10

LISTAS ORDENADAS

A estrutura de dados lista ordenada será explicada neste capítulo como um caso particular de lista encadeada. Mostraremos também sua aplicação na criação de programas para manipular polinômios.

10.1 Fundamentos

Lista ordenada é uma lista encadeada em que os itens aparecem em *ordem*. Para manter essa ordem, cada item inserido na lista encadeada deve ser corretamente posicionado entre aqueles já existentes na lista.

Por exemplo, a Figura 10.1 mostra como inserir o item `3` na lista ordenada `[1,2,4]`, de modo que a ordem crescente seja mantida, sem que nenhum item já existente na lista precise ser movido na memória.

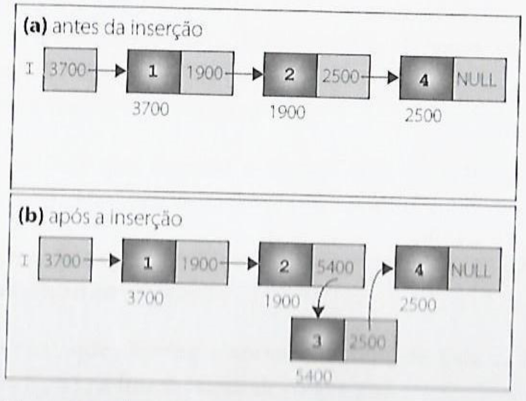

Figura 10.1 | Inserção de um item numa lista ordenada.

10.2 Operações em listas ordenadas

Para criar uma lista ordenada, vamos usar o tipo `Lista`, definido no Capítulo 9. Na verdade, como listas ordenadas são listas encadeadas, toda operação definida para lista encadeada também pode ser usada com lista ordenada.

Por exemplo, usando as funções `no()` e `exibe()`, definidas para lista encadeada, podemos criar e exibir a lista ordenada `I`, da Figura 10.1a, como segue:

```
Lista I = no(1,no(2,no(4,NULL)));
exibe(I);
```

O problema é que, nesse caso, a ordem dos itens na lista encadeada `I` não é definida pela própria função `no()`, mas sim pela ordem em que suas chamadas são feitas. Portanto, não há nenhuma garantia de que uma lista criada com a função `no()` seja uma lista ordenada. Para garantir a criação e manutenção de uma lista ordenada, precisamos ter funções específicas para essa finalidade.

Uma lista ordenada suporta diversas operações, mas as principais são:

- `ins(x,&L)`: insere o item `x` na lista ordenada `L`, em ordem crescente.
- `rem(x,&L)`: remove o item `x` da lista ordenada `L`, se ele estiver em `L`.
- `em(x,L)`: devolve `1` (*verdade*) se o item `x` está em `L`; senão, devolve `0` (*falso*).

Entre essas operações, `ins()` é a mais importante, pois é ela que garante que cada novo item inserido numa lista ordenada seja corretamente posicionado em relação aos demais itens já existentes na lista (independentemente da ordem em que suas chamadas são feitas num programa). A operação `rem()` deve garantir que a lista resultante de uma remoção permaneça ordenada. A operação `em()` deve considerar a ordenação dos itens, a fim de tornar a busca mais eficiente.

10.2.1 Inserção em lista ordenada

A função para inserção em lista ordenada é definida na Figura 10.2. Para inserir um item `x` em uma lista ordenada `I` com essa função, basta chamar `ins(x,&I)`.

```
void ins(Item x, Lista *L) {
   while( *L != NULL && (*L)->item < x ) L = &(*L)->prox;
   *L = no(x,*L);
}
```

Figura 10.2 | Função para inserção em lista ordenada.

A lógica dessa função consiste de duas etapas:

- Primeiro, uma repetição é executada para mover o ponteiro indireto `L` até que ele aponte um ponteiro `NULL` (se `x` for maior que todo item na lista) ou um ponteiro que aponta um item maior ou igual a `x`. Quando essa repetição termina, o ponteiro `L` está apontando justamente o ponteiro que irá apontar o novo nó com o item inserido. Além disso, o ponteiro apontado por `L` indica o endereço do nó que será o sucessor desse novo nó na lista.

- Depois, um novo nó é criado para guardar o item `x` e o valor corrente do ponteiro apontado por `L` e, em seguida, o endereço desse novo nó é atribuído ao ponteiro apontado por `L`.

Note que, se o primeiro item da lista for maior ou igual a `x`, o novo item é inserido no início da lista (causando a alteração do ponteiro inicial dessa lista).

Os passos para inserir o item `3` na lista ordenada `[1,2,4]`, apontada pelo ponteiro `I`, são ilustrados na Figura 10.3.

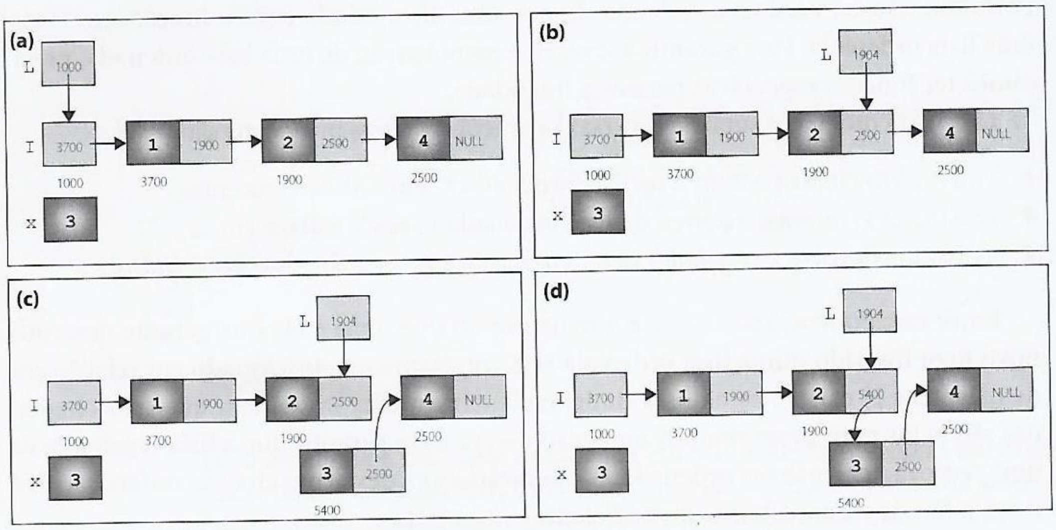

Figura 10.3 | Passos para inserir um item em uma lista ordenada.

Quando a chamada `ins(3,&I)` é feita, o valor `3` é atribuído a `x` e o endereço de `I` é atribuído a `L` (Figura 10.3a). Então, a repetição `while` move `L` até que ele fique apontando o ponteiro que irá apontar o novo nó a ser inserido (Figura 10.3b). Nesse ponto, a chamada `no(x,*L)` é feita, com `x` valendo `3` e `*L` valendo `2500`. Como resultado, um novo nó é criado e preenchido com os valores `3` e `2500` (Figura 10.3c). Finalmente, o endereço do novo nó, devolvido pela função `no()`, é atribuído ao ponteiro apontado por `L` (Figura 10.3d).

10.2.2 Remoção em lista ordenada

A função para remoção em lista ordenada é dada na Figura 10.4. Para remover um item `x` de uma lista ordenada `I` com essa função, basta chamar `rem(x,&I)`.

```
void rem(Item x, Lista *L) {
   while( *L != NULL && (*L)->item < x ) L = &(*L)->prox;
   if( *L == NULL || (*L)->item > x ) return;
   Lista n = *L;
   *L = n->prox;
   free(n);
}
```

Figura 10.4 | Função para remoção em lista ordenada.

A primeira etapa dessa função, feita com o comando `while`, é a mesma usada pela função de inserção: ela move `L` até que ele aponte um ponteiro `NULL` (caso `x` seja maior que todo item da lista) ou um ponteiro que aponta um item maior ou igual a `x`. Quando essa repetição termina, há três possibilidades:

- **O ponteiro `L` aponta *um ponteiro nulo*:** nesse caso, o item `x` é maior que todo item da lista e, portanto, não pertence a ela.
- **O ponteiro `L` aponta *um ponteiro que aponta um item maior que* `x`:** nesse caso, como foi encontrado um item maior que `x`, e a lista está ordenada, concluímos que `x` não pertence à lista (pois ele não pode estar mais à frente).
- **O ponteiro `L` aponta *um ponteiro que aponta um item igual a* `x`:** nesse caso, o nó apontado indiretamente por `L` precisa ser eliminado. Para isso, guardamos o valor apontado por `L` num ponteiro `n`, atribuímos ao ponteiro apontado por `L` o valor de `n->prox` e, depois, desalocamos o nó apontado por `n`.

Os passos para remover o item `2` da lista ordenada `[1,2,4]`, apontada por `I`, são ilustrados na Figura 10.5.

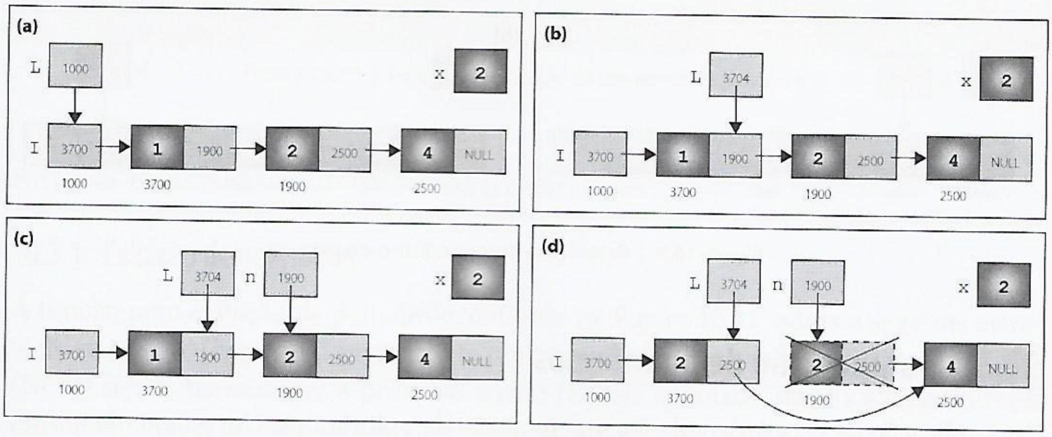

Figura 10.5 | Passos para remover um item de uma lista ordenada.

Quando a chamada `rem(2,&I)` é feita, o valor `2` é atribuído a `x` e o endereço de `I` é atribuído a `L` (Figura 10.5a). Então, a repetição `while` move `L` até que ele fique apontando o ponteiro que aponta o item a ser removido (Figura 10.5b). Nesse ponto, o valor do ponteiro apontado por `L` é copiado para o ponteiro `n` (Figura 10.4c). Finalmente, o valor de `n->prox` é atribuído ao ponteiro apontado por `L` e o nó apontado por `n` é desalocado (Figura 10.5d).

10.2.3 Busca em lista ordenada

A função para busca em lista ordenada é definida na Figura 10.6. Para verificar se um item `x` está em uma lista ordenada `I`, basta chamar `em(x,I)`. Como essa função não precisa alterar o ponteiro inicial da lista ordenada, podemos usar um ponteiro direto para o seu primeiro nó.

```
int em(Item x, Lista L) {
   while( L != NULL && L->item < x ) L = L->prox;
   return (L != NULL && L->item == x);
}
```

Figura 10.6 | Função para busca em lista ordenada.

A primeira etapa dessa função, feita com o comando `while`, é similar àquelas usadas pelas funções de inserção e remoção: ela move `L` até que ele tenha valor `NULL` (o que ocorre quando `x` é maior que todo item da lista) ou, então, até que ele aponte um item maior ou igual a `x`. Quando essa repetição termina, a função verifica se `L` está apontando um nó da lista (isto é, se `L != NULL`) e se esse nó guarda o item `x` (isto é, se `L->item == x`). Caso essas condições sejam verdadeiras, a função devolve `1` como resposta; caso contrário, ela devolve `0`.

Os passos para verificar se o item `4` está na lista ordenada `[1,2,4]`, apontada por `I`, são ilustrados na Figura 10.7.

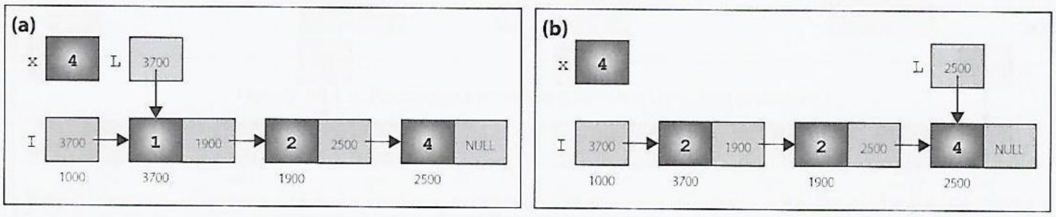

Figura 10.7 | Passos para busca em lista ordenada.

10.3 Manipulação de polinômios

Uma aplicação de lista ordenada é a manipulação de polinômios. Um *polinômio univariado* $P(x)$ é uma expressão da forma $a_n x^n + ... + a_2 x^2 + a_1 x + a_0$, em que:

- x é uma variável real (um polinômio *univariado* tem apenas *uma* variável).
- n, chamado *grau* do polinômio, é uma constante inteira não negativa.
- a_n, ..., a_2, a_1 e a_0, chamados *coeficientes* do polinômio, são constantes reais diferentes de 0.

Um polinômio univariado pode ser representado por uma lista de pares da forma `(c,e)`, cada um deles representando um termo de coeficiente `c` e expoente `e`. Como um polinômio sempre pode ser escrito na *forma reduzida*, na qual não há expoentes repetidos, os pares nessa lista podem ser ordenados em função do expoente, de modo estritamente decrescente. Assim, por exemplo, o polinômio $P(x) = 3x^5 - 6x^2 + 1$ pode ser representado pela lista ordenada na Figura 10.8.

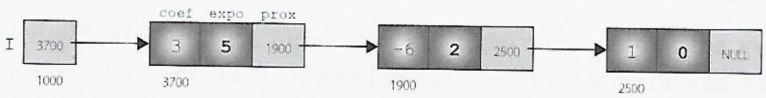

Figura 10.8 | Representação do polinômio $P(x) = 3x^5 - 6x^2 + 1$.

Para criar um polinômio, vamos usar o tipo `Poli`, definido na Figura 10.9.

```
typedef struct poli {
   float coef;
   int expo;
   struct poli *prox;
} *Poli;
```

Figura 10.9 | Definição de tipo para criação de polinômio.

Para criar um termo do polinômio, podemos usar a função na Figura 10.10.

```
Poli termo(float c, int e, Poli p) {
   Poli n = malloc(sizeof(struct poli));
   n->coef = c;
   n->expo = e;
   n->prox = p;
   return n;
}
```

Figura 10.10 | Função para criação de um termo de polinômio.

Usando essa função, podemos criar o polinômio da Figura 10.8 como segue:

```
Poli P = termo(3,5,termo(-6,2,termo(1,0,NULL)));
```

10.3.1 Exibição de polinômio

A função para exibição de polinômio, definida na Figura 10.11, adota a seguinte estratégia recursiva: (*base*) se o ponteiro de polinômio `P` é `NULL`, nada precisa ser exibido; (*passo*) senão, basta exibir o primeiro termo (aquele apontado por `P`) e, depois, fazer uma chamada recursiva para que os demais termos sejam exibidos.

```
void exibep(Poli P) {
   if( P == NULL ) return;
   printf("%+.1f",P->coef);
   if( P->expo!=0 ) printf("*x^%d",P->expo);
   exibep(P->prox);
}
```

Figura 10.11 | Função para exibição de polinômio.

Por exemplo, supondo que `P` aponte o polinômio na Figura 10.8, a chamada `exibep(P)` produz a saída `+3.0*x^5-6.0*x^2+1`. Note que o formato `"%+.1f"`, usado na chamada à função `printf()`, garante que o sinal do número seja exibido, mesmo quando ele for positivo. O teste do valor de `P->expo` evita que a potência de `x` elevado a `0` seja exibida.

10.3.2 Adição em polinômio

A função para adição de um termo a um polinômio é definida na Figura 10.12. Por exemplo, para adicionar o termo $4x^3$ ao polinômio $P(x)$, apontado pelo ponteiro `P`, basta chamar `adt(4,3,&P)`. Note que, como `P` pode ser alterado durante a adição de um termo ao polinômio, ele precisa ser passado por referência.

```
void adt(float c, int e, Poli *P) {
   if( *P == NULL || (*P)->expo<e )
      *P = termo(c,e,*P);
   else if( (*P)->expo==e ) {
      (*P)->coef += c;
      if( (*P)->coef==0 ) {
         Poli n = *P;
         *P = n->prox;
         free(n);
      }
   }
   else adt(c,e,&(*P)->prox);
}
```

Figura 10.12 | Função para adição de um termo a um polinômio.

A função `adt()` adota a seguinte estratégia recursiva:

- (*Base*) Se o ponteiro apontado por `P` é `NULL`, ou aponta um termo com expoente menor que aquele a ser inserido, então basta *inserir* o novo termo no início da lista apontada indiretamente por `P`. Por exemplo, a adição do termo x^6 ao polinômio $3x^5 - 6x^2 + 1$ deve resultar em $x^6 + 3x^5 - 6x^2 + 1$.
- (*Base*) Senão, se o ponteiro apontado por `P` aponta um termo com o mesmo expoente do termo a ser inserido, então basta *alterar* o valor do coeficiente no primeiro nó da lista. Por exemplo, a adição do termo $-2x^5$ ao polinômio $3x^5 - 6x^2 + 1$ deve resultar em $x^5 - 6x^2 + 1$. Porém, se a alteração gerar um termo com coeficiente igual a `0`, então é preciso *remover* o nó que representa esse termo. Por exemplo, a adição do termo $-3x^5$ ao polinômio $3x^5 - 6x^2 + 1$ deve resultar em $-6x^2 + 1$.
- (*Passo*) Senão, basta fazer uma chamada recursiva para adicionar o novo termo ao polinômio apontado por `(*P)->prox`.

Note que a função `adt()` sempre cria polinômios na forma reduzida. Por exemplo, o polinômio na Figura 10.8 pode ser criado do seguinte modo:

```
Poli P = NULL;
adt(+1,0,&P);
adt(-6,2,&P);
adt(+3,5,&P);
```

10.3.3 Avaliação de polinômio

Dada uma constante c, a *avaliação* de um polinômio $P(x)$ consiste em calcular o valor da expressão $P(c)$, obtida pela substituição da variável x pela constante c. Por exemplo, para $c = 2$ e $P(x) = 3x^5 - 6x^2 + 1$, a avaliação de $P(c)$ resulta em $3(2)^5 - 6(2)^2 + 1 = 73$.

A função para avaliação de polinômio, definida na Figura 10.13, usa a função `pow()`, declarada em `math.h`, para calcular uma potência.

```
float valor(Poli P, float x) {
   if( P == NULL ) return 0;
   else return P->coef*pow(x,P->expo) + valor(P->prox,x);
}
```

Figura 10.13 | Função para avaliação de polinômio.

A função `valor()` adota a seguinte estratégia recursiva: (*base*) se o ponteiro `P` é `NULL`, o valor do polinômio é `0`; (*passo*) senão, o valor do polinômio é a soma do valor do primeiro termo, dado pela expressão `P->coef*pow(x,P->expo)`, com o valor do resto do polinômio, que é devolvido pela chamada `valor(P->prox,x)`.

Por exemplo, para `P` apontando a lista que representa o polinômio $P(x) = 3x^5 - 6x^2 + 1$, a chamada `valor(P,2)` deve devolver o valor `73`.

10.3.4 Derivada de polinômio

A *derivada* de um termo da forma $c.x^n$ é $n.c.x^{n-1}$. A derivada de um polinômio é a soma das derivadas de seus termos. Particularmente, se o expoente de um termo é 0, sua derivada também é 0.

A função para derivação de polinômio, na Figura 10.14, adota a seguinte estratégia recursiva: (*base*) se `P` é `NULL`, ou aponta um termo de expoente `0`, a derivada do polinômio é uma lista vazia; (*passo*) senão, a derivada do polinômio é um termo com coeficiente `P->coef*P->expo` e expoente `P->expo-1`, seguido da derivada do restante do polinômio, que é a lista dada por `derivada(P->prox)`.

```
Poli derivada(Poli P) {
   if( P==NULL || P->expo==0 ) return NULL;
   return termo(P->coef*P->expo,P->expo-1,derivada(P->prox));
}
```

Figura 10.14 | Função para derivação de polinômio.

Por exemplo, para `P` apontando a lista que representa o polinômio $P(x) = 3x^5 - 6x^2 + 1$, a execução do comando `exibep(derivada(P))` deve produzir a seguinte saída em vídeo:

```
+15*x^4-12*x^1
```

10.3.5 Destruição de polinômio

A função para destruição de polinômio é definida na Figura 10.15. Para destruir um polinômio apontado por um ponteiro `P`, basta chamar `destroip(&P)`. Como `P` deve ser anulado com a destruição da lista, ele deve ser passado por referência.

```
void destroip(Poli *P) {
   if( *P==NULL) return;
   destroip(&(*P)->prox);
   free(*P);
   *P = NULL;
}
```

Figura 10.15 | Função para destruição de polinômio.

A função `destroip()` implementa a seguinte estratégia recursiva: (*base*) se o ponteiro apontado por `P` (isto é, `*P`) é `NULL`, nada precisa ser feito; (*passo*) senão, para destruir o polinômio apontado por `*P`, basta destruir o resto do polinômio (isto é, a lista apontada por `(*P)->prox`) e, depois, destruir o primeiro termo do polinômio (isto é, o nó apontado por `*P`) e anular o ponteiro que o apontava.

Exercícios

10.1 Crie o arquivo `lstord.h`, que estende o arquivo `lista.h` com as funções para listas ordenadas definidas nesse capítulo, e teste o programa a seguir.

```
#include <stdio.h>
#include "../ed/lstord.h"
int main(void) {
   Lista L = NULL;
   ins(2,&L);
   ins(5,&L);
   ins(1,&L);
   ins(4,&L);
   ins(3,&L);
   printf("Sequencia em ordem crescente: ");
   exibe(L);
   puts("");
   return 0;
}
```

10.2 Usando o tipo `Lista`, faça um programa para ler uma sequência aleatória de *n* inteiros e exibir a sequência correspondente em ordem *decrescente*.

10.3 A função para inserção em lista ordenada, definida na Figura 10.2, permite a criação de listas ordenadas com itens repetidos. Com base nessa função, crie a função `insSR(x,L)`, que insere o item `x` em `L` só se `x` não estiver em `L`.

10.4 Um *conjunto* é uma coleção de elementos distintos. Embora a ordem dos elementos num conjunto seja irrelevante, conjuntos podem ser representados por listas ordenadas (sem itens repetidos). Dadas duas listas ordenadas representando dois conjuntos `A` e `B`, crie a função:

(a) `uniao(A,B)`, que devolve uma lista ordenada representando o conjunto de todos os itens que estão no conjunto `A` ou no conjunto `B`.

(b) `interseccao(A,B)`, que devolve uma lista ordenada representando o conjunto de todos os itens que estão no conjunto `A` e no conjunto `B`.

(c) `diferenca(A,B)`, que devolve uma lista ordenada representando o conjunto de todos os itens que estão no conjunto `A`, mas não no conjunto `B`.

(d) `subconjunto(A,B)`, que informa se todo elemento do conjunto `A` é também um elemento do conjunto `B`.

10.5 Em C, *cadeias* não podem ser atribuídas pelo operador de atribuição (=), nem comparadas pelos operadores relacionais (==, !=, <, <=, > e >=). Então, as funções para listas ordenadas definidas nesse capítulo não podem ser usadas com listas de cadeias. Usando as funções `strcpy()` e `strcmp()`, declaradas em `string.h`, crie novas versões das funções `no()`, `ins()`, `rem()`, `em()` e `exibe()`, que possam ser usadas com listas ordenadas de cadeias.

10.6 A função recursiva `exibep()`, definida na Figura 10.11, exibe o polinômio $P(x) = 4x^3 - 2x$ como `+4.0*x^3-2.0*x^1`. Crie uma versão iterativa dessa função, que não mostra o expoente `1`, isto é, que exibe `+4.0*x^3-2.0*x`.

10.7 Usando a função `adt()`, definida na Figura 10.12, crie a função `soma(P,Q)`, que devolve o polinômio correspondente à soma dos polinômios `Q` e `P`.