Memory allocation, garbage collection

Lecture 17

Types of memory management

Static: fixed when a program is loaded

Dynamic

- Stack (sometimes more than one)
- Heap
  - Explicitly managed : must call free
    - malloc / free
    - --many variations
  - Implicitly managed: done behind the scenes
    - Reference Counting
    - Garbage Collection (GC)
    - --Mark&Sweep, Copying, Conservative..

Why Dynamic?

Static means you have to know at compile time how big your program's data can be. FORTRAN (pre 1990) did this.

You allocate the largest array you think you will need and hope for the best. *Advantage: You will never be tossed out by OS for using too much memory in the middle of computing.*

Stack allocation means you can grow, but only last in, first out. You must return storage in the reverse order of its allocation. Sometimes works just fine for a language design with nested scope. Sometimes you run out of stack space.☹ This could never happen in fortran 66☺

Why Implicit Heap Management?

Explicit management is extremely error prone. (malloc, free) It is a source of subtle bugs in every system that uses it (including almost any large system written in C!)

Lisp/Java argument: don't even ask the programmer to manage memory.

Heap allocation solve lots of technical problems: in a programming language (Java), having EVERYTHING be in the heap and *all* values pointers to objects in the heap makes semantics neater.

Another solution to Heap Management?

.

Some systems solve the problem by never deallocating memory, assuming that you will eventually kill the whole process.

How long does your browser stay up?

Sometimes you have to reboot the whole operating system.

In Lisp, is all data in the heap? Often not

Conses (i.e. lists, dotted pairs), yes
Arrays, yes usually
Arbitrary precision integers, yes

Numbers: maybe.

--Here's a trick. If a pointer is a negative number (leftmost bit is 1) maybe it can't really be a pointer at all. So make it an "immediate" number. You can do arithmetic etc. with this "FIXNUM". Instead of "following a pointer" to a number, you fake it.

(Lisp also uses a stack for binding values and most implementations use static space for binary programs e.g. loaded from fasl files or written in asm, C,…

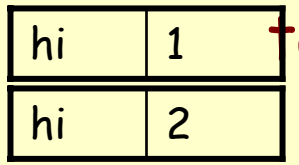
Reference Counts: an easy method

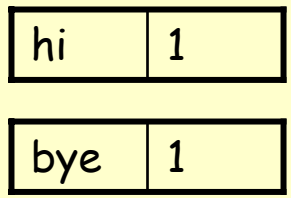
Every Heap cell has a count field , full-address size.

B= new cell init. to "hi" take cell from freelist

A:=B increment

A:="bye" decrease hi's count, increase bye's count.





When count decrements to 0, there are no users of that cell: put it on list of free storage.

Why use Reference Counts

If the cost of maintaining counts is small compared to the other operations.

If it is important that the cost is assessed immediately and is predictable (no clumpiness like GC). (though this has mostly gone away with fast memory, generational GC)

Why not use Reference Counts

Fear of not being able to collect "cycles" is often cited as a problem.

When all is said and done, not as fast as GC, and uses lots of memory for the count fields. In fact you can have a lisp-like system with reference counts but a cons cell would grow from 64 to 96 bits (with 32 bit addresses) .

Why does a ref. count field have to be so large? Can we use only a few bits?

Who uses Reference Counts

File systems. How many references or links are there to a file? If none, you can delete it. The cost of maintaining counts is small compared to the overhead of file open/close.

Some computer systems with largish data objects (e.g. something like Matlab, or Mathematica.)

Some defunct experimental lisp or lisp-like systems: esp. if GC/paging is slow, RefCounts seems more plausible (REFCO, BBN Lisp used a limited-width counter, 1,2, many).

Why Garbage Collection (GC)?

GC is a winner if memory is cheap and easily available.
This combination is a relatively new phenomenon.
GC continues to be a popular academic topic that can cross boundaries of software, architecture, OS. Parallelism, distributed GC.
Revived interest with Java, too.
Conservative GC can be used even with systems for which GC would not seem to be plausible.

Why not GC?

If you have so much memory, why not put it to use instead of keeping it in reserve for GC?

Some GC algorithms stop the computation at odd moments and keep the CPU and perhaps paging system very busy for a while ("not real-time").

Speed: Explicit allocation can be faster, assuming you know what you are doing. (*Can you prove your program has no memory leak? Sometimes.*) Stack allocation is safe, too.

(depending on implementation) A "real" implementation is complex: when to grow the free space, how to avoid moving objects pointed to from registers, etc. Bad implementations are common. See Allegro CL implementation notes on GC parameters.

Kinds of GC

Mark and Sweep
Copying
Generational
Incremental, concurrent
Conservative (not in Appel)

Mark-and-Sweep. The simplest.

When you ask for a new record and the free list is empty, you start a GC:

Mark: Start with roots: static names, stack variables. March through all reachable nodes, marking them. [how do you mark a node? In the node? In another block of storage, 1 bit per node?]. If you reach an already marked node, great. Turn back and do other stuff. {You might use a lot of stack doing this. Problem??}

Sweep: Go through all the possible nodes in order, in memory. For each one that is NOT marked, put it on the free list. For each one that IS marked, clear the mark.

Where are the roots?

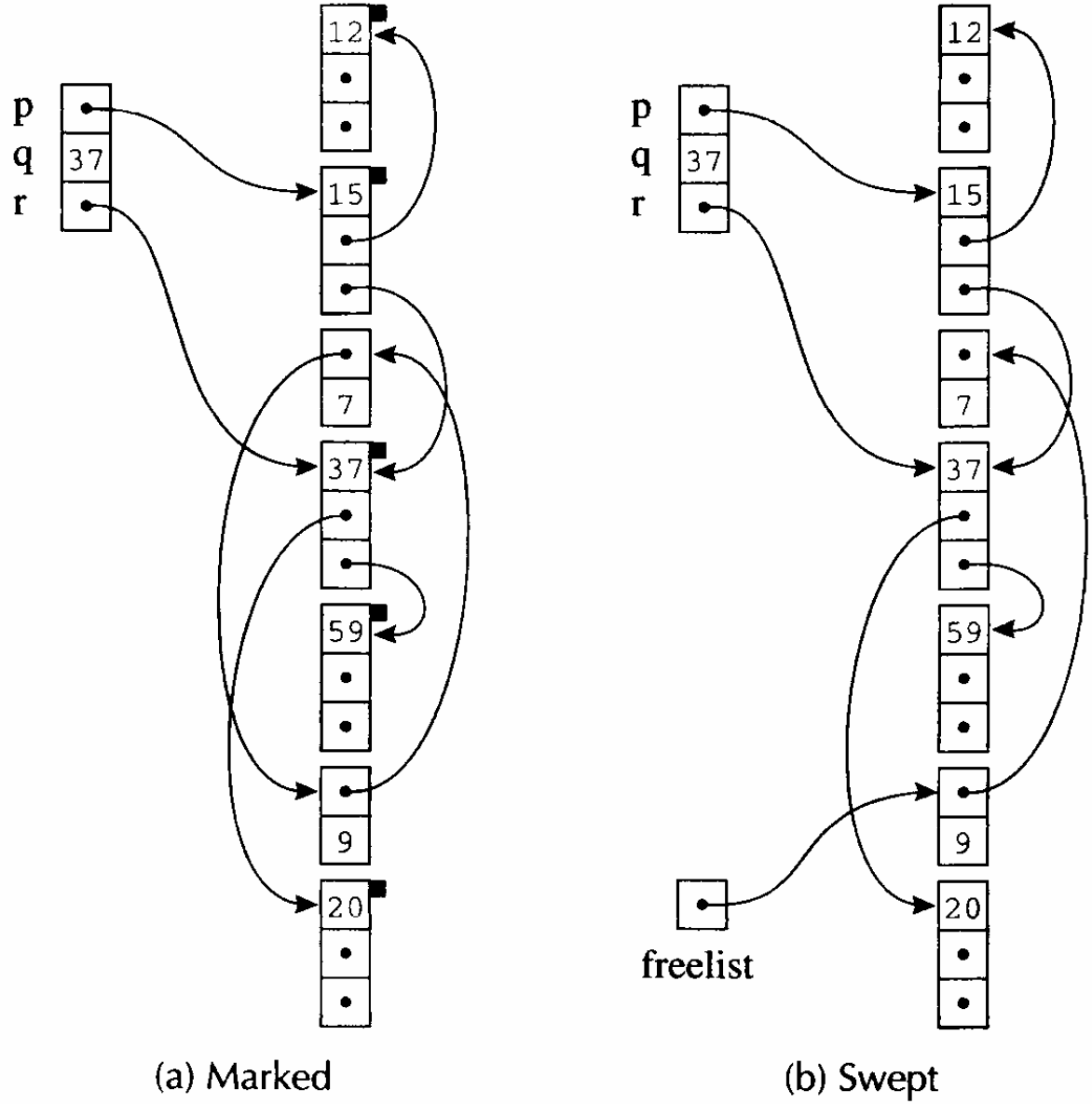


FIGURE 13.4. Mark-and-sweep collection.

Cost of Mark-and-Sweep

Mark: suppose R cells of data are reachable. Marking is linear in R so the cost is $c_1 \times R$

Sweep: suppose H cells are in the heap. Sweeping is linear in H so the cost is $c_2 \times H$

Number of cells freed is H-R. We hope this is large, but it might be small as you run out of memory…

Amortized cost (= cost per cell freed) is

$(c_1 R + c_2 H)/(H-R)$

If the cost is too high, algorithm should get more H from Operating System!

Other considerations for Mark/Sweep: stack space

Mark: This is done by a depth first search of the reachable data, and just using calls could require stack space linear in R. It is possible to simulate recursive calls more economically but still linearly. (p 280) or by hacking pointers backward as you mark, and then reversing them later, you can use no storage. Timing tests with pointer reversal suggest it is not a good idea.

Improved Sweeping

Sweep: If you have several sizes of records, finding a record of suitable size on the freelist may be some work. Keep a separate freelist on a per-size basis? If you run out of size X try size 2X, split it into two size X pieces.

Copying GC

Divide Heap into two regions, OLD and NEW, delimited by high/low limit markers.

Allocate space from OLD Heap.

When you run out, start from roots and copy all live data from OLD to NEW.
Switch OLD/NEW.

Copying is not so obvious: when you copy a cell, look at all its pointers. If they point to NEW space, fine. If they point to OLD space, those items must also be copied.

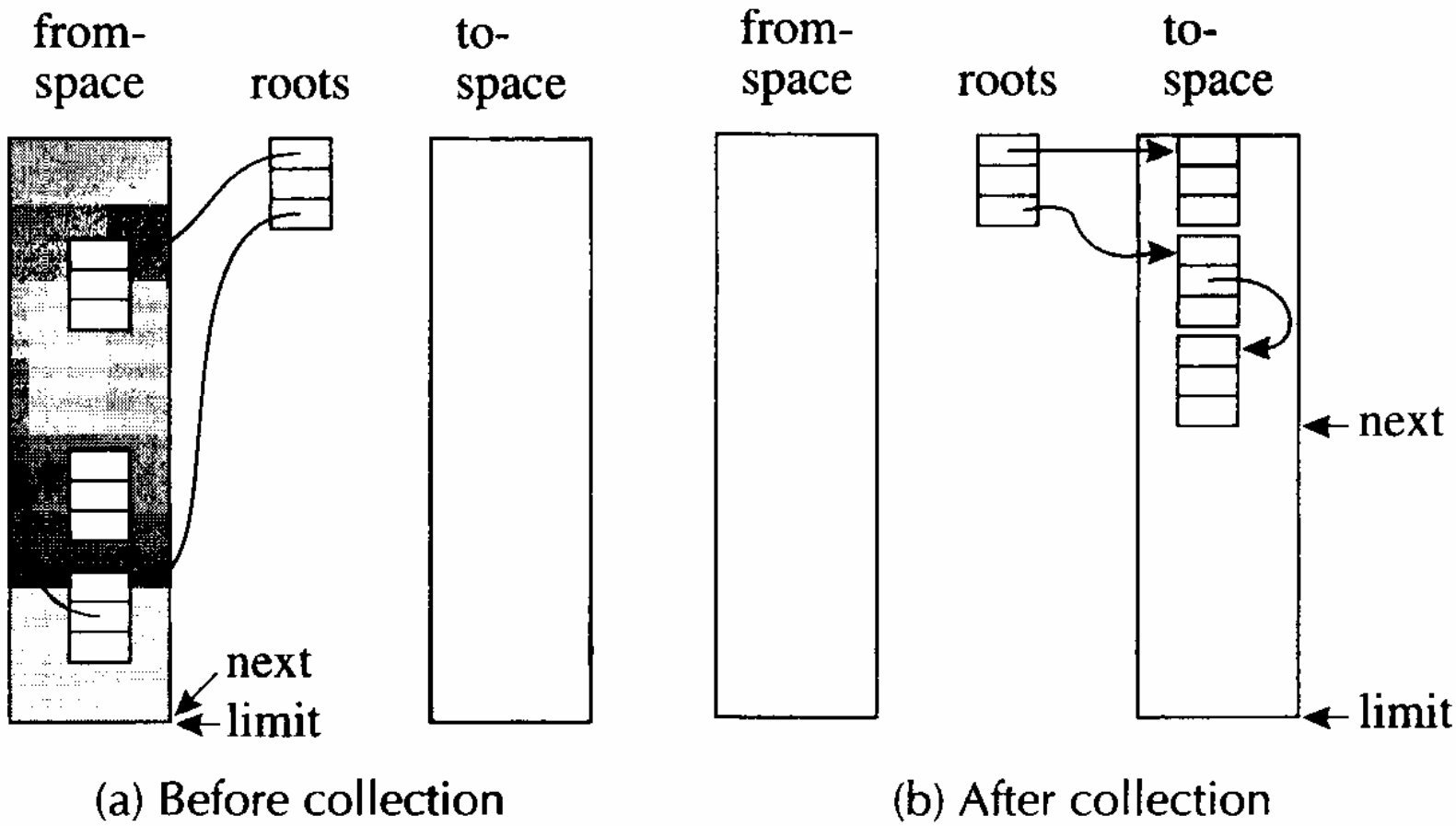


FIGURE 13.7. Copying collection.

Pro: Copying GC

Storage in use is compacted. Good for memory cache. If there is a pointer from object A to object B, there is a good chance that A and B will be adjacent in memory.

Newly constructed lists are going to be in same cache line, since the freelist is also contigouous.

Unused storage locations are not ever examined, saving cache misses.

Con: Copying GC

Half the storage is not even used. That means that GC is twice as frequent.

Items are being moved even if they don't change: if they are large, this is costly.

All references to storage must be indirect/ locations can change at any time.

Generational GC

Based on the observation that in many systems (certainly in long-running Lisp programs) many cons cell have a very short life-span. Only a few last for a long time.

Idea: Divide up heap cells into generations. GC the short-lived generation frequently. Promote cells that live through a GC to an older generation. This promotion is done by copying into a more permanent space.

Rarely do a "complete" GC.

Pro: Generational GC

Usual GC is extremely fast (small fraction of second)
A good implementation reduces typical time in GC from 30% to much less… 5%?

Con: Generational GC

The (rare) full GC can be expensive.

Elaborate programming and instrumentation:

Extra bookkeeping to maintain pointers from old generations to new; this can add to the in-line instruction generation. *When something in an old generation changes, GC must use it to trace new data (new root info).*

Similar to copying, but with more than 2 spaces; data can move at any time a GC is possible.

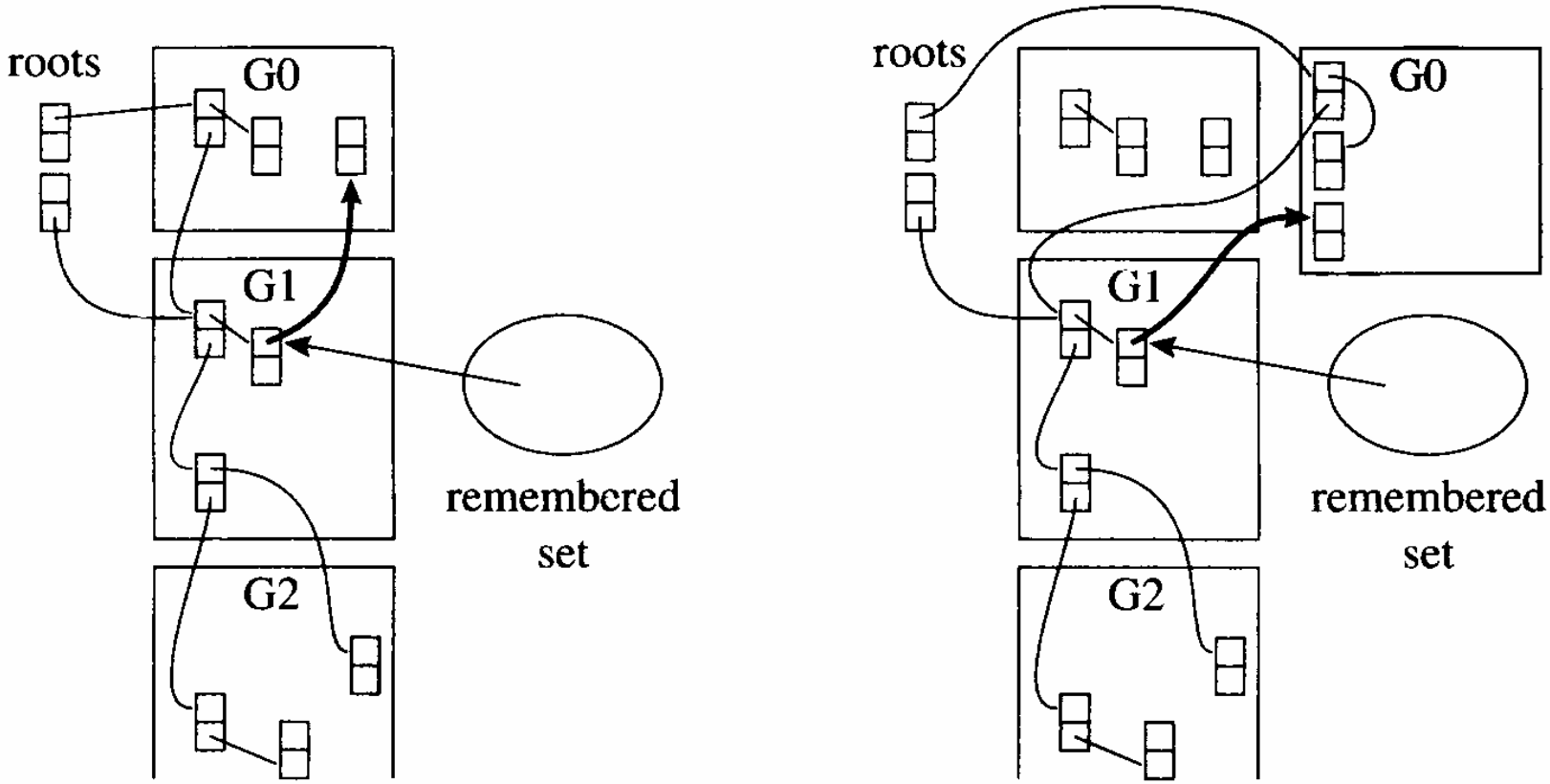


FIGURE 13.12. Generational collection. The bold arrow is one of the rare pointers from an older generation to a newer one.

Conservative GC

Imagine doing a mark and sweep GC, but not knowing for sure if a cell has a pointer in it or some other data.

If it looks like a pointer (that is, is a **valid word-aligned address within heap memory bounds**), assume that it IS a pointer, and trace that **and other pointers in that record** too.

Any heap data that is not marked in this way is garbage and can be collected. (There are no pointers to it.)

Pro: Conservative GC

It can be imposed upon systems "externally" and after the fact.

Doesn't need extra mark bits (presumably finds some other place for them)

Con: Conservative GC

Assumes we know what a pointer looks like: it is not munged up or encoded in an odd way, it doesn't point to the middle of a structure, or if so, we make special efforts to keep pointers live.

Not so fast or efficient or clever as generational GC.

Sometimes marks nonsense when a data item looks like an address but is not.

(Note: real lisp systems tend not to just use full-word pointers = addresses. This wastes too many bits! E.g. fixnum encoding etc.)

Current technology

Almost all serious Lisp systems use generational GC.
Java implementations apparently vary (e.g in C might use generational GC on top of C).

For any long-term continuously-running system, a correct and efficient memory allocation system is extremely important. Rebooting an application (or even a whole operating system) periodically to kill off bloated memory is very inconvenient for "24/7" available systems.
I have to kill my Netscape browser every few days…
(ESS5 anecdote)
Further reading: Paul Wilson Survey of Garbage Collection