10. Listas Encadeadas

W. Celes e J. L. Rangel

Para representarmos um grupo de dados, já vimos que podemos usar um vetor em C. O vetor é a forma mais primitiva de representar diversos elementos agrupados. Para simplificar a discussão dos conceitos que serão apresentados agora, vamos supor que temos que desenvolver uma aplicação que deve representar um grupo de valores inteiros. Para tanto, podemos declarar um vetor escolhendo um número máximo de elementos.

```
#define MAX 1000
int vet[MAX];
```

Ao declararmos um vetor, reservamos um espaço contíguo de memória para armazenar seus elementos, conforme ilustra a figura abaixo.

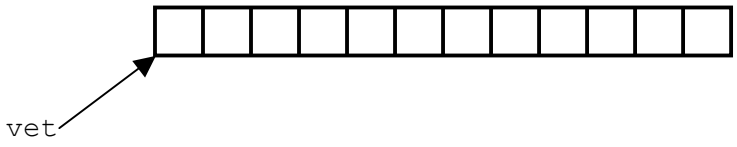

Figura 9.1: Um vetor ocupa um espaço contíguo de memória, permitindo que qualquer elemento seja acessado indexando-se o ponteiro para o primeiro elemento.

O fato de o vetor ocupar um espaço contíguo na memória nos permite acessar qualquer um de seus elementos a partir do ponteiro para o primeiro elemento. De fato, o símbolo `vet`, após a declaração acima, como já vimos, representa um ponteiro para o primeiro elemento do vetor, isto é, o valor de `vet` é o endereço da memória onde o primeiro elemento do vetor está armazenado. De posse do ponteiro para o primeiro elemento, podemos acessar qualquer elemento do vetor através do operador de indexação `vet[i]`. Dizemos que o vetor é uma estrutura que possibilita acesso randômico aos elementos, pois podemos acessar qualquer elemento aleatoriamente.

No entanto, o vetor não é uma estrutura de dados muito flexível, pois precisamos dimensioná-lo com um número máximo de elementos. Se o número de elementos que precisarmos armazenar exceder a dimensão do vetor, teremos um problema, pois não existe uma maneira simples e barata (computacionalmente) para alterarmos a dimensão do vetor em tempo de execução. Por outro lado, se o número de elementos que precisarmos armazenar no vetor for muito inferior à sua dimensão, estaremos sub-utilizando o espaço de memória reservado.

A solução para esses problemas é utilizar estruturas de dados que cresçam à medida que precisarmos armazenar novos elementos (e diminuam à medida que precisarmos retirar elementos armazenados anteriormente). Tais estruturas são chamadas *dinâmicas* e armazenam cada um dos seus elementos usando alocação dinâmica.

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.

10.1. Lista encadeada

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 percorrer todos os elementos da lista, devemos explicitamente guardar o encadeamento dos elementos, o que é feito armazenando-se, junto com a informação de cada elemento, um ponteiro para o próximo elemento da lista. A Figura 9.2 ilustra o arranjo da memória de uma lista encadeada.

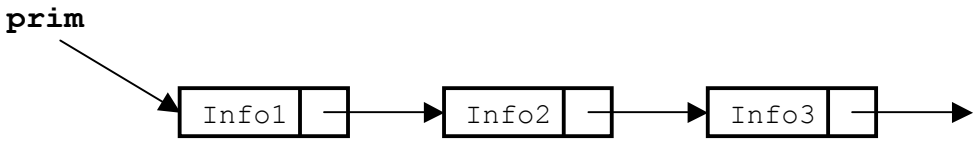

Figura 9.2: Arranjo da memória de uma lista encadeada.

A estrutura consiste numa seqüência encadeada de elementos, em geral chamados de *nós da lista*. A lista é representada por um ponteiro para o primeiro elemento (ou nó). Do primeiro elemento, podemos alcançar o segundo seguindo o encadeamento, e assim por diante. O último elemento da lista aponta para `NULL`, sinalizando que não existe um próximo elemento.

Para exemplificar a implementação de listas encadeadas em C, vamos considerar um exemplo simples em que queremos armazenar valores inteiros numa lista encadeada. O nó da lista pode ser representado pela estrutura abaixo:

```
struct lista {
   int info;
   struct lista* prox;
};

typedef struct lista Lista;
```

Devemos notar que trata-se de uma estrutura auto-referenciada, 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. Embora não seja essencial, é uma boa estratégia definirmos o tipo `Lista` como sinônimo de `struct lista`, conforme ilustrado acima. O tipo `Lista` representa um nó da lista e a estrutura de lista encadeada é representada pelo ponteiro para seu primeiro elemento (tipo `Lista*`).

Considerando a definição de `Lista`, podemos definir as principais funções necessárias para implementarmos uma lista encadeada.

Função de inicialização

A função que inicializa uma lista deve criar uma lista vazia, sem nenhum elemento. Como a lista é representada pelo ponteiro para o primeiro elemento, uma lista vazia é

representada pelo ponteiro NULL, pois não existem elementos na lista. A função tem como valor de retorno a lista vazia inicializada, isto é, o valor de retorno é NULL. Uma possível implementação da função de inicialização é mostrada a seguir:

```
/* função de inicialização: retorna uma lista vazia */
Lista* inicializa (void)
{
   return NULL;
}
```

Função de inserção

Uma vez criada a lista vazia, podemos inserir novos elementos nela. Para cada elemento inserido na lista, devemos alocar dinamicamente a memória necessária para armazenar o elemento 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 ter seu valor atualizado, pois a lista deve passar a ser representada pelo ponteiro para o novo primeiro elemento. Por esta razão, a função de inserção recebe como parâmetros de entrada a lista onde será inserido o novo elemento e a informação do novo elemento, e tem como valor de retorno a nova lista, representada pelo ponteiro para o novo elemento.

```
/* inserção no início: retorna a lista atualizada */
Lista* insere (Lista* l, int i)
{
   Lista* novo = (Lista*) malloc(sizeof(Lista));
   novo->info = i;
   novo->prox = l;
   return 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 elemento) o elemento que era o primeiro da lista. A função então retorna o novo valor que representa a lista, que é o ponteiro para o novo primeiro elemento. A Figura 9.3 ilustra a operação de inserção de um novo elemento no início da lista.

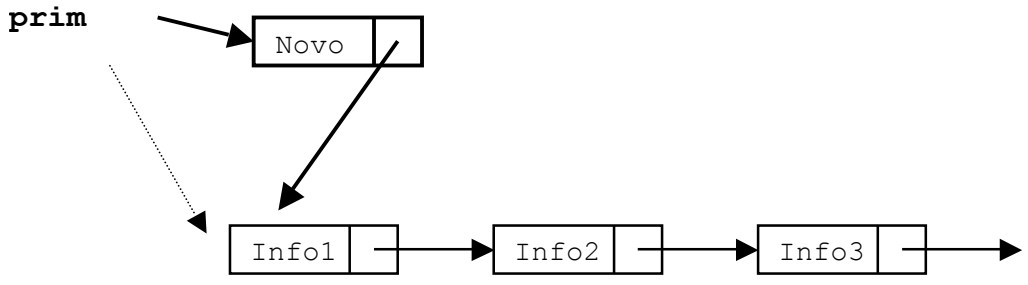

Figura 9. 3: Inserção de um novo elemento no início da lista.

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

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

Observe que não podemos deixar de atualizar a variável que representa a lista a cada inserção de um novo elemento.

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 imprima os valores dos elementos armazenados numa lista. Uma possível implementação dessa função é mostrada a seguir.

```
/* função imprime: imprime valores dos elementos */
void imprime (Lista* l)
{
   Lista* p;   /* variável auxiliar para percorrer a lista */
   for (p = l; p != NULL; p = p->prox)
      printf("info = %d\n", p->info);
}
```

Função que verifica se lista está vazia

Pode ser útil implementarmos uma função que verifique 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 valor é `NULL`. Uma implementação dessa função é mostrada a seguir:

```
/* função vazia: retorna 1 se vazia ou 0 se não vazia */
int vazia (Lista* l)
{
   if (l == NULL)
      return 1;
   else
      return 0;
}
```

Essa função pode ser re-escrita de forma mais compacta, conforme mostrado abaixo:

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

Função de busca

Outra função útil consiste em verificar se um determinado elemento está presente na lista. A função recebe a informação referente ao elemento que queremos buscar e fornece 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 */
Lista* busca (Lista* l, int v)
{
   Lista* p;
   for (p=l; p!=NULL; p=p->prox)
      if (p->info == v)
         return p;
   return NULL;       /* não achou o elemento */
}
```

Função que retira um elemento da lista

Para completar o conjunto de funções que manipulam uma lista, devemos implementar uma função que nos permita retirar um elemento. A função tem como parâmetros de entrada a lista e o valor do elemento que desejamos retirar, e deve retornar o valor atualizado da lista, pois, se o elemento removido for o primeiro da lista, o valor da lista deve ser atualizado.

A função para retirar um elemento da lista é mais complexa. Se descobrirmos que o elemento a ser retirado é o primeiro da lista, devemos fazer com que o novo valor 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. Devemos notar que, no segundo caso, precisamos do ponteiro para o elemento anterior para podermos acertar o encadeamento da lista. As Figuras 9.4 e 9.5 ilustram as operações de remoção.

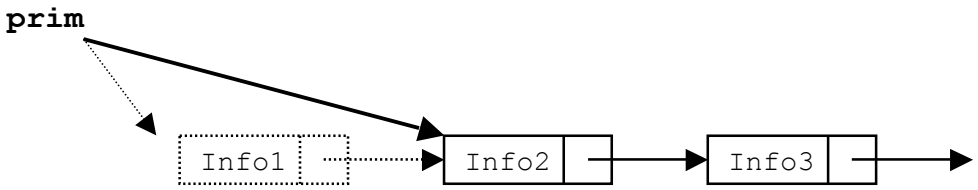

Figura 9.4: Remoção do primeiro elemento da lista.

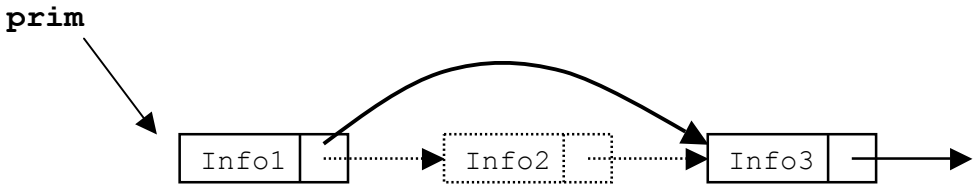

Figura 9.5: Remoção de um elemento no meio da lista.

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 */
Lista* retira (Lista* l, int v) {
   Lista* ant = NULL; /* ponteiro para elemento anterior */
   Lista* p = l;      /* ponteiro para percorrer a lista*/

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

   /* verifica se achou elemento */
   if (p == NULL)
      return l;   /* não achou: retorna lista original */

   /* retira elemento */
   if (ant == NULL) {
      /* retira elemento do inicio */
      l = p->prox;
   }
   else {
      /* retira elemento do meio da lista */
      ant->prox = p->prox;
   }
   free(p);
   return l;
}
```

O caso de retirar o último elemento da lista recai no caso de retirar um elemento no meio da lista, conforme pode ser observado na implementação acima. Mais adiante, estudaremos a implementação de filas com listas encadeadas. Numa fila, devemos armazenar, além do ponteiro para o primeiro elemento, um ponteiro para o último elemento. Nesse caso, se for removido o último elemento, veremos que será necessário atualizar a fila.

Função para liberar a lista

Uma outra função útil que devemos considerar destrói a lista, liberando todos os elementos alocados. Uma implementação dessa função é mostrada abaixo. A função percorre elemento a 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, estaríamos acessando um espaço de memória que não estaria mais reservado para nosso uso).

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

Um programa que ilustra a utilização dessas funções é mostrado a seguir.

```
#include <stdio.h>

int main (void) {
   Lista* l;          /* declara uma lista não iniciada */
   l = inicializa();  /* inicia lista vazia */
   l = insere(l, 23); /* insere na lista o elemento 23 */
   l = insere(l, 45); /* insere na lista o elemento 45 */
   l = insere(l, 56); /* insere na lista o elemento 56 */
   l = insere(l, 78); /* insere na lista o elemento 78 */
   imprime(l);        /* imprimirá: 78 56 45 23 */
   l = retira(l, 78);
   imprime(l);        /* imprimirá: 56 45 23 */
   l = retira(l, 45);
   imprime(l);        /* imprimirá: 56 23 */
   libera(l);
   return 0;
}
```

Mais uma vez, observe que não podemos deixar de atualizar a variável que representa a lista a cada inserção e a cada remoção de um elemento. Esquecer de atribuir o valor de retorno à variável que representa a lista pode gerar erros graves. Se, por exemplo, a função retirar o primeiro elemento da lista, a variável que representa a lista, se não fosse atualizada, estaria apontando para um nó já liberado. Como alternativa, poderíamos fazer com que as funções `insere` e `retira` recebessem o endereço da variável que representa a lista. Nesse caso, os parâmetros das funções seriam do tipo ponteiro para lista (`Lista** l`) e seu conteúdo poderia ser acessado/atualizado de dentro da função usando o operador conteúdo (`*l`).

Manutenção da lista ordenada

A função de inserção vista acima 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 acharmos 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 disso 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. Com isto, temos que saber inserir um elemento no meio da lista. A Figura 9.6 ilustra a inserção de um elemento no meio da lista.

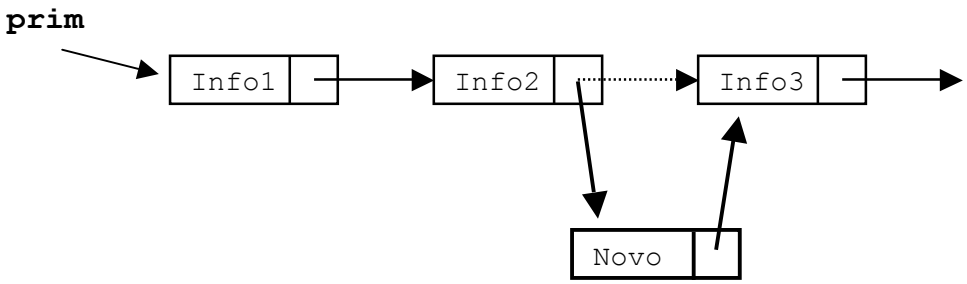

Figura 9.6: Inserção de um elemento no meio da lista.

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, podemos encadear o novo elemento na lista. O novo apontará para o próximo elemento na lista e o elemento precedente apontará para o novo. O código abaixo ilustra a implementação dessa função. Neste caso, utilizamos uma função auxiliar responsável por alocar memória para o novo nó e atribuir o campo da informação.

```
/* função auxiliar: cria e inicializa um nó */
Lista* cria (int v)
{
   Lista* p = (Lista*) malloc(sizeof(Lista));
   p->info = v;
   return p;
}

/* função insere_ordenado: insere elemento em ordem */
Lista* insere_ordenado (Lista* l, int v)
{
   Lista* novo = cria(v); /* cria novo nó */
   Lista* ant = NULL;     /* ponteiro para elemento anterior */
   Lista* p = l;          /* ponteiro para percorrer a lista*/

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

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

Devemos notar que essa função, analogamente ao observado para a função de remoção, também funciona se o elemento tiver que ser inserido no final da lista.

10.2. Implementações recursivas

Uma lista pode ser definida de maneira recursiva. Podemos dizer que uma lista encadeada é representada por:

- uma lista vazia; ou
- um elemento seguido de uma (sub-)lista.

Neste caso, o segundo elemento da lista representa o primeiro elemento da sub-lista. Com base na definição recursiva, podemos implementar as funções de lista recursivamente. Por exemplo, a função para imprimir os elementos da lista pode ser re-escrita da forma ilustrada abaixo:

```
/* Função imprime recursiva */
void imprime_rec (Lista* l)
{
   if (vazia(l))
      return;
   /* imprime primeiro elemento */
   printf("info: %d\n",l->info);
   /* imprime sub-lista */
   imprime_rec(l->prox);
}
```

É fácil alterarmos o código acima para obtermos a impressão dos elementos da lista em ordem inversa: basta invertermos a ordem das chamadas às funções `printf` e `imprime_rec`.

A função para retirar um elemento da lista também pode ser escrita de forma recursiva. Neste caso, só retiramos um elemento se ele for o primeiro da lista (ou da sub-lista). Se o elemento que queremos retirar não for o primeiro, chamamos a função recursivamente para retirar o elemento da sub-lista.

```
/* Função retira recursiva */
Lista* retira_rec (Lista* l, int v)
{
   if (vazia(l))
      return l;   /* lista vazia: retorna valor original */

   /* verifica se elemento a ser retirado é o primeiro */
   if (l->info == v) {
      Lista* t = l;     /* temporário para poder liberar */
      l = l->prox;
      free(t);
   }
   else {
      /* retira de sub-lista */
      l->prox = retira_rec(l->prox,v);
   }
   return l;
}
```

A função para liberar uma lista também pode ser escrita recursivamente, de forma bastante simples. Nessa função, se a lista não for vazia, liberamos primeiro a sub-lista e depois liberamos a lista.

```
void libera_rec (Lista* l)
{
   if (!vazia(l))
   {
      libera_rec(l->prox);
      free(l);
   }
}
```

Exercício: Implemente uma função que verifique se duas listas encadeadas são iguais. Duas listas são consideradas iguais se têm a mesma seqüência de elementos. O protótipo da função deve ser dado por:

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

Exercício: Implemente uma função que crie uma cópia de uma lista encadeada. O protótipo da função deve ser dado por:

```
Lista* copia (Lista* l);
```

10.3. Listas genéricas

Um nó de uma lista encadeada contém basicamente duas informações: o encadeamento e a informação armazenada. Assim, a estrutura de um nó para representar uma lista de números inteiros é dada por:

```
struct lista {
   int info;
   struct lista *prox;
};
typedef struct lista Lista;
```

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

```
struct lista {
   float info;
   struct lista *prox;
};
typedef struct lista Lista;
```

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. Cada retângulo é definido pela base `b` e pela altura `h`. Assim, a estrutura do nó pode ser dada por:

```
struct lista {
   float b;
   float h;
   struct lista *prox;
};
typedef struct lista Lista;
```

Esta mesma composição 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.

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

```
struct lista {
   Retangulo info;
   struct lista *prox;
};
typedef struct lista Lista;
```

Aqui, a informação volta a ser representada por um único campo (`info`), que é uma estrutura. Se `p` fosse um ponteiro para um nó da lista, o valor da base do retângulo armazenado nesse nó seria acessado por: `p->info.b`.

Ainda mais interessante é termos o campo da informação representado por um ponteiro para a estrutura, em vez da estrutura em si.

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

struct lista {
   Retangulo *info;
   struct lista *prox;
};
typedef struct lista Lista;
```

Neste caso, para criarmos um nó, temos que fazer duas alocações dinâmicas: uma para criar a estrutura do retângulo e outra para criar a estrutura do nó. O código abaixo ilustra uma função para a criação de um nó.

```
Lista* cria (void)
{
   Retangulo* r = (Retangulo*) malloc(sizeof(Retangulo));
   Lista* p = (Lista*) malloc(sizeof(Lista));
   p->info = r;
   p->prox = NULL;
   return p;
}
```

Naturalmente, o valor da base associado a um nó `p` seria agora acessado por: `p->info->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.

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. Diversas aplicações precisam construir listas heterogêneas, pois necessitam agrupar elementos afins mas não necessariamente iguais. Como exemplo, vamos considerar uma aplicação que necessite manipular listas de objetos geométricos planos para cálculos 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:

$$r = b * h \qquad t = \frac{b * h}{2} \qquad c = \pi r^2$$

Devemos definir um tipo para cada objeto a ser representado:

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

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

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

O nó da lista deve 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 as mesmas informações. 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*`. A função 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 a 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, acessarmos os campos do objeto.

Como identificador de tipo, podemos usar valores inteiros definidos como constantes simbólicas:

```
#define RET 0
#define TRI 1
#define CIR 2
```

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

```
/* Define o nó da estrutura */
struct listagen {
   int   tipo;
   void *info;
   struct listagen *prox;
};
typedef struct listagen ListaGen;
```

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.

```
/* Cria um nó com um retângulo, inicializando os campos base e altura
*/
ListaGen* cria_ret (float b, float h)
{
   Retangulo* r;
   ListaGen* p;

   /* aloca retângulo */
   r = (Retangulo*) malloc(sizeof(Retangulo));
   r->b = b;
   r->h = h;

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

   return p;
}

/* Cria um nó com um triângulo, inicializando os campos base e altura
*/
ListaGen* cria_tri (float b, float h)
{
   Triangulo* t;
   ListaGen* p;

   /* aloca triângulo */
   t = (Triangulo*) malloc(sizeof(Triangulo));
   t->b = b;
   t->h = h;

   /* aloca nó */
   p = (ListaGen*) malloc(sizeof(ListaGen));
   p->tipo = TRI;
   p->info = t;

   p->prox = NULL;
   return p;
}


/* Cria um nó com um círculo, inicializando o campo raio */
ListaGen* cria_cir (float r)
{
   Circulo* c;
   ListaGen* p;

   /* aloca círculo */
   c = (Circulo*) malloc(sizeof(Circulo));
   c->r = r;

   /* aloca nó */
   p = (ListaGen*) malloc(sizeof(ListaGen));
   p->tipo = CIR;
   p->info = c;
   p->prox = NULL;

   return p;
}
```

Uma vez criado o nó, podemos inseri-lo na lista como já vínhamos fazendo com nós de listas homogêneas. As constantes simbólicas que representam os tipos dos objetos podem ser agrupadas numa enumeração (ver seção 7.5):

```
enum {
   RET,
   TRI,
   CIR
};
```

Manipulação de listas heterogêneas

Para exemplificar a manipulação de listas heterogêneas, considerando a existência de uma lista com os objetos geométricos apresentados acima, vamos implementar uma função que forneça como valor de retorno a maior área entre os elementos da lista. Uma implementação dessa função é mostrada abaixo, onde criamos uma função auxiliar que calcula a área do objeto armazenado num determinado nó da lista:

```
#define PI 3.14159


/* função auxiliar: calcula área correspondente ao nó */
float area (ListaGen *p)
{
   float a;          /* área do elemento */

   switch (p->tipo) {

     case RET:
     {
        /* converte para retângulo e calcula área */
        Retangulo *r = (Retangulo*) p->info;
        a = r->b * r->h;
     }
     break;

     case TRI:
     {
        /* converte para triângulo e calcula área */
        Triangulo *t = (Triangulo*) p->info;
        a = (t->b * t->h) / 2;
     }
     break;

     case CIR:
     {
        /* converte para círculo e calcula área */
        Circulo *c = (Circulo)p->info;
        a = PI * c->r * c->r;
     }
     break;
   }
   return a;
}

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

A função para o cálculo da área mostrada acima pode ser subdivida em funções específicas para o cálculo das áreas de cada objeto geométrico, resultando em um código mais estruturado.

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

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

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

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

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, para um ponteiro específico[1].

Devemos salientar que, quando trabalhamos com conversão de ponteiros genéricos, temos que garantir que o ponteiro armazene o endereço onde de fato existe o tipo específico correspondente. O compilador não tem como checar se a conversão é válida; a verificação do tipo passa a ser responsabilidade do programador.

10.4. Listas circulares

Algumas aplicações necessitam representar conjuntos cíclicos. Por exemplo, as arestas que delimitam uma face 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 falarmos em primeiro ou último

[1] Este código não é válido em C++. A linguagem C++ não tem conversão implícita de um ponteiro genérico para um ponteiro específico. Para compilar em C++, devemos fazer a conversão explicitamente. Por exemplo:

```
a = ret_area((Retangulo*)p->info);
```

elemento. A lista pode ser representada por um ponteiro para um elemento inicial qualquer da lista. A Figura 9.7 ilustra o arranjo da memória para a representação de uma lista circular.

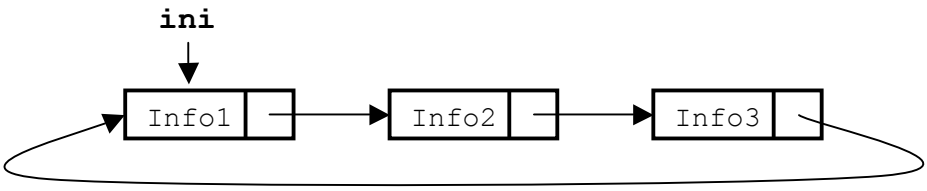

Figura 9.7: Arranjo da memória de uma lista circular.

Para percorrer os elementos de uma lista circular, visitamos todos os elementos a partir do ponteiro do elemento inicial até alcançarmos novamente esse mesmo elemento. O código abaixo exemplifica essa forma de percorrer os elementos. Neste caso, 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 imprime_circular (Lista* l)
{
   Lista* p = l;        /* faz p apontar para o nó inicial */
   /* testa se lista não é vazia */
   if (p) {
   {
      /* percorre os elementos até alcançar novamente o início */
      do {
         printf("%d\n", p->info);   /* imprime informação do nó */
         p = p->prox;               /* avança para o próximo nó */
      } while (p != l);
}
```

Exercício: Escreva as funções para inserir e retirar um elemento de uma lista circular.

*10.5. 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 encontrarmos o elemento anterior, pois, dado um 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 9.8 esquematiza a estruturação de uma lista duplamente encadeada.

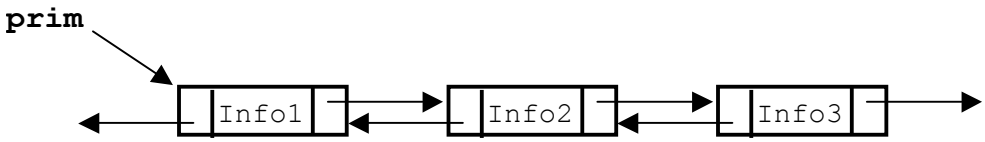

Figura 9.8: Arranjo da memória de uma lista duplamente encadeada.

Para exemplificar a implementação de listas duplamente encadeadas, vamos novamente considerar o exemplo simples no qual queremos armazenar valores inteiros na lista. O nó da lista pode ser representado pela estrutura abaixo e a lista pode ser representada através do ponteiro para o primeiro nó.

```
struct lista2 {
   int info;
   struct lista2* ant;
   struct lista2* prox;
};

typedef struct Lista2 Lista2;
```

Com base nas definições acima, exemplificamos a seguir a implementação de algumas funções que manipulam listas duplamente encadeadas.

Função de inserção

O código a seguir mostra uma possível implementação da função que insere novos elementos no início da lista. Após a alocação do novo elemento, a função acertar o duplo encadeamento.

```
/* inserção no início */
Lista2* insere (Lista2* l, int v)
{
   Lista2* novo = (Lista2*) malloc(sizeof(Lista2));
   novo->info = v;
   novo->prox = l;
   novo->ant = NULL;
   /* verifica se lista não está vazia */
   if (l != NULL)
      l->ant = novo;
   return novo;
}
```

Nessa função, o novo elemento é encadeado no início da lista. Assim, ele 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, pois, neste caso, o elemento anterior do então primeiro elemento passa a ser o novo elemento. De qualquer forma, o novo elemento passa a ser o primeiro da lista, e deve ser retornado como valor da lista atualizada. A Figura 9.9 ilustra a operação de inserção de um novo elemento no início da lista.

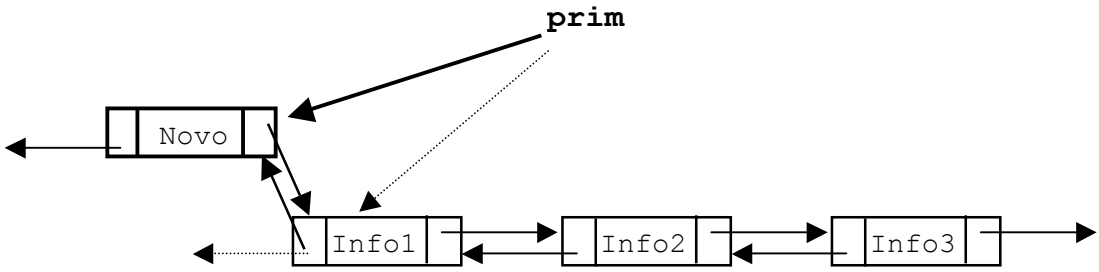

Figura 9.9: Inserção de um novo elemento no início da lista.

Função de busca

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 */
Lista2* busca (Lista2* l, int v)
{
   Lista2* p;
   for (p=l; p!=NULL; p=p->prox)
      if (p->info == v)
         return p;
   return NULL;       /* não achou o elemento */
}
```

Função que retira um elemento da lista

A função de remoção fica mais complicada, pois temos que acertar o 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 acima 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 acima 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 primeiro, não podemos escrever `p->ant->prox`, pois `p->ant` é `NULL`; além disso, temos que atualizar o valor da lista, pois o primeiro elemento será removido.

Uma implementação da função para retirar um elemento é mostrada a seguir:

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

   if (p == NULL)
      return l;   /* não achou o elemento: retorna lista inalterada */

   /* retira elemento do encadeamento */
   if (l == p)
      l = p->prox;
   else
      p->ant->prox = p->prox;

   if (p->prox != NULL)
      p->prox->ant = p->ant;

   free(p);

   return l;
}
```

Lista circular duplamente encadeada

Uma lista circular também pode ser construída com encadeamento duplo. Neste caso, o que seria o último elemento da lista passa 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 de um ponteiro para um elemento qualquer. Abaixo, ilustramos o código para imprimir a lista no sentido reverso, isto é, percorrendo o encadeamento dos elementos anteriores.

```
void imprime_circular_rev (Lista2* l)
{
   Lista2* p = l;        /* faz p apontar para o nó inicial */
   /* testa se lista não é vazia */
   if (p) {
   {
      /* percorre os elementos até alcançar novamente o início */
      do {
         printf("%d\n", p->info);   /* imprime informação do nó */
         p = p->ant;                /* "avança" para o nó anterior */
      } while (p != l);
}
```

Exercício: Escreva as funções para inserir e retirar um elemento de uma lista circular duplamente encadeada.