



COLETA E COMPACTAÇÃO DE DADOS

Simone Prado

GERENCIAMENTO DE MEMÓRIA

- O gerenciamento de memória, em tempo de execução, é importante para a eficiência dos programas.
- Algumas linguagens permitem que o gerenciamento seja feito pelo programador, outras deixam isso somente para o sistema, por considerar arriscado e propenso a erros.




GERENCIAMENTO DE MEMÓRIA

- A linguagens C ou C++ permitem o gerenciamento da memória do sistema explicitamente, pelo programador.
- Em Java não se tem a responsabilidade de gerenciar a memória. O gerenciamento fica para o sistema resolver. A JVM esconde do programador o gerenciamento de memória.
- O mesmo acontece para C#, Smalltalk, Pearl, dentre outras.



GERENCIAMENTO DE MEMÓRIA

- O gerenciamento enfrenta um problema: lixo gerado pelos programas.
 - Lixo: qualquer bloco da memória que não pode ser acessado pelo programa, ou seja, não há um ponteiro acessível para ele.
 - **Coleta de Lixo** é uma forma automática de realizar gerenciamento de memória com a intenção de liberar espaço de objetos inacessíveis, ou seja, reciclar blocos de memória que não vão mais ser usados, promovendo maior capacidade de alocação e utilização da memória.
- 

GERENCIAMENTO DE MEMÓRIA

- A liberação do espaço (lixo) pode ser:
 - Manual, em C ou C++ (ex: `free(p)`)
 - Automática, como em Java (Garbage Collector)
- Para realizar a coleta de lixo são usados algoritmos que fazem a detecção e desalocação dos objetos inacessíveis na memória, mas a solução pode ser aproximada devido ao custo de processamento.



ALOCAÇÃO DE MEMÓRIA (0-N)

0 **Área Estática:** contem valores cujos requisitos de armazenamento são conhecidos antes do tempo de execução e permanecem constantes por toda a vida do programa em execução

PILHA de tempo de execução: é o centro de controle para despachar funções ativas, variáveis declaradas localmente e a ligação parâmetro-argumento.

a-1

a
h-1



h

HEAP: contém valores que são alocados e estruturados dinamicamente enquanto o programa está sendo executado, como strings, matrizes dinâmicas, objetos e diversas estruturas dinâmicas de dados como listas encadeadas

n

Aqui aparece o problema de alocação e desalocação de memória. Aqui é feita a coleta de lixo.

ALOCAÇÃO DE MEMÓRIA

h	15	Não usado	555	0
	25	Não usado	Não usado	Não usado
	<u>Não definido</u>	77	Não usado	Não usado
	Não usado	Não usado	Não usado	Não usado

n

Suponha que cada palavra possa ter um dentre três estados:

- com um valor;
- Não usada: espaço não alocado;
- Não definida: está alocada para o programa, mas ainda não recebeu um valor;



ALOCAÇÃO DE MEMÓRIA

h	15	Não usado	555	0
	25	Não usado	67	Não usado
	<u>Não definido</u>	77	<u>Não definido</u>	<u>Não definido</u>
	<u>Não definido</u>	<u>Não definido</u>	Não usado	Não usado

```
typedef struct no{
```

```
    int info;
```

```
    struct no* primeiro;
```

```
    struct no* segundo;
```

```
    struct no* terceiro;
```

```
} *def_arvore;
```

n

```
P1 =(def_arvore)malloc(sizeof(struct no));
```



ALOCAÇÃO DE MEMÓRIA

h	15	Não usado	555	0
	25	Não usado	67	Não usado
	<u>Não definido</u>	77	<u>Não definido</u>	<u>Não definido</u>
	<u>Não definido</u>	<u>Não definido</u>	Não usado	Não usado

```
typedef struct no{
    int info;
    struct no* primeiro;
    struct no* segundo;
    struct no* terceiro;
} *def_arvore;
```

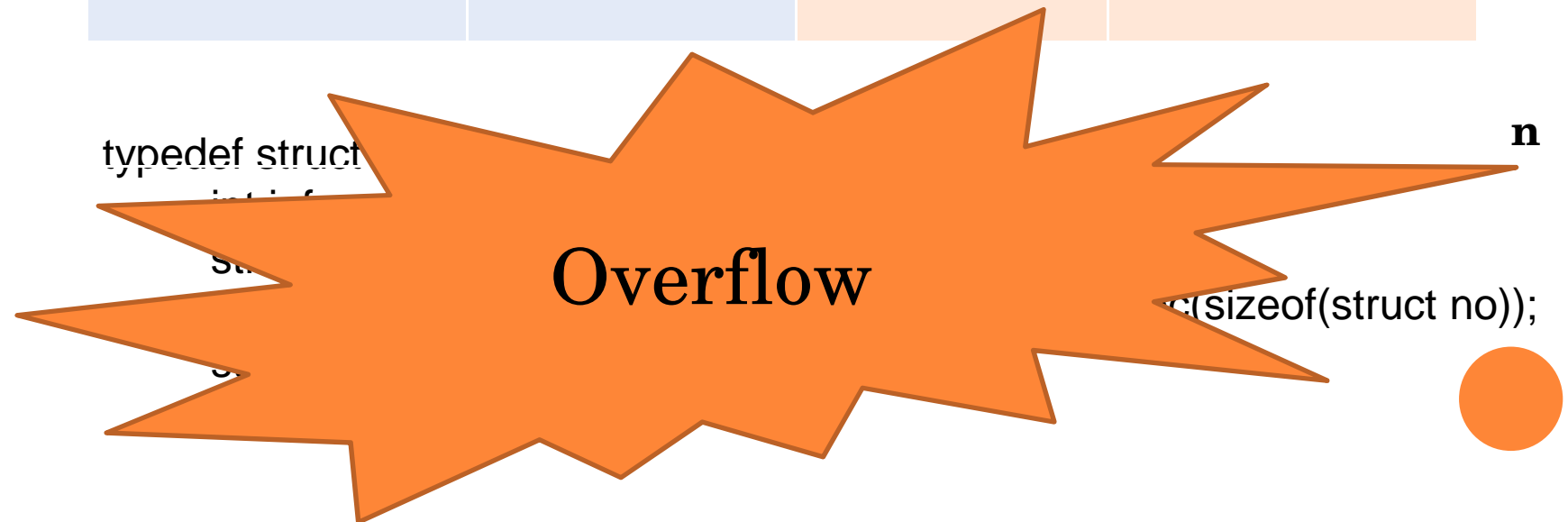
P2=(def_arvore)malloc(sizeof(struct no));

n



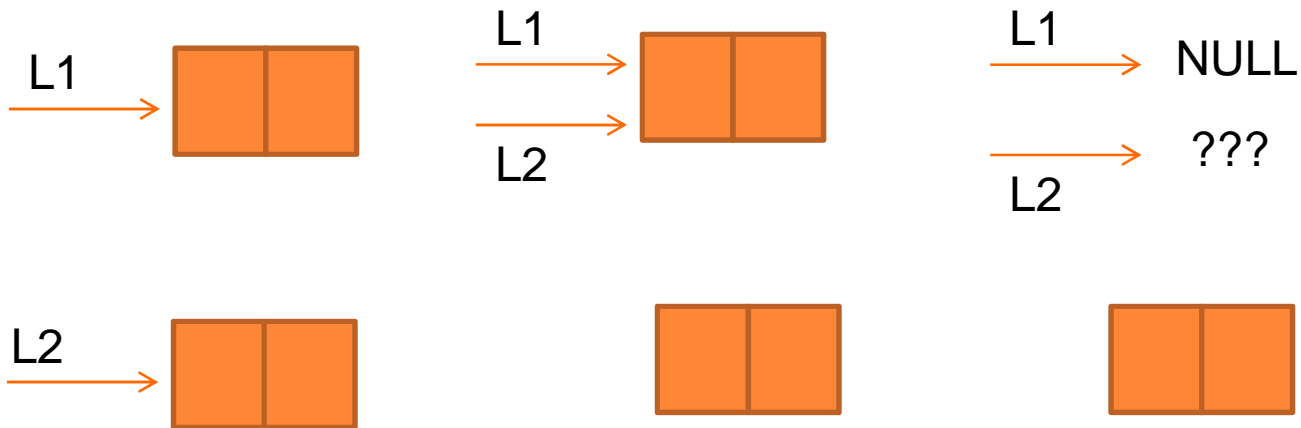
ALOCAÇÃO DE MEMÓRIA

h	15	Não usado	555	0
	25	Não usado	67	Não usado
	Não definido	77	Não definido	Não definido
	Não definido	Não definido	Não usado	Não usado



PROBLEMAS COMUNS COM PONTEIROS

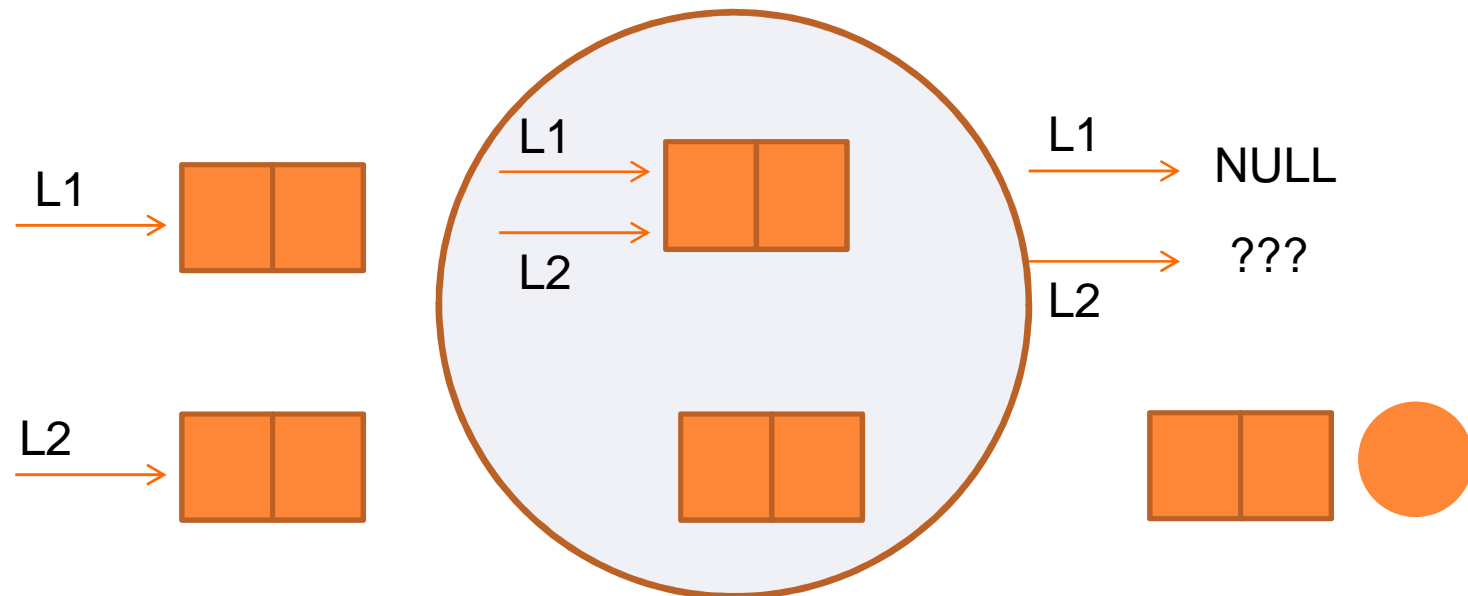
```
def_lista L1, L2;  
L1 = malloc(sizeof(struct no));  
L2 = malloc(sizeof(struct no));  
L2 = L1;  
free(L1);
```



PROBLEMAS COMUNS COM PONTEIROS

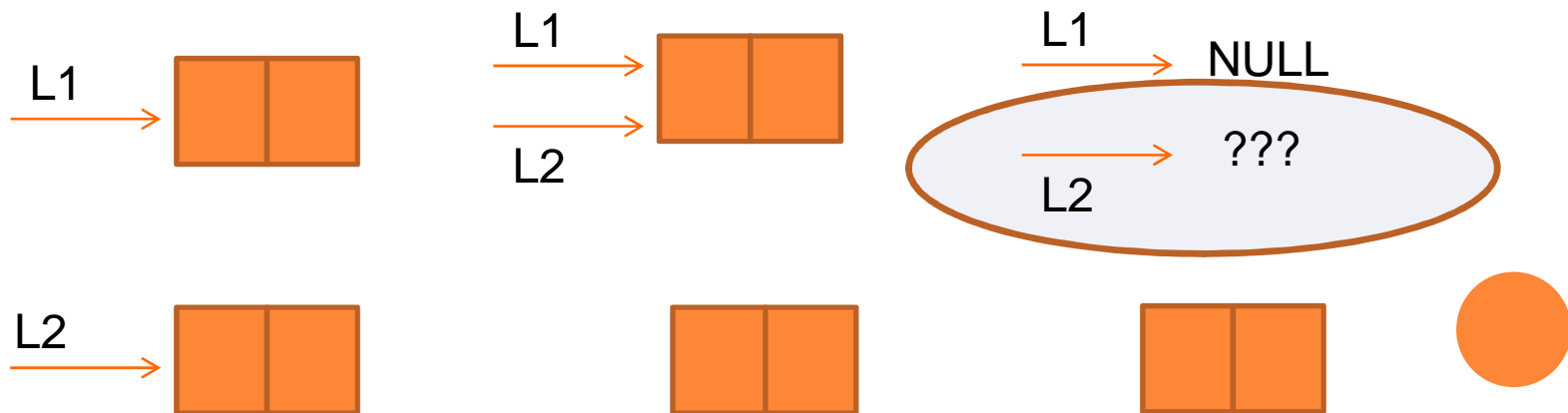
○ Vazamento de memória (*memory leak*)

- Uma área de memória alocada para o programa é “esquecida”
- Em alguns casos, se o procedimento é repetido muitas vezes, o programa acaba falhando por falta de memória



PROBLEMAS COMUNS COM PONTEIROS

- Ponteiro no vazio (*dangling pointer*)
 - Um ponteiro acaba apontando para uma área de memória não mais sob controle do programa, ou seja, um ponteiro acaba apontando para uma área qualquer de memória do programa sem que o programador se dê conta.



COLETA DE LIXO

- Alguns algoritmos:
 - Contagem de referências (técnica mais simples)
 - Marcar e Varrer
 - Coleta de cópias

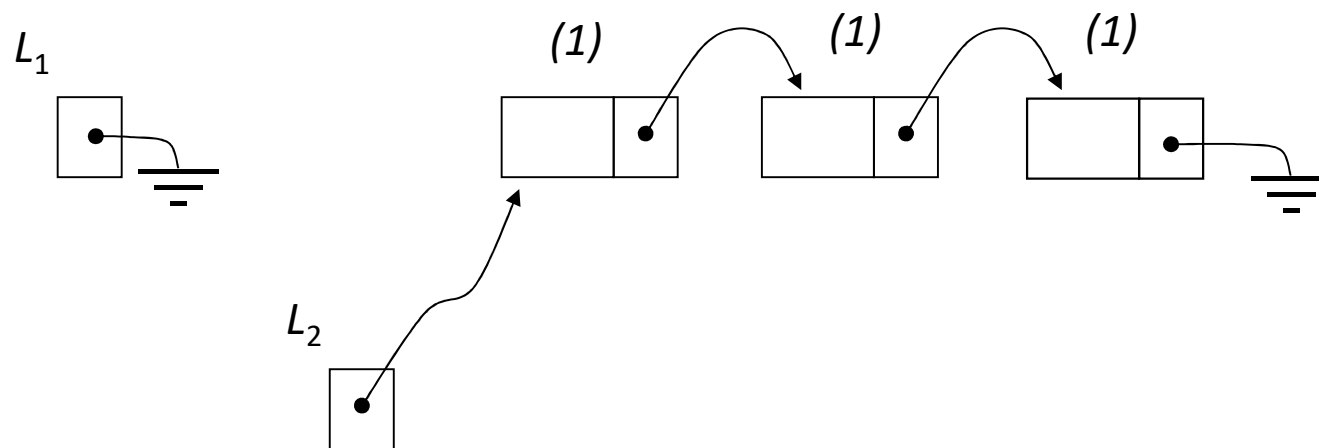


CONTADOR DE REFERÊNCIAS

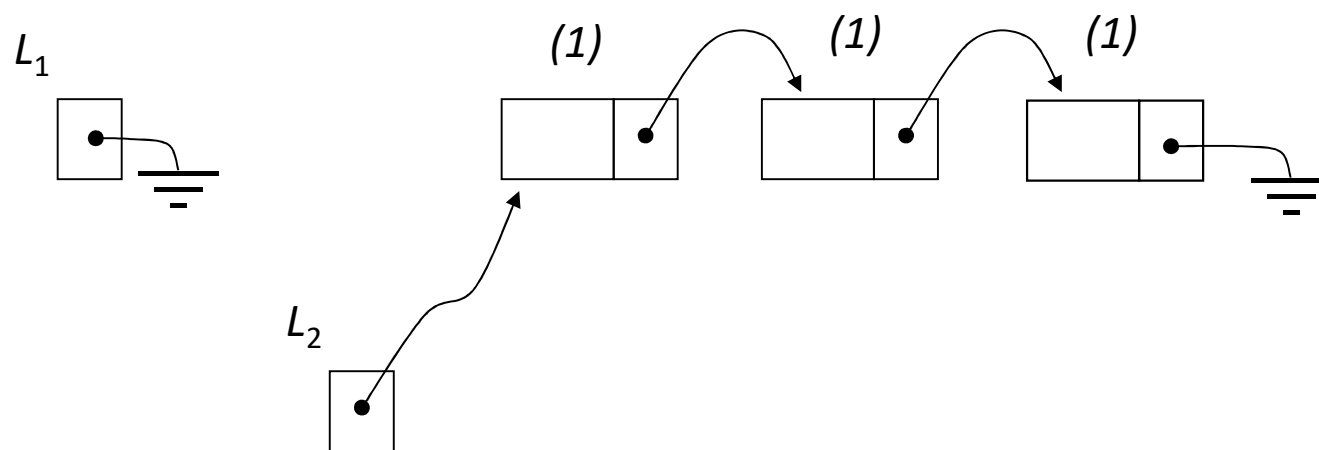
- A cada variável alocada dinamicamente é associado um contador de referências, isto é, um inteiro que indica o número de ponteiros que apontam para a variável
- Quando o contador de referências chega a 0, a variável dinâmica é retornada para o banco de memória livre



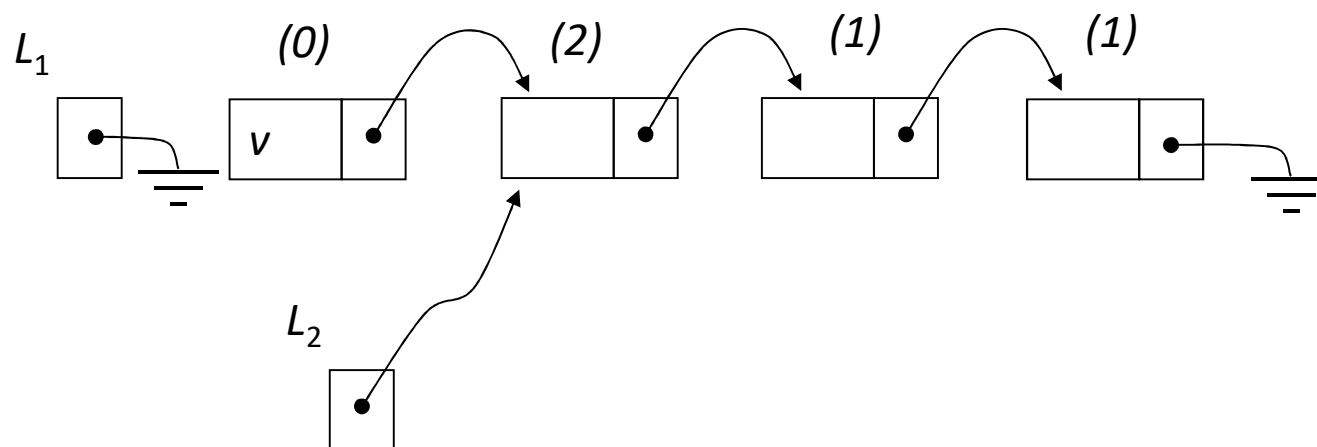
CONTADOR DE REFERÊNCIAS



CONTADOR DE REFERÊNCIAS



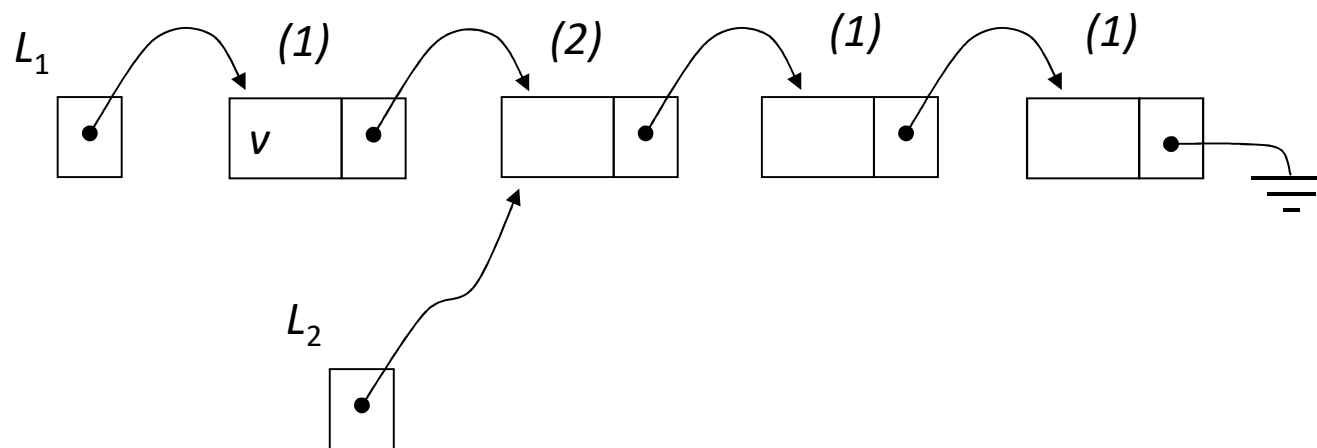
CONTADOR DE REFERÊNCIAS



$L_1 \leftarrow \text{Nulo}$



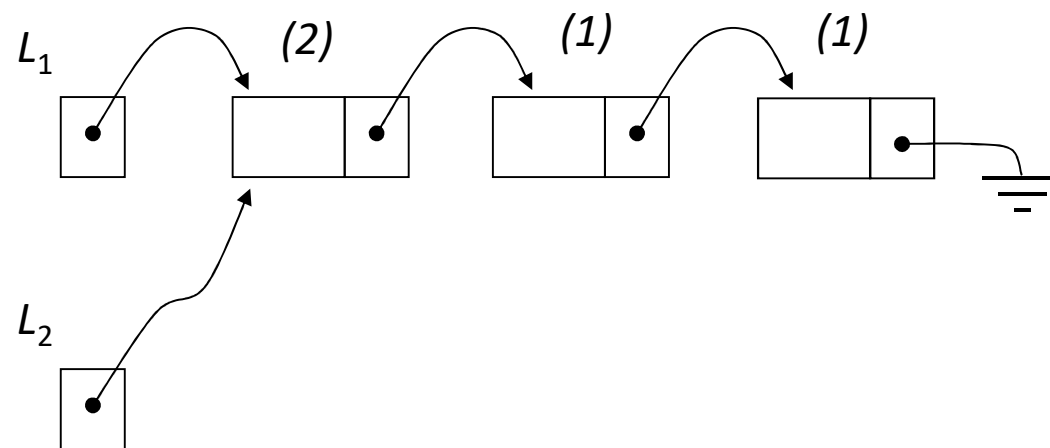
CONTADOR DE REFERÊNCIAS



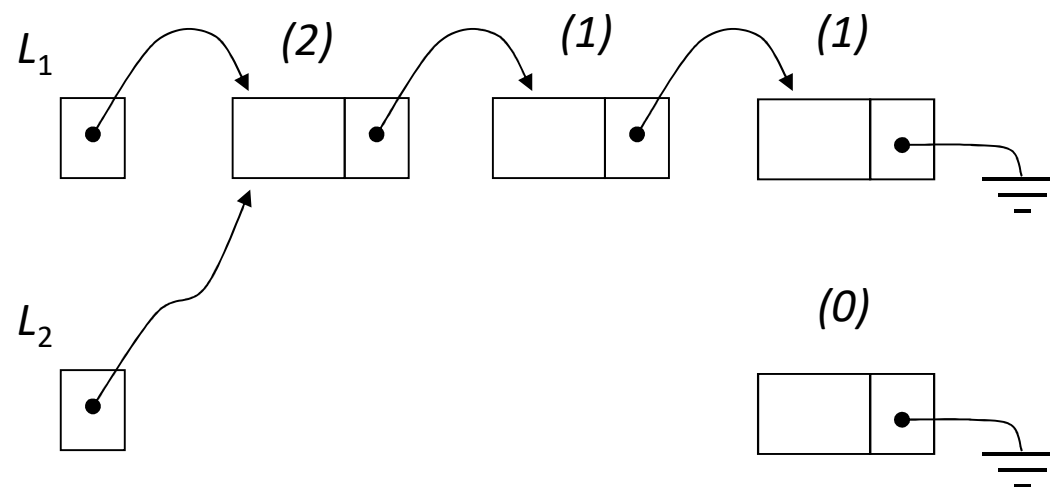
$L_1 \leftarrow \text{CriaNoLista}(v, L_1)$



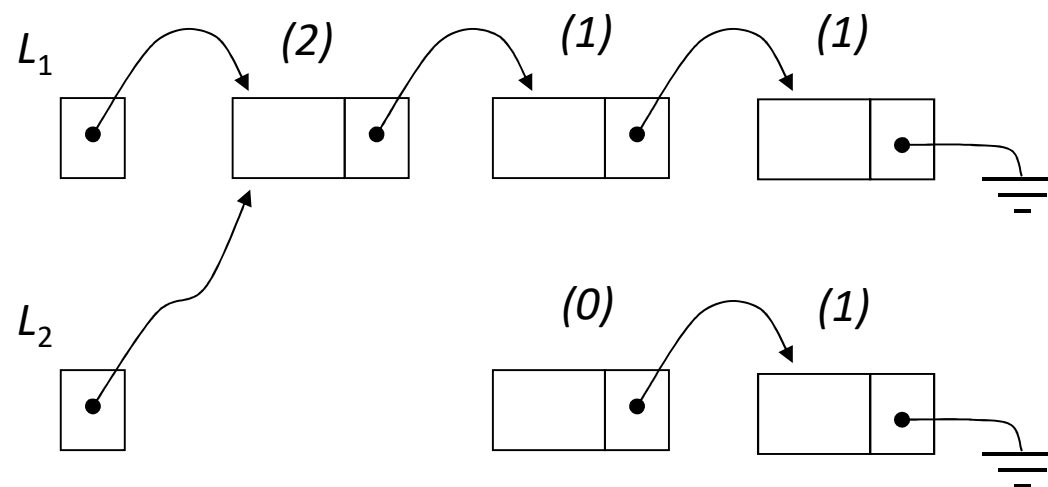
CONTADOR DE REFERÊNCIAS



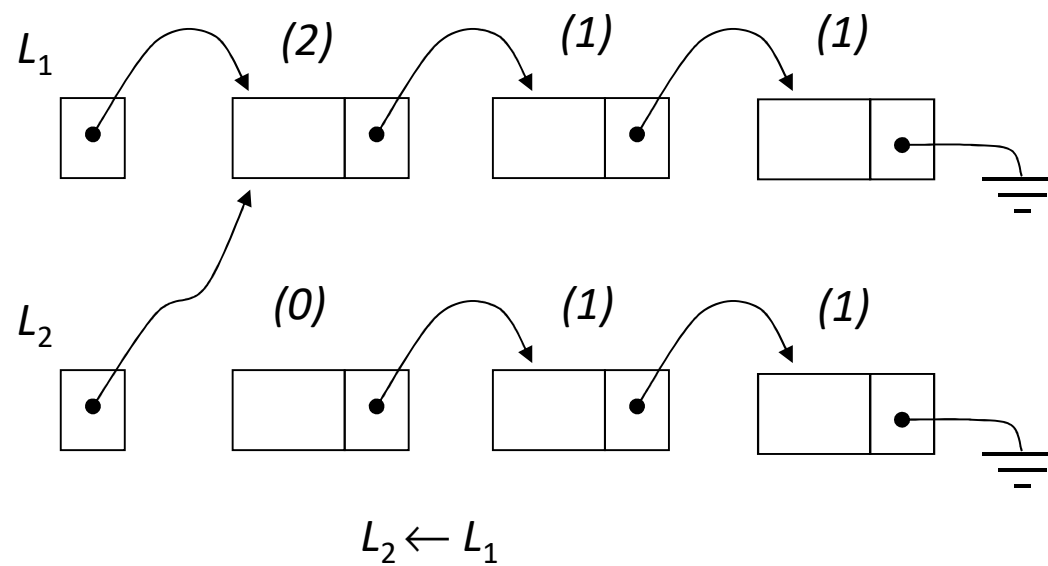
CONTADOR DE REFERÊNCIAS



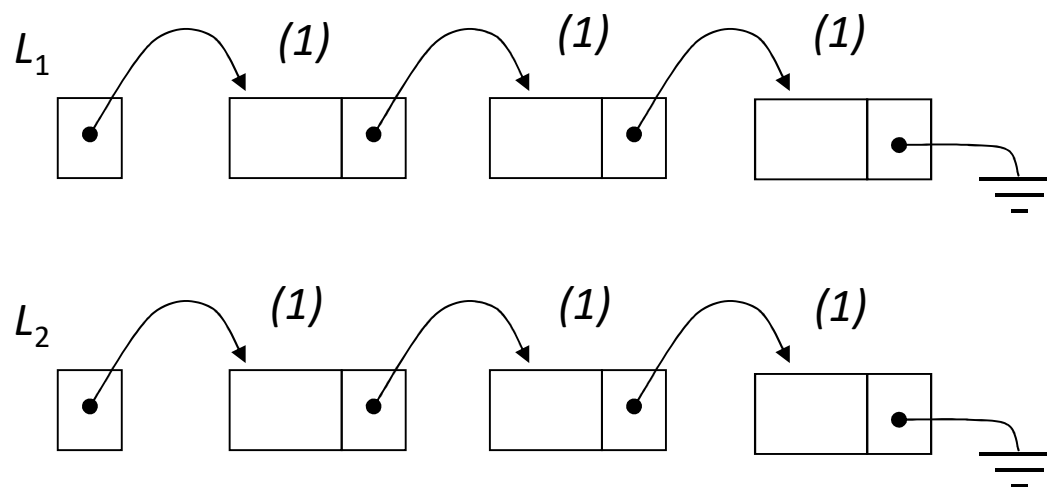
CONTADOR DE REFERÊNCIAS



CONTADOR DE REFERÊNCIAS



CONTADOR DE REFERÊNCIAS



CONTADOR DE REFERÊNCIAS

- Vantagem: ocorre dinamicamente, sempre que uma atribuição de ponteiro ou uma ação na Heap for disparada pelo programa.
- Desvantagens:
 - Falha em detectar cadeias inacessíveis
 - Uso de um contador interno de referências a cada nó da heap, e
 - desempenho

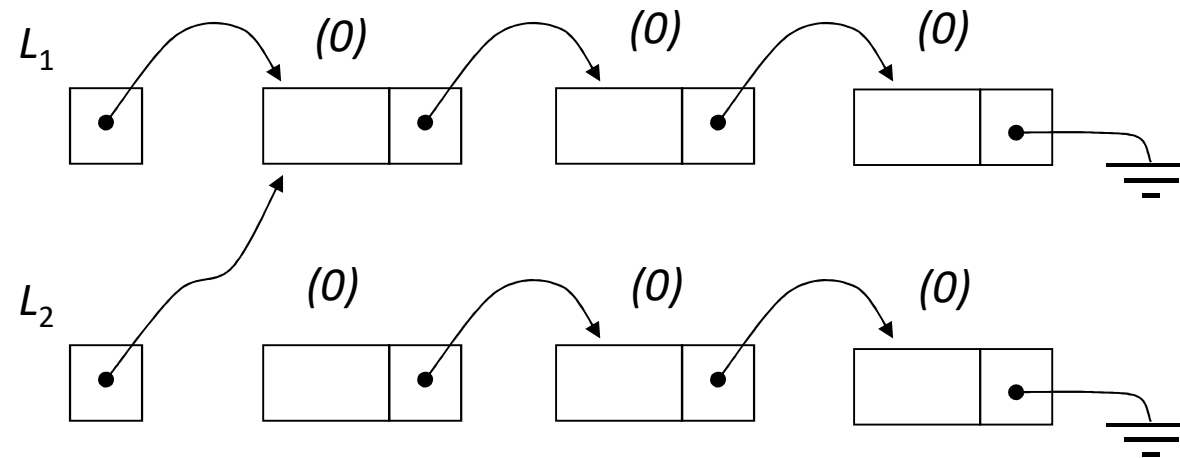


MARCAR E VARRER

- É chamado somente se a heap está cheia, mas quando o é, o processo é mais elaborado e demorado.
- Ele executa duas passagens pela heap:
 1. Marcar cada bloco da heap que possa ser alcançado
 2. Varrer, liberando o espaço e desmarcando os blocos que tenham sido marcados em (1).



MARCAR E VARRER

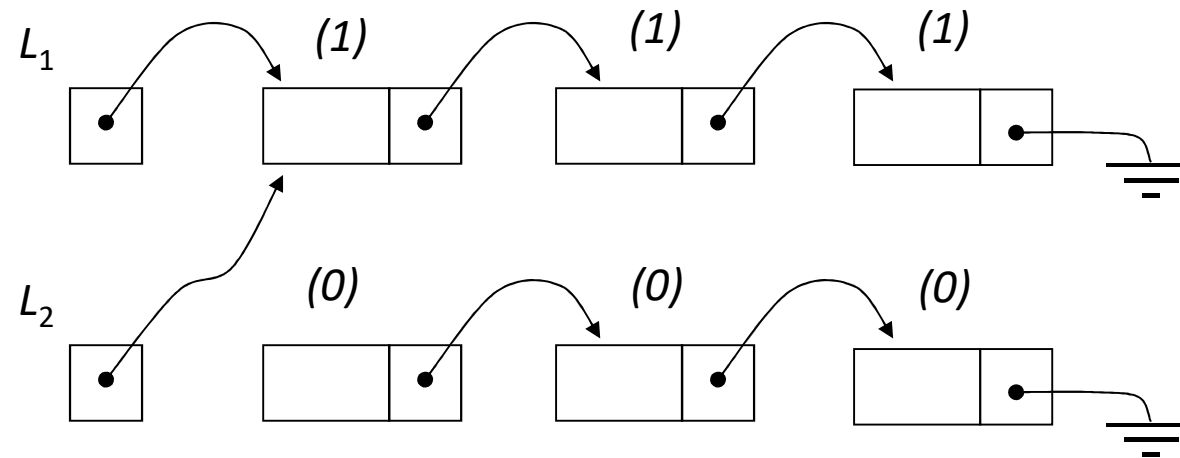


```
if (cheia(heap))  
    marcar_varrer();
```

Marcar cada bloco de memória (bit de marca – MB)

```
marcar(R):  
    if(R.MB == 0) R.MB=1;  
    if(R.next != NULL) marcar(R);
```

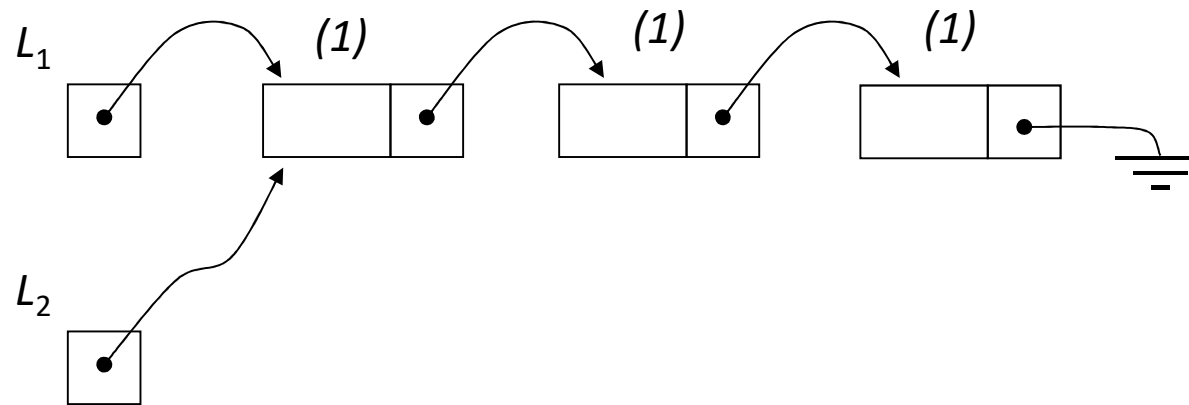
MARCAR E VARRER



```
marcar(R):  
    if(R.MB == 0) R.MB=1;  
    if(R.next != NULL) marcar(R);
```

```
Varrer():  
    i=h;  
    while (i <= n){  
        if (i.MB ==0) free(i);  
        else i.MB = 0;  
        i++;  
    }
```

MARCAR E VARRER

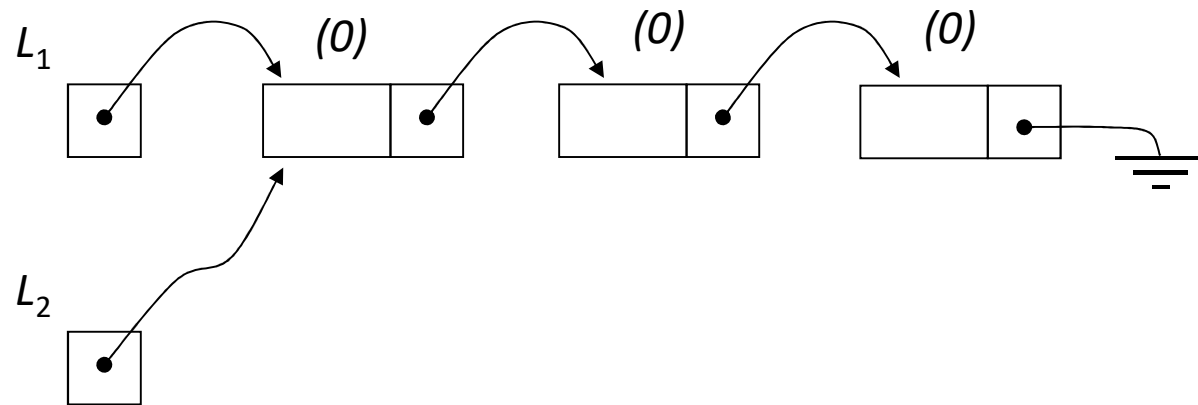


Varrer():

```
i=h;  
while (i <= n){  
    if (i.MB ==0) free(i);  
    else i.MB = 0;  
    i++;  
}
```



MARCAR E VARRER



MARCAR E VARRER

- Problemas:
 - Fragmentação da memória, impossibilitando alocação de objetos grandes mesmo tendo memória livre suficiente.
 - Cada ciclo exige que a toda a memória heap ser percorrida
- Uma solução seria o método Marcar e Compactar.

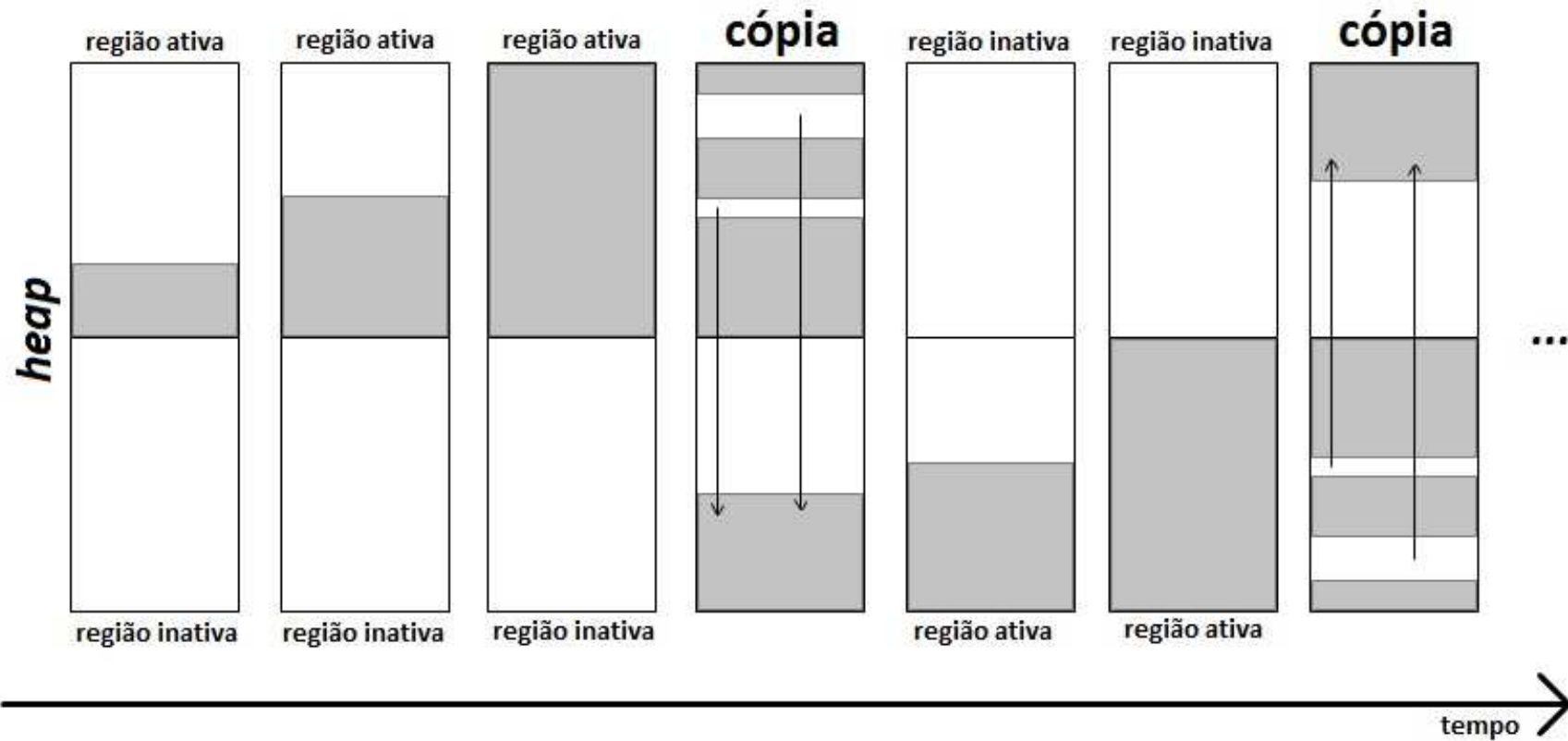


COLETA DE CÓPIAS (OU STOP-COPY)

- Trabalha com duas regiões de memória (heap dividida):
 - Região ativa
 - Região inativa;
- Execução:
 - Busca por objetos não mais alcançáveis;
 - Copia objetos alcançáveis para a nova região de memória;
 - Troca entre as regiões ativa e inativa;



COLETA DE CÓPIAS



REFERÊNCIAS

- www.ime.usp.br/~song/mac5710/slides/04gc.pdf
- www.maxwell.lambda.ele.puc-rio.br/7645/7645_3.PDF
- Tucker, A B & Noonan, R E. Linguagens de Programação: Princípios e paradigmas. 2ª Ed. São Paulo: Mc Graw-Hill, 2008. Capítulo 11.

