

Capítulo 14

Listas encadeadas

Para representar um conjunto de dados, já vimos que podemos usar um vetor. O vetor é a forma mais primitiva de representar diversos elementos agrupados. Ao declarar um vetor, reservamos um espaço contíguo de memória para armazenar seus elementos. Este fato nos permite acessar qualquer um de seus elementos a partir do ponteiro para o primeiro elemento. Dizemos que o vetor é uma estrutura que possibilita o acesso randômico aos elementos, pois podemos acessar qualquer elemento diretamente. No entanto, o vetor não é uma estrutura de dados muito flexível, pois precisamos dimensioná-lo com um número máximo de elementos. O uso de vetores dinâmicos, através de realocação da memória, ameniza o problema, mas não necessariamente é uma solução em todas as situações, pois o custo computacional para realocar um vetor é alto.

A alternativa consiste em utilizar estruturas de dados que cresçam à medida que precisamos armazenar novos elementos (e diminuam à medida que precisamos retirar elementos armazenados anteriormente). Estas estruturas dinâmicas fazem uma alocação de memória para cada novo elemento armazenado.

Nas seções a seguir, discutiremos a estrutura de dados conhecida como *lista encadeada*. As listas encadeadas são amplamente usadas para implementar diversas outras estruturas de dados com semânticas próprias, que serão tratadas nos capítulos seguintes.

14.1 Listas simplesmente encadeadas

Numa lista encadeada, para cada novo elemento inserido na estrutura, alocamos um espaço de memória para armazená-lo. Desta forma, o espaço total de memória gasto pela estrutura é proporcional ao número de elementos nela armazenado. No entanto, não podemos garantir que os elementos armazenados na lista ocuparão um espaço de memória contíguo; portanto, não temos acesso direto aos elementos da lista. Para que seja possível acessar todos os elementos da lista, devemos explicitamente guardar o encadeamento dos elementos, o que é feito armazenando-se, junto com a informação, um ponteiro para o próximo elemento (ou nó) da lista. A Figura 14.1 ilustra o arranjo da memória de uma lista encadeada.

A estrutura consiste numa sequência encadeada de elementos, em geral chamados de *nós da lista*. Um nó da lista é representado por uma estrutura que contém, conceitualmente, dois campos: a informação armazenada e o ponteiro para o próximo elemento da lista. A partir do primeiro nó, podemos alcançar o segundo seguindo o encadeamento e

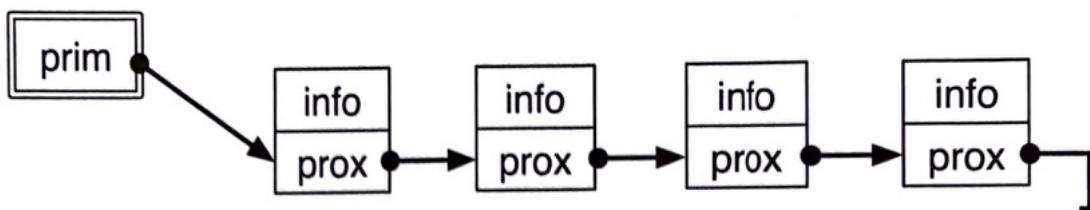


Figura 14.1: Arranjo da memória de uma lista encadeada.

assim por diante. O último nó da lista armazena, como próximo nó, um ponteiro inválido, com valor `NULL`, sinalizando que não existe um próximo nó. A lista em si, portanto, deve ser representada por um ponteiro para o primeiro nó.

Podemos então considerar dois tipos diferentes de estruturas: uma estrutura para representar a lista em si e outra estrutura para representar o nó da lista. Para exemplificar, vamos considerar inicialmente que as informações armazenadas são números inteiros:

```

typedef struct lista Lista;
typedef struct listano Listano;

struct lista {
    Listano* prim;
};

struct listano {
    int info;
    Listano* prox;
};
  
```

Devemos notar que a estrutura que representa o nó da lista é uma estrutura autoreferenciada, pois, além do campo que armazena a informação (no caso, um número inteiro), há um campo que é um ponteiro para uma próxima estrutura do mesmo tipo.

Considerando estas definições, podemos apresentar as principais funções necessárias para manipular listas encadeadas (de inteiros, no caso).

14.1.1 Função de criação

A função que cria uma lista vazia deve ter como valor de retorno uma lista sem nenhum elemento. Como a lista armazena o ponteiro para o primeiro elemento, uma lista vazia é representada por armazenar o valor `NULL`, pois não existem elementos na lista. A função tem como valor de retorno o ponteiro para a estrutura de lista criada dinamicamente. Uma possível implementação da função de criação é mostrada a seguir (usamos o prefixo `lst` para indicar que tratam-se de funções para manipular listas encadeadas):

```

/* função de criação: retorna uma lista inicialmente vazia */
Lista* lst_cria (void)
{
    Lista* l = (Lista*) malloc(sizeof(Lista));
    l->prim = NULL;
    return l;
}
  
```

14.1.2 Função de inserção

Uma vez criada a lista vazia, podemos inserir nela novos elementos. Para cada elemento inserido na lista, devemos alocar dinamicamente a memória necessária para armazenar um novo nó e encadeá-lo na lista existente. A função de inserção mais simples insere o novo elemento no início da lista.

Uma possível implementação dessa função é mostrada a seguir. Devemos notar que o ponteiro que representa a lista deve ser passado como parâmetro, além da informação do novo elemento.

```
/* inserção no início */
void lst_insere (Lista* l, int v)
{
    ListaNo* novo = (ListaNo*) malloc(sizeof(ListaNo));
    novo->info = v;
    novo->prox = l->prim;
    l->prim = novo;
}
```

Esta função aloca dinamicamente o espaço para armazenar o novo nó da lista, guarda a informação no novo nó e faz este nó apontar para (isto é, ter como próximo nó) o nó que era o primeiro da lista. O primeiro nó da lista é então atualizado. A Figura 14.2 ilustra a operação de inserção de um novo elemento no início da lista.

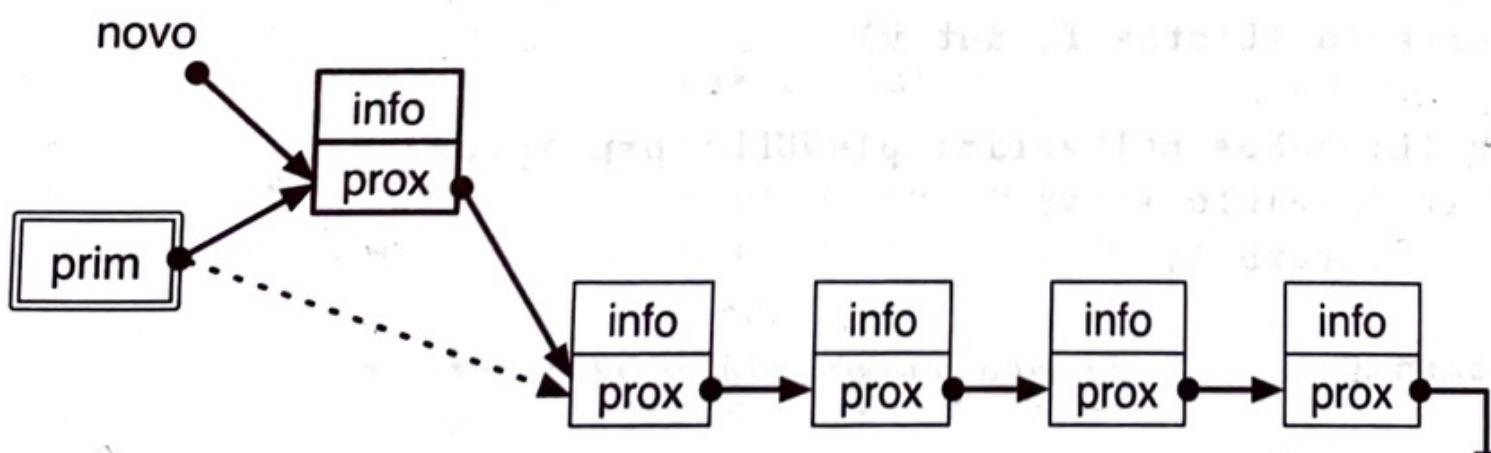


Figura 14.2: Inserção de um novo elemento no início da lista.

A seguir, ilustramos um trecho de código que cria uma lista inicialmente vazia e insere nela novos elementos:

```
int main (void)
{
    Lista* l = lst_cria();           /* cria uma lista não inicializada */
    lst_insere(l, 23);              /* insere na lista o elemento 23 */
    lst_insere(l, 45);              /* insere na lista o elemento 45 */
    ...
    return 0;
}
```

14.1.3 Função que percorre os elementos da lista

Para ilustrar a implementação de uma função que percorre todos os elementos da lista, vamos considerar a criação de uma função que imprime os valores associados aos elementos armazenados numa lista. Uma possível implementação dessa função é mostrada a seguir:

```
/* função imprime: imprime valores dos elementos */
void lst_imprime (Lista* l)
{
    ListaNo* p;
    for (ListaNo* p=l->prim; p!=NULL; p=p->prox)
        printf("info = %d\n", p->info);
}
```

Recordamos que, para percorrer os elementos de um vetor, usamos uma variável auxiliar inteira para armazenar o índice de cada elemento. No caso da lista encadeada, a variável auxiliar tem que ser um ponteiro, usada para armazenar o endereço de cada elemento. Dentro do laço da função, a variável *p* aponta para cada um dos elementos da lista, do primeiro até o último. Note ainda que só conseguimos percorrer os elementos do primeiro ao último, pois só temos o ponteiro para o próximo nó. E também que não temos acesso randômico aos elementos da lista: para acessar determinado elemento, temos que percorrer a lista desde o início, até que o elemento de interesse seja alcançado.

Outra função útil consiste em verificar se determinado elemento está presente na lista. A função recebe a informação referente ao elemento que queremos buscar e fornece como valor de retorno 1 se o elemento for encontrado ou 0 se não for:

```
/* função pertence: verifica se um elemento está na lista */
int pertence (Lista* l, int v)
{
    ListaNo* p;
    for (ListaNo* p=l->prim; p!=NULL; p=p->prox) {
        if (p->info == v)
            return 1;
    }
    return 0; /* não encontrou o elemento */
}
```

14.1.4 Manutenção da lista ordenada

A função de inserção apresentada armazena os elementos na lista na ordem inversa à ordem de inserção, pois um novo elemento é sempre inserido no início da lista. Se quisermos manter os elementos na lista numa determinada ordem, temos que encontrar a posição correta para inserir o novo elemento. Essa função não é eficiente, pois temos que percorrer a lista, elemento por elemento, para achar a posição de inserção. Se a ordem de armazenamento dos elementos dentro da lista não for relevante, optamos por fazer inserções no início, pois o custo computacional dessa operação independe do número de elementos na lista.

No entanto, se desejarmos manter os elementos em ordem, cada novo elemento deve ser inserido na ordem correta. Para exemplificar, vamos considerar que queremos manter nossa lista de números inteiros em ordem crescente. A função de inserção, neste caso, tem a mesma assinatura da função de inserção mostrada, mas percorre os elementos da

lista a fim de encontrar a posição correta para a inserção do novo elemento. Com isso, temos que saber inserir um elemento no meio da lista. A Figura 14.3 ilustra a inserção de um elemento no meio da lista.

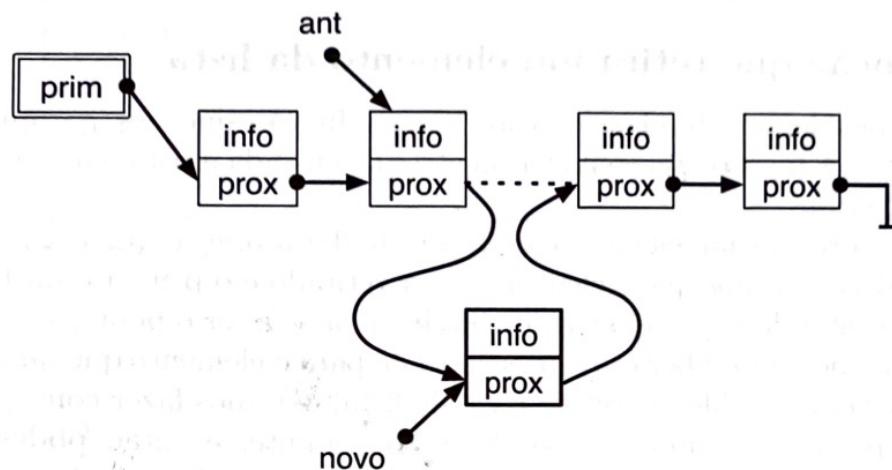


Figura 14.3: Inserção de um elemento no meio da lista.

Conforme ilustrado na figura, devemos localizar o elemento da lista que irá preceder o elemento novo a ser inserido. De posse do ponteiro para esse elemento anterior (*ant*), podemos encadear o novo elemento na lista. O novo apontará para o próximo elemento na lista e o elemento precedente apontará para o novo. Se o elemento tiver que ser inserido antes do primeiro elemento, recaímos no caso de inserção no início da lista. O código a seguir ilustra a implementação dessa função.

```
/* função insere_ordenado: insere elemento em ordem */
void lst_insere_ordenado (Lista* l, int v)
{
    ListaNo* ant = NULL;      /* ponteiro para elemento anterior */
    ListaNo* p = l->prim;    /* ponteiro para percorrer a lista */

    /* localiza posição de inserção */
    while (p != NULL && p->info < v) {
        ant = p;
        p = p->prox;
    }

    /* cria novo elemento */
    ListaNo* novo = (ListaNo*) malloc(sizeof(ListaNo));
    novo->info = v;

    /* encadeia elemento */
    if (ant == NULL) { /* insere elemento no início */
        novo->prox = l->prim;
        l->prim = novo;
    }
    else {           /* insere elemento no meio da lista */
        novo->prox = ant->prox;
        ant->prox = novo;
    }
}
```

Devemos notar que essa implementação funciona corretamente se o elemento tiver que ser inserido no final da lista, não sendo portanto um caso particular adicional (equivale a inserir no meio).

14.1.5 Função que retira um elemento da lista

Vamos agora considerar a implementação de uma função que nos permita retirar um elemento da lista. A função tem como parâmetros de entrada a lista e o valor do elemento que desejamos retirar.

A função para retirar um elemento da lista é similar à função que insere os elementos em ordem. Se descobrirmos que o elemento a ser retirado é o primeiro da lista, devemos fazer com que o valor do primeiro elemento da lista passe a ser o ponteiro para o segundo elemento, e então podemos liberar o espaço alocado para o elemento que queremos retirar. Se o elemento a ser removido estiver no meio da lista, devemos fazer com que o elemento anterior a ele passe a apontar para o elemento seguinte, e então podemos liberar o elemento que queremos retirar. Portanto, aqui também precisamos do ponteiro para o elemento anterior para poder acertar o encadeamento da lista. As Figuras 14.4 e 14.5 ilustram as operações de remoção.

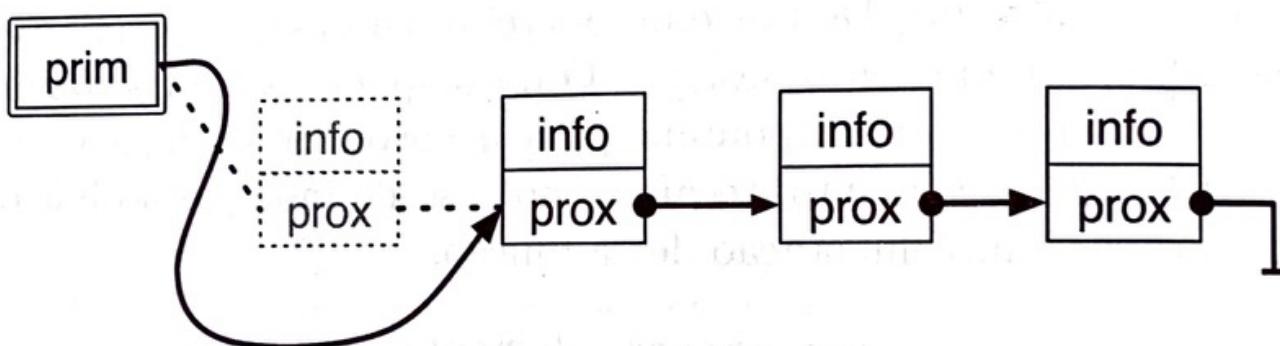


Figura 14.4: Remoção do primeiro elemento da lista.

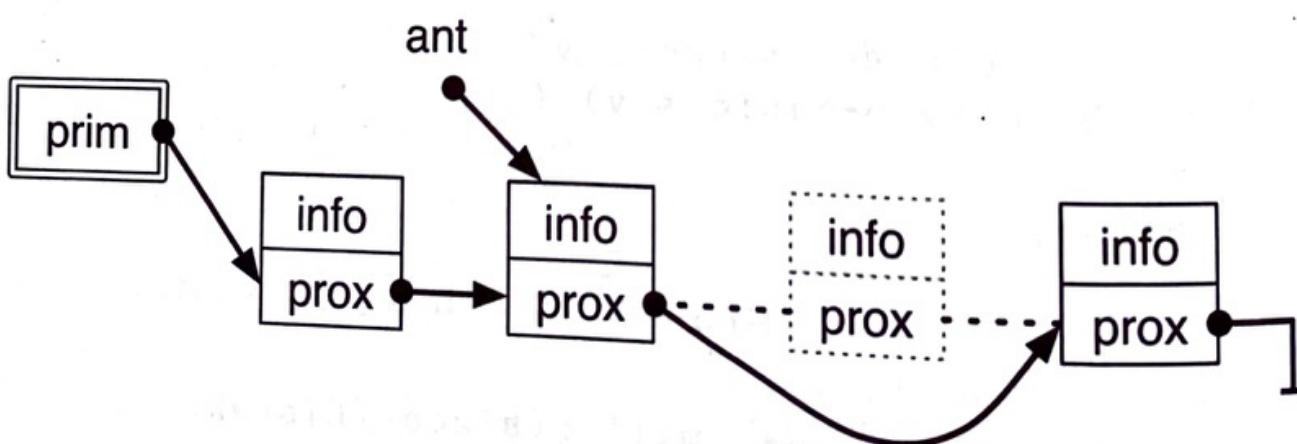


Figura 14.5: Remoção de um elemento no meio da lista.

Uma possível implementação da função para retirar um elemento da lista é mostrada a seguir. Inicialmente, busca-se o elemento que se deseja retirar, guardando uma referência para o elemento anterior.

```
/* função retira: retira elemento da lista */
void lst_retira (Lista* l, int v)
{
    ListaNo* ant = NULL;
    ListaNo* p = l->prim;           /* ponteiro para elemento anterior */
                                    /* ponteiro para percorrer a lista */
    ...
```

```

/* procura elemento na lista guardando anterior */
while (p != NULL && p->info != v) {
    ant = p;
    p = p->prox;
}

if (p != NULL) { /* verifica se achou elemento */
    /* retira elemento */
    if (ant == NULL) { /* retira elemento do inicio */
        l->prim = p->prox;
    }
    else { /* retira elemento do meio da lista */
        ant->prox = p->prox;
    }
    free(p); /* libera elemento (nó) */
}
}

```

Análogo à inserção, o caso de retirar o último elemento é naturalmente tratado como uma retirada do meio da lista na implementação mostrada.

14.1.6 Função para verificar se vazia e função para liberar

Pode ser útil implementar uma função para verificar se uma lista está vazia ou não. A função recebe a lista e retorna 1 se estiver vazia ou 0 se não estiver vazia. Como sabemos, uma lista está vazia se seu primeiro elemento é NULL:

```

/* função vazia: retorna 1 se vazia ou 0 se não vazia */
int lst_vazia (Lista* l)
{
    return (l->prim == NULL);
}

```

Para completar o conjunto de funções básicas que manipulam uma lista, devemos considerar a função que destrói a lista, liberando todos os elementos alocados. Uma implementação dessa função é mostrada a seguir. A função primeiro libera o encadeamento dos elementos e depois libera a estrutura da lista. Para liberar o encadeamento, a função percorre elemento por elemento, liberando-os. É importante observar que devemos guardar a referência para o próximo elemento antes de liberar o elemento corrente – se liberássemos o elemento e depois tentássemos acessar o encadeamento, estariamos acessando um espaço de memória que não estaria mais reservado para nosso uso.

```

void lst_libera (Lista* l)
{
    ListaNo* p = l->prim;
    while (p != NULL) {
        ListaNo* t = p->prox; /* guarda referência p/ próx elemento */
        free(p); /* libera a memória apontada por p */
        p = t; /* faz p apontar para o próximo */
    }
    free(l);
}

```

14.1.7 TAD Lista de inteiros

Com base na implementação exemplificada, podemos criar um tipo abstrato de dados para representar uma lista encadeada de valores inteiros. A interface do módulo pode ser dada pelo arquivo `lista.h`:

```
/* TAD: lista de inteiros */

typedef struct lista Lista;

Lista* lst_cria (void);
void lst_libera (Lista* l);

void lst_insere (Lista* l, int v);
void lst_insere_ordenado (Lista* l, int v);
void lst_retira (Lista* l, int v);

int lst_vazia (Lista* l);
int lst_pertence (Lista* l, int v);
void lst_imprime (Lista* l);
```

Note que nessa interface optamos por encapsular inclusive o tipo que representa um nó da lista. Utilizando essa interface, podemos criar um programa que utiliza as funções de lista exportadas.

```
#include <stdio.h>
#include "lista.h"

int main (void)
{
    Lista* l = lst_cria(); /* cria lista vazia */
    lst_insere(l, 23); /* insere na lista o elemento 23 */
    lst_insere(l, 45); /* insere na lista o elemento 45 */
    lst_insere(l, 56); /* insere na lista o elemento 56 */
    lst_insere(l, 78); /* insere na lista o elemento 78 */
    lst_imprime(l); /* imprimirá: 78 56 45 23 */
    lst_retira(l, 78);
    lst_imprime(l); /* imprimirá: 56 45 23 */
    lst_retira(l, 45);
    lst_imprime(l); /* imprimirá: 56 23 */
    lst_libera(l);
    return 0;
}
```

14.2 Implementações recursivas

A estrutura de encadeamento de uma lista pode ser definida de maneira recursiva. Podemos dizer que o encadeamento é:

- vazio (isto é, sem nenhum nó); ou
- um nó seguido de um (sub) encadeamento.

No primeiro caso, o ponteiro que representa o primeiro nó da lista é **NULL**; no segundo caso, o ponteiro para o segundo, armazenado como próximo nó do primeiro, representa o primeiro nó do subencadeamento. A Figura 14.6 mostra uma representação desta definição recursiva do encadeamento de uma lista.

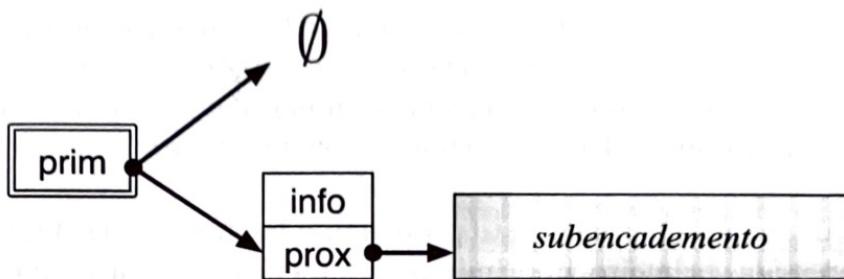


Figura 14.6: Representação gráfica da definição recursiva.

Se trabalharmos com esta definição, devemos considerar que o subencadeamento é, por si, uma estrutura abstrata. Temos acesso apenas ao primeiro nó do encadeamento; para acessar os demais nós, devemos chamar a função recursivamente, passando o acesso ao primeiro elemento do subencadeamento.

Podemos então implementar funções de listas recursivamente. Note que a estrutura que representa a lista em si (o ponteiro para o primeiro elemento) não faz parte da definição recursiva. A recursão é definida para o encadeamento – estrutura do nó da lista.

Para exemplificar, vamos considerar uma função para imprimir os elementos da lista. Devemos ter a função da interface que chama a função auxiliar que imprime os elementos do encadeamento. Nesta função auxiliar, devemos seguir a definição recursiva na implementação. Assim, devemos primeiramente checar se o encadeamento é vazio (condição de contorno da recursão). Se o encadeamento for vazio, não temos nada para imprimir. Caso contrário, o encadeamento é composto pelo primeiro nó, dado por *p*, e por um subencadeamento, representado por *p->prox*. Assim, devemos imprimir a informação associada ao primeiro nó, acessando *p->info*, e então imprimir as informações do subencadeamento, usando a própria função auxiliar. Uma possível implementação dessa função é mostrada a seguir:

```

/* Função auxiliar recursiva para imprimir o encadeamento */
static void imprime_nos (ListaNo* p)
{
    if (p != NULL) {
        printf("%d\n", p->info);      /* imprime primeiro elemento */
        imprime_nos(p->prox);       /* imprime subencadeamento */
    }
}

/* Função da interface que usa auxiliar recursiva */
void lst_imprime_rec (Lista* l)
{
    imprime_nos(l->prim);
}

```

Não recomendamos tentar seguir, passo a passo, a execução de uma implementação recursiva e sim entendê-la com base apenas na definição recursiva do objeto – no caso, o encadeamento da lista.

Em alguns casos, a implementação recursiva é bem mais simples do que uma implementação iterativa (que percorre os elementos com repetição). Por exemplo, se queirarmos imprimir os elementos da lista em ordem inversa, do último para o primeiro. Para fazer isso de forma não recursiva, teríamos que percorrer a lista para achar o último elemento; em seguida, percorrer novamente a lista para achar o penúltimo; e assim por diante. Isto é necessário pois não temos como acessar o anterior.

Com a implementação recursiva, este problema é facilmente resolvido, bastando invertida a ordem da impressão: primeiro imprimimos o subencadeamento (que irá imprimir os elementos em ordem inversa) e, em seguida, imprimimos o primeiro elemento.

```
/* Função auxiliar recursiva para imprimir o encadeamento */
static void imprime_nos_rev (ListaNo* p)
{
    if (p != NULL) {
        imprime_nos_rev(p->prox); /* imprime subencadeamento */
        printf("%d\n", p->info);   /* imprime primeiro elemento */
    }
}

/* Função da interface que usa auxiliar recursiva */
void lst_imprime_rev (Lista* l)
{
    imprime_nos_rev(l->prim);
}
```

Como dissemos, esta definição recursiva só nos permite processar o primeiro elemento da lista. Os demais são tratados pela recursão. Com isso, podemos pensar na implementação de uma função que retira o elemento da lista de forma recursiva. Como vimos, retirar um elemento do topo da lista é fácil, mas retirar do meio da lista envolve mais esforço. A implementação recursiva pode então facilitar a codificação, já que só poderemos retirar o próprio elemento do (sub-)encadeamento.

A função auxiliar recursiva só retira um elemento se ele for o primeiro do encadeamento. Se o elemento que queremos retirar não for o primeiro, chamamos a função recursivamente para retirar o elemento do subencadeamento. Aqui, precisamos tomar um cuidado adicional: ao retirar um elemento do encadeamento, o encadeamento precisa ser representado por um novo primeiro nó; assim, é necessário que a função auxiliar tenha como valor de retorno o eventual novo primeiro nó.

```
/* Função auxiliar retira recursiva */
static ListaNo* retira_nos (ListaNo* p, int v)
{
    if (p != NULL) {
        if (p->info == v) { /* verifica se elemento é o primeiro */
            ListaNo* t = p; /* temporário para poder liberar */
            p = p->prox;   /* primeiro nó é alterado */
            free(t);
        }
    }
}
```

```

    else {
        p->prox = retira_nos(p->prox,v); /* retira de sublista */
    }
}
return p; /* retorna (eventual novo) primeiro nó */
}

/* Função da interface que usa auxiliar recursiva */
void lst_retira_rec (Lista* l, int v)
{
    l->prim = retira_nos(l->prim,v);
}

```

Note a necessidade de reatribuir o valor do primeiro nó na chamada recursiva, já que a função pode alterar o valor do subencadeamento.

A função para liberar um encadeamento também pode ser escrita recursivamente, de forma bastante simples. Nessa função, se o encadeamento não for vazio, liberamos primeiro o subencadeamento e depois liberamos a lista.

```

static void libera_nos (ListaNo* p)
{
    if (p != NULL)
    {
        libera_nos(p->prox);
        free(p);
    }
}

void lst_libera_rec (Lista* l)
{
    libera_nos(l->prim); /* libera encadeamento */
    free(l); /* libera estrutura da lista */
}

```

A utilização de implementações recursivas para listas encadeadas é apenas uma opção. Em geral, a implementação não recursiva é mais eficiente do ponto de vista do esforço computacional dispensado, pois minimiza o número de chamadas de funções, que são operações relativamente caras. No entanto, como dissemos, algumas implementações podem ficar mais simples se feitas de forma recursiva.

Para ilustrar esta discussão, vamos considerar como último exemplo a implementação de uma função para testar se duas listas dadas são iguais. Duas listas são consideradas iguais se elas têm a mesma sequência de elementos, naturalmente com o mesmo número de elementos. O protótipo dessa função é dado por:

```
int lst_igual (Lista* l1, Lista* l2);
```

A implementação dessa função de forma não recursiva requer que tenhamos dois ponteiros auxiliares para percorrer as duas listas, simultaneamente, comparando as informações associadas a cada par de elementos. Se encontrarmos informações diferentes, podemos concluir que as listas são diferentes. Esse teste deve ser feito até que uma das

listas (ou as duas) chegue ao fim. Fora do laço, testamos se os dois ponteiros auxiliares são iguais. Se forem, significa que ambos são NULL, isto é, as duas listas têm o mesmo número de elementos. Uma implementação dessa função é mostrada a seguir:

```
int lst_igual (Lista* l1, Lista* l2)
{
    ListaNo* p1;
    ListaNo* p2;
    for (p1=l1->prim, p2=l2->prim;
        p1!=NULL && p2!=NULL;
        p1=p1->prox, p2=p2->prox) {
        if (p1->info != p2->info)
            return 0;
    }
    return p1==p2;
}
```

Uma implementação recursiva dessa função deve ser pensada com base na definição recursiva do encadeamento. Primeiramente, temos que testar os casos bases, nos quais os encadeamentos podem ser vazios. Dessa forma, verificamos se os dois encadeamentos dados são vazios. Se forem, logicamente eles são iguais. Se não forem ambos vazios, devemos verificar se um deles é vazio. Se for, concluímos que tratam-se de encadeamentos diferentes. Se ambos não forem vazios, devemos testar a igualdade entre as informações associadas aos primeiros nós dos encadeamentos e verificar a igualdade dos subencadeamentos. Uma possível implementação é mostrada a seguir:

```
static int nos_iguais (ListaNo* p1, ListaNo* p2)
{
    if (p1==NULL && p2==NULL)
        return 1;
    else if (p1==NULL || p2==NULL)
        return 0;
    else
        return (p1->info==p2->info) && nos_iguais(p1->prox,p2->prox);
}

int lst_igual_rec (Lista* l1, Lista* l2)
{
    return nos_iguais(l1->prim,l2->prim);
}
```

14.2.1 Listas circulares

Algumas aplicações necessitam representar conjuntos cíclicos. Por exemplo, as arestas que delimitam um polígono podem ser agrupadas por uma estrutura circular. Para esses casos, podemos usar listas circulares.

Numa lista circular, o último elemento tem como próximo o primeiro elemento da lista, formando um ciclo. A rigor, neste caso, não faz sentido falar em primeiro ou último elemento. A lista pode ser representada por um ponteiro para um elemento inicial

qualquer da lista. A Figura 14.7 ilustra o arranjo da memória para a representação de uma lista circular.

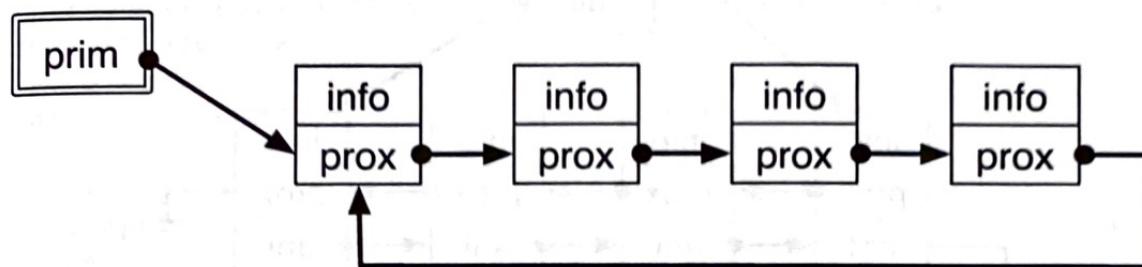


Figura 14.7: Arranjo da memória de uma lista circular.

Para percorrer os elementos de uma lista circular, visitamos todos os elementos a partir do ponteiro do elemento inicial até alcançar novamente esse mesmo elemento. O código a seguir exemplifica uma forma de percorrer os elementos. Para simplificar, consideramos uma lista que armazena valores inteiros. Devemos salientar que o caso em que a lista é vazia ainda deve ser tratado (se a lista é vazia, o ponteiro para um elemento inicial vale `NULL`).

```

void lcirc_imprime (Lista* l)
{
    ListaNo* p = l->prim; /* faz p apontar para 'nó inicial' */
    if (p != NULL) do { /* se não vazia, percorre */
        printf("%d\n", p->info); /* imprime informação do nó */
        p = p->prox; /* avança para o próximo nó */
    } while (p != l->prim);
}
  
```

14.3 Listas duplamente encadeadas

A estrutura de lista encadeada vista nas seções anteriores caracteriza-se por formar um encadeamento simples entre os elementos: cada elemento armazena um ponteiro para o próximo elemento da lista. Desta forma, não temos como percorrer eficientemente os elementos em ordem inversa, isto é, do final para o início da lista. O encadeamento simples também dificulta a retirada de um elemento da lista. Mesmo se tivermos o ponteiro do elemento que desejamos retirar, temos que percorrer a lista, elemento por elemento, para encontrar o elemento anterior, pois, dado o ponteiro para determinado elemento, não temos como acessar diretamente seu elemento anterior.

Para solucionar esses problemas, podemos formar o que chamamos de *listas duplamente encadeadas*. Nelas, cada elemento tem um ponteiro para o próximo elemento e um ponteiro para o elemento anterior. Desta forma, dado um elemento, podemos acessar ambos os elementos adjacentes: o próximo e o anterior. Se tivermos um ponteiro para o último elemento da lista, podemos percorrer a lista em ordem inversa, bastando acessar continuamente o elemento anterior, até alcançar o primeiro elemento da lista, que não tem elemento anterior (o ponteiro do elemento anterior vale `NULL`). A Figura 14.8 esquematiza a estruturação de uma lista duplamente encadeada.

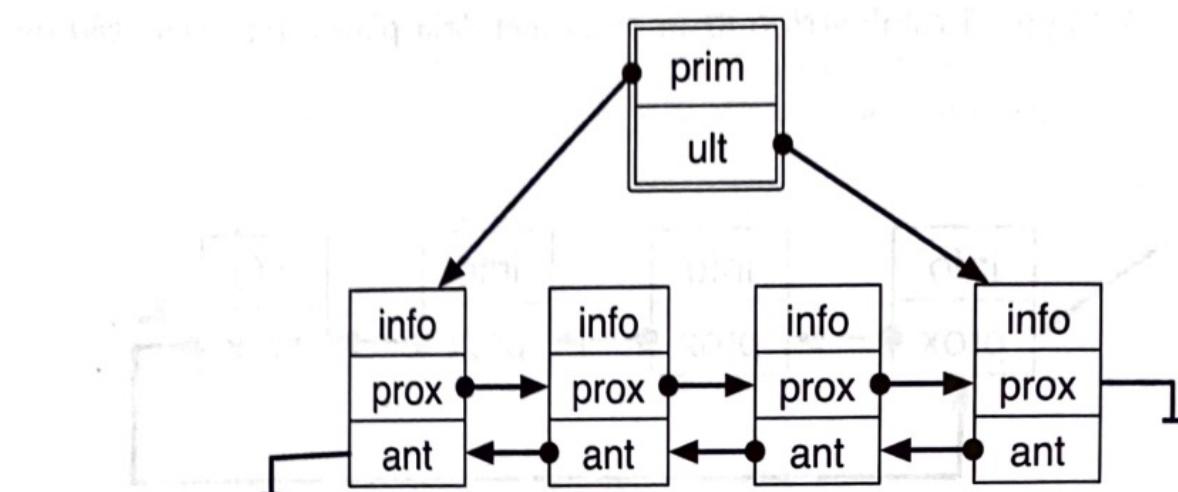


Figura 14.8: Arranjo da memória de uma lista duplamente encadeada.

Para exemplificar a implementação de listas duplamente encadeadas, vamos novamente considerar o exemplo simples no qual queremos armazenar valores inteiros na lista. A estrutura da lista pode armazenar dois ponteiros: um para o primeiro nó e outro para o último nó. O nó da lista armazena, além da informação, ponteiros para o próximo nó e para o nó anterior:

```

typedef struct lista2 Lista2;
typedef struct listano2 ListaNo2;

struct lista2 {
    ListaNo2* prim;
    ListaNo2* ult;
};

struct listano2 {
    int info;
    ListaNo2* ant;
    ListaNo2* prox;
};

```

Com base nessas definições, exemplificaremos a seguir a implementação de algumas funções que manipulam listas duplamente encadeadas. A função que cria uma lista vazia é simples:

```

Lista2* lst2_cria (void)
{
    Lista2* l = (Lista2*) malloc(sizeof(Lista2));
    l->prim = NULL;
    l->ult = NULL;
    return l;
}

```

No caso da lista duplamente encadeada, as inserções no início e no final são simples e eficientes. Note o encadeamento duplo dos novos elementos na lista:

```

/* inserção no início */
void lst2_insere_inicio (Lista2* l, int v)
{
    ListaNo2* novo = (ListaNo2*) malloc(sizeof(ListaNo2));
    novo->info = v;
    novo->prox = l->prim; /* próximo de novo é antigo primeiro */
    novo->ant = NULL; /* anterior é nulo (novo é primeiro) */

    if (l->prim != NULL) /* testa se lista é vazia */
        l->prim->ant = novo; /* novo é anterior do antigo primeiro */
    else
        l->ult = novo; /* novo também é último */
    l->prim = novo; /* novo é primeiro */
}

/* inserção no final */
void lst2_insere_final (Lista2* l, int v)
{
    ListaNo2* novo = (ListaNo2*) malloc(sizeof(ListaNo2));
    novo->info = v;
    novo->ant = l->ult; /* anterior de novo é antigo último */
    novo->prox = NULL; /* próximo é nulo, pois novo é último */
    if (l->ult != NULL) /* testa se lista é vazia */
        l->ult->prox = novo; /* novo é próximo do antigo último */
    else
        l->prim = novo; /* novo também é primeiro */
    l->ult = novo; /* novo é último */
}

```

Na inserção no início, o novo elemento tem como próximo elemento o antigo primeiro elemento da lista e como anterior o valor `NULL`. A seguir, a função testa se a lista não era vazia; neste caso, o elemento anterior do então primeiro elemento passa a ser o novo elemento; caso contrário, o novo elemento passa a ser também o último elemento da lista. De qualquer forma, o novo elemento passa a ser o primeiro da lista. A Figura 14.9 ilustra a operação de inserção de um novo elemento no início da lista. Na inserção no final, o processo é similar.

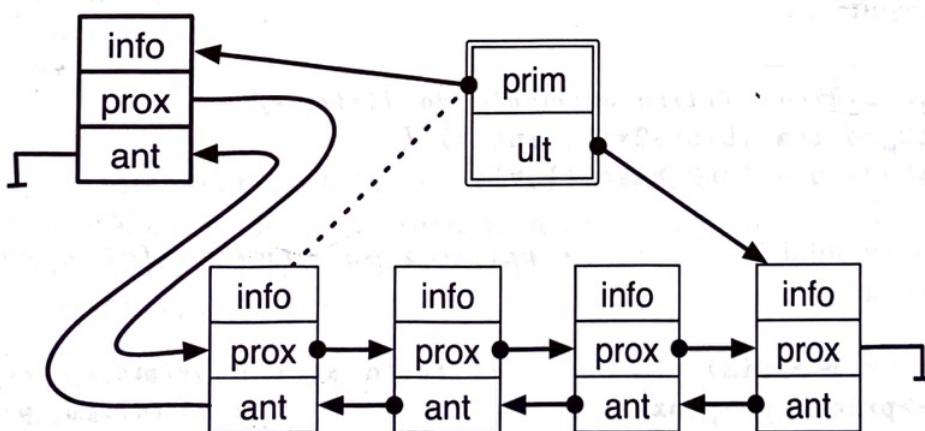


Figura 14.9: Inserção de um novo elemento no início de lista duplamente encadeada.

A função de busca recebe a informação referente ao elemento que queremos buscar e tem como valor de retorno o ponteiro do nó da lista que representa o elemento. Caso o elemento não seja encontrado na lista, o valor retornado é NULL:

```
/* função busca: busca um elemento na lista */
ListaNo2* lst2_busca (Lista2* l, int v)
{   ListaNo2 *p;
    for (ListaNo2 *p=l->prim; p!=NULL; p=p->prox)
        if (p->info == v)
            return p;
    return NULL;           /* não achou o elemento */
}
```

Conforme notamos, esta função tem uma implementação igual ao caso da lista simplesmente encadeada, pois só usamos o ponteiro para o próximo elemento. Na busca, a única diferença é que seria possível fazer a busca de trás para frente, mas isso não necessariamente é melhor.

A função que retira um elemento da lista requer o acerto do encadeamento duplo. Em contrapartida, podemos retirar um elemento da lista conhecendo apenas o ponteiro para esse elemento. Desta forma, podemos usar a função de busca apresentada para localizar o elemento e, em seguida, acertar o encadeamento, liberando o elemento ao final. Se p representa o ponteiro do elemento que desejamos retirar, para acertar o encadeamento devemos conceitualmente fazer:

```
p->ant->prox = p->prox;
p->prox->ant = p->ant;
```

Isto é, o anterior passa a apontar para o próximo e o próximo passa a apontar para o anterior. Quando p apontar para um elemento no meio da lista, as duas atribuições são suficientes para efetivamente acertar o encadeamento da lista. No entanto, se p for um elemento no extremo da lista, devemos considerar as condições de contorno. Se p for o último elemento, não podemos escrever `p->prox->ant`, pois `p->prox` é NULL. Analogamente, se p apontar para o primeiro elemento, não podemos escrever `p->ant->prox`; além disso, temos que atualizar o valor da lista, pois o primeiro elemento e/ou último elemento será removido. Uma implementação da função para retirar um elemento é mostrada a seguir:

```
/* função retira: retira elemento da lista */
void lst2_retira (Lista2* l, int v) {
    ListaNo2* p = lst2_busca(l,v);

    if (p == NULL)           /* verifica se elemento foi encontrado */
        return;

    if (p == l->prim)        /* testa se é o primeiro elemento */
        l->prim = p->prox;    /* atualiza primeiro */
    else
        p->ant->prox = p->prox; /* faz anterior apontar p/ próximo */
```

```

if (p == l->ult) /* testa se é o último elemento */
    l->ult = p->ant; /* atualiza último */
else
    p->prox->ant = p->ant; /* faz próximo apontar p/ anterior */
    free(p); /* libera memória do nó */
}

```

Uma lista circular também pode ser construída com encadeamento duplo. Neste caso, o que seria o último elemento da lista passa a ter como próximo o primeiro elemento, que, por sua vez, passa a ter o último como anterior. Com essa construção podemos percorrer a lista nos dois sentidos, a partir do ponteiro para um elemento qualquer.

14.4 Listas de tipos estruturados

Nos exemplos anteriores, trabalhamos sempre com informações simples, pois tínhamos como principal objetivo discutir a estrutura das listas. Logicamente, a informação associada a cada nó de uma lista encadeada pode ser mais complexa. Como veremos, independente da informação armazenada e da forma como a informação é representada internamente, o encadeamento dos elementos da lista não é alterado. As funções apresentadas para manipular listas de inteiros podem ser facilmente adaptadas para tratar listas de outros tipos. Para simplificar esta exposição, vamos discutir a representação de tipos estruturados numa lista simplesmente encadeada. As mesmas técnicas de programação podem ser usadas em outras estruturas de listas.

Conforme mencionamos, um nó de uma lista encadeada contém basicamente dois componentes: o encadeamento e a informação armazenada. Assim, a estrutura de um nó para representar uma lista de números inteiros é dada por:

```

struct listano {
    int info;
    ListaNo* prox;
};

```

Analogamente, se quisermos representar uma lista de números reais, podemos definir a estrutura do nó como sendo:

```

struct listano {
    float info;
    ListaNo* prox;
};

```

A informação armazenada na lista não precisa ser necessariamente um dado simples. Podemos, por exemplo, considerar a construção de uma lista para armazenar um conjunto de retângulos, com cada retângulo sendo definido pela base *b* e pela altura *h*. Assim, a estrutura do nó pode ser dada por:

```

struct listano {
    float b;
    float h;
    ListaNo* prox;
};

```

Com isso, uma função auxiliar para alocar um nó dessa lista inicializando a informação pode ser dada por:

```
static ListaNo* aloca (float b, float h)
{
    ListaNo* p = (ListaNo*)malloc(sizeof(ListaNo));
    p->b = b;
    p->h = h;
    return p;
}
```

Essa mesma composição de nó pode ser escrita de forma mais clara se definirmos um tipo adicional que represente a informação. Podemos definir um tipo `Retangulo` e usá-lo para representar a informação armazenada na lista.

```
typedef struct retangulo Retangulo;
struct retangulo {
    float b;
    float h;
};

struct listano {
    Retangulo info;
    ListaNo* prox;
};
```

Assim, a nossa função auxiliar ficaria:

```
static ListaNo* aloca (float b, float h)
{
    ListaNo* p = (ListaNo*)malloc(sizeof(ListaNo));
    p->info.b = b;
    p->info.h = h;
    return p;
}
```

Aqui, a informação volta a ser representada por um único campo (`info`), que é uma estrutura. Ainda mais interessante é termos o campo da informação representado por um ponteiro para uma estrutura, em vez da estrutura em si.

```
struct lista {
    Retangulo *info;
    ListaNo* prox;
};
```

Neste caso, para alocar um nó, temos que fazer duas alocações dinâmicas: uma para criar a estrutura do retângulo e outra para criar a estrutura do nó. Outra possibilidade seria a função já receber o ponteiro do retângulo que deve ser inserido na lista. O código a seguir ilustra a função com as duas alocações:

```

static ListaNo* aloca (float b, float h)
{
    Retangulo* r = (Retangulo*) malloc(sizeof(Retangulo));
    ListaNo* p = (ListaNo*) malloc(sizeof(ListaNo));
    r->b = b;
    r->h = h;
    p->info = r;
    p->prox = NULL;
    return p;
}

```

Dessa forma, o valor da base associado a um nó p seria acessado por: $p \rightarrow \text{info} \rightarrow b$. A vantagem dessa representação (utilizando ponteiros) é que, independente da informação armazenada na lista, a estrutura do nó é sempre composta por um ponteiro para a informação e um ponteiro para o próximo nó da lista. Além disso, podemos ter outras listas também armazenando os mesmos ponteiros de retângulos.

14.5 Listas heterogêneas

A representação da informação por um ponteiro nos permite construir listas heterogêneas, isto é, listas em que as informações armazenadas diferem de nó para nó. Como exemplo, vamos considerar uma aplicação que necessite manipular listas de objetos geométricos planos para cálculo de áreas. Para simplificar, vamos considerar que os objetos podem ser apenas retângulos, triângulos ou círculos. Sabemos que as áreas desses objetos são dadas por:

$$A_{ret} = b \cdot h, \quad A_{tri} = \frac{b \cdot h}{2}, \quad A_{circ} = \pi \cdot r^2$$

Devemos definir um tipo para cada objeto a ser representado:

```

typedef struct retangulo Retangulo;
struct retangulo {
    float b;
    float h;
};

typedef struct triangulo Triangulo;
struct triangulo {
    float b;
    float h;
};

typedef struct circulo Circulo;
struct circulo {
    float r;
};

```

O nó da lista deve então ser composto por três campos:

- um identificador de qual objeto está armazenado no nó
- um ponteiro para a estrutura que contém a informação
- um ponteiro para o próximo nó da lista

É importante salientar que, a rigor, a lista é homogênea, no sentido de que todos os nós contêm os mesmos campos. O ponteiro para a informação deve ser do tipo genérico, pois não sabemos a princípio para que estrutura ele irá apontar: pode apontar para um retângulo, um triângulo ou um círculo. Um ponteiro genérico em C é representado pelo tipo `void*`. Uma variável do tipo ponteiro genérico pode representar qualquer endereço de memória, independente da informação de fato armazenada nesse espaço. No entanto, de posse de um ponteiro genérico, não podemos acessar a memória por ele apontada, já que não sabemos o tipo da informação armazenada. Por esta razão, o nó de uma lista genérica deve guardar explicitamente um identificador do tipo de objeto de fato armazenado. Consultando esse identificador, podemos converter o ponteiro genérico no ponteiro específico para o objeto em questão e, então, acessar os campos do objeto.

Como identificador de tipo, podemos usar uma enumeração:

```
typedef enum tipo Tipo;
enum tipo {
    RET,
    TRI,
    CIRC
};
```

Assim, na criação do nó, armazenamos o identificador de tipo correspondente ao objeto a ser representado. A estrutura que representa o nó pode então ser dada por:

```
/* Define o nó da estrutura */
typedef struct listanohet ListaNoHet;
struct listanohet {
    Tipo tipo;
    void* info;
    ListaNoHet* prox;
};
```

A estrutura que representa a lista pode ser:

```
typedef struct listahet ListaHet;
struct listahet {
    ListaNoHet* prim;
};
```

A função para a criação de um nó da lista pode ser definida por três variações, uma para cada tipo de objeto que pode ser armazenado. Uma vez criados os nós, podemos inserí-los na lista como já vimos fazendo com os nós de listas homogêneas.

```

/* Cria um nó com um retângulo */
ListaNoHet* cria_ret (float b, float h)
{
    /* aloca retângulo */
    Retangulo* r = (Retangulo*) malloc(sizeof(Retangulo));
    r->b = b;
    r->h = h;

    /* aloca nó */
    ListaNoHet* p = (ListaNoHet*) malloc(sizeof(ListaNoHet));
    p->tipo = RET;
    p->info = r;
    p->prox = NULL;
    return p;
}

/* Cria um nó com um triângulo */
ListaNoHet* cria_tri (float b, float h)
{
    /* aloca triângulo */
    Triangulo* t = (Triangulo*) malloc(sizeof(Triangulo));
    t->b = b;
    t->h = h;

    /* aloca nó */
    ListaNoHet* p = (ListaNoHet*) malloc(sizeof(ListaNoHet));
    p->tipo = TRI;
    p->info = t;
    p->prox = NULL;
    return p;
}

/* Cria um nó com um círculo */
ListaNoHet* cria_circ (float r)
{
    /* aloca círculo */
    Circulo* c = (Circulo*) malloc(sizeof(Circulo));
    c->r = r;

    /* aloca nó */
    ListaNoHet* p = (ListaNoHet*) malloc(sizeof(ListaNoHet));
    p->tipo = CIRC;
    p->info = c;
    p->prox = NULL;
    return p;
}

```

Para exemplificar a manipulação de listas heterogêneas, considerando a existência de uma lista com os objetos geométricos apresentados, vamos implementar uma função que fornece como valor de retorno a maior área entre os elementos da lista. Uma implementação dessa função é mostrada a seguir, em que criamos uma função auxiliar que calcula a área do objeto armazenado em determinado nó da lista. Como essa função recebe um

nó da lista heterogênea, ela tem que testar o tipo do objeto armazenado. Uma vez identificado o tipo do objeto, a função converte o ponteiro genérico `info` para um ponteiro do tipo específico do objeto em questão. A partir do ponteiro convertido, pode-se acessar os campos da estrutura que define aquele tipo de objeto.

```
#define PI 3.14159

/* função auxiliar: calcula área correspondente ao nó */
static float area (ListaNoHet* p)
{
    switch (p->tipo) {
        case RET: {
            /* converte para retângulo e calcula área */
            Retangulo *r = (Retangulo*) p->info;
            return r->b * r->h;
        }
        break;
        case TRI: {
            /* converte para triângulo e calcula área */
            Triangulo *t = (Triangulo*) p->info;
            return (t->b * t->h) / 2;
        }
        break;
        case CIRC: {
            /* converte para círculo e calcula área */
            Circulo *c = (Circulo*) p->info;
            return PI * c->r * c->r;
        }
        break;
    }
}
```

Com o auxílio dessa função, podemos escrever o código da função que tem como valor de retorno a maior área dos objetos armazenados na lista. Neste código, o tipo `ListaHet` representa a estrutura da lista heterogênea em si, com um ponteiro para o primeiro nó:

```
/* Função para cálculo da maior área */
float max_area (ListaHet* l)
{
    float amax = 0.0;          /* maior área */
    for (ListaNoHet* p=l->prim; p!=NULL; p=p->prox) {
        float a = area(p);    /* área do nó */
        if (a > amax)
            amax = a;
    }
    return amax;
}
```

Como vemos, a função que acessa a lista não traz nenhuma novidade com relação às funções antes apresentadas para listas homogêneas. Apenas o acesso à informação associada a cada nó é que não pode ser feito de forma direta, pois primeiro precisamos identificar o tipo da informação e então converter o ponteiro genérico para um ponteiro específico.

Para obter um código mais estruturado na manipulação das informações, podemos reescrever a função para o cálculo da área associada a um nó. Vamos agora utilizar mais funções auxiliares, específicas para o cálculo da área de cada objeto geométrico. A função genérica é responsável apenas por chamar a função específica correspondente ao tipo de objeto armazenado no nó:

```
/* função para cálculo da área de um retângulo */
static float ret_area (Retangulo* r)
{
    return r->b * r->h;
}

/* função para cálculo da área de um triângulo */
static float tri_area (Triangulo* t)
{
    return (t->b * t->h) / 2;
}

/* função para cálculo da área de um círculo */
static float cir_area (Circulo* c)
{
    return PI * c->r * c->r;
}

/* função para cálculo da área do nó (versão 2) */
static float area (ListaNoHet* p)
{
    switch (p->tipo) {
        case RET:
            return ret_area(p->info);
            break;
        case TRI:
            return tri_area(p->info);
            break;
        case CIRC:
            return cir_area(p->info);
            break;
    }
}
```

Neste caso, a conversão de ponteiro genérico para ponteiro específico é feita quando chamamos uma das funções de cálculo da área: passa-se um ponteiro genérico que é atribuído, através da conversão implícita de tipo, a um ponteiro específico.¹

Devemos salientar que, quando trabalhamos com conversão de ponteiros genéricos, temos que garantir que o ponteiro armazene o endereço no qual, de fato, existe o tipo específico correspondente. O compilador não tem como verificar se a conversão é válida; a verificação do tipo passa a ser *responsabilidade do programador*.

¹A linguagem C++ não faz essa conversão implícita; teríamos, por exemplo, que codificar:
return ret_area ((Retangulo *)r-> info).

Exercícios

1. Implemente uma função que tenha como valor de retorno o comprimento de uma lista encadeada, isto é, que calcule o número de nós de uma lista. Essa função deve obedecer ao protótipo:

```
int comprimento (Lista* l);
```

2. Considerando listas encadeadas de valores inteiros, implemente uma função que retorne o número de nós da lista que possuem o campo `info` com valores maiores do que `x`. Essa função deve obedecer ao protótipo:

```
int maiores (Lista* l, int x);
```

3. Implemente uma função que retorne o último valor de uma lista encadeada de inteiros. Essa função deve obedecer ao protótipo:

```
int ultimo (Lista* l);
```

4. Implemente uma função que receba duas listas encadeadas de valores reais e transfira para o final da primeira lista os elementos da segunda. No final, a primeira lista representará a concatenação das duas listas e a segunda lista estará vazia. Essa função deve obedecer ao protótipo:

```
void concatena (Lista* l1, Lista* l2);
```

5. Considerando listas de valores inteiros, implemente uma função que receba como parâmetros uma lista encadeada e um valor inteiro `x`, e retire da lista todas as ocorrências de `x`. Essa função deve obedecer ao protótipo:

```
void retira_n (Lista* l, int x);
```

6. Considerando listas de valores inteiros, implemente uma função que receba como parâmetro uma lista encadeada e um valor inteiro `x` e divida a lista em duas, de tal forma que a segunda lista, criada dentro da função, comece no primeiro nó logo após a primeira ocorrência de `x` na lista original. A função deve ter como valor de retorno a lista criada, mesmo que ela seja vazia. A Figura 14.10 ilustra este procedimento.

Essa função deve obedecer ao protótipo:

```
Lista* separa (Lista* l, int x);
```

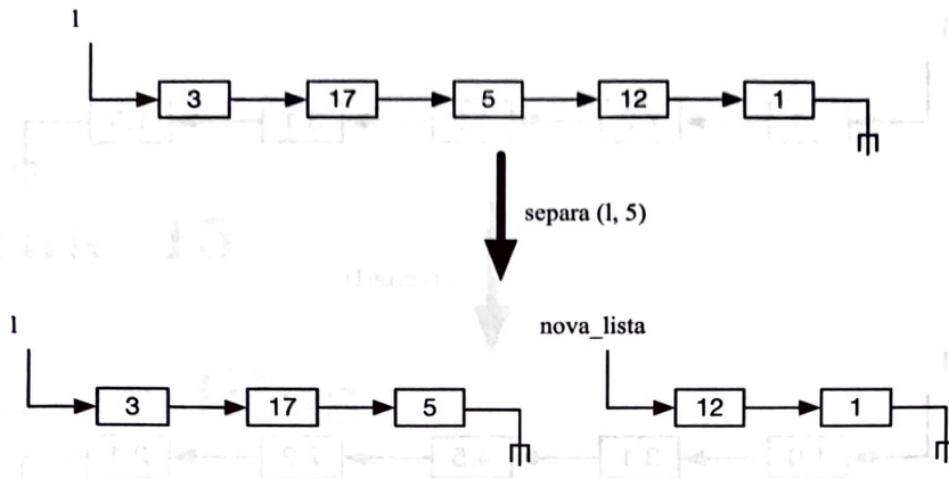


Figura 14.10: Efeito da função que separa duas listas.

7. Implemente uma função que construa uma nova lista, intercalando os nós de outras duas listas passadas como parâmetros. Essa função deve retornar a nova lista resultante, criada dentro da função, conforme ilustrado na Figura 14.11.

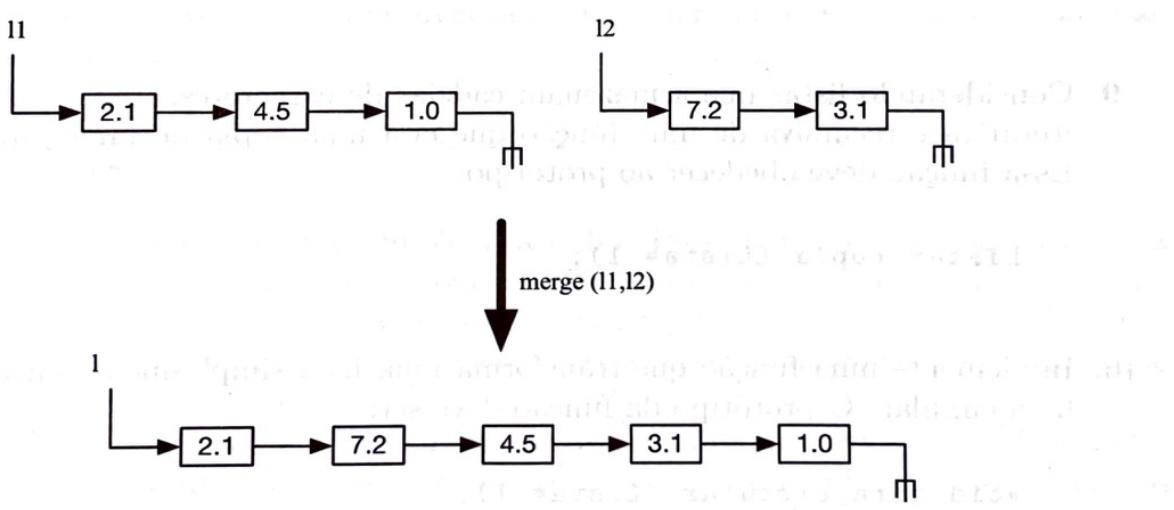


Figura 14.11: Efeito da função que combina duas listas.

Ao final da função, as listas originais devem ficar vazias e ser liberadas. Se uma lista tiver mais elementos que a outra, os elementos excedentes são transferidos na mesma ordem para a nova lista. Essa função deve obedecer ao protótipo:

```
List* merge (List* l1, List* l2);
```

8. Implemente uma função que receba como parâmetro uma lista encadeada e inverta o encadeamento de seus nós. Após a execução dessa função, cada nó da lista vai estar apontando para o nó que originalmente era seu antecessor, e o último nó da lista passará a ser o primeiro nó da lista invertida, conforme ilustrado na Figura 14.12.

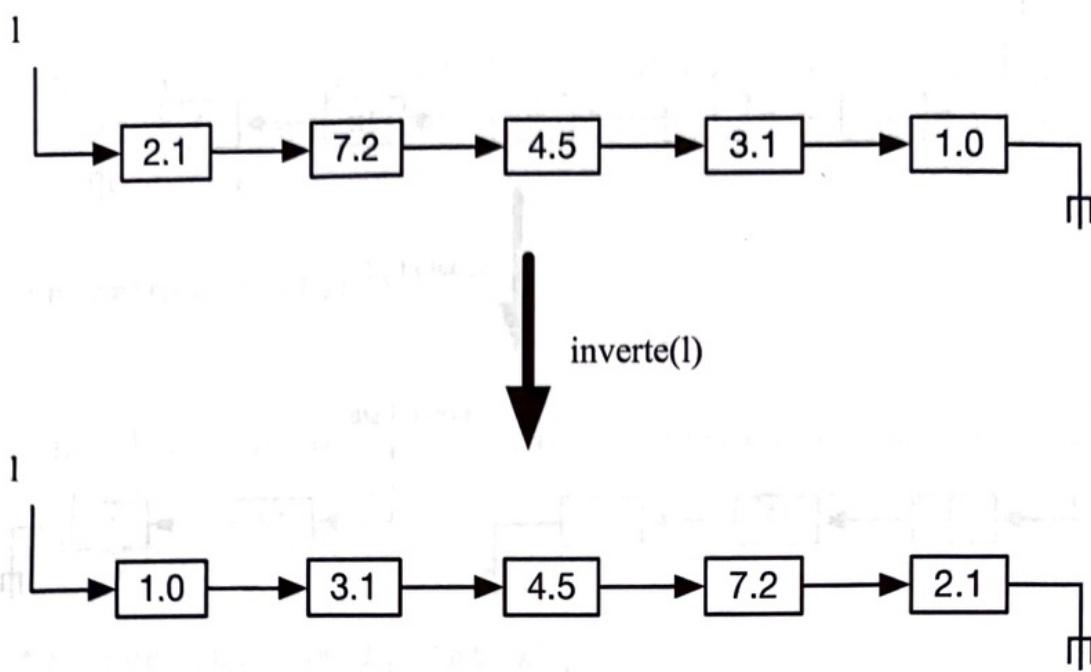


Figura 14.12: Efeito da função que inverte uma lista.

Essa função deve obedecer ao protótipo:

```
void inverte (Lista* l);
```

9. Considerando listas que armazenam cadeias de caracteres, implemente as versões iterativa e recursiva de uma função que cria uma cópia de uma lista encadeada. Essa função deve obedecer ao protótipo:

```
List* copia (List* l);
```

10. Implemente uma função que transforma uma lista simplesmente encadeada numa lista circular. O protótipo da função deve ser:

```
void para_circular (List* l);
```

11. Implemente as funções para retirar elementos do início e do final de uma lista duplamente encadeada. Os protótipos das funções devem ser:

```
void retira_inicio (List2* l);
void retira_final (List2* l);
```

12. Implemente funções para inserir e retirar um elemento de uma lista circular duplamente encadeada.