Generational GC

Generational GC

- Empirical observation:
  - “Younger” objects tend to have short lives
  - “Older” objects tend to stay around for a while longer
- Can measure “youth” by time or by growth rate
- Common Lisp: 50-90% of objects die before they are 10KB old
- Glasgow Haskell: 75-95% die within 10KB
  - No more than 5% survive beyond 1MB
- Standard ML of NJ reclaims over 98% of objects of any given generation during a collection
- C: one study showed that over 1/2 of the heap was garbage within 10KB and less than 10% lived for longer than 32KB

Incremental vs. generational

- An **incremental** garbage collector is any garbage-collector that can run incrementally
  - It does not collect all unreachable objects during a cycle

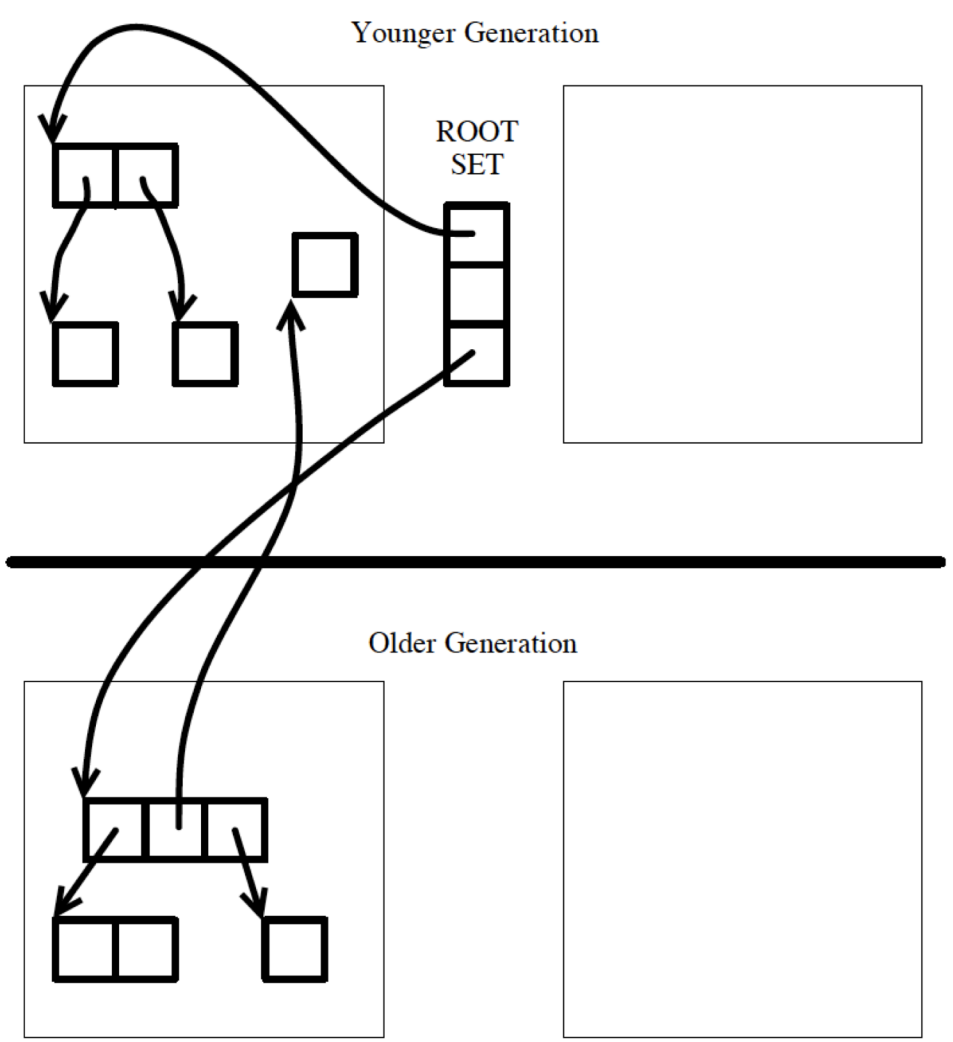
- A **generational** garbage collector differentiates between old, medium and new objects
  - Objects are gathered together in generations.
  - New objects are allocated in the youngest or nursery generation, and promoted to older generations if they survive.
  - Objects in older generations are condemned less frequently, saving CPU time.

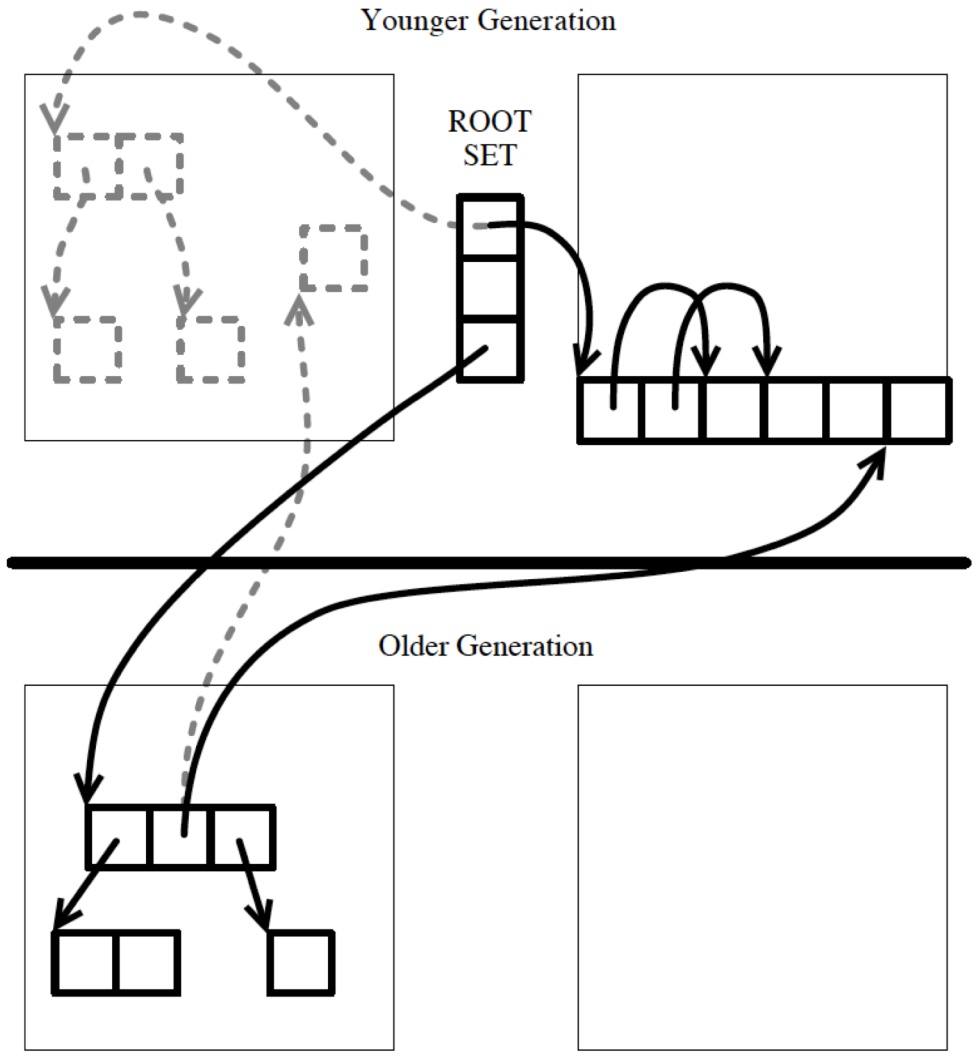
- Most pointers are from younger objects to older objects
  - In Java, pointers go in both directions, but older to younger pointers across many objects are rare
    - less than 1%
  - Most mutations among young objects
    - 92 to 98% of pointer mutations

Generational GC pseudocode

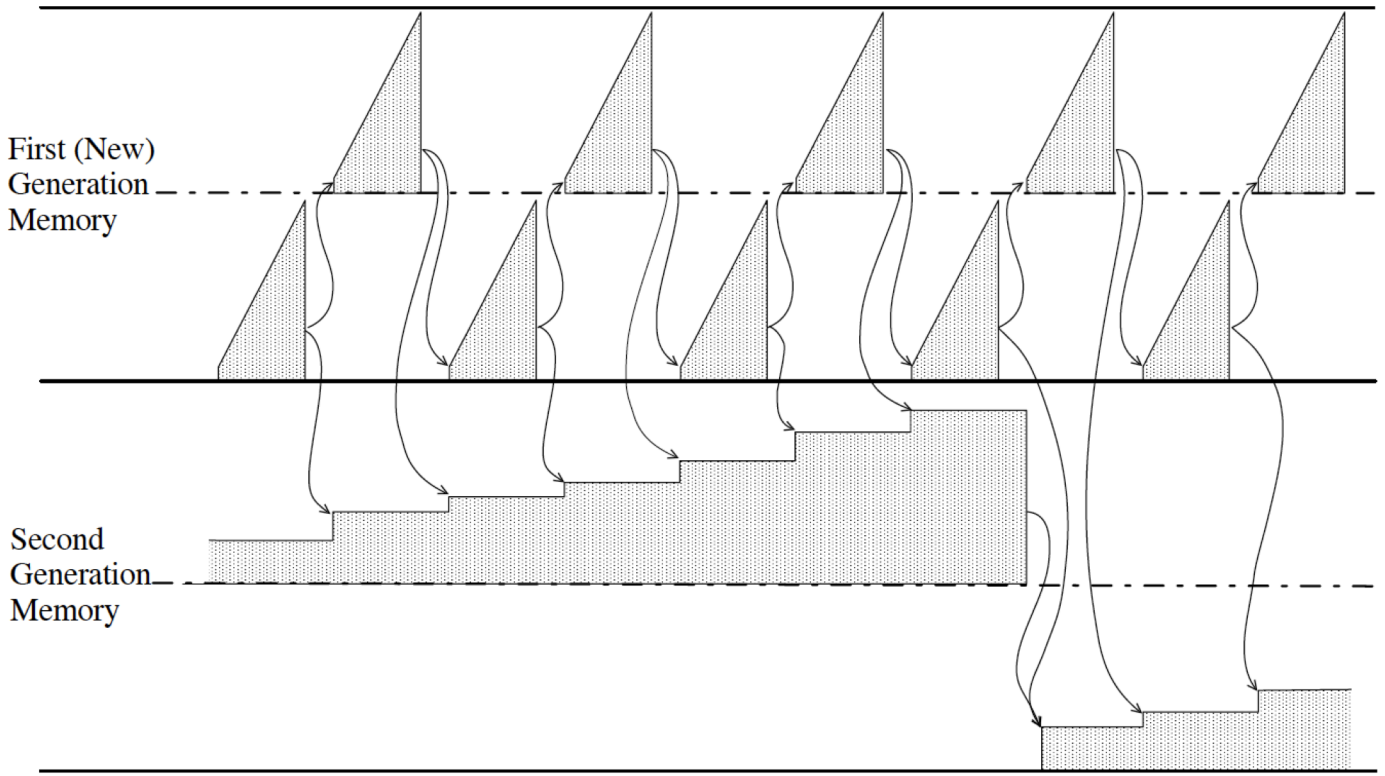
- **Divide the heap in to two spaces: young and old**
- Allocate in to the young space
- When the young space fills up,
  - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through n-1

Generational collectors





Generational collectors



Collect and Promote

- If you promote objects as soon as they survive a GC phase of a younger generation, you don't have to have separate semispaces for that generation — just copy to the older space during collection

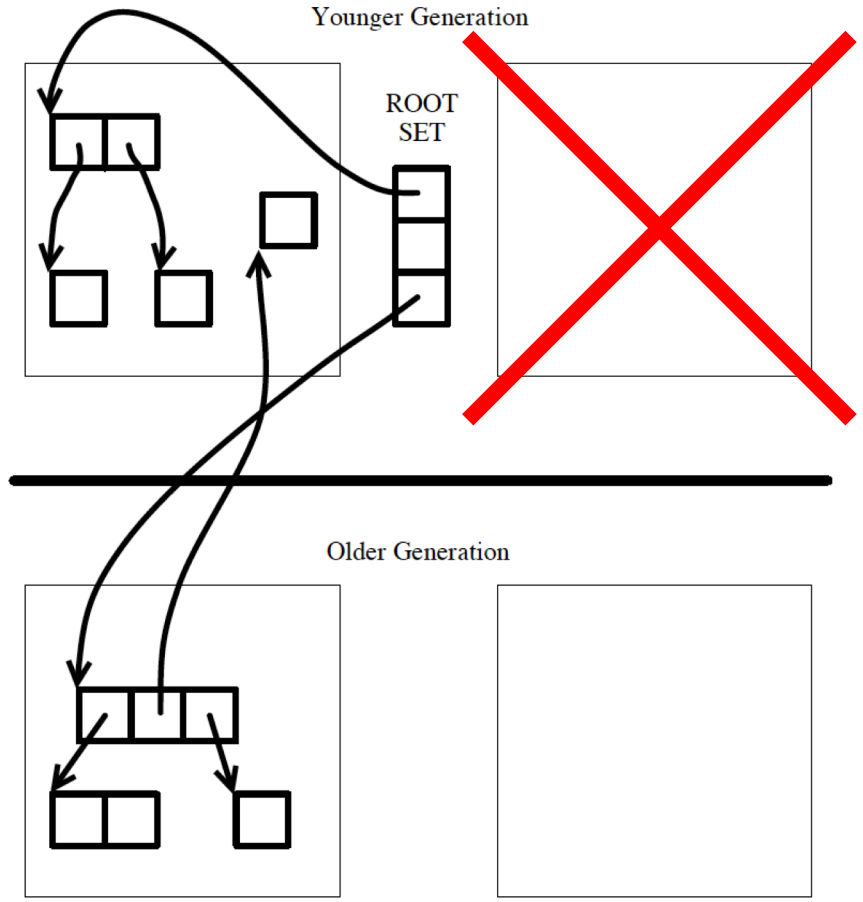
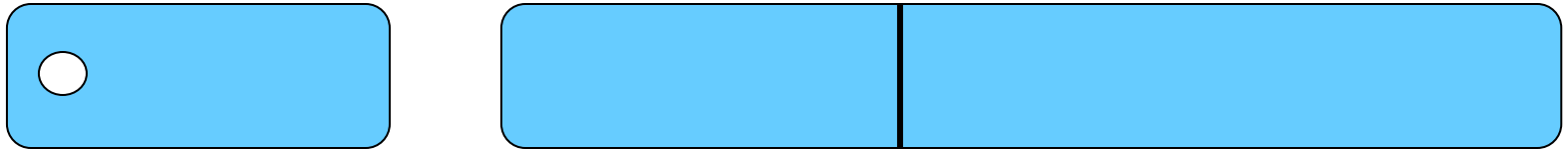
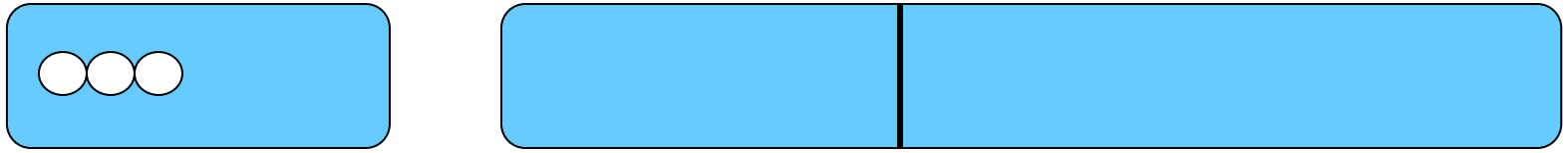
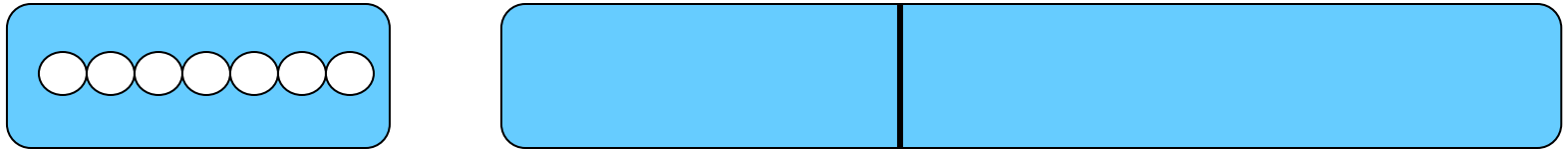


Figure 9: A generational copying garbage collector before garbage collection.

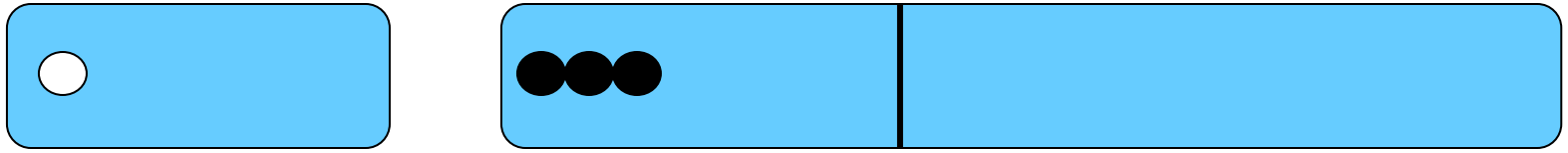

to space
to space
from space
Young
Old


to space
to space
from space
Young
Old


to space
to space
from space
Young
Old

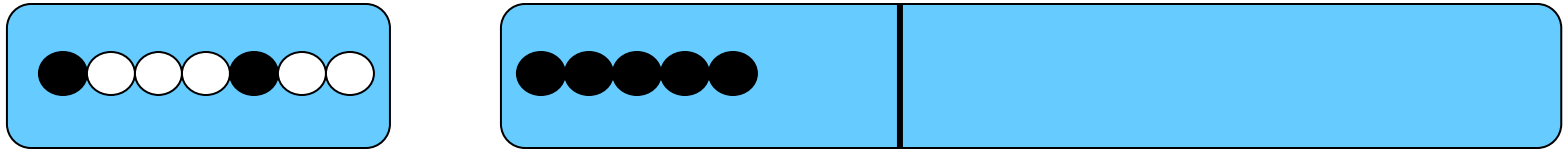

to space
to space
from space
Young
Old

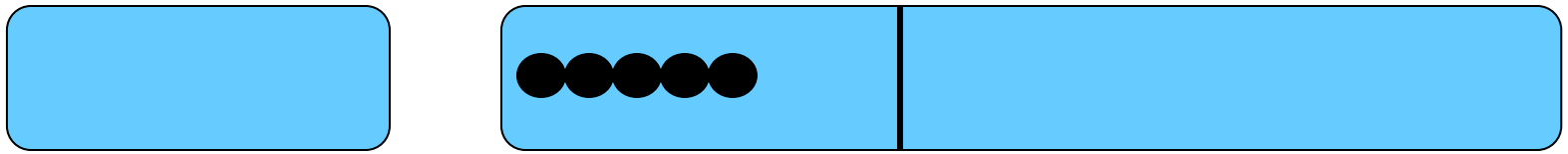

to space
to space
from space
Young
Old

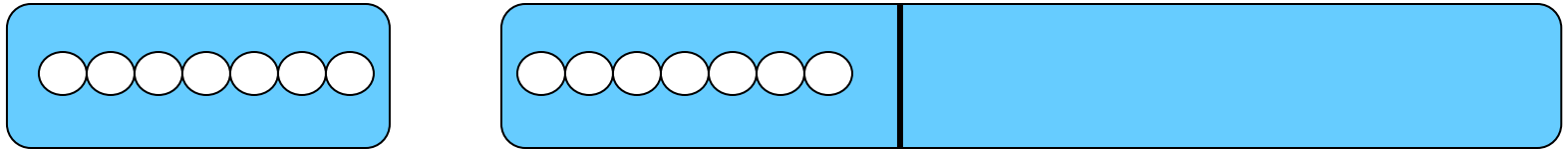

to space
to space
from space
Young
Old

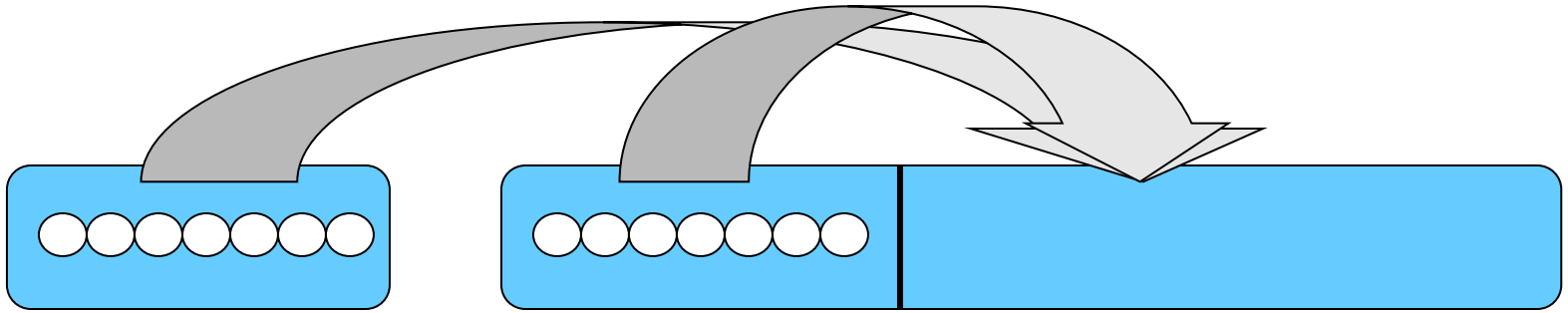

to space
to space
from space
Young
Old

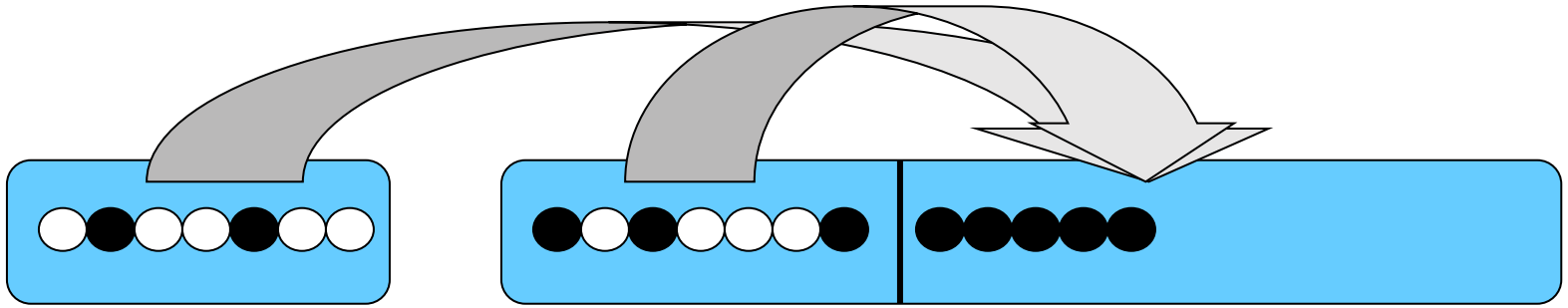

to space
to space
from space
Young
Old

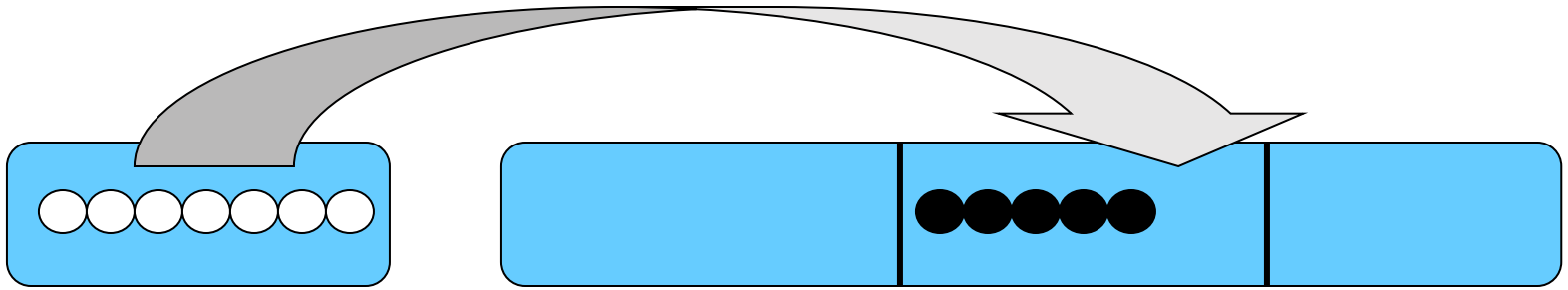

to space
to space
from space
Young
Old


to space
to space
from space
Young
Old


to space
to space
from space
Young
Old


to space
from space
to space
Young
Old


to space
from space
to space
Young
Old

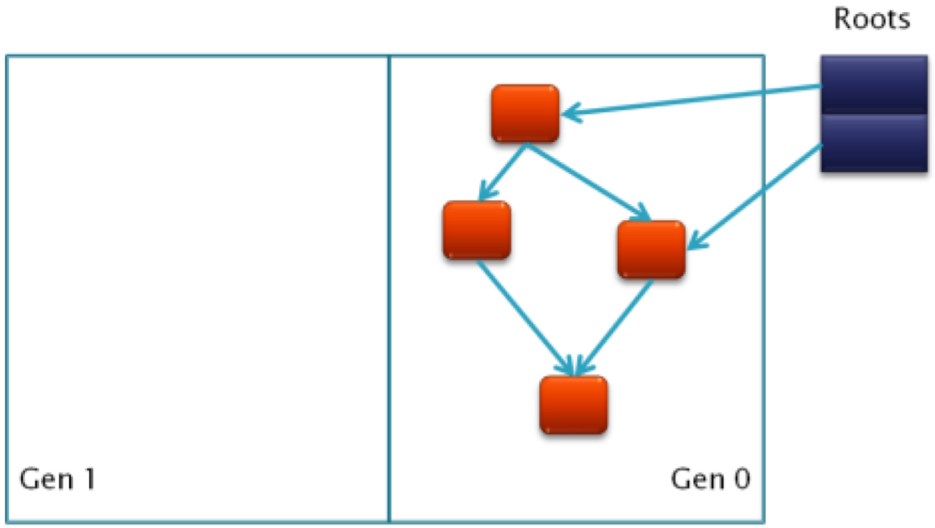

to space
from space
to space
Young
Old

Case Study #1: .NET

GC in .NET is based on the following observations:

- Most objects die young
- Over 90% garbage collected in a GC is newly created post the previous GC cycle
- If an object survives a GC cycle the chances of it becoming garbage in the short term is low and hence the GC wastes time marking it again and again in each cycle

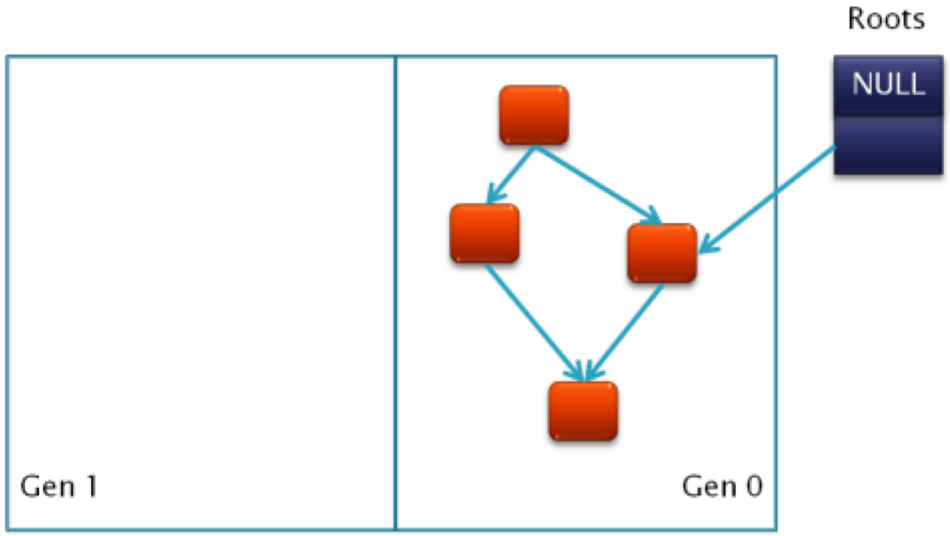
In .NET, 3 generations are used.



Case study: .NET

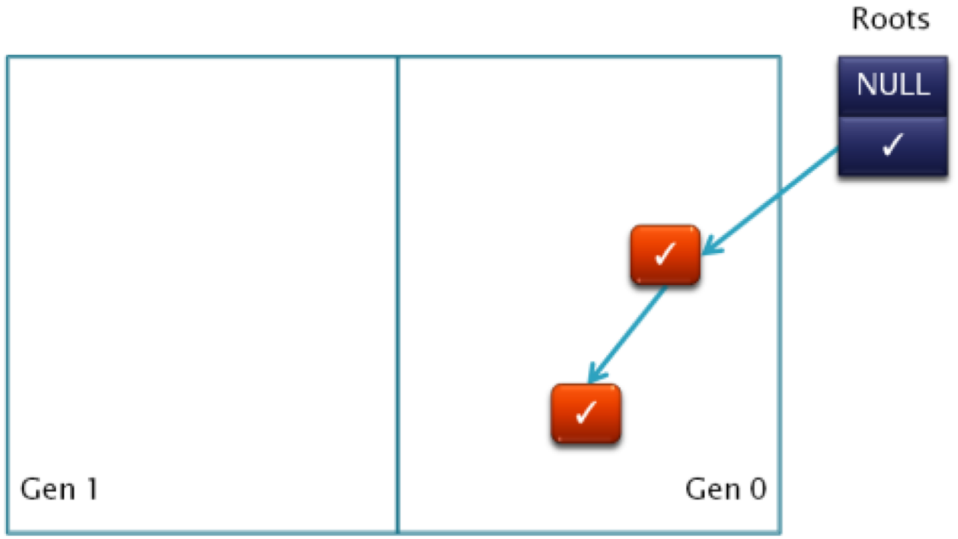
Imagine one of the references to an object is released.

.NET uses mark & sweep on Gen 0



Case study: .NET

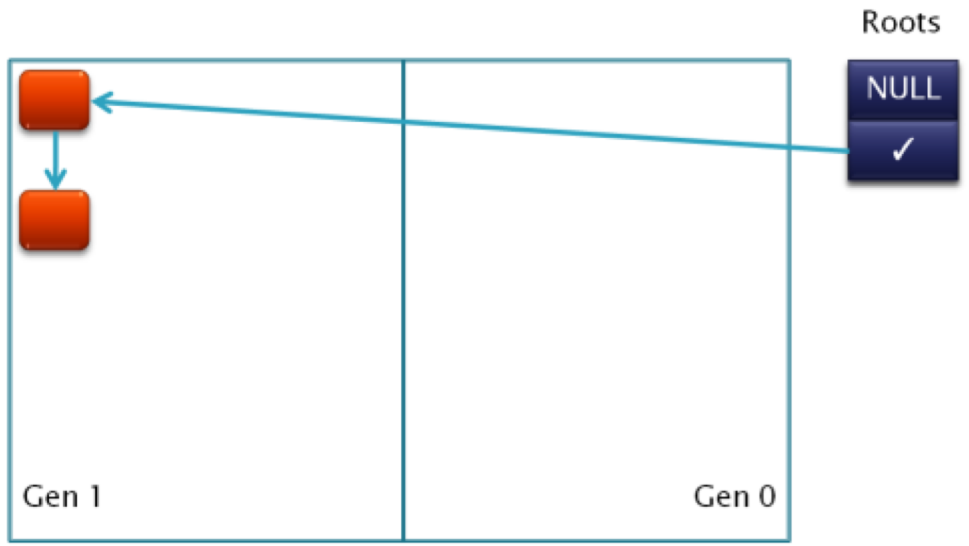
Now, the heap looks like this.



Case study: .NET

The surviving objects are promoted to Gen 1.

Promotion includes copying the objects and updating references to them.

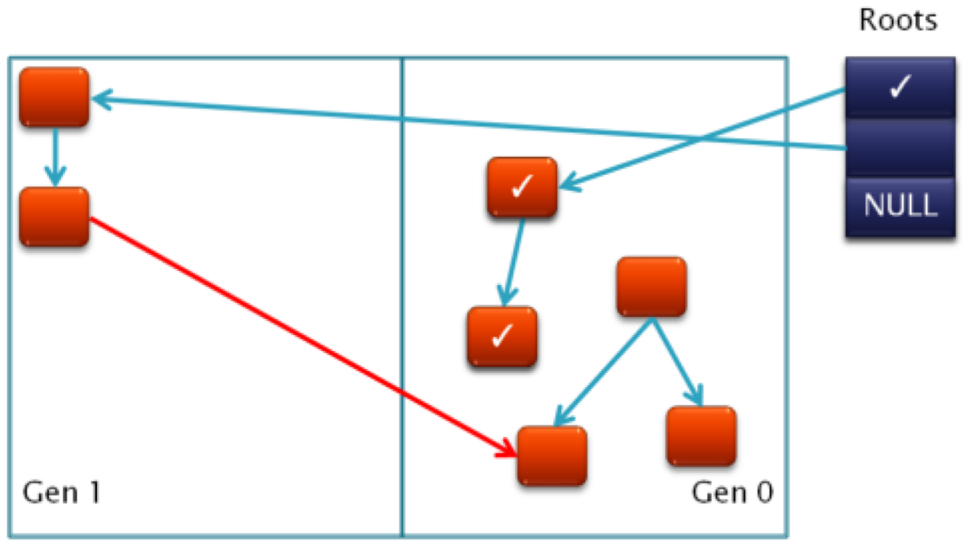


Case study: .NET

Now, assume a lot of allocation / deallocation has happened. The whole purpose of segregating into generations is to reduce the number of objects to inspect for marking.

So the first root is used for marking as it points to a Gen0 object. While using the second root the moment the marker sees that the reference is into a Gen1 object it **does not follow the reference**, speeding up marking process.

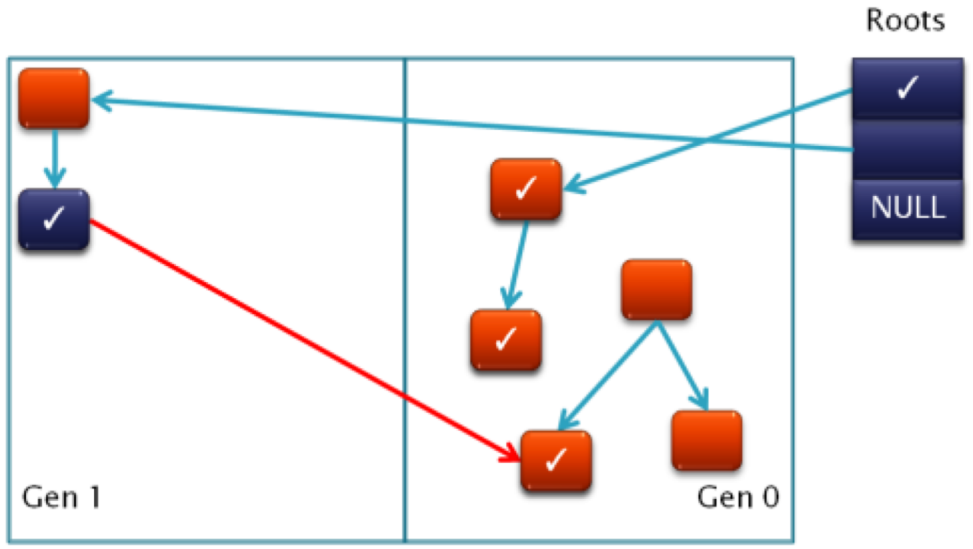
Now if we only consider the Gen0 objects for marking then we only mark the objects indicated by ✓. The marking algorithm will fail to locate the Gen1 to Gen0 references (shown in red) and some object marking will be left out leading to dangling pointers.



Case study: .NET

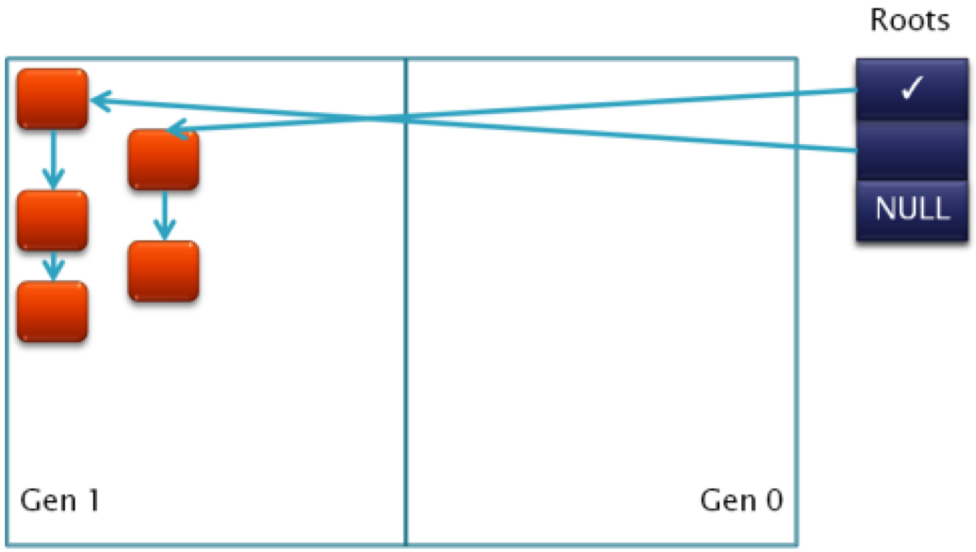
One of the way to handle this is to somehow record all references from Gen1 to and then use these objects as new roots for the marking phase.

If we use this method then we get a new set of marked objects as follows:



Case study: .NET

This now gives the full set of marked objects. Post another GC and promotion of surviving objects to higher generation we get

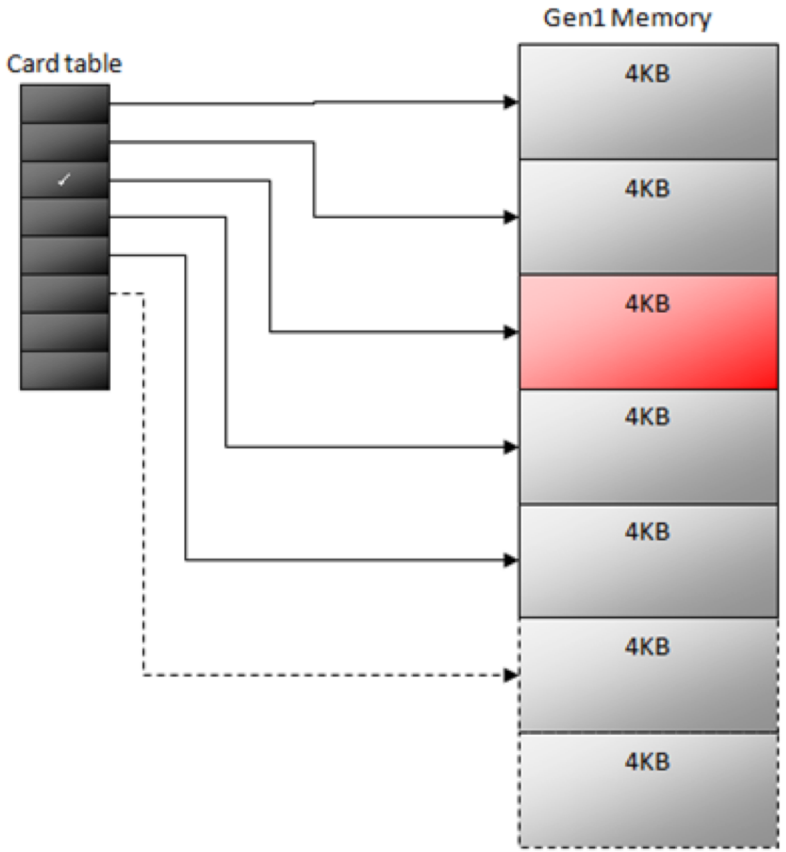


Tracking higher to lower generation references

In general applications there are very few (some studies show < 1% of all references) of these type of references.

However, they all need to be recorded. The main way of doing this is with a **Write barrier + card-table.**

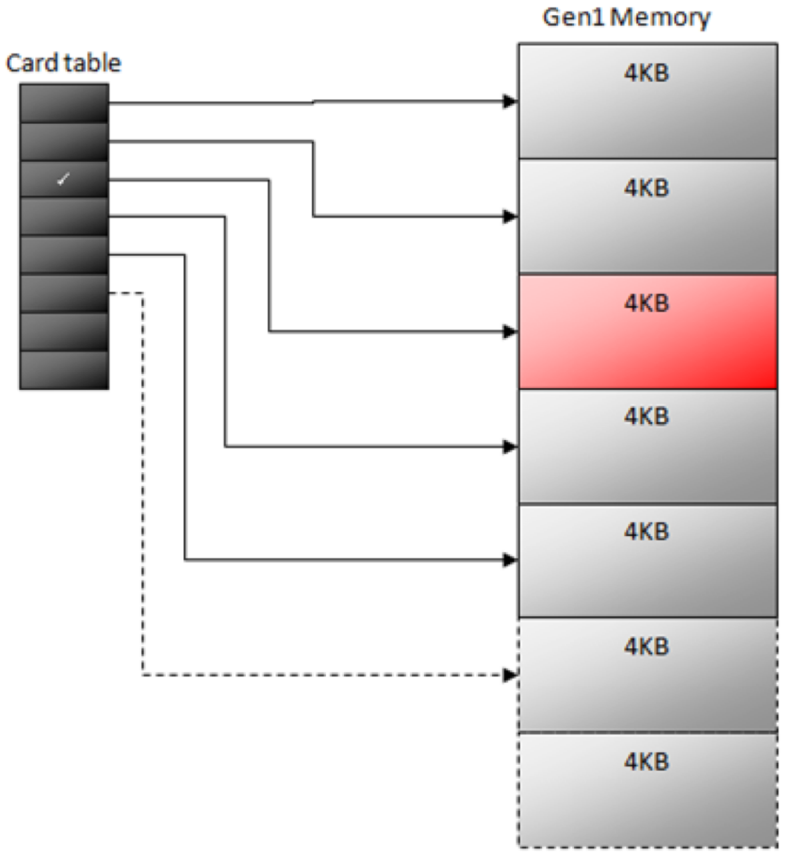
First a table called a card table is created. This is essentially an array of bits. Each bit indicates if a given range of memory is dirty (contains a write to a lower generation object). E.g. we can use a single bit to mark a 4KB block.



Tracking higher to lower generation references

- Whenever a reference assignment is made, we compare the assignee's address to that of the Gen1 memory range.
- If it falls within, we update the corresponding bit in the card table

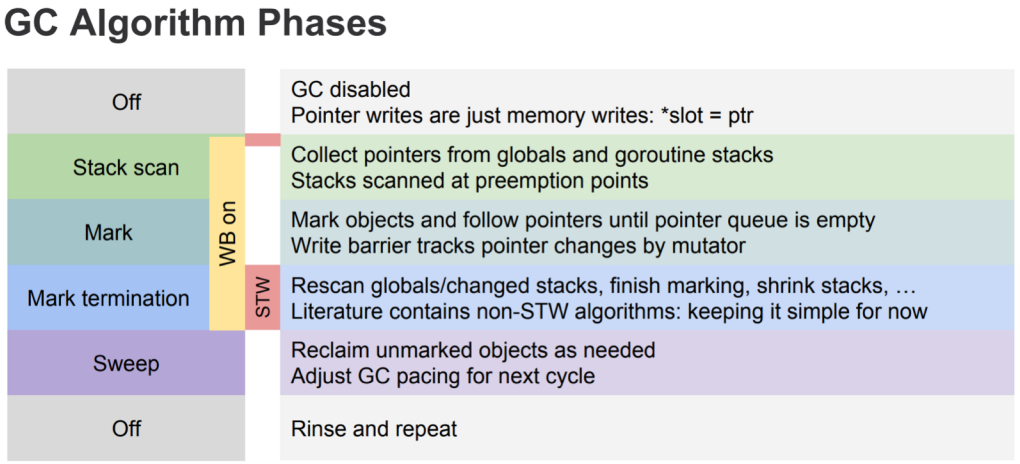
- During a GC pass, we start marking with root set / Gen 0 objects.
- Then, find dirty blocks, and consider every object in the dirty block to be a new root

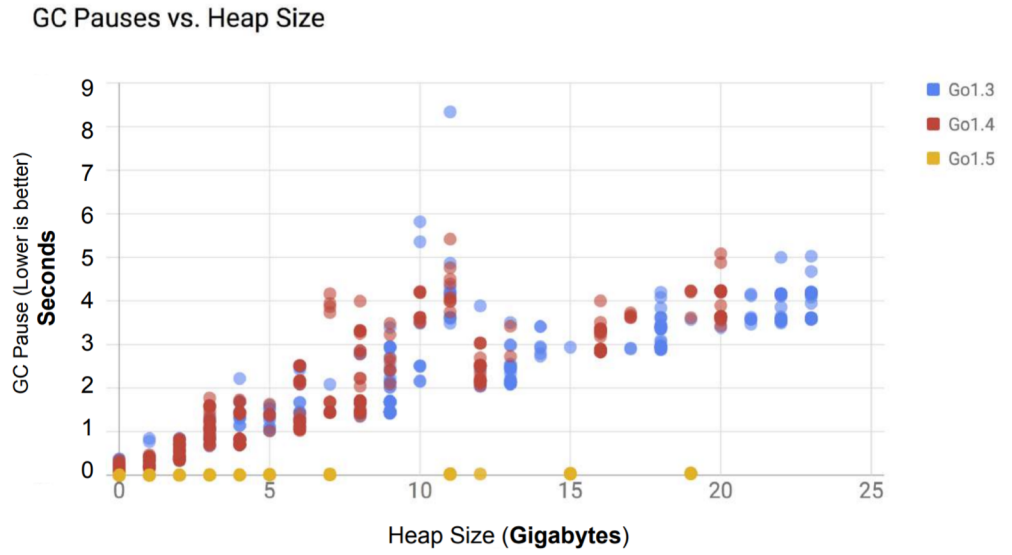


Case Study #2: Go

To create a garbage collector for the next decade, we turned to an algorithm from decades ago. Go's new garbage collector is a **concurrent, tri-color, mark-sweep** *collector, an idea first proposed by [Dijkstra in 1978](#).*

***This is a deliberate divergence from most "enterprise" grade garbage collectors of today**, and one that we believe is well suited to the properties of modern hardware and the latency requirements of modern software*





GC considerations

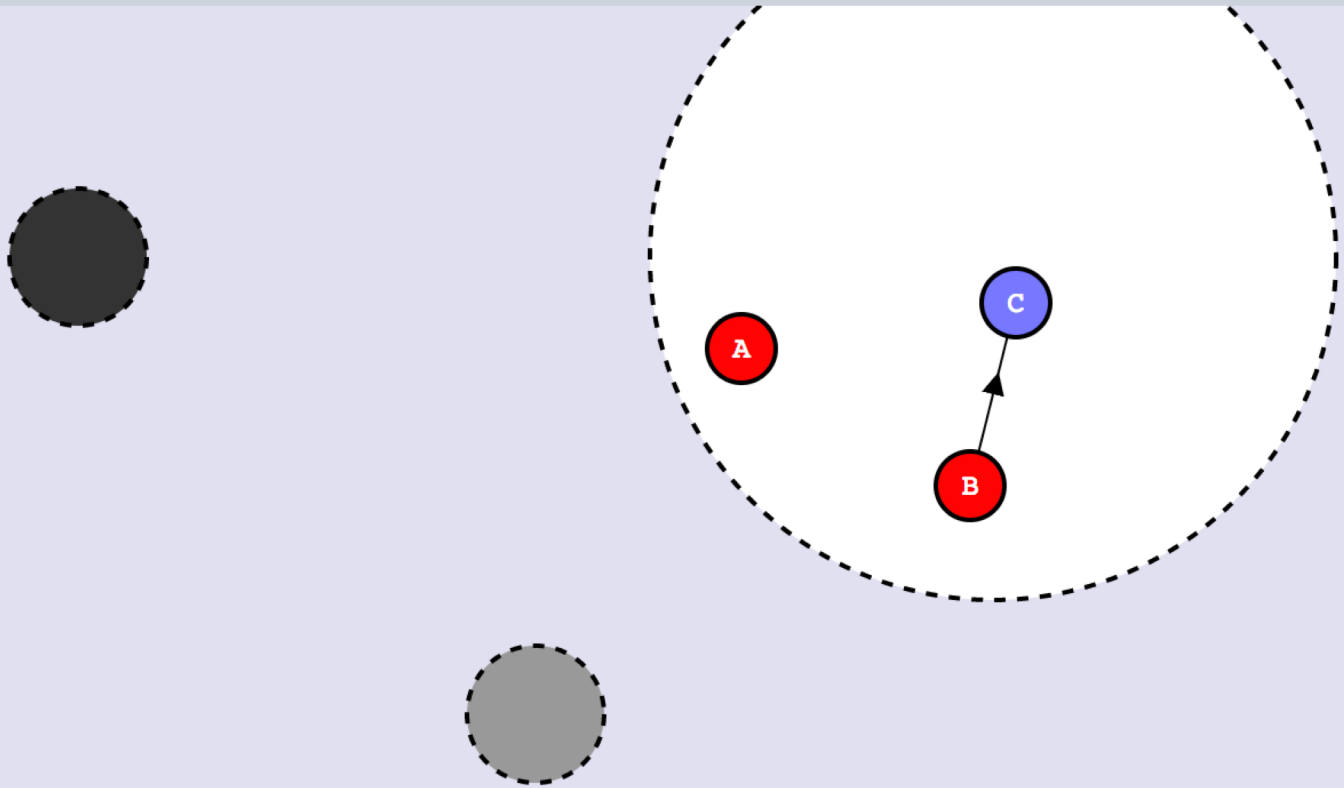
- **Program throughput**: how much does your algorithm slow the program down? This is sometimes expressed as a percentage of CPU time spent doing collection vs useful work.
- **GC throughput:** how much garbage can the collector clear given a fixed amount of CPU time?
- **Heap overhead:** how much additional memory over the theoretical minimum does your collector require? If your algorithm allocates temporary structures whilst collecting, does that make memory usage of your program very spiky?
- **Pause times:** how long does your collector stop the world for?
- **Pause frequency:** how often does your collector stop the world?
- **Pause distribution:** do you typically have very short pauses but sometimes have very long pauses? Or do you prefer pauses to be a bit longer but consistent?
- **Allocation performance:** is allocation of new memory fast, slow, or unpredictable?
- **Compaction:** does your collector ever report an out-of-memory (OOM) error even if there's sufficient free space to satisfy a request, because that space has become scattered over the heap in small chunks? If it doesn't you may find your program slows down and eventually dies, even if it actually had enough memory to continue.
- **Concurrency:** how well does your collector use multi-core machines?
- **Scaling:** how well does your collector work as heaps get larger?
- **Tuning:** how complicated is the configuration of your collector, out of the box and to obtain optimal performance?
- **Warmup time:** does your algorithm self-adjust based on measured behaviour and if so, how long does it take to become optimal?
- **Page release:** does your algorithm ever release unused memory back to the OS? If so, when?
- **Portability:** does your GC work on CPU architectures that provide weaker memory consistency guarantees than x86?
- **Compatibility:** what languages and compilers does your collector work with?

We join the process after it has run for a while.

The program manipulates linked lists. It has created nodes `A`, `B`, and `C`. The red objects `A` and `B` are *root objects*: they are always reachable. There is one pointer: `B.next = &C`. The garbage collector assigns objects to three sets: black, grey, and white. Currently, all three objects are in the white set, because the garbage collector is not running a cycle.

```
var A LinkedListNode;
var B LinkedListNode;
// ...
B.next = &LinkedListNode{next: nil};
// ...
A.next = &LinkedListNode{next: nil};
*(B.next).next = &LinkedListNode{next: nil};
B.next = *(B.next).next;
B.next = nil;
```

Phase: PROGRAM

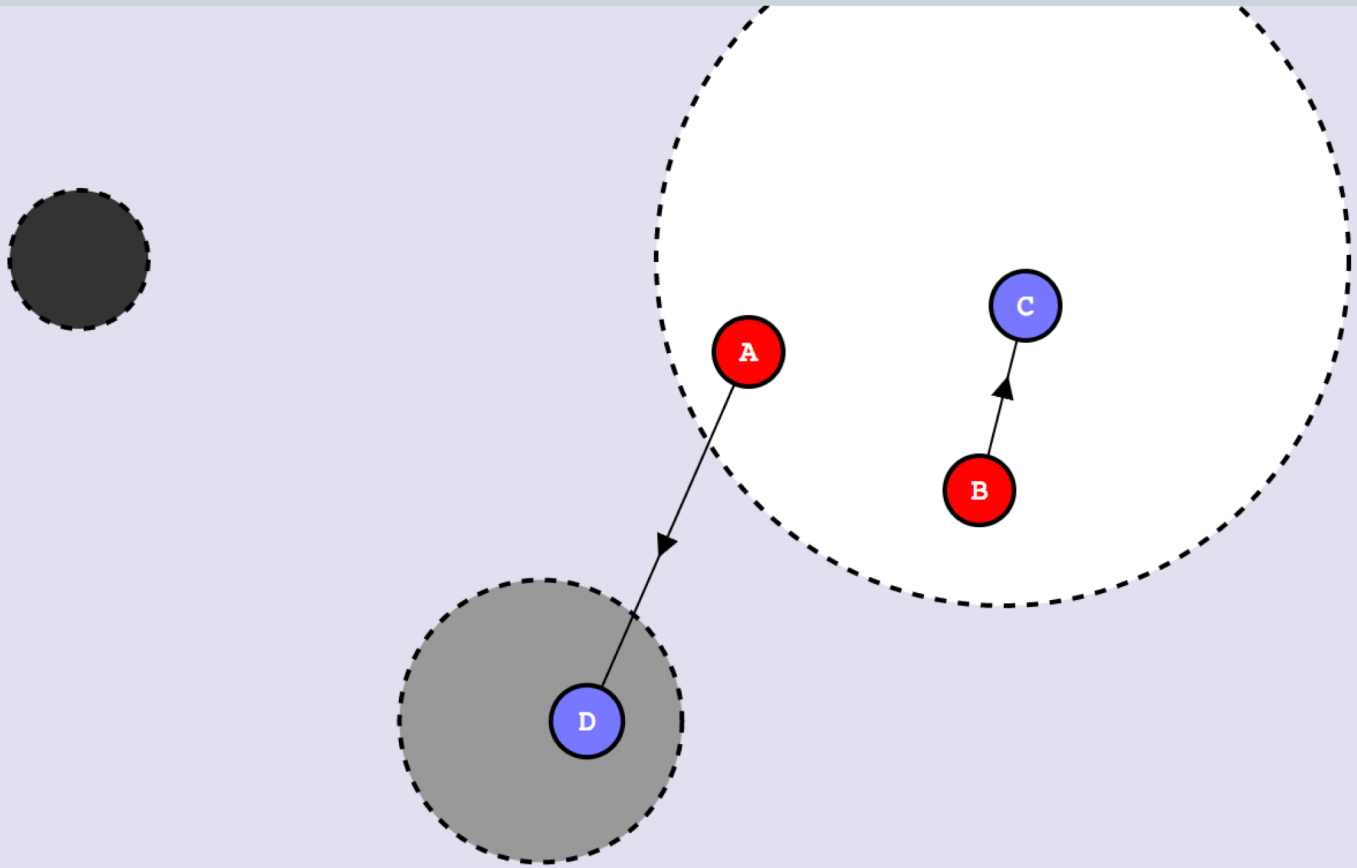


The program creates a new object, `D`.

The program assigns `D`'s address to `A.next`. Notice that `D`, as a new object, is put in the grey set. This is due to a general rule: `D` is colored grey because its address is assigned to `A.next`. When a pointer field is changed, the pointed-to object is colored. Since all new objects have their address assigned somewhere, they immediately get colored grey.

```
var A LinkedListNode;
var B LinkedListNode;
// ...
B.next = &LinkedListNode{next: nil};
// ...
A.next = &LinkedListNode{next: nil};
*(B.next).next = &LinkedListNode{next: nil};
B.next = *(B.next).next;
B.next = nil;
```

Phase: PROGRAM

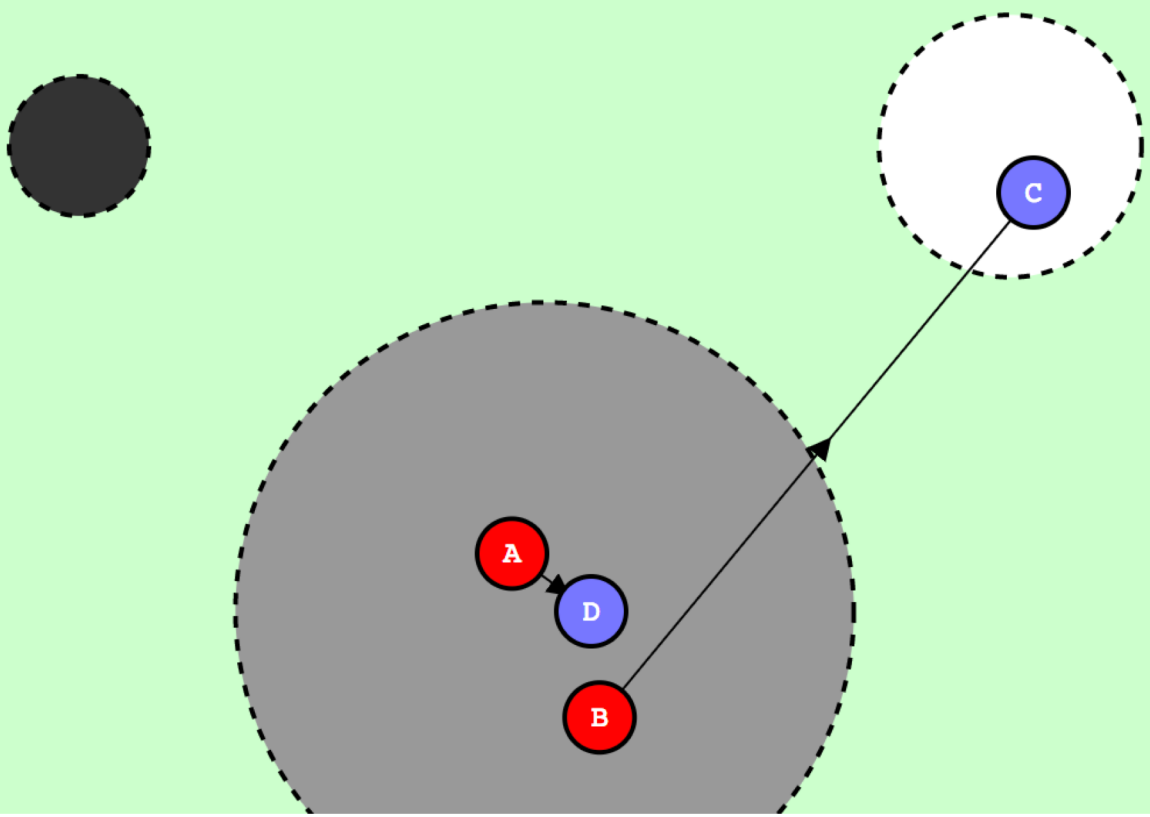


A GC cycle starts!

At the start of a GC cycle, the root objects are moved to the grey set. Here, the roots are A and B, which join the already-grey object D. Any process step is either a program step or a GC step. Since the program and the collector run concurrently, we will now see an interleaving of program steps and GC steps.

```
var A LinkedListNode;
var B LinkedListNode;
// ...
B.next = &LinkedListNode{next: nil};
// ...
A.next = &LinkedListNode{next: nil};
*(B.next).next = &LinkedListNode{next: nil};
B.next = *(B.next).next;
B.next = nil;
```

Phase: GC

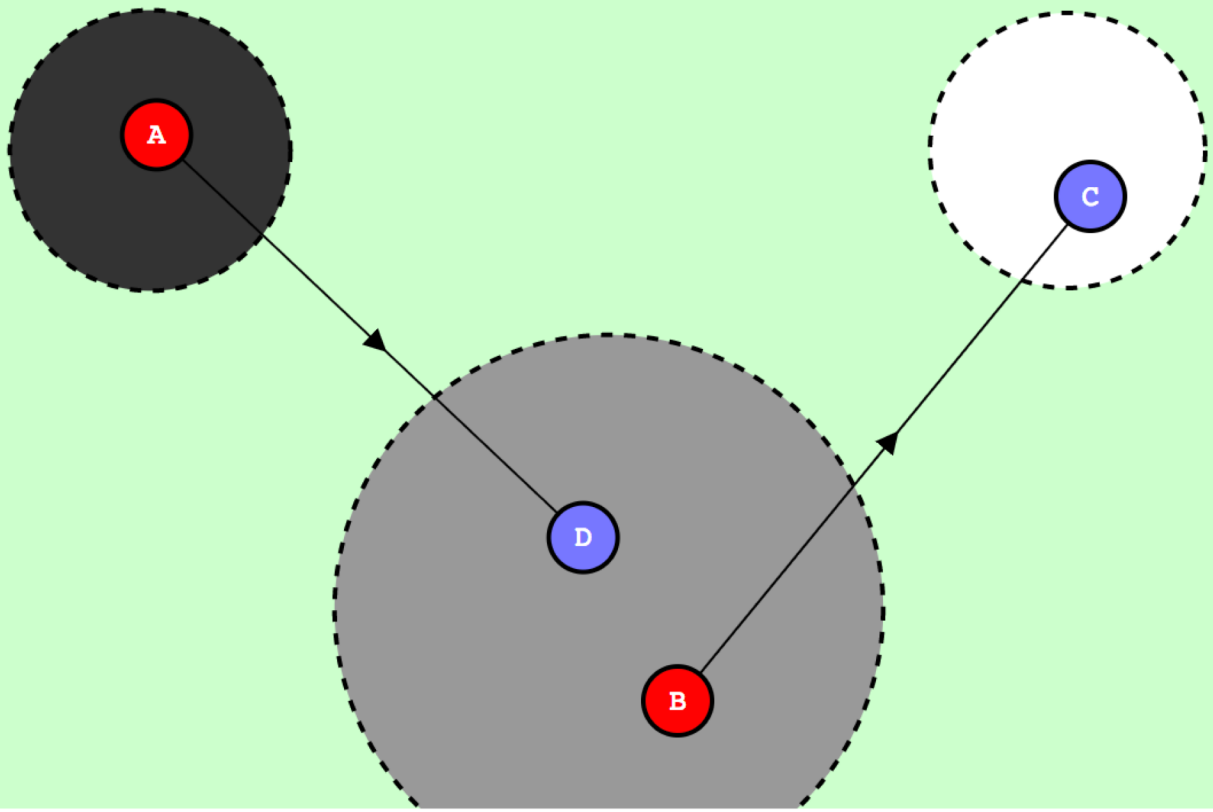


A GC step runs. The GC picks object A to scan.

To scan an object, the collector colors it black, and colors its children grey. Object A has one child, D, and it is already in the grey set. At any stage, we can count the number of moves the GC has left to do: `2*|white| + |grey|`. The collector does at least one move at every step, and when it reaches zero, it is finished.

```
var A LinkedListNode;
var B LinkedListNode;
// ...
B.next = &LinkedListNode{next: nil};
// ...
A.next = &LinkedListNode{next: nil};
*(B.next).next = &LinkedListNode{next: nil};
B.next = *(B.next).next;
B.next = nil;
```

Phase: GC

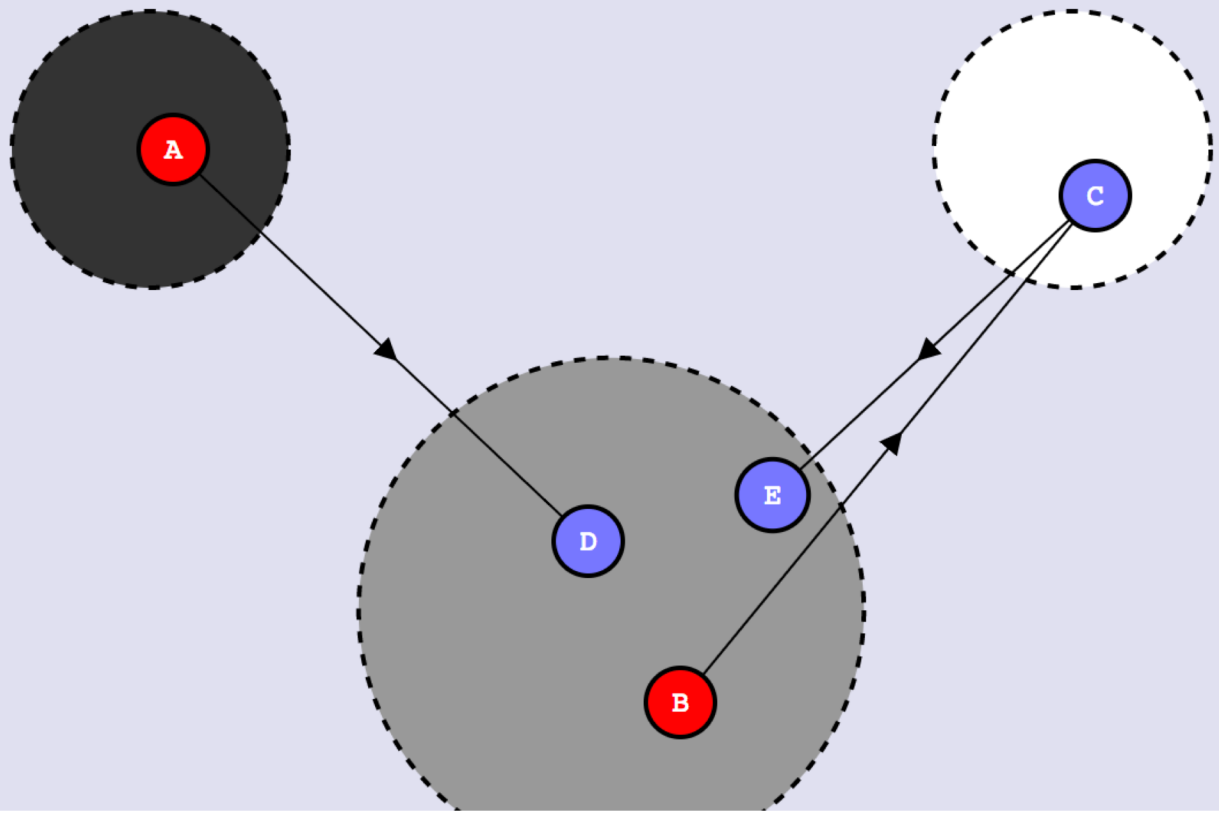


The program creates a new object, `E`.

The program assigns `E`'s address to `C.next`. `E` is then put in the grey set. By doing so, the program has increased the steps left for the collector; by allocating lots of new objects, the program can delay the final sweep. Notice that the white set only ever decreases in size, and is only re-populated when the collector sweeps the heap.

```
var A LinkedListNode;
var B LinkedListNode;
// ...
B.next = &LinkedListNode{next: nil};
// ...
A.next = &LinkedListNode{next: nil};
*(B.next).next = &LinkedListNode{next: nil};
B.next = *(B.next).next;
B.next = nil;
```

Phase: PROGRAM

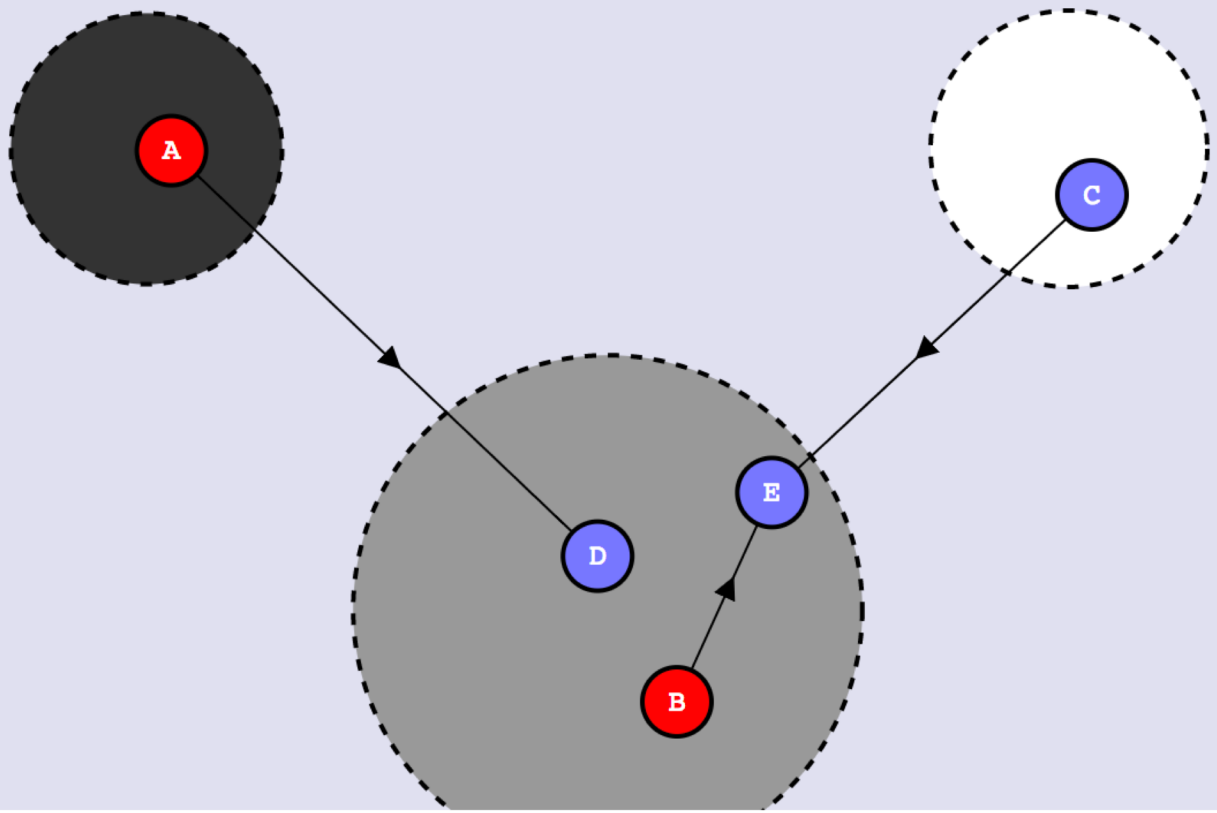


The program moves a pointer!

It runs `B.next = *(B.next).next`. In doing so, the object `C` has become unreachable: there is no way for the program to recover a pointer to `C`. This means the collector will leave `C` in the white set, and it will be collected at the end of the GC cycle.

```
var A LinkedListNode;
var B LinkedListNode;
// ...
B.next = &LinkedListNode{next: nil};
// ...
A.next = &LinkedListNode{next: nil};
*(B.next).next = &LinkedListNode{next: nil};
B.next = *(B.next).next;
B.next = nil;
```

Phase: PROGRAM

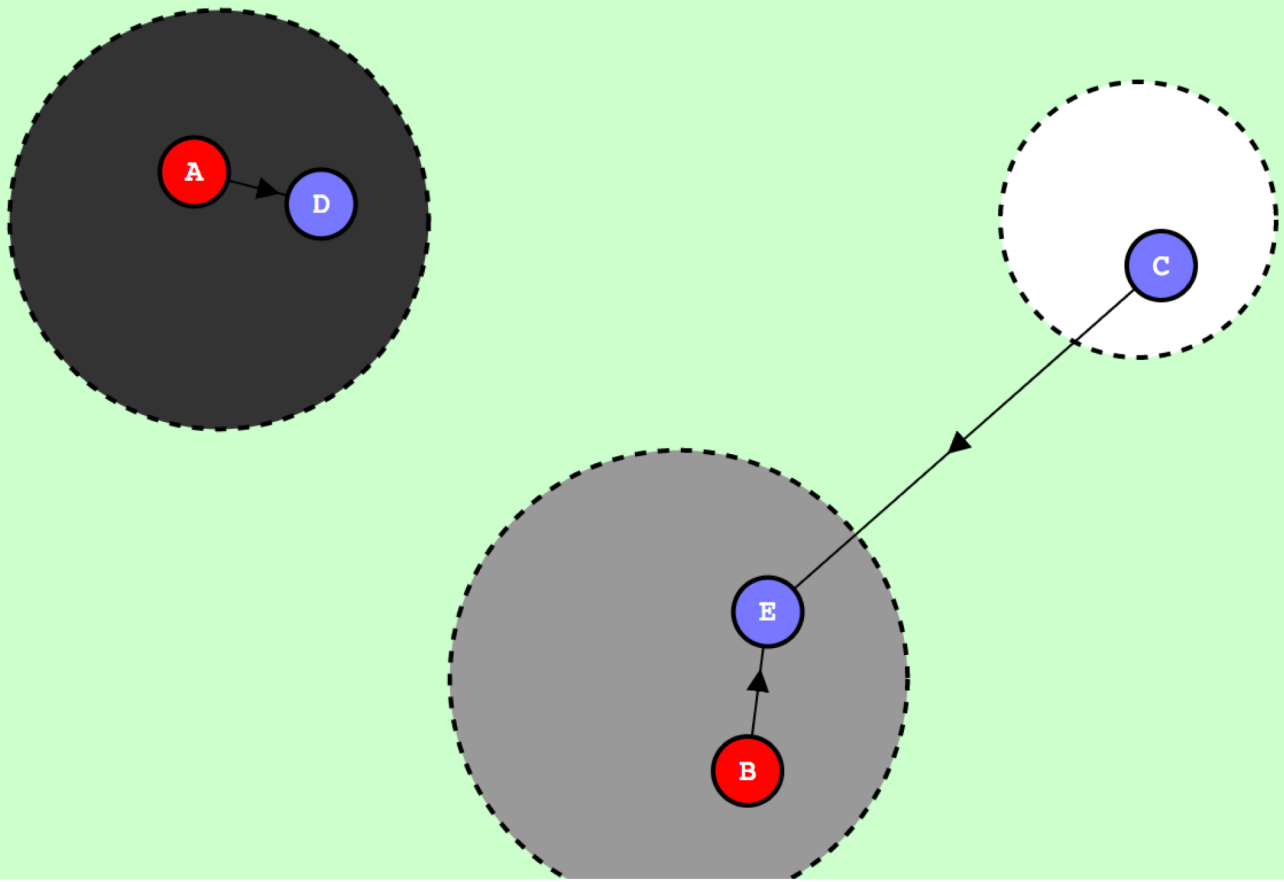


The GC picks object D to scan.

D has no descendants to pull into the grey set.

```
var A LinkedListNode;
var B LinkedListNode;
// ...
B.next = &LinkedListNode{next: nil};
// ...
A.next = &LinkedListNode{next: nil};
*(B.next).next = &LinkedListNode{next: nil};
B.next = *(B.next).next;
B.next = nil;
```

Phase: GC

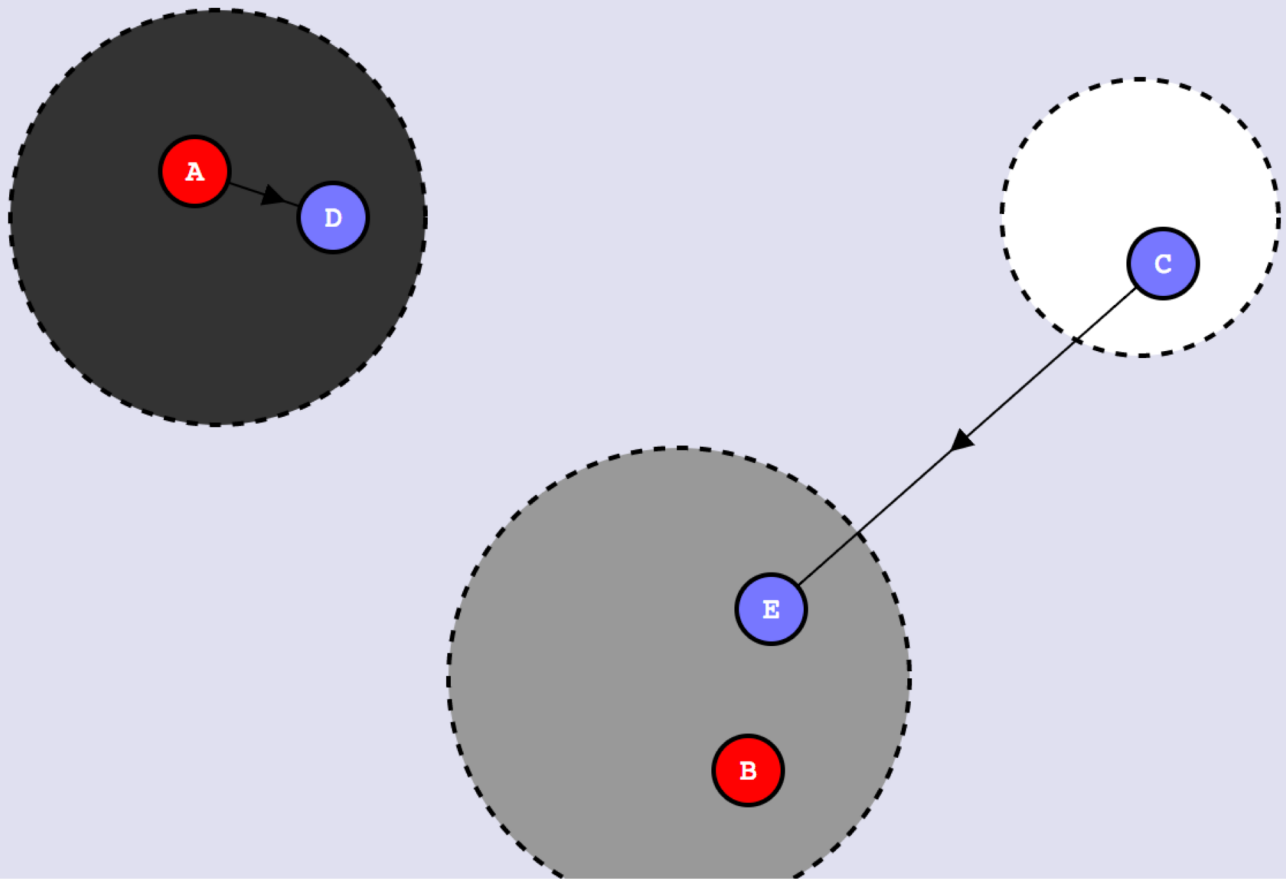


The program sets `B.next = nil`.

Object `E` has just become unreachable. Hmm ... `E` is in the grey set, so it won't be collected! Doesn't this mean we have a memory leak? Actually, it's okay: `E` will be collected on the *next* GC cycle. The tricolor algorithm only guarantees that, if an object is unreachable at the *start* of a GC cycle, it will be freed at the *end* of that cycle.

```
var A LinkedListNode;
var B LinkedListNode;
// ...
B.next = &LinkedListNode{next: nil};
// ...
A.next = &LinkedListNode{next: nil};
*(B.next).next = &LinkedListNode{next: nil};
B.next = *(B.next).next;
B.next = nil;
```

Phase: PROGRAM

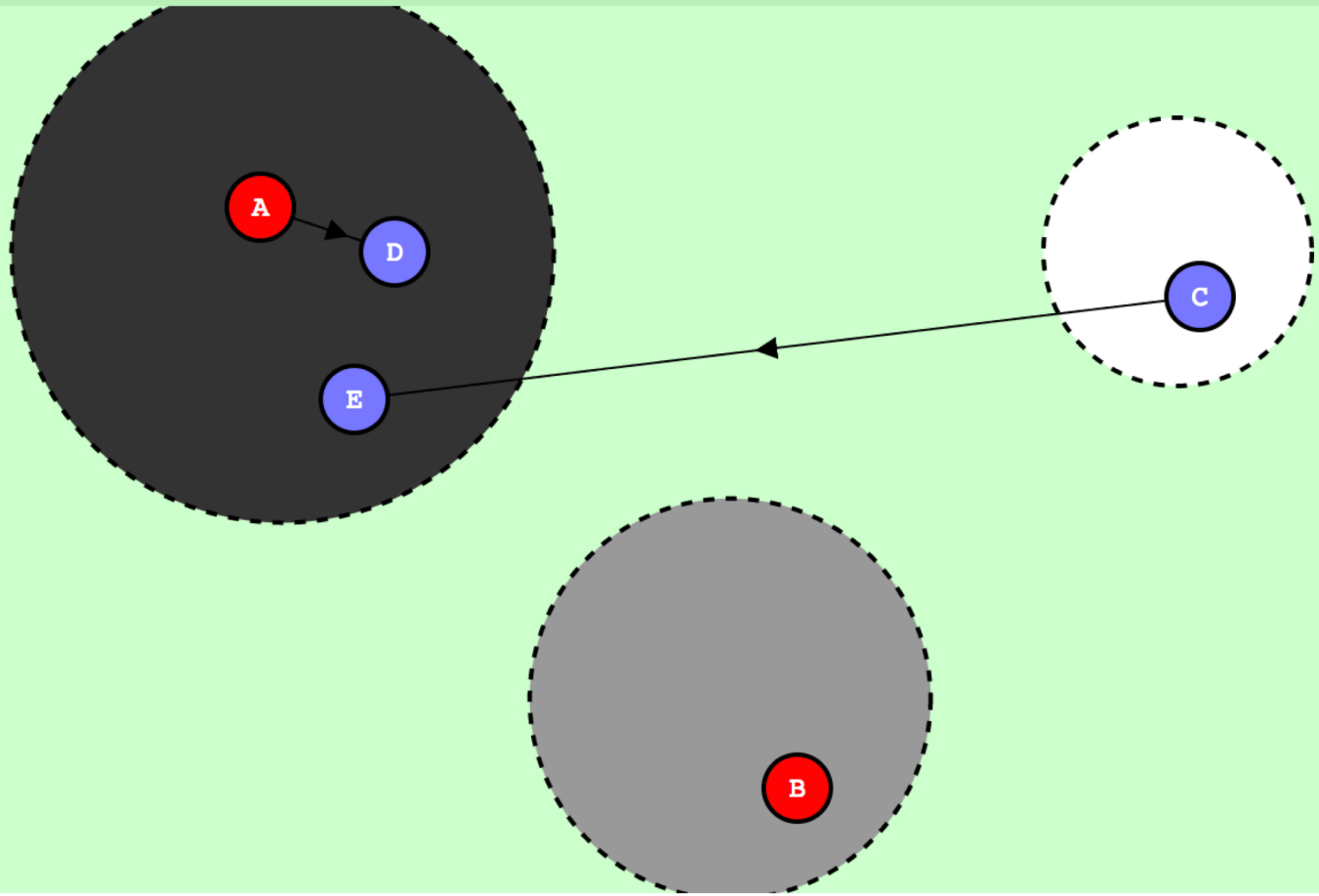


The collector scans object E.

The collector moves E to the black set. Note that C doesn't move: it points to E, but is not pointed to by E.

```
var A LinkedListNode;
var B LinkedListNode;
// ...
B.next = &LinkedListNode{next: nil};
// ...
A.next = &LinkedListNode{next: nil};
*(B.next).next = &LinkedListNode{next: nil};
B.next = *(B.next).next;
B.next = nil;
```

Phase: GC

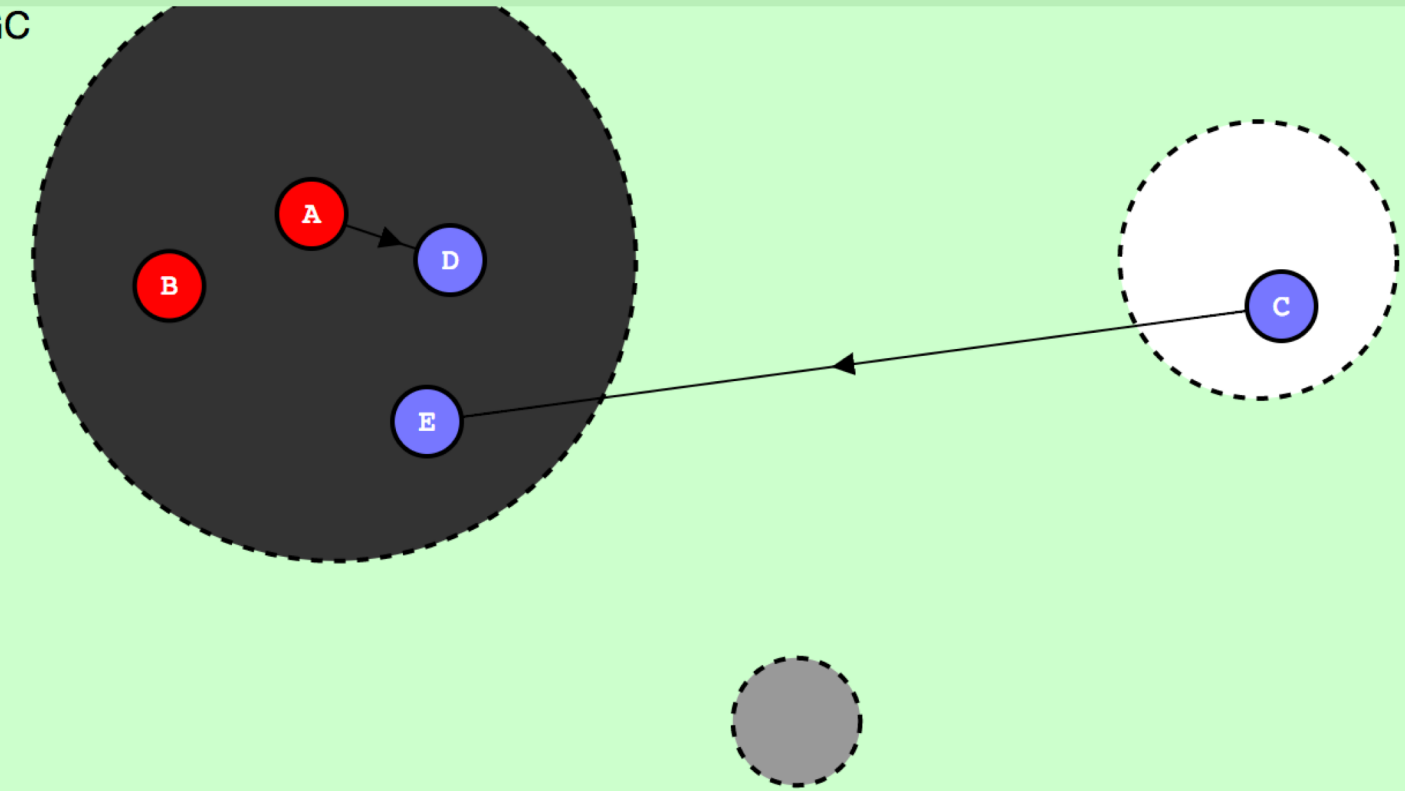


The collector scans the last grey object, B.

The grey set is now empty!

```
var A LinkedListNode;
var B LinkedListNode;
// ...
B.next = &LinkedListNode{next: nil};
// ...
A.next = &LinkedListNode{next: nil};
*(B.next).next = &LinkedListNode{next: nil};
B.next = *(B.next).next;
B.next = nil;
```

Phase: GC

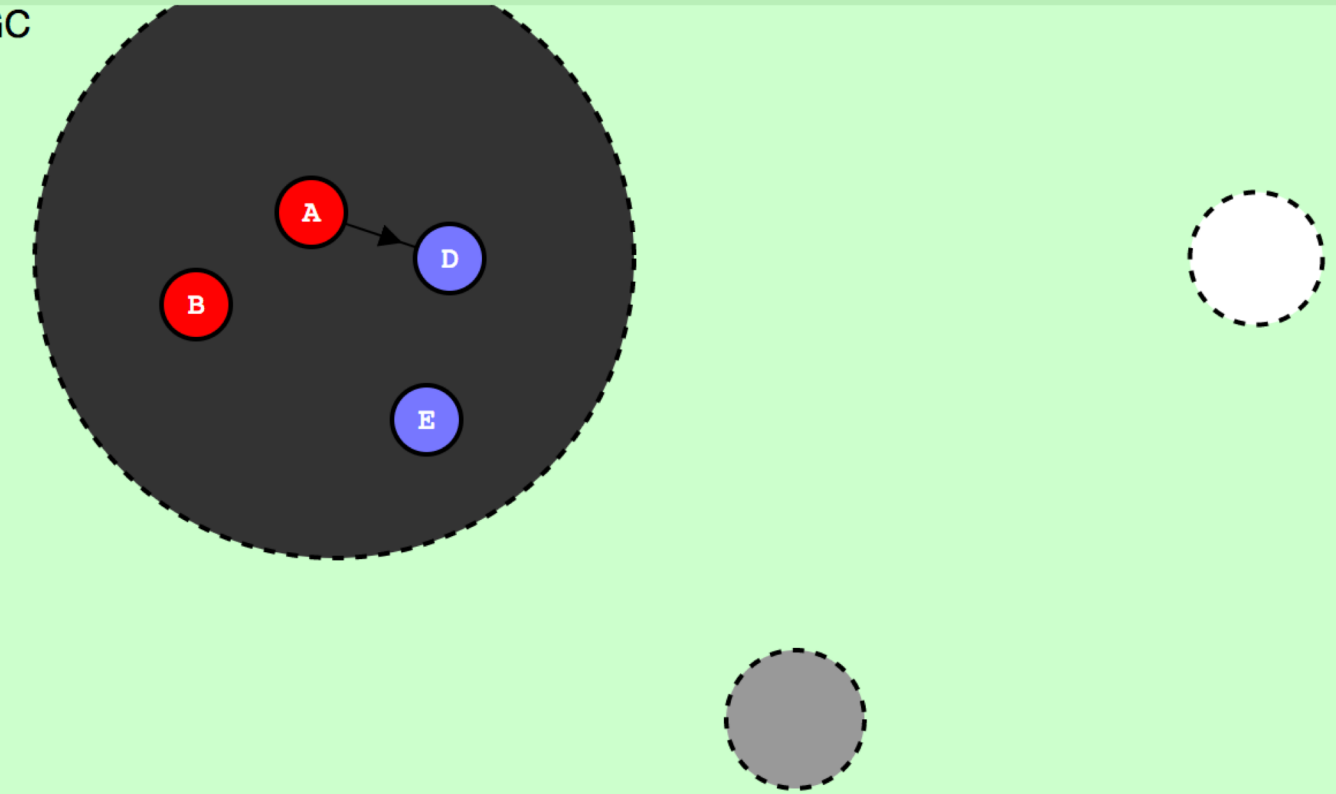


The collector frees the white set (the garbage).

There are no more grey objects! Finally, the collector knows that everything in the white set is unreachable. Here, the white set is just the object `C`, which the collector now frees. The unreachable object `E` lives on until the next GC cycle, because it only became unreachable during this GC cycle.

```
var A LinkedListNode;
var B LinkedListNode;
// ...
B.next = &LinkedListNode{next: nil};
// ...
A.next = &LinkedListNode{next: nil};
*(B.next).next = &LinkedListNode{next: nil};
B.next = *(B.next).next;
B.next = nil;
```

Phase: GC



The collector changes the set colors. We're done!

In implementation, the collector doesn't need to move or recolor any objects; instead it just reinterprets the *black* to instead mean *white*, and vice versa, for the next GC cycle. This is simpler and faster.

```
var A LinkedListNode;
var B LinkedListNode;
// ...
B.next = &LinkedListNode{next: nil};
// ...
A.next = &LinkedListNode{next: nil};
*(B.next).next = &LinkedListNode{next: nil};
B.next = *(B.next).next;
B.next = nil;
```

Phase: GC