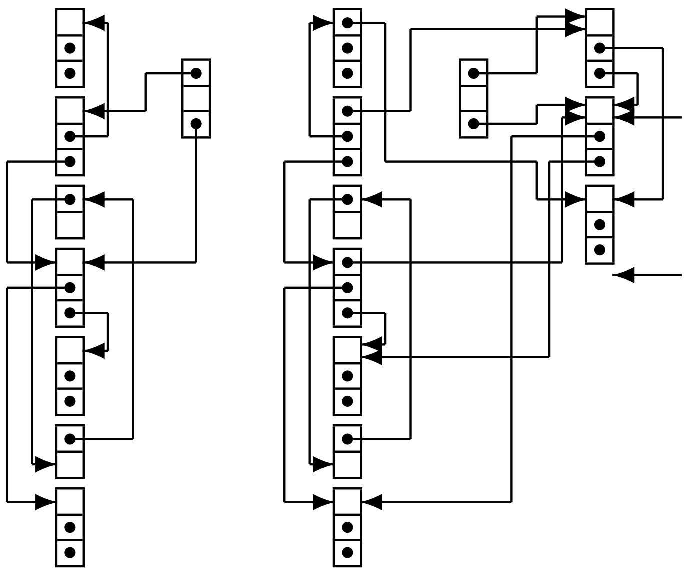
Garbage Collection

COMP 520: Compiler Design (4 credits)

Alexander Krolik

alexander.krolik@mail.mcgill.ca

MWF 13:30-14:30, MD 279



McCompiley

Announcements

Milestones:

- Milestone 1 grades returned
- Milestone 2 due **Friday, March 10th 11:59PM** on GitHub

Midterm:

- **Friday, March 17th**, either 13:00-14:30 or 13:30-15:00
- Watch for an email regarding room/time assignment later this week

Heap memory allocation:

- is very dynamic in nature:
  - unknown size;
  - unknown time;
- allows space to be allocated and deallocated as needed and in any order; and
- requires additional runtime support for managing the heap space.

A heap allocator (i.e. `malloc`):

- manages the memory in the heap space;
- takes as input an integer representing the size needed for the allocation;
- finds unallocated space in the heap large enough to accommodate the request; and
- returns a pointer to the newly allocated space.

Note: without runtime support it is now up to the *program* to return the memory when it is no longer needed (i.e. `free`).

You will find more details in an operating systems course

Deallocations can be either:

- manual: user code making the necessary decisions on what is live;
- continuous: runtime code determining on the spot which objects are live; or
- periodic: runtime code determining at specific times which objects are live.

Note: each mechanism has its own advantages/disadvantages. What are they?

When deallocations occur, we will assume the freed heap blocks are stored on a `freelist` (a linked list of heap blocks)

Manual deallocation mechanisms:

- leave programmers to determine when an object is no longer live; and
- require calls to a deallocator (i.e. `free`).

Consider the following code:

```
int *a = malloc(sizeof(int));
[...]
free(a);

*a = 5; // what happens?
```

Manual deallocations:

Advantages:

- reduces runtime complexity;
- gives the programmer full control on what is live; and
- can be more efficient in some circumstances.

Disadvantages:

- gives the programmer full control on what is live;
- requires extensive effort from the programmer;
- error-prone; and
- can be less efficient in some circumstances.

A *garbage collector*:

- is part of the runtime system;
- it automatically reclaims heap-allocated records that are no longer used.

A garbage collector should:

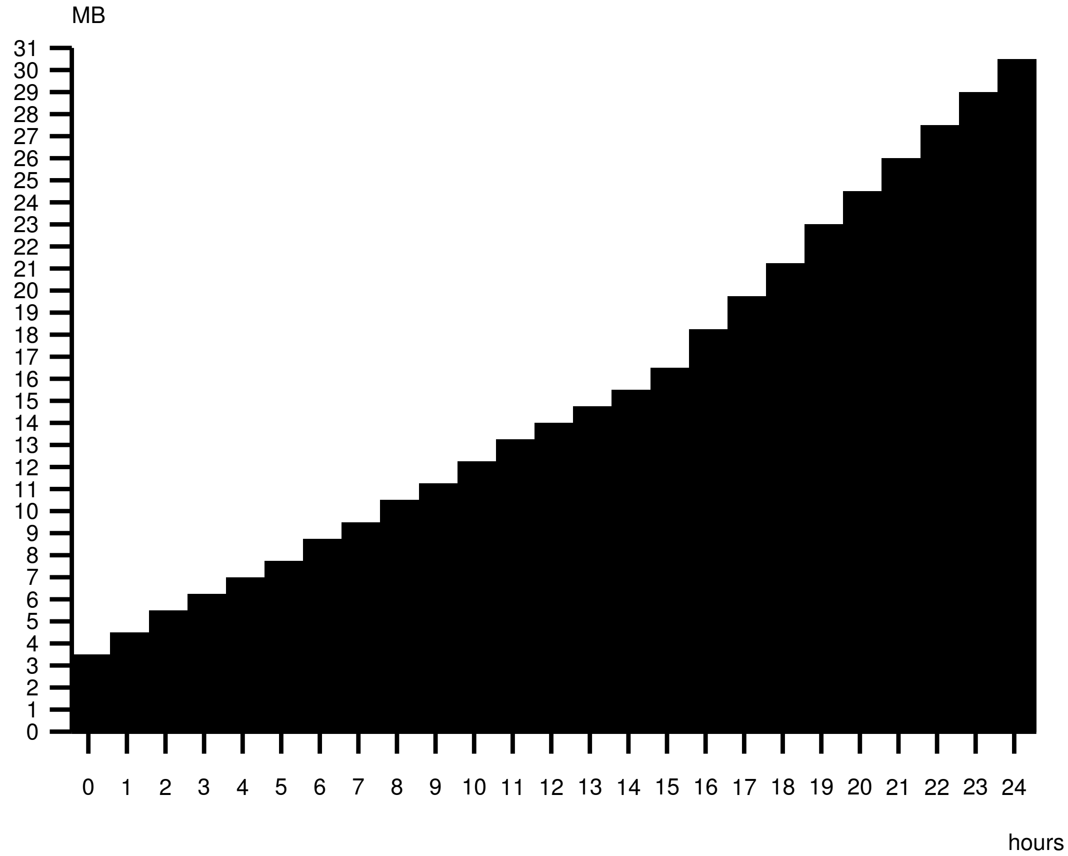
- reclaim *all* unused records;
- spend very little time per record;
- not cause significant delays; and
- allow all of memory to be used.

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 *dead*, i.e. no longer in use?

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

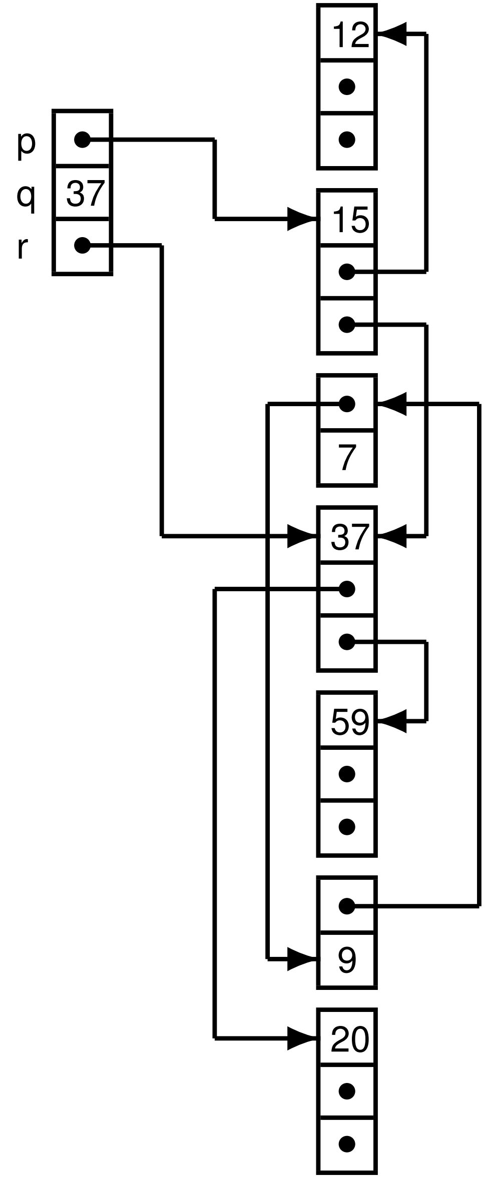
But that is of course undecidable...

Basic conservative assumption:

A record is *live* if it is reachable from a stack-based program variable (or global variable), otherwise dead.

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

A heap with live and dead records:



Reference counting:

- is a type of continuous (or incremental) garbage collection;
- uses a field on each object (the reference count) to track incoming pointers; and
- determines an object is dead when its reference count reaches zero.

The reference count is updated:

- whenever a reference is changed:
  - created

    e.g. `int *a = b; // b refcount++`
  - destroyed

    e.g. `a = c; // b refcount--`
- whenever a local variable goes out of scope;
- whenever an object is deallocated (all objects it points to have their reference counts decremented).

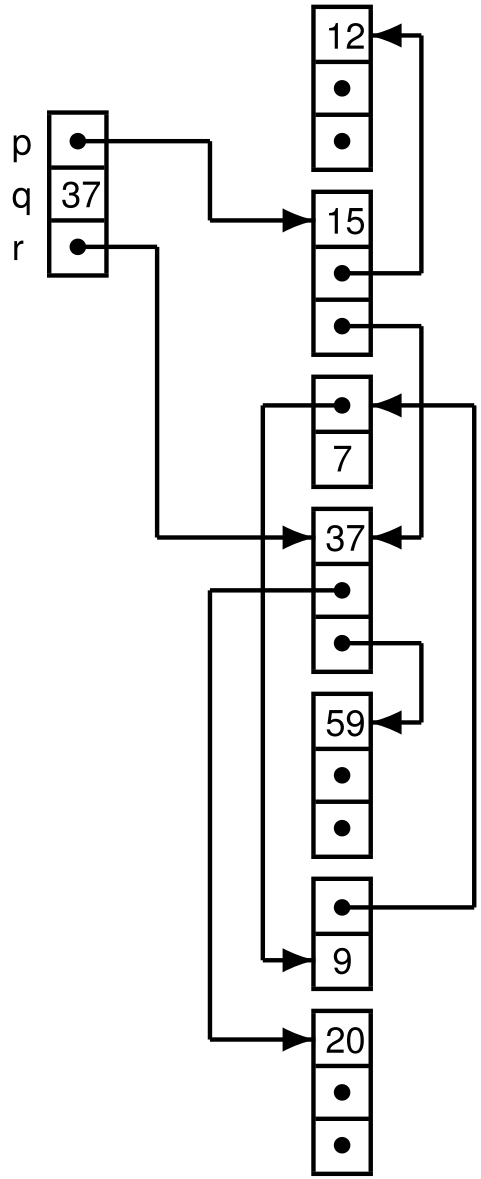
Pseudo code for reference counting:

function Increment(x)
    x.count := x.count$+1$

function Decrement(x)
    x.count := x.count-1
    if x.count=0 **then**
        Free(x)

function Free(x)
    for i:=1 **to** $|x|$ **do**
        Decrement($x.f_i$)
    $x.f_1$:= `freelist`
    `freelist` := x

Reference counting has one large problem:



What about objects 7 and 9?

Reference counting:

Advantages:

- is incremental, distributing the cost over a long period;
- catches dead objects immediately;
- does not require long pauses to handle deallocations; and
- requires no effort from the user.

Disadvantages:

- is incremental, slowing down the program continuously and unnecessarily;
- requires a more complex runtime system; and
- cannot handle circular data structures.

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`.

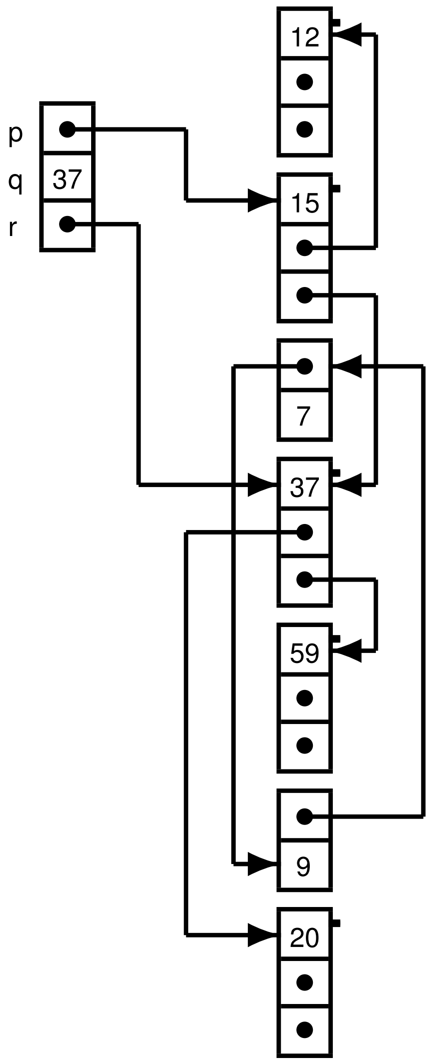
Pseudo code for mark-and-sweep:

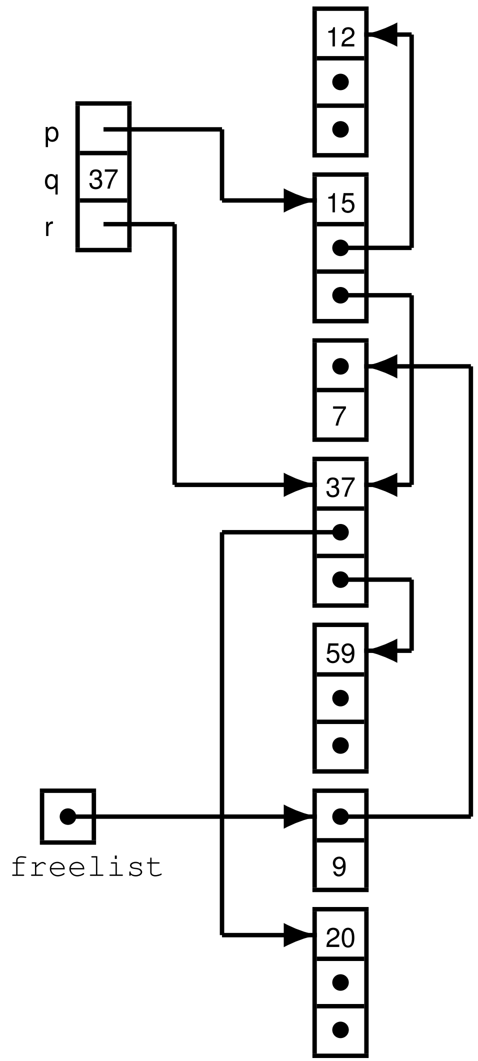
function DFS(x)
  if x is a pointer into the heap **then**
    if record x is not marked **then**
      mark record x
      for i:=1 **to** $|x|$ **do**
        DFS($x.f_i$)

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+sizeof(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.

To deal with fragmented heaps we use *compaction*:

- once mark-and-sweep has finished, collect all live objects are the beginning of the heap;
- adjust pointers pointing to all moved objects;
- the adjustment depends on the amount of space freed before the object;
- removes fragmentation and improves locality.

As we will see though, this is not possible in all programming languages due to the conservative nature of garbage collection.

Announcements

Welcome to spring =)

Milestones:

- Milestone 2 due **Sunday, March 12th 11:59PM** on GitHub
- Terminating statements

Midterm:

- **Friday, March 17th**, either 13:00-14:30 or 13:30-15:00
- Sign up `https://goo.gl/forms/ONXwSnPpKg2tkLbZ2`

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; and
- switch the roles of the two parts.

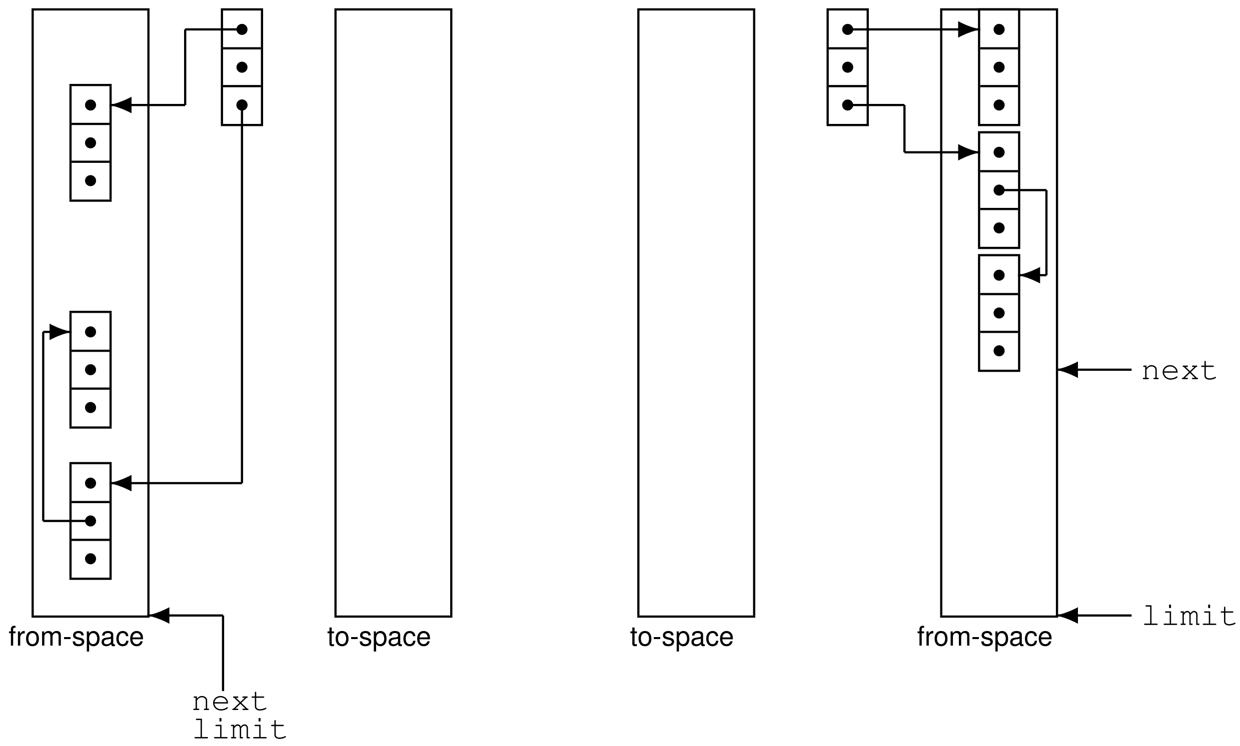
Advantages:

- allows fast allocation (no `freelist`);
- avoids fragmentation;
- collects in time proportional to $\boldsymbol{R}$; and
- avoids stack and pointer reversal.

Disadvantage:

- wastes half your memory.

Before and after stop-and-copy:



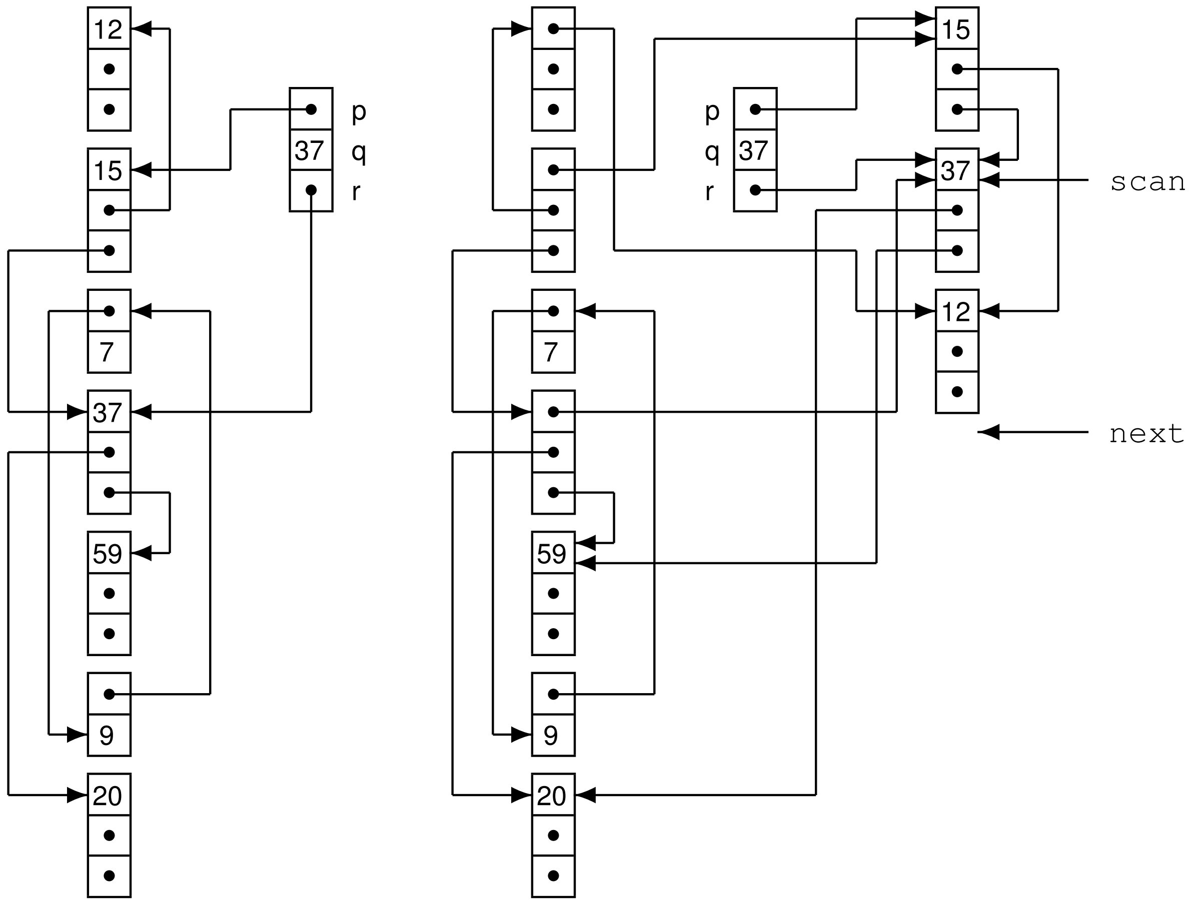
- `next` and `limit` indicate the available heap space; and
- copied records are contiguous in memory.

Pseudo code for stop-and-copy:

function Forward(p)
  if $p \in$ from-space **then**
    if $p.f_1 \in$ to-space **then**
      return $p.f_1$
    else
      for i:=1 **to** $|p|$ **do**
        `next`.f_i := $p.f_i$
      $p.f_1$:= `next`
      `next` := `next` + sizeof(record p)
      return $p.f_1$
  else return p

function Copy()
  `scan` := `next` := start of to-space
  for each program variable v **do**
    v := Forward(v)
  while `scan` $<$ `next` **do**
    for i:=1 **to** |`scan`| **do**
      `scan`.f_i := Forward(`scan`.f_i)
    `scan` := `scan` + sizeof(record `scan`)

Snapshots of stop-and-copy:



before

after forwarding p and q and scanning 1 record

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: $\approx$ 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: $G_0, G_1, G_2, \ldots$;
- 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:

- 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.