Memory management

Advanced Compiler Construction

Michel Schinz – 2014-04-10

Memory management

The memory of a computer is a finite resource. Typical programs use a lot of memory over their lifetime, but not all of it at the same time.

The aim of **memory management** is to use that finite resource as efficiently as possible, according to some criterion.

In general, programs dynamically allocate memory from two different areas: the stack and the heap. Since the management of the stack is trivial, the term memory management usually designates that of the heap.

The memory manager

The **memory manager** is the part of the run time system in charge of managing heap memory.
Its job consists in maintaining the set of free **memory blocks** (also called objects later) and to use them to fulfill allocation requests from the program.
Memory deallocation can be either explicit or implicit:

- it is **explicit** when the program asks for a block to be freed,
- it is **implicit** when the memory manager automatically tries to free unused blocks when it does not have enough free memory to satisfy an allocation request.

Explicit deallocation

Explicit memory deallocation presents several problems:

1. memory can be freed too early, which leads to **dangling pointers** — and then to data corruption, crashes, security issues, etc.
2. memory can be freed too late — or never — which leads to **space leaks**.

Due to these problems, most modern programming languages are designed to provide **implicit deallocation**, also called **automatic memory management** — or **garbage collection**, even though garbage collection refers to a specific kind of automatic memory management.

Implicit deallocation

Implicit memory deallocation is based on the following conservative assumption:

> If a block of memory is reachable, then it will be used again in the future, and therefore it cannot be freed. Only unreachable memory blocks can be freed.

Since this assumption is conservative, it is possible to have memory leaks even with implicit memory deallocation. This happens whenever a reference to a memory block is kept, but the block is not accessed anymore.

However, implicit deallocation completely eliminates dangling pointers.

Garbage collection

Garbage collection (**GC**) is a common name for a set of techniques that automatically reclaim objects that are not reachable anymore.

We will examine several garbage collection techniques:

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

All these techniques have concepts in common, which we introduce now.

Reachable objects

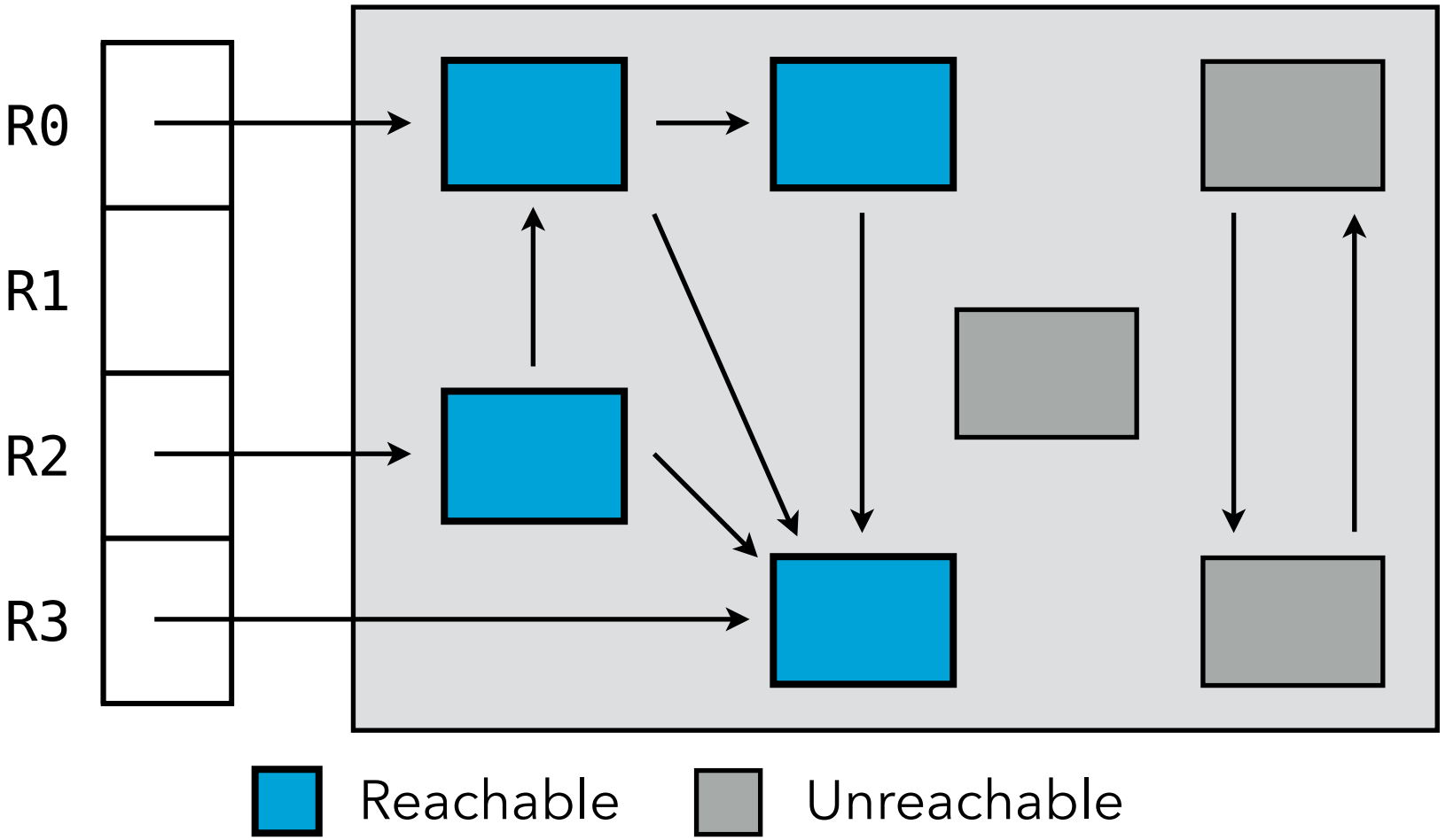
Reachable objects

At any time during the execution of a program, we can define the set of **reachable objects** as being:

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

These objects form the **reachability graph**.

Reachability graph example



(Im)precision

To compute the reachability graph at run time, it must be possible to identify unambiguously pointers in registers, in the stack, and in heap objects.

If this is not possible, the reachability graph must be approximated. Such an approximation must be safe for the task at hand!

For example, when the reachability graph is used to free memory, it is only safe to over-approximate it.

Memory manager data structures

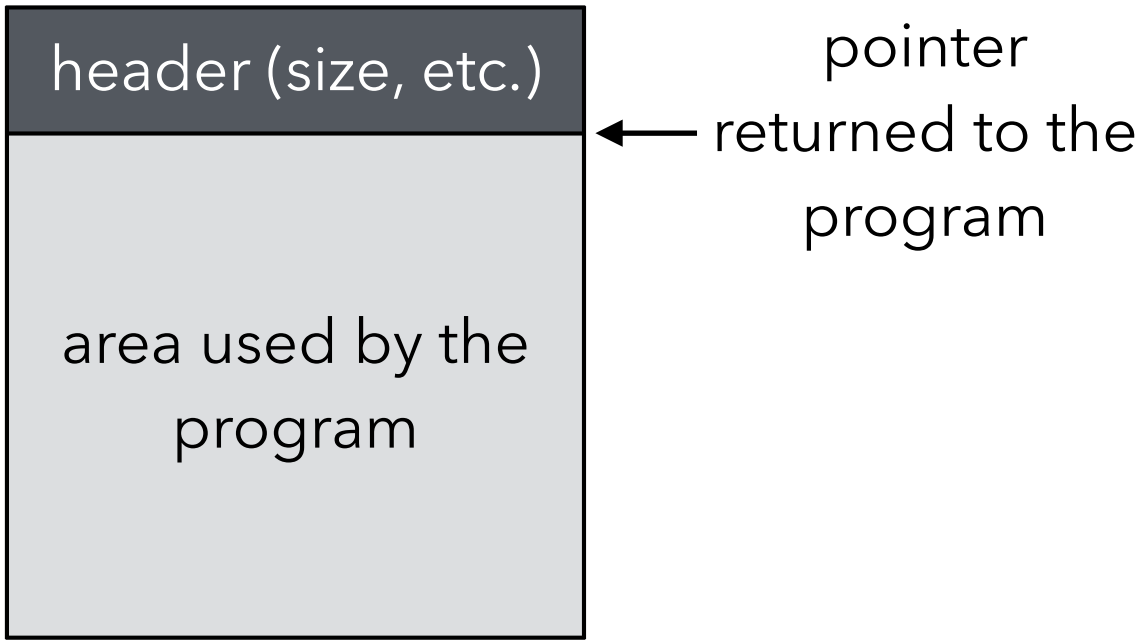
Free list

The memory manager must know which parts of the heap are free, and which are allocated.
It typically does that by maintaining a collection of free blocks, called the **free list** – even when it is not represented as a list.
The actual representation of the free list varies depending on the organization of the heap and the properties of the garbage collector.

Block header

The memory manager must associate a set of properties to the blocks it manages, for example their size.

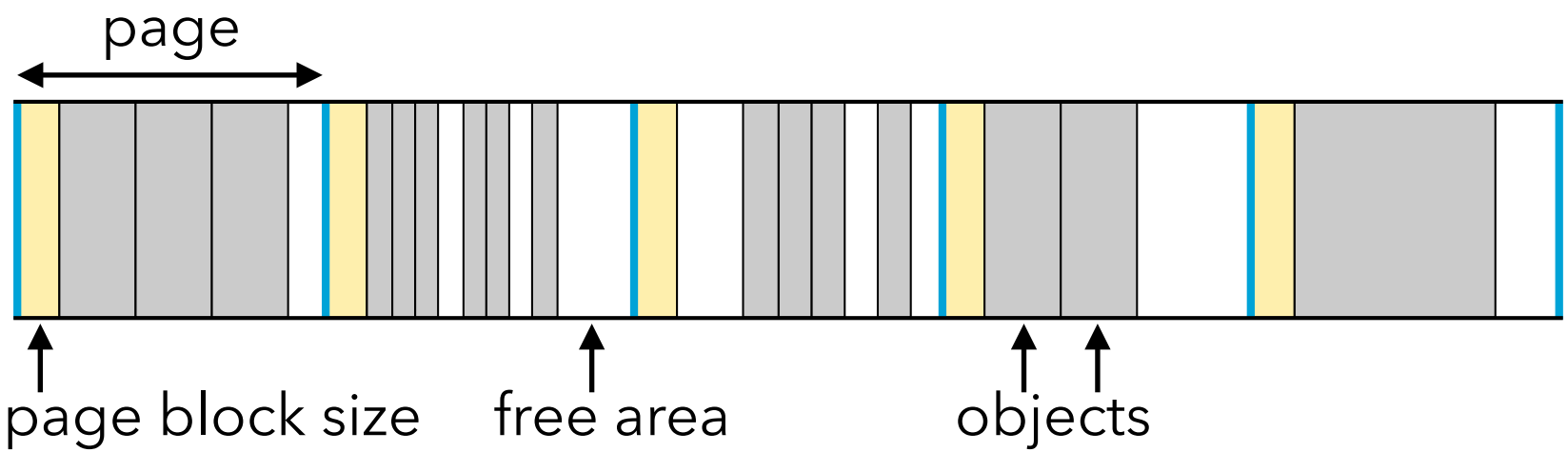
This is typically done by storing these properties in a **block header**, stored just before the area used by the program.



BiBOP

To decrease the overhead of headers, the technique known as **BiBOP** (for **big bag of pages**) groups objects of identical size into contiguous areas of memory called pages. All pages have the same power-of-two size $s = 2^b$ and start at an address which is a multiple of s.

The size of all the objects on a page is stored at the page's beginning, and can be retrieved by masking the b least-significant bits of an object's address.



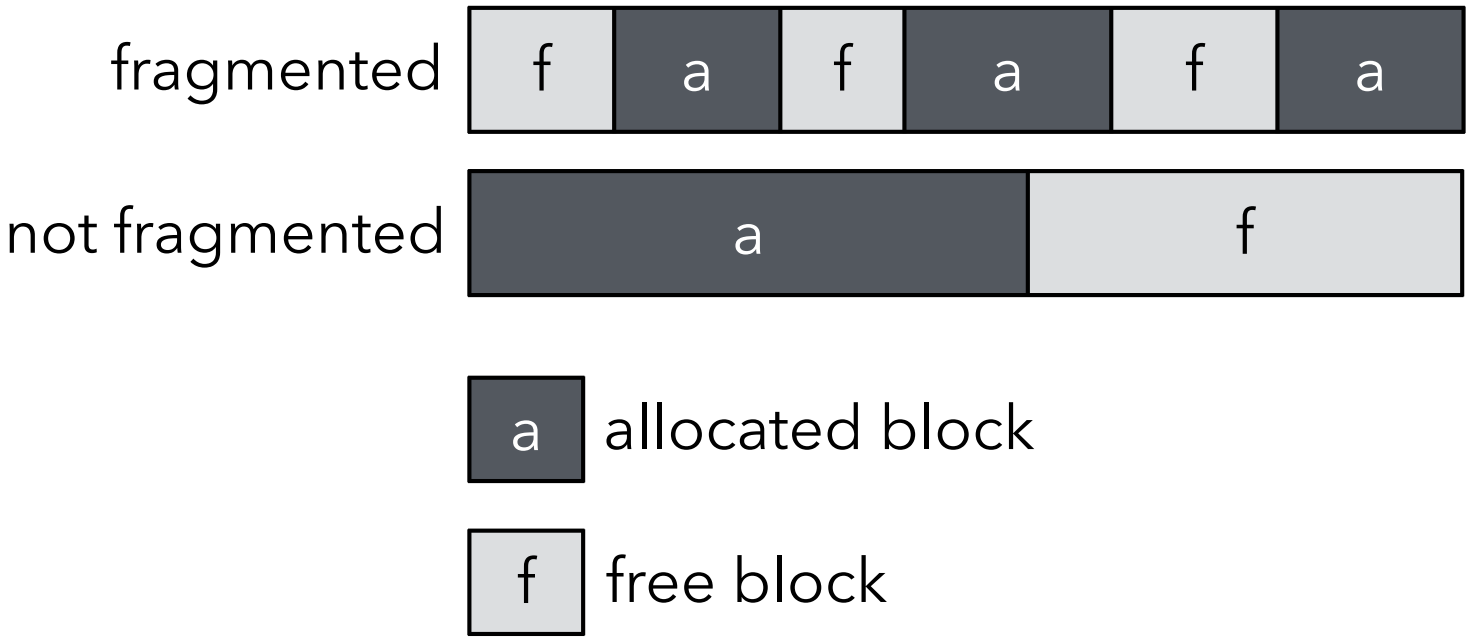
Fragmentation

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.

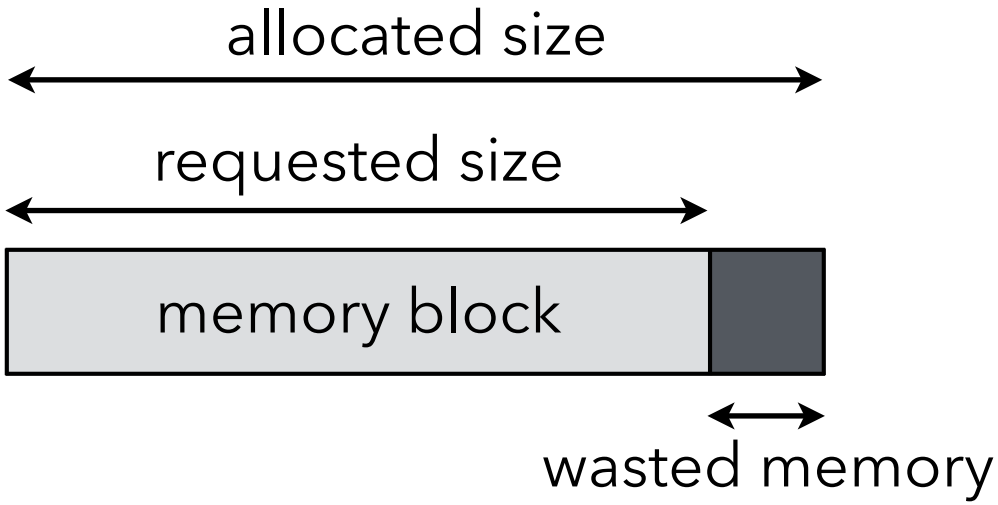
External fragmentation

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



Internal fragmentation

For various reasons – e.g. alignment constraints – the memory manager sometimes allocates slightly more memory than requested by the client. This results in small amounts of wasted memory scattered in the heap.
This phenomenon is called **internal fragmentation**.



GC technique #1: reference counting

Reference counting

The idea of **reference counting** is simple:

> Every object carries a count of the number of pointers that reference it. When this count reaches zero, the object is guaranteed to be unreachable and can be deallocated.

Reference counting requires collaboration from the compiler and/or programmer to make sure that reference counts are properly maintained!

Pros and cons

Reference counting is relatively easy to implement, even as a library. It reclaims memory immediately.

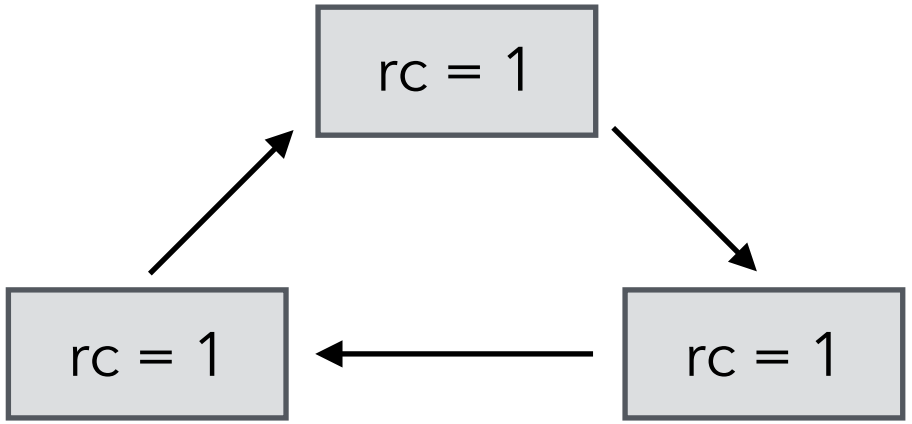
However, it has an important impact on space consumption, and speed of execution: every object must contain a counter, and every pointer write must update it.

But the biggest problem is 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

The problem with cyclic structures is due to the fact that reference counts provide only an approximation of reachability.

In other words, we have:

reference_count(x) = 0 $\Rightarrow$ x is unreachable

but the opposite is not true!

Uses of reference counting

Due to its problem with cyclic structures, reference counting is seldom used.

It is still interesting for systems that do not allow cyclic structures to be created – e.g. hard links in Unix file systems.

It has also been used in combination with a mark & sweep GC, the latter being run infrequently to collect cyclic structures.

GC technique #2: mark & sweep

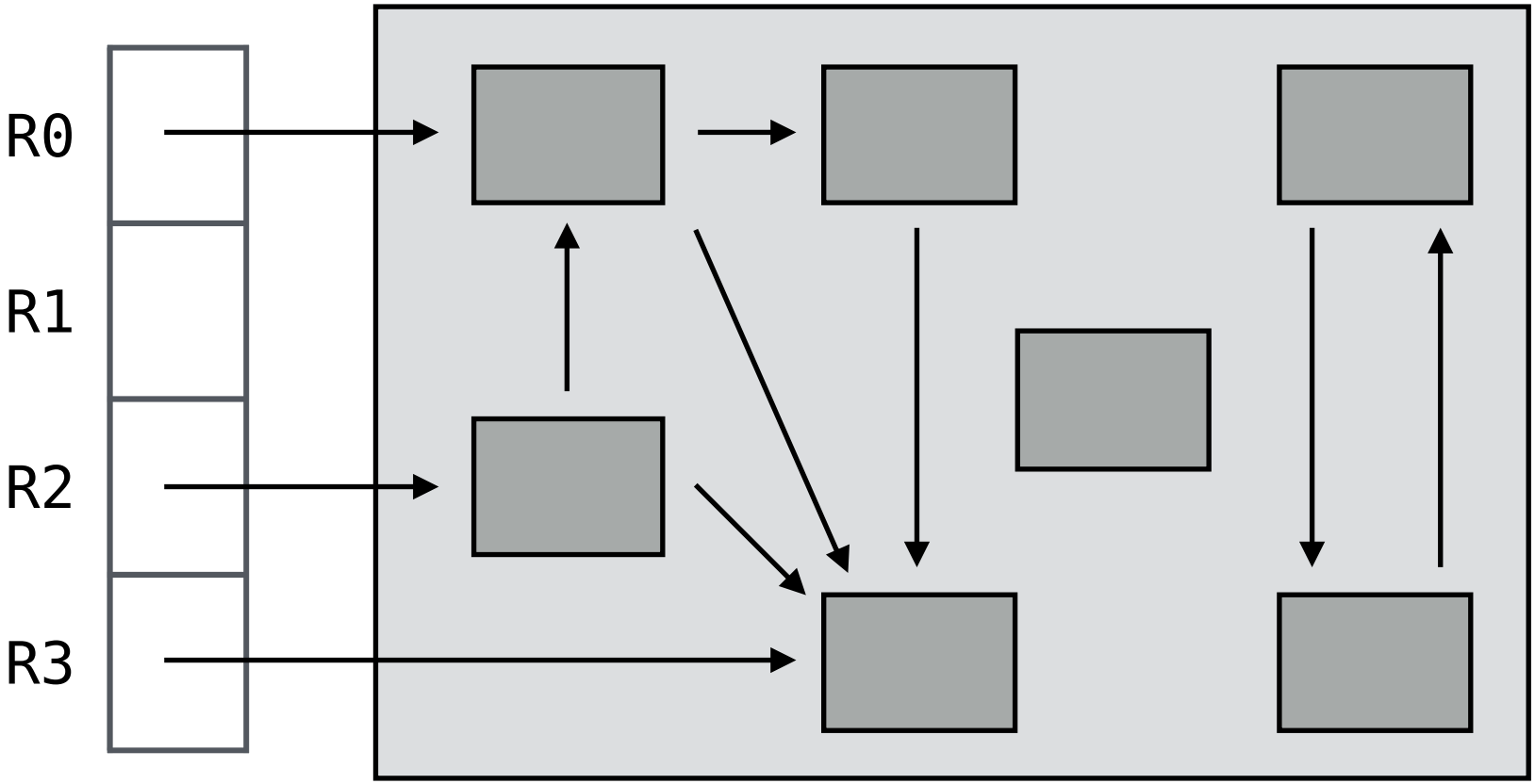
Mark & sweep GC

Mark & sweep garbage collection is a GC technique that proceeds in two successive phases:

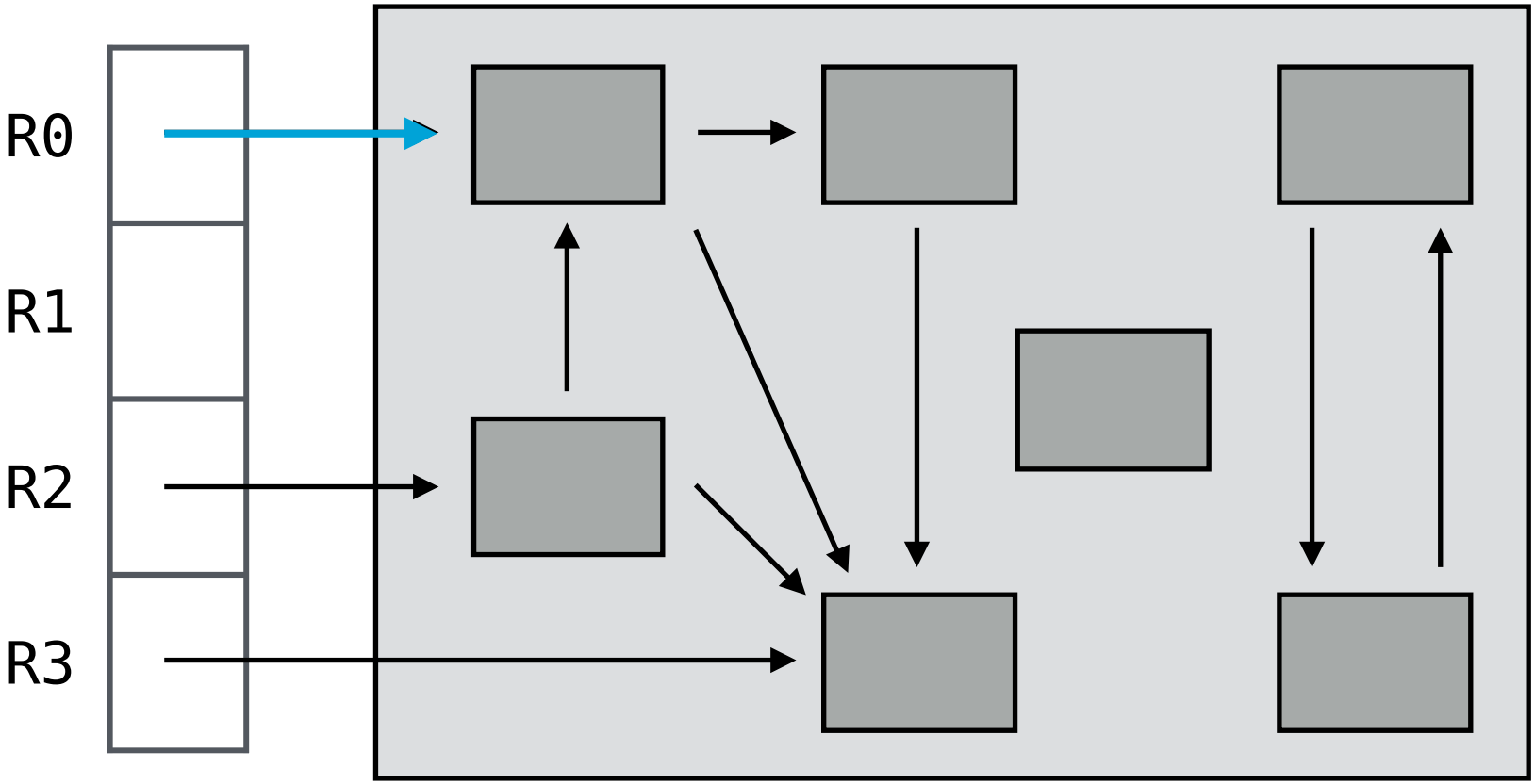
1. in the **marking** phase, the reachability graph is traversed and reachable objects are marked,
2. in the **sweeping** phase, all allocated objects are examined, and unmarked ones are freed.

GC is triggered by a lack of memory, and must complete before the program can be resumed. This is necessary to ensure that the reachability graph is not modified by the program while the GC traverses it.

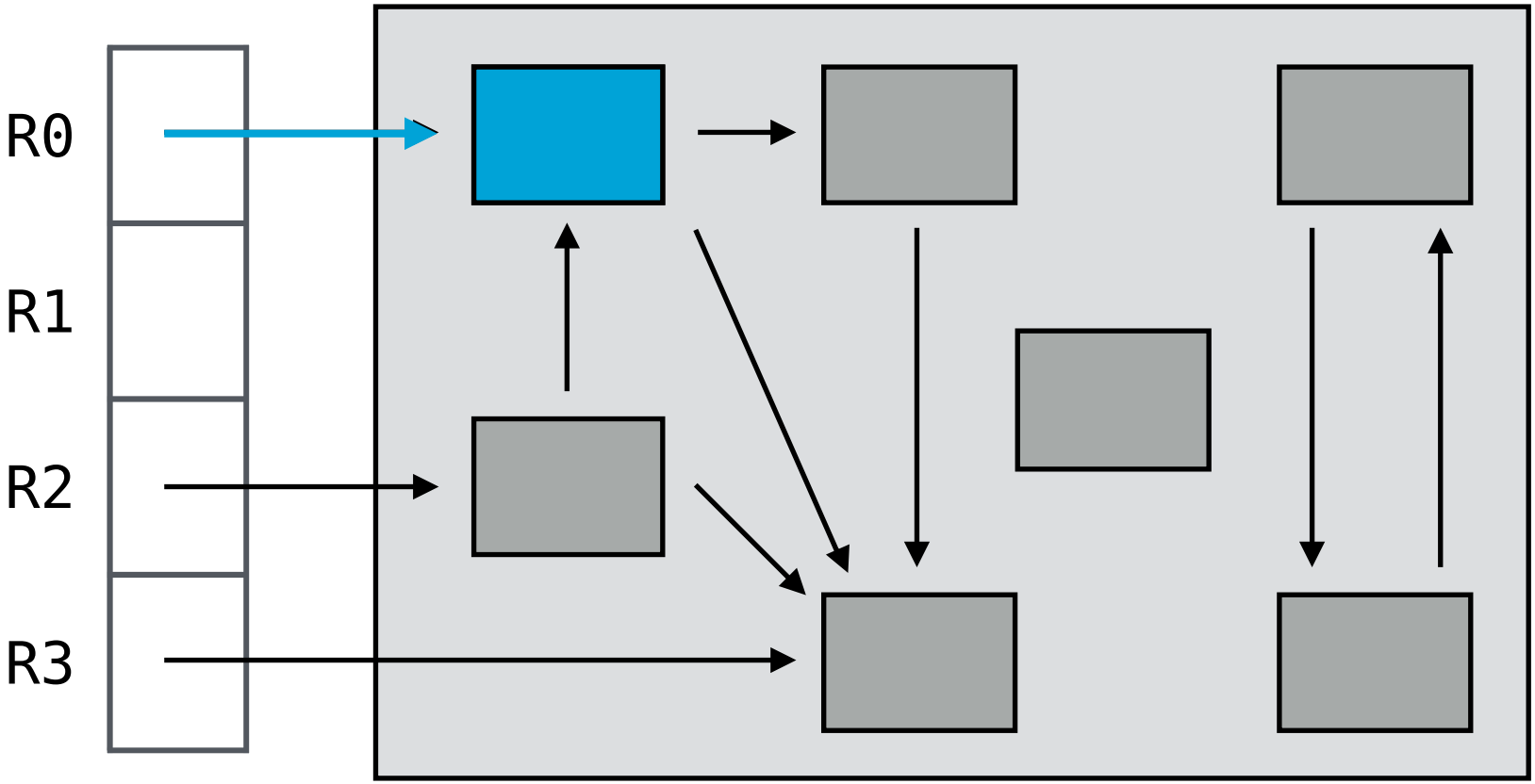
Mark & sweep GC



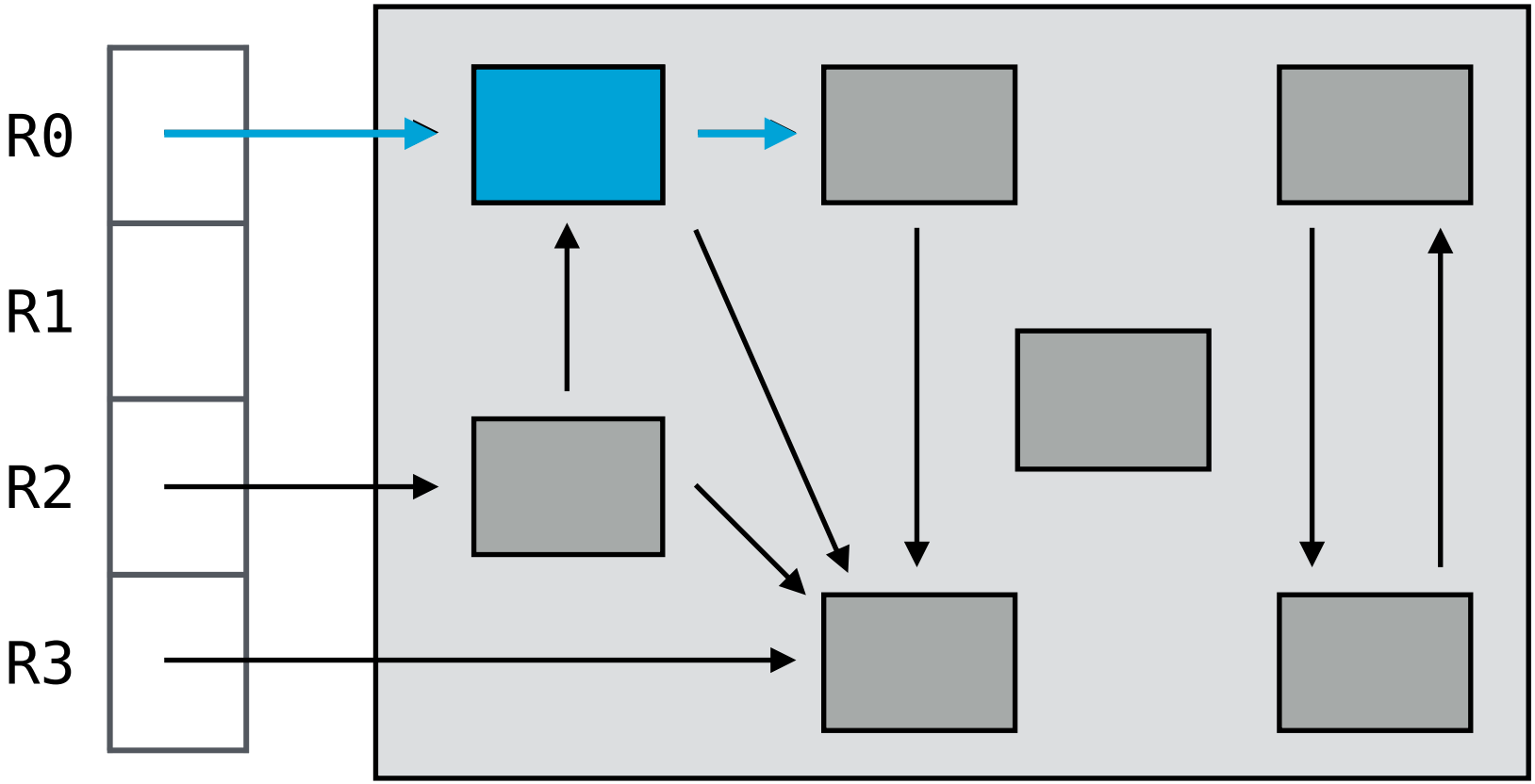
Mark & sweep GC



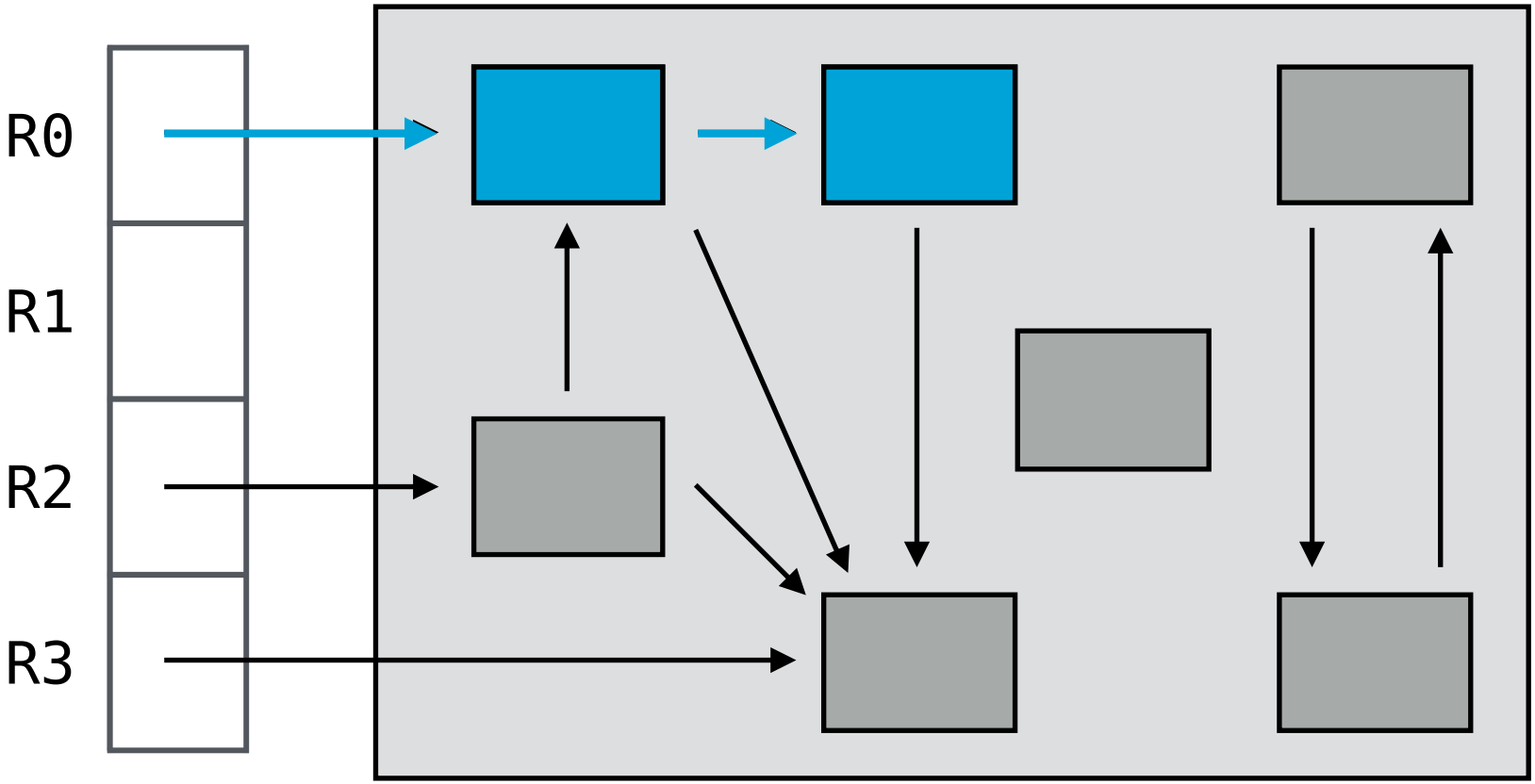
Mark & sweep GC



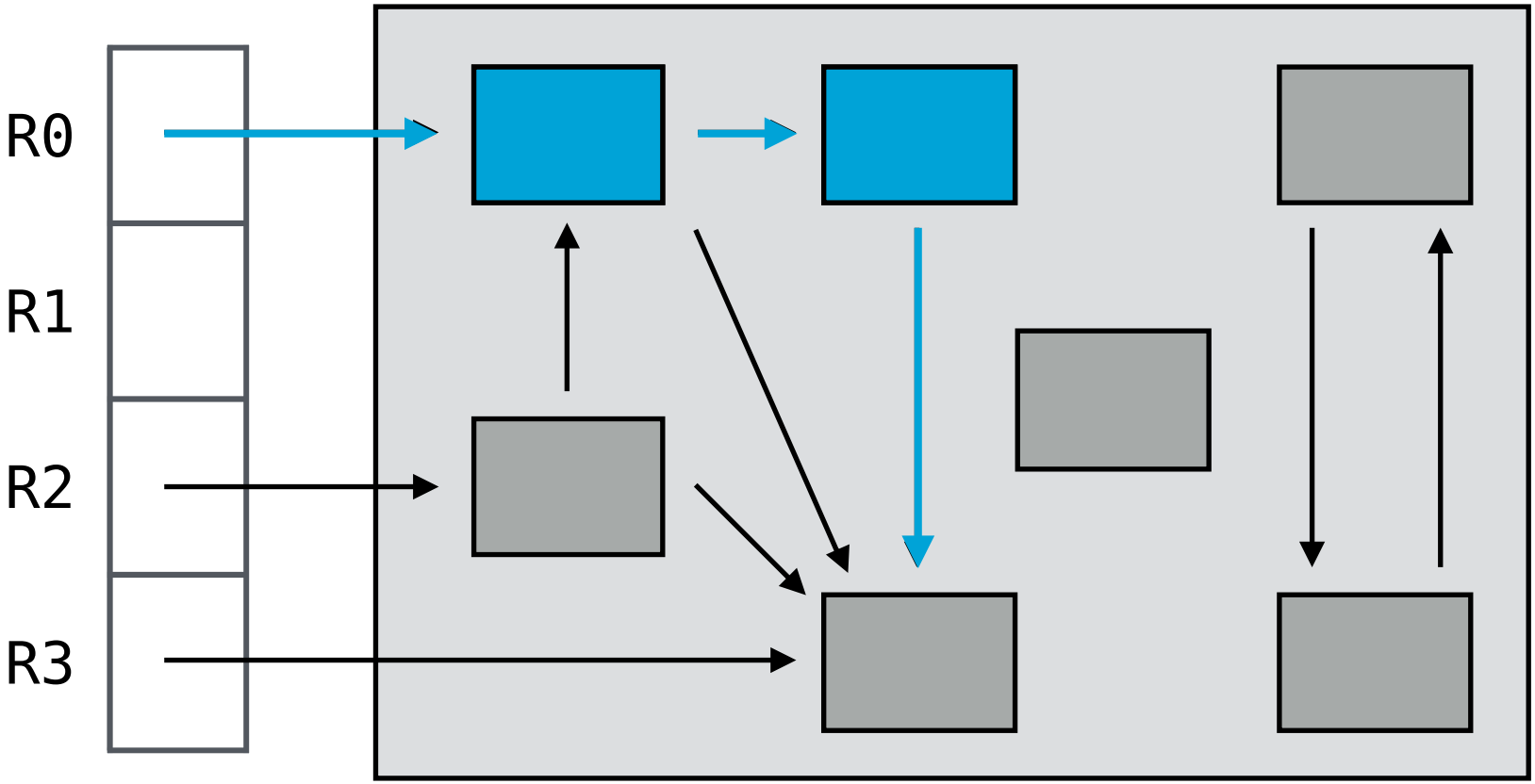
Mark & sweep GC



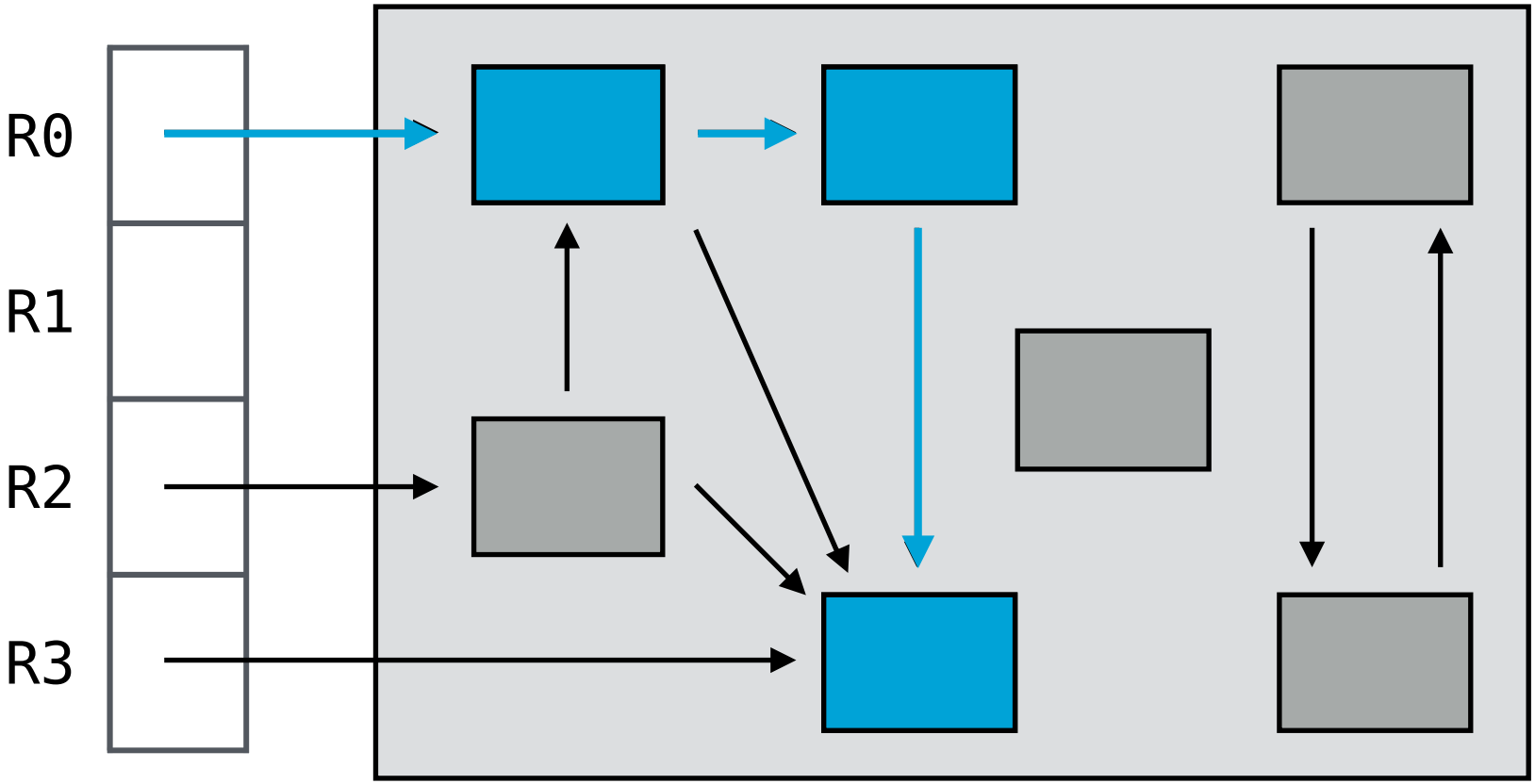
Mark & sweep GC



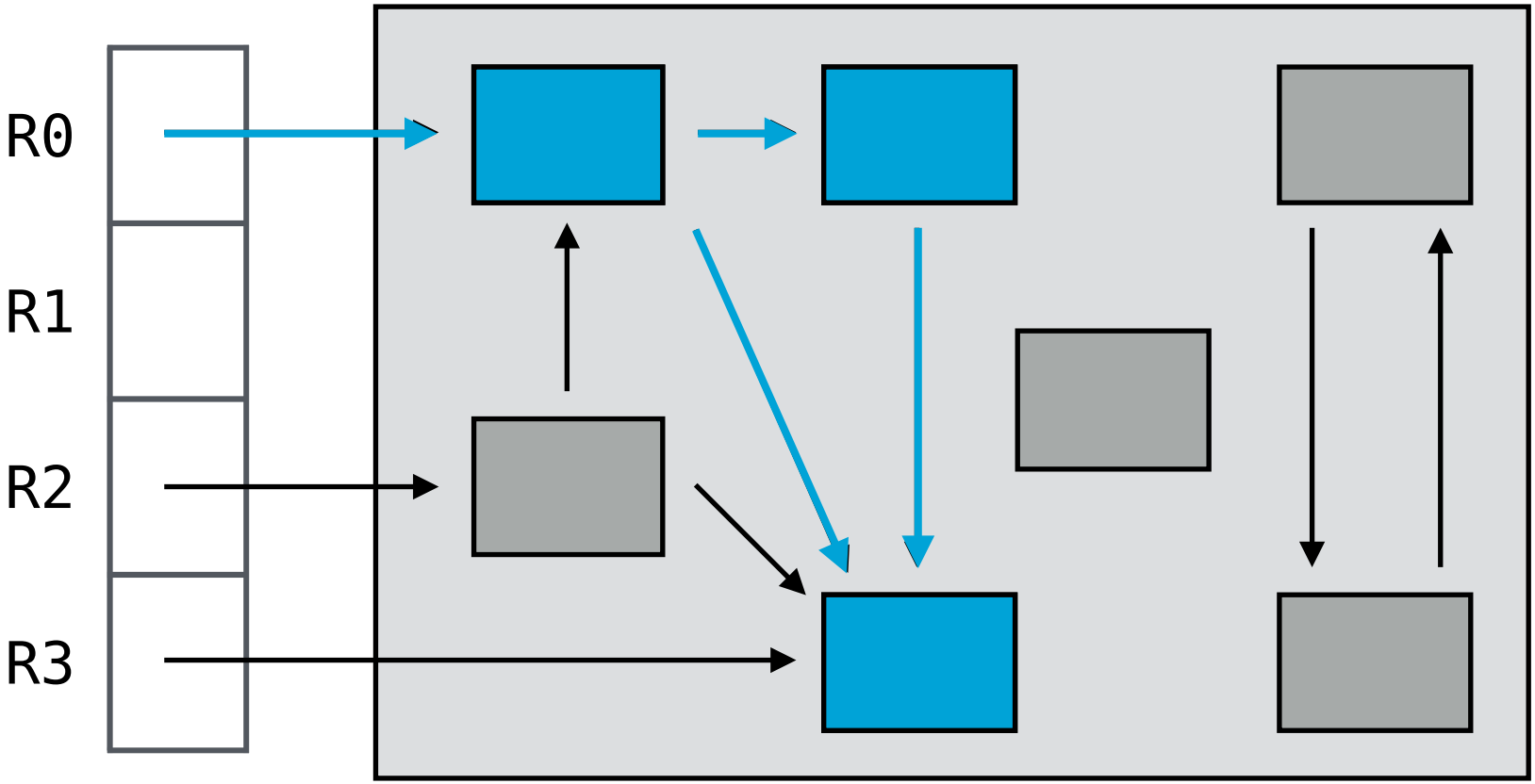
Mark & sweep GC



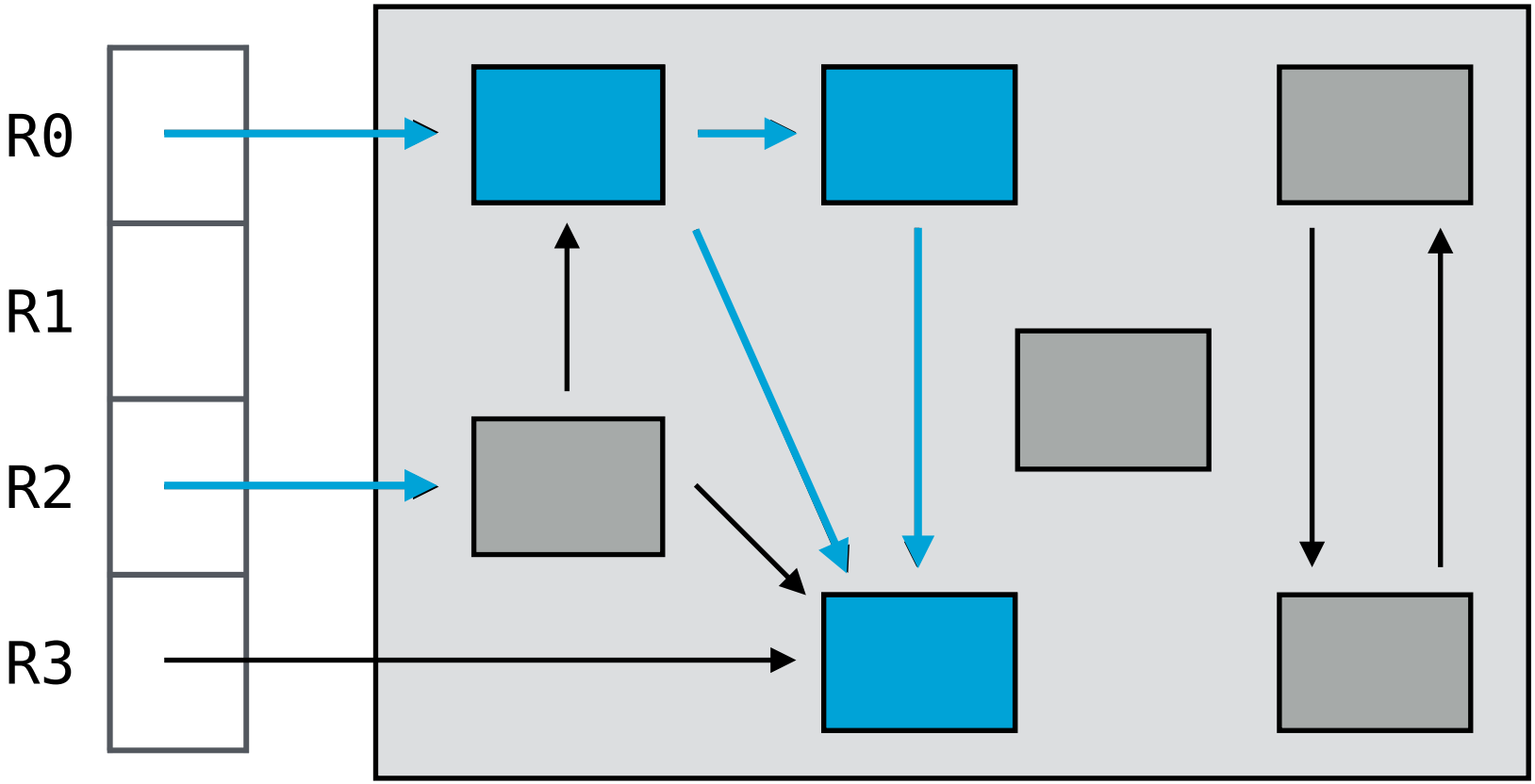
Mark & sweep GC



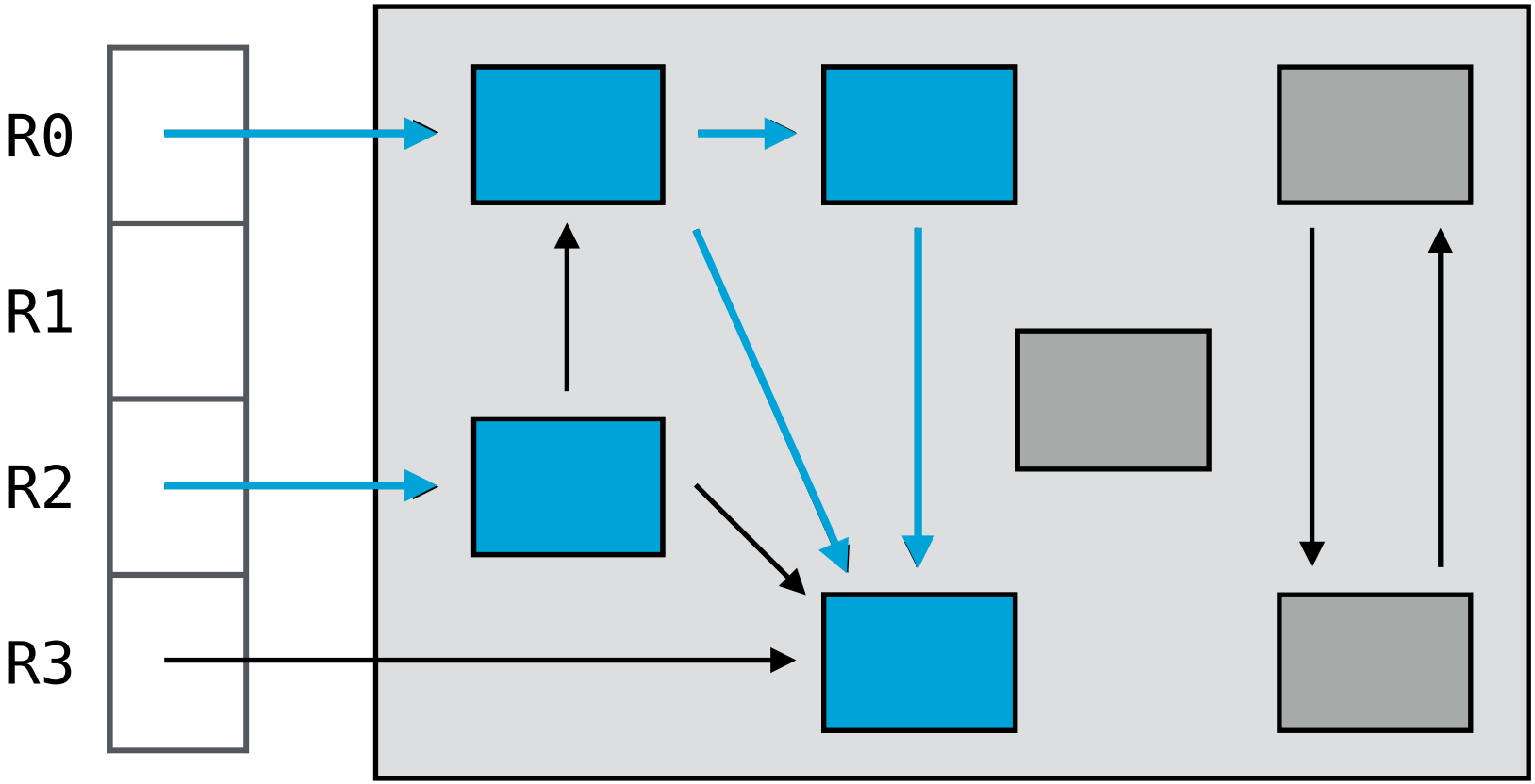
Mark & sweep GC



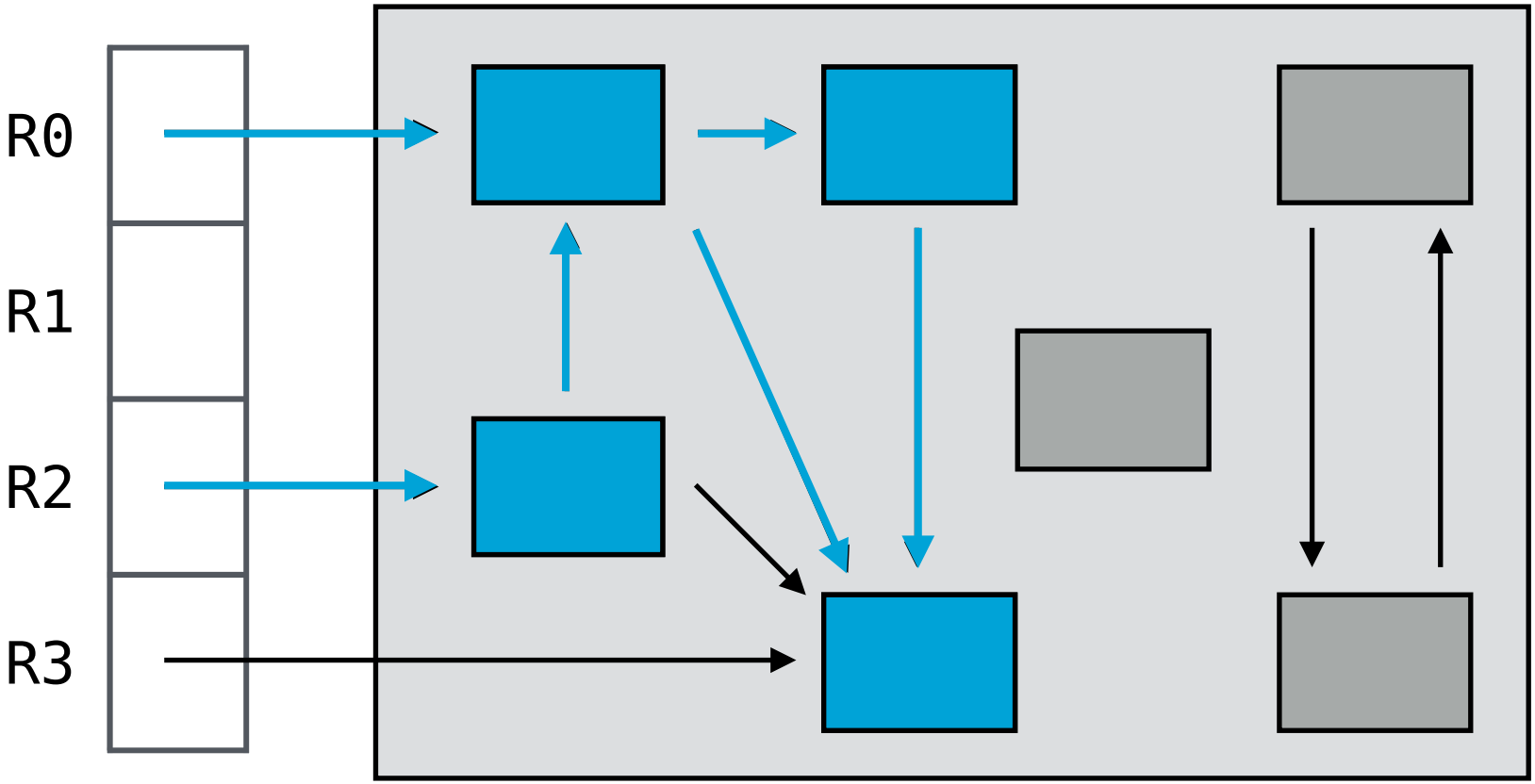
Mark & sweep GC



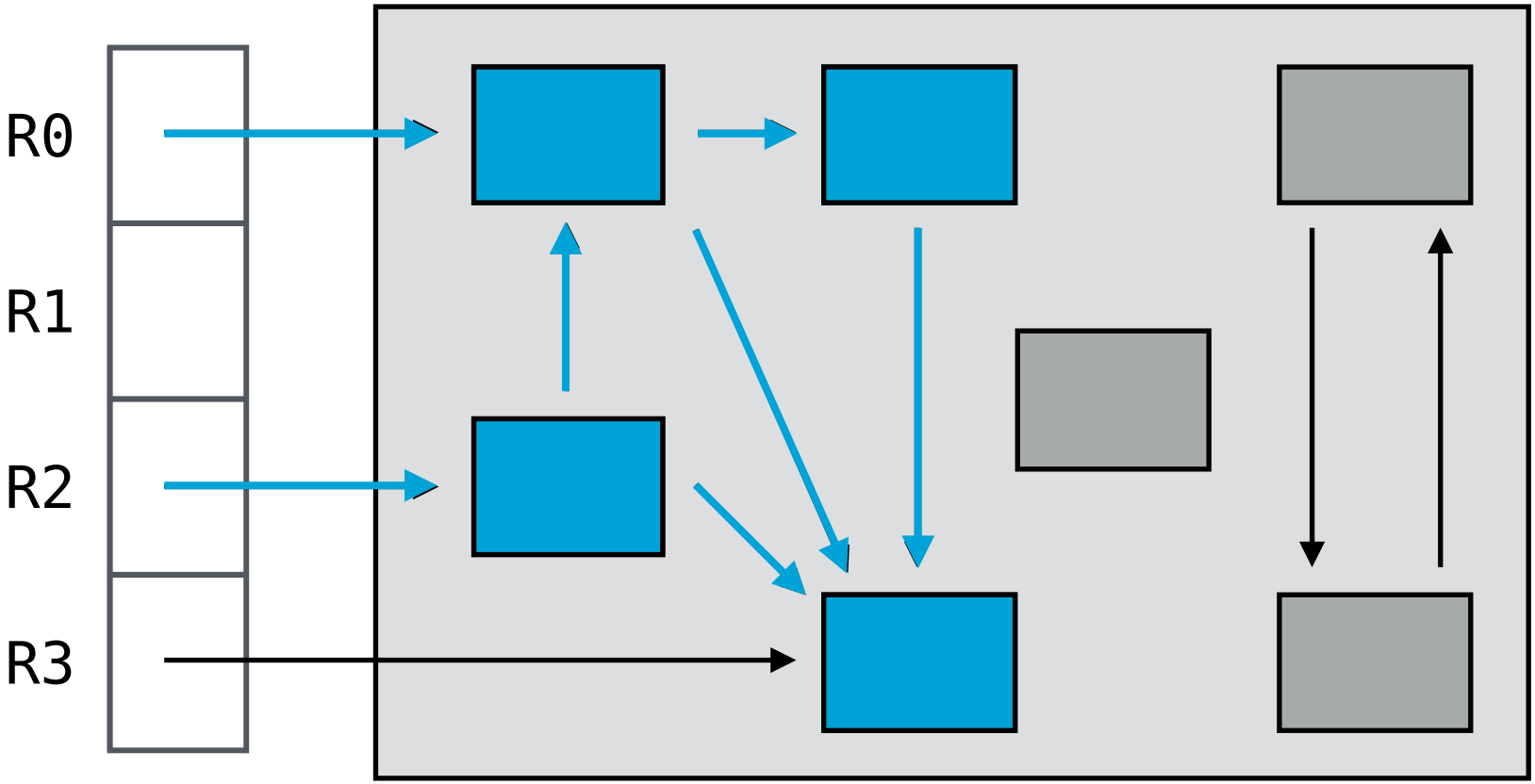
Mark & sweep GC



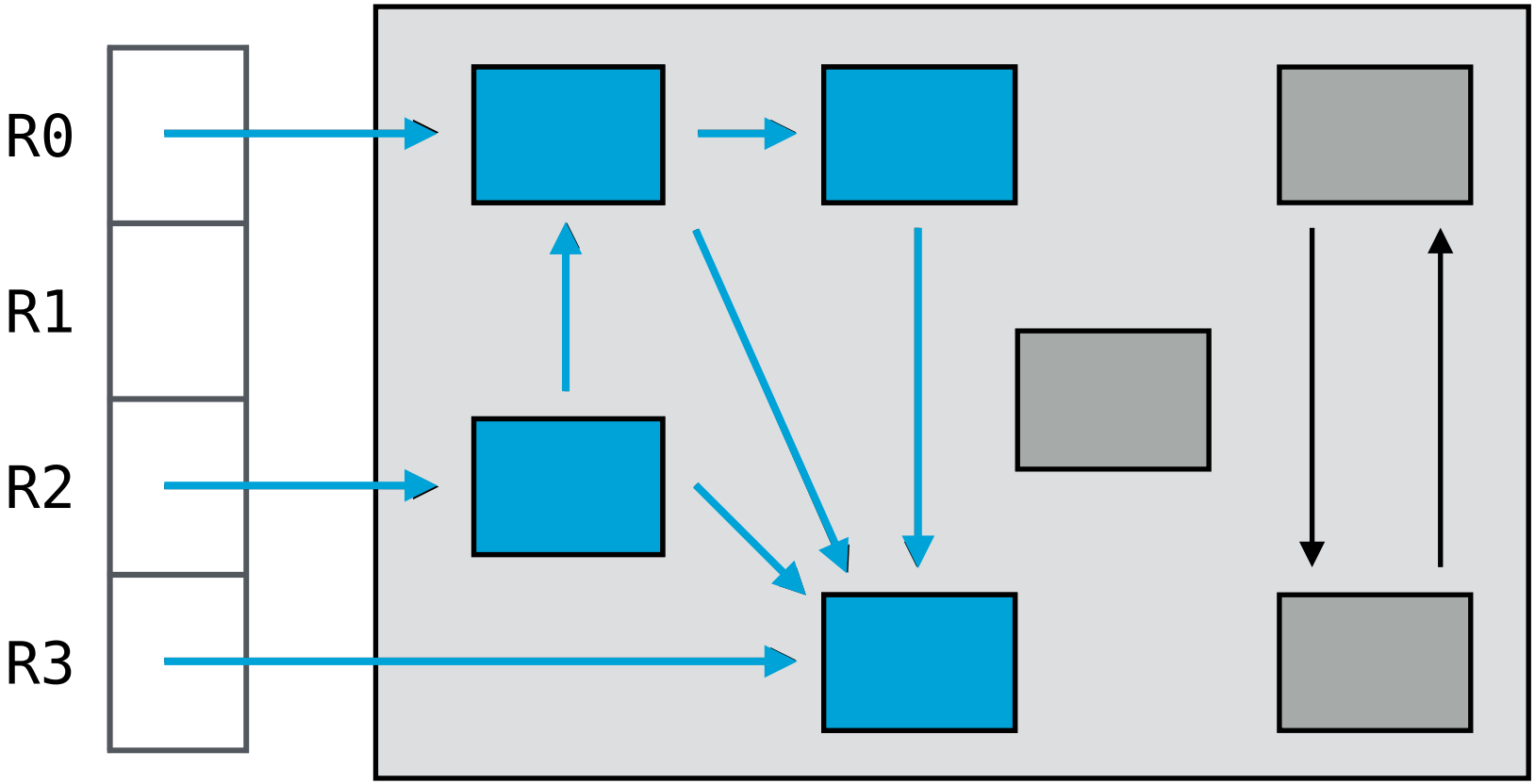
Mark & sweep GC



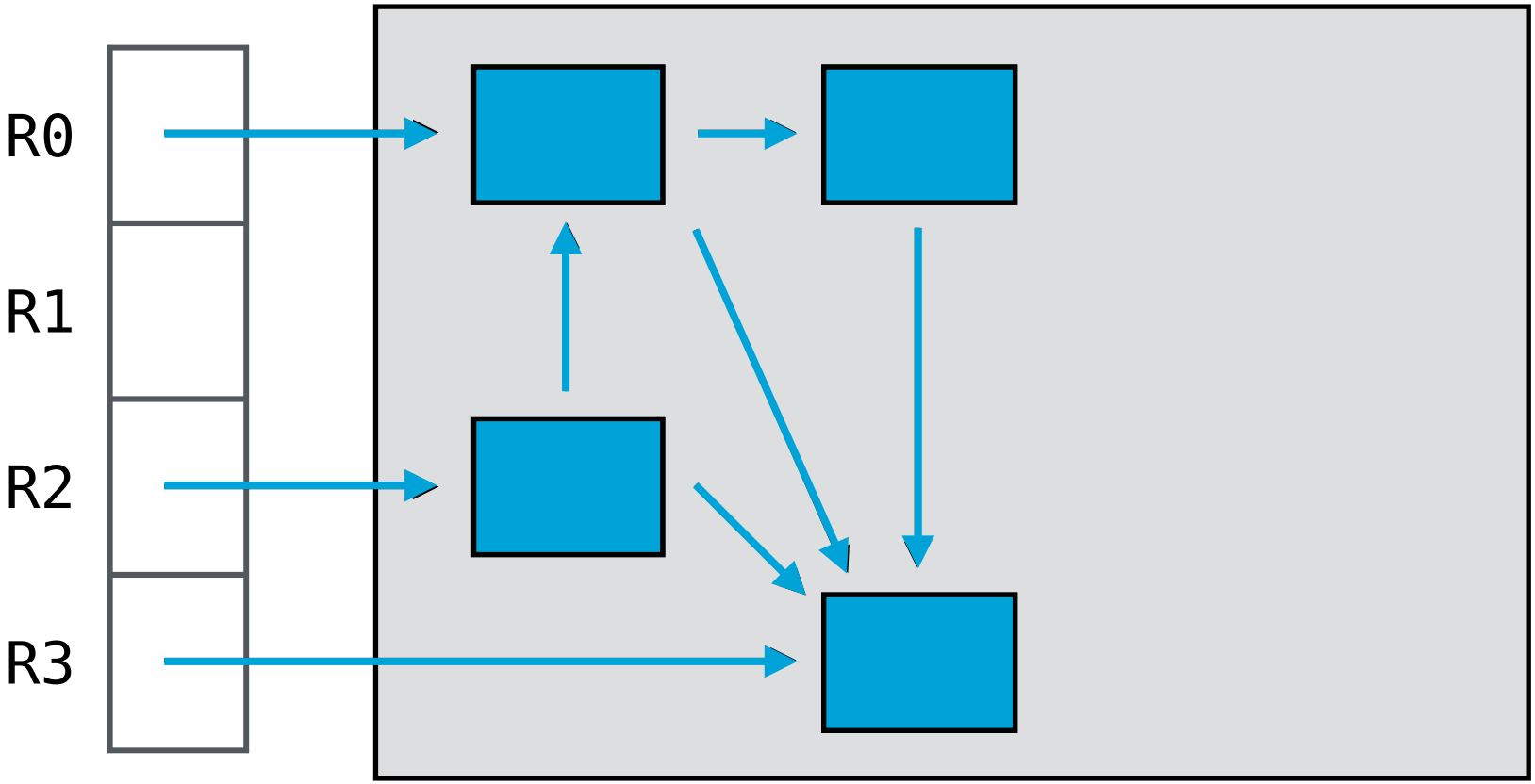
Mark & sweep GC



Mark & sweep GC



Mark & sweep GC



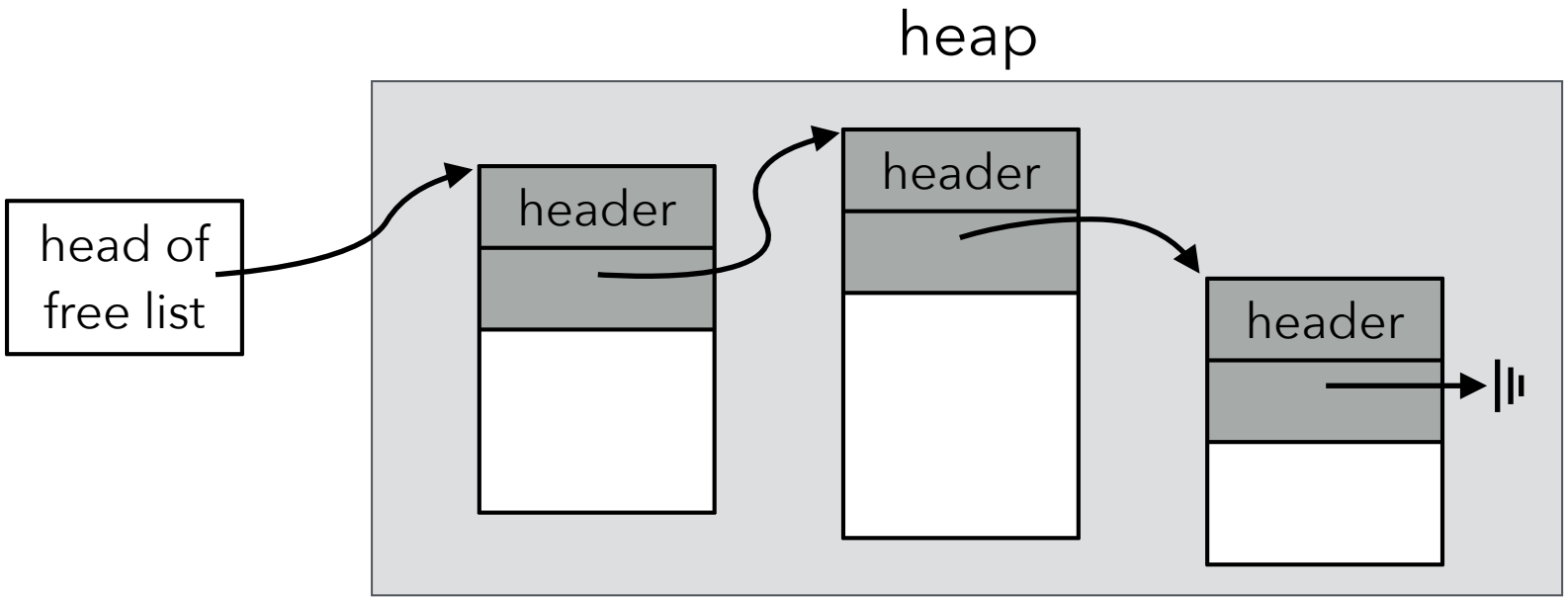
Marking objects

Reachable objects must be marked in some way.
Since only one bit is required for the mark, it is possible to store it in the block header, along with the size.
For example, if the system guarantees that all blocks have an even size, then the least significant bit (LSB) of the block size can be used for marking.
It is also possible to use “external” bit maps – stored in a memory area that is private to the GC – to store mark bits.

Free list

In a mark & sweep GC, free blocks are not contiguous in memory. Therefore, they have to be stored in a data structure usually known as the **free list**.

In a simple GC, this free list could effectively be a linked list. Since the free blocks are, by definition, not used by the program, the links can be stored in the blocks themselves!



Allocation policy

When a block of memory is requested, there are in general many free blocks big enough to satisfy the request.

An **allocation policy** must therefore be used to decide which of those candidates to choose. A good allocation policy should minimize fragmentation while being fast to implement.

The most commonly used policies are:

- **first fit**, which uses the first suitable block,
- **best fit**, which uses the smallest block big enough to satisfy the request.

Splitting and coalesing

When the memory manager has found a free block big enough to satisfy an allocation request, it is possible for that block to be bigger than the size requested. In that case, the block must be **split** in two parts: one part is returned to the client, while the other is put back into the free list.

The opposite must be done during deallocation: if the block being freed is adjacent to one or two other free blocks, then they all should be **coalesced** to form a bigger free block.

Reachability graph traversal

The mark phase requires a depth-first traversal of the reachabilty graph. This is usually implemented by recursion. Recursive function calls use stack space, and since the depth of the reachability graph is not bounded, the GC can overflow its stack!

Several techniques – not presented here – have been developed to either recover from those overflows, or avoid them altogether by storing the stack in the objects being traced.

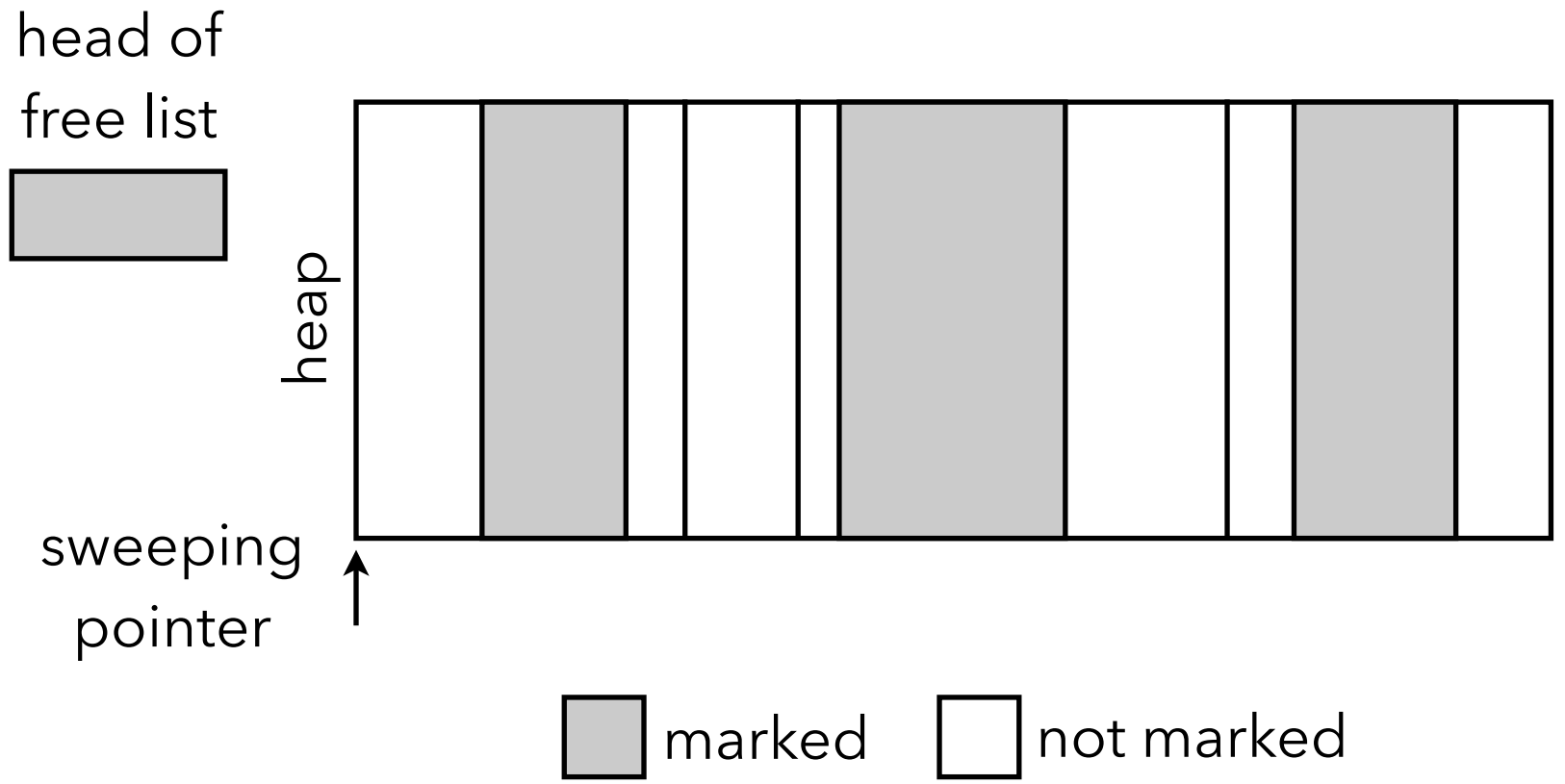
Sweeping objects

Once the mark phase has terminated, all allocated but unmarked objects can be freed. This is the job of the sweep phase, which traverses the whole heap sequentially, looking for unmarked objects and adding them to the free list.

Notice that unreachable objects cannot become reachable again. It is therefore possible to sweep objects on demand, to only fulfill the current memory need. This is called **lazy sweep**.

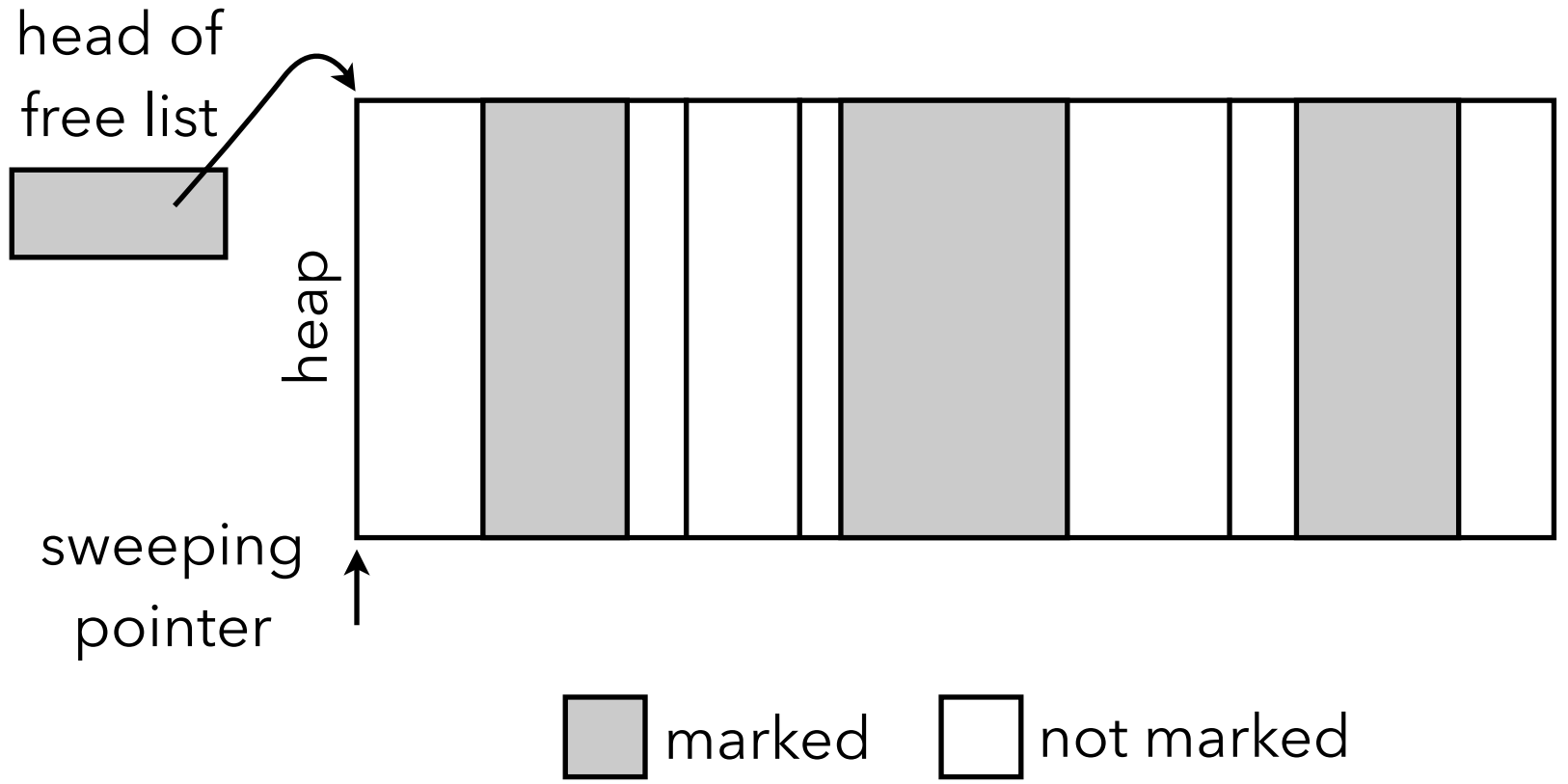
Sweeping and coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



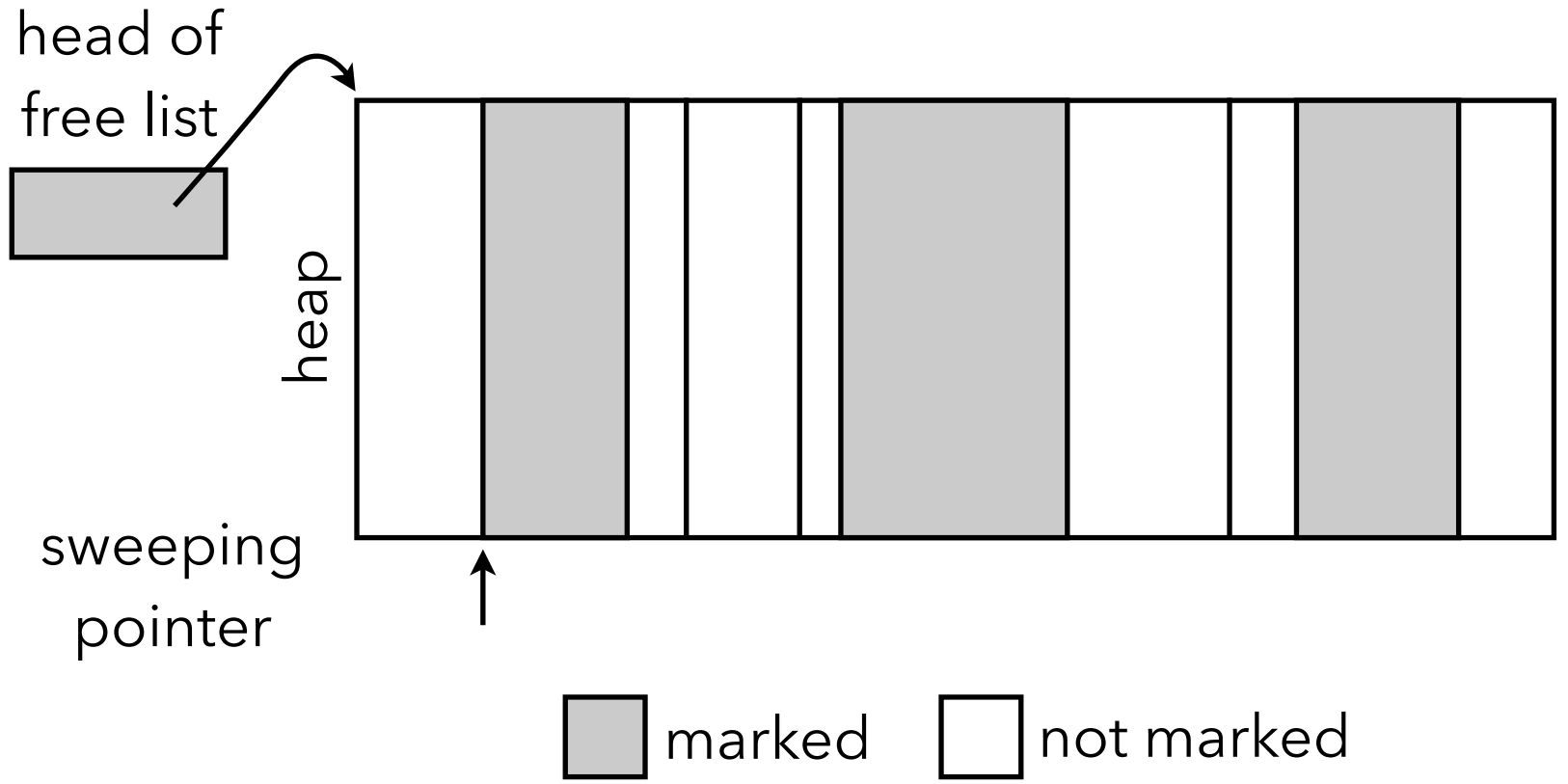
Sweeping and coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



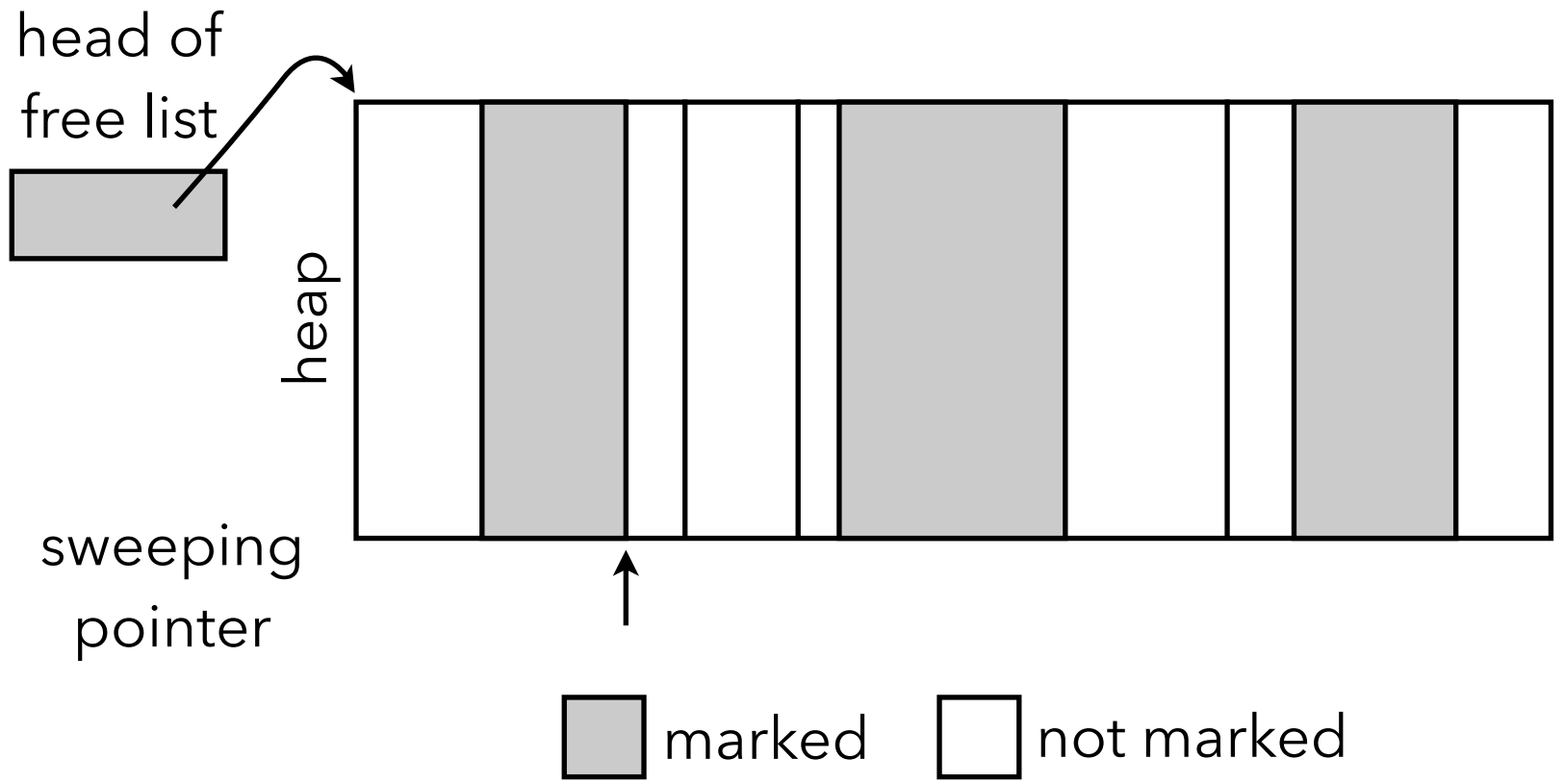
Sweeping and coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



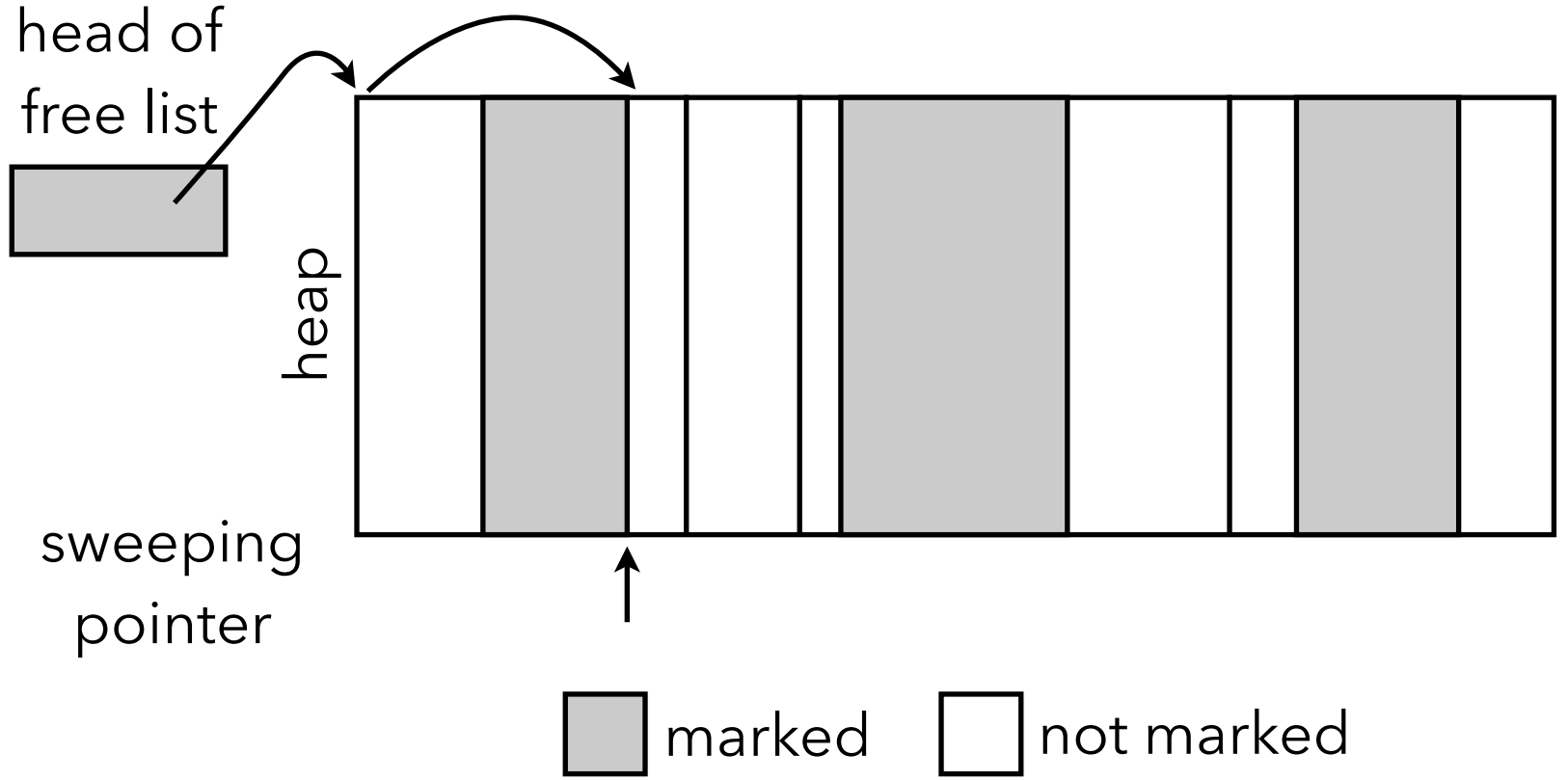
Sweeping and coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



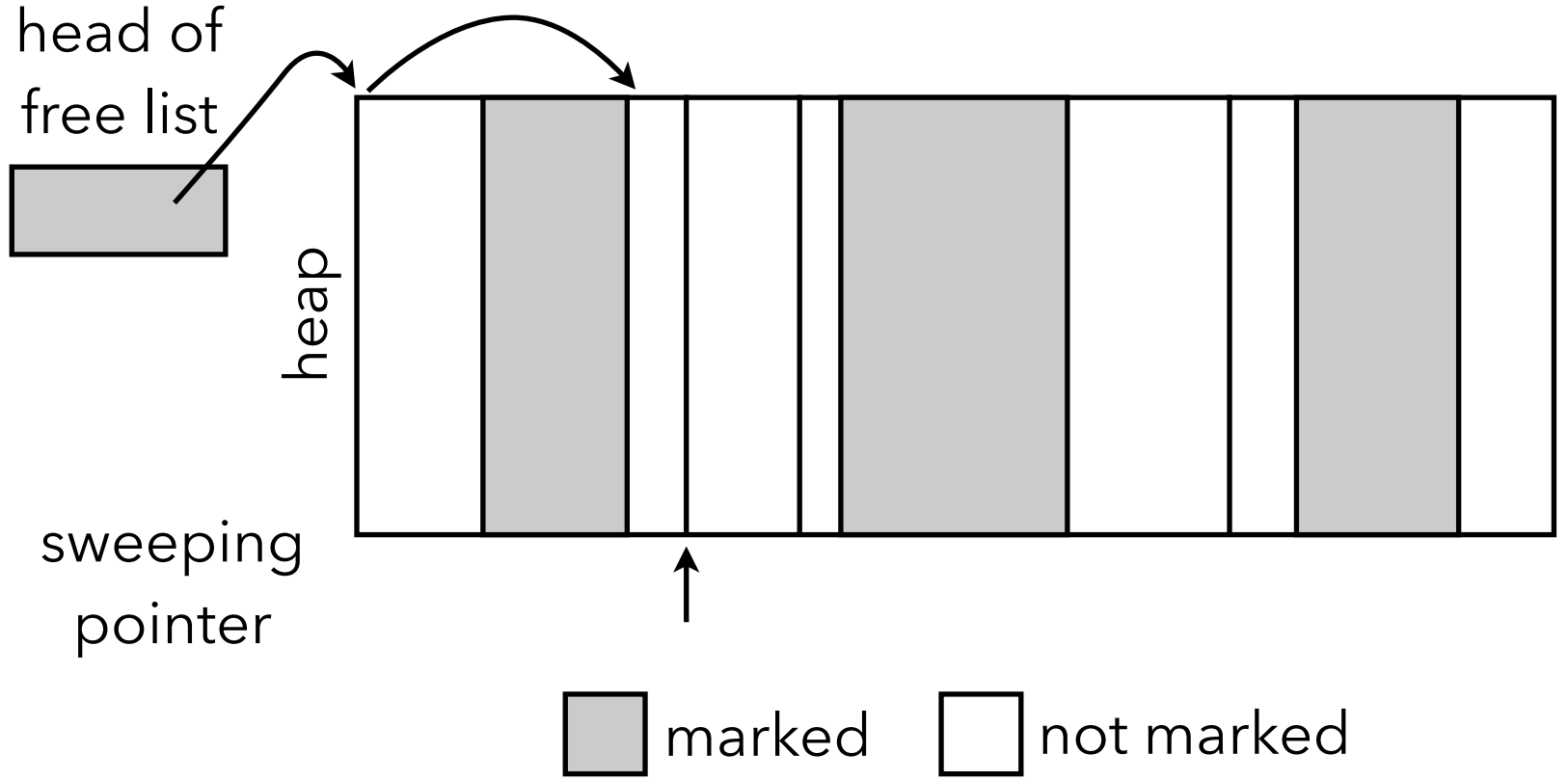
Sweeping and coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



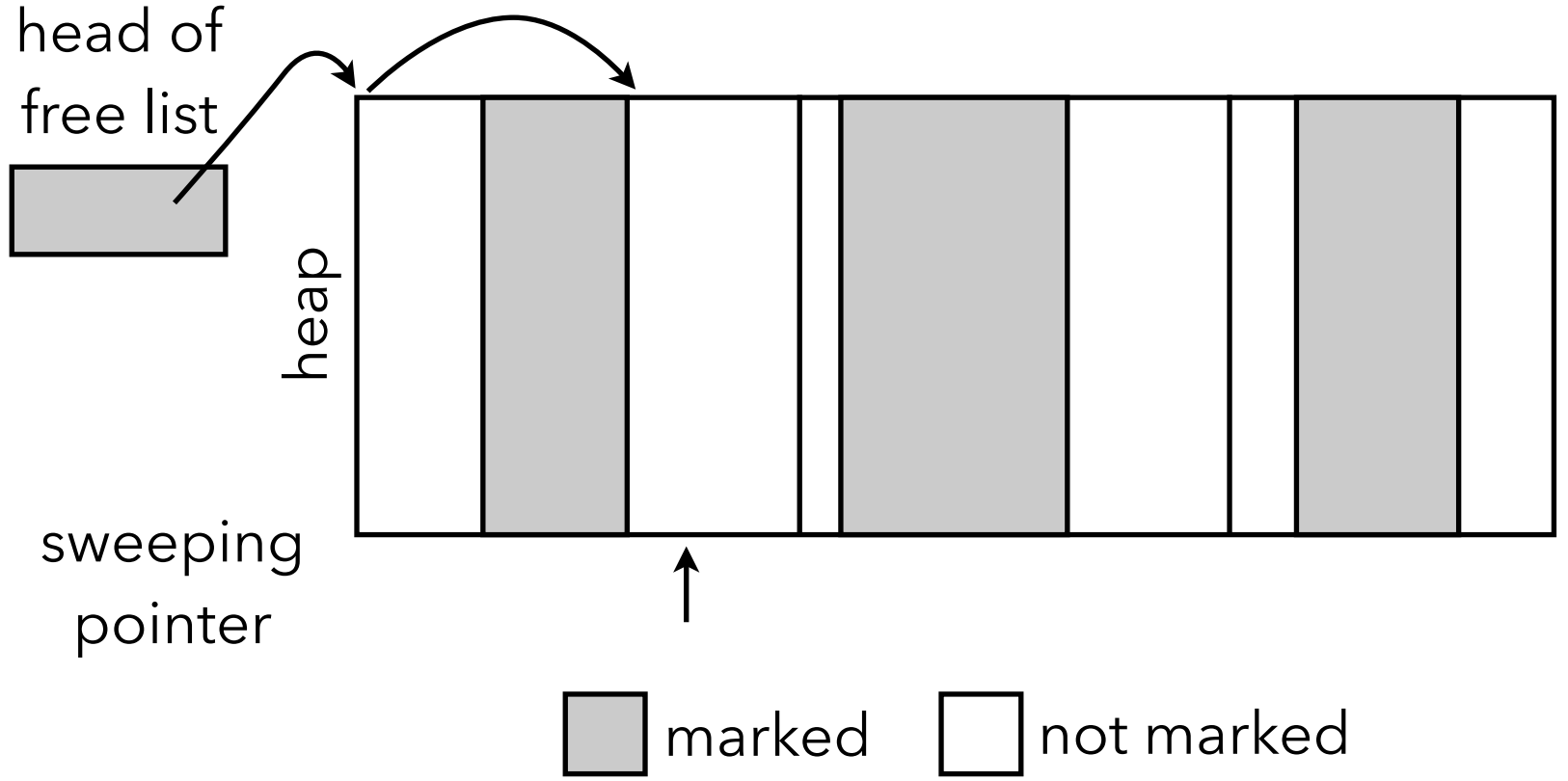
Sweeping and coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



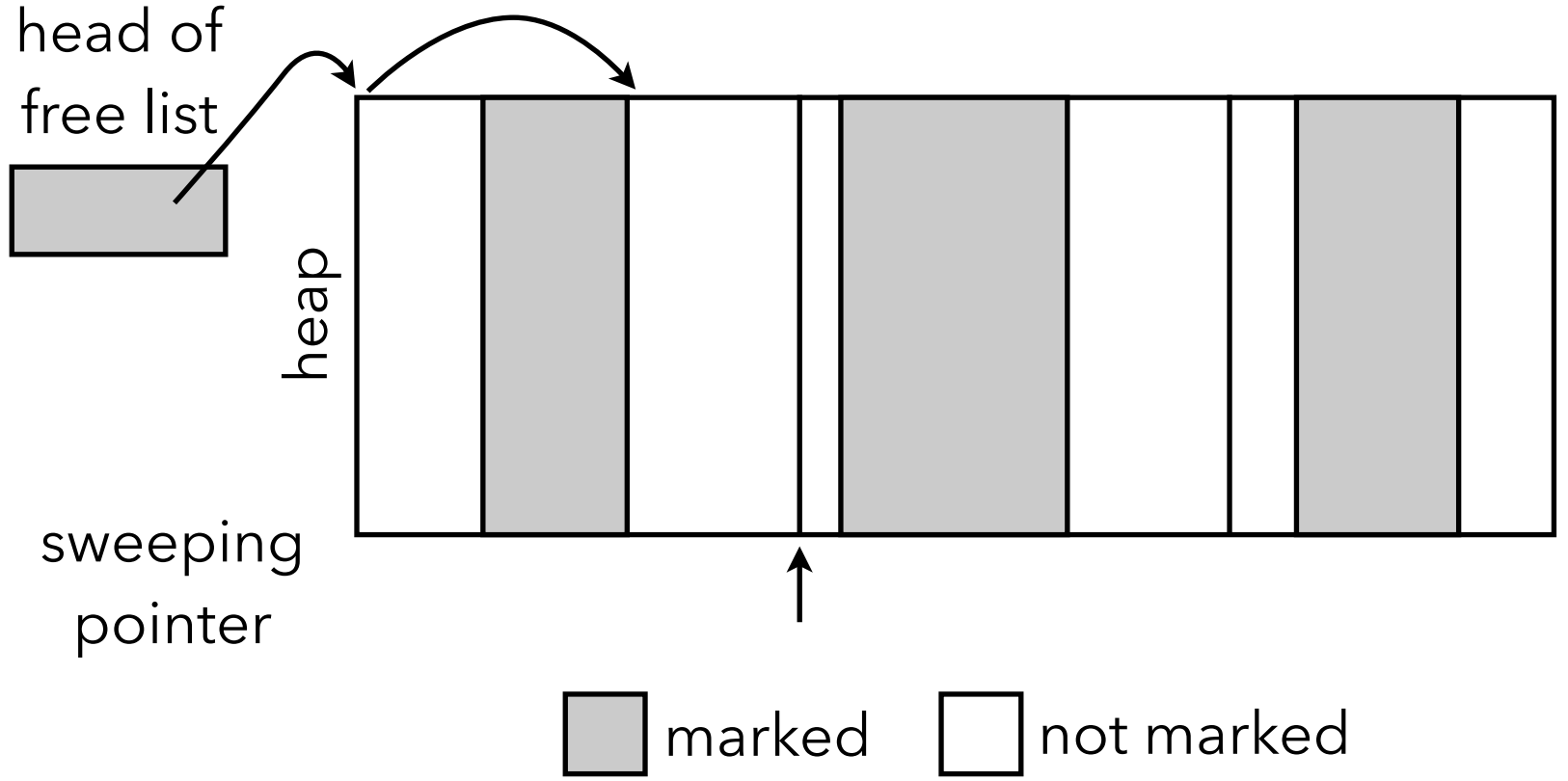
Sweeping and coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



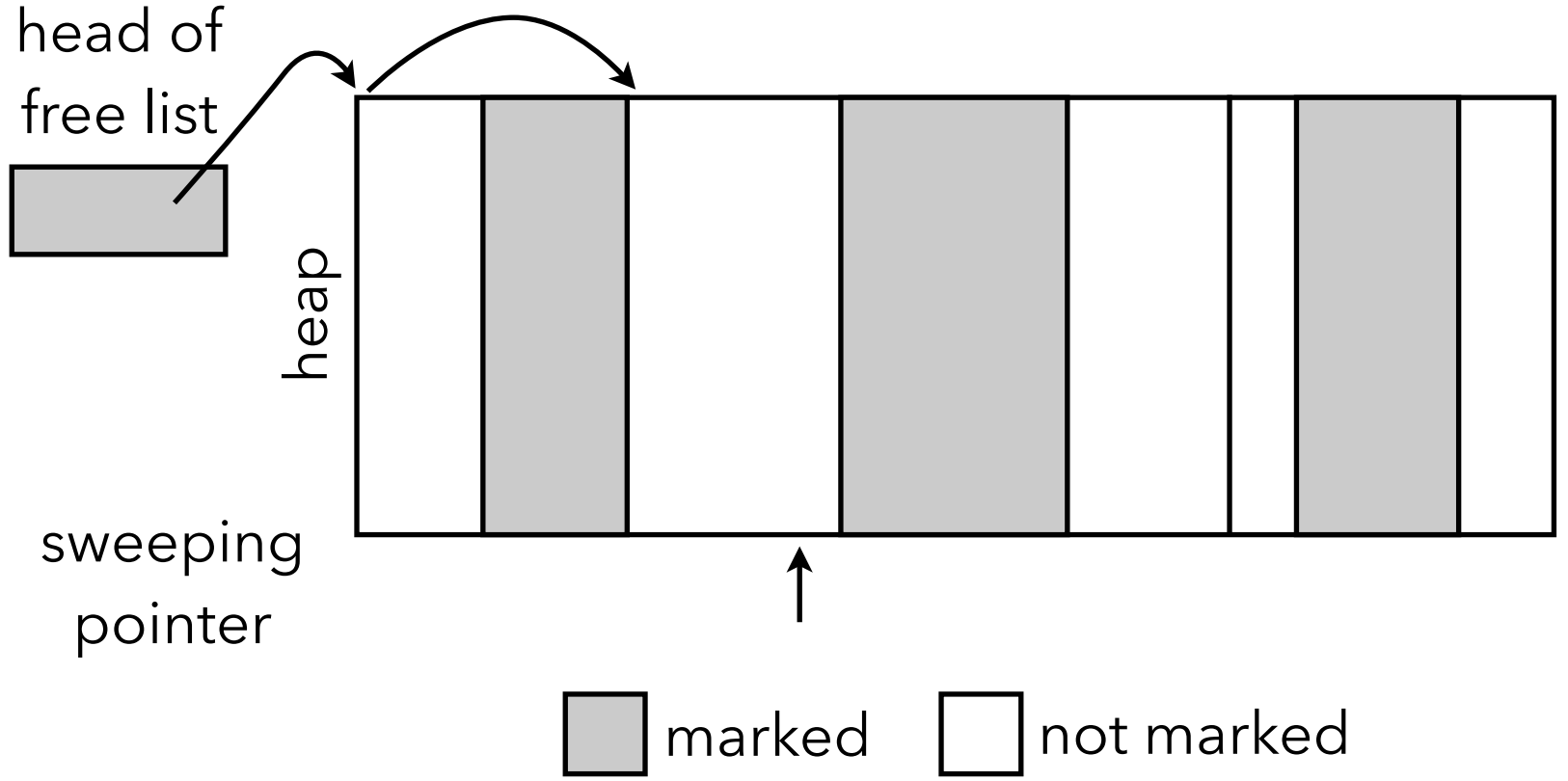
Sweeping and coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



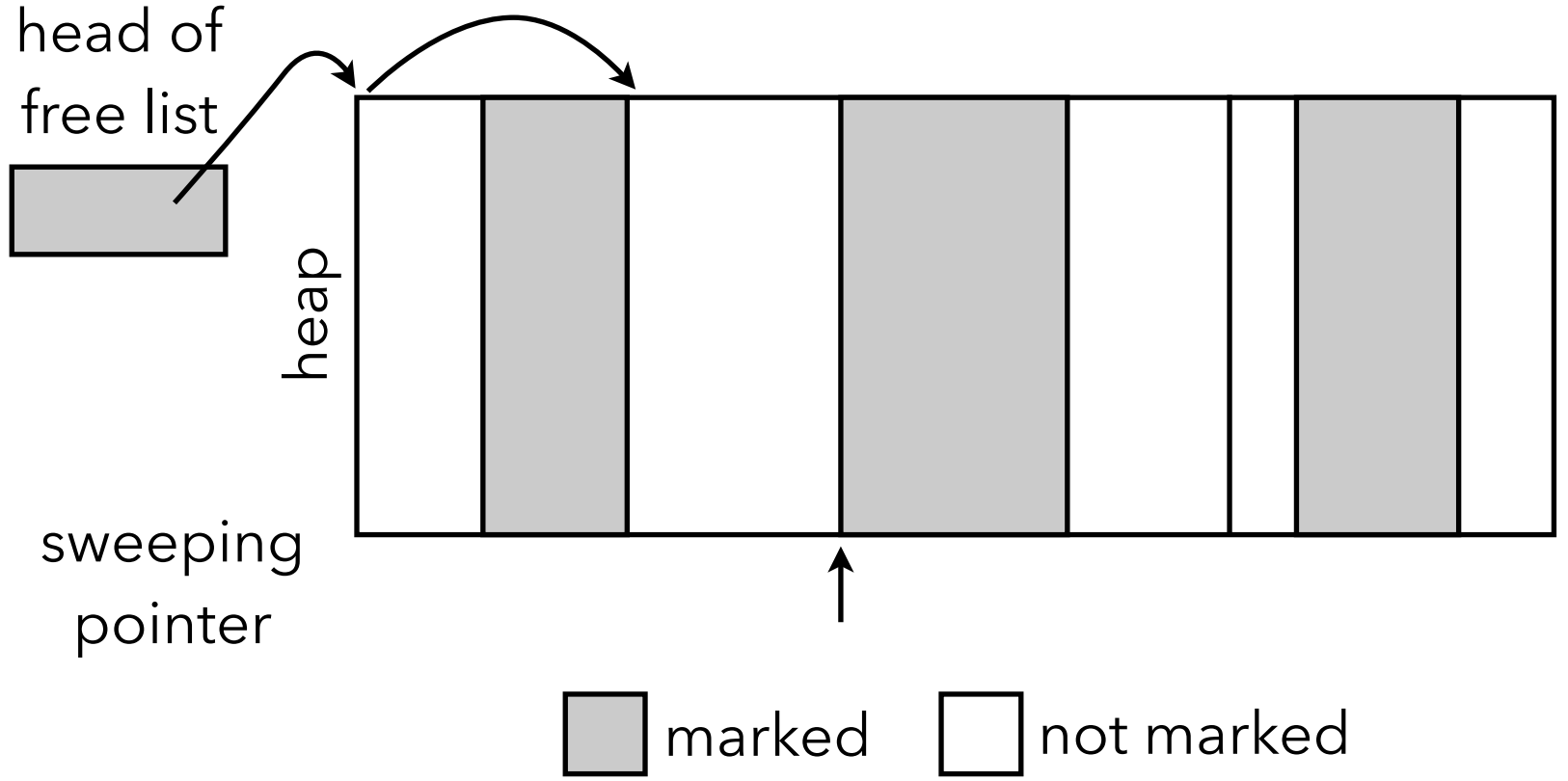
Sweeping and coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



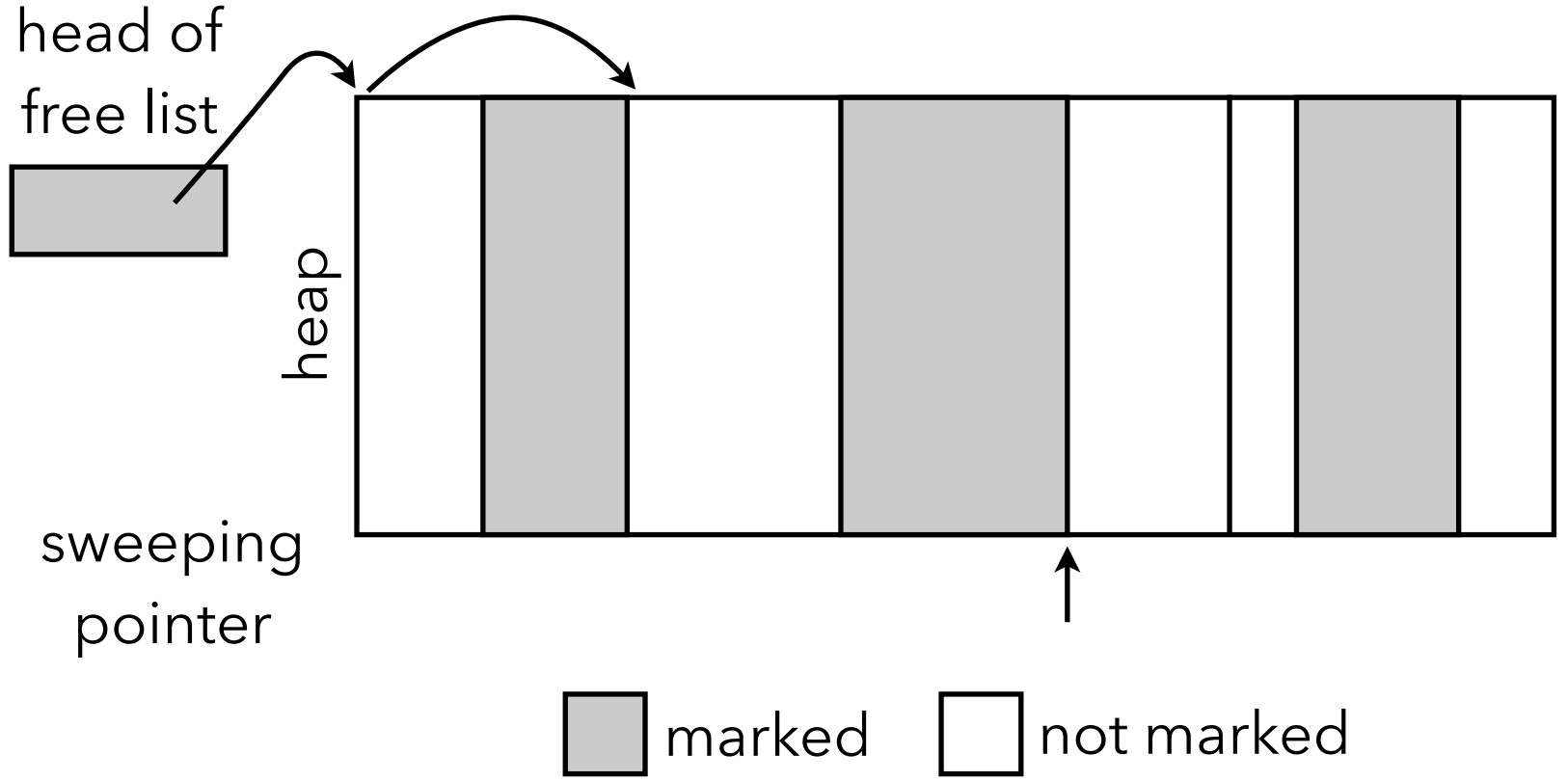
Sweeping and coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



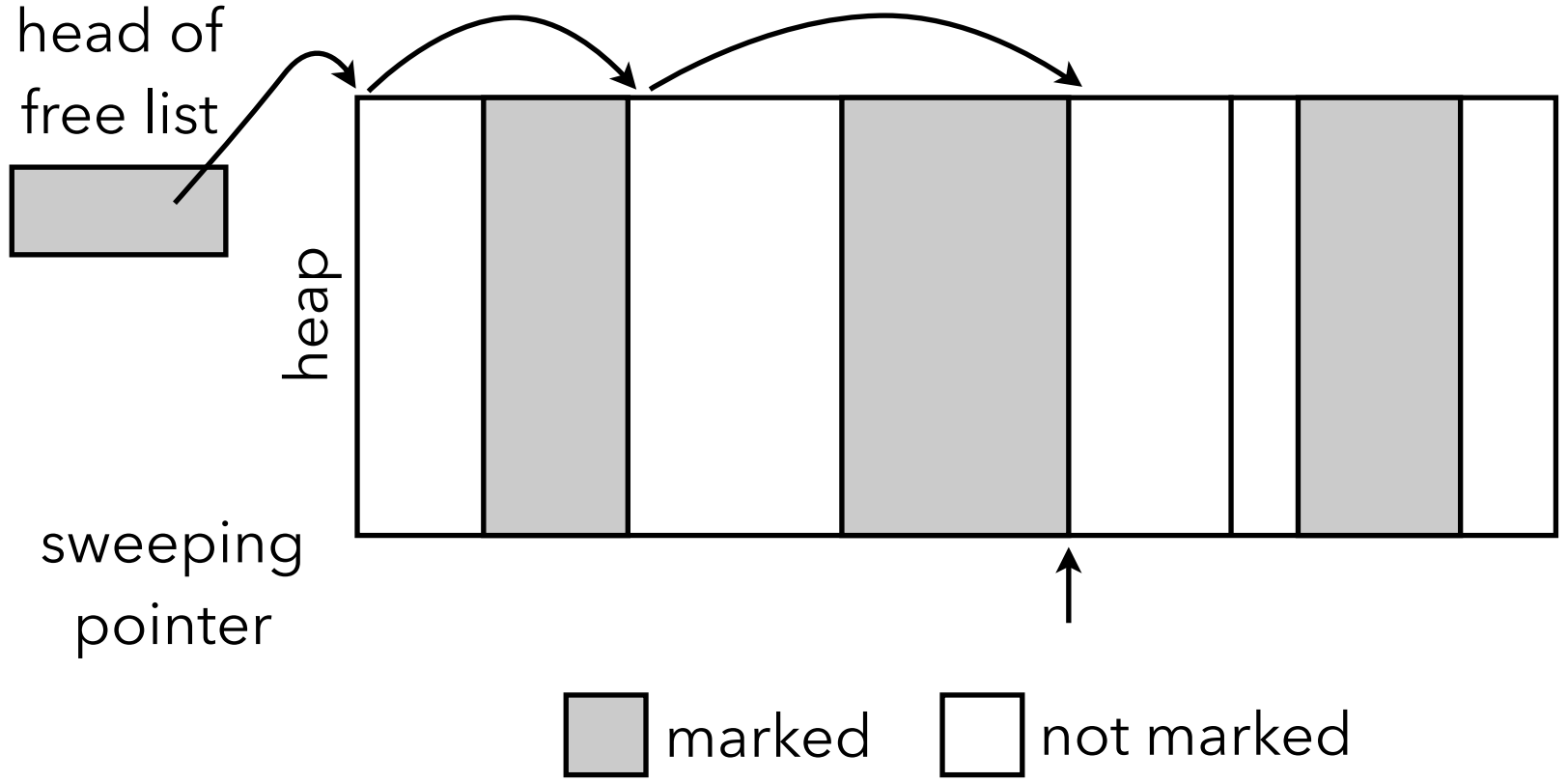
Sweeping and coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



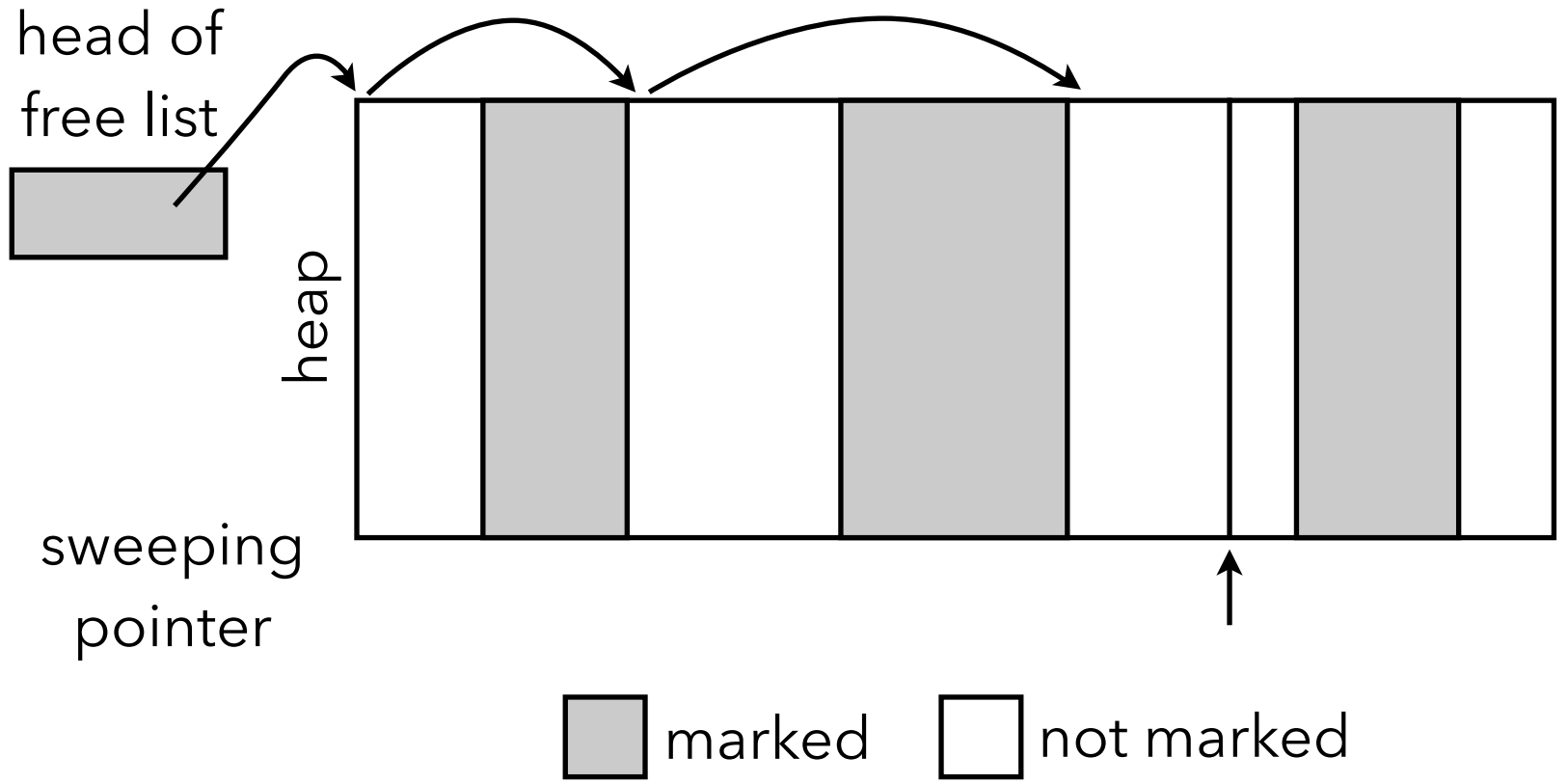
Sweeping and coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



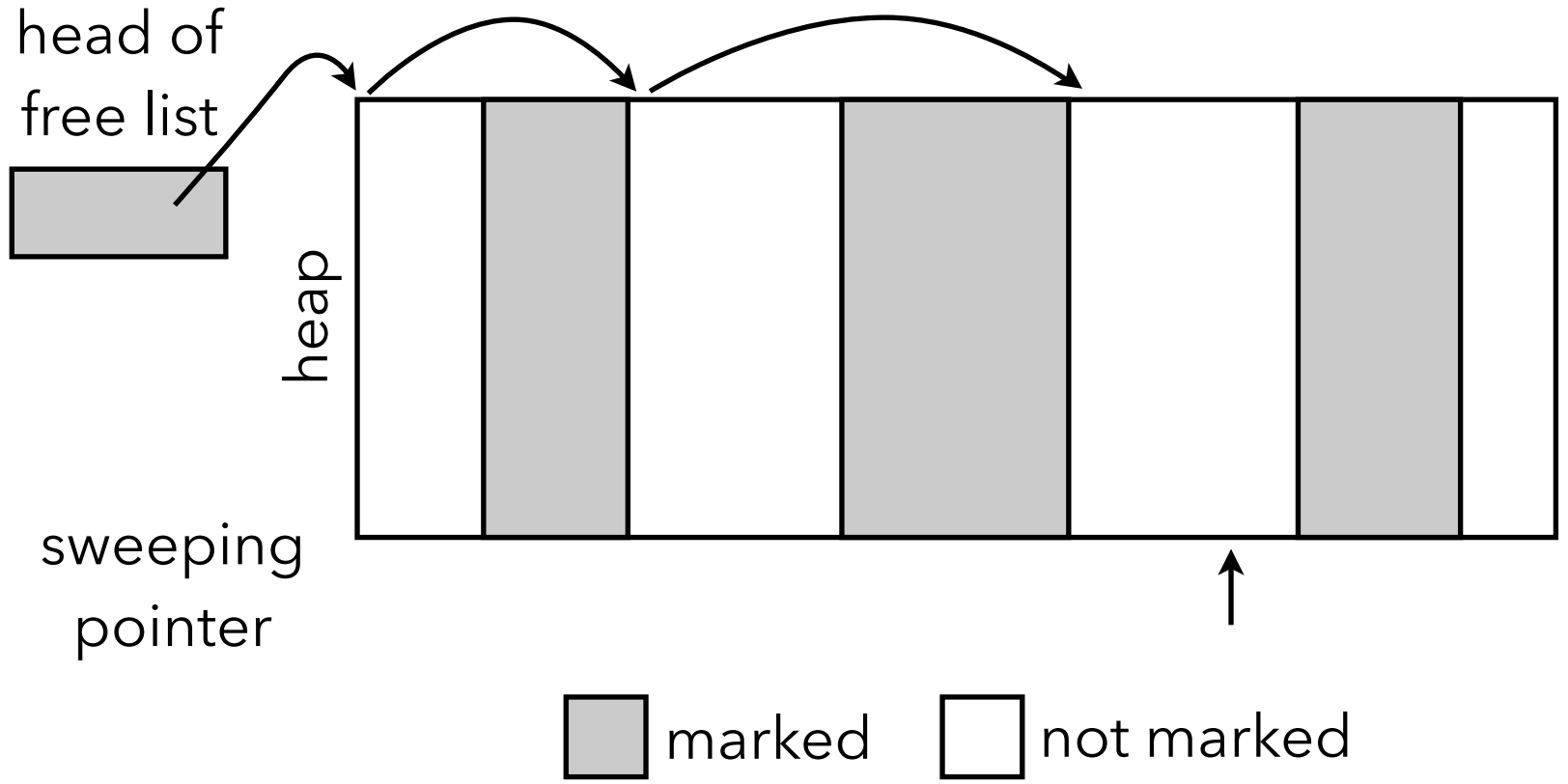
Sweeping and coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



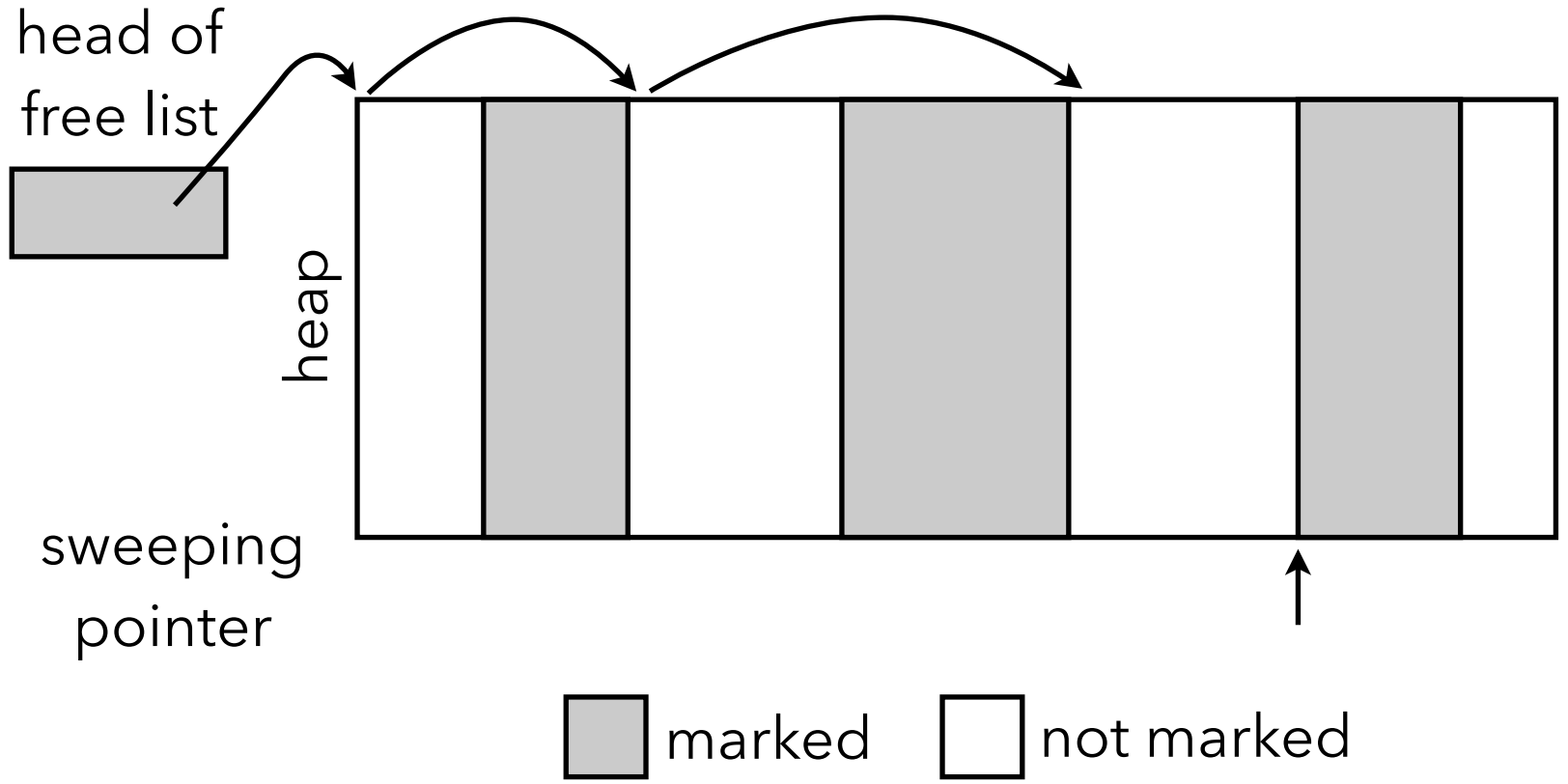
Sweeping and coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



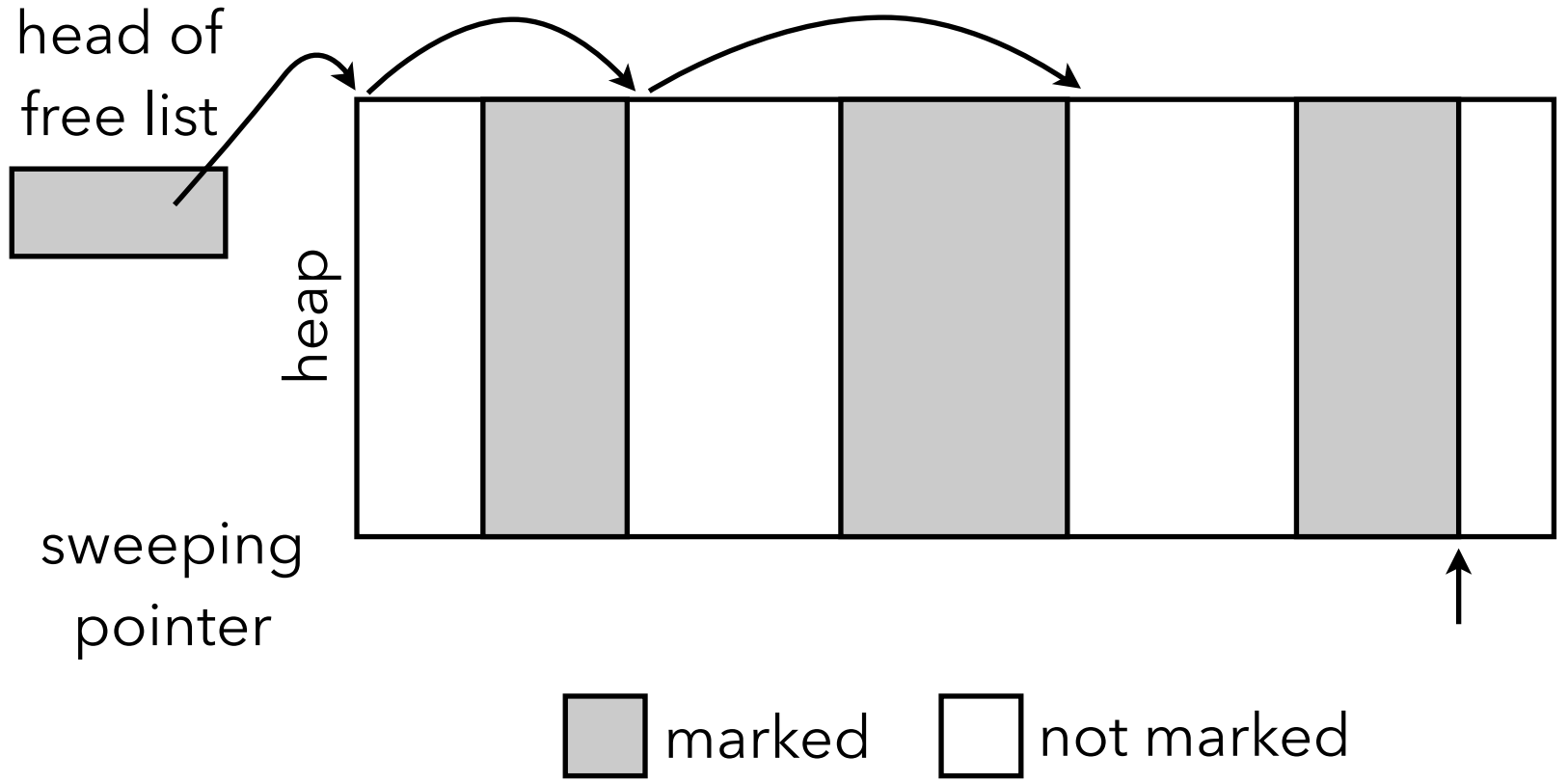
Sweeping and coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



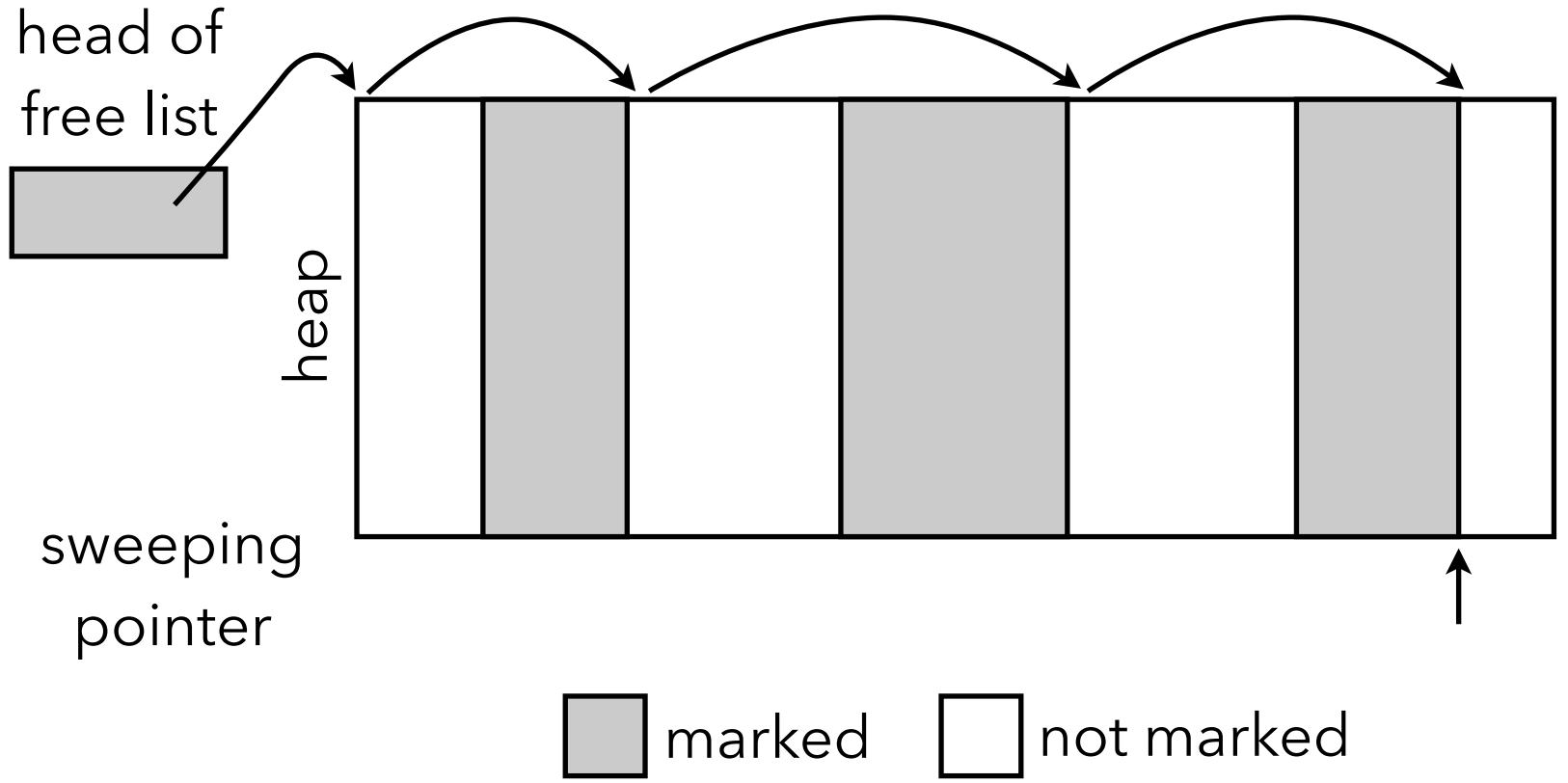
Sweeping and coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



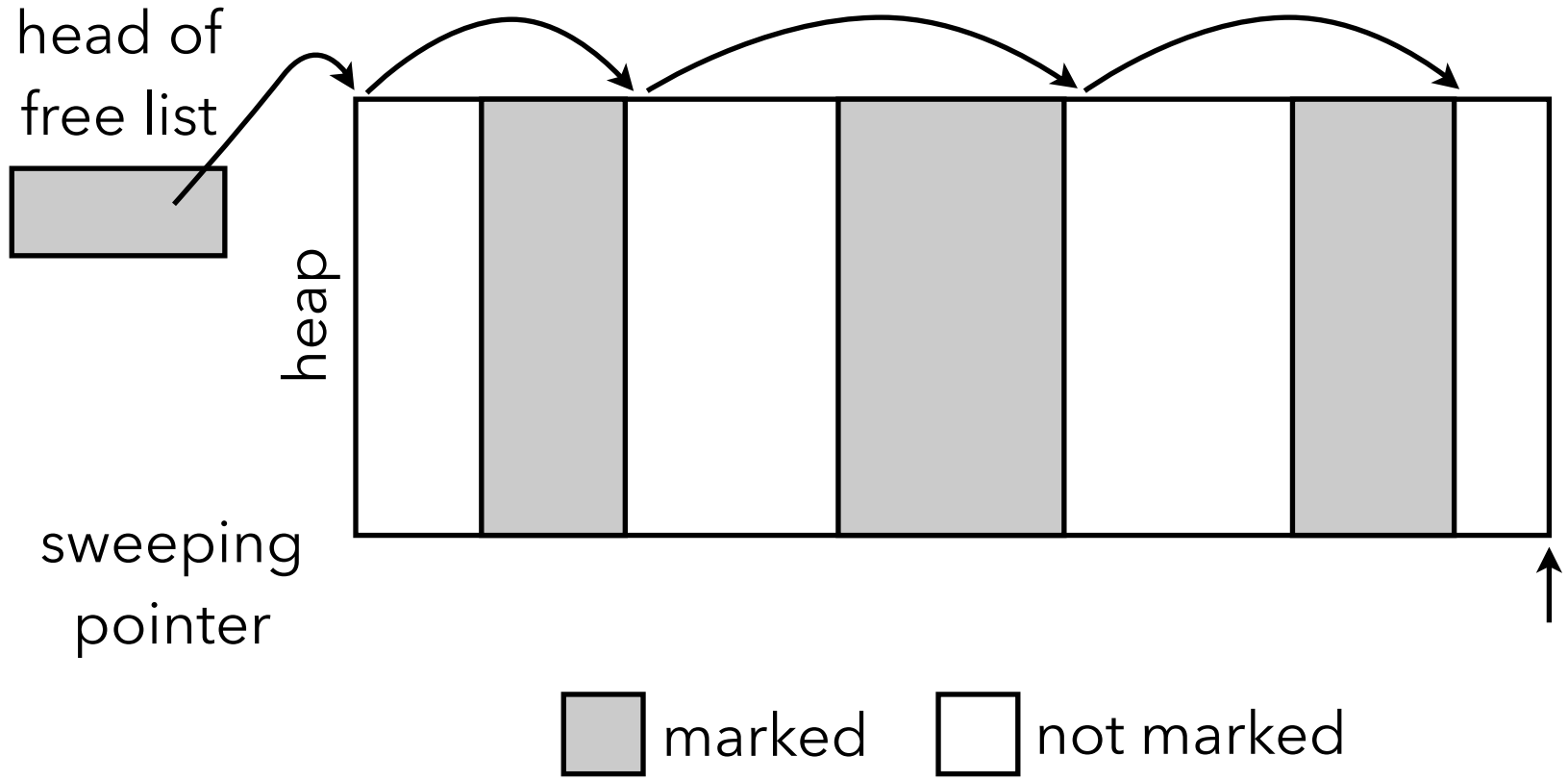
Sweeping and coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



Sweeping and coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



Conservative GC

A **conservative** (**mark & sweep**) **garbage collector** is one that is able to collect memory without having to unambiguously identify pointers at run time.

This is possible because, for a mark & sweep GC, an approximation of the reachability graph is sufficient to collect (some) garbage, as long as that approximation includes the actual reachability graph.

In other words, a conservative GC assumes that everything that looks like a pointer to an allocated object is a pointer to an allocated object. This assumption is conservative — in that it can lead to the retention of dead objects — but safe — in that it cannot lead to the freeing of live objects.

Pointer identification

A conservative garbage collector works like a normal one except that it must try to guess whether a value is a pointer to a heap-allocated object or not. The quality of the guess determines the quality of the GC...

Some characteristics of the architecture or compiler can be used to improve the quality of the guess, for example:

- Many architectures require pointers to be aligned in memory on 2 or 4 bytes boundaries. Therefore, unaligned potential pointers can be ignored.
- Many compilers guarantee that if an object is reachable, then there exists at least one pointer to its beginning. Therefore, potential pointers referring to the inside of allocated heap objects can be ignored.

Exercise

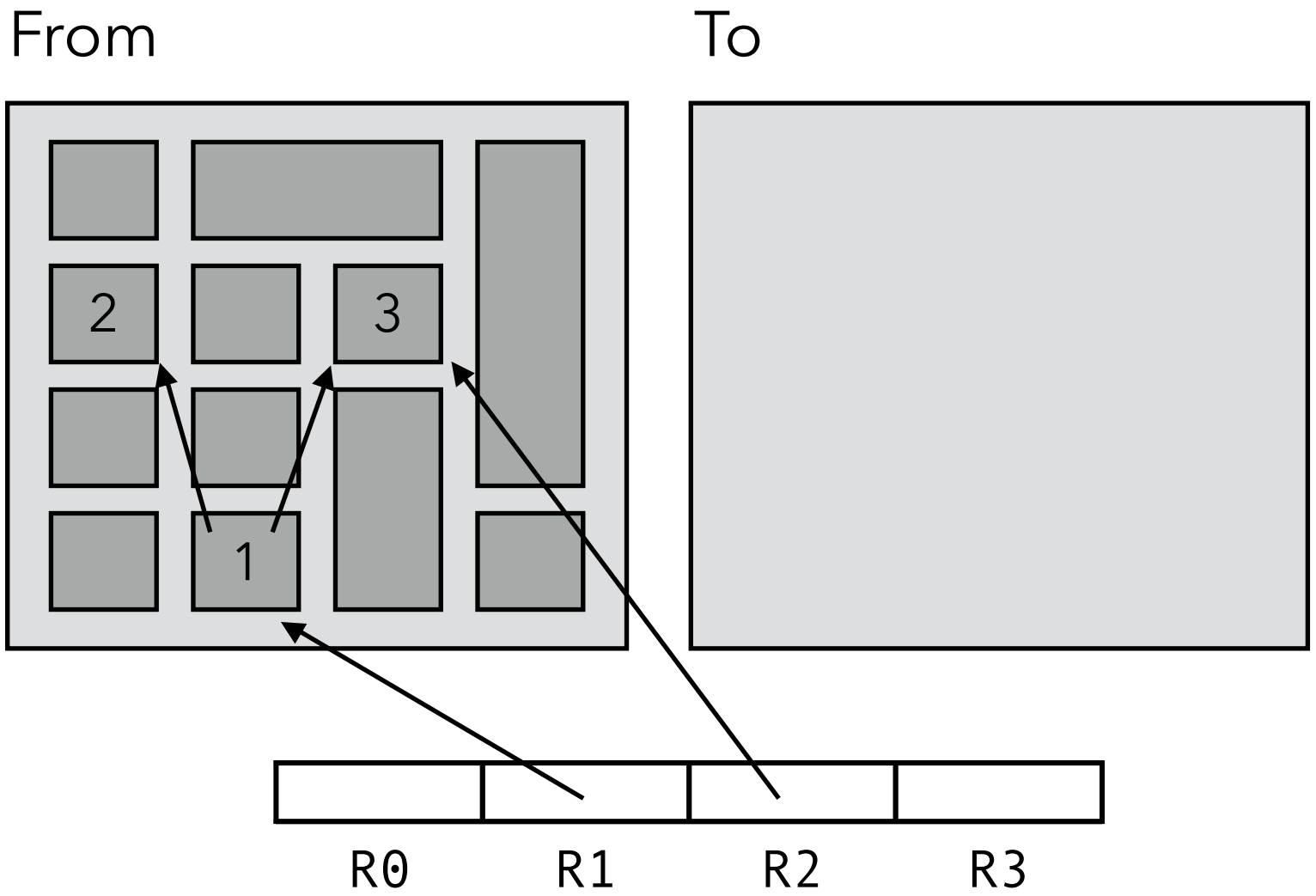
The POSIX malloc function does not clear the memory it returns to the user program, for performance reasons.
In a garbage collected environment, is it also a good idea to return freshly-allocated blocks to the program without clearing them first? Explain.

GC technique #3: copying GC

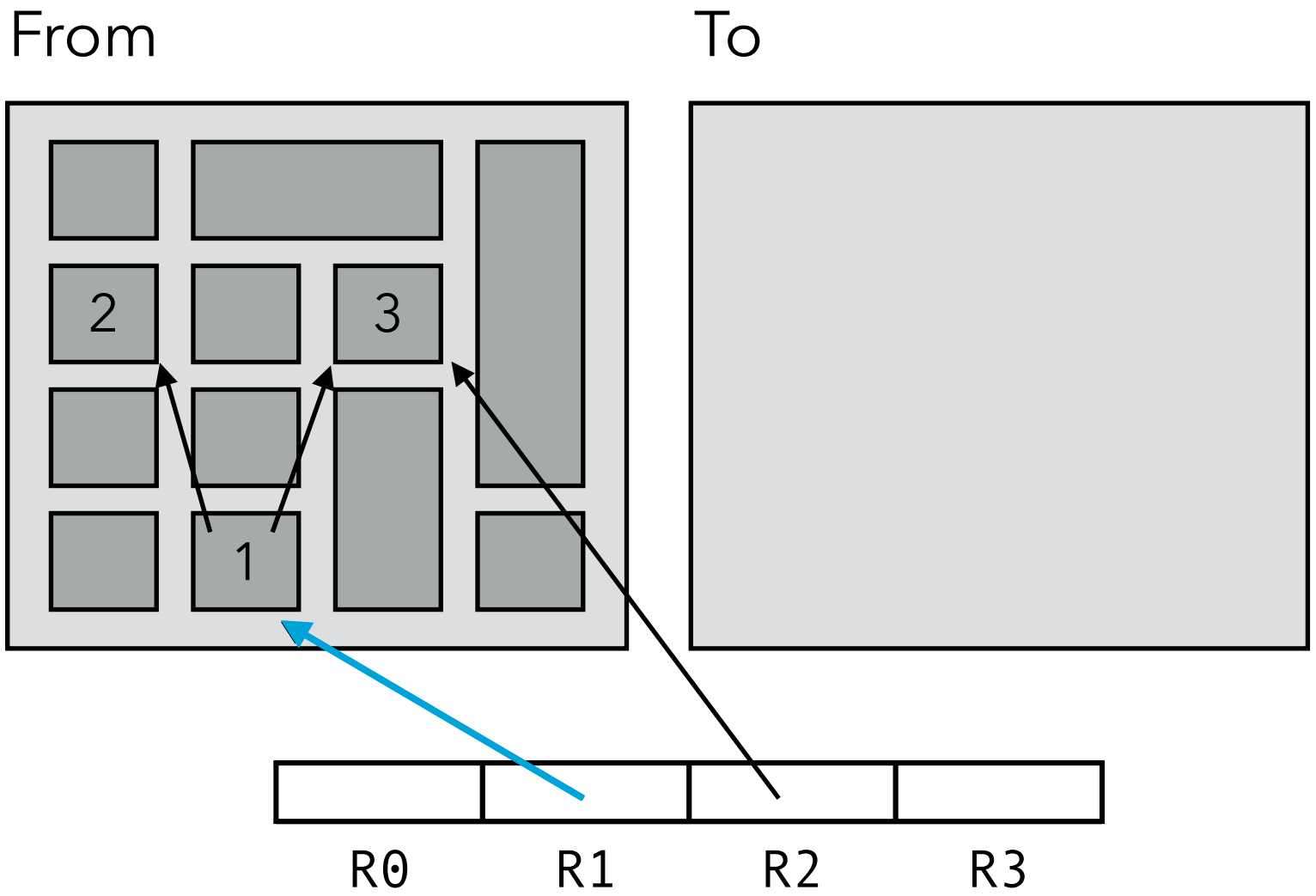
Copying GC

The idea of **copying garbage collection** is to split the heap in two semi-spaces of equal size: the **from-space** and the **to-space**.
Memory is allocated in from-space, while to-space is left empty.
When from-space is full, all reachable objects in from-space are copied to to-space, and pointers to them are updated accordingly.
Finally, the role of the two spaces is exchanged, and the program resumed.

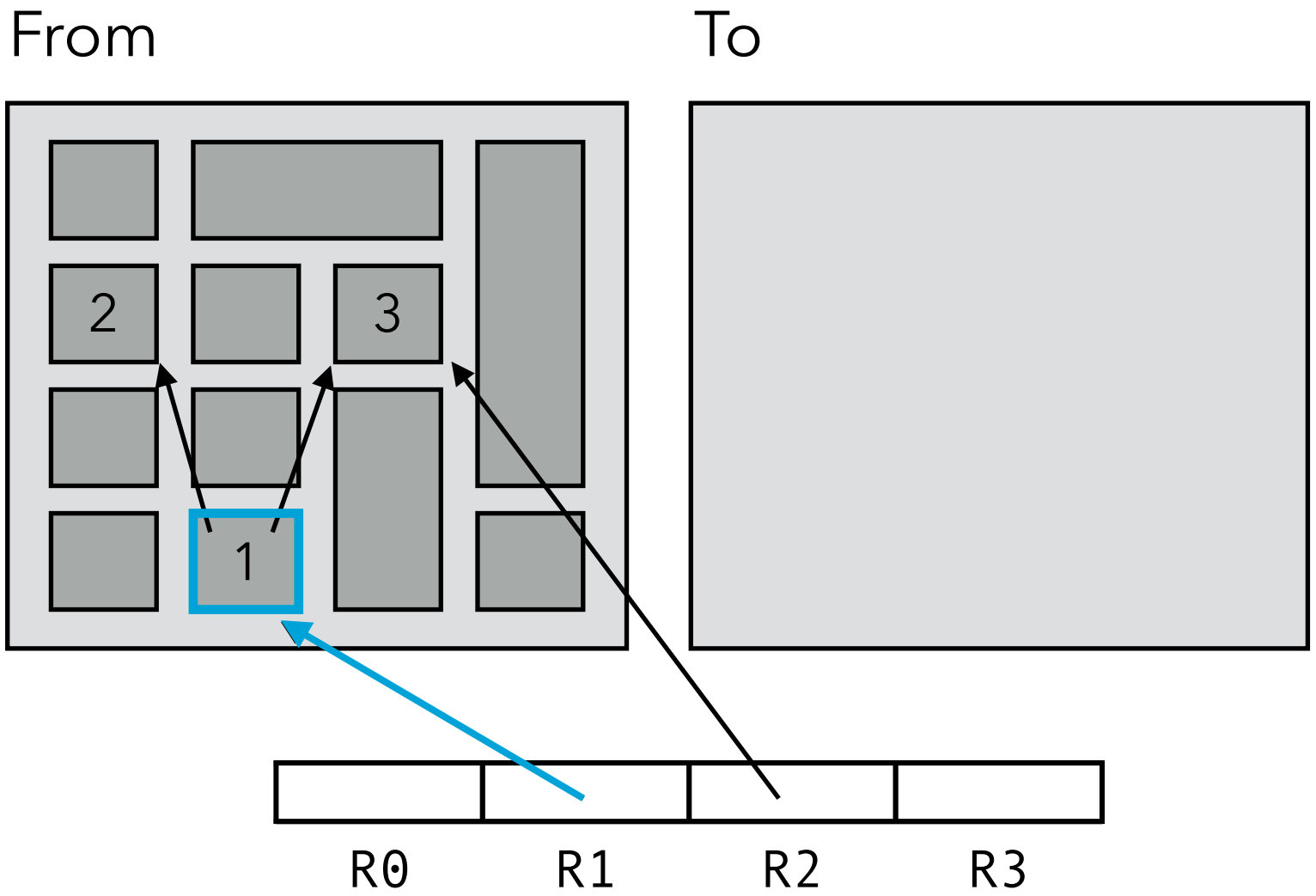
Copying GC



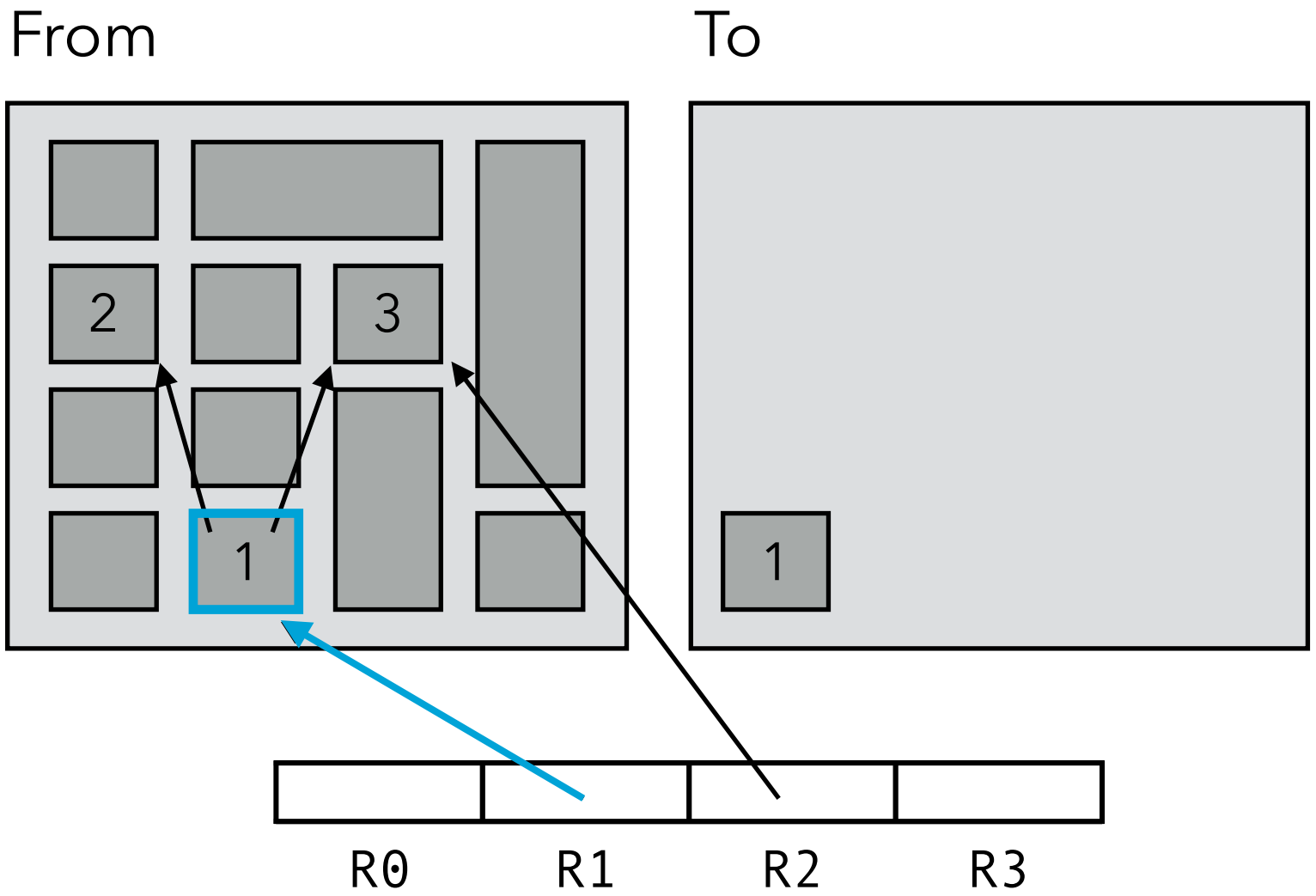
Copying GC



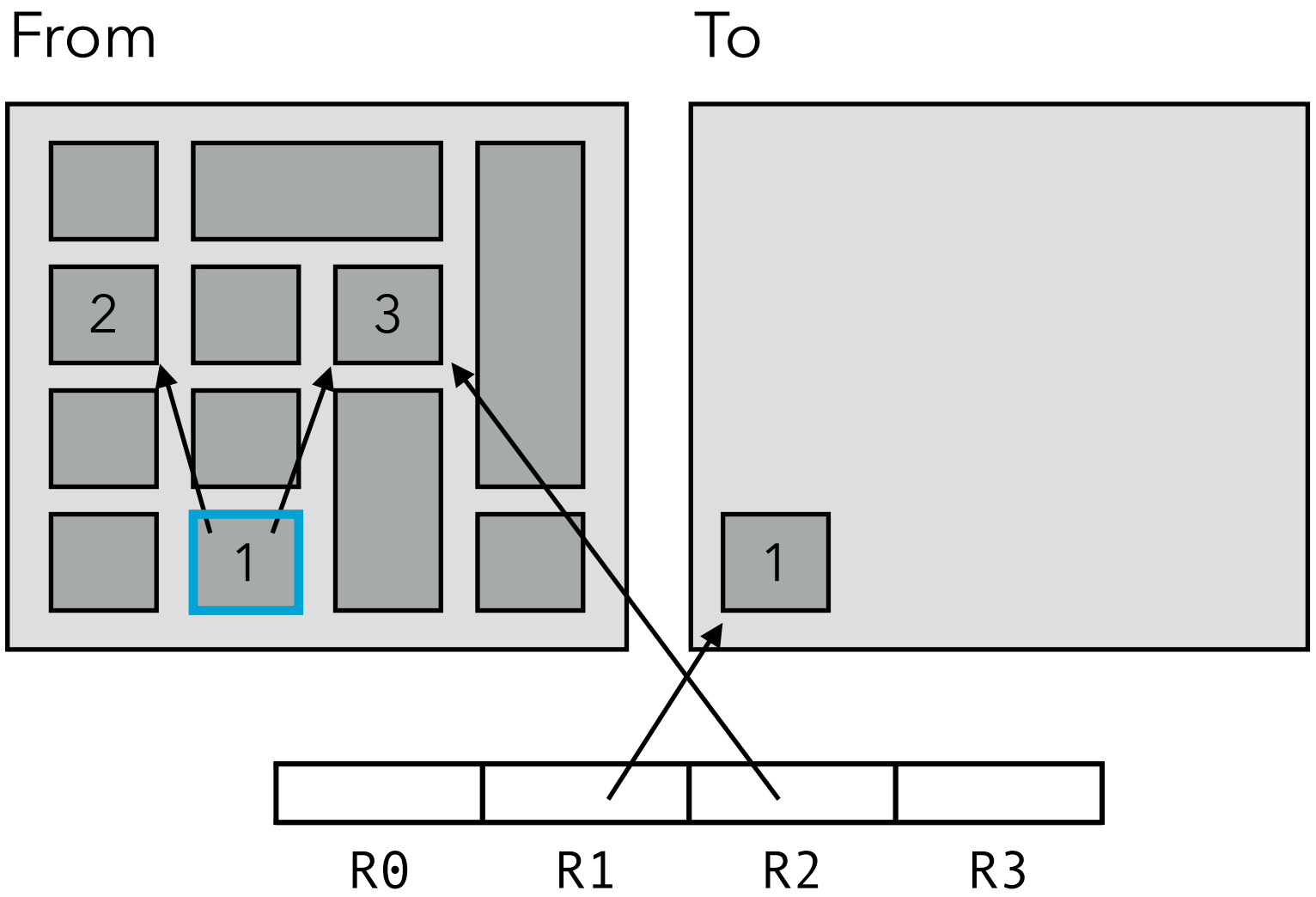
Copying GC



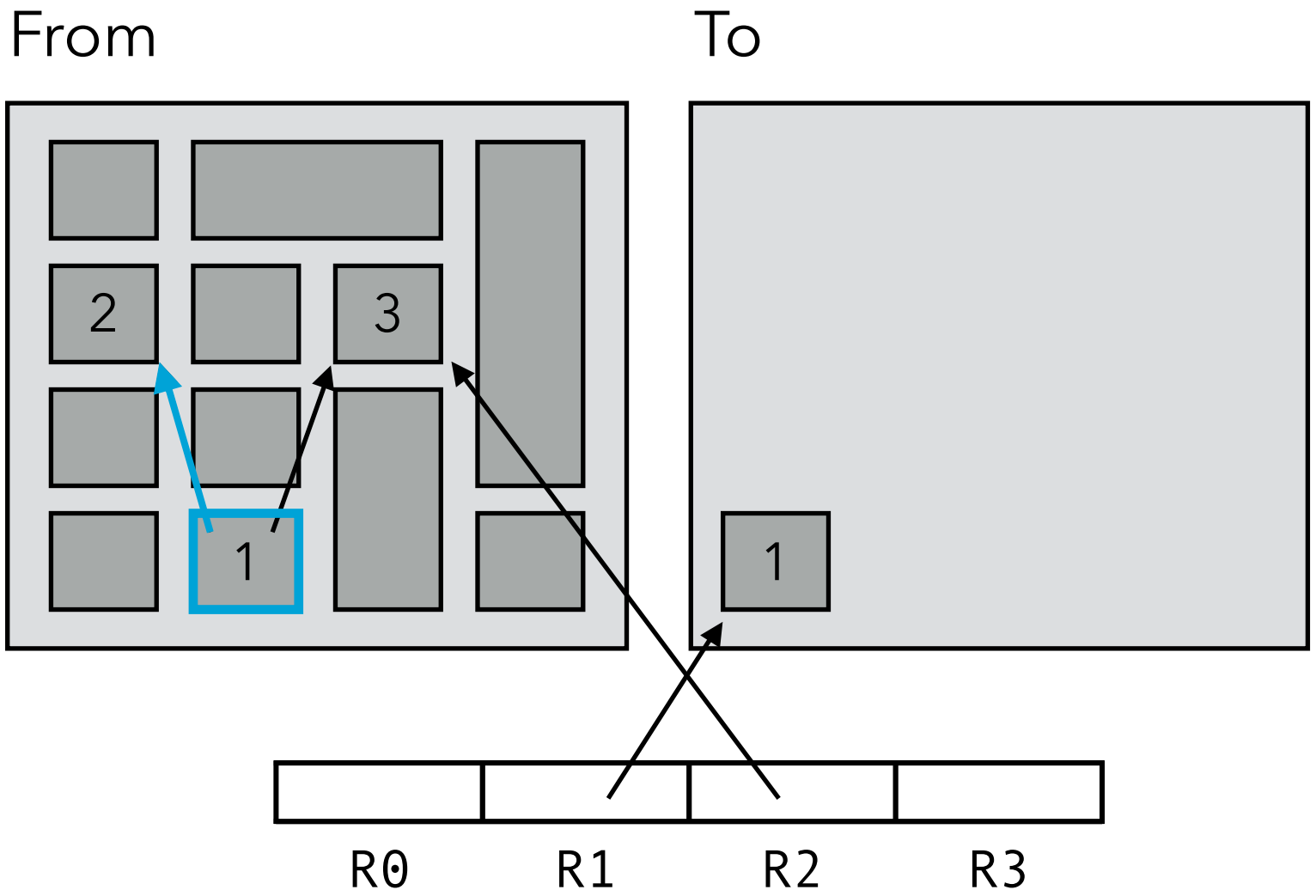
Copying GC



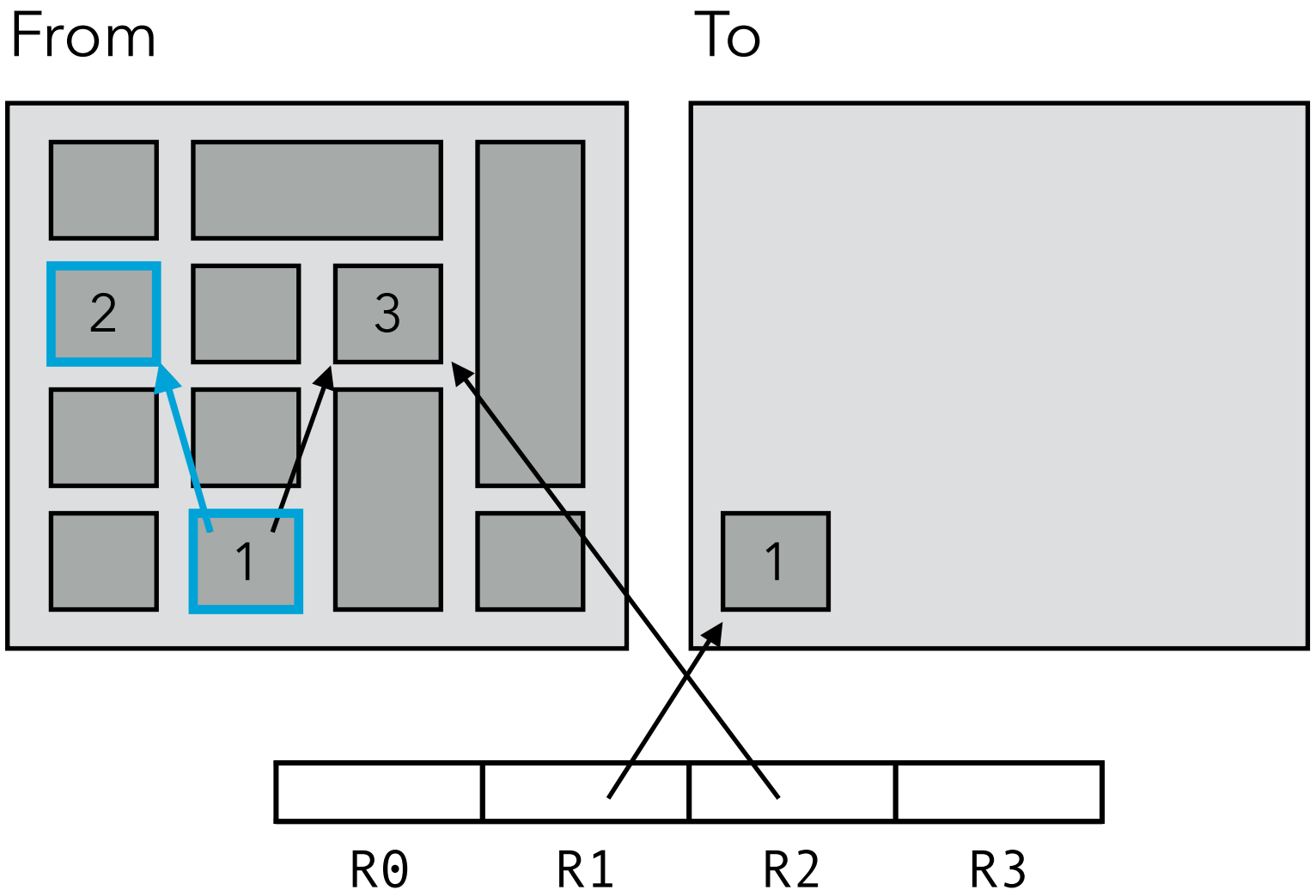
Copying GC



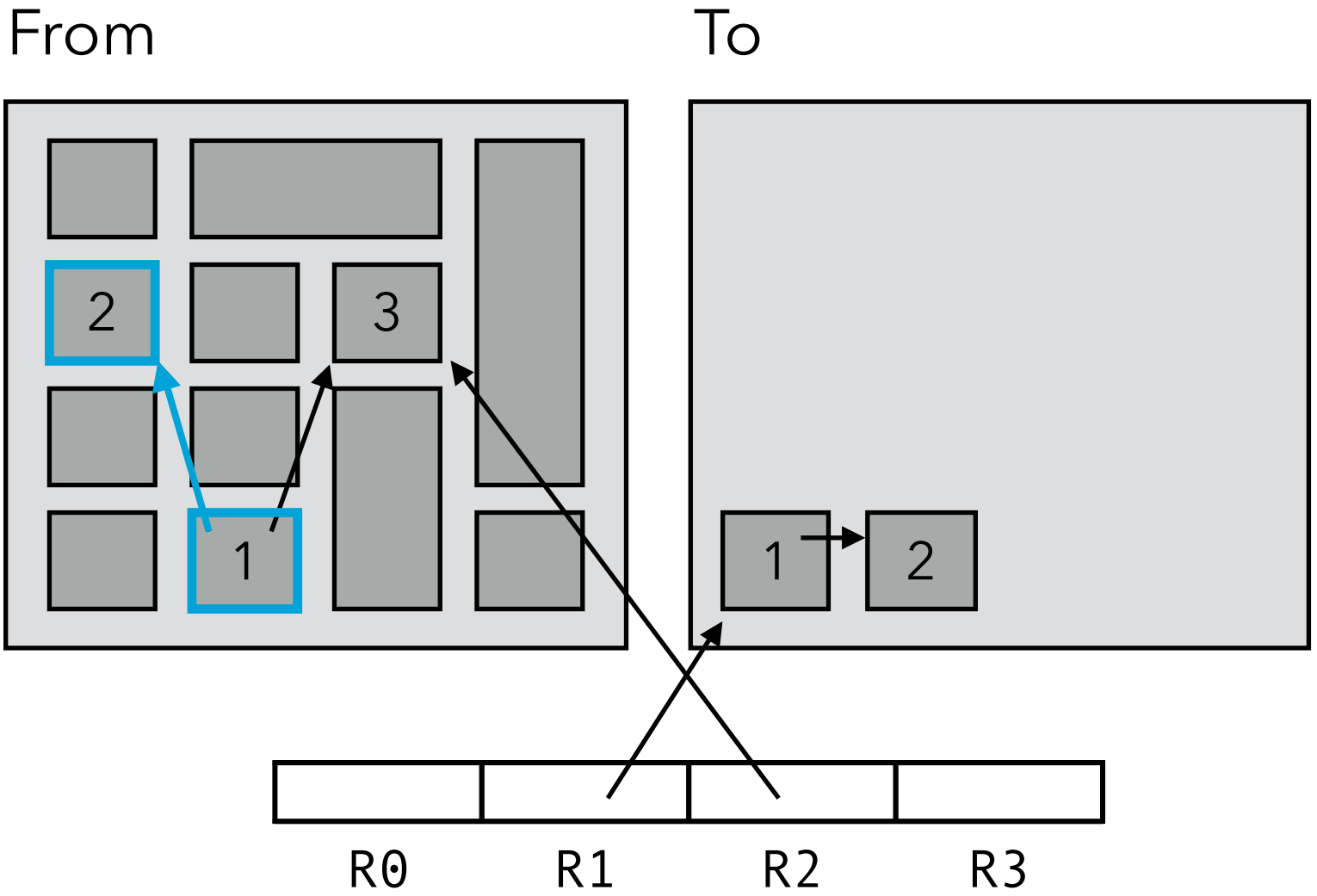
Copying GC



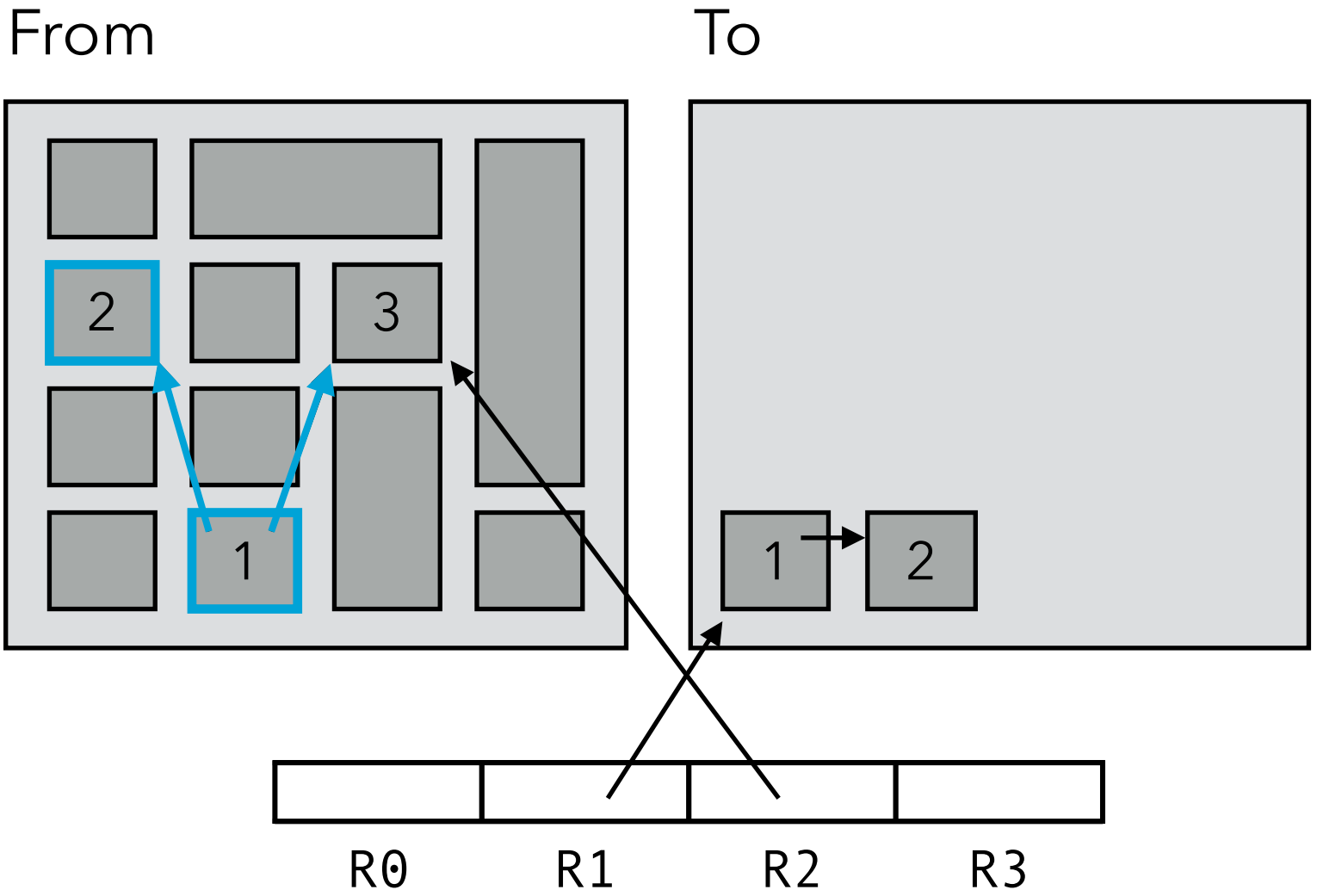
Copying GC



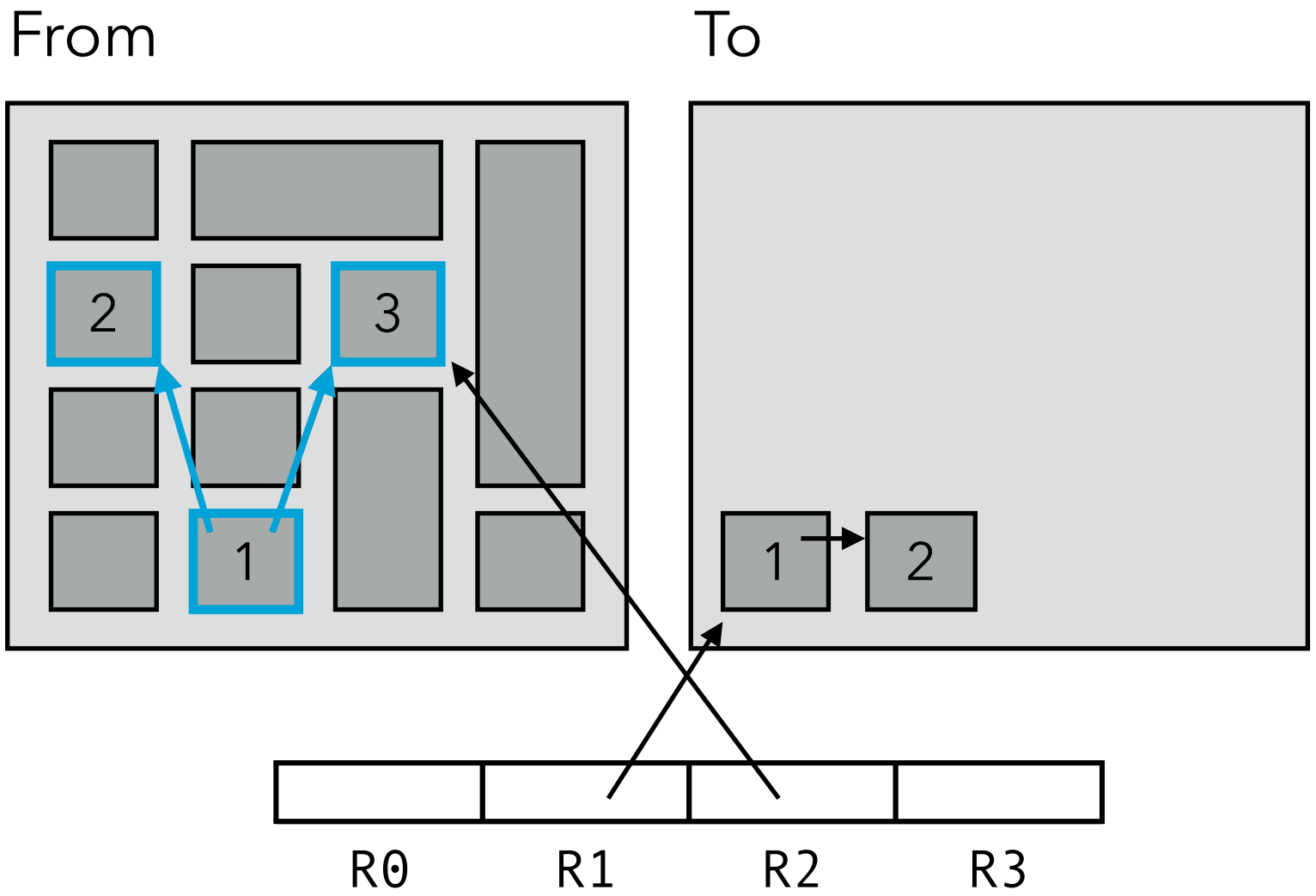
Copying GC



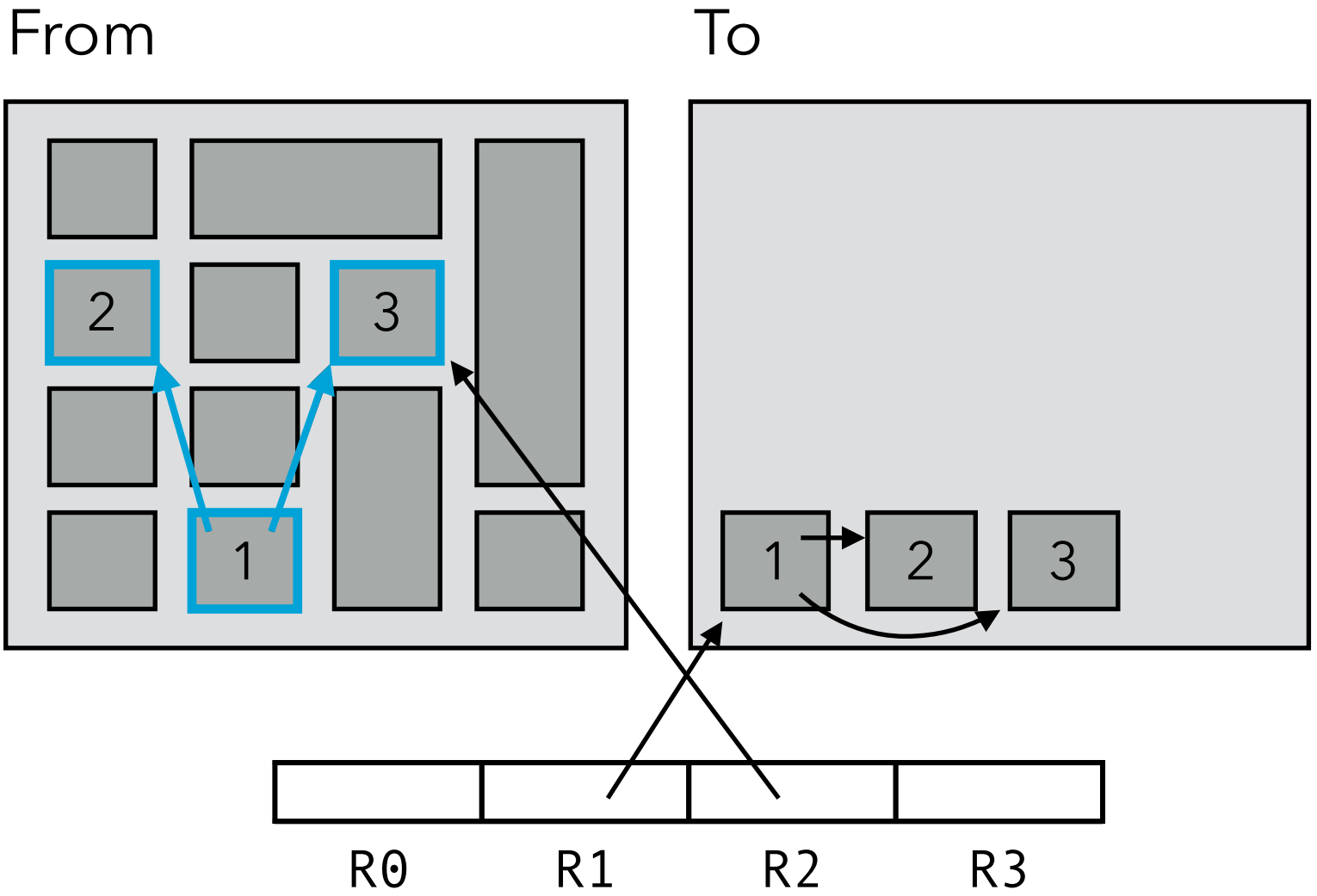
Copying GC



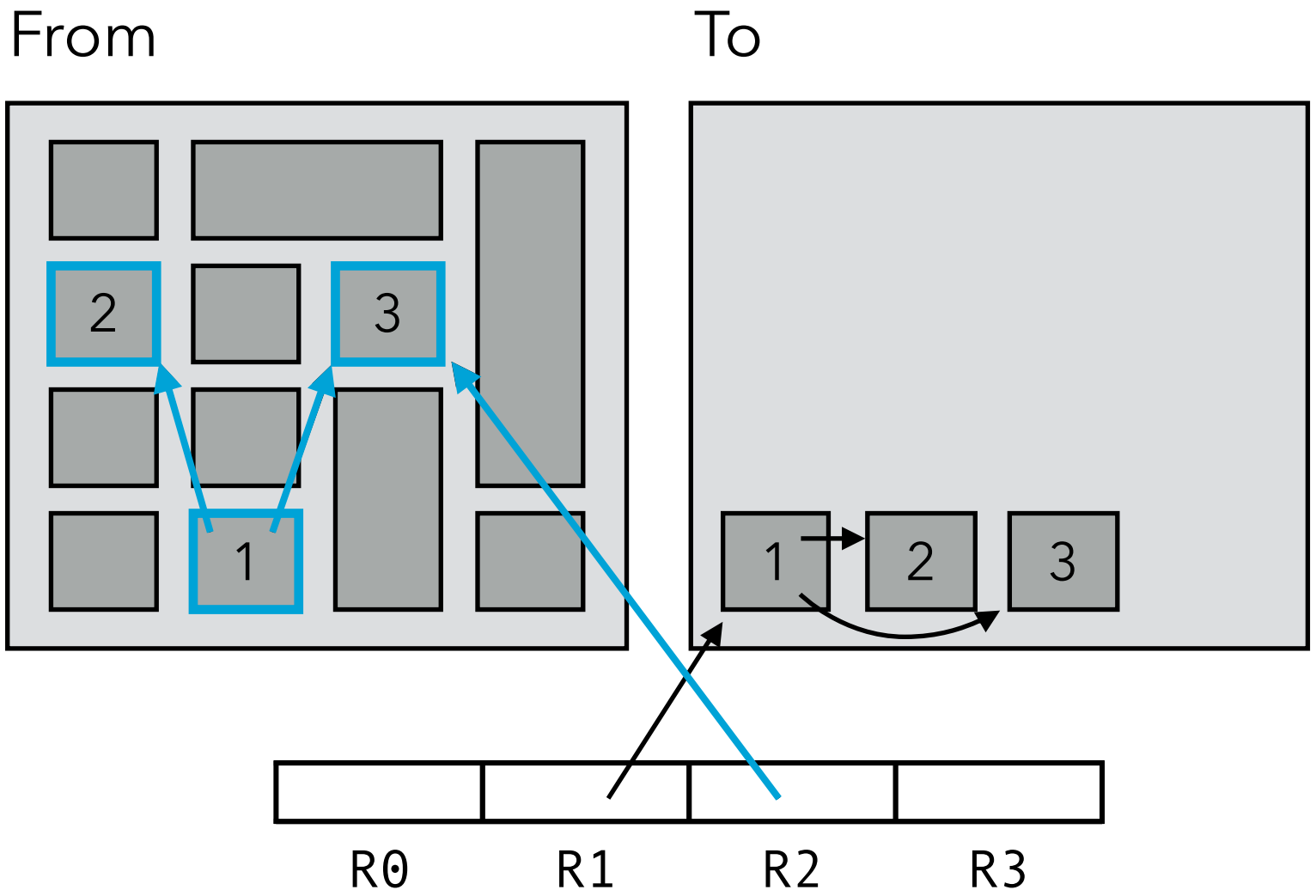
Copying GC



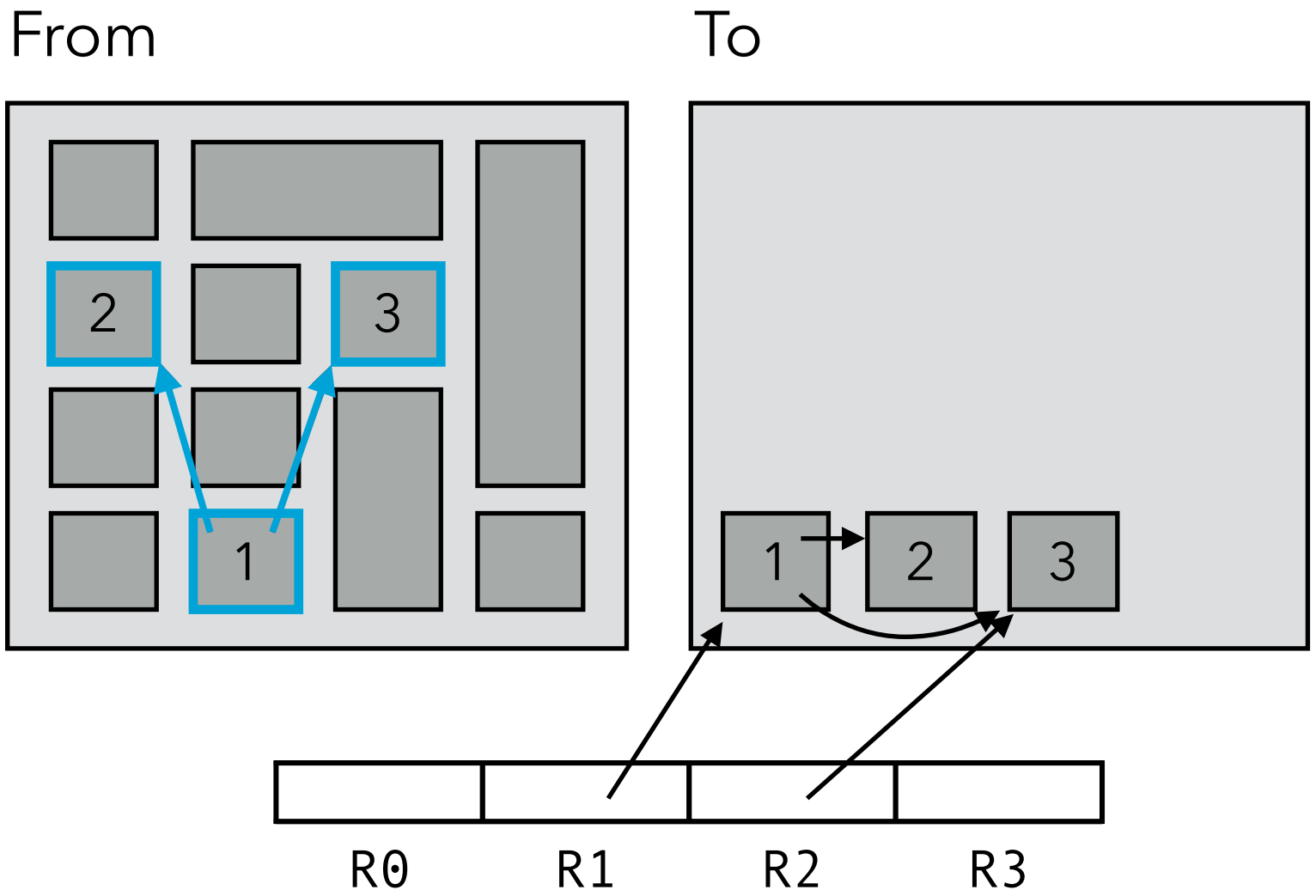
Copying GC



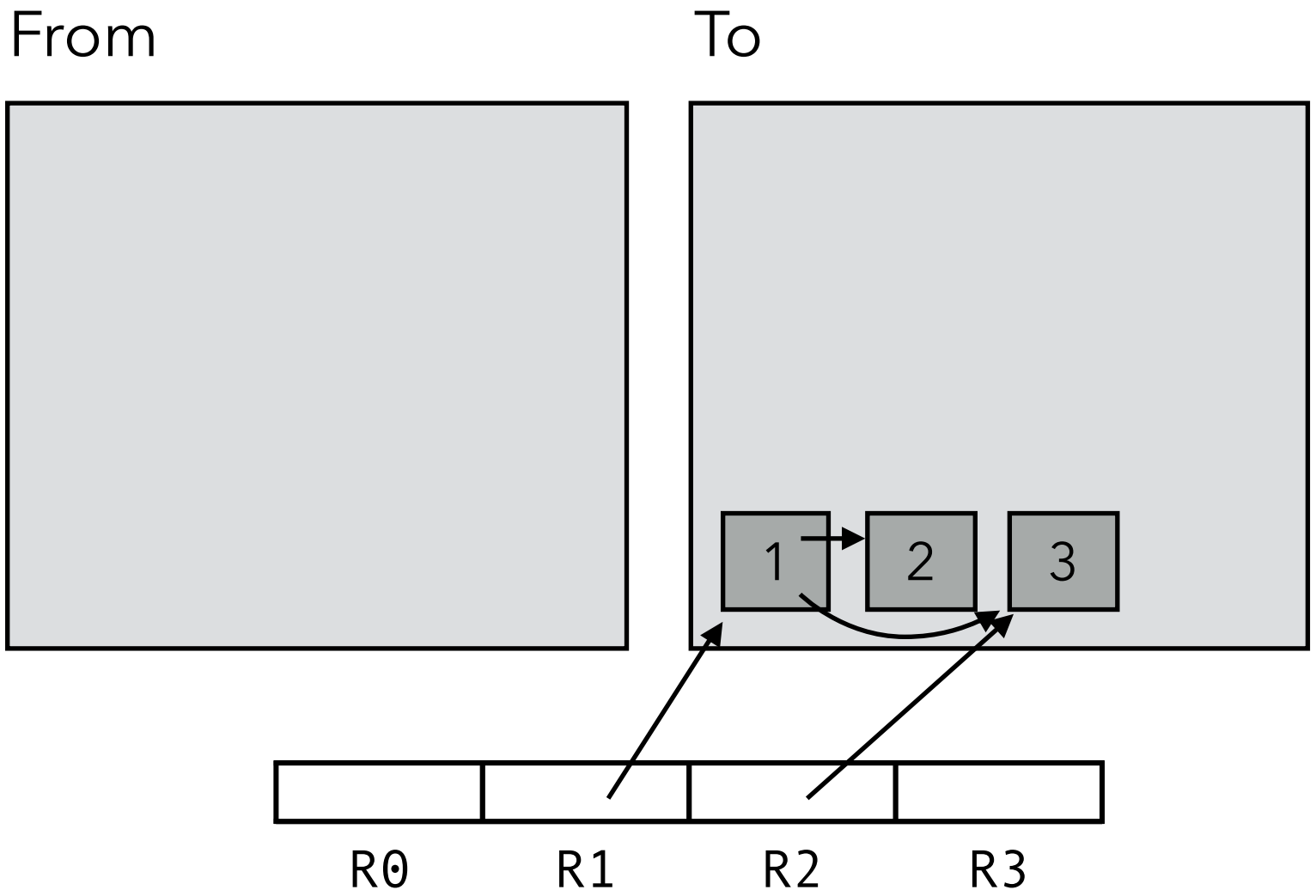
Copying GC



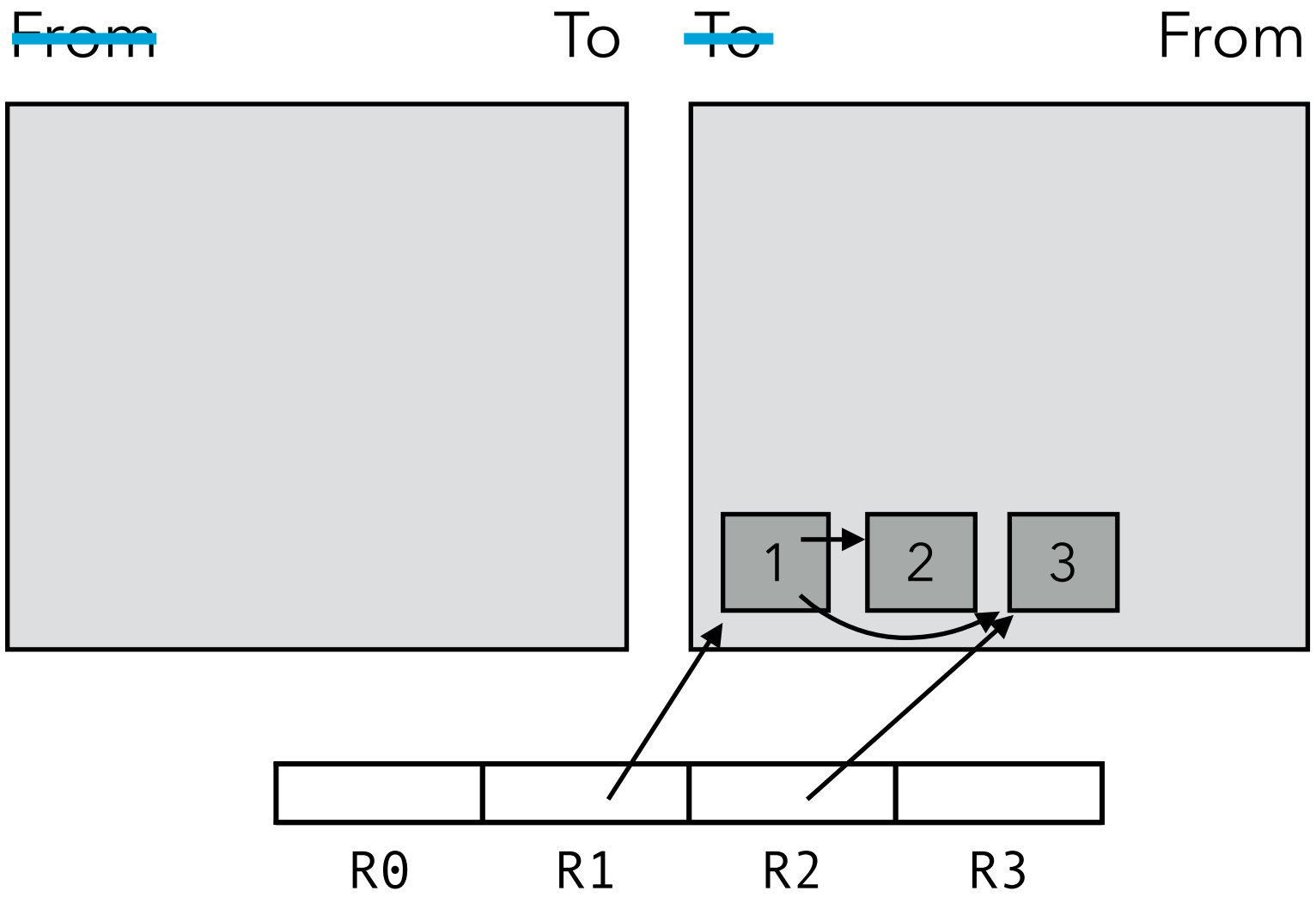
Copying GC



Copying GC



Copying GC



Allocation in a copying GC

In a copying GC, memory is allocated linearly in from-space. There is no free list to maintain, and no search to perform in order to find a free block. All that is required is a pointer to the border between the allocated and free area of from-space.

Allocation in a copying GC is therefore very fast – as fast as stack allocation.

Forwarding pointers

Before copying an object, a check must be made to see whether it has already been copied. If this is the case, it must not be copied again. Rather, the already-copied version must be used.

How can this check be performed? By storing a **forwarding pointer** in the object in from-space, after it has been copied.

Cheney's copying GC

The copying GC algorithm presented before does a depth-first traversal of the reachable graph. When it is implemented using recursion, it can lead to stack overflow. **Cheney's copying GC** is an elegant GC technique that does a breadth-first traversal of the reachable graph, requiring only one pointer as additional state.

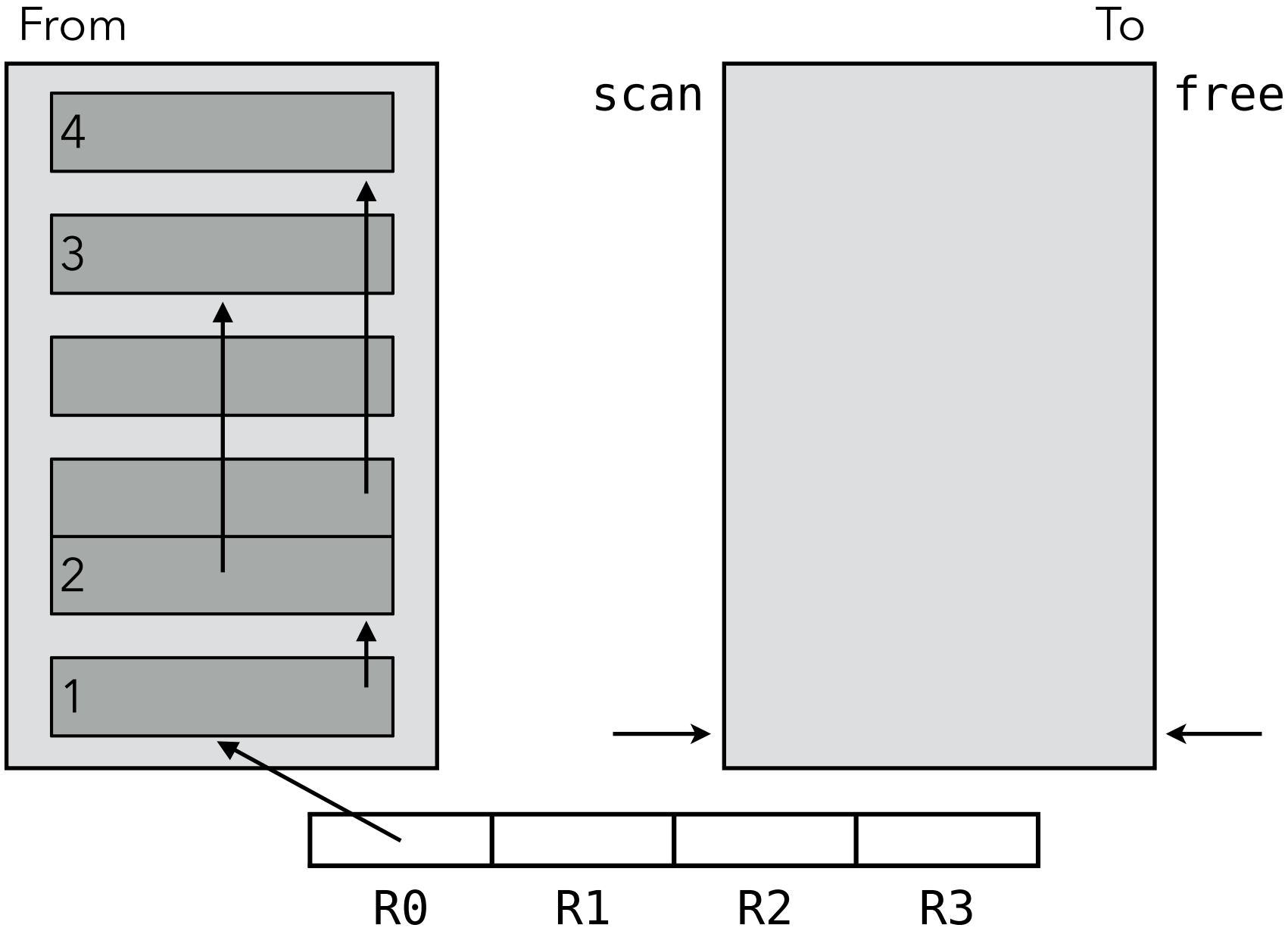
Cheney's copying GC

In any breadth-first traversal, one has to remember the set of nodes that have been visited, but whose children have not been.

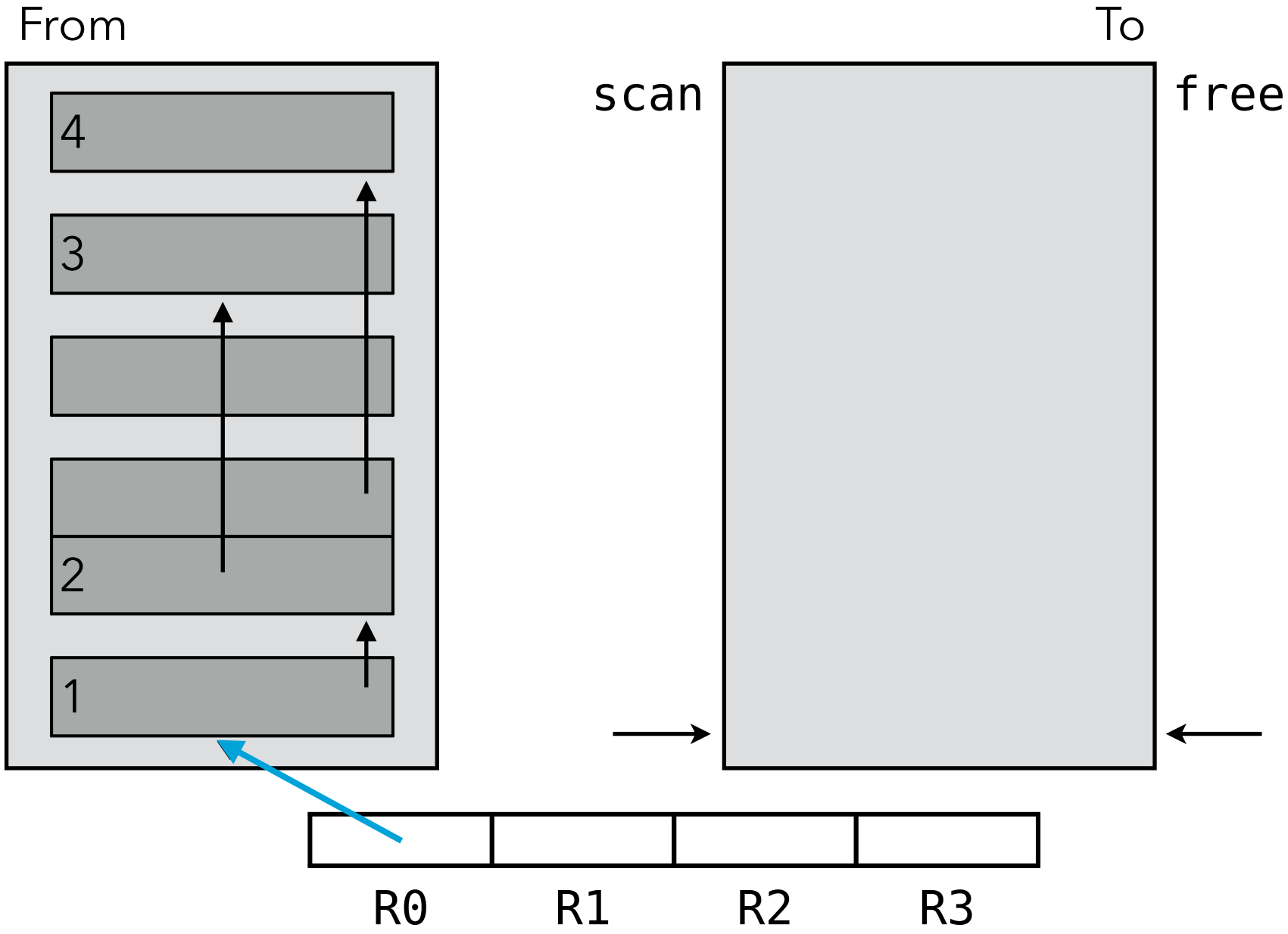
The basic idea of Cheney's algorithm is to use to-space to store this set of nodes, which can be represented using a single pointer called `scan`.

This pointer partitions to-space in two parts: the nodes whose children have been visited, and those whose children have not been visited.

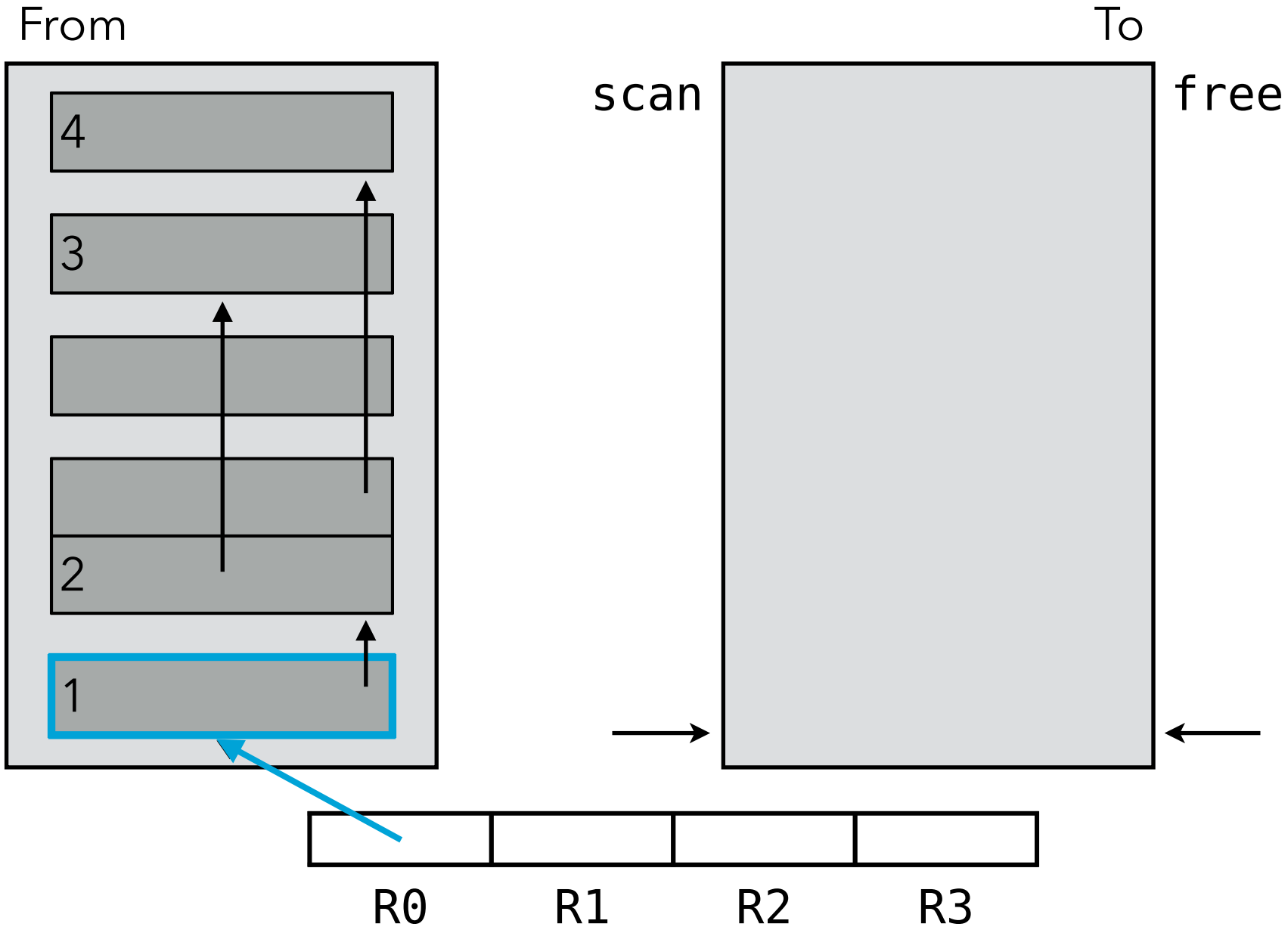
Cheney's copying GC



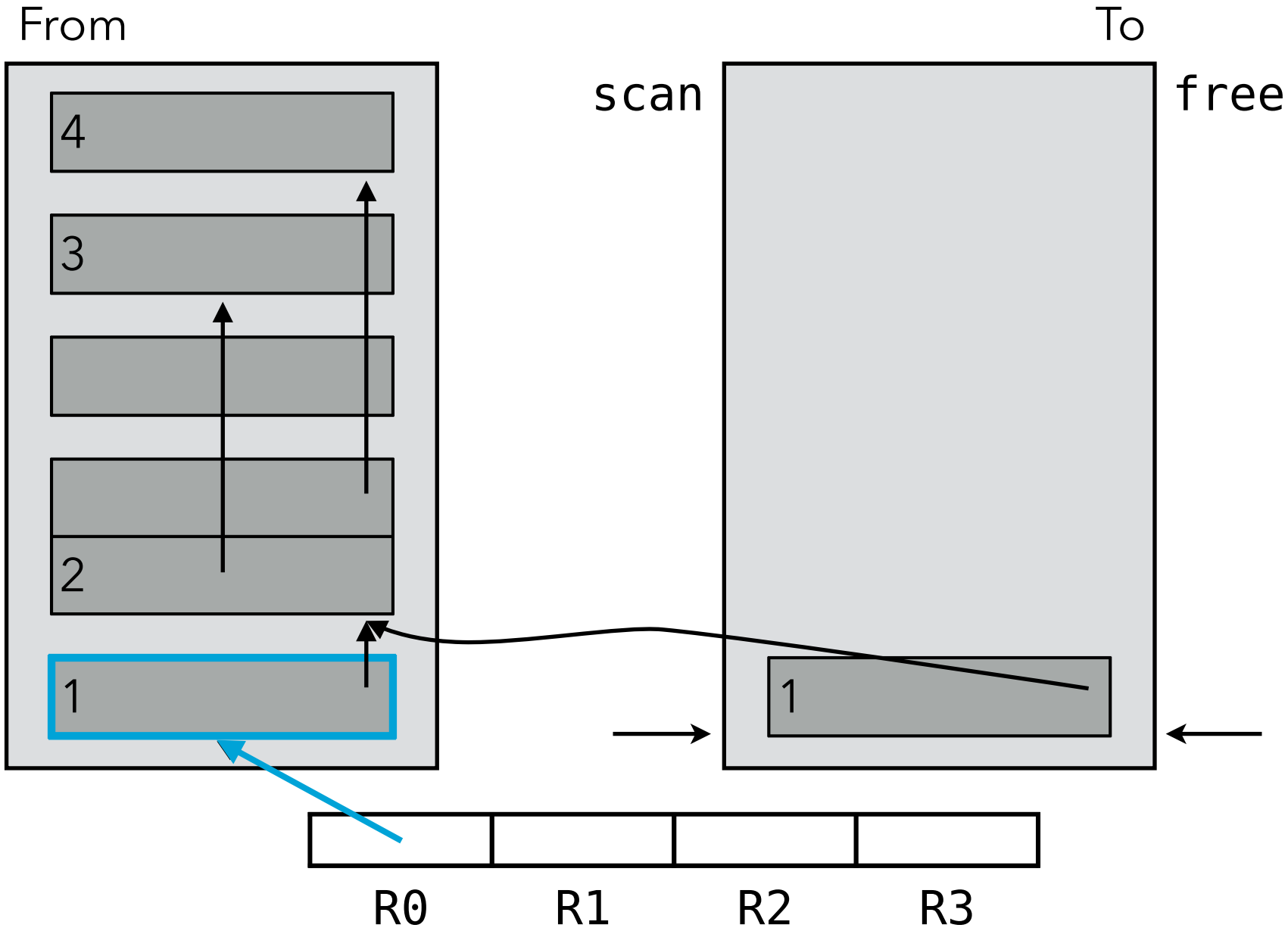
Cheney's copying GC



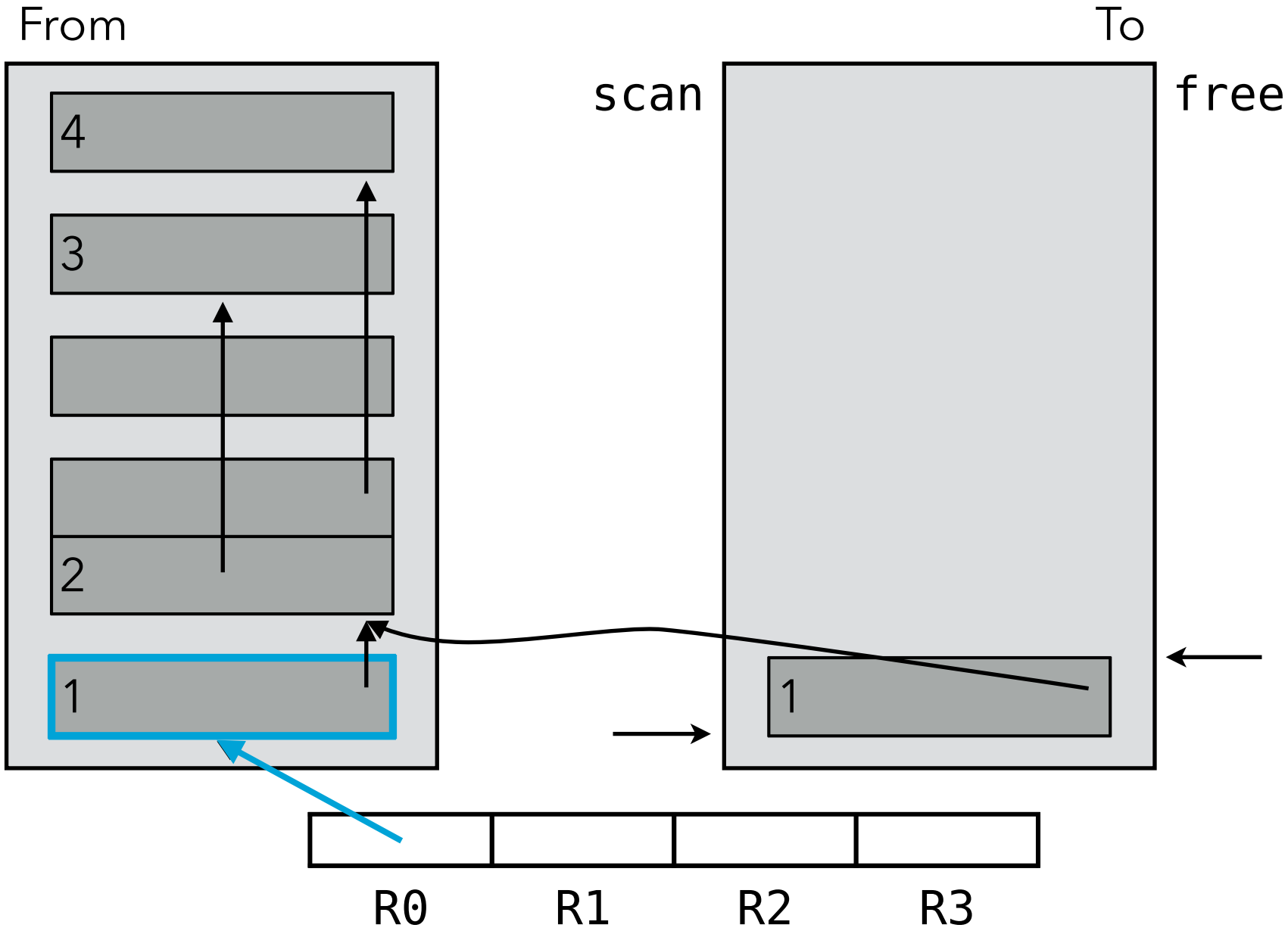
Cheney's copying GC



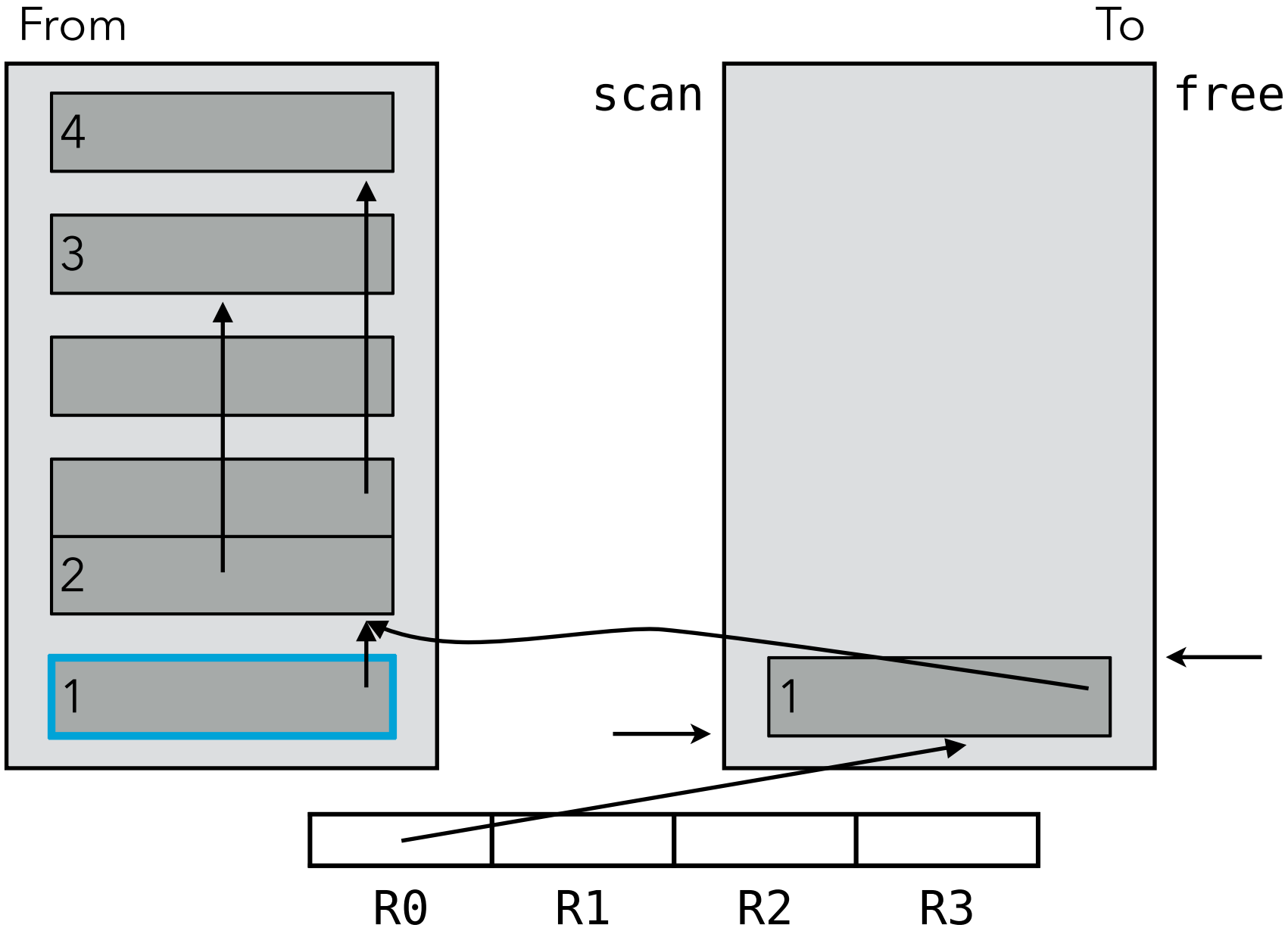
Cheney's copying GC



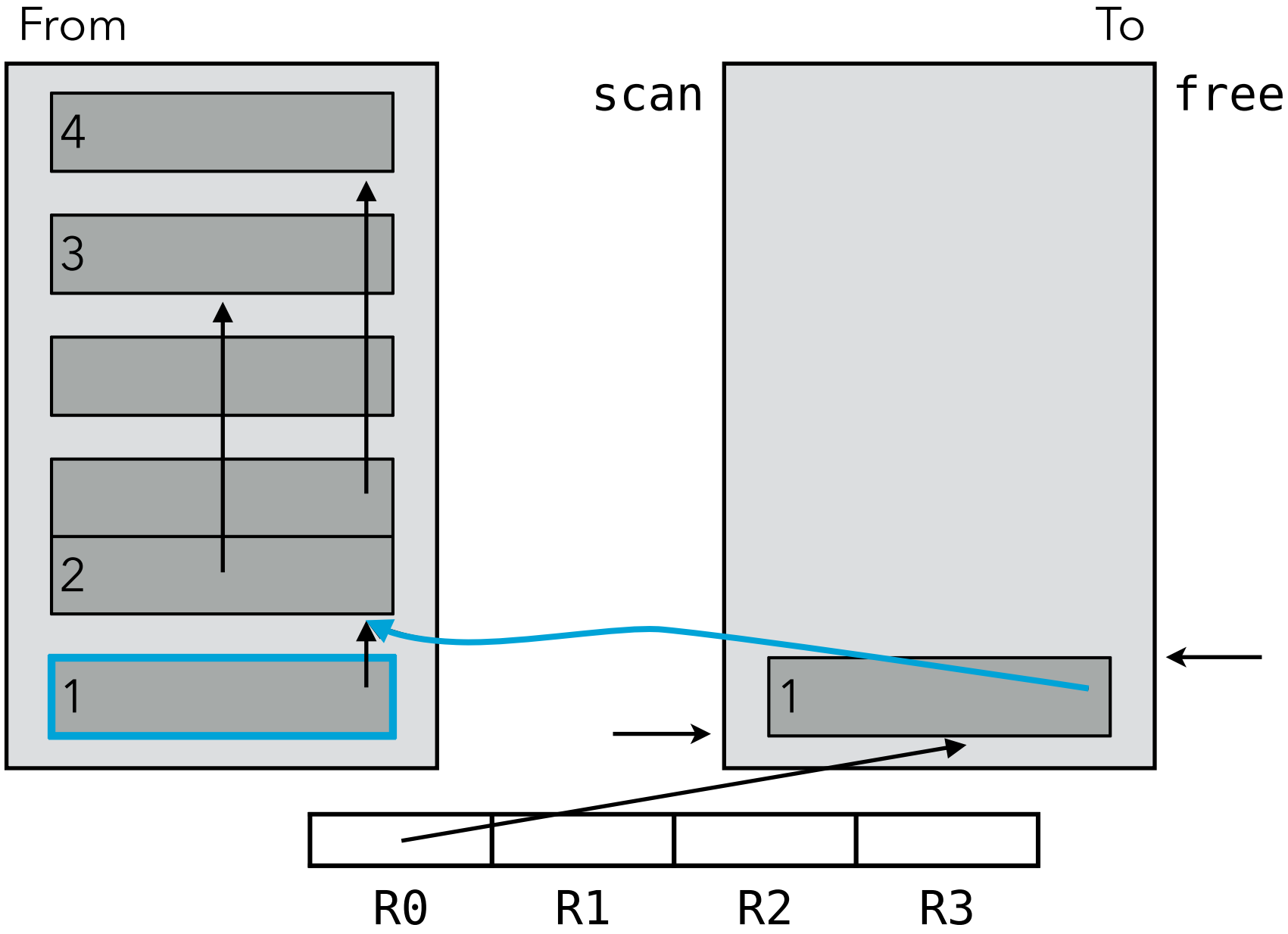
Cheney's copying GC



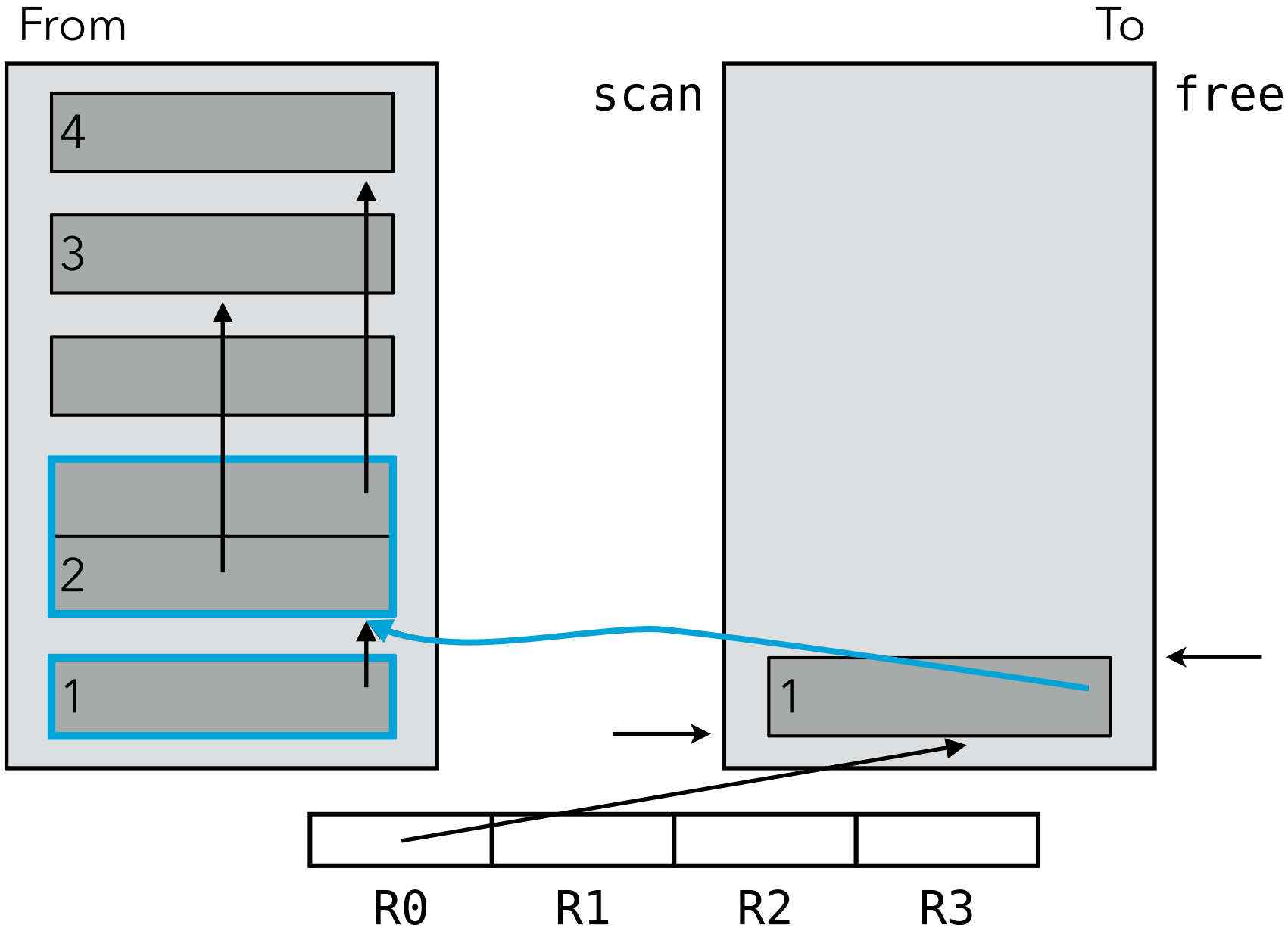
Cheney's copying GC



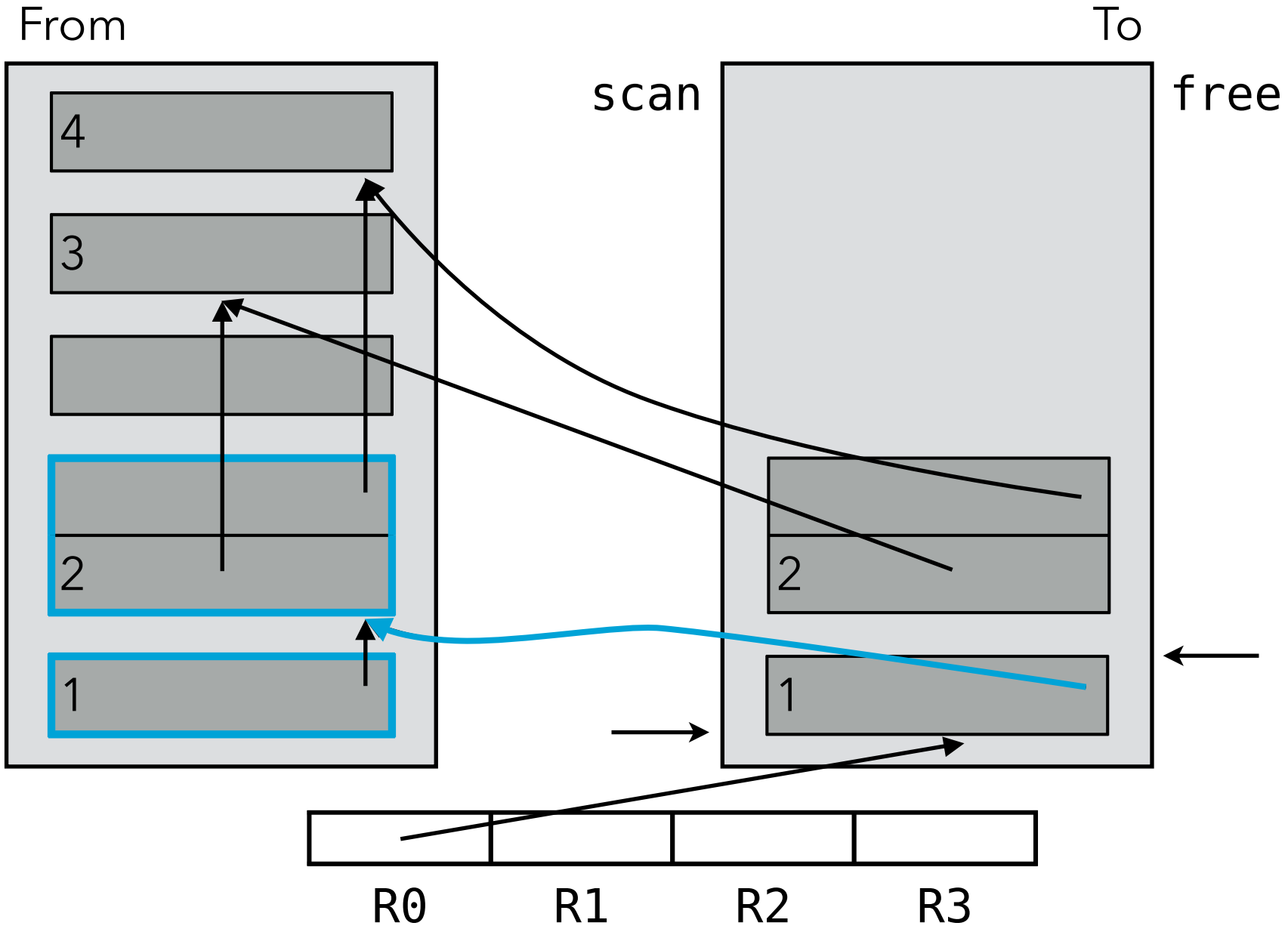
Cheney's copying GC



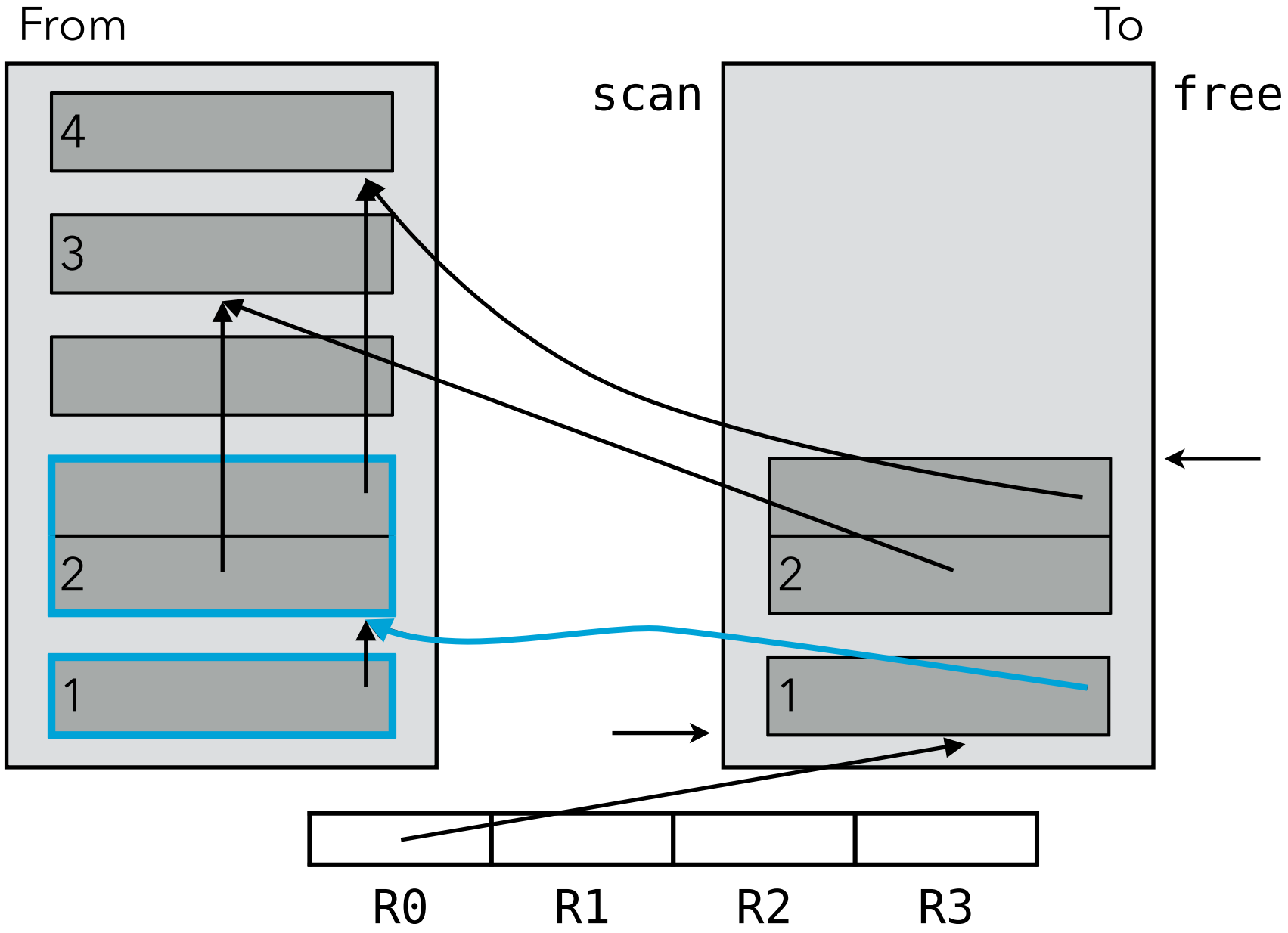
Cheney's copying GC



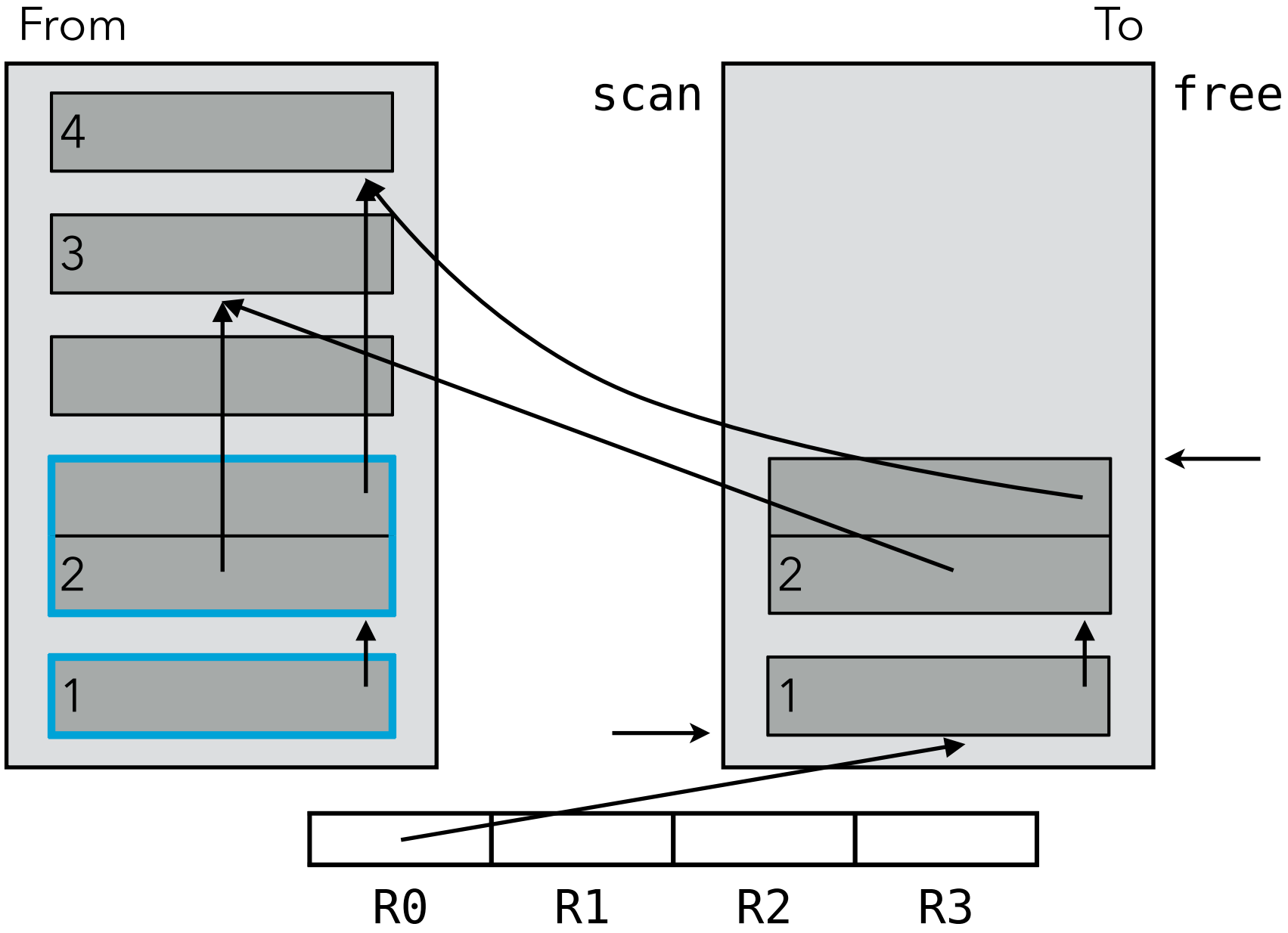
Cheney's copying GC



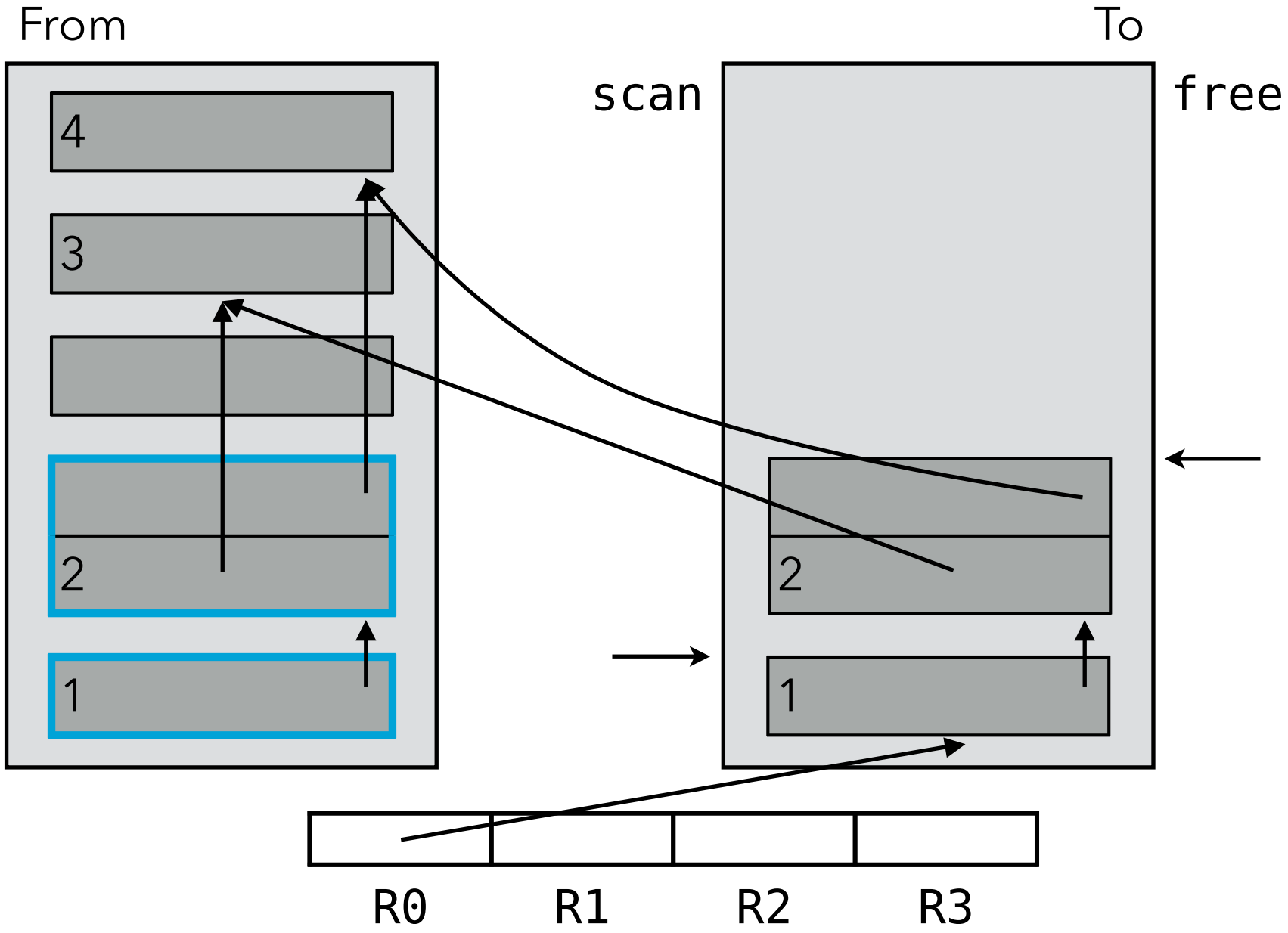
Cheney's copying GC



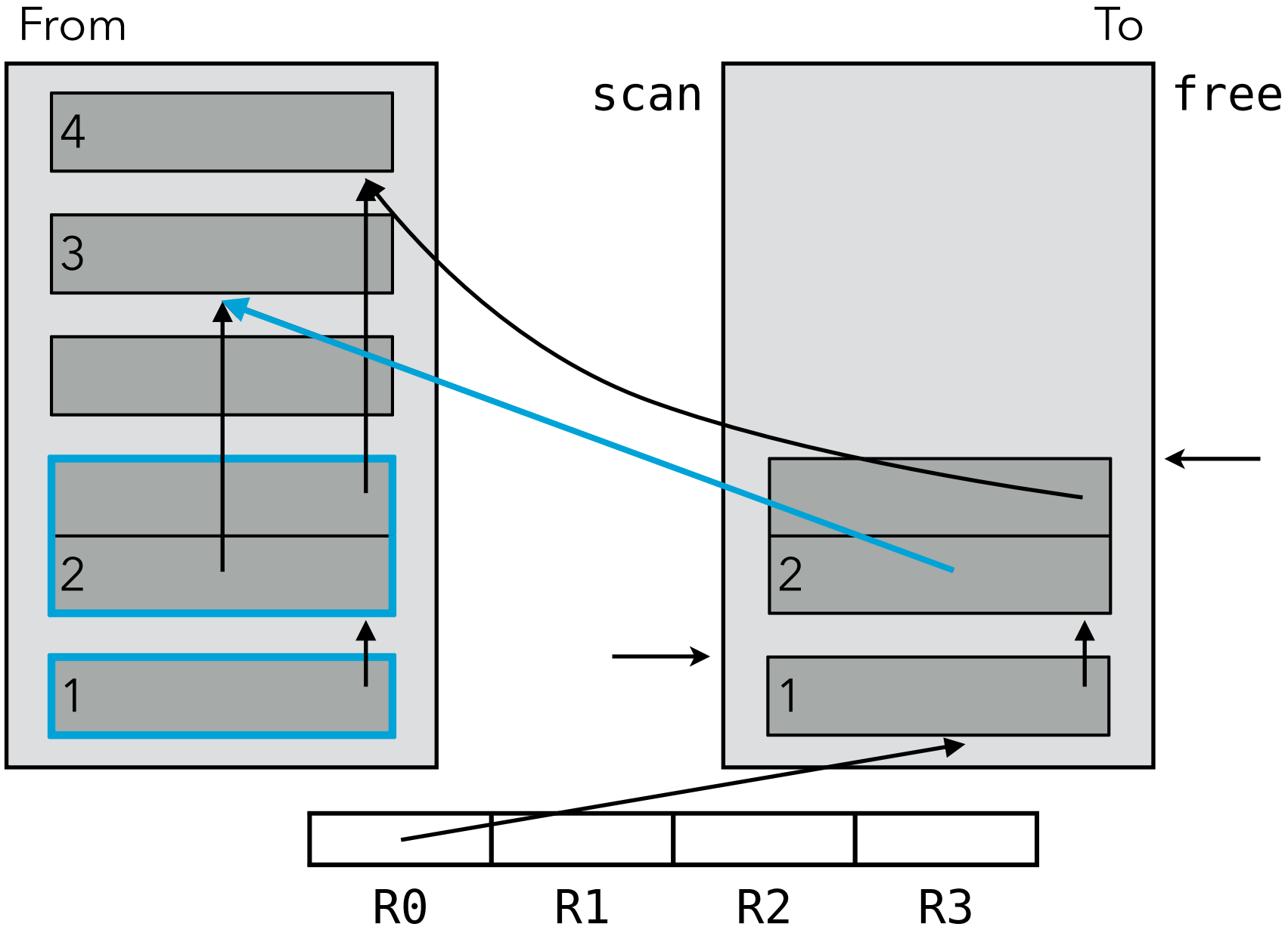
Cheney's copying GC



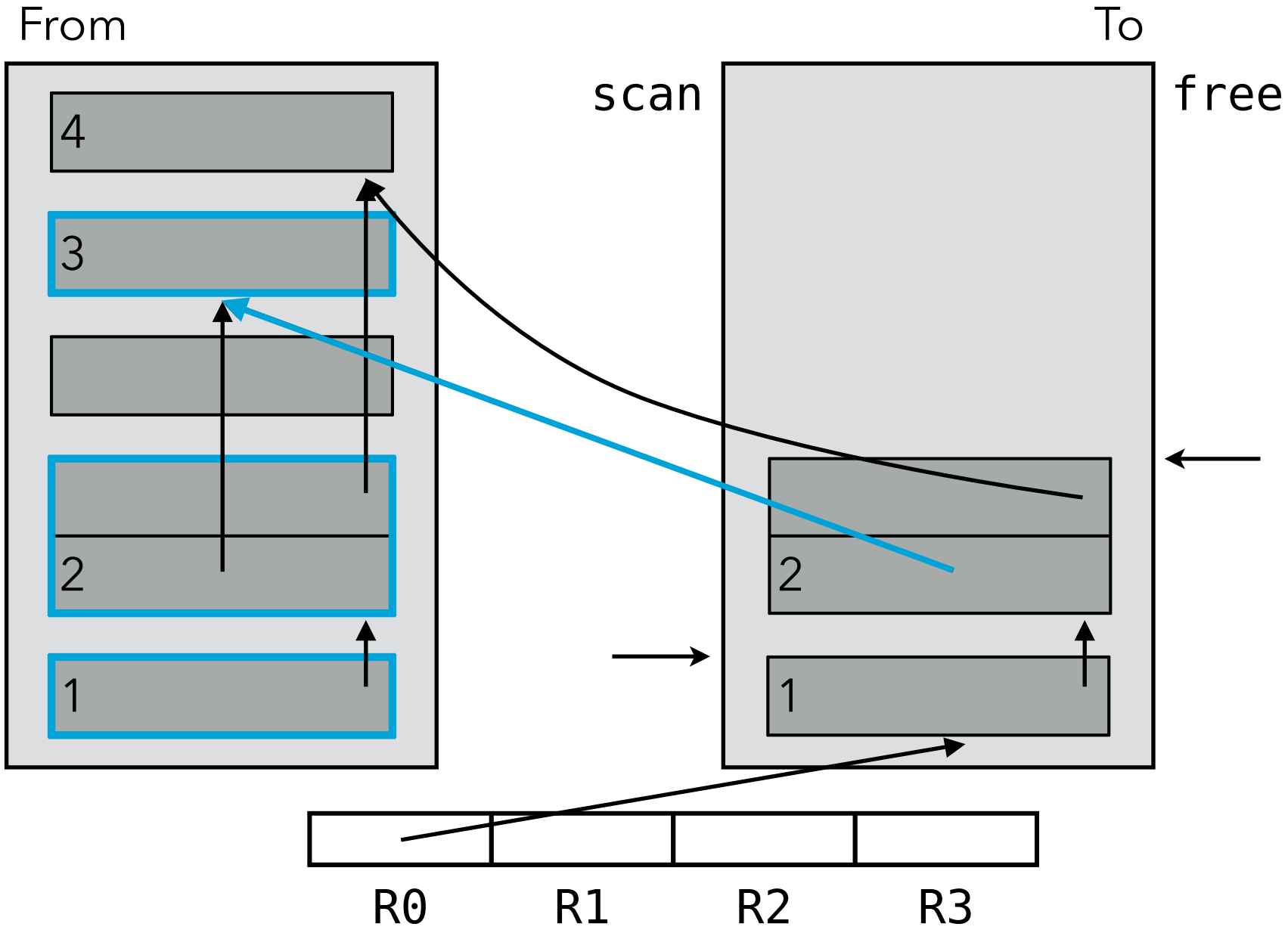
Cheney's copying GC



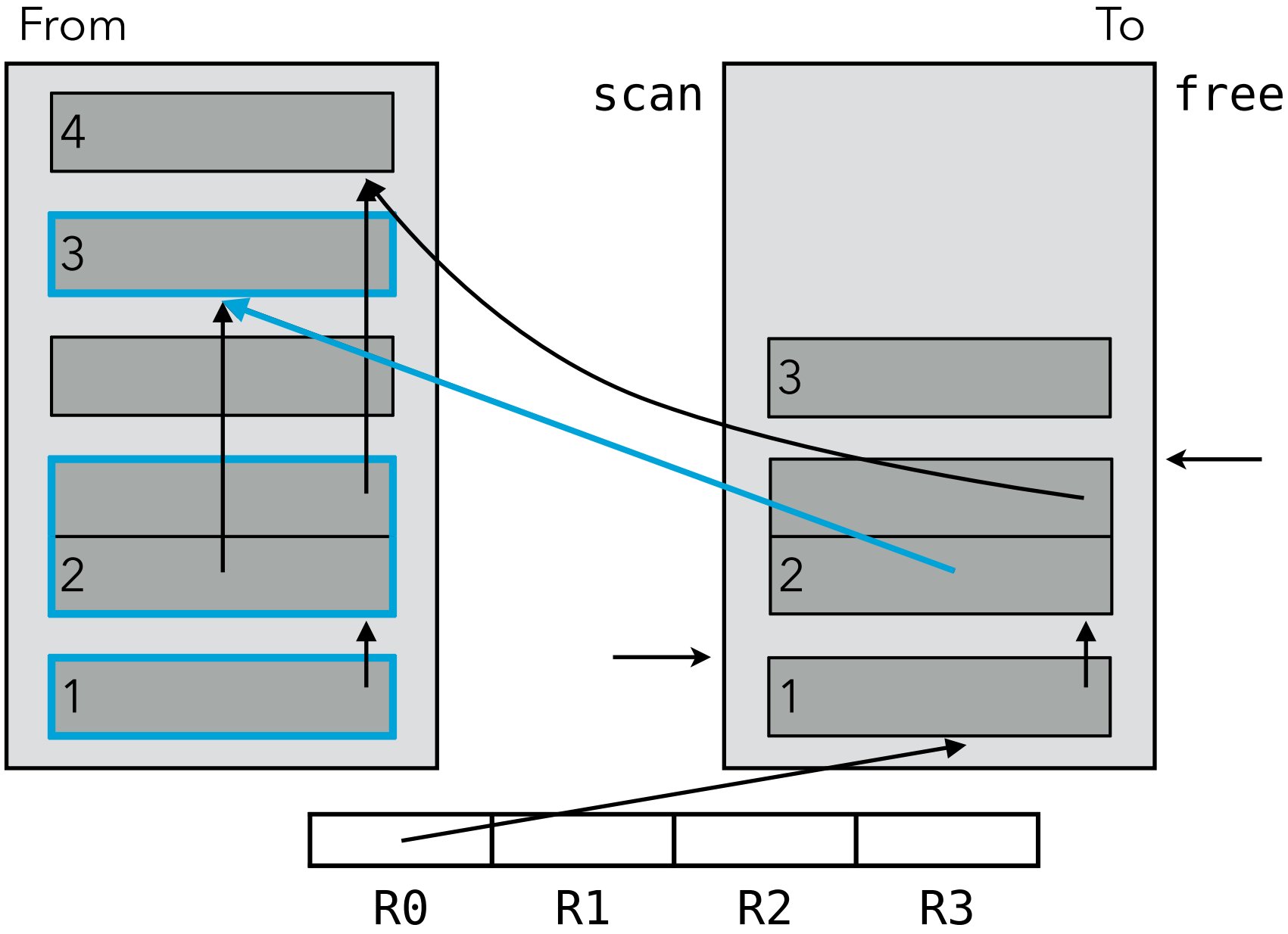
Cheney's copying GC



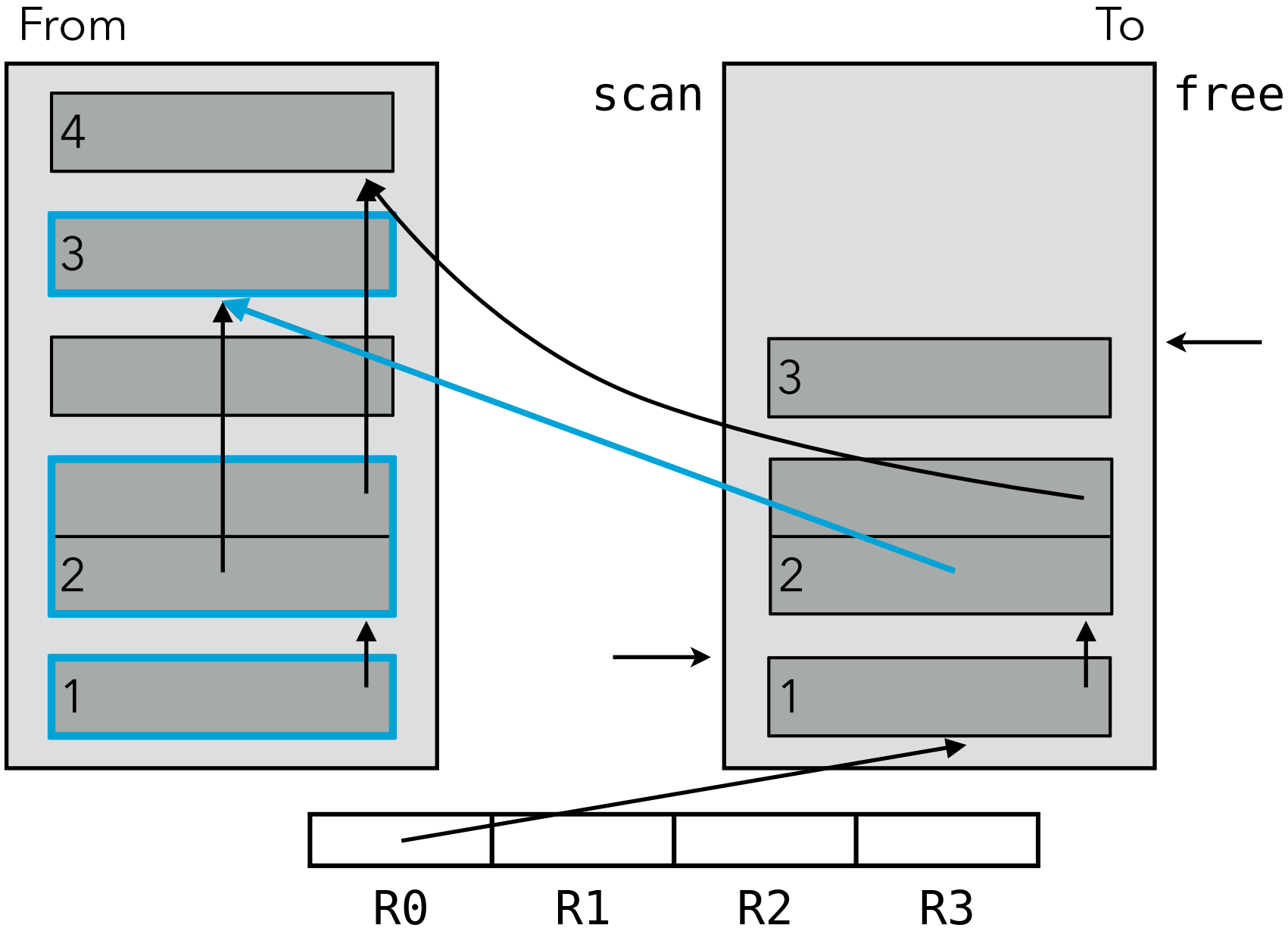
Cheney's copying GC



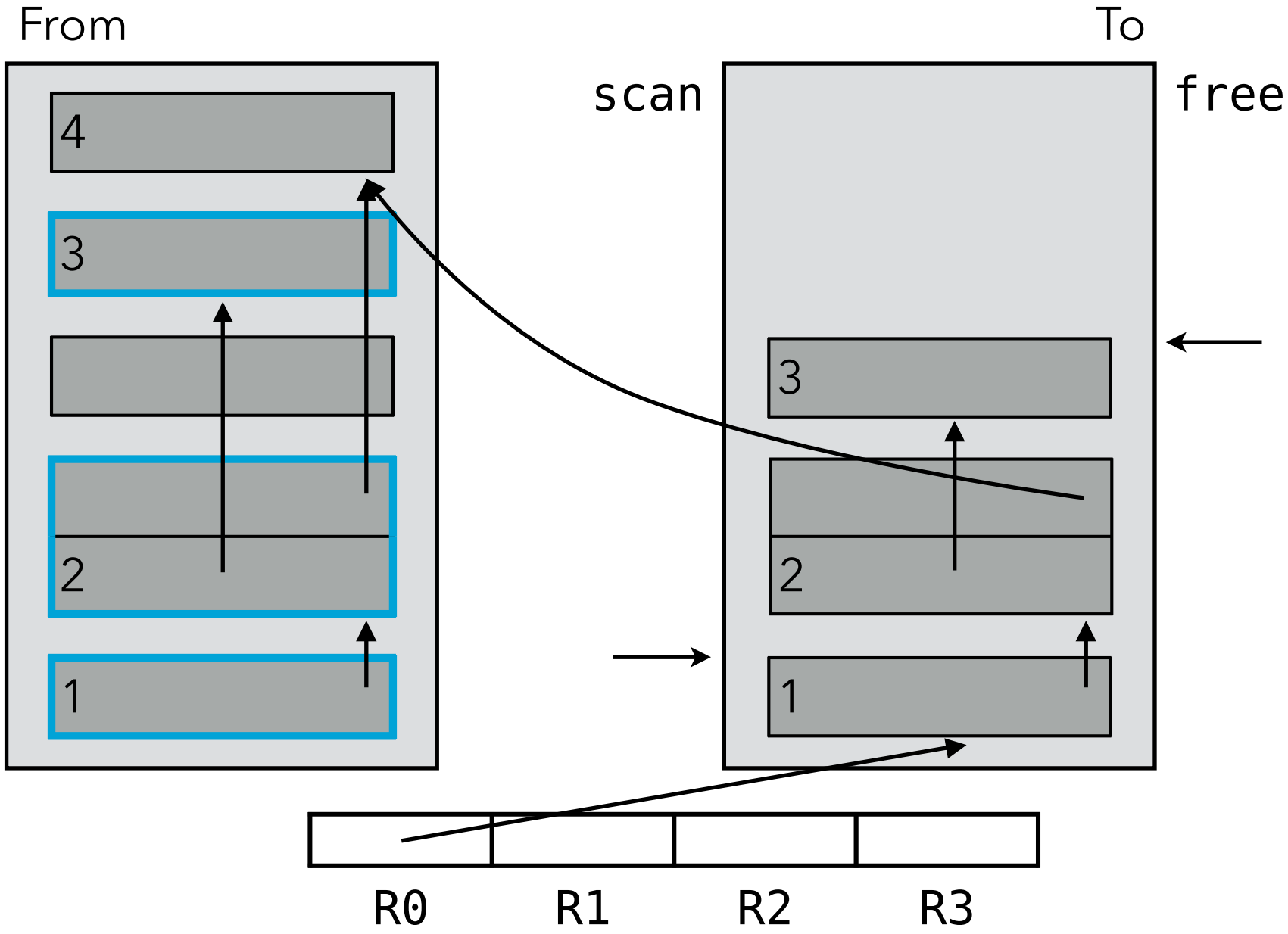
Cheney's copying GC



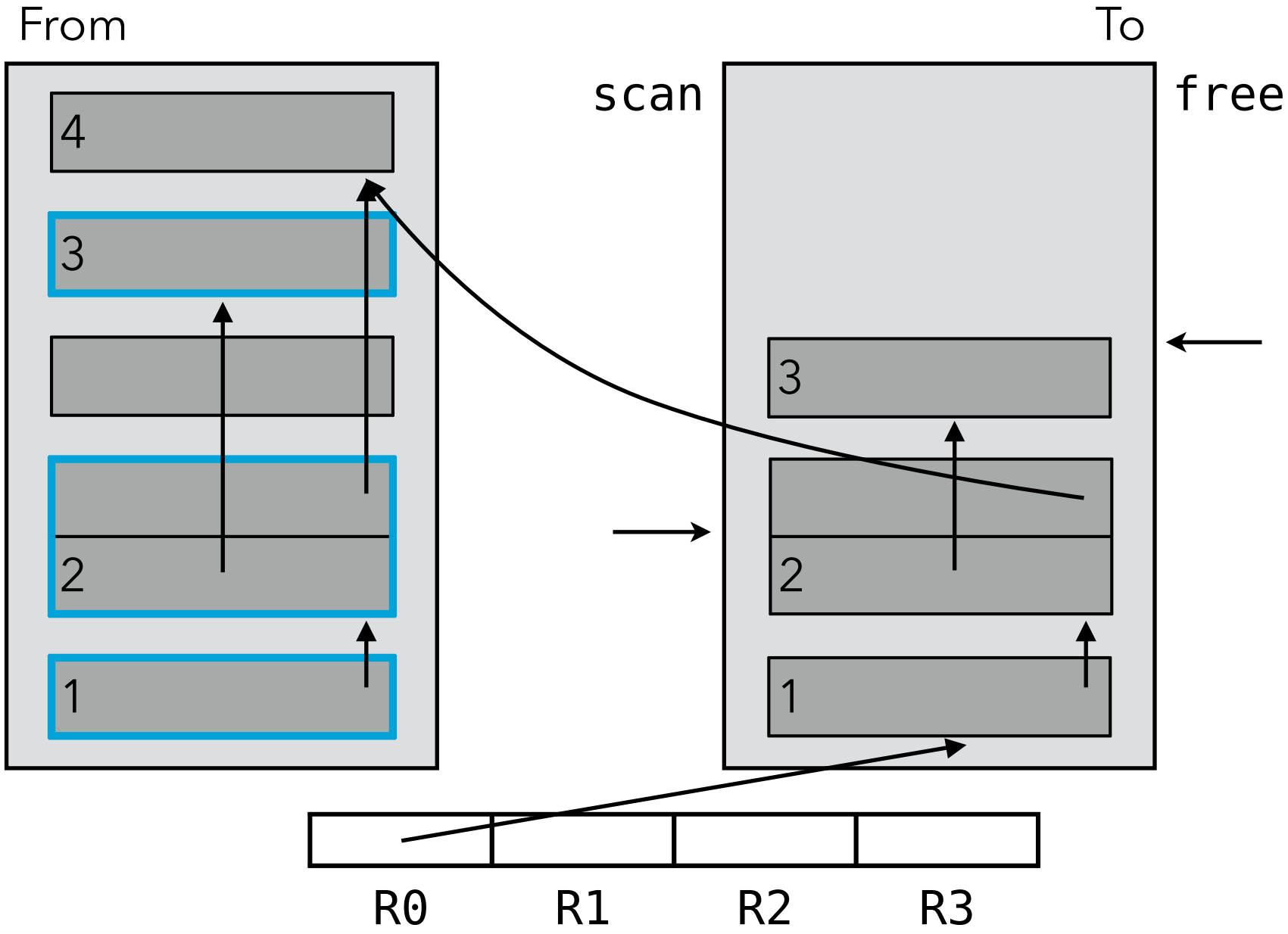
Cheney's copying GC



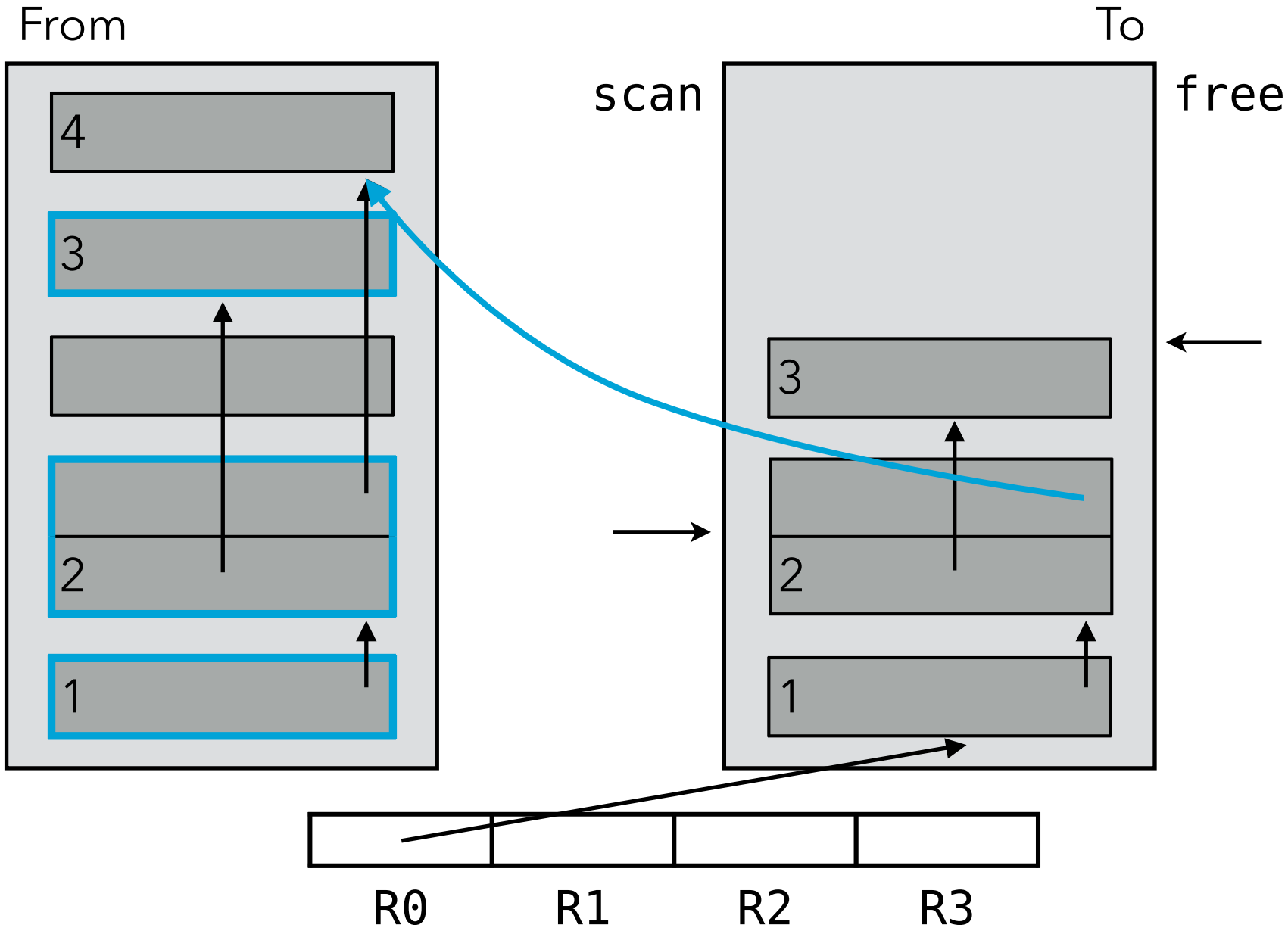
Cheney's copying GC



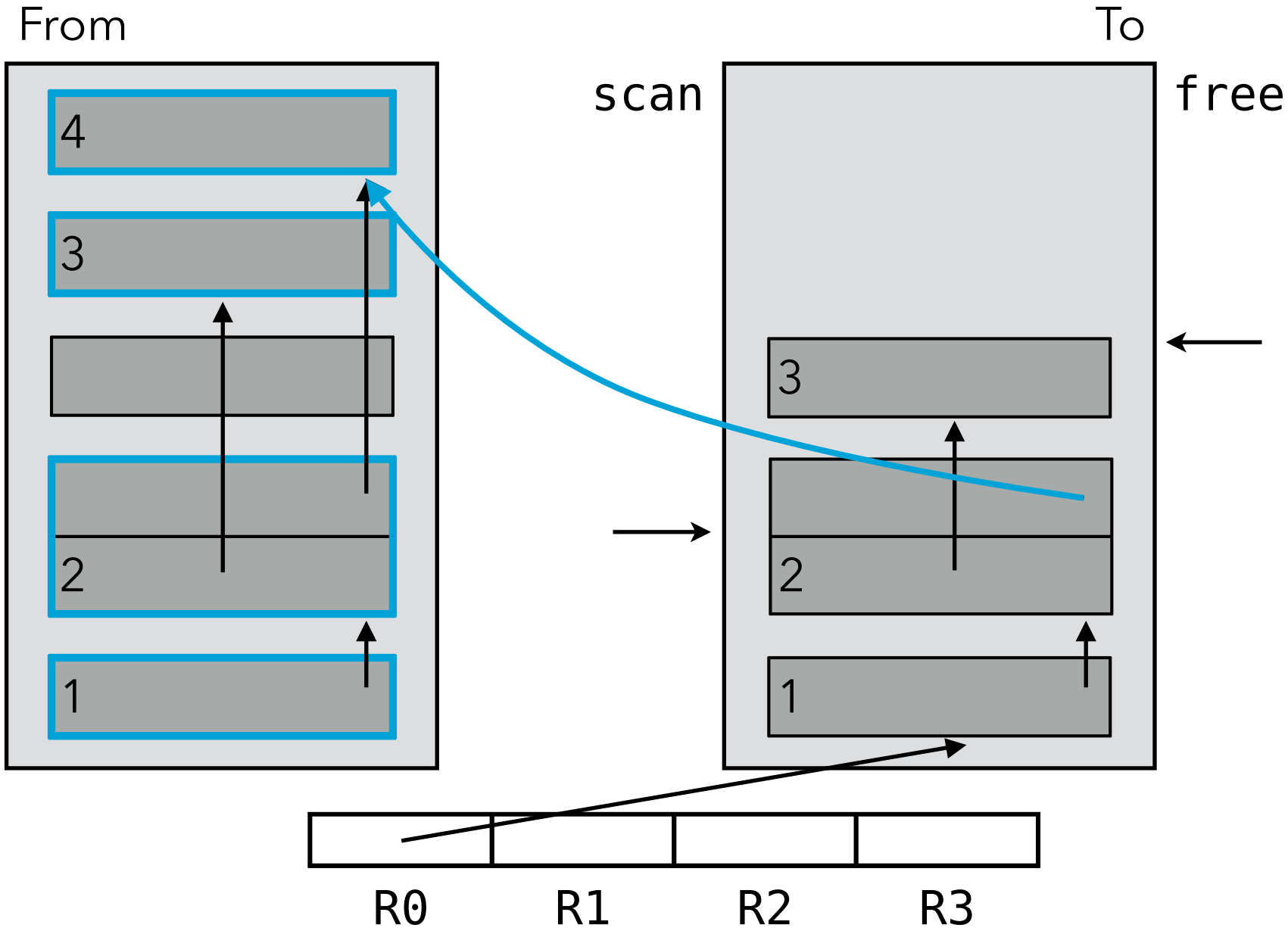
Cheney's copying GC



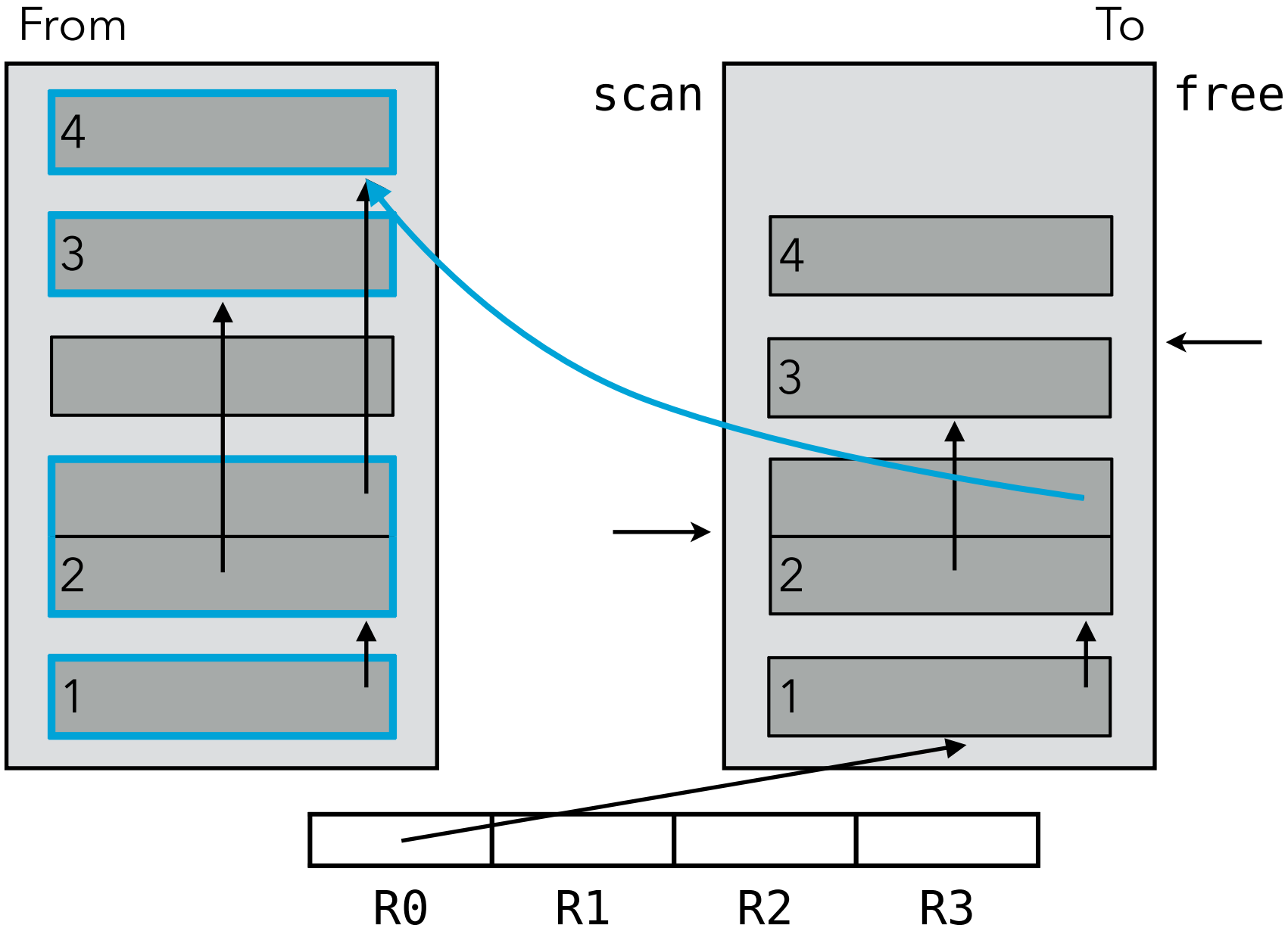
Cheney's copying GC



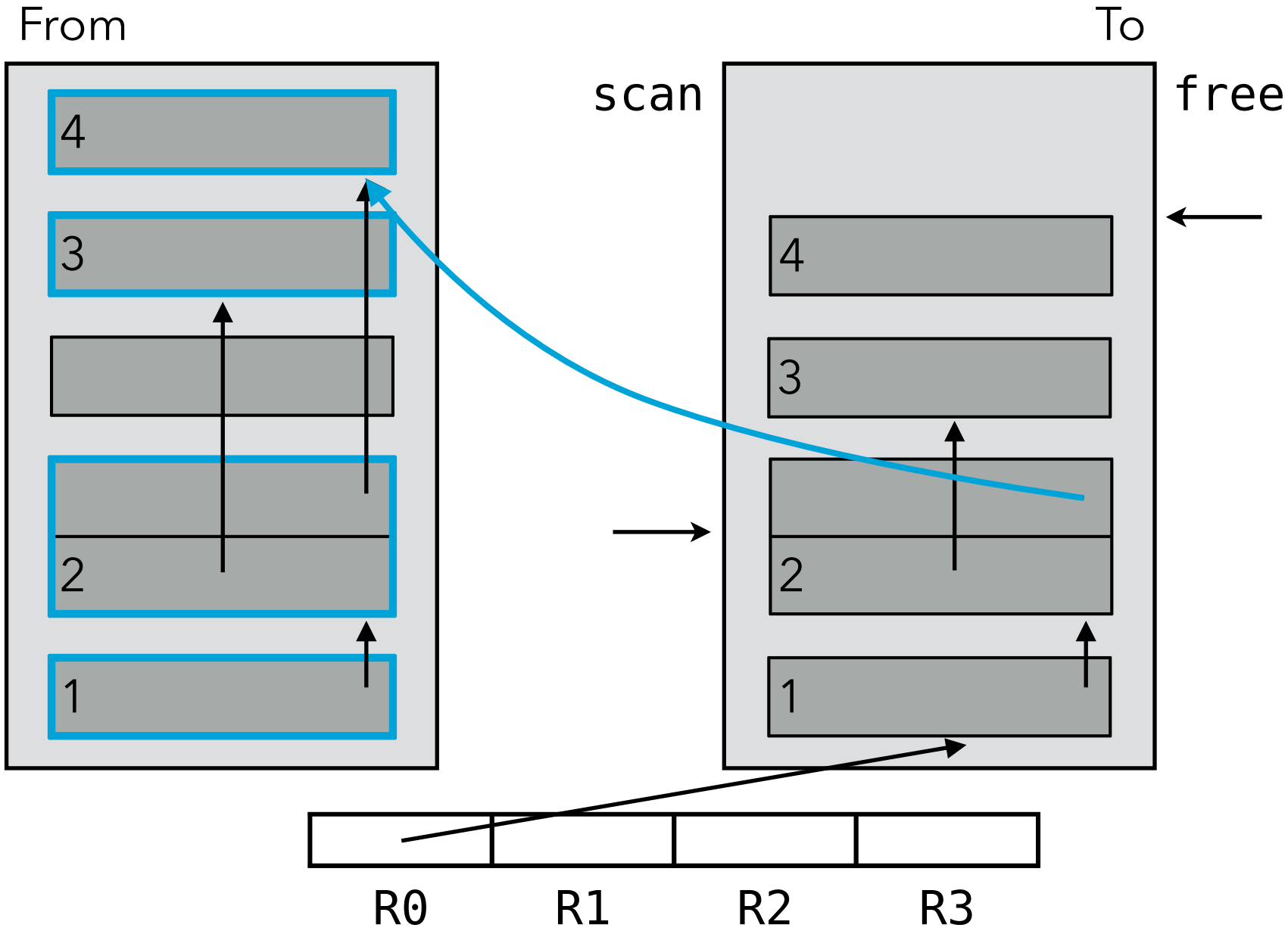
Cheney's copying GC



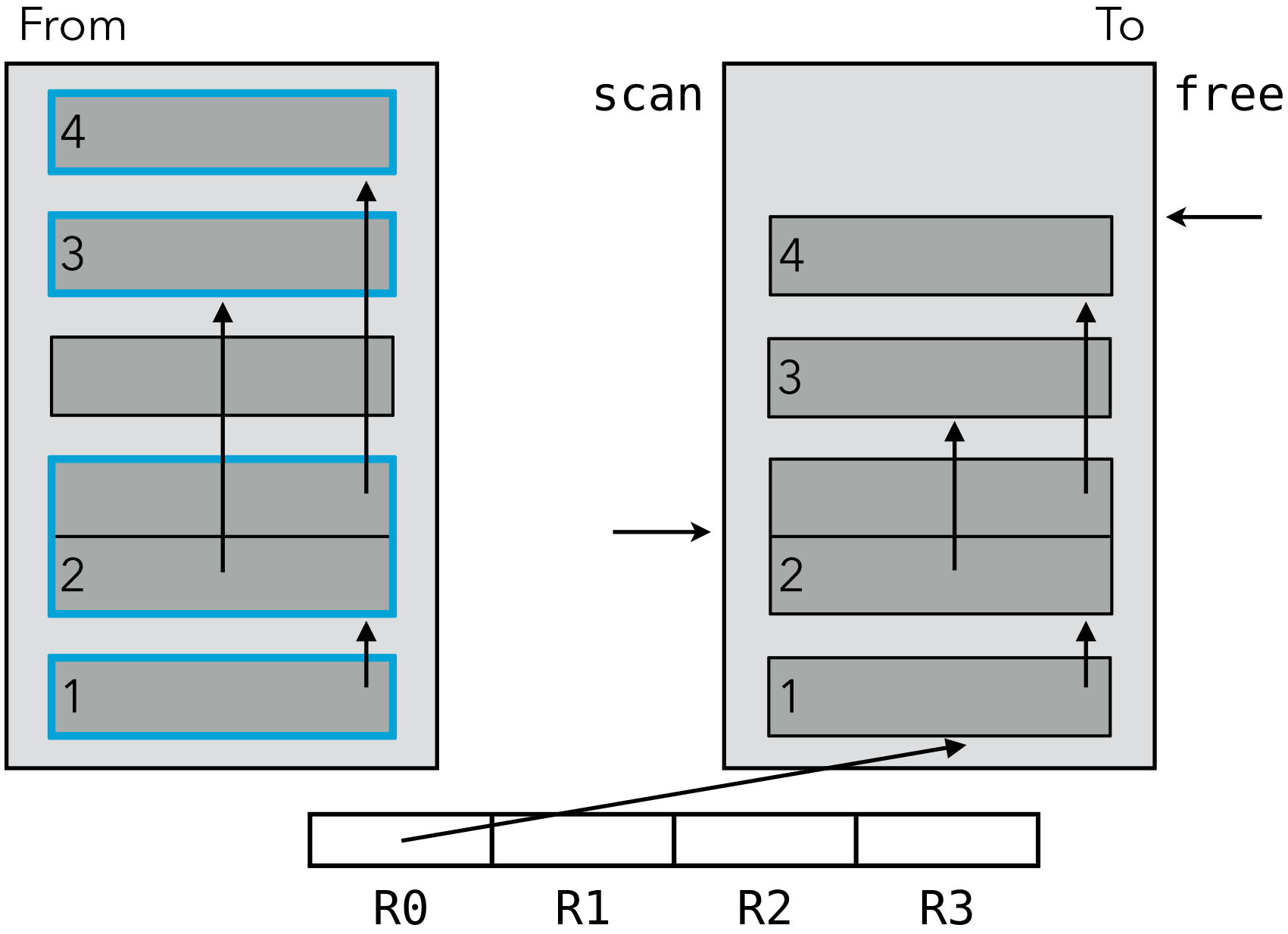
Cheney's copying GC



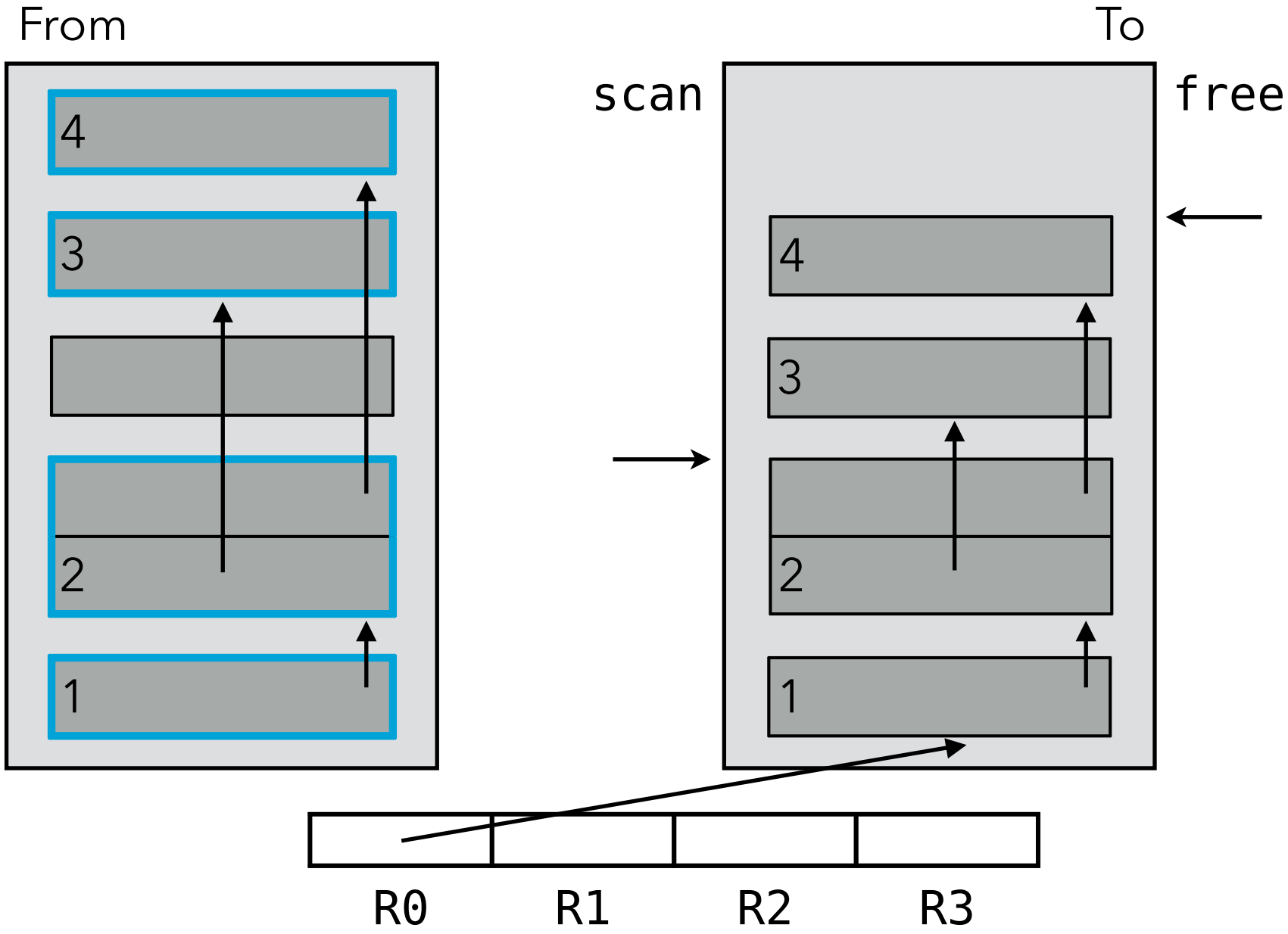
Cheney's copying GC



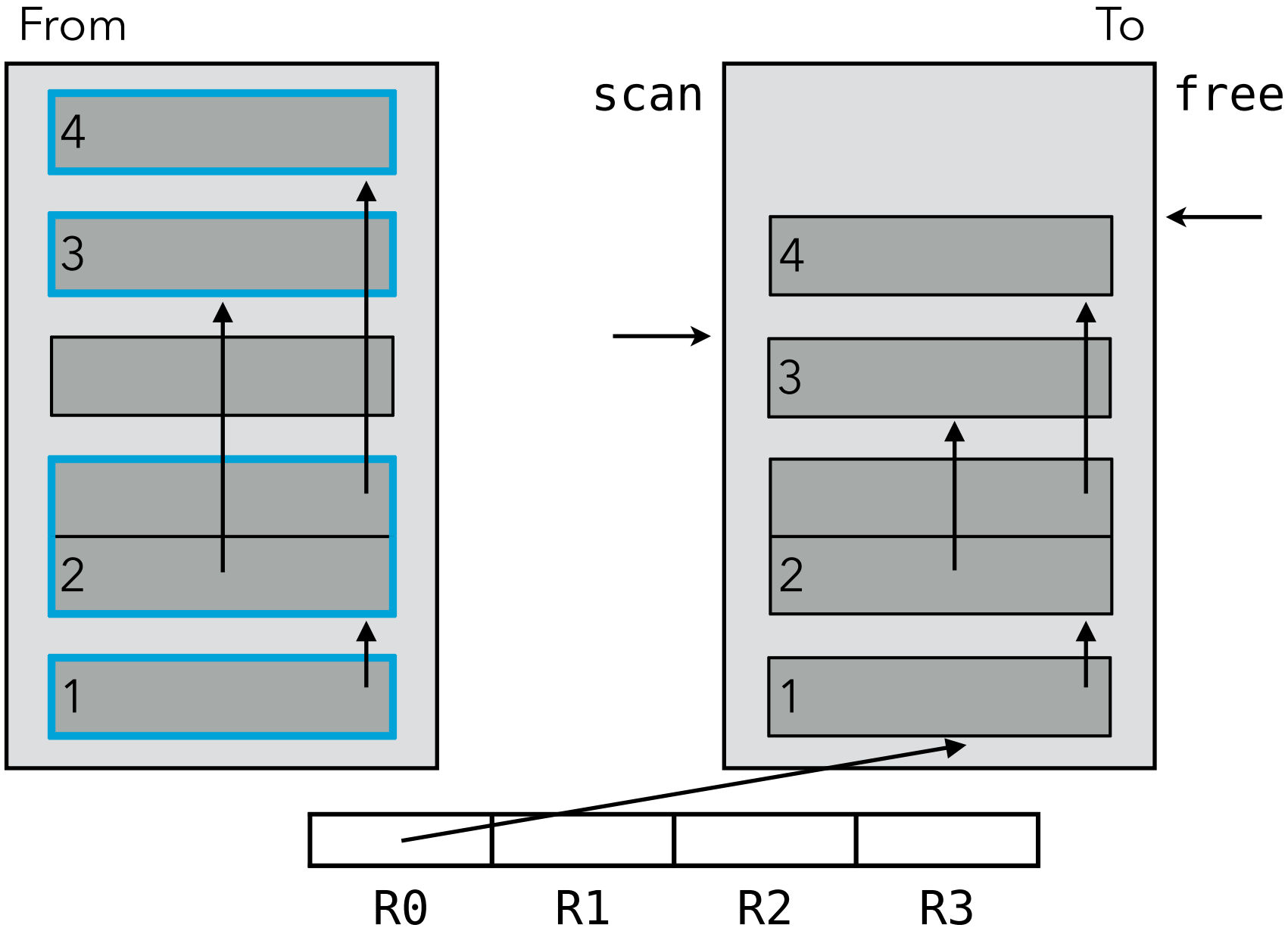
Cheney's copying GC



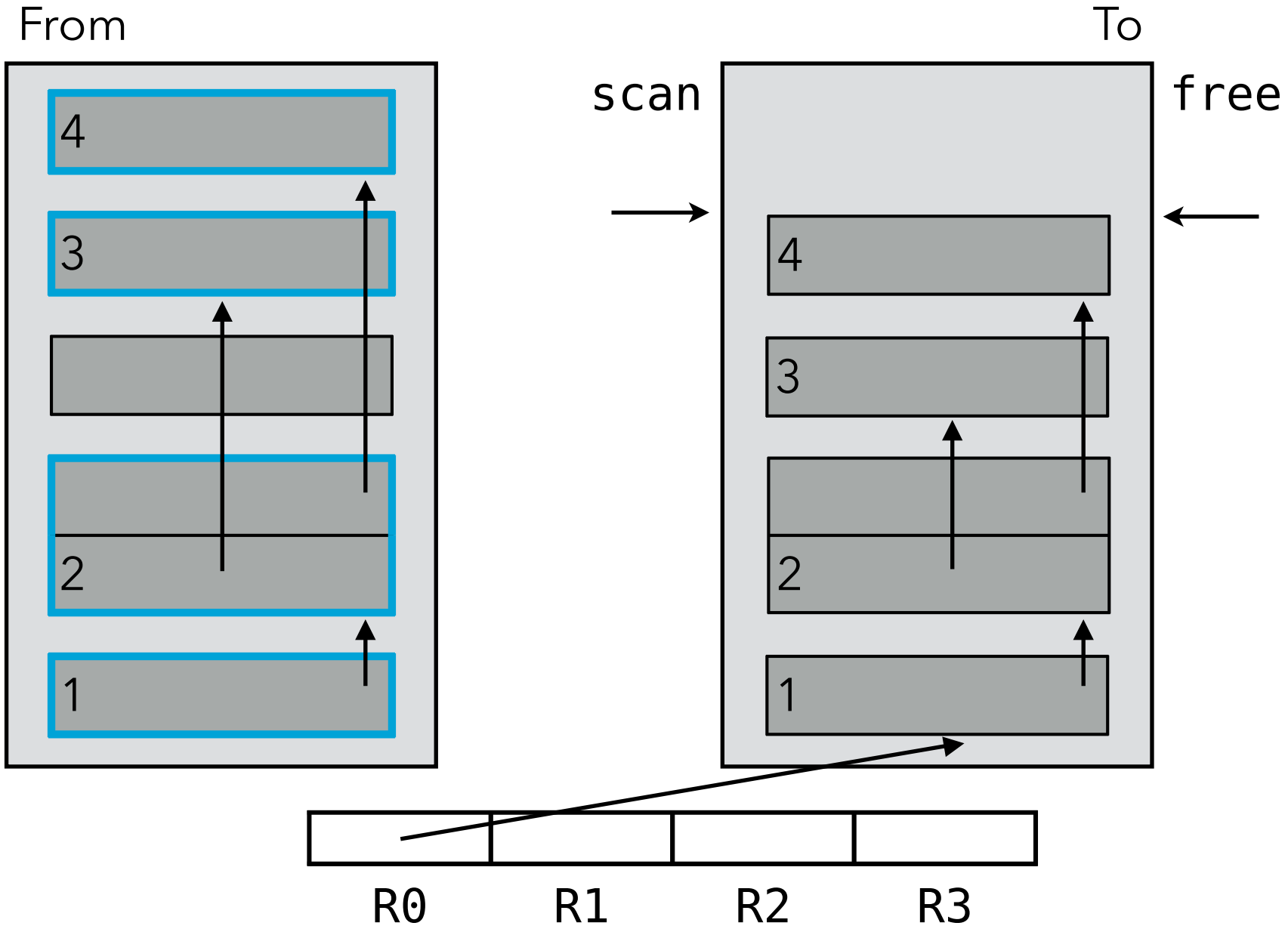
Cheney's copying GC



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.

GC technique #4: generational GC

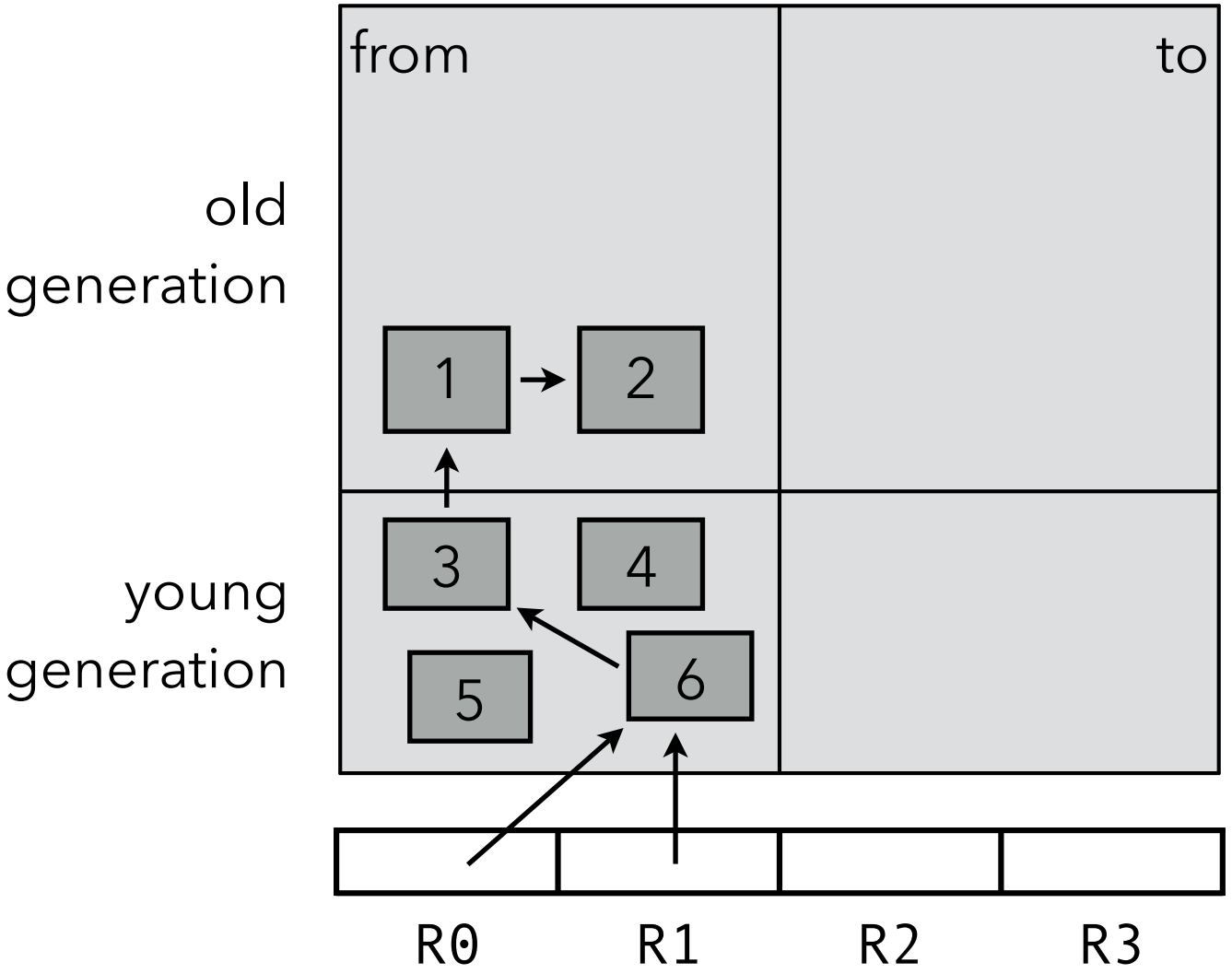
Generational GC

Empirical observation suggests that a large majority of the objects die young, while a small minority lives for very long.
The idea of **generational garbage collection** is to partition objects in generations – based on their age – and to collect the young generation(s) more often than the old one(s).
This should improve the amount of memory collected per objects visited. In a copying GC, this also avoids repeatedly copying long-lived objects.
Note: The principles of generational garbage collection will be presented here in the context of copying GCs, but can also be applied to other GCs like mark & sweep.

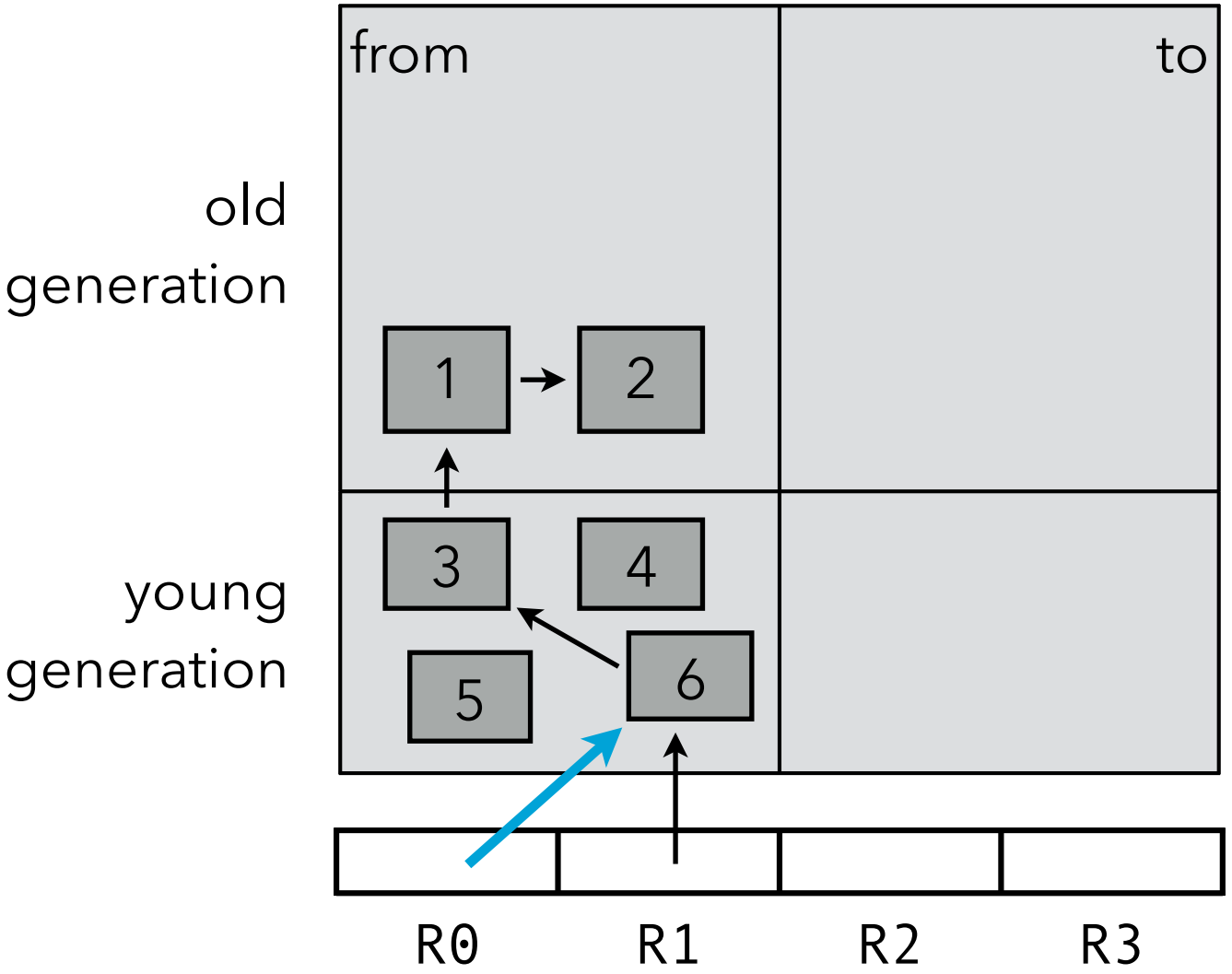
Generational GC

In a generational GC, objects are partitioned into a given number (often 2) of generations. The younger a generation is, the smaller the amount of memory reserved to it.
All objects are initially allocated in the youngest generation.
When it is full, a **minor collection** is performed, to collect memory in that generation only. Some of the surviving objects are promoted to the next generation, based on a **promotion policy**.
When an older generation is itself full, a **major collection** is performed to collect memory in that generation and all the younger ones.

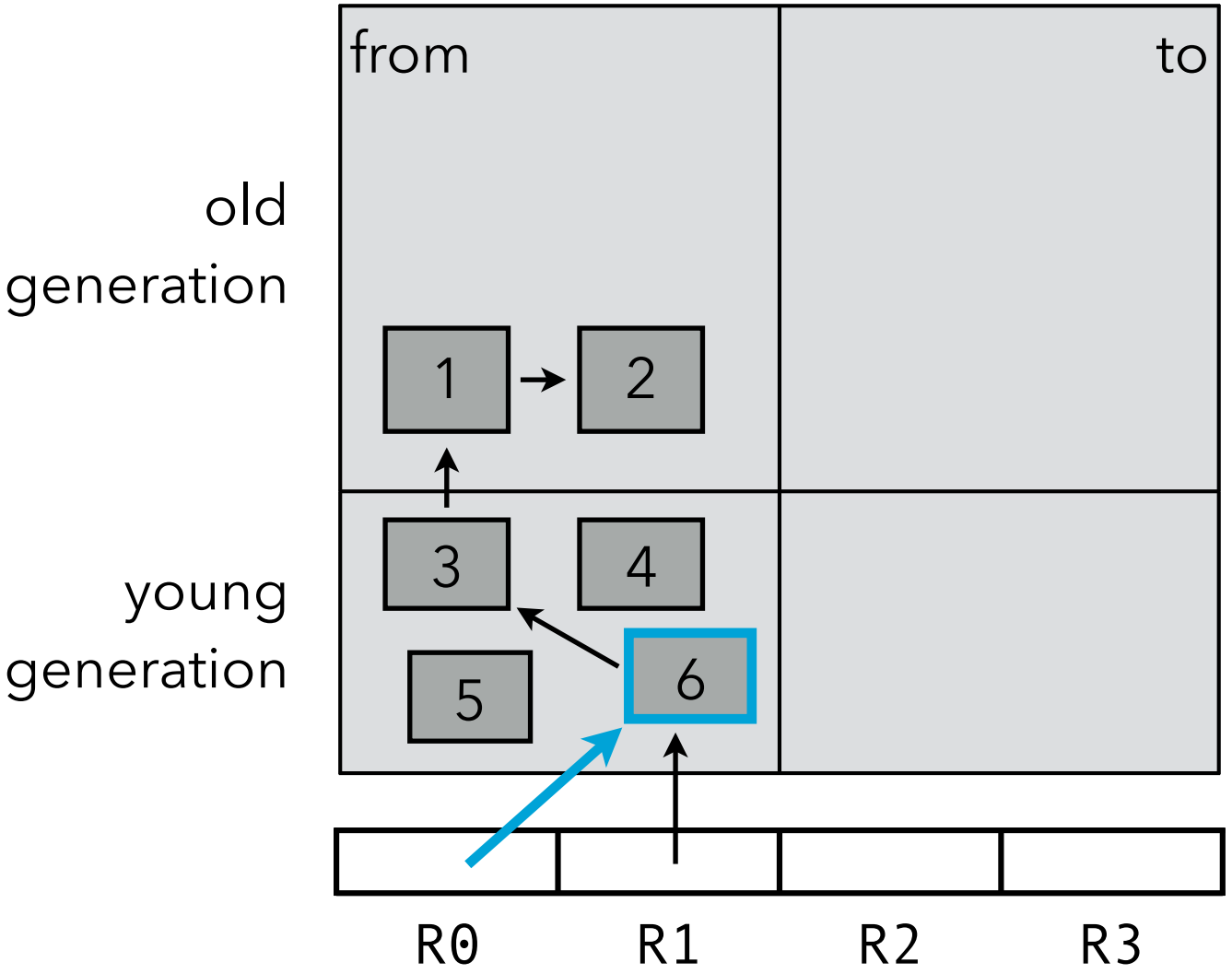
Minor collection example



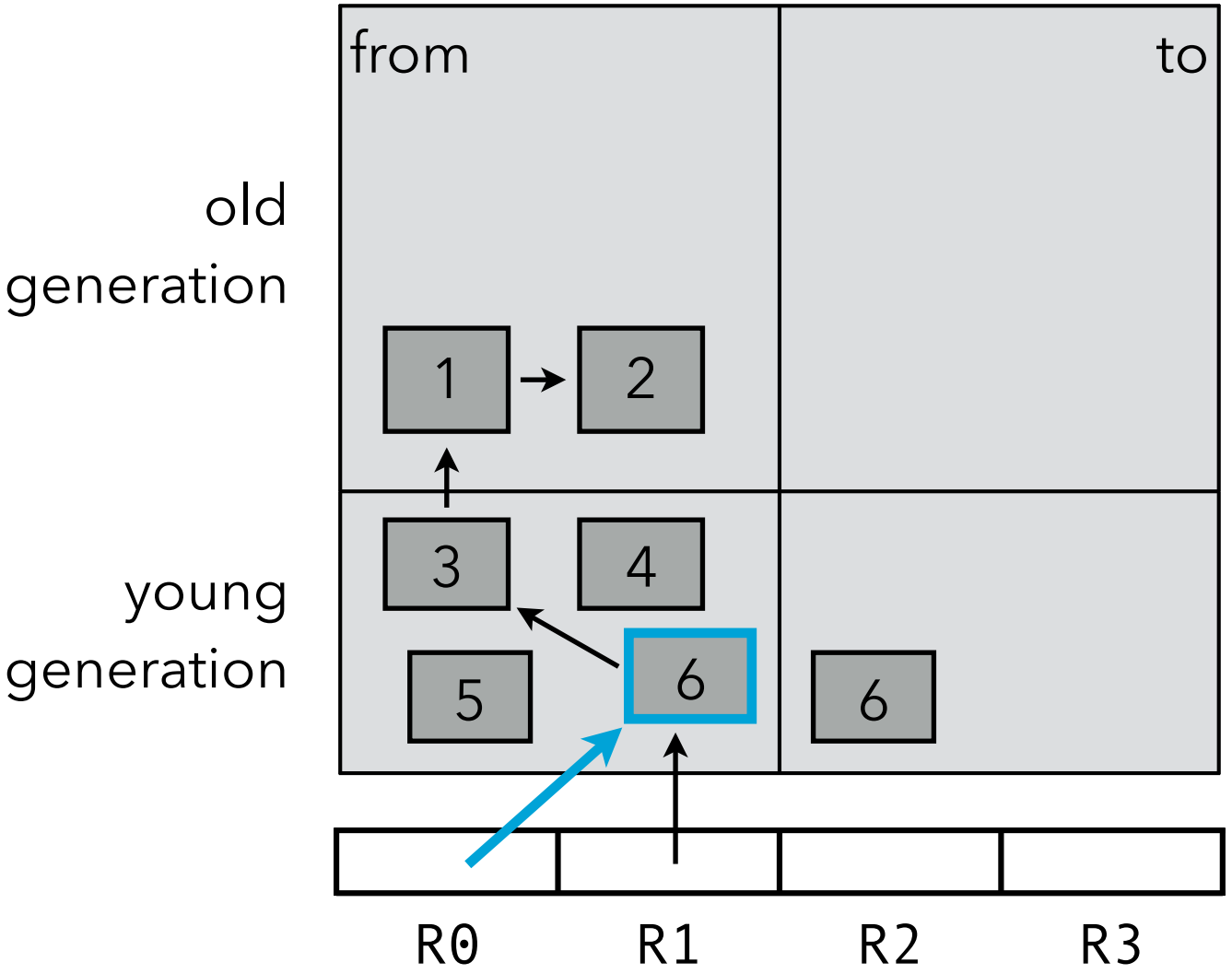
Minor collection example



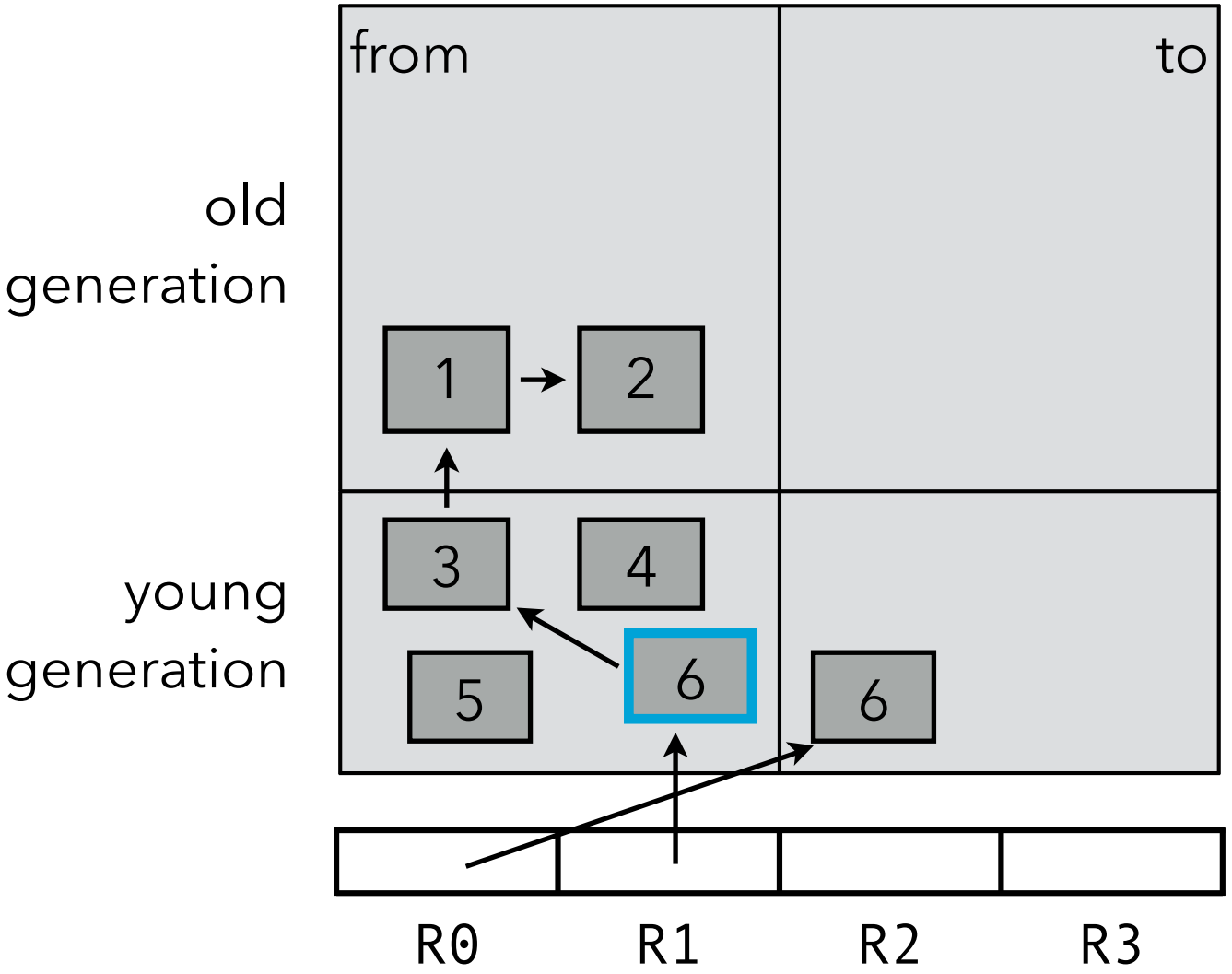
Minor collection example



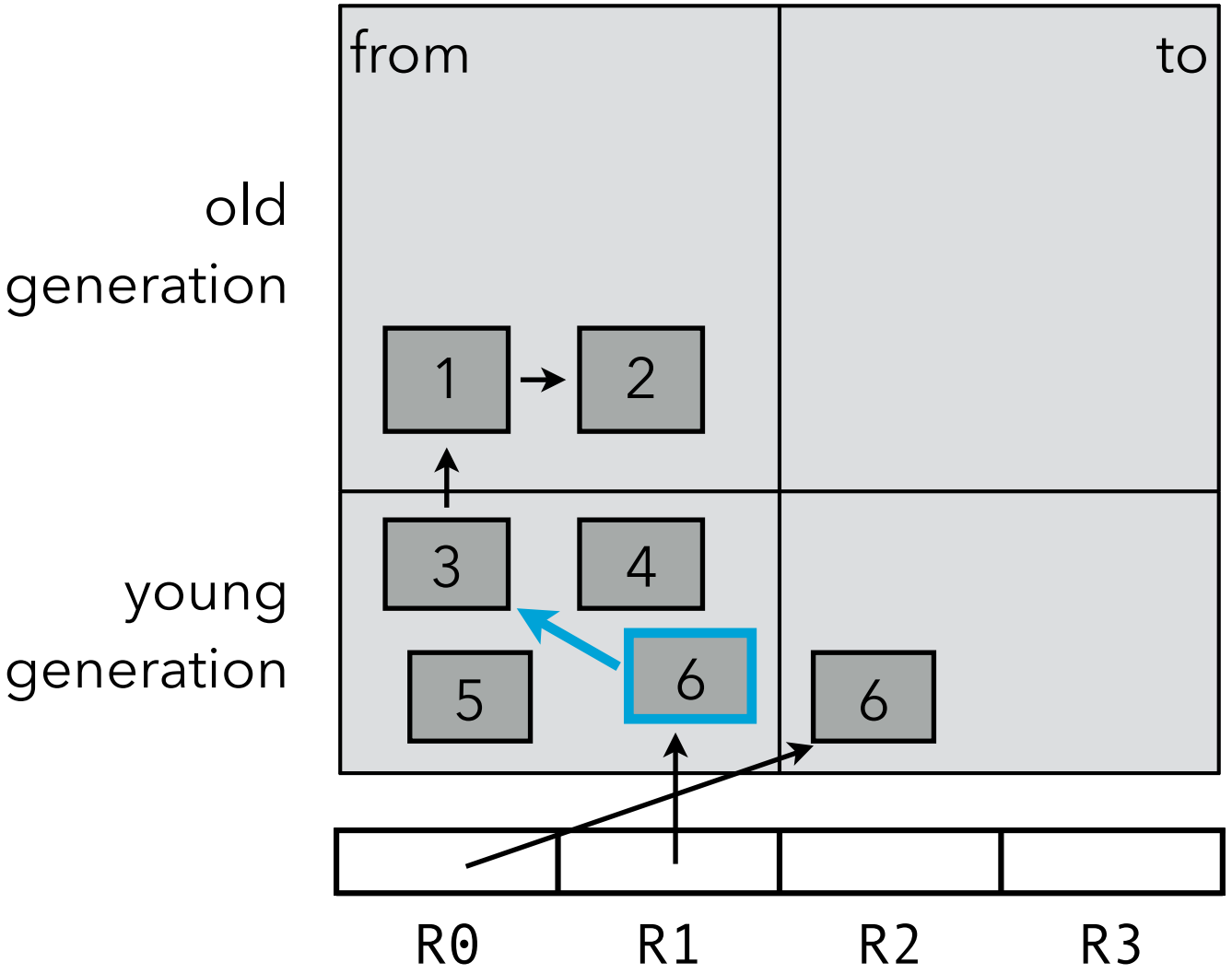
Minor collection example



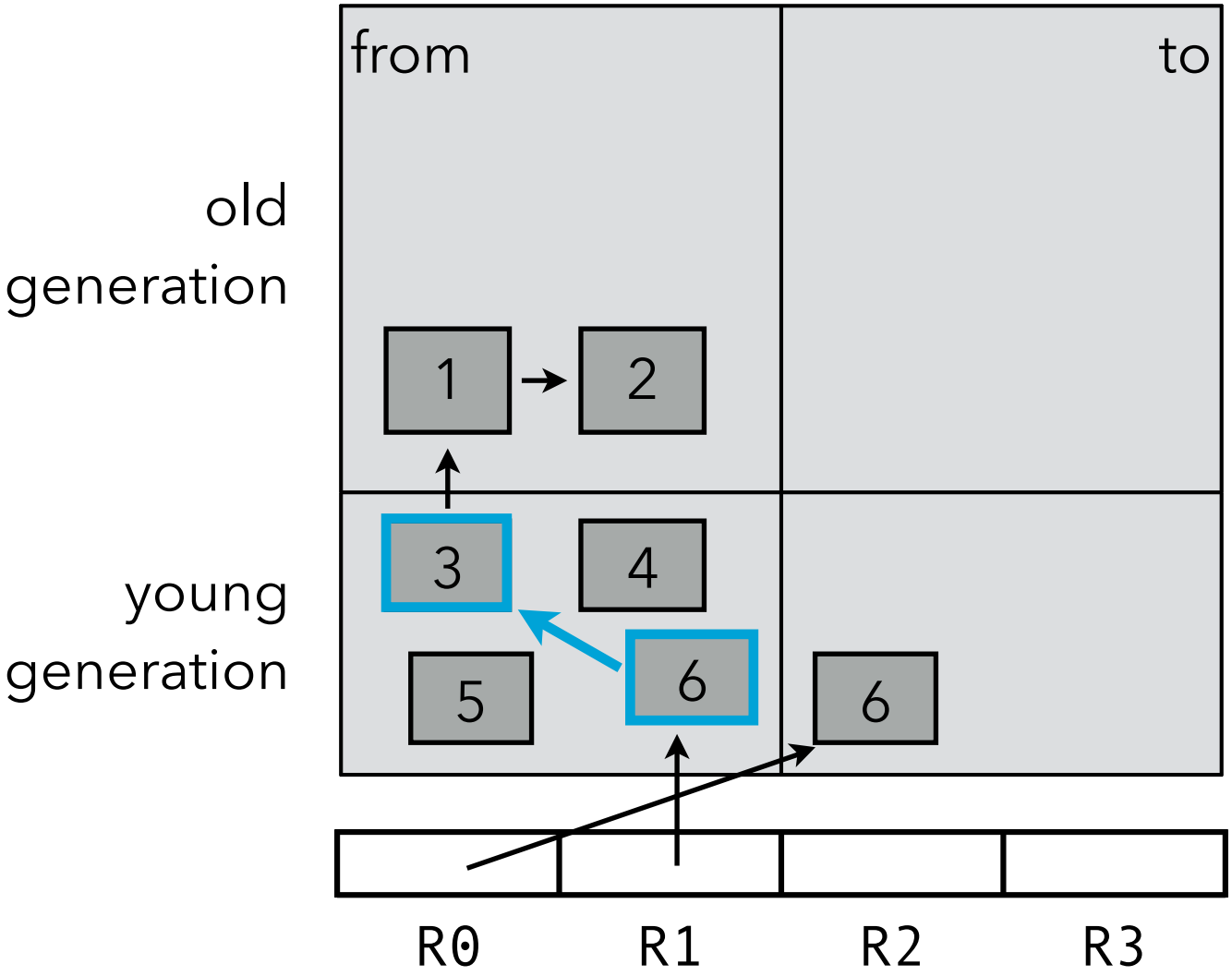
Minor collection example



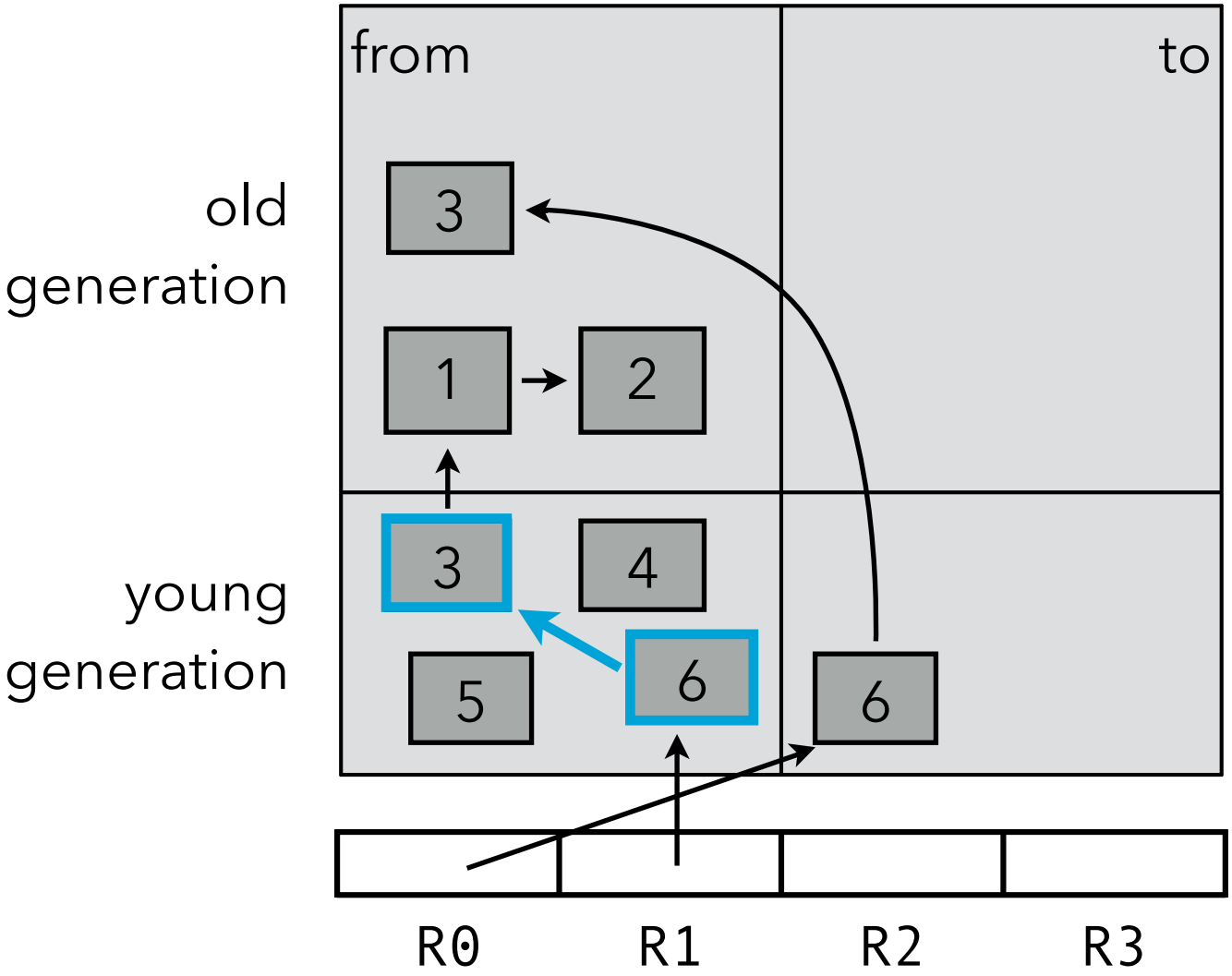
Minor collection example



Minor collection example

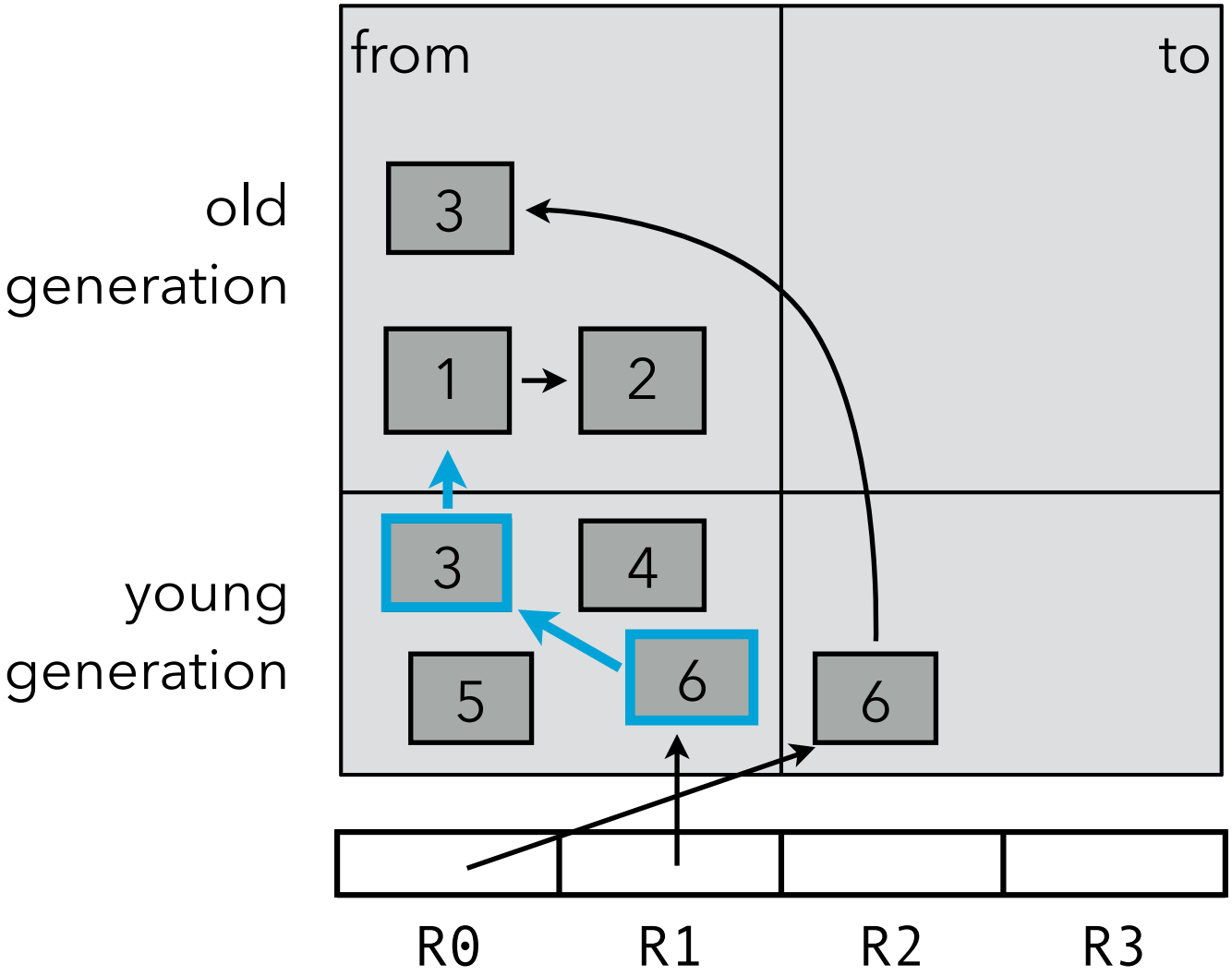


Minor collection example



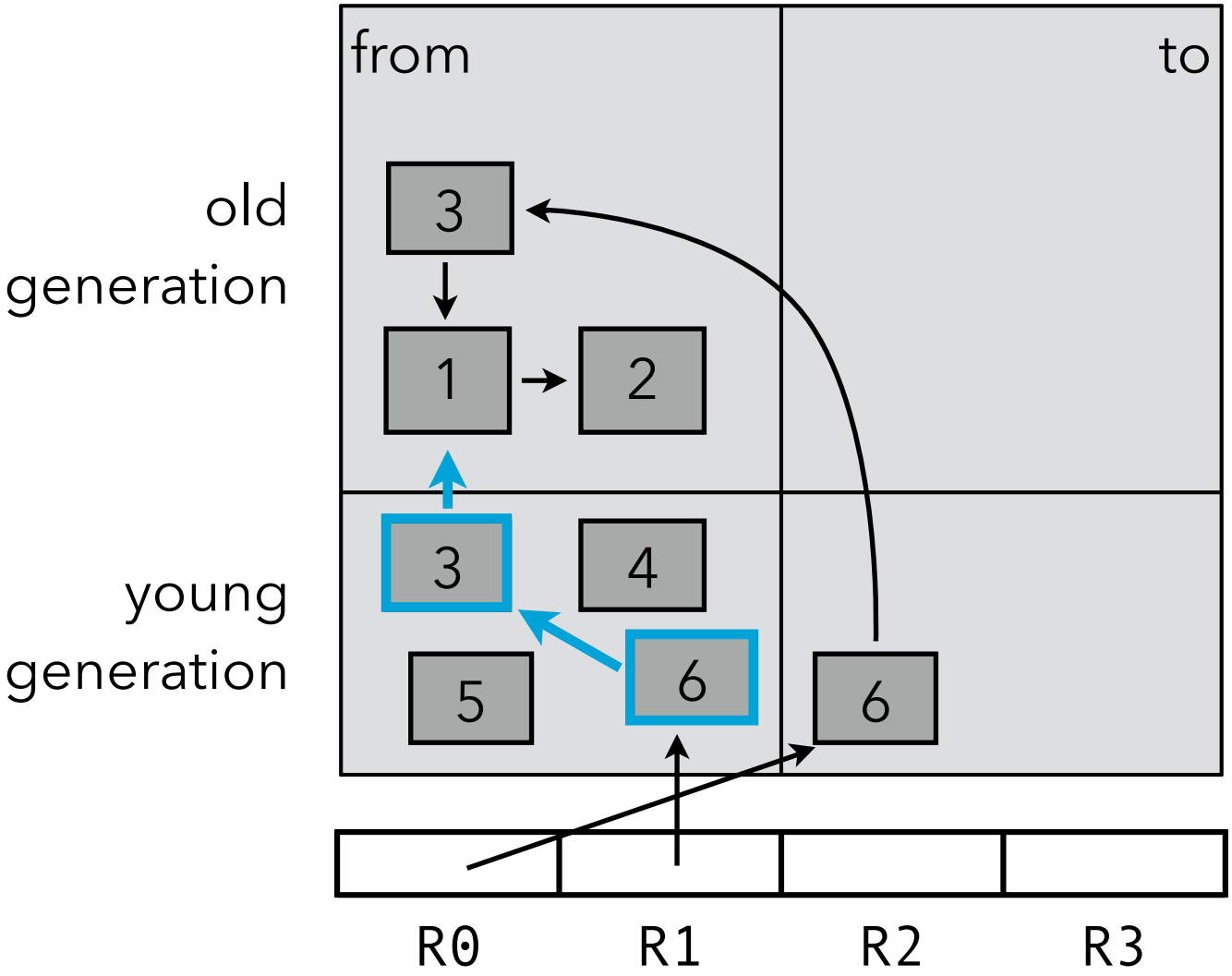
Object 3 is considered old enough to be promoted.

Minor collection example



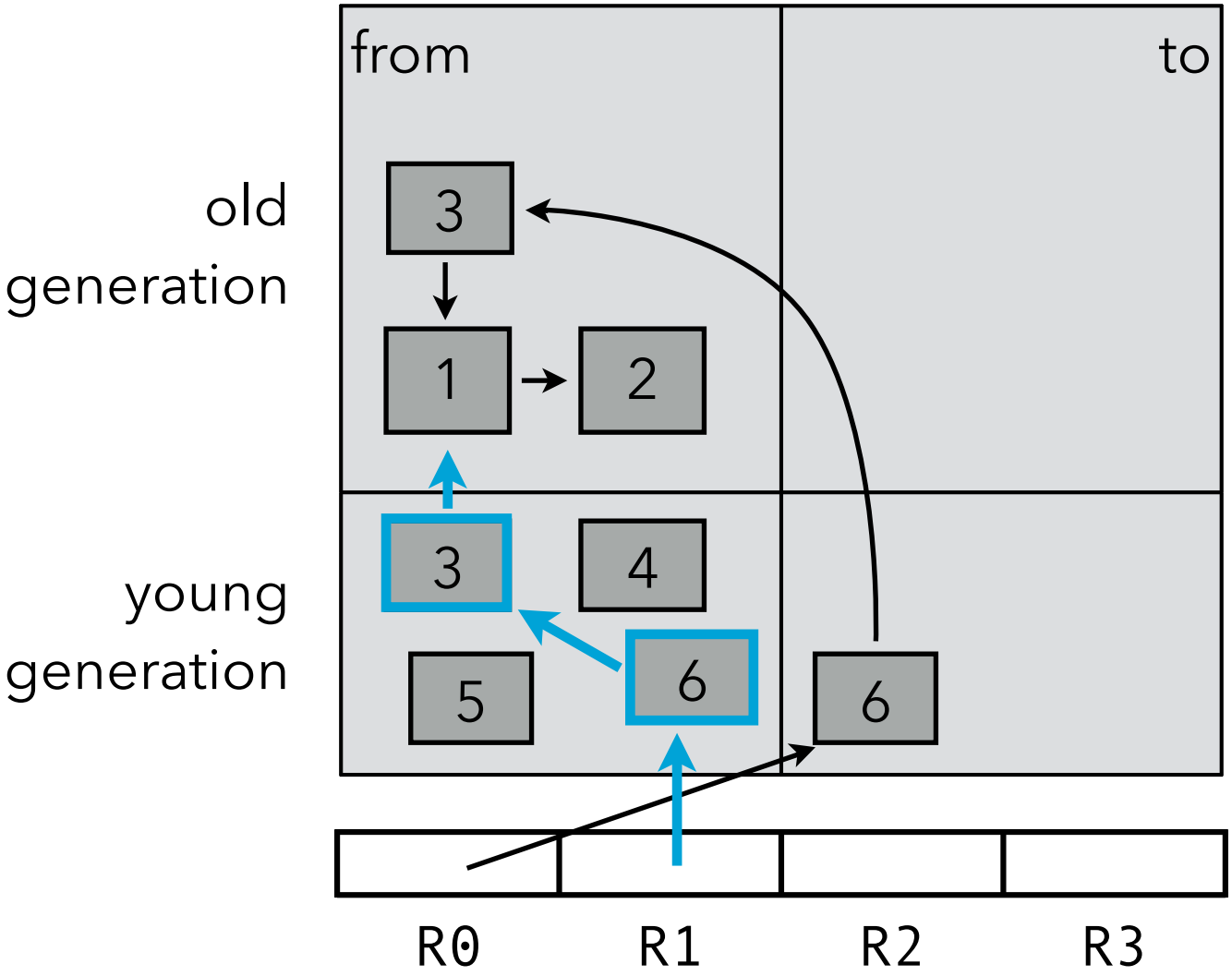
Object 3 is considered old enough to be promoted.

Minor collection example



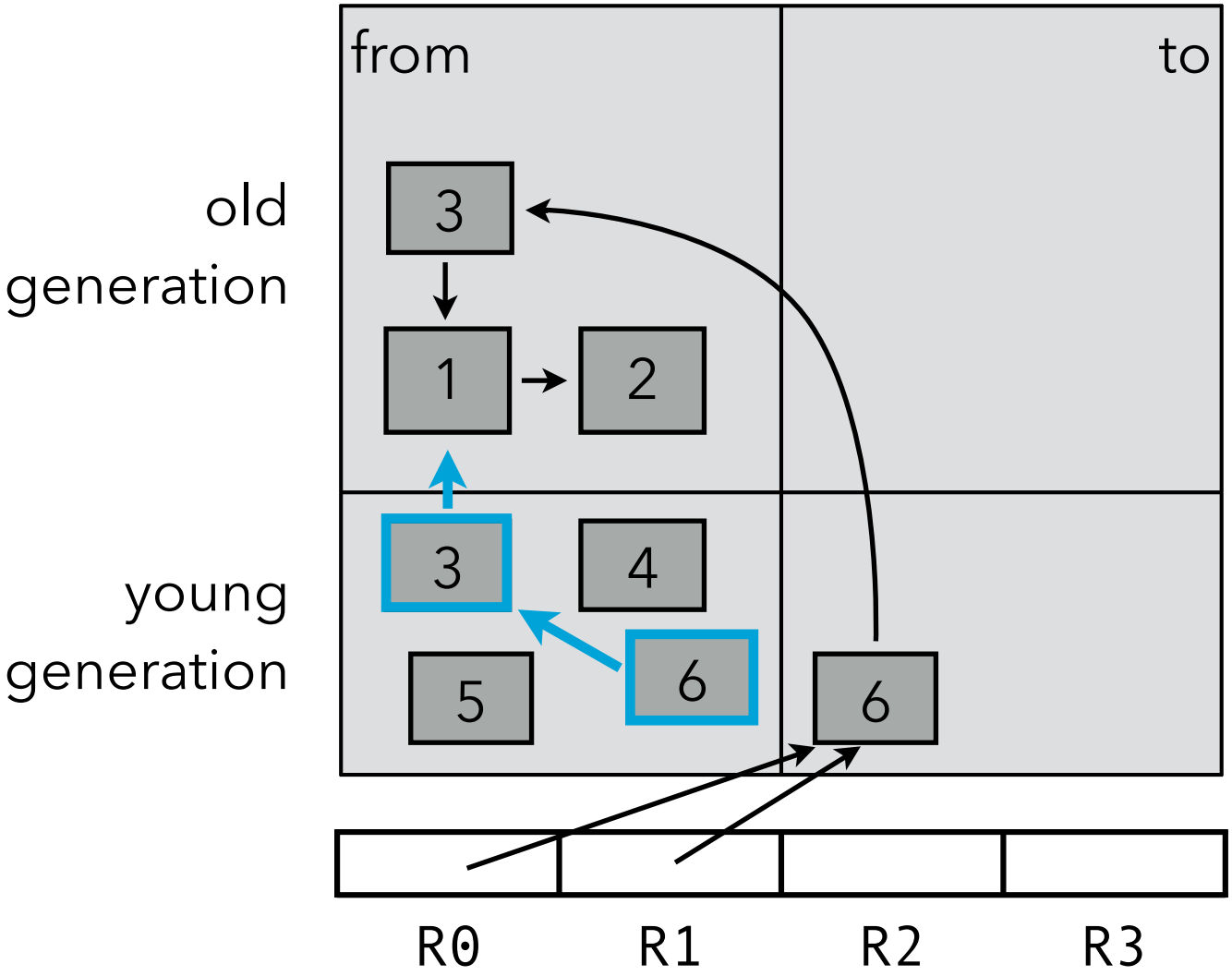
Object 3 is considered old enough to be promoted.

Minor collection example



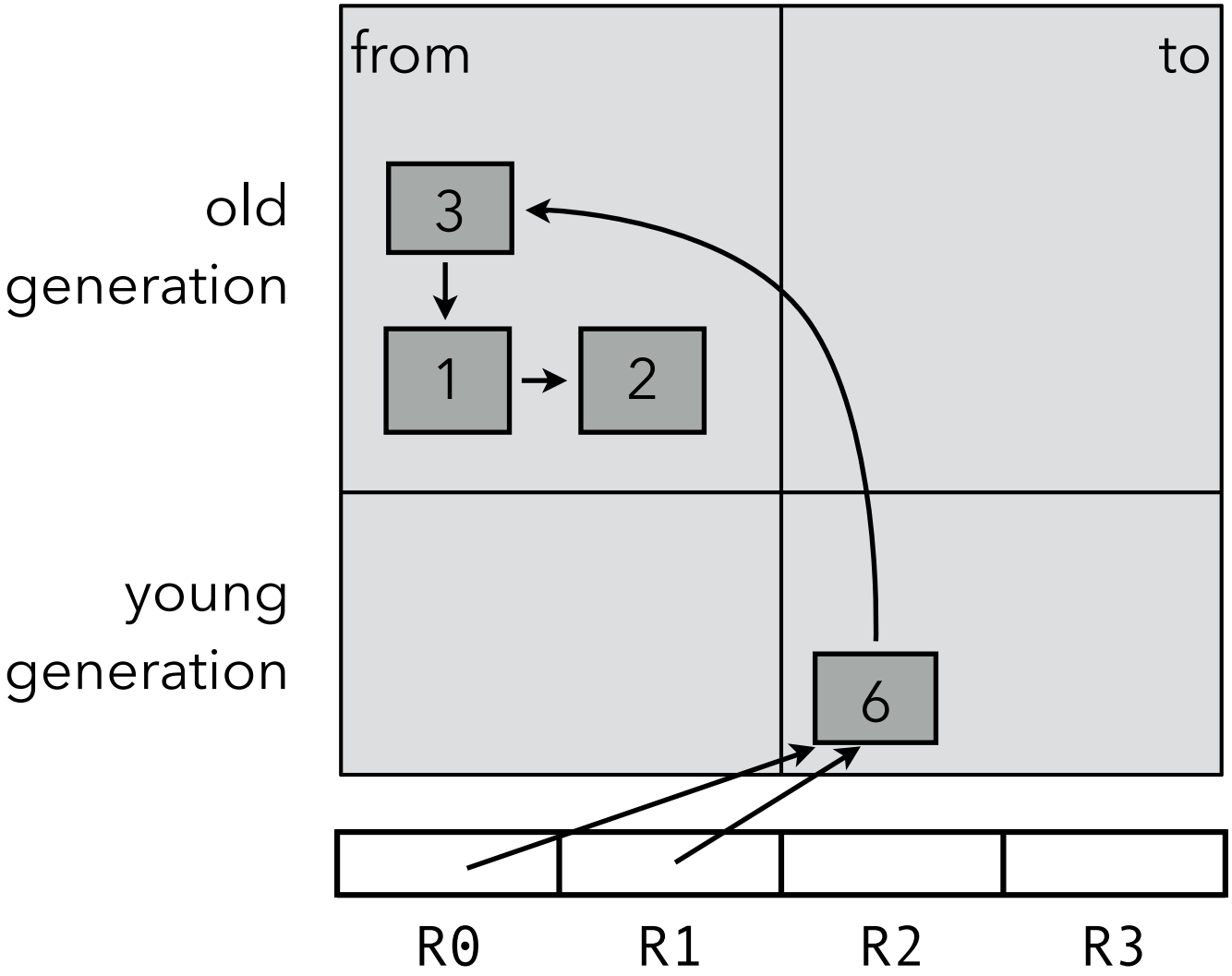
Object 3 is considered old enough to be promoted.

Minor collection example



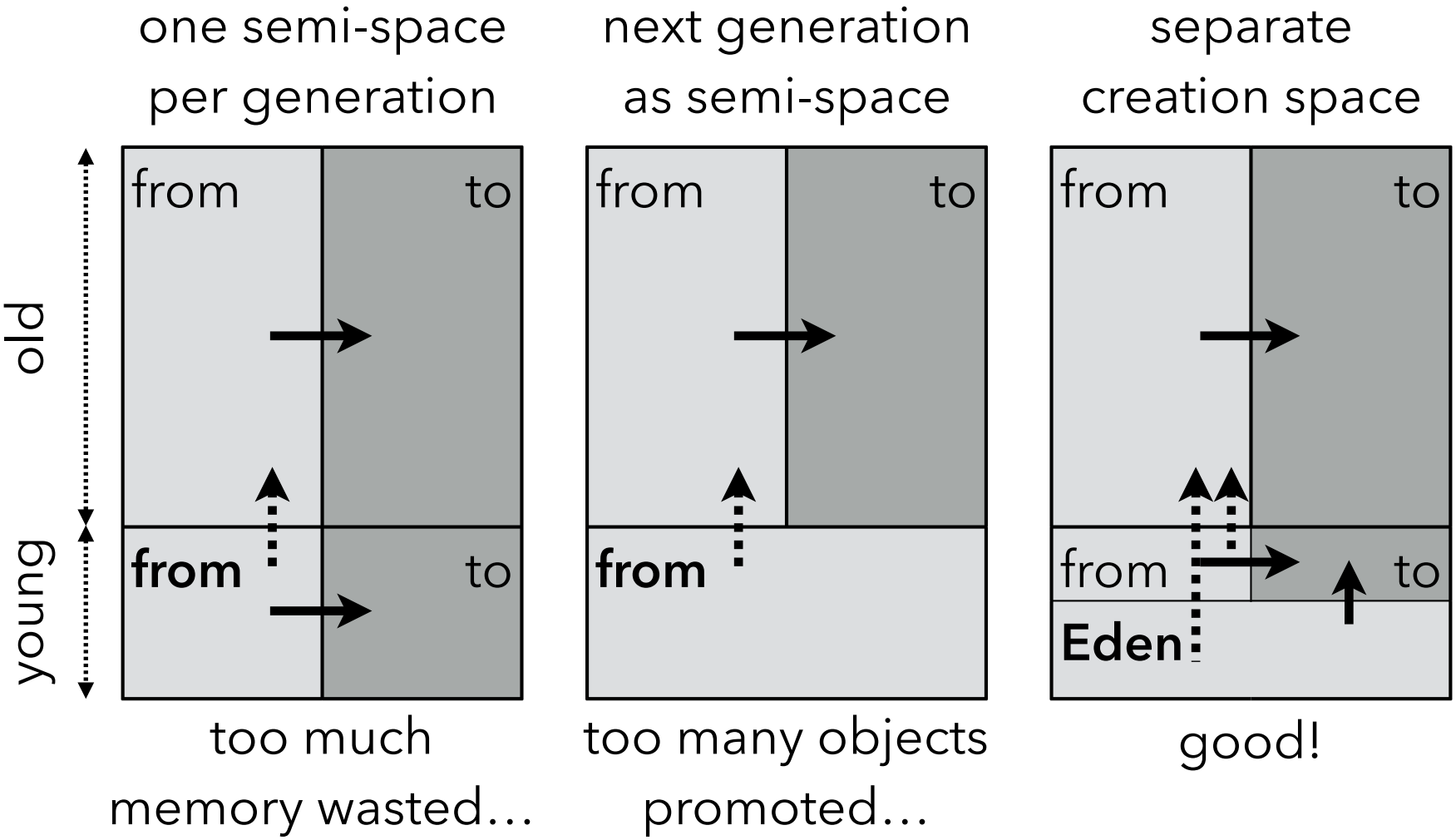
Object 3 is considered old enough to be promoted.

Minor collection example



Object 3 is considered old enough to be promoted.

Heap organization



Hybrid heap organization

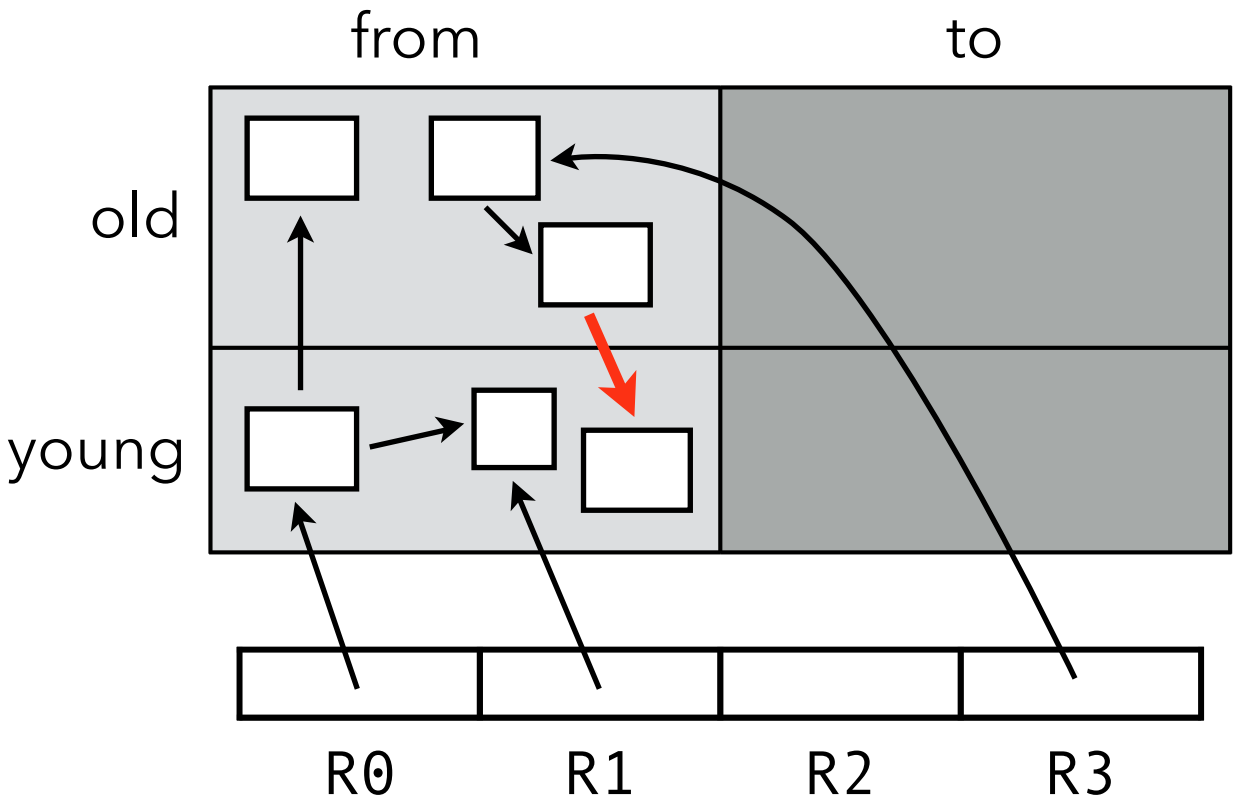
Instead of managing all generations using a copying algorithm, it is also possible to manage some of them – the oldest, typically – using a mark & sweep algorithm.

Promotion policies

Generational GCs use a **promotion policy** to decide when objects should be advanced to an older generation.
The simplest one – all survivors are advanced – can promote very young objects, but is simple as object age does not need to be recorded.
To avoid promoting very young objects it is sufficient to wait until they survive a second collection before advancing 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:

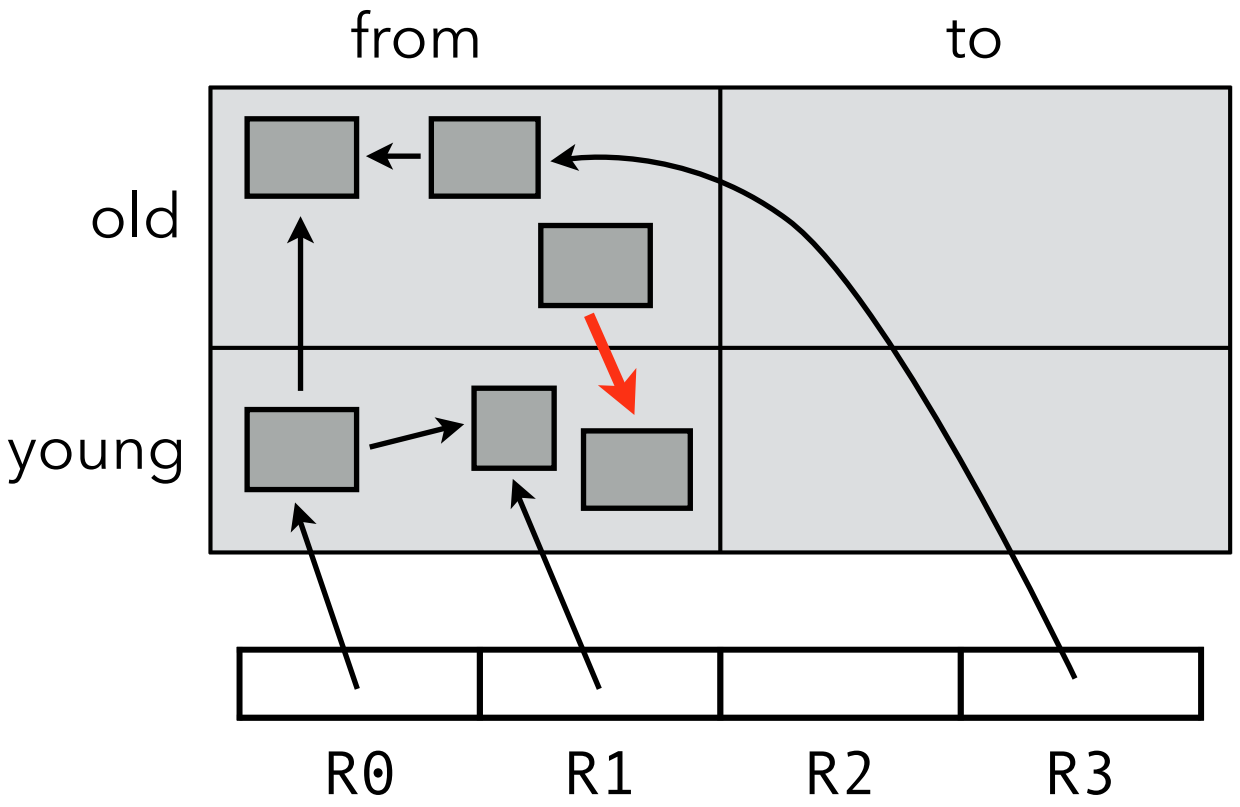
- the object into which the pointer is stored is not yet in the remembered set, and
- the pointer is stored in an old object, and points to a young one — although this can also be checked later by the collector.

Card marking

Card marking is another technique to detect inter-generational pointers.
Memory is divided into small, fixed sized areas called cards.
A card table remembers, for each card, whether it potentially contains inter-generational pointers.
On each pointer write, the card is marked in the table, and 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

Generational GC tends to reduce GC pause times since only the youngest generation – which is also the smallest – is collected most of the time.

In copying GCs, the use of generations also avoids copying long-lived objects over and over.

The only problems of generational GCs are the cost of maintaining the remembered set and nepotism.

Other kinds of garbage collectors

Incremental/concurrent GC

An **incremental garbage collector** can collect memory in small, incremental steps, thereby reducing the length of GC pauses – a very important characteristic for interactive applications.

Incremental GCs must be able to deal with modifications to the reachability graph made by the main program – called the **mutator** – while they attempt to compute it. This is usually achieved using a write barrier that ensures that the reachability graph observed by the GC is a valid approximation of the real one.

Several techniques, not covered here, exist to guarantee the validity of this approximation.

Parallel GC

Some parts of garbage collection can be sped up considerably by performing them in parallel on several processors. This is becoming important with the popularization of multi-core architectures.
For example, the marking phase of a mark & sweep GC can easily be done in parallel by several processors.
(Remember that parallelism and concurrency are separate and orthogonal concepts! A parallel GC does not have to be concurrent, and a concurrent GC does not have to be parallel.)

Virtual-memory-aware GC

The GCs presented until now are oblivious to the virtual memory manager. Unfortunately, this can lead them to perform badly when little physical memory is available: by traversing all live objects, even those residing on pages evicted to disk, they can incur considerable paging activity. **Bookmarking GC** is an example of a GC that avoids this problem. Its basic idea is to bookmark memory-resident objects that are referenced by evicted objects. These bookmarked objects are then considered reachable, and garbage collection is performed without looking at – and therefore loading – evicted objects.

Additional GC features

Finalizers

Some GCs make it possible to associate **finalizers** with objects.

Finalizers are functions that are called when an object is about to be collected. They 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.

Finalizers 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?

Flavors of references

When the GC encounters a reference, it usually treats it as a strong reference, meaning that the referenced object is considered as reachable and survives the collection.

It is sometimes useful to have weaker kinds of refrences, which can refer to an object without preventing it from being collected.

Weak references

The term **weak reference** (or **weak pointer**) designates a reference that does not prevent an object from being collected.

During a GC, if an object is only weakly reachable, it is collected, and all (weak) references referencing it are cleared.

Weak references are useful to implement caches, “canonicalizing” mappings, etc.