AULA 18

LISTAS LINEARES

Listas lineares são as próximas estruturas de dados que aprendemos. Essas estruturas são muito usadas em diversas aplicações importantes para organização de informações na memória tais como representações alternativas para expressões aritméticas, armazenamento de argumentos de funções, compartilhamento de espaço de memória, entre outras. Nesta aula, baseada nas referências [2, 7, 13], aprenderemos a definição dessa estrutura, sua implementação em alocação dinâmica e suas operações básicas.

18.1 Definição

Informalmente, uma lista linear é uma estrutura de dados que armazena um conjunto de informações que são relacionadas entre si. Essa relação se expressa apenas pela ordem relativa entre os elementos. Por exemplo, nomes e telefones de uma agenda telefônica, as informações bancárias dos funcionários de uma empresa, as informações sobre processos em execução pelo sistema operacional, etc, são informações que podem ser armazenadas em uma lista linear. Cada informação contida na lista é na verdade um registro contendo os dados relacionados, que chamaremos daqui por diante de célula. Em geral, usamos um desses dados como uma chave para realizar diversas operações sobre essa lista, tais como busca de um elemento, inserção, remoção, etc. Já que os dados que acompanham a chave são irrelevantes e participam apenas das movimentações das células, podemos imaginar que uma lista linear é composta apenas pelas chaves dessas células e que essas chaves são representadas por números inteiros.

Agora formalmente, uma **lista linear** é um conjunto de $n \geqslant 0$ células $c_1, c_2, \ldots, c_n$ determinada pela ordem relativa desses elementos:

(*i*) se $n > 0$ então c_1 é a primeira célula;

(*ii*) a célula c_i é precedida pela célula c_{i-1}, para todo i, $1 < i \leqslant n$.

As operações básicas sobre uma lista linear são as seguintes:

- busca;
- inclusão; e
- remoção.

Dependendo da aplicação, muitas outras operações também podem ser realizadas sobre essa estrutura como, por exemplo, alteração de um elemento, combinação de duas listas, ordenação da lista de acordo com as chaves, determinação do elemento de menor ou maior chave, determinação do tamanho ou número de elementos da lista, etc.

As listas lineares podem ser armazenadas na memória de duas maneiras distintas:

Alocação estática ou seqüencial: os elementos são armazenados em posições consecutivas de memória, com uso da vetores;

Alocação dinâmica ou encadeada: os elementos podem ser armazenados em posições não consecutivas de memória, com uso de ponteiros.

A aplicação, ou problema que queremos resolver, é que define o tipo de armazenamento a ser usado, dependendo das operações sobre a lista, do número de listas envolvidas e das características particulares das listas. Na aula 4 vimos especialmente as operações básicas sobre uma lista linear em alocação seqüencial, especialmente a operação de busca, sendo que as restantes foram vistas nos exercícios.

As células de uma lista linear em alocação encadeada encontram-se dispostas em posições aleatórias da memória e são ligadas por ponteiros que indicam a posição da próxima célula da lista. Assim, um campo é acrescentado a cada célula da lista indicando o endereço do próximo elemento da lista. Veja a figura 18.1 para um exemplo de uma célula de uma lista.

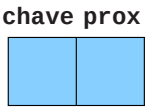

Figura 18.1: Representação de uma célula de uma lista linear em alocação encadeada.

A definição de uma célula de uma lista linear encadeada é então descrita como a seguir:

```
struct cel {
   int chave;
   struct cel *prox;
};
```

É boa prática de programação definir um novo tipo de dados para as células de uma lista linear em alocação encadeada:

```
typedef struct cel celula;
```

Uma célula `c` e um ponteiro `p` para uma célula podem ser declarados da seguinte forma:

```
   celula c;
   celula *p;
```

Se `c` é uma célula então `c.chave` é o conteúdo da célula e `c.prox` é o endereço da célula seguinte. Se `p` é o endereço de uma célula então `p->chave` é o conteúdo da célula apontada por `p` e `p->prox` é o endereço da célula seguinte. Se `p` é o endereço da última célula da lista então `p->prox` vale `NULL`.

Uma ilustração de uma lista linear em alocação encadeada é mostrada na figura 18.2.

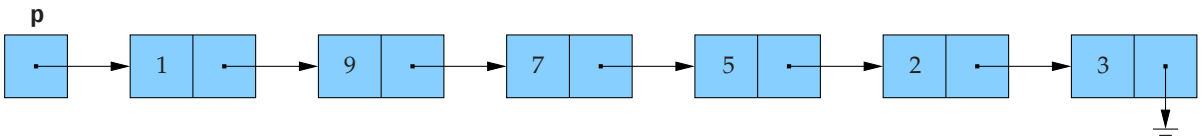

Figura 18.2: Representação de uma lista linear em alocação encadeada na memória.

Dizemos que o endereço de uma lista encadeada é o endereço de sua primeira célula. Se `p` é o endereço de uma lista, podemos dizer que "`p` é uma lista" ou ainda "considere a lista `p`". Por outro lado, quando dizemos "`p` é uma lista", queremos dizer que "`p` é o endereço da primeira célula de uma lista".

Uma lista linear pode ser vista de duas maneiras diferentes, dependendo do papel que sua primeira célula representa. Em uma lista linear **com cabeça**, a primeira célula serve apenas para marcar o início da lista e portanto o seu conteúdo é irrelevante. A primeira célula é a **cabeça** da lista. Em uma lista linear **sem cabeça** o conteúdo da primeira célula é tão relevante quanto o das demais.

Uma lista linear está vazia se não tem célula alguma. Para criar uma lista vazia `lista` com cabeça, basta escrever as seguintes sentenças:

```
celula c, *lista;
c.prox = NULL;
lista = &c;
```

ou ainda

```
celula *lista;
lista = (celula *) malloc(sizeof (celula));
lista->prox = NULL;
```

A figura 18.3 mostra a representação na memória de uma lista linear encadeada com cabeça e vazia.

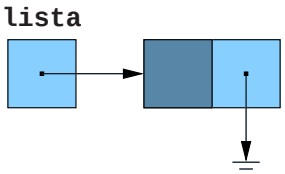

Figura 18.3: Uma lista linear encadeada com cabeça e vazia `lista`.

Para criar uma lista vazia **lista** sem cabeça, basta escrever as seguintes sentenças:

```
celula *lista;
lista = NULL;
```

A figura 18.4 mostra a representação na memória de uma lista linear encadeada sem cabeça vazia.

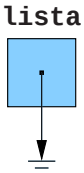

Figura 18.4: Uma lista linear encadeada sem cabeça e vazia **lista**.

Listas lineares com cabeça são mais fáceis de manipular do que aquelas sem cabeça. No entanto, as listas com cabeça têm sempre a desvantagem de manter uma célula a mais na memória.

Para imprimir o conteúdo de todas as células de uma lista linear podemos usar a seguinte função:

```
/* Recebe um ponteiro para uma lista linear encadeada
   e mostra o conteúdo de cada uma de suas células */
void imprime_lista(celula *lst)
{
   celula *p;

   for (p = lst; p != NULL; p = p->prox)
      printf("%d\n", p->chave);
}
```

Se **lista** é uma lista linear com cabeça, a chamada da função deve ser:

```
imprime_lista(lista->prox);
```

Se **lista** é uma lista linear sem cabeça, a chamada da função deve ser:

```
imprime_lista(lista);
```

A seguir vamos discutir e implementar as operações básicas de busca, inserção e remoção sobre listas lineares em alocação encadeada. Para isso, vamos dividir o restante do capítulo em listas sem cabeça e com cabeça.

18.2 Listas lineares com cabeça

Nesta seção tratamos das operações básicas nas listas lineares encadeadas com cabeça. Como mencionamos, as operações básicas são facilmente implementadas neste tipo de lista, apesar de sempre haver necessidade de manter uma célula a mais sem armazenamento de informações na lista.

18.2.1 Busca

O processo de busca de um elemento em uma lista linear em alocação encadeada com cabeça é muito simples. Basta verificar se o elemento é igual ao conteúdo de alguma célula da lista, como pode ser visto na descrição da função abaixo. A função `busca_C` recebe uma lista linear encadeada com cabeça apontada por `lst` e um número inteiro `x` e devolve o endereço de uma célula contendo `x`, caso `x` ocorra em `lst`. Caso contrário, a função devolve o ponteiro nulo `NULL`.

```
/* Recebe um número inteiro x e uma lista encadeada com cabeça lst e devol-
   ve o endereço da célula que contém x ou NULL se tal célula não existe */
celula *busca_C(int x, celula *lst)
{
   celula *p;

   p = lst->prox;

   while (p != NULL && p->chave != x)
      p = p->prox;

   return p;
}
```

É fácil notar que este código é muito simples e sempre devolve o resultado correto, mesmo quando a lista está vazia. Um exemplo de execução da função `busca_C` é mostrado na figura 18.5.

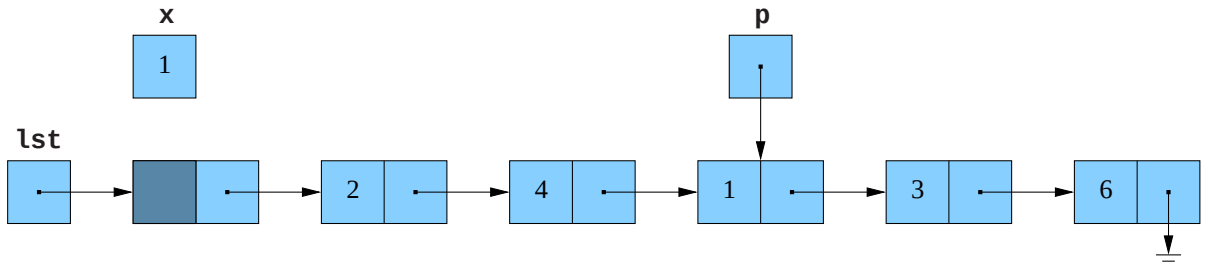

Figura 18.5: Resultado de uma chamada da função `busca_C` para uma dada lista linear e a chave 1.

Uma versão recursiva da função acima é apresentada a seguir.

```
/* Recebe um número inteiro x e uma lista encadeada com cabeça lst e devol-
   ve o endereço da célula que contém x ou NULL se tal célula não existe */
celula *buscaR_C(int x, celula *lst)
{
   if (lst->prox == NULL)
      return NULL;

   if (lst->prox->chave == x)
      return lst->prox;

   return buscaR_C(x, lst->prox);
}
```

18.2.2 Remoção

O problema nesta seção é a remoção de uma célula de uma lista linear encadeada. Parece simples remover tal célula apontada pelo ponteiro, mas esta idéia tem um defeito fundamental: a célula anterior a ela deve apontar para a célula seguinte, mas neste caso não temos o endereço da célula anterior. Dessa forma, no processo de remoção, é necessário apontar para a célula anterior àquela que queremos remover. Como uma lista linear encadeada com nó cabeça sempre tem uma célula anterior, a implementação da remoção é muito simples neste caso. A função `remove_C` recebe um ponteiro `p` para uma célula de uma lista linear encadeada e remove a célula seguinte, supondo que o ponteiro `p` aponta para uma célula e existe uma célula seguinte àquela apontada por `p`.

```
/* Recebe o endereço p de uma célula em uma lista encadeada e remo-
   ve a célula p->prox, supondo que p != NULL e p->prox != NULL */
void remove_C(celula *p)
{
   celula *lixo;

   lixo = p->prox;
   p->prox = lixo->prox;
   free(lixo);
}
```

A figura 18.6 ilustra a execução dessa função.

Observe que não há movimentação de dados na memória, como fizemos na aula 4 ao remover um elemento de um vetor.

18.2.3 Inserção

Nesta seção, queremos inserir uma nova célula em uma lista linear encadaeda com cabeça. Suponha que queremos inserir uma nova célula com chave `y` entre a célula apontada por `p` e a célula seguinte. A função `insere_C` abaixo recebe um número inteiro `y` e um ponteiro `p` para uma célula da lista e realiza a inserção de uma nova célula com chave `y` entre a célula apontada por `p` e a célula seguinte da lista.

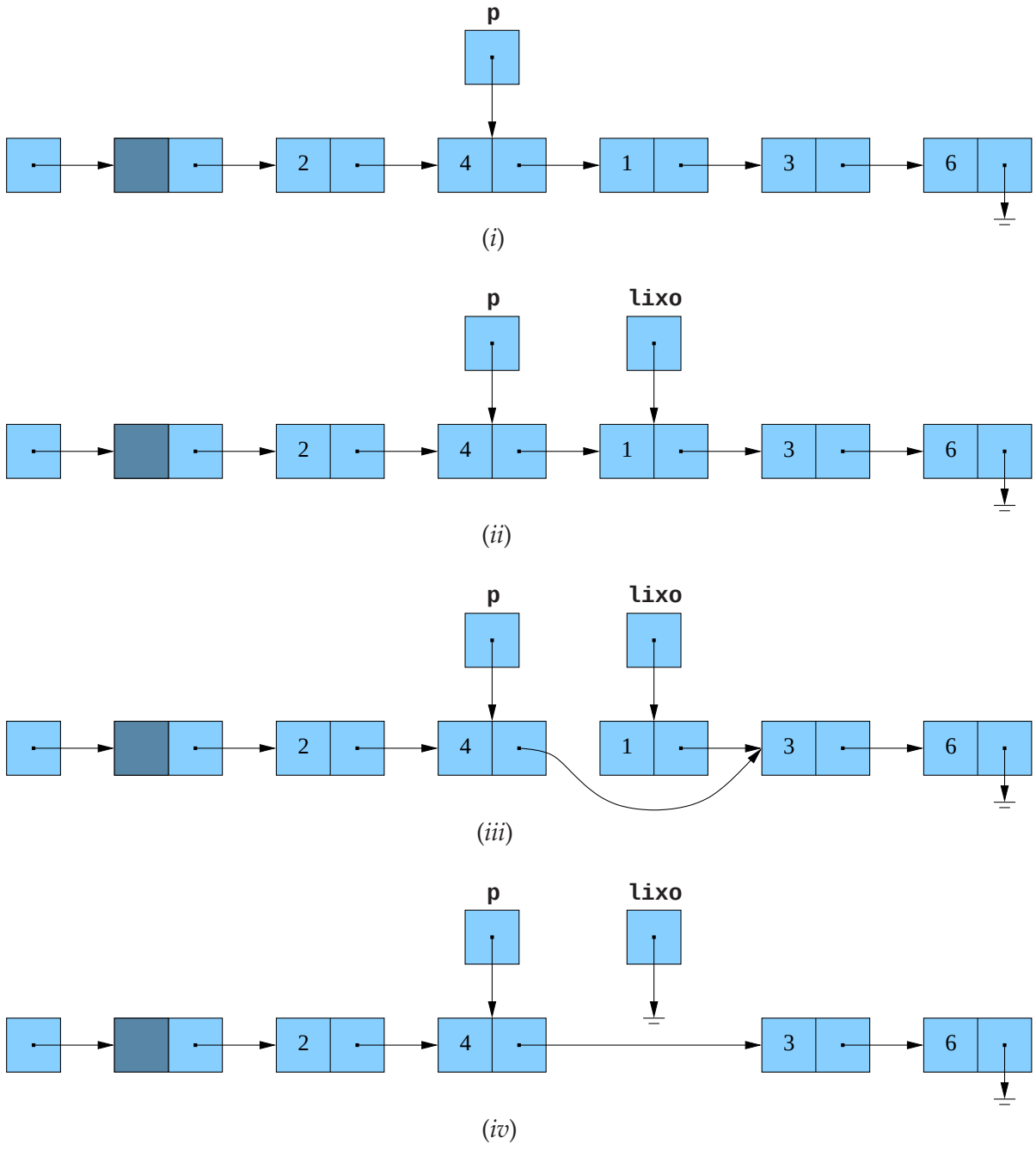

Figura 18.6: Resultado de uma chamada da função `remove_C` para uma célula apontada por `p` de uma dada lista linear. As figuras de (*i*) a (*iv*) representam a execução linha a linha da função.

```
/* Recebe um número inteiro y e o endereço p de uma célula de
   uma lista encadeada e insere uma nova célula com conteúdo
   y entre a célula p e a seguinte; supomos que p != NULL */
void insere_C(int y, celula *p)
{
   celula *nova;

   nova = (celula *) malloc(sizeof (celula));
   nova->chave = y;
   nova->prox = p->prox;
   p->prox = nova;
}
```

A figura 18.7 ilustra a execução dessa função.

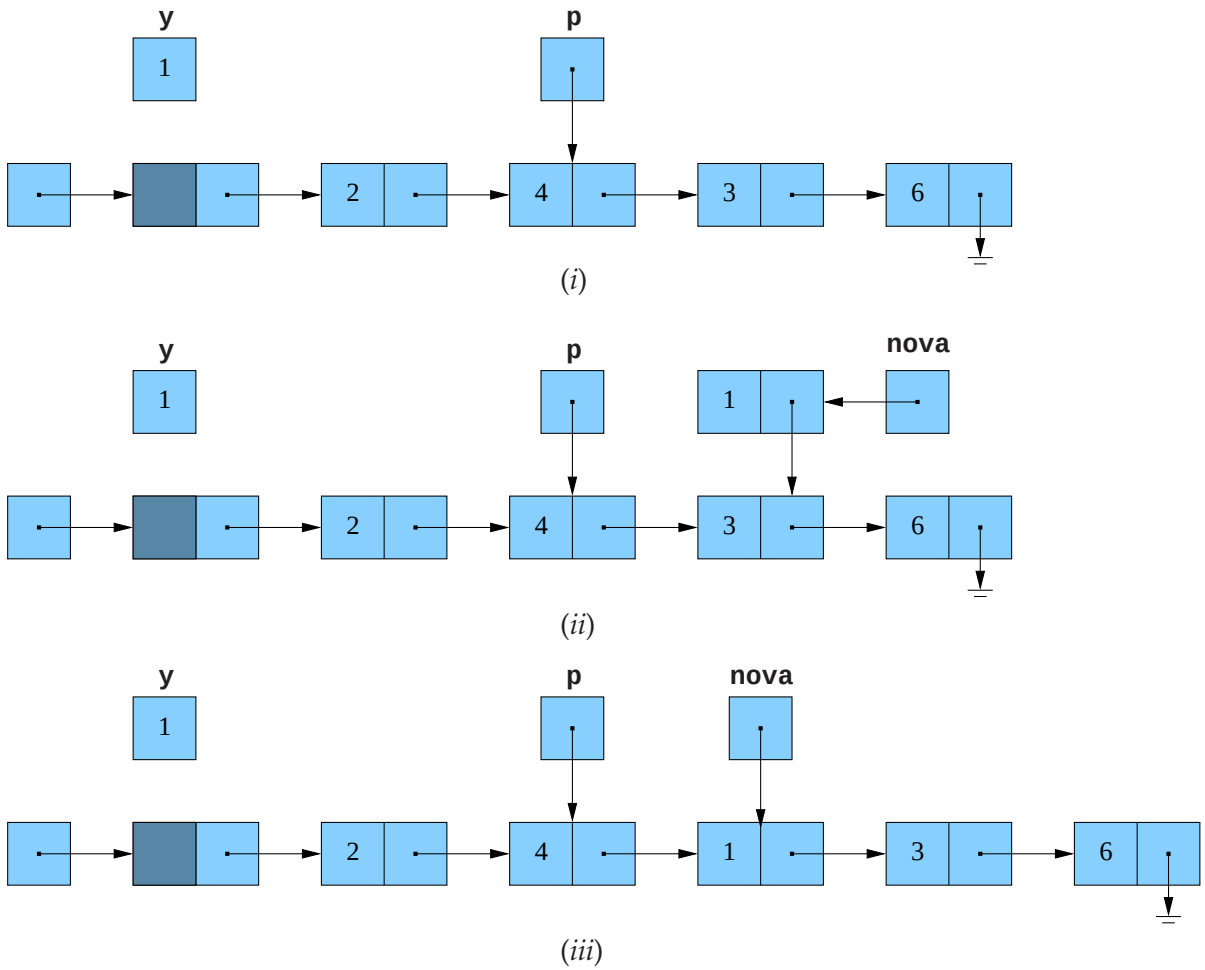

Figura 18.7: Resultado de uma chamada da função `insere_C` para uma nova chave 1 e a célula apontada por `p` de uma dada lista linear.

Observe que também não há movimentação de dados na memória na inserção realizada nessa função.

18.2.4 Busca com remoção ou inserção

Nesta seção, temos dois problemas a serem resolvidos. No primeiro, dado um número inteiro, nosso problema agora é aquele de remover a célula de uma lista linear encadaeda que contenha tal número. Se tal célula não existe, nada fazemos. A função `busca_remove_C` apresentada a seguir implementa esse processo. A figura 18.8 ilustra a execução dessa função.

```
/* Recebe um número inteiro x e uma lista encadeada com cabeça lst e remo-
   ve da lista a primeira célula que contiver x, se tal célula existir */
void busca_remove_C(int x, celula *lst)
{
   celula *p, *q;

   p = lst;
   q = lst->prox;
   while (q != NULL && q->chave != x) {
      p = q;
      q = q->prox;
   }
   if (q != NULL) {
      p->prox = q->prox;
      free(q);
   }
}
```

Agora, nosso problema é o de inserir uma nova célula com uma nova chave em uma lista linear encadaeda com cabeça, antes da primeira célula que contenha como chave um número inteiro fornecido. Se tal célula não existe, inserimos a nova chave no final da lista. A função `busca_remove_C` apresentada a seguir implementa esse processo. Um exemplo de execução dessa função é ilustrado na figura 18.8.

```
/* Recebe dois números inteiros y e x e uma lista encadeada
   com cabeça lst e insere uma nova célula com chave y nessa
   lista antes da primeira que contiver x; se nenhuma célula
   contiver x, a nova célula é inserida no final da lista */
void busca_insere_C(int y, int x, celula *lst)
{
   celula *p, *q, *nova;

   nova = (celula *) malloc(sizeof (celula));
   nova->chave = y;
   p = lst;
   q = lst->prox;
   while (q != NULL && q->chave != x) {
      p = q;
      q = q->prox;
   }
   nova->prox = q;
   p->prox = nova;
}
```

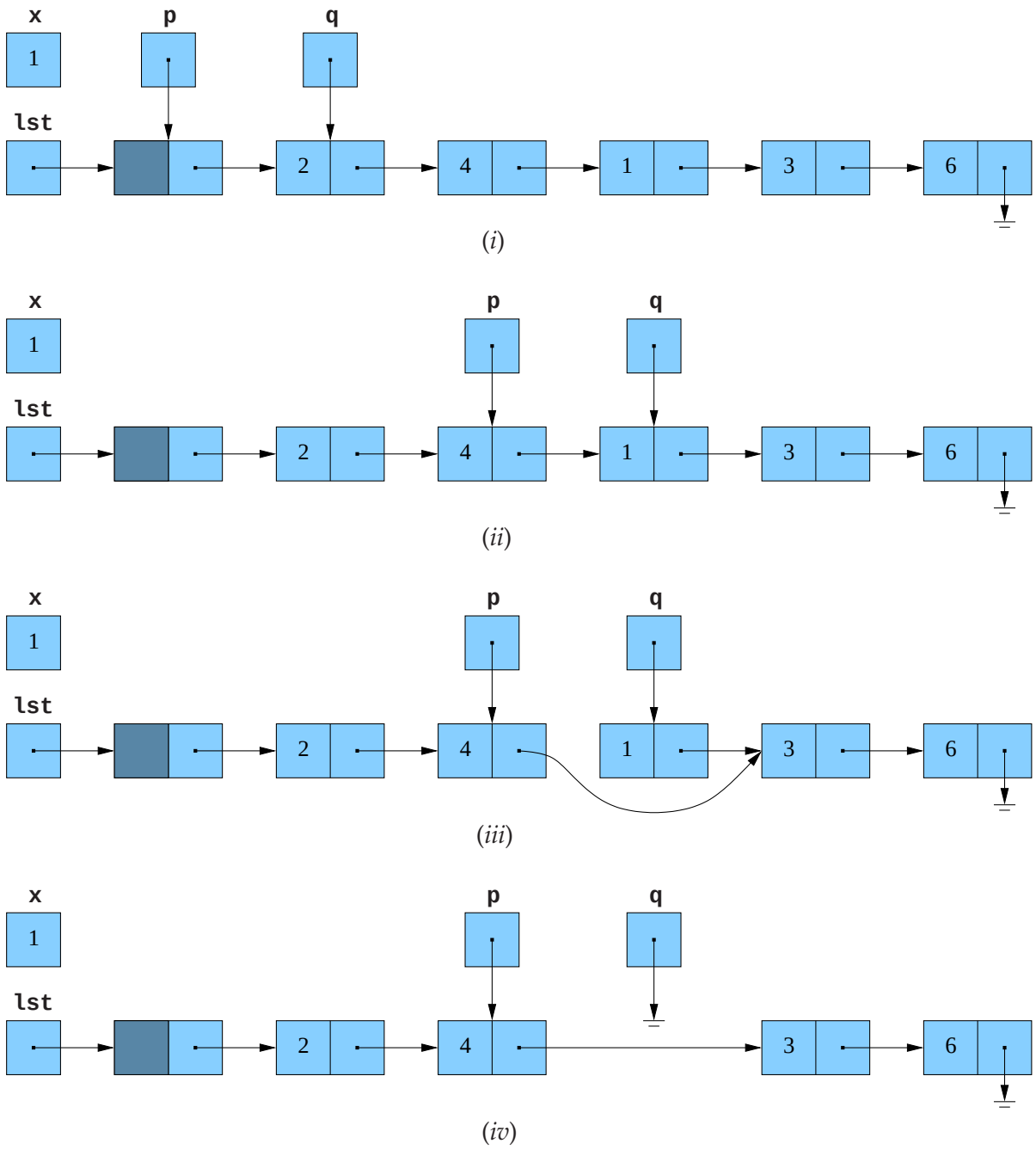

Figura 18.8: Resultado de uma chamada da função `busca_remove_C` para a chave 1 e uma lista `lst`. As figuras de (*i*) a (*iv*) representam a execução linha a linha da função.

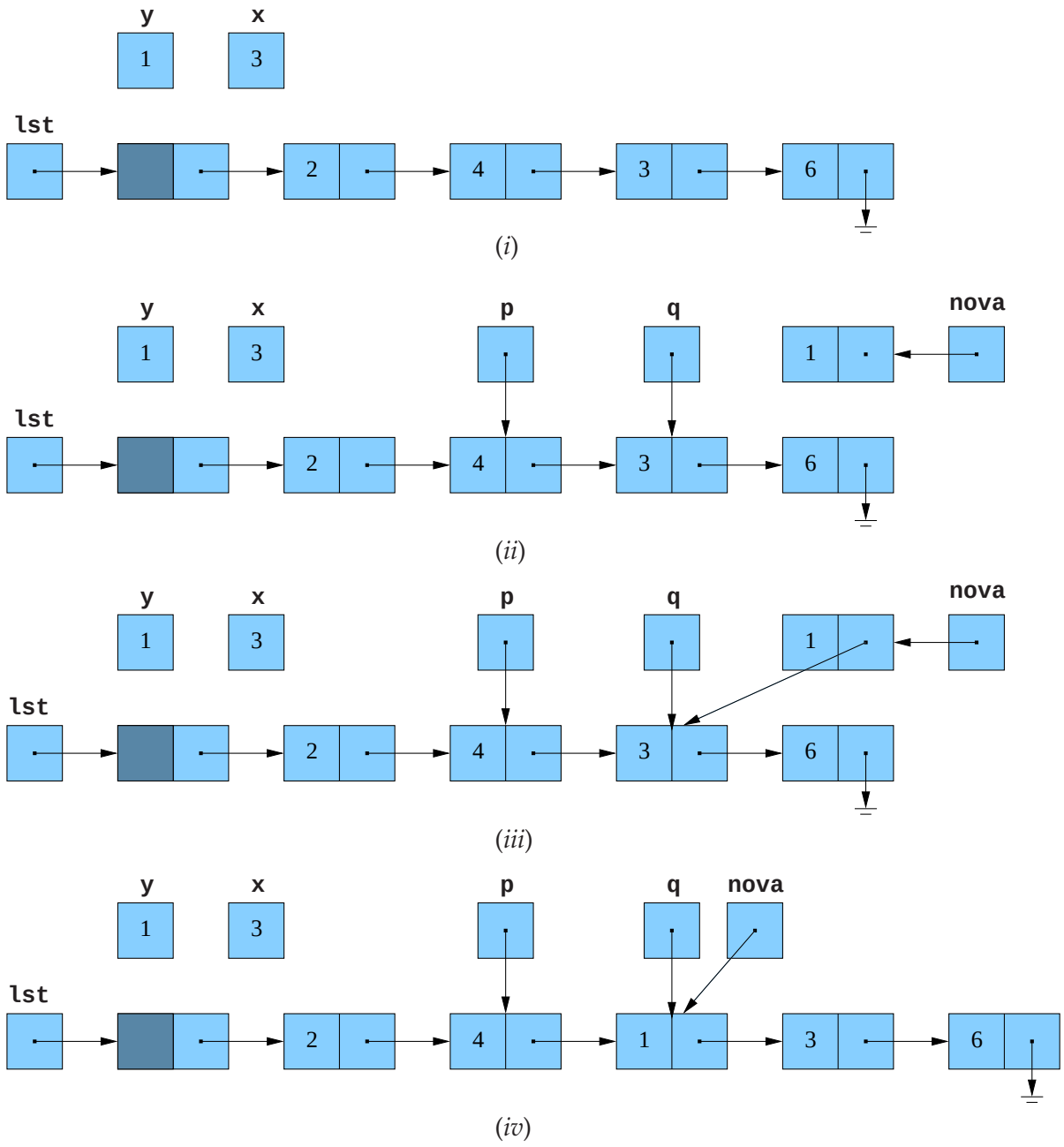

Figura 18.9: Resultado de uma chamada da função `busca_insere_C` para as chaves 1 e 3 e uma lista `lst`. As figuras de (*i*) a (*iv*) representam a execução linha a linha da função.

18.3 Listas lineares sem cabeça

Nesta seção tratamos das operações básicas nas listas lineares encadeadas sem cabeça. Como mencionamos, sempre economizamos uma célula em listas desse tipo, mas a implementação as operações básicas são um pouco mais complicadas.

18.3.1 Busca

A busca de um elemento em uma lista linear em alocação encadeada sem cabeça é muito simples e semelhante ao mesmo processo realizado sobre uma lista linear encadeada com cabeça. Seguem os códigos da função não-recursiva `busca_S` e da função recursiva `buscaR_S`.

```
/* Recebe um número inteiro x e uma lista encadeada sem cabeça lst e devol-
   ve o endereço da célula que contém x ou NULL se tal célula não existe */
celula *busca_S(int x, celula *lst)
{
   celula *p;

   p = lst;
   while (p != NULL && p->chave != x)
      p = p->prox;

   return p;
}
```

```
/* Recebe um número inteiro x e uma lista encadeada sem cabeça lst e devol-
   ve o endereço da célula que contém x ou NULL se tal célula não existe */
celula *buscaR_S(int x, celula *lst)
{
   if (lst == NULL)
      return NULL;

   if (lst->chave == x)
      return lst;

   return buscaR_S(x, lst->prox);
}
```

18.3.2 Busca com remoção ou inserção

Nesta seção apresentamos uma função que realiza uma busca seguida de remoção em uma lista linear encadeada sem cabeça.

Há muitas semelhanças entre a função de busca seguida de remoção em uma lista linear encadeada com cabeça e sem cabeça. No entanto, devemos reparar na diferença fundamental entre as duas implementações: uma remoção em uma lista linear encadeada sem cabeça pode modificar o ponteiro que identifica a lista quando removemos a primeira célula da lista. Dessa forma, a função `busca_remove_S` descrita abaixo tem como parâmetro que identifica a lista linear um ponteiro para um ponteiro.

```
/* Recebe um número inteiro x e uma lista encadeada com cabeça lst e remo-
   ve da lista a primeira célula que contiver x, se tal célula existir */
void busca_remove_S(int x, celula **lst)
{
   celula *p, *q;

   p = NULL;
   q = *lst;
   while (q != NULL && q->chave != x) {
      p = q;
      q = q->prox;
   }
   if (q != NULL)
      if (p != NULL) {
         p->prox = q->prox;
         free(q);
      }
      else {
         *lst = q->prox;
         free(q);
      }
}
```

Veja a figura 18.10 para um exemplo de busca seguida de remoção.

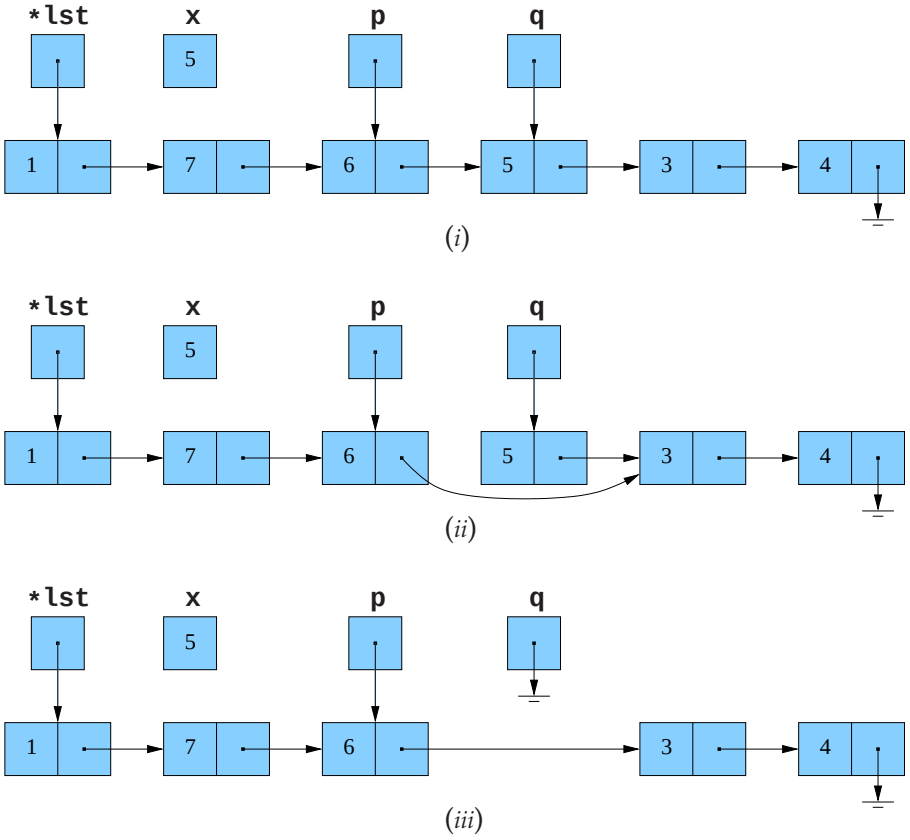

Figura 18.10: Remoção de uma chave em uma lista linear encadeada sem cabeça. (*i*) Após a busca do valor. (*ii*) Modificação do ponteiro `p->prox`. (*iii*) Liberação da memória.

Uma chamada à função `busca_remove_S` é ilustrada abaixo, para um número inteiro `x` e uma `lista`:

```
busca_remove_S(x, &lista);
```

Observe que os ponteiros `p` e `q` são variáveis locais à função `busca_remove_S`, que auxiliam tanto na busca como na remoção propriamente. Além disso, se `p == NULL` depois da execução da estrutura de repetição da função, então a célula que contém `x` que queremos remover, encontra-se na primeira posição da lista. Dessa forma, há necessidade de alterar o conteúdo do ponteiro `*lst` para o registro seguinte desta lista.

Agora, nosso problema é realizar uma bsuca seguida de uma inserção de uma nova célula em uma lista linear ecncadeada sem cabeça. Do mesmo modo, há muitas semelhanças entre a função de busca seguida de inserção em uma lista linear encadeada com cabeça e sem cabeça. De novo, devemos reparar na diferença fundamental entre as duas implementações: uma inserção em uma lista linear encadeada sem cabeça pode modificar o ponteiro que identifica a lista quando inserimos uma nova célula em uma lista vazia. Dessa forma, a função `busca_insere_S` descrita abaixo tem como parâmetro que identifica a lista linear um ponteiro para um ponteiro.

```
/* Recebe dois números inteiros y e x e uma lista encadeada
   com cabeça lst e insere uma nova célula com chave y nessa
   lista antes da primeira que contiver x; se nenhuma célula
   contiver x, a nova célula é inserida no final da lista */
void busca_insere_S(int y, int x, celula **lst)
{
   celula *p, *q, *nova;

   nova = (celula *) malloc(sizeof (celula));
   nova->chave = y;
   p = NULL;
   q = *lst;
   while (q != NULL && q->chave != x) {
      p = q;
      q = q->prox;
   }
   nova->prox = q;
   if (p != NULL)
      p->prox = nova;
   else
      *lst = nova;
}
```

Uma chamada à função `busca_insere_S` pode ser feita como abaixo, para números inteiros `y` e `x` e uma `lista`:

```
busca_insere_S(y, x, &lista);
```

Exercícios

18.1 Se conhecemos apenas o ponteiro `p` para uma célula de uma lista linear em alocação encadeada, como na figura 18.11, e nada mais é conhecido, como podemos modificar a lista linear de modo que passe a conter apenas os valores $20, 4, 19, 47$, isto é, sem o conteúdo da célula apontada por `p`?

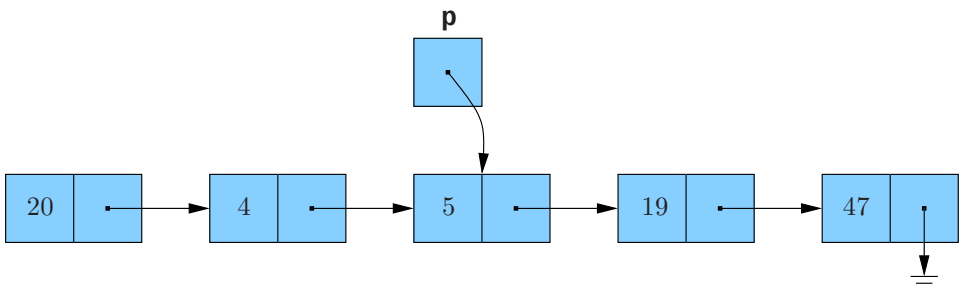

Figura 18.11: Uma lista linear encadeada com um ponteiro `p`.

18.2 O esquema apresentado na figura 18.12 permite percorrer uma lista linear encadeada nos dois sentidos, usando apenas o campo `prox` que contém o endereço do próximo elemento da lista. Usamos dois ponteiros `esq` e `dir`, que apontam para dois elementos vizinhos da lista. A idéia desse esquema é que à medida que os ponteiros `esq` e `dir` caminham na lista, os campos `prox` são invertidos de maneira a permitir o tráfego nos dois sentidos.

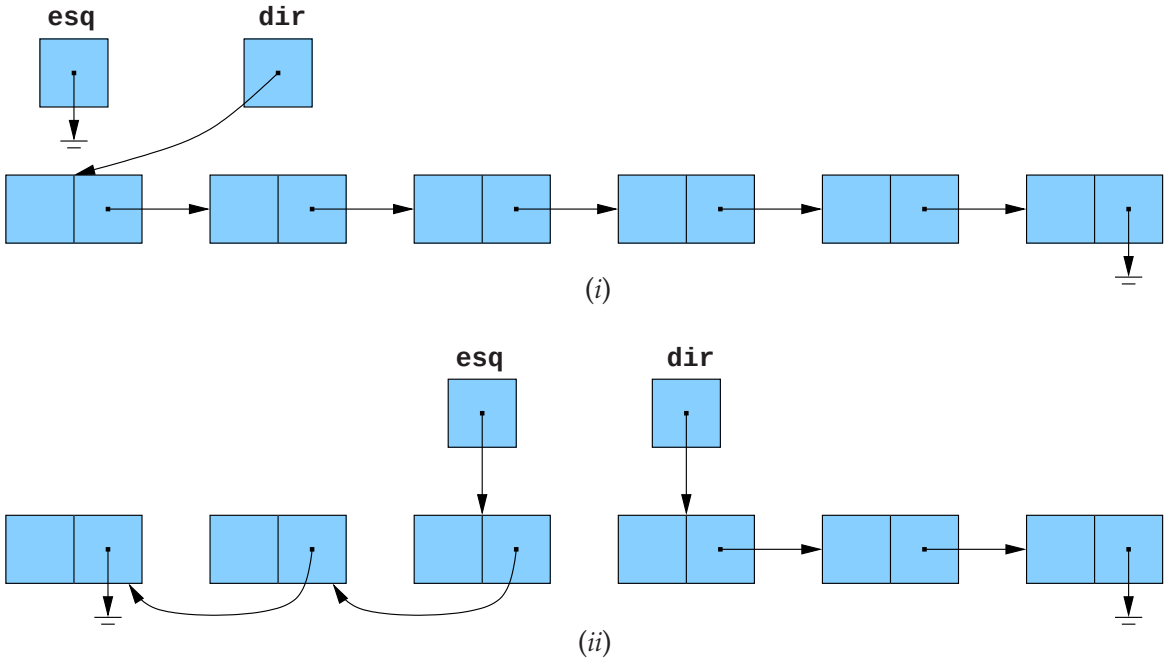

Figura 18.12: A figura (*ii*) foi obtida de (*i*) através da execução da função dada no exercício 18.2(a) por três vezes.

Escreva funções para:

(a) mover `esq` e `dir` para a direita de uma posição.

(b) mover `esq` e `dir` para a esquerda de uma posição.

18.3 Que acontece se trocarmos `while (p != NULL && p->chave != x)` por `while (p->chave != x && p != NULL` na função `busca_C`?

18.4 Escreva uma função que encontre uma célula cuja chave tem valor mínimo em uma lista linear encadeada. Considere listas com e sem cabeça e escreva versões não-recursivas e recursivas para a função.

18.5 Escreva uma função `busca_insere_fim` que receba um número inteiro `y`, uma lista linear encadeada `lst` e um ponteiro `f` para o fim da lista e realize a inserção desse valor no final da lista. Faça uma versões para listas lineares com cabeça e sem cabeça.

18.6 (a) Escreva duas funções: uma que copie um vetor para uma lista linear encadeada com cabeça; outra, que gfaça o mesmo para uma lista linear sem cabeça.

(b) Escreva duas funções: uma que copie uma lista linear encadeada com cabeça em um vetor; outra que copie uma lista linear encadeada sem cabeça em um vetor.

18.7 Escreva uma função que decida se duas listas dadas têm o mesmo conteúdo. Escreva duas versões: uma para listas lineares com cabeça e outra para listas lineares sem cabeça.

18.8 Escreva uma função que conte o número de células de uma lista linear encadeada.

18.9 Seja `lista` uma lista linear com seus conteúdos dispostos em ordem crescente. Escreva funções para realização das operações básicas de busca, inserção e remoção, respectivamente, em uma lista linear com essa característica. Escreva conjuntos de funções distintas para listas lineares com cabeça e sem cabeça. As operações de inserção e remoção devem manter a lista em ordem crescente.

18.10 Sejam duas listas lineares `lst1` e `lst2`, com seus conteúdos dispostos em ordem crescente. Escreva uma função `concatena` que receba `lst1` e `lst2` e construa uma lista `R` resultante da intercalação dessas duas listas, de tal forma que a lista construída também esteja ordenada. A função `concatena` deve destruir as listas `lst1` e `lst2` e deve devolver `R`. Escreva duas funções para os casos em que as listas lineares encadeadas são com cabeça e sem cabeça.

18.11 Seja `lst` uma lista linear encadeada composta por células contendo os valores $c_1, c_2, \ldots, c_n$, nessa ordem. Para cada item abaixo, escreva duas funções considerando que `lst` é com cabeça e sem cabeça.

(a) Escreva uma função `roda1` que receba uma lista e modifique e devolva essa lista de tal forma que a lista resultante contenha as chaves $c_2, c_3, \ldots, c_n, c_1$, nessa ordem.

(b) Escreva uma função `inverte` que receba uma lista e modifique e devolva essa lista de tal forma que a lista resultante contenha as chaves $c_n, c_{n-1}, \ldots, c_2, c_1$, nessa ordem.

(c) Escreva uma função `soma` que receba uma lista e modifique e devolva essa lista de tal forma que a lista resultante contenha as chaves $c_1 + c_n, c_2 + c_{n-1}, \ldots, c_{n/2} + c_{n/2+1}$, nessa ordem. Considere n par.

18.12 Sejam S_1 e S_2 dois conjuntos disjuntos de números inteiros. Suponha que S_1 e S_2 estão implementados em duas listas lineares em alocação encadeada. Escreva uma função `uniao` que receba as listas representando os conjuntos S_1 e S_2 e devolva uma lista resultante que representa a união dos conjuntos, isto é, uma lista linear encadeada que representa o conjunto $S = S_1 \cup S_2$. Considere os casos em que as listas lineares encadeadas são com cabeça e sem cabeça.

18.13 Seja um polinômio $p(x) = a_0x^n + a_1x^{n-1} + \ldots + a_n$, com coeficientes de ponto flutuante. Represente $p(x)$ adequadamente por uma lista linear encadeada e escreva as seguintes funções. Para cada item a seguir, considere o caso em que a lista linear encadeada é com cabeça e sem cabeça.

(a) Escreva uma função `pponto` que receba uma lista `p` e um número de ponto flutuante `x0` e calcule e devolva $p(x_0)$.

(b) Escreva uma função `psoma` que receba as listas lineares `p` e `q`, que representam dois polinômios, e calcule e devolva o polinômio resultante da operação $p(x) + q(x)$.

(c) Escreva uma função `pprod` que receba as listas lineares `p` e `q`, que representam dois polinômios, e calcule e devolva o polinômio resultante da operação $p(x) \cdot q(x)$.

18.14 Escreva uma função que aplique a função `free` a todas as células de uma lista linear encadeada, supondo que todas as suas células foram alocadas com a funçao `malloc`. Faça versões considerando listas lineares encadeadas com cabeça e sem cabeça.