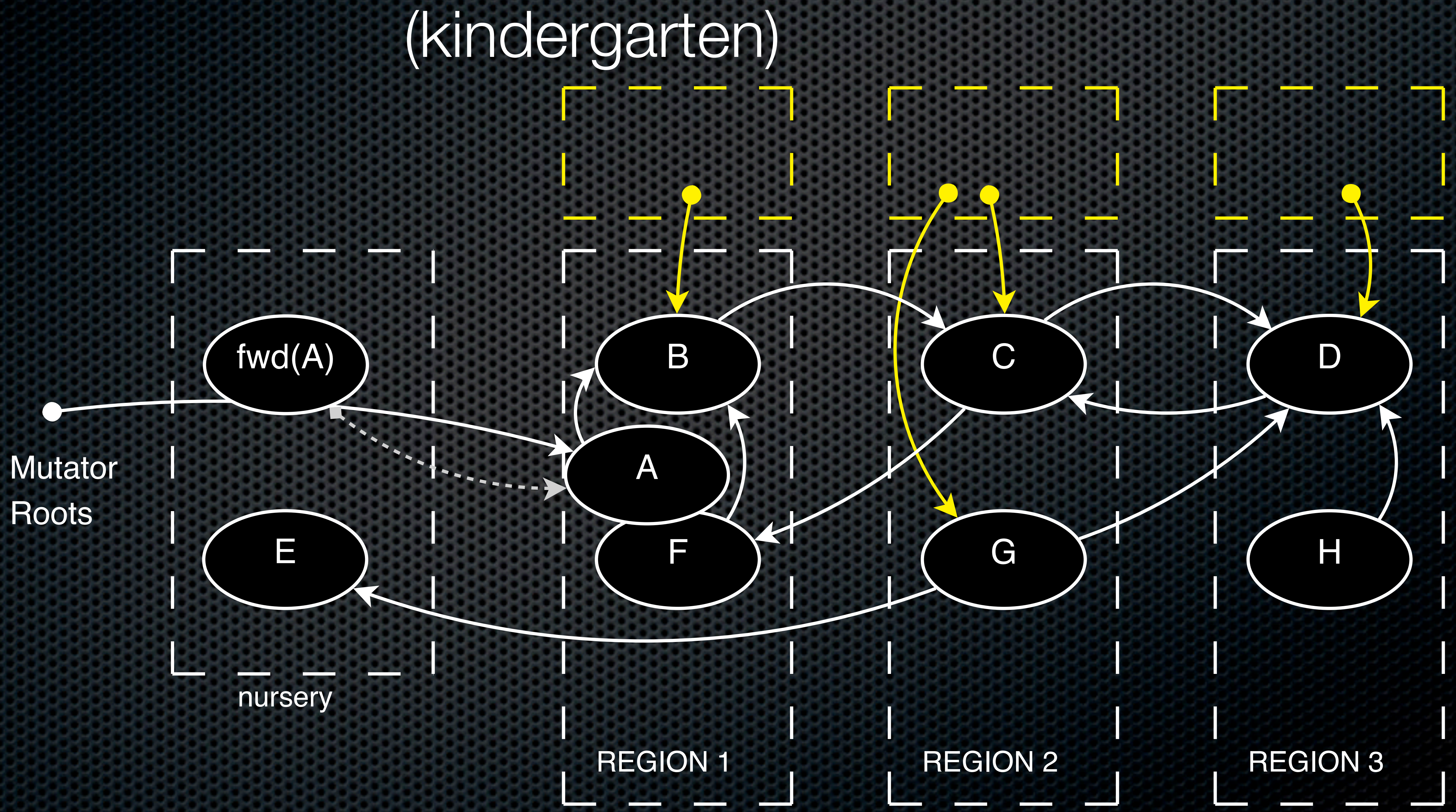
Round-robin means: periodically do major gc of region (i++ mod (N/R)). Repeat.
First going to demo minor gc. On minor gc, evacuate objects out of the nursery into some region with available space.

Minor GC into region 1



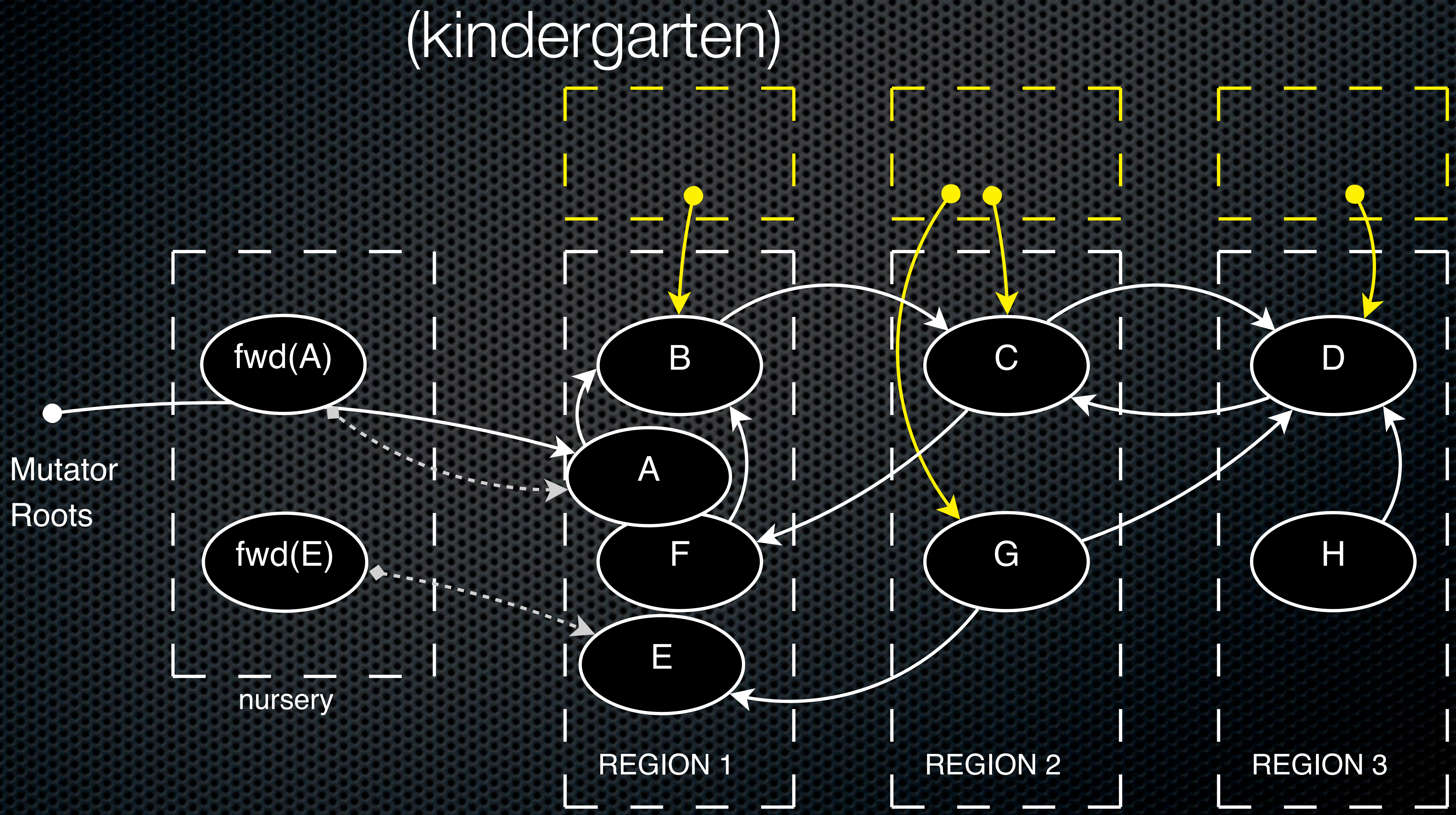
During a minor collection, we evacuate the reachable objects in the nursery into some region (labelled “kindergarten” here). So first we evacuate A into region 1

Minor GC into region 1



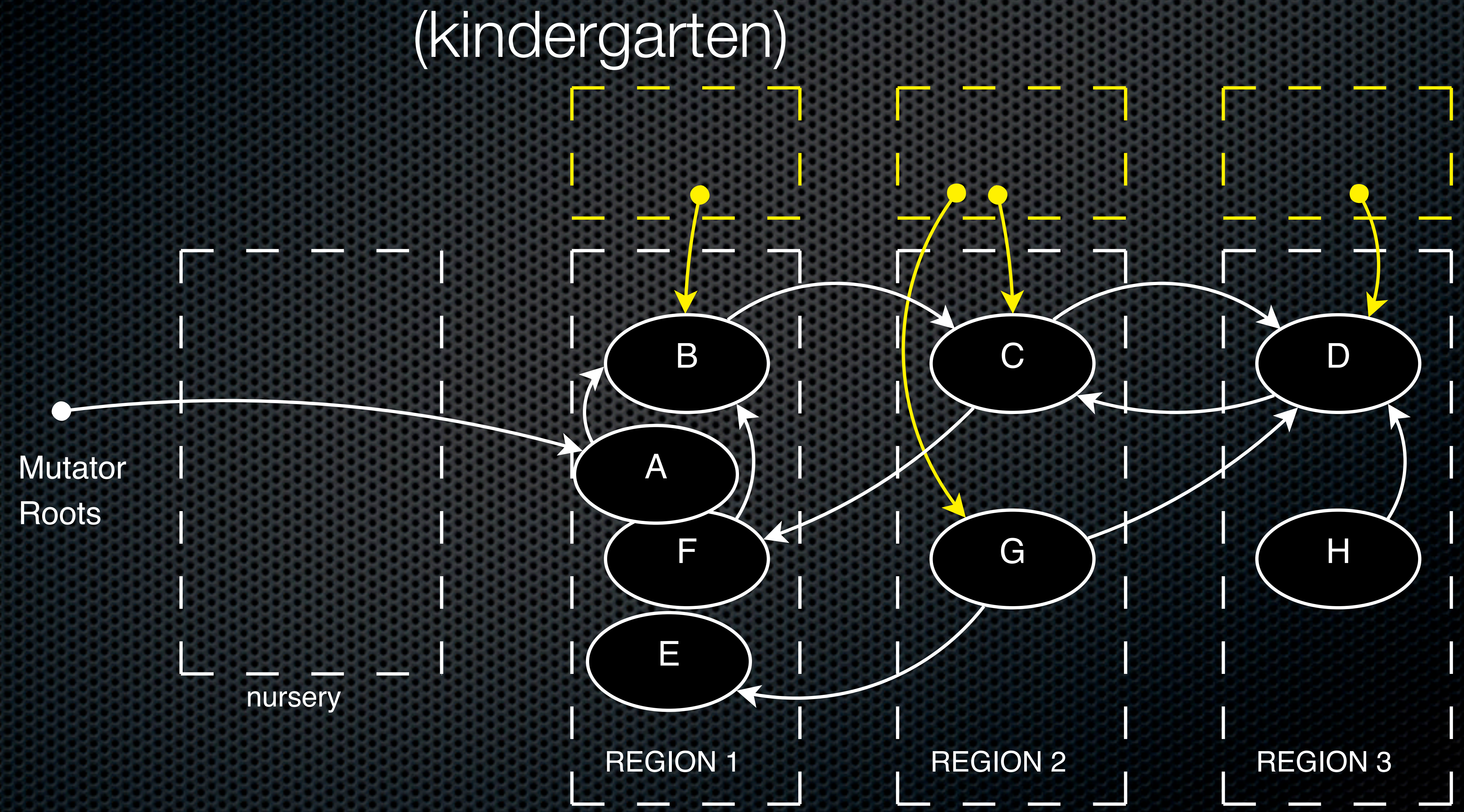
After that, evacuate E as well, because G's reference to E is keeping it alive

Minor GC into region 1



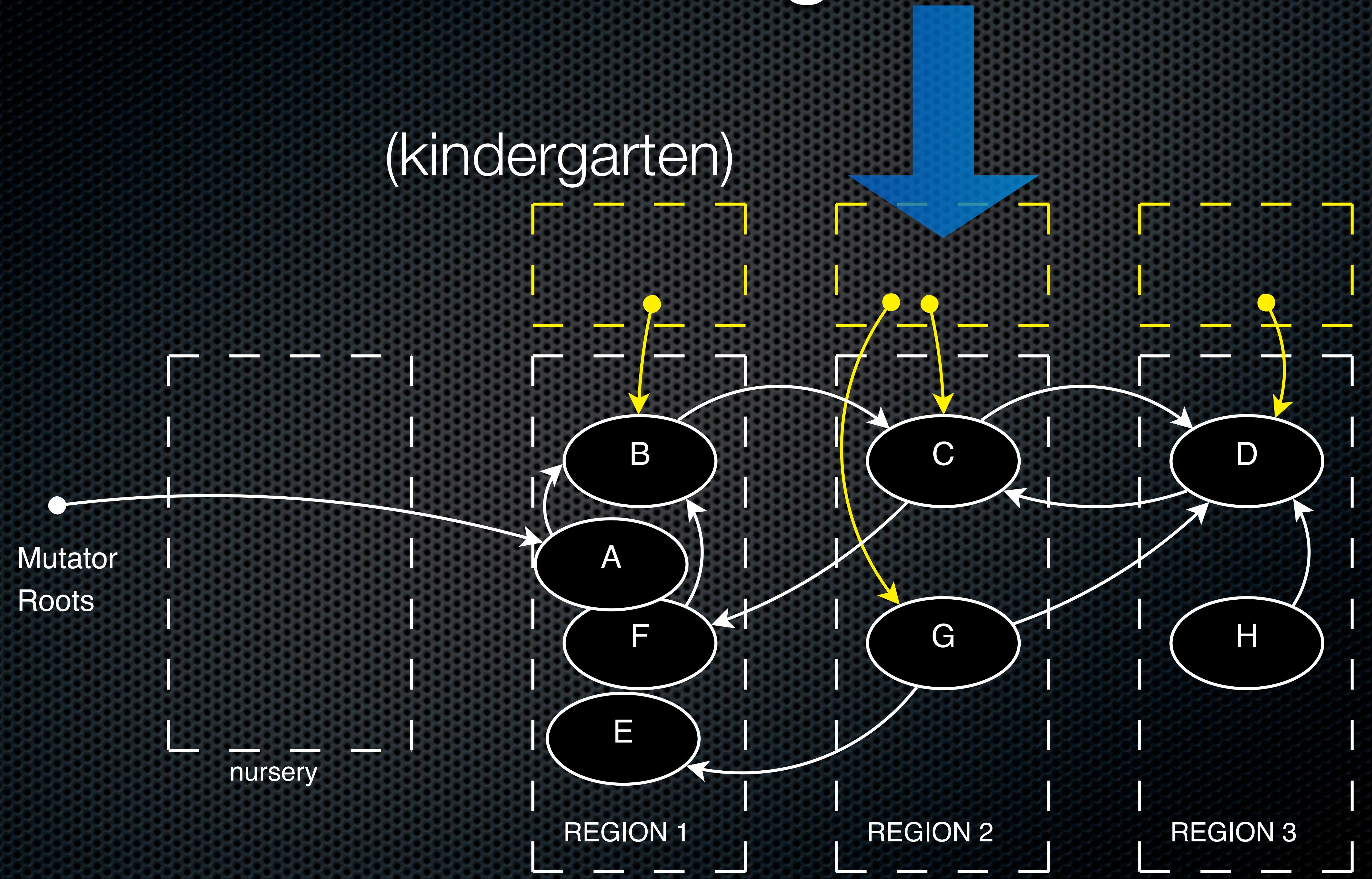
Now nursery has been completed evacuated; we can reclaim its storage

Minor GC into region 1



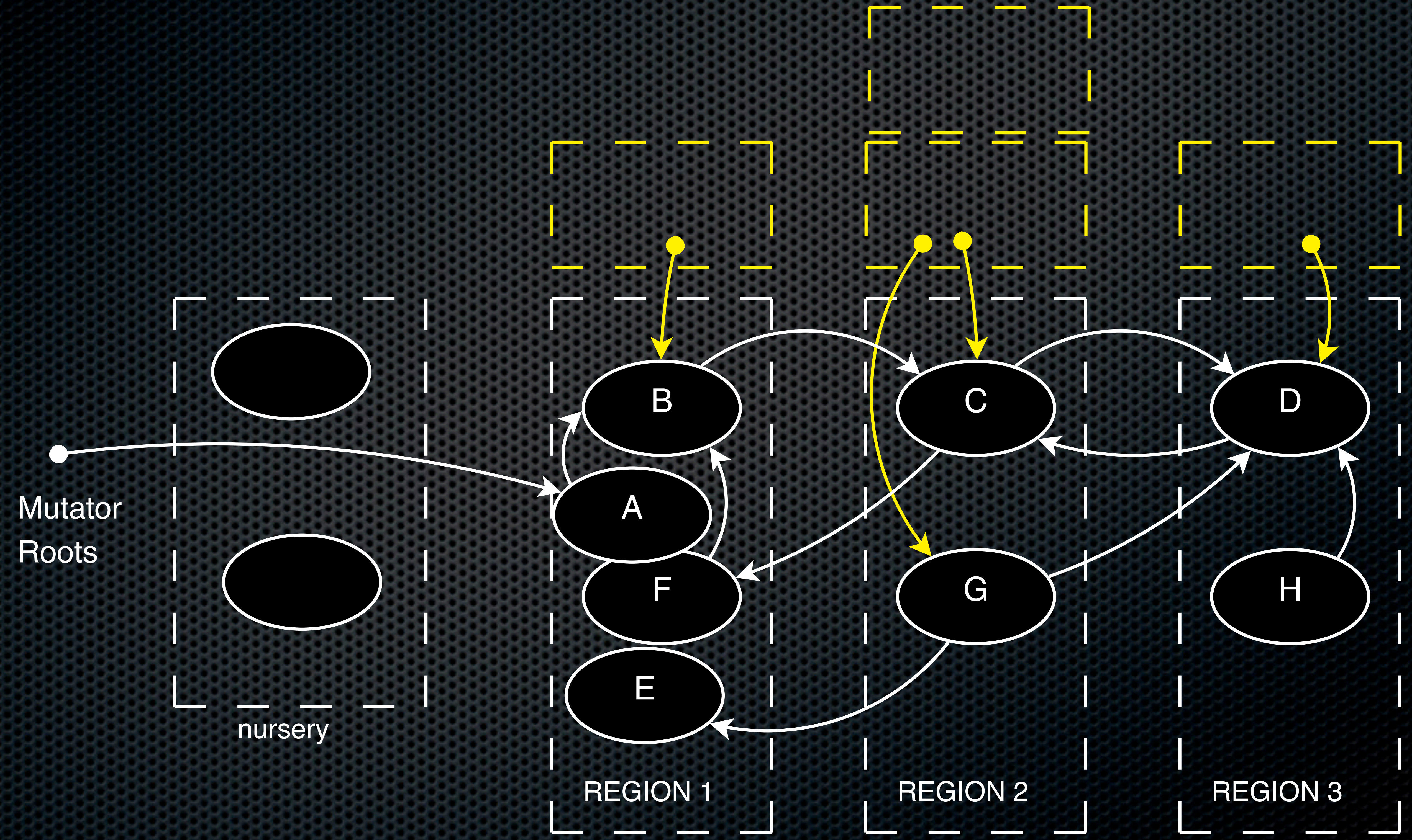
So that's minor collections; now lets consider a major collection of region 2; the mutator will run for a while

Minor GC into region 1



So that’s minor collections; now lets consider a major collection of region 2; the mutator will run for a while

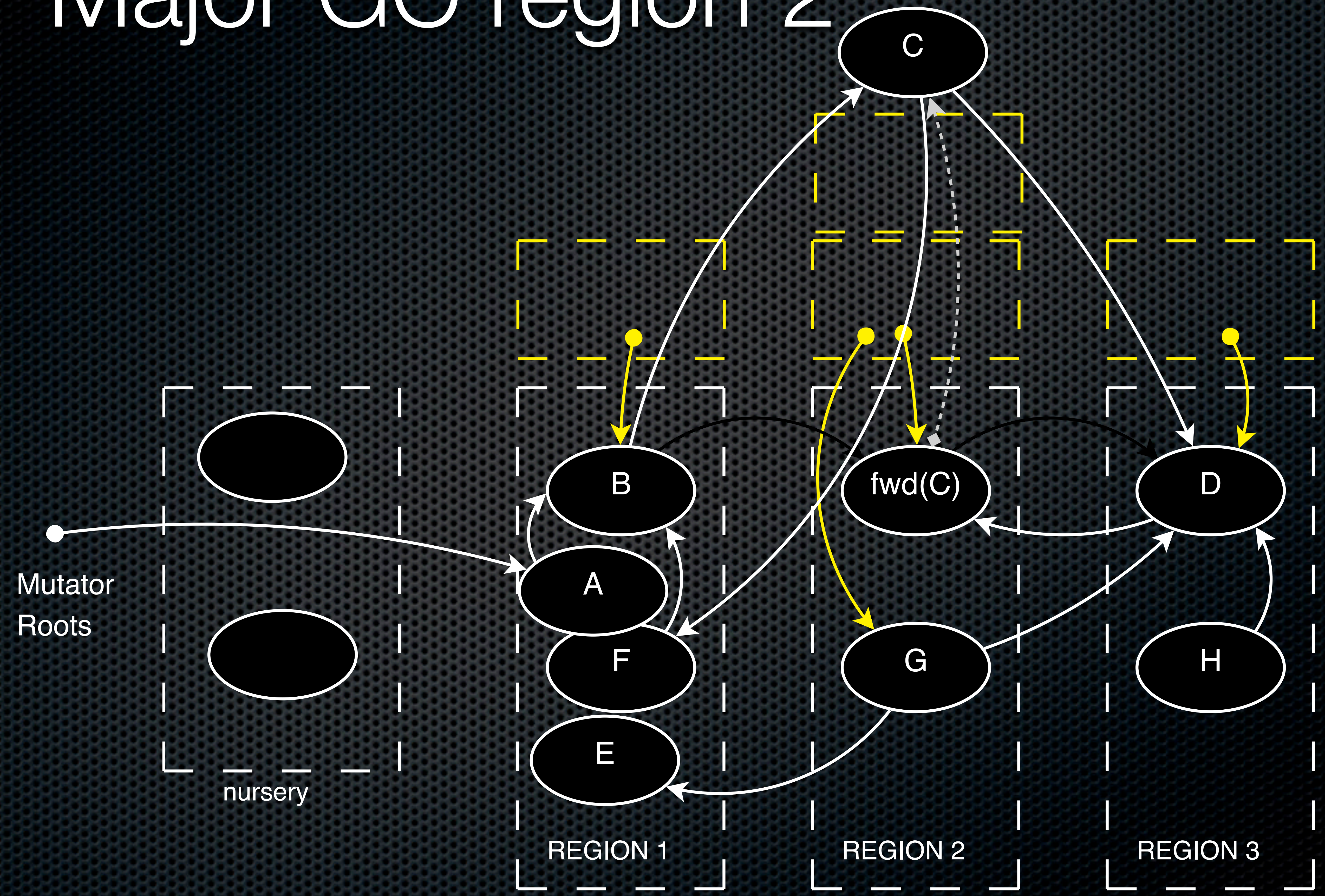
Major GC region 2



39

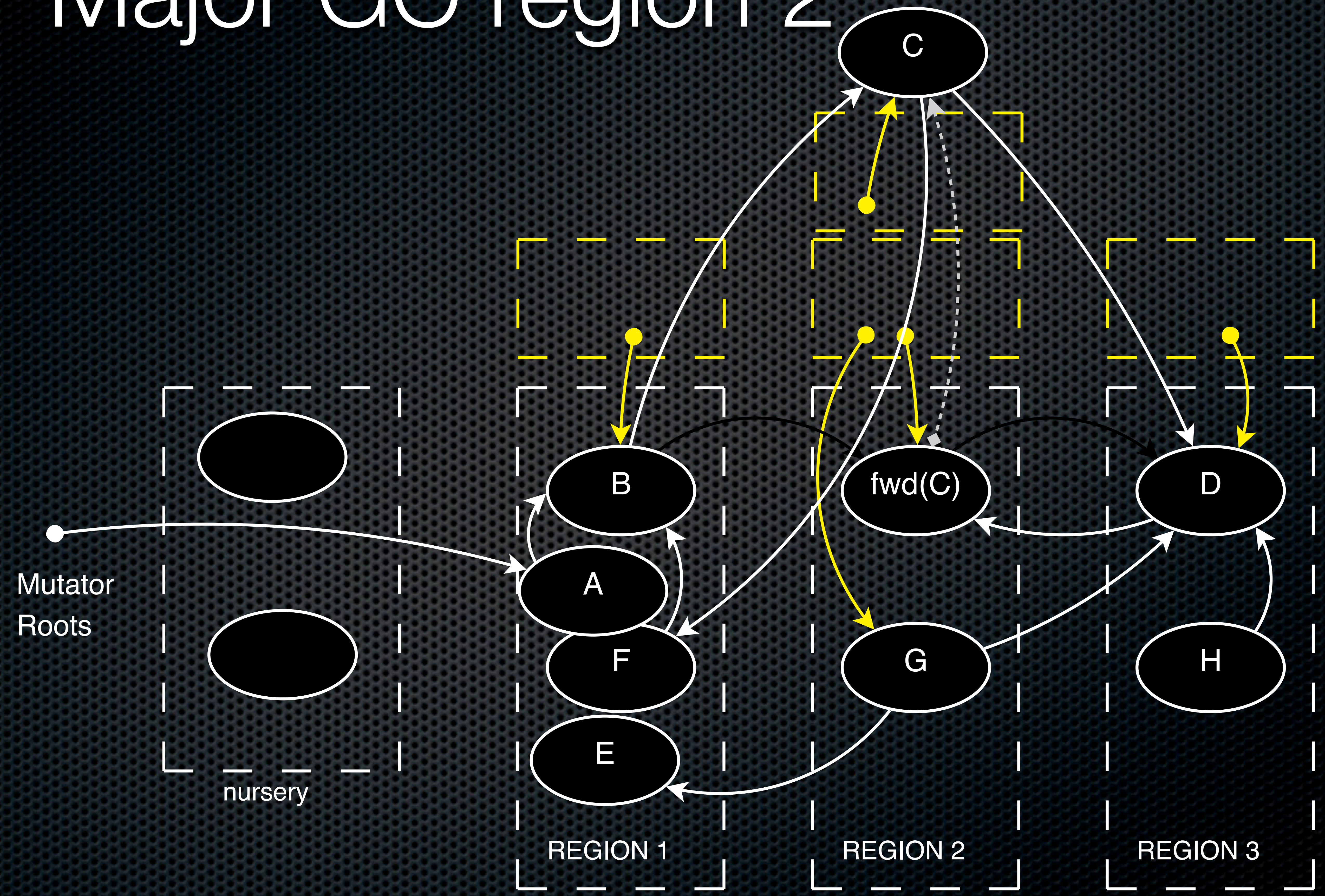
Explain the yellow box by saying GC is going to be updating the remset structure.
Now we evacuate the nursery (where nothing happens in be alive in this simplified example) as well as region 2. No mutator roots point directly into region 2, but there are objects in other regions, like B in region 1 (which we discover in region 1's remembered set). So we have to evacuate C.

Major GC region 2



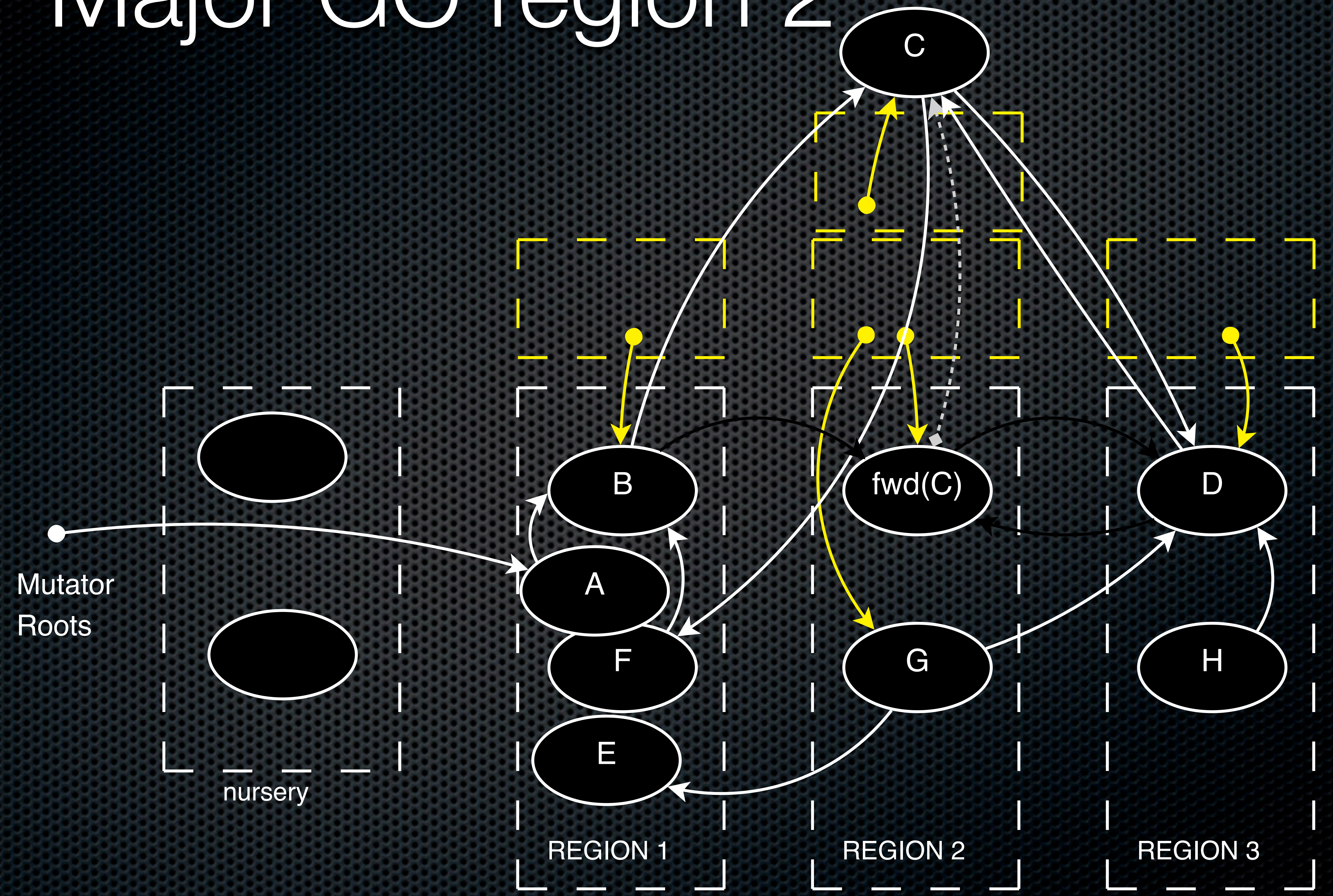
During the evacuation of C, we have to scan it for any objects it points to within region 2. That scan discovers region-crossing pointers, so C has to be kept in the new remembered set for region 2.

Major GC region 2



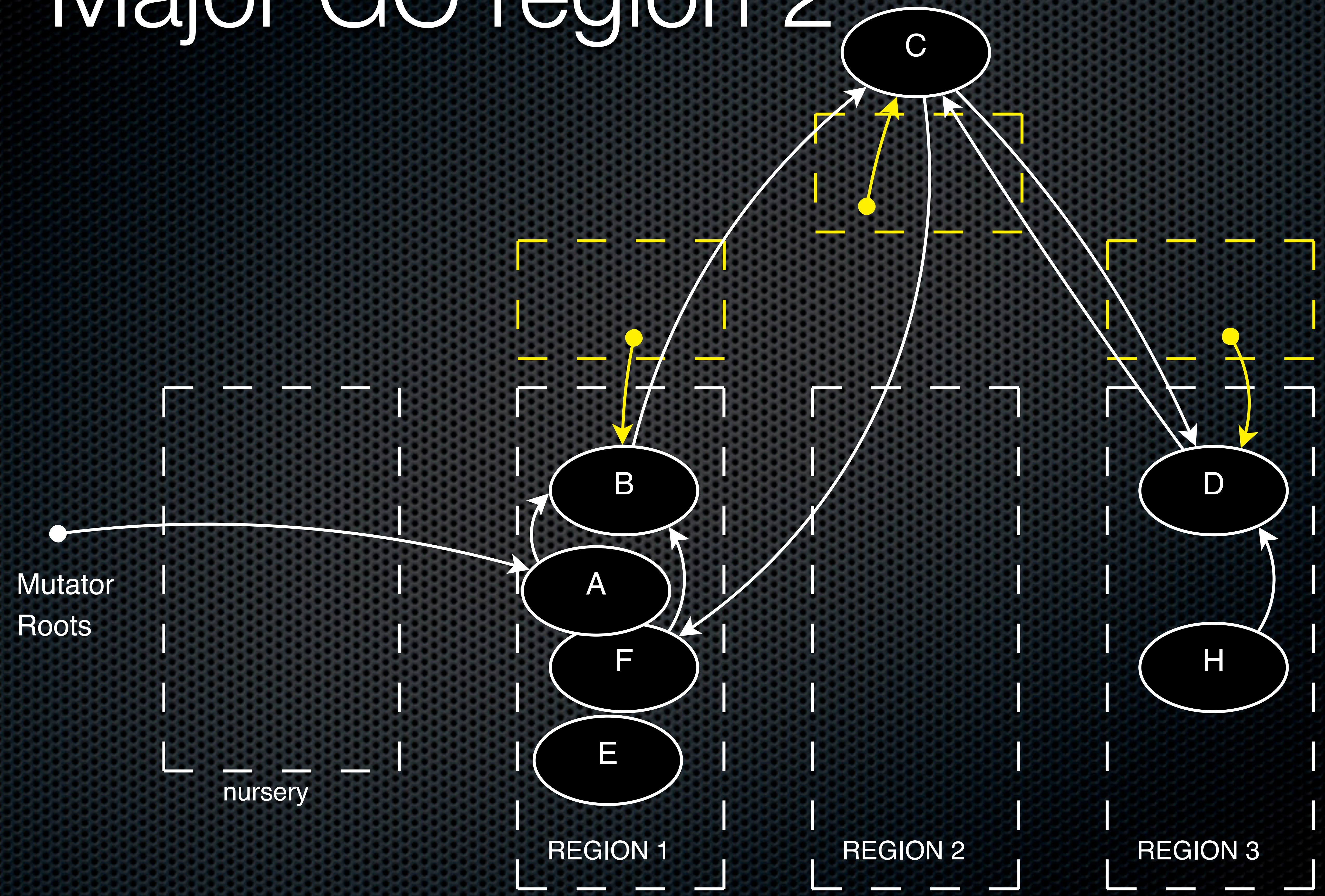
And we also scan the other remembered sets; thus we discover that D’s reference to C needs to be updated.

Major GC region 2



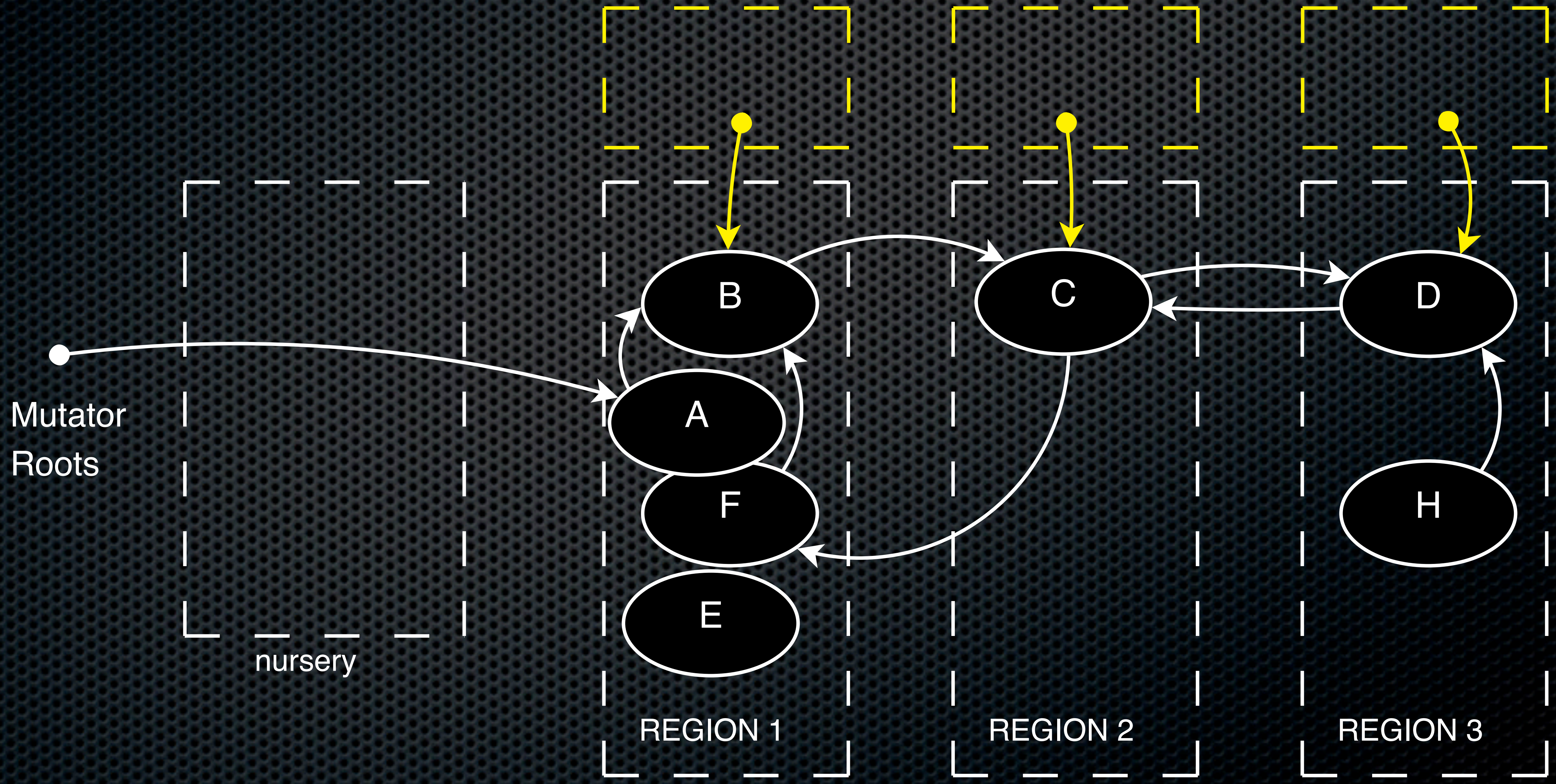
Now the memory for the nursery, the region 2's portion of the heap, and region 2's remembered set can be reclaimed.

Major GC region 2

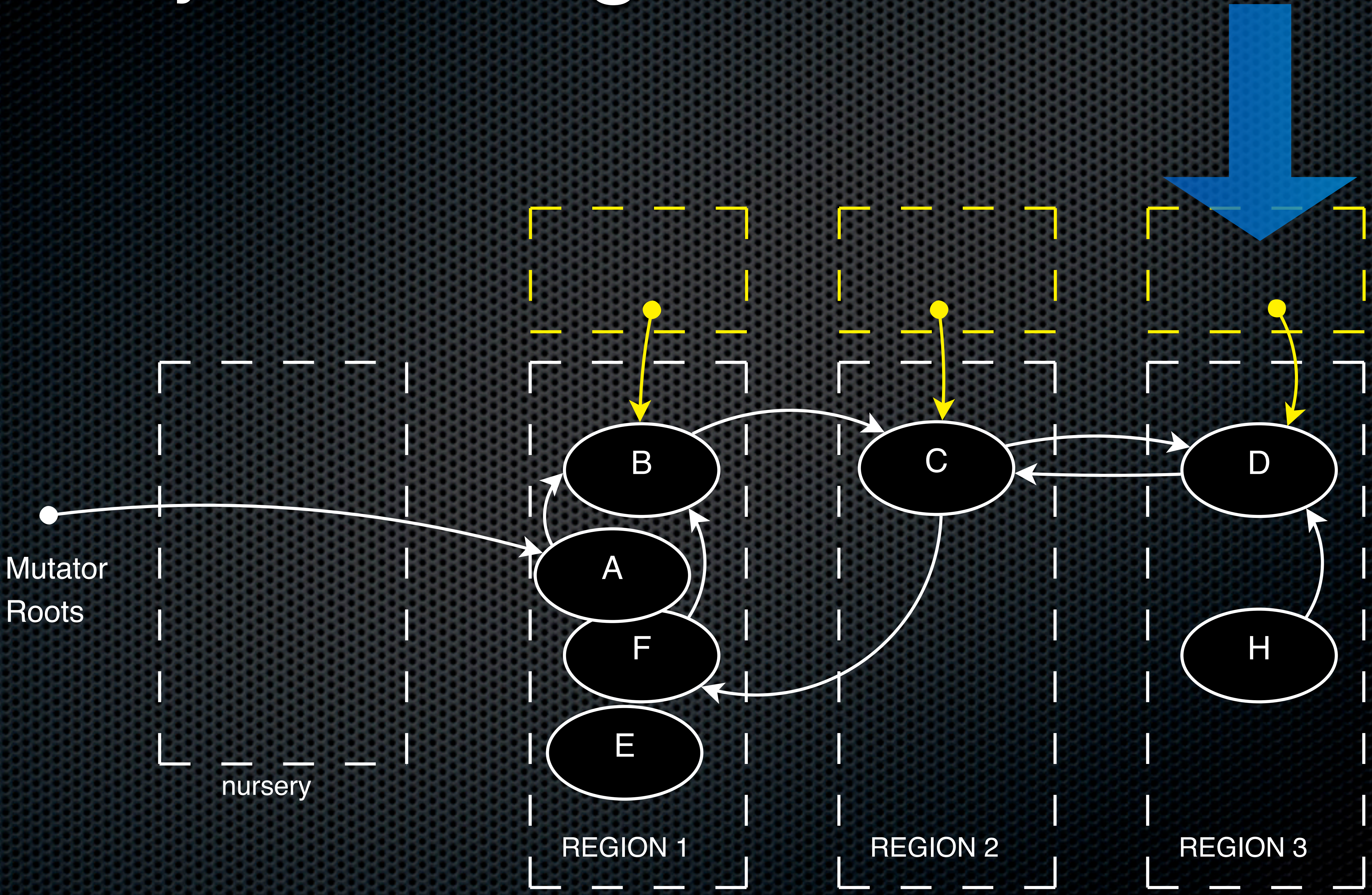


And the newly copied objects can be associated with region 2.

Major GC region 2

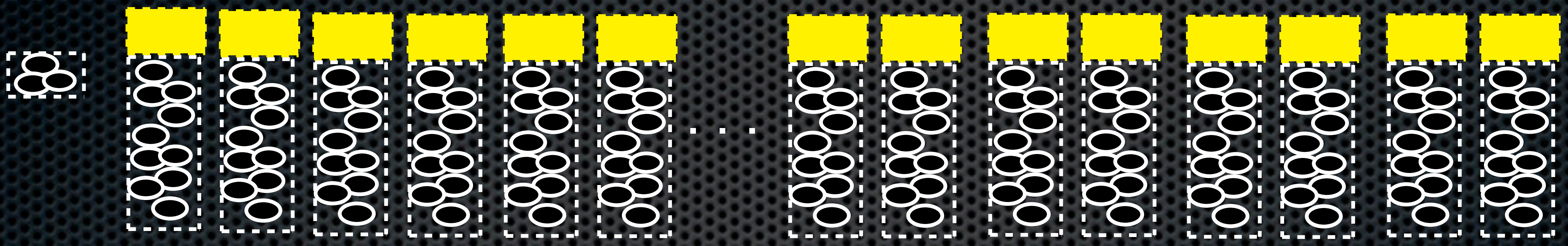


Major GC region 2



Problem

Remembered Sets: Problem



- Each remembered set potentially holds O(R) objects
- Collecting one region generally requires information from O(N/R) remembered sets

Remembered Sets: Problem

- Collecting one requires traversing remsets for *all* others
- Traversal takes worst case Θ(N) time
  - Cannot perform during collection pause without violating O(R) bound on pause time
- Points-into remsets of Garbage-First counter this ...
  - ... but at unacceptable worst-case space cost

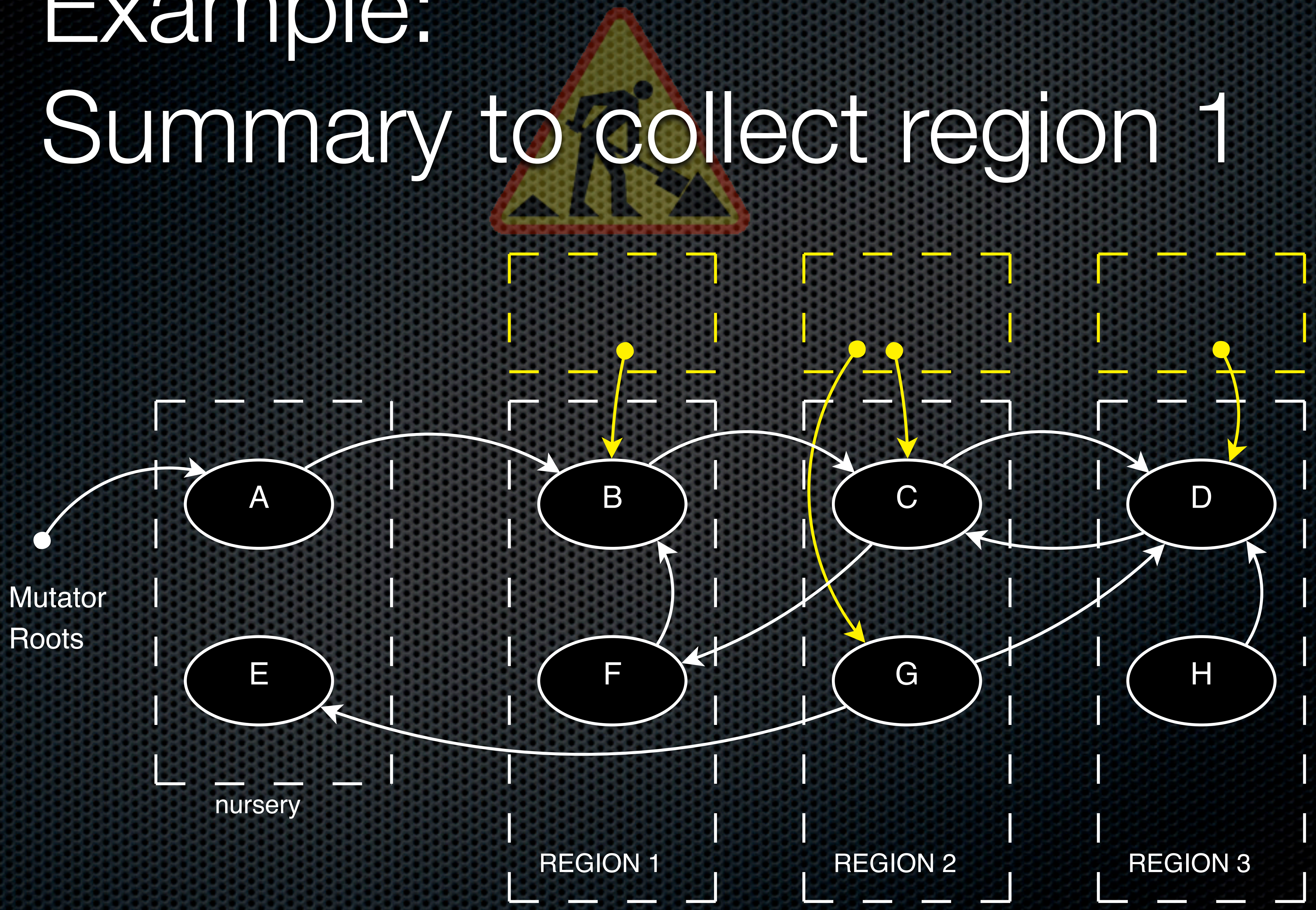
[[While one expects the remembered sets to be small in practice, in general every object in a region could have region-crossing pointers.]]

Remembered Sets: Problem

- Leverage points-into remset structure?
- Idea: do not maintain complete points-into remset
- Instead construct similar structures “just in time”
- Call these *points-into summaries*
- Concurrently scan remembered sets; discard summary after each collection

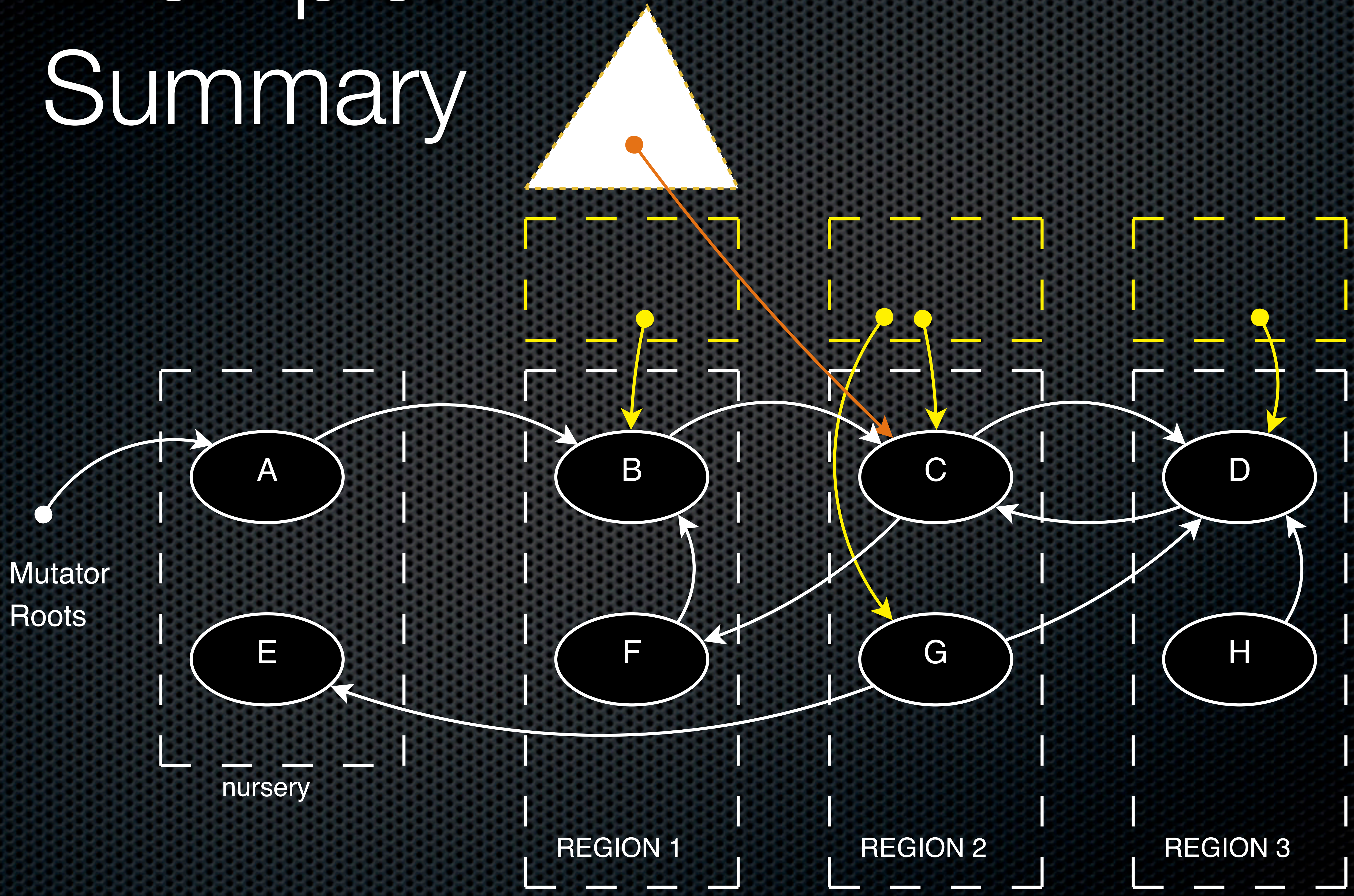
Do not maintain complete points-into for all regions through entire computation

Example:
Summary to collect region 1



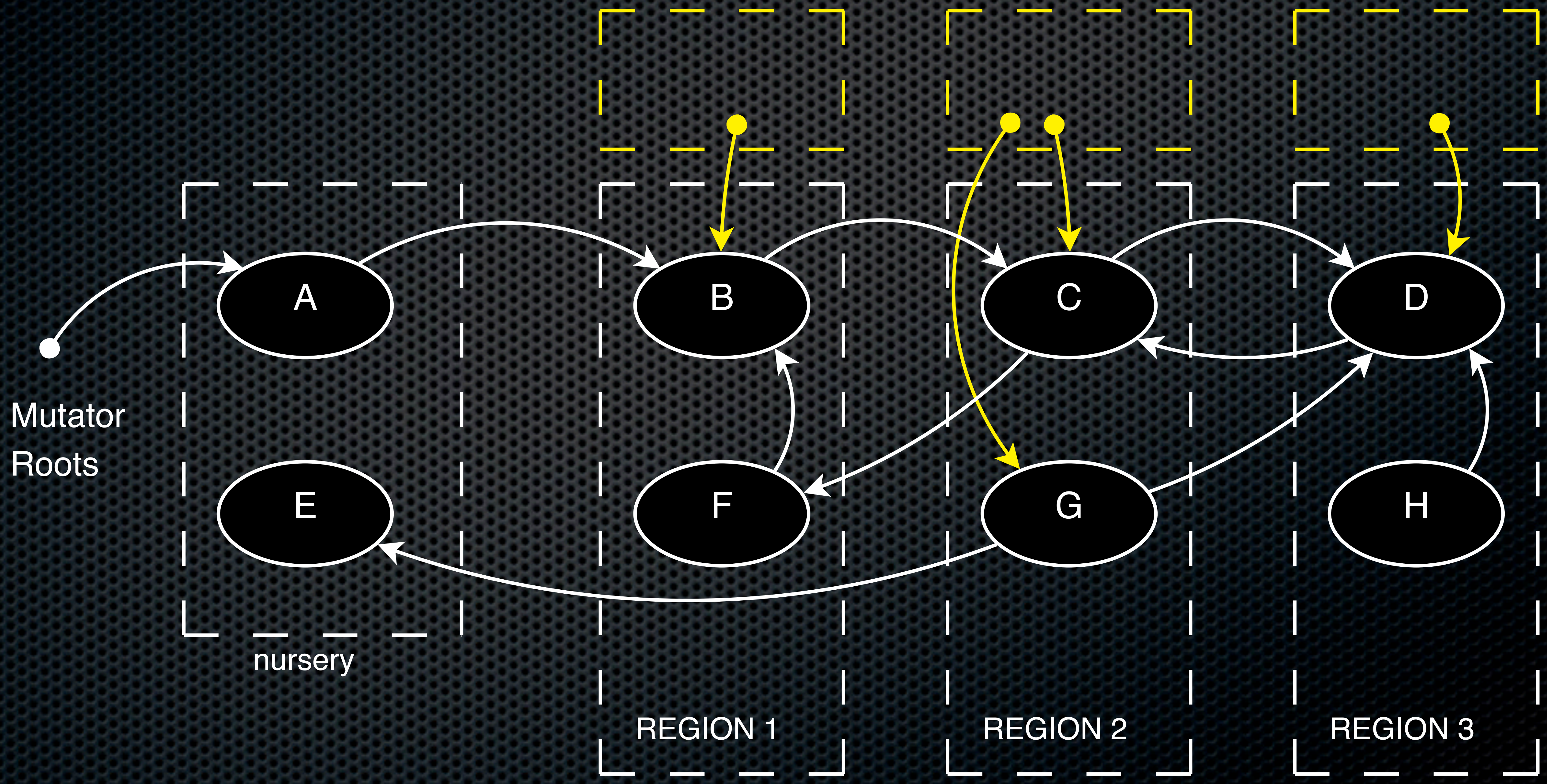
If policy said next major gc is rgn1. Scan remembered sets of regions 2 and 3, to find. . .

Example: Summary



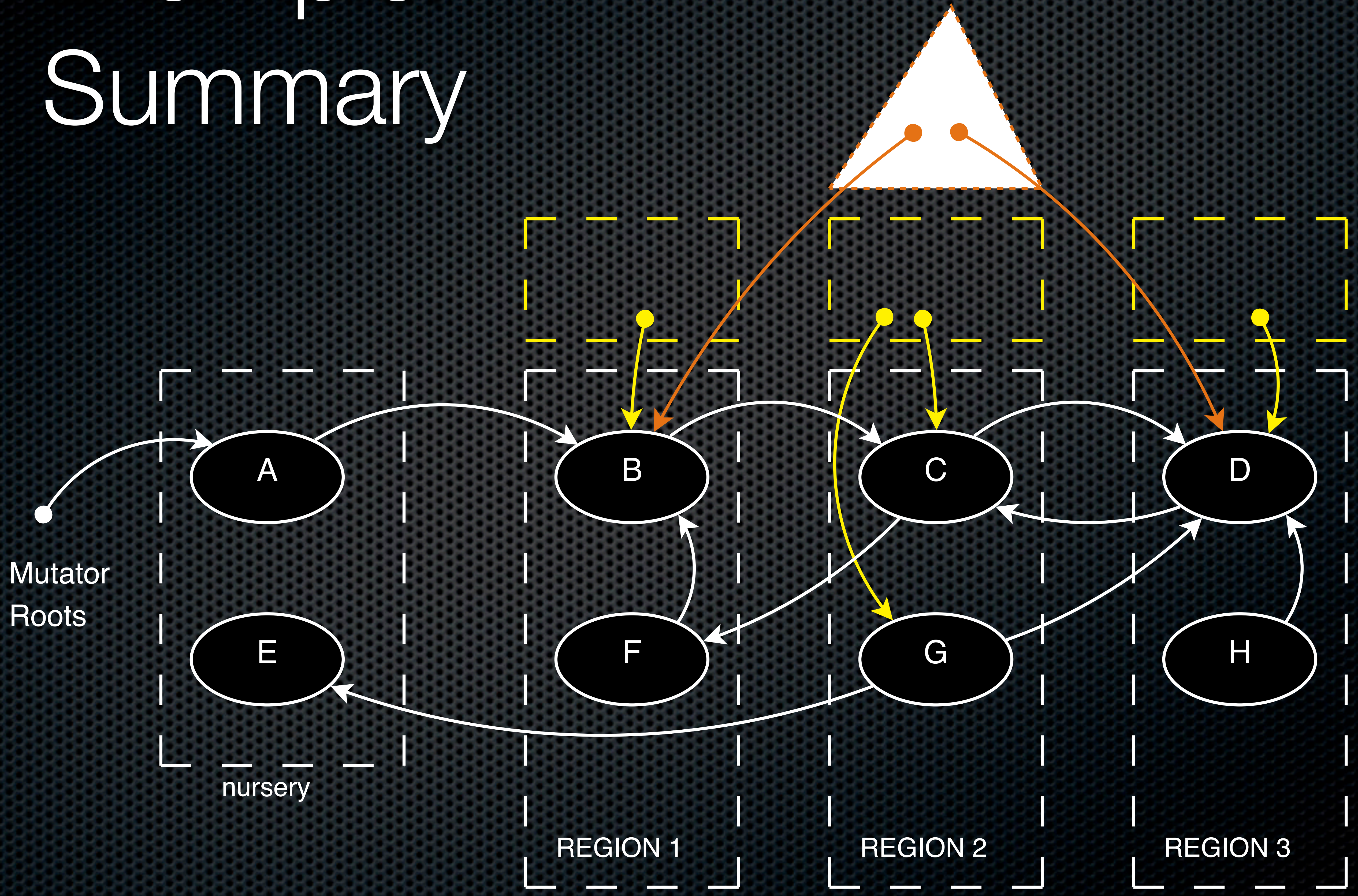
Object C points into region 1

Example:
Summary to collect region 2



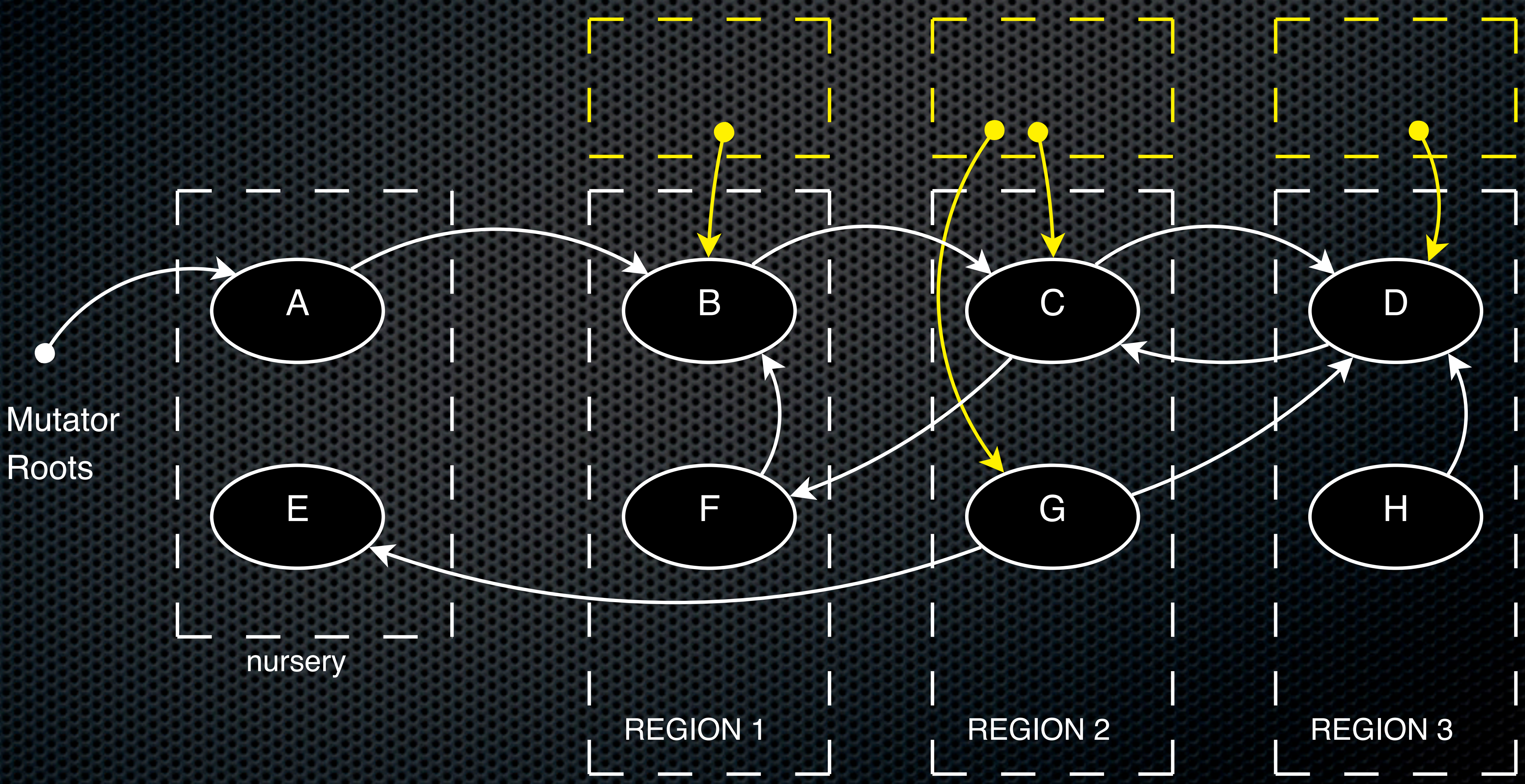
OTOH, if policy said next major gc is rgn2. scan remembered sets of regions 1 and 3, to find. . .

Example: Summary



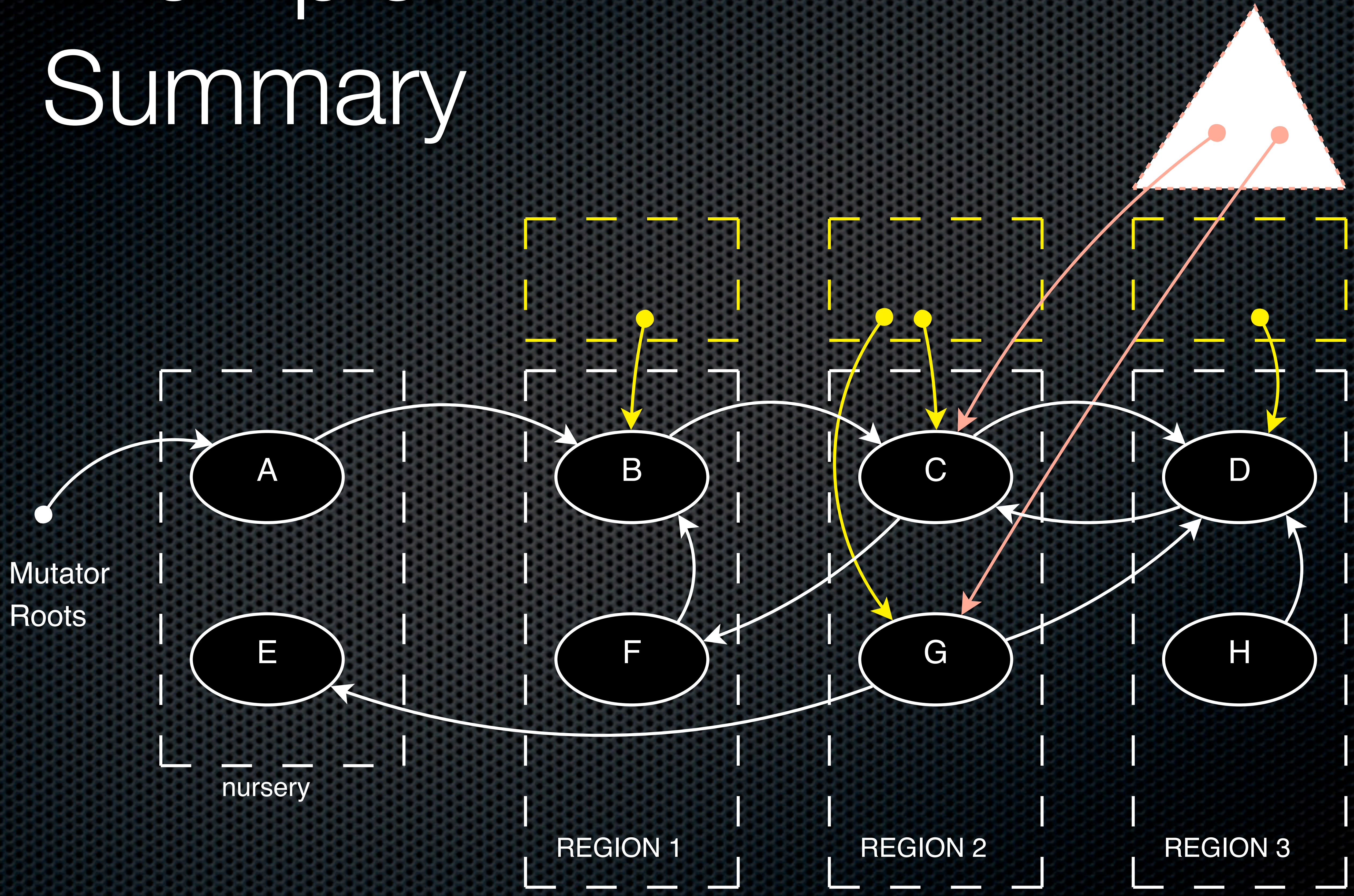
Objects B and D point into region 2

Example:
Summary to collect region 3



OTOH if policy said next major gc is rgn3. scan remembered sets of regions 1 and 2, to find. . .

Example: Summary



Objects C and G point into region 3.
The strategy of Just-In-Time summarization does raise some questions. . .

Just-in-Time Summarization

- Question: Can’t such summaries grow arbitrarily large?
- Answer: Cap how large the summaries can grow (called the *wave-off* limit)
  - If summary reaches wave-off limit then abandon attempt
  - Wave-off limit bounds size and time

[[optional vocal: How does this solve anything?]] Say “popular” explicitly and describe the problem.
Wave-off is a military term; refers to an action to abort a landing, initiated by the someone on the ground (or the pilot) at their discretion. (Another analogy: the collector is forcing relocation of the regions selected for collection, but will abort relocation if a region is a “high-profile” target.)
[[This answer raises a new question. . .]]

Just-in-Time Summarization

- Question: With wave-off, can't *all* regions have large summaries, and thus none are collectable?
- Answer: Choose appropriate wave-off limit
  - At any instant, heap has ≤ N references into N/R regions
  - With wave-off limit = 2R, at most 1/2 of the regions could reach the limit; the other half are collectable.

This argument generalizes to larger choices for the wave-off limit
[[The wave-off limit is bigger than you might have thought it would be (2R is a lot of references)]].

Concurrent Summarization: Core of Algorithm

- Start with K summaries; K = Θ(N/R)
- Thus K mutator activity periods between K major gc's to traverse remsets and construct ≥ K new summaries
- Usual case: single traversal of N/R remembered sets
  - Distribute objects into ≥ K new points-into summaries

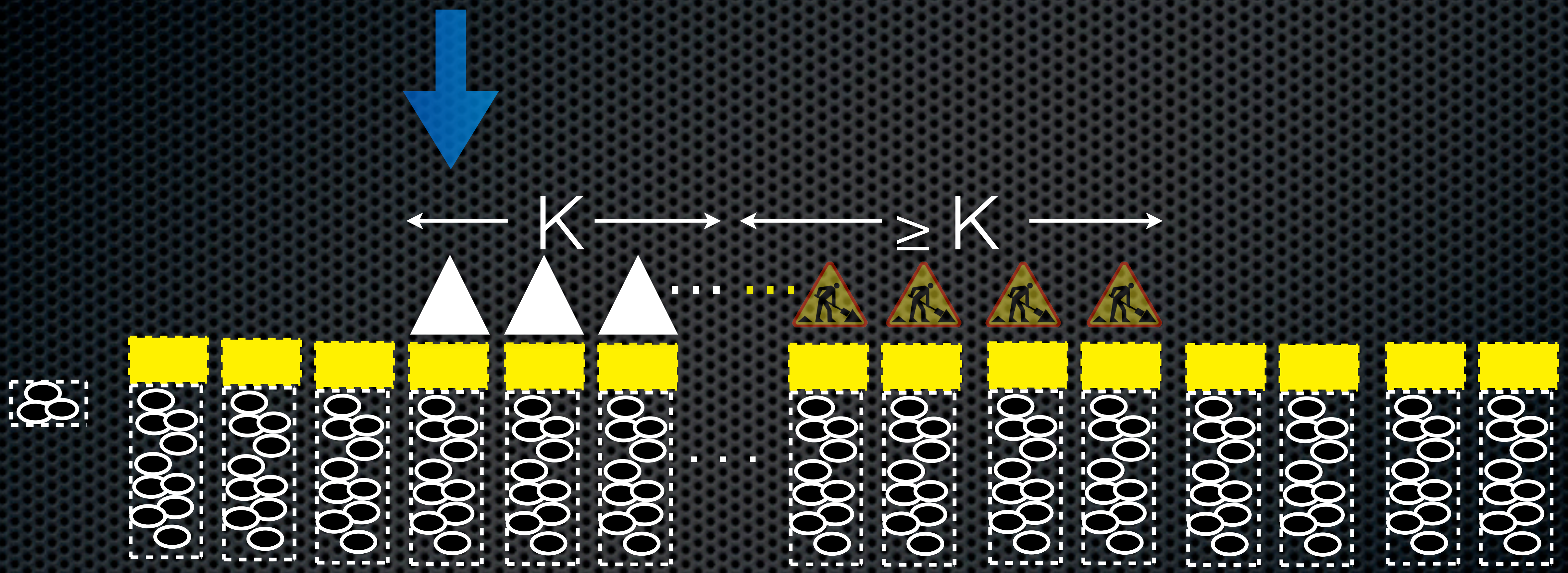
K *proportional* to N/R!
[[the K factor is going to dictate one time/space tradeoff between how much storage is given to summaries and how much time we have to spend constructing summaries...]]

Concurrent Summarization: Core of Algorithm

- Usual case: single traversal of N/R remembered sets
- Unusual (worst case) scenario: system hits wave-off limit on (many) summarization attempts
  - Build more than K summaries on a traversal
  - Do *not* schedule work of one traversal evenly across K mutator activity periods; instead schedule to permit multiple traversals

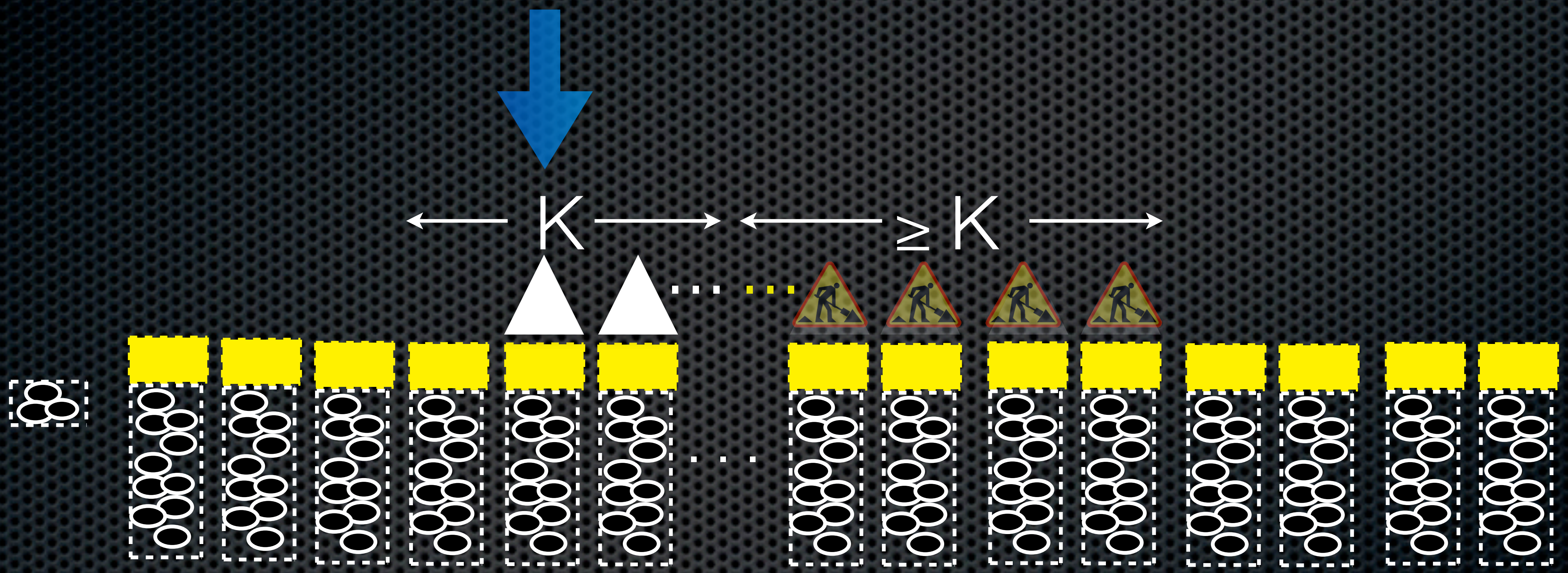
There is a time/space tradeoff here; devoting more space to summary construction permits a less aggressive summarization schedule. Note that the schedule does need to finish in a bounded number of traversals.

Concurrent Summarization



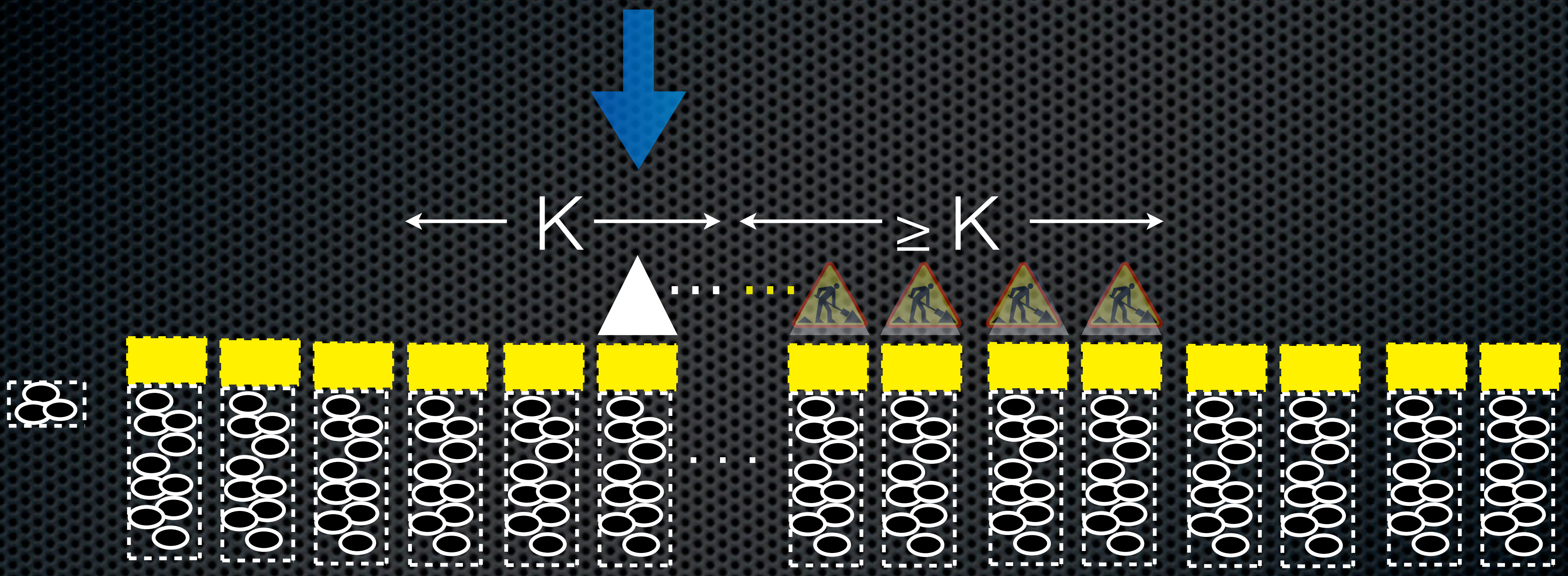
remember: K = Θ(N/R)

Concurrent Summarization



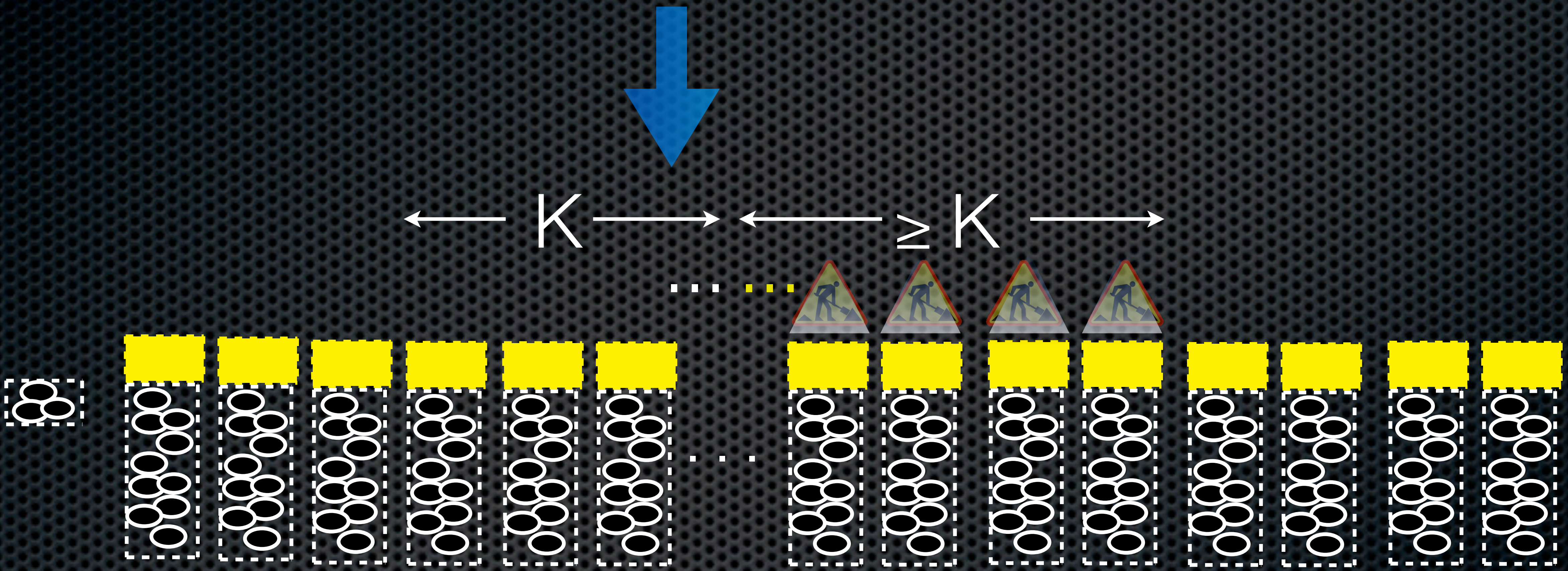
remember: $K = \Theta(N/R)$

Concurrent Summarization



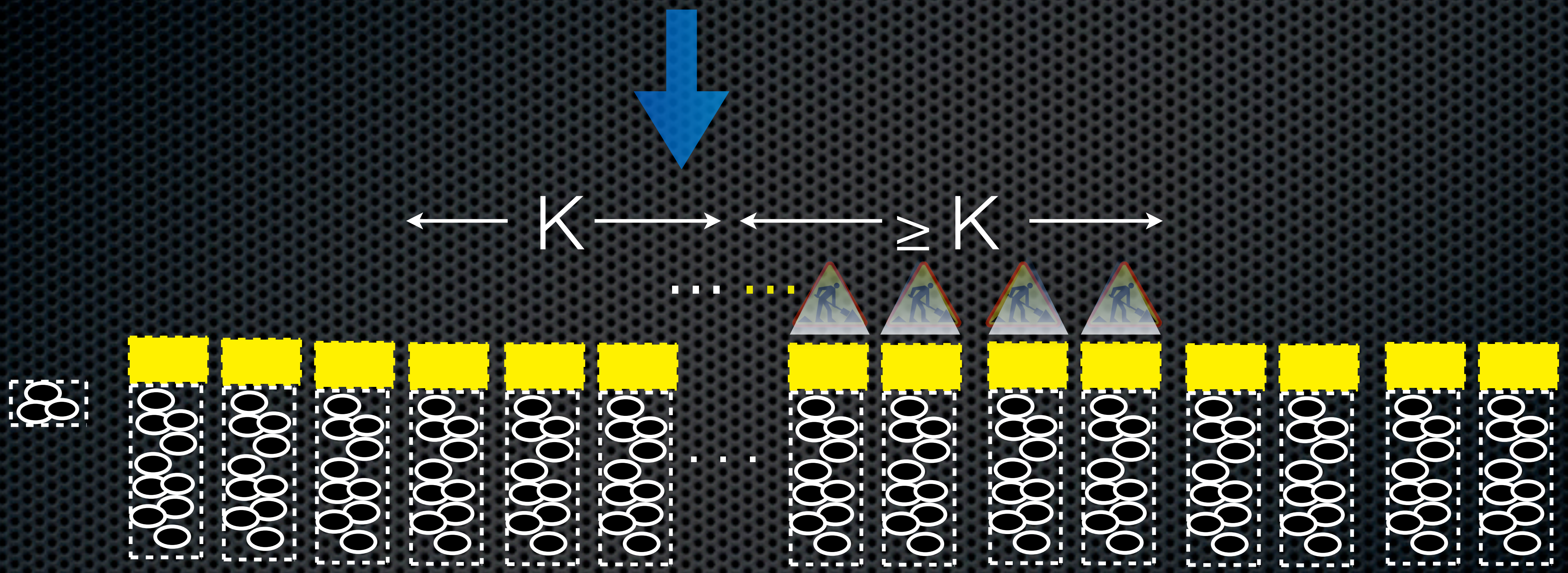
remember: K = Θ(N/R)

Concurrent Summarization



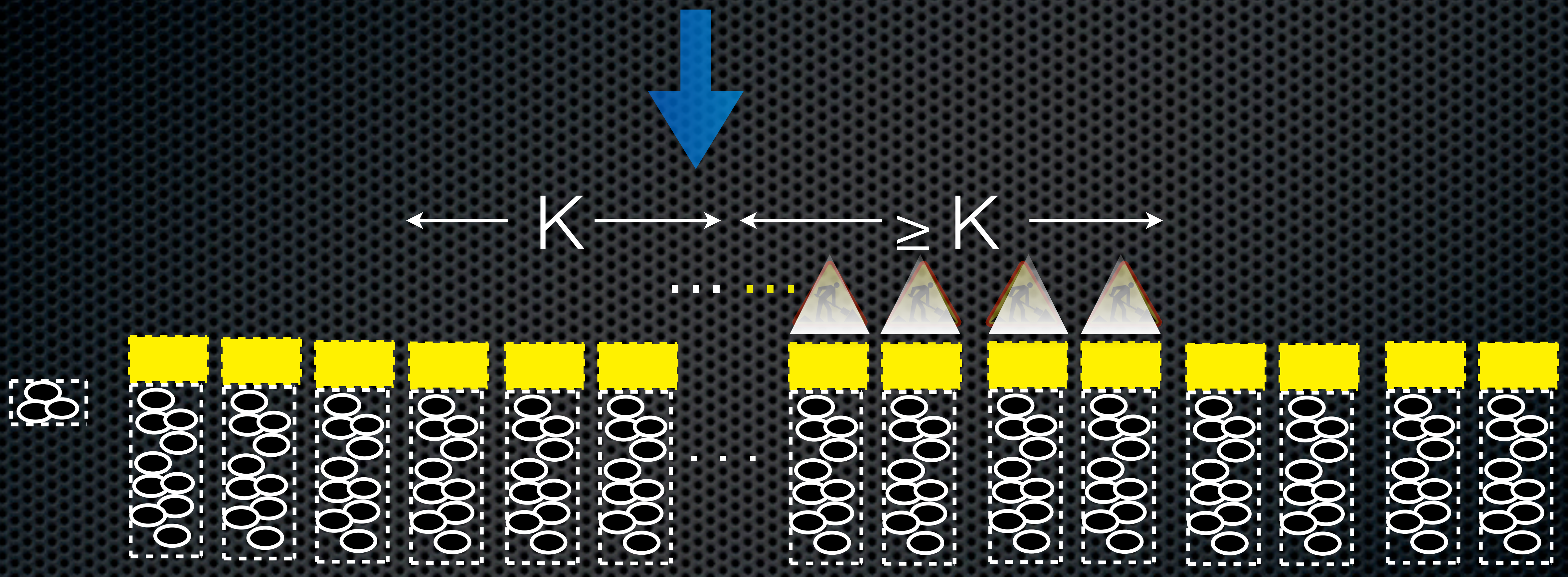
remember: $K = \Theta(N/R)$

Concurrent Summarization



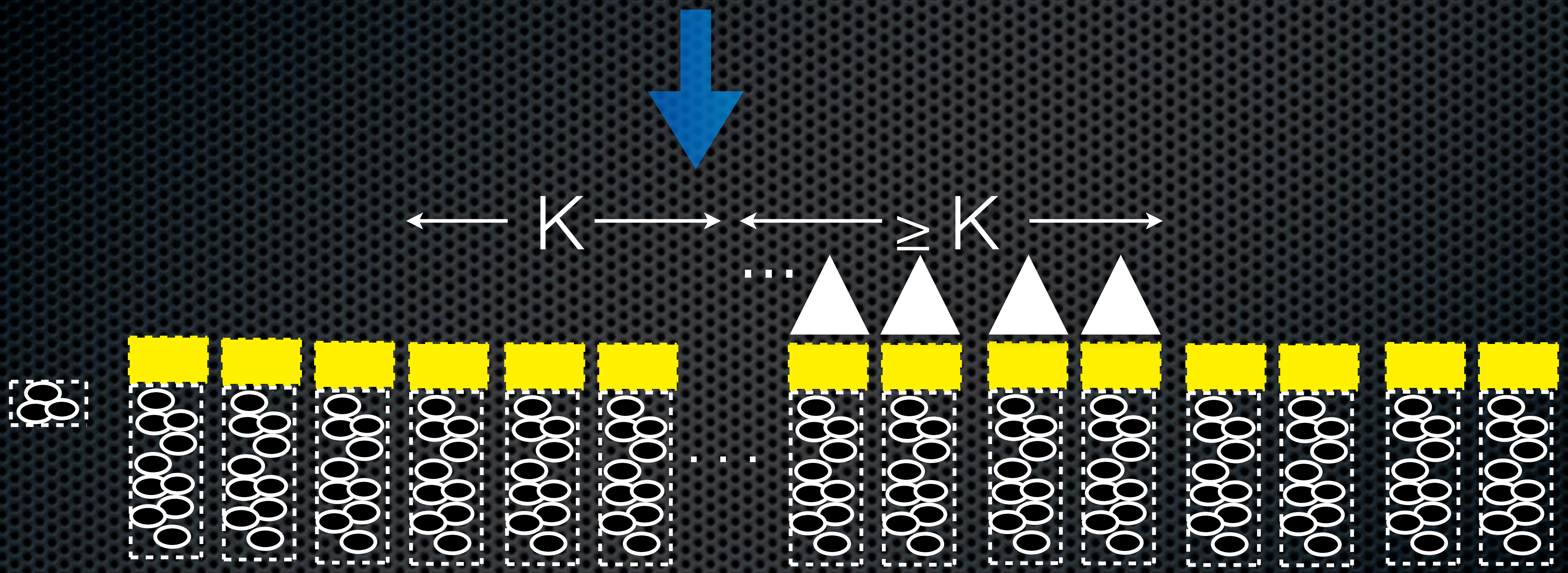
remember: K = Θ(N/R)

Concurrent Summarization



remember: K = Θ(N/R)

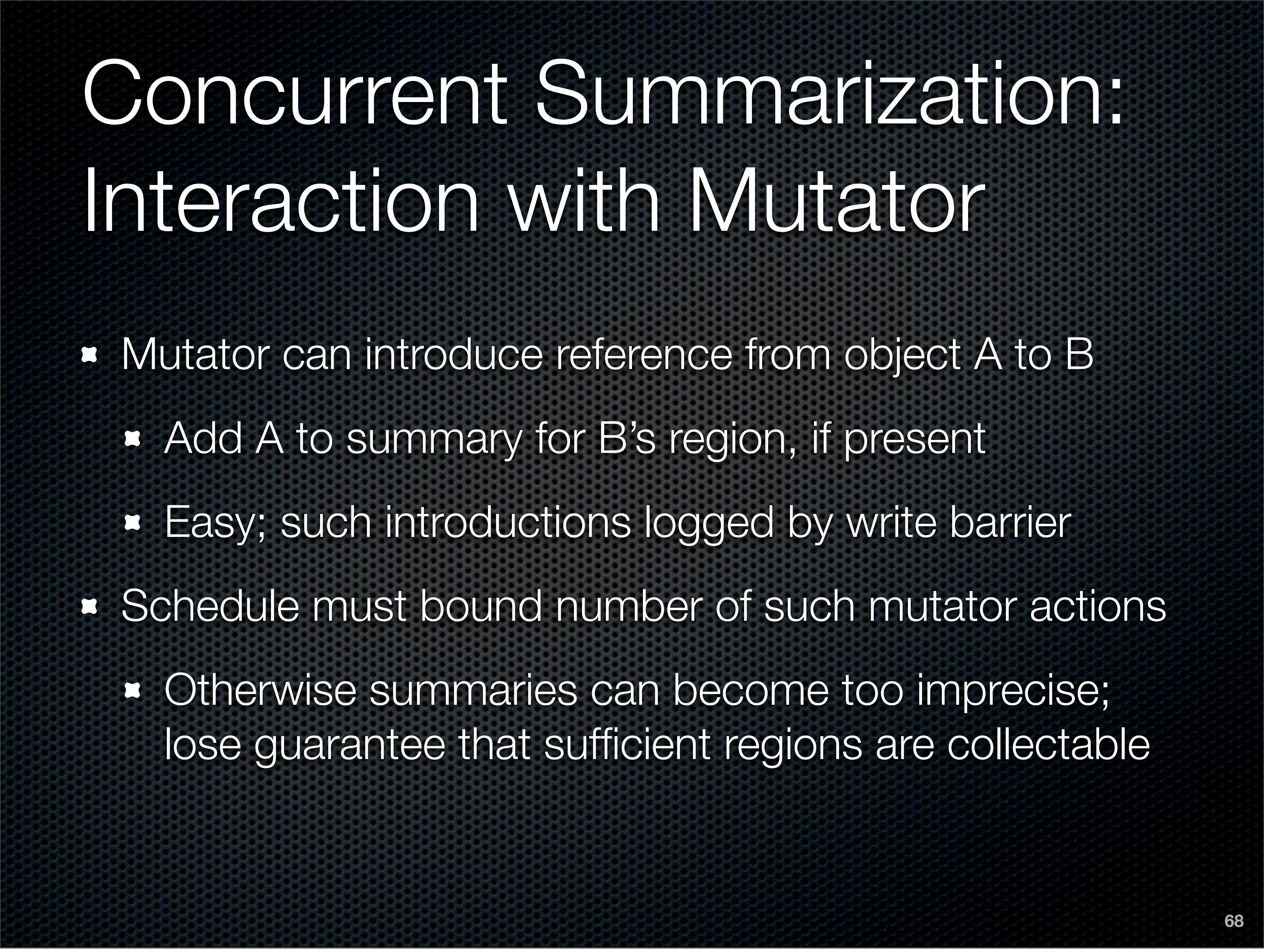
Concurrent Summarization



remember: K = Θ(N/R)

Concurrent Summarization: Core of Algorithm

- There are some implementation subtleties
- Summarization progresses concurrently with mutator
- Summarization will be interrupted by collector activity



Concurrent Summarization: Interaction with Mutator

- Mutator can introduce reference from object A to B
  - Add A to summary for B’s region, if present
  - Easy; such introductions logged by write barrier
- Schedule must bound number of such mutator actions
  - Otherwise summaries can become too imprecise; lose guarantee that sufficient regions are collectable

68

[[This last bullet implies that we have a MAX mutator utilization bound, which might be a bad thing in the eyes of some...]]

Concurrent Summarization: Interaction with Collector

- Collector moves live objects that point to other regions
  - Collector must *update* corresponding summary entries
- Collector reclaims dead objects
  - Collector must *clear* corresponding summary entries
- Need both to avoid dangling pointers to freed storage in points-into summary structures

Note: these interactions are analogous to the handling of weak-references; entries are not roots, but they do need to be kept consistent with object migration.
[[Second main bullet indicates that maybe “Just-in-time” is not appropriate term for summary construction.]]

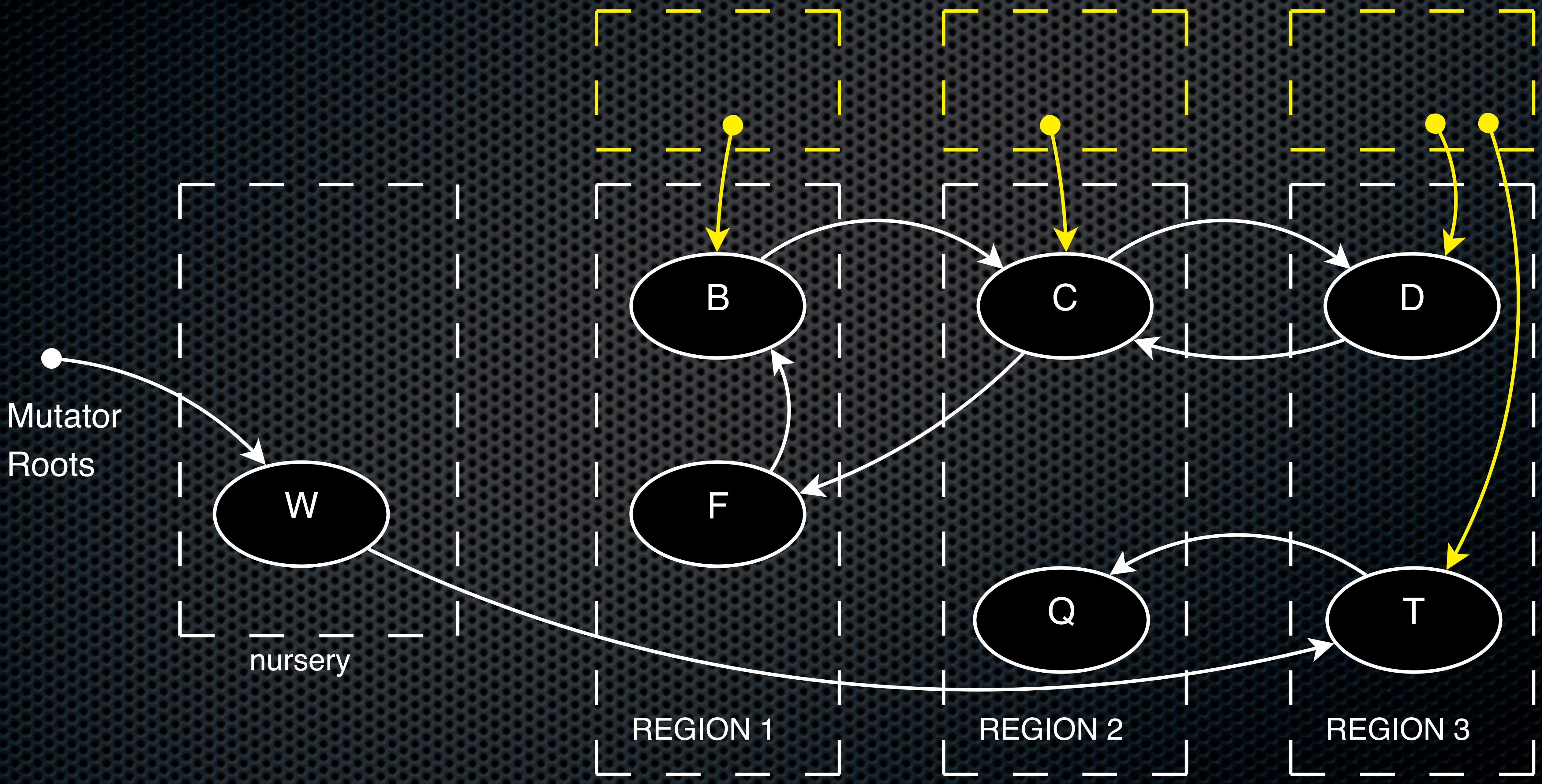
Ensuring Completeness

Completeness

- The problem: garbage cycles that cross regions

Say: “Nothing presented so far solves this problem.”

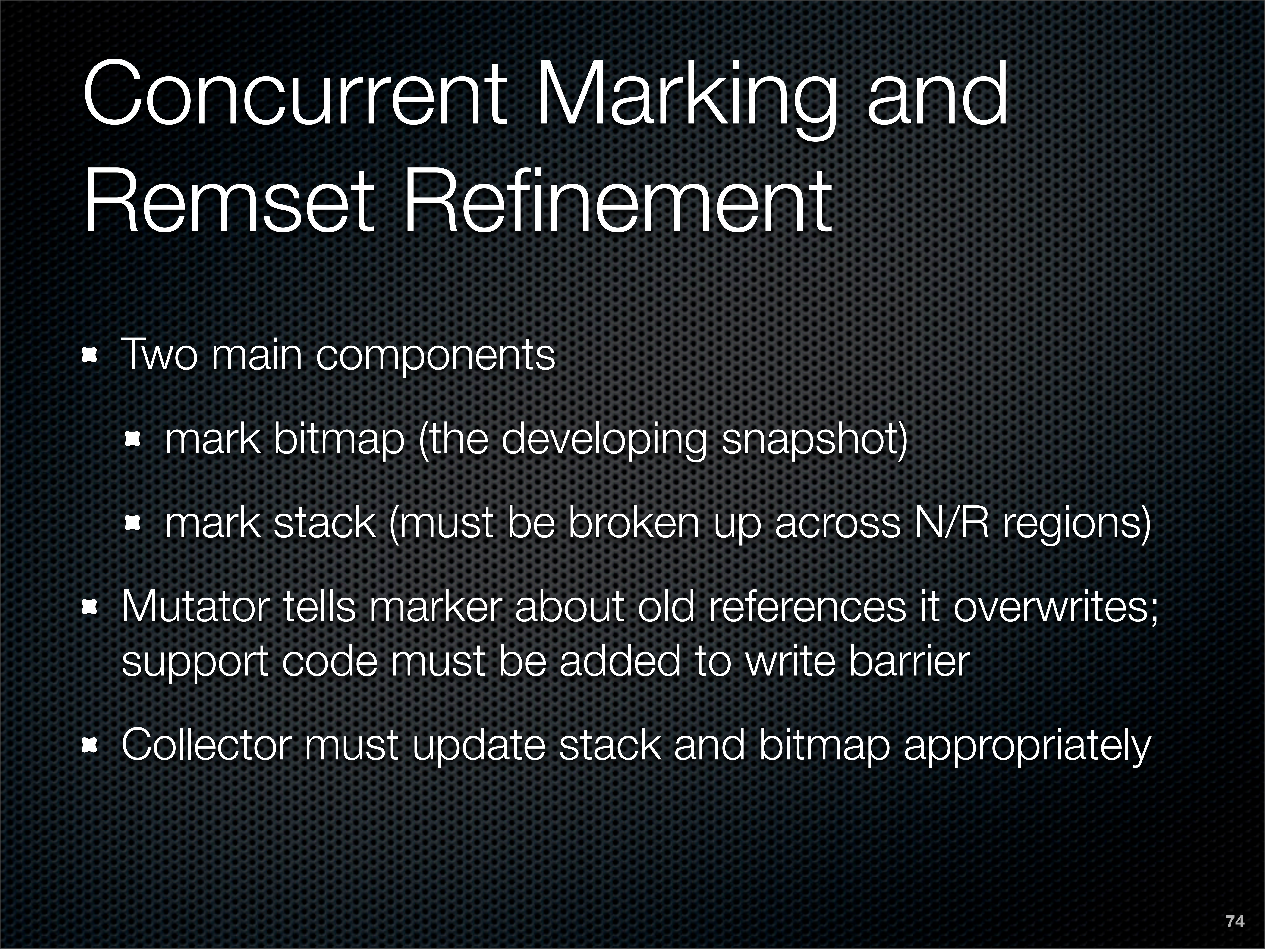
Garbage Cycle that Crosses Regions



Concurrent Marking and Remset Refinement

- Old idea [Yuasa '90]: periodically "snapshot" heap; commonly called "Snapshot-at-the-Beginning" (SATB)
- Our use inspired by Garbage-First collector
- Snapshot developed concurrently with the mutator
  - Provides exact information about objects *at time of snapshot*
- Remset Refinement: remove dead objects of snapshot from remembered sets

I think of the "snapshot" as being like a Polaroid picture. The subjects of the photo have usually moved by the time development is complete, but any corpses in the picture will still be dead.
[[Secondary purpose: collection of long acyclic chains against flow of collector.]]
Note that GF used SATB to guide its search for garbage-rich regions; we're just using it for collection completeness [[we thought this would imply the marker could be low-priority... but...]]
Note that marked objects *are* considered for collection!

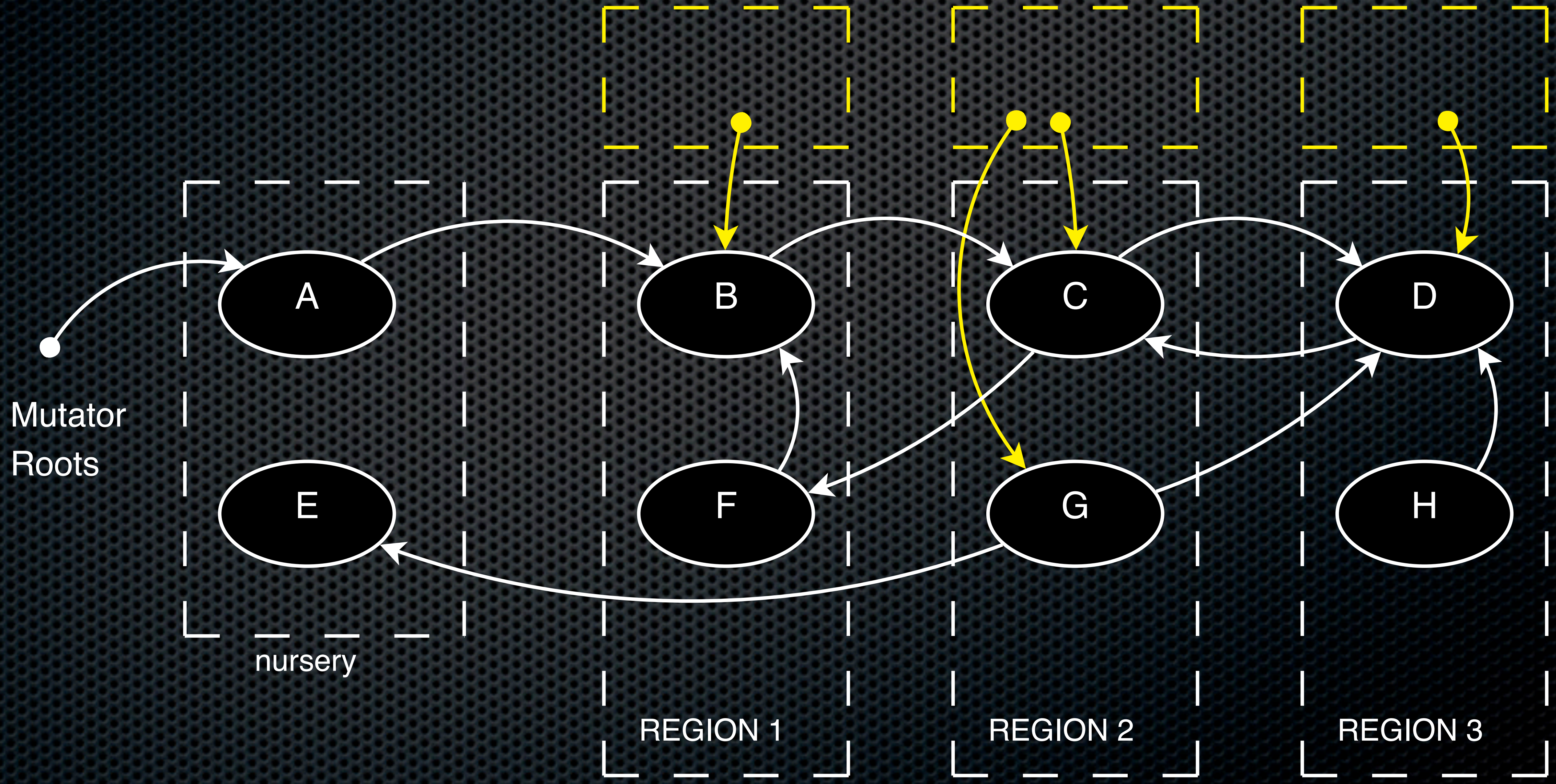


Concurrent Marking and Remset Refinement

- Two main components
  - mark bitmap (the developing snapshot)
  - mark stack (must be broken up across N/R regions)
- Mutator tells marker about old references it overwrites; support code must be added to write barrier
- Collector must update stack and bitmap appropriately

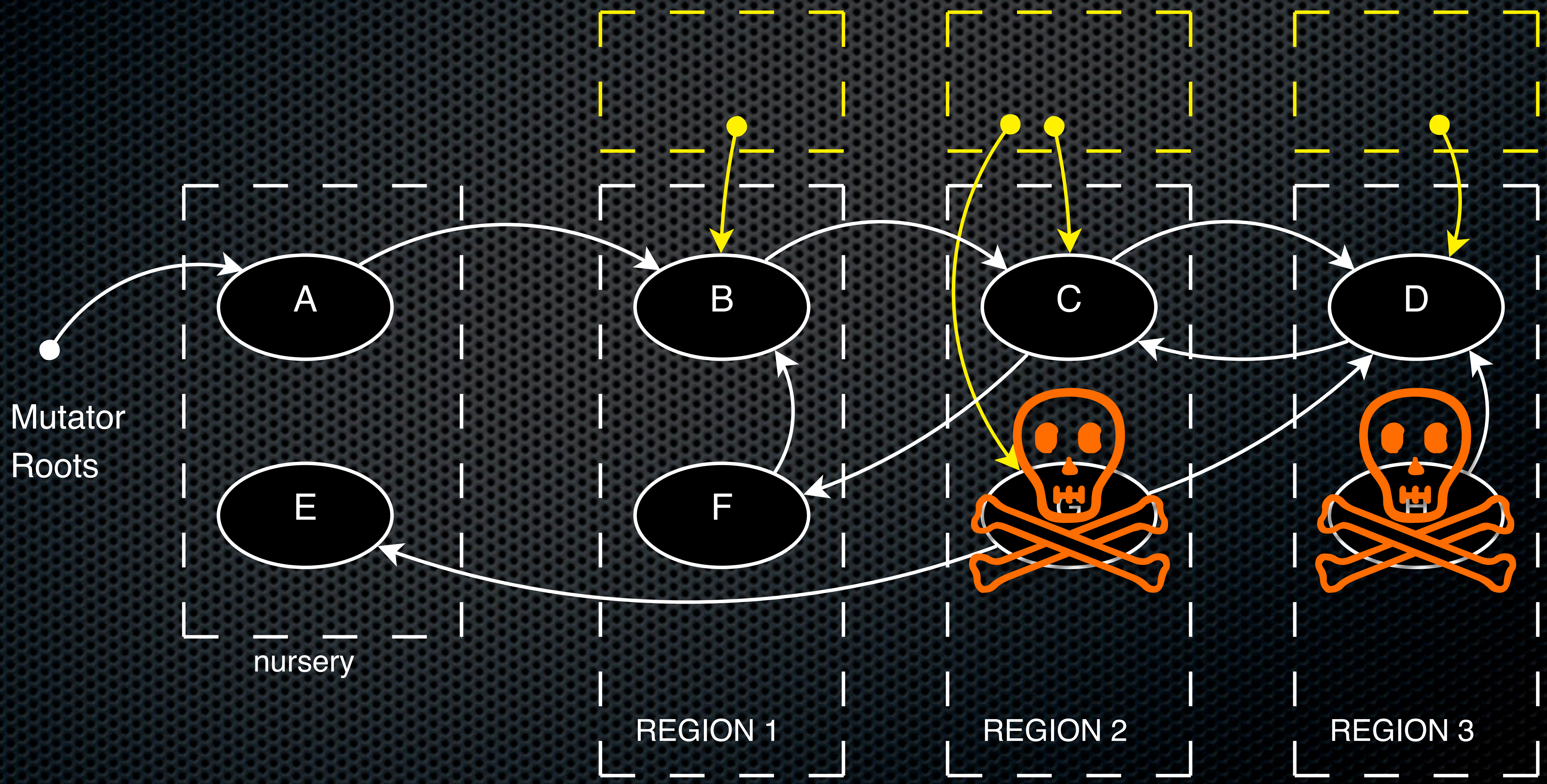
The update to the mark stack and mark bitmap has the same “weak reference” property that I discussed with respect to the summarization structures.

Concurrent SATB Mark and Refinement



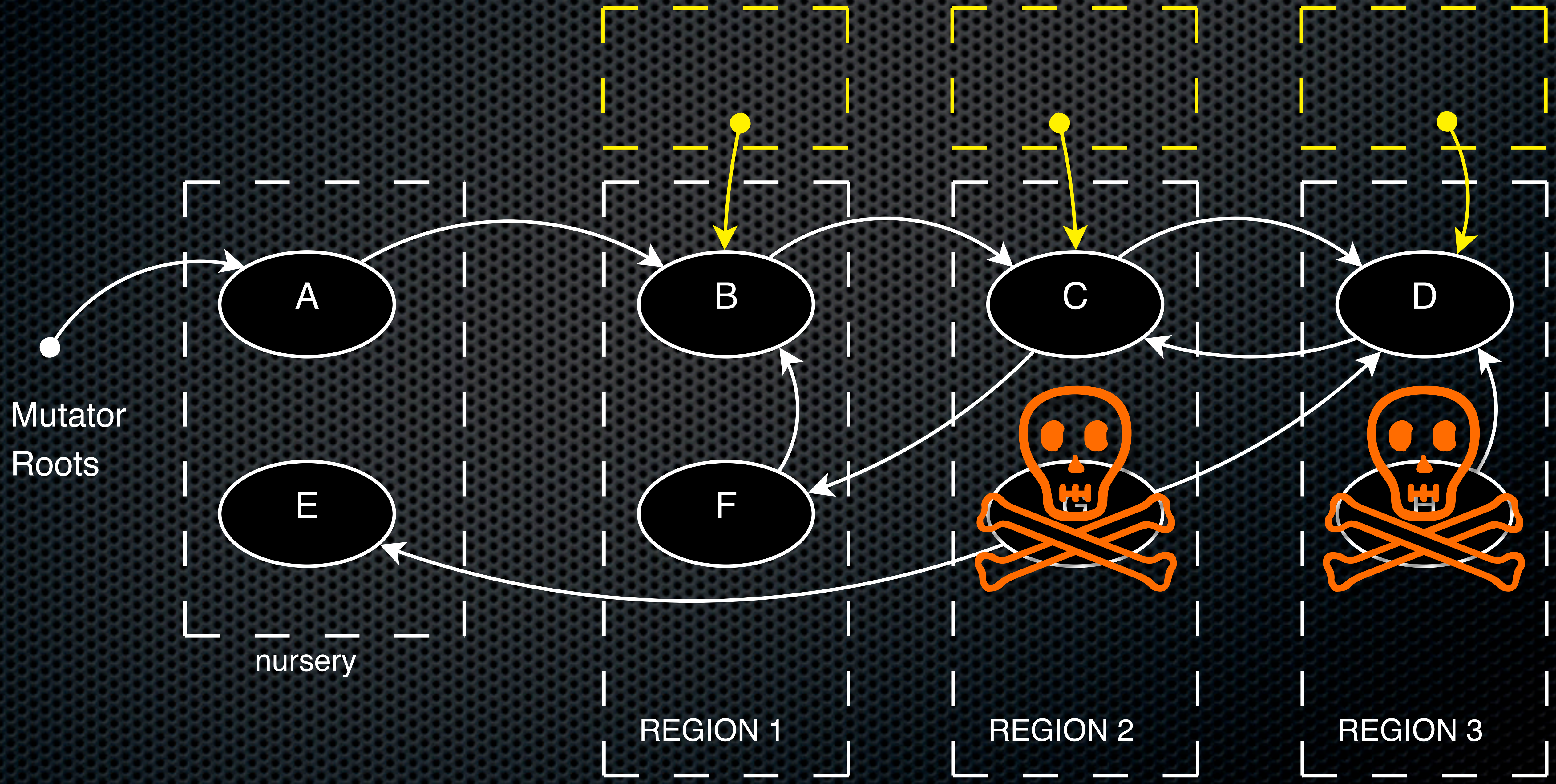
Initial illustration: a snapshot of this heap would indicate that in the regions, the objects { G, H } are dead

Concurrent SATB Mark and Refinement



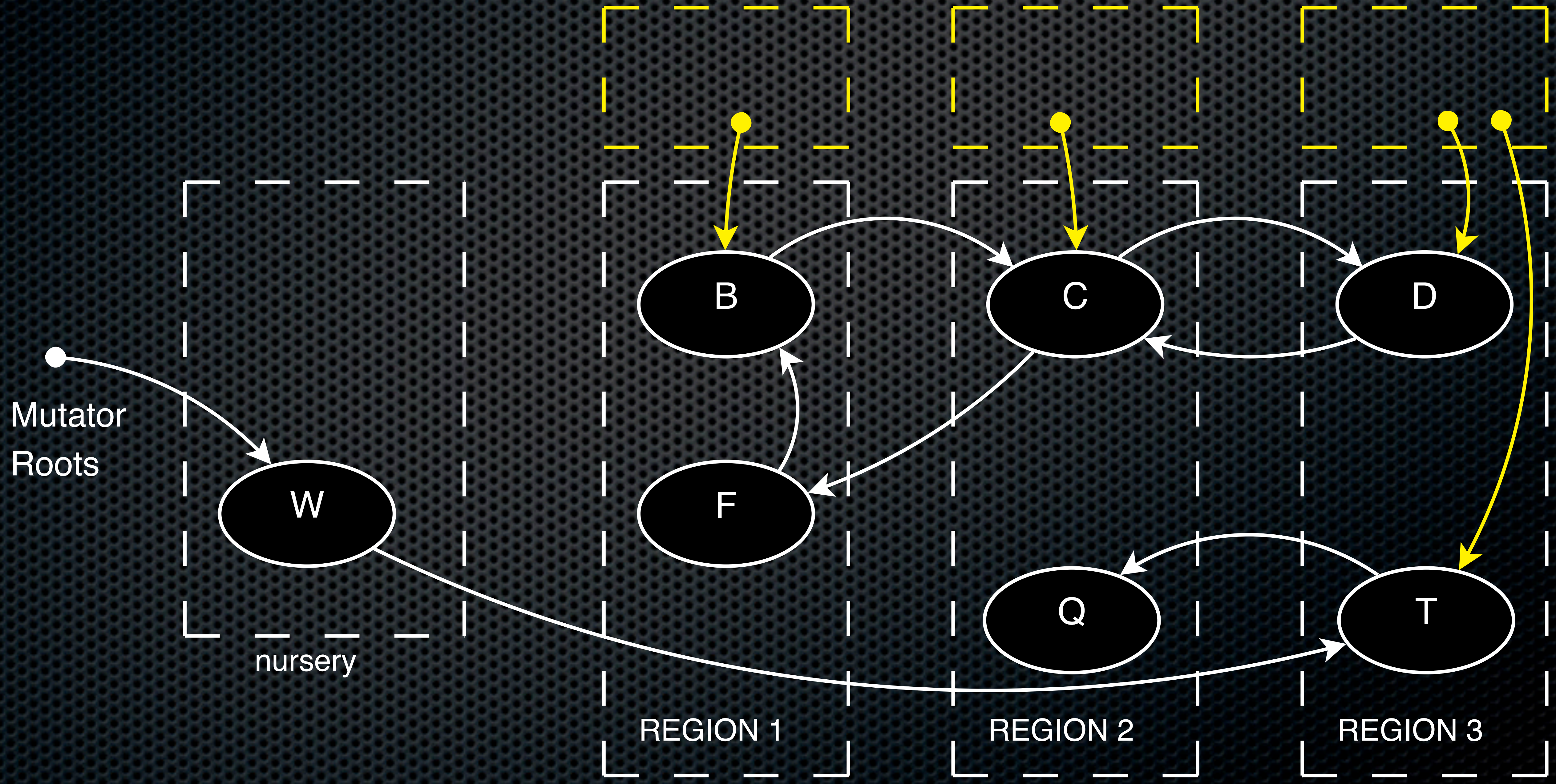
{ G, H } are dead, and therefore G can be removed from the remembered set for region 2.

Concurrent SATB Mark and Refinement



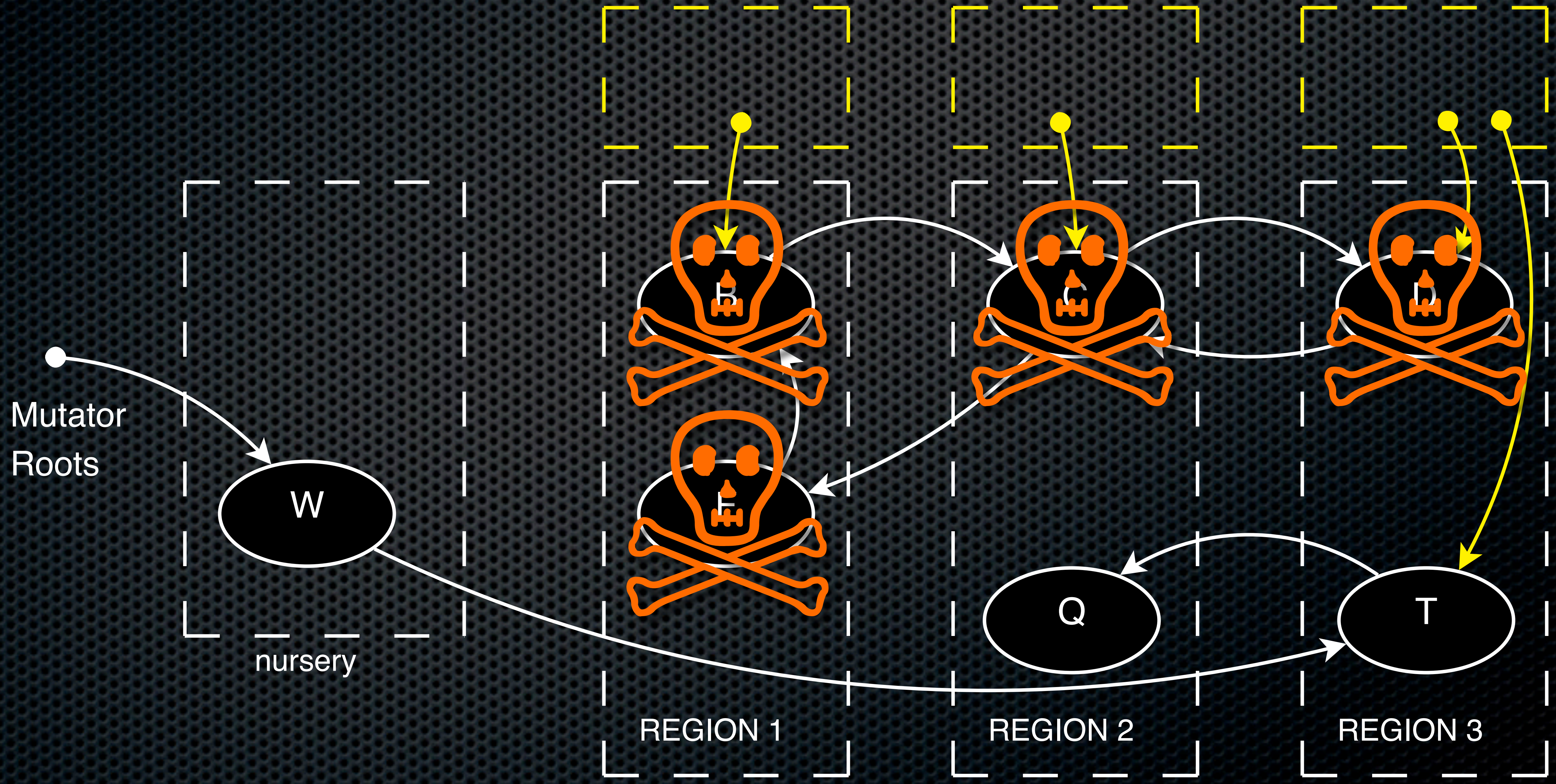
Why does Mark&Refine matter? Well, what happens if A dies during the computation?

Concurrent SATB Mark and Refinement



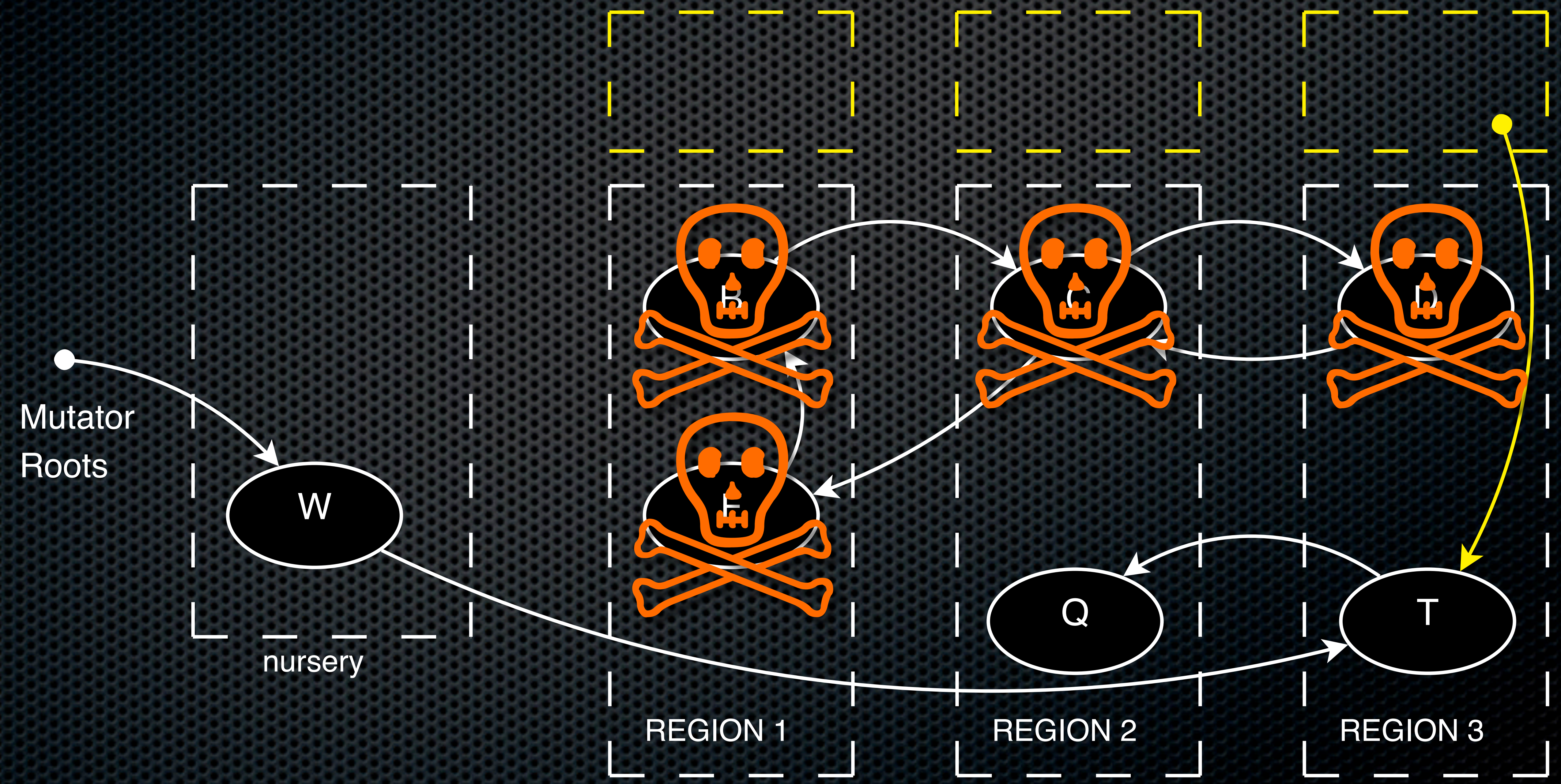
If object A unreachable, objects {B, C, D, F} keep each other alive until the system takes a new snapshot that reveals that they can be removed from the remembered sets
[[or until all four objects happen to migrate to the same region, but we *do* *not* rely on such luck]]

Concurrent SATB Mark and Refinement

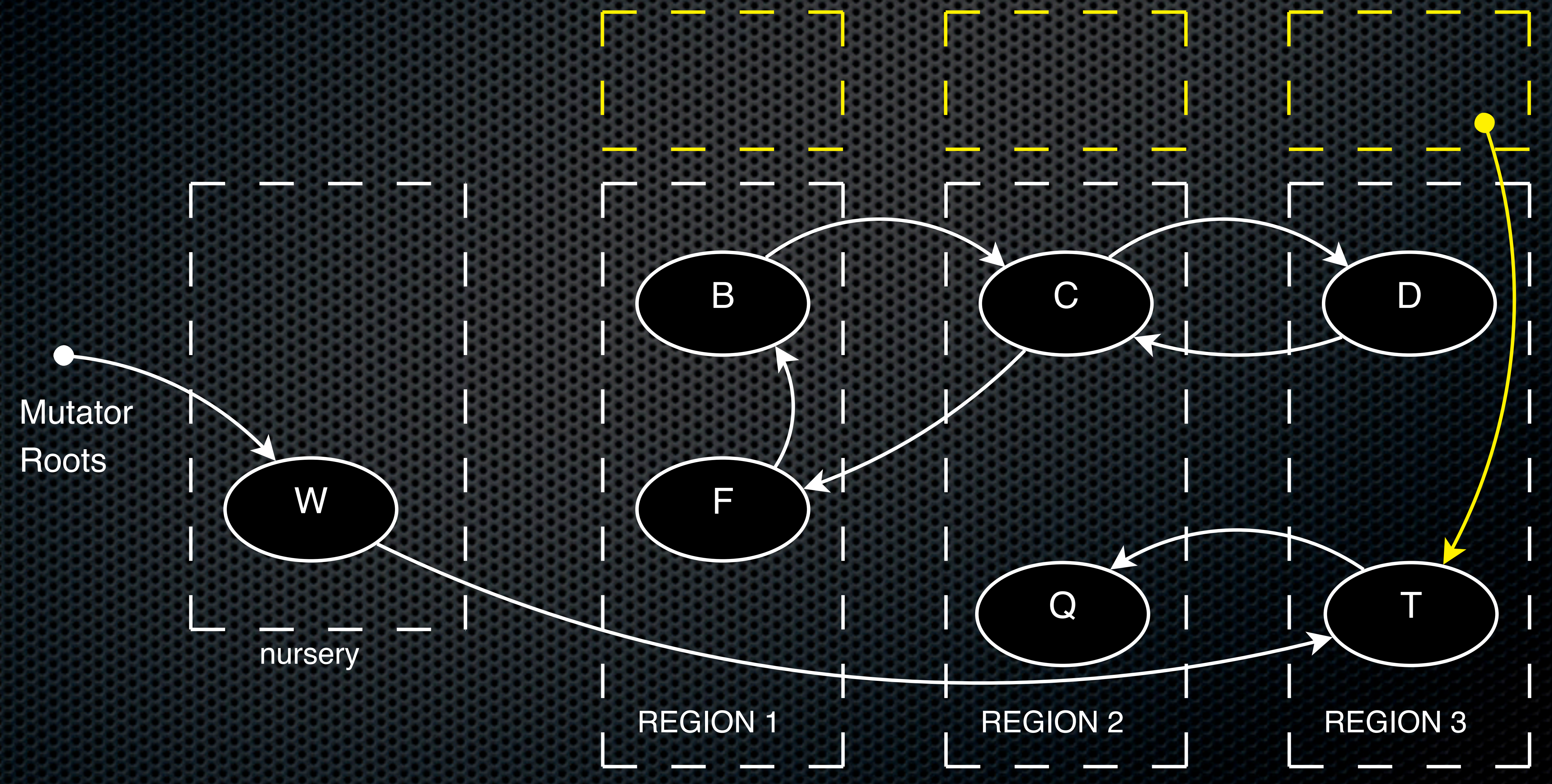


Thus corresponding entries can be removed from the remembered sets

Concurrent SATB Mark and Refinement



Concurrent SATB Mark and Refinement



Now the remsets do not see the cycle, and the collector will reclaim the storage for {B, F} when it collects region 1 (and likewise for objects C and D).

Worst Case Bounds

Worst Case Space Bounds

- Each of the N/R regions has O(R) associated state
  - Total heap size is O(N)
- Heap size kept proportional to reachable storage
  - Wave-off = 2R ensures 1/2 N/R regions collectable
  - Refinement keeps waved-off regions from preserving float elsewhere

I am assuming here that the marking process is run sufficiently often to keep region-crossing garbage from accumulating. Also, this is only a quick sketch; not unreasonable to calculate the constant factors involved in the worst-case asymptotic bounds.

Worst Case Time Bounds

- Pause Time Bounds
  - Traverse the O(R) summary, collect O(R) objects, and update O(R) structure of region's remembered set, mark stack, and points-into summaries
  - Therefore work at each pause bounded by O(R)
- Minimum Mutator Utilization
  - Pauses spaced apart by allocation + mutation limits

I am assuming here that the summarization task is given sufficient priority to keep collector on schedule. Some side issues: MMU, Max Mutations / Summarization Cycle.
[[That is work-in-progress]]

Common Case Behavior

Common case performance will outperform worst case. A typical region has a small mark-stack and small points-into summary, as well as a reasonably sized points-outof remembered set. To see worst-case performance, an adversarial mutator would need to create region-crossing garbage cycles as quickly as it can, and it would also need to cause maximal wave-off. Experiments so far with our prototype support (the size) claim. I need to gather more data to fully support both of these claims.

Empirical Results

To test the design, I have constructed a prototype.

Prototype

- Regional collector for Larceny
- Compared against Larceny’s other collectors

I am using the Larceny runtime for the Scheme language as the basis for the prototype, and comparing it against Larceny’s other collectors (a generational one and a non-generational stop-and-copy collector).

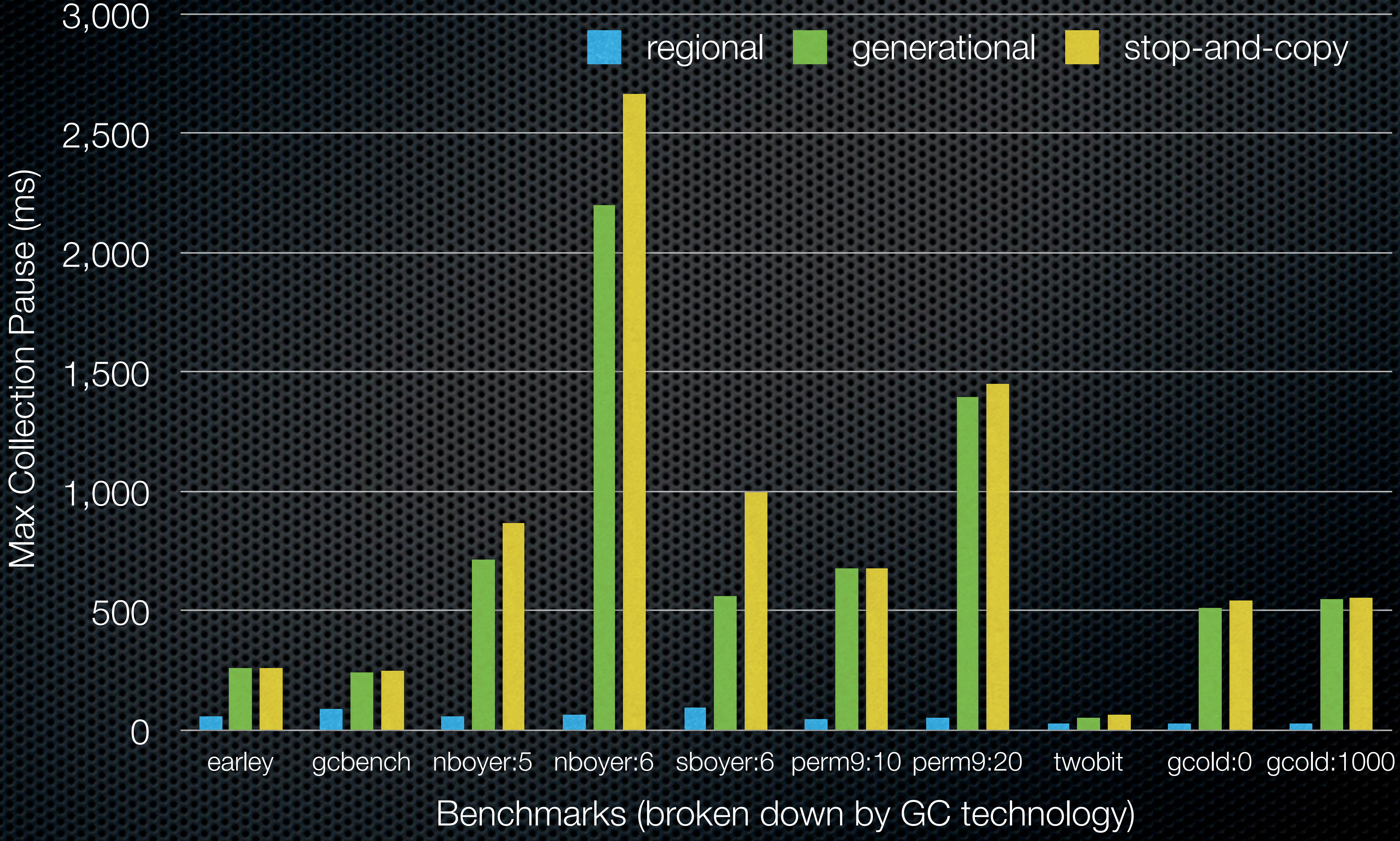
Problems with prototype

- Entirely sequential
- SATB marking emulated
  - SATB write barrier code is emitted in-line but ignored
- Points-into summarization emulated
  - Each traversal over all remembered sets computes only *one* points-into summary, instead of K = Θ(N/R)

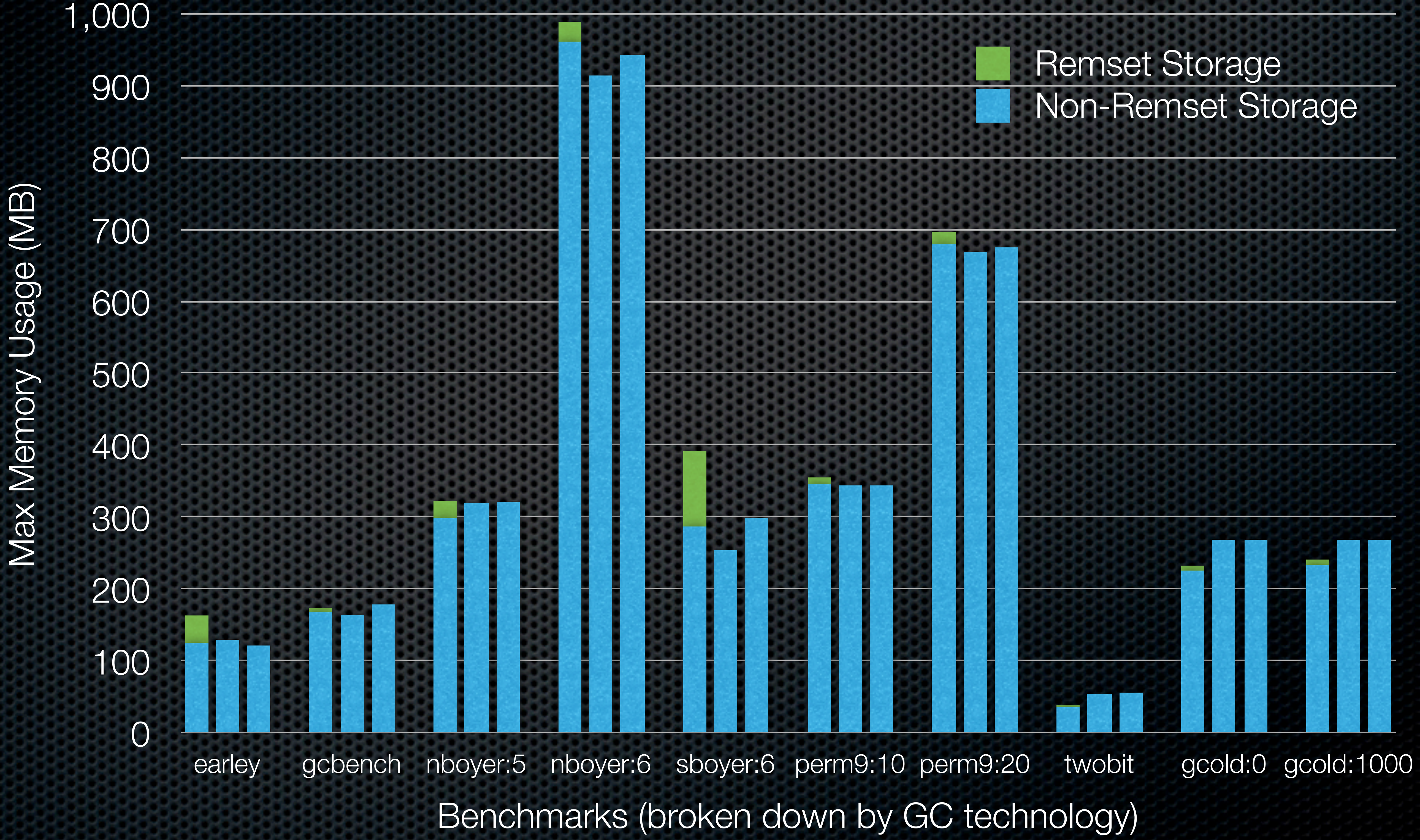
The stop-and-mark computation was scheduled so that the system did 1 word of marking for every 3 words of allocation. Experiments used a 1 MB nursery and 5 MB regions

[[If time and feeling good, patter about experiencing the dual of second system effect.]]

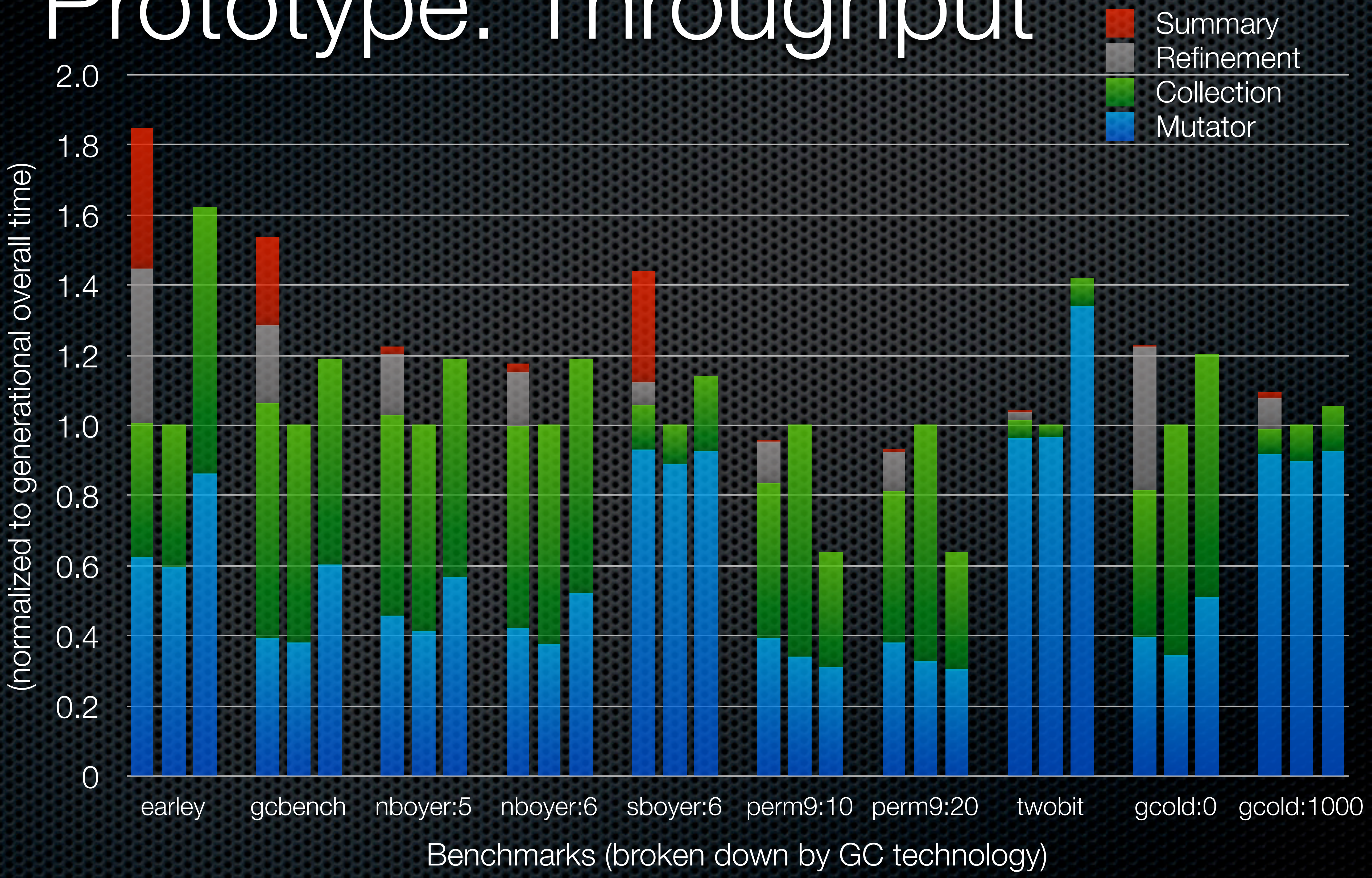
Prototype: Pause Time



Prototype: Memory Usage



Prototype: Throughput



Fixing the Prototype

- Summary representation
- Incremental tasks
- Concurrent tasks
- Heap contraction
- 64-bit Larceny

Related Work: Generational Collection

- Originated with [Lieberman and Hewitt: CACM ’83]
- Simplified in [Ungar: ’84]

Regional collector is more like LiebermanHewitt than Ungar
Already described GF.
MOS algorithm has uses a policy to migrate cycles into the same “train”; hard to reason about how long it takes to reclaim cyclic garbage
Beltway allows flexible policy selection, but one must choose between *either* incremental or complete collection; the Regional collector offers both at once.

Related Work: Heap Partitioning

- Mature Object Space (“Train”) [Hudson and Moss: IWMM ’92]
- Beltway [Blackburn et al: PLDI ’02]
- MarkCopy [Sachindran and Moss OOPSLA ’03]
- Garbage-First GC: [Detlefs et al: ISMM ’04]

Related Work

- Incremental, Concurrent and/or Real-Time GC
  - (Many!)
  - Multiprocessing, Compactifying [Steele: CACM '75]
  - On-the-Fly [Dijkstra et al: LNCS '76, CACM '78]
  - Parallel Real-Time [Blelloch and Cheng: PLDI '99, PLDI '01]
  - Metronome [Bacon et al: LCTES '03]

Collectors of Steele and of Dijkstra et al are notoriously complex and subtle to implement. Blelloch and Cheng work is notable for provided hard bounds on space and time usage (and also introduced notion of MMU).
Metronome collector is a real-time mark-sweep design where they got the measured read-barrier cost down to an impressive 4% overhead. Regional collector is a totally different point in the design space.

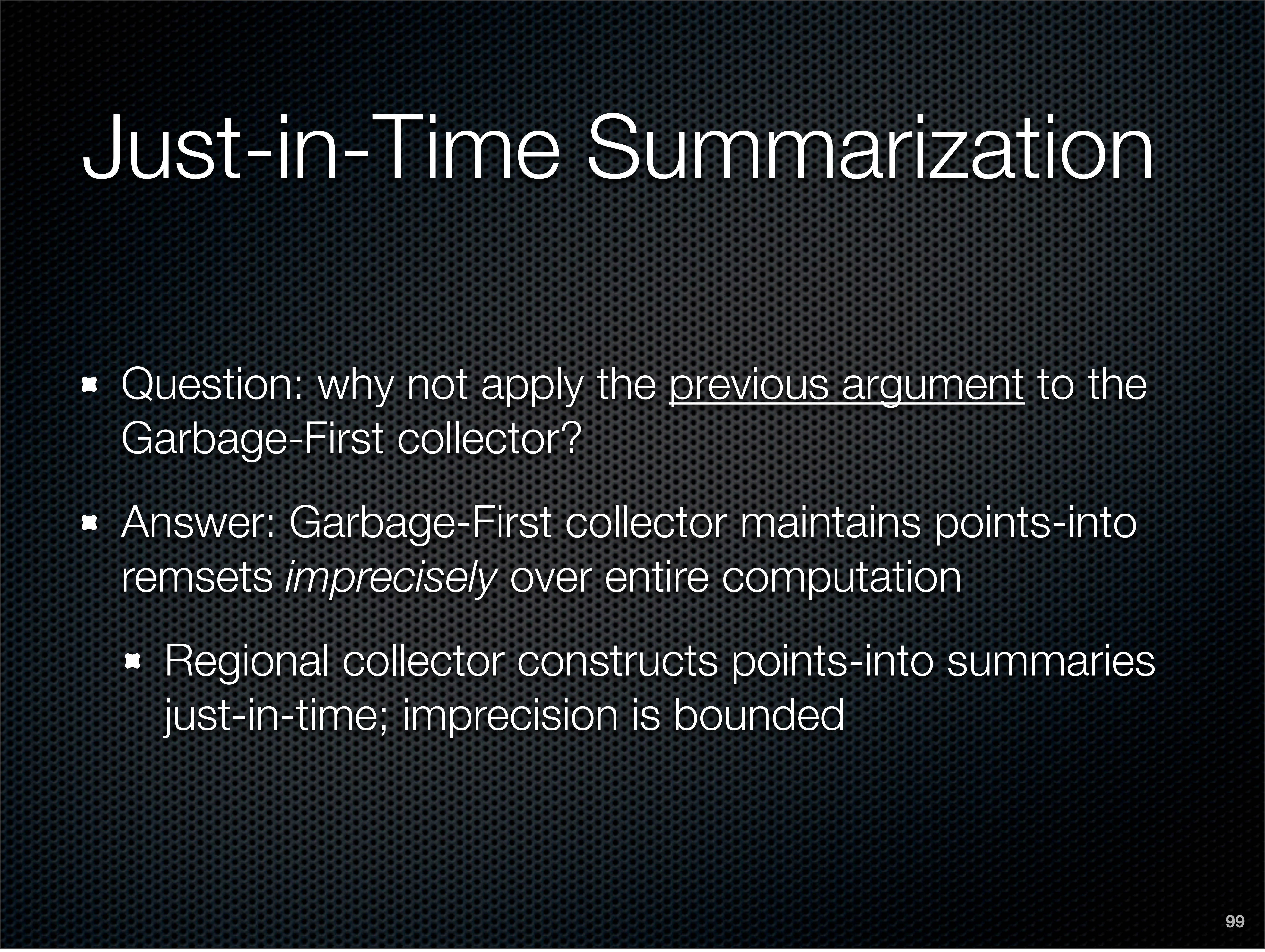
Proposed Schedule

May – September 2008	Implementation work: incremental summarization and marking; concurrent marking; concurrent summarization.
October – January 2009	Gather benchmark results on time and memory usage; write thesis
February – April 2009	Defend Thesis

Conclusion

- Regional Collection
  - Provable worst case bounds on space usage and pause times
  - Low implementation complexity (for mutator)
  - High throughput if concurrent task available

Thanks for listening!



Just-in-Time Summarization

- Question: why not apply the previous argument to the Garbage-First collector?
- Answer: Garbage-First collector maintains points-into remsets *imprecisely* over entire computation
  - Regional collector constructs points-into summaries just-in-time; imprecision is bounded

Note: The points-into summaries' imprecision can and *must* be bounded. If imprecision unbounded, then previous argument falls apart b/c cannot reason based on heap state at particular instant.
[[Note: a possible experiment would be to try variant of Garbage-First that maintains *precise* points-into remsets. We suspect the overhead of such a GC would be inacceptable.]]