

Memory Management

In the beginning, there was FORTRAN:

- no dynamically allocated memory
- everything in static arrays with fixed limits
- a significant style as recently as 1983 (T_EX)

Dynamic allocation, explicit memory management:

- Pascal (new/dispose)
- C (malloc/free)
- C++ (new/delete)

Still an area for some research:

- efficiency
- locality (important for memory hierarchy)
- avoiding fragmentation

...many schemes

(Knuth, vol. 1; also Grunwald & Zorn)

The terror of `free`

When do you call `free`?

- must call sometime
 - `malloc` without `free` leads to “memory leak”
- premature call to `free` \Rightarrow dangling reference
 - hello, core dump!

The things we do...

- forget about `free`, just pray
- crufty interfaces — who must call `free`?
 - two ways to call every procedure?
- return pointers to static memory? **Ouch!**
- make copies of things
 - I own my copy, you own yours

We can do better...

Automatic Memory Management

Why not have `malloc` without `free`?

let memory be freed automatically

why not call it `cons`

Can be required in the language definition:

No Scheme object is ever destroyed. The reason that implementations of Scheme do not (usually!) run out of storage is that they are permitted to reclaim the storage occupied by an object if they can prove that the object cannot possibly matter to any future computation.

Plan of study:

- see how garbage is created
- see how we identify it
- discuss ways of reclaiming it

What is garbage?

A cell is garbage if

- its contents **can't affect any future (legal) computation**

As an approximation, we call a cell garbage if

- it is not ***reachable*** from a ***root set***

Any cell that is not garbage is ***live data***

- live data might affect a future computation

What values can affect a computation?

We divide these values into two parts.

Roots:

- local variables of procedures
- parameters of procedures
- global variables

Other values reachable from the roots:

- what if a formal parameter is a cons cell?

Tracing pointers

Confusing pointer & integer could cause leaks

- suppose integer n is address of an object?

Two major approaches

- Type tagging:
 - each type of heap object gets unique tag
 - tag leads to descriptive info
 - compiler puts “which fields are pointers” in desc

Example: Modula-3

- Pointer tagging:
 - use special bits to distinguish pointers, integers
 - “LISP machines” had special “tag bits” on stock hardware, loses a bit from integer space
 - e.g., all integers have low bit set
 - heap object descriptors only show size
 - some objects are ‘non-pointer-containing’
 - e.g., strings

Example: Standard ML of New Jersey

Two families of garbage collectors

Using “start at roots, follow pointers” approach:

- mark-and-sweep collection
 1. unmark all objects
 2. trace pointers, marking all live data
 3. “sweep away” unmarked objectsunmarked objects moved to “free list”,
reallocated
- copying collection: works with *two*
“**semi-spaces**”
all objects, some free space in “from-space”
“to-space” is empty
from-space full? **copy live data to to-space**
to-space is *not* full (not all objects copied)
now objects, free space are in to-space
“**flip**”

The evil third family

Using heavy compiler support: **reference counting**

if no pointer to an object, it must be garbage
keep in each object a “reference count”

number of references (ptrs) to object

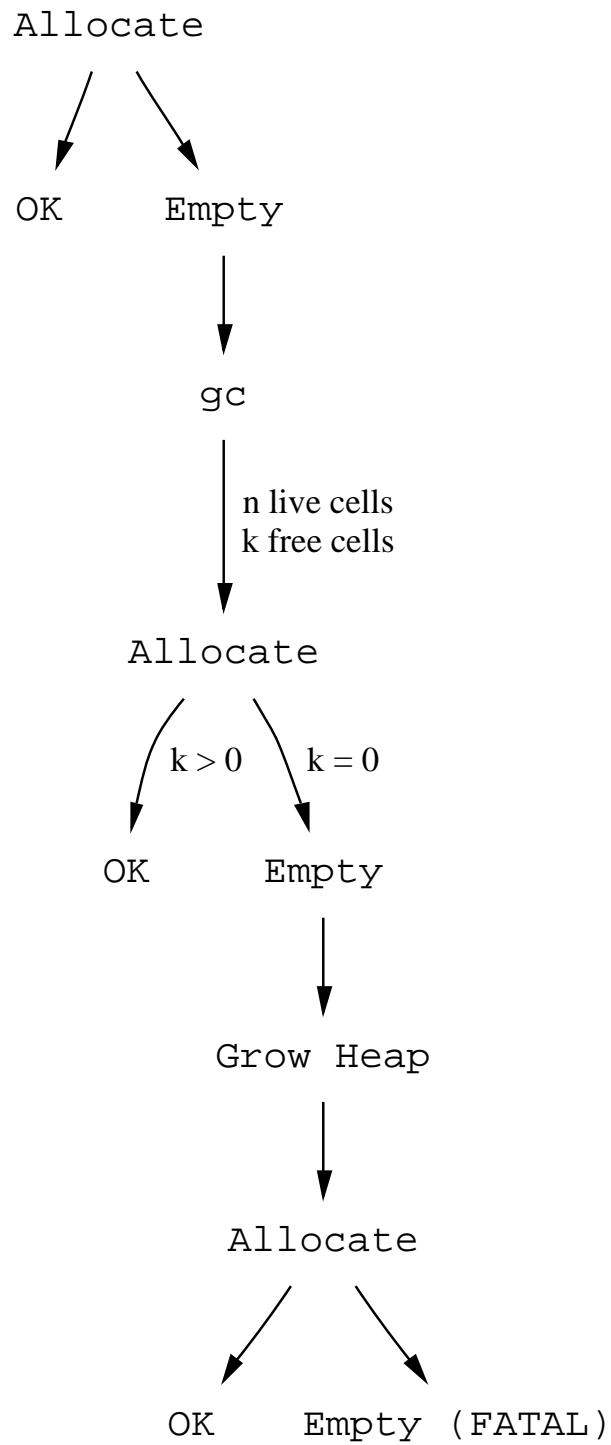
if reference count becomes zero, put on free
list

code must adjust ref count **at every**
assignment!

also, can't reclaim cycles

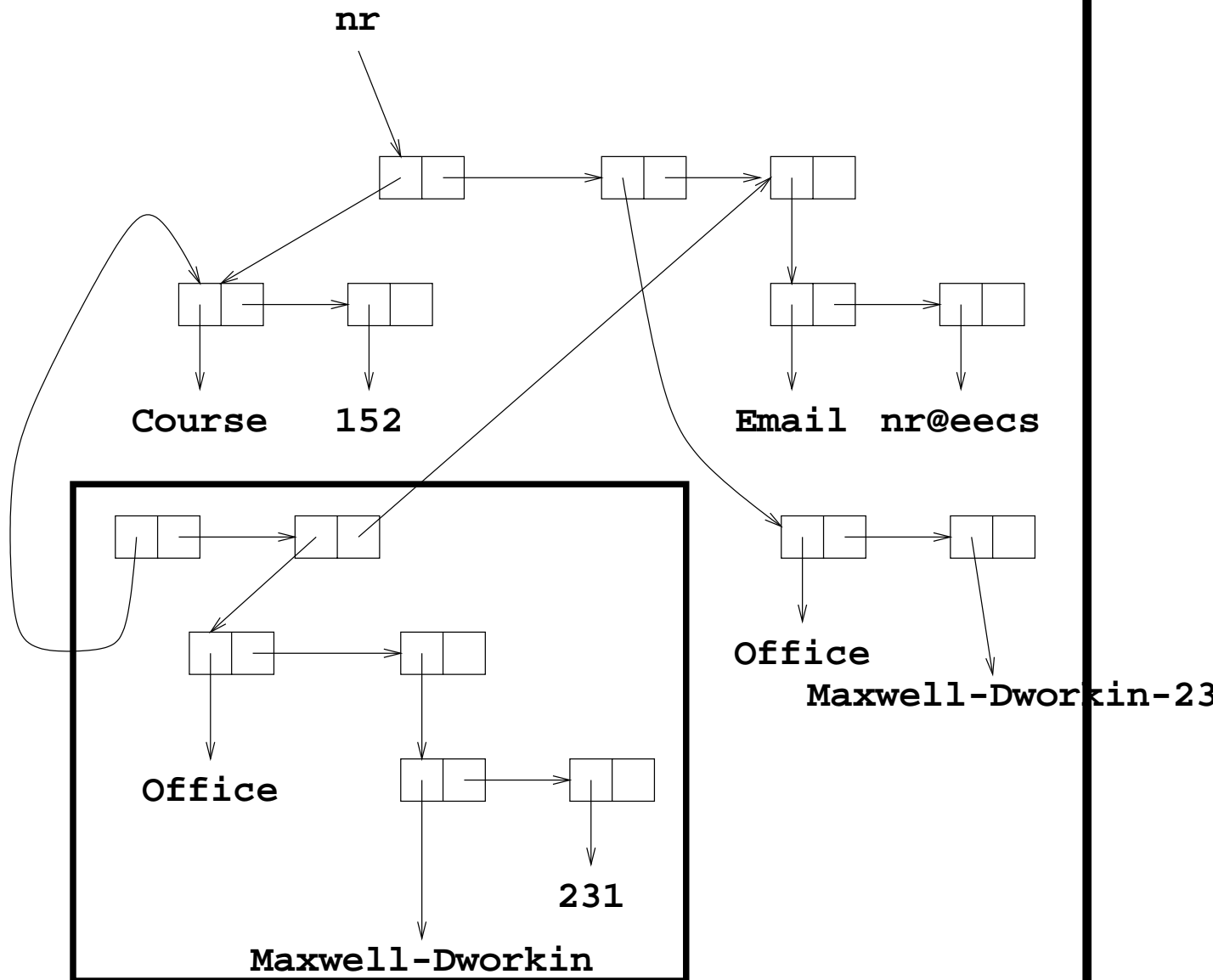
BUT! No pause times

Garbage collection and heap growth



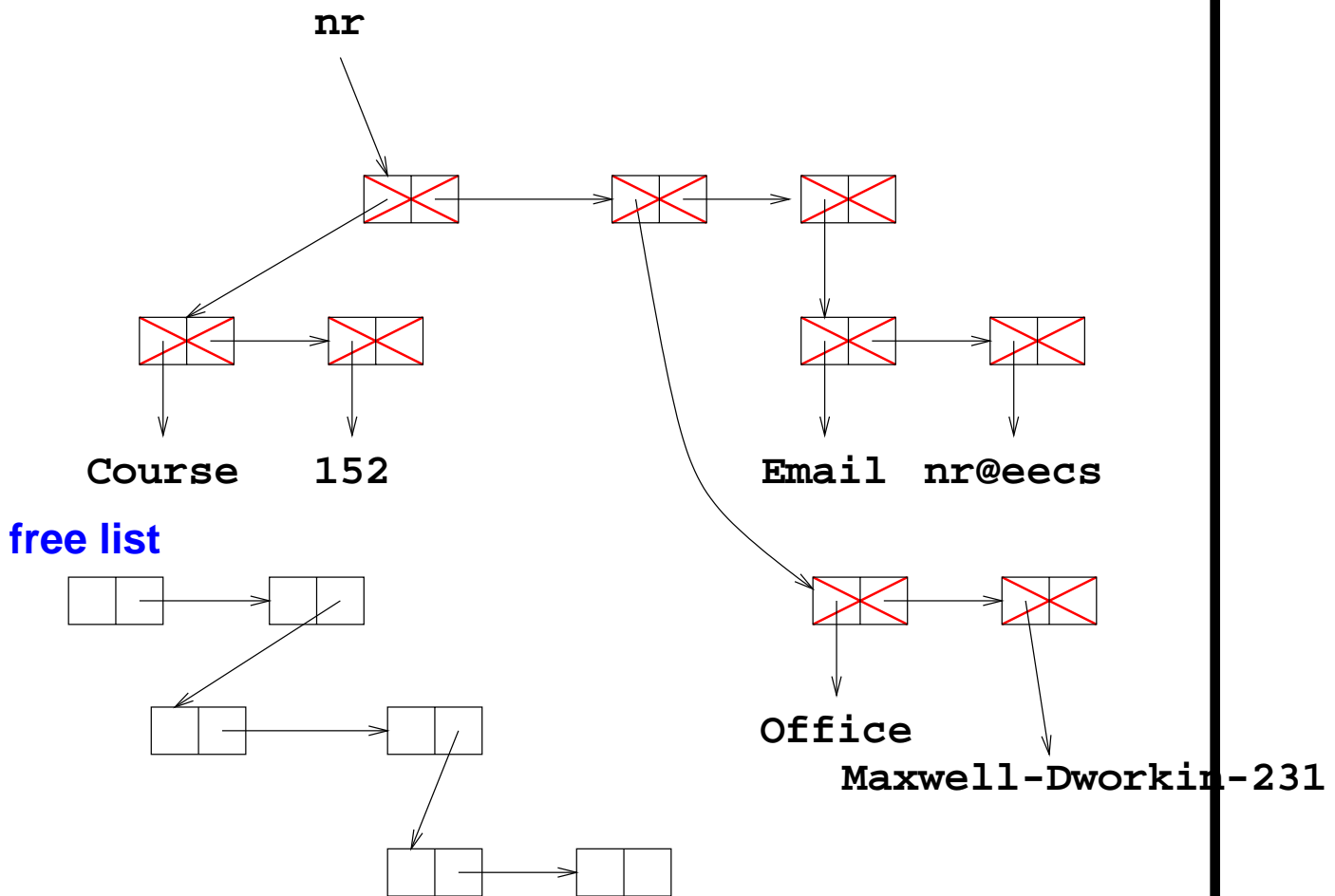
Mark and Sweep Collection

Return to association list:



Sweep

Sweep phase looks at every cell in the heap
puts unmarked cells on free list:



Store marks (X) in objects or in separate bitmap

Tracking pointers in the mark phase

While marking, must track marked nodes whose children have not yet been marked
will go back and mark those children later
essentially a depth-first traversal
depth-first walk requires a stack
but we're out of memory!

Early collectors allotted a special stack
about 5% of memory size
dump core if special stack exhausted

Along came:

Peter Deutsch

Herbert Schorr and William M. Waite

Marking algorithm uses only three extra pointers,
plus one bit per node!

Idea:

if you have pointer to “current node,” you don't
need copy in a predecessor node

So—temporarily save stack in `car` and `cdr`
fields!!!

For details, see Knuth, vol 1, §23.5

This problem vanishes in a copying collector

Shifting work from collection to allocation

Work done at collection:

unmark phase proportional to **total heap size**

mark phase proportional to **live data**

sweep phase proportional to **total heap size**

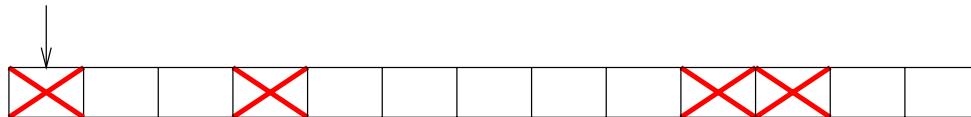
large heap \implies **long pause times**

Work done at allocation:

constant work (take one cell off free list)

Clever ideas:

- **unmark & sweep in allocator**—cuts pause time
- **use free-space pointer**



Allocate: step free pointer to next unmarked cell
Free list is eliminated! (saves pointer fiddling)

During one GC cycle:

collector work is **proportional to live data**

total allocator work is

constant per allocation, PLUS

amount of live data at last collection

Most cells garbage:

expected work per allocation is small

Mark and sweep with objects of varying size

What if not everything in life is a cons cell?

Can't have a single free list to satisfy every request.

May have

- **multiple free lists of different sizes**
- **free memory broken into arbitrary blocks**
(first fit, best fit)
- **“buddy system” allocator**

Just like traditional dynamic memory allocators

And we have the same problem: **fragmentation hurts locality of reference**

**When we ask for an object,
we may have plenty of memory but...
it may be in tiny unusable pieces between
allocated bits**

Beating fragmentation

Good strategy called BIBOP: Big Bag of Pages

divide memory into fixed-size pages

**each page contains objects of a single type
and size**

**objects larger than 1 page typically get special
treatment**

Advantages:

can identify an object's type just by its address

no tag bits or descriptor word needed

(cons cells—33% off!)

can put mark bits in a separate bitmap

(less overhead, better locality)

can bound losses to fragmentation

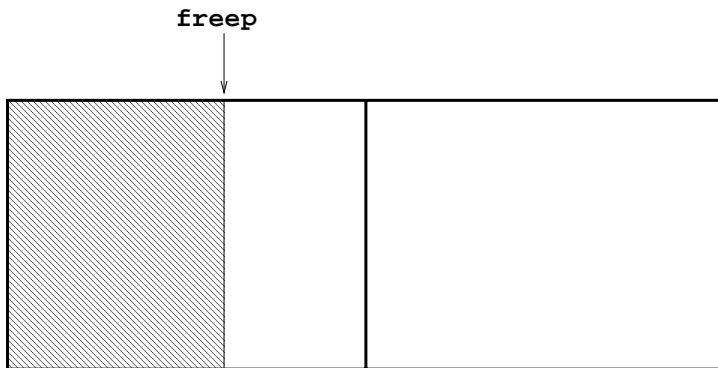
Stop and copy — trading space for time

Divide memory into two contiguous *semispaces*

One semispace unused except for collection

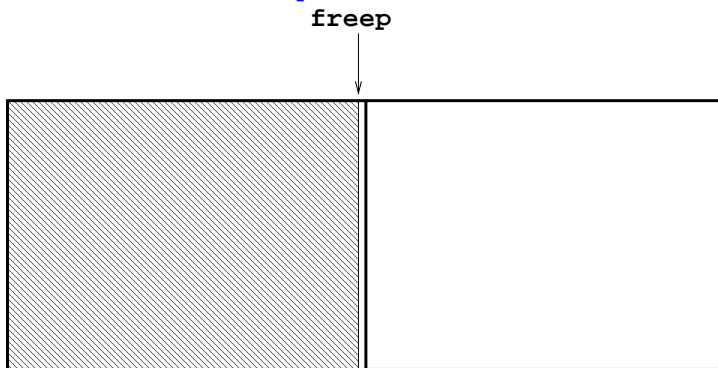
Allocate from the other

Allocation by incrementing a pointer:



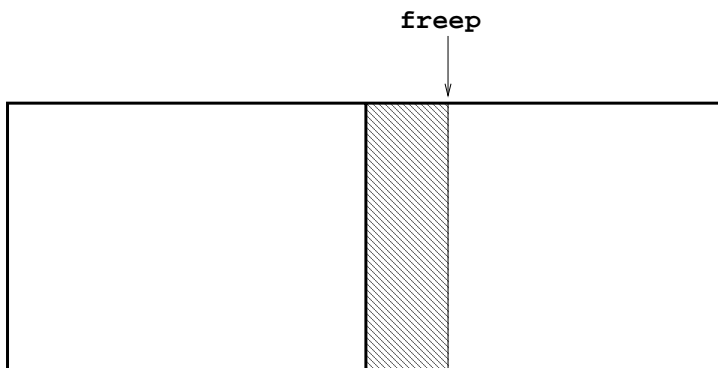
Stop and copy – collection step

When semispace is exhausted:



Only some of the shaded area is live data
the rest is garbage

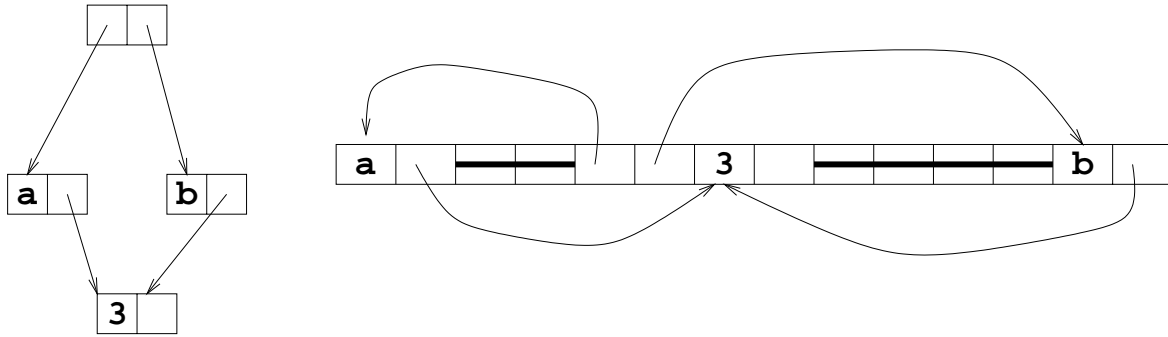
If we copy only the live data to the other
semi-space,
we'll have plenty of room to allocate more:



Copying goes from *from-space* to *to-space*
Change of roles (from one semi-space to the other)
is a *flip*

Stop and copy details

Collector must preserve sharing, cycles:



Must copy each live object exactly once.

What if there is more than one path to an object?

At first visit, replace with *forwarding pointer*
don't need extra space; overwrite object

Marks object as already copied

Shows location in to-space

All references to object must be updated to new location

To mark "forwarded," could use 1 bit/object

but can tell just by the value of the pointer!

only forwarding pointers point into to-space

Must distinguish pointers (updated) from
non-pointers (not updated)

Stop and copy – graph traversal

As we walk the graph of live objects,
we must track objects that have been copied,
but the things they point to haven't been
copied.

(Compare mark-and-sweep, we had to track objects
that were marked, but things they pointed to hadn't
been marked.)

For mark-and-sweep, we used a stack: depth-first
traversal

For copying, we use a queue: breadth-first
traversal

Use to-space as the queue, at no cost!

Need only two pointers for copying collector:
`freep` points to next free location in to-space
`scanp` points to next object whose referents
not copied

Invariant:

Objects before `scanp` point to to-space
Objects between `scanp` and `freep` point to
from-space

Stop and copy algorithm

Basic operation is to forward a pointer:

```
forward (p) =  
  if *p is forwarding pointer  
  then return *p  
  else  
    copy the object *p to location freep  
    *p := freep  
    freep := freep + length(*p)  
  return *p
```

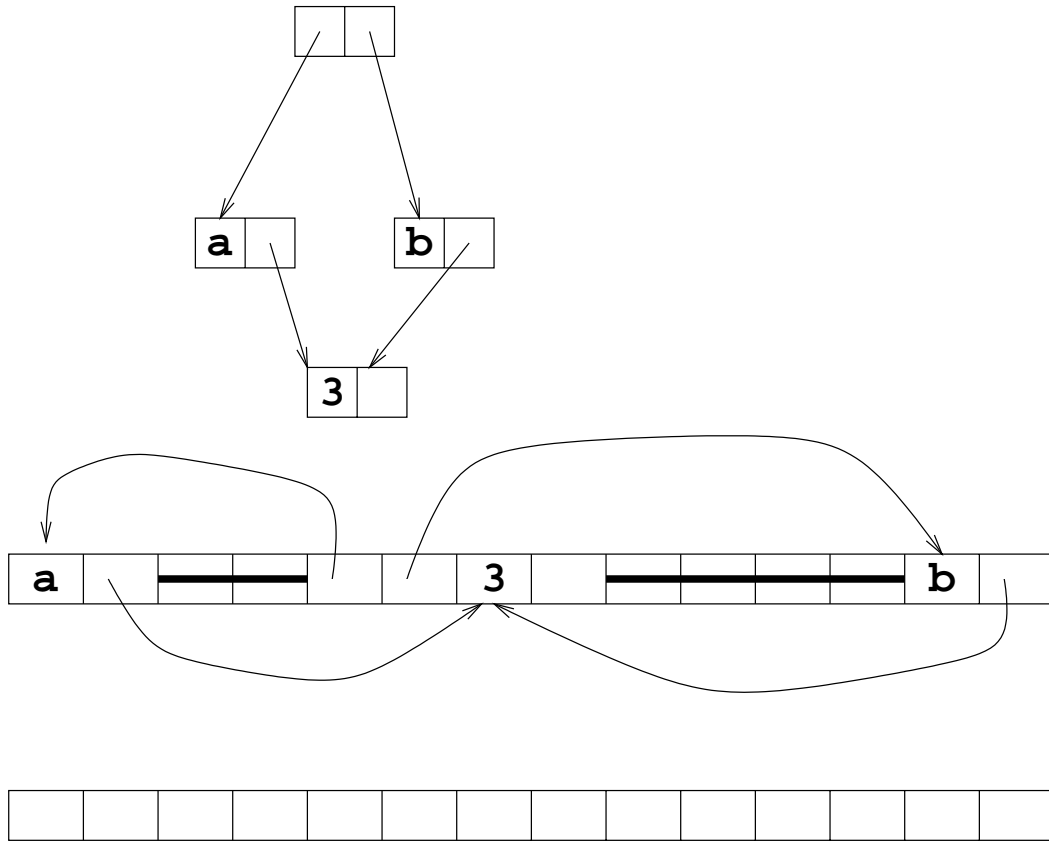
Copies a single object (or its forwarding pointer.)

To copy everything, start with roots,

then consume queue from scanp to freep:

```
scanp := freep := beginning of to-space  
for each root r do  
  r := forward(r)  
while scanp < freep do  
  let L be length of *scanp  
  for i := 0 to L-1 do  
    if scanp[i] is a pointer then  
      scanp[i] := forward(scanp[i])  
scanp := scanp + L
```

Stop and copy example



Properties of stop and copy

**Efficiency very good in limit of large memory
(but so is mark and sweep)**

**Crucial advantage comes from lightning-fast
allocator:**

**allocate by check and increment
(exactly the same cost as stack allocation)
(garbage collection can be cheaper than stack
allocation)**

Can reduce cost of check two ways:

**VM hardware (cute, but hard to port)
check bounds register
check only at loops!
one check for many allocations**

Copying step compacts all data

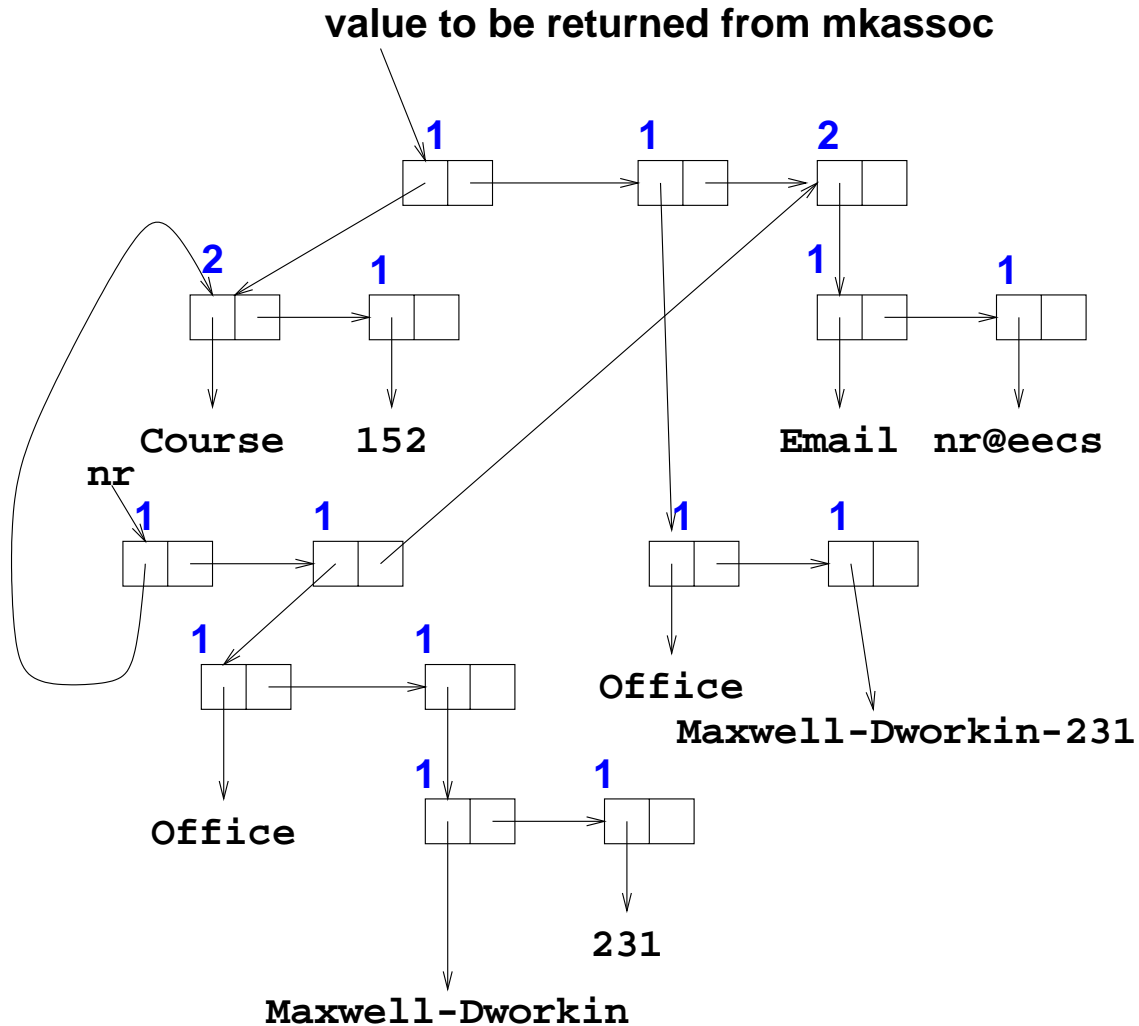
**there is never any fragmentation of free
memory!
(can do compacting mark-and-sweep, too)**

Reference counting

Widely used technique in early days:

each object tracks number of references to it
when count goes to zero, can put object on
free list

Example:



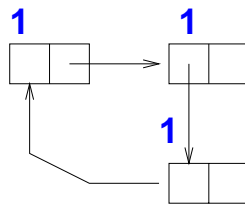
Reference counting costs

Must change ref counts at every assignment!
(the killer – CPU overhead can reach 20-30%)
(various tricks used to cut down costs here)

Expensive to follow pointers when costs go to 0
can defer to allocation, like mark-sweep

So, can get good bounds on costs
good for real-time properties

But **ref counting can't collect cyclic garbage:**



Serious implementations need extra collector
there go the real-time properties

Ref-counting can be tricky to get right
discovered by many implementors of C++ classes

Old ref-counting collectors gave GC a bad name

Generational collection

Two observations about functional programs:

- newer cells tend to point to older cells
(always true except where there is mutation)
- young cells tend to be short-lived, old cells long-lived
(if cell kept for even 1 collection, it's probably important: will live for a while)
(new cells often throwaway intermediate results—e.g, in simple version of `rev`)

Divide heap into *generations*

collect younger generations more frequently

When collecting younger generations,

roots include pointers from old objects to young ones

such pointers can be created only by assignment!

(think about cons)

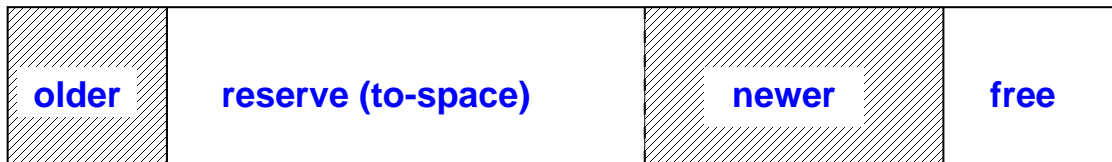
common solution: track all assignments

Works well in LISP systems

even better in ML systems!

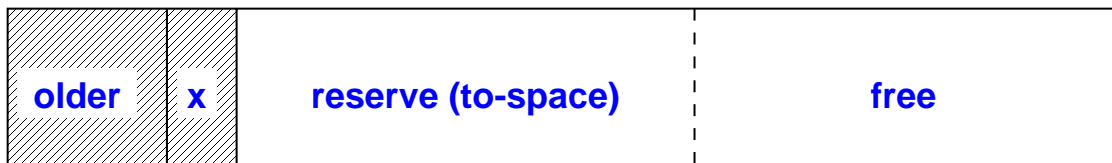
Generational example: 2 generations

So, for example, 2 generations:



When free space exhausted, copy newer to reserve
(so-called **minor collection**)

older plus survivors (x) are new older generation
survivors are *promoted*



When older generation approaches half of memory
copy-collect older generation (**major collection**)
copy result to beginning of heap, start over...

Massive savings

try to estimate (fine exam question...)

Copying generations easiest to understand

but, surprisingly, can do with mark and sweep

Many generations can save more

keep youngest generation entirely in cache!

Conservative collection – copying

**What if you don't know where roots are,
but you do know object layouts?**

**“Mostly-copying” conservative collection
(**Bartlett** Scheme compiler)**

**Anything on stack or in globals could be a pointer...
so treat it as a pointer!**

**Entire stack, data, bss segments are pointers
Can't forward (change) pointers on the stack,
so objects are “pinned” in memory**

**But, if you know your own objects,
you can forward anything pointed to only by
your own objects**

BIBOP lets you identify object addresses

**Conservative guess at roots may mistakenly keep
garbage**

- **surprisingly little in practice**
- **get many advantages of regular copying collector, including little fragmentation**

Conservative collection – mark & sweep

What if you don't know object layouts?

You can still garbage-collect!

but you can't move anything,
because you can't tell pointers from integers
anywhere

Mark-and-sweep to the rescue (**Boehm/Weiser**)!

Super-conservative assumptions:

Anything on stack or in globals could be a
pointer ...

and anything on the heap could be a pointer

Treat them all as pointers

Works astonishingly well in practice

simple replacement for `malloc/free` yields

approximately the same CPU cost

30–150% memory overhead

a few tricks can make it even better

plus can write faster code by relying on GC

State of the art collector (Boehm et al.) widely used

Java, Cedar, libscheme, ...

dovetails nicely with C, C++ code

Dueling memory-management schemes

Simple versions have different pros and cons

Mark-and-sweep

- **fragments**
- **hard to do generational collection**
- **poor locality**
- **easy to make conservative**

Copying

- **needs big memories**
- **copying large objects is expensive**
- **identifying & forwarding pointers can be hard**
- **easy to make generational**

State-of-the-art collectors resemble one another ever more closely...

Asymptotic behaviors are all the same.

Important practical aspects of performance are:

constant factors

locality of reference

Plus Ultra

(There's more)

Memory management a hot area in research

incremental collection

concurrent collection

real-time collection

collection of persistent store

many strategies for better performance

Things to remember

Garbage collection makes safety possible
huge classes of common bugs are eliminated.
(estimated 40% of bugs in Xerox Mesa)
(Purify works by conservative-collection
techniques)

All schemes are based on
root set
following pointers to live data
distinguishing pointers from non-pointers
(always possible to some degree)

Performance even of simple schemes can be good

Sophisticated collectors outperform `malloc/free`
interfaces simpler
no copying to control ownership

Conservative collectors can support any language
even C or C++

A highly effective use of programmer time
use it in your next program