Lecture 25

CIS 341: COMPILERS

Announcements

- Project 6
  - Due tonight!
- Project 7
  - Available soon
  - Due: May 5th
- Final Exam:
  - Tuesday, April 30th noon-2:00 pm
  - Moore 216

GARBAGE COLLECTION

Why Garbage Collection?

- Manual memory management is cumbersome & error prone:
  - Freeing the same pointer twice is ill defined (seg fault or other bugs)
  - Calling free on some pointer not created by `malloc` (e.g. to an element of an array) is also ill defined
  - `malloc` and `free` aren't modular: To properly free all allocated memory, the programmer has to know what code "owns" each object. Owner code must ensure free is called just once.
  - Not calling free leads to *space leaks*: memory never reclaimed
    - Many examples of space leaks in long-running programs

- Garbage collection:
  - Have the language runtime system determine when an allocated chunk of memory will no longer be used and free it automatically.
  - But… garbage collector is usually the most complex part of a language's runtime system.
  - Garbage collection does impose costs (performance, predictability)

Memory Use & Reachability

- When is a chunk of memory no longer needed?
  - In general, this problem is undecidable.

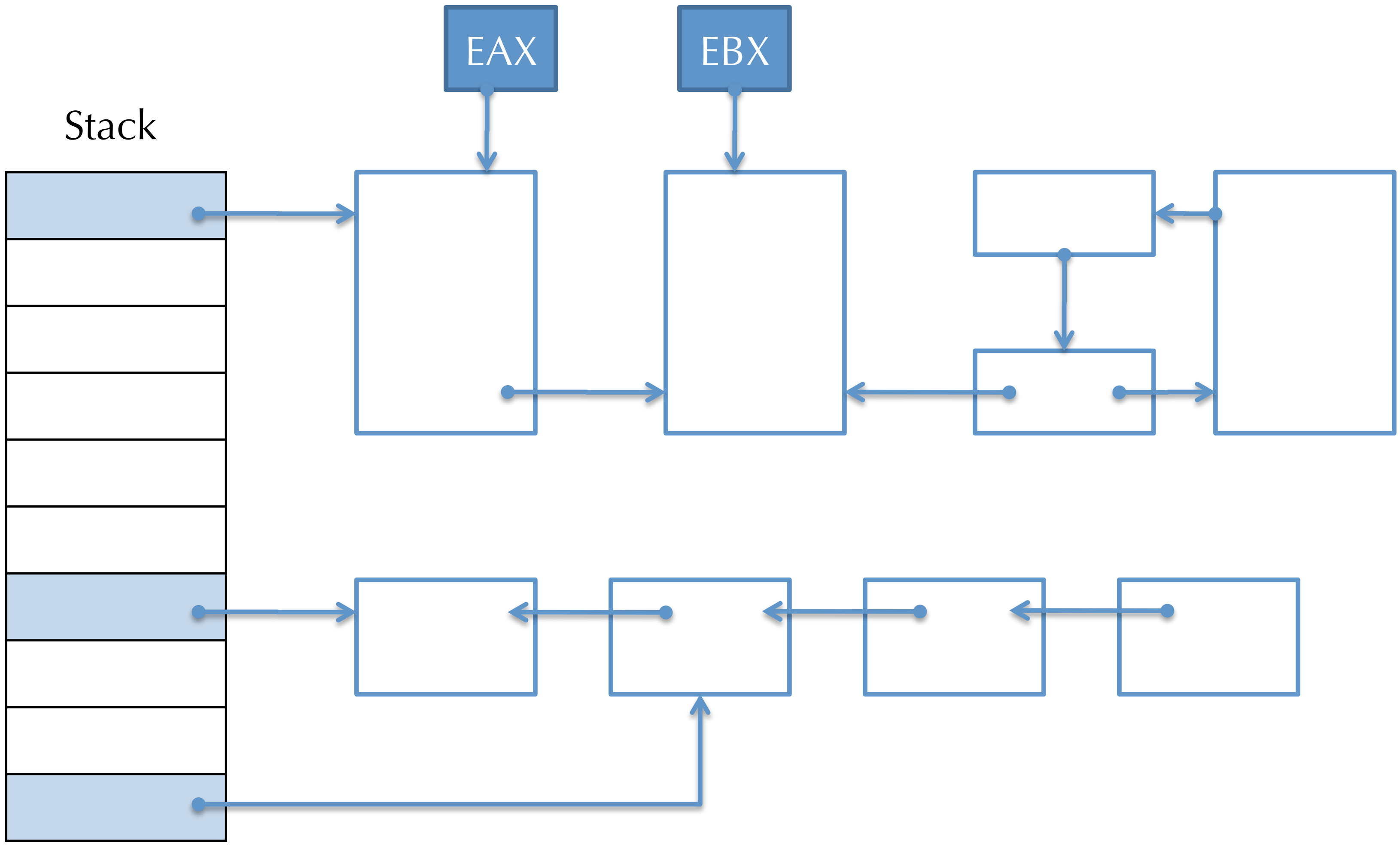
- We can approximate this information by freeing memory that can't be reached from any *root* references.
  - A *root pointer* is one that might be accessible directly from the program (i.e. they're not in the heap).
  - Root pointers include pointer values stored in registers, in global variables, or on the stack.
- If a memory cell is part of a record (or other data structure) that can be reached by traversing pointers from the root, it is *live*.
- It is safe to reclaim all memory cells not reachable from a root (such cells are *garbage*).

Reachability & Pointers

- Starting from stack, registers, & globals (*roots*), determine which objects in the heap are reachable following pointers.
- Reclaim any object that isn't reachable.
- Requires being able to distinguish pointer values from other values (e.g., ints).
- Type safe languages:
  - OCaml, SML/NJ use the low bit:
    1 it's a scalar, 0 it's a pointer. (Hence 31-bit ints in OCaml)
  - Java puts the tag bits in the object meta-data (uses more space).
  - Type safety implies that casts can't introduce new pointers
  - Also, pointers are abstract (references), so objects can be moved without changing the meaning of the program
- Unsafe languages:
  - Pointers aren't abstract, they can't be moved.
  - Boehm-Demers-Weiser *conservative* collector for C use heuristics: (e.g., the value doesn't point into an allocated object, pointers are multiples of 4, etc.)
  - May not find as much garbage due to conservativity.

Example Object Graph

- Pointers in the stack, registers, and globals are *roots*

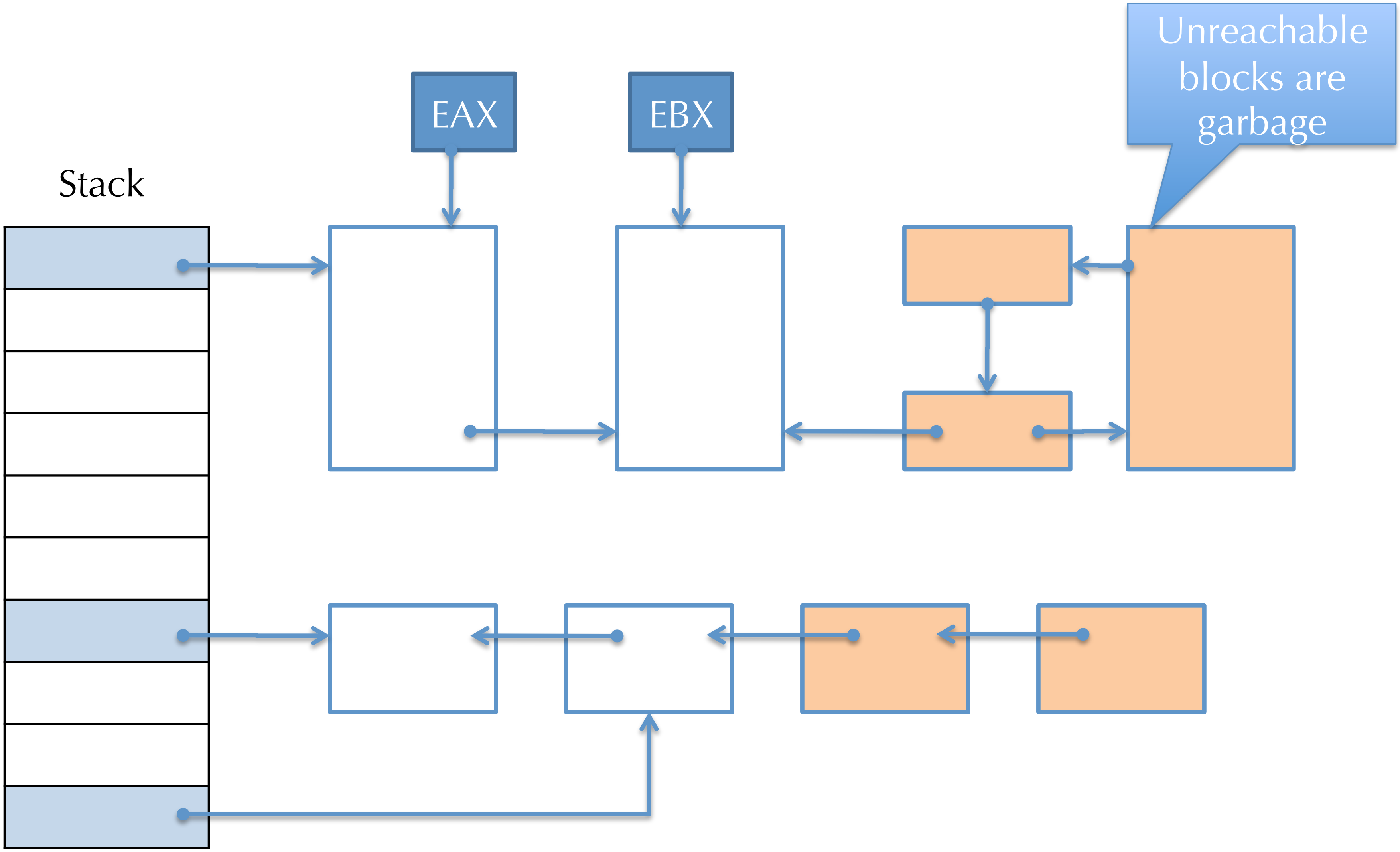


MARK & SWEEP GC

Mark and Sweep Garbage Collection

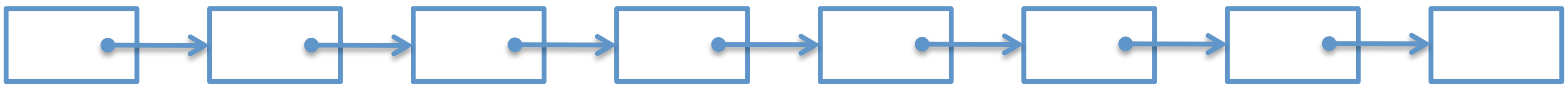
- Classic algorithm with two phases:
- Phase 1: Mark
  - Start from the roots
  - Do depth-first traversal, marking every object reached.
- Phase 2: Sweep
  - Walk over *all* allocated objects and check for marks.
  - Unmarked objects are reclaimed.
  - Marked objects have their marks cleared.
  - Optional: compact all live objects in heap by moving them adjacent to one another. (needs extra work & indirection to "patch up" pointers)

Results of Marking Graph



Implementing the Mark Phase

- Depth-first search has a natural recursive algorithm.
- Question: what happens when traversing a long linked list?



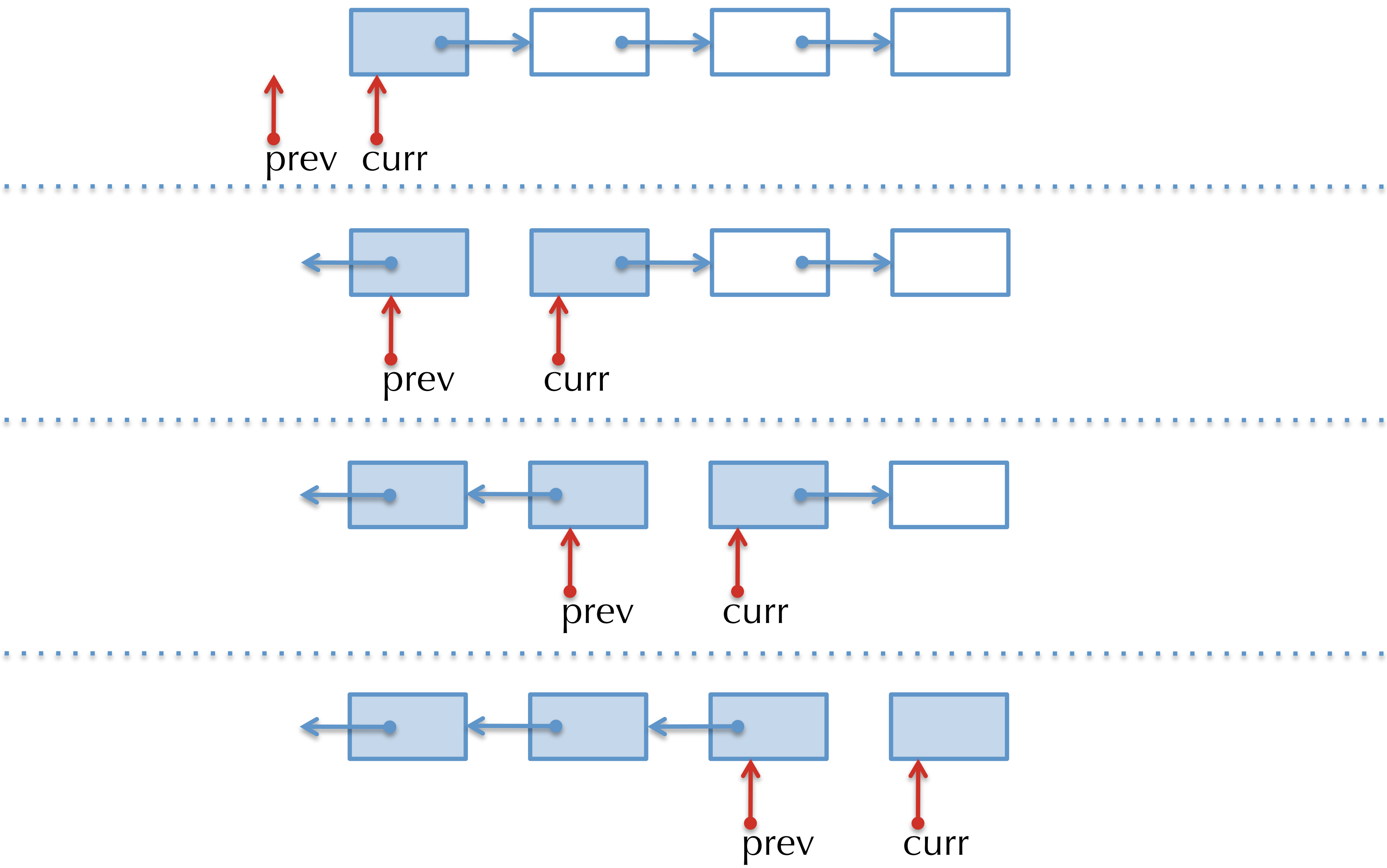
- Where do we store the information needed to perform the traversal?
  - (In general, garbage collectors are tricky to implement because if they allocate memory who manages that?!)

Deutsch-Schorr-Waite (DSW) Algorithm

- No need for a stack, it is possible to use the graph being traversed itself to store the data necessary…
- Idea: during depth-first-search, each pointer is followed only once. The algorithm can reverse the pointers on the way down and restore them on the way back up.
  - Mark a bit on each object traversed on the way down.

- Two pointers:
  - curr: points to the current node
  - prev points to the previous node
- On the way down, flip pointers as you traverse them:
  - tmp := curr
curr := curr.next
tmp.next := prev
prev := curr

Example of DSW (traversing down)



Costs & Implications

- Need to generalize to account for objects that have multiple outgoing pointers.
- Depth-first traversal terminates when there are no children pointers or all children are already marked.
  - Accounts for cycles in the object graph.
- The Deutsch-Schorr-Waite algorithm breaks objects during the traversal.
  - All computation must be halted during the mark phase. (Bad for concurrent programs!)
- Mark & Sweep algorithm reads all memory in use by the program (even if it's garbage!)
  - Running time is proportional to the total amount of allocated memory (both live and garbage).
  - Can pause the programs for long times during garbage collection.

COPYING COLLECTION

Copying Garbage Collection

- Like mark & sweep: collects all garbage.
- Basic idea: use *two* regions of memory
  - One region is the memory in use by the program. New allocation happens in this region.
  - Other region is idle until the GC requires it.

- Garbage collection algorithm:
  - Traverse over live objects in the active region (called the "*from- space*"), copying them to the idle region (called the "*to-space*").
  - After copying all reachable data, switch the roles of the from-space and to-space.
  - All dead objects in the (old) from-space are discarded en masse.
  - A side effect of copying is that all live objects are compacted together.

Cheney's Algorithm (1)

- Idea: maintain two pointers into the to-space
  - *Scan* – points to the next piece of data to be examined
  - *Free* – points to the next available word of memory
  - Invariant: data pointed to by values between the scan and free pointers might need to be copied to the to-space
  - Leave behind "*forwarding pointers*" to the new copies.

- Crucial subroutine: (note implicit use of type information)

  `pointer copy-forward(pointer p)`
  - If structure pointed to by `p` has already been copied, return the corresponding forwarding pointer.
  - Otherwise:
    - Copy the structure pointed to by `p` into the to-space. (Incrementing the free pointer)
    - Mark the structure in from-space as copied and put a forwarding pointer in from-space to the copy in to-space
    - Return the pointer to the new copy in to-space

Cheney's Algorithm (2)

- When garbage collection is triggered:
  - Initialize the free pointer to be beginning of to-space
- For each root R containing a pointer ptr:

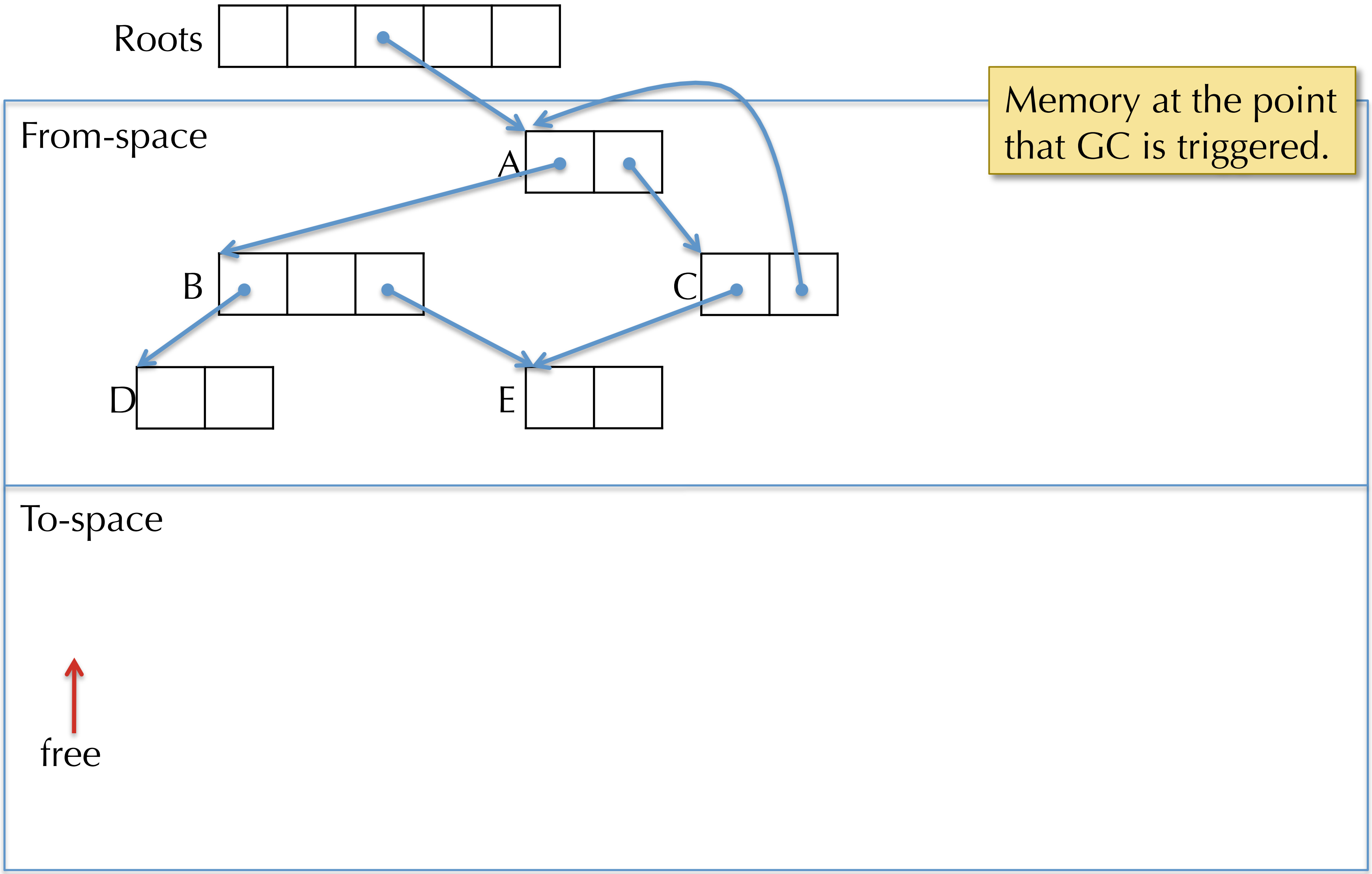
  Set ptr′ = copy-forward(ptr)

  Set R := ptr′

  Set the scan pointer to ptr′.

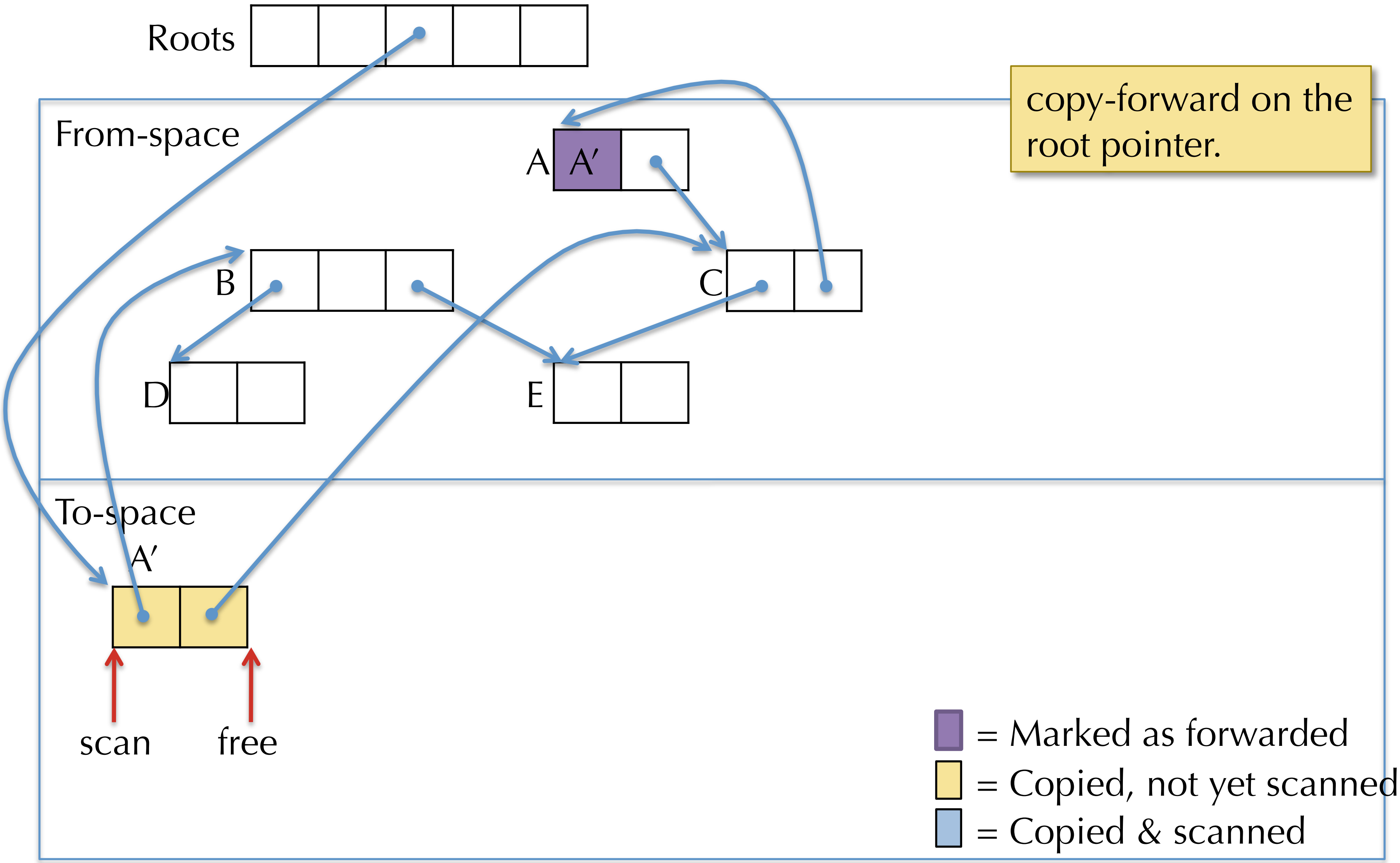
  While (scan != free)
  - Increment the scan pointer (element-wise according to types of the fields in the underlying structure)
  - If the scan pointer points to a pointer ptr
    - Set `*scan := copy-forward(ptr)`

Run of Cheney's Algorithm

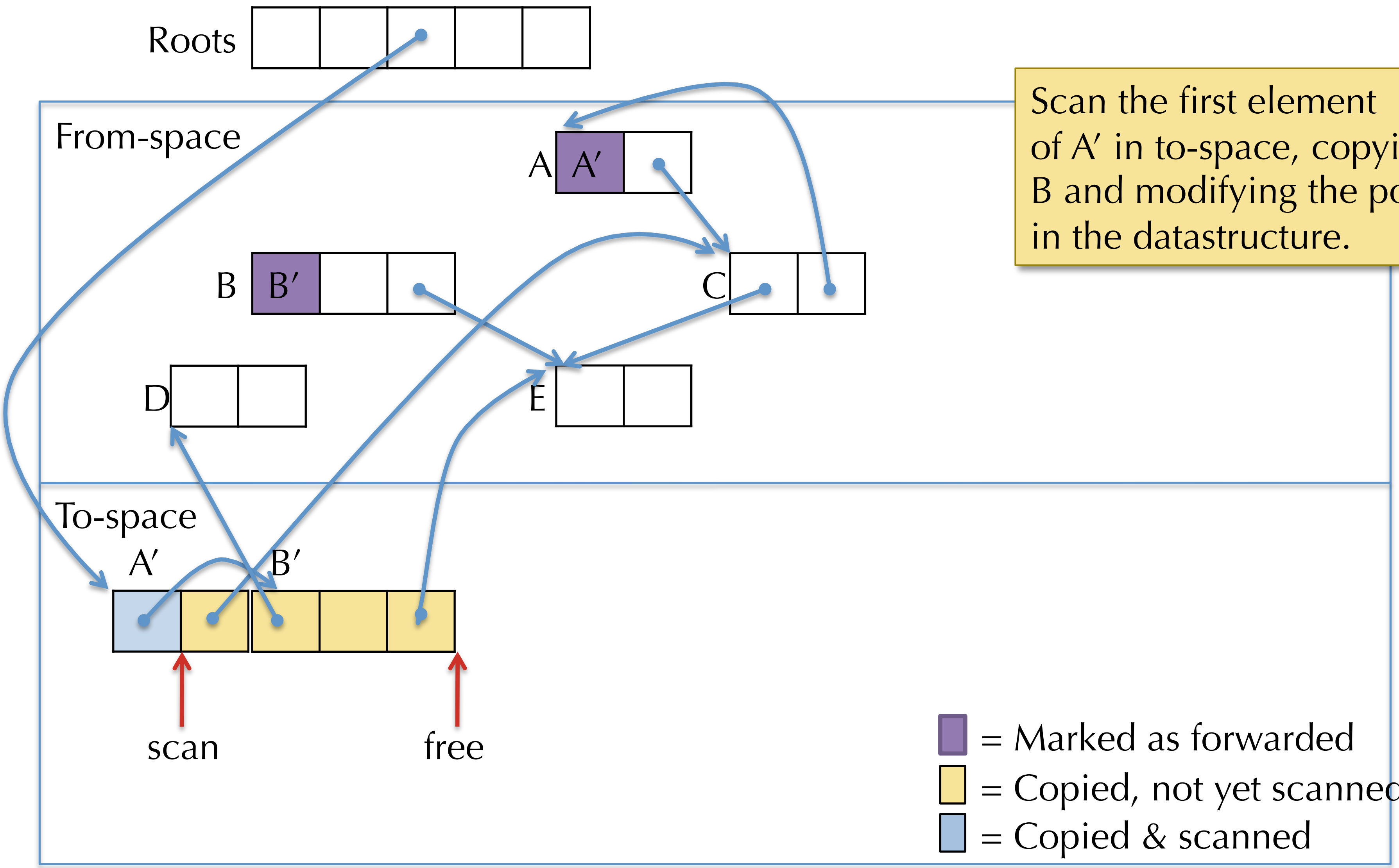


Memory at the point that GC is triggered.

Run of Cheney's Algorithm

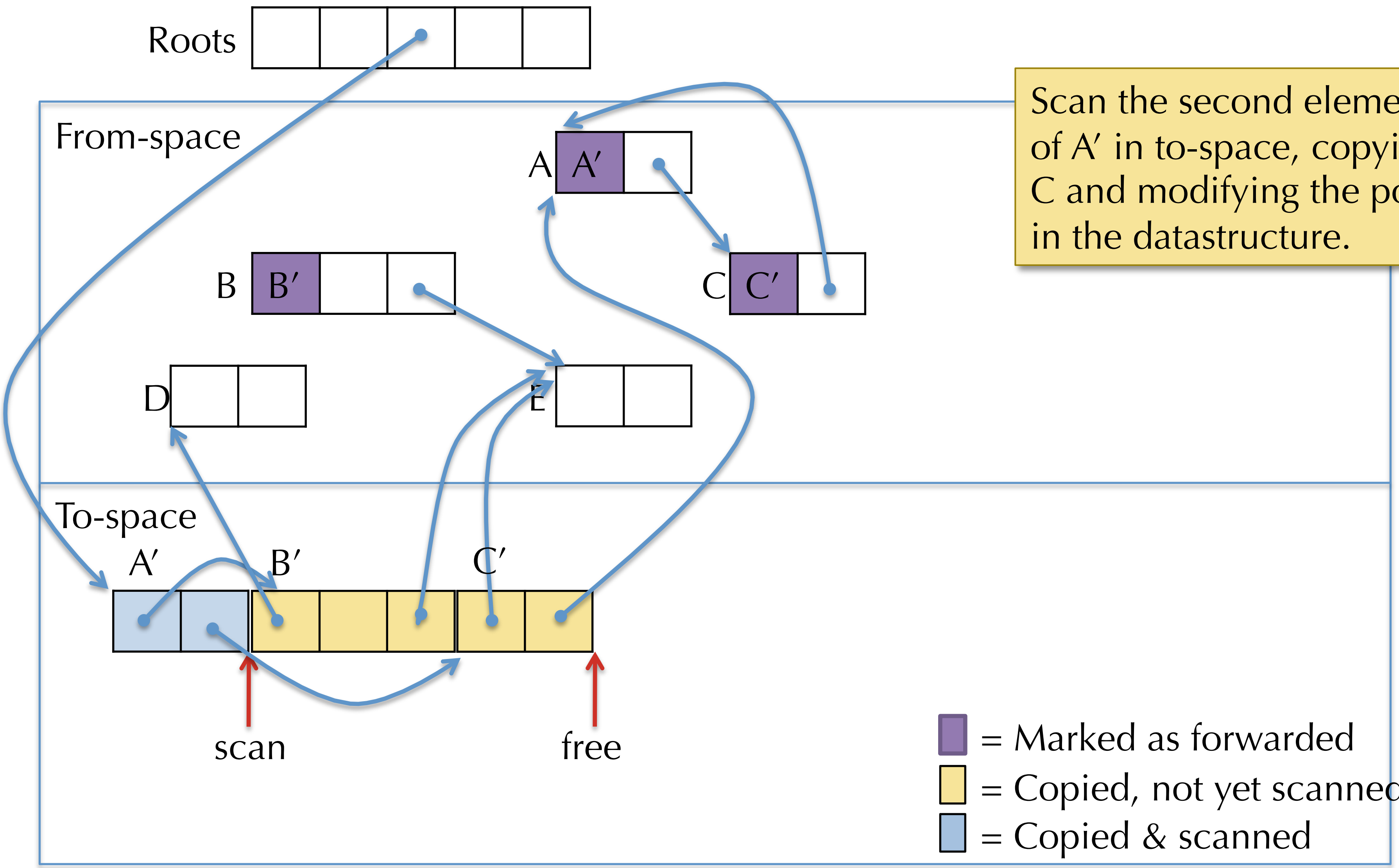


Run of Cheney's Algorithm



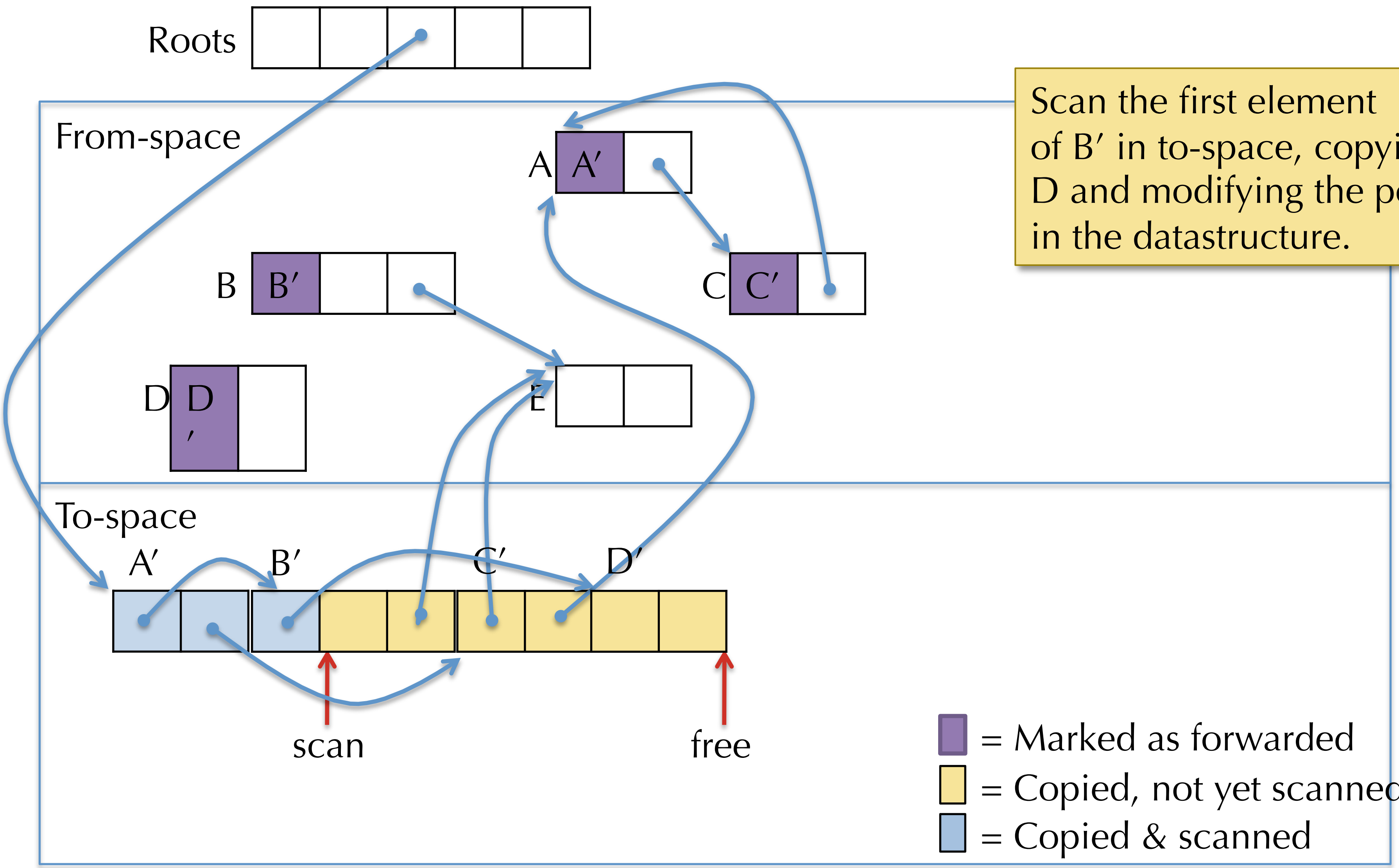
Scan the first element of A′ in to-space, copying B and modifying the pointer in the datastructure.

Run of Cheney's Algorithm



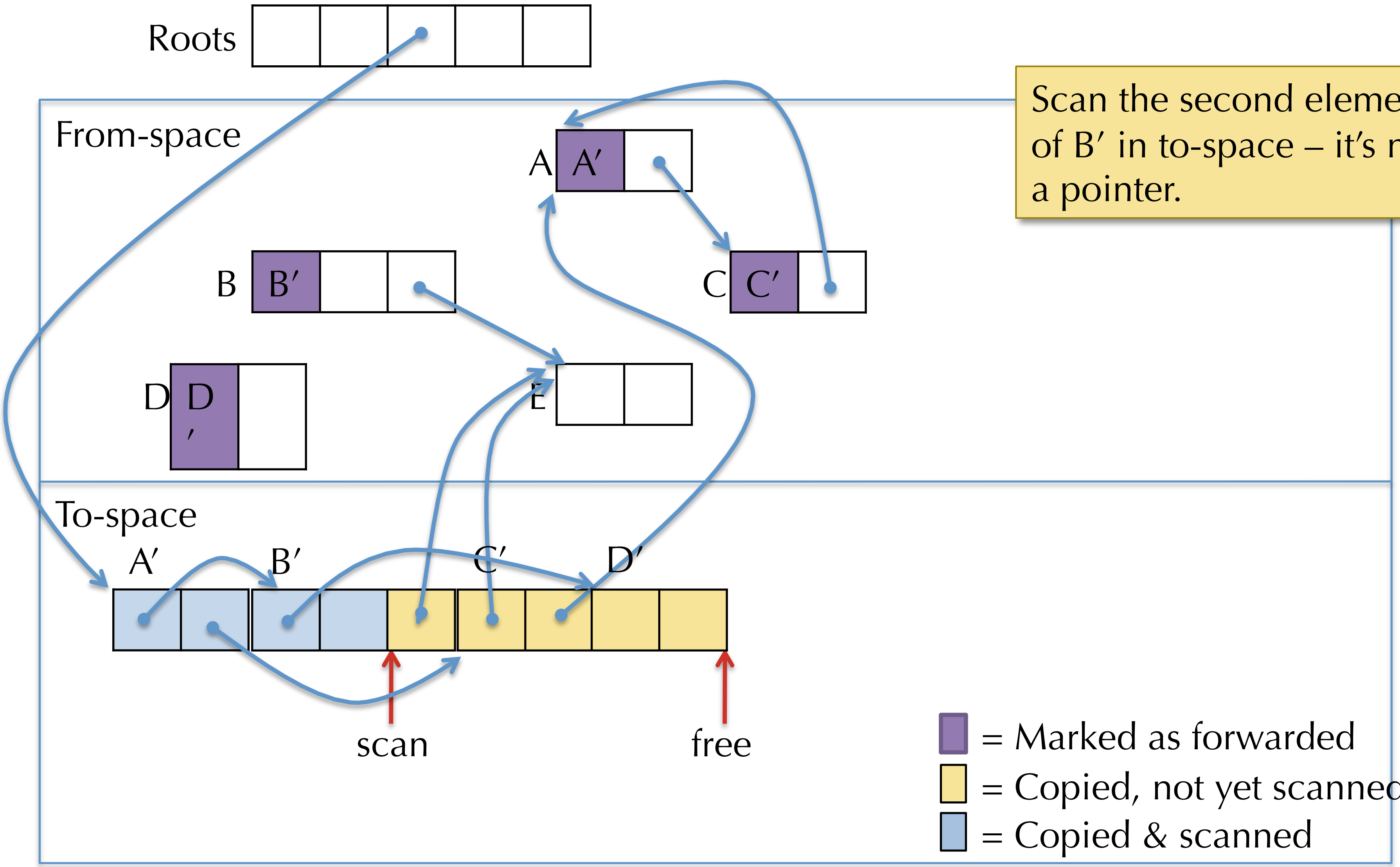
Scan the second element of A′ in to-space, copying C and modifying the pointer in the datastructure.

Run of Cheney's Algorithm



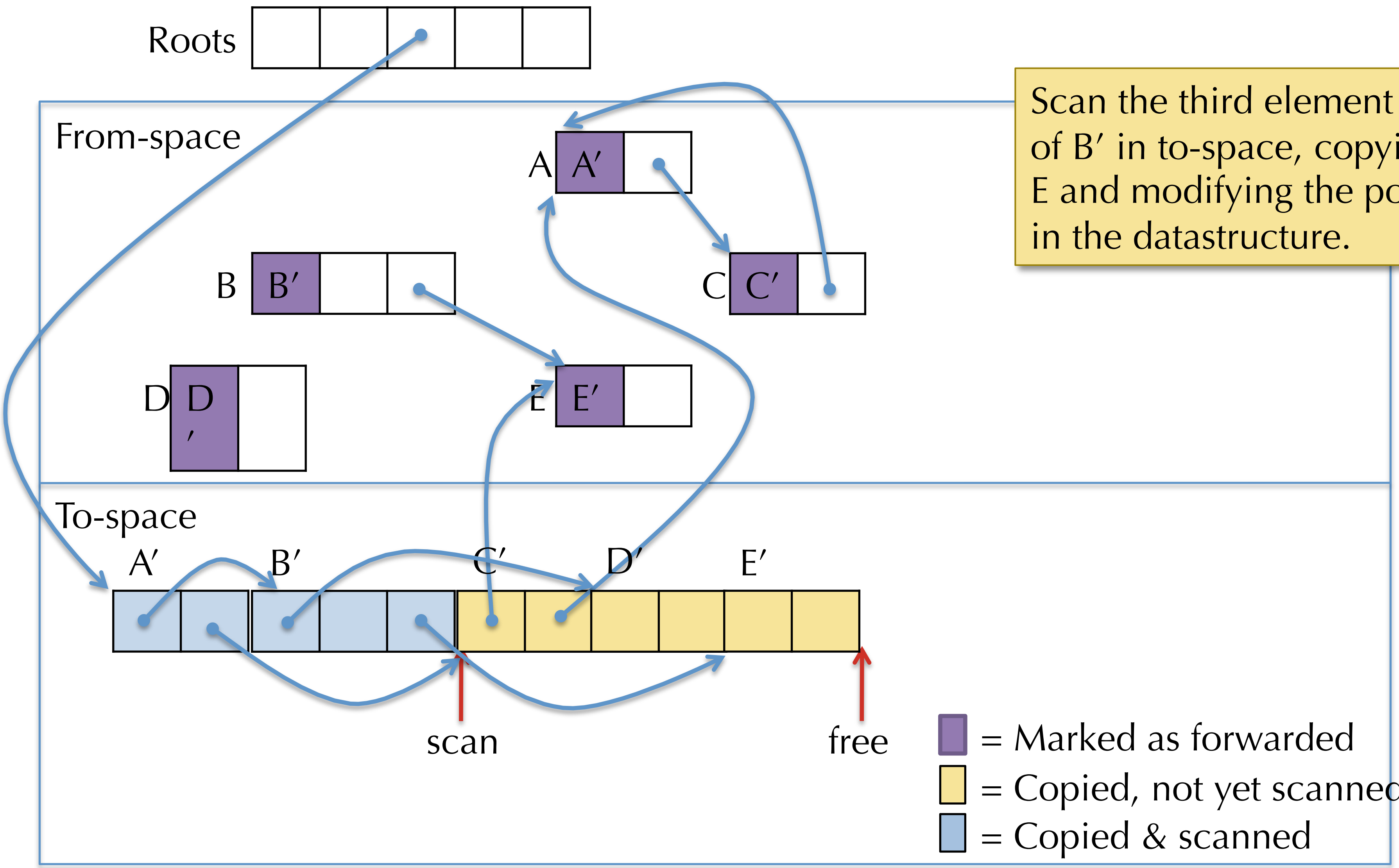
Scan the first element of B′ in to-space, copying D and modifying the pointer in the datastructure.

Run of Cheney's Algorithm



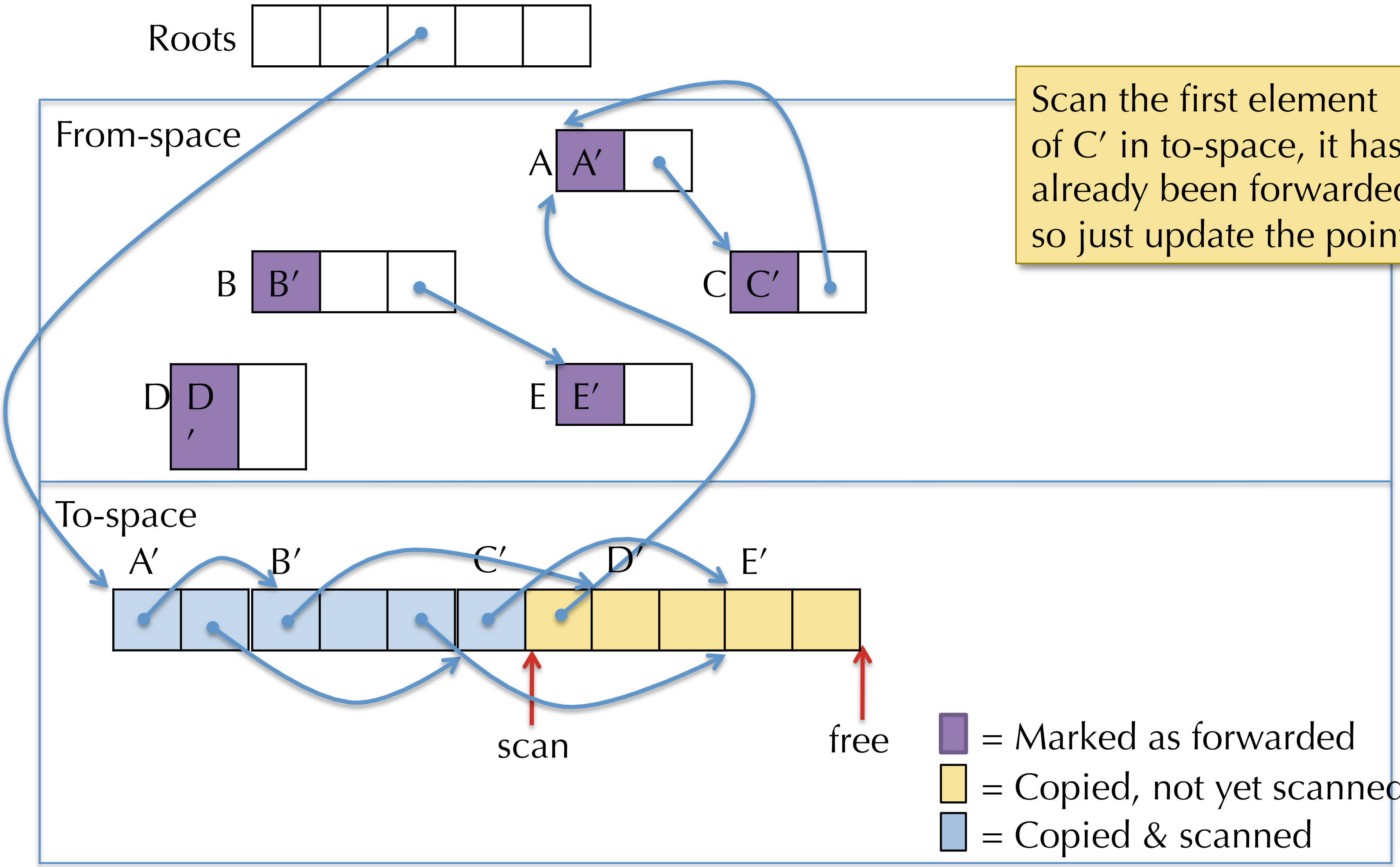
Scan the second element of B′ in to-space – it's not a pointer.

Run of Cheney's Algorithm



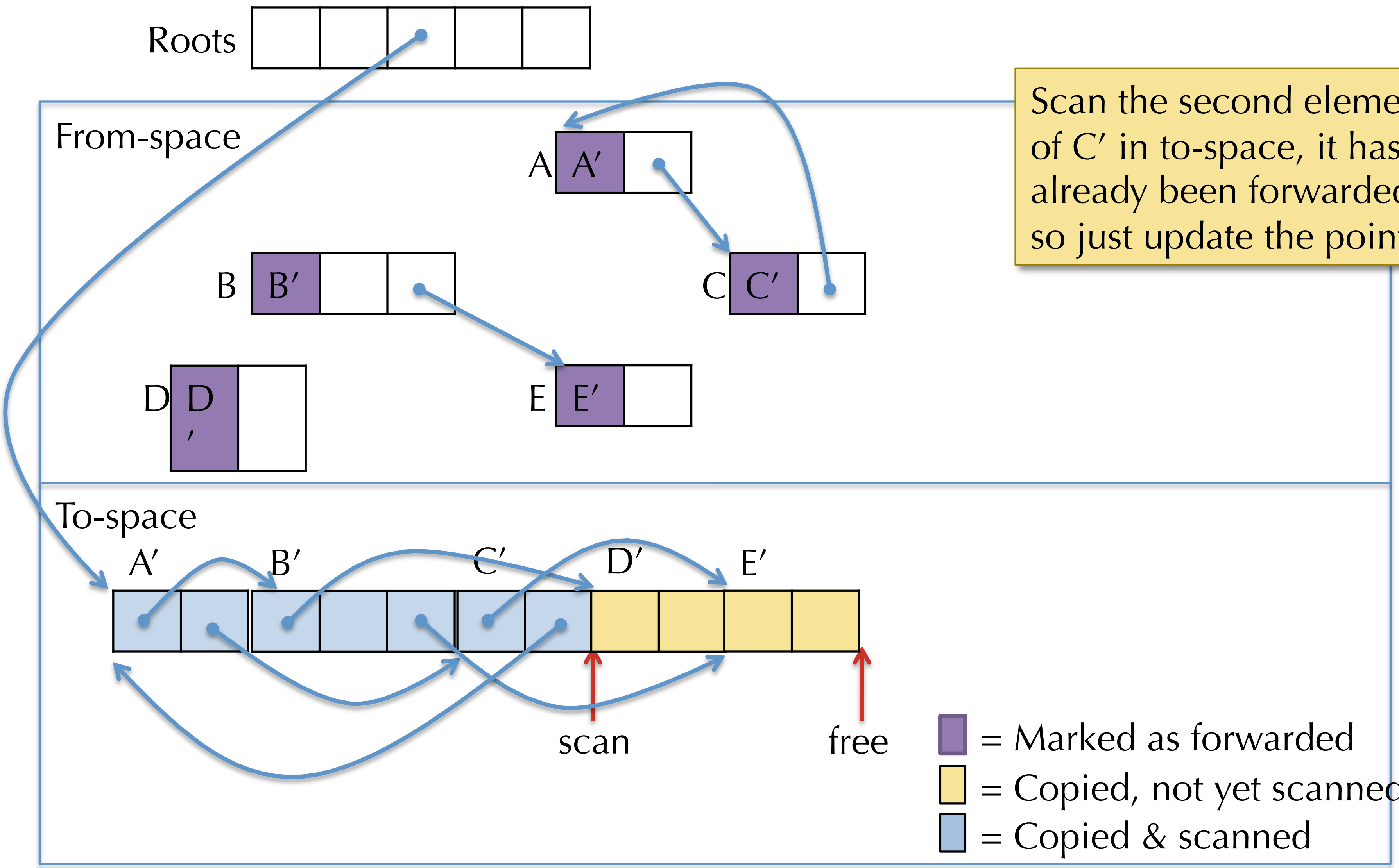
Scan the third element of B′ in to-space, copying E and modifying the pointer in the datastructure.

Run of Cheney's Algorithm



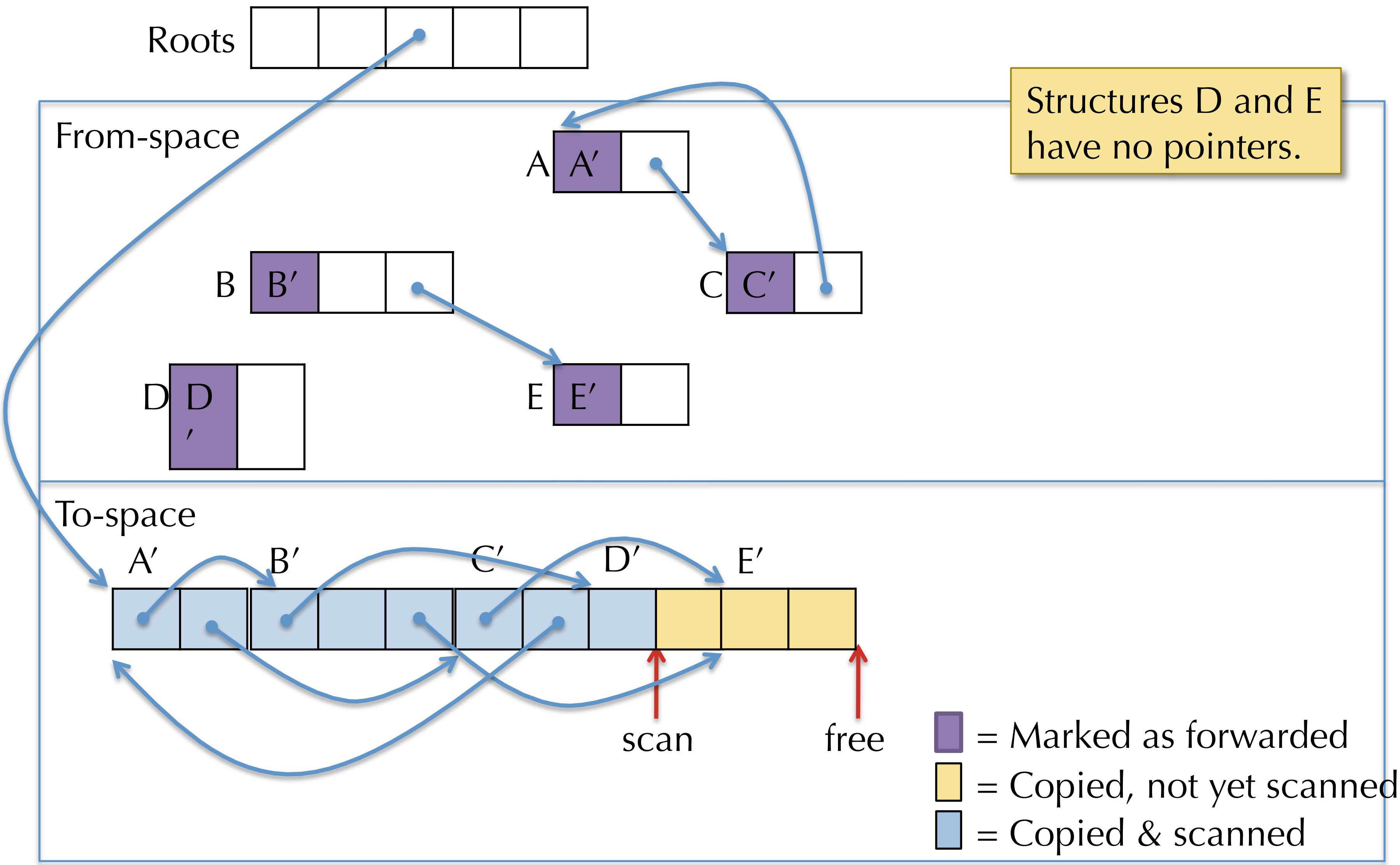
Scan the first element of C′ in to-space, it has already been forwarded so just update the pointer.

Run of Cheney's Algorithm

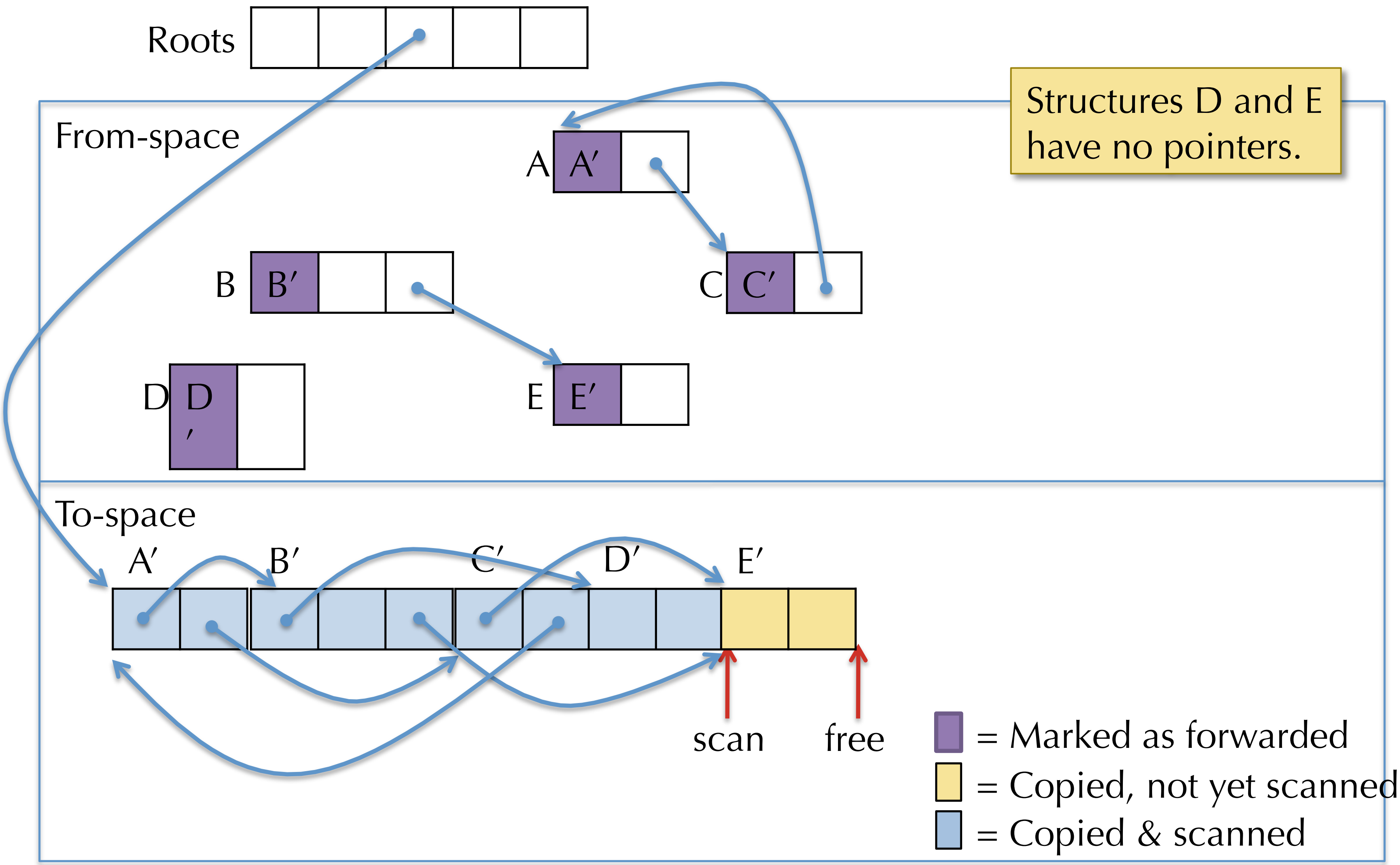


Scan the second element of C′ in to-space, it has already been forwarded so just update the pointer.

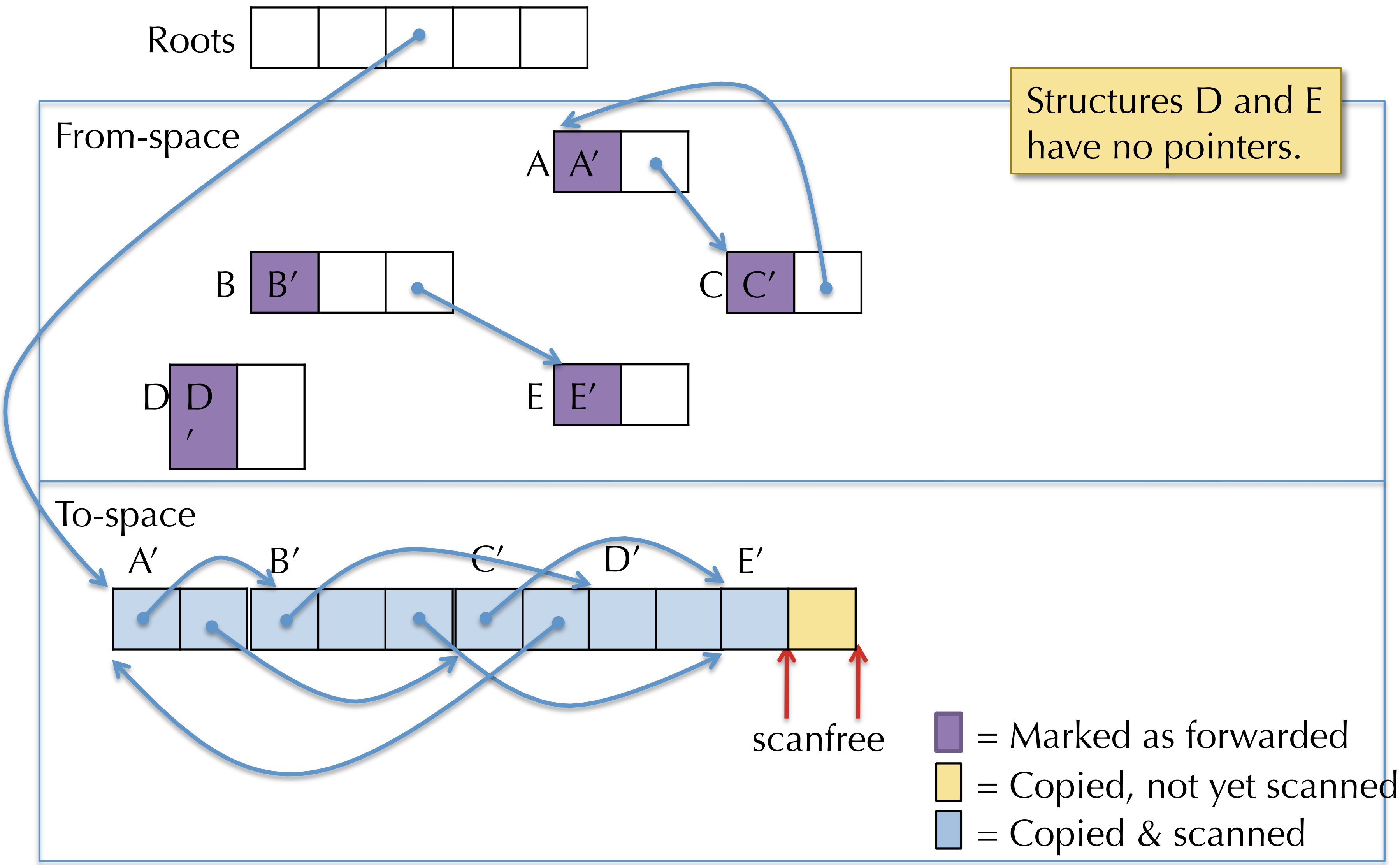
Run of Cheney's Algorithm



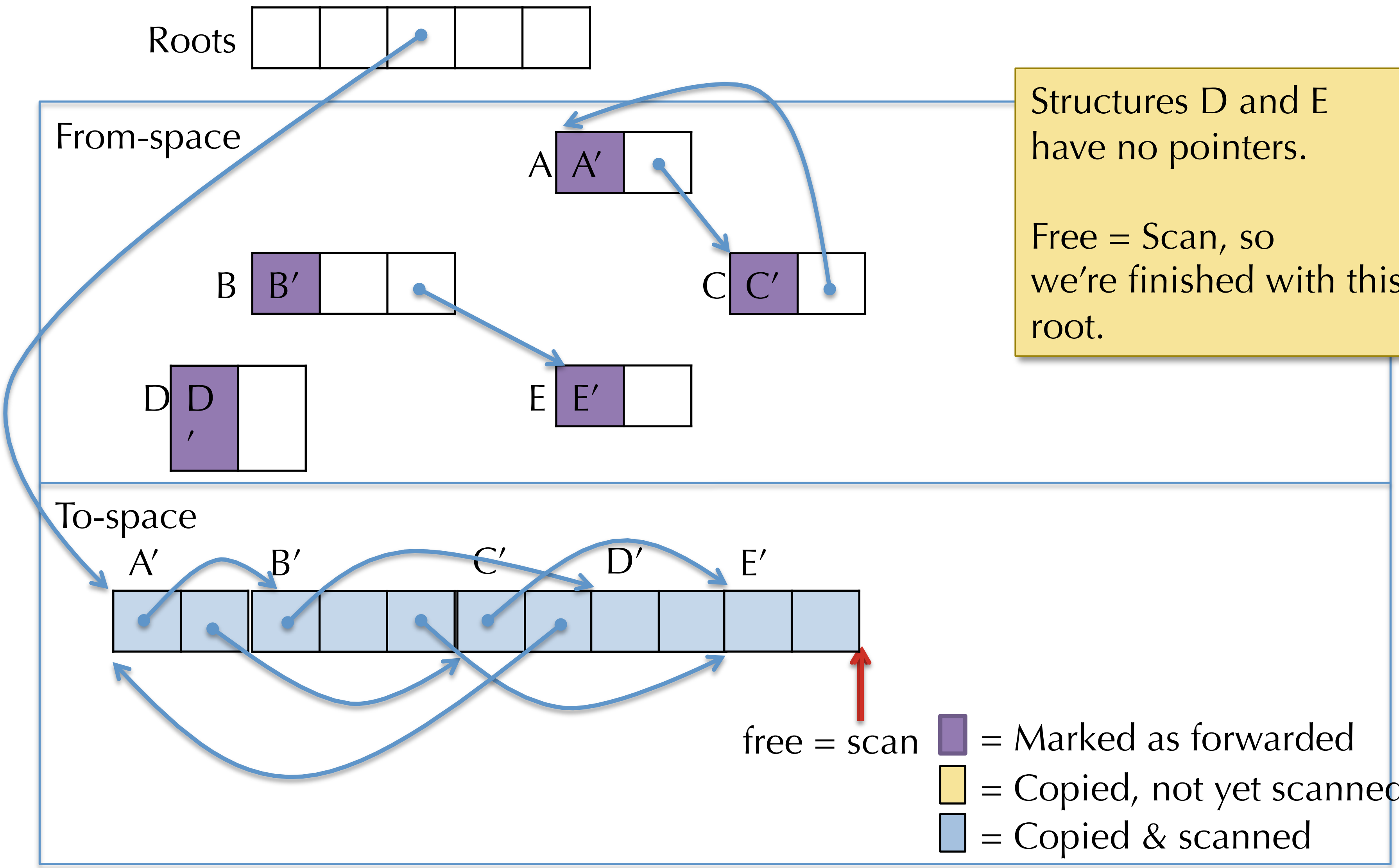
Run of Cheney's Algorithm



Run of Cheney's Algorithm



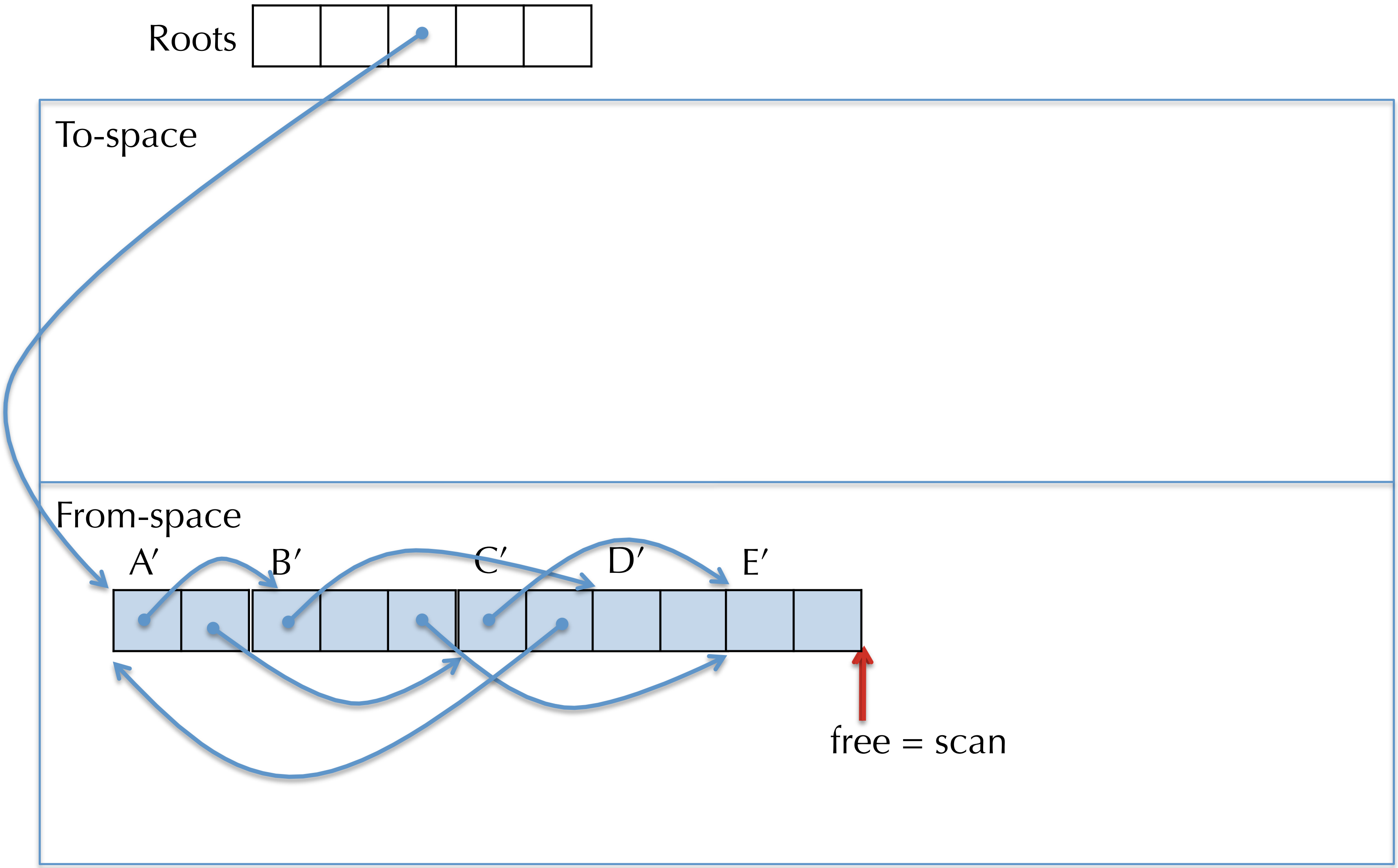
Run of Cheney's Algorithm



Structures D and E have no pointers.

Free = Scan, so we're finished with this root.

Run of Cheney's Algorithm



Tradeoffs of Copying Collection

- Benefits:
  - Simple, no stack space needed to implement the algorithm.
  - Running time is proportional to the number of reachable objects (not all allocated objects)
  - Automatically eliminates fragmentation by compacting memory during copy phase.
  - `malloc(n)` is implemented by `free := free + n`

- Drawbacks:
  - Twice as much memory is needed
  - Lots of memory traffic
  - Precise pointer/type information is required for traversal
  - Still can have long pauses

Baker's Concurrent GC

- Variant of copying collection in which the program and the garbage collector run concurrently.
- Program holds only pointers to to-space
- On field-fetch operation, if the pointer is in from-space, run copy-forward instead of directly fetching.
  - Moves the structure to to-space to maintain the invariant
  - Incrementally garbage collects as the program touches data.
- When the to-space fills up, swap to/from by copying the roots and fixing up the stack and registers.

- Avoids long pauses due to copying

Generational Garbage Collection

- Observation: If an object has been reachable for a long time, it is likely to remain so.
- In long-running programs, mark & sweep and copying collection waste time and cache by scanning/copying old objects.
- Idea: Assign objects to different *generations* G_0, G_1, G_2, …
  - Generation G_0 contains newest objects, most likely to become garbage (< 10% live)
  - Younger generations scanned for garbage much more frequently than older generations.
  - New object eventually given tenure (promoted to the next generation) if they last long enough.
  - Roots of garbage collection for G_0 include objects in G_1
- *Remembered sets*:
  - Avoid scanning all tenured objects by keeping track of pointers from old objects to new objects. Compiler emits extra code to keep track of such pointer updates.
  - Pointers from old generations to new generations are uncommon

GC in Practice

- Combination of generational and incremental GC techniques reduce delay
  - Millisecond pause times
- Very large objects (e.g. big arrays) can be copied in a "virtual" fashion without doing a physical copy
  - Complicates the book keeping
- Some systems combine copying collection (for young data) with mark & sweep (for old data)
- Challenging to scale to server-scale systems with terabytes of memory
- Interactions with OS matter a lot
  - It can be cheaper to do GC than it is to start paging
- GC is here to stay (thanks to Java, C#, etc.)

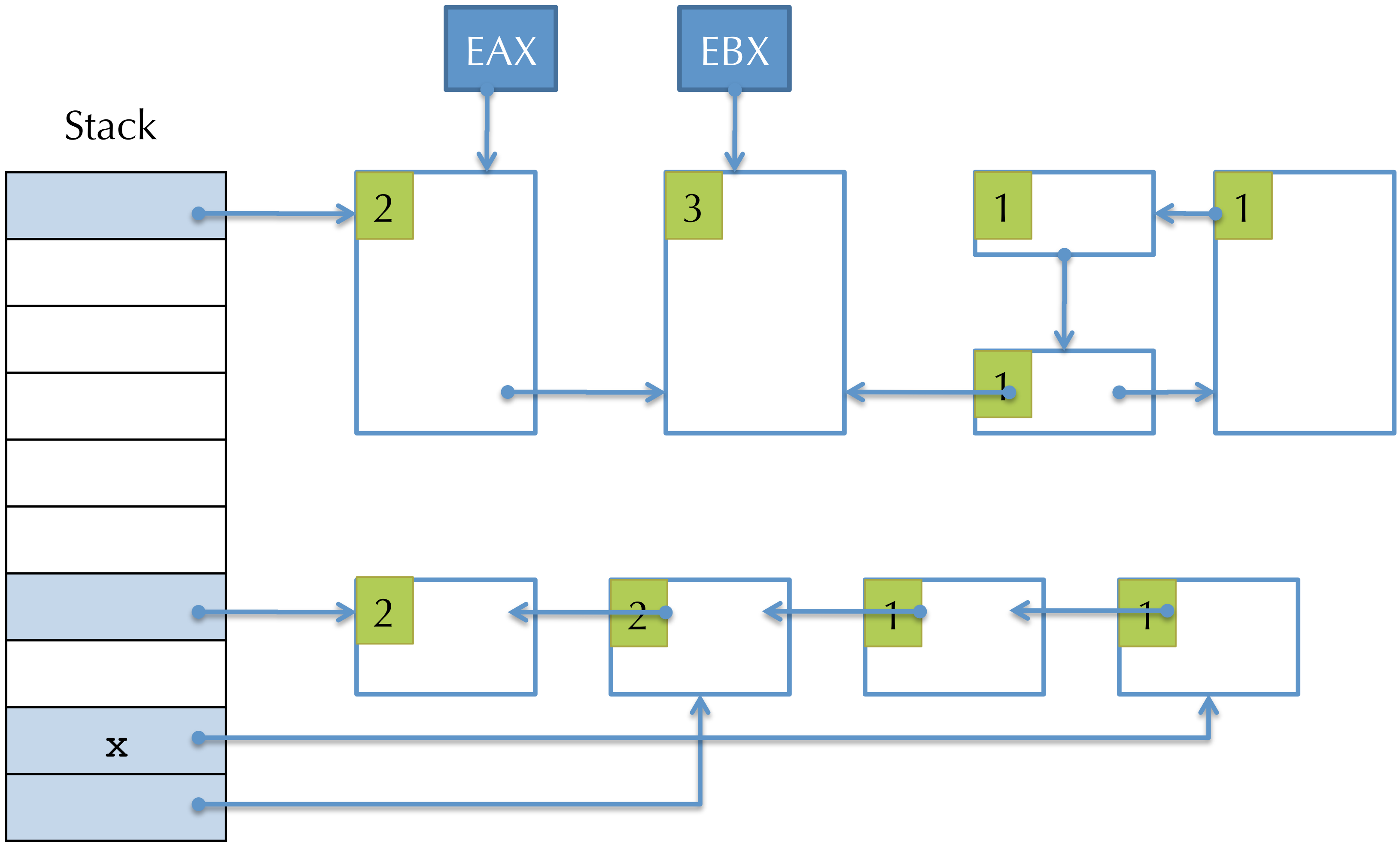
REFERENCE COUNTING

Reference Counting

- Idea: Keep track of the number of references to a given object.
  - When creating a new reference to the object, increase the reference count
  - On a call to `free`, decrement the reference count
  - If the reference count is 0, the object can be deallocated immediately
- Deallocating an object will decrement reference counts of objects it points to
  - Deallocations can "cascade," causing lots of objects to be deallocated
- Benefit: immediate reclamation of the space (no need to wait for garbage collector)
- Challenges:
  - Tracking reference counts efficiently
  - Cyclic data structures

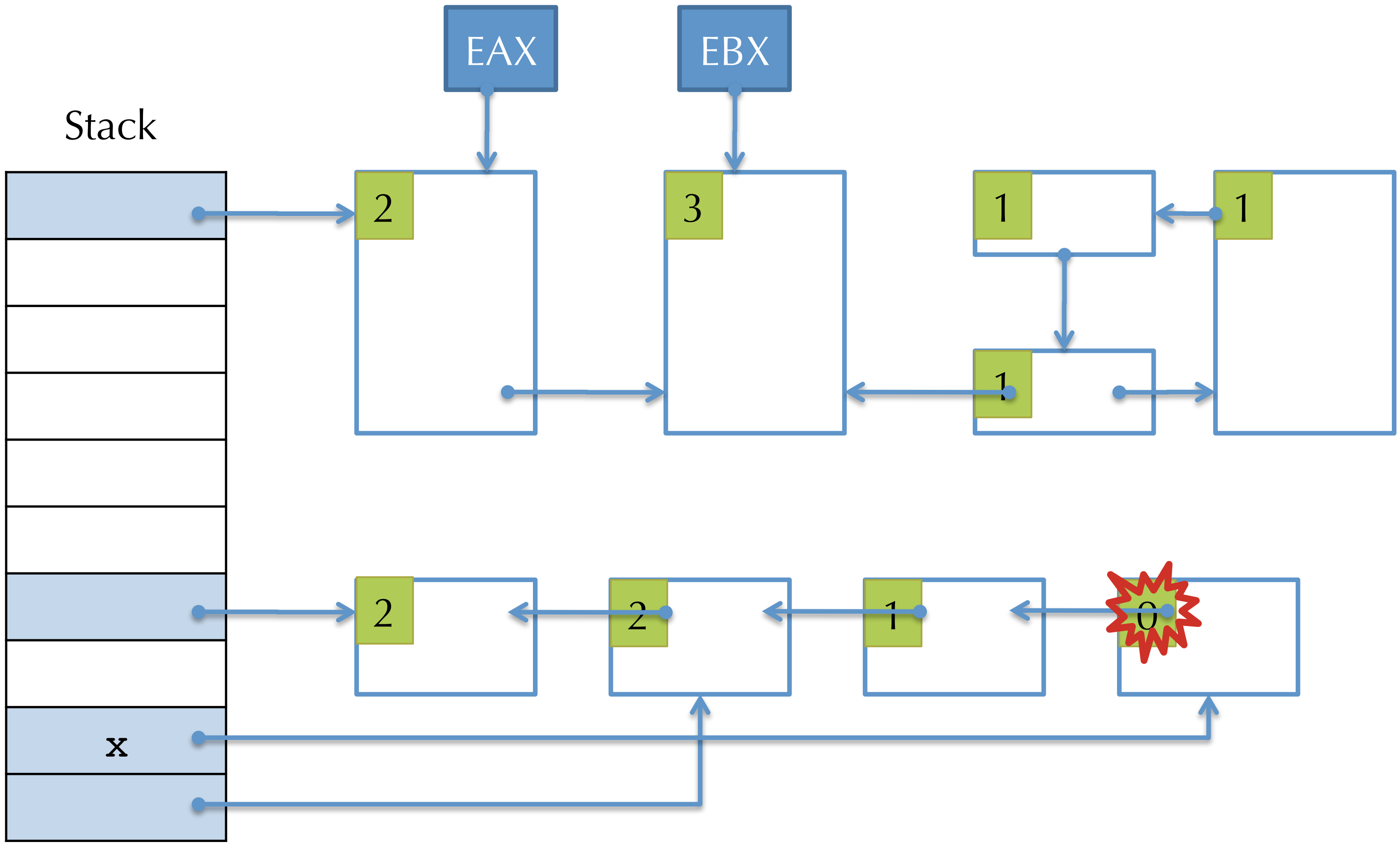
Example Reference Counts

- Objects track reference counts.



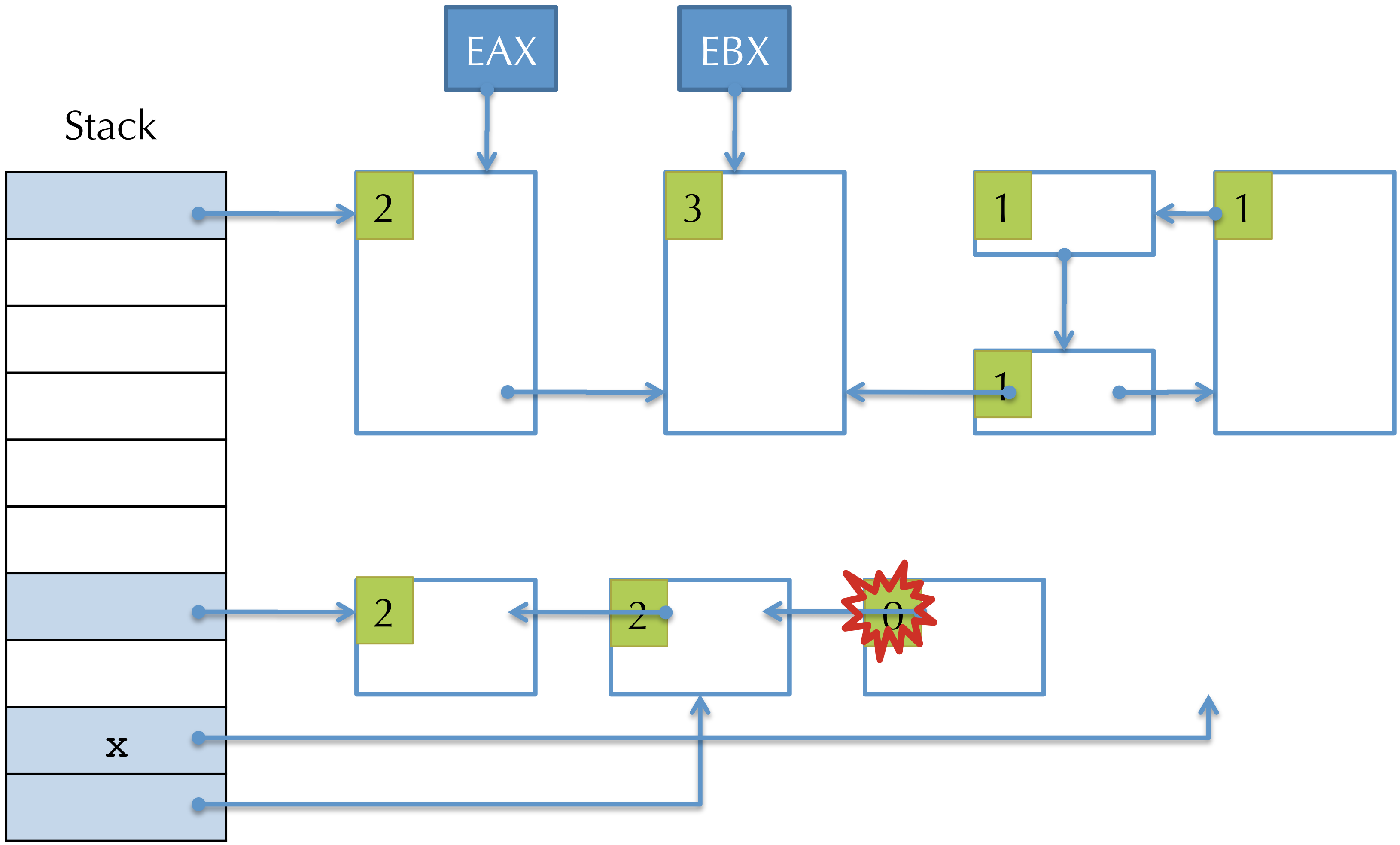
Example Reference Counts

- On `free(x)`



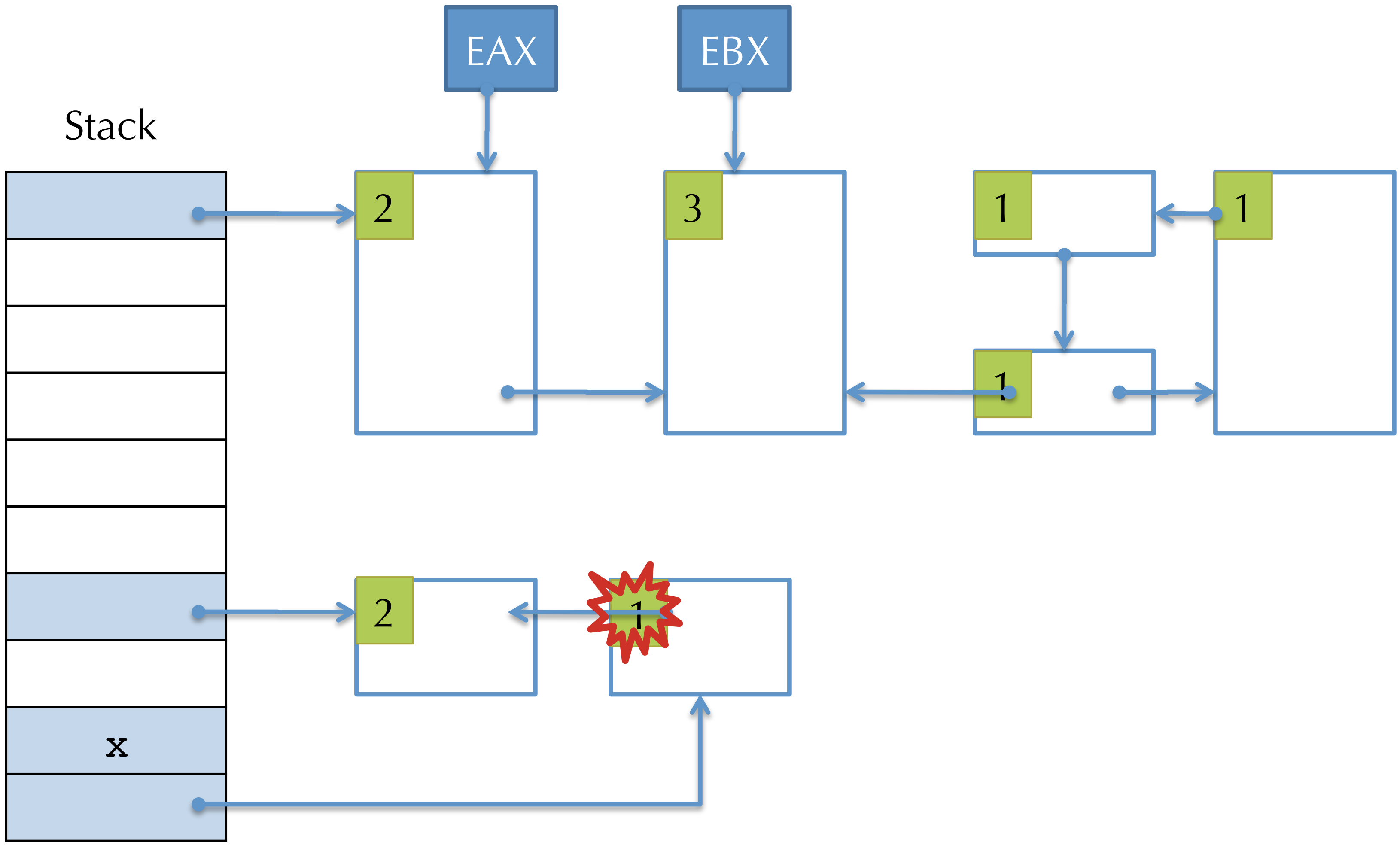
Example Reference Counts

- On `free(x)`

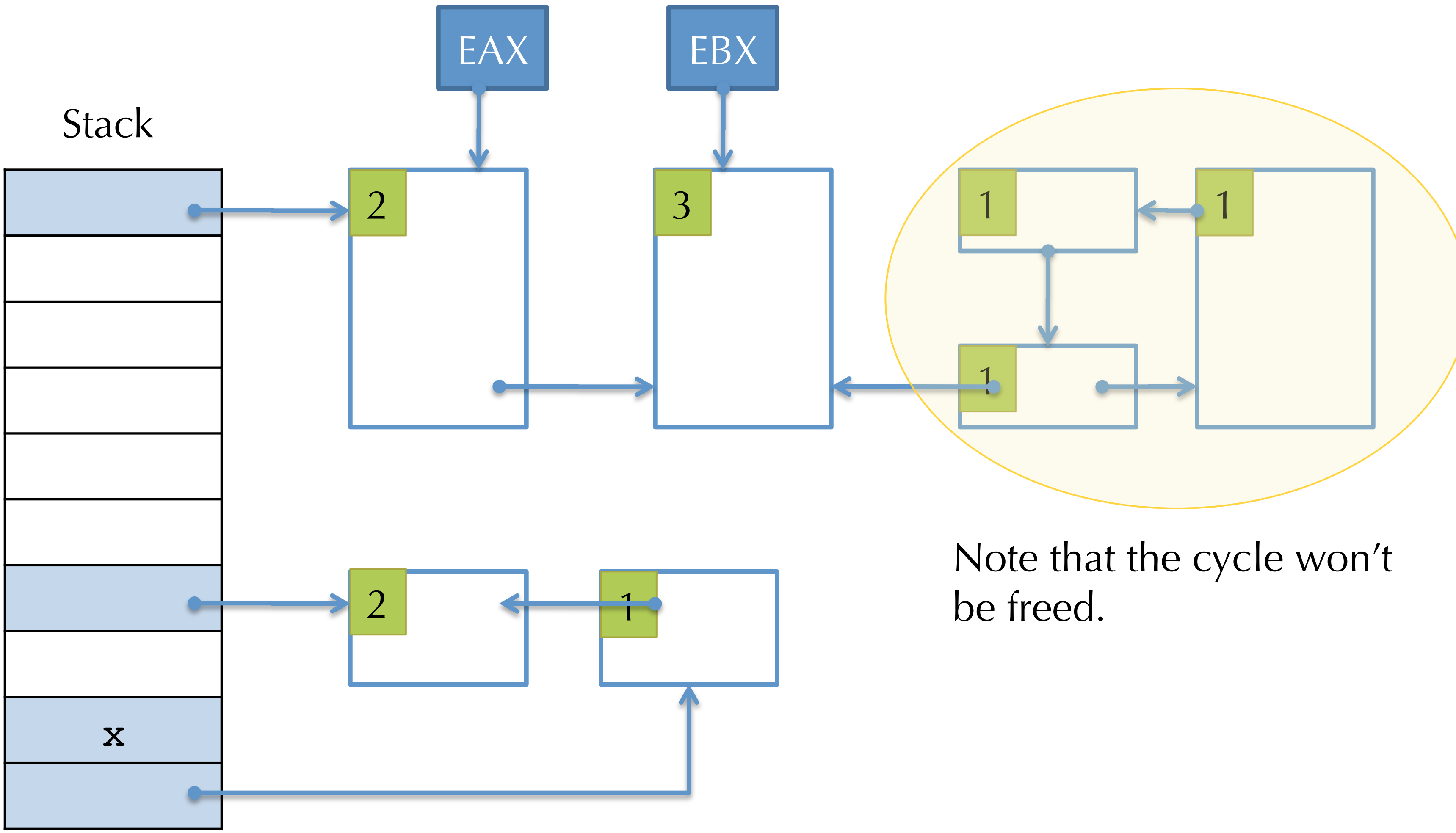


Example Reference Counts

- On `free(x)`



Example Reference Counts



Note that the cycle won't be freed.

Dealing with Cycles

- Option 1: Require programmers to explicitly null-out references to break cycles.
- Option 2: Periodically run GC to collect cycles
- Option 3: Require programmers to distinguish “weak pointers” from “strong pointers”
  - *weak pointers*: if all references to an object are “weak” then the object can be freed even with non-zero reference count.
  - “Back edges” in the object graph should be designated as weak
  - (Aside: weak pointers useful in GC settings too.)
- In practice: Reference counts
  - Apples Cocoa framework used ref counts, recent versions use GC
  - iOS supports “automatic reference counting”