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 ?

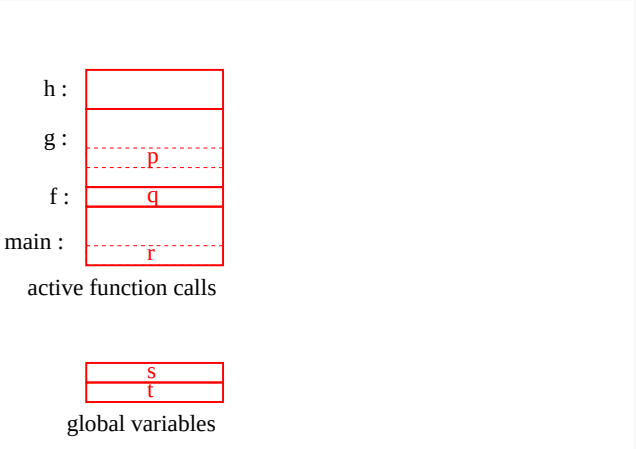
```
int * s, * t;
void h() { ... }

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 ?

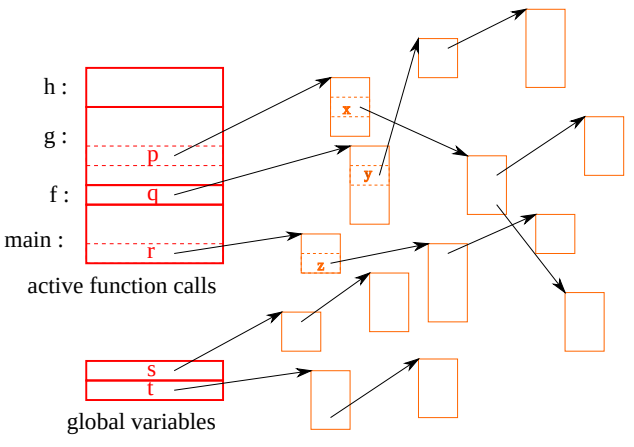
```
int * s, * t;
void h() { ... }

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

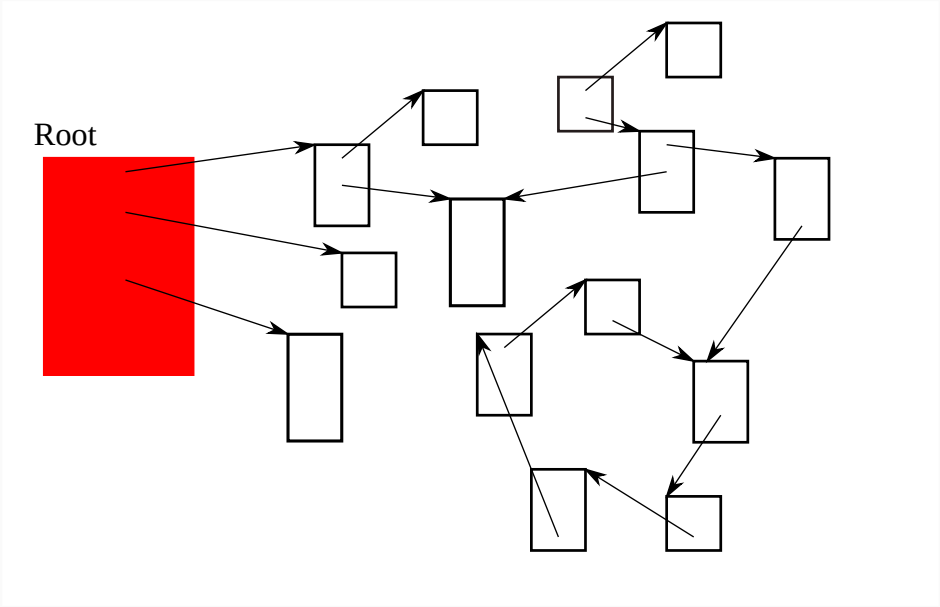
- simply traverses pointers from the root, to find (or *visit*) objects *reachable from the root*
- reclaim objects not visited
- two basic traversing methods
  - ***mark&sweep*** GC
  - ***copying*** GC

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*
  - *∵ an object's reference count is zero → it's unreachable from the root*
- note: "GC" sometimes narrowly refers to the traversing GC only

How traversing GC works

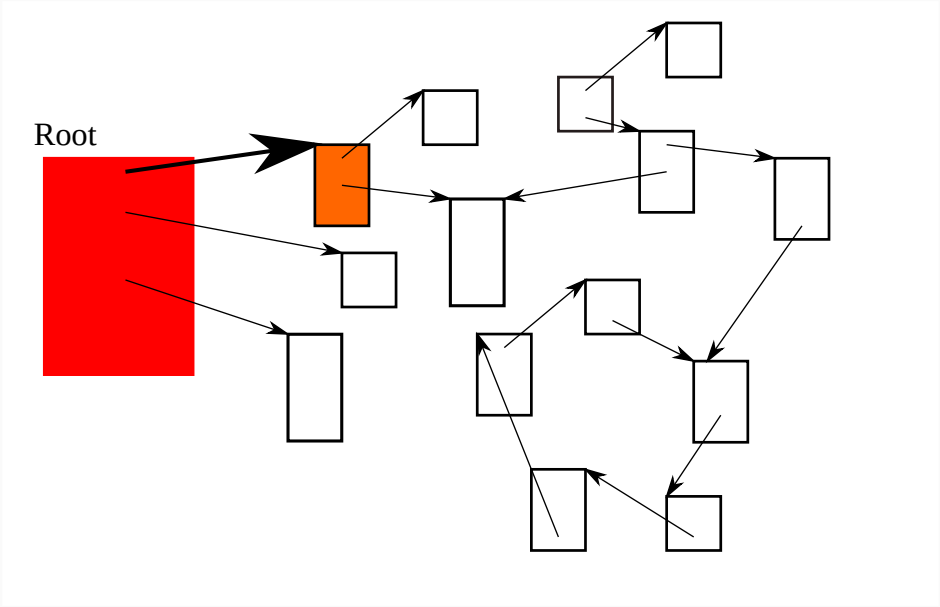
- traverse pointers from the root



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

How traversing GC works

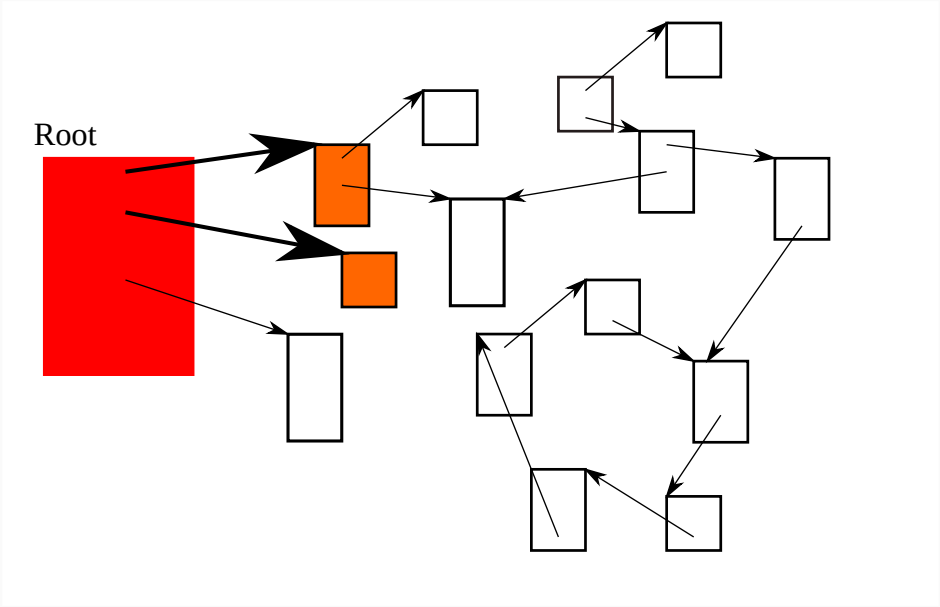
- traverse pointers from the root



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

How traversing GC works

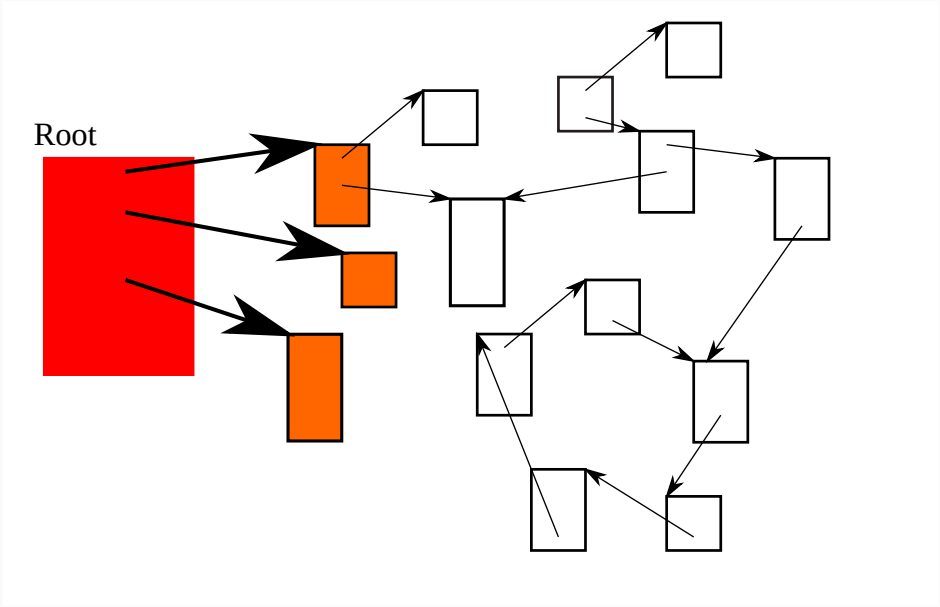
- traverse pointers from the root



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

How traversing GC works

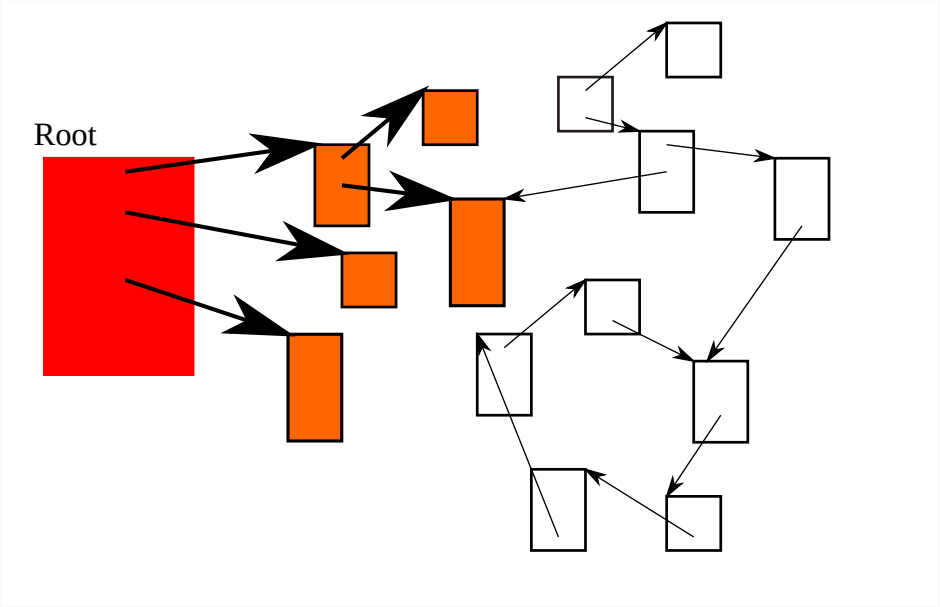
- traverse pointers from the root



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

How traversing GC works

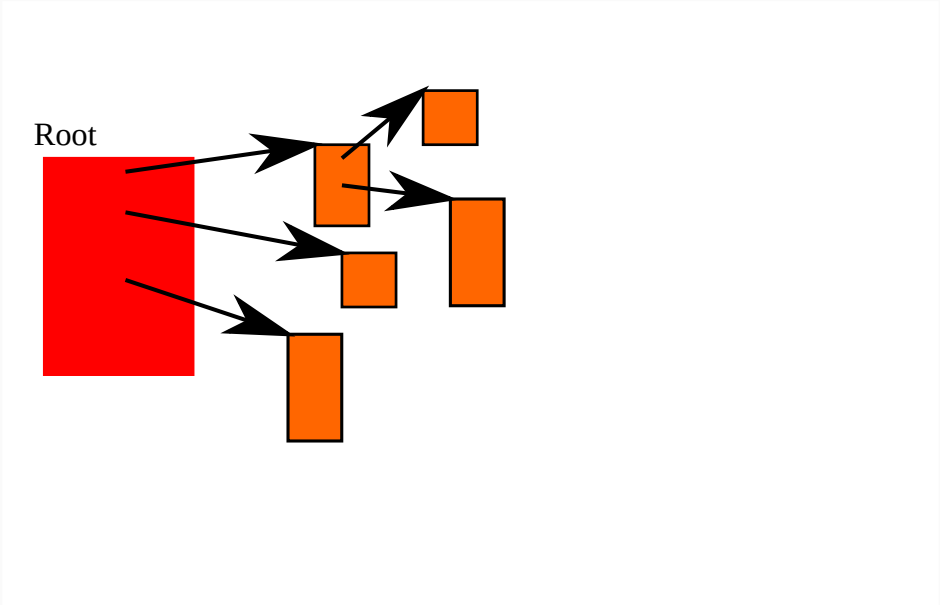
- traverse pointers from the root



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

How traversing GC works

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



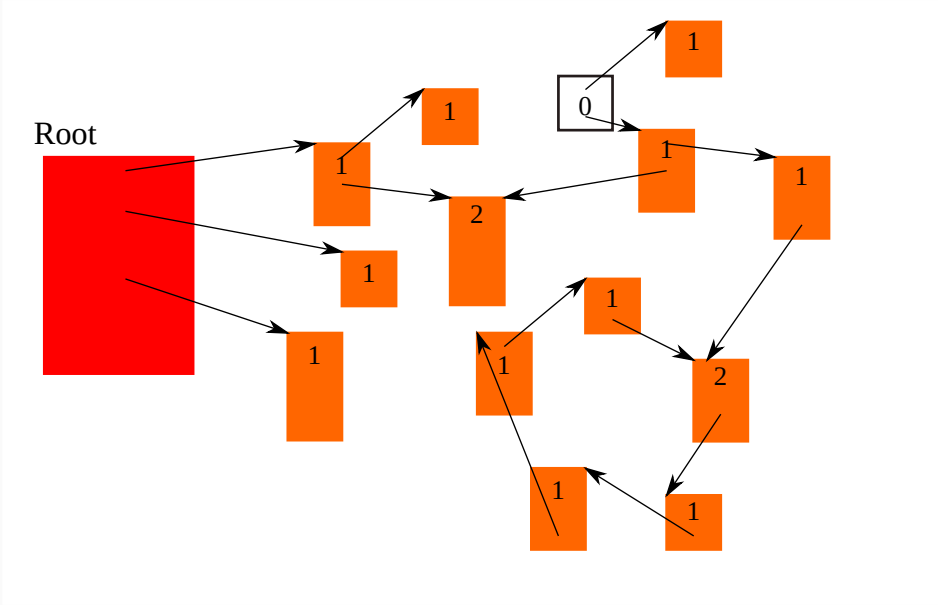
- note: the difference between mark&sweep and copying is covered later

How reference counting works

- each object has a reference count (RC)
- 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`

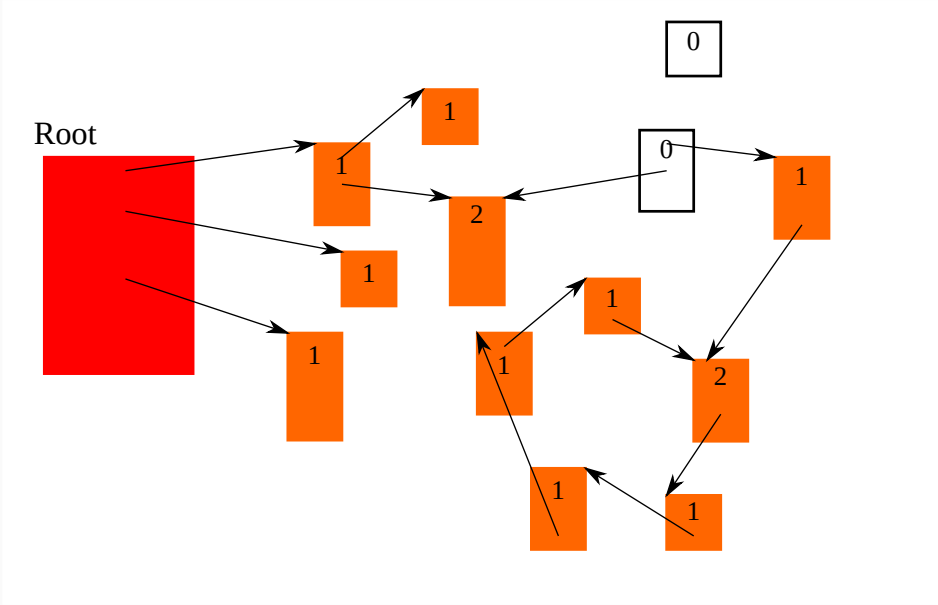
How reference counting works

- reclaim an object when its RC drops to zero



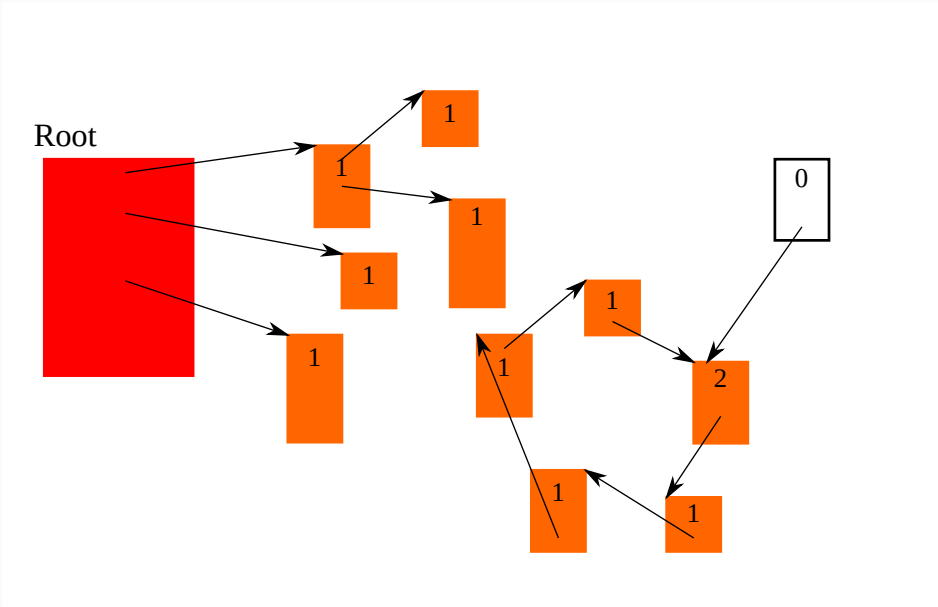
How reference counting works

- 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



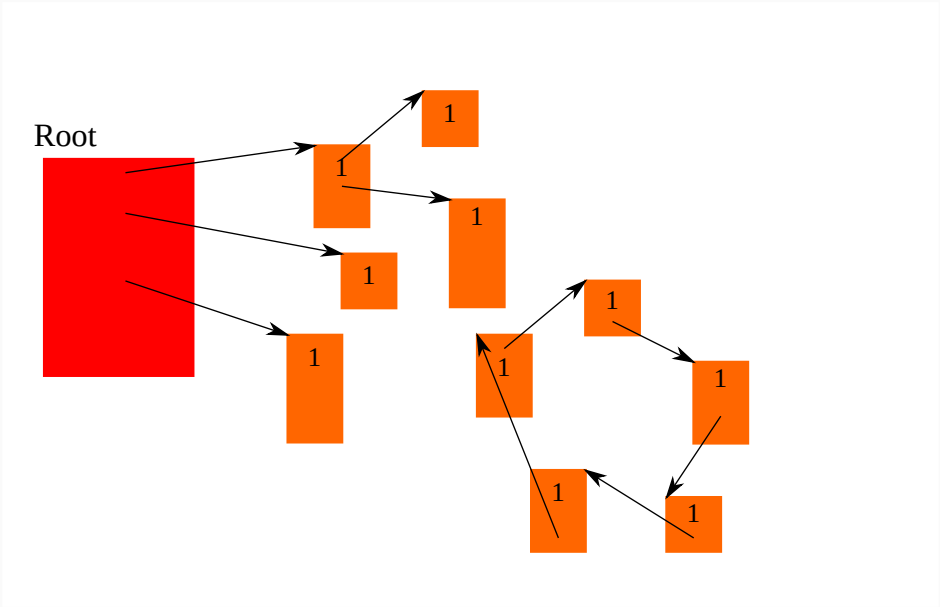
How reference counting works

- 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



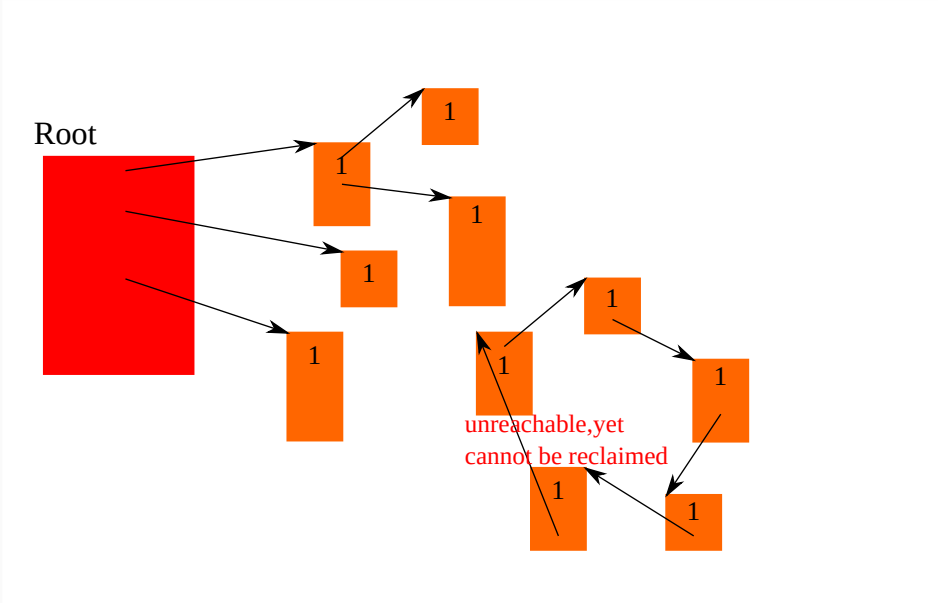
How reference counting works

- 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 */
}
```