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 sotto-programmi (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

Allocazione dinamica: stack

- Per ogni istanza di una funzione/procedura 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)
- Lo stack è necessario nei linguaggi con ricorsione (più istanze dello stesso record di attivazione attive contemporaneamente)
- Lo stack potrebbe non essere usato nei linguaggi senza ricorsione (es. Fortran): i record di attivazione potrebbero essere tutti allocati staticamente
- Anche in un linguaggio senza ricorsione può essere utile usare la pila per risparmiare memoria...

Allocazione dinamica con heap

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

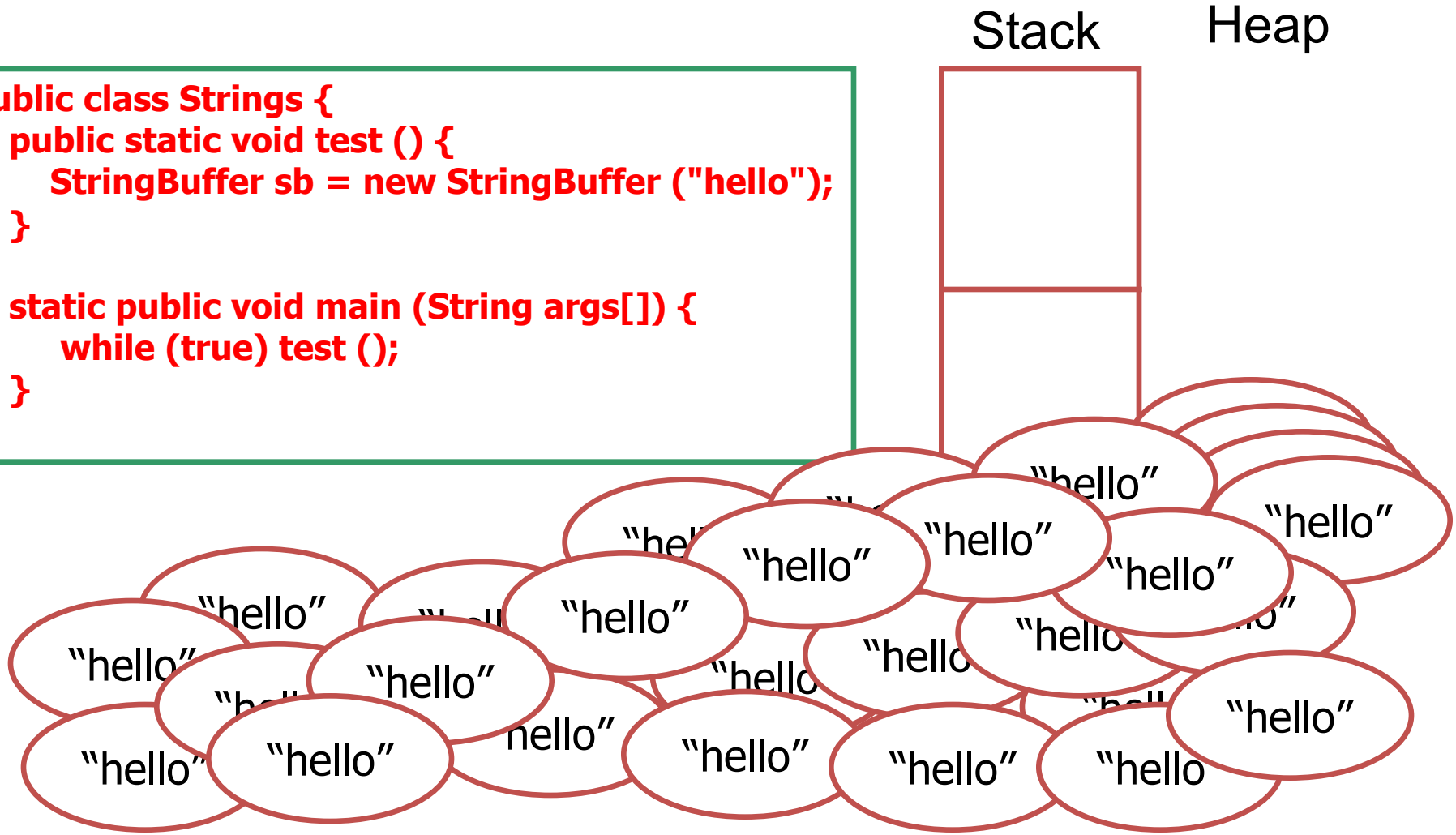
- Necessario quando il linguaggio permette
  - allocazione esplicita di memoria a run-time (es. malloc)
  - oggetti di dimensioni variabili e con un ciclo di vita

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

Garbage Heap (Java)

```
public class Strings {
   public static void test () {
      StringBuffer sb = new StringBuffer ("hello");
   }

   static public void main (String args[]) {
       while (true) test ();
   }
}
```



Gestione memoria heap

Esempio (**OCAML**)

```
let rec rev lis = match lis with
                  | [] -> []
                  | x::lis' -> (rev lis')@[x]
```

Assumiamo che **lis** abbia lunghezza 10, che cosa succede in memoria quando **rev lis** è invocata?

Gestione memoria heap

Esempio (**OCAML**)

```
let rec rev lis = match lis with
                  | [] -> []
                  | x::lis' -> (rev lis')@[x]
```

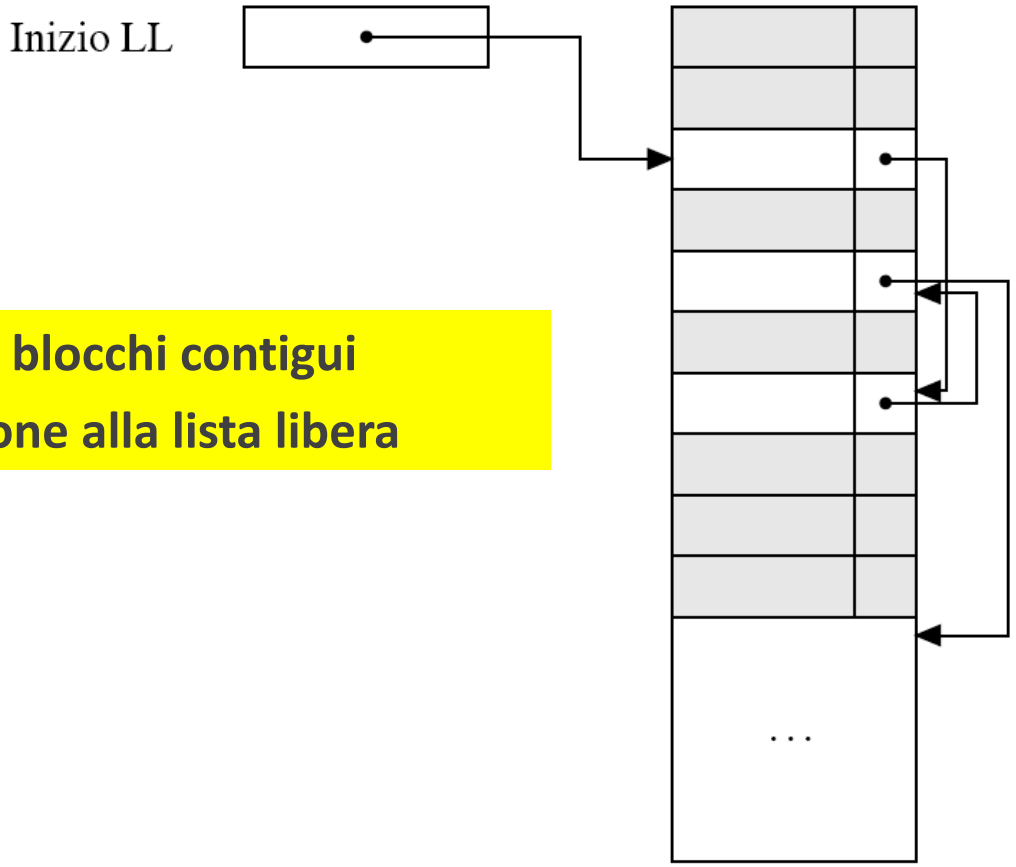
Assumiamo che **lis** abbia lunghezza 10, che cosa succede in memoria quando **rev lis** è invocata?

- Dato che @ fa una copia del proprio primo operando, ad ogni chiamata ricorsiva in memoria viene allocata una nuova lista temporanea

Heap con blocchi di dimensione fissa

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

Heap con blocchi di dimensione fissa



Allocazione di uno o più blocchi contigui

Deallocazione: restituzione alla lista libera

…o blocchi dimensione variabile

- 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)

GC: perché è interessante?

- Applicazioni moderne sembrano non avere limiti allo spazio di memoria
  - esempio: 16GB sui PC, 512GB sui server
  - spazio di indirizzi a 64-bit
- Ma l'uso scorretto fa emergere problemi come
  - memory leak (mancata deallocazione), dangling reference, null pointer dereferencing, heap fragmentation

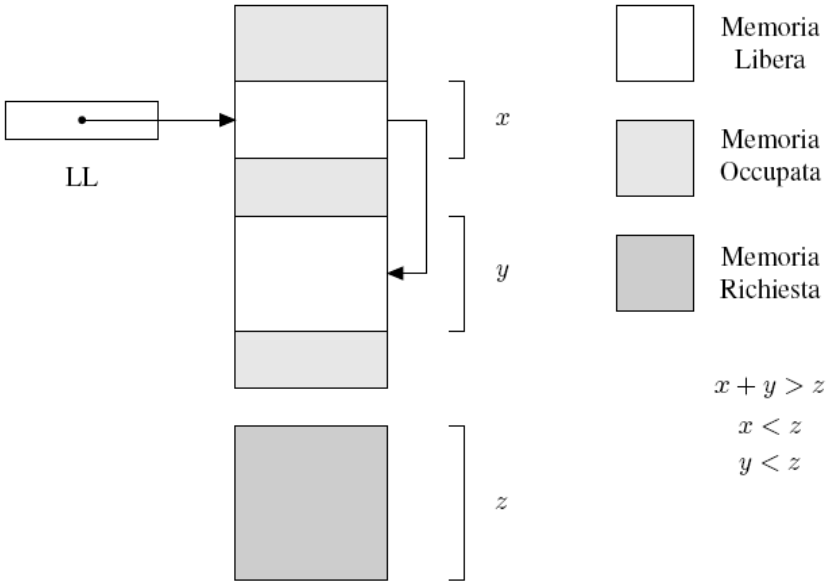
La gestione della memoria esplicita (es. malloc) viola il principio d'astrazione dei linguaggi di programmazione (porta a occuparsi di cosa accade a un livello più basso)

GC e astrazioni linguistiche

- GC non è una astrazione linguistica (non fa parte del linguaggio)
- 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: Java, C#
  - linguaggi Funzionali: Haskell, F#

Frammentazione

- Frammentazione **interna**
  - lo spazio richiesto è X
  - vengono allocati uno o più blocchi 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



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

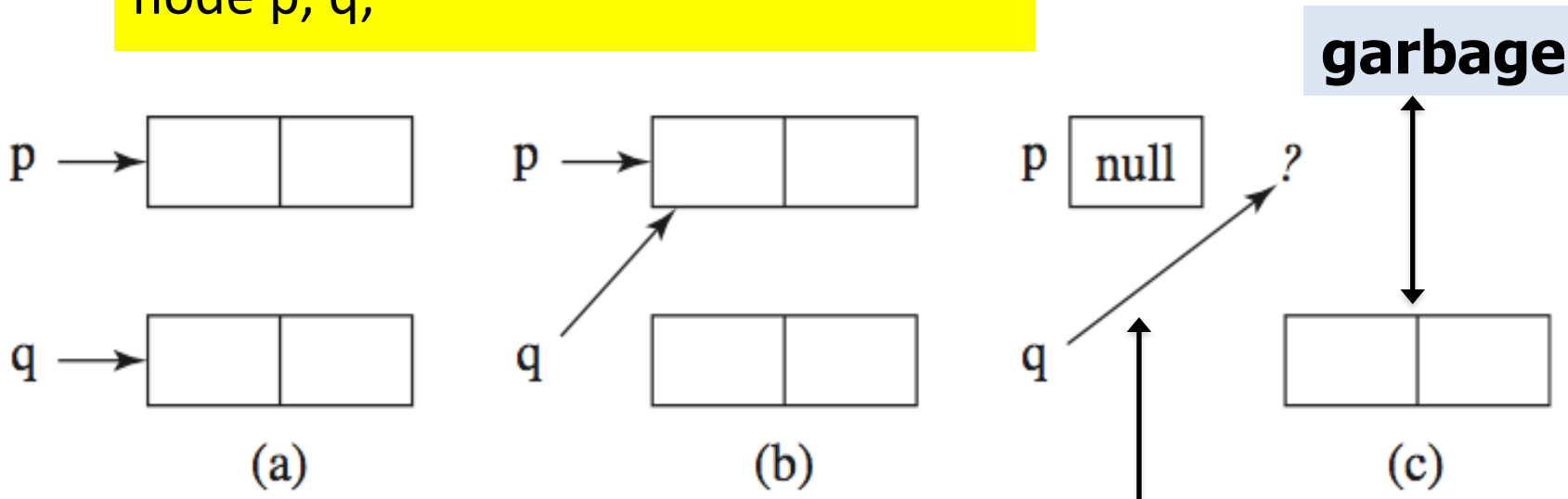
Problema: identificazione dei blocchi da de-allocare

- 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"

Garbage e dangling reference

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

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



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 riferimenti**
  - 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 allo stato dell'arte: **generational GC**

Reference counting

- Aggiungere un contatore dei riferimenti ai blocchi allocati (numero di cammini di accesso attivi verso ogni blocco)
- Overhead di gestione
  - spazio per i contatori di riferimenti
  - operazioni che modificano i puntatori richiedono incremento o decremento del valore del contatore.
  - gestione in “real-time" (continua, durante l'esecuzione)

Reference Counting

- Unix (file system) usa la tecnica dei reference count per la gestione dei file
  - un file non viene veramente cancellato fintanto che ci sono hard link verso di esso
- C++ “smart pointer” (shared_ptr)
  - allocazione e creazione di alias

```
// dichiara e inizializza uno smart pointer (shared)
// implicitamente alloca anche un contatore (reference counter)
// inizializzato a 1
shared_ptr<Persona> sp1(new Persona("Mario", "Rossi"));

// crea un alias (incrementa il reference counter)
shared_ptr<Persona> sp2 = sp1;

// il reference counter si decrementa ogni volta che un alias
// va fuori scope (es. chiusura del blocco).
// quando raggiunge 0 avviene automaticamente la free
```

Reference counting su oggetti

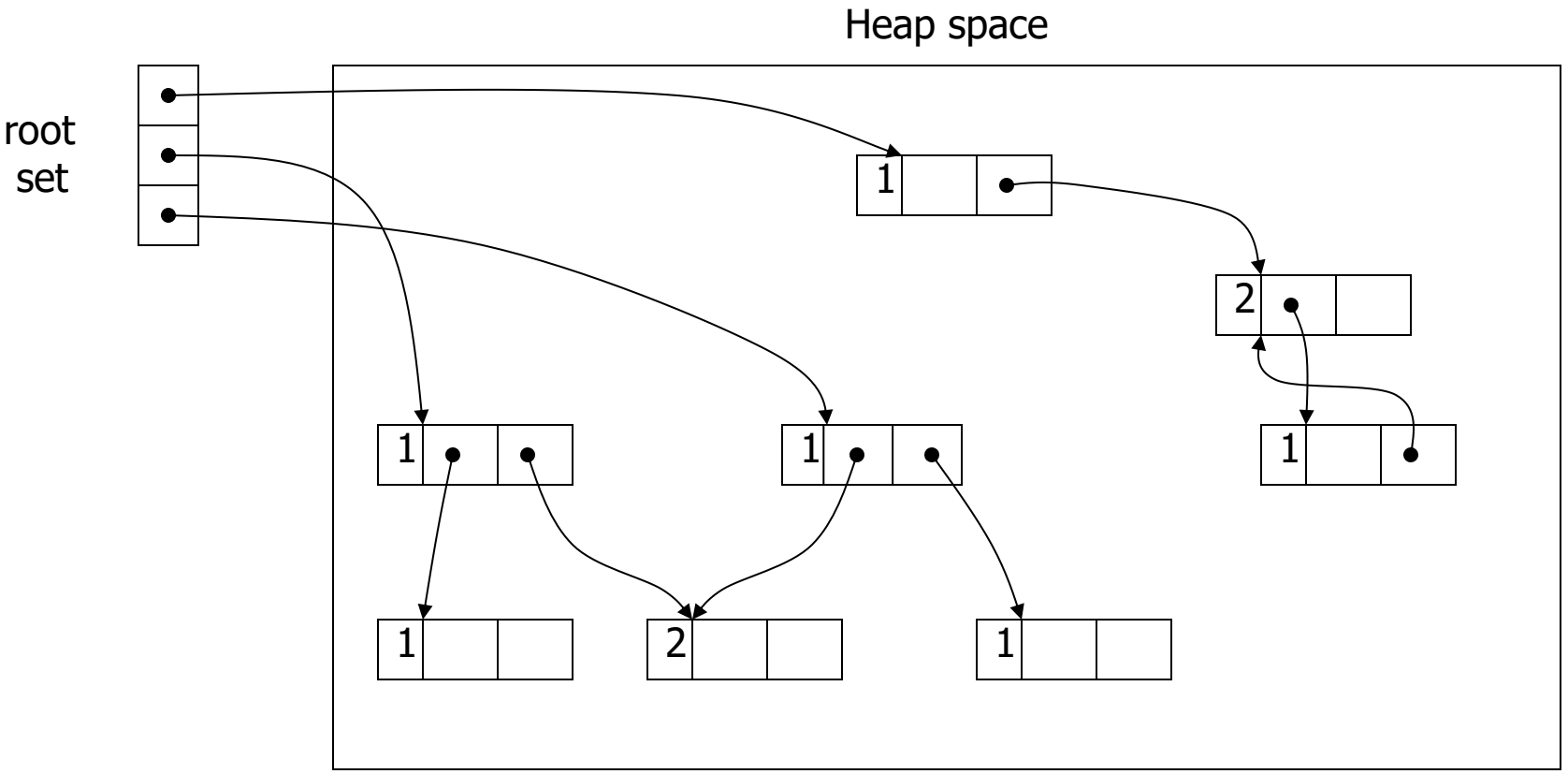
Persona p =
new Persona("Mario","Rossi");

- **RC (p) = 1.**

j = k;
//j, k si riferiscono a oggetti

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

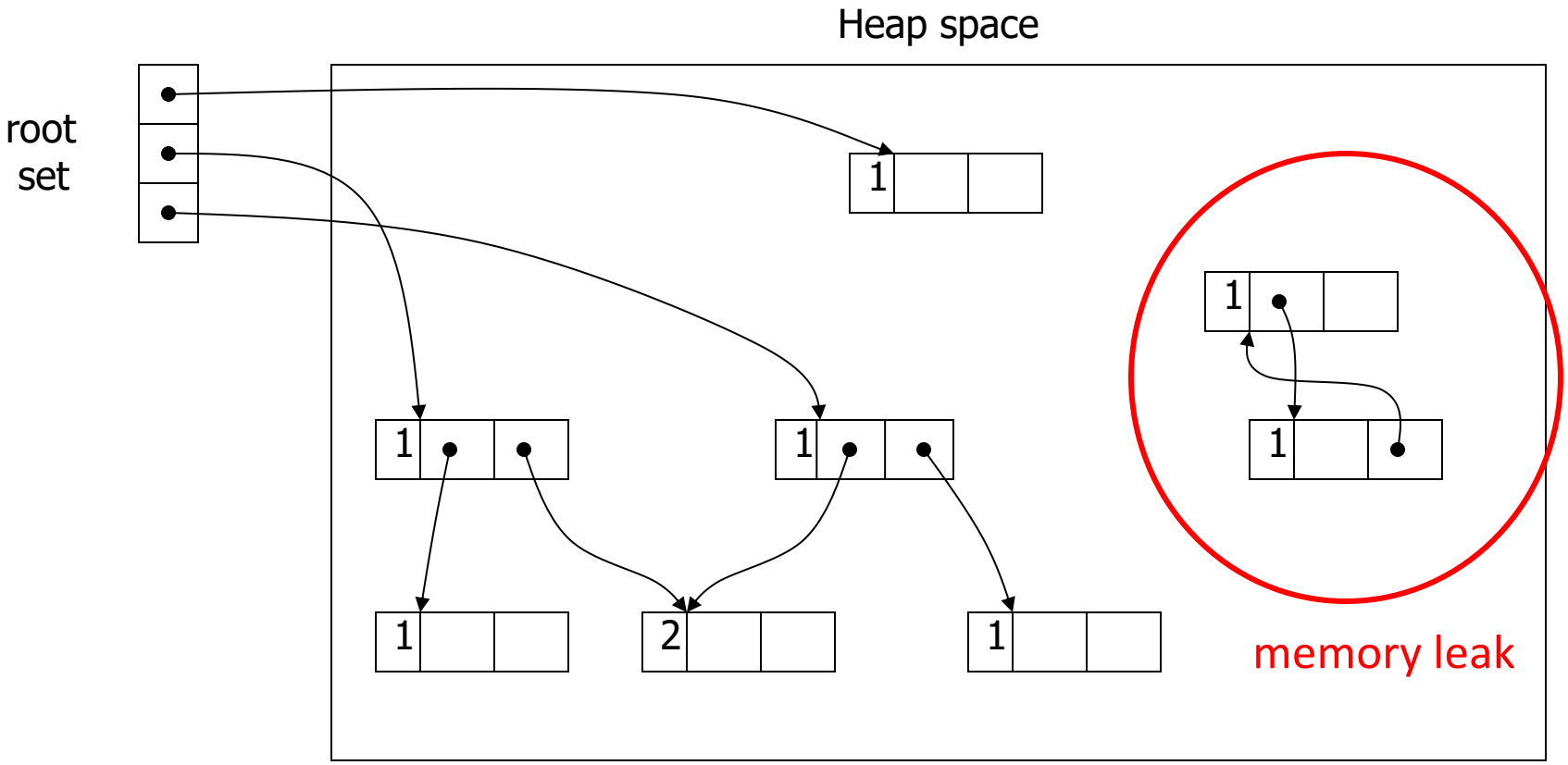
Reference counting: esempio



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>

Reference counting: cicli



Reference counting: limitazioni

Overhead spazio tempo

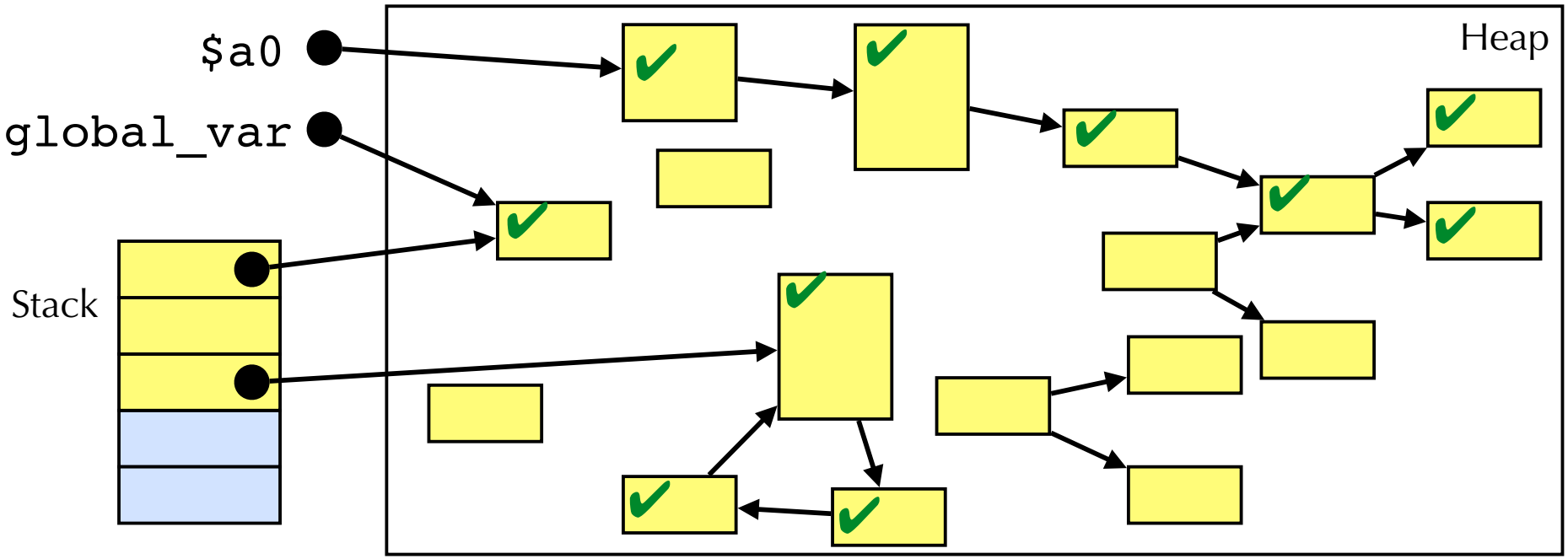
- spazio per il contatore
- la modifica di un puntatore richiede diverse operazioni

Non permette di gestire strutture dati con cicli interni

Modello a grafo della memoria

- È necessario determinare il ***root set***, l'insieme dei dati "attivi" **(variabili statiche + variabili 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 dal **root set** seguendo i puntatori

Raggiungibilità



Celle, “liveness”, blocchi e garbage

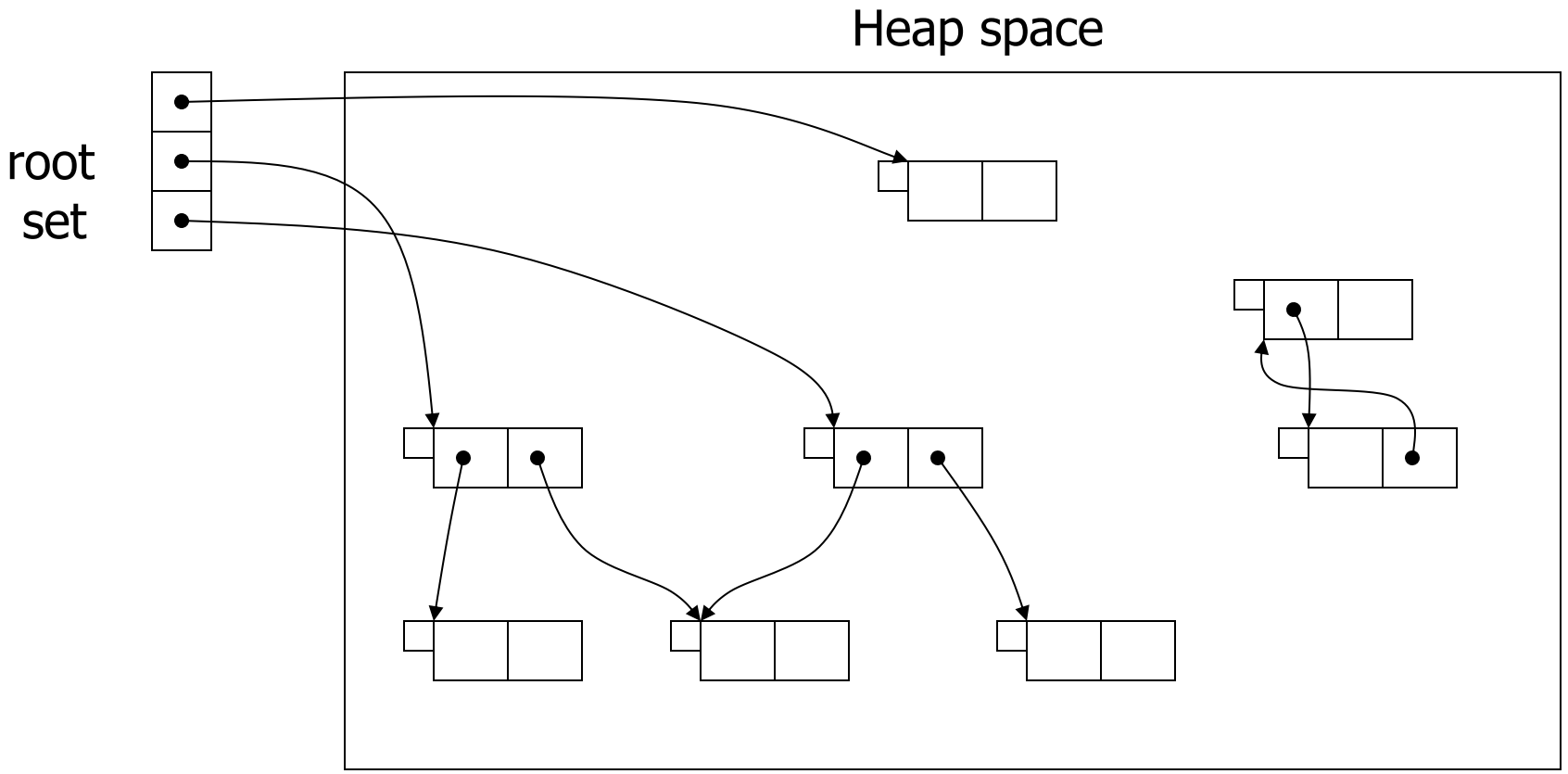
- **Cella** = blocco di memoria sullo heap
- Una cella viene detta **live** se il suo indirizzo è memorizzato in una radice o in una altra cella live
  - quindi: una cella è live se e solo se appartiene ai *Reachable active data*
- Una cella è garbage se non è live

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

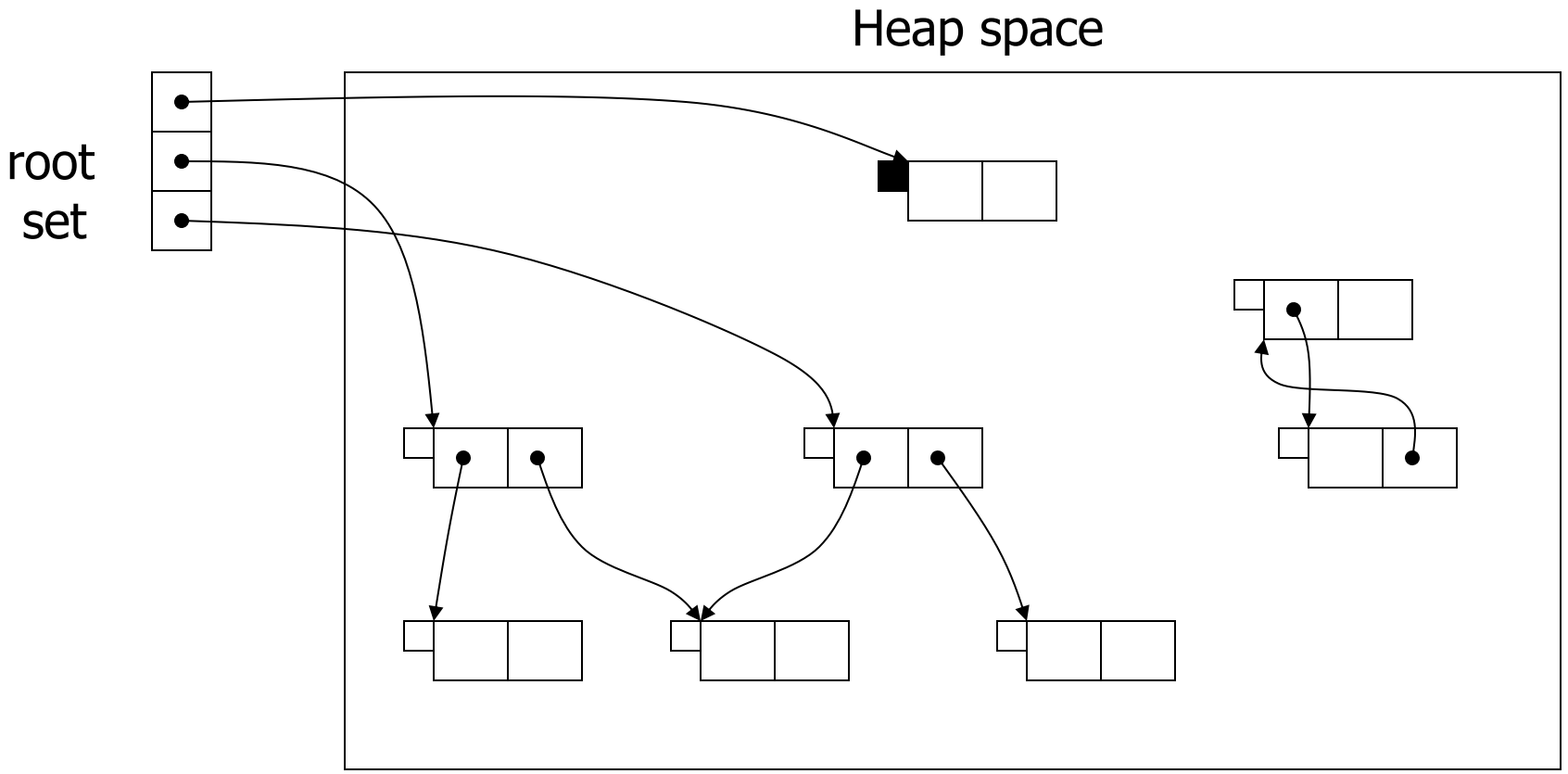
mark-sweep

- 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

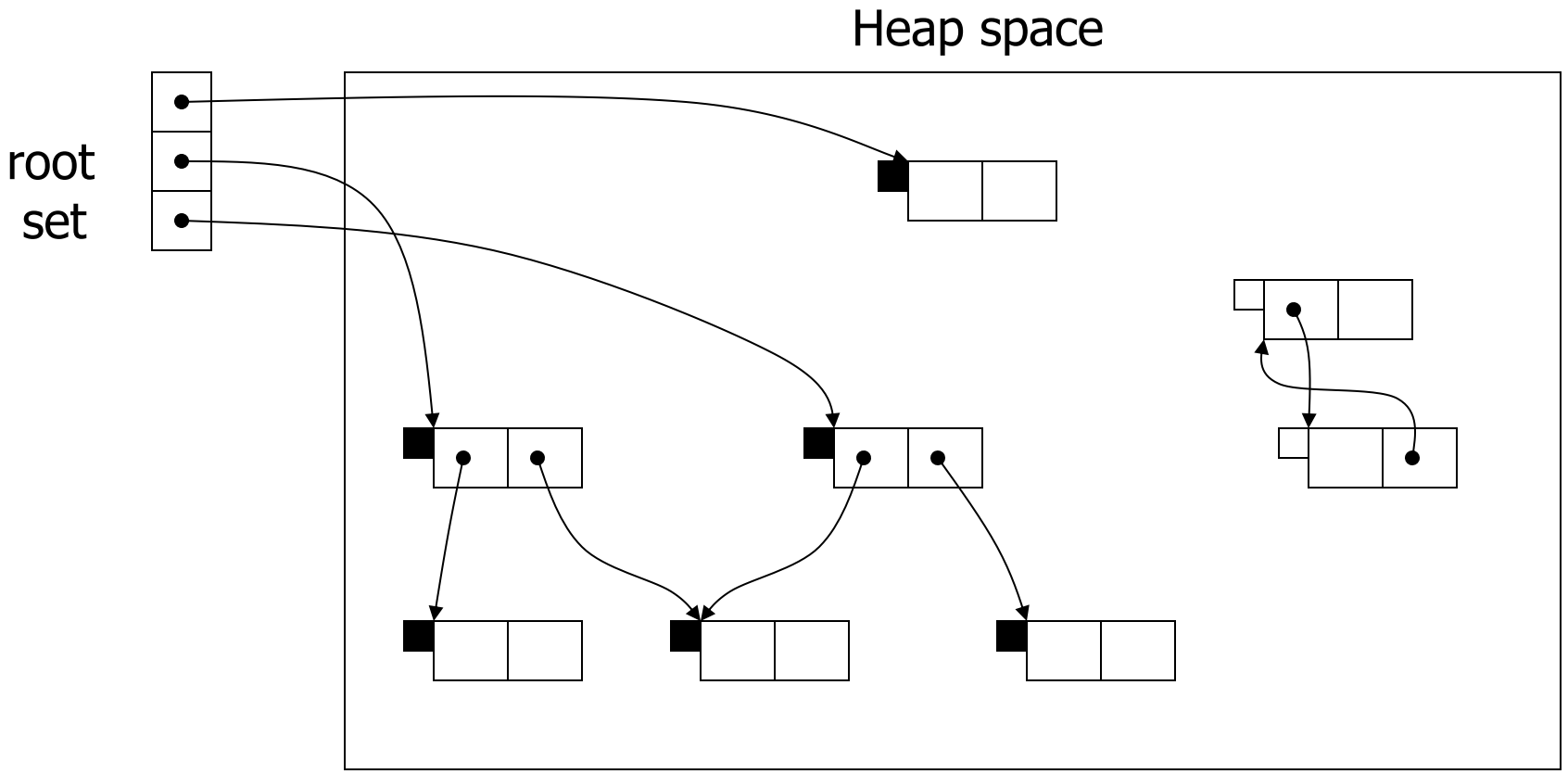
mark-sweep (1)



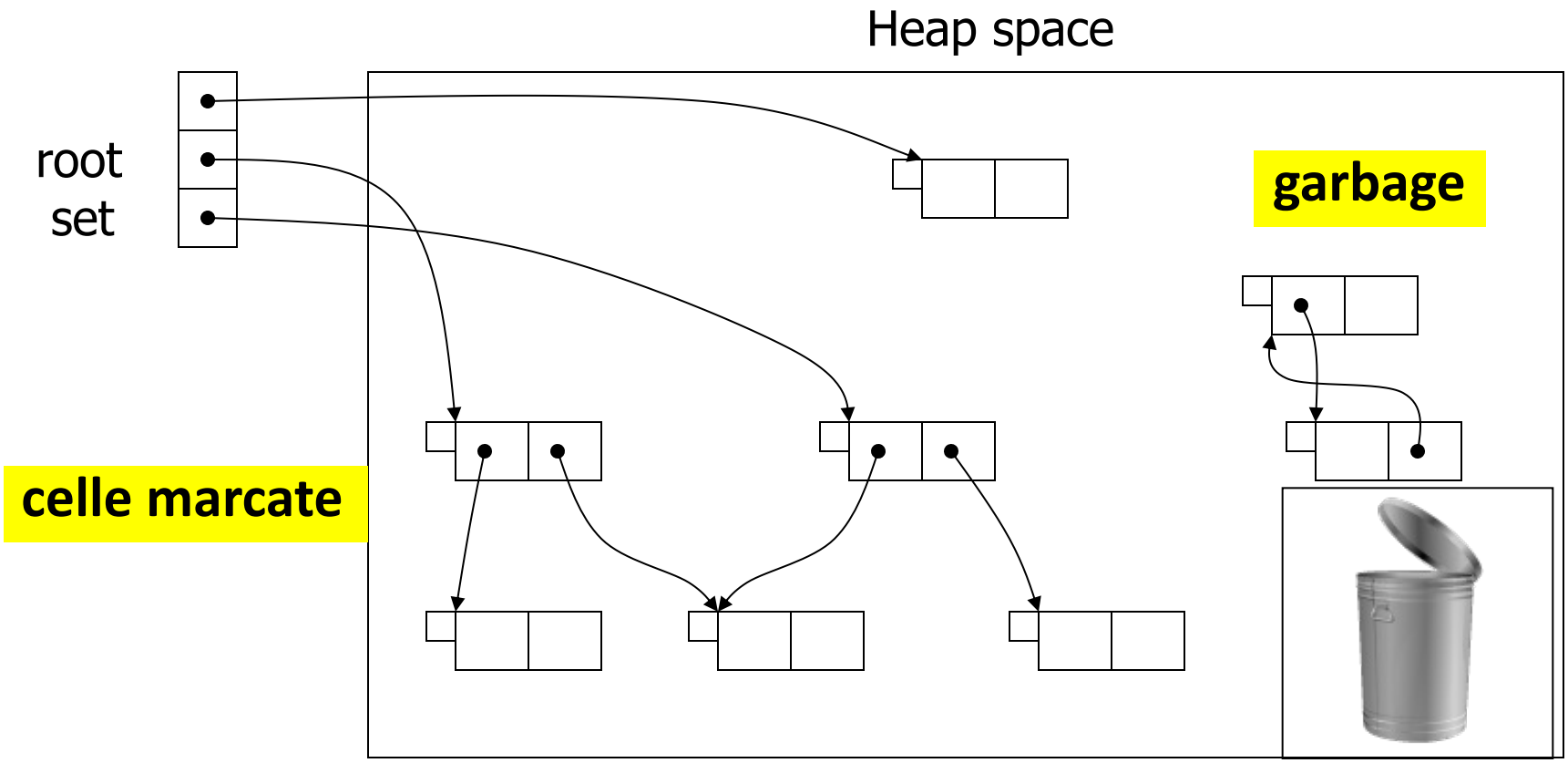
mark-sweep (2)



mark-sweep (3)



mark-sweep (4)



Mark & Sweep

MARK

```
For each root v:
    DFS(v)
```

```
function DFS(x):
    if x is a pointer into heap
        if record x is not marked
            mark x
            for each field f_i of record x
                DFS(x.f_i)
```

SWEEP

```
p ← first address in heap
while p < last address in heap
    if record p is marked
        unmark p
    else let f1 be the first field in p
        p.f1 ← freelist
        freelist ← p
    p ← p + (size of record p)
```

mark-sweep: valutazione

Opera correttamente sulle strutture circolari (+)

Nessun overhead di spazio (+)

Sospensione dell'esecuzione (-)

Non interviene sulla frammentazione dello heap (-)

Copying collection

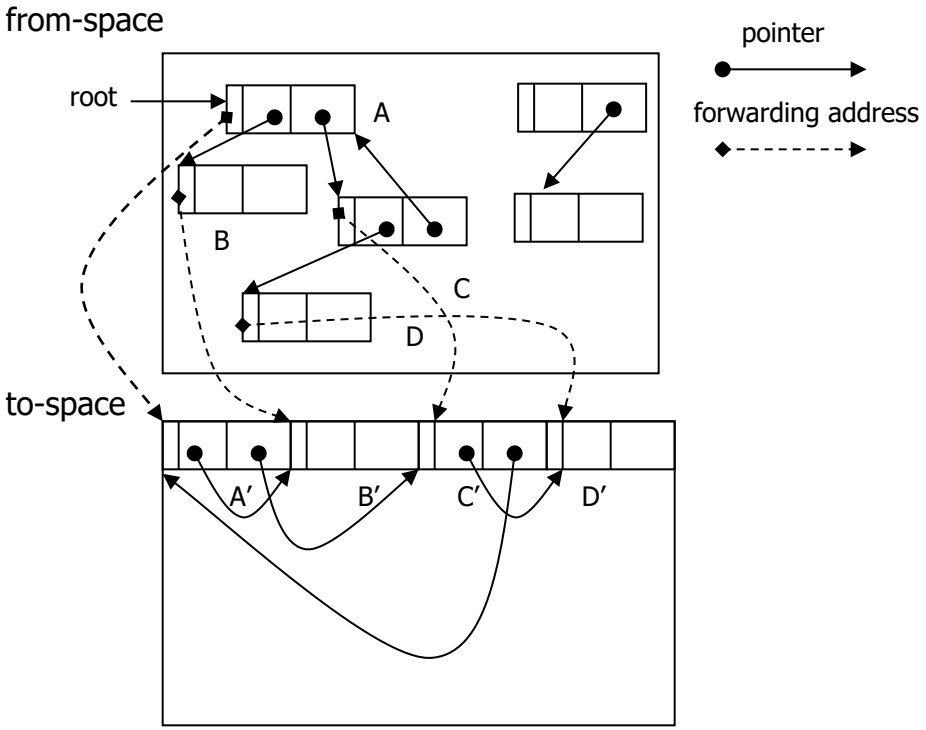
- L'**Algoritmo di Cheney** è un algoritmo di garbage collection che opera suddividendo la memoria heap in due parti
  - "**from-space**" e "**to-space**"
- Solamente una delle due parti dello heap è attiva (permette pertanto di allocare nuove celle)
- Quando viene attivato il garbage collector, le celle 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

Esempio

I blocchi vivi (raggiungibili) vengono spostati nella nuova area.

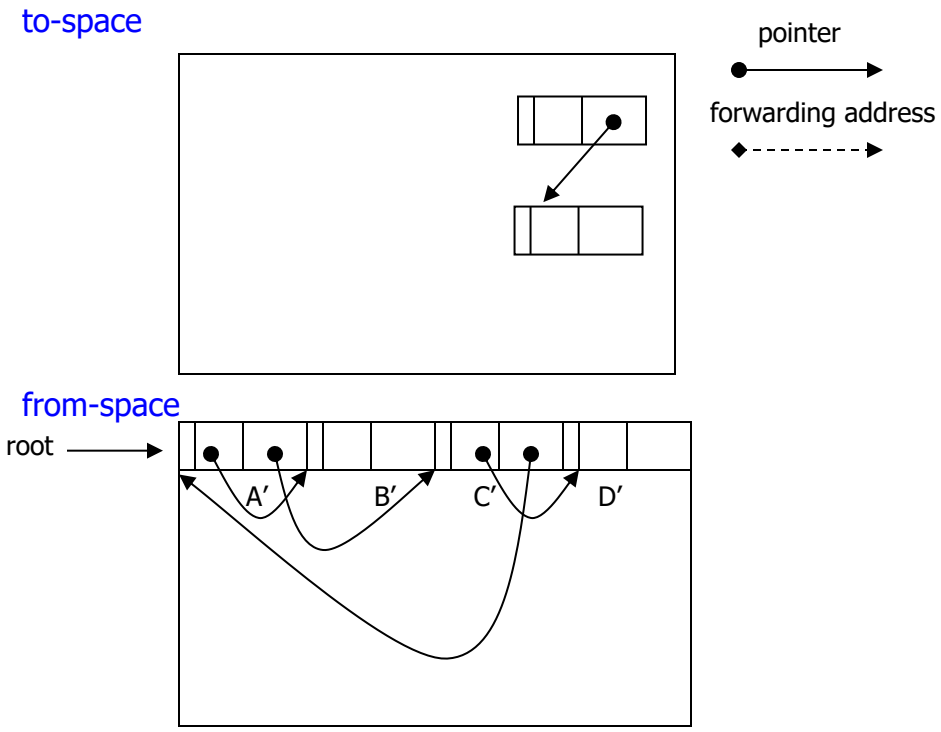
- Si esegue in tempo lineare nel numero dei blocchi vivi

Eventuali altri blocchi rimangono nella vecchia area



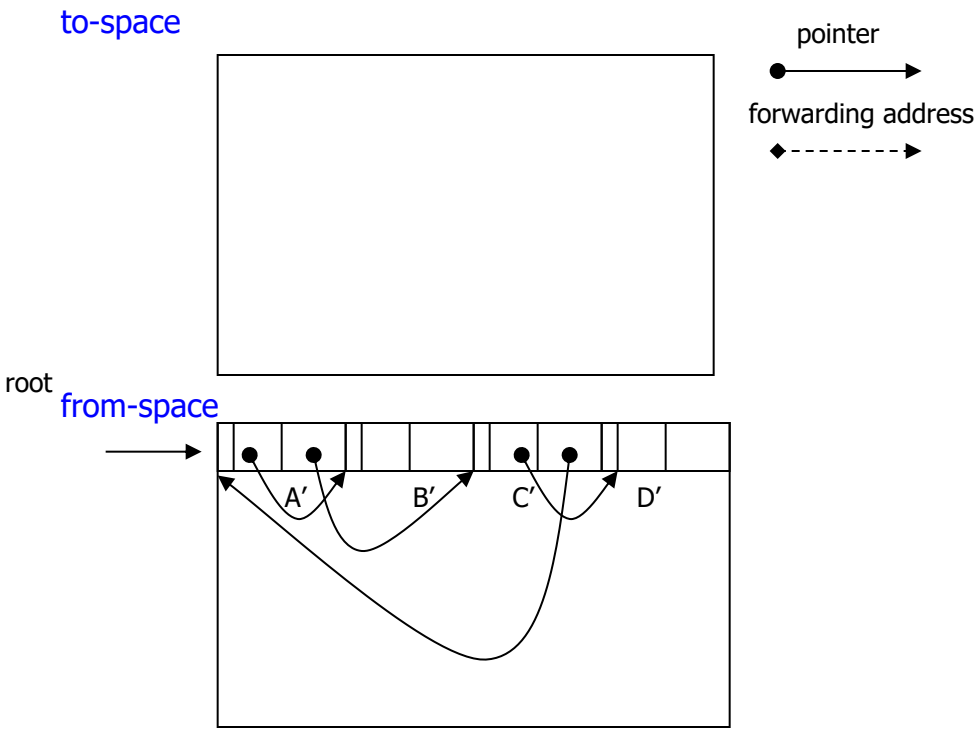
Scambio dei ruoli

Dopo lo spostamento si può fare sweep della prima area in tempo costante! (si "resetta" la free list)



Scambio dei ruoli

I ruoli delle due aree vengono invertiti



Copying collector: valutazione

È efficace nella allocazione di porzioni di spazio di dimensioni differenti e evita problemi di frammentazione

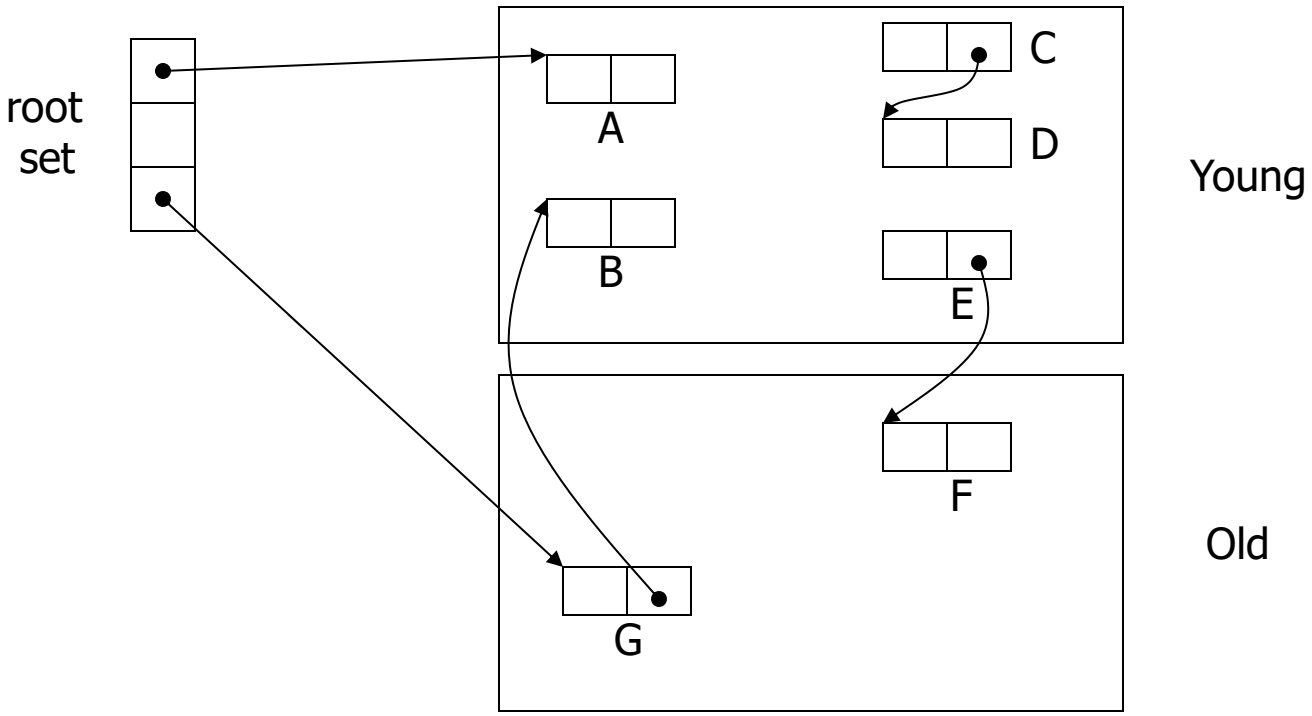
Caratteristica negativa: duplicazione dello heap

- dati sperimentali dicono che funziona molto bene su architetture hardware a 64-bit (tanta memoria)

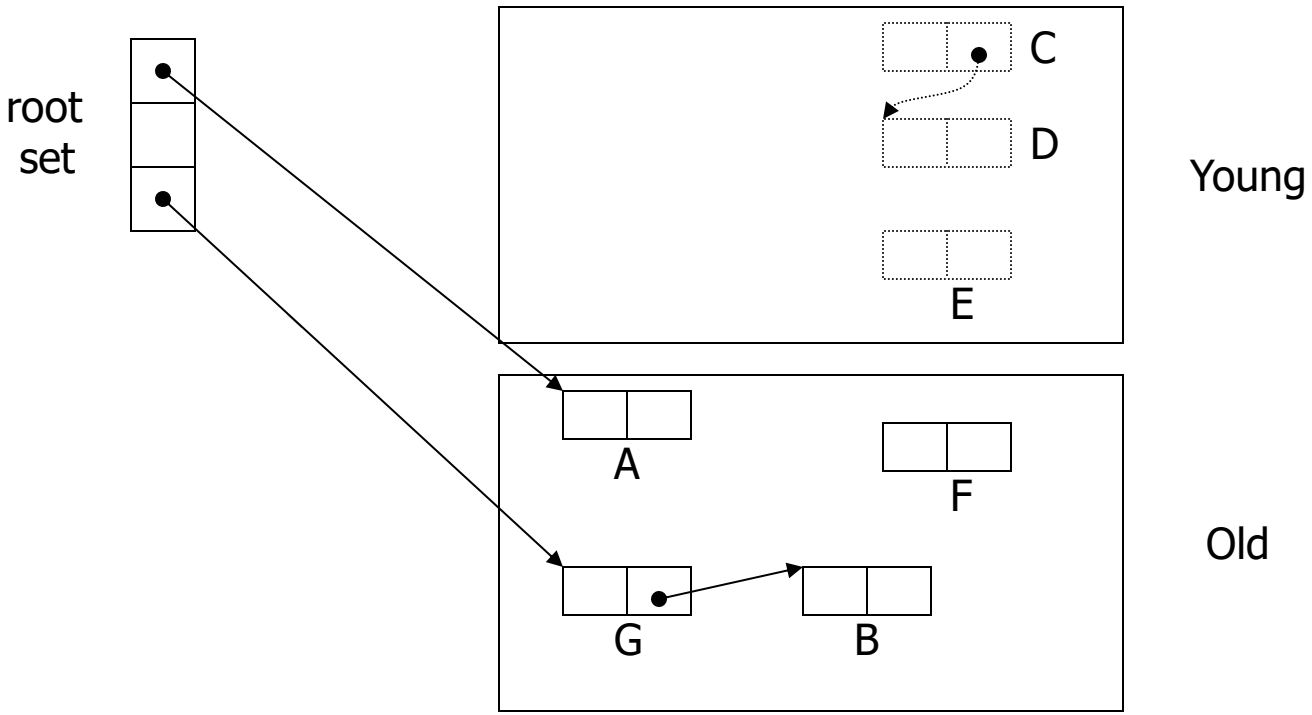
Generational Garbage Collection

- Osservazione di base
  - “most cells that die, die young” (ad esempio a causa delle regole di scope dei blocchi: i riferimenti memorizzati in variabili locali, che in linguaggi come Java sono la maggioranza, hanno un ciclo di vita brevissimo)
- Si divide lo heap in un insieme di **generazioni**
- Diversi approcci di garbage collection nelle diverse generazioni

Esempio (1)



Esempio (2)



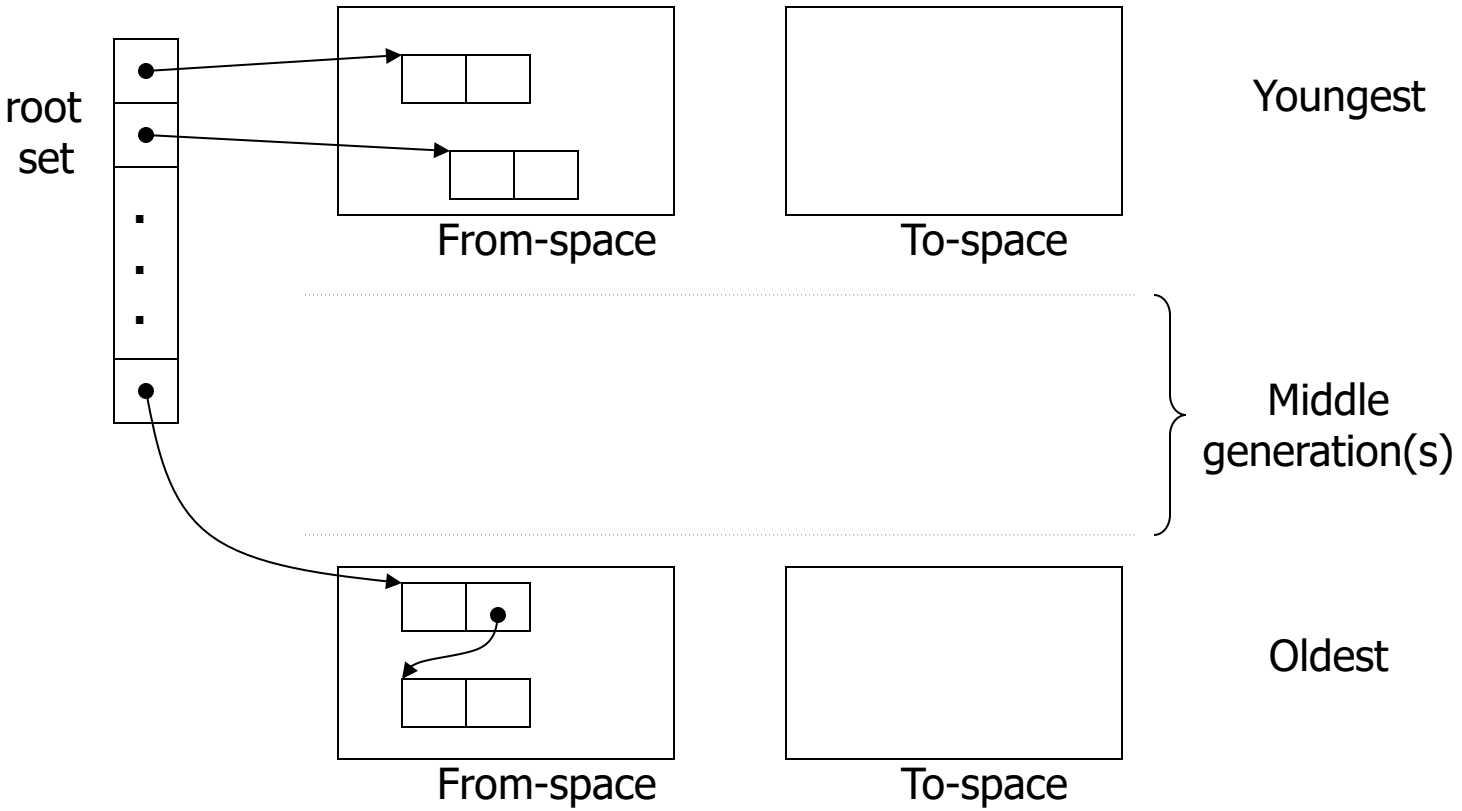
Generational Garbage Collection

- In sostanza
  1. si copiano i blocchi vivi in Old
     - Si fa rapidamente, perchè i blocchi vivi di solito sono la minoranza
  2. si ripulisce Young
     - In tempo costante
  3. si continua ad allocare nuovi blocchi in Young

- L'area di memoria Young può essere molto più piccola di quella Old (che mantiene i blocchi che hanno una vita più lunga)

- Per ripulire l'area Old si può usare un qualunque altro metodo di garbage collection (es. mark and sweep)

Copying con più generazioni



GC nella pratica

- Sun/Oracle Hotspot JVM
  - GC di default con tre generazioni (0, 1, 2)
  - Gen. 0,1 copy collection
  - Gen. 2 mark-sweep con meccanismi per evitare la frammentazione
  - esistono 4 GC alternativi tra i quali scegliere settando parametri a linea di comando (es. multithreading o low latency)
  - Per maggiori dettagli si veda

  https://docs.oracle.com/en/java/javase/15/gctuning/garbage-collector-implementation.html#GUID-71D796B3-CBAB-4D80-B5C3-2620E45F6E5D