Automatic Memory Management

Storage Allocation

- Static Allocation
  - Bind names at compile time
  - Pros:
    - Fast (no run time allocation, no indirection)
    - Safety : memory requirements known in advance
  - Cons:
    - Sizes must be known at compile time
    - Data structures can't be dynamically allocated
    - No recursion

Storage Allocation

- Stack Allocation
  - activation records (frames)
  - push + pop on proc entrance / exit
  - Implications:
    - Recursion is possible
    - Size of local data structures may vary
    - Stack allocated local names can't persist
    - Can only return objects of statically known size
    - Enables function pointers (no nesting though)

Storage Allocation

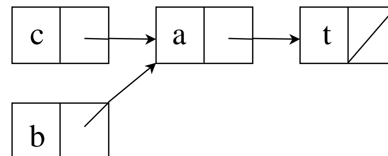
- Heap Allocation
  - Alloc and dealloc from a heap in any order
  - Advantages:
    - Local data structures can outlive procedure
    - Facilitates varying sized recursive data structures
    - Can return dynamically sized objects
    - Closures (function + environment)

Reachability

- What can a program manipulate directly?
  - Globals
  - Locals (in registers, on stack, etc)
  - In C, random locations
- Root nodes
- Live nodes - pointer reachability

Problems w/ Manual Allocation

- Garbage - “unreachable” but not free
- Dangling references
- Sharing



- Failures
  - Invalid accesses, out of memory errors, etc...

Why else would we want AMM?

- Language requirements
  - sharing (system does sharing to save space)
  - delayed execution
- Problem requirements
  - Should pop() dealloc? Sometimes…
- More abstraction, easier to understand
- Manual management is **hard**.

Tradeoffs

- Problem specification (hard real time)
- Costs (time + space)
  - Traditionally very slow
    - early 80's - 40% of time in large LISP programs
    - typical: 2-20%

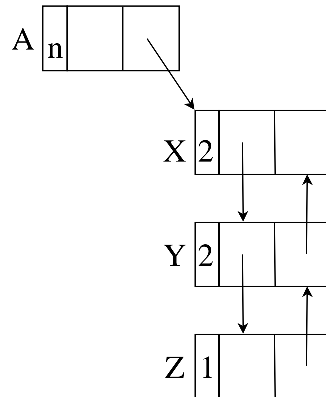
Reference Counting

- Count the number of references to each node
- Each mode has a field for the count (rc)
- Free nodes: rc = 0
- On referencing, rc++
- On dereferencing, rc--
- When rc returns to 0, free it.

Reference Counting

- Advantages
  - Overhead is distributed
  - Probably won't affect locality of reference
  - Little data is shared, most is short-lived
- Disadvantages
  - High overhead on mutator operations (rc++.rc--)
  - Recursive freeing
  - Can't reclaim cyclic structures (Why?)

Cyclic Structures



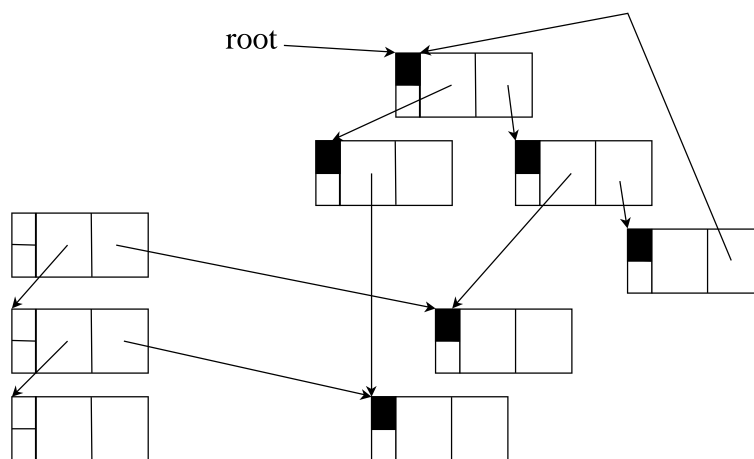
Cyclic Structures

- Look for "cycle making" references
  - 2 invariants:
    - active nodes are reachable from root by a chain of "strong pointers"
    - strong pointers do not form cycles
  - Non termination
  - Reclaiming objects prematurely

Mark and Sweep

- *Garbage collection*
- Leave stuff unreachable until a collection
- Suspend program during a collection
- Mark nodes reachable from the roots
- Sweep up the garbage

Mark and Sweep



Mark and Sweep

```
mark_and_sweep:
   for each R in Roots:
      mark(R)
   sweep()
mark(N):
   N.mark = MARKED
   for each C in N.children:
      mark(C)
```

Mark and Sweep

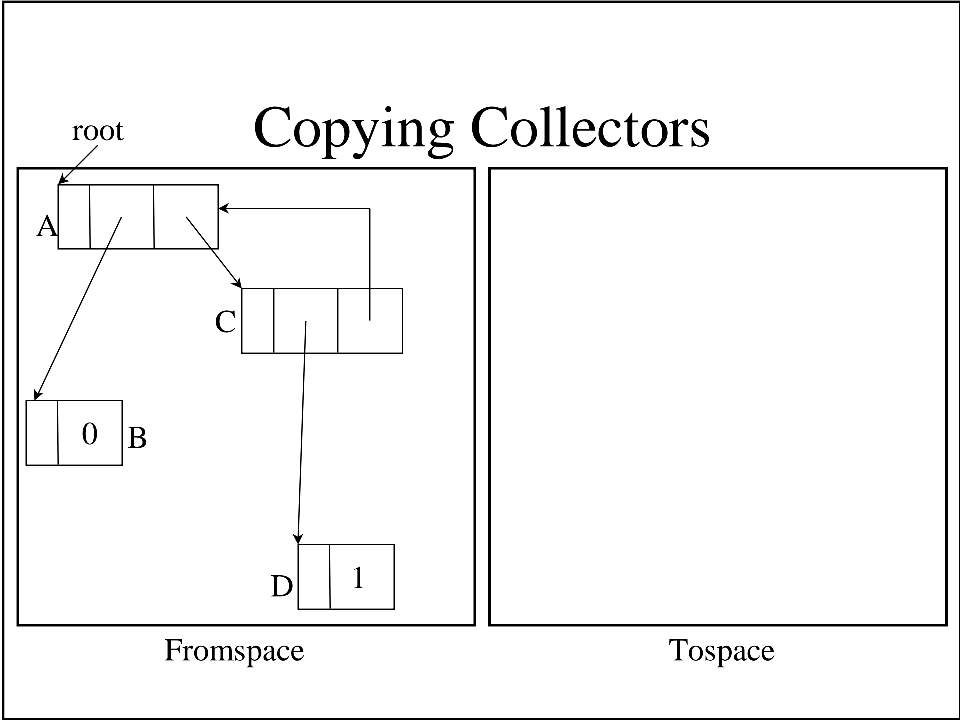
```
sweep():
   Free_List = []
   for each Obj in Heap:
       if Obj.mark == UNMARKED:
          Free_List.append(Obj)
       else:
          Obj.mark = UNMARKED
```

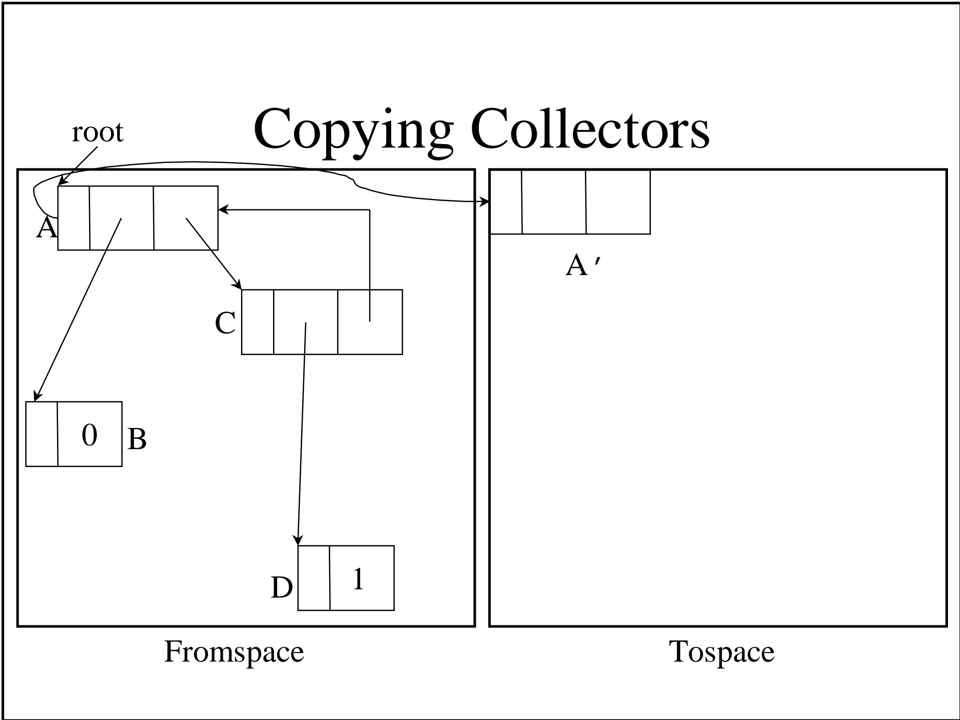
Mark and Sweep

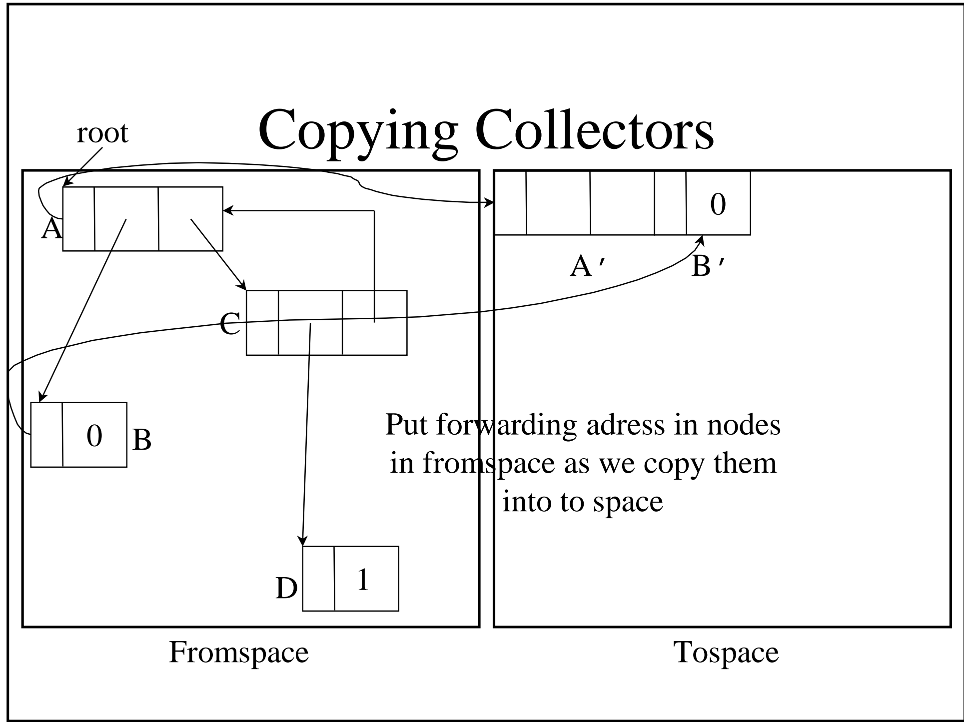
- Advantages:
  - Cycles are handled naturally
  - No overhead on pointer manipulations
- Disadvantages:
  - Computation halts
  - Potentially long pauses (O(seconds))
  - Locality
  - Fragmentation

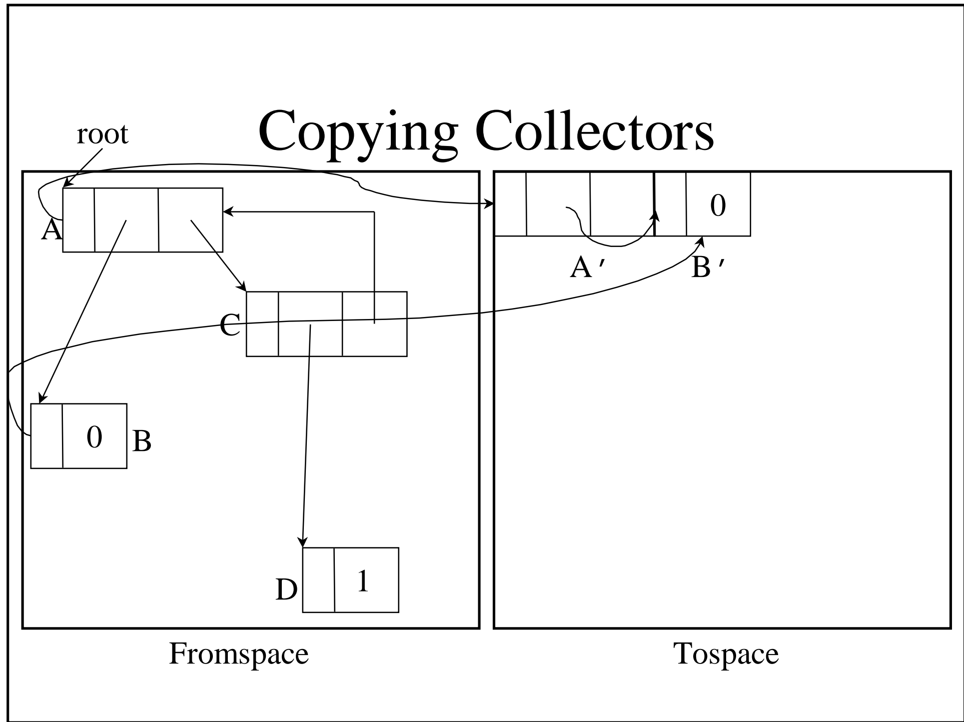
Copying Collectors (scavenging)

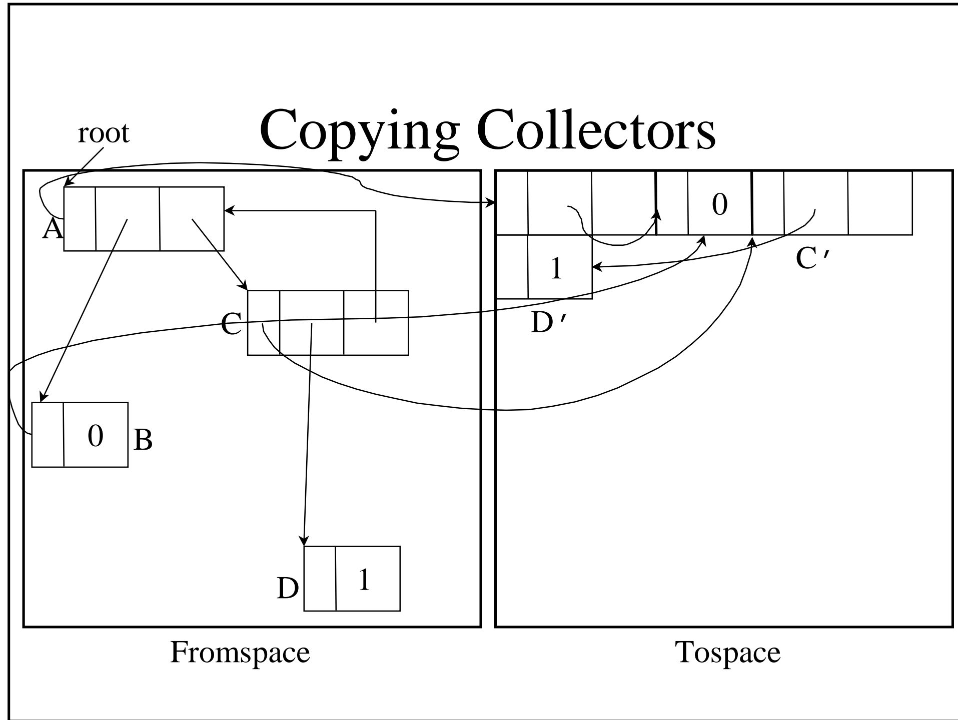
- Divide heap into two semi-spaces
- Allocate only into one space at a time
- On collection, copy out live data

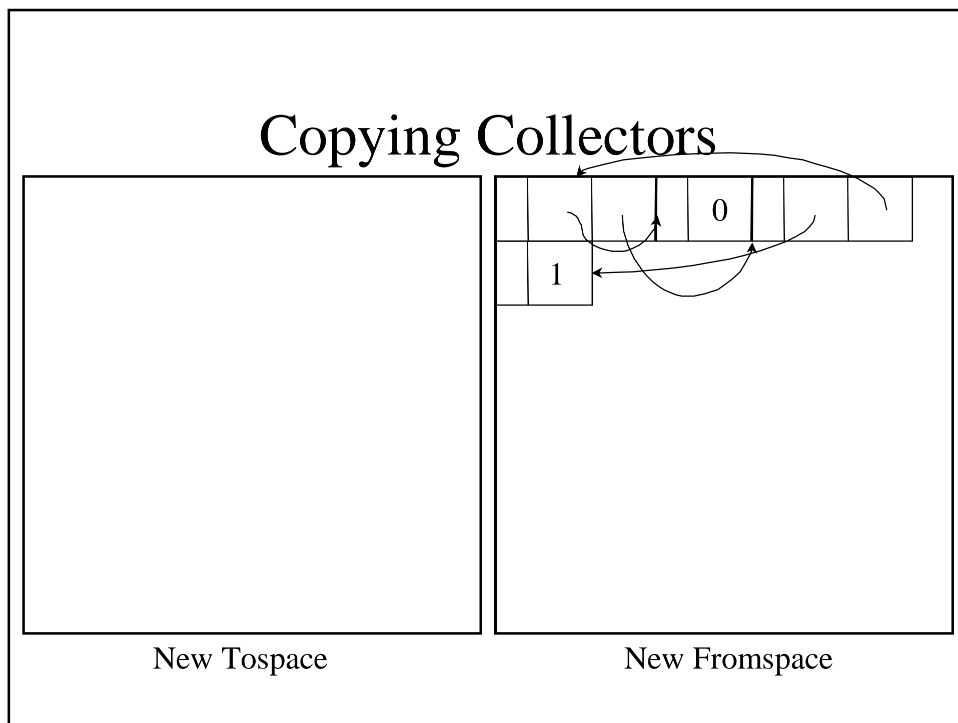

Copying Collectors
root
A
C
0
B
D
1
Fromspace
Tospace


Copying Collectors
root
A
A′
C
0
B
D
1
Fromspace
Tospace


Copying Collectors
root
A
0
A′
B′
C
0
B
Put forwarding adress in nodes
in fromspace as we copy them
into to space
D
1
Fromspace
Tospace


Copying Collectors
root
A
0
A′
B′
C
0
B
D
1
Fromspace
Tospace


Copying Collectors
root
A
C
B
0
D
1
0
1
C′
D′
Fromspace
Tospace


Copying Collectors
0
1
New Tospace
New Fromspace

Copying Collectors

- Advantages:
  - No fragmentation
  - Only touch cells in use
  - No free list
- Disadvantages:
  - 1/2 your memory is always unused
  - Overhead of copying
  - Copy long-lived objects every time

Other Algorithms

- The previous algorithms are naïve
- Solutions for most problems:
  - Incremental + Concurrent collection
  - Tricolor marking
    - Black: visited
    - Grey: mutated, or not fully traversed
    - White: untouched
  - Generational collection
    - collect newer spaces; not older ones

Garbage Collection for C/C++

- Don't want to recompile code
- Distinguishing pointers w/ a bit flag is bad
- Headers in general are bad
- Where are the roots?
- What are the stack/register conventions?
- Which words are pointers?

Conservative Collection

- Roots: registers, stack, static areas.
- Pointer tests:
  - Does the "address" point into the heap?
  - Has that heap block been allocated?
  - Is the address properly aligned?
    - Machine dependent
- Pointer tests on sparc: 30 instructions or so
  - every time to determine if something is a ptr