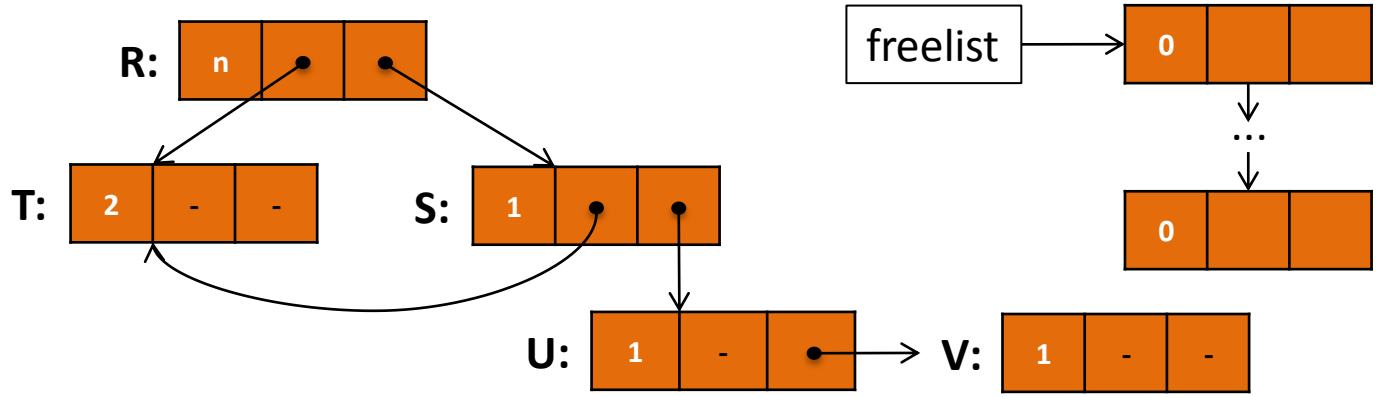
Exercício

- Considerando um gerenciador de Heap com algoritmo de coleta de lixo do tipo *reference counting* que manipule estruturas de dados com o seguinte formato:

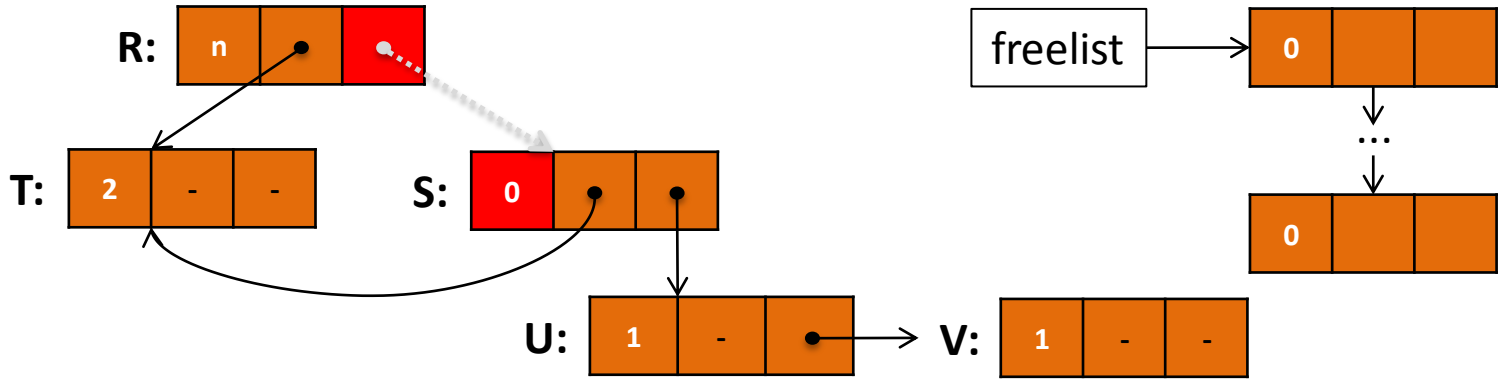
- Considerando ainda que o seguinte *snapshot* corresponda ao conteúdo atual do Heap:



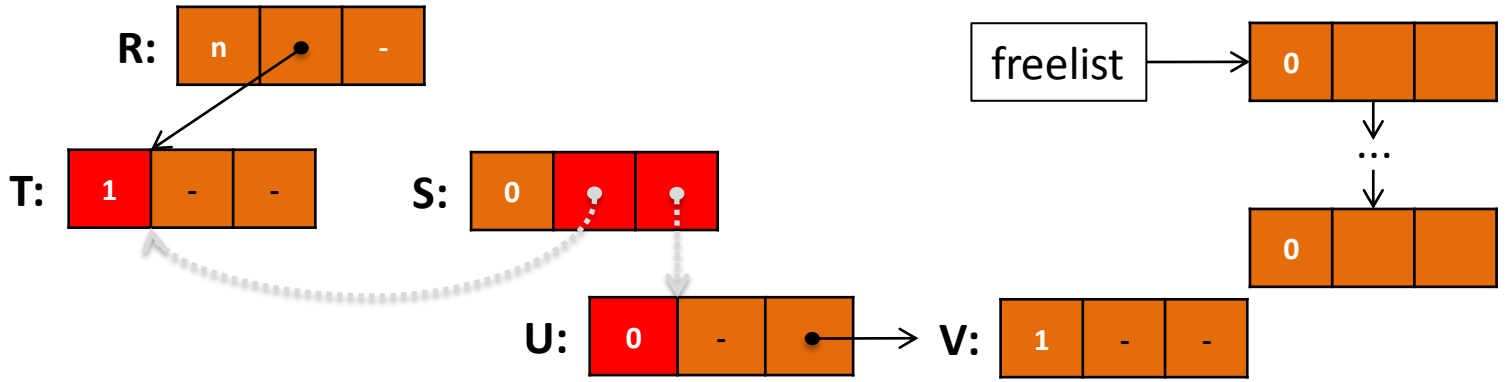
- Como ficaria o conteúdo do Heap após o comando: update(right(R), nil)?

Resposta

1) delete(right(R)):

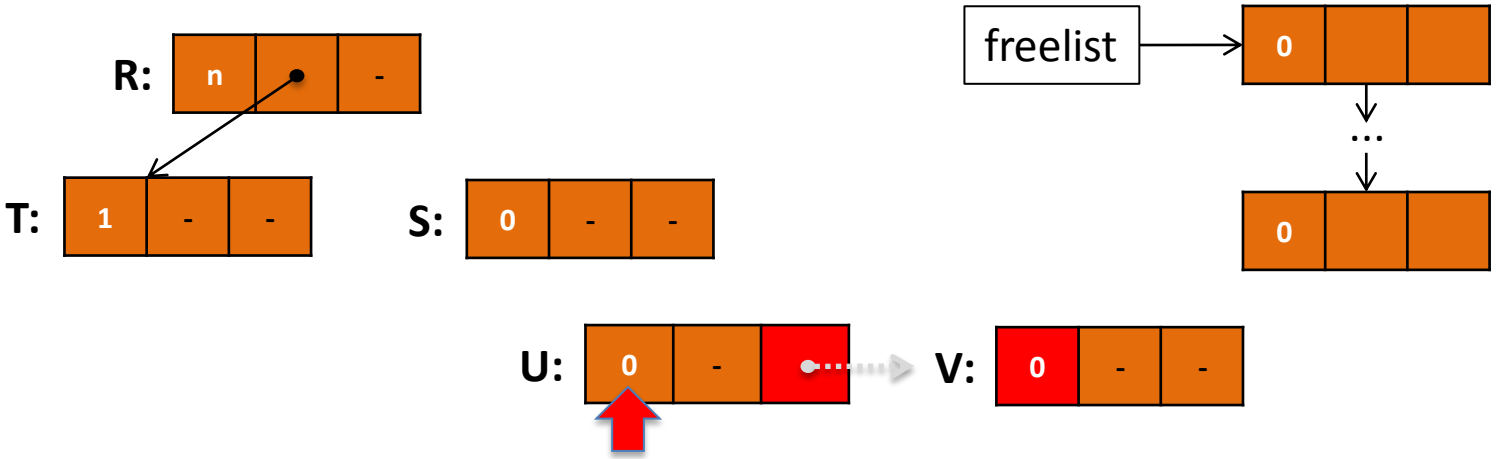


2) RC(S) == 0, então delete(childrens(S)) → delete(left(S)) e delete(right(S)):

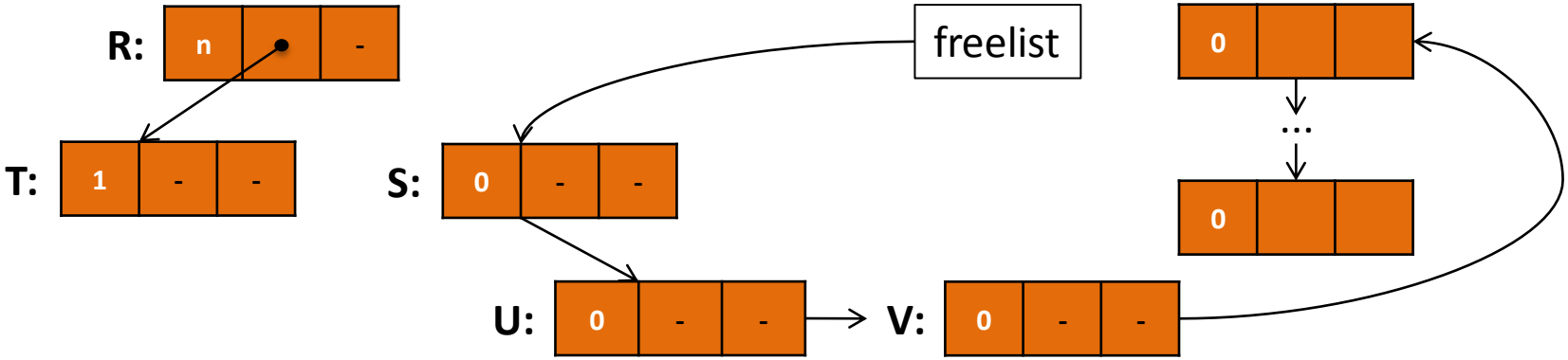


Resposta

2) (continuação) – recursivamente testa rc filhos e os apaga quando é zero:

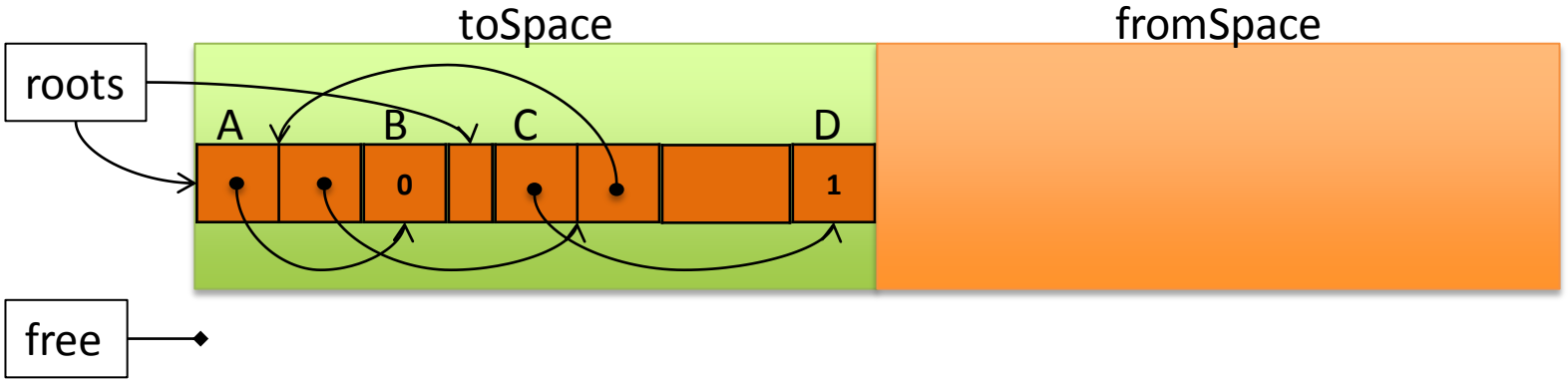


3) free(V), free(U), free(S):



Exercício

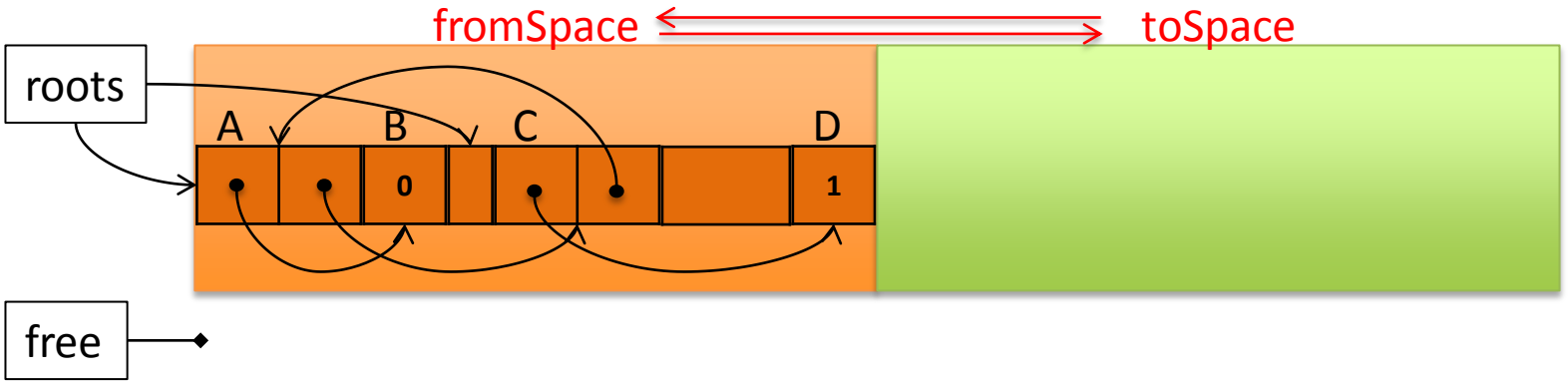
- Considere uma linguagem de programação que utilize um gerenciador de Heap com *copying collector* . Em tal linguagem, a execução de um programa gerou a seguinte configuração de memória Heap em dado momento, representando uma estrutura cíclica em um espaço finito (e.g., [0,1,0,1,...]:



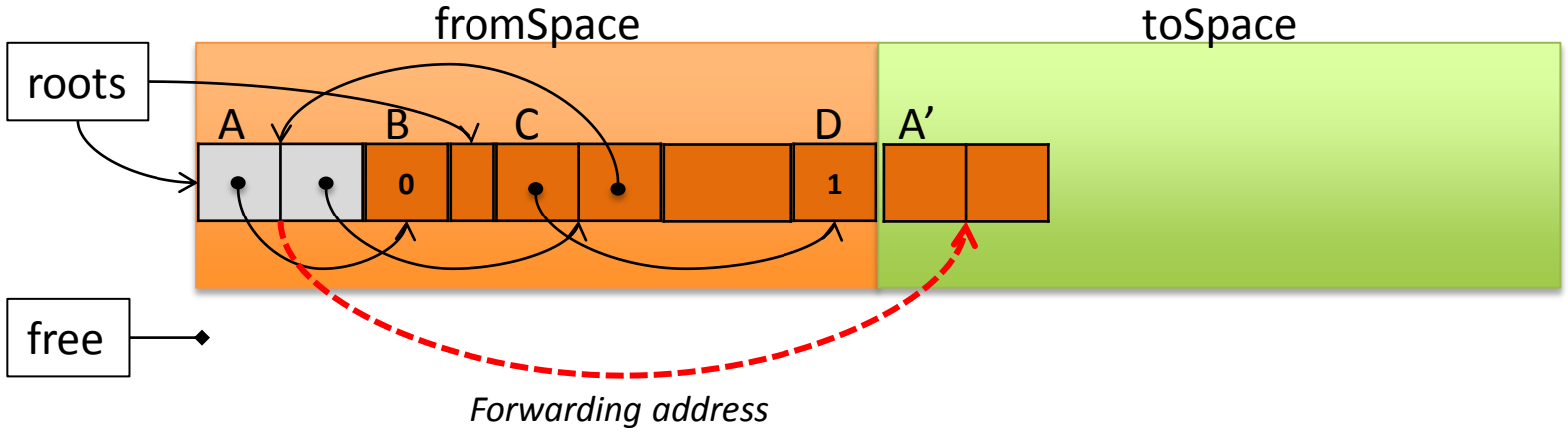
- É possível observar que não há espaço livre e que qualquer nova alocação provocaria a execução do *garbage collector*. Simule a execução de tal algoritmo, demonstrando como ficaria o snapshot de memória após a execução do *garbage collector*.

Resposta

1. Inverte os semiespaços:

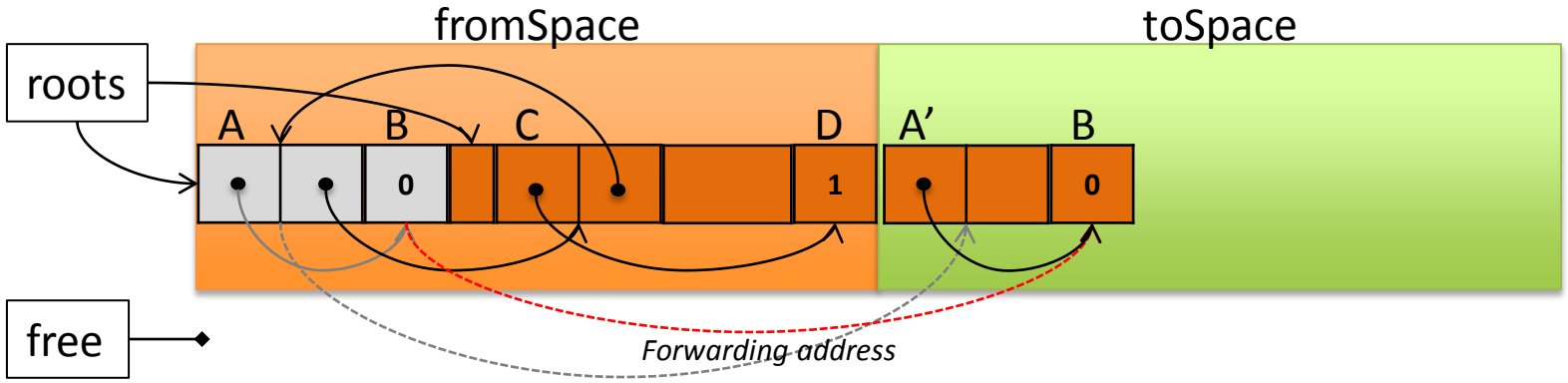


2. Inicia cópia pelos roots e anota *forwarding address*:

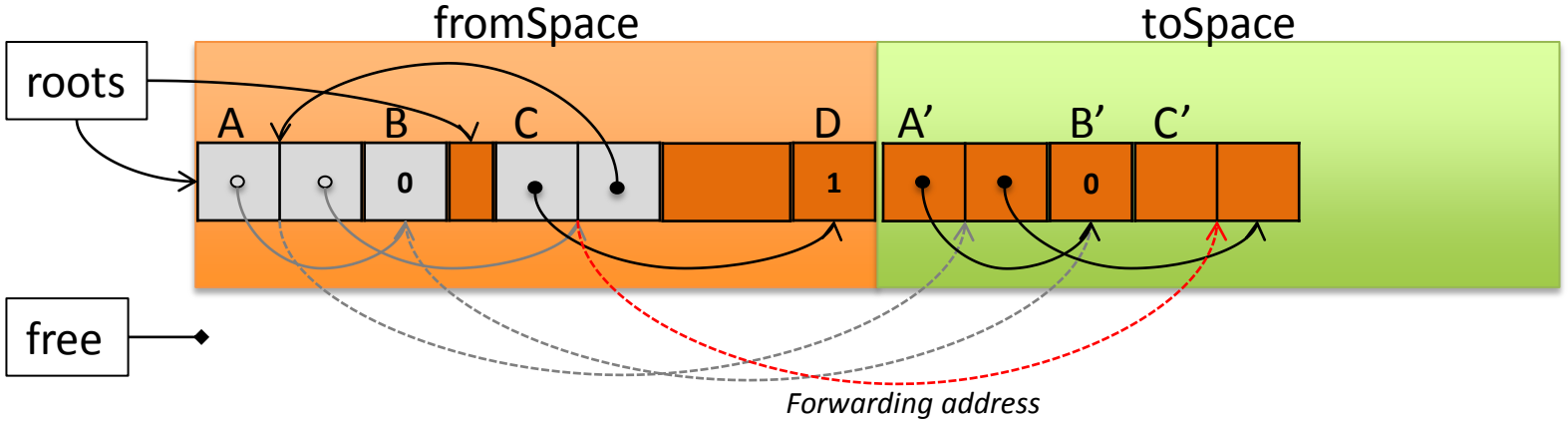


Resposta

3. Copia filho esquerdo, (a) reservando espaços e (b) atualizando endereço no novo pai:

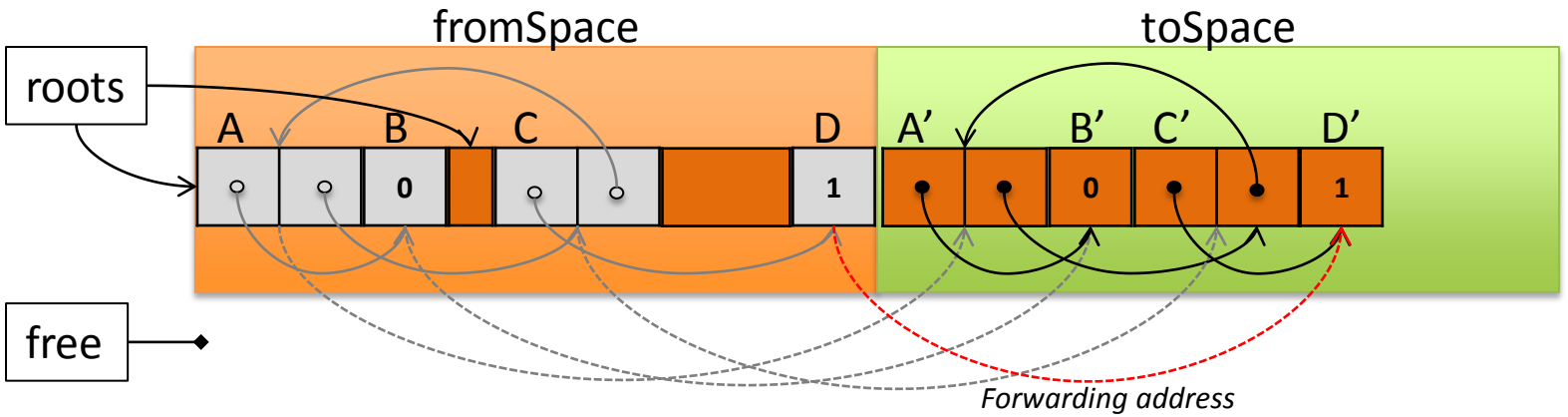


4. Copia filho direito, (a) reservando espaço e (b) atualizando endereço no novo pai

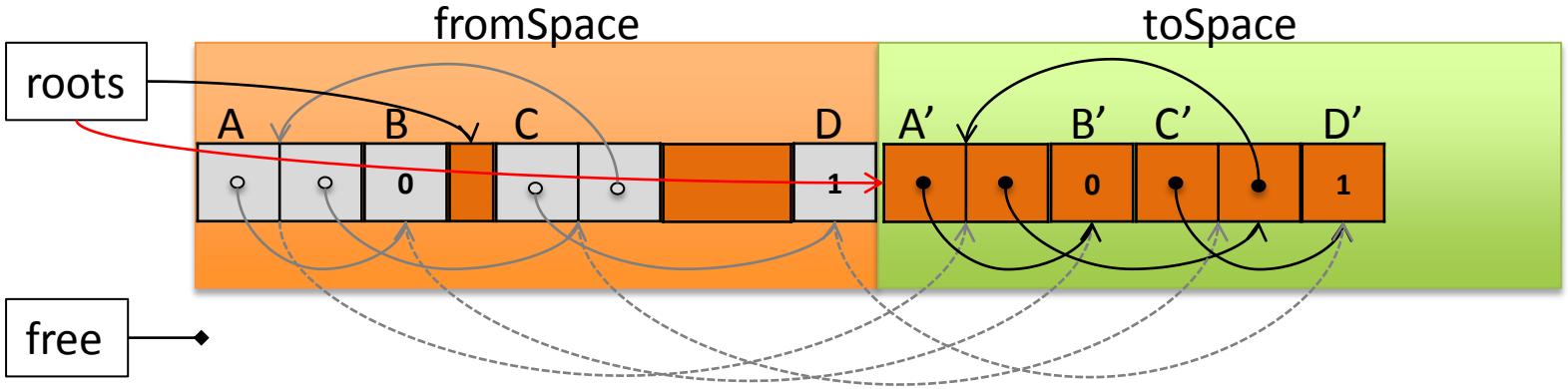


Resposta

5. Termina de copiar elementos restantes (recursivamente):

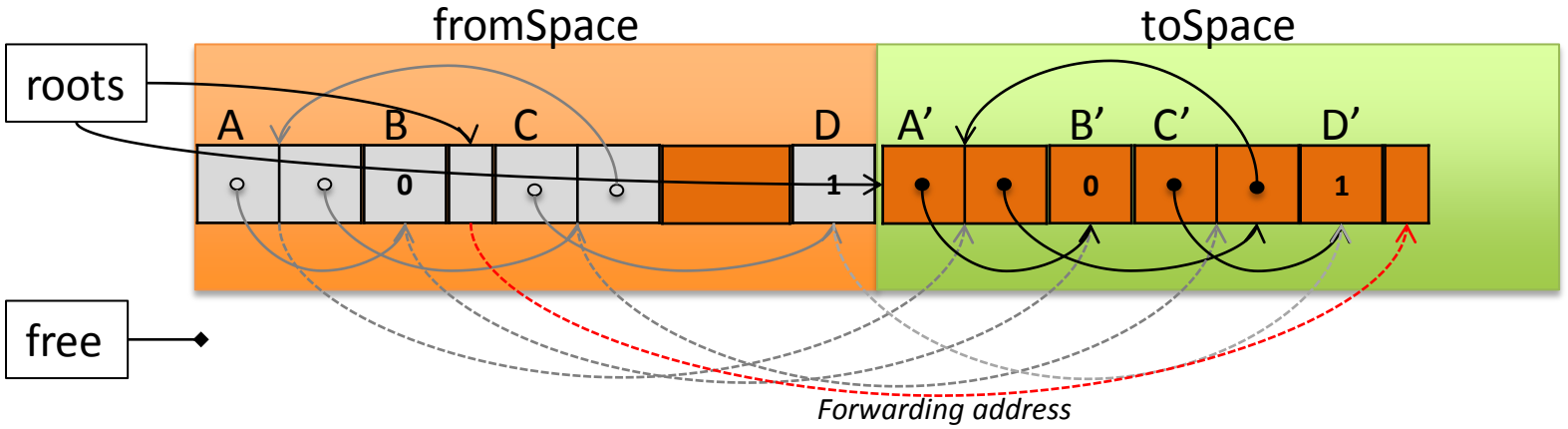


6. Ao terminar de copiar elemento (e seus filhos), atualiza roots:

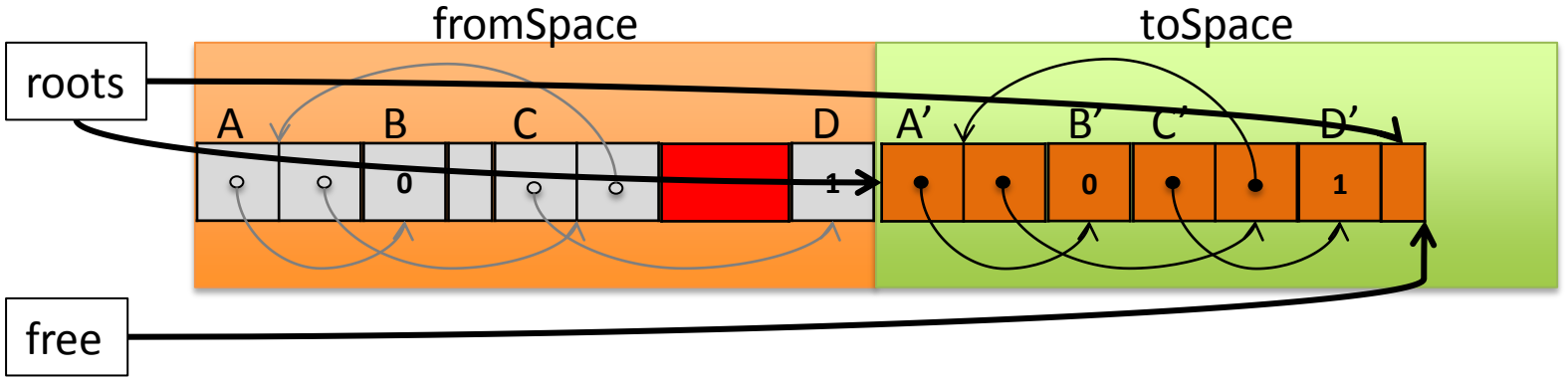


Resposta

7. Copia elementos restantes:



8. Ao final, roots e ponteiros internos estarão apontando para o novo toSpace, sem fragmentação. Free aponta para início do espaço livre:



Resposta

9. Quando terminar, a memória vista é o novo toSpace: