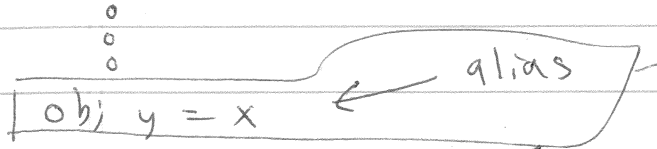
10-2 | 100% completeness at low low prices

1. obj x = malloc();



line 87.

88. if (f()) { — is x live (is x's extent ⊂ [88,91]?

89. print x → g(12) → h(19, 20)

90. }

91. exit

COMPLETEness depends on knowing what f() can return

TM-Halts(M)

---

malloc()/free() as a person (Jay Method)

- sound — possible, but mistakes happen → NO
- completeness — anything (leak = diff) (mem hog = Δ>>)
- mem — lg n + lg σ
- time — lg n
- latency — naive → high latency possible/queuing → low latency

11-1 | Smart-Pointers use a "reference" count

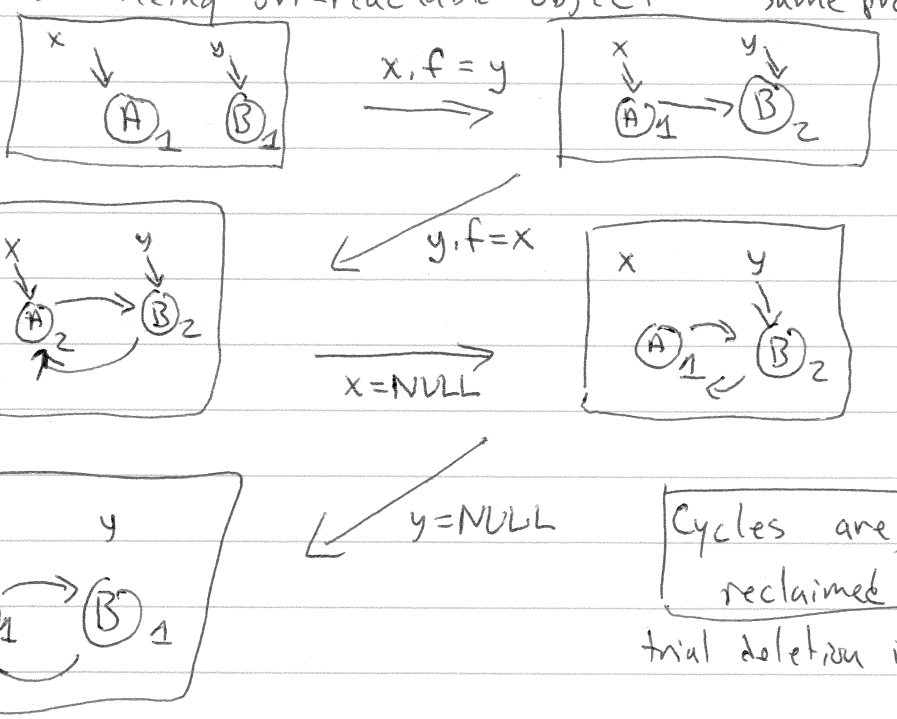
"o.f = ptr" → ptr's count to inc
when a new ref/alias is made

"o.f = NULL" → o.f is no accessible
"o.f = ptr2" → ptr's count decreases
"return false" → ptr is not accessible

when count == 0, free()

Sound → Yes. * In some impl, there's human trust/frailty

Complete → freeing un-reachable object ↓ same problem




```
retain(p)
    if (p.count ≠ 255)
        p.count++
release(p)
    if (p.count ≠ 255)
        p.count--
    if (!p.count)
        free(p)
```

Cycles are not reclaimed
trial deletion is expensive

mem efficiency → store the count
if small, infinite extend obj (due to modulo)
small objs have a great % memory cost
most objs are small

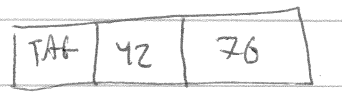
time → same as manual
cost ∝ work (vs. size)

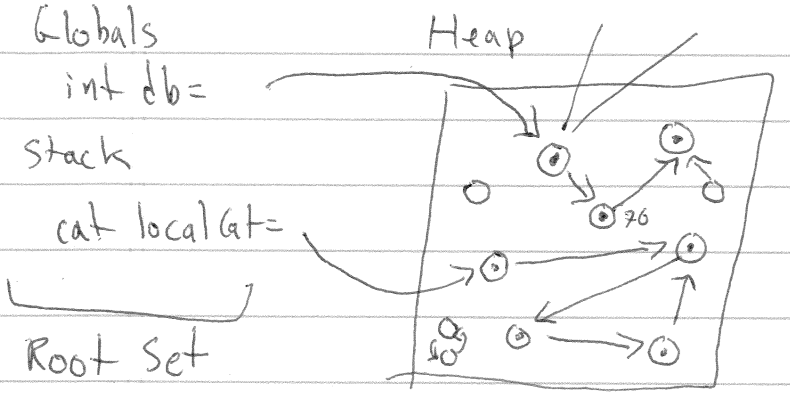
latency → same as manual
due to cascading

Mark + Sweep

"Garbage Collection"
↳ a correct memory manager
sound ↑ as complete as can be

```
for o in Object-Graph
    mark(o)
for o in memory
    if !marked(o)
        free(o)
```





how does GC know the roots?
how does GC know obj layout?
↳ tag, BiBoP
where is the mark?
↳ tag or bitmask per page

time ⇒ malloc is O(lg n) / free is O(n) vs O(n lg n)
mem ⇒ tags
time ⇒ mark is O(live) / sweep O(memory)
sound + complete / as can be

Hans Böhm GC for C