Memory management

Advanced Compiler Construction

Michel Schinz – 2021–05–06

Memory management

During execution, programs often use:

- more memory than is physically available, but
- not all of it at the same time.

The goal of **memory management** is to make good use of the available physical memory.

Typically, programs allocate memory from:

- the **stack**, whose management is trivial,
- the **heap**, whose management is more complex.

The memory manager

The **memory manager** is the part of the run time system in charge of managing the heap by (de)allocating **blocks** (or **objects**).

Allocation is usually explicit, but deallocation can be:

- **explicit** if the programmer asks for a block to be freed,
- **implicit** if the memory manager automatically tries to free unused blocks, e.g. when running out of memory.

Explicit deallocation

Explicit memory deallocation presents several problems:

1. memory can be freed too *early* (**dangling pointers**),
2. memory can be freed too *late* (**space leaks**).

Therefore, most modern programming languages provide **implicit deallocation**, a.k.a. **automatic memory management**.
(Note: often also called **garbage collection**, but this term designates a specific kind of automatic memory management).

Implicit deallocation

Assumption of implicit memory deallocation:

Only unreachable blocks can be freed, as reachable ones could be accessed in the future.

This assumption is:

- conservative, as reachable blocks might never be accessed anymore, but
- safe, as no pointer will ever point to deallocated memory.

Therefore, implicit memory deallocation:

- does *not* avoid space leaks, but
- completely avoids dangling pointers.

Garbage collection

Garbage collection (GC) is a common name for techniques that automatically reclaim unreachable objects.

We'll look at:

1. reference counting,
2. mark & sweep,
3. copying, and
4. generational garbage collection.

Concepts common to all of them are introduced first.

Reachable objects

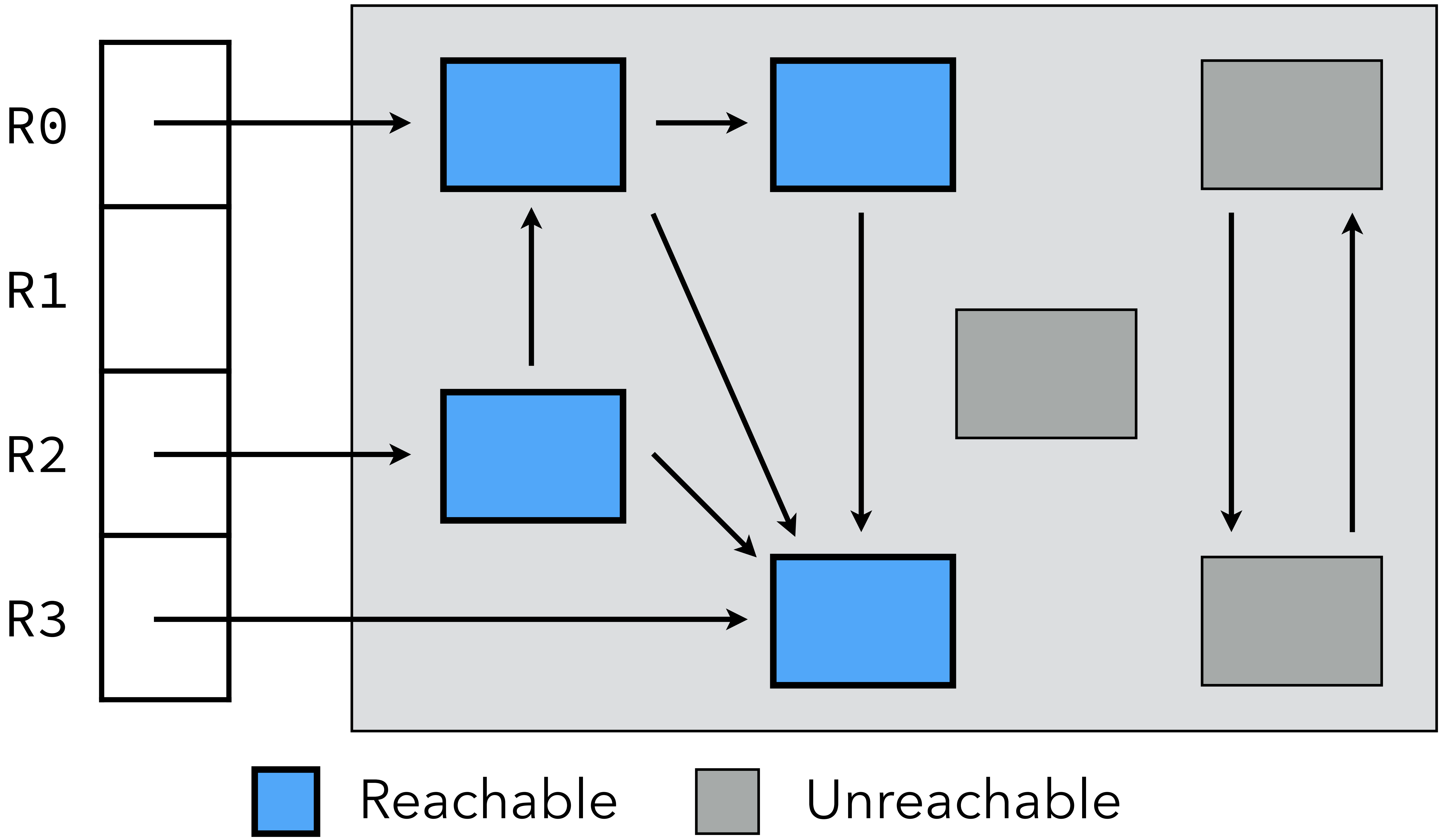
Reachable objects

The **reachable objects** are:

- those immediately accessible from global variables, the stack or registers : the **roots** or **root set**,
- those reachable from other reachable objects, by following pointers.

They form the **reachability graph**.

Reachability graph example



(Im)precision

To compute the reachability graph, all pointers must be identifiable unambiguously at run time!

If that is not possible, the graph can be approximated conservatively:

- it is *safe* (but sub-optimal) to consider unreachable objects as reachable,
- it is *unsafe* to consider reachable objects as unreachable.

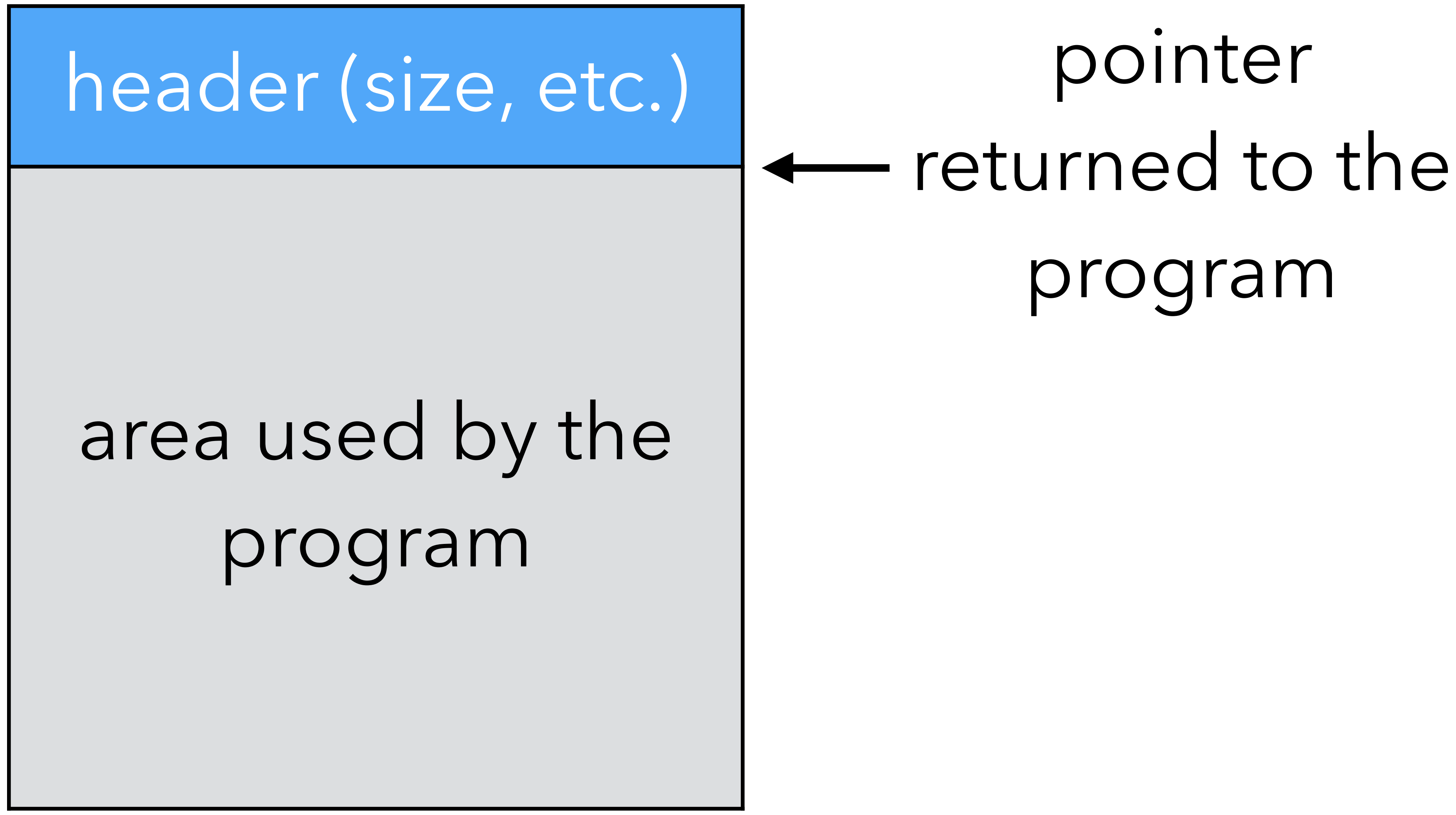
Memory manager data structures

Free list

The memory manager must know which parts of the heap are free and allocated.
Free blocks are stored in a **free list**, which is often a more sophisticated data structure than a simple list.

Block header

The memory manager must know several properties of the blocks it manages, e.g. their size.
They are often stored in a **block header**, just before the area used by the program.

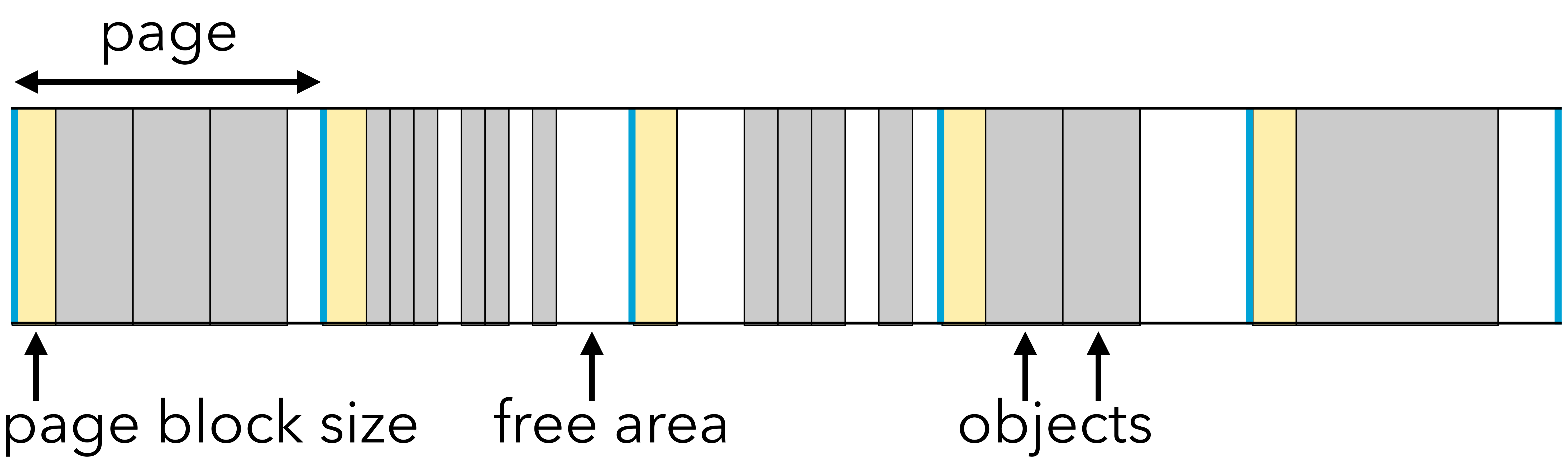


BiBoP

BiBoP (**big bag of pages**) decreases header overhead by splitting memory in pages which:

- all have the same power-of-two size ($s = 2^b$),
- are aligned on multiples of their size,
- only contain objects of identical size o,
- store the objects' size (o) at the beginning.

The size of an object can be retrieved by masking the b least-significant bits of its address.



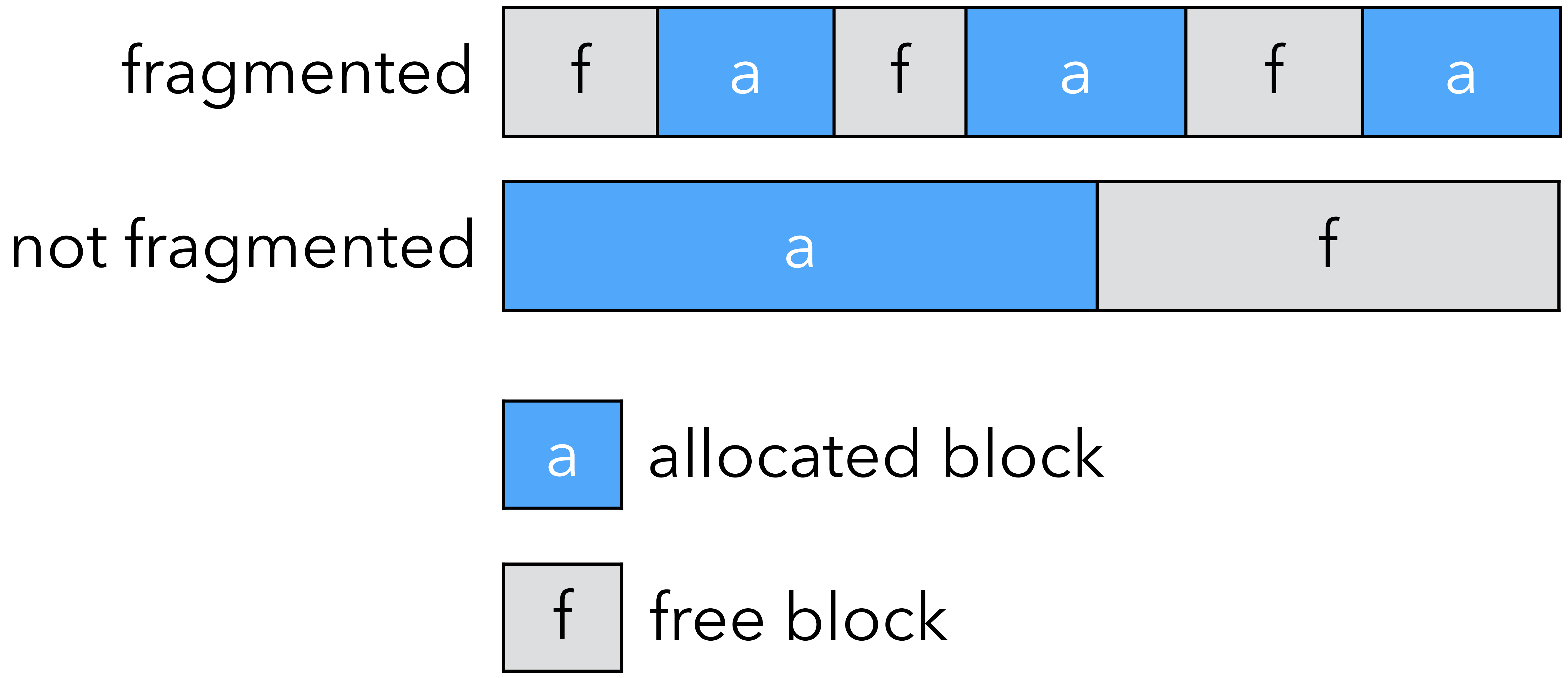
Fragmentation

The term **fragmentation** is used to designate two different but similar problems associated with memory management:

- **external fragmentation** refers to the fragmentation of free memory in many small blocks,
- **internal fragmentation** refers to the waste of memory due to the use of a free block larger than required to satisfy an allocation request.

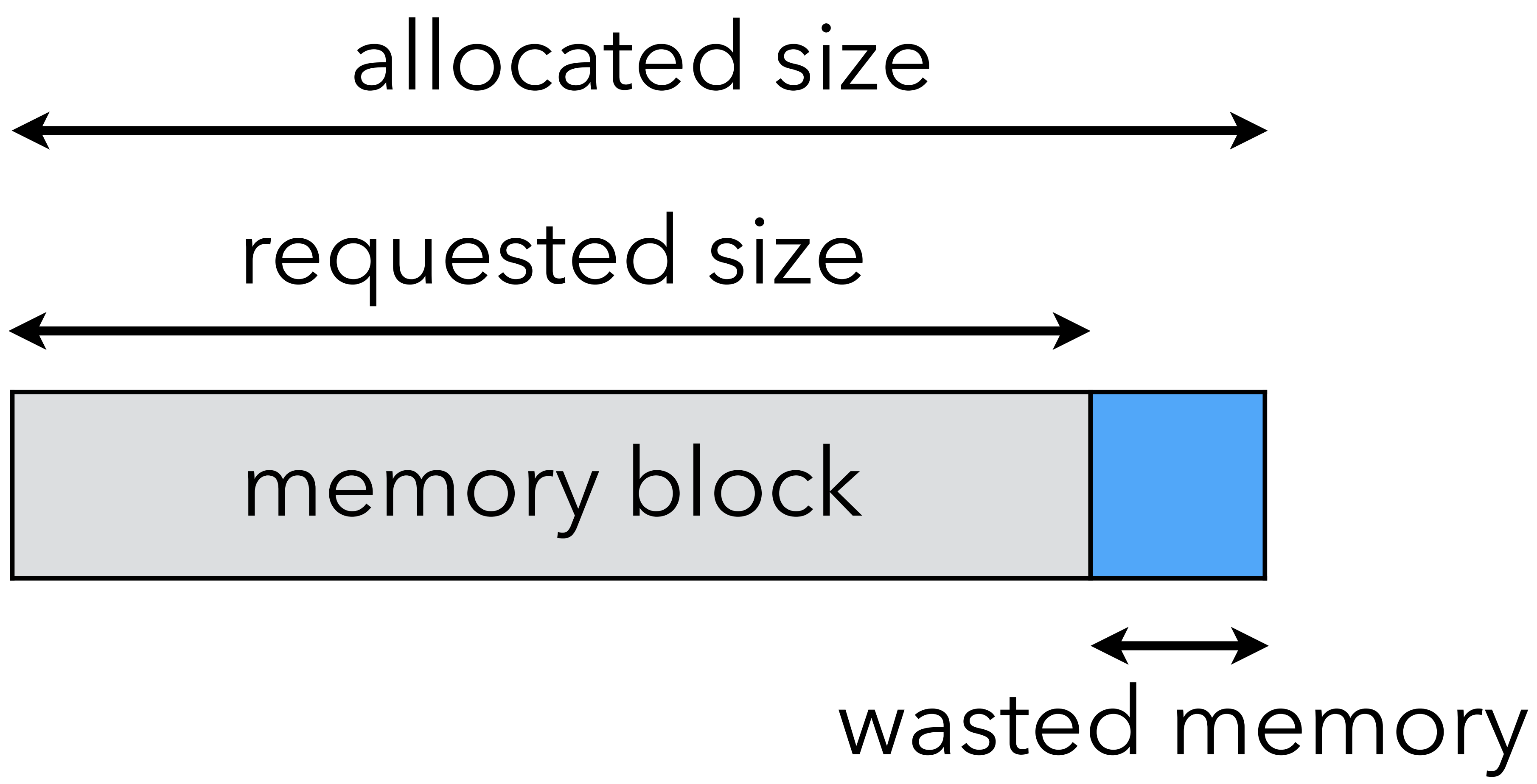
External fragmentation

The two heaps below have the same amount of free memory, but the first suffers from **external fragmentation** while the second does not. Therefore, some requests can be fulfilled by the second but not by the first.



Internal fragmentation

The memory manager sometimes allocates more memory than requested, e.g. to satisfy alignment constraints. This results in small amounts of wasted memory scattered in the heap, and is called **internal fragmentation**.



GC technique #1: reference counting

Reference counting

The idea of **reference counting** (RC) is simple:

- every object carries a count of the number of pointers referencing it,
- when that count reaches 0, the object is unreachable and gets deallocated.

The maintenance of reference counts requires collaboration from the compiler and/or the programmer.

Pros and cons of RC

Pros of reference counting:

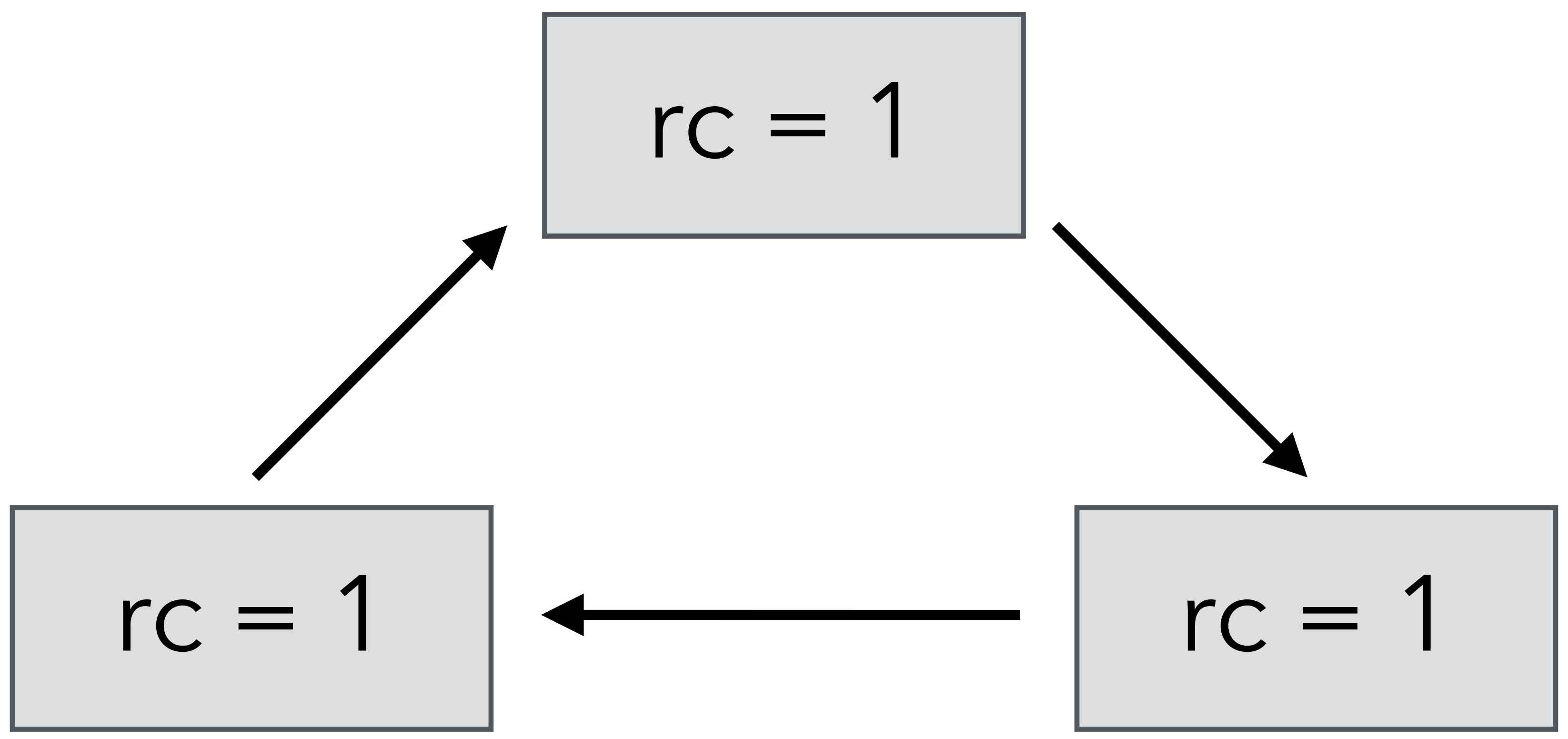
- relatively easy to implement, even as a library,
- memory is reclaimed immediately.

Cons of reference counting:

- the counters take space,
- updating the counters takes time,
- cannot deal with cyclic structures.

Cyclic structures

The reference count of objects that are part of a cycle in the object graph never reaches zero, even when they become unreachable!
This is *the* major problem of reference counting.



Cyclic structures

Problem: reference counts provide only an approximation of reachability. In other words, we have:

$$\text{reference\_count}(x) = 0 \;\Rightarrow\; x \text{ is unreachable}$$

but the opposite is not true!

Uses of reference counting

Due to its problem with cyclic structures, reference counting is only used:

- in systems that disallow the creation of cycles (e.g. hard links on Unix file systems),
- in combination with another GC that periodically collect cyclic structures (e.g. in Python).

GC technique #2: mark & sweep

Mark & sweep GC

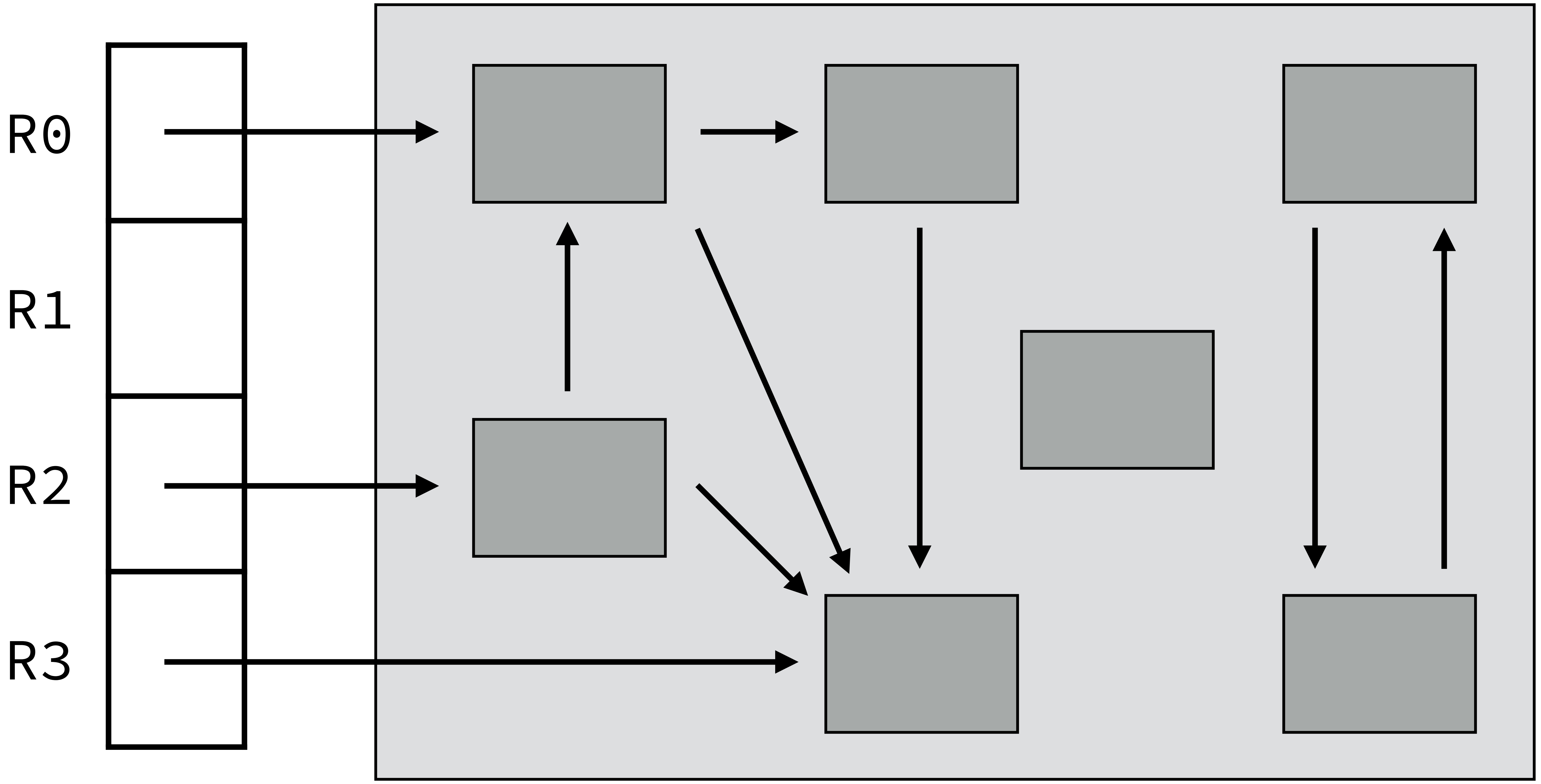
Mark & sweep GC proceeds in two phases:

1. **mark**: reachable objects are marked,
2. **sweep**: unmarked, allocated objects are deallocated.

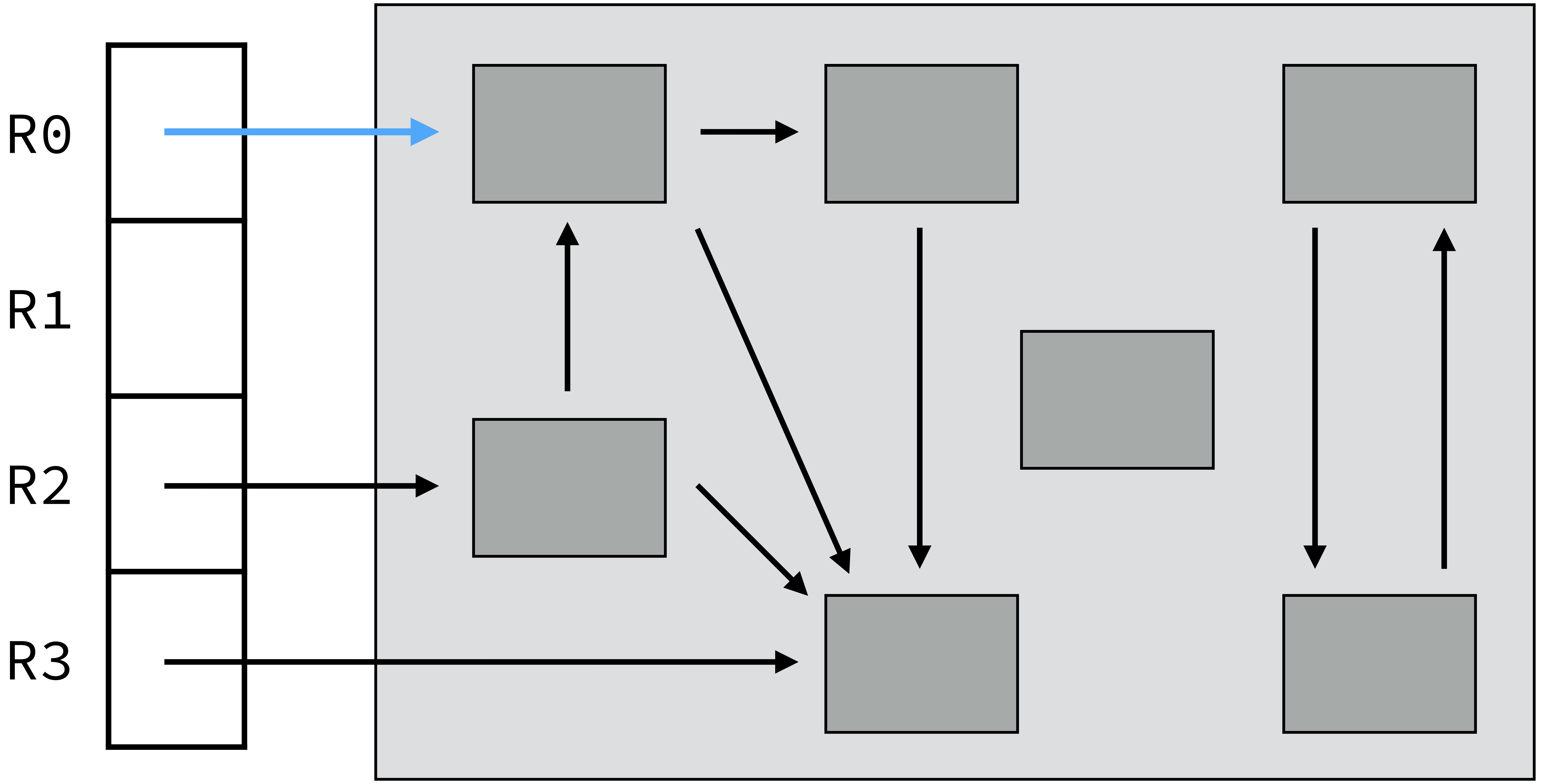
Typically:

- GC is triggered by lack of memory,
- the program is stopped until GC is done, to ensure that the reachability graph is not modified while the GC traverses it.

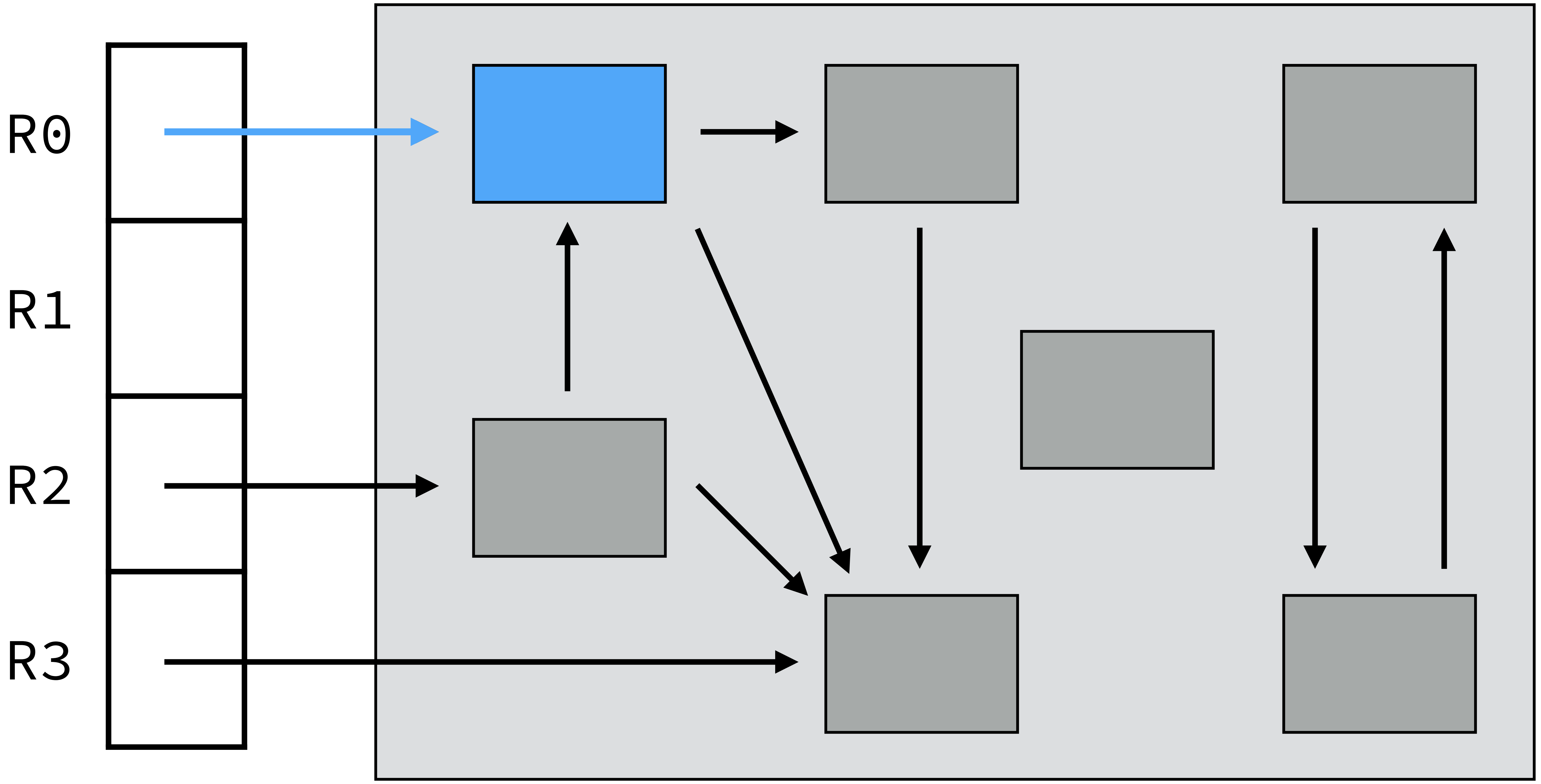
Mark & sweep GC



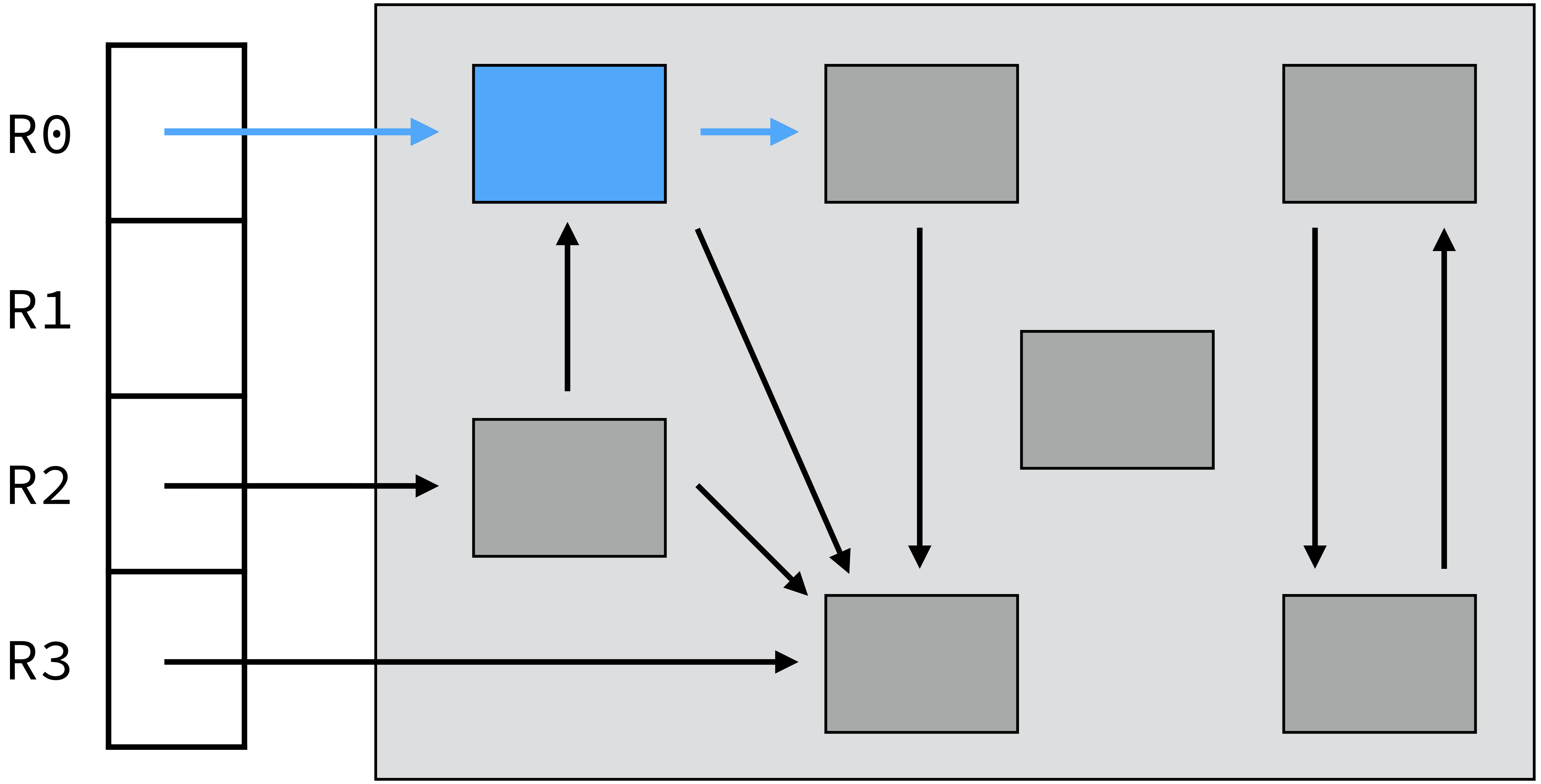
Mark & sweep GC



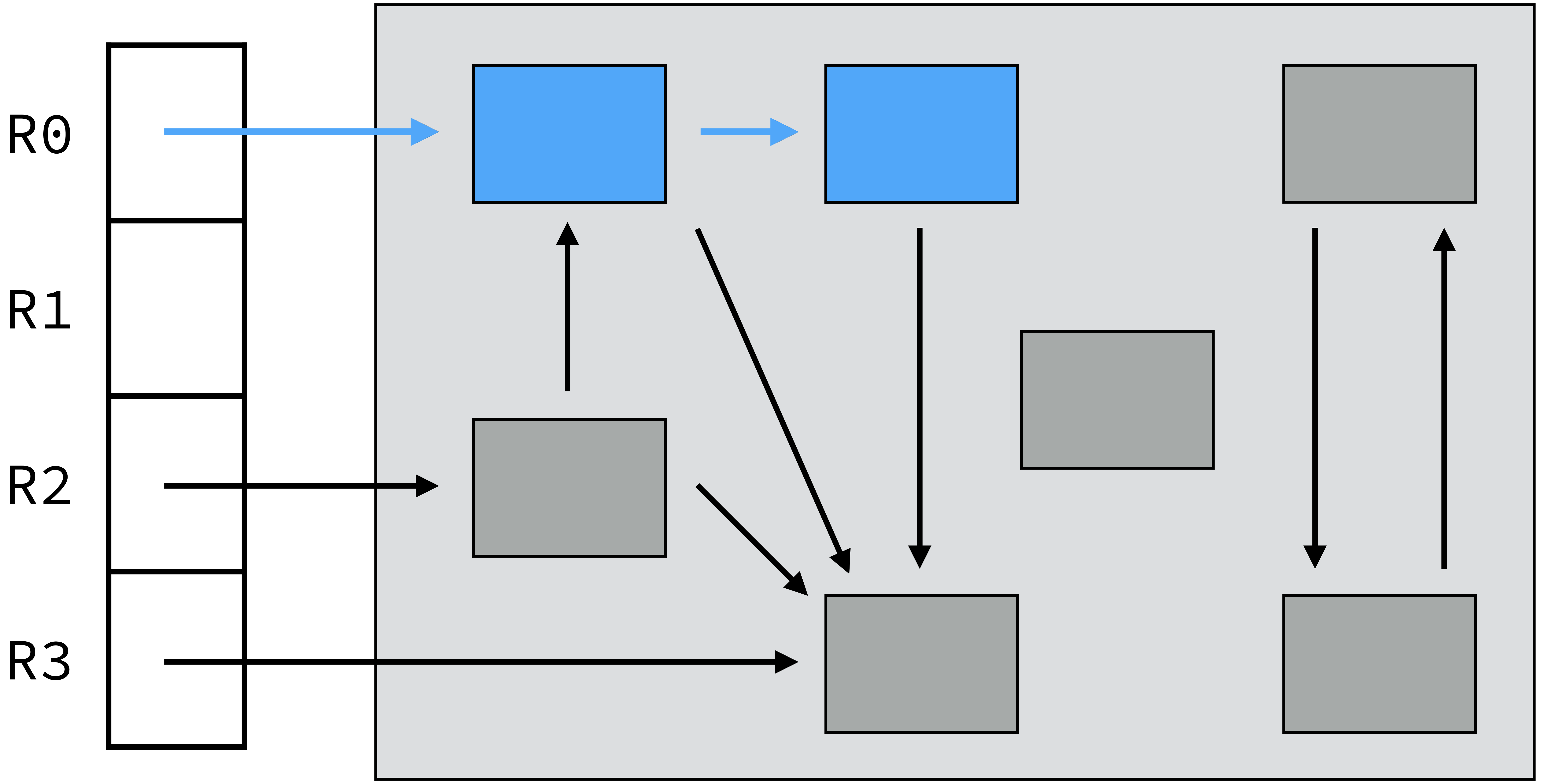
Mark & sweep GC



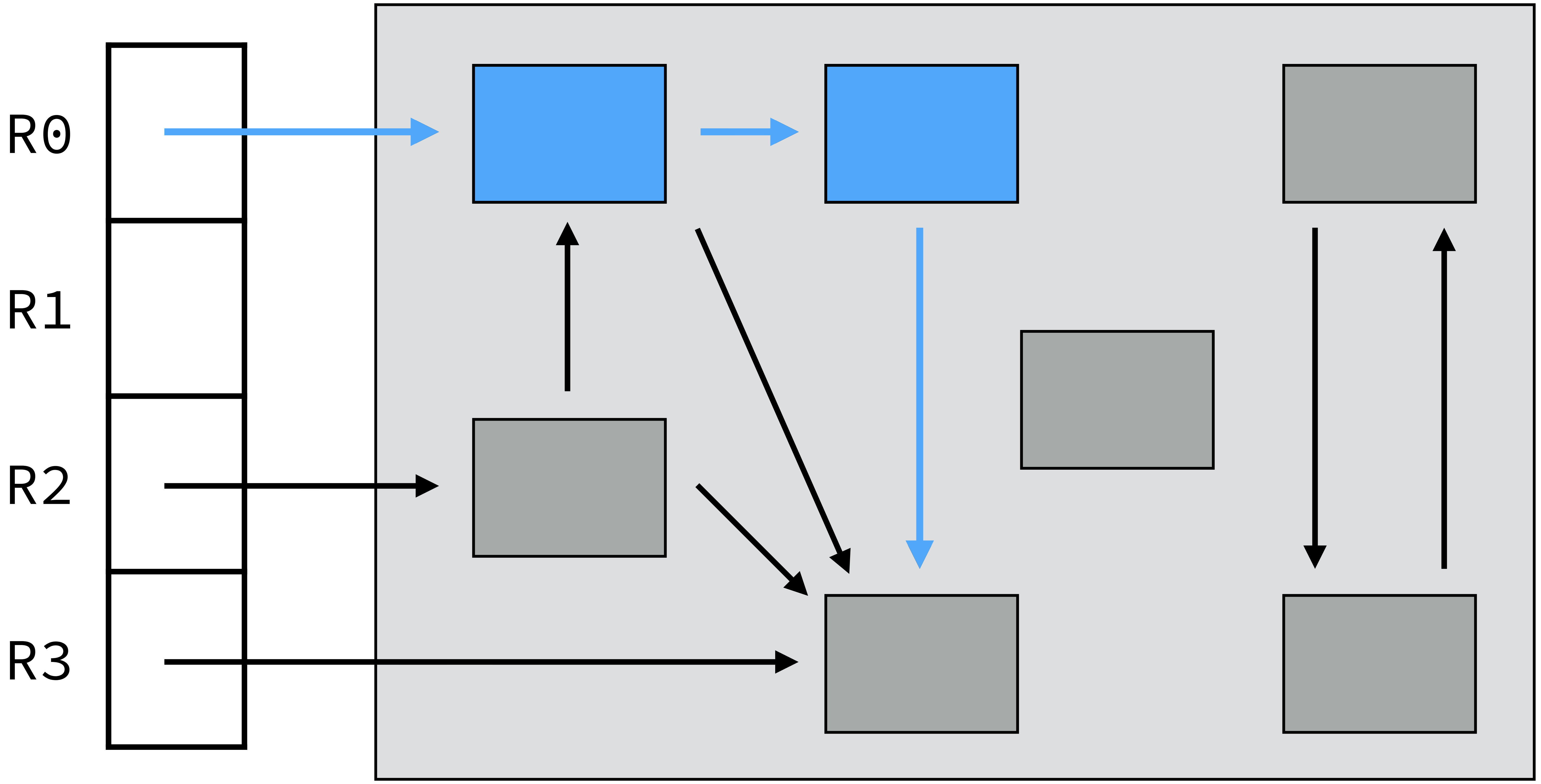
Mark & sweep GC



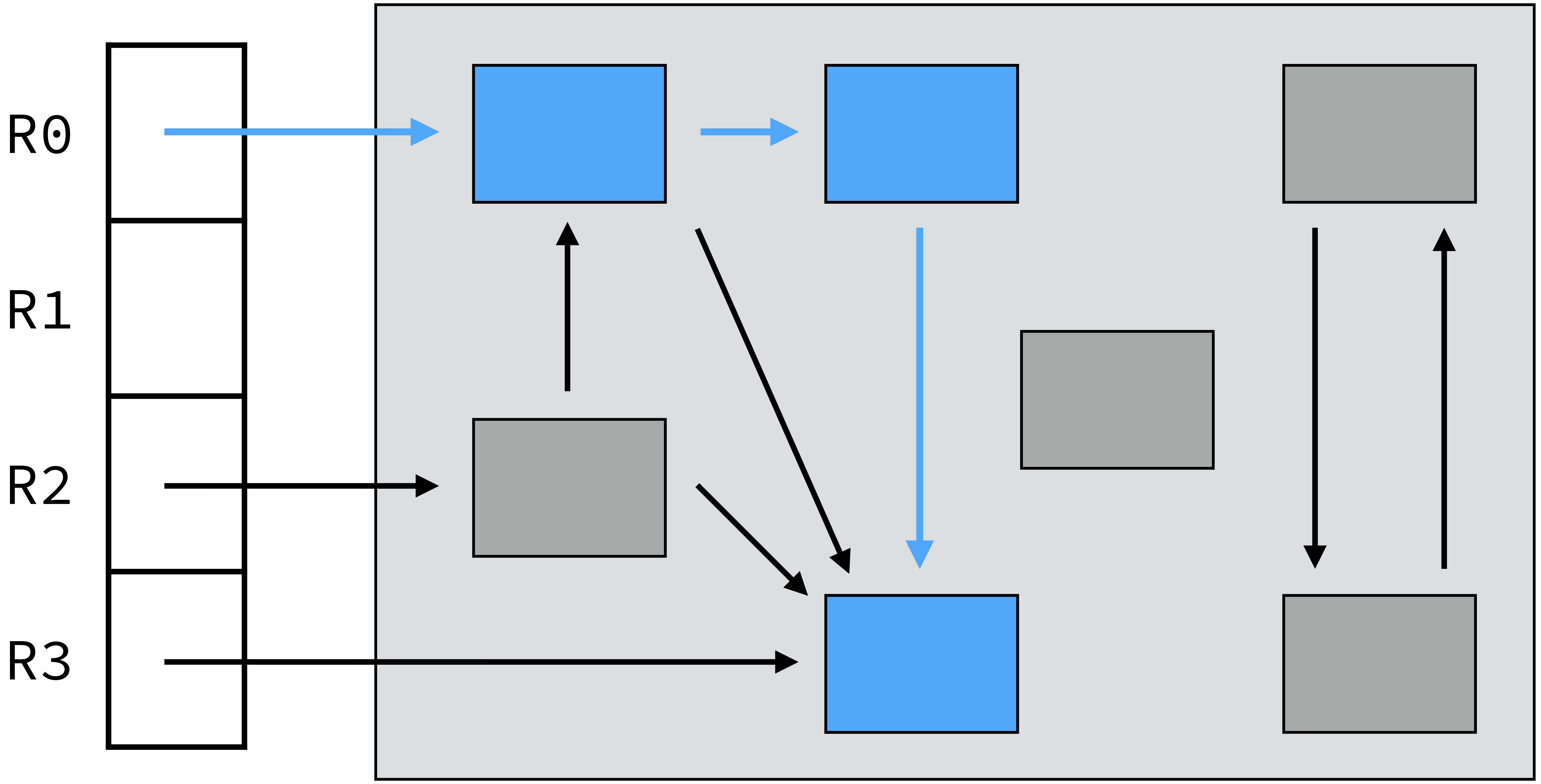
Mark & sweep GC



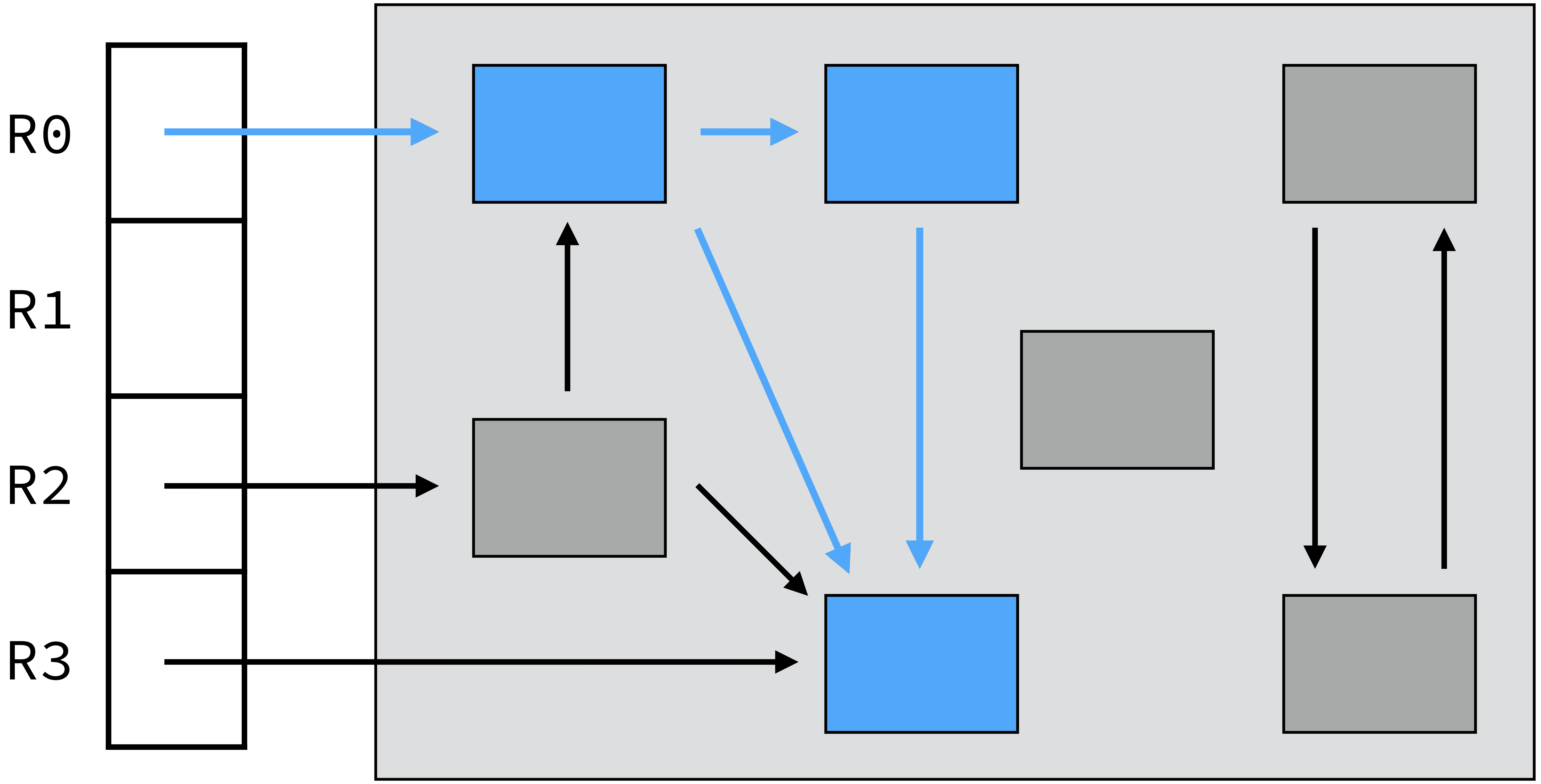
Mark & sweep GC



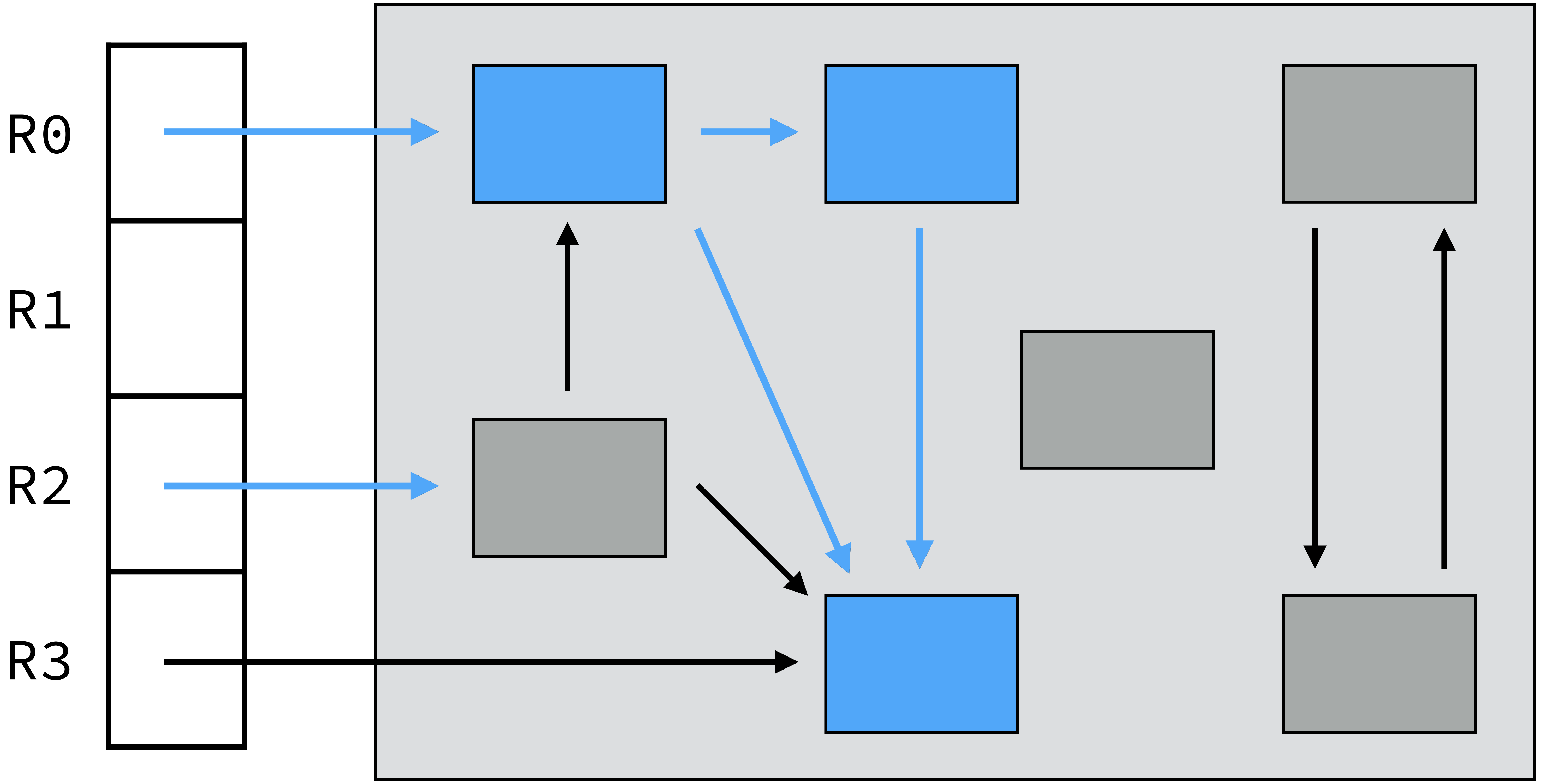
Mark & sweep GC



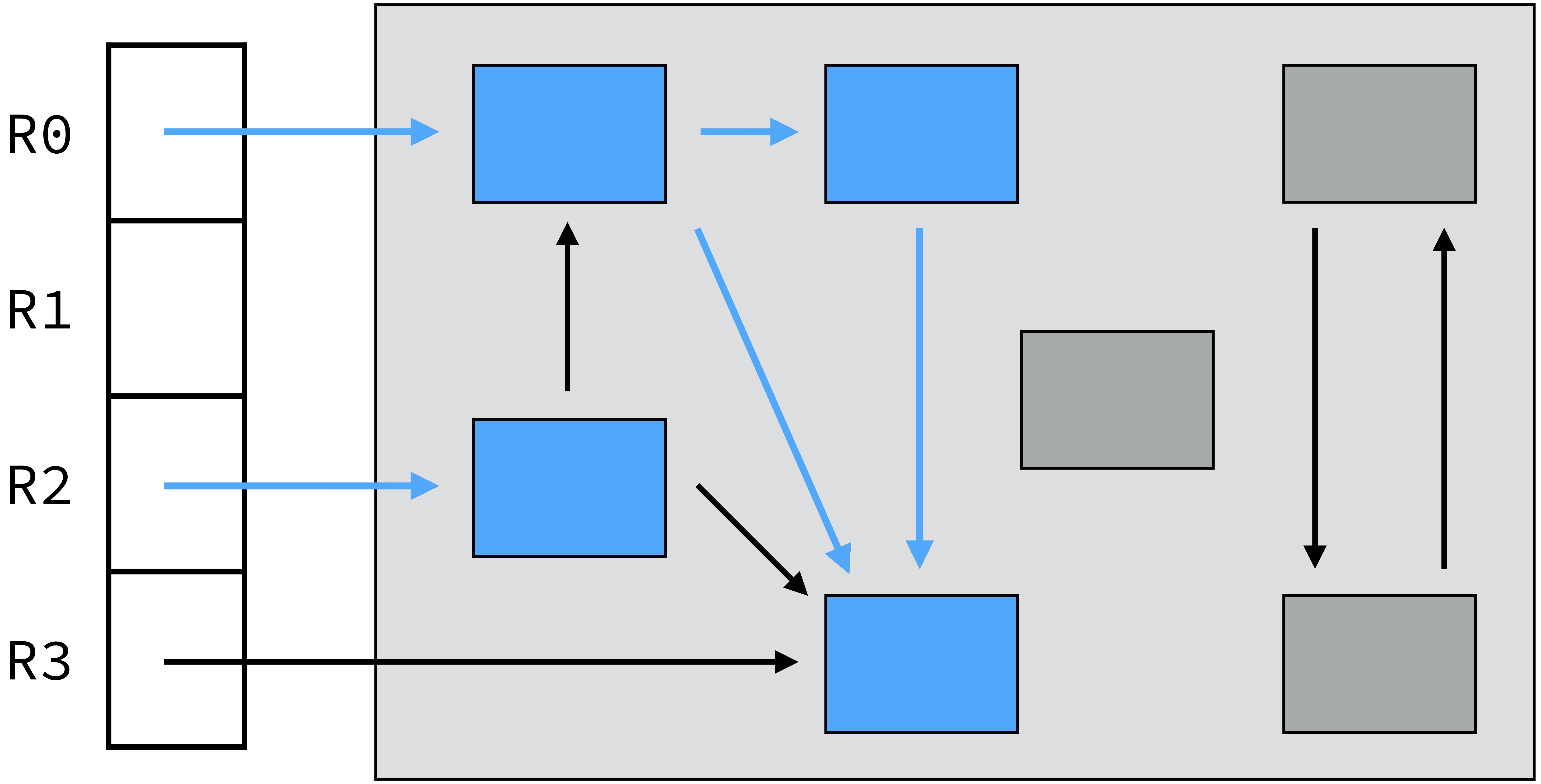
Mark & sweep GC



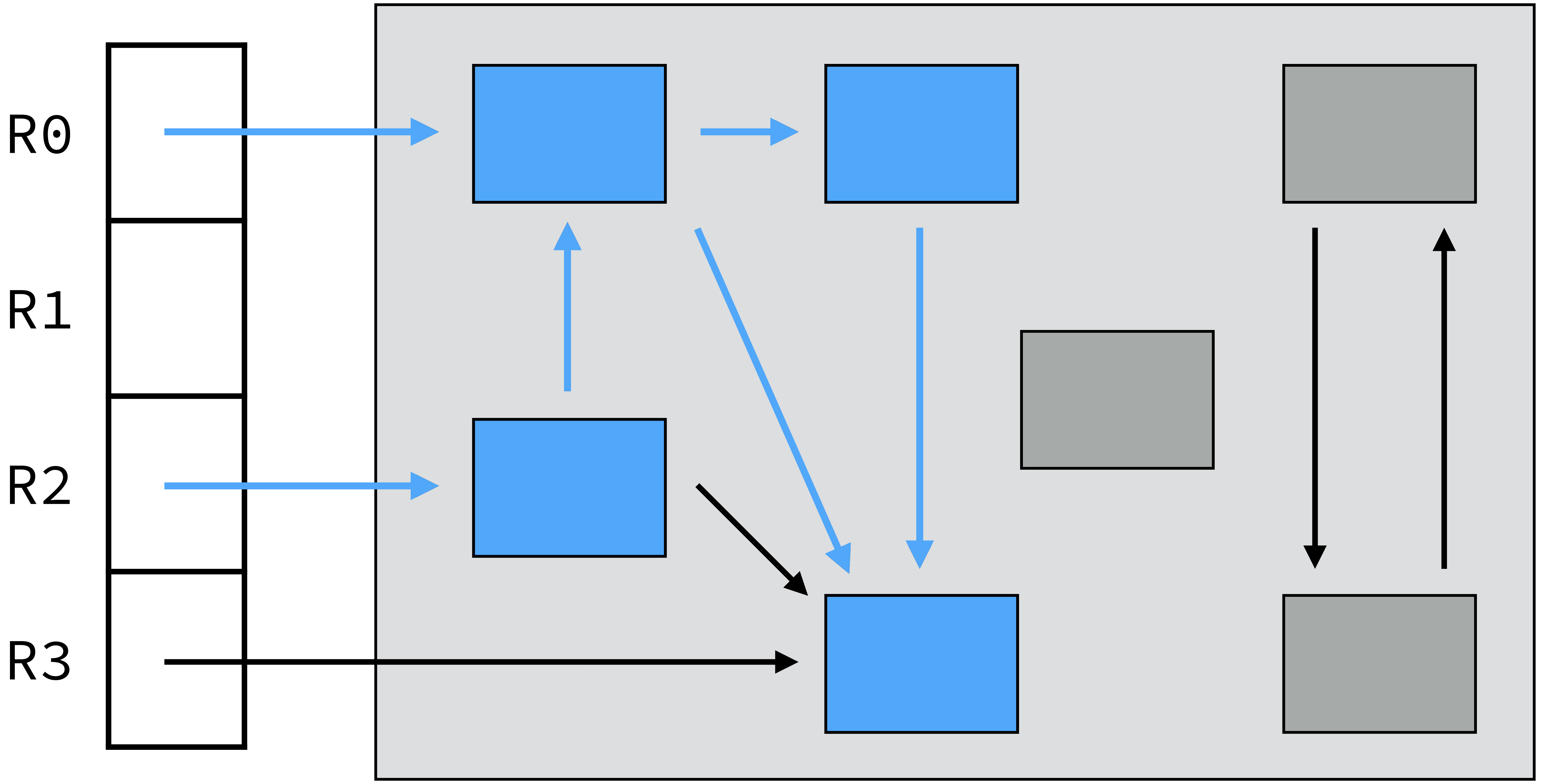
Mark & sweep GC



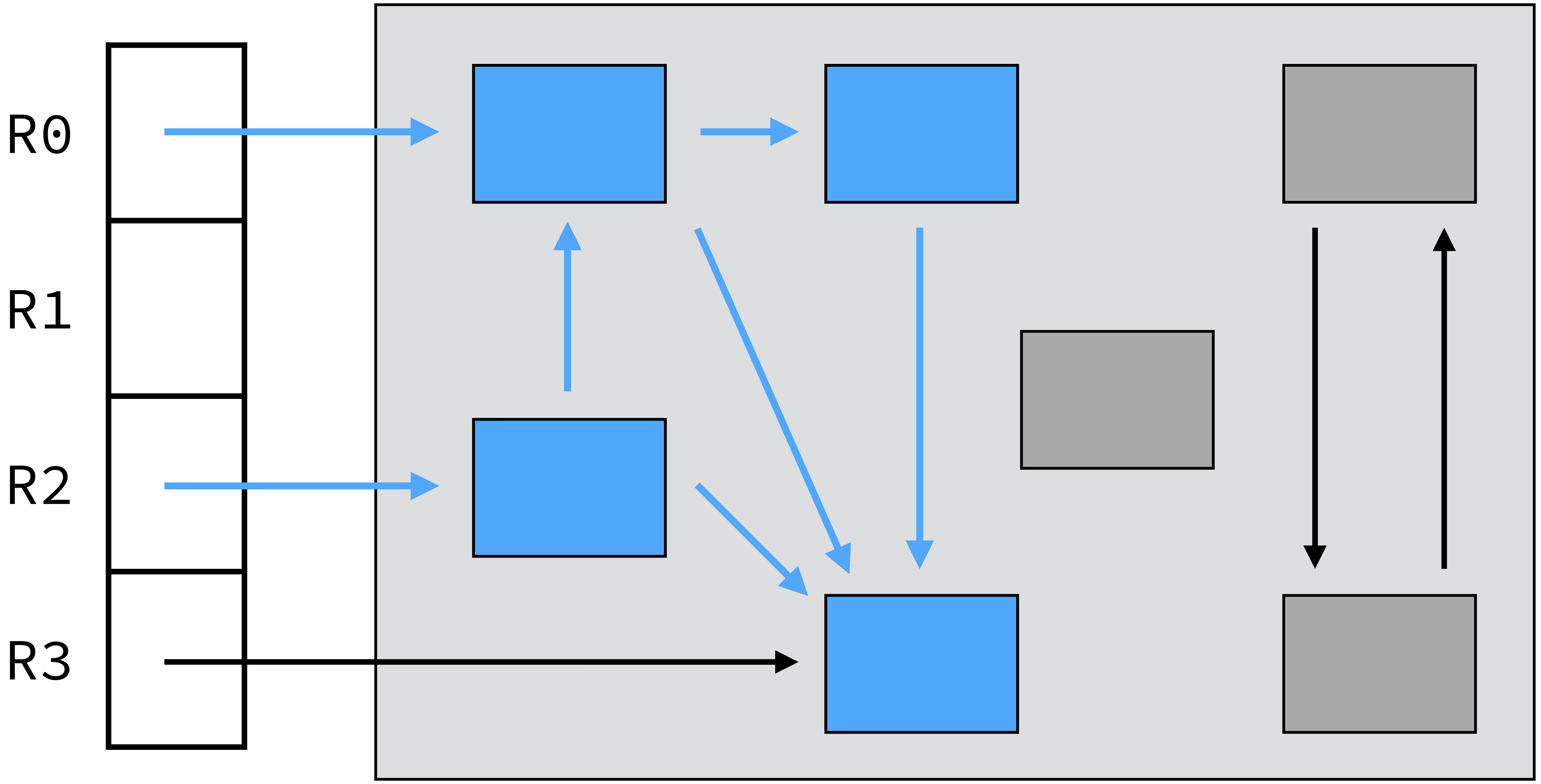
Mark & sweep GC



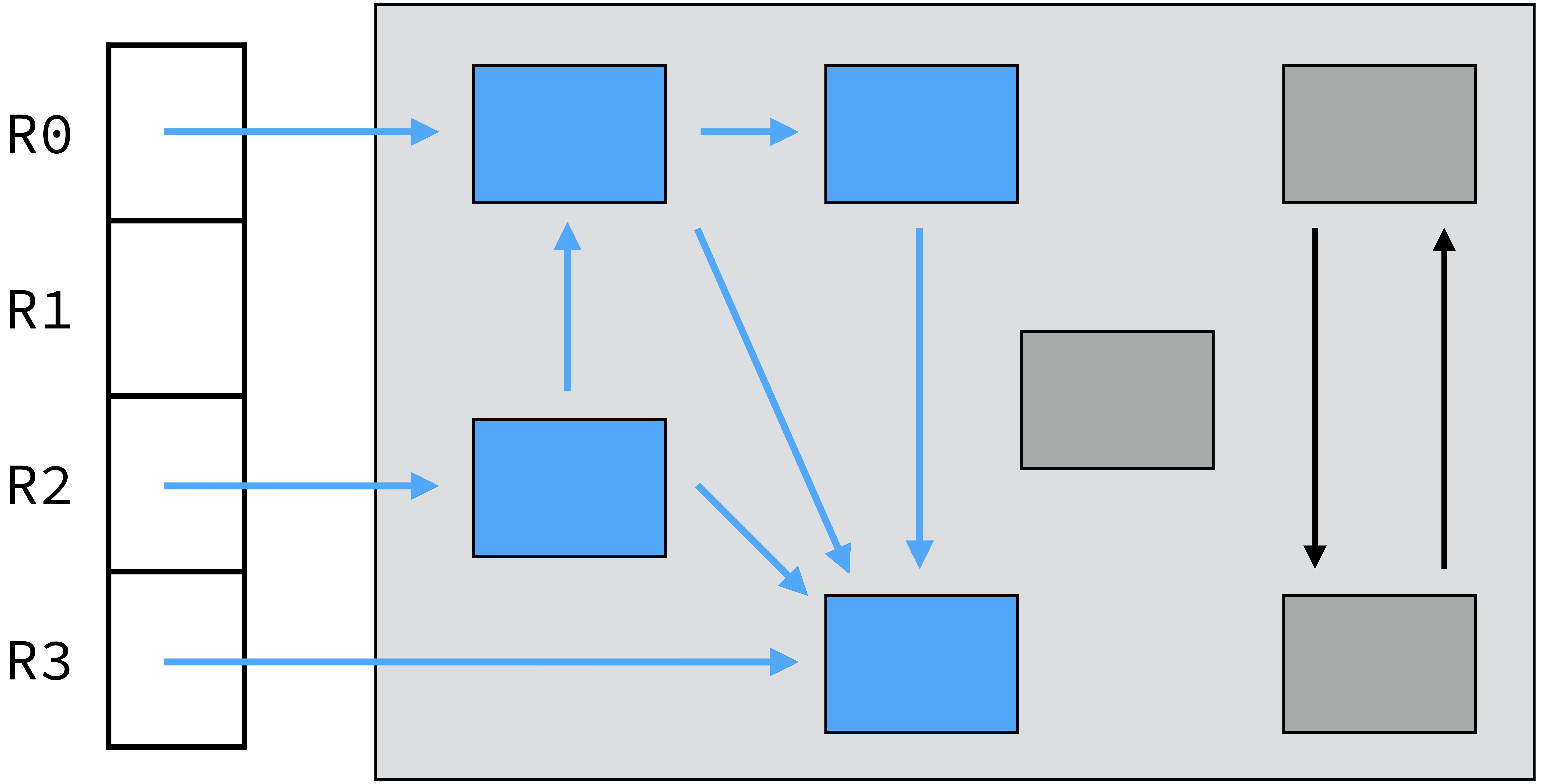
Mark & sweep GC



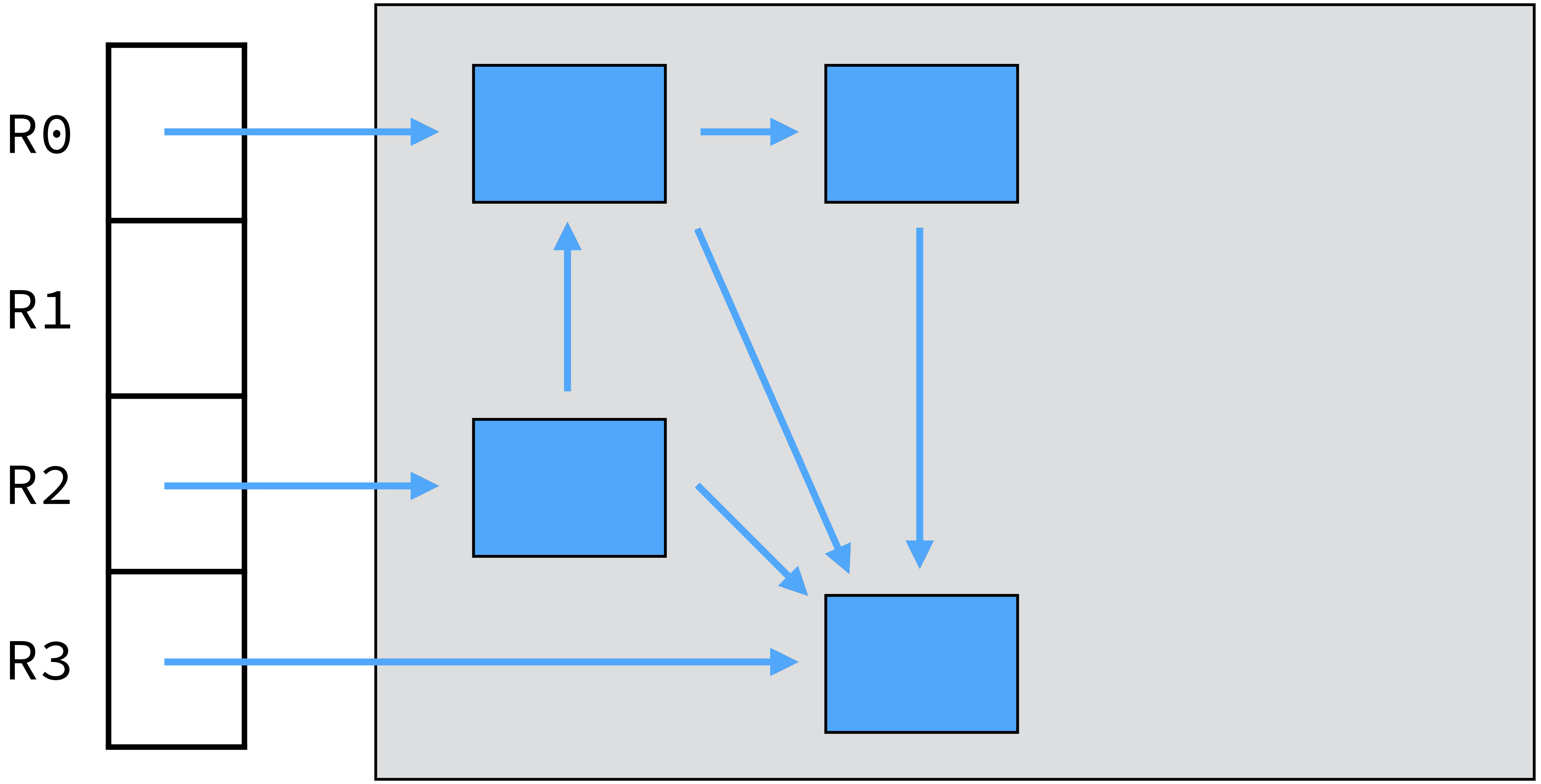
Mark & sweep GC



Mark & sweep GC



Mark & sweep GC



Marking objects

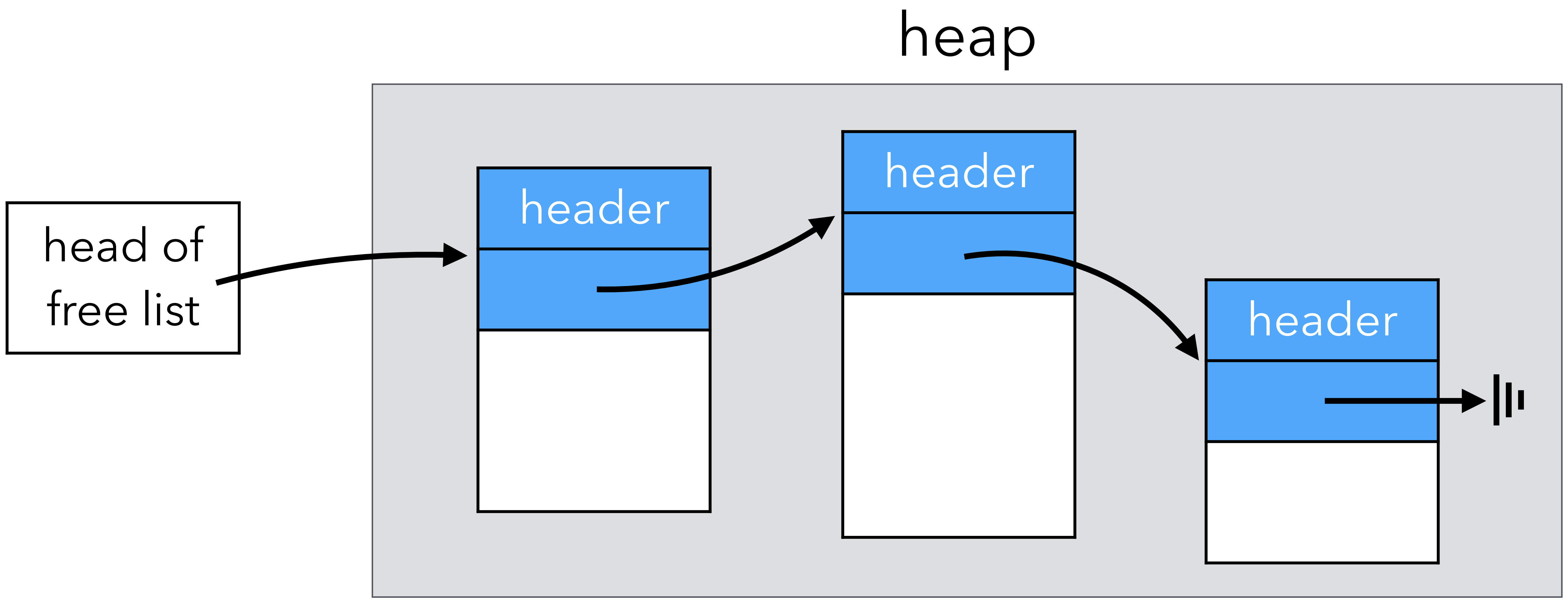
Reachable objects can be marked by:

- setting one bit in the header (e.g. the LSB of the size, if it's always even),
- setting one bit in an external bit map, stored in an area that is private to the GC.

Free list

In a mark & sweep GC, free blocks are not contiguous, and must be stored in a free list of some sort.

Note: the list links can be stored in the blocks themselves, as they are free!



Allocation policy

When more than one free block can be used to satisfy a request, the memory manager uses an **allocation policy** to decide which one to use.

A good policy should:

- be fast,
- minimize fragmentation.

The most commonly used are:

- **first fit**: use the first suitable block,
- **best fit**: use the smallest suitable block.

Splitting and coalescing

During allocation, if the chosen block is bigger than what is requested, it must be **split** in two:

- the first part is returned to the program,
- the other part is put back into the free list.

During deallocation, if the freed block is adjacent to other free blocks, they must be **coalesced** into one.

Reachability graph traversal

Marking objects is usually done by depth-first traversal of the reachability graph. When done recursively, this can consume an unbounded amount of stack space, which can lead to stack overflow.

Solutions (not examined here):

- recover from stack overflows,
- do pointer reversal (i.e., store the stack in the traversed objects, in a way).

Sweeping objects

The sweeping phase:

– traverses *the whole heap*,

– rebuilds the free list by adding unmarked objects to it,

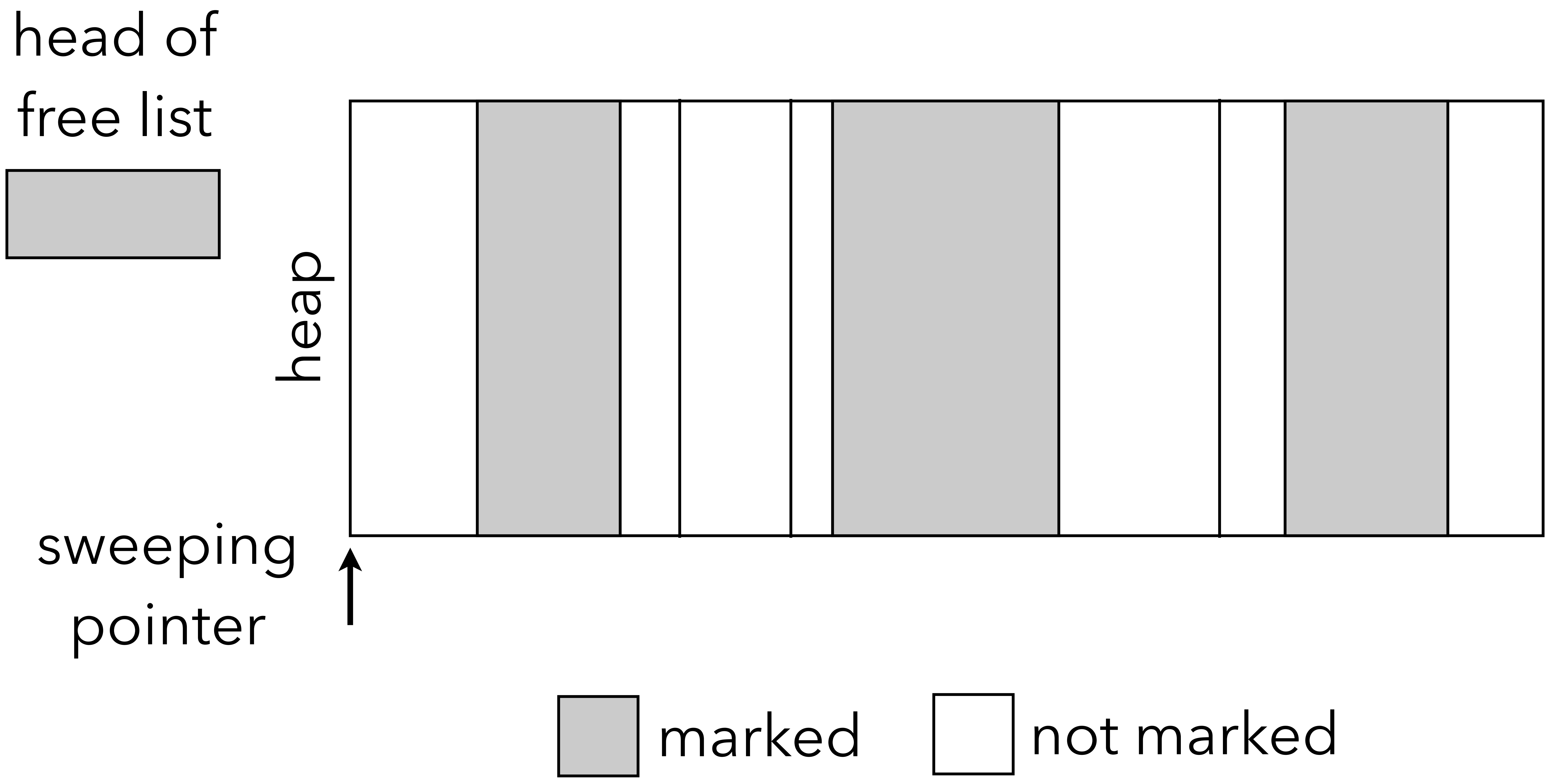
– does coalescing at the same time.

Since unreachable objects cannot become reachable again, the sweeping phase can be done incrementally. This is called **lazy sweep**.

Sweeping and coalescing

The sweeping phase:

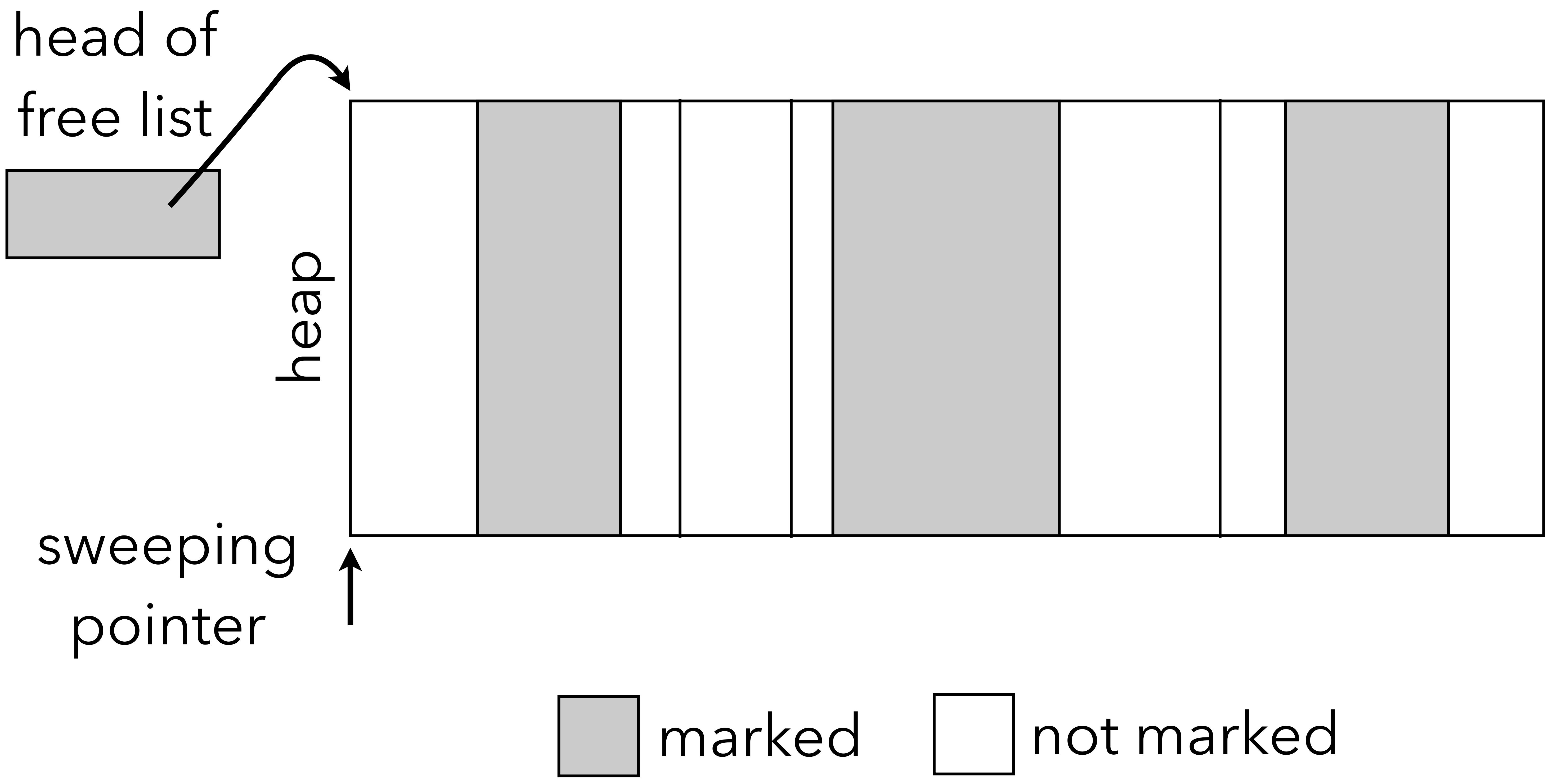
- traverses the whole heap, rebuilding the free list,
- performs coalescing at the same time.



Sweeping and coalescing

The sweeping phase:

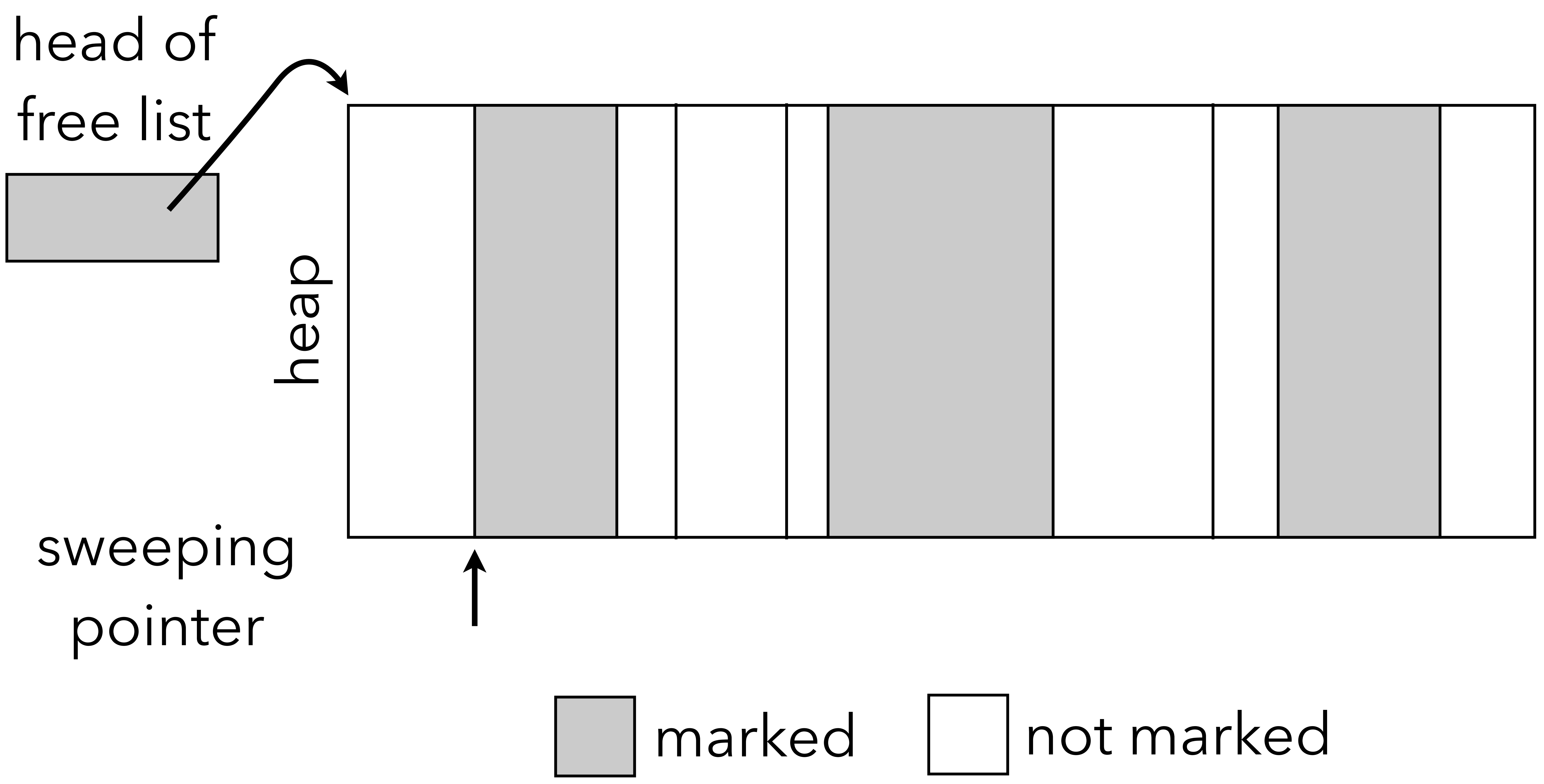
- traverses the whole heap, rebuilding the free list,
- performs coalescing at the same time.



Sweeping and coalescing

The sweeping phase:

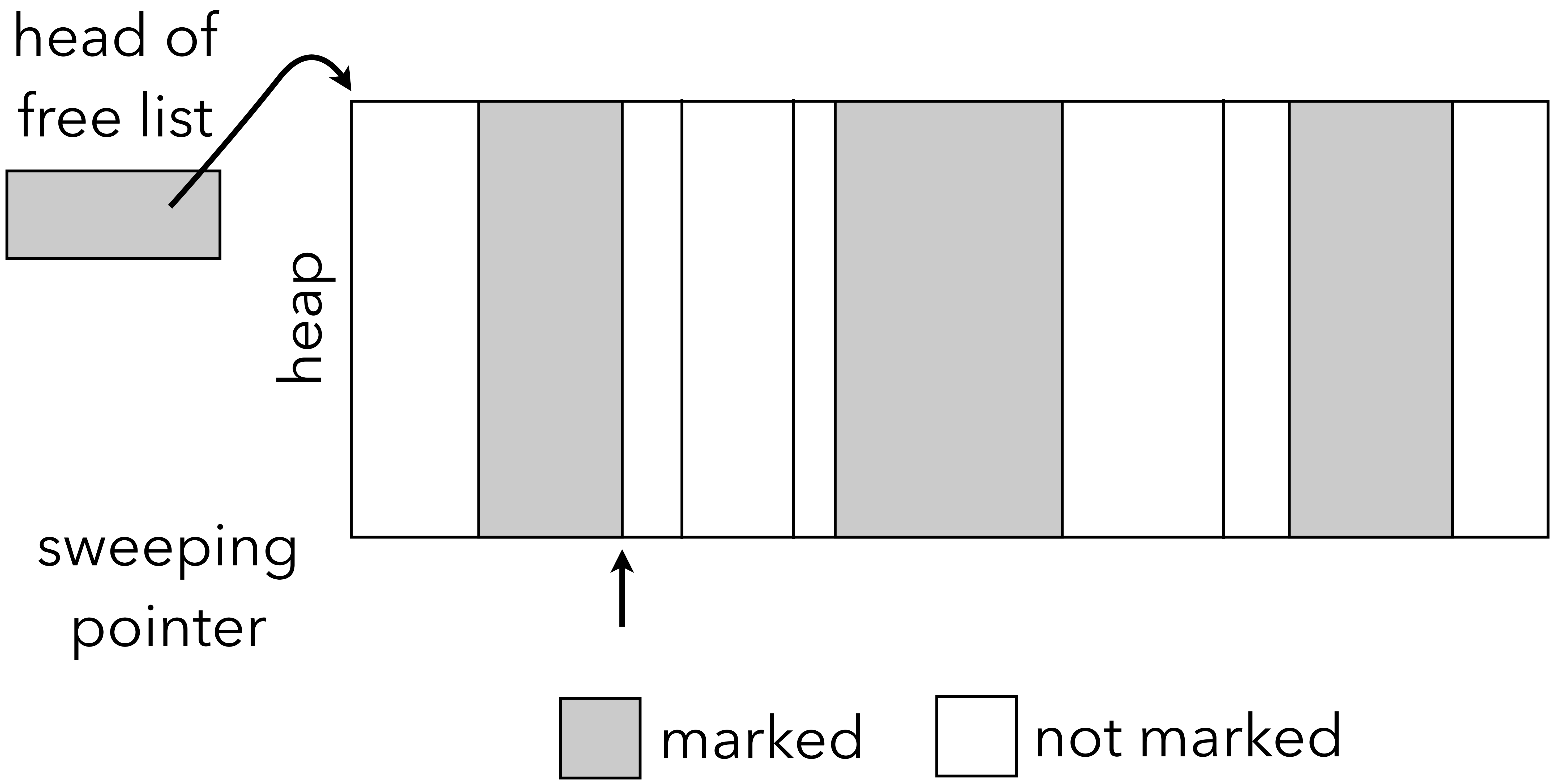
- traverses the whole heap, rebuilding the free list,
- performs coalescing at the same time.



Sweeping and coalescing

The sweeping phase:

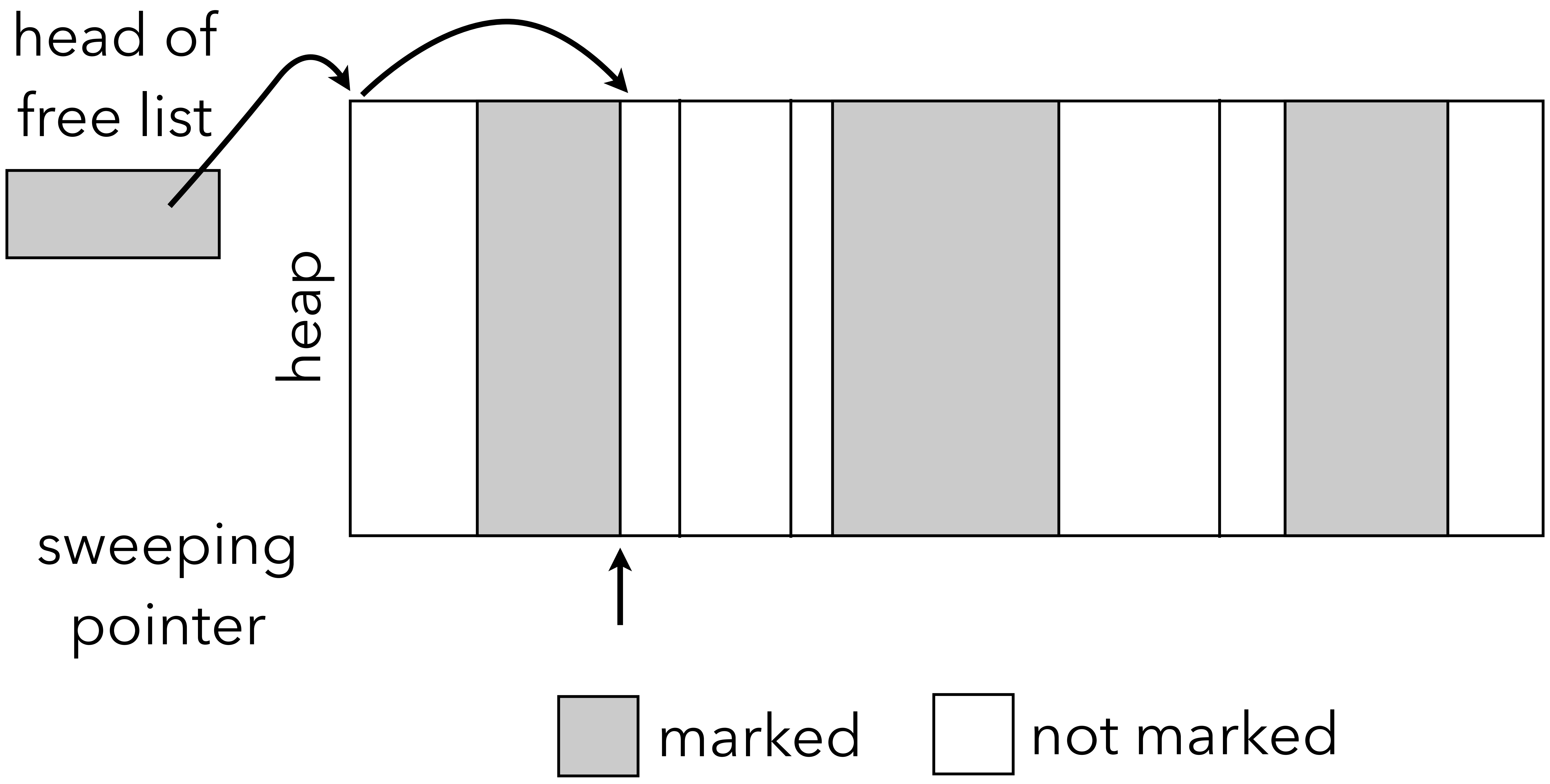
- traverses the whole heap, rebuilding the free list,
- performs coalescing at the same time.



Sweeping and coalescing

The sweeping phase:

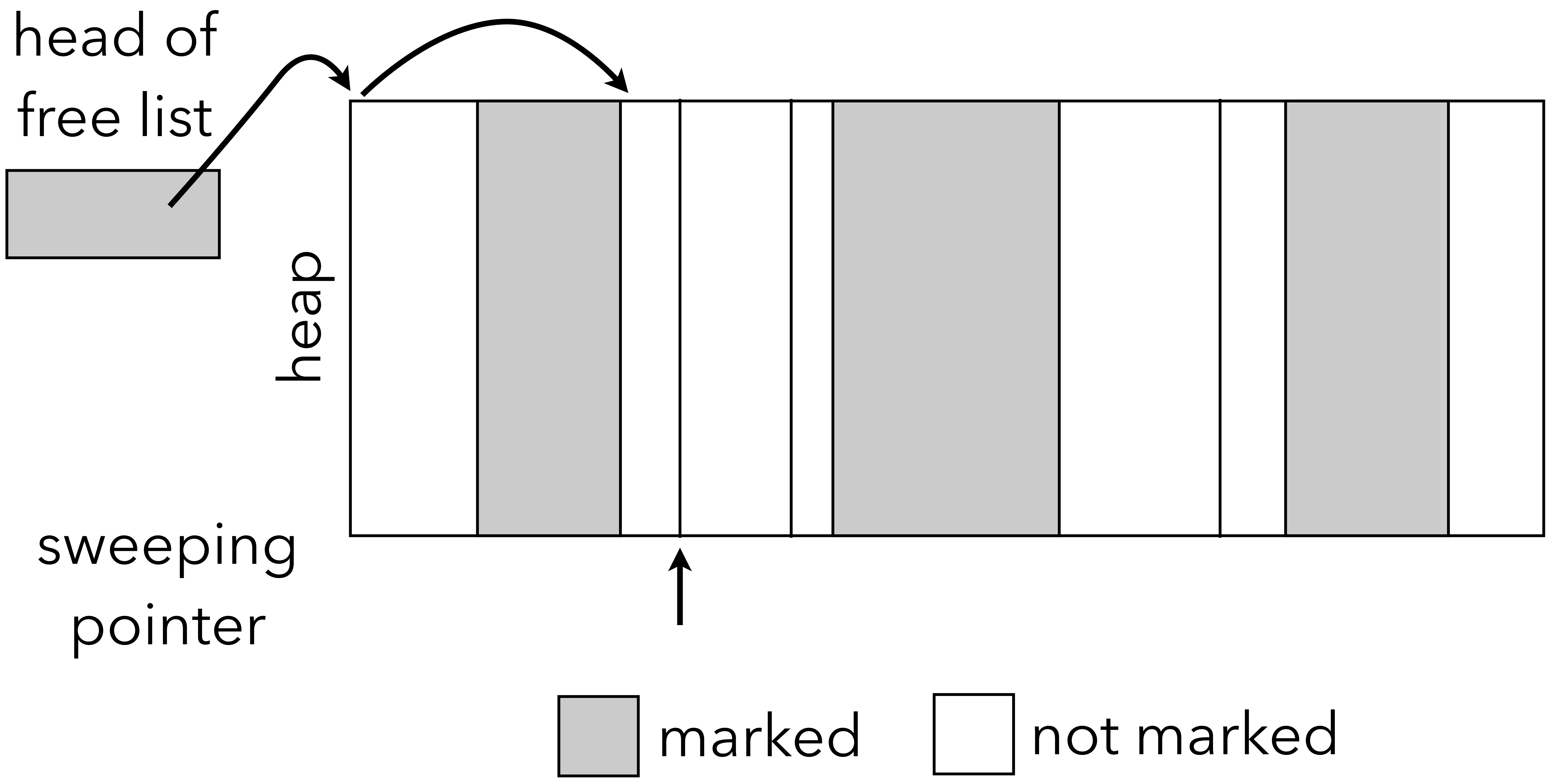
- traverses the whole heap, rebuilding the free list,
- performs coalescing at the same time.



Sweeping and coalescing

The sweeping phase:

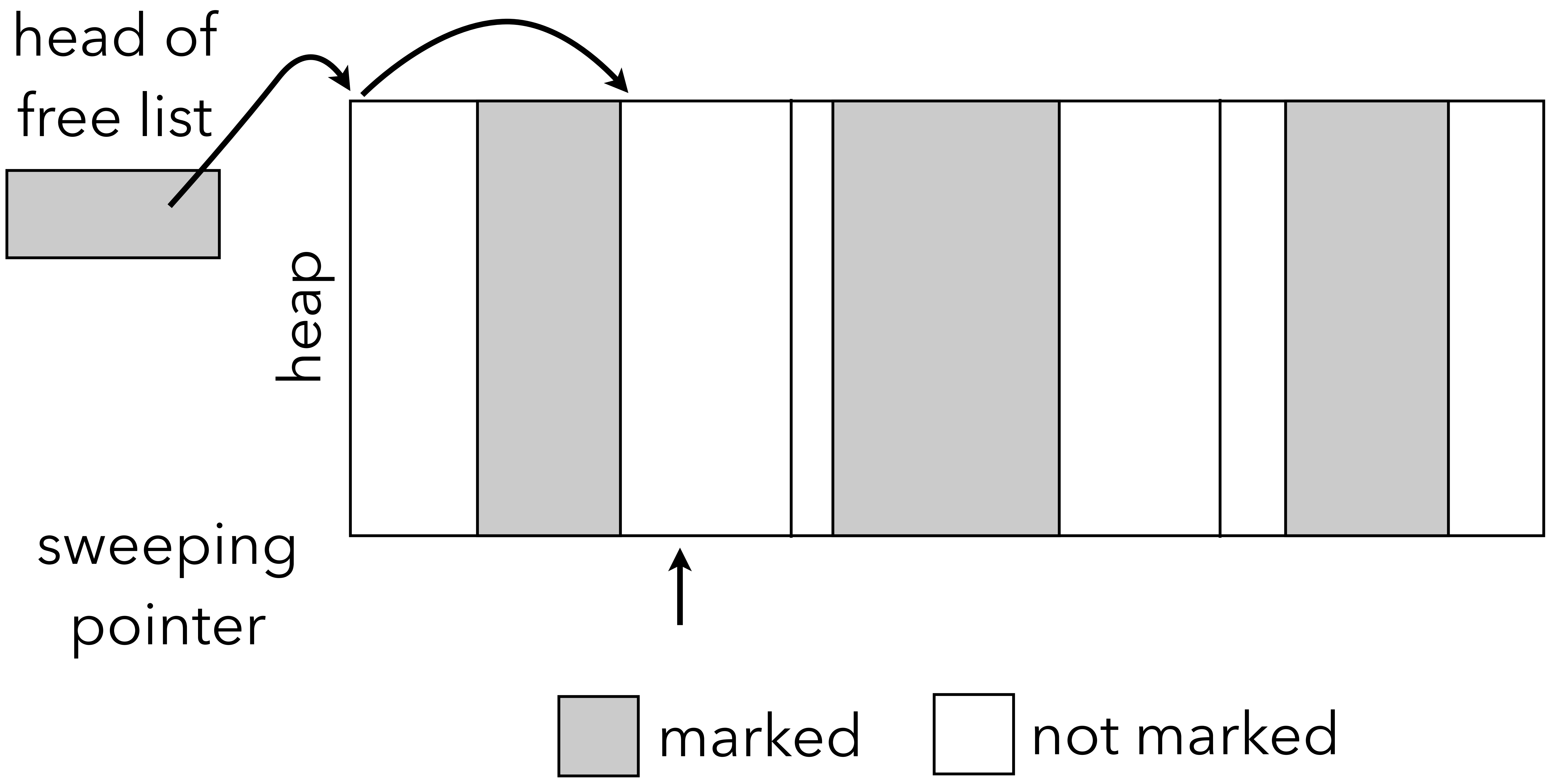
- traverses the whole heap, rebuilding the free list,
- performs coalescing at the same time.



Sweeping and coalescing

The sweeping phase:

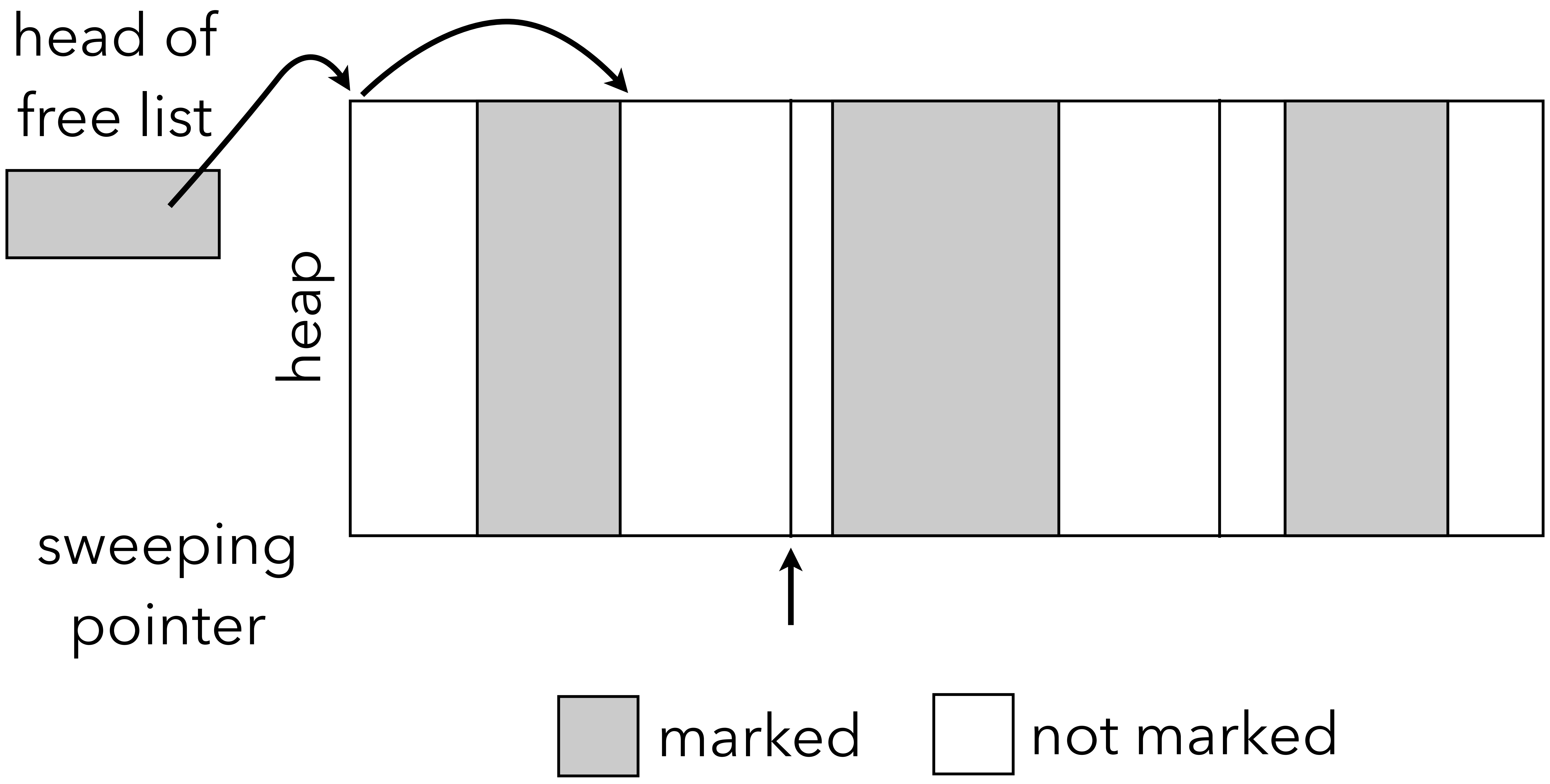
- traverses the whole heap, rebuilding the free list,
- performs coalescing at the same time.



Sweeping and coalescing

The sweeping phase:

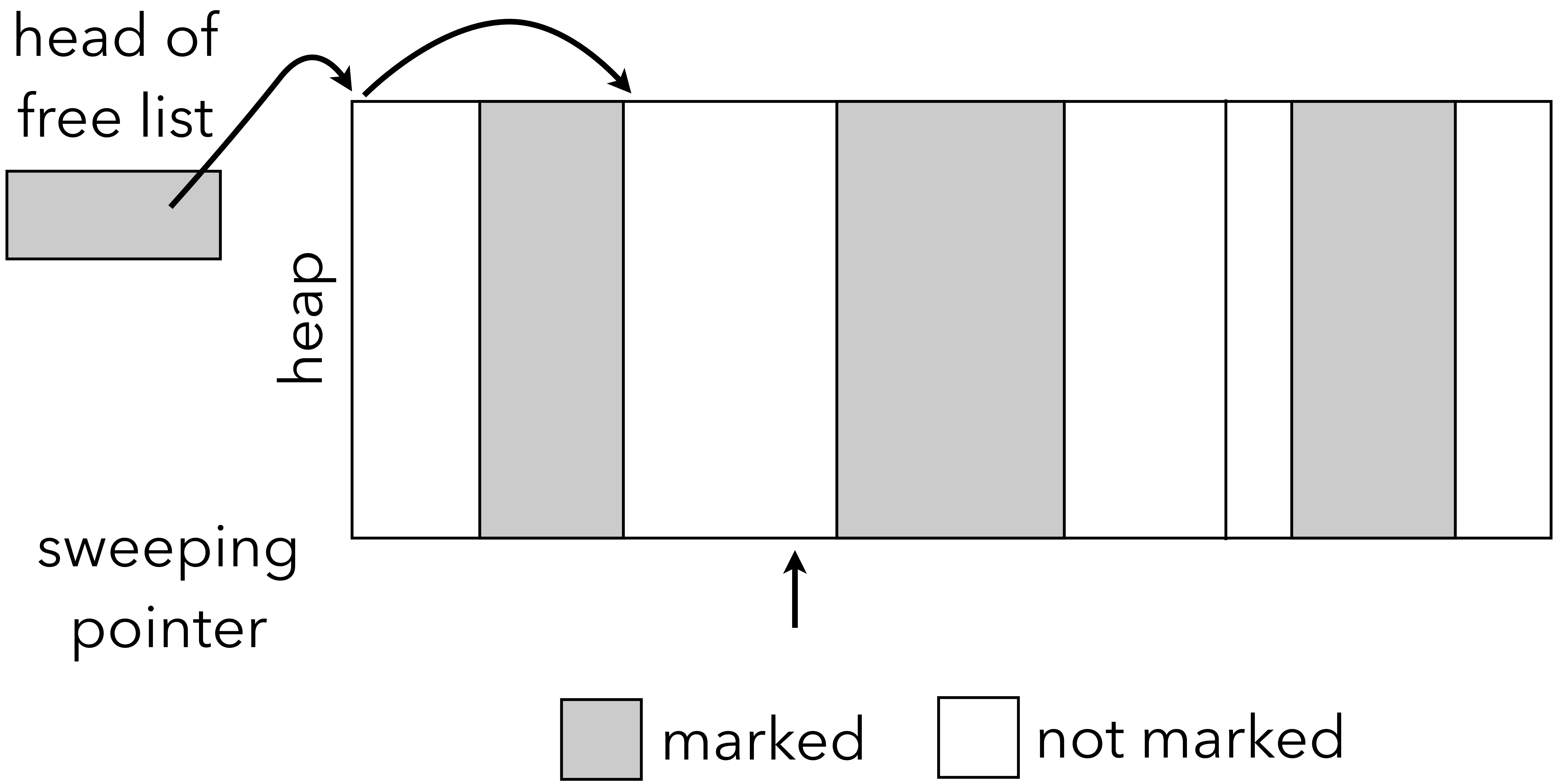
- traverses the whole heap, rebuilding the free list,
- performs coalescing at the same time.



Sweeping and coalescing

The sweeping phase:

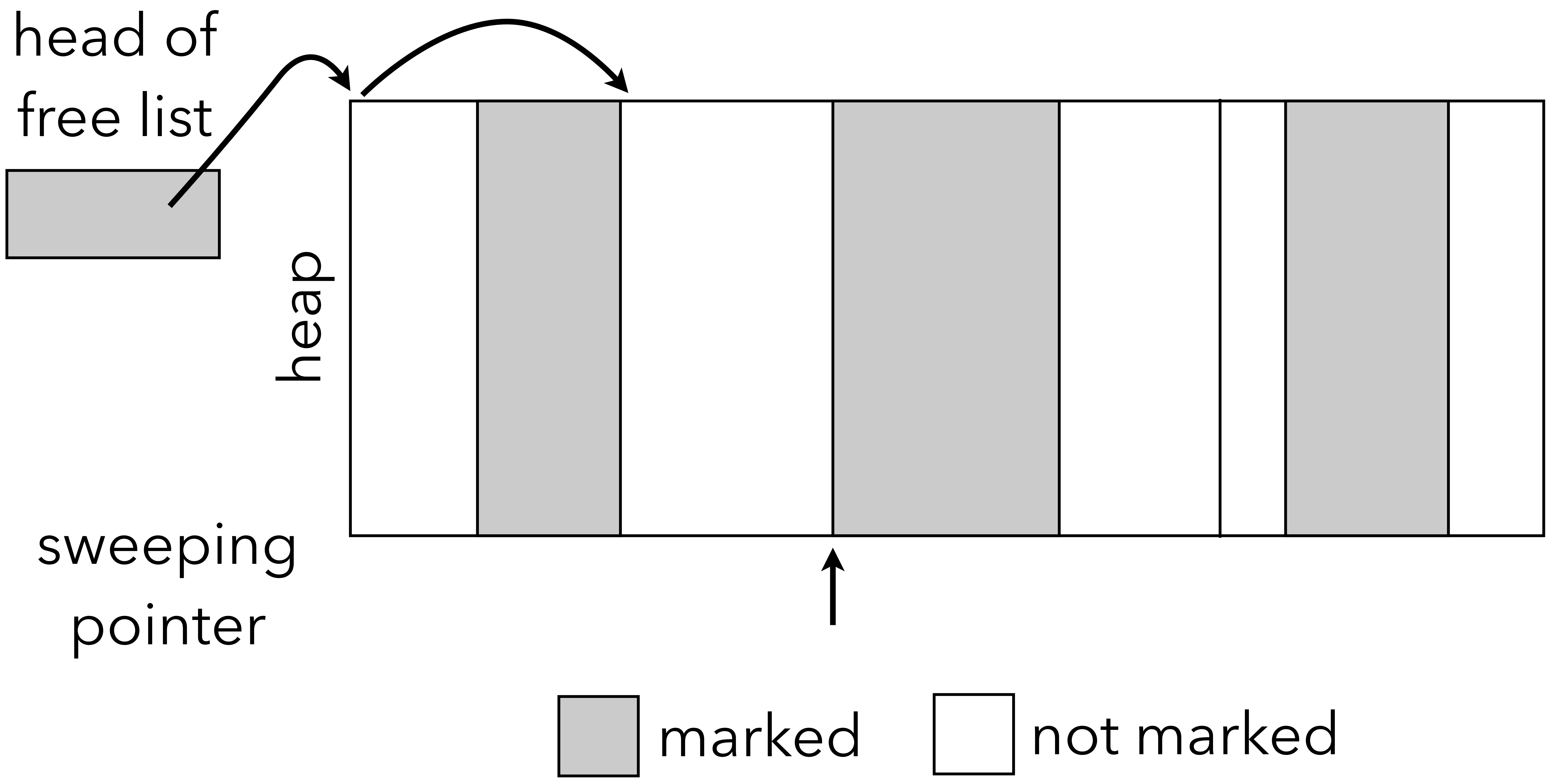
- traverses the whole heap, rebuilding the free list,
- performs coalescing at the same time.



Sweeping and coalescing

The sweeping phase:

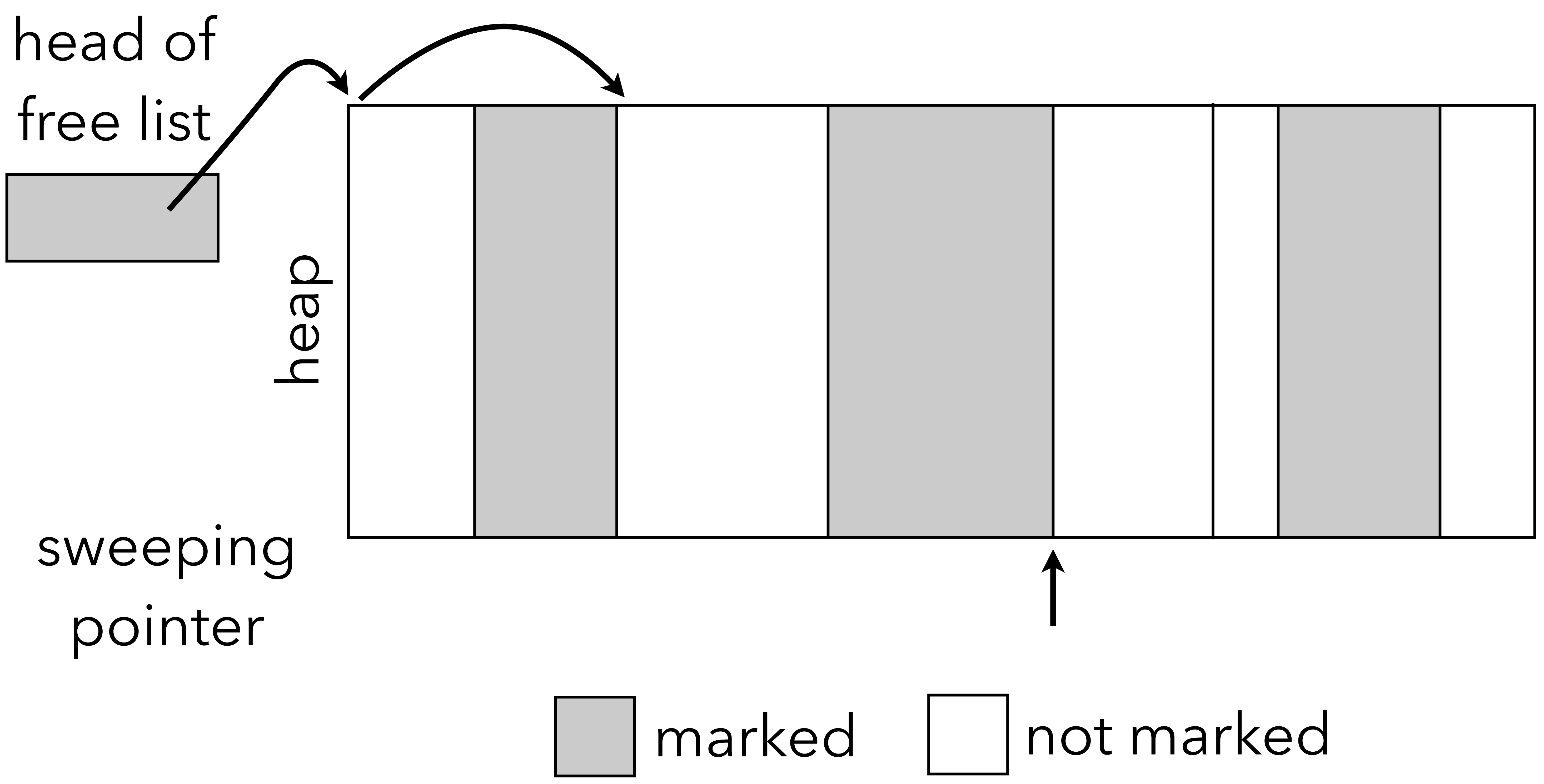
- traverses the whole heap, rebuilding the free list,
- performs coalescing at the same time.



Sweeping and coalescing

The sweeping phase:

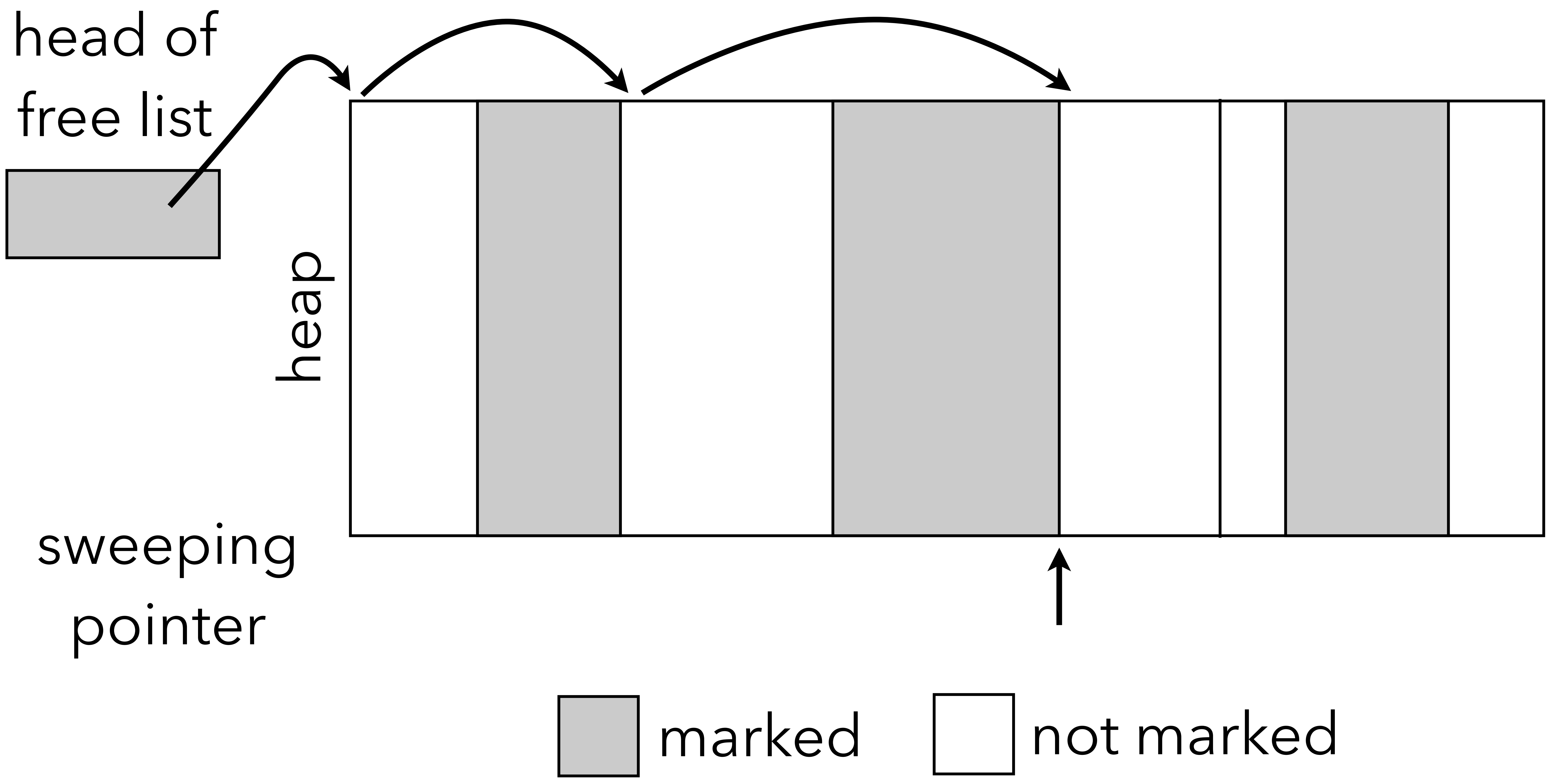
- traverses the whole heap, rebuilding the free list,
- performs coalescing at the same time.



Sweeping and coalescing

The sweeping phase:

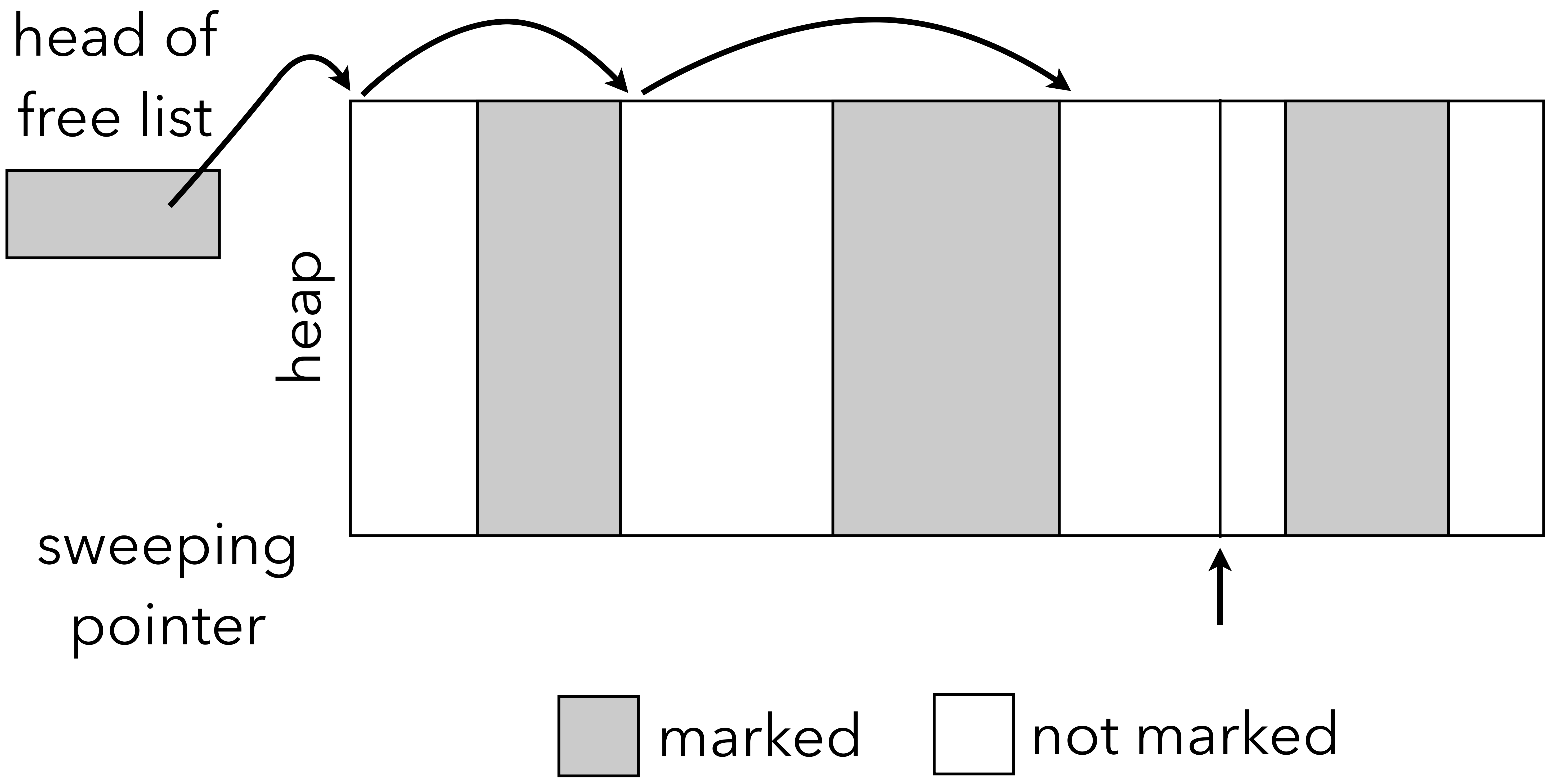
- traverses the whole heap, rebuilding the free list,
- performs coalescing at the same time.



Sweeping and coalescing

The sweeping phase:

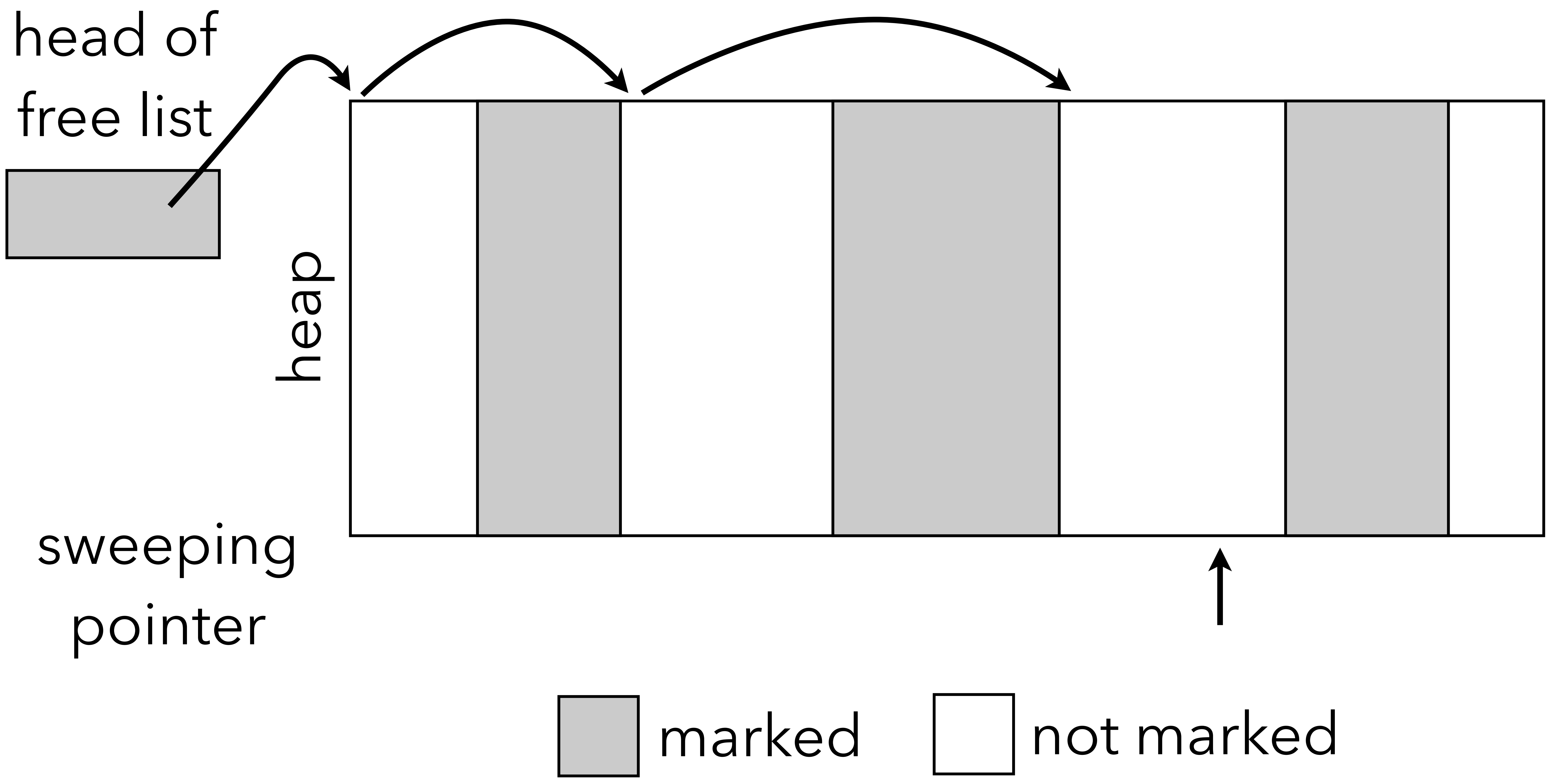
- traverses the whole heap, rebuilding the free list,
- performs coalescing at the same time.



Sweeping and coalescing

The sweeping phase:

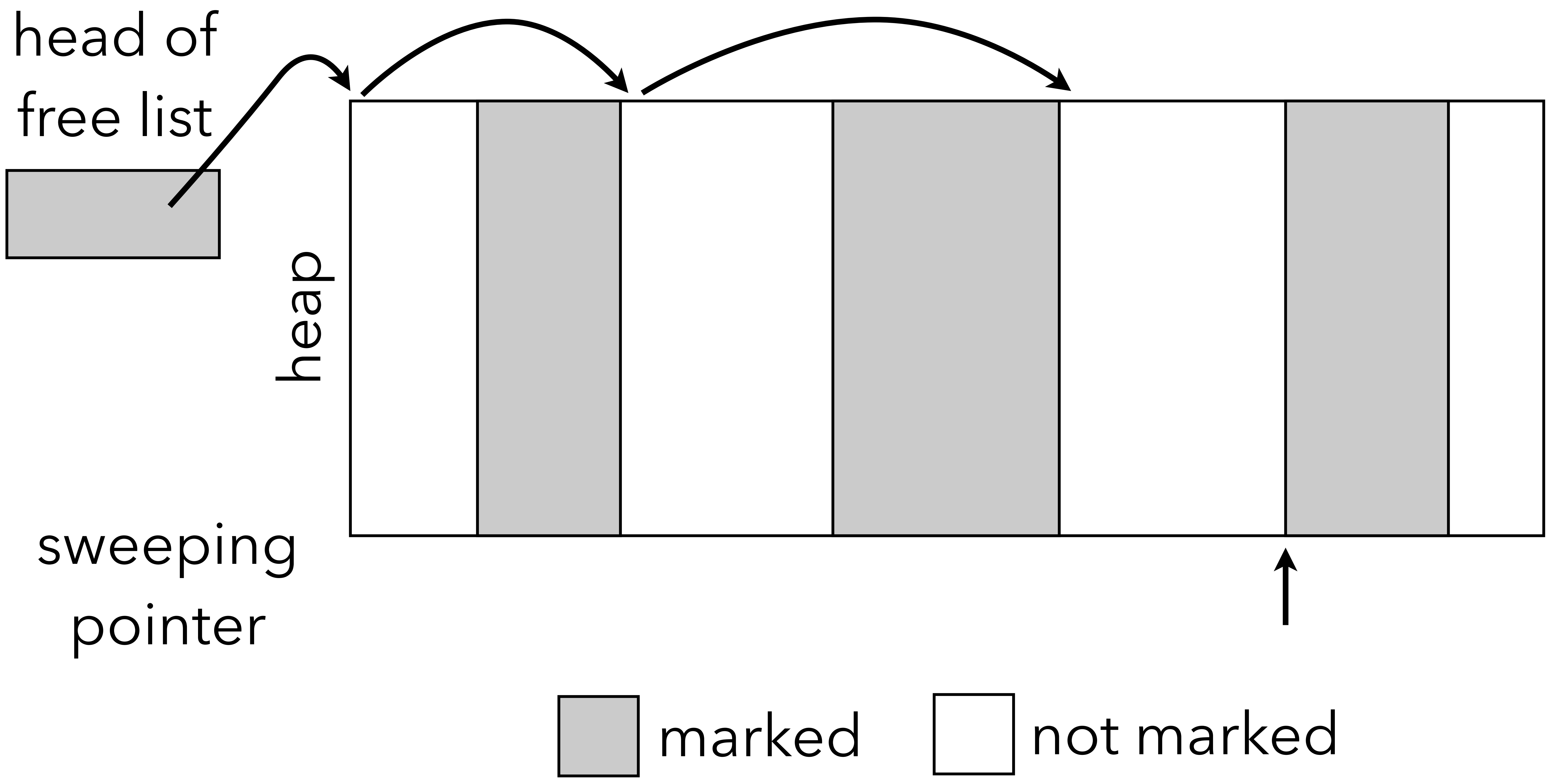
- traverses the whole heap, rebuilding the free list,
- performs coalescing at the same time.



Sweeping and coalescing

The sweeping phase:

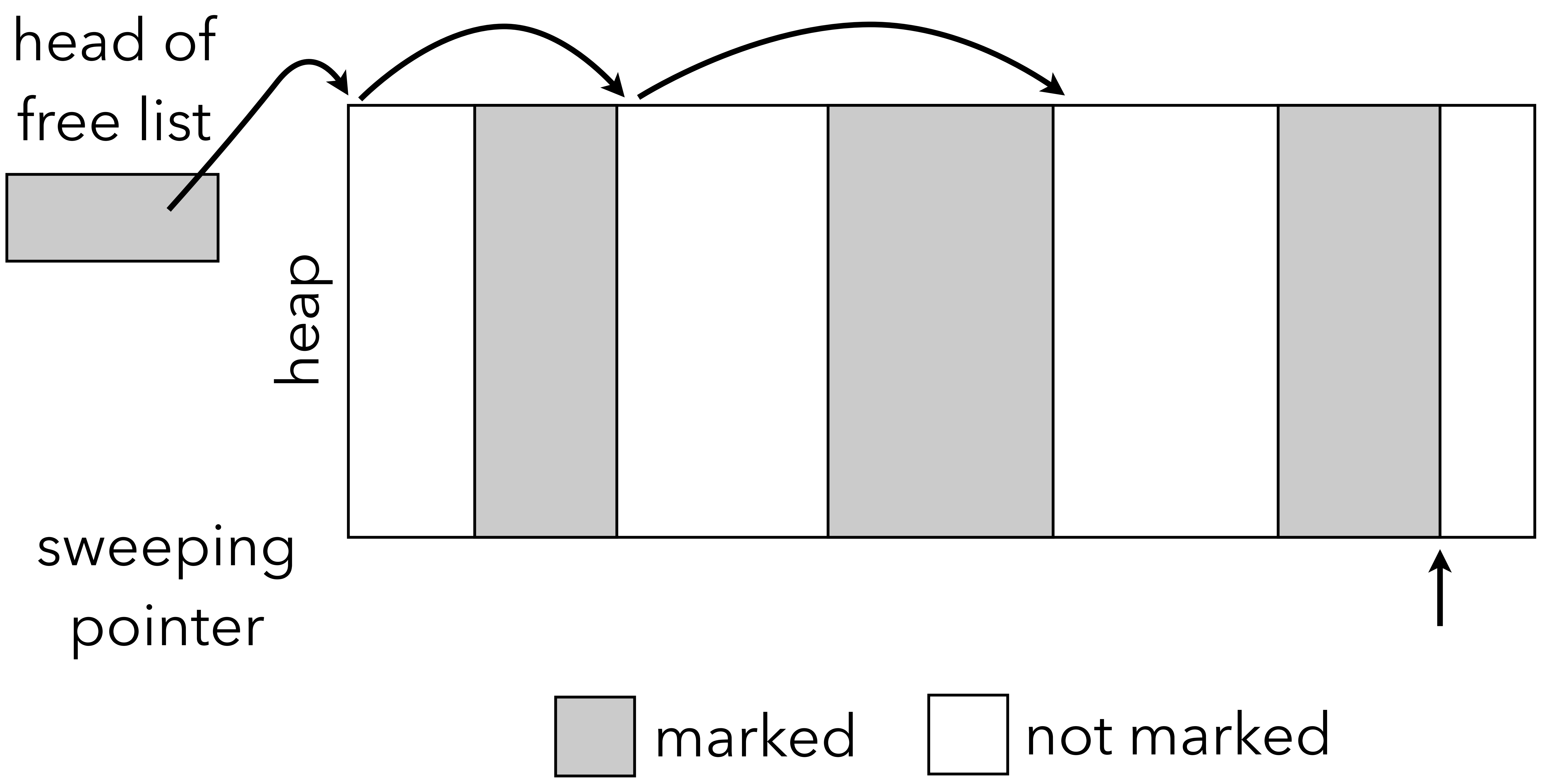
- traverses the whole heap, rebuilding the free list,
- performs coalescing at the same time.



Sweeping and coalescing

The sweeping phase:

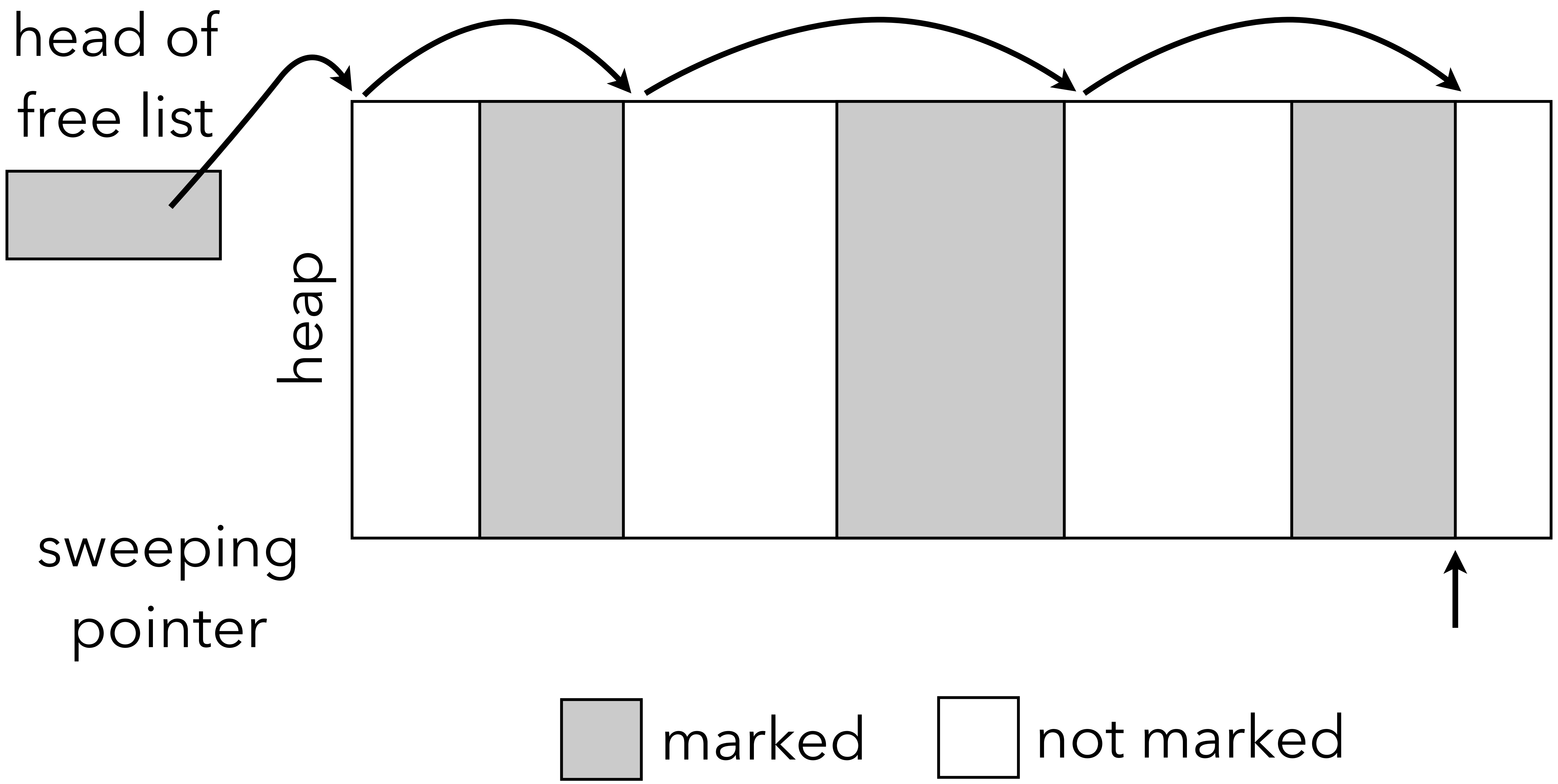
- traverses the whole heap, rebuilding the free list,
- performs coalescing at the same time.



Sweeping and coalescing

The sweeping phase:

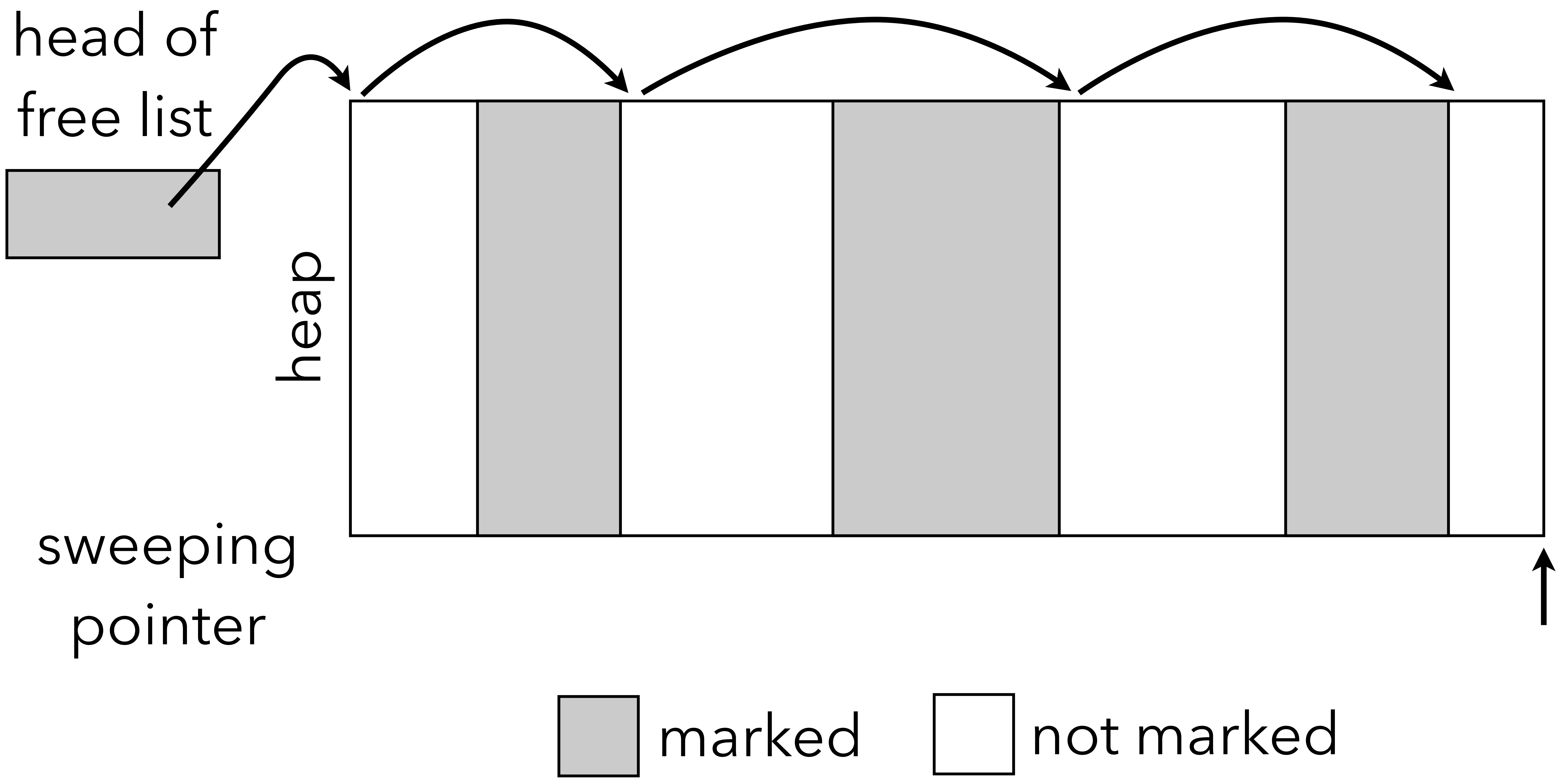
- traverses the whole heap, rebuilding the free list,
- performs coalescing at the same time.



Sweeping and coalescing

The sweeping phase:

- traverses the whole heap, rebuilding the free list,
- performs coalescing at the same time.



Conservative M&S GC

A **conservative** mark & sweep garbage collector collects memory without unambiguously identify pointers at run time.
Doable iff the approximation of the reachability graph *includes* the actual reachability graph.

Conservative M&S GC

Conservative GC assumption:

Everything that *looks* like a pointer to an allocated object *is* a pointer to an allocated object.

This assumption is:

- conservative (i.e. can lead to retention of unreachable objects),
- safe (i.e. cannot lead to reachable object being freed).

But: the GC must use as many hints as possible to minimize the mis-identification of pointers.

Pointer identification

Several characteristics of the architecture or compiler can be used to filter the set of potential pointers, e.g.:

- Many architectures require pointers to be aligned on 2 or 4 bytes boundaries. Therefore, ignore unaligned potential pointers.
- Many compilers guarantee that if an object is reachable, then there exists at least one pointer to its beginning. Therefore, ignore potential interior pointers.

Exercise

The POSIX `malloc` function does not clear the memory it returns to the user program, for performance reasons.

In a garbage collected environment, is it also a good idea to return freshly-allocated blocks to the program without clearing them first? Explain.

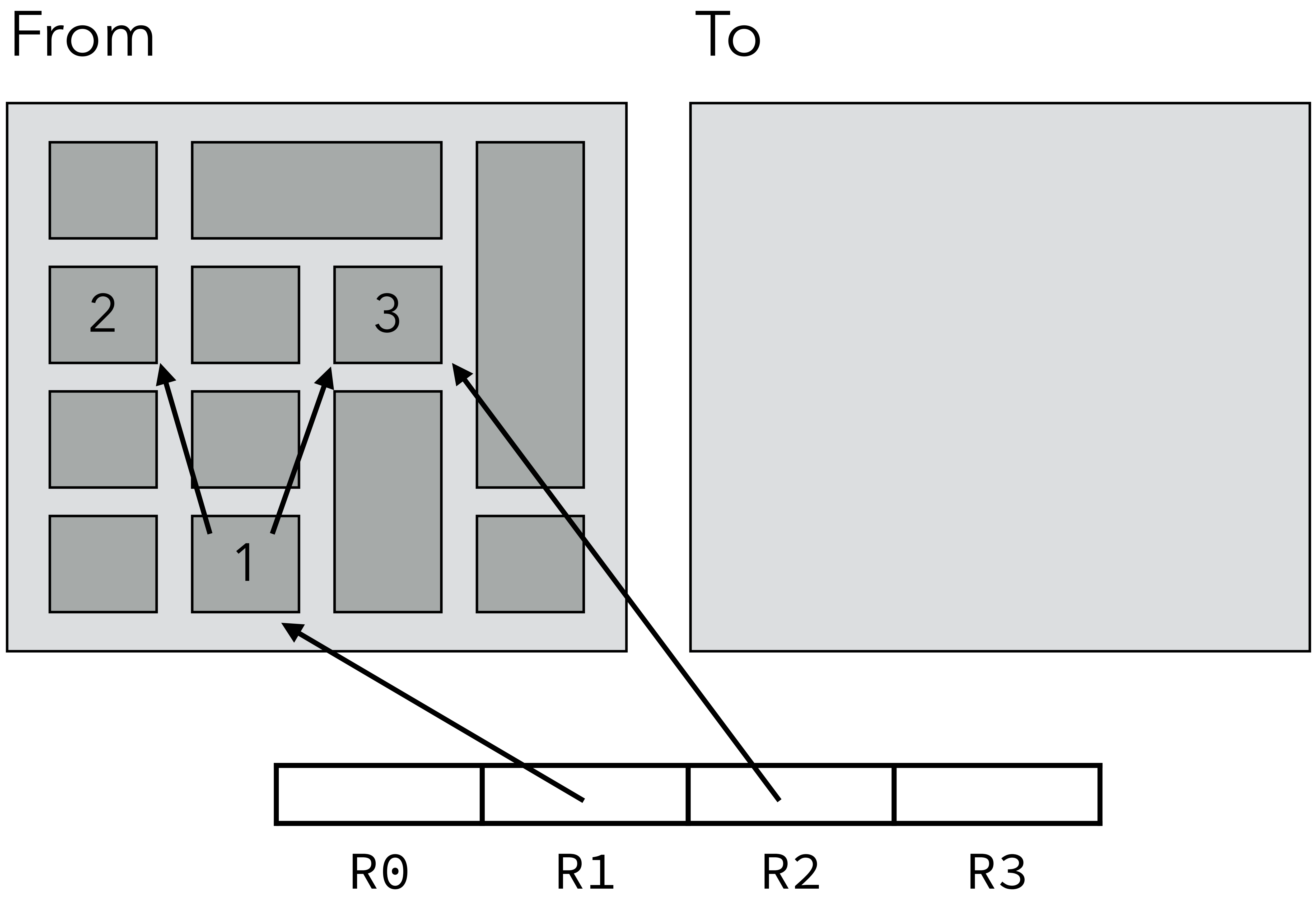
GC technique #3: copying GC

Copying GC

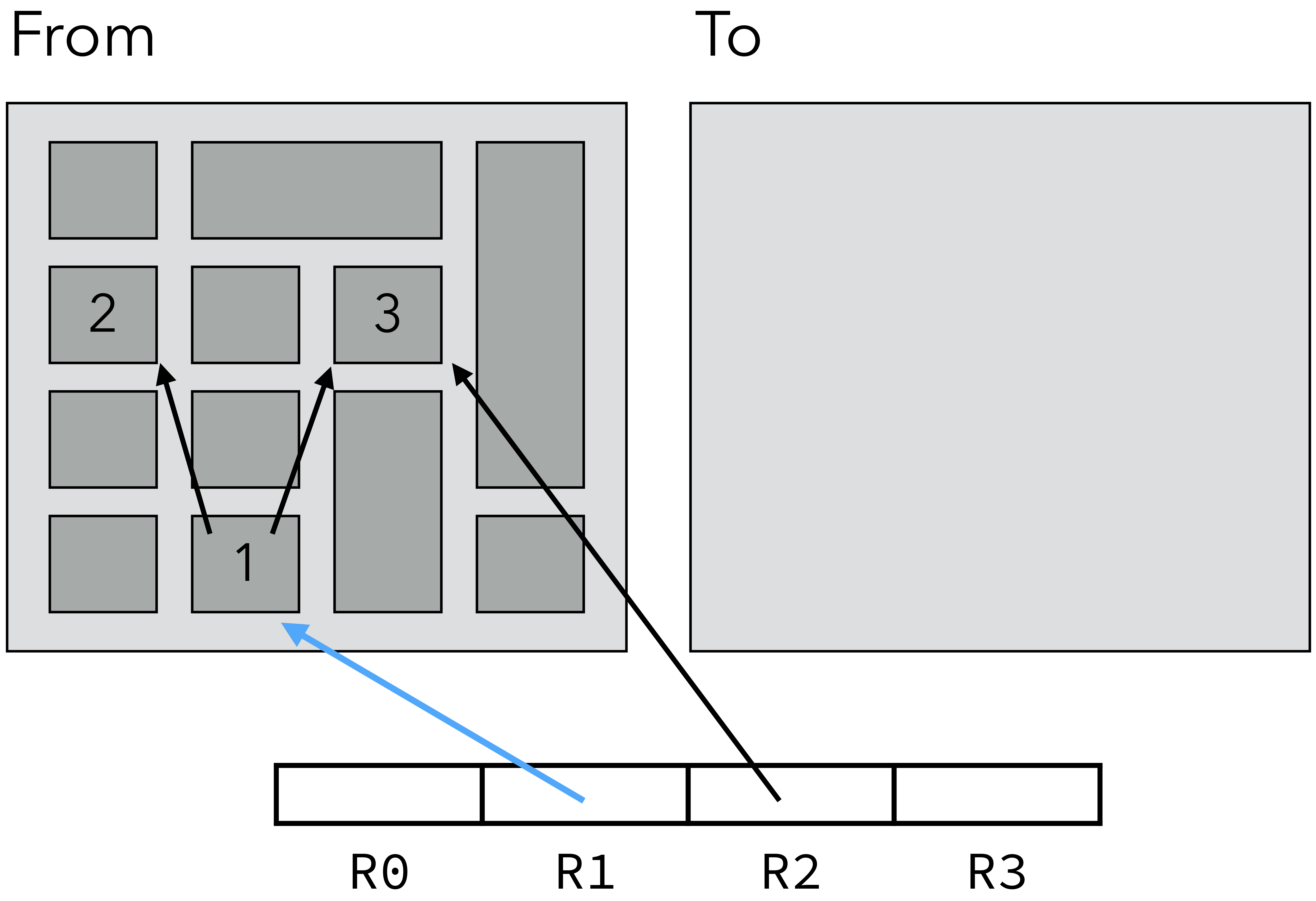
Copying garbage collection works by:

- splitting the heap in two semi-spaces of equal size:
  1. the **from-space**, and
  2. the **to-space**,
- allocating memory from from-space only,
- when from-space is full:
  - copying all reachable object to to-space,
  - updating pointers accordingly,
  - exchanging the role of the two spaces.

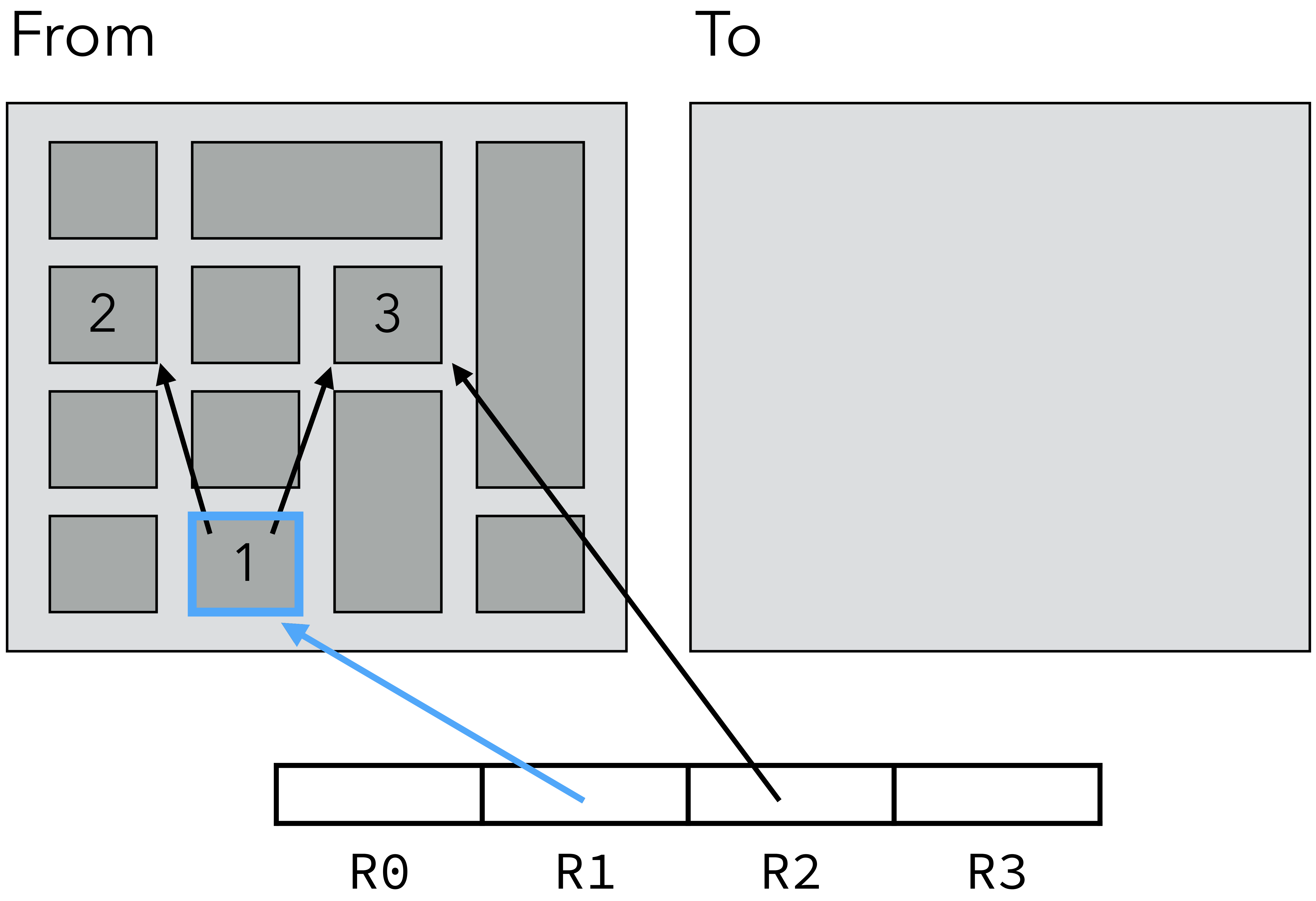
Copying GC



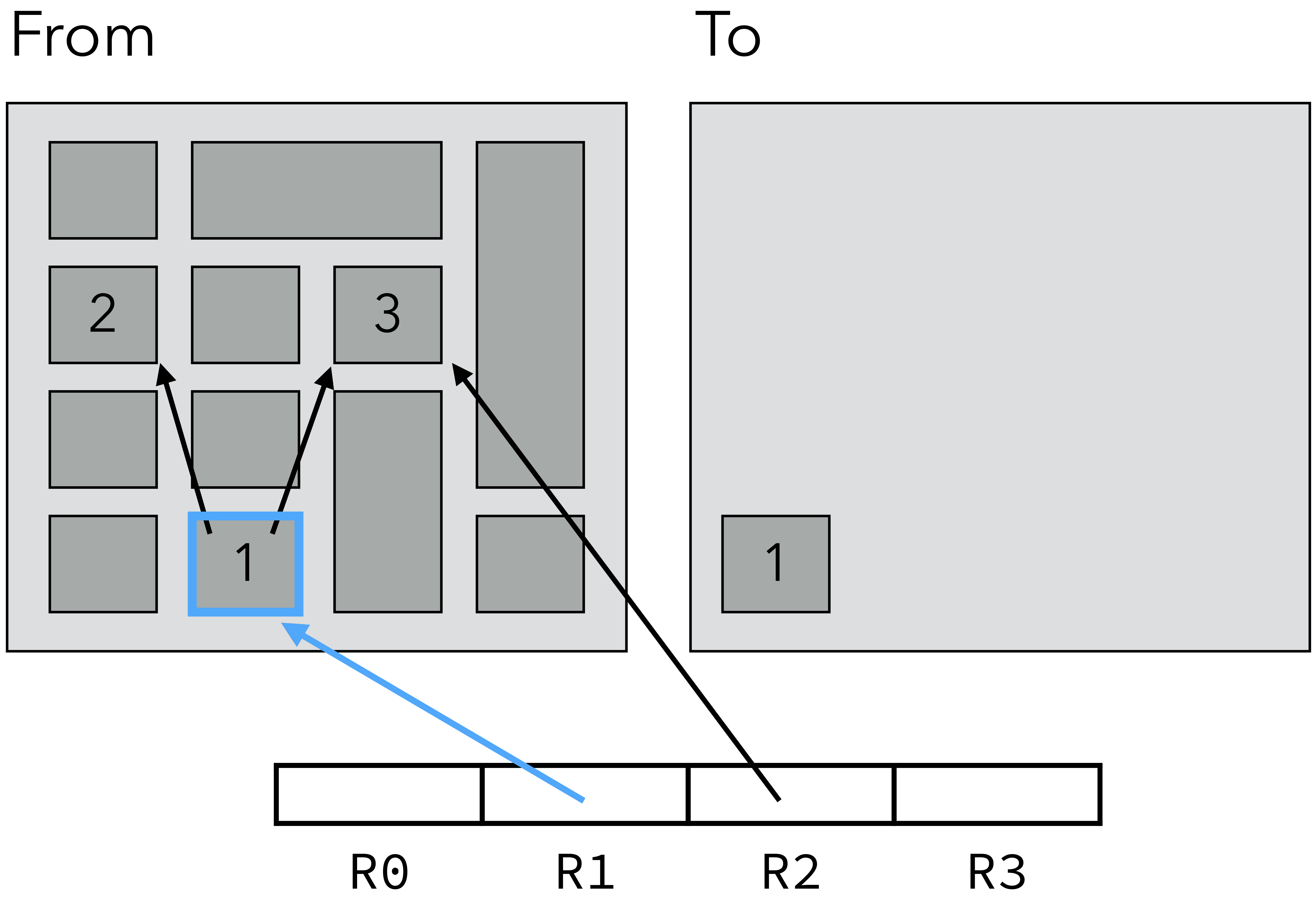
Copying GC



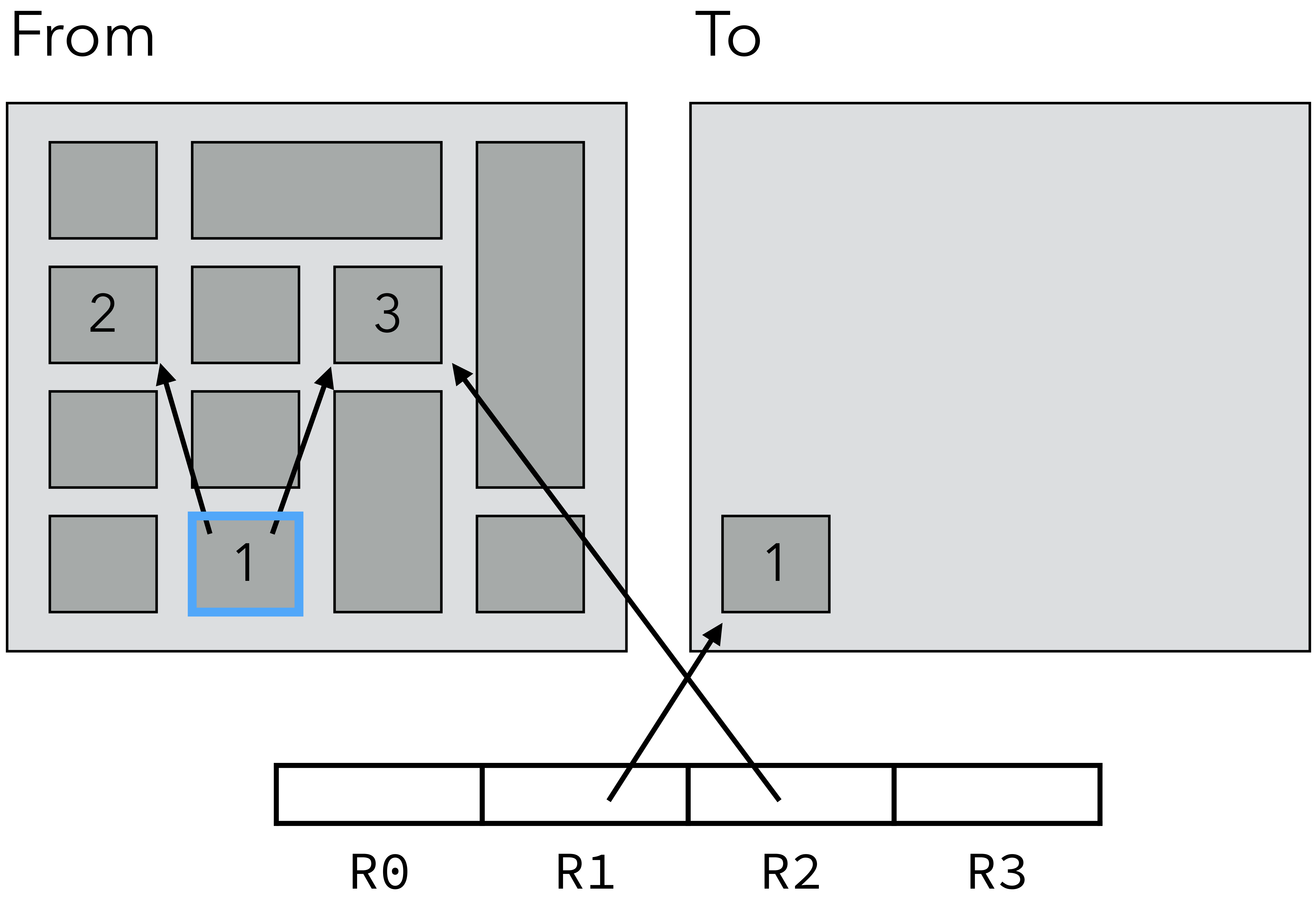
Copying GC



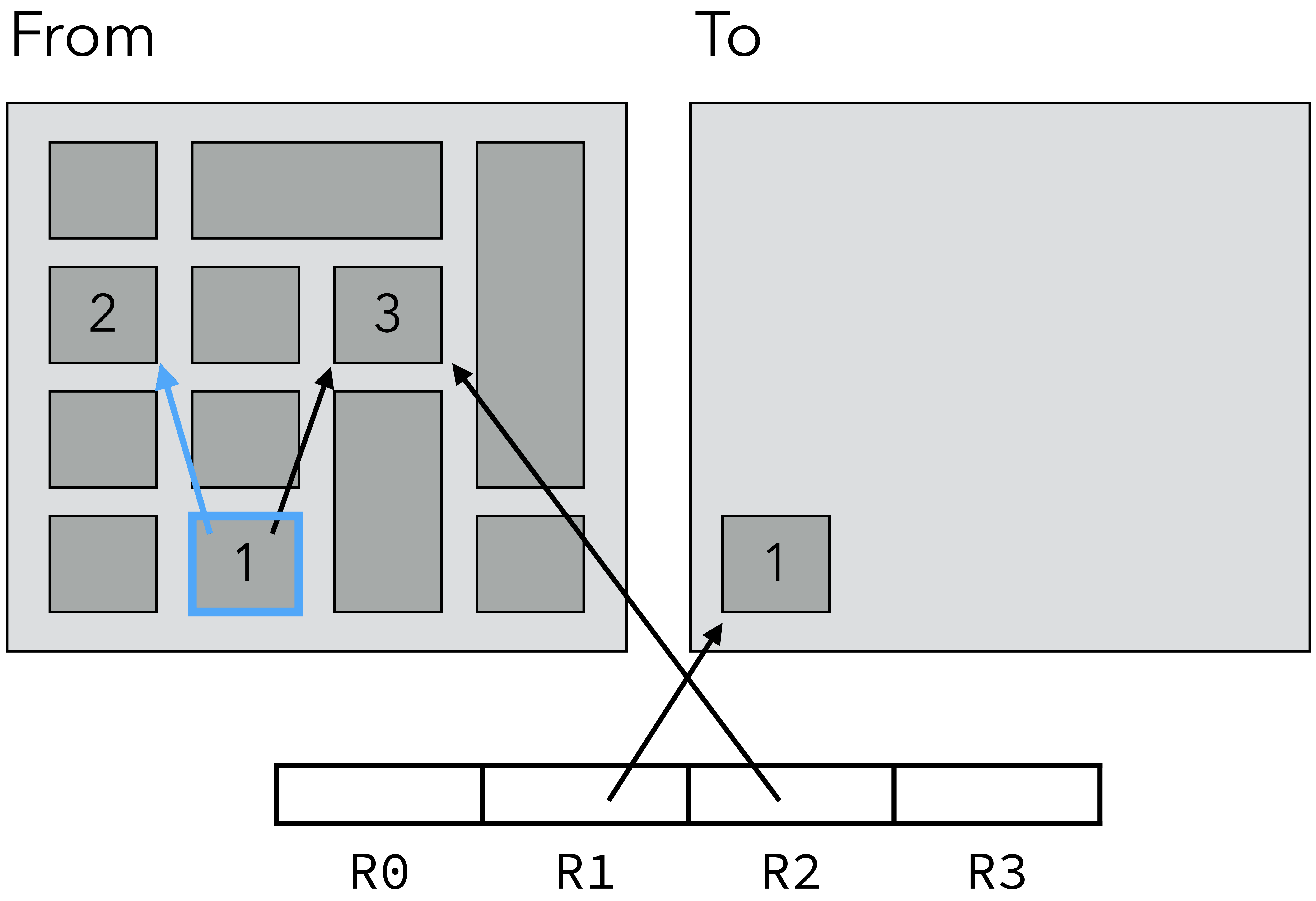
Copying GC



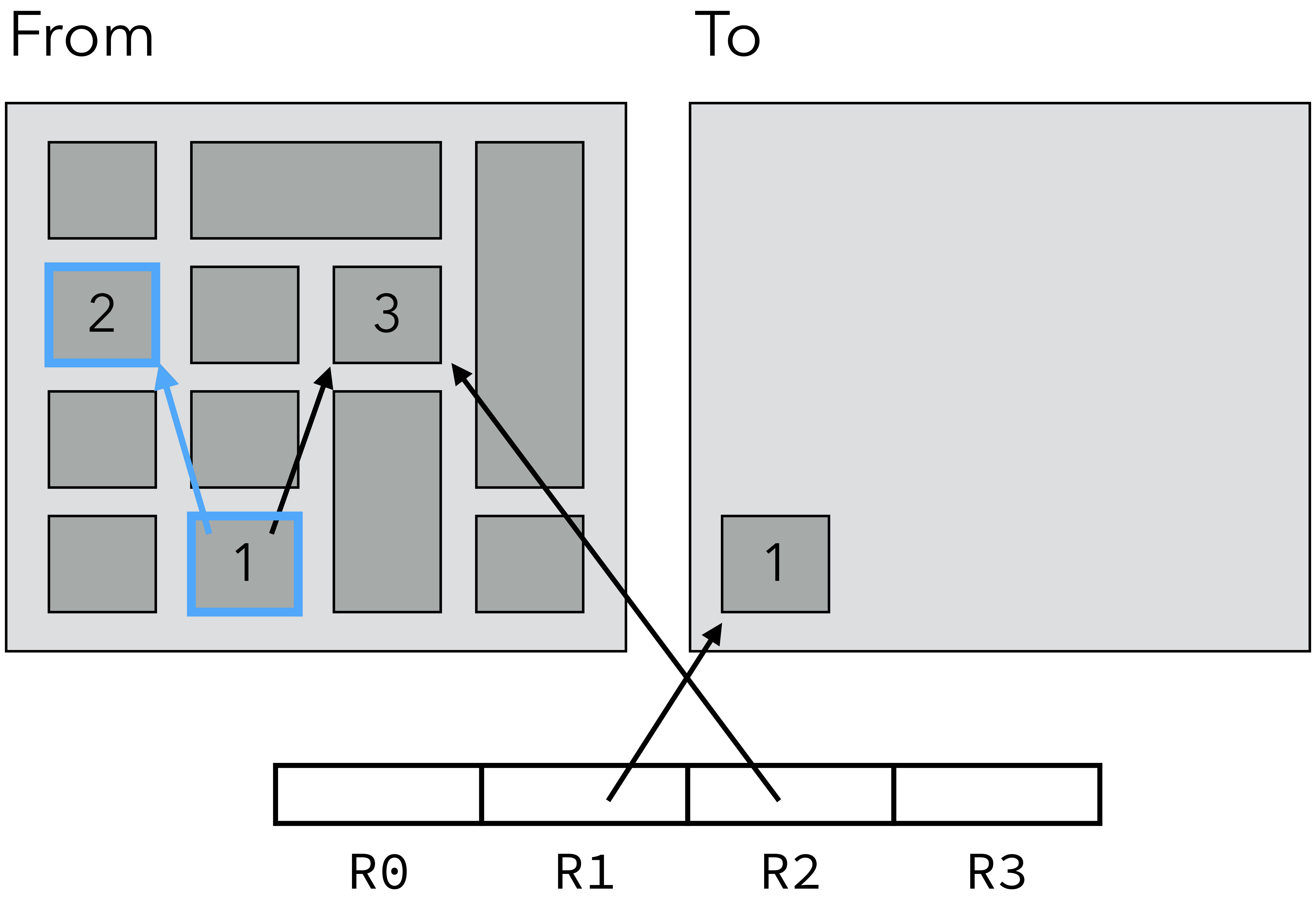
Copying GC



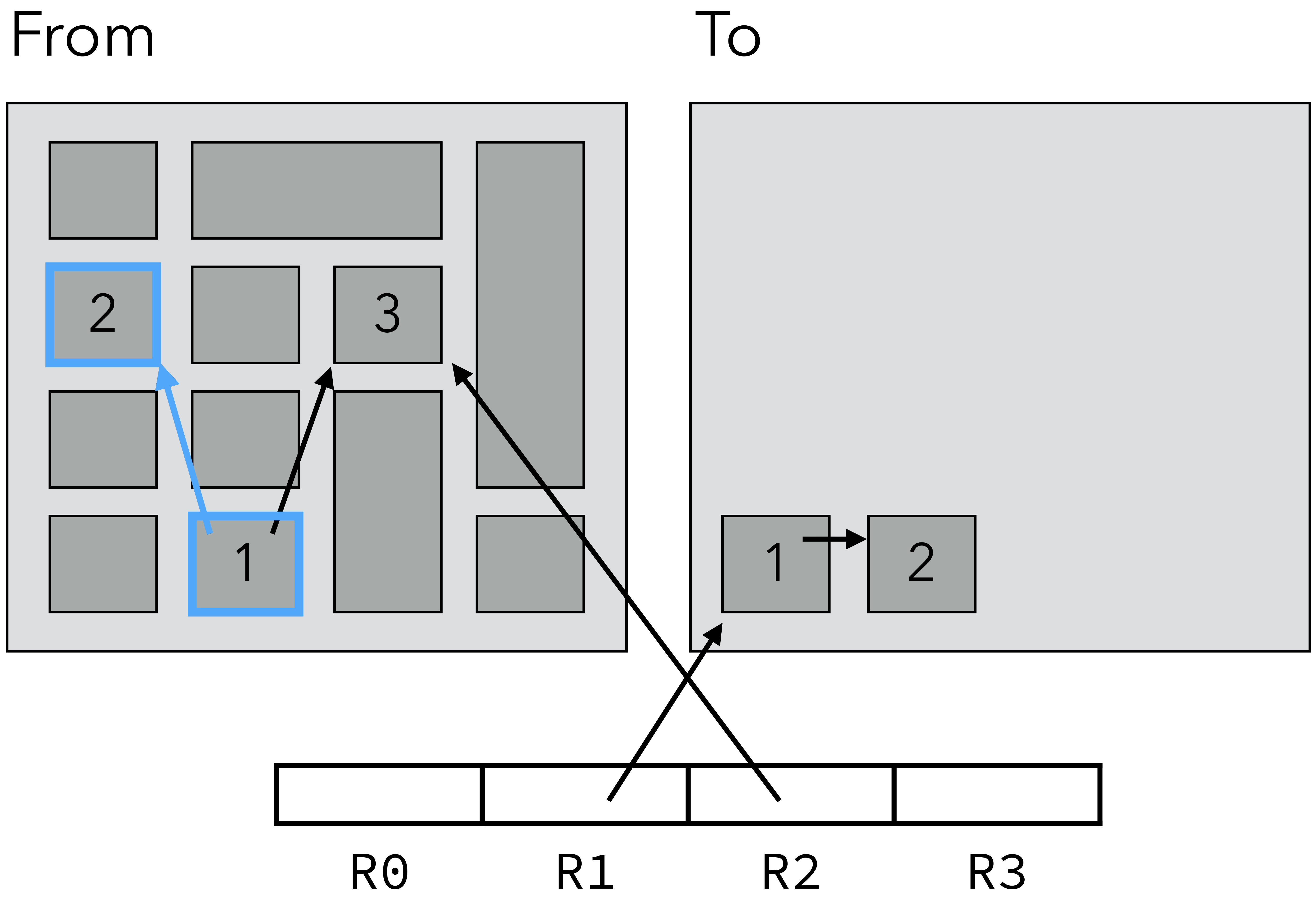
Copying GC



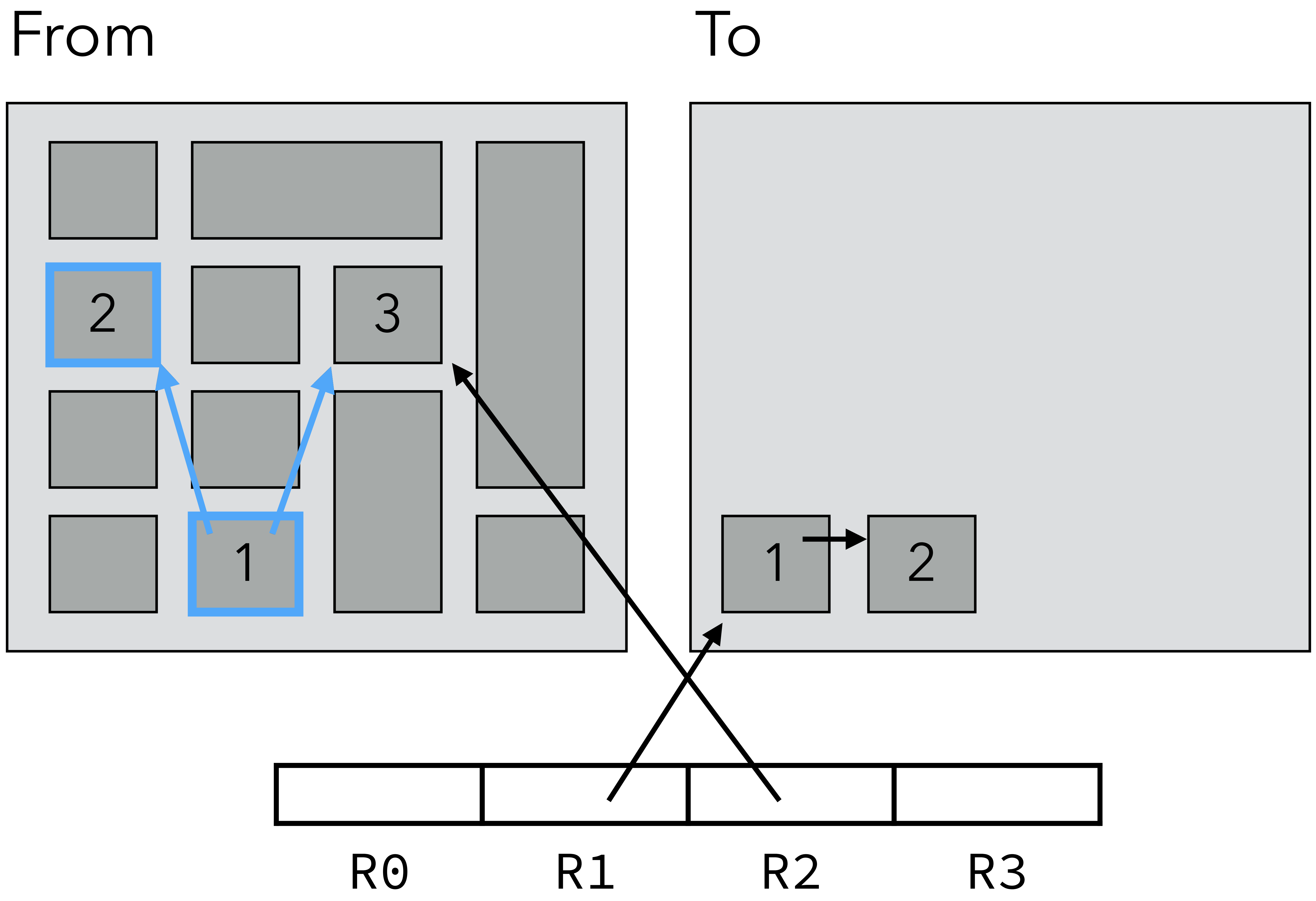
Copying GC



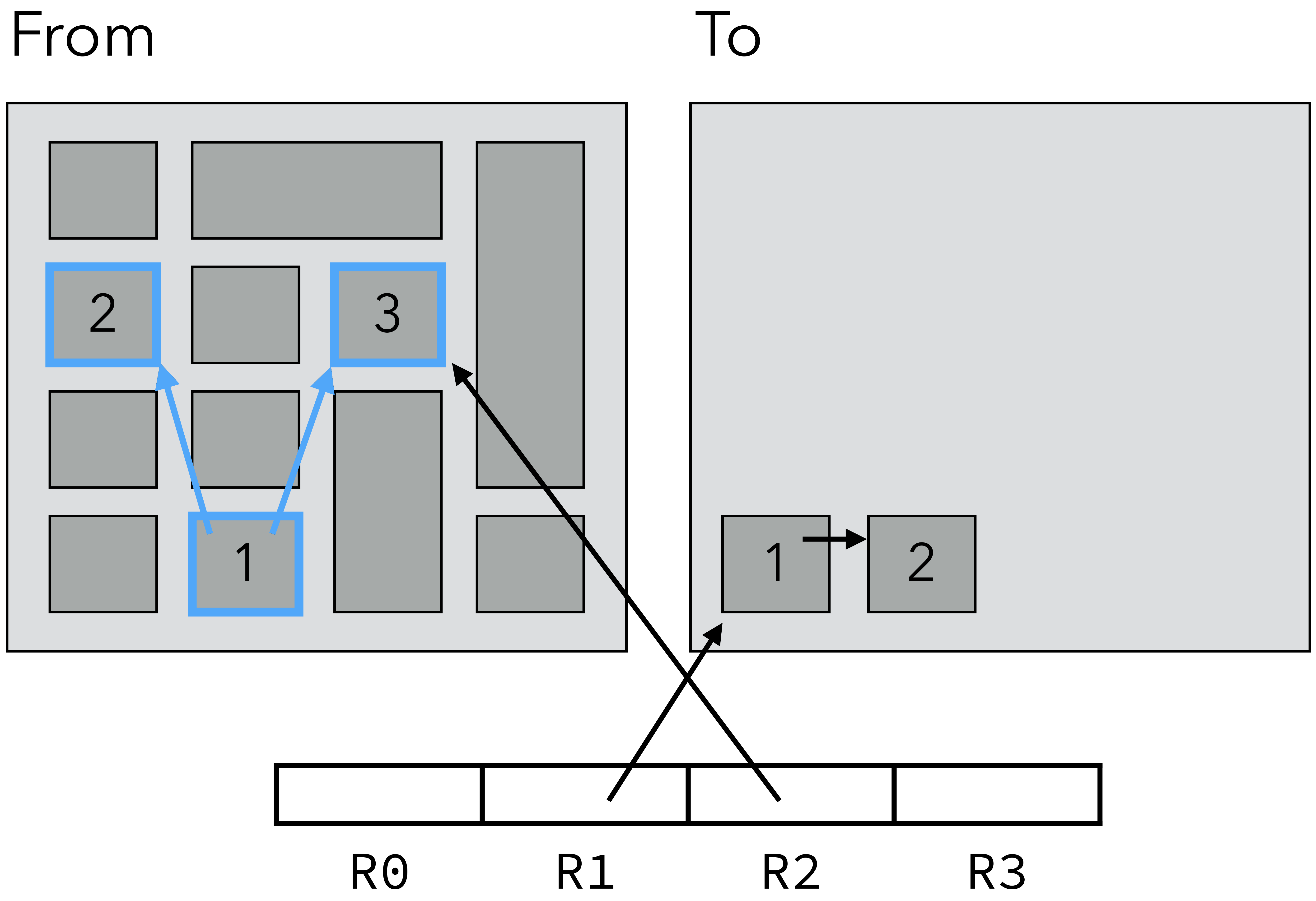
Copying GC



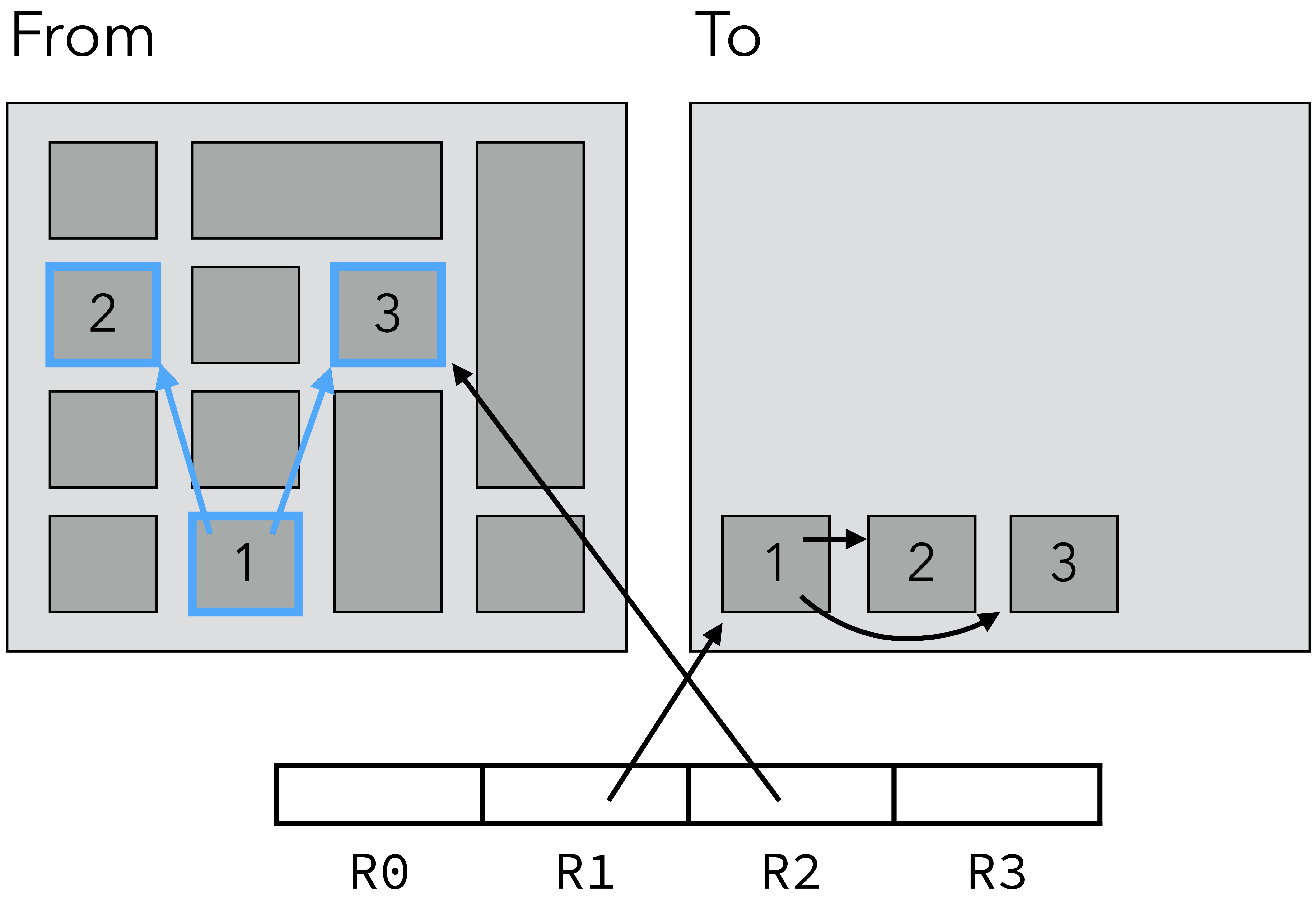
Copying GC



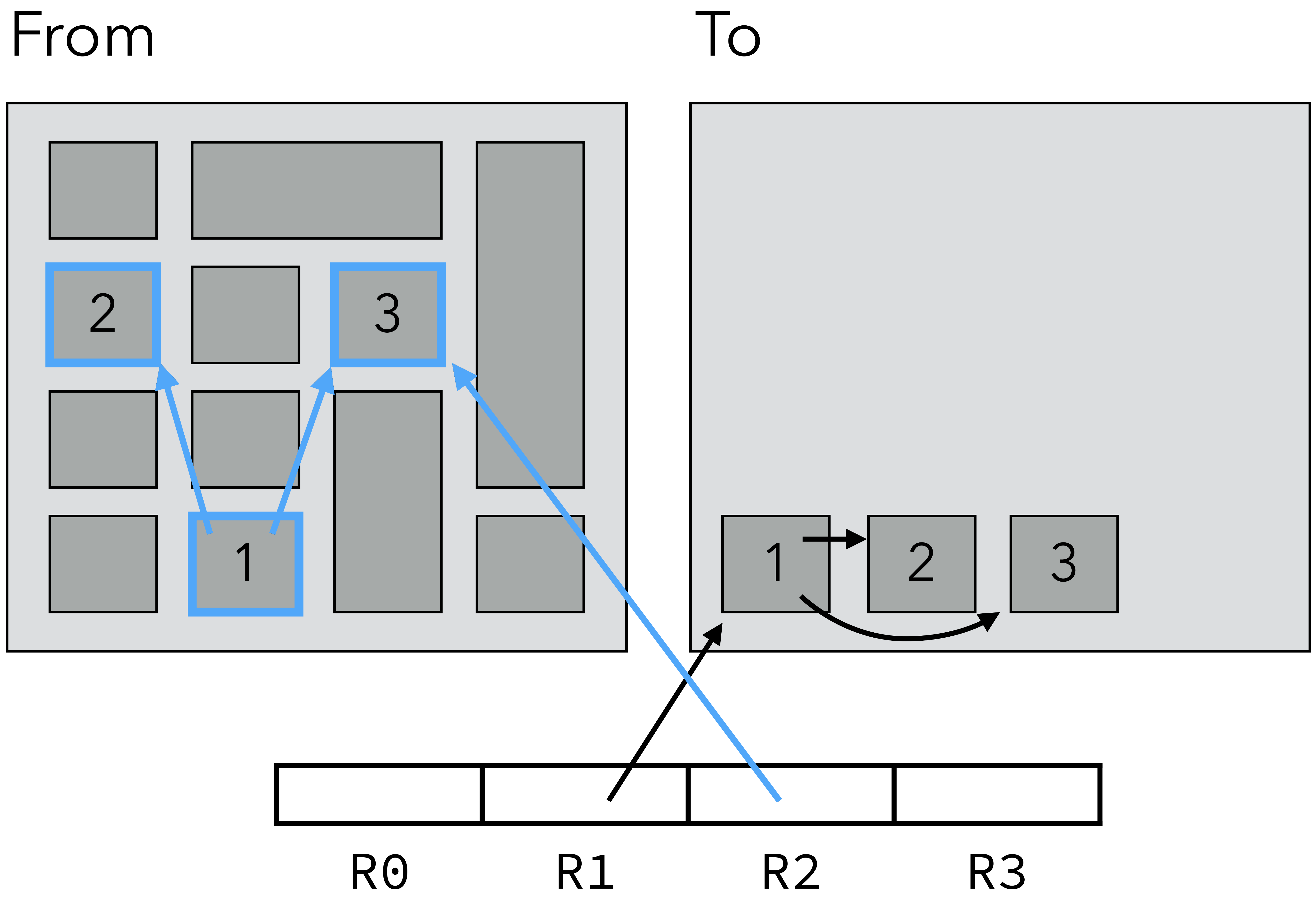
Copying GC



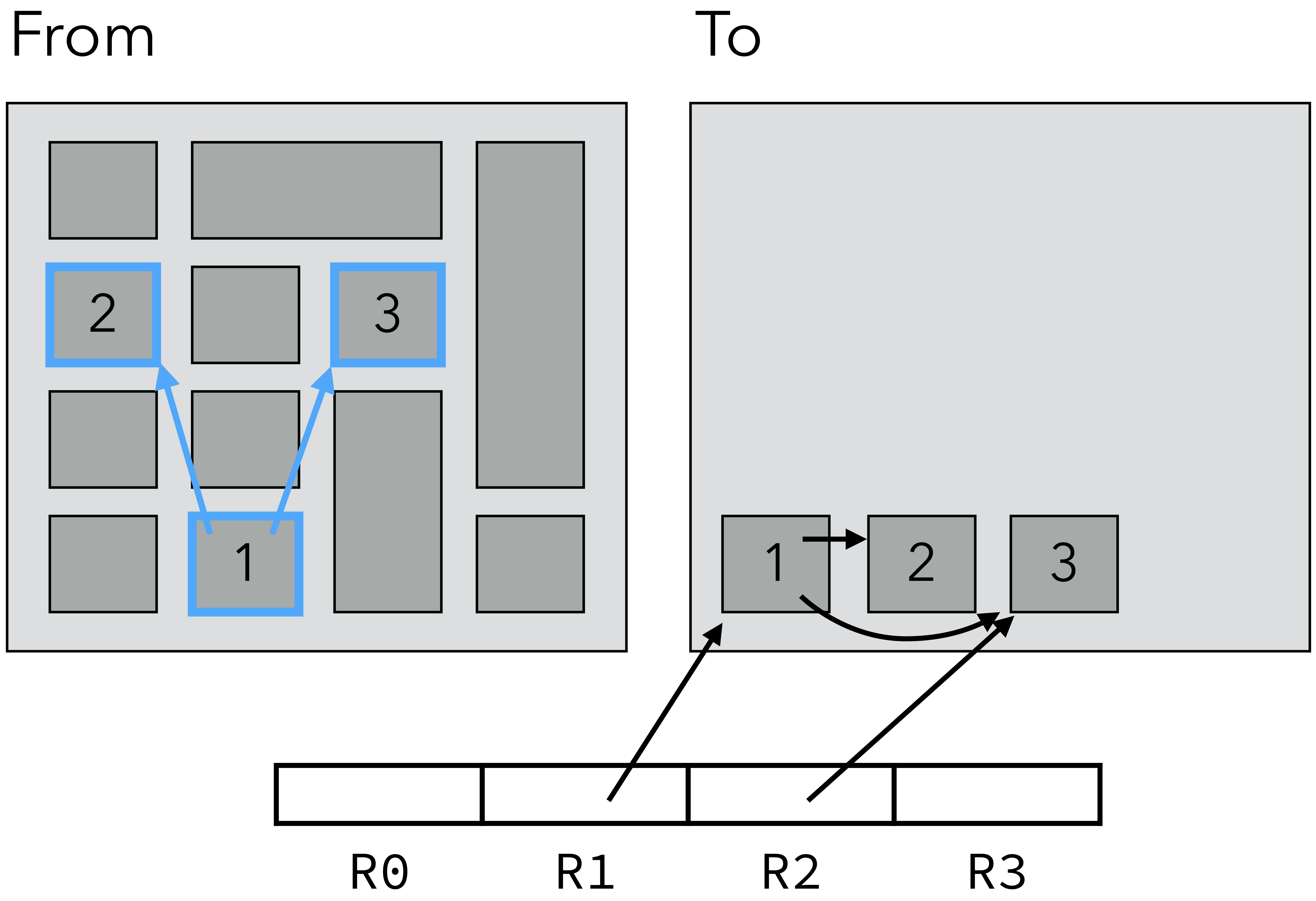
Copying GC



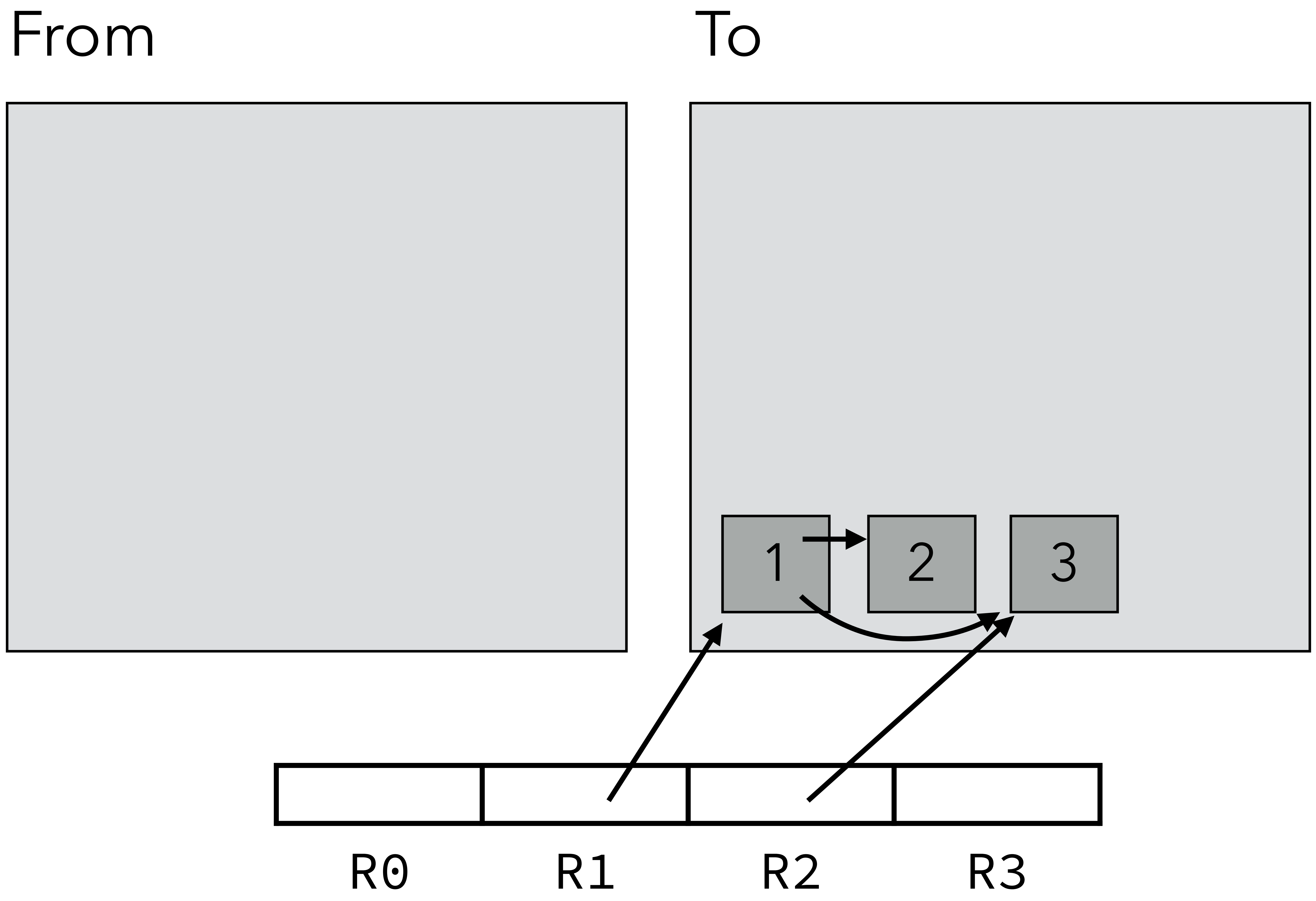
Copying GC



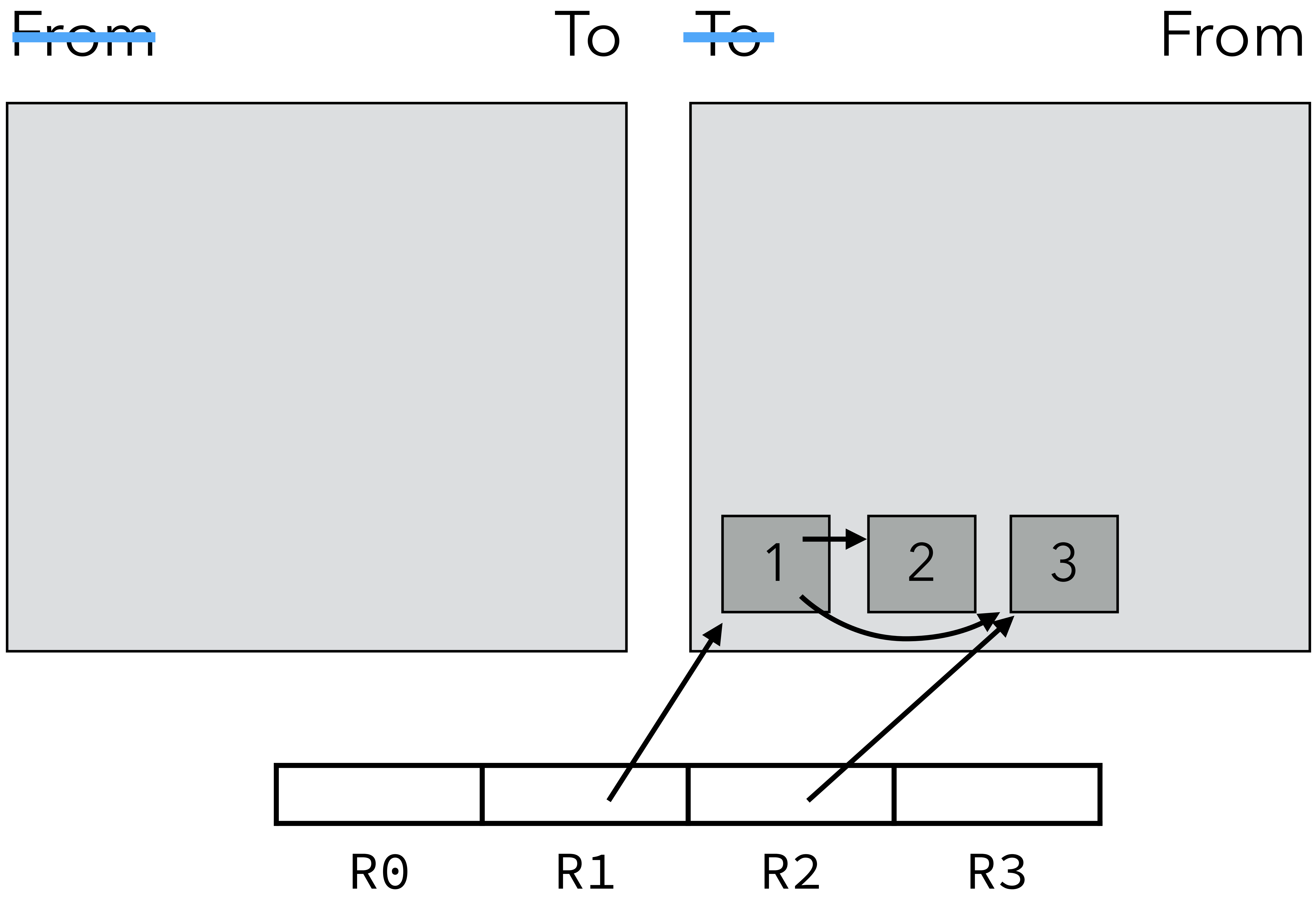
Copying GC



Copying GC



Copying GC



Allocation in a copying GC

In a copying GC, memory is allocated linearly in from-space:

- no free list to maintain,
- no search to perform to find a free block,
- no allocation policy.

All that is required is a pointer to the border between the allocated and free area of from-space.

Therefore: allocation in a copying GC is as fast as stack allocation.

Forwarding pointers

Objects must be copied to to-space only once! This is obtained by:

- storing a **forwarding pointer** in the from-space version of the object once it has been copied,
- checking for the presence of a forwarding pointer when visiting an object and:
  - copying it if no forwarding pointer is found,
  - using the forwarding pointer otherwise.

Cheney's copying GC

Copying can be done by depth-first traversal of the reachability graph, but this can lead to stack overflow.

Cheney's copying GC does:

- a breadth-first traversal of the reachability graph,
- requires *only one pointer* as additional state.

Cheney's copying GC

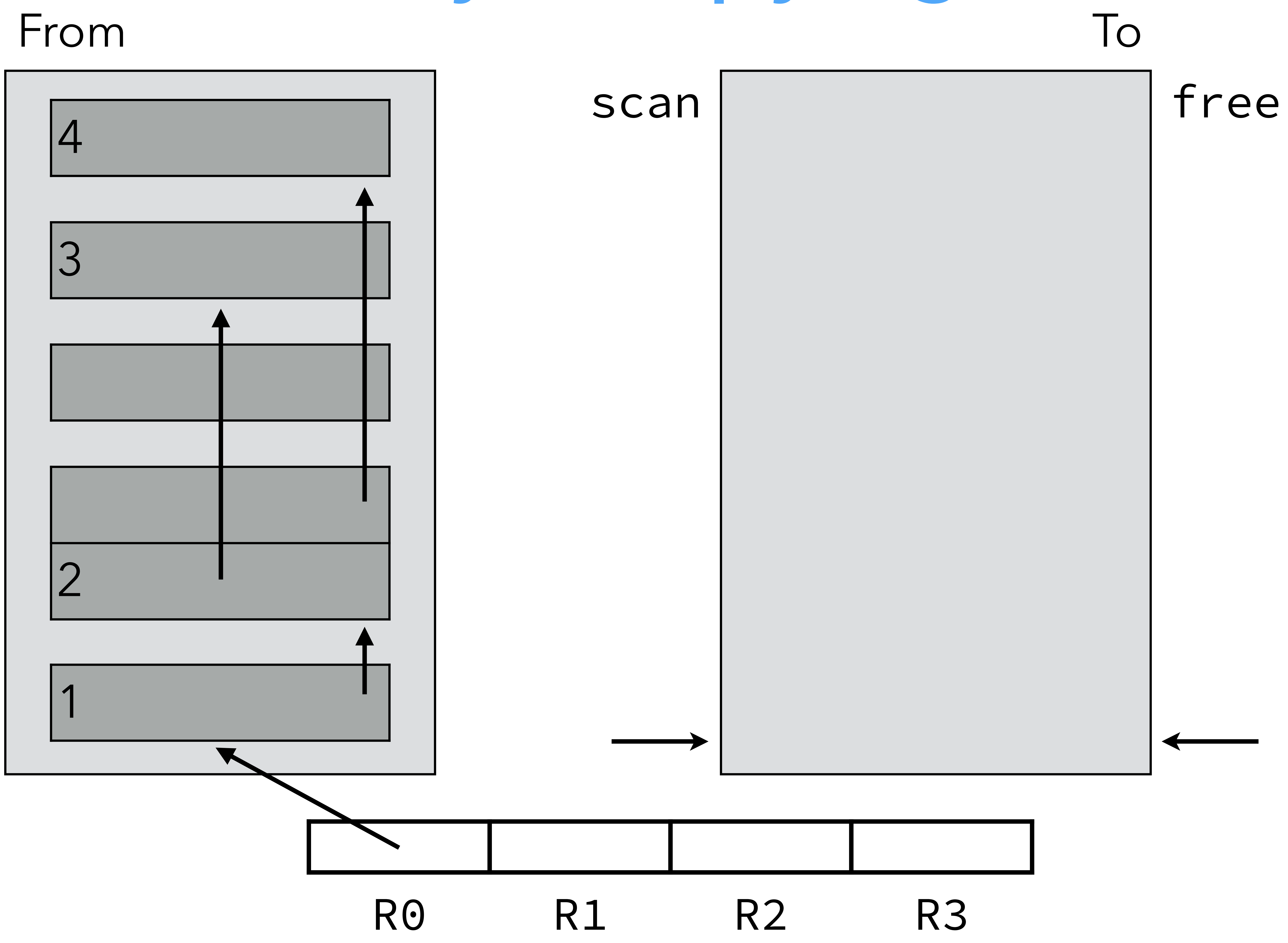
Breadth-first traversal requires remembering the set of objects that:

- have been visited, but
- whose children haven't been visited.

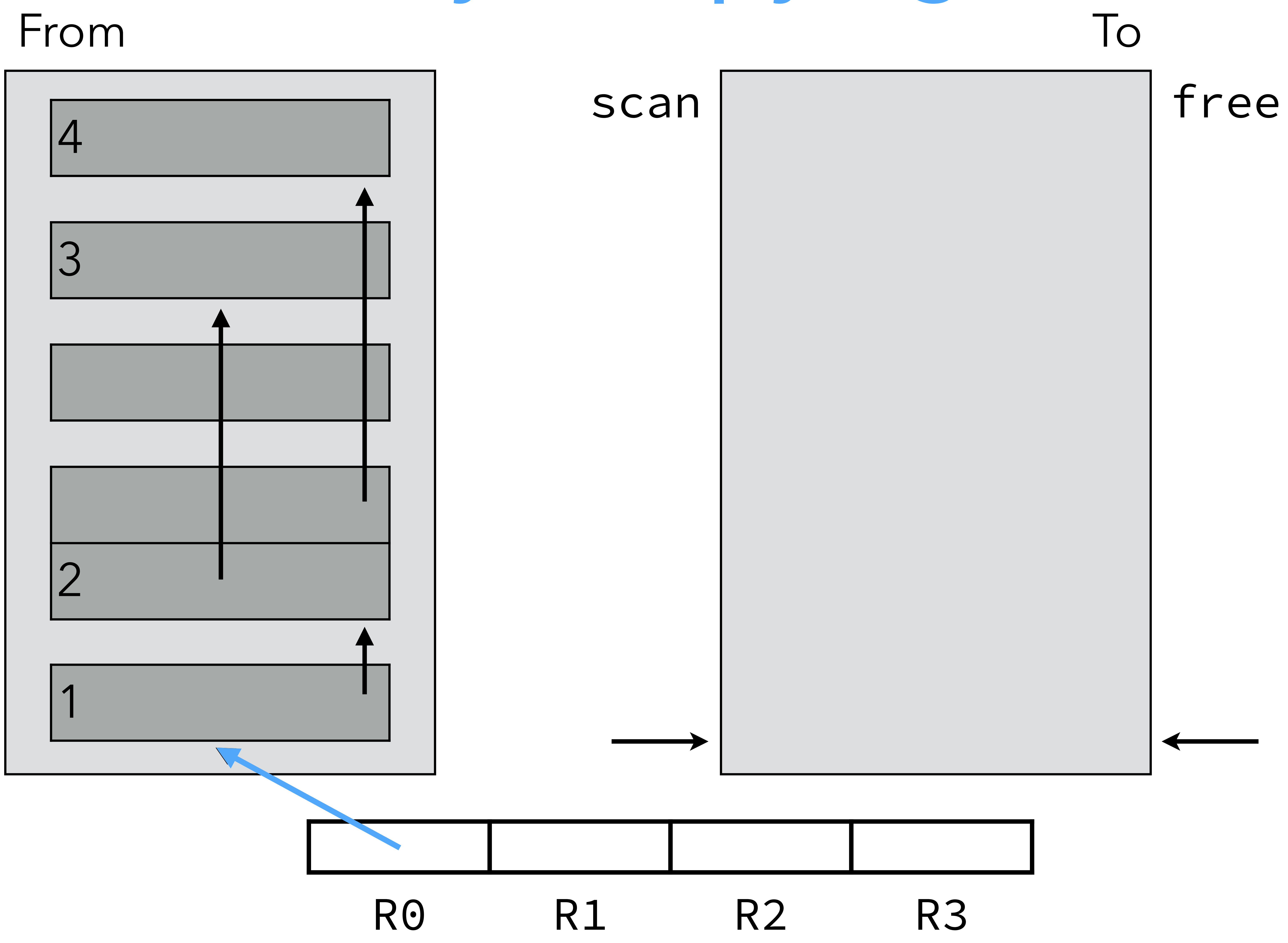
Cheney's observation:

This set can be represented as a pointer into to-space (called `scan`) that partitions pointers to objects that have been visited and pointers to objects that haven't been visited.

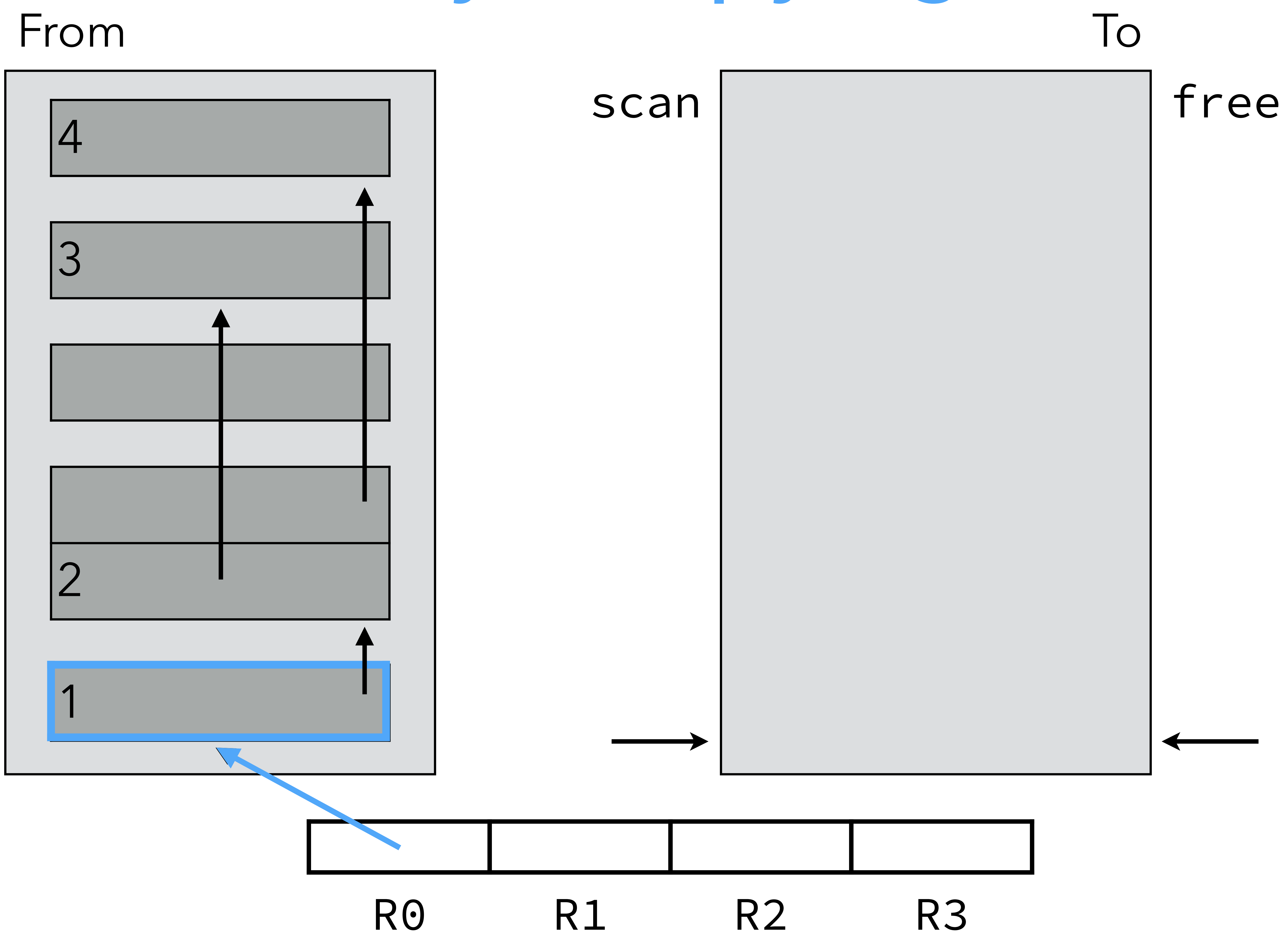
Cheney's copying GC



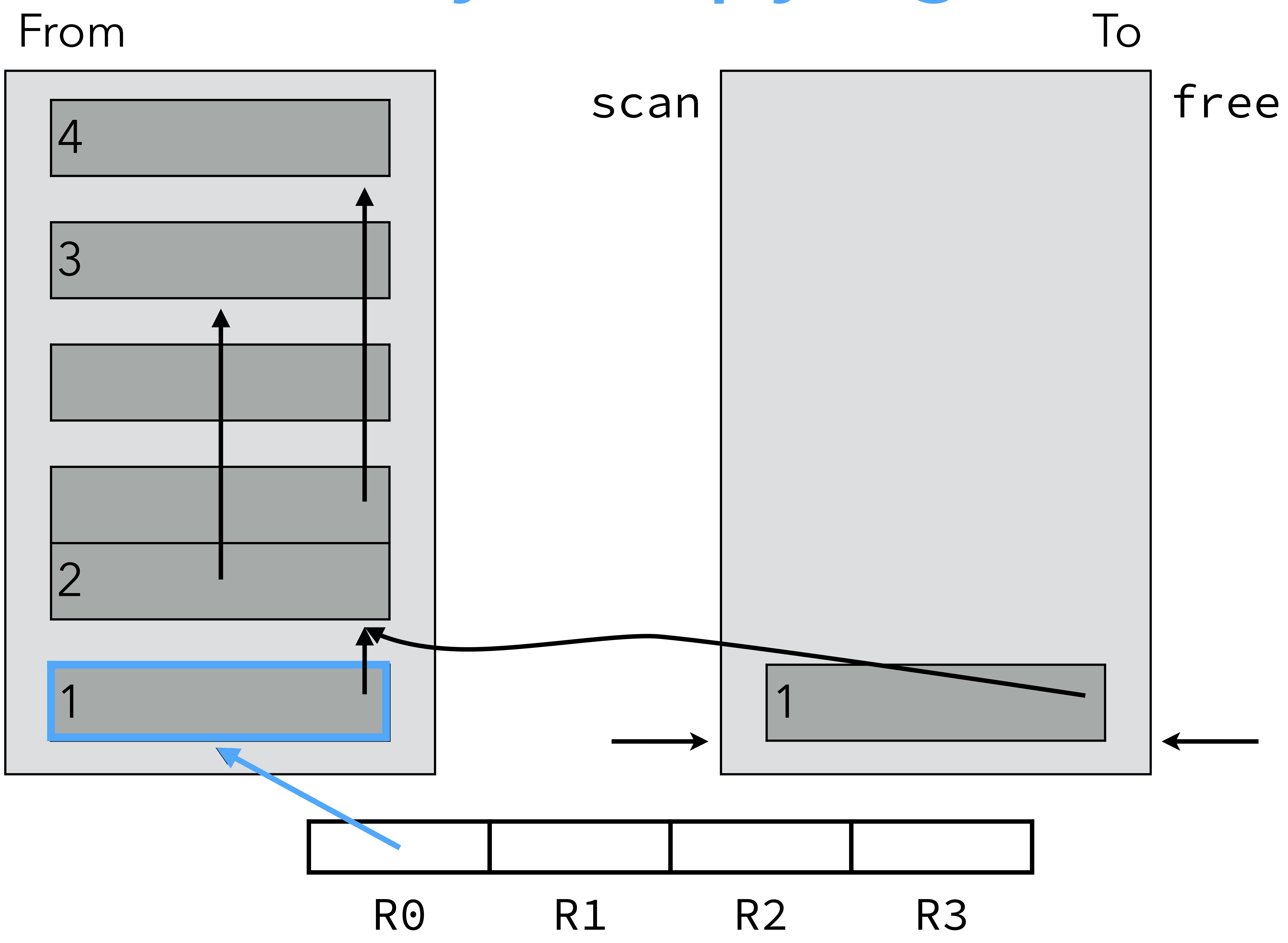
Cheney's copying GC



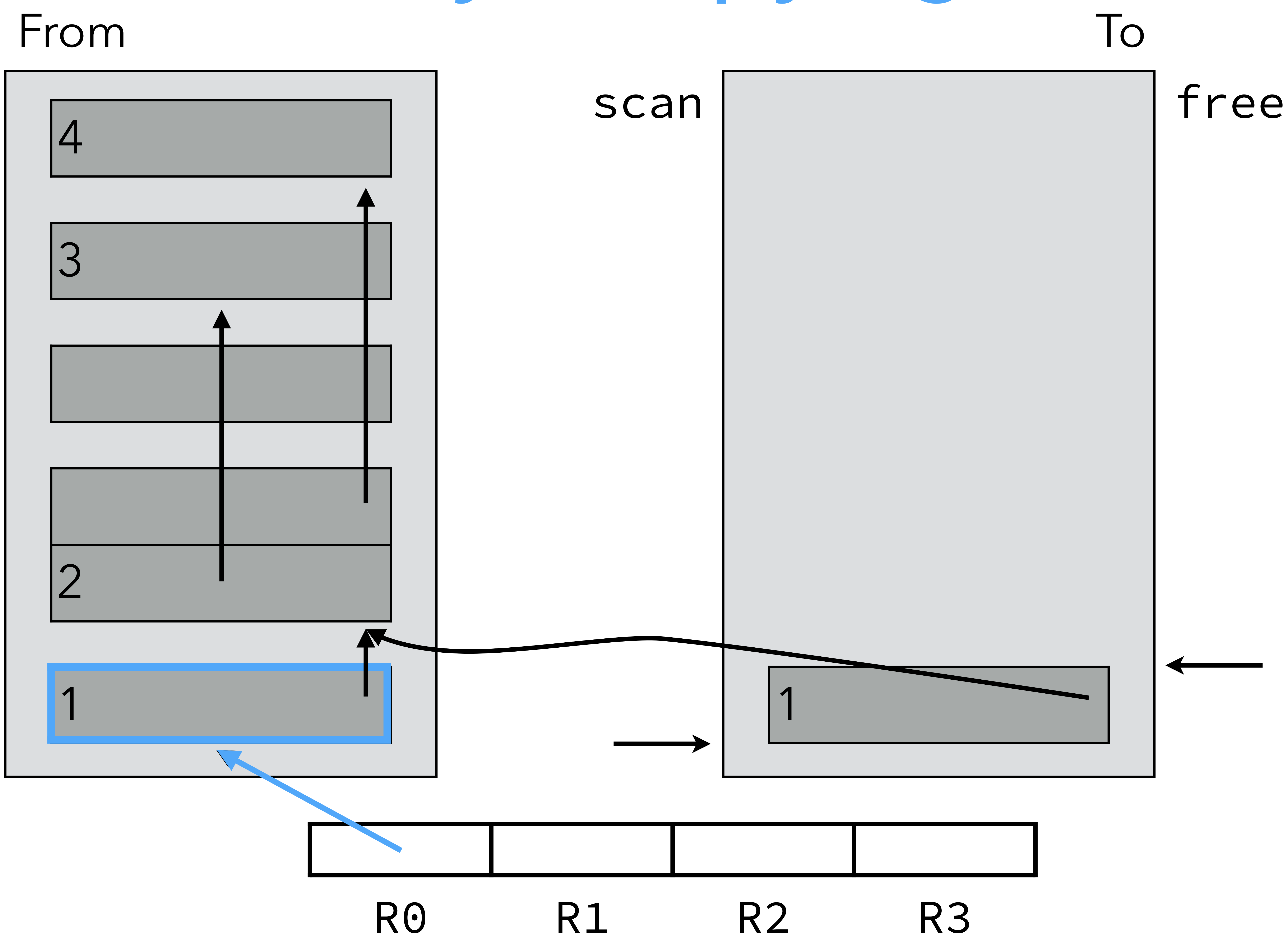
Cheney's copying GC



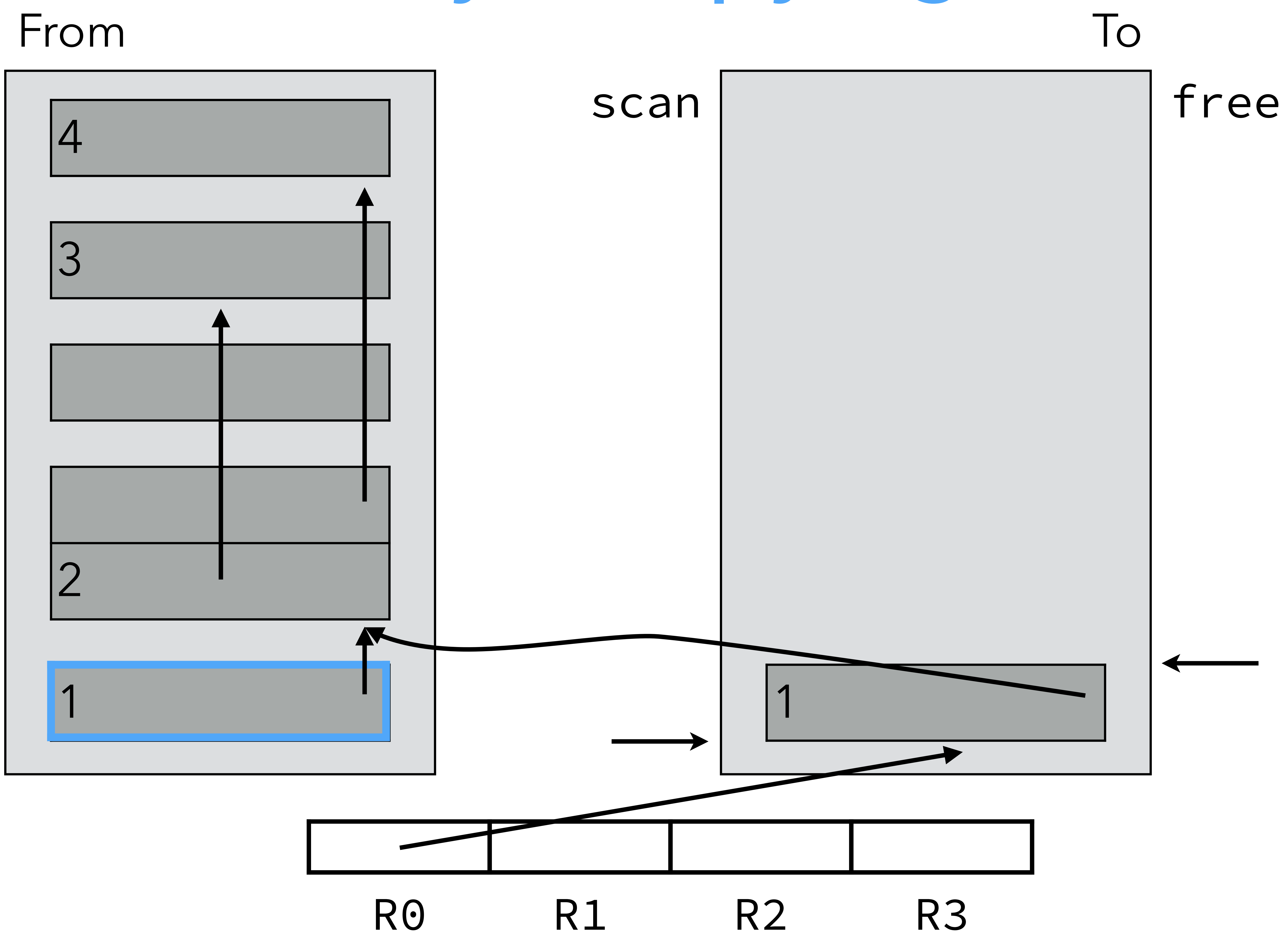
Cheney's copying GC



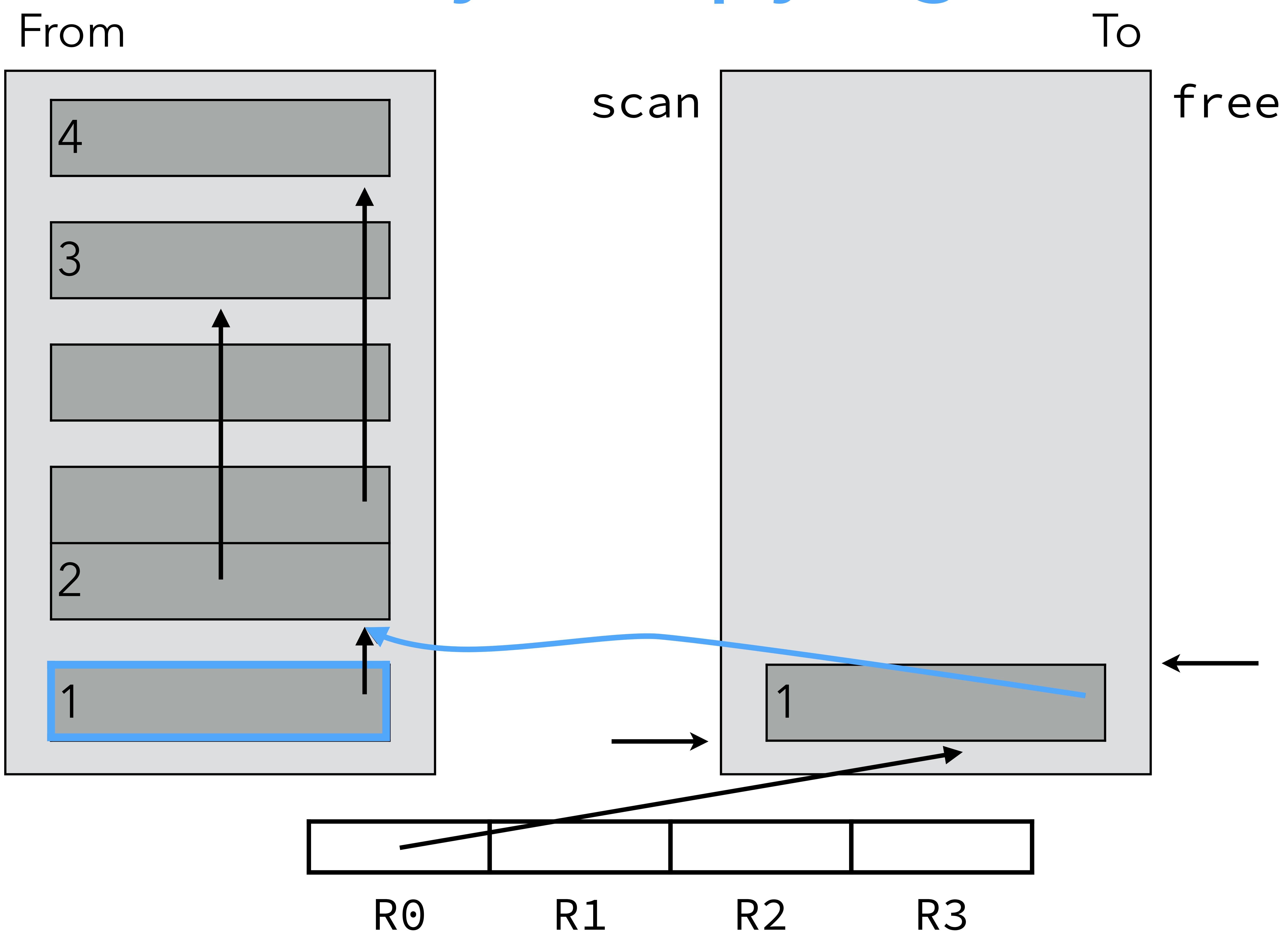
Cheney's copying GC



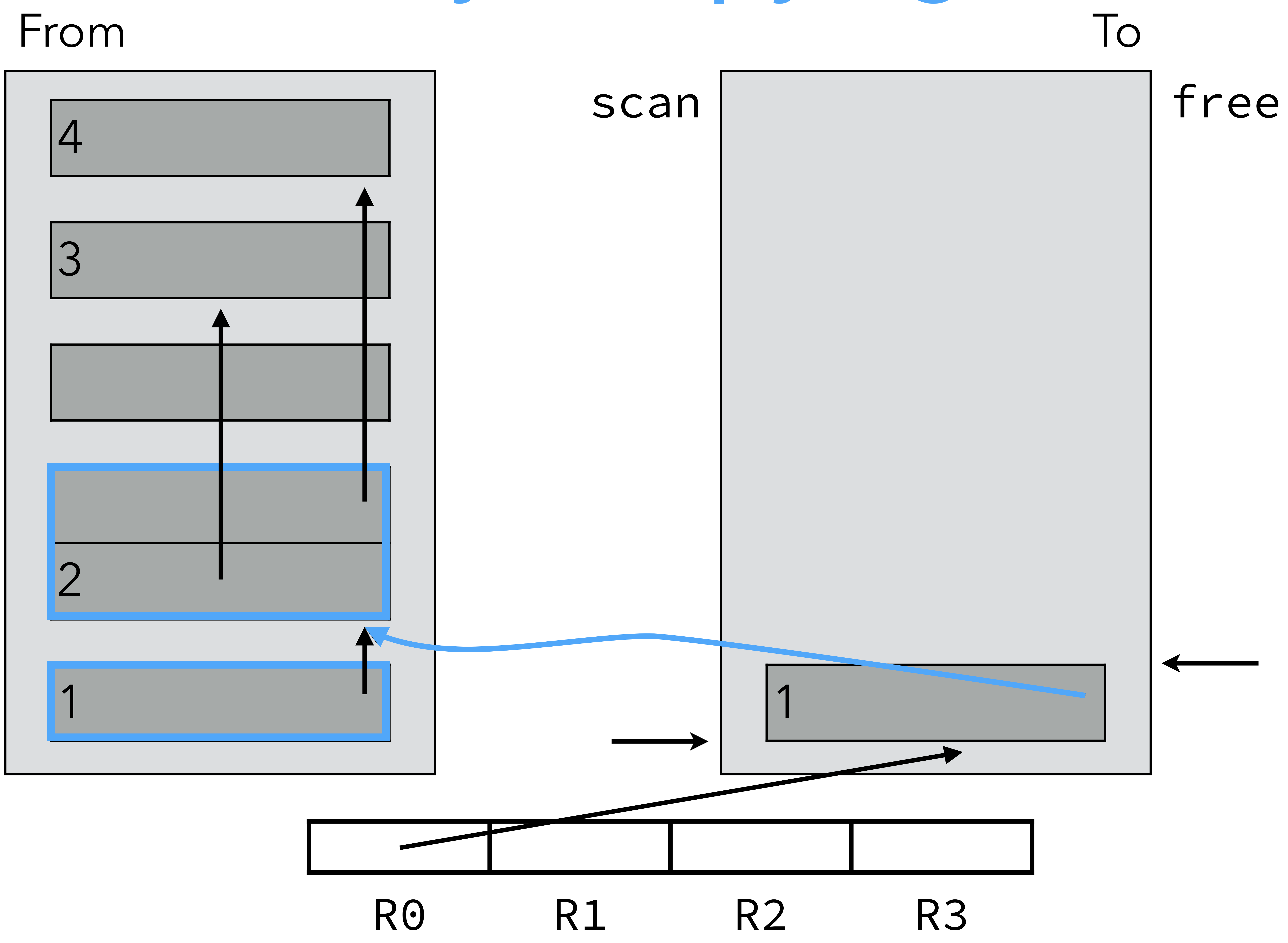
Cheney's copying GC



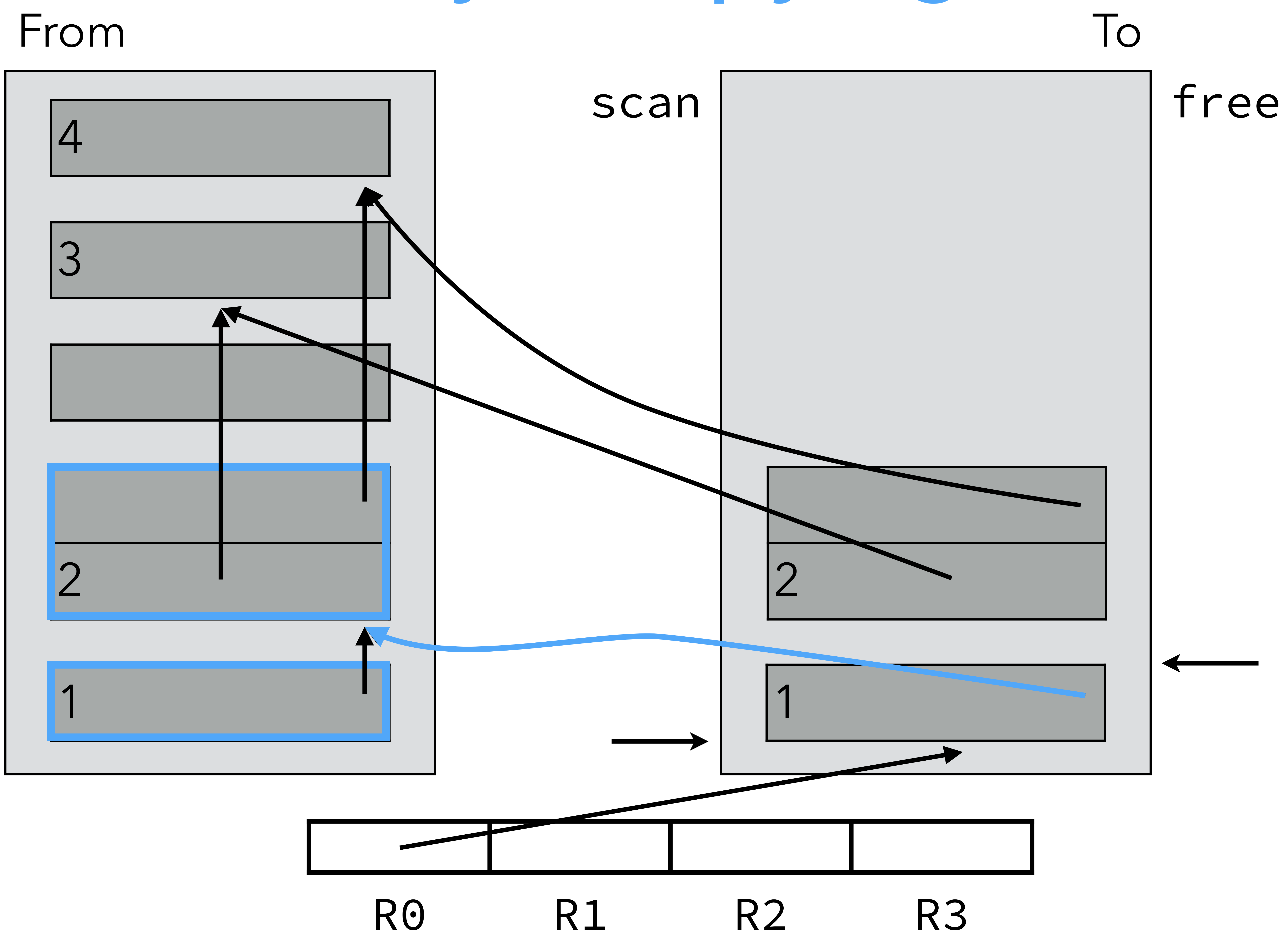
Cheney's copying GC



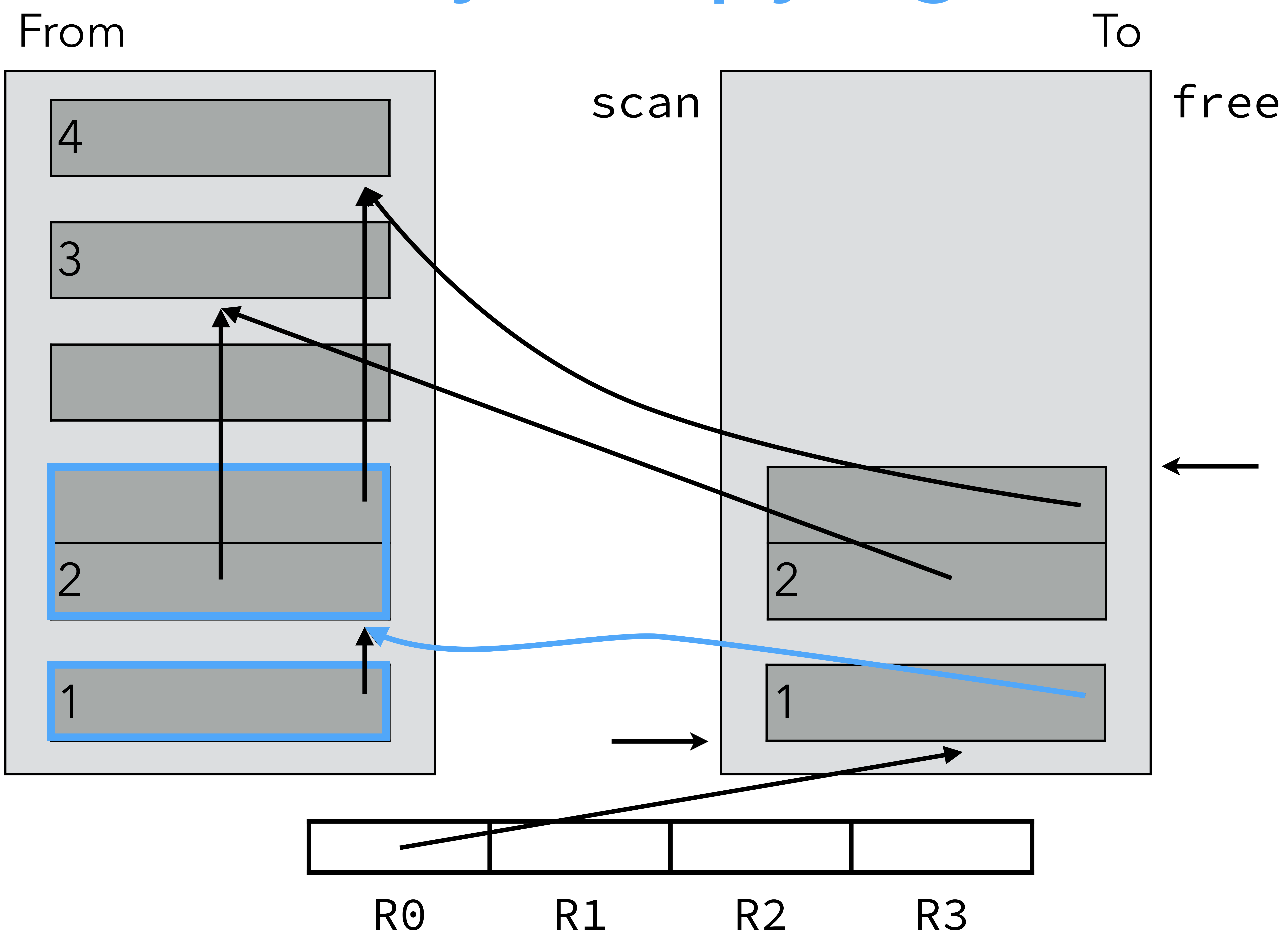
Cheney's copying GC



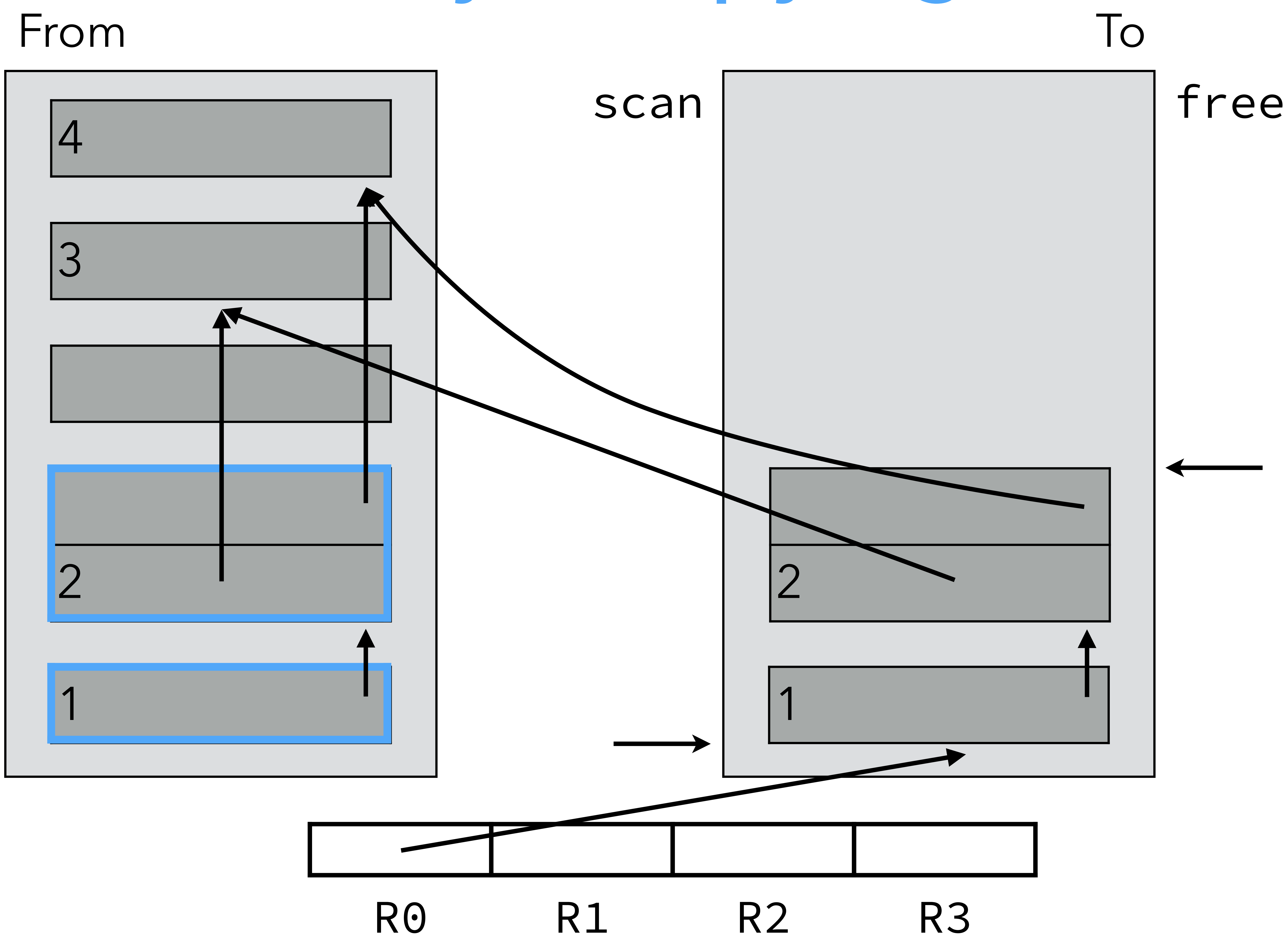
Cheney's copying GC



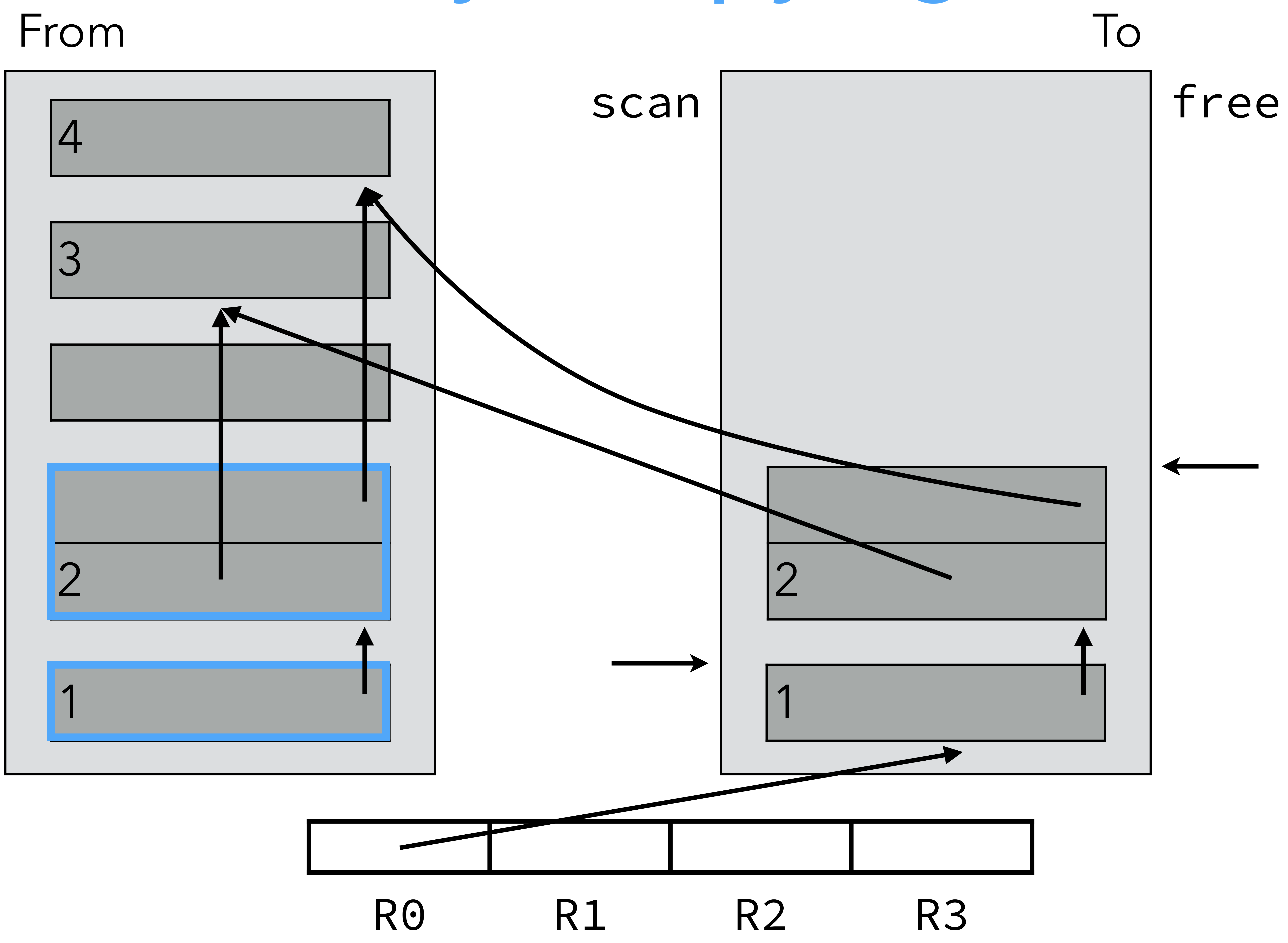
Cheney's copying GC



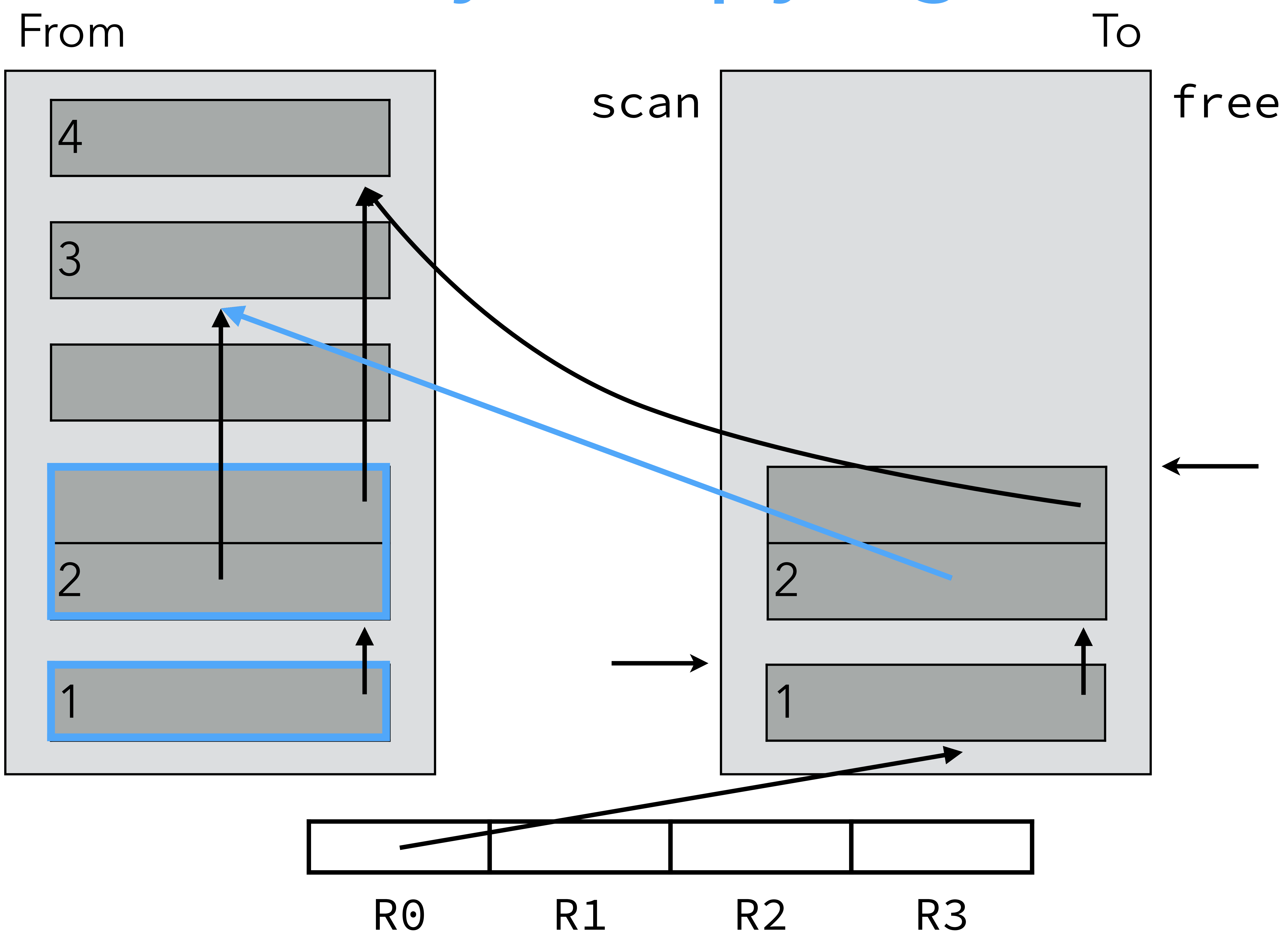
Cheney's copying GC



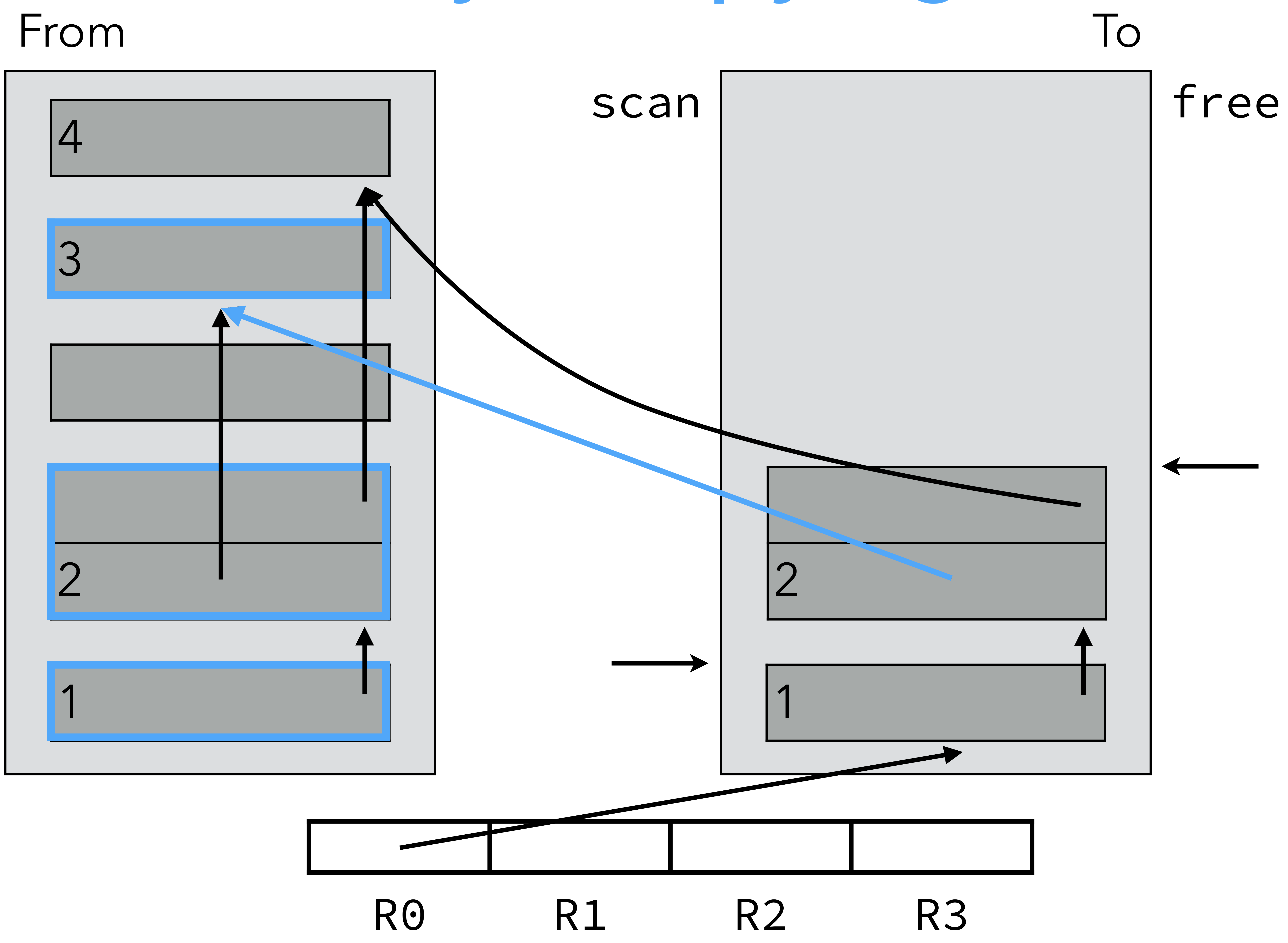
Cheney's copying GC



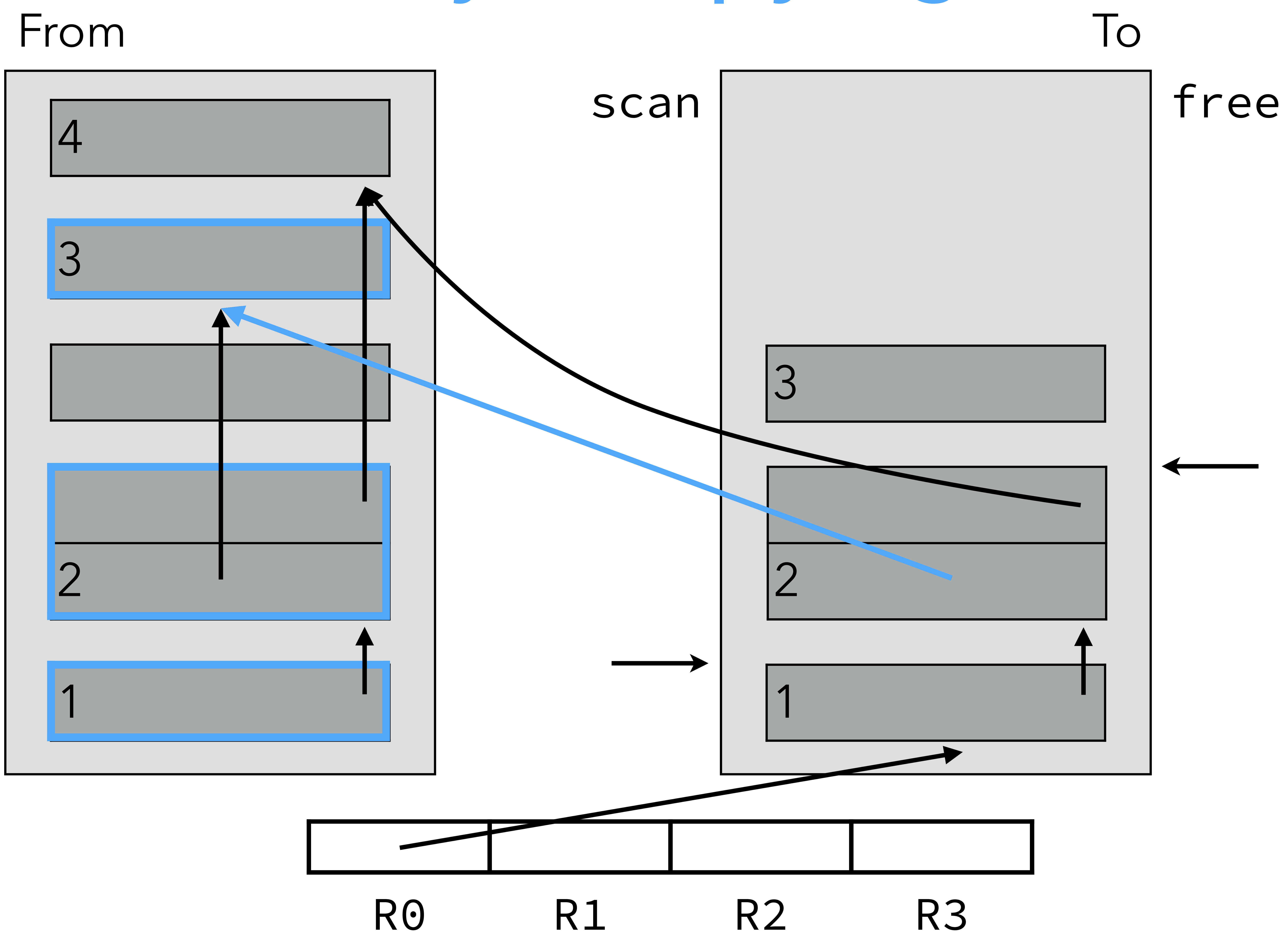
Cheney's copying GC



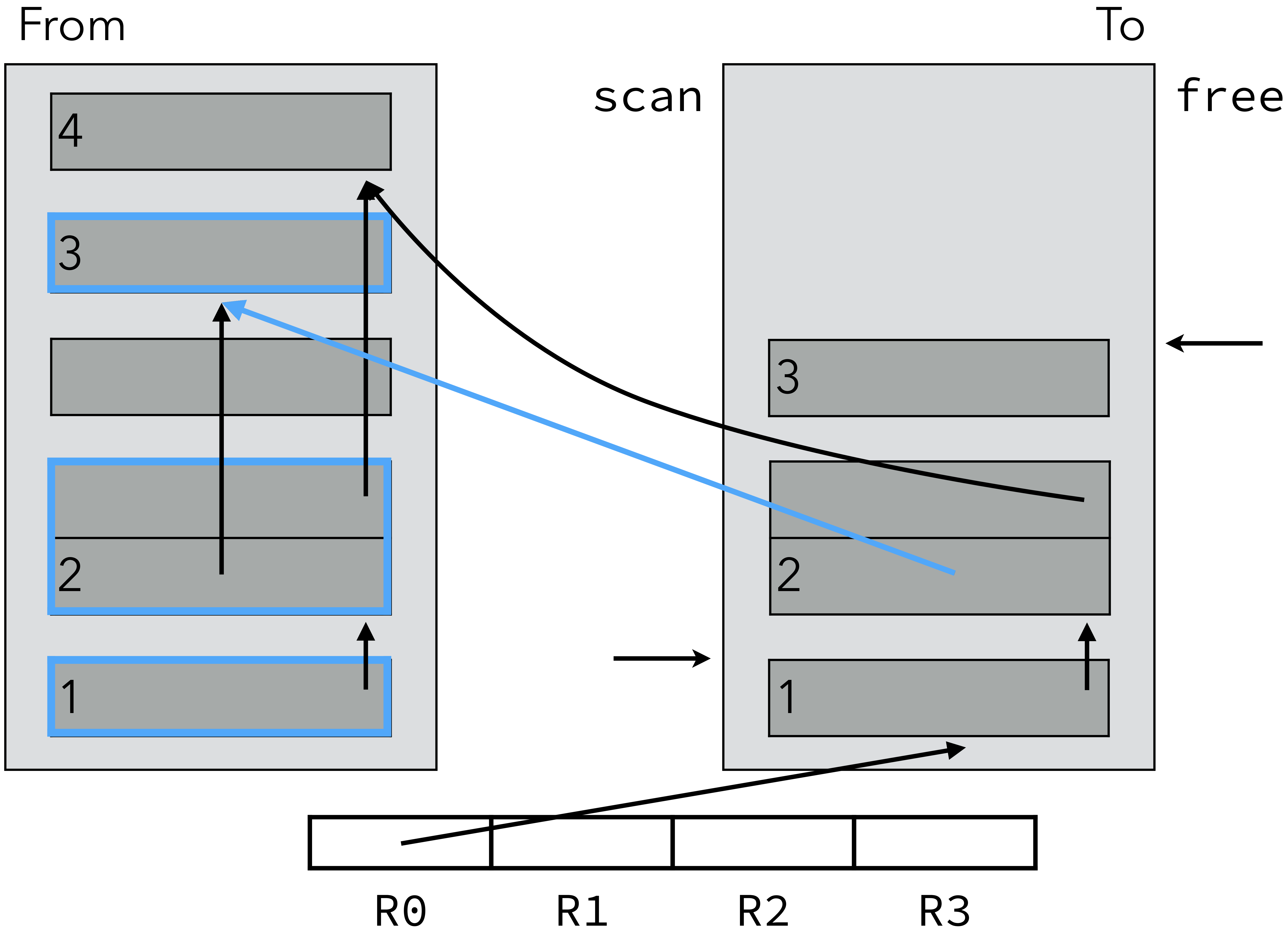
Cheney's copying GC



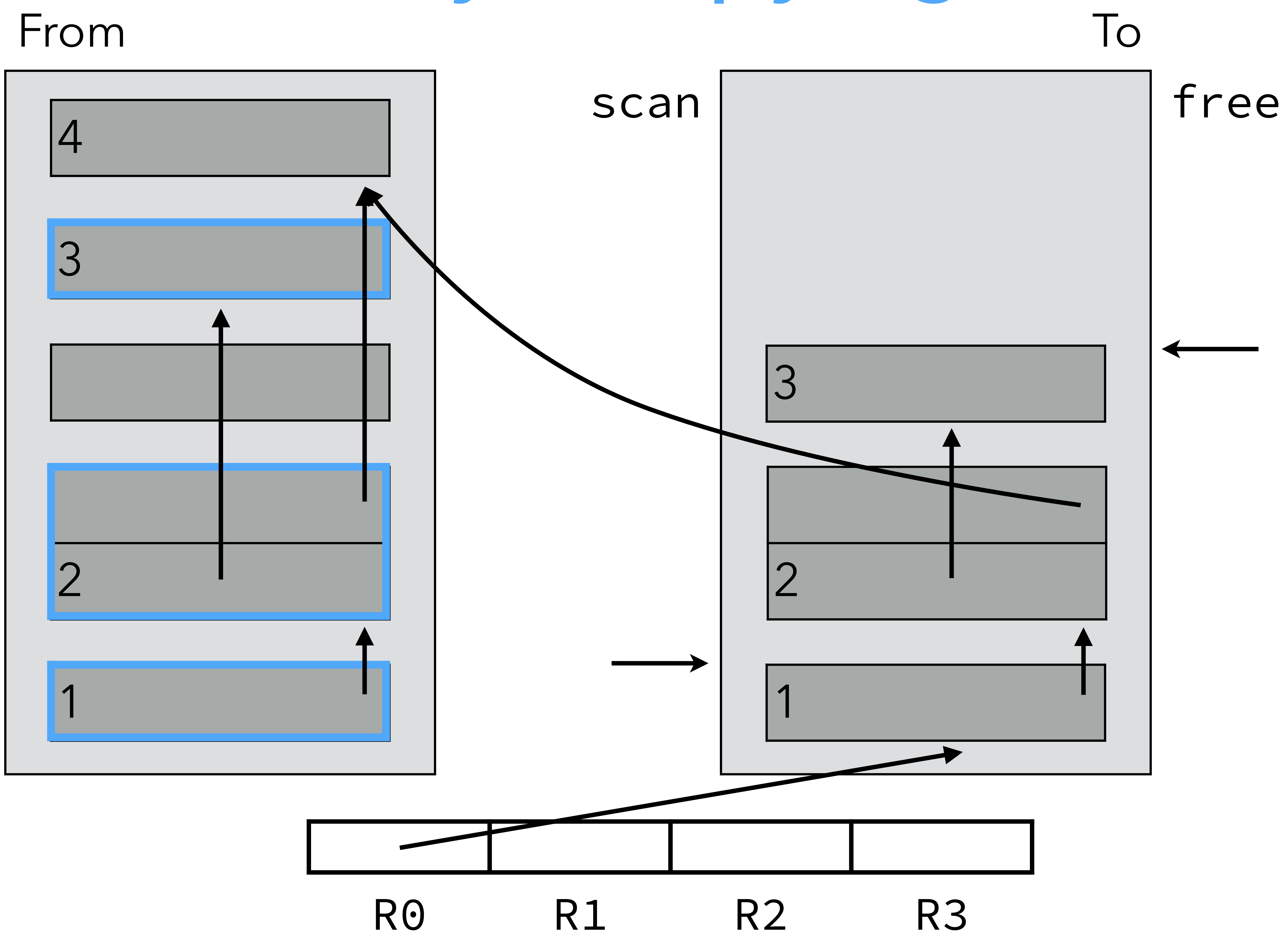
Cheney's copying GC



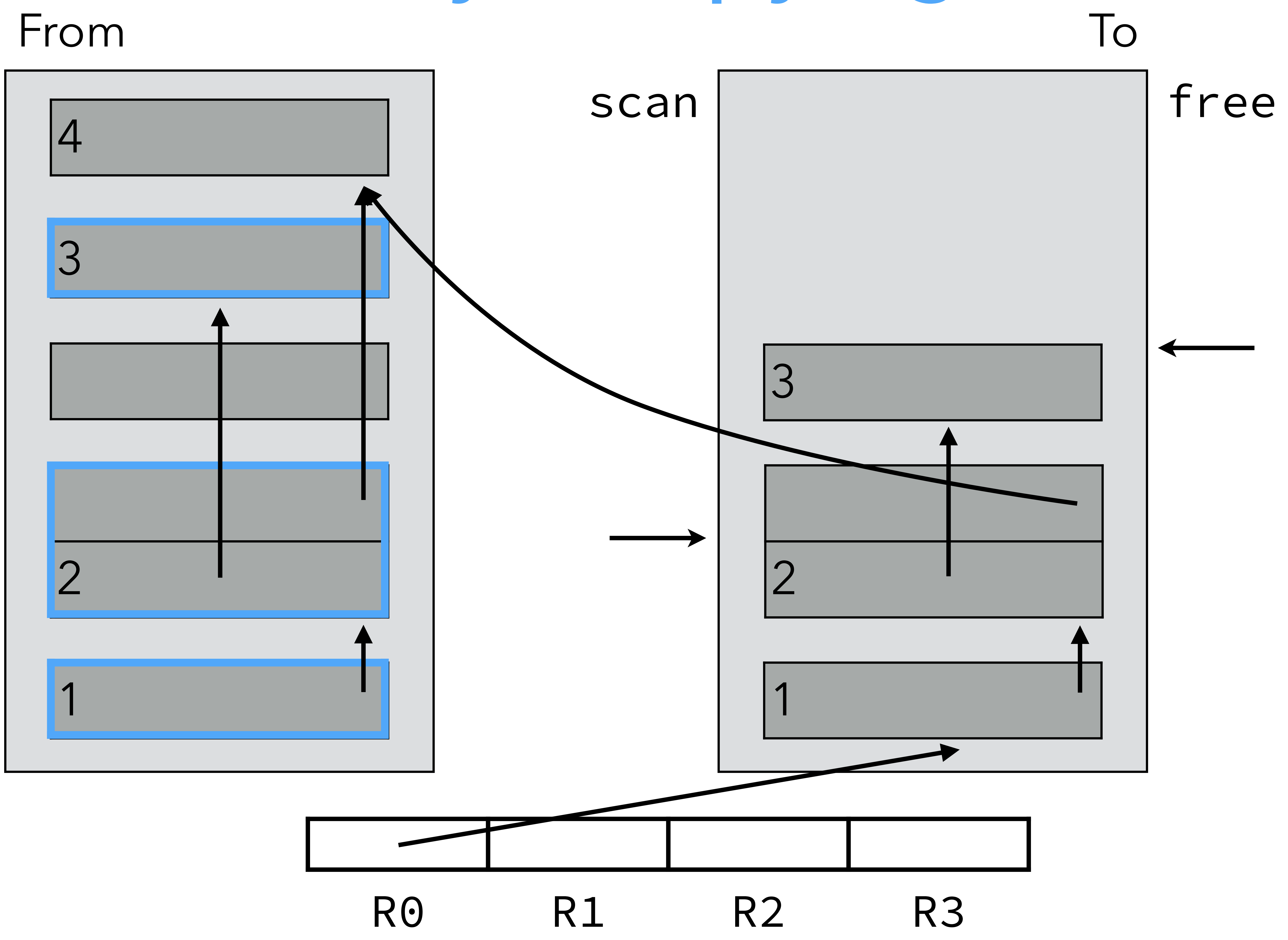
Cheney's copying GC



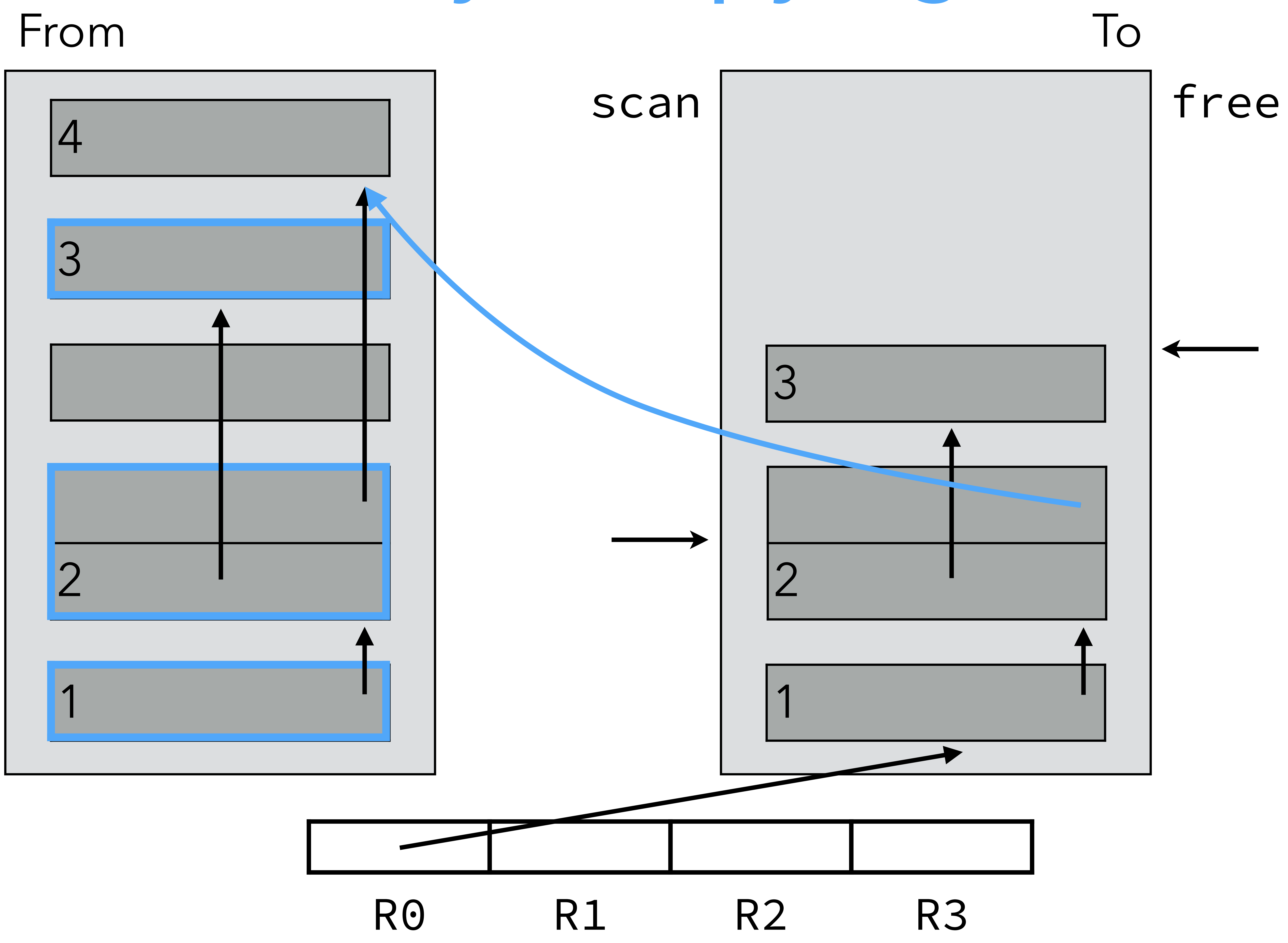
Cheney's copying GC



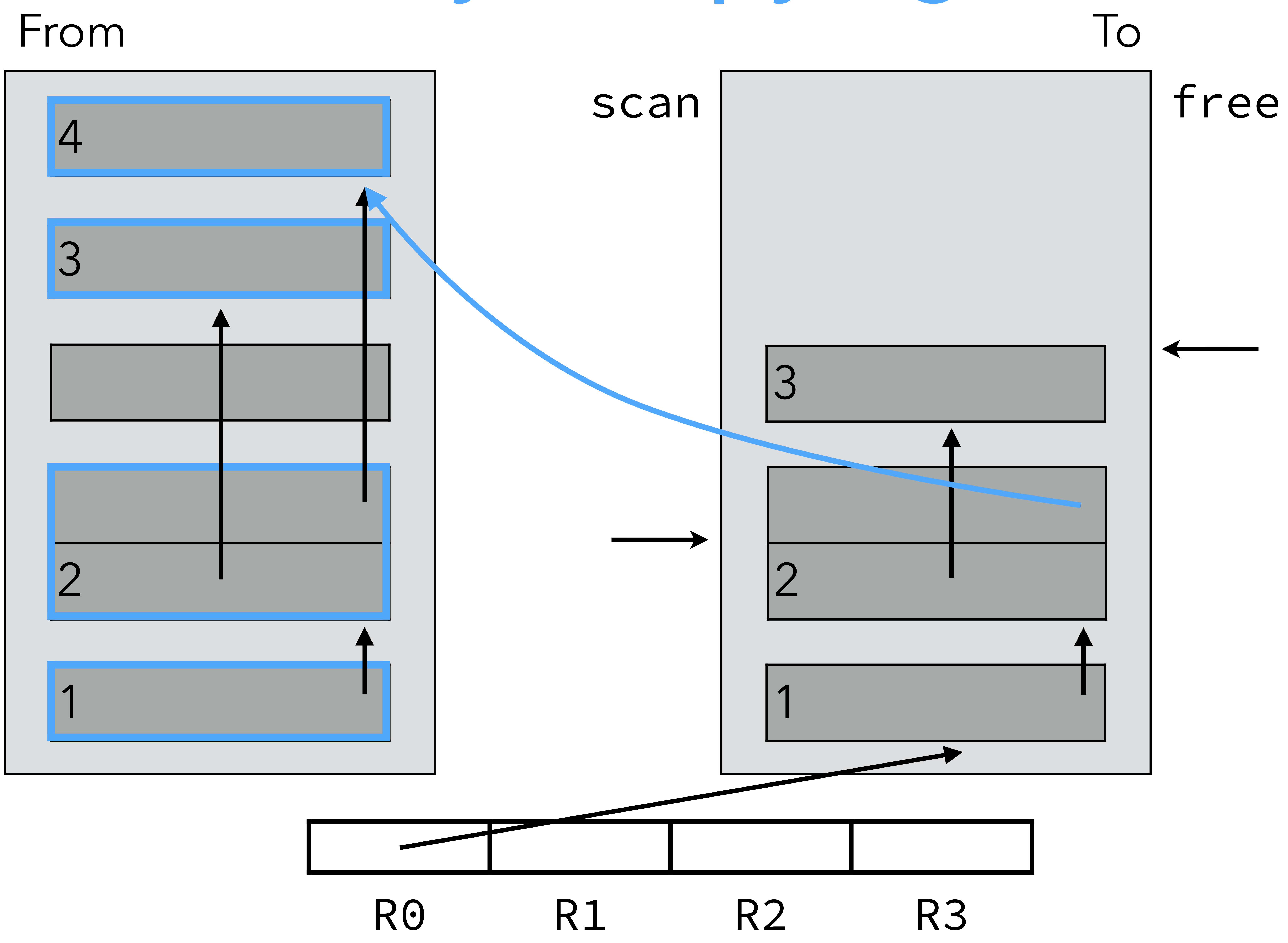
Cheney's copying GC



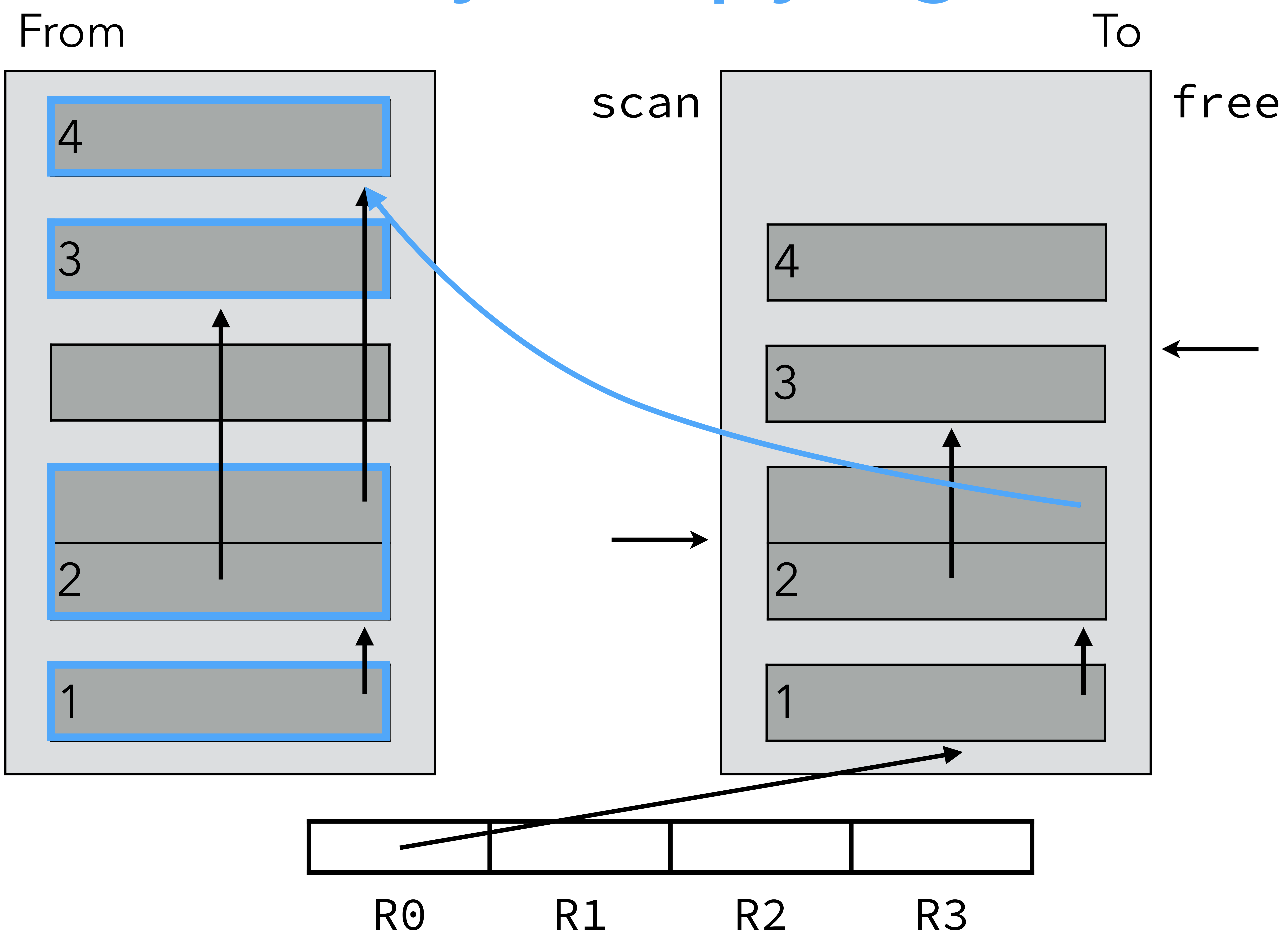
Cheney's copying GC



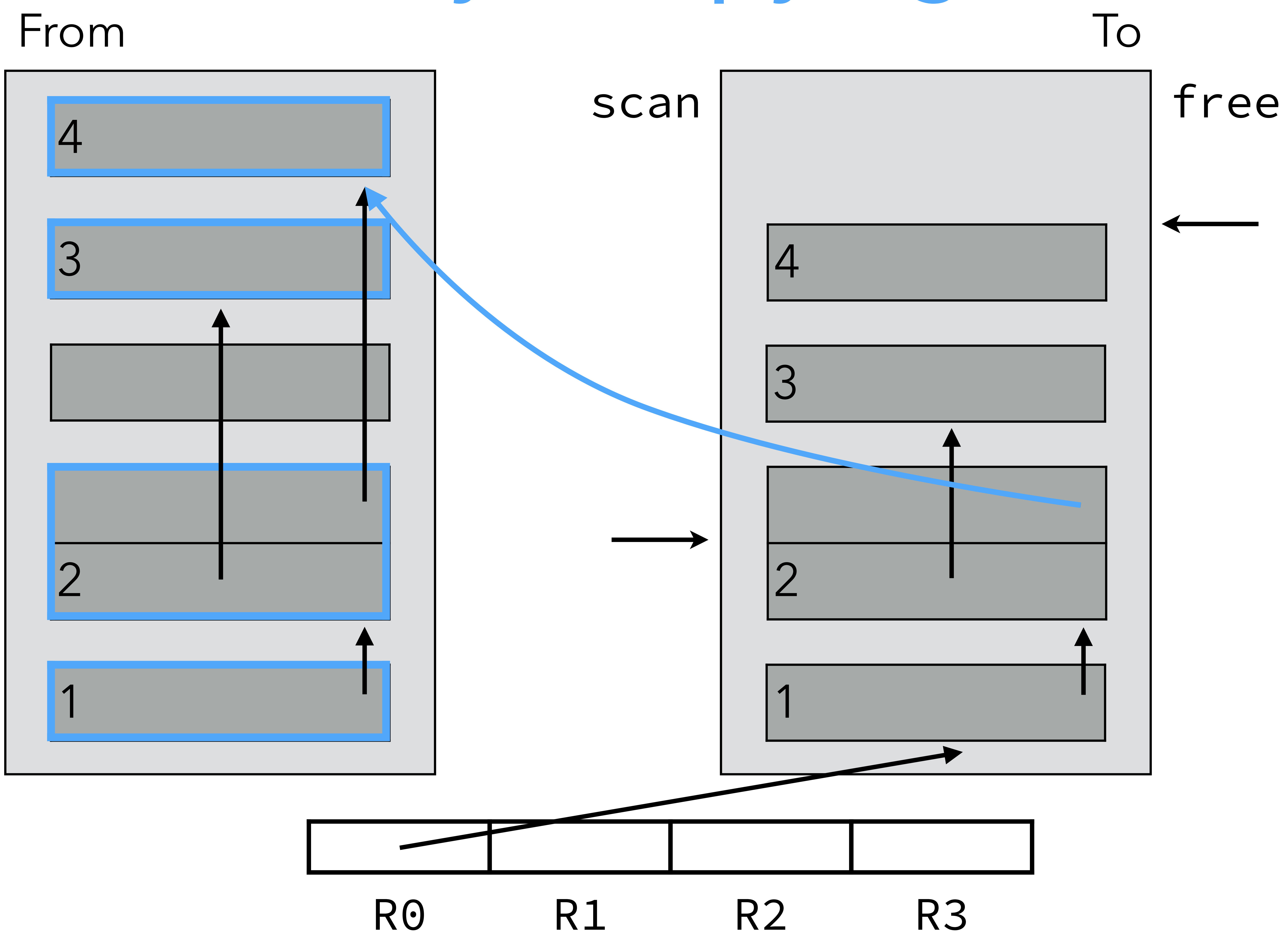
Cheney's copying GC



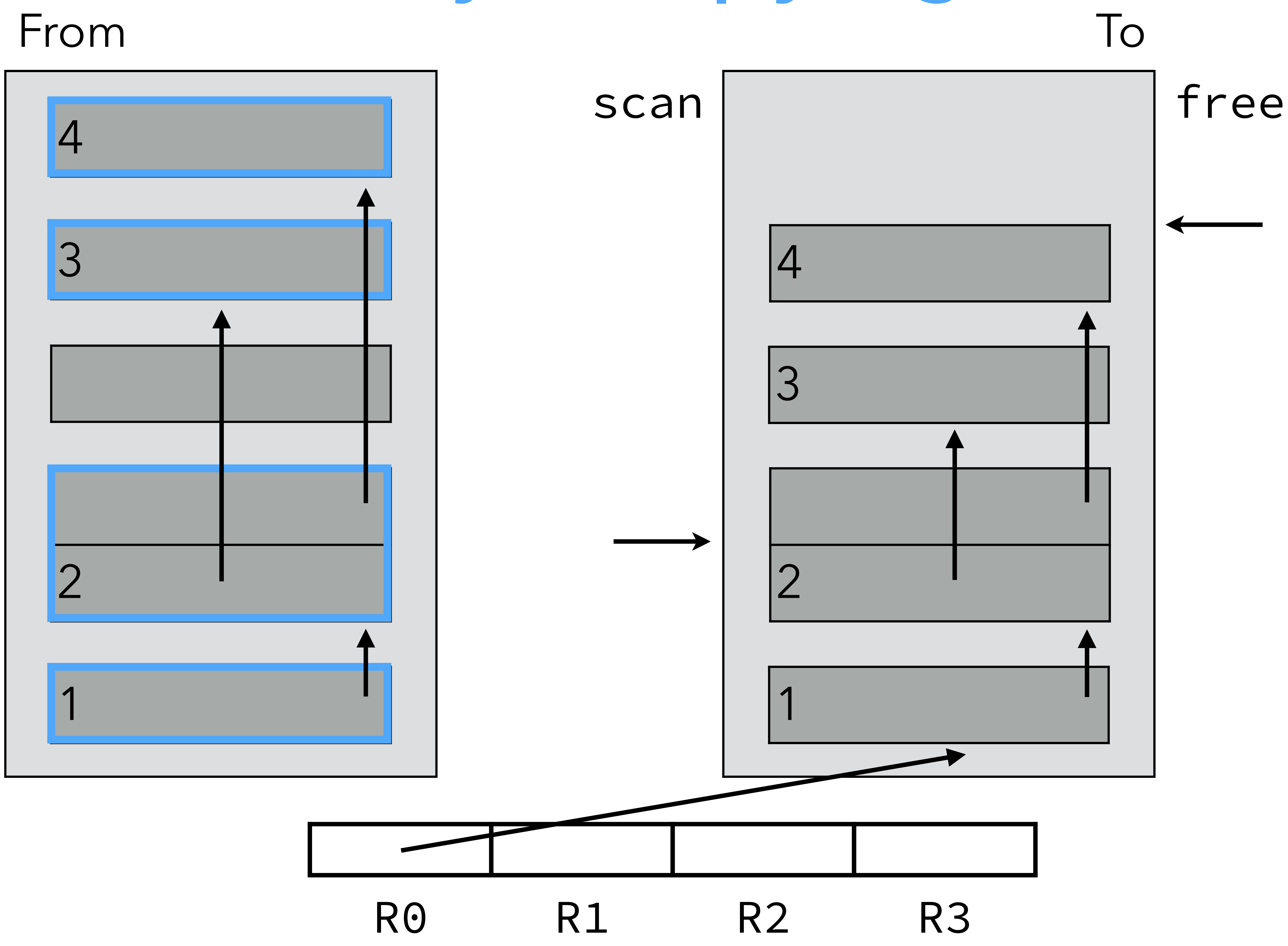
Cheney's copying GC



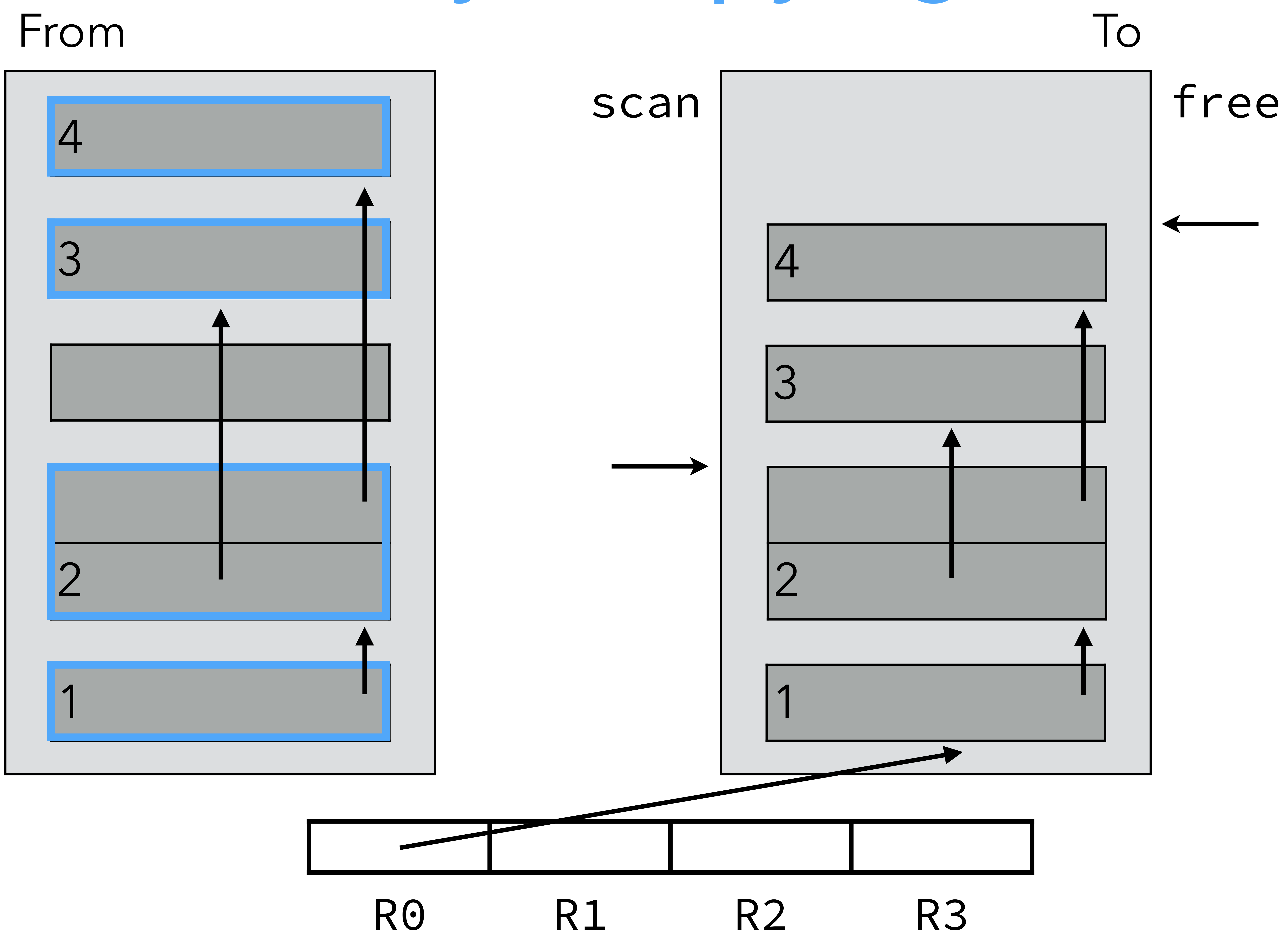
Cheney's copying GC



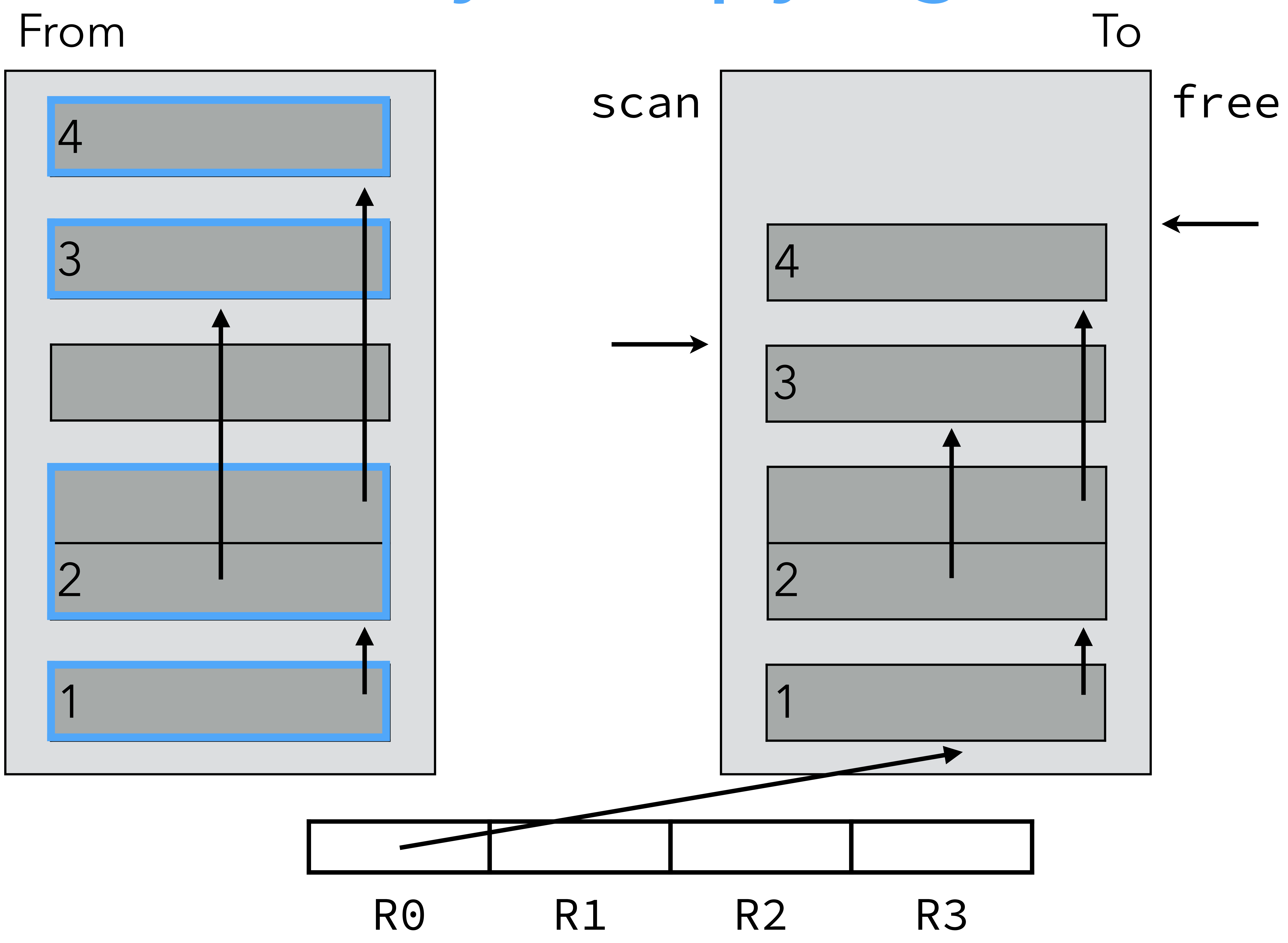
Cheney's copying GC



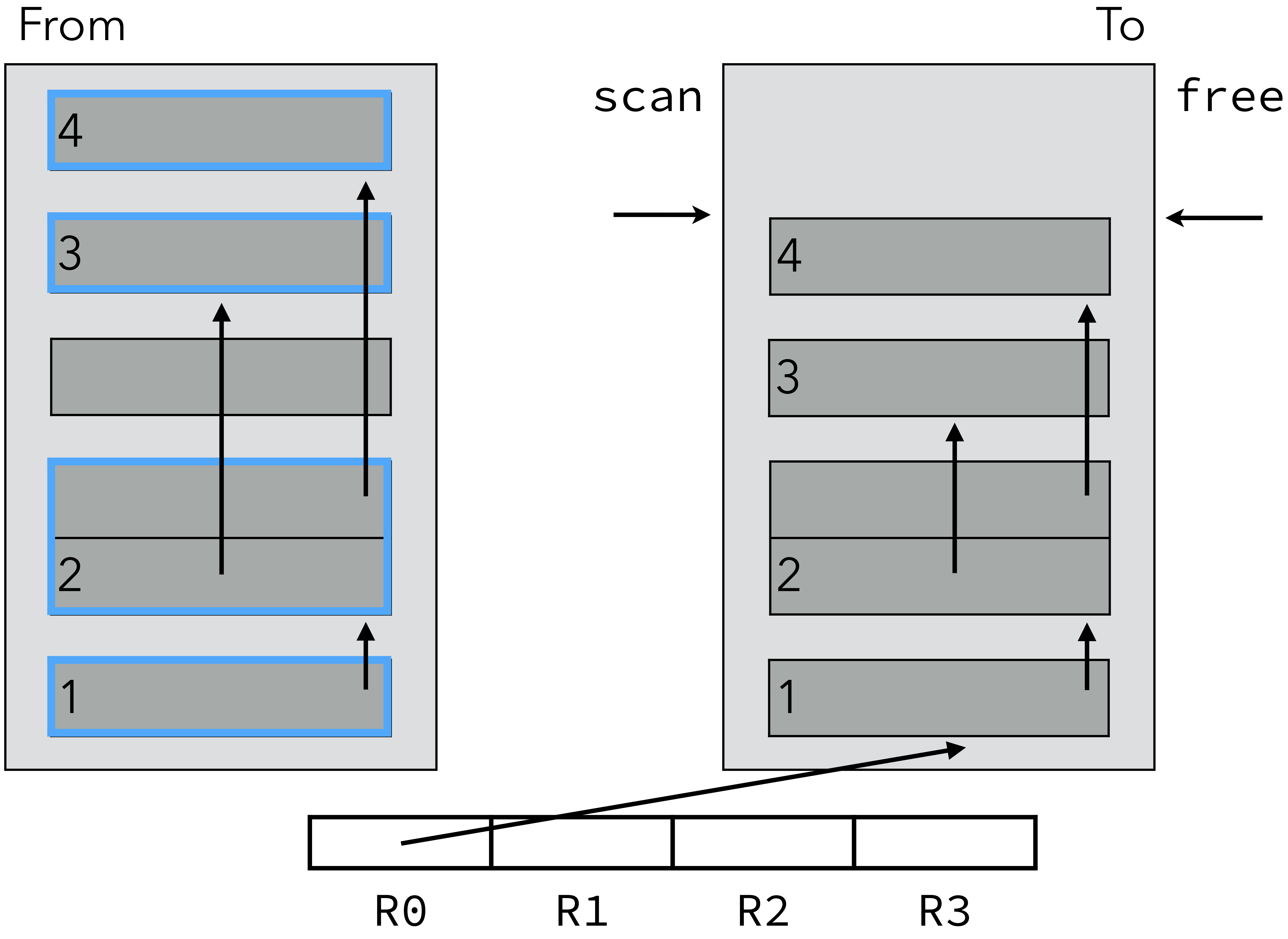
Cheney's copying GC



Cheney's copying GC



Cheney's copying GC



Copying *vs* mark & sweep

The pros and cons of copying garbage collection, compared to mark & sweep.

Pros	Cons
no external fragmentation	uses twice as much (virtual) memory
very fast allocation	requires precise identification of pointers
no traversal of dead objects	copying can be expensive

Exercise

In a system where integers are tagged, a GC can differentiate integers from pointers by looking at the tag bit.
However, during an arithmetic operation, an integer can temporarily be stored in untagged form in a register. Therefore, the GC could mistake it for a real pointer.
Is this problematic? If yes, propose a solution. If not, explain why.

Mostly-copying GC

A copying GC can also be used in situations where not all pointers can be identified unambiguously. This is the idea of **mostly-copying GC**, due to Bartlett.

In such a GC, objects are partitioned in two classes:

1. those for which some pointers are ambiguous, usually because they appear in the stack or registers,
2. those for which all pointers are known unambiguously.

Objects of the first class are **pinned**, i.e. left where they are, while the others – the vast majority, generally – are copied as usual.

Mostly-copying GC

Objects cannot be pinned if the from and to spaces are organized as two separate areas of memory, because from-space must be completely empty after GC.

Therefore, a mostly-copying GC organizes memory in **pages** of fixed size, tagged with the space to which they belong. Then, during GC :

- pinned objects are left on their page, whose tag is updated to "move" them to to-space,
- other objects are copied (and compacted) as usual.

GC technique #4:
generational GC

Generational GC

Empirical observation: a large majority of the objects die young, while a small minority lives for very long.

Generational garbage collection refines other GC techniques and takes advantage of this by:

- partitioning objects in generations, based on age,
- collecting the young generation(s) more often.

Goals:

- augment the amount of memory collected per objects visited,
- (in a copying GC): avoid repeatedly copying long-lived objects.

Generational GC

In a generational GC, objects are partitioned in two (or more) generations, the young and the old:

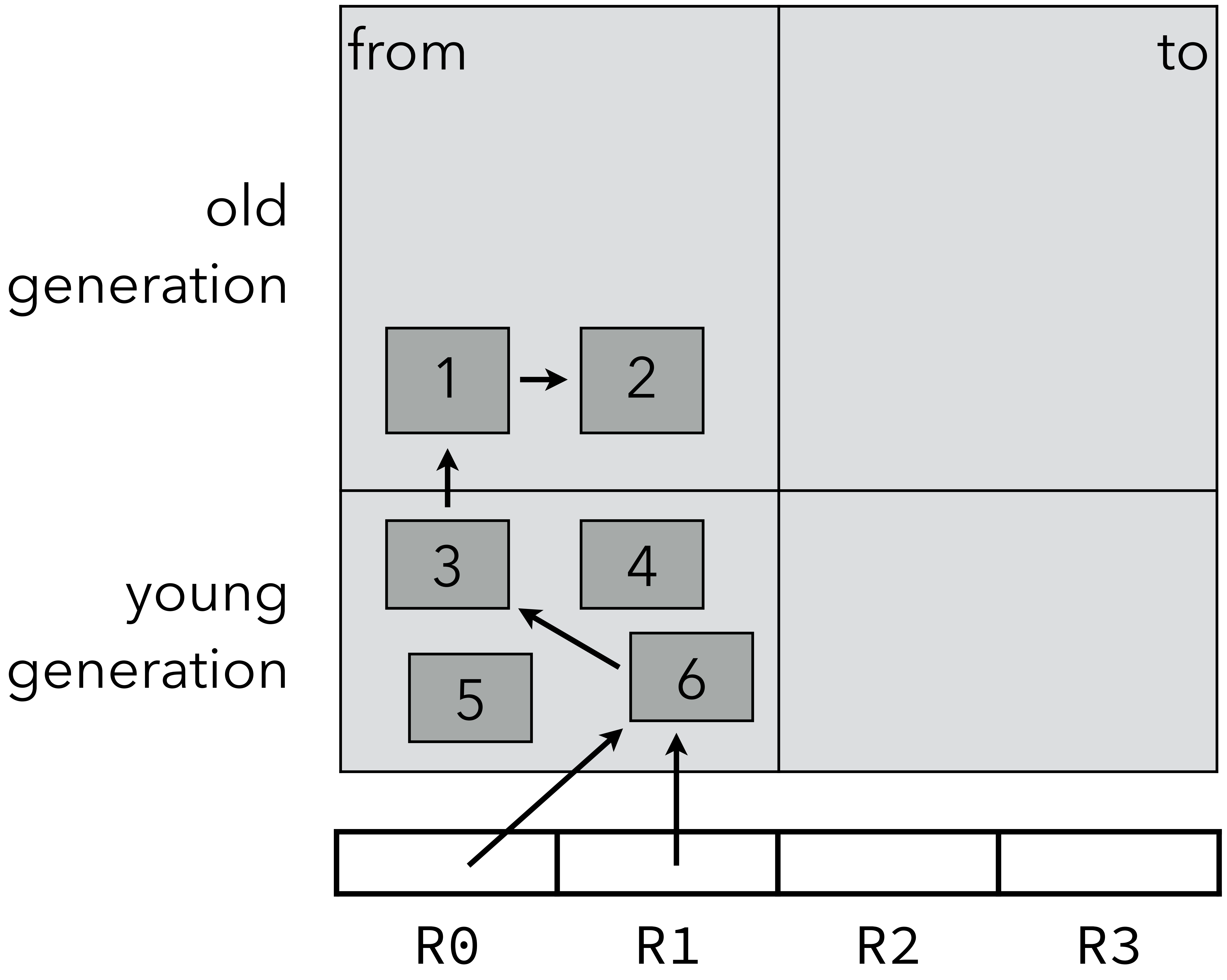
- the young generation is smaller than the old one,
- all objects are allocated in the young generation.

When the young generation is full, a **minor collection** is performed to:

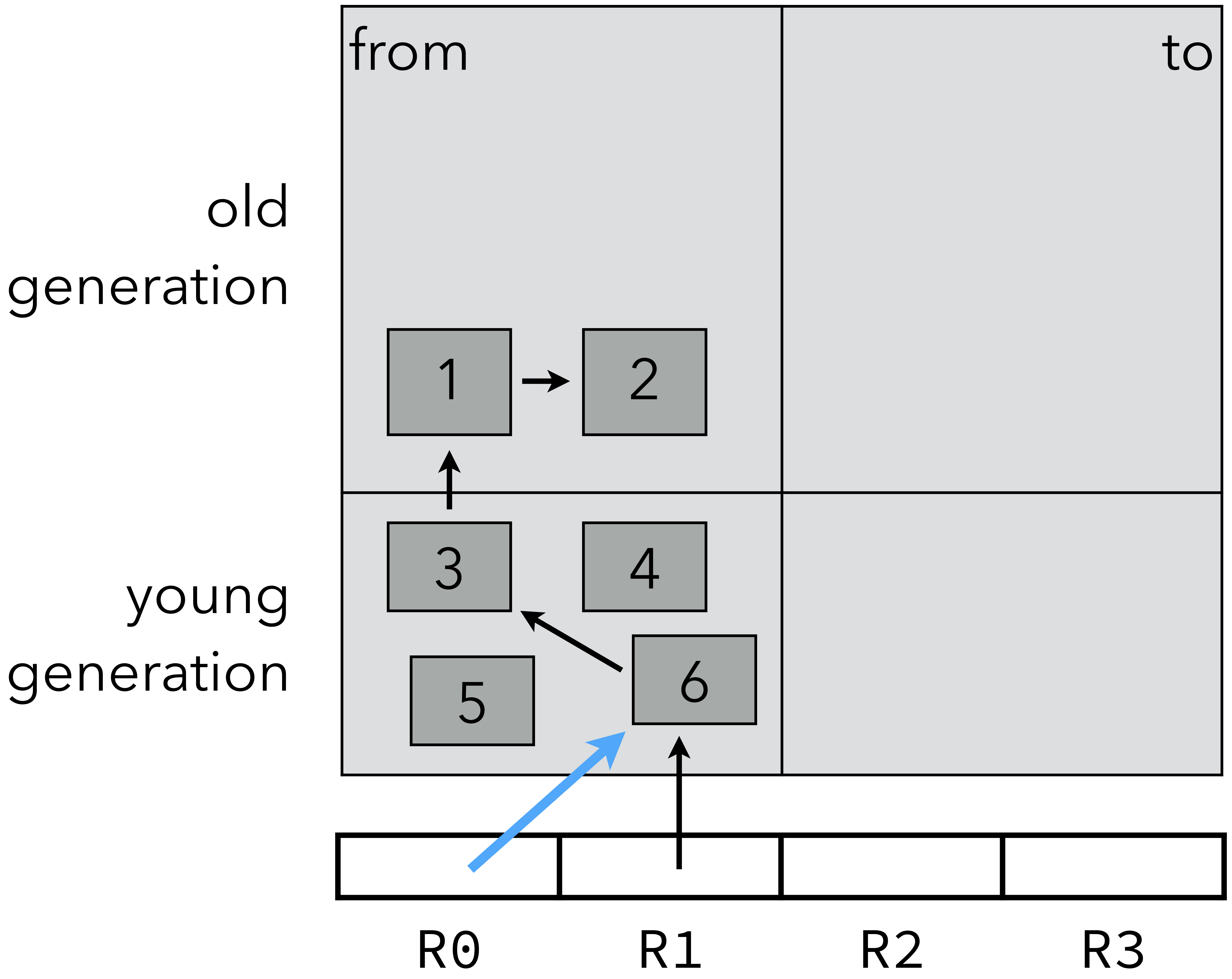
- collect memory in that generation only,
- promote some objects to the old generation, based on a **promotion policy**.

When the old generation is full, a **major** (or **full**) **collection** is performed to collect memory in *all* generations.

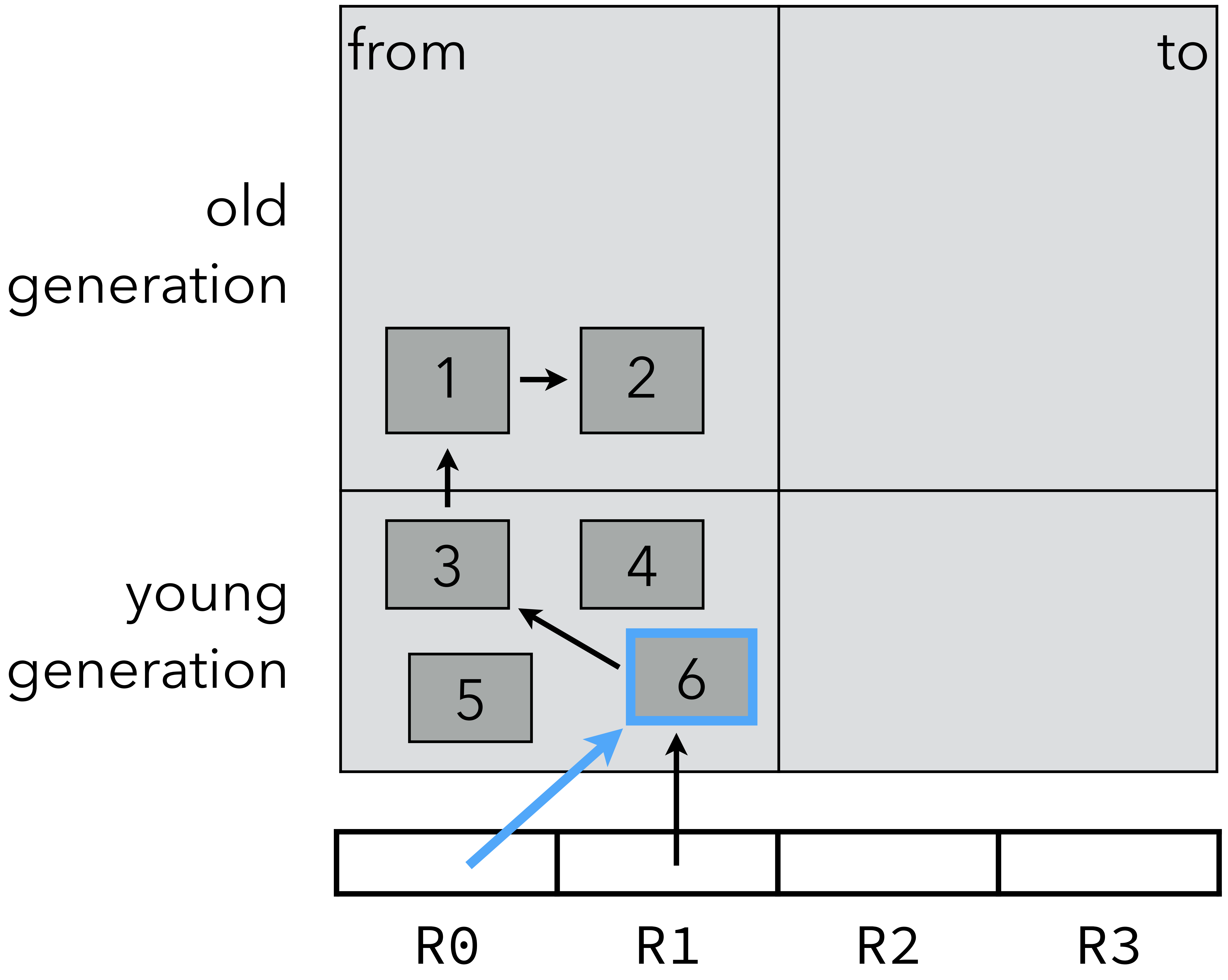
Minor collection example



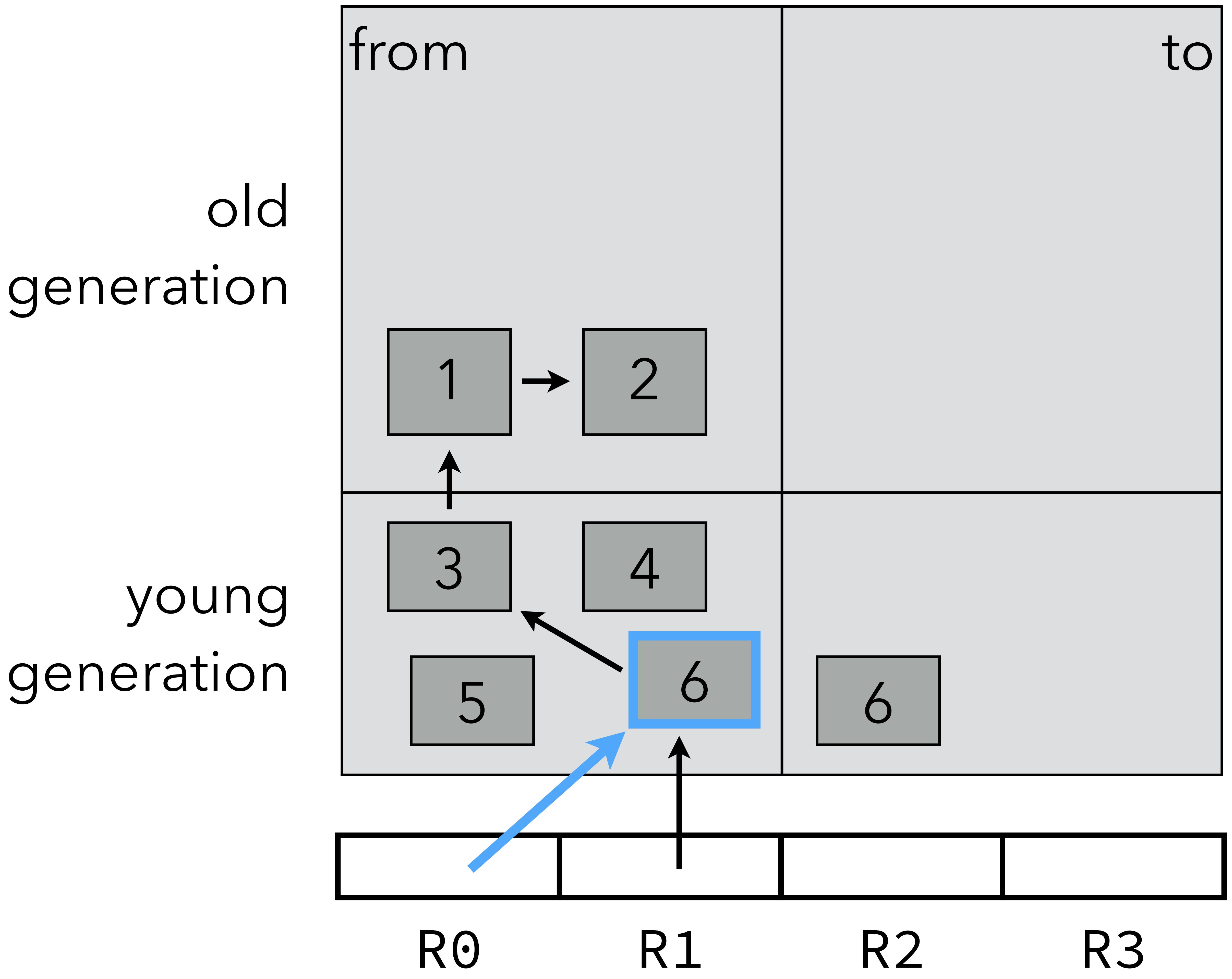
Minor collection example



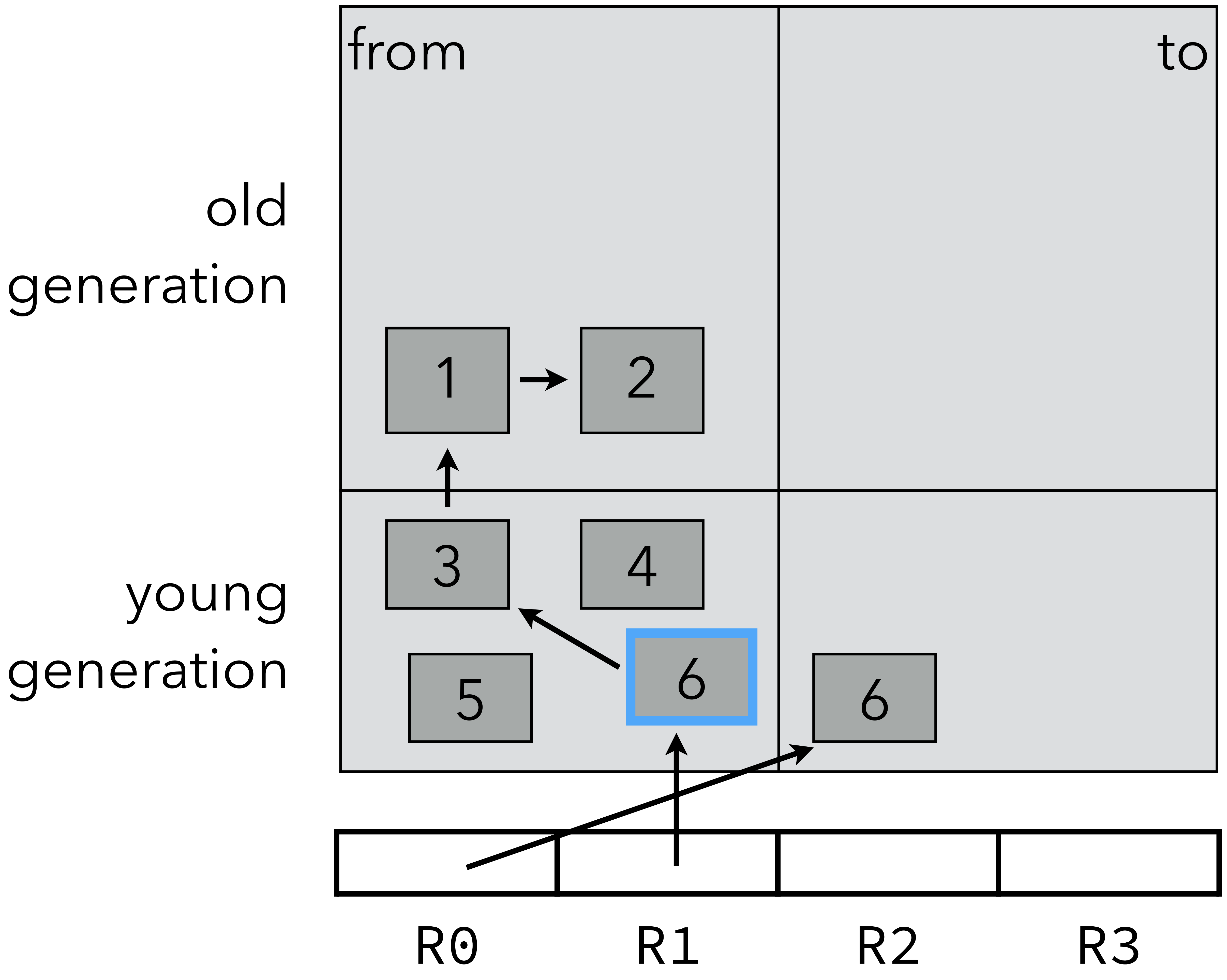
Minor collection example



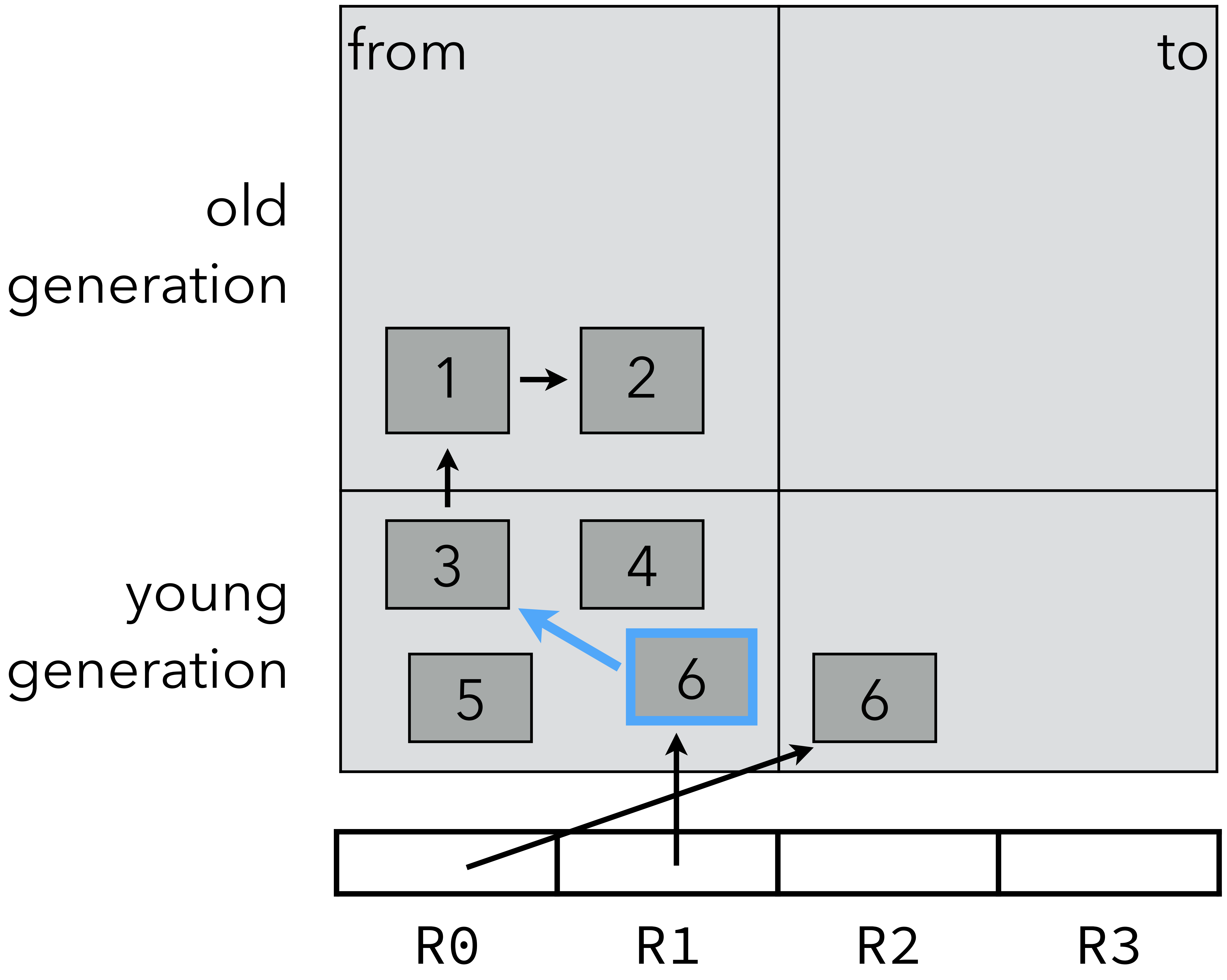
Minor collection example



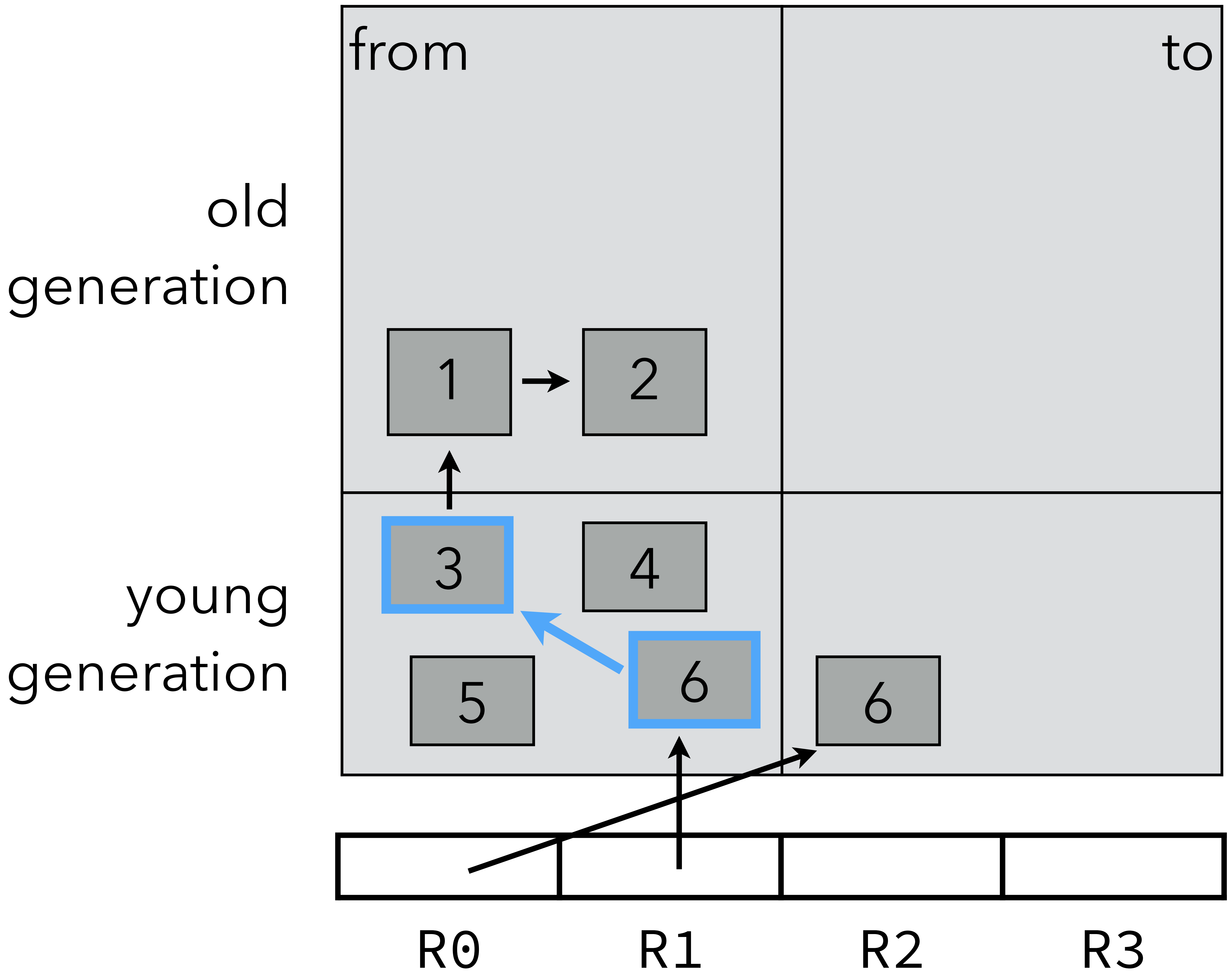
Minor collection example



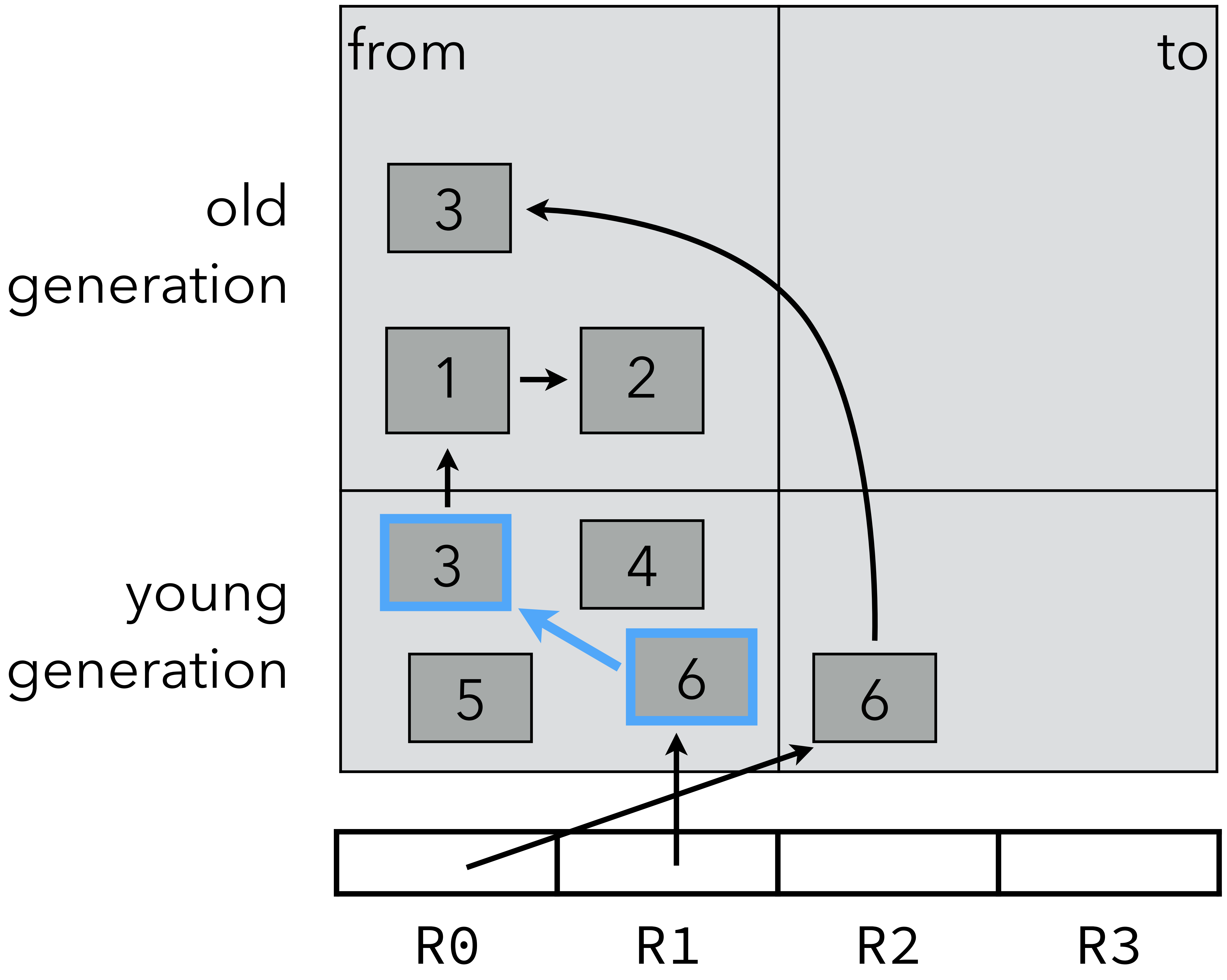
Minor collection example



Minor collection example

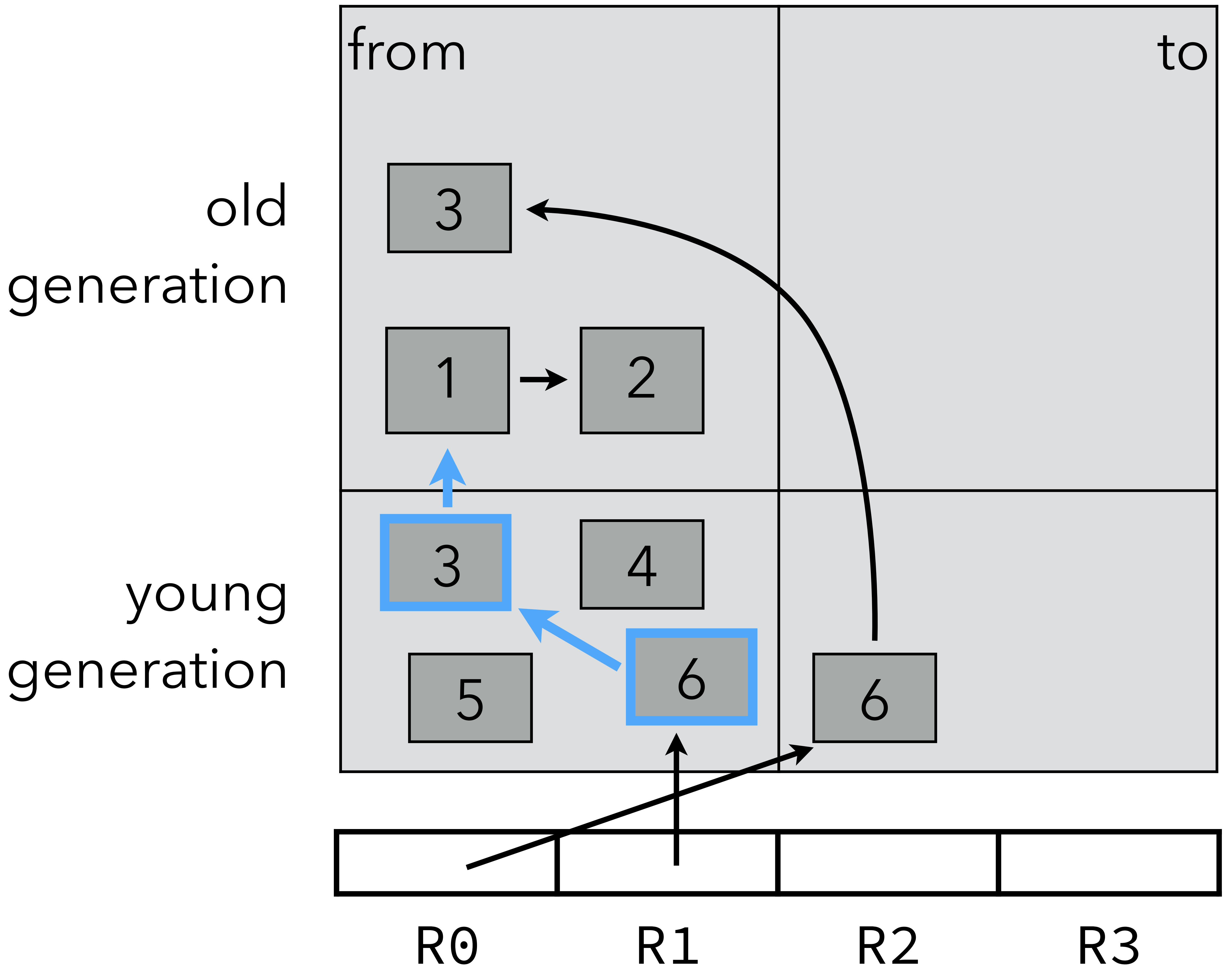


Minor collection example



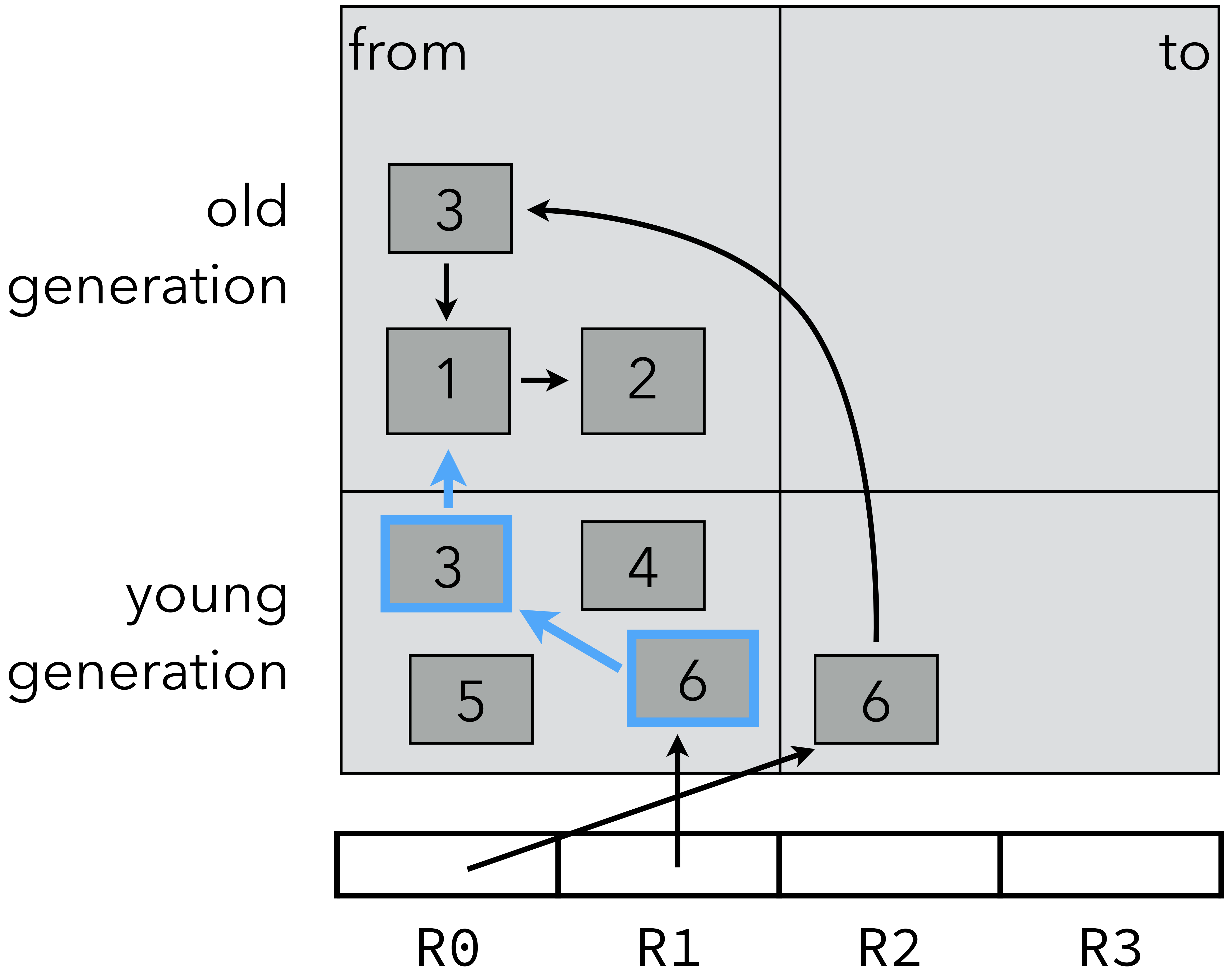
Object 3 is considered old enough to be promoted.

Minor collection example



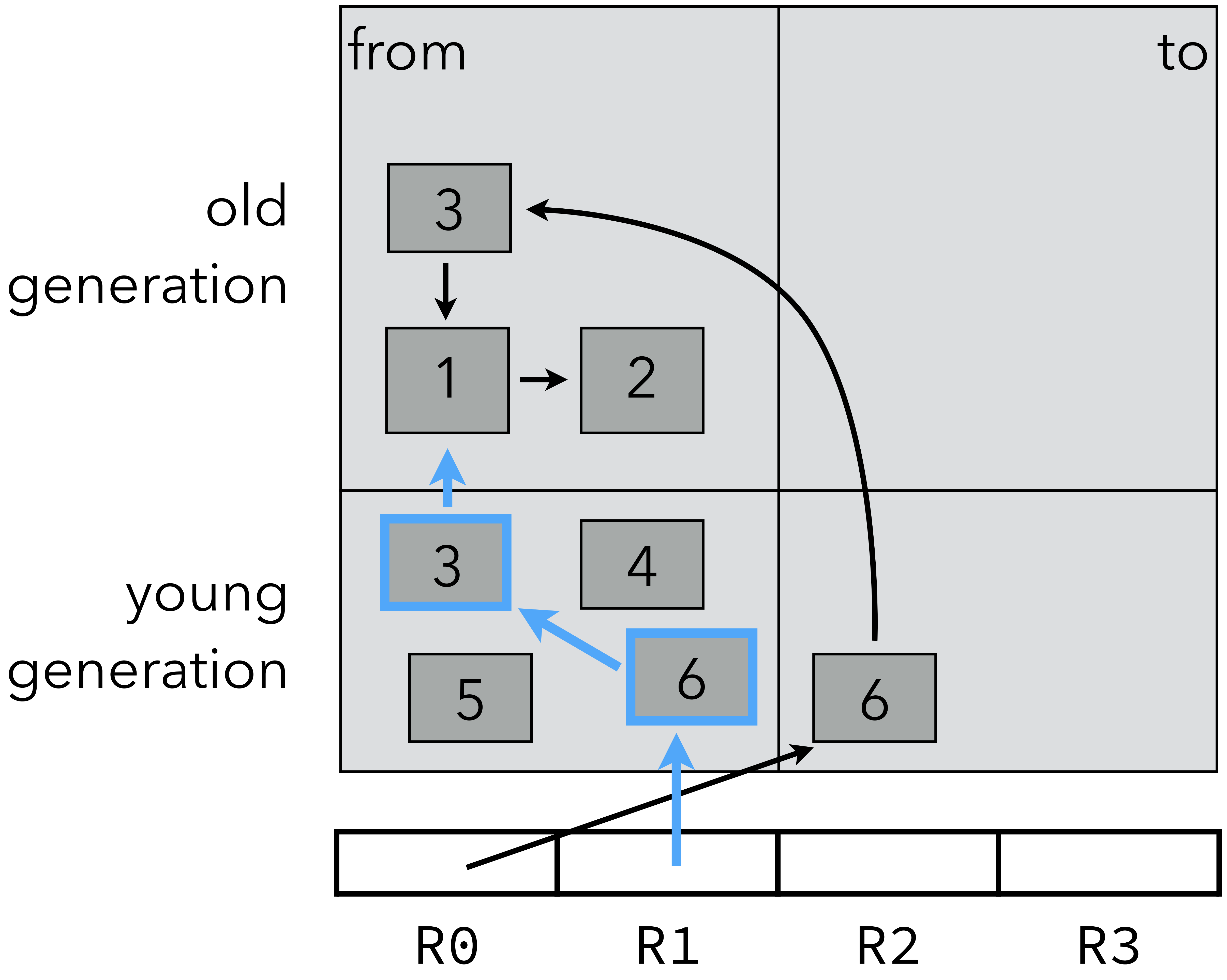
Object 3 is considered old enough to be promoted.

Minor collection example



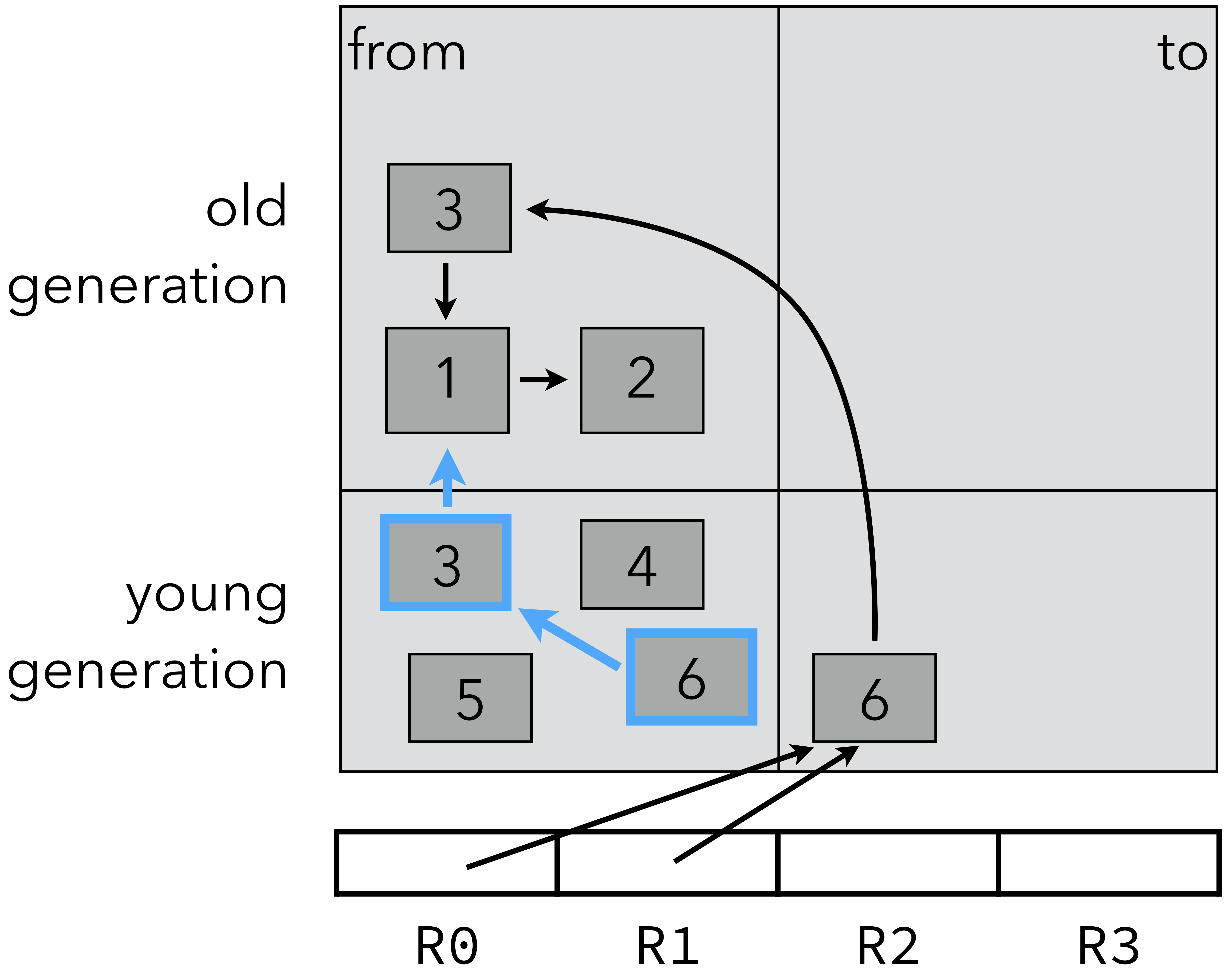
Object 3 is considered old enough to be promoted.

Minor collection example



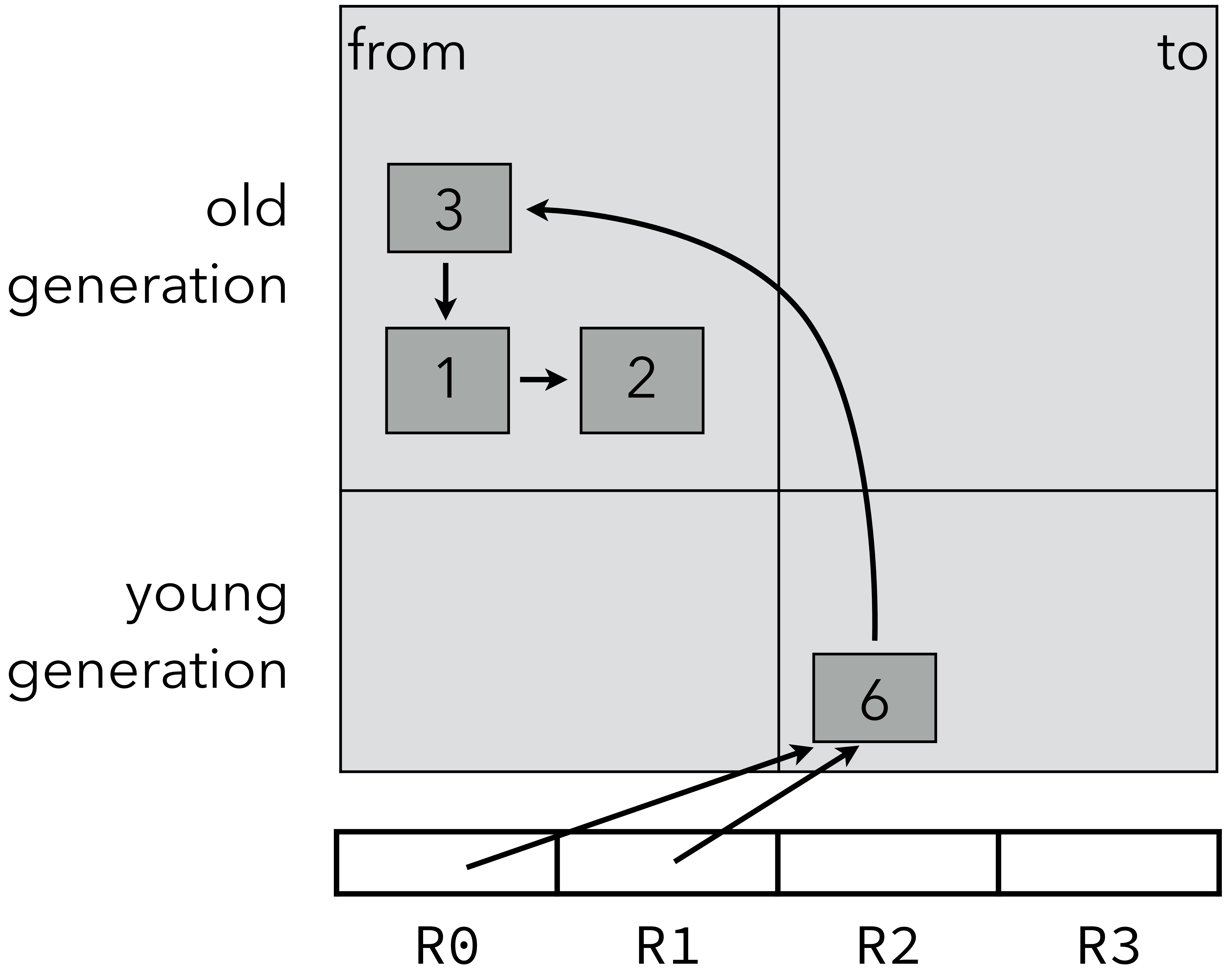
Object 3 is considered old enough to be promoted.

Minor collection example



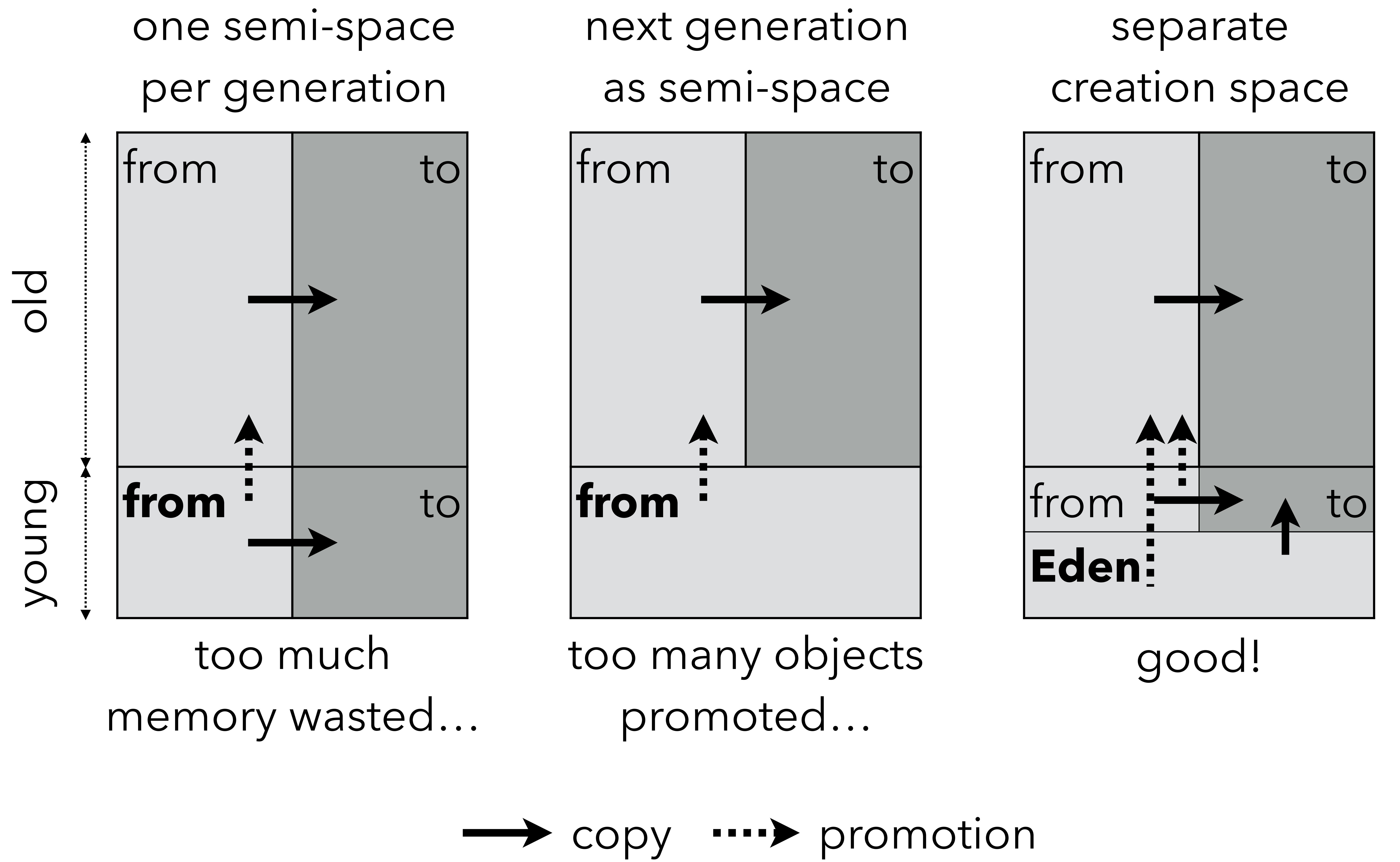
Object 3 is considered old enough to be promoted.

Minor collection example



Object 3 is considered old enough to be promoted.

Heap organization



Hybrid heap organization

Instead of managing all generations using a copying algorithm, it is also possible to manage some of them – the oldest, typically – using a mark & sweep algorithm.

Promotion policies

Generational GCs use a **promotion policy** to decide when objects should be advanced to an older generation.

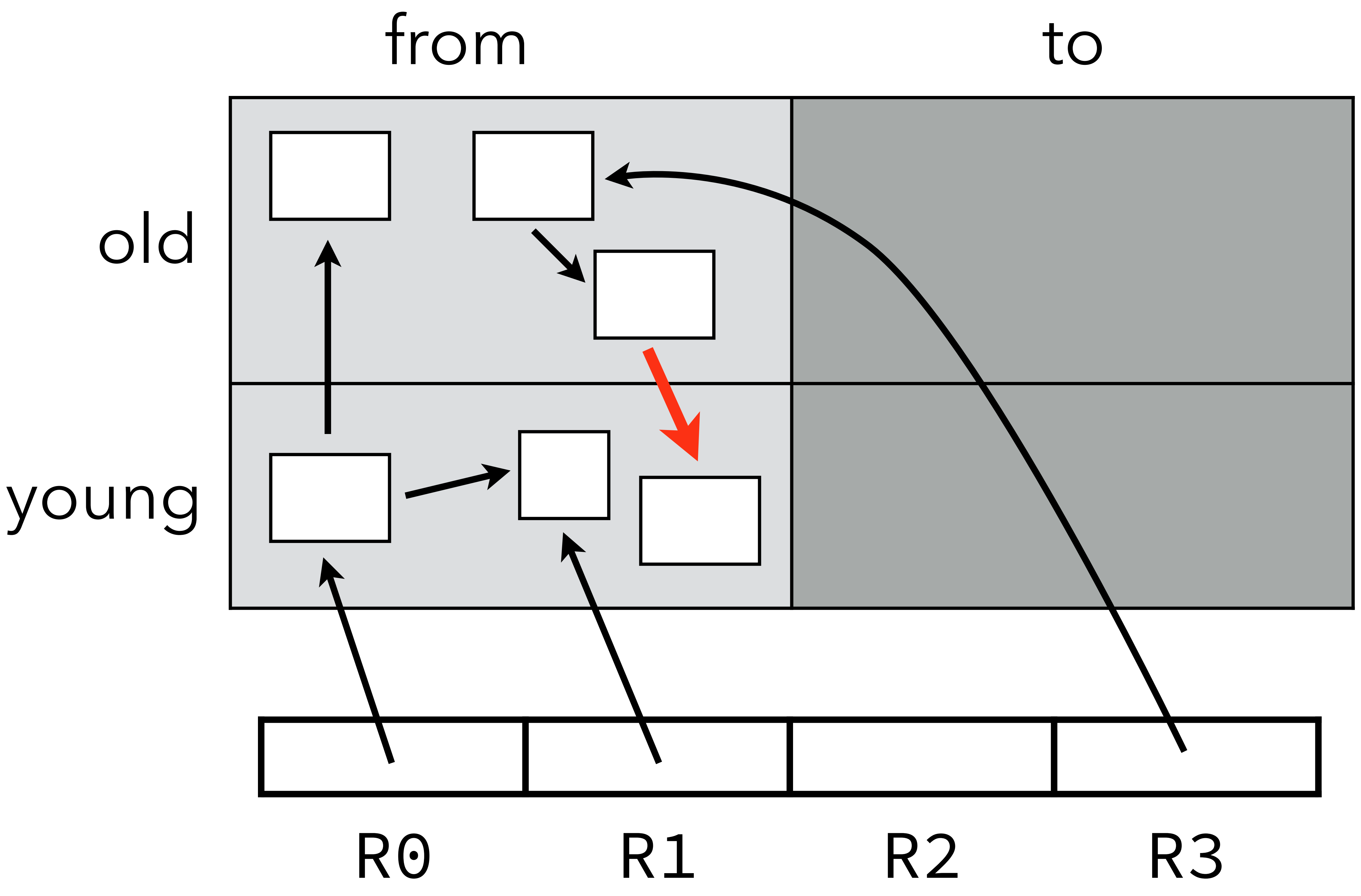
The simplest one is to advance all survivors, which is:

- simple to implement (no object age to record),
- bad, as extremely young objects can be promoted.

A better policy is to wait until objects survive a second collection before promoting them.

Minor collection roots

The roots used for a minor collection must also include all pointers from older generations to younger ones. Otherwise, objects reachable only from the old generation would incorrectly get collected!



Inter-generational pointers

Pointers from old to young generations, called **inter-generational** pointers can be handled in two different ways:

1. by scanning – without collecting – older generations during a minor collection,
2. by detecting pointer writes using a write barrier – implemented either in software or through hardware support – and remembering inter-generational pointers.

Remembered set

A **remembered set** contains all old objects pointing to young objects.

The write barrier maintains this set by adding objects to it if and only if:

- the object into which the pointer is stored is not yet in the remembered set, and
- the pointer is stored in an old object, and points to a young one (can also be checked later by the collector).

Card marking

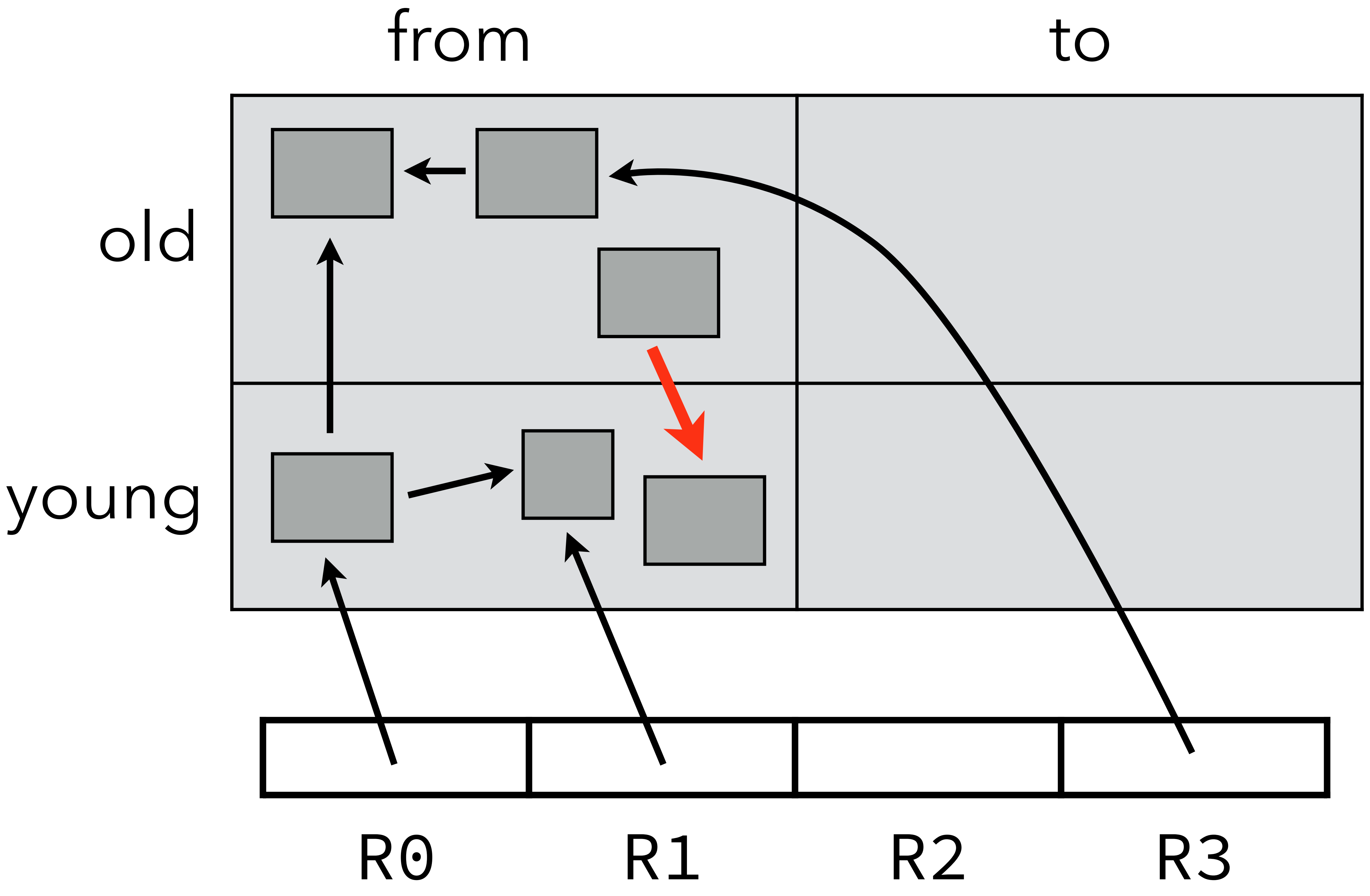
Card marking is another technique to detect inter-generational pointers:

- memory is divided into **cards** (small, fixed sized areas),
- a **card table** remember which cards potentially contain inter-generational pointers,
- on each pointer write, the card is marked in the table,
- marked cards are scanned for inter-generational pointers during collection.

Nepotism

Since old generations are not collected as often as young ones, it is possible for dead old objects to prevent the collection of dead young objects.

This problem is called **nepotism**.



Pros and cons

Pros of generational GCs:

- reduce pause time, since only the youngest generation is collected most of the time,
- avoid repeatedly copying long-lived objects in copying GCs.

Cons of generational GCs:

- maintaining the remembered set costs time,
- nepotism.

Other kinds of garbage collectors

Incremental/concurrent GC

To reduce GC pauses, which is very important for interactive applications:

- **incremental GCs** collect memory in incremental steps,
- **concurrent GCs** collect memory in a thread executing concurrently with the application.

Main difficulty:

Deal with modifications to the reachability graph made by the application while they attempt to compute it.

Usually solved using read or write barriers, used to ensure that the reachability graph observed by the GC is a valid approximation of the real one.

Parallel GC

Some parts of GC can be performed in parallel on several processor cores, e.g. the marking phase of a M&S GC.

Remember: parallelism ≠ concurrency, so a parallel GC doesn't have to be concurrent, and a concurrent GC doesn't have to be parallel.

Additional GC features

Finalizers

Some GCs make it possible to associate **finalizers** with objects, which are functions called when an object is about to be collected.

Finalizers are generally used to free “external” resources associated with the object about to be freed.

Since there is no guarantee about when finalizers are invoked, the resource in question should not be scarce.

Finalizer issues

Finalizers are tricky for a number of reasons:

- what should be done if a finalizer makes the finalized object reachable again – e.g. by storing it in a global variable?
- how do finalizers interact with concurrency – e.g. in which thread are they run?
- how can they be implemented efficiently in a copying GC, which doesn't visit dead objects?

Weak references

When a GC encounters a reference (i.e. pointer), it usually considers it as **strong**, in the sense that it will prevent the referenced object from being deallocated.

Some GCs offer **weak references**:

- if an object is reachable only through weak references, it is deallocated,
- when that happens, all (weak) references to it are atomically cleared.

This is useful to implement caches, for example.