CAPÍTULO

9

LISTAS ENCADEADAS

Neste capítulo, introduziremos a estrutura de lista encadeada, descrevendo as operações básicas que essa estrutura suporta, e mostraremos como implementá-la com alocação dinâmica encadeada.

9.1 Fundamentos

Lista encadeada é uma sequência de *nós*, em que cada nó guarda um *item* e um ponteiro para o *próximo* nó da sequência. O endereço do primeiro nó é mantido em um *ponteiro inicial* `I`, a partir do qual todos os nós da sequência podem ser acessados. O último nó da sequência não tem sucessor, isto é, tem um ponteiro com valor `NULL`, definido em `stdio.h`. Um ponteiro inicial com valor `NULL` representa uma *lista vazia*. Por exemplo, a Figura 9.1 apresenta uma lista encadeada composta por três nós, que ocupam posições arbitrárias de memória.

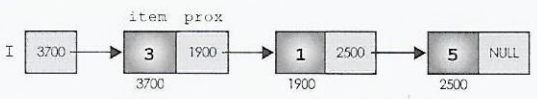


Figura 9.1 | Uma lista encadeada composta por três nós.

Listas encadeadas são úteis em programas que precisam lidar com coleções *dinâmicas*, cujas quantidades de itens podem variar em tempo de execução. Por exemplo, listas encadeadas podem ser usadas para implementar pilhas e filas.

9.2 Operações em listas encadeadas

Para criar uma lista encadeada, é preciso definir a *estrutura* dos nós que serão usados em sua composição, bem como o tipo de *ponteiro* que será usado para apontar seu nó inicial, como indicado na Figura 9.2.

```
#define fmt "%d "                    // formato de exibicao dos itens
typedef int Item;                    // tipo dos itens na lista
typedef struct no {                  // estrutura dos nos da lista
   Item item;
   struct no *prox;
} *Lista;                            // tipo de ponteiro para lista
```

Figura 9.2 | Definições para criação de lista encadeada.

O tipo `Item`, definido como `int`, indica que os itens na lista serão números inteiros. Assim, por exemplo, para criar uma lista de caracteres, basta definir o tipo `Item` como `char`. A constante `fmt` indica o formato de exibição de um item da lista e, portanto, deve ser compatível com o tipo `Item`.

O tipo `Lista`, definido como ponteiro para `struct no`, é usado para declarar um ponteiro para lista encadeada (isto é, um ponteiro que aponta o primeiro nó de uma lista encadeada). Assim, por exemplo, se `I` é um ponteiro para uma lista encadeada, então `I->item` é o item guardado no primeiro nó da lista e `I->prox` é um ponteiro para o segundo nó da lista (isto é, para o *resto* da lista).

9.2.1 Criação de lista

Para facilitar a criação de uma lista encadeada, vamos usar a função definida na Figura 9.3, que cria um nó contendo um item e um ponteiro para seu sucessor.

```
Lista no(Item x, Lista p) {
   Lista n = malloc(sizeof(struct no));
   n->item = x;
   n->prox = p;
   return n;
}
```

Figura 9.3 | Função para criação de um nó de lista encadeada.

Essa função executa os seguintes passos:

- Chama a função `malloc()` para alocar a área de memória em que o nó será criado, cujo tamanho em *bytes* é `sizeof(struct no)`. O endereço da área alocada, devolvido por `malloc()`, é atribuído ao ponteiro `n`.
- Atribui os valores recebidos como parâmetros aos campos do nó.
- Devolve como resposta o endereço da área em que o nó foi criado.

Com essa função, a lista da Figura 9.1 pode ser criada do seguinte modo:

```
Lista I = no(3,no(1,no(5,NULL)));
```

A execução dessa instrução ocorre da seguinte forma:

- Primeiro, a chamada `no(5,NULL)` cria um nó, preenchido com `5` e `NULL`, e devolve seu endereço como resposta. Então, supondo que esse nó tenha sido criado no endereço `2500`, como mostra a Figura 9.4a, a composição `no(3,no(1,no(5,NULL)))` é reduzida a `no(3,no(1,2500))`.
- Depois, a chamada `no(1,2500)` cria outro nó, preenchido com `1` e `2500`, e devolve seu endereço como resposta. Supondo que esse segundo nó tenha sido criado no endereço `1900`, como mostra a Figura 9.4b, a composição `no(3,no(1,2500))` é reduzida a `no(3,1900)`. Além disso, como `2500` é o endereço do primeiro nó criado, o segundo fica apontando o primeiro.
- Finalmente, a chamada `no(3,1900)` cria um terceiro nó, preenchido com `3` e `1900` (logo, esse nó fica apontando o segundo), e devolve seu endereço, que é armazenado no ponteiro inicial `I`. Então, supondo que o terceiro nó tenha sido criado no endereço `3700`, como mostra a Figura 9.4c, `I` ficará apontando para o endereço `3700` (ou seja, para o início da lista encadeada).

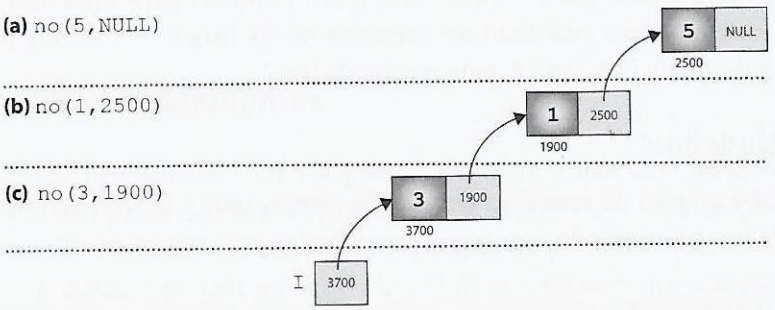


Figura 9.4 | Passos para criação de uma lista encadeada.

9.2.2 Exibição de lista

Para exibir uma lista encadeada, usaremos a função definida na Figura 9.5.

```
void exibe(Lista L) {
   while( L != NULL ) {
      printf(fmt,L->item);
      L = L->prox;
   }
}
```

Figura 9.5 | Função para exibição de lista encadeada.

Por exemplo, quando a chamada `exibe(I)` é feita, o valor de `I` é atribuído a `L`, que fica apontando o primeiro nó da lista (Figura 9.6a). Como `L` é diferente de `NULL`, a repetição `while` é iniciada. Na primeira iteração, `L` aponta um nó contendo `3` e `1900`. Então, a execução de `printf(fmt,L->item)` causa a exibição do item `3` no vídeo e a execução de `L = L->prox` faz com que o ponteiro `L` receba o valor `1900`, passando a apontar o segundo nó da lista (Figura 9.6b). Após a primeira iteração, `L` continua diferente de `NULL` e, portanto, a segunda iteração é iniciada. Nessa iteração, `L` aponta um nó contendo `1` e `2500`. Então, o item `1` é exibido no vídeo e `L` fica apontando o terceiro nó da lista (Figura 9.6c). Finalmente, na terceira iteração, `L` aponta um nó contendo `5` e `NULL`. Então, o item `5` é exibido no vídeo e o ponteiro `L` torna-se `NULL` (Figura 9.6d), fazendo com que a repetição `while` termine. No final da repetição, o ponteiro `L` sempre terá valor `NULL`; porém, o endereço do primeiro nó da lista ainda estará armazenado no ponteiro `I` (ou seja, os itens da lista encadeada continuarão acessíveis).

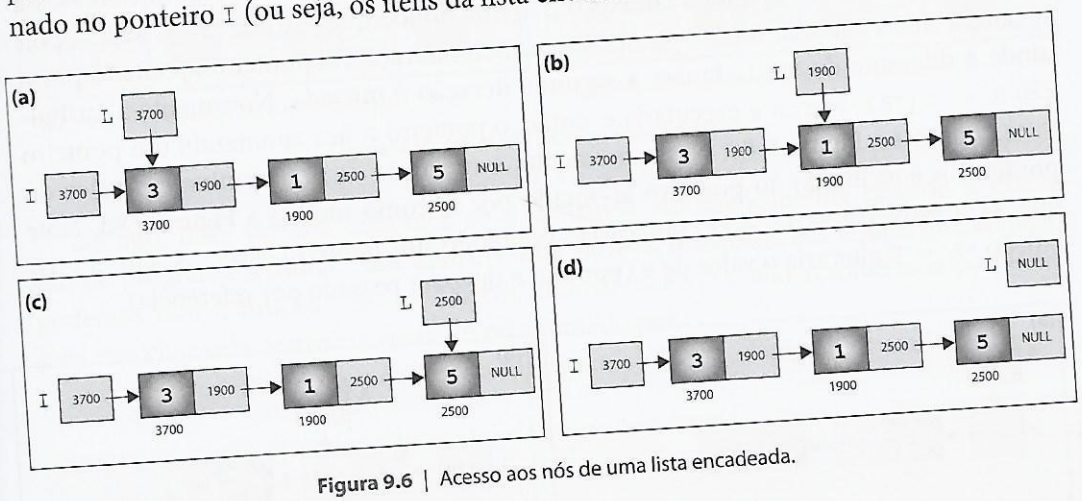


Figura 9.6 | Acesso aos nós de uma lista encadeada.

9.2.3 Anexação de listas

Anexação (ou *concatenação*) é uma operação que anexa uma lista ao final de outra. Por exemplo, suponha que o ponteiro `H` aponta a lista `[4,2]` e que o ponteiro `I` aponta a lista `[3,1,5]`, como na Figura 9.8a. Então, após a anexação dessas listas, `H` apontará a lista `[4,2,3,1,5]`; porém, `I` continuará apontando a lista `[3,1,5]`, que se tornará um *sufixo* da lista apontada por `H`, como na Figura 9.8d.

A função para anexação de listas encadeadas é definida na Figura 9.7. Para anexar duas listas `H` e `I`, basta chamar `anexa(&H,I)`. Note que essa função recebe como parâmetros o *endereço* do ponteiro `H` e o *valor* do ponteiro `I`. Como `H` é do tipo `Lista` (que já é um ponteiro), o primeiro parâmetro da função deve ser do tipo `Lista *` (isto é, um *ponteiro de ponteiro*). Nesse caso, dizemos que `H` é passado por *referência* e que `I` é passado por *valor*. Apenas ponteiros passados por referência podem ter seus valores alterados por uma função.

```
void anexa(Lista *A, Lista B) {
   if( B == NULL ) return;
   while( *A != NULL ) A = &(*A)->prox;
   *A = B;
}
```

Figura 9.7 | Função para anexação de listas encadeadas.

Por exemplo, quando a chamada `anexa(&H,I)` é feita, o endereço de `H` é atribuído ao ponteiro *indireto* `A` e o valor de `I` é atribuído ao ponteiro *direto* `B`, como na Figura 9.8a. Então, como `B` não é `NULL`, a execução continua na repetição `while` (se `B` fosse `NULL`, não haveria o que anexar e a execução poderia terminar). O valor do ponteiro apontado por `A` (denotado por `*A`) é testado e, como ele é diferente de `NULL`, o endereço do campo `prox` do nó apontado *indiretamente* por `A` (denotado por `&(*A)->prox`) é atribuído ao próprio ponteiro `A`. Como resultado, temos a situação na Figura 9.8b. Nessa situação, o ponteiro `A` aponta o endereço `3704` (supondo que `sizeof(int)` seja `4`, pois o campo `item` aguarda um `int`). Após a primeira iteração, o ponteiro apontado por `A` ainda é diferente de `NULL`. Então, a segunda iteração é iniciada. Novamente a atribuição `A = &(*A)->prox` é executada e, então, o ponteiro `A` fica apontando um ponteiro `NULL`, como na Figura 9.8c. Nesse momento, a repetição `while` termina e o valor do ponteiro `B` é atribuído ao ponteiro apontado por `A`, como mostra a Figura 9.8d. Note que, se `H` estivesse vazia, `A` já começaria apontando um ponteiro `NULL`. Assim, a atribuição `*A = B` alteraria o valor de `H` (por isso `H` deve ser passado por referência).

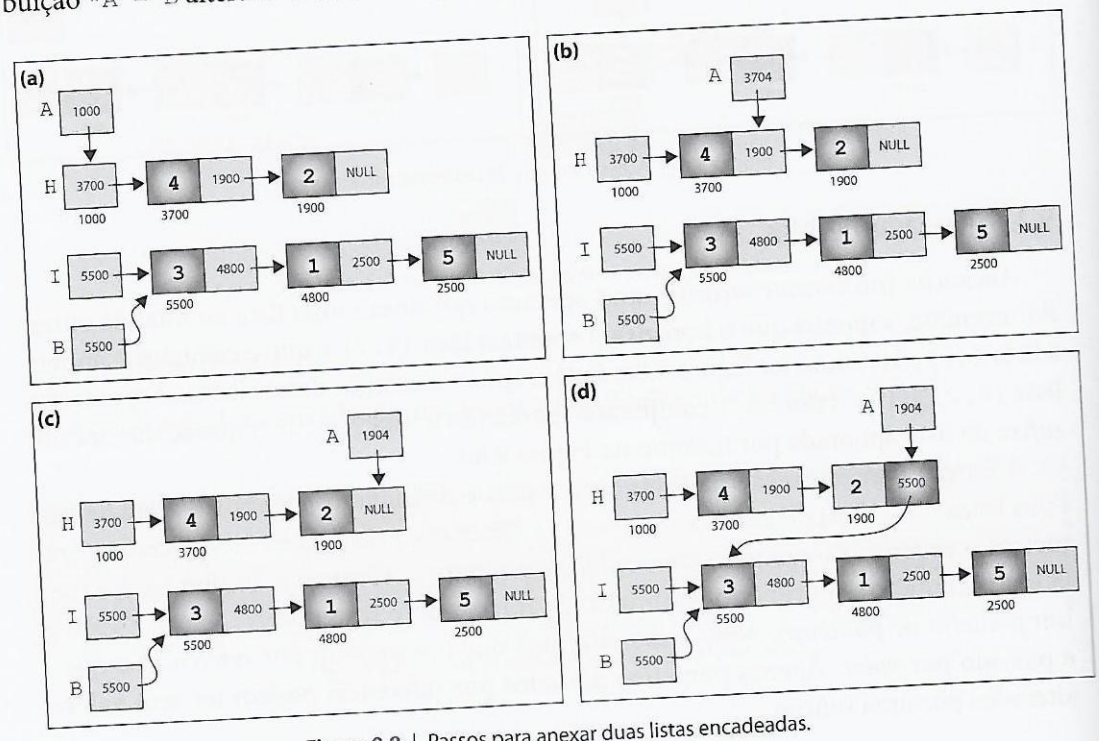


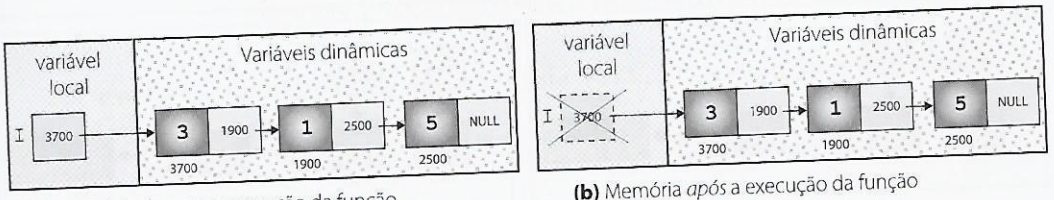
Figura 9.8 | Passos para anexar duas listas encadeadas.

9.2.4 Destruição de lista

Considere a função na Figura 9.9, que cria uma lista encadeada apontada por `I`. Sendo `I` uma variável local, quando a execução da função termina, ela é automaticamente destruída, mas os nós da lista apontada por ela não (Figura 9.10).

```
void f(void) {
   Lista I = no(3,no(1,no(5,NULL)));
   exibe(I);
}
```

Figura 9.9 | Lista encadeada apontada por uma variável local.



(a) Memória *durante* a execução da função **(b)** Memória *após* a execução da função

Figura 9.10 | Variável local e variável dinâmica.

De fato, uma variável dinâmica só é destruída automaticamente quando a execução do programa termina. Para destruir uma variável dinâmica, em tempo de execução, podemos usar a função `free()`. Porém, chamar `free(I)` não resolveria o problema, pois essa chamada destruiria apenas o nó apontado por `I`.

Para destruir uma lista, usaremos a função `destroi()`, dada na Figura 9.11.

```
void destroi(Lista *L) {
   while( *L != NULL ) {
      Lista n = *L;
      *L = n->prox;
      free(n);
   }
}
```

Figura 9.11 | Função para destruição de lista.

Por exemplo, quando a chamada `destroi(&I)` é feita, o endereço do ponteiro *direto* `I` é atribuído ao ponteiro *indireto* `L`. Portanto, o ponteiro `L` fica apontando o ponteiro que aponta o primeiro nó da lista (Figura 9.12a). Então, como o ponteiro apontado por `L` (denotado por `*L`) é diferente de `NULL`, a repetição `while` é iniciada. Na primeira iteração, a instrução `n = *L` atribui ao ponteiro `n` o valor do ponteiro apontado por `L` (que é `3700`). Como resultado, `n` fica apontando o primeiro nó da lista (Figura 9.12b). Em seguida, a instrução `*L = n->prox` copia o valor do campo `prox` do nó apontado por `n` (que é `1900`) para o ponteiro apontado por `L`, que passa a apontar o segundo nó da lista (Figura 9.12c). Finalmente, a instrução `free(n)` destrói o nó apontado por `n` e a lista encadeada passa a ter apenas dois nós (Figura

9.12d). A partir daí, o processo se repete de forma análoga ao que já foi descrito. Quando a repetição `while` termina, todos os nós da lista encadeada foram destruídos e o ponteiro apontado por `L` (isto é, o ponteiro inicial `I`) tem valor `NULL` (isto é, `I` representa uma *lista vazia*).

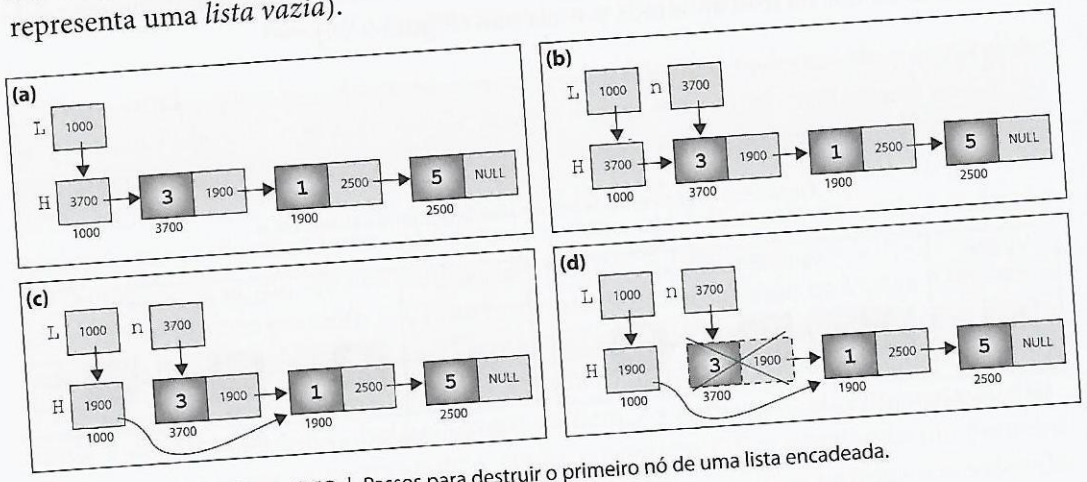


Figura 9.12 | Passos para destruir o primeiro nó de uma lista encadeada.

9.3 Manipulação recursiva de listas encadeadas

Listas são naturalmente definidas de forma recursiva: uma lista `L` é *vazia* (`NULL`) ou é um *item* (`L->item`) seguido de uma *lista* (`L->prox`). Explorando esse fato, podemos criar funções bem simples e concisas para manipular listas encadeadas.

9.3.1 Tamanho de lista

A função que determina o *tamanho* de uma lista `L`, na Figura 9.13, usa a seguinte ideia: (*base*) se `L` está vazia, então seu tamanho é `0`; (*passo*) senão, seu tamanho é `1` a mais que o tamanho de seu *resto* (isto é, a lista `L->prox`).

```
int tam(Lista L) {
   if( L == NULL ) return 0;
   return 1 + tam(L->prox);
}
```

Figura 9.13 | Função para determinar o tamanho de uma lista.

Por exemplo, supondo que `L` aponte a lista `[3,5]`, a execução da chamada `tam(L)` pode ser simulada do seguinte modo:

```
tam([3,5])
= 1 + tam([5])
= 1 + 1 + tam([])
= 1 + 1 + 0
= 2
```

9.3.2 Pertinência em lista

A função que verifica se um item `x` *pertence* a uma lista `L`, na Figura 9.14, usa a seguinte ideia: (*base*) se `L` está vazia, então `x` não pertence a `L`; senão, se `x` é igual ao primeiro item de `L`, então `x` pertence a `L`; (*passo*) senão, `x` pertence a `L` se `x` pertence ao *resto* de `L`.

```
int pert(Item x, Lista L) {
   if( L == NULL ) return 0;
   if( x == L->item ) return 1;
   return pert(x,L->prox);
}
```

Figura 9.14 | Função para verificação de pertinência em lista.

Por exemplo, supondo que `L` aponta a lista `[3,1,5]`, a simulação da execução da chamada `pert(1,L)` é a seguinte:

```
pert(1,[3,1,5])
= pert(1,[1,5])
= 1
```

Analogamente, a simulação para a execução da chamada `pert(7,L)` é:

```
pert(7,[3,1,5])
= pert(7,[1,5])
= pert(7,[5])
= pert(7,[])
= 0
```

9.3.3 Clonagem de lista

A função que cria um *clone* de uma lista `L`, na Figura 9.15, usa a seguinte ideia: (*base*) se `L` está vazia, seu clone é uma lista vazia; (*passo*) senão, seu clone é uma lista cujo primeiro nó guarda o primeiro item de `L` e um ponteiro para um clone do *resto* de `L`.

```
Lista clone(Lista L) {
   if( L == NULL ) return NULL;
   return no(L->item,clone(L->prox));
}
```

Figura 9.15 | Função para clonagem de lista.

Por exemplo, supondo que `L` aponta a lista `[3,1,5]`, a simulação da execução da chamada `clone(L)` é a seguinte:

```
clone([3,1,5])
= no(3,clone([1,5]))
= no(3,no(1,clone([5])))
= no(3,no(1,no(5,clone([]))))
= no(3,no(1,no(5,NULL)))
```

Note que a avaliação da expressão `no(3,no(1,no(5,NULL)))` resulta na criação de uma nova lista, contendo os itens `3`, `1` e `5` (isto é, um clone da lista `L`).

9.3.4 Exibição inversa de lista

A função que exibe uma lista `L` em ordem *inversa*, na Figura 9.16, usa a seguinte ideia: (*base*) se `L` está vazia, nenhum item precisa ser exibido; (*passo*) senão, exiba o *resto* de `L` em ordem inversa e, depois, exiba diretamente seu primeiro item.

```
void exibe_inv(Lista L) {
   if( L == NULL ) return;
   exibe_inv(L->prox);
   printf(fmt,L->item);
}
```

Figura 9.16 | Função para exibição de lista em ordem inversa.

A Figura 9.17 mostra a simulação da execução da chamada `exibe_inv(L)`.

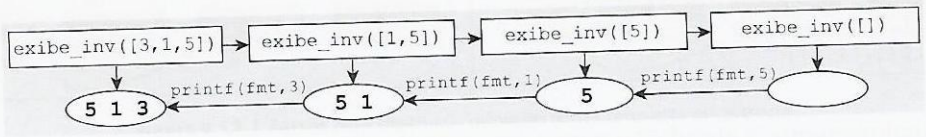


Figura 9.17 | Fluxo de execução para uma chamada `exibe_inv(L)`.

Exercícios

9.1 Crie o arquivo `lista.h`, contendo as definições de tipos e funções para listas encadeadas apresentadas nesse capítulo, e teste o programa a seguir.

```
#include <stdio.h>
#include "../ed/lista.h" // lista de int
int main(void) {
   Lista L = no(3,no(1,no(5,NULL)));
   exibe_inv(L);
   return 0;
}
```

9.2 Crie a função iterativa `ocorrencias(x,L)`, que informa quantas vezes o item `x` ocorre na lista `L`. Por exemplo, para `L` apontando a lista `[1,2,1,4,1]`, a chamada `ocorrencias(1,L)` deve devolver `3` como resposta.

9.3 Crie a função iterativa `ultimo(L)`, que devolve o último item da lista `L`. Por exemplo, para `L` apontando a lista `[a,b,c]`, a função deve devolver o item `c`.

9.4 Crie a função iterativa `inversa(L)`, que devolve a lista inversa de `L`. Por exemplo, para `L` apontando a lista `[7,9,2]`, a função deve devolver `[2,9,7]`.

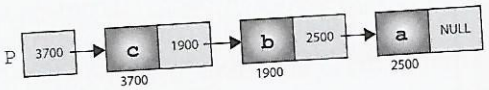
9.5 Crie a função recursiva `soma(L)`, que devolve a soma dos itens da lista `L`. Por exemplo, para `L` apontando a lista `[3,1,5,4]`, a função deve devolver `13`.

9.6 Crie a função recursiva `substitui(x,y,L)`, que substitui toda ocorrência do item `x` pelo item `y` na lista `L`. Por exemplo, se `L` aponta a lista `[b,o,b,o]`, após a chamada `substitui('o','a',L)`, `L` deverá apontar a lista `[b,a,b,a]`.

9.7 Crie a função recursiva `igual(A,B)`, que verifica se a lista `A` é igual à lista `B`. Por exemplo, se `I` aponta `[1,2,3]`, `J` aponta `[1,2,3]` e `K` aponta `[1,3,2]`, as chamadas `igual(I,J)` e `igual(I,K)` devem devolver `1` e `0`, respectivamente.

9.8 Crie a função recursiva `enesimo(n,L)`, que devolve o `n`-ésimo item da lista `L`. Por exemplo, para `L` apontando a lista `[a,b,c,d]`, a chamada `enesimo(3,L)` deve devolver o item `c`. Para `n` inválido, a função deve parar com erro fatal.

9.9 Na implementação *dinâmica encadeada* de pilha, os itens são mantidos numa lista (com um ponteiro `P` para seu início) e as inserções e remoções são feitas no início dessa lista. Crie as funções `empilha(x,&P)` e `desempilha(&P)`, para pilhas dinâmicas encadeadas, e faça um programa para testá-las.



9.10 Na implementação *dinâmica encadeada* de fila, os itens são mantidos numa lista *circular* (com um ponteiro `F` para o último nó inserido na lista, que aponta o primeiro), as inserções são feitas no final na lista e as remoções são feitas no início da lista. Crie as funções `enfileira(x,&F)` e `desenfileira(&F)`, para filas dinâmicas encadeadas, e faça um programa para testá-las.