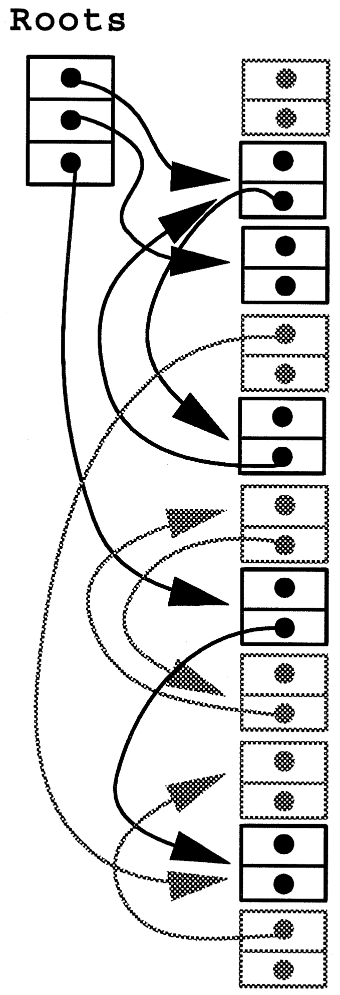
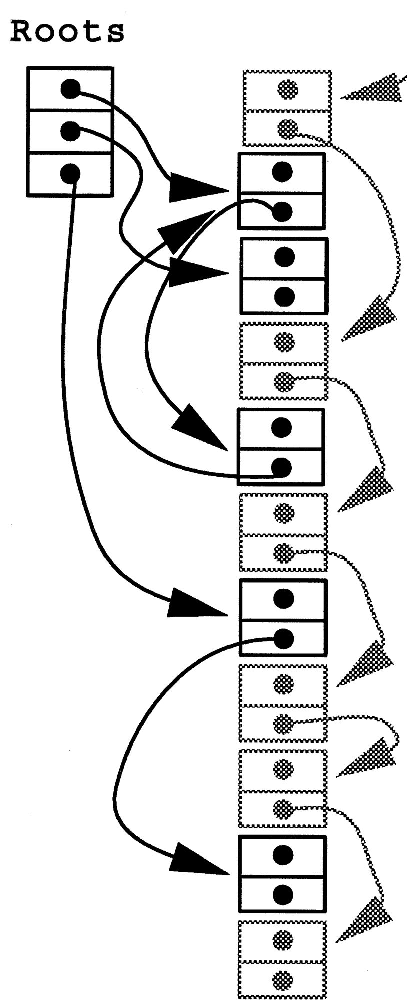
Mark and Sweep

- Depth-first search to mark live nodes
- Sweep through all memory,
  putting unmarked nodes on free-list
- Sweep also unmarks the marked nodes

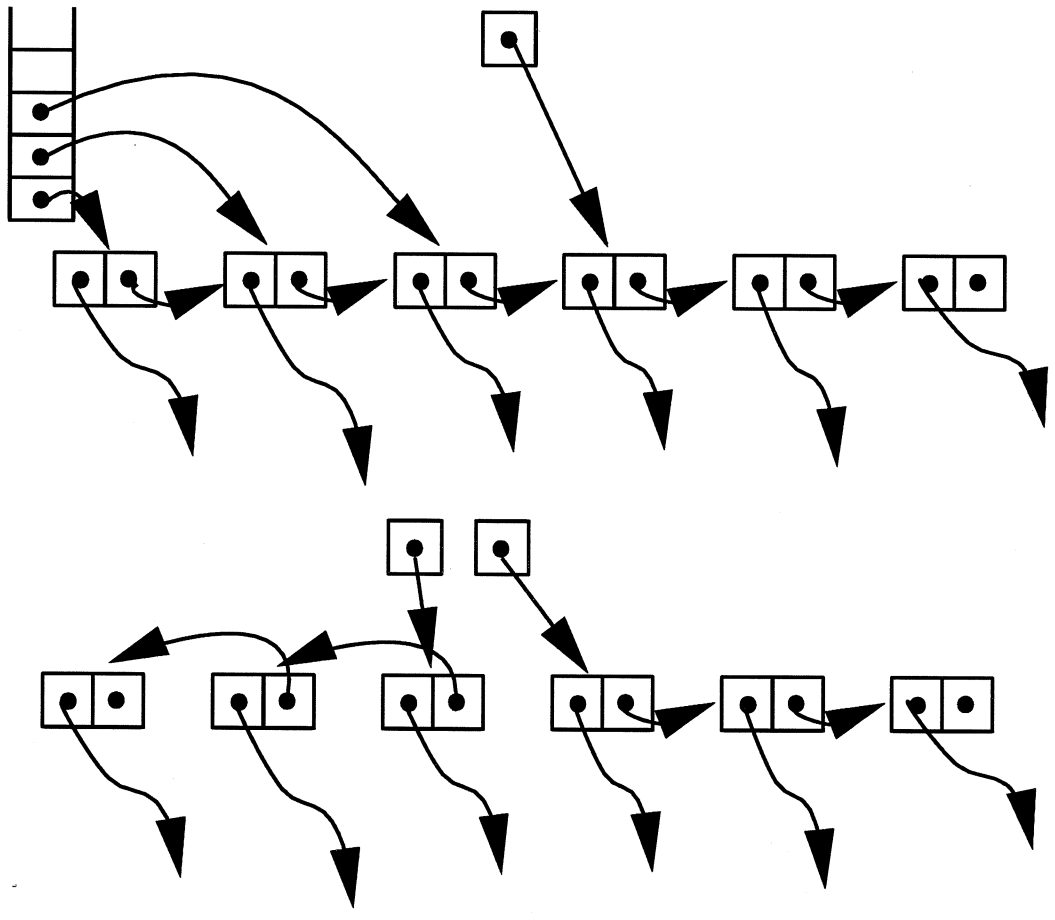




Pointer Reversal

Schorr & Waite 1965, Deutsch 1965
(see Knuth vol. 1)

- Problem: Depth-first search needs a stack
- Stack depth could be as big as the graph
- Solution: Chain the stack inside the graph

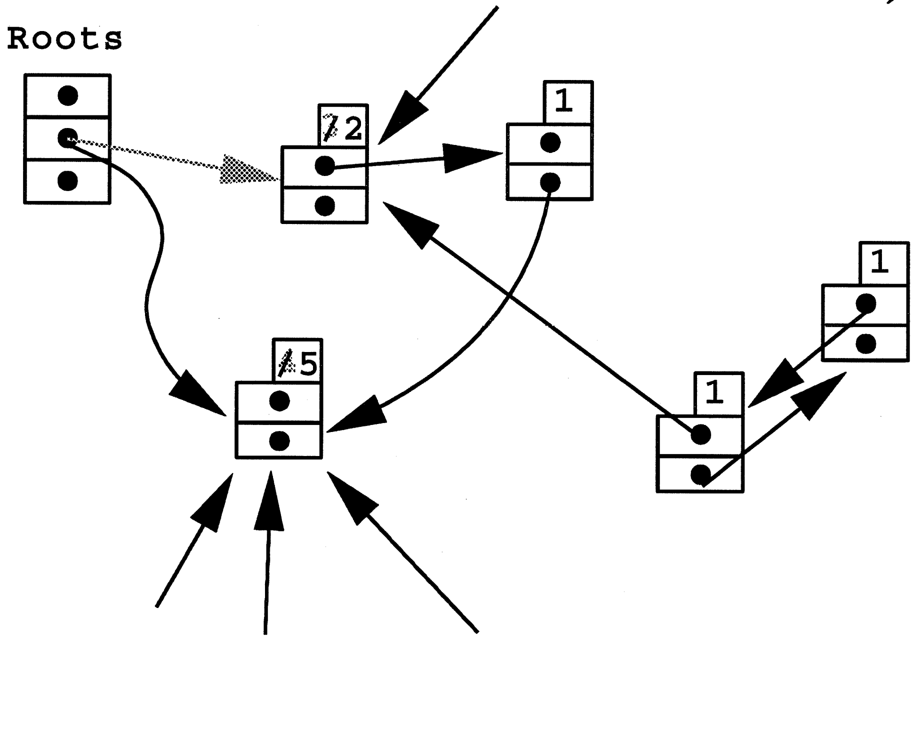


Reference Counts

Keep a count of the number of pointers to each object.

On every store operation, adjust counts.

When a reference count hits 0, put object's storage back on free list (after decrementing reference counts of what it pointed to).



Pros & Cons of Reference Counts

Disadvantages:

- Large cost to adjust reference counts (ameliorate by dataflow analysis)
- Can't reclaim circular structures
- Long latency to reclaim large structures (ameliorate by "lazy decrement").

Advantages:

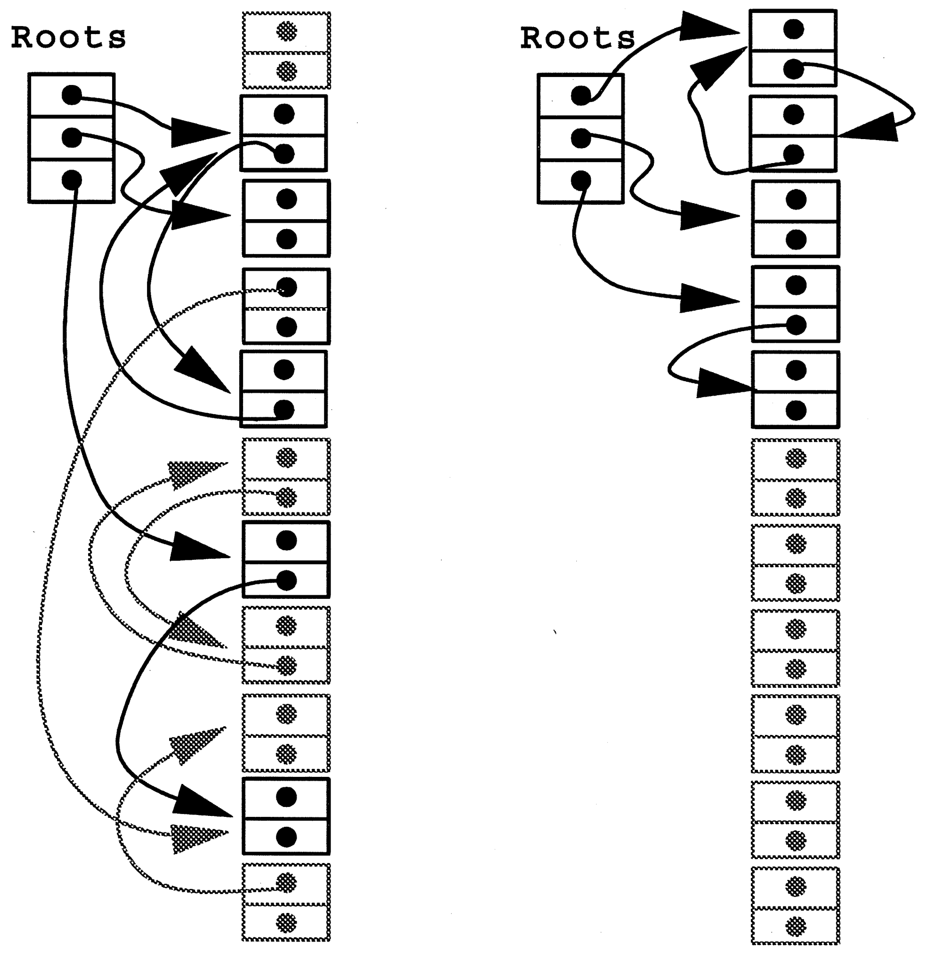
- Effective for non-cyclic structures if memory is very tight.

Hint: DO NOT USE

Stop and Copy

Fenichel & Yochelson 1969

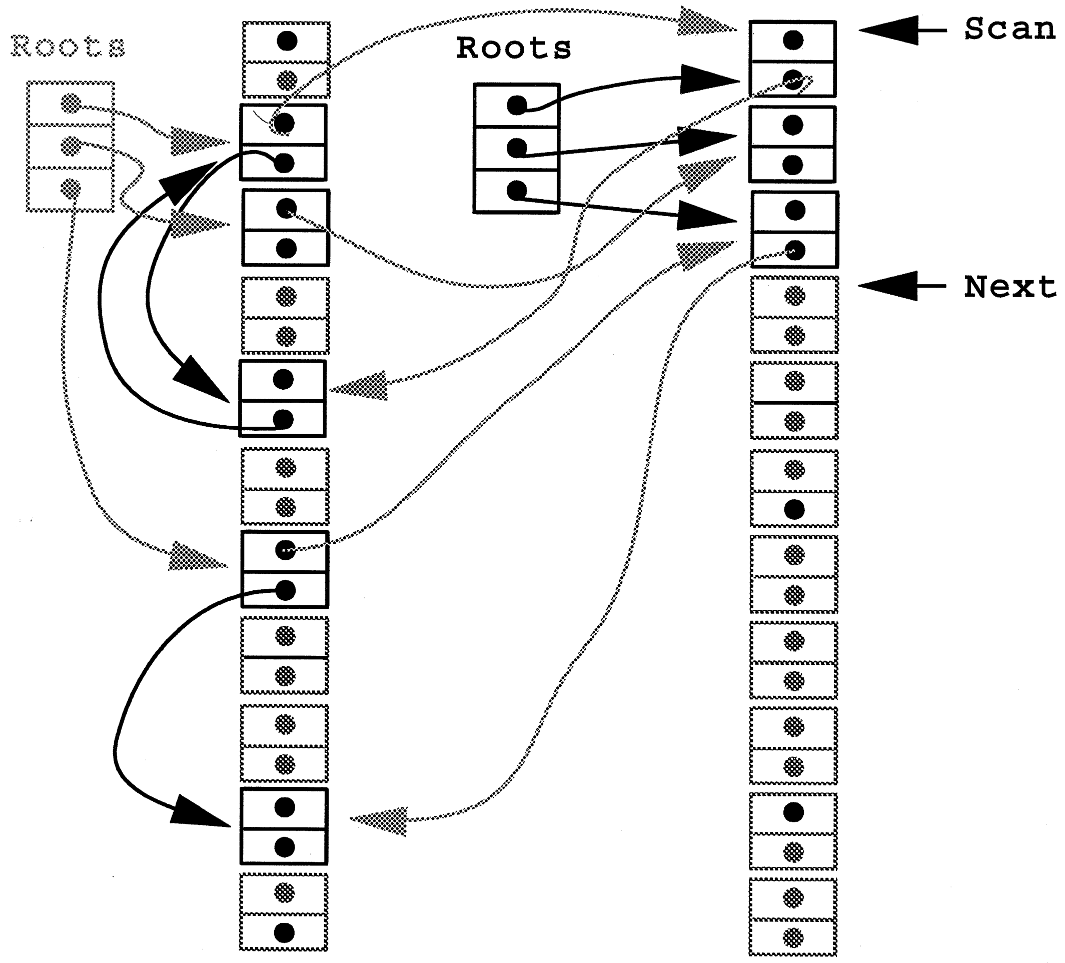
- Divide memory into two "semispaces"
- Allocate in space A until it's full
- Copy the live data to space B (DFS)
- Switch roles of A and B



Breadth-first Stop and Copy

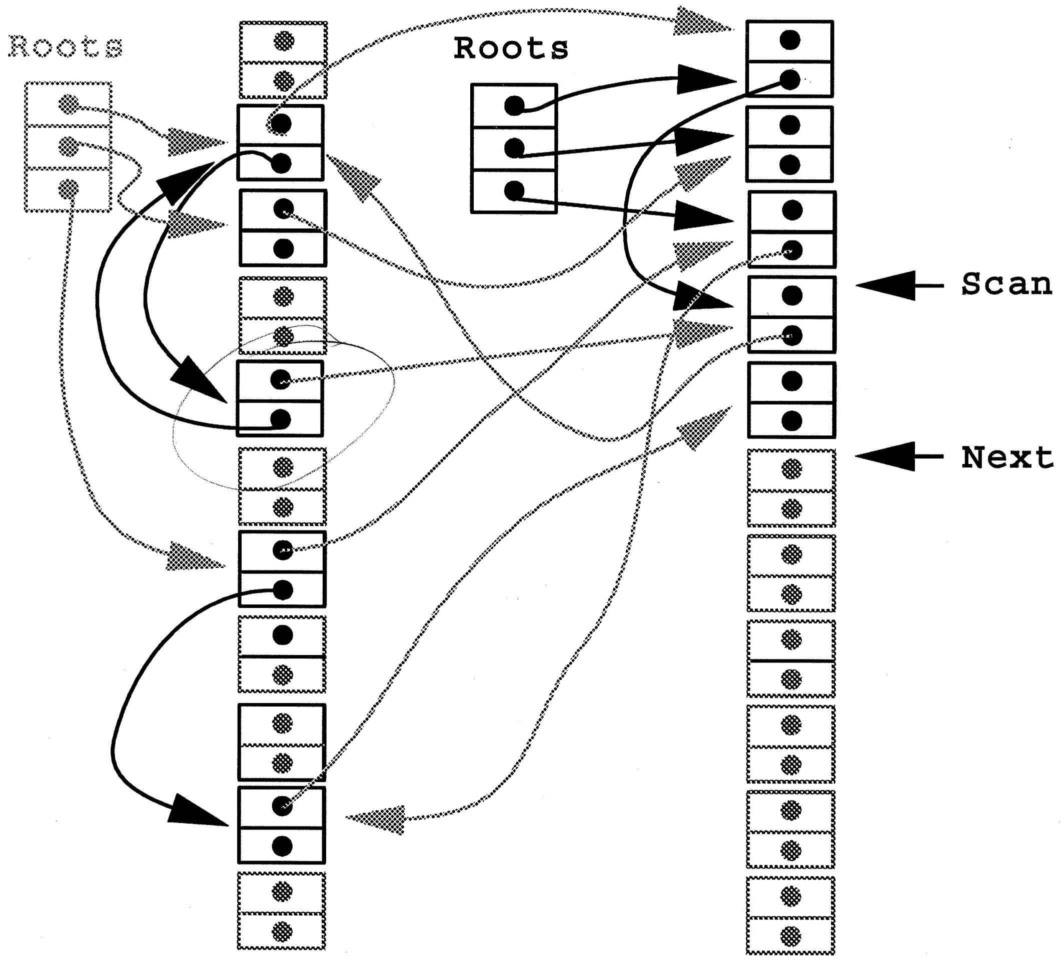
Cheney 1970

- Like Fenichel & Yochelson's algorithm
- Breadth-first search instead of Depth-first
- Keep BFS queue in the "to-space."



Cheney's Algorithm, cont'd

- Breadth-first "Queue" is area between **scan** and **next**.
- When **scan** catches up with **next**, then done!



Forwarding Pointers

```
forward(p)
     IF p is a pointer
     THEN   IF M[p] points to to-space
            THEN  RETURN M[p]
            ELSE  M[next] := M[p]
                  M[next+1] := M[p+1]
                  M[p] := next
                  next := next + 2
                  RETURN M[p]
     ELSE return p

GarbageCollect()
    scan := next := beginning of to-space
    FOR each root r
        DO r := forward(r)
    WHILE scan < next
        DO M[scan] := forward(M[scan])
           scan := scan + 1
```

Advantages of Cheney's Algorithm

- Doesn't touch the garbage
- Leaves free-space contiguous (eliminates fragmentation)
- Requires no auxiliary storage
- Ignores object boundaries in scanning to-space
- Linear time
- Low constant factor
- Simple to implement
- Forms basis of many later algorithms

Disadvantages

- "Wastes" half of memory!
- Breadth-first copy has poor locality

Incremental Collection

Baker 1978

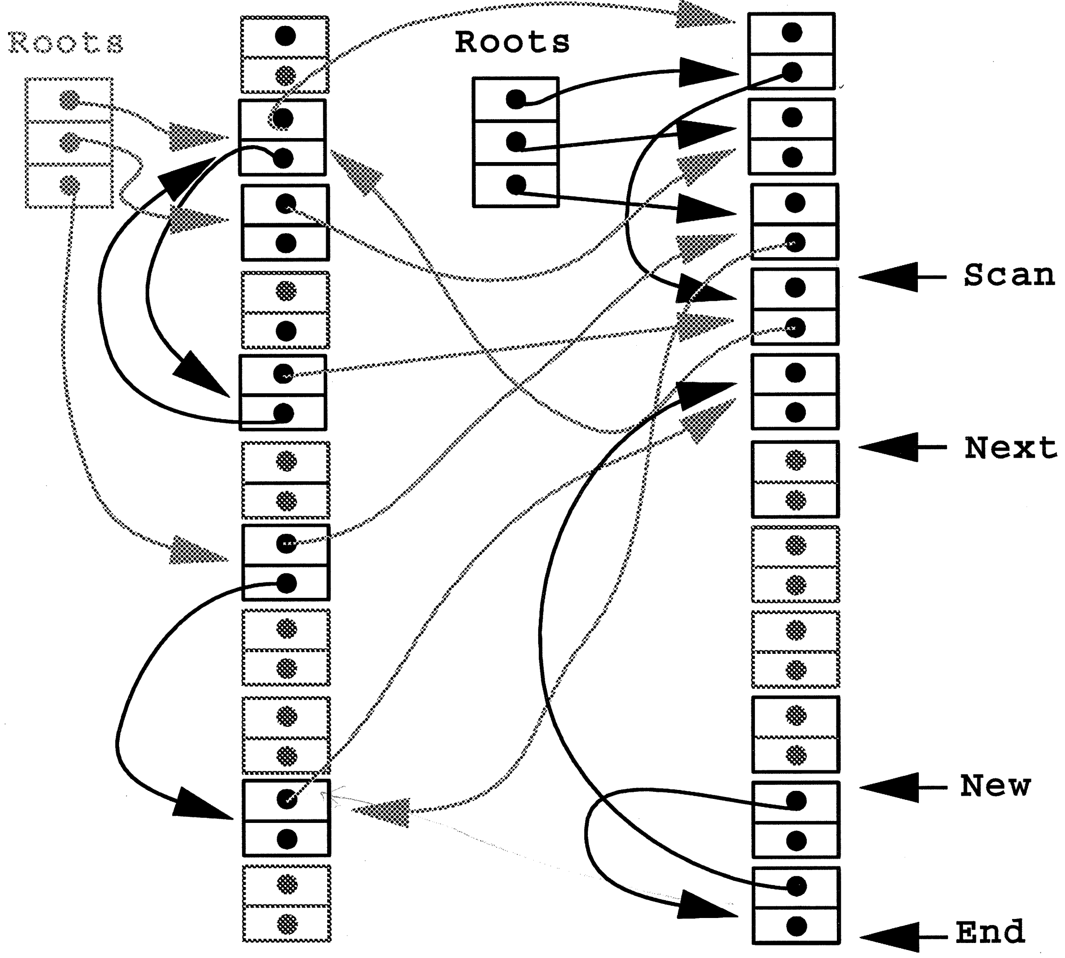
Avoids long waits for garbage collection

- Like Cheney's algorithm, but resume mutator as soon as the roots are forwarded
- Copy a little on every allocation
- INVARIANT:
  registers always point to to-space

To maintain the INVARIANT:
On every fetch, check the register; if necessary, forward the register.

Mutator allocates at end of to-space; freshly allocated cells never need scanning, because of INVARIANT.

Baker's Algorithm, cont'd



Analysis of G.C. Algorithms

Knuth 1967
Hoare 1974
Appel 1987

Assume A accessible cells, memoroy size M.

Mark and Sweep collection:

Mark phase takes $c_1 A$ for depth-first search.
($c_1 =$ size of inner loop.

Sweep phase take $c_2 M$.

Can allocate $M - A$ cells before next collection.

Amortized cost/allocation: $\frac{c_1 A + c_2 M}{M - A}$

Min cost/allocation (limit as $M \to \infty$): c_2

Analysis of Copying Collection

Breadth-first or depth-first copy: $c_3 A$

No sweep phase!

Can allocate $\frac{M}{2}$ cells before next collection.

Amortized cost/allocation: $\frac{c_3 A}{\frac{M}{2} - A}$

Min cost/allocation (limit as $M \to \infty$): 0

No inherent lower bound on the cost of garbage collection!

Baker's algorithm Copies, scans exactly the same set of cells as Stop & Copy; but high overhead on mutator to check each FETCH.

The cost of explicit freeing

"I had to stand on my head getting the calls to **free()** in the right place, but at least my program runs faster because there's no painful garbage collection overhead."

C. Hacker

Not necessarily true!

Why is explicit deallocation nasty?

- Graph structures have "sharing"

- Crosses boundaries of abstract data types

- Hard to make a "safe" programming language that has explicit deallocation.

Generational Collection

Liebermann & Hewitt 1983

Two observations:

1. New objects more likely to die
2. Newer objects point to older objects

Focus the G.C. effort on the newer objects

- Partition memory into *Generations* G_i
- Frequently copy youngest generation G_0
- Rarely copy older generations

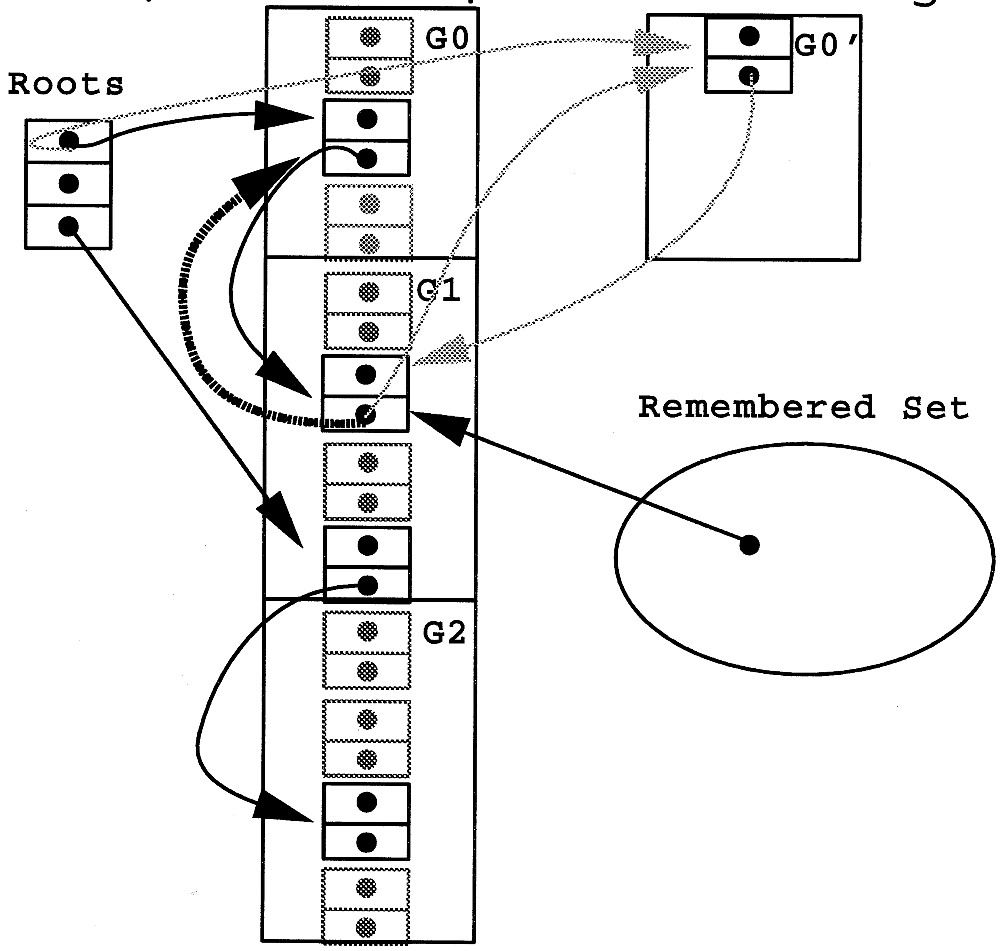
To copy a set of generations $G_0 \ldots G_i$:

- Forward the "roots"
- Make *scan* catch up with *next*

What *are* the roots?

- Registers, local variables, etc.
- Pointers from other generations (but observation 2 assures that these are rare).

Must keep track of updates to existing cells



Keeping track of Updates

- With compiler assistance [Ungar 1986]
  Keep a vector of pointers to modified cells. With each update to an existing cell, compiler generates code to put cell in vector.
- With virtual memory [Moon 1984, Shaw 1987]
  Make older generations read-only. On a page fault, make page writeable and put page in a set of modified pages.

Scan each modified cell (or page) for roots on each younger-generation collection.

Generation collection makes **allocate** cheaper, but **update** more expensive. Lisp, ML, Smalltalk, etc. have more **allocate**s than **update**s.