

# Gerência do Heap: Garbage Collector

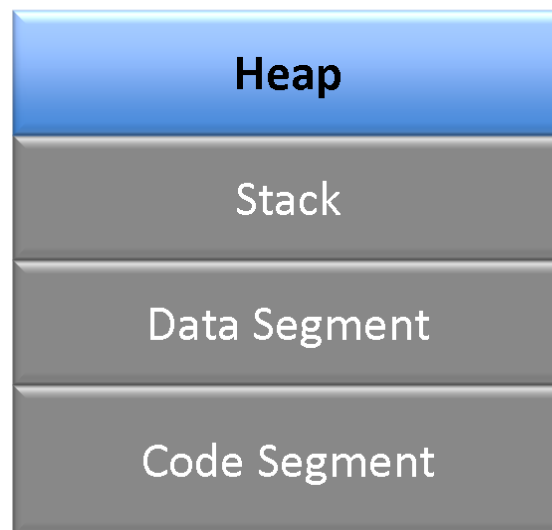
Modelos de Linguagens de  
Programação

# Contextualização

- Memória: recurso escasso e caro?
- Programas maiores e complexos, estruturas de dados e algoritmos genéricos:
  - gerenciar sua memória e evitar problemas pode dar bastante trabalho
- Algoritmos de Garbage Collection:
  - Diminuem a complexidade de programação
  - Abstraem preocupação com gerenciamento de memória
  - Diminuem quantidade de erros

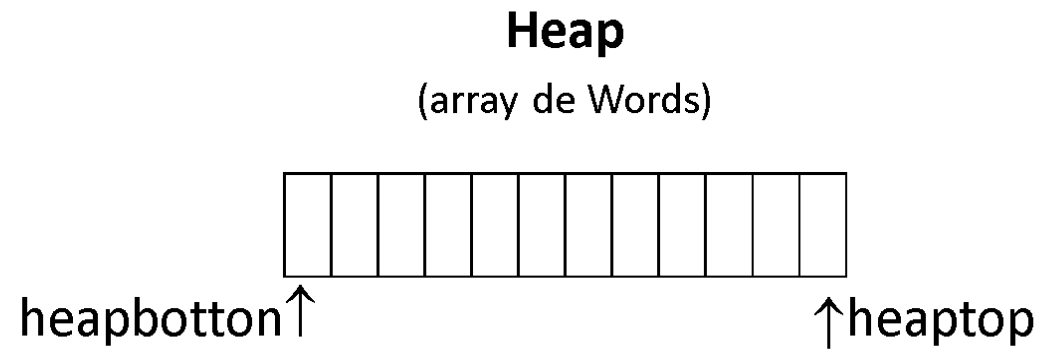
# Conceitos

- Modelo de Memória



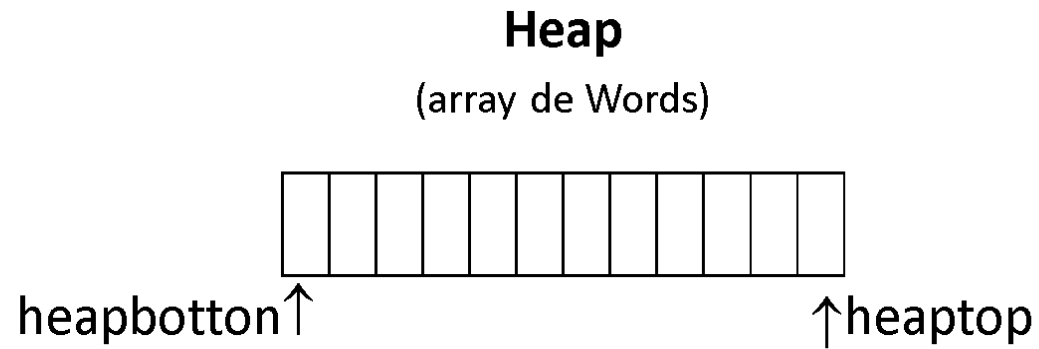
# Conceitos

- Heap



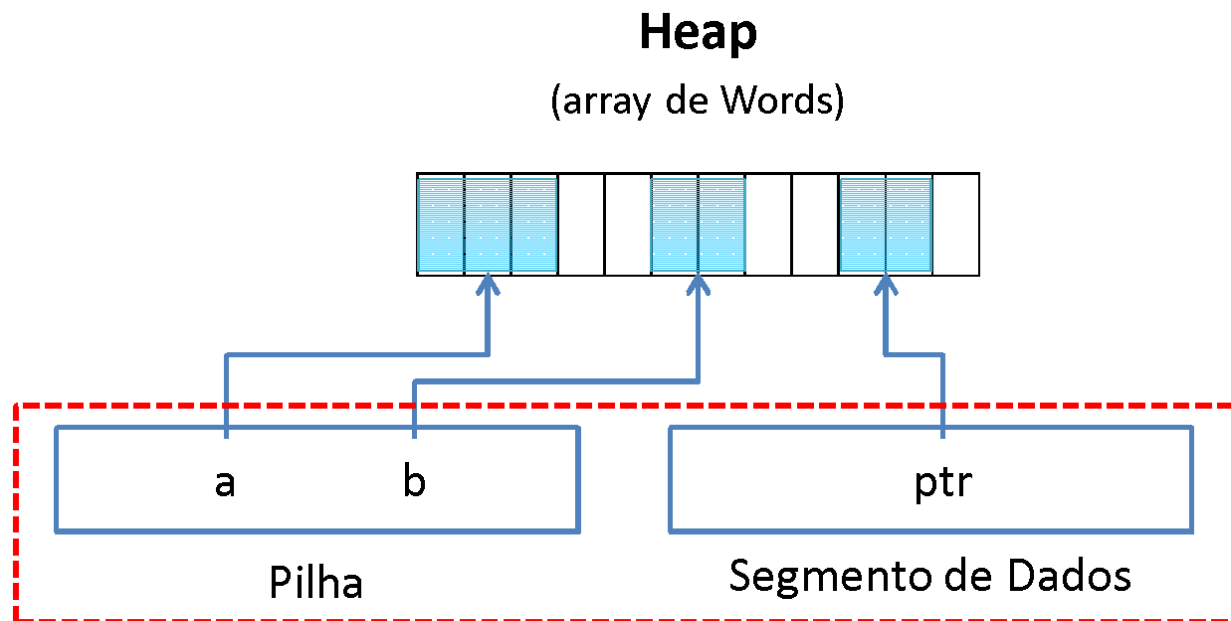
# Conceitos

- Heap

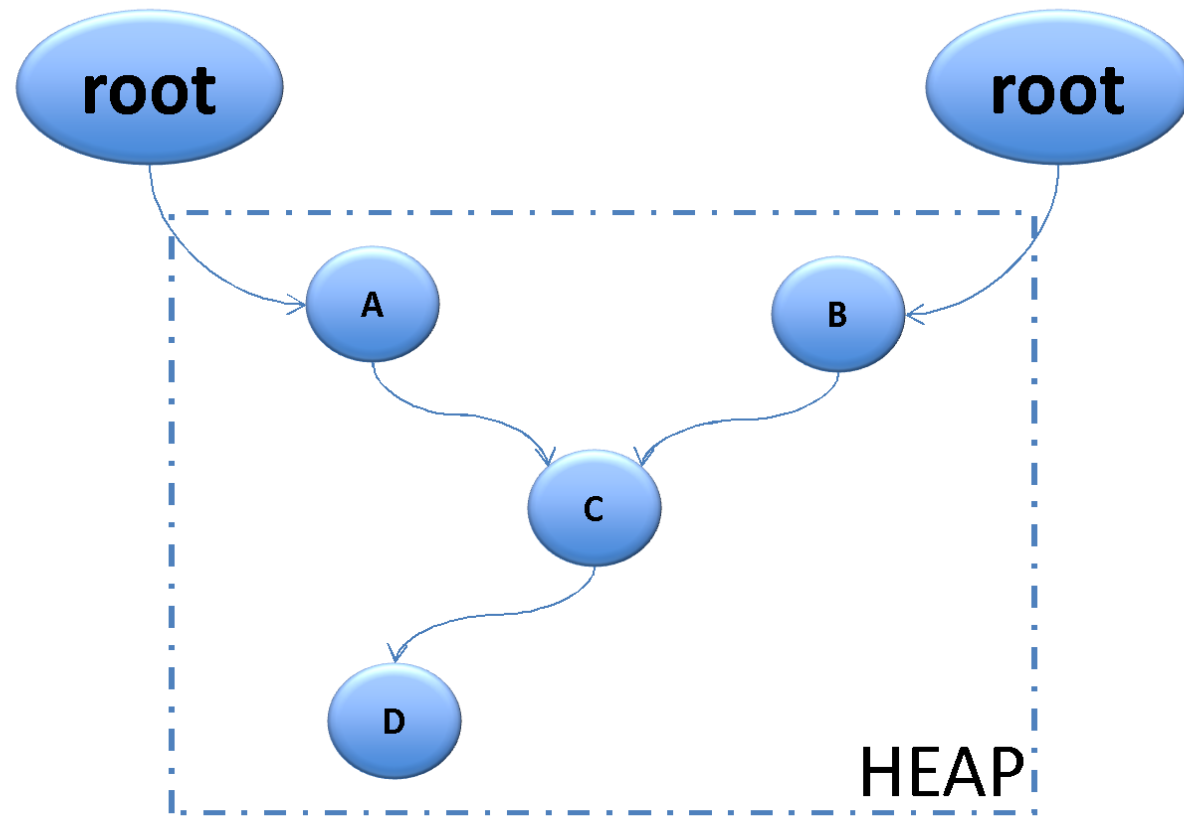


# Conceitos


- Roots



# Conceitos



# Principais Famílias de Algoritmos

- Reference Counting
  - Mark-Sweep
  - Copying Collector
- 
- Tracing



# Reference Counting

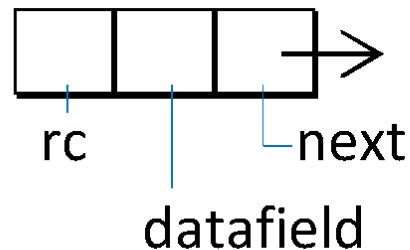
- Conta o número de referências que apontam para cada célula de memória, a partir de outras células ativas ou *roots*
- Inicialmente desenvolvido para LISP, mas também foi utilizado em várias versões iniciais de Smalltalk, Modula, etc.
- Usado por Python, SmartPointers (C, C++)

# Reference Counting

- 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...).

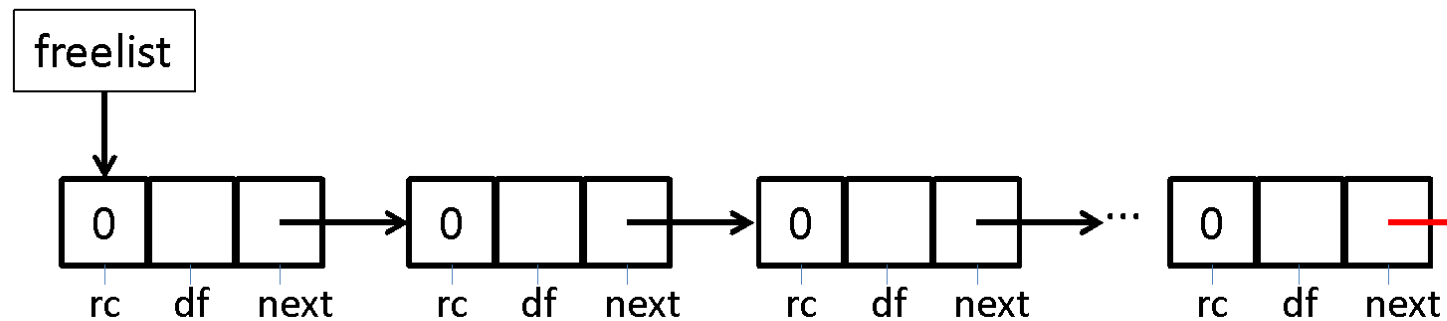
# Reference Counting

- Versão simplificada
  - Heap é uma lista encadeada de células livres, todas de mesmo tamanho e com estrutura fixa
  - Estrutura:



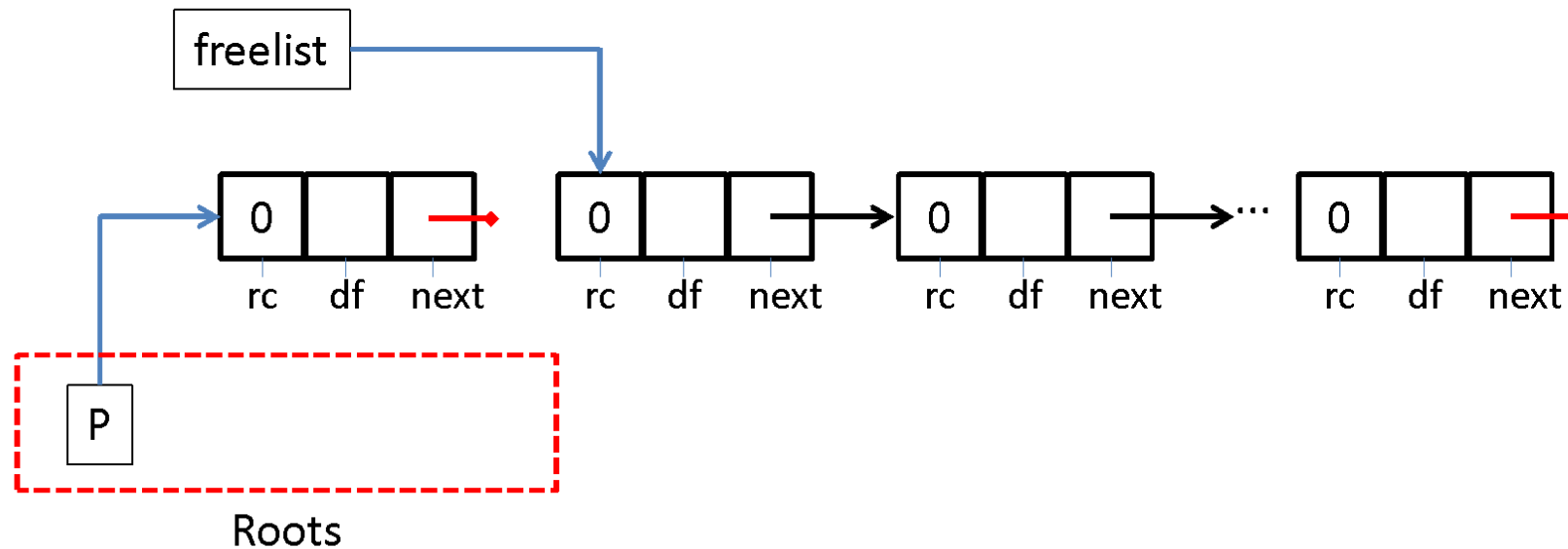
# Reference Counting

- Versão simplificada
  - Heap inicial:



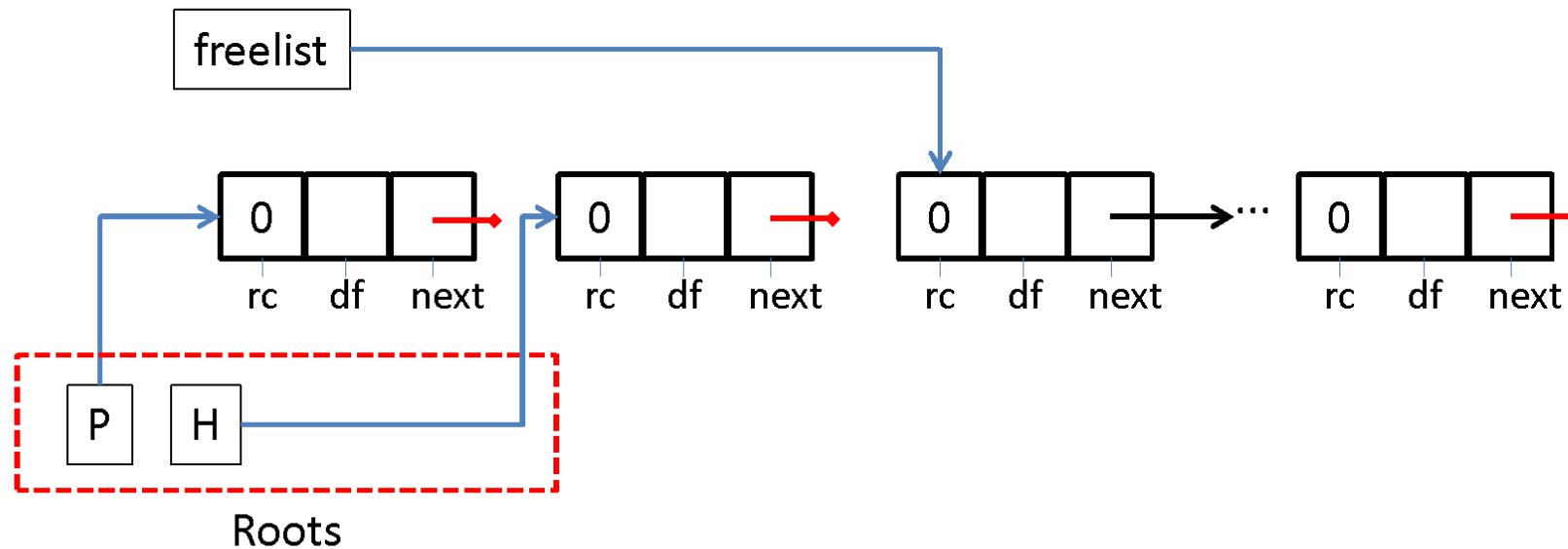
# Reference Counting

- Versão simplificada
  - $P = \text{new}()$



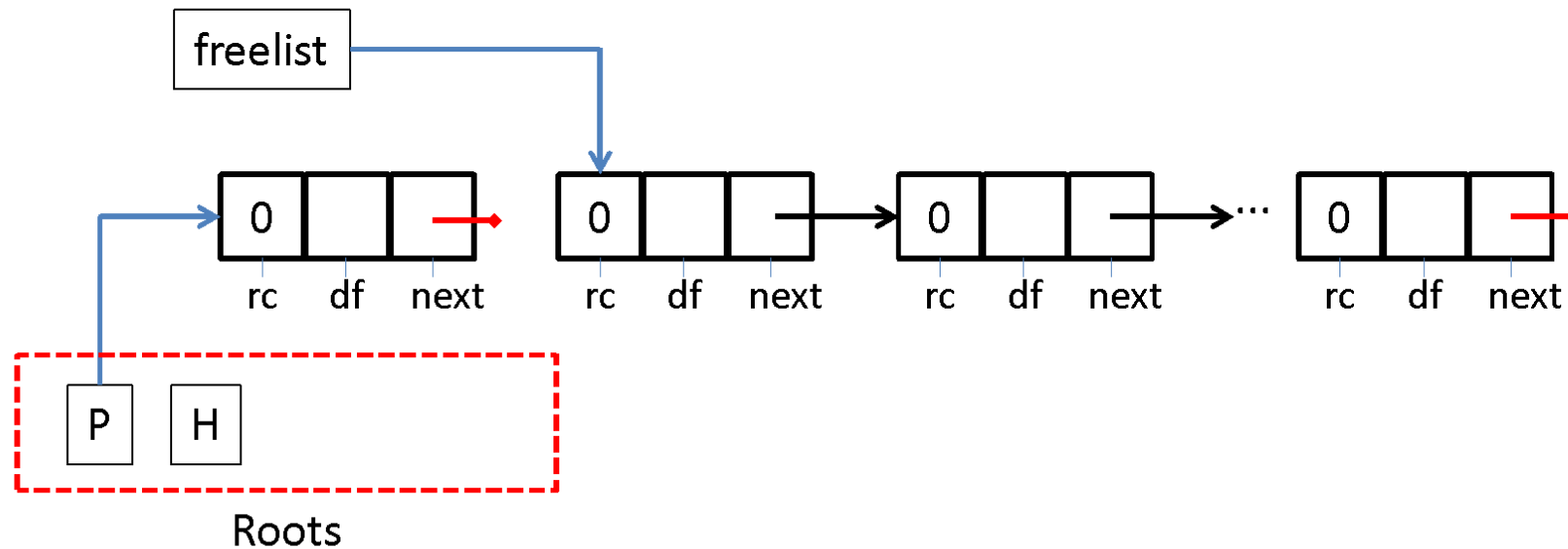
# Reference Counting

- Versão simplificada
  - $H = \text{new}()$



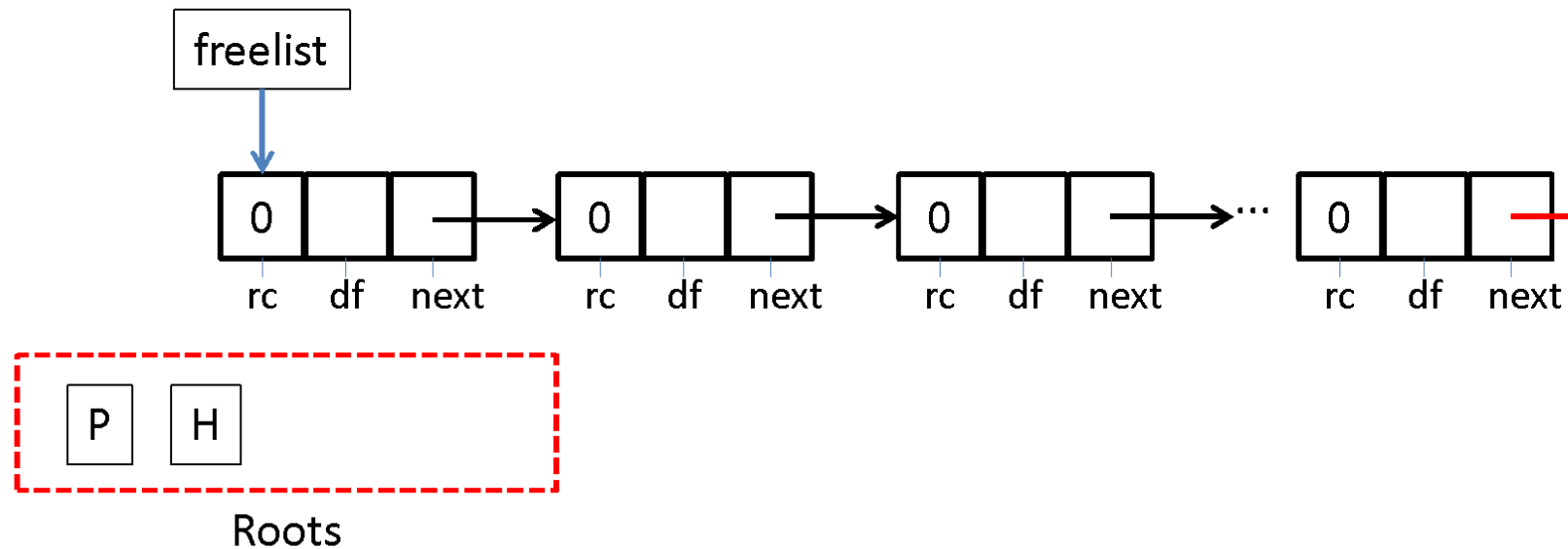
# Reference Counting

- Versão simplificada
  - delete(H)



# Reference Counting

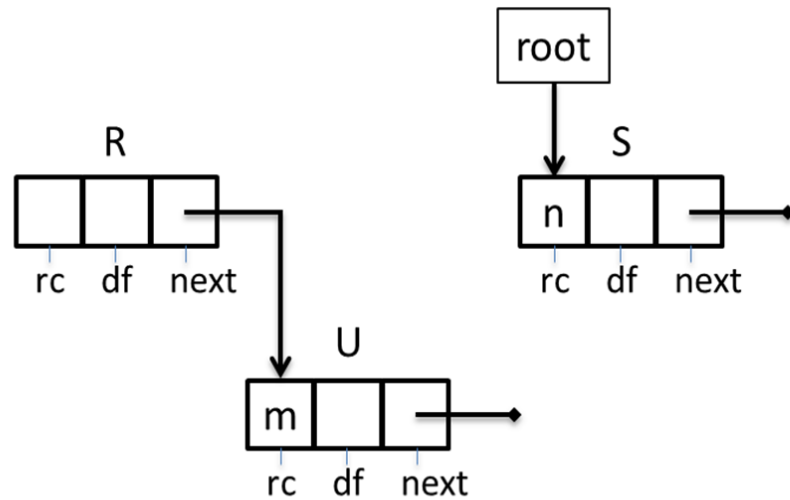
- Versão simplificada
  - delete(P)





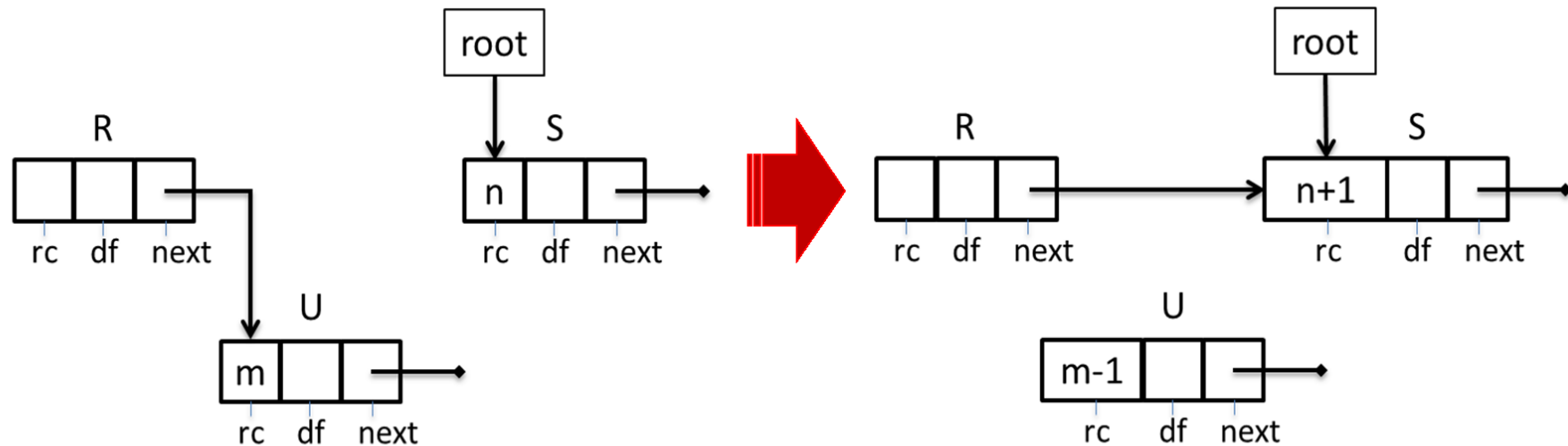
# Reference Counting

- Versão simplificada



# Reference Counting

- Versão simplificada
  - `update(next(R), S)`



# Reference Counting

- Versão simplificada: algoritmos

- 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

# Reference Counting

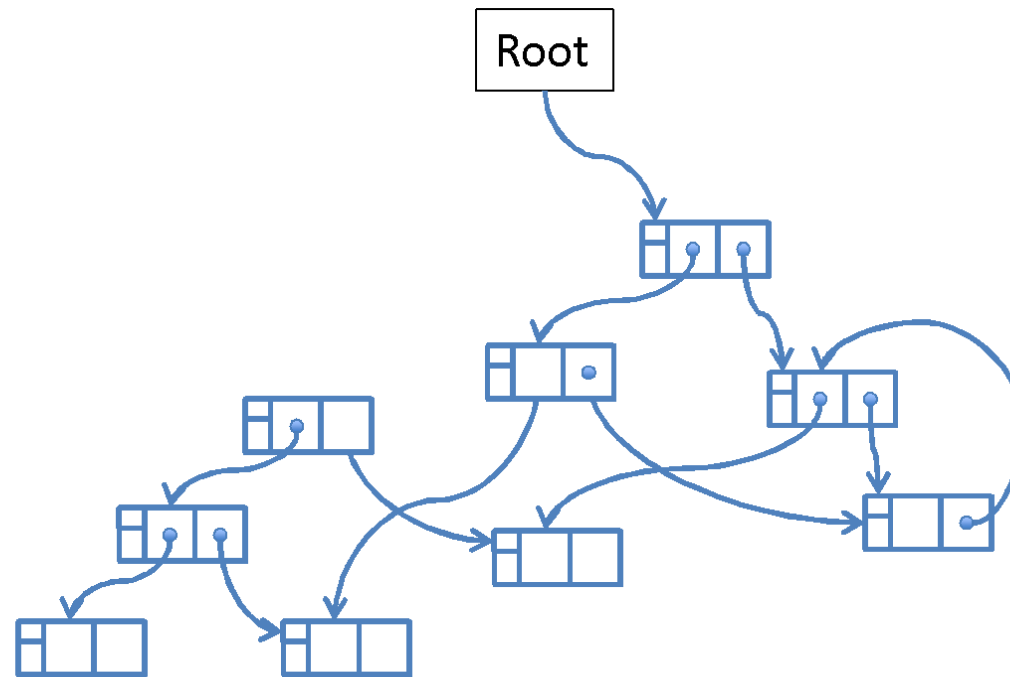
- Análise:
  - custo associado a cada operação de manipulação de memória (mas diluído no programa)
  - não lida com ciclos
  - células de memória podem ser utilizadas imediatamente após terem sido liberadas
  - menos *pagefaults*
  - facilita a implementação de blocos de finalização (Java)

# Mark-Sweep

- Algoritmo roda quando o Heap está cheio
  - as células-lixo não são liberadas imediatamente
  - permanecem inalcançáveis e indetectáveis até que o espaço disponível no Heap acabe
  - processamento útil é suspenso e uma rotina de limpeza (coleta de lixo) é executada
  - Duas fases:
    - Marcar
    - Limpar
- Também foi desenvolvido para LISP

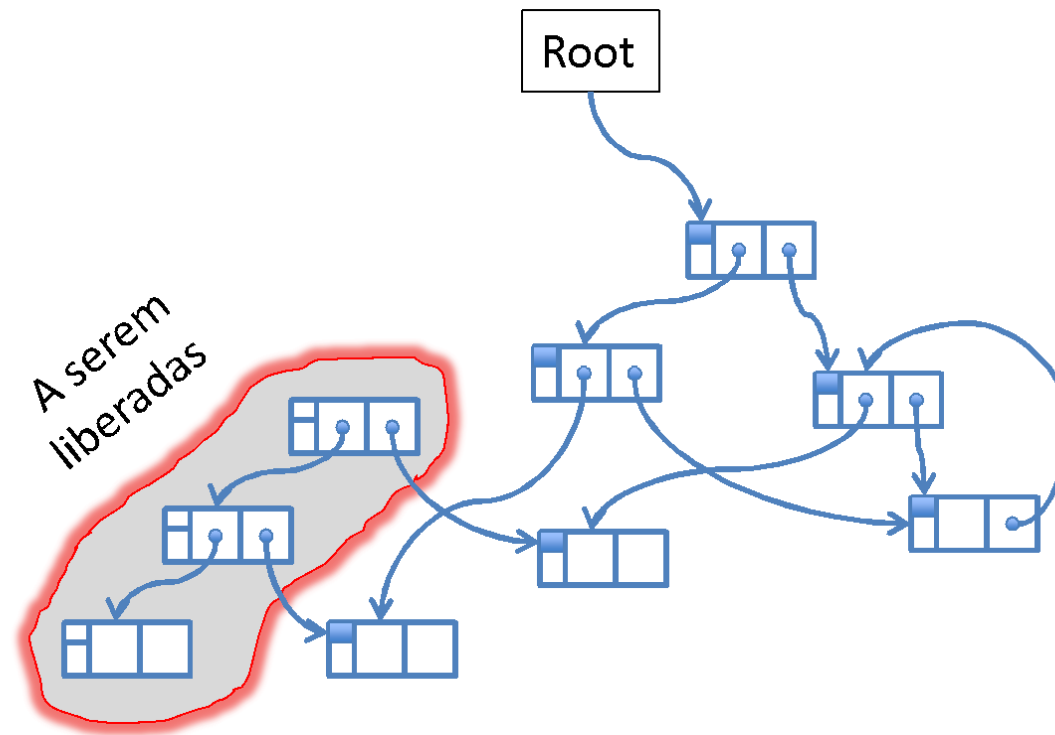
# Mark-Sweep

- Exemplo



# Mark-Sweep

- Exemplo



# Mark-Sweep

- Algoritmos:

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

- marksweep() =>  for R in Roots mark(R)
                  sweep()
                  if empty(freepool) then abort "memory full"

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

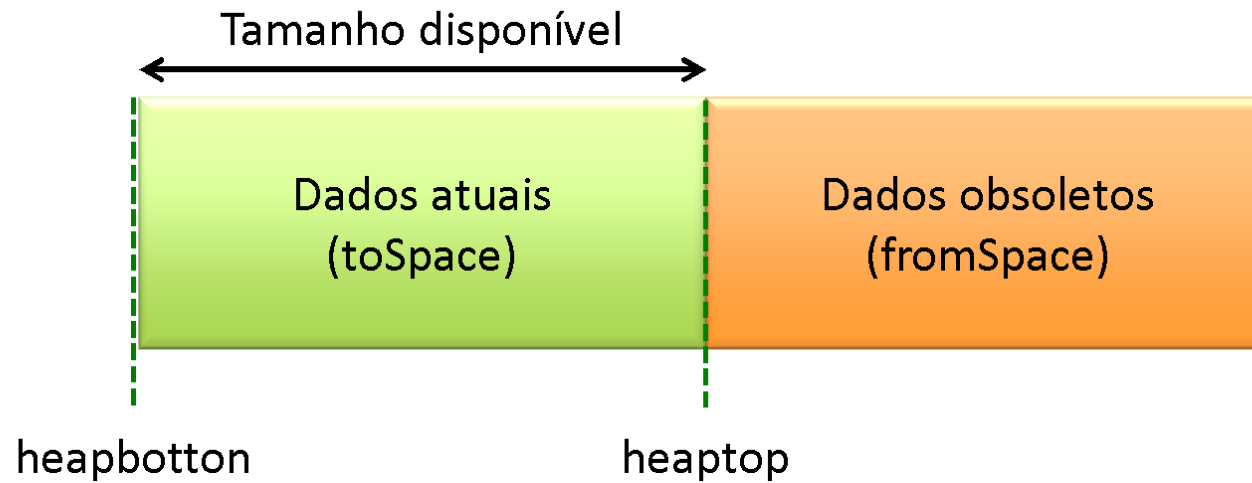
- sweep()    =>  N = heapbotton
                  while N < heaptop
                    if markbit(N) == unmarked then free(N)
                    else markbit(N) = unmarked
                    N = N + size(N)
```



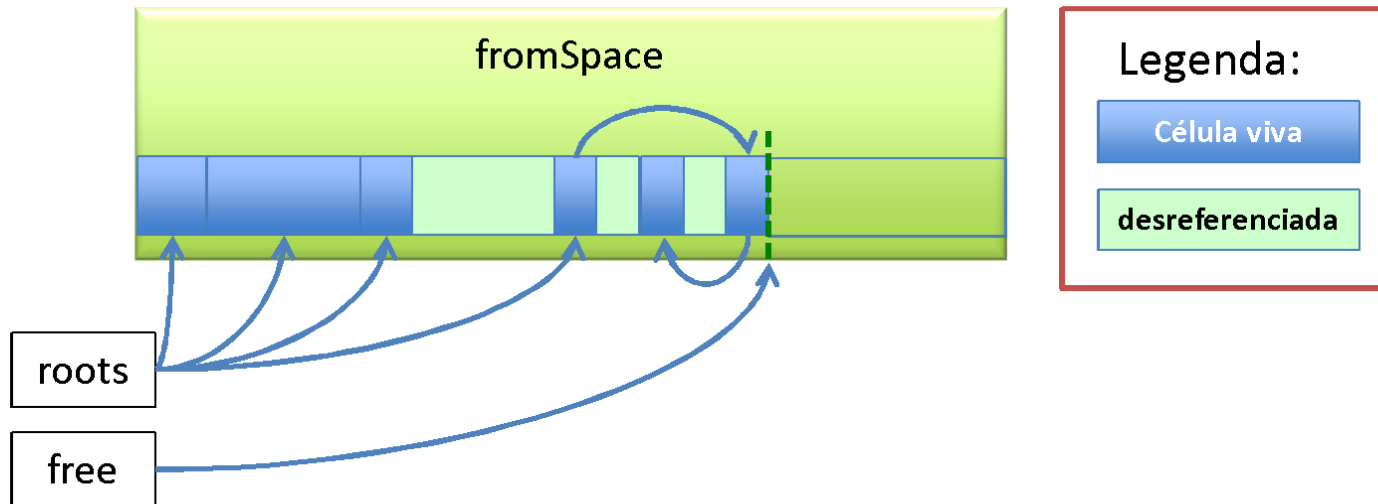
# Mark-Sweep

- Análise:
  - Não há overhead nas operações de manipulação de memória
  - Lida bem com ciclos
  - Gera fragmentação de memória
  - Suspende a execução do programa enquanto a memória é limpa
  - Não é indicado para processamento em tempo real nem sistemas distribuídos

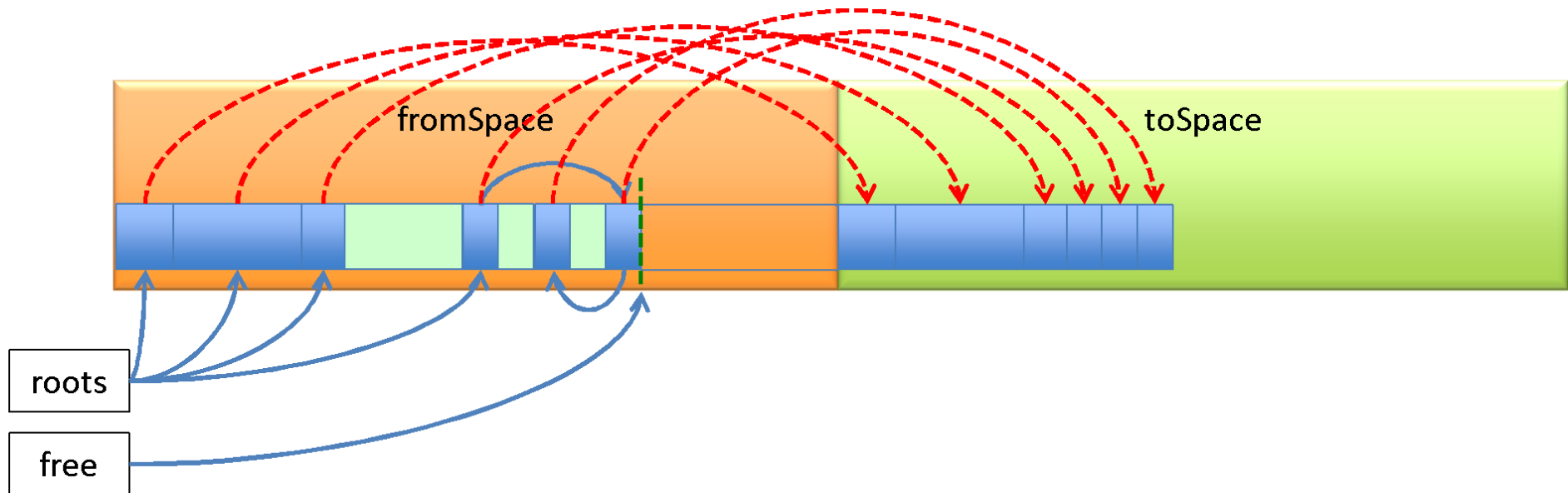
# Copying Collector



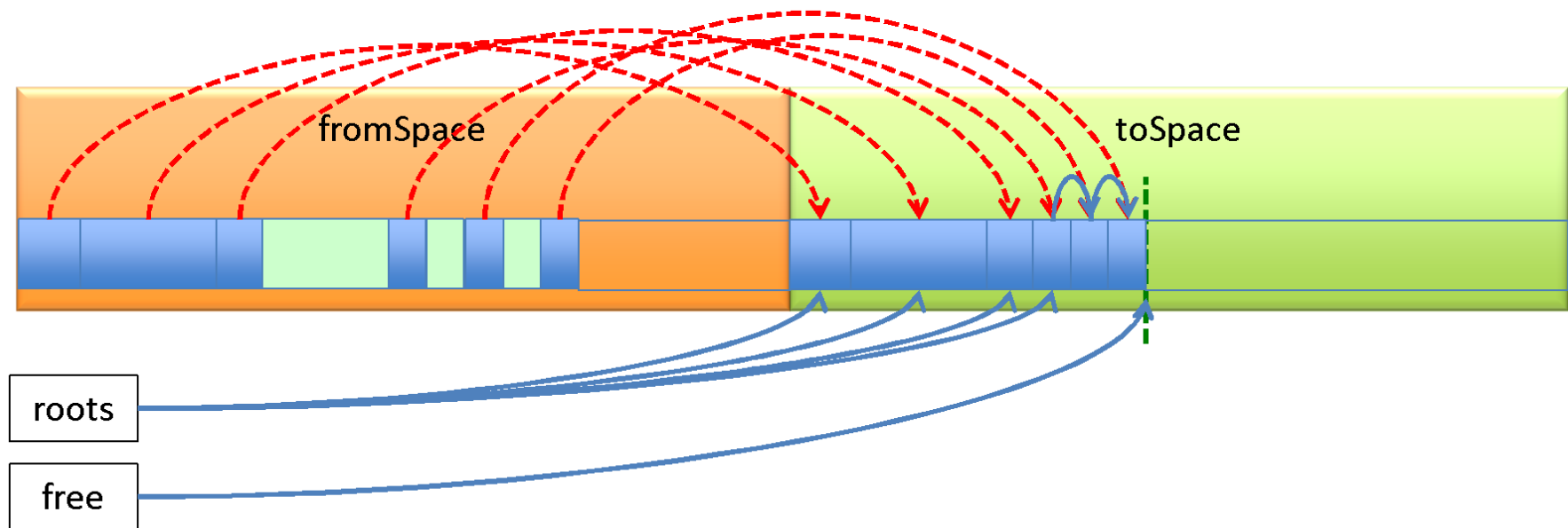
# Copying Collector



# Copying Collector



# Copying Collector



# Copying Collector

- Algoritmos:

- `init()`     $\Rightarrow$  `toSpace = heapbottom`  
                  `spacesize = heapsize / 2`  
                  `topofspace = toSpace + spacesize`  
                  `fromSpace = topofspace + 1`  
                  `free = toSpace + 1`
- `new(n)`     $\Rightarrow$  `if (free+n)>topofspace then flip()`  
                  `if (free+n)>topofspace then abort "memory full"`  
                  `newcell = free`  
                  `free = free + n`  
                  `return newcell`
- `flip()`     $\Rightarrow$  `fromSpace = toSpace`  
                  `toSpace = fromSpace`  
                  `topofspace = toSpace + spaceSize`  
                  `free = toSpace`  
                  `for R in Roots`  
                  `R = copy(R)`

# Copying Collector

- **Algoritmos:**

- `copy(P)`     $\Rightarrow$  if `atomic(P)` or `P == nil` then  
                  return `P`  
                  if not `forwarded(P)` then  
                  `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)`

# Copying Collector

- Análise:
  - Não há overhead nas operações de manipulação de memória
  - Operação de alocação simplificada
  - Lida bem com ciclos
  - Trata a fragmentação de memória
  - Suspende a execução do programa enquanto a memória é limpa
  - Não é indicado para processamento em tempo real nem sistemas distribuídos
  - Uso de dois espaços de memória

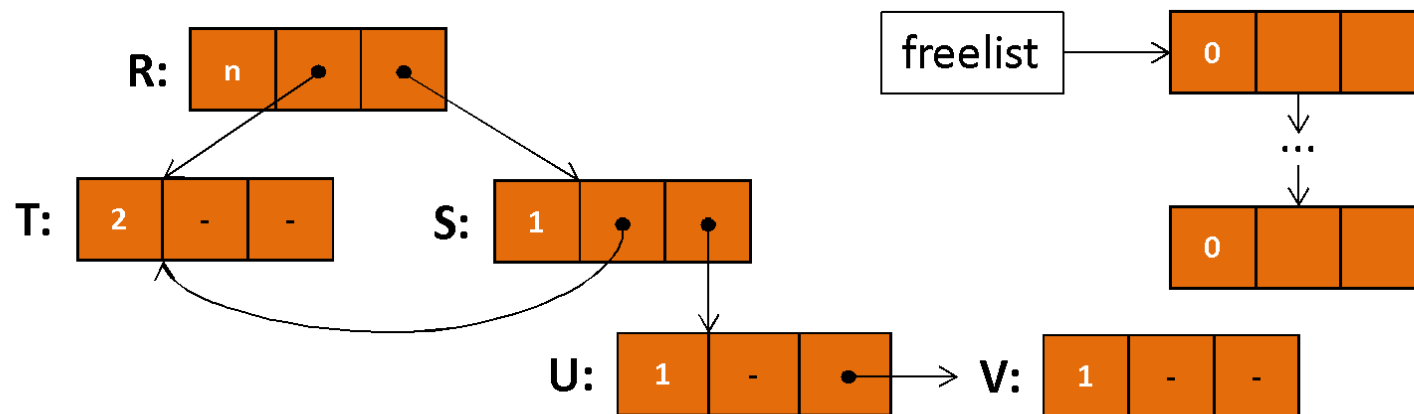


# Exercício

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



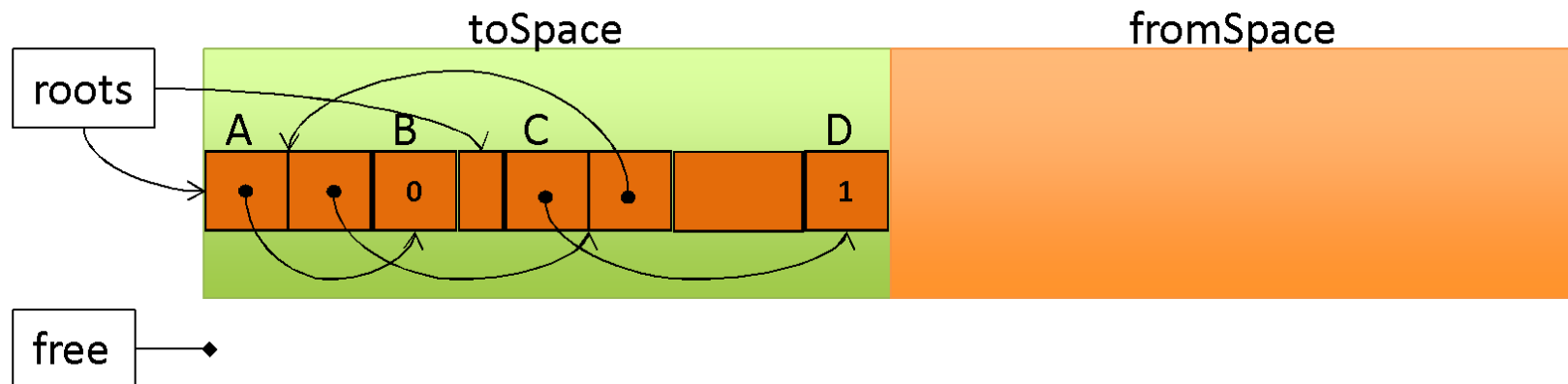
- Considerando ainda que o seguinte *snapshot* corresponda ao conteúdo atual do Heap:



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

# 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*.