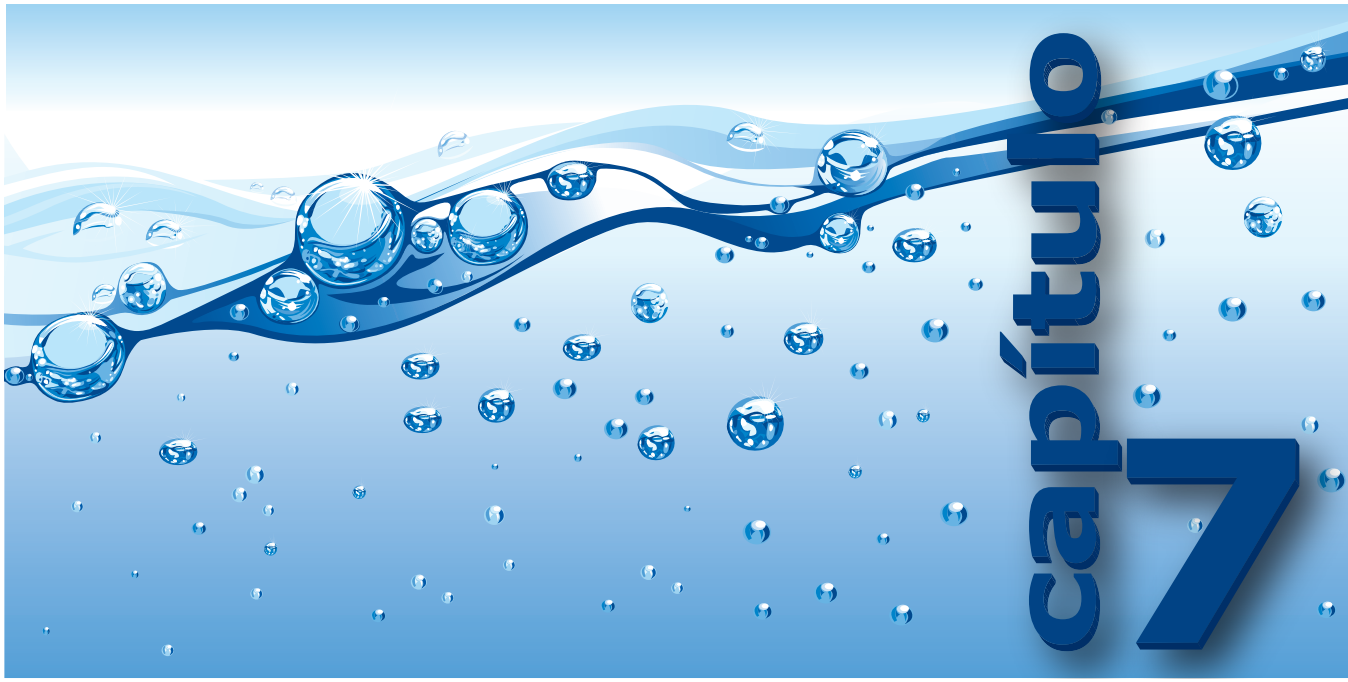

Listas Indexadas

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar os seguintes conceitos:
  - ❒ Array estático
  - ❒ Array dinâmico
  - ❒ Lista indexada
  - ❒ Índice de elemento
  - ❒ Acesso direto
  - ❒ Sucessor
  - ❒ Antecessor
  - ❒ Lista vazia
  - ❒ Lista ordenada
  - ❒ Lista sem ordenação
  - ❒ Chave de busca
  - ❒ Chave primária
  - ❒ Chave secundária
  - ❒ Busca sequencial
  - ❒ Operação essencial
  - ❒ Operação complementar
  - ❒ Busca binária
  - ❒ Exceção
  - ❒ Tratamento de exceção
  - ❒ Precondição
  - ❒ Matriz esparsa
- Descrever operações sobre listas indexadas ordenadas e sem ordenação
- Explicar a diferença entre lista indexada e array
- Usar análise assintótica para avaliar os custos de melhor caso e de pior caso de cada operação sobre lista indexada
- Implementar todas as operações sobre listas indexadas
- Diferenciar acréscimo e inserção em lista indexada
- Descrever o funcionamento de uma macro usada em tratamento de exceção
- Explicar como uma matriz esparsa pode ser armazenada utilizando uma lista indexada ordenada
- Discorrer sobre menu sensível ao contexto

objetivos

ESTE CAPÍTULO LIDA com um dos conceitos mais importantes de Estruturas de Dados, que é o conceito de lista. Aqui, será apresentada uma categoria particular de listas denominada listas indexadas devido ao fato de elas permitirem **acesso direto** (i.e., por meio de índices), assim como ocorre com os arrays. Este capítulo também mostrará como essa abstração pode ser naturalmente implementada por meio de **arrays estáticos**, cujos tamanhos são conhecidos em tempo de programação (v. **Seção 3.1**). Arrays estáticos contrastam com arrays dinâmicos, cujos tamanhos podem variar durante a execução de um programa. No **Capítulo 9**, serão estudadas listas indexadas implementadas por meio de arrays dinâmicos, enquanto o **Capítulo 10** mostrará como implementar listas encadeadas. É importante notar que, qualquer que seja a implementação considerada, o conceito (i. e., a abstração) de lista é sempre o mesmo.

7.1 Listas sem Ordenação

7.1.1 Abstração

Uma **lista** é uma coleção de **elementos** (ou **itens**) do mesmo tipo e que pode aumentar ou diminuir de tamanho, de acordo com a necessidade. Numa lista, elementos podem ser acessados, inseridos ou removidos em qualquer posição.

Uma lista pode ser **vazia** (i.e., sem nenhum elemento) ou pode ter um número indeterminado, mas finito, de elementos. Numa lista não vazia, cada elemento, com exceção do primeiro e do último elementos, possui um elemento que o antecede, denominado **antecessor**, e outro que o sucede, denominado **sucessor**. O primeiro elemento de uma lista possui apenas sucessor, ao passo que o último elemento de uma lista possui apenas antecessor.

O **índice** (ou **posição**) de um elemento de uma lista não vazia é definido como o número de elementos que o antecedem, de modo que o primeiro elemento possui índice *0*, o segundo elemento possui índice *1* e assim por diante. O último elemento de uma lista com n elemento possui índice igual a $n - 1$.

Abstratamente, pode-se definir **lista indexada** como sendo vazia (i.e., sem nenhum elemento) ou com o formato:

$$(a_0, a_1, ..., a_{n-1})$$

em que os elementos a_i são membros de algum conjunto e i são os índices associados a esses elementos, sendo $0 \leq i \leq n - 1$. O conceito de lista não faz nenhuma restrição quanto ao fato de haver ou não elementos com os mesmos valores numa lista. Ou seja, numa lista, dois elementos são diferentes se e somente se eles estão associados a índices diferentes, independentemente do fato de eles possuírem os mesmos valores.

Podem-se definir várias operações sobre listas, mas existem apenas cinco **operações essenciais**:

[1] **Iniciação** (ou **criação**) de uma lista, que consiste em criar ou tornar uma lista vazia (i.e., sem nenhum elemento).

[2] Cálculo do **comprimento da lista**, que é uma propriedade da lista cujo valor varia de acordo com o uso da lista.

[3] **Acesso ao elemento** a_i, $0 \leq i \leq n - 1$, em que n é o comprimento da lista.

[4] **Inserção de um elemento** na posição i, $0 \leq i \leq n$, fazendo com que os elementos antes ordenados por $i, i + 1, ..., n - 1$ passem a ser ordenados, respectivamente, por $i + 1, i + 2, ..., n$. Quando $i = n$, nenhum elemento que já se encontre na lista tem seu índice alterado.

[5] **Remoção do elemento localizado na posição** i, $0 \leq i \leq n - 1$, fazendo com que os elementos antes ordenados por $i + 1, i + 2, ..., n - 1$ passem a ser ordenados, respectivamente, por $i, i + 1, ..., n - 2$.

Outras operações podem ser acrescentadas ao rol acima, mas todas podem ser implementadas usando as operações essenciais apresentadas acima. Por exemplo, uma operação complementar consiste em alterar o valor de um elemento a_i ($0 \leq i \leq n - 1$), mas apesar de ser bem conveniente, ela pode ser implementada por meio das

operações [5] e [4], nessa ordem. Ou seja, pode-se alterar o valor de um elemento de uma lista, removendo-o e, em seguida, inserindo um elemento com o novo valor na posição do elemento removido. De qualquer modo, as seguintes **operações complementares** serão consideradas aqui como parte da abstração de lista indexada:

[6] **Checagem de lista vazia**, que verifica se uma lista possui algum elemento ou não.

[7] **Acréscimo de um elemento** ao final da lista. Essa operação é o mesmo que inserção na posição n, em que n é o tamanho da lista.

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

[9] **Busca de um elemento** a_i $(0 \leq i \leq n - 1)$ cujo valor é igual a um determinado valor a.

A escolha desse último conjunto de operações é um tanto arbitrária e depende de como a lista será usada.

Existem semelhanças entre os conceitos de lista indexada e array, mas, de fato, eles são diferentes. Em particular, um índice de elemento de uma lista é diferente de um índice de elemento de um array, pois índices de arrays são estáticos, enquanto os índices dos elementos de uma lista variam de acordo com as operações de inserção e remoção efetuadas na lista. Por exemplo, se o primeiro elemento de uma lista for removido, os índices de todos os elementos remanescentes serão alterados. A **Tabela 7–1** ajuda a esclarecer essas diferenças.

Semelhanças	Diferenças
Elementos de listas e arrays são do mesmo tipo	Arrays estáticos possuem tamanho fixo, enquanto o tamanho de uma lista é variável
Elementos de listas e arrays são associados a índices	O índice de um elemento de uma lista pode ser alterado, enquanto o índice de elemento de array não pode
Consulta e alteração são operações permitidas sobre listas e arrays	Essas são as únicas operações sobre arrays; por outro lado, listas possuem outras operações

Tabela 7–1: Semelhanças e Diferenças entre Listas Indexadas e Arrays

Antes de prosseguir, é importante salientar a diferença entre inserir e acrescentar. Neste livro, *acrescentar numa lista* significa sempre adicionar um elemento ao final de uma lista. Por outro lado, *inserir numa lista* significa adicionar um elemento em qualquer posição de uma lista, inclusive ao final.

7.1.2 Implementação Usando Arrays Estáticos

Lista indexada é um conceito abstrato que pode ser implementado de várias maneiras. O modo mais simples de implementação desse conceito é por meio de arrays estáticos (i.e., arrays cujos tamanhos são conhecidos em tempo de compilação).

Aqui, será demonstrado por meio de listas de elementos do tipo **int** como implementar o conceito de lista por meio de arrays estáticos. Se você entender o raciocínio utilizado nessa implementação, provavelmente não terá dificuldade de implementar listas cujos elementos sejam dos tipos mais diversos, como tipos estruturados (v. **Seção 7.6.1**) e genéricos (v. **Capítulo 11**).

Antes de prosseguir, é importante observar que o modo de implementação de listas e suas operações como tipos de dados é arbitrário. Por exemplo, a operação de remoção pode proceder de maneiras diferentes quando o item a ser removido não se encontra na lista, como, por exemplo, abortar o programa que usa a operação ou simplesmente ignorar o fato[1].

[1] Operações de alteração de valores de elementos de listas são relativamente triviais, de modo que suas implementações não são apresentada neste livro. Elas podem, contudo, ser encontradas no site dedicado ao livro na internet.

Definições de Tipos

O primeiro passo na implementação de uma estrutura de dados consiste na definição do tipo que será utilizado. A escolha desse tipo é arbitrária, mas deve ser justificável em termos de custos temporal e espacial. Aqui, a escolha recai sobre um tipo que usa estrutura com dois campos: **(1)** um array que armazena os elementos da lista em si e **(2)** um campo que armazena o comprimento da lista. A vantagem dessa escolha é que o custo temporal da operação que calcula o comprimento da lista, que é uma operação frequentemente utilizada, é $\theta(1)$, enquanto que o custo espacial de todas operações sobre listas indexadas permanece sendo $\theta(1)$ (v. **Seção 7.3**).

Levando em consideração o que foi exposto no parágrafo anterior, o tipo que representa listas nessa implementação é definido como:

```
typedef int tElemento;

typedef struct {
           int       nElementos; /* Número de elementos */
           tElemento elementos[MAX_ELEMENTOS]; /* Os elementos */
        } tListaIdx;
```

Nessa última definição, `MAX_ELEMENTOS` é o número máximo de elementos que uma lista desse tipo pode conter.

Iniciação

A função `IniciaListaIdx()` inicia uma lista do tipo `tListaIdx`. Essa função simplesmente atribui zero ao comprimento da lista, indicando o fato de que toda lista é iniciada vazia (i.e., com comprimento zero).

```
void IniciaListaIdx(tListaIdx *lista)
{
   lista->nElementos = 0;
}
```

Comprimento

A função `ComprimentoListaIdx()` retorna o comprimento de uma lista do tipo `tListaIdx` definido acima. Como uma lista desse tipo armazena seu próprio comprimento, não há nada o que calcular.

```
int ComprimentoListaIdx(const tListaIdx *lista)
{
   return lista->nElementos; /* Não há o que calcular */
}
```

Acesso

A função `ObtemElementoListaIdx()` retorna o elemento que se encontra numa determinada posição de uma lista do tipo em discussão:

```
tElemento ObtemElementoListaIdx(const tListaIdx *lista, int indice)
{
   if (indice < 0 || indice >= lista->nElementos) {
      printf("\nElemento inexistente\n");
      exit(1);
   }

   return lista->elementos[indice];
}
```

A função `ObtemElementoListaIdx()` apresenta uma peculiaridade interessante que é o fato de ela ser capaz de abortar o programa-cliente por meio de uma chamada da função **exit()**. A função em questão assim procede, pois não lhe resta nenhuma outra alternativa quando o índice recebido como parâmetro é inválido. Quer dizer,

essa função deve retornar o valor do elemento que se encontra na posição indicada pelo segundo parâmetro, mas, se o índice que lhe é passado como parâmetro for inválido, não haverá elemento na referida posição e ela não terá nenhum valor disponível para retornar. Isso ocorre porque, como o tipo do elemento é **int**, qualquer valor que ela retorne nessa circunstância será interpretado como um valor legítimo de elemento que se encontra na lista, mesmo quando esse não é o caso. Esse assunto será discutido em maiores detalhes na **Seção 7.4**.

Inserção

A **Figura 7–1** ilustra o processo de inserção de um novo item numa lista com `n` elementos implementada num array com `MAX` elementos. Note a movimentação de alguns elementos que ocorre na lista antes da inserção do novo elemento. Essa movimentação é necessária para tornar disponível um espaço no interior da lista antes que ocorra a inserção. Compare essa figura com a **Figura 7–2** e constate a diferença entre as operações de inserção e acréscimo de elemento numa lista. Em especial, note que o acréscimo de um elemento não requer movimentação de elementos já presentes na lista.

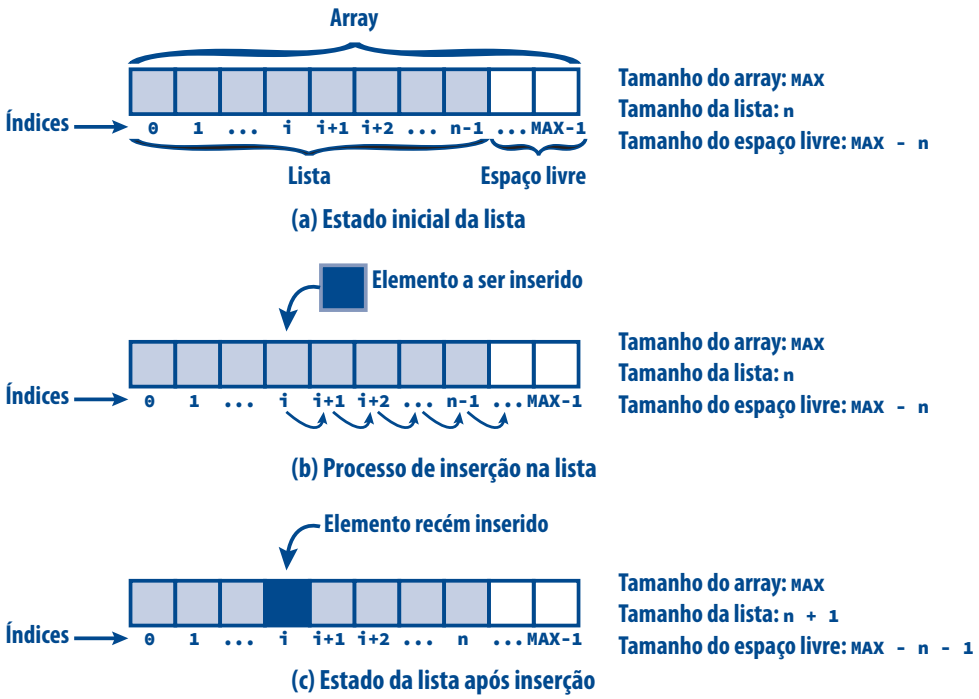

Figura 7–1: Inserção de Item em Lista Indexada

A função `InsereListaIdx()` efetua a inserção de um elemento numa lista conforme foi discutido acima.

```
int InsereListaIdx(tListaIdx *lista, tElemento item, int indice)
{
      /* Se a lista estiver cheia, não há como inserir mais um elemento */
   if (EstaCheiaListaIdx(lista)) {
      printf("\nA lista esta cheia\n");
      return 1;
   }

      /* Verifica se o índice é válido */
   if (indice < 0 || indice > lista->nElementos) {
      printf("\nPosicao de insercao inexistente\n");
      return 1;
   }
```

```
      /* Abre espaço para o novo elemento */
   for (int i = lista->nElementos - 1; i >= indice; --i)
      lista->elementos[i + 1] = lista->elementos[i];

   lista->elementos[indice] = item; /* Insere o novo elemento */

   lista->nElementos++; /* O tamanho da lista aumentou */

   return 0;
}
```

Note que a função `InsereListaIdx()` retorna um valor que indica se a operação de inserção foi bem-sucedida (retorno igual a `0`) ou não (retorno igual a `1`). Existem duas condições que impedem essa função de ser bem-sucedida em sua missão:

1. O índice (i.e., a posição) da inserção é inválido. Observe que essa função permite que o índice seja igual ao número de elementos da lista porque se pode inserir um elemento após o último elemento da lista.
2. A lista está cheia. Uma lista implementada como array estático é considerada cheia quando o array no qual seus elementos são armazenados não comporta mais nenhum elemento.

Para verificar se uma lista está cheia, utiliza-se a função `EstaCheiaListaIdx()`:

```
int EstaCheiaListaIdx(const tListaIdx *lista)
{
   return lista->nElementos >= MAX_ELEMENTOS;
}
```

É interessante notar que a função `EstaCheiaListaIdx()` não implementa nenhuma operação apresentada na **Seção 7.1.1**. Isso significa que essa deveria ser uma função auxiliar de implementação e, assim, deveria ser definida com o qualificador **static**, indicando que ela é uma função local que auxilia a implementação das operações descritas naquela seção. Portanto não deveria haver nenhuma alusão a essa função no arquivo de interface do tipo (v. **Seção 5.4**) que implementa o conceito de lista. Essa questão diz respeito à diferença entre abstração e implementação discutida no **Capítulo 5**. Ou seja, na definição do conceito de lista apresentado na **Seção 7.1.1**, faz-se referência a lista vazia, mas não há nenhuma referência a lista cheia. Isso ocorre porque, conceitualmente, uma lista nunca está cheia. O fato de uma lista estar cheia é decorrente dessa implementação específica que utiliza array estático que tem tamanho fixo. Você verá no **Capítulo 10**, no qual listas são implementadas usando alocação dinâmica de memória, que *lista cheia* não faz sentido.

O raciocínio apresentado acima é perfeito do ponto de vista conceitual, mas não se pode dizer o mesmo do ponto de vista pragmático. Quer dizer, do ponto de vista prático, é útil para um cliente ser capaz de determinar se uma lista está cheia ou ainda é possível acrescentar mais elementos (v. exemplo **na Seção 7.6.1**).

Acréscimo

O modo mais eficiente de acrescentar um elemento a uma lista implementada por meio de array é ao final da lista, como ilustra a **Figura 7–2**. Note nessa figura que, diferentemente do que ocorre quando um elemento é inserido numa posição arbitrária da lista (v. **Figura 7–1**), numa operação de acréscimo nunca ocorre movimentação de elementos.

A função `AcrescentaListaIdx()` acrescenta um novo elemento ao final de uma lista do tipo sob discussão. Essa função retorna `0`, se a operação for bem-sucedida ou `1`, em caso contrário (i.e., quando não ocorre nenhuma acréscimo porque não há mais espaço no array).

```
int AcrescentaListaIdx(tListaIdx *lista, tElemento elemento)
{
      /* Verifica se é possível acrescentar mais um elemento na lista */
```

```
   if (EstaCheiaListaIdx(lista))
      return 1; /* A lista está cheia */

      /* Acrescenta um novo elemento ao final da lista */
   lista->elementos[lista->nElementos] = elemento;

   ++lista->nElementos; /* O tamanho da lista aumentou */

   return 0;
}
```

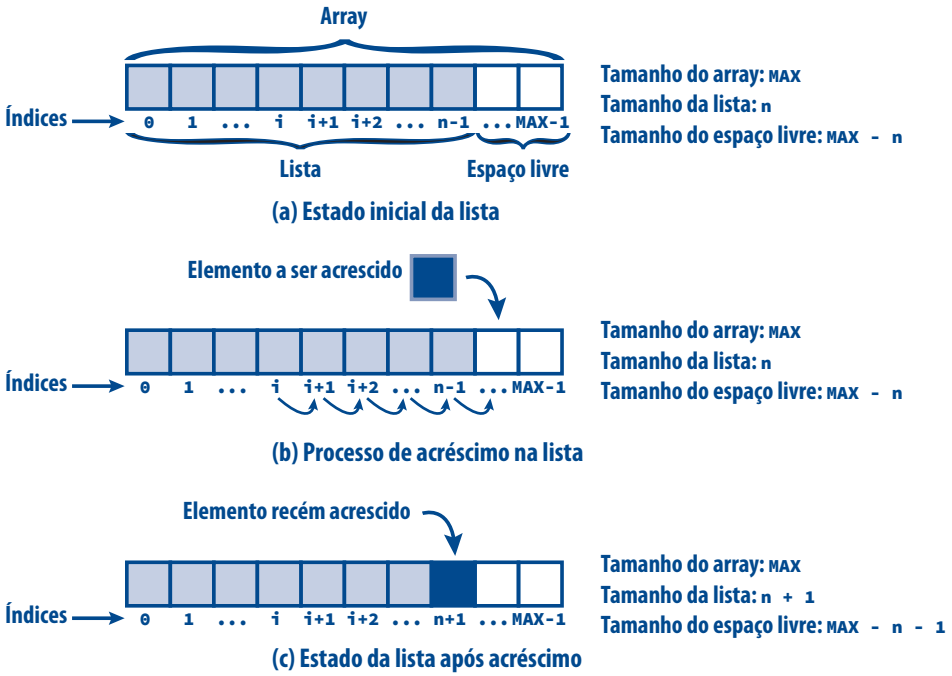

FIGURA 7–2: ACRÉSCIMO DE ITEM EM LISTA INDEXADA SEM ORDENAÇÃO

A função `AcrescentaListaIdx()` apresenta mais uma situação na qual a sua missão pode não ser cumprida, mas, nesse caso, ela não aborta o programa [como faz a função `RemoveListaIdx()`], pois ela pode indicar por meio do valor retornado quando é bem-sucedida ou não.

Remoção

A remoção de um elemento de uma lista implementada em array requer a movimentação dos elementos que sucedem o elemento a ser removido na lista, como mostra a **Figura 7–3**.

A função `RemoveListaIdx()` apresentada a seguir implementa a remoção de um elemento de uma lista do tipo em questão. Essa função retorna, quando é possível, o valor do elemento removido.

```
tElemento RemoveListaIdx(tListaIdx *lista, int indice)
{
   tElemento itemRemovido;

         /* Verifica se o índice é válido */
   if (indice < 0 || indice >= lista->nElementos) {
      printf("\nPosicao de remocao inexistente\n");
      exit(1);
   }

   itemRemovido = lista->elementos[indice];
```

```
      /* Remover um elemento significa mover cada elemento uma posição para   */
      /* trás a partir do próximo elemento adiante daquele que será  removido */
   for (int i = indice; i < lista->nElementos - 1; i++)
      lista->elementos[i] = lista->elementos[i + 1];

   --lista->nElementos; /* O tamanho da lista diminuiu */

   return itemRemovido;
}
```

Observe que a função `RemoveListaIdx()` causa o aborto do programa quando não lhe resta nenhum valor para retornar, assim como faz a função `ObtemElementoListaIdx()` apresentada antes.

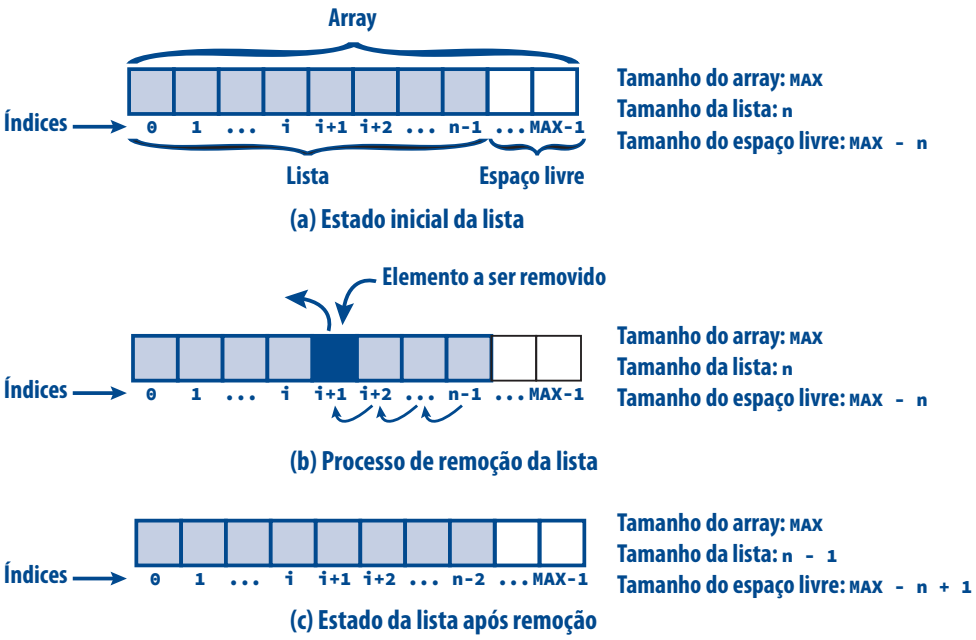

Figura 7–3: Remoção de Item em Lista Indexada

Checagem de Lista Vazia

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

```
int EstaVaziaListaIdx(const tListaIdx *lista)
{
   return lista->nElementos == 0;
}
```

7.1.3 Busca Sequencial

A tentativa de localização de um elemento de uma lista, denominada **busca**, é uma operação básica em processamento de listas, pois outras operações dependem dela. Por exemplo, para remover, consultar ou alterar um elemento é necessário, em primeiro lugar, saber se ele se encontra na lista e, se for o caso, em que posição ele se encontra. Para efetuar uma operação de busca numa lista de estruturas (**registros**) é necessário especificar um campo dessas estruturas, denominado **chave de busca**, que será comparado com um determinado valor. Assim uma operação de busca numa lista consiste em comparar um valor específico da chave de busca com a chave de busca de cada estrutura da lista. Então, essa operação encerra quando é encontrado um elemento da lista cuja chave casa com o valor específico da chave ou quando se pode afirmar que o elemento procurado não

se encontra na lista. No caso de lista sem ordenação, só é possível garantir que não ocorre nenhum casamento quando cada elemento da lista é examinado. Esse tipo de busca é denominado **busca sequencial**. No caso de lista ordenada, é possível acelerar a busca (v. **Seção 7.2.3**).

Uma chave que não se repete num conjunto de registros (p. ex., número de CPF no banco de dados da Receita Federal) é denominada **chave primária**; caso contrário (p. ex., nome de pessoa no mesmo banco de dados), ela é denominada **chave secundária**. Se uma chave secundária for usada como chave de busca uma busca sequencial encontrará apenas o primeiro registro cuja chave de busca casa com o valor especificado. Como analogia, a **Figura 7–4** mostra um conjunto de registros impressos com suas respectivas chaves primárias no topo de cada registro.

FIGURA 7–4: REGISTROS COM RESPECTIVAS CHAVES

Quando uma chave é considerada primária, deve-se tomar a devida precaução para evitar que um registro contendo uma chave repetida não seja inserido numa lista. Apesar de números de identificação, como é o caso de matrícula no exemplo da **Seção 7.6.2**, serem normalmente considerados chaves primárias, esse cuidado não é levado em consideração neste livro para manter os exemplos mais simples. O algoritmo de busca sequencial é apresentado na **Figura 7–5**.

ALGORITMO BUSCASEQUENCIAL

ENTRADA: Uma lista indexada com *n* elementos e uma chave de busca

RETORNO: O índice do primeiro elemento da lista cuja chave casa com a chave de busca ou um valor indicando que a chave não foi encontrada

1. Considere como elemento corrente o primeiro elemento da lista
2. Enquanto o último elemento da lista não tiver sido visitado, faça o seguinte:
   - **2.1** Se a chave do elemento corrente casar com a chave de busca, retorne o índice desse elemento
   - **2.2** Considere como elemento corrente o próximo elemento da lista
3. Retorne um valor que indique que a chave de busca não foi encontrada

FIGURA 7–5: ALGORITMO DE BUSCA SEQUENCIAL

A função `BuscaListaIdx()` procura um elemento numa lista que apresenta um determinado valor. Ela retorna o índice do elemento procurado se ele for encontrado ou um valor negativo indicando que não existe elemento na lista que apresente o valor recebido como parâmetro.

```
int BuscaListaIdx(const tListaIdx *lista, tElemento elemento)
{
   for (int i = 0; i < lista->nElementos; ++i)
      if (lista->elementos[i] == elemento)
         return i; /* Elemento foi encontrado */

   return -1; /* Elemento não foi encontrado */
}
```

A busca levada a efeito pela função `BuscaListaIdx()` é denominada busca sequencial. Essa técnica rudimentar de busca consiste em comparar cada elemento da lista com aquele procurado do primeiro ao último elemento ou até que o referido elemento seja encontrado. Na busca sequencial, quando um valor é encontrado, ele corresponde sempre ao primeiro elemento do array que possui o referido valor, mas o mesmo não ocorre com busca binária.

O pior caso da função `BuscaListaIdx()` ocorre quando o elemento procurado não se encontra na lista ou ele é o último elemento da lista. Nesse caso, o custo da operação é $\theta(n)$, já que toda lista deve ser acessada. O melhor caso de busca sequencial ocorre quando o elemento procurado encontra-se no início da lista. Nesse caso, o custo da operação é $\theta(1)$, pois apenas o primeiro elemento da lista precisa ser acessado, independentemente do tamanho da lista. No caso médio, espera-se encontrar o elemento procurado na posição central. Portanto, no caso médio, o custo temporal dessa função é $\theta(n/2)$. Mas, $\theta(n/2)$ é o mesmo que $\theta(n)$, como foi visto na **Seção 6.6**.

7.2 Listas Ordenadas

7.2.1 Abstração

Uma **lista ordenada** é uma lista constituída de elementos que ocupam posições em uma determinada ordem. Numa lista ordenada, os elementos são ordenados em ordem crescente ou decrescente de seus conteúdos ou parte de seus conteúdos.

Quando cada elemento de uma lista é estruturado (v. **Seção 5.1**), a ordenação é ditada por um componente de seu conteúdo denominado **chave de ordenação**, que pode ser primária ou secundária (v. **Seção 7.1.3**).

O conjunto de operações sobre listas ordenadas deve ser escolhido de modo a satisfazer os conceitos de lista, apresentados na **Seção 7.1.1**, e de ordenação apresentados acima. Assim, dentre as operações essenciais apresentadas naquela seção, apenas a operação **[4]** precisa ser revista de modo que a inserção de um elemento mantenha a lista ordenada. Assim essa operação deve ser reescrita para listas ordenadas como:

[4']Inserção de um novo elemento numa posição tal que a lista permaneça ordenada

Dentre as operações complementares apresentadas na **Seção 7.1.1**, as seguintes operações deixam de fazer sentido devido ao fato de serem capazes de desordenar uma lista ordenada:

[7] Acréscimo de um elemento ao final da lista

[8] Alteração de valor de um elemento

7.2.2 Implementação Usando Arrays Estáticos

Esta seção apresenta uma implementação de listas ordenadas utilizando array. Novamente, como foi considerado na **Seção 7.1.2**, o tipo de cada elemento armazenado nessas listas é **int**, de modo que as mesmas definições de tipos utilizadas naquela seção se aplicam aqui. Também, as implementações das operações **[1]**, **[2]**, **[3]**, **[5]** e **[6]** discutidas naquela seção não serão reapresentadas aqui, pois elas são idênticas para listas ordenadas e sem ordenação. A operação de busca **(9)** implementada na **Seção 7.1.3** pode ser utilizada tanto com listas sem ordenação quanto com listas ordenadas, mas, nesse último caso, existe um modo mais eficiente de implementação que será apresentado na próxima seção. Além disso, aqui, serão consideradas apenas listas ordenadas em ordem crescente, mas o raciocínio envolvido na implementação de listas ordenadas em ordem decrescente é similar.

A função `InsereEmOrdemIdx()` apresentada a seguir insere um elemento numa lista indexada ordenada preservando a ordenação da lista. Essa função retorna `0` quando a operação é bem-sucedida e `1` em caso contrário (i.e., quando é impossível inserir um elemento porque a lista está cheia).

```
int InsereEmOrdemIdx(tListaIdx *lista, tElemento elemento)
{
   int posicao, /* Posição de inserção do elemento */
       i;

      /* Verifica se ainda há espaço para inserção */
   if (EstaCheiaListaIdx(lista))
      return 1; /* Não há mais espaço no array */

      /* Se a lista estiver vazia, o elemento é inserido */
      /* na primeira posição do array. De fato, isso não */
      /* é necessário. É apenas um atalho.               */
   if (EstaVaziaListaIdx(lista)) {
         /* O elemento será o primeiro da lista */
      lista->elementos[0] = elemento;

      ++lista->nElementos; /* A lista cresceu */

      return 0; /* Serviço completo */
   }

      /* Encontra a posição no array onde o elemento será inserido */
   for (posicao = 0; posicao < lista->nElementos; ++posicao) {
      /*                                                                */
      /* Se for encontrado um elemento do array cujo valor é maior do que o    */
      /* valor do elemento a ser inserido, é na posição atual desse elemento   */
      /* que o elemento recebido como parâmetro será inserido. Se tal elemento */
      /* não for encontrado, a inserção será efetuada ao final do array.       */
      /*                                                                */

      if (lista->elementos[posicao] > elemento)
         break; /* Posição de inserção encontrada */
   }

   /*********************************************/
   /*                                           */
   /* Neste ponto, a variável 'posicao' armazena o */
   /* índice no qual o novo elemento será inserido */
   /*                                           */
   /*********************************************/

      /* Abre espaço para o novo elemento */
   for (i = lista->nElementos - 1; i >= posicao; --i) {
         /* Move cada elemento uma posição adiante a partir do  */
         /* elemento que ora se encontra na posição de inserção */
      lista->elementos[i + 1] = lista->elementos[i];
   }

   lista->elementos[posicao] = elemento; /* Insere o novo elemento */

   ++lista->nElementos; /* A lista cresceu */

   return 0;
}
```

7.2.3 Busca Binária

Uma técnica mais eficiente de busca do que aquela apresentada na **Seção 7.1.3** pode ser empregada quando a lista está ordenada. Essa técnica é denominada **busca binária** e segue o algoritmo da **Figura 7–6**.

Algoritmo BuscaBinária

Entrada: Uma lista indexada ordenada com *n* elementos e uma chave de busca

Retorno: O índice do elemento da lista cuja chave casar primeiro com a chave de busca ou um valor indicando que a chave não foi encontrada

1. Atribua à *inf* o menor índice e a *sup* o maior índice da lista
2. Enquanto *inf* ≤ *sup* faça:
   - **2.1** Atribua ⌊*inf* + *(sup* – *inf)/2*⌋ à variável *meio*
   - **2.2** Compare a chave de busca com a chave do elemento que se encontra no índice *meio* da lista
   - **2.3** Se as chaves forem iguais, retorne *meio*
   - **2.4** Se a chave de busca for menor do que a chave do elemento que se encontra no índice *meio* da lista, atribua *meio* – *1* a *sup*
   - **2.5** Caso contrário (se a chave de busca for maior do que a chave do elemento que se encontra no índice meio da lista), atribua *meio* + *1* a *inf*
3. Retorne um valor que indique que a chave de busca não foi encontrada

Figura 7–6: Algoritmo de Busca Binária

A **Figura 7–7** ilustra a execução do algoritmo acima quando o elemento procurado se encontra na lista, enquanto a **Figura 7–8** mostra a execução desse algoritmo quando o elemento procurado não se encontra na lista.

O algoritmo de busca binária pode ser implementado como mostra a definição da função `BuscaBinaria()` abaixo.

```
int BuscaBinaria(const tListaIdx *lista, tElemento elemento)
{
   int inf, /* Limite inferior da busca */
       sup, /* Limite superior da busca */
       meio; /* Meio do intervalo */

      /* limites inferior e superior iniciais */
   inf = 0;
   sup = lista->nElementos - 1;

      /* Efetua a busca binária */
   while (inf <= sup) {
         /* Calcula o meio do intervalo */
      meio = inf + (sup - inf)/2;

         /* Verifica se o elemento se encontra no meio do intervalo */
      if (lista->elementos[meio] == elemento)
         return meio; /* Elemento encontrado */

         /* Ajusta o intervalo de busca */
      if (elemento  < lista->elementos[meio])
         sup = meio - 1;
      else
         inf = meio + 1;
   }

   return -1; /* Elemento não foi encontrado */
}
```

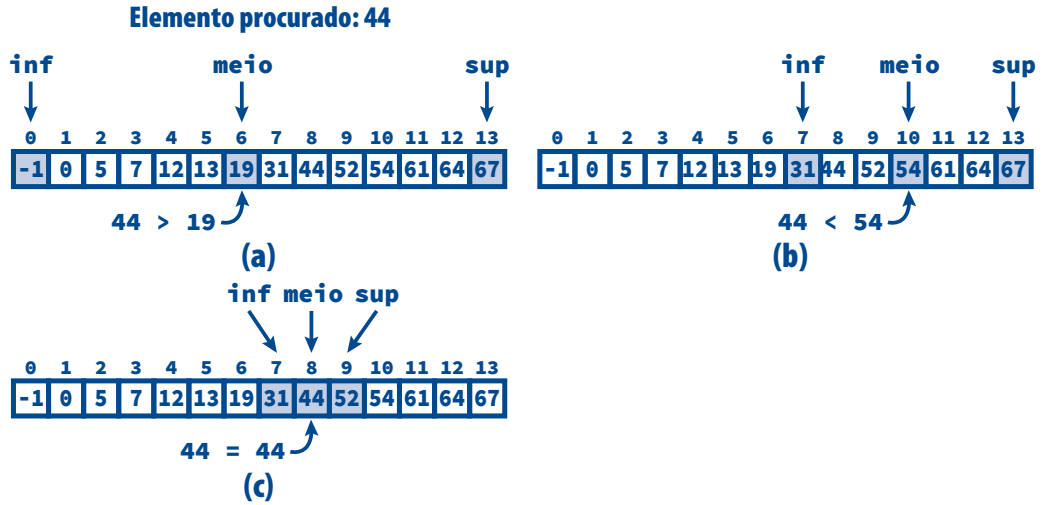

FIGURA 7–7: ELEMENTO ENCONTRADO NUMA BUSCA BINÁRIA

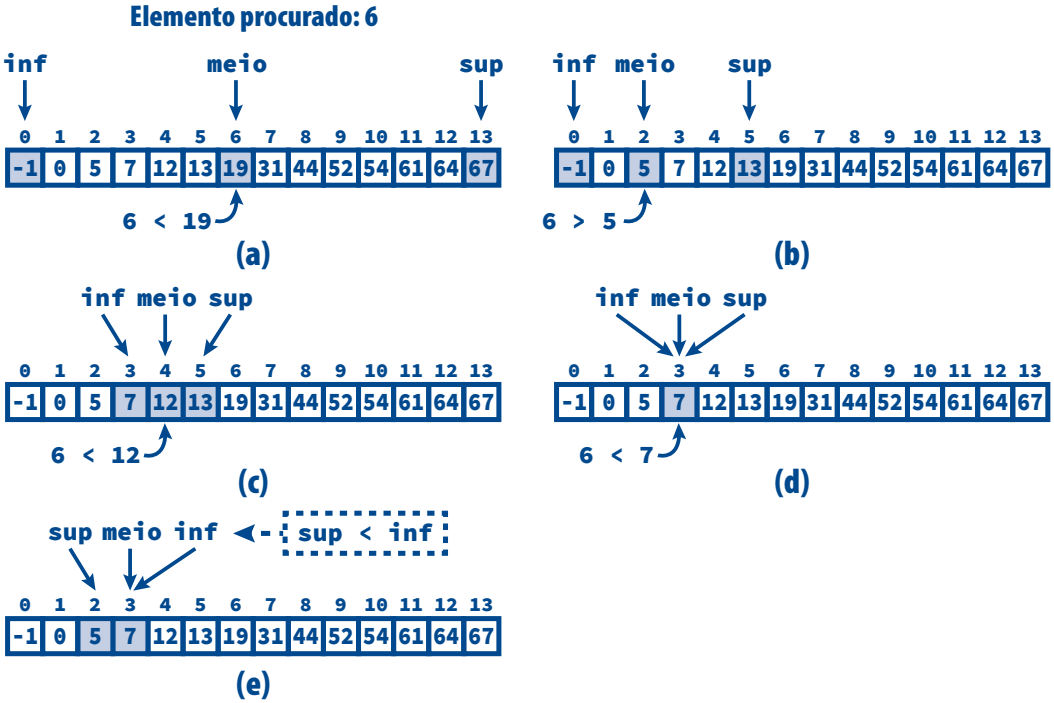

FIGURA 7–8: ELEMENTO NÃO ENCONTRADO NUMA BUSCA BINÁRIA

É importante observar que, na busca sequencial (v. **Seção 7.1.2**), quando um valor é encontrado, ele corresponde sempre ao primeiro elemento do array que possui o referido valor, mas o mesmo não ocorre com busca binária.

Embora possa ser facilmente implementada de modo iterativo, como foi mostrado acima, a busca binária tem natureza recursiva porque o problema é continuamente reduzido a porções menores até ser exaurido ou até que uma solução seja encontrada. Mais especificamente, o array recebido como parâmetro pode ser reduzido à metade, essa metade pode ser então reduzida à metade e assim por diante. O processo continua até que o valor procurado seja encontrado na lista original ou em alguma das metades ou a lista seja reduzida a zero.

A função `BuscaBinariaRec()` mostra como a busca binária pode ser implementada recursivamente:

```
static int BuscaBinariaRec( const tElemento elementos[],
                            int inf, int sup, tElemento valor )
{
   int meio;

   if (inf > sup)
      return -1;

   meio = inf + (sup - inf)/2;

   if (valor == elementos[meio])
      return meio;
   else if (valor < elementos[meio])
      return BuscaBinariaRec(elementos, inf, meio - 1, valor);

   return BuscaBinariaRec(elementos, meio + 1, sup, valor);
}
```

Note que as funções iterativa e recursiva que implementam busca binária apresentam diferentes listas de parâmetros, conforme foi discutido na **Seção 4.6**. Portanto o uso de **static** na função acima e de uma função acionadora, como mostrado abaixo, se fazem necessários.

```
int BuscaBinaria2(const tListaIdx *lista, tElemento elemento)
{
   return BuscaBinariaRec(lista->elementos, 0, lista->nElementos - 1, elemento);
}
```

As funções `BuscaBinaria()` e `BuscaBinariaRec()` apresentam o mesmo custo temporal, mas os custos espaciais são diferentes. O custo temporal dessas funções no pior caso é $\theta(\log n)$ (v. **Seção 6.11.4**) enquanto, nos casos melhor e médio, esse custo é $\theta(1)$. A função iterativa tem custo espacial $\theta(1)$ e a função recursiva tem custo espacial $\theta(\log n)$ (mostre isso).

7.3 Análise de Operações sobre Listas Indexadas

Todas as operações sobre listas implementadas pelas seguintes funções apresentam custo temporal $\theta(1)$ no pior caso, pois essas operações independem do tamanho da entrada (comprimento da lista):

- `IniciaListaIdx()`
- `ComprimentoListaIdx()`
- `ObtemElementoListaIdx()`
- `EstaVaziaListaIdx()`
- `AcrescentaListaIdx()`

As funções `InsereListaIdx()` e `InsereEmOrdemIdx()` apresentam custo temporal $\theta(1)$ no melhor caso, que ocorre quando a inserção se dá ao final da lista. Essas funções apresentam custo temporal $\theta(n)$ nos casos pior e mediano. O pior caso ocorre quando a inserção se dá no início da lista, pois todos os elementos presentes na lista precisam ser movidos para ceder espaço para o novo elemento. O caso mediano ocorre quando a inserção é no meio da lista. Nesse último caso, o custo temporal é $\theta(n/2)$, que é o mesmo que $\theta(n)$, conforme foi visto no **Capítulo 6**.

A função `RemoveListaIdx()` apresenta custo temporal $\theta(1)$ no melhor caso e $\theta(n)$ nos casos pior e mediano. O melhor caso ocorre quando a remoção é efetuada no final da lista, pois, nesse caso, não há movimentação de elementos. O pior caso ocorre quando a remoção é efetuada no início, pois essa operação requer a movimentação de todos os elementos que restam após a remoção. O caso mediano de remoção é quando ela ocorre no meio da lista.

A análise de custo temporal da função `BuscaListaIdx()`, que implementa busca sequencial, foi apresentada na **Seção 7.1.3** e os custos temporais das funções que implementam busca binária foram apresentadas na **Seção 7.2.3**.

É interessante notar que a remoção de um elemento de uma lista ordenada requer que ele seja encontrado. Como a lista é ordenada, pode-se usar busca binária, que tem custo $\theta(\log n)$, para localizar o elemento a ser removido. Mas, mesmo quando esse é o caso, o custo temporal da remoção continua sendo $\theta(n)$, pois $\theta(n) + \theta(\log n) = \theta(n)$, conforme foi visto no **Capítulo 6**.

Com exceção da operação implementada pela função `BuscaBinariaRec()`, todas as demais operações sobre listas indexadas têm custo espacial $\theta(1)$, pois nenhuma delas requer espaço adicional que dependa do tamanho da lista.

7.4 Tratamento de Exceções

Algumas funções que representam operações sobre listas ordenadas precisam assumir certos pressupostos (ou **precondições**) para funcionarem conforme o esperado. Por exemplo, a função `AcrescentaListaIdx()` apresentada na **Seção 7.1.2** assume que há espaço para mais um elemento no array que armazena a lista. Nessa mesma seção, a função `InsereListaIdx()` também assume que o array ainda tem espaço e, além disso, que a posição de inserção existe ou é a última na lista. A **Tabela 7–2** enumera as precondições assumidas pelas funções apresentadas nas **Seções 7.1.2** e **7.2.2**. Funções apresentadas nessas seções que não aparecem nessa tabela não têm precondições.

Função	Precondições
`ObtemElementoListaIdx()`	O índice (posição) do elemento é válido
`InsereListaIdx()`	❒ A posição de inserção é válida ❒ Há espaço disponível para inserção no array
`RemoveListaIdx()`	O índice (posição) do elemento é válido
`AcrescentaListaIdx()`	Há espaço disponível para inserção no array
`InsereEmOrdemIdx()`	Há espaço disponível para inserção no array

Tabela 7–2: Precondições Assumidas por Funções de Processamento de Listas

Uma **exceção** ou **condição de exceção** pode ser definida como o não atendimento das precondições de uma função durante sua execução. Ou seja, exceção é uma tentativa de executar uma operação que simplesmente não pode ser executada. Por exemplo, um elemento pode ser acrescentado a uma lista implementada num array estático apenas quando nele há espaço sobressalente. Quando o acréscimo de um elemento não pode ocorrer, a função `AcrescentaListaIdx()` deve detectar (i.e., **capturar**) e **lançar uma exceção** (i.e., responder adequadamente). Essa resposta pode ser abortar o programa no qual ela é chamada.

Abortar um programa propositadamente pode parecer uma medida drástica, mas, de fato, ela previne um mal maior (i.e., um erro lógico ou um aborto involuntário).

É responsabilidade do trecho de programa no qual é feita uma chamada de função verificar se as precondições dessa função são satisfeitas. Por outro lado, toda função que tem precondições a serem satisfeitas deve verificar se isto realmente ocorre e, quando esse não for o caso, exibir uma mensagem de erro e abortar o programa para que o programador verifique o que há de errado com o trecho de programa no qual foi feita a chamada. Essa

verificação de precondições, e a consequente reação, se for o caso, efetuadas por uma função são denominadas em conjunto **tratamento de exceção**.

A maneira mais simples de implementar tratamento de exceção em C é por meio de uma macro com parâmetros (v. **Seção 2.5.1**), aqui denominada `ASSEGURA`, que pode ser implementada assim:

```
#include <stdio.h>  /* fprintf() */
#include <stdlib.h> /* exit()    */

#define ASSEGURA(condicao, msg) if (!(condicao)) {\
                                  fprintf(stderr, "\n%s\n", msg);\
                                  exit(1); \
                                }
```

Nessa macro, `condicao` representa a condição sendo testada e `msg` é a mensagem que será exibida se essa condição não for satisfeita. A chamada de função `exit(1)` no corpo da macro aborta o programa e, finalmente, a barra invertida (\) é interpretada pelo pré-processador de C como continuação de linha.

Entender o funcionamento dessa macro é simples se você entender como macros são processadas em C (v. **Seção 2.5.1**). Do mesmo modo que constantes simbólicas (macros sem parâmetro), macros com parâmetros são processadas pelo pré-processador de C antes de a compilação iniciar.

Considere, por exemplo, a função `ObtemElementoListaIdx()` apresentada na **Seção 7.1.2**. Conforme se observa na **Tabela 7–2**, essa função assume como precondição que o índice do elemento desejado é válido. Assim, para implementar tratamento de exceção nessa função, deve-se acrescentar a seguinte linha no início do seu corpo:

```
ASSEGURA( indice >= 0 && indice < lista->nElementos, "Elemento inexistente" );
```

Antes do início do processo de compilação, o pré-processador de C expandiria essa macro transformando-a na seguinte instrução:

```
if (!(indice >= 0 && indice < lista->nElementos) ) {
   printf("\n%s", "Elemento inexistente");
   exit(1);
}
```

As funções que implementam operações sobre listas indexadas implementadas com arrays estáticos e apresentadas nas **Seções 7.1.2** e **7.2.2** que requerem tratamento de exceções serão apresentadas com a inclusão desse tratamento a seguir.

```
tElemento ObtemElementoListaIdx(const tListaIdx *lista, int indice)
{
   ASSEGURA( indice >= 0 && indice < lista->nElementos, "Elemento inexistente" );

   return lista->elementos[indice];
}

int InsereListaIdx(tListaIdx *lista, tElemento item, int indice)
{
      /* Se a lista estiver cheia, não há como inserir mais um elemento */
   ASSEGURA (!EstaCheiaListaIdx(lista), "A lista esta cheia");

      /* Verifica se o índice é válido */
   ASSEGURA( indice >= 0 && indice <= lista->nElementos,
             "Posicao de insercao inexistente" );

      /* Abre espaço para o novo elemento */
   for (int i = lista->nElementos - 1; i >= indice; --i)
      lista->elementos[i + 1] = lista->elementos[i];

   lista->elementos[indice] = item; /* Insere o novo elemento */
```

```
   lista->nElementos++; /* O tamanho da lista aumentou */

   return 0;
}

tElemento RemoveListaIdx(tListaIdx *lista, int indice)
{
   tElemento itemRemovido;

         /* Verifica se o índice é válido */
   ASSEGURA( indice >= 0 && indice < lista->nElementos,
             "Posicao de remocao inexistente" );

   itemRemovido = lista->elementos[indice];

      /* Remover um elemento significa mover cada elemento uma posição para   */
      /* trás a partir do próximo elemento adiante daquele que será  removido */
   for (int i = indice; i < lista->nElementos; i++)
      lista->elementos[i] = lista->elementos[i + 1];

   --lista->nElementos; /* O tamanho da lista diminuiu */

   return itemRemovido;
}

int AcrescentaListaIdx(tListaIdx *lista, tElemento elemento)
{
      /* Verifica se é possível acrescentar mais um elemento na lista */
   ASSEGURA (!EstaCheiaListaIdx(lista), "A lista esta cheia");

      /* Acrescenta um novo elemento ao final da lista */
   lista->elementos[lista->nElementos] = elemento;

   ++lista->nElementos; /* O tamanho da lista aumentou */

   return 0;
}

int InsereEmOrdemIdx(tListaIdx *lista, tElemento elemento)
{
   int posicao, /* Posição de inserção do elemento */
       i;

      /* Se a lista estiver cheia, não há como inserir mais um elemento */
   ASSEGURA (!EstaCheiaListaIdx(lista), "A lista esta cheia");

      /* Se a lista estiver vazia, o elemento é */
      /* inserido na primeira posição do array  */
   if (EstaVaziaListaIdx(lista)) {
         /* O elemento será o primeiro da lista */
      lista->elementos[0] = elemento;

      ++lista->nElementos; /* A lista cresceu */

      return 0; /* Serviço completo */
   }

      /* Encontra a posição no array onde o elemento será inserido */
   for (posicao = 0; posicao < lista->nElementos; ++posicao)
      if (lista->elementos[posicao] > elemento)
         break; /* Posição de inserção encontrada */

      /* Abre espaço para o novo elemento */
   for (i = lista->nElementos - 1; i >= posicao; --i)
```

```
        /* Move cada elemento uma posição adiante a partir do  */
        /* elemento que ora se encontra na posição de inserção */
     lista->elementos[i + 1] = lista->elementos[i];

  lista->elementos[posicao] = elemento; /* Insere o novo elemento */
  ++lista->nElementos; /* A lista cresceu */

  return 0;
}
```

7.5 Aplicações de Listas

7.5.1 Representação de Polinômios

Uma aplicação de listas ordenadas é a representação de polinômios. Serão apresentados dois esquemas para representação de polinômios e uma função para somar polinômios ilustrará a utilização de um destes esquemas. Um conjunto completo de operações sobre polinômios deveria incluir, além de soma, outras operações, tais como subtração, multiplicação e divisão.

Um polinômio é uma soma de termos da forma ax^m, em que a é o coeficiente do termo, x é a variável e m é o expoente. Para que seja desenvolvido um esquema de representação de polinômios com listas ordenadas, uma exigência inicial é que os expoentes sejam únicos e estejam em ordem decrescente.

Um polinômio de grau n deve ter a forma geral:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

em que $a_n \neq 0$. Assim uma forma de representar $P(x)$ como uma lista ordenada é por meio de um array unidimensional de comprimento $n + 2$, em que o primeiro elemento é o grau do polinômio e os $n + 1$ elementos restantes representam os coeficientes na ordem decrescente dos expoentes. Desse modo, o polinômio $P(x)$ seria apresentado em forma de lista ordenada como:

$$(n, a_n, a_{n-1}, \ldots, a_0)$$

Utilizando esse esquema, se evita armazenar explicitamente os expoentes dos termos, pois se pode deduzir seus valores conhecendo-se suas posições na lista e do grau do polinômio. Por exemplo, o polinômio:

$$P(x) = 3x^5 + x^3 - 2x + 1$$

seria representado na forma descrita acima pela lista:

$$(5, 3, 0, 1, 0, -2, 1)$$

Numa rápida visualização desta lista, pode-se inferir que o coeficiente do termo de expoente 5 é 3, que não existe termo com expoente 4 e assim por diante.

As operações de soma e subtração são relativamente fáceis de implementar com essa representação. Contudo, existe um inconveniente nessa representação. Considere, por exemplo, o polinômio: $P(x) = x^{1000} - 1$. De acordo com o esquema proposto, esse polinômio seria representado como uma lista com 1002 elementos, sendo que apenas três deles não seriam nulos. Logo um esquema alternativo precisa ser desenvolvido.

A nova proposta é que sejam representados na lista ordenada apenas os coeficientes diferentes de zero. Mas, dessa forma, não seria mais possível deduzir a que expoente corresponde um determinado coeficiente, a não ser que ele seja explicitamente representado. Portanto, se apenas os termos não nulos de um polinômio forem representados numa lista, os expoentes desse polinômio devem também fazer parte da lista. Nesse caso, um polinômio com m termos não nulos é representado por uma lista ordenada de comprimento $2m + 1$, em que o primeiro elemento é o número de elementos não nulos e o restante são pares expoente/coeficiente:

$$(m, e_{m-1}, c_{m-1}, e_{m-2}, c_{m-2}, \ldots, e_0, c_0)$$

Essa nova representação resolve o problema de armazenamento ocioso dos coeficientes nulos do polinômio $P(x) = x^{1000} - 1$, que seria então representado por: *(2, 1000, 1, 0, –1)*, isto é, por uma lista de apenas *5* elementos, ao invés de *1002* elementos como antes. No entanto, este esquema é pior que o anterior para polinômios completos ou com poucos termos ausentes. Por exemplo, o polinômio $x^3 + 2x^2 + 3x + 1$ apareceria como *(3, 1, 2, 3, 1)* de acordo com o primeiro esquema e *(4, 3, 1, 2, 2, 1, 3, 0, 1)* pelo segundo esquema. No segundo esquema, seria gasto quase o dobro do espaço ocupado no primeiro esquema. Então, qual é a melhor representação? A escolha mais sensata é a segunda, pois, no pior caso (polinômio completo), seria requerido menos do dobro do armazenamento utilizado pela primeira representação. Por outro lado, num caso como o do polinômio $x^{1000} - 1$, seria gasto mais de 200 vezes mais espaço no primeiro esquema do que no segundo.

Como ilustração, será apresentada abaixo uma função em C que adiciona dois polinômios representados conforme o segundo esquema visto. Esse exemplo serve ainda para mostrar uma limitação dos arrays para representação de dados, pois, como eles são variáveis estruturadas homogêneas, não permitem elementos de tipos diferentes. Mas, num polinômio, os expoentes são inteiros não negativos, enquanto os coeficientes podem ser valores reais quaisquer. Aqui, neste exemplo, fez-se a opção por utilizar coeficientes e expoentes inteiros em virtude dessa restrição.

A função `SomaPol()` soma dois polinômios representados de acordo com o último esquema apresentado acima. Nessa função, os parâmetros `A[]` e `B[]` representam os polinômios que serão somados, ao passo que `C[]` representa a soma desses polinômios.

```
void SomaPol(const int A[], const int B[], int C[])
{
   int m = A[0],  /* Número de termos do polinômio A */
       n = B[0],  /* Número de termos do polinômio B */
       p = 1,     /* Um índice do polinômio A */
       q = 1,     /* Um índice do polinômio B */
       r = 1;     /* Um índice do polinômio C */

      /* Soma os coeficientes dos termos de mesmo expoente e */
      /* armazena os termos que não possuem o mesmo expoente */
   while ( (p <= 2*m) && (q <= 2*n) ) {
         /* Verifica se os expoentes são iguais */
      if ( A[p] == B[q] ) {
            /* Os expoentes são iguais. Portanto os coeficientes são somados */
         C[r + 1] = A[p + 1] + B[q + 1];

            /* O termo só será armazenado em C[] se o  */
            /* resultado da soma for diferente de zero */
         if ( C[r + 1] ) {
            C[r] = A[p]; /* Armazena expoentes em C[] */
            r += 2; /* Avança para o próximo termo de C[] */
         }

         p += 2; /* Avança para o próximo termo de A[] */
         q += 2; /* Avança para o próximo termo de B[] */

         /* Armazena em C[] o termo que tem maior */
         /* expoente e avança-se um termo adiante */
      } else if ( A[p] < B[q] ) {
            /* O termo corrente de B[] é maior do que o termo */
            /* corrente de A[] e será armazenado em C[]       */
         C[r + 1] = B[q + 1];
         C[r] = B[q];
```

```
         q += 2; /* Passa para o próximo termo de B[] */
         r += 2; /* Passa para o próximo termo de C[] */
      } else  { /* A[p] > B[q]. Armazena novo termo em C */
            /* O termo corrente de A[] é maior do que o termo */
            /* corrente de B[] e será armazenado em C[]       */
         C[r + 1] = A[p + 1];
         C[r] = A[p];

         p += 2; /* Passa para o próximo termo de A[] */
         r += 2; /* Passa para o próximo termo de C[] */
      } /* if */
   } /* while */

   /*                                                  */
   /* Neste ponto, uma das seguintes condições causaram */
   /* o encerramento do laço while acima:              */
   /* (1) Todos os termos de ambos os polinômios foram  */
   /*     levados em consideração. Neste caso, nenhum   */
   /*     dos laços a seguir será executado.            */
   /* (2) Todos os termos de apenas um dos polinômios   */
   /*     foram levados em consideração. Neste caso,    */
   /*     apenas um dos laços a seguir será executado.  */
   /*                                                   */

      /* Se houver termos em A[] que não foram levados */
      /* em consideração, armazena-os em C[]           */
   while ( p <= 2*m ) {
      C[r] = A[p];
      C[r+1] = A[p+1];
      p += 2;
      r += 2;
   }

      /* Se houver termos em B[] que não foram levados */
      /* em consideração, armazena-os em C[]           */
   while ( q <= 2*n ) {
      C[r] = B[q];
      C[r+1] = B[q+1];
      q = q+2;
      r = r+2;
   }

      /* Armazena o número de termos da soma em C[] */
   C[0] = r / 2;
}
```

Polinômios podem ainda ser representados utilizando listas encadeadas, conforme será visto na **Seção 10.7.3**.

7.5.2 Representação de Matrizes Esparsas

Matrizes são abstrações matemáticas que aparecem em muitas situações reais. Aqui, o interesse será o estudo de meios para representação de matrizes de modo que operações sobre elas possam ser executadas eficientemente.

Uma **matriz** é em geral constituída de n linhas e m colunas de números, o que se denota por $n \times m$. Uma matriz na qual o número de linhas é igual ao número de colunas é uma **matriz quadrada**, enquanto uma matriz que contém um número muito grande de zeros é chamada de **matriz esparsa**. A **Figura 7–9** apresenta um exemplo de matriz quadrada esparsa $M_{5\times5}$.

$$M_{5 \times 5} = \begin{vmatrix} 10 & 9 & 0 & 0 & 0 \\ 0 & 0 & 3 & 8 & 0 \\ 6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{vmatrix}$$

Figura 7–9: Exemplo de Matriz Esparsa

Na realidade, a definição de matriz esparsa não é precisa, no sentido de que não existe um limite definido de zeros que uma matriz deve conter para ser classificada como esparsa ou não. No entanto, intuitivamente, pode-se decidir se uma é matriz esparsa ou não. A matriz do exemplo dado foi classificada como esparsa por possuir apenas 6 elementos não nulos do total de 25 elementos da matriz.

A maneira mais natural de representação de matrizes em programação é por meio de arrays bidimensionais. Essa representação convencional, no entanto, pode gerar um desperdício muito grande de memória no caso de matrizes esparsas, de modo que é preciso elaborar uma forma de representação alternativa para matrizes esparsas que armazene apenas os coeficientes não nulos da matriz.

Como a cada elemento de uma matriz corresponde um par de índices *(linha, coluna)* e vice-versa, pode-se levar em consideração o armazenamento de matrizes esparsas em forma de triplas:

(linha, coluna, valor)

em que seriam considerados apenas os valores não nulos da matriz. Essas triplas poderiam então ser armazenadas num array bidimensional $(n+1) \times 3$, em que n é o número de termos não nulos da matriz. Na primeira linha dessa matriz seria armazenado o número de linhas, o número de colunas e o número de termos não nulos da matriz.

É útil, ainda, manter uma certa ordenação entre os elementos nesta representação. Aqui será adotada a ordenação que parece ser a mais natural: as linhas são colocadas em ordem crescente e, para cada linha, as colunas também são colocadas em ordem crescente. Com essa representação, a matriz da **Figura 7–9** apareceria em um array `A[7][3]` como na **Figura 7–10**.

```
     [0] [1] [2]
A[0] 5   5   6
A[1] 1   1   10
A[2] 1   2   9
A[3] 2   3   3
A[4] 2   4   8
A[5] 3   1   6
A[6] 4   4   4
```

Figura 7–10: Representação de Matriz Esparsa

Os elementos `A[0][0]`, `A[0][1]` e `A[0][2]` da **Figura 7–10** representam o número de linhas, o número de colunas e o número de termos não nulos da matriz, respectivamente.

A seguir, serão desenvolvidos algoritmos para executar operações sobre matrizes representadas segundo esse último esquema.

Uma das operações comuns sobre uma matriz é o cálculo de sua matriz transposta. Os elementos t_{ij} da transposta de uma matriz A são os elementos a_{ji} dessa matriz. Ou seja, na matriz transposta, as linhas são formadas pelas colunas da matriz original e as colunas são formadas pelas linhas da matriz original. A matriz transposta da matriz da **Figura 7–9** seria aquela apresentada na **Figura 7–11**.

```
      [0] [1] [2]
T[0]   5   5   6
T[1]   1   1  10
T[2]   1   3   6
T[3]   2   1   9
T[4]   3   2   3
T[5]   4   2   8
T[6]   4   4   4
```

Figura 7–11: Transposta de uma Matriz Esparsa

A primeira ideia para um algoritmo que calcula a transposta de uma matriz representada de acordo com o esquema descrito acima seria:

1. Para cada linha *i*
   1.1 Armazene cada elemento *(i, j, valor)* no elemento *(j, i, valor)* da transposta

O problema desse algoritmo é que os elementos não serão armazenados diretamente na ordem em que deverão permanecer, a não ser que sejam feitas inserções com o deslocamento de alguns elementos. Por exemplo, a tripla `(2, 1, 9)` deveria ser armazenada depois da tripla `(1, 3, 6)` na transposta, mas não é isso que ocorre quando o algoritmo acima é seguido. Assim, para colocarem-se os elementos diretamente na ordem em que deverão permanecer, esse algoritmo deve ser modificado para:

1. Para cada elemento na coluna *j*
   1.1 Armazene o elemento *(i, j, valor)* no elemento *(j, i, valor)* da transposta

A função `Transpoe()` apresentada a seguir é uma tradução desse último algoritmo em C. Nessa função, `A[][]` é a matriz que se deseja calcular a transposta e `T[][]` armazenará o resultado desse cálculo.

```
void Transpoe(int A[][3], int T[][3])
{
   int m = A[0][0], /* Número de linhas de A[][] */
       n = A[0][1], /* Número de colunas de A[][] */
       t = A[0][2], /* Número de elementos não nulos de A[][] */
       p, /* Uma linha de A[][] */
       q, /* Uma linha de T[][] */
       col; /* Uma coluna de A[][] */

   T[0][0] = n; /* Número de linhas de T[][] */
   T[0][1] = m; /* Número de colunas de T[][] */
   T[0][2] = t; /* Número de elementos não nulos de T[][] */

      /* Se matriz for nula, não há o que fazer aqui */
   if (t <= 0)
      return;

   q = 1; /* Apontador da próxima tripla em T[][] */

      /* Efetua a transposição por colunas */
   for (col = 1; col <= n; col++)
      for (p = 1; p <= t; p++)
         if (A[p][1] == col) {
               /* Coluna correta: armazena tripla em T[][] */
            T[q][0] = A[p][1];
            T[q][1] = A[p][0];
            T[q][2] = A[p][2];
            q++;
         }
}
```

Outra operação a ser considerada aqui é o produto de duas matrizes esparsas. Sejam $A_{m \times n}$ e $B_{n \times p}$ duas matrizes, então o produto $A \times B$ é a matriz $C_{m \times p}$, cujos elementos são dados por:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

com $1 \leq i \leq m$, $1 \leq j \leq p$ e $1 \leq k \leq n$.

Deve-se observar inicialmente que o produto de duas matrizes esparsas pode estar longe de ser uma matriz esparsa. Por exemplo, o produto de matrizes esparsas apresentado na **Figura 7–12** não resulta numa matriz esparsa.

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 2 \end{bmatrix}$$

Figura 7–12: Produto de Matrizes Esparsas

Considerando o esquema de representação de matrizes esparsas visto acima, um algoritmo para calcular os elementos da matriz produto $C = A \times B$, colocando-os em seus devidos lugares (i.e., sem ter que mover os elementos antes computados), deveria deixar fixa uma linha de A e encontrar todos os elementos de cada coluna j de B, $j = 1, 2, \ldots, p$. Mas, para encontrar todos os elementos de uma coluna j de B, deve-se acessar toda a matriz B. Isso pode ser evitado calculando-se a transposta de B, na qual todos os elementos de cada coluna são colocados consecutivamente. A implementação de uma função que realiza a multiplicação de duas matrizes seguindo esse raciocínio é deixada como exercício para o leitor, que poderá comparar sua solução com aquela que se encontra no site dedicado ao livro na internet.

7.6 Exemplos de Programação

7.6.1 Gerenciando uma Lista de Compras com Sensibilidade

Preâmbulo: Numa **interação dirigida por menus**, quando as opções oferecidas para o usuário dependem do estado de interação dele com o programa diz-se que o menu é **sensível ao contexto**. Praticamente, qualquer aplicativo moderno utiliza esse princípio básico de interação com o usuário. Por exemplo, em qualquer editor de texto decente que você utilize, quando você seleciona uma porção de texto, há uma opção de menu que lhe permite copiar ou recortar o texto selecionado, mas, se não houver texto selecionado, essas opções estarão indisponíveis, o que é indicado pelo fato de elas aparecerem esmaecidas no respectivo menu. Outro exemplo comum: quando você clica com o botão direito do mouse sobre um ícone num sistema operacional com interface gráfica, o menu que aparece depende do tipo de objeto selecionado. Ou seja, quando você clica com o botão direito do mouse sobre um arquivo de texto, uma das opções que aparecem no menu é imprimir. Por outro lado, quando você clica sobre um ícone associado a um arquivo de vídeo, essa opção não aparece porque não faz sentido imprimir um arquivo dessa natureza. Aliás, muitos autores denominam esse último tipo de menu exatamente **menu de contexto**.

Problema: Escreva um programa para gerenciamento de uma lista de compras. O programa deve implementar uma interação dirigida por menu de tal modo que os menus e as opções disponíveis para o usuário dependam do status da lista. Isto é, o menu deve apresentar como opções disponíveis apenas aquelas operações sobre a lista que realmente estejam disponíveis. Por exemplo, se a lista estiver vazia, as únicas opções disponíveis para o usuário são acréscimo de item e encerramento do

programa, porque, nesse caso, não faz sentido que a opção que remove um item esteja disponível, já que a lista está vazia.

Solução:

As seguintes definições de constantes simbólicas e tipos serão utilizadas na implementação deste programa:

```
#define MAX_ITEM   10 /* Número máximo de caracteres num item */
#define MAX_ITENS  20 /* Número máximo de itens numa lista */

typedef char tItemCompra[MAX_ITEM]; /* Tipo de cada elemento da lista */

   /* Tipo que representa as listas */
typedef struct {
           int         nItens; /* Número de itens na lista */
           tItemCompra itens[MAX_ITENS]; /* Os itens */
        } tListaCompras;
```

A função `ApresentaMenu()` exibe na tela o menu de opções do programa de acordo com o conteúdo da lista recebida como parâmetro e, portanto, é sensível ao contexto. Essa função retorna o endereço de um string cujos caracteres representam as opções que o usuário poderá escolher.

```
char *ApresentaMenu( const tListaCompras *lista )
{
   static char opcoes[MAX_OPCOES + 1];

      /* O array opcoes[] armazenará um string cujos caracteres representam  */
      /* as opções que o usuário poderá escolher. Aqui, ele recebe as opções */
      /* de encerramento que são sempre disponíveis. Isso não pode ser feito */
      /* como iniciação pois o array tem duração fixa                        */
   strcpy(opcoes, "Ee");

   printf( "\n\n\t*** Opcoes ***" ); /* Apresenta o cabeçalho do menu */

      /*** Apresenta o menu de acordo com o estado da lista ***/

   if (!EstaCheiaListaIdx(lista)) {
         /* Se a lista não está cheia, o usuário pode acrescentar mais itens */
      printf( "\n\t[A]crescenta" );
      strcat(opcoes, "Aa");

      if (ComprimentoListaIdx(lista) > 1) {
            /* Se a lista tiver mais de um elemento, o usuário pode inserir */
         printf( "\n\t[I]nsere" );
         strcat(opcoes, "Ii");
      }
   }

      /* Se a lista não estiver vazia, o usuário pode remover e modificar */
   if (!EstaVaziaListaIdx(lista)) {
      printf( "\n\t[R]emove" );
      printf( "\n\t[M]odifica" );

      strcat(opcoes, "RrMm");
   }

      /* Apresenta a última opção e o prompt */
   printf( "\n\t[E]ncerra o programa\n\nEscolha sua opcao >>> " );

   return opcoes;
}
```

A função **main()** do programa que implementa a interação dirigida por menu solicitada é apresentada a seguir:

```
int main(void)
{
   int           opcao, /* Uma opção escolhida pelo usuário */
                 pos, /* Uma posição na lista */
                 nItens; /* Número de itens da lista */
   tListaCompras lista; /* A lista de compras */
   tItemCompra   umItem; /* Um item da lista de compras */
   char         *removido; /* Um item removido da lista */
   char         *opcoes; /* Aponta para o string contendo as   */
                         /* opções que o usuário pode escolher */

   IniciaListaIdx(&lista); /* Inicia a lista de compras  */

      /* Apresenta o programa ao usuário e explica seu funcionamento */
   printf( "\n\t>>> Este programa gerencia uma lista de compras\n" );

   while (1) {
      ApresentaLista(&lista); /* Apresenta a lista */

         /* Alerta o usuário quando a lista está cheia */
      if (EstaCheiaListaIdx(&lista))
         printf("\n*** A lista esta' cheia ***\n");

         /* Apresenta o menu e obtém as opções disponíveis */
      opcoes = ApresentaMenu(&lista);

         /* Lê a opção do usuário (o prompt já foi apresentado ) */
      opcao = LeOpcao(opcoes);

         /* É melhor tratar a opção de encerramento aqui */
      if (opcao == 'E' || opcao == 'e')
         break; /* Usuário quer encerrar o programa */

         /* Processa as demais opções */
      switch (opcao) {
         case 'A': /* Acrescenta */
         case 'a':
            ASSEGURA(!EstaCheiaListaIdx(&lista), "Lista cheia");

            printf("\nDigite o nome do item >>> ");
            LeString( (char *) umItem, MAX_ITEM );
            AcrescentaItem(&lista, umItem);
            break;
         case 'I': /* Insere */
         case 'i':
            ASSEGURA(!EstaCheiaListaIdx(&lista), "Lista cheia");

            nItens = ComprimentoListaIdx(&lista);

            if (nItens > 1) {
               printf( "\nInforme a posicao entre 1 e %d >>> ", nItens );
               pos = LeNaturalPositivo();
            else
               pos = 1;

            if (pos > nItens) {
               printf( "\n>>> A posicao deveria ser menor "
                       "do que %d. Nao havera' insercao.", nItens + 1 );
            } else {
                  /* A correção a seguir é necessária porque usuário normal conta */
                  /* a partir de 1, mas programador de C conta a partir de zero   */
               --pos;
```

```
            printf("\nDigite o nome do item >>> ");
            LeString( (char *) umItem, MAX_ITEM );

            InsereItem(&lista, umItem, pos);
         }
         break;
      case 'R': /* Remove */
      case 'r':
         ASSEGURA(!EstaVaziaListaIdx(&lista), "Lista vazia");

         nItens = ComprimentoListaIdx(&lista);
         printf( "\nInforme a posicao entre 1 e %d >>> ", nItens );
         pos = LeNaturalPositivo();

         if (pos > nItens) {
            printf( "\n>>> A posicao deveria ser menor "
                    "do que %d. Nao havera' remocao.", nItens + 1 );
         } else {
               /* A correção a seguir é necessária porque usuário normal conta */
               /* a partir de 1, mas programador de C conta a partir de zero   */
            --pos;

            removido = RemoveItem(&lista, pos);

            printf( "\n>>> \"%s\" foi removido da lista\n", removido );
         }
         break;
      case 'M': /* Modifica */
      case 'm':
         ASSEGURA(!EstaVaziaListaIdx(&lista), "Lista vazia");

         nItens = ComprimentoListaIdx(&lista);
         printf( "\nInforme a posicao entre 1 e %d >>> ", nItens );
         pos = LeNaturalPositivo();

         if (pos > nItens) {
            printf( "\n>>> A posicao deveria ser menor "
                    "do que %d. Nao havera' modificacao.", nItens + 1 );
         } else {
               /* A correção a seguir é necessária porque usuário normal conta */
               /* a partir de 1, mas programador de C conta a partir de zero   */
            --pos;

            printf( "\nDigite o novo nome do item \""
                    "%s\" >>> ", ObtemItem(&lista, pos) );
            LeString( (char *) umItem, MAX_ITEM );

            AlteraItem(&lista, umItem, pos);
         }
         break;
      default:
         ASSEGURA(1, "Ocorreu um erro impossivel");
         break;
    }
  }

    /* Despede-se do usuário */
  printf( "\n\t>>> Obrigado por usar este programa. Boas compras!\n" );

  return 0;
}
```

A função `ApresentaLista()` chamada no corpo de **main()** é definida como:

```
void ApresentaLista(const tListaCompras *aLista)
{
   int tamanho = ComprimentoListaIdx(aLista);

   if (!tamanho) {
      printf("\n\t*** Lista vazia ***\n");
      return;
   }

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

   for (int i = 0; i < tamanho; ++i)
      printf( "\n\t   %2d. %s", i + 1, ObtemItem(aLista, i) );
}
```

A implementação de lista utilizada nesta implementação é basicamente a mesma apresentada na **Seção 7.1.2**, mas algumas funções precisaram ser alteradas para levar em consideração o fato de o tipo de cada elemento da lista ser um string. Essas funções serão apresentadas a seguir.

```
char *ObtemItem(const tListaCompras *lista, int pos)
{
   ASSEGURA( pos >= 0 && pos < lista->nItens, "Elemento inexistente" );
   return (char *) lista->itens[pos];
}

void InsereItem(tListaCompras *lista, tItemCompra elemento, int pos)
{
   ASSEGURA(!EstaCheiaListaIdx(lista), "A lista esta cheia");

      /* Abre espaço para o novo elemento */
   for (int i = lista->nItens; i >= pos; --i)
      strcpy( (char *)lista->itens[i + 1], (char *)lista->itens[i] );

   strcpy((char *)lista->itens[pos], (char *)elemento); /* Insere o novo elemento */

   ++lista->nItens; /* O tamanho da lista aumentou */
}

char *RemoveItem(tListaCompras *lista, int pos)
{
   int i;
   static tItemCompra itemRemovido;

   ASSEGURA(!EstaVaziaListaIdx(lista), "A lista esta' vazia");
   strcpy( (char *)itemRemovido, (char *)lista->itens[pos] );

      /* Remover um elemento significa mover cada elemento uma posição para   */
      /* trás a partir do próximo elemento adiante daquele que será  removido */
   for (i = pos; i < lista->nItens - 1; i++)
      strcpy( (char *)lista->itens[i], (char *)lista->itens[i + 1] );

   --lista->nItens; /* O tamanho da lista diminuiu */

   return (char *)itemRemovido;
}

int BuscaItem(const tListaCompras *lista, tItemCompra elemento)
{
   for (int i = 0; i < lista->nItens; ++i)
      if (!strcmp(lista->itens[i], elemento))
         return i; /* Elemento foi encontrado */
```

```
   return -1; /* Elemento não foi encontrado */
}
void AcrescentaItem(tListaCompras *lista, tItemCompra elemento)
{
   ASSEGURA(!EstaCheiaListaIdx(lista), "A lista esta cheia");
      /* Acrescenta um novo elemento ao final da lista */
   strcpy( (char *)lista->itens[lista->nItens], (char *)elemento );
   ++lista->nItens; /* O tamanho da lista aumentou */
}
void AlteraItem(tListaCompras *lista, tItemCompra valor, int pos)
{
   ASSEGURA( pos >= 0 && pos < lista->nItens, "Elemento inexistente" );
   strcpy((char *)lista->itens[pos], (char *)valor);
}
```

Os protótipos das funções `ObtemElementoListaIdx()` e `RemoveListaIdx()` foram alterados com relação às funções com mesmos nomes apresentadas na **Seção 7.1.2** porque os elementos da lista são strings e a linguagem C não permite que uma função retorne um array (v. **Seção 3.5**).

Exemplo de execução do programa:

```
       >>> Este programa gerencia uma lista de compras
        *** Lista vazia ***
        *** Opcoes ***
        [A]crescenta
        [E]ncerra o programa
Escolha sua opcao >>> a
Digite o nome do item >>> Arroz
        *** Lista ***
            1. Arroz
       *** Opcoes ***
        [A]crescenta
        [R]emove
        [M]odifica
        [E]ncerra o programa
Escolha sua opcao >>> a
Digite o nome do item >>> Feijao
        *** Lista ***
            1. Arroz
            2. Feijao
        *** Opcoes ***
        [A]crescenta
        [I]nsere
        [R]emove
        [M]odifica
        [E]ncerra o programa
Escolha sua opcao >>> i
```

```
Informe a posicao entre 1 e 2 >>> 2
Digite o nome do item >>> Acucar
        *** Lista ***
            1. Arroz
            2. Acucar
            3. Feijao
        *** Opcoes ***
        [A]crescenta
        [I]nsere
        [R]emove
        [M]odifica
        [E]ncerra o programa
Escolha sua opcao >>> m
Informe a posicao entre 1 e 2 >>> 2
Digite o novo nome do item "Acucar" >>> Adocante
       *** Lista ***
            1. Arroz
            2. Adocante
            3. Feijao
        *** Opcoes ***
        [A]crescenta
        [I]nsere
        [R]emove
        [M]odifica
        [E]ncerra o programa
Escolha sua opcao >>> r
Informe a posicao entre 1 e 3 >>> 1
>>> "Arroz" foi removido da lista
       *** Lista ***
            1. Adocante
            2. Feijao
       *** Opcoes ***
        [A]crescenta
        [I]nsere
        [R]emove
        [M]odifica
        [E]ncerra o programa
Escolha sua opcao >>> e
       >>> Obrigado por usar este programa. Boas compras!
```

7.6.2 Lista Ordenada Armazenada em Arquivo

Preâmbulo: Nesta seção e nos próximos capítulos, serão apresentados vários exemplos que irão processar os mesmos dados de maneiras diversas. Os dados constituem uma turma escolar surreal formada pelo rei Henrique VIII e suas seis esposas. O minúsculo banco de dados formado por membros da dinastia Tudor é armazenado num arquivo de texto, denominado **`Tudor.txt`** e tem seu seguinte conteúdo apresentado na Tabela 7–4.

Henrique VIII	1029	9.5	9.0
Catarina Aragon	1014	5.5	6.5
Ana Bolena	1012	7.8	8.0
Joana Seymour	1017	7.7	8.7
Ana de Cleves	1022	4.5	6.0
Catarina Howard	1340	6.0	7.7
Catarina Parr	1440	4.0	6.0

Tabela 7–3: Conteúdo do Arquivo Tudor.txt

Para tornar os exemplos mais palpáveis, supõe-se que o conteúdo do arquivo representa informações referentes a uma turma fictícia de uma disciplina imaginária. Cada linha desse arquivo (**registro**) é dividida em quatro partes (**campos**) separadas por caracteres de tabulação. Outros tipos de separadores, como vírgula ou ponto e vírgula, podem ser usados. Tabulação foi escolhida como separador de campos porque facilita a visualização do conteúdo do arquivo. Por outro lado, o uso de tabulação pode causar a impressão de que as linhas são do mesmo tamanho. Mas, observando-se atentamente, verifica-se que elas têm tamanhos variados.

Os campos de cada registro são interpretados como mostra a **Tabela 7–4**.

Campo	Interpretação	Comentário
1	Nome do aluno	String limitado em 20 caracteres (sem incluir o caractere terminal)
2	Matrícula do aluno	String contendo exatamente 4 caracteres (sem incluir o terminal de string) que constituem a matrícula do aluno
3	Primeira nota	Valor real compreendido entre `0.0` e `10.0` (inclusive)
4	Segunda nota	Idem

Tabela 7–4: Campos do Arquivo Tudor.txt

Considerando as interpretações de campos apresentadas na **Tabela 7–4**, na primeira linha da **Tabela 7–3**, por exemplo, o nome do aluno é `Henrique VIII`, sua matrícula é `1029`, sua primeira nota é `9.5` e sua segunda nota é `9.0`.

Problema: (a) Implemente uma lista para armazenamento de registros de alunos conforme foi descrito no preâmbulo. A lista deve ser ordenada em ordem crescente dos valores do campo que representa o nome do aluno e a matrícula deve ser considerada uma chave primária, de modo que cada uma delas deve ser única no conjunto de registros. (b) Escreva uma função que lê o conteúdo do arquivo `Tudor.txt` descrito no preâmbulo e armazena-o numa lista do tipo solicitado no item (a). (c) Escreva uma função que armazena o conteúdo de uma lista do tipo solicitado no item (a) num arquivo binário denominado `Tudor.bin`. (d) Escreva um programa que apresenta um menu com as seguintes opções para o usuário:

```
[1] Acrescenta um aluno
[2] Remove um aluno
[3] Consulta dados de aluno
[4] Altera dados de aluno
[5] Exibe a turma na tela
[6] Encerra o programa
```

Então, enquanto o usuário não escolher a opção de encerramento, o programa deve ler a opção escolhida pelo usuário e executar a operação correspondente. Antes de encerrar, o programa deve gravar o conteúdo de uma lista do tipo solicitado no item (a) num arquivo binário denominado `Tudor.bin`.

Solução de (a): As seguintes definições de constantes simbólicas e tipos serão utilizadas na implementação da lista.

```
#define MAX_NOME      20 /* Número máximo de caracteres num nome */
#define TAM_MATR      4  /* Número de caracteres numa matrícula  */
#define MAX_ELEMENTOS 20 /* Número máximo de alunos na lista  */

typedef char tNome[MAX_NOME + 1];      /* Tipo do nome */
typedef char tMatricula[TAM_MATR + 1]; /* Tipo da matrícula */

typedef struct {
           tNome      nome;   /* Nome do aluno */
           tMatricula matr;   /* Sua matrícula */
           double     n1, n2; /* Suas notas    */
        } tAluno;

typedef struct {
           tAluno alunos[MAX_ELEMENTOS];
           int    nAlunos;
        } tListaAlunos;
```

A seguir serão apresentadas apenas as funções de implementação do tipo de lista solicitado que diferem daquelas implementadas nas **Seções** 7.1 e 7.2. Portanto ficam de fora desse rol as funções `EstaCheiaListaIdx()`, `IniciaListaIdx()`, `ComprimentoListaIdx()`, `ObtemElementoListaIdx()`, `RemoveListaIdx()` e `EstaVaziaListaIdx()`. Assim o que resta apenas é apresentar as funções: `InsereEmOrdemIdx()`, `BuscaBinariaNome()` e `BuscaSequencialMatr()`.

```
int InsereEmOrdemIdx(tListaAlunos *lista, const tAluno *aluno)
{
   int posicao, /* Posição de inserção do elemento */
       i;

      /* Se a lista estiver cheia, não há como inserir mais um elemento */
   if (EstaCheiaListaIdx(lista))
      return 1;

      /* Se a lista estiver vazia, o elemento é */
      /* inserido na primeira posição do array  */
   if (EstaVaziaListaIdx(lista)) {
      lista->alunos[0] = *aluno; /* O elemento será o primeiro da lista */

      ++lista->nAlunos; /* A lista cresceu */

      return 0; /* Serviço completo */
   }

      /* Encontra a posição na lista onde o elemento será inserido */
   for (posicao = 0; posicao < lista->nAlunos; ++posicao) {
      /*                                                          */
      /* Se for encontrado um elemento da lista cujo valor é maior do que o  */
      /* valor do elemento a ser inserido, é na posição atual desse elemento */
      /* que o elemento recebido como parâmetro será inserido. Se ele não    */
      /* for encontrado, a inserção será efetuada ao final da lista.         */
```

```
      if ( strcmp(lista->alunos[posicao].nome, aluno->nome) > 0 )
         break; /* Posição de inserção encontrada */
   }

   /* Neste ponto, a variável 'posicao' armazena o */
   /* índice no qual o novo elemento será inserido */

      /* Abre espaço para o novo elemento */
   for (i = lista->nAlunos - 1; i >= posicao; --i)
         /* Move cada elemento uma posição adiante a partir do  */
         /* elemento que ora se encontra na posição de inserção */
      lista->alunos[i + 1] = lista->alunos[i];

   lista->alunos[posicao] = *aluno; /* Insere o novo elemento */

   ++lista->nAlunos; /* A lista cresceu */

   return 0;
}
```

A função `InsereEmOrdemIdx()` retorna `0`, quando o elemento é realmente inserido, ou `1`, quando não ocorre inserção. A principal diferença entre essa função e aquela de idêntica denominação apresentada na **Seção 7.1** é que ela usa **strcmp()** para comparar as chaves de ordenação para encontrar o local de inserção do elemento recebido como parâmetro, visto que essa chave é um string.

```
int BuscaBinariaNome(const tListaAlunos *lista, tNome nomeProcurado)
{
   int inf, /* Limite inferior da busca */
       sup, /* Limite superior da busca */
       meio; /* Meio do intervalo */

      /* limites inferior e superior iniciais */
   inf = 0;
   sup = lista->nAlunos - 1;

      /* Efetua a busca binária */
   while (inf <= sup) {
      meio = inf + (sup - inf)/2; /* Calcula o meio do intervalo */

         /* Verifica se o elemento se encontra no meio do intervalo */
      if (!strcmp(lista->alunos[meio].nome, nomeProcurado))
         return meio; /* Elemento encontrado */

         /* Ajusta o intervalo de busca */
      if (strcmp(nomeProcurado, lista->alunos[meio].nome) < 0)
         sup = meio - 1;
      else
         inf = meio + 1;
   }

   return -1; /* Elemento não foi encontrado */
}

int BuscaSequencialMatr(const tListaAlunos *lista, tMatricula matrProcurada)
{
   for (int i = 0; i < lista->nAlunos; ++i)
      if (!strcmp(lista->alunos[i].matr, matrProcurada))
         return i; /* Elemento foi encontrado */

   return -1; /* Elemento não foi encontrado */
}
```

As funções `BuscaBinariaNome()` e `BuscaSequencialMatr()` usam a função **strcmp()** para comparar chaves de ordenação. É importante observar que, apesar de a lista deste exemplo ser ordenada, tanto busca binária quanto busca sequencial se fazem necessárias neste programa, pois ele efetua buscas não apenas usando como chave o campo `nome`, que é chave de ordenação, como também o campo `matr`, que não é chave de ordenação.

Solução de (b):

A função `LeArquivo()` apresentada a seguir lê um arquivo com a estrutura do arquivo `Tudor.txt` apresentado no preâmbulo e armazena seu conteúdo numa lista do tipo definido no item (a).

```
void LeArquivo( const char *arq, tListaAlunos *lista )
{
   char    linha[TAMANHO_LINHA + 1];
   FILE   *stream;
   tAluno  umAluno;
   char   *str;

   stream = fopen(arq, "r"); /* Abre arquivo em formato texto para leitura */

      /* Se o arquivo não puder ser aberto, nada mais pode ser feito */
   ASSEGURA(stream, "Arquivo nao pode ser aberto");

      /* Lê cada linha do arquivo e a usa para construir um registro. */
      /* Então, o registro é inserido na lista ordenada.              */
   while ( fgets(linha, TAMANHO_LINHA + 1, stream) ) {
         /* Obtém o nome e acrescenta-o ao registro*/
      str = strtok(linha, "\t\n");
      strcpy(umAluno.nome, str);

         /* Obtém a matrícula e acrescenta-a ao registro*/
      str = strtok(NULL, "\t\n");
      strcpy(umAluno.matr, str);

         /* Obtém a 1a. nota e acrescenta-a convertida em double ao registro */
      str = strtok(NULL, "\t\n");
      umAluno.n1 = strtod(str, NULL);

         /* Obtém a 2a. nota e acrescenta-a convertida em double ao registro */
      str = strtok(NULL, "\t\n");
      umAluno.n2 = strtod(str, NULL);

      ASSEGURA( !InsereEmOrdemIdx(lista, &umAluno),
                "Nao foi possivel inserir um elemento na lista" );
   }

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

Para ler cada linha de um arquivo de texto, a função `LeArquivo()` chama **fgets()**, que tem como protótipo:

```
char *fgets(char *ar, int n, FILE *stream)
```

O primeiro parâmetro dessa função é o endereço do array que armazenará o string lido e o segundo parâmetro é o tamanho desse array. O terceiro parâmetro de **fgets()** especifica o stream associado ao arquivo no qual a leitura será efetuada. A função **fgets()** é capaz de ler `n - 1` caracteres, mas a leitura pode encerrar prematuramente se uma quebra de linha (`'\n'`) for encontrada ou o final do arquivo for atingido. Essa função armazena automaticamente um caractere nulo após o último caractere armazenado no array `ar[]`. Essa função retorna o endereço do array recebido como parâmetro quando consegue cumprir sua missão ou **NULL**, quando ocorre erro ou tentativa de leitura além do final do arquivo sem que ela tenha conseguido ler nenhum caractere.

A função `LeArquivo()` apresentada acima efetua uma tradução de cada linha lida no arquivo de texto antes de armazená-la como um registro na lista. Essa tradução se faz necessária porque o referido arquivo é um arquivo de texto e usa chamadas de **strtok()**, que divide um string em tokens e tem como protótipo:

```
char *strtok(char *str, const char *separadores)
```

Nesse protótipo, `str` é o string a ser dividido em tokens e `separadores` é um string contendo os caracteres que separam os tokens. A primeira chamada de **strtok()** retorna o endereço do primeiro token encontrado no string `str` e um caractere terminal de string é colocado nesse parâmetro ao final do referido token. Chamadas subsequentes dessa função usando **NULL** como primeiro parâmetro retornarão os tokens seguintes até que nenhuma deles seja remanescente no string original. Quando nenhum token é encontrado, a função **strtok()** retorna **NULL**.

Solução de (c):

A função `AtualizaArquivoBin()` a seguir, realiza uma tarefa *aparentemente* (v. adiante) oposta àquela realizada pela função `LeArquivo()`. Quer dizer, a função `AtualizaArquivoBin()` armazena os elementos da lista gerenciada pelo programa em arquivo binário.

```
void AtualizaArquivoBin( const char *nomeArquivo, const tListaAlunos *lista )
{
   FILE   *stream; /* Stream associado ao arquivo no qual será feita a escrita */
   int     nElementosEscritos; /* Número de elementos do array */
                               /* que foram realmente escritos */

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

      /* Se o arquivo não foi aberto, nada mais pode ser feito */
   ASSEGURA(stream, "Arquivo nao pode ser aberto");

      /* Escreve todos os elementos da lista no arquivo */
   nElementosEscritos = fwrite( lista->alunos, sizeof(lista->alunos[0]),
                                lista->nAlunos, stream );

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

      /* Se o número de elementos escritos for diferente do   */
      /* número de elementos do array, deve ter ocorrido erro */
   ASSEGURA( nElementosEscritos == lista->nAlunos,
             "Erro de escrita no arquivo binario" );
}
```

A função `AtualizaArquivoBin()` chama **fwrite()** que lê bytes armazenados em memória e escreve-os num stream. Essa última função tem como protótipo:

```
size_t fwrite(const void *ar, size_t tamanho, size_t n, FILE *stream)
```

Os parâmetros dessa função são interpretados como:

- `ar` é o endereço do array que armazena os bytes que serão escritos no stream. O tipo **void** * utilizado na declaração desse parâmetro permite que ele seja compatível com ponteiros e endereços de variáveis de quaisquer tipos (v. **Seção 9.3**).
- `tamanho` é o tamanho de cada elemento do array.
- `n` é o número de elementos do array que serão escritos no stream.
- `stream` representa o stream no qual será feita a escrita.

A função **fwrite()** retorna o número de itens que foram realmente escritos no stream especificado.

Observe que a função `AtualizaArquivoBin()` não efetua nenhuma tradução dos dados armazenados no arquivo binário e isso ocorre exatamente porque os dados são armazenados em formato binário. Note ainda que uma única chamada de **fwrite()** é suficiente para armazenar toda a lista no arquivo. Mas, isso não significa que essa operação é $\theta(1)$, pois cada chamada de **fwrite()** tem custo temporal $\theta(n)$, em que n é o número de bytes escritos por essa função.

Solução de (d): A função **main()** apresentada a seguir implementa o que foi solicitado.

```
int main (void)
{
   tListaAlunos lista;
   tAluno       aluno;
   const char  *opcoes[] = { "Acrescenta um aluno",
                             "Remove um aluno",
                             "Consulta dados de aluno",
                             "Altera dados de aluno",
                             "Exibe a turma na tela",
                             "Encerra o programa"
                           };
   int          indice,
                op,
                nOpcoes = sizeof(opcoes)/sizeof(opcoes[0]);
   char         umNome[MAX_NOME + 1],
                umaMatr[TAM_MATR + 1 ];

   IniciaListaIdx(&lista);

      /* Armazena conteúdo do arquivo de dados e o armazena na lista */
   LeArquivo(NOME_ARQUIVO, &lista);

   while (1) {
      ApresentaMenu(opcoes, nOpcoes);

      op = LeOpcao("123456");

      if (op == '6') { /* Encerra o programa */
         AtualizaArquivoBin(NOME_ARQ_BIN, &lista); /* Atualiza arquivo binário */

         break; /* Saída do laço */
      }

      switch (op) {
         case '1': /* Acrescenta um aluno */
                   /* Lê dados do novo aluno, incluindo matrícula */
            LeDadosAluno(&aluno, 1);

            if (InsereEmOrdemIdx(&lista, &aluno))
               printf("\n>>> Impossivel acrescentar o aluno\n");
            else
               printf("\n>>> Acrescimo bem sucedido\n");

            break;

         case '2': /* Remove um aluno */
                   /* Lê a matrícula introduzida pelo usuário */
            LeMatricula(umaMatr, TAM_MATR + 1);

            indice = BuscaSequencialMatr(&lista, umaMatr);
```

```
            if (indice >= 0) {
               (void) RemoveListaIdx(&lista, indice);
               printf("\n>>> Remocao bem sucedida\n");
            else
               printf("\n>>> Matricula nao encontrada\n");

            break;

         case '3': /* Consulta dados de um aluno */
            printf("\n>>> Deseja consultar por nome (N) ou por matricula (M)? ");
            op = LeOpcao("NnMm");

            if (op == 'N' || op == 'n') {
               LeNome(umNome, MAX_NOME + 1);
               indice = BuscaBinariaNome(&lista, umNome);
            } else {
               LeMatricula(umaMatr, TAM_MATR + 1);
               indice = BuscaSequencialMatr(&lista, umaMatr);
            }

            if (indice < 0) {
               printf("\n>>> Aluno nao foi encontrado\n");
            } else {
               aluno = ObtemElementoListaIdx(&lista, indice);
               ExibeElemento(&aluno);
            }

            break;

         case '4': /* Altera dados de um aluno */
            printf( "\n>>> A implementacao dessa opcao fica "
                    "\n>>> como exercicio para o leitor\n" );
            break;

         case '5': /* Apresenta a turma na tela */
            ExibeLista(&lista);
            break;

         default: /* O programa não deve chegar até aqui */
            printf("\nEste programa contem um erro\n");
            return 1;
      }
   }

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

   return 0;
}
```

A função **main()** definida acima chama a função `ExibeLista()` para apresentar o conteúdo da lista que ela gerencia. Essa última função é definida como:

```
void ExibeLista(const tListaAlunos *lista)
{
   tAluno aluno;
   int    tamanho = ComprimentoListaIdx(lista);

      /* Se a lista estiver vazia, não há o que exibir */
   if (!tamanho) {
      printf("\n*** Lista Vazia *** \n");
      return;
   }
```

```
      /* Cabeçalho de apresentação na tela */
   printf( "\n   Nome   \tMatr\tN1\tN2"
           "\n   ====   \t====\t==\t==\n");

      /* Apresenta cada estrutura numa linha separada */
   for (int i = 0; i < tamanho; ++i) {
      aluno = ObtemElementoListaIdx(lista, i);
      ExibeElemento(&aluno);
   }

   putchar('\n'); /* Embelezamento apenas */
}
```

A função `ExibeLista()` chama a função `ExibeElemento()` para exibir o conteúdo de cada elemento da lista na tela. Essa última função bem como outras funções utilizadas neste programa que não foram apresentadas são relativamente triviais e podem ser encontradas no site dedicado ao livro na internet.

7.6.3 Removendo Duplicatas de uma Lista

Problema: Considere a implementação de lista indexada apresentada na **Seção 7.1.2**. (a) Acrescente à essa implementação uma função que remove elementos duplicados de uma lista de elementos do tipo **int**. (b) Determine o custo temporal dessa função. (c) Determine o custo espacial dessa função.

Solução de (a): A função `RemoveDuplicatas()` remove elementos duplicados de uma lista do tipo definido na **Seção 7.1.2**.

```
void RemoveDuplicatas(tListaIdx *lista)
{
   int i;

      /* Remove cada elemento encontrado pela segunda vez na lista. Note que */
      /* o acesso inicia no segundo elemento da lista, pois é impossível que */
      /* o primeiro elemento seja encontrado pela segunda vez.               */
   for (i = 1; i < lista->nElementos; ++i) {
         /* Verifica se o elemento atual é igual a algum */
         /* dos elementos que o precedem na lista        */
      if (EmArray(lista->elementos, i, lista->elementos[i]) >= 0){
            /* O elemento corrente é duplicata e deve ser removido. Se */
            /* o valor retornado por RemoveListaIdx() for menor do que */
            /* ou igual a zero, houve erro na remoção.                 */
         (void) RemoveListaIdx(lista, i);

            /* O índice deve ser decrementado porque o  elemento que ocupou */
            /*o lugar do elemento removido também pode ser uma duplicata    */
         --i;
      }
   }
}
```

A função `RemoveDuplicatas()` chama a função `EmArray()`, definida na **Seção 4.6**, para verificar se um elemento da lista é duplicata e, se for o caso, chama a função `RemoveListaIdx()` para removê-la. Essa última função foi definida na **Seção 7.1.2**.

Solução de (b): A função `EmArray()` tem custo espacial $\theta(n)$ (v. **Seção 4.6**). Como essa função é chamada repetidamente no corpo de um laço **for** que teria custo temporal $\theta(n)$ se todas as instruções nesse corpo fossem $\theta(1)$, então o custo temporal da função `RemoveDuplicatas()` é $\theta(n^2)$.

Solução de (c): Nem a função `RemoveDuplicatas()` nem qualquer função chamada por ela usa espaço adicional. Portanto o custo espacial de `RemoveDuplicatas()` é $\theta(1)$.

7.7 Exercícios de Revisão

Listas sem Ordenação (Seção 7.1)

1. O que é um array estático?
2. (a) Quais são as operações permitidas sobre arrays? (b) Qual é o custo temporal de cada uma dessas operações?
3. Quais são as diferenças entre listas indexadas e arrays?
4. Qual é a diferença entre índice de elemento de array e índice de elemento de lista?
5. O que é acesso direto a um elemento de uma lista?
6. O que é uma lista indexada?
7. Como um índice de elemento de uma lista indexada é definido?
8. Descreva (a) sucessor e (b) antecessor de um elemento de uma lista indexada.
9. (a) Quais são as operações essenciais que devem ser definidas sobre listas sem ordenação? (b) Quais são as operações complementares definidas sobre listas sem ordenação apresentadas neste capítulo? (c) Cite uma operação complementar sobre listas sem ordenação que não foi agraciada neste capítulo.
10. Por que operações sobre listas denominadas *complementares* fazem jus a essa denominação?
11. Como cada uma das seguintes operações complementares sobre listas pode ser efetuada se utilizando apenas as operações essenciais sobre listas?
    (a) Busca de um elemento
    (b) Acréscimo de um elemento
    (c) Checagem de lista vazia
    (d) Alteração de valor de um elemento
12. Um algoritmo deve remover todos os valores negativos de uma lista de elementos do tipo **int**. (a) Qual é o pior caso desse algoritmo e qual é o custo temporal desse caso em termos de notação teta? (b) Qual é o melhor caso desse algoritmo e qual é o custo temporal desse caso em termos de notação teta?
13. Por que a inserção de um elemento numa lista indexada requer movimentação de elementos da lista?
14. (a) Por que a remoção de um elemento de uma lista indexada pode requerer a movimentação de posição de outros elementos da lista? (b) Em que situação não ocorre tal movimentação?
15. Por que a operação de acréscimo de elemento não requer movimentação de qualquer elemento?
16. Por que acrescentar um elemento ao final de uma lista sem ordenação é uma operação válida, mas o mesmo não ocorre com uma lista ordenada?
17. Por que a função `InsereListaIdx()` não causa aborto de programa quando não é bem-sucedida?
18. Por que a função `ObtemElementoListaIdx()` pode abortar o programa no qual ela é chamada?
19. Como a função `ObtemElementoListaIdx()` pode ser implementada de modo que ela indique quando não é bem-sucedida sem causar aborto de programa?
20. Qual é a polêmica que envolve a função `EstaCheiaListaIdx()`?
21. O que é busca?
22. No contexto de busca, o que é (a) chave de busca, (b) chave primária e (c) chave secundária?
23. Qual é o problema do uso de chave secundária em busca?
24. Uma operação de busca numa lista indexada depende do fato de a lista ser ordenada ou não?
25. Em que posição de uma lista indexada sem ordenação é mais fácil acrescentar um elemento? Explique.

26. Por que inserir um elemento numa lista indexada ordenada é mais complicado do que inseri-lo numa lista indexada não ordenada?

27. (a) Como uma operação de busca sequencial numa lista indexada pode ser abreviada quando a lista é ordenada? (b) O uso dessa otimização melhora o custo temporal de busca?

28. Por que a remoção de um elemento de uma lista independe do fato de ela ser ordenada ou não?

Listas Ordenadas (Seção 7.2)

29. (a) O que é uma lista ordenada? (b) Cite cinco operações possíveis sobre listas ordenadas.

30. Que operações definidas para listas ordenadas não fazem sentido para listas sem ordenação?

31. Em que circunstância uma busca sequencial sobre uma lista ordenada é mais eficiente do que uma busca binária sobre a mesma lista?

32. Explique o funcionamento do algoritmo de inserção de elementos em ordem seguido pela função `InsereEmOrdemIdx()`.

33. Como uma função que altera o conteúdo de um elemento de uma lista ordenada pode ser implementada?

34. (a) O que é busca sequencial? (b) O que é busca binária? (c) Busca sequencial pode ser aplicada com listas ordenadas? (d) Busca binária pode ser aplicada com listas sem ordenação?

35. Descreva o algoritmo de busca binária.

36. Se dois elementos de uma lista indexada possuem a mesma chave, qual deles será encontrado primeiro se for efetuada (a) uma busca sequencial ou (b) uma busca binária?

37. Suponha que se tenha um array de elementos do tipo **int** iniciado como:

```
int ar[] = {1, 1, 1, 2, 5, 9, 12};
```

(a) Apresente diagramas semelhantes àqueles da **Figura 7–7** que mostrem como uma busca sequencial encontra a posição de um elemento com valor igual a `1`. (b) Qual dos elementos com valor igual a `1` uma busca binária encontra? (c) Qual desses elementos seria encontrado primeiro numa busca sequencial? (d) Apresente diagramas semelhantes àqueles ilustrados na **Figura 7–8** que mostrem como uma busca sequencial *não* encontra a posição de um elemento com valor igual a `7`.

38. Suponha que se sabe que um elemento está entre os primeiros três elementos de uma lista indexada ordenada contendo 50 elementos inteiros. Que tipo de busca encontra mais rapidamente esse elemento: busca sequencial ou busca binária?

39. A busca binária tem custo temporal $\theta(\log n)$ (v. **Seção 6.11.4**). Sugira um algoritmo que utilize busca binária e que tenha custo temporal $\theta(n \log n)$.

Análise de Operações sobre Listas Indexadas (Seção 7.3)

40. Por que todas as operações sobre listas indexadas apresentam custo espacial $\theta(1)$?

41. (a) Qual é o melhor caso de busca sequencial? (b) Qual é o pior caso de busca sequencial?

42. (a) Qual é o melhor caso de busca binária? (b) Qual é o pior caso de busca binária?

Tratamento de Exceções (Seção 7.4)

43. (a) O que é uma exceção em programação? (b) O que é tratamento de exceções?

44. Descreva o funcionamento da macro `ASSEGURA`.

45. O que significa precondição de uma função?

46. Apresente dois exemplos de condições de exceções em processamento de lista.

Aplicações de Listas (Seção 7.5)

47. Quando todos os elementos acima ou abaixo da diagonal principal de uma matriz quadrada são nulos, tem-se uma **matriz triangular** (inferior ou superior, respectivamente). Numa matriz triangular inferior M

com n linhas, o número máximo de termos não nulos na linha i é i. Quando n é muito grande é um desperdício reservar espaço para os zeros do triângulo superior da matriz. Mostre que o número de elementos não nulos de uma matriz triangular é $n \cdot (n+1)/2$.

48. Em que situações o primeiro esquema de representação de polinômios apresentado é conveniente? (b) Em que situações o segundo esquema de representação de polinômios apresentado é conveniente? (c) Por que, em geral, o segundo esquema de representação de polinômios é mais conveniente?

49. (a) O que são matrizes esparsas? (b) Apresente um exemplo de matriz esparsa.

50. Explique como uma matriz esparsa pode ser armazenada utilizando uma lista indexada ordenada.

51. A forma mais natural para representação de uma matriz $N \times M$ é obviamente por meio de um array bidimensional com $N \times M$ elementos. Então, por que razão foi desenvolvida aqui uma representação alternativa para matrizes esparsas?

Exemplos de Programação (Seção 7.6)

52. O que é um menu sensível ao contexto?

53. (a) No exemplo apresentado na **Seção 7.6.1**, por que a função `ApresentaMenu()` recebe a lista gerenciada pelo programa? (b) Qual é a importância da função `EstaCheiaListaIdx()` nesse programa?

54. Por que o programa apresentado na **Seção 7.6.2** usa tanto busca sequencial quanto busca binária?

7.8 Exercícios de Programação

EP7.1 Implemente o tipo `tListaIdx` definido na **Seção 7.1.2** como um TAD.

EP7.2 Escreva uma função que remove o primeiro elemento que apresenta um determinado valor de uma lista do tipo `tListaIdx` definido na **Seção 7.1.2**. O protótipo dessa função deve ser:

```
int RemoveValor(tListaIdx lista, tElemento valor)
```

O retorno da função `RemoveValor()` deve ser `0`, se a remoção for bem-sucedida ou `1`, se não for encontrado nenhum elemento com o valor especificado como parâmetro.

EP7.3 Escreva uma função que remove todos os elementos que apresentam um determinado valor de uma lista de elementos do tipo **int**. O protótipo dessa função deve ser:

```
int RemoveTodos(int lista[], int *n, int valor)
```

O retorno da função `RemoveTodos()` deve ser `0`, se pelo menos um elemento for removido ou `1`, se não for encontrado nenhum elemento com o valor especificado como parâmetro.

EP7.4 Escreva um programa para gerenciamento de lista de afazeres semelhante ao programa apresentado como exemplo na **Seção 7.6.1**.

EP7.5 Escreva uma função para exibição de todos os elementos de uma lista de elementos do tipo `tListaIdx` definido na **Seção 7.1.2**. Essa função deve utilizar apenas as funções definidas naquela seção.

EP7.6 (a) Escreva uma versão recursiva da função `RemoveListaIdx()` apresentada na **Seção 7.1.2**. (b) Qual é o custo temporal dessa função recursiva? (c) Qual é o custo espacial dessa função recursiva?

EP7.7 (a) Escreva uma versão recursiva da função `InsereListaIdx()` apresentada na **Seção 7.1.2**. (b) Qual é o custo temporal dessa função recursiva? (c) Qual é a custo espacial dessa função recursiva?

EP7.8 Escreva uma função para remover cada elemento de ordem par de uma lista do tipo `tListaIdx` definido na **Seção 7.1.2**.

EP7.9 Escreva uma função para deslocar um elemento de uma lista implementada num array estático n elementos adiante na lista.

EP7.10 Escreva uma função recursiva para inverter uma lista do tipo `tListaIdx` definido na **Seção 7.1.2**.

EP7.11 Escreva uma função para encontrar a interseção de duas listas do tipo `tListaIdx` definido na **Seção 7.1.2**. O protótipo dessa função deve ser:

```
void Intersecao(tListaIdx *inter, const tListaIdx *l1, const tListaIdx *l2)
```

EP7.12

(a) Escreva uma função em C que aceite um polinômio como entrada via teclado e armazene este polinômio numa lista ordenada conforme o segundo esquema de representação de polinômios discutido na **Seção 7.5.1**. O polinômio não precisa necessariamente ser digitado na forma usual (i.e., como uma cadeia de caracteres).

(b) Escreva uma função em C que exiba na tela do computador, numa forma legível, um polinômio representado por uma lista ordenada (segundo esquema de representação).

(c) Escreva uma função que multiplica dois polinômios representados por listas ordenadas de acordo com o segundo esquema de representação de polinômios discutido na **Seção 7.5.1**. Considere os expoentes inteiros positivos e os coeficientes inteiros quaisquer.

(d) Utilizando os resultados das tarefas anteriores, escreva um programa em C que ofereça as opções de soma ou multiplicação de dois polinômios, aceite dois polinômios como entrada, execute a operação escolhida e apresente o resultado na tela do computador.

EP7.13 (a) Escreva uma função em C que calcule a soma de duas matrizes esparsas representadas conforme o esquema apresentado na **Seção 7.5.2**. (b) Escreva um programa que aceite duas matrizes esparsas como entrada (levando em consideração inclusive os zeros da matriz), calcule sua soma e apresente o resultado numa forma legível (incluindo os zeros).

EP7.14 **Preâmbulo:** Suponha que se deseja implementar uma lista ordenada de inteiros num array. Então, pode-se armazenar o tamanho da lista na primeira posição do array e mantê-lo atualizado a cada operação de inserção ou remoção. As demais posições do array são utilizados para armazenar os elementos da lista. **Problema:** Utilize essa abordagem para implementar listas ordenadas de inteiros de modo semelhante ao que foi apresentado na **Seção 7.2.2**.

EP7.15 Escreva uma função que retorna o sucessor de um dado elemento de uma lista do tipo `tListaIdx` definido na **Seção 7.1.2**. O protótipo da função deve ser:

```
tElemento Sucessor(const tListaIdx lista, tElemento elemento)
```

EP7.16 (a) Escreva uma função que remove o último elemento de uma lista do tipo `tListaIdx` definido na **Seção 7.1.2**. (b) Qual é o custo temporal dessa função?

EP7.17 Escreva uma função com protótipo:

```
tListaIdx Copia(tListaIdx *destino, const tListaIdx *fonte)
```

que copia os elementos da lista `fonte` que não fazem parte da lista `destino` para essa última lista.

EP7.18 Sejam T_1 e T_2 duas matrizes triangulares inferiores, cada uma das quais com n linhas. (a) Crie uma estratégia para representar ambas as matrizes num único array bidimensional `A[N][N + 1]`. (b) Escreva trechos de programa para determinar os valores $T_1[i, j]$ e $T_2[i, j]$, $1 \leq i, j \leq n$ a partir da representação das matrizes T_1 e T_2 no array `A[]`. [**Sugestão:** represente o triângulo inferior de T_1 no triângulo inferior de `A[]` e a transposta de `B[]`, que uma matriz triangular superior, no triângulo superior de `C[]`.]