3.4.6. Converta o seguinte esquema de programa recursivo numa versão iterativa que não use uma pilha. f(n) é uma função que retorna TRUE ou FALSE, com base no valor de n, e g(n) é uma função que retorna um valor do mesmo tipo de n (sem modificar n).

```
rec(n)
int n;
{
    if (f(n) == FALSE) {
        /* qualquer grupo de comando em C que */
        /* nao mude o valor de n              */
        rec(g(n));
    } /* fim if */
} /* fim rec */
```

Generalize seu resultado para o caso em que rec retorna um valor.

3.4.7. Seja f(n) uma função e g(n) e h(n) funções que retornam um valor do mesmo tipo de n sem modificar n. Permita que (stmts) represente qualquer grupo de comando em C que não modifiquem o valor de n. Prove que o esquema do programa recursivo rec é equivalente ao esquema iterativo iter:

```
rec(n)
int n;
{
    if (f(n) == FALSE) {
        (stmts);
        rec(g(n));
        rec(h(n));
    } /* fim if */
} /* fim rec */

struct stack {
    int top;
    int nvalues[MAXSTACK];
};

iter(n)
int n;
{
    struct stack s;

    s.top = -1;
```

```
        push(&s, n);
        while(empty(&s) == FALSE ) {
            n = pop(&s);
            if (f(n) == FALSE) {
                (stmts);
                push(&s, h(n));
                push(&s, g(n));
            } /* fim if */
        } /* fim while */
    } /* fim iter */
```

Prove que os comandos if em iter podem ser substituídos pela seguinte repetição:

```
while (f(n) == FALSE) {
    (stmts)
    push(&s, h(n));
    n = g(n);
} /* fim while */
```

3.5. EFICIÊNCIA DA RECURSIVIDADE

Em geral, uma versão não-recursiva de um programa executará com mais eficiência, em termos de tempo e espaço, do que uma versão recursiva. Isso acontece porque o trabalho extra dispendido para entrar e sair de um bloco é evitado na versão não-recursiva. Conforme constatamos, é possível identificar um número considerável de variáveis locais e temporárias que não precisam ser salvas e restauradas pelo uso de uma pilha. Num programa não-recursivo, essa atividade de empilhamento desnecessária pode ser eliminada. Entretanto, num procedimento recursivo, geralmente o compilador não consegue identificar essas variáveis, e elas são, portanto, empilhadas e desempilhadas para evitar problemas.

Contudo, verificamos também que, às vezes, uma solução recursiva é o método mais natural e lógico de solucionar um problema. Não é certo que um programador consiga desenvolver a solução não-recursiva para o problema das Torres de Hanoi diretamente a partir da declaração do problema. Podemos fazer um comentário semelhante sobre o problema de converter formas prefixas em posfixas, em que a solução recursiva flui diretamente das

definições. Uma solução não-recursiva envolvendo pilhas é mais difícil de desenvolver e mais propensa a erros.

Dessa forma, ocorre um conflito entre a eficiência da máquina e a do programador. Com o custo da programação aumentando consideravelmente e o custo da computação diminuindo, chegamos ao ponto em que, na maioria dos casos, não compensa para o programador construir exaustivamente uma solução não-recursiva para um problema que é resolvido com mais naturalidade de forma recursiva. Evidentemente, um programador incompetente e supostamente inteligente pode surgir com uma solução recursiva complicada para um problema simples, que pode ser resolvido de imediato por métodos não-recursivos. (Um exemplo disso é a função fatorial ou até mesmo a busca binária.) Entretanto, se um programador competente identificar uma solução recursiva como o método mais simples e mais objetivo para solucionar determinado problema, provavelmente não compensará perder tempo e esforço descobrindo um método mais eficiente.

Entretanto, nem sempre esse é o caso. Se um programa precisa ser executado com freqüência (em geral, computadores inteiros são dedicados a executar continuamente o mesmo programa), de forma que o aumento de eficiência na velocidade de execução aumente consideravelmente a produtividade operacional, o investimento extra no tempo de programação será compensado. Mesmo nesses casos, é provável que seja melhor criar uma versão não-recursiva simulando e transformando a solução recursiva, do que tentar criar uma solução não-recursiva a partir do enunciado do problema.

Para fazer isso com mais eficiência, é necessário primeiro escrever uma rotina recursiva e, em seguida, sua versão simulada, incluindo todas as pilhas e temporárias. Depois que isso for feito, elimine todas as pilhas e variáveis supérfluas. A versão final será um refinamento do programa original e, com certeza, muito mais eficiente. Evidentemente, a eliminação de toda operação supérflua e redundante aumenta a eficiência do programa resultante. Entretanto, toda transformação aplicada num programa é mais uma abertura pela qual pode surgir um erro imprevisto.

Quando uma pilha não pode ser eliminada da versão não-recursiva de um programa e quando a versão recursiva não contém nenhum dos parâmetros adicionais ou variáveis locais, a versão recursiva pode ser tão ou mais veloz que a versão não-recursiva, sob um compilador eficiente. As Torres de Hanoi representam um exemplo desse programa recursivo. O fatorial, cuja versão não-recursiva não precisa de uma pilha, e o cálculo de números de Fibonacci, que contém uma segunda chamada recursiva desne-

cessária (e não precisa de uma pilha também), é um exemplo onde a recursividade deve ser evitada numa implementação prática. Examinaremos outro exemplo de recursividade eficiente (no percurso de árvores ordenadas) na Seção 5.2.

Outro aspecto a lembrar é que as chamadas explícitas a pop, push e empty, bem como os testes de ocorrência de underflow e estouro, são bastante dispendiosas. Na realidade, elas podem freqüentemente ultrapassar o custo adicional da recursividade. Sendo assim, para maximizar a real eficiência do tempo de execução de uma tradução não-recursiva, essas chamadas devem ser substituídas por código em-linha e os testes de ocorrência de underflow/estouro devem ser eliminados se soubermos que estamos operando dentro dos limites do vetor.

As idéias e transformações que desenvolvemos ao apresentar a função fatorial e o problema das Torres de Hanoi podem ser aplicadas a problemas mais complexos, cuja solução não-recursiva não esteja prontamente evidente. Até onde uma solução recursiva (real ou simulada) pode ser transformada numa solução direta dependerá do problema em questão e da engenhosidade do programador.

EXERCÍCIOS

3.5.1. Execute as versões recursiva e não-recursiva da função fatorial das Seções 3.2 e 3.4, e examine quanto espaço e tempo cada uma exige quando n fica maior.

3.5.2. Faça o mesmo que no Exercício 3.5.1 para o problema das Torres de Hanoi.

Capítulo 4

Filas e Listas

Este capítulo apresenta a fila e a fila de prioridade, duas importantes estruturas de dados usadas freqüentemente para simular situações do mundo real. Os conceitos de pilha e fila são, na seqüência, estendidos a uma nova estrutura, a lista. Várias formas de listas e suas operações associadas serão examinadas e apresentaremos várias aplicações.

4.1. A FILA E SUA REPRESENTAÇÃO SEQÜENCIAL

Uma fila é um conjunto ordenado de itens a partir do qual podem-se eliminar itens numa extremidade (chamada início da fila) e no qual podem-se inserir itens na outra extremidade (chamada final da fila).

A Figura 4.1.1a ilustra uma fila contendo três elementos, A, B e C. A é o início da fila e C é o final. Na Figura 4.1.1b, foi eliminado um elemento da fila. Como os elementos só podem ser eliminados a partir do início da fila, A é removido e B passa a ocupar o início da fila. Na Figura 4.1.1c, quando os itens D e E forem inseridos, essa operação deverá ocorrer no final da fila.

Como D foi inserido na fila antes de E, ele será removido em primeiro lugar. O primeiro elemento inserido numa fila é o primeiro a ser removido. Por essa razão, uma fila é ocasionalmente chamada lista fifo (first-in,

first-out — o primeiro que entra é o primeiro a sair), ao contrário de uma pilha, que é uma lista lifo (last-in, first-out — o último a entrar é o primeiro a sair). Existem muitos exemplos de fila no mundo real. Uma fila de banco ou no ponto de ônibus e um grupo de carros aguardando sua vez no pedágio são exemplos conhecidos de filas.

Três operações primitivas podem ser aplicadas a uma fila. A operação insert(q,x) insere o item x no início da fila q. A operação x = remove(q) elimina o último elemento da fila q e define x com seu conteúdo. A terceira operação, empty(q), retorna falso ou verdadeiro, dependendo de a fila conter ou não algum elemento. A fila que aparece na Figura 4.1.1 pode ser obtida pela seguinte seqüência de operações. Pressupomos que a fila esteja inicialmente vazia.

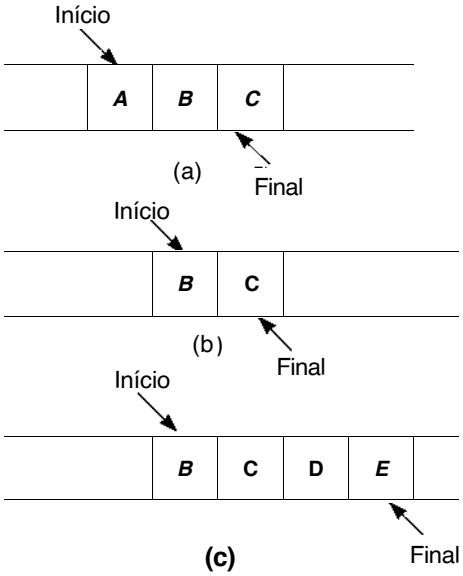

Figura 4.1.1 Uma fila.

```
insert(q, A);
insert(q, B);
insert(q, C);          (Figura 4.1.1a)
x = remove(q);         (Figura 4.1.1b; x eh definido com A)
insert(q, D);
insert(q, E)           (Figura 1.1.1c)
```

A operação inseri sempre pode ser executada, uma vez que não há limite para o número de elementos que uma fila pode conter. A operação remove, contudo, só pode ser aplicada se a fila não estiver vazia; não existe um método para remover um elemento de uma fila sem elementos. O resultado de uma tentativa inválida de remover um elemento de uma fila vazia é chamado de underflow. Evidentemente, a operação empty é sempre aplicável.

A FILA COMO UM TIPO DE DADO ABSTRATO

A representação de uma fila como um tipo de dado abstrato é simples. Usamos eltype pra indicar o tipo do elemento da fila e parametrizamos o tipo da fila com eltype.

```
abstract typedef«eltype» QUEUE(eltype);

abstract empty(g)
QUEUE(eltype) q;
postcondition empty == (len(q) == 0);
abstract eltype remove(q)
QUEUE(eltype) q;
precondition empty(q) == FALSE;
postcondition remove == first(q');
              q == sub(q', 1, len(q') - 1);

abstract insert(q, elt)
QUEUE(eltype) q;
eltype elt;
postcondition q == q'+ <elt>;
```

IMPLEMENTAÇÃO DE FILAS EM C

Como uma fila pode ser representada em C? Uma idéia é usar um vetor para armazenar os elementos da fila e duas variáveis, front e rear, para armazenar as posições dentro do vetor do primeiro e último elementos da fila. Poderíamos declarar uma fila q de inteiros com: