Programmiertechnik II

Automatische Speicherverwaltung

Speichermüll

- Objekte verweisen über Referenzen auf andere Objekte
  - ➡ Objekte bilden Graph
- Explizites Freigeben von Objekten
  - C: free(<pointer>)
  - C++: delete <pointer>
- Problem: Freigeben eines Objekts führt zu Nicht-Erreichbarkeit anderer Objekte
  - C++: Destruktor gibt rekursiv weitere Objekte frei
    - Problem: Mehrfache Verweise auf dasselbe Objekte
      - “dangling pointer”: Speicher ist bereits freigegeben, aber es gibt noch Pointer auf das Objekt
      - “double free”: Objekt wird mehrfach freigegeben
- Automatische Speicherverwaltung: Allozierung explizit, Deallozierung automatisch

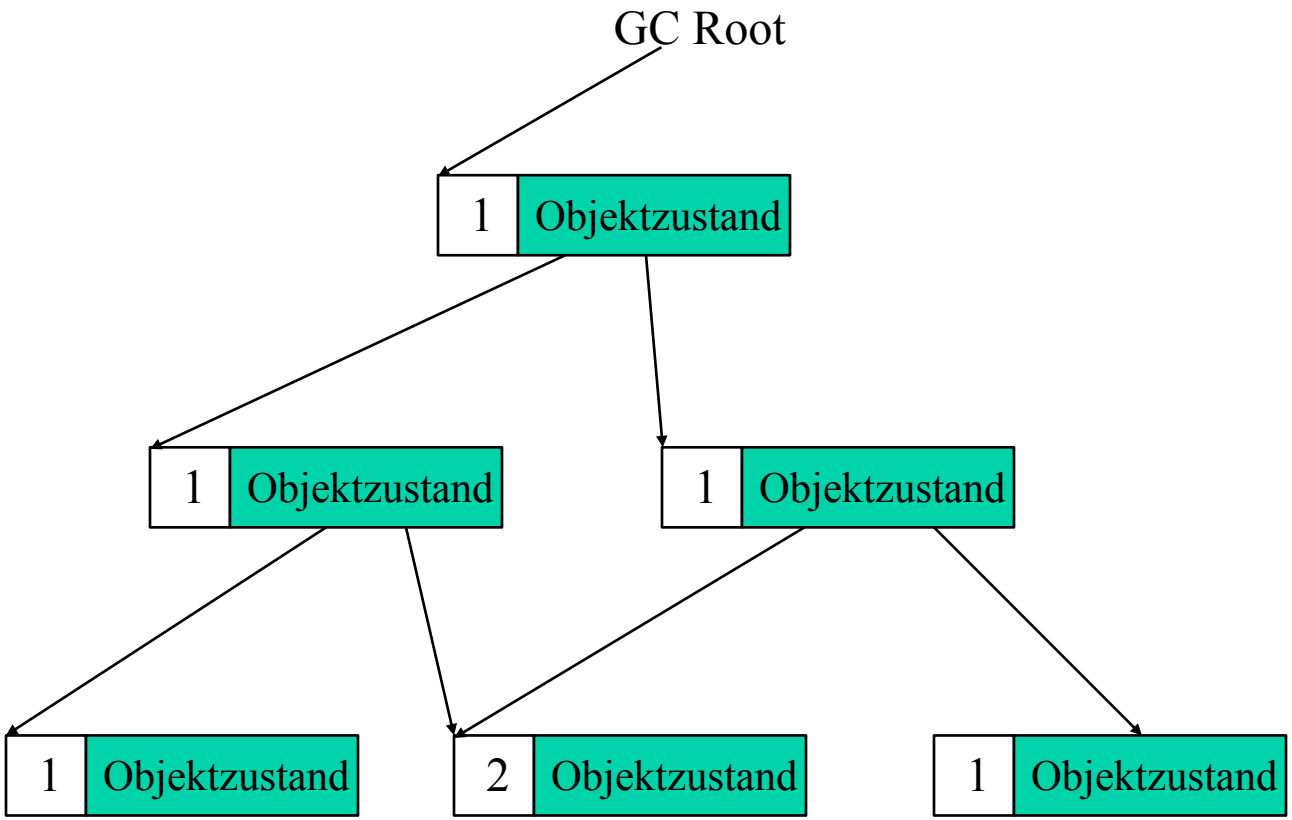
Erreichbarkeit und Lebendigkeit

- inherent lebendige Objekte
  - globale Variablen
  - "*GC roots*"
  - Java
    - laufende Threads
    - lokale Variablen aller aktiven Funktionen
    - Systemklassen (z.B. java.lang.String)
    - "native roots"
- erreichbare Objekte
  - von *GC root* referenziert
  - von erreichbarem Objekt referenziert
- erreichbare Objekte sind u.U. ebenfalls nicht mehr verwendet
  - automatische Speicherverwaltung muss diese Objekte trotzdem aufheben

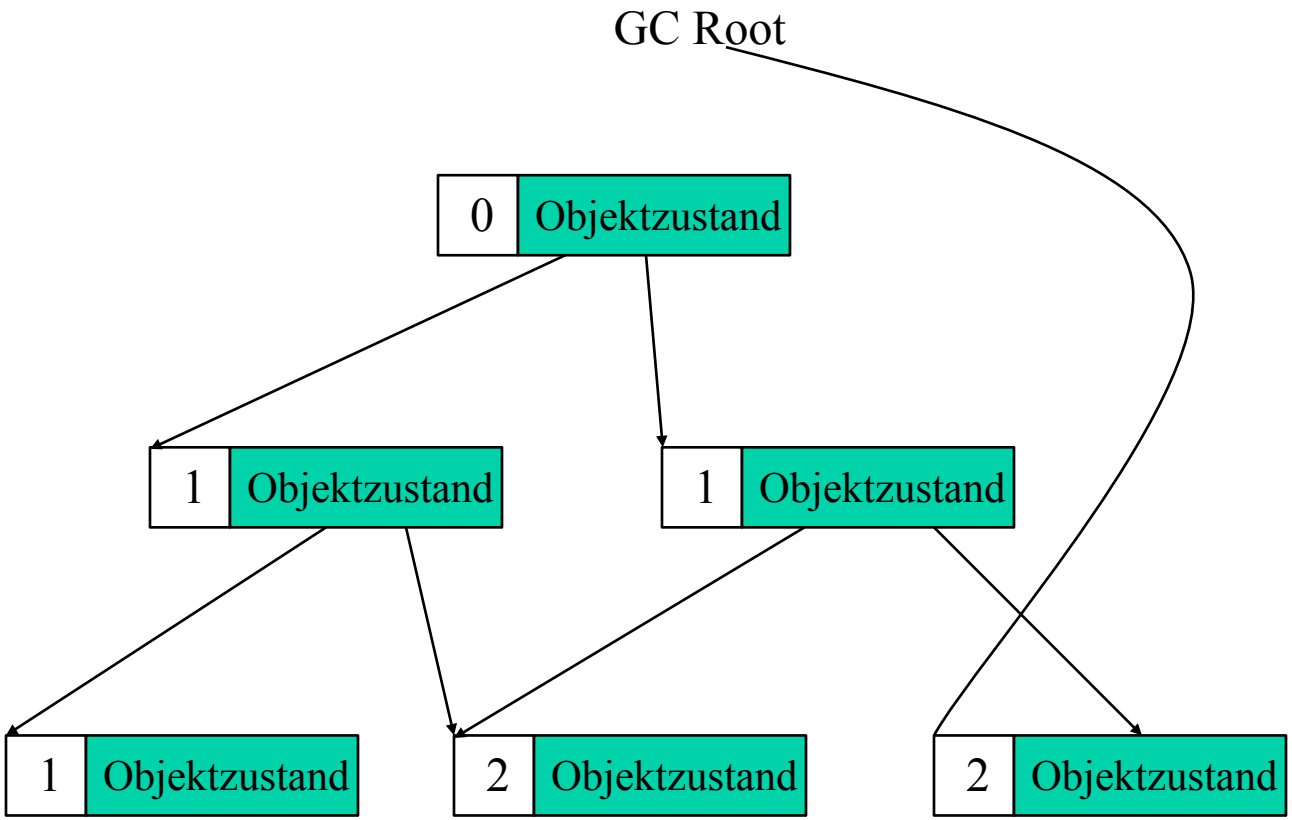
Referenzzählung

- Collins 1960
  - A Method for overlapping and erasure of lists, CACM 3:655-657, 1960
- Idee: Jedes Objekt enthält Zähler der Referenzen auf das Objekt
  - Referenzen entweder in anderen Objekten, oder in lokalen oder globalen Variablen
  - Objekt entsteht mit Referenzzähler 1
  - Erzeugen einer Referenz auf ein Objekt erhöht Referenzzähler
  - Ändern einer Referenz senkt Zähler
  - Wird der Referenzzähler 0, wird das Objekt freigegeben
    - In Folge müssen alle Zähler referenzierter Objekte freigegeben werden

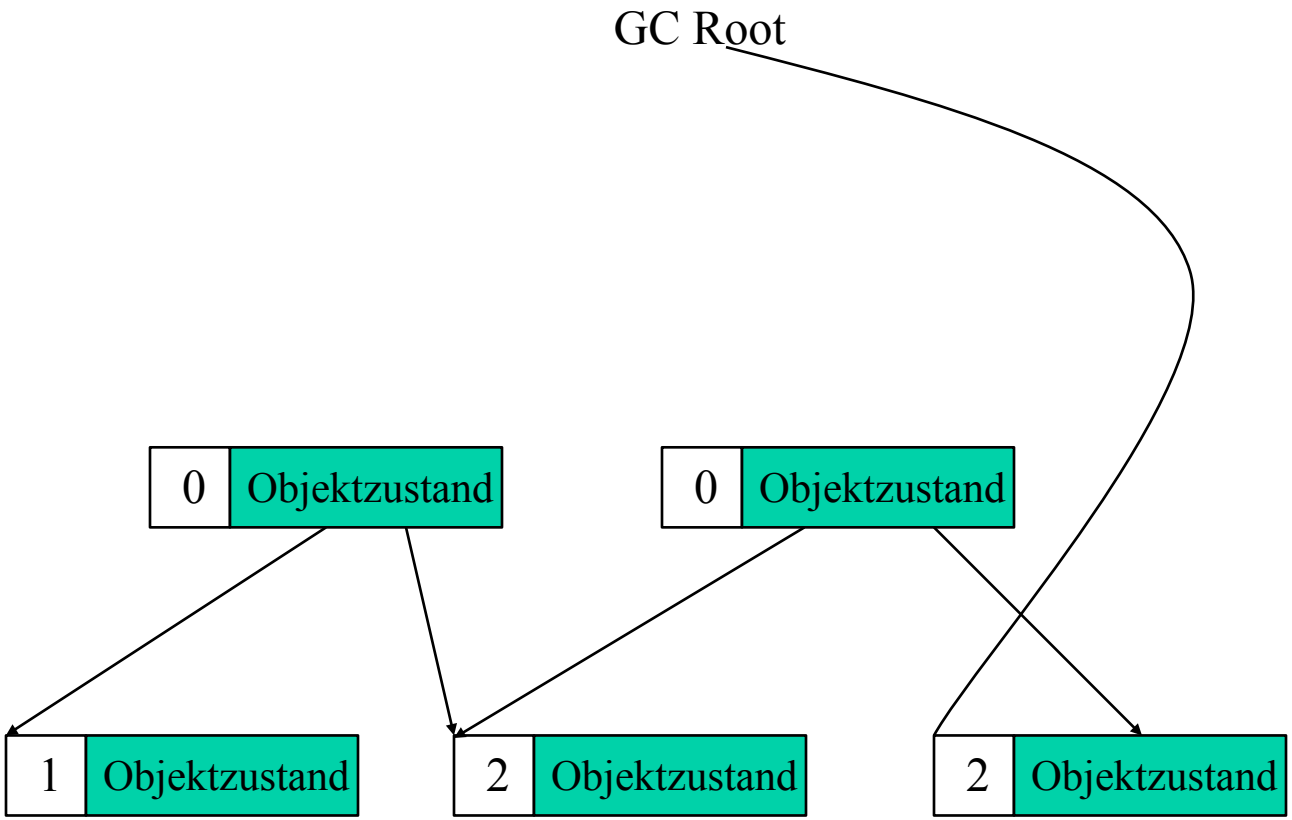
Referenzzählung



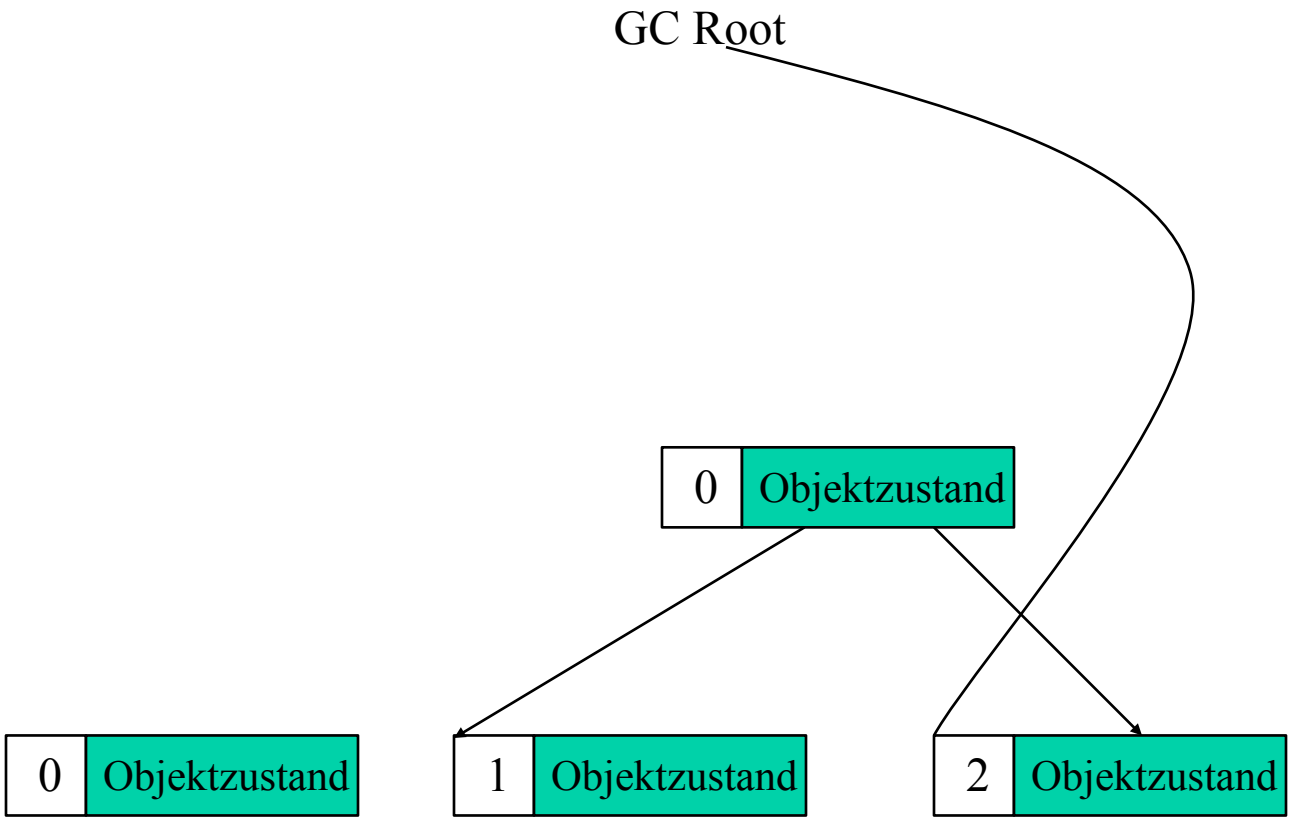
Referenzzählung



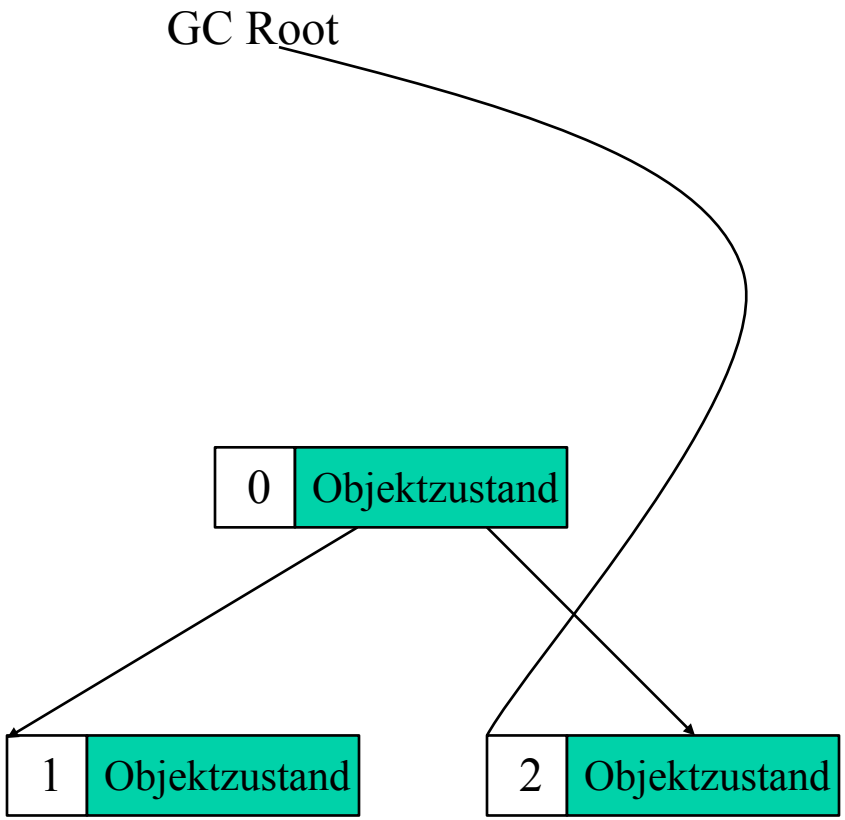
Referenzzählung



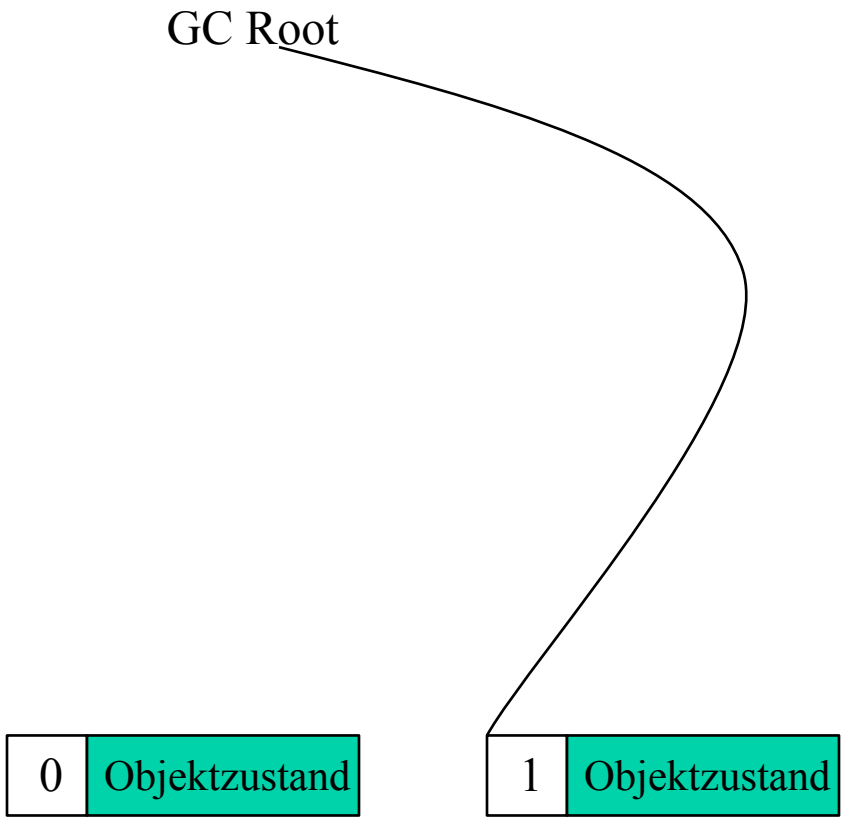
Referenzzählung



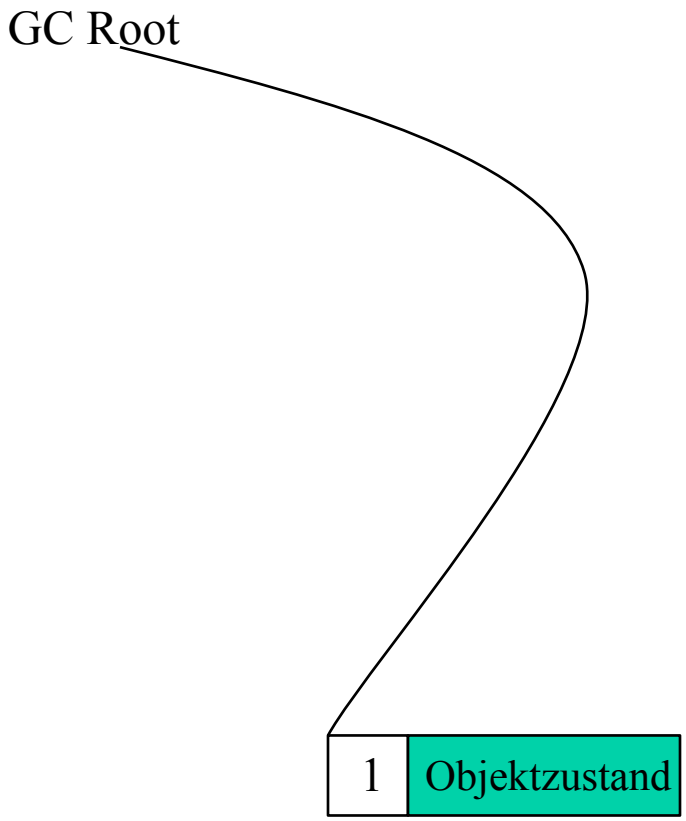
Referenzzählung



Referenzzählung

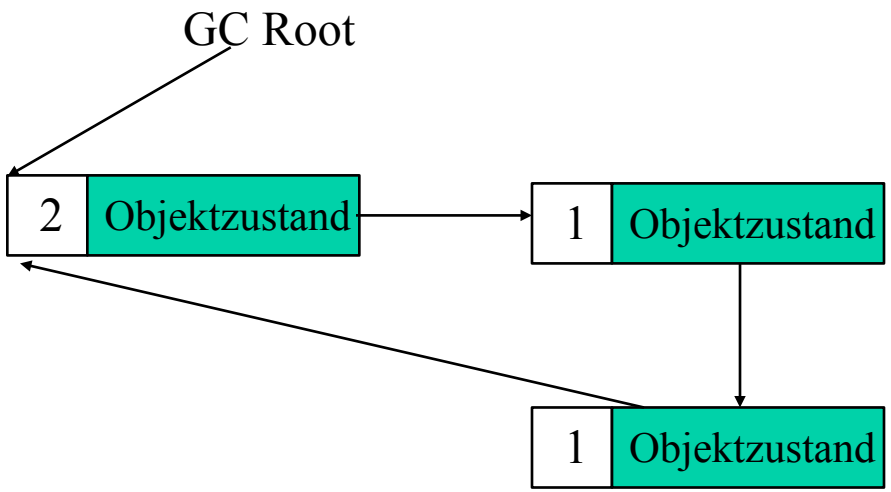


Referenzzählung



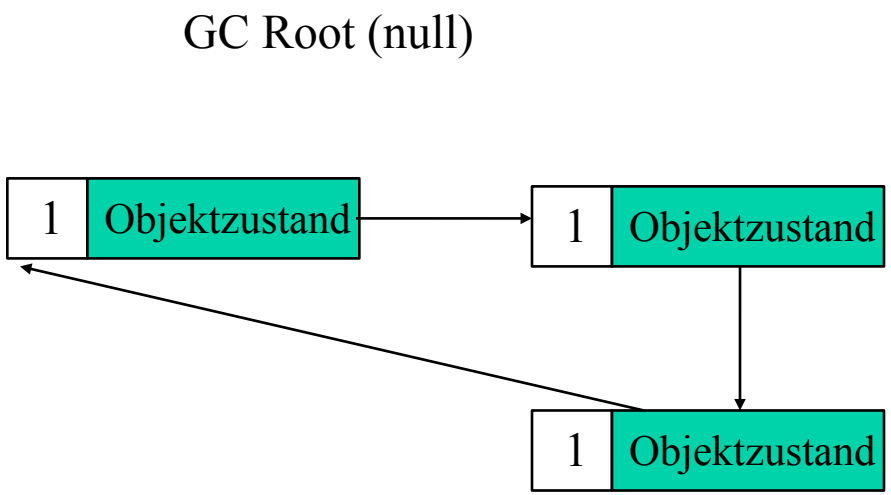
Zyklische Strukturen

- Objekte verweisen gegenseitig aufeinander
  - Beispiel: doppelt verkettete Liste, Baum mit Elternreferenz in jedem Knoten
- Referenzzähler wird nie 0



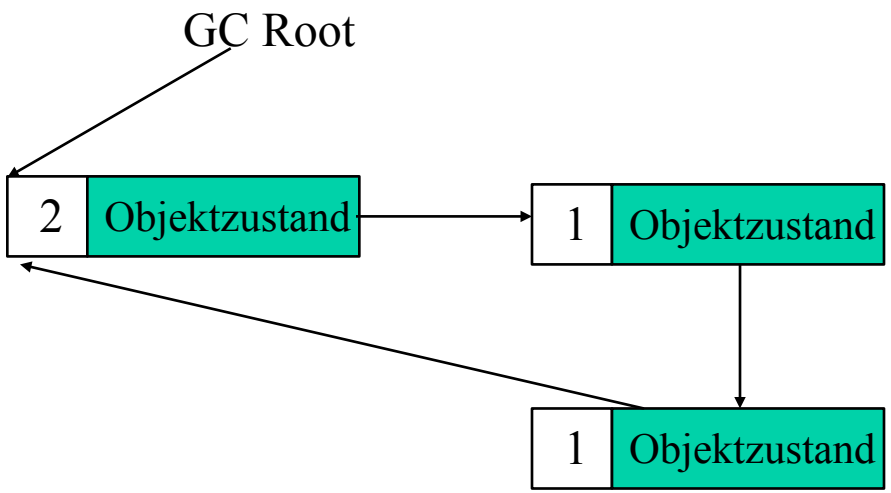
Zyklische Strukturen

- Objekte verweisen gegenseitig aufeinander
- Referenzzähler wird nie 0



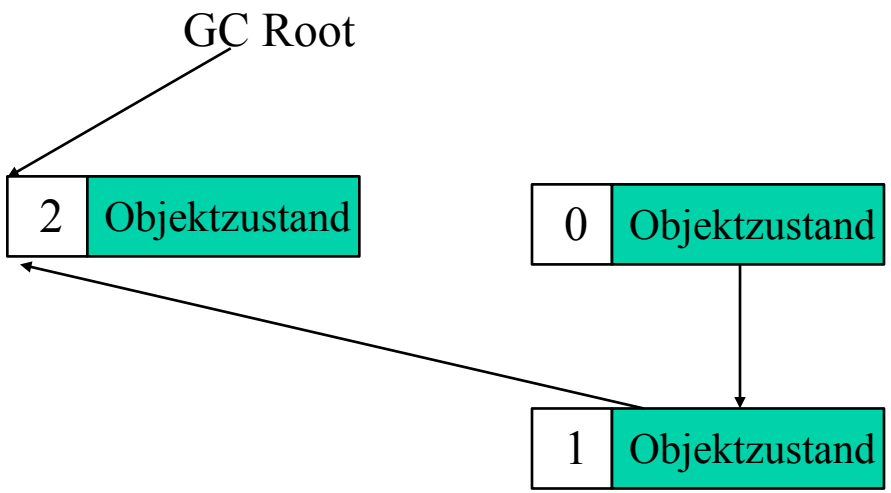
Zyklische Strukturen

- Lösung: Zyklus explizit aufbrechen
- GCRoot.next = null



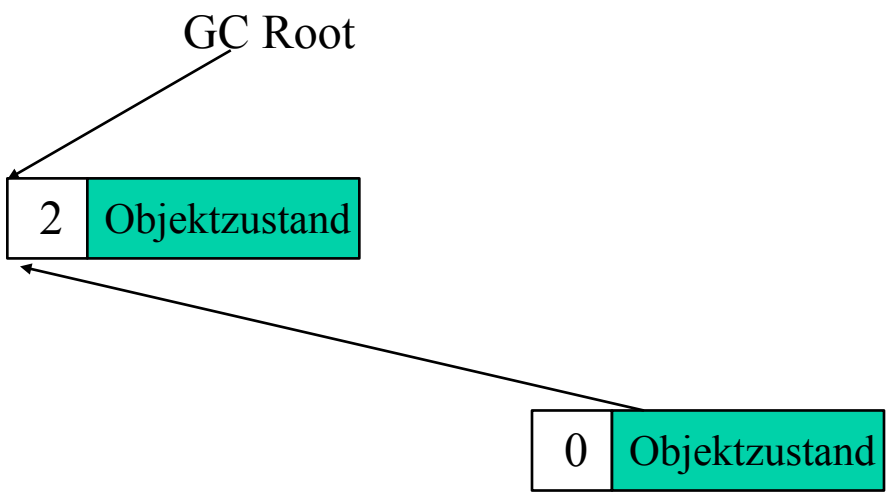
Zyklische Strukturen

- Lösung: Zyklus explizit aufbrechen



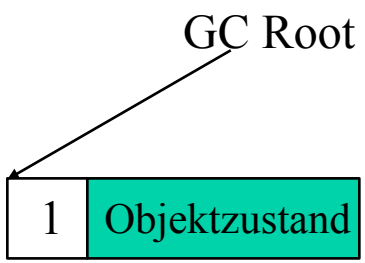
Zyklische Strukturen

- Lösung: Zyklus explizit aufbrechen



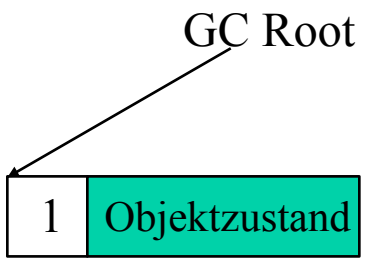
Zyklische Strukturen

- Lösung: Zyklus explizit aufbrechen



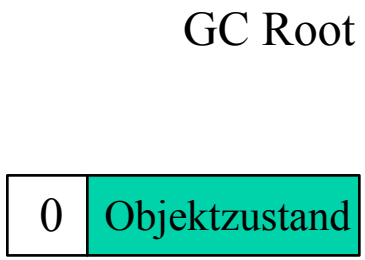
Zyklische Strukturen

- Lösung: Zyklus explizit aufbrechen
- GCRoot = null



Zyklische Strukturen

- Lösung: Zyklus explizit aufbrechen



Zyklische Strukturen

- Lösung: Zyklus explizit aufbrechen

GC Root (null)

Referenzzählung

- Vorteile:
  - Objekte werden sofort freigegeben, wenn keine Referenz mehr existiert
    - Für die meisten Objekte gibt es nur eine oder “wenige” Referenzen
  - Kosten der Speicherverwaltung werden gleichmäßig auf Operationen verteilt
    - potenziell echtzeitfähig
    - aber: die Freigabe eines Objekts kann rekursiv die Freigabe weiterer Objekte bewirken
- Nachteile:
  - zyklische Strukturen werden nicht automatisch beseitigt
    - explizite Freigabe möglich
    - Python: zusätzlich GC für zyklische Strukturen (erstmalig von Weizenbaum 1969 vorgeschlagen)
  - Kosten pro Operation relativ hoch
    - Thread-Sicherheit?: atomares Inkrementieren, dekrementieren
  - Zusätzlicher Speicher für Referenzzähler pro Objekt

Garbage Collection

- Auffinden aller erreichbarer Objekte beginnend bei GC Roots
- Freigeben aller nicht-erreichbarer Objekte
- Optional: Beseitigung der Fragmentierung durch Verschieben der erreichbaren Objekte (*compacting GC*)
  - Aktualisierung aller Zeiger auf erreichbare Objekte
- Häufigkeit von GC
  - Wenn Speicher erschöpft ist
  - Regelmäßig: nach n sec
  - Regelmäßig: nach m Objektallokationen
  - …

Mark-and-Sweep GC

- McCarthy, 1960
  - Recursive functions of symbolic expressions and their computation by machine, CACM 3:184-195, 1981
- Zwei Phasen: Mark, Sweep
- Durchwandern erreichbarer Objekte
  - jedes gefundene Objekt wird markiert
- Rekursives Weiterverfolgen aller Objektreferenzen
  - Abbruch, wenn referenziertes Objekte bereits markiert ist
- Sweep: Löschen aller Objekte, die nicht markiert sind

Mark-and-Sweep GC

```
def New():
    if free_pool is Empty:
        mark_and_sweep()
    return allocate()

def mark_and_sweep():
    for R in Roots:
        mark(R)
    sweep()
    if free_pool is Empty:
        raise “Memory exhausted”
```

Mark-and-Sweep GC

```
def mark(o):
    if not mark_bit(o) :
            set_mark_bit(o)
            for C in children(o):
                mark(C)

def sweep():
    N = Heap_bottom
    while N < Heap_top:
        if mark_bit(N):
            clear_mark_bit(N)
        else:
            deallocate(N)
        N += size(N)
```

Mark-and-Sweep GC

- Implementierungsaspekte:
  - Repräsentation des Mark-Bits
    - separater Speicher
    - freies bit im Objekt
  - Rekursiv oder iterativ
    - Stacküberlauf bei tief verschachtelten Datenstrukturen
    - iterativ: Speicherung einer Liste noch zu bearbeitender Objekte
- Vorteile gegenüber Referenzzählung
  - Unterstützung für Zyklen
  - keine Laufzeitkosten bei Pointermanipulation
- Nachteile:
  - Stopp-Start-Algorithmus: Berechnung wird für GC unterbrochen
  - Laufzeitkosten schwer vorhersagbar (abhängig von Zahl allozierter Objekte)

Copying GC

- Minsky 1963
  - A Lisp garbage collector algorithm using serial secondary storage. Technical Report Memo 58, Project MAC, MIT, 1963
- Idee: Speicher wird in zwei Halbräume unterteilt
  - “aktuelle”, “alt”
- Allokationen reservieren Speicher im aktuellen Halbraum
- GC: Halbräume tauschen ihre Rollen
  - lebendige Objekte werden vom alten Raum in den aktuellen Raum kopiert
  - Referenzen in kopierten Objekten werden rekursiv verfolgt, Objekte werden kopiert
- Nicht kopierte Objekte sind einfach verworfen
- Copying GC kompaktiert Speicher

Copying GC

- Implementierungsaspekte
  - Repräsentation von Objektadressen
    - Globale Objekttabelle: Aktualisierung der Pointer vom alten Halbraum auf den neuen Halbraum
    - Pointer: altes Objekt kann Pointer auf neues Objekt enthalten
      - beim Kopieren der zweiten Referenz auf ein Objekt muss dann nur die Adresse aktualisiert werden
- Vorteile
  - effiziente Allozierung: Speicher stets am Ende des aktuellen Halbraums
  - keine Fragmentierung des Speichers
- Nachteile
  - ineffiziente Speichernutzung
  - Kopieren ganzer Objekte ist aufwendig

Varianten

- Mark-Compact: erreichbare Objekte werden in die Lücken verschoben
- Objektgenerationen (*generational GC*)
  - “most objects die young”
  - junge Objekte verweisen i.d.R. auf alte, nicht umgekehrt
  - Objekte werden in Generationen unterteilt, GC betrachtet meist nur die “jüngste” Generation
    - erreichbare Objekte der jüngsten Generation (*survivors*) rücken in die nächste Generation auf (*promotion*)
  - GC Roots: auch Referenzen aus älteren Generationen
    - Python: Vergleich des Referenzzählers mit Zahl der Referenzen in einer Generation
- Vermeidung von Rekursion
- Tiefe-zuerst oder Breite-zuerst
- Garbage Collection im Hintergrund (multi-threading)
  - inkrementelle GC

Verfolgung von Referenzen

- Welche Hauptspeicherzellen enthalten Pointer?
- konservativ: Was wie ein Pointer aussieht, ist ein Pointer
- exakt: Meta-Daten definieren Objektlayout; GC kann exakt ermitteln, an welchen Offsets sich Referenzen auf andere Objekte befinden
  - Meta-Daten auch über lokale Variablen

Finalisierung

- Bevor ein Objekt freigegeben wird, wird eine Finalisierungsroutine aufgerufen
  - explizite Speicherverwaltung (C++): Freigabe referenzierter Objekte
  - automatische Speicherverwaltung: Freigabe lediglich externer Ressourcen (etwa Schließen von Fenstern, Dateien, Datenbankverbindungen, …)
- erfordert Kenntnis aller freigegebenen Objekte
  - Copying GC: Traversierung des alten Halbraums
- Wiederherstellung der Erreichbarkeit in Finalisierung (*resurrection*)
  - Java: Finalisierung wird nur einmal ausgeführt