Algoritmos de ordenação básicos

Os cientistas da computação desenvolveram muitas estratégias engenhosas para ordenar uma lista de itens. Várias delas são discutidas aqui. Os algoritmos examinados nesta seção são fáceis de escrever, mas, mais ineficientes; os algoritmos discutidos na próxima seção são mais difíceis

de escrever, mas, mais eficientes. (Isso é um equilíbrio comum.) Cada uma das funções de ordenação do Python desenvolvidas opera em uma lista de inteiros e usa uma função `swap` para trocar as posições de dois itens da lista. Eis o código dessa função:

```
def swap(lyst, i, j):
    """Troca os itens nas posições i e j."""
    # Você poderia dizer lyst[i], lyst[j] = lyst[j], lyst[i]
    # mas o código a seguir mostra o que realmente está acontecendo
    temp = lyst[i]
    lyst[i] = lyst[j]
    lyst[j] = temp
```

Ordenação por seleção

Talvez a estratégia mais simples seja pesquisar em toda a lista a posição do menor item. Se essa posição não é igual à primeira posição, o algoritmo permuta os itens nessas posições. O algoritmo então retorna à segunda posição e repete o processo, trocando o menor item pelo item na segunda posição se necessário. Quando o algoritmo alcança a última posição no processo geral, a lista está ordenada. O algoritmo chama-se **ordenação por seleção** porque cada passagem pelo laço principal seleciona um único item a ser movido. A Figura 3-8 mostra os estados de uma lista de cinco itens após cada pesquisa e passagem de troca de uma ordenação por seleção. Os dois itens recém-trocados em cada passagem têm asteriscos ao lado deles e a parte ordenada da lista é sombreada.

Lista não ordenada	Após primeira passagem	Após segunda passagem	Após terceira passagem	Após quarta passagem
5	1*	1	1	1
3	3	2*	2	2
1	5*	5	3*	3
2	2	3*	5*	4*
4	4	4	4	5*

Figura 3-8 Um traço dos dados durante uma ordenação por seleção

Eis a função Python para uma ordenação por seleção:

```
def selectionSort(lyst):
    i = 0
    while i < len(lyst) - 1:          # Faz n - 1 pesquisas
        minIndex = i                  # pelo menor
        j = i + 1
        while j < len(lyst):          # Inicia uma pesquisa
            if lyst[j] < lyst[minIndex]:
                minIndex = j
            j += 1
        if minIndex != i:             # Troca se necessário
            swap(lyst, minIndex, i)
        i += 1
```

Essa função inclui um laço aninhado. Para uma lista de tamanho *n*, o laço externo executa $n - 1$ vez. Na primeira passagem pelo laço externo, o laço interno executa $n - 1$ vez. Na segunda passagem pelo laço externo, o laço interno executa $n - 2$ vezes. Na última passagem pelo laço externo, o laço interno é executado uma vez. Assim, o número total de comparações para uma lista de tamanho *n* é o seguinte:

$$(n-1)+(n-2)+\ \ldots\ +1=$$
$$n(n-1)/2=$$
$$\tfrac{1}{2}n^2-\tfrac{1}{2}n$$

Para *n* grande, você pode escolher o termo com o maior grau e descartar o coeficiente, assim a ordenação por seleção é $O(n^2)$ em todos os casos. Para grandes conjuntos de dados, o custo de trocar itens também pode ser significativo. Como os itens de dados são trocados apenas no laço externo, esse custo adicional da ordenação por seleção é linear nos casos pior e médio possíveis.

Ordenação por bolha

Outro algoritmo de ordenação relativamente fácil de conceber e codificar chama-se ordenação por bolha. A estratégia dele é começar no início da lista e comparar pares de itens de dados à medida que alcança o final. Sempre que os itens no par estão fora de ordem, o algoritmo permuta-os. Esse processo tem o efeito de borbulhar os maiores itens no final da lista. O algoritmo então repete o processo do início da lista e passa para o penúltimo item etc., até começar com o último item. Nesse ponto, a lista está ordenada.

A Figura 3-9 mostra um traço do processo de bolha por meio de uma lista de cinco itens. Esse processo cria quatro passagens por meio de um laço aninhado para inserir o maior item no final da lista. Mais uma vez, os itens que acabaram de ser trocados são marcados com asteriscos e a parte ordenada é sombreada.

Lista não ordenada	Após primeira passagem	Após segunda passagem	Após terceira passagem	Após quarta passagem
5	4*	4	4	4
4	5*	2*	2	2
2	2	5*	1*	1
1	1	1	5*	3*
3	3	3	3	5*

Figura 3-9 Um traço dos dados durante uma ordenação por bolha

Eis a função Python para uma ordenação por bolha:

```
def bubbleSort(lyst):
    n = len(lyst)
    while n > 1:                        # Faz n - 1 bolha
        i = 1                           # Inicia cada bolha
        while i < n:
            if lyst[i] < lyst[i - 1]:   # Troca se necessário
```

```
                swap(lyst, i, i - 1)
            i += 1
        n -= 1
```

Assim como acontece com a ordenação por seleção, uma ordenação por bolha tem um laço aninhado. A parte ordenada da lista agora cresce do final da lista ao início, mas o desempenho da ordenação por bolha é bastante semelhante ao comportamento de uma ordenação por seleção: o laço interno é executado $\frac{1}{2}n^2 - \frac{1}{2}n$ vezes para uma lista de tamanho n. Assim, a ordenação por bolha é $O(n^2)$. Como a ordenação por seleção, a ordenação por bolha não realizará nenhuma troca se a lista já estiver ordenada. No entanto, comportamento no pior caso possível da ordenação por bolha para trocas é maior do que linear. A prova disso é deixada como um exercício para você.

Você pode fazer um pequeno ajuste na ordenação por bolha para melhorar o desempenho no melhor caso para linear. Se nenhuma troca ocorrer durante uma passagem pelo laço principal, a lista será ordenada. Isso pode acontecer em qualquer passagem e, na melhor das hipóteses, acontecerá na primeira passagem. Você pode rastrear a presença da troca com um flag booleano e retornar da função quando o laço interno não definir esse flag. Eis a função de ordenação por bolha modificada:

```
def bubbleSortWithTweak(lyst):
    n = len(lyst)
    while n > 1:
        swapped = False
        i = 1
        while i < n:
            if lyst[i] < lyst[i - 1]:    # Troca se necessário
                swap(lyst, i, i - 1)
                swapped = True
            i += 1
        if not swapped: return           # Retorna se não houver trocas
        n -= 1
```

Observe que essa modificação apenas melhora o comportamento no melhor caso possível. Em média, o comportamento dessa versão da ordenação por bolha ainda é $O(n^2)$.

Ordenação por inserção

Nossa ordenação por bolha modificada tem um desempenho melhor do que uma ordenação por seleção para listas que já estão ordenadas. Mas nossa ordenação por bolha modificada ainda pode ter um desempenho ruim se muitos itens estiverem fora de ordem na lista. Outro algoritmo, denominado ordenação por inserção, tenta explorar a ordenação parcial da lista de maneira diferente. A estratégia é como a seguir:

- Na i-ésima passagem pela lista, onde i varia de 1 a $n - 1$, o i-ésimo item deve ser inserido em seu devido lugar entre os primeiros itens i na lista.
- Depois de i-ésima passagem, os primeiros itens i devem estar em ordem.

- Esse processo é análogo à maneira como muitas pessoas organizam cartas de baralho nas mãos. Ou seja, se você segurar as primeiras $i - 1$ cartas em ordem, você escolhe a i-ésima carta e a compara com essas cartas até que seu lugar apropriado seja encontrado.
- Assim como acontece com nossos outros algoritmos de ordenação, a ordenação por inserção consiste em dois laços. O laço externo percorre as posições de 1 a $n - 1$. Para cada posição i nesse laço, você salva o item e inicia o laço interno na posição $i - 1$. Para cada posição j nesse laço, você move o item para a posição $j + 1$ até encontrar o ponto de inserção para o (i-ésimo) item salvo.

Eis o código para a função **`insertionSort`**:

```
def insertionSort(lyst):
    i = 1
    while i < len(lyst):
        itemToInsert = lyst[i]
        j = i - 1
        while j >= 0:
            if itemToInsert < lyst[j]:
                lyst[j + 1] = lyst[j]
                j -= 1
            else:
                break
        lyst[j + 1] = itemToInsert
        i += 1
```

A Figura 3-10 mostra os estados de uma lista de cinco itens após cada passagem pelo laço externo de uma ordenação por inserção. O item a ser inserido na próxima passagem é marcado com uma seta; depois de inserido, esse item é marcado com um asterisco.

Mais uma vez, a análise focaliza o laço aninhado. O laço externo executa $n - 1$ vez. No pior caso, quando todos os dados estão fora de ordem, o laço interno itera uma vez na primeira passagem pelo laço externo, duas vezes na segunda passagem e assim por diante, para um total de $\frac{1}{2}n^2 - \frac{1}{2}n$ vezes. Assim, o comportamento no pior caso possível da ordenação por inserção é $O(n^2)$.

Quanto mais itens na lista estão em ordem, melhor será a ordenação por inserção até que, no melhor caso de uma lista ordenada, o comportamento da ordenação seja linear. No caso médio, porém, a ordenação por inserção ainda é quadrática.

Lista não ordenada	Após primeira passagem	Após segunda passagem	Após terceira passagem	Após quarta passagem
2	2	1*	1	1
5 ←	5 (sem inserções)	2	2	2
1	1 ←	5	4*	3*
4	4	4 ←	5	4
3	3	3	3 ←	5

Figura 3-10 Um traço dos dados durante uma ordenação por inserção

Desempenho no melhor caso, pior caso e caso médio revisitado

Como mencionado anteriormente, para muitos algoritmos, você não pode aplicar uma única medida de complexidade a todos os casos. Às vezes, o comportamento de um algoritmo melhora ou piora quando ele encontra determinado arranjo de dados. Por exemplo, o algoritmo da ordenação por bolha pode terminar assim que a lista é ordenada. Se a lista de entrada já está ordenada, a ordenação por bolha requer aproximadamente n comparações. Mas em muitos outros casos, a ordenação por bolha requer em torno de n^2 comparações. Claramente, você pode precisar de uma análise mais detalhada para tornar os programadores cientes desses casos especiais.

Como discutido, uma análise completa da complexidade de um algoritmo divide seu comportamento em três tipos de casos:

- **Melhor caso** — Em que circunstâncias um algoritmo faz a menor quantidade de trabalho? Qual é a complexidade do algoritmo nesse melhor caso?
- **Pior caso** — Em que circunstâncias um algoritmo faz a maior parte do trabalho? Qual é a complexidade do algoritmo nesse pior caso?
- **Caso médio** — Em que circunstâncias um algoritmo realiza uma quantidade normal de trabalho? Qual é a complexidade do algoritmo nesse caso típico?

Agora, você revisará três exemplos desse tipo de análise para uma pesquisa por um mínimo, pesquisa sequencial e ordenação por bolha.

Como a pesquisa por um algoritmo mínimo deve visitar cada número na lista, a menos que esteja ordenada, o algoritmo sempre é linear. Portanto, os desempenhos nos casos melhor, pior e médio são $O(n)$.

A pesquisa sequencial é um pouco diferente. O algoritmo para e retorna um resultado assim que encontra o item-alvo. Claramente, na melhor das hipóteses, o elemento-alvo está na primeira posição. No pior dos casos, o alvo está na última posição. Portanto, o desempenho no melhor caso do algoritmo é $O(1)$ e o desempenho no pior é $O(n)$. Para calcular o desempenho no caso médio, você adiciona todas as comparações que devem ser feitas para localizar um alvo em cada posição e divide por n. Ou seja, $(1 + 2 + \ldots + n)/n$, ou $(n + 1)/2$. Portanto, por aproximação, o desempenho em caso médio possível da pesquisa sequencial também é $O(n)$.

A versão mais inteligente da ordenação por bolha pode terminar assim que a lista é ordenada. Na melhor das hipóteses, isso acontece quando a lista de entrada já está ordenada. Portanto, o desempenho em melhor caso da ordenação por bolha é $O(n)$. No entanto, esse caso é raro (1 de $n!$). Na pior das hipóteses, mesmo essa versão de ordenação por bolha tem de colocar cada item na posição adequada na lista. O pior caso de desempenho do algoritmo é claramente $O(n^2)$. O desempenho médio do tipo de bolha está mais próximo de $O(n^2)$ do que de $O(n)$, embora a demonstração desse fato seja um pouco mais complicada do que para a pesquisa sequencial.

Como veremos, existem algoritmos cujos desempenhos em caso médio e melhor caso são semelhantes, mas cujo desempenho pode degradar para pior caso. Se você está escolhendo um algoritmo ou desenvolvendo um novo, é importante estar ciente dessas distinções.

Exercícios

1. Qual configuração dos dados em uma lista resulta no menor número de trocas em uma ordenação por seleção? Qual configuração dos dados resulta no maior número de trocas?
2. Explique o papel que o número de trocas de dados desempenha na análise da ordenação por seleção e ordenação por bolha. Qual papel, se houver algum, o tamanho dos objetos de dados desempenha?
3. Explique por que a ordenação por bolha modificada ainda exibe comportamento $O(n^2)$ na média.
4. Explique por que a ordenação por inserção funciona bem em listas parcialmente ordenadas.

Ordenação mais rápida

Os três algoritmos de ordenação considerados até agora têm $O(n^2)$ tempos de execução. Existem muitas variações desses algoritmos de ordenação, alguns dos quais são ligeiramente mais rápidos, mas também são $O(n^2)$ nos piores casos e nos casos médios. No entanto, você pode tirar proveito de alguns algoritmos melhores que são $O(n \log n)$. O segredo desses algoritmos melhores é uma estratégia de dividir e conquistar. Isto é, cada algoritmo encontra uma maneira de dividir a lista em sublistas menores. Essas sublistas são então ordenadas recursivamente. Idealmente, se o número dessas subdivisões for $\log(n)$ e a quantidade de trabalho necessária para reorganizar os dados em cada subdivisão for n, então a complexidade total de tal algoritmo de ordenação será $O(n \log n)$. Na Tabela 3-3, você pode ver que a taxa de crescimento do trabalho de um algoritmo $O(n \log n)$ é muito mais lenta do que a de um algoritmo $O(n^2)$.

Esta seção examina dois algoritmos recursivos que dividem o barreira n^2 — quicksort e ordenação por mesclagem.

N	*n* log *n*	n^2
512	4608	262.144
1024	10.240	1.048.576
2048	22.458	4.194.304
8192	106.496	67.108.864
16.384	229.376	268.435.456
32.768	491.520	1.073.741.824

Tabela 3-3 Comparando $n \log n$ e n^2

Visão geral do quicksort

Eis um esboço da estratégia usada no algoritmo **quicksort**:

1. Comece selecionando o item no ponto médio da lista. Esse item chama-se **pivô**. (Mais adiante, este capítulo aborda maneiras alternativas de escolher o pivô.)
2. Particione os itens na lista de tal forma que todos os itens menores que o pivô sejam movidos para a esquerda do pivô e o restante seja movido para a direita. A posição final do próprio pivô varia, dependendo dos itens reais envolvidos. Por exemplo, o pivô acaba ficando mais à direita na lista se for o maior item e mais à esquerda se for o menor. Mas onde quer que o pivô termine, essa é a posição final na lista totalmente ordenada.
3. Divida e conquiste. Reaplique o processo recursivamente às sublistas formadas pela divisão da lista no pivô. Uma sublista consiste em todos os itens à esquerda do pivô (agora as menores) e a outra sublista tem todos os itens à direita (agora os maiores).
4. O processo termina sempre que encontra uma sublista com menos de dois itens.

Particionamento

Da perspectiva do programador, a parte mais complicada do algoritmo é a operação de particionar os itens em uma sublista. Existem duas maneiras principais de fazer isso. Informalmente, o que se segue é uma descrição do método mais fácil aplicado a qualquer sublista:

1. Troque o pivô pelo último item na sublista.
2. Estabeleça um limite entre os itens conhecidos por serem menores que o pivô e o restante dos itens. Inicialmente, esse limite é posicionado imediatamente antes do primeiro item.
3. Começando com o primeiro item na sublista após o limite, percorrer a sublista. Sempre que você encontrar um item menor que o pivô, troque-o pelo primeiro item após o limite e avance o limite.
4. Fechar trocando o pivô pelo primeiro item após o limite.

A Figura 3-11 ilustra as etapas aplicadas aos números **12 19 17 18 14 11 15 13 16**. Na Etapa 1, o pivô é estabelecido e trocado pelo último item. Na Etapa 2, o limite é estabelecido antes do primeiro item. Nos Passos 3–12, a sublista é varrida para itens menores que o pivô, eles são trocados pelo primeiro item após o limite, e o limite é avançado. Observe que os itens à esquerda do limite são sempre menores que o pivô. Por fim, na etapa 13, o pivô é trocado pelo primeiro item após o limite, e a sublista foi particionada de modo bem-sucedido.

Depois de ter particionado uma sublista, reaplique o processo às suas sublistas esquerda e direita (**12 11 13** e **16 19 15 17 18**) e assim por diante, até que as sublistas tenham comprimentos de no máximo um. A Figura 3-12 mostra um traço dos segmentos da lista antes de cada etapa de partição e o item pivô selecionado em cada etapa.

Passo	Ação	Estado da lista após a ação
	Faça a sublista conter os números mostrados com 14 como pivô.	12 19 17 18 14 11 15 13 16
1	Troque o pivô e o último item.	12 19 17 18 16 11 15 13 14
2	Estabeleça o limiar antes do primeiro item.	: 12 19 17 18 16 11 15 13 14
3	Busque o primeiro item logo abaixo do pivô.	: 12 19 17 18 16 11 15 13 14
4	Troque este item e o primeiro item além do limiar. Neste exemplo, o item é trocado por ele mesmo.	: 12 19 17 18 16 11 15 13 14
5	Avance o limiar.	12 : 19 17 18 16 11 15 13 14
6	Busque o próximo item logo abaixo do pivô.	12 : 19 17 18 16 11 15 13 14
7	Troque este item e o primeiro item acima do limiar.	12 : 11 17 18 16 19 15 13 14
8	Avance o limiar.	12 11 : 17 18 16 19 15 13 14
9	Busque o próximo item logo abaixo do pivô.	12 11 : 17 18 16 19 15 13 14
10	Troque este item e o primeiro item acima do limiar.	12 11 : 13 18 16 19 15 17 14
11	Avance o limiar.	12 11 13 : 18 16 19 15 17 14
12	Busque o próximo item logo abaixo do pivô; observe que não existe nenhum.	12 11 13 : 18 16 19 15 17 14
13	Troque o pivô com o primeiro item além do limiar. Neste ponto, todos os items abaixo do pivô estão à esquerda do pivô e os demais estão à direita.	12 11 13 : 14 16 19 15 17 18

Figura 3-11 Particionando uma sublista

Análise de complexidade do Quicksort

Agora veremos uma análise informal da complexidade do quicksort. Durante a primeira operação de partição, você percorre todos os itens do início ao fim da lista. Portanto, a quantidade de trabalho durante essa operação é proporcional a n, o comprimento da lista.

A quantidade de trabalho após essa partição é proporcional ao comprimento da sublista à esquerda mais o comprimento da sublista à direita, que juntos resultam em $n - 1$. E quando essas sublistas são divididas, há quatro partes cujo comprimento combinado é aproximadamente n, portanto o trabalho combinado é proporcional a n novamente. À medida que a lista é dividida em mais partes, o trabalho total permanece proporcional a n.

Segmento de lista	Item pivô
12 19 17 18 14 11 15 13 16	14
12 11 13	11
13 12	13
12	
16 19 15 17 18	15
19 18 17 16	18
16 17	16
17	
19	

Figura 3-12 Particionando uma sublista

Para completar a análise, você precisa determinar quantas vezes as listas são particionadas. Faça a suposição otimista de que, cada vez, a linha de divisão entre as novas sublistas acaba sendo a mais próxima possível do centro da sublista atual. Na prática, geralmente não é esse o caso. Você já conhece a discussão sobre o algoritmo de pesquisa binária que, ao dividir uma lista pela metade repetidamente, chega-se a um único elemento em cerca de $\log_2 n$ passos. Assim, o algoritmo é $O(n \log n)$ na melhor das hipóteses de desempenho.

Para obter o desempenho no pior caso, considere uma lista que já está ordenada. Se o elemento pivô escolhido for o primeiro, então há $n - 1$ elemento à sua direita na primeira partição, $n - 2$ elementos à sua direita na segunda partição e assim por diante, conforme mostrado na Figura 3-13.

Embora nenhum elemento seja trocado, o número total de partições é $n - 1$ e o número total de comparações realizadas é $\frac{1}{2} n^2 - \frac{1}{2} n$, o mesmo número que na ordenação por seleção e ordenação por bolha. Assim, no pior caso, o algoritmo quicksort é $O(n^2)$.

Se você implementar um quicksort rápido como um algoritmo recursivo, sua análise também deve considerar o uso de memória para a pilha de chamadas. Cada chamada recursiva requer uma quantidade constante de memória para um quadro de pilha, e há duas chamadas recursivas após cada partição. Assim, o uso de memória é $O(\log n)$ no melhor caso e $O(n)$ no pior caso.

Embora desempenho no pior caso possível do quicksort seja raro, os programadores certamente preferem evitá-lo. Escolher o pivô na primeira ou última posição não é uma estratégia inteligente. Outros métodos para escolher o pivô, como selecionar uma posição aleatória ou escolher a mediana do primeiro, intermediário e último elemento, podem ajudar a aproximar o desempenho $O(n \log n)$ no caso médio.

Implementação do quicksort

O algoritmo quicksort é mais facilmente codificado usando-se uma abordagem recursiva. O script a seguir define uma função **`quicksort`** de alto nível para o cliente, uma função recursiva

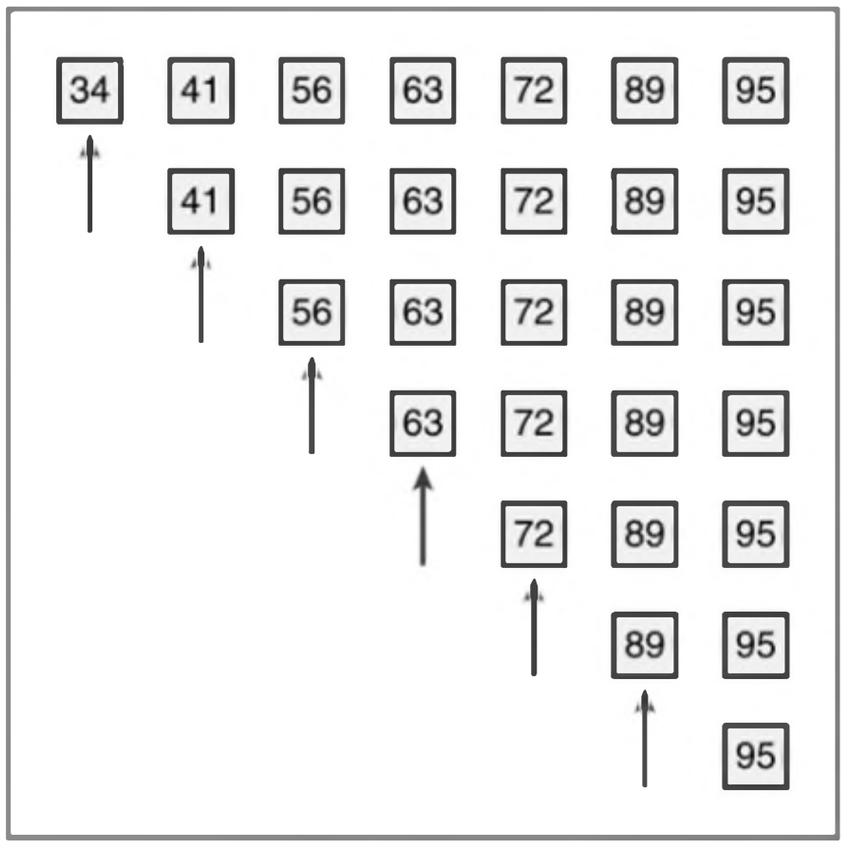


Figura 3-13 O pior cenário para quicksort (as setas indicam os elementos do pivô)

quicksortHelper para ocultar os argumentos extras para os pontos finais de uma sublista e uma função **partição**. O script executa o **quicksort** em uma lista de 20 inteiros ordenados aleatoriamente.

```
def quicksort(lyst):
    quicksortHelper(lyst, 0, len(lyst) - 1)

def quicksortHelper(lyst, left, right):
    if left < right:
        pivotLocation = partition(lyst, left, right)
        quicksortHelper(lyst, left, pivotLocation - 1)
        quicksortHelper(lyst, pivotLocation + 1, right)

def partition(lyst, left, right):
    # Encontra o pivô e troca-o pelo último item
    middle = (left + right) // 2
    pivot = lyst[middle]
    lyst[middle] = lyst[right]
    lyst[right] = pivot
    # Configura o ponto limite para a primeira posição
    boundary = left
    # Move os itens menores que o pivô para a esquerda
    for index in range(left, right):
        if lyst[index] < pivot:
            swap(lyst, index, boundary)
            boundary += 1
    # Troca o item de pivô e o item de limite
    swap (lyst, right, boundary)
    return boundary

# A definição anterior da função de troca entra aqui

import random

def main(size = 20, sort = quicksort):
    lyst = []
    for count in range(size):
        lyst.append(random.randint(1, size + 1))
    print(lyst)
    sort(lyst)
    print(lyst)

if __name__ == "__main__":
    main()
```

Filas

Este capítulo explora a fila, outra coleção linear que tem amplo uso na ciência da computação. Existem várias estratégias de implementação para filas — algumas baseadas em arrays e outras em estruturas ligadas. Para ilustrar a aplicação de uma fila, este capítulo desenvolve um estudo de caso que simula uma fila de caixa de supermercado. Ele termina com o exame de um tipo especial de fila, conhecido como fila com prioridades e mostra como é usada em um segundo estudo de caso.

Visão geral das filas

Como as pilhas, as filas são coleções lineares. Mas com filas, as inserções são restritas a uma extremidade, chamada de **traseira** (*rear*, em inglês) e remoções para a outra extremidade, chamadas de **frente** (*front*, em inglês). Uma fila, portanto, suporta um protocolo **primeiro a entrar, primeiro a sair** (*first-in first-out*, Fifo). Filas são onipresentes na vida cotidiana e ocorrem em qualquer situação em que pessoas ou coisas são alinhadas para serviço ou processamento por ordem de chegada. Filas de caixa em lojas, filas de pedágio em rodovias e filas de check-in de bagagem em aeroportos são exemplos conhecidos de filas.

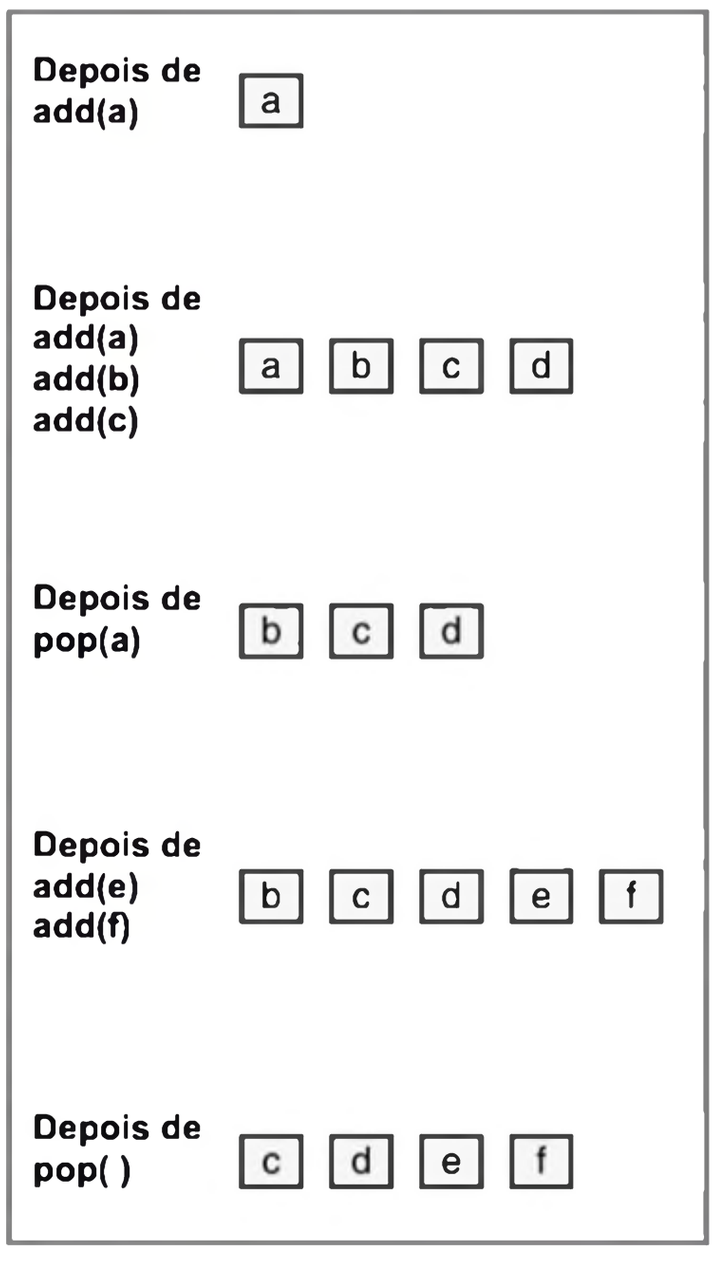


Figura 8-1 Os estados na vida de uma fila

As filas têm duas operações fundamentais: **add**, que adiciona um item ao final de uma fila e **pop**, que remove um item da frente. A Figura 8-1 mostra como uma fila pode aparecer em vários estágios de sua vida útil. Na figura, a frente da fila está à esquerda e a parte traseira está à direita.

Inicialmente, a fila está vazia. Em seguida, um item chamado **a** é adicionado. Depois, mais três itens chamados **b**, **c** e **d** são adicionados, após o que um item é exibido e assim por diante.

Há uma coleção chamada **fila com prioridades** relacionada a filas. Em uma fila, o item exibido, ou servido em seguida, é sempre o que está esperando há mais tempo. Mas em algumas circunstâncias, essa restrição é bem rígida e é preferível combinar a ideia da espera com a noção de prioridade. Em uma fila com prioridades, os itens com prioridade mais alta são exibidos antes dos com prioridade mais baixa e os itens com prioridade igual são exibidos na ordem Fifo. Considere, por exemplo, a maneira como os passageiros embarcam em uma aeronave. Os passageiros da primeira classe fazem fila e embarcam primeiro e os da classe econômica de baixa prioridade fazem fila e embarcam em segundo lugar. Mas isso não é uma fila com prioridades real porque, depois que a fila da primeira classe se esvazia e a fila da classe econômica começa a embarcar, os passageiros da primeira classe que chegam atrasados geralmente vão para o final da segunda fila. Em uma fila real com prioridades, eles passariam imediatamente para a frente de todos os passageiros da classe econômica.

A maioria das filas na ciência da computação envolve agendamento de acesso a recursos compartilhados. A lista a seguir descreve alguns exemplos:

- **Acesso à CPU** — Os processos são enfileirados para acesso a uma CPU compartilhada.
- **Acesso ao disco** — Os processos são enfileirados para acesso a um dispositivo de armazenamento secundário compartilhado.
- **Acesso à impressora** — Os trabalhos de impressão são enfileirados para acesso a uma impressora a laser compartilhada.

O agendamento de processos pode envolver filas simples ou filas de prioridades. Por exemplo, os processos que exigem entrada de teclado e saída na tela geralmente recebem acesso com prioridade mais alta à CPU do que aqueles que exigem muita computação. O resultado é que os usuários humanos, que tendem a avaliar a velocidade de um computador de acordo com o tempo de resposta, têm a impressão de que o computador é rápido.

Os processos que estão esperando por um recurso compartilhado também podem ser priorizados de acordo com a duração esperada, com processos curtos tendo maior prioridade do que os mais longos, novamente com a intenção de melhorar o tempo de resposta aparente de um sistema. Imagine 20 trabalhos de impressão enfileirados para acesso a uma impressora. Se 19 trabalhos tiverem 1 página e 1 trabalho, 200 páginas, mais usuários ficarão felizes se os trabalhos curtos tiverem prioridade mais alta e forem impressos primeiro.

A interface da fila e seu uso

Se estiverem com pressa, os programadores Python podem usar uma lista Python para emular uma fila. Embora não importe quais extremidades da lista você vê como a frente e a retaguarda da fila, a estratégia mais simples é usar o método **`append`** de **`list`** para adicionar um item ao final dessa fila e usar o método **`pop(0)`** de **`list`** para remover e devolver o item no início da fila. Como vemos no caso das pilhas, a principal desvantagem dessa opção é que todas as outras operações de lista também podem manipular a fila. Isso inclui a inserção, substituição

e remoção de um item em qualquer posição. Essas operações extras violam o espírito de uma fila como um tipo de dado abstrato. Além disso, a remoção de um item no início de um objeto **`list`** do Python é uma operação de tempo linear. Esta seção define uma interface mais restrita, ou conjunto de operações, para qualquer implementação de fila e mostra como essas operações são usadas.

Além das operações **add** e **pop**, será útil ter uma operação **peek**, que retorna o item na frente da fila. As demais operações na interface da fila são padrão para qualquer coleção. A Tabela 8-1 lista todos eles.

Método de fila	O que ele faz
`q.isEmpty()`	Retorna **`True`** se **`q`** estiver vazio, ou **`False`** caso contrário.
`__len__(q)`	O mesmo que **`len(q)`**. Retorna o número de itens em **`q`**.
`__str__(q)`	O mesmo que **`str(q)`**. Retorna a representação de string de **`q`**.
`q.__iter__()`	Igual a **`iter (q)`**, ou **`para o item in q:`**. Visita cada item em **`q`**, da frente para trás.
`q.__contains__(item)`	O mesmo que **`item in q`**. Retorna **`True`** se o item estiver em **`q`** ou **`False`** caso contrário.
`q1__add__(q2)`	Igual a **`q1 + q2`**. Retorna uma nova fila contendo os itens em **`q1`** seguidos pelos itens em **`q2`**.
`q.__eq__(anyObject)`	O mesmo que **`q == anyObject`**. Retorna **`True`** se **`q`** for igual a **`anyObject`** ou **`False`** caso contrário. Duas filas são iguais se os itens nas posições correspondentes são iguais.
`q.clear()`	Torna **`q`** vazio.
`q.peek()`	Retorna o item na frente de **`q`**. *Precondição*: **`q`** não deve estar vazio; levanta um **`KeyError`** se a fila estiver vazia.
`q.add(item)`	Adiciona **`item`** à parte traseira de **`q`**.
`q.pop()`	Remove e retorna o item para a frente de **`q`**. *Precondição* **`q`** não deve estar vazio; levanta um **`KeyError`** se a fila estiver vazia.

Tabela 8-1 Os métodos na interface de fila

Observe que os métodos **pop** e **peek** têm uma precondição importante e geram uma exceção se o usuário da fila não atender a essa precondição.

Agora que uma interface de fila foi definida, veremos como usá-la. A Tabela 8-2 mostra como as operações listadas anteriormente afetam uma fila chamada **`q`**.

Suponha que qualquer classe de fila que implementa essa interface também terá um construtor que permite ao usuário criar uma instância de fila. Posteriormente, neste capítulo, duas implementações diferentes, chamadas **`ArrayQueue`** e **`LinkedQueue`**, serão consideradas. Por enquanto, suponha que alguém tenha codificado essas implementações para que você possa usá-las. O próximo segmento de código mostra como elas podem ser instanciadas:

```
q1 = ArrayQueue()                    # Cria fila de array vazia
q2 = LinkedQueue([3, 6, 0])          # Cria fila ligada com determinados itens
```

Operação	Estado da fila após a operação	Valor retornado	Comentário
`q = <Tipo de fila>()`			Inicialmente, a fila está vazia.
`q.add(a)`	`a`		A fila contém o único item **a**.
`q.add(b)`	`a b`		**a** está na parte da frente da fila e **b** está na parte traseira.
`q.add(c)`	`a b c`		**c** é adicionado na parte traseira.
`q.isEmpty()`	`a b c`	`False`	A fila não está vazia.
`len(q)`	`a b c`	`3`	A fila contém três itens.
`q.peek()`	`a b c`	`a`	Retorna o item na frente da fila sem removê-lo.
`q.pop()`	`b c`	`a`	Remove o item da frente da fila e o devolve. **b** agora é o item da frente.
`q.pop()`	`c`	`b`	Remove e retorna **b**.
`q.pop()`		`c`	Remove e retorna **c**.
`q.isEmpty()`		`True`	A fila está vazia.
`q.peek()`		`exception`	Uma fila vazia lança uma exceção.
`q.pop()`		`Exception`	Tenta abrir uma fila vazia por meio de uma exceção.
`q.add(d)`	`d`		**d** é o item na frente.

Tabela 8-2 Os efeitos das operações de fila

EXERCÍCIOS

1. Usando o formato de Tabela 8-2, preencha uma tabela que envolve a seguinte sequência de operações de fila.

Operação	Estado da fila após a operação	Valor retornado
`Crie q`		
`q.add(a)`		
`q.add(b)`		
`q.add(c)`		
`q.pop()`		
`q.pop()`		
`q.peek()`		
`q.add(x)`		
`q.pop()`		

(continua)

Operação	Estado da fila após a operação	Valor retornado
`q.pop()`		
`q.pop()`		

2. Defina uma função chamada **`stackToQueue`**. Essa função espera uma pilha como um argumento. A função cria e retorna uma instância de **`LinkedQueue`** que contém os itens da pilha. A função considera que a pilha tem a interface descrita no Capítulo 7, "Pilhas". As pós-condições da função são que a pilha seja mantida no mesmo estado em que estava antes de a função ser chamada e que o item na frente da fila seja o que está no topo da pilha.

Duas aplicações de filas

Este capítulo agora examina rapidamente duas aplicações das filas: uma envolvendo simulações por computador e a outra envolvendo escalonamento de CPU round-robin.

Simulações

Simulações em computador são usadas para estudar o comportamento de sistemas do mundo real, especialmente quando é impraticável ou perigoso fazer experimentos com esses sistemas diretamente. Por exemplo, uma simulação por computador pode representar o fluxo de tráfego em uma rodovia movimentada. Os planejadores urbanos podem então experimentar fatores que afetam o fluxo do tráfego, como o número e tipos de veículos na rodovia, os limites de velocidade para diferentes tipos de veículos, o número de faixas na rodovia e a frequência dos pedágios. Os resultados dessa simulação podem incluir o número total de veículos capazes de se mover entre pontos especificados em um período designado e a duração média de uma viagem. Executando a simulação com muitas combinações de entradas, os planejadores podem determinar a melhor forma de atualizar as seções da rodovia, sujeito às sempre presentes restrições de tempo, espaço e dinheiro.

Como um segundo exemplo, considere o problema enfrentado pelo gerente de um supermercado que está tentando determinar o número necessário de pessoas trabalhando nos caixas em vários horários do dia. Alguns fatores importantes nessa situação são:

- A frequência com que chegam novos clientes
- O número de pessoas trabalhando nos caixas disponíveis
- O número de itens no carrinho de compras de um cliente
- O período de tempo considerado

Esses fatores poderiam ser entradas para um programa de simulação, o que determinaria o número total de clientes processados, o tempo médio que cada cliente espera pelo serviço e o número de clientes que permanecem na fila no final do período simulado. Variando as

entradas, especialmente a frequência das chegadas de clientes e o número de pessoas disponíveis no caixa, um programa de simulação pode ajudar o gerente a tomar decisões eficazes sobre a equipe em horários movimentados e desacelerados do dia. Adicionando uma entrada que quantifica a eficiência de diferentes equipamentos de pagamento no caixa, o gerente pode até decidir se é mais econômico adicionar mais caixas ou comprar equipamentos melhores e mais eficientes.

Uma característica comum dos dois exemplos e dos problemas de simulação em geral, é a variabilidade momento a momento dos fatores essenciais. Considere a frequência das chegadas de clientes nas estações de pagamento no caixa. Se os clientes chegassem em intervalos precisos, cada um com o mesmo número de itens, seria fácil determinar o número de caixas de plantão. Mas essa regularidade não reflete a realidade de um supermercado. Às vezes, vários clientes aparecem praticamente no mesmo momento e, em outras ocasiões, nenhum novo cliente chega por vários minutos. Além disso, o número de itens varia de cliente para cliente; portanto, o mesmo acontece com a quantidade de serviço que cada cliente requer. Toda essa variabilidade torna difícil o desenvolvimento de fórmulas para responder a perguntas simples sobre o sistema, por exemplo, a maneira como o tempo de espera de um cliente varia de acordo com o número de caixas em serviço. Um programa de simulação, por outro lado, evita a necessidade de fórmulas mimetizando a situação real e coletando estatísticas pertinentes.

Os programas de simulação usam uma técnica simples para simular a variabilidade. Por exemplo, suponha que se espera que novos clientes cheguem, em média, uma vez a cada 4 minutos. Então, durante cada minuto do tempo simulado, um programa pode gerar um número aleatório entre 0 e 1. Se o número é menor que 1/4, o programa adiciona um novo cliente a uma fila de checkout; do contrário, não adiciona. Esquemas mais sofisticados baseados em funções de distribuição de probabilidade produzem resultados ainda mais realistas. Obviamente, sempre que o programa é executado, os resultados mudam ligeiramente, mas isso só contribui para o realismo da simulação.

Agora veremos o papel comum desempenhado pelas filas nesses exemplos. Os dois exemplos envolvem provedores de serviços e consumidores de serviços. No primeiro exemplo, os provedores de serviço incluem cabines de pedágio e faixas de tráfego, e os consumidores de serviço são os veículos que esperam nas cabines de pedágio e dirigem nas faixas de tráfego. No segundo exemplo, os caixas fornecem um serviço que é consumido pelos clientes em espera. Para emular essas condições em um programa, associe cada provedor de serviço a uma fila de consumidores de serviço.

As simulações operam manipulando essas filas. A cada tique de um relógio imaginário, uma simulação adiciona vários números de consumidores às filas e fornece aos consumidores no início de cada fila outra unidade de serviço. Depois que um consumidor recebeu a quantidade necessária de serviço, ele sai da fila e o próximo consumidor dá um passo à frente. Durante a simulação, o programa acumula estatísticas como quantos tiques cada consumidor esperou em uma fila e a porcentagem de tempo que cada provedor está ocupado. A duração de um tique é escolhida para corresponder ao problema que está sendo simulado. Pode representar um milissegundo, um minuto ou uma década. No próprio programa, um tique provavelmente corresponde a uma passagem pelo laço de processamento principal do programa.

Você pode usar métodos orientados a objetos para implementar programas de simulação. Por exemplo, em uma simulação de supermercado, cada cliente é uma instância de uma classe **`Customer`**. Um objeto cliente controla quando o cliente começa a permanecer na fila, quando o serviço é prestado pela primeira vez e quanto serviço é necessário. Da mesma forma, um caixa é uma instância de uma classe **`Cashier`** e cada objeto caixa contêm uma fila de objetos cliente. Uma classe simuladora coordena as atividades dos clientes e caixas. A cada tique do relógio, o objeto de simulação faz o seguinte:

- Gera objetos cliente conforme apropriado
- Atribui clientes a caixas
- Informa cada caixa para fornecer uma unidade de serviço ao cliente no início da fila

No primeiro estudo de caso deste capítulo, você desenvolve um programa baseado nas ideias anteriores. Nos exercícios, você estende o programa.

Agendamento de CPU round-robin

A maioria dos computadores modernos permite que vários processos compartilhem uma única CPU. Existem várias técnicas para programar esses processos. O mais comum, chamado **programação round-robin**, adiciona novos processos ao final de um **fila pronta**, que consiste em processos esperando para usar a CPU. Cada processo na fila pronta é exibido de cada vez e recebe uma fatia do tempo de CPU. Quando a fração de tempo se esgota, o processo retorna para o final da fila, como mostra a Figura 8-2.

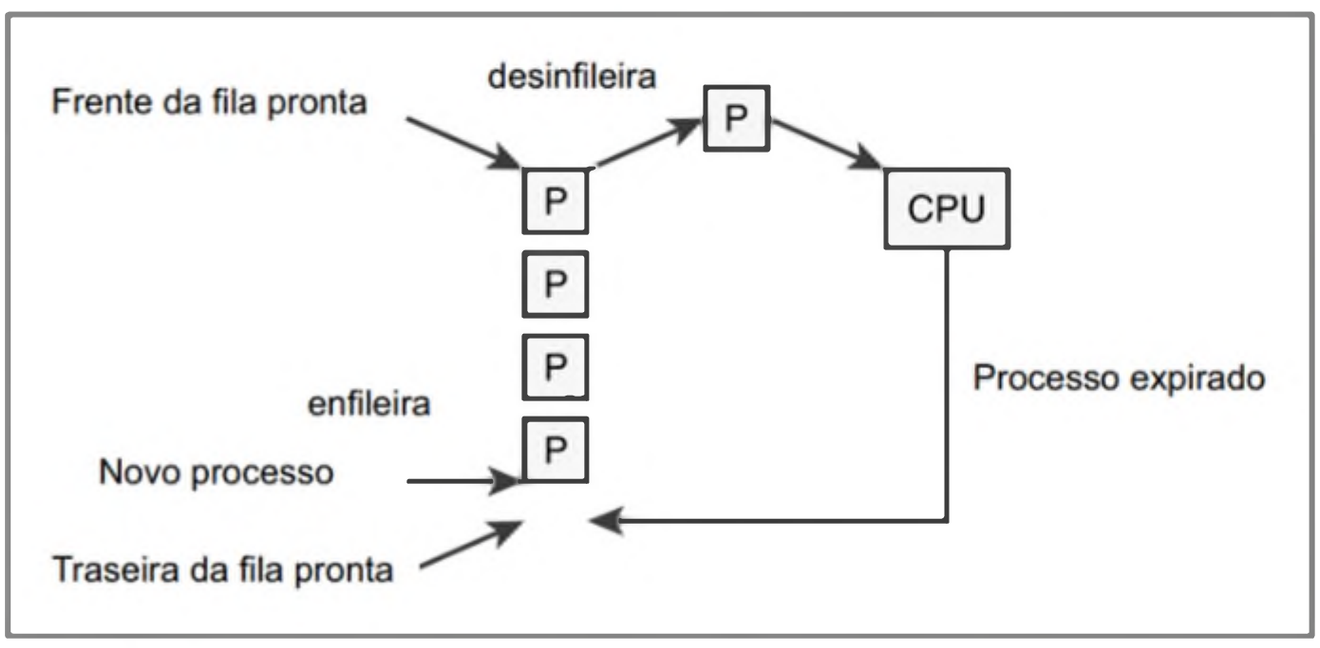


Figura 8-2 Processos de agendamento para uma CPU

Geralmente, nem todos os processos precisam da CPU com a mesma urgência. Por exemplo, a satisfação do usuário com um computador é significativamente influenciada pelo tempo de resposta do computador às entradas de teclado e mouse. Portanto, faz sentido atribuir precedência aos processos que tratam essas entradas. O agendamento round-robin se adapta a esse requisito usando uma fila com prioridades e atribuindo a cada processo uma prioridade apropriada. Como acompanhamento para essa discussão, o segundo estudo de caso neste capítulo mostra como você pode usar uma fila com prioridades para agendar pacientes em um pronto-socorro.

Implementações de filas

A abordagem deste capítulo à implementação de filas é semelhante àquela usada para pilhas. A estrutura de uma fila se presta a uma implementação de array ou a uma implementação ligada. Para obter algum comportamento padrão gratuitamente, crie uma subclasse de cada implementação de fila sob a classe **`AbstractCollection`** em sua estrutura de coleção (consulte o Capítulo 6, "Herança e classes abstratas"). Como a implementação ligada é relativamente simples e direta, considere-a primeiro.

EXERCÍCIOS

1. Suponha que os clientes em um supermercado 24 horas estejam prontos para fazer o pagamento no valor exato de um a cada dois minutos. Suponha também que leve exatamente cinco minutos para um caixa processar um cliente. Quantos caixas precisam estar de plantão para atender a demanda? Os clientes precisarão esperar na fila? Quanto tempo ocioso cada caixa terá por hora?
2. Agora, suponha que as taxas — um cliente a cada dois minutos e cinco minutos por cliente — representem as médias. Descreva de maneira qualitativa como isso afetará o tempo de espera do cliente. Essa mudança afetará a quantidade média de tempo ocioso por caixa? Para ambas as situações, descreva o que acontece se o número de caixas diminuir ou aumentar.

Uma implementação ligada das filas

As implementações ligadas das pilhas e filas têm muito em comum. Ambas as classes, **`LinkedStack`** e **`LinkedQueue`**, utilizam uma classe **`Node`** unicamente ligada para implementar nós. A operação **`pop`** remove o primeiro nó na sequência em ambas as coleções. Contudo, **`LinkedQueue.add`** e **`LinkedStack.push`** diferem. A operação **`push`** adiciona um nó no início da sequência, enquanto **`add`** adiciona um nó no fim. Para fornecer acesso rápido a ambas as extremidades da estrutura ligada de uma fila, existem ponteiros externos para as duas extremidades. A Figura 8-3 mostra uma fila ligada contendo quatro itens.

As variáveis de instância **`front`** e **`rear`** da classe **`LinkedQueue`** recebem um valor inicial de **`None`**. Uma variável chamada **`size`**, já definida na estrutura da coleção, monitora o número de elementos atualmente na fila.

Durante uma operação **`add`**, crie um novo nó, defina o próximo ponteiro do último nó para o novo nó e defina a variável **`rear`** para o novo nó, conforme mostra a Figura 8-4.

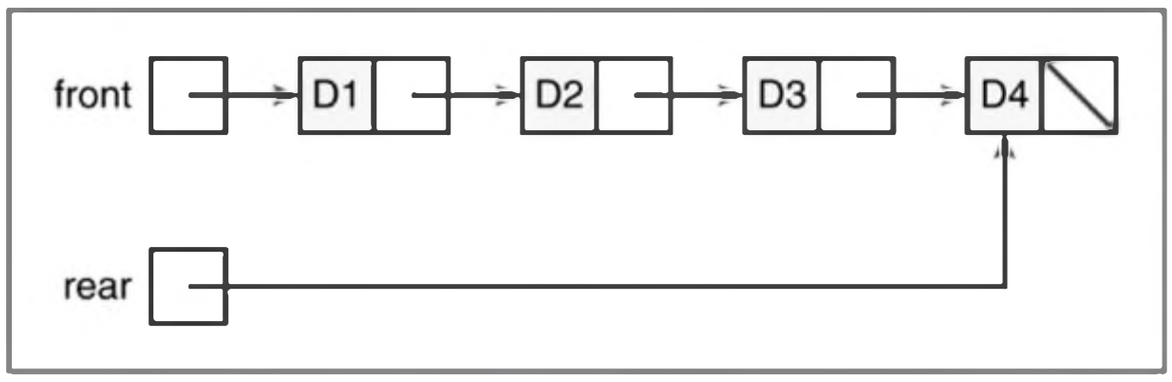


Figura 8-3 Uma fila ligada com quatro itens

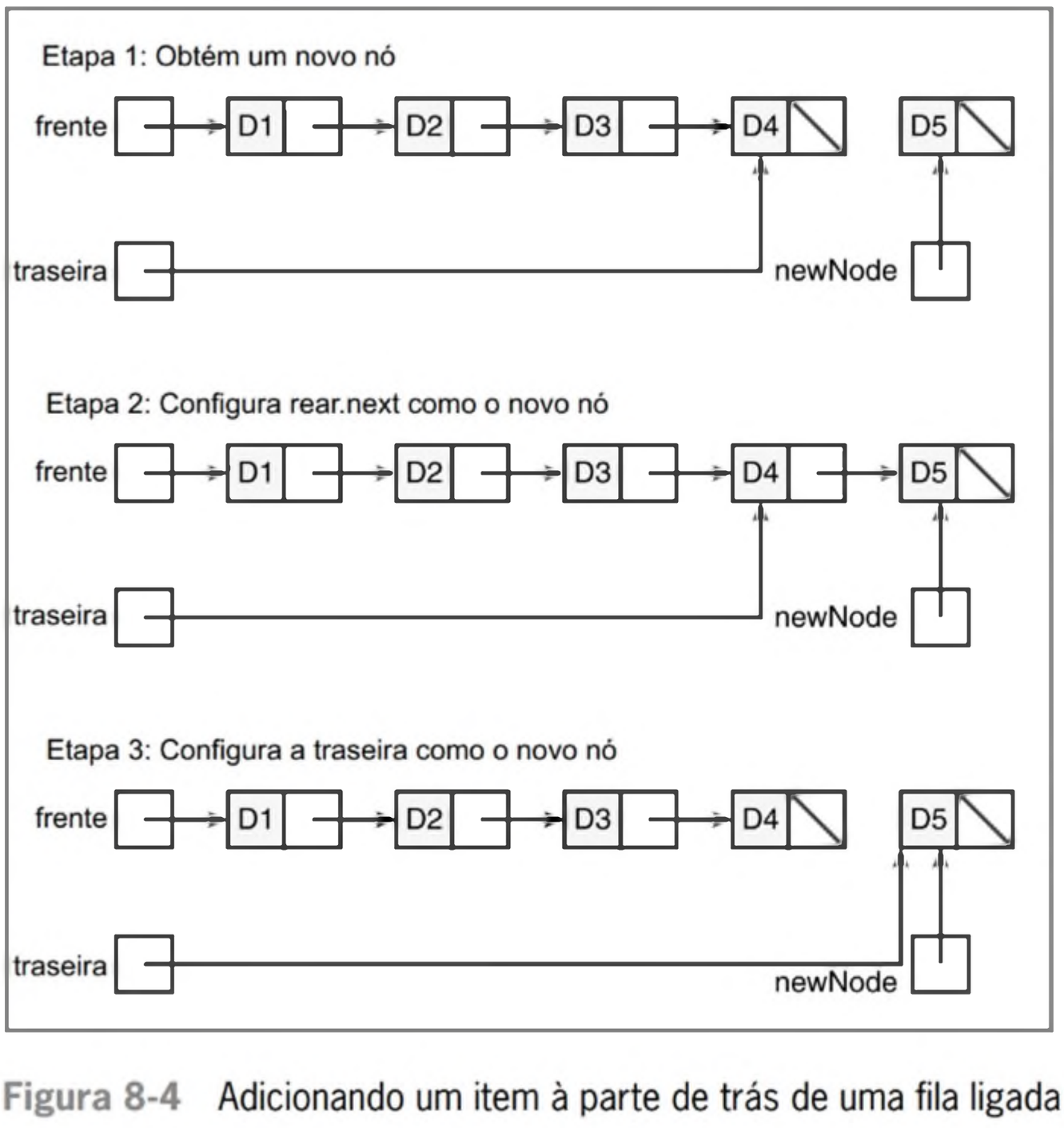


Figura 8-4 Adicionando um item à parte de trás de uma fila ligada

Eis o código para o novo método **add**:

```
def add(self, newItem):
    """Adiciona newItem ao final da fila."""
    newNode = Node(newItem, None)
    if self.isEmpty():
        self.front = newNode
    else:
        self.rear.next = newNode
    self.rear = newNode
    self.size += 1
```

Como mencionado anteriormente, **LinekdQueue.pop** é similar a **LinkedStack.pop**. Mas se a fila ficar vazia após uma operação **pop**, os ponteiros **front** e **rear** devem ser configurados como **None**. Eis o código:

```
def pop(self):
    """Retorna o item no topo da pilha.
    Precondição: a pilha não está vazia."""
    # Verifica a precondição aqui
    oldItem = self.front.data
    self.front = self.front.next
    if self.front is None:
        self.rear = None
    self.size -= 1
    return oldItem
```

A conclusão da classe **LinkedQueue**, incluindo a aplicação das precondições nos métodos **pop** e **peek**, é deixada como um exercício para você.

Uma implementação de array

As implementações de array de pilhas e filas têm menos em comum do que as implementações ligadas. A implementação do array de uma pilha precisa acessar itens apenas no final lógico do array. Entretanto, a implementação do array de uma fila deve acessar itens no início lógico e no final lógico. Fazer isso de maneira computacionalmente eficiente é complexo, portanto, é melhor abordar o problema em uma sequência de três tentativas.

Primeira tentativa

A primeira tentativa de implementação mantém fixa a fila à frente da posição de índice 0 e mantém uma variável de índice, chamada **rear**, que aponta para o último item na posição $n - 1$, em que n é o número de itens na fila. Uma imagem dessa fila, com quatro itens de dados em um array de seis células, é mostrada na Figura 8-5.

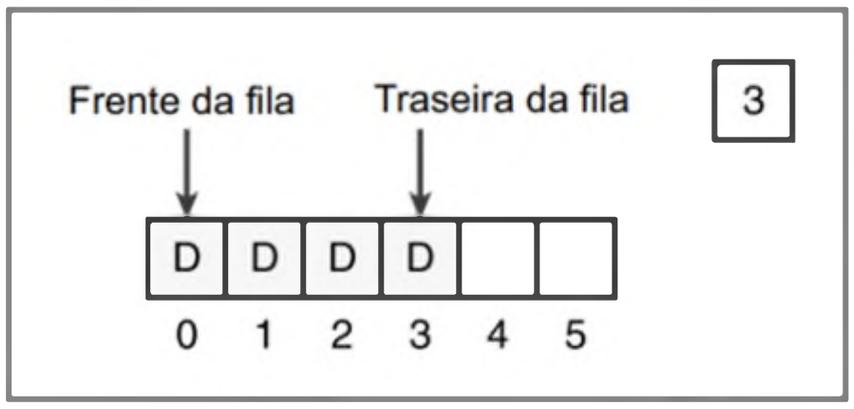


Figura 8-5 Uma implementação de array de uma fila com quatro itens

Para essa implementação, a operação **add** é eficiente. Mas a operação **pop** envolve deslocar todos, exceto o primeiro item, no array para a esquerda, o que é um processo $O(n)$.

Segunda tentativa

Você pode evitar o comportamento linear de **pop** não deslocando itens deixados a cada vez que a operação é aplicada. A implementação modificada mantém um segundo índice, chamado

front, que aponta para o item na frente da fila. O ponteiro **front** começa em 0 e avança pelo array à medida que os itens são removidos. A Figura 8-6 mostra essa fila depois das cinco operações **add** e duas **pop**.

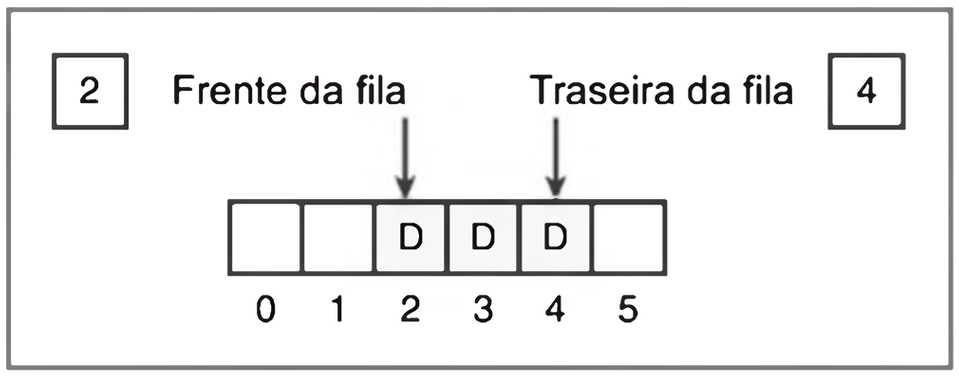


Figura 8-6 Uma implementação de array de uma fila com um ponteiro frontal

Observe que, nesse esquema, as células à esquerda do ponteiro frontal da fila só são usadas depois que todos os elementos foram deslocados para a esquerda, o que é feito sempre que o ponteiro traseiro está prestes a extrapolar o final. Agora, o tempo máximo de execução de **pop** é O(1), mas vem a custo de aumentar o tempo máximo de execução de **add** de O(1) para O(*n*). Além disso, a memória do array à esquerda do ponteiro **front** não está disponível para a fila.

Terceira tentativa

Usando uma **implementação de array circular**, você pode alcançar simultaneamente bons tempos de execução tanto para **add** como para **pop**. A implementação se assemelha à anterior em um aspecto: os ponteiros **front** e **rear** começam no início do array.

Mas o ponteiro **front** agora "persegue" o ponteiro **rear** ao longo do array. Durante a operação **add**, o ponteiro **rear** se move mais para a frente do ponteiro **front** e durante a operação **pop**, o ponteiro **front** fica à frente uma posição. Quando um dos ponteiros está prestes a extrapolar o final do array, o ponteiro é redefinido como 0. Isso tem o efeito de dar uma volta na fila para o início do array sem o custo de mover itens.

Como exemplo, suponha que uma implementação de array use seis células, que seis itens foram adicionados e que dois itens foram removidos. De acordo com esse esquema, o próximo **add** reinicia o ponteiro **rear** como 0. A Figura 8-7 mostra o estado do array antes e depois de o ponteiro **rear** ser redefinido para 0 pela última operação **add**.

O ponteiro **rear** agora parece perseguir o ponteiro **front** até que **front** atinja o final do array e, nesse ponto, também é redefinido como 0. Como você pode ver prontamente, os tempos máximos de execução de **add** e **pop** agora são O(1).

Você naturalmente se perguntará o que acontece quando a fila fica cheia e como a implementação pode detectar essa condição. Mantendo uma contagem dos itens na fila, você pode determinar se a fila está cheia ou vazia. Quando essa contagem é igual ao tamanho da array, você sabe que é hora de redimensionar.

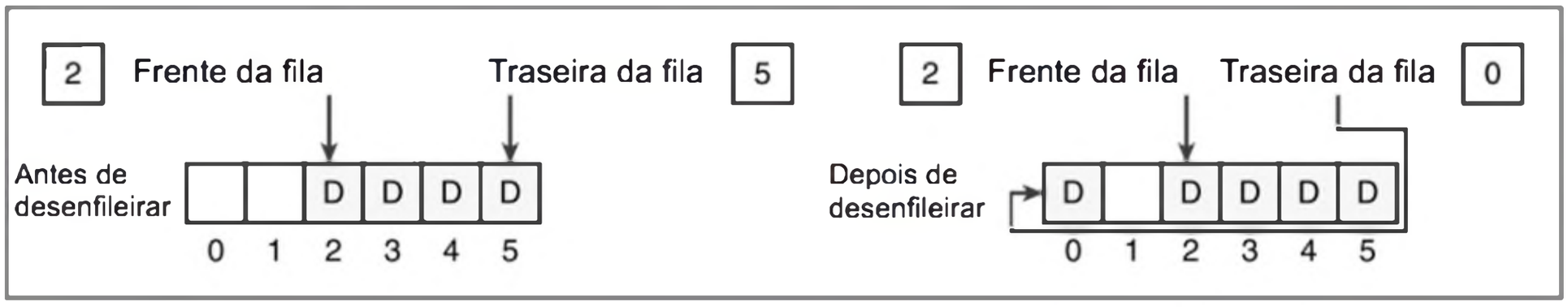


Figura 8-7 Circulando dados em torno de uma implementação de array circular de uma fila

Após o redimensionamento, você quer que a fila ocupe o segmento inicial do novo array, com o ponteiro **`front`** configurado como 0. Para alcançar isso, execute as seguintes etapas:

1. Crie um novo array com o dobro do tamanho do array atual.
2. Itere pela fila, usando seu laço **`for`**, para copiar itens para o novo array, começando na posição 0 nesse array.
3. Reinicie a variável **`items`** para o novo array.
4. Configure **`front`** como 0 e **`rear`** como o comprimento da fila menos 1.

O código de redimensionamento para uma fila baseada em array depende do iterador da fila. Como esse iterador pode ter de percorrer o array circular, o código é mais complexo do que o código para os iteradores de coleções baseadas em array desenvolvidas anteriormente neste livro. A conclusão da implementação do array circular da classe **`ArrayQueue`** é deixada como um exercício para você.

Análise de tempo e espaço para as duas implementações

A análise de tempo e espaço para as duas classes de fila é similar àquela para as classes de pilha correspondentes, portanto, não se detenha aos detalhes. Considere primeiro a implementação ligada de filas. O tempo de execução dos métodos **`__str__`**, **`__add__`** e **`__eq__`** é O(n). O tempo máximo de execução de todos os outros métodos é O(1). Em particular, como há ligações externas para os nós inicial e final na estrutura ligada da fila, você pode acessar esses nós em tempo constante. A necessidade de espaço total é $2n + 3$, onde n é o tamanho da fila. Há uma referência a um dado e um ponteiro para o próximo nó em cada um dos n nós e há três células para o tamanho lógico da fila e ponteiros de início e fim.

Para a implementação de array circular de filas, se o array for estático, o tempo máximo de execução de todos os métodos além de **`__str__`**, **`__add__`** e **`__eq__`** é O(1). Em particular, nenhum item no array é deslocado durante **`add`** ou **`pop`**. Se o array for dinâmico, **`add`** e **`pop`** pulam para O(n) sempre que o array é redimensionado, mas mantém um tempo médio de execução de O(1). A utilização do espaço para a implementação do array depende novamente do fator de carga, conforme discutido no Capítulo 4, "Arrays e estruturas ligadas". Para fatores de carga acima de ½, uma implementação de array faz uso mais eficiente da memória do que uma implementação ligada e para fatores de carga abaixo de ½, o uso de memória é menos eficiente.

EXERCÍCIOS

1. Escreva um segmento de código que use uma instrução **`if`** durante um método **`add`** para ajustar o índice traseiro da implementação do array circular de **`ArrayQueue`**. Você pode considerar que a implementação da fila use as variáveis **`self.rear`** e **`self.items`** para referenciar o índice traseiro e o array, respectivamente.
2. Escreva um segmento de código que use o operador **`%`** durante um método **`add`** a fim de ajustar o índice traseiro da implementação do array circular de **`ArrayQueue`** para evitar o uso de uma instrução **`if`**. Você pode considerar que a implementação da fila use as variáveis **`self.rear`** e **`self.items`** para referenciar o índice traseiro e o array, respectivamente.

ESTUDO DE CASO: Simulando uma fila de caixa de supermercado

Neste estudo de caso, você desenvolve um programa para simular caixas de supermercado. Para manter o programa simples, alguns fatores importantes encontrados em uma situação realista de supermercado foram omitidos; você é solicitado a adicioná-los de volta como parte dos exercícios.

Solicitação

Escreva um programa que permita ao usuário prever o comportamento de uma fila de caixa de supermercado sob várias condições.

Análise

Para simplificar, as seguintes restrições são impostas:

- Há apenas uma fila de checkout, operada por um caixa.
- Cada cliente tem o mesmo número de itens para checkout e requer o mesmo tempo de processamento.
- A probabilidade de um novo cliente chegar ao caixa não varia ao longo do tempo.

As entradas para o programa de simulação são:

- O tempo total, em minutos abstratos, que a simulação deve ser executada.
- O número de minutos necessários para atender a um cliente individual.
- A probabilidade de que um novo cliente chegue à fila do caixa durante o próximo minuto. Essa probabilidade deve ser um número de ponto flutuante maior que 0 e menor ou igual a 1.

As saídas do programa são o número total de clientes processados, o número de clientes restantes na fila quando o tempo se esgota e o tempo médio de espera de um cliente. A Tabela 8-3 resume as entradas e saídas.

(continua)

(continuação)

Entradas	Intervalo de valores para entradas	Saidas
Minutos totais	0 < = total <=1000	Total de clientes processados
Média de minutos por cliente	0 < média <= total	Os clientes que permanecem na fila
Probabilidade de uma nova chegada no próximo minuto	0 < probabilidade <= 1	Tempo médio de espera

Tabela 8-3 Entradas e saídas do simulador de caixa de supermercado

A interface de usuário

A seguinte interface de usuário para o sistema foi proposta:

```
Bem-vindo ao Market Simulator
Insira o tempo total de execução: 60
Insira o tempo médio por cliente: 3
Insira a probabilidade de uma nova entrada: 0.25
TOTAIS PARA A CAIXA
"Número de clientes atendidos: 16
Número de clientes restantes na fila: 1
Tempo médio que os clientes gastam
Esperando para ser servido: 2.3
```

Classes e responsabilidades

No que diz respeito às classes e suas responsabilidades gerais, o sistema é dividido em uma função **main** e várias classes de modelo. A função **main** é responsável por interagir com o usuário, validar os três valores de entrada e se comunicar com o modelo. O projeto e a implementação dessa função não requerem comentários e o código da função não é apresentado. As classes no modelo estão listadas na Tabela 8-4.

Os relacionamentos entre essas classes são mostrados na Figura 8-8.

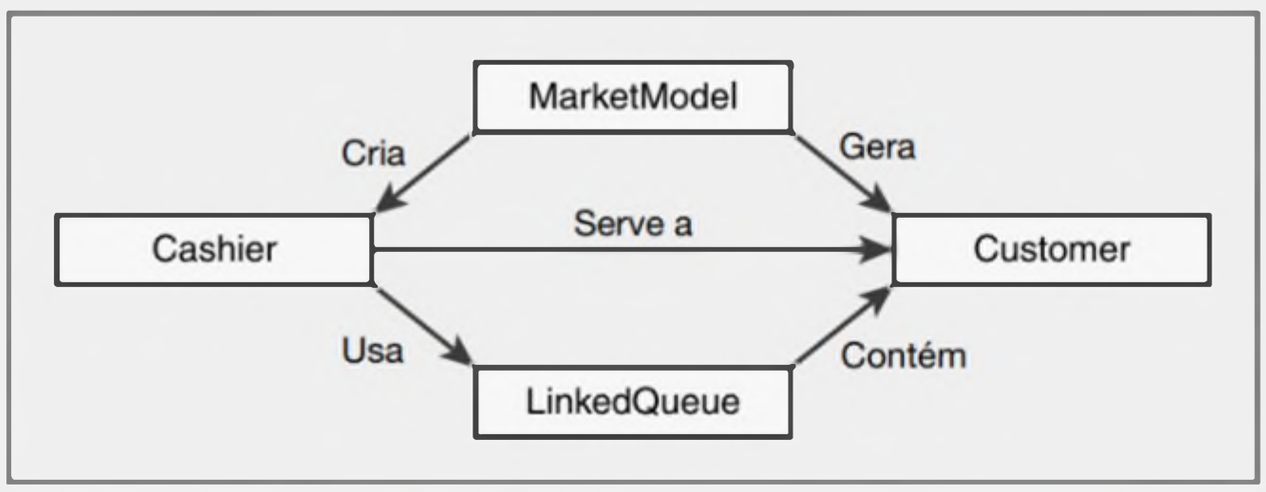


Figura 8-8 Um diagrama de classes do simulador de caixa de supermercado

(continua)

(continuação)

Classe	Responsabilidades
`MarketModel`	Um modelo de mercado faz o seguinte: 1. Executa a simulação. 2. Cria um objeto caixa. 3. Envia novos objetos cliente para o caixa. 4. Mantém um relógio abstrato. 5. Durante cada tique-taque do relógio, instrui o caixa a fornecer outra unidade de atendimento a um cliente.
`Cashier`	Um objeto caixa faz o seguinte: 1. Contém uma fila de objetos cliente. 2. Adiciona novos objetos cliente a essa fila quando instruído a fazê-lo. 3. Remove clientes da fila sucessivamente. 4. Fornece ao cliente atual uma unidade de atendimento quando instruído a fazê-lo e libera o cliente quando o atendimento é concluído.
`Customer`	Um objeto cliente: 1. Conhece a hora de chegada e o atendimento necessário. 2. Conhece quando o caixa prestou atendimento suficiente. A classe como um todo gera novos clientes quando orientada a fazer isso de acordo com a probabilidade de chegada de um novo cliente.
`LinkedQueue`	Usado por um caixa para representar uma fila de clientes.

Tabela 8-4 As classes no modelo

O projeto geral do sistema reflete-se no diagrama de colaboração mostrado na Figura 8-9.

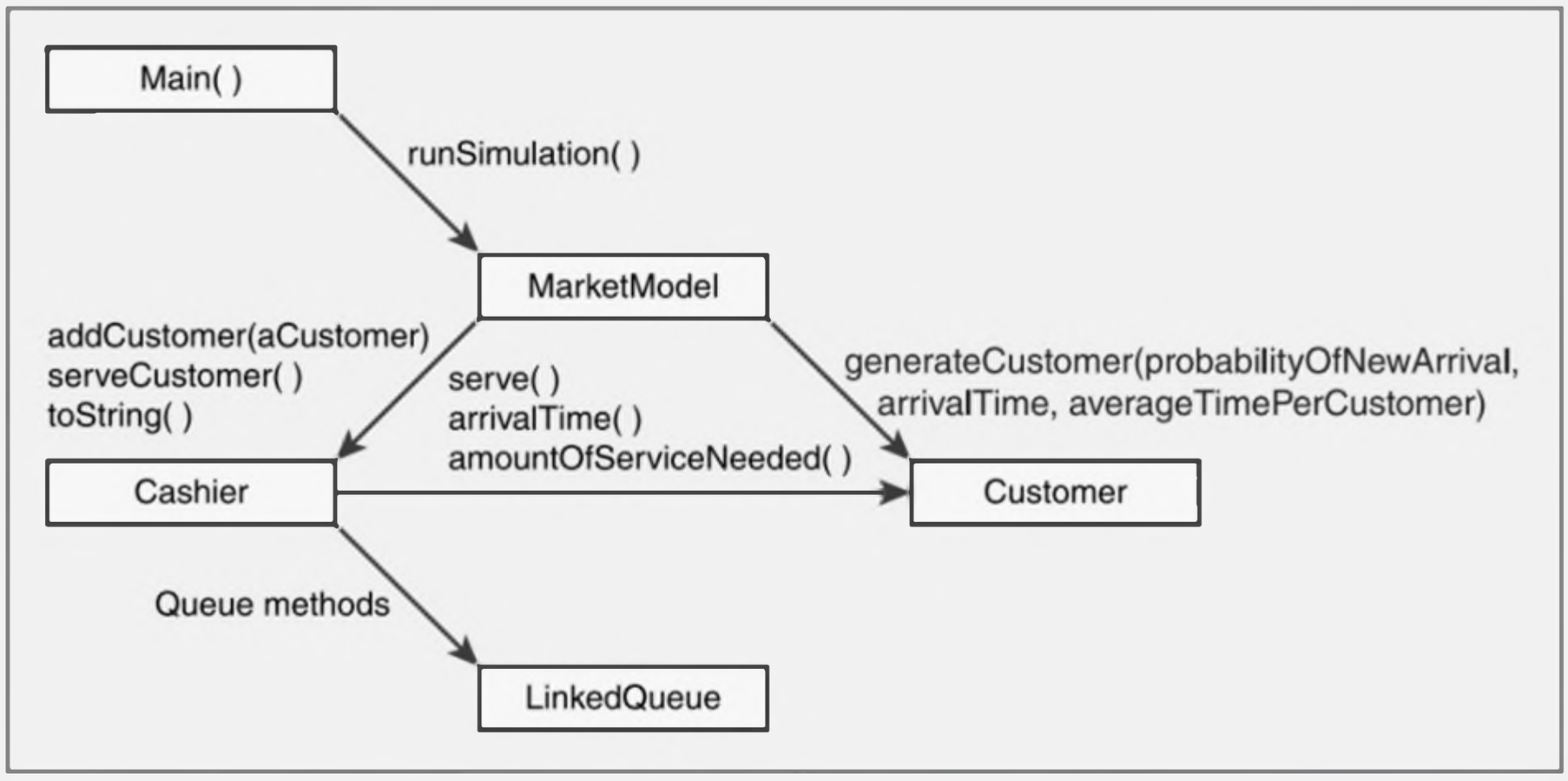


Figura 8-9 Um diagrama de colaboração para o simulador de caixa de supermercado

(continua)

(continuação)

Agora você pode projetar e implementar cada classe sucessivamente.

Como a situação de checkout foi restrita, o projeto da classe **`MarketModel`** é bastante simples. O construtor faz o seguinte:

4. Salva as entradas — probabilidade de nova chegada, duração da simulação e tempo médio por cliente.
5. Cria o único caixa.

O único outro método necessário é **`runSimulation`**. O método executa o relógio abstrato que conduz o processo de checkout. A cada tique do relógio, o método faz três coisas:

1. Pede para a classe **`Customer`** gerar um cliente, o que pode ou não acontecer, dependendo da probabilidade de uma nova chegada ou uma nova saída de um gerador de números aleatórios.
2. Se um novo cliente é gerado, envia o novo cliente para o caixa.
3. Instrui o caixa para fornecer uma unidade de atendimento ao cliente atual.

Quando a simulação termina, o método **`runSimulation`** retorna os resultados do caixa para a exibição. Eis o pseudocódigo para o método:

```
para cada minuto da simulação
     pede à classe Cliente para gerar um cliente
     se um cliente for gerado
          cashier.addCustomer(customer)
          cashier.serveCustomers(current time)
return cashier's results
```

Observe que o algoritmo do pseudocódigo pede à classe **`Customer`** uma instância dela própria. Como é apenas provável que um cliente chegue a dado minuto, às vezes um cliente não será gerado. Em vez de codificar a lógica para fazer essa escolha nesse nível, você pode inseri-la em um **`método de classe`** na classe **`Customer`**. A partir do modelo, o método **`generateCustomer`** da classe **`Customer`** recebe a probabilidade de chegada de um novo cliente, a hora atual e o tempo médio necessário por cliente. O método usa essas informações para determinar se deve criar um cliente e, se sim, como inicializá-lo. O método retorna o novo objeto **`Customer`** ou o valor **`None`**. A sintaxe da execução de um método de classe é semelhante à de um método de instância, exceto que o nome à esquerda do ponto é o nome da classe.

Eis uma listagem completa da classe **`MarketModel`**:

```
"""
Arquivo: marketmodel.py
"""
from cashier import Cashier
from customer import Customer
class MarketModel(object):

    def __init__(self, lengthOfSimulation, averageTimePerCus,
                 probabilityOfNewArrival):
        self.probabilityOfNewArrival = probabilityOfNewArrival
        self.lengthOfSimulation = lengthOfSimulation
        self.averageTimePerCus = averageTimePerCus
        self.cashier = Cashier()

    def runSimulation(self):
```

(continua)

(continuação)

```
        """Executa o relógio por n tiques."""
        for currentTime in range(self.lengthOfSimulation):
            # Tenta gerar um novo cliente
            customer = Customer.generateCustomer(
                self.probabilityOfNewArrival,
                currentTime,
                self.averageTimePerCus)
            # Envia o cliente para o caixa se gerado
            # bem-sucedido
            if customer != None:
                self.cashier.addCustomer(customer)
            # Instrui o caixa a fornecer outra unidade de atendimento
            self.cashier.serveCustomers(currentTime)

    def __str__(self):
        return str(self.cashier)
```

Um caixa é responsável por atender uma fila de clientes. Durante esse processo, ele registra os clientes atendidos e os minutos que passam esperando na fila. No final da simulação, o método **__str__** da classe retorna esses totais, bem como o número de clientes restantes na fila. A classe tem as seguintes variáveis de instância:

totalCustomerWaitTime
customersServed
queue
currentCustomer

A última variável contém o cliente que está sendo processado.

Para permitir que o modelo de mercado envie um novo cliente para um caixa, a classe implementa o método **addCustomer**. O método espera um cliente como parâmetro e adiciona o cliente à fila do caixa.

O método **servir os clientes** lida com a atividade do caixa durante um tique do relógio. O método espera a hora atual como parâmetro e responde de uma das várias maneiras, conforme listado na Tabela 8-5.

Eis o pseudocódigo para o método **serveCustomers**:

```
if currentCustomer is None
     if queue is empty
         return
     else
         currentCustomer = queue.pop()
         totalCustomerWaitTime = totalCustomerWaitTime +
             currentTime - currentCustomer.arrivalTime()
         increment customersServed
         currentCustomer.serve()
         if currentCustomer.amountOfServiceNeeded () == 0
              currentCustomer = None
```

(continua)

(continuação)

Condição	O que significa	Ação a ser executada
O cliente atual é **None** e a fila está vazia.	Não há clientes para atender.	Nenhum; apenas retorne.
O cliente atual é **None** e a fila não está vazia.	Há um cliente esperando na frente da fila.	1. Exibir um cliente e torná-lo o cliente atual. 2. Perguntar quando foi instanciado, determinar quanto tempo está esperando e somar esse tempo ao tempo total de espera de todos os clientes. 3. Aumentar o número de clientes atendidos. 4. Atribuir ao cliente uma unidade de atendimento e descartá-la quando concluído.
O cliente atual não é **None**.	Atende o cliente atual.	Atribuir ao cliente uma unidade de atendimento e descartá-la quando concluída.

Tabela 8-5 Respostas de um caixa durante um tique do relógio

Eis o código para a classe **Cashier**:

```
"""
Arquivo: cashier.py
"""
from linkedqueue import LinkedQueue
class Cashier(object):

    def __init__(self):
        self.totalCustomerWaitTime = 0
        self.customersServed = 0
        self.currentCustomer = None
        self.queue = LinkedQueue()

    def addCustomer(self, c):
        self.queue.add(c)

    def serveCustomers(self, currentTime):
        if self.currentCustomer is None:
            # Nenhum cliente ainda
            if self.queue.isEmpty():
                return
            else:
                # Remove o primeiro cliente esperando
                # e conta os resultados do final
                self.currentCustomer = self.queue.pop()
                self.totalCustomerWaitTime += \
                    currentTime - \
                    self.currentCustomer.getArrivalTime()
                self.customersServed += 1
        # Fornece uma unidade de serviço
        self.currentCustomer.serve()
```

(continua)

(continuação)

```
            # Se o cliente atual terminou, manda-o embora
            if self.currentCustomer.getAmountOfServiceNeeded() == 0:
            self.currentCustomer = None

        def __str__(self):
            result = "TOTALS FOR THE CASHIER\n" + \
                     "Número de clientes atendidos:      " + \
                     str(self.customersServed) + "\n"
            if self.customersServed != 0:
                aveWaitTime = self.totalCustomerWaitTime / \
                    self.customersServed
                result += "Number of customers left in queue: " \
                    + str(len(self.queue)) + "\n" + \
                        "Tempo médio que os clientes gastam\n" + \
                        "esperando para ser servido: "
                        + "%5.2f" % aveWaitTime
            return result
```

A classe **Customer** mantém a hora de chegada do cliente e a quantidade de atendimento necessária. O construtor inicializa com os dados fornecidos pelo modelo de mercado. Os métodos de instância incluem:

- **getArrivalTime()** — Retorna a hora em que o cliente chegou à fila do caixa.
- **getAmountOfServiceNeeded()** — Retorna o número de unidades de atendimento restantes.
- **Serve()** — O número de unidades de atendimento é diminuído por um.

O último método, **generateCustomer**, é um *método de classe*. Um método de classe difere de um método de instância, pois é chamado na própria classe, e não em uma instância ou objeto dessa classe. O método **generateCustomer** espera como argumentos a probabilidade de chegada de um novo cliente, a hora atual e o número de unidades de atendimento por cliente. O método retorna uma nova instância de **Customer** com as unidades de tempo e serviço fornecidas, desde que a probabilidade seja maior ou igual a um número aleatório entre 0 e 1. Caso contrário, o método retorna **None**, indicando que nenhum cliente foi gerado. Eis um exemplo de seu uso:

```
>>> Customer.generateCustomer(.6, 50, 4)
>>> Customer.generateCustomer(.6, 50, 4)
<__main__.Customer object at 0x11409e898>
```

Observe que a primeira chamada do método não parece retornar nada, porque a função de fato retornou **None**, o que IDLE não imprime. A segunda chamada retorna um novo objeto **Customer**.

A sintaxe para definir um método de classe no Python é:

```
@classmethod
def <method name>(cls, <other parameters>):
    <instruções>
```

Eis o código da classe **Customer**:

```
"""
Arquivo: customer.py
"""
```

(continua)

(continuação)

```
import random

class Customer(object):

    @classmethod
    def generateCustomer(cls, probabilityOfNewArrival,
                         arrivalTime,
                         averageTimePerCustomer):
        """Retorna um objeto Customer se a probabilidade
        de chegada for maior ou igual a um número aleatório.
        Caso contrário, retorna None, indicando nenhum novo cliente.
        """
        if random.random() <= probabilityOfNewArrival:
            return Customer(arrivalTime, averageTimePerCustomer)
        else:
            return None

    def __init__(self, arrivalTime, serviceNeeded):
        self.arrivalTime = arrivalTime
        self.amountOfServiceNeeded = serviceNeeded

    def getArrivalTime(self):
        return self.arrivalTime

    def getAmountOfServiceNeeded(self):
        return self.amountOfServiceNeeded

    def serve(self):
        """Aceita uma unidade de serviço do caixa."""
        self.amountOfServiceNeeded -= 1
```

Árvores

Uma terceira categoria importante de coleções, que foi chamada **hierárquicas** no Capítulo 2, "Visão geral das coleções", consiste em vários tipos de estruturas em árvore. A maioria das linguagens de programação não inclui árvores como um tipo padrão. Mas as árvores têm usos generalizados. Elas representam coleções de objetos, como uma estrutura de diretório de arquivos ou o índice de um livro, de maneira bem natural. Árvores também podem ser usadas para implementar outras coleções, como conjuntos ordenados e dicionários ordenados, que requerem pesquisa eficiente ou que, como filas de prioridade, devem impor alguma ordem de prioridade aos elementos. Este capítulo examina as propriedades das árvores que as tornam estruturas de dados úteis e explora seu papel na implementação de vários tipos de coleções.

Visão geral das árvores

Nas estruturas de dados lineares que analisamos até agora, todos os itens, exceto o primeiro, têm um predecessor distinto, e todos os itens, exceto o último, têm um sucessor distinto. Em uma árvore, as ideias de predecessor e sucessor são substituídas pelas de um **pai** e um **filho**. As árvores têm dois recursos principais:

- Cada item pode ter vários filhos.
- Todos os itens, exceto um item privilegiado chamado **root** (ou seja, raiz), têm exatamente um pai. A raiz não tem pai.

Termo	Definição
Nó	Um item armazenado em uma árvore.
Raiz	O nó superior em uma árvore. É o único nó sem um pai.
Filho	Um nó logo abaixo e diretamente conectado a determinado nó. Um nó pode ter mais de um filho, e os filhos são vistos organizados da esquerda para a direita. O filho mais à esquerda é chamado primeiro filho e o mais à direita chama-se último filho.
Pai	Um nó logo acima e diretamente conectado a determinado nó. Um nó pode ter apenas um pai.
Irmãos	Os filhos de um pai comum.
Folha	Um nó que não tem filhos.
Nó interno	Um nó que tem pelo menos um filho.
Aresta/Ramo/ Ligação	A linha que conecta um pai ao filho.
Descendente	Os filhos de um nó, os filhos de seus filhos etc. até as folhas.
Antepassado	O pai de um nó, o pai de seu pai e assim por diante até a raiz.
Caminho	A sequência de arestas que conecta um nó e um de seus descendentes.
Comprimento do caminho	O número de arestas em um caminho.

(continua)

(continuação)

Termo	Definição
Profundidade ou nível	A profundidade ou nível de um nó é igual ao comprimento do caminho que o conecta à raiz. Portanto, a profundidade da raiz ou nível da raiz é 0. Seus filhos estão no nível 1 e assim por diante.
Altura	O comprimento do caminho mais longo na árvore; dito de outra forma, o número do nível máximo entre as folhas na árvore.
Subárvore	A árvore formada considerando um nó e todos os seus descendentes.

Tabela 10-1 Um resumo dos termos usados para descrever árvores

Terminologia de árvore

A terminologia de árvore é uma mistura peculiar de termos biológicos, genealógicos e geométricos. A Tabela 10-1 fornece um rápido resumo desses termos. A Figura 10-1 mostra uma árvore e algumas de suas propriedades.

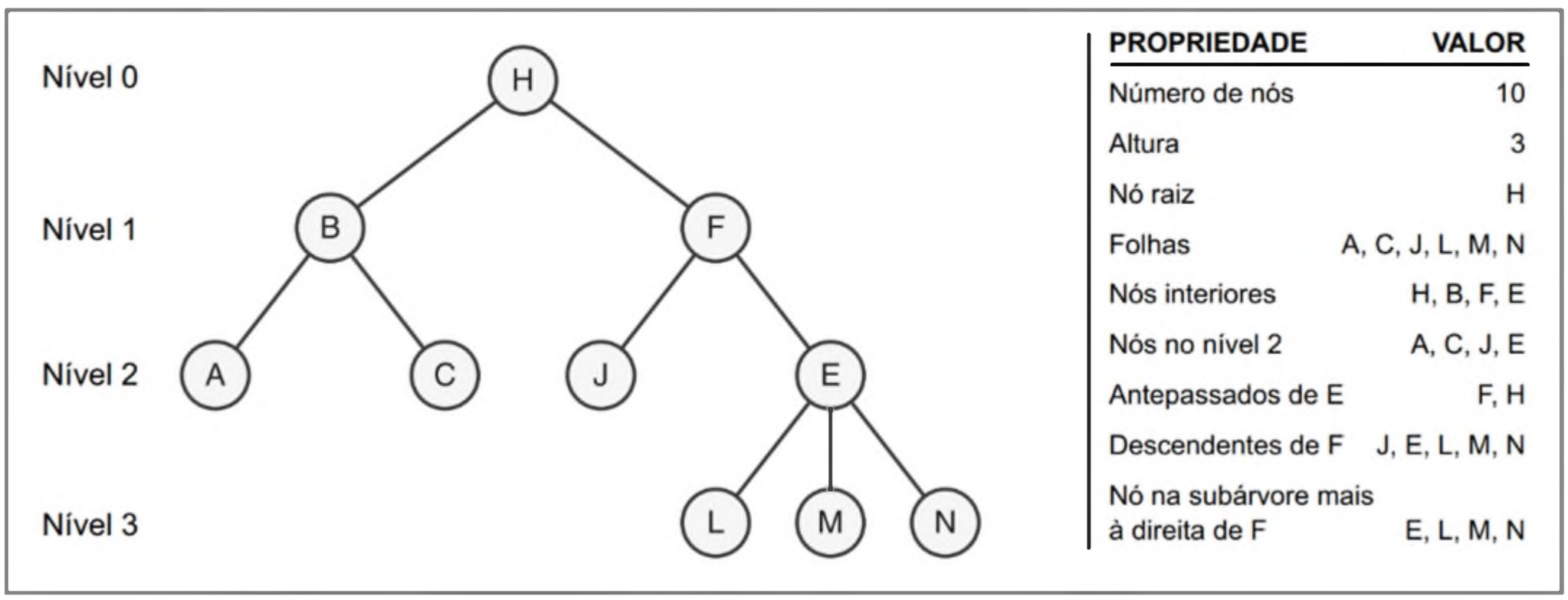


Figura 10-1 Algumas propriedades de uma árvore

Observe que a altura de uma árvore é diferente do número de nós contidos nela. A altura de uma árvore contendo um nó é 0 e, por convenção, a altura de uma árvore vazia é −1.

Árvores gerais e árvores binárias

A árvore mostrada na Figura 10-1 às vezes é chamada de **árvore geral** para distingui-la de uma categoria especial chamada de **árvore binária**. Em uma árvore binária, cada nó tem no máximo dois filhos, chamados de **filho esquerdo** e **filho direito**. Em uma árvore binária, quando um nó tem apenas um filho, você o distingue como filho à esquerda ou à direita. Assim, as duas

árvores mostradas na Figura 10-2 não são as mesmas quando consideradas árvores binárias, embora sejam iguais quando consideradas árvores gerais.

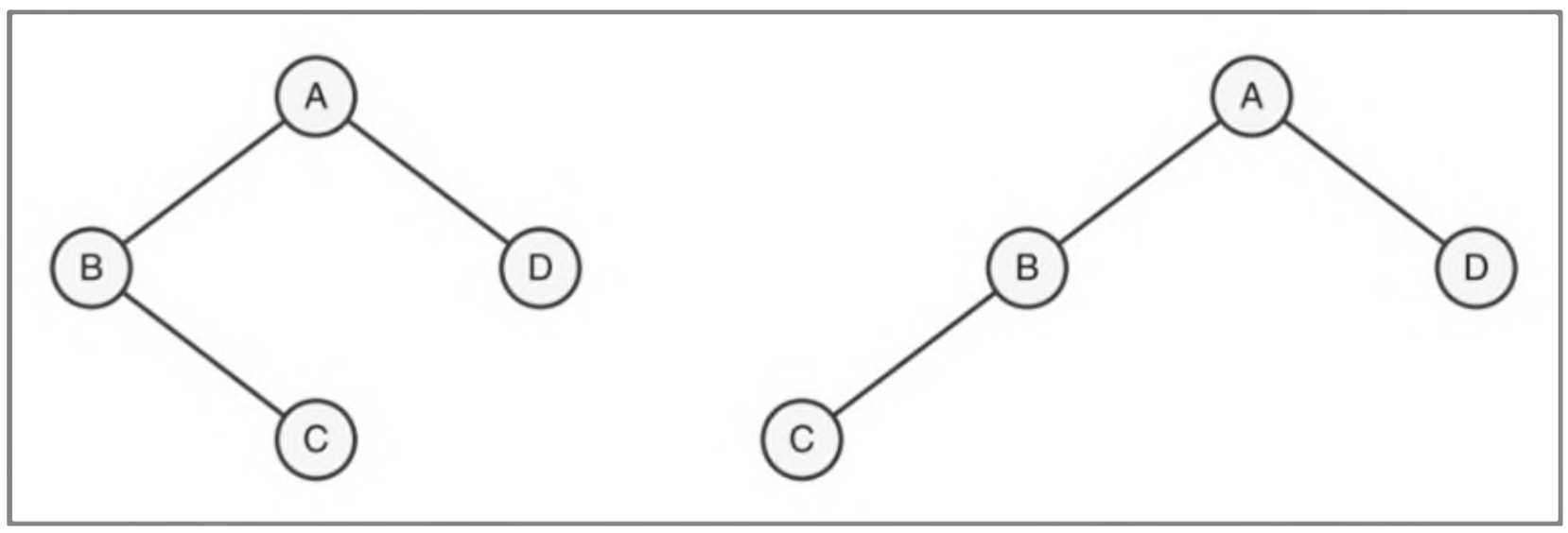


Figura 10-2 Duas árvores binárias desiguais que têm os mesmos conjuntos de nós

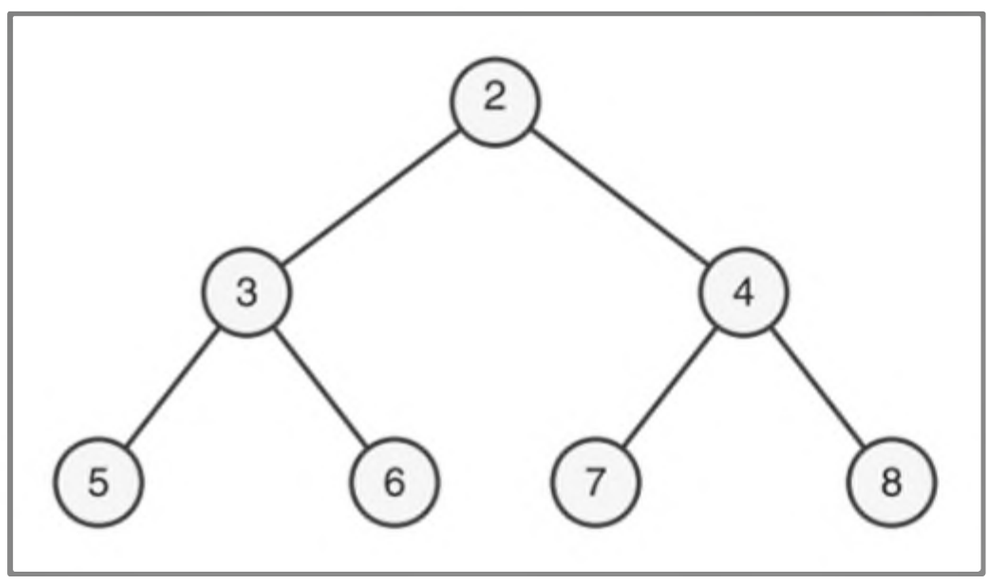
Definições recursivas das árvores

Agora veremos as definições mais formais das árvores gerais e árvores binárias. Como costuma acontecer, não é possível entender a definição formal sem uma compreensão intuitiva do conceito que está sendo definido. Mas a definição formal é importante porque fornece uma base precisa para uma discussão posterior. Além disso, como o processamento recursivo de árvores é comum, aqui estão as definições recursivas dos dois tipos de árvore:

- **Árvore geral** — A árvore geral está vazia ou consiste em um conjunto finito de nós T. Um nó r é distinto de todos os outros e é chamado de raiz. Além disso, o conjunto $T - \{r\}$ é particionado em subconjuntos separados, cada um dos quais é uma árvore geral.
- **Árvore binária** — Uma árvore binária está vazia ou consiste em uma raiz mais uma subárvore esquerda e uma subárvore direita, cada uma das quais é uma árvore binária.

Exercícios

Use a árvore a seguir para responder às próximas seis perguntas.



1. Quais são os nós de folha na árvore?
2. Quais são os nós internos na árvore?
3. Quais são os irmãos do nó 7?
4. Qual é a altura da árvore?
5. Quantos nós estão no nível 2?
6. A árvore é uma árvore geral, uma árvore binária ou ambas?

Por que usar uma árvore?

Conforme mencionado anteriormente, as árvores representam bem as estruturas hierárquicas. Por exemplo, podemos considerar a **análise sintática** de uma frase específica em um idioma. A **árvore de análise** descreve a estrutura sintática de uma frase em termos de suas partes componentes, como sintagmas nominais e sintagmas verbais. A Figura 10-3 mostra a árvore de análise para a seguinte frase: "A menina rebateu a bola com um taco".

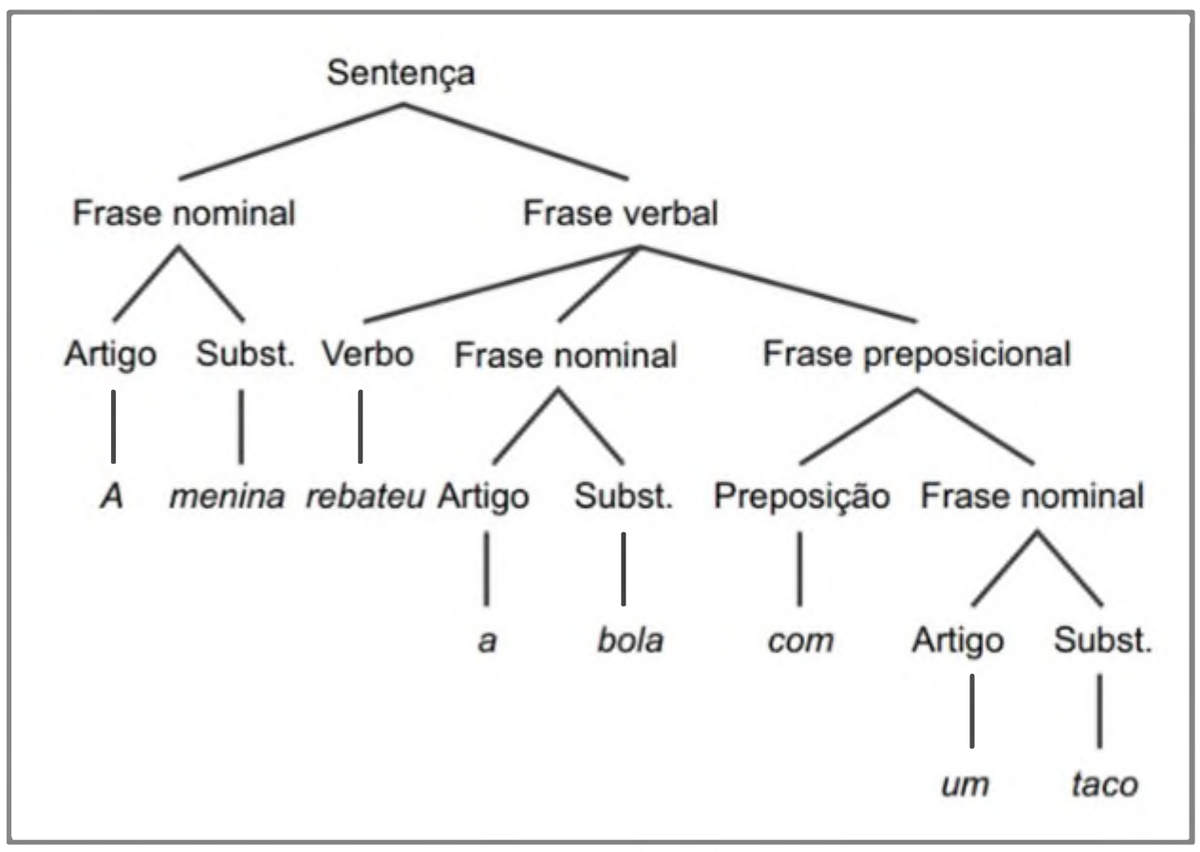


Figura 10-3 Uma árvore de análise para uma frase

O nó raiz dessa árvore, denominado "Sentença", representa a frase de nível superior nessa estrutura. Seus dois filhos, rotulados "Frase nominal" e "Frase verbal", representam as frases constituintes dessa frase. O nó rotulado "Frase preposicional" é filho de "Frase verbal", que indica que a frase preposicional "com um taco" modifica o verbo "rebater" em vez do sintagma nominal "a bola". No nível inferior, os nós de folha, como "bola", representam as palavras dentro das frases.

Como veremos mais adiante neste capítulo, programas de computador podem construir árvores de análise durante a análise de expressões aritméticas. Você pode então usar essas árvores

para processamento posterior, como verificar nas expressões erros gramaticais e interpretá-las por seus significados ou valores.

As estruturas do sistema de arquivos também são semelhantes a árvores. A Figura 10-4 mostra uma estrutura, na qual os diretórios (agora comumente conhecidos como pastas) são rotulados "D" e os arquivos são rotulados "F".

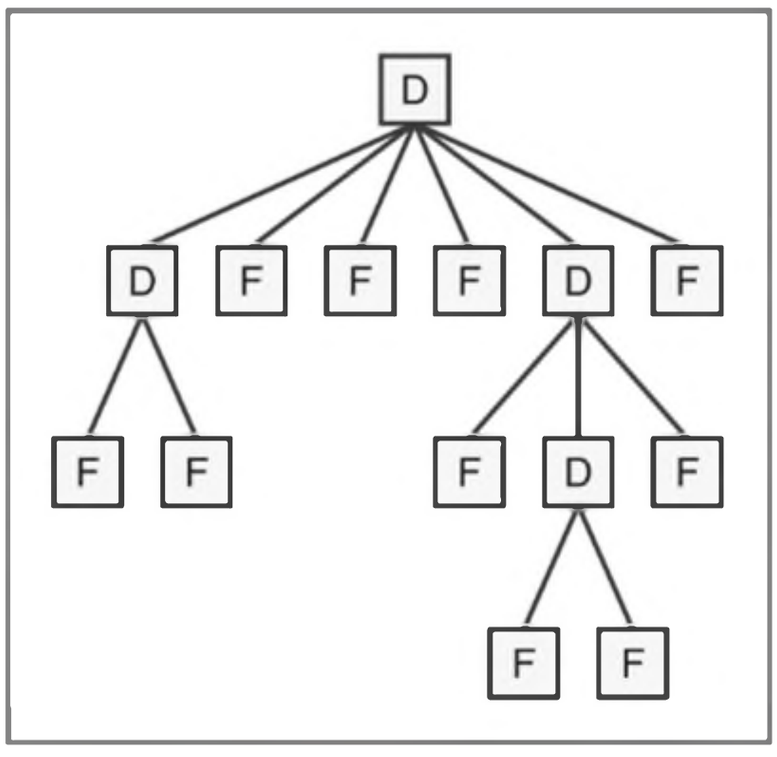


Figura 10-4 Uma estrutura de sistema de arquivos

Observe que o nó raiz representa o diretório raiz. Os outros diretórios são nós internos quando não estão vazios ou folhas quando estão vazios. Todos os arquivos são folhas.

Algumas coleções ordenadas também podem ser representadas como estruturas semelhantes a árvores. Esse tipo de árvore é chamado de **árvore binária de pesquisa**, ou BST (do inglês Binary Search Tree). Cada nó na subárvore à esquerda de determinado nó é menor do que esse nó e cada nó na subárvore à direita de determinado nó é maior do que esse nó. A Figura 10-5 mostra uma representação da árvore binária de pesquisa de uma coleção ordenada que contém as letras *A* a *G*.

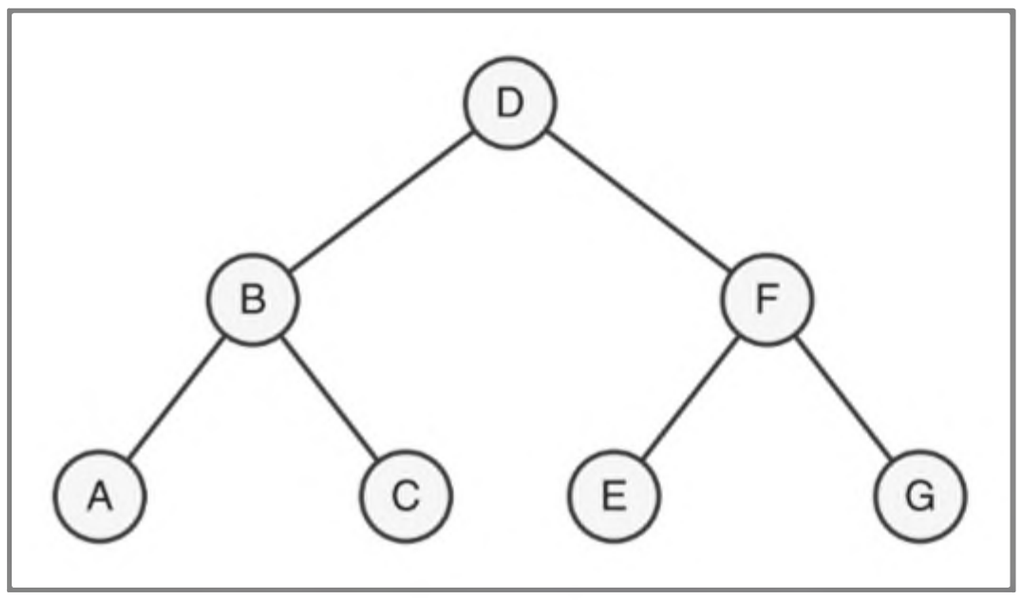


Figura 10-5 Uma coleção ordenada como uma árvore binária de pesquisa

Ao contrário da sacola ordenada discutida no Capítulo 6, "Herança e classes abstratas", uma árvore binária de pesquisa pode suportar não apenas pesquisas logarítmicas, mas inserções e remoções logarítmicas.

Esses três exemplos mostram que a característica mais importante e útil de uma árvore não são as posições de seus itens, mas os relacionamentos entre pais e filhos. Esses relacionamentos são essenciais para o significado dos dados da estrutura. Eles podem indicar ordem alfabética, estrutura de frase, contenção em um subdiretório ou qualquer relacionamento de um-para-muitos em determinado domínio de problema. O processamento dos dados dentro das árvores é baseado nos relacionamentos pai/filho entre os dados.

As seções a seguir focalizam diferentes tipos, aplicações e implementações das árvores binárias.

A forma das árvores binárias

Árvores na natureza têm várias formas e tamanhos e árvores como estruturas de dados têm várias formas e tamanhos. Falando informalmente, algumas árvores são parecidas com videiras e têm forma quase linear, enquanto outras são espessas. Os dois extremos dessas formas são mostrados na Figura 10-6.

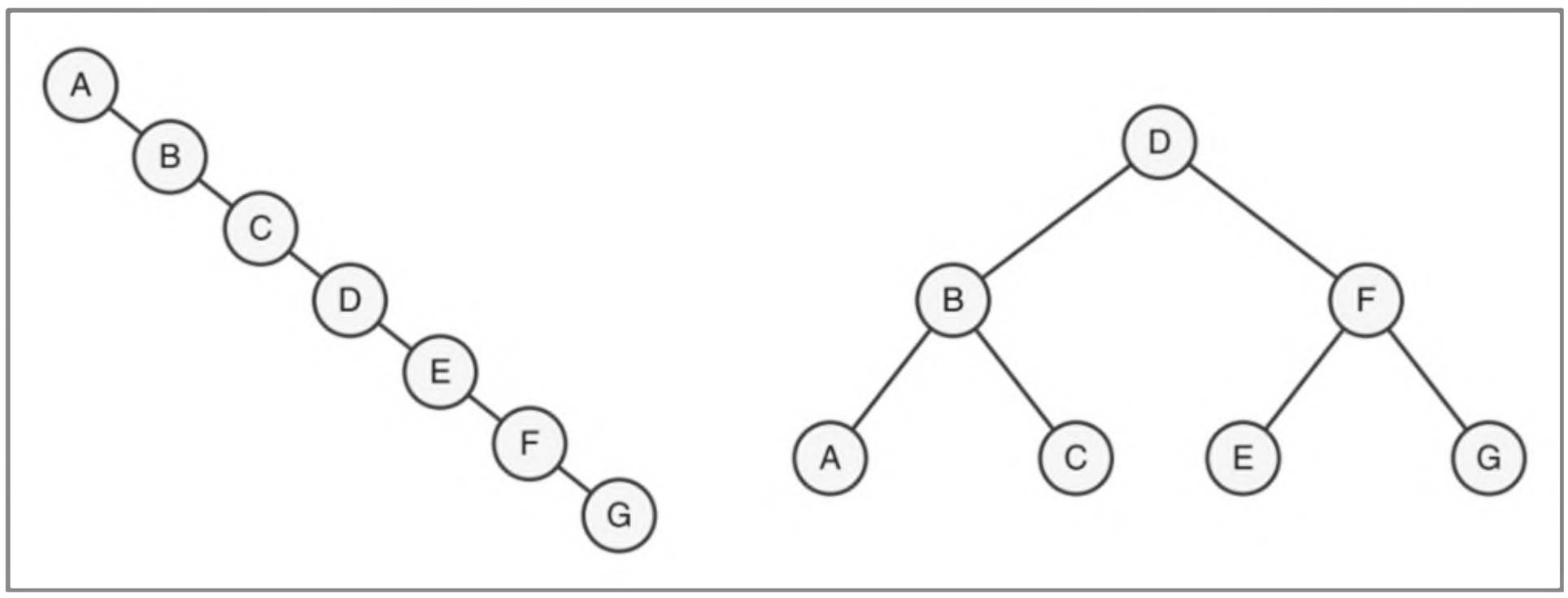


Figura 10-6 Uma árvore tipo videira e uma tipo arbusto

A forma de uma árvore binária pode ser descrita mais formalmente especificando o relacionamento entre a altura e o número de nós que ela contém. Esse relacionamento também fornece informações sobre a potencial eficiência de algumas operações na árvore.

Em um extremo, uma árvore binária pode ser semelhante a uma videira, com N nós e uma altura de $N - 1$. (Ver o lado esquerdo da Figura 10-6.) Essa árvore lembra uma cadeia linear de nós em uma lista ligada.

Um acesso, uma inserção ou uma remoção de um nó nessa estrutura seria, portanto, linear no pior dos casos.

No outro extremo, considere um **árvore binária cheia** (*full binary tree*) que contém o número máximo de nós para determinada altura *H*. (Ver o lado direito de Figura 10-6.) Uma árvore que apresenta essa forma contém o complemento total dos nós em cada nível. Todos os nós internos têm dois filhos e todas as folhas estão no nível mais baixo. Tabela 10-2 lista a altura e o número de nós para árvores binárias cheias de quatro alturas.

Altura da árvore	Número de nós na árvore
0	1
1	3
2	7
3	15

Tabela 10-2 O relacionamento entre a altura e o número de nós na árvore binária cheia

Vamos generalizar a partir dessa tabela. Qual é o número de nós, *N*, contido em uma árvore binária cheia de altura *H*? Para expressar *N* em termos de *H*, você começa com a raiz (1 nó), adiciona os filhos (2 nós), adiciona os filhos (4 nós) e assim por diante, da seguinte maneira:

$$N = 1 + 2 + 4 + \cdots + 2^{H}$$
$$= 2^{H+1} - 1$$

E qual é a altura, *H*, de uma árvore binária cheia com *N* nós? Usando álgebra simples, obtemos

$$H = \log_2(N+1) - 1$$

Como o número de nós em determinado caminho da raiz até uma folha está próximo de $\log_2(N)$, a quantidade máxima de trabalho necessária para acessar determinado nó em uma árvore binária cheia é O(log *N*).

Nem todas as árvores espessas são árvores binárias cheias. Mas a **árvore binária perfeitamente balanceada**, que inclui um complemento completo dos nós em cada nível, exceto o último, é espessa o suficiente para suportar o acesso logarítmico no pior dos casos aos nós folha. A **árvore binária completa** (*complete binary tree*), em que quaisquer nós no último nível são preenchidos da esquerda para a direita é, como uma árvore binária cheia, um caso especial de árvore binária perfeitamente balanceada. A Figura 10-7 resume esses tipos das formas das árvores binárias com alguns exemplos.

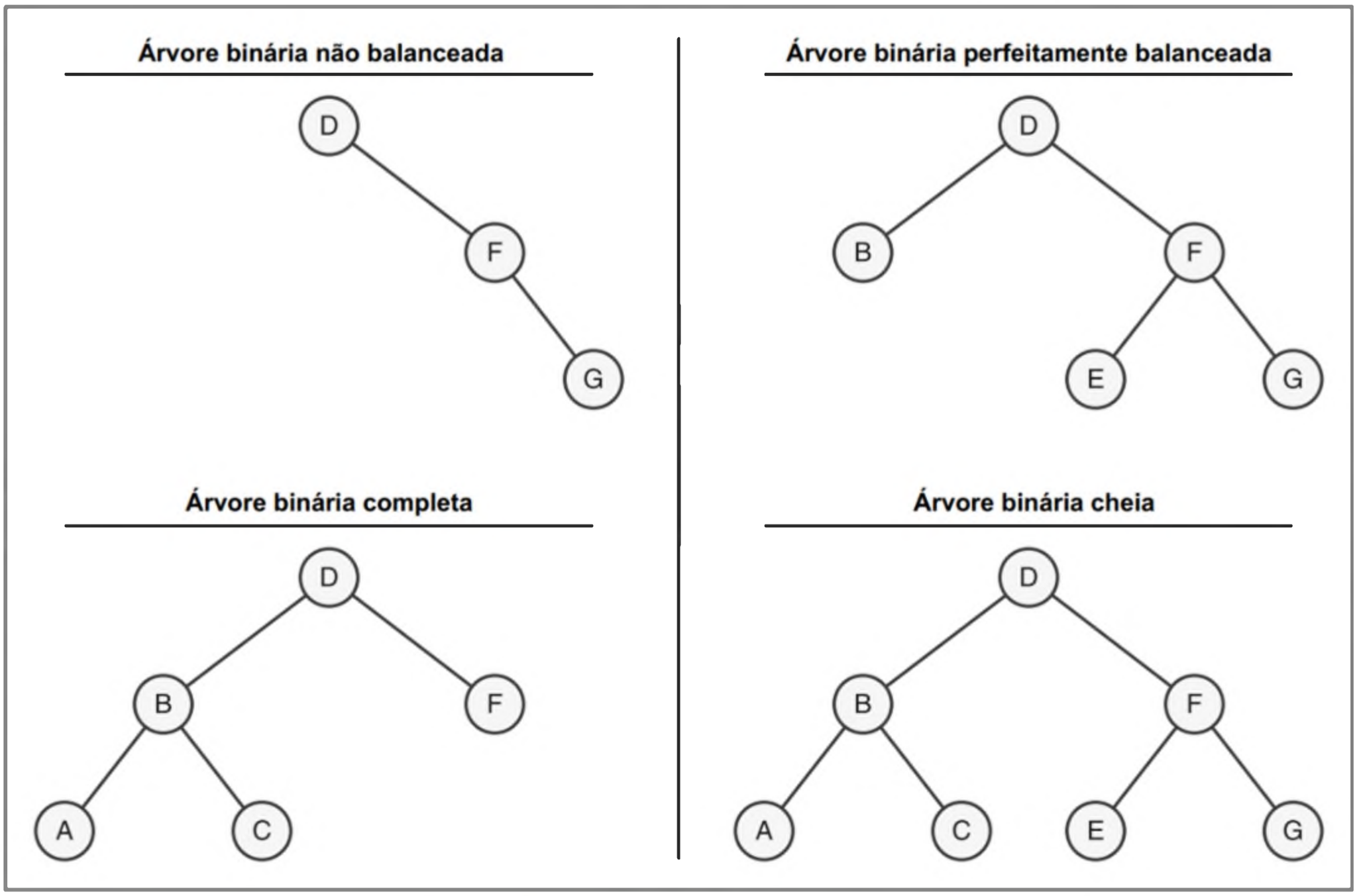


Figura 10-7 Quatro tipos de formas de árvores binárias

De modo geral, à medida que uma árvore binária se torna mais balanceada, o desempenho de acessos, inserções e remoções melhora.

Exercícios

1. Qual é a diferença entre uma árvore binária perfeitamente balanceada e uma árvore binária completa?
2. Qual é a diferença entre uma árvore binária completa e uma árvore binária cheia?
3. Uma árvore binária cheia tem uma altura de 5. Quantos nós ela contém?
4. Uma árvore binária completa contém 125 nós. Qual é a altura?
5. Quantos nós estão em determinado nível *L* em uma árvore binária cheia? Expresse sua resposta em termos de *L*.

Percursos em uma árvore binária

Nos capítulos anteriores, vimos como percorrer os itens em coleções lineares usando um laço `for` ou um iterador. Existem quatro tipos padrão de percursos para árvores binárias: pré-ordem, pós-ordem, in-ordem e ordem de níveis. Cada tipo de percurso segue um caminho e direção específicos à medida que visita os nós da árvore. Esta seção mostra diagramas de cada tipo de percurso em árvores binárias de pesquisa; algoritmos para os percursos são desenvolvidos mais adiante neste capítulo.

Percurso em pré-ordem

O algoritmo do **percurso em pré-ordem** visita o nó raiz de uma árvore e, em seguida, percorre a subárvore à esquerda e a subárvore à direita de maneira semelhante. A sequência dos nós visitados por um percurso em pré-ordem é ilustrada na Figura 10-8.

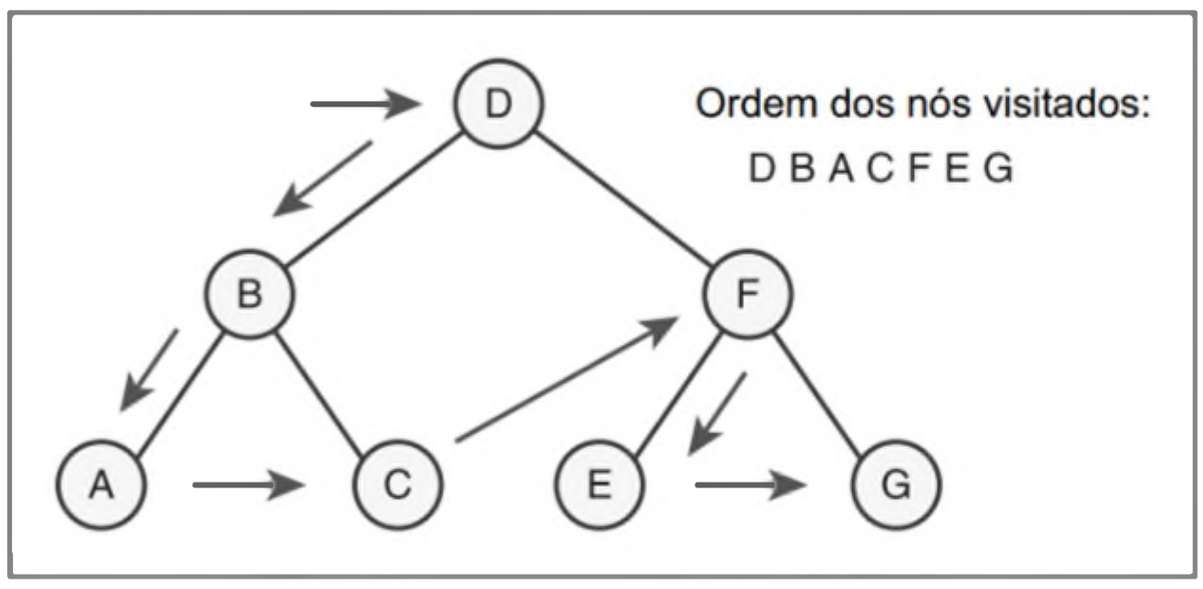


Figura 10-8 Um percurso em pré-ordem

Percurso em in-ordem

O algoritmo do **percurso em in-ordem** percorre a subárvore esquerda, visita o nó raiz e percorre a subárvore direita. Esse processo se move o mais para a esquerda possível na árvore antes de visitar um nó. A sequência dos nós visitados por um percurso em in-ordem é ilustrada na Figura 10-9.

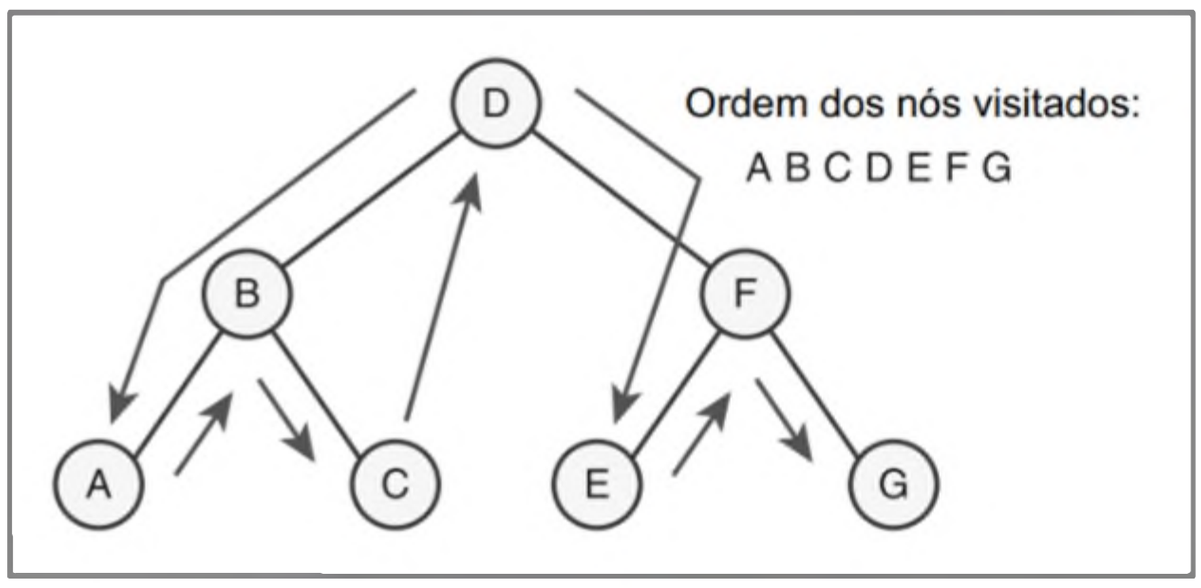


Figura 10-9 Um percurso em in-ordem

Percurso em pós-ordem

O algoritmo do **percurso em pós-ordem** percorre a subárvore esquerda, percorre a subárvore direita e visita o nó raiz. O caminho percorrido por um percurso em pós-ordem é ilustrado na Figura 10-10.

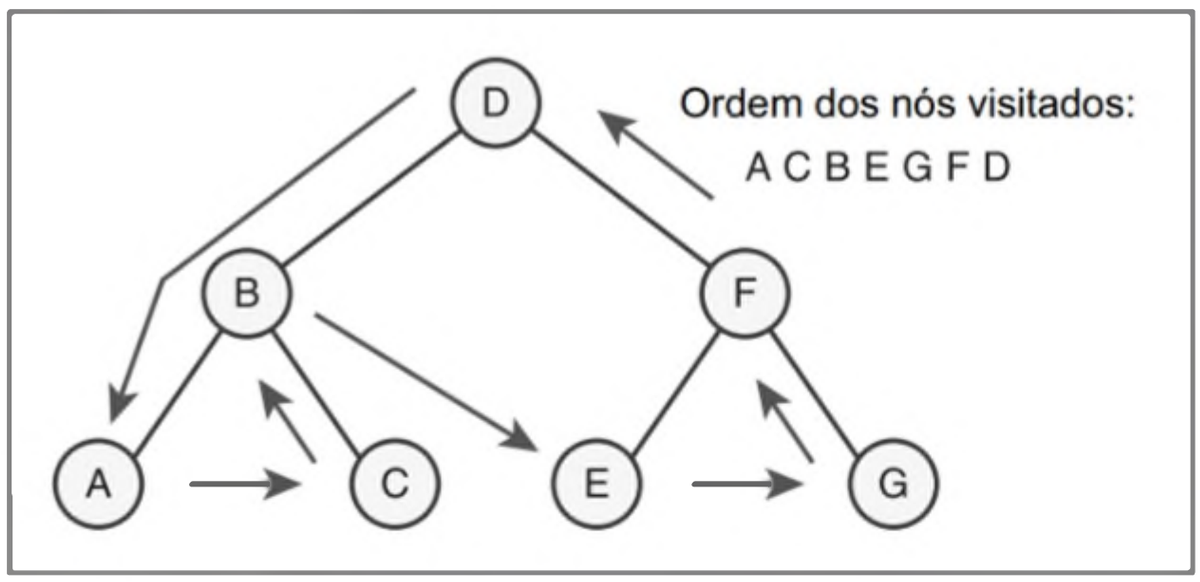


Figura 10-10 Um percurso em pós-ordem

Percurso em ordem de níveis

Começando com o nível 0, o algoritmo do **percurso em ordem de níveis** visita os nós em cada nível na ordem da esquerda para a direita. O caminho percorrido por um percurso em ordem de níveis é ilustrado na Figura 10-15.

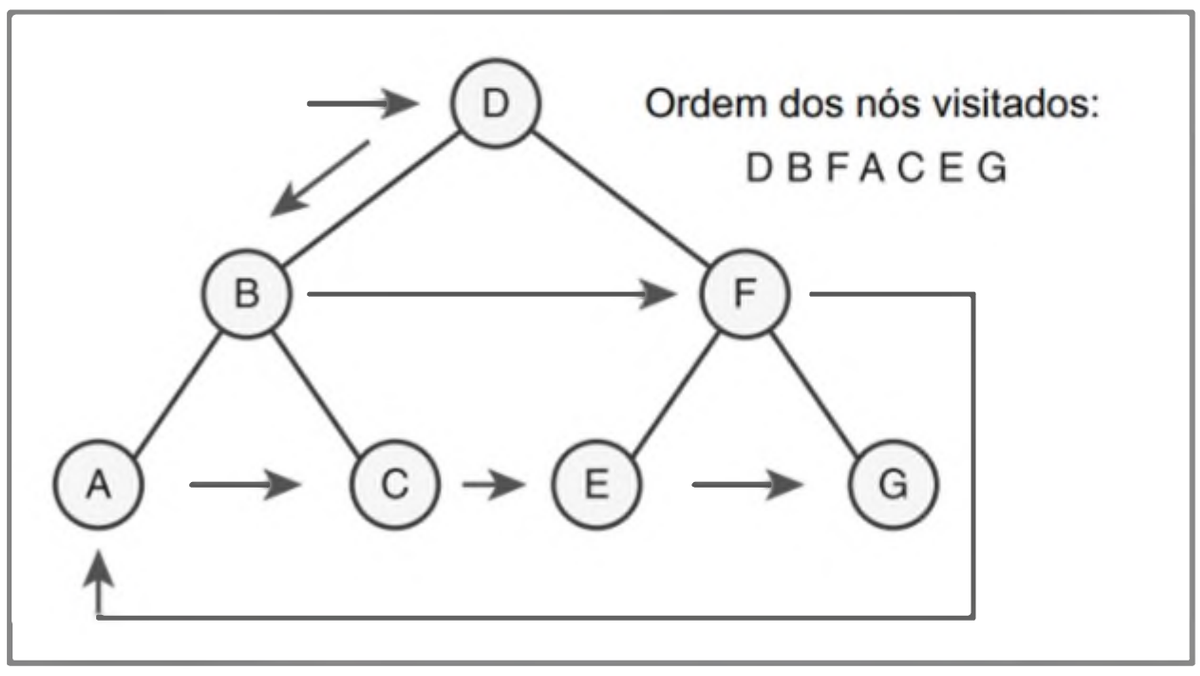


Figura 10-11 Um percurso em ordem de níveis

Como podemos ver, um percurso em in-ordem é apropriado para visitar os itens em uma árvore binária de pesquisa de forma ordenada. Os percursos em pré-ordem, in-ordem e pós-ordem das árvores de expressão podem ser usados para gerar as representações de prefixo, infixo e pós-fixo das expressões, respectivamente.

Três aplicações comuns das árvores binárias