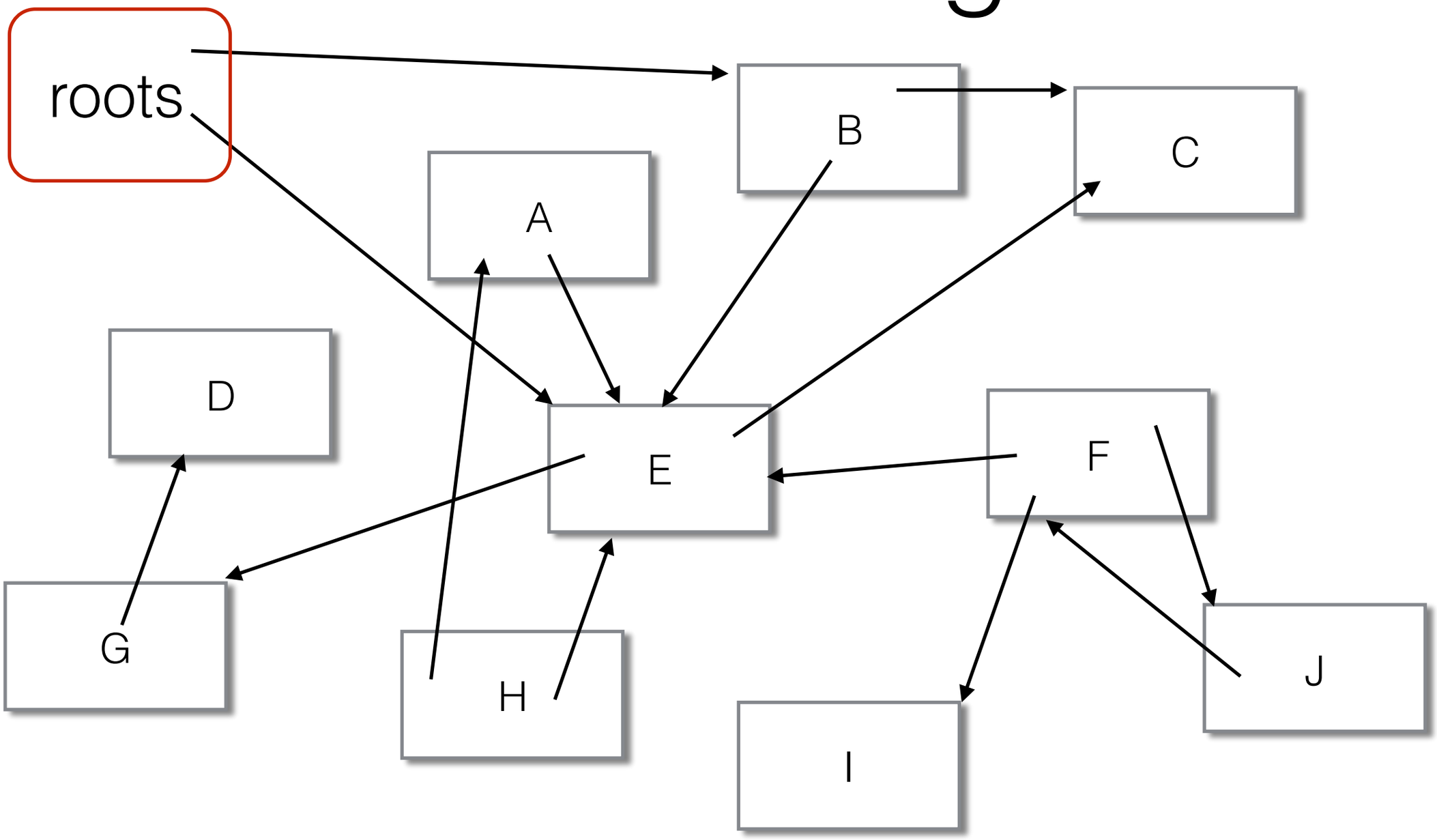
Memory Management

Part Two

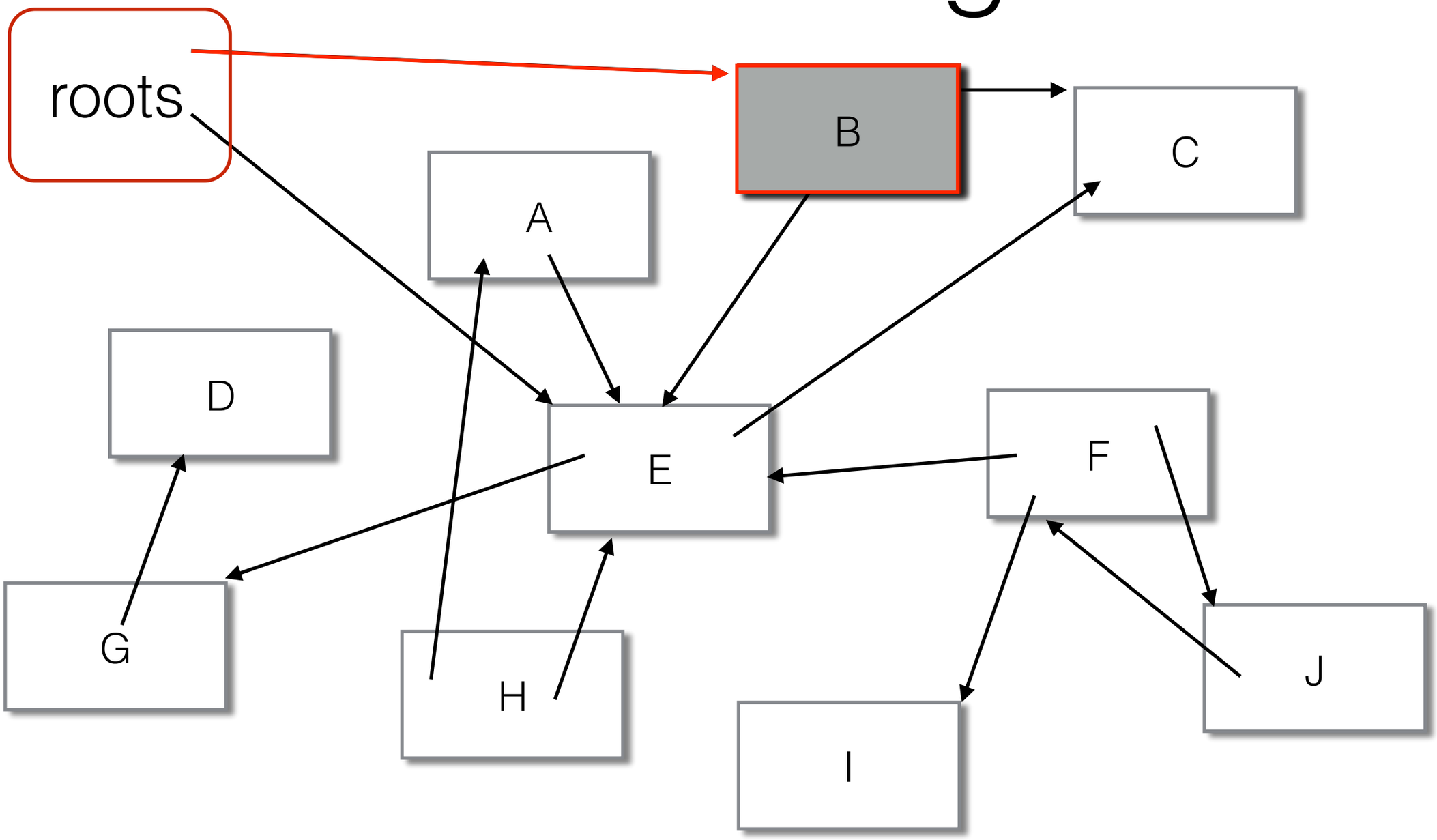
The tri-color abstraction

- Introduced in a 1976 paper, this is commonly used to describe or reason about collectors.
- We partition the graph into three colors of node:
  - White nodes may be dead or alive (we don't know which, yet). Initially, all nodes are **white**.
  - When a node is first encountered during tracing (i.e., is known to be live), it is colored **grey**.
  - When a grey node has been scanned and all its children have been identified, it is colored **black**.
- Tracing proceeds from each grey node until there are no grey nodes remaining. White nodes remaining at the end of tracing are dead.

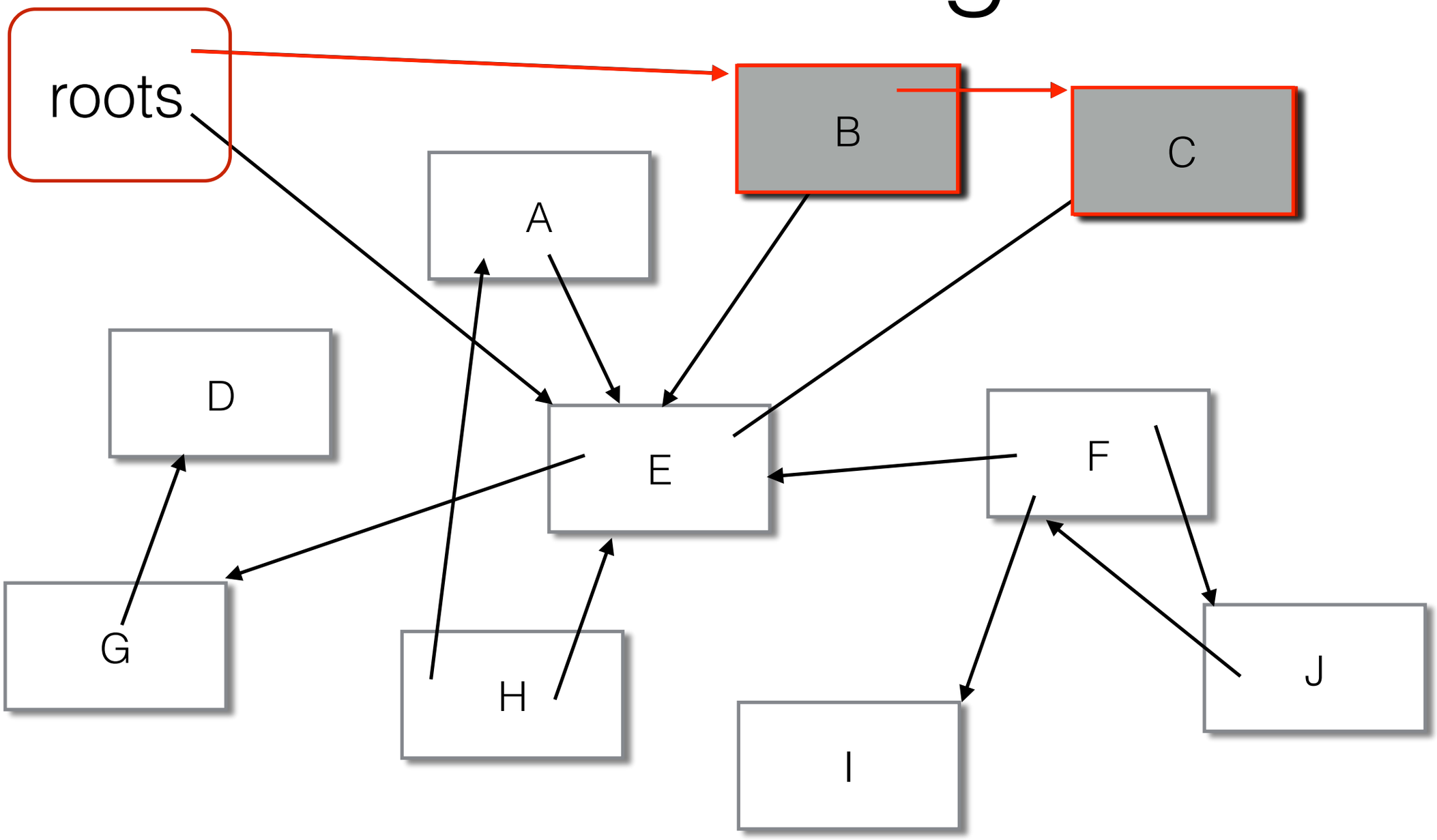
The tri-color abstract in marking



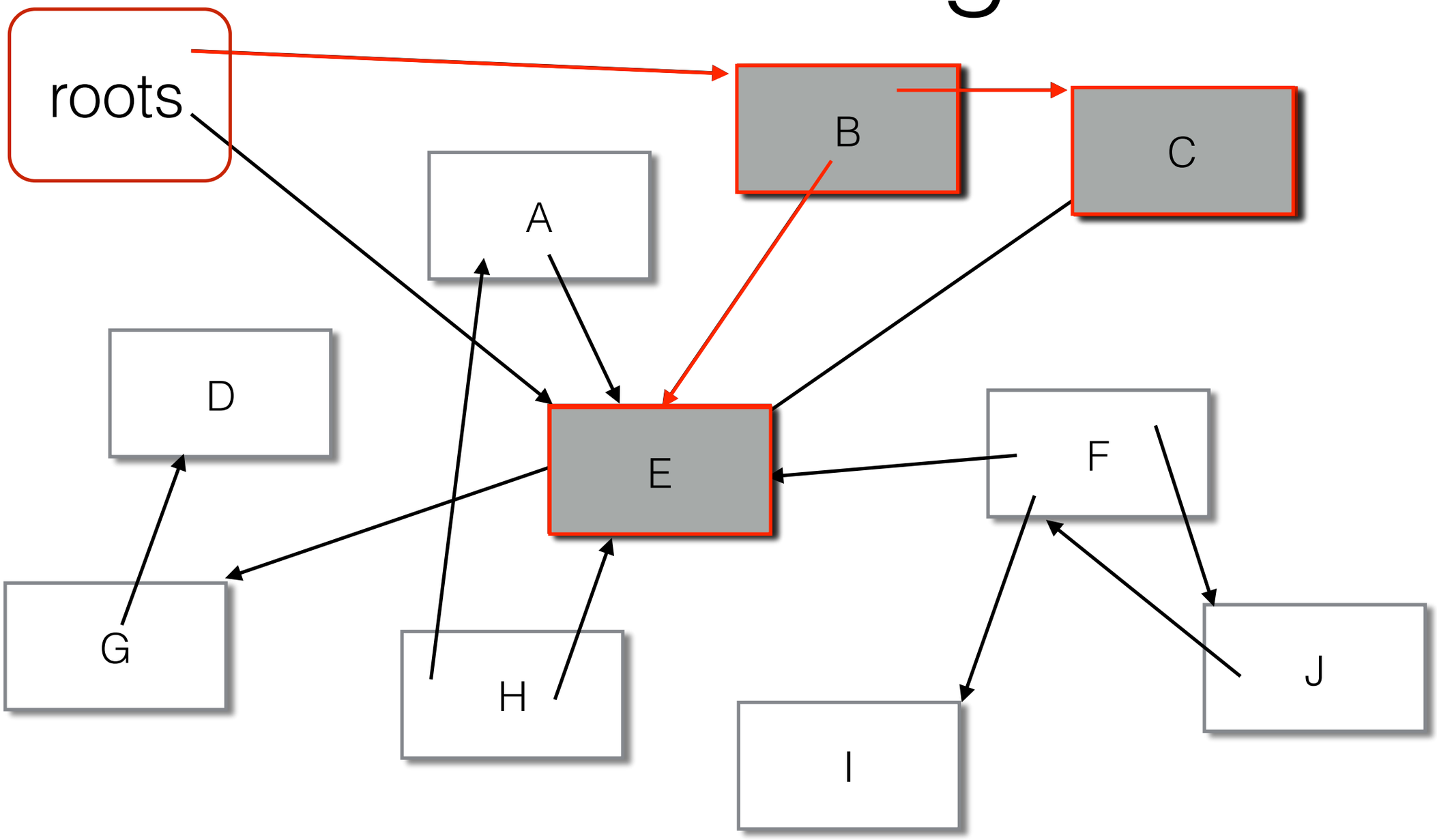
The tri-color abstract in marking



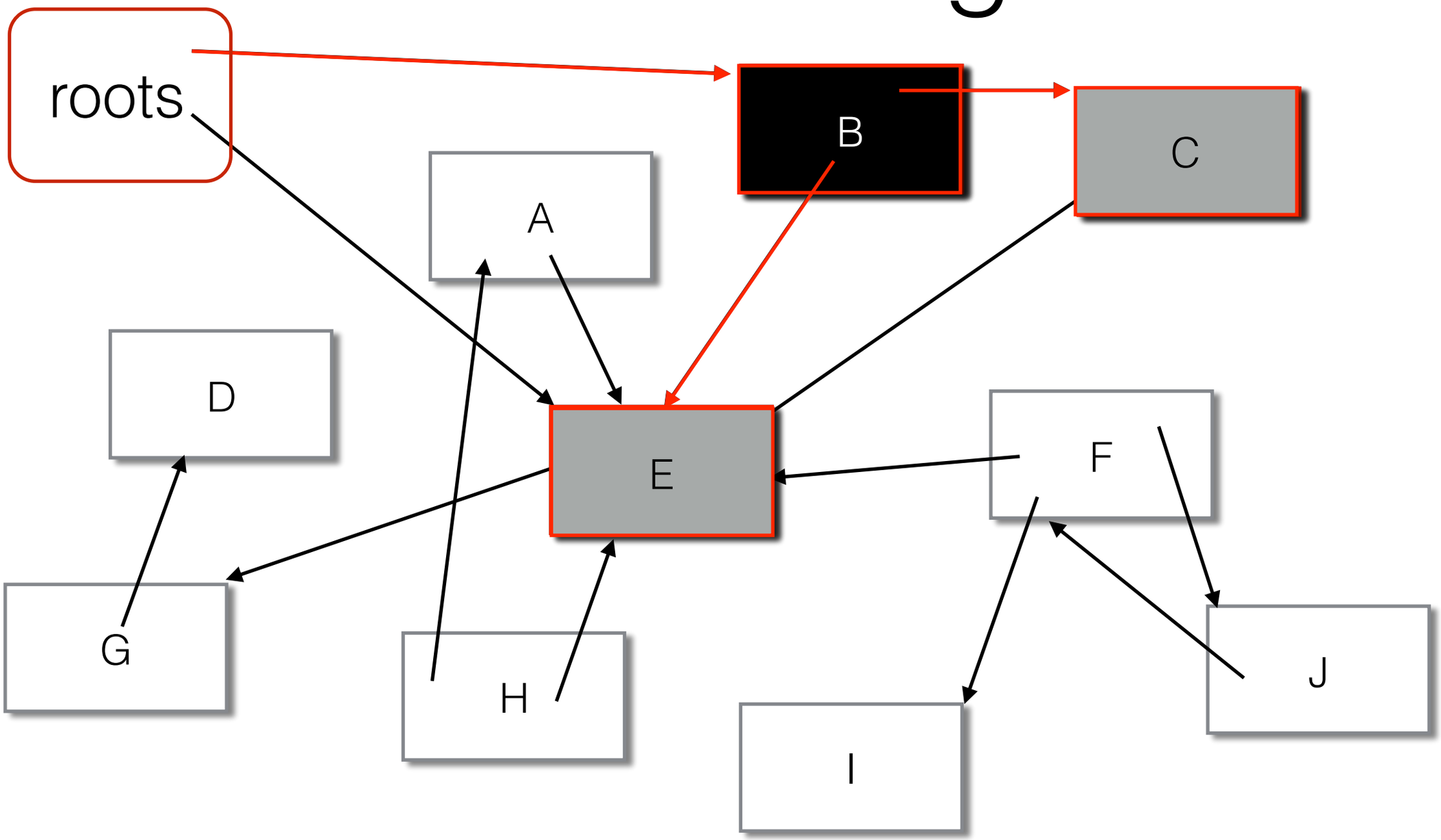
The tri-color abstract in marking



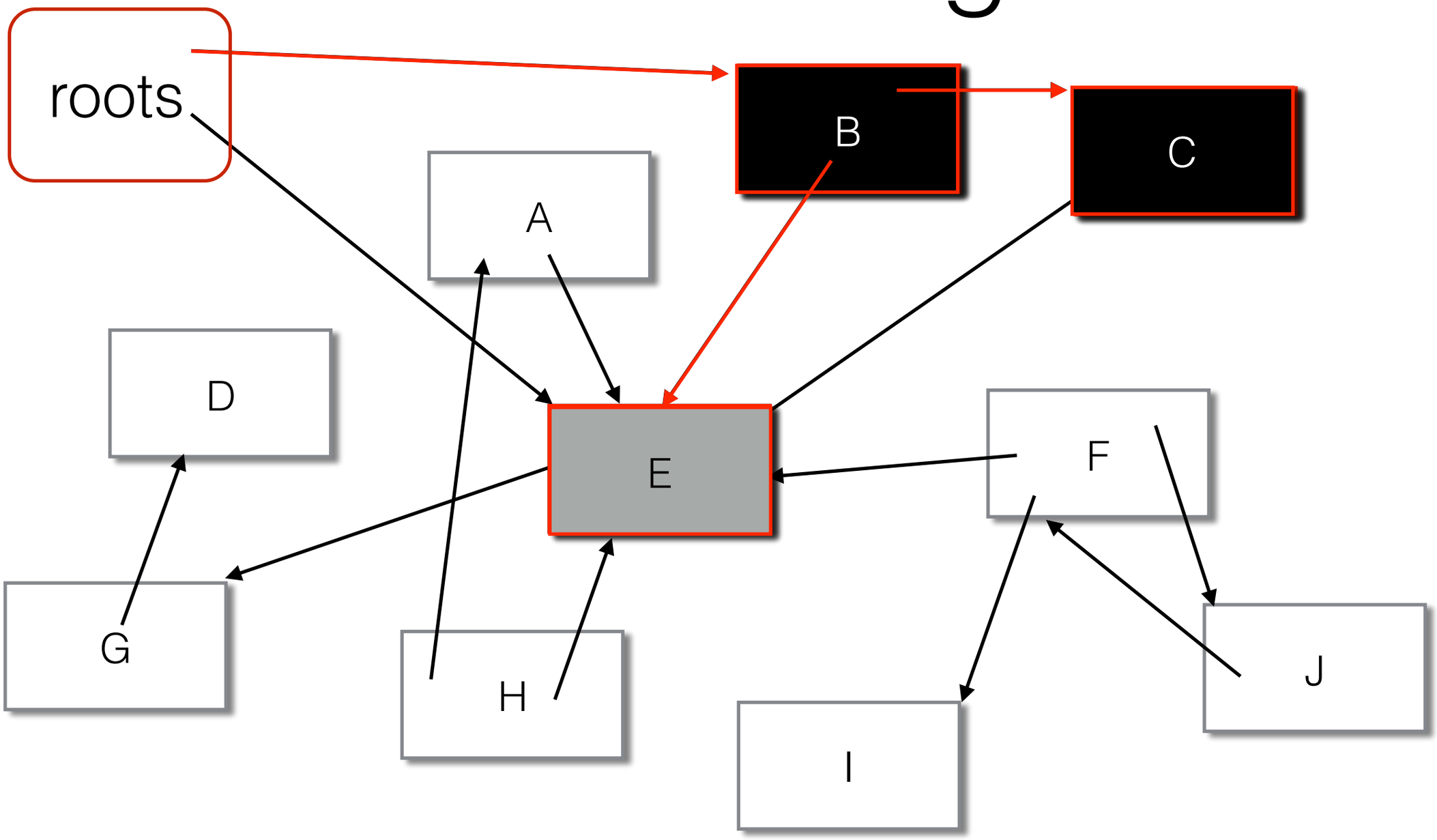
The tri-color abstract in marking



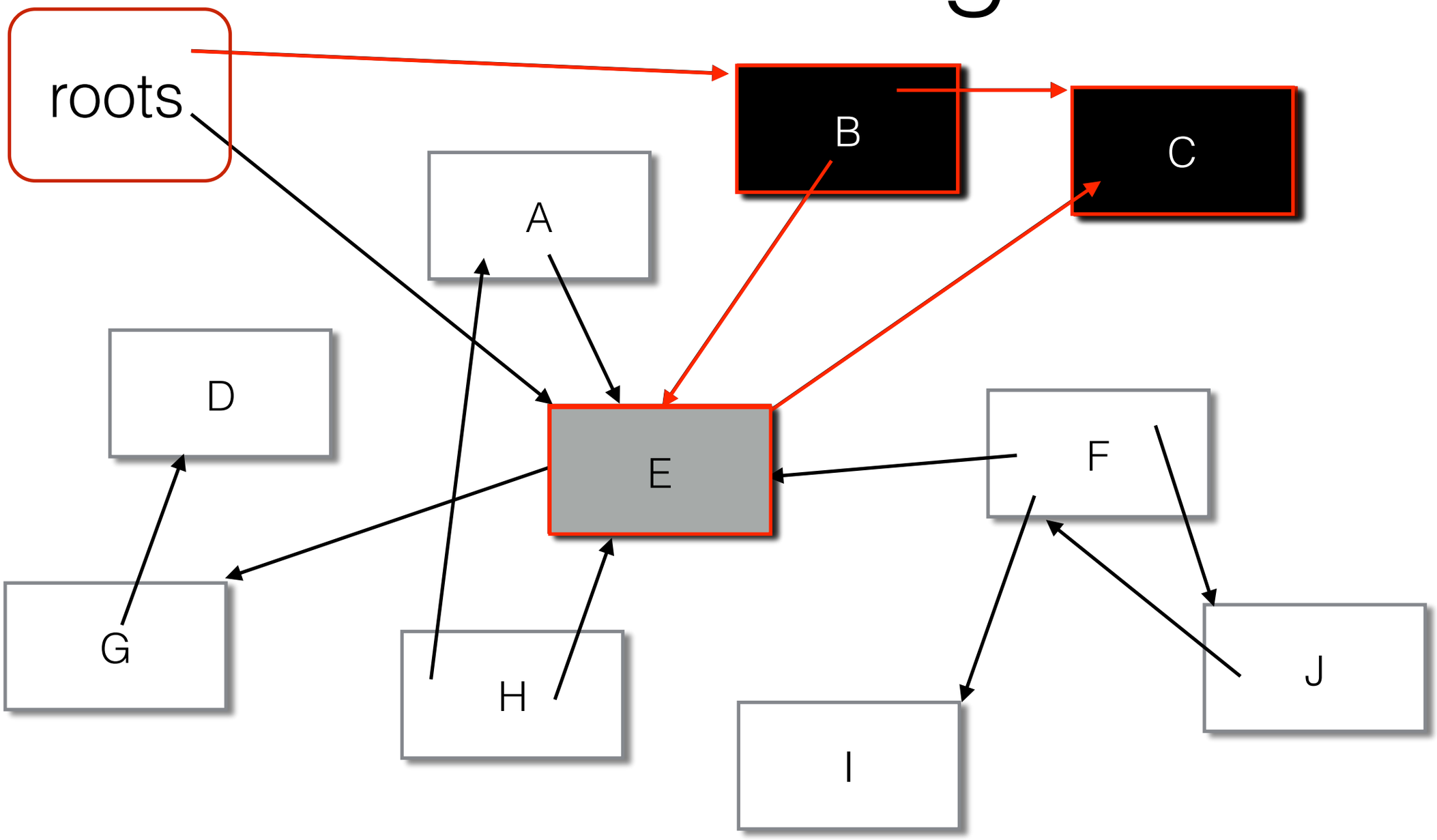
The tri-color abstract in marking



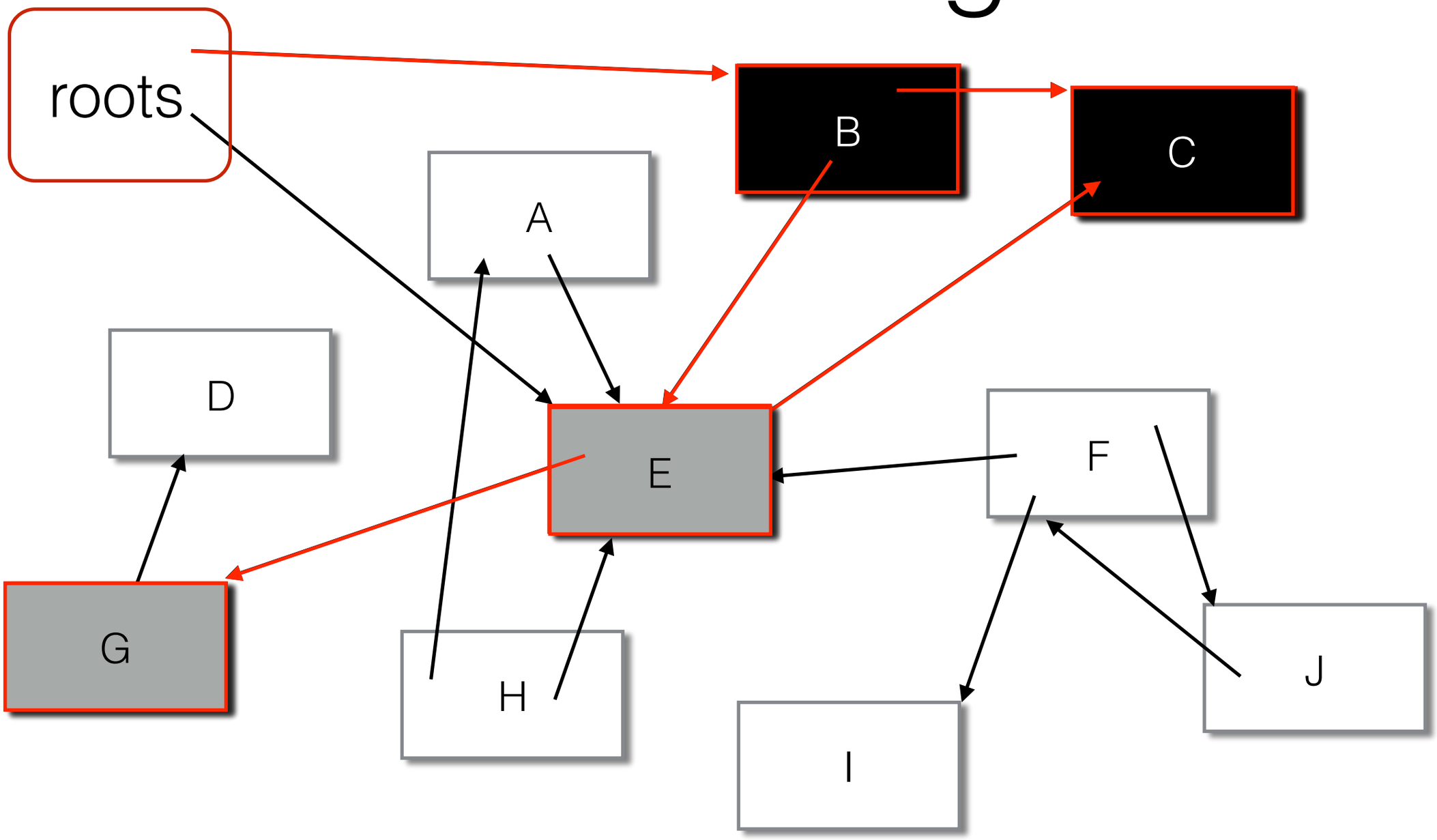
The tri-color abstract in marking



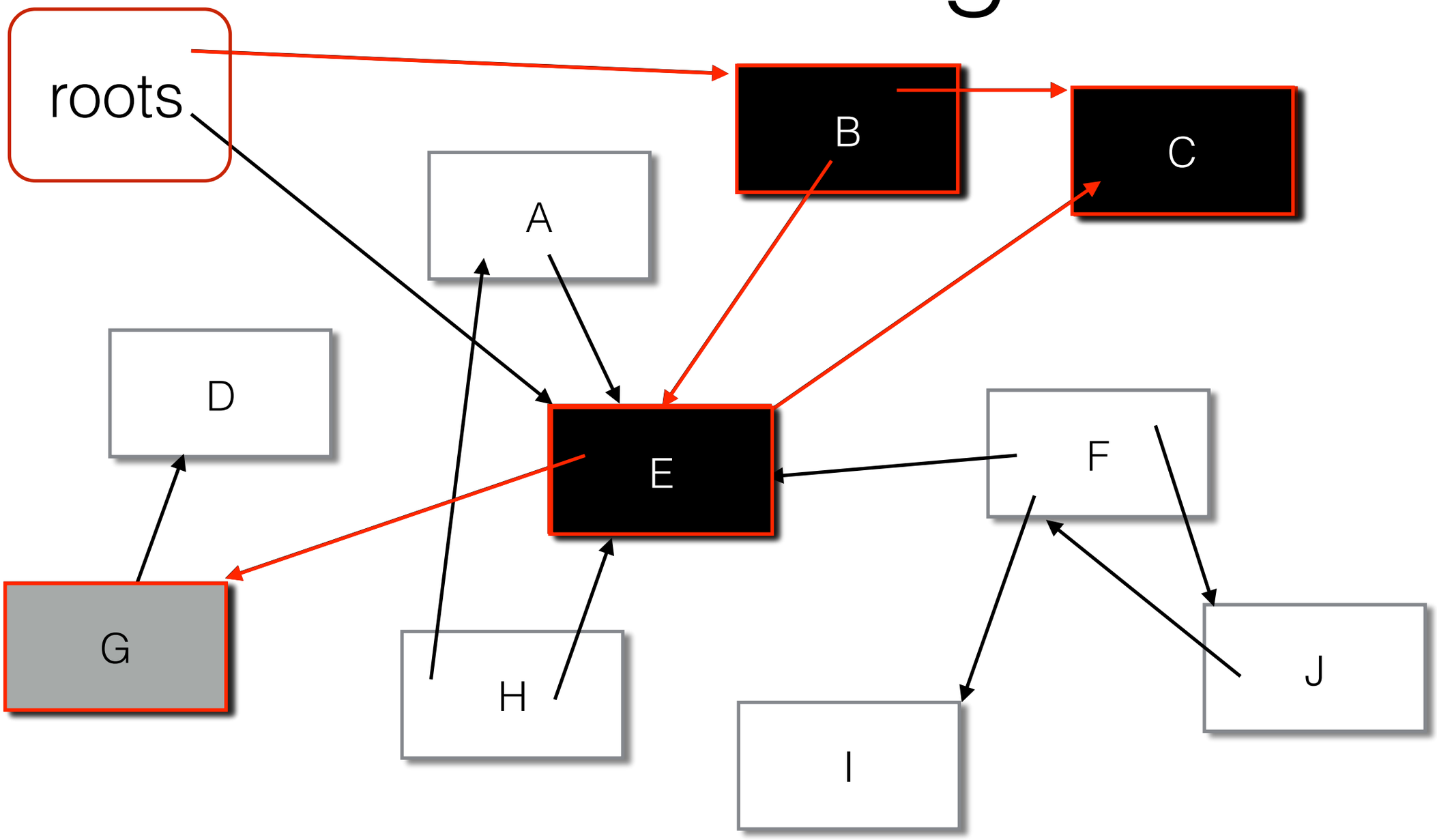
The tri-color abstract in marking



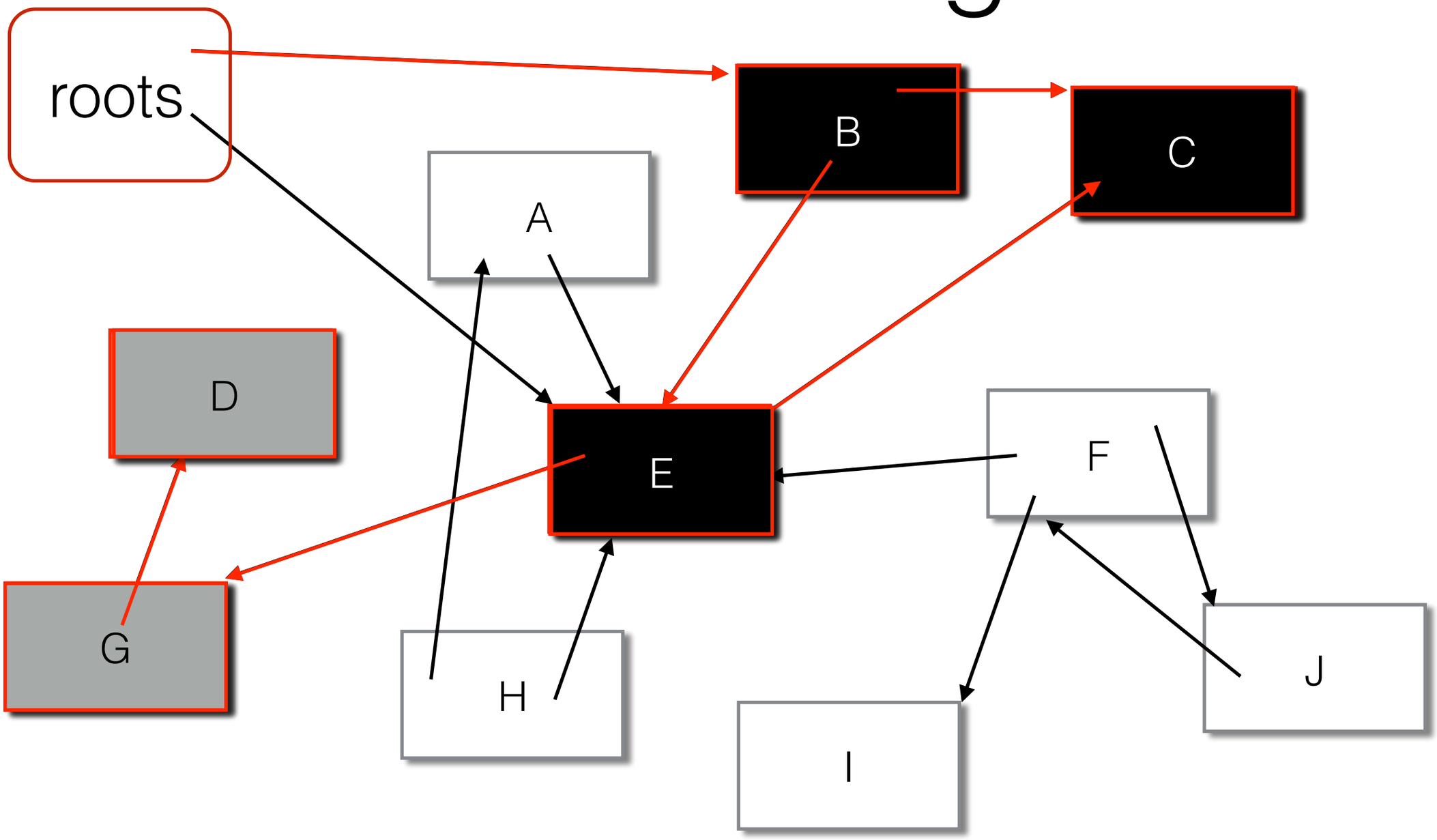
The tri-color abstract in marking



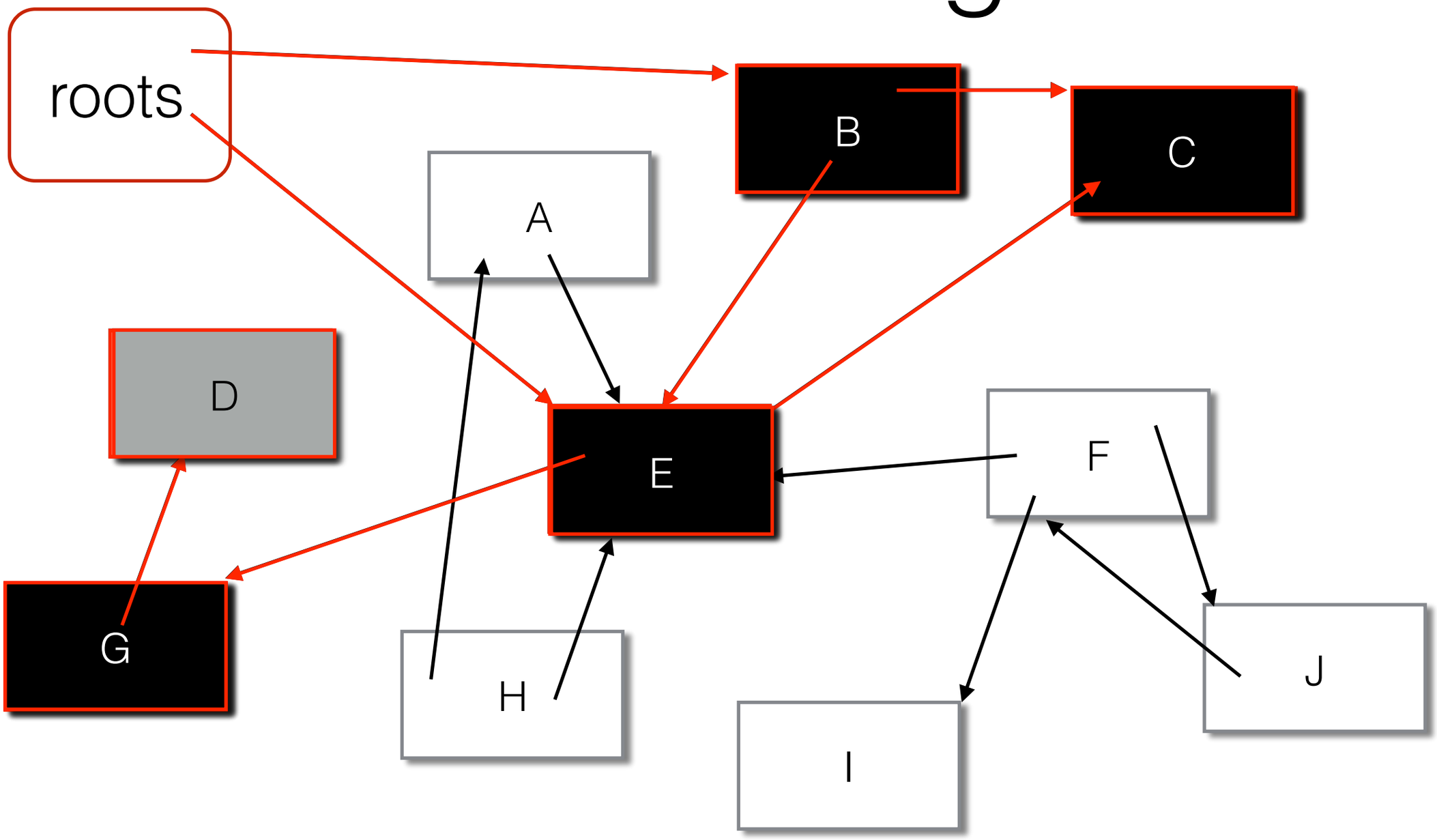
The tri-color abstract in marking



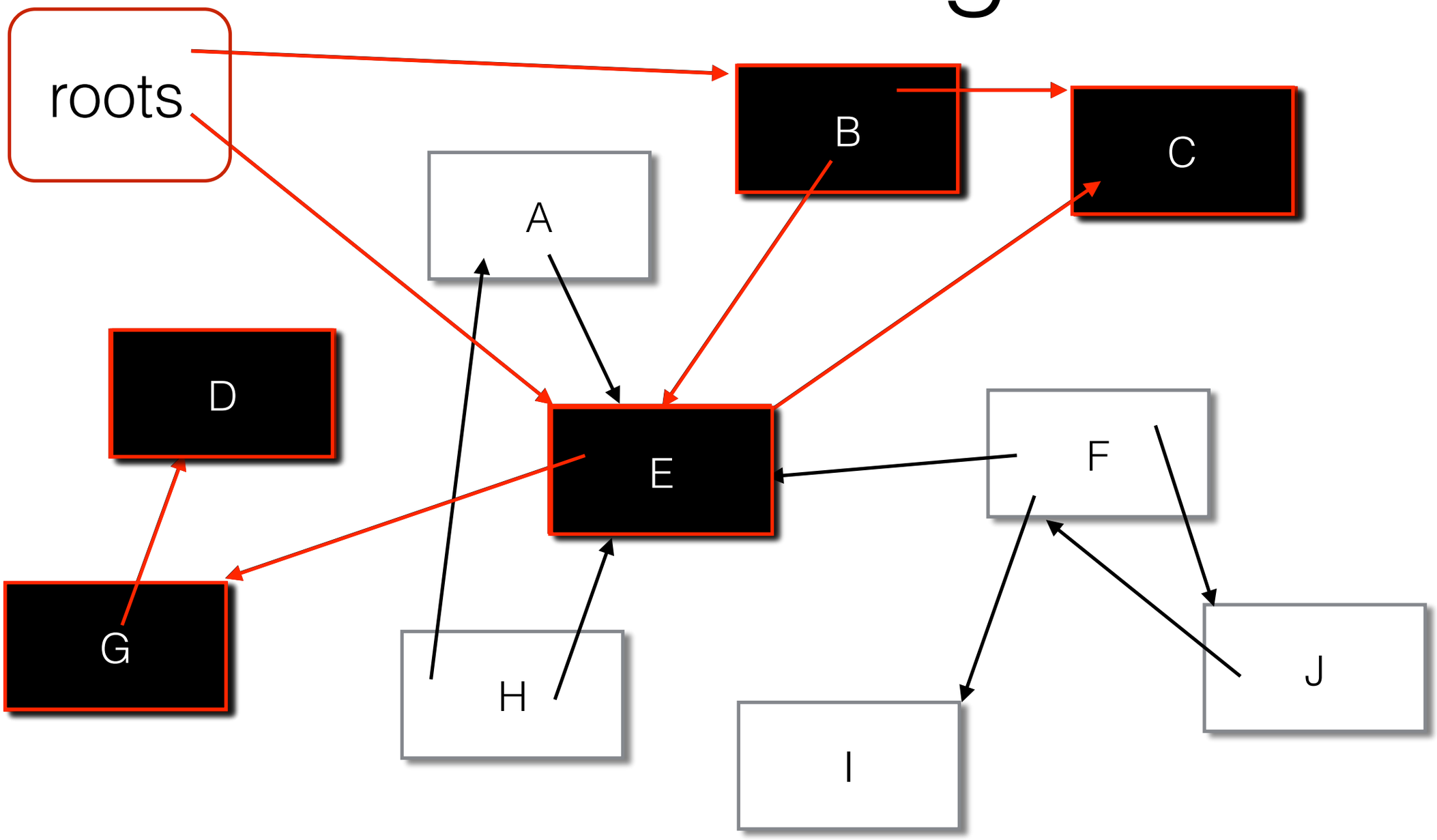
The tri-color abstract in marking



The tri-color abstract in marking



The tri-color abstract in marking

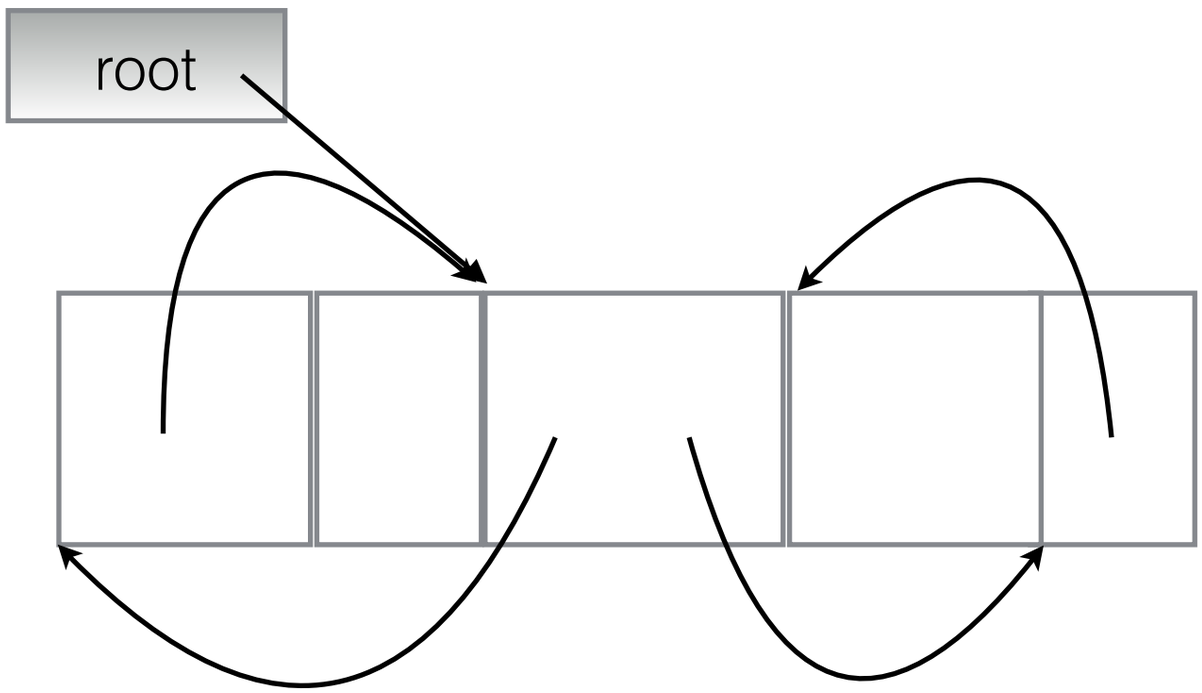


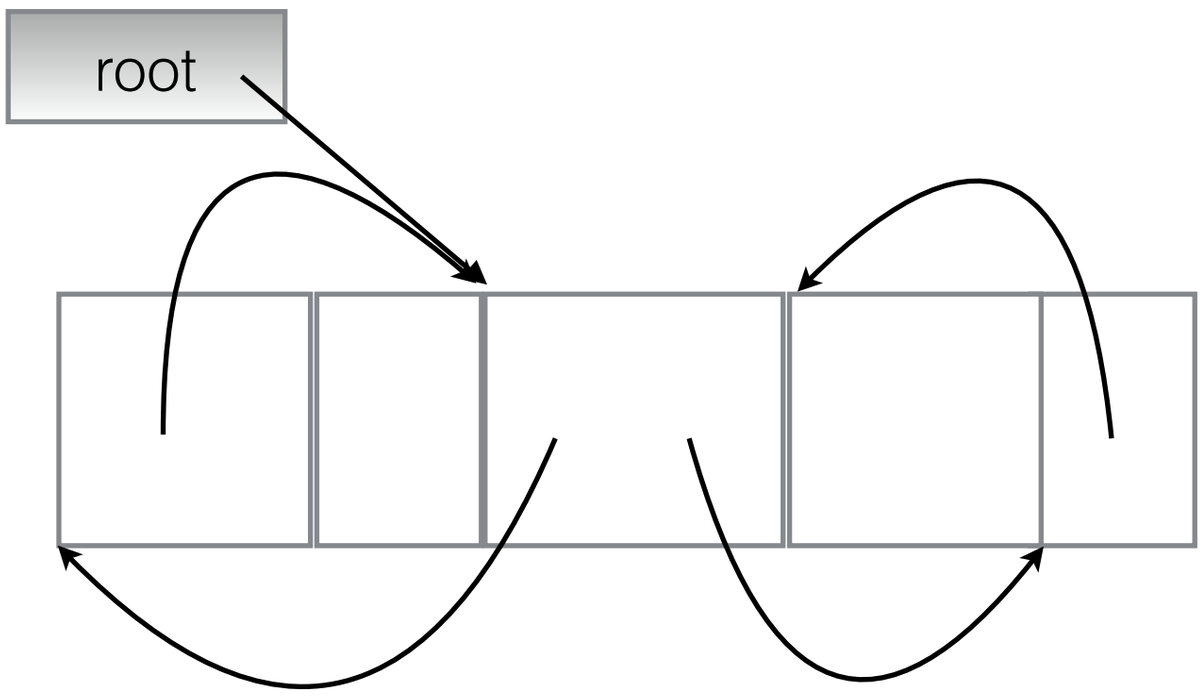
The tri-color abstraction in copying

root

root

root

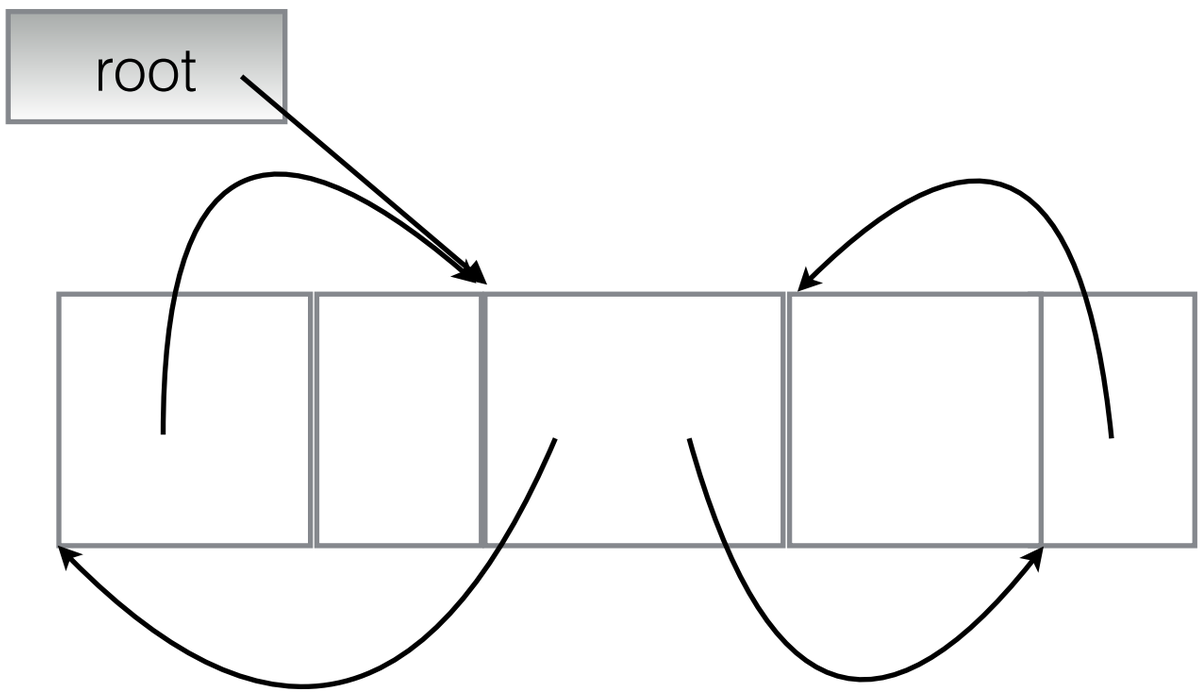

root



From

To

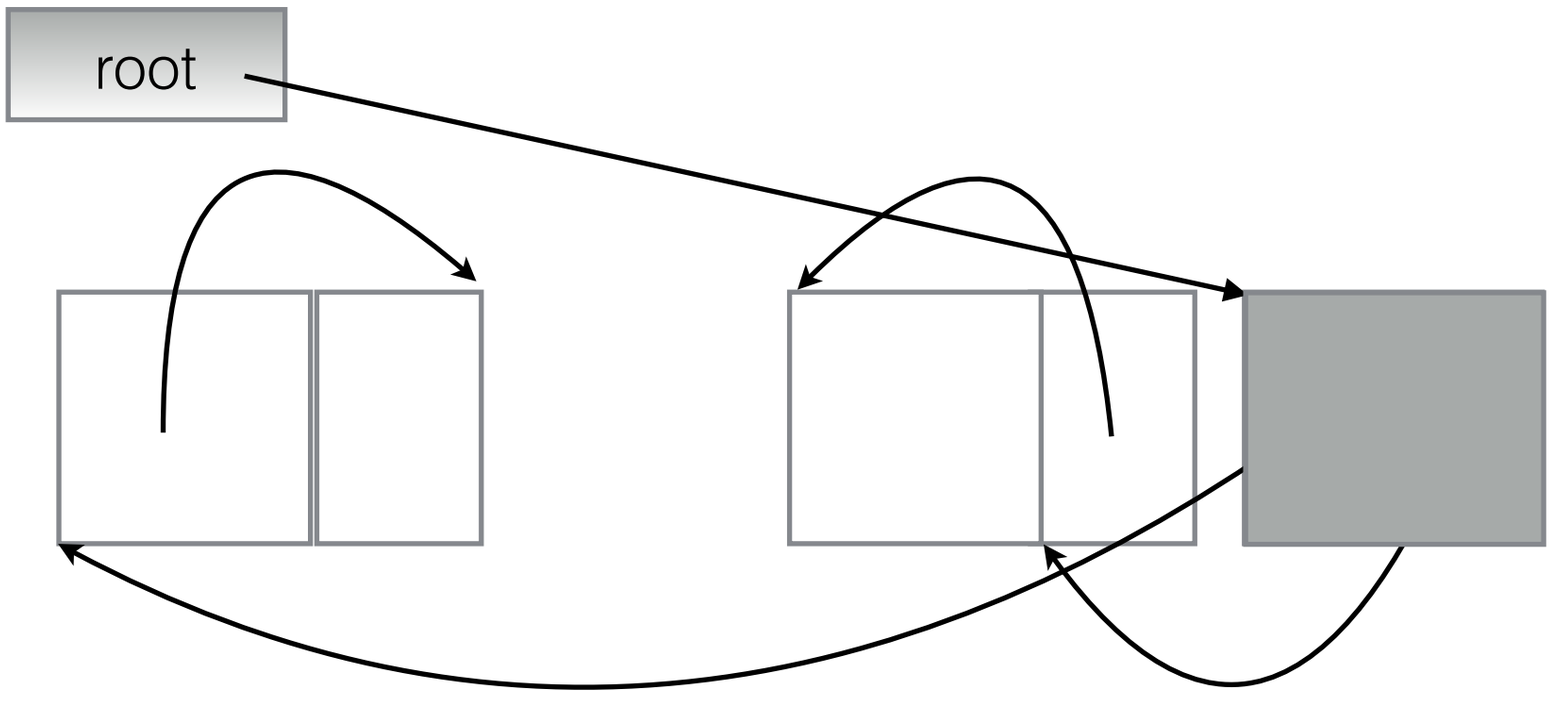
Process roots



From

To

Process roots

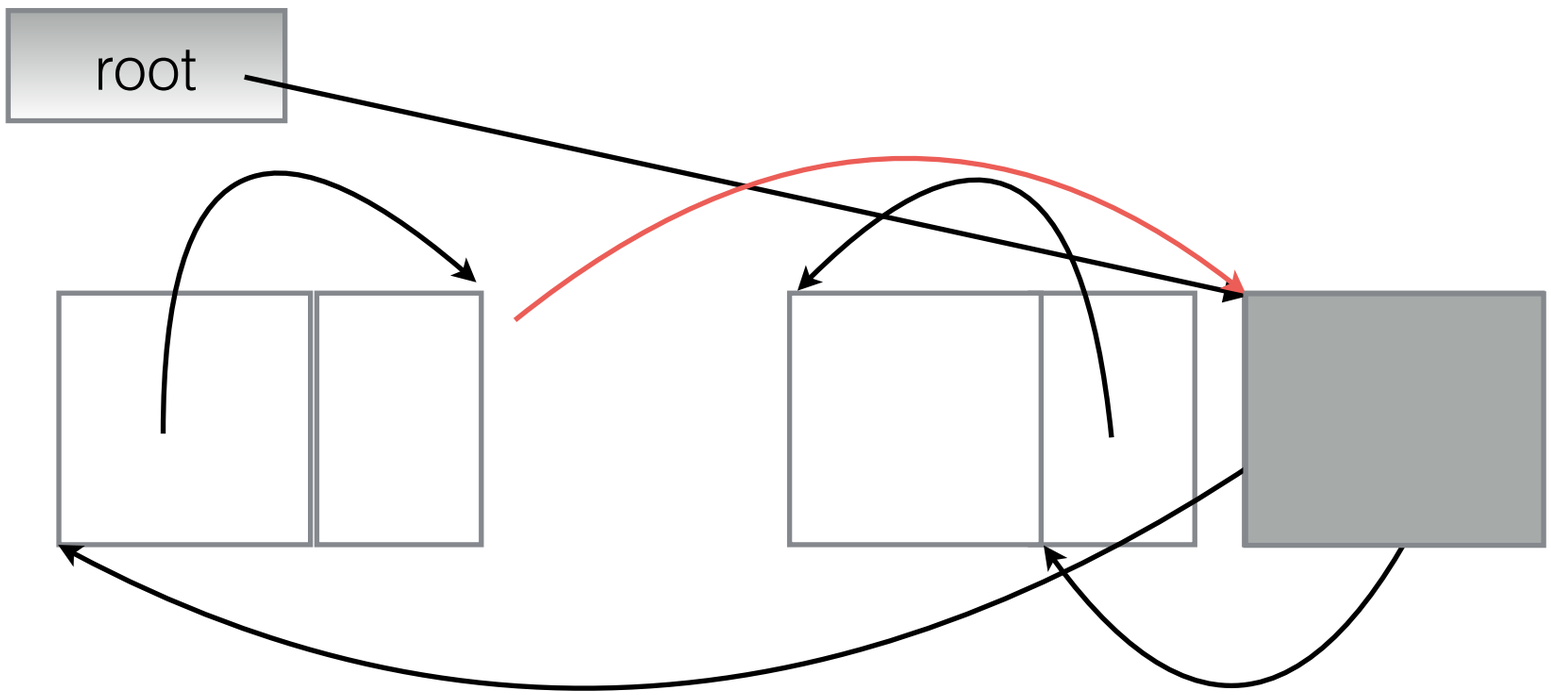


From

To

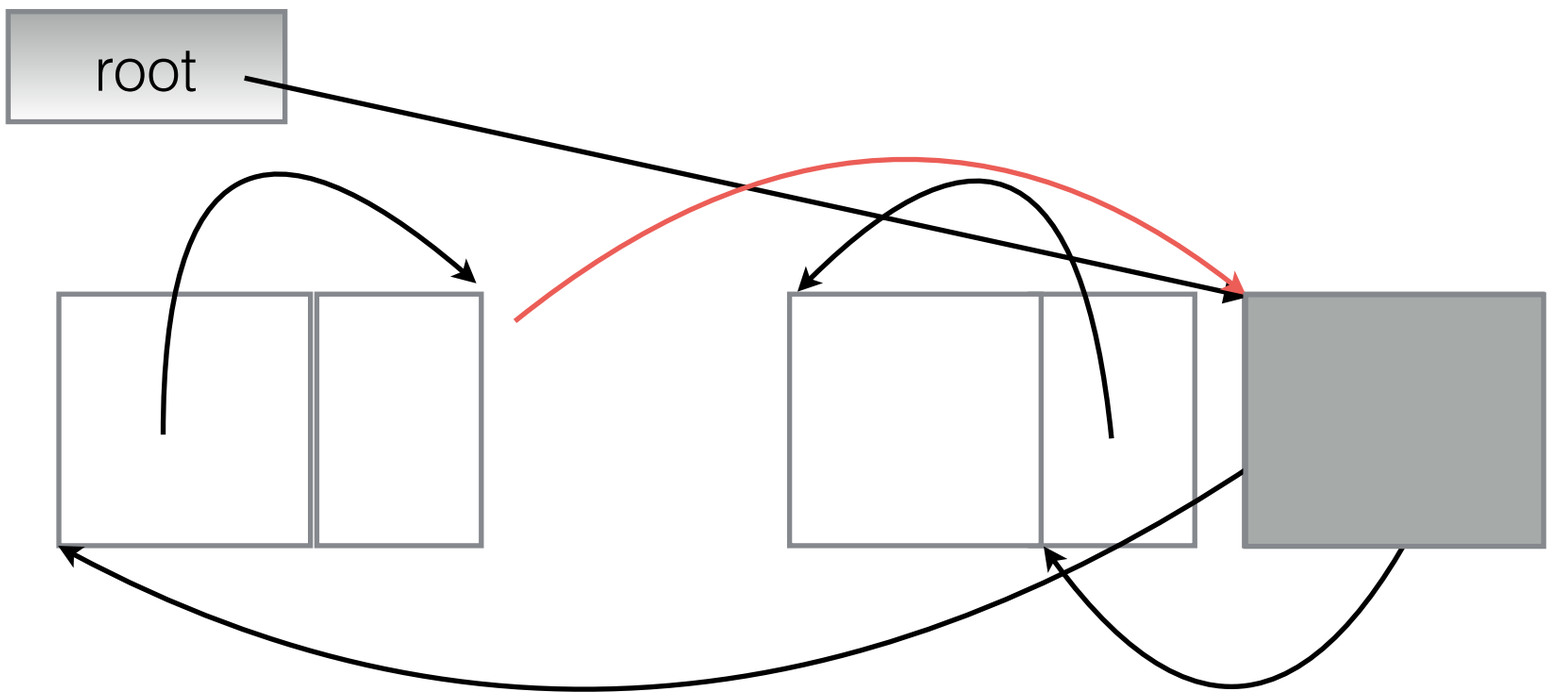
Process roots

Forwarding pointer



From

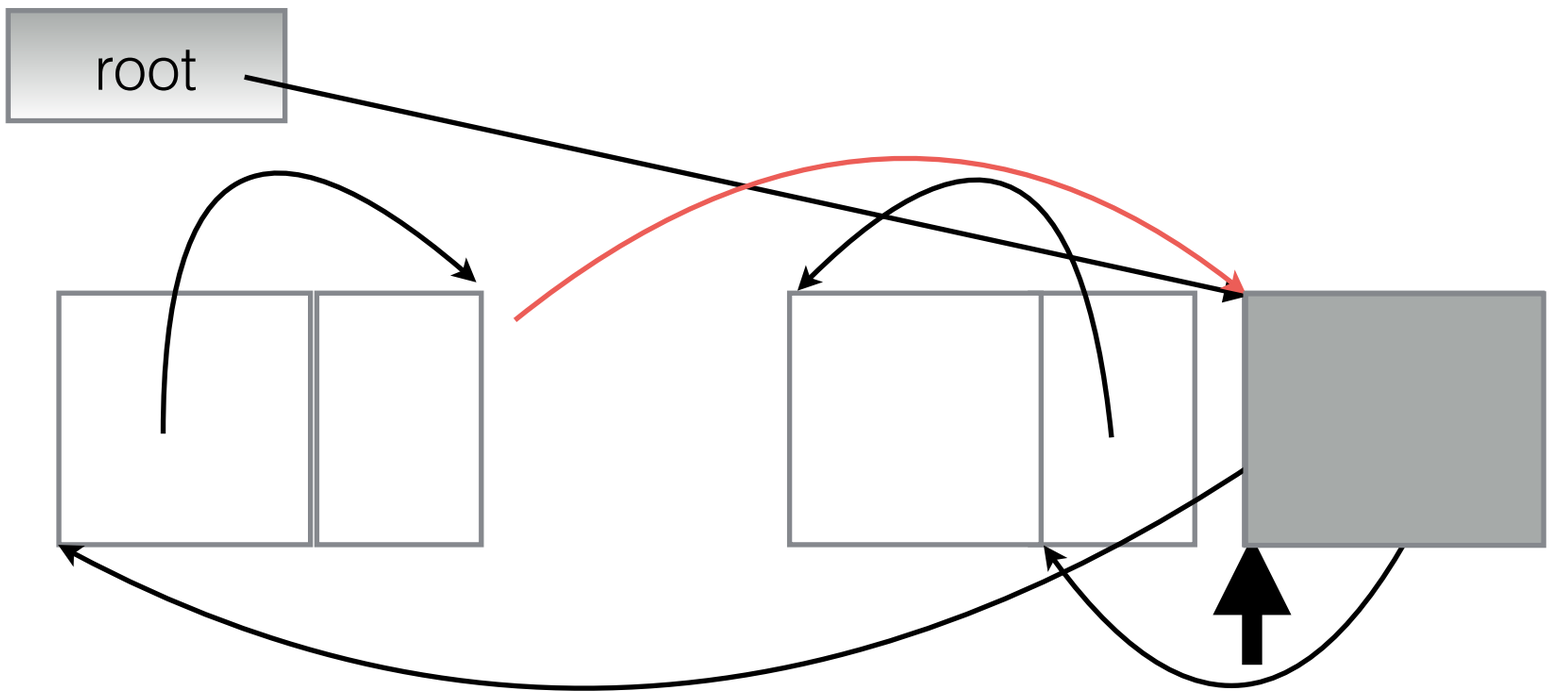
To



From

To

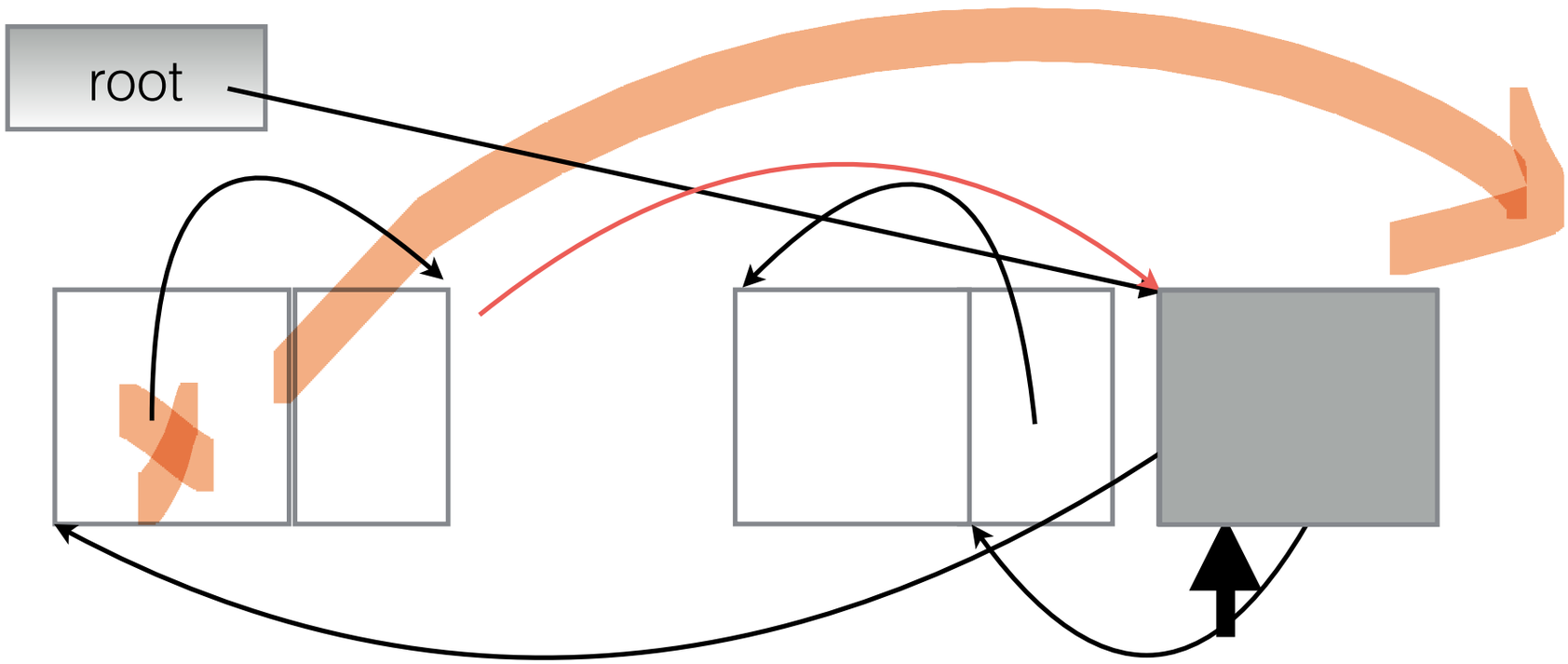
Begin scan



From

To

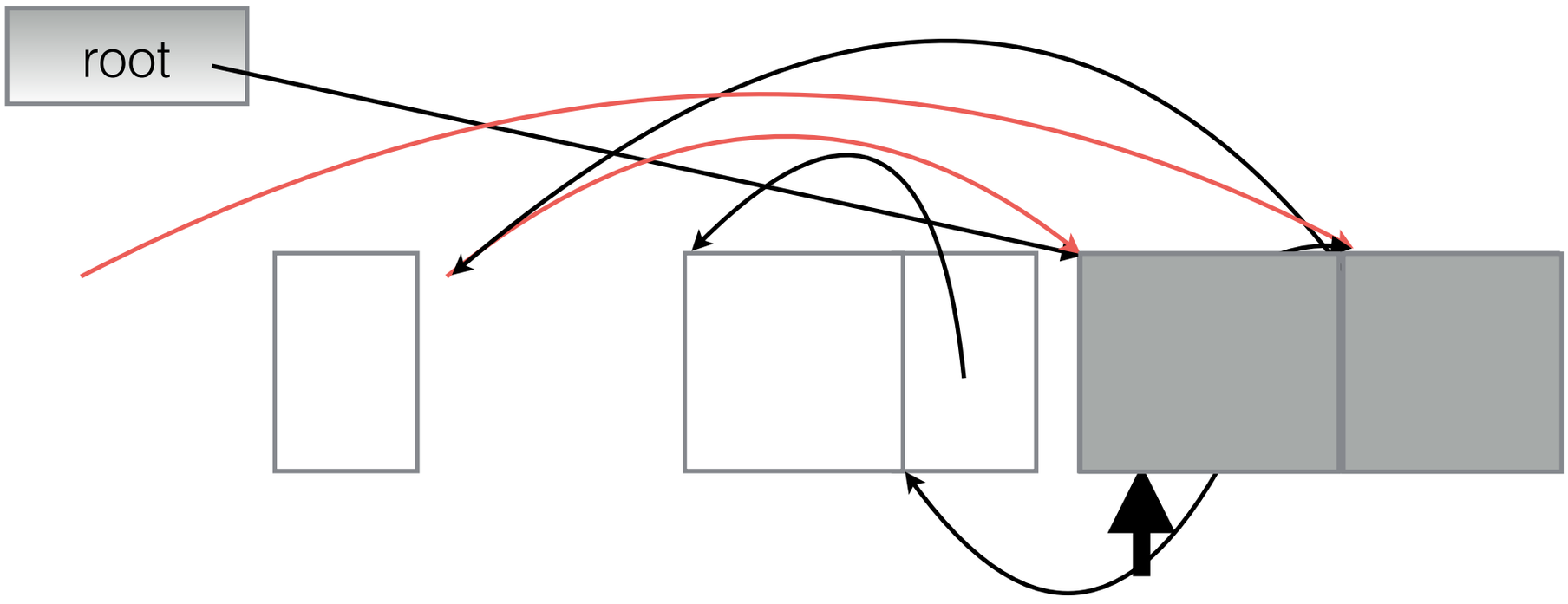
Scan



From

To

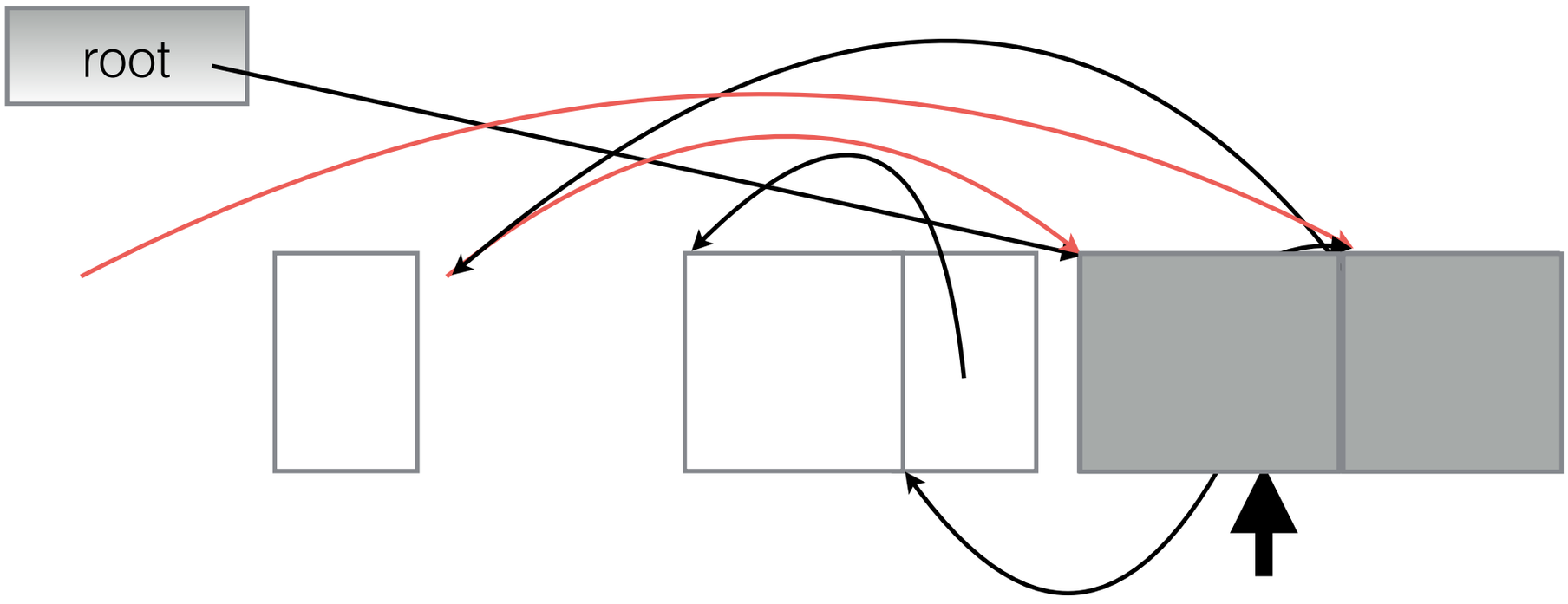
Scan



From

To

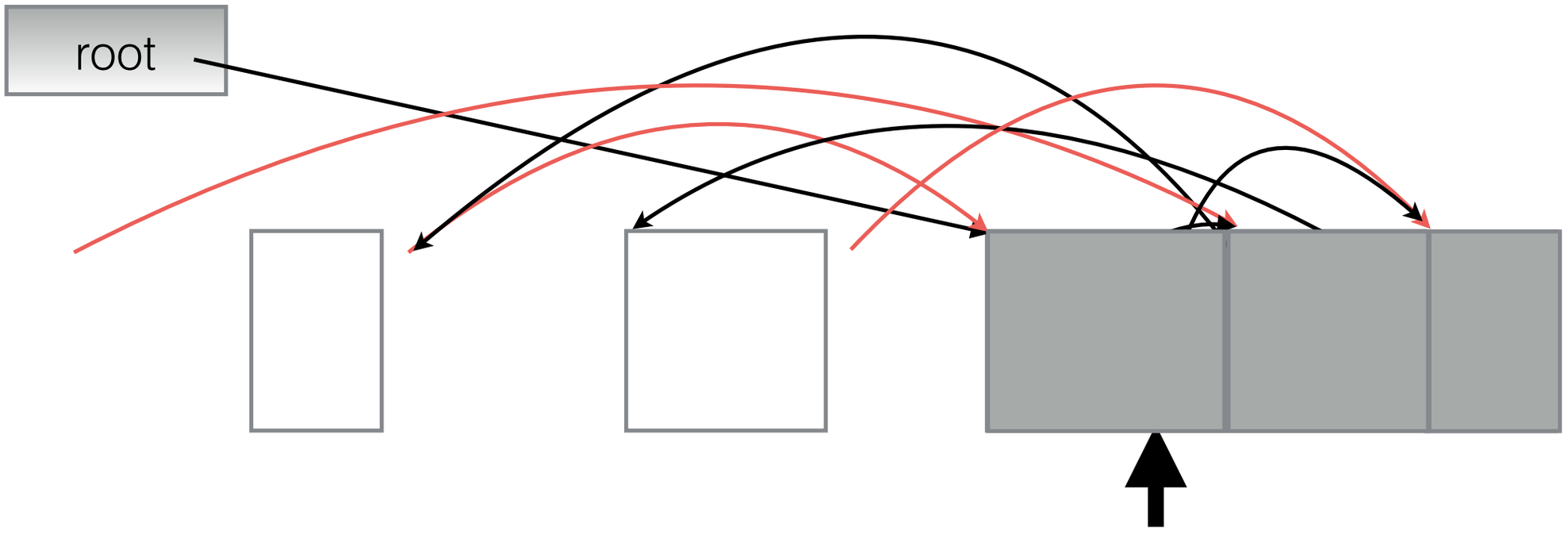
Scan



From

To

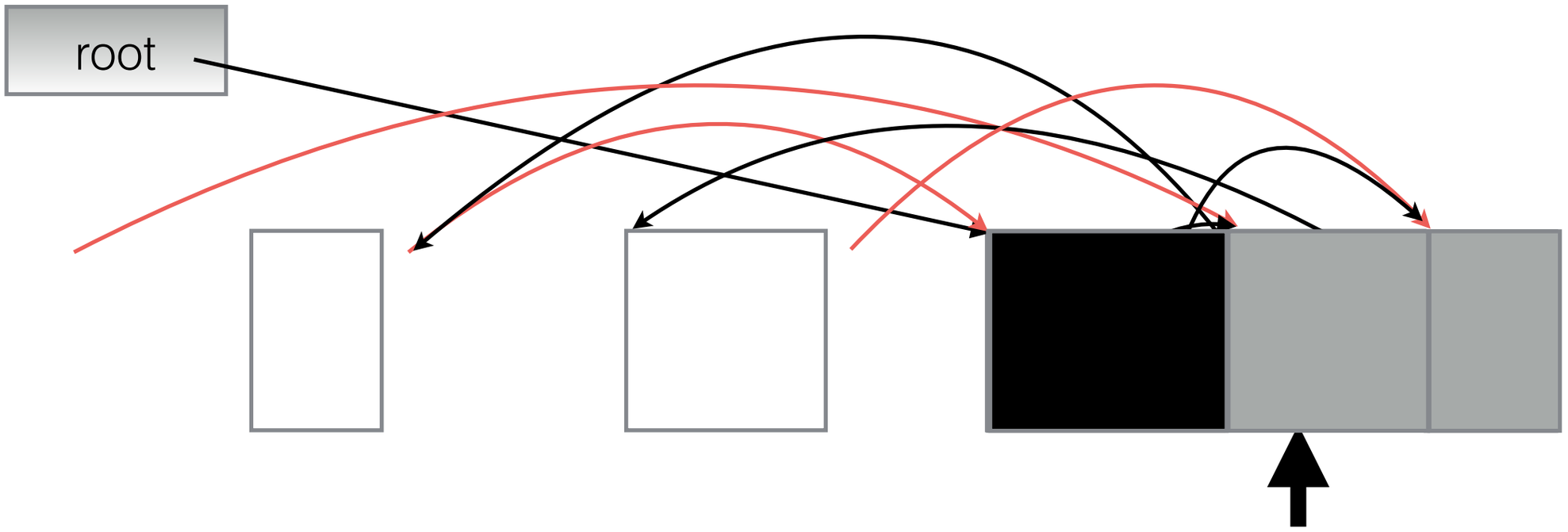
Scan



From

To

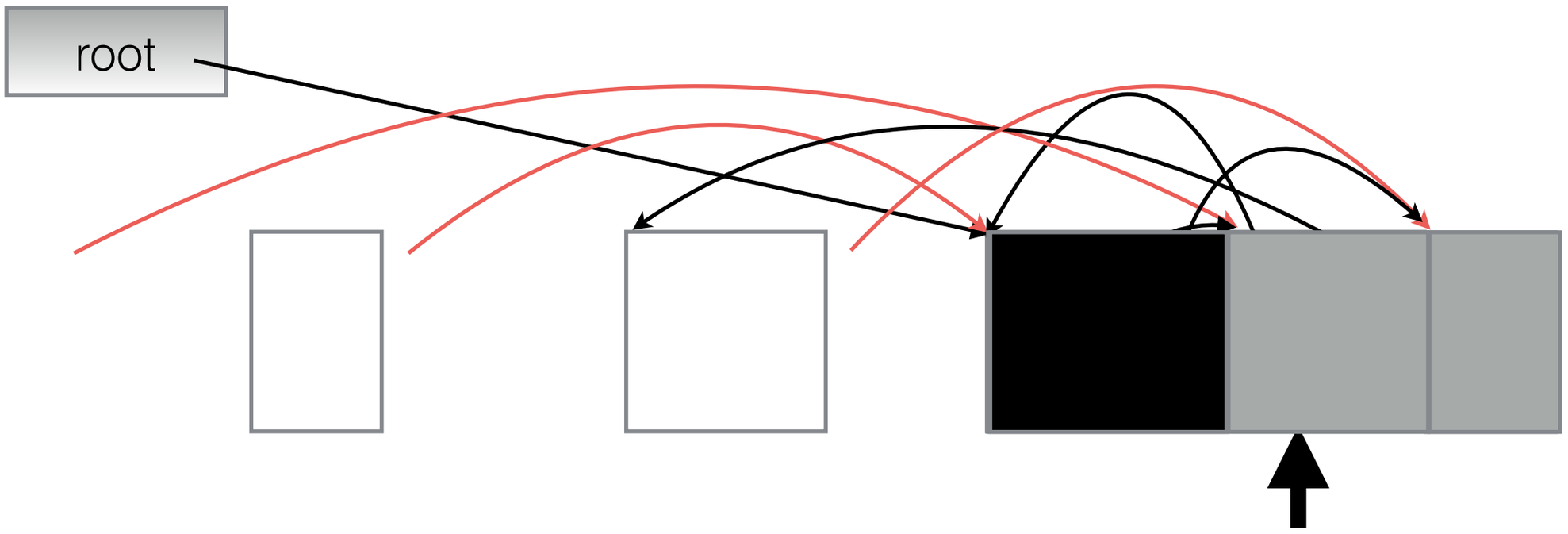
Scan



From

To

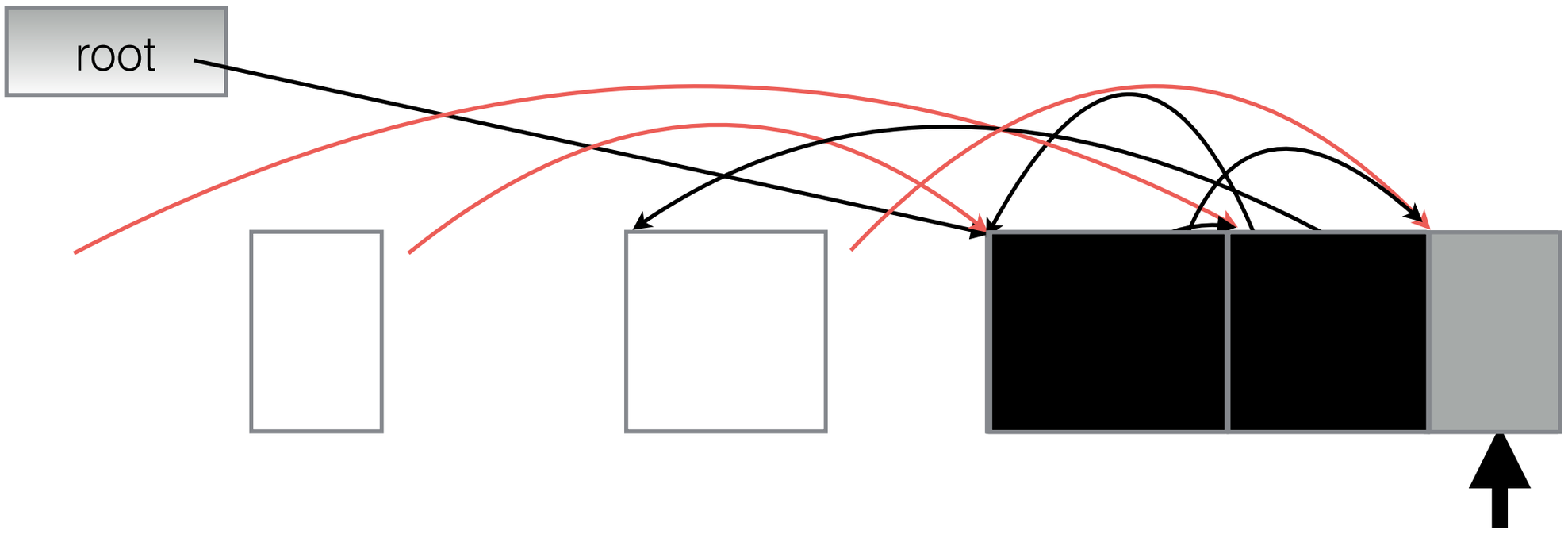
Scan



From

To

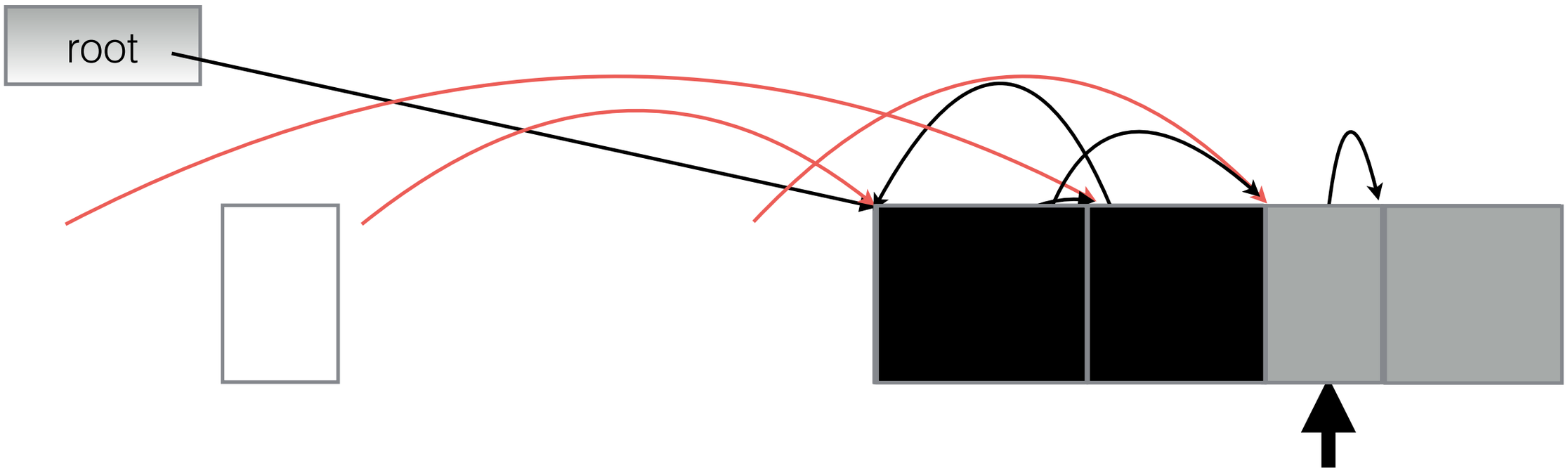
Scan



From

To

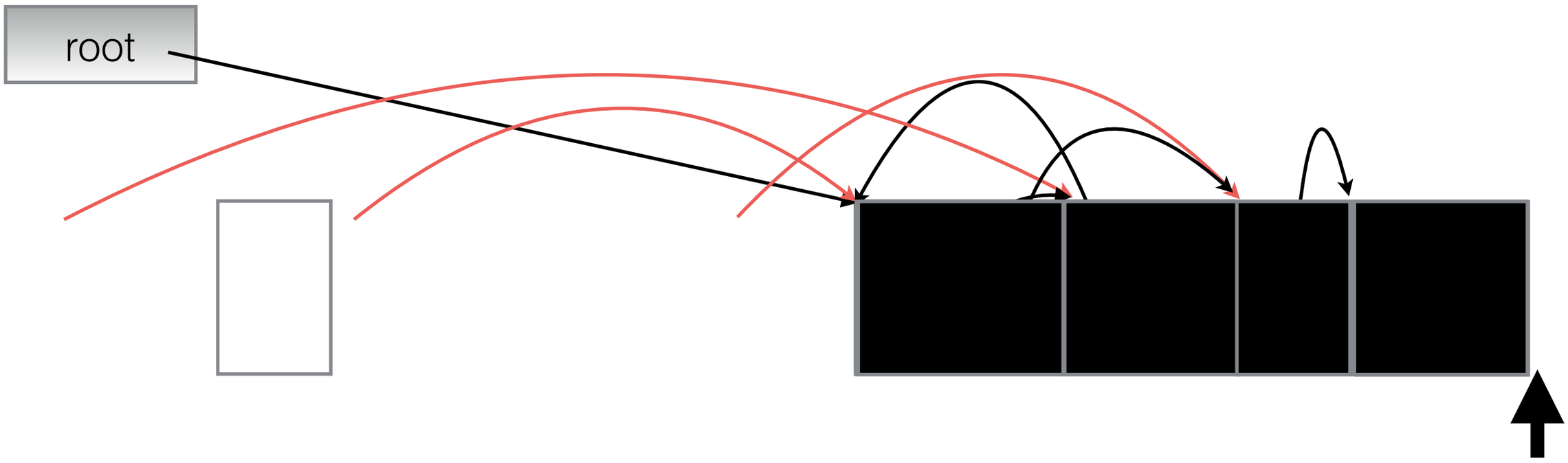
Scan



From

To

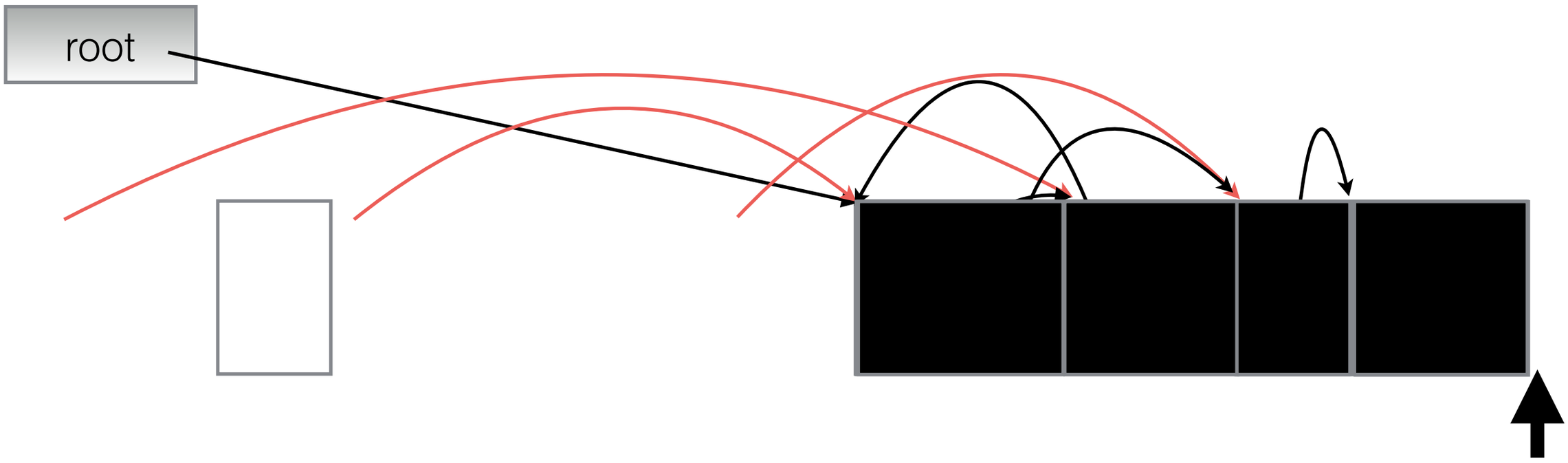
Scan



From

To

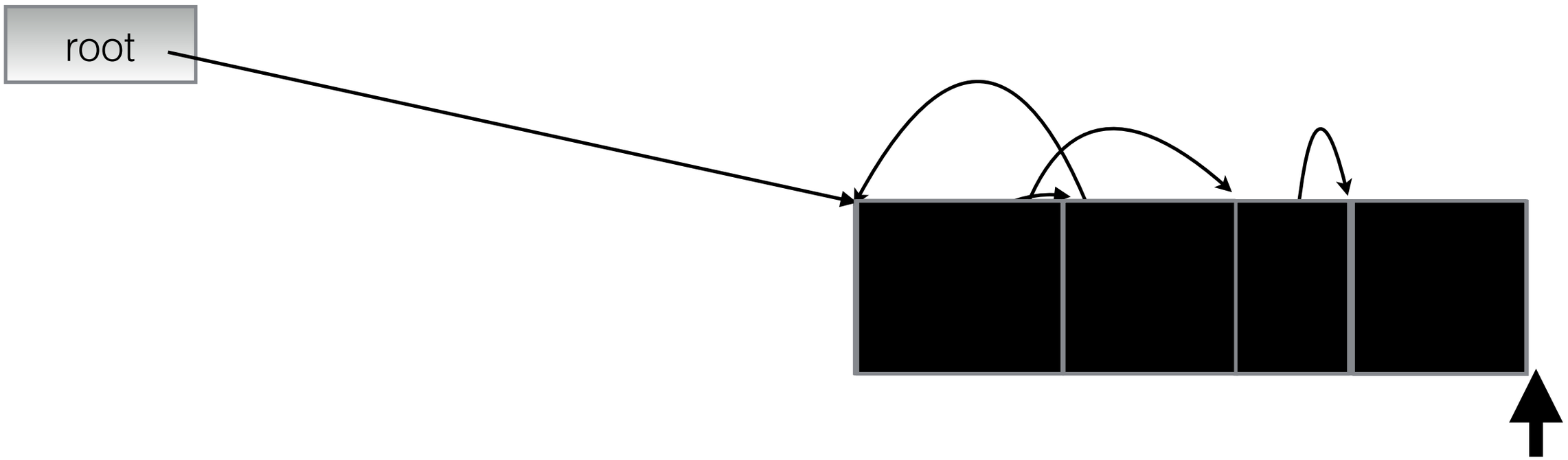
Flip



From

To

Flip



To

From

Incremental collection

- One of the problems of mark-sweep, mark-compact, and simple copying collection is that each induces a long *stop-the-world* pause during which the whole heap is processed and the *mutator* is suspended.

- The mutator is the thread (or threads) modifying the reference graph — i.e., the application. Pausing the mutator is also known as *stopping the world*.

- One solution (among many) is to perform copying *incrementally*.

Baker's algorithm

- In Baker's algorithm, objects referenced from roots are copied to to-space in a bounded stop-the-world pause. Then the mutator restarts, and scanning can proceed concurrently with mutation.

- If the mutator tries to read a reference to a white object (from a gray object), it is paused while the referenced object is copied and the reference forwarded. Thus it never sees references to white objects.

- For synchronization, we consider everything to the left of the scan pointer as black (i.e., the object under the scan pointer is partially black, partially gray).

GC read and write barriers

- The mutator code to read a reference field might look like this:

```
Obj readRef(Obj p, int offset) {
  Obj *refLoc= &p[offset];
  Obj ref= *refLoc;
  if (ref != NULL && scan <= refLoc)
    ref= forward(ref); // copy if needed and forward
  return ref;
}
```

- This is called a *read barrier*; some collectors only intercept writes (a *write barrier*). All other things being equal, why is a write barrier preferable?

Are we done yet?

- The main drawback of the Baker (and other straightforward) copying collectors is that they require twice the memory, and we end up copying all the long-lived objects every time.

- The key insight to address this is the observation that **most objects die young**.

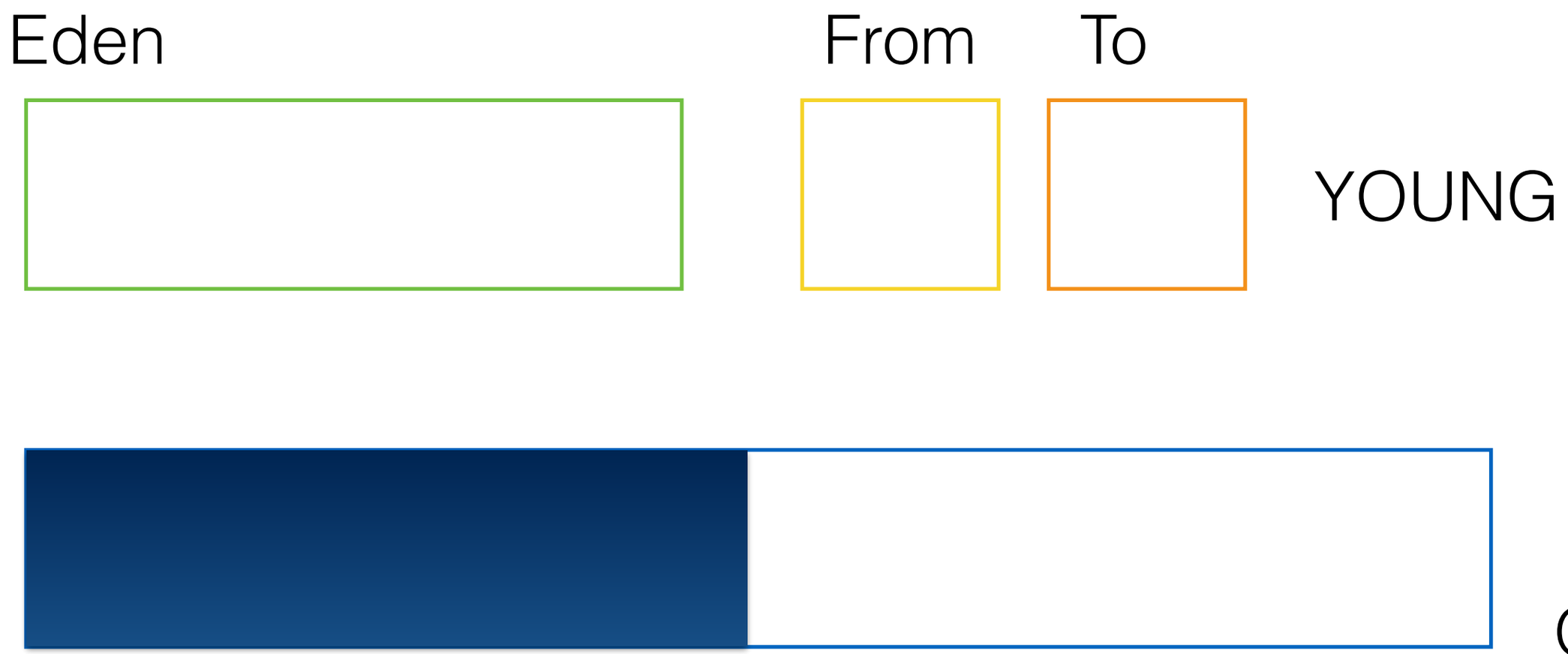
Generational collection

- In a generational system, we segregate the heap into two areas: *young* and *old*. Objects are allocated in the young generation. We do most of our reclamation in the young generation.

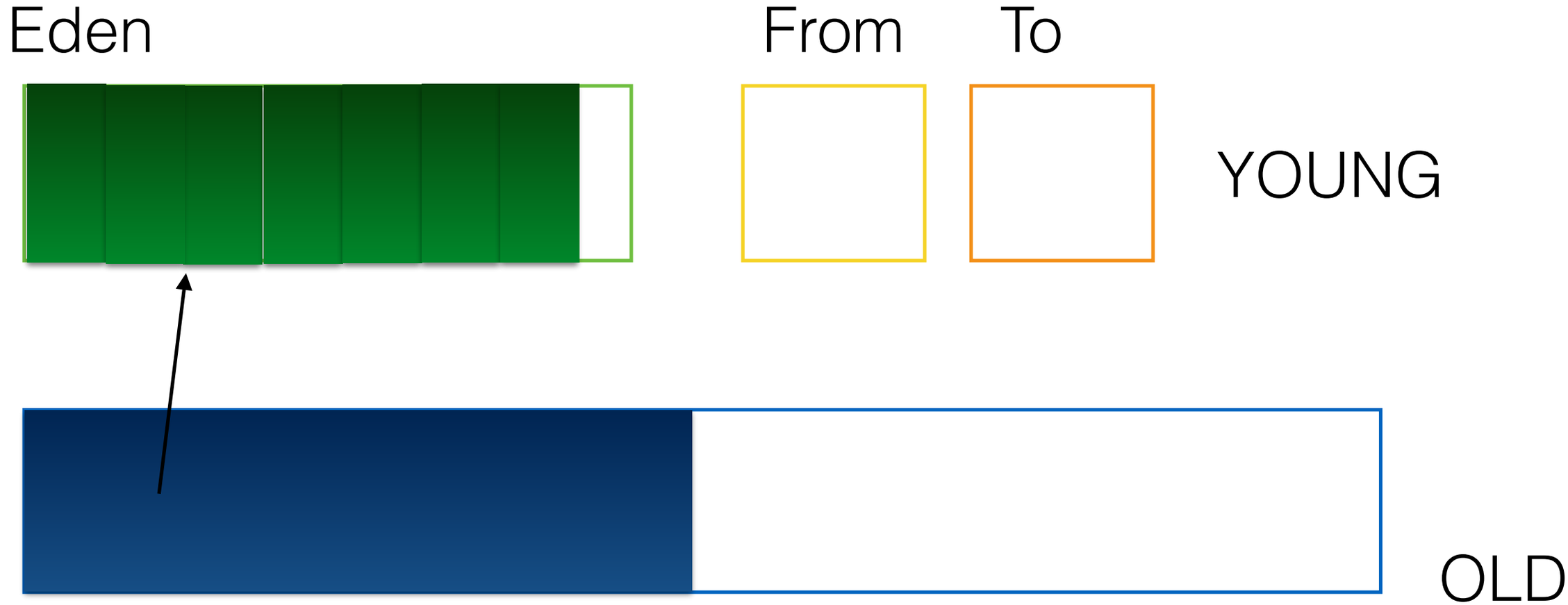
- If a young object lives long enough, it is moved into the old generation — a process called *tenuring*.

- We need to track references from old to young, to avoid scanning the old generation when collecting the young; to do this we need a write barrier.

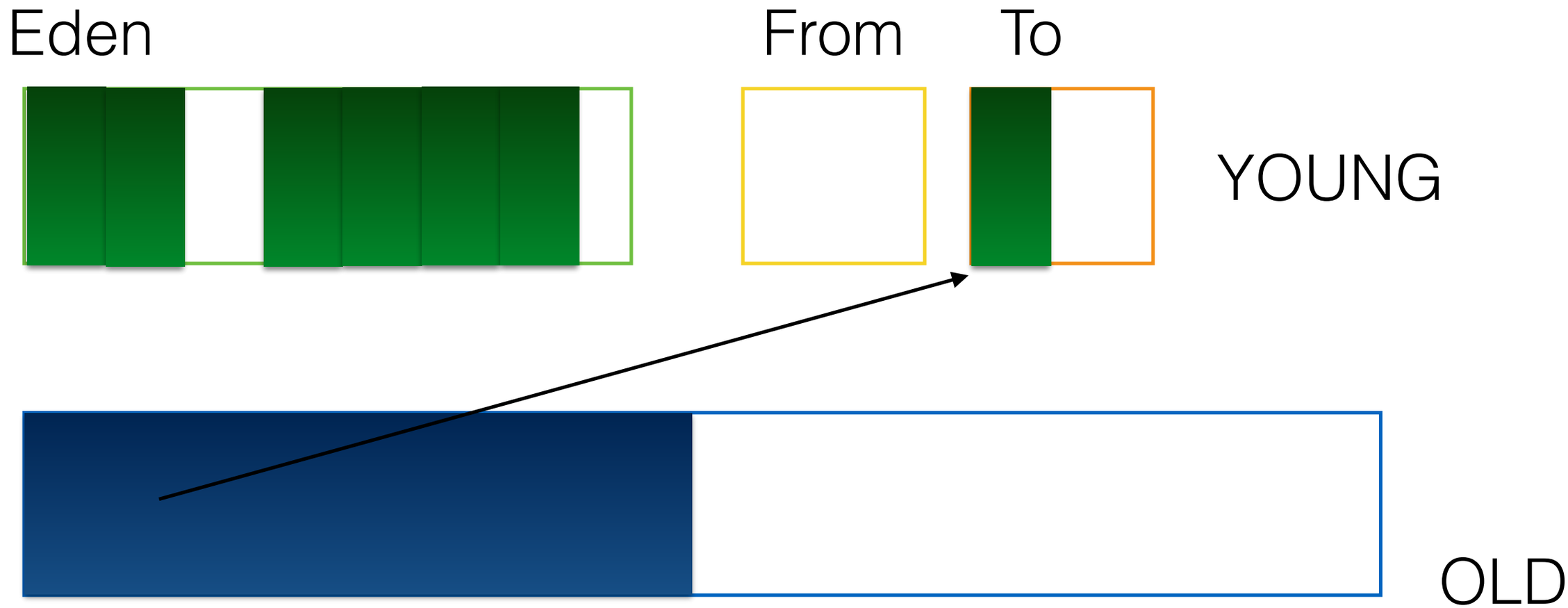
Generation scavenging



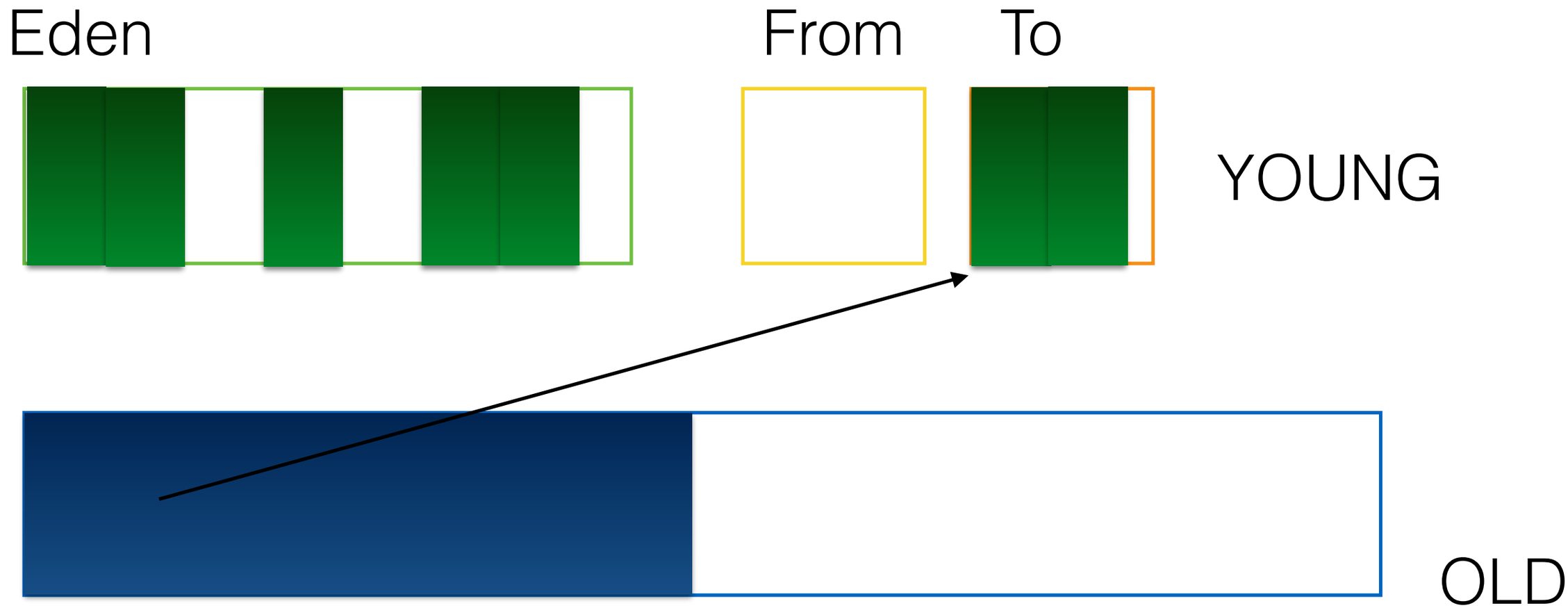
Generation scavenging



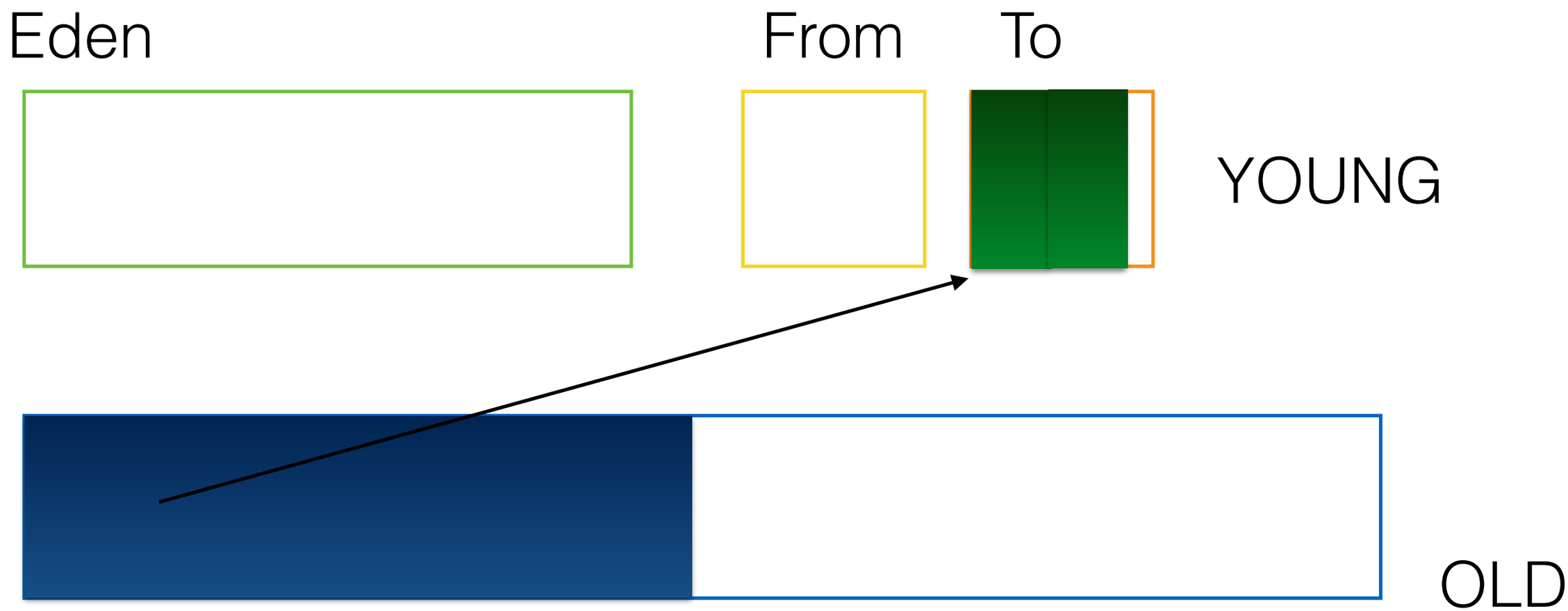
Generation scavenging



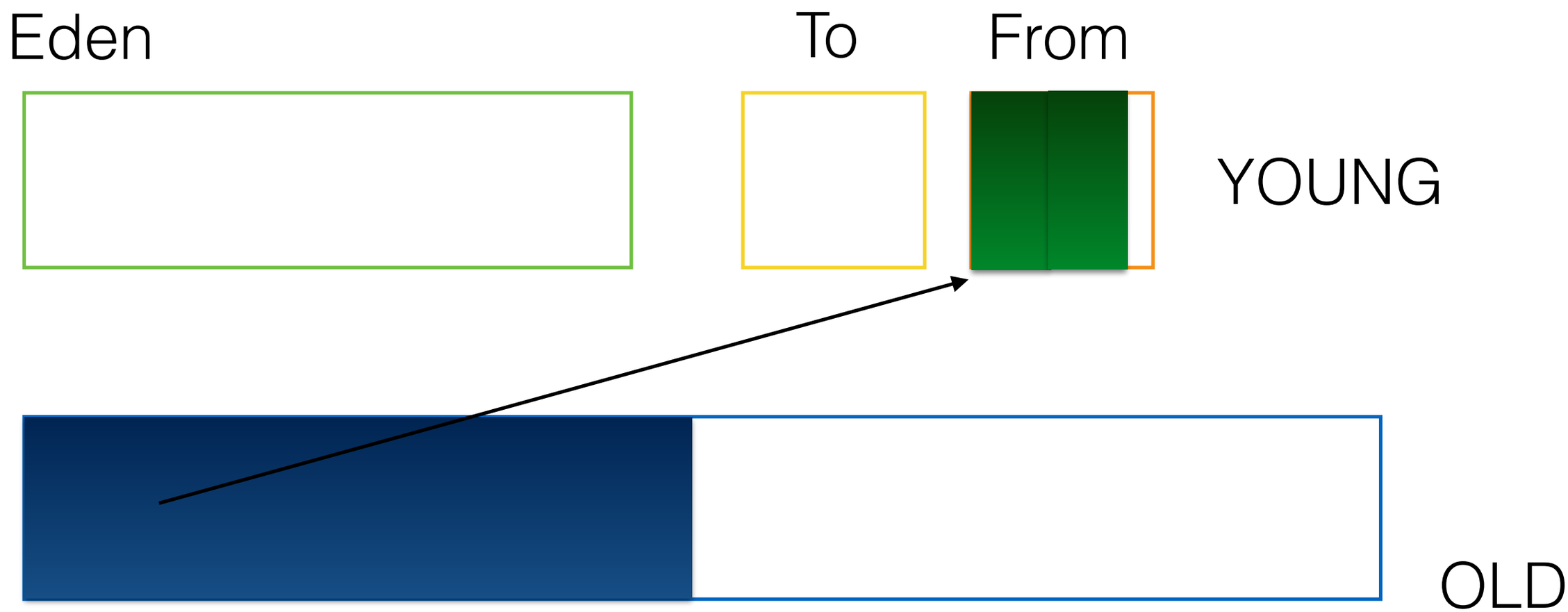
Generation scavenging



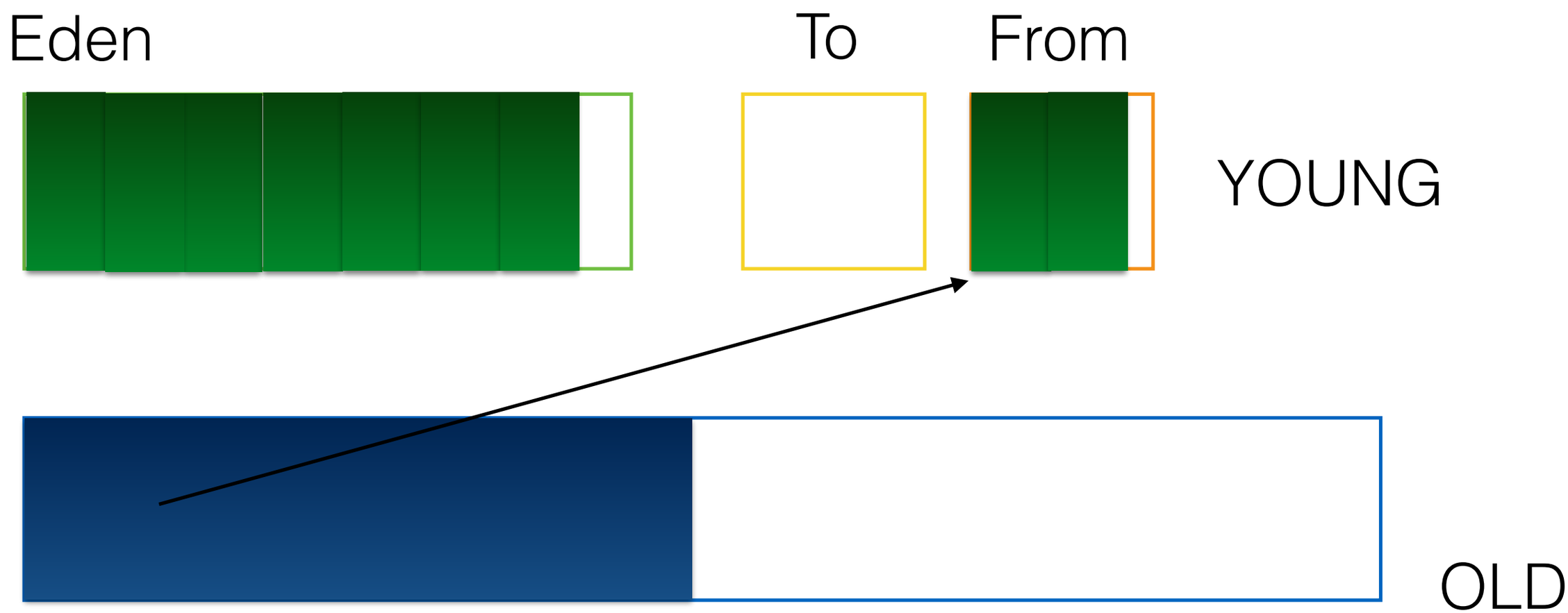
Generation scavenging



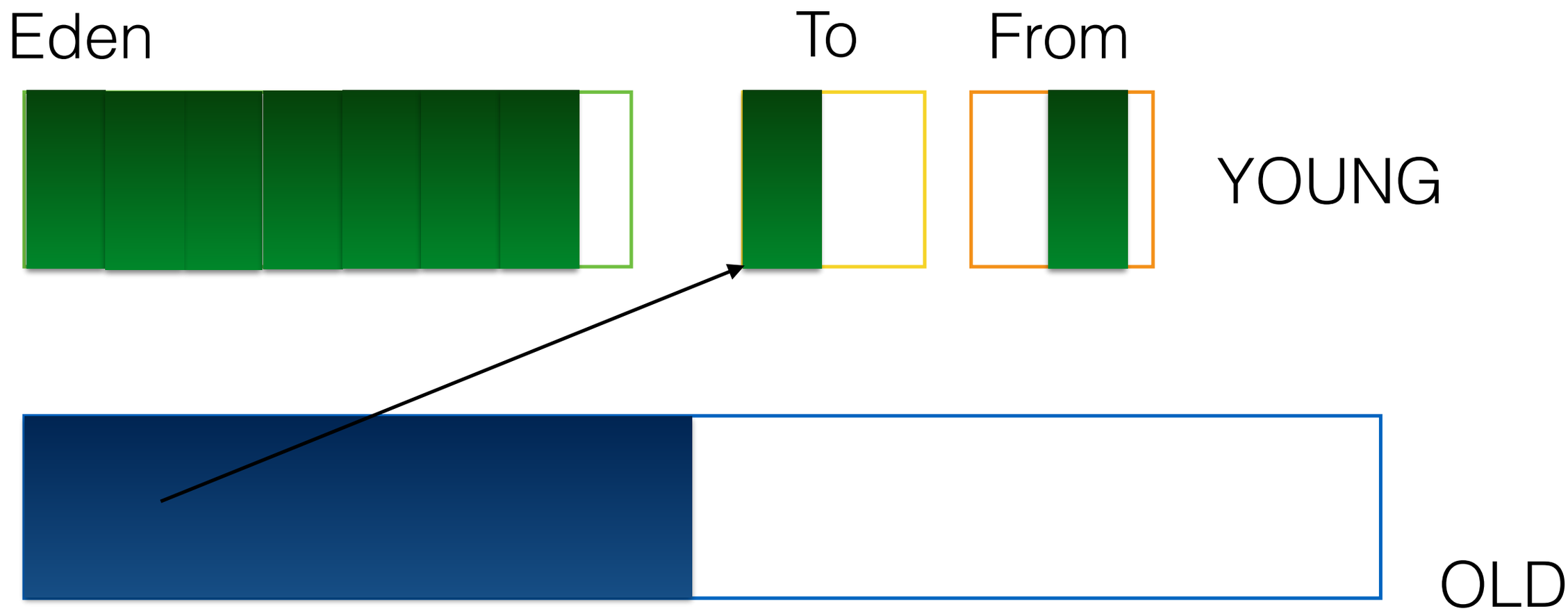
Generation scavenging



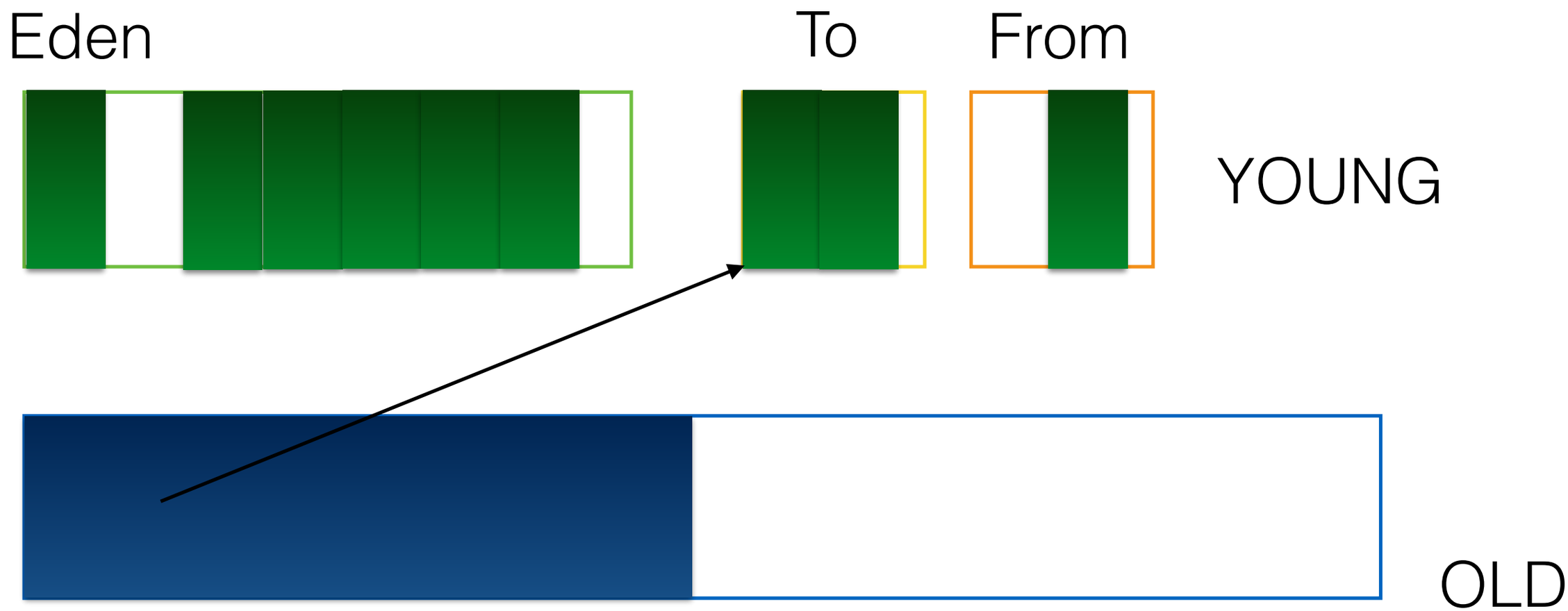
Generation scavenging



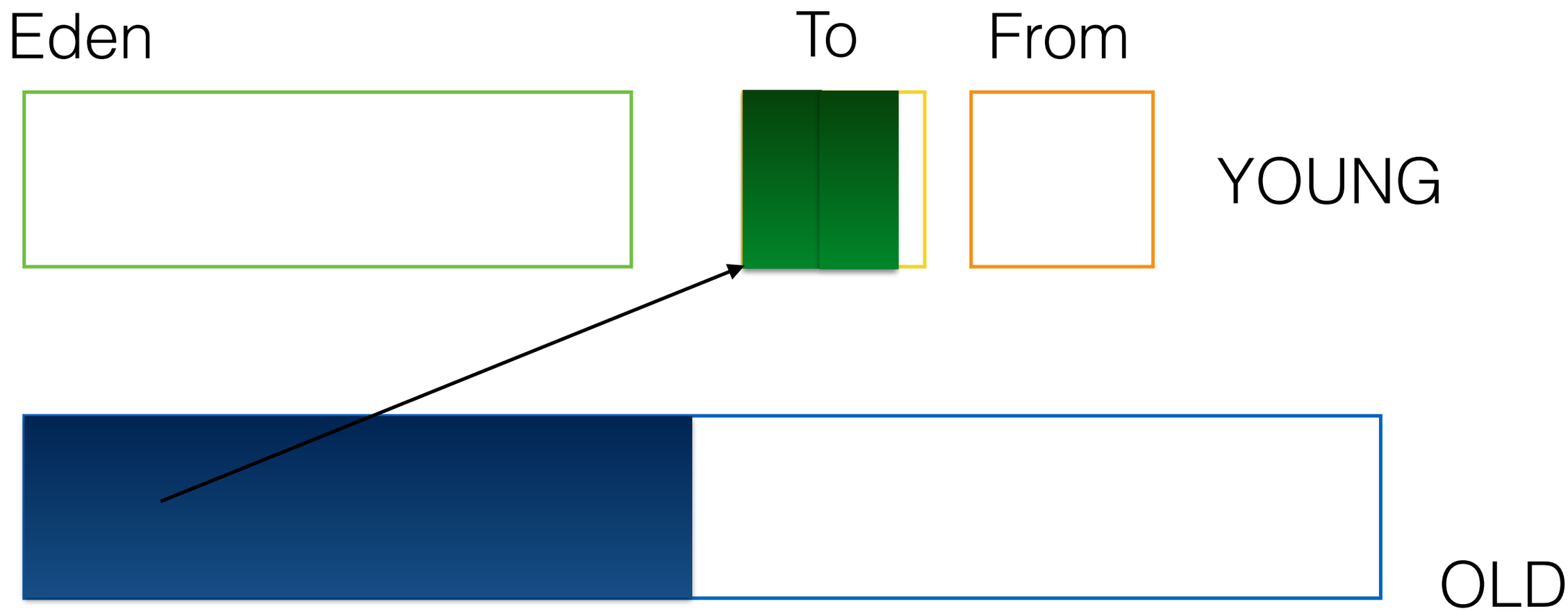
Generation scavenging



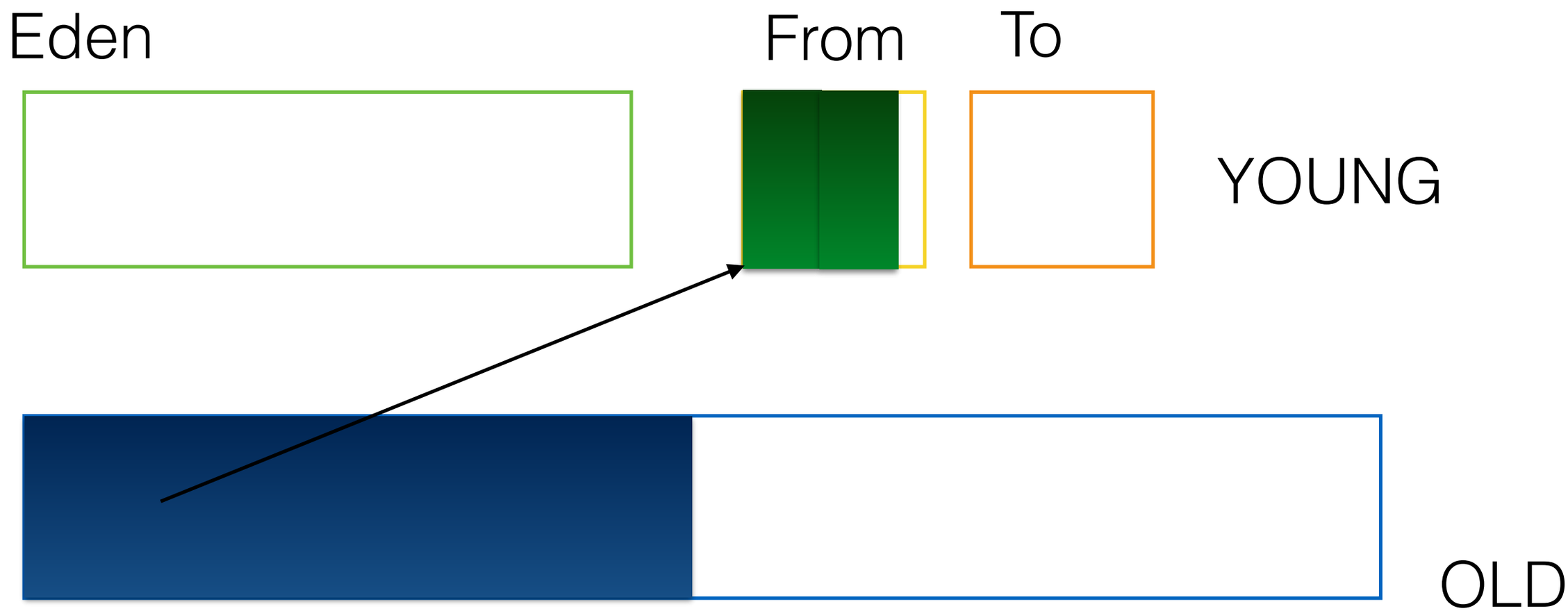
Generation scavenging



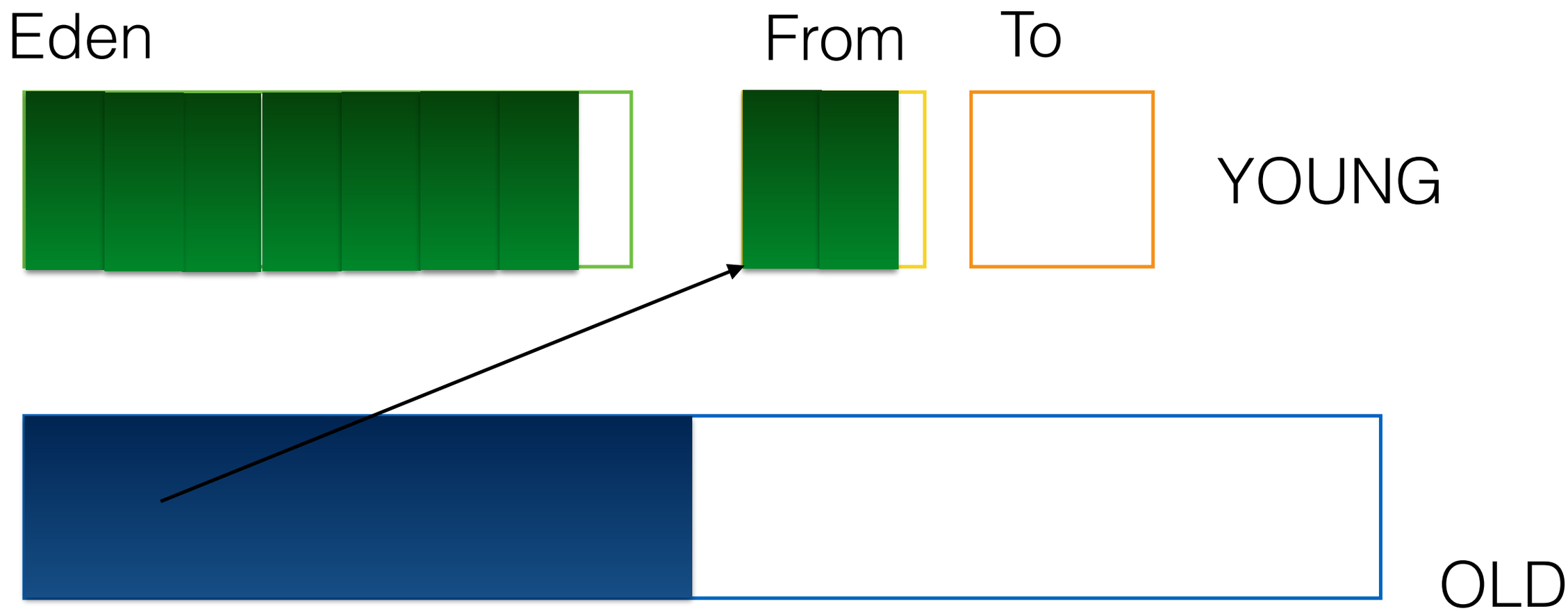
Generation scavenging



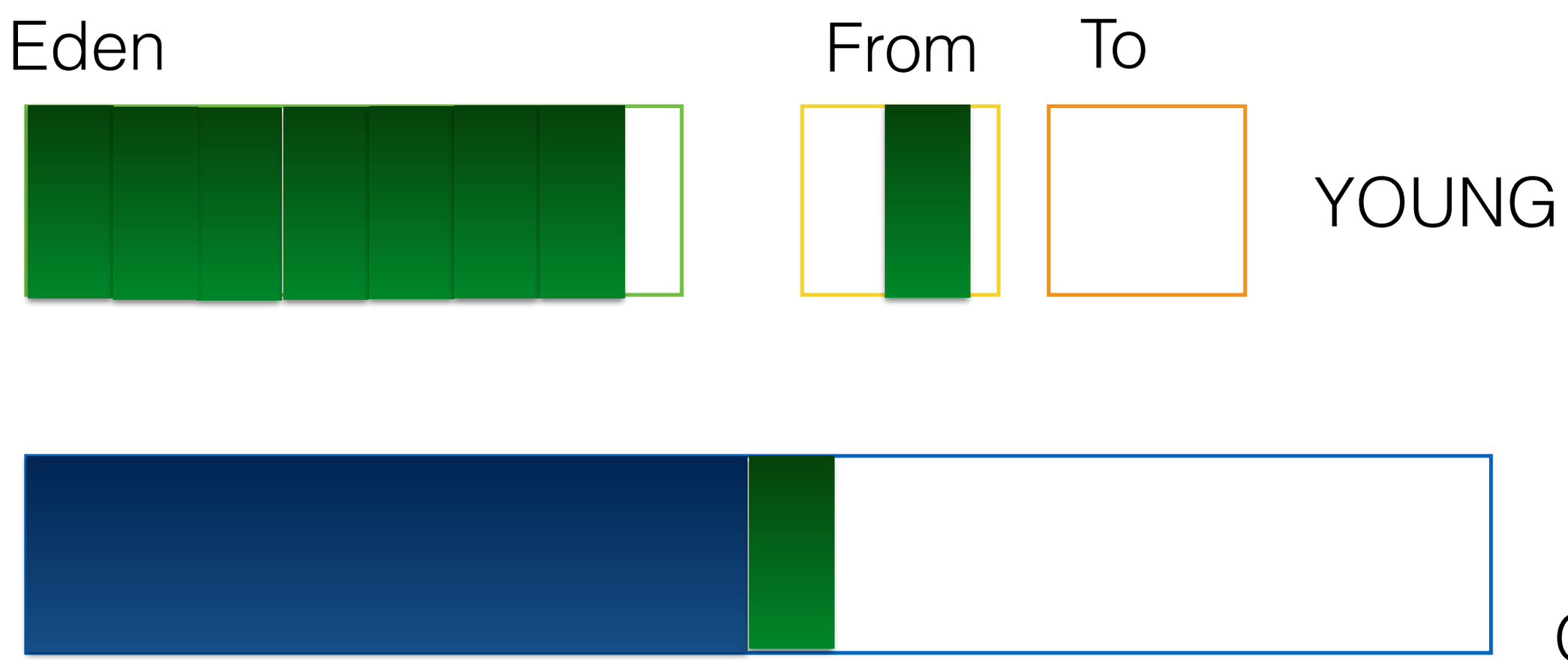
Generation scavenging



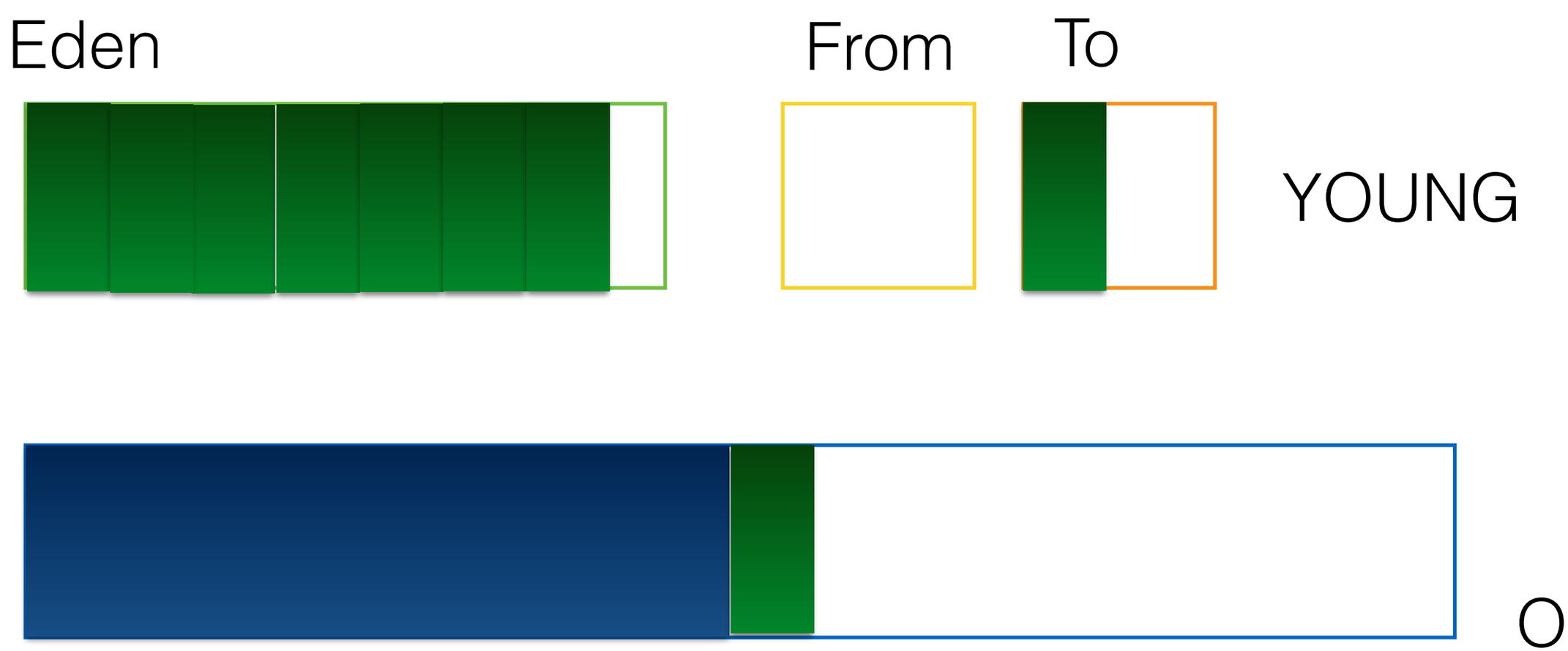
Generation scavenging



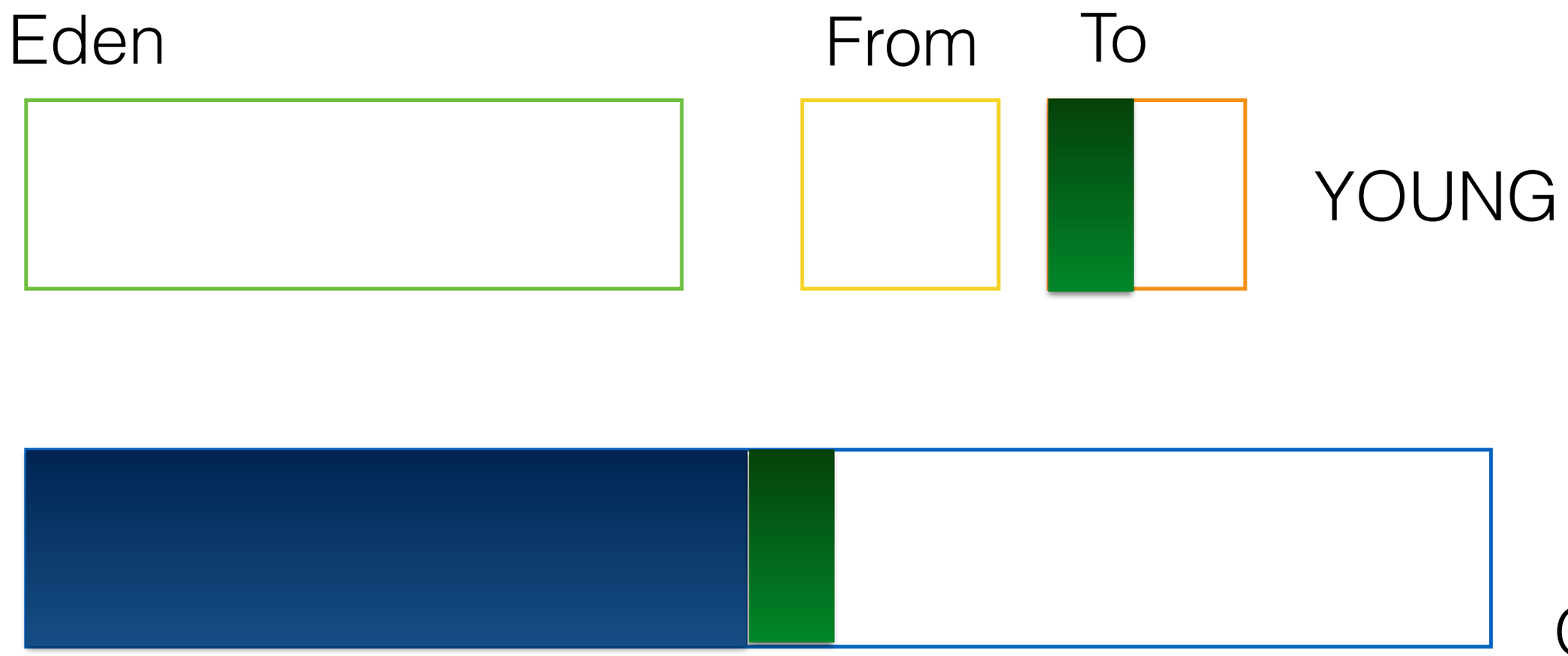
Generation scavenging



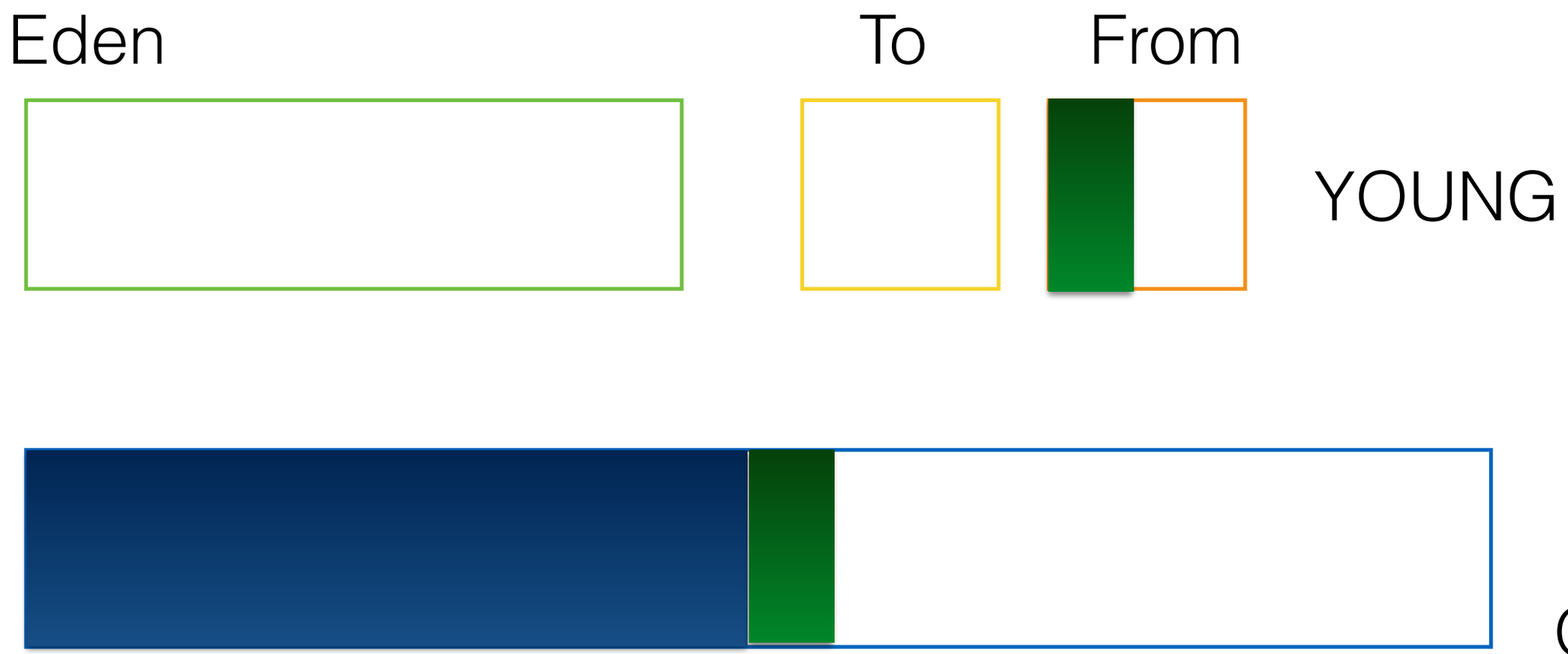
Generation scavenging



Generation scavenging



Generation scavenging



The remembered set

- At the start of a scavenge we must locate all old-to-young pointers without incurring a full old-space scan. This is the job of the *remembered set*, maintained by the write barrier.

- In the original paper, whenever a pointer to new-space is stored in an old-space object, the address of the old-space object is added to the remembered set.

- Q. What about references in old space that are overwritten?

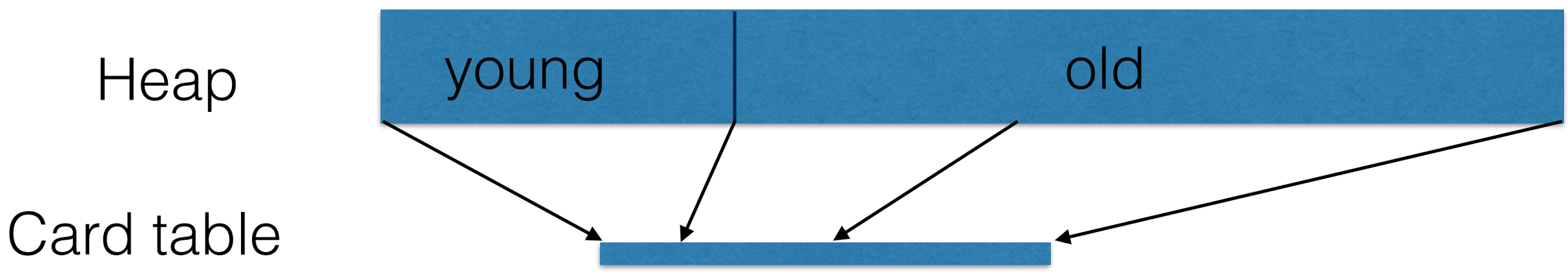
Card marking

- The GS write barrier has two tests. We could remove the new-space test if we simply record all writes to old-space, and filter during the scavenge.

- A more efficient way to do this is divide the heap into fixed-size blocks, and record every write by marking an associated *card*.

- A card is typically a single byte; and the associated blocks are typically 512 bytes long.

Card marking



By having zero represent a mark, a single clear-byte instruction can be used. If the base of the card table is in a register, with a suitable bias the card marking can be done in two instructions:

```
shift-right address-of-field, 9, temp
clear-byte card-base[temp]
```

Generation scavenging - wrap-up

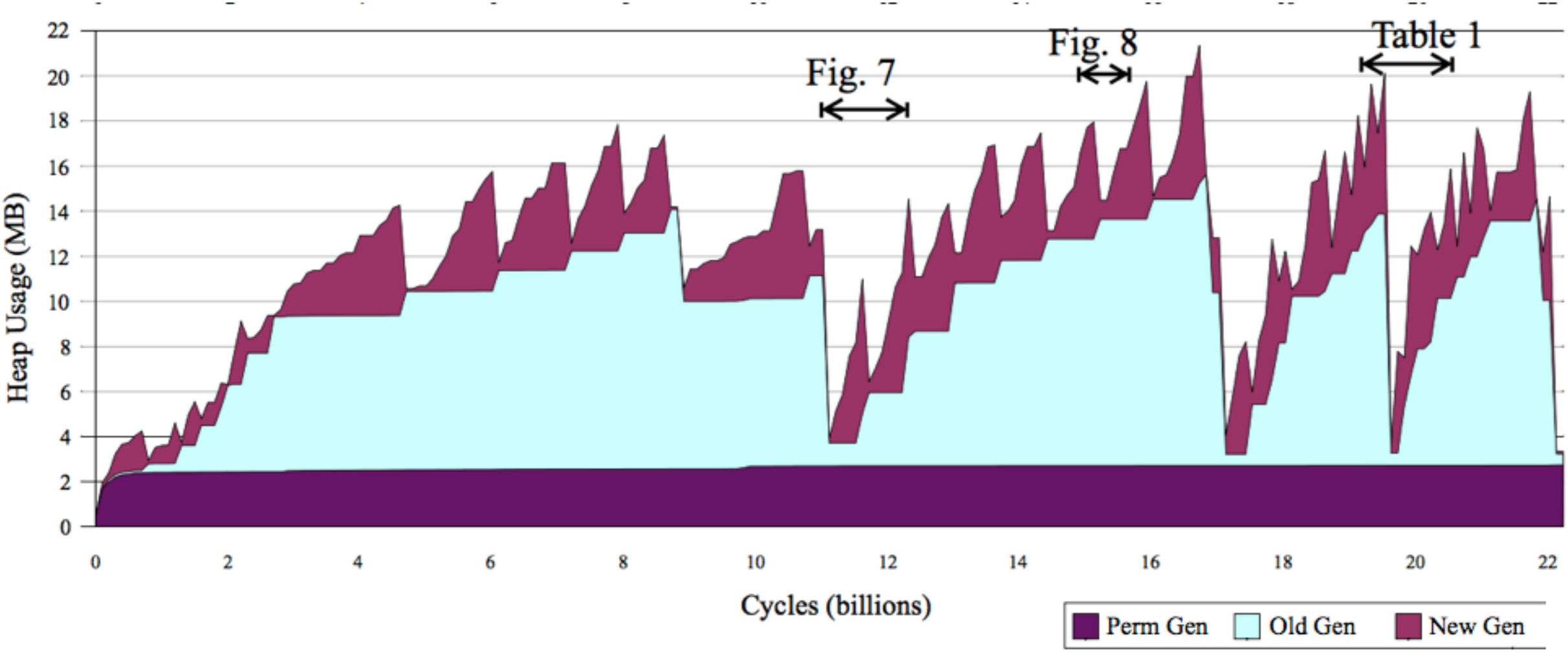
- Most garbage is reclaimed during scavenges, which have short pauses. Pause lengths are determined by the fraction of objects in new space which are live.

- Old space collections are much less frequent. There are many possible choices of old-space collector.

- Bump allocation is used for small objects in new space.

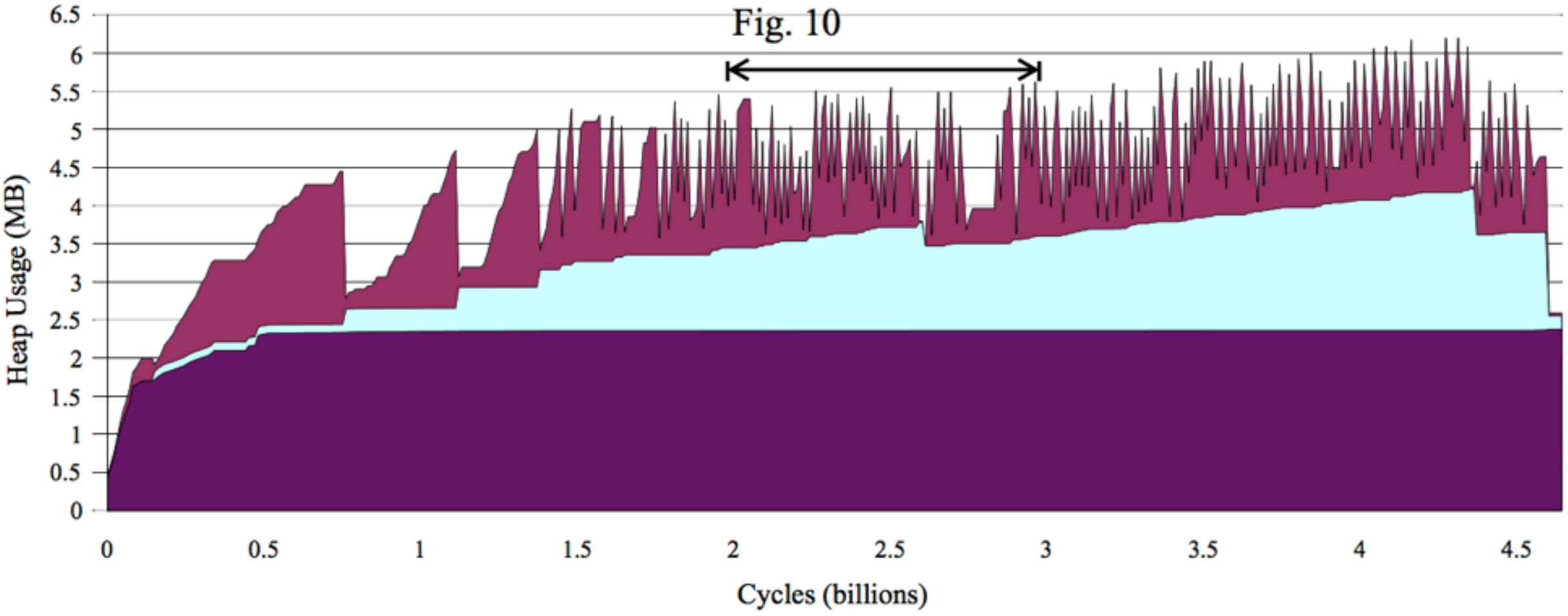
- The card-marking write barrier is cheap.

Example allocation behaviors



- From *Introspection of a Java Virtual Machine under Simulation*, Wright et al, Sun Labs TR-2006-159

Fig. 10


Heap Usage (MB)
6.5
6
5.5
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
Cycles (billions)

Generation scavenging: Advanced topics

- Aging and tenuring decisions
  - See, e.g., Ungar & Jackson 1988
- Pre-tenuring: allocating directly into old space.
- Sizing (and re-sizing) the spaces (e.g., relative to cache, memory)
- Prefetching during bump allocation