AA 2015-2016

28. Garbage collection

Gestione della memoria

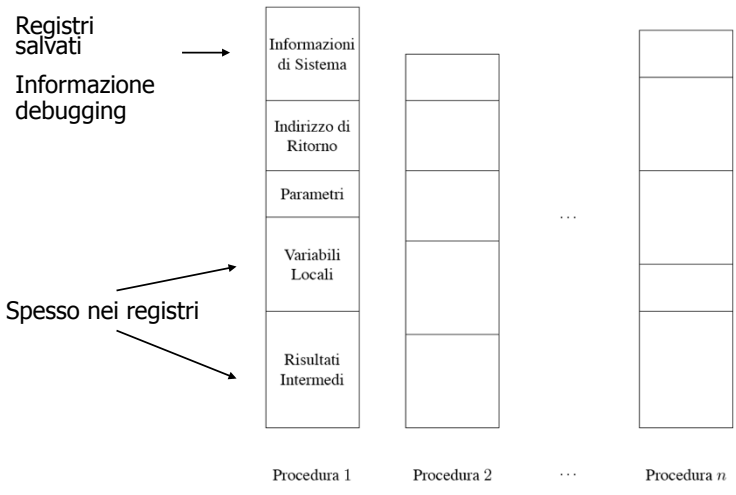
- Static area
  - dimensione fissa, contenuti determinati e allocati a compilazione
- Run-time stack
  - dimensione variabile (record attivazione)
  - gestione sottoprogrammi
- Heap
  - dimensione fissa/variabile
  - supporto alla allocazione di oggetti e strutture dati dinamiche
    - ✓ **malloc** in C, **new** in Java

Allocazione statica

- Entità che ha un indirizzo assoluto che è mantenuto per tutta l'esecuzione del programma
- Solitamente sono allocati staticamente
  - variabili globali
  - variabili locali sottoprogrammi (senza ricorsione)
  - costanti determinabili a tempo di compilazione
  - tabelle usate dal supporto a run-time (per type checking, garbage collection, ecc.)
- Spesso usate in zone protette di memoria

3

Allocazione statica per sottoprogrammi



4

Allocazione dinamica: pila

- Per ogni istanza di un sottoprogramma a run-time abbiamo un record di attivazione contenente le informazioni relative a tale istanza
- Analogamente, ogni blocco ha un suo record di attivazione (più semplice)
- Anche in un linguaggio senza ricorsione può essere utile usare la pila per risparmiare memoria. Perche?

5

Allocazione dinamica con heap

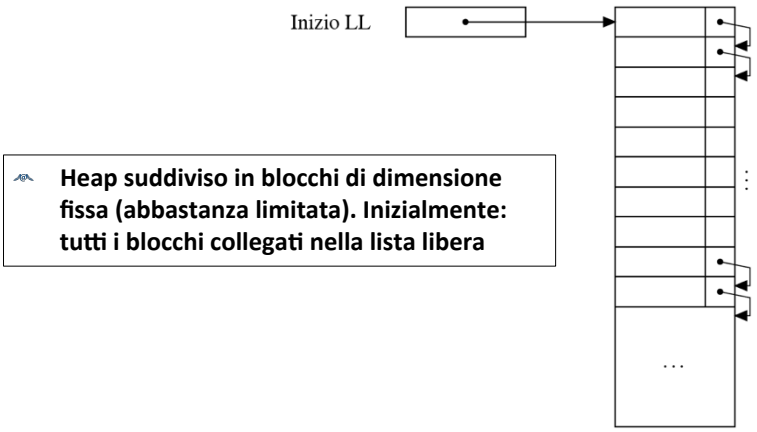
- **Heap**: regione di memoria i cui blocchi di memoria possono essere allocati e deallocati in momenti arbitrari

- Necessario quando il linguaggio permette
  - allocazione esplicita di memoria a run-time
  - oggetti di dimensioni variabili
  - oggetti con vita non LIFO

- La gestione dello heap non è banale
  - gestione efficiente dello spazio: frammentazione
  - velocità di accesso

6

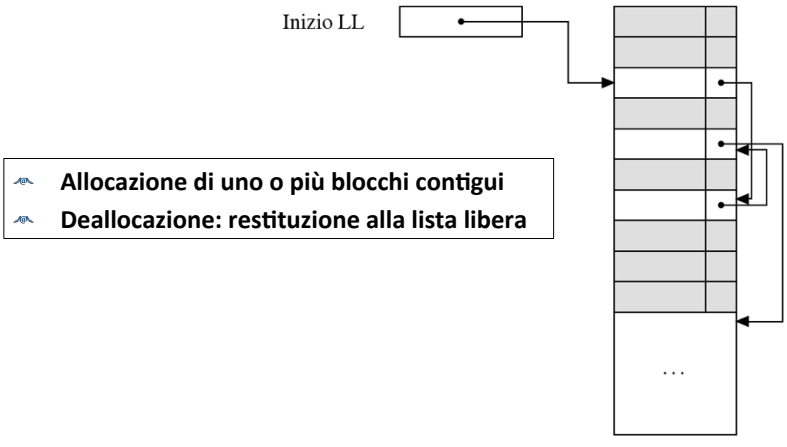
Heap: blocchi di dimensione fissa



- **Heap suddiviso in blocchi di dimensione fissa (abbastanza limitata). Inizialmente: tutti i blocchi collegati nella lista libera**

7

Heap: blocchi di dimensione fissa



- **Allocazione di uno o più blocchi contigui**
- **Deallocazione: restituzione alla lista libera**

8

Heap: blocchi di dimensione variabile

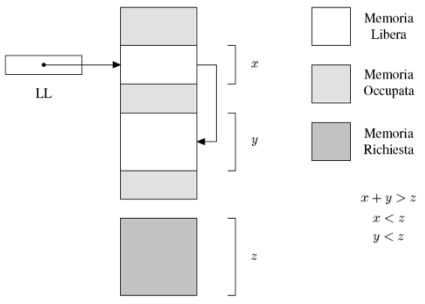
- Inizialmente **unico blocco** nello heap
- **Allocazione**: determinazione di un blocco libero della dimensione opportuna
- **Deallocazione**: restituzione alla lista libera

- **Problemi**:
  - le operazioni devono essere efficienti
  - evitare lo spreco di memoria
    - ✓frammentazione interna
    - ✓frammentazione esterna

9

Frammentazione

- Frammentazione **interna**
  - lo spazio richiesto è X
  - viene allocato un blocco di dimensione Y > X
  - lo spazio Y-X è sprecato
- Frammentazione **esterna**
  - ci sarebbe lo spazio necessario ma è inusabile perché suddiviso in "pezzi" troppo piccoli



10

Gestione della lista libera

- Inizialmente un solo blocco, della dimensione dello heap
- Ad ogni richiesta di allocazione cerca blocco di dimensione opportuna
  - **first fit**: primo blocco grande abbastanza
  - **best fit**: quello di dimensione più piccola, grande abbastanza
- Se il blocco scelto è molto più grande di quello che serve viene diviso in due e la parte inutilizzata è aggiunta alla LL
- Quando un blocco è de-allocato, viene restituito alla LL (se un blocco adiacente è libero i due blocchi sono ``fusi'' in un unico blocco)

11

Gestione heap

- First fit o Best fit? Solita situazione conflittuale…
  - First fit: più veloce, occupazione memoria peggiore
  - Best fit: più lento, occupazione memoria migliore
- Con unica LL costo allocazione lineare nel numero di blocchi liberi
- Per migliorare liste libere multiple: la ripartizione dei blocchi fra le varie liste può essere
  - ✓statica
  - ✓dinamica

12

Problema: identificazione dei blocchi da de-allocare

- Nella LL vanno reinseriti i blocchi da de-allocare
- Come vengono individuati?
  - linguaggi con de-allocazione esplicita (tipo **free**): se **p** punta a struttura dati, **free p** de-alloca la memoria che contiene la struttura
  - linguaggi senza de-allocazione esplicita: una porzione di memoria è recuperabile se non è più raggiungibile "in nessun modo"
- Il primo meccanismo è più semplice, ma lascia la responsabilità al programmatore, e può causare errori (***dangling pointer***)
- Il secondo meccanismo richiede un opportuno modello della memoria per definire "raggiungibilità"

13

Gestione memoria

- I "moderni" linguaggi di programmazione assumono un modello di gestione automatica della memoria a heap
- Esempio (da OCAML)
  - let rec append x y = if x = [] then y else
    hd x :: append (tl x) y
    let rec rev l if l =[] then [] else
    append(rev (tl l)) @ [hd l]
  - Assumiamo che length(l) = 10, cosa succede quando rev(l) e' invocata?

•slide 14

Modello a grafo della memoria

- È necessario determinare il ***root set***, cioè l'insieme dei dati sicuramente "attivi"
  - **Java root set** = variabili statiche + variabile allocate sul run-time stack
- Per ogni struttura dati allocata (nello stack e nello heap) occorre sapere dove ci possono essere puntatori a elementi dello heap (informazione presente nei **type descriptor**)
- ***Reachable active data***: la chiusura transitiva del grafo a partire dalle radici, cioè tutti i dati raggiungibili anche indirettamente dal **root set** seguendo i puntatori

15

Celle, "liveness", blocchi e garbage

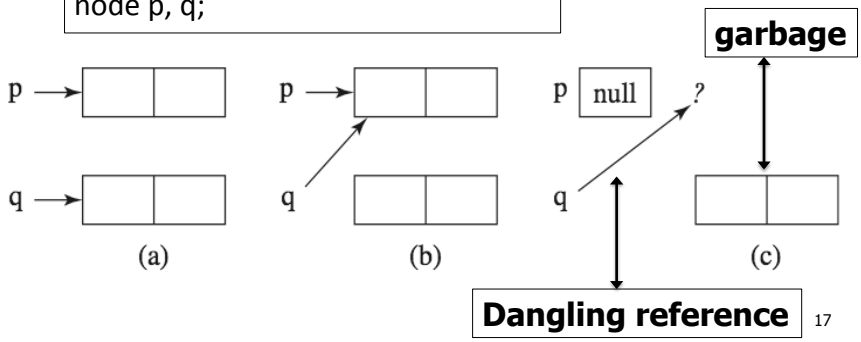
- Cella = blocco di memoria sullo heap
- Una cella viene detta live se il suo indirizzo è memorizzato in una radice o in una altra cella live
  - quindi: una cella è live se e solo se appartiene ai *Reachable active data*
- Una cella è garbage se non è live
- Garbage collection (GC): attività di gestione della memoria dinamica consistente nell'individuare le celle garbage (o "il garbage") e renderle riutilizzabili, per es. inserendole nella Lista Libera

16

Garbage e dangling reference


```
class node {
    int value;
    node next;
}
node p, q;
```

```
p = new node();
q = new node();
q = p;
free p;
```



17

Garbage collection
Perché è interessante?

- Applicazioni moderne sembrano non avere limiti allo spazio di memoria
  - 4GB laptop, 8GB desktop, 8-512GB server
  - spazio di indirizzi a 64-bit
- Ma l'uso scorretto fa emergere problemi come
  - memory leak, dangling reference, null pointer dereferencing, heap fragmentation
  - problemi di interazione con caching e paginazione
- **La gestione della memoria esplicita viola il principio dell'astrazione dei linguaggi di programmazione**

18

GC e astrazioni linguistiche

- GC non è una astrazione linguistica
- GC è una componente della macchina virtuale
  - VM di Lisp, Scheme, Prolog, Smalltalk …
  - VM di C and C++ non lo avevano ma librerie di garbage collection sono state introdotte per C/C++
- Sviluppi recenti del GC
  - linguaggi OO: Modula-3, Java, C#
  - linguaggi Funzionali: ML, Haskell, F#

19

Il garbage collector perfetto

- Nessun impatto visibile sull'esecuzione dei programmi
- Opera su ogni tipo di programma e su ogni tipo di struttura dati dinamica (esempio: strutture cicliche)
- Individua il garbage (e solamente il garbage) in modo efficiente e veloce
- Nessun overhead sulla gestione della memoria complessiva (caching e paginazione)
- Gestione heap efficiente (nessun problema di frammentazione)

20

Quali sono le tecniche di GC?

- **Reference counting – Contatori di riferimento**
  - gestione diretta delle celle live
  - la gestione è associata alla fase di allocazione della memoria dinamica
  - non ha bisogno di determinare la memoria garbage
- **Tracing**: identifica le celle che sono diventate garbage
  - **mark-sweep**
  - **copy collection**
- Tecnica up-to-date: **generational GC**

21

Reference counting

- Aggiungere un contatore di riferimenti alla celle (numero di cammini di accesso attivi verso la cella)
- Overhead di gestione
  - spazio per i contatori di riferimento
  - operazioni che modificano i puntatori richiedono incremento o decremento del valore del contatore.
  - gestione “real-time”
- Unix (file system) usa la tecnica dei reference count per la gestione dei file
- Java per la Remote Method Invocation (RMI)
- C++ “smart pointer”

22

Reference counting

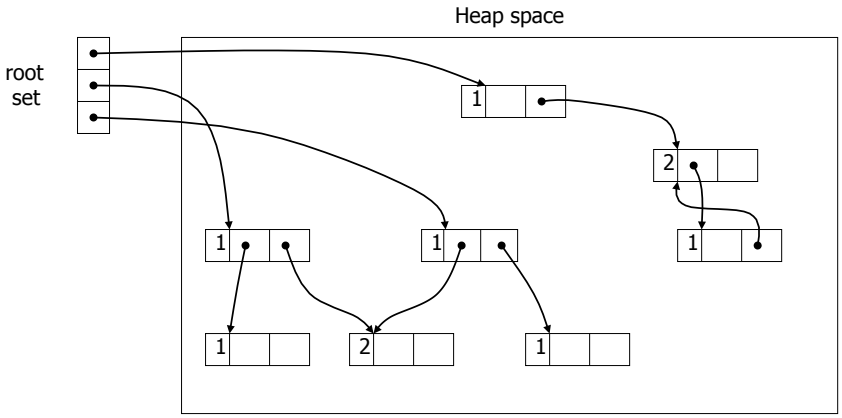
Integer i = new Integer(10);

- RC (i) = 1.

j = k; (j, k riferiscono a oggetti)

- RC(j) --.
- RC(k) ++.

Reference counting: esempio



Reference counting: caratteristiche

- Incrementale
  - la gestione della memoria è amalgamata direttamente con le operazioni delle primitive linguistiche
- Facile da implementare
- Coesiste con la gestione della memoria esplicita da programma (esempio malloc e free)
- Riuso delle celle libere immediato
  - if (RC == 0) then <restituire la cella alla lista libera>

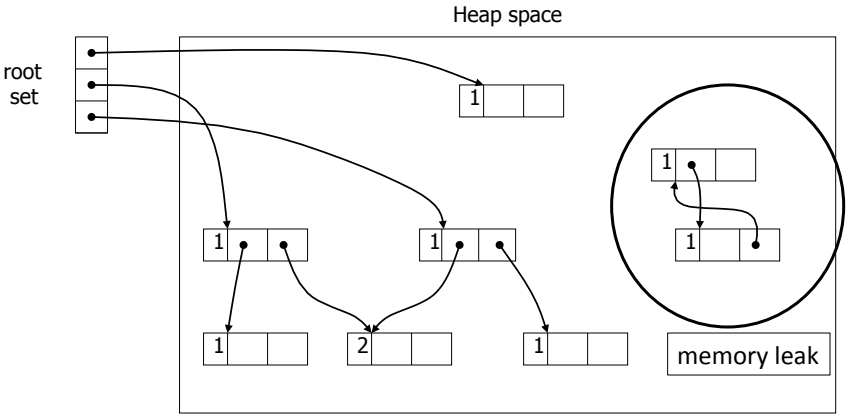
25

Reference counting: limitazioni

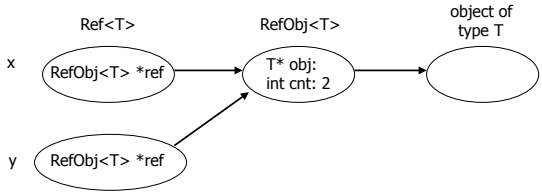
- Overhead spazio tempo
  - spazio per il contatore
  - la modifica di un puntatore richiede diverse operazioni
- Mancata esecuzione di una operazione sul valore di RC può generare garbage
- ***Non permette di gestire strutture dati con cicli interni***

26

Reference counting: cicli



"Smart pointer" (C++)



sizeof(RefObj<T>) = 8 byte per reference-counter dell'oggetto

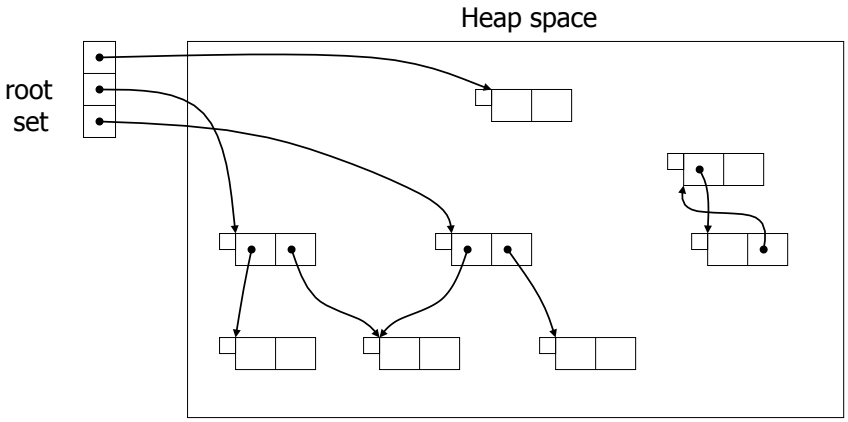
sizeof(Ref<T>) = 4 byte
- un normale puntatore

mark-sweep

- Ogni cella prevede spazio per un **bit di marcatura**
- Garbage può essere generato dal programma (non sono previsti interventi preventivi)
- L'attivazione del GC causa la sospensione del programma in esecuzione
- Marking
  - si parte dal **root set** e si marcano le celle **live**
- Sweep
  - tutte le celle non marcate sono garbage e sono restituite alla lista libera.
  - reset del bit di marcatura sulle celle live

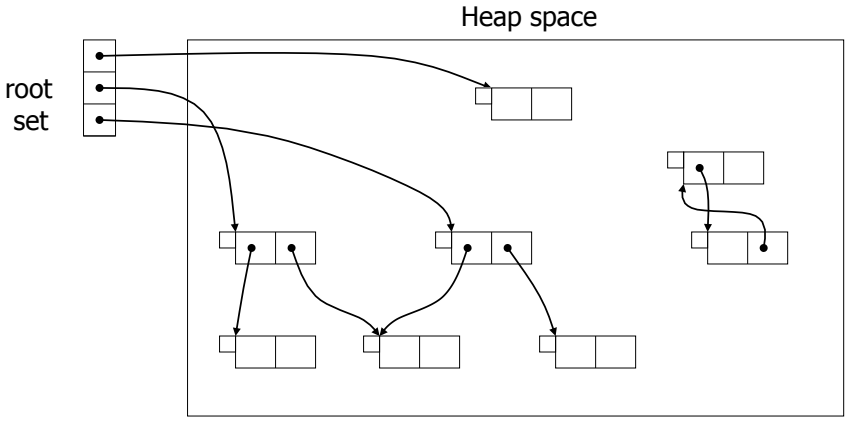
29

mark-sweep (1)



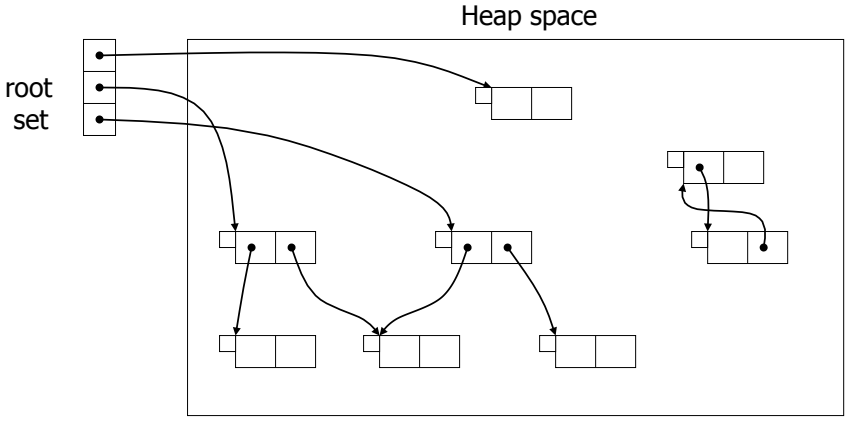
30

mark-sweep (2)



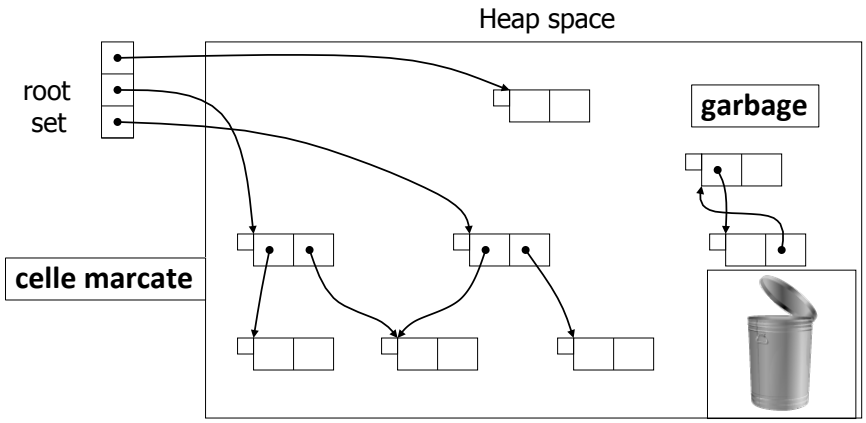
31

mark-sweep (3)



32

mark-sweep (4)



33

mark-sweep: valutazione

- Opera correttamente sulle structture circolari (+)
- Nessun overhead di spazio (+)
- Sospensione dell'esecuzione (-)
- Non interviene sulla frammentazione dello heap (-)

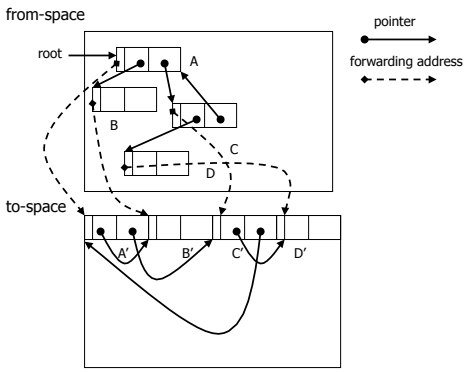
34

Copying collection

- L'**Algoritmo di Cheney** è un algoritmo di garbage collection che opera suddividendo la memoria heap in due parti
  - "**from-space**" e "**to-space**"
- Solamente una delle due parti dello heap è attiva (permette pertanto di allocare nuove celle)
- Quando viene attivato il garbage collector, le celle live vengono copiate nella seconda porzione dello heap (quella non attiva)
  - alla fine della operazione di copia i ruoli tra le due parti delle heap vengono scambiati (la parte non attiva diventa attiva e viceversa)
- Le celle nella parte non attiva vengono restituite alla lista libera in un unico blocco evitando problemi di frammentazione

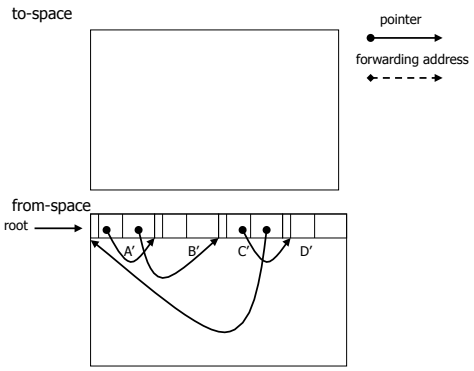
35

Esempio



36

Scambio dei ruoli



Copying collector: valutazione

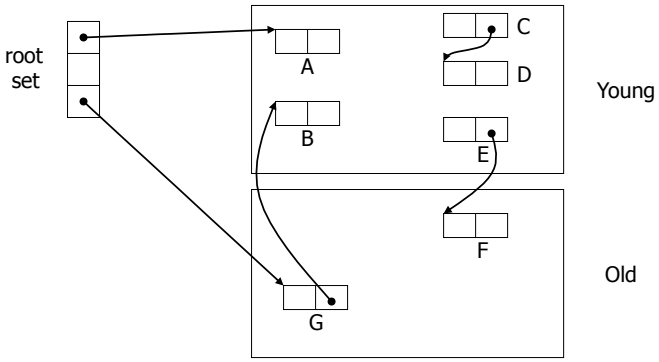
- Efficace nella allocazione di porzioni di spazio di dimensioni differenti e evita problemi di frammentazione
- Caratteristica negativa: duplicazione dello heap
  - dati sperimentali dicono che funziona molto bene su architetture hardware a 64-bit

Generational Garbage Collection

- Osservazione di base
  - "most cells that die, die young" (ad esempio a causa delle regole di scope dei blocchi)
- Si divide lo heap in un insieme di **generazioni**
- Il garbage collector opera sulle generazioni più giovani

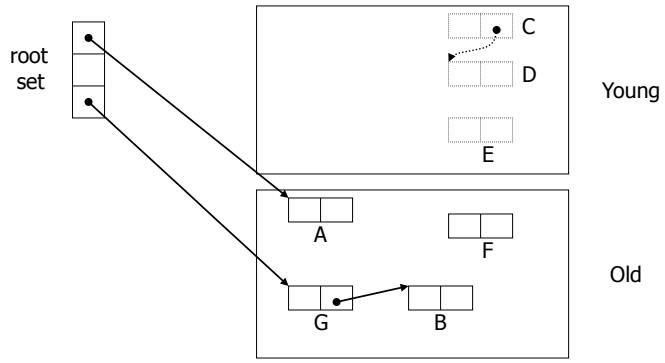
39

Esempio (1)



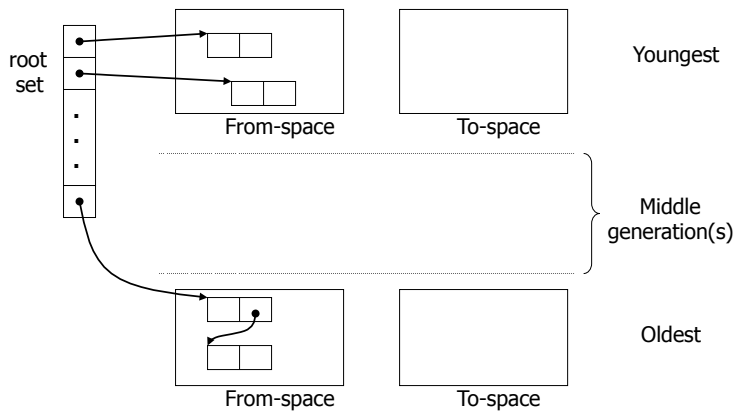
40

Esempio (2)



41

Copying + generazioni



42

GC nella pratica

- Sun/Oracle Hotspot JVM
  - GC con tre generazioni (0, 1, 2)
  - Gen. 1 copy collection
  - Gen. 2 mark-sweep con meccanismi per evitare la frammentazione
- Microsoft .NET
  – GC con tre generazioni (0, 1, 2)
  – Gen. 2 mark-sweep (non sempre compatta i blocchi sullo heap)

43