Tópicos selecionados de programação em Java

Gerência de Memória em Java

Parte I: Arquitetura da JVM e algoritmos de coleta de lixo

Helder da Rocha

Setembro 2005

Por que gerenciar memória?

- Há linguagens em que a alocação de memória é trivial, e não requerem gerenciamento complexo
- Estratégias de alocação de memória
  - Estática: áreas de memória são alocadas antes do início do programa; não permite mudanças nas estruturas de dados em tempo de execução (ex: Fortran)
  - Linear: memória alocada em fila ou em pilha; não permite remoção de objetos fora da ordem de criação (ex: Forth)
  - Dinâmica: permite liberdade de criação e remoção em ordem arbitrária; requer gerência complexa do espaço ocupado e identificação dos espaços livres (ex: Java, C++)
- Java utiliza alocação dinâmica (heap) para objetos e alocação linear (pilha) para procedimentos seqüenciais
  - Mas todo o gerenciamento é feito automaticamente

Gerencia de memória? Em Java?

- Então, por que se preocupar com memória em Java?
  - Diferentemente de C ou C++, programadores Java não têm a responsabilidade e nem a possibilidade de gerenciar a memória do sistema explicitamente
  - Programação em alto-nível: alocação e liberação de memória dinâmica é realizada automaticamente usando algoritmos: programador preocupa-se apenas com a lógica do programa
- Mas algoritmos são configurados para situações típicas
  - Determinadas aplicações podem requerer ajustes (tuning): performance, escalabilidade, segurança, *throughput vs. liveness*
  - Saber o quanto, quando, onde ajustar requer conhecimentos elementares da organização da memória e dos algoritmos de coleta de lixo empregados pela implementação da JVM usada

Assuntos abordados

- Este minicurso explora detalhes sobre o uso de memória virtual em aplicações Java
- Está dividido em três partes
  - Parte I: este módulo (arquitetura da JVM, alocação de memória e algoritmos de coleta de lixo)
  - Parte II: arquitetura da HotSpot JVM e estratégias de ajuste e otimização de performance
  - Parte III: finalização, controle do coletor de lixo, memory leaks e objetos de referência
- Tópicos deste módulo (primeira parte)
  1. Anatomia da JVM
  2. Coleta de lixo: algoritmos elementares
  3. Coleta de lixo: algoritmos combinados (estratégias)
  4. Coleta de lixo em paralelo (coletores não-seriais)

1. Anatomia da JVM

- A máquina virtual Java (JVM) é uma máquina imaginária implementada como uma aplicação de software [JVMS]
  - Ela executa um código de máquina portável (chamado de Java bytecode) armazenado em um formato class
  - O formato class geralmente é gerado como resultado de uma compilação de código-fonte Java

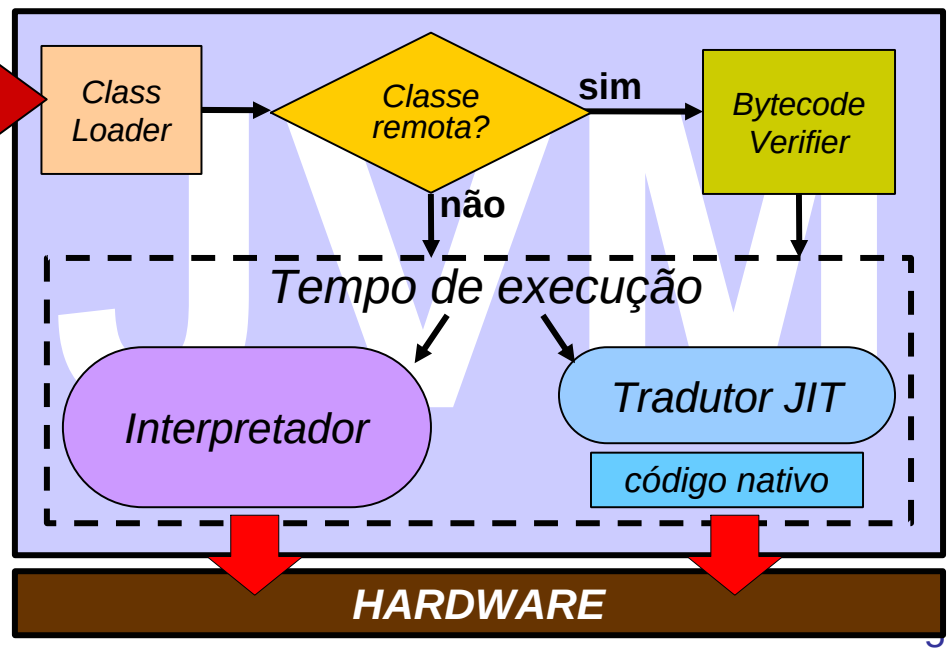
javac

```
F4 D9 00 03 0A B2
FE FF FF 09 02 01
01 2E 2F 30 62 84
3D 29 3A C1 00 00
```

HelloWorld.class

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

java



***Tradutor JIT**: Just-In-Time Compiler. Gera código nativo a partir de bytecodes*

Java esconde detalhes da memória

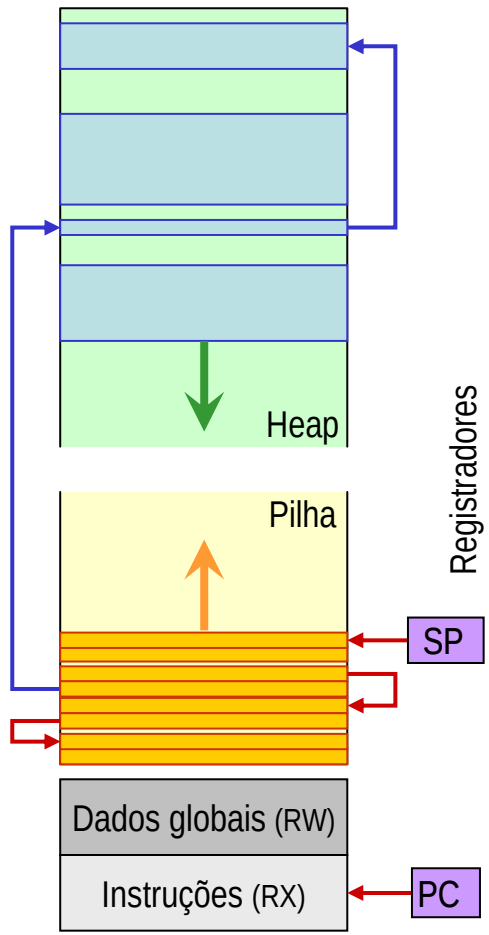
- A especificação da máquina virtual (Java Virtual Machine Specification [JVMS]) não determina:
  - Detalhes de segmentação da memória (como ocorre o uso de memória virtual, onde fica a pilha, o heap, etc.)
  - O algoritmo de coleta de lixo usado para liberar memória (diz apenas que deve haver um)
  - Outros aspectos de baixo nível como formato de tipos, etc.
- Diferentes implementações da JVM têm a liberdade de organizar a memória diferentemente e escolher algoritmos de coleta de lixo diferentes
  - Sun HotSpot JVM (mais popular; a da IBM é similar)
  - Jikes RVM (experimental; popular entre cientistas)

A pilha, o heap e a JVM

- Do ponto de vista de um programador Java, as áreas de memória virtual conhecidas como a pilha e o heap são lugares imaginários na memória de um computador
  - Não interessa nem adianta saber onde estão
  - Java não oferece opções de escolha: “tipos primitivos ficam na pilha e objetos ficam no heap e ponto final!”
- Implementações da especificação da JVM, (como a HotSpot JVM), oferecem parâmetros que permitem algum controle sobre a gerência de memória
  - Conhecer as escolhas de algoritmos e arquitetura da máquina virtual usada é importante para saber como configurá-la e quais parâmetros ajustar para obter melhor performance
  - Ainda assim, o controle é limitado e não existe, em Java, a disciplina “gerência de memória” como existe em C ou C++

Segmentação de memória virtual: esquema lógico de baixo nível

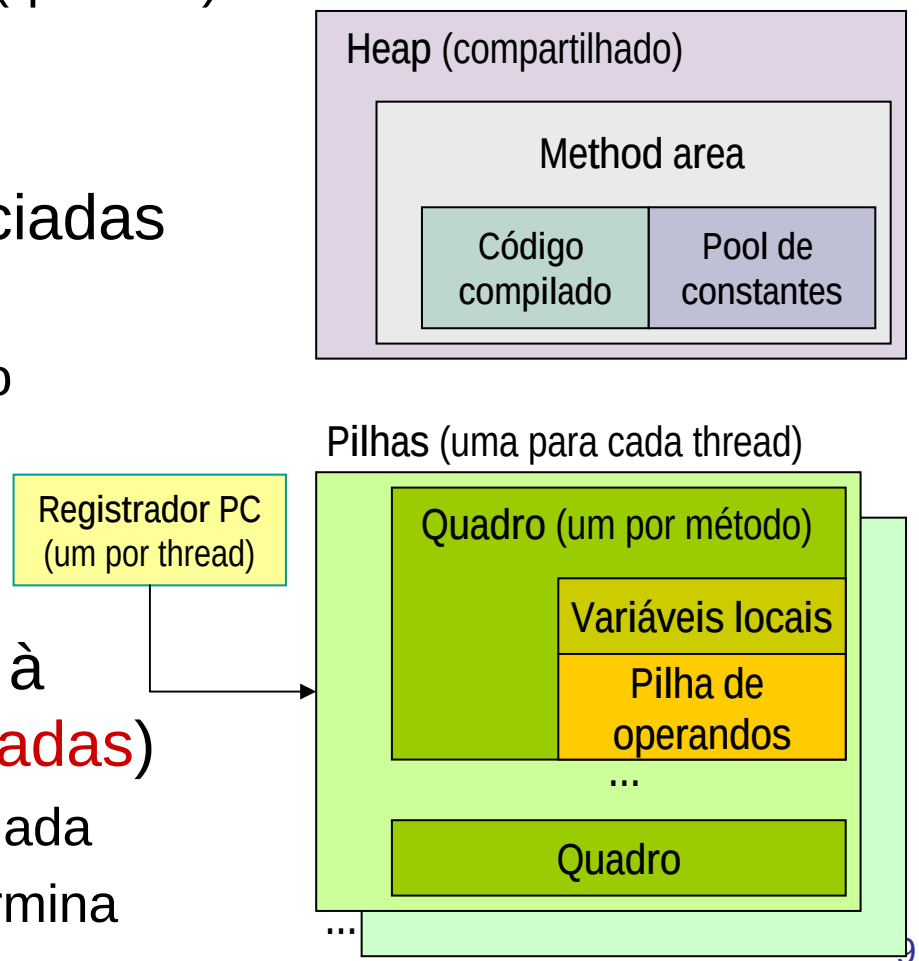
- O diagrama ao lado é apenas um modelo genérico* e não reflete *nenhuma* implementação real
  - Os blocos azuis no heap indicam memória alocada dinamicamente
  - Os blocos amarelos na pilha podem indicar *frames* (seqüências de instruções de cada método) de *um único thread*
  - As setas azuis indicam ponteiros



** Modelo inspirado em segmentação C/C++. A especificação da JVM descreve a memória em nível mais alto (ex: não sugere onde o heap ou pilha devem ser localizados, não descreve organização do heap, permite pilha não contígua, etc.)*

Anatomia da JVM: áreas de dados

- A máquina virtual define várias áreas de dados que podem ser usadas durante a execução de um programa
  - Pilhas e segmentos de pilha (quadros)
  - Heaps e área de métodos
  - Registradores
- Algumas áreas estão associadas a *threads* (privativas)
  - São criadas/alocadas quando um *thread* novo é criado
  - São destruídas/liberadas quando o *thread* termina
- Outras áreas estão ligadas à máquina virtual (compartilhadas)
  - Criadas quando a JVM é iniciada
  - Destruídas quando a JVM termina

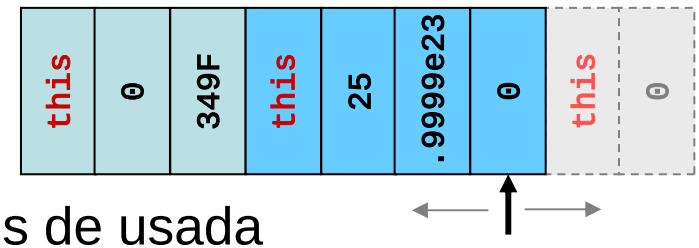


O registrador PC

- Cada *thread* de execução tem um registrador PC (program counter)
- Em qualquer momento, cada *thread* estará executando o código de um único método
  - Um método (Java ou bytecode) consiste de uma lista de instruções executadas em uma seqüência definida
- O registrador PC contém o endereço da instrução da JVM que está sendo executada
  - O valor do registrador PC só não é definido se o método for um método nativo (implementado em linguagem de máquina da plataforma onde roda)

Pilhas da JVM

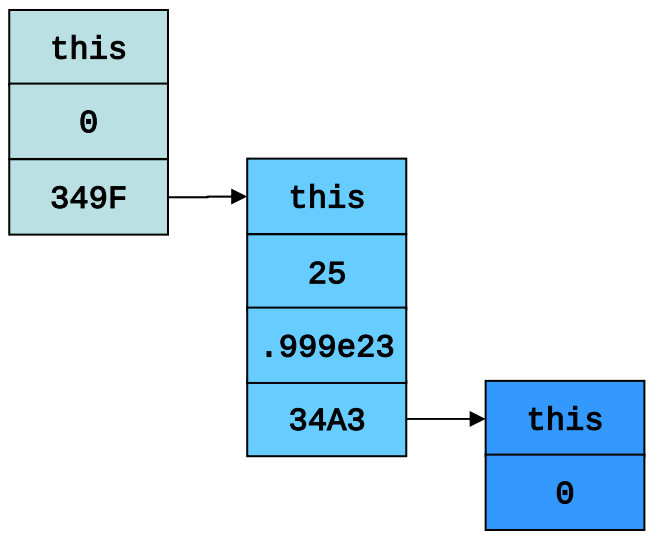
- Cada thread é criado com *uma* pilha associada
  - É usada para guardar variáveis locais e resultados parciais
- A memória usada pela pilha
  - *Pode* ser alocada no heap
  - Não precisa ser contígua
  - É liberada automaticamente depois de usada
  - Pode ter um tamanho fixo ou expandir-se e contrair-se na medida em que for necessário (implementações de JVM podem oferecer controles para ajustar tamanho de pilhas)



- Quando a memória acaba, dois erros podem ocorrer
  - StackOverflowError: se a computação de um thread precisar de uma pilha maior que a permitida (métodos que criam muitas variáveis locais, funções recursivas)
  - OutOfMemoryError: se não houver memória suficiente para expandir uma pilha que pode crescer dinamicamente.

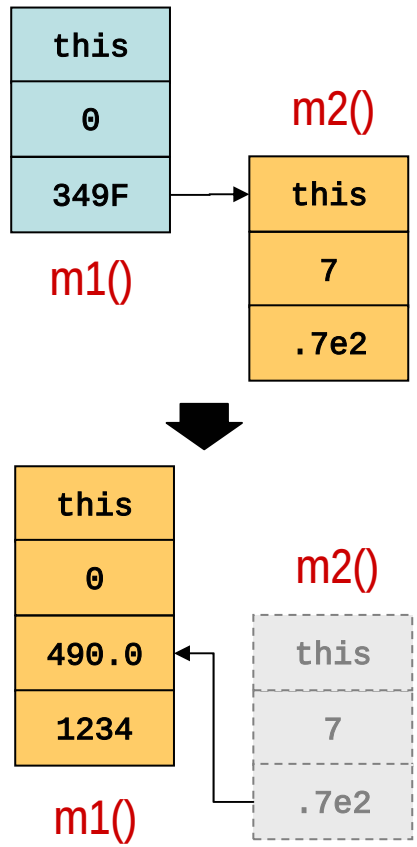
Quadros de pilha (frames)

- Um quadro (frame) é um segmento alocado a partir da pilha de um *thread*, sempre que um método é chamado
  - É criado cada vez que um método é chamado (todo método tem um quadro associado)
  - É destruído quando a chamada termina (normalmente ou não)
  - É sempre local ao thread (outros threads não têm acesso a ele)
- É usado para
  - guardar resultados parciais e dados
  - realizar ligação dinâmica
  - retornar valores de métodos
  - despachar exceções
- Cada quadro possui
  - Um array de variáveis locais
  - Uma pilha de operandos
  - Referência ao pool de constantes de runtime da classe corrente.



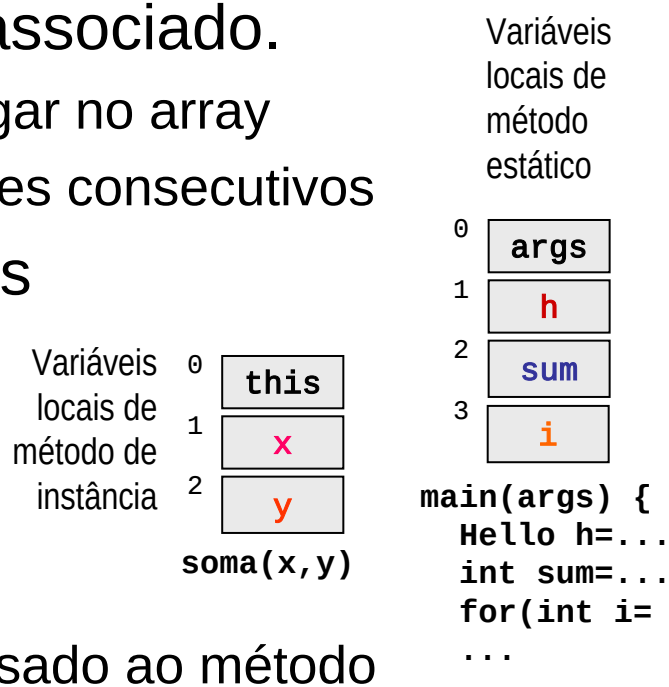
Quadros correntes e chamadas

- Em um determinado *thread*, apenas um quadro está ativo em um determinado momento
  - Chama-se quadro corrente
  - Seu método é o método corrente
  - Sua classe é a classe corrente
- Quando o método corrente m1, associado ao quadro q1, chama outro método m2
  - Um novo quadro q2 é criado (que passa a ser o quadro corrente)
  - Quando o método m2 retornar, o quadro q2 retorna o resultado da sua chamada (se houver) ao quadro q1
  - O quadro q2 é descartado e q1 volta a ser o quadro corrente



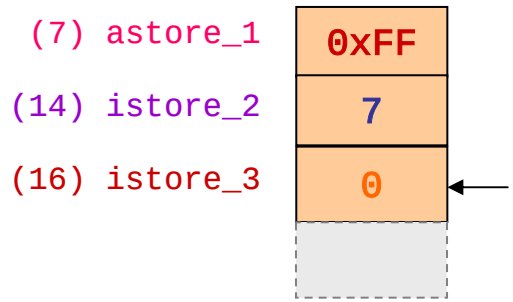
Variáveis locais

- Cada frame possui um vetor de variáveis contendo as variáveis locais do seu método associado.
  - Variáveis de até 32 bits ocupam um lugar no array
  - Variáveis de 64 bits ocupam dois lugares consecutivos
- São usadas para passar parâmetros durante a chamada de métodos
- Variáveis locais são acessadas pelo seu índice (a partir de 0)
  - Em métodos estáticos, a variável local de índice 0 é o primeiro parâmetro passado ao método
  - Em métodos de instância, a variável local de índice 0 sempre contém a referência this; os parâmetros são passados a partir da variável local de índice 1



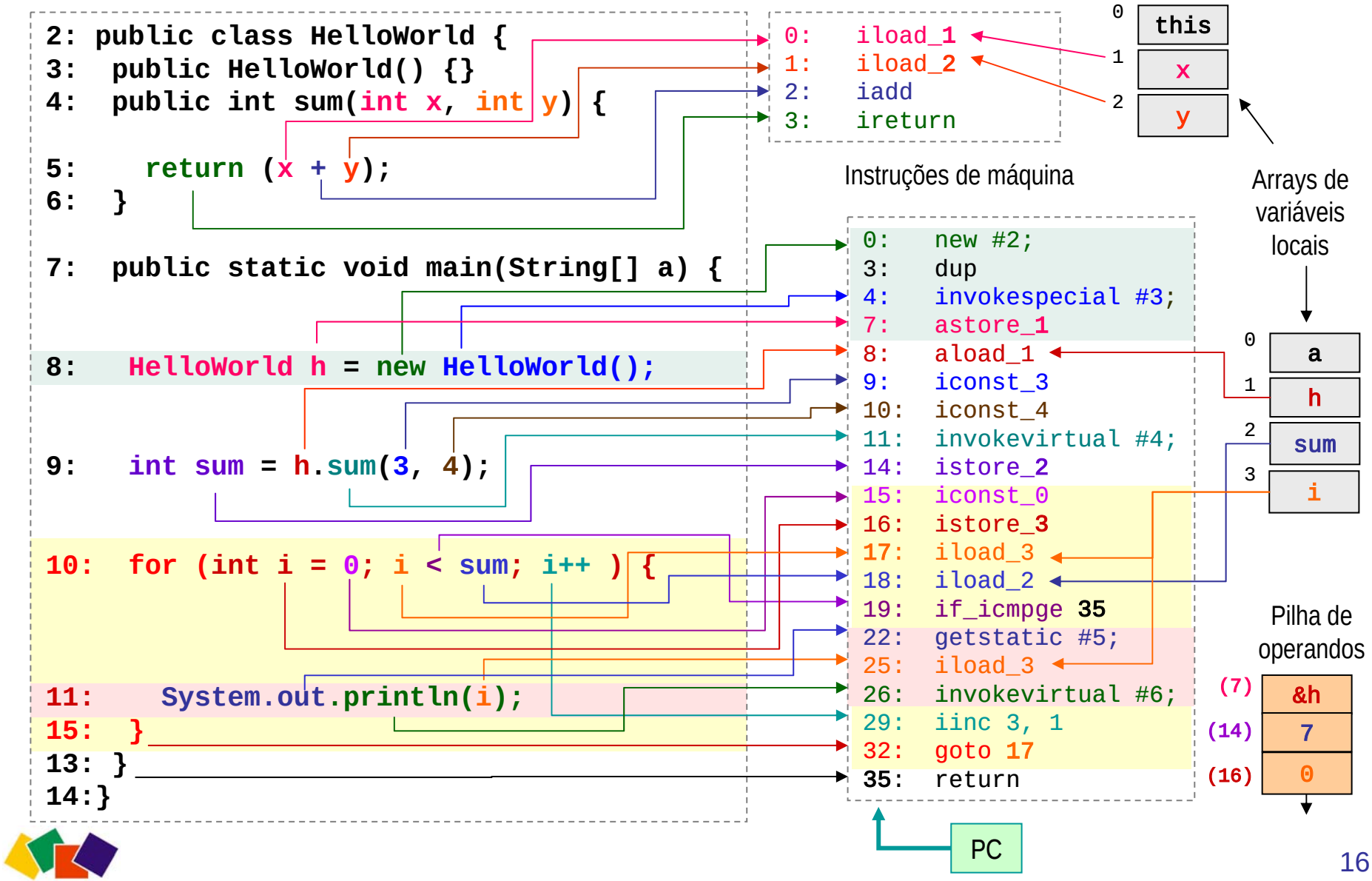
Pilha de operandos

- Cada quadro contém uma pilha LIFO conhecida como Pilha de Operandos
  - Quando o quadro é criado, a pilha é vazia
  - Instruções da máquina virtual carregam constantes ou valores de variáveis locais ou campos de dados para a pilha de operandos, e vice-versa
  - A pilha de operandos também serve para preparar parâmetros a serem passados a métodos e para receber seus resultados



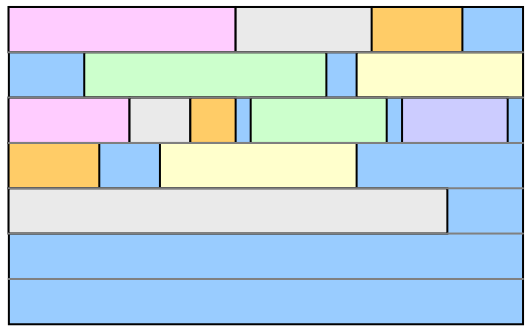
- Qualquer tipo primitivo pode ser armazenado e lido da pilha de operandos.
  - Tipos long e double ocupam duas unidades da pilha.
- Operações sobre a pilha de operandos respeitam os tipos dos dados guardados

Código Java e instruções de pilha



O heap

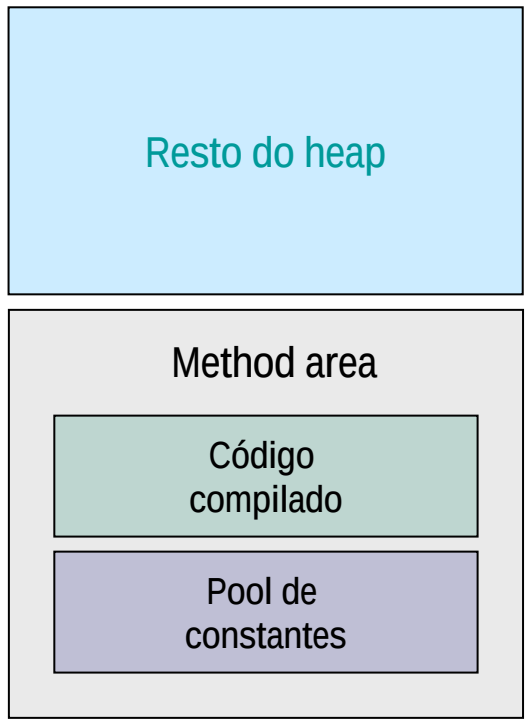
- O heap é a área de dados onde todas as instâncias e vetores são alocados
  - Compartilhada por todos os *threads*
  - Criada quando a máquina virtual é iniciada
  - Não precisa ser uma área contígua



- O espaço de objetos no heap é reciclado por um sistema de gerenciamento de memória (coletor de lixo)
  - Algoritmo depende da implementação da JVM
- Pode ter tamanho fixo ou ser expandido/contraído
  - Implementações podem oferecer controles para ajustar tamanho do mínimo/máximo ou fixo do heap
- Se um programa precisar de mais heap que o que foi disponibilizado, a JVM causará OutOfMemoryError.

Área de métodos

- Parte do heap usada para guardar código compilado de métodos e construtores
  - Criada quando a máquina virtual inicia
  - Geralmente armazenada em uma área de alocação permanente (a especificação não determina a localização)
  - Compartilhada por todos os *threads*
- Guarda estruturas compartilhadas por todos os métodos de uma classe como
  - Pool de constantes de *runtime* (constantes de diversos tipos usados pelo método)
  - Dados usados em campos e métodos
- OutOfMemoryError ocorre se em algum momento não houver mais espaço para armazenar código de métodos



Ferramenta **javap**

- Para obter informações sobre a estrutura de uma classe e instruções da JVM usadas use **`javap -c Classe`**
- A ferramenta javap permite visualizar o conteúdo de um arquivo de classe

  `javap [-opções] classe`
- Usando opções **`-c`** e **`-verbose`** é possível ver
  - Seqüência de instruções da JVM
  - Tamanho dos quadros de cada método
  - Conteúdo dos quadros
  - Pool de constantes
- Opção **`-l`** imprime tabela de variáveis locais
- Sem opções, mostra a interface da classe

2. Algoritmos de coleta de lixo

- Dados armazenados na pilha são automaticamente liberados, quando a pilha esvazia para ser reutilizada
- Mas dados armazenados no heap precisam ser reciclados através de liberação...
  - ... manual, em linguagens como C e C++ (freelists, delete, free)
  - ... automática, como em Java e maior parte das linguagens*
- Liberação automática de memória do heap é realizada através de algoritmos de coleta de lixo
  - Há várias estratégias, com vantagens e desvantagens de acordo com a taxa em que objetos são criados e descartados
  - Têm considerável impacto na performance (gerência explícita de memória também tem, e é muito mais complicada)

* Perl, Python, Rubi, Basic, LISP, Algol, Dylan, Prolog, PostScript, Scheme, Simula, Smalltalk, ML e Modula-3 (na maior parte – suporta controle manual em alguns módulos)

Coleta de lixo

- De acordo com a especificação da linguagem Java, uma JVM precisa incluir um algoritmo de coleta de lixo para reciclar memória não utilizada (objetos inalcançáveis)
  - A especificação não informa qual algoritmo deve ser usado – apenas que deve existir um
  - A máquina virtual HotSpot, da Sun, usa vários algoritmos e permite diversos níveis de ajuste*
- O comportamento desse algoritmo é o principal gargalo na maior parte das aplicações de vida longa
  - Em média: 2% a 20% do tempo de execução de uma aplicação típica é gasto com coleta de lixo
  - Conhecer os detalhes do funcionamento do algoritmo de coleta de lixo é importante para saber como ajustá-lo para melhorar a eficiência de um sistema

** Novos algoritmos são suportados a cada nova versão*

Desafios da coleta de lixo

- Distinguir o que é lixo do que não é lixo
  - Algoritmos exatos garantem a identificação precisa de todos os ponteiros e a coleta de todo o lixo
  - Algoritmos conservadores faz suposições e pode deixar de coletar lixo que pode não ser lixo (possível memory leak)
  - Todos os coletores usados nas JVMs atuais são exatos
- Influência da coleta em alocações posteriores
  - A remoção de objetos deixa buracos no heap (fragmentação)
  - Para alocar novos objetos, é preciso procurar nas listas de espaços vazios (free lists) um que caiba o próximo objeto: alocação será mais demorada e uso do espaço é ineficiente.
  - Alguns algoritmos compactam o heap, movendo os objetos para o início do heap e atualizando os ponteiros: torna a alocação mais simples e eficiente, porém são mais complexos.

Algoritmos para coleta de lixo

- Contagem de referências
  - Reference counting algorithm: mantém, em cada objeto, uma contagem das referências para ele; coleta os objetos que têm contagem zero
  - Cycle collecting algorithm: extensão do algoritmo de contagem de referência para coletar ciclos (referências circulares)
- Rastreamento de memória
  - Mark and sweep algorithm: rastreia objetos do heap, marca o que não é lixo e depois varre o lixo (libera memória)
  - Mark and compact algorithm: extensão do algoritmo Mark and Sweep que mantém o heap desfragmentado
  - Copying algorithm: divide o heap em duas partes; cria objetos em uma parte do heap e deixa outra parte vazia; recolhe o que não é lixo e copia para a área limpa, depois esvazia a área suja

Estratégias de coleta de lixo

- Algoritmos combinados
  - Estratégias usam ou baseiam-se em um ou mais dos algoritmos citados para obter melhores resultados em um dado cenário
- Classificação quanto à organização da memória
  - Generational (objetos transferidos para áreas de memória diferentes conforme sobrevivem a várias coletas de lixo)
  - Age-oriented (algoritmos diferentes conforme idade dos objetos)
- Classificação quanto ao nível de paralelismo
  - Serial
  - Incremental
  - Concorrente
- Principal argumento de escolha: eficiência vs. pausas

Metas ajustáveis

- Eficiência (*throughput*): tempo útil / tempo de faxina
  - Taxa entre o tempo em que uma aplicação passa fazendo sua função útil dividido pelo tempo que passa fazendo coleta de lixo
  - Ideal que seja a maior possível
- Pausas: quando o mundo pára (stop-the-world)
  - Tempo em que a aplicação inteira (todos os *threads*) da aplicação param para roda o GC e liberar memória
  - Ideal que seja o menor possível
- Como alcançar essas metas?
  - Escolhendo um coletor de lixo adequado (diferentes estratégias usam algoritmos diferentes, de formas diferentes)
  - Configurando parâmetros que modificam o espaço usado, influenciando a forma como o coletor gerencia memória

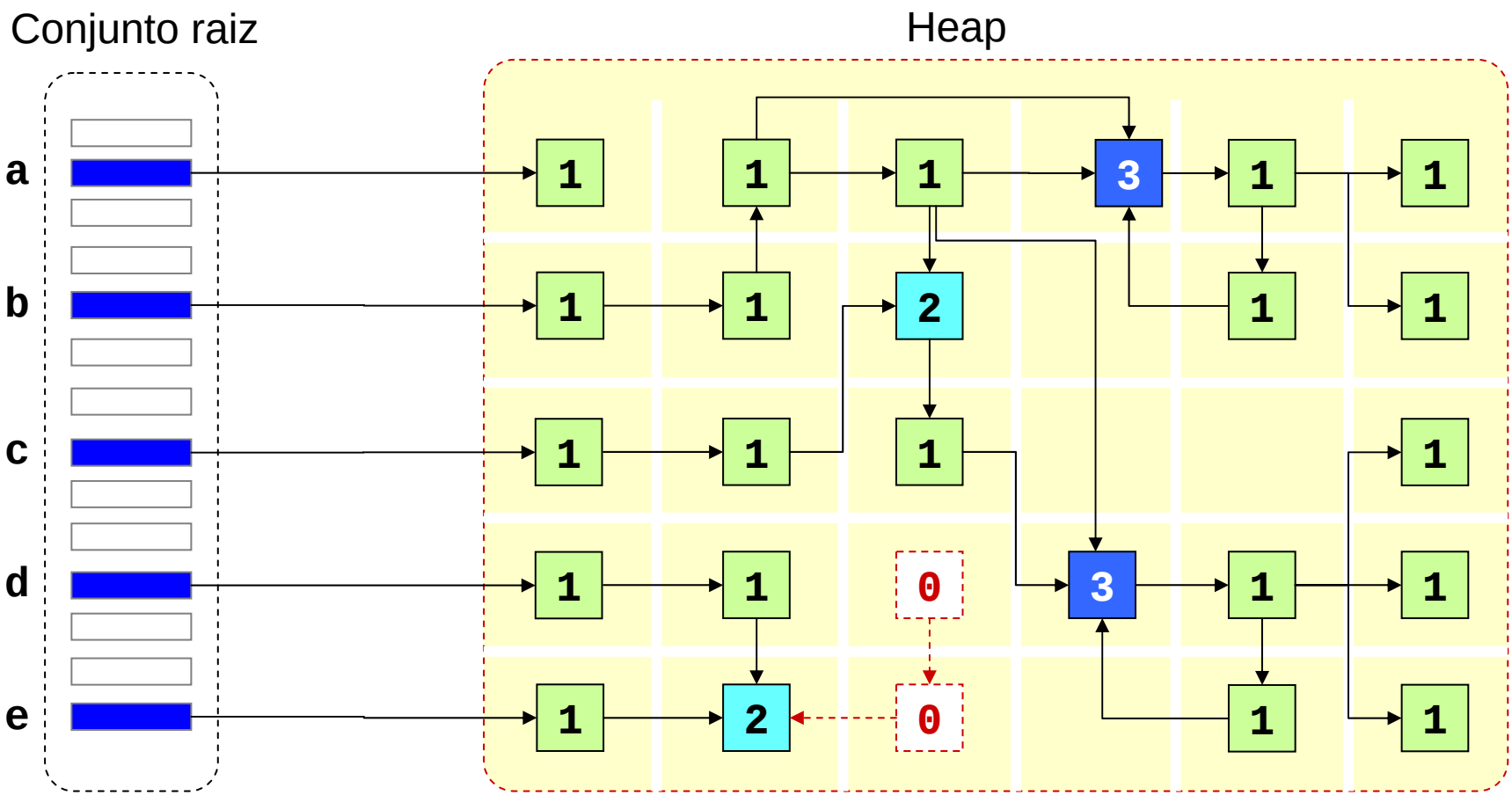
Contagem de referências (RC*)

- É o algoritmo mais simples [Collins 1960]
- Cada objeto possui um campo extra que conta quantas referências apontam para ele
  - Compilador precisa gerar código para atualizar esse campo sempre que uma referência for modificada
- Funcionamento:
  - Objeto criado em thread ativo: contagem = 1
  - Objeto ganha nova referência para ele (atribuição ou chamada de método com passagem de referência): contagem++
  - Uma das referências do objeto é perdida (saiu do escopo onde foi definida, ganhou novo valor por atribuição, foi atribuída a *null* ou objeto que a continha foi coletado): contagem--
  - Se contagem cair a zero, o objeto é considerado lixo e pode ser coletado a qualquer momento

**RC = Reference Counting*

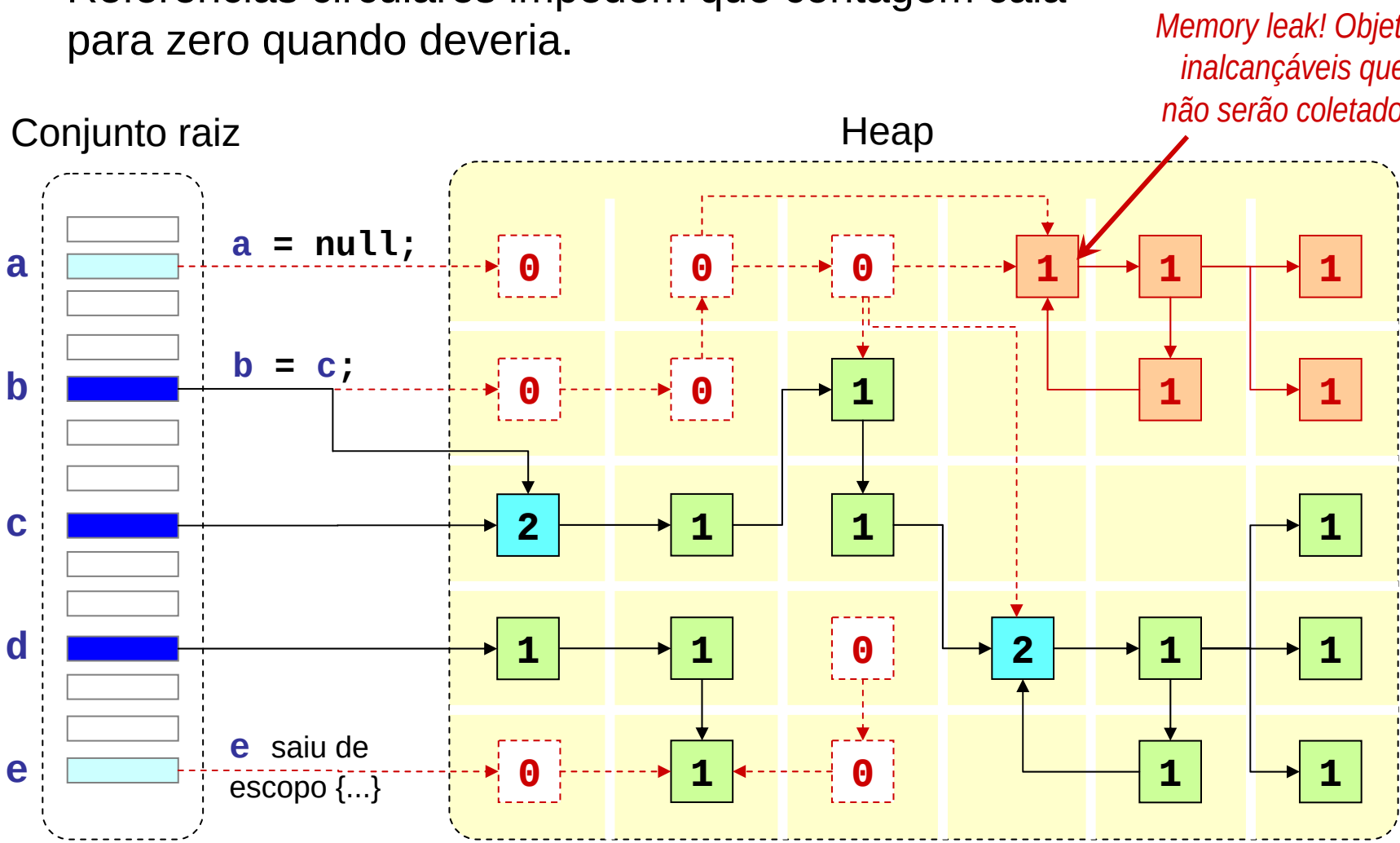
Contagem de referências (1)

- Cada seta que chega em um objeto é contada como uma referência para ele (independente de onde tenha vindo)



Contagem de referências (2)

- Referências circulares impedem que contagem caia para zero quando deveria.



RC: considerações

- Vantagens
  - Rápido: não precisa varrer o heap inteiro (só espaço ocupado)
  - Pode executar em paralelo com a aplicação
- Desvantagens do RC clássico
  - *Overhead* alto
  - Paralelismo caro
  - Incapacidade de recuperar ciclos (objetos que mantém referências circulares entre si)
- Soluções
  - Coletores incrementais RC on-the-fly [Levanoni-Petrank 2001] reduzem overhead e custo do paralelismo, sem pausas
  - Algoritmo eficiente de Coleta de Ciclos [Paz-Petrank 2003] ou backup com algoritmo de rastreamento (MS ou MC)

Coleta de ciclos (CC)

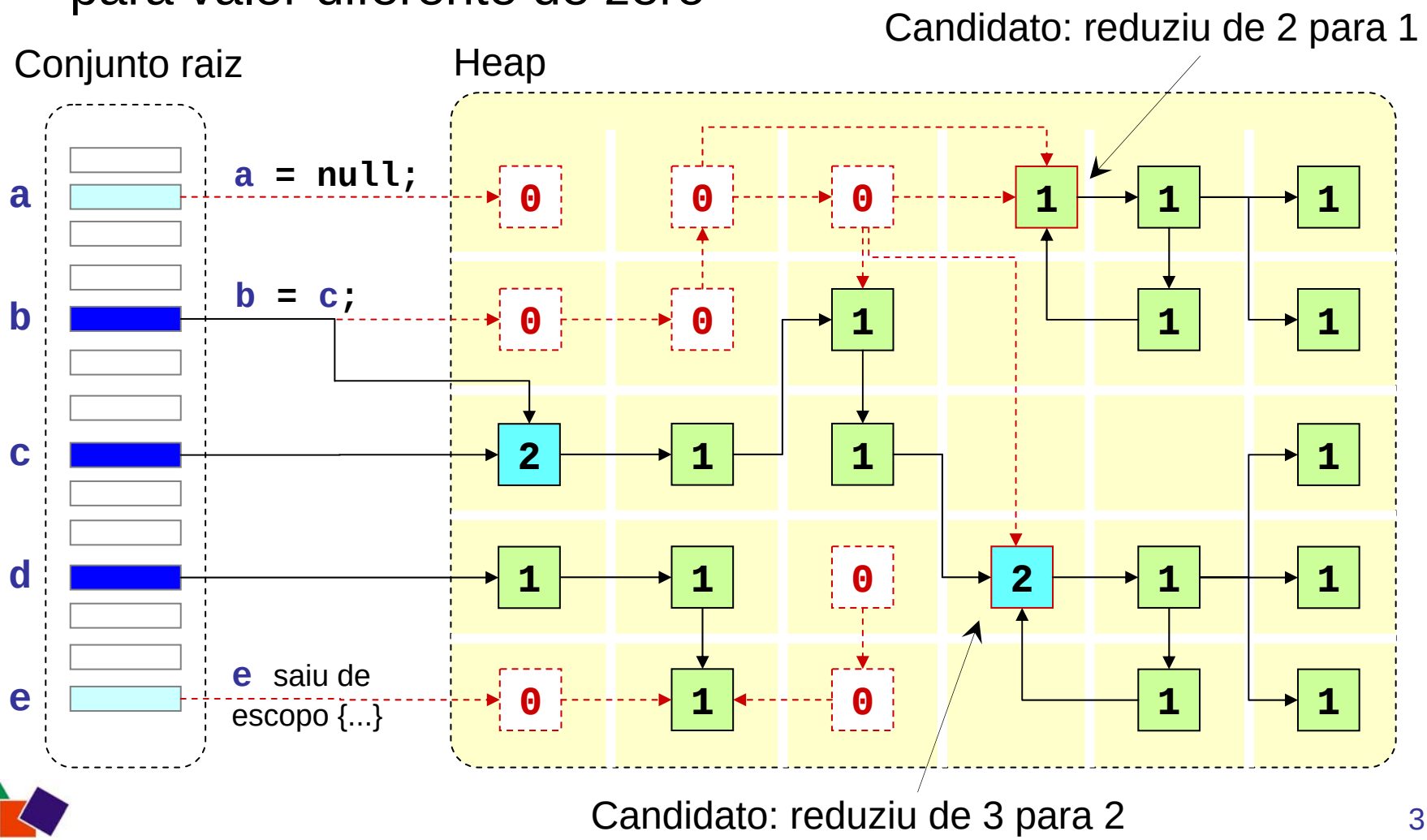
- Contagem de referências clássica não recolhe ciclos
- Coleção de ciclos baseia-se em duas observações

  Obs1: Ciclos-lixo só podem ser criados quando uma contagem cai para valor diferente de zero

  Obs2: Em ciclos-lixo, toda a contagem é devido a ponteiros internos
- Portanto, objetos que tem contagem decrementada para valores diferente de zero são candidatos (obs1)
- Algoritmo realiza três passos locais nos candidatos
  1. Mark: marca apenas objetos que têm ponteiros externos (obs2)
  2. Scan: varre o ciclo a partir do objeto candidato com ponteiro externo e restaura a marcação de objetos que forem alcançáveis
  3. Collect: coleta os nós cuja contagem for zero

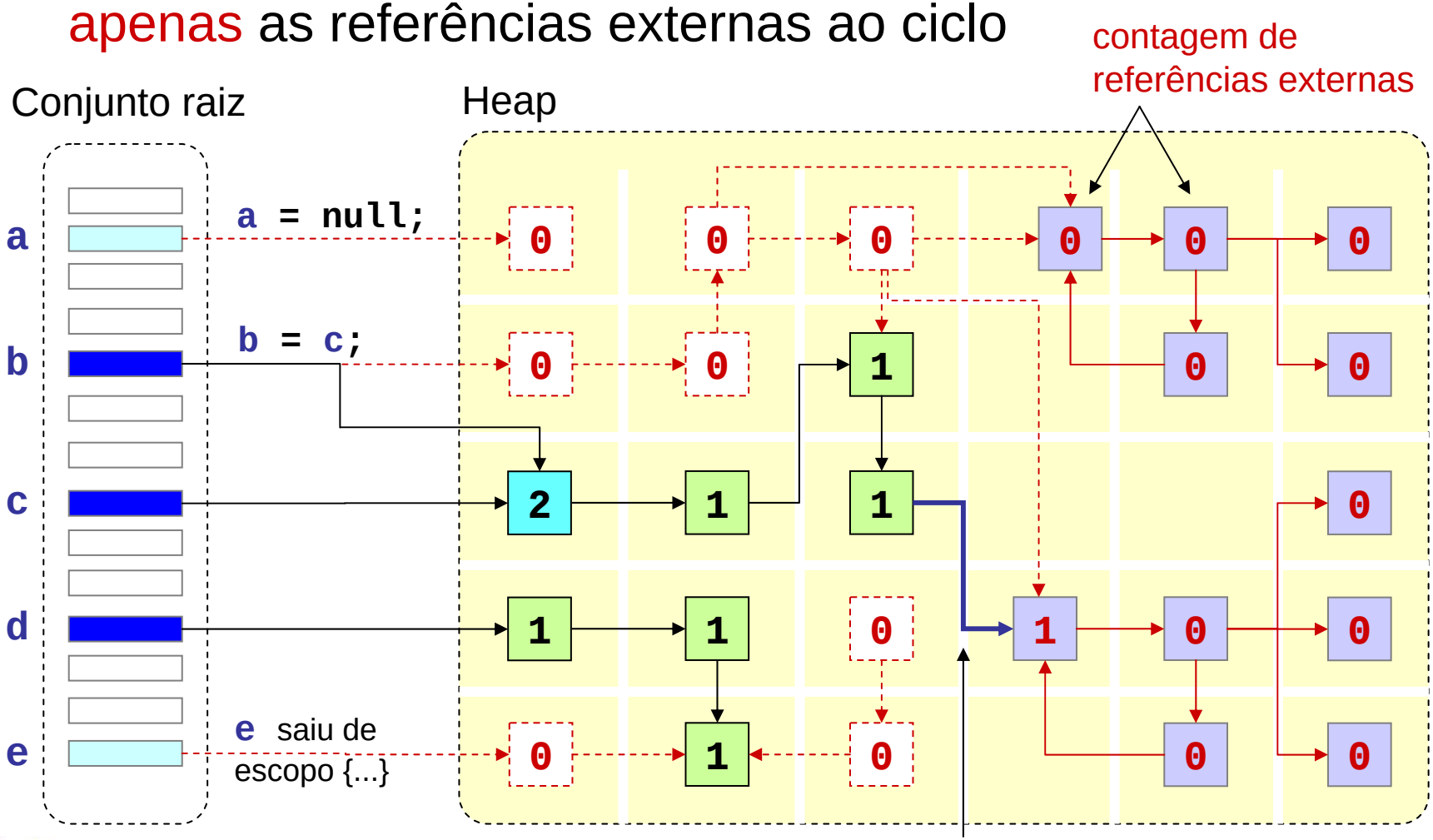
CC: identifica candidatos à remoção

- Candidatos: objetos cuja contagem foi decrementada para valor diferente de zero



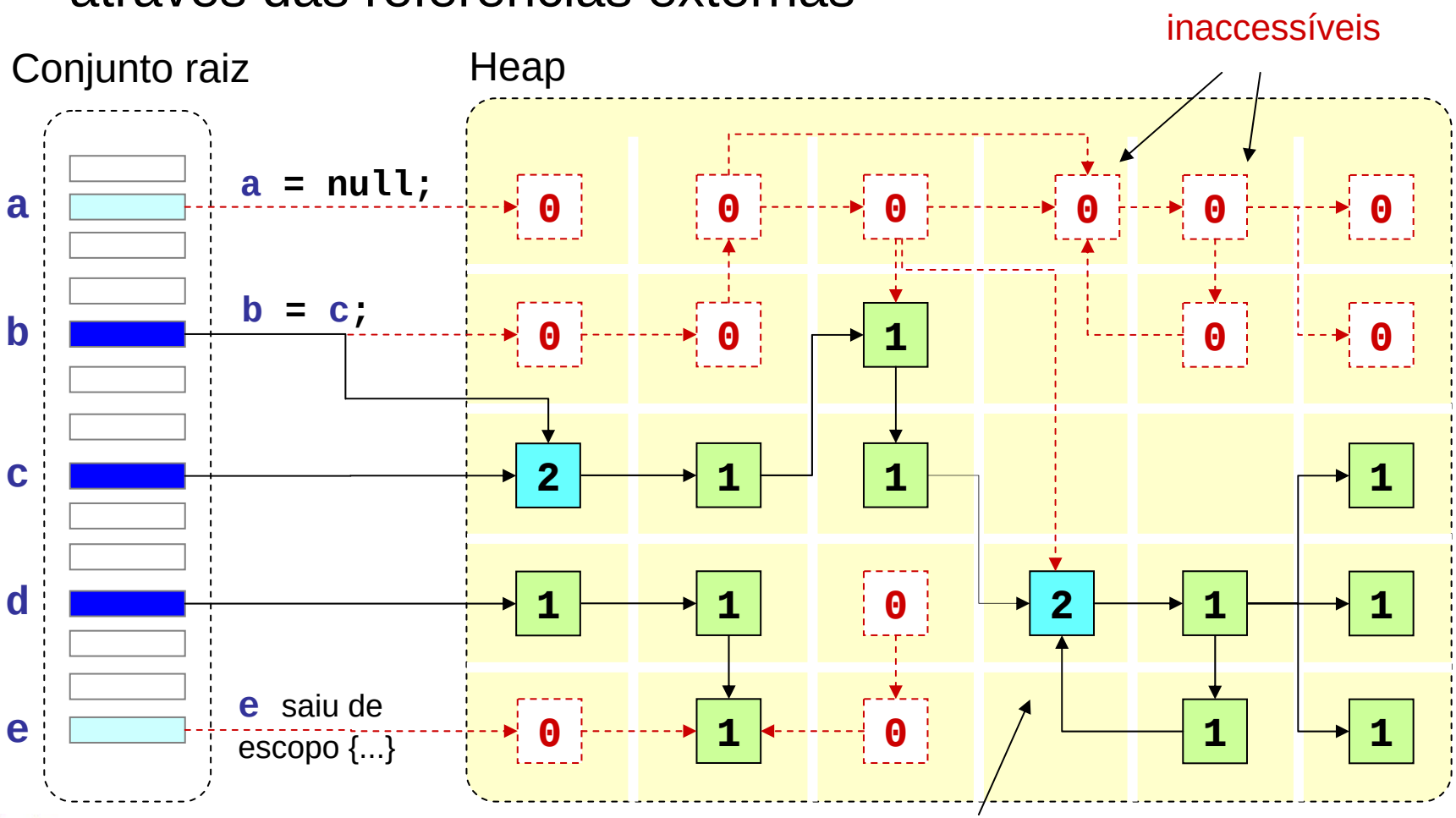
CC: passo 1 - mark

- Navega nas referências a partir do candidato e conta apenas as referências externas ao ciclo



CC: passo 2 - scan

- Restaura contagem dos nós que forem acessíveis através das referências externas

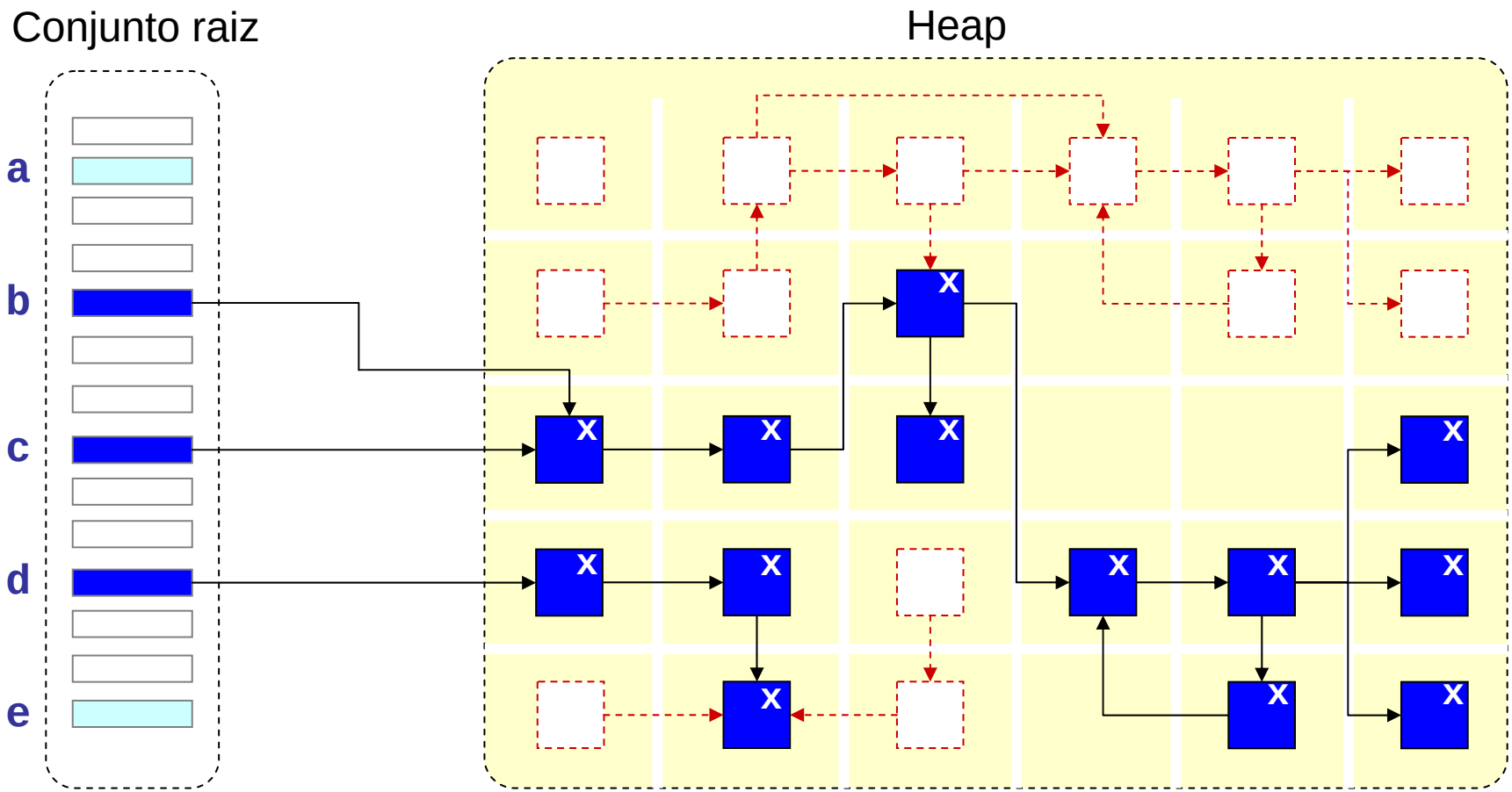


CC: considerações

- Vantagens
  - Trabalha com objetos ativos (não precisa pesquisar todo o heap)
  - Pode trabalhar em paralelo com a aplicação (não pára tudo)
- Desvantagens
  - Ineficiente se houver muitos ciclos (precisa passar três vezes por cada um deles)
  - Quando trabalha em paralelo, precisa garantir atomicidade das etapas de coleta de ciclos (complexo em multiprocessadores)
- Soluções que usam CC
  - *"Efficient age-oriented concurrent cycle collector"* [Paz et al. 2005]
- Uso experimental apenas (Jikes RVM)
  - A JVM HotSpot não tem, até o momento, empregado nenhum tipo de algoritmo de contagem de referências (até Java 5.0)

Algoritmos de rastreamento*

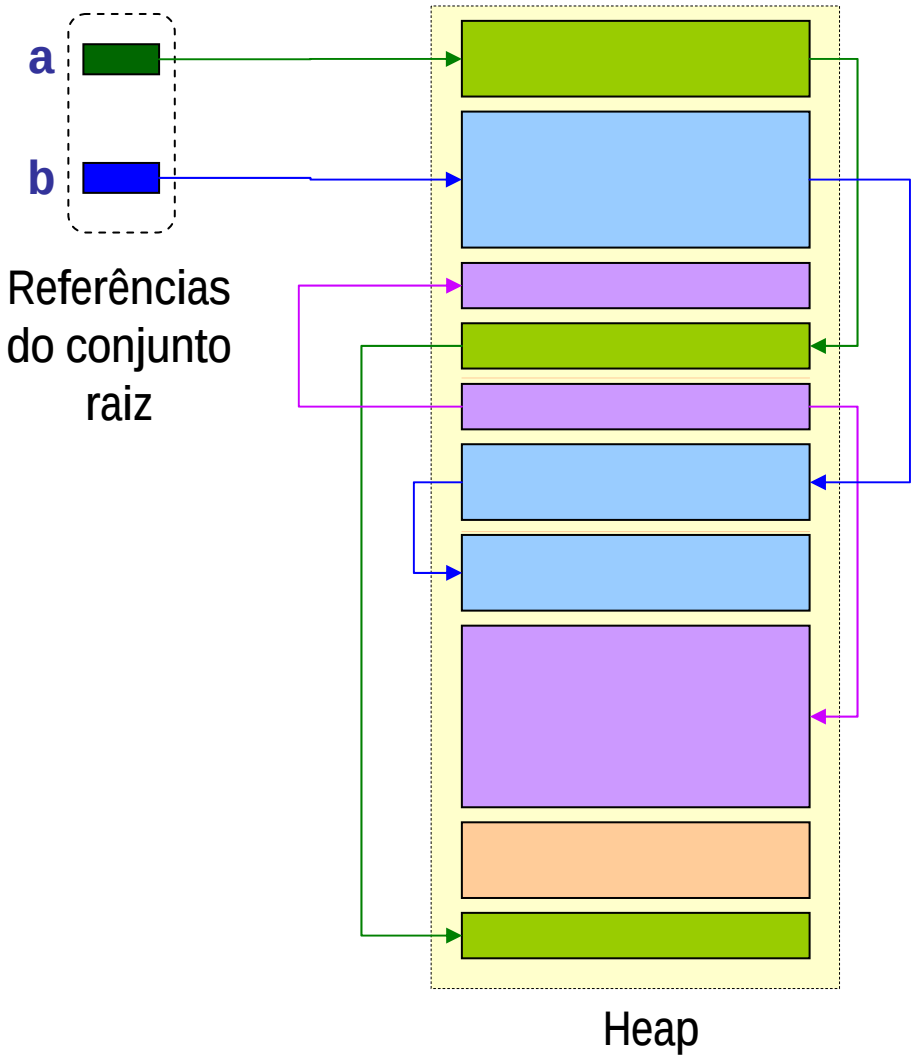
- Marcam as referências que são alcançáveis (navegando a partir do conjunto raiz), e remove todas as que sobrarem



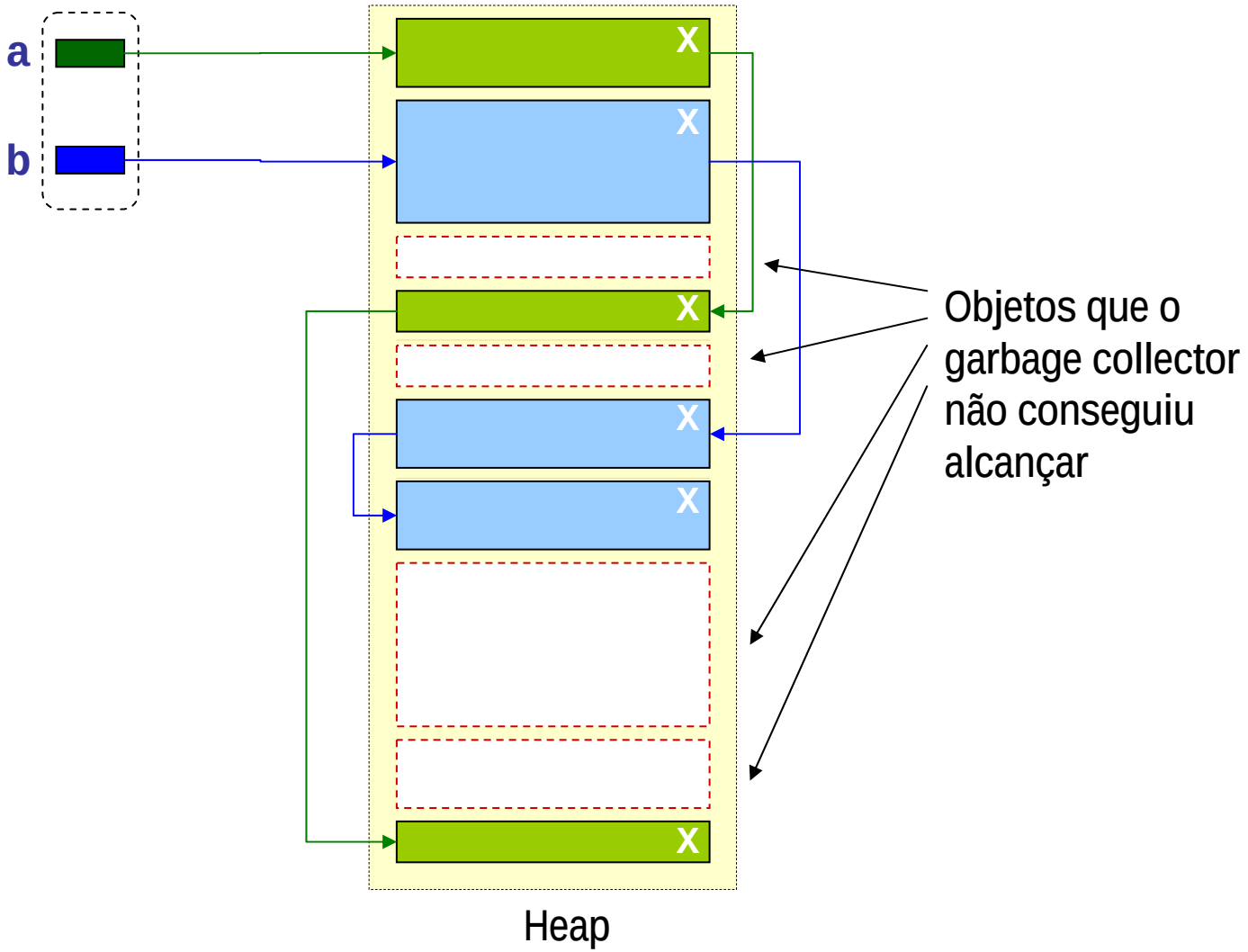
Algoritmo Mark and Sweep (MS)

- É o mais simples algoritmo de rastreamento (tracing)
- É executado quando a memória do heap atinge um nível crítico (ou acaba)
- Todos os threads da aplicação param para executá-lo
  - Comportamento chamado de "Stop-the-World"
- Originalmente projetado para a linguagem LISP pelo seu criador [McCarthy 1960]
- Duas fases
  - Mark: navega pelos objetos alcançáveis a partir do conjunto raiz e deixa uma marca neles
  - Sweep: varre o heap inteiro para remover os objetos que não estiverem marcados (lixo), liberando a memória

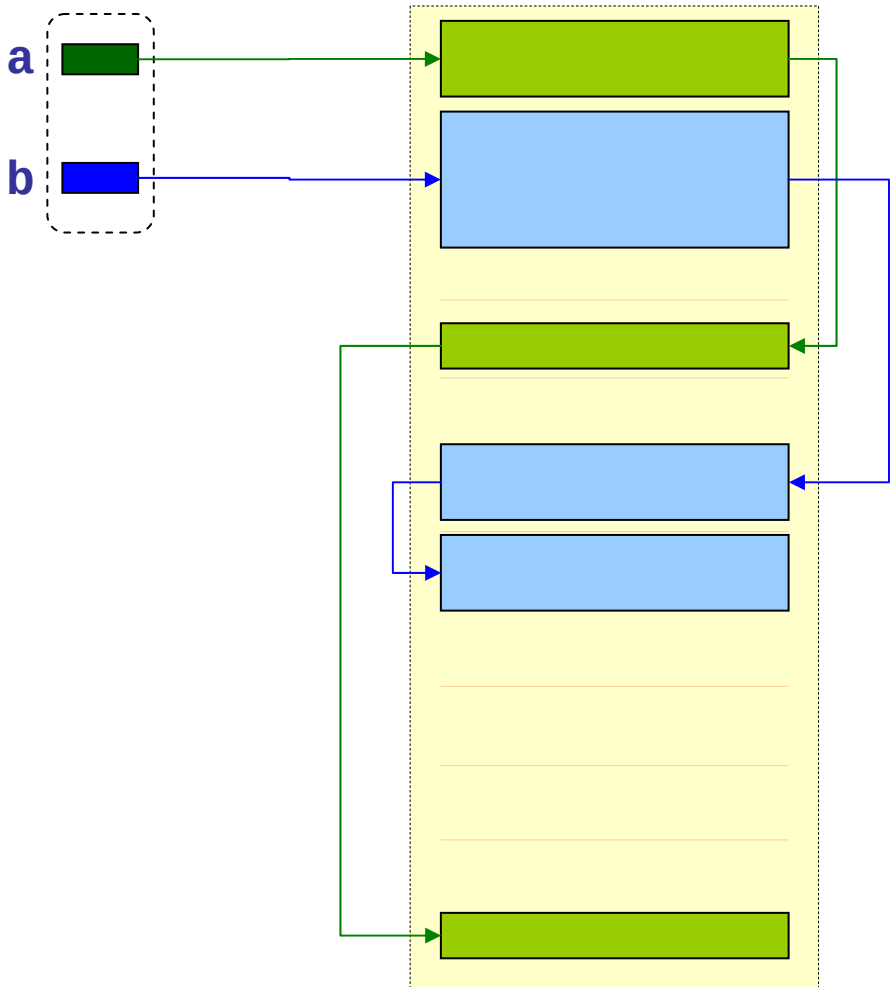
MS: o heap antes da coleta



MS: fase de marcação: Mark

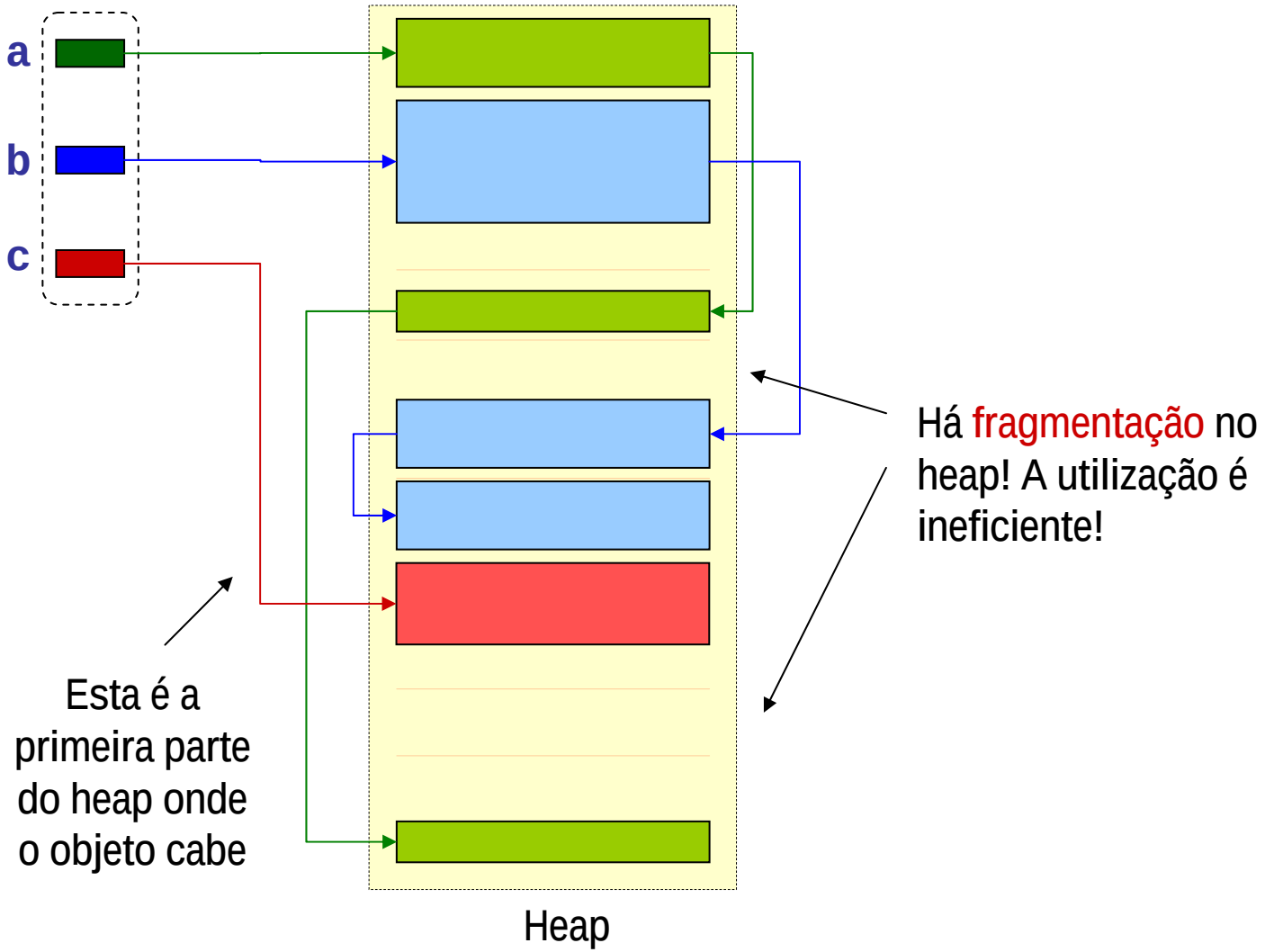


MS: fase da faxina: Sweep



Objetos que não foram marcados foram varridos do heap!

MS: alocação de novos objetos



MS: considerações

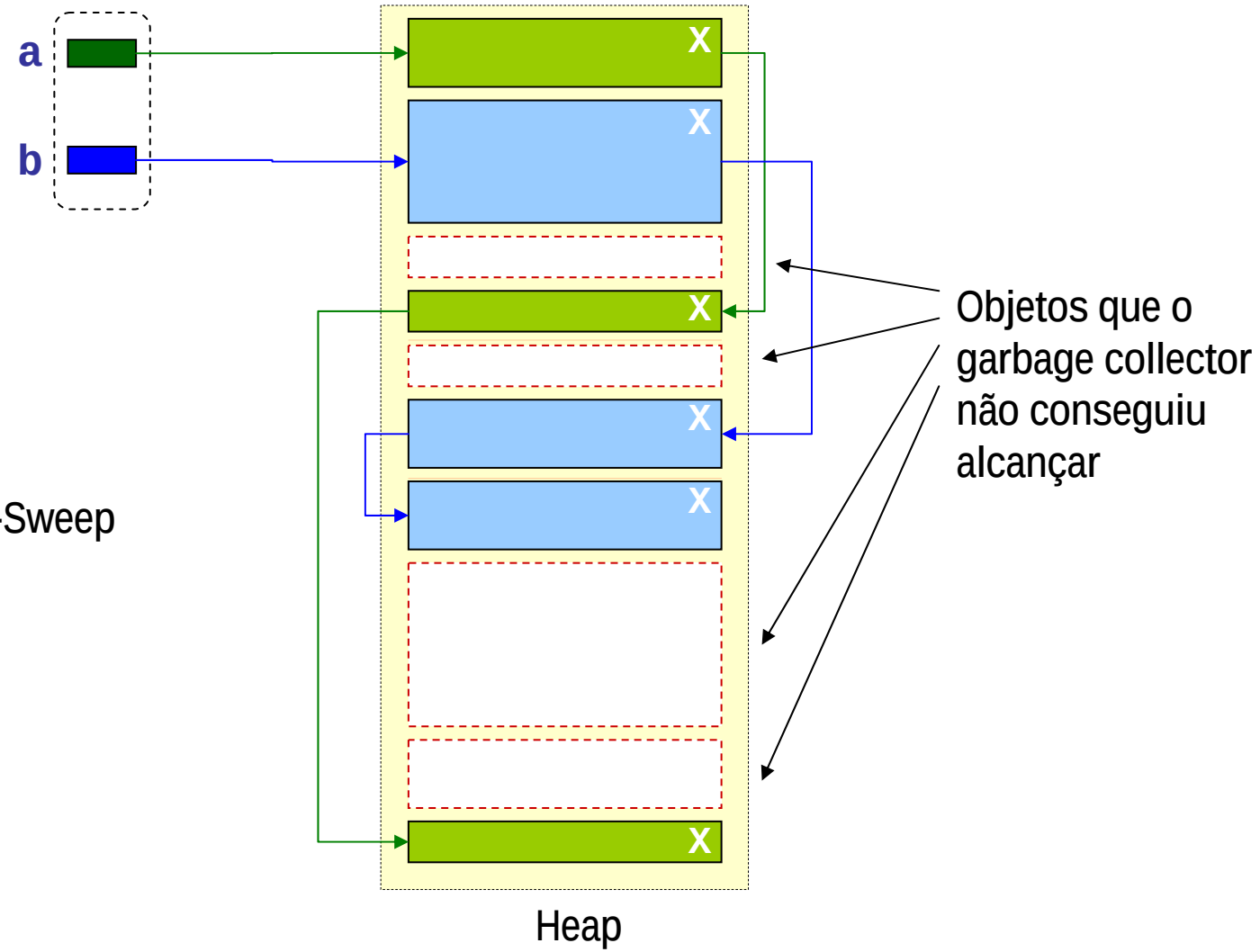
- Vantagens
  - Não precisa de algoritmo complicado para remover objetos com referências circulares
  - Pode ser mais rápido que contagem de referências *se* heap não for excessivamente grande e objetos morrerem com freqüência
- Desvantagens
  - Interrompe a aplicação principal (provoca pausa)
  - Fragmentação pode aumentar a freqüência em que o CG ocorre, com o tempo
  - Precisa visitar todos os objetos alcançáveis na fase de marcação e varrer o heap inteiro para localizar objetos não marcados e liberar a memória

Algoritmo Mark-Compact (MC)

[Edwards]

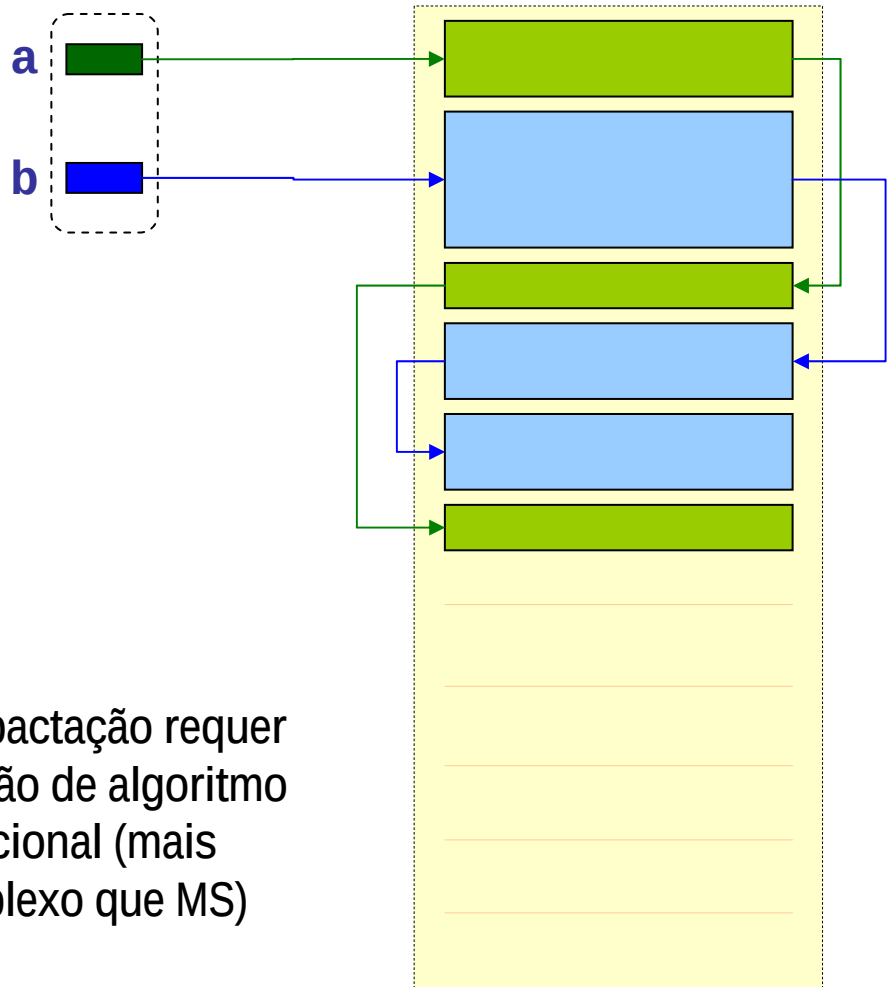
- Algoritmo de rastreamento baseado no algorítmo mark-sweep (MS)
  - Acrescenta um algorítmo de compactação que não provoca fragmentação de memória
- Duas fases
  - Mark: idêntica ao mark-sweep
  - Compact: objetos alcançáveis são movidos para frente até que a memória que ocupa seja contígua
- Também é “Stop-the-World”
  - Precisa interromper a aplicação para rodar

MS: fase de marcação: Mark



Igual a Mark-Sweep

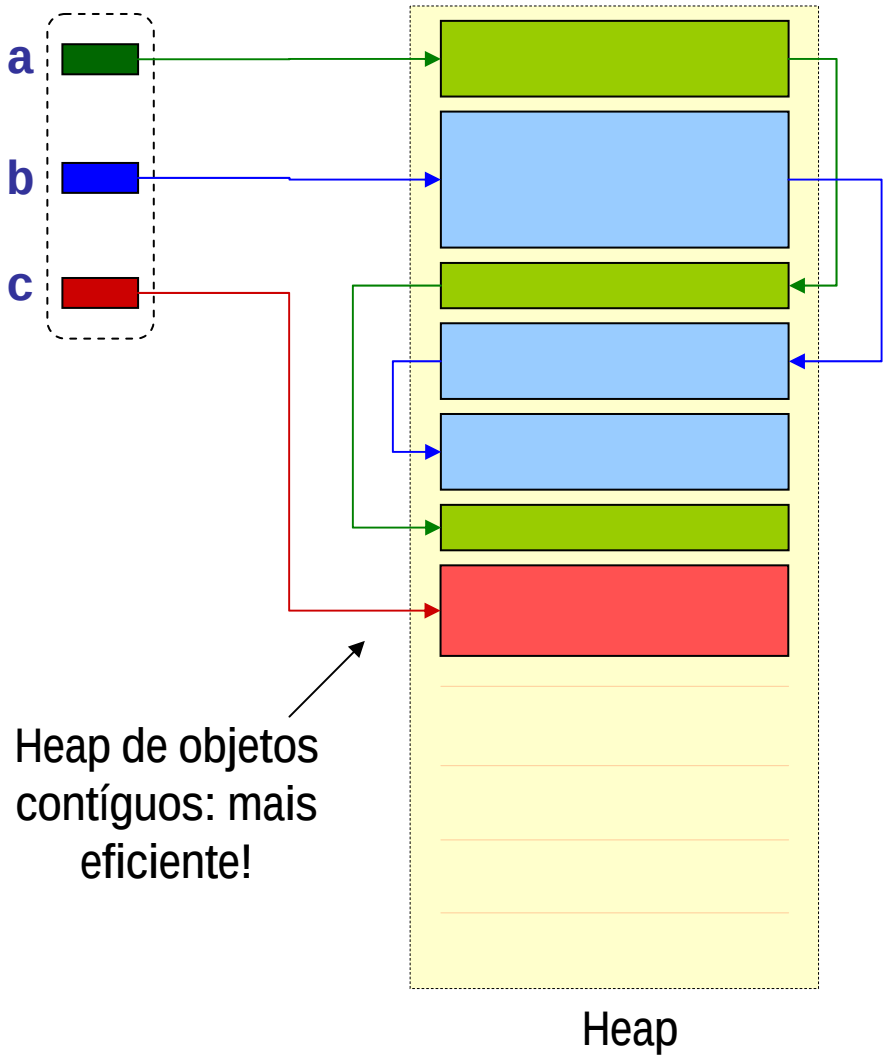
MC: fase de compactação: Compact



A compactação requer execução de algoritmo adicional (mais complexo que MS)

Objetos marcados são movidos para os espaços vazios do início do heap

MC: alocação de novos objetos



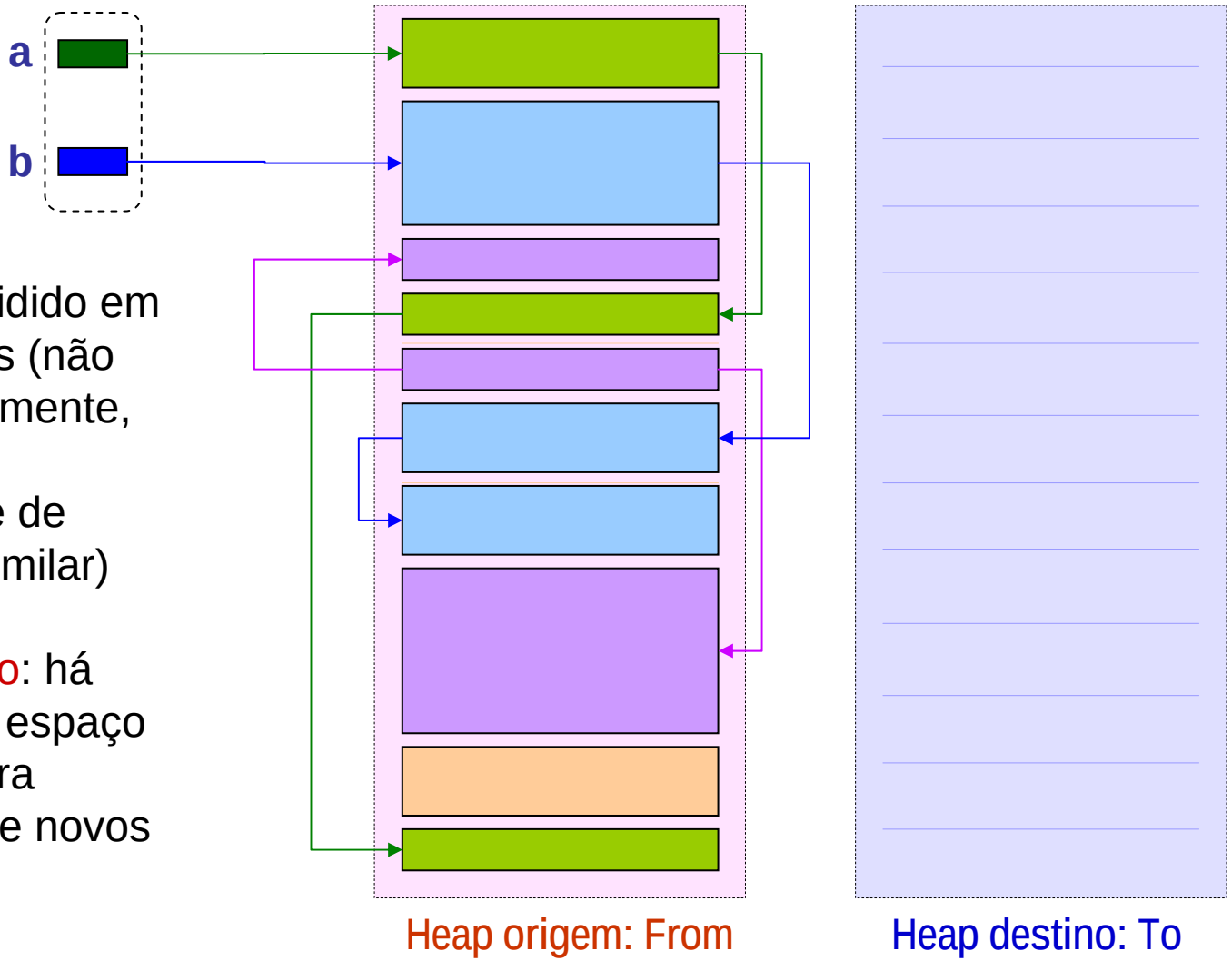
MC: considerações

- Vantagens
  - Não causa fragmentação da memória: alocação é rápida e performance não se degrada com o tempo devido ao aumento das coletas
- Desvantagens
  - Continua sendo Stop-the-World e pausas tendem a ser maiores que as pausas em MS
  - O algoritmo de compactação tem *overhead* (requer várias visitas) e é mais complicado de implementar em CGs concorrentes (multiprocessadores)
- Veja exemplo gráfico interativo de algoritmo Mark-Sweep-Compact (applet Java “Heap of Fish”)
  - http://www.artima.com/insidejvm/applets/HeapOfFish.html (Bill Venners, do livro “Inside the Virtual Machine”) [Venners]

Algoritmo de cópia: CA

- Copying algorithm [Chenney 1970]
- Divide o heap em duas áreas iguais chamadas de origem (from space) e destino (to space)
- Funcionamento
  1. Objetos são alocados na área "from"
  2. Quando o coletor de lixo é executado, ele navega pela corrente de referências e copia os objetos alcançáveis para a área "to"
  3. Quando a cópia é completada, os espaços "to" e "from" trocam de papel

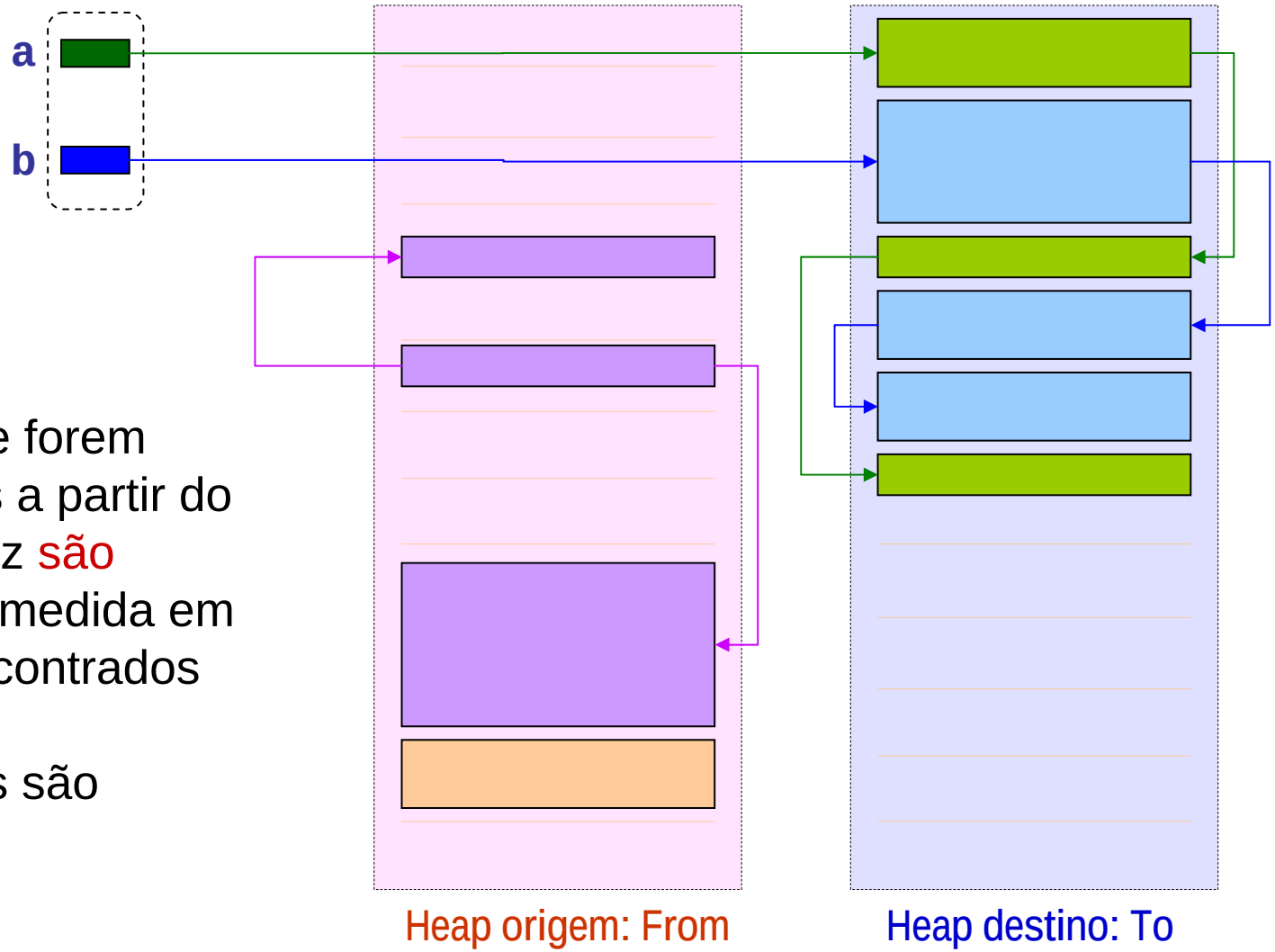
CA: antes da coleta



Heap é dividido em duas partes (não necessariamente, iguais mas geralmente de tamanho similar)

Desperdício: há metade do espaço anterior para alocação de novos objetos!

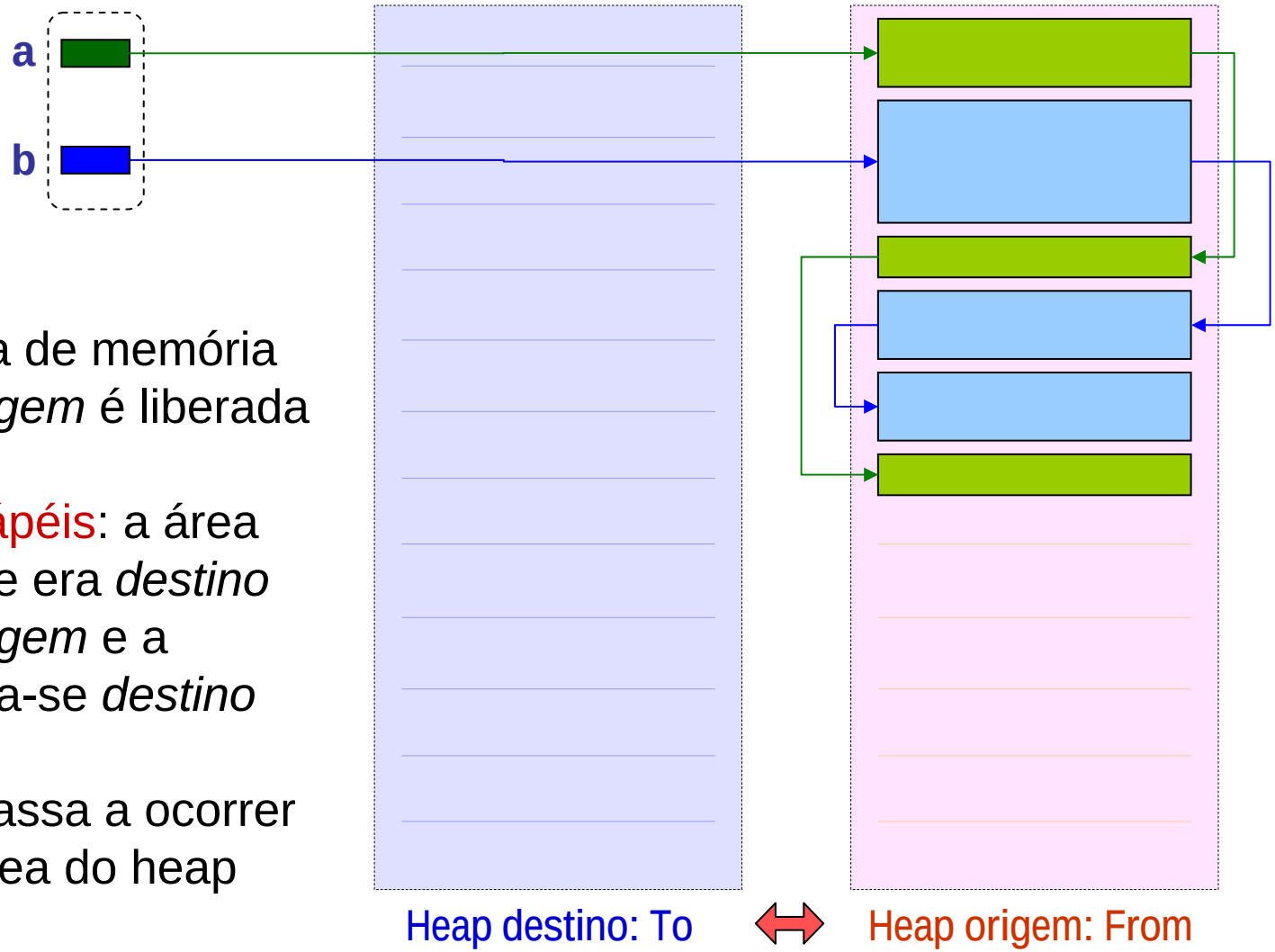
CA: copia objetos alcançáveis



Objetos que forem alcançáveis a partir do conjunto raiz são copiados à medida em que são encontrados

Referências são atualizadas

CA: esvazia e troca de papel

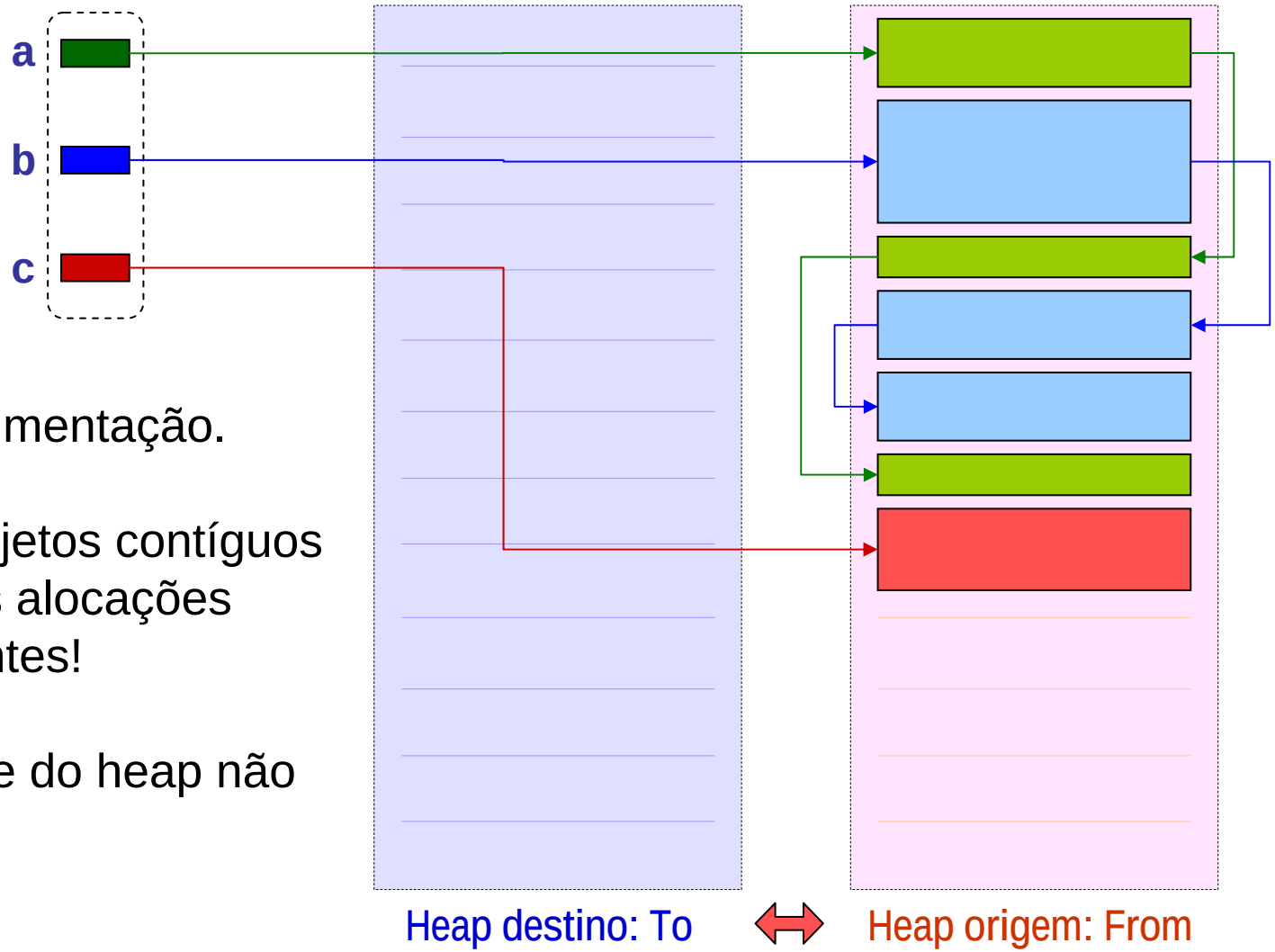


Toda a área de memória do heap *origem* é liberada

Troca de pápéis: a área do heap que era *destino* torna-se *origem* e a *origem* torna-se *destino*

Alocação passa a ocorrer em outra área do heap

AC: alocação de novos objetos



Não há fragmentação.

Heap de objetos contíguos torna novas alocações mais eficientes!

Mas metade do heap não é utilizado

AC: considerações

- Vantagens
  - Cópia é rápida, principalmente se a quantidade de objetos alcançáveis for pequena (o que é comum)
  - Não precisa visitar o heap inteiro: apenas objetos alcançáveis
  - Não fragmenta a memória do heap
- Desvantagens
  - Aplicação precisa parar (stop-the-world) enquanto o algoritmo está sendo executado (como em qualquer algoritmo de rastreamento); versão concorrente [Baker] tem menos pausas
  - Dobra a necessidade de memória do heap: um problema se o heap necessário for muito grande; heaps menores (com metade do tamanho normal) podem disparar o GC com mais freqüência, reduzindo o *throughput*
  - Uso ineficiente de memória (metade está sempre vazia)

Comparação (em coletor serial)

Em vermelho, geralmente um critério negativo

	precisa varrer heap inteiro	coleta todo o lixo	precisa parar a aplicação*	fragmenta o heap	permite uso total do heap	usado no HotSpot JVM**	
Contagem de referências	não	não	não	sim	sim	não	
Coleção de ciclos	não	sim	não	sim	sim	não	
Mark-sweep	sim	sim	sim	sim	sim	sim	
Mark-compact	sim	sim	sim	não	sim	sim	
Copying	não	sim	sim	não	não	sim	

* não funciona de forma incremental (stop-the-world) ** até versão 5.0

3. Estratégias de coleta de lixo

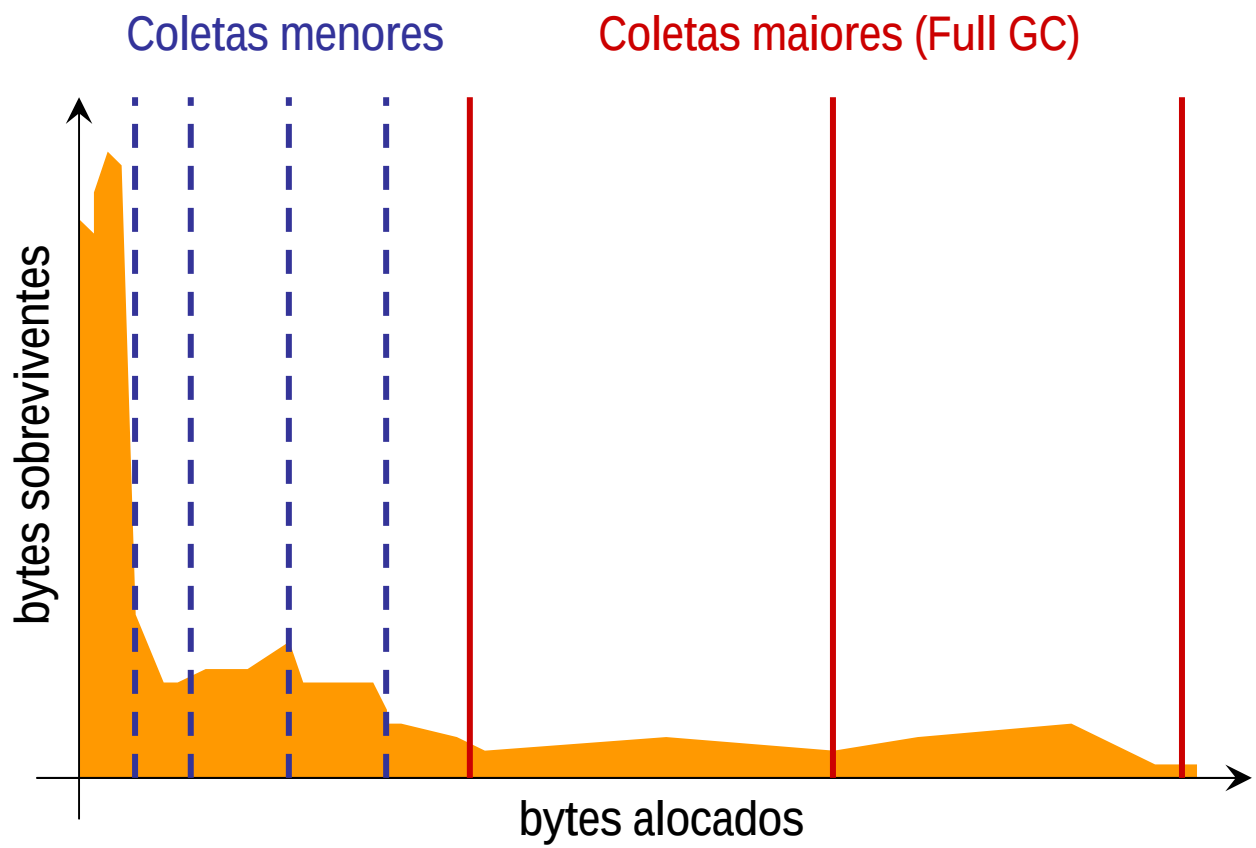
- Coletores modernos combinam vários algoritmos em estratégias mais complexas
  - Aplicando algoritmos diferentes conforme as idades e localização dos objetos
  - Utilizando técnicas que possibilitem a coleta de lixo paralela (algoritmos incrementais e concorrentes)
- Nesta seção apresentaremos as principais estratégias usadas (e propostas) para coletores seriais e paralelos
  - Generational garbage collection (usada na JVM HotSpot)
  - Age-oriented garbage collection (em fase experimental)
- Ambas baseiam-se na idade dos objetos para tornar as coletas mais eficientes

Vida dos objetos

- Observações empíricas
  1. Se um objeto tem sido alcançável por um longo período, é provável que continue assim
  2. Em linguagens funcionais, a maior parte dos objetos morre pouco depois de criados
  3. Referências de objetos velhos para objetos novos são incomuns
- Conclusão
  - Pode-se tornar mais eficiente o coletor de lixo se analisando os objetos jovens com mais freqüência que os objetos mais velhos

Mortalidade infantil dos objetos

- Objetos morrem jovens!
  - A maior parte, pouco depois de serem alocados



Fonte: Sun [Sun 05]

Generational GC

[Lieberman-Hewitt 83] e [Ungar 84]

- Classifica objetos em diferentes gerações: G_0, G_1, ...
  - G_0 contém objetos jovens recém-criados
  - Pressupõe-se que a maior parte dos objetos jovens (90%) é lixo antes da próxima coleta
  - G_n é varrida mais freqüentemente que G_{n+1}
  - Objetos sobreviventes são promovidos para a geração seguinte
- As gerações mais velhas devem ser maiores que as gerações mais novas
  - Tipicamente são exponencialmente maiores
- Implementações típicas usam apenas duas gerações
  - Geração jovem (G_0)
  - Geração estável (G_1)

Fundamentos do Generational GC

- Duas hipóteses
  - Mortalidade infantil dos objetos: a maior parte dos objetos (95%) morre pouco depois que são criados
  - Haverá poucas referências de objetos mais velhos para objetos mais jovens
- As gerações são áreas do heap
  - Geração jovem: área menor, onde é inicialmente alocada a memória para novos objetos
  - Geração antiga, ou estável: área maior, para onde são copiados objetos que sobrevivem a uma ou mais coletas de lixo na área menor

Ponteiros entre gerações

- Quando um objeto é criado, suas referências geralmente apontarão para objetos mais antigos
  - Se houver ponteiros entre gerações, provavelmente serão da geração nova para a geração velha
- Pode acontecer de um objeto antigo receber referência para um objeto novo algum tempo depois de criado
  - Neste caso o sistema precisa interceptar modificações em objetos antigos e manter uma lista de referências
  - Isto deve ocorrer raramente (se ocorrer com freqüência, as coletas menores serão demoradas)
- Na *HotSpot* JVM, é usada uma tabela de referências
  - Geração antiga é dividida em blocos de 512kb (cards)
  - Alterações são interceptadas e blocos são marcados
  - Coletas menores verificam apenas blocos marcados

Generational GC: algorítmos

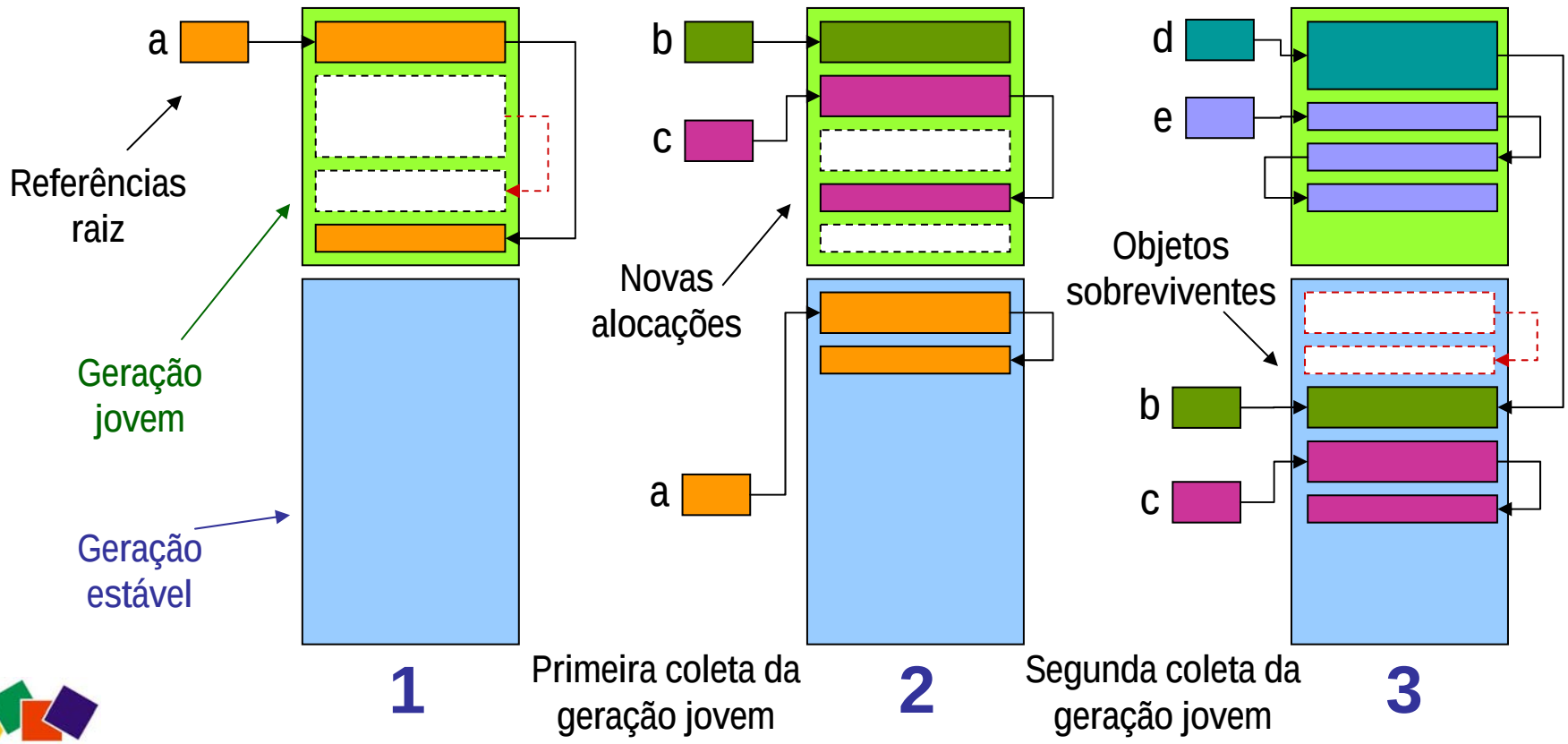
- Usa mais de um algorítmo, uma vez que cada geração possui tamanhos e comportamentos diferentes
- Geração jovem
  - 90% dos objetos estão mortos
  - Área total do heap usado é pequeno
  - Algoritmo de cópia é a melhor opção pois seu custo é proporcional aos objetos ativos
- Geração estável
  - Pode haver muitos objetos ativos e área é grande
  - Não há unanimidade quanto ao algoritmo
  - Pesquisas recentes exploram o uso de algoritmos de contagem de referência (CC) com coletas freqüentes e incrementais
  - *HotSpot* usa MS (na versão concorrente) e MC

Funcionamento: generational GC

- Ilustrando caso típico (há muitas variações)
  - Duas gerações (G_0: jovem e G_1: estável)
  - Algoritmo de cópia usado na geração jovem
- Quando cada geração enche, ocorrem coletas de lixo
  - Parciais na geração jovem, e completas no heap inteiro
- GC Parcial: A geração **jovem** enche primeiro, já que acumula objetos mais rapidamente
  - Quando a geração jovem enche, causa uma coleta menor, que é rápida (proporcional ao número de objetos ativos)
  - Sobreviventes da coleta serão copiados para a geração antiga
- GC Completa: A geração **antiga** cresce ao receber os sobreviventes da geração jovem. Vários irão morrer.
  - Depois de várias coletas menores, a geração antiga enche
  - Quando encher, haverá uma coleta maior (lenta) no heap inteiro

Funcionamento ilustrado

- Geração jovem recebe novas alocações até encher
- Quando a geração jovem enche ocorre a coleta que copia os objetos sobreviventes para a geração estável
- Coleta na geração estável é mais demorada, porém menos freqüente



Considerações

- Vantagens
  - Pausas menores: coletas rápidas e freqüentes distribuem as pausas que podem tornar-se imperceptíveis
  - Eficiência (throughput): a coleta concentra-se nas áreas de memória onde o lixo se encontra, gastando menos tempo
- Desvantagens
  - Pequena geração jovem pode causar início mais lento devido a muitas coletas curtas (baixa eficiência)
  - Coleta na geração antiga ainda é lenta e algoritmos usados atualmente ainda não conseguem eliminar pausas
- Implementações
  - Na JVM HotSpot, geração antiga usa diversos algorítmos (MC, MS)
  - Há pesquisas usando RC (CC) para coletar geração antiga eficientemente: [Azatchi-Petrank 03] e [Blackburn-Mckinley 03] com implementações experimentais testadas no Jikes RVM

Age-oriented

[Paz, Petrank & Blackburn 2005]

- Divide objetos em gerações
  - Ocupam tamanho variável do heap
  - Sempre coleta heap inteiro
- Busca reduzir pausas com concorrência
- Implementação recomendada usa
  - Algoritmo de rastreamento (cópia) na geração jovem (da mesma forma que implementações típicas do Generational GC)
  - Algoritmo de contagem de referências (CC) na geração antiga
- Inicialmente geração jovem ocupa todo o espaço
  - Alta eficiência (demora ocorrência de primeira coleta)
- Geração antiga cresce com coletas
  - Pequena geração antiga com mais objetos ativos que mortos e pouca atividade permite eficiência máxima do algoritmo CC

Generational vs. Age-oriented

- Generational
  - Geração jovem menor que geração velha
  - Faz coletas freqüentes apenas na geração jovem
  - Faz coleta do heap inteiro, com algoritmo diferente, após várias coletas da geração jovem
- Age-oriented
  - Geração jovem maior que geração velha
  - Sempre coleta o heap inteiro, usando algoritmos diferentes para cada geração

Generational vs. Age-oriented

Fonte: [Paz, Petrank & Blackburn 2005]

Considerações

- Vantagens
  - Maior geração jovem possível (coletas raras e menos pausas)
  - Cada geração tratada diferentemente (eficiência)
- Desvantagens
  - Pausas serão longas na geração jovem se não for usado um coletor concorrente (usando algoritmo de cópia concorrente)
  - Coleta na geração antiga será ineficiente se for usado algoritmo de rastreamento (ideal é usar contagem de referências)
- Suporte
  - Experimental (objeto de pesquisa atual)
  - Não é suportado por nenhuma JVM HotSpot no momento
  - Implementação em Jikes RVM obteve performance melhor que implementação HotSpot

4. Coleta de lixo em paralelo

- Há três* estratégias de coleta de lixo quanto à execução paralela do coletor de lixo
  - Coleta serial: o coletor ocorre em série com a aplicação, parando o mundo (stop-the-world) quando precisar liberar memória
  - Coleta incremental (on-the-fly): o coletor executa em paralelo realizando coletas pequenas (não necessariamente completas) sempre que possível, usando vários threads visando menos (ou zero) pausas
  - Coleta concorrente: o coletor realiza suas principais tarefas em um processador ou *thread* exclusivo (pode parar todos os *threads* para marcar, se necessário) visando maior eficiência

** HotSpot divide em quatro, devido a diferentes estratégias para gerações individuais*

Parando o mundo

- Os algoritmos seriais de rastreamento precisam parar todos os threads para realizar coleta de memória
  - Rastreamento: qualquer estratégia que use MC, MS ou CA
- Por que é preciso parar o mundo?
  - Enquanto o thread de rastreamento varre o heap à procura de objetos alcançáveis, alguns dos já marcados poderiam tornar-se inalcançáveis se o programa principal não fosse interrompido
- Por que não usar os algoritmos de contagem de referência (RC e CC) ?
  - É uma solução. Podem operar em paralelo sem parar tudo
  - Ainda estão pouco maduros (são foco atual de pesquisas)
  - Atualmente são usados apenas experimentalmente; é possível que venham a ser usados no futuro nas JVMs HotSpot

Rastreamento incremental

- Os algoritmos de rastreamento mostrados não podem ser usados em sistemas de tempo real pois introduzem a qualquer momento pausas de duração imprevisível
  - Sistemas de tempo real requerem tempos de resposta previsíveis e determinísticos
- Para usar coleta de lixo em sistemas de tempo real é preciso eliminar totalmente as pausas!
- Solução: buscar algoritmos capazes de executar em paralelo e não interferir na execução da aplicação
  - Soluções baseadas em contagem de referências RC/CC
  - Versões incrementais de algoritmos de cópia e MS/MC
  - Rastreamento baseado em marcação tricolor (TCM)

Marcação tricolor (TCM)

[Dijkstra 76]

- Um algoritmo de rastreamento que atribui um entre três estados (cores) a um nó do grafo de objetos
  - Principal algoritmo de coleta de lixo incremental
- Cores e tipos de nó
  - Branco: nó e seus filhos (objetos ao qual se refere) ainda não alcançados (não foram marcados)
  - Cinza: nó já foi alcançado (e marcado), mas seus filhos ainda não foram
  - Preto: nó e seus filhos já foram alcançados e marcados
- No final da fase de marcação do algoritmo, os nós que ainda estão marcados com a cor branca podem ser coletados com segurança

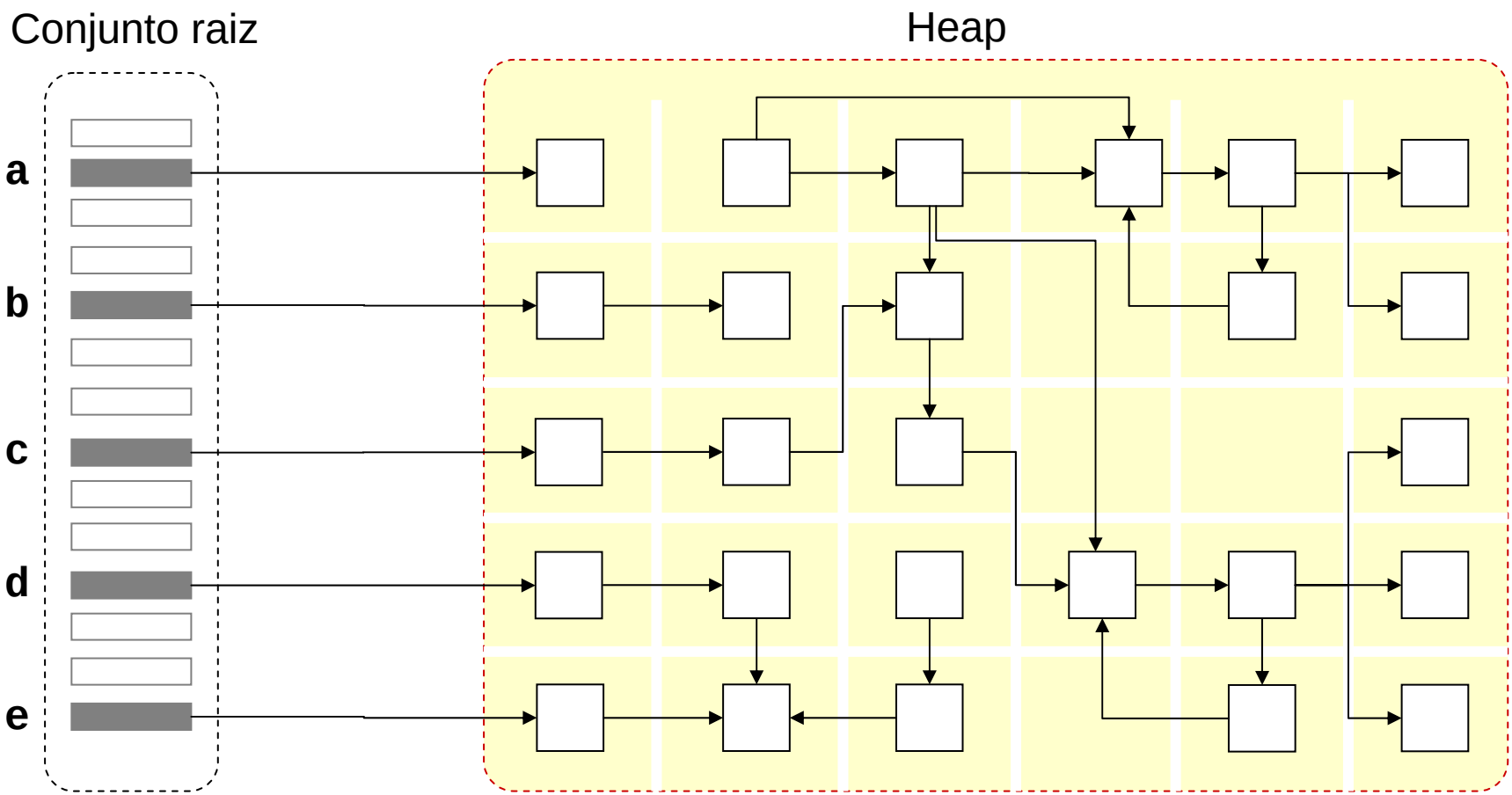
* Tri-Colour Marking

TCM: funcionamento

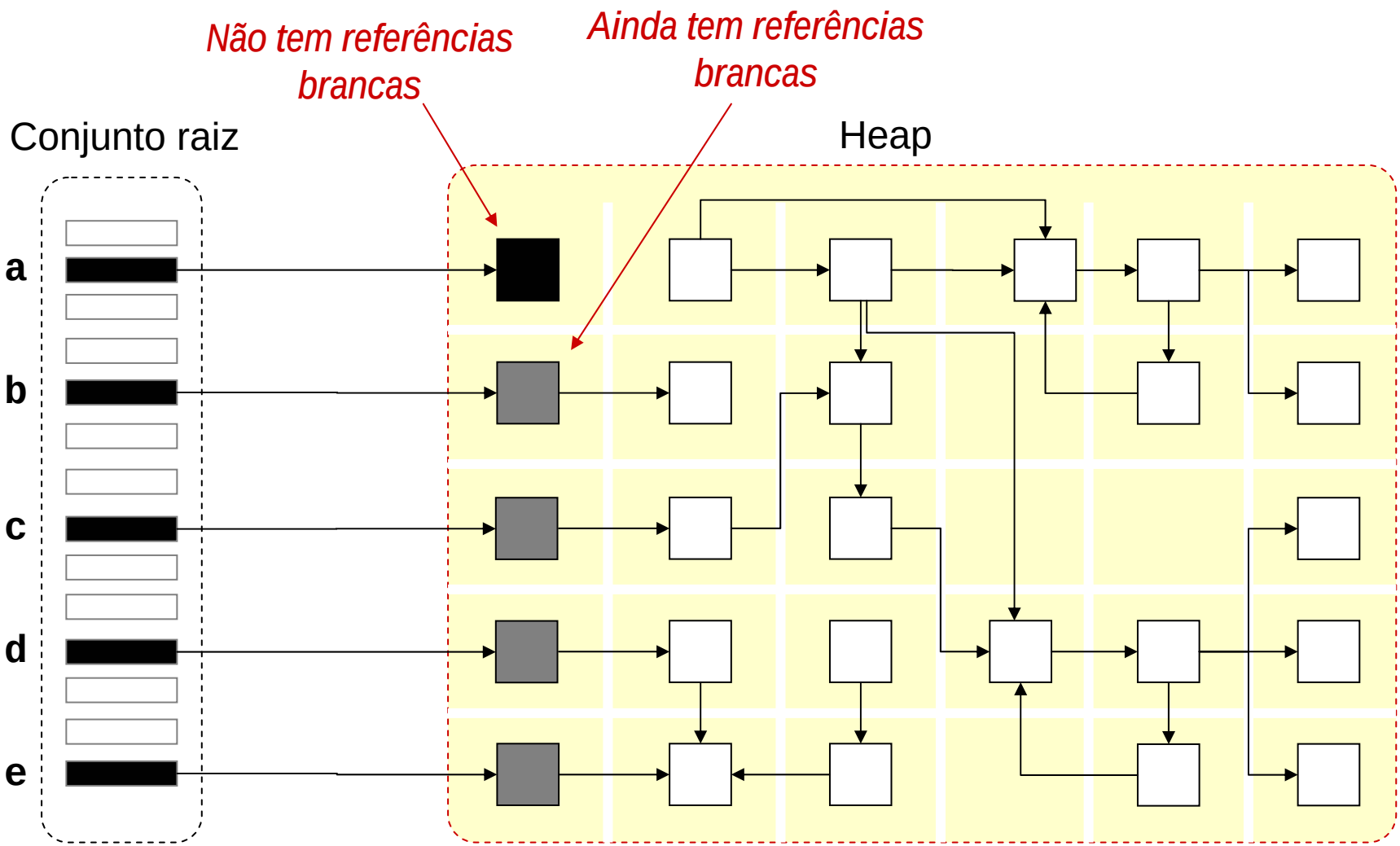
1. Inicialização
   - Inicialmente todos os nós são brancos (inalcançáveis)
   - O conjunto de referências raiz é marcada cinza
2. Rastreamento e marcação 1 (cinza):
   - Quando o coletor encontra um caminho entre um nó cinza e um nó branco, pinta o nó branco de cinza
   - Prossegue recursivamente até encontrar todos os objetos alcançáveis a partir dele, pintando cada um de cinza
3. Rastreamento e marcação 2 (preto):
   - Quando todos os caminhos de um nó cinza levam a nós cinza ou pretos, o nó é pintado de preto
4. Reciclagem e liberação de memória:
   - Quando não houver mais nós cinzas, todos os nós alcançáveis foram encontrados. Os nós brancos restantes são reciclados.

TCM (1): inicialização

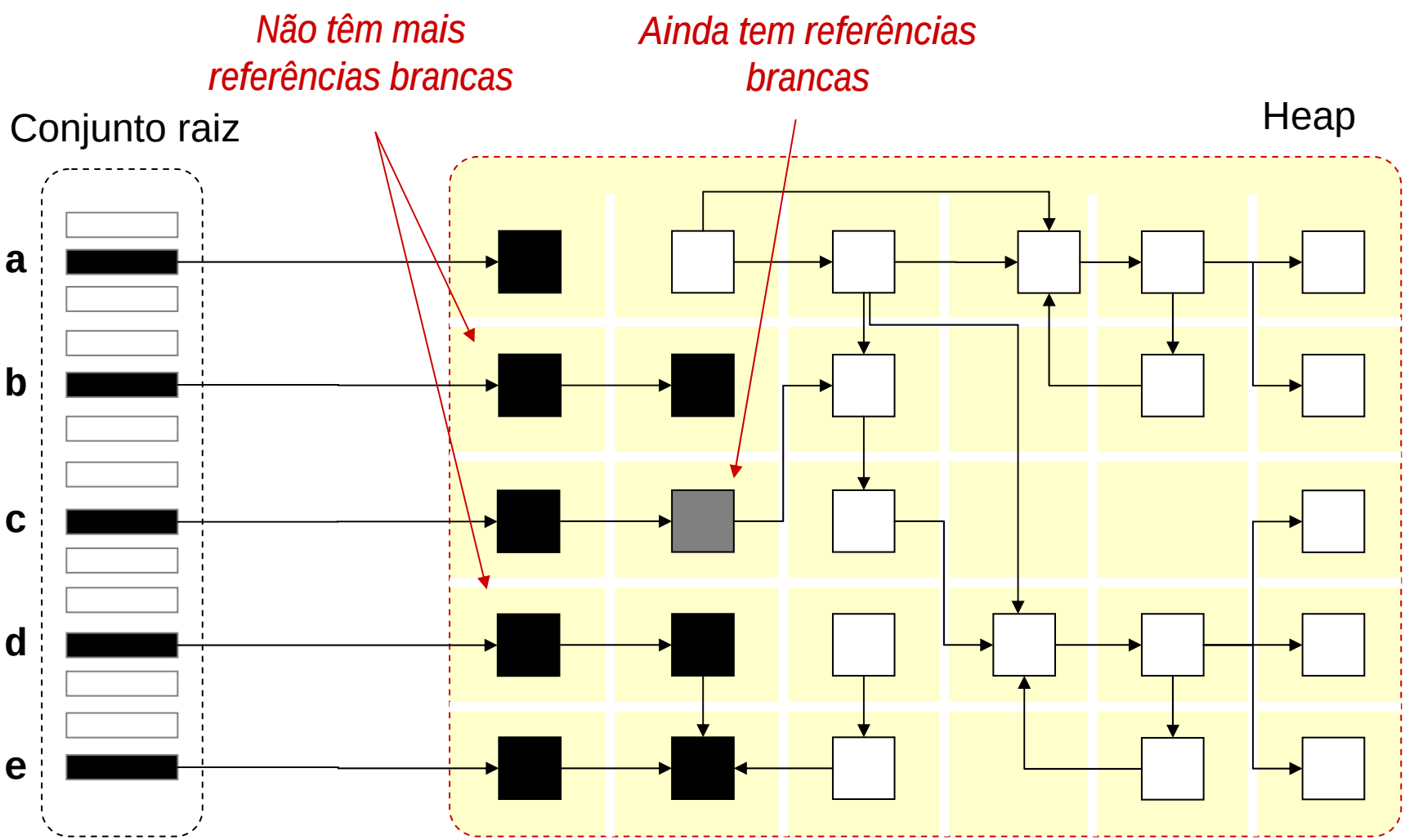
- Todos os nós inicialmente marcados como brancos
- referências raiz em cinza



TCM (2): marcação cinza

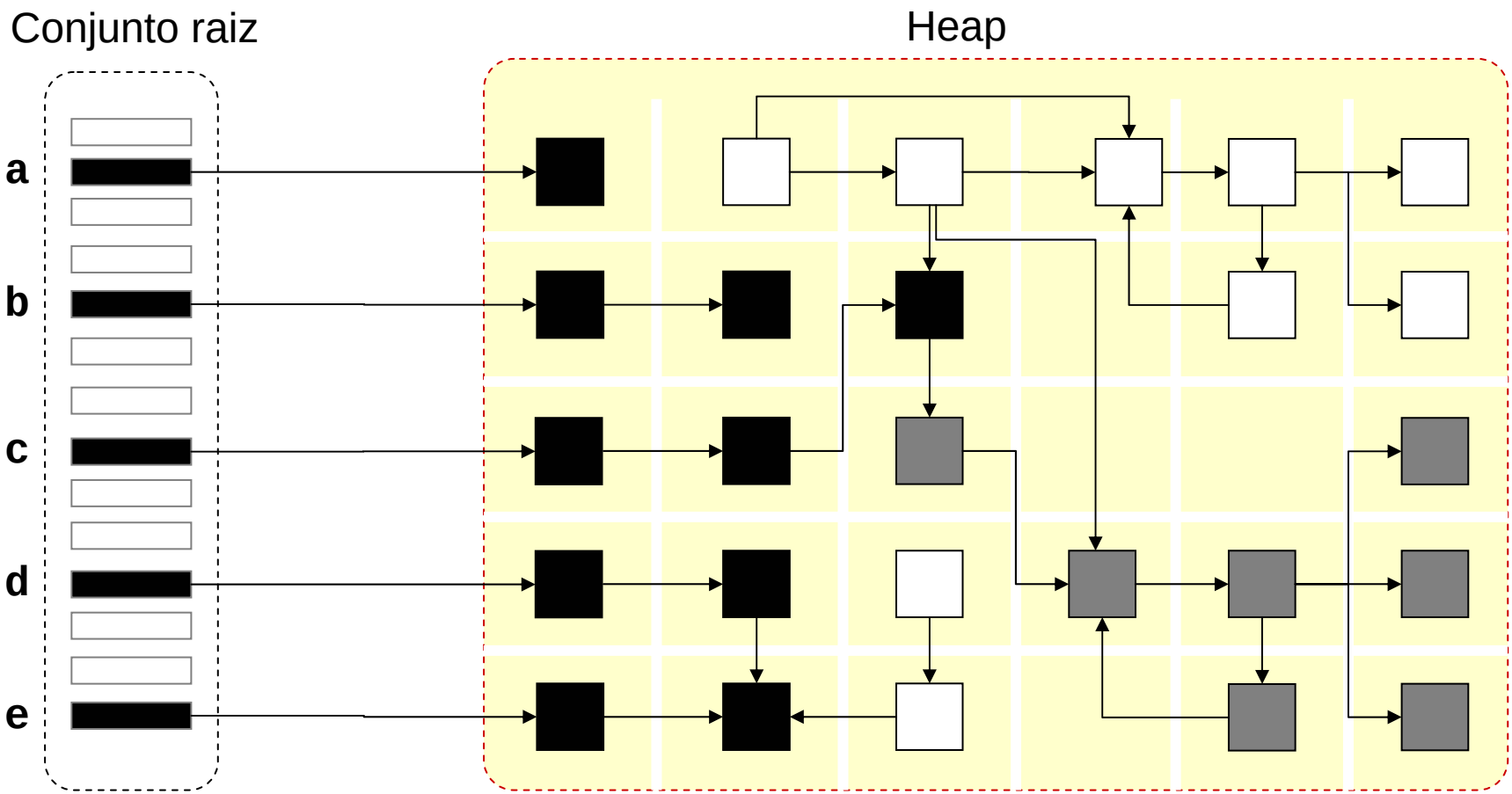


TCM (3): marcação preta



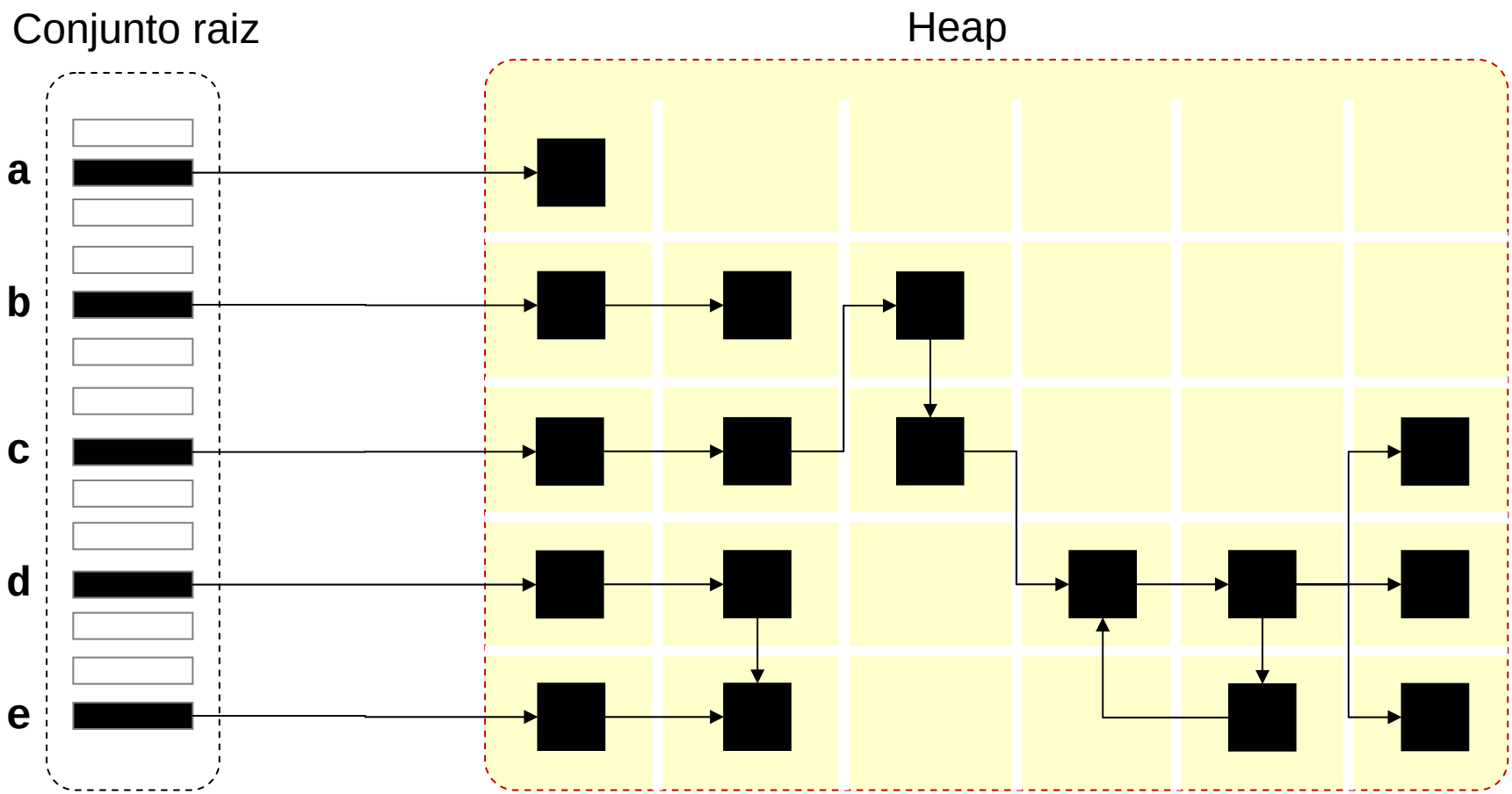
TCM (4): marcações recursivas

Brancos que sobraram serão coletados pois são inaccessíveis



TCM (5): reciclagem dos brancos

Heap final contém apenas objetos ativos



Invariante tricolor

- Um objeto preto nunca poderá ter referências para objetos brancos
  - Quando aplicação gravar uma referência entre um nó preto e um branco, o coletor precisará pintar ou o nó pai ou o nó filho de cinza
  - Quando a aplicação quiser ler um nó branco, ele tem que ser pintado de cinza
- Para realizar isto, o sistema precisa
  - Rastrear gravações em nós pretos (usando uma barreira de gravação – *write barrier*)
  - Rastrear leituras em nós brancos (usando uma barreira de leitura – *read barrier*)

TCM: considerações

- Vantagens
  - Possibilidade de uso incremental e eliminação de pausas na coleta de lixo (permite uso em aplicações de tempo real)
  - Melhor performance aparente
- Desvantagens
  - Sincronização é complexa entre a aplicação e o coletor de lixo
  - Barreiras podem dificultar a implementação em diferentes sistemas e diminuir a eficiência (menos *throughput*)
- Suporte em JVMs
  - Atualmente, em máquinas virtuais Java é usado apenas experimentalmente (HotSpot JVM não usa este algoritmo; usa versões incrementais de outros algoritmos de rastreamento)

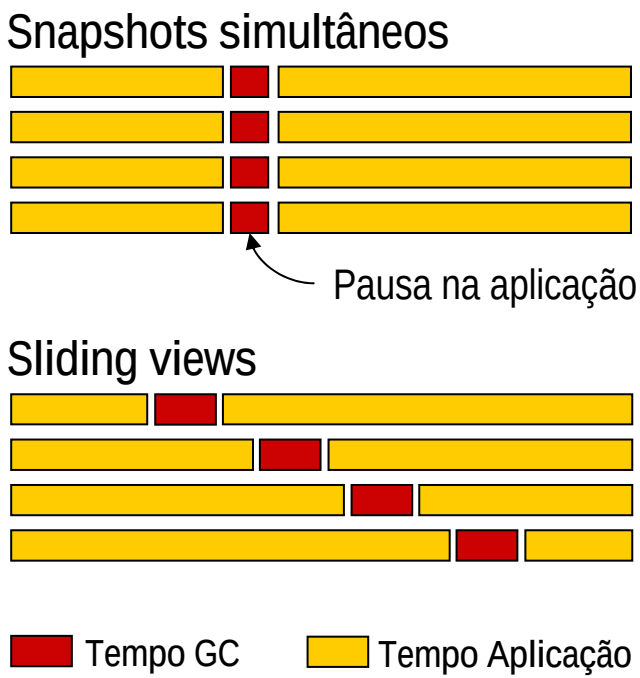
Train algorithm

[Hudson & Moss 92]

- Algoritmo incremental usado para coleta de geração estável em sistemas que usam Generational GC
- Divide a memória em blocos de tamanho fixo
  - Metáfora: blocos menores são “vagões” e coleções de tamanho arbitrário desses blocos são “trens”
  - Trens e vagões são ordenados por idade; os mais antigos são coletados enquanto novos trens e vagões se formam
  - Entre a formação e coleta, atualiza-se referências entre objetos
  - Muito ineficiente com objetos populares (objetos que têm muitas referências) que podem ocorrer com freqüência nas gerações estáveis; versões eficientes podem ter pausas pequenas
- O Train algorithm é usado na JVM HotSpot
  - Mas parou de ser mantido a partir da versão 1.4.2

Snapshots e Sliding Views

- Coletores paralelos precisam trabalhar com heaps que mudam durante a coleta
  - Enquanto um *thread* marca os objetos outro thread pode estar liberando referências (gerando lixo)
  - É preciso trabalhar com modelos estáticos do heap (snapshots ou sliding views) e coletar de forma incremental
- Snapshots (concurrent GC)
  - Precisa parar todos os threads para obter um modelo do heap em determinado momento
- Sliding views (on-the-fly GC)
  - Pára um thread de cada vez, em tempos desencontrados para obter a visão do heap (sem pausa na aplicação)



On-the-fly GC (incremental)

- Área de pesquisa emergente
  - Contagem de referência (CC) torna-se mais popular com heaps grandes (já que pesquisa objetos vivos) e sistemas paralelos (onde suas desvantagens diminuem)
- Vários artigos recentes
  - RC (CC) on-the-fly com sliding views [Levanoni-Petrank 01]
  - Mark Sweep on-the-fly com sliding views [Azatchi-Levanoni 03]
  - On-the-fly cycle collection [Paz et al 2003]
  - On-the-fly generational collector [Domani et al 2000]
- Implementações
  - Todas experimentais (Jikes RVM)
  - Podem aparecer em versões futuras de VMs comerciais de Java (e também de C#)

Concurrent GC (rastreamento)

- Algoritmo de cópia concorrente
  - Cópia incremental ([Baker 78]): ponteiros são lidos apenas em *to_space*; se ponteiro estiver em *from_space* na leitura, primeiro copia objeto depois obtém ponteiro
  - Algoritmo similar é usado pela HotSpot JVM para coletar paralelamente a geração jovem (mais sobre isto na seção seguinte). Veja [Flood et al 2001].
- Mark-sweep concorrente
  - Usado pelo HotSpot JVM para coletar paralelamente a geração antiga (mais na seção seguinte) [Printezis 00]
  - Causa fragmentação (não compacta)
  - Causa pausa pequena para obter snapshot (pára todos os *threads* ao mesmo tempo)
  - Versão com compactação em desenvolvimento [Flood et al 01]

Conclusões

- Existem muitas estratégias de coleta de lixo
  - Há muito, muito mais do que foi exposto aqui
- Embora o programador Java não tenha a opção de escolher qual usar, as JVMs podem permitir essa escolha e configuração
- Muito pode mudar nas próximas versões das JVMs existentes atualmente
  - Há muitas estratégias experimentais que poderão ser usadas em versões futuras, em diferentes plataformas
  - Há estratégias antigas caindo em desuso
- Conhecer o funcionamento dos principais algoritmos ajudará a configurar e ajustar a performance da JVM em diferentes tipos de aplicações

Referências: algoritmos (artigos, 1)

[Collins 60] G. Collins. A Method for Overlapping and Erasure of Lists, IBM, CACM, 1960. *Algoritmo de contagem de referências.*

[McCarthy 60] J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I, MIT, CACM, 1960. *Artigo original do Mark-Sweep algorithm (em Lisp).*

[Edwards] D.J. Edwards. Lisp II Garbage Collector. MIT. AI Memo 19. ftp://publications.ai.mit.edu/ai-publications/0-499/AIM-019.ps. *Mark-Compact.*

[Cheney 70] C. J. Cheney. A Nonrecursive List Compacting Algorithm. CACM, Nov 1970. *Artigo original do copying algorithm.*

[Baker 78] H. G. Baker. List processing in real time on a serial computer. CACM, Apr 1978. *Uma versão concorrente do copying algorithm.*

[Lieberman-Hewitt 83] H. Lieberman, C. Hewitt. A Real Time Garbage Collector Based on the Lifetimes of Objects. CACM, June 1983. *Artigo principal do Generational GC.*

[Dijkstra 76] E. W. Dijkstra, L. Lamport, et al. On-the-fly Garbage Collection: An Exercise in Cooperation. Lecture Notes in Computer Science, Vol. 46. 1976. *Tri-color marking (citado em [Jones & Lins 95]).*

[Ungar 84] David Ungar. Generation Scavenging: A Non-disruptlve High Performance Storage Reclamation Algorithm. ACM, 1984. *Um dos artigos do Generational GC.*

A maioria dos artigos pode ser obtido via ACM Portal (www.acm.org), CiteSeer (http://citeseer.ist.psu.edu/), sites da Sun, MIT ou IBM, ou localizados via Google

Referências: algoritmos (artigos, 2)

[Hudson & Moss 92] R. Hudson, J.E.B. Moss. Incremental Collection of Mature Objects, ACM/IWMM, Sep 1992. *Artigo do Train algorithm.*

[Domani 00] T. Domani et. al. A Generational On-The-Fly Garbage Collector for Java, IBM, 2000.

[Printezis 00] Tony Printezis and David Detlefs. A Generational Mostly-concurrent Garbage Collector, 2000. *Algoritmo usado no HotSpot.*

[Flood et al 02] Christine Flood et al. Parallel Garbage Collection for Shared Memory Multiprocessors. Sun Microsystems. Usenix, 2001. *Algoritmos usados no HotSpot.*

[Bacon-Rajan 01] D. Bacon, V. T. Rajan. Concurrent Cycle Collection in Reference Counted Systems. IBM, 2001.

[Levanoni-Petrank 01] Y. Levanoni, E. Petrank. An On-the-fly Reference Counting Garbage Collector for Java, IBM, 2001.

[Azatchi 03] H. Azatchi et al. An On-the-Fly Mark and Sweep Garbage Collector Based on Sliding Views. OOPSLA 03, ACM, 2003.

[Paz 05] H. Paz et al. Efficient On-the-Fly Cycle Collection. IBM (Haifa), 2005.

[Paz-Petrank-Blackburn 05] H. Paz, E. Petrank, S. Blackburn. Age-Oriented Concurrent Garbage Collection, 2005.

Referências: outros tópicos

- Gerência de memória

[Memory] The Memory Management Reference. http://www.memorymanagement.org/. *Várias referências e textos sobre gerêncie de memória em geral.*

- Máquina virtual da Sun

[JVMS] T. Lindholm, F. Yellin. The Java Virtual Machine Specification, second edition, Sun Microsystems, 1999. *Formato de memória, pilha, heap, registradores na JVM.*

[Sun 05] Sun Microsystems. Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine. 2005. *Generational GC e estratégias paralelas no HotSpot.*

[HotSpot] Sun Microsystems. The Java HotSpot™ Virtual Machine, v1.4.1, Technical White Paper. Sept. 2002. *Algoritmos usados no HotSpot.*

[Printezis 05] Tony Printezis. Garbage Collection in the Java HotSpot Virtual Machine. http://www.devx.com/Java/Article/21977, DevX, 2005.

- Livros

[Jones & Lins 96] R. Jones, R.Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley 1996. *Várias estratégias de GC explicadas.*

- Simulações

[Venners] Bill Venners, Inside the Virtual Machine. Applet Heap of Fish: http://www.artima.com/insidejvm/applets/HeapOfFish.html