Capítulo 4

Listas encadeadas

Uma lista encadeada é uma representação de uma sequência de objetos na memória do computador. Cada elemento da sequência é armazenado em uma "célula" da lista. As células que armazenam elementos consecutivos da sequência não ficam necessariamente em posições consecutivas da memória.

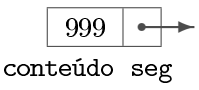
4.1 Definição

Uma **lista encadeada** é uma sequência de registros (veja Apêndice E) que chamaremos **células**. Cada célula contém um objeto de determinado tipo e o endereço (veja Seção D.1) da célula seguinte (no caso da última célula, esse endereço é `NULL`).

Suporemos neste capítulo que os objetos armazenados nas células são do tipo `int`. A estrutura das células pode, então, ser definida assim:

```
struct cel {
   int         conteúdo;
   struct cel *seg;
};
```

[1]



É conveniente tratar as células como um novo tipo de dados (veja Seção J.3), que chamaremos `célula`:

```
typedef struct cel célula;
```

Uma célula `c` e um ponteiro `p` para uma célula podem agora ser declarados

[1] Veja Seção A.4.

assim:

```
célula  c;
célula *p;
```

Se `c` é uma célula então `c.conteúdo` é o conteúdo da célula e `c.seg` é o endereço da célula seguinte. Se `p` é o endereço de uma célula, então `p->conteúdo` é o conteúdo da célula e `p->seg` é o endereço da célula seguinte (veja Seção E.2). Se `p` é o endereço da última célula da lista então `p->seg` vale `NULL`.

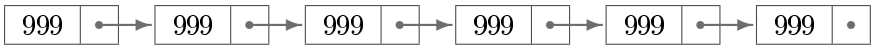


Figura 4.1: Uma lista encadeada (veja Figura D.2 no Apêndice D).

O **endereço** de uma lista encadeada é o endereço de sua primeira célula. Se `p` é o endereço de uma lista, podemos dizer, simplesmente, "`p` é uma lista" e "considere a lista `p`". Reciprocamente, a expressão "`p` é uma lista" deve ser interpretada como "`p` é o endereço da primeira célula de uma lista".

A seguinte observação coloca em evidência a natureza recursiva das listas encadeadas: para toda lista encadeada `p`,

1. `p` é `NULL` ou
2. `p->seg` é uma lista encadeada.

Muitos algoritmos que manipulam listas ficam mais simples quando escritos em estilo recursivo.

Exercício

4.1.1 Dizemos que uma `célula D` é *sucessora* de uma `célula C` se `C.seg` = `&D`. Nas mesmas condições, dizemos que `C` é *antecessora* de `D`. Um *ciclo* é uma sequência $(C_1, C_2, \ldots, C_k)$ de `células` tal que C_{i+1} é sucessora de C_i para $i = 1, 2, \ldots, k-1$ e C_1 é sucessora de C_k. Mostre que uma coleção $\mathcal{L}$ de `células` é uma lista encadeada se e somente se (1) a sucessora de cada elemento de $\mathcal{L}$ está em $\mathcal{L}$, (2) cada elemento de $\mathcal{L}$ tem no máximo uma antecessora, (3) exatamente um elemento de $\mathcal{L}$ não tem antecessora em $\mathcal{L}$ e (4) não há ciclos em $\mathcal{L}$.

4.2 Listas com cabeça e sem cabeça

Uma lista encadeada pode ser vista de duas maneiras diferentes, dependendo do papel que sua primeira célula desempenha. Na lista **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. Se `lst` é o endereço da cabeça então `lst->seg` vale `NULL` se e somente se a lista está vazia. Para criar uma lista vazia deste tipo, basta dizer

```
célula c, *lst;
c.seg = NULL;
lst = &c;
```

ou

```
célula *lst;
lst = malloc (sizeof (célula));
lst->seg = NULL;
```

[2]

Na lista **sem cabeça**, o conteúdo da primeira célula é tão relevante quanto o das demais. A lista está vazia se não tem célula alguma. Para criar uma lista vazia `lst` basta dizer

```
célula *lst;
lst = NULL;
```

Embora as listas sem cabeça sejam mais "puras", trataremos preferencialmente de listas *com* cabeça, pois elas são mais fáceis de manipular. O caso das listas sem cabeça será relegado aos exercícios.

```
void Imprima (célula *lst) {
   célula *p;
   for (p = lst; p != NULL; p = p->seg)
      printf ("%d\n", p->conteúdo);
}
```

Figura 4.2: A função imprime o conteúdo de uma lista encadeada `lst` sem cabeça. Para aplicar a função a uma lista *com* cabeça, diga `Imprima (lst->seg)`. Outra possibilidade é trocar o fragmento "`for (p = lst`" por "`for (p = lst->seg`", obtendo assim uma versão da função que só se aplica a listas com cabeça.

4.3 Busca em lista encadeada

É fácil verificar se um objeto x pertence a uma lista encadeada, ou seja, se x é igual ao conteúdo de alguma célula da lista:

[2] Veja Seção F.2.

```
/* Esta função recebe um inteiro x e uma lista encadeada lst
 * com cabeça. Devolve o endereço de uma célula que contém x
 * ou devolve NULL se tal célula não existe. */
célula *Busca (int x, célula *lst) {
   célula *p;
   p = lst->seg;
   while (p != NULL && p->conteúdo != x)
      p = p->seg;
   return p;
}
```

(Observe como o código é simples. Observe como produz o resultado correto mesmo quando a lista está vazia.) Eis uma versão recursiva da função:

```
célula *BuscaR (int x, célula *lst) {
   if (lst->seg == NULL)
      return NULL;
   if (lst->seg->conteúdo == x)
      return lst->seg;
   return BuscaR (x, lst->seg);
}
```

Exercícios

4.3.1 Que acontece se trocarmos "`while (p != NULL && p->conteúdo != x)`" por "`while (p->conteúdo != x && p != NULL)`" na função `Busca`?

4.3.2 Critique a seguinte variante da função `Busca`:

```
int achou = 0;
célula *p;
p = lst->seg;
while (p != NULL && achou != 0) {
   if (p->conteúdo == x) achou = 1;
   p = p->seg; }
if (achou) return p;
else return NULL;
```

4.3.3 LISTA SEM CABEÇA. Escreva uma versão da função `Busca` para listas sem cabeça.

4.3.4 MÍNIMO. Escreva uma função que encontre uma célula de conteúdo mínimo. Faça duas versões: uma iterativa e uma recursiva.

4.3.5 LISTA CRESCENTE. Uma lista é *crescente* se o conteúdo de cada célula não é maior que o conteúdo da célula seguinte. Escreva uma função que faça uma busca

em uma lista crescente. Faça versões para listas com e sem cabeça. Faça uma versão recursiva e outra iterativa.

4.4 Remoção de uma célula

Suponha que queremos remover uma célula de uma lista. Como devemos especificar a célula a ser removida? Parece natural apontar para a célula em questão, mas é fácil perceber o defeito da ideia. É melhor apontar para a célula *anterior* à que queremos remover. (É bem verdade que esta convenção não permite remover a primeira célula da lista, mas esta operação não é necessária no caso de listas com cabeça.) A função abaixo implementa a ideia:

```
/* Esta função recebe o endereço p de uma célula em uma
 * lista encadeada e remove da lista a célula p->seg.
 * A função supõe que p != NULL e p->seg != NULL. */
void Remove (célula *p) {
   célula *lixo;
   lixo = p->seg;
   p->seg = lixo->seg;
   free (lixo);[3]
}
```

A função não faz mais que alterar o valor de um ponteiro. Não é preciso copiar coisas de um lugar para outro, como fizemos na Seção 3.3 ao remover um elemento de um vetor. A função consome sempre o mesmo tempo, quer a célula a ser removida esteja perto do início da lista, quer esteja perto do fim.

Exercícios

4.4.1 Critique a seguinte variante da função `Remove`:

```
void Remove (célula *p, célula *lst) {
   célula *lixo;
   lixo = p->seg;
   if (lixo->seg == NULL) p->seg = NULL;
   else p->seg = lixo->seg;
   free (lixo); }
```

4.4.2 Lista sem cabeça. Escreva uma função que remova uma determinada célula de uma lista encadeada sem cabeça. (Será preciso tomar algumas decisões de projeto antes de começar a programar.)

[3] Veja Seção F.3.

4.5 Inserção de nova célula

Suponha que queremos inserir uma nova célula com conteúdo y entre a célula apontada por `p` e a seguinte. É claro que isso só faz sentido se `p` for diferente de `NULL`.

```
/* A função insere uma nova célula em uma lista encadeada
 * entre a célula p e a seguinte (supõe-se que p != NULL).
 * A nova célula terá conteúdo y. */
void Insere (int y, célula *p) {
   célula *nova;
   nova = malloc (sizeof (célula));
   nova->conteúdo = y;
   nova->seg = p->seg;
   p->seg = nova;
}
```

A função não faz mais que alterar os valores de alguns ponteiros. Não há movimentação de células para "abrir espaço" para um nova célula, como fizemos na Seção 3.4 ao inserir um novo elemento em um vetor. Assim, o tempo que a função consome não depende do ponto de inserção: tanto faz inserir uma nova célula na parte inicial da lista quanto na parte final.

A função se comporta corretamente mesmo quando a inserção se dá no fim da lista, isto é, quando `p->seg` vale `NULL`. Se a lista tem cabeça, a função pode ser usada para inserir no início da lista: basta que `p` aponte para a célula-cabeça. Mas a função não é capaz de inserir antes da primeira célula de uma lista sem cabeça.

Exercícios

4.5.1 Por que a seguinte versão de `Insere` não funciona?

```
célula nova;
nova.conteúdo = y;
nova.seg = p->seg;
p->seg = &nova;
```

4.5.2 Escreva uma função que insira uma nova célula entre a célula cujo endereço é `p` e a *anterior*.

4.5.3 LISTA SEM CABEÇA. Escreva uma função que insira uma nova célula numa dada posição de uma lista encadeada sem cabeça. (Será preciso tomar algumas decisões de projeto antes de começar a programar.)

4.6 Busca seguida de remoção ou inserção

Considere uma lista encadeada com cabeça. Dado um inteiro x, queremos remover da lista a primeira célula que contiver x; se tal célula não existe, não é preciso fazer nada.

```
/* Esta função recebe uma lista encadeada lst com cabeça
 * e remove da lista a primeira célula que contiver x,
 * se tal célula existir. */
void BuscaERemove (int x, célula *lst) {
   célula *p, *q;
   p = lst;
   q = lst->seg;
   while (q != NULL && q->conteúdo != x) {
      p = q;
      q = q->seg;
   }
   if (q != NULL) {
      p->seg = q->seg;
      free (q);
   }
}
```

No início de cada iteração, imediatamente antes da comparação de `q` com `NULL`, vale a relação `q` = `p->seg` (ou seja, `q` está sempre um passo à frente de `p`).

Suponha agora que queremos inserir na lista uma nova célula com conteúdo y imediatamente antes da primeira célula que tiver conteúdo x; se tal célula não existe, devemos inserir y no fim da lista.

```
/* Recebe uma lista encadeada lst com cabeça e insere uma
 * nova célula na lista imediatamente antes da primeira que
 * contiver x. Se nenhuma célula contiver x, a nova célula
 * será inserida no fim da lista. A nova célula terá
 * conteúdo y. */
void BuscaEInsere (int y, int x, célula *lst) {
   célula *p, *q, *nova;
   nova = malloc (sizeof (célula));
   nova->conteúdo = y;
   p = lst;
   q = lst->seg;
```

```
    while (q != NULL && q->conteúdo != x) {
       p = q;
       q = q->seg;
    }
    nova->seg = q;
    p->seg = nova;
}
```

Exercícios

4.6.1 Escreva uma versão da função `BuscaERemove` para listas encadeadas sem cabeça. (Veja Exercício 4.4.2.)

4.6.2 Escreva uma versão da função `BuscaEInsere` para listas encadeadas sem cabeça. (Veja Exercício 4.5.3.)

4.6.3 Escreva uma função para remover de uma lista encadeada todos os elementos que contêm x. Faça uma versão iterativa e uma recursiva.

4.6.4 Escreva uma função que remova de uma lista encadeada uma célula cujo conteúdo tem valor mínimo. Faça uma versão iterativa e uma recursiva.

4.7 Exercícios: manipulação de listas

A maioria dos exercícios desta seção tem duas versões: uma para lista com cabeça e outra para lista sem cabeça. Além disso, é interessante resolver cada exercício de duas maneiras: uma iterativa e uma recursiva.

4.7.1 Vetor para lista. Escreva uma função que copie um vetor para uma lista encadeada.

4.7.2 Lista para vetor. Escreva uma função que copie uma lista encadeada para um vetor.

4.7.3 Cópia. Escreva uma função que faça uma cópia de uma lista dada.

4.7.4 Comparação. Escreva uma função que decida se duas listas dadas têm o mesmo conteúdo.

4.7.5 Concatenação. Escreva uma função que concatene duas listas encadeadas (isto é, "amarre" a segunda no fim da primeira).

4.7.6 Contagem. Escreva uma função que conte o número de células de uma lista encadeada.

4.7.7 Ponto médio. Escreva uma função que receba uma lista encadeada e devolva o endereço de uma célula que esteja o mais próximo possível do ponto médio da lista. Faça isso sem calcular explicitamente o número n de células da lista e o quociente $n/2$.

4.7.8 CONTAGEM E REMOÇÃO. Escreva uma função que remova a k-ésima célula de uma lista encadeada.

4.7.9 CONTAGEM E INSERÇÃO. Escreva uma função que insira uma nova célula com conteúdo x entre a k-ésima e a $(k+1)$-ésima células de uma lista encadeada.

4.7.10 LIBERAÇÃO. Escreva uma função que aplique a função `free` a todas as células de uma lista encadeada. Estamos supondo, é claro, que cada célula da lista foi originalmente alocado por `malloc`.

4.7.11 INVERSÃO. Escreva uma função que inverta a ordem das células de uma lista encadeada (a primeira passa a ser a última, a segunda passa a ser a penúltima etc.). Faça isso sem criar novas células; apenas altere os ponteiros.

4.7.12 PROJETO DE PROGRAMAÇÃO. Digamos que um *documento* é um vetor de caracteres contendo apenas letras, espaços e sinais de pontuação. Digamos que uma *palavra* é um segmento maximal que consiste apenas de letras. Escreva uma função que imprima uma relação de todas as palavras de um documento dado juntamente com o número de ocorrências de cada palavra.

4.8 Outros tipos de listas encadeadas

Poderíamos definir vários outros tipos de listas encadeadas além do tipo básico discutido acima. Seguem dois exemplos importantes.

a. Numa lista encadeada **circular**, a última célula aponta para a primeira. A lista pode ou não ter uma célula-cabeça. (Se não tiver cabeça, as expressões "primeira célula" e "última célula" não fazem muito sentido.)

b. Numa lista **duplamente encadeada**, cada célula contém o endereço da célula anterior e o da célula seguinte. A lista pode ou não ter uma célula-cabeça, conforme as conveniências do programador.

As seguintes questões são apropriadas para qualquer tipo de lista encadeada: Em que condições a lista está vazia? Como remover a célula apontada por `p`? Como remover a célula seguinte à apontada por `p`? Como remover a célula anterior à apontada por `p`? Como inserir uma nova célula entre a apontada por `p` e a anterior? Como inserir uma nova célula entre a apontada por `p` e a seguinte?

Exercícios

4.8.1 Descreva, em C, a estrutura de uma célula de uma lista duplamente encadeada.

4.8.2 Escreva uma função que remova de uma lista duplamente encadeada a célula cujo endereço é `p`. Que dados sua função recebe? Que coisa devolve?

4.8.3 Suponha uma lista duplamente encadeada. Escreva uma função que insira uma nova célula com conteúdo y logo após a célula cujo endereço é `p`. Que dados sua função recebe? Que coisa devolve?

4.8.4 PROBLEMA DE JOSEPHUS. Imagine n pessoas dispostas em círculo. Suponha que as pessoas estão numeradas de 1 a n no sentido horário. Começando com a pessoa de número 1, percorra o círculo no sentido horário e elimine cada m-ésima pessoa enquanto o círculo tiver duas ou mais pessoas. (Veja *Josephus problem* na Wikipedia [21].) Qual o número do sobrevivente? Escreva e teste uma função que resolva o problema.

4.8.5 Leia o verbete *Linked list* na Wikipedia [21].

Capítulo 5

Filas

Fila: Fileira de pessoas que se colocam umas atrás das outras,
pela ordem cronológica de chegada a guichês
ou a quaisquer estabelecimentos onde haja grande afluência de interessados.

— *Novo Dicionário Aurélio*

Uma fila é uma sequência dinâmica, isto é, uma sequência da qual elementos podem ser removidos e na qual novos elementos podem ser inseridos. Mais especificamente, uma **fila** é uma sequência de objetos, todos do mesmo tipo, sujeita às seguintes regras de comportamento: (1) sempre que solicitamos a remoção de um elemento, o elemento removido é o primeiro da sequência e (2) sempre que solicitamos a inserção de um novo objeto, o objeto é inserido no fim da sequência.

Podemos resumir o comportamento de uma fila com a seguinte frase: o elemento removido da fila é sempre o que está lá há mais tempo. Outra maneira de dizer isso: o primeiro objeto inserido na fila é também o primeiro a ser removido. Esta política é conhecida pela abreviatura FIFO da expressão *First-In-First-Out*.

5.1 Implementação em vetor

Uma fila pode ser armazenada em um segmento `f[s..t-1]` de um vetor `f[0..N-1]`. É claro que devemos ter $0 \le s \le t \le N$. O primeiro elemento da fila está na posição `s` e o último na posição `t-1`. A fila está **vazia** se `s` é igual a `t` e **cheia** se `t` é igual a `N`. Para **remover** um elemento da fila basta dizer

```
x = f[s++];
```

o que equivale ao par de comandos "`x = f[s]; s += 1;`" (veja Seção J.1). É

claro que o programador não deve fazer isso se a fila estiver vazia. Para **inserir** um objeto `y` na fila basta dizer

```
f[t++] = y;
```

Se o programador fizer isso quando a fila já está cheia, dizemos que a fila transbordou. Em geral, a tentativa de inserir em uma fila cheia é um evento excepcional, que resulta de um mau planejamento lógico do seu programa.

Figura 5.1: O vetor `f[s..t-1]` armazena uma fila.

Exercício

5.1.1 Suponha que, diferentemente da convenção adotada no texto, a parte do vetor ocupada pela fila é `f[s..t]`. Escreva o comando que remove um elemento da fila. Escreva o comando que insere um objeto `y` na fila.

5.2 Aplicação: distâncias em uma rede

Imagine n cidades numeradas de 0 a $n-1$ e interligadas por estradas de mão única. As ligações entre as cidades são representadas por uma matriz `A` (veja Seção F.5) definida da seguinte maneira: `A[`x`][`y`]` vale `1` se existe estrada da cidade x para a cidade y e vale `0` em caso contrário. (Veja Figura 5.2.)

A **distância**[1] de uma cidade o a uma cidade x é o menor número de estradas que é preciso percorrer para ir de o a x. Nosso problema: *determinar a distância de uma dada cidade o a cada uma das outras cidades.*

As distâncias serão armazenadas em um vetor d de tal modo que d`[`x`]` seja a distância de o a x. Se for impossível chegar de o a x, podemos dizer que d`[`x`]` vale ∞. Usaremos -1 para representar ∞ (uma vez que nenhuma distância "real" pode ter valor -1).

O seguinte algoritmo usa o conceito de fila para resolver nosso problema das distâncias. Uma cidade é considerada *ativa* se já foi visitada mas as estradas

[1] A palavra *distância* já traz embutida a ideia de minimalidade. As expressões "distância mínima" e "menor distância" são redundantes.

que nela começam ainda não foram exploradas. O algoritmo mantém as cidades ativas numa fila. Em cada iteração, o algoritmo remove da fila uma cidade x e insere na fila todas as vizinhas a x que ainda não foram visitadas. Eis uma implementação do algoritmo:

```
/* A matriz A representa as interligações entre cidades
 * 0,1,..,n-1: há uma estrada (de mão única) de x a y se
 * e somente se A[x][y] == 1. A função devolve um vetor d
 * tal que d[x] é a distância da cidade o à cidade x. */
int *Distâncias (int **A, int n, int o) {
   int *d, x, y;
   int *f, s, t;
   d = malloc (n * sizeof (int));[2]
   for (x = 0; x < n; x++) d[x] = -1;
   d[o] = 0;
   f = malloc (n * sizeof (int));
   s = 0; t = 1; f[s] = o;
   while (s < t) {
      /* f[s..t-1] é uma fila de cidades */
      x = f[s++];
      for (y = 0; y < n; y++)
         if (A[x][y] == 1 && d[y] == -1) {
            d[y] = d[x] + 1;
            f[t++] = y;
         }
   }
   free (f);
   return d;
}
```

Ao longo da execução do algoritmo, o vetor `f[s..t-1]` armazena a fila de cidades, enquanto `f[0..s-1]` armazena as cidades que já saíram da fila. Para compreender o algoritmo (e provar que ele está correto), basta observar que as seguintes propriedades valem no início de cada iteração, imediatamente antes da comparação "`s < t`":

1. para cada v no vetor `f[0..t-1]`, existe um caminho de o a v, de comprimento `d[`v`]`, cujas cidades estão todas no vetor `f[0..t-1]`;

[2] Veja Seção F.2.

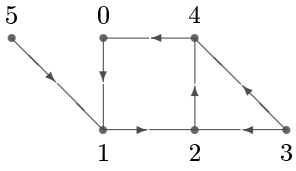
2. para cada v no vetor `f[0..t-1]`, todo caminho de o a v tem comprimento pelo menos d`[`v`]`;
3. toda estrada que começa em `f[0..s-1]` termina em `f[0..t-1]`.

Deduz-se imediatamente de 1 e 2 que, para cada v no vetor `f[0..t-1]`, o número d`[`v`]` é a distância de o a v. Para provar que as três propriedades são invariantes, é preciso observar que duas outras propriedades valem no início de cada iteração:

4. d`[f[s]]` $\le$ d`[f[s+1]]` $\le \cdots \le$ d`[f[t-1]]` e
5. d`[f[t-1]]` $\le$ d`[f[s]]` $+ 1$.

Em outras palavras, a sequência de números d`[f[s]]`, ..., d`[f[t-1]]` tem a forma $k, \ldots, k$ ou a forma $k, .., k, k{+}1, .., k{+}1$.

	0	1	2	3	4	5
0	0	1	0	0	0	0
1	0	0	1	0	0	0
2	0	0	0	0	1	0
3	0	0	1	0	1	0
4	1	0	0	0	0	0
5	0	1	0	0	0	0



	0	1	2	3	4	5
d	2	3	1	0	1	6

Figura 5.2: A matriz representa cidades $0, \ldots, 5$ interligadas por estradas de mão única. O vetor d dá as distâncias da cidade 3 a cada uma das demais.

Exercícios

5.2.1 Transbordamento. Na função `Distâncias`, o espaço alocado para o vetor `f` é suficiente? O comando "`f[t++]` = y" pode provocar o transbordamento da fila?

5.2.2 Última iteração. Suponha que os invariantes 1 a 3 valem no início da última iteração da função `Distâncias` (quando `s` é igual a `t`). Mostre que, para cada v no vetor `f[0..t-1]`, o número d`[`v`]` é a distância de o a v. Mostre também que é impossível ir da cidade o a uma cidade que esteja fora do vetor `f[0..t-1]`.

5.2.3 Primeira iteração. Verifique que os invariantes 1 a 5 valem no início da primeira iteração da função `Distâncias`.

5.2.4 Invariantes. Suponha que os invariantes 1 a 5 da função `Distâncias` valem no início de uma iteração qualquer que não a última. Mostre que elas continuam valendo no início da próxima iteração. (A prova é surpreendentemente longa e delicada.)

5.2.5 Labirinto. Imagine um tabuleiro quadrado 10–por–10. As casas "livres" são

marcadas com 0 e as casas "bloqueadas" com -1. As casas $(1,1)$ e $(10,10)$ estão livres. Ajude uma formiga que está na casa $(1,1)$ a chegar à casa $(10,10)$. Em cada passo, a formiga só pode se deslocar para uma casa livre que esteja à direita, à esquerda, acima ou abaixo da casa em que está.

5.3 Implementação circular

No problema discutido na seção anterior, o vetor que abriga a fila não precisa ter mais componentes que o número total de cidades, pois cada cidade entra na fila no máximo uma vez. Em geral, entretanto, é difícil prever o espaço necessário para abrigar a fila. Nesses casos, é mais seguro implementar a fila de maneira *circular*. Suponha que os elementos da fila estão dispostos no vetor `f[0..N-1]` de uma das seguintes maneiras:

`f[s..t-1]` ou `f[s..N-1] f[0..t-1]`

(veja Figura 5.3). Teremos sempre $0 \le s < N$ e $0 \le t < N$, mas não podemos supor que $s \le t$. A fila está **vazia** se $t = s$ e **cheia** se

$$t+1 = s \quad \text{ou} \quad t+1 = N \ \text{ e } \ s = 0,$$

ou seja, se $(t+1)\,\%\,N = s$.[3] A posição `t` ficará sempre desocupada, para que possamos distinguir uma fila cheia de uma vazia. Para remover um elemento da fila basta fazer

```
x = f[s++];
if (s == N) s = 0;
```

(supondo que a fila não está vazia). Para inserir um objeto `y` na fila (supondo que ela não está cheia), faça

```
f[t++] = y;
if (t == N) t = 0;
```

Exercício

5.3.1 Considere a manipulação de uma fila circular. Escreva uma função que devolva o tamanho da fila. Escreva uma função que remova um elemento da fila e devolva esse elemento; se a fila estiver vazia, não faça nada. Escreva uma função que verifique se a fila está cheia e em caso negativo insira um objeto dado na fila. (Lembre-se de que uma fila é um pacote com três objetos: um vetor e dois índices. Não use variáveis globais.)

[3] O valor da expressão `a%b` é o resto da divisão de `a` por `b`, ou seja, $a - b\lfloor a/b \rfloor$.

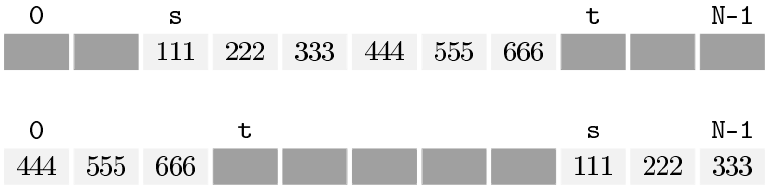


Figura 5.3: Fila circular. Na primeira parte da figura, a fila está armazenada no vetor `f[s..t-1]`. Na segunda parte, a fila está armazenada no vetor `f[s..N-1]` concatenado com `f[0..t-1]`.

5.4 Implementação em lista encadeada

Considere agora a implementação de uma fila em uma lista encadeada. Digamos que as células da lista são do tipo `célula`:

```
typedef struct cel {
   int         valor;
   struct cel *seg;
} célula;
```

É preciso tomar algumas decisões de projeto sobre a maneira de acomodar a fila na lista. Vamos supor que nossa lista encadeada não tem cabeça, que o primeiro elemento da fila ficará na primeira célula e que o último elemento da fila ficará na última célula.

Para manipular a fila, precisamos de dois ponteiros: um ponteiro `s` apontará o primeiro elemento da fila e um ponteiro `t` apontará o último. A fila estará vazia se `s = t = NULL`. Suporemos que `s = NULL` sempre que `t = NULL` e vice-versa. Uma fila vazia pode ser criada assim:

```
célula *s, *t;
s = t = NULL;
```

Para remover um elemento da fila (supondo que ela não está vazia), será preciso passar à função de remoção os *endereços* das variáveis `s` e `t` para que os valores dessas variáveis possam ser alterados:

```
int Remove (célula **es, célula **et) {
   célula *p;
   int x;
   p = *es;
   x = p->valor;
```

```
    *es = p->seg;
    free (p);
    if (*es == NULL) *et = NULL;
    return x;
}
```

A função de inserção precisa levar em conta a possibilidade de inserção em fila vazia:

```
void Insere (int y, célula **es, célula **et) {
    célula *nova;
    nova = malloc (sizeof (célula));
    nova->valor = y;
    nova->seg = NULL;
    if (*et == NULL) *et = *es = nova;
    else {
        (*et)->seg = nova;
        *et = nova;
    }
}
```

Exercícios

5.4.1 Implemente uma fila em uma lista encadeada com célula-cabeça.

5.4.2 Implemente uma fila em uma lista encadeada *circular* com célula-cabeça. O primeiro elemento da fila ficará na segunda célula e o último elemento ficará na célula anterior à cabeça. Para manipular a fila basta conhecer o endereço `ff` da célula-cabeça.

5.4.3 Implemente uma fila em uma lista duplamente encadeada sem célula-cabeça. Mantenha um ponteiro para a primeira célula e um ponteiro para a última.

5.4.4 Leia o verbete *Queue* na Wikipedia [21].

Capítulo 6

Pilhas

Pilha: Porção de objetos dispostos uns sobre os outros.
— *Dicionário Houaiss*

Uma pilha é uma sequência dinâmica, isto é, uma sequência da qual elementos podem ser removidos e na qual novos elementos podem ser inseridos. Mais especificamente, uma **pilha** é uma sequência de objetos, todos do mesmo tipo, sujeita às seguintes regras de comportamento: (1) sempre que solicitamos a remoção de um elemento, o elemento removido é o último da sequência e (2) sempre que solicitamos a inserção de um novo objeto, o objeto é inserido no fim da sequência.

Podemos resumir o comportamento de uma pilha com a seguinte frase: o elemento removido da pilha é sempre o que está lá há menos tempo. Outra maneira de dizer isso: o primeiro objeto inserido na pilha é o último a ser removido. Esta política é conhecida pela abreviatura LIFO da expressão *Last-In-First-Out*.

6.1 Implementação em vetor

Suponha que nossa pilha está armazenada em um vetor `p[0..N-1]`. A parte do vetor efetivamente ocupada pela pilha é `p[0..t-1]`. O índice `t-1` define o **topo** da pilha.

A pilha está **vazia** se `t` vale 0 e **cheia** se `t` vale `N`. Para **remover** um elemento da pilha, ou seja, para **desempilhar** um elemento, faça

```
x = p[--t];
```

o que equivale ao par de comandos "`t -= 1; x = p[t];`" (veja Seção J.1). É claro que o programador não deve fazer isto se a pilha estiver vazia. Para **consultar**

a pilha sem desempilhar basta fazer `x = p[t-1]`. Para **empilhar** um objeto `y`, ou seja, para **inserir** `y` na pilha faça

```
p[t++] = y;
```

o que equivale ao par de comandos "`p[t] = y; t += 1;`". Antes de empilhar, é preciso ter certeza de que a pilha não está cheia. Em geral, a tentativa de inserir em uma pilha cheia é um indício de mau planejamento lógico do seu programa.

Figura 6.1: O vetor `p[0..t-1]` armazena uma pilha.

Exercícios

6.1.1 Suponha que, diferentemente da convenção adotada no texto, a parte do vetor ocupada pela pilha é `p[0..t]`. Escreva o comando que remove um elemento da pilha. Escreva o comando que insere um objeto na pilha.

6.1.2 Inversão de palavras. Escreva uma função que inverta a ordem das letras de cada palavra de uma sentença, preservando a ordem das palavras. Suponha que as palavras da sentença são separadas por espaços. A aplicação da operação à sentença `AMU MEGASNEM ATERCES`, por exemplo, deve produzir `UMA MENSAGEM SECRETA`.

6.1.3 Permutações produzidas pelo desempilhar [10, sec. 2.2.1]. Suponha que os números inteiros $1, 2, 3, 4$ são colocados, nesta ordem, numa pilha inicialmente vazia. Depois de cada operação de empilhar, você pode retirar zero ou mais elementos da pilha. Cada número retirado da pilha é impresso numa folha de papel. Por exemplo, a sequência de operações $\mathsf{E}, \mathsf{E}, \mathsf{D}, \mathsf{E}, \mathsf{D}, \mathsf{D}, \mathsf{E}, \mathsf{D}$, onde E significa "empilhar o próximo número da sequência" e D significa "desempilhar", produz a impressão da sequência $2, 3, 1, 4$. Quais das 24 permutações de $1, 2, 3, 4$ podem ser obtidas desta maneira?

6.2 Aplicação: parênteses e chaves

Considere o problema de decidir se uma dada sequência de parênteses e chaves é bem-formada. Por exemplo, a sequência

```
( ( ) { ( ) } )
```

é bem-formada, enquanto a sequência `( { ) }` é malformada.

Suponha que a sequência de parênteses e chaves está armazenada em uma string `s` (veja Apêndice G). De acordo com as convenções da linguagem C, o último elemento da string é o caractere nulo `'\0'` (veja Apêndice B).

```
/* Esta função devolve 1 se a string s contém uma sequência
 * bem-formada de parênteses e chaves e devolve 0 se
 * a sequência está malformada. */
int BemFormada (char s[]) {
   char *p; int t;
   int n, i;
   n = strlen (s);
   p = malloc (n * sizeof (char));
   t = 0;
   for (i = 0; s[i] != '\0'; i++) {
      /* p[0..t-1] é uma pilha */
      switch (s[i]) {
         case ')': if (t != 0 && p[t-1] == '(') --t;
                   else return 0;
                   break;
         case '}': if (t != 0 && p[t-1] == '{') --t;
                   else return 0;
                   break;
         default: p[t++] = s[i];
      }
   }
   return t == 0;[1]
}
```

(Eu deveria ter invocado `free (p)` antes de cada `return`; só não fiz isso para não obscurecer a lógica da função.) A pilha `p` jamais transborda porque nunca terá mais elementos do que o número, `n`, de caracteres de `s`.

Exercícios

6.2.1 A função `BemFormada` funciona corretamente se `s` tem apenas dois elementos? apenas um? nenhum?

6.2.2 Mostre que o processo iterativo na função `BemFormada` tem o seguinte invariante:

[1] Veja Seção J.2.

no início de cada iteração, a string `s` está bem-formada se e somente se a sequência `p[0..t-1] s[i...]`, formada pela concatenação de `p[0..t-1]` com `s[i...]`, estiver bem-formada.

6.3 Aplicação: notação posfixa

Expressões aritméticas são usualmente escritas em notação *infixa*: os operadores ficam entre os operandos. Na notação *posfixa* (ou *polonesa*) os operadores ficam depois dos operandos. Os exemplos da Figura 6.2 esclarecem o conceito. (A propósito, veja o Exercício 14.2.7.)

notação infixa	notação posfixa
`(A+B*C)`	`ABC*+`
`(A*(B+C)/D-E)`	`ABC+*D/E-`
`(A+B*(C-D*(E-F)-G*H)-I*3)`	`ABCDEF-*-GH*-*+I3*-`
`(A+B*C/D*E-F)`	`ABC*D/E*+F-`
`(A*(B+(C*(D+(E*(F+G))))))`	`ABCDEFG+*+*+*`

Figura 6.2: Expressões aritméticas em notação infixa e notação posfixa. A notação posfixa dispensa parênteses. Os operandos (`A`, `B` etc.) aparecem na mesma ordem nas duas notações.

Nosso problema: traduzir para notação posfixa uma expressão infixa dada. Para simplificar, suporemos que a expressão infixa está correta e contém apenas letras, parênteses e os símbolos `+`, `-`, `*` e `/`. Suporemos também que cada nome de variável tem uma letra apenas. Finalmente, suporemos que a expressão toda está embrulhada em um par de parênteses. Se a expressão está armazenada na string `infix`, o primeiro caractere da string é `'('` e os dois últimos são `')'` e `'\0'`.

Usaremos uma pilha para resolver o problema de tradução. Como a expressão infixa está embrulhada em parênteses, não será preciso preocupar-se com pilha vazia.

```
/* A função abaixo recebe uma expressão infixa infix e
 * devolve a correspondente expressão posfixa. */
char *InfixaParaPosfixa (char infix[]) {
   char *posfix, x;
```

```
   char *p; int t;
   int n, i, j;
   n = strlen (infix);
   posfix = malloc (n * sizeof (char));
   p = malloc (n * sizeof (char));
   t = 0; p[t++] = infix[0];  /* empilha '(' */
   for (j = 0, i = 1; /*X*/ infix[i] != '\0'; i++) {
      /* p[0..t-1] é uma pilha de caracteres */
      switch (infix[i]) {
         case '(': p[t++] = infix[i];  /* empilha */
                   break;
         case ')': while (1) {  /* desempilha */
                      x = p[--t];
                      if (x == '(') break;
                      posfix[j++] = x; }
                   break;
         case '+':
         case '-': while (1) {
                      x = p[t-1];
                      if (x == '(') break;
                      --t;  /* desempilha */
                      posfix[j++] = x; }
                   p[t++] = infix[i];  /* empilha */
                   break;
         case '*':
         case '/': while (1) {
                      x = p[t-1];
                      if (x == '(' || x == '+' || x == '-')
                         break;
                      --t;
                      posfix[j++] = x; }
                   p[t++] = infix[i];
                   break;
         default: posfix[j++] = infix[i]; }
   }
   free (p);
   posfix[j] = '\0';
   return posfix;
}
```

```
infix[0..i-1]   p[0..t-1]  posfix[0..j-1]
(               (
(A              (          A
(A*             (*         A
(A*(            (*(        A
(A*(B           (*(        AB
(A*(B*          (*(*       AB
(A*(B*C         (*(*       ABC
(A*(B*C+        (*(+       ABC*
(A*(B*C+D       (*(+       ABC*D
(A*(B*C+D)      (*         ABC*D+
(A*(B*C+D))                ABC*D+*
```

Figura 6.3: Resultado da aplicação da função `InfixaParaPosfixa` à expressão infixa `(A*(B*C+D))`. A figura registra os valores das variáveis no início de cada iteração (ou seja, a cada passagem pelo ponto `X` do código). Constantes e variáveis vão diretamente de `infix` para `posfix`. Todo parêntese esquerdo vai para a pilha. Ao encontrar um parêntese direito, a função remove tudo da pilha até o primeiro parêntese esquerdo que encontrar. Ao encontrar `+` ou `-`, a função desempilha tudo até encontrar um parêntese esquerdo. Ao encontrar `*` ou `/`, desempilha tudo até um parêntese esquerdo ou um `+` ou um `-`.

Exercícios

6.3.1 Aplique à expressão infixa `(A+B)*D+E/(F+A*D)+C` o algoritmo de conversão para notação posfixa.

6.3.2 Na função `InfixaParaPosfixa`, suponha que a string `infix` tem n caracteres (sem contar o caractere nulo final). Que altura a pilha pode atingir, no pior caso? Em outras palavras, qual o valor máximo da variável `t`? Que acontece se o número de parênteses for limitado (menor que 10, por exemplo)?

6.3.3 Reescreva o código da função `InfixaParaPosfixa` de maneira um pouco mais compacta, sem os "`while (1)`". Tire proveito dos recursos sintáticos da linguagem C.

6.3.4 Reescreva a função `InfixaParaPosfixa` sem supor que a expressão infixa está embrulhada em um par de parênteses.

6.3.5 Reescreva a função `InfixaParaPosfixa` supondo que a expressão infixa pode estar incorreta.

6.3.6 Reescreva a função `InfixaParaPosfixa` supondo que a expressão pode ter parênteses e chaves.

6.3.7 Valor de expressão posfixa. Suponha dada uma expressão aritmética em notação posfixa sujeita às seguintes restrições: cada variável consiste em uma única letra do conjunto `A..Z`; não há constantes; os únicos operadores são `+`, `-`, `*`, `/` (todos exigem *dois* operandos). Suponha dado também um vetor inteiro `val`, indexado por

`A..Z`, que dá os valores das variáveis. Escreva uma função que calcule o valor da expressão. Cuidado com divisões por zero.

6.4 Implementação em lista encadeada

Uma pilha pode ser implementada em uma lista encadeada. Digamos que as células da lista são do tipo `célula`:

```
typedef struct cel {
   int         valor;
   struct cel *seg;
} célula;
```

Suporemos que nossa lista tem uma célula-cabeça e que o topo da pilha está na segunda célula (e não na última). Uma pilha (vazia) pode ser criada assim:

```
célula cabeça;
célula *p;
p = &cabeça;
p->seg = NULL;
```

Para manipular a pilha, basta dispor do ponteiro `p`, cujo valor será sempre `&cabeça`. A pilha estará **vazia** se `p->seg` for `NULL`. Eis a função que insere um número `y` na pilha:

```
void Empilha (int y, célula *p) {
   célula *nova;
   nova = malloc (sizeof (célula));
   nova->valor = y;
   nova->seg = p->seg;
   p->seg = nova;
}
```

Eis uma função que remove um elemento de uma pilha não vazia:

```
int Desempilha (célula *p) {
   int x; célula *q;
   q = p->seg;
   x = q->valor;
   p->seg = q->seg;
   free (q);
   return x;
}
```

Exercícios

6.4.1 Implemente uma pilha em uma lista encadeada *sem* célula-cabeça. A pilha será especificada pelo endereço da primeira célula da lista.

6.4.2 Reescreva as funções `BemFormada` e `InfixaParaPosfixa` (Seções 6.2 e 6.3 respectivamente) armazenando a pilha em uma lista encadeada.

6.4.3 Leia o verbete *Stack (data structure)* na Wikipedia [21].

6.5 A pilha de execução de um programa

Todo programa C é composto por uma ou mais funções, sendo `main` a primeira função a ser executada. Para executar um programa, o computador usa uma "pilha de execução". A operação pode ser descrita conceitualmente da seguinte maneira. Ao encontrar a invocação de uma função, o computador cria um novo "espaço de trabalho", que contém todos os parâmetros e todas as variáveis locais da função. Esse espaço de trabalho é colocado na pilha de execução (sobre o espaço de trabalho que invocou a função) e a execução da função começa (confinada ao seu espaço de trabalho). Quando a execução da função termina, o seu espaço de trabalho é retirado da pilha e descartado. O espaço de trabalho que estiver agora no topo da pilha é reativado e a execução é retomada do ponto em que havia sido interrompida.

Considere o seguinte exemplo:

```
int G (int a, int b) {
   return a + b;
}

int F (int i, int j, int k) {
   int x;
   x = /*2*/ G (i, j) /*3*/;
   return x + k;
}

int main (void) {
   int i, j, k, y;
   i = 111; j = 222; k = 444;
   y = /*1*/ F (i, j, k) /*4*/;
   printf ("%d\n", y);
   return EXIT_SUCCESS;
}
```

[2]

[2] Veja Seção K.1.

O programa é executado da seguinte maneira:

1. Um espaço de trabalho é criado para a função `main` e colocado na pilha de execução. O espaço contém as variáveis locais `i`, `j`, `k` e `y`. A execução de `main` começa.
2. No ponto `1`, a execução de `main` é suspensa e um espaço de trabalho para a função `F` é colocado na pilha. Esse espaço contém os parâmetros `i`, `j`, `k` da função (com valores `111`, `222` e `444` respectivamente) e a variável local `x`. Começa então a execução de `F`.
3. No ponto `2`, a execução de `F` é suspensa e um espaço de trabalho para a função `G` é colocado na pilha. Esse espaço contém os parâmetros `a` e `b` da função (com valores `111` e `222` respectivamente). Em seguida, começa a execução de `G`.
4. Quando a execução de `G` termina, a função devolve `333`. O espaço de trabalho de `G` é removido da pilha e descartado. O espaço de trabalho de `F` (que agora está no topo da pilha de execução) é reativado e a execução é retomada no ponto `3`. A primeira instrução executada é "`x = 333;`".
5. Quando a execução de `F` termina, a função devolve `777`. O espaço de trabalho de `F` é removido da pilha e descartado. O espaço de trabalho de `main` é reativado e a execução é retomada no ponto `4`. A primeira instrução executada é "`y = 777;`".

No nosso exemplo, `F` e `G` são funções distintas. Mas tudo funcionaria da mesma maneira se `F` e `G` fossem idênticas, ou seja, se `F` fosse uma função recursiva.

Exercício

6.5.1 Escreva uma função iterativa que simule o comportamento da função recursiva abaixo. Use uma pilha.

```
int TTT (int x[], int n) {
   if (n == 0) return 0;
   if (x[n] > 0) return x[n] + TTT (x, n - 1);
   else return TTT (x, n - 1); }
```