Garbage Collection: Basics

Kenjiro Taura

2024/06/09

Contents

Introduction

Two ways memory management goes wrong

- **premature free:** reclaim/reuse space for data when it may still be used in future
- **memory leak:** do not reclaim/reuse space for date when it is no longer used

What is Garbage Collection (GC)?

- GC automates memory management, by identifying when data becomes "*never used in future*"
  - or, conversely, by identifying which data "*may be*" used in future

Data that “may be” used in future ?

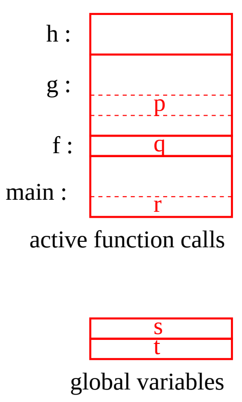
```
int * s, * t;
void h() { ... }

void g() {
  ...
  h();
  ... = p->x ... }

void f() {
  ...
  g()
  ... = q->y ... }

int main() {
  ...
  f()
  ... = r->z ... }
```

- global variables
- local variables of active function calls (calls that have started but have not finished)
- and ...



Data that “may be” used in future ?

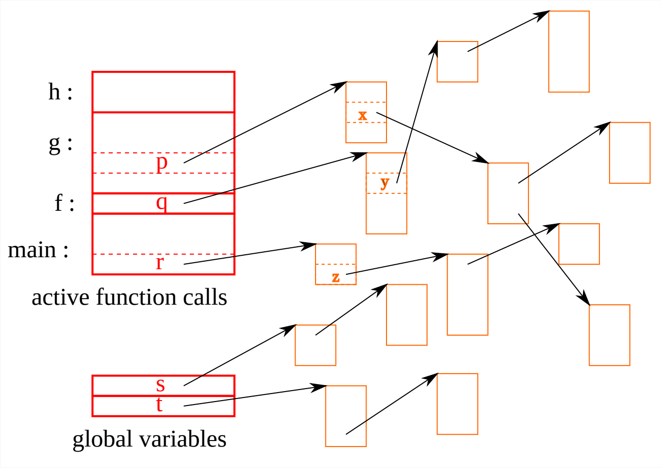
```
int * s, * t;
void h() { ... }

void g() {
  ...
  h();
  ... = p->x ... }

void f() {
  ...
  g()
  ... = q->y ... }

int main() {
  ...
  f()
  ... = r->z ... }
```

- global variables
- local variables of active function calls (calls that have started but have not finished)
- objects reachable from them by pointers



How GC basically works

Terminologies and the basic principle

- ***an object:*** the unit of data subject to memory allocation/release (malloc in C; objects in Java; etc.)
- ***the root:*** objects accessible without traversing pointers, such as global variables and local variables of active function calls
- ***(un)reachable objects:*** objects (un)reachable from the root by traversing pointers
- ***live objects:*** objects that may be accessed in future
- ***dead objects*** or ***garbage:*** objects that are never accessed in future

Terminologies and the basic principle

- ***collector:*** the program (or the thread/process) doing GC
- ***mutator:*** the user program
  - very GC-centric terminology, viewing the user program as someone simply “mutating” the graph of objects

the basic principle of GC:

objects unreachable from the root are dead

The two major GC methods (traversing GC and reference counting)

The two major GC methods (1) — traversing GC

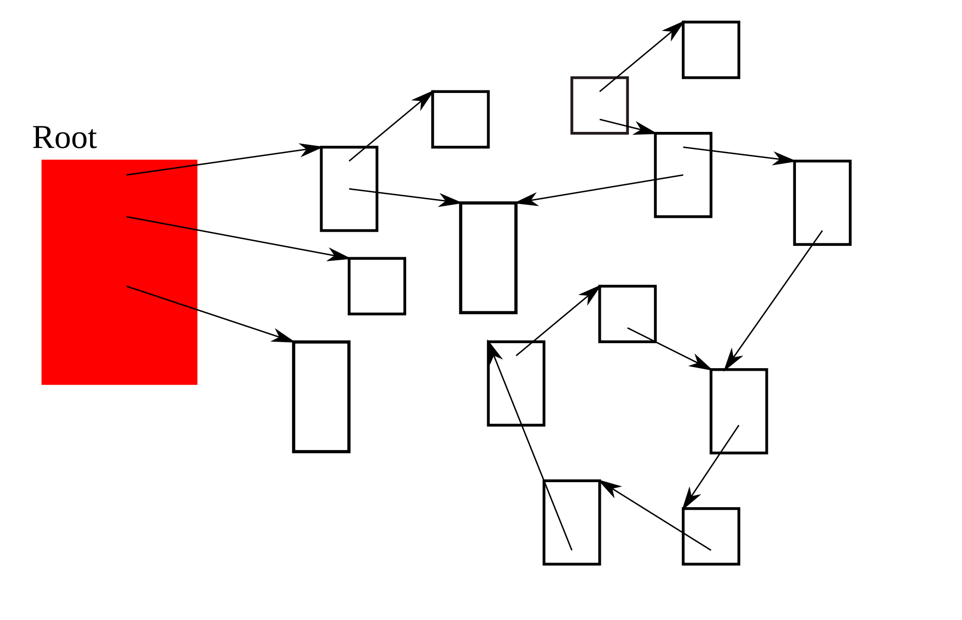
- simply traverses pointers from the root, to find (or *visit*) objects *reachable from the root*
- reclaim objects not visited
- two basic traversing methods
  - ***mark&sweep*** GC
  - ***copying*** GC

The two major GC methods (2) — reference counting (or RC)

- during execution, *maintain the number of pointers* pointing to each object ***(reference count)***
- *reclaim an object when its reference count drops to zero*
  - $\because$ *an object's reference count is zero* $\rightarrow$ *it's unreachable from the root*
- note: "GC" sometimes narrowly refers to the traversing GC only

How traversing GC works

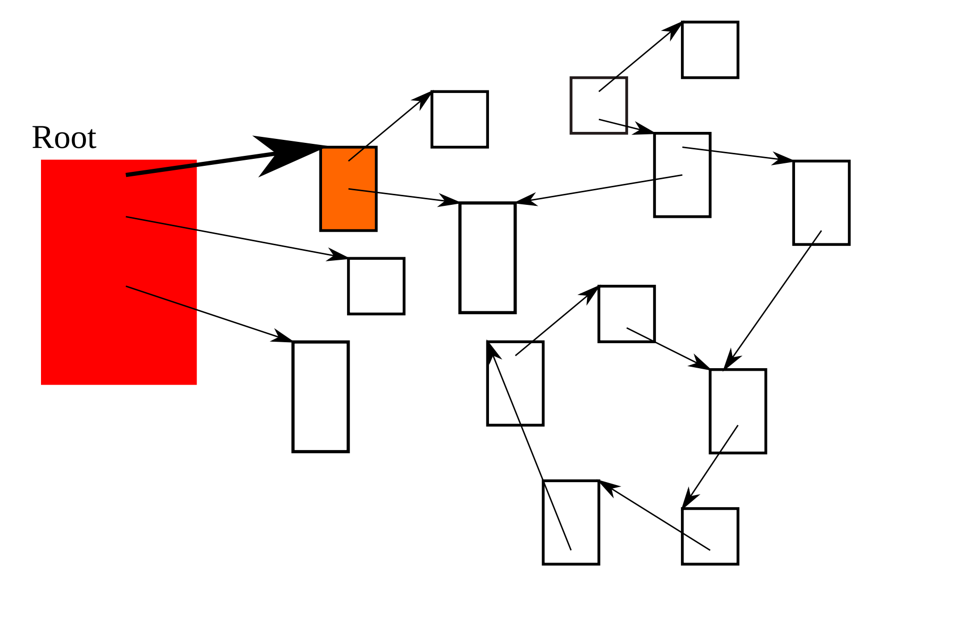
- traverse pointers from the root



- note: the difference between mark&sweep and copying is covered later

How traversing GC works

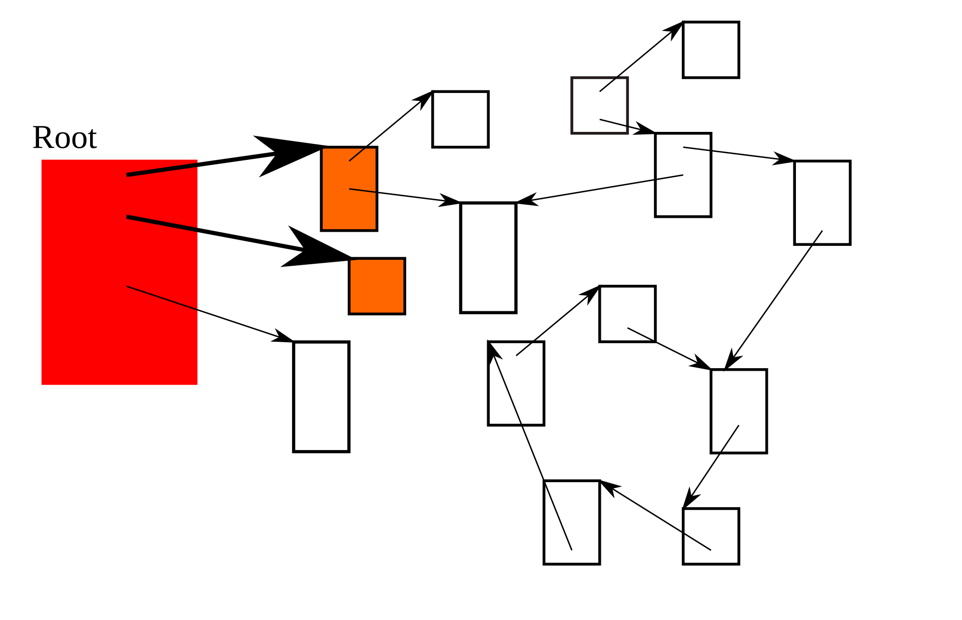
- traverse pointers from the root



- note: the difference between mark&sweep and copying is covered later

How traversing GC works

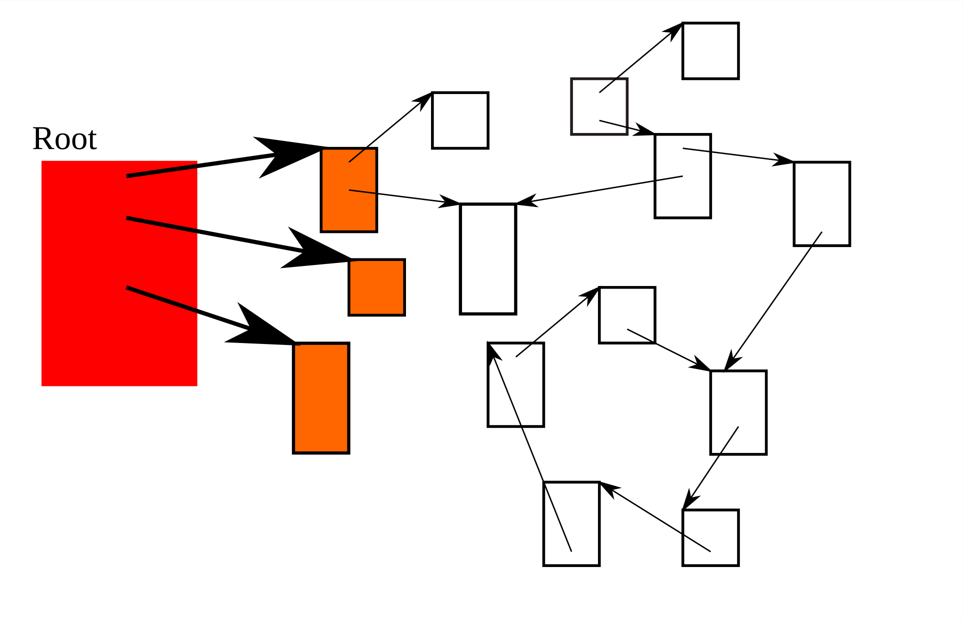
- traverse pointers from the root



- note: the difference between mark&sweep and copying is covered later

How traversing GC works

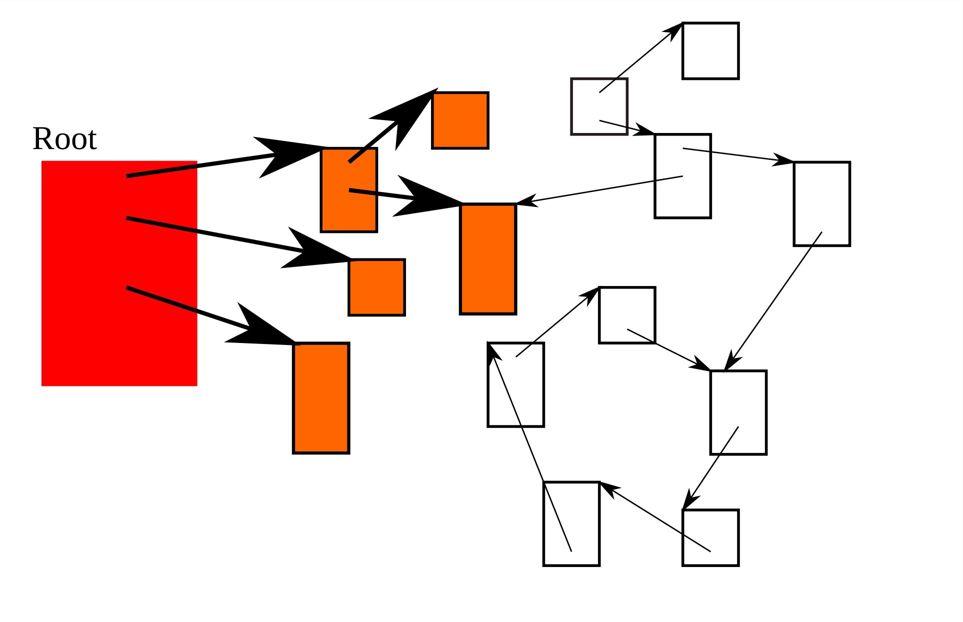
- traverse pointers from the root



- note: the difference between mark&sweep and copying is covered later

How traversing GC works

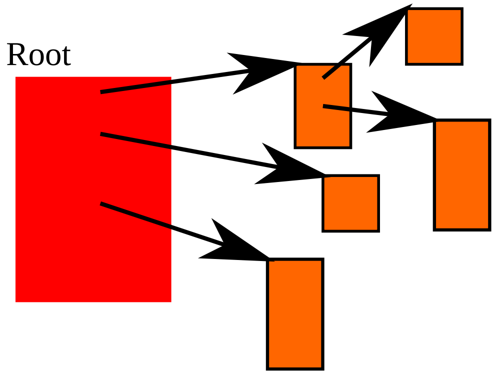
- traverse pointers from the root



- note: the difference between mark&sweep and copying is covered later

How traversing GC works

- traverse pointers from the root
- when no more "visited → unvisited" pointers are found, objects that have not been visited are garbage



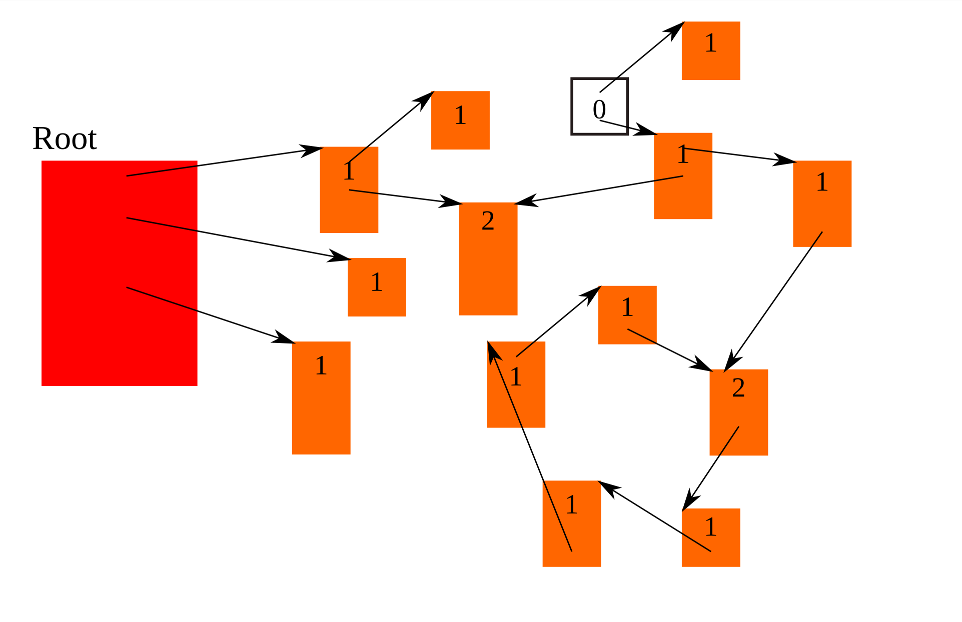
- note: the difference between mark&sweep and copying is covered later

How reference counting works

- each object has a ***reference count (RC)***, the number of pointers referencing it
- update RCs during execution; e.g., upon p = q
  - the RC of the object p points to `-= 1`
  - the RC of the object q points to `+= 1`

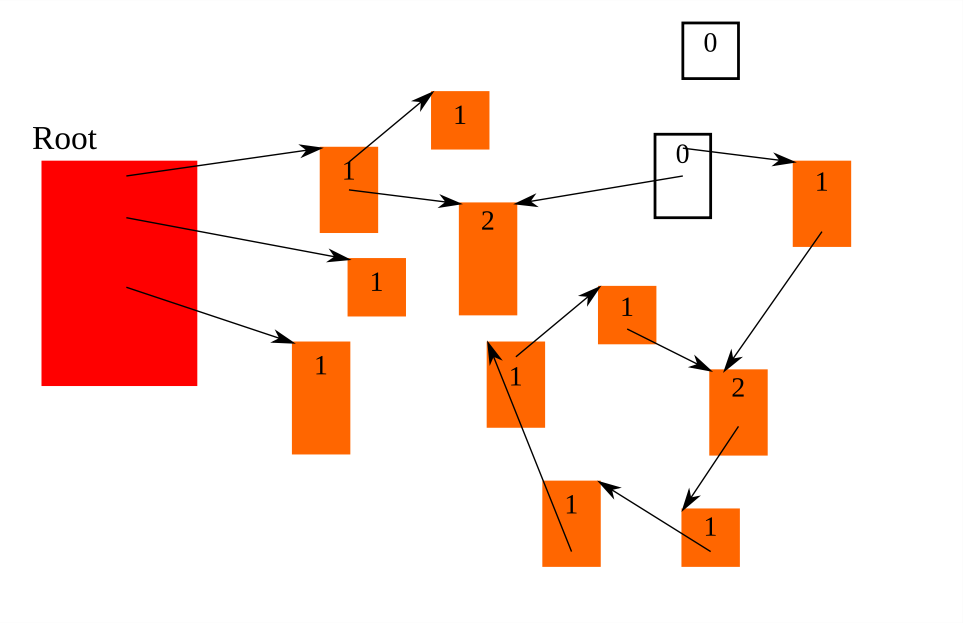
How reference counting works

- reclaim an object when its RC drops to zero



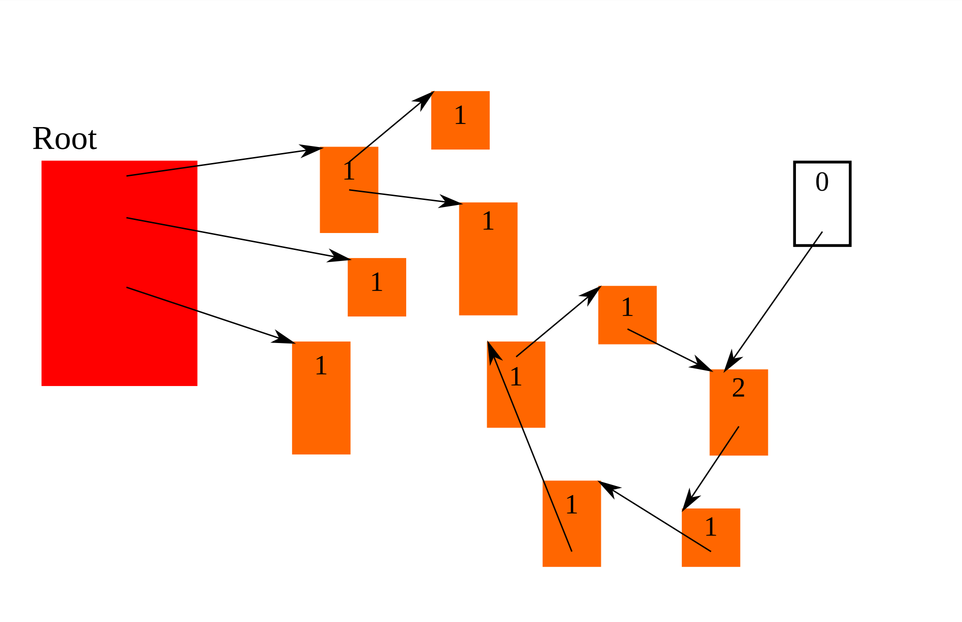
How reference counting works

- reclaim an object when its RC drops to zero
- RCs of objects pointed to by the reclaimed object decrease, which may result in reclaiming them too



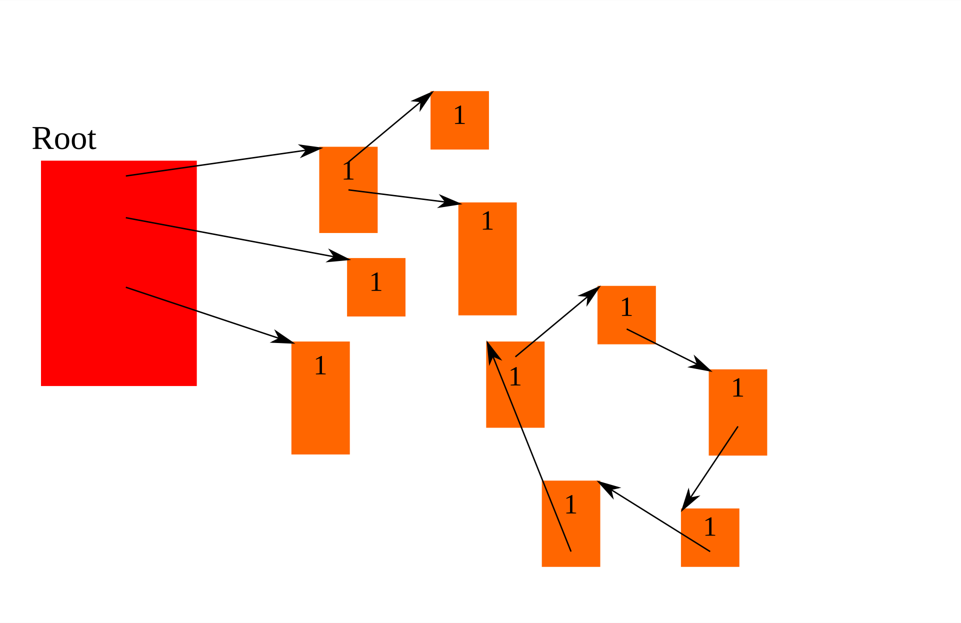
How reference counting works

- reclaim an object when its RC drops to zero
- RCs of objects pointed to by the reclaimed object decrease, which may result in reclaiming them too



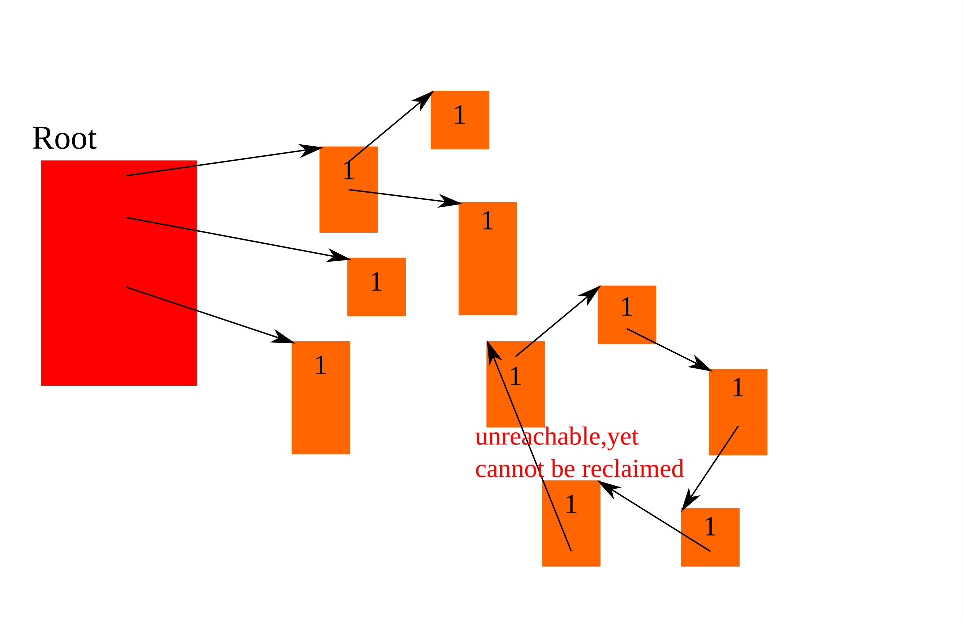
How reference counting works

- reclaim an object when its RC drops to zero
- RCs of objects pointed to by the reclaimed object decrease, which may result in reclaiming them too



How reference counting works

- reclaim an object when its RC drops to zero
- RCs of objects pointed to by the reclaimed object decrease, which may result in reclaiming them too



- ***note:*** unreachable cycles cannot be reclaimed (RC = 0 $\Rightarrow$ unreachable, but *not vice versa*)

When an RC changes

- a pointer variable is updated
- a reference is passed to a function
- a variable goes out of scope or a function returns
- ≈ any point when pointers get copied / dropped
- summary: *expensive*

```
p = q; p->f = q; etc.

int main() {
  object * q = ...;
  f(q); /* rc of q += 1 */
}

void f(object * p) {
  ...
  {
    object * r = ...;

  } /* RC of r -= 1 */
  ...
  return ...; /* RC of p -= 1 */
}
```

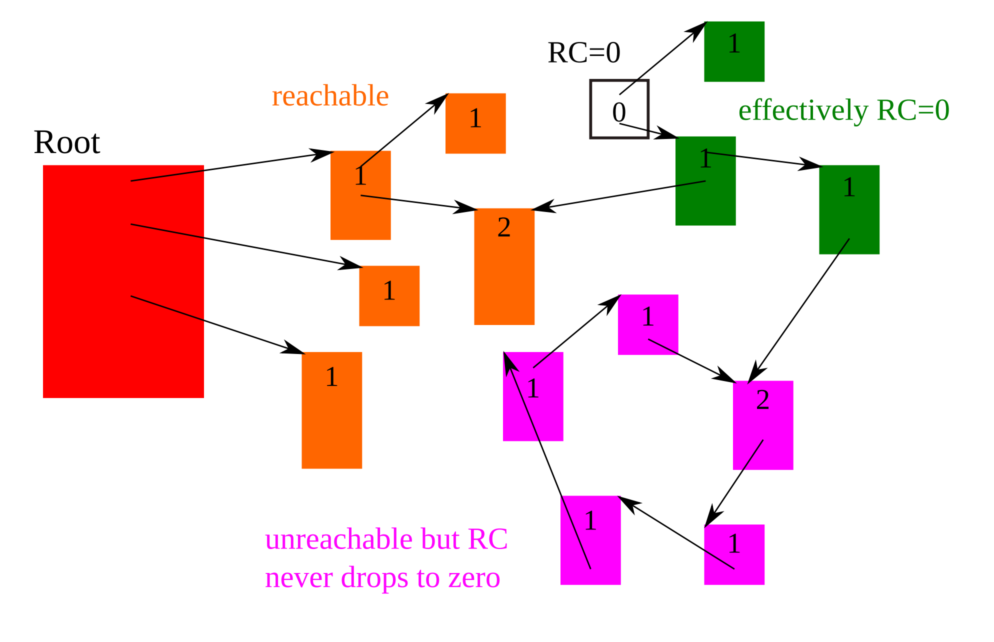
Evaluating GCs

Evaluating GCs

- preciseness:
  - garbage that can be collected
- memory allocation cost:
  - the work (including GC) required to allocate memory
- pause time:
  - the (worst case) time the mutator has to (temporarily) suspend for GC to function
- mutator overhead:
  - the overhead imposed on the mutator for GC to function

Criteria 1: preciseness

- *reference counting cannot reclaim cyclic garbage*
- reference counting < traversing GC (better)



Criteria 2: memory allocation cost (details ahead)

- traversing GC:
  - determined by the ratio "reachable objects" / "unreachable (reclaimed) objects" (later)
  - an advanced technique: ***generational GC***
- reference counting:
  - the cost of reclaiming an object once its RC drops to zero is small and constant
  - constant even if memory is scarce (good)

Criteria 3: pause time

- (better) reference counting $<$ traversing GC
- traversing GC:
  - stop the user program while it is traversing live objects
    - traverse *all* live objects, *en masse*, and reclaim *all* unreached objects
  - why so? troubled if the mutator runs (= changes the graph of objects) during traversing
    - a solution: ***incremental GC***
    - generational GCs mitigate it too
- reference counting:
  - when an object's RC drops to zero (as a result of mutator's action), it can be reclaimed *immediately*
  - reclaim garbage as they arise

Criteria 4: mutator overhead

- (better) traversing < reference counting
- reference counting has a large overhead for updating RCs
- e.g.,

```
object * p, * q;
p = q;
```

will perform:

```
if (p) p->rc--;
if (q) q->rc++;
p = q;
```

- moreover,

Criteria 4: mutator overhead

- what if it is multithreaded?
- what if the counter overflows (how to check it)?

- techniques: ***deferred reference counting, sticky reference counting, 1 bit reference counting***
- remark: some traversing GCs (e.g., generational and incremental) add overhead to pointer updates too

Summary

	traversing GC	reference counting
preciseness	$+$	$-$
allocation cost	? ($*$)	?
pause time	$-$ (†)	$+$
mutator overhead	$+$ (‡)	$-$

- ($*$) depends on size of reachable graph and memory; *generational* GC helps
- (†) *incremental* GC helps
- (‡) both *generational* and *incremental* GC impose some mutator overheads

Two Types of Traversing Collectors

mark&sweep GC vs. copying GC

they differ in what to do on reachable objects

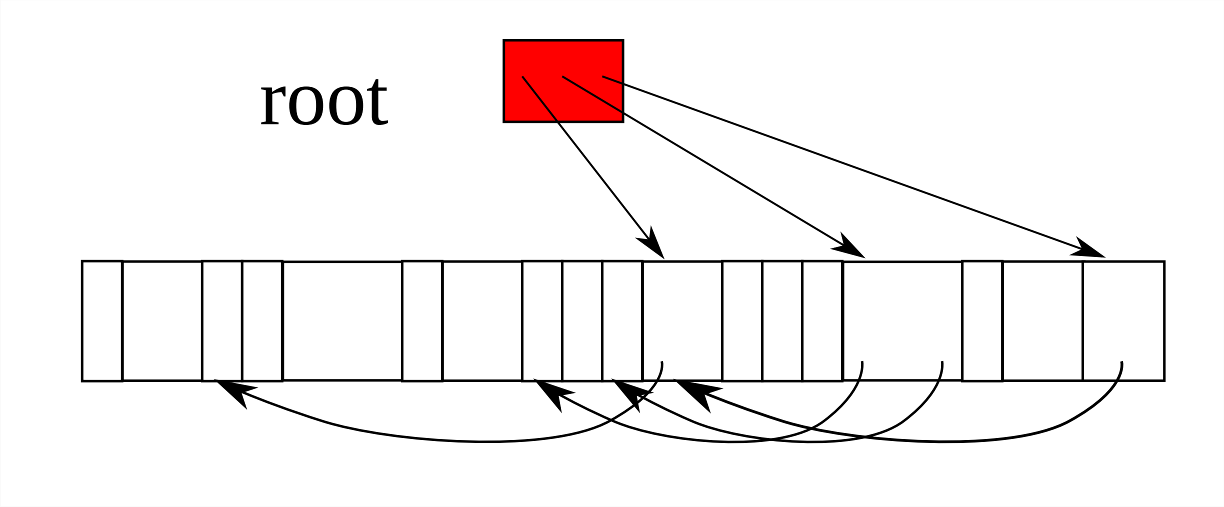
- mark&sweep GC: mark them as “visited”
- copying GC: copy them into a distinct (contiguous) region

Mark&sweep GC

- ***mark-phase:***
  - traverses objects from the root, *marking* encountered objects as *visited*
  - maintains ***mark stack***, the set of objects marked but whose children may have not been marked
- ***sweep phase:***
  - reclaims all memory blocks not visited in the mark phase
  - free memory blocks are not contiguous, so must be managed by an appropriate data structure ***(free lists)***

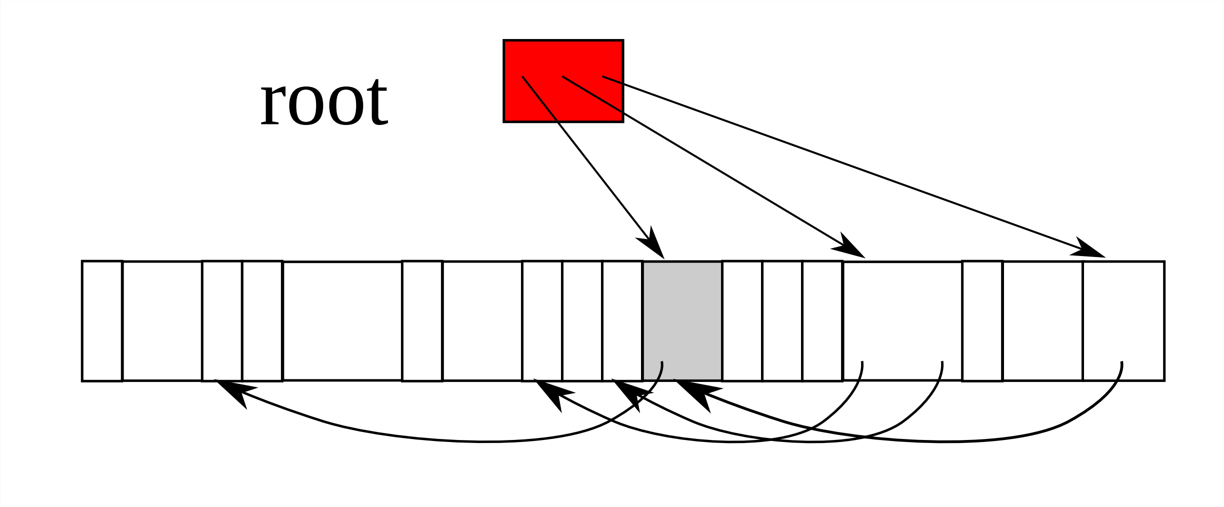
Mark&sweep GC in action

- mark stack (not shown in the figure) = light gray objects



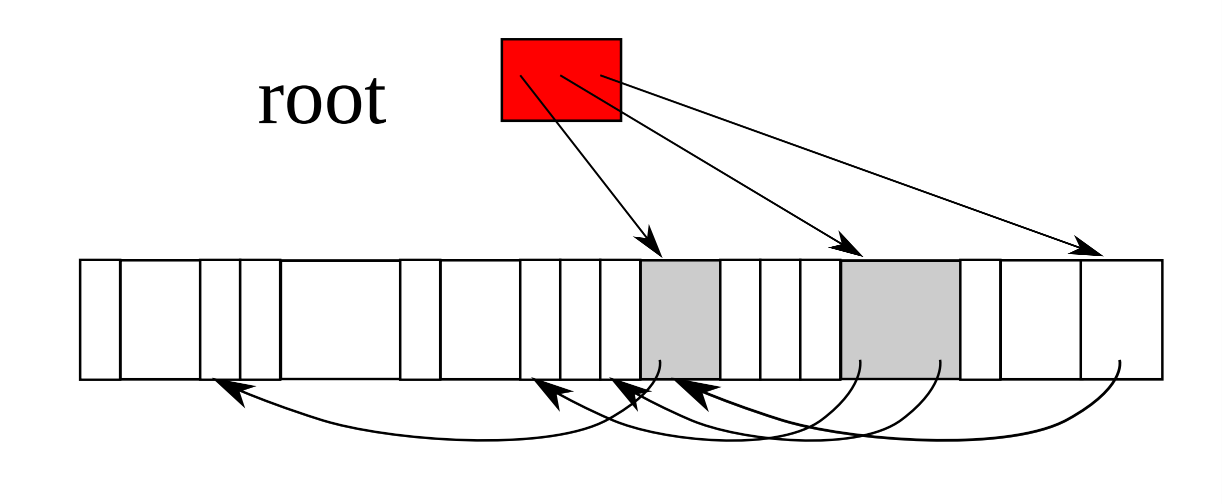
Mark&sweep GC in action

- mark stack (not shown in the figure) = light gray objects



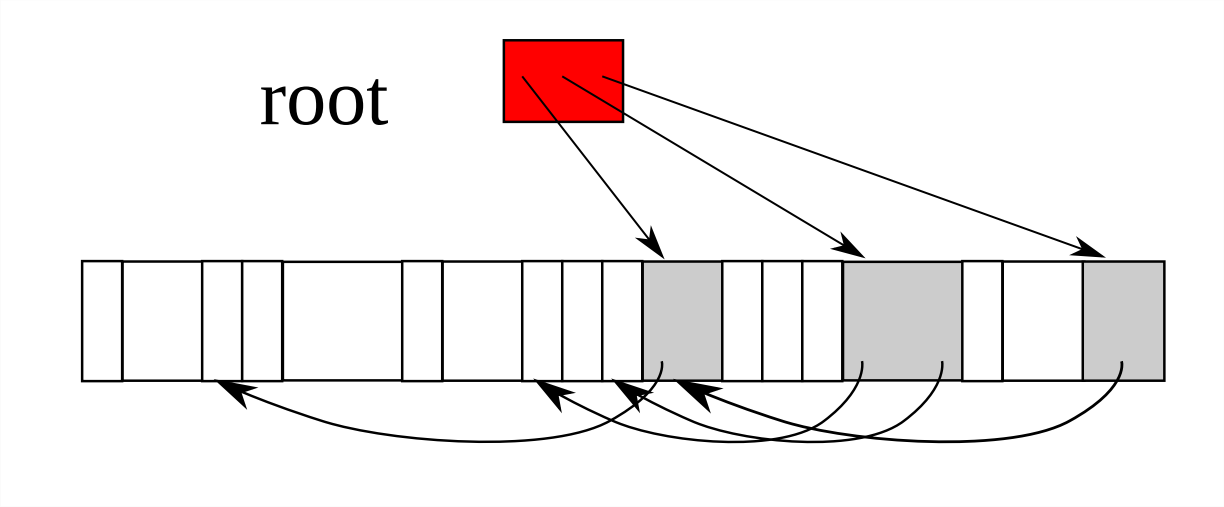
Mark&sweep GC in action

- mark stack (not shown in the figure) = light gray objects



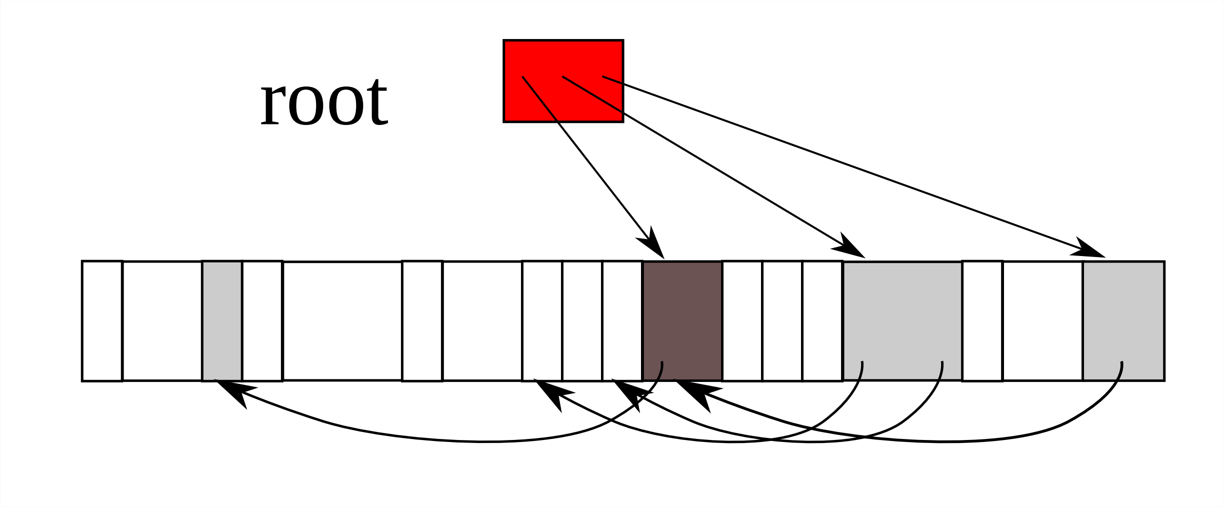
Mark&sweep GC in action

- mark stack (not shown in the figure) = light gray objects



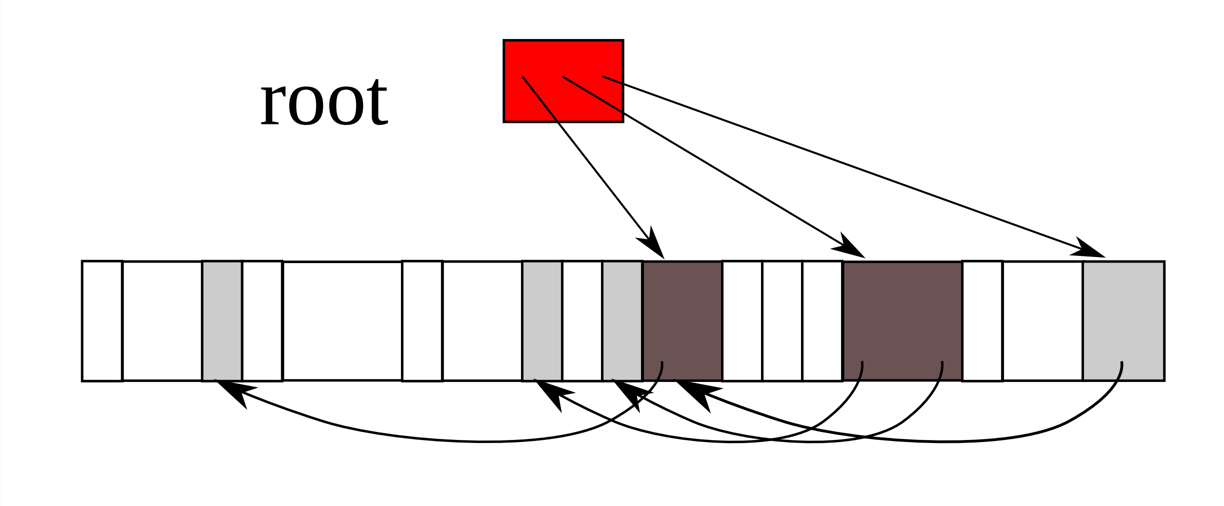
Mark&sweep GC in action

- mark stack (not shown in the figure) = light gray objects



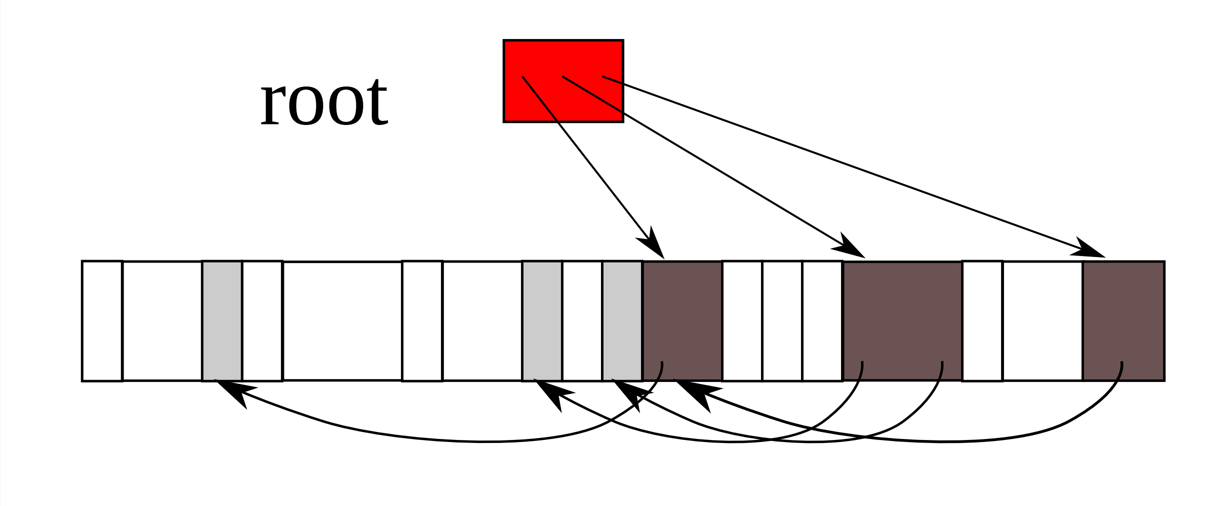
Mark&sweep GC in action

- mark stack (not shown in the figure) = light gray objects



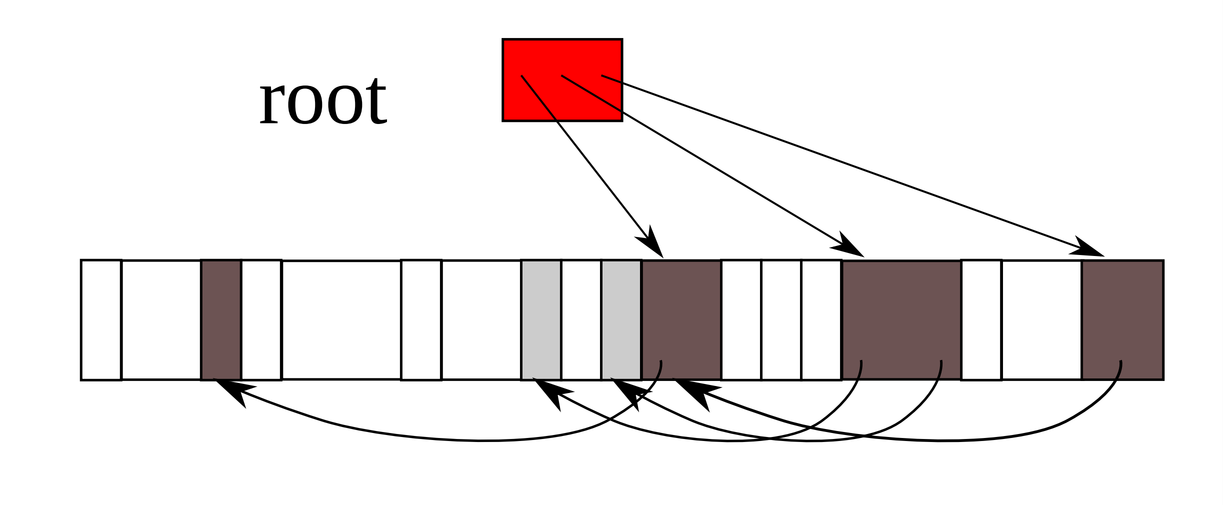
Mark&sweep GC in action

- mark stack (not shown in the figure) = light gray objects



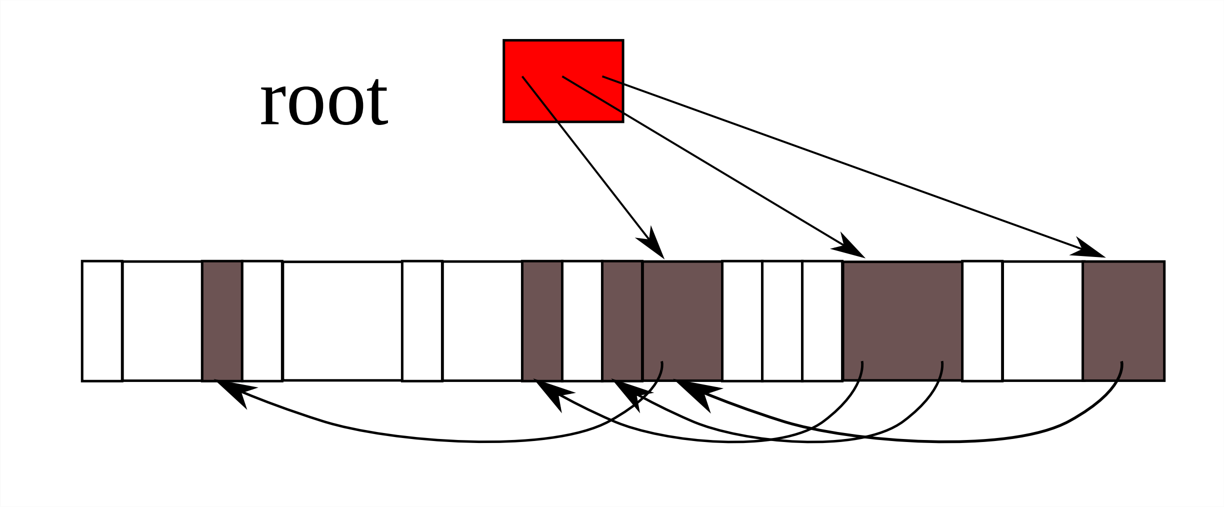
Mark&sweep GC in action

- mark stack (not shown in the figure) = light gray objects



Mark&sweep GC in action

- mark stack (not shown in the figure) = light gray objects

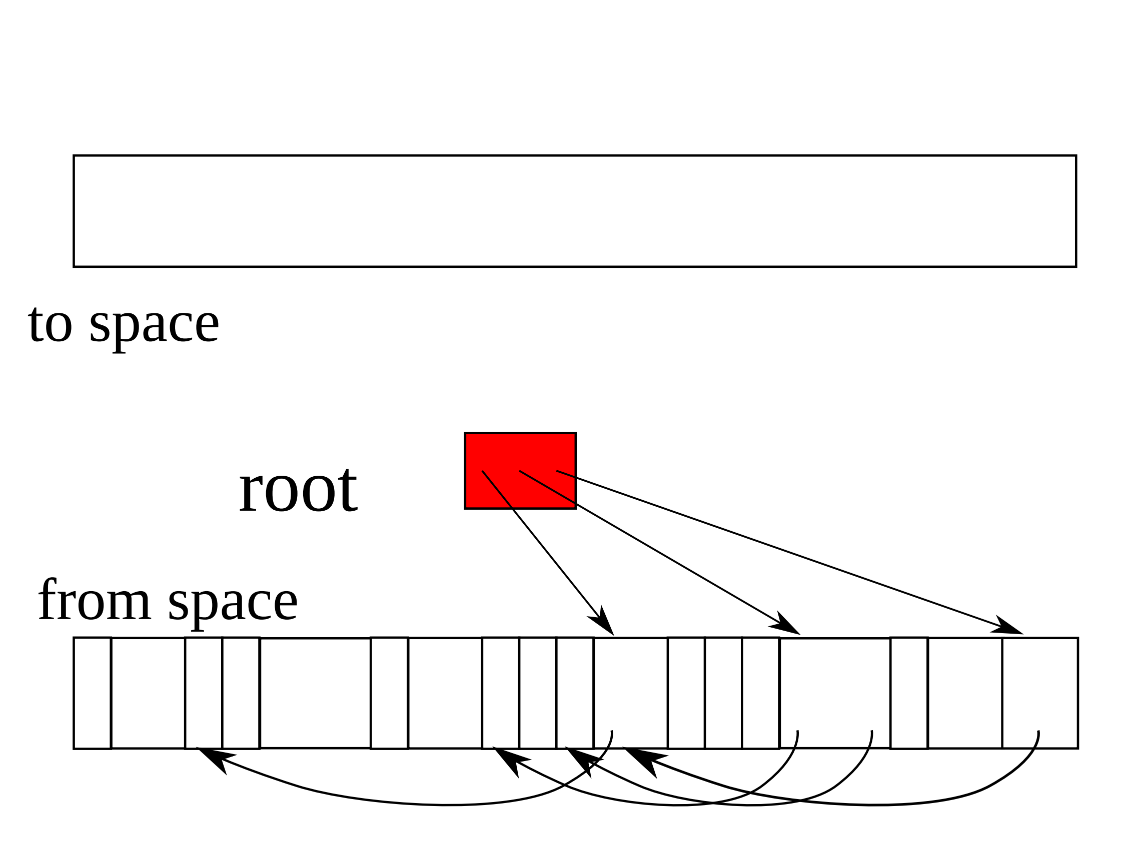


Copying GC

- $\approx$ copying a graph (possibly with merges and cycles)
- the crux : *pointers to the same object must remain the same after copy*
- ***semi-space GC:*** splits the available memory into equal-sized two spaces
  - when one space fills up, copy all reachable objects to the other

Simi-space copying GC in action

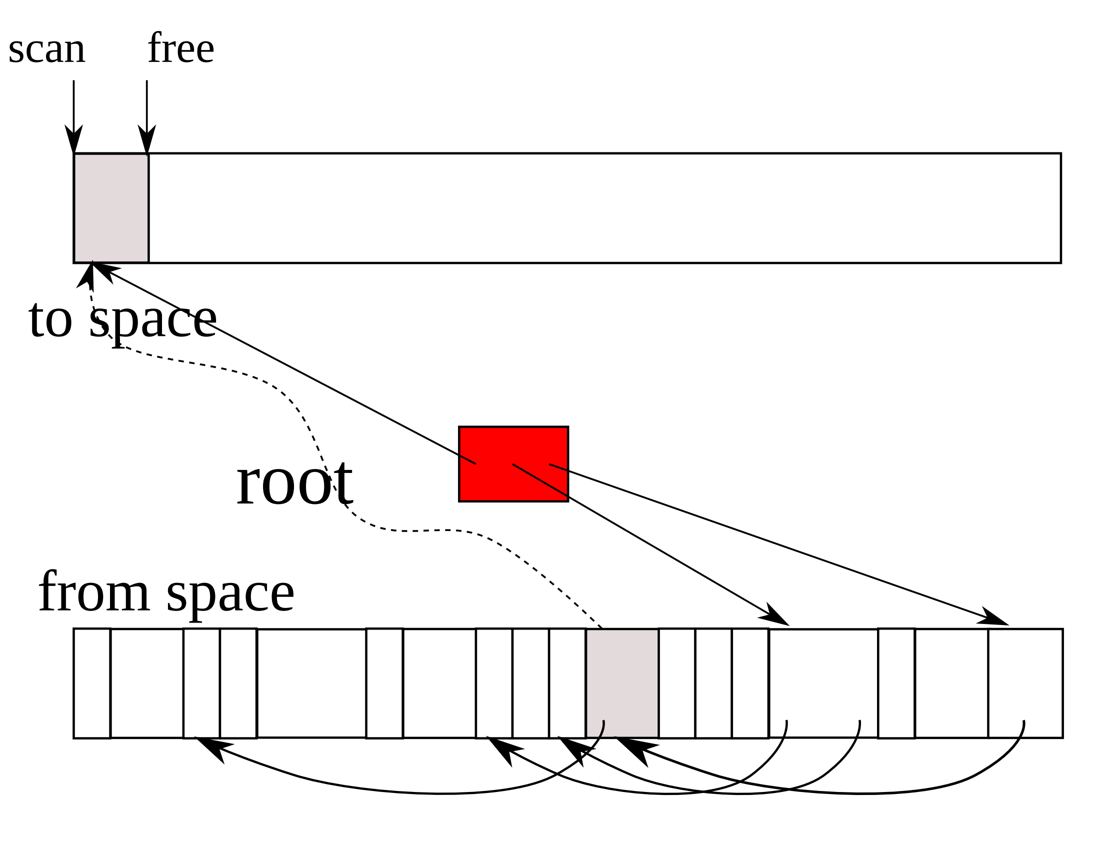
```
void *free, *scan;
void copy_gc() {
 free = scan = to_space;
 redirect_ptrs(root);
 while (scan < free) {
  redirect_ptrs(scan);
  scan += the size of object scan points to;
 }
 swap to_space and from_space;
}
void redirect_ptrs(void * o) {
 for (f : pointer fields of o) {
  if ([o->f] has been copied) {
   o->f = [o->f]'s forward pointer;
  } else {
   copy [o->f] to to free;
   [o->f]'s forward pointer = free;
   o->f = free;
   free += the size of [o->f];
  } } }
```



- $[p]$ is the object p points to (at address p)

Simi-space copying GC in action

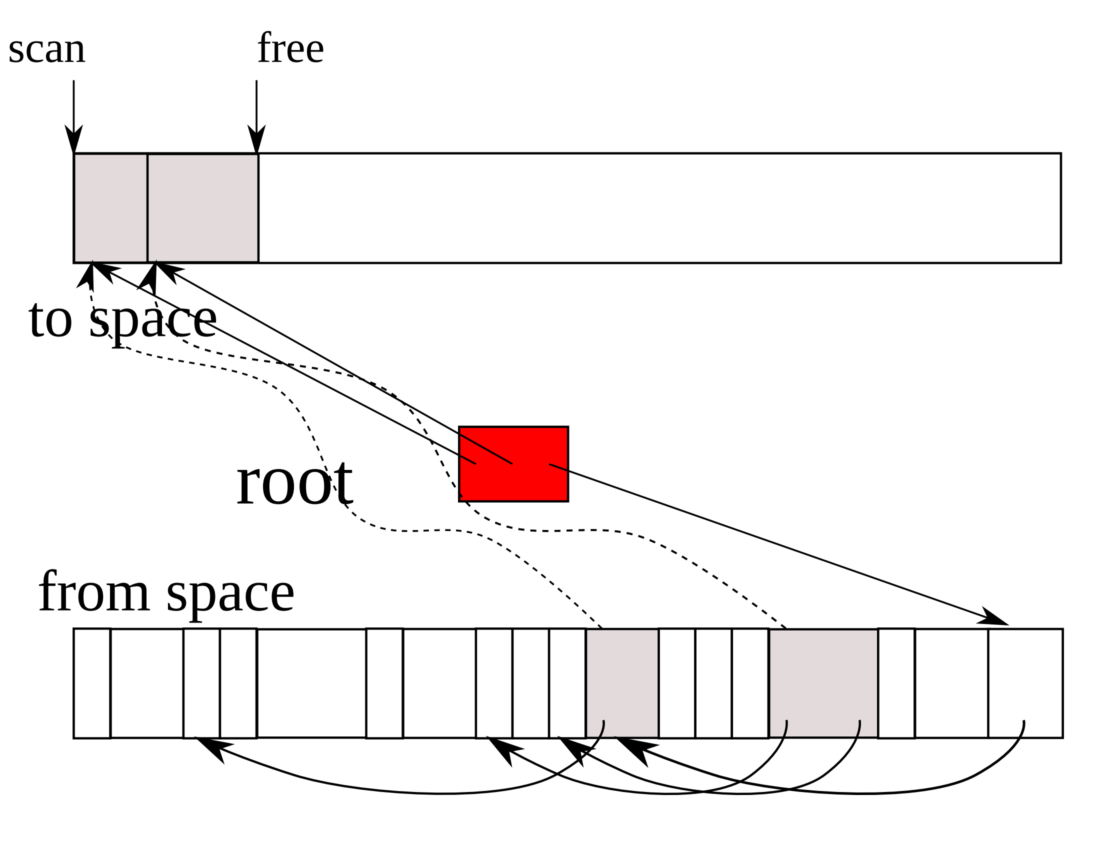
```
void *free, *scan;
void copy_gc() {
 free = scan = to_space;
 redirect_ptrs(root);
 while (scan < free) {
  redirect_ptrs(scan);
  scan += the size of object scan points to;
 }
 swap to_space and from_space;
}
void redirect_ptrs(void * o) {
 for (f : pointer fields of o) {
  if ([o->f] has been copied) {
   o->f = [o->f]'s forward pointer;
  } else {
   copy [o->f] to to free;
   [o->f]'s forward pointer = free;
   o->f = free;
   free += the size of [o->f];
  } } }
```



- $[p]$ is the object p points to (at address p)

Simi-space copying GC in action

```
void *free, *scan;
void copy_gc() {
 free = scan = to_space;
 redirect_ptrs(root);
 while (scan < free) {
  redirect_ptrs(scan);
  scan += the size of object scan points to;
 }
 swap to_space and from_space;
}
void redirect_ptrs(void * o) {
 for (f : pointer fields of o) {
  if ([o->f] has been copied) {
   o->f = [o->f]'s forward pointer;
  } else {
   copy [o->f] to to free;
   [o->f]'s forward pointer = free;
   o->f = free;
   free += the size of [o->f];
  } } }
```

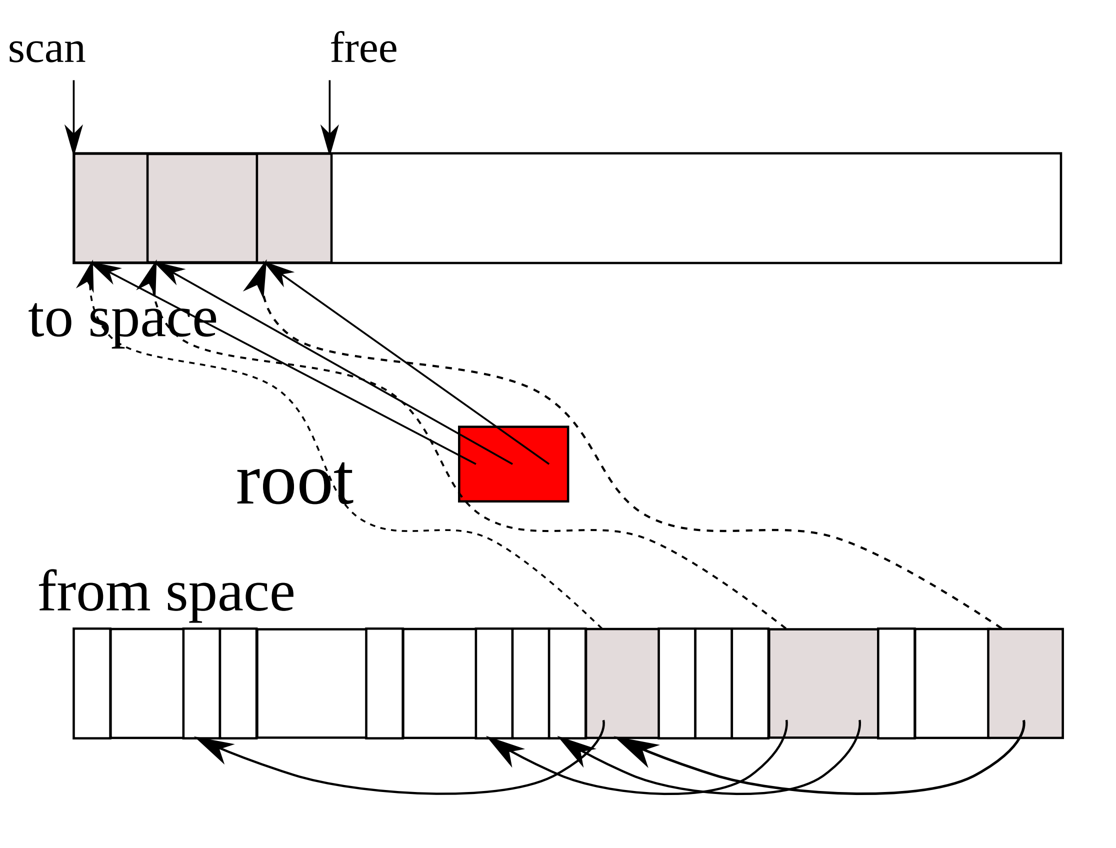


- $[p]$ is the object p points to (at address p)

Simi-space copying GC in action

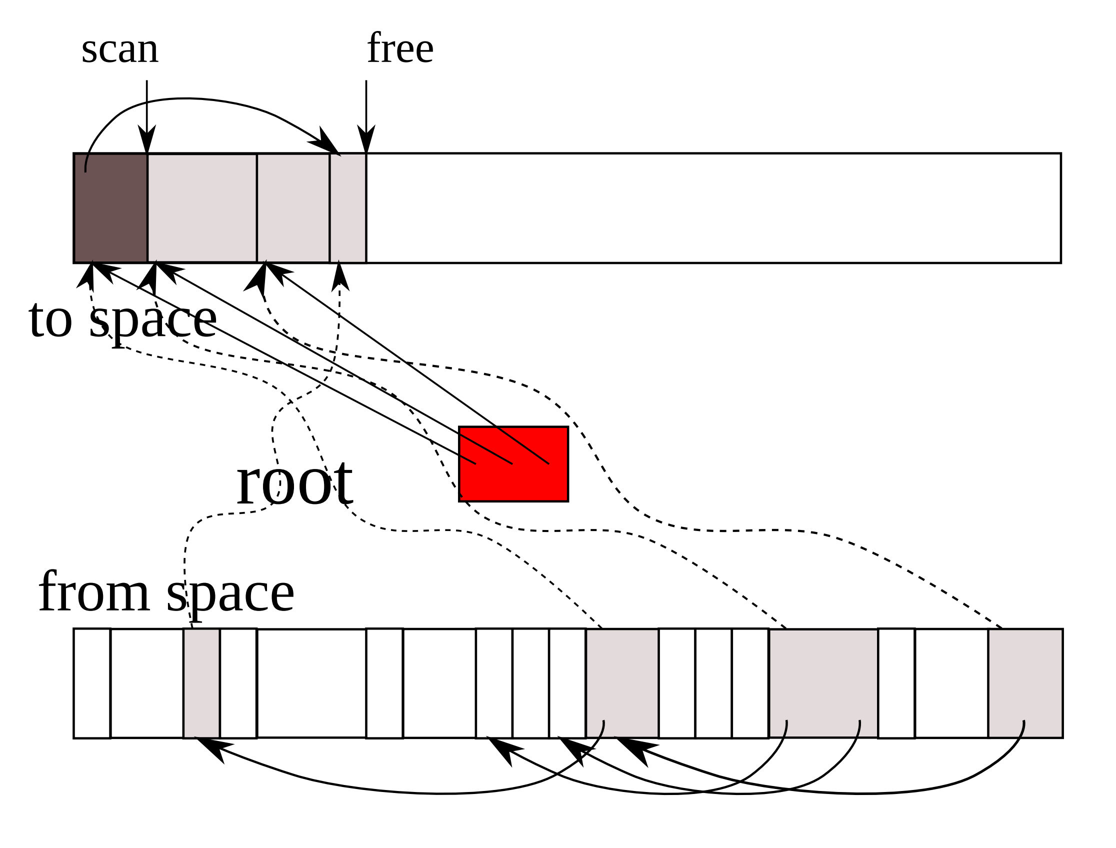
```
void *free, *scan;
void copy_gc() {
 free = scan = to_space;
 redirect_ptrs(root);
 while (scan < free) {
  redirect_ptrs(scan);
  scan += the size of object scan points to;
 }
 swap to_space and from_space;
}
void redirect_ptrs(void * o) {
 for (f : pointer fields of o) {
  if ([o->f] has been copied) {
   o->f = [o->f]'s forward pointer;
  } else {
   copy [o->f] to to free;
   [o->f]'s forward pointer = free;
   o->f = free;
   free += the size of [o->f];
  } } }
```

- $[p]$ is the object p points to (at address p)



Simi-space copying GC in action

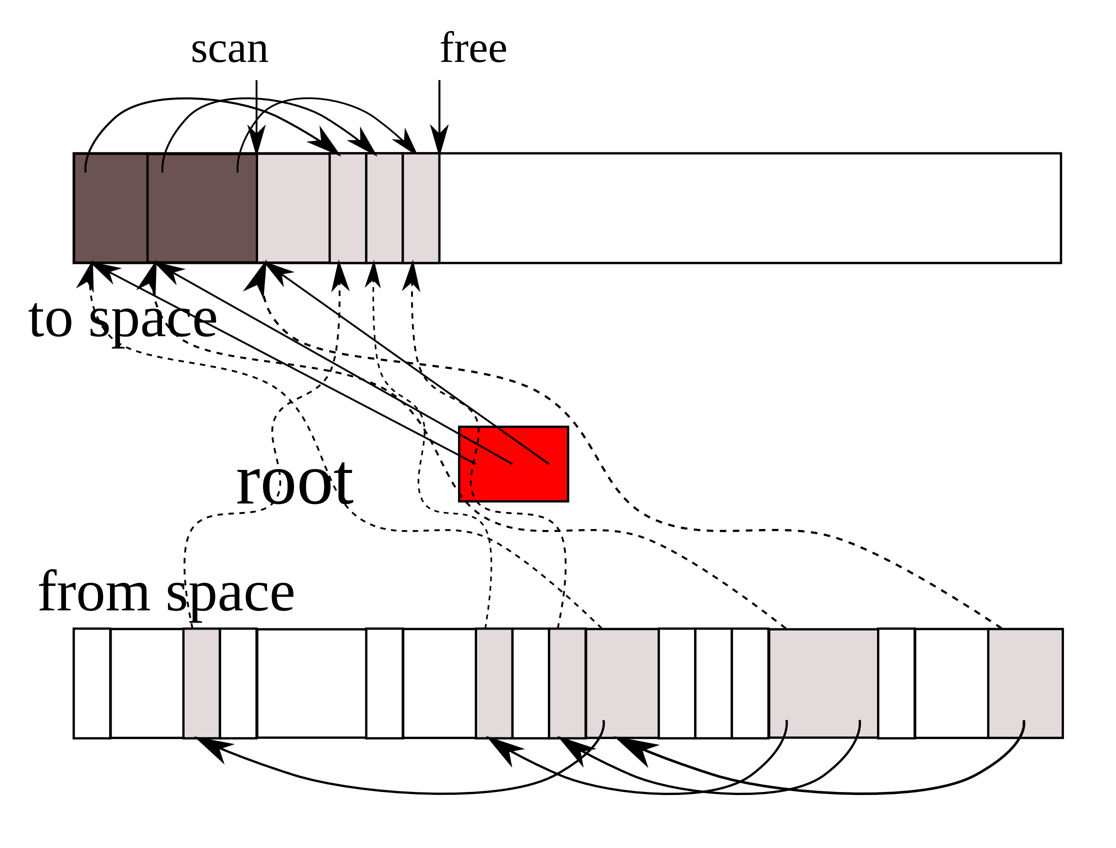
```
void *free, *scan;
void copy_gc() {
 free = scan = to_space;
 redirect_ptrs(root);
 while (scan < free) {
  redirect_ptrs(scan);
  scan += the size of object scan points to;
 }
 swap to_space and from_space;
}
void redirect_ptrs(void * o) {
 for (f : pointer fields of o) {
  if ([o->f] has been copied) {
   o->f = [o->f]'s forward pointer;
  } else {
   copy [o->f] to to free;
   [o->f]'s forward pointer = free;
   o->f = free;
   free += the size of [o->f];
  } } }
```



- $[p]$ is the object p points to (at address p)

Simi-space copying GC in action

```
void *free, *scan;
void copy_gc() {
 free = scan = to_space;
 redirect_ptrs(root);
 while (scan < free) {
  redirect_ptrs(scan);
  scan += the size of object scan points to;
 }
 swap to_space and from_space;
}
void redirect_ptrs(void * o) {
 for (f : pointer fields of o) {
  if ([o->f] has been copied) {
   o->f = [o->f]'s forward pointer;
  } else {
   copy [o->f] to to free;
   [o->f]'s forward pointer = free;
   o->f = free;
   free += the size of [o->f];
  } } }
```

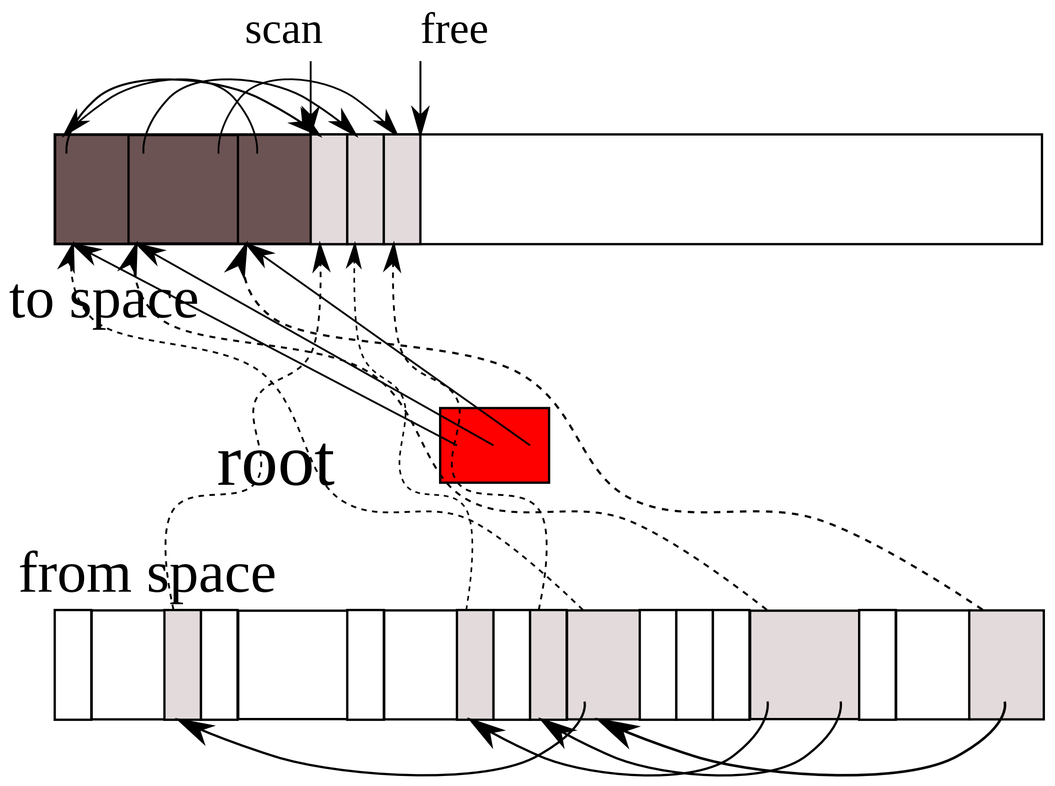


- $[p]$ is the object p points to (at address p)

Simi-space copying GC in action

```
void *free, *scan;
void copy_gc() {
 free = scan = to_space;
 redirect_ptrs(root);
 while (scan < free) {
  redirect_ptrs(scan);
  scan += the size of object scan points to;
 }
 swap to_space and from_space;
}
void redirect_ptrs(void * o) {
 for (f : pointer fields of o) {
  if ([o->f] has been copied) {
   o->f = [o->f]'s forward pointer;
  } else {
   copy [o->f] to to free;
   [o->f]'s forward pointer = free;
   o->f = free;
   free += the size of [o->f];
  } } }
```

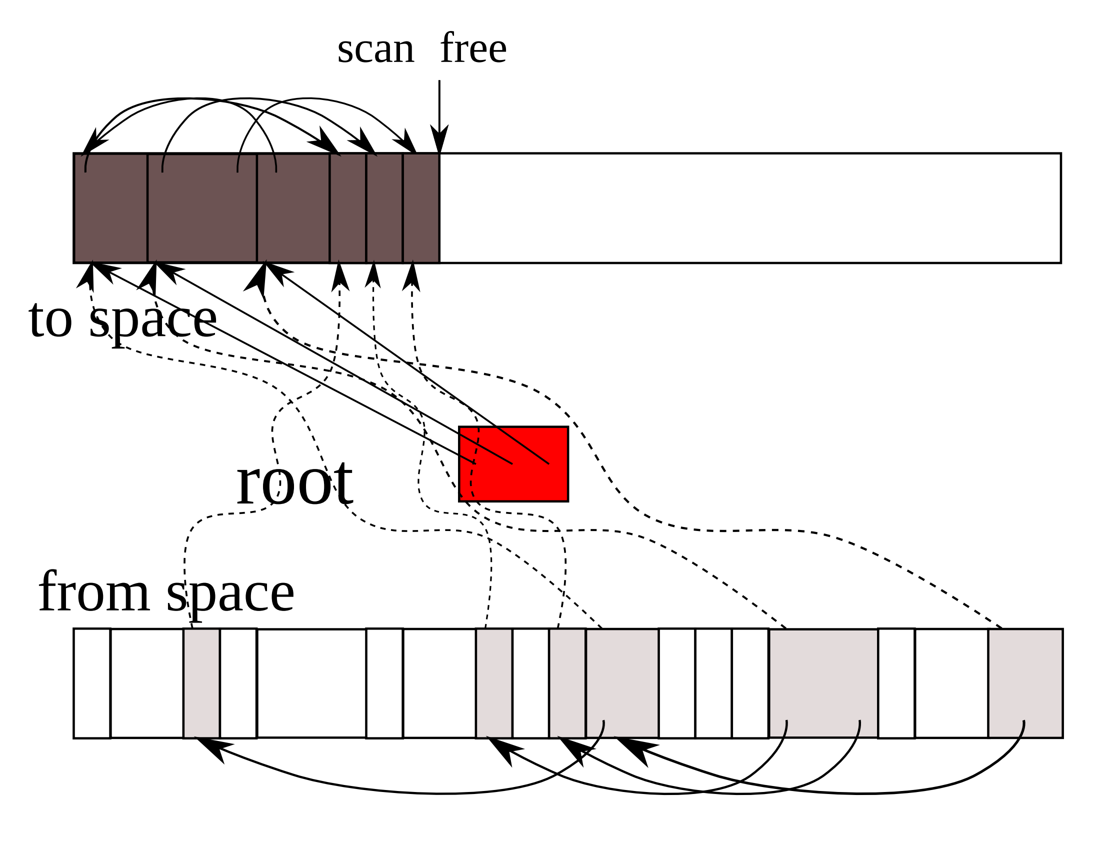
- $[p]$ is the object p points to (at address p)



Simi-space copying GC in action

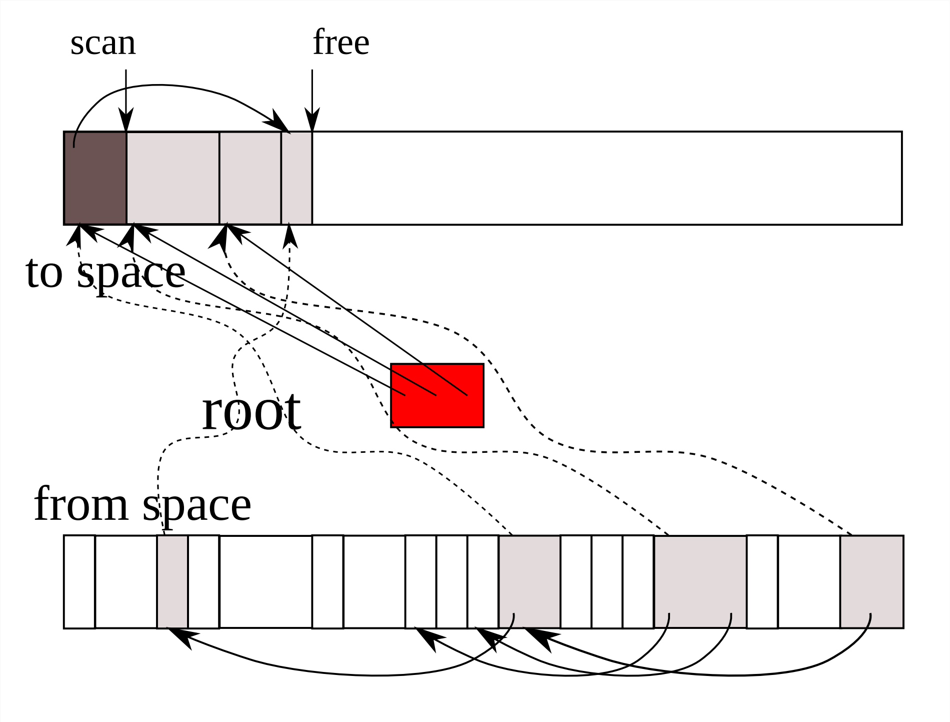
```
void *free, *scan;
void copy_gc() {
 free = scan = to_space;
 redirect_ptrs(root);
 while (scan < free) {
  redirect_ptrs(scan);
  scan += the size of object scan points to;
 }
 swap to_space and from_space;
}
void redirect_ptrs(void * o) {
 for (f : pointer fields of o) {
  if ([o->f] has been copied) {
   o->f = [o->f]'s forward pointer;
  } else {
   copy [o->f] to to free;
   [o->f]'s forward pointer = free;
   o->f = free;
   free += the size of [o->f];
  } } }
```

- $[p]$ is the object p points to (at address p)



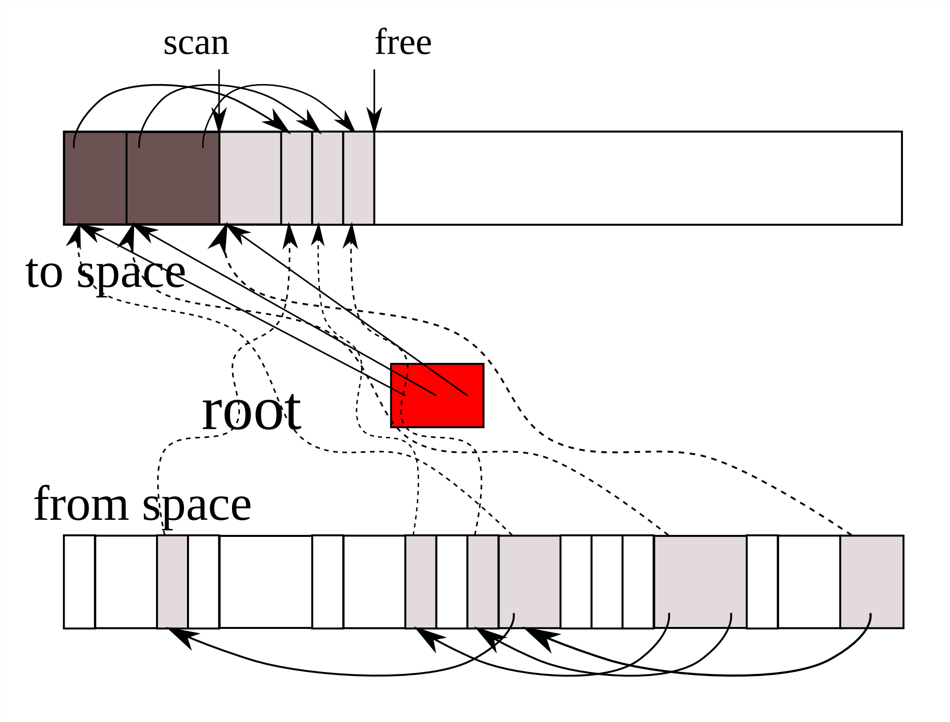
Simi-space copying GC : algorithm

- invariants
  - $p <$ free $\Rightarrow p$ has been visited
  - $p <$ scan $\Rightarrow p$ has been visited; so has its direct children
- area between scan and free serve a role similar to the mark stack



Simi-space copying GC : algorithm

- invariants
  - $p <$ free $\Rightarrow p$ has been visited
  - $p <$ scan $\Rightarrow p$ has been visited; so has its direct children
- area between scan and free serve a role similar to the mark stack



Mark&sweep vs. copying GC

- copying GC pros:
  - live objects occupy a contiguous region after a GC
  - $\rightarrow$ the free region becomes contiguous too
  - $\rightarrow$ *the overhead for memory allocation is small* (no need to “search” the free region)

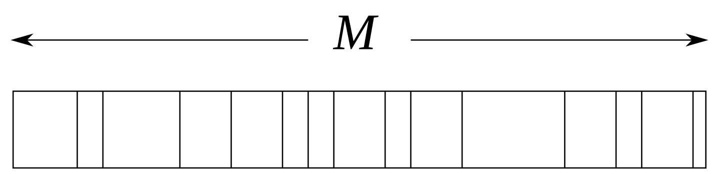
Mark&sweep vs. copying GC

- copying GC cons:
  - *copying is more expensive than marking*, obviously
  - *the free region must be reserved* to accommodate objects copied (low memory utilization)
    - must ensure “size of objects that may be copied” $\leq$ “size of the region to copy them into”
    - $\rightarrow$ “from space” = “to space” (semi-space)
  - pointers must be “precisely” distinguished from non-pointers (*ambiguous pointers are not allowed*)
    - $\because$ pointers are updated to the destinations of copies
    - a disaster occurs if you update non-pointers

Memory allocation cost of traversing GCs (mark-cons ratio)

Memory allocation cost of traversing GCs

- let's quantify *the cost of allocating a byte* including GC's work
- assume:
  - heap size (size of a semi-space in case of copying GC) $= M$
  - reached objects $= r$
  - assume for the sake of argument it's *always* r

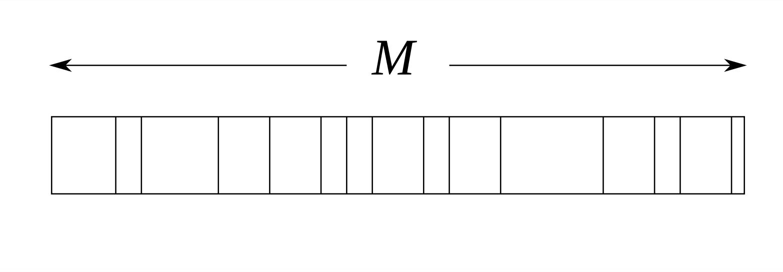


Memory allocation cost of traversing GCs

- behavior at equilibrium: the program repeats:

- a key observation:

the time (cost) of a single GC is roughly proportional to the amount of reached objects (i.e., $\propto r$)



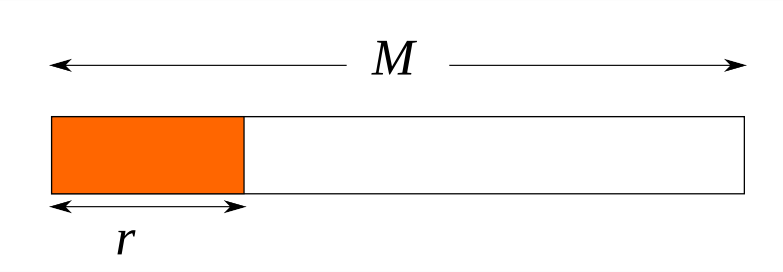
Memory allocation cost of traversing GCs

- behavior at equilibrium: the program repeats:

1. a GC occurs $\rightarrow$ *scan r bytes*, to make a free space of $(M - r)$bytes

- a key observation:

the time (cost) of a single GC is roughly proportional to the amount of reached objects (i.e., $\propto r$)



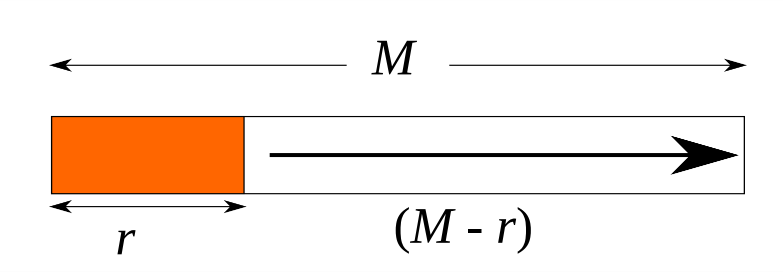
Memory allocation cost of traversing GCs

- behavior at equilibrium: the program repeats:

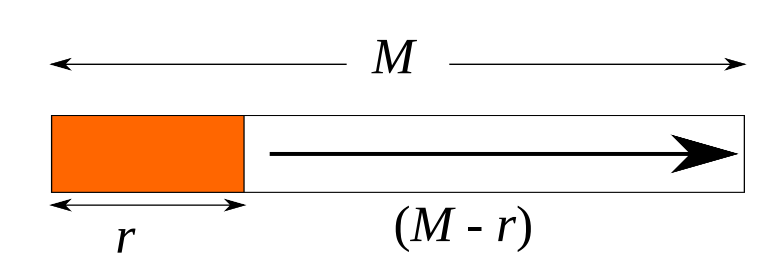
1. a GC occurs $\rightarrow$ *scan* r *bytes*, to make a free space of $(M - r)$bytes
2. *allocate* $(M - r)$ *bytes* without triggering a GC

- a key observation:

the time (cost) of a single GC is roughly proportional to the amount of reached objects (i.e., $\propto r$*)*



Memory allocation cost of traversing GCs



$$
\begin{aligned}
&\text{the cost of allocating a byte} \\
&= \alpha + \frac{\text{time spent on a GC}}{\text{space reclaimed by a GC}} \\
&= \alpha + \beta \frac{\text{space visited by a GC}}{\text{space reclaimed by a GC}} \\
&= \alpha + \beta \frac{r}{M - r}
\end{aligned}
$$

- α : an always-needed constant allocation cost per byte, even if you don't need to reclaim memory at all
- β : an average cost to examine a single byte
  - copy it (in a copying GC)
  - see if it is a pointer to an unvisited object

Note on copying GC vs mark-sweep GC

- the key observation the time (cost) of a single GC is roughly proportional to the amount of reached objects (i.e., $\propto r$) ignores the cost of so-called "sweep phase"
- a more accurate quantification will be

$$\text{the time (cost) of a single GC} \approx \beta r + \gamma(M - r),$$

- the second term reflects a constant cost to reclaim a byte

Note on copying GC vs mark-sweep GC

- i.e., the allocation cost per byte will be

$$\begin{aligned} & \alpha + \frac{\beta r + \gamma(M - r)}{M - r} \\ = & \alpha + \gamma + \beta \frac{r}{M - r} \end{aligned}$$

- $\Rightarrow$ the overall characterization of the cost is the same

Memory allocation cost of traversing GCs

- important fact:

$$\text{allocation cost per byte} = \text{constant} + \propto \frac{r}{M - r}$$

- $\frac{r}{M-r}$ is often called *mark-cons ratio*. Its origin:
  - *mark*: the amount of work to *mark* reachable objects
  - *cons*: the synonym of memory allocation in the ancient Lisp language, as in `(cons x y)`

Memory allocation cost of traversing GCs

$$\text{allocation cost per byte} = \text{constant} + \propto \frac{r}{M - r}$$

- r (primarily) depends only on app (not dependent on GCs)
  - note: may fluctuate depending on "when" GCs occur
- M is an *adjustable parameter (up to GC's choice)*
- M is large $\rightarrow$ the cost is small
  - $\rightarrow$ you can reduce the cost by making M (memory usage) larger
  - may sound obvious, but remember that what is important is the cost *per allocation (byte)*, not the frequency of GCs

How large do we make M (memory usage)?

- alright, the larger we make M, the smaller the cost becomes
  - how large should we make it in practice?
- we normally set M proportionally the *actual* data size

$$M \propto r$$

  i.e., choose a constant $k > 1$ and set:

$$M = kr$$

- a GC measures the amount of reachable objects to get r and sets M according to the above formula

How large do we make M (memory usage)?

- in this setting:
  - allocation cost per byte

$$= \text{mark-cons ratio} = \frac{r}{kr - r} = \frac{1}{k-1} \text{ (const)}$$

  - memory usage

$$\propto \text{the size of reachable objects at a point during execution}$$

- both are reasonable
  - most GCs allow you to set k (or M directly)
  - normally, $k = 1.5 \sim 2$, but it is worth knowing that you can reduce the cost by setting it large