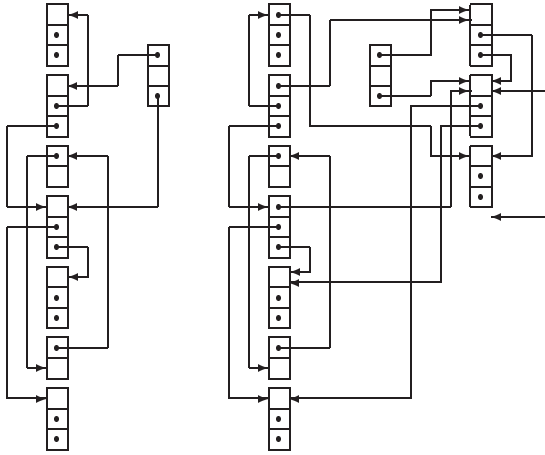


Garbage collection



A *garbage collector* is part of the run-time system: it reclaims heap-allocated records that are no longer used.

A garbage collector should:

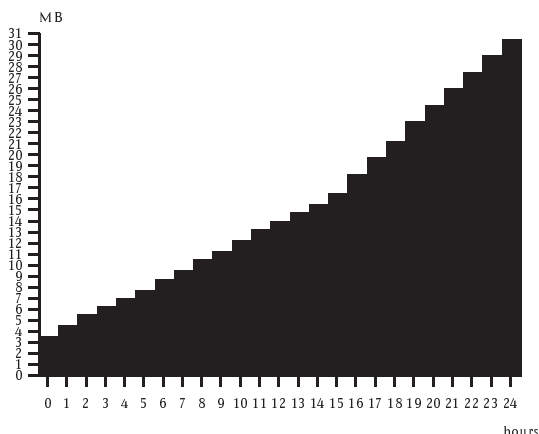
- reclaim *all* unused records;
- spend very little time per record;
- not cause significant delays; and
- allow all of memory to be used.

These are difficult and often conflicting requirements.

Life without garbage collection:

- unused records must be explicitly deallocated;
- superior if done correctly;
- but it is easy to miss some records; and
- it is dangerous to handle pointers.

Memory leaks in real life (ical v.2.1):



Which records are still in use?

Ideally, records that will never be accessed in the future execution of the program.

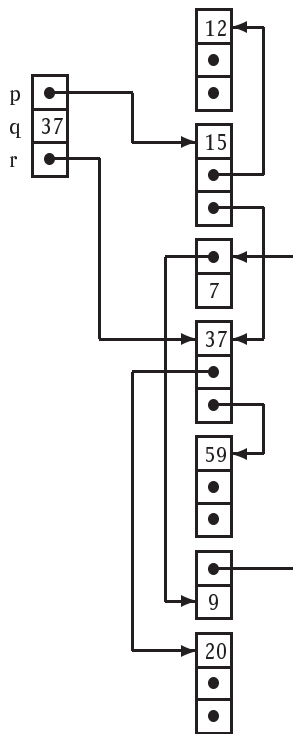
But that is of course undecidable...

Basic conservative assumption:

A record is *live* if it is reachable from a stack-based program variable.

Dead records may still be pointed to by other dead records.

A heap with live and dead records:



The mark-and-sweep algorithm:

- explore pointers starting from the program variables, and *mark* all records encountered;
- *sweep* through all records in the heap and reclaim the unmarked ones; also
- unmark all marked records.

Assumptions:

- we know the size of each record;
- we know which fields are pointers; and
- reclaimed records are kept in a *freelist*.

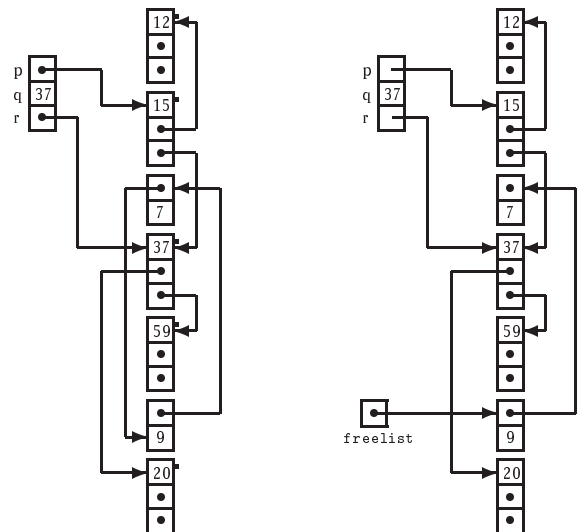
Pseudo code for mark-and-sweep:

```
function DFS( $x$ )
  if  $x$  is a pointer into the heap then
    if record  $x$  is not marked then
      mark record  $x$ 
      for  $i := 1$  to  $|x|$  do
        DFS( $x.f_i$ )
```

```
function Mark()
  for each program variable  $v$  do
    DFS( $v$ )
```

```
function Sweep()
   $p :=$  first address in heap
  while  $p <$  last address in heap do
    if record  $p$  is marked then
      unmark record  $p$ 
    else
       $p.f_1 :=$  freelist
      freelist :=  $p$ 
       $p := p + \text{sizeof}(\text{record } p)$ 
```

Marking and sweeping:



Analysis of mark-and-sweep:

- assume the heap has size H words; and
- assume that R words are reachable.

The cost of garbage collection is:

$$c_1 R + c_2 H$$

Realistic values are:

$$10R + 3H$$

The cost per reclaimed word is:

$$\frac{c_1 R + c_2 H}{H - R}$$

- if R is close to H , then this is expensive;
- the lower bound is c_2 ;
- increase the heap when $R > 0.5H$; then
- the cost per word is $c_1 + 2c_2 \approx 16$.

Other relevant issues:

- The DFS recursion stack could have size H (and has at least size $\log H$), which may be too much; however, the recursion stack can cleverly be embedded in the fields of marked records (pointer reversal).
- Records can be kept sorted by sizes in the freelist. Records may be split into smaller pieces if necessary.
- The heap may become *fragmented*: containing many small free records but none that are large enough.

The reference counting algorithm:

- maintain a counter of the references to each record;
- for each assignment, update the counters appropriately; and
- a record is dead when its counter is zero.

Advantages:

- is simple and attractive;
- catches dead records immediately; and
- does not cause long pauses.

Disadvantages:

- cannot detect cycles of dead records; and
- is much too expensive.

Pseudo code for reference counting:

function Increment(x)

$x.\text{count} := x.\text{count} + 1$

function Decrement(x)

$x.\text{count} := x.\text{count} - 1$

if $x.\text{count} = 0$ **then**

PutOnFreelist(x)

function PutOnFreelist(x)

Decrement($x.f_1$)

$x.f_1 := \text{freelist}$

$\text{freelist} := x$

function RemoveFromFreelist(x)

for $i := 2$ **to** $|x|$ **do**

Decrement($x.f_i$)

The stop-and-copy algorithm:

- divide the heap into two parts;
- only use one part at a time;
- when it runs full, copy live records to the other part;
- switch the roles of the two parts.

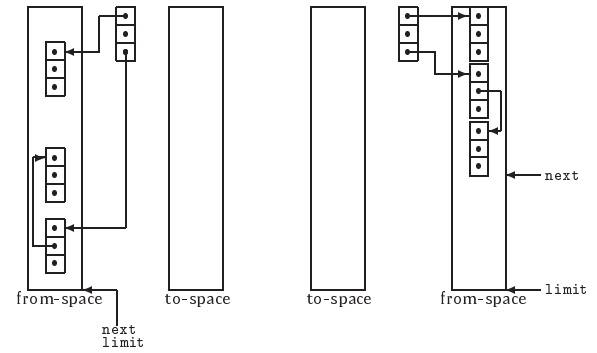
Advantages:

- allows fast allocation (no freelist);
- avoids fragmentation;
- collects in time proportional to R ; and
- avoids stack and pointer reversal.

Disadvantage:

- wastes half your memory.

Before and after stop-and-copy:



- next and limit indicate the available heap space; and
- copied records are contiguous in memory.

Pseudo code for stop-and-copy:

function Forward(p)

 if $p \in \text{from-space}$ then

 if $p.f_1 \in \text{to-space}$ then

 return $p.f_1$

 else

 for $i := 1$ to $|p|$ do

$\text{next}.f_i := p.f_i$

$p.f_1 := \text{next}$

$\text{next} := \text{next} + \text{sizeof}(\text{record } p)$

 return $p.f_1$

 else return p

function Copy()

$\text{scan} := \text{next} := \text{start of to-space}$

 for each program variable v do

$v := \text{Forward}(v)$

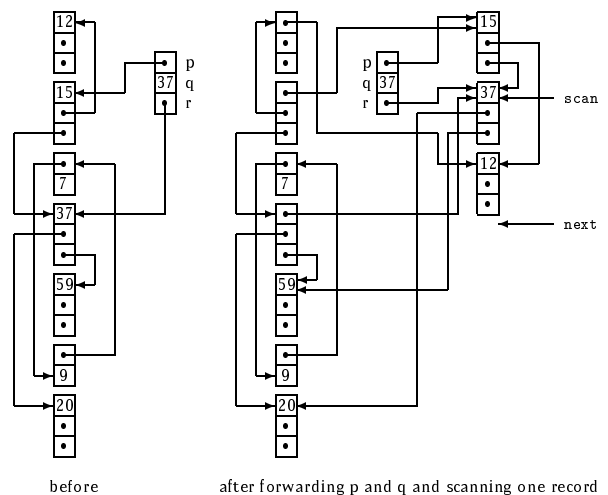
 while $\text{scan} < \text{next}$ do

 for $i := 1$ to $|\text{scan}|$ do

$\text{scan}.f_i := \text{Forward}(\text{scan}.f_i)$

$\text{scan} := \text{scan} + \text{sizeof}(\text{record scan})$

Snapshots of stop-and-copy:



Analysis of stop-and-copy:

- assume the heap has size H words; and
- assume that R words are reachable.

The cost of garbage collection is:

$$c_3 R$$

A realistic value is:

$$10R$$

The cost per reclaimed word is:

$$\frac{c_3 R}{\frac{H}{2} - R}$$

- this has no lower bound as H grows;
- if $H = 4R$ then the cost is $c_3 \approx 10$.

Earlier assumptions:

- we know the size of each record; and
- we know which fields are pointers.

For object-oriented languages, each record already contains a pointer to a class descriptor.

For general languages, we must sacrifice a few bytes per record.

We use mark-and-sweep or stop-and-copy.

But garbage collection is still expensive:
 ≈ 100 instructions for a small object!

Each algorithm can be further extended by:

- generational collection (to make it run faster); and
- incremental (or concurrent) collection (to make it run smoother).

Generational collection:

- observation: the young die quickly;
- hence the collector should focus on young records;
- divide the heap into generations:
 G_0, G_1, G_2, \dots ;
- all records in G_i are younger than records in G_{i+1} ;
- collect G_0 often, G_1 less often, and so on; and
- promote a record from G_i to G_{i+1} when it survives several collections.

How to collect the G_0 generation:

- roots are no longer just program variables but also pointers from G_1, G_2, \dots ;
- it might be very expensive to find those pointers;
- fortunately, they are rare; so
- we can try to remember them.

Ways to remember:

- maintain a list of all updated records (use marks to make this a set); or
- mark pages of memory that contain updated records (in hardware or software).

Incremental collection:

- garbage collection may cause long pauses;
- this is undesirable for interactive or real-time programs; so
- try to interleave the garbage collection with the program execution.

Two players access the heap:

- the *mutator*: creates records and moves pointers around; and
- the *collector*: tries to collect garbage.

Some invariants are clearly required to make this work.

The mutator will suffer some slowdown to maintain these invariants.