Capítulo 1

Introdução

Todo currículo de ciência da computação no mundo inclui um curso sobre estruturas de dados e algoritmos. Estruturas de dados são importantes; elas melhoram a nossa qualidade de vida e até mesmo podem salvar vidas rotineiramente. Muitas empresas multi-bilionárias foram construídas em torno de estruturas de dados.

Como isso acontece? Se paramos para pensar sobre isso, percebemos que interagimos constantemente com as estruturas de dados.

- Abrir um arquivo: As estruturas de dados do sistema de arquivos são usadas para localizar as partes desse arquivo no disco para que possam ser recuperadas. Isso não é fácil: discos contêm centenas de milhões de blocos. O conteúdo do seu arquivo pode ser armazenado em qualquer um deles.

- Procurar um contato em seu telefone: uma estrutura de dados é usada para procurar um número de telefone em sua lista de contatos com base em informações parciais mesmo antes de terminar de discagem/digitação. Isso não é fácil: seu telefone pode conter informações sobre um grande número de pessoas — todos os que você já contactou por telefone ou e-mail — e seu telefone não tem um processador muito rápido ou muita memória.

- Fazer login na sua rede social favorita: os servidores de rede usam suas informações de login para procurar as informações de sua conta. Isso não é fácil: as redes sociais mais populares têm centenas de milhões de usuários ativos.

- Fazer uma pesquisa na web: O mecanismo de pesquisa usa estruturas de dados para encontrar as páginas da web que contêm seus termos de pesquisa. Isso não é fácil: há mais de 8,5 bilhões de páginas na Internet e cada página contém muitos termos de pesquisa em potencial.

- Telefone de serviços de emergência (1-9-0): A rede de serviços de emergência procura o seu número de telefone em uma estrutura de dados que mapeia números de telefone para endereços para que carros de polícia, ambulâncias ou caminhões de bombeiros possam ser enviados para lá sem demora. Isso é importante: a pessoa que faz a chamada pode não ser capaz de fornecer o endereço exato que eles estão chamando e um atraso pode significar a diferença entre a vida ou a morte.

1.1 A Necessidade de Eficiência

Na próxima seção, analisamos as operações suportadas pelas estruturas de dados mais usadas. Qualquer pessoa com um pouco de experiência de programação verá que essas operações não são difíceis de implementar corretamente. Podemos armazenar os dados em uma matriz ou uma lista vinculada e cada operação pode ser implementada iterando sobre todos os elementos da matriz ou lista e possivelmente adicionando ou removendo um elemento.

Este tipo de implementação é fácil, mas não muito eficiente. Mas isso realmente importa? Os computadores estão se tornando cada vez mais rápidos. Talvez a implementação óbvia seja boa o suficiente. Vamos fazer alguns cálculos aproximados para descobrir.

Número de operações: Imagine um aplicativo com um conjunto de dados de tamanho moderado, digamos de um milhão (10^6) de itens. É razoável, na maioria das aplicações, assumir que o aplicativo vai procurar cada item pelo menos uma vez. Isso significa que podemos esperar fazer pelo menos um milhão (10^6) de pesquisas nesses dados. Se cada uma dessas 10^6 inspeções inspecionar cada um dos 10^6 itens, isto dá um total de

$10^6 \times 10^6 = 10^{12}$ (um trilhão) de inspeções.

Velocidade do processador: No momento da escrita deste texto, mesmo um computador desktop muito rápido não pode fazer mais de um bilhão (10^9) de operações por segundo.[1] Isto significa que esta aplicação tomará pelo menos $10^{12}/10^9 = 1000$ segundos, ou cerca de 16 minutos e 40 segundos. Dezesseis minutos é uma eternidade no tempo do computador, mas uma pessoa pode estar disposta a aturar isso (Se ele ou ela saiu para uma pausa para o café).

Grandes conjuntos de dados: Agora considere uma empresa como o Google, que indexa mais de 8,5 bilhões de páginas da web. Pelo nossos cálculos, fazer qualquer tipo de consulta sobre esses dados levaria pelo menos 8,5 segundos. Já sabemos que não é esse o caso; pesquisas na web concluem em menos de 8,5 segundos, e fazemos consultas muito mais complicadas do que apenas perguntar se uma determinada página está em sua lista de páginas indexadas. Atualmente, o Google recebe aproximadamente 4.500 consultas por segundo, o que significa que elas exigem pelo menos $4.500 \times 8.5 = 38.250$ servidores muito rápidos apenas para manter-se.

A solução: Esses exemplos nos dizem que as implementações óbvias de estruturas de dados não se dimensionam bem quando o número de itens, n, na estrutura de dados e o número de operações, m, realizados na estrutura de dados são ambos grandes. Nestes casos, o tempo (medido em, digamos, instruções de máquina) é aproximadamente $n \times m$.

A solução, é claro, é organizar cuidadosamente os dados dentro da estrutura de dados para que nem todas as operações exijam que todos os itens de dados sejam inspecionados. Embora pareça impossível no início, veremos estruturas de dados onde uma pesquisa requer apenas dois itens em média, independentemente do número de itens armazenados na estrutura de dados. Em nosso computador de um bilhão de instruções por segundo, levamos apenas 0.000000002 segundos para pesquisar em uma

[1] As velocidades do computador são, no máximo, de alguns gigahertz (bilhões de ciclos por segundo), e cada operação tipicamente leva alguns ciclos.

estrutura de dados contendo um bilhão de itens (ou um trilhão, ou um quatrilhão, ou até mesmo um quintilhão de itens).

Também veremos implementações de estruturas de dados que mantêm os itens em uma ordem específica, na qual o número de itens inspecionados durante uma operação cresce muito lentamente em função do número de itens na estrutura de dados. Por exemplo, podemos manter um conjunto ordenado de um bilhão de itens enquanto inspecionamos no máximo 60 itens durante qualquer operação. Em nosso computador de um bilhão de instruções por segundo, essas operações levam 0.00000006 segundos cada.

O restante deste capítulo revisa brevemente alguns dos principais conceitos utilizados ao longo do restante do livro. A Seção 1.2 descreve as interfaces implementadas por todas as estruturas de dados descritas neste manual e deve ser considerada leitura obrigatória. As seções restantes discutem:

- Alguma revisão matemática incluindo exponenciais, logaritmos, fatoriais, notação assintótica (big-O, às vezes usada no Brasil como grande-O ou O-grande), probabilidade e randomização;
- o modelo de computação;
- correção, tempo de execução e espaço;
- uma visão geral do resto dos capítulos; e
- o código de exemplo e as convenções de escrita.

Um leitor com ou sem um conhecimento nessas áreas pode facilmente ignorá-las agora e voltar a elas mais tarde, se necessário.

1.2 Interfaces

Ao discutir estruturas de dados, é importante entender a diferença entre a interface de uma estrutura de dados e sua implementação. Uma interface descreve o que uma estrutura de dados faz, enquanto uma implementação descreve como a estrutura de dados o faz.

Uma *interface*, às vezes também chamada de *tipo abstrato de dados*, define o conjunto de operações suportado por uma estrutura de dados e a semântica, ou significado, dessas operações. Uma interface não nos diz nada sobre como a estrutura de dados implementa essas operações; ela fornece somente uma lista de operações suportadas junto com especificações sobre quais tipos de argumentos cada operação aceita e o valor retornado por cada operação.

Uma *implementação* de estrutura de dados, por outro lado, inclui a representação interna da estrutura de dados, bem como as definições dos algoritmos que implementam as operações suportadas pela estrutura de dados. Assim, pode haver muitas implementações de uma única interface. Por exemplo, no Capítulo 2, veremos implementações da interface Lista usando arrays e no Capítulo 3 veremos implementações da interface Lista usando estruturas de dados baseadas em ponteiro. Cada uma implementa a mesma interface, Lista, mas de maneiras diferentes.

1.2.1 As Interfaces Fila (Queue), Pilha (Stack) e Deque

A interface de Fila representa uma coleção de elementos aos quais podemos adicionar elementos e remover o próximo elemento. Mais precisamente, as operações suportadas pela interface Fila são

- add(x): enfileira o valor x na Fila
- remove(): remove o próximo (enfileirado anteriormente) valor, y, da Fila e retorna y

Observe que a operação remove() não assume nenhum argumento. A disciplina de enfileiramento da Fila decide qual elemento deve ser removido. Existem muitas disciplinas de filas possíveis, a mais comum incluem FIFO, prioridade e LIFO.

Uma Fila *FIFO (first-in-first-out)*, ilustrada na Figura 1.1, remove os itens na mesma ordem em que foram adicionados, da mesma forma que uma fila de uma caixa de supermercado. Este é o tipo mais comum de Fila, de modo que o qualificador FIFO é muitas vezes omitido. Em outros textos, as operações add(x) e remove() em uma fila FIFO são frequentemente chamadas de enqueue(x) e dequeue(), respectivamente.

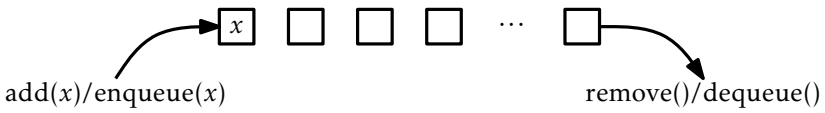

Figura 1.1: Uma Fila FIFO.

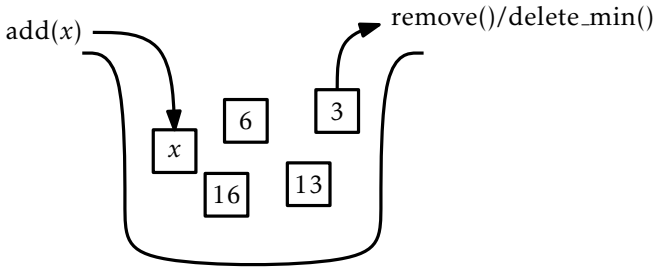

Figura 1.2: Uma Fila de Prioridade.

Uma *Fila de prioridade*, ilustrada na Figura 1.2, sempre remove o menor elemento da Fila, quebrando os laços arbitrariamente. Isto é semelhante à maneira pela qual é feita a triagem de pacientes em uma sala de emergência do hospital. À medida que os pacientes chegam, eles são avaliados e depois colocados em uma sala de espera. Quando um médico se torna disponível, ele ou ela primeiro trata o paciente com a condição mais fatal. A operação remove() em uma Fila de Prioridade é normalmente chamada de delete_min() em outros textos.

Uma disciplina de enfileiramento muito comum é a disciplina LIFO (last-in-first-out), ilustrada na Figura 1.3. Em uma *Fila LIFO*, o elemento adicionado mais recentemente é o próximo removido. Isto é melhor visualizado em termos de uma pilha de pratos. Os pratos são colocados no topo da pilha e também removidos da parte superior da pilha. Essa estrutura é tão comum que ele recebe seu próprio nome: Stack. Muitas vezes, ao discutir uma Stack, os nomes de add(x) e remove() são alterados para push(x) e pop(); isso é para evitar confundir as disciplinas das filas LIFO e FIFO.

Uma Deque é uma generalização de ambas Filas FIFO e LIFO (Stack). Uma Deque representa uma sequência de elementos, com uma frente e uma parte traseira. Os elementos podem ser adicionados na frente da

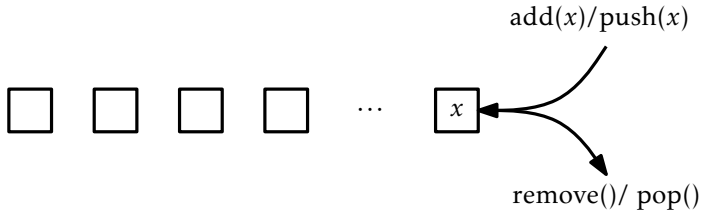

Figura 1.3: Uma Pilha.

sequência ou na parte de trás da sequência. Os nomes das operações de Deque são auto-explicativos: add_first(x),remove_first(), add_last(x) e remove_last(). Vale a pena notar que uma Pilha pode ser implementada usando apenas add_last(x) e remove_last(), enquanto uma fila FIFO pode ser implementada usando add_last(x) e remove_first().

1.2.2 A interface de Lista: sequências lineares

Este livro falará muito pouco sobre as interfaces Fila FIFO, Pilha ou Deque. Isso ocorre porque essas interfaces são um subconjunto da interface Lista. Uma Lista, ilustrada na Figura 1.4, representa uma sequência, $x_0,\ldots,x_{n-1}$ de valores. A interface Lista inclui as seguintes operações:

1. size(): retorna n, o tamanho da lista

2. get(i): retorna o valor x_i

3. set(i,x): faz o valor de x_i igual a x

4. add(i,x): enfileira x na posição i, deslocando $x_i,\ldots,x_{n-1}$;
   Faz $x_{j+1} = x_j$, para todo $j \in \{n-1,\ldots,i\}$, incrementa n, e faz $x_i = x$

5. remove(i) remove o valor x_i, deslocando $x_{i+1},\ldots,x_{n-1}$;
   Faz $x_j = x_{j+1}$, para todo $j \in \{i,\ldots,n-2\}$ e decrementa n

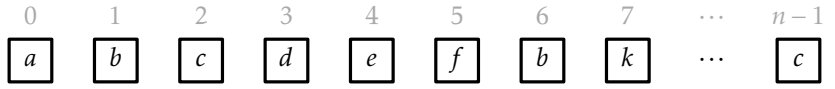

Figura 1.4: Uma Lista representa uma sequência indexada por $0, 1, 2, \ldots, n-1$. Nesta Lista, uma chamada para get(2) deve retornar o valor c.

Observe que essas operações são mais que suficientes para implementar a interface Deque:

$$\begin{aligned} \text{add\_first}(x) &\Rightarrow \text{add}(0, x) \\ \text{remove\_first}() &\Rightarrow \text{remove}(0) \\ \text{add\_last}(x) &\Rightarrow \text{add}(\text{size}(), x) \\ \text{remove\_last}() &\Rightarrow \text{remove}(\text{size}() - 1) \end{aligned}$$

Embora normalmente não discutamos as interfaces Pilha, Deque e Fila FIFO nos capítulos subsequentes, os termos Pilha e Deque às vezes são usados nos nomes das estruturas de dados que implementam a interface Lista. Quando isso acontece, destaca o fato de que essas estruturas de dados podem ser usadas para implementar a interface Pilha ou Deque de forma muito eficiente. Por exemplo, a classe ArrayDeque é uma implementação da interface Lista que implementa todas as operações Deque em tempo constante por operação.

1.2.3 A interface USet: conjuntos não ordenados

A interface USet representa um conjunto não ordenado de elementos únicos, que imita a operação matemática *set*. Uma USet contém n elementos *distintos*; nenhum elemento aparece mais de uma vez; eles não estão em nenhuma ordem específica. Uma USet suporta as seguintes operações:

1. size(): retorna o número, n, de elementos no conjunto;

2. add(x): acrescenta o elemento x ao conjunto se ele ainda não estiver presente.
   Acrescenta x ao conjunto desde que não exista nenhum elemento y no conjunto de tal modo que x seja igual a y. Retorna *true* se x foi acrescentado ao conjunto e *false* caso contrário.

3. remove(x): remove x do conjunto;
   Encontra um elemento y no conjunto de modo que x seja igual a y e remove y. Retorna y, ou *nil* se tal elemento não existe.

4. find(x): encontra x no conjunto se ele existe;
   Encontra um elemento y no conjunto de modo que y seja igual a x. Retorna y, ou *nil* se tal elemento não existe.

Essas definições são um pouco exigentes em distinguir x, o elemento que estamos removendo ou encontrando, de y, o elemento que podemos remover ou encontrar. Isso ocorre porque x e y podem realmente ser objetos distintos que são tratados como iguais. Tal distinção é útil porque permite a criação de *dicionários* ou *mapas* que mapeiam chaves em valores.

Para criar um dicionário/mapa, formamos objetos compostos chamados Pares, cada um dos quais contém uma *chave* e um *valor*. Dois Pares são tratados como iguais se suas chaves são iguais. Se armazenarmos algum par (k, v) em uma USet e depois chamamos o método find(x) usando o par $x = (k, nil)$, o resultado será $y = (k, v)$. Em outras palavras, é possível recuperar o valor, v, dado apenas a chave, k.

1.2.4 A interface SSet: conjuntos ordenados

A interface SSet representa um conjunto ordenado de elementos. Uma SSet armazena elementos com algum ordenamento geral, de modo que quaisquer dois elementos x e y podem ser comparados. Nos exemplos de código, isso será feito com um método chamado compare(x, y), no qual

$$\text{compare}(x,y)\begin{cases} < 0 & \text{se } x < y \\ > 0 & \text{se } x > y \\ = 0 & \text{se } x = y \end{cases}$$

Uma SSet suporta os métodos size(), add(x) e remove(x) com a mesma semântica da interface USet. A diferença entre uma USet e uma SSet é o método find(x):

4. find(x): localiza x no conjunto ordenado;
   Encontra o menor elemento y no conjunto de modo que $y \geq x$. Retorna y ou *nil* se tal elemento não existir.

Esta versão da operação find(x) é algumas vezes referida como uma *busca do sucessor*. Ela difere de uma maneira fundamental de USet.find(x), uma vez que retorna um resultado significativo, mesmo quando não há nenhum elemento igual a x no conjunto.

A distinção entre as operações find(x) em USet e SSet é muito importante e muitas vezes não é atendida. A funcionalidade adicional fornecida por um SSet geralmente vem com um preço que inclui tanto um maior tempo de execução e uma maior complexidade de implementação. Por exemplo, na maioria das implementações SSet discutidas neste livro, todas as operações find(x) possuem tempos de execução que são logarítmicos de acordo com o tamanho do conjunto. Por outro lado, a implementação de um USet como um ChainedHashTable no Capítulo 5 tem uma operação find(x) que é executada em tempo esperado constante. Ao escolher qual dessas estruturas usar, deve-se sempre usar um USet a menos que a funcionalidade adicional oferecida por um SSet seja realmente necessário.

1.3 Base Matemática

Nesta seção, revisamos algumas notações matemáticas e ferramentas usadas ao longo deste livro, incluindo logaritmos, notação Big-O e teoria de probabilidade. Esta revisão será breve e não pretende ser uma introdução. Os leitores que sentem dificuldades com essas bases são encorajados a ler e fazer exercícios a partir das seções apropriadas do livro excelente (e gratuito) sobre matemática para a ciência da computação [50].

1.3.1 Exponenciais e logaritmos

A expressão b^x denota o número b elevado à potência x. Se x é um inteiro positivo, então este é apenas o valor de b multiplicado por si próprio $x-1$ vezes:

$$b^x = \underbrace{b \times b \times \cdots \times b}_{x} \ .$$

Quando x é um inteiro negativo, $b^{-x} = 1/b^x$. Quando $x = 0$, $b^x = 1$. Quando b não é um inteiro, ainda podemos definir a exponenciação em termos da função exponencial e^x (ver abaixo), que é, ela própria, definida em termos da série exponencial, mas isso é melhor deixar para um texto de cálculo .

Neste livro, a expressão $\log_b k$ denota o *logaritmo base b* de k. Ou seja, o valor único x que satisfaz

$$b^x = k \ .$$

A maioria dos logaritmos neste livro é de base 2 (*logaritmos binários*). Para estes, omitimos a base, de modo que $\log k$ é abreviação para $\log_2 k$.

Uma maneira informal, porém útil, de pensar em logaritmos, é pensar em $\log_b k$ como o número de vezes que temos de dividir k por b antes que o resultado seja menor ou igual a 1. Por exemplo, quando se faz pesquisa binária, cada comparação reduz o número de respostas possíveis por um fator de 2. Isso é repetido até que haja no máximo uma resposta possível. Portanto, o número de comparação feita por pesquisa binária quando há inicialmente no máximo $n+1$ respostas possíveis é no máximo $\lceil \log_2(n+1) \rceil$.

Outro logaritmo que aparece várias vezes neste livro é o *logaritmo natural*. Aqui usamos a notação $\ln k$ para denotar $\log_e k$, onde e — *constante de Euler* — é dada por

$$e = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n \approx 2.71828 \ .$$

O logaritmo natural surge frequentemente porque é o valor de uma integral particularmente comum:

$$\int_1^k 1/x \, \mathrm{d}x = \ln k \ .$$

Duas das manipulações mais comuns que fazemos com logaritmos são removê-los de um expoente:

$$b^{\log_b k} = k$$

e mudar a base de um logaritmo:

$$\log_b k = \frac{\log_a k}{\log_a b} \ .$$

Por exemplo, podemos usar essas duas manipulações para comparar os logaritmos naturais e binários

$$\ln k = \frac{\log k}{\log e} = \frac{\log k}{(\ln e)/(\ln 2)} = (\ln 2)(\log k) \approx 0.693147 \log k \ .$$

1.3.2 Fatoriais

Em um ou dois lugares neste livro, a função *fatorial* é usada. Para um inteiro não negativo n, a notação $n!$ (Pronunciada "n fatorial") é definida como significando

$$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n \ .$$

Fatoriais aparecem porque $n!$ conta o número de permutações, isto é, ordenamentos, de n elementos distintos. Para o caso especial $n = 0$, $0!$ é definido como 1.

A quantidade $n!$ pode ser aproximada usando *Aproximação de Stirling*:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha(n)} \ ,$$

onde

$$\frac{1}{12n+1} < \alpha(n) < \frac{1}{12n} \ .$$

A aproximação de Stirling também aproxima $\ln(n!)$:

$$\ln(n!) = n \ln n - n + \frac{1}{2}\ln(2\pi n) + \alpha(n)$$

(De fato, a Aproximação de Stirling é mais facilmente comprovada pela aproximação $\ln(n!) = \ln 1 + \ln 2 + \cdots + \ln n$ pela integral $\int_1^n \ln n \, \mathrm{d}n = n \ln n - n + 1$.)

Os *coeficientes binomiais* são relacionadas à função fatorial. Para um inteiro não-negativo n e um inteiro $k \in \{0, \ldots, n\}$, a notação $\binom{n}{k}$ indica:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \ .$$

O coeficiente binomial $\binom{n}{k}$ (pronunciado "n escolhido k") conta o número de subconjuntos de um conjunto de elementos n que têm o tamanho k, isto é, o número de maneiras de escolher k inteiros distintos a partir do conjunto $\{1, \ldots, n\}$.

1.3.3 Notação assintótica

Ao analisar as estruturas de dados neste livro, queremos falar sobre os tempos de execução de várias operações. Os tempos de execução exatos, naturalmente, variam de computador para computador e até mesmo entre as execuções em um computador individual. Quando falamos sobre o tempo de execução de uma operação, estamos nos referindo ao número de instruções do computador realizadas durante a operação. Mesmo para o código simples, essa quantidade pode ser difícil de calcular exatamente. Portanto, em vez de analisar exatamente os tempos de execução, usaremos a chamada *notação big-O*: para uma função $f(n)$, $O(f(n))$ denota um conjunto de funções,

$$O(f(n)) = \left\{ \begin{array}{l} g(n) : \text{existe } c > 0, \text{ e } n_0 \text{ tais que} \\ \quad g(n) \leq c \cdot f(n) \text{ para todo } n \geq n_0 \end{array} \right\} .$$

Pensando graficamente, este conjunto consiste das funções $g(n)$ onde $c \cdot f(n)$ começa a dominar $g(n)$ quando n é suficientemente grande.

Geralmente, utilizamos notação assintótica para simplificar funções. Por exemplo, no lugar de $5n\log n + 8n - 200$ podemos escrever $O(n\log n)$. Isto é comprovado da seguinte forma:

$$\begin{aligned} 5n\log n + 8n - 200 &\leq 5n\log n + 8n \\ &\leq 5n\log n + 8n\log n \qquad \text{para } n \geq 2 \text{ (então } \log n \geq 1) \\ &\leq 13n\log n \ . \end{aligned}$$

Isso demonstra que a função $f(n) = 5n\log n + 8n - 200$ está no conjunto $O(n\log n)$ usando as constantes $c = 13$ e $n_0 = 2$.

Diversos atalhos úteis podem ser aplicados ao usar notação assintótica. Primeiro

$$O(n^{c_1}) \subset O(n^{c_2}) \ ,$$

para qualquer $c_1 < c_2$. Segundo: para quaisquer constantes $a, b, c > 0$,

$$O(a) \subset O(\log n) \subset O(n^b) \subset O(c^n) \ .$$

Essas relações de inclusão podem ser multiplicadas por qualquer valor positivo, e elas ainda são válidas. Por exemplo, a multiplicação por n

produz:

$$O(n) \subset O(n \log n) \subset O(n^{1+b}) \subset O(nc^n) \ .$$

Continuando em uma longa e distinta tradição, abusaremos desta notação escrevendo coisas como $f_1(n) = O(f(n))$ quando o que realmente queremos dizer é $f_1(n) \in O(f(n))$. Também faremos declarações como "o tempo de execução desta operação é $O(f(n))$" quando essa instrução deve ser "o tempo de execução desta operação é *membro de* $O(f(n))$." Esses atalhos são principalmente para evitar incômodos da linguagem e para facilitar a utilização de notação assintótica dentro de cadeias de equações.

Um exemplo particularmente estranho disso ocorre quando escrevemos

$$T(n) = 2 \log n + O(1) \ .$$

Novamente, isso seria mais corretamente escrito como

$$T(n) \leq 2 \log n + [\text{algum membro de } O(1)] \ .$$

A expressão $O(1)$ também traz outra questão. Como não há nenhuma variável nessa expressão, pode não estar claro qual variável está ficando arbitrariamente grande. Sem contexto, não há maneira de dizer. No exemplo acima, uma vez que a única variável no restante da equação é n, podemos supor que isto deve ser lido como $T(n) = 2 \log n + O(f(n))$, onde $f(n) = 1$.

A notação Big-O não é algo novo ou exclusivo da ciência da computação. Foi usada pelo teórico do número Paul Bachmann já em 1894, e é imensamente útil para descrever os tempos de execução de algoritmos de computador. Considere o seguinte código:

```
fragmento()
    i ← 0
    while i < n do
        a[i] ← i
        i++
```

Uma execução deste método envolve

- 1 atribuição ($i \leftarrow 0$),

- $n + 1$ comparações ($i < n$),
- n incrementos ($i++$),
- n cálculos de deslocamentos no vetor ($a[i]$), e
- n atribuições indiretas ($a[i] \leftarrow i$).

Então, nós poderíamos escrever este tempo de execução como

$$T(n) = a + b(n+1) + cn + dn + en \ ,$$

Onde a, b, c, d, e e são constantes que dependem da máquina que está executando o código e representam o tempo para executar atribuições, comparações, operações de incremento, cálculos de deslocamento no array e atribuições, respectivamente. No entanto, se esta expressão representa o tempo de execução de duas linhas de código, então claramente este tipo de análise não será tratável para códigos ou algoritmos complicados. Usando a notação big-O, o tempo de execução pode ser simplificado para

$$T(n) = O(n) \ .$$

Não só isso é mais compacto, mas também dá quase tanta informação quanto a expressão anterior. O fato de que o tempo de execução depende das constantes a, b, c, d e e no exemplo acima significa que, em geral, não será possível comparar dois tempos de execução para saber qual é mais rápido sem conhecer os valores dessas constantes. Mesmo se fizermos o esforço para determinar essas constantes (digamos, por meio de testes de tempo), então nossa conclusão será válida apenas para a máquina em que executamos nossos testes.

A notação Big-O nos permite raciocinar a um nível muito mais alto, tornando possível analisar funções mais complicadas. Se dois algoritmos tiverem o mesmo tempo de execução big-O, então não saberemos qual é o mais rápido, e pode não haver um vencedor claro. Um pode ser mais rápido em uma máquina, e o outro pode ser mais rápido em uma máquina diferente. No entanto, se os dois algoritmos têm comprovadamente diferentes tempos de execução big-O, então podemos ter certeza de que aquele com menor tempo de execução será mais rápido *para valores suficientemente grandes de n*.

Um exemplo de como a notação de big-O nos permite comparar duas funções diferentes é mostrado em Figura 1.5, que compara a taxa de crescimento de $f_1(n) = 15n$ versus $f_2(n) = 2n \log n$. Pode ser que $f_1(n)$ seja o tempo de execução de um algoritmo de tempo linear complicado enquanto $f_2(n)$ é o tempo de execução de um algoritmo consideravelmente mais simples baseado no paradigma de divisão e conquista. Isso ilustra que, embora $f_1(n)$ seja maior que $f_2(n)$ para valores pequenos de n, o oposto é verdadeiro para valores grandes de n. Eventualmente $f_1(n)$ ganha, por uma margem cada vez maior. A análise usando a notação Big-O nos disse que isso aconteceria, já que $O(n) \subset O(n \log n)$.

Em alguns casos, usaremos notação assintótica em funções com mais de uma variável. Parece não haver um padrão para isso, mas para nossos propósitos, a seguinte definição é suficiente:

$$O(f(n_1,\ldots,n_k)) = \left\{ \begin{array}{l} g(n_1,\ldots,n_k) : \text{existe } c > 0, \text{ e } z \text{ tal que} \\ \quad g(n_1,\ldots,n_k) \leq c \cdot f(n_1,\ldots,n_k) \\ \quad\quad \text{para todo } n_1,\ldots,n_k \text{ tal que } g(n_1,\ldots,n_k) \geq z \end{array} \right\} .$$

Esta definição capta a situação que realmente nos interessa: quando os argumentos $n_1,\ldots,n_k$ fazem com que g assuma grandes valores. Esta definição também concorda com a definição univariada de $O(f(n))$ quando $f(n)$ é uma função crescente de n. O leitor deve ser advertido que, embora isso funcione para nossos propósitos, outros textos podem tratar funções multivariadas e notação assintótica de forma diferente.

1.3.4 Randomização e Probabilidade

Algumas das estruturas de dados apresentadas neste livro são *randomizadas*; elas fazem escolhas aleatórias que são independentes dos dados que estão sendo armazenados nelas ou as operações que estão sendo realizadas sobre eles. Por esta razão, executar o mesmo conjunto de operações mais de uma vez, usando essas estruturas, pode resultar em tempos de execução diferentes. Ao analisar essas estruturas de dados, estamos interessados em sua média ou tempo de execução *esperado*.

Formalmente, o tempo de execução de uma operação em uma estrutura de dados aleatória é uma variável aleatória, e queremos estudar seu *valor esperado*. Para uma variável aleatória discreta X assumindo valores

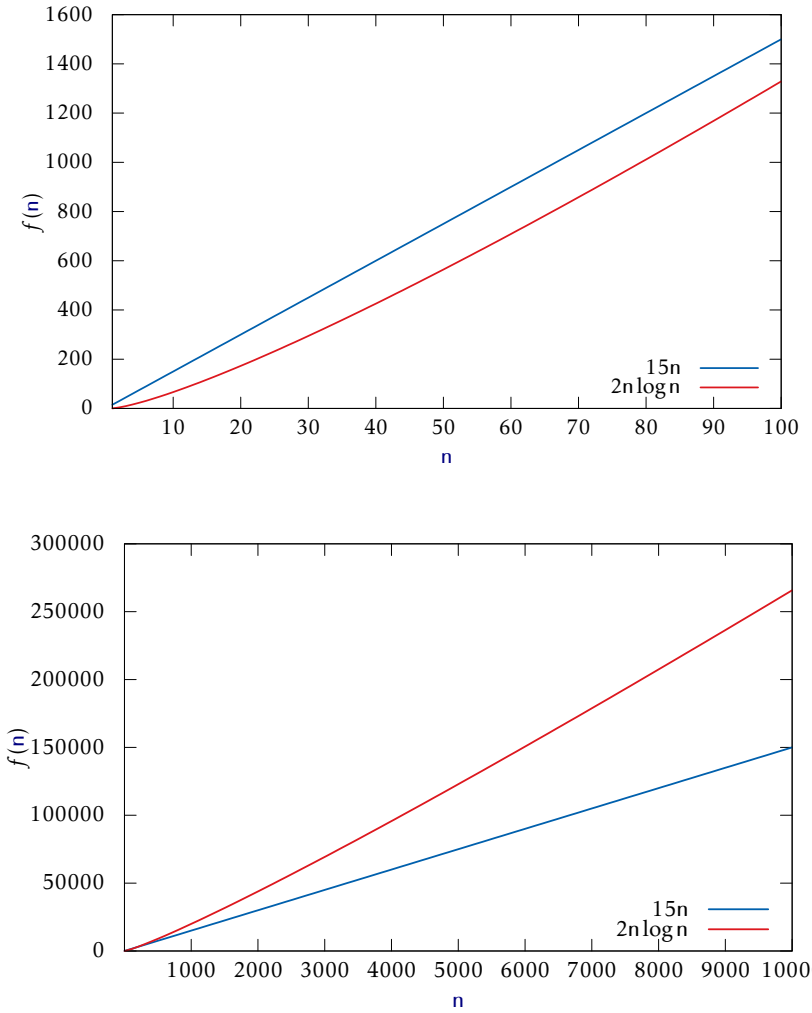

Figura 1.5: Gráfico para $15n$ versus $2n \log n$.

em algum universo contábil U, o valor esperado de X, denotado por $\mathrm{E}[X]$, é dado pela fórmula

$$\mathrm{E}[X] = \sum_{x \in U} x \cdot \Pr\{X = x\} \ .$$

Aqui $\Pr\{\mathcal{E}\}$ denota a probabilidade de ocorrência do evento $\mathcal{E}$. Em todos os exemplos neste livro, essas probabilidades são apenas com relação às escolhas aleatórias feitas pela estrutura de dados randomizados; não há nenhuma suposição de que os dados armazenados na estrutura, ou que a sequência de operações realizadas na estrutura de dados, seja aleatória.

Uma das propriedades mais importantes dos valores esperados é a *linearidade de expectativa*. Para quaisquer duas variáveis aleatórias X e Y,

$$\mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y] \ .$$

De maneira mais geral, para quaisquer variáveis aleatórias $X_1, \ldots, X_k$,

$$\mathrm{E}\left[\sum_{i=1}^{k} X_i\right] = \sum_{i=1}^{k} \mathrm{E}[X_i] \ .$$

A linearidade da expectativa nos permite quebrar variáveis aleatórias complicadas (como os lados da esquerda das equações acima) em somas de variáveis aleatórias mais simples (os lados da direita).

Um truque útil, que iremos usar repetidamente, é definir um *indicador de variáveis aleatórias*. Estas variáveis binárias são úteis quando queremos contar alguma coisa e são melhor ilustradas por um exemplo. Suponha que jogamos uma moeda honesta k vezes e queremos saber o número esperado de vezes que a moeda aparece como cara. Intuitivamente, sabemos que a resposta é $k/2$, mas se tentarmos prová-la usando a definição de valor esperado, obtemos

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=0}^{k} i \cdot \Pr\{X = i\} \\
&= \sum_{i=0}^{k} i \cdot \binom{k}{i} / 2^k \\
&= k \cdot \sum_{i=0}^{k-1} \binom{k-1}{i} / 2^k
\end{aligned}$$

$$= k/2 \ .$$

Isso exige que saibamos o suficiente para calcular que $\Pr\{X = i\} = \binom{k}{i}/2^k$, e que conheçamos as identidades binomiais $i\binom{k}{i} = k\binom{k-1}{i-1}$ e $\sum_{i=0}^{k}\binom{k}{i} = 2^k$.

Usar indicador de variáveis e linearidade de expectativa torna as coisas muito mais fáceis. Para cada $i \in \{1,\ldots,k\}$, defina o indicador de variável aleatória

$$I_i = \begin{cases} 1 & \text{se o } i\text{-ésimo lançamento de moeda é cara} \\ 0 & \text{caso contrário.} \end{cases}$$

Então

$$\mathrm{E}[I_i] = (1/2)1 + (1/2)0 = 1/2 \ .$$

Agora, $X = \sum_{i=1}^{k} I_i$, so

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{k} I_i\right] \\ &= \sum_{i=1}^{k} \mathrm{E}[I_i] \\ &= \sum_{i=1}^{k} 1/2 \\ &= k/2 \ . \end{aligned}$$

Este é um pouco mais longo, mas não requer que nós conheçamos quaisquer identidades mágicas ou compute quaisquer probabilidades não triviais. Melhor ainda, concorda com a intuição de que esperamos que metade das moedas apareça como cara precisamente porque cada moeda individual aparece como cara com uma probabilidade de 1/2.

1.4 O Modelo de Computação

Neste livro, analisaremos os tempos teóricos de funcionamento das estruturas de dados que estudamos. Para fazer isso precisamente, necessitamos de um modelo matemático de computação. Para isso, usamos o modelo de palavra-RAM de *w-bits*. RAM significa Random Access

Machine. Neste modelo, temos acesso a uma memória de acesso aleatório constituída por *células*, cada uma das quais armazena uma *palavra* de w-bits. Isto implica que uma célula de memória pode representar, por exemplo, qualquer inteiro no conjunto $\{0, \ldots, 2^w - 1\}$.

No modelo de palavra-RAM, operações básicas em palavras levam tempo constante. Isso inclui operações aritméticas ($+$, $-$, $\cdot$, $/$, mod), comparações ($<$, $>$, $=$, $\leq$, $\geq$), e operações booleanas bit a bit (AND, OR e exclusivo-OR bit a bit).

Qualquer célula pode ser lida ou escrita em tempo constante. A memória de um computador é gerenciada por um sistema de gerenciamento de memória a partir do qual podemos alocar ou desalocar um bloco de memória de qualquer tamanho que gostaríamos. Alocar um bloco de memória de tamanho k leva um tempo $O(k)$ e retorna uma referência (um ponteiro) para o bloco de memória recém-alocado. Esta referência é suficientemente pequena para ser representada por uma única palavra.

O tamanho da palavra w é um parâmetro muito importante deste modelo. O único pressuposto que faremos sobre w é o limite inferior $w \geq \log n$, onde n é o número de elementos armazenados em qualquer uma das nossas estruturas de dados. Esta é uma suposição bastante modesta, uma vez que caso contrário uma palavra não é mesmo grande o suficiente para contar o número de elementos armazenados na estrutura de dados.

O espaço é medido em palavras, de modo que quando falamos sobre a quantidade de espaço usado por uma estrutura de dados, estamos nos referindo ao número de palavras de memória usadas pela estrutura. Todas as nossas estruturas de dados armazenam valores de um tipo genérico T, e assumimos que um elemento do tipo T ocupa uma palavra de memória.

As estruturas de dados apresentadas neste manual não usam truques especiais que não sejam implementáveis.

1.5 Corretude, Complexidade no Tempo e Complexidade no Espaço

Ao estudar o desempenho de uma estrutura de dados, há três coisas que mais importam:

Corretude: A estrutura de dados deve implementar corretamente sua interface.

Complexidade no Tempo: Os tempos de execução das operações na estrutura de dados devem ser tão pequenos quanto possível.

Complexidade no Espaço: A estrutura de dados deve usar a menor memória possível.

Neste texto introdutório, tomaremos a correção como um dado; não consideraremos estruturas de dados que dão respostas incorretas a consultas ou não executam atualizações corretamente. Iremos, no entanto, ver estruturas de dados que fazem um esforço extra para manter o uso do espaço a um mínimo. Isso normalmente não afeta os tempos de operação (assintóticos) das operações, mas pode tornar as estruturas de dados um pouco mais lentas na prática.

Ao estudar tempos de execução no contexto das estruturas de dados, tendemos a obter três tipos diferentes de garantias de tempo de execução:

Tempo de execução do pior caso: Estes são o tipo mais forte de garantias de tempo de execução. Se uma operação de estrutura de dados tiver um pior tempo de execução de $f(n)$, então uma dessas operações *nunca* demora mais de $f(n)$ unidades de tempo.

Tempo de execução amortizado: Se dissermos que o tempo de execução amortizado de uma operação em uma estrutura de dados é $f(n)$, então isso significa que o custo de uma operação típica é no máximo $f(n)$. Mais precisamente, se uma estrutura de dados tem um tempo de execução amortizado de $f(n)$, então uma sequência de m operações leva no máximo $mf(n)$ unidades de tempo. Algumas operações individuais podem demorar mais de $f(n)$ unidades de tempo, mas a média, ao longo de toda a sequência de operações, é no máximo $f(n)$.

Tempo de execução esperado: Se dissermos que o tempo de execução esperado de uma operação em uma estrutura de dados é $f(n)$, isso significa que o tempo de execução real é uma variável aleatória (ver Seção 1.3.4) e o valor esperado desta variável aleatória é no máximo

$f(n)$. A randomização aqui é com respeito a escolhas aleatórias feitas pela estrutura de dados.

Para entender a diferença entre os tempos de execução de pior caso, amortizado e esperado, ajuda considerar um exemplo financeiro. Considere o custo de comprar uma casa:

Pior caso versus custo amortizado: Suponha que uma casa custa \$120 000. Para comprar esta casa, podemos obter uma hipoteca de 120 meses (10 anos) com pagamentos mensais de \$1 200 por mês. Neste caso, o pior caso para o custo mensal de pagar esta hipoteca é \$1 200 por mês.

Se tivermos dinheiro suficiente à mão, poderemos escolher comprar a casa de uma vez, com um pagamento de \$120 000. Neste caso, durante um período de 10 anos, o custo mensal amortizado da compra desta casa é

$$\$120\,000/120 \text{ meses} = \$1\,000 \text{ por mês} \ .$$

Isso é muito menor do que o \$1 200 por mês que teríamos que pagar se usássemos uma hipoteca.

Pior caso versus custo esperado: Em seguida, considere a questão do seguro de incêndio em nossa casa de \$120 000. Ao estudar centenas de milhares de casos, as companhias de seguros determinaram que a quantidade esperada de danos causados por incêndio causados a uma casa como a nossa é de \$10 por mês. Este é um número muito pequeno, uma vez que a maioria das casas nunca têm incêndios, algumas casas podem ter alguns pequenos incêndios que causam um pouco de dano de fumaça, e um pequeno número de casas queimam até suas fundações. Com base nestas informações, a companhia de seguros cobra \$15 por mês pelo seguro de incêndio.

Agora é hora da decisão. Deveríamos pagar o custo mensal de \$15 para o seguro de incêndio, ou deveríamos apostar e auto-segurar a um custo esperado de \$10 por mês? Claramente, o \$10 por mês custa menos *na expectativa*, mas temos que ser capazes de aceitar a possibilidade de que o *custo real* pode ser muito maior. No caso improvável de que toda a casa queime, o custo real será \$120 000.

Esses exemplos financeiros também oferecem uma visão sobre porque às vezes nos conformamos com um tempo de execução amortizado ou esperado em vez de um pior tempo de execução. Muitas vezes é mais provável obter um menor tempo de execução esperado ou amortizado do que um pior caso de tempo de execução. Pelo menos, muitas vezes é possível obter uma estrutura de dados muito mais simples se estamos dispostos a usar tempos de execução amortizados ou esperados.

1.6 Exemplos de Código

Os exemplos de código neste livro são escritos em pseudocódigo. Estes devem ser fáceis de ler para qualquer pessoa que tenha alguma experiência de programação em qualquer uma das linguagens de programação mais comuns dos últimos 40 anos. Para ter uma ideia de como o pseudocódigo neste livro parece, aqui está uma função que calcula a média de uma matriz, a:

```
average(a)
    s ← 0
    for i in 0, 1, 2, ..., length(a) − 1 do
        s ← s + a[i]
    return s/length(a)
```

Como esse código ilustra, a atribuição a uma variável é feita usando a notação ←. Usamos a convenção de que o tamanho de uma matriz, a, é denotado por length(a) e os índices de matriz começam em zero, então $0, 1, 2, \ldots,$ length(a) $-$ 1 são os índices válidos para a. Para encurtar o código, e às vezes torná-lo mais fácil de ler, o nosso pseudocódigo permite atribuições na (sub)matriz. As duas funções a seguir são equivalentes:

```
left_shift_a(a)
    for i in 0, 1, 2, ..., length(a) − 2 do
        a[i] ← a[i + 1]
    a[length(a) − 1] ← nil
```

```
left_shift_b(a)
    a[0,1,...,length(a) - 2] ← a[1,2,...,length(a) - 1]
    a[length(a) - 1] ← nil
```

O código a seguir define todos os valores de uma matriz como zero:

```
zero(a)
    a[0,1,...,length(a) - 1] ← 0
```

Ao analisar o tempo de execução de um código como este, temos de ter cuidado; declarações como $a[0,1,\ldots,\text{length}(a)-1] \leftarrow 1$ ou $a[1,2,\ldots,\text{length}(a)-1] \leftarrow a[0,1,\ldots,\text{length}(a)-2]$ não executam em tempo constante. Eles são executados em tempo $O(\text{length}(a))$.

Tomamos atalhos semelhantes com atribuições de variáveis, de modo que o código $x, y \leftarrow 0, 1$ define x como zero e y como 1 e o código $x, y \leftarrow y, x$ troca os valores das variáveis x e y.

Nosso pseudocódigo usa alguns operadores que podem não ser familiares. Como é padrão na matemática, a divisão (normal) é indicada pelo operador /. Em muitos casos, queremos fazer divisão inteira em vez disso, caso em que usamos o operador div, de modo que $a \operatorname{div} b = \lfloor a/b \rfloor$ é a parte inteira de a/b. Assim, por exemplo, $3/2 = 1.5$ mas $3 \operatorname{div} 2 = 1$. Ocasionalmente, também usamos o operador mod para obter o resto da divisão inteira, mas isso será definido quando ele for usado. Mais adiante no livro, podemos usar alguns operadores bit a bit, incluindo shift esquerdo ($\ll$), shift direito ($\gg$), bit a bit e ($\wedge$) e bit a bit exclusivo-ou ($\oplus$).

As amostras de pseudocódigo neste livro são traduções automáticas de código Python que podem ser baixadas do site do livro. [2] Se você encontrar uma ambiguidade no pseudocódigo que não possa resolver por si mesmo, então você sempre pode consultar o código Python correspondente. Se você não souber Python, o código também está disponível em Java e C ++. Se você não consegue decifrar o pseudocódigo, ou ler Python, C ++ ou Java, então você pode ainda não estar pronto para este livro.

[2] http://opendatastructures.org

Implementações de Lista			
	get(i)/set(i,x)	add(i,x)/remove(i)	
ArrayStack	$O(1)$	$O(1+n-i)^{\mathrm{A}}$	§ 2.1
ArrayDeque	$O(1)$	$O(1+\min\{i,n-i\})^{\mathrm{A}}$	§ 2.4
DualArrayDeque	$O(1)$	$O(1+\min\{i,n-i\})^{\mathrm{A}}$	§ 2.5
RootishArrayStack	$O(1)$	$O(1+n-i)^{\mathrm{A}}$	§ 2.6
DLList	$O(1+\min\{i,n-i\})$	$O(1+\min\{i,n-i\})$	§ 3.2
SEList	$O(1+\min\{i,n-i\}/b)$	$O(b+\min\{i,n-i\}/b)^{\mathrm{A}}$	§ 3.3
SkiplistList	$O(\log n)^{\mathrm{E}}$	$O(\log n)^{\mathrm{E}}$	§ 4.3

Implementações de USet			
	find(x)	add(x)/remove(x)	
ChainedHashTable	$O(1)^{\mathrm{E}}$	$O(1)^{\mathrm{A,E}}$	§ 5.1
LinearHashTable	$O(1)^{\mathrm{E}}$	$O(1)^{\mathrm{A,E}}$	§ 5.2

[A] Indica um tempo de execução *amortizado*.
[E] Indica um tempo de execução *esperado*.

Tabela 1.1: Sumário das implementações de Lista and USet.

1.7 Lista de Estruturas de Dados

As tabelas 1.1 e 1.2 resumem o desempenho das estruturas de dados neste livro que implementam cada uma das interfaces, List, USet e SSet descritas em Seção 1.2. Figura 1.6 mostra as dependências entre vários capítulos neste livro. Uma seta tracejada indica apenas uma dependência fraca, na qual apenas uma pequena parte do capítulo depende de um capítulo anterior ou apenas dos principais resultados do capítulo anterior.

1.8 Discussões e Exercícios

As interfaces Lista, USet e SSet descritas em Seção 1.2 são influenciadas pelo Framework Java Collections [54]. Essas são versões essencialmente

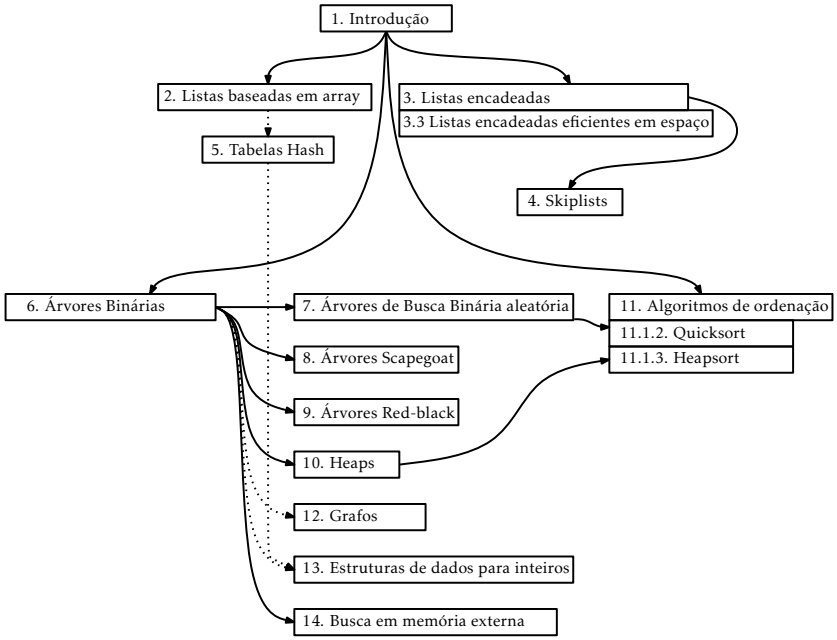

Figura 1.6: As dependências entre capítulos neste livro.

Implementações de SSet			
	find(x)	add(x)/remove(x)	
SkiplistSSet	$O(\log n)^{E}$	$O(\log n)^{E}$	§ 4.2
Treap	$O(\log n)^{E}$	$O(\log n)^{E}$	§ 7.2
ScapegoatTree	$O(\log n)$	$O(\log n)^{A}$	§ 8.1
RedBlackTree	$O(\log n)$	$O(\log n)$	§ 9.2
BinaryTrie^{I}	$O(w)$	$O(w)$	§ 13.1
XFastTrie^{I}	$O(\log w)^{A,E}$	$O(w)^{A,E}$	§ 13.2
YFastTrie^{I}	$O(\log w)^{A,E}$	$O(\log w)^{A,E}$	§ 13.3

(Priority) Implementações de Queue			
	find_min()	add(x)/remove()	
BinaryHeap	$O(1)$	$O(\log n)^{A}$	§ 10.1
MeldableHeap	$O(1)$	$O(\log n)^{E}$	§ 10.2

^{I} Esta estrutura só pode armazenar dados inteiros de w-bit.

Tabela 1.2: Sumário das implementações de SSet e da Fila de prioridade.

simplificadas das interfaces Lista, Set, Map, SortedSet e SortedMap encontradas no Framework Java Collections.

Para um tratamento soberbo (e livre) da matemática discutido neste capítulo, incluindo a notação assintótica, logaritmos, fatoriais, aproximação de Stirling, probabilidade básica e muito mais, veja o livro de texto de Leyman, Leighton e Meyer [50]. Para um texto de cálculo suave que inclui definições formais de exponenciais e logaritmos, veja o texto clássico (livremente disponível) por Thompson [71].

Para obter mais informações sobre probabilidade básica, especialmente no que se refere à ciência da computação, consulte o livro de texto de Ross [63]. Outra boa referência, que abrange tanto a notação assintótica quanto a probabilidade, é o livro de Graham, Knuth e Patashnik [37].

Exercício 1.1. Este exercício foi projetado para ajudar a familiarizar o leitor com a escolha da estrutura de dados correta para o problema correto. Se implementado, as partes deste exercício devem ser feitas usando uma implementação da interface relevante (Stack, Queue, Deque, USet ou SSet).

Resolva os seguintes problemas lendo um arquivo de texto uma linha

por vez e executando operações em cada linha na(s) estrutura(s) de dados apropriada(s). Suas implementações devem ser rápidas o suficiente para que mesmo arquivos contendo um milhão de linhas possam ser processados em poucos segundos.

1. Leia a entrada de uma linha de cada vez e, em seguida, escreva as linhas em ordem inversa, de modo que a última linha de entrada seja impressa primeiro, depois a segunda última linha de entrada, e assim por diante.

2. Leia as primeiras 50 linhas de entrada e depois escreva-as em ordem inversa. Leia as próximas 50 linhas e depois escreva-as em ordem inversa. Faça isso até que não haja mais linhas deixadas para ler, neste ponto, quaisquer linhas restantes devem ser impressas na ordem inversa.

   Em outras palavras, sua saída começará com a 50ª linha, depois com a 49ª, depois com a 48ª, e assim por diante até a primeira linha. Isto será seguido pela 100ª linha, seguida pela 99ª, e assim por diante até a 51ª. O processo continua indefinidamente.

   Seu código nunca deve ter que armazenar mais de 50 linhas em um determinado momento.

3. Leia a entrada uma linha de cada vez. Em qualquer ponto depois de ler as primeiras 42 linhas, se alguma linha estiver em branco (ou seja, uma sequência de comprimento 0), imprima a linha que ocorreu 42 linhas anteriores a essa. Por exemplo, se a Linha 242 estiver em branco, então seu programa deve imprimir a linha 200. Este programa deve ser implementado de modo que nunca armazene mais de 43 linhas da entrada a qualquer momento.

4. Leia a entrada uma linha de cada vez e escreva cada linha na saída se não for uma duplicata de alguma linha de entrada anterior. Tome especial cuidado para que um arquivo com um monte de linhas duplicadas não use mais memória do que o necessário para o número de linhas únicas.

5. Leia a entrada uma linha de cada vez e escreva cada linha para a saída somente se você já leu esta linha antes. (O resultado final é que

você remove a primeira ocorrência de cada linha.) Tome especial cuidado para que um arquivo com um monte de linhas duplicadas não use mais memória do que o necessário para o número de linhas únicas.

6. Leia toda a entrada uma linha de cada vez. Em seguida, imprima todas as linhas ordenadas por comprimento, com as linhas mais curtas primeiro. No caso em que duas linhas tenham o mesmo comprimento, resolva sua ordem usando a "ordem classificada". As linhas duplicadas devem ser impressas apenas uma vez.

7. Faça o mesmo que a pergunta anterior, exceto que as linhas duplicadas devem ser impressas o mesmo número de vezes que aparecem na entrada.

8. Leia toda a entrada uma linha de cada vez e, em seguida, imprimir as linhas pares numeradas (começando com a primeira linha, linha 0) seguida pelas linhas ímpares.

9. Leia toda a entrada uma linha de cada vez e permute aleatoriamente as linhas antes de imprimi-las. Para ser claro: Você não deve modificar o conteúdo de qualquer linha. Em vez disso, a mesma coleção de linhas deve ser impressa, mas em uma ordem aleatória....

Exercício 1.2. Uma *palavra Dyck* é uma sequência de +1's e -1's com a propriedade que a soma de qualquer prefixo da sequência nunca seja negativo. Por exemplo, $+1,-1,+1,-1$ é uma palavra *Dyck*, porém $+1,-1,-1,+1$ não é uma palavra *Dyck* posto que o prefixo $+1-1-1<0$. Descreva qualquer relação entre palavras Dyck e as operações na Stack push(x) e pop().

Exercício 1.3. Uma *string casada* é uma sequência de caracteres {, }, (,), [, e] que estão casados de forma correta. Por exemplo, "{{()[]}}" é uma string casada, porém "{{()]}" não é, posto que o segundo { casa com um]. Mostre como usar uma pilha para que, dada uma string de comprimento n, você possa determinar se ela é uma string casada em um tempo $O(n)$.

Exercício 1.4. Suponha que você tenha uma Stack, s, que suporta somente as operações push(x) e pop(). Mostre como, usando apenas uma Fila FIFO, f, você pode inverter a ordem de todos os elementos em s.

Exercício 1.5. Usando um USet, implementar um Bag. Um Bag é como um USet — suporta os métodos add(x), remove(x) e find(x) mas permite que elementos duplicados sejam armazenados. A operação find(x) em um Bag retorna algum elemento (se houver) que é igual a x. Além disso, um Bag suporta a operação find_all(x) que retorna uma lista de todos os elementos no Bag que são iguais a x.

Exercício 1.6. A partir do zero, escreva e teste as implementações das interfaces Lista, USet e SSet. Estas não têm de ser eficientes. Elas podem ser usados mais tarde para testar a correção e o desempenho de implementações mais eficientes. (A maneira mais fácil de fazer isso é armazenar os elementos em uma matriz.)

Exercício 1.7. Trabalhe para melhorar o desempenho de suas implementações a partir da pergunta anterior usando quaisquer truques que você possa pensar. Experimente e pense sobre como você poderia melhorar o desempenho de add(i, x) e remove(i) em sua implementação Lista. Pense em como você poderia melhorar o desempenho da operação find(x) em suas implementações de USet e SSet. Este exercício é projetado para darlhe uma sensação de como é difícil obter implementações eficientes dessas interfaces.