GARBAGE COLLECTION

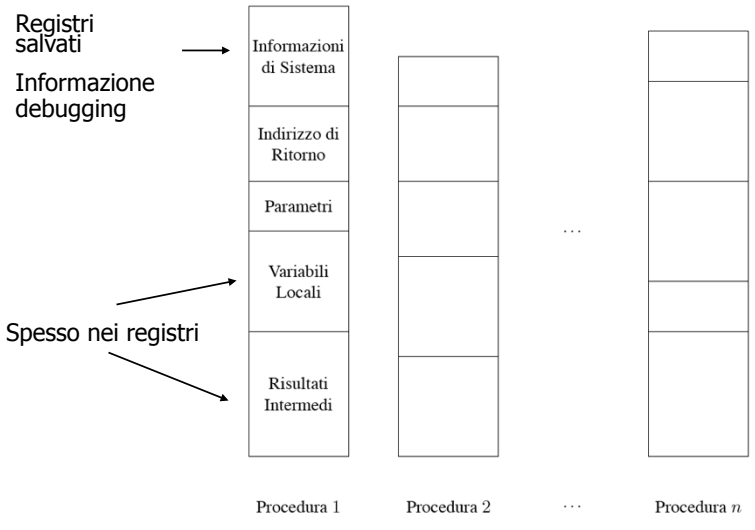
Gestione della memoria

- 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 zone protette di memoria

Allocazione statica per sottoprogrammi



Allocazione dinamica: pila

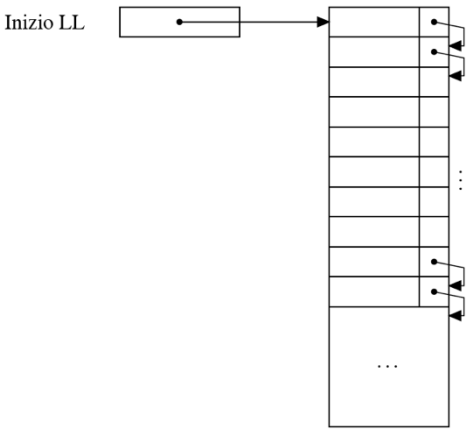
- Per ogni istanza di un sottoprogramma a run-time abbiamo un record di attivazione contente le informazioni relative a tale istanza
- Analogamente, ogni blocco ha un suo record di attivazione (più semplice)
- Anche in un linguaggio senza ricorsione può essere utile usare la pila per risparmiare memoria.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

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

Heap: blocchi di dimensione fissa

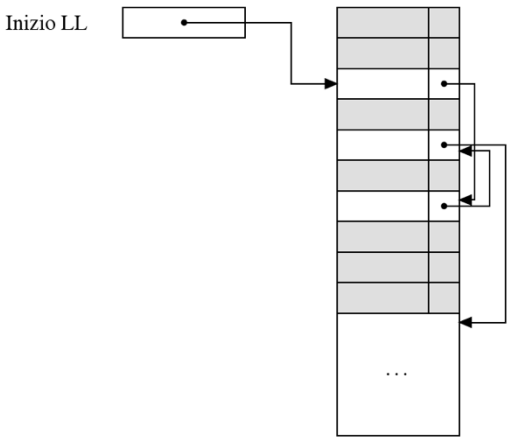
Heap suddiviso in blocchi di dimensione fissa (abbastanza limitata). Inizialmente : tutti i blocchi collegati nella lista libera



•slide 7

Heap: blocchi di dimensione fissa

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



•slide 8

Heap: blocchi di dimensione variabile

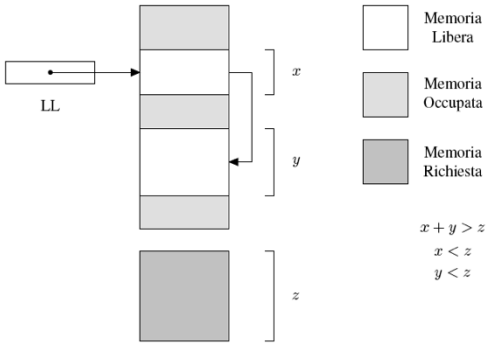
- Inizialmente **unico blocco** nello heap
- **Allocazione**: determinazione di un blocco libero della dimensione opportuna
- **Deallocazione**: restituzione alla lista libera

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

•slide 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



•slide 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).

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

Problema: identificazione di blocchi da de-allocare

- Nella LL vanno reinseriti i blocchi da de-allocare
- Come vengono individuati?
  - Linguaggi con de-allocazione esplicita (es: **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"
- Primo meccanismo più semplice, ma lascia la responsabilità al programmatore, e può causare errori (***dangling pointers***)
- Secondo meccanismo richiede un opportuno modello della memoria per definire "raggiungibilità"

•slide 13

Modello a grafo della memoria

- E' necessario determinare il ***root set***, cioè l'insieme dei dati sicuramente "attivi"
  - Es: **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 descriptors**)
- Def: ***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.

•slide 14

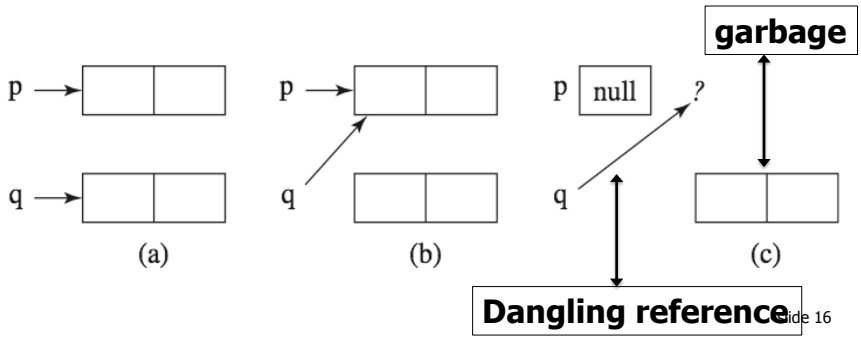
Celle, “liveness”, blocchi e garbage

- 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.
  - 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, p.es. inserendole nella Lista Libera

Garbage e Dangling References

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

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



Garbage Collection Perché è interessante?

- Applicazioni moderne sembra che non abbiamo limiti allo spazio di memoria
  - 4GB laptop, 8GB deskop, 8-512GB server
  - Spazio di indirizzi a 64-bit
- Ma uso scorretto fa emergere problemi come
  - Memory leak, dangling reference, null pointer dereferencing, heap fragmentation
  - Problemi di caching
- **La gestione della memoria esplicita viola il principio dell'astrazione dei linguaggi di programmazione**

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#

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)

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
  - DoNon deve determinare la memoria garbage
- **Tracing**: identifica le celle che sono diventate garbage
  - **Mark-sweep**
  - **Copy collection**
- Tecnica up-to-date: **generational GC**

Reference Counting

- Aggiungere un contatori 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 puntator: increemento 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”

•slide 21

Reference counting

Integer i = new Integer(10);

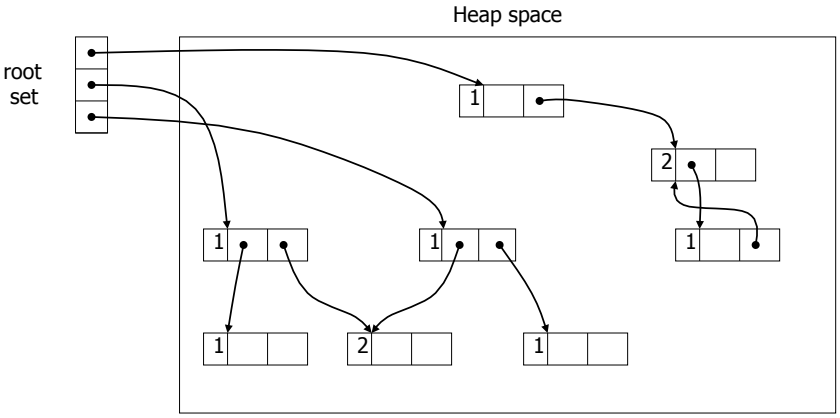
- RC (i) = 1.

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

- RC(i) --.
- RC(k) ++.

•slide 22

Reference Counting: Esempio



•slide 23

Reference Counting: Caratteristiche

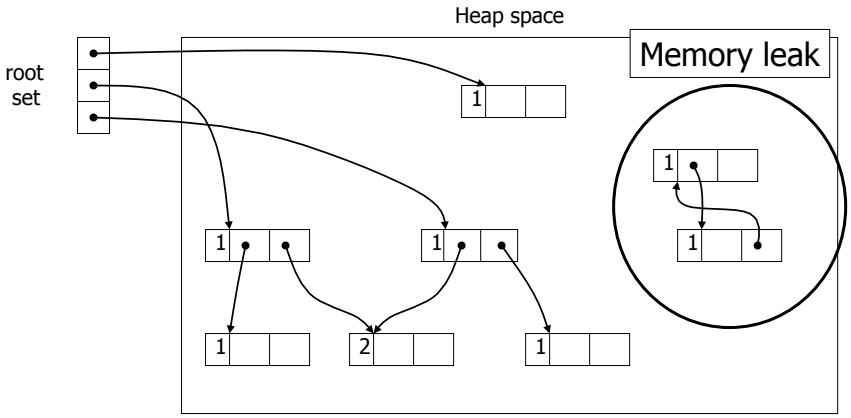
- Incrementale
  - La gestione della memoria è amalgamata direttamente con le operazioni delle primitive linguistiche
- Facile da implementare
- Coesiste con la gestione della memoria esplicita da programma (esempio malloc e free)
- Ri-uso delle celle libere immediato
  - If RC == 0 the <restituire la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

•slide 24

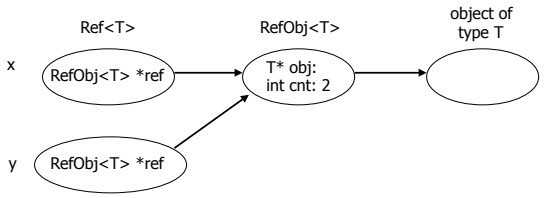
Reference Counting: Limitazioni

- 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***

Reference Counting: Cycles



“Smart Pointer” (C++)



sizeof(RefObj<T>) = 8 bytes per reference-counter dell’oggetto

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

Smart Pointer: Implementazione

```
template<class T> class Ref {
    RefObj<T>* ref;
    Ref<T>* operator&() {}
public:
    Ref() : ref(0) {}
    Ref(T* p) : ref(new RefObj<T>(p)) { ref->inc();}
    Ref(const Ref<T>& r) : ref(r.ref) { ref->inc(); }
    ~Ref() { if (ref->dec() == 0) delete ref; }

    Ref<T>& operator=(const Ref<T>& that) {
      if (this != &that) {
        if (ref->dec() == 0) delete ref;
          ref = that.ref;
          ref->inc(); }
      return *this; }
    T* operator->() { return *ref; }
    T& operator*() { return *ref; }
};
```

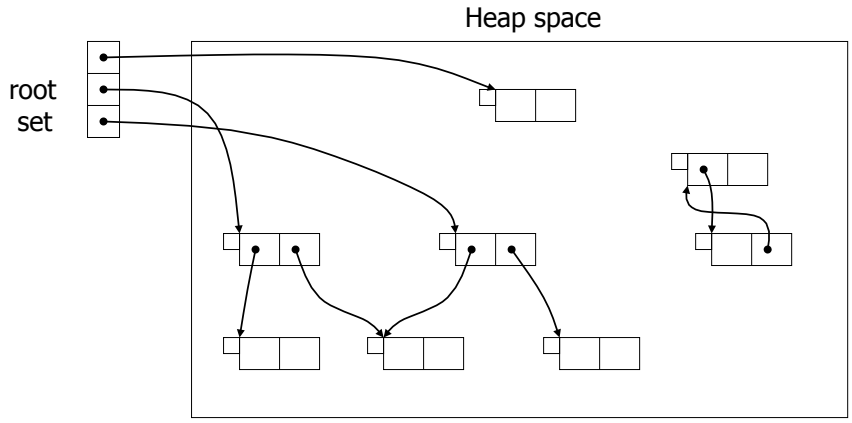
```
template<class T> class RefObj {
      T* obj;
      int cnt;
public:
      RefObj(T* t) : obj(t), cnt(0) {}
      ~RefObj() { delete obj; }

      int inc() { return ++cnt; }
      int dec() { return --cnt; }

      operator T*() { return obj; }
      operator T&() { return *obj; }
      T& operator *() { return *obj; }
};
```

Mark-Sweep

- Ogni cella prevede spazio per un **bit di marcatura**.
- Garbage può essere generato dal programma (non sono previsti interventi preventivi)
- Attivazione del GC causa la sospensione del programma in esecuzione.
- Marking
  - Si parte dalle radice 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.

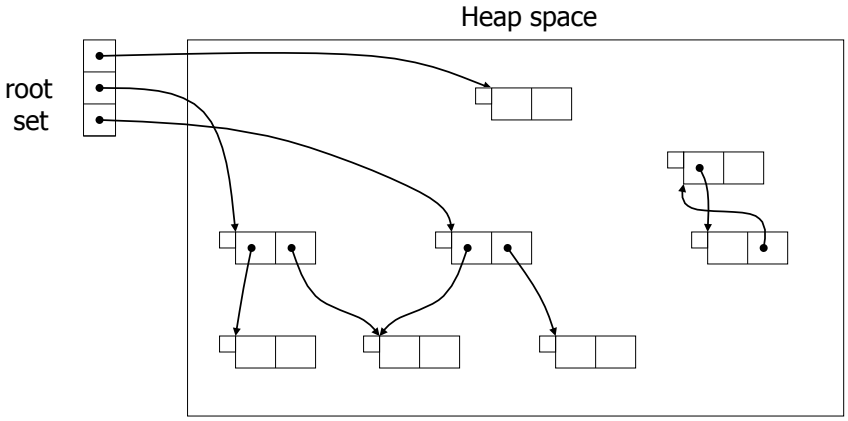
•slide 29

Mark-Sweep (1)



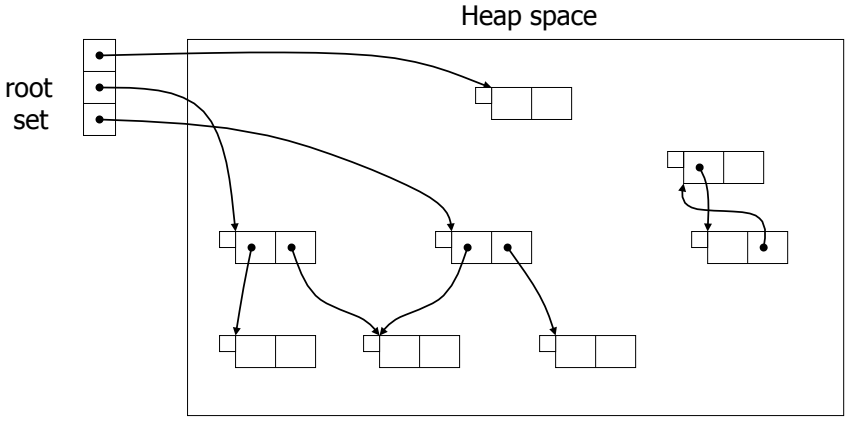
•slide 30

Mark-Sweep (2)



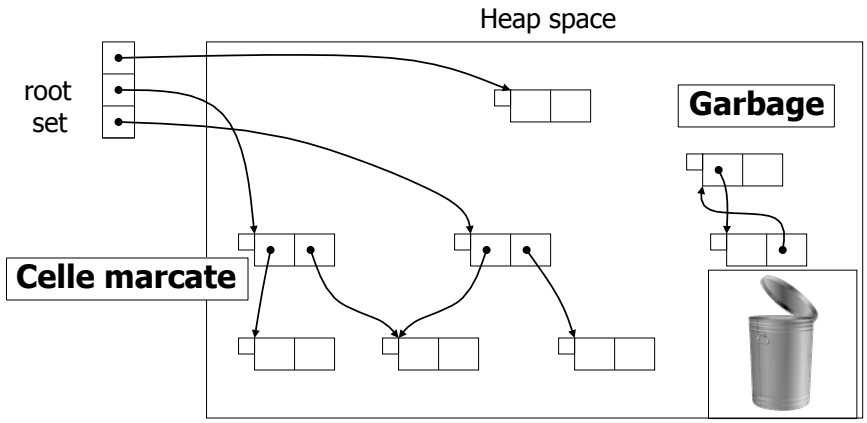
•slide 31

Mark-Sweep (3)



•slide 32

Mark-Sweep (4)



Mark-Sweep: valutazione

- Opera correttamente sulle structture circolari (+)
- Nessun overhead di spazio (+)
- Sospenzione dell'esecusione (-)
- Puo' causare frammentazione dello heap (-)

Copying collection

- 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 collection 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.

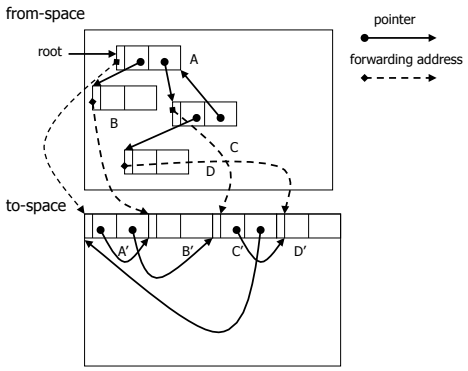
•slide 35

Copying Collector

- Heap viene strutturato in due parti
  - "**from-space**" e "**to-space**"
- Le celle nella parte *from-space* sono analizzate dalla marcatura e le celle live sono copiate nella parte *to-space*
  - Problema: si deve mantanere la struttura a lista libera dello hep.
  - "Forward the referred object to to-space if it has not already been forwarded. This means allocating an identical object in to-space and copying the object data over.."

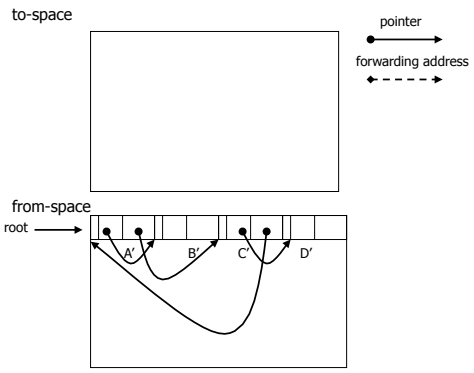
•slide 36

Esempio



•slide 37

Scambio dei ruoli



•slide 38

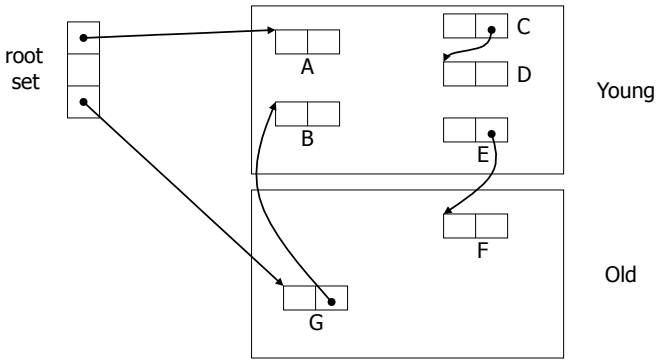
Copying Collector: valutazione

- 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

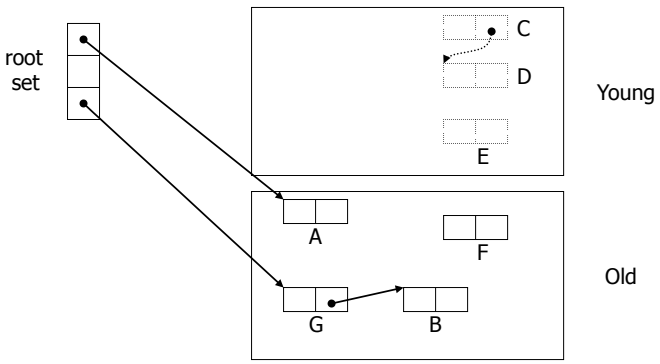
- Osservazione di base:
  - "most cells that die, die young" (ad esempio a causa delle regole di scope dei blocchi)
- Si divide lo heap in un insieme di **generazioni**
- Il garbage collector opera sulle generazioni più giovani

Esempio (1)



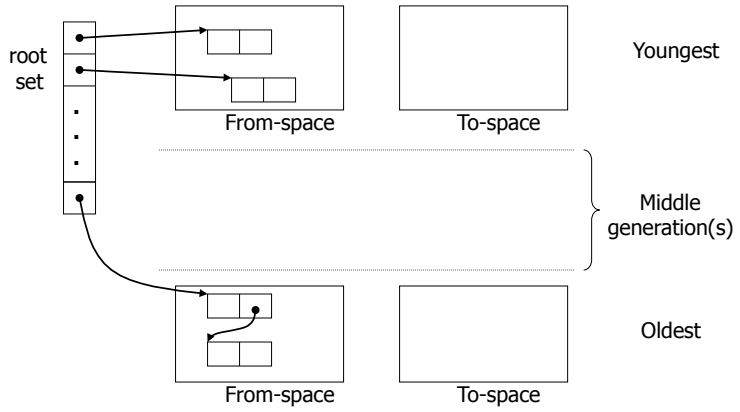
•slide 41

Esempio (2)



•slide 42

Copying + generazioni



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)