Informática – UFRGS – Disciplina de Modelos de Linguagens de Programação
Prof. Leandro Krug Wives

GERÊNCIA DE MEMÓRIA HEAP – *GARBAGE COLLECTOR*

A memória física sempre foi recurso escasso e caro na história da computação. Atualmente, isso não chega a ser um problema, pois a memória física está mais acessível. No entanto, se comparada com outros recursos como o disco rígido (HD), ela ainda é um recurso relativamente limitado e caro, já que, pelo mesmo preço, podemos comprar um HD que armazena muito mais memória.

Se você é programador experiente, concordará com o seguinte argumento de Jones e Lins (1996). Segundo eles, prever o consumo de memória de programas pode ser simples em alguns casos, mas, para programas maiores e mais complexos em termos de estruturas de dados, estimar e gerenciar sua memória pode dar bastante trabalho. Linguagens de escopo dinâmico, funcionais, lógicas e mesmo orientadas a objetos tipicamente manipulam estruturas de dados grandes e complexas. Muitas delas não podem ser determinadas em tempo de compilação, pelo seu dinamismo. Nesses casos, o gerenciamento automático é necessário (op. cit.).

O gerenciamento automático citado no parágrafo anterior é tradicionalmente feito por mecanismos denominados de "coletores de lixo" (*garbage collectors*). O seu nome advém do fato de que toda memória previamente alocada e que não está mais ativa é considerada lixo, e pode ser reutilizada.

Ainda segundo Jones e Lins (1996), os mecanismos de coleta de lixo também são interessantes porque diminuem a complexidade de programação, já que abstraem (escondem) do programador a preocupação com o gerenciamento de memória. São, portanto, uma poderosa ferramenta de Engenharia de Software, pois livram o programador do trabalho de descobrir erros relacionados com a gerência de memória, garantindo que eles não aconteçam, tornando os componentes mais seguros e independentes entre si.

Um dos grandes problemas, então, passa a ser diferenciar os objetos ativos ("vivos") daquilo que é lixo (i.e., os não ativos). Segundo Venners (2000), há duas abordagens principais para diferenciar objetos "vivos" de lixo, sendo elas (i) *reference counting* e (ii) *tracing*.

Desde 1960, diversos algoritmos de coleta de lixo baseados nessas duas abordagens foram desenvolvidos. Atualmente, as principais famílias de algoritmos de coleta são (a) *Reference Counting*, (b) *Mark-Sweep* e (c) *Copying Collector*. Essas duas últimas seguem a abordagem de *tracing*.

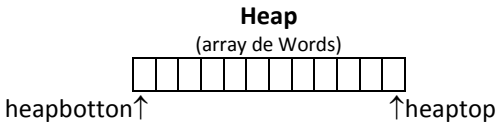
Esse texto descreve tais famílias, suas vantagens e desvantagens. Cabe salientar que os algoritmos apresentados constituem-se nas versões mais simplificadas dos mesmos. Na prática, extensões ou refinamentos são utilizados. O livro de Jones e Lins (1996) apresenta capítulos específicos para os algoritmos aplicados na prática, inclusive para sistemas distribuídos. Antes, porém, vamos analisar algumas definições e conceitos relacionados.

1 DEFINIÇÕES E CONCEITOS

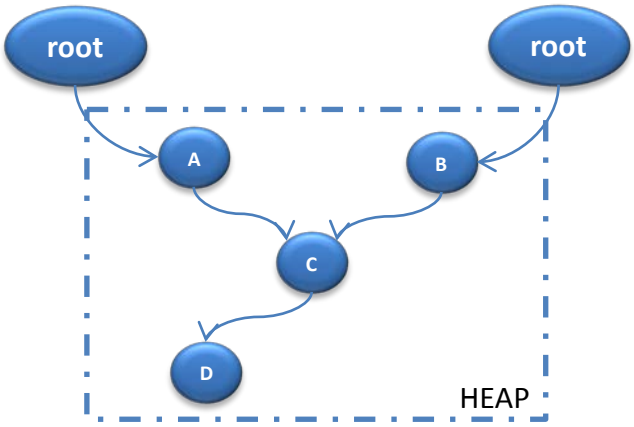
A fim de compreender adequadamente os algoritmos de gerência de memória Heap (i.e., os *garbage collector*s), é importante revisitar os seguintes conceitos:

- Memória Heap: o Heap (também chamado de “monte”) é um bloco de memória RAM usado para armazenar variáveis dinâmicas. Em muitas linguagens, como C e C++, tal região é controlada pelo próprio programador, ou seja, é alocada e desalocada com o uso de operadores ou funções específicos. Em C, por exemplo, utiliza-se *malloc()* para alocar memória no Heap e *free()* para desalocar. Em C++ utiliza-se *new* e *delete* para, respectivamente, alocar e desalocar memória no Heap. Um dos grandes problemas da memória Heap é a fragmentação decorrente do seu uso, visto que só podemos alocar regiões contíguas de memória Heap. Com isso, eventualmente, uma requisição de alocação pode não ser atendida, mesmo que a memória Heap total disponível seja suficiente. Algumas linguagens, como Java e C#, conseguem minimizar esse problema fazendo a

gerência automática do Heap, utilizando algoritmos de coleta de lixo (*garbage collectors*). Também é possível, em C e C++, utilizar *frameworks* ou bibliotecas de gerência de Heap obtidos na Web ou construídos pelo programador. O Heap pode ser visto como um *array* de *Words* (i.e., unidade de dados natural de uma arquitetura de hardware). Nesse contexto, dois atributos são importantes: o *heapbotton* (início do Heap) e o *heaptop* (topo ou fim do Heap).



- Roots: em se tratando de algoritmos de gerência de Heap, eventualmente é necessário identificar ou saber quais são as variáveis em uso em determinado instante. Tal identificação é feita tendo-se como base um conjunto de variáveis ou entidades denominadas de *roots* (raízes). Os *roots* são, normalmente, as variáveis locais presentes na pilha, variáveis globais e atributos de classes, desde que façam referência a algum objeto/instância. Pode-se dizer que os roots são as variáveis consideradas como estando sempre "vivas".
- Objetos, nodos, células: elementos armazenados no Heap, eventualmente consistindo em *arrays* contíguos de *Words*, divididos em campos, em especial um ponteiro e um dado atômico (um átomo é um objeto que não possui ponteiro). Para efeitos didáticos, consideraremos somente elementos simples (i.e., ocupando um único Word e apontando para um único filho), mas, na prática, os nodos (ou células) podem conter diferentes tipos de dados e apontar para *n* outras células. Em um dado instante, podemos "enxergar" o Heap como um grafo de elementos nele armazenados, onde cada objeto é um nodo que pode, eventualmente, apontar para outros nodos (objetos). Dependendo da situação (da relação entre os objetos), diferentes subgrafos estarão sobrepostos no Heap.



No exemplo acima, dois *roots* apontam para diferentes objetos no Heap (nodo A e nodo B). Tais objetos são compostos de outro objeto (nodo C), que é comum aos anteriores (tanto A quanto B apontam para C), criando grafos sobrepostos.

- Dangling reference: uma referência que incorretamente "aponta" para uma célula de memória desalocada.
- Memory leak: também chamado de *space leak*. Consiste em uma área do Heap que não pode ser utilizada, mas não está em uso.

Depois de termos revisado tais conceitos, podemos estudar os algoritmos de coleta de lixo. Tais algoritmos são descritos a seguir.

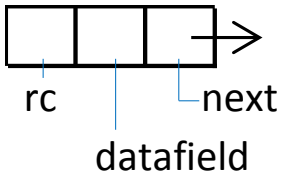
2 REFERENCE COUNTING

O *Reference Counting* representa uma família de algoritmos que conta o número de referências que apontam para cada célula de memória, a partir de outras células ativas ou *roots*. Tal algoritmo foi inicialmente desenvolvido para

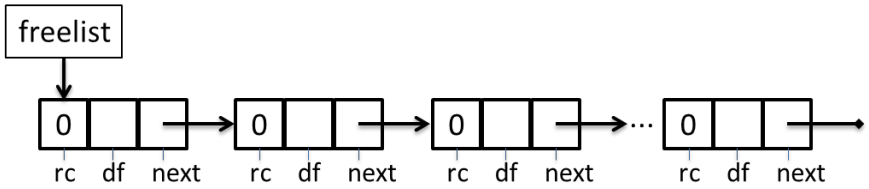
o LISP, mas também foi utilizado em várias versões iniciais de linguagens, entre elas Smalltalk, Modula e InterLisp, além do aplicativo Adobe Photoshop e em sistemas Unix para determinar arquivos deletados.

A ideia é simples. Toda a célula de memória possui um campo específico (o *reference count*) para armazenar a quantidade de referências que apontam para ela. Tal campo é atualizado a cada operação de manipulação de memória (e.g., alocar, liberar, realocar...). Com isso, o gerenciamento do *Heap* é feito a cada operação (que se torna, portanto, mais custosa). No entanto, o *overhead* acaba sendo distribuído ao longo do programa.

A figura seguinte apresenta um nodo típico que representa um objeto ou célula alocado na memória Heap gerenciada pelo algoritmo de *Reference Counting*. Tal nodo é uma versão simplificada, que contém um campo para contar a quantidade de referências a ele (*rc*), um campo de dados (*datafield*) e um ponteiro para o próximo (*next*). No caso, é como se os dados alocados no Heap só pudessem ser deste tipo, que é, na verdade, um elemento de uma lista encadeada. Tal estrutura foi adotada porque os algoritmos de gerência de Heap ficam mais fáceis de serem explicados quando os objetos nele alocados possuem um tamanho fixo e simples.



A memória Heap, nesta versão simplificada, pode ser vista como uma lista encadeada de células livres, todas de mesmo tamanho, com o *rc* delas inicializado em zero. A lista possui uma cabeça, denominada de *freelist*, que aponta ou referencia o primeiro elemento da lista de células livres. Toda vez que for necessário alocar uma célula de memória, ela é retirada dessa lista. Quando não houver mais referências a uma célula de memória, ela volta pra essa lista de células livres.



Com base nesse modelo de memória simplificado, onde todas as células são idênticas e possuem tal estrutura, é possível pensar nas operações básicas de gerenciamento de memória. Elas são descritas a seguir:

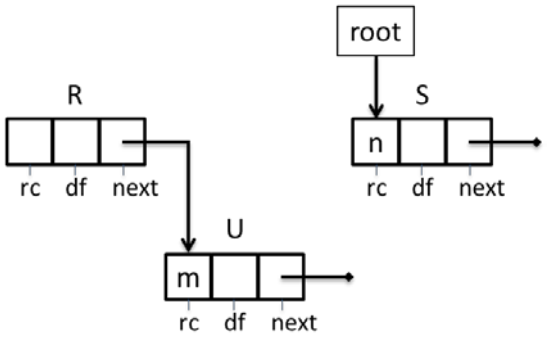
- new(): aloca uma célula de memória e retorna o seu endereço. Para tanto, remove um elemento da *freelist* e incrementa seu *rc*.
- rc(T): retorna o conteúdo do campo rc (*reference count*) de um nodo T.
- delete(T): remove uma referência ao nodo T. Caso não existam mais referências ao nodo T, ele é liberado, voltando para lista de livres (ver operação *free*, abaixo).
- update(R, S): atualiza uma referência R, fazendo-a apontar para S. Funciona como uma atribuição/substituição.
- next(T): retorna o endereço da célula apontada pelo nodo T.
- free(T): zera o *rc* de uma célula e a devolve para a lista de livres.

Possíveis algoritmos para algumas das operações citadas acima são apresentados a seguir, adaptados de Jones e Lins (1996), utilizando pseudocódigo:

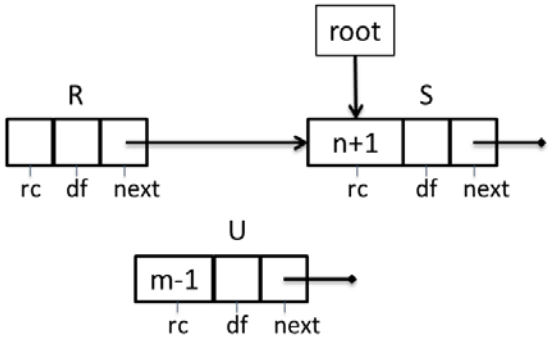
```
new()       => if empty(freelist)
                   abort "Memory full"
               newcell = allocate()
```

```
                RC(newcell) = 1
                return newcell

allocate()  => newcell = freelist
               freelist = next(freelist)
               return newcell

free(N)     => next(N)=freelist
               freelist = N

delete(T)   => rc(T) = RC(T)-1
               if rc(T) == 0
                   delete( next(T) )
                   free(T)

update(R,S) => delete(*R)
               RC(S) = RC(S) + 1
               *R = S
```

Com tais algoritmos, imagine 3 nodos: R, T e S, com a seguinte configuração, onde *m* e *n* são os valores atuais dos contadores de referência de U e de S, respectivamente:



Imagine agora que o programador desejasse fazer com que o R apontasse para S (i.e., update(next(R), S)). O resultado de tal operação é apresentado na figura seguinte.



2.1 ANÁLISE DO REFERENCE COUNTING

O *Reference Counting* tradicional funciona mais ou menos como apresentado acima. O que muda é que os nodos são um pouco mais complexos em situações reais, mas a ideia básica é a mesma.

O maior problema deste algoritmo é que há um alto custo associado a cara operação de manipulação de memória, visto que os contadores devem ser atualizados em toda operação, seja ela de alocação, desalocação ou atualização

(*update*). Em especial na operação de *update*, tanto o contador da referência antiga quanto da nova devem ser atualizados. Além disso, toda célula deve guardar seu contador de referências, o que acarreta certo custo de espaço também.

O *Reference Counting* é muito atrelado ao programa que o utiliza e/ou compilador, visto que as atualizações devem ser feitas sempre que ponteiros ou referências são passadas como parâmetro para subprogramas ou retornadas.

Outro problema importante é que o *Reference Counting* apresenta problemas quando há referências cíclicas.

É importante salientar que, segundo Jones e Lins (1996), estudos empíricos demonstraram que poucas células de memória são compartilhadas e que muitas possuem um tempo de vida muito curto. Com isso, tal algoritmo teria vantagem em relação a outros, visto que as células de memória podem ser utilizadas imediatamente após terem sido liberadas, causando, inclusive, menos *pagefaults*. Em outros algoritmos, o *Garbage Collector* executa o processo de liberação em momentos específicos e as células de memória não ficam disponíveis até que tal processo seja executado (mesmo que não estejam mais sendo mais utilizadas).

Outro benefício é que o *Reference Counting* facilita a implementação de blocos de finalização (e.g., *finally* em Java).

3 MARK-SWEEP

O *Mark-Sweep* (marcar e varrer/limpar) é um algoritmo que marca as células de memória em uso para depois liberá-las. Ele também é conhecido por algoritmo *Mark-Scan* (McCarty, 1960 apud Jones e Lins 1996). Também foi um algoritmo desenvolvido para a linguagem LISP. Atualmente é utilizado por linguagens com coletores de lixo conservadores em termos de gerência de memória, como Miranda.

Neste algoritmo, as células-lixo não são liberadas imediatamente. Elas permanecem inalcançáveis e indetectáveis até que o espaço disponível no Heap acabe. Quando isso ocorre, e uma alocação de memória for solicitada, o processamento útil é suspenso e uma rotina de limpeza (coleta de lixo) é executada. Tal rotina consegue identificar as células de memória que não estão mais em uso e coloca-as de volta na lista de células livres.

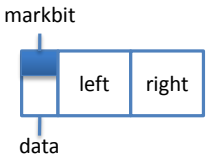
Para tanto, o algoritmo de limpeza realiza uma travessia global de todos os objetos “vivos” a fim de determinar as células que estão e que não estão em uso. Identificar objetos vivos consiste em identificar as células acessíveis, ativas. Todo o resto é considerado lixo e vai para a lista de células livres. Tal identificação é feita com base nas variáveis *roots*. As células precisam, portanto, serem marcadas. Com isso, um *flag* deve ser mantido em cada uma delas – o *markbit*:

Seguem os algoritmos principais dessa categoria de *Garbage Collector*, adaptados de Jones e Lins (1996):

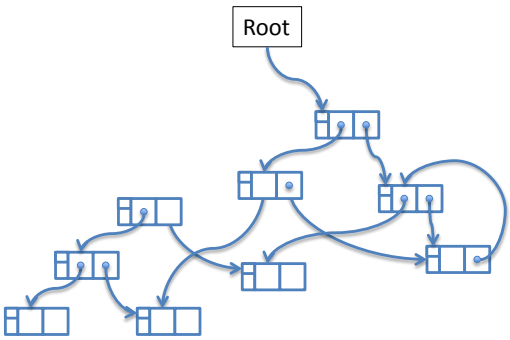
```
new()       => if empty(freelist)
                   marksweep()
               newcell = allocate()
               return newcell

marksweep() => for R in Roots
                   mark(R)
               sweep()
```

```
                    if empty(freepool)
                        abort “memory is full”

mark(T)         => if markbit(T) == unmarked
                        markbit(T) = marked
                        mark(*next(T))

sweep()         => N = heapbotton
                    while N < heaptop
                        if markbit(N) == unmarked
                          free(N)
                        else
                          markbit(N) = unmarked
                        N = N + size(N)
```

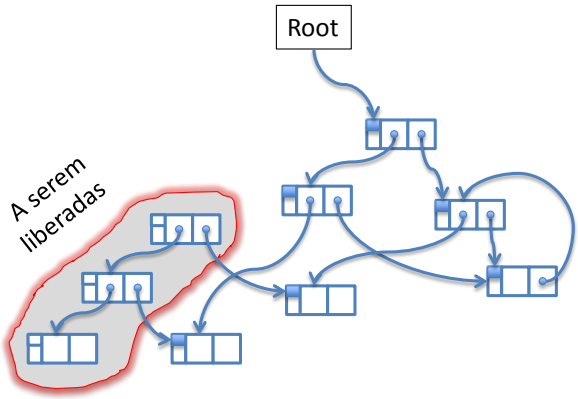
A seguir é apresentado um exemplo de aplicação do algoritmo *Mark-Sweep*. Para tanto, vamos considerar uma nova estrutura de nodo:



Nessa estrutura, os objetos representam nodos de uma árvore binária, ou seja, possuem um filho esquerdo (*left*) e um filho direito (*right*). Com base nessa estrutura, suponha que, em determinado momento, a seguinte configuração seja encontrada na memória Heap:



Como resultado do algoritmo de marcação (*mark*), as células assinaladas na figura seguinte seriam liberadas por não terem sido marcadas, visto que não há acesso a eles a partir de uma variável “viva” válida (root).



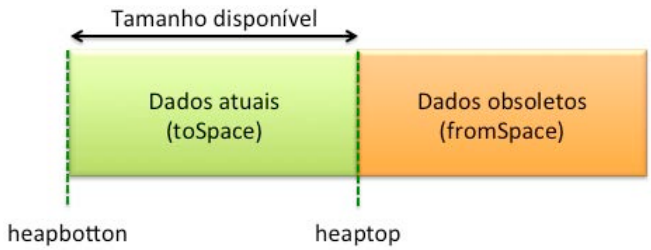
3.1 ANÁLISE DO MARK-SWEEP

Esse algoritmo é interessante porque a fase de limpeza (*sweep*) varre linearmente o Heap (do fim para o topo) recuperando as áreas marcadas para a lista de livres (e desmarcando as demais). Ao fazer isso, naturalmente os ciclos são gerenciados. Além disso, não há *overhead* nas operações de manipulação de ponteiros/referências.

Um dos problemas é que o algoritmo, ao menos na forma simples como foi apresentado, gera fragmentação de memória. Além disso, a computação realizada pelo programa deve parar enquanto o algoritmo é executado. Tais paradas podem, eventualmente, ser substanciais. E.g., segundo Jones e Lins (1996), programas LISP perdiam entre 20% e 40% do seu tempo realizando *sweeping*. Por este motivo, seu uso não é indicado para sistemas distribuídos e de processamento em tempo real.

4 COPYING COLLECTOR

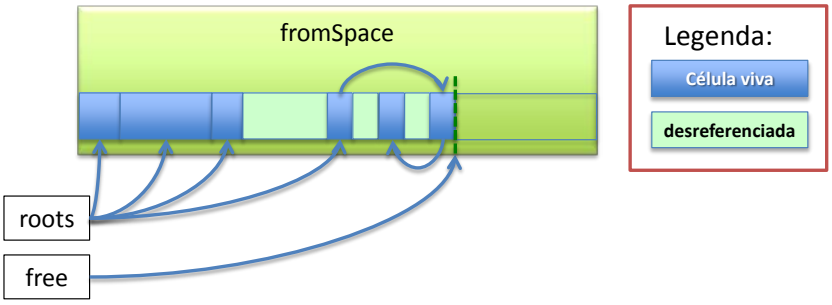
Os algoritmos baseados em cópia são também chamados de *Scavengers*. Neste caso, a memória Heap é dividida em duas partes – *toSpace* e *fromSpace*, sendo que somente uma delas é válida para alocação de memória (i.e., o *toSpace*), conforme ilustrado abaixo:



No *toSpace* são armazenados/alocados os elementos, da esquerda para a direita. O tamanho dele varia entre o *heapbotton* (início do Heap) e o *heaptop* (topo do Heap). O *fromSpace* fica intacto e não serve para alocação. Os elementos desreferenciados continuam em memória e não se faz nada com eles até que o *toSpace* fique cheio. Uma vez que o *toSpace* esteja cheio e seja solicitada uma alocação de memória, a execução do programa é suspensa e o *garbage collector* entra em ação.

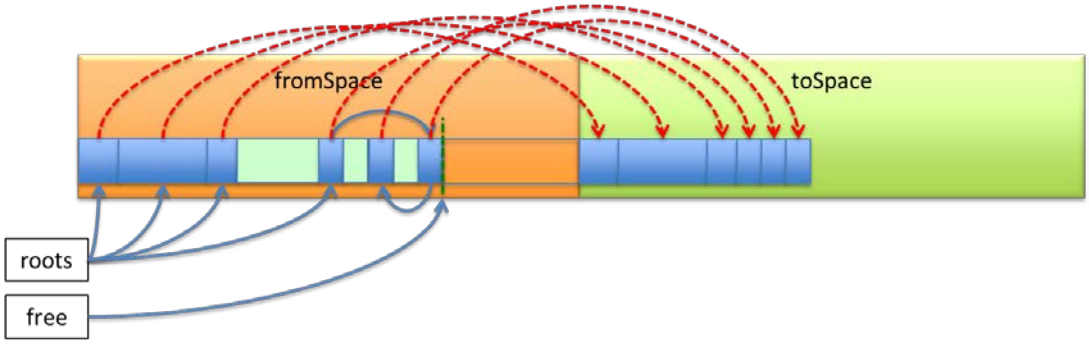
O algoritmo funciona da seguinte forma: os papéis do *toSpace* e do *fromSpace* são invertidos (um passa a ser o outro). Após, o coletor atravessa a estrutura de dados ativa no espaço velho (*fromSpace*), copiando toda célula viva (as que possuem alguma referência, tendo como base os *roots*) para o *toSpace*. Depois que todas as células ativas tenham sido visitadas (uma única vez basta), o Heap terá sido reconstruído de forma a conter somente células vivas, reorganizadas no novo espaço. O espaço livre estará desfragmentado e pode ser utilizado para novas alocações. O programa pode ser, então, reativado.

Mas vamos por partes. A figura seguinte demonstra um snapshot de memória em determinado momento.

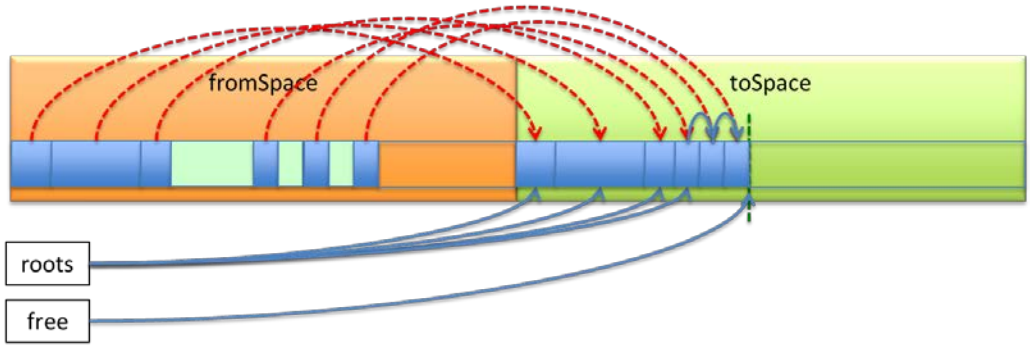


É possível perceber que o Heap disponível para o programador é o espaço do *toSpace*. Também é possível verificar que algumas células de memória estão sendo referenciadas por variáveis do tipo *root*, e algumas delas estão interconectadas. Essas células são ditas células vivas e devem ser mantidas em uma eventual execução do algoritmo de coleta de lixo. Há ainda algumas células de memória previamente alocadas, mas não mais referenciadas. Na figura, elas estão entre algumas das células vivas. Tais células podem ser liberadas pelo algoritmo de coleta. Verifica-se, ainda, que a região livre é facilmente mantida, bastando um ponteiro para onde ela inicia (*free*). Nessa estrutura, pelo fato da região livre estar sempre à direita dos objetos já alocados, repare que a operação de alocação é muito simples, pois basta alocar espaço necessário à direita do ponteiro para a área livre, que é contínua.

Quando a limpeza inicia, o *toSpace* é invertido com o *fromSpace*. Em seguida, todas as variáveis vivas são copiadas para o *ToSpace*, como no exemplo a seguir:



Repare que, a partir dos roots, as referências são seguidas e copiadas na ordem em que aparecem. O *toSpace* fica organizado e o espaço livre é recuperado. É importante atualizar os ponteiros originais dos roots durante o processo. A figura anterior não considerou isso. Na verdade, a cada cópia, o ponteiro original é substituído pelo ponteiro da célula no novo espaço. Ao final, o Heap fica organizado como apresentado na próxima figura, e o espaço disponível para o Heap passa a ser o novo *ToSpace*. Tal inversão é realizada toda vez que o algoritmo de limpeza executa, havendo uma alternância entre os espaços.



A seguir encontram-se os algoritmos das principais operações relacionadas com esta abordagem, conforme especificados por Jones e Lins (1996):

```
init()      => toSpace = heapbotton
               spacesize = heapsize / 2
               topofspace = toSpace + spacesize
               fromSpace = topofspace + 1
               free = toSpace + 1

new(n)      => if free+n > topofspace flip()
               if free+n > topofspace abort “memory full”
```

```
                newcell = free
                free = free + n
                return newcell

flip()     => fromSpace = toSpace
              toSpace = fromSpace
              topofspace = toSpace + spaceSize
              free = toSpace
              for R in Roots
                R = copy(R)

copy(P)    => if atomic(P) or P == nil
                return P
              if not forwarded(P)
                n = size(P)
                P'=free
                free = free+n
                temp=P[0]
                forwardingAddress(P)=P'
                P'[0]=copy(temp)
                for i=1 to n-1
                  P[i]=copy(P[i])
              return forwardingAddress(P)
```

Uma atenção especial deve ser dada à operação de cópia (*copy*). Ela deve analisar se o conteúdo do objeto a ser copiado para o toSpace é atômico ou não. Caso o seja, basta retornar seu conteúdo para que possa ser colocado no novo local. Caso não seja atômico (i.e., é uma referência a outra célula), tal referência deve ser também atualizada para o novo local. Caso o elemento referenciado por ela já esteja na nova região, seu endereço é atualizado para corresponder ao novo local. Caso ele não na nova região, ele deve ser deslocado para lá, incluindo o seu conteúdo (de maneira recursiva). O endereço novo (i.e., *forwardingAddress*) é então retornado. *ForwardingAddress* é, portanto, o endereço da cópia, no espaço novo.

4.1 ANÁLISE DO COPY-COLLECTOR

Esse algoritmo é amplamente adotado pelas linguagens de programação, eventualmente com algum refinamento ou combinação com outro algoritmo. Nele, o custo de alocação é baixo, visto que a memória é alocada bastando incrementar o ponteiro para o espaço livre (no tamanho do objeto alocado), visto que este sempre estará do lado direito. A verificação do espaço disponível também é simples, envolvendo somente comparação entre ponteiros. Outra vantagem consiste no fato de que o algoritmo elimina a fragmentação, pois naturalmente compacta os elementos quando faz sua cópia. Finalmente, o algoritmo lida naturalmente com elementos de tamanho variável.

Suas desvantagens incluem o uso de dois espaços de memória, sendo que somente um deles está disponível em cada momento. Em máquinas com memória virtual isso não chega a ser problema. Além disso, o programa deve ser parado para a realização das cópias.

REFERÊNCIAS

Jones, Richard; Lins, Rafael. **Garbage Collection: Algorithms for automatic dynamic memory management**. John Wiley & Sons: Chichester, 1996. 377p.

Venners, Bill. *Garbage Collection*. In: **Inside the Java 2 Virtual Machine**, 2a edição. McGraw-Hill: [s.l.], 2000. Disponível em: < http://www.artima.com/insidejvm/ed2/gcP.html >. Acesso em: < 16/11/2011 >.