Listas Encadeadas

Após estudar este capítulo, você deverá ser capaz de:

- Descrever os seguinte conceitos:
  - ❒ Nó de lista encadeada
  - ❒ Visitação de nó
  - ❒ Lista encadeada linear
  - ❒ Lista encadeada circular
  - ❒ Lista encadeada com cabeça
  - ❒ Lista simplesmente encadeada
  - ❒ Lista duplamente encadeada
- Apresentar algumas situações nas quais arrays dinâmicos são satisfatórios e outras nas quais isso não ocorre
- Explicar por que listas encadeadas são consideradas variáveis dinâmicas
- Discorrer sobre vantagens do uso de listas encadeadas em relação a listas indexadas
- Apresentar exemplos de aplicação para os quais o uso de listas encadeadas é mais adequado do que o uso de listas indexadas
- Justificar por que cada nó de uma lista encadeada armazena o endereço de seu sucessor
- Explicar como é efetuado o acesso sequencial a nós de uma lista encadeada
- Descrever as operações sobre listas encadeadas com e sem ordenação
- Implementar qualquer variedade de lista encadeada
- Explicar como são efetuadas operações de inserção e remoção em listas encadeadas
- Implementar pilhas e filas usando listas encadeadas
- Avaliar os custos temporal e espacial de operações sobre listas encadeadas
- Implementar operações sobre listas encadeadas de modo recursivo
- Representar polinômios usando listas encadeadas

objetivos

TODAS AS ESTRUTURAS DE DADOS apresentadas até aqui foram implementadas por meio de arrays, que são variáveis que permitem acesso direto. O presente capítulo introduz o conceito de estruturas de dados implementadas de forma encadeada, que não permitem acesso direto. Este capítulo começa apresentando as deficiências apresentadas pelos arrays dinâmicos e a necessidades de listas encadeadas e prossegue mostrando como as diversas modalidades de listas encadeadas podem ser implementadas em C.

10.1 Deficiências de Arrays Dinâmicos

As soluções apresentadas no **Capítulo 9** para problemas decorrentes de alocação estática de memória ainda não são ideais, apesar de já constituírem boas soluções para problemas de subdimensionamento ou superdimensionamento de arrays. Isto é, as referidas soluções podem demandar redimensionamento de um array quando é feita uma inserção ou remoção de elemento, o que não é ideal, visto que uma chamada de **realloc()** pode requerer que o bloco que será redimensionado seja realocado para uma nova posição no heap (v. **Seção 9.2**). Esse ônus, inerente a chamadas de **realloc()**, pode acarretar dois prejuízos para um programa:

[1] A realocação seguida de cópia do conteúdo de um bloco previamente alocado para um novo bloco demanda certo tempo de processamento. Quanto maior for o bloco a ser copiado, maior será o tempo despendido durante esse processo. Assim, no pior caso, o custo temporal dessa operação é $\theta(n)$.

[2] Para que os dados possam ser copiados de um bloco para o outro, ambos devem permanecer alocados até que a cópia esteja finalizada. Portanto, durante uma realocação, a quantidade de memória requerida nesse processo poderá ser mais do que o dobro do tamanho do bloco que está sendo redimensionado. Assim, no pior caso, o custo espacial dessa operação é $\theta(n)$.

A *solução* para essa possível sobrecarga associada ao uso de **realloc()** talvez fosse alocar individualmente elementos do array por meio de chamadas de **malloc()** ou **calloc()**, mas essa solução não funciona. Ou seja, como todos os elementos de um array alocado dinamicamente, como aqueles descritos nas **Seções 9.6.1** e **9.6.2**, são alocados como um único bloco, é garantido que eles estarão em posições contíguas de memória. Consequentemente, esses elementos podem ser acessados por meio de índices do mesmo modo que ocorre com um array alocado estaticamente. Por outro lado, se esses elementos fossem alocados individualmente por meio de chamadas sucessivas de **malloc()** ou qualquer outra função de alocação dinâmica de memória, a contiguidade deles em memória não estaria garantida. Por exemplo, suponha que `p1`, `p2` e `p3` sejam ponteiros para o tipo **int**. Então, após a execução das chamadas:

```
p1 = malloc(sizeof(int));
p2 = malloc(sizeof(int));
p3 = malloc(sizeof(int));
```

não haveria garantia de que os blocos apontados por `p1`, `p2` e `p3` fossem contíguos em memória. Ou seja, o bloco apontado por `p2` pode não começar logo após o final do bloco apontado por `p1` e o bloco apontado por `p3` pode não começar logo após o final do bloco apontado por `p2`. Pode ser que, nesse caso, a contiguidade em memória eventualmente aconteça, mas o programador não deve jamais contar com tal casualidade.

A ausência de contiguidade impede que elementos que fazem parte de uma coleção de dados alocados individualmente possam ser acessados por meio de índices (como ocorre com arrays). Uma solução para esse problema é fazer com que cada elemento dessa coleção indique, por meio de um ponteiro, onde se encontra o próximo elemento da mesma coleção. Assim, tendo-se um ponteiro para o primeiro elemento da coleção, pode-se acessar o segundo elemento porque o primeiro elemento contém o endereço do segundo; o terceiro elemento também pode ser acessado porque o segundo elemento contém o endereço do terceiro elemento e assim por diante. Essa ideia dá origem ao conceito de lista encadeada, que será o tema central desta seção.

10.2 Lista Simplesmente Encadeada sem Ordenação

10.2.1 Abstração

Uma **lista encadeada** é uma lista que não é indexada, de modo que o acesso a seus elementos não se dá por meio de índices. Uma lista encadeada é composta de elementos, usualmente denominados **nós**, sendo que cada um deles é conceitualmente dividido em duas partes:

[1] O **conteúdo efetivo do nó** — essa parte será doravante identificada pela denominação *conteúdo*.

[2] A localização do **próximo nó** da lista, que será doravante identificada por *próximo*.

O diagrama que compõe a **Figura 10–1** apresenta esquematicamente uma lista simplesmente encadeada com quatro nós cujos conteúdos efetivos são inteiros. A seta mais à esquerda acompanhada da palavra *lista* indica qual é o primeiro nó da lista, enquanto a barra invertida na parte próximo de um nó indica que ele é o último nó da lista.

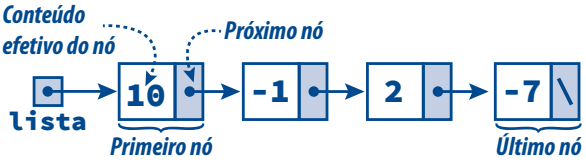


Figura 10–1: Diagrama de Lista Simplesmente Encadeada com Quatro Nós

A lista esquematizada na **Figura 10–1** é denominada simplesmente encadeada porque os ponteiros apontam numa única direção. Existem listas, denominadas duplamente encadeadas, nas quais cada nó possui ponteiros que apontam para o nó anterior bem como para o próximo nó.

Como listas encadeadas não são indexadas, as operações definidas na **Seção 7.1** que fazem uso de índices precisam ser redefinidas. Assim as operações sobre listas simplesmente encadeadas sem ordenação são as seguintes:

[1] **Iniciação** (ou **criação**) de uma lista.

[2] Cálculo do **comprimento da lista**.

[3] **Inserção de um elemento** na lista.

[4] **Remoção de um elemento** que apresenta um determinado valor *a*.

[5] **Checagem de lista vazia**.

[6] **Alteração de valor de um elemento**.

[7] **Busca de um elemento** que apresenta um determinado valor *a*.

Agora, esse rol de operações está incompleto porque ele não permite a um cliente implementar uma operação de acesso sequencial, como aquela levada a efeito pela função `ExibeLista()` apresentada na **Seção 7.6.2**. Portanto é necessário acrescentar uma operação que permita tal acesso. Essa operação é definida assim:

[8] **Obtenção do próximo elemento da lista**, de modo que, na primeira vez em que essa operação for executada, ela resultará o primeiro elemento da lista, na chamada seguinte, ela resultará no segundo elemento da lista e assim por diante até que o último elemento tenha sido obtido. Essa operação é cíclica, de modo que, depois que ela resultar no último elemento da lista, ela recomeça a partir do início da lista.

10.2.2 Implementação

Definições de Tipos

Os nós de uma lista encadeada podem ser implementados em C como estruturas contendo dois campos que correspondem às partes *conteúdo* e *próximo* dos nós. Assim o tipo dos nós de uma lista encadeada pode ser definido como:

```
typedef struct rotNoLSE {
           tConteudo        conteudo;
           struct rotNoLSE *proximo;
        } tNoListaSE;
```

Nessa definição de tipo, `tConteudo` é o tipo do conteúdo efetivo de cada nó. Essa parte de um nó recebe a denominação conteúdo efetivo porque ela representa os dados que são realmente processados por um programa. A segunda parte de cada nó é apenas um acessório com o qual se pode localizar o próximo nó que compõe uma lista encadeada. Na definição de tipo acima, foi utilizado um rótulo de estrutura, de modo a tornar possível a autorreferência no segundo campo da estrutura (v. **Seção 3.10**).

Na **Figura 10–2**, a parte `dados` de cada nó é representada por uma estrutura do tipo `tAluno` definido na **Seção 7.6.2**, enquanto a parte `proximo` é representada por setas que indicam a posição do próximo nó da lista. A parte `proximo` do último nó assume o valor **NULL**, indicando que esse nó não aponta para nenhum outro nó.

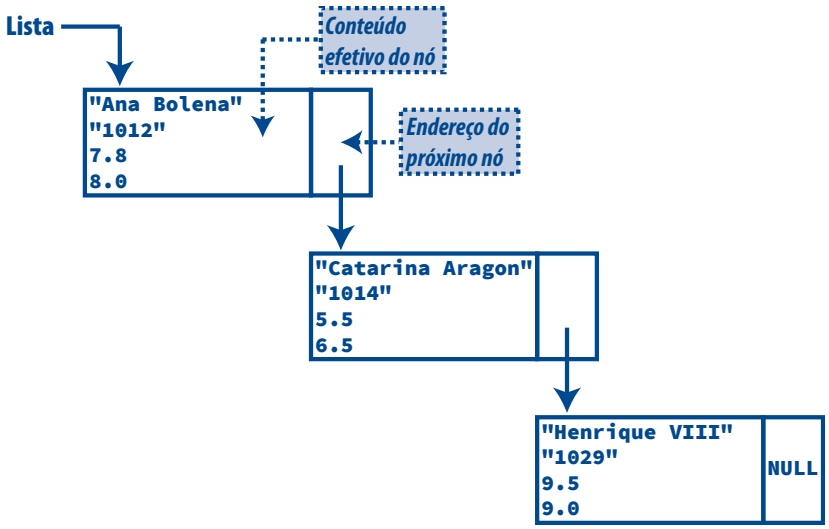


FIGURA 10–2: LISTA SIMPLESMENTE ENCADEADA COM TRÊS NÓS

Do mesmo modo que um array é representado pelo endereço de seu primeiro elemento (v. **Seção 3.3**), uma lista encadeada é representada por um ponteiro que aponta para o primeiro nó da lista. Portanto o tipo de uma variável que representa uma lista encadeada é ponteiro para o tipo `tNoListaSE` definido acima. Ou seja, o tipo `tListaSE` dessa variável pode ser definido como:

```
typedef tNoListaSE *tListaSE;
```

Entretanto, é mais comum definirem-se os tipos `tNoListaSE` e `tListaSE` simultaneamente numa única definição de tipos, que é equivalente às duas definições de tipos precedentes:

```
typedef struct rotNoLSE {
           tConteudo        conteudo;
           struct rotNoLSE *proximo;
        } tNoListaSE, *tListaSE;
```

Não há restrição sobre o tipo `tConteudo`, de modo que ele pode ser qualquer tipo primitivo ou derivado. Por razões didáticas, nos exemplos que serão explorados adiante, o tipo do conteúdo efetivo de cada nó será **int**. Assim a seguinte definição de tipo deve preceder a última definição de tipo apresentada acima.

```
   /* Tipo do conteúdo de cada nó de uma lista encadeada */
typedef int tConteudo;
```

A **Seção 10.7** apresentará exemplos de listas com outros tipos de conteúdos efetivos.

Iniciação

É de importância crucial que toda lista encadeada (ou, mais precisamente, todo ponteiro que representa uma lista encadeada) seja iniciada com **NULL**. Caso contrário, as funções que implementam operações sobre listas encadeadas não têm como identificar quando uma lista é vazia. Portanto nunca esqueça a seguinte recomendação:

A função `IniciaListaSE()` inicia o ponteiro para o primeiro nó de uma lista simplesmente encadeada com **NULL**.

```
void IniciaListaSE(tListaSE *lista)
{
   *lista = NULL;
}
```

Note que o parâmetro da função `IniciaListaSE()` é um ponteiro para ponteiro, visto que o tipo `tListaSE` é um tipo de ponteiro.

Checagem de Lista Vazia

A função `EstaVaziaListaSE()` verifica se uma lista simplesmente encadeada do tipo `tListaSE` está vazia. Essa função retorna `1`, quando a lista está vazia, ou `0` em caso contrário.

```
int EstaVaziaListaSE(tListaSE lista)
{
   return lista == NULL;
}
```

Comprimento

Para calcular o comprimento (i.e., o número de nós) de uma lista simplesmente encadeada, usa-se uma variável de contagem iniciada com zero e acessa-se cada nó da lista, incrementando-se a variável de contagem cada vez que um nó é acessado. A função `ComprimentoListaSE()` abaixo implementa essa ideia.

```
int ComprimentoListaSE(tListaSE lista)
{
   int tamanho = 0; /* Armazena o número de nós da lista */

      /* Acessa cada nó da lista e conta quantos nós são acessados */
   while (lista) {
      ++tamanho; /* Mais um nó foi encontrado */
      lista = lista->proximo; /* Passa para o próximo nó */
   }

   return tamanho;
}
```

A função `ComprimentoListaSE()` funciona mesmo quando a lista está vazia, pois, nesse caso, o corpo do laço não é executado e, assim, o valor da variável `tamanho` permanece sendo zero, que é o comprimento de uma lista que não contém nenhum nó. Note que, diferentemente das demais funções que representam operações sobre listas encadeadas exploradas antes, essa função independe do tipo do conteúdo efetivo (i.e., do tipo do campo `conteudo`) de cada nó da lista e também do fato de a lista ser ou não ordenada.

Inserção no Início

A maneira mais simples de inserir um nó numa lista simplesmente encadeada sem ordenação é fazê-lo no início da lista. Essa operação é implementada pela função `InsereListaSE()` abaixo.

```
void InsereListaSE(tListaSE *lista, tConteudo conteudo)
{
   tNoListaSE *ptrNovoNo; /* Apontará para o novo nó alocado */

      /* Tenta alocar um novo nó */
   ASSEGURA(ptrNovoNo = malloc(sizeof(tNoListaSE)), "Nao foi possivel alocar no'");

      /* Armazena no novo nó os dados recebidos como parâmetro */
   ptrNovoNo->conteudo = conteudo;

      /* O novo nó apontará para o início corrente da lista */
   ptrNovoNo->proximo = *lista;

      /* O início da lista passa a apontar para o novo nó */
   *lista = ptrNovoNo;
}
```

A função `InsereListaSE()` usa os seguintes parâmetros:

- `lista` (entrada e saída) — esse parâmetro é um ponteiro para ponteiro para a lista na qual será feita a inserção (v. esclarecimentos adiante).
- `conteudo` (entrada) — esse parâmetro é um ponteiro para os dados que serão armazenados no nó que será inserido.

Antes de discutir o algoritmo seguido pela função `InsereListaSE()`, faz-se necessário examinar atentamente o parâmetro `lista` usado por essa função, que é declarado como:

```
tListaSE *lista
```

Agora, `tListaSE` é o tipo da variável que armazena o endereço do início (i.e., que aponta para o primeiro nó) de uma lista encadeada. Ora, mas como o parâmetro `lista` é declarado como um ponteiro para o tipo `tListaSE`, isso significa que ele é um ponteiro para ponteiro para variáveis do tipo `tNoListaSE`. Em outras palavras, o parâmetro `lista` deve armazenar o endereço de um ponteiro que aponta para o primeiro nó de uma lista encadeada. Esse ponteiro é declarado desse modo porque, como a inserção é efetuada no início da lista, sempre que ocorre essa operação, o início da lista muda (i.e., o ponteiro que representa a lista deve ter seu valor alterado). Generalizando, pode-se enunciar a seguinte regra: quando uma função pode eventualmente alterar o início de uma lista encadeada, o parâmetro que representa a lista deve ser um ponteiro para ponteiro para o primeiro nó da lista.

A função `InsereListaSE()` acompanha o seguinte algoritmo:

1. Tente alocar um nó. Esse passo do algoritmo é implementado por uma chamada de **malloc()**:

   ```
   ASSEGURA( ptrNovoNo = malloc(sizeof(tNoListaSE)),
             "Nao foi possivel alocar no'" );
   ```

   Esquematicamente, se a chamada de **malloc()** for bem-sucedida, esse passo pode ser representado pela **Figura 10–3 (b)**. Os símbolos de interrogação que aparecem nessa figura significam que o conteúdo do nó alocado é indeterminado.

2. Armazene no novo nó os dados recebidos como parâmetro:

   ```
   ptrNovoNo->conteudo = conteudo;
   ```

   A execução desse passo faz com que o conteúdo da parte `conteudo` do novo nó alocado seja preenchido. Supondo que o valor desse conteúdo recebido como parâmetro seja `10`, a execução desse passo resulta na configuração apresentada na **Figura 10–3 (c)**. Nessa figura, o campo `conteudo` do nó foi preenchido, mas o campo `proximo` continua indeterminado (esse fato é representado pelo símbolo de interrogação).

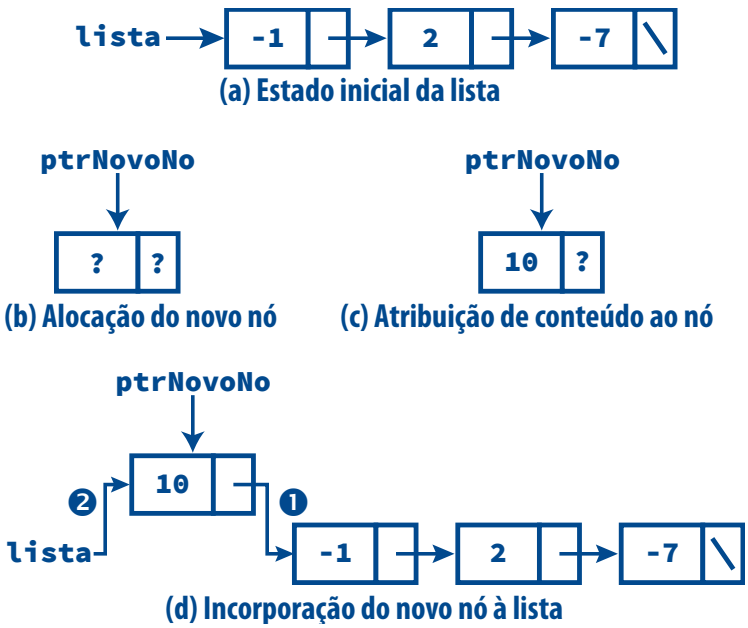


FIGURA 10–3: INSERÇÃO NO INÍCIO DE UMA LISTA SIMPLESMENTE ENCADEADA

3. Faça com que o novo nó aponte para o início corrente da lista. Esse passo é realizado pela instrução:

```
ptrNovoNo->proximo = *lista;
```

A execução desse passo faz com que o novo nó passe a apontar para o nó para o qual o *conteúdo* do parâmetro `lista` está correntemente apontando. É importante observar nessa instrução o uso do operador de indireção sobre o parâmetro `lista`. O uso desse operador se faz necessário porque, como foi afirmado, `lista` é um ponteiro para o ponteiro que aponta para o primeiro nó da lista. Portanto, aplicando-se o operador de indireção sobre esse parâmetro, obtém-se o endereço do primeiro nó da lista, que é o resultado desejado.

Assumindo que o estado inicial seja aquele ilustrado na **Figura 10–3 (a)**, a execução desse passo faz com que o ponteiro `proximo` do novo nó passe a apontar para o primeiro elemento da lista [representado pela conexão ❶ da **Figura 10–3 (d)**]. Nessa última ilustração, se a lista estivesse vazia, o campo `proximo` do novo nó assumiria o valor **NULL** porque, logo após sua inserção na lista, ele seria o primeiro e único nó da lista. Note que o novo nó ainda não foi incorporado à lista encadeada, o que será efetuado no próximo passo do algoritmo.

4. Faça com que o início da lista passe a apontar para o novo nó. Isso é realizado por meio da instrução:

```
*lista = ptrNovoNo;
```

Novamente, note a aplicação do operador de indireção sobre o parâmetro `lista`. Com o uso desse operador, essa última instrução é interpretada como: atribua o endereço do novo nó ao ponteiro que aponta para o início da lista. Se o parâmetro `lista` não tivesse sido definido como um ponteiro para o tipo `tListaSE`, não seria possível alterar o ponteiro para o início da lista, que é do tipo `tListaSE`. A situação ao final desse passo é representada pela conexão ❷ da **Figura 10–3 (d)**.

Remoção

A função `RemoveListaSE()`, apresentada abaixo, é usada para remover de uma lista simplesmente encadeada um nó cujo conteúdo efetivo apresenta um determinado valor especificado como parâmetro.

```
int RemoveListaSE(tListaSE *lista, tConteudo conteudo)
{
   tListaSE p = *lista, /* p aponta para o nó corrente     */
            q = NULL;   /* q aponta para o nó anterior a p */

      /* O laço while procura um nó cujo conteúdo case com o parâmetro 'conteudo' */
      /* e tem duas condições de parada: (1) todos os nós são acessados sem que o */
      /* referido nó tenha sido encontrado (neste  caso, p assume NULL); (2) o nó */
      /* contém o valor procurado que é encontrado.                               */
   while (p && p->conteudo != conteudo) {
      q = p; /* q passa a apontar para o nó corrente */
      p = p->proximo; /* p passa a apontar para o próximo nó */
   }

      /* Se p for NULL, a lista foi totalmente examinada   */
      /* sem que o nó a ser removido tenha sido encontrado */
   if (!p)
      return 1; /* Tentativa de remoção de nó inexistente */

      /*****************************************************/
      /* q aponta para o nó imediatamente anterior a ele  */
      /*****************************************************/

      /* Verifica se p aponta para o primeiro nó da lista porque */
      /* remoção no início deve ser tratada separadamente        */
   if (p == *lista) /* Remoção será no início da lista */
      (*lista) = p->proximo; /* Altera início da lista */
   else /* Nó a ser removido NÃO é o primeiro da lista */
      q->proximo = p->proximo; /* Desvia o nó anterior do nó a ser removido */

      /* Neste ponto, o nó desejado não faz mais parte da */
      /* lista, mas ele ainda ocupa espaço em memória.    */
      /* Portanto é necessário liberar este espaço.       */
   free(p); /* Libera o nó removido */

   return 0;
}
```

A função `RemoveListaSE()` usa os seguintes parâmetros:

- `lista` (entrada e saída) — esse parâmetro é um ponteiro para o ponteiro que representa a lista na qual será feita a remoção (v. observação sobre esse parâmetro abaixo).
- `conteudo` (entrada) — conteúdo do nó a ser removido.

A função sob discussão retorna zero, se a remoção for bem-sucedida, ou um valor diferente de zero se a remoção não for possível, o que ocorre apenas quando o nó a ser removido não for encontrado.

Novamente, como ocorre com a função `InsereListaSE()` apresentada acima, o primeiro parâmetro da função `RemoveListaSE()` deve conter o endereço de um ponteiro que aponta para o primeiro nó da lista. Isso ocorre porque, em princípio, qualquer nó pode ser removido, inclusive o primeiro nó da lista e, quando isso ocorre, o ponteiro para o início da lista deve ser alterado. O algoritmo seguido pela função `RemoveListaSE()` é o seguinte:

1. Tente encontrar um nó cujo conteúdo coincida com o parâmetro `conteudo`. Esse passo é concretizado pelo seguinte laço **while**:

```
while (p && p->conteudo != conteudo) {
   q = p;
   p = p->proximo;
}
```

A função `RemoveListaSE()` usa duas variáveis locais (ponteiros), `p` e `q`, para visitar os nós da lista, sendo que o `p` sempre aponta para um nó que é sucessor daquele para o qual `q` aponta. Esses ponteiros são iniciados como:

```
tListaSE  p = *lista,
          q = NULL;
```

O laço **while** acima procurará um nó cujo conteúdo é igual ao valor do segundo parâmetro da função `RemoveListaSE()`. Esse laço tem duas condições de parada: **(1)** todos os nós da lista são examinados sem que o nó procurado tenha sido encontrado (nesse caso, `p` assume o valor **NULL**); **(2)** o nó é encontrado (nesse caso, `p` aponta para esse nó e `q` aponta para seu antecessor). A ordem dos operandos do operador **&&** na expressão condicional do laço **while** é de suma importância, pois se essa ordem for invertida, um programa-cliente será abortado quando `p` assumir **NULL**.

2. Se o nó a ser removido não for encontrado, retorne um valor diferente de zero que indique esse fato. Esse passo é realizado pela instrução **if**:

```
if (!p)
   return 1;
```

Quando o nó procurado não é encontrado, o ponteiro `p` assume o valor **NULL** na saída do laço **while** da função em discussão. Assim o valor retornado indica que não houve remoção.

3. Se o nó a ser removido é o nó inicial da lista, faça com que o início da lista passe a ser o nó que o segue. Caso contrário, faça com que o nó anterior ao nó a ser removido aponte para o nó que segue o nó a ser removido. A seguinte instrução **if** realiza esse passo:

```
if (p == *lista)
   (*lista) = p->proximo;
else
   q->proximo = p->proximo;
```

A **Figura 10–4** ilustra a execução da função `RemoveListaSE()` quando o nó a ser removido é aquele cujo conteúdo é igual a `10`, o que corresponde ao primeiro elemento da lista.

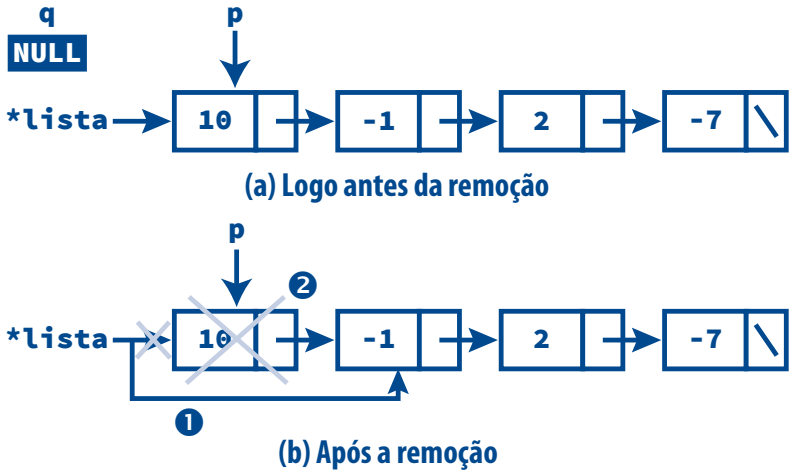


FIGURA 10–4: REMOÇÃO DO PRIMEIRO NÓ DE UMA LISTA SIMPLESMENTE ENCADEADA

Suponha, por exemplo, que na lista ilustrada na **Figura 10–5 (a)**, o nó que se deseja remover é aquele cujo conteúdo é igual a `2`. Então, essa figura mostra a situação ao final da execução do referido laço **while**, enquanto a **Figura 10–5 (b)** mostra a situação após a remoção ter sido efetuada.

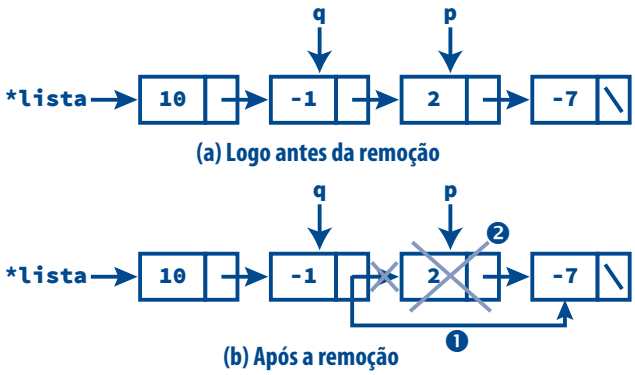


FIGURA 10–5: REMOÇÃO DE UM NÓ INTERNO DE UMA LISTA SIMPLESMENTE ENCADEADA

Como mostram as duas últimas figuras, após a execução do **Passo 3** do algoritmo, o nó que se deseja remover não faz mais parte da lista. Isto é, acessando-se o primeiro nó da lista e seguindo-se os ponteiros `proximo` dos nós não é mais possível atingir o nó removido. Alguns autores denominam a execução desse passo como remoção lógica.

4. Libere o espaço ocupado pelo nó removido da lista. Esse passo é efetuado pela seguinte chamada de **free**():

   ```
   free(p);
   ```

   Logo após a execução do **Passo 3**, o nó já está removido da lista, pois ele não pode mais ser acessado a partir do primeiro nó da lista. Entretanto, ele ainda ocupa espaço em memória desnecessariamente. Portanto é necessário liberar esse espaço que não está mais em uso. Alguns autores referem-se a execução desse passo como remoção física.

O último passo do algoritmo de remoção é retornar zero informando que não ocorreu erro durante a operação de remoção. Esse passo é trivial e não merece comentários adicionais.

Busca

A função `BuscaListaSE()`, apresentada a seguir, recebe como parâmetros um ponteiro para lista encadeada e um valor válido de conteúdo, e retorna o endereço do campo `conteudo` do nó que contém esse valor. Se o valor do segundo parâmetro não for encontrado em nenhum nó dessa lista, a referida função retorna **NULL**.

```
tConteudo *BuscaListaSE(tListaSE lista, tConteudo valor)
{
      /* Enquanto o ponteiro 'lista' não assume NULL ou o   */
      /* campo 'conteudo' de um nó não casa com o parâmetro */
      /* 'conteudo', a busca prossegue                      */
   while (lista && lista->conteudo != valor)
      lista = lista->proximo;

      /* Se 'lista' assume NULL é porque o conteúdo     */
      /* especificado como parâmetro não foi encontrado */
   if(!lista)
      return NULL; /* Nó não foi encontrado */

      /* O nó foi encontrado. Então, retorna-se o endereço de seu campo 'conteudo'. */
   return &lista->conteudo;
}
```

Deve-se salientar que, se o nó que armazena o conteúdo cujo endereço é retornado pela função `BuscaListaSE()` for removido da lista, esse conteúdo se tornará um zumbi. Também, como já foi comentado, os operandos do operador **&&** na expressão condicional do laço **while** não podem ter suas ordens trocadas (v. acima).

Destruição

Destruir uma lista significa visitar cada nó da lista liberando seu espaço em memória. Essa é a tarefa executada pela função `DestroiListaSE()` abaixo.

```
void DestroiListaSE(tListaSE *lista)
{
   tListaSE p; /* Aponta para o próximo nó a ser liberado */

   if (!*lista) /* Verifica se a lista está vazia */
      return; /* Lista vazia não precisa ser destruída */

   p = *lista; /* Faz p apontar para o início da lista */

      /* Visita cada nó da lista liberando-o */
   do {
         /* Passa para o próximo nó antes que o nó corrente seja destruído.  */
         /* Não importa que o ponteiro para o início da lista seja alterado. */
         /* Afinal, esse ponteiro será mesmo anulado.                         */
      *lista = (*lista)->proximo;

      free(p); /* Libera o espaço do nó corrente */

        /* Faz p apontar para o próximo nó, cujo endereço está armazenado em *lista */
      p = *lista;
   } while (p);

      /* Neste ponto, a lista ficou vazia, pois o  */
      /* último valor assumido por *lista foi NULL */
}
```

A operação implementada por `DestroiListaSE()` parece ser trivial, mas ela esconde uma armadilha na qual muitos programadores descuidados caem, tentados a imaginar que a destruição de uma lista encadeada pode ser implementada meramente como:

```
while (*lista) {
   free(*lista);
   (*lista) = (*lista)->proximo;
}
```

No entanto, esta abordagem não funciona porque, quando um bloco é liberado por meio de **free()**, ele não deve jamais ser usado novamente. Sendo assim, a instrução:

```
(*lista) = (*lista)->proximo;
```

não deveria vir em seguida à instrução:

```
free(*lista);
```

A mesma discussão apresentada na **Seção 3.5** a respeito de zumbis aplica-se aqui. Naquela seção, uma variável considerada zumbi era uma variável de duração automática que continuava sendo utilizada após ter sido liberada ao retorno da função na qual ela era definida. Aqui, zumbi é uma variável anônima alocada dinamicamente (bloco) liberada por meio de **free()** e que continua a ser usada. A única diferença substancial entre as duas categorias de zumbis são seus habitats. Zumbis alocados estaticamente habitam a pilha de execução, ao passo que o habitat natural de zumbis alocados dinamicamente é o heap.

A maneira correta de destruir uma lista encadeada é usando um ponteiro auxiliar (variável local) que guarda o endereço do próximo nó a ser liberado antes que o nó corrente seja liberado (nesse caso, foi usado o próprio parâmetro como ponteiro auxiliar).

Outro detalhe importante com respeito à função `DestroiListaSE()` é que, ao acessar os nós usando o parâmetro `lista`, ela altera o ponteiro para o início da lista, de modo que, ao final, a lista estará irremediavelmente perdida. Mas, no caso específico dessa função, não importa que o ponteiro para o início da lista seja alterado porque o objetivo dessa função é exatamente destruir a lista.

Acesso Sequencial

Várias operações sobre listas encadeadas requerem que os nós sejam visitados sequencialmente do primeiro ao último nó. Visitar um elemento de uma lista (encadeada ou não) significa acessá-lo com o objetivo de realizar alguma operação sobre os dados que ele armazena. Esta seção apresenta as operações mais comuns que envolvem acesso sequencial a listas encadeadas.

A função `ProximoListaSE()` retorna o endereço do conteúdo efetivo do sucessor de cada nó de uma lista simplesmente encadeada a partir do seu primeiro nó, permitindo, assim, acesso sequencial à lista. Essa função é definida como:

```
tConteudo *ProximoListaSE(tListaSE lista)
{
   static tNoListaSE *proximoNo = NULL;

   if (lista) {
      if (!proximoNo)
         proximoNo = lista;
      else
         proximoNo = proximoNo->proximo;

      return proximoNo ? &proximoNo->conteudo : NULL;
   } else
      return NULL;
}
```

A função `ProximoListaSE()` define uma variável local de duração fixa, denominada `proximoNo`, que é iniciada com **NULL** no início da execução do programa (v. **Seção 2.3**). Assim, na primeira chamada dessa função, o valor dessa variável continua sendo **NULL**. Quando essa função é chamada e seu parâmetro, que representa a lista em questão é **NULL**, ela retorna esse valor imediatamente. Caso contrário, se a variável `proximoNo` for **NULL**, a função faz com que essa variável aponte para o início da lista recebida como parâmetro. Se essa variável não for **NULL**, ela apontará para o sucessor do nó que ela antes apontava. Em qualquer dos casos, a função retorna o endereço do campo `conteudo` do nó para o qual `proximoNo` aponta.

Idealmente, a função `ProximoListaSE()` deve ser chamada por um cliente no interior de um laço de repetição, como faz a função `ExibeDoInicio()` a seguir, que exibe na tela do primeiro ao último elemento de uma lista simplesmente encadeada do tipo `tListaSE`. Exibir uma lista encadeada na tela significa visitar cada nó da lista com o objetivo de apresentar no meio de saída padrão o conteúdo efetivo do nó.

```
void ExibeDoInicio(tListaSE lista)
{
   tConteudo *pConteudo = NULL;

   while (pConteudo = ProximoListaSE(lista))
      printf("%d\t", *pConteudo);
}
```

A função `ExibeDoInicio()` constitui um exemplo de visita sequencial aos nós de uma lista encadeada com o objetivo de realizar alguma operação sobre cada um deles. No caso corrente, a operação executada é a exibição do conteúdo efetivo de cada nó, mas, para a execução de qualquer outra operação a abordagem básica é a

mesma. Ou seja, para executar qualquer outra operação sobre cada nó de uma lista encadeada, a única alteração a ser feita é substituir as chamadas de **printf()** na função `ExibeDoInicio()` por chamadas de funções que executem a devida operação.

Outro exemplo de acesso sequencial aos elementos de uma lista consiste em acessar sequencialmente cada nó de uma lista encadeada com o objetivo de escrever seu conteúdo efetivo num arquivo binário, como faz a função `AtualizaArquivoBinLSE()`, definida abaixo.

```
int AtualizaArquivoBinLSE(const char *nomeArq, tListaSE lista)
{
   FILE      *stream; /* Stream associado ao arquivo no qual ocorrerá a escrita */
   tConteudo *pConteudo = NULL;

      /* Abre arquivo em formato binário para escrita */
   stream = fopen(nomeArq, "wb");

      /* Se o arquivo não foi aberto, nada mais pode ser feito */
   if (!stream)
      return 1; /* Arquivo não foi aberto */

      /* Garante que o acesso começa do início da lista */
   while (ProximoListaSE(lista))
      ; /* Instrução vazia */

      /* Acessa cada nó da lista e escreve seu conteúdo efetivo no arquivo binário */
   while ((pConteudo = ProximoListaSE(lista))) {
         /* Tenta escrever o conteúdo do nó corrente no arquivo */
      fwrite(&lista->conteudo, sizeof(*pConteudo), 1, stream);

         /* Se ocorreu erro de escrita, fecha o arquivo */
         /* e retorna um valor indicativo do fato       */
      if (ferror(stream)) {
         fclose(stream);
         return 1;
      }
   }

   fclose(stream); /* Processamento terminado. Fecha o arquivo. */

   return 0; /* Se ocorreu erro, já houve retorno */
}
```

A função `AtualizaArquivoBinLSE()` abre o arquivo, cujo nome é recebido como parâmetro, para escrita em modo binário. Então, ela usa um laço **while** para visitar sequencialmente cada nó da lista cujo endereço inicial é recebido como parâmetro. Quando um nó é visitado, seu campo `conteudo` é escrito no arquivo por meio de uma chamada de **fwrite()** (v. **Seção 7.6.2**). O laço encerra quando a função `ProximoListaSE()` retorna **NULL**, o que significa que todos os nós foram visitados, ou quando ocorre erro de escrita no arquivo. Nesse último caso, a função em discussão fecha o arquivo e retorna um valor que indica a ocorrência de erro.

10.3 Lista Simplesmente Encadeada com Ordenação

10.3.1 Abstração

Conforme foi visto na **Seção 7.2**, uma **lista ordenada** é uma lista na qual os elementos ocupam posições de acordo com uma determinada ordem, de modo que a operação **(3)** definida na **Seção 10.2** precisa ser redefinida para que a lista seja mantida ordenada. Mais precisamente, a operação:

[5] [3] **Inserção de um elemento** na lista.

precisa ser reescrita como:

[6] [3'] **Inserção de um elemento** numa posição tal que a lista permaneça ordenada

A operação:

[7] [6] **Alteração de valor de um elemento**

descrita na **Seção 10.2** deixa de fazer sentido para listas ordenadas visto que ela pode causar desordem da lista.

10.3.2 Implementação

Inserção em Ordem

Para manter uma lista encadeada ordenada, é necessário escrever uma nova função de inserção, pois, nesse caso, a inserção pode ocorrer em qualquer posição na lista (e não apenas no início, como foi o caso na última função apresentada). Essa nova função, denominada `InsereEmOrdemLSE()`, será definida a seguir.

```
void InsereEmOrdemLSE(tListaSE *lista, tConteudo conteudo)
{
   tNoListaSE *ptrNovoNo; /* Apontará para o novo nó alocado */
   tListaSE    p = *lista, /* p aponta para o nó corrente     */
               q = NULL;   /* q aponta para o nó anterior a p */

      /* Tenta alocar um novo nó */
   ASSEGURA(ptrNovoNo = malloc(sizeof(tNoListaSE)), "Nao foi possivel alocar no'");

      /* Armazena no novo nó os dados recebidos como parâmetro */
   ptrNovoNo->conteudo = conteudo;

     /* Se a lista estiver vazia, basta adicionar o nó ao início da lista e retornar */
   if (!*lista) {
      *lista = ptrNovoNo; /* Início da lista apontará para o novo nó */
      (*lista)->proximo = NULL; /* O novo nó é o único da lista */
      return;
   }

         /*************************************************************/
         /* A lista tem pelo menos um nó. Nesse caso, a posição de    */
         /* inserção é aquela do primeiro nó cujo conteúdo seja maior */
         /* do que aquele do nó a ser inserido. Se tal nó não for     */
         /* encontrado, o novo nó será o último da lista.             */
         /*************************************************************/

      /* Procura o local de inserção */
   while ( p && p->conteudo <= conteudo ) {
      q = p;    /* q passa a apontar para o nó corrente */
      p = p->proximo; /* p passa a apontar para o próximo nó */
   }

      /*********************************************************/
      /* Neste instante, p aponta para o local da inserção e q  */
      /* aponta para o nó imediatamente antes desse local. Se   */
      /* p for NULL, o novo nó será o último da lista, mas isso */
      /* não constitui um caso especial a ser tratado à parte.  */
      /*********************************************************/

      /* O novo nó apontará para o nó apontado por p */
   ptrNovoNo->proximo = p;

      /* Verifica se p aponta para o primeiro nó da lista pois */
      /* inserção no início deve ser tratada separadamente     */
```

```
   if (p == *lista) /* Inserção será no início da lista */
      *lista = ptrNovoNo;/* Inserção no início é especial */
   else
         /* O nó anterior àquele apontado por p apontará para o novo nó */
      q->proximo = ptrNovoNo;
}
```

Os parâmetros da função `InsereEmOrdemLSE()` têm a mesma interpretação daqueles da função `InsereListaSE()` apresentada na **Seção 10.2** e ambas as funções também possuem a mesma especificação de retorno. Entretanto, essas funções implementam algoritmos diferentes.

A função `InsereEmOrdemLSE()` segue o algoritmo abaixo:

1. Tente alocar um nó. Esse passo do algoritmo é implementado por uma chamada de **malloc()**:

   ```
   ptrNovoNo = malloc(sizeof(tNoListaSE));
   ```

2. Se a alocação do nó não for possível, aborte o programa:

   ```
   ASSEGURA(ptrNovoNo = malloc(sizeof(tNoListaSE)), "Nao foi possivel alocar no'");
   ```

3. Armazene no novo nó os dados recebidos como parâmetro:

   ```
   ptrNovoNo->conteudo = conteudo;
   ```

4. Se a lista estiver vazia, adicione o novo nó ao início da lista e encerre. Esse passo é realizado pela instrução **if**:

   ```
   if (!*lista) {
      *lista = ptrNovoNo;
      (*lista)->proximo = NULL;
      return;
   }
   ```

   Até esse passo, as funções `InsereEmOrdemLSE()` e `InsereListaSE()` são semelhantes, apesar de, aqui, a inserção no início da lista ter sido escrita de maneira ligeiramente diferente (mas equivalente). Note o uso de parênteses na instrução:

   ```
   (*lista)->proximo = NULL;
   ```

   Esses parênteses são necessários porque o operador `->` tem precedência maior do que o operador `*` (v. **Apêndice A**).

5. Encontre o local de inserção. Esse passo é implementado pelo laço **while**:

   ```
   while ( p && p->conteudo <= conteudo ){
      q = p;
      p = p->proximo;
   }
   ```

   A função em discussão usa dois ponteiros (variáveis locais), denominados `p` e `q`, que são usados para localização da posição na qual o novo nó será inserido. Esses ponteiros são assim iniciados:

   ```
   tListaSE  p = *lista,
             q = NULL;
   ```

   Quando a posição de inserção for encontrada, `p` apontará para o nó que cederá lugar ao nó sendo inserido e `q` apontará para o nó imediatamente anterior àquele para o qual `p` aponta. Esses dois ponteiros são necessários porque, como será visto adiante, os dois nós para os quais eles apontam precisarão ser acessados durante o processo de inserção.

Como a lista deve ser ordenada pelo conteúdo efetivo de cada nó, o laço **while** acima encerra quando é encontrado o primeiro nó cujo valor de seu conteúdo efetivo é maior do que o valor do novo nó. Se não existir nenhum nó na lista com essa propriedade, o novo nó será inserido como último nó da lista. Nesse último caso, o valor de `p` será **NULL**. É importante salientar que a ordem dos operandos do operador **&&** na expressão condicional desse laço **while** não pode ser alterada. Quer dizer, se a expressão do laço **while** for substituída por:

```
p->conteudo <= conteudo && p
```

o programa será abortado quando não houver nenhum nó na lista com conteúdo maior do aquele do nó sendo inserido (i.e., quando `p` for **NULL**).

A **Figura 10–6** mostra diagramaticamente a situação logo após a execução do **Passo 5** do algoritmo em discussão durante a inserção de um nó com conteúdo igual a `2` numa lista que, neste instante, contém três nós.

Dando seguimento ao algoritmo de inserção em ordem, o próximo passo a ser seguido é:

6. Faça o novo nó apontar para `p`, o que pode ser implementado como:

```
ptrNovoNo->proximo = p;
```

Após a execução dessa instrução, a inserção do nó com conteúdo igual a `2` na lista da **Figura 10–6** pode ser ilustrada pela conexão ❶ na **Figura 10–7**.

7. Se `p` apontar para o primeiro nó da lista, faça o ponteiro para o início da lista apontar para o novo nó. Caso contrário, faça o nó anterior a `p` apontar para o novo nó. Esse passo é implementado pela seguinte instrução **if**:

```
if (p == *lista)
   *lista = ptrNovoNo;
else
   q->proximo = ptrNovoNo;
```

Logo após a execução dessa instrução, a inserção do nó com conteúdo igual a `2` na lista da **Figura 10–6** pode ser ilustrada pela conexão ❷ na **Figura 10–7**.

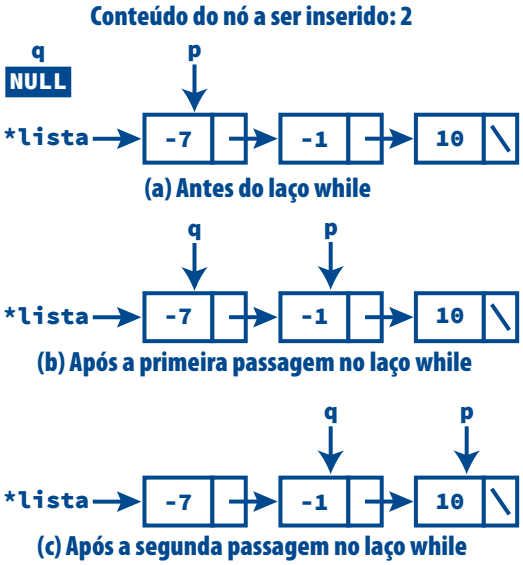


Figura 10–6: Inserção em Lista Encadeada Ordenada 1: Encontrando a Posição

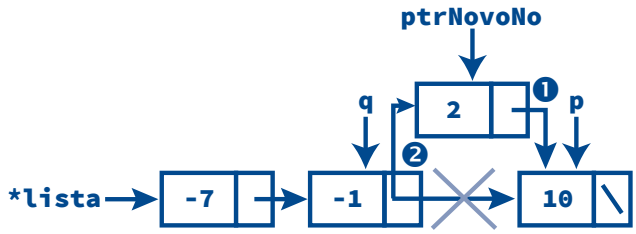


FIGURA 10–7: INSERÇÃO EM LISTA ENCADEADA ORDENADA 2: INSERINDO

Busca

Conforme foi visto nas **Seções 7.2.3** e **9.6.2**, no caso de listas indexadas, operações de busca podem apresentar um desempenho superior quando as listas são ordenadas. Ocorre, porém que listas encadeadas não permitem implementação de busca binária de modo eficiente. Mesmo assim, quando uma lista encadeada é ordenada, a busca sequencial pode ser implementada de modo mais eficiente do que foi visto antes, como mostra a função `BuscaListaSEOrd()` apresentada a seguir.

```
tConteudo *BuscaListaSEOrd(tListaSE lista, tConteudo valor)
{
      /* Enquanto o ponteiro 'lista' não assume NULL ou o campo 'conteudo' de */
      /* um nó não é maior do que o parâmetro 'valor', a busca prossegue      */
   while (lista && lista->conteudo < valor)
      lista = lista->proximo;

      /* Se 'lista' assume NULL é porque o conteúdo     */
      /* especificado como parâmetro não foi encontrado */
   if(!lista)
      return NULL; /* Nó não foi encontrado */

      /* Verifica se o nó foi encontrado */
   if (lista->conteudo == valor)
      return &lista->conteudo;
   else
      return NULL; /* Nó não foi encontrado */
}
```

A função `BuscaListaSEOrd()` é mais eficiente do que a função `BuscaListaSE()` apresentada na **Seção 10.2**, pois ela pode encurtar a busca quando o nó procurado não se encontrar na lista. Isso ocorre porque o laço **while** da função `BuscaListaSEOrd()` encerra quando é encontrado um valor que é maior do que ou igual àquele procurado. Assim, quando o conteúdo do nó procurado não é maior do que o maior conteúdo armazenado na lista, a busca encerra antes de a lista ser totalmente examinada. Mesmo assim, do ponto de vista assintótico, ambas as funções em discussão apresentam o mesmo custo temporal, que é $\theta(n)$ (v. **Seção 10.5**).

10.4 Outros Tipos de Listas Encadeadas

O tipo de lista encadeada visto nas seções precedentes é denominado *lista simplesmente encadeada*, porque o encadeamento é feito num único sentido e existem listas com encadeamento duplo. Além disso as listas vistas anteriormente são consideradas **lineares**, pois existem listas encadeadas circulares. Esta seção dedica-se a discutir outros tipos listas encadeadas.

Uma lista encadeada pode ser classificada como:

[1] Simples ou duplamente encadeada

[2] Linear ou circularmente encadeada

[3] Com ou sem cabeça

Como estes atributos são independentes, existem oito (2^3) tipos de listas encadeadas, mas, do ponto de vista prático, muitas dessas combinações não fazem muito sentido. O programador deve determinar qual destes tipos utilizará examinando as operações requeridas sobre a lista e determinando que casos especiais (lista vazia, final de lista, etc.) são difíceis de tratar. O tipo de lista que torna os algoritmos que as processam mais simples deve ser o escolhido.

10.4.1 Lista Duplamente Encadeada Linear

A impossibilidade de se percorrer uma lista simplesmente encadeada no sentido inverso ao indicado pelos ponteiros sugere a construção de listas duplamente encadeadas. Cada nó de uma lista duplamente encadeada possui dois ponteiros: um apontando para o nó que o segue e outro apontando para o nó que o antecede.

A **Figura 10–8** mostra esquematicamente uma lista duplamente encadeada linear com três nós.

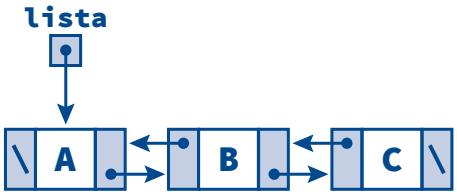


FIGURA 10–8: LISTA DUPLAMENTE ENCADEADA COM TRÊS NÓS

É evidente que uma lista duplamente encadeada ocupa mais espaço em memória do que uma lista simplesmente encadeada. No entanto, a liberdade de movimento em ambos os sentidos simplifica algumas operações sobre listas encadeadas, economizando tempo de execução e espaço para armazenamento dos programas. Por outro lado, o maior número de conexões envolvidas em algumas operações (notadamente, inserção e remoção) requerem atenção redobrada por parte do programador para evitar erros de programação. Assim, antes de implementar operações de inserção e remoção sobre listas, é importante que o programador construa representações gráficas que mostrem como as conexões são alteradas durante essas operações.

A seguinte definição de tipos representa uma possível implementação de listas duplamente encadeadas, cujos campos de informação são do tipo `tConteudo`, podendo esse tipo ser qualquer tipo pré-definido ou definido pelo programador.

```
typedef struct rotNoLDE {
           struct rotNoLDE *anterior;
           tConteudo        conteudo;
           struct rotNoLDE *proximo;
        } tNoListaDE, *tListaDE;
```

Inserção

A **Figura 10–9** ilustra a inserção de um nó interno, enquanto a **Figura 10–10** mostra a inserção do primeiro nó numa lista duplamente encadeada linear. Por outro lado, a **Figura 10–11** mostra a inserção do último nó de uma lista dessa natureza.

Nas figuras em questão, pode-se observar, por exemplo, que, nos três casos, as instruções ❶ e ❷ são idênticas, de modo que elas não representam nenhum caso especial.

É relevante notar que, em muitos casos, a ordem com que as conexões são efetuadas nas três últimas figuras faz grande diferença.

A função `InsereEmOrdemLDE()`, que implementa a inserção de nós em listas duplamente encadeadas lineares levando em consideração os três casos ilustrados nas três últimas figuras, encontra-se no site dedicado ao livro.

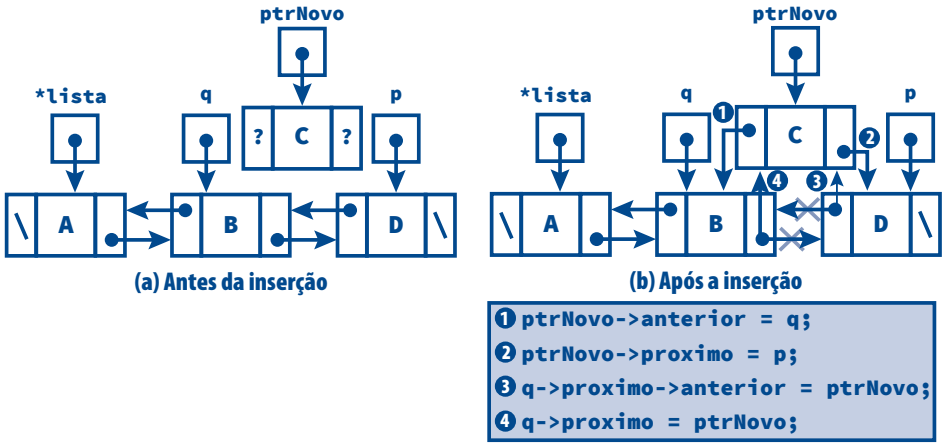


FIGURA 10–9: INSERÇÃO DE NÓ INTERNO NUMA LISTA DUPLAMENTE ENCADEADA LINEAR

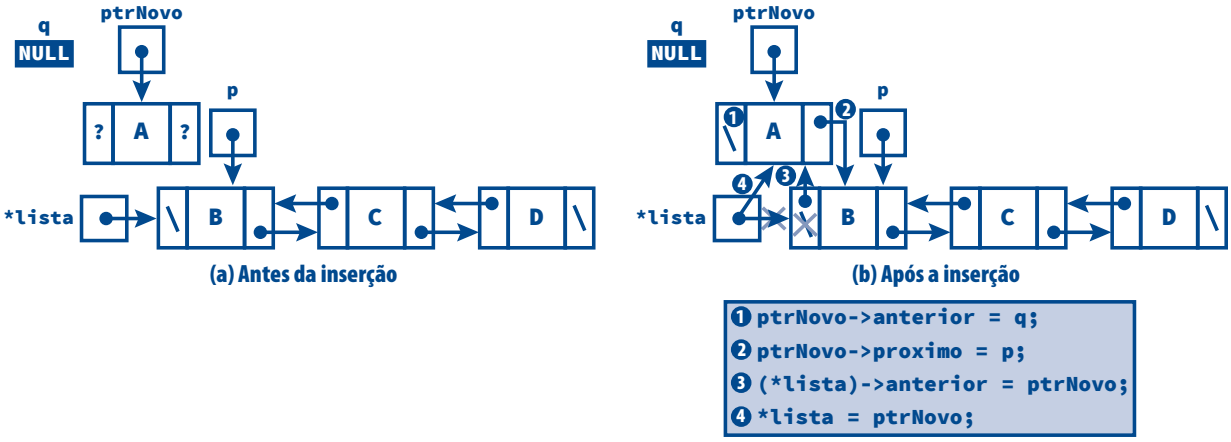


FIGURA 10–10: INSERÇÃO DO PRIMEIRO NÓ NUMA LISTA DUPLAMENTE ENCADEADA LINEAR

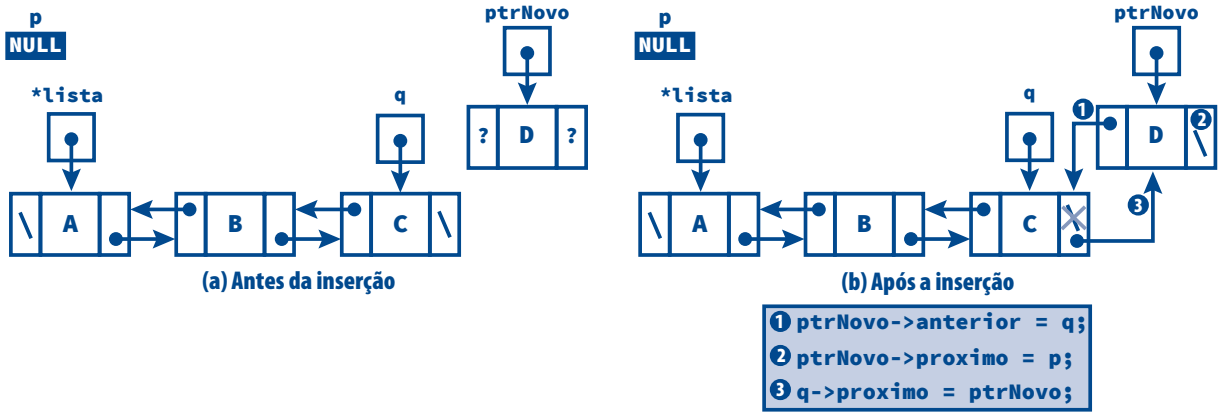


FIGURA 10–11: INSERÇÃO DO ÚLTIMO NÓ NUMA LISTA DUPLAMENTE ENCADEADA LINEAR

Remoção

A **Figura 10–12** ilustra a remoção de um nó interno de uma lista duplamente encadeada, a **Figura 10–13** mostra a remoção do primeiro nó de uma lista duplamente encadeada e a **Figura 10–14** ilustra a remoção do último nó interno de uma lista duplamente encadeada.

A função `RemoveListaDE()`, que remove um nó de uma lista duplamente encadeada linear considerando as situações expostas nas três últimas figuras, encontra-se no site dedicado a este livro.

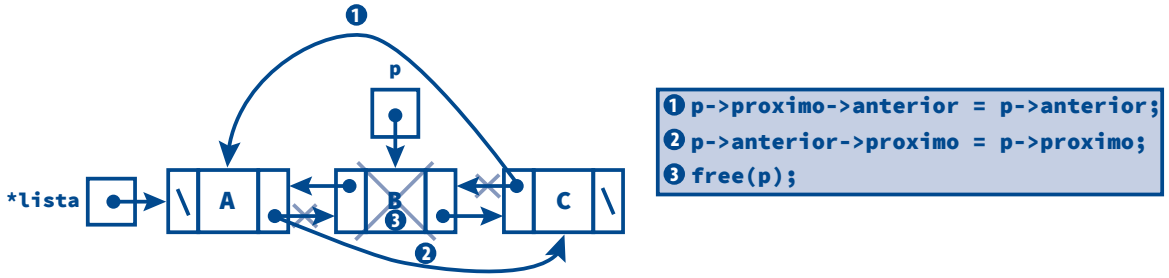


FIGURA 10–12: REMOÇÃO DE NÓ INTERNO DE UMA LISTA DUPLAMENTE ENCADEADA LINEAR

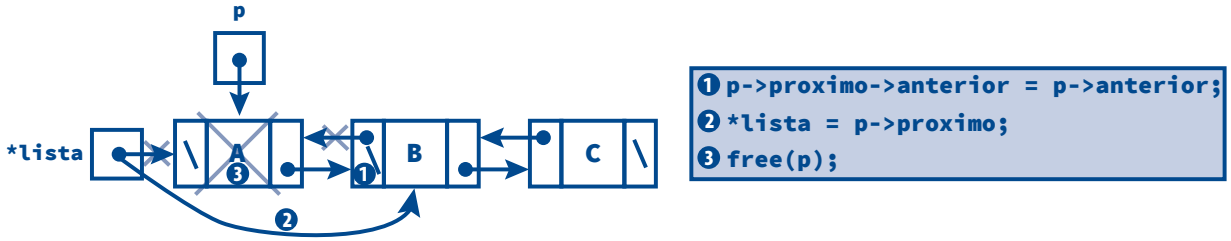


FIGURA 10–13: REMOÇÃO DO PRIMEIRO NÓ DE UMA LISTA DUPLAMENTE ENCADEADA LINEAR

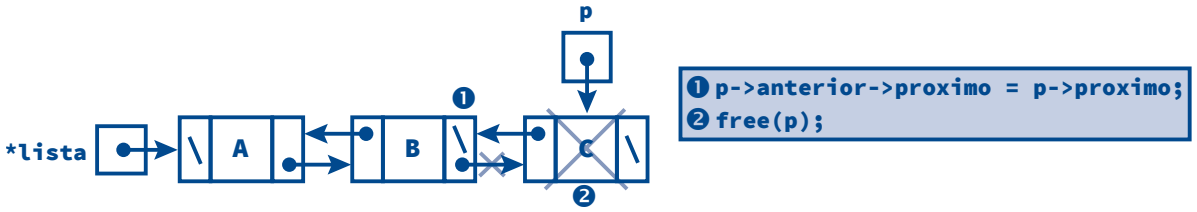


FIGURA 10–14: REMOÇÃO DO ÚLTIMO NÓ DE UMA LISTA DUPLAMENTE ENCADEADA LINEAR

Acesso Sequencial Invertido

A função `AnteriorListaDE()` retorna o endereço do conteúdo efetivo do antecessor de cada nó de uma lista duplamente encadeada linear a partir do último nó da lista. Essa função permite a um programa-cliente acesso sequencial a uma lista e pode ser definida como:

```
tConteudo *AnteriorListaDE(tListaDE lista)
{
   static tNoListaDE *noAnterior = NULL;

   if (lista) {
      if (!noAnterior)
         noAnterior = UltimoNo(lista);
      else
         noAnterior = noAnterior->anterior;

      return noAnterior ? &noAnterior->conteudo : NULL;
   } else
      return NULL;
}
```

A função `AnteriorListaDE()` chama a função `UltimoNo()` para fazer o ponteiro apontar `noAnterior` apontar para o último nó da lista. A função `UltimoNo()` é uma função auxiliar de implementação e, por isso, é definida com o qualificador **static** (v. **Seção 2.6.2**). Essa função é implementada como:

```
static tListaDE UltimoNo(tListaDE lista)
{
   tListaDE p = lista,
            q = NULL;

   while (p) {
      q = p;
      p = p->proximo;
   }

   return q;
}
```

Utilizando a função um programa-cliente pode implementar uma operação de acesso sequencial invertida sobre uma lista duplamente encadeada linear, como a função `ExibeDoFinal()`, apresentada a seguir, que exibe na tela do último ao primeiro elemento de uma lista duplamente encadeada.

```
void ExibeDoFinal(tListaDE lista)
{
   tConteudo *pConteudo = NULL;

   printf("\n\t***** Lista Invertida ***** \n");

   while ((pConteudo = AnteriorListaDE(lista)))
      printf("%d\t", *pConteudo);

   putchar('\n');
}
```

10.4.2 Lista Simplesmente Encadeada Circular

Em algumas funções que executam operações sobre listas encadeadas, o último nó da lista deve ser tratado de modo diferente dos demais nós, porque ele não aponta para nenhum outro nó. Essa situação pode ser modificada fazendo-se com que o último nó aponte para o primeiro elemento da lista. Tem-se, assim, a chamada **lista encadeada circular**. Tal lista pode ser simples ou duplamente encadeada. O final de uma lista circular encadeada pode ser reconhecido encontrando-se o nó que aponta para o mesmo nó que a própria variável lista. Uma lista simplesmente encadeada circular é mostrada na **Figura 10–15**.

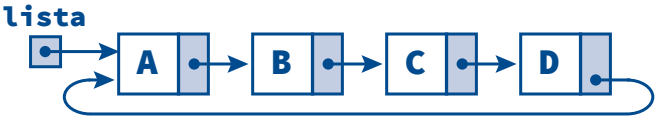


Figura 10–15: Lista Simplesmente Encadeada Circular com Quatro Nós

Note na **Figura 10–15** que qualquer nó de uma lista circular possui um sucessor e um antecessor. Assim, utilizando-se uma lista circular, pode-se visitar todos os nós da lista a partir de qualquer nó.

Uma lista simplesmente encadeada circular utiliza as mesmas definições de tipo de uma lista simplesmente encadeada linear que armazena o mesmo conteúdo efetivo. Porém, as implementações de algumas operações devem ser diferentes das implementações correspondentes para listas simplesmente encadeadas lineares. Mais especificamente, todas as operações que requerem que os nós da lista sejam visitados requerem alteração para levar em consideração que não há mais ponteiro igual a **NULL** na lista. Assim qualquer operação que requeira percorrer toda uma lista circular deve ter como nó de encerramento o mesmo nó em que o percurso começou.

Inserção

A inserção do primeiro nó de uma lista simplesmente encadeada circular é mostrada esquematicamente na **Figura 10–16**. Note que o novo nó aponta para si mesmo, já que ele é o único nó na lista.

Figura 10–16: Primeiro Nó de uma Lista Simplesmente Encadeada Circular

A **Figura 10–17** mostra a inserção de um nó interno numa lista simplesmente encadeada circular.

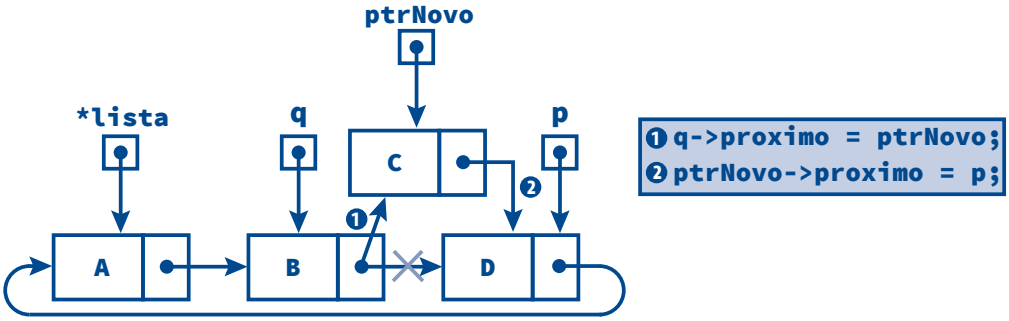


Figura 10–17: Inserção numa Lista Simplesmente Encadeada Circular

A inserção de um novo nó no início de uma lista simplesmente encadeada circular é mostrada esquematicamente na **Figura 10–18**.

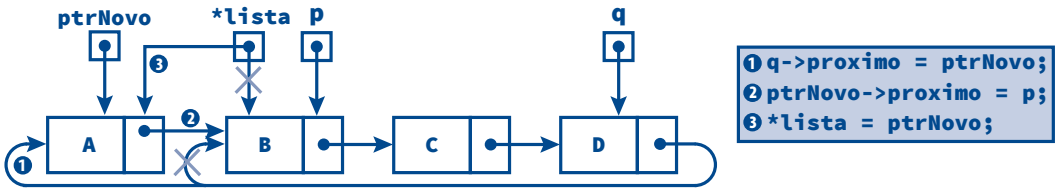


Figura 10–18: Inserção no Início de uma Lista Simplesmente Encadeada Circular

A inserção de um novo nó no final de uma lista encadeada circular ordenada é mostrada esquematicamente na **Figura 10–19**. Note que as posições dos ponteiros **p** e **q** são as mesmas quando a inserção se dá no início ou ao final de uma lista simplesmente encadeada circular, de modo que os valores desses ponteiros não permitem discernir os dois tipos de inserção. Observe ainda que inserção no início da lista deixa de ser o melhor caso de inserção, como ocorre com lista simplesmente encadeada lineares.

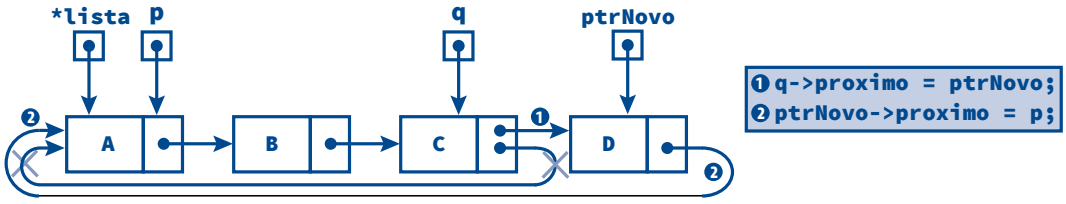


Figura 10–19: Inserção no Final de uma Lista Simplesmente Encadeada Circular

Uma função de inserção de nós numa lista simplesmente encadeada circular pode ser encontrada no site dedicado a este livro.

É interessante notar que, numa lista simplesmente encadeada circular, os conceitos de início e final de lista deixam de fazer sentido se a lista não for ordenada. Quer dizer, qualquer nó pode ser considerado nó inicial e qualquer nó pode ser considerado nó final de uma lista circular encadeada sem ordenação, como mostra a **Figura 10–20**.

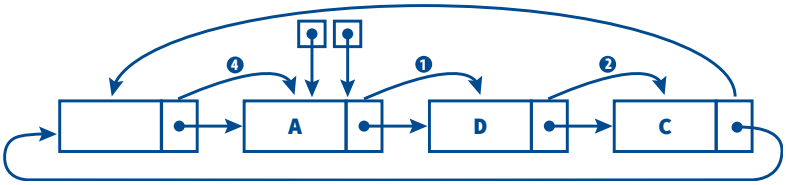


Figura 10–20: Não Há Início nem Final em Lista Encadeada Circular sem Ordenação

Remoção

A **Figura 10–21** mostra a remoção de um nó interno numa lista simplesmente encadeada circular.

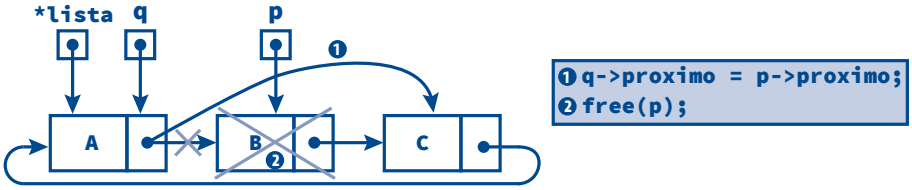


Figura 10–21: Remoção de Nó Interno em Lista Simplesmente Encadeada Circular

A remoção do primeiro nó de uma lista simplesmente encadeada circular é ilustrada na **Figura 10–22**.

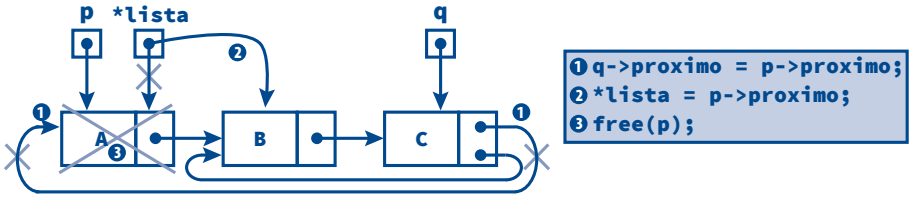


Figura 10–22: Remoção no Início de uma Lista Simplesmente Encadeada Circular

A remoção do único nó de uma lista duplamente encadeada circular é apresentada na **Figura 10–23**.

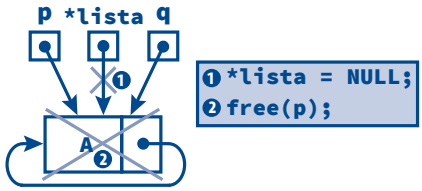


Figura 10–23: Remoção do Único Nó de uma Lista Simplesmente Encadeada Circular

A **Figura 10–24** mostra a remoção do último nó de uma lista encadeada circular ordenada.

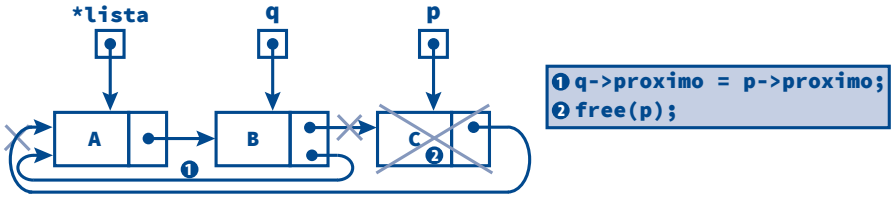


Figura 10–24: Remoção no Final de uma Lista Simplesmente Encadeada Circular

Observe que a remoção do primeiro nó de uma lista simplesmente encadeada circular não é caso especial (v. **Figura 10–22**), a não ser que ele seja o nó único da lista (v. **Figura 10–23**). A remoção do último nó de uma lista simplesmente encadeada circular também não constitui um caso especial, como mostram a **Figura 10–21** e a **Figura 10–24**.

A função `RemoveListaSEC()`, que remove um nó de uma lista simplesmente encadeada circular pode ser encontrada no site dedicado ao livro.

Busca

A função `BuscaListaSEC()` implementa a operação de busca em listas simplesmente encadeadas circulares ordenadas ou sem ordenação. Essa função encontra-se no site dedicado a este livro na internet.

Destruição

A função `DestroiListaSEC()` libera o espaço ocupado pelos nós de uma lista simplesmente encadeada circular, tornando-a vazia. No site dedicado a este livro, você encontrará o código dessa função.

Acesso Sequencial

A função `ProximoListaSEC()` retorna o endereço do conteúdo do próximo nó de uma lista simplesmente encadeada circular. A implementação dessa função encontra-se no site dedicado a este livro na internet.

10.4.3 Lista Duplamente Encadeada Circular

Certamente, listas duplamente encadeadas circulares constituem o tipo de lista encadeada mais flexível que existe (v. exemplo na **Seção 10.7.5**). No entanto, inserção e remoção de nós em listas duplamente encadeadas são mais complicadas do que operações correspondentes para listas simplesmente encadeadas e requerem cuidados especiais por causa do número de ponteiros envolvidos nessas operações.

A **Figura 10–25** mostra uma lista circular duplamente encadeada com quatro nós.

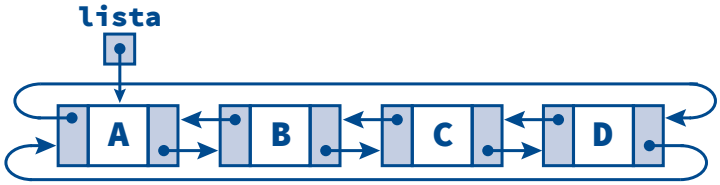


FIGURA 10–25: LISTA CIRCULAR DUPLAMENTE ENCADEADA COM QUATRO NÓS

Listas duplamente encadeadas circulares podem ser criadas utilizando as mesmas definições de tipos usadas para listas duplamente encadeadas lineares (v **Seção 10.4.1**).

Inserção

Quando o nó a ser incorporado na lista é o primeiro, procede-se como no caso de lista simplesmente encadeada (v. **Seção 10.2.2**), mas deve-se ainda fazer com que o nó aponte para si mesmo, como mostra a **Figura 10–26**.

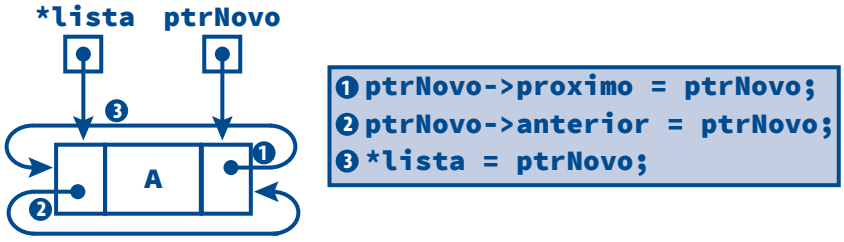


FIGURA 10–26: ACRÉSCIMO DO NÓ INICIAL DE LISTA DUPLAMENTE ENCADEADA CIRCULAR

A **Figura 10–27** ilustra o acréscimo de um nó no início de uma lista circular duplamente encadeada.

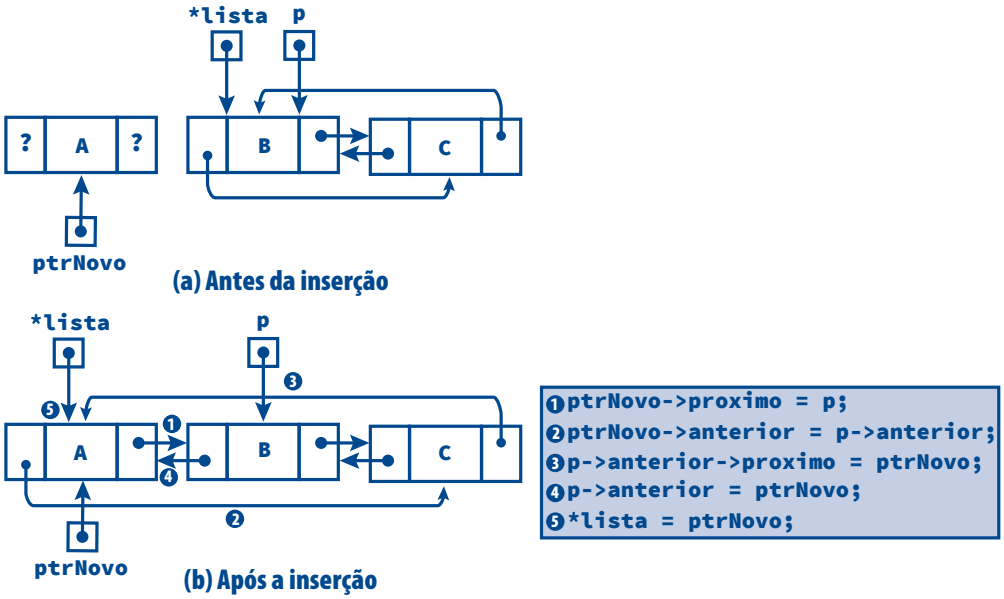


Figura 10–27: Inserção de Nó Inicial numa Lista Duplamente Encadeada Circular

Note na **Figura 10–27** que cinco ponteiros precisam ser alterados e que essa alteração deve obedecer uma certa ordem:

- Os ponteiros `anterior` e `proximo` do novo nó devem ser alterados primeiro. A ordem relativa das alterações ❶ e ❷ não importa, mas elas devem vir antes das demais.
- As alterações ❸ e ❹ devem ser efetuadas em seguida e exatamente nessa ordem.
- O último ponteiro a ser alterado deve ser aquele que aponta para o início da lista.

Se a lista estiver vazia, o acréscimo de um nó ao seu final é o mesmo que o acréscimo de um nó no seu início e ocorre conforme foi mostrado acima. A **Figura 10–28** ilustra o acréscimo de um nó no final de uma lista circular duplamente encadeada contendo dois nós.

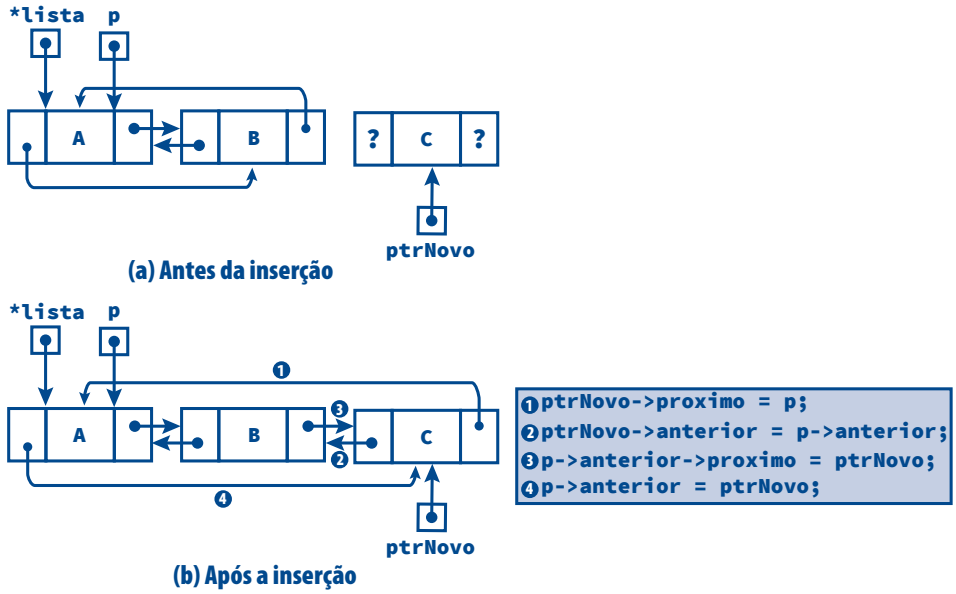


Figura 10–28: Acréscimo de Nó Final de uma Lista Duplamente Encadeada Circular

Compare a **Figura 10–28** com a **Figura 10–27** e note que a única diferença entre os conjuntos de instruções que promovem as devidas alterações nos ponteiros é que o ponteiro para o início da lista deve ser alterado quando a inserção ocorre no início da lista. Agora, observe a **Figura 10–29** que mostra a inserção de um nó interno numa lista circular duplamente encadeada.

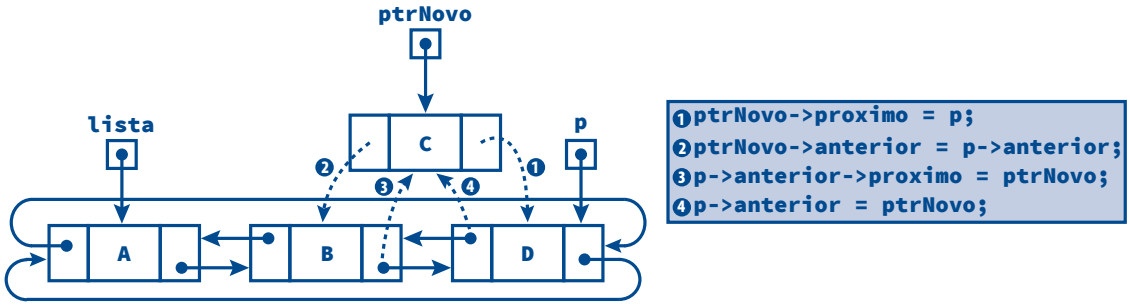


Figura 10–29: Inserção de Nó Interno numa Lista Duplamente Encadeada Circular

Note que os casos especiais de inserção em lista circular duplamente encadeada ocorrem no início da lista, quer ela esteja vazia ou não.

A função `InsereEmOrdemLDEC()`, que executa a operação de inserção levando em consideração todos os casos discutidos acima pode ser implementada como, encontra-se no site dedicado a este livro na internet.

Remoção

Da **Figura 10–30** à **Figura 10–33**, são ilustradas todas as possíveis situações de remoção de nós numa lista duplamente encadeada circular.

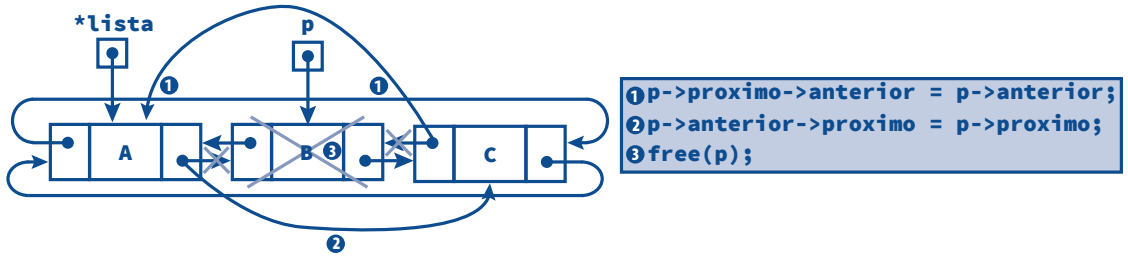


Figura 10–30: Remoção de Nó Interno em Lista Duplamente Encadeada Circular

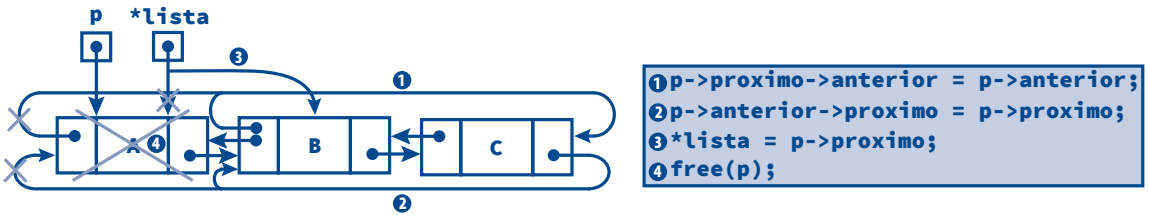


Figura 10–31: Remoção no Início de uma Lista Duplamente Encadeada Circular

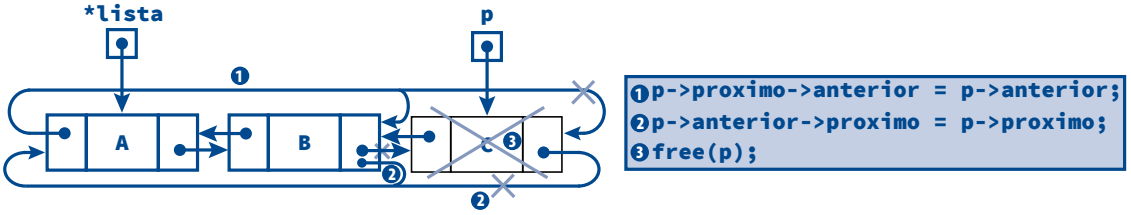


Figura 10–32: Remoção no Final de uma Lista Duplamente Encadeada Circular

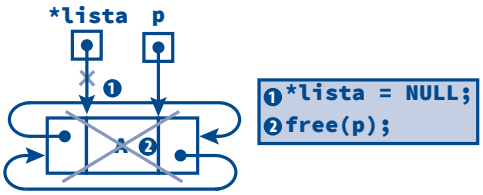


FIGURA 10–33: REMOÇÃO DO ÚNICO NÓ DE UMA LISTA DUPLAMENTE ENCADEADA CIRCULAR

Examinando-se as figuras que ilustram remoção de nós em listas duplamente encadeadas circulares, podem-se tirar as seguintes conclusões:

- A remoção de nós de uma lista circular duplamente encadeada requer apenas um ponteiro auxiliar (o ponteiro `p` nas referidas figuras).
- Remoção no início da lista constitui o único caso a ser tratado separadamente.

A função `RemoveListaDEC()` que implementa o que foi exposto acima sobre remoção em lista duplamente encadeada circular é encontrada no site dedicado a este livro.

Busca

Busca em listas duplamente encadeadas circulares é igual à busca em lista simplesmente encadeada circular (v. **Seção 10.4.2**).

Destruição

A função `DestroiListaDEC()`, responsável pela destruição de listas duplamente encadeadas circulares, pode ser encontrada no site dedicado ao livro.

Acesso Sequencial

Acesso sequencial em lista duplamente encadeada circular é igual a acesso sequencial em lista simplesmente encadeada circular.

Acesso Sequencial Invertido

A única diferença entre a função `AnteriorListaDEC()`, definida a seguir, e a função `AnteriorListaDE()` apresentada na **Seção 10.4.1**, é que a função a seguir é mais eficiente, pois ela não requer que o final da lista seja atingido antes de acessar os elementos do final para o início da lista. Precisamente, a função apresentada a seguir tem custo $\theta(1)$, enquanto aquela funcionalmente equivalente para listas duplamente encadeadas lineares tem custo $\theta(n)$.

```
tConteudo *AnteriorListaDEC(tListaDEC lista)
{
   static tNoListaDEC *noAnterior = NULL;

   if (lista) {
      if (!noAnterior) {
         noAnterior = lista->anterior;
      } else {
         noAnterior = noAnterior->anterior;
         if (noAnterior == lista->anterior)
            noAnterior = NULL;
      }
      return noAnterior ? &noAnterior->conteudo : NULL;
   } else
      return NULL;
}
```

10.4.4 Lista Encadeada com Cabeça

Lista encadeada vazia é um caso que frequentemente deve ser tratado separadamente em operações de inserção e remoção. A utilização de um **nó cabeça** numa lista elimina essa necessidade, ele estará sempre presente, mesmo quando a lista está vazia. Seu campo de informação pode conter um valor especial (p. ex., o comprimento da lista) ou ser indefinido.

A **Figura 10–34 (a)** mostra esquematicamente uma lista duplamente encadeada circular com cabeça vazia, ao passo que a **Figura 10–34 (b)** exibe uma lista duplamente encadeada circular com cabeça contendo três elementos. Nos dois casos, o nó cabeça distingue-se dos demais por meio do símbolo # que aparece em seu campo de conteúdo.

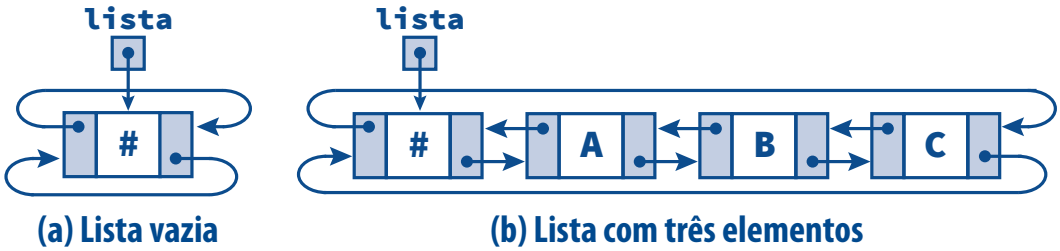


Figura 10–34: Listas Duplamente Encadeadas Circulares com Cabeça

Uma lista duplamente encadeada circular com cabeça utiliza definições de tipos semelhantes àquelas utilizadas para listas duplamente encadeadas sem cabeça (circulares ou não), conforme se mostra abaixo.

```
typedef struct rotNoLDECCab {
           struct rotNoLDECCab *anterior;
           tConteudo            conteudo;
           struct rotNoLDECCab *proximo;
        } tNoLDECCab, *tListaDECCab;
```

Quando se implementam operações sobre listas encadeadas com cabeça, deve-se tomar a devida precaução para não considerar a cabeça como um elemento integrante da lista. Por exemplo, uma lista encadeada com cabeça contendo apenas o nó cabeça possui comprimento igual a `0`, e não `1`, pois a cabeça não deve ser incluída nessa contagem.

Iniciação

Uma função que inicia uma lista encadeada com cabeça precisa criar um nó que será a cabeça da lista, como faz a função `CriaListaDECCab()` a seguir.

```
tListaDECCab CriaListaDECCab(void)
{
   tListaDECCab L = malloc(sizeof(tNoDECCab));

   ASSEGURA(L, "Impossivel alocar a cabeca da lista");

     /* Numa lista vazia, a cabeça aponta para si própria */
   L->anterior = L;
   L->proximo = L;

   return L;
}
```

Comprimento

Ao calcular o comprimento de uma lista encadeada com cabeça deve-se tomar a precaução de não contar a cabeça como um elemento da lista, como faz a função `ComprimentoListaDECCab()` apresentada a seguir.

```
int ComprimentoListaDECCab(tListaDECCab lista)
{
   int          tamanho = 0; /* Armazenará o tamanho da lista */
   tListaDECCab inicio = lista; /* Guarda a cabeça da lista */

      /* Faz o ponteiro 'lista' apontar para o */
      /* primeiro nó da lista (se ele existir) */
   lista = lista->proximo;

      /* Acessa cada nó da lista e conta  */
      /* quantos nós são acessados.       */
    while (lista != inicio) {
      lista = lista->proximo; /* Passa para o próximo nó */
      ++tamanho; /* Mais um nó foi encontrado */
   }

   return tamanho;
}
```

Inserção

A função `InsereEmOrdemLDECCab()`, que insere um nó numa lista duplamente encadeada circular ordenada com cabeça, encontra-se no site dedicado a este livro na internet. As figuras apresentadas na **Seção 10.4.3** podem ser utilizadas para facilitar o entendimento do raciocínio que norteia essa função.

Na função `InsereEmOrdemLDECCab()`, o conteúdo do nó a ser inserido é armazenado no nó cabeça. Desse modo, garante-se que um nó cujo conteúdo seja maior do que ou igual ao conteúdo do novo nó será sempre encontrado, de modo que o laço **while** que procura a posição de inserção avalia apenas uma condição, ao invés de duas condições avaliadas em situações semelhantes: se o item ainda não fora encontrado e se o final da lista ainda não fora atingido. Em resumo, essa técnica de busca consiste em colocar uma cópia do item a ser procurado na cabeça da lista e começar a busca no nó imediatamente seguindo a cabeça. A busca será, então, bem-sucedida se o item for encontrado antes do final (cabeça) da lista e malsucedida se o item for encontrado no final da lista.

É importante notar ainda que não há caso especial de inserção numa lista duplamente encadeada circular com cabeça.

Remoção

A função `RemoveListaDECCab()`, que efetua remoção em lista duplamente encadeada circular com cabeça, encontra-se no site do livro. Note que não há caso especial de remoção numa lista duplamente encadeada circular com cabeça.

Busca

A função `BuscaListaDECCab()`, que efetua busca em lista duplamente encadeada circular com cabeça, pode ser encontrada no site dedicado a este livro na internet.

Destruição

A função `DestroiListaDECCab()` libera o espaço ocupado pelos nós de uma lista duplamente encadeada circular com cabeça, tornando-a vazia e sem cabeça. O código dessa função também se encontra no site do livro.

Acessos Sequenciais Direto e Invertido

No site dedicado a este livro, encontra-se funções que implementam acesso sequencial e acesso sequencial invertido para listas duplamente encadeadas circulares com cabeça.

10.5 Pilhas e Filas Encadeadas

Pilhas e filas são tipos específicos de listas e, sendo assim, podem ser implementadas em forma de listas encadeadas. Esta seção mostra como os conceitos de pilha e fila estudados no **Capítulo 8** podem ser implementados como um tipo abstrato de dados usando listas encadeadas.

10.5.1 TAD Pilha 2

Como o acesso a uma pilha ocorre sempre no topo da mesma, necessita-se apenas de um ponteiro para executar as operações sobre pilhas e esse ponteiro aponta exatamente para o início da lista. Assim o topo da pilha é representado por um ponteiro para o início da lista, como mostra a **Figura 10–35** que representa esquematicamente uma pilha simplesmente encadeada com três elementos.

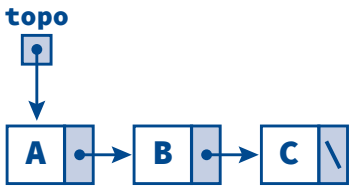


Figura 10–35: Pilha Simplesmente Encadeada

Arquivo de Cabeçalho

O arquivo de cabeçalho do TAD proposto é apresentado a seguir. Esse arquivo contém a definição do tipo de item a ser armazenado numa pilha desse tipo e uma definição incompleta do tipo. Para completar, o arquivo de cabeçalho contém as alusões das funções que representam as operações sobre pilhas.

```
/****************** Definições de Tipos ******************/

typedef  char tItemPilha;
typedef struct rotPilhaSE *tPilhaSE;

/****************** Alusões de Funções ******************/

extern void CriaPilhaSE(tPilhaSE *p);
extern void DestroiPilhaSE(tPilhaSE *pilha);
extern int EstaVaziaPilhaSE(const tPilhaSE *p);
extern tItemPilha ElementoTopoSE(const tPilhaSE *p);
extern void EmpilhaSE(tPilhaSE *p, const tItemPilha *item);
extern tItemPilha DesempilhaSE(tPilhaSE *p);
```

É importante observar ainda que as respectivas funções que implementam as operações sobre pilhas possuem os mesmos protótipos nas duas implementações apresentadas aqui. Isto significa que um programa pode utilizar qualquer dessas implementações sem que precise sofrer qualquer alteração.

Arquivo de Implementação

A seguir, serão discutidos os componentes do arquivo de implementação do TAD em discussão que acrescentem algum conhecimento novo com relação ao que já foi visto com relação a pilhas, TADs e listas encadeadas.

Antes das definições das funções do referido arquivo de implementação, são definidas a constante simbólica `ACRESCIMO` e a variável `tamanhoArray`, que tem escopo de arquivo, conforme foi visto na **Seção 9.6.2**. O complemento do tipo `tPilhaSE` segue as definições desses componentes e é apresentada abaixo.

```
struct rotPilhaSE {
   tItemPilha         conteudo;
   struct rotPilhaSE *proximo;
};
```

```
typedef struct tItemPilha tNoPilhaSE;
```

A seguir, serão apresentadas as definições de funções que fazem parte do arquivo de implementação do TAD proposto.

```
void CriaPilhaSE(tPilhaSE *p)
{
      /* Torna nulo o ponteiro para o topo da pilha, que é o   */
      /* primeiro elemento de uma lista simplesmente encadeada */
   *p = NULL;
}

int EstaVaziaPilhaSE(const tPilhaSE *p)
{
   return !*p; /* A pilha está vazia quando o ponteiro que a representa é NULL */
}

tItemPilha ElementoTopoSE(const tPilhaSE *p)
{
      /* Se a pilha estiver vazia, aborta o programa */
   ASSEGURA(!EstaVaziaPilhaSE(p), "A pilha esta' vazia");
   return (*p)->conteudo;
}

void EmpilhaSE(tPilhaSE *p, const tItemPilha *item)
{
   tPilhaSE pNovoNo;

   pNovoNo = malloc(sizeof(struct rotPilhaSE)); /* Tenta alocar um nó da pilha */

      /* Se a alocação não foi possível aborta o programa */
   ASSEGURA(pNovoNo, "Nao houve alocacao");

   /* O topo da pilha é o primeiro elemento de uma lista encadeada. Logo empilhar */
   /* é o mesmo que inserir um elemento no início de uma lista encadeada.         */

   pNovoNo->conteudo = *item; /* Armazena o conteúdo do novo nó */

   pNovoNo->proximo = *p; /* Faz o novo nó apontar para o topo corrente da pilha */

   *p = pNovoNo; /* O novo nó passa a ser o topo */
}

tItemPilha DesempilhaSE(tPilhaSE *p)
{
   tPilhaSE   aux; /* Ponteiro que auxilia a operação */
   tItemPilha conteudo;

      /* Se a pilha estiver vazia, aborta o programa */
   ASSEGURA(!EstaVaziaPilhaSE(p), "A pilha esta' vazia");

   /* O topo da pilha é o primeiro elemento de uma lista encadeada. Logo desempilhar */
   /* é o mesmo que remover o elemento inicial de uma lista encadeada.              */

   conteudo = (*p)->conteudo; /* Armazena o conteúdo do elemento do topo */

   aux = *p; /* Faz 'aux' apontar para o topo da pilha */

      /* Faz com que o topo passe a ser o segundo elemento da lista */
   *p = (*p)->proximo;

   free(aux); /* Libera o nó que constituía o antigo topo */

   return conteudo; /* Retorna o valor do segundo parâmetro */
}
```

10.5.2 TAD Fila 2

Como os acessos numa fila podem ocorrer tanto em seu início quanto em seu final, é mais conveniente e eficiente (embora não necessário) utilizar dois ponteiros para executar as operações sobre elas. Um destes ponteiros deve apontar para o início da lista, considerado a frente da fila, e o outro ponteiro deve apontar para o último elemento da lista, considerado o fundo da fila. A **Figura 10–36** apresenta, esquematicamente, uma fila simplesmente encadeada com três elementos.

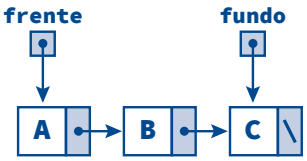


FIGURA 10–36: FILA SIMPLESMENTE ENCADEADA

Arquivo de Cabeçalho

Além das alusões às funções que representam as operações sobre filas, o arquivo de cabeçalho do TAD contém a definição do tipo de item a ser armazenado em filas desse tipo e uma definição incompleta do tipo.

```
/****************** Definições de Tipos ******************/

typedef  char tItemFila;

typedef struct rotFilaSE *tFilaSE;

/******************* Alusões de Funções ******************/

extern void CriaFilaSE(tFilaSE *f);
extern void DestroiFilaSE(tFilaSE *f);
extern int EstaVaziaFilaSE(const tFilaSE *f);
extern tItemFila ElementoFrenteSE(const tFilaSE *f);
extern void EnfileiraSE(tFilaSE *f, const tItemFila *item);
extern tItemFila DesenfileiraSE(tFilaSE *f);
```

Observe que essas alusões apresentam exatamente os mesmos protótipos das funções discutidas na **Seção 8.2** e na **Seção 9.7.2**. Desse modo, um programa-cliente que utilize essas funções não precisa ser alterado quando se muda a implementação de fila.

Arquivo de Implementação

O tipo `tNoListaSE` definido a seguir é o tipo dos nós da lista simplesmente encadeada que será utilizada para armazenar os itens de uma fila. Além desse tipo, também é definido o tipo `tListaSE`, que é o tipo de lista encadeada discutido na **Seção 10.2**. A vantagem de incluir essa última definição de tipo é que se pode usar qualquer função definida para esse tipo, como a função `DestroiListaSE()` (v. adiante). As referidas definições de tipo são apresentadas a seguir.

```
typedef  struct rotNoLSE {
            tItemFila        conteudo;
            struct rotNoLSE *proximo;
         } tNoListaSE, *tListaSE;
```

Uma representação conveniente para filas encadeadas é uma estrutura que incorpore os dois apontadores ilustrados na **Figura 10–36**. É essa a ideia usada no complemento do tipo `tFilaSE` apresentado a seguir.

```
struct rotFilaSE {
   tNoListaSE *frente;
   tNoListaSE *fundo;
};
```

As definições de funções do presente TAD serão vistas em seguida.

```
void CriaFilaSE(tFilaSE *f)
{
   ASSEGURA(*f = malloc(sizeof(struct rotFilaSE)), "Impossivel alocar fila");

   (*f)->frente = (*f)->fundo = NULL;
}

void DestroiFilaSE(tFilaSE *fila)
{
      /* Se a fila não estiver vazia, libera o espaço */
      /* ocupado pelo array que contém os elementos    */
   if (!EstaVaziaFilaSE(fila))
      DestroiListaSE(&(*fila)->frente);

   free((*fila)); /* Libera a estrutura que representa a fila */
}

int EstaVaziaFilaSE(const tFilaSE *f)
{
   return !(*f)->frente;
}

tItemFila ElementoFrenteSE(const tFilaSE *f)
{
   ASSEGURA(!EstaVaziaFilaSE(f), "Erro: Fila vazia");

   return (*f)->frente->conteudo;
}

void EnfileiraSE(tFilaSE *f, const tItemFila *item)
{
   tNoListaSE *aux;

      /* Tenta alocar um novo nó e causa aborto se isto não for possível */
   ASSEGURA( aux = malloc(sizeof(tNoListaSE)), "Erro de alocação de memória." );

   aux->conteudo = *item; /* preenche o conteúdo do nó */

   aux->proximo = NULL; /* Este será o último elemento da fila */

         /* Se a fila não estiver vazia, o último elemento atual deve */
         /* apontar para o novo elemento. Se a fila estiver vazia, a  */
         /* frente também deve apontar para esse elemento             */
   if (EstaVaziaFilaSE(f))
      (*f)->frente = aux;
   else
      (*f)->fundo->proximo = aux;

      /* O fundo deve apontar para o novo elemento */
   (*f)->fundo = aux;
}

tItemFila DesenfileiraSE(tFilaSE *f)
{
   tItemFila   elementoNaFrente;
   tNoListaSE *aux;

   ASSEGURA(!EstaVaziaFilaSE(f), "Erro: Fila vazia.");

   elementoNaFrente = (*f)->frente->conteudo; /* Guarda valor a ser retornado */

   aux = (*f)->frente; /* Guarda endereço do elemento a ser removido */
```

```
      /* Faz a frente apontar para o elemento seguinte */
   (*f)->frente = (*f)->frente->proximo;

   free(aux); /* Libera o elemento removido */

   return elementoNaFrente;
}
```

A função `DestroiFilaSE()`, apresentada acima, chama função `DestroiListaSE()` definida na **Seção 10.2** para liberar o espaço ocupado pela lista encadeada que armazena os itens da fila.

Note que, no módulo de implementação acima, não existe a função privada para testar se uma fila está cheia.

10.6 Análise de Operações sobre Listas Encadeadas

Informalmente, listas encadeadas apresentam vantagens e desvantagens com relação a listas indexadas conforme mostrado na **Tabela 10–1**.

Vantagens de Listas Encadeadas	Desvantagens de Listas Encadeadas
Uma lista encadeada nunca está cheia, a não ser que toda a memória alocada para o heap do programa seja exaurida	Listas encadeadas requerem espaço adicional para ponteiros
Inserção e remoção são mais simples, pois mover ponteiros é mais fácil do que elementos inteiros	Listas encadeadas não permitem acesso direto

Tabela 10–1: Comparando Listas Encadeadas e Arrays

A **Tabela 10–2** apresenta uma comparação entre os custos temporais de operações sobre listas indexadas e listas simplesmente encadeadas no pior caso[1].

Operação	Custo Temporal Quando a Lista É Implementada usando...	
	Array	Lista Simplesmente Encadeada Linear
Acréscimo ou remoção de elemento ao final	$\theta(1)$	$\theta(n)$
Inserção de elemento no início	$\theta(n)$	$\theta(1)$
Inserção de elemento no interior	$\theta(n)$	$\theta(n)$
Busca sequencial	$\theta(n)$	$\theta(n)$
Busca binária em lista ordenada	$\theta(\log n)$	Não faz sentido
Acesso sequencial	$\theta(n)$	$\theta(n)$

Tabela 10–2: Custos Temporais de Operações sobre Listas Encadeadas e Listas Indexadas

O custo espacial de qualquer implementação de lista encadeada é $\theta(n)$ devido uso de um ou dois ponteiros adicionais (i.e., além do conteúdo efetivo) em cada nó.

10.7 Exemplos de Programação

10.7.1 Invertendo uma Lista Simplesmente Encadeada

Preâmbulo: Inverter uma lista simplesmente encadeada significa fazer com que, após essa operação, o último nó torne-se o primeiro nó, o penúltimo nó torne-se o segundo nó e assim por diante, até que o

[1] Se a lista for implementada usando array dinâmico, o custo temporal de acréscimo ou remoção é $\theta(n)$.

primeiro nó passe a ser o último nó. Essa operação é útil, por exemplo, quando se tem uma lista encadeada ordenada em ordem crescente de algum campo de seus nós e, partir de um dado instante, deseja-se tê-la em ordem decrescente do mesmo campo ou vice-versa.

Problema: Escreva uma função que inverte uma lista simplesmente encadeada.

Solução: A função `InverteListaSE()`, apresentada adiante, implementa, iterativamente, a operação de inversão de uma lista encadeada. Essa função requer o uso de três ponteiros auxiliares, denominados `noCorrente`, `noAnterior` e `noSeguinte` e independe do tipo de conteúdo efetivo de cada nó. Essa função recebe como parâmetros um ponteiro para a lista encadeada que será invertida e retorna o endereço do primeiro nó da lista invertida.

```
tListaSE InverteListaSE(tListaSE *lista)
{
      /* Cada ponteiro a seguir aponta para o nó... */
   tListaSE noAnterior = NULL,       /* ...anterior */
            noCorrente = *lista,     /* ...corrente */
            noSeguinte = NULL;       /* ...seguinte */

      /* Ver ilustrações dos passos nas figuras */
   while (noCorrente) {
      noSeguinte = noCorrente->proximo; /* Instrução 1 */
      noCorrente->proximo = noAnterior; /* Instrução 2 */
      noAnterior = noCorrente;          /* Instrução 3 */
      noCorrente = noSeguinte;          /* Instrução 4 */
   }

      /* Neste instante, o ponteiro 'noAnterior' aponta para o início  */
      /* da lista invertida. Portanto atribui-se seu valor ao ponteiro */
      /* que representa a lista e retorna-se o resultado.              */
   return *lista = noAnterior;
}
```

Para entender melhor o funcionamento da função `InverteListaSE()`, acompanhe a execução dessa função ilustrada na **Figura 10–37** desde o início da execução da função até o final da primeira iteração do corpo do laço **while**. O parâmetro `lista` não aparece nessa figura, porque ele é irrelevante para o entendimento do funcionamento da função, mas, em todas as situações ilustradas, ele continua apontado para o primeiro nó (i.e., o nó mais à esquerda nas figuras). Aliás, esse parâmetro é alterado apenas na última instrução da função `InverteListaSE()`.

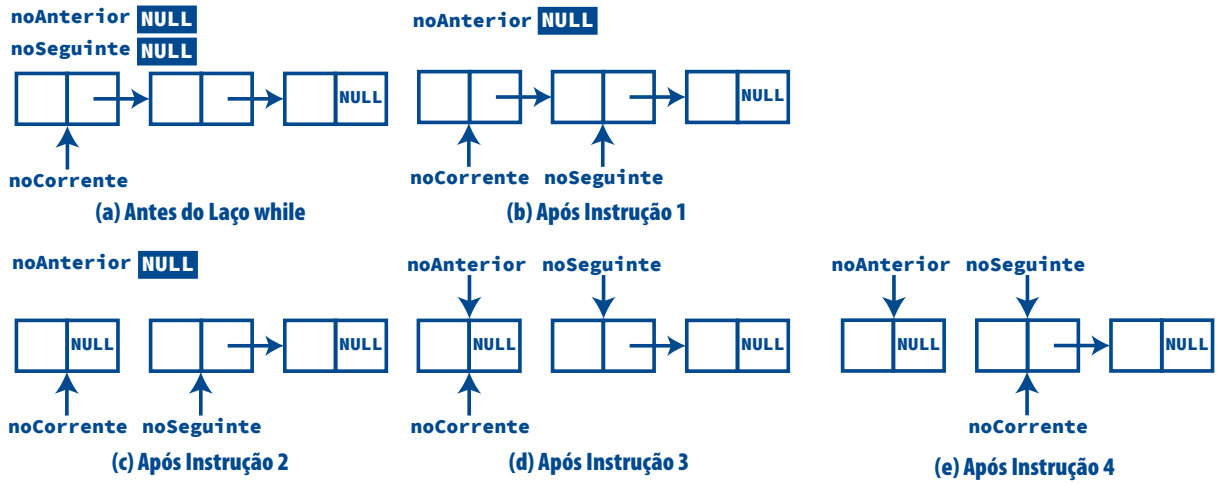


FIGURA 10–37: INVERSÃO DE UMA LISTA ENCADEADA: ANTES DO LAÇO WHILE

Exercício: Desenhe figuras semelhantes às apresentadas nesta seção que mostrem as alterações dos ponteiros envolvidos no processo de inversão na segunda iteração do laço **while** da função `InverteListaSE()`. Após resolver esse exercício, provavelmente, não lhe restarão dúvidas quanto ao funcionamento dessa função.

10.7.2 Números Felizes

Preâmbulo: Considere uma sequência numérica na qual o primeiro termo é um número inteiro maior do que *1* e cada termo subsequente consiste na soma dos quadrados dos dígitos do termo que o antecede. Então, o primeiro termo dessa sequência é um **número feliz** quando seu último termo for igual a *1*. Por exemplo, *13* é um número feliz, como mostra a **Figura 10–38**.

$$1^2 + 3^2 = 10$$
$$1^2 + 0^2 = 1$$

Figura 10–38: Um Número Feliz

Quando o primeiro termo de uma sequência obtida conforme a descrição acima não é um número feliz, a sequência torna-se cíclica em algum ponto. Por exemplo, *3* não é um número feliz, como mostra a **Figura 10–39**.

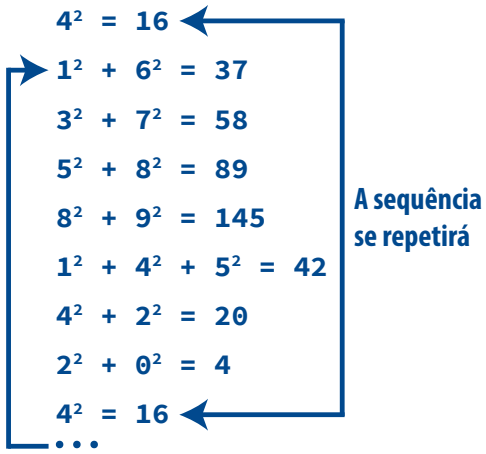


Figura 10–39: Um Número Infeliz

Problema: (a) Escreva uma função que verifica se um número inteiro maior do que `1` é feliz. (b) Escreva um programa que usa a função solicitada no item (a) para verificar se números introduzidos via teclado são felizes.

Solução de (a): A função `EhFeliz()`, apresentada a seguir, verifica se seu parâmetro é um número feliz retornando `1`, se esse for o caso, ou `0`, em caso contrário. Essa função usa uma lista encadeada do tipo `tListaSE` definido na **Seção 10.2**, bem como as funções `IniciaListaSE()`, `InsereListaSE()`, `BuscaListaSE()` e `DestroiListaSE()`, que também foram definidas nessa mesma seção.

```
int EhFeliz(int num)
{
   tListaSE lista; /* Lista encadeada que armazena os termos */
                   /* da sequência de números gerada        */

   IniciaListaSE(&lista);

   if (num <= 1) /* Números felizes devem ser maiores do que 1 */
      return 0; /* Esse número é um infeliz */
```

```
      /* O laço termina quando for concluído e o número é feliz ou não */
   while (1) {
         /* Insere o termo corrente da sequência numa lista encadeada */
      InsereListaSE(&lista, num);

         /* Obtém o termo seguinte da sequência somando */
         /* os quadrados dos dígitos do termo corrente  */
      num = SomaDigitosQuadrados(num);

         /* Se o novo termo for 1, o número é feliz. Se já apareceu na sequência, */
         /* ele não é feliz. Em outro caso, continua gerando novos termos.        */
      if (num == 1) { /* O número é feliz */
            /* Libera o espaço ocupado pela lista antes de retornar */
         DestroiListaSE(&lista);

         return 1; /* Felicíssimo! */
      } else if ( BuscaListaSE(lista, num) ) {
            /* O novo termo já foi gerado e, portanto, o número não é */
            /* feliz. Libera o espaço ocupado pela lista e retorna.   */
         DestroiListaSE(&lista);
         return 0; /* O coitado é infeliz */
      }
   }

      /* O fluxo de execução não deve chegar até aqui */
   printf("\nEste programa nao e' feliz\n");

   return 0;
}
```

Observações sobre função `EhFeliz()`:

- A função `EhFeliz()` utiliza um laço **while** no corpo do qual os termos da sequência definida no preâmbulo são armazenados numa lista encadeada por meio da função `InsereListaSE()` (v. **Seção 10.2**).
- O primeiro termo da sequência armazenado na referida lista encadeada é o número recebido como parâmetro e os demais termos são gerados pela função `SomaDigitosQuadrados()` (v. adiante), que calcula o próximo termo como a soma dos dígitos do termo corrente ao quadrado.
- O laço **while** encerra quando o mais novo termo da sequência é: **(1)** igual a `1` ou **(2)** um valor que já foi gerado. No primeiro caso, o número é considerado feliz, enquanto, no segundo caso, conclui-se que ele não é feliz.
- Para verificar se um termo da sequência já foi gerado, a função `EhFeliz()` chama a função `BuscaListaSE()` (v. **Seção 10.2**).
- Antes de encerrar, a função sob discussão chama a função `DestroiLista()`, definida na **Seção 10.2**, para liberar o espaço ocupado pela lista encadeada mencionada.

A definição da função `SomaDigitosQuadrados()`, citada acima, será apresentada a seguir. Essa função retorna a soma dos quadrados dos dígitos do número inteiro positivo recebido como parâmetro.

```
int SomaDigitosQuadrados(int numero)
{
   int digito ,/* Armazenará cada dígito do número */
       soma = 0; /* Armazenará a soma dos quadrados dos dígitos */

      /* Verifica se o número é válido */
   ASSEGURA(numero >= 0, "Numero invalido");
```

```
      /* O laço encerra quando o número cujos dígitos */
      /* serão somados assume zero como valor         */
   do {
      digito = numero % 10; /* Obtém o próximo dígito do número */

         /* Atualiza o valor do número que ainda falta levar em conta */
      numero = numero / 10;

      soma = soma + digito*digito; /* Atualiza a soma dos quadrados dos dígitos */
   } while (numero);

   return soma;
}
```

Solução de (b): A função **main()** definida abaixo testa se números introduzidos via teclado são felizes.

```
int main(void)
{
   int numero;

   printf( "\n\t>>> Este programa determina se os numeros "
           "introduzidos\n\t>>> sao felizes ou nao. Um valor "
           "menor do que 2 encerra\n\t>>> o programa.\n" );
   while (1) {
      printf("\n\t>>> Digite o numero que sera' checado: ");

         /* Se o usuário digitar um valor menor do que 2, encerra o laço */
      if ((numero = LeInteiro()) < 2)
         break;

         /* Apresenta o resultado do exame do número */
      printf( "\n\t>>> O numero %d %se' feliz\n", numero,
              EhFeliz(numero) ? "" : "nao " );
   }

   printf( "\n\t>>> Obrigado por usar este programa. Felicidades!\n" );

   return 0;
}
```

O complemento do programa é relativamente trivial e pode ser obtido no site dedicado ao livro na internet.

10.7.3 Representação de Polinômios Usando Listas Encadeadas

Preâmbulo: Se você nunca estudou polinômios ou não recorda o assunto, consulte o **Apêndice B**.

Problema: Apresente uma representação de polinômios utilizando listas simplesmente encadeadas e implemente as seguintes operações:

(a) Criação de um polinômio
(b) Destruição de um polinômio
(c) Valor numérico de um polinômio (quando aplicado a um valor real)
(d) Determinação do grau de um polinômio
(e) Soma de polinômios
(f) Subtração de polinômios
(g) Exibição na tela de um polinômio

Solução de (a): Os polinômios serão representados por listas simplesmente encadeadas ordenadas em ordem decrescente do expoente de cada termo, como o exemplo apresentado na **Figura 10–40**.

$$P(x) = x^4 - 2x^3 + x - 1$$

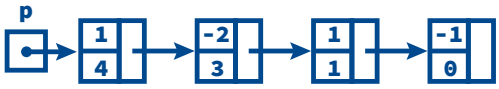


Figura 10–40: Polinômio Representado como Lista Encadeada

Os seguintes tipos serão utilizados nesta implementação:

```
  /* Tipo de um termo de polinômio */
typedef struct {
          int coef; /* Coeficiente do termo */
          int exp;  /* Expoente do termo    */
       } tTermo;

   /* Tipo de um polinômio */
typedef struct rotNoPoli {
          tTermo            termo;   /* Termo         */
          struct rotNoPoli *proximo; /* Próximo termo */
       } tNoPoli, *tPoli;
```

A criação de um polinômio consiste em ler os valores dos seus termos e representar o polinômio correspondente. A corrente implementação apresenta duas limitações:

[1] Os coeficientes são valores inteiros e não reais, como poderiam ser. Essa não é uma limitação séria e procede-se assim apenas por uma questão de elegância. Não é assim que a maioria dos livros de matemática elementar apresenta polinômios?

[2] A introdução dos termos de um polinômio encerra quando o usuário digita um valor de coeficiente igual a zero. Isso significa que o usuário não pode introduzir um termo nulo, o que não é grave na prática, a não ser que o usuário deseje introduzir um polinômio nulo.

A função `LePolinomio()` lê termos de um polinômio introduzidos via teclado e representa-os utilizando uma lista simplesmente encadeada. Ela retorna o polinômio lido e representado por meio dessa lista.

```
tPoli LePolinomio(void)
{
   tPoli  umPolinomio = NULL; /* Ponteiro para a lista que armazenará o polinômio */
   tTermo umTermo; /* Armazenará um termo do polinômio */

      /* O laço a seguir encerra quando o usuário */
      /* digitar 0 como valor de coeficiente      */
   while (1) {
         /* O coeficiente deveria ser real, mas, como fazem os livros */
         /* de matemática, por comodidade, ele será inteiro           */
      printf("\n\tCoeficiente: ");
      umTermo.coef = LeInteiro();

         /* Se o coeficiente for zero, encerra o laço */
      if (!umTermo.coef)
         break;

         /* O expoente deve ser um inteiro não negativo */
      printf("\tExpoente: ");
      umTermo.exp = LeNatural();

         /* Tenta inserir o novo termo na lista que representa o polinômio. */
         /* Se a função InsereTermo() retornar um valor diferente de zero,  */
         /* o termo já existe e o programa será abortado.                   */
      ASSEGURA(!InsereTermo(&umPolinomio, &umTermo), "\nEste termos ja' existe");
   }
```

```
      /* Se 'umPolinomio' for NULL, nenhum termo foi inserido */
      /* no polinômio. Nesse caso, aborta o programa.         */
   ASSEGURA(umPolinomio, "Polinomio nao possui nenhum termo");

   return umPolinomio; /* Done */
}
```

A função `InsereTermo()` é chamada por `LePolinomio()` para inserir um novo termo (segundo parâmetro) num polinômio representado usando uma lista simplesmente encadeada (primeiro parâmetro). Ela retorna `0`, se não ocorrer erro e `1`, em caso contrário.

```
int InsereTermo(tPoli *poli, const tTermo *oTermo)
{
   tNoPoli *pNovoNo;  /* Apontará para o novo nó alocado */
   tPoli    p = *poli, /* p aponta para o nó corrente     */
            q = NULL;  /* q aponta para o nó anterior a p */

   ASSEGURA(pNovoNo = malloc(sizeof(*pNovoNo)), "Impossivel alocar um termo");

      /* Armazena no novo nó o termo recebido como parâmetro */
   pNovoNo->termo = *oTermo;

     /* Se a lista estiver vazia, basta adicionar o nó ao início da lista e retornar */
   if (!*poli) {
      *poli = pNovoNo; /* O início da lista apontará para novo nó */
      (*poli)->proximo = NULL; /* Novo nó é o único da lista */
      return 0; /* Serviço completo */
   }

         /* A lista tem pelo menos um nó. Nesse caso, a posição de  */
         /* inserção é aquela do primeiro nó cujo campo 'exp' seja  */
         /* menor do que aquele do nó a ser inserido. Se tal nó não */
         /* for encontrado, o novo nó será o último da lista.       */

      /* Procura o local de inserção */
   while (p && oTermo->exp < p->termo.exp) {
      q = p;    /* q passa a apontar para o nó corrente */
      p = p->proximo; /* p passa a apontar para o próximo nó */
   }

   if (p && oTermo->exp == p->termo.exp) {/* Verifica se o termo já existe */
      free(pNovoNo);
      return 1; /* Termo já existe */
   }

      /* Neste ponto, p aponta para o local da inserção e q aponta para o nó     */
      /* imediatamente antes desse local. Se p for NULL, o novo nó será o        */
      /* último da lista, mas isso não é um caso especial a ser tratado à parte. */

      /* O novo nó apontará para o nó apontado por p */
   pNovoNo->proximo = p;

      /* Verifica se p aponta para o primeiro nó da lista pois */
      /* inserção no início deve ser tratada separadamente     */
   if (p == *poli) { /* Inserção será no início da lista */
      *poli = pNovoNo;/* Inserção no início é especial */
   } else {
         /* O nó anterior àquele apontado por p apontará para o novo nó */
      q->proximo = pNovoNo;
   }
   return 0; /* Serviço completo */
}
```

Solução de (b): A destruição de um polinômio representado de acordo com o esquema sob discussão utiliza uma função semelhante àquela apresentada na **Seção 10.3**.

Solução de (c): A função `ValorNumerico()` apresentada a seguir calcula o valor numérico de um polinômio quando aplicada a um valor real.

```
double ValorNumerico(const tNoPoli *poli, double a)
{
   double resultado = 0.0;

   while (poli) {
      resultado += poli->termo.coef*pow(a, poli->termo.exp);
      poli = poli->proximo;
   }

   return resultado;
}
```

A função `ValorNumerico()` usa a função **pow**() declarada no cabeçalho `<math.h>`.

Solução de (d): A determinação do grau de um polinômio é a mais simples de todas as operações requisitadas, pois, devido ao modo como os polinômios são representados, o grau de um polinômio é obtido simplesmente acessando-se o expoente do primeiro termo do polinômio, como mostra a função `Grau()` a seguir:

```
int Grau(const tNoPoli *poli)
{
      /* Como os polinômios são armazenados em ordem decrescente de expoentes, */
      /* o grau de um polinômio corresponde ao expoente do primeiro termo      */
   return poli->termo.exp;
}
```

Solução de (e): A soma de polinômios representados conforme discutido aqui requer um pouco mais de atenção para entender como ela é efetuada. Portanto, antes de tentar entender a função `SomaPolinomios()` apresentada adiante, examine cuidadosamente a **Figura 10–41** que apresenta um exemplo de soma de dois polinômios representados de acordo com esse esquema.

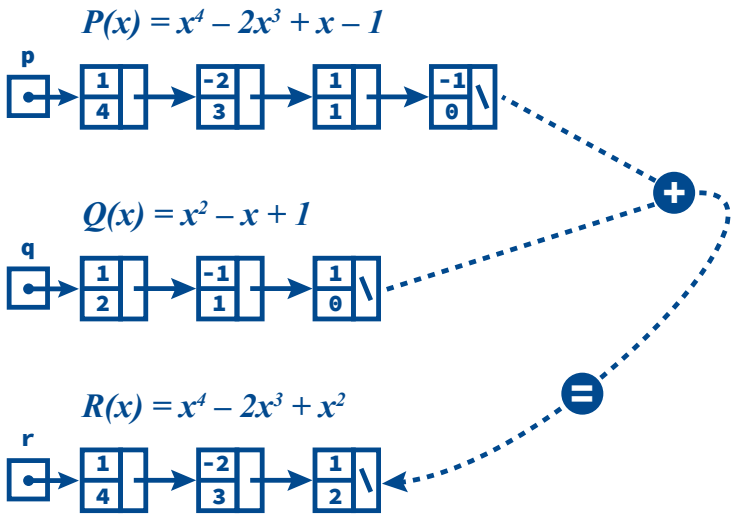


FIGURA 10–41: SOMA DE POLINÔMIOS REPRESENTADOS COMO LISTAS ENCADEADAS

```
tPoli SomaPolinomios(tNoPoli *poli1, tNoPoli *poli2)
{
   tPoli resultado = NULL; /* Ponteiro para a lista qu  armazenará o resultado */
   tTermo novoTermo; /* Armazenará um termo resultante da soma */
                     /* de dois termos dos polinômios somados  */

      /* Acessa cada termo de cada polinômio e encerra o laço quando */
      /* todos os termos de um dos polinômios tiverem sido acessados */
   while (poli1 && poli2) {
         /* Se os termos correntes tiverem o mesmo expoente, o coeficiente */
         /* do novo termo será a soma dos coeficentes desses termos        */
      if (poli1->termo.exp == poli2->termo.exp) {
         novoTermo.coef = poli1->termo.coef + poli2->termo.coef;

            /* O expoente do novo termo é o mesmo    */
            /* expoente dos termos que foram somados */
         novoTermo.exp = poli1->termo.exp;

         poli1 = poli1->proximo;
         poli2 = poli2->proximo;
      } else if (poli1->termo.exp > poli2->termo.exp) {
            /* Armazena no resultado o termo do polinômio com maior  */
            /* expoente e passa para o próximo termo desse polinômio */
         novoTermo = poli1->termo;
         poli1 = poli1->proximo;
      } else {
            /* Armazena no resultado o termo do polinômio com maior  */
            /* expoente e passa para o próximo termo desse polinômio */
         novoTermo = poli2->termo;
         poli2 = poli2->proximo;
      }

         /* Armazena um novo termo na lista que representa o resultado */
         /* apenas quando o coeficiente for diferente de 0             */
      if (novoTermo.coef) {
            /* Insere o novo termo. O programa não deve ser abortado */
            /* aqui, a não ser que ele esteja incorreto.             */
         ASSEGURA(!InsereTermo(&resultado, &novoTermo), "Ocorreu erro inesperado!");
      }
   }

      /* Acrescenta termos remanescentes do primeiro polinômio ao resultado */
   while (poli1) {
      ASSEGURA(!InsereTermo(&resultado,&poli1->termo),"Ocorreu um erro inesperado!");
      poli1 = poli1->proximo;
   }

      /* Acrescenta termos remanescentes do segundo polinômio ao resultado */
   while (poli2) {
      ASSEGURA( !InsereTermo(&resultado, &poli2->termo),
                "Ocorreu um erro inesperado!" );
      poli2 = poli2->proximo;
   }

      /* Se nenhum termo foi acrescentado à lista que representa */
      /* o resultado, o resultado só pode ser zero               */
   if (!resultado) {
      novoTermo.coef = novoTermo.exp = 0;
      ASSEGURA(!InsereTermo(&resultado, &novoTermo), "Ocorreu um erro inesperado!");
   }
```

```
   return resultado;
}
```

Solução de (f): Se você entendeu a função que realiza soma de polinômios, entender como se implementa a subtração de polinômios torna-se relativamente trivial, pois a implementação de uma função que realiza essa tarefa é semelhante àquela que realiza a soma. Mais precisamente, basta copiar o código da função `SomaPolinomios()` e trocar o sinal de soma na seguinte instrução dessa função:

```
novoTermo.coef = poli1->termo.coef + poli2->termo.coef;
```

pelo sinal de subtração, de modo a obter:

```
novoTermo.coef = poli1->termo.coef - poli2->termo.coef;
```

Solução de (g): A exibição na tela de um polinômio de modo amigável é um tanto trabalhosa, mas não envolve nenhum algoritmo sofisticado que merece comentários dentro do escopo deste livro. Essas funções serão apresentadas a seguir apenas por questão de complemento.

A função `ExibeTermo()` exibe um termo de um polinômio na tela. Seu primeiro parâmetro representa o termo, enquanto o segundo é a ordem do termo.

```
void ExibeTermo(const tTermo *termo, int ordem)
{
      /* Se o termo não for o primeiro, apresenta o sinal do termo */
   if (ordem > 1)
      if (termo->coef > 0)
         printf(" + ");
      else
         printf(" - ");

   if (termo->exp == 0) { /* Escreve o termo independente */
      printf("%d", ordem > 1 ? ABS(termo->coef) : termo->coef);
      return;
   }

   if (termo->exp == 1) { /* Escreve o termo linear */
      if (termo->coef == 1)
         putchar('x');
      else if (termo->coef == -1)
         printf("%sx", ordem > 1 ? "" : "-");
      else
         printf( "%dx", ordem > 1 ? ABS(termo->coef) : termo->coef );

      return;
   }

      /* Neste ponto, o expoente deve ser maior do 1 */
   ASSEGURA( termo->exp > 1, "Encontrado um expoente estranho" );

   if (termo->coef == 1) /* Escreve os demais termos */
      printf("x^%d", termo->exp);
   else if (termo->coef == -1)
      printf("%sx^%d", ordem > 1 ? "" : "-", termo->exp);
   else
      printf("%dx^%d", ordem > 1 ? ABS(termo->coef) : termo->coef, termo->exp);
}
```

A função `ExibePolinomio()` exibe o polinômio recebido como parâmetro na tela.

```
void ExibePolinomio(const tNoPoli *poli)
{
   int i = 1; /* Indica a ordem do termo */

   ASSEGURA(poli, "\nO polinomio está vazio");

   while (poli) { /* Exibe os termos */
      ExibeTermo(&poli->termo, i);
      ++i;
      poli = poli->proximo;
   }
}
```

O programa completo que processa polinômios por meio de listas encadeadas se encontra no site dedicado a este livro na internet.

10.7.4 A História de Josephus

Preâmbulo: O **problema** (ou a **permutação**) **de Josephus** é relatado pelo historiador romano Flavius Josephus. Segundo ele, durante a guerra judaico-romana de 67 AD, os romanos se apoderaram da cidade que ele comandava (à época ele lutava ao lado dos judeus; depois, ele mudou de lado, como alguns políticos que o leitor conhece...). Pois bem, ele e seus 40 companheiros conseguiram escapar do cerco romano, mas, ao final, terminaram encurralados numa caverna. Temendo serem capturados, os companheiros decidiram se matar. Entretanto, Josephus e um companheiro discordaram da proposta e propuseram uma alternativa. Essa proposta alternativa consistia em dispor todos eles em círculo e, então contá-los a partir de uma posição no círculo de três em três. Desse modo, cada terceiro companheiro contado assim seria morto até que existisse apenas um companheiro que então se suicidaria. Os companheiros acharam justa a proposta e a aceitaram. Josephus e seu amigo próximo escolheram, respectivamente, as posições *31* e *16* no círculo e terminaram escapando da morte. A **Figura 10–42** ilustra a proposta de Josephus, com seus companheiros numerados de *1* a *40*.

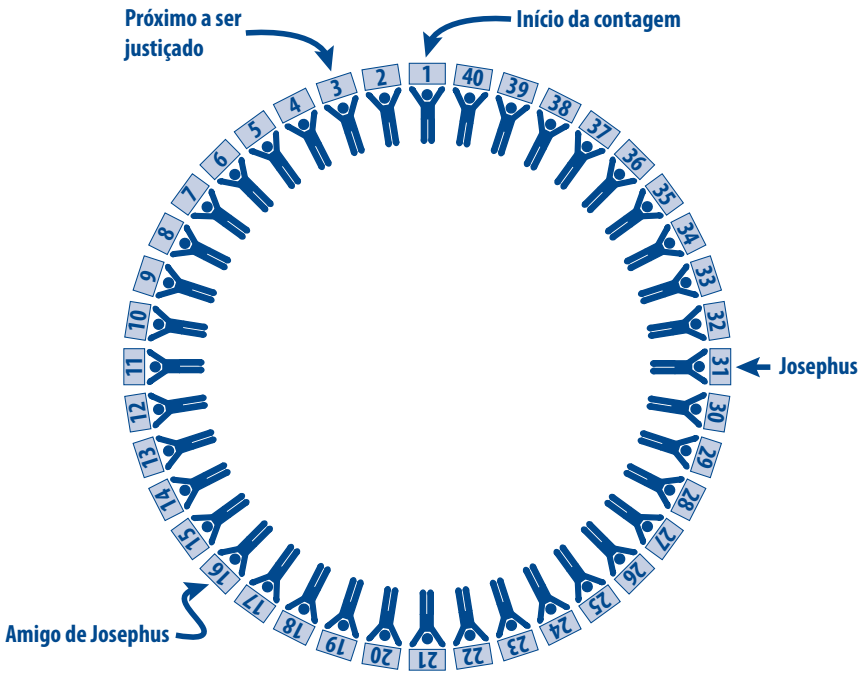


Figura 10–42: A Roda de Josephus

Problema: Em matemática e em programação, na história de Josephus, só sobra um companheiro. Escreva um programa em C que mostra a ordem na qual os companheiros de Josephus são *justiçados* e qual é aquele que sobra para contar a história. O programa deve solicitar ao usuário os nomes dos soldados que serão incluídos na fatídica lista e que ele informe o tamanho do salto fatal (i.e., o número de indivíduos que serão saltados antes que o próximo seja executado).

Solução: O programa a seguir implementa uma solução para o Problema de Josephus. Esse programa usa uma lista encadeada circular para armazenar Josephus e seus companheiros. A constante simbólica `MAX_NOME` e os tipos `tNoJosephus` e `tListaJosephus` são definidos no início do programa como:

```
#define MAX_NOME 25 /* Tamanho máximo de um nome de soldado de Josephus */

typedef struct rotNoJosephus {
          char                  nome[MAX_NOME];
          struct rotNoJosephus *proximo;
        } tNoJosephus, *tListaJosephus;
```

A função `CriaListaDeSoldados()` lê os nomes dos soldados de Josephus e cria uma lista circular simplesmente encadeada para contê-los.

```
void CriaListaDeSoldados(tListaJosephus *inicio)
{
   tNoJosephus *pAux, /* Ponteiro usado na alocação de nós */
               *ultimo; /* Apontará sempre para o último nó */
   int          opcao;  /* Opção escolhida pelo usuário */

   do {
      pAux = malloc(sizeof(tNoJosephus)); /* Aloca espaço para mais um nó da lista */

         /* Solicita o nome do próximo soldado a ser incluído na lista */
      printf( "\nDigite o nome do soldado (max = %d) >>> ", MAX_NOME - 1 );
      LeString(pAux->nome, MAX_NOME);

      pAux->proximo = NULL; /* Este será o último nó da lista */

      if (*inicio == NULL)
         *inicio = pAux; /* Este é o primeiro nó da lista */
      else
            /* O nó que antes era o último passa a apontar para o nó mais recente */
         ultimo->proximo = pAux;

      ultimo = pAux; /* O nó mais recentemente alocado agora é o último */

         /* Verifica se ainda há soldado a ser incluído */
      printf("\nDeseja acrescentar outro soldado (s/n)? ");
      opcao = LeOpcao("SsNn");
   } while (opcao == 'S' || opcao == 's');

      /* Torna a lista circular fazendo o último nó apontar para o primeiro nó */
   ultimo->proximo = *inicio;
}
```

A função `Sobrevivente()` elimina soldados da lista de Josephus até sobrar apenas um sobrevivente. Os parâmetros dessa função são:

- `inicio` (entrada e saída) — endereço de um ponteiro para o início da lista
- `salto` (entrada) — o número de elementos que serão saltados a cada rodada de justiçamentos

Essa função retorna o endereço do nome do elemento que se encontra na frente da lista ao final da chacina (i.e., o sobrevivente).

```
char *Sobrevivente(tListaJosephus *inicio, int salto)
{
   tNoJosephus *p, *q;
   int          i;

   q = p = *inicio;

      /* Remove nós da lista até que sobre apenas um nó */
   while (p->proximo != p) {
      for (i = 0; i < salto - 1; i++) {/* Elimina nós de salto em salto */
         q = p;
         p = p->proximo;
      }

      q->proximo = p->proximo; /* Remove da lista o elemento para o qual p aponta */

         /* Informa qual é o nome do elemento removido */
      printf("%s foi morto\n", p->nome);

      free(p); /* Libera o espaço ocupado pelo elemento removido */

      p = q->proximo;
   }
   *inicio = p; /* p aponta para o sobrevivente */

   return (p->nome); /* Retorna o endereço do nome do sobrevivente */
}
```

A seguir, será apresentada a função do programa proposto. Você encontrará o programa completo que resolve este problema no site dedicado ao livro na internet.

```
int main(void)
{
   tListaJosephus inicio = NULL;
   int            salto;
   char          *sobrevive;

      /* Apresenta o programa ao usuário */
   printf( "\n\t>>> Este programa resolve o Problema de Josephus\n" );

   CriaListaDeSoldados(&inicio); /* Cria a lista circular com os soldados */

      /* Apresenta os soldados na lista antes da chacina */
   printf("\n\n>>> Soldados na lista <<<\n");
   ExibeLista(inicio);

      /* Lê o tamanho do salto */
   printf("Qual e' o numero de soldados a serem saltados? ");
   salto = LeNatural();

   printf("\n>>> Lista de infelizes <<<\n\n");

      /* Apresenta os soldados à medida em que são mortos */
      /* e recebe o nome do soldado que sobrevive         */
   sobrevive = Sobrevivente(&inicio, salto);

   printf( "\n>>> O soldado sobrevivente e': %s\n", sobrevive );

      /* Neste ponto, a lista contém apenas um nó que  */
      /* não é mais necessário e, portanto, é liberado */
   free(inicio);

   return 0;
}
```

10.7.5 Números Inteiros de Larguras Ilimitadas

Problema: Implemente um tipo de dado que represente números inteiros de tamanhos arbitrários (inteiros ilimitados). Essa implementação deve disponibilizar as seguintes operações:

(a) Criação de um número inteiro ilimitado
(b) Destruição de um número inteiro ilimitado
(c) Comparação de números inteiros ilimitados
(d) Soma de números inteiros ilimitados
(e) Subtração de números inteiros ilimitados
(f) Exibição na tela de um número inteiro ilimitado

Solução: Os dígitos que constituem um número inteiro ilimitado serão implementados em listas circulares duplamente encadeadas. A razão para tal escolha é fato de esses dígitos precisarem ser acessados do dígito mais significativo para o dígito menos significativo e vice-versa. Os números em si serão armazenados em estruturas contendo três campos:

1. `sinal`, que representa o sinal do número
2. `valor`, que é um ponteiro para uma lista circular duplamente encadeada contendo os dígitos que compõem o número
3. `nDigitos`, que um campo que armazena o número de dígitos do número

Assim as seguintes definições de tipos são necessárias para a implementação de inteiros ilimitados:

```
typedef enum {MAIS, MENOS} tSinal; /* Tipo do sinal de um número */

  /* Tipo de nó e ponteiro para nó da lista encadeada */
   /* que armazena um número inteiro ilimitado         */
typedef struct no {
           char       digito;
           struct no *anterior, *proximo;
        } tDigitos, *tDigitosPtr;

   /* Tipo de variável que representa um número inteiro ilimitado */
typedef struct {
           tSinal      sinal;
           tDigitosPtr valor;
           long long   nDigitos;
        } tNumero, *tNumeroPtr;
```

A **Figura 10–43** mostra como o número `-1970` é representado do modo descrito.

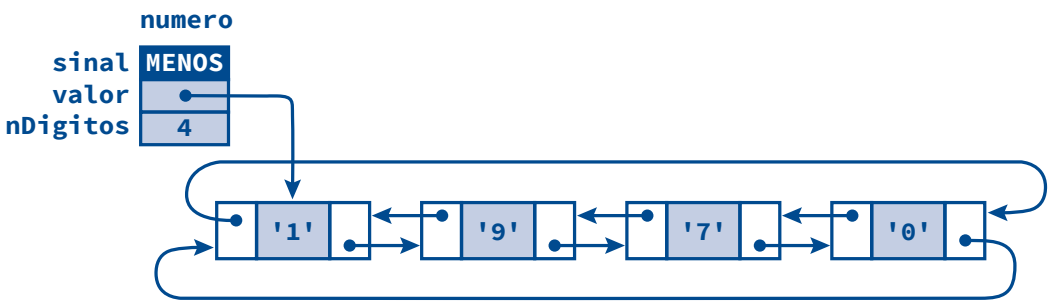


Figura 10–43: Um Número Inteiro de Largura Ilimitada

Solução de (a): A função `CriaInteiro()` apresentada a seguir, recebe um string como parâmetro e retorna o endereço de uma estrutura do tipo `tNumero` que representa o número inteiro cujos dígitos fazem parte do string.

```
tNumeroPtr CriaInteiro(const char *str)
{
   tNumero *ptrNumero;

   if (!*str)
      return NULL; /* O string é vazio */

     /* Tenta alocar espaço para a estrutura que contém as informações sobre o número */
   ASSEGURA(ptrNumero = malloc(sizeof(tNumero)), "Nao foi possivel alocar numero");

      /* A lista que contém os dígitos ainda está vazia */
   ptrNumero->valor = NULL;
   ptrNumero->nDigitos = 0;

      /* Salta espaços em branco iniciais */
   while (*str && isspace(*str))
      ++str;

      /* Obtém o sinal do número */
   if (*str == '-') {
      ptrNumero->sinal = MENOS;
      str++;
   } else if (*str == '+') {
      ptrNumero->sinal = MAIS;
      str++;
   } else
      ptrNumero->sinal = MAIS;

   while (*str && *str == '0') /* Salta zeros iniciais */
      ++str;

      /* Só foram encontrados zeros */
   if (!*str && str[-1] == '0') {
      str--;
      ptrNumero->sinal = MAIS;
   }

      /* Acessa cada caractere do string e armazena os dígitos na lista que */
      /* representará o número. Se for encontrado algum caractere que não   */
      /* é dígito, libera o espaço alocado e retorna NULL.                  */
   while (*str) {
         /* Verifica se o caractere corrente é dígito */
      if (isdigit(*str)) {
            /* Acrescenta novo nó contendo o dígito ao */
            /* final da lista que representa o número  */
         AcrescentaDigitoFim(ptrNumero, *str);

         ++str; /* Passa para o próximo caractere */
      } else { /* Encontrado um caractere que não é dígito */
         DestroiNumero(ptrNumero); /* Libera o espaço alocado até então */

         return NULL; /* Criação do número falhou */
      }
   }

   return ptrNumero;  /* Número foi criado com sucesso */
}
```

A função `CriaInteiro()` é relativamente fácil de entender seguindo os comentários dela. Essa função chama a função `AcrescentaDigitoFim()` para acrescentar os dígitos que constituem o número em construção.

Além disso, a função `AcrescentaDigitoFim()` incrementa o campo `nDigitos` do número ora sendo criado. Essa função é definida como:

```
void AcrescentaDigitoFim(tNumero *numero, int digito)
{
   tDigitos *ptrNoDigito;

      /* Tenta alocar o novo nó */
   ASSEGURA( ptrNoDigito = malloc(sizeof(tDigitos)),
             "Nao foi possivel alocar no' da lista" );

      /* Armazena o dígito no novo nó */
   ptrNoDigito->digito = digito;

      /* Insere novo nó no final da lista */
   if (!numero->valor) { /* Primeiro dígito da lista */
         /* O nó anterior e o próximo nó do novo nó serão ele próprio */
      ptrNoDigito->anterior = ptrNoDigito;
      ptrNoDigito->proximo = ptrNoDigito;

         /* Ponteiro que representa a lista apontará para o novo nó */
      numero->valor = ptrNoDigito;
   } else { /* A lista não está vazia */
         /* O nó adiante do novo nó será o primeiro da lista */
      ptrNoDigito->proximo = numero->valor;

         /* O nó anterior ao novo nó será o último nó atual */
      ptrNoDigito->anterior = numero->valor->anterior;

         /* O nó adiante do último nó atual será o novo nó */
      numero->valor->anterior->proximo = ptrNoDigito;

         /* O nó anterior do primeiro nó será o novo nó */
      numero->valor->anterior = ptrNoDigito;
   }
   ++numero->nDigitos; /* Número de dígitos aumentou */
}
```

Solução de (b): A função `DestroiNumero()`, apresentada a seguir, é responsável pela destruição de um número inteiro ilimitado alocado dinamicamente.

```
void DestroiNumero(tNumero *numero)
{
   DestroiListaDEC(&numero->valor);

   free(numero);
}
```

A função `DestroiNumero()` chama a função `DestroiListaDEC()` para destruir a lista circular duplamente encadeada que armazena os dígitos do número. A implementação dessa última função encontra-se no site dedicado a este livro.

Solução de (c): A comparação de dois números inteiros ilimitados desde que se tome a devida precaução de evitar que zeros à esquerda de um número façam parte da representação. Assim a função `RemoveZerosIniciais()`, apresentada a seguir, deve ser chamada sempre que uma operação possa fazer com que esses zeros indesejáveis sejam armazenados num número.

```
void RemoveZerosIniciais(tNumero *num)
{
   tDigitos *p, /* Apontará para o nó ora examinado */
            *ultimo; /* Apontará para o último nó da lista */
```

```
   p = num->valor; /* Faz p apontar para o primeiro nó */
   ultimo = num->valor->anterior; /* Faz 'ultimo' apontar para o último nó */

      /* Enquanto p não estiver apontando para o último nó */
      /* e o nó corrente contiver '0', remove esse nó      */
   while (p != ultimo && p->digito == '0') {
      num->valor = num->valor->proximo; /* O segundo nó passa a ser o primeiro */
      num->valor->anterior = ultimo;
      ultimo->proximo = num->valor;

      --num->nDigitos; /* A lista terá um nó a menos */

      free(p); /* Libera espaço ocupado pelo primeiro nó */

      p = p->proximo; /* Passa para o próximo nó */
   }
}
```

A comparação de dois números inteiros ilimitados confronta os sinais e os valores absolutos dos números e segue um raciocínio bastante conhecido:

- Se um número for negativo e o outro número for positivo, esse último número será considerado maior.
- Se os dois números forem positivos, o maior será aquele que tiver o maior valor absoluto.
- Se os dois números forem negativos, o maior será aquele que tiver o menor valor absoluto.

A função `ComparaInteiros()` implementa o raciocínio acima e retorna uma das seguintes constantes do tipo `tComparacao` (v. adiante):

- `IGUAL` — se os números forem iguais
- `MENOR` — se o primeiro número for menor do que o segundo
- `MAIOR` — se o primeiro número for maior do que o segundo

```
tComparacao ComparaInteiros(const tNumero *num1, const tNumero *num2)
{
   tComparacao compModulos;

      /* O resultado da comparação depende dos sinais */
      /* dos números e de seus valores absolutos      */
   if (num1->sinal == MAIS) { /* O primeiro número é positivo */
      if (num2->sinal == MAIS)
            /* Os dois números são positivos. O resultado da comparação é */
            /* obtido comparando-se seus  valores absolutos.              */
         return ComparaModulos(num1, num2);
      else
            /* O primeiro número é positivo e o segundo */
            /* é negativo. Logo o primeiro é o maior.   */
         return MAIOR;
   else /* O primeiro número é negativo */
      if (num2->sinal == MAIS)
            /* O primeiro número é negativo e o segundo */
            /* é positivo. Logo o primeiro é o menor.   */
         return MENOR;
      else { /* Os dois números são negativos */
            /* Compara os valores absolutos dos números */
         compModulos = ComparaModulos(num1, num2);

            /* O número que tem o menor valor absoluto é o maior e vice-versa */
         if (compModulos == MENOR)
            return MAIOR;
```

```
         else if (compModulos == MAIOR)
            return MENOR;

         return IGUAL; /* Os números são iguais */
      }
   }

   ASSEGURA(0, "O programa nao deveria ter chegado ate' aqui");

   return IGUAL;
}
```

A função `ComparaInteiros()` chama `ComparaModulos()` para comparar valores absolutos. Essa última função, que será apresentada a seguir, assume que os números a ser comparados não possuem zeros à esquerda e sua especificação de retorno é semelhante à da função `ComparaInteiros()`.

```
tComparacao ComparaModulos( const tNumero *num1, const tNumero *num2 )
{
   tDigitosPtr p1, p2;

      /* Essa função só funciona se os números não tiverem zeros à esquerda */

      /* O número que possuir mais dígitos é o maior */
   if (num1->nDigitos > num2->nDigitos)
      return MAIOR;
   else if (num1->nDigitos < num2->nDigitos)
      return MENOR;

      /*** Os números possuem a mesma quantidade de dígitos ***/

      /* p1 e p2 apontarão para nós das listas que representam os números */
   p1 = num1->valor; /* p1 passa a apontar para o primeiro dígito de num1 */
   p2 = num2->valor; /* p2 passa a apontar para o primeiro dígito de num2 */

      /* Compara dígitos correspondentes nas duas listas */
   do {
         /* Se for encontrado um dígito maior do que o outro, o número   */
         /* correspondente será maior; se for encontrado um dígito menor */
         /* do que o outro, o número correspondente será menor; caso     */
         /* contrário, as comparações continuam                          */
      if (p1->digito > p2->digito)
         return MAIOR;
      else if (p1->digito < p2->digito)
         return MENOR;

         /* Passa para os próximos dígitos */
      p1 = p1->proximo;
      p2 = p2->proximo;
   } while (p1 != num1->valor && p2 != num2->valor);

      /* Se não ocorreu retorno no corpo do laço, os números são iguais */
   return IGUAL;
}
```

O tipo de retorno de `ComparaInteiros()` e de `ComparaModulos()` é `tComparacao`, definido como:

```
   /* Tipo do resultado da comparação de dois números inteiros ilimitados */
typedef enum {IGUAL, MENOR, MAIOR} tComparacao;
```

Solução de (d):

Solução de (e):

Soma e subtração são operações que devem ser pensadas em conjunto. Afinal, a soma de dois números inteiros com sinais opostos significa subtração e a subtração de dois números inteiros com sinais opostos significa soma. A **Figura 10–44** mostra a soma de dois números inteiros de largura ilimitada.

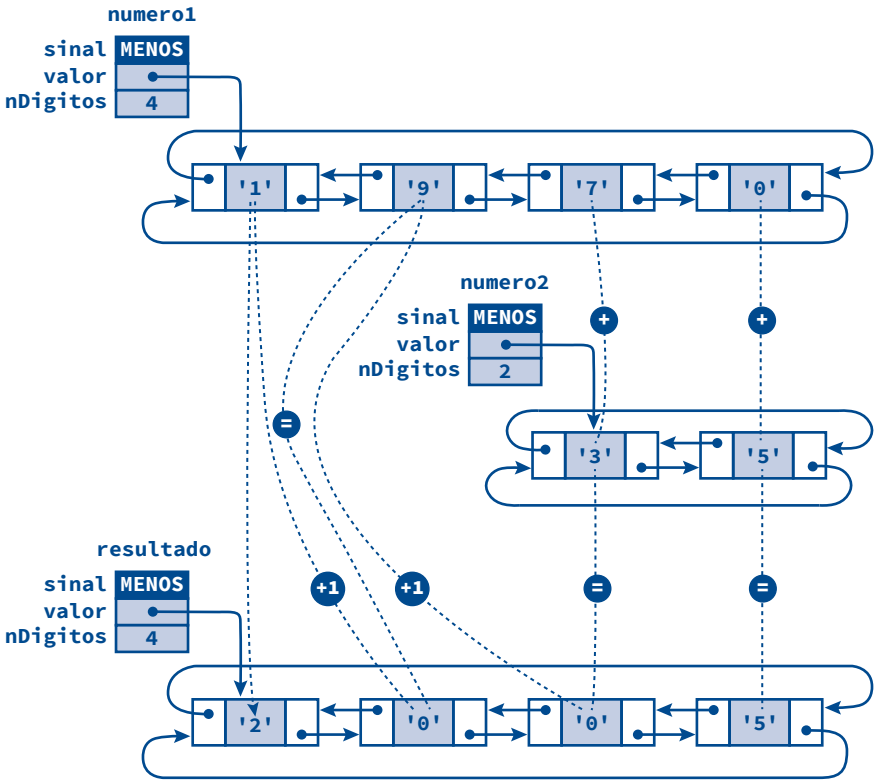


FIGURA 10–44: SOMA DE DOIS NÚMEROS INTEIROS ILIMITADOS

A abordagem a ser adotada aqui para a soma de dois números inteiros ilimitados é a seguinte:

- Se os números possuem sinais opostos, subtrai-se o número com menor valor absoluto daquele com maior valor absoluto e o sinal da operação é aquele do maior número.
- Se os números possuem o mesmo sinal, somam-se os números e o sinal da operação é o sinal comum dos operandos.

A função `SomaInteiros()` implementa a abordagem delineada acima:

```
tNumeroPtr SomaInteiros( const tNumero *num1, const tNumero *num2 )
{
   tSinal      oSinal = MAIS; /* Sinal do resultado */
   tDigitosPtr pDigitos1, /* Aponta para um nó de num1 */
               pDigitos2; /* Aponta para um nó de num2 */
   tNumero    *resultado; /* Ponteiro para o resultado */
   int         sobra = 0, /* Armazena o 'vai um' */
               digitoParcial, /* A soma de dois dígitos */
               resultadoZero = 0; /* Informa se o resultado é zero */

      /*** A operação a ser efetuada depende do     ***/
      /*** sinal e do valor absoluto de cada número ***/

   if (num1->sinal == MAIS && num2->sinal == MENOS) {
         /* O primeiro número é positivo e o segundo número é negativo */
```

```
   if (ComparaModulos(num1, num2) == MENOR) {
         /* O primeiro número é menor do que o segundo em valores */
         /* absolutos. Subtrai o primeiro número do segundo e     */
         /* atribui sinal negativo ao resultado.                  */
      resultado = SubtraiMenor(num2, num1);
      resultado->sinal = MENOS;
      return resultado;
   } else if (ComparaModulos(num1, num2) == MAIOR) {
         /* O primeiro número é maior do que o segundo em valores     */
         /* absolutos. A função SubtraiMenor() resolve esse problema. */
      return SubtraiMenor(num1, num2);
   } else {
         /* Os dois números são iguais em valores  */
         /* valores absolutos e têm sinais opostos */
      resultadoZero = 1;
   }
} else if (num1->sinal == MENOS && num2->sinal == MAIS) {
      /* O primeiro número é negativo e o segundo número é positivo */
   if (ComparaModulos(num1, num2) == MENOR) {
         /* O primeiro número é menor do que o segundo em valores     */
         /* absolutos. A função SubtraiMenor() resolve esse problema. */
      return SubtraiMenor(num2, num1);
   } else if (ComparaModulos(num1, num2) == MAIOR) {
         /* O primeiro número é maior do que o segundo em valores absolutos. */
         /* Subtrai o segundo número do primeiro e atribui sinal negativo ao */
         /* resultado.                                                       */
      resultado = SubtraiMenor(num1, num2);
      resultado->sinal = MENOS;
      return resultado;
   } else
         /* Os dois números são iguais em valores  */
         /* valores absolutos e têm sinais opostos */
      resultadoZero = 1;
} else if (num1->sinal == MENOS && num2->sinal == MENOS) {
      /* Os dois números são negativos. Portanto eles */
      /* serão somados e o resultado será negativo    */
   oSinal = MENOS;
}

   /* >>> A verdadeira soma começa aqui <<< */

   /* Tenta alocar espaço para a estrutura que */
   /* contém as informações sobre o resultado  */
ASSEGURA(resultado = malloc(sizeof(tNumero)), "Nao foi possivel alocar numero");

resultado->sinal = oSinal; /* O sinal do resultado foi obtido acima */

   /* Inicialmente, o resultado não contém digito nenhum */
resultado->valor = NULL;
resultado->nDigitos = 0;

   /* Se o resultado for 0, não há o que calcular */
if (resultadoZero) {
   AcrescentaDigitoInicio(resultado, '0');
   return resultado;
}

   /* Faz os ponteiros apontarem para os dígitos menos */
   /* significativos, que são os últimos das listas    */
```

```
   pDigitos1 = num1->valor->anterior;
   pDigitos2 = num2->valor->anterior;

      /* Percorre as listas a partir de seus finais até */
      /* atingir o início de alguma delas ou ambas      */
   do {
         /* Soma os dígitos correspondentes */
      digitoParcial = SomaDigitos( pDigitos1->digito, pDigitos2->digito, &sobra );

         /* Acrescenta o resultado da última soma de dígitos ao resultado */
      AcrescentaDigitoInicio(resultado, digitoParcial);

         /* Passa para os próximos dígitos */
      pDigitos1 = pDigitos1->anterior;
      pDigitos2 = pDigitos2->anterior;
   } while (pDigitos1 != num1->valor->anterior && pDigitos2 != num2->valor->anterior);

      /* Talvez alguns dígitos do primeiro número não tenham */
      /* sido levados em consideração no laço acima          */
   while (pDigitos1 != num1->valor->anterior) {
         /* Soma um dígito que sobrou com zero */
      digitoParcial = SomaDigitos( pDigitos1->digito, '0', &sobra );

         /* Acrescenta o resultado da última soma de dígitos ao resultado */
      AcrescentaDigitoInicio(resultado, digitoParcial);

         /* Passa para o próximo dígito do primeiro número */
      pDigitos1 = pDigitos1->anterior;
   }

      /* Talvez alguns dígitos do segundo número não tenham */
      /* sido levados em consideração no laço acima          */
   while (pDigitos2 != num2->valor->anterior) {
         /* Soma um dígito que sobrou com zero */
      digitoParcial = SomaDigitos( '0', pDigitos2->digito, &sobra );

         /* Acrescenta o resultado da última soma de dígitos ao resultado */
      AcrescentaDigitoInicio(resultado, digitoParcial);

         /* Passa para o próximo dígito do segundo número */
      pDigitos2 = pDigitos2->anterior;
   }

   if (sobra) /* Se houver 'mais um' para acrescentar, acrescenta-o */
      AcrescentaDigitoInicio(resultado, '1');
   return resultado; /* Missão concluída */
}
```

A função `SomaInteiros()` acima chama a função `SomaDigitos()` para somar os dígitos correspondentes dos dois números. Os dois primeiros parâmetros dessa última função são os dígitos que serão somados, enquanto o terceiro parâmetro é um ponteiro para um valor a ser acrescido à soma e que passará a ser acrescido à próxima soma. Essa função retorna o resultado dessa soma e é implementada como:

```
int SomaDigitos(int digito1, int digito2, int *sobra)
{
      /* A expressão dígito - '0' resulta no número inteiro que */
      /* o dígito representa em qualquer código de caracteres   */
   int valor1 = digito1 - '0',
       valor2 = digito2 - '0',
       resultado;

      /* A variável 'sobra' aponta para o que sobrou da última soma */
```

```
   resultado = valor1 + valor2 + *sobra;

      /* Se o resultado for maior do que 9 sobra 1 para a próxima soma */
   *sobra = resultado > 9 ? 1 : 0;

      /* Se 0 =< número <= 9, número + '0' resulta no dígito que representa o número */
   return resultado%10 + '0';
}
```

A função `SomaInteiros()` chama a função `AcrescentaDigitoInicio()` para acrescentar dígitos resultantes da soma dos dígitos descrita acima. Essa última função é semelhante à função `InsereListaSE()`, apresentada na **Seção 10.2.2**, que insere um nó no início de uma lista circular duplamente encadeada. Adicionalmente, a função `AcrescentaDigitoInicio()` incrementa o campo `nDigitos` do número ora sendo criado.

A função `SubtraiMenor()` chamada por `SomaInteiros()` assume que segundo número é sempre menor do que o primeiro número e é implementada como:

```
tNumeroPtr SubtraiMenor( const tNumero *num1, const tNumero *num2 )
{
   tDigitosPtr pDigitos1, /* Aponta para um nó de num1 */
               pDigitos2; /* Aponta para um nó de num2 */
   tNumero    *resultado; /* Ponteiro para o resultado */
   int         sobra = 0, /* Armazena o 'vai um' */
               digitoParcial; /* A diferença de dois dígitos */

      /* Tenta alocar espaço para a estrutura que */
      /* contém as informações sobre o resultado  */
   ASSEGURA(resultado = malloc(sizeof(tNumero)), "Nao foi possivel alocar numero");

      /* Como o segundo número é sempre menor do que o   */
      /* primeiro o sinal do resultado é sempre positivo */
   resultado->sinal = MAIS;

      /* O resultado ainda não possui nenhum dígito */
   resultado->valor = NULL;
   resultado->nDigitos = 0;

      /* Faz os ponteiros apontarem para os dígitos menos */
      /* significativos, que são os últimos das listas    */
   pDigitos1 = num1->valor->anterior;
   pDigitos2 = num2->valor->anterior;

      /* Percorre as listas a partir de seus finais até */
      /* atingir o início de alguma delas ou ambas      */
   do {
         /* Subtrai os dígitos correspondentes */
      digitoParcial = SubtraiDigitos(pDigitos1->digito, pDigitos2->digito, &sobra);

         /* Acrescenta o resultado da última subtração de dígitos ao resultado */
      AcrescentaDigitoInicio(resultado, digitoParcial);

         /* Passa para os próximos dígitos */
      pDigitos1 = pDigitos1->anterior;
      pDigitos2 = pDigitos2->anterior;
   } while (pDigitos1 != num1->valor->anterior && pDigitos2 != num2->valor->anterior);

      /* Se o número de dígitos do primeiro número for maior do que o número */
      /* de dígitos do segundo número, alguns dígitos no primeiro número não */
      /* foram ainda levados em consideração                                 */
   while (pDigitos1 != num1->valor->anterior) {

         /* Subtrai zero de um dígito que sobrou */
```

```
      digitoParcial = SubtraiDigitos( pDigitos1->digito, '0', &sobra );

         /* Acrescenta o resultado da última subtração de dígitos ao resultado */
      AcrescentaDigitoInicio(resultado, digitoParcial);

         /* Passa para o próximo dígito do primeiro número */
      pDigitos1 = pDigitos1->anterior;
   }

   RemoveZerosIniciais(resultado); /* Remove eventuais zeros iniciais do resultado */

   return resultado; /* Done */
}
```

A função `SubtraiMenor()` acima chama `SubtraiDigitos()` para subtrair os dígitos correspondentes dos números sendo subtraídos. O segundo parâmetro dessa última função é o dígito que será subtraído do primeiro parâmetro e o terceiro parâmetro é um ponteiro para um valor a ser subtraído do resultado e que passará a ser subtraído da próxima subtração de dígitos. A seguir, será exibida a implementação dessa função.

```
int SubtraiDigitos(int digito1, int digito2, int *sobra)
{
      /* A expressão dígito - '0' resulta no número inteiro que */
      /* o dígito representa em qualquer código de caracteres   */
   int valor1 = digito1 - '0', /* Inteiros representados */
       valor2 = digito2 - '0', /* por digito1 e digito2  */
       resultado;

      /* A operação depende de qual dos dígitos é maior */
   if (valor1 >= valor2) {
      resultado = valor1 - valor2 - *sobra;
      *sobra = 0;
   } else {
        /* Quando o segundo valor é maior do que o primeiro,  */
        /* soma-se 10 ao primeiro antes de subtrair e sobra 1 */
      resultado = 10 + valor1 - valor2 - *sobra;
      *sobra = 1;
   }

     /* Se 0 =< número <= 9, número + '0' resulta no dígito que representa o número */
   return resultado%10 + '0';
}
```

A operação de subtração consiste simplesmente em trocar o sinal do segundo operando e chamar a função `SomaInteiros()` para efetuar a soma, como mostrado a seguir:

```
tNumeroPtr SubtraiInteiros(tNumero *num1, tNumero *num2)
{
   tNumeroPtr resultado; /* Resultado da operação */

      /* Troca o sinal do segundo número e soma */
   num2->sinal = num2->sinal == MAIS ? MENOS : MAIS;
   resultado = SomaInteiros(num1, num2);

      /* Destroca o sinal do segundo número */
   num2->sinal = num2->sinal == MAIS ? MENOS : MAIS;

   return resultado;
}
```

Solução de (f): Uma função para exibição na tela de um número inteiro ilimitado é apresentada a seguir.

```
void ExibeNumero(const tNumero *numero)
{
   tDigitosPtr p;

      /* Se o número de dígitos do número for 0, o programa será abortado */
   ASSEGURA( numero->nDigitos, "Numero nao contem digitos" );

      /* Se o sinal do número for indefinido, o programa será abortado */
   ASSEGURA( numero->sinal == MENOS || numero->sinal == MAIS,
             "Sinal do numero e' indefinido" );

      /* Se o número for negativo, escreve '-' na tela */
   if (numero->sinal == MENOS)
      putchar('-');

      /* Faz p apontar para o primeiro nó da lista que representa o número */
   p = numero->valor;

      /* Visita cada nó da lista que representa o */
      /* número, escrevendo seu conteúdo na tela  */
   do {
      putchar(p->digito); /* Escreve dígito corrente */
      p = p->proximo; /* Passa para o próximo dígito */
   } while (p != numero->valor);
}
```

Um programa completo que testa as funções apresentadas nesta seção pode ser encontrado no site dedicado ao livro na internet.

10.8 Exercícios de Revisão

Deficiências de Arrays Dinâmicos (Seção 10.1)

1. (a) Em que situações arrays dinâmicos são satisfatórios? (a) Em que situações arrays dinâmicos *não* são satisfatórios?
2. Por que elementos de um array dinâmico não devem ser alocados individualmente?

Lista Simplesmente Encadeada sem Ordenação (Seção 10.2)

3. (a) O que é uma lista encadeada? (b) Por que listas encadeadas são consideradas variáveis dinâmicas?
4. Cite duas desvantagens do uso de listas encadeadas em relação ao uso de listas indexadas estáticas.
5. Qual é a vantagem do uso de listas encadeadas em relação a listas indexadas estáticas?
6. Qual é a vantagem do uso de listas encadeadas em relação a listas indexadas dinâmicas?
7. Apresente um exemplo de aplicação de lista que é melhor resolvido com o uso de lista encadeada.
8. Por que cada nó de uma lista encadeada armazena o endereço do próximo nó da lista?
9. O que significa visitar um nó de uma lista encadeada?
10. Uma lista simplesmente encadeada pode ser implementada utilizando-se dois **arrays associados**: `valor[]` e `encadeia[]`, em que `encadeia[i]` indica o sucessor de `valor[i]`. Se um elemento que esteja inicialmente na lista for atribuído a `valor[j]`, qual é o resultado do seguinte fragmento de programa?

```
encadeia[j] = encadeia[i];
encadeia[i] = j;
```

11. (a) Em que posição de uma lista encadeada a inserção de um elemento é mais simples? (b) Em que posição de uma lista encadeada a remoção de um elemento é mais simples?
12. Por que não é eficiente inserir um elemento como último nó de uma lista encadeada?

13. (a) Qual é o papel desempenhado pelos ponteiros `p` e `q` na função `RemoveListaSE()` explorada na **Seção 10.2.2**? (b) Por que são necessários dois ponteiros na implementação dessa função, em vez de apenas um ponteiro?

14. Por que a função `RemoveListaSE()`, discutida na **Seção 10.2.2**, recebe um parâmetro do tipo `tListaSE *`, enquanto a função `BuscaListaSE()` dessa mesma seção recebe um parâmetro do tipo `tListaSE`?

15. Por que a função `BuscaListaSE()` (v. **Seção 10.2.2**) pode usar seu parâmetro `lista` para visitar os nós da lista encadeada que ela processa, mas a função `RemoveListaSE()` (v. **Seção 10.2.2**) não deve assim proceder?

16. Que precaução deve ser levada em consideração quando se implementa uma função que libera todos os nós de uma lista encadeada, como faz a função `DestroiListaSE()` (v. **Seção 10.2.2**)?

17. Por que a função `DestroiListaSE()` precisa de um ponteiro auxiliar para cumprir sua missão?

18. Quais são as operações sobre listas encadeadas apresentadas neste capítulo que independem do tipo de conteúdo efetivo dos nós?

19. Por que não se deve usar incremento de ponteiro para acessar sequencialmente os elementos de uma lista encadeada, do mesmo modo como se faz em acesso sequencial a arrays?

20. Na função `BuscaListaSE()`, apresentada na **Seção 10.2.2**, se o nó que armazena o conteúdo cujo endereço é retornado for removido da lista, esse conteúdo se tornará um zumbi (v. **Seção 3.5**). Se houver possibilidade de isso acontecer antes de o referido conteúdo ter sido processado, apresente três soluções para esse problema.

21. (a) Para que serve a função `ProximoListaSE()`? (b) Apresente três situações nas quais essa função se faz necessária num programa-cliente.

22. Para quais das seguintes operações sobre listas simplesmente encadeadas, implementações recursivas são mais adequadas?
(a) Inserção de nós
(b) Remoção de nós
(c) Busca
(d) Consulta de dados
(e) Alteração de dados
(f) Cálculo de comprimento
(g) Inversão

23. Suponha que `L` seja uma lista simplesmente encadeada do tipo `tLista` definido como:

```
typedef struct no {
           int        conteudo;
           struct no *proximo;
        } tNo, *tLista;
```

O seguinte fragmento de programa foi construído com a intenção de inserir um item na lista `L`, cujos valores inteiros dos campos `conteudo` dos nós estão em ordem crescente:

```
tNo *p, *q;
int  chave;
...
p = L;

while (p && p->conteudo < chave)
   p = p->proximo

q = malloc(sizeof(tNo));
q->conteudo = chave;
q->proximo = p->proximo;
p->proximo = q;
```

(a) Quando esse fragmento de programa é executado com um valor para `chave` que não é igual a nenhum valor na lista, mas é menor que qualquer valor na lista, o que acontece?

(b) Das afirmações a seguir, quais delas são verdadeiras?

(i) Se `L` não for vazia e `chave` for maior do que `n->conteudo` para algum nó na lista apontado por `n`, o programa tentará seguir um ponteiro **NULL**.

(ii) Se `L` for **NULL**, após a execução do fragmento de programa, `L` apontará para um nó contendo `chave`.

(iii) Se `chave` for igual a `n->conteudo` para algum nó na lista apontado por `n`, então uma duplicata será inserida na lista.

24. (a) Quais são os casos especiais de inserção numa lista simplesmente encadeada? (b) Quais são os casos especiais de remoção numa lista simplesmente encadeada?

25. Suponha que `L` seja uma lista simplesmente encadeada do tipo `tLista` definido como:

```
typedef struct no {
           int        conteudo;
           struct no *proximo;
        } tNo, *tLista;
```

(a) Qual é a implementação mais simples da operação *insira p depois de q*, onde `q` aponta para um elemento da lista e `p` aponta para o elemento a ser inserido? (b) Usando a notação teta, qual é o custo temporal dessa operação?

Lista Simplesmente Encadeada com Ordenação (Seção 10.3)

26. (a) Qual é o papel dos ponteiros `p` e `q` na função `InsereEmOrdemLSE()` apresentada na **Seção 10.3.2**? (b) Por que são necessários dois ponteiros na implementação dessa função, e não apenas um ponteiro?

27. A função `InsereEmOrdemLSE()` (v. **Seção 10.3.2**) utiliza o seguinte laço de repetição para localizar o local de inserção de um nó:

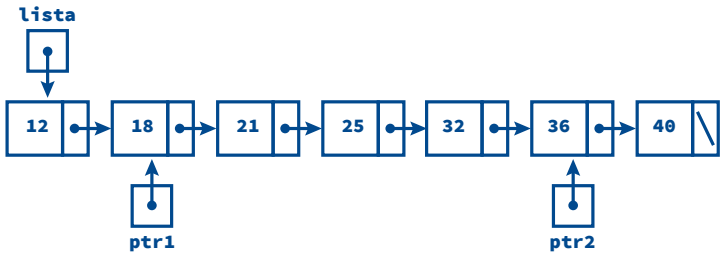
```
while (p && p->conteudo <= conteudo){
   q = p;
   p = p->proximo;
}
```

Por que se a ordem dos operandos do operador **&&** for invertida, como em:

```
while (p->conteudo <= conteudo && p) {
   q = p;
   p = p->proximo;
}
```

o programa que chama a referida função poderá ser abortado?

Nas questões de **28** a **33**, considere a lista encadeada e a definição de tipos a seguir:




```
typedef struct no {
           int        conteudo;
           struct no *proximo;
        } tNo, *tLista;
```

28. Quais das seguintes instruções são inválidas para o compilador? Para cada instrução que você julgue ser inválida, justifique sua resposta.

(a) `printf("%d", ptr1->conteudo);`
(b) `*ptr2 = 10;`
(c) `*ptr1.proximo = ptr2;`
(d) `(*ptr1).proximo = lista;`

29. Qual é o resultado de cada uma das seguintes operações?

(a) `lista->proximo->proximo->conteudo`
(b) `ptr1 == lista->proximo`
(c) `++ptr1`
(d) `lista->conteudo + ptr1->proximo->conteudo > ptr2->conteudo`
(e) `lista->conteudo + ptr1->conteudo + 2 == ptr2->conteudo - 4`
(f) `--ptr2->conteudo`
(g) `(--ptr2)->conteudo`

30. Escreva um fragmento de programa capaz de remover o nó com conteúdo efetivo igual a `21`.

31. Escreva um fragmento de programa capaz de inserir um novo nó com conteúdo efetivo igual a `28` de modo a manter lista ordenada. Suponha que esse nó já tenha sido alocado, iniciado e tenha um ponteiro denominado `pNovo` apontando para ele.

32. Escreva um fragmento de programa que, quando executado, troca as posições (e não os conteúdos) dos nós com conteúdos efetivos `21` e `32`.

33. Escreva uma única instrução que implemente cada uma das seguintes operações:

(a) Faça `ptr2` apontar para o início da lista
(b) Faça `ptr1` apontar para o último nó
(c) Torne a lista circular
(d) Faça `lista` apontar para o nó cujo conteúdo efetivo é `25`
(e) Faça `ptr2` apontar para uma lista vazia

34. Suponha que `p1` seja um ponteiro para o primeiro nó de uma lista simplesmente encadeada e `p2` seja um ponteiro para último nó dessa mesma lista. Qual das seguintes operações tem tempo de execução que depende do tamanho da lista? Justifique sua resposta.

(a) Remoção do primeiro elemento da lista
(b) Remoção do último elemento da lista
(c) Acréscimo de um elemento no início da lista
(d) Acréscimo de um elemento ao final da lista
(e) Troca de posição entre os dois primeiros elementos da lista

35. Por que busca binária não faz sentido para listas encadeadas ordenadas?

Outros Tipos de Listas Encadeadas (Seção 10.4)

36. Quantas categorias de listas encadeadas existem?

37. Em que situação é recomendável o uso de listas duplamente encadeadas ao invés de listas simplesmente encadeadas?

38. Por que se diz que não faz sentido falar em nó inicial e nó final de uma lista encadeada circular sem ordenação?

39. Em que situação é recomendável o uso de listas encadeadas com cabeça ao invés de listas encadeadas sem cabeça?

40. Em que situação é recomendável o uso de listas encadeadas circulares ao invés de listas encadeadas lineares?

41. (a) Que vantagem listas duplamente encadeadas apresentam com relação a listas simplesmente encadeadas? (b) Qual é a desvantagem de listas duplamente encadeadas?

42. (a) O que é uma lista encadeada com cabeça? (b) Que precaução se deve tomar ao implementar operações sobre listas encadeadas com cabeça?

43. (a) Quais são os casos especiais de inserção numa lista simplesmente encadeada circular? (b) Quais são os casos especiais de remoção numa lista simplesmente encadeada circular?

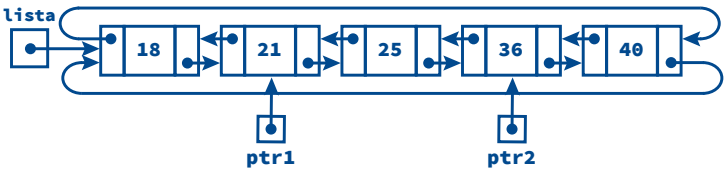
44. (a) Quais são os casos especiais de inserção numa lista duplamente encadeada linear? (b) Quais são os casos especiais de remoção numa lista duplamente encadeada linear?

45. (a) Quais são os casos especiais de inserção numa lista duplamente encadeada circular? (b) Quais são os casos especiais de remoção numa lista circular duplamente encadeada circular?

46. Suponha que você tenha um ponteiro para um nó arbitrário de uma lista encadeada. Que tipos de lista permitem visitar todos os seus nós utilizando esse ponteiro?

(a) Simplesmente encadeada linear
(b) Duplamente encadeada linear
(c) Simplesmente encadeada circular
(d) Duplamente encadeada circular

Nas questões de **47** a **50**, considere a lista duplamente encadeada circular e a definição de tipos a seguir:




```
typedef struct no {
            int        conteudo;
            struct no *anterior,
                      *proximo;
        } tNo, *tLista;
```

47. Qual é o resultado de cada uma das seguintes operações?

(a) `lista->proximo->anterior->anterior->conteudo`
(b) `ptr1->anterior->anterior == lista->anterior`
(c) `ptr2->anterior == ptr1->proximo`
(d) `lista + ptr1 == ptr2`
(e) `--ptr2->proximo->proximo->conteudo`

48. Escreva um fragmento de programa capaz de remover o nó com conteúdo efetivo igual a `40`.

49. Escreva um fragmento de programa capaz de inserir um novo nó com conteúdo efetivo igual a `15` de modo a manter lista ordenada. Suponha que esse nó já tenha sido alocado, iniciado e tenha um ponteiro denominado `pNovo` apontando para ele.

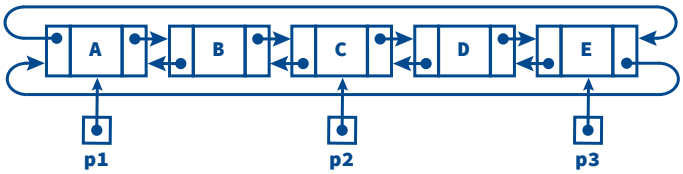
50. Escreva um fragmento de programa que, quando executado, troca as posições (e não os conteúdos) dos nós com conteúdos efetivos `18` e `36`.

51. (a) Qual é o significado da operação de acesso sequencial invertido? (b) Que categorias de lista encadeada permitem esse tipo de operação?

52. Que tipo de lista seria mais apropriado para cada uma das seguintes operações:

(a) O problema de Josephus (v. **Seção 10.7.4**)
(b) Representação de polinômios, na qual é necessária a operação de soma (v. **Seção 10.7.3**)
(c) Implementação de números inteiros de largura ilimitada (v. **Seção 10.7.5**)

53. Utilizando a representação gráfica da lista duplamente encadeada apresentada na figura a seguir, escreva a menor expressão possível para acessar o que é solicitado adiante. Suponha que os ponteiros à esquerda sejam identificados como `anterior` e os ponteiros à direita sejam identificados como `proximo`.



(a) O nó que contém `A` usando o ponteiro `p3`
(b) O nó que contém `E` usando o ponteiro `p1`
(c) O nó que contém `B` usando qualquer ponteiro
(d) O nó apontado por `p2` usando `p3`

54. Como se verifica se uma lista com cabeça está vazia?

Pilhas e Filas Encadeadas (Seção 10.5)

55. Apresente três implementações diferentes de pilhas.

56. Existe alguma vantagem em implementar pilha como lista (a) duplamente encadeada, (b) lista circular ou (c) lista com cabeça?

57. Quais são os custos temporais das operações sobre pilhas implementadas como listas simplesmente encadeadas.

58. Apresente três implementações diferentes de filas.

59. Existe alguma vantagem em implementar pilha como lista (a) duplamente encadeada, (b) lista circular ou (c) lista com cabeça?

60. Quais são os custos temporais das operações sobre filas implementadas como lista simplesmente encadeadas.

61. Em termos de abstração, fila circular e a fila encadeada são tipos de dados distintos? Explique.

Análise de Operações sobre Listas Encadeadas (Seção 10.6)

62. Compare inserção em lista encadeada com inserção em lista indexada em termos de vantagens e desvantagens.

63. Escolha **(1)** lista indexada ou **(2)** lista encadeada como melhor opção para implementação de cada uma das operações a seguir, justificando sua escolha.
(a) Inserção numa lista ordenada
(b) Busca numa lista ordenada
(c) Acesso ao enésimo elemento de uma lista

64. (a) Como se pode tornar a busca sequencial em listas encadeadas ordenadas mais eficiente? (b) Em termos de análise assintótica, é realmente possível tornar essa busca mais eficiente?

Exemplos de Programação (Seção 10.7)

65. Descreva o funcionamento da função `InverteListaSE()`, definida na **Seção 10.7.1**.

66. Descreva a implementação de polinômios usando listas encadeadas apresentada na **Seção 10.7.3**.

67. Quais são os custos temporais das operações de soma e subtração de polinômios apresentadas na **Seção 10.7.3**.

68. Por que a lista encadeada escolhida para a resolução do problema de Josephus discutido na **Seção 10.7.4** é circular?

10.9 Exercícios de Programação

EP10.1 Escreva uma função que recebe como parâmetro um ponteiro para o início de uma lista simplesmente encadeada linear e retorna um ponteiro para o último nó da lista.

EP10.2 Escreva uma função que recebe como parâmetro um ponteiro para o início de uma lista duplamente encadeada circular e retorna um ponteiro para o penúltimo nó da lista.

EP10.3 Escreva uma função em C que insira um elemento na enésima posição de uma lista simplesmente encadeada linear.

EP10.4 Modifique a função `RemoveListaSE()`, apresentada na **Seção 10.2.2**, de forma que ela seja capaz de remover com uma única chamada todas as ocorrências de nós que contenham o valor do parâmetro `conteudo` especificado e não apenas o primeiro nó encontrado.

EP10.5 Escreva uma função que recebe como parâmetros dois ponteiros para nós de uma lista simplesmente encadeada e troca as posições desses nós na lista.

EP10.6 Escreva uma função, cujo protótipo é dado por:

```
tListaSE ClonaLista(tListaSE L)
```

que recebe um ponteiro para uma lista encadeada do tipo `tListaSE`, definido na **Seção 10.2.2** e retorna o endereço de uma lista encadeada que é uma cópia da lista recebida como parâmetro. Ou seja, os conteúdos efetivos dos elementos da lista criada pela função devem corresponder aos respectivos conteúdos efetivos dos elementos da lista recebida como parâmetro e na mesma ordem.

EP10.7 Supondo que o conteúdo efetivo dos nós de uma lista encadeada seja do tipo **int**, escreva uma função em C que para ordenar a lista em ordem crescente desses valores.

EP10.8 Escreva uma função recursiva que inverte uma lista simplesmente encadeada linear.

EP10.9 Escreva uma função que recebe como parâmetros o endereço do primeiro nó de uma lista simplesmente encadeada e dois valores do tipo **int**. Essa função deve verificar se o primeiro valor aparece antes do segundo valor na lista.

EP10.10 Escreva uma função recursiva que calcula o comprimento de uma lista simplesmente encadeada linear.

EP10.11 Escreva uma função recursiva que calcula o comprimento de uma lista simplesmente encadeada circular.

EP10.12 Escreva uma função que recebe como parâmetros dois ponteiros para nós de uma lista duplamente encadeada e troca as posições desses nós na lista.

EP10.13 Supondo que o conteúdo efetivo de cada nó de uma lista simplesmente encadeada linear seja do tipo **int**, construa uma função em C que verifica se uma lista dessa natureza possui elementos com valores duplicados e remove os elementos com valores duplicados da lista.

EP10.14 Escreva uma função, cujo protótipo é dado por:

```
tListaSE ConcatenaListas(tListaSE L1, tListaSE L2)
```

que recebe como parâmetros dois ponteiros para listas encadeadas do tipo `tListaSE`, definido na **Seção 10.2.2** e retorna o endereço de uma terceira lista que consiste na concatenação dos dois parâmetros. As duas listas recebidas como parâmetros devem permanecer imutáveis.

EP10.15 Considerando listas simplesmente encadeadas lineares, escreva funções em C que executem as seguintes operações:

(a) Remova o enésimo elemento da lista
(b) Mova o enésimo elemento da lista `m` posições frente
(c) Troque o enésimo elemento com o elemento de índice `m`

EP10.16 Suponha que os registros de uma lista simplesmente encadeada contenham um campo do tipo **int**. Escreva uma função em C que receba uma lista como parâmetro de entrada e retorne uma cópia

desta lista classificada em ordem crescente de seu campo inteiro. A lista original deve permanecer imutável.

EP10.17 Escreva uma função recursiva que exibe na tela o conteúdo efetivo de cada nó de uma lista simplesmente encadeada. Suponha que o referido conteúdo é do tipo **int**.

EP10.18 Escreva uma função que remove o enésimo elemento de uma lista simplesmente encadeada linear.

EP10.19 Escreva uma função para multiplicar números inteiros de largura ilimitada como aqueles da **Seção 10.7.5**.

EP10.20 Escreva uma função para dividir números inteiros de largura ilimitada como aqueles da **Seção 10.7.5**.

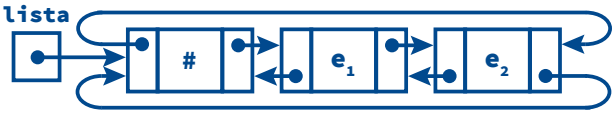
EP10.21 Escreva uma função para calcular a derivada de uma função polinomial representada conforme foi descrito na **Seção 10.7.3**.

EP10.22 Escreva uma função para calcular a integral definida de uma função polinomial num intervalo representada conforme foi descrito na **Seção 10.7.3**.

EP10.23 Escreva uma função para multiplicar dois polinômios representados conforme foi descrito na **Seção 10.7.3**.

EP10.24 Reimplemente a função `ValorNumerico()` apresentada na **Seção 10.7.3** usando o método de Horner apresentado no **Apêndice B**.

EP10.25 Considere listas duplamente encadeadas circulares com cabeça como aquela esquematizada na figura abaixo:



Suponha ainda a existência da definição de tipos:

```
typedef struct no {
          struct no *anterior, *proximo;
          char       conteudo;
} tNo, *tLista;
```

Escreva as seguintes funções: (a) `InsereInicio()`, (b) `InsereFinal()`, (c) `RemoveInicio()` e (d) `RemoveFinal()`, para inserção e remoção de nós nas duas extremidades de uma lista desse tipo.

EP10.26 Escreva uma função recursiva que insere um novo item ao final de uma lista simplesmente encadeada linear.

EP10.27 (a) Construa uma função em C que retorna o endereço do nó que é antecessor imediato de um dado nó numa lista simplesmente encadeada linear.

EP10.28 Escreva uma função que inverte uma lista circular simplesmente encadeada.

EP10.29 Escreva uma função iterativa que retorna o menor valor do tipo **int** armazenado numa lista simplesmente encadeada, supondo que os conteúdos efetivos dos nós dessa lista são do tipo **int**.

EP10.30 Escreva uma função recursiva que calcula o comprimento de uma lista simplesmente encadeada do tipo `tListaSE` definido na **Seção 10.2.2**.

EP10.31 Escreva uma função em C que recebe duas listas simplesmente encadeadas como parâmetros e retorna o endereço de uma nova lista contendo a interseção dessas duas listas. Essa função deve utilizar apenas operações básicas sobre listas; i.e., aquelas definidas na **Seção 10.2.2**.

EP10.32 Escreva uma função em C que recebe duas listas simplesmente encadeadas como parâmetros e retorna o endereço de uma nova lista contendo a união dessas duas listas. Essa função deve utilizar apenas operações básicas sobre listas; i.e., aquelas definidas na **Seção 10.2.2**.

EP10.33 Suponha que o tipo do conteúdo efetivo das listas simplesmente encadeadas envolvidas neste problema seja **int**. Escreva uma função que recebe como parâmetros dois endereços de ponteiro para listas simplesmente encadeadas e um valor do tipo **int**. A função deverá dividir a lista representada pelo primeiro parâmetro em duas listas. Cada nó da primeira dessas listas deverá ter conteúdo efetivo menor do que o inteiro recebido como parâmetro e cada nó da segunda dessas listas deverá ter conteúdo efetivo maior do que o mesmo inteiro. O protótipo da função deverá ser:

```
void DivideListas(tListaSE *l1, tListaSE *l2, int valor)
```

EP10.34 (a) Seja `tListaDupla` o tipo de um ponteiro para listas duplamente encadeadas. Escreva uma função que recebe como parâmetros apenas dois ponteiros desse tipo que apontam para dois nós de uma lista duplamente encadeada e troca as posições desses nós. (b) Se a lista não fosse duplamente encadeada, seria possível escrever essa função? Explique.

EP10.35 (a) Seja `tListaCirc` o tipo de um ponteiro para listas circulares simplesmente encadeadas. Escreva uma função que recebe como parâmetros apenas dois ponteiros desse tipo que apontam para dois nós de uma lista circular simplesmente encadeada e troca as posições desses nós. (b) Se a lista não fosse circular, seria possível escrever essa função? Explique.

EP10.36 Considerando que o arquivo binário `Tudor.bin`, criado pelo programa da **Seção 7.6.2**, armazena dados de uma turma escolar. Escreva um programa que faça o seguinte:

(a) Lê estruturas do tipo `tAluno` no arquivo `Tudor.bin` e armazena-as numa lista encadeada ordenada por um campo de estrutura especificado pelo usuário por:

```
1. Nome
2. Matrícula
3. Primeira nota
4. Segunda nota
5. Média
6. Sem ordenação
```

(b) Apresenta um menu com as seguintes opções para o usuário:

```
A. Acrescenta um aluno
R. Remove um aluno
E. Exibe turma na tela
C. Consulta dados de aluno
L. Altera dados de aluno
T. Altera ordenação da turma
I. Inverte ordenação da turma
N. Encerra o programa
```

(c) Enquanto o usuário não escolher a opção de saída do programa, lê a opção escolhida pelo usuário e executa a operação correspondente.

(d) Logo antes de encerrar, se houve alteração de dados durante a execução do programa, o arquivo utilizado deve ser atualizado.