

---

# Gerenciamento Automático De Memória

## *Garbage Collection*

**Sandro Rigo**  
**sandro@ic.unicamp.br**

# Introdução

---

- Alocação dinâmica ocorre no chamado espaço-livre ou *heap*
- Gerenciamento compreende
  - Alocação de novos recursos
  - Recuperação de recursos não mais necessários
- Duas formas de fazer
  - Explícita:
    - Fica a cargo do programador
  - Automática:
    - Coletor de lixo

# Introdução

---

- **Explícita:**
  - Maior desempenho
  - Maior dificuldade
  - Mais suscetível a erros
- **Automático:**
  - Facilidade de desenvolvimento
  - Impacto negativo no desempenho

# Contagem de Referências

---

- Cada objeto possui um contador
- Deve ser atualizado a cada operação
  - Nova referência criada
  - Referência antiga destruída
- O compilador gera as instruções para atualizar a contagem
- Contador = 0
  - Objeto é recuperado
  - Atualiza todos os referenciados por ele

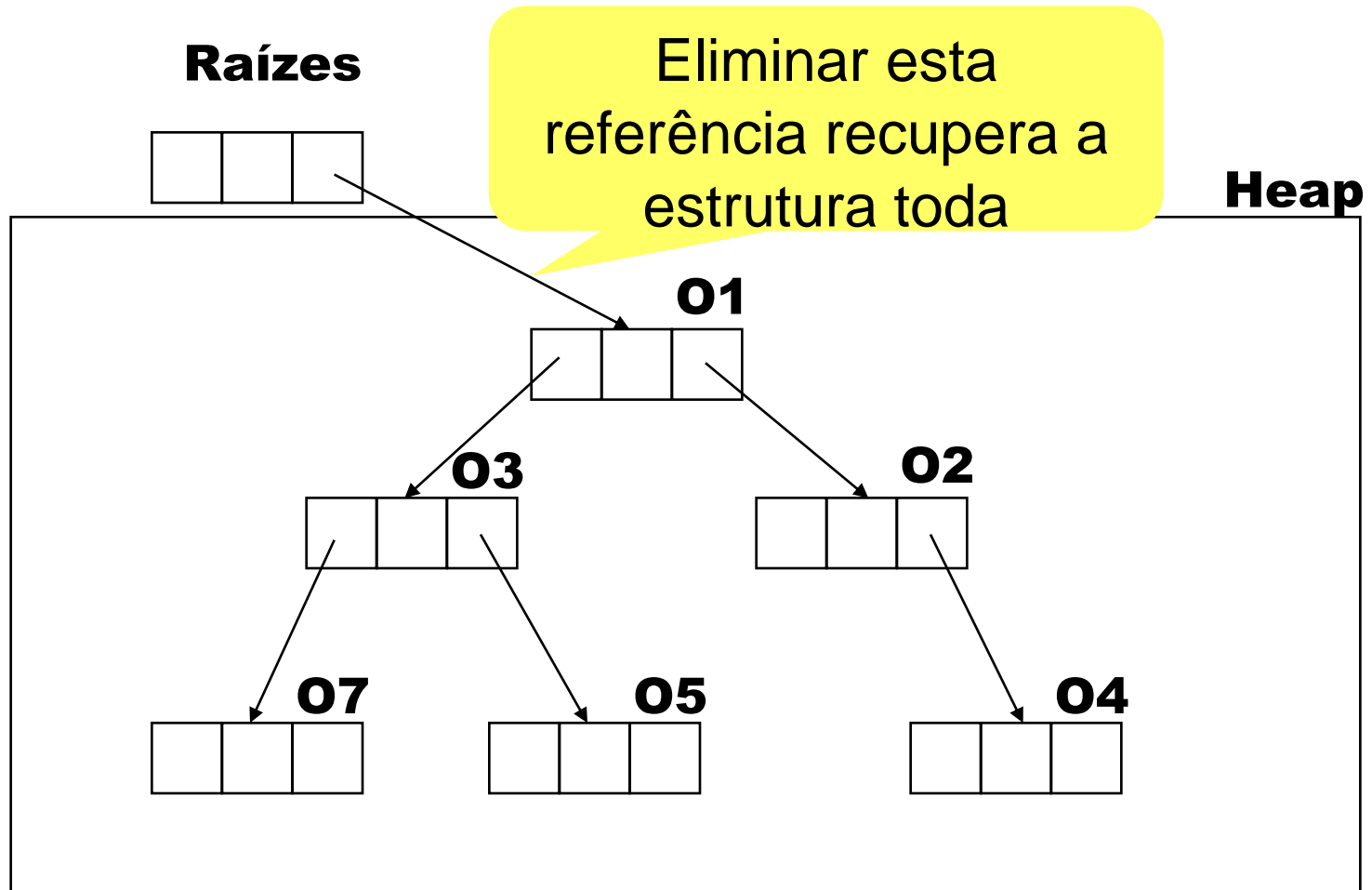
# Contagem de Referências

---

- Vantagens

- Coleta é distribuída ao longo da execução
- Não tem interrupção para recuperação
- Pode ser útil para sistemas de tempo real
- Recuperação de grandes estruturas não precisa ser feito todo de uma vez
- Localidade de referência
  - Objeto é recuperado sem visitar outras páginas da memória

# Contagem de Referências

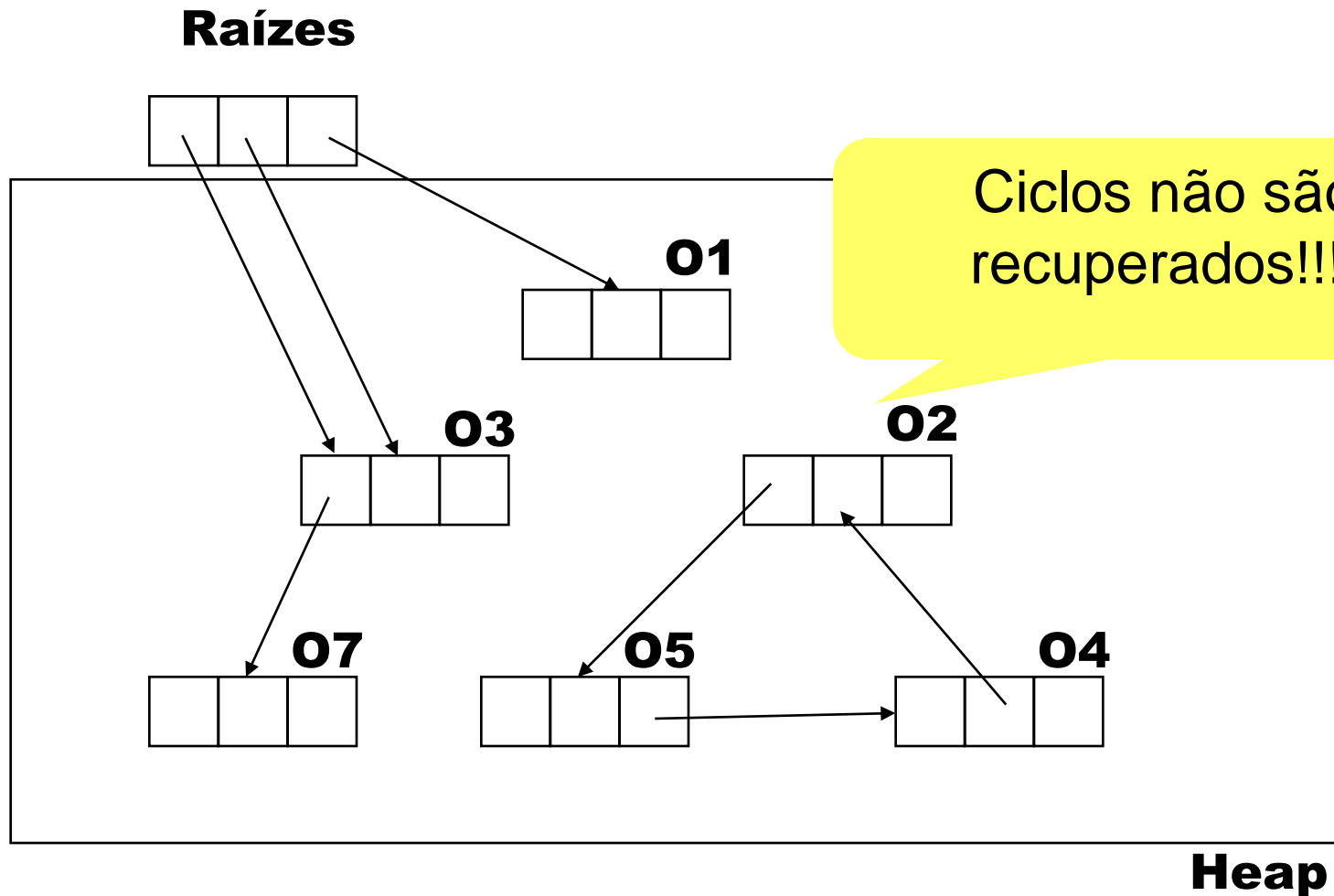


# Contagem de Referências

---

- Desvantagens
  - Alto custo para manter todos os contadores atualizados
- Algum outro problema?
- Veja a estrutura do exemplo a seguir

# Contagem de Referências





# Contagem de Referências

---

- Desvantagens
  - Alto custo para manter todos os contadores atualizados
  - **Não recupera estruturas cíclicas !!!**
- Possíveis soluções
  - Obrigar programador romper os ciclos
  - Alternar com coletas de outro algoritmo

# Coleta de Lixo

---

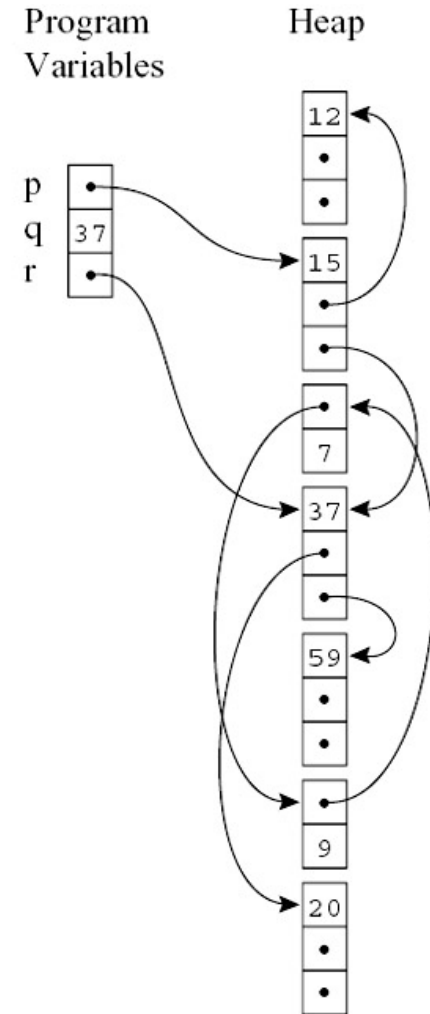
- Ponto de vista teórico:
  - O heap é um grafo dirigido  $G$
  - Vértices: objetos
  - Arestas  $(x.f,y)$  indica que existe um apontador do campo  $f$  em  $x$  para  $y$
  - Raízes de  $G$  são as variáveis locais do programa
  - Um objeto  $x_n$  está morto se não existir um caminho  $r \rightarrow x_1 \rightarrow x_2 \rightarrow \dots \rightarrow x_n$  em  $G$ , para qualquer raiz  $r$

# Exemplo

```
class list {list link;
           int key; }

class tree {int key;
           tree left;
           tree right; }

class main {
  static tree maketree() { ... }
  static void showtree(tree t) { ... }
  static void main() {
    {list x = new list(nil,7);
     list y = new list(x,9);
     x.link = y;
    }
    {tree p = maketree();
     tree r = p.right;
     int q = r.key;
     garbage-collect here
     showtree(r);
    }
  }
}
```



# Mark-and-Sweep

---

- A idéia é marcar os nós vivos do grafo
- Nós vivos são os alcançáveis a partir das raízes
- Isso pode ser feito através de uma busca no grafo (DFS)
- Os nós não marcados são lixo
  - Devem ser recuperados

# Mark-and-Sweep

---

- Varredura (sweep):
  - Varre-se todo o heap
  - Qualquer objeto não marcado é colocado numa lista de objetos livres
  - Desmarcar todos os nós marcados
- Interrompe a execução durante a coleta
- O programa aloca novos objetos da freelist
- Freelist vazia: dispara nova coleta

# Mark-and-Sweep

---

function DFS(x)

if x is a pointer into the heap

if record x is not marked

mark x

for each field  $f_i$  of record x

DFS(x.  $f_i$ )

# Mark-and-Sweep

---

## Mark phase:

for each root  $v$

DFS( $v$ )

## function DFS( $x$ )

if  $x$  is a pointer into the heap

if record  $x$  is not marked

mark  $x$

for each field  $f_i$  of record  $x$

DFS( $x.f_i$ )

# Mark-and-Sweep

---

## Sweep phase:

```
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)
```



# Custo

---

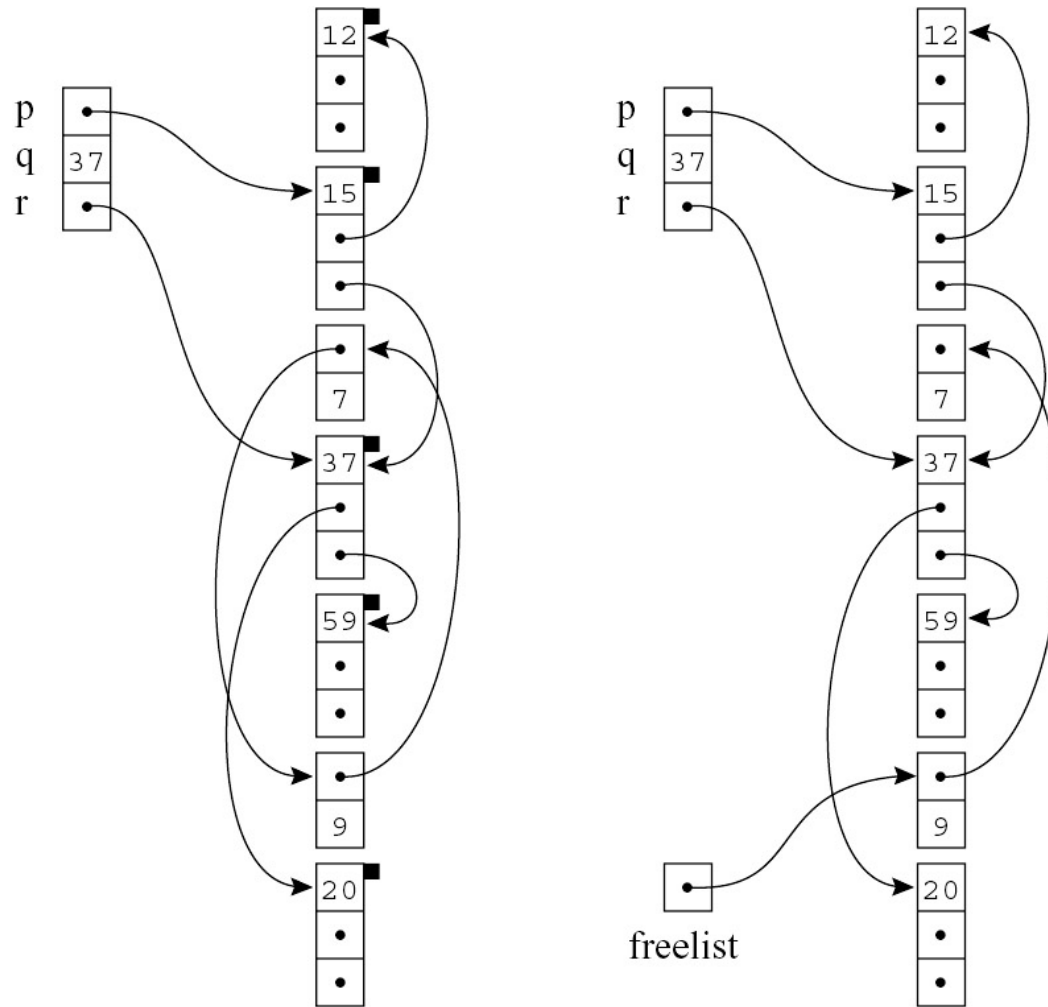
- DFS  $\approx$  # nós do grafo
  - Em GC é o # objetos vivos (R)
- Sweep  $\approx$  tamanho do heap (H)
- 1 coleta
  - $c1.R + c2.H$ 
    - $c1$  e  $c2$  constantes
- Recupera H-R palavras
- Amortizando:  $(c1.R + c2.H) / (H-R)$

# Custo

---

- Se torna grande se  $R$  é próximo de  $H$
- Recupera muito pouco
- É possível para o GC pedir mais memória ao SO
  - Aumenta  $H$
- Ex.: se  $R/H \geq 0.5$

# Exemplo



(a) Marked

(b) Swept

# Cuidados com Implementação

---

- DFS é recursivo
- Máxima profundidade pode ser  $H$
- Pilha de execução teria tamanho maior do que o heap!
- Podemos adotar uma pilha explícita

# Pilha Explícita

---

function DFS(x)

if x is a pointer and record x is not marked

mark x

t  $\leftarrow$  1

stack[t]  $\leftarrow$  x

while t > 0

    x  $\leftarrow$  stack[t]; t  $\leftarrow$  t - 1

    for each field fi of record x

        if x. fi is a pointer and record x. fi is not marked

            mark x. fi

            t  $\leftarrow$  t + 1; stack[t]  $\leftarrow$  x. fi

# Cuidados com Implementação

---

- Pilha agora pode chegar a tamanho  $H$  palavras
  - Ainda assim é inaceitável
- Podemos melhorar?
- Sim
  - Pointer Reversal

# Pointer Reversal

---

- Após colocar  $x.fi$  na pilha, DFS não precisa dele novamente
- Podemos usar  $x.fi$  para armazenar um elemento da pilha!
- $x.fi$  apontará para o registro do qual  $x$  foi alcançado
- Pop restaurará  $x.fi$

# Pointer Reversal

---

```
function DFS(x)
  if x is a pointer and record x is not marked
    t ← nil
    mark x; done[x] 0
    while true
      i done[x]
      if i < # of fields in record x
        y ← x. fi
        if y is a pointer and record y is not marked
          x. fi ← t; t ← x; x ← y
          mark x; done[x] 0
        else
          done[x] ← i + 1
      else
        y ← x; x ← t
        if x = nil then return
        i ← done[x]
        t ← x. fi; x. fi ← y
        done[x] ← i + 1
```



# Lista de Objetos Livres

---

- Chamadas de freelists
- Pode haver objetos de tamanhos diferentes
- Uma lista única não é eficiente
- É comum existir um array de freelists
  - $\text{Freelist}[i]$  é uma lista de nós de tamanho  $2^i$
- Tentativa de alocar de uma lista vazia:
  - Pega o primeiro livre  $\geq$  tamanho requerido

# Fragmentação

---

- Pode acontecer de haver vários objetos pequenos livres
- Porém não haver objetos grandes
- A soma dos objetos livres seria suficiente para alocar pelo menos um grande
- Isso é fragmentação externa!

# Fragmentação

---

- Um objeto pode ser menor do que o mínimo da lista de livres
- Ele será alocado para um registro maior do que seu tamanho
- Isto é fragmentação interna!

# Copy Collection

---

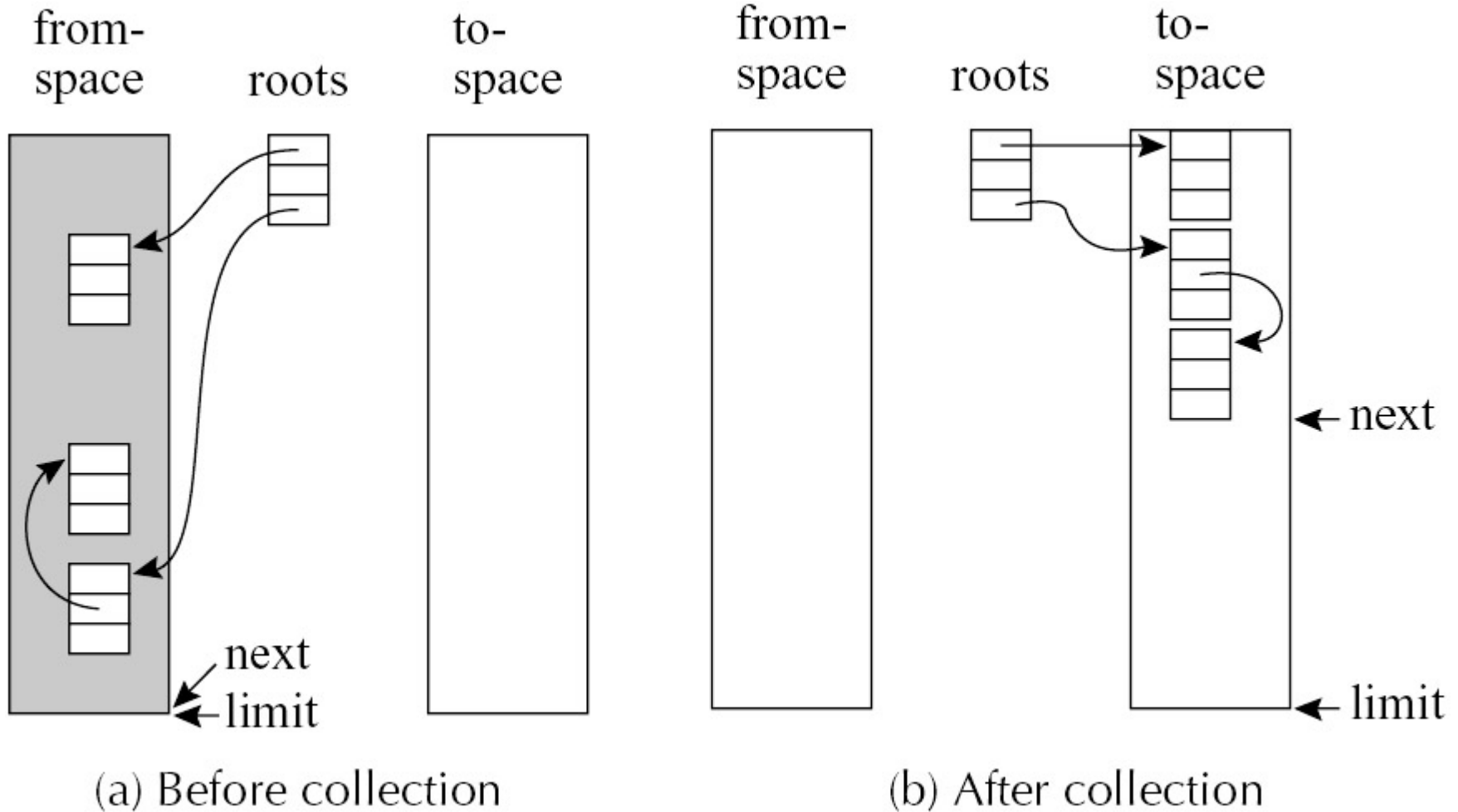
- Atravessa o heap para encontrar objetos vivos
- Não tem custo nas operações com referências
- Não tem problemas com ciclos
- Divide o heap em dois espaços
  - From-space
  - To-space

# Copy Collection

---

- Atravessa os objetos vivos no from-space
- Cria uma cópia no to-space
- Deixa os mortos no from-space
- Recomeça a execução pelo to-space

# Exemplo



# Copy Collection

---

- Next: próxima posição para alocação
- Limit: final do heap
- Next = limit?
  - Nova coleta iniciada
- Coleta
  - Next vai para o início do to-space
  - Os registros vivos são copiados para a posição apontada por next
  - Forwarding: operação que ajusta os apontadores

# Forwarding

---

- Para cada  $p$  apontando para o from-space
  - Altere para  $p$  passar a apontar para o to-space
- 1.  $p$  aponta para um registro já copiado
  - $P.f1$  é um ponteiro especial que mostra para onde foi copiado no to-space
- 2.  $p$  aponta para um registro não copiado
  - Copia o registro para o local indicado por next
  - Armazena o forwarding pointer em  $p.f1$



# Forwarding

## 3. p aponta para fora do from-space ou não é um ponteiro

- Ignore p

```
function Forward(p)
  if p points to from-space
    then if p. f1 points to to-space
      then return p. f1
    else for each field fi of p
      next. fi ← p. fi
      p. f1 ← next
      next ← next+ size of record p
    return p. f1
  else return p
```

# Algoritmo de Cheney

---

scan  $\leftarrow$  next  $\leftarrow$  beginning of to-space

for each root r

    r  $\leftarrow$  Forward(r)

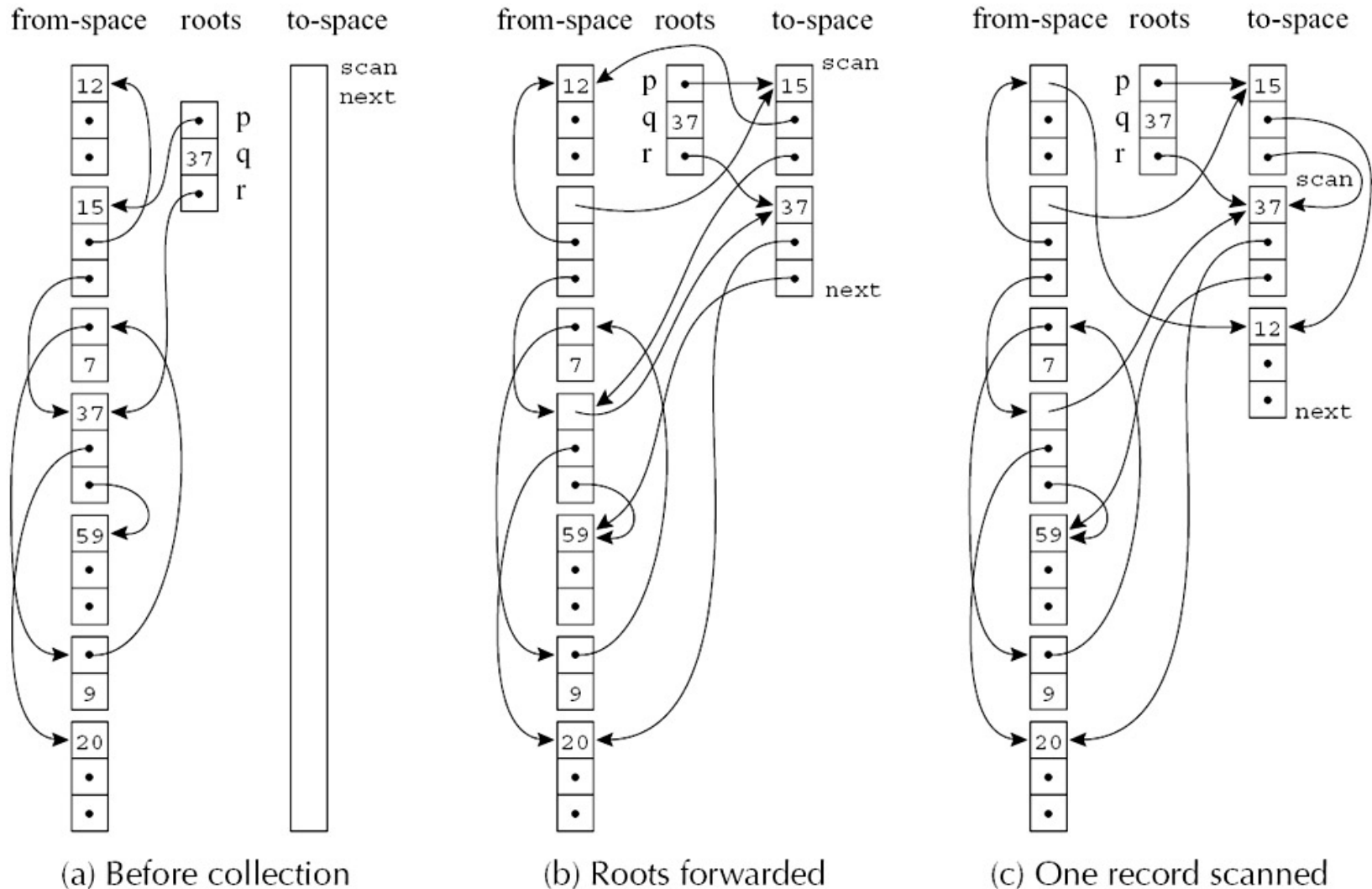
while scan < next

    for each field fi of record at scan

        scan.fi  $\leftarrow$  Forward(scan.fi)

    scan  $\leftarrow$  scan+ size of record at scan

# Exemplo



# Comparação

---

- Quais vantagens/desvantagens você vê em relação ao Mark-and-Sweep?
- Quando ambos começam a apresentar problemas de eficiência?
- Qual o principal efeito colateral benéfico de copy-collection?

# Custo

---

- Mark-and-Sweep

- $(c1.R + c2.H) / (H-R)$

- Copy-Collection

- $(c3.R) / (H/2 - R)$

# Generational Collection

---

- Observação empírica:
  - Objetos recém criados tendem a morrer logo
  - Objetos que sobreviveram a várias coletas, têm alta probabilidade de sobreviver várias outras coletas
- Coletor concentra esforços nos objetos mais jovens
- O heap é dividido em gerações G0, G1, ..., Gn
- A coleta é mais freqüente em G0

# Generational Collection

---

- Coleta de cada geração
  - Usar Mark-Sweep, Copy collection
- Para coletar G0
  - As raízes incluem
    - Variáveis locais
    - Apontadores vindo de gerações mais antigas
- Apontadores de objetos antigos para novos não são comuns!

# Generational Collection

---

- Buscar raízes nas outras gerações seria muito esforço
- Solução
  - Fazer o programa compilador lembrar onde existem ponteiros de objetos velhos para mais novos
- Existem várias alternativas:



# Generational Collection

---

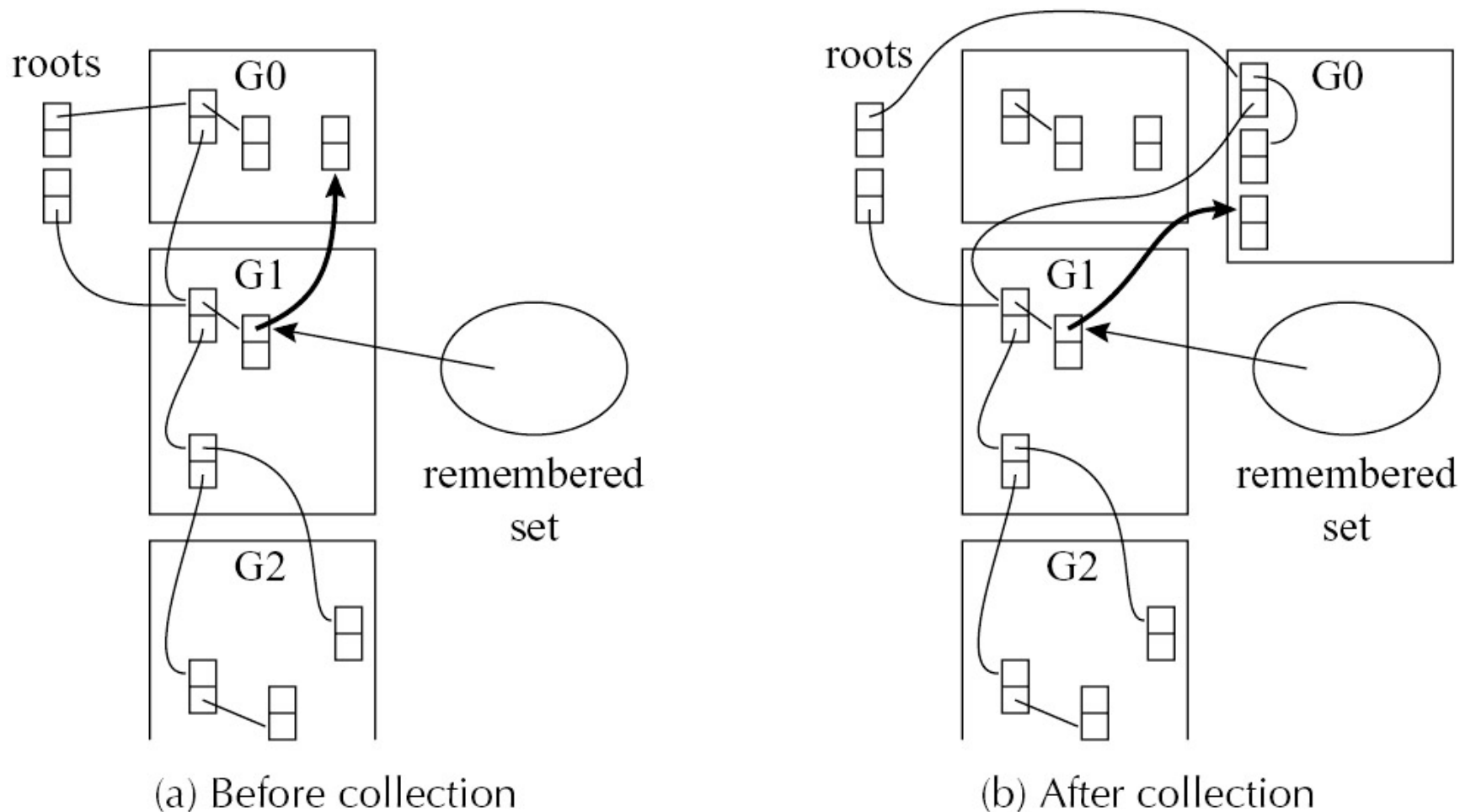
- Remembered List:

- $b.fi \leftarrow a$  coloca  $b$  em um vetor de objetos atualizados
- A cada GC, o vetor é percorrido buscando os  $b$ 's antigos que apontam para objetos em  $G_0$

- Remebered Set:

- O mesmo que o acima, porém o compilador usa um bit a mais nos objetos  $b$ 's para indicar que ele já está no vetor
- Evita duplicar entradas na lista
- Início do GC:
  - Esse conjunto é percorrido em busca de raízes apontando para objetos em  $G_0$

# Generational Collection



# Generational Collection

---

- Após várias coletas em G0:
  - Coletar G1
  - Mas para isso também devemos coletar G0 junto, pois deve conter vários apontadores para G1
  - Remembered set deve ser percorrido para buscar raízes em G2, G3, ...
- G2 será coletada menos freqüentemente ainda
  - E assim sucessivamente para G3, ...

# Generational Collection

---

- Gerações mais antigas devem ser maiores
- Um objeto passa de  $G_i$  para  $G_{i+1}$  quando sobreviver 2 ou 3 coletas de  $G_i$
- Custo:
  - Supondo  $G_0$  com 10% de dados vivos ( $H/R = 10$ ) e um coletor por cópia:  $c3 R / (10R - R)$
  - Para gerações mais antigas usamos  $H/R$  menores
  - Se existem muitos updates em apontadores, pode ser tornar mais cara do que coleta sem gerações

# Incremental Collection

---

- Principal objetivo:
  - Evitar longas interrupções na aplicação
- Coleta é feita de forma incremental
- Aplicado em programas de tempo real ou interativos
- Intercala o trabalho de GC com a execução da aplicação

# Incremental Collection

---

- Aplicação é chamada de *mutator*.
  - Altera o grafo do heap durante a coleta
- Contagem de referências é naturalmente incremental
- Mark-sweep e Copy-collection também podem se tornar incrementais

# Marcação por Três Cores

---

- 3 listas de objetos
  - Brancos:
    - Não alcançados pelo GC
  - Cinzas:
    - Visitados, mas cujos campos ainda não foram percorridos
  - Pretos:
    - Visitados, assim como seus descendentes imediatos.

# Marcação por Três Cores

---

- Todos os objetos iniciam como brancos
- Algoritmo:

```
while there are any grey objects
  select a grey record p
  for each field fi of p
    if record p. fi is white
      color record p. fi grey
  color record p black
```



# Marcação por Três Cores

---

- Ao final:
  - O algoritmo termina quando não houver mais objetos cinzas
  - Todos os objetos vivos devem estar pretos
  - Os brancos que restam devem ser recuperados
- Esse algoritmo pode ser usado com Mark-sweep ou Copy collection
- Invariantes mantidas:
  - Nenhum obj preto aponta para um branco
  - Os cinzas estão na estrutura de dados do GC, pilha ou fila

# Marcação por Três Cores

---

- Mutator pode alterar o grafo durante a coleta
  - Isto pode quebrar uma das invariantes
- Técnicas para preservá-las e permitir que o mutator altere o grafo
  - Write e read barriers
  - Dijkstra: Se o mutator armazena ou altera um ponteiro em um objeto preto, este se torna cinza

# Marcação por Três Cores

---

- **Write-barriers:**

- Dijkstra: Se o mutator armazena ou altera um ponteiro em um objeto preto, este se torna cinza
- Steele: Se o mutator armazena um ponteiro para um objeto branco a em um objeto preto b, b se torna cinza

- **Read-barrier:**

- Baker: Sempre que o mutator ler um apontador para um objeto branco, este se torna cinza. O mutator nunca estará manipulando um objeto branco

# Algoritmo de Baker

---

- Baseado em Copy-Collection de Cheney
- Usa read-barrier
- A cópia é intercalada com execução do mutator
- Início atômico:
  - Inversão dos espaços to e from-space
  - Forwarding das raízes

# Algoritmo de Baker

---

- **Invariante:**
  - Mutator sempre usa apontadores para o to-space
- **Principal overhead:**
  - Instruções para cada acesso a um apontador
  - Checar se o objeto ainda está no from-space
  - Eventual cópia
- **Conservativo**
  - Todo objeto alocado durante um ciclo de coleta está vivo em todo o ciclo