PROGRAMMAZIONE 2
23. Gestione della Memoria e Garbage Collection

Gestione della memoria

- Static area
  - dimensione fissa, contenuti determinati e allocati a tempo di compilazione
- Runtime stack
  - dimensione variabile (record attivazione)
  - gestione sotto-programmi
- Heap
  - dimensione fissa/variabile
  - supporto alla allocazione di oggetti e strutture dati dinamiche
    - **`malloc`** in **C**, **`new`** `in` **`Java`**

Allocazione statica

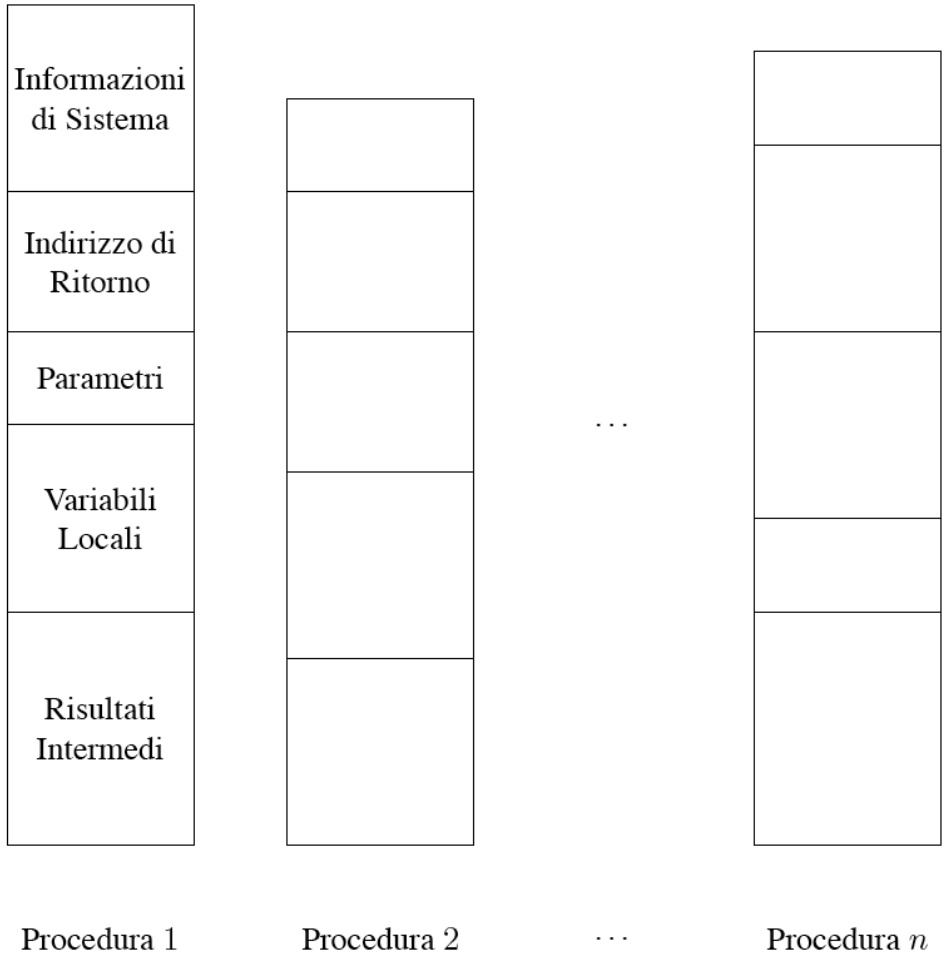
- Entità che ha un indirizzo assoluto che è mantenuto per tutta l'esecuzione del programma
- Solitamente sono allocati staticamente
  - variabili globali
  - costanti determinabili a tempo di compilazione
  - tabelle usate dal supporto a runtime (per type checking, garbage collection, ecc.)
  - variabili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

Allocazione statica: sotto-programmi

Registri salvati

Informazione debugging

Spesso nei registri



Chiamate successive della stessa procedura condividono la stessa zona di memoria

Allocazione dinamica: pila

- Per ogni istanza di un sotto-programma a runtime 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…
- Blocchi in-line seguono una politica LIFO

Allocazione dinamica con heap

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

- Necessario quando il linguaggio (es. Pascal, C, Java) permette
  - allocazione esplicita di memoria a runtime
  - 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

Situazione iniziale

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

Heap: blocchi di dimensione fissa

Allocazione di uno o più blocchi contigui

Deallocazione: restituzione a lista libera

Gestione semplice a patto che si riesca
a recuperare la memoria
da restituire alla lista libera

Heap: blocchi di dimensione variabile

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

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

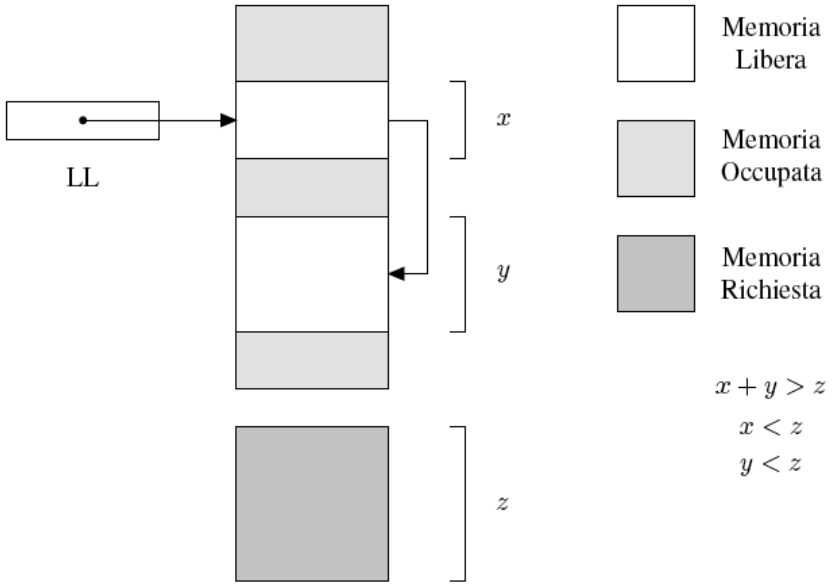
Frammentazione

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 non è usabile perché suddiviso in "pezzi" troppo piccoli



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 non utilizzata è aggiunta alla lista libera
- Quando un blocco è de-allocato, viene restituito alla lista libera (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 lista libera 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

Come identificare i blocchi da de-allocare?

- Nella lista libera 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 la "raggiungibilità" e un associato meccanismo automatico di garbage collection

Modello a grafo della memoria

- È necessario determinare il **root set** cioè l'insieme dei dati sicuramente "attivi"
  - **`Java root set`** = variabili statiche + variabili allocate su runtime 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

Celle, “liveness”, blocchi e garbage

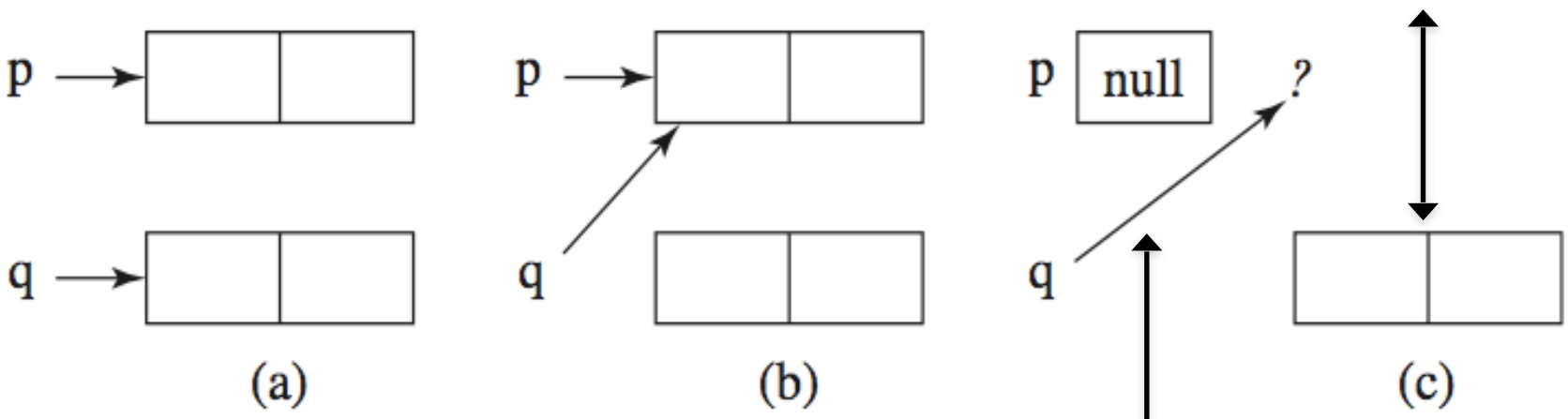
- Cella = blocco di memoria sullo heap
- Una cella viene detta live se il suo indirizzo è memorizzato in una radice o in una altra cella live
  - quindi: una cella è live se e solo se appartiene ai *reachable active data*
- Una cella è garbage se non è live
- Garbage collection (GC): attività di gestione della memoria dinamica consistente nell’individuare le celle garbage (o “il garbage”) e renderle riutilizzabili, per esempio inserendole nella lista libera

Garbage e dangling reference

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

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

garbage



Dangling reference

GC: perché è interessante?

- Applicazioni moderne sembrano non avere limiti allo spazio di memoria
  - 8 GB laptop, 32 GB desktop, 32-1024 GB 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*

GC e linguaggi di programmazione

- 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 (per 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 Garbage Collection?

- *Reference counting – Contatori dei 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 up-to-date: generational GC*

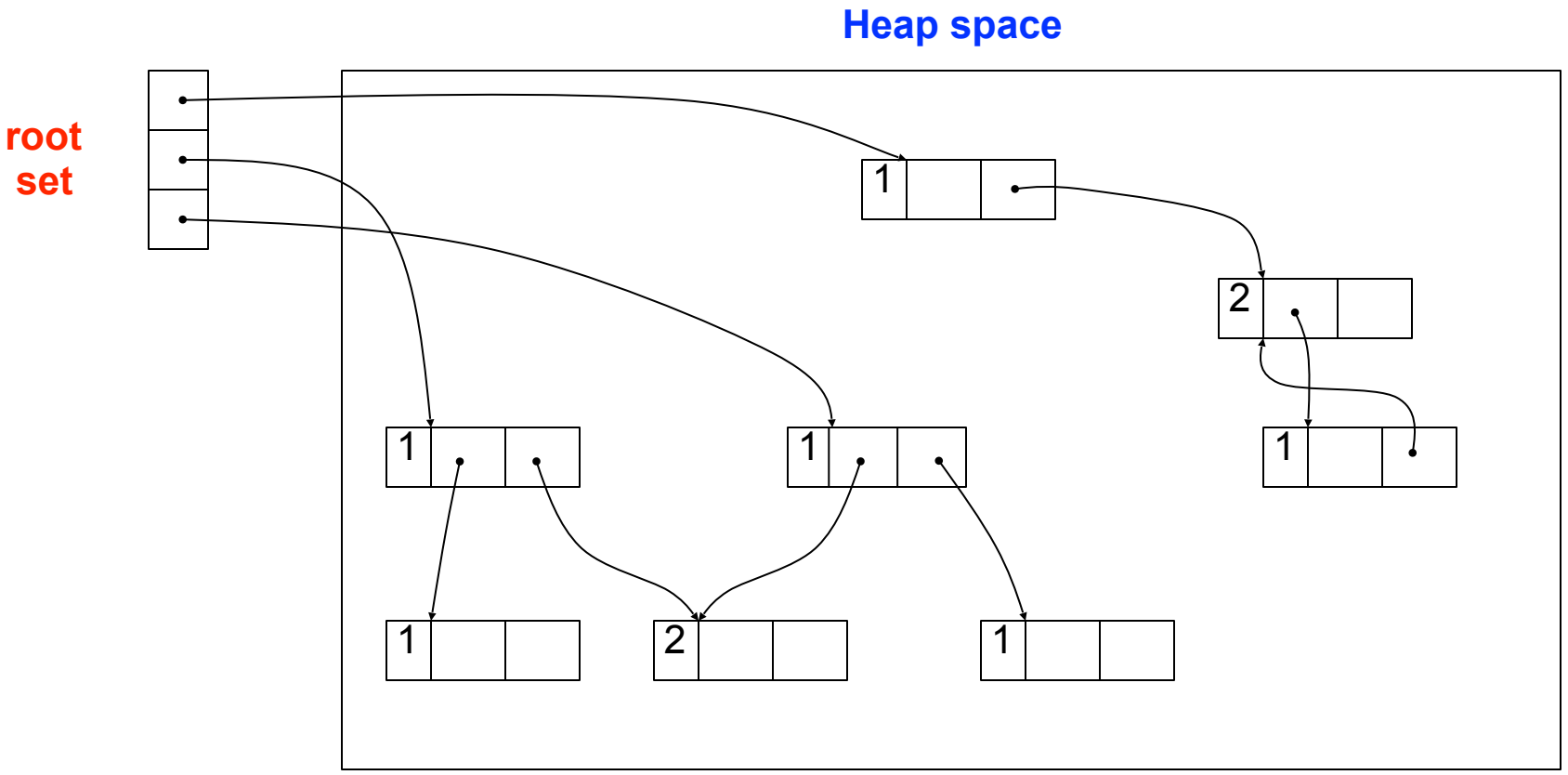
Reference counting

- Aggiungere un contatore di riferimenti ad ogni cella (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”

Reference counting

- `Integer i = new Integer(10);`
  - RC (i) = 1
- `j = k;` (con `j, k` che riferiscono a oggetti)
  - RC(j) --
  - RC(k) ++

Reference counting: esempio



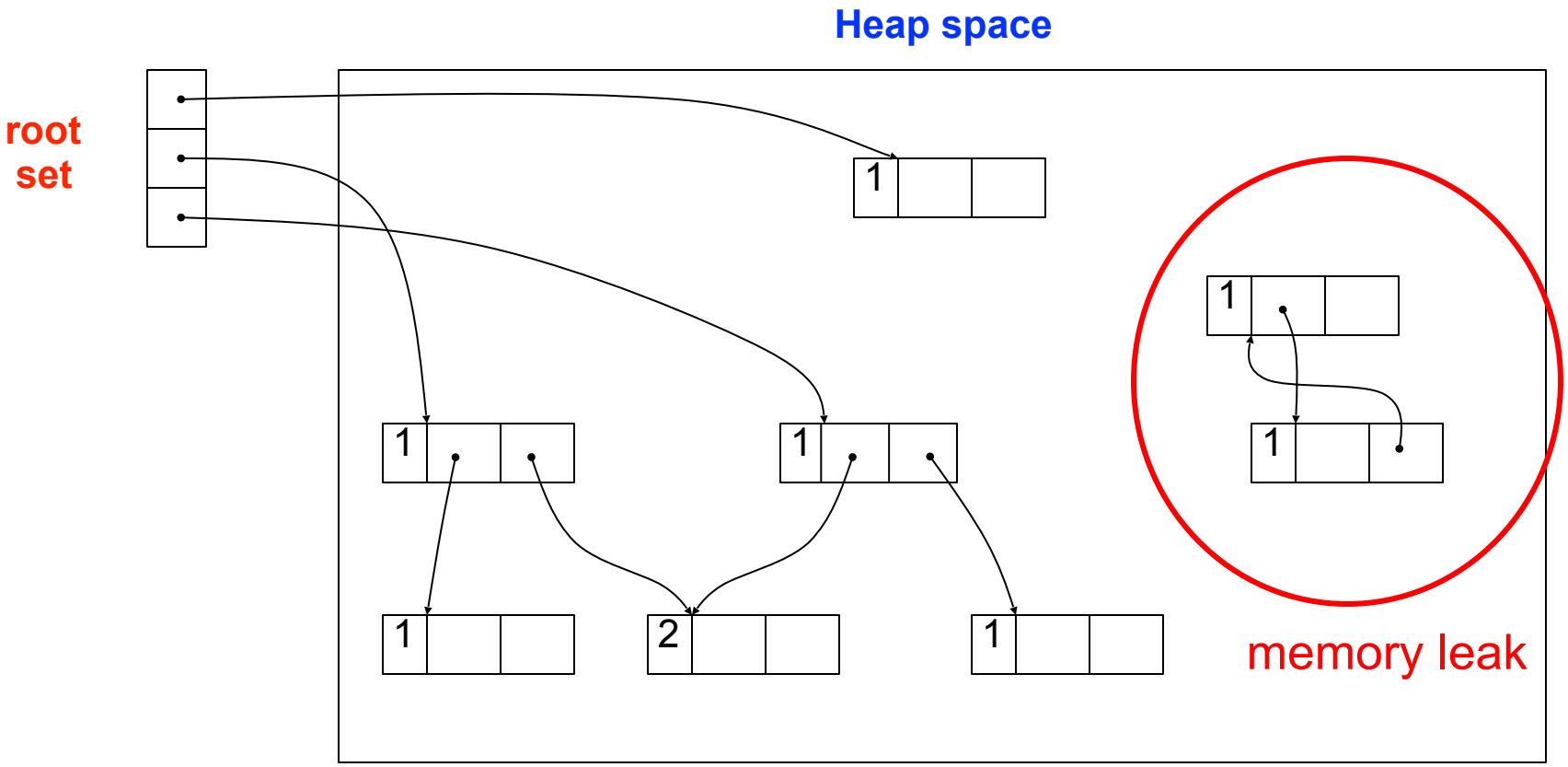
Reference counting: caratteristiche

- 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: limitazioni

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

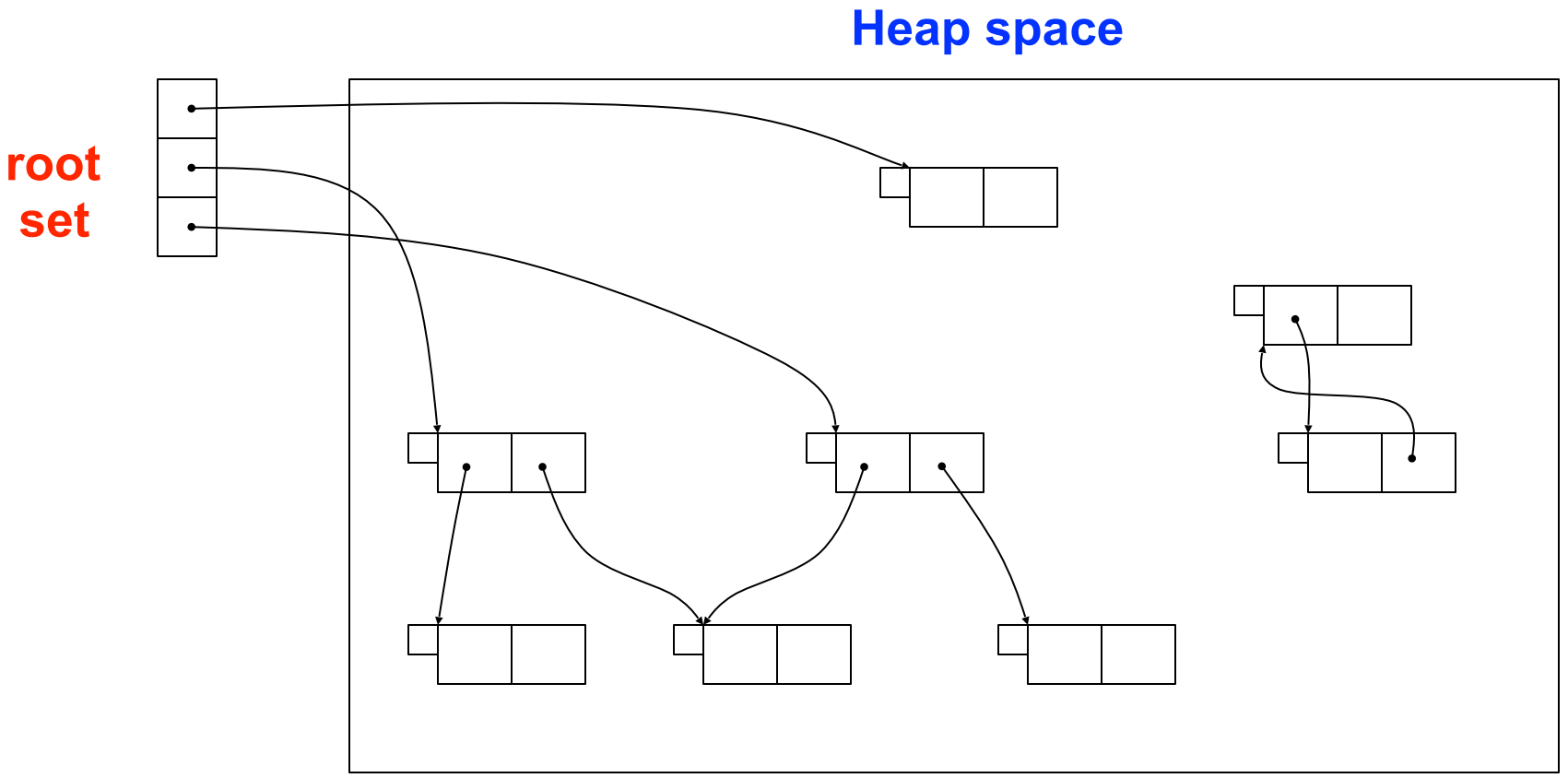
Reference counting: cicli



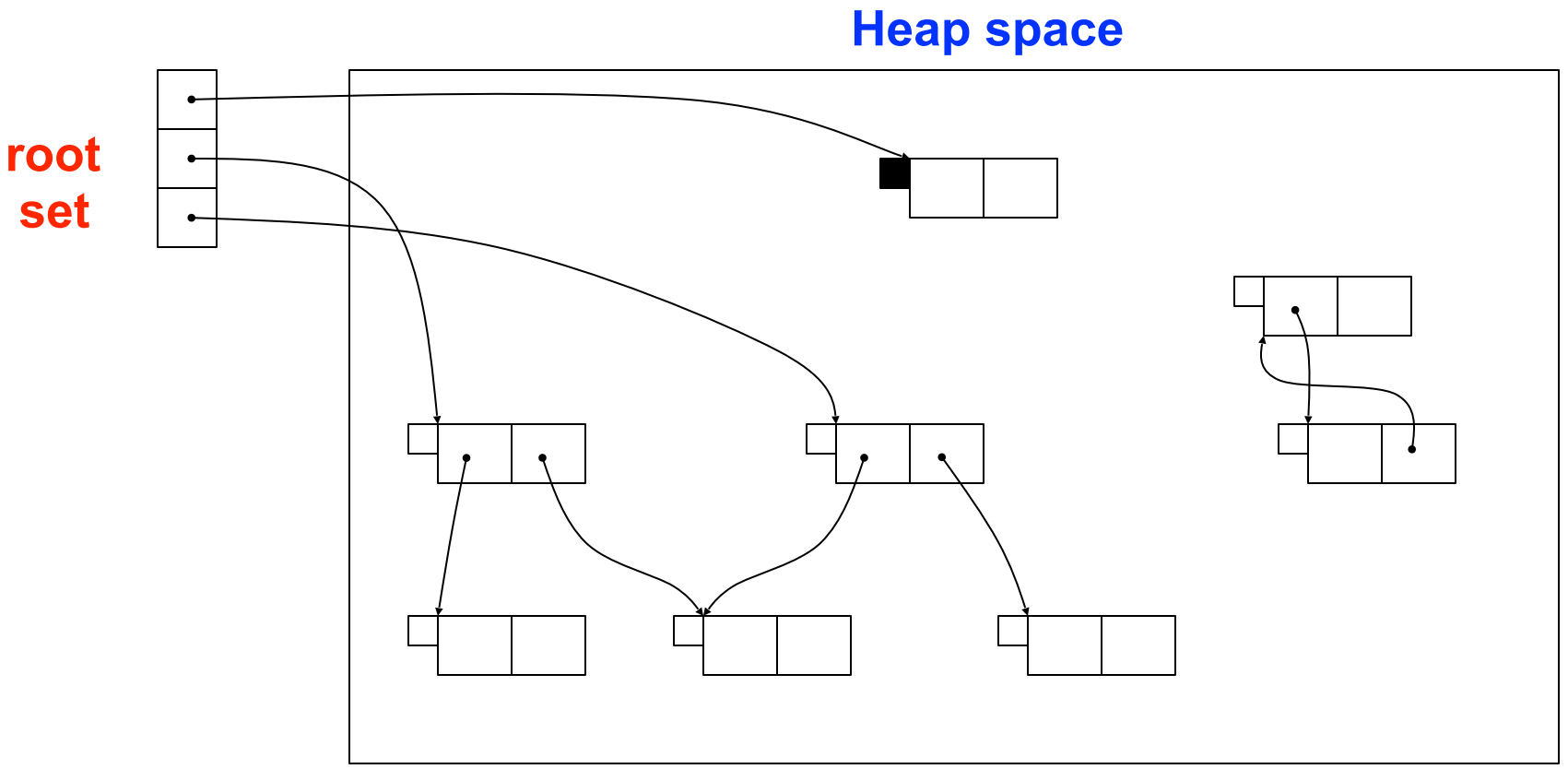
mark-sweep

- Ogni cella prevede spazio per un bit di marcatura
- GC 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

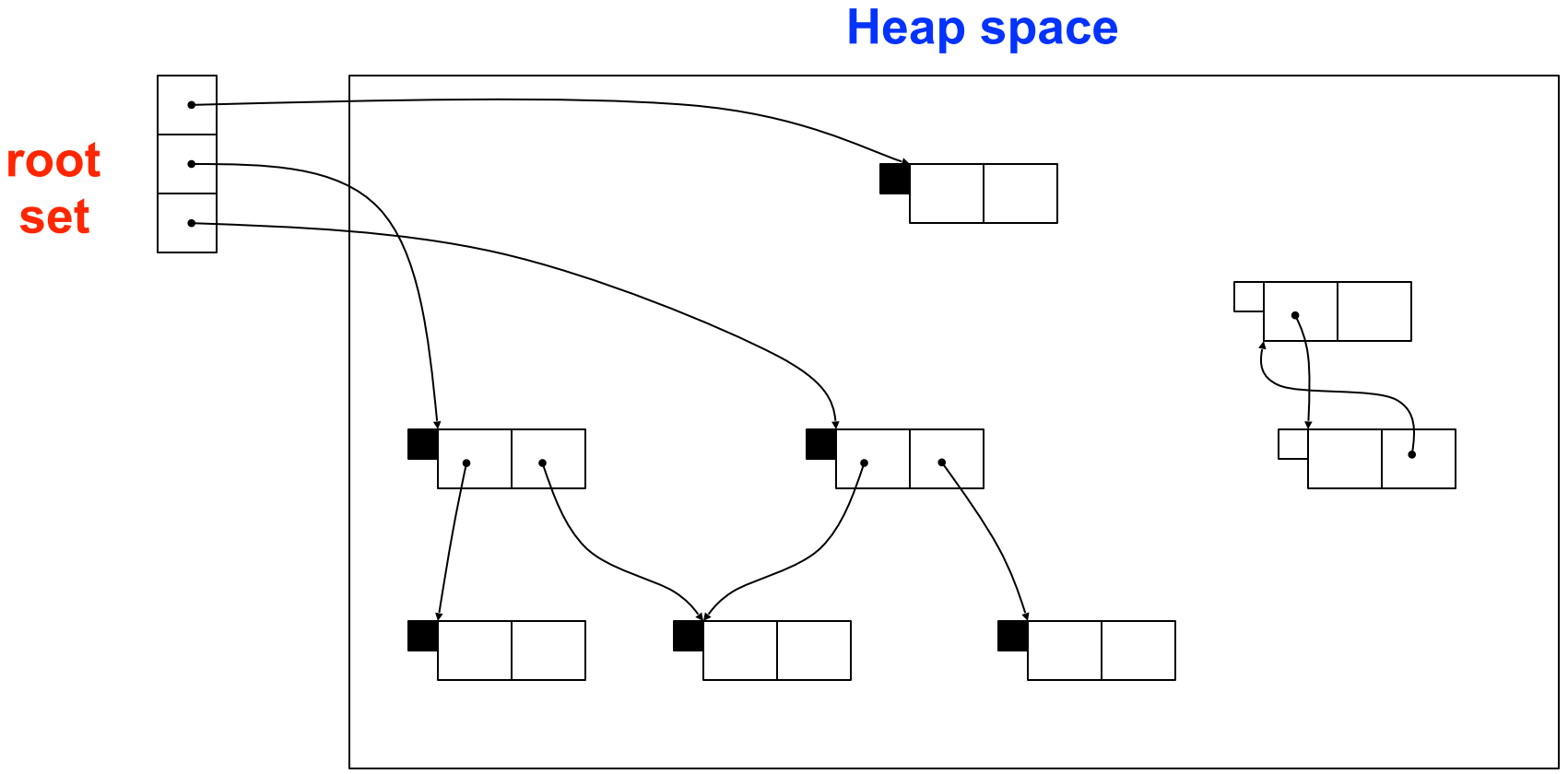
mark-sweep (1)



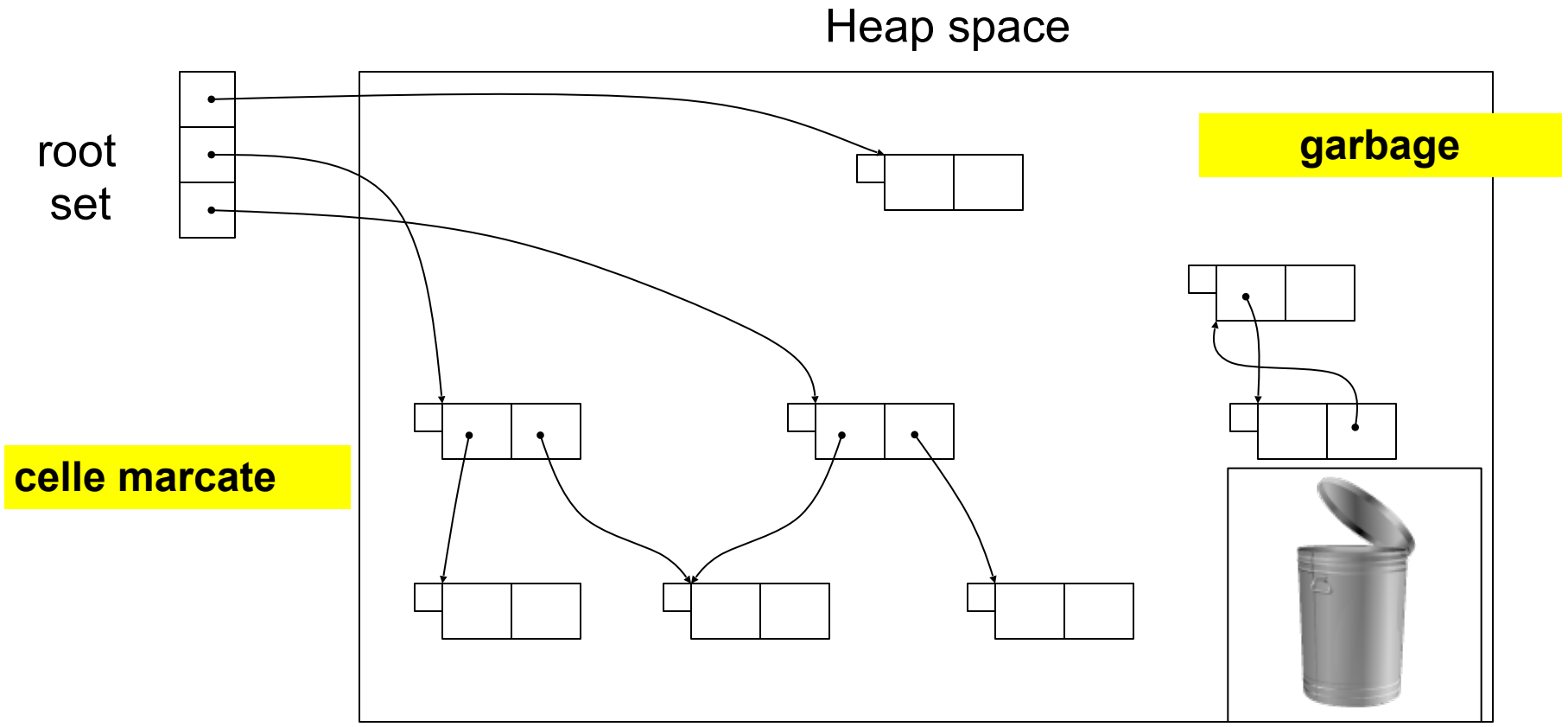
mark-sweep (2)



mark-sweep (3)



mark-sweep (4)



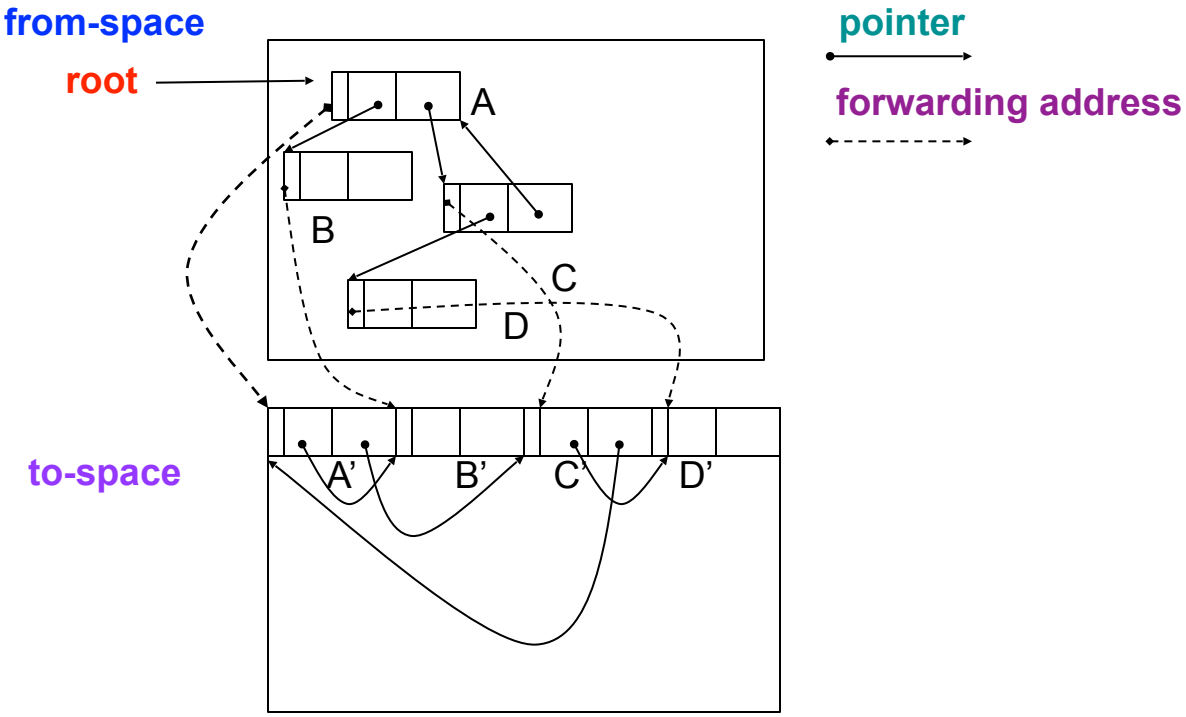
mark-sweep: valutazione

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

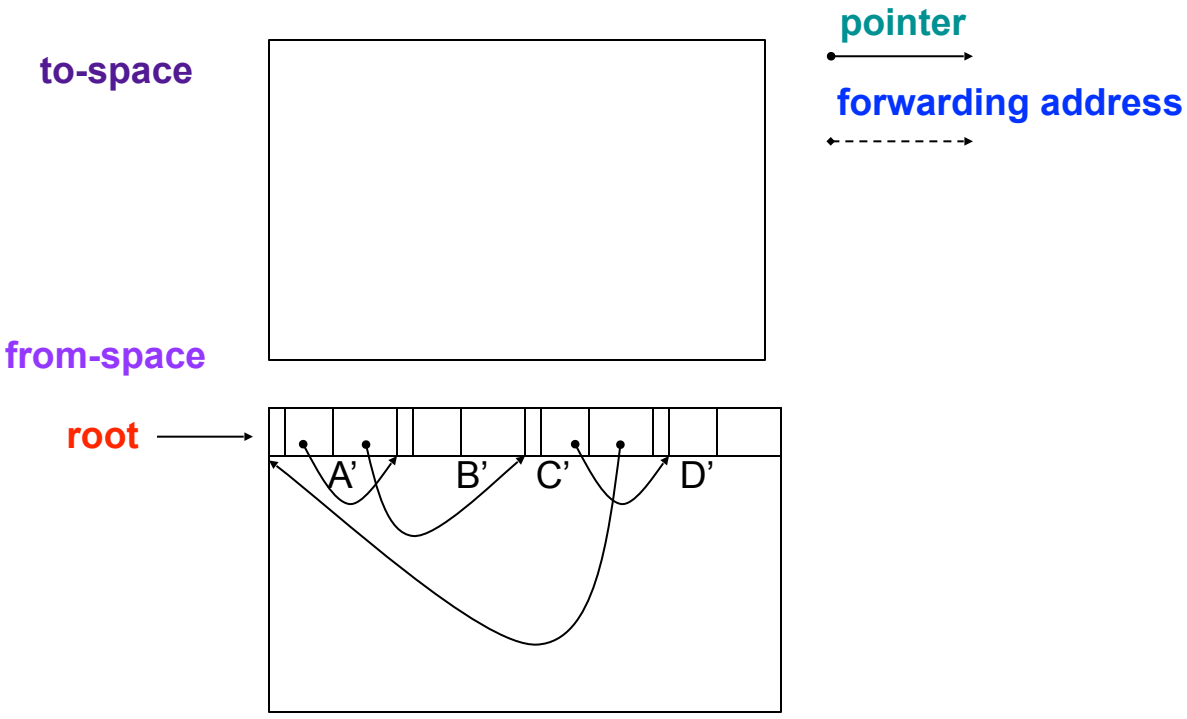
Copying collection

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

Esempio


from-space
root
A
B
C
D
to-space
A'
B'
C'
D'
pointer
forwarding address

Scambio dei ruoli



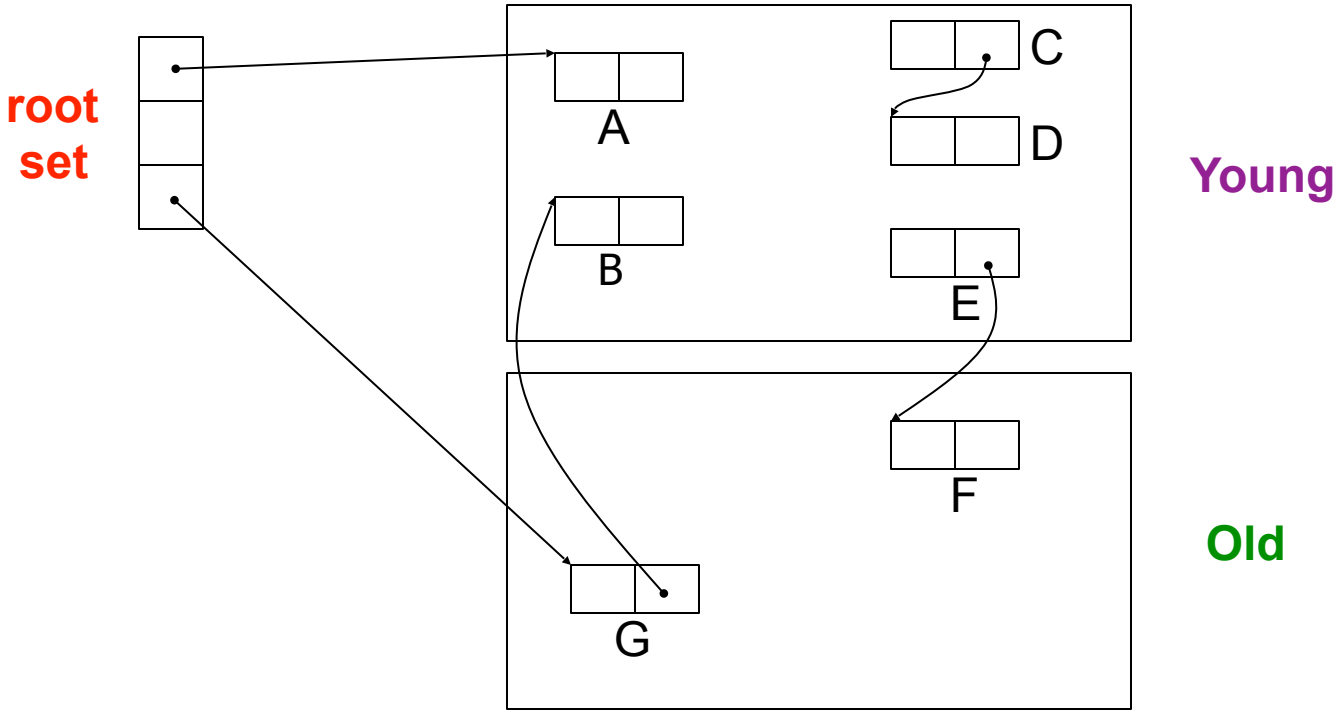
Copying collection: valutazione

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

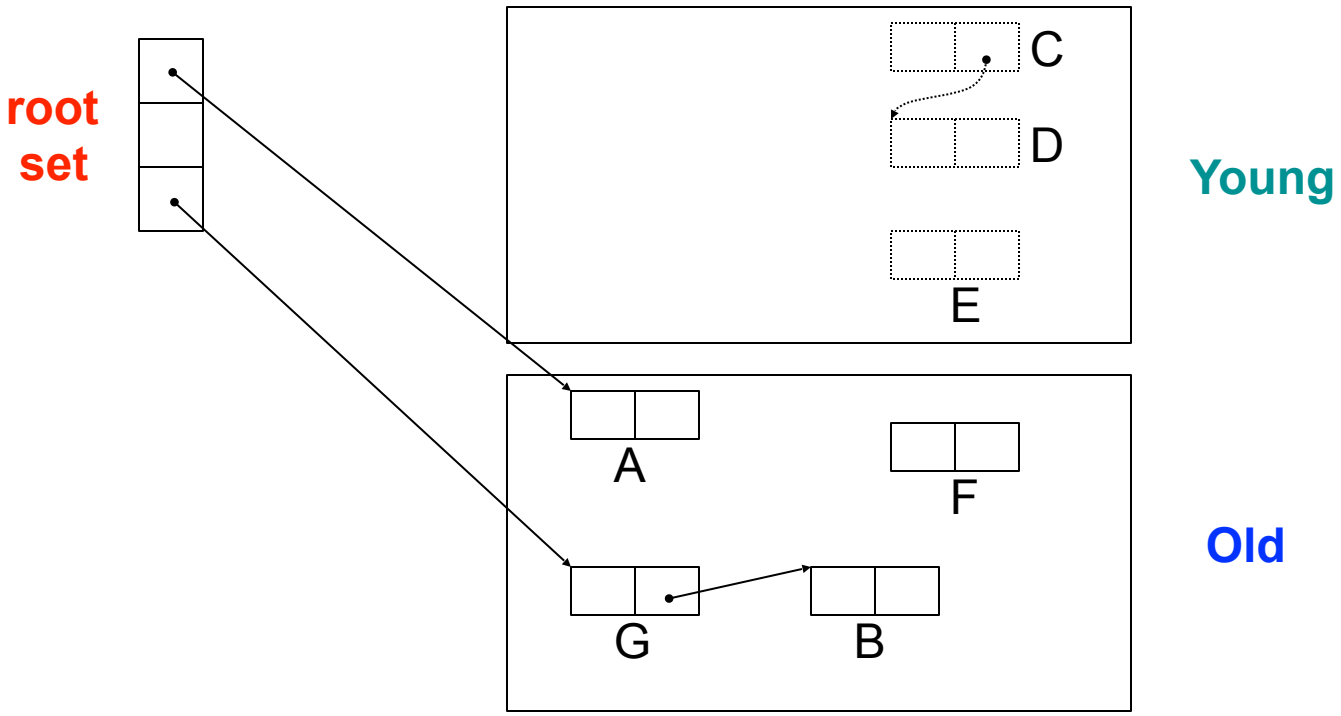
Generational Garbage Collection

- Osservazione di base
  - “most cells that die, die young” (per 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

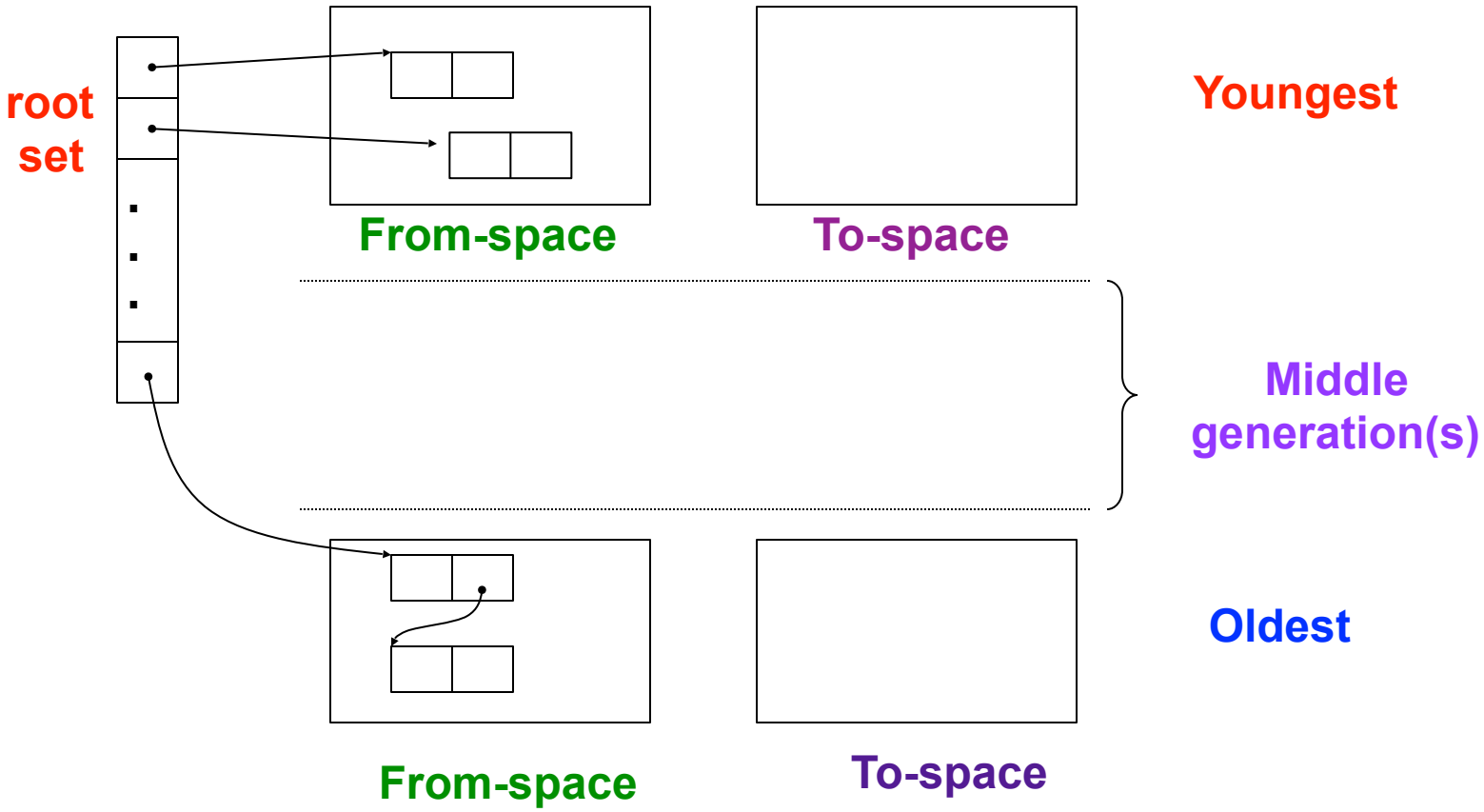
Esempio (1)



Esempio (2)



Copying + generazioni



Garbage Collector nella pratica

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