

# Capítulo 15

## Pilhas e filas

Neste capítulo, vamos discutir a implementação de duas estruturas de dados largamente utilizadas em computação: pilha e fila. Nestas estruturas, a inserção e a retirada de elementos seguem regras específicas. A pilha segue a regra *LIFO* (*last in, first out*): o último elemento a entrar é o primeiro a sair. A fila segue a regra *FIFO* (*first in, first out*): o primeiro elemento a entrar é o primeiro a sair. Com estas regras específicas, a implementação das estruturas é facilitada. Ao mesmo tempo, essas regras atendem as demandas de diversas aplicações que precisam armazenar conjunto de valores, por isso são estruturas populares.

### 15.1 Pilhas

Uma das estruturas de dados mais simples é a pilha. Possivelmente por essa razão, é a estrutura de dados mais utilizada em programação. A ideia fundamental da pilha é que todo o acesso a seus elementos é feito através do seu *topo*. Assim, quando um elemento novo é introduzido na pilha, passa a ser o elemento do topo, e o único elemento que pode ser removido da pilha é o do topo.

Para entender o funcionamento de uma estrutura de pilha, podemos fazer uma analogia com uma pilha de pratos. Se quisermos adicionar um prato na pilha, nós o colocamos no topo. Para pegar um prato da pilha, retiramos o do topo. Assim, temos que retirar o prato do topo para ter acesso ao próximo prato. A estrutura de pilha funciona de maneira análoga. Cada novo elemento é inserido no topo e só temos acesso ao elemento do topo da pilha. Isto faz com que os elementos da pilha sejam retirados na ordem inversa à ordem em que foram introduzidos (regra *LIFO*, *last in, first out*).

Existem duas operações básicas que devem ser implementadas numa estrutura de pilha: a operação para empilhar um novo elemento, inserindo-o no topo, e a operação para desempilhar um elemento, removendo-o do topo. É comum nos referirmos a essas duas operações pelos termos em inglês *push* (empilhar) e *pop* (desempilhar). A Figura 15.1 ilustra uma pilha de valores inteiros após a seguinte sequência de operações: *push 4, push 3, push 7, pop, push 6, push 5*, nesta ordem.

O exemplo de utilização de pilha mais próximo é a própria pilha de execução da linguagem C. As variáveis locais das funções são dispostas numa pilha (a pilha de execução) e uma função só tem acesso às variáveis da função que está no topo (não é possível acessar as variáveis locais às outras funções).

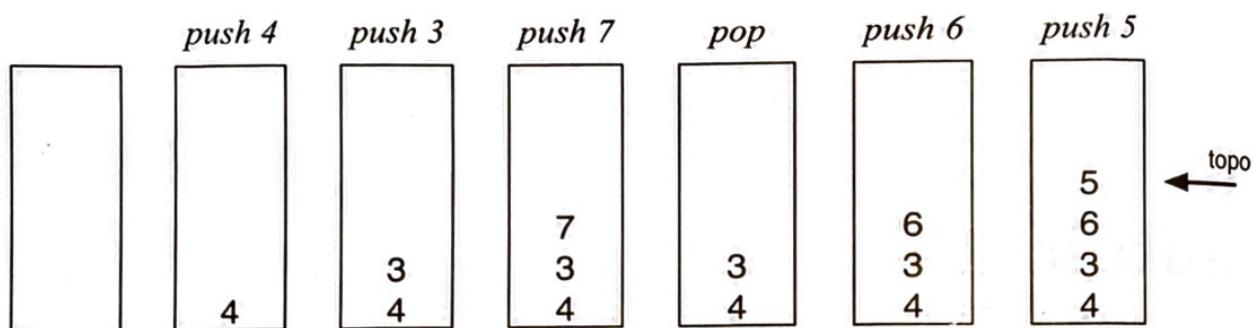


Figura 15.1: Funcionamento da pilha.

Há diferentes implementações possíveis de uma pilha, que se distinguem pela natureza dos seus elementos, pela maneira como os elementos são armazenados e pelas operações disponíveis para o tratamento da pilha.

### 15.1.1 Interface do tipo pilha

Vamos considerar duas implementações de pilha: usando um vetor e usando uma lista encadeada. Para simplificar esta exposição, consideraremos uma pilha que armazena valores reais. Independentemente da estratégia de implementação, podemos definir a interface do tipo abstrato que representa uma estrutura de pilha. Ela é composta pelas operações que estarão disponibilizadas para manipular e acessar as informações da pilha. Neste exemplo, vamos considerar a implementação de cinco operações:

- Criar uma pilha vazia
- Inserir um elemento no topo (push)
- Remover o elemento do topo (pop)
- Verificar se a pilha está vazia
- Liberar a estrutura de pilha

O arquivo “pilha.h”, que representa a interface do tipo, pode conter o seguinte código:

```
typedef struct pilha Pilha;

Pilha* pilha_cria (void);
void pilha_push (Pilha* p, float v);
float pilha_pop (Pilha* p);
int pilha_vazia (Pilha* p);
void pilha_libera (Pilha* p);
```

A função `cria` aloca dinamicamente a estrutura da pilha, inicializa seus campos e retorna seu ponteiro; as funções `push` e `pop` inserem e removem, respectivamente, um valor na pilha; a função `vazia` informa se a pilha está ou não vazia; e a função `libera` destrói a pilha, liberando toda a memória usada pela estrutura.

### 15.1.2 Implementação de pilha com vetor

A implementação com vetor é bastante simples. Devemos ter um vetor (`vet`) para armazenar os elementos da pilha. Os elementos inseridos ocupam as primeiras posições do vetor. Desta forma, se temos `n` elementos armazenados na pilha, o elemento `vet[n-1]` representa o elemento do topo. Em aplicações computacionais que precisam de uma estrutura de pilha, é comum saber de antemão o número máximo de elementos que podem estar armazenados simultaneamente na pilha, isto é, a estrutura da pilha tem um limite conhecido. Nestes casos, a implementação da pilha pode ser feita usando um vetor estático, mas é necessário verificar a capacidade da pilha para inserção. Neste exemplo, vamos implementar a pilha usando um vetor dinâmico, sem impor um limite prévio no número de elementos na pilha.

A estrutura que representa o tipo pilha deve, portanto, ser a estrutura que representa um vetor dinâmico:

```
struct pilha {
    int n;          /* número de elementos armazenados */
    int dim;        /* dimensão corrente do vetor */
    float* vet;     /* vetor dos elementos */
};
```

A função para criar a pilha aloca dinamicamente essa estrutura, escolhe uma dimensão inicial para o vetor e inicializa a pilha como sendo vazia, isto é, com o número de elementos igual a zero.

```
Pilha* pilha_cria (void)
{
    Pilha* p = (Pilha*) malloc(sizeof(Pilha));
    p->dim = 2;      /* dimensão inicial */
    p->vet = (float*) malloc(p->dim*sizeof(float));
    p->n = 0;        /* inicializa com zero elementos */
    return p;
}
```

Para inserir um elemento na pilha, usamos a próxima posição livre do vetor, verificando a necessidade de realocação do vetor:

```
void pilha_push (Pilha* p, float v)
{
    if (p->n == p->dim) { /* capacidade esgotada */
        p->dim *= 2;
        p->vet = (float*) realloc(p->vet, p->dim*sizeof(float));
    }
    /* insere elemento na próxima posição livre */
    p->vet[p->n++] = v;
}
```

A função `pop` retira o elemento do topo da pilha, fornecendo seu valor como retorno. Vamos assumir que esta função nunca será chamada quando a pilha estiver vazia.

```
float pilha_pop (Pilha* p)
{
    float v = p->vet[--p->n]; /* retira elemento do topo */
    return v;
}
```

Nestas duas funções, note o uso dos operadores de incremento e decremento em expressões. Na função `push`, incrementamos o número de elementos após armazenar o elemento (operador pós-fixado): o valor é armazenado no índice `n`, que depois é incrementado. Na função `pop`, decrementamos o número de elementos antes de acessar o vetor (operador prefixado); assim, acessamos o vetor com o valor de `n` já decrementado.

A função que verifica se a pilha está vazia pode ser dada por:

```
int pilha_vazia (Pilha* p)
{
    return (p->n == 0);
}
```

Por fim, a função para liberar a memória alocada pela pilha pode ser:

```
void pilha_libera (Pilha* p)
{
    free(p->vet);
    free(p);
}
```

### 15.1.3 Implementação de pilha com lista

Vamos agora considerar o uso de uma lista encadeada na implementação da estrutura pilha. Os elementos inseridos na pilha serão armazenados na lista. Esta implementação fica simples se considerarmos que o início da lista representa o topo da pilha: é fácil inserir e retirar elementos do início de uma lista encadeada.

A estrutura que representa o tipo pilha deve, portanto, ser a estrutura que representa uma lista encadeada:

```
typedef struct listano Listano;
struct listano {
    float info;
    Listano* prox;
};

struct pilha {
    Listano* prim;
};
```

A função `cria` aloca a estrutura da pilha e inicializa a lista como vazia.

```
Pilha* pilha_cria (void)
{
    Pilha* p = (Pilha*) malloc(sizeof(Pilha));
    p->prim = NULL;
    return p;
}
```

O primeiro elemento da lista representa o topo da pilha. Cada novo elemento é inserido no início da lista e, consequentemente, sempre que solicitado, retiramos o elemento também do início da lista. A implementação dessas funções é ilustrada a seguir. Novamente, assumimos que a função `pop` só é chamada para pilhas não vazias.

```
void pilha_push (Pilha* p, float v)
{
    ListaNo* n = (ListaNo*) malloc(sizeof(ListaNo));
    n->info = v;
    n->prox = p->prim;
    p->prim = n;
}

float pilha_pop (Pilha* p)
{
    ListaNo* t = p->prim;
    float v = t->info;
    p->prim = t->prox;
    free(t);
    return v;
}
```

A pilha estará vazia se a lista estiver vazia:

```
int pilha_vazia (Pilha* p)
{
    return (p->prim==NULL);
```

Por fim, a função que libera a pilha deve antes liberar todos os elementos da lista:

```
void pilha_libera (Pilha* p)
{
    ListaNo* q = p->prim;
    while (q!=NULL) {
        ListaNo* t = q->prox;
        free(q);
        q = t;
    }
    free(p);
}
```

A rigor, pela definição da estrutura de pilha, só temos acesso ao elemento do topo. No entanto, para testar o código, pode ser útil implementar uma função que imprime os valores armazenados na pilha. Os códigos a seguir ilustram a implementação dessa função nas duas versões de pilha (vetor e lista). A ordem de impressão adotada é do topo para a base.

```
/* imprime: versão com vetor */
void pilha_imprime (Pilha* p)
{
    for (int i=p->n-1; i>=0; i--)
        printf("%f\n",p->vet[i]);
}

/* imprime: versão com lista */
void pilha_imprime (Pilha* p)
{
    for (ListaNo* q=p->prim; q!=NULL; q=q->prox)
        printf("%f\n",q->info);
}
```

#### 15.1.4 Exemplo de uso: calculadora pós-fixada

Um bom exemplo de aplicação de pilha é o funcionamento das calculadoras RNP (sigla em inglês para Notação Polonesa Reversa). Elas trabalham com expressões pós-fixadas; então, para avaliar uma expressão como  $(1 - 2) * (4 + 5)$  podemos digitar 1 2 - 4 5 + \*. O funcionamento dessas calculadoras é muito simples. Cada operando é empilhado numa pilha de valores. Quando se encontra um operador, desempilha-se o número apropriado de operandos (dois para operadores binários e um para operadores unários), realiza-se a operação devida e empilha-se o resultado. Deste modo, na expressão em questão, são empilhados os valores 1 e 2. Quando aparece o operador  $-$ , 2 e 1 são desempilhados e o resultado da operação, no caso  $-1 (= 1 - 2)$ , é colocado no topo da pilha. A seguir, 4 e 5 são empilhados. O operador seguinte,  $+$ , desempilha o 5 e o 4 e empilha o resultado da soma, 9. Nesta hora, estão na pilha os dois resultados parciais,  $-1$  na base e 9 no topo. O operador  $*$ , então, desempilha os dois e coloca  $-9 (= -1 * 9)$  no topo da pilha.

Como exemplo de aplicação de uma estrutura de pilha, vamos implementar uma calculadora pós-fixada. Ela deve ter uma pilha de valores reais para representar os operandos. Para enriquecer a implementação, vamos considerar que o formato com que os valores da pilha são impressos seja um dado adicional associado à calculadora. Esse formato pode, por exemplo, ser passado quando da criação da calculadora.

Para representar a interface exportada pela calculadora, podemos criar o arquivo “calc.h”:

```
/* Arquivo que define a interface da calculadora */
typedef struct calc Calc;

/* funções exportadas */
Calc* calc_cria (char* f);
void calc_operando (Calc* c, float v);
void calc_operador (Calc* c, char op);
void calc_libera (Calc* c);
```

A implementação da calculadora faz uso do TAD pilha criado anteriormente e independe da implementação usada (vetor ou lista). O tipo que representa a calculadora pode ser dado por:

```
struct calc {
    char f[21];      /* formato para impressão */
    Pilha* p;        /* pilha de operandos */
};
```

A função `cria` recebe como parâmetro de entrada uma cadeia de caracteres com o formato que será utilizado pela calculadora para imprimir os valores. Essa função cria uma calculadora inicialmente sem operandos na pilha.

```
Calc* calc_cria (char* formato)
{
    Calc* c = (Calc*) malloc(sizeof(Calc));
    strcpy(c->f, formato);
    c->p = pilha_cria();           /* cria pilha vazia */
    return c;
}
```

A função `operando` coloca no topo da pilha o valor passado como parâmetro. A função `operador` retira os dois valores do topo da pilha (só consideraremos operadores binários), efetua a operação correspondente e coloca o resultado no topo da pilha. As operações válidas são: '+' para somar, '-' para subtrair, '\*' para multiplicar e '/' para dividir. Se não existirem operandos na pilha, consideraremos que seus valores são zero. Tanto a função `operando` quanto a função `operador` imprimem, utilizando o formato especificado na função `cria`, o novo valor do topo da pilha.

```
void calc_operando (Calc* c, float v)
{
    pilha_push(c->p, v);          /* empilha operando */
    printf(c->f, v);              /* imprime topo da pilha */
}

void calc_operador (Calc* c, char op)
{
    /* desempilha operandos */
    float v2 = pilha_vazia(c->p) ? 0.0f : pilha_pop(c->p);
    float v1 = pilha_vazia(c->p) ? 0.0f : pilha_pop(c->p);
    /* faz operação */
    float v;
    switch (op) {
        case '+': v = v1+v2; break;
        case '-': v = v1-v2; break;
        case '*': v = v1*v2; break;
        case '/': v = v1/v2; break;
    }
    pilha_push(c->p, v);          /* empilha resultado */
    printf(c->f, v);              /* imprime topo da pilha */
}
```

Por fim, a função para liberar a memória usada pela calculadora libera a pilha de operandos e a estrutura da calculadora.

```
void calc_libera (Calc* c)
{
    pilha_libera(c->p);
    free(c);
}
```

Um programa cliente que faz uso da calculadora é mostrado a seguir:

```
/* Programa para ler expressão e chamar funções da calculadora */
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"

int main (void)
{
    Calc* calc = calc_cria ("%.2f\n");           /* cria calculadora */
    while (1) {
        /* lê próximo caractere não branco */
        char c;
        scanf(" %c",&c);

        /* processa caractere lido */
        if (c=='+' || c=='-' || c=='*' || c=='/')
            calc_operador(calc,c);

        else if (c == 'q')                         /* abortou programa? */
            break;

        else {          /* devolve caractere lido e tenta ler número */
            float v;
            ungetc(c, stdin);
            if (scanf("%f",&v) == 1)
                calc_operando(calc,v);
            else {
                printf("Erro na leitura\n");
                exit(1);
            }
        }
    }
    calc_libera (calc);
    return 0;
}
```

Esse programa cliente lê os dados fornecidos pelo usuário e opera a calculadora. Para tanto, o programa lê um caractere não branco e verifica se é um operador válido. Em caso negativo, o programa “devolve” o caractere lido para o buffer de leitura padrão (`stdin`), através da função `ungetc`, e tenta ler um operando. O usuário finaliza a execução do programa digitando `q`.

Se executado, e considerando-se as expressões digitadas pelo usuário mostradas em negrito, esse programa teria como saída:

**3 5 8 \* +**      *digitado pelo usuário*

3.00

5.00

8.00

40.00

43.00

**7 /**      *digitado pelo usuário*

7.00

**6.14**      *digitado pelo usuário*

**q**      *digitado pelo usuário*

## 15.2 Filas

Outra estrutura de dados bastante usada em computação é a fila. A estrutura de fila é uma analogia natural com o conceito de fila que usamos no nosso dia a dia: quem primeiro entra numa fila é o primeiro a ser atendido (a sair da fila). A ideia fundamental da fila é que só podemos inserir um novo elemento no final da fila e só podemos retirar o elemento do início (regra *FIFO, first in, first out*).

Um exemplo de utilização em computação é a implementação de uma fila de impressão. Se uma impressora é compartilhada por várias máquinas, deve-se adotar uma estratégia para determinar que documento será impresso primeiro. A estratégia mais simples é tratar todas as requisições com a mesma prioridade e imprimir os documentos na ordem em que forem submetidos – o primeiro submetido é o primeiro a ser impresso.

De modo análogo ao que fizemos com a estrutura de pilha, discutiremos duas estratégias para a implementação de uma estrutura de fila: usando um vetor e usando uma lista encadeada. Para implementar uma fila, devemos ser capazes de inserir novos elementos em uma extremidade, o *fim*, e retirar elementos da outra extremidade, o *início*.

### 15.2.1 Interface do tipo fila

Antes de discutir as duas estratégias de implementação, podemos definir a interface disponibilizada pela estrutura, isto é, definir quais operações serão implementadas para manipular a fila. Mais uma vez, para simplificar a exposição, consideraremos uma estrutura que armazena valores reais. Independentemente da estratégia de implementação, a interface do tipo abstrato que representa uma estrutura de fila pode ser composta pelas seguintes operações:

- Criar uma fila vazia
- Inserir um elemento no fim
- Retirar o elemento do início
- Verificar se a fila está vazia
- Liberar a fila

O arquivo “fila.h”, que representa a interface do tipo, pode conter o seguinte código:

```
typedef struct fila Fila;
Fila* fila_cria (void);
void fila_insere (Fila* f, float v);
float fila_retira (Fila* f);
int fila_vazia (Fila* f);
void fila_libera (Fila* f);
```

A função `cria` aloca dinamicamente a estrutura da fila, inicializa seus campos e retorna seu ponteiro; a função `insere` adiciona um novo elemento no final da fila e a função `retira` remove o elemento do início; a função `vazia` informa se a fila está vazia ou não; e a função `libera` destrói a estrutura, liberando toda a memória alocada.

### 15.2.2 Implementação de fila com vetor

Assim como no caso da pilha, nossa primeira implementação de fila será feita usando um vetor para armazenar os elementos. Se o número máximo de elementos a serem armazenados na fila for conhecido, podemos usar um vetor estático. Neste exemplo, vamos considerar o uso de um vetor dinâmico, sem limitar o número máximo de elementos armazenados na estrutura.

Uma estratégia simplista para implementar a fila faria inserção de novos elementos no final do vetor e a retirada do início, deslocando os elementos do vetor para preencher o espaço vazio no início do vetor. No entanto, esta operação teria um custo computacional alto. É melhor evitar a necessidade de deslocar os elementos no vetor. Assim, o processo de inserção e remoção em extremidades opostas fará com que a fila “ande” no vetor. Por exemplo, se inserirmos os elementos 1.4, 2.2, 3.5, 4.0 e depois retirarmos dois elementos, o início da fila não estará mais na posição inicial do vetor. A Figura 15.2 ilustra a configuração da fila após a inserção dos primeiros quatro elementos e a Figura 15.3 após a remoção de dois elementos.

0	1	2	3	4	5	...
1.4	2.2	3.5	4.0			
↑				↑		
<i>ini</i>				<i>fim</i>		

Figura 15.2: Fila após inserção de quatro novos elementos.

0	1	2	3	4	5	...
1.4	2.2	3.5	4.0			
↑			↑			
<i>ini</i>			<i>fim</i>			

Figura 15.3: Fila após retirar dois elementos.

Com essa estratégia, é fácil observar que, em um dado instante, a parte ocupada do vetor pode chegar à última posição. Os elementos antes do índice `ini` não fazem mais parte da fila, e suas posições podem ser reaproveitadas para armazenar novos elementos. De fato, para aproveitar as primeiras posições livres do vetor sem implementar uma re-arrumação trabalhosa dos elementos, podemos incrementar as posições do vetor de forma “circular”: se o último elemento da fila ocupa a última posição do vetor, e existirem posições livres no início, inserimos os novos elementos a partir do início do vetor. Desta forma, em um dado momento, podemos ter quatro elementos, 20.0, 22.8, 21.2 e 24.3, distribuídos no vetor, com dois no fim do vetor e dois no início. A Figura 15.4 ilustra esse incremento circular.

0	1	2	$\dots$	$n - 2$	$n - 1$
21.2	24.3			20.0	22.8

$\uparrow$        $\uparrow$   
`fim`      `ini`

Figura 15.4: Fila com incremento circular.

Para essa implementação, os índices do vetor são incrementados de maneira que seus valores progridam “circularmente”. Este incremento circular, considerando `dim` a dimensão do vetor, poder ser dada por:

`i = (i == (dim-1)) ? 0 : i+1;`

ou, usando o operador módulo, simplesmente por:

`i = (i+1) % dim;`

Podemos declarar o tipo `fila` como sendo uma estrutura com três componentes: um vetor dinâmico `vet`, a dimensão corrente do vetor `dim`, o número de elementos armazenados na fila `n` e um índice `ini` para o início da fila.

Conforme ilustrado nas figuras, usamos as seguintes convenções para a identificação da fila:

- `ini` marca a posição do próximo elemento a ser retirado da fila.
- `fim` marca a posição (vazia) na qual será inserido o próximo elemento.

De posse do índice `ini` e do número de elementos, podemos calcular o índice `fim` incrementando `ini` de `n` unidades, também de forma circular:

`fim = (ini+n)%dim`

A estrutura de fila pode então ser dada por:

```
struct fila {
    int n;          /* número de elementos armazenados */
    int ini;        /* índice do início da fila */
    int dim;        /* dimensão corrente do vetor */
    float* vet;     /* vetor dos elementos */
};
```

A função para criar a fila aloca dinamicamente essa estrutura, escolhe uma dimensão inicial do vetor e inicializa a fila como sendo vazia.

```
Fila* fila_cria (void)
{
    Fila* f = (Fila*) malloc(sizeof(Fila));
    f->dim = 4; /* dimensão inicial */
    f->vet = (float*) malloc(f->dim*sizeof(float));
    f->n = 0; /* inicializa fila vazia */
    f->ini = 0; /* escolhe uma posição inicial */
    return f;
}
```

Para inserir um elemento na fila, usamos a próxima posição livre do vetor, indicada por `fim`. Devemos ainda assegurar que há espaço para a inserção do novo elemento; caso necessário, devemos reallocar o vetor. A realocação, no entanto, pode exigir uma rearrumação dos elementos. Se há necessidade de realocação, significa que o vetor está cheio. No caso especial de o índice `ini` ser zero, não precisamos rearrumar os elementos; qualquer valor de `ini` diferente de zero significa que os primeiros `n - ini` elementos da fila ocupam as últimas posições do vetor. Estes valores precisam ser deslocados para o final do novo vetor. Para fazer este deslocamento, usaremos a função `memmove` definida na biblioteca `string.h`. Esta função recebe três parâmetros: o endereço de memória destino da cópia, o endereço de memória origem e o número de bytes a ser movido (copiado). Note que, para a função `memmove`, o espaço de memória destino pode ter sobreposição com o espaço origem.

```
void fila_insere (Fila* f, float v)
{
    int fim;
    if (f->n == f->dim) { /* capacidade esgotada */
        /* realoca, dobrando o tamanho */
        f->dim *= 2;
        f->vet = (float*) realloc(f->vet, f->dim*sizeof(float));
        if (f->ini != 0)
            memmove(&f->vet[f->dim-f->ini], /* endereço destino */
                    &f->vet[f->ini], /* endereço origem */
                    (f->n-f->ini)*sizeof(float) /* número de bytes */);
    }
    /* insere elemento na próxima posição livre */
    fim = (f->ini + f->n) % f->dim;
    f->vet[fim] = v;
    f->n++;
}
```

A função para retirar o elemento do início da fila fornece o valor do elemento retirado como retorno. Vamos assumir que esta função não é chamada com a fila vazia.

```

float fila_retira (Fila* f)
{
    float v;
    v = f->vet[f->ini]; /* retira elemento do início */
    f->ini = (f->ini + 1) % f->dim;
    f->n--;
    return v;
}

```

A função que verifica se a fila está vazia pode ser dada por:

```

int fila_vazia (Fila* f)
{
    return (f->n == 0);
}

```

Finalmente, a função para liberar a memória alocada pela fila pode ser:

```

void fila_libera (Fila* f)
{
    free(f->vet);
    free(f);
}

```

### 15.2.3 Implementação de fila com lista

Vamos agora ver como implementar uma fila através de uma lista encadeada, que será, como nos exemplos anteriores, uma lista simplesmente encadeada, em que cada nó guarda um ponteiro para o próximo nó da lista. Como teremos que inserir e retirar elementos das extremidades opostas da lista, que representarão o início e o fim da fila, teremos que usar dois ponteiros, `ini` e `fim`, que apontam respectivamente para o primeiro e para o último elemento da fila. Essa situação é ilustrada na Figura 15.5:

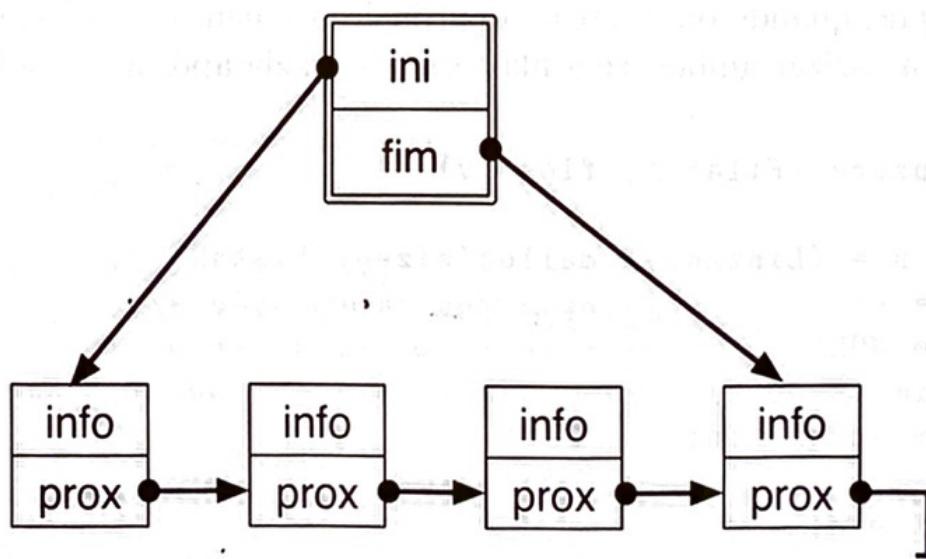


Figura 15.5: Estrutura de fila com lista encadeada.

A operação para retirar um elemento se dá no início da lista (fila) e consiste essencialmente em fazer com que, após a remoção, `ini` aponte para o sucessor do nó retirado. (Observe que seria mais complicado remover um nó do fim da lista simplesmente encadeada, porque o antecessor de um nó não é encontrado com a mesma facilidade que seu sucessor.) A inserção também é simples, pois basta acrescentar à lista um sucessor para o último nó, apontado por `fim`, e fazer com que `fim` aponte para este novo nó.

A estrutura da fila agrupa os ponteiros para o início e o fim da lista:

```
typedef struct listano Listano;
struct listano {
    float info;
    Listano* prox;
};

struct fila {
    Listano* ini;
    Listano* fim;
};
```

A função `cria` aloca a estrutura da fila e inicializa a lista como sendo vazia.

```
Fila* fila_cria (void)
{
    Fila* f = (Fila*) malloc(sizeof(Fila));
    f->ini = f->fim = NULL;
    return f;
}
```

Cada novo elemento é inserido no fim da lista e, sempre que solicitado, retiramos o elemento do início da lista. Desta forma, precisamos de dois procedimentos: para inserir no fim e para remover do início. O procedimento para inserir no fim ainda não foi discutido, mas é simples, uma vez que temos explicitamente armazenado o ponteiro para o último elemento. Devemos salientar que a função de inserção deve atualizar ambos os ponteiros, `ini` e `fim`, quando da inserção do primeiro elemento. Analogamente, a função para retirar deve atualizar ambos, se a fila tornar-se vazia após a remoção do elemento:

```
void fila_insere (Fila* f, float v)
{
    Listano* n = (Listano*) malloc(sizeof(Listano));
    n->info = v; /* armazena informação */
    n->prox = NULL; /* novo nó passa a ser o último */
    if (f->fim != NULL) /* verifica se lista não estava vazia */
        f->fim->prox = n;
    else /* fila estava vazia */
        f->ini = n;
    f->fim = n; /* fila aponta para novo elemento */
}
```

```

float fila_retira (Fila* f)
{
    ListaNo* t = f->ini;
    float v = t->info;
    f->ini = t->prox;
    if (f->ini == NULL) /* verifica se fila ficou vazia */
        f->fim = NULL;
    free(t);
    return v;
}

```

A fila estará vazia se a lista estiver vazia:

```

int fila_vazia (Fila* f)
{
    return (f->ini==NULL);
}

```

Por fim, a função que libera a fila deve antes liberar todos os elementos da lista.

```

void fila_libera (Fila* f)
{
    ListaNo* q = f->ini;
    while (q!=NULL) {
        ListaNo* t = q->prox;
        free(q);
        q = t;
    }
    free(f);
}

```

Analogamente à pilha, para testar o código pode ser útil implementar uma função que imprime os valores armazenados na fila. Os códigos a seguir ilustram a implementação dessa função nas duas versões de fila (vetor e lista). A ordem de impressão adotada é do início para o fim.

```

/* imprime: versão com vetor */
void fila_imprime (Fila* f)
{
    for (int i=0; i<f->n; i++)
        printf("%f\n", f->vet[(f->ini+i)%f->dim]);
}

/* imprime: versão com lista */
void fila_imprime (Fila* f)
{
    for (ListaNo* q=f->ini; q!=NULL; q=q->prox)
        printf("%f\n", q->info);
}

```

Um exemplo simples de utilização da estrutura de fila é apresentado a seguir:

```
/* Módulo para ilustrar utilização da fila */
#include <stdio.h>
#include "fila.h"

int main (void)
{
    Fila* f = fila_cria();
    fila_insere(f,20.0);
    fila_insere(f,20.8);
    fila_insere(f,21.2);
    fila_insere(f,24.3);
    printf("Primeiro elemento: %f\n", fila_retira(f));
    printf("Segundo elemento: %f\n", fila_retira(f));
    printf("Configuracao da fila:\n");
    fila_imprime(f);
    fila_libera(f);
    return 0;
}
```

### 15.3 Fila dupla

A estrutura de dados que chamamos de *fila dupla* consiste numa fila na qual é possível inserir novos elementos em ambas as extremidades, no início e no fim. Consequentemente, permite-se também retirar elementos de ambos os extremos. É como se, dentro de uma mesma estrutura de fila, tivéssemos duas filas, com os elementos dispostos em ordem inversa uma da outra.

A interface do tipo abstrato que representa uma fila dupla acrescenta novas funções para inserir e retirar elementos. Podemos enumerar as seguintes operações:

- Criar uma estrutura de fila dupla
- Inserir um elemento no início
- Inserir um elemento no fim
- Retirar o elemento do início
- Retirar o elemento do fim
- Verificar se a fila está vazia
- Liberar a fila

O arquivo “fila2.h”, que representa a interface do tipo, pode conter o seguinte código:

```
typedef struct fila2 Fila2;

Fila2* fila2_cria (void);
void fila2_insere_ini (Fila2* f, float v);
void fila2_insere_fim (Fila2* f, float v);
float fila2_retira_ini (Fila2* f);
float fila2_retira_fim (Fila2* f);
int fila2_vazia (Fila2* f);
void fila2_libera (Fila2* f);
```

### 15.3.1 Fila dupla com vetor

A implementação dessa estrutura usando um vetor para armazenar os elementos não traz grandes dificuldades, pois o vetor permite acesso randômico aos elementos. Vamos analisar as duas novas funções: `insere_ini` e `retira_fim`. Para inserir no início, devemos inserir o elemento no índice que precede `ini`, adotando um decreimento circular. O índice precedente pode ser obtido fazendo:

```
i = --i < 0 ? dim-1 : i;
```

onde `dim` representa a dimensão do vetor. Esse decremento circular também pode ser feito usando o operador módulo:

```
i = (i-1+dim)%dim;
```

Desta forma, a função para inserir no início pode ser dada por:

```
void fila2_insere_ini (Fila2* f, float v)
{
    if (f->n == f->dim) { /* capacidade esgotada */
        f->dim *= 2;
        f->vet = (float*) realloc(f->vet, f->dim*sizeof(float));
        if (f->ini != 0)
            memmove(&f->vet[f->dim-f->ini],
                     &f->vet[f->ini],
                     (f->n-f->ini)*sizeof(float));
    }
    /* decrementa (circularmente) índice de início */
    f->ini = (f->ini - 1 + f->dim) % f->dim;
    f->vet[f->ini] = v;
    f->n++;
}
```

Para retirar do final da fila, devemos acessar o último elemento armazenado na fila. De posse de `ini` e `n`, como vimos, o índice do último elemento é dado por:  $(\text{ini} + n - 1) \% \text{dim}$ . Assim, uma possível implementação da função que retira do final pode ser:

```
float fila2_retira_fim (Fila2* f)
{
    int ult = (f->ini + f->n - 1) % f->dim; /* índice do último */
    float v = f->vet[ult];
    f->n--;
    return v;
}
```

### 15.3.2 Implementação de fila dupla com lista

A implementação de uma fila dupla com lista encadeada merece uma discussão mais detalhada. A dificuldade que encontramos reside na implementação da função para retirar um elemento do final da lista. Todas as outras funções, incluindo inserir no início, poderiam ser facilmente implementadas usando uma lista simplesmente encadeada. No entanto, na lista simplesmente encadeada, a função para retirar do fim não pode ser implementada de forma eficiente, pois, dado o ponteiro para o último elemento da lista, não temos como acessar o anterior, que passaria a ser o último elemento.

Para solucionar esse problema, temos que lançar mão da estrutura de lista duplamente encadeada (ver Seção 14.3). A lista duplamente encadeada resolve o problema de acessar o elemento anterior ao último. Devemos salientar que o uso de uma lista duplamente encadeada para implementar a fila é simples, pois só manipulamos os elementos das extremidades da lista.

O arranjo de memória para implementar a fila dupla é idêntico ao ilustrado para lista dupla (Figura 14.8). A estrutura da fila dupla agrupa os ponteiros para o início e o fim da lista (chamados aqui `ini` e `fim`, para manter a nomenclatura que temos usado para filas):

```
typedef struct listano2 ListaNo2;
struct listano2 {
    float info;
    ListaNo2* ant;
    ListaNo2* prox;
};
struct fila2 {
    ListaNo2* ini;
    ListaNo2* fim;
};
```

Interessa-nos apresentar as funções para retirar elementos. As funções para inserir já foram apresentadas quando introduzimos o conceito de listas duplamente encadeadas. As funções que retiram elementos da fila podem ser dadas por (assumindo que a fila não está vazia):

```

float fila2_retira_ini (Fila2* f)
{
    ListaNo2* t = f->ini;
    float v = t->info;
    f->ini = t->prox;
    if (f->ini == NULL) /* verifica se fila ficou vazia */
        f->fim = NULL;
    else
        f->ini->ant = NULL;
    free(t);
    return v;
}

float fila2_retira_fim (Fila2* f)
{
    ListaNo2* t = f->fim;
    float v = t->info;
    f->fim = t->ant;
    if (f->fim == NULL) /* verifica se fila ficou vazia */
        f->ini = NULL;
    else
        f->fim->prox = NULL;
    free(t);
    return v;
}

```

Por fim, lembramos que a implementação de tipos abstratos pode ser feita utilizando tipos abstratos já existentes. Nesse sentido, se temos os tipos abstratos que representam listas, podemos construir os tipos abstratos de pilha e fila usando os tipos de lista. Fica como exercício reescrever os tipos de pilha e fila usando TADs de listas.

## Exercícios

1. Considere a existência de um tipo abstrato **Pilha** de números reais, cuja interface está definida no arquivo “pilha.h” da seguinte forma:

```

typedef struct pilha Pilha;
Pilha* pilha_cria(void);
void pilha_push (Pilha* p, float v);
float pilha_pop (Pilha* p);
int pilha_vazia (Pilha* p);
void pilha_libera (Pilha* p);

```

Sem conhecer a representação interna desse tipo abstrato e usando apenas as funções declaradas no arquivo de interface, pede-se:

- (a) Implemente uma função que receba uma pilha como parâmetro e retorne o valor armazenado em seu topo, restaurando o conteúdo da pilha. Essa função deve obedecer ao protótipo:

```
float topo (Pilha* p);
```

- (b) Implemente uma função que receba duas pilhas,  $p1$  e  $p2$ , e passe todos os elementos da pilha  $p2$  para o topo da pilha  $p1$ . A Figura 15.6 ilustra essa concatenação de pilhas. Note que, ao final dessa função, a pilha  $p2$  vai estar vazia e a pilha  $p1$  conterá todos os elementos das duas pilhas.

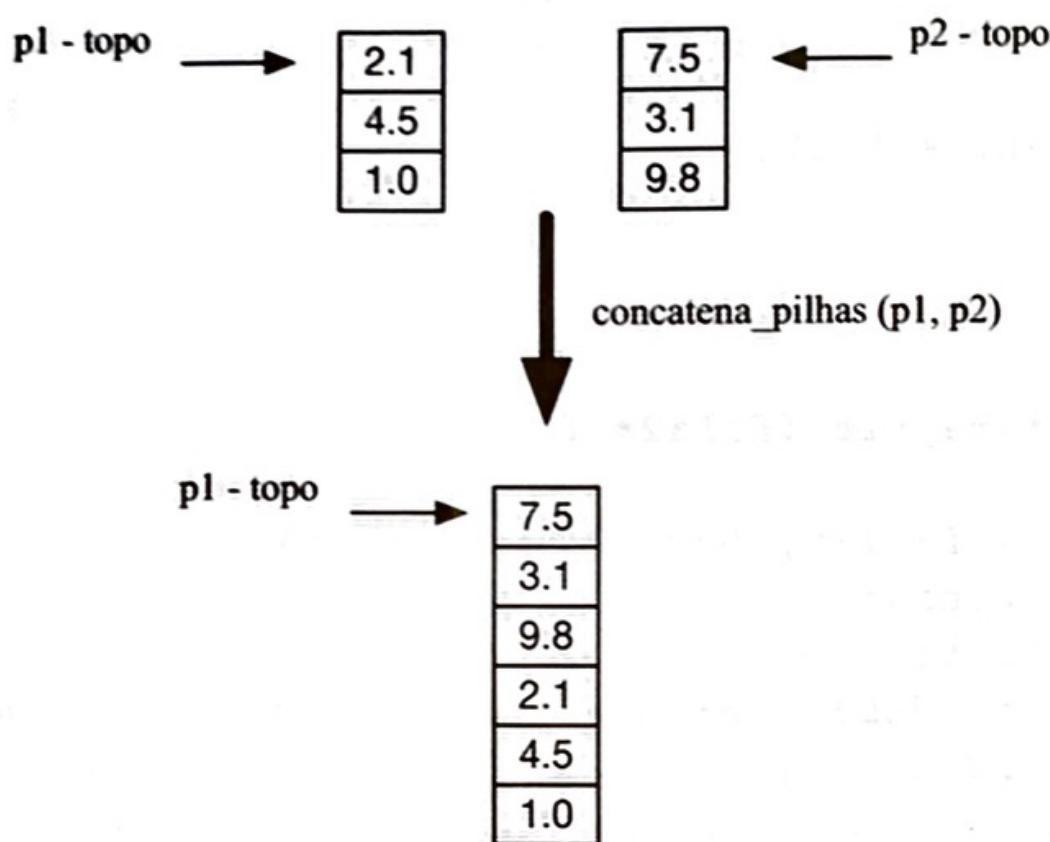


Figura 15.6: Operação de concatenação de duas pilhas.

Essa função deve obedecer ao protótipo:

```
void concatena_pilhas (Pilha* p1, Pilha* p2);
```

Implemente esta função de duas formas distintas: (i) usando uma função recursiva; (ii) usando uma terceira pilha auxiliar.

- (c) Implemente uma função que receba uma pilha como parâmetro e retorne como resultado uma cópia dessa pilha. Essa função deve obedecer ao protótipo:

```
Pilha* copia_pilha (Pilha* p);
```

Ao final da função `copia_pilha`, a pilha  $p$  recebida como parâmetro deve ter seu conteúdo original. Implemente duas versões desta função: usando recursão e usando uma pilha auxiliar.

2. Considere a existência de um tipo abstrato `Fila` de números reais, cuja interface está definida no arquivo “fila.h” da seguinte forma:

```

typedef struct fila Fila;
Fila* fila_cria(void);
void fila_insere (Fila* f, float v);
float fila_retira (Fila* f);
int fila_vazia (Fila* f);
void fila_libera (Fila* f);
  
```

Sem conhecer a representação interna desse tipo abstrato e usando apenas as funções declaradas no arquivo de interface, implemente uma função que receba três filas, `f_res`, `f1` e `f2`, e transfira alternadamente os elementos de `f1` e `f2` para `f_res`, conforme ilustrado na Figura 15.7.

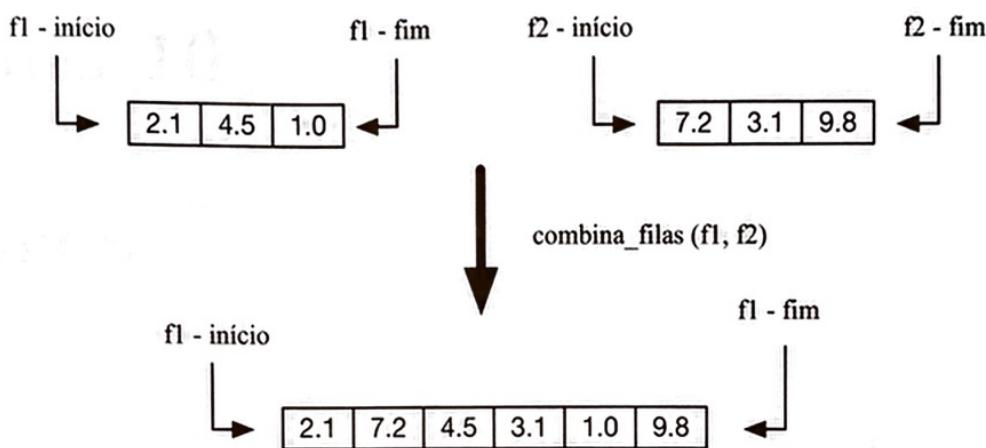


Figura 15.7: Operação de combinação de duas listas numa terceira.

Note que, ao final dessa função, as filas `f1` e `f2` vão estar vazias e a fila `f_res` vai conter todos os valores que estavam originalmente em `f1` e `f2` (inicialmente `f_res` pode ou não estar vazia). Se uma fila for maior que a outra, os valores excedentes devem ser transferidos para a nova fila no final. Essa função deve obedecer ao protótipo:

```
void combina_filas (Fila* f_res, Fila* f1, Fila* f2);
```

3. Estenda a funcionalidade da calculadora pós-fixada, que usa uma pilha de valores reais, incluindo novos operadores unários e binários (sugestão: `-` como menos unário, `#` como raiz quadrada, `^` como exponenciação).
4. Implemente uma calculadora pós-fixada para operar sobre números complexos, fornecidos no formato `(a, b)`.