Memory management

Michel Schinz – based on Erik Stenman's slides
Advanced compiler construction, 2008-03-07

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.

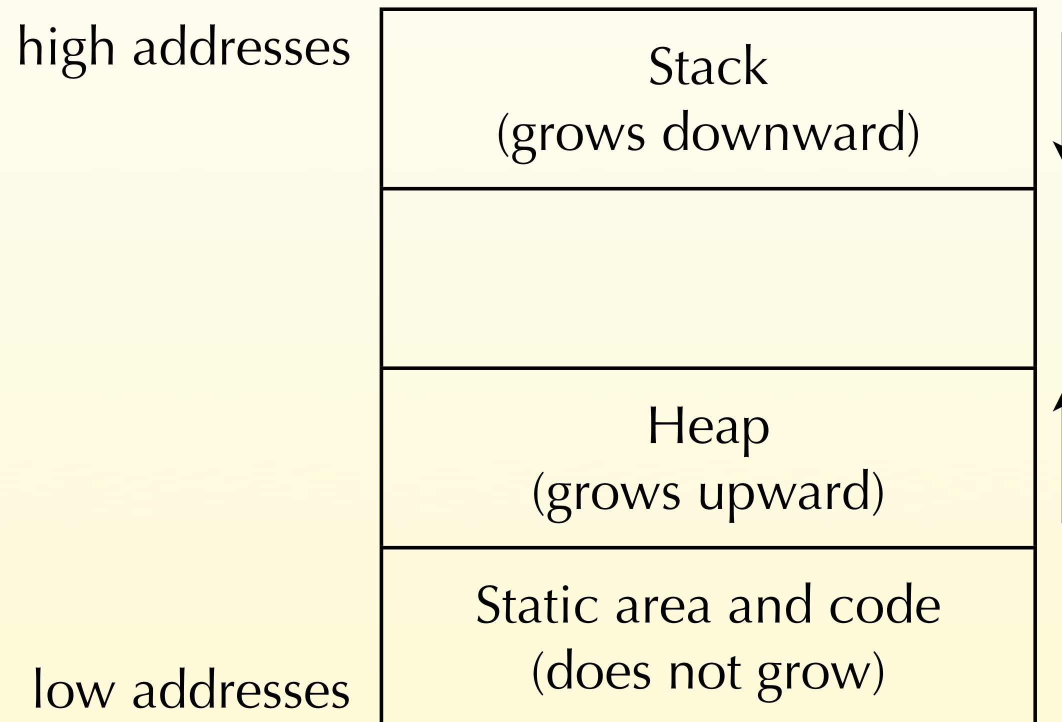
Memory areas

Every piece of memory used by a program is allocated from one of three different areas:

- A **static** area, which is laid out at compilation time and allocated when the program starts. The static area is used to store global variables and constants.
- A **stack**, from which memory is allocated and freed dynamically, in LIFO order. The stack is used to store the arguments and local variables of functions, since in most languages function calls happen in LIFO order.
- A **heap**, from which memory is allocated and freed dynamically, in any order. The heap is used to store objects that outlives the function that created them.

Memory organisation

The three areas just described can be organised as follows in the address space of a running program:



The memory manager

The memory manager

Managing the static area and the stack is trivial.

Managing the heap is much more difficult because of the irregular lifetimes of the blocks it contains. The **memory manager** is the part of the run time system in charge of managing heap memory.

Its job consists in answering to two kinds of requests:

1. allocation requests, which consist in finding a free block of memory big enough to satisfy the request, remove it from the set of free blocks, and return it to the program,
2. deallocation requests, which consist in returning a previously-allocated block to the set of free blocks, to make it available for further allocation requests.

Free list

The memory manager must keep track of which parts of the heap are free, and which are allocated.

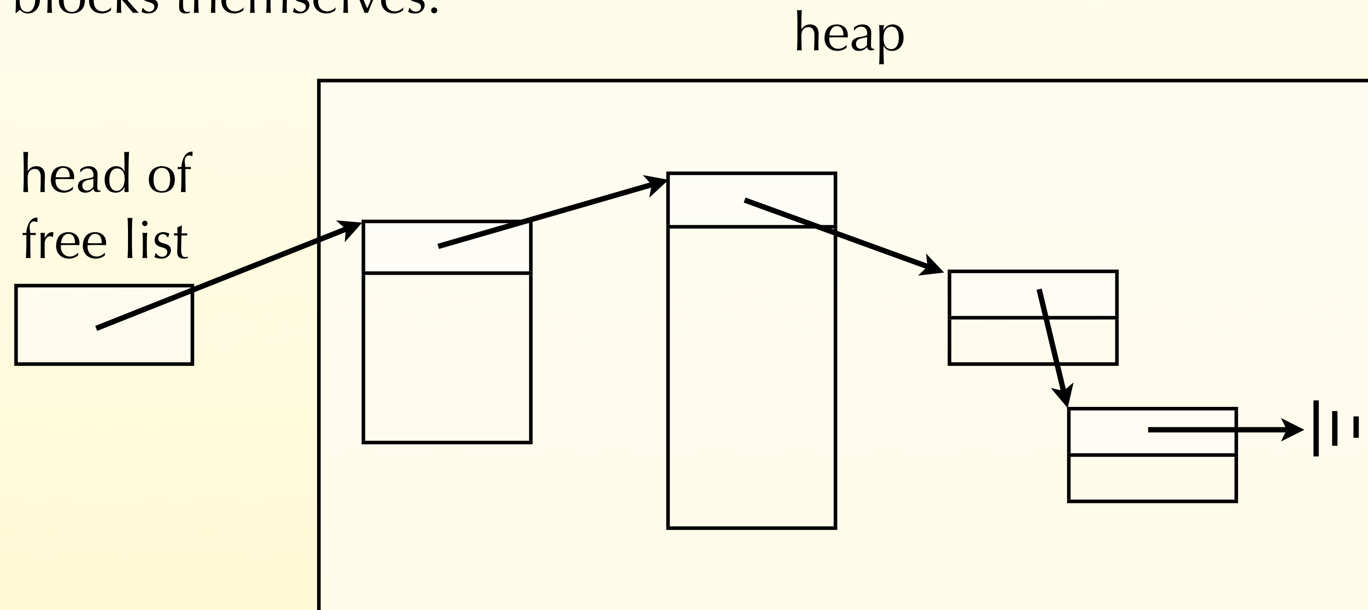
For that purpose, free memory blocks are stored in a data-structure called the **free list**. Notice that the term *free list* is used even when the data-structure used to track free memory is not a list.

There is no need to keep a list of allocated blocks, as it can be computed using the free list – all blocks that are not in the free list are allocated.

Free list storage

Since the blocks stored in the free list are by definition not used by the program, the memory manager can store information in them!

For example, if the free list is represented as a singly linked list, then the pointer to the next block can be stored in the blocks themselves:

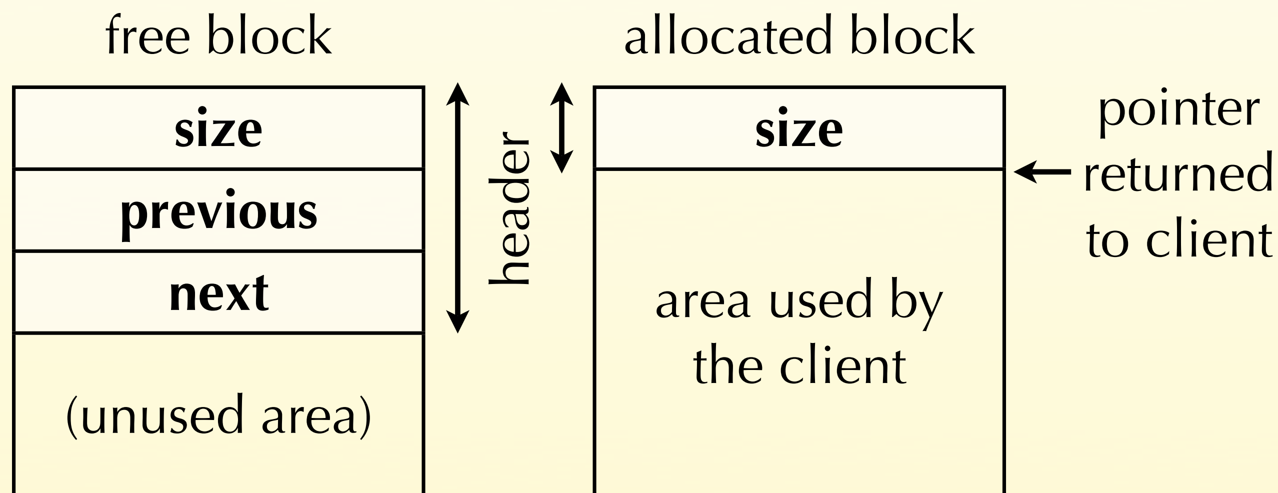


Block header

Apart from the link to their successor and/or to their predecessor, free blocks must contain their size.

Allocated blocks do not require links to other blocks, but must also contain their size.

This information is stored in the block's **header**, situated just before the area used by the client, and invisible to it.



Splitting and coalescing

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.

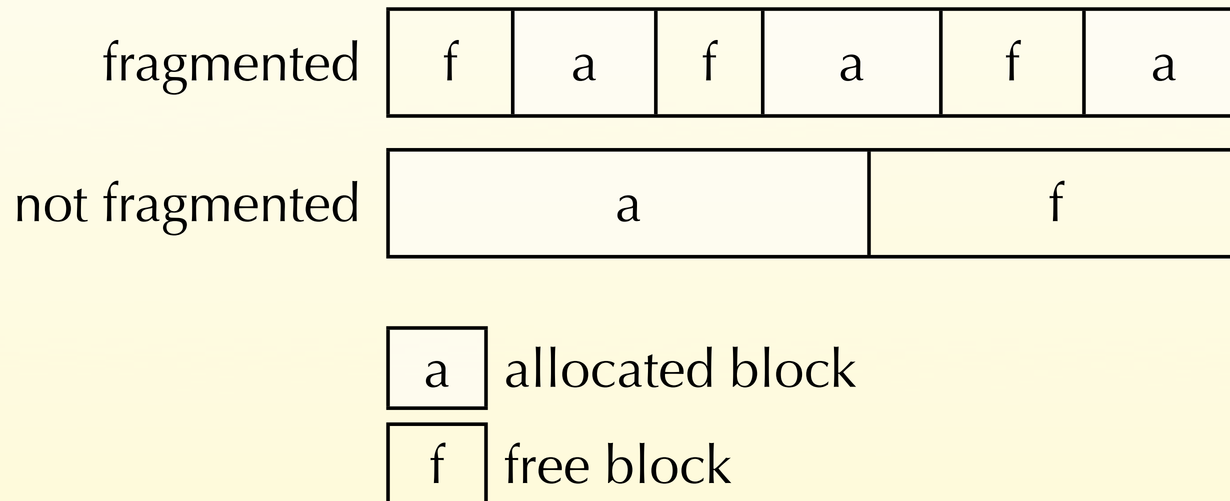
Fragmentation

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

1. **external fragmentation** refers to the fragmentation of free memory in many small blocks,
2. **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

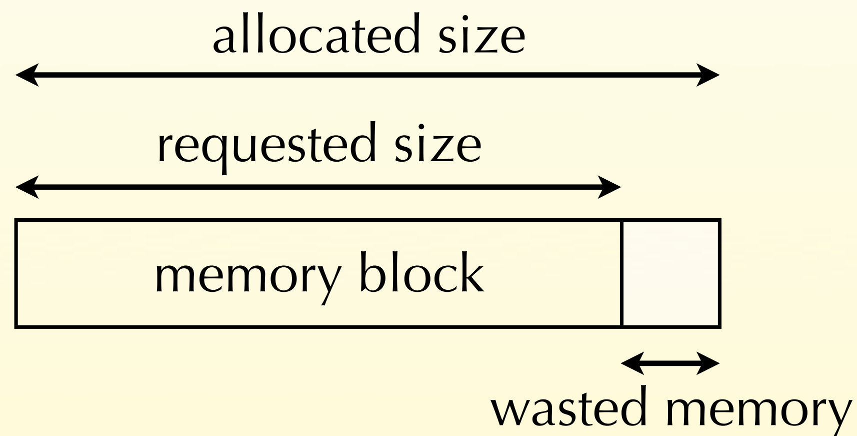
The following two heaps have the same amount of free memory, but the first suffers from **external fragmentation** while the second does not. As a consequence, some requests can be fulfilled by the second but not by the first.



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**.



Memory allocation

Allocation policies

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 minimise fragmentation while being fast to implement.

There are several such policies: first fit, next fit, best fit, worst fit, etc.

First fit, next fit

First fit chooses the first block in the free list big enough to satisfy the request, and splits it if necessary.

Next fit is like first fit, except that the search for a fitting block starts where the last one ended, instead of at the beginning of the free list.

It appears that next fit results in significantly more fragmentation than first fit, as it mixes blocks allocated at very different times.

Best fit, worst fit

Best fit chooses the smallest block big enough to satisfy the request.

Worst fit chooses the biggest, with the aim of avoiding the creation of too many small fragments. It doesn’t work well in practice.

The major problem of these techniques is that they require an exhaustive search of the free list, unless **segregation techniques** are used.

Segregated free lists

Instead of having a single free list, it is possible to have several of them, each holding free blocks of (approximately) the same size.

These **segregated free lists** are organised in an array, to quickly find the appropriate free list given a block size.

When a given free list is empty, bigger blocks taken from adjacent lists are split in order to repopulate it.

Buddy systems

Buddy systems are a variant of segregated free lists.

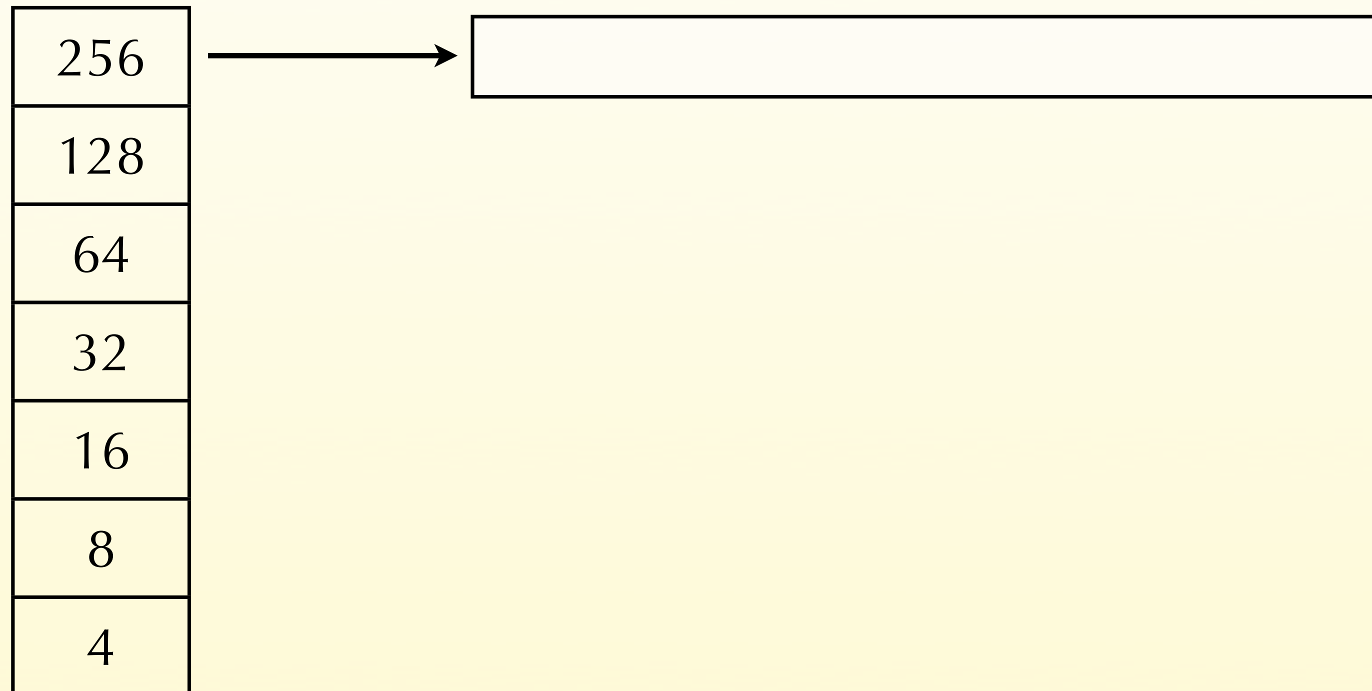
The heap is initially viewed as one large block that can be split in two smaller blocks – called buddies – of a given size. Those smaller blocks can again be split in two smaller buddies, and so on.

In a **binary buddy system**, a block is split in two buddies of the same size. In a **Fibonacci buddy system**, a block is split in two buddies whose size is given by a Fibonacci sequence ($s_n = s_{n-1} + s_{n-2}$).

Coalescing is fast in buddy systems, since a block can only be coalesced with its buddy, provided it is free too.

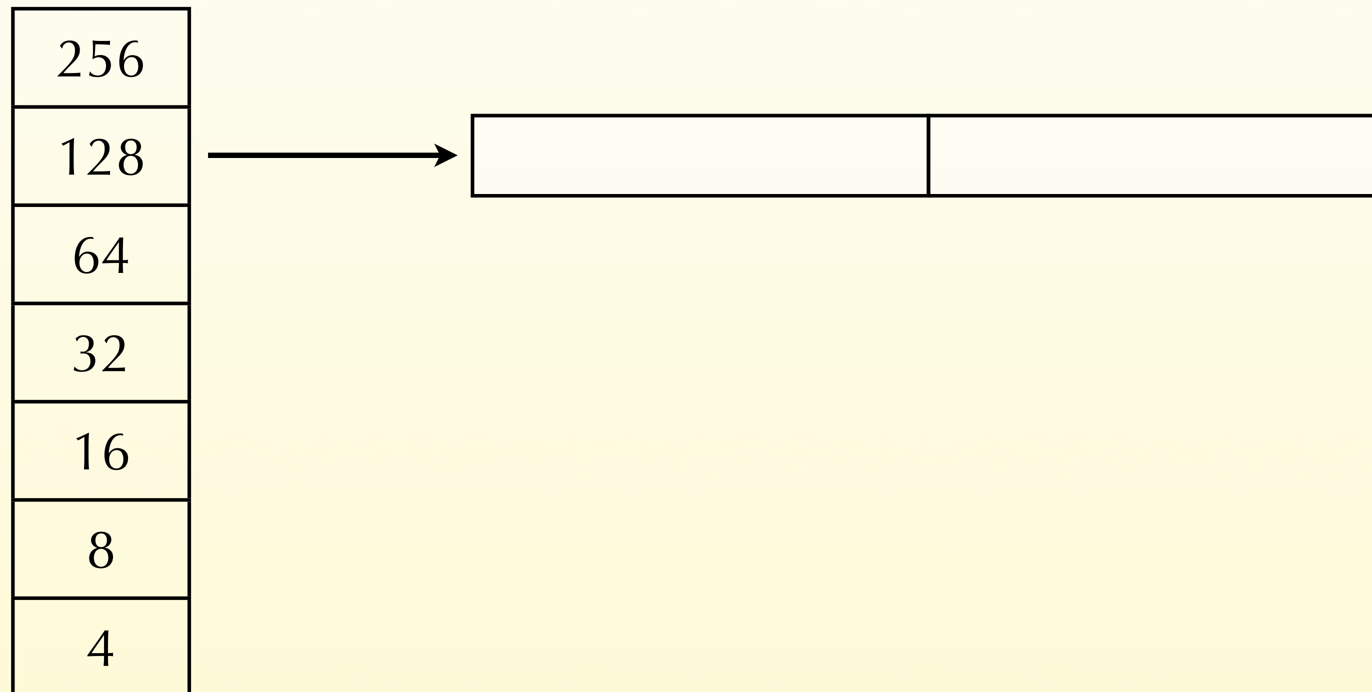
Allocation in a buddy system

This example illustrates how a 10 bytes block is allocated in a binary buddy system with a heap of 256 bytes, initially free.



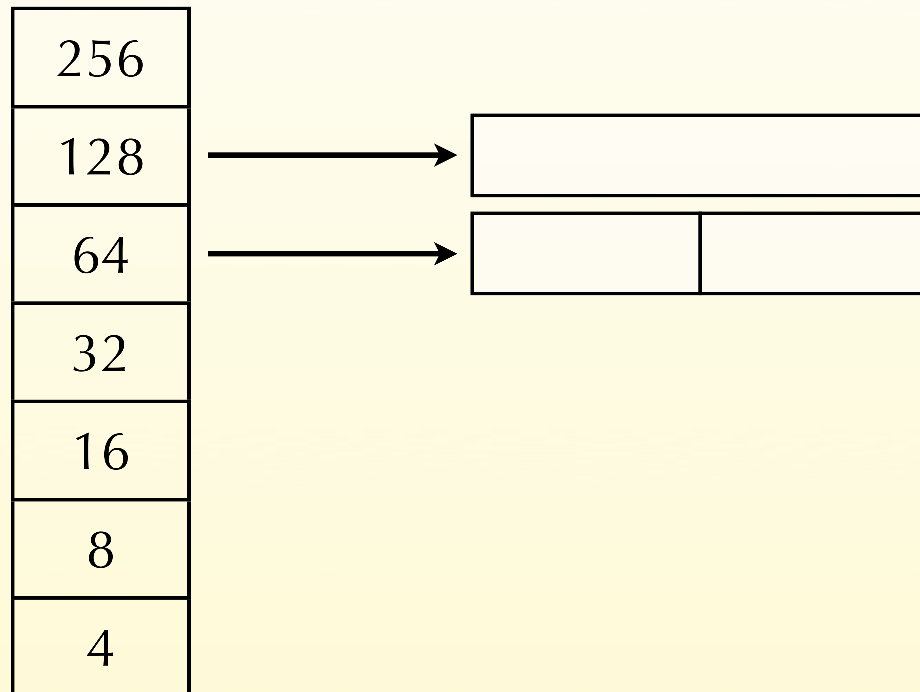
Allocation in a buddy system

This example illustrates how a 10 bytes block is allocated in a binary buddy system with a heap of 256 bytes, initially free.



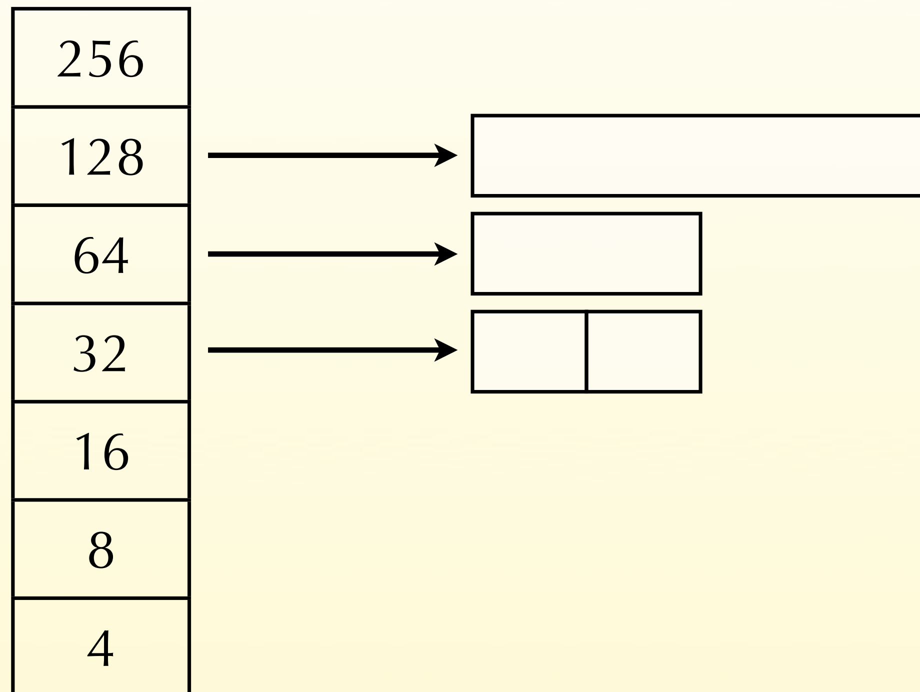
Allocation in a buddy system

This example illustrates how a 10 bytes block is allocated in a binary buddy system with a heap of 256 bytes, initially free.



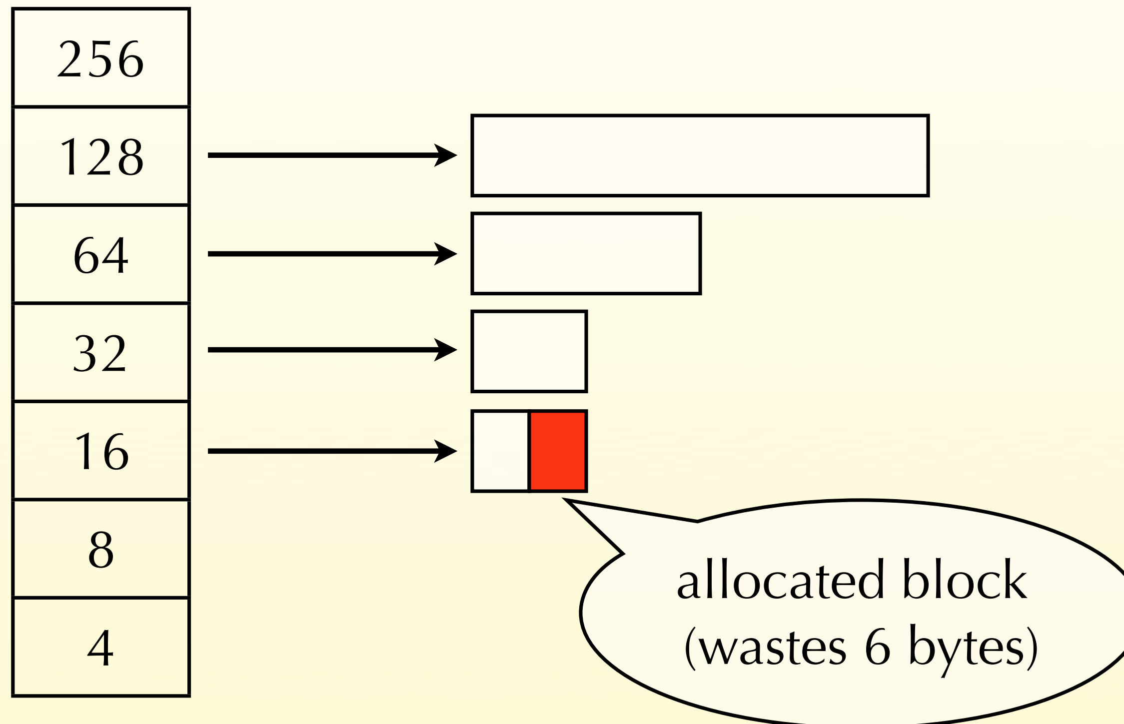
Allocation in a buddy system

This example illustrates how a 10 bytes block is allocated in a binary buddy system with a heap of 256 bytes, initially free.



Allocation in a buddy system

This example illustrates how a 10 bytes block is allocated in a binary buddy system with a heap of 256 bytes, initially free.



Memory deallocation

Memory deallocation

In a programming language, deallocation of heap memory can be either **explicit** or **implicit**.

It is explicit when the language offers a way to declare a memory block as being free – *e.g.* using `delete` in C++ or `free()` in C.

It is implicit when the run time system infers that information itself, usually by finding which allocated blocks are not reachable anymore.

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 recent 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 space 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 prevents dangling pointers.

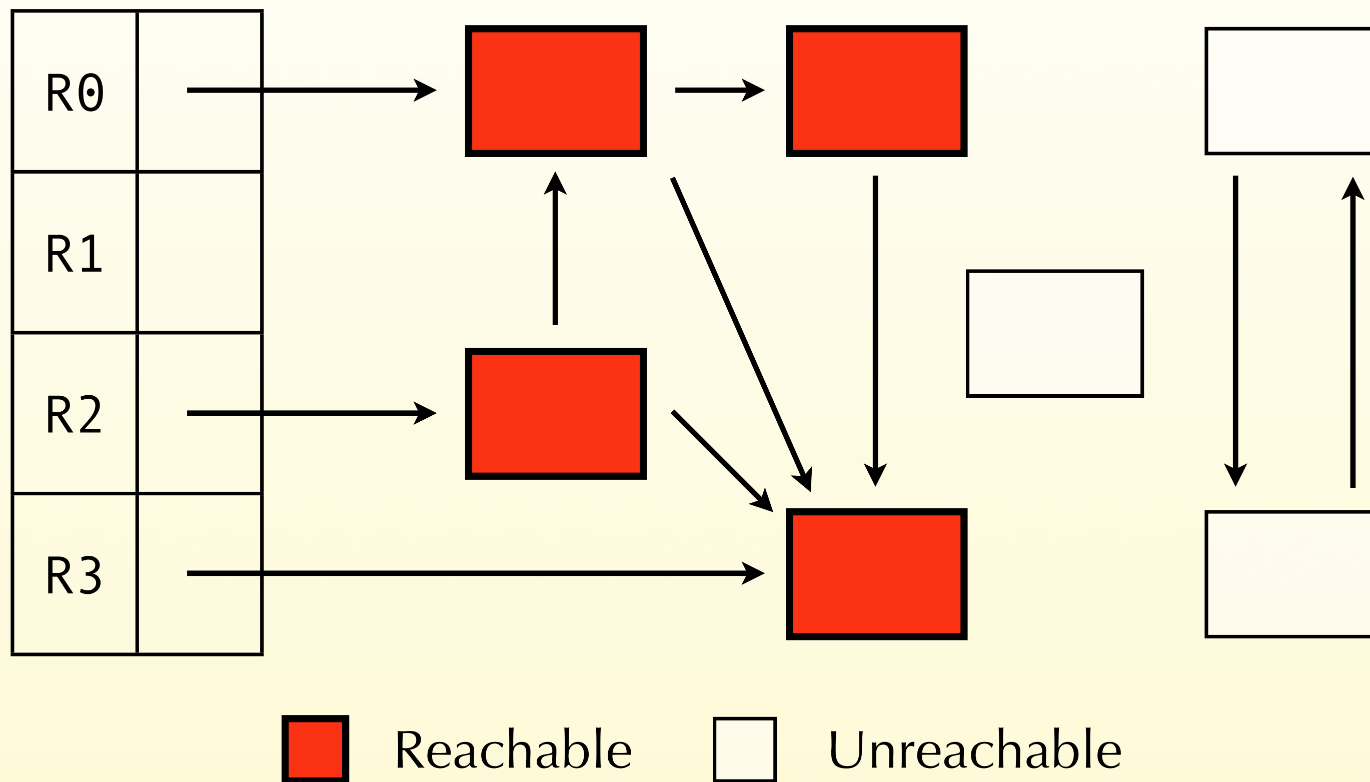
Reachable objects

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**,
- the objects reachable from other reachable objects, by following pointers.

Those objects form the **reachability graph**.

Reachability graph example



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.

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 – or the 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.

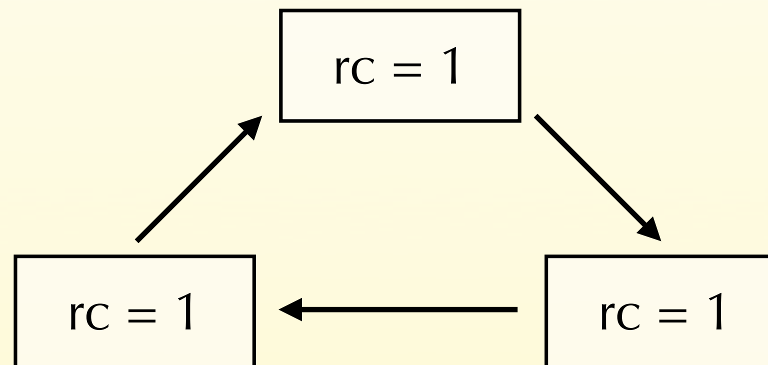
However, it has an important impact on space consumption, and speed of execution: every object must contain a counter, and every pointer write must update it.

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:

$$\text{reference_count}(x) = 0 \Rightarrow x \text{ is unreachable}$$

but the opposite is not true!

Uses of reference counting

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

Mark & sweep garbage collection

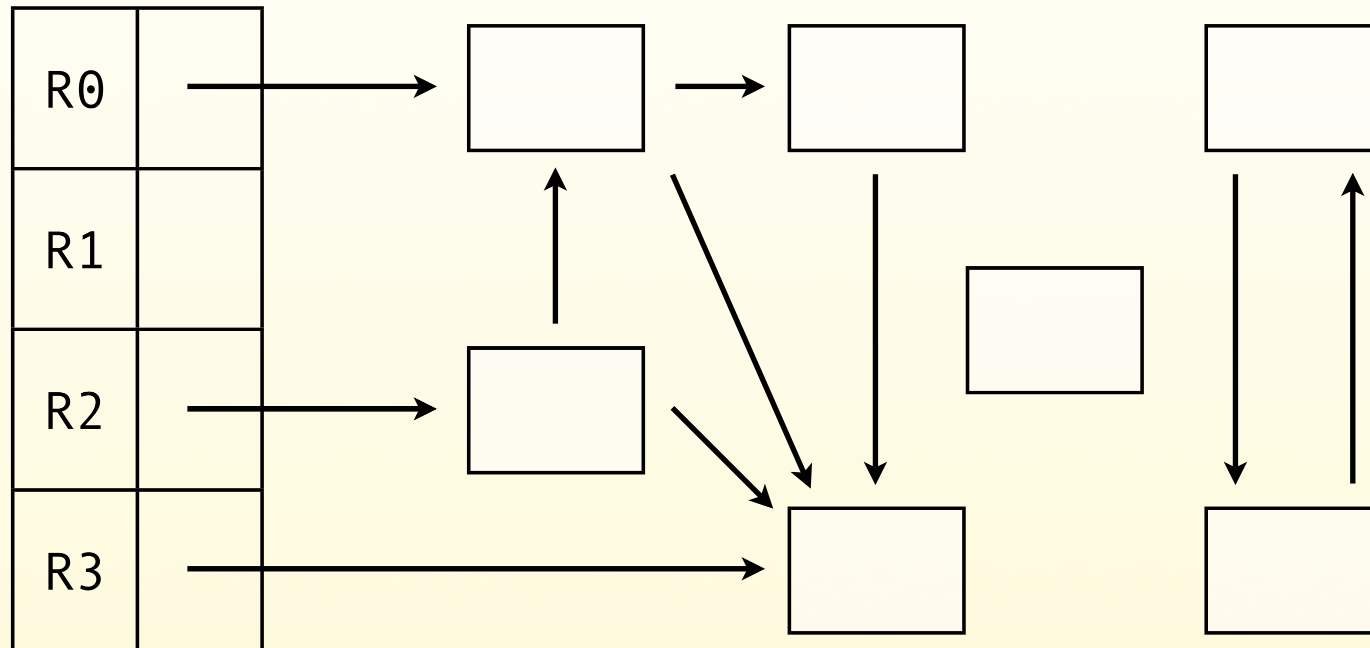
Mark & sweep GC

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

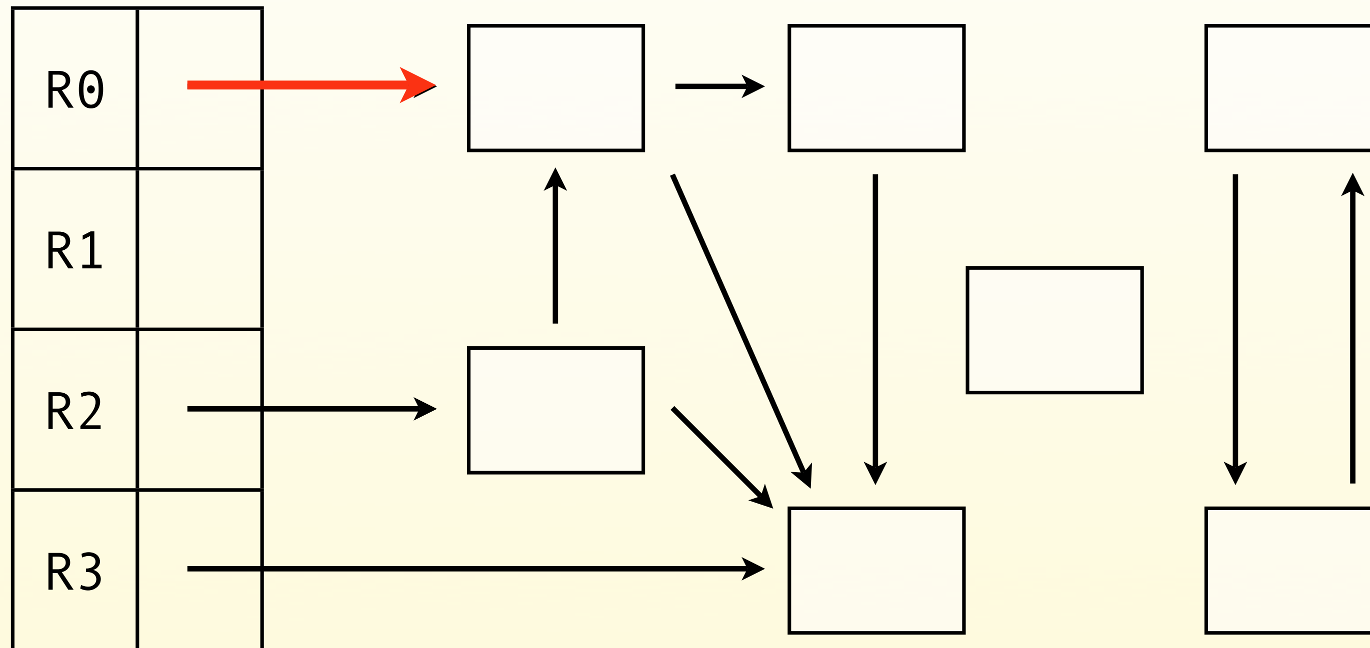
1. in the **marking phase**, the reachability graph is traversed and reachable objects are marked,
2. in the **sweeping phase**, all allocated objects are examined, and unmarked ones are freed.

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.

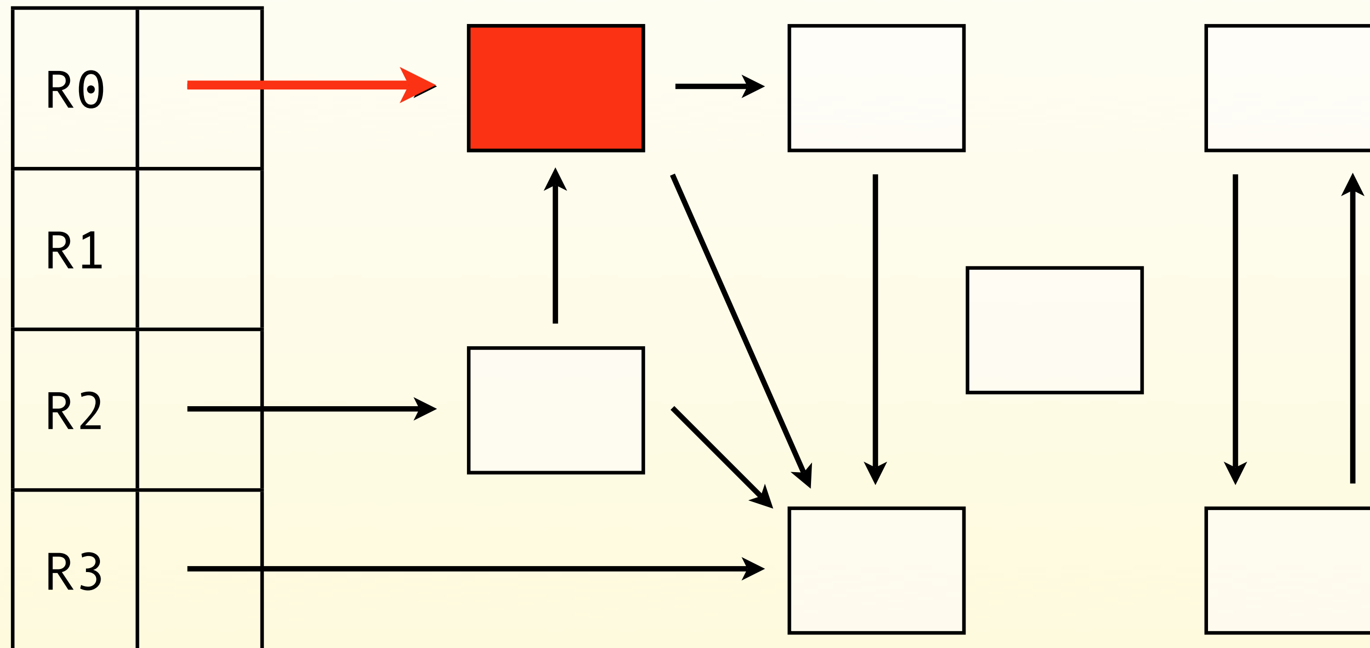
Mark & sweep GC



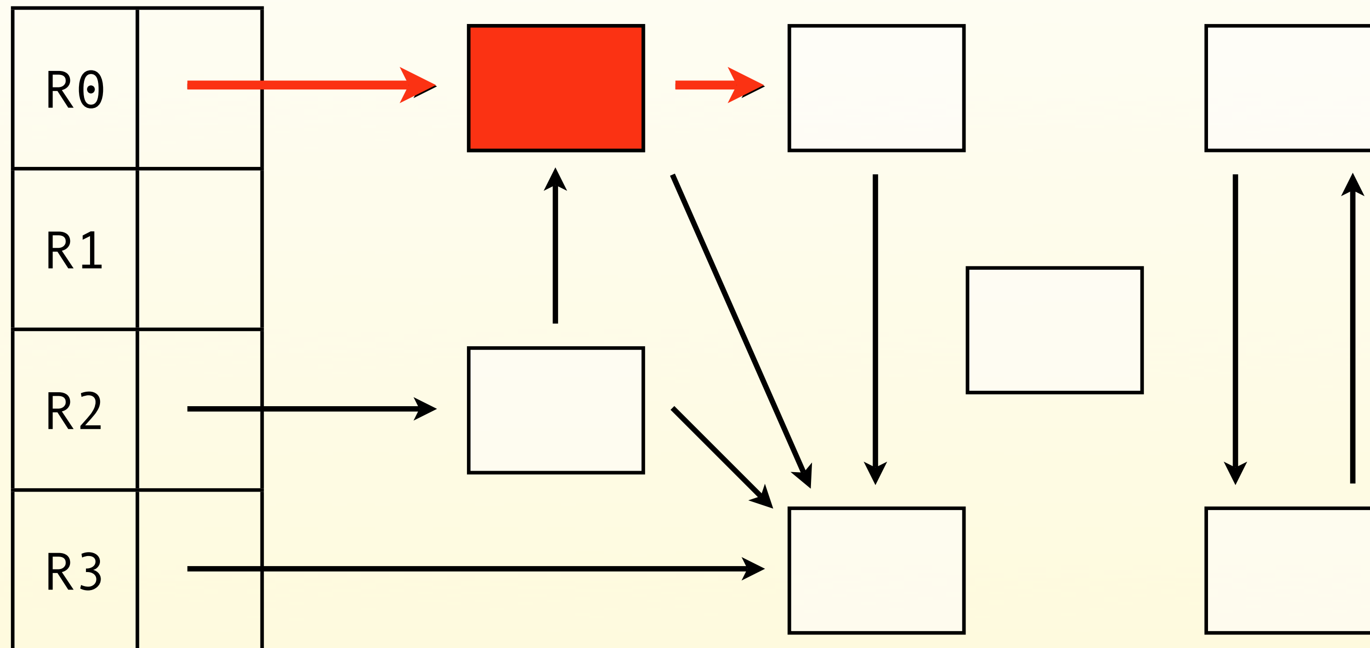
Mark & sweep GC



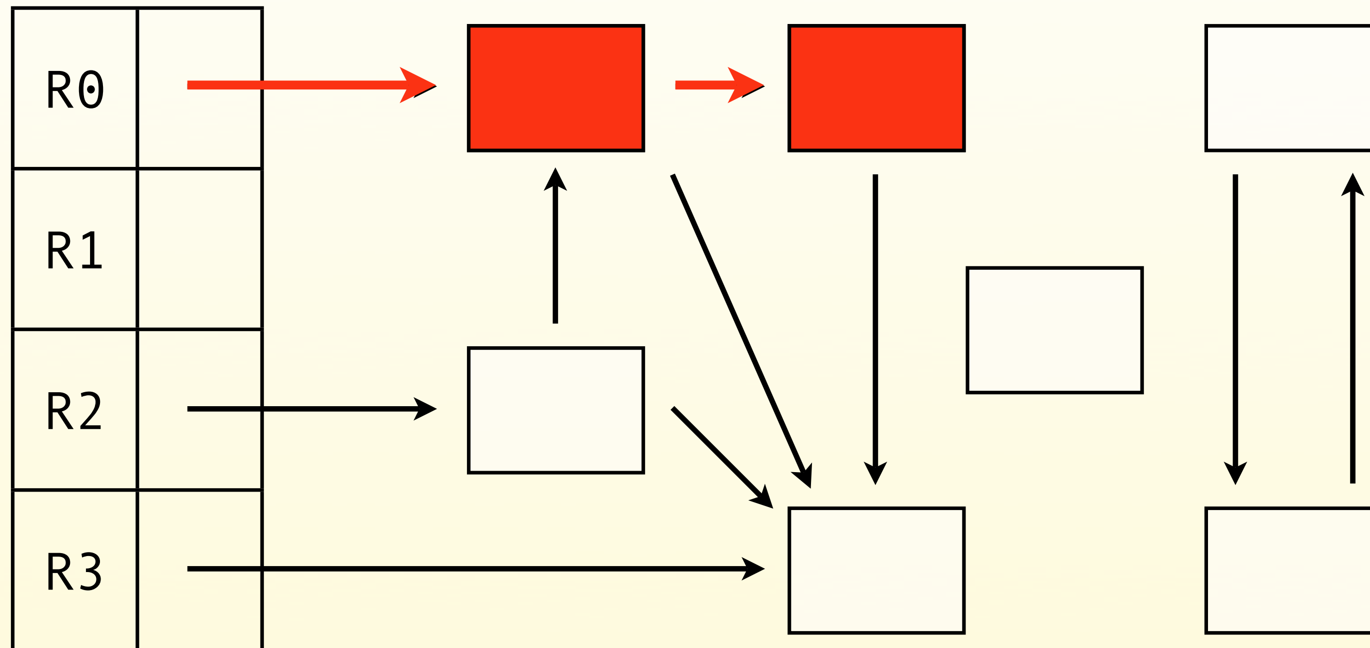
Mark & sweep GC



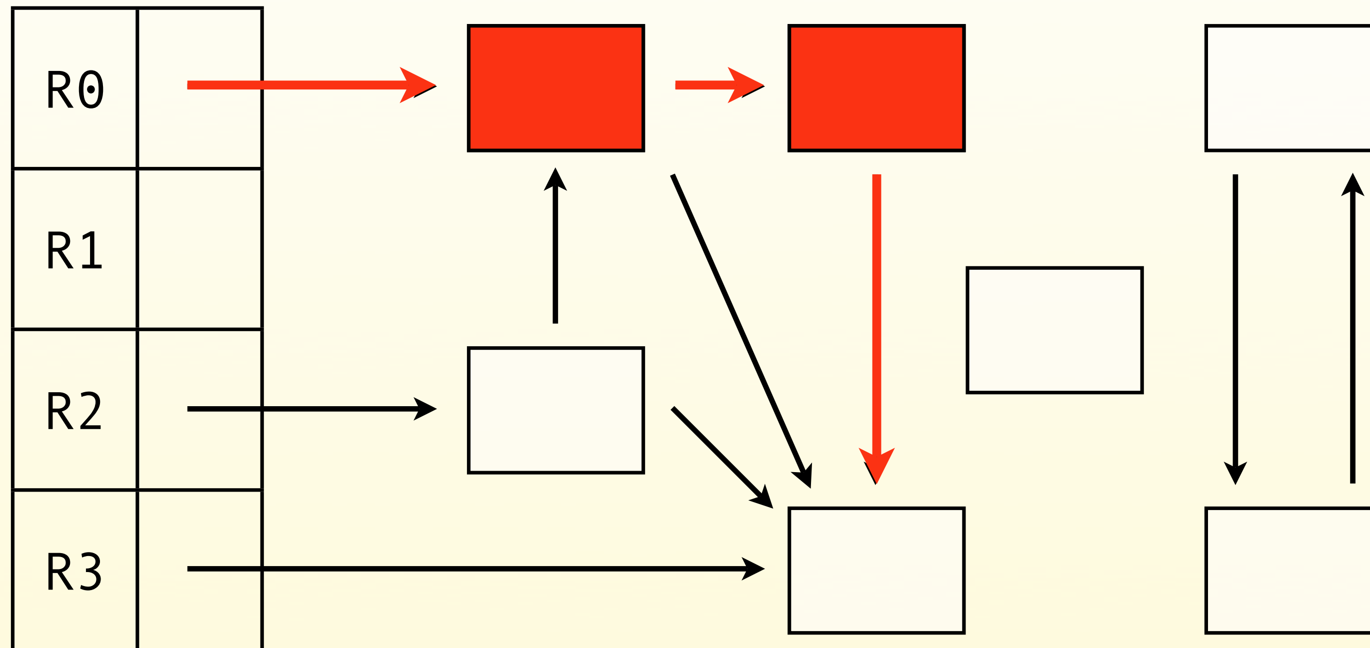
Mark & sweep GC



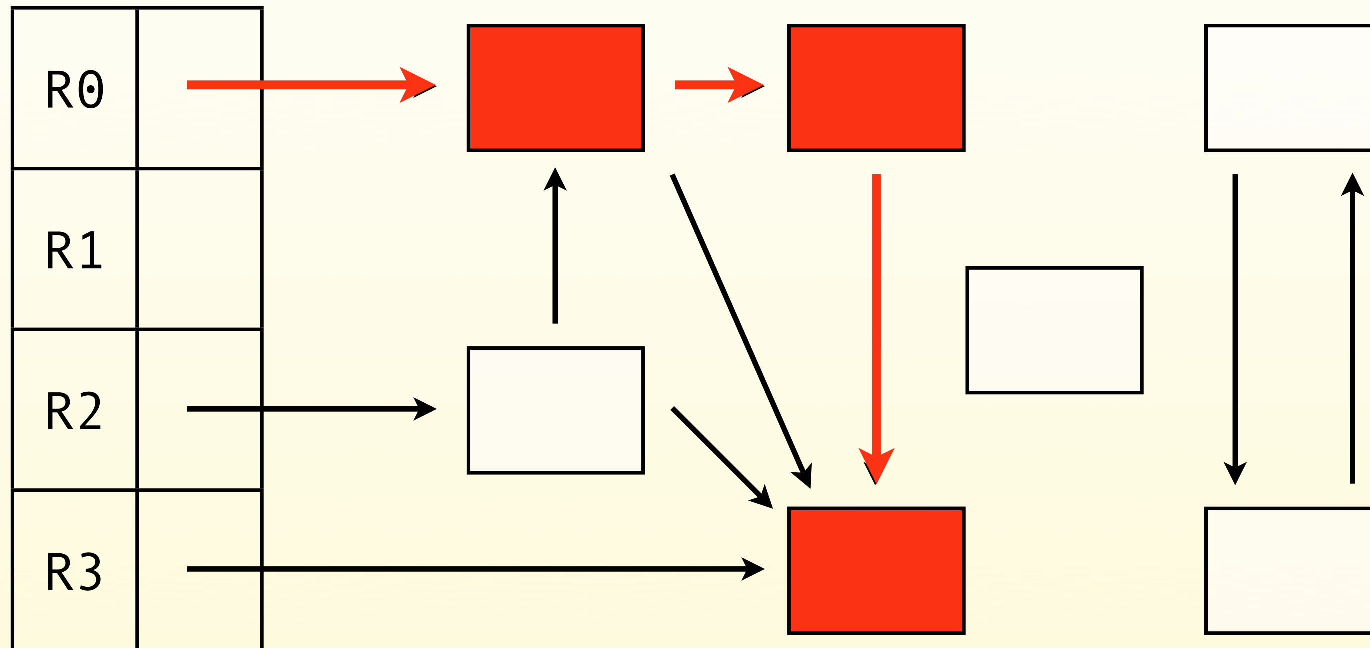
Mark & sweep GC



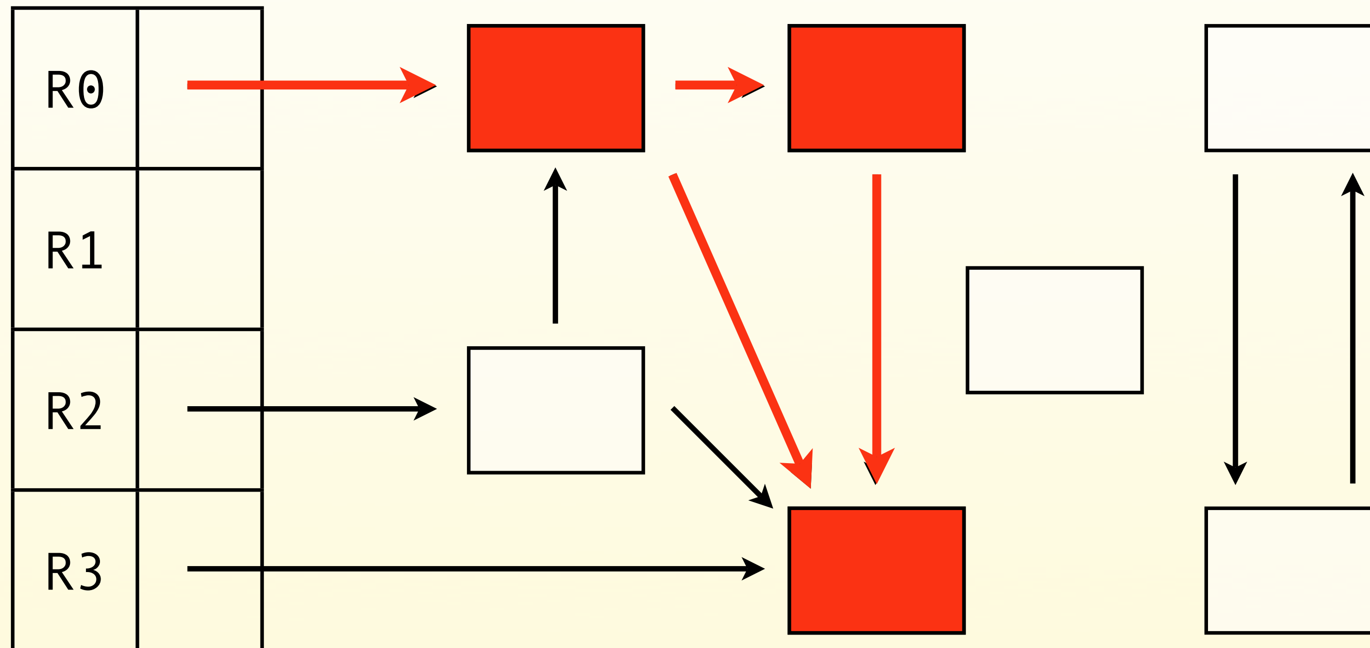
Mark & sweep GC



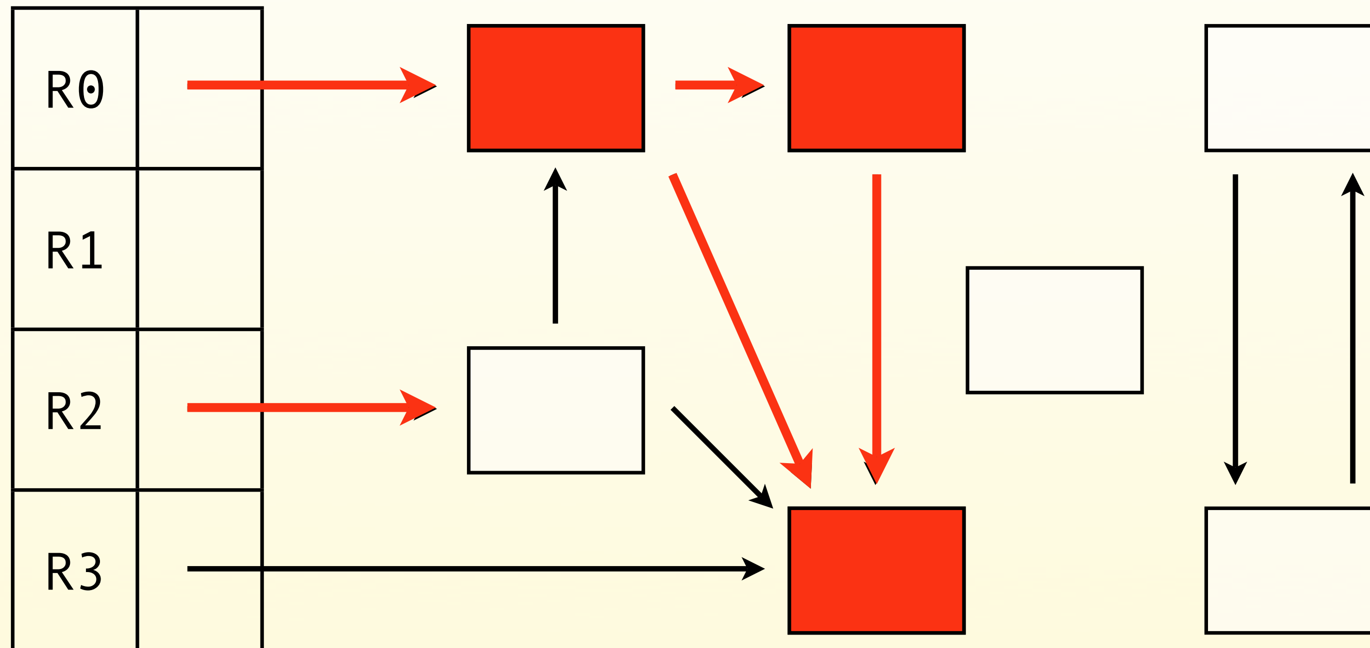
Mark & sweep GC



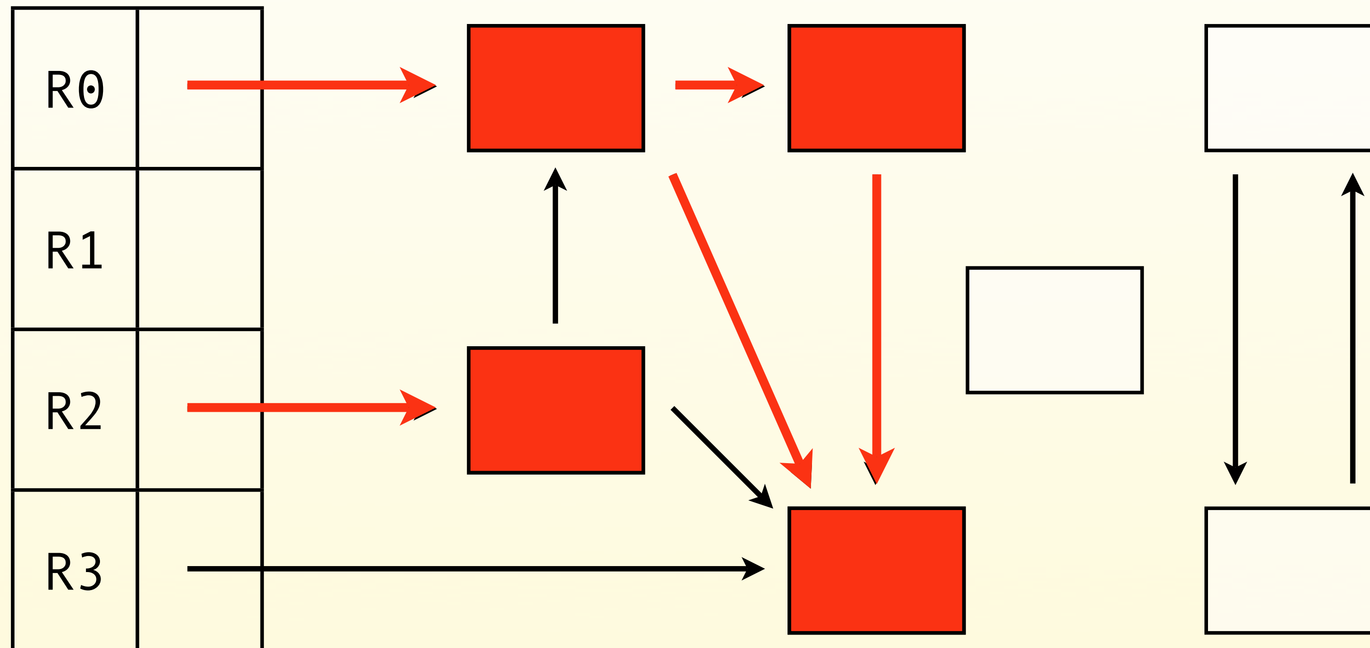
Mark & sweep GC



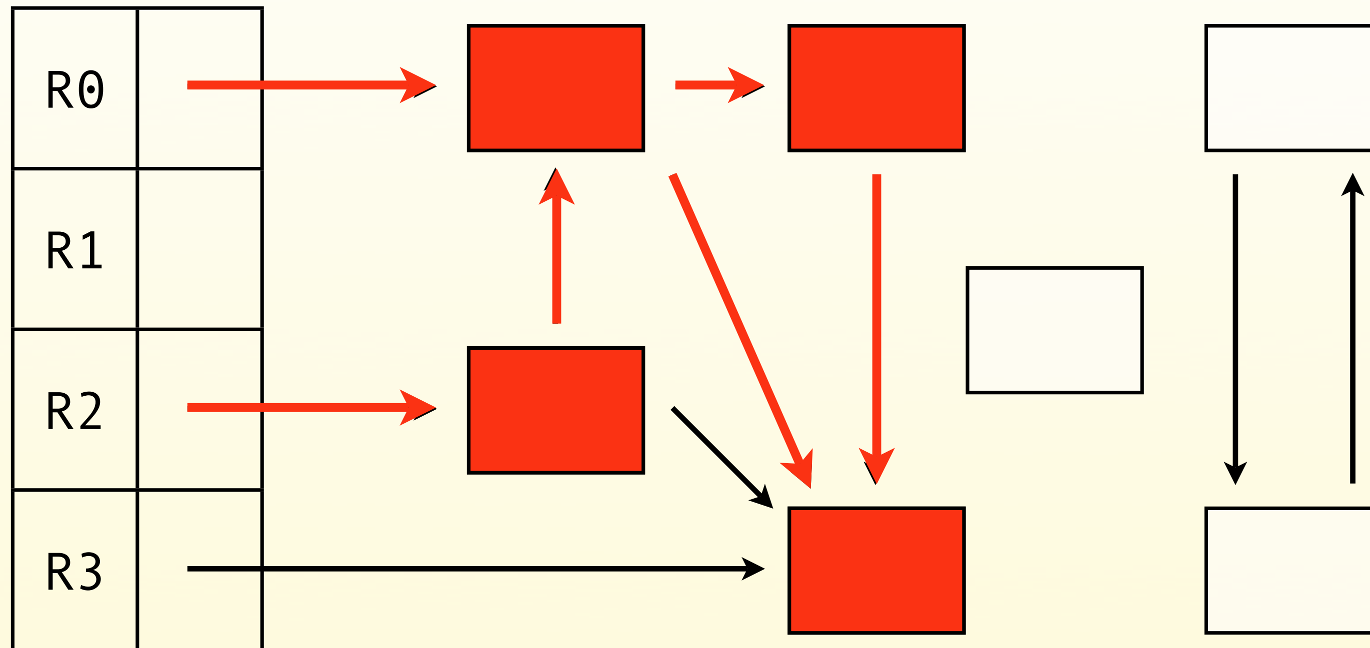
Mark & sweep GC



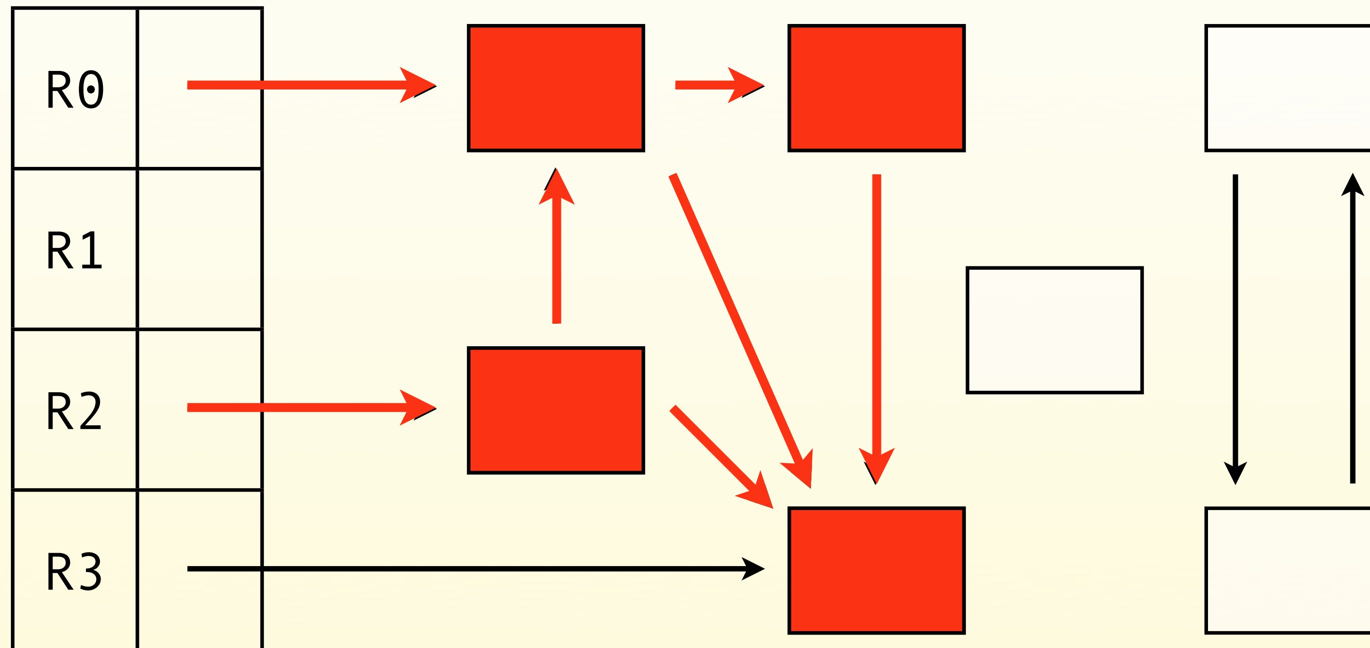
Mark & sweep GC



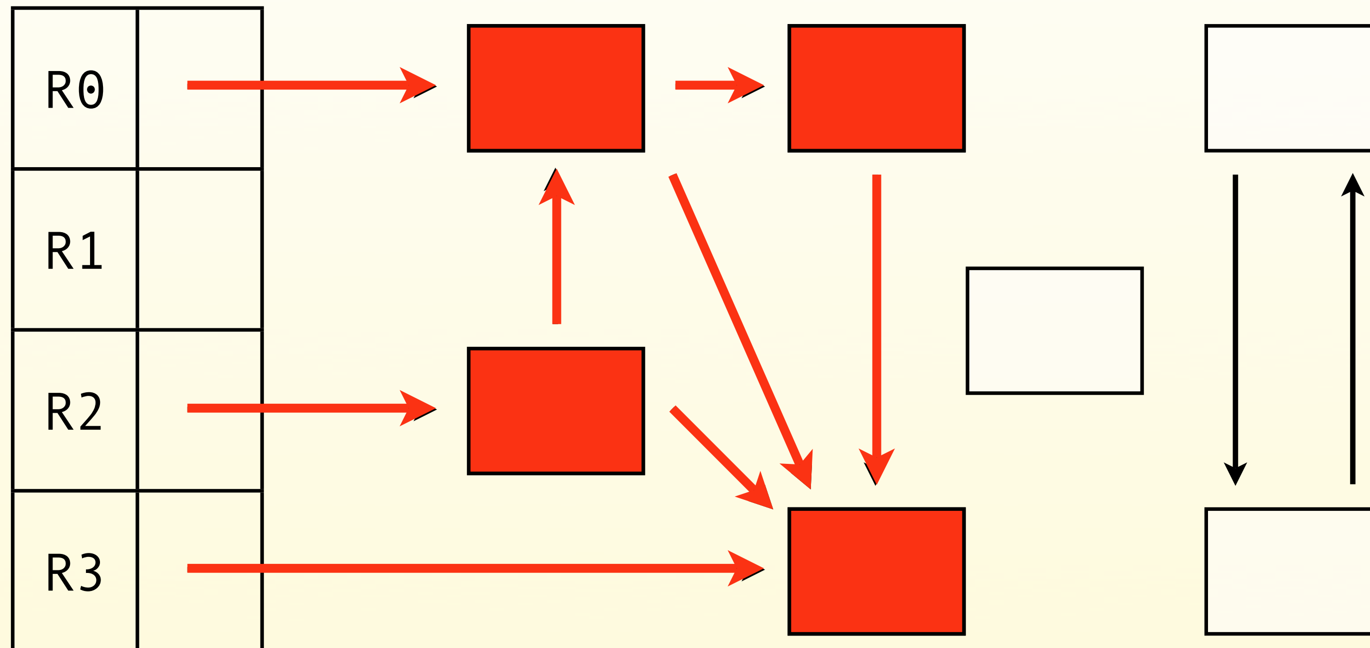
Mark & sweep GC



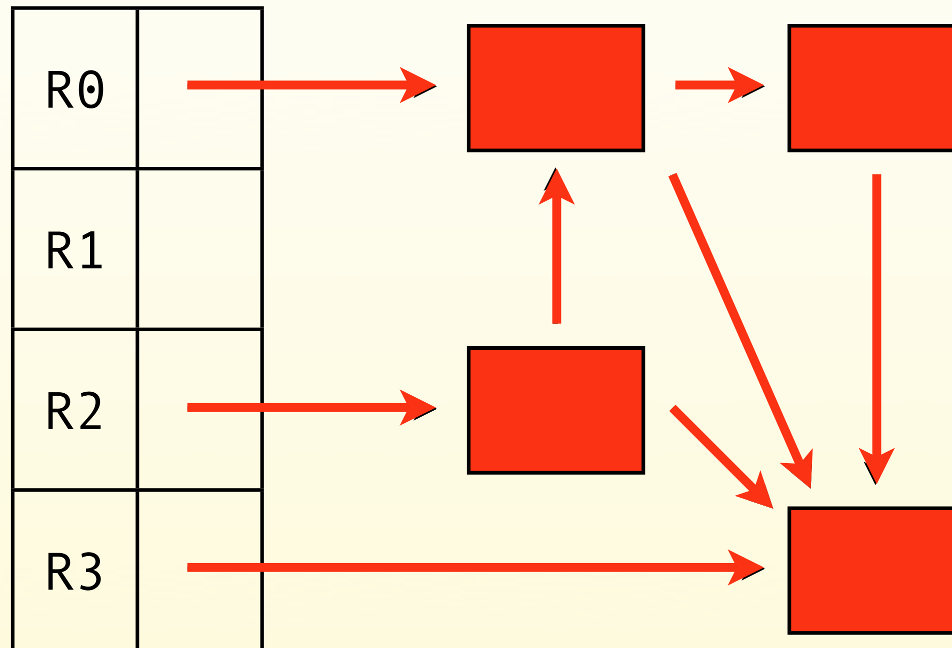
Mark & sweep GC



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.

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 fulfil the current memory need. This is called **lazy sweep**.

Data representation

Data representation

Until now, we have assumed that the garbage collector is able to traverse the object graph at run time. However, we have not explained how it can do that, and in particular how it is able to distinguish pointers from other data.

This ability depends on how data is represented in memory, which itself depends on the features of the language being compiled.

We will quickly examine several techniques to represent data in memory, as well as their impact on the design of the garbage collector.

Uniform data representation

In dynamically typed languages – e.g. Lisp, Scheme, Python, Ruby, etc. – nothing is known at compilation time about the type of data that the program will manipulate at run time. For that reason, all data has to be represented in a uniform way.

Uniformity is often obtained by representing every value as a pointer to a heap-allocated object containing the actual data, as well as a header giving information about the type of the data.

Even small values like integers or floating-point numbers are heap allocated – they are said to be **boxed**.

Uniform data representation

When data is represented uniformly, any object in the heap can be either:

- an **atom**, that is a basic value – integer, floating-point number, character, etc. – containing no pointers to further data, or
- a compound object, consisting only of pointers to other objects.

The information about whether an object is an atom or a compound object can be given by a single bit in its header.

Traversing the object graph with a uniform data representation is trivial: atoms are known to contain no pointers, while compound objects are known to contain only pointers, all of which must be followed.

Tagging

Representing all values as heap-allocated objects has a cost that is especially high for small objects like integers.

Tagging is a technique that can be used to avoid boxing integers or other kinds of small data. It takes advantage of the fact that, on most architectures, the least significant bit (LSB) of all pointers is zero. Therefore, if the integer n is represented by the value $2n+1$, then it is possible to distinguish pointers from integers just by looking at the LSB! Of course, arithmetic operations on tagged integers have to be adapted to take tagging into account.

The only problem of tagging is that it halves the range of integers, which can sometimes be problematic.

Specialised data representation

In statically typed, monomorphic languages, the compiler knows the type of all data that the program will manipulate at run time. Therefore, it doesn't need to represent all data uniformly, but can use the natural representation for every type of data.

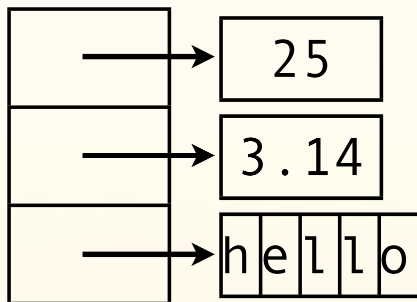
In such a situation, integers and pointers are typically represented by values that are indistinguishable at run time, differing only in the way they are used.

For that reason, traversing the object graph at run time requires help from the compiler, which must include enough information in object headers to make the identification of pointers possible.

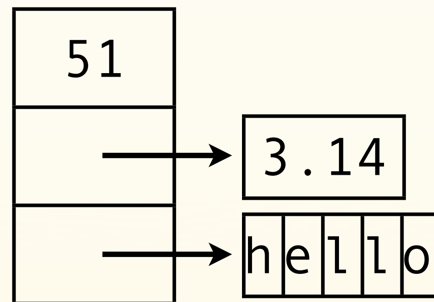
Data representations

The following drawings show how an object containing the integer 25, the real 3.14 and the string *hello* could be represented using the three techniques described earlier.

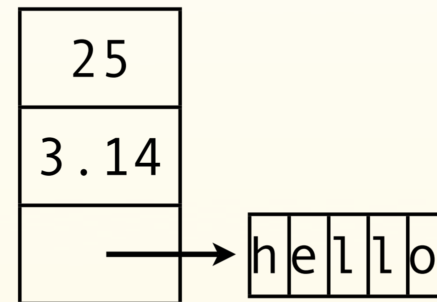
uniform



uniform with tagging



specialised



Polymorphism

Statically typed languages that offer polymorphism present the same problem as dynamically typed languages: the type of (some) data is not known at compilation time.

Three strategies are commonly used for such languages:

1. a uniform data representation is used for *all* data – except maybe for integers that can be tagged, or
2. a specialised representation is used for data stored in monomorphic containers, and a uniform one is used for data stored in polymorphic containers – which implies the generation of (un)boxing code, or
3. all polymorphism is “compiled away” through specialisation.

Specialisation

Specialisation consists in removing polymorphism by producing specialised, monomorphic code, each time some polymorphic code is used.

For example, when the type `List[Int]` appears in a program, the compiler produces a special class that represents lists of integers – and of nothing else.

Specialisation removes polymorphism, and therefore the need for a uniform representation of data. However, this is achieved at a considerable cost in terms of code size. Moreover, the specialisation process can loop for ever in pathological cases like:

```
class C[T];
class D[T] extends C[D[D[T]]];
```

Pointers in the stack

So far, we have only explained how pointers can be found in heap-allocated objects. But what about those appearing on the stack?

The stack is nothing but a singly- (and often implicitly-) linked list of stack frames. Therefore, the same solution as for heap-allocated objects can be used: every stack frame contains a header specifying the location of pointers in it.

This header can even be omitted if the compiler can guarantee that only pointers, tagged integers or return addresses are put on the stack.

Pointers in registers

For pointers appearing in registers, it is also possible to use a "header" stored in a known location in memory, giving the set of registers containing pointers.

Another solution is to partition the register set and guarantee that some registers contain only pointers, while other registers contain only other values.

Unknown data representation

All the data representation techniques presented until now enable the GC to unambiguously identify pointers at run time.

However, in some languages – *e.g.* C – it is not possible to obtain that information, neither statically nor dynamically. Is it still possible to perform garbage collection under such conditions?

Perhaps surprisingly, the answer to that question is yes!

A garbage collector that is able to work even without knowing how data is represented at run time is said to be **conservative**.

Conservative garbage collection

Conservative GC

A **conservative garbage collector** is one that is able to do its job without having to unambiguously identify pointers at run time.

The crucial observation behind conservative GC is that an approximation of the reachability graph is sufficient to collect (some) garbage, as long as that approximation encompasses 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.

Copying garbage collection

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**.

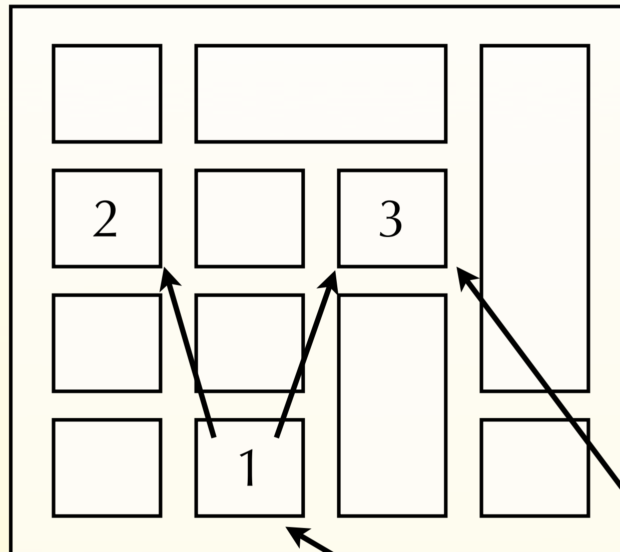
Memory is allocated in from-space, while to-space is left empty.

When from-space is full, all reachable objects in from-space are copied to to-space, and pointers to them are updated accordingly.

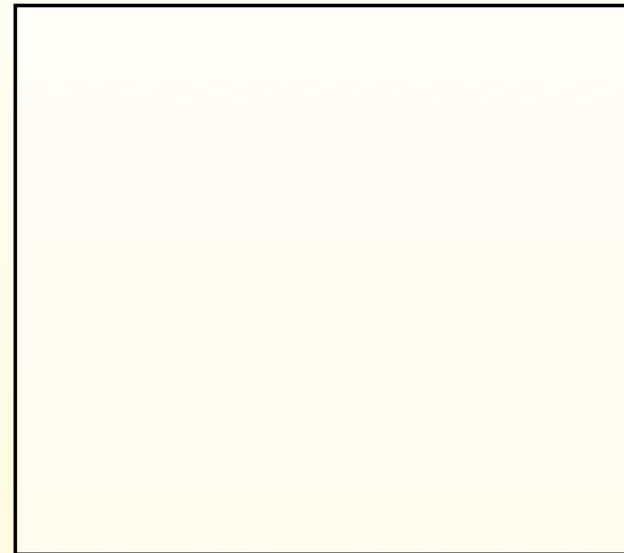
Finally, the role of the two spaces is exchanged, and the program resumed.

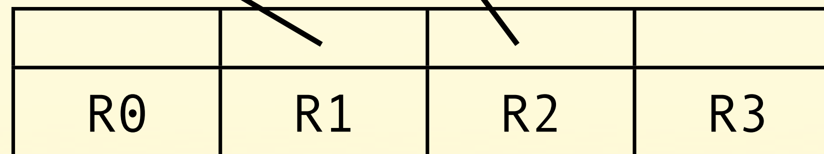
Copying GC

From



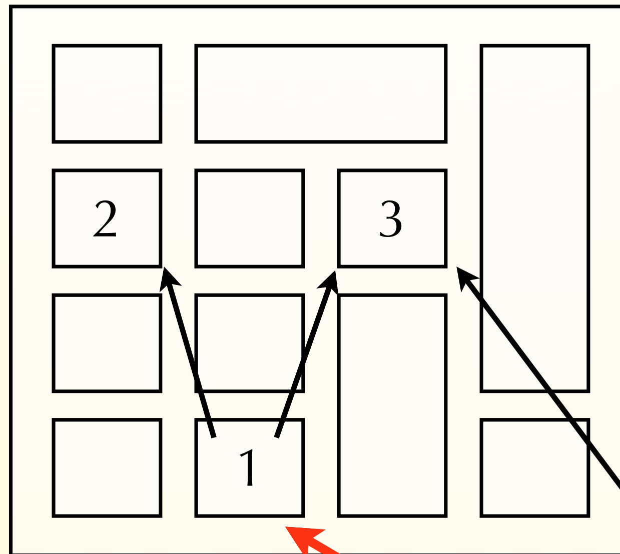
To



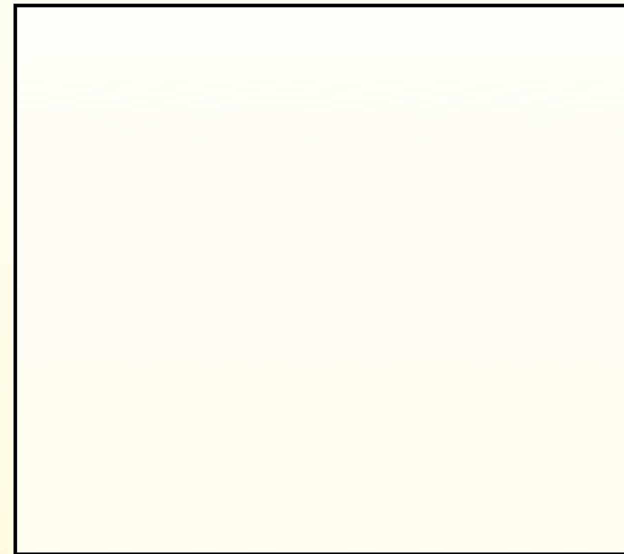


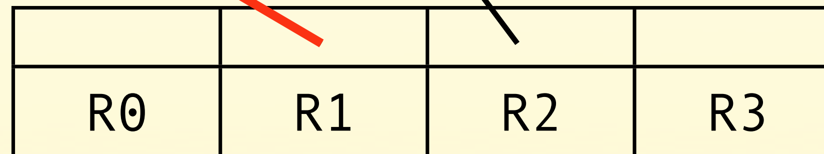
Copying GC

From



To

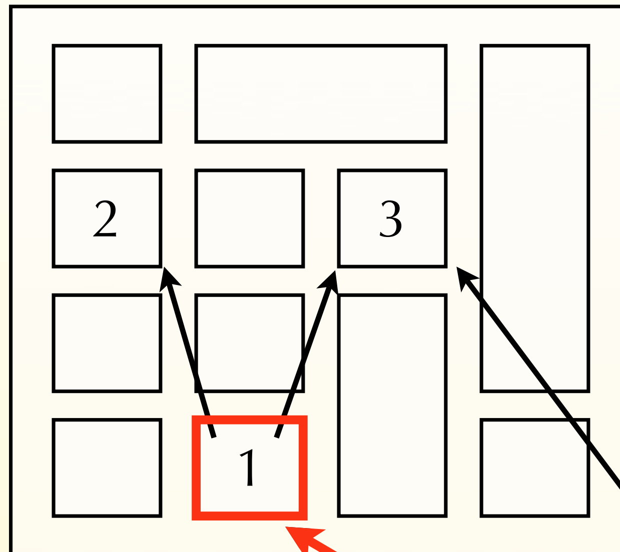


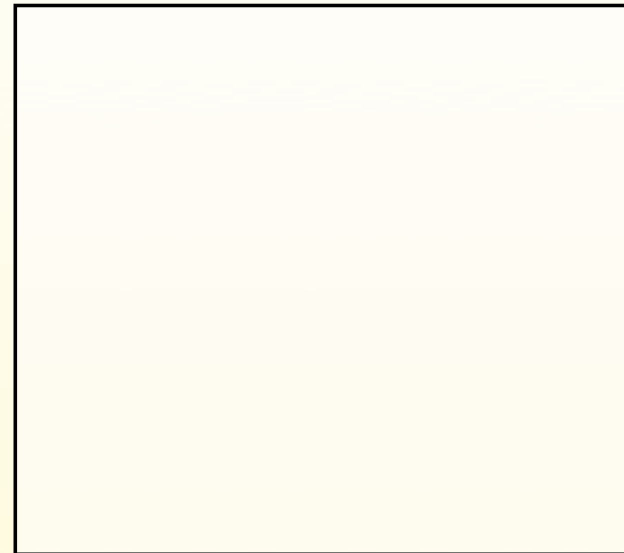


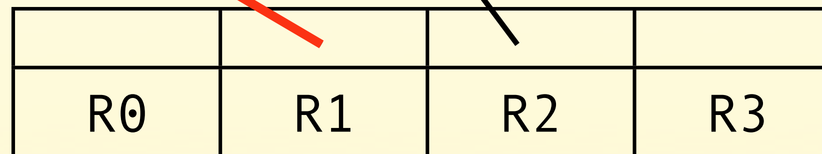
Copying GC

From

To

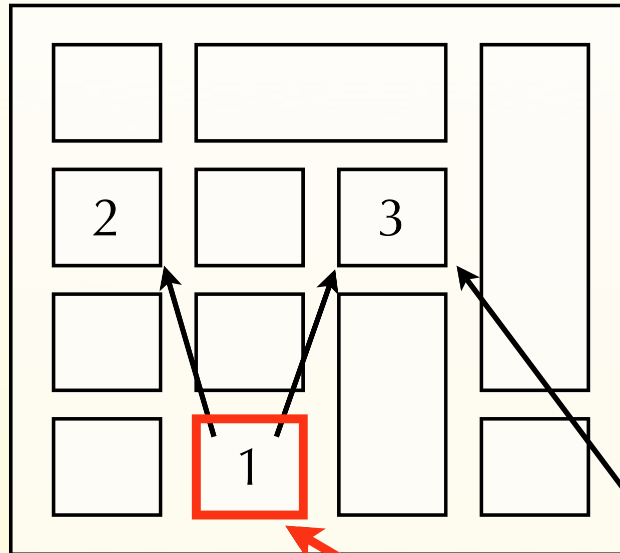




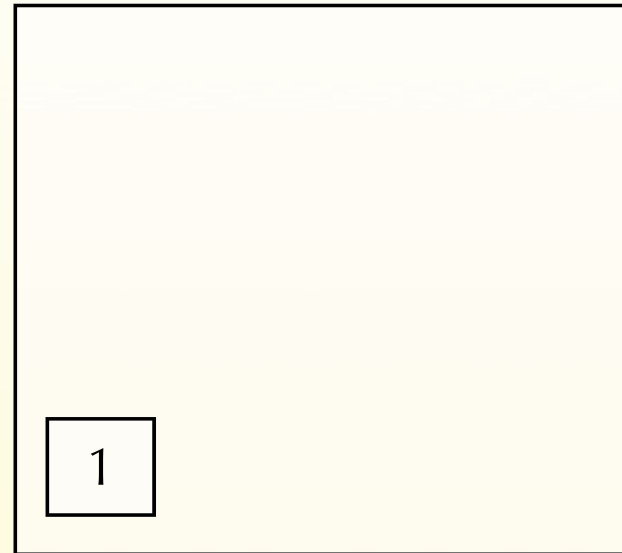


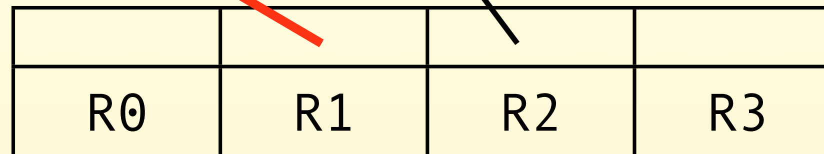
Copying GC

From



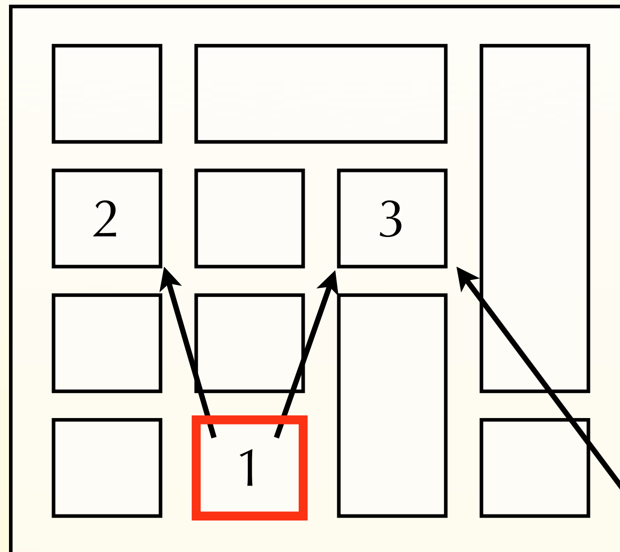
To



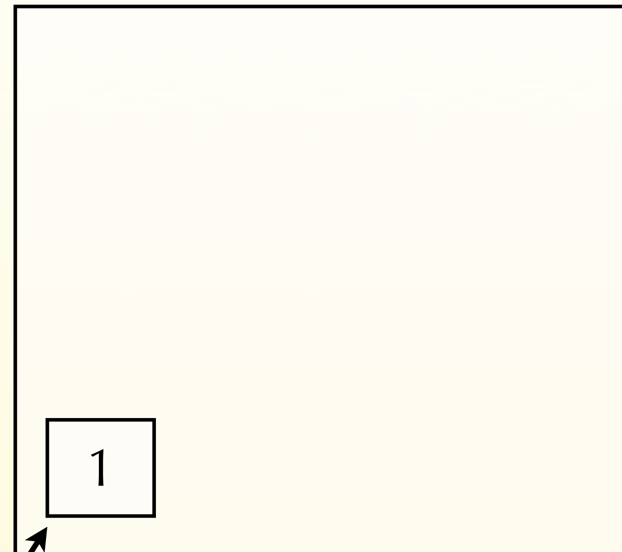


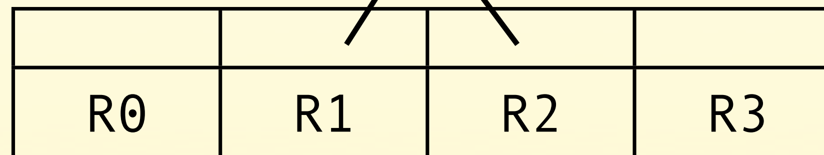
Copying GC

From

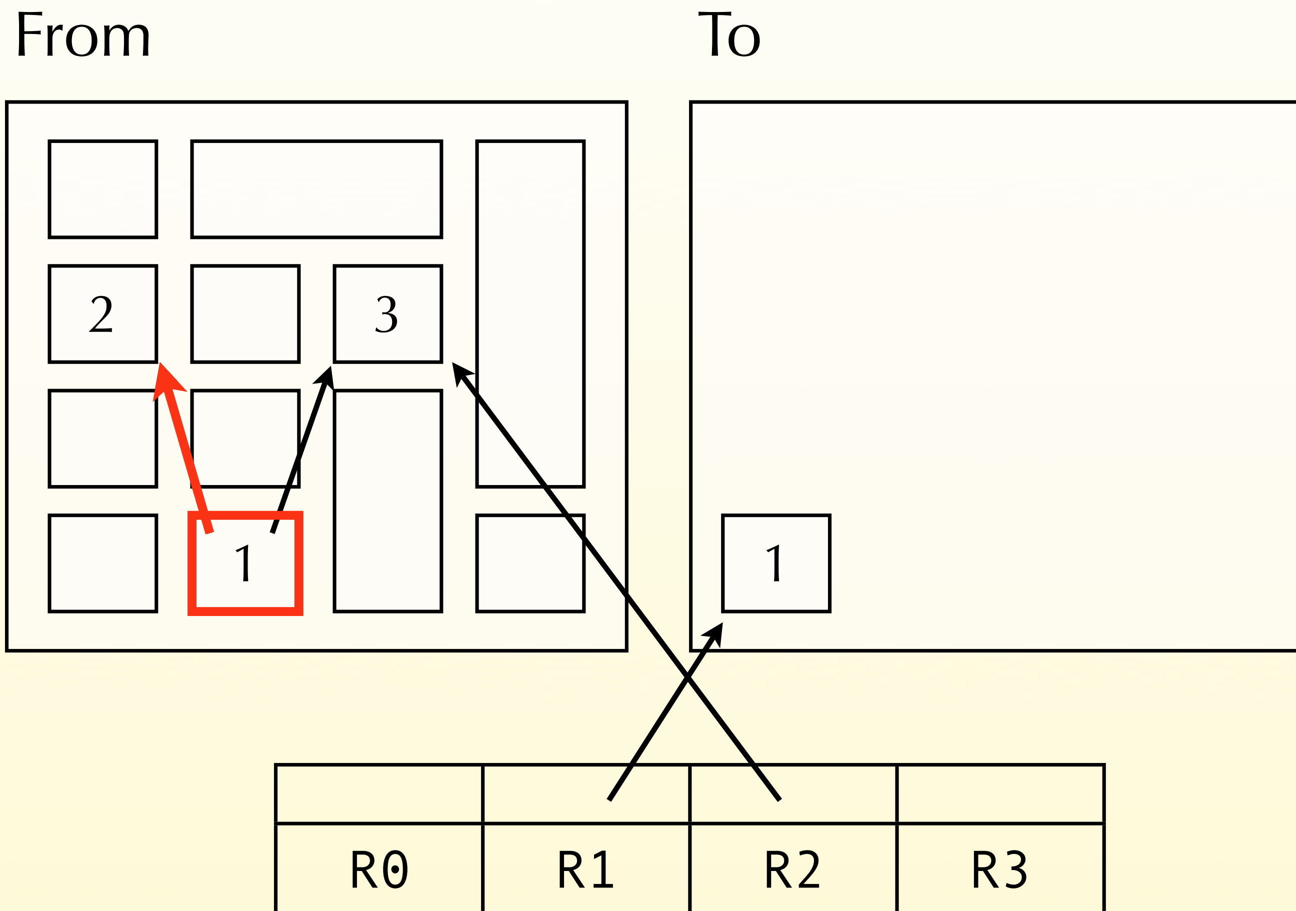


To

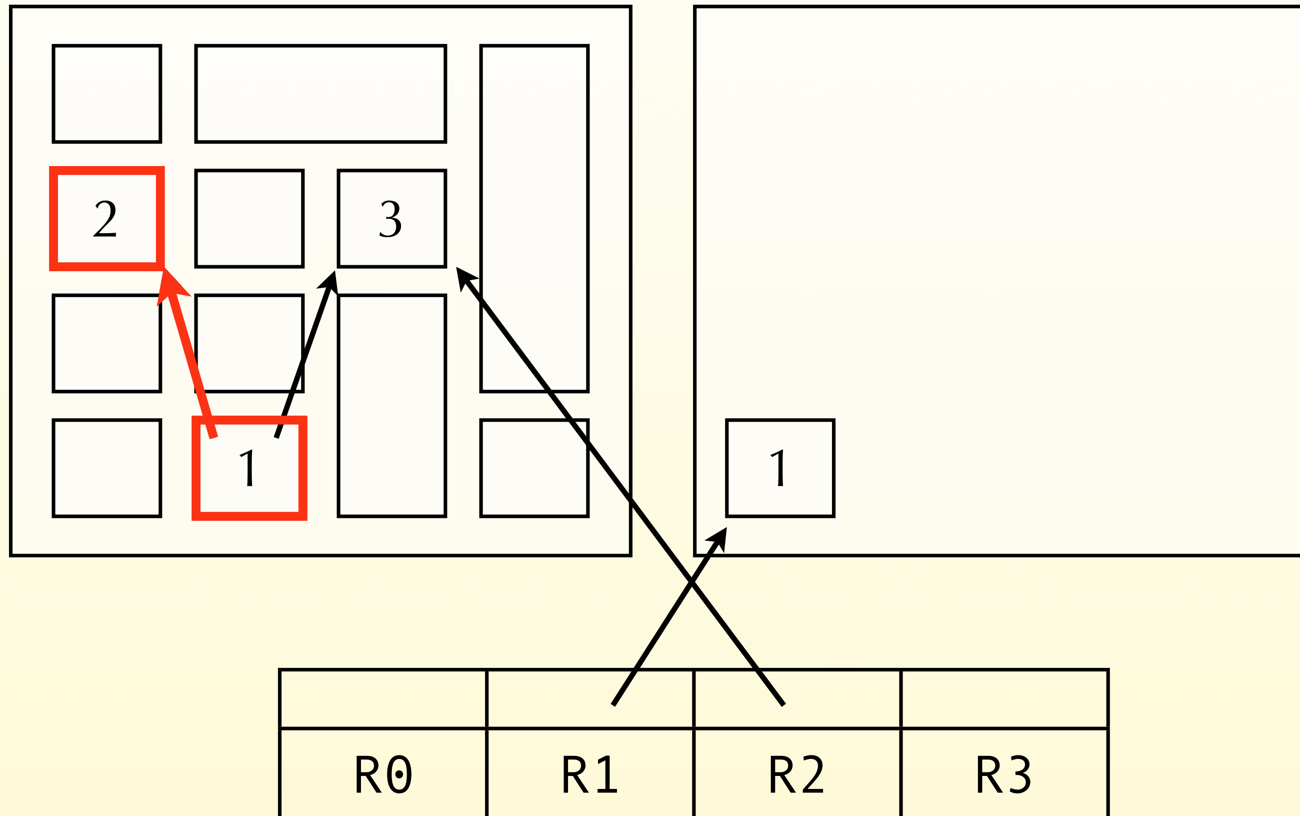




Copying GC

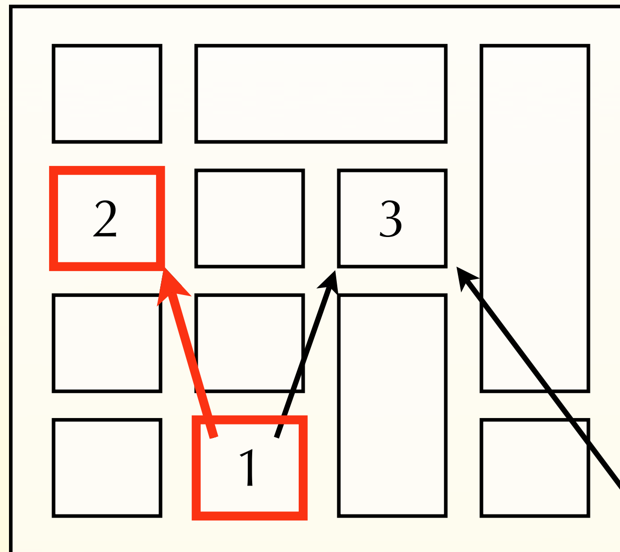


Copying GC

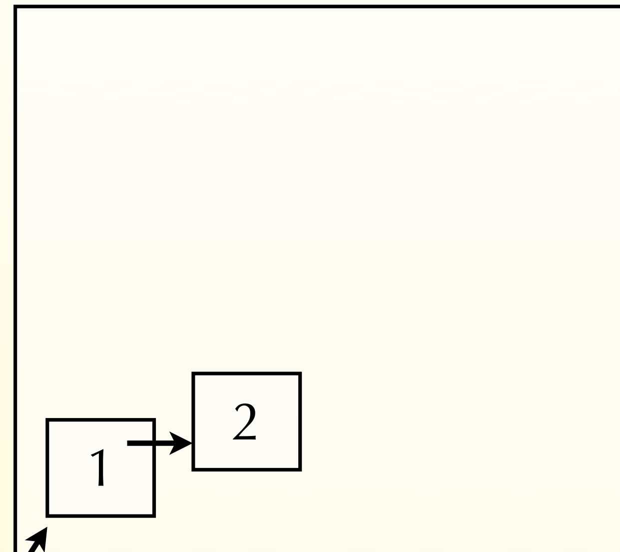


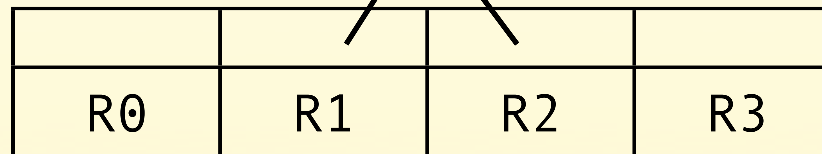
Copying GC

From



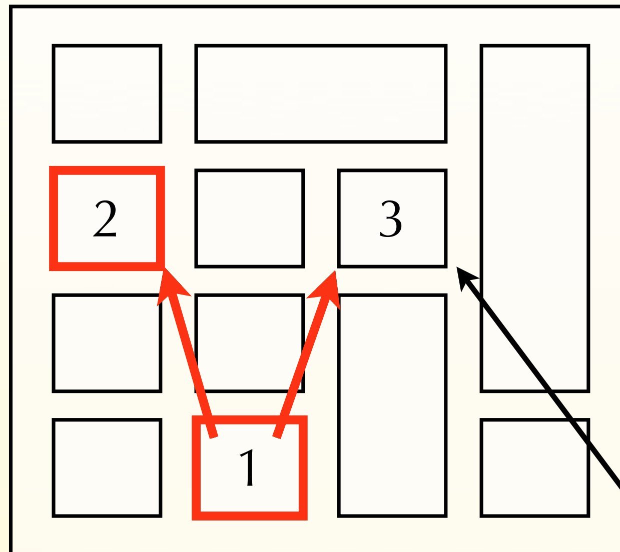
To



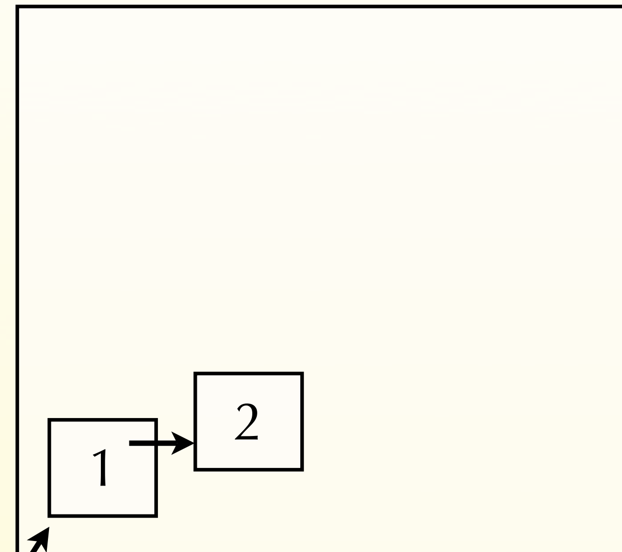


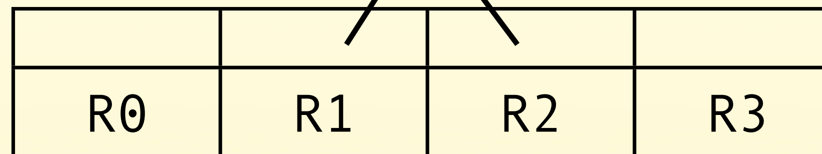
Copying GC

From



To

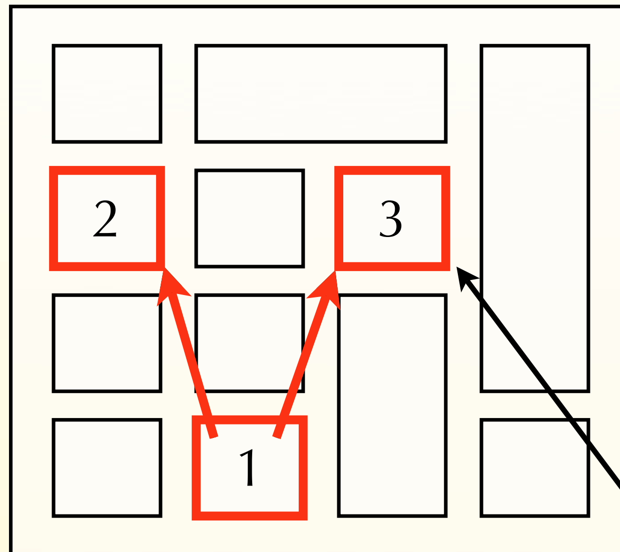


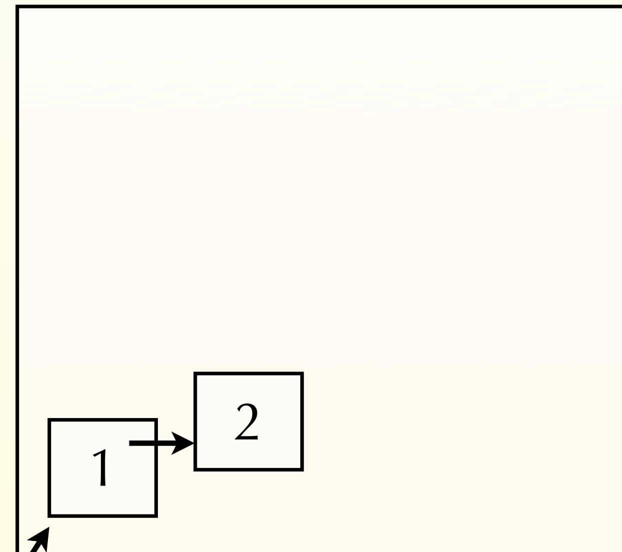


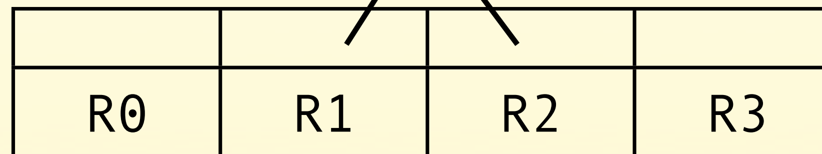
Copying GC

From

To



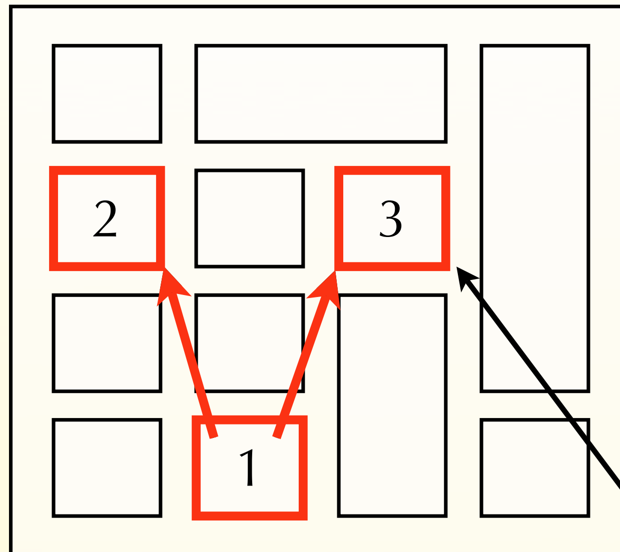


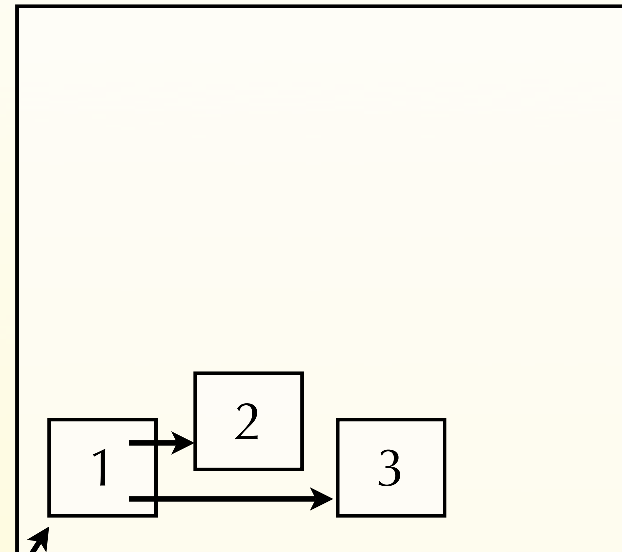


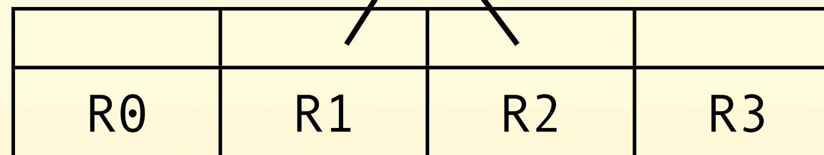
Copying GC

From

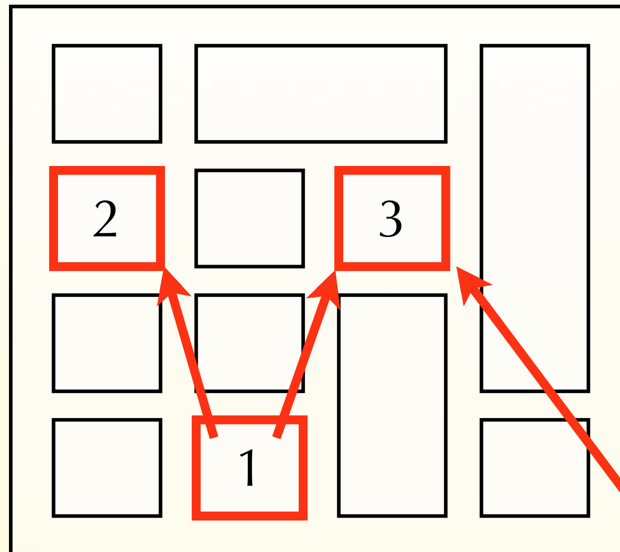
To

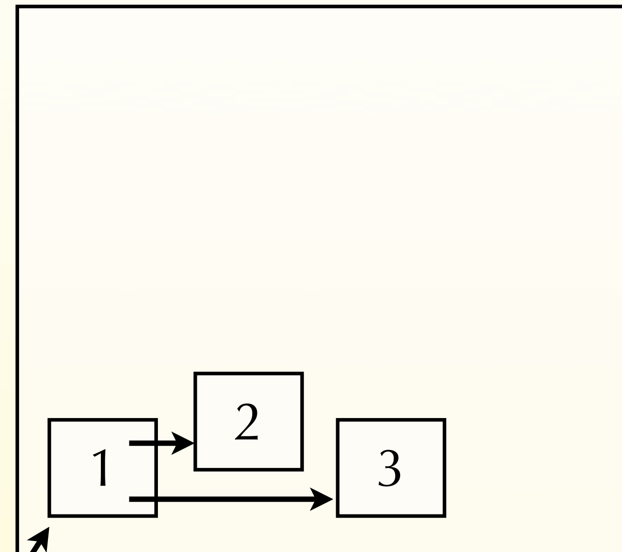






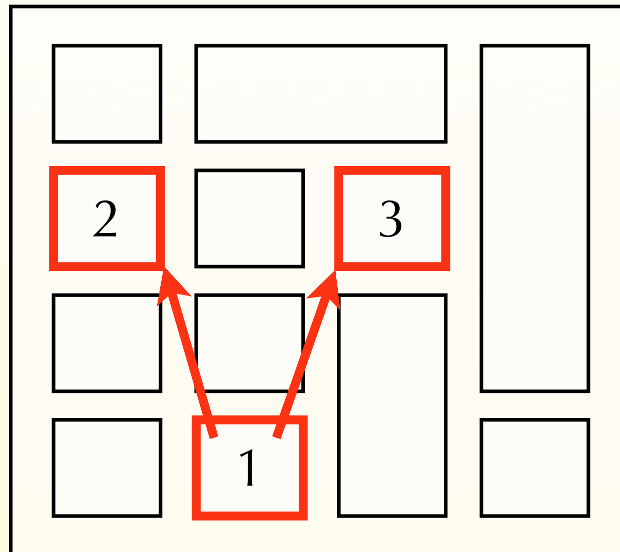
Copying GC



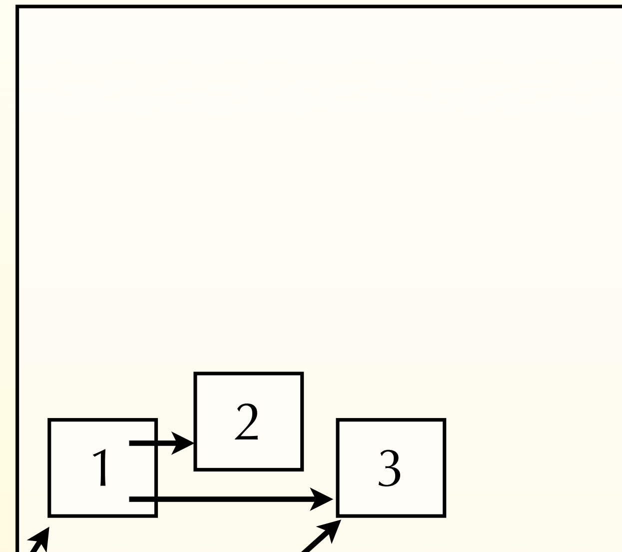


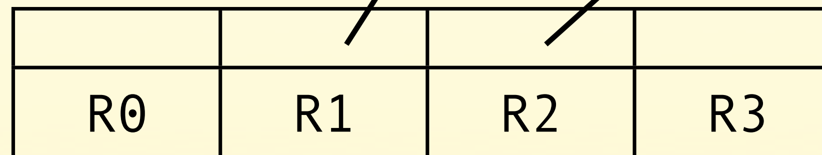
Copying GC

From

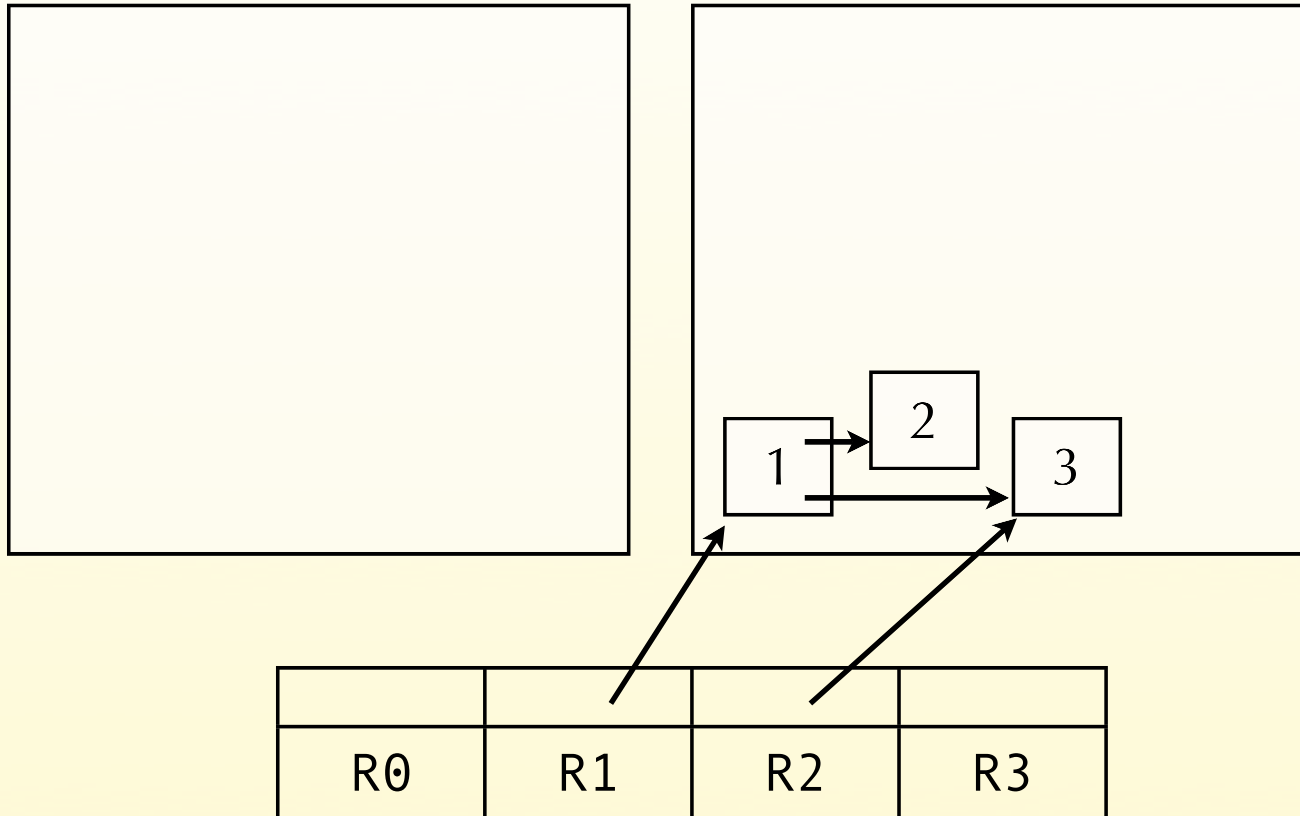


To

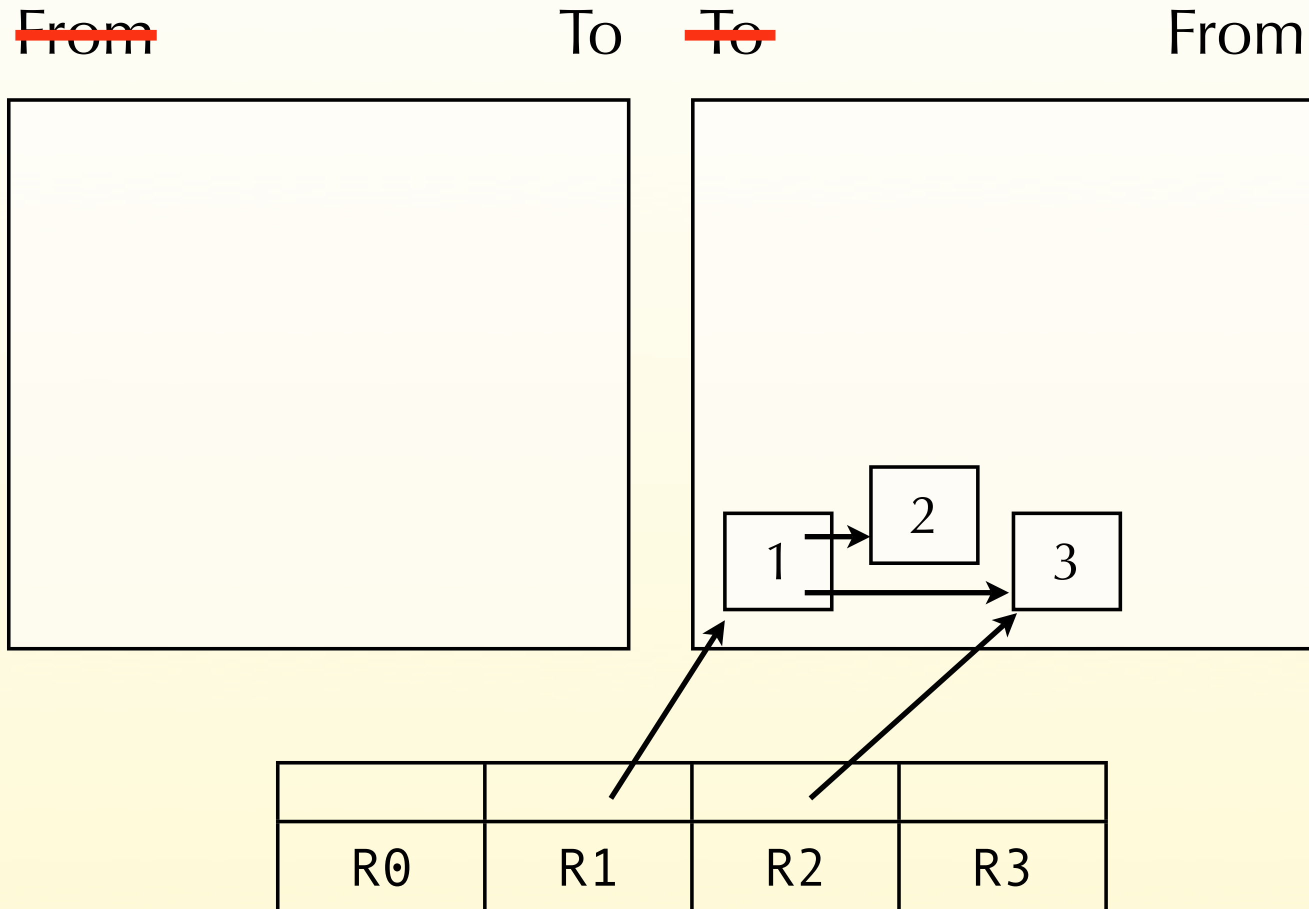




Copying GC



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.

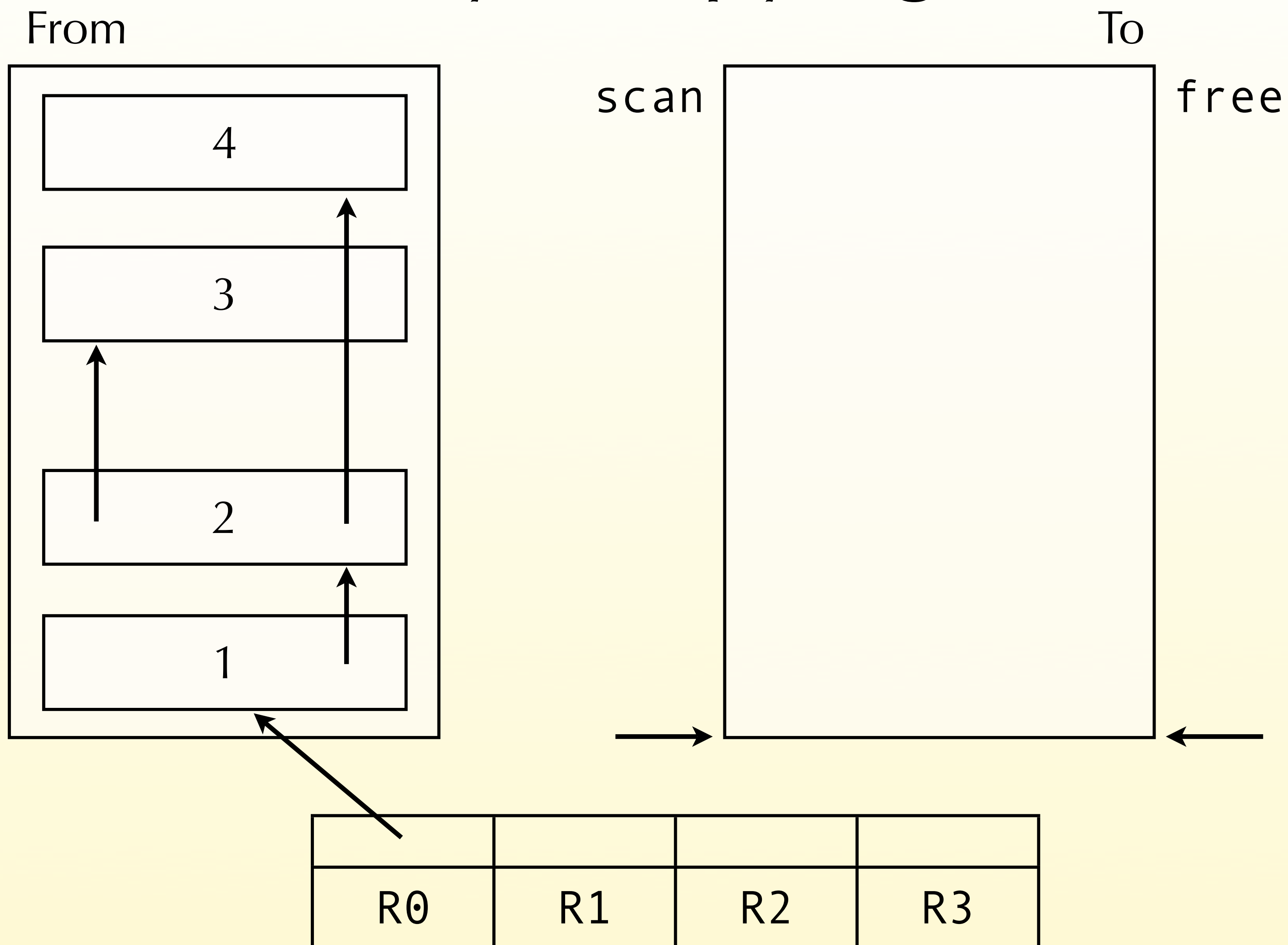
Cheney’s copying GC

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

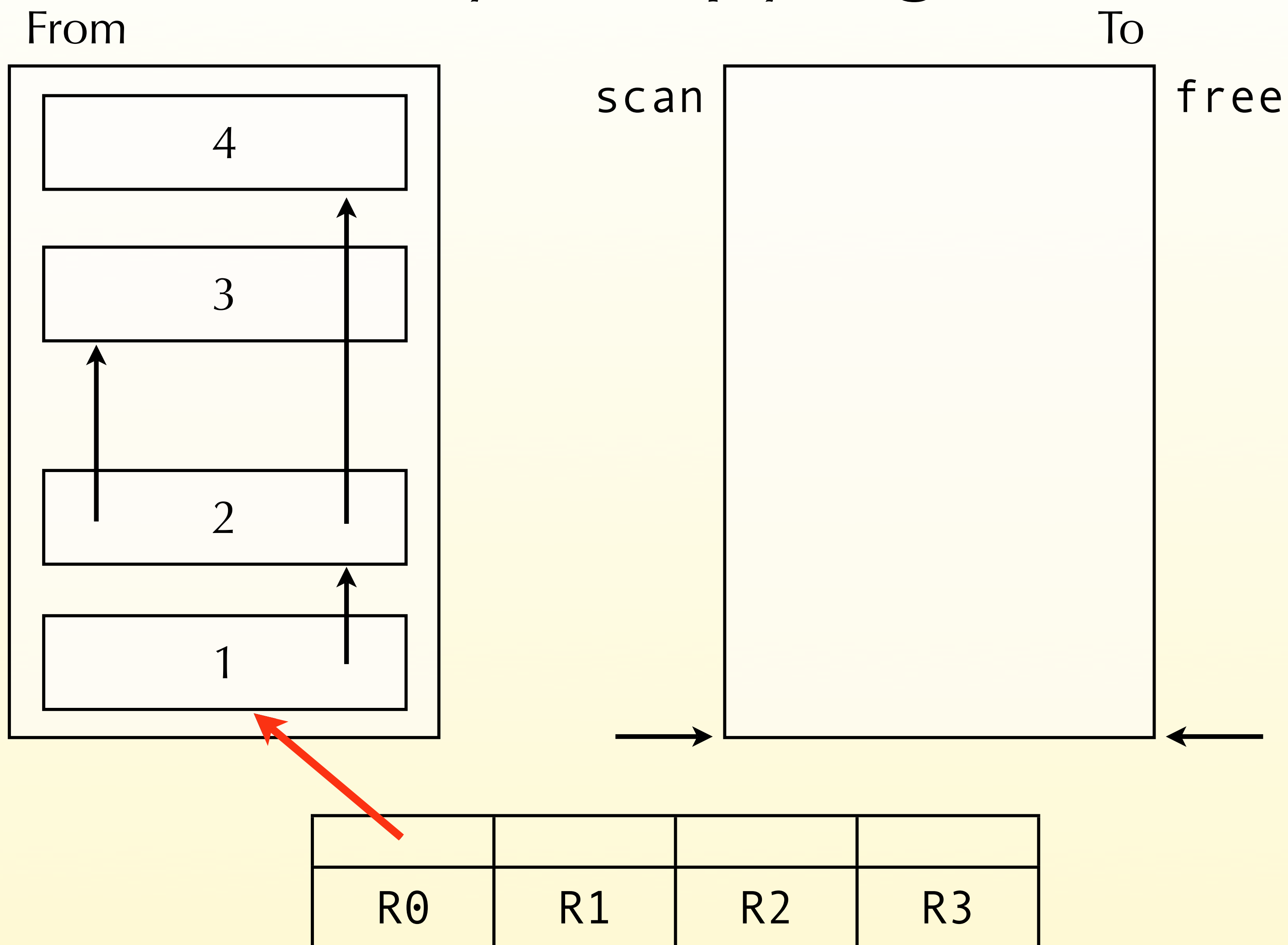
The basic idea of Cheney’s algorithm is to use to-space to store this set of nodes, which can be represented using a single pointer called `scan`.

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

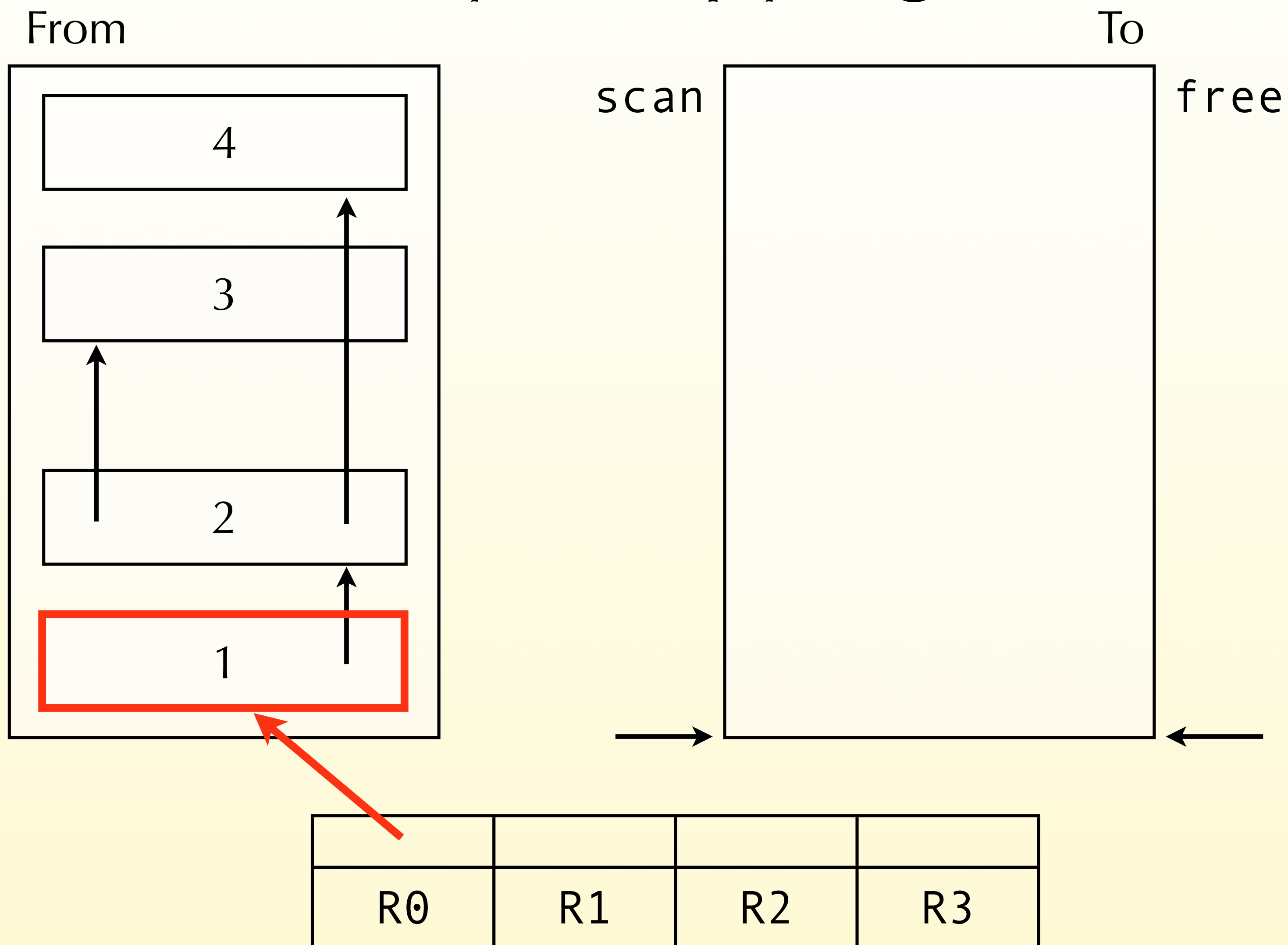
Cheney's copying GC



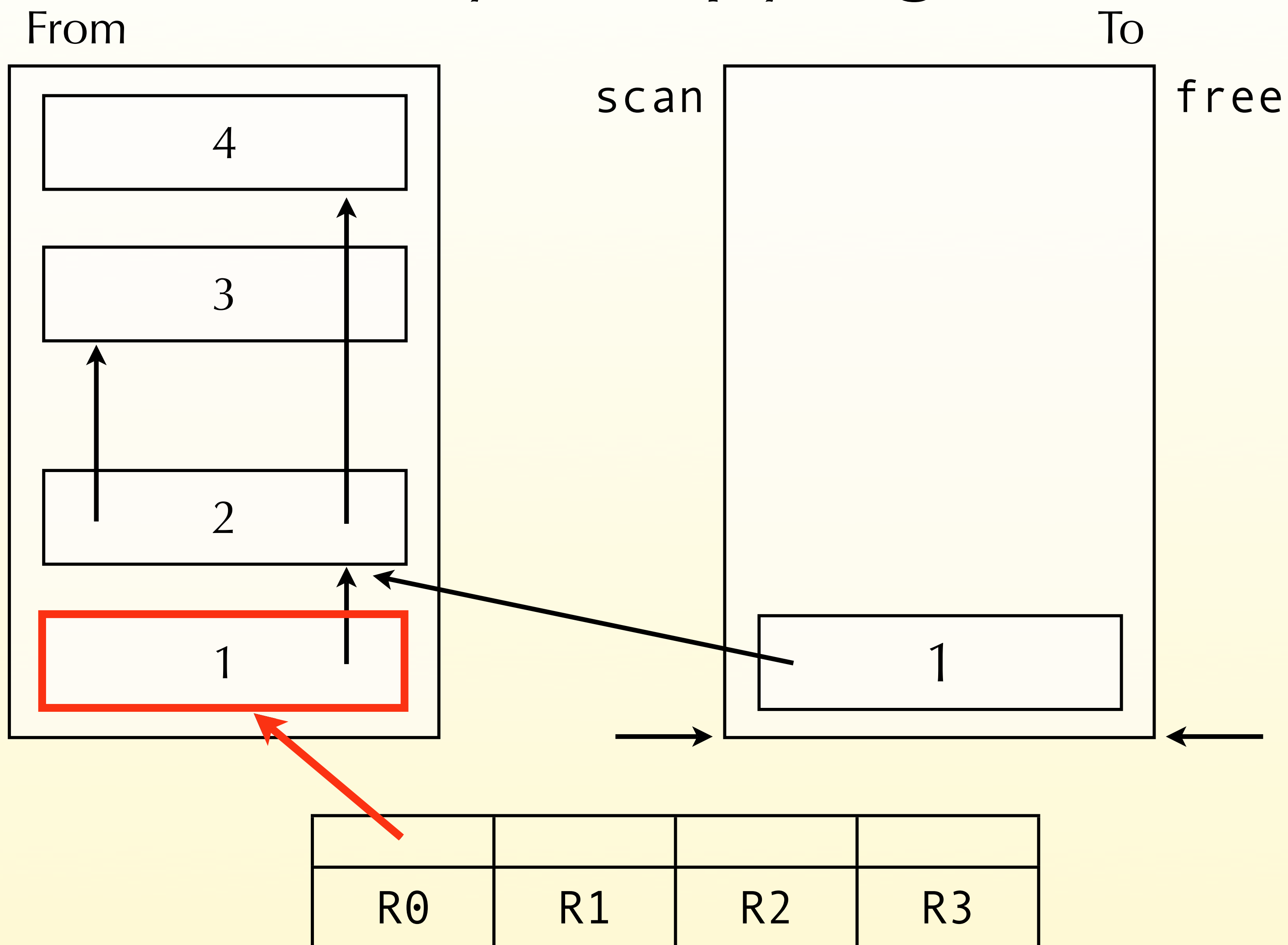
Cheney's copying GC



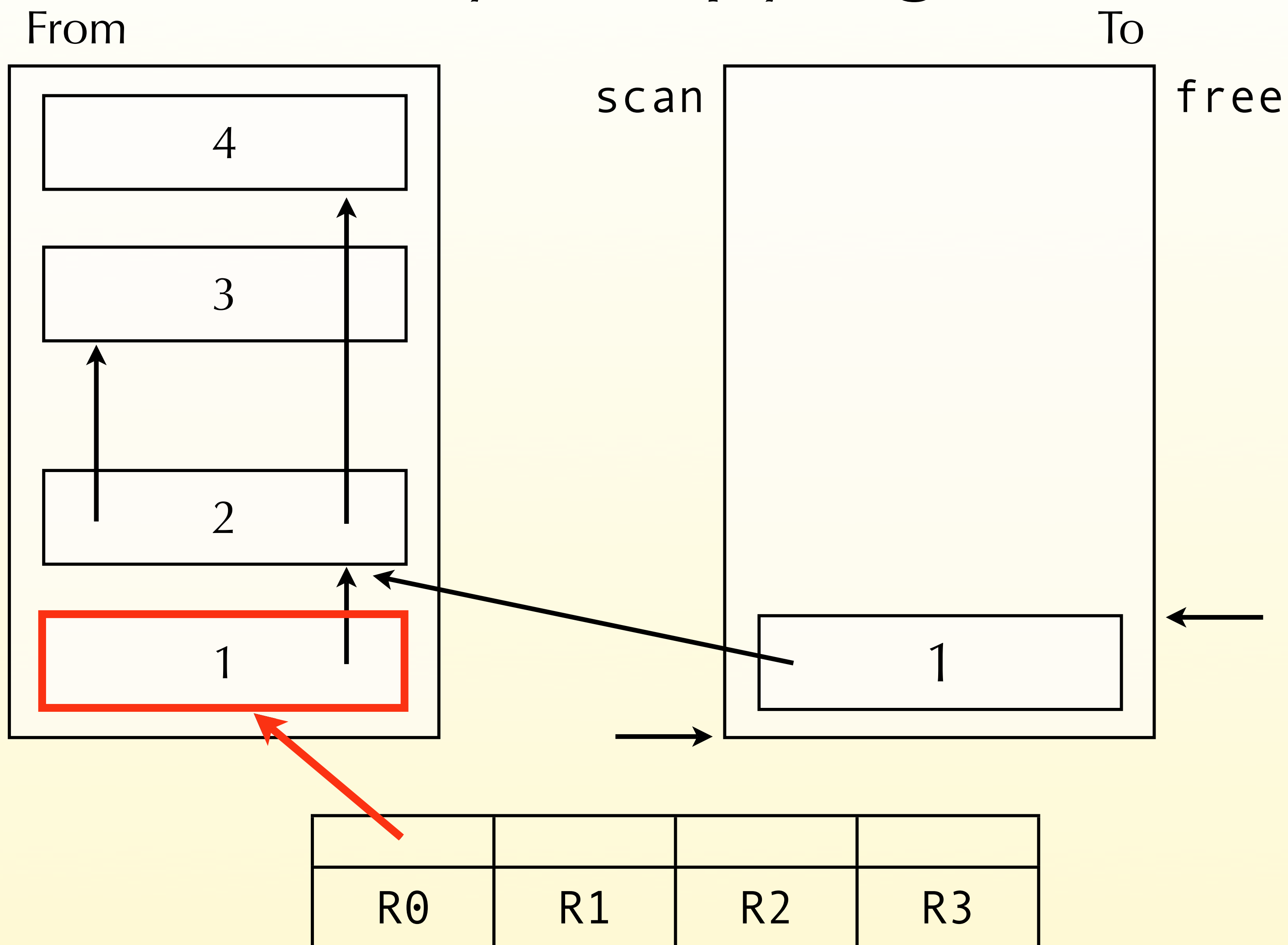
Cheney's copying GC



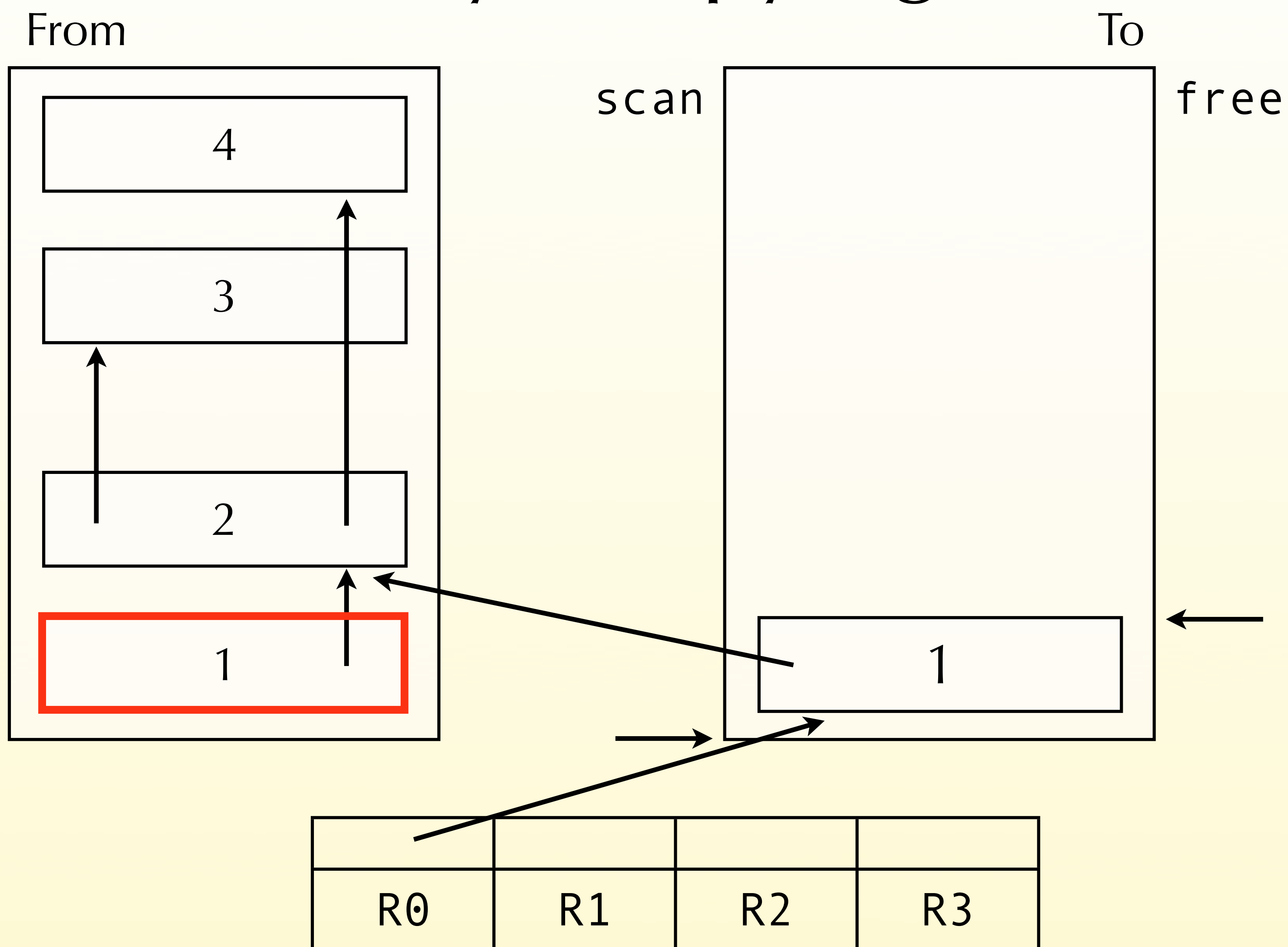
Cheney's copying GC



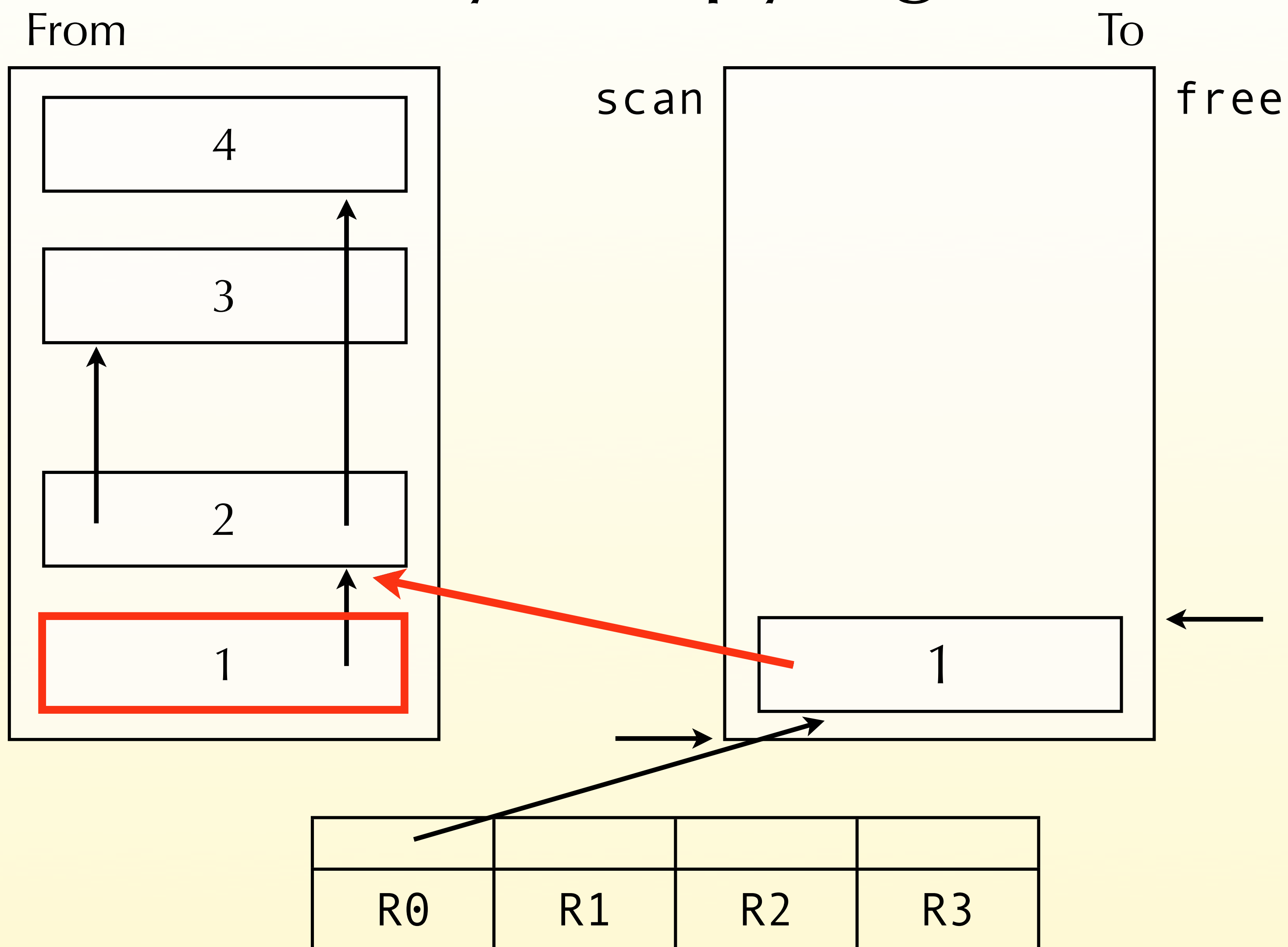
Cheney's copying GC



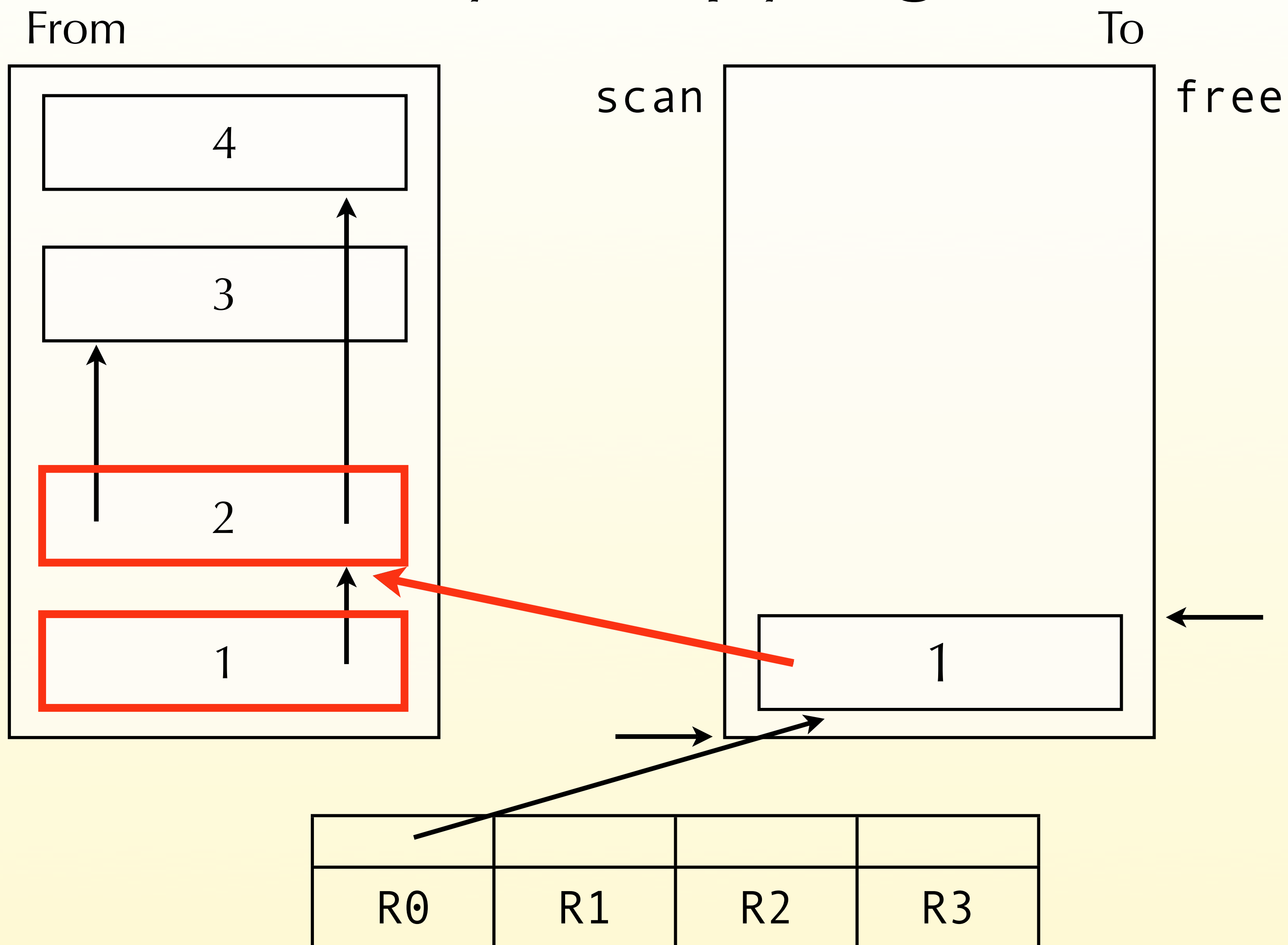
Cheney's copying GC



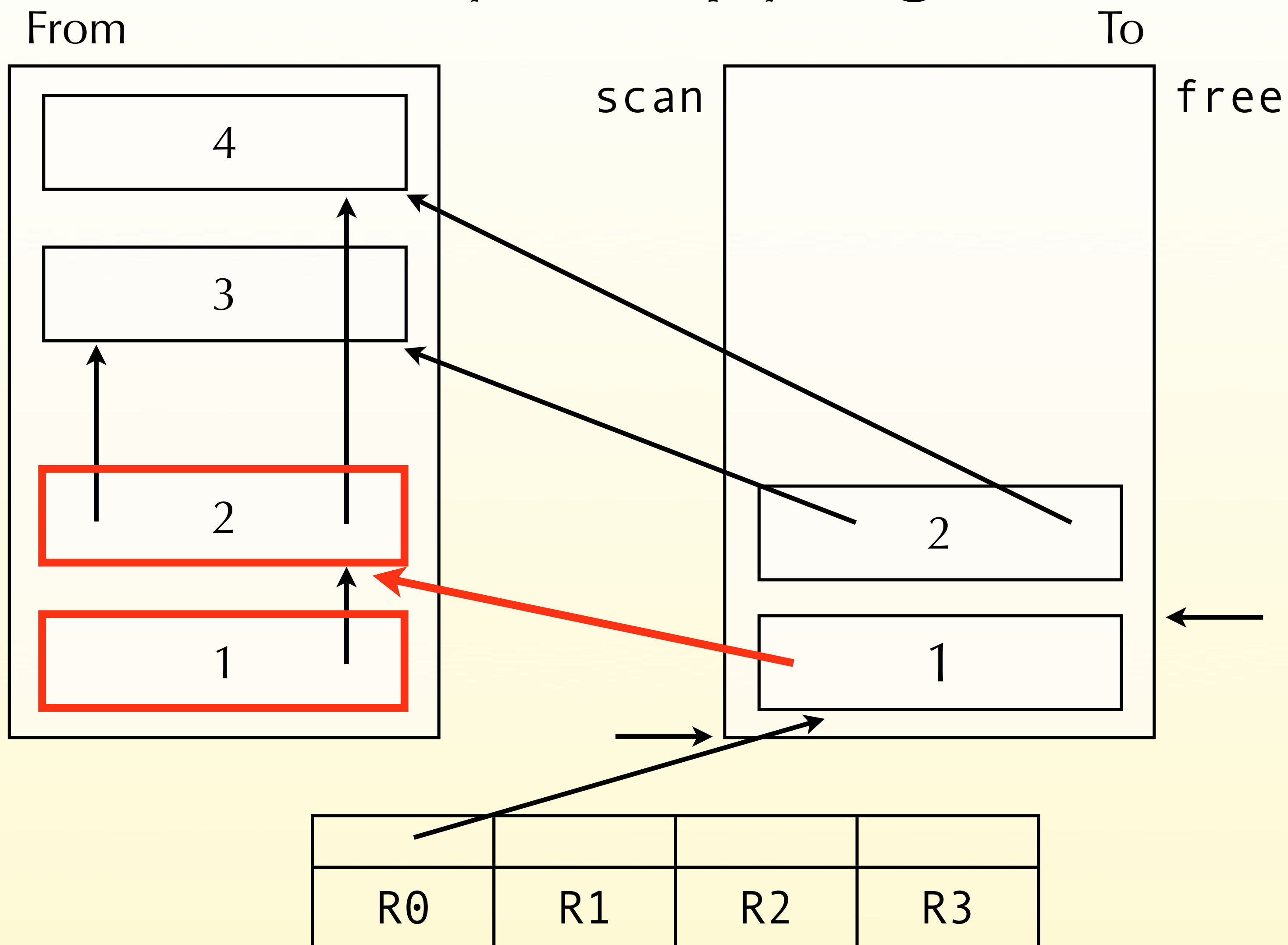
Cheney's copying GC



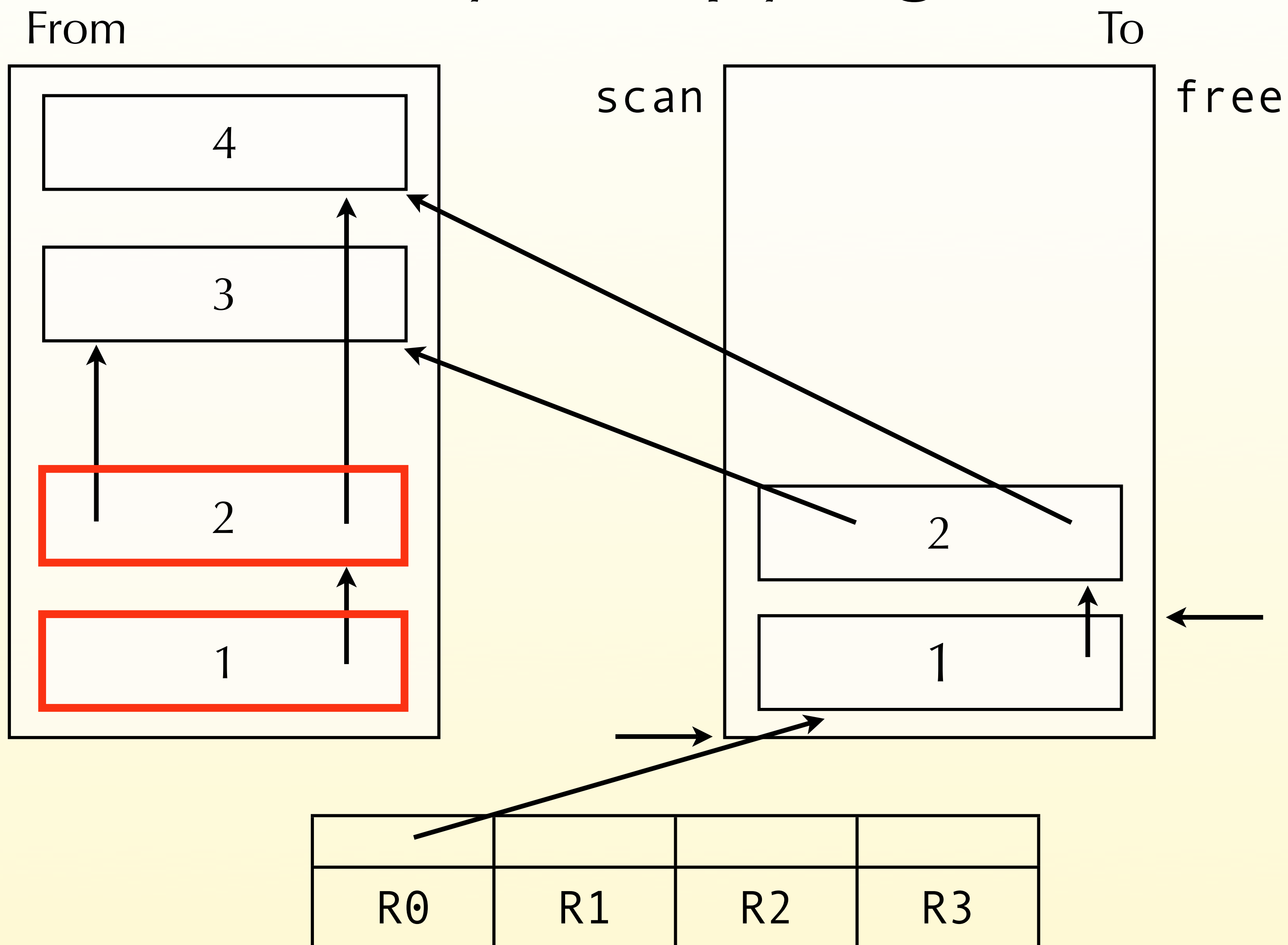
Cheney's copying GC



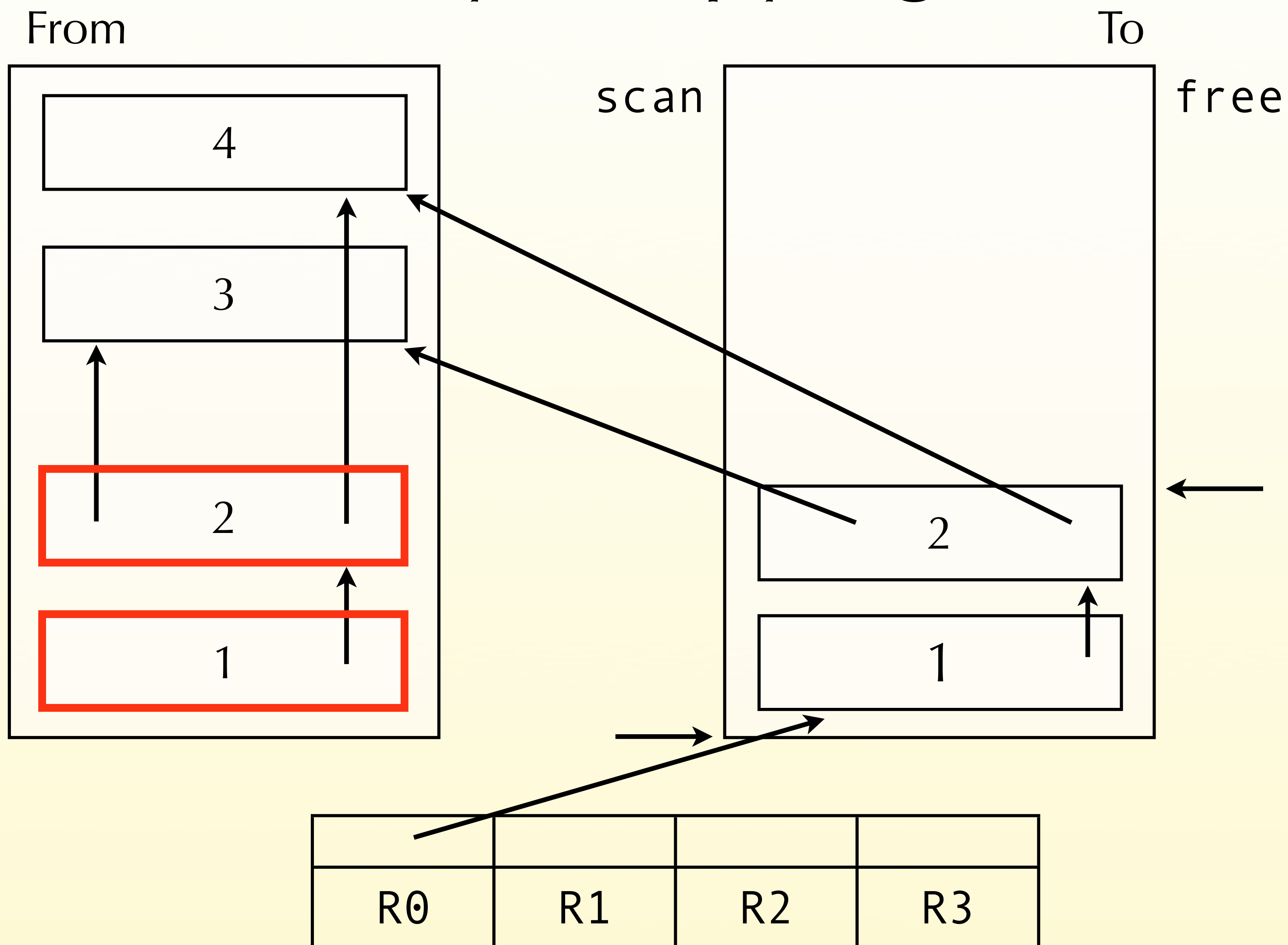
Cheney's copying GC



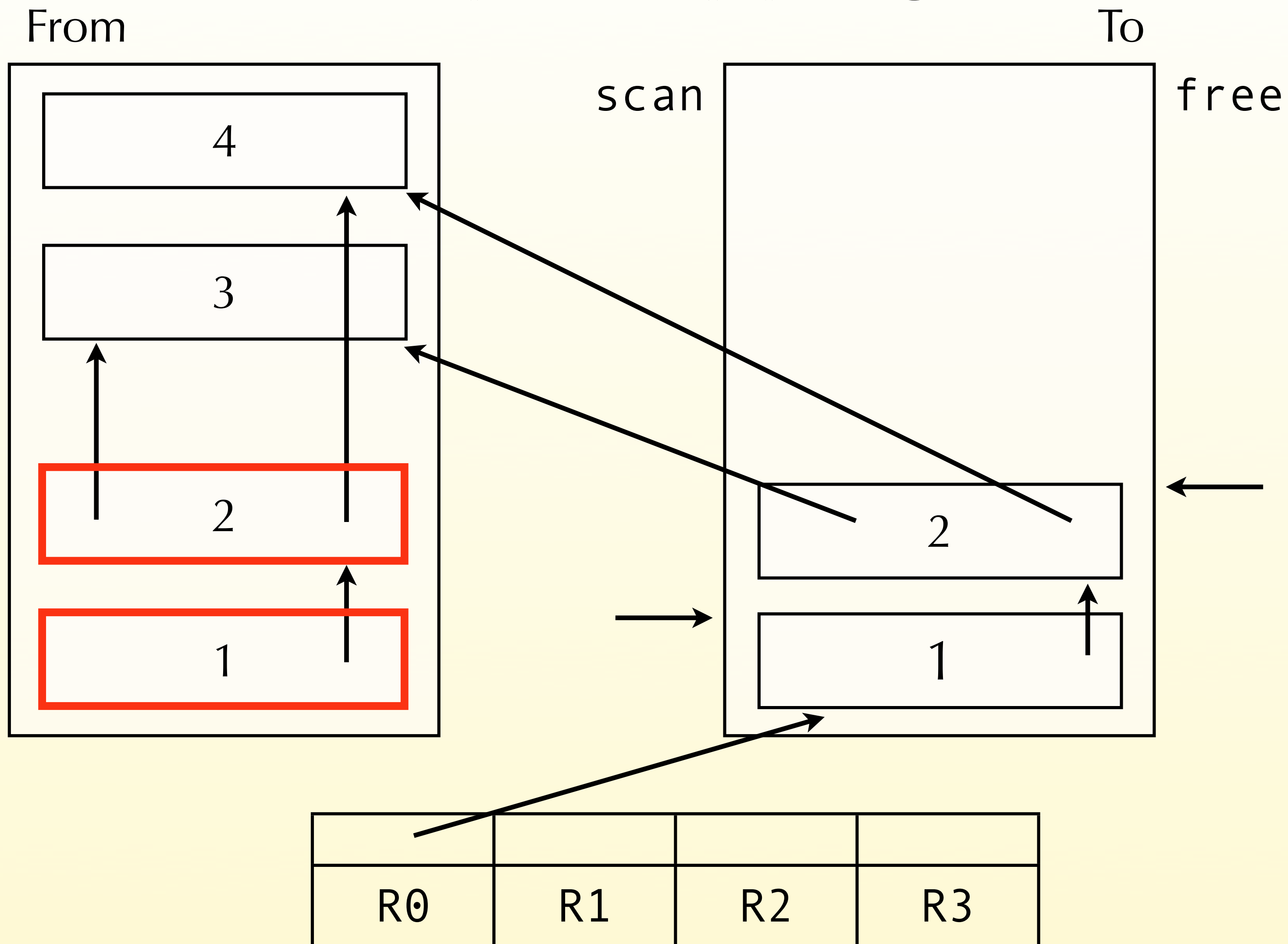
Cheney's copying GC



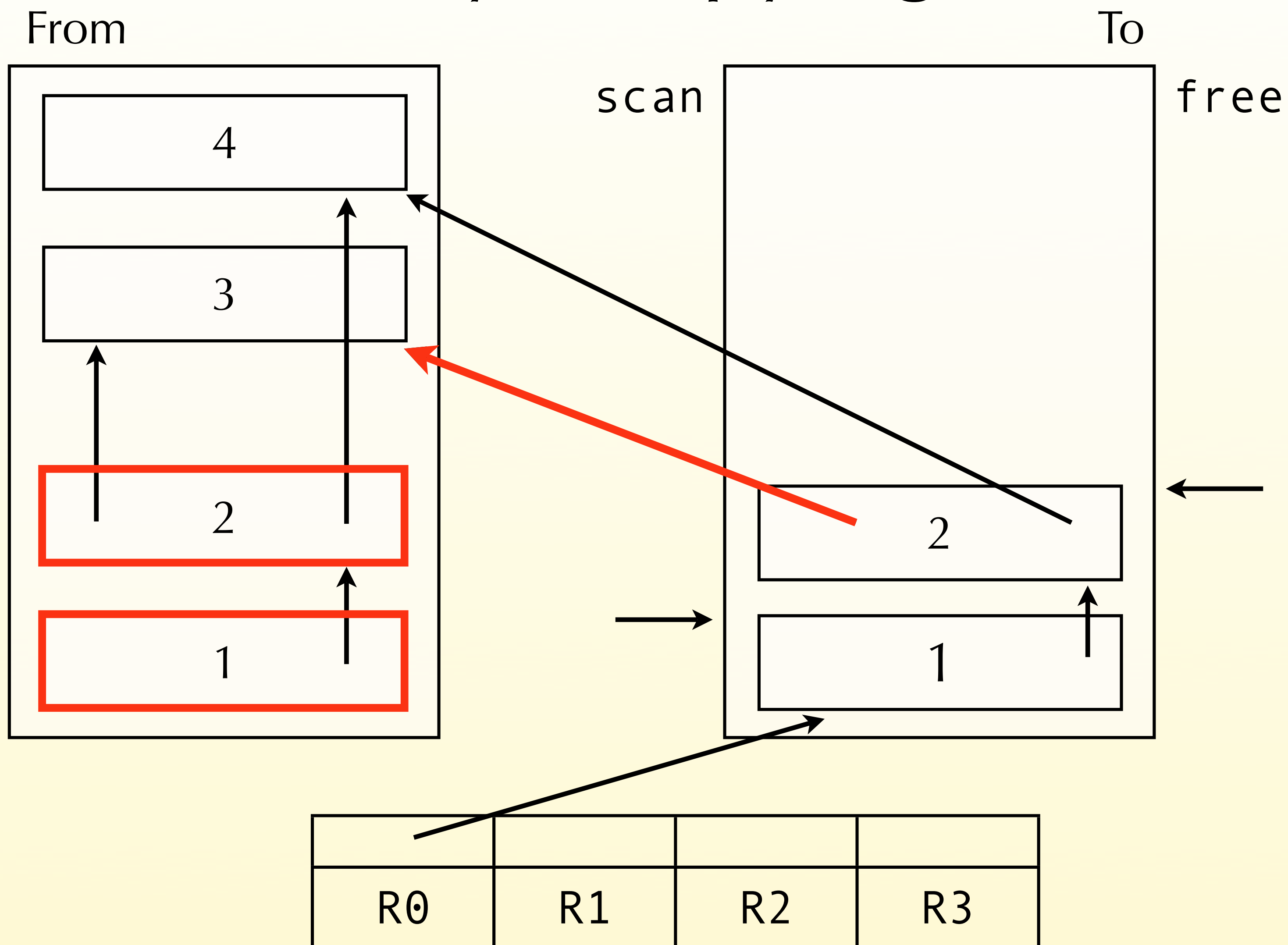
Cheney's copying GC



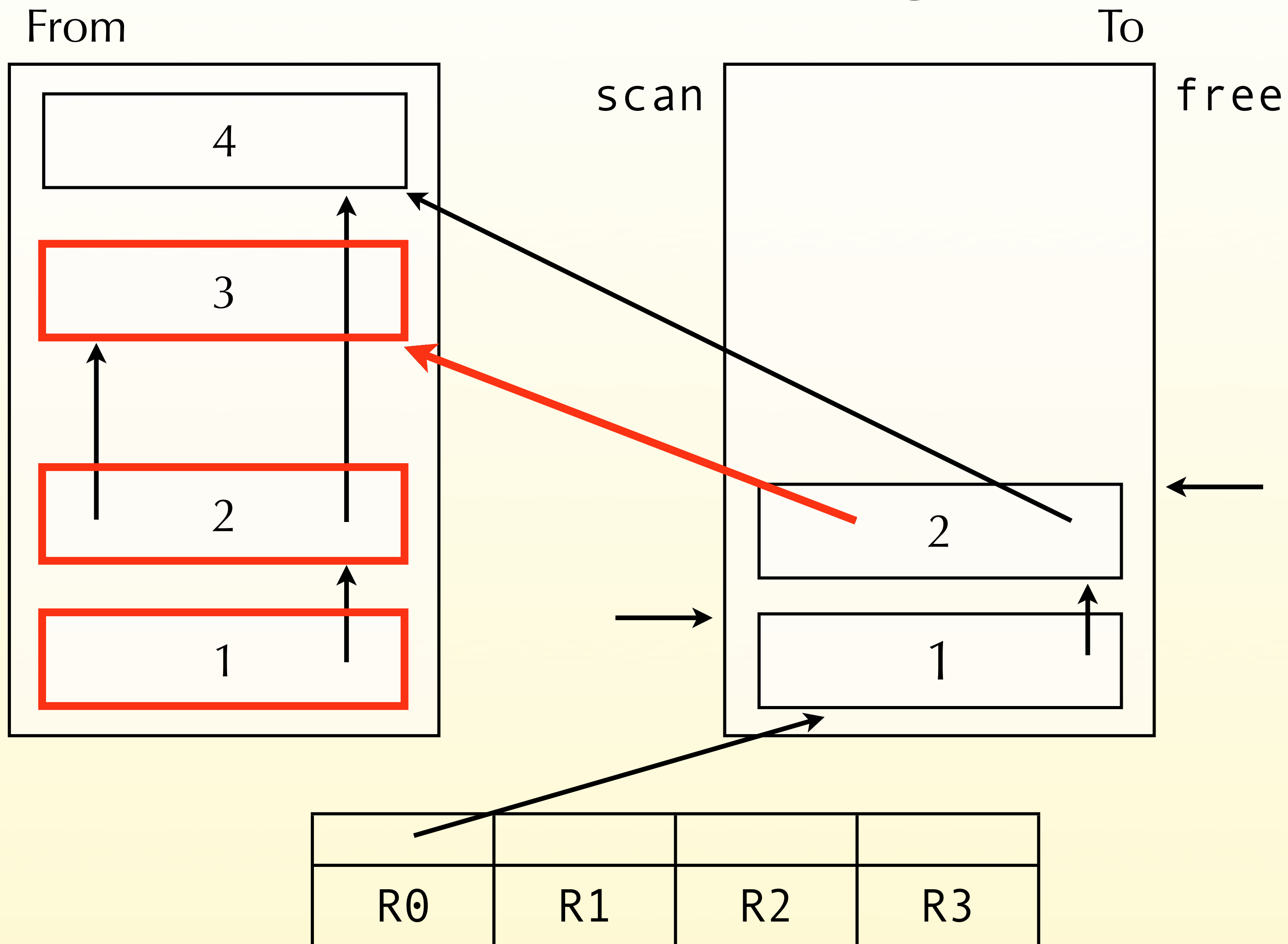
Cheney's copying GC



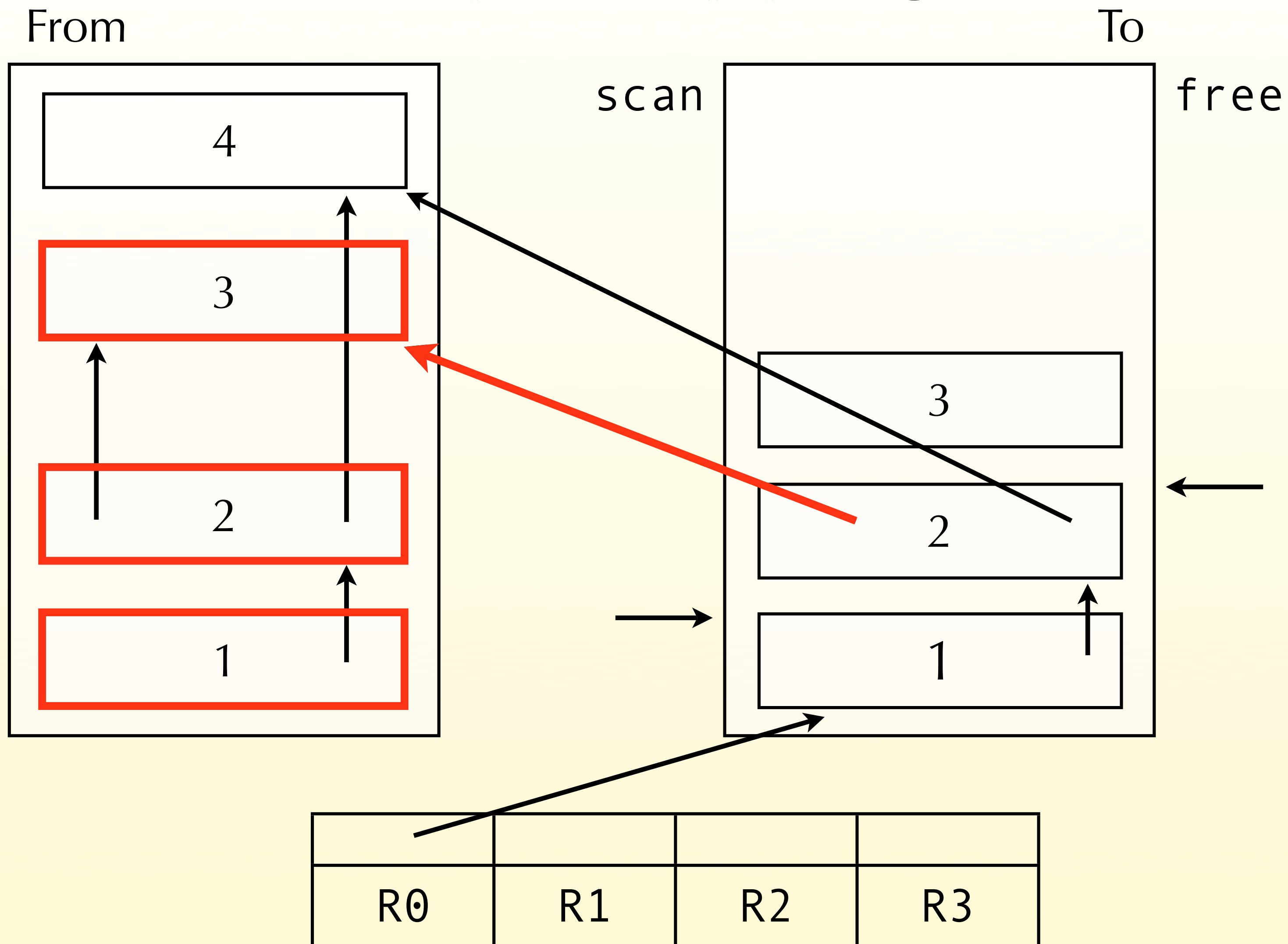
Cheney's copying GC



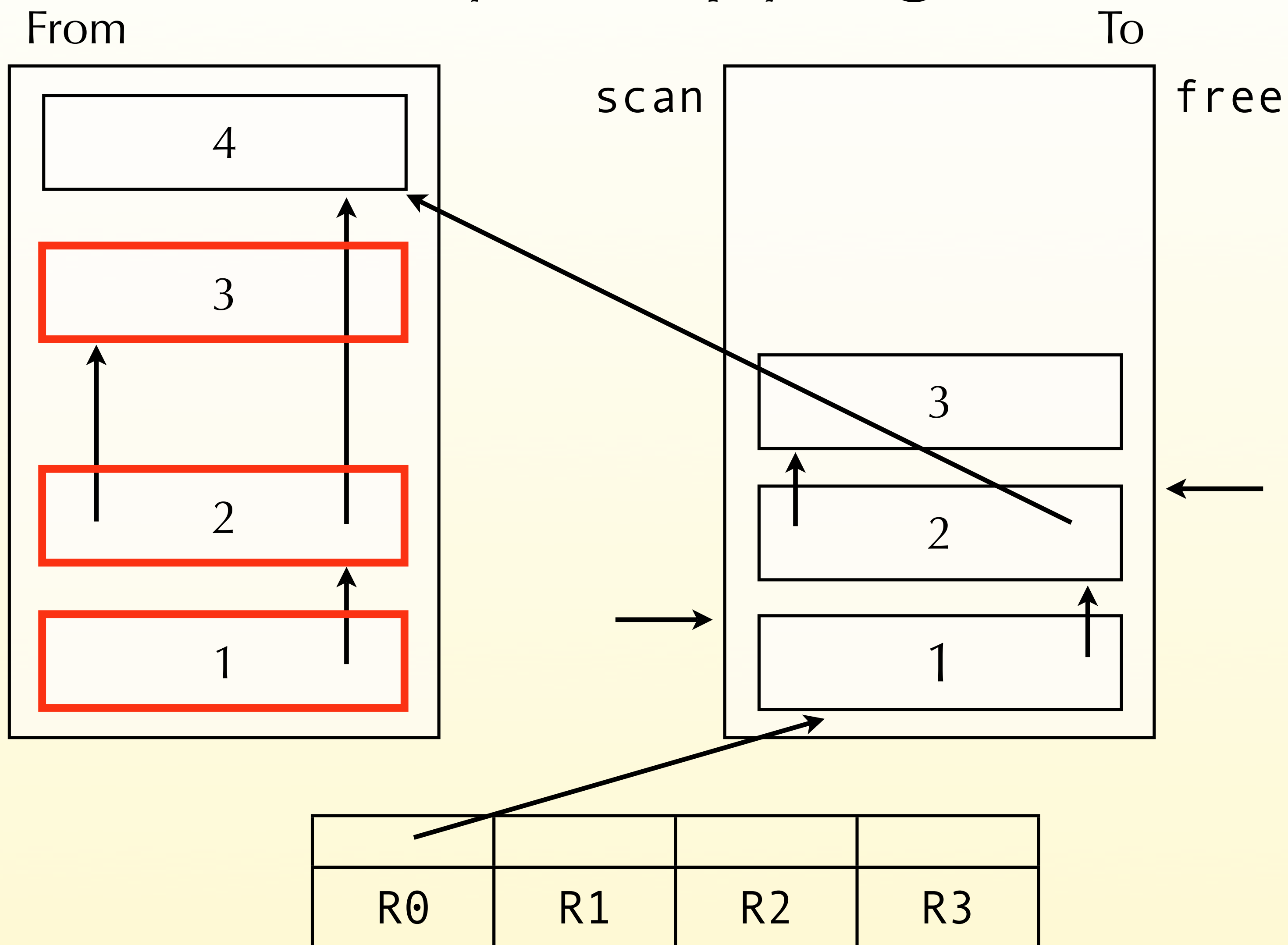
Cheney's copying GC



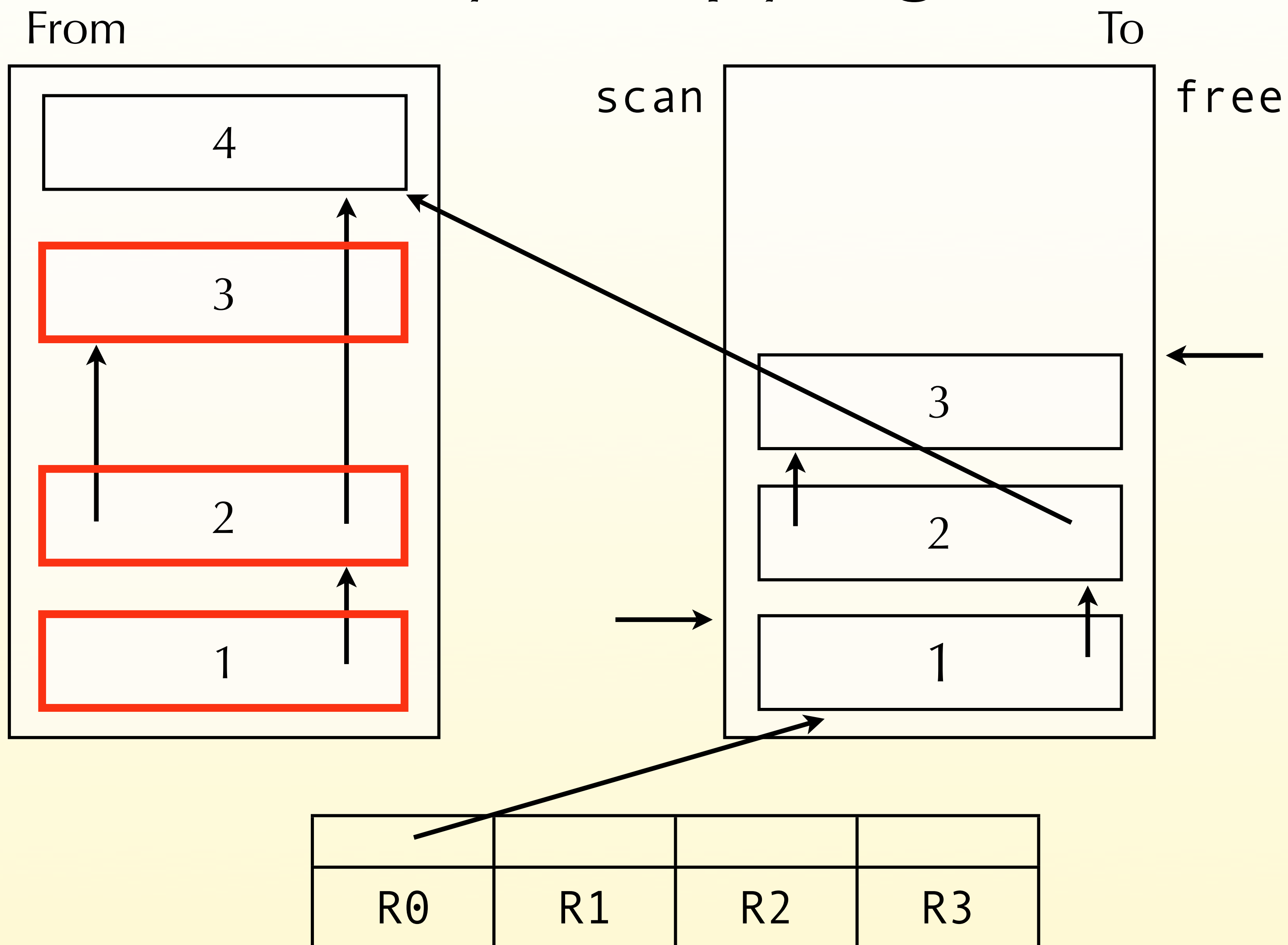
Cheney's copying GC



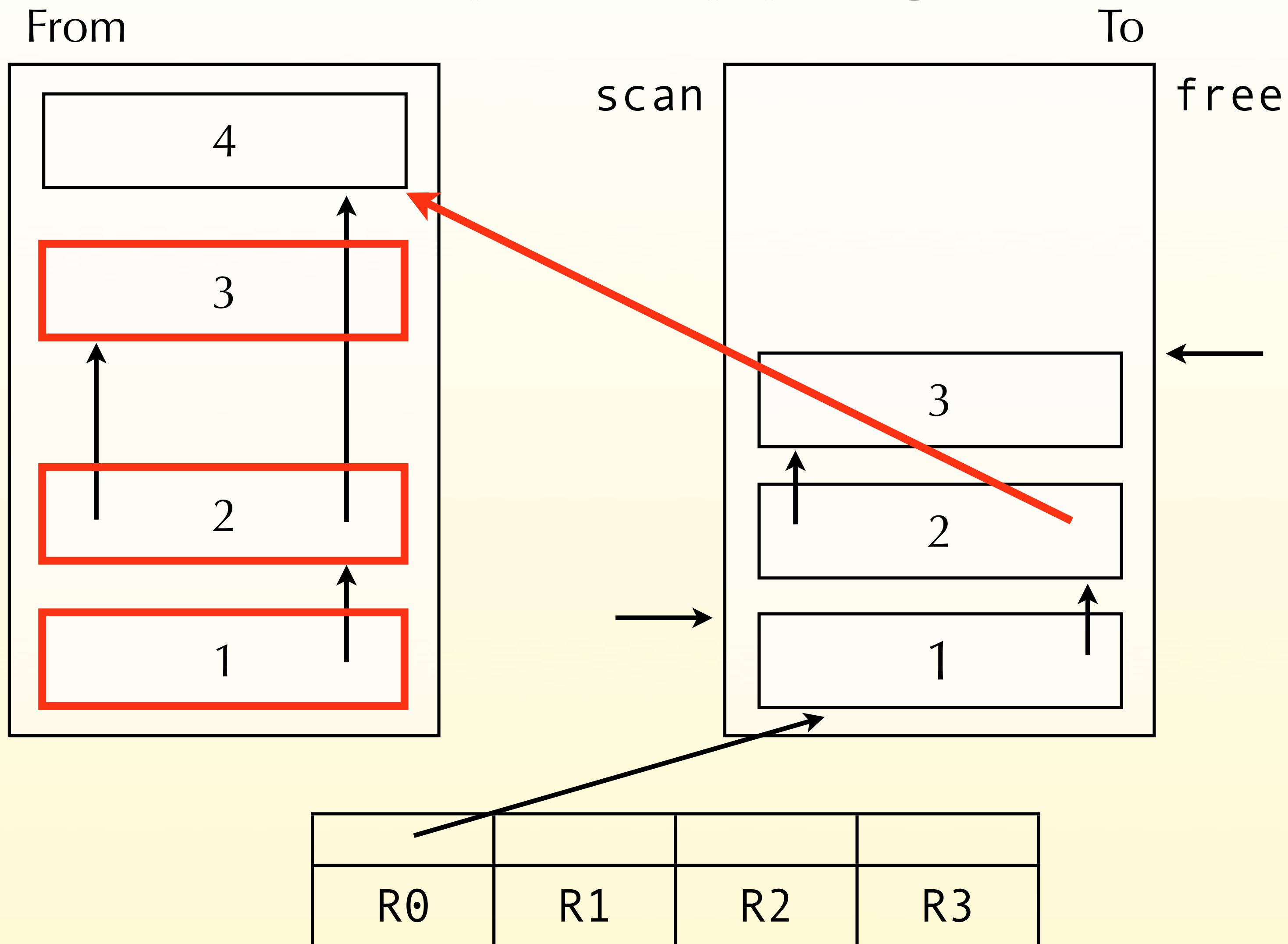
Cheney's copying GC



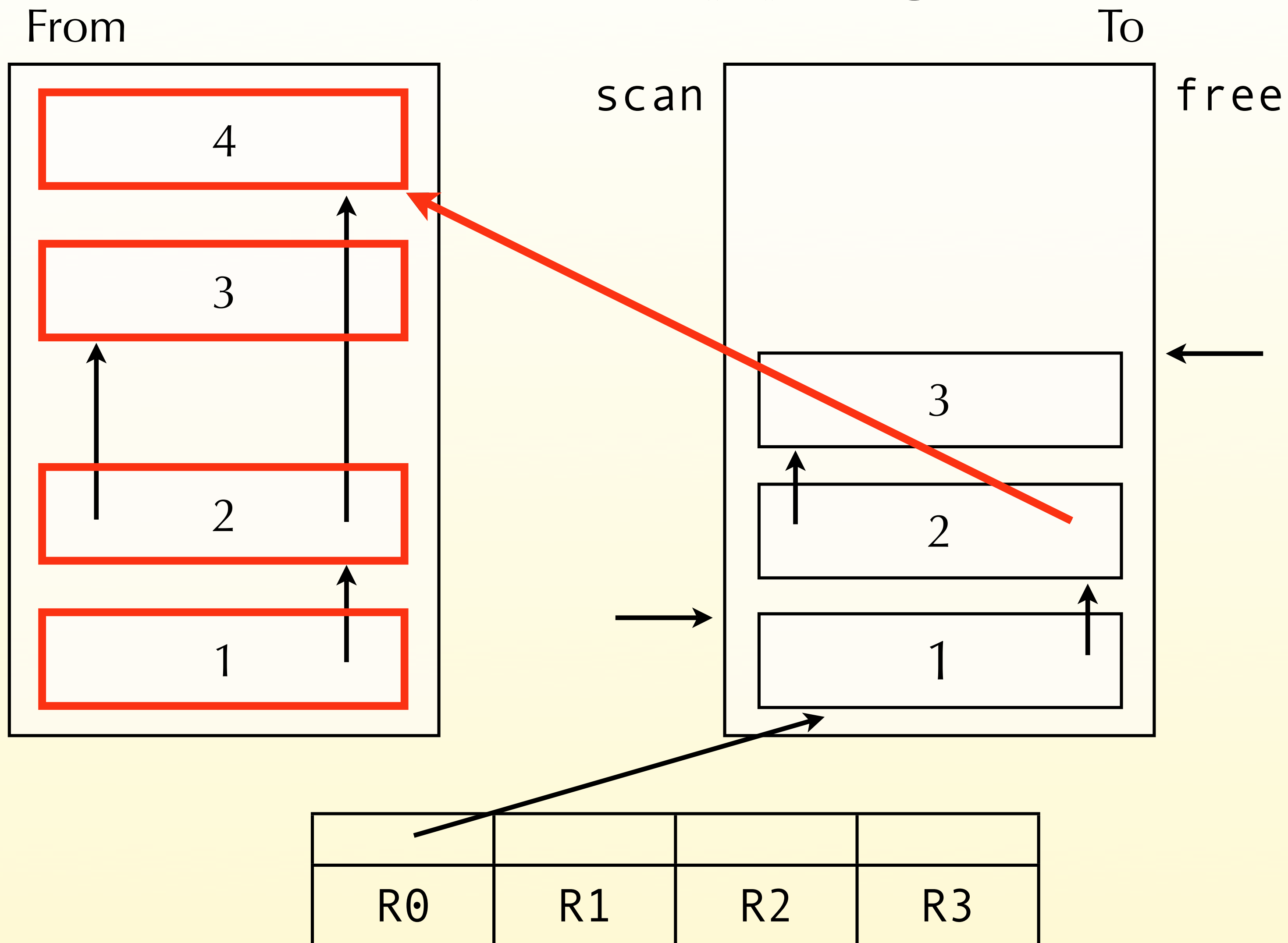
Cheney's copying GC



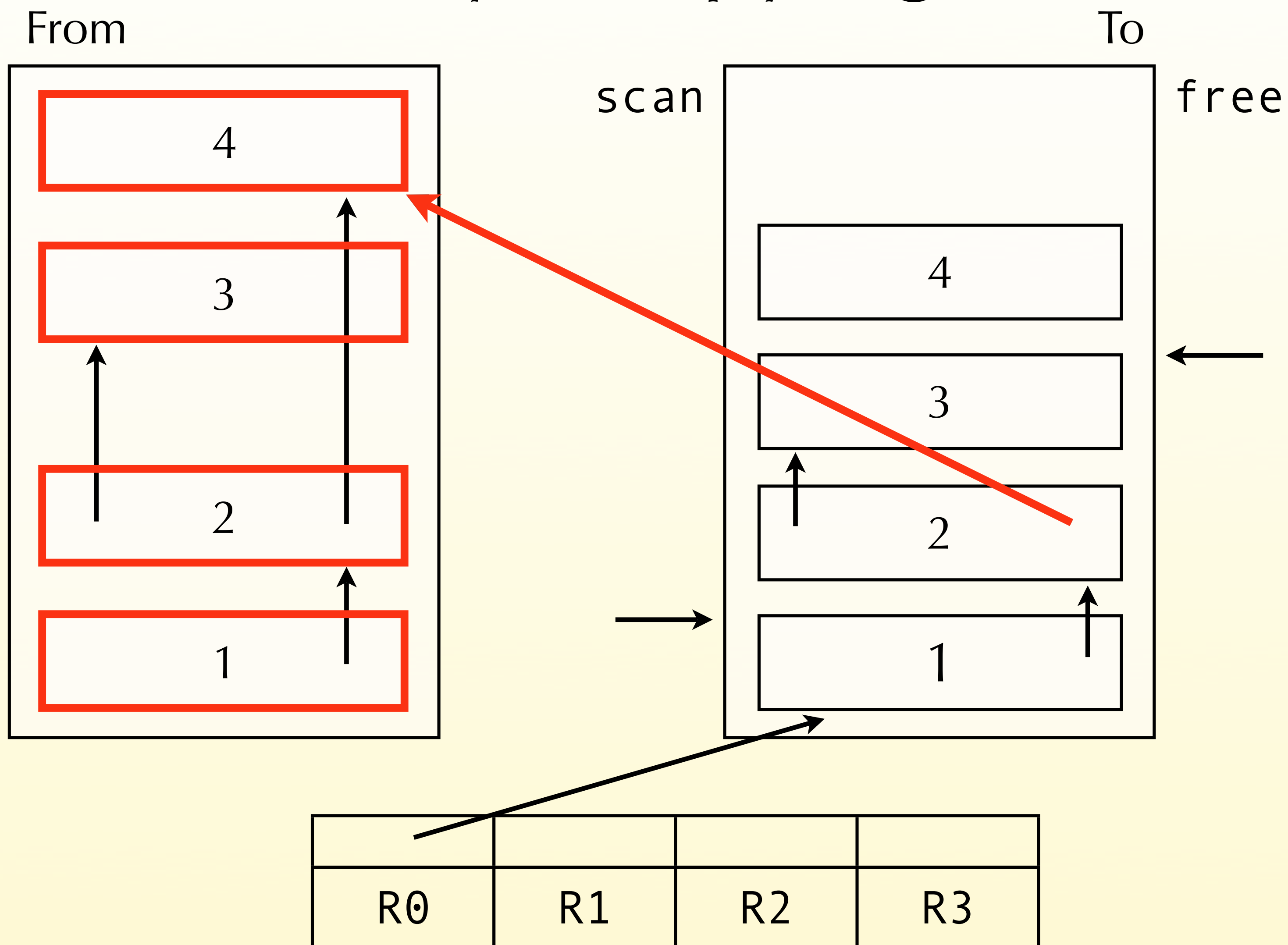
Cheney's copying GC



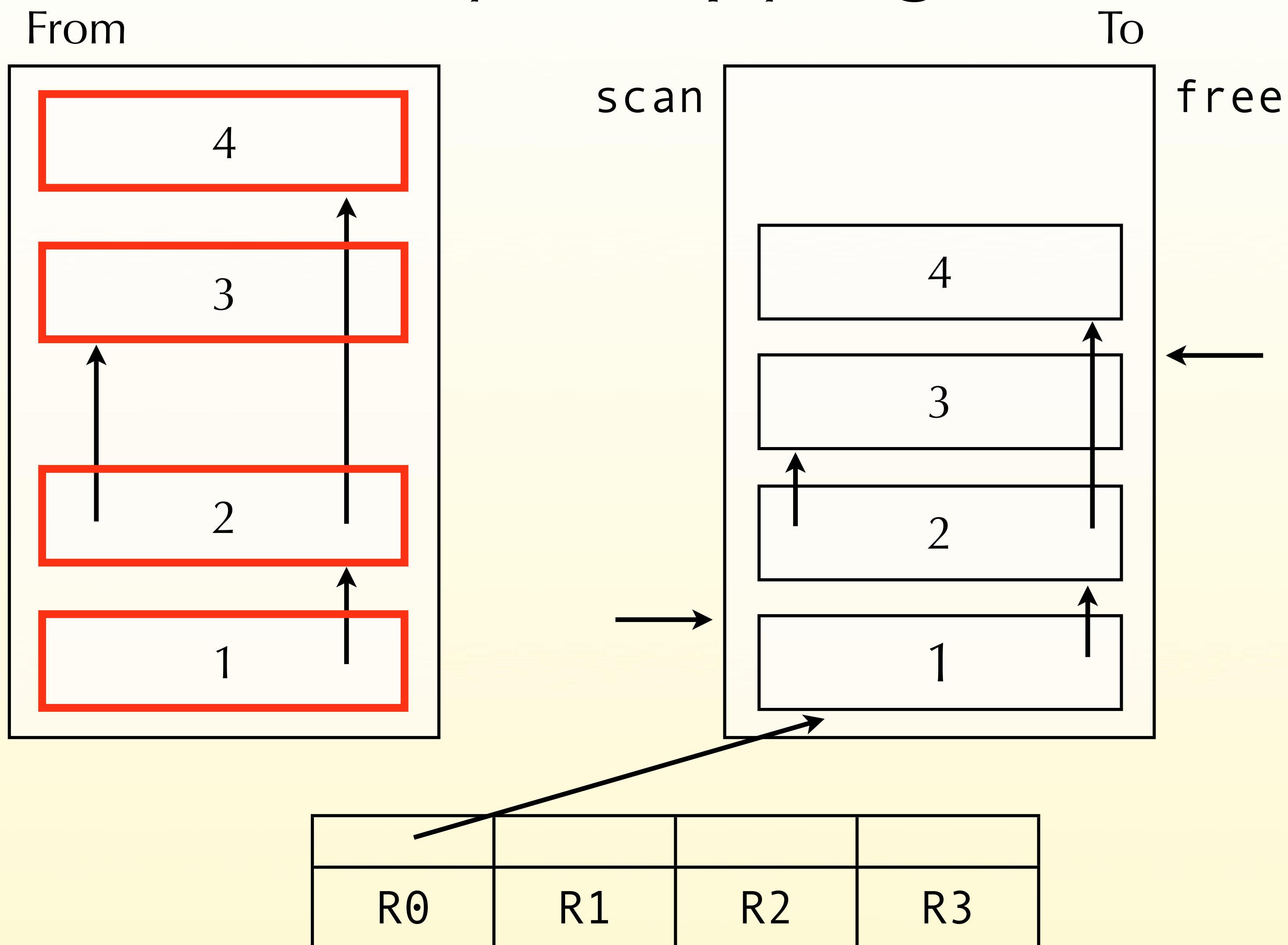
Cheney's copying GC



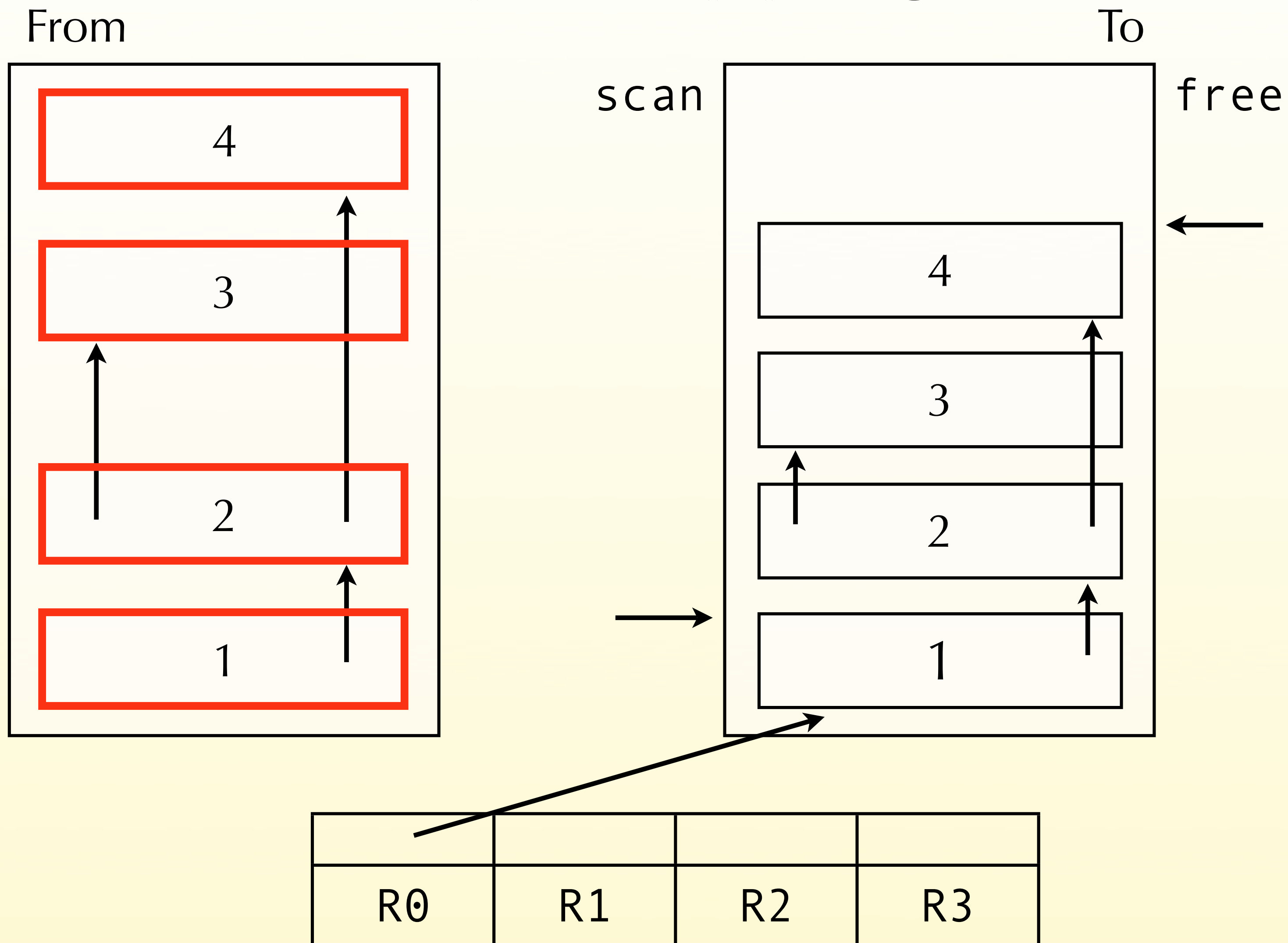
Cheney's copying GC



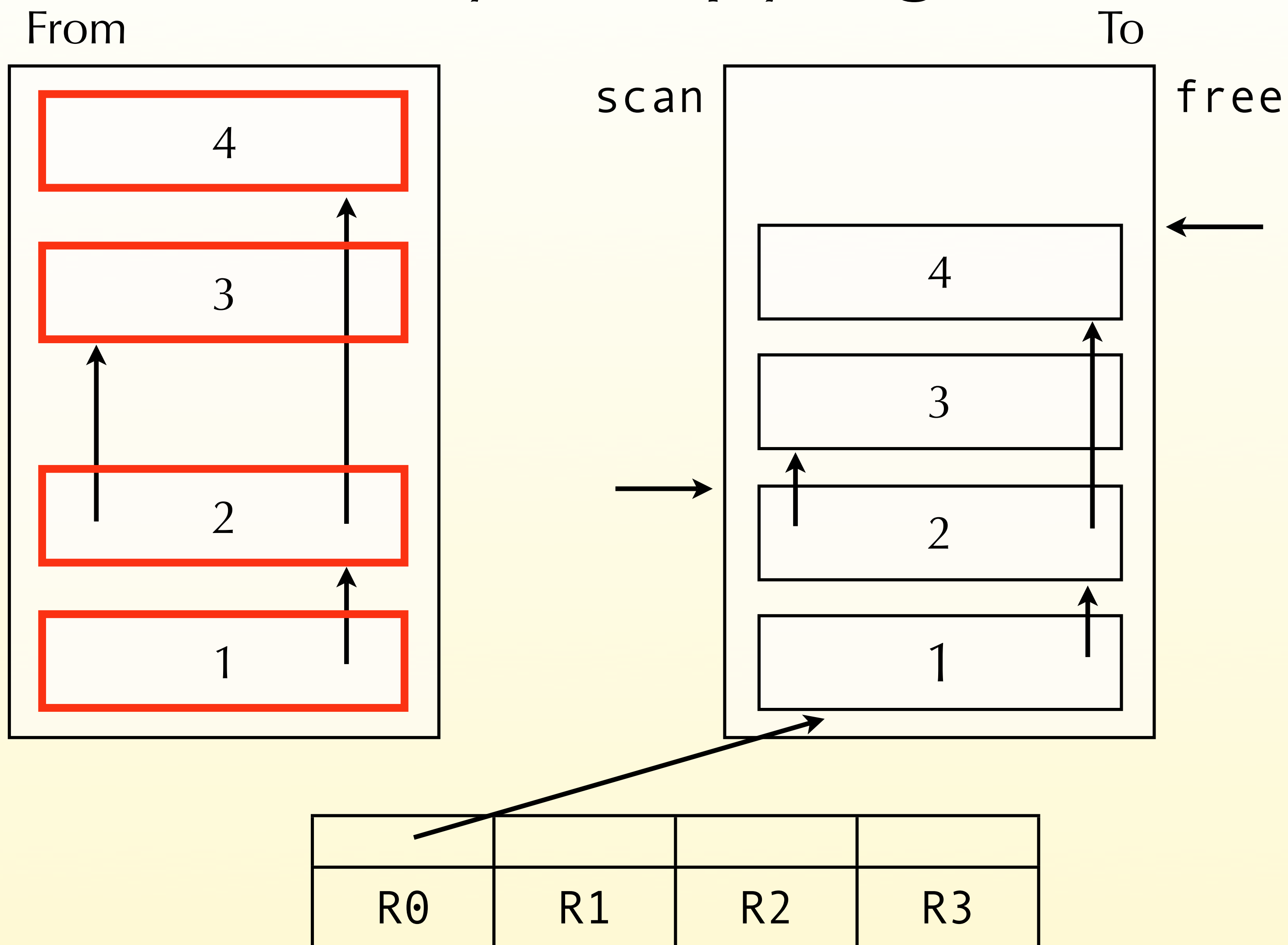
Cheney's copying GC



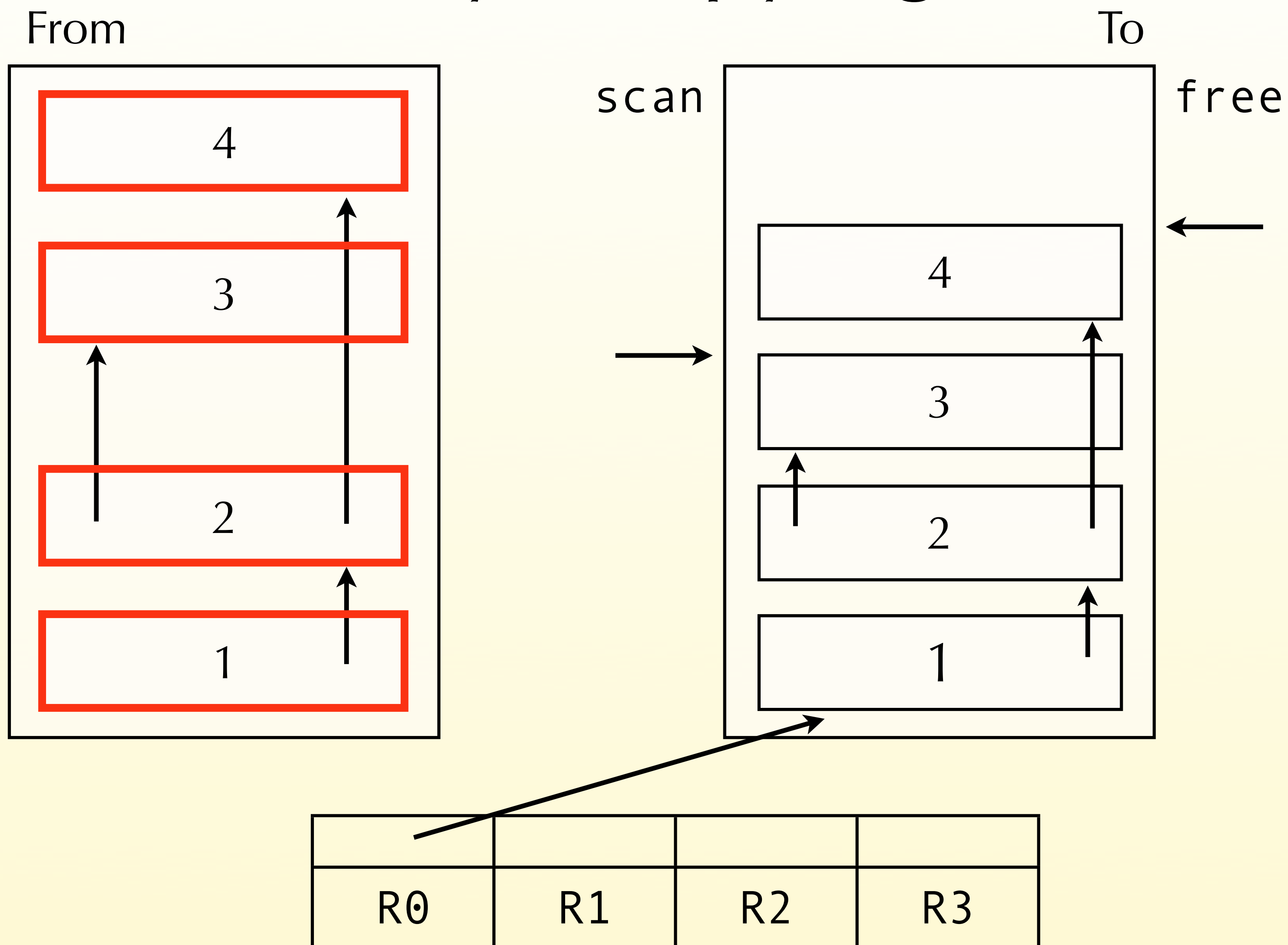
Cheney's copying GC



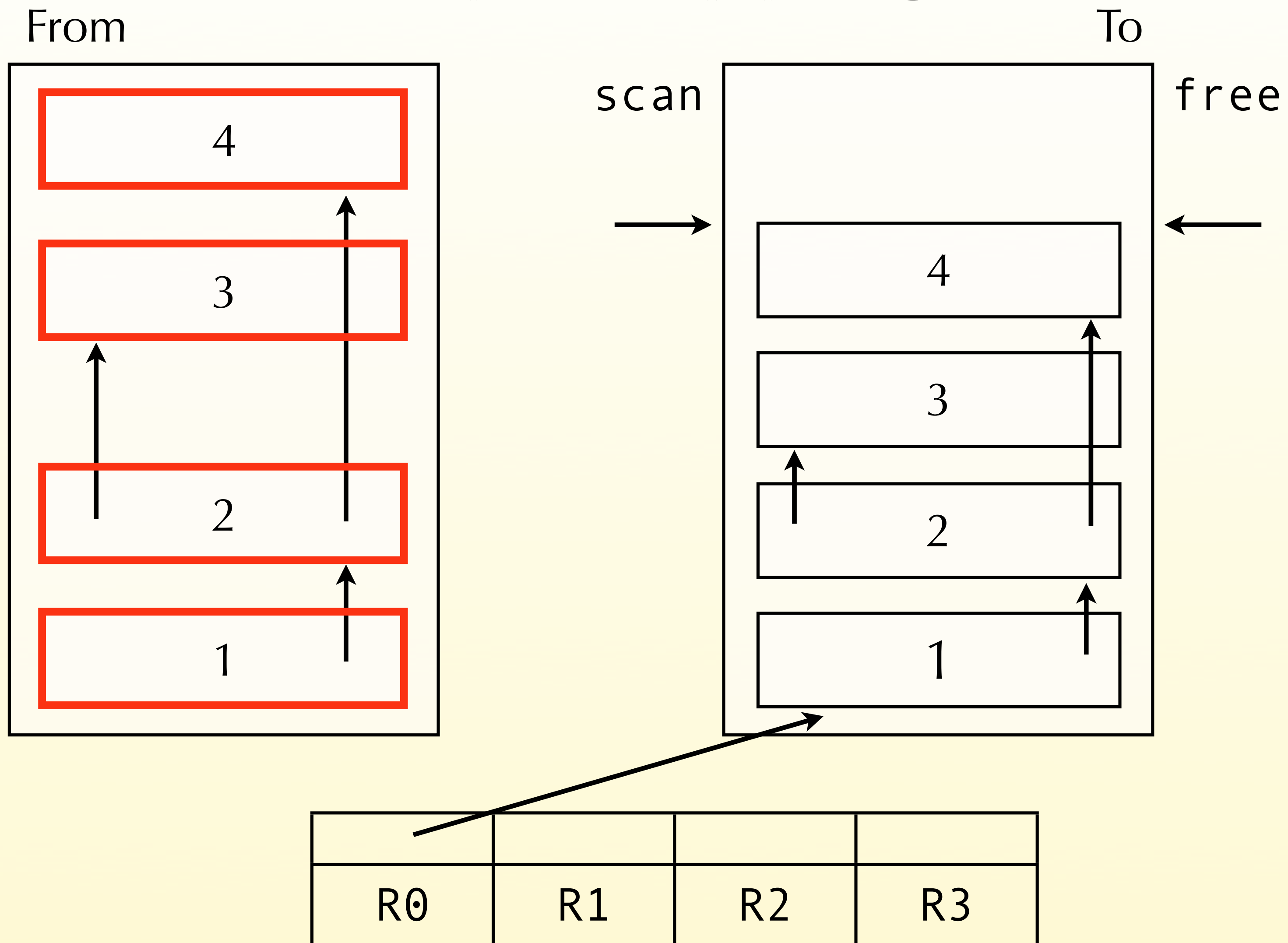
Cheney's copying GC



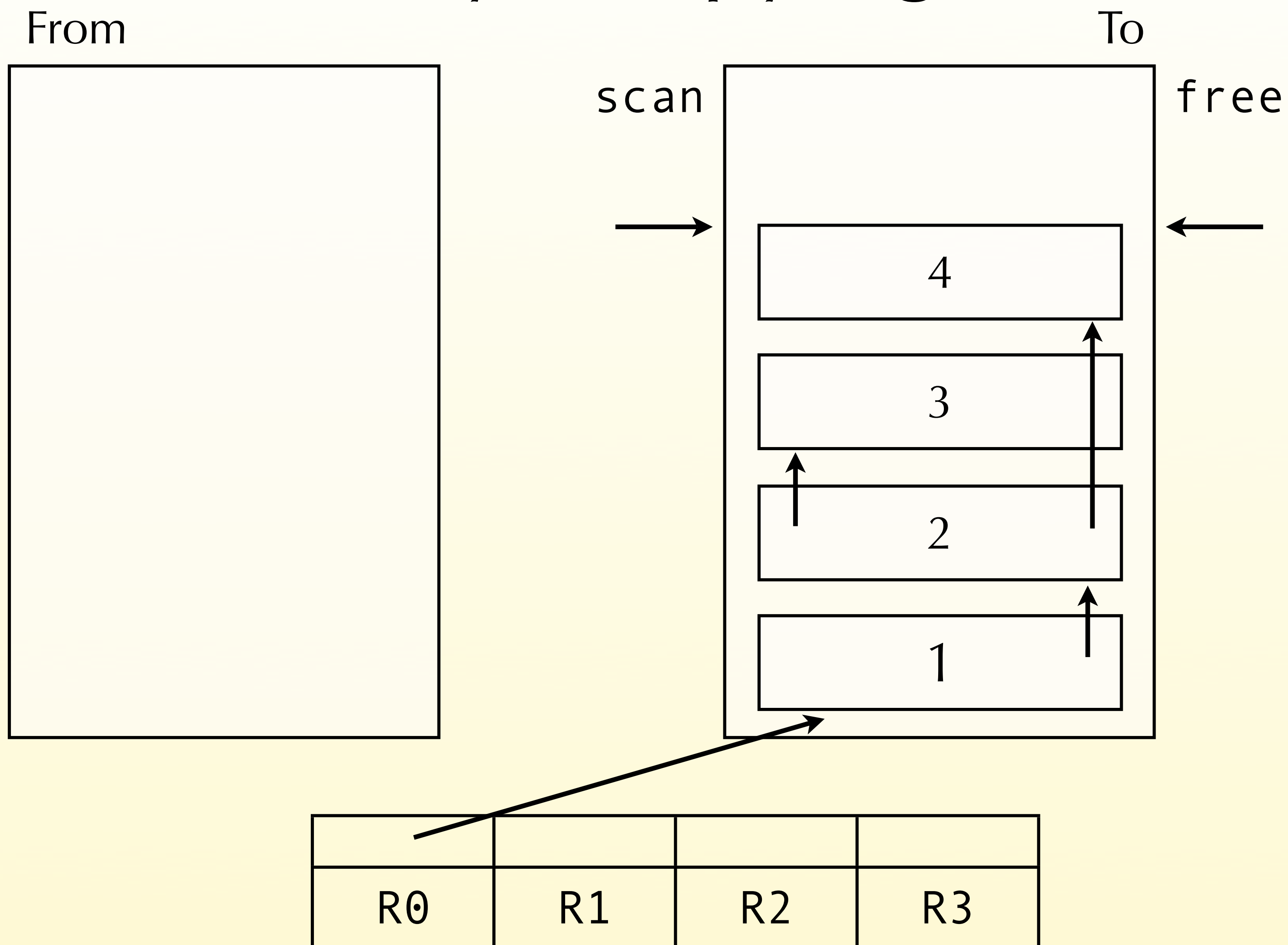
Cheney's copying GC



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

Generational garbage collection

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.

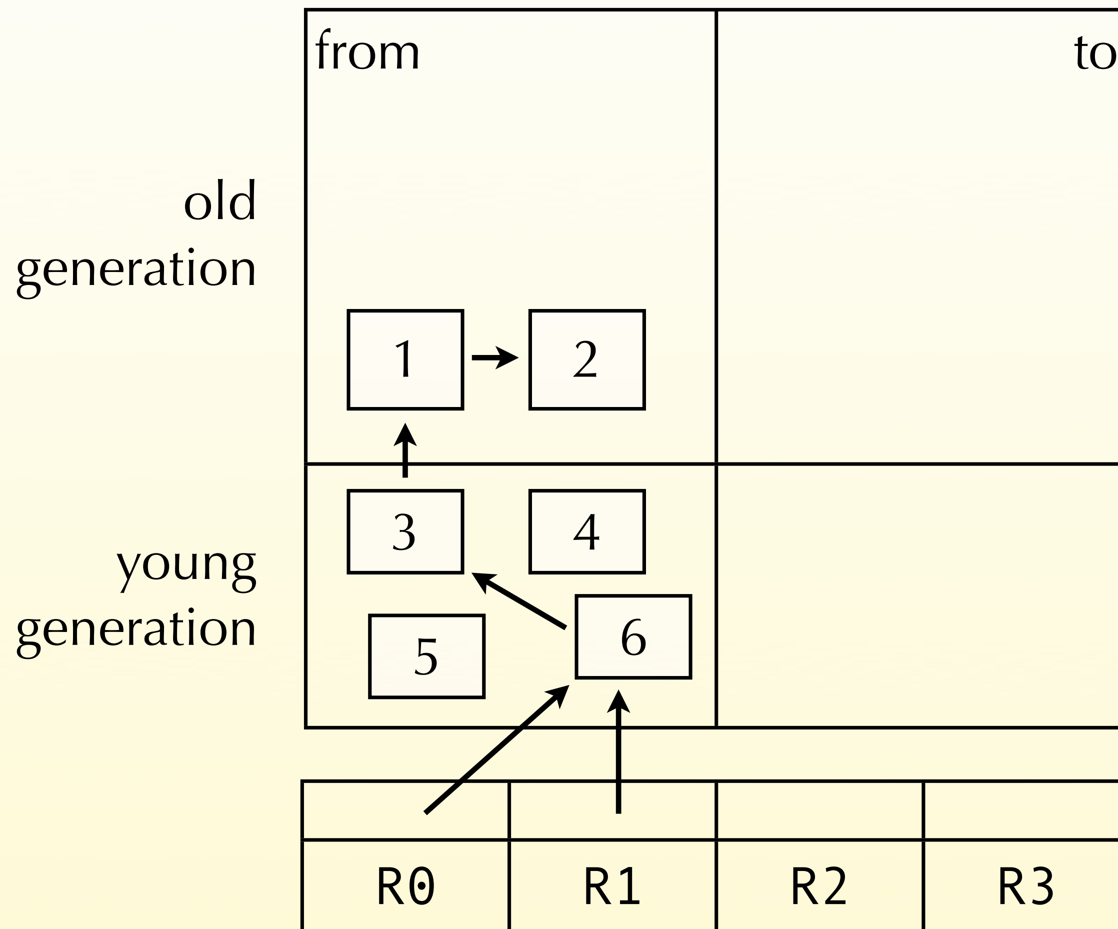
Generational GC

In a generational GC, objects are partitioned into *n* generations – often 2. The younger a generation is, the smaller the amount of memory reserved to it.

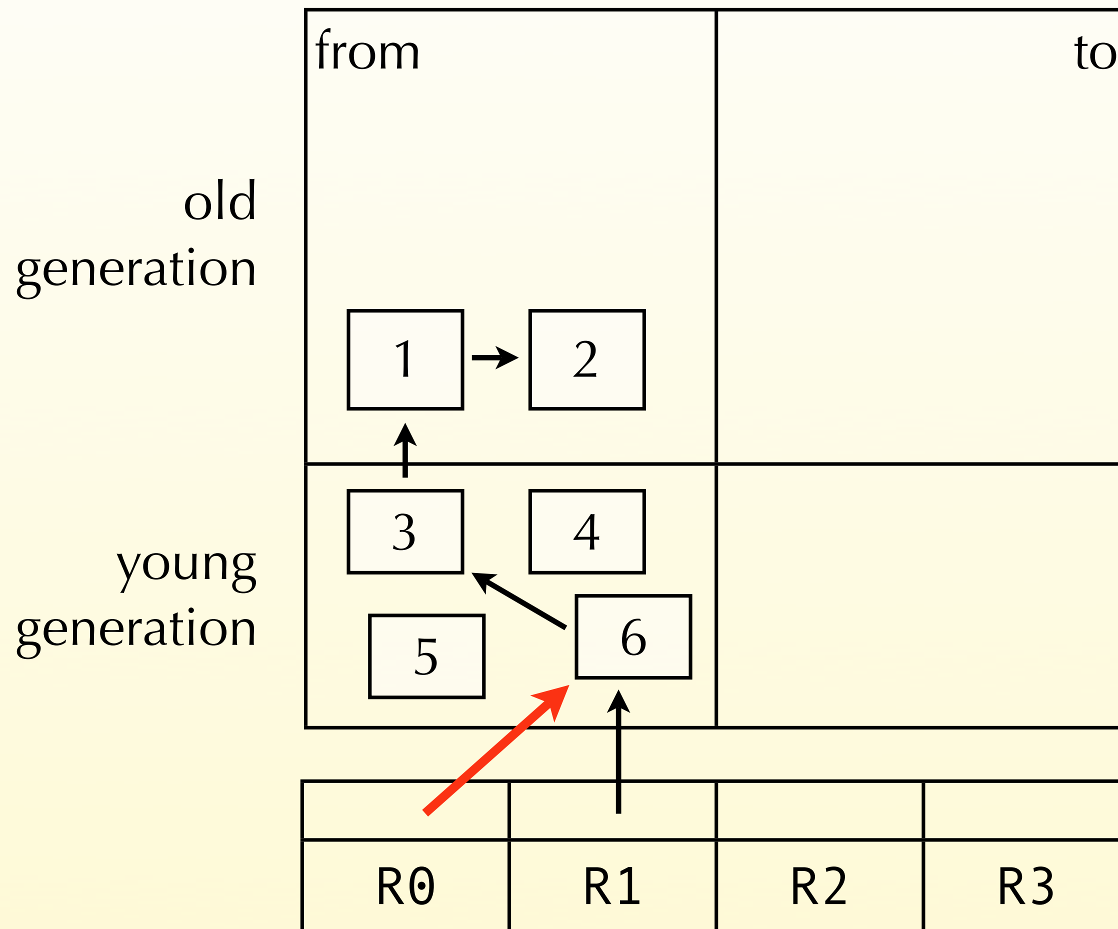
All objects are initially allocated in the youngest generation. When it is full, a **minor collection** is performed, to collect memory in that generation only. Some of the surviving objects are promoted to the next generation, based on a **promotion policy**.

When an older generation is itself full, a **major collection** is performed to collect memory in that generation and all the younger ones.

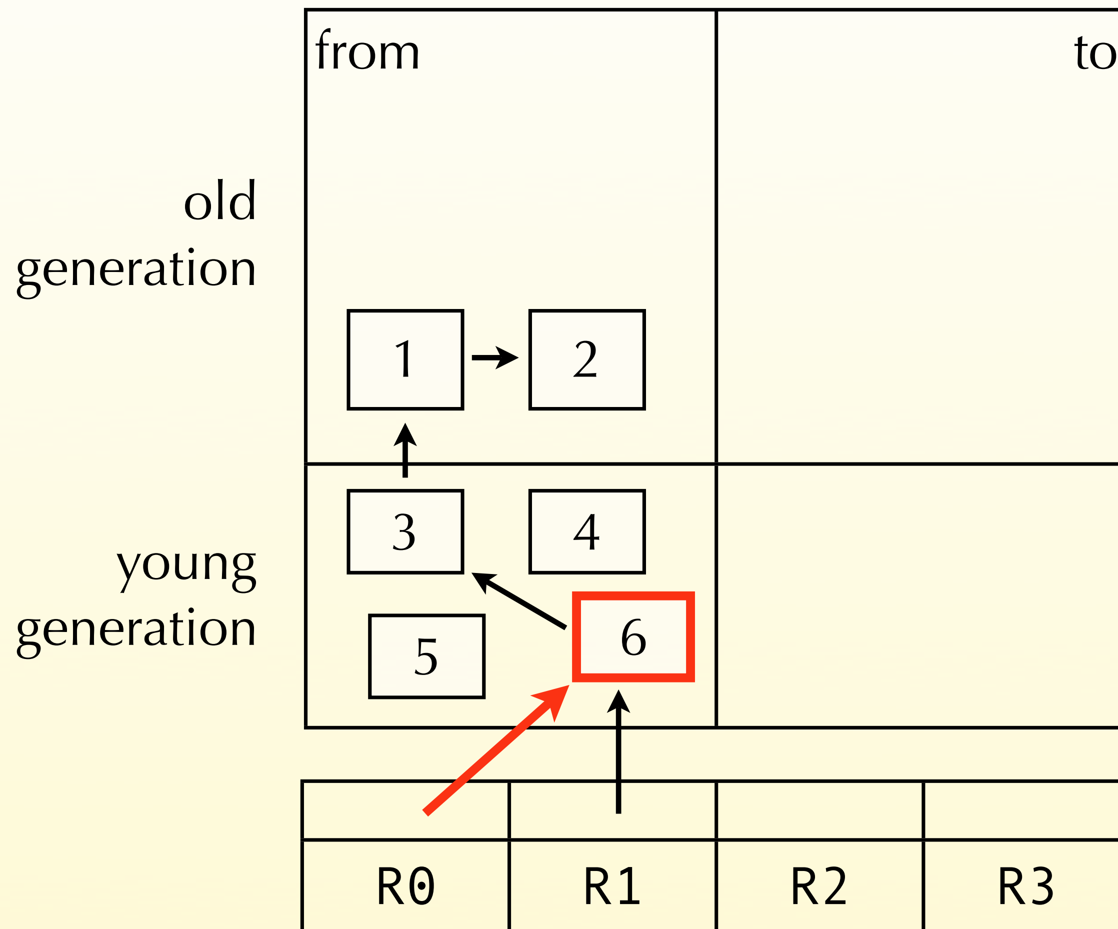
Minor collection example



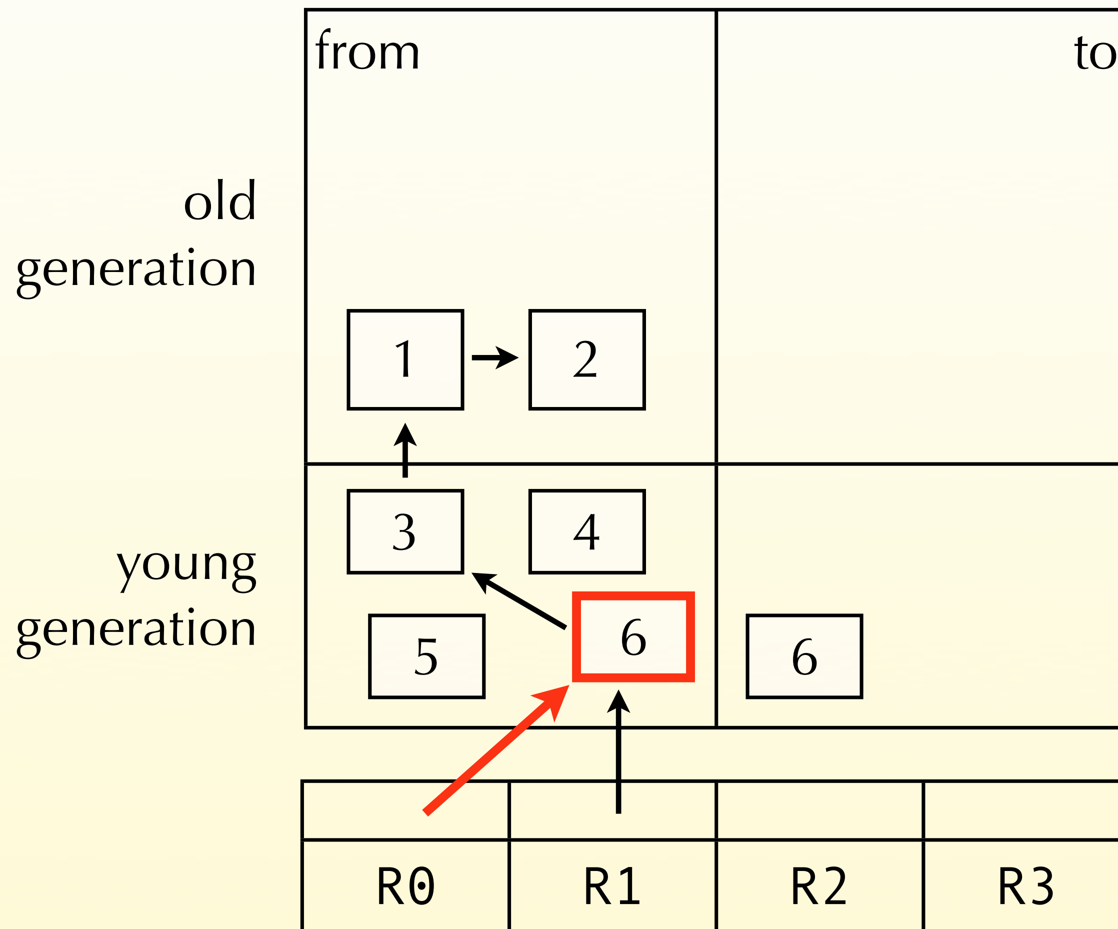
Minor collection example



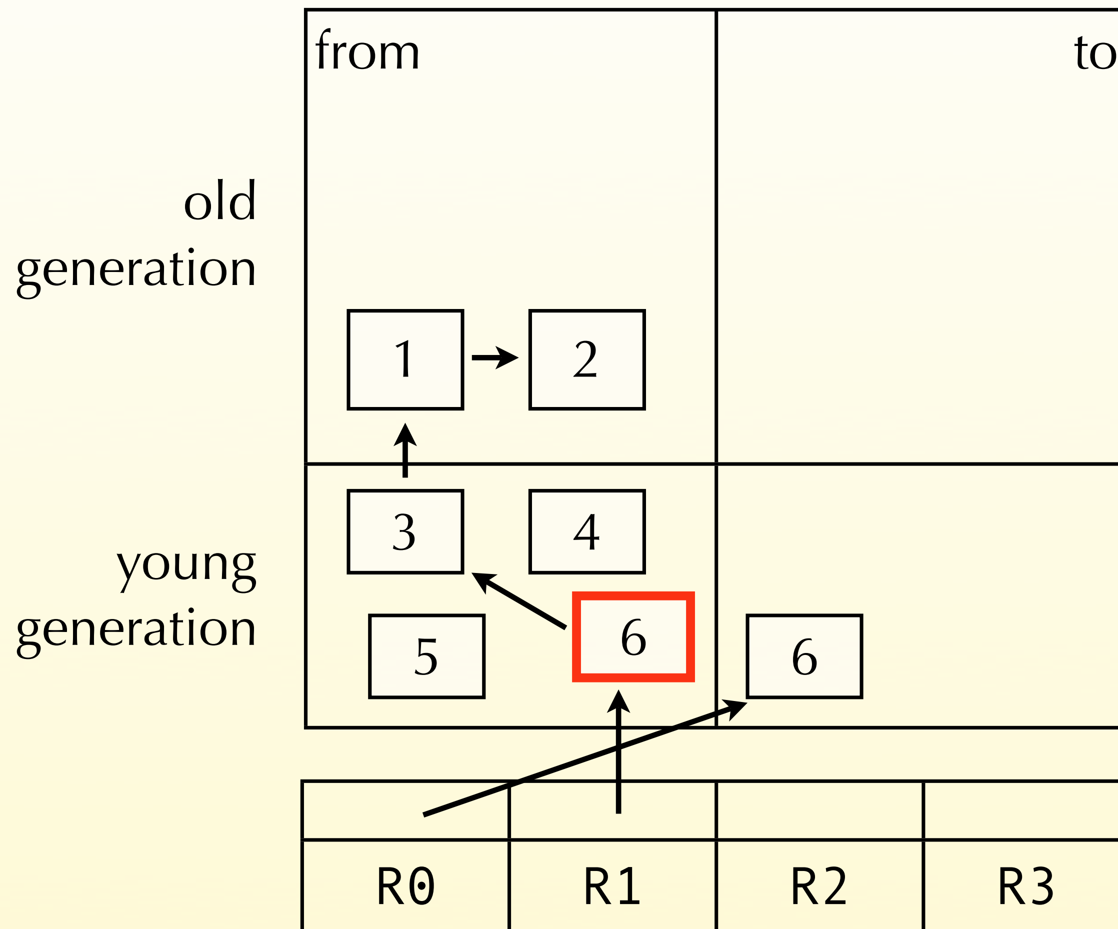
Minor collection example



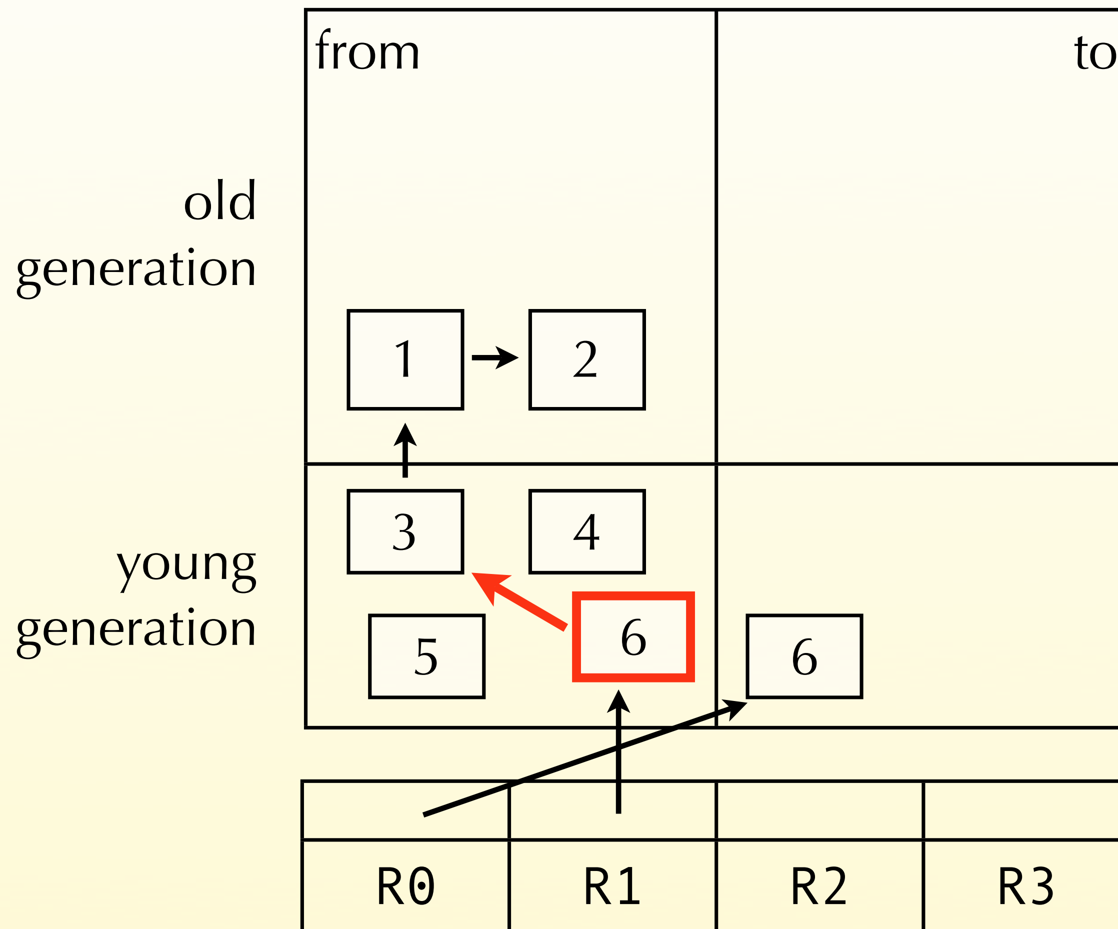
Minor collection example



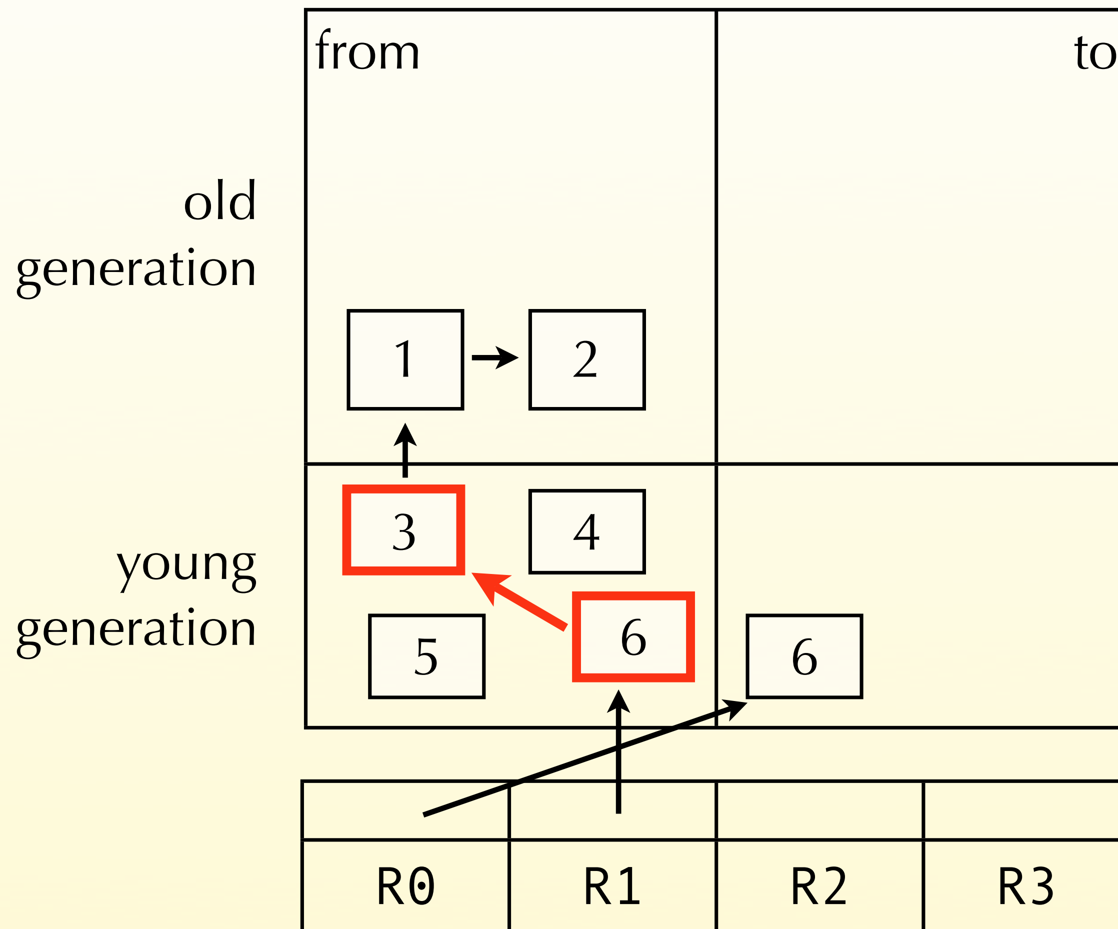
Minor collection example



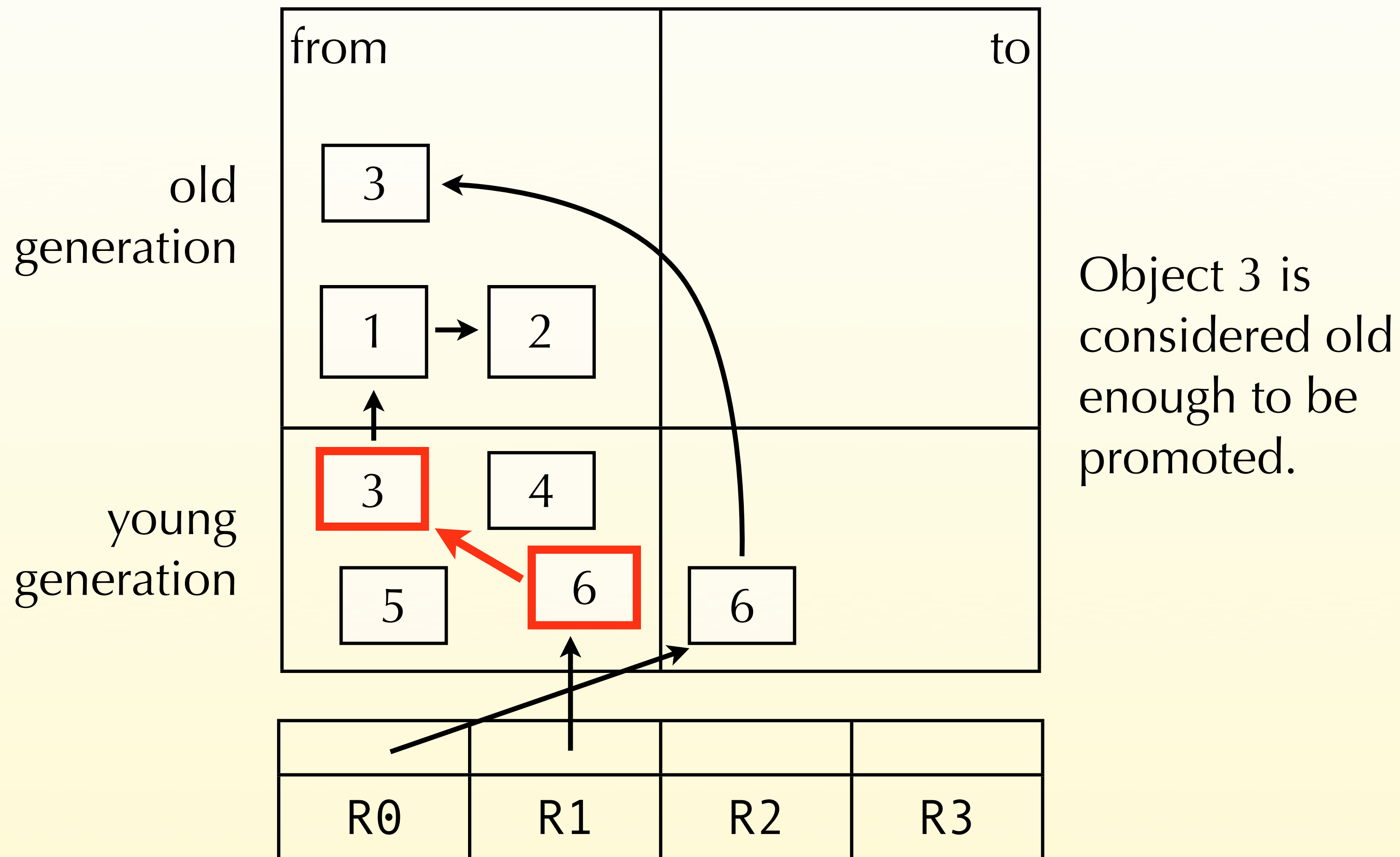
Minor collection example



Minor collection example

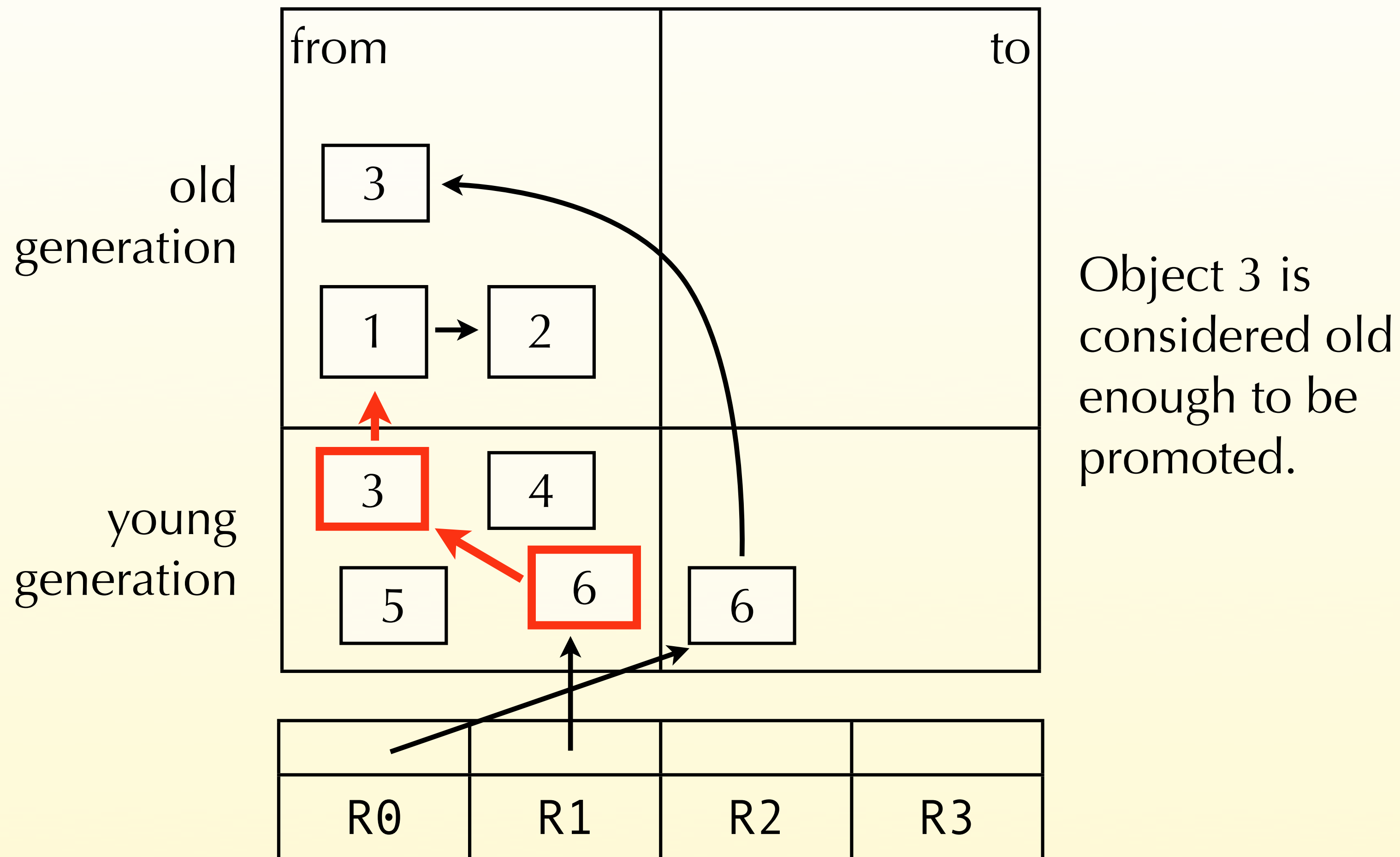


Minor collection example



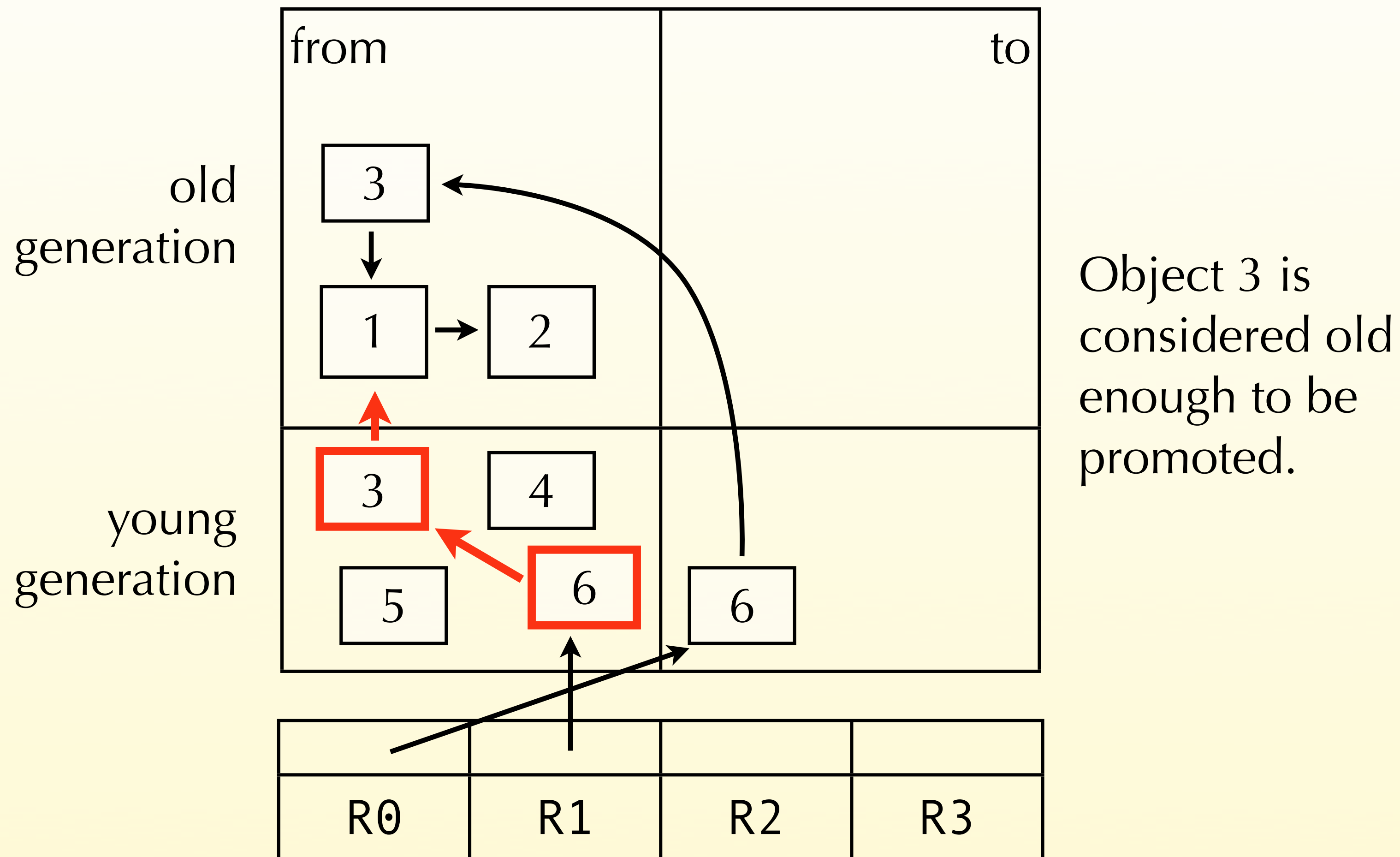
Object 3 is considered old enough to be promoted.

Minor collection example



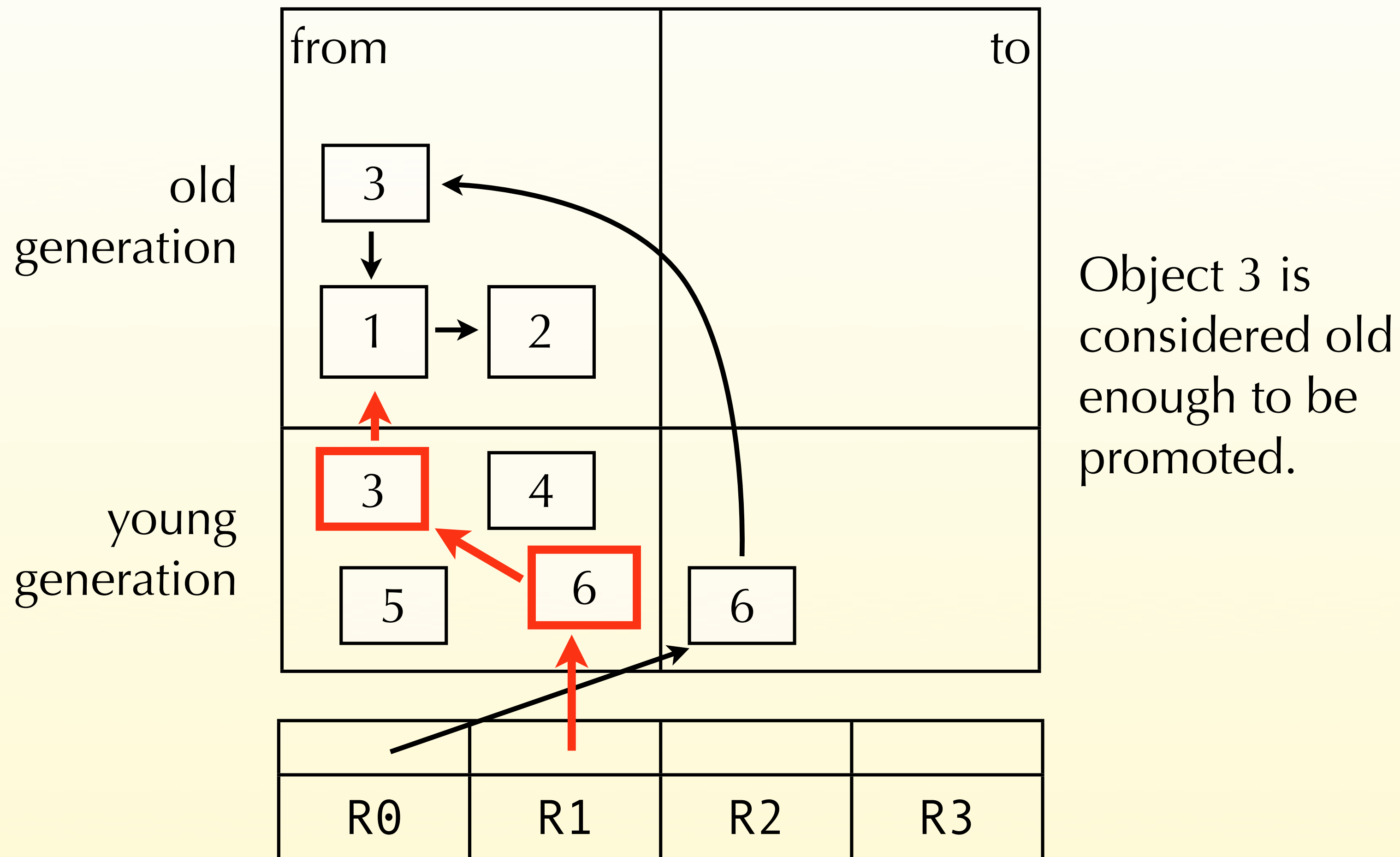
Object 3 is considered old enough to be promoted.

Minor collection example



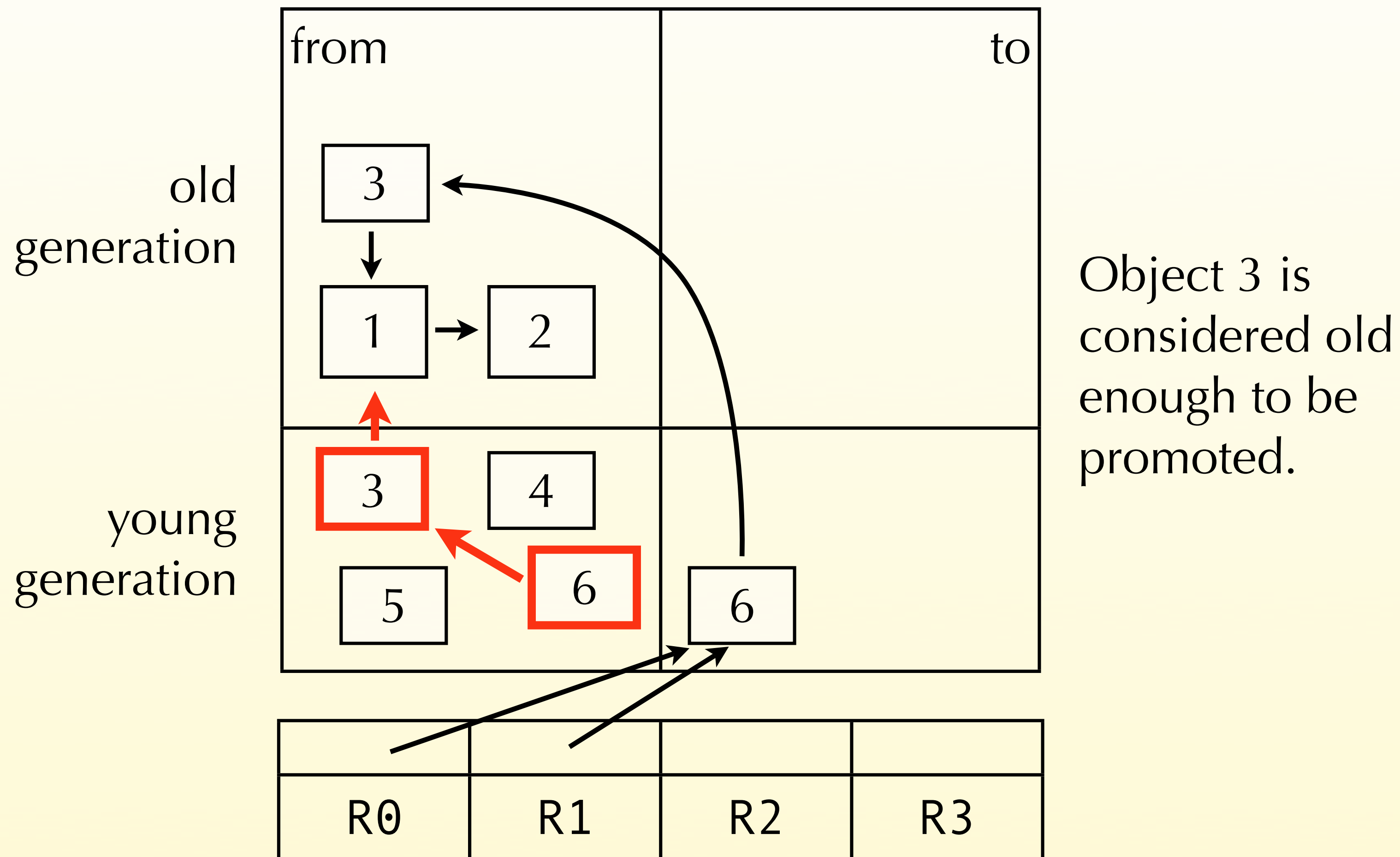
Object 3 is considered old enough to be promoted.

Minor collection example



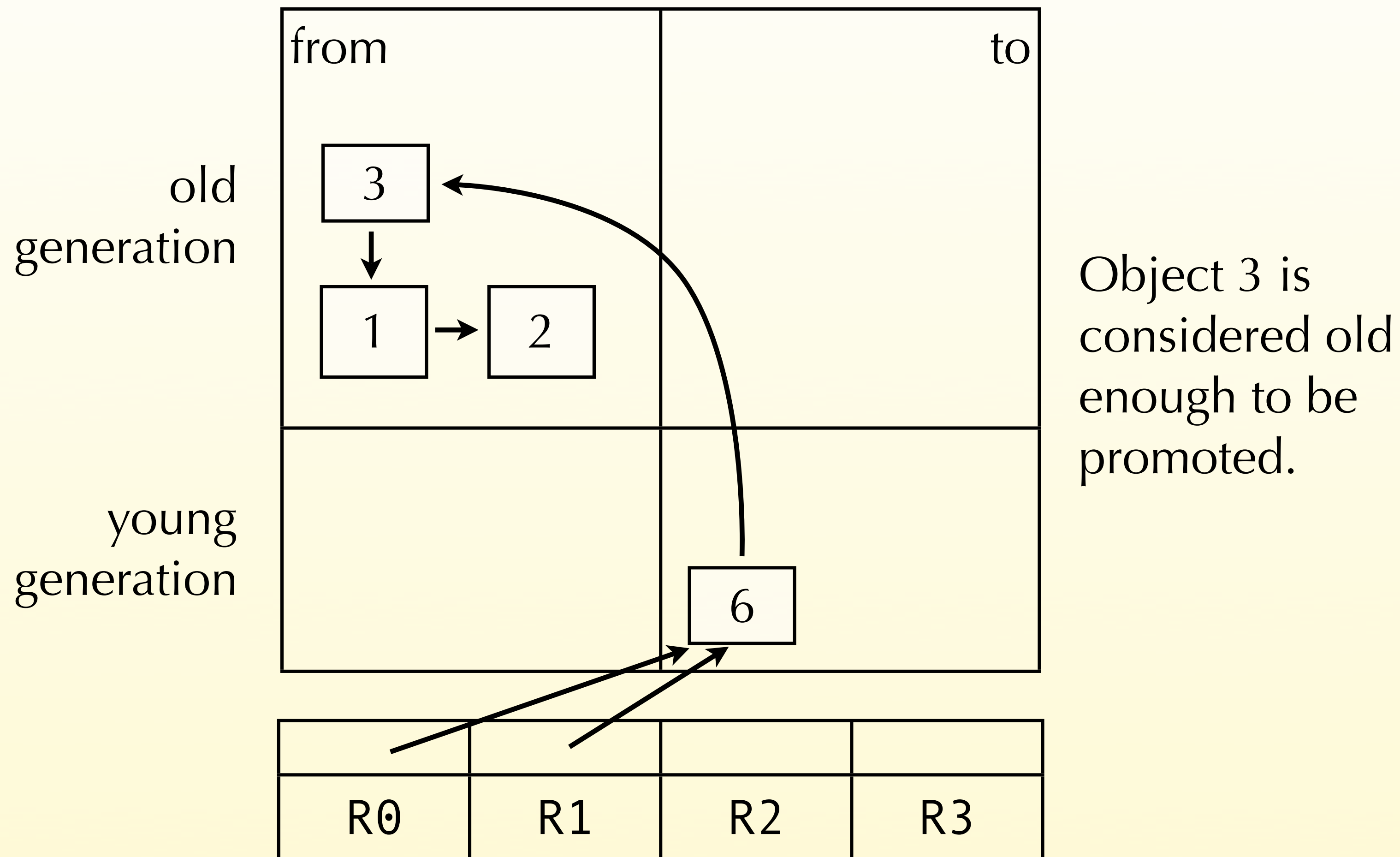
Object 3 is considered old enough to be promoted.

Minor collection example



Object 3 is considered old enough to be promoted.

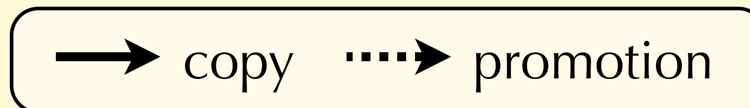
Minor collection example



Object 3 is considered old enough to be promoted.

Heap organisation

one semi-space per generation	next generation as semi-space	separate creation space
old: from → to; young: **from** → to	old: from → to; young: **from**	old: from → to; young: from → to; **Eden**
too much memory wasted…	too many objects promoted…	good!



Hybrid heap organisation

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

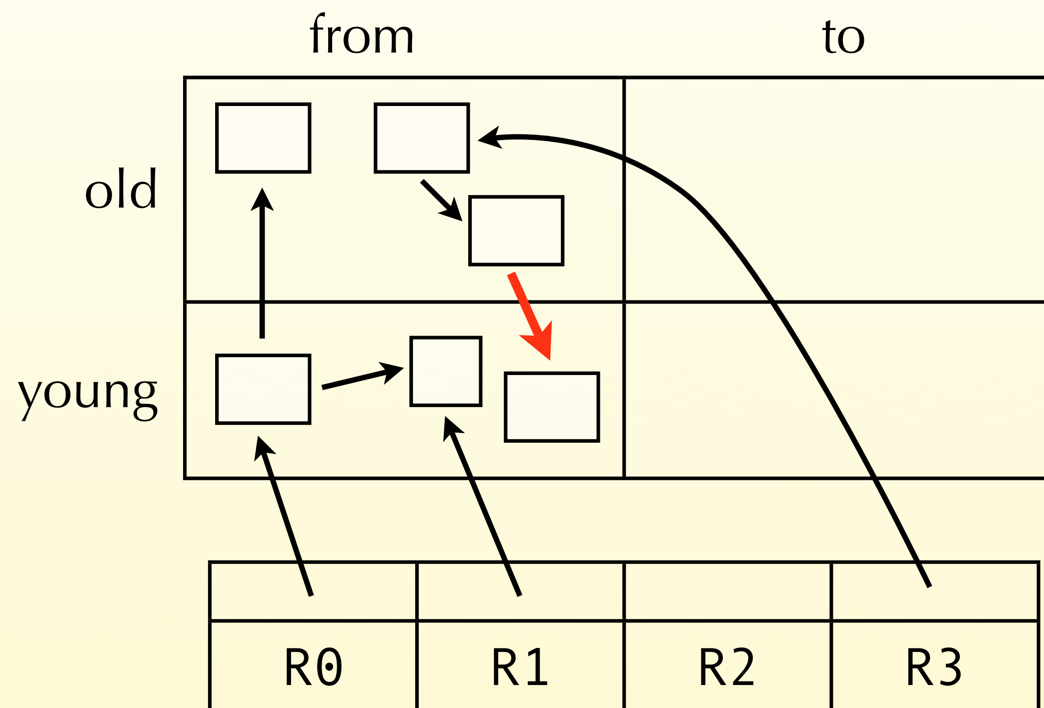
Generational GCs use a **promotion policy** to decide when objects should be advanced to an older generation.

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.

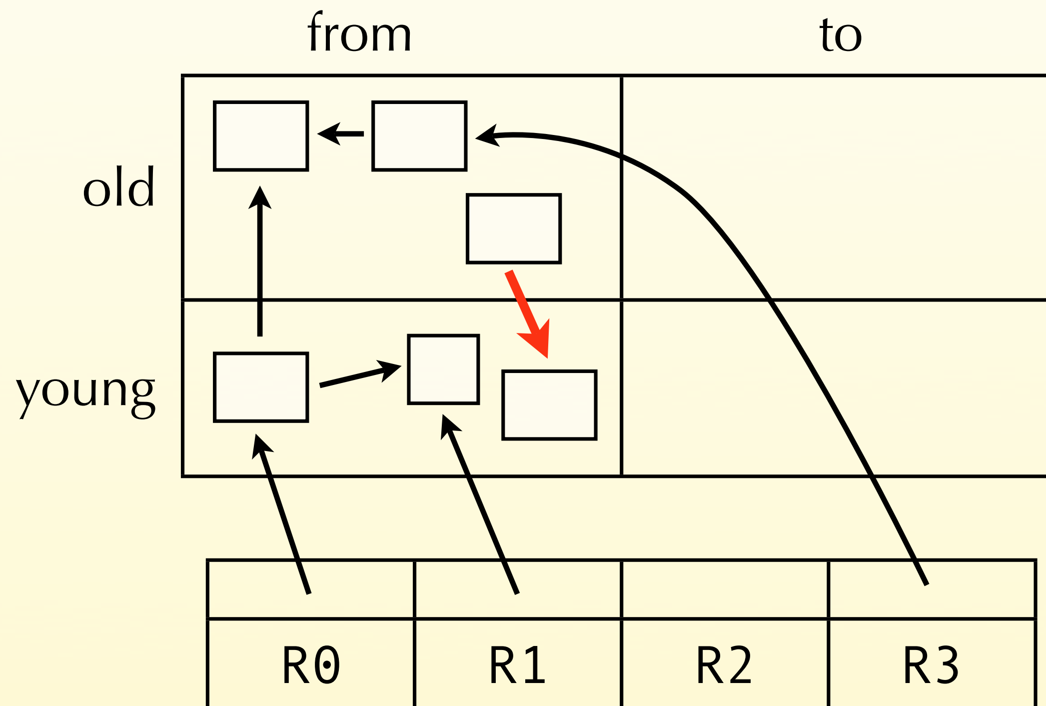
Memory is divided into small, fixed sized areas called cards. A card table remembers, for each card, whether it potentially contains inter-generational pointers.

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 popularisation 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 garbage collector features

Finalisers

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

Finalisers 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 finalisers are invoked, the resource in question should not be scarce.

Finalisers issues

Finalisers are tricky for a number of reasons:

1. what do we do if a finaliser makes the finalised object reachable again – *e.g.* by storing it in a global variable?
2. how do finalisers interact with concurrency – *e.g.* in which thread are they run?
3. how can they be implemented efficiently in a copying GC, which doesn't visit dead objects?

Flavours of pointers

When the GC encounters a pointer, it usually treats it as a **strong** pointer, meaning that the referenced object will be considered as reachable and survive the collection.

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

Weak pointers

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

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

Weak pointers are useful to implement caches, canonicalising mappings, etc.

Example: Java references

Java provides several kinds of “non-strong” pointers, which are, from strongest to weakest:

- **soft references**, cleared only when memory is low,
- **weak references**, cleared as early as possible,
- **phantom references**, similar to weak references except that the referenced object is not available – and therefore cannot be resurrected.

Summary

Memory management is an important part of the run time system, especially for languages offering implicit memory deallocation.

Implicit memory deallocation generally uses reachability as a good but conservative approximation of liveness.

Reference counting cannot reclaim cyclic structures while other forms of garbage collection, like mark & sweep, can.

Copying GCs copy reachable objects from one semi-space to the other on every collection. This avoids all fragmentation, and makes allocation very fast.

Generational GCs put young objects in a separate, smaller area, collected more often. This reduces collection pauses, and avoids the repeated copying of long-lived objects.