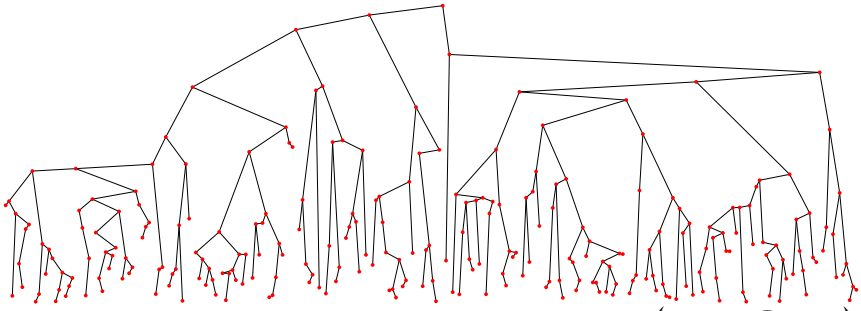


Estruturas de Dados Abertas(em C++)

Edição 0.1G β
18/08/2021

Pat Morin
tradução e adaptação para o português do Brasil:
João Araujo

Sumário

Agradecimentos

Sou grato a Nima Hoda, que passou um verão corrigindo incansavelmente muitos dos capítulos deste livro; aos alunos da disciplina COMP2402/2002 no outono 2011, que aguentaram o primeiro rascunho deste livro e alertaram sobre muitos erros tipográficos, gramaticais e factuais; e a Morgan Tunzelmann da Athabasca University Press, por ter pacientemente editado vários rascunhos quase finais.

0.1 Agradecimentos da Edição Brasileira

Gostaria de agradecer aos meus alunos do curso de Estruturas de Informação, da Universidade do Estado do Rio de Janeiro que gentilmente me auxiliaram na tradução deste livro. Foram eles: Diana Almeida Barros, Leonardo Lobão, Ester Gomes Pais, Fábio Cavallari, Lucas Ferreira Stefe, Matheus Caldeira Matias e Pedro Yuri dos Reis de Moraes Lopes.

Ass. João Araujo Ribeiro

Por que este Livro?

Há uma abundância de livros que ensinam estruturas introdutórias de dados. Alguns deles são muito bons. A maioria deles custa caro, e a grande maioria dos estudantes de graduação em ciência da computação desembolsará pelo menos algum dinheiro em um livro de estruturas de dados.

Vários livros de código aberto de estruturas de dados estão disponíveis on-line. Alguns são muito bons, mas a maioria deles estão ficando velhos. A maioria desses livros tornaram-se gratuitos quando seus autores e/ou editores decidiram parar de atualizá-los. A atualização desses livros geralmente não é possível, por duas razões: (1) O copyright pertence ao autor e/ou editor, qualquer um dos quais não pode permitir. (2) O *código fonte* para esses livros muitas vezes não está disponível. Ou seja, o Word, WordPerfect, FrameMaker ou fonte LaTeX para o livro não está disponível, e até mesmo a versão do software que manipula essa fonte pode não estar disponível.

O objetivo deste projeto é liberar estudantes de ciência da computação de graduação de ter que pagar por um livro introdutório de estruturas de dados. Decidi implementar este objetivo tratando este livro como um projeto de software Open Source . Os scripts do fonte em LaTeX, fontes em C++ e de construção para o livro estão disponíveis para download no site do autor[1] ou da versão em português[2], e também, mais importante, em uma fonte confiável de gerenciamento de código.[3][4]

O código-fonte no site é disponível sob uma licença Creative Com-

[1] http://opendatastructures.org
[2] https://www.araujo.eng.uerj.br/
[3] https://github.com/patmorin/ods
[4] https://github.com/jaraujouerj/Estruturas-de-Dados-Abertos

mons Attribution, o que significa que qualquer pessoa é livre para *compartilhar*, para copiar, distribuir e transmitir a obra; e para *remixar*: para adaptar o trabalho, incluindo o direito de fazer uso comercial da obra. A única condição para esses direitos é a *atribuição*: você deve reconhecer que o trabalho derivado contém código e/ou texto de `opendatastructures.org`.

Qualquer um pode contribuir com correções usando o sistema de gerenciamento de código-fonte `git`. Qualquer pessoa pode também derivar fontes do livro para desenvolver uma versão separada (por exemplo, em outra linguagem de programação). Minha esperança é que, fazendo as coisas desta maneira, este livro continue a ser um livro de texto útil muito depois de terminar meu interesse no projeto, ou meu pulso (o que ocorrer primeiro).

Preface to the C++ Edition

This book is intended to teach the design and analysis of basic data structures and their implementation in an object-oriented language. In this edition, the language happens to be C++.

This book is not intended to act as an introduction to the C++ programming language. Readers of this book need only be familiar with the basic syntax of C++ and similar languages. Those wishing to work with the accompanying source code should have some experience programming in C++.

This book is also not intended as an introduction to the C++ Standard Template Library or the generic programming paradigm that the STL embodies. This book describes implementations of several different data structures, many of which are used in implementations of the STL. The contents of this book may help an STL programmer understand how some of the STL data structures are implemented and why these implementations are efficient.

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.

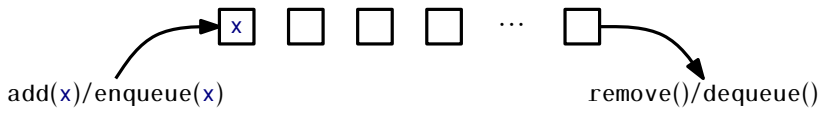


Figura 1.1: Uma Fila FIFO.

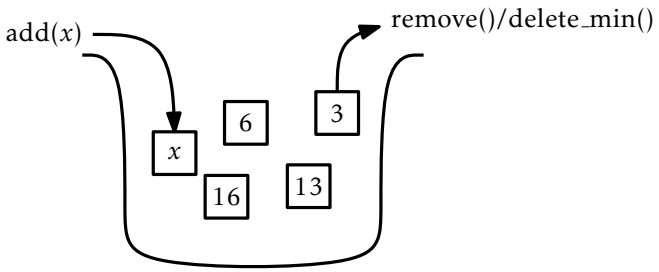


Figura 1.2: Uma Fila de Prioridade.

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 deleteMin() 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

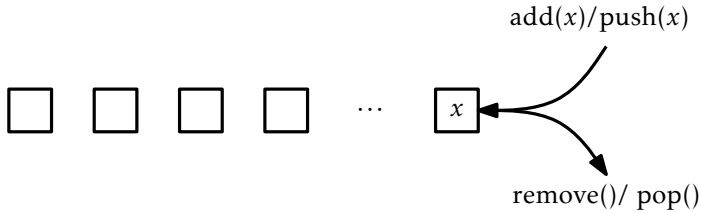


Figura 1.3: Uma Pilha.

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

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

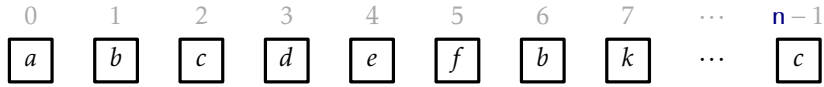


Figura 1.4: Uma Lista representa uma sequência indexada por $0, 1, 2, \ldots, n-1$. Nesta Lista, uma chamada para get(2) deve retornar o valor *c*.

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

$$\begin{aligned} \texttt{addFirst}(x) &\Rightarrow \texttt{add}(0, x) \\ \texttt{removeFirst}() &\Rightarrow \texttt{remove}(0) \\ \texttt{addLast}(x) &\Rightarrow \texttt{add}(\texttt{size}(), x) \\ \texttt{removeLast}() &\Rightarrow \texttt{remove}(\texttt{size}() - 1) \end{aligned}$$

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 `null` 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 `null` 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,null)`, 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

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

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 `null` 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:

Simple

```
void snippet() {
  for (int i = 0; i < n; i++)
    a[i] = i;
}
```

Uma execução deste método envolve

- 1 atribuição (`int i = 0`),
- `n` + 1 comparações (`i < n`),

- n incrementos (i++),
- n cálculos de deslocamentos no vetor (a[i]), e
- n atribuições indiretas (a[i] = 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

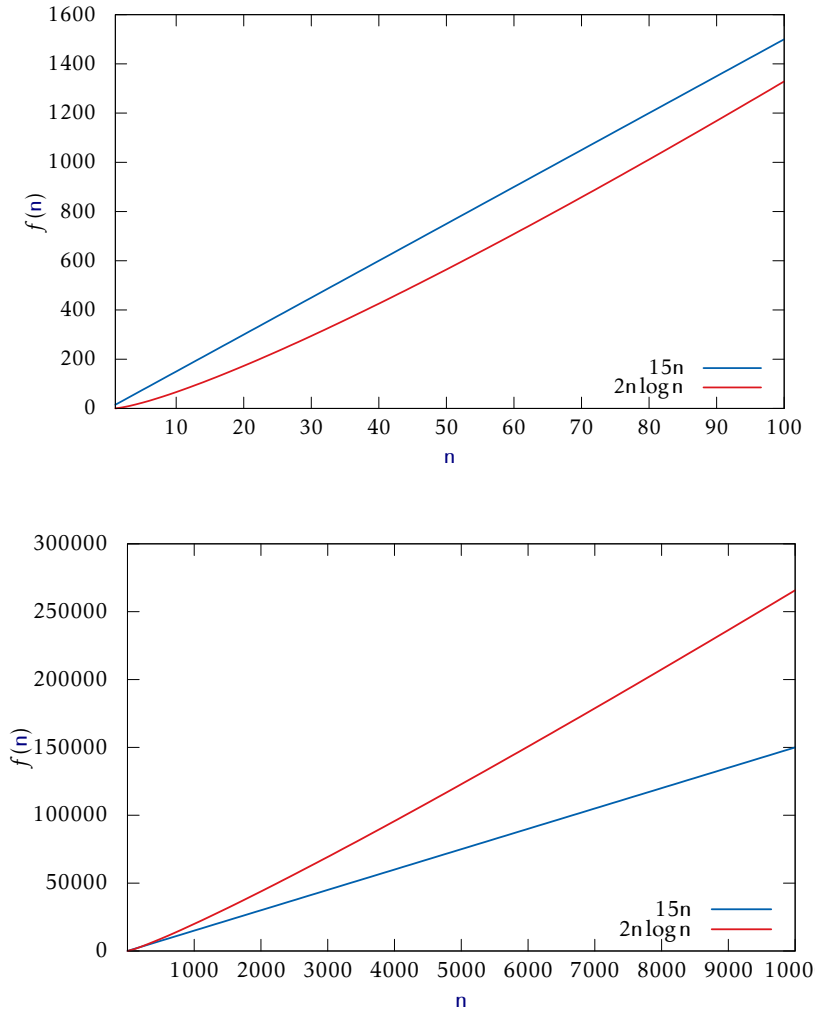


Figura 1.5: Gráfico para $15n$ versus $2n \log n$.

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 ($+$, $-$, $*$, $/$, $\%$), 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.

O modelo de palavra-RAM de w-bit é relativamente próximo dos modernos computadores desktop quando $w = 32$ ou $w = 64$. As estruturas de dados apresentadas neste manual não usam truques especiais que não sejam implementáveis em C++ na maioria das arquiteturas.

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 na linguagem de programação C++. No entanto, para tornar o livro acessível aos leitores não familiarizados com todas as construções e palavras-chave de C++, as amostras de código foram simplificadas. Por exemplo, um leitor não encontrará nenhuma das palavras-chave `public`, `protected`, `private` ou `static`. Um leitor também não encontrará muita discussão sobre hierarquias de classe. Quais as interfaces que uma determinada classe implementa ou que classe ela estende, se relevante para a discussão, deve ser claro a partir do texto que acompanha.

Estas convenções devem tornar os exemplos de código compreensíveis por qualquer pessoa com uma base em qualquer uma das linguagens da tradição ALGOL, incluindo B, C, C++, C#, Objective-C, D, Java, Javascript e assim por diante. Os leitores que desejam obter detalhes completos de todas as implementações são encorajados a consultar o código fonte C++ que acompanha este livro.

Este livro mistura análises matemáticas dos tempos de execução com o código-fonte C++ para os algoritmos que estão sendo analisados. Isso significa que algumas equações contêm variáveis também encontradas no código-fonte. Essas variáveis são compostas de forma consistente, tanto no código-fonte quanto nas equações. A variável mais comum é a variável `n` que, sem exceção, sempre se refere ao número de itens atualmente armazenados na estrutura de dados.

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

Implementações de `USet`			
	`find(x)`	`add(x)`/`remove(x)`	
`ChainedHashTable`	$O(1)^{E}$	$O(1)^{A,E}$	§ 5.1
`LinearHashTable`	$O(1)^{E}$	$O(1)^{A,E}$	§ 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 simplificadas das interfaces `Lista`, `Set`, `Map`, `SortedSet` e `SortedMap` en-

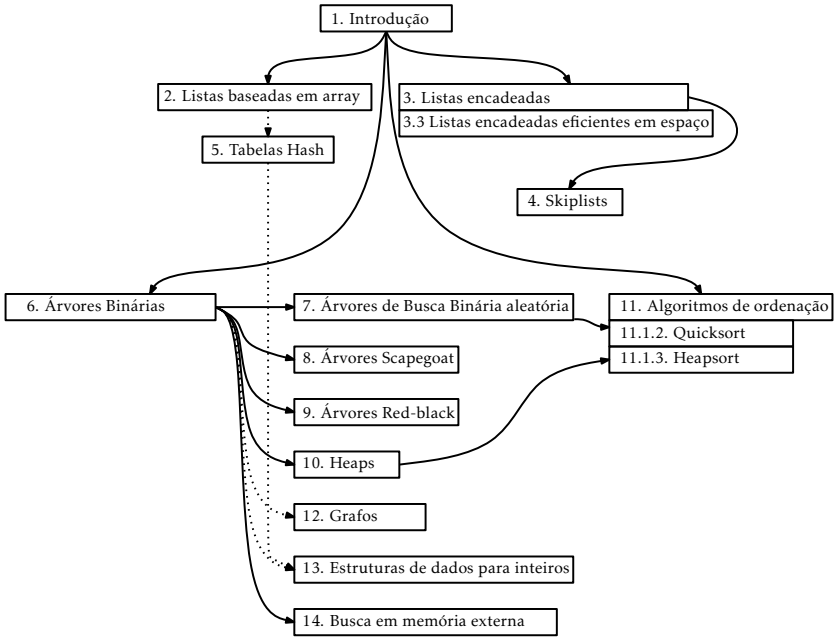


Figura 1.6: As dependências entre capítulos neste livro.

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			
	`findMin()`	`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.

contradas 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`) fornecida pelo C++ Standard Template Library.

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 findAll(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.

Capítulo 2

Listas Baseadas em Array

Neste capítulo, estudaremos as implementações das interfaces `Lista` e `Fila` nas quais os dados subjacentes são armazenados em um array, chamado de *array de base*. A tabela a seguir resume os tempos de execução das operações das estruturas de dados apresentadas neste capítulo:

	`get(i)`/`set(i,x)`	`add(i,x)`/`remove(i)`
`ArrayStack`	$O(1)$	$O(n-i)$
`ArrayDeque`	$O(1)$	$O(\min\{i,n-i\})$
`DualArrayDeque`	$O(1)$	$O(\min\{i,n-i\})$
`RootishArrayStack`	$O(1)$	$O(n-i)$

Estruturas de dados que funcionam armazenando dados em um único array têm muitas vantagens e limitações em comum:

- Os arrays oferecem acesso com tempo constante a qualquer valor no array. Isto é o que permite que `get(i)` e `set(i,x)` sejam executados em tempo constante.
- Arrays não são muito dinâmicos. Adicionar ou remover um elemento perto do meio de uma lista significa que um grande número de elementos no array precisa ser deslocado para abrir espaço para o elemento recém-adicionado ou para preencher a lacuna criada pelo elemento excluído. É por isso que as operações `add(i,x)` e `remove(i)` têm tempos de execução que dependem de `n` e `i`.
- Arrays não podem expandir ou encolher. Quando o número de elementos na estrutura de dados excede o tamanho do array de base,

um novo array precisa ser alocado e os dados do array antigo precisam ser copiados para o novo array. Esta é uma operação cara.

O terceiro ponto é importante. Os tempos de execução citados na tabela acima não incluem o custo associado ao crescimento e ao encolhimento do array de base. Veremos que, se cuidadosamente gerenciado, o custo de crescer e encolher o array de base não aumenta muito o custo de uma operação *média*. Mais precisamente, se começarmos com uma estrutura de dados vazia e executarmos qualquer sequência de m operações `add(i,x)` ou `remove(i)`, então o custo total do crescimento e encolhimento do array de base, sobre a sequência inteira de m operações é $O(m)$. Embora algumas operações individuais sejam mais caras, o custo amortizado, quando amortizado em todas as operações de m, é de apenas $O(1)$ por operação.

Neste capítulo, e ao longo deste livro, será conveniente ter arrays que acompanhem o seu tamanho. Os arrays usuais de C++ não fazem isso, então definimos uma classe, `array`, que mantém o controle de seu comprimento. A implementação desta classe é simples. Ela é implementada como um array padrão de C++ padrão, `a`, e um inteiro, `length`:

array
```
T *a;
int length;
```

O tamanho de um `array` é especificado no momento da criação:

array
```
array(int len) {
  length = len;
  a = new T[length];
}
```

Os elementos de uma array podem ser indexados:

array
```
T& operator[](int i) {
  assert(i >= 0 && i < length);
  return a[i];
}
```

Finalmente, quando um array é atribuído a outra, esta é apenas uma manipulação de ponteiro que leva tempo constante:

array

```
array<T>& operator=(array<T> &b) {
  if (a != NULL) delete[] a;
  a = b.a;
  b.a = NULL;
  length = b.length;
  return *this;
}
```

2.1 ArrayStack: Operações Rápidas de Pilha usando um Array

Um ArrayStack implementa a interface de lista usando um array a, chamado de *array de base*. O elemento de lista com índice i é armazenado em a[i]. Na maioria das vezes, a é maior do que o estritamente necessário, então um número inteiro n é usado para manter o controle do número de elementos realmente armazenados em a. Desta forma, os elementos da lista são armazenados em $a[0],\ldots,a[n-1]$ e, sempre, $a.\texttt{length} \geq n$.

ArrayStack

```
array<T> a;
int n;
int size() {
  return n;
}
```

2.1.1 O Básico

Acessar e modificar os elementos de um ArrayStack usando get(i) e set(i,x) é trivial. Depois de realizar qualquer verificação de limites necessária, simplesmente retornamos ou atribuímos, respectivamente, a[i].

ArrayStack

```
T get(int i) {
  return a[i];
}
T set(int i, T x) {
```

```
  T y = a[i];
  a[i] = x;
  return y;
}
```

As operações de adicionar e remover elementos de um `ArrayStack` são ilustradas na Figura 2.1. Para implementar a operação `add(i,x)`, verificamos primeiro se `a` já está cheio. Em caso afirmativo, chamamos o método `resize()` para aumentar o tamanho de `a`. Como `resize()` será implementado discutiremos mais tarde. Por enquanto, basta saber que, após uma chamada a `resize()`, podemos ter certeza de que `a.length` > `n`. Com isto resolvido, agora deslocamos os elementos `a[i]`,...,`a[n-1]` uma posição à direita para abrir espaço para `x`, fazemos `a[i]` igual a `x` e incrementamos `n`.

ArrayStack

```
void add(int i, T x) {
  if (n + 1 > a.length) resize();
  for (int j = n; j > i; j--)
    a[j] = a[j - 1];
  a[i] = x;
  n++;
}
```

Se ignorarmos o custo da possível chamada para `resize()`, então o custo da operação `add(i,x)` é proporcional ao número de elementos que temos de deslocar para criar espaço para `x`. Portanto, o custo desta operação (ignorando o custo de redimensionar `a`) é $O(n-i)$.

Implementar a operação `remove(i)` é semelhante. Deslocamos os elementos `a[i+1]`,...,`a[n-1]` uma posição para a esquerda (sobrescrevendo `a[i]`) e diminuindo o valor de `n`. Depois de fazer isso, verificamos se `n` está ficando muito menor que `a.length` verificando se `a.length` ≥ 3`n`. Em caso afirmativo, chamamos `resize()` para reduzir o tamanho de `a`.

ArrayStack

```
T remove(int i) {
  T x = a[i];
  for (int j = i; j < n - 1; j++)
```

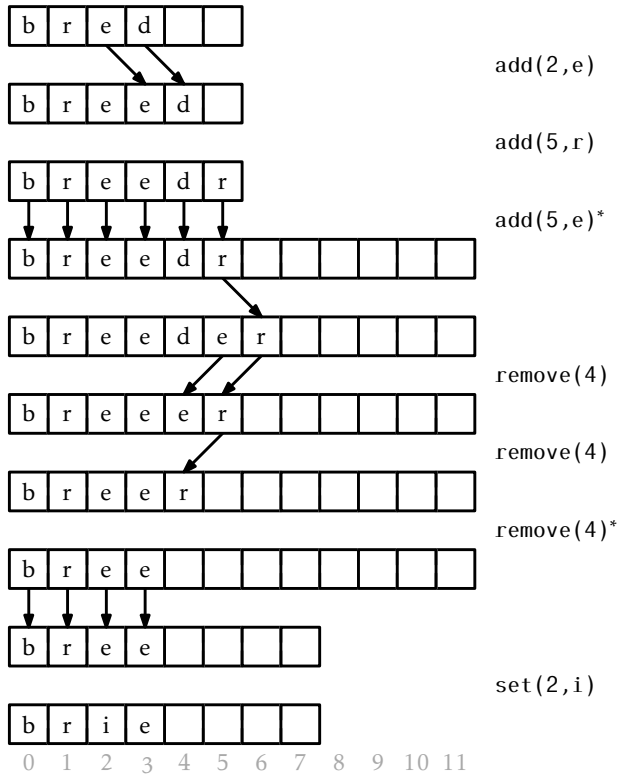


Figura 2.1: Uma sequência de operações `add(i,x)` e `remove(i)` em um `ArrayStack`. As setas indicam elementos que estão sendo copiados. As operações que resultam em uma chamada para `resize()` são marcadas com um asterisco.

```
    a[j] = a[j + 1];
  n--;
  if (a.length >= 3 * n) resize();
  return x;
}
```

Se ignorarmos o custo do método `resize()`, o custo de uma operação `remove(i)` é proporcional ao número de elementos que deslocamos, que é $O(n - i)$.

2.1.2 Crescendo e Encolhendo

O método `resize()` é bastante direto; ele aloca um novo array `b` cujo tamanho é $2n$ e copia os n elementos de `a` para as primeiras n posições em `b` e, em seguida, define `a` como `b`. Assim, depois de uma chamada para `resize()`, `a.length` $= 2n$.

ArrayStack

```
void resize() {
  array<T> b(max(2 * n, 1));
  for (int i = 0; i < n; i++)
    b[i] = a[i];
  a = b;
}
```

A análise do tempo de execução da seção anterior ignorou o custo de chamar o `resize()`. Nessa seção analisamos esse custo usando uma técnica chamada de *análise amortizada*. Esta técnica não tenta determinar o custo do resize em cada operação individual de `add(i,x)` e `remove(i)`. Em vez disso, ela considera o custo de todas as chamadas ao `resize()` numa sequência de m chamadas a `add(i,x)` ou `remove(i)`. Em particular, mostraremos:

Lema 2.1. *Se um `ArrayStack` vazio é criado e qualquer sequência de $m \geq 1$ chamadas a `add(i,x)` e `remove(i)` é executada, o tempo total gasto durante todas as chamadas a `resize()` é $O(m)$.*

Demonstração. Nós vamos mostrar que em qualquer momento que o `resize()` é chamado, o número de chamadas a `add` ou `remove` desde a última chamada a `resize()` é pelo menos $n/2 - 1$. Portanto, se n_i denota o valor de n

durante a i-ésima chamada a `resize()` e r denota o número de chamadas a `resize()`, então o número total de chamadas a `add(i,x)` ou `remove(i)` é pelo menos

$$\sum_{i=1}^{r}(n_i/2-1) \leq m \ ,$$

o que é equivalente a

$$\sum_{i=1}^{r} n_i \leq 2m+2r \ .$$

Por outro lado, o tempo total gasto durante todas as chamadas a `resize()` é

$$\sum_{i=1}^{r} O(n_i) \leq O(m+r) = O(m) \ ,$$

uma vez que r não é maior que m. Tudo que nos resta é mostrar que o o número de chamadas a `add(i,x)` ou `remove(i)` entre a $(i-1)$-ésima e a i-ésima chamada a `resize()` é de pelo menos $n_i/2$.

Existem dois casos a considerar. No primeiro caso, `resize()` está sendo chamado por `add(i,x)` porque o array de base `a` está cheio, i.e., `a.length` $= n = n_i$. Considere a chamada anterior a `resize()`: depois desta chamada, o tamanho de `a` era `a.length`, mas o número de elementos armazenados em `a` era no máximo `a.length`$/2 = n_i/2$. Porém agora o número de elementos armazenados em `a` é $n_i =$ `a.length`, então devem ter ocorrido pelo menos $n_i/2$ chamadas a `add(i,x)` desde a chamada anterior ao `resize()`.

O segundo caso ocorre quando `resize()` está sendo chamado pelo `remove(i)` porque `a.length` $\geq 3n = 3n_i$. Novamente, depois da chamada anterior ao `resize()` o número de elementos armazenados em `a` era pelo menos `a.length`$/2-1$.[1] Agora existem $n_i \leq$ `a.length`$/3$ elementos armazenados em `a`. Portanto, o número de operações `remove(i)` desde a última chamada ao `resize()` é de pelo menos

$$\begin{aligned} R &\geq \texttt{a.length}/2 - 1 - \texttt{a.length}/3 \\ &= \texttt{a.length}/6 - 1 \\ &= (\texttt{a.length}/3)/2 - 1 \\ &\geq n_i/2 - 1 \ . \end{aligned}$$

[1] O -1 nessa fórmula é responsável pelo caso especial que acontece quando $n = 0$ e `a.length` $= 1$.

Em ambos os casos, o número de chamadas ao add(i,x) ou remove(i) que ocorrem entre a $(i-1)$-ésima chamada a resize() e a i-ésima chamada a resize() é pelo menos $n_i/2-1$, como exigido para completar a prova. □

2.1.3 Resumo

O teorema a seguir resume o desempenho de um ArrayStack:

Teorema 2.1. *Um ArrayStack implementa a interface Lista. Ignorando o custo das chamadas a* resize(), *um ArrayStack suporta as operações*

- get(i) *e* set(i,x) *em um tempo* $O(1)$ *por operação; e*
- add(i,x) *e* remove(i) *em um tempo* $O(1+n-i)$ *por operação.*

Além disso, começando com um ArrayStack vazio e executando qualquer sequência de m *operações* add(i,x) *e* remove(i) *resulta em um tempo gasto total de* $O(m)$ *durante as chamadas a* resize().

O ArrayStack é um meio eficiente para implementar o Stack. Em particular, nós podemos implementar push(x) como add(n,x) e pop() como remove(n−1), em cada caso essas operações irão executar em um tempo amortizado de $O(1)$.

2.2 FastArrayStack: Um ArrayStack Otimizado

Grande parte do trabalho feito pelo ArrayStack envolve deslocamento (pelo add(i,x) e remove(i)) e cópias (pelo resize()) de dados. Nas implementações acima, isso foi feito usando loops for. Acontece que muitos ambientes de programação têm funções específicas que são muito eficientes em copiar e mover blocos de dados. Na linguagem de programação C, existem as funções memcpy(d,s,n) e memmove(d,s,n). A linguagem C++ tem o algoritmo std::copy(a0,a1,b). Em Java existe o método System.arraycopy(s,i,d,j,n) .

FastArrayStack

```
void resize() {
  array<T> b(max(1, 2*n));
```

```
  std::copy(a+0, a+n, b+0);
  a = b;
}
void add(int i, T x) {
  if (n + 1 > a.length) resize();
  std::copy_backward(a+i, a+n, a+n+1);
  a[i] = x;
  n++;
}
```

Essas funções são geralmente altamente otimizadas e podem ainda usar instruções de máquinas especiais que podem fazer essa cópia muito mais rápida do que usando um loop `for`. Embora o uso dessas funções não reduza assintóticamente o tempo de execução, ainda pode ser uma otimização que vale a pena.

Nas C++ implementações aqui, o uso do nativo `std::copy(a0,a1,b)` resultou em um aumento de velocidade de um fator entre 2 e 3, dependendo dos tipos de operações executadas. Os benefícios podem variar.

2.3 ArrayQueue: Uma Fila Baseada em Array

Nesta seção, nós apresentamos a estrutura de dados `ArrayQueue`, que implementa a fila FIFO (first-in-first-out); elementos são removidos (usando a operação `remove()`) da fila na mesma ordem em que são adicionados (usando a operação `add(x)`).

Note que um `ArrayStack` é uma má escolha para uma implementação de uma fila FIFO. Não é uma boa escolha pois devemos escolher um final da lista para adicionar elementos e então remover elementos do outro final. Uma das duas operações deve trabalhar no cabeçalho da lista, que envolve chamar `add(i,x)` ou `remove(i)` com um valor `i` = 0. Isso dá um tempo de operação proporcional a `n`.

Para obter uma implementação eficiente de uma fila, nós primeiro notamos que o problema seria fácil se tivéssemos um array infinito `a`. Poderíamos manter um índice `j` que mantém um registro para o próximo a ser removido e um inteiro `n` que conta o número de elementos na fila. Os

elementos da fila devem sempre ser armazenados em

$$a[j], a[j+1], \ldots, a[j+n-1] \ .$$

Inicialmente, ambos `j` e `n` seriam definidos como 0. Para adicionar um elemento, poderíamos colocá-lo em $a[j+n]$ e incrementar `n`. Para remover um elemento, nós o removeríamos de $a[j]$, incrementando `j`, e decrementando `n`.

Naturalmente, o problema com esta solução é que ela requer um array infinito. Um `ArrayQueue` simula isso usando um array finito `a` e *aritmética modular*. Este é o tipo de aritmética usada quando estamos falando sobre a hora do dia. Por exemplo 10:00 mais cinco horas dá 3:00. Formalmente, dizemos que

$$10 + 5 = 15 \equiv 3 \pmod{12} \ .$$

Nós lemos a última parte desta equação como "15 é congruente a 3 módulo 12." Podemos também tratar mod como um operador binário, de modo que

$$15 \bmod 12 = 3 \ .$$

De modo mais geral, para um número inteiro a e inteiro positivo m, $a \bmod m$ é o único inteiro $r \in \{0, \ldots, m-1\}$ de tal modo que $a = r + km$ para algum inteiro k. Menos formalmente, o valor r é o resto que obtemos quando dividimos a por m. Em muitas linguagens de programação, incluindo C++, o operador mod é representado usando o símbolo %.[2]

A aritmética modular é útil para simular um array infinito, posto que `i mod a.length` sempre dá um valor no intervalo $0, \ldots,$ `a.length` $- 1$. Usando a aritmética modular, podemos armazenar os elementos da fila nos locais do array

$$a[j\%a.length], a[(j+1)\%a.length], \ldots, a[(j+n-1)\%a.length] \ .$$

Isso trata o array `a` como um *array circular* em que os índices de array maiores do que `a.length` $- 1$ "retornam" para o início do array.

A única coisa que resta para se preocupar é ter o cuidado de que o número de elementos em `ArrayQueue` não exceda o tamanho de `a`.

[2] Isto às vezes é chamado de operador *acéfalo* mod, uma vez que não implementa corretamente o operador matemático mod quando o primeiro argumento é negativo.

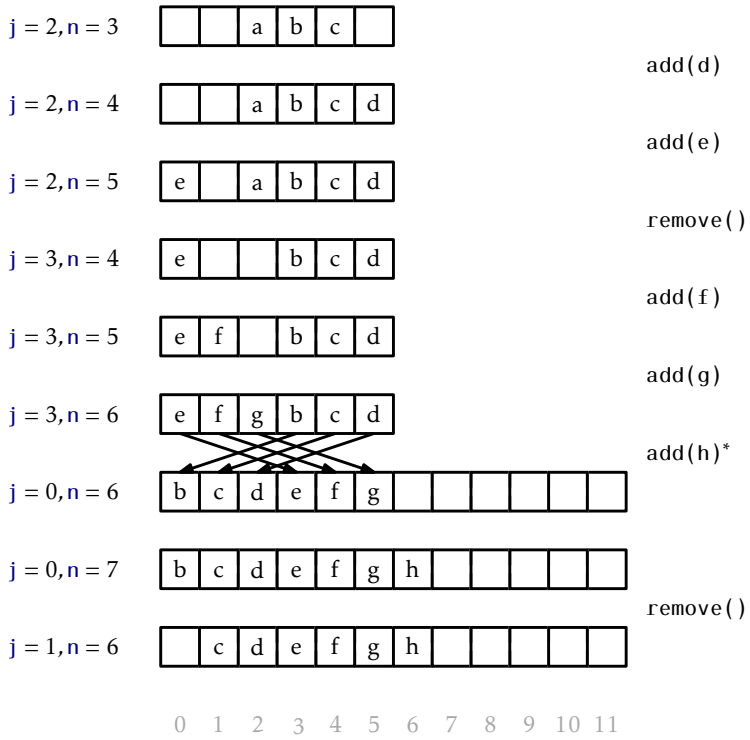


Figura 2.2: Sequência de operações add(x) e remove(i) em uma ArrayQueue. As setas indicam elementos que estão sendo copiados. Operações que resultam em uma chamada de resize() estão marcadas com um asterisco.

ArrayQueue

```
array<T> a;
int j;
int n;
```

Uma sequência de operações add(x) e remove() na ArrayQueue é ilustrada na Figura 2.2. Para implementar add(x), nós primeiro checamos se a está cheio e, se necessário, chamamos resize() para incrementar o tamanho de a. Em seguida, armazenamos x em a[(j+n)%a.length] e incrementamos n.

ArrayQueue
```
bool add(T x) {
  if (n + 1 > a.length) resize();
  a[(j+n) % a.length] = x;
  n++;
  return true;
}
```

Para implementar remove(), primeiro armazenamos a[j] para que possamos devolvê-lo mais tarde. Finalmente, decrementamos n e incrementamos j (modulo a.length) pela configuração $j = (j+1) \bmod a.length$. Finalmente, retornamos o valor armazenado de a[j]. Se necessário, podemos chamar resize() para diminuir o tamanho de a.

ArrayQueue
```
T remove() {
  T x = a[j];
  j = (j + 1) % a.length;
  n--;
  if (a.length >= 3*n) resize();
  return x;
}
```

Finalmente, a operação resize() é muito similar à operação resize() de ArrayStack. Aloca um novo array, b, de tamanho 2n e copia

$$a[j], a[(j+1)\%a.length], \ldots, a[(j+n-1)\%a.length]$$

para

$$b[0], b[1], \ldots, b[n-1]$$

e faz $j = 0$.

ArrayQueue
```
void resize() {
  array<T> b(max(1, 2*n));
  for (int k = 0; k < n; k++)
    b[k] = a[(j+k)%a.length];
  a = b;
  j = 0;
}
```

2.3.1 Resumo

O seguinte teorema resume o desempenho da estrutura de dados ArrayQueue:

Teorema 2.2. *Um* ArrayQueue *implementa a interface de* Fila *(FIFO). Ignorando o custo de chamada para* resize(), *um* ArrayQueue *suporta as operações* add(x) *e* remove() *com tempo por operação de* $O(1)$. *Além disso, começando com um* ArrayQueue *vazio, qualquer sequência de* m *operações* add(i,x) *e* remove(i) *resultam em um tempo gasto total de* $O(m)$ *durante todas as chamadas para* resize().

2.4 ArrayDeque: Operações Rápidas em um Deque Usando um Array

Um ArrayQueue da seção anterior é uma estrutura de dados para representar uma sequência que nos permite adicionar eficientemente a um extremidade da sequência e remover da outra extremidade. A estrutura de dados ArrayDeque permite uma adição e remoção eficientes em ambas as extremidades. Essa estrutura implementa a interface Lista usando a mesma técnica de array circular usada para representar um ArrayQueue.

ArrayDeque

```
array<T> a;
int j;
int n;
```

As operações get(i) e set(i,x) em um ArrayDeque são diretas. Elas obtêm ou definem o elemento do array a[(j + i) mod a.length].

ArrayDeque

```
T get(int i) {
  return a[(j + i) % a.length];
}
T set(int i, T x) {
  T y = a[(j + i) % a.length];
  a[(j + i) % a.length] = x;
```

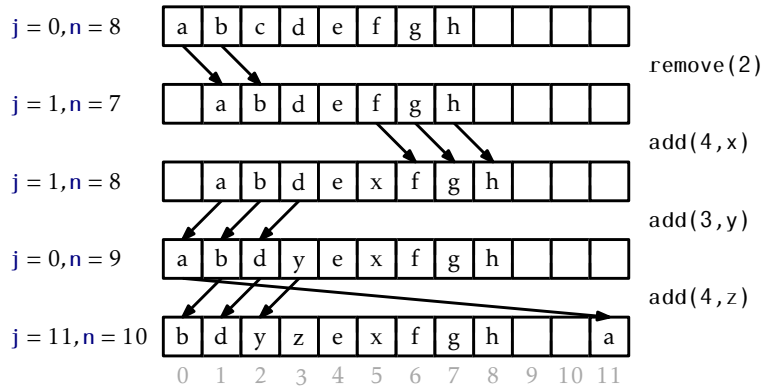


Figura 2.3: Uma sequência de operações `add(i,x)` e `remove(i)` em um `ArrayDeque`. As setas indicam elementos que estão sendo copiados.

```
  return y;
}
```

A implementação de `add(i,x)` é um pouco mais interessante. Como de costume, primeiro verifica se `a` está cheio e, se necessário, chama `resize()` para redimensionar `a`. Lembre-se que queremos que esta operação seja rápida quando `i` é pequeno (perto de 0) ou quando `i` é grande (perto de `n`). Portanto, verificamos se $i < n/2$. Se sim, deslocamos os elementos $a[0],\ldots,a[i-1]$ para a esquerda. Caso contrário ($i \geq n/2$), deslocamos os elementos $a[i],\ldots,a[n-1]$ para a direita. Veja Figura 2.3 para uma ilustração das operações `add(i,x)` e `remove(x)` em um `ArrayDeque`.

ArrayDeque

```
void add(int i, T x) {
  if (n + 1 > a.length)  resize();
  if (i < n/2) { // shift a[0],..,a[i-1] left one position
    j = (j == 0) ? a.length - 1 : j - 1;
    for (int k = 0; k <= i-1; k++)
      a[(j+k)%a.length] = a[(j+k+1)%a.length];
  } else { // shift a[i],..,a[n-1] right one position
    for (int k = n; k > i; k--)
      a[(j+k)%a.length] = a[(j+k-1)%a.length];
  }
```

```
  a[(j+i)%a.length] = x;
  n++;
}
```

Ao fazer o deslocamento desta maneira, garantimos que add(i,x) nunca tenha que deslocar mais de $\min\{i, n-i\}$ elementos. Assim, o tempo de execução da operação add(i,x) (ignorando o custo de uma operação resize()) é $O(1+\min\{i, n-i\})$.

A implementação da operação remove(i) é semelhante. Desloca os elementos $a[0],\ldots,a[i-1]$ à direita por uma posição ou desloca os elementos $a[i+1],\ldots,a[n-1]$ para esquerda por uma posição dependendo se $i < n/2$. Novamente, isso significa que remove(i) nunca gasta mais do que um tempo $O(1+\min\{i, n-i\})$ para deslocar elementos.

ArrayDeque

```
T remove(int i) {
  T x = a[(j+i)%a.length];
  if (i < n/2) { // shift a[0],..,[i-1] right one position
    for (int k = i; k > 0; k--)
      a[(j+k)%a.length] = a[(j+k-1)%a.length];
    j = (j + 1) % a.length;
  } else { // shift a[i+1],..,a[n-1] left one position
    for (int k = i; k < n-1; k++)
      a[(j+k)%a.length] = a[(j+k+1)%a.length];
  }
  n--;
  if (3*n < a.length) resize();
  return x;
}
```

2.4.1 Resumo

O seguinte teorema resume o desempenho da estrutura de dados ArrayDeque:

Teorema 2.3. *Um* ArrayDeque *implementa a interface* Lista. *Ignorando o custo das chamadas para* resize(), *um* ArrayDeque *suporta as operações*

- get(i) *e* set(i,x) *com tempo de* $O(1)$ *por operação; e*

- `add(i,x)` *e* `remove(i)` *com tempo* $O(1+\min\{i, n-i\})$ *por operação.*

Além disso, começando com um `ArrayDeque` *vazio, executar qualquer sequência de m operações* `add(i,x)` *e* `remove(i)` *resulta em um total de* $O(m)$ *de tempo gasto durante todas as chamadas para* `resize()`.

2.5 DualArrayDeque: Construindo um Deque com Duas Pilhas

Em seguida, apresentamos uma estrutura de dados, o DualArrayDeque que atinge os mesmos limites de desempenho que um ArrayDeque usando dois ArrayStacks. Embora o desempenho assintótico do DualArrayDeque não seja melhor do que o ArrayDeque, ainda vale a pena estudar, uma vez que oferece um bom exemplo de como fazer uma estrutura de dados sofisticada, combinando duas estruturas de dados mais simples.

O DualArrayDeque representa uma lista usando dois ArrayStacks. Lembre-se de que ArrayStack é rápido quando as operações nele modificam elementos perto do final. O DualArrayDeque coloca dois ArrayStacks, chamados de `front` e `back`, unidos pelos suas extremidades, para que as operações sejam rápidas em qualquer extremidade.

DualArrayDeque
```
ArrayStack<T> front;
ArrayStack<T> back;
```

O DualArrayDeque não armazena explicitamente o número, `n`, de elementos que ele contém. Ele não precisa, uma vez que contém `n` = `front.size()`+ `back.size()` elementos. No entanto, ao analisar o DualArrayDeque vamos ainda usar `n` para indicar o número de elementos que ele contém.

DualArrayDeque
```
int size() {
  return front.size() + back.size();
}
```

O `front` do ArrayStack armazena os elementos da lista cujos índices são $0,\ldots,$ `front.size()` -1, mas os armazena na ordem inversa. O `back` do

ArrayStack contém elementos da lista com índices em front.size(),...,size()−1 na ordem normal. Desta forma, get(i) e set(i,x) traduzem para chamadas apropriadas para get(i) ou set(i,x) em front ou back, levando um tempo de $O(1)$ por operação.

DualArrayDeque
```
T get(int i) {
  if (i < front.size()) {
    return front.get(front.size() - i - 1);
  } else {
    return back.get(i - front.size());
  }
}
T set(int i, T x) {
  if (i < front.size()) {
    return front.set(front.size() - i - 1, x);
  } else {
    return back.set(i - front.size(), x);
  }
}
```

Note que se um índice i < front.size(), então ele corresponde ao elemento de front na posição front.size()−i−1, uma vez que os elementos de front são armazenados na ordem inversa.

Adicionar e remover elementos de um DualArrayDeque é ilustrado na Figura 2.4. A operação add(i,x) manipula ou front ou back, conforme apropriado:

DualArrayDeque
```
void add(int i, T x) {
  if (i < front.size()) {
    front.add(front.size() - i, x);
  } else {
    back.add(i - front.size(), x);
  }
  balance();
}
```

O método add(i,x) realiza o reequilíbrio dos dois ArrayStacks front e back, chamando o método balance(). A implementação de balance()

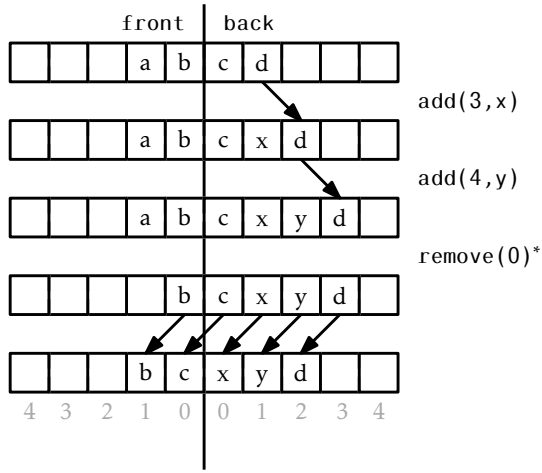


Figura 2.4: Uma sequência de operações add(i,x) e remove(i) em um DualArrayDeque. As setas indicam elementos que estão sendo copiados. Operações que resultam em um rebalanceamento por balance() são marcadas com um asterisco.

é descrita abaixo, mas por agora é suficiente saber que balance() garante que, a menos que size() < 2, front.size() e back.size() não diferem em mais de um fator de 3. Em particular, $3 \cdot$ front.size() $\geq$ back.size() e $3 \cdot$ back.size() $\geq$ front.size().

Em seguida, analisamos o custo de add(i,x), ignorando o custo das chamadas para balance(). Se i < front.size(), então add(i,x) é implementada pela chamada para front.add(front.size() − i − 1,x). Posto que front é um ArrayStack, o custo desta é

$$O(\texttt{front.size}() - (\texttt{front.size}() - \texttt{i} - 1) + 1) = O(\texttt{i} + 1) \ . \tag{2.1}$$

Por outro lado, se i ≥ front.size(), então add(i,x) é implementado como back.add(i − front.size(),x). O custo disso é

$$O(\texttt{back.size}() - (\texttt{i} - \texttt{front.size}())) + 1) = O(\texttt{n} - \texttt{i} + 1) \ . \tag{2.2}$$

Observe que o primeiro caso (2.1) ocorre quando i < n/4. O segundo caso (2.2) ocorre quando i ≥ 3n/4. Quando n/4 ≤ i < 3n/4, não podemos ter certeza se a operação afeta front ou back, mas em ambos os casos, a

operação leva um tempo $O(n) = O(i) = O(n-i)$, uma vez que $i \geq n/4$ e $n - i > n/4$. Resumindo a situação, temos

$$\text{Tempo de execução de } \texttt{add}(i,x) \leq \begin{cases} O(1+i) & \text{se } i < n/4 \\ O(n) & \text{se } n/4 \leq i < 3n/4 \\ O(1+n-i) & \text{se } i \geq 3n/4 \end{cases}.$$

Assim, o tempo de execução de `add(i,x)`, se ignorarmos o custo da chamada para `balance()`, é $O(1+\min\{i, n-i\})$.

A operação `remove(i)` e sua análise se assemelham à operação e análise de `add(i,x)`.

DualArrayDeque

```
T remove(int i) {
    T x;
    if (i < front.size()) {
            x = front.remove(front.size()-i-1);
    } else {
            x = back.remove(i-front.size());
    }
    balance();
    return x;
}
```

2.5.1 Balanceamento

Finalmente, voltamos para a operação `balance()` executada por `add(i,x)` e `remove(i)`. Esta operação garante que nem `front` nem `back` tornem-se muito grandes (ou muito pequenos). Garante que, a menos que haja menos de dois elementos, `front` e `back` contenham, cada um, pelo menos $n/4$ elementos. Se este não for o caso, então ela move elementos entre eles de modo que `front` e `back` contenham exatamente $\lfloor n/2 \rfloor$ elementos e $\lceil n/2 \rceil$ elementos, respectivamente.

DualArrayDeque

```
void balance() {
  if (3*front.size() < back.size()
      || 3*back.size() < front.size()) {
    int n = front.size() + back.size();
```

```
    int nf = n/2;
    array<T> af(max(2*nf, 1));
    for (int i = 0; i < nf; i++) {
      af[nf-i-1] = get(i);
    }
    int nb = n - nf;
    array<T> ab(max(2*nb, 1));
    for (int i = 0; i < nb; i++) {
      ab[i] = get(nf+i);
    }
    front.a = af;
    front.n = nf;
    back.a = ab;
    back.n = nb;
  }
}
```

Aqui há pouco para analisar. Se `balance()` faz o rebalanceamento, então ela move $O(n)$ elementos e isso leva um tempo $O(n)$. Isso é ruim uma vez que `balance()` é chamada juntamente com cada chamada de `add(i,x)` e `remove(i)`. Porém, o seguinte lema mostra que, em média, `balance()` só gasta uma quantidade constante de tempo por operação.

Lema 2.2. *Se um DualArrayDeque vazio for criado, e qualquer sequência de $m \geq 1$ chamadas de* `add(i,x)` *e* `remove(i)` *ocorrerem, então o tempo total gasto durante todas as chamadas de* `balance()` *é $O(m)$.*

Demonstração. Vamos mostrar que se `balance()` é forçada a deslocar elementos, então, o número de operações `add(i,x)` e `remove(i)` desde a última vez que quaisquer elementos foram deslocados por `balance()` é pelo menos $n/2 - 1$. Como na prova do Lema 2.1, isso é suficiente para provar que o tempo total gasto por `balance()` é $O(m)$.

Realizaremos nossa análise utilizando uma técnica conhecida como *método potencial*. Defina o *potencial*, Φ, do `DualArrayDeque` como a diferença de tamanho entre `front` e `back`:

$$\Phi = |\texttt{front.size()} - \texttt{back.size()}| \ .$$

O interessante sobre este potencial é que uma chamada de `add(i,x)` ou `remove(i)` que não faz nenhum balanceamento pode aumentar o potencial por no máximo 1.

Observe que, imediatamente após uma chamada a `balance()` que desloque elementos, o potencial, Φ_0, é pelo menos 1, posto que

$$\Phi_0 = |\lfloor n/2 \rfloor - \lceil n/2 \rceil| \leq 1 \ .$$

Considere a situação imediatamente antes de uma chamada `balance()` que desloca elementos, e suponha, sem perda de generalidade, que `balance()` está deslocando elementos porque $3\texttt{front.size()} < \texttt{back.size()}$. Observe que, neste caso,

$$\begin{aligned} \texttt{n} &= \texttt{front.size()} + \texttt{back.size()} \\ &< \texttt{back.size()}/3 + \texttt{back.size()} \\ &= \frac{4}{3}\texttt{back.size()} \end{aligned}$$

Além disso, o potencial neste momento é

$$\begin{aligned} \Phi_1 &= \texttt{back.size()} - \texttt{front.size()} \\ &> \texttt{back.size()} - \texttt{back.size()}/3 \\ &= \frac{2}{3}\texttt{back.size()} \\ &> \frac{2}{3} \times \frac{3}{4}\texttt{n} \\ &= \texttt{n}/2 \end{aligned}$$

Portanto, o número de chamadas `add(i,x)` ou `remove(i)` desde a última vez que `balance()` deslocou elementos é pelo menos $\Phi_1 - \Phi_0 > \texttt{n}/2 - 1$. Isso completa a prova. □

2.5.2 Resumo

O seguinte teorema resume as propriedades de um `DualArrayDeque`:

Teorema 2.4. *O `DualArrayDeque` implementa a interface `Lista`. Ignorando o custo de chamadas `resize()` e `balance()`, um `DualArrayDeque` suporta as operações*

- `get(i)` *e* `set(i,x)` *com um tempo* $O(1)$ *por operação; e*
- `add(i,x)` *e* `remove(i)` *com um tempo* $O(1 + \min\{\texttt{i}, \texttt{n} - \texttt{i}\})$ *por operação.*

Além disso, começando com um `DualArrayDeque` *vazio, qualquer sequência de m operações* `add(i,x)` *e* `remove(i)` *resulta em um tempo total gasto de* $O(m)$ *durante todas as chamadas a* `resize()` *e* `balance()`.

2.6 RootishArrayStack: Um Array Stack Eficiente em Espaço

Uma das desvantagens de todas as estruturas de dados anteriores neste capítulo é que, porque armazenam seus dados em um ou dois arrays e evitam o redimensionamento desses arrays com muita frequência, os arrays frequentemente não estão muito cheios. Por exemplo, imediatamente após uma operação `resize()` em um `ArrayStack`, o array de base `a` está apenas meio cheio. Pior ainda, há momentos no qual apenas um terço de `a` contém dados.

Nesta seção, discutimos a estrutura de dados `RootishArrayStack`, que aborda o problema do desperdício de espaço. O `RootishArrayStack` armazena `n` elementos usando $O(\sqrt{n})$ arrays. Nesses arrays, no máximo $O(\sqrt{n})$ locais do array ficam sem utilização. Todos os locais restantes são usados para armazenar dados. Portanto, essas estruturas de dados desperdiçam um espaço de no máximo $O(\sqrt{n})$ ao armazenar `n` elementos.

Uma `RootishArrayStack` armazena seus elementos em uma lista de `r` arrays chamados *blocos*, que são numerados $0, 1, \ldots, r-1$. Veja Figura 2.5. O bloco b contém $b+1$ elementos. Portanto, todos os blocos `r` contêm um total de

$$1 + 2 + 3 + \cdots + r = r(r+1)/2$$

elementos. A fórmula acima pode ser obtida como mostrado na Figura 2.6.

RootishArrayStack

```
ArrayStack<T*> blocks;
int n;
```

Como seria de esperar, os elementos da lista são dispostos dentro dos blocos. O elemento de lista com índice 0 é armazenado no bloco 0, os elementos com índices de lista 1 e 2 são armazenados no bloco 1, os elementos com índices de lista 3, 4 e 5 são armazenados no bloco 2 e assim

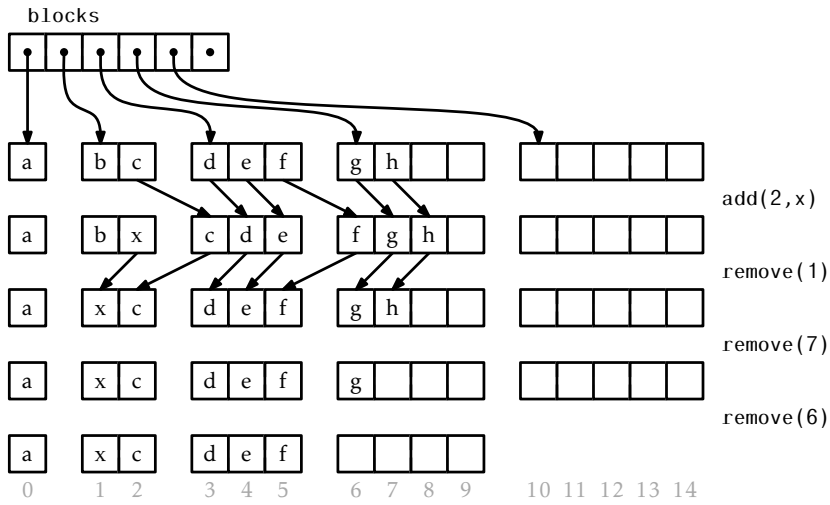


Figura 2.5: Uma sequência de operações `add(i,x)` e `remove(i)` em um `RootishArrayStack`. As flechas indicam elementos sendo copiados.

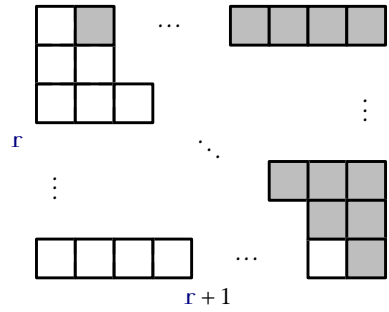


Figura 2.6: O número de quadrados brancos é $1 + 2 + 3 + \cdots + r$. O número de quadrados cinzas é o mesmo. Juntando os quadrados brancos e cinzas cria um retângulo consistindo de $r(r+1)$ quadrados.

por diante. O principal problema que temos de resolver é o de determinar, dado um índice `i`, qual bloco contém `i` bem como o índice correspondente ao `i` dentro desse bloco.

Determinar o índice de `i` dentro de seu bloco, acaba sendo fácil. Se o índice `i` está no bloco `b`, então o número de elementos nos blocos $0,\ldots,b-1$ é $b(b+1)/2$. Assim sendo, `i` é armazenado no local

$$j = i - b(b+1)/2$$

dentro do bloco `b`. Um pouco mais desafiador é o problema de determinar o valor de `b`. O número de elementos com índices inferiores ou iguais a `i` é $i+1$. Por outro lado, o número de elementos nos blocos $0,\ldots,b$ é $(b+1)(b+2)/2$. Assim sendo, `b` é o menor inteiro de tal modo que

$$(b+1)(b+2)/2 \geq i+1 \ .$$

Podemos reescrever essa equação como

$$b^2 + 3b - 2i \geq 0 \ .$$

A equação quadrática correspondente $b^2 + 3b - 2i = 0$ possui duas soluções: $b = (-3+\sqrt{9+8i})/2$ e $b = (-3-\sqrt{9+8i})/2$. A segunda solução não faz sentido em nossa aplicação, pois ela sempre dá um valor negativo. Assim sendo, obtemos a solução $b = (-3+\sqrt{9+8i})/2$. Em geral, essa solução não é um inteiro, mas voltando à nossa desigualdade, queremos o menor número inteiro `b` de tal modo que $b \geq (-3+\sqrt{9+8i})/2$. Isto é simplesmente

$$b = \left\lceil (-3+\sqrt{9+8i})/2 \right\rceil \ .$$

RootishArrayStack

```
int i2b(int i) {
  double db = (-3.0 + sqrt(9 + 8*i)) / 2.0;
  int b = (int)ceil(db);
  return b;
}
```

Com isso resolvido, os métodos `get(i)` e `set(i,x)` são mais fáceis. Primeiro, calculamos o bloco apropriado `b` e o índice apropriado `j` dentro do bloco e executamos a operação apropriada:

RootishArrayStack
```
T get(int i) {
  int b = i2b(i);
  int j = i - b*(b+1)/2;
  return blocks.get(b)[j];
}
T set(int i, T x) {
  int b = i2b(i);
  int j = i - b*(b+1)/2;
  T y = blocks.get(b)[j];
  blocks.get(b)[j] = x;
  return y;
}
```

Se usarmos qualquer estrutura de dados deste capítulo para representar a lista de `blocos`, então `get(i)` e `set(i,x)` funcionarão em tempo constante.

O método `add(i,x)` será, agora, familiar. Verificamos primeiro se a nossa estrutura de dados está cheia, verificando se o número de blocos, `r`, é tal que $r(r+1)/2 = n$. Se sim, chamamos `grow()` para adicionar outro bloco. Feito isso, deslocamos elementos com índices $i,\ldots,n-1$ para a direita de uma posição para dar espaço para o novo elemento com índice `i`:

RootishArrayStack
```
void add(int i, T x) {
  int r = blocks.size();
  if (r*(r+1)/2 < n + 1) grow();
  n++;
  for (int j = n-1; j > i; j--)
    set(j, get(j-1));
  set(i, x);
}
```

O método `grow()` faz o que esperamos. Adiciona um novo bloco:

RootishArrayStack
```
void grow() {
  blocks.add(blocks.size(), new T[blocks.size()+1]);
}
```

Ignorando o custo da operação grow(), o custo da operação add(i,x) é dominado pelo custo da mudança e é portanto $O(1+n-i)$, exatamente como um ArrayStack.

A operação remove(i) é similar a add(i,x). Desloca os elementos com índices $i+1,\ldots,n$ uma posição para a esquerda e então, se tiver mais de um bloco vazio, é chamado o método shrink() para remover todos exceto um dos blocos não usados:

RootishArrayStack
```
T remove(int i) {
  T x = get(i);
  for (int j = i; j < n-1; j++)
    set(j, get(j+1));
  n--;
  int r = blocks.size();
  if ((r-2)*(r-1)/2 >= n) shrink();
  return x;
}
```

RootishArrayStack
```
void shrink() {
    int r = blocks.size();
    while (r > 0 && (r-2)*(r-1)/2 >= n) {
            delete [] blocks.remove(blocks.size()-1);
            r--;
    }
}
```

Mais uma vez, ignorando o custo da operação shrink() , o custo da operação remove(i) é dominado pelo custo do deslocamento e é portanto $O(n-i)$.

2.6.1 Análise de Crescimento e Diminuição

A análise acima de add(i,x) e remove(i) não representa o custo de grow() e shrink(). Note que, diferentemente da operação ArrayStack.resize() , grow() e shrink() não copiam nenhum dado. Eles apenas alocam ou liberam um array de tamanho r. Em alguns ambientes, isso leva apenas um

tempo constante, enquanto em outros, pode requerer tempo proporcional a r.

Notamos que, imediatamente após chamar `grow()` ou `shrink()`, a situação é clara. O bloco final está completamente vazio, e todos os outros blocos estão completamente cheios. Outra chamada para `grow()` ou `shrink()` não irá acontecer até pelo menos $r-1$ elementos terem sido adicionados ou removidos. Assim sendo, mesmo que `grow()` e `shrink()` levem um tempo $O(r)$, esse custo pode ser amortizado por pelo menos $r-1$ operações `add(i,x)` ou `remove(i)`, de forma que o custo amortizado `grow()` e `shrink()` é $O(1)$ por operação.

2.6.2 Uso de Espaço

Em seguida, analisamos a quantidade extra de espaço usado pela `RootishArrayStack`. Em particular, queremos contar qualquer espaço usado pela `RootishArrayStack` que não seja um elemento do array atualmente usado para manter um elemento de lista. Podemos chamar tal espaço de *espaço desperdiçado*.

A operação `remove(i)` garante que um `RootishArrayStack` nunca tenha mais de dois blocos que não estejam completamente cheios. O número de blocos, r, usado por `RootishArrayStack` que armazena n elementos, portanto, satisfaz

$$(r-2)(r-1)/2 \le n \ .$$

Novamente, usando a equação quadrática nele fornece

$$r \le \frac{1}{2}\left(3+\sqrt{8n+1}\right) = O(\sqrt{n}) \ .$$

Os dois últimos blocos têm tamanhos r e $r-1$, então o espaço perdido por esses dois blocos é no máximo $2r-1 = O(\sqrt{n})$. Se armazenamos os blocos em (por exemplo) um `ArrayStack`, então a quantidade de espaço desperdiçado pela `Lista` que armazena esses r blocos também é $O(r) = O(\sqrt{n})$. O outro espaço necessário para armazenar n e outras informações contábeis é $O(1)$. Portanto, a quantidade total de espaço desperdiçado em um `RootishArrayStack` é $O(\sqrt{n})$.

Em seguida, argumentamos que este uso do espaço é ideal para qualquer estrutura de dados que começa vazia e pode suportar a adição de um

item de cada vez. Mais precisamente, mostraremos que, em algum ponto durante a adição de n itens, a estrutura de dados está desperdiçando um espaço de pelo menos $\sqrt{n}$ (embora possa estar desperdiçado apenas durante um momento).

Suponha que começamos com uma estrutura de dados vazia e adicionamos n itens, um de cada vez. No final deste processo, todos os n itens são armazenados na estrutura e distribuídos entre uma coleção de r blocos de memória. Se $r \geq \sqrt{n}$, então a estrutura de dados deve estar usando r ponteiros (ou referências) para acompanhar esses r blocos, e esses ponteiros são espaço desperdiçado. Por outro lado, se $r < \sqrt{n}$, então, pelo princípio da "casa de pombos", algum bloco deve ter tamanho de pelo menos $n/r > \sqrt{n}$. Considere o momento em que este bloco foi alocado pela primeira vez. Imediatamente após ele ter sido alocado, esse bloco estava vazio e, portanto, estava desperdiçando $\sqrt{n}$ de espaço. Portanto, em algum ponto no tempo durante a inserção dos elementos n, a estrutura de dados estava desperdiçando $\sqrt{n}$ de espaço.

2.6.3 Resumo

O seguinte teorema resume nossa discussão da estrutura de dados `RootishArrayStack`:

Teorema 2.5. *Um `RootishArrayStack` implementa a interface `Lista`. Ignorando o custo das chamadas para* `grow()` *e* `shrink()`*, um `RootishArrayStack` suporta as operações*

- `get(i)` *e* `set(i,x)` *com tempo* $O(1)$ *por operação; e*
- `add(i,x)` *e* `remove(i)` *com tempo* $O(1 + n - i)$ *por operação.*

Além disso, começando com um `RootishArrayStack` vazio, qualquer sequência de m *operações* `add(i,x)` *e* `remove(i)` *resulta em um tempo gasto total de* $O(m)$ *durante todas as chamadas para* `grow()` *e* `shrink()`*.*

O espaço (medido em palavras)[3] *usado por um `RootishArrayStack` que armazena* n *elementos é* $n + O(\sqrt{n})$*.*

[3] Reveja Seção 1.4 para uma discussão sobre como a memória é medida.

2.6.4 Calculando Raízes Quadradas

Um leitor que tenha tido alguma exposição a modelos de computação pode notar que o `RootishArrayStack`, como descrito acima, não se encaixa no modelo usual de palavra-RAM de computação (Seção 1.4) porque ele requer ter raízes quadradas. A operação de raiz quadrada geralmente não é considerada uma operação básica e, portanto, não é geralmente parte do modelo palavra-RAM.

Nesta seção, mostramos que a operação de raiz quadrada pode ser implementada de forma eficiente. Em particular, mostramos que para qualquer número inteiro $x \in \{0,\ldots,n\}$, $\lfloor\sqrt{x}\rfloor$ pode ser calculada em tempo constante, depois de um pré-processamento $O(\sqrt{n})$ que cria dois arrays de comprimento $O(\sqrt{n})$. O seguinte lema mostra que podemos reduzir o problema de calcular a raiz quadrada de x para a raiz quadrada de um valor relacionado x'.

Lema 2.3. *Seja $x \geq 1$ e seja $x' = x - a$, onde $0 \leq a \leq \sqrt{x}$. Então $\sqrt{x'} \geq \sqrt{x} - 1$.*

Demonstração. É suficiente mostrar que

$$\sqrt{x - \sqrt{x}} \geq \sqrt{x} - 1 \ .$$

Elevando ao quadrado ambos os lados da desigualdade, obtemos

$$x - \sqrt{x} \geq x - 2\sqrt{x} + 1$$

E reunindo termos para obter

$$\sqrt{x} \geq 1$$

Que é claramente verdade para qualquer $x \geq 1$. □

Comece restringindo o problema um pouco e suponha que $2^r \leq x < 2^{r+1}$, de modo que $\lfloor \log x \rfloor = r$, ou seja, x é um número inteiro com $r+1$ bits em sua representação binária. Podemos tomar $x' = x - (x \bmod 2^{\lfloor r/2 \rfloor})$. Agora, x' satisfaz as condições do Lema 2.3, então $\sqrt{x} - \sqrt{x'} \leq 1$. Além disso, x' tem todos os seus $\lfloor r/2 \rfloor$ bits de ordem inferior iguais a 0, portanto, há apenas

$$2^{r+1-\lfloor r/2 \rfloor} \leq 4 \cdot 2^{r/2} \leq 4\sqrt{x}$$

valores possíveis de x'. Isso significa que podemos usar um array, sqrttab, que armazena o valor de $\lfloor\sqrt{x'}\rfloor$ para cada possível valor de x'. Um pouco mais precisamente, temos

$$\texttt{sqrttab}[i] = \left\lfloor \sqrt{i2^{\lfloor r/2 \rfloor}} \right\rfloor \ .$$

Deste modo, sqrttab$[i]$ está 2 distante de $\sqrt{x}$ para todo $x \in \{i2^{\lfloor r/2 \rfloor}, \ldots, (i+1)2^{\lfloor r/2 \rfloor}-1\}$. Dito de outra forma, a entrada do array s = sqrttab[x>>⌊r/2⌋] é igual a $\lfloor\sqrt{x}\rfloor$, $\lfloor\sqrt{x}\rfloor - 1$, ou $\lfloor\sqrt{x}\rfloor - 2$. A partir de s podemos determinar o valor de $\lfloor\sqrt{x}\rfloor$ pelo incremento de s até $(s+1)^2 > x$.

FastSqrt

```
int sqrt(int x, int r) {
  int s = sqrtab[x>>r/2];
  while ((s+1)*(s+1) <= x) s++; // executes at most twice
  return s;
}
```

Agora, isso só funciona para $x \in \{2^r, \ldots, 2^{r+1}-1\}$ e sqrttab é uma tabela especial que só funciona para um valor específico de $r = \lfloor \log x \rfloor$. Para superar isso, poderíamos calcular $\lfloor \log n \rfloor$ diferentes arrays sqrttab, um para cada possível valor de $\lfloor \log x \rfloor$. Os tamanhos dessas tabelas formam uma sequência exponencial cujo maior valor é no máximo $4\sqrt{n}$, então o tamanho total de todas as tabelas é $O(\sqrt{n})$.

No entanto, verifica-se que mais de um array sqrttab é desnecessário; só precisamos de um array sqrttab para o valor $r = \lfloor \log n \rfloor$. Qualquer valor x com $\log x = r' < r$ pode ser *promovido* multiplicando x por $2^{r-r'}$ e usando a equação

$$\sqrt{2^{r-r'}x} = 2^{(r-r')/2}\sqrt{x} \ .$$

A quantidade $2^{r-r'}x$ está no intervalo $\{2^r, \ldots, 2^{r+1}-1\}$ de modo que podemos procurar sua raiz quadrada em sqrttab. O código a seguir implementa essa ideia para calcular $\lfloor\sqrt{x}\rfloor$ para todos os inteiros não-negativos x no intervalo $\{0, \ldots, 2^{30}-1\}$ usando um array, sqrttab, de tamanho 2^{16}.

FastSqrt

```
int sqrt(int x) {
  int rp = log(x);
  int upgrade = ((r-rp)/2) * 2;
```

```
  int xp = x << upgrade;  // xp has r or r-1 bits
  int s = sqrtab[xp>>(r/2)] >> (upgrade/2);
  while ((s+1)*(s+1) <= x) s++;  // executes at most twice
  return s;
}
```

Algo que tomamos como certo até agora é a questão de como calcular $r' = \lfloor \log x \rfloor$. Novamente, este é um problema que pode ser resolvido com um array, `logtab`, de tamanho $2^{r/2}$. Neste caso, o código é particularmente simples, uma vez que $\lfloor \log x \rfloor$ é apenas o índice do bit 1 mais significativo na representação binária de `x`. Isto significa que, para $x > 2^{r/2}$, podemos deslocar o lado direito dos bits de `x` por $r/2$ posições antes de usá-lo como um índice em `logtab`. O código a seguir usa um array `logtab` de tamanho 2^{16} para calcular $\lfloor \log x \rfloor$ para todos `x` no intervalo $\{1, \ldots, 2^{32} - 1\}$.

FastSqrt

```
int log(int x) {
  if (x >= halfint)
    return 16 + logtab[x>>16];
  return logtab[x];
}
```

Finalmente, para completar, incluímos o seguinte código que inicializa `logtab` e `sqrttab`:

FastSqrt

```
void inittabs() {
  sqrtab = new int[1<<(r/2)];
  logtab = new int[1<<(r/2)];
  for (int d = 0; d < r/2; d++)
    for (int k = 0; k < 1<<d; k++)
      logtab[1<<d+k] = d;
  int s = 1<<(r/4);                    // sqrt(2^(r/2))
  for (int i = 0; i < 1<<(r/2); i++) {
    if ((s+1)*(s+1) <= i << (r/2)) s++; // sqrt increases
    sqrtab[i] = s;
  }
}
```

Resumindo, os cálculos feitos pelo método `i2b(i)` podem ser implementados em tempo constante no word-RAM usando $O(\sqrt{n})$ memória extra

para armazenar os arrays `sqrttab` e `logtab`. Esses arrays podem ser reconstruídos quando `n` aumenta ou diminui por um fator de dois, e o custo desta reconstrução pode ser amortizado ao longo do número de operações `add(i,x)` e `remove(i)` que causaram a alteração em `n` da mesma forma que o custo de `resize()` é analisado na implementação `ArrayStack`

2.7 Discussões e Exercícios

A maioria das estruturas de dados descritas neste capítulo são tradicionais. Elas podem ser encontrados em implementações que datam de mais de 30 anos. Por exemplo, as implementações de pilhas, filas e deques, que generalizam facilmente as estruturas `ArrayStack`, `ArrayQueue` e `ArrayDeque` descritas aqui, são discutidas por Knuth [46, Section 2.2.2].

Brodnik *et al.* [13] parece ter sido o primeiro a descrever o `RootishArrayStack` e provar um limite inferior de $\sqrt{n}$ assim na Seção 2.6.2. Eles também apresentam uma estrutura diferente que usa uma escolha mais sofisticada de tamanhos de bloco para evitar a computação de raízes quadradas no método `i2b(i)`. Dentro de seu esquema, o bloco contendo `i` é o bloco $\lfloor \log(i+1) \rfloor$, que é simplesmente o índice do bit 1 mais significativo na representação binária de $i+1$. Algumas arquiteturas de computador fornecem uma instrução para calcular o índice do bit 1 mais significativo em um inteiro.

Uma estrutura relacionada ao `RootishArrayStack` é o *vetor em camadas* de dois níveis de Goodrich e Kloss [35]. Essa estrutura suporta as operações `get(i,x)` e `set(i,x)` em tempo constante e `add(i,x)` e `remove(i)` em $O(\sqrt{n})$. Esses tempos de execução são semelhantes aos que podem ser alcançados com a implementação mais cuidadosa de um `RootishArrayStack` discutido em Exercício 2.10.

Exercício 2.1. O método de `Lista` `addAll(i,c)` insere todos os elementos do `Collection` `c` na lista na posição `i`. (O método `add(i,x)` é um caso especial onde `c` = {`x`}.) Explique porque, para as estruturas de dados neste capítulo, não é eficiente implementar `addAll(i,c)` por chamadas repetidas para `add(i,x)`. Conceber e implementar uma implementação mais eficiente.

Exercício 2.2. Crie e implemente um *RandomQueue*. Esta é uma implementação da interface `Fila` na qual a operação `remove()` remove um elemento que é escolhido uniformemente ao acaso entre todos os elementos atualmente na fila. (Pense em uma `RandomQueue` como um saco em que podemos adicionar elementos ou alcançar e remover às cegas algum elemento aleatório.) As operações `add(x)` e `remove()` na `RandomQueue` devem ser executadas em tempo amortizado constante por operação.

Exercício 2.3. Projete e implemente uma `Treque` (fila triplamente terminada). Esta é uma implementação de `Lista` em que `get(i)` e `set(i,x)` são executadas em tempo constante e `add(i,x)` e `remove(i)` executam no tempo

$$O(1+\min\{i, n-i, |n/2-i|\}) \ .$$

Em outras palavras, as modificações são rápidas se estiverem perto de uma das extremidades ou perto do meio da lista.

Exercício 2.4. Implementar um método `rotate(a,r)` que "gira" o array `a` para que `a[i]` se mova para `a[(i+r) mod a.length]`, para todo $i \in \{0,\ldots,a.length\}$.

Exercício 2.5. Implemente um método `rotate(r)` que "gire" uma `Lista` para que o item de lista `i` se torne o item de lista $(i+r) \bmod n$. Quando executado em um `ArrayDeque`, ou um `DualArrayDeque`, `rotate(r)` deve ser executado em tempo $O(1+\min\{r, n-r\})$.

Exercício 2.6. Modifique a implementação de `rrayDeque` para que o deslocamento feito por `add(i,x)`, `remove(i)` e `resize()` seja feito usando o método `System.arraycopy(s,i,d,j,n)` mais rápido.

Exercício 2.7. Modifique a implementação `ArrayDeque` para que ele não use o operador % (que tem alto custo em alguns sistemas). Em vez disso, deve fazer uso do fato de que, se `a.length` é uma potência de 2, então

$$k\%a.length = k\&(a.length-1) \ .$$

(Aqui, & é um operador bit-a-bit.)

Exercício 2.8. Projete e implemente uma variante de `ArrayDeque` que não faça nenhuma aritmética modular. Em vez disso, todos os dados ficam em

blocos consecutivos, ordenados, dentro de um array. Quando os dados excedem o início ou o fim desse array, uma operação `rebuild()` modificada é executada. O custo amortizado de todas as operações deve ser o mesmo que em um `ArrayDeque`.
Dica: Conseguir que isso funcione diz respeito realmente a sobre como você implementa a operação `rebuild()`. Você gostaria que `rebuild()` colocasse a estrutura de dados em um estado onde os dados não podem ultrapassar qualquer final até que pelo menos $n/2$ operações sejam realizadas.

Teste o desempenho da sua implementação com o `ArrayDeque`. Otimize sua implementação (usando `System.arraycopy(a,i,b,i,n)`) e veja se você pode superar a implementação de `ArrayDeque`.

Exercício 2.9. Crie e implemente uma versão de um `RootishArrayStack` que tenha apenas $O(\sqrt{n})$ de espaço desperdiçado, mas que possa executar as operações `add(i,x)` e `remove(i,x)` e tempo $O(1+\min\{i, n-i\})$.

Exercício 2.10. Crie e implemente uma versão de um `RootishArrayStack` que tenha apenas $O(\sqrt{n})$ de espaço desperdiçado, mas que possa executar as operações `add(i,x)` e `remove(i,x)` em um tempo $O(1+\min\{\sqrt{n}, n-i\})$. (Para uma idéia sobre como fazer isso, veja Seção 3.3.)

Exercício 2.11. Crie e implemente uma versão de um `RootishArrayStack` que tenha apenas $O(\sqrt{n})$ de espaço desperdiçado, mas que possa executar as operações `add(i,x)` e `remove(i,x)` em um tempo $O(1+\min\{i, \sqrt{n}, n-i\})$. (Veja Seção 3.3 para obter ideias sobre como conseguir isso.)

Exercício 2.12. Crie e implemente um `CubishArrayStack`. Essa estrutura de três níveis implementa a interface `Lista` com um desperdício de espaço de $O(n^{2/3})$. Nesta estrutura, `get(i)` e `set(i,x)` tomam tempo constante; enquanto `add(i,x)` e `remove(i)` tomam um tempo amortizado de $O(n^{1/3})$.

Capítulo 3

Listas Encadeadas

Neste capítulo, continuamos a estudar implementações da interface `List`, desta vez utilizando estruturas de dados baseadas em ponteiro em vez de arrays. As estruturas neste capítulo são constituídas por nós que contêm os itens da lista. Usando referências (ponteiros), os nós são encadeados em uma sequência. Primeiro, estudamos listas simplesmente encadeadas, que podem implementar as operações de uma `Stack` e de uma `Queue` (FIFO) em tempo constante por operação e, em seguida, passamos para listas duplamente encadeadas, que podem implementar operações de `Deque` em tempo constante.

As listas encadeadas têm vantagens e desvantagens quando comparadas com implementações baseadas em array da interface `List`. A principal desvantagem é que perdemos a capacidade de acessar qualquer elemento usando `get(i)` ou `set(i,x)` em tempo constante. Em vez disso, temos de percorrer a lista, um elemento de cada vez, até chegar ao `i`-ésimo elemento. A principal vantagem é que elas são mais dinâmicas: dada uma referência a qualquer nó de lista `u`, podemos apagar `u` ou inserir um nó adjacente a `u` em tempo constante. Isso é verdade, não importa onde `u` esteja na lista.

3.1 SLList: Uma Lista Simplesmente Encadeada

Uma `SLList` (lista simplesmente encadeada) é uma sequência de *Nó*s. Cada nó `u` armazena um valor de dados `u.x` e uma referência `u.next` para o

próximo nó na sequência. Para o último nó `w` na sequência, `w.next = null`

SLList
```
class Node {
public:
  T x;
  Node *next;
  Node(T x0) {
    x = x0;
    next = NULL;
  }
};
```

Para eficiência, um `SLList` usa as variáveis `head` e `tail` para manter o registro do primeiro e do último nó na sequência, bem como um número inteiro `n` para acompanhar o tamanho da sequência:

SLList
```
Node *head;
Node *tail;
int n;
```

Uma sequência de operações em um `Stack` e uma `Queue` em um `SLList` é ilustrada na Figura 3.1.

Uma `SLList` pode implementar eficientemente as operações `push()` e `pop()` de uma `Stack`, adicionando e removendo elementos na cabeça da sequência. A operação `push()` simplesmente cria um novo nó `u` com valor de dados `x`, define `u.next` no cabeçalho antigo da lista e torna `u` o novo cabeçalho da lista. Finalmente, ele incrementa `n`, uma vez que o tamanho da `SLList` aumentou em um:

SLList
```
T push(T x) {
  Node *u = new Node(x);
  u->next = head;
  head = u;
  if (n == 0)
    tail = u;
  n++;
  return x;
}
```

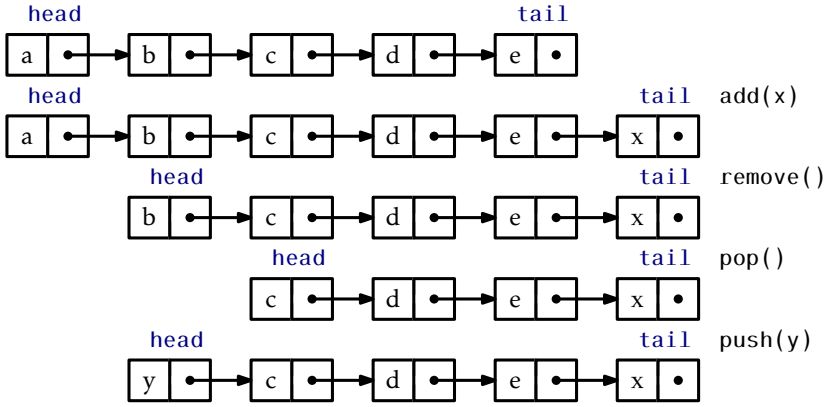


Figura 3.1: Uma sequência de operações de Queue (add(x) e remove()) e de Stack (push(x) e pop()) em uma SLList.

A operação pop(), depois de verificar que a SLList não está vazia, remove a cabeça definindo head = head.next e decrementando n. Um caso especial ocorre quando o último elemento está sendo removido, caso em que tail é definido como null:

SLList

```
T pop() {
  if (n == 0)  return null;
  T x = head->x;
  Node *u = head;
  head = head->next;
  delete u;
  if (--n == 0) tail = NULL;
  return x;
}
```

Claramente, as operações push(x) e pop() são executadas em tempo $O(1)$.

3.1.1 Operações de Fila

Uma SLList também pode implementar as operações de fila FIFO, add(x) e remove(), em tempo constante. As remoções são feitas a partir da cabeça da lista e são idênticas à operação pop():

SLList
```
T remove() {
  if (n == 0)  return null;
  T x = head->x;
  Node *u = head;
  head = head->next;
  delete u;
  if (--n == 0) tail = NULL;
  return x;
}
```

Adições, por outro lado, são feitas no final da lista. Na maioria dos casos, isso é feito definindo tail.next = u, onde u é o nó recém-criado que contém x. No entanto, um caso especial ocorre quando n = 0, caso em que tail = head = null. Nesse caso, tanto tail como head são definidos como u.

SLList
```
bool add(T x) {
  Node *u = new Node(x);
  if (n == 0) {
    head = u;
  } else {
    tail->next = u;
  }
  tail = u;
  n++;
  return true;
}
```

Claramente, ambos add(x) e remove() levam tempo constante.

3.1.2 Resumo

O seguinte teorema resume o desempenho de uma SLList:

Teorema 3.1. *Uma SLList implementa a interface para Stack e (FIFO) Queue. As operações* push(x), pop(), add(x) *e* remove() *são executadas em um tempo $O(1)$ por operação.*

Uma SLList quase implementa o conjunto completo de operações de uma Deque. A única operação que falta é a remoção da cauda de uma SLList. Remover a cauda de uma SLList é difícil porque requer a atualização do valor da tail para que ele aponte para o nó w que precede tail na SLList; este é o nó w tal que w.next = tail. Infelizmente, a única maneira de chegar ao w é atravessar a SLList começando em head e tomando n – 2 passos.

3.2 DLList: Uma lista duplamente encadeada

A DLList (lista duplamente encadeada) é muito semelhante a uma SLList, exceto que cada nó u em uma DLList tem referências tanto ao nó u.next que o sucede, quanto ao nó u.prev que o precede.

DLList

```
struct Node {
  T x;
  Node *prev, *next;
};
```

Ao implementar uma SLList, vimos que sempre havia vários casos especiais para se preocupar. Por exemplo, remover o último elemento ou adicionar um elemento vazio a uma SLList requer cuidado para garantir que head e tail sejam atualizados corretamente. Numa DLList, o número destes casos especiais aumenta consideravelmente. Talvez a maneira mais limpa de cuidar de todos esses casos especiais numa DLList é introduzir um nó dummy. Este é um nó que não contém quaisquer dados, mas age como um espaço reservado para que não haja nós especiais; cada nó tem um next e um prev, com o dummy agindo como o nó que sucede

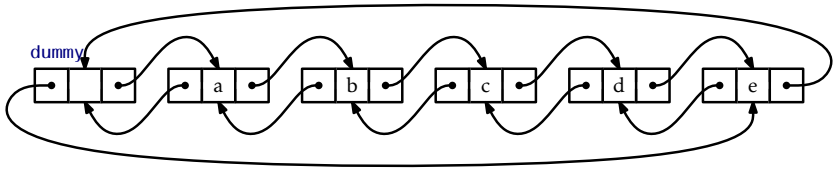


Figura 3.2: Uma DLList contendo a,b,c,d,e.

o último nó na lista e que precede o primeiro nó na lista. Desta forma, os nós da lista são (duplamente) ligados em um ciclo, como ilustrado na Figura 3.2.

DLList

```
Node dummy;
int n;
DLList() {
  dummy.next = &dummy;
  dummy.prev = &dummy;
  n = 0;
}
```

Encontrar um nó com um índice específico em uma DLList é fácil. Podemos começar na cabeça da lista (dummy.next) e trabalhar para a frente, ou começar no final da lista (dummy.prev) e trabalhar para trás. Isso nos permite alcançar o i-ésimo nó em $O(1+\min\{i, n-i\})$:

DLList

```
Node* getNode(int i) {
  Node* p;
  if (i < n / 2) {
    p = dummy.next;
    for (int j = 0; j < i; j++)
      p = p->next;
  } else {
    p = &dummy;
    for (int j = n; j > i; j--)
      p = p->prev;
  }
  return (p);
```

```
}
```

As operações `get(i)` e `set(i,x)` agora também são fáceis. Em primeiro lugar, localizamos o `i`-ésimo nó e depois obtemos(get) ou definimos(set) seu valor `x`:

DLList

```
T get(int i) {
    return getNode(i)->x;
}
T set(int i, T x) {
  Node* u = getNode(i);
  T y = u->x;
  u->x = x;
  return y;
}
```

O tempo de execução dessas operações é dominado pelo tempo que leva para encontrar o `i`-ésimo nó, sendo assim, $O(1+\min\{i, n-i\})$.

3.2.1 Adicionando e Removendo

Se tivermos uma referência a um nó `w` numa `DLList` e quisermos inserir um nó `u` antes de `w`, então isto é apenas uma questão de definir `u.next = w`, `u.prev = w.prev` e, em seguida, ajustar `u.prev.next` e `u.next.prev`. (Ver Figura 3.3.) Graças ao nó dummy, não há necessidade de se preocupar se `w.prev` ou `w.next` não existam.

DLList

```
Node* addBefore(Node *w, T x) {
  Node *u = new Node;
  u->x = x;
  u->prev = w->prev;
  u->next = w;
  u->next->prev = u;
  u->prev->next = u;
  n++;
  return u;
}
```

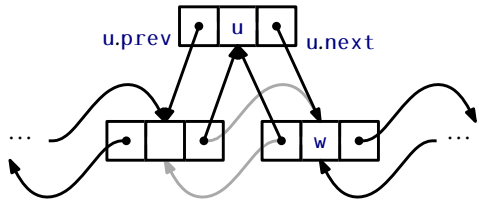


Figura 3.3: Adicionando o nó `u` antes do nó `w` em uma `DLList`.

Dessa forma, a operação de lista `add(i,x)` se torna trivial para implementar. Encontramos o `i`-ésimo nó na `DLList` e inserimos um novo nó `u` que contém `x` imediatamente antes dele.

DLList

```
void add(int i, T x) {
  addBefore(getNode(i), x);
}
```

A única parte não constante do tempo de execução de `add(i,x)` é o tempo necessário para encontrar o `i`-ésimo nó (usando `getNode(i)`). Assim, `add(i,x)` é executado em $O(1+\min\{i,n-i\})$.

Remover um nó `w` da `DLList` é fácil. Nós só precisamos ajustar os ponteiros em `w.next` e `w.prev` para que eles pulem `w`. Novamente, o uso do nó dummy elimina a necessidade de considerar quaisquer casos especiais:

DLList

```
void remove(Node *w) {
  w->prev->next = w->next;
  w->next->prev = w->prev;
  delete w;
  n--;
}
```

Agora a operação `remove(i)` é trivial. Achamos o nó com índice `i` e removemos:

DLList

```
T remove(int i) {
  Node *w = getNode(i);
```

```
  T x = w->x;
  remove(w);
  return x;
}
```

Novamente, a única parte cara dessa operação é achar o `i`-ésimo nó usando `getNode(i)`, então `remove(i)` roda em $O(1+\min\{i, n-i\})$.

3.2.2 Resumo

O seguinte teorema resume o desempenho de uma `DLList`:

Teorema 3.2. *A `DLList` implementa a interface de uma `Lista`. Na implementação, as operações `get(i)`, `set(i,x)`, `add(i,x)` e `remove(i)` executam em $O(1+\min\{i, n-i\})$ por operação.*

Vale notar que, se ignorarmos o custo da operação `getNode(i)`, então todas as operações da `DLList` levam tempo constante. Portanto, a única parte cara das operações na `DLList` é achar o nó relevante. Uma vez tenhamos o nó relevante, adicionar, remover, ou acessar os dados no nó leva tempo constante.

Isso contrasta claramente com as implementações da `Lista` baseada em array do Capítulo 2; nessas implementações, o item relevante no array pode ser encontrado com tempo constante. Entretanto, adicionar ou remover requer uma mudança de elementos no array e, em geral, leva um tempo não constante.

Por essa razão, as estruturas de lista encadeadas são bem adequadas para aplicações em que as referências a nós de lista podem ser obtidas por meios externos. Por exemplo, ponteiros para os nós de uma lista encadeada podem ser guardados em uma `USet`. Então, para remover um item `x` de uma lista encadeada, o nó que contém `x` pode ser rapidamente encontrado usando `Uset` e o nó pode ser removido da lista em tempo constante.

3.3 SEList: Uma Lista Encadeada Eficiente em Espaço

Uma das desvantagens das listas encadeadas (além do tempo que leva para acessar elementos que estão no final da lista) é o seu uso de espaço. Cada nó na DLList requer duas referências adicionais para o próximo nó e o anterior na lista. Dois desses campos no nó Node são dedicados a manter a lista, e só um dos campos serve para armazenar dados!

Uma SEList (lista eficiente em espaço) reduz o desperdício de espaço usando uma ideia simples: em vez de guardar os elementos individualmente numa DLList, guardamos um bloco (array) contendo vários itens. Mais precisamente, a SEList é parametrizada pelo *tamanho do bloco* b. Cada nó individual em uma SEList guarda um bloco que suporta até b + 1 elementos.

Por razões que ficarão claras mais à frente, será útil se pudermos fazer operações do Deque em cada bloco. A estrutura de dados que escolhemos para isso é a BDeque (bounded deque), derivada da ArrayDeque, estrutura descrita na Seção 2.4. O BDeque se diferencia do ArrayDeque numa pequena coisa: quando um novo BDeque é criado, o tamanho do array de suporte a é fixado em b + 1 e nunca cresce ou encolhe. A propriedade importante do BDeque é que ele permite adição e remoção de elementos por qualquer dos lados em tempo constante. Isso será útil quando elementos forem trocados de um bloco para outro.

SEList

```
class BDeque : public ArrayDeque<T> {
public:
  BDeque(int b) {
    n = 0;
    j = 0;
    array<int> z(b+1);
    a = z;
  }
  ~BDeque() { }
  // C++ Question: Why is this necessary?
  void add(int i, T x) {
    ArrayDeque<T>::add(i, x);
  }
  bool add(T x) {
```

```
    ArrayDeque<T>::add(size(), x);
    return true;
  }
  void resize() {}
};
```

Uma SEList é então uma lista duplamente encadeada de blocos:

SEList

```
class Node {
public:
  BDeque d;
  Node *prev, *next;
  Node(int b) : d(b) { }
};
```

SEList

```
int n;
Node dummy;
```

3.3.1 Requisitos de Espaço

Uma SEList coloca restrições rígidas sobre o número de elementos em um bloco: a menos que um bloco seja o último bloco, então esse bloco contém pelo menos $b-1$ e no máximo $b+1$ elementos. Isto significa que, se um SEList contém n elementos, então ele tem no máximo

$$n/(b-1)+1 = O(n/b)$$

blocos. A BDeque para cada bloco contém um array de comprimento $b+1$ mas, para cada bloco exceto o último, no máximo uma quantidade constante de espaço é desperdiçada nesse array. A memória restante usada por um bloco também é constante. Isso significa que o espaço perdido em uma SEList é apenas $O(b+n/b)$. Ao escolher um valor de b dentro de um fator constante $\sqrt{n}$, podemos fazer o overhead de espaço de uma SEList se aproximar do limite inferior $\sqrt{n}$ dado na Seção 2.6.2.

3.3.2 Encontrando Elementos

O primeiro desafio que encontramos em uma `SEList` é encontrar o item da lista com um dado índice `i`. Note que a localização do elemento é constituída por duas partes:

1. O nó `u` que contém o bloco que contém o elemento com o índice `i`; e
2. o índice `j` do elemento dentro do bloco.

SEList
```
class Location {
public:
  Node *u;
  int j;
  Location() { }
  Location(Node *u, int j) {
    this->u = u;
    this->j = j;
  }
};
```

Para encontrar o bloco que contém um determinado elemento, procedemos da mesma maneira que em uma `DLList`. Ou começamos pela frente da lista se deslocando para frente, ou por trás da lista se deslocando para trás até chegar ao nó que queremos. A única diferença é que, cada vez que passamos de um nó para o próximo, nós pulamos um bloco inteiro de elementos.

SEList
```
void getLocation(int i, Location &ell) {
  if (i < n / 2) {
    Node *u = dummy.next;
    while (i >= u->d.size()) {
      i -= u->d.size();
      u = u->next;
    }
    ell.u = u;
    ell.j = i;
  } else {
```

```
    Node *u = &dummy;
    int idx = n;
    while (i < idx) {
      u = u->prev;
      idx -= u->d.size();
    }
    ell.u = u;
    ell.j = i - idx;
  }
}
```

Lembre-se que, com exceção de no máximo um bloco, cada bloco contém pelo menos $b-1$ elementos, de modo que cada etapa em nossa pesquisa nos deixa $b-1$ elementos mais próximos do elemento procurado. Se nós estamos procurando para a frente, isto significa que atingimos o nó procurado após $O(1+i/b)$ passos. Se buscarmos para trás, então alcançamos o nó procurado após $O(1+(n-i)/b)$ passos. O algoritmo leva a menor dessas duas quantidades dependendo do valor de `i`, então o tempo para localizar o item com o índice `i` é $O(1+\min\{i,n-i\}/b)$.

Uma vez que saibamos como localizar o item com o índice `i`, as operações `get(i)` e `set(i,x)` traduzem-se em obter ou definir um determinado índice no bloco correto:

SEList

```
T get(int i) {
  Location l;
  getLocation(i, l);
  return l.u->d.get(l.j);
}
T set(int i, T x) {
  Location l;
  getLocation(i, l);
  T y = l.u->d.get(l.j);
  l.u->d.set(l.j, x);
  return y;
}
```

Os tempos de execução destas operações são dominados pelo tempo que leva para localizar o item, então eles também são executados no tempo $O(1+\min\{i,n-i\}/b)$.

3.3.3 Adicionando um Elemento

Adicionar elementos em uma SEList é um pouco mais complicado. Antes de considerar o caso geral, consideramos a operação mais fácil, add(x), na qual x é adicionado ao final da lista. Se o último bloco estiver cheio (ou não existir porque ainda não tem blocos), então nós primeiro alocamos um novo bloco e o anexamos à lista de blocos. Agora que temos certeza de que o último bloco existe e não está cheio, anexamos x no último bloco.

SEList

```
void add(T x) {
  Node *last = dummy.prev;
  if (last == &dummy || last->d.size() == b+1) {
    last = addBefore(&dummy);
  }
  last->d.add(x);
  n++;
}
```

As coisas ficam mais complicadas quando adicionamos ao interior da lista usando add(i,x). Primeiro localizamos i para obter o nó u cujo bloco contém o i-ésimo item da lista. O problema é que queremos inserir x no bloco do u, mas temos de estar preparados para o caso onde o bloco do u já contém b + 1 elementos, já estando cheio e sem espaço para x.

Suponha que $u_0, u_1, u_2, \ldots$ indica u, u.next, u.next.next, e assim por diante. Exploramos $u_0, u_1, u_2, \ldots$ procurando um nó que pode fornecer espaço para x. Três casos podem ocorrer durante a busca por espaço (veja a Figura 3.4):

1. Nós, rapidamente, (em $r + 1 \leq b$ passos) achamos um nó u_r cujo bloco não está cheio. Neste caso, executamos r deslocamentos de um elemento de um bloco para o próximo, de modo que um espaço livre em u_r se torne um espaço livre em u_0. Podemos, então, inserir x no bloco u_0.

2. Nós, rapidamente, (em $r + 1 \leq b$ passos) chegamos ao fim da lista de blocos. Neste caso, adicionamos um novo bloco vazio ao final da lista de blocos e procedemos como no primeiro caso.

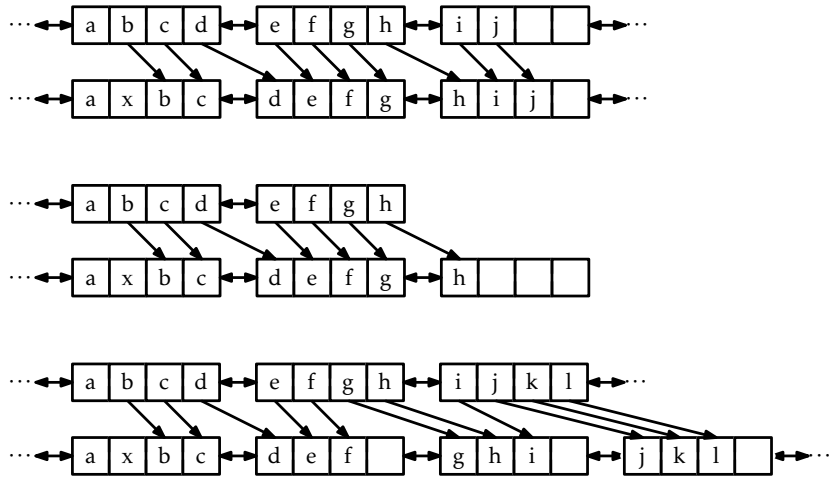


Figura 3.4: Os três casos que ocorrem durante a adição de um item x no interior de uma SEList. (Essa SEList tem bloco de tamanho b = 3.)

3. Após b passos não encontramos nenhum bloco que não está cheio. Neste caso, $u_0, \ldots, u_{b-1}$ é uma sequência de blocos b, em que cada um contém b + 1 elementos. Inserimos um novo bloco u_b ao final desta sequência e *distribuímos* os b(b + 1) elementos originais para que cada bloco de $u_0, \ldots, u_b$ contenha exatamente b elementos. Agora, o bloco de u_0 contém apenas b elementos, portanto ele tem espaço para inserirmos x.

SEList

```
void add(int i, T x) {
  if (i == n) {
    add(x);
    return;
  }
  Location l; getLocation(i, l);
  Node *u = l.u;
  int r = 0;
  while (r < b && u != &dummy && u->d.size() == b+1) {
    u = u->next;
```

```
    r++;
  }
  if (r == b) {// b blocks each with b+1 elements
    spread(l.u);
    u = l.u;
  }
  if (u == &dummy) { // ran off the end - add new node
    u = addBefore(u);
  }
  while (u != l.u) { // work backwards, shifting elements
    u->d.add(0, u->prev->d.remove(u->prev->d.size()-1));
    u = u->prev;
  }
  u->d.add(l.j, x);
  n++;
}
```

O tempo de execução da operação `add(i,x)` depende que cada um dos três casos acima ocorra. Os casos 1 e 2 envolvem examinar e deslocar elementos através de, no máximo, `b` blocos e demoram $O(b)$. O caso 3 envolve chamar o método `spread(u)`, que move $b(b+1)$ elementos e demora $O(b^2)$. Se nós ignorarmos o custo do Caso 3 (o que explicaremos mais adiante com a amortização), isto significa que o tempo de execução total para localizar `i` e executar a inserção de `x` é $O(b + \min\{i, n-i\}/b)$.

3.3.4 Remover um Elemento

Remover um elemento de uma `SEList` é similar a adicionar um elemento. Primeiro, localizamos o nó `u` que contém o elemento de índice `i`. Agora, temos que estar preparados para o caso em que não podemos remover um elemento de `u` sem fazer com que o bloco `u` fique menor que $b-1$.

Novamente, deixe $u_0, u_1, u_2, \ldots$ indicar `u`, `u.next`, `u.next.next`, e assim por diante. Examinamos $u_0, u_1, u_2, \ldots$ para procurar um nó do qual podemos pegar emprestado um elemento para fazer o tamanho do bloco u_0 ser, no mínimo, $b-1$. Há três casos a considerar (ver Figura 3.5):

1. Nós, rapidamente, (em $r+1 \leq b$ passos) achamos um nó cujo bloco contém mais de $b-1$ elementos. Neste caso, executamos r deslocamentos de um elemento de um bloco para o anterior, de modo que o

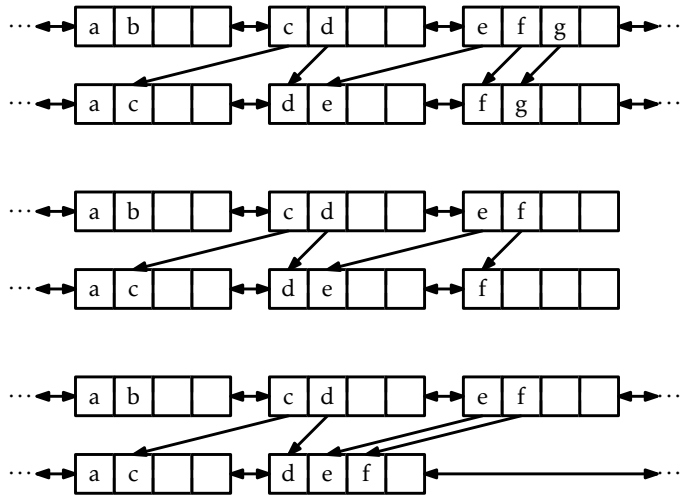


Figura 3.5: Os três casos que ocorrem durante a remoção de um item x no interior de uma SEList. (Esta SEList tem bloco de tamanho b = 3.)

elemento extra em u_r se torne um elemento extra em u_0. Podemos, então,remover o elemento apropriado do bloco u_0.

2. Nós, rapidamente, (em $r+1 \leq b$ passos) chegamos ao fim da lista de blocos. Neste caso, u_r é o último bloco, e não há razão para o bloco u_r conter, no mínimo, $b-1$ elementos. Portanto, procedemos como acima, pegando emprestado um elemento de u_r para fazer um elemento extra em u_0. Se isto fizer com que o bloco u_r fique vazio, então removemos ele.

3. Após b passos, não encontramos nenhum bloco contendo mais de $b-1$ elementos. Neste caso, $u_0, \ldots, u_{b-1}$ é uma sequência de b blocos, em que cada um contém $b-1$ elementos. Nós *concentramos* estes $b(b-1)$ elementos em $u_0, \ldots, u_{b-2}$ de modo que cada um destes $b-1$ blocos contenha exatamente b elementos e removemos u_{b-1}, que agora está vazio. Agora, o bloco de u_0 contém b elementos e, então, podemos remover o elemento apropriado dele.

SEList
```
T remove(int i) {
  Location l; getLocation(i, l);
  T y = l.u->d.get(l.j);
  Node *u = l.u;
  int r = 0;
  while (r < b && u != &dummy && u->d.size() == b - 1) {
    u = u->next;
    r++;
  }
  if (r == b) { // found b blocks each with b-1 elements
    gather(l.u);
  }
  u = l.u;
  u->d.remove(l.j);
  while (u->d.size() < b - 1 && u->next != &dummy) {
    u->d.add(u->next->d.remove(0));
    u = u->next;
  }
  if (u->d.size() == 0)
    remove(u);
  n--;
  return y;
}
```

Como a operação add(i,x), o tempo de execução da operação remove(i) é $O(b + \min\{i, n-i\}/b)$ se ignorarmos o custo do método gather(u) que ocorre no Caso 3.

3.3.5 Análise Amortizada de Distribuição e Concentração

Em seguida, consideramos o custo dos métodos gather(u) e spread(u) que podem ser executados pelos métodos add(i,x) e remove(i). Por uma questão de completude, aqui estão:

SEList
```
void spread(Node *u) {
  Node *w = u;
  for (int j = 0; j < b; j++) {
    w = w->next;
```

```
  }
  w = addBefore(w);
  while (w != u) {
    while (w->d.size() < b)
      w->d.add(0, w->prev->d.remove(w->prev->d.size()-1));
    w = w->prev;
  }
}
```

SEList

```
void gather(Node *u) {
  Node *w = u;
  for (int j = 0; j < b-1; j++) {
    while (w->d.size() < b)
      w->d.add(w->next->d.remove(0));
    w = w->next;
  }
  remove(w);
}
```

O tempo de execução de cada um destes métodos é dominado pelos dois loops aninhados. Ambos os loops, internos e externos, executam no máximo $b+1$ vezes, de modo que o tempo de execução total de cada um destes métodos é $O((b+1)^2) = O(b^2)$. No entanto, o seguinte lema mostra que estes métodos executam no máximo um de cada b chamadas para `add(i,x)` ou `remove(i)`.

Lema 3.1. *Se for criado uma* `SEList` *vazia e qualquer sequência de* $m \geq 1$ *chamadas para* `add(i,x)` *e* `remove(i)` *for executado, então o tempo total gasto durante as chamadas para* `spread()` *e* `gather()` *é* $O(bm)$.

Demonstração. Nós usaremos o método potencial de análise amortizada. Dizemos que um nó `u` é *frágil* se o bloco `u` não contém `b` elementos (de modo que `u` seja o último nó, ou contenha $b-1$ ou $b+1$ elementos). Qualquer nó cujo bloco contenha `b` elementos é *rígido*. Defina o *potential* de uma `SEList` como o número de nós frágeis que contém. Consideramos apenas a operação `add(i,x)` e sua relação com o número de chamadas para `spread(u)`. A análise de `remove(i)` e `gather(u)` é idêntica.

Observe que, se o Caso 1 ocorrer durante o método add(i,x), então somente um nó, u_r, tem o tamanho de seu bloco alterado. Portanto, no máximo um nó, ou seja u_r, irá de rígido a frágil. Se ocorrer o Caso 2, então um novo nó é criado, e este novo nó será frágil, mas nenhum outro nó mudará de tamanho, então o número de nós frágeis aumentará em um. Assim, tanto no Caso 1 ou no Caso 2 o potencial do SEList aumentará para no máximo um.

Finalmente, se ocorre o Caso 3, é porque $u_0,\ldots,u_{b-1}$ são todos nós frágeis. Então spread(u_0) é chamado e estes b nós frágeis são substituídos por $b+1$ nós rígidos. Finalmente, x é adicionado ao bloco u_0, fazendo u_0 frágil. No total, o potencial diminui em $b-1$.

Em resumo, o potencial começa em 0 (não há nós na lista). Cada vez que Caso 1 ou Caso 2 ocorre, o potencial aumenta, em no máximo, 1. Cada vez que ocorre o Caso 3, o potencial diminui para $b-1$. O potencial (que conta o número de nós frágeis) nunca é menor que 0. Nós concluímos que, para cada ocorrência do Caso 3, há pelo menos $b-1$ ocorrências do Caso 1 ou Caso 2. Assim, para cada chamada para spread(u) existem pelo menos b chamadas para add(i,x). Isso completa a verificação. □

3.3.6 Resumo

O seguinte teorema resume o desempenho das estruturas de dados de SEList:

Teorema 3.3. *Um* SEList *implementa a interface de* Lista. *Ignorando o custo das chamadas para* spread(u) *e* gather(u), *uma* SEList *com tamanho de bloco* b *supporta as operações*

- get(i) *e* set(i,x) *em* $O(1+\min\{i,n-i\}/b)$ *vezes por operação; e*
- add(i,x) *e* remove(i) *em* $O(b+\min\{i,n-i\}/b)$ *vezes por operação.*

Além disso, começando com uma SEList *vazia, qualquer sequência de operações* m add(i,x) *e* remove(i) *resulta em um tempo total gasto de* $O(bm)$ *durante todas as chamadas para* spread(u) *e* gather(u).

O espaço (medido em palavras)[1] *usado por uma* SEList *que armazena* n *elementos é* $n+O(b+n/b)$.

[1] Releia a Seção 1.4 para uma discussão de como a memória é medida.

A SEList é um compromisso entre uma ArrayList e uma DLList na qual a mistura relativa destas duas estruturas depende do tamanho do bloco b. No extremo b = 2, cada SEList armazena no máximo três valores, o que não é muito diferente da DLList. No outro extremo, b > n, todos os elementos são armazenados em um único array, assim como em um ArrayList. Entre esses dois extremos reside uma troca entre o tempo que leva para adicionar ou remover um item da lista e o tempo que leva para localizar um item de lista particular.

3.4 Discussão e Exercícios

Tanto as listas simplesmente encadeadas e as listas duplamente encadeadas são técnicas estabelecidas, tendo sido utilizadas em programas há mais de 40 anos. Elas são discutidas, por exemplo, por Knuth [46, Seções 2.2.3–2.2.5]. Mesmo a estrutura de dados SEList parece ser um exercício bem conhecido de estruturas de dados. A SEList é as vezes chamada de *unrolled linked list* [67].

Outra maneira de economizar espaço em uma lista duplamente encadeada é usar as chamadas listas XOR. Em uma lista XOR, cada nó, u, contém apenas um ponteiro, chamado u.nextprev, que contém o *ou exclusivo bi-a-bit* de u.prev e u.next. A própria lista precisa armazenar dois ponteiros, um para o nó dummy e um para dummy.next (o primeiro nó, ou dummy se a lista estiver vazia). Esta técnica utiliza o fato de que, se temos um ponteiro para u e u.prev, então podemos extrair u.next usando a fórmula

$$\texttt{u.next} = \texttt{u.prev}\hat{}\texttt{u.nextprev} \ .$$

(Aqui ^ calcula a operação bit-a-bit ou-exclusivo de seus dois argumentos.) Esta técnica complica um pouco o código e não é possível em algumas linguagens como Java e Python, que têm coleta de lixo, porém fornece uma lista duplamente encadeada que necessita apenas de um ponteiro por nó, Veja o artigo de Sinha [68] para uma discussão detalhada para listas XOR.

Exercício 3.1. Por que não é possível usar um nó dummy num SLList para evitar todos os casos especiais que ocorrem nas operações push(x),

pop(), add(x) e remove()?

Exercício 3.2. Projete e implemente um método para a SLList, secondLast(), que retorna o penúltimo elemento de SLList. Faça isso sem usar a variável de membro, n, que acompanha o tamanho da lista.

Exercício 3.3. Implementar as operações de List, get(i), set(i,x), add(i,x) e remove(i) em SLList. Cada uma dessas operações deve ser executada em um tempo $O(1+i)$.

Exercício 3.4. Projete e implemente um método para a SLList, reverse() que inverte a ordem dos elementos em SLList. Este método deve ser executado em tempo $O(n)$, não deve usar recursão, não deve usar nenhuma estrutura de dados secundária e não deve criar novos nós.

Exercício 3.5. Projete e implemente os métodos para a SLList e DLList chamados checkSize(). Esses métodos percorrem a lista e contam o número de nós para ver se isso corresponde ao valor, n, armazenado na lista. Esses métodos não retornam nada, mas lançam uma exceção se o tamanho que eles compõem não corresponde ao valor de n.

Exercício 3.6. Tente recriar o código para a operação addBefore(w) que cria um nó, u, e adiciona-o em DLList antes do nó w. Não consulte este capítulo. Mesmo que seu código não coincida exatamente com o código fornecido neste livro, ele ainda pode estar correto. Teste e veja se ele funciona.

Os próximos exercícios envolvem a realização de manipulações em DLList. Você deve completá-los sem alocar novos nós ou matrizes temporárias. Todos podem ser feitos apenas alterando os valores prev e next dos nós existentes.

Exercício 3.7. Escreva um método isPalindrome() para uma DLList que retorna true se a lista for *palíndromo*, ie, o elemento na posição i é igual ao elemento na posição $n-i-1$ para todos $i \in \{0,\ldots,n-1\}$. Seu código deve ser executado em tempo $O(n)$.

Exercício 3.8. Implementar um método rotate(r) que "rotaciona" uma DLList para que o item da lista i se torne o item da lista $(i+r) \bmod n$. Esse método deve ser executado em tempo $O(1+\min\{r, n-r\})$ e não deve modificar nenhum nó na lista.

Exercício 3.9. Escreva um método, `truncate(i)`, que trunca uma `DLList` na posição `i`. Depois de executar este método, o tamanho da lista será `i` e deve conter apenas os elementos nos índices $0,\ldots,i-1$.. O valor de retorno é outra `DLList` que contém os elementos nos índices $i,\ldots,n-1$. Esse método deve ser executado em tempo $O(\min\{i, n-i\})$.

Exercício 3.10. Escreva um método `absorb(l2)` para uma `DLList`, que leva como argumento uma `DLList`, `l2`, esvazia-o e acrescenta seu conteúdo, em ordem, ao receptor. Por exemplo, se `l1` contiver a,b,c e `l2` contém d,e,f, e depois de chamar `l1.absorb(l2)`, `l1` conterá a,b,c,d,e,f e `l2` estará vazia.

Exercício 3.11. Escreva um método `deal()` que remove todos os elementos com índices de números ímpares da `DLList` e retorna uma `DLList` contendo esses elementos. Por exemplo, se `l1` contém os elementos a,b,c,d,e,f, depois de chamar `l1.deal()`, `l1` deve conter a,c,e e uma lista contendo b,d,f deve ser devolvida.

Exercício 3.12. Escreva um método, `reverse()`, que inverta a ordem dos elementos em uma `DLList`.

Exercício 3.13. Este exercício orienta você através de uma implementação do algoritmo merge-sort para classificar uma `DLList`, como discutido na Seção 11.1.1.

1. Escreva o método `DLList` chamado `takeFirst(l2)`. Este método leva o primeiro nó de `l2` e adiciona-o à lista de recepção. Isso equivale a `add(size(),l2.remove(0))`, exceto que ele não deve criar um novo nó.

2. Escreva um método estático de `DLList`, `merge(l1,l2)`, que recebe duas listas classificadas `l1` e `l2`, mescla-as e retorna uma nova lista contendo o resultado. Isso tem como consequência que `l1` e `l2` se tornam vazias no processo. Por exemplo, se `l1` contém a,c,d e `l2` contém b,e,f, assim este método retorna uma nova lista contendo a,b,c,d,e,f.

3. Escreva um método `DLList sort()` que classifica os elementos contidos na lista usando o algoritmo de ordenação por mesclagem. Este algoritmo recursivo funciona da seguinte maneira :

(a) Se a lista contém 0 ou 1 elementos, então não há nada a fazer. Caso contrário,

(b) Usando o método `truncate(size()/2)`, divide a lista em duas listas de comprimento aproximadamente igual, l1 e l2;

(c) Recursivamente classificar l1;

(d) Recursivamente classificar l2; e, finalmente,

(e) Mesclar l1 e l2 em uma única lista ordenada.

Os próximos exercícios são mais avançados e requerem uma compreensão do que acontece com o valor mínimo armazenado em uma `Stack` ou `Queue` à medida que os itens são adicionados e removidos.

Exercício 3.14. Projetar e implementar uma estrutura de dados `MinStack` que pode armazenar elementos comparáveis e suporta as operações de pilha `push(x)`, `pop()`, e `size()`, assim como a operação de `min()`, que retorna o valor mínimo atualmente armazenado na estrutura de dados. Todas as operações devem ser executadas em tempo constante.

Exercício 3.15. Projetar e implementar uma estrutura de dados `MinQueue` que pode armazenar elementos comparáveis e suporta as operações de fila `add(x)`, `remove()`, e `size()`, assim como a operação de `min()`, que retorna o valor mínimo atualmente armazenado na estrutura de dados. Todas as operações devem ser executadas em tempo constante.

Exercício 3.16. Projetar e implementar a `MinDeque` uma estrutura de dados que pode armazenar elementos comparáveis e suporta todas as operações de deque `addFirst(x)`, `addLast(x)` `removeFirst()`, `removeLast()` e `size()`,e a operação `min()`, que retorna o menor valor atualmente armazenado em estrutura de dados. Todas as operações devem ser executadas em tempo constante.

Os próximos exercícios são projetados para testar o entendimento do leitor da implementação e análise da lista eficiente em espaço `SEList`:

Exercício 3.17. Prove que, se uma `SEList` é usada como uma `Stack` (para que as únicas modificações no `SEList` sejam feitas usando `push(x)` ≡ `add(size(), x)` e `pop()` ≡ `remove(size()-1)`), então esta operação executado em tempo amortizado constante, independente do valor de `b`.

Exercício 3.18. Projetar e implementar uma versão de um `SEList` que suporte todas as operações `Deque` em tempo amortizado constante por operação, independente do valor de `b`.

Exercício 3.19. Explicar como usar o operador exclusivo de bit ou operador, `^`, para trocar os valores de duas variáveis `int` sem usar uma terceira variável.

Capítulo 4

Skiplists

Neste capítulo discutiremos uma bela estrutura de dados: a skiplist, que possui diversas de aplicações. Usando uma skiplist, podemos implementar uma `Lista` que tenha tempo de $O(\log n)$ para implementações de `get(i)`, `set(i,x)`, `add(i,x)`, e `remove(i)`. Nós também podemos implementar uma `SSet` em que todas as operações são executadas com tempo esperado de $O(\log n)$.

A eficiência das skiplists se baseia no uso da randomização. Quando um novo elemento é adicionado à skiplist, ela utiliza lançamentos aleatórios de moeda para determinar a altura do novo elemento. O desempenho das skiplists é expresso em termos do tempo de execução esperado e do tamanho do caminho. Esta expectativa é baseada nos lançamentos aleatórios de moeda usados pela skiplist. Na implementação, os lançamentos aleatórios de moedas usados pela skiplist são simulados usando um gerador de números (ou bits) pseudo aleatórios.

4.1 A Estrutura Básica

Conceitualmente, uma skiplist é uma sequência de listas simplesmente encadeadas $L_0, \ldots, L_h$. Cada lista L_r contém um subconjunto de itens em L_{r-1}. Começamos com a lista inicial L_0 que contém `n` itens e construímos L_1 a partir de L_0, L_2 a partir de L_1, e assim por diante.

Os itens em L_r são obtidos lançando a moeda para cada elemento, `x`, em L_{r-1} e incluindo `x` em L_r se a moeda der cara. Este processo termina

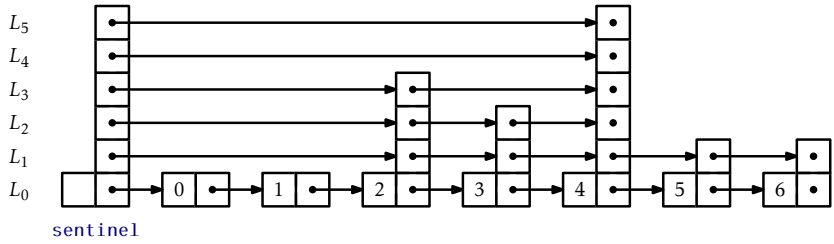


Figura 4.1: Uma skiplist com sete elementos.

quando criamos uma lista L_r que está vazia. Um exemplo de uma skiplist é mostrado na Figura 4.1.

Para um elemento, `x`, na skiplist, nós chamamos de *altura* de `x` o valor mais alto de r de modo que `x` apareça em L_r. Assim, por exemplo, elementos que só aparecem em L_0 tem altura 0. Se pararmos um momento pra pensar sobre isso, percebemos que a altura de `x` corresponde à seguinte experiência: lance uma moeda repetidamente até aparecer como coroa. Quantas vezes apareceu cara? A resposta, sem surpresa, é que a altura esperada de um nó é 1. (Esperamos lançar a moeda duas vezes antes de aparecer coroa, mas não contamos a última jogada.) A *altura* de uma skiplist é a altura do seu nó mais alto.

No começo de cada lista está um nó especial, chamado de *sentinela*, que atua como um pseudo nó (*dummy*) para a lista. A propriedade chave das skiplists é que existe um caminho curto para busca, chamado de *caminho de busca*, do sentinela em L_h para cada nó em L_0. Veja como é fácil construir um caminho de busca para o nó `u` (veja Figura 4.2): comece no canto superior esquerdo da skiplist (o sentinela em L_h) e sempre vá para a direita, a menos que ultrapasse `u`, nesse caso você deve dar um passo para a lista de baixo.

Mais precisamente, para construir um caminho de busca para o nó `u` em L_0, começaremos pelo sentinela, `w`, em L_h. Em seguida, verificamos `w.next`. Se `w.next` contém um elemento que aparece antes de `u` em L_0, então nós ajustamos `w = w.next`. Caso contrário, movemos para baixo e continuamos a busca da ocorrência de `w` na lista L_{h-1}. Continuamos assim até chegar ao antecessor de `u` em L_0.

O resultado a seguir, que vamos provar em Seção 4.4, mostra que o

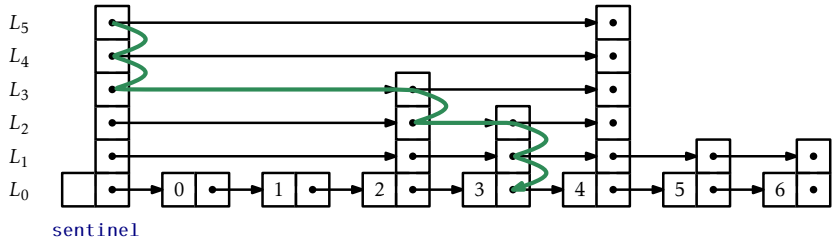


Figura 4.2: Caminho de busca para o nó que contém o valor 4 em uma skiplist.

caminho de busca é bastante curto:

Lema 4.1. *O comprimento esperado do caminho de busca para qualquer nó,* u, *em* L_0 *é de no máximo* $2\log n + O(1) = O(\log n)$.

Uma maneira eficiente em termos de espaço para implementar uma skiplist é definir um Node, u, consistindo em um valor de dados, x e um array, next, de ponteiros, onde u.next[i] aponta para o sucessor de u na lista L_i. Desta forma, os dados, x, em um nó são armazenados apenas uma vez, embora x possa aparecer em várias listas.

SkiplistSSet

```
struct Node {
  T x;
  int height;      // length of next
  Node *next[];
};
```

As próximas duas seções deste capítulo abordam duas aplicações diferentes de skiplists. Em cada uma dessas aplicações, L_0 armazena a estrutura principal (uma lista de elementos ou um conjunto de elementos ordenados). A principal diferença entre essas estruturas é a forma como um caminho de busca é navegado; em particular, elas se diferem em como é decidido se um caminho de busca deve descer em L_{r-1} ou ir para a direita dentro de L_r.

4.2 SkiplistSSet: Uma SSet eficiente

Uma SkiplistSSet usa uma skiplist para implementar a interface SSet. Quando usada desta maneira, a lista L_0 armazena os elementos de SSet de forma ordenada. O método find(x) funciona seguindo o caminho de busca para o menor valor y tal que y ≥ x:

SkiplistSSet

```
Node* findPredNode(T x) {
  Node *u = sentinel;
  int r = h;
  while (r >= 0) {
    while (u->next[r] != NULL
                && compare(u->next[r]->x, x) < 0)
      u = u->next[r]; // go right in list r
    r--; // go down into list r-1
  }
  return u;
}
T find(T x) {
  Node *u = findPredNode(x);
  return u->next[0] == NULL ? null : u->next[0]->x;
}
```

Seguir o caminho de busca para y é fácil. Quando situado em algum nó, u, em L_r, olhamos diretamente para u.next[r].x, se x > u.next[r].x, então damos um passo à direita em L_r. Caso contrário, descemos para L_{r-1}. Cada passo (para direita ou descendo) nesta pesquisa leva um tempo constante; assim, para Lema 4.1, o tempo de execução esperado de find(x) é de $O(\log n)$.

Antes de poder adicionar um elemento a SkipListSSet, precisamos de um método para simular o lançamento de moedas que determina a altura, k, de um novo nó. Fazemos isso escolhendo um número inteiro aleatório, z, e contando o número de 1s na representação binária de z: [1]

[1] Este método não reproduz exatamente a experiência de lançar moedas, uma vez que o valor de k será sempre inferior ao número de bits em um int. Contudo, isso terá um impacto insignificante, a menos que o número de elementos na estrutura seja muito maior do que $2^{32} = 4294967296$.

SkiplistSSet

```
int pickHeight() {
  int z = rand();
  int k = 0;
  int m = 1;
  while ((z & m) != 0) {
    k++;
    m <<= 1;
  }
  return k;
}
```

Para implementar o método add(x) na SkiplistSSet procuramos x e então colocamos x em algumas listas $L_0, \ldots, L_k$, onde k é selecionado usando o método pickHeight(). A maneira mais fácil de fazer isso é usar um array, stack, que acompanha os nós em que o caminho de busca desce de alguma lista L_r para L_{r-1}. Mais precisamente, stack[r] é o nó em L_r onde o caminho de busca prosseguiu para L_{r-1}. Os nós que modificamos para inserir x são precisamente os nós stack[0],...,stack[k]. O código seguinte implementa este algoritmo para add(x):

SkiplistSSet

```
bool add(T x) {
  Node *u = sentinel;
  int r = h;
  int comp = 0;
  while (r >= 0) {
    while (u->next[r] != NULL
               && (comp = compare(u->next[r]->x, x)) < 0)
      u = u->next[r];
    if (u->next[r] != NULL && comp == 0)
      return false;
    stack[r--] = u;         // going down, store u
  }
  Node *w = newNode(x, pickHeight());
  while (h < w->height)
    stack[++h] = sentinel; // height increased
  for (int i = 0; i <= w->height; i++) {
    w->next[i] = stack[i]->next[i];
    stack[i]->next[i] = w;
  }
```

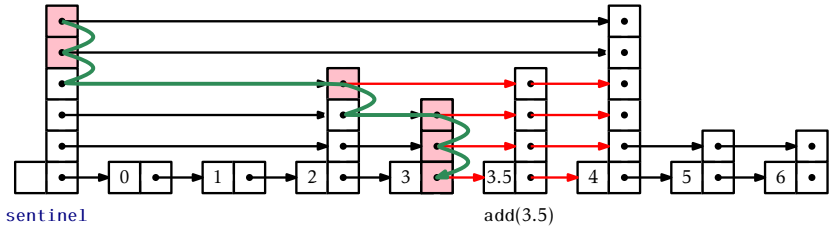


Figura 4.3: Adicionando o nó contendo o valor 3.5 a uma skiplist. Os nós armazenados em `stack` estão marcados.

```
  n++;
  return true;
}
```

A remoção de um elemento, `x`, é feita de forma semelhante, exceto que não há necessidade do `stack` para manter o controle do caminho de busca. A remoção pode ser feita enquanto seguimos o caminho de busca. Procuramos por `x` e cada vez que a pesquisa se move para baixo a partir do nó `u`, verificamos se `u.next.x = x` e, em caso afirmativo, desligamos `u` da lista:

SkiplistSSet

```
bool remove(T x) {
  bool removed = false;
  Node *u = sentinel, *del;
  int r = h;
  int comp = 0;
  while (r >= 0) {
    while (u->next[r] != NULL
               && (comp = compare(u->next[r]->x, x)) < 0) {
      u = u->next[r];
    }
    if (u->next[r] != NULL && comp == 0) {
      removed = true;
      del = u->next[r];
      u->next[r] = u->next[r]->next[r];
      if (u == sentinel && u->next[r] == NULL)
        h--; // skiplist height has gone down
    }
    r--;
```

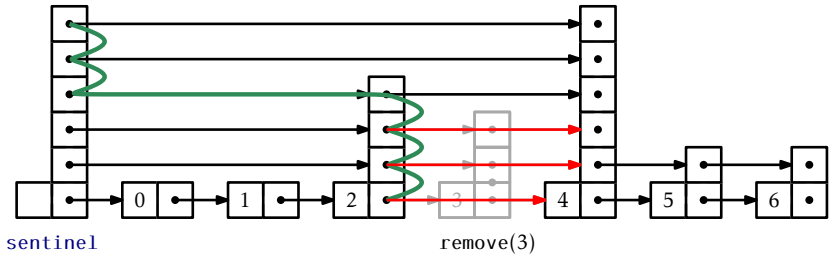


Figura 4.4: Removendo o nó que contém o valor 3 da skiplist.

```
  }
  if (removed) {
    delete del;
    n--;
  }
  return removed;
}
```

4.2.1 Resumo

O seguinte teorema resume o desempenho de uma skiplists quando usada para implementar conjuntos ordenados:

Teorema 4.1. *`SkiplistSSet` implementa a interface `SSet`. Uma `SkiplistSSet` suporta as operações* `add(x)`, `remove(x)`, *e* `find(x)` *em um tempo esperado* $O(\log n)$ *por operação.*

4.3 `SkiplistList`: Uma `List` de acesso aleatório eficiente

Uma `SkiplistList` implementa a interface `List` usando uma estrutura skiplist. Em uma `SkiplistList`, L_0 contém os elementos da lista na ordem em que aparecem na lista. Como em uma `SkiplistSSet`, os elementos podem ser adicionados, removidos, e acessados em um tempo $O(\log n)$.

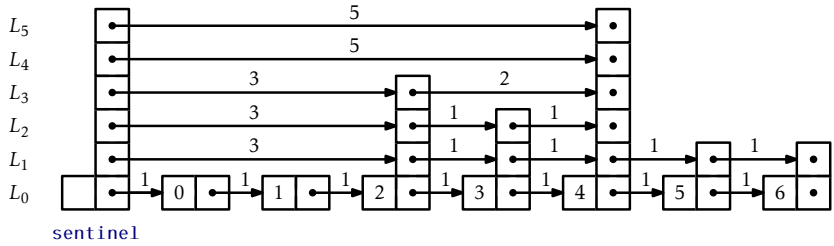


Figura 4.5: Os tamanhos das arestas em uma skiplist.

Para que isso seja possível, precisamos de uma maneira para seguir o caminho de busca para o `i`ésimo elemento em L_0. A maneira mais fácil de fazer isso é definir o conceito de *comprimento* de uma aresta em uma lista, L_r. Definimos o tamanho de cada aresta em L_0 como 1. O tamanho de uma aresta, `e`, em L_r, $r > 0$, é definido como a soma dos tamanhos das arestas abaixo de `e` em L_{r-1}. De forma equivalente, o tamanho de `e` é o número de arestas de L_0 abaixo de `e`. Veja Figura 4.5 para um exemplo de uma skiplist mostrando o tamanho de suas arestas. Uma vez que as arestas das skiplists são armazenados em arrays, os tamanhos podem ser armazenados da mesma maneira.

SkiplistList
```
struct Node {
  T x;
  int height;     // length of next
  int *length;
  Node **next;
};
```

A propriedade útil desta definição de tamanho é que, se estamos em um nó na posição `j` em L_0 e seguimos uma aresta de tamanho ℓ, então movemos para um nó cuja posição, em L_0, é $j + \ell$. Desta forma, enquanto seguimos um caminho de busca, podemos manter o controle da posição, `j`, do nó atual em L_0. Em um nó, `u`, em L_r, vamos para a direita se `j` mais o tamanho de `u.next[r]` é menor que `i`. Caso contrário, desceremos para L_{r-1}.

SkiplistList

```
Node* findPred(int i) {
  Node *u = sentinel;
  int r = h;
  int j = -1;   // the index of the current node in list 0
  while (r >= 0) {
    while (u->next[r] != NULL && j + u->length[r] < i) {
      j += u->length[r];
      u = u->next[r];
    }
    r--;
  }
  return u;
}
```

SkiplistList

```
T get(int i) {
  return findPred(i)->next[0]->x;
}
T set(int i, T x) {
  Node *u = findPred(i)->next[0];
  T y = u->x;
  u->x = x;
  return y;
}
```

Como a parte mais difícil das operações `get(i)` e `set(i,x)` é encontrar o `i`-ésimo nó em L_0, essas operações são executadas em um tempo $O(\log n)$.

Adicionar um elemento a uma `SkiplistList` em uma posição, `i`, é relativamente simples. Ao contrário do que acontece em uma `SkiplistSSet`, temos certeza que um novo nó será realmente adicionado, assim podemos realizar a adição ao mesmo tempo em que buscamos a localização do novo nó. Primeiramente, pegamos a altura, `k`, do nó recentemente inserido, `w`, e então avançamos o caminho de busca para `i`. Toda vez que o caminho de busca desce de L_r com $r \leq k$, juntamos `w` em L_r. O único cuidado extra necessário é assegurar que o comprimento das arestas seja atualizado devidamente. Veja Figura 4.6.

Note que, cada vez que o caminho de busca desce um nó, `u`, em L_r, o comprimento da aresta `u.next[r]` aumenta em um, uma vez que estamos

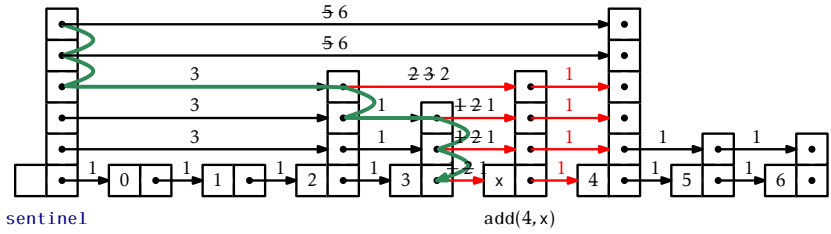


Figura 4.6: Adicionando um elemento em uma `SkiplistList`.

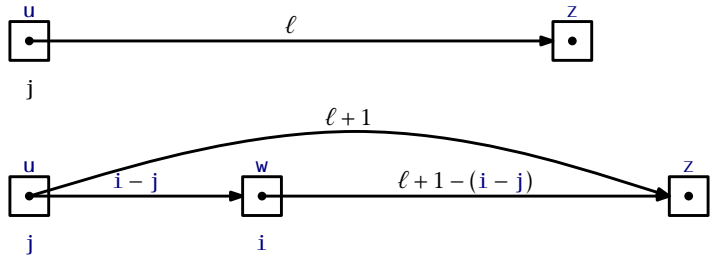


Figura 4.7: Atualizando os comprimentos das arestas enquanto junta-se o nó `w` em uma skiplist.

adicionando um elemento abaixo desta aresta na posição `i`. Unir o nó `w` entre dois nós, `u` e `z`, funciona como mostrado na Figura 4.7. Enquanto seguimos o caminho de busca, já estamos cientes da posiçao, `j`, de `u` em L_0. Portanto, sabemos que o comprimento da aresta de `u` a `w` é `i - j`. Também podemos deduzir o comprimento da aresta de `w` a `z` através do comprimento, ℓ, da aresta de `u` a `z`. Assim sendo, podemos juntar em `w` e atualizar os comprimentos das arestas em tempo constante.

Isto soa mais complicado do que é, o código, na verdade, é bem simples:

SkiplistList

```
void add(int i, T x) {
  Node *w = newNode(x, pickHeight());
  if (w->height > h)
    h = w->height;
  add(i, w);
```

```
}
```

SkiplistList

```
Node* add(int i, Node *w) {
  Node *u = sentinel;
  int k = w->height;
  int r = h;
  int j = -1; // index of u
  while (r >= 0) {
    while (u->next[r] != NULL && j + u->length[r] < i) {
      j += u->length[r];
      u = u->next[r];
    }
    u->length[r]++; // to account for new node in list 0
    if (r <= k) {
      w->next[r] = u->next[r];
      u->next[r] = w;
      w->length[r] = u->length[r] - (i - j);
      u->length[r] = i - j;
    }
    r--;
  }
  n++;
  return u;
}
```

Agora a implementação da operação `remove(i)` em uma `SkiplistList` deveria ser óbvia. Seguimos o caminho de busca para o nó na posição `i`. Cada vez que o caminho de busca desce de um nó, `u`, no nível `r` diminuímos o comprimento da aresta deixando `u` neste nível. Também checamos se `u.next[r]` é o elemento de posição `i` e, caso seja, o tiramos da lista neste nível. Um exemplo é mostrado na Figura 4.8.

SkiplistList

```
T remove(int i) {
  T x = null;
  Node *u = sentinel, *del;
  int r = h;
  int j = -1; // index of node u
  while (r >= 0) {
```

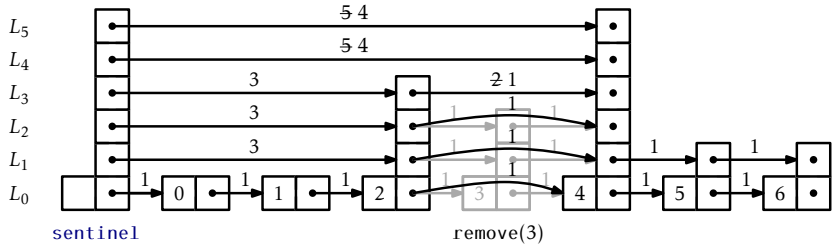


Figura 4.8: Removing an element from a SkiplistList.

```
    while (u->next[r] != NULL && j + u->length[r] < i) {
      j += u->length[r];
      u = u->next[r];
    }
    u->length[r]--; // for the node we are removing
    if (j + u->length[r] + 1 == i && u->next[r] != NULL) {
      x = u->next[r]->x;
      u->length[r] += u->next[r]->length[r];
      del = u->next[r];
      u->next[r] = u->next[r]->next[r];
      if (u == sentinel && u->next[r] == NULL)
        h--;
    }
    r--;
  }
  deleteNode(del);
  n--;
  return x;
}
```

4.3.1 Resumo

O teorema a seguir resume o desempenho da estrutura de dados SkiplistList:

Teorema 4.2. *Uma SkiplistList implementa a interface List. Uma SkiplistList suporta as operações* get(i), set(i,x), add(i,x), *e* remove(i) *no tempo de execução* $O(\log n)$ *esperado.*

4.4 Análise de Skiplists

Nesta seção, analisamos a altura, tamanho e comprimento esperados do caminho de busca em uma skiplist. Esta seção requer um conhecimento de probabilidade básica. Várias provas são baseadas nas seguintes observações básicas sobre lançamentos de moedas.

Lema 4.2. *Faça com que T seja o número de vezes que uma moeda é jogada e incluindo a primeira vez que a moeda dá cara. Logo* $\mathrm{E}[T] = 2$.

Demonstração. Suponhamos que paremos de jogar a moeda na primeira vez que a moeda der cara. Define-se a variável indicadora

$$I_i = \begin{cases} 0 & \text{se a moeda for lançada menos que } i \text{ vezes} \\ 1 & \text{se a moeda for lançada } i \text{ ou mais vezes} \end{cases}$$

Note que $I_i = 1$ se e apenas se, os primeiros $i-1$ lançamentos da moeda derem coroa, então $\mathrm{E}[I_i] = \Pr\{I_i = 1\} = 1/2^{i-1}$. Observe que T, o número total de lançamentos da moeda, pode ser escrito como $T = \sum_{i=1}^{\infty} I_i$. Portanto,

$$\begin{aligned} \mathrm{E}[T] &= \mathrm{E}\left[\sum_{i=1}^{\infty} I_i\right] \\ &= \sum_{i=1}^{\infty} \mathrm{E}[I_i] \\ &= \sum_{i=1}^{\infty} 1/2^{i-1} \\ &= 1 + 1/2 + 1/4 + 1/8 + \cdots \\ &= 2 \ . \end{aligned}$$

□

Os dois próximos lemas nos dizem que as skiplists têm tamanho linear:

Lema 4.3. *O número esperado de nós em uma skiplist contendo* n *elementos, não incluindo ocorrências do sentinela, é* $2n$.

Demonstração. A probabilidade de que qualquer elemento particular, x, esteja incluso na lista L_r é $1/2^r$, então o número de nós esperado na L_r é

$n/2^r$.[2] Portanto, o número total de nós esperado em todas as listas é

$$\sum_{r=0}^{\infty} n/2^r = n(1 + 1/2 + 1/4 + 1/8 + \cdots) = 2n \ . \qquad \square$$

Lema 4.4. *A altura esperada de uma skiplist contendo n elements é no máximo* $\log n + 2$.

Demonstração. Para cada $r \in \{1, 2, 3, \ldots, \infty\}$, defina a variável indicadora aleatória

$$I_r = \begin{cases} 0 & \text{se } L_r \text{ é vazia} \\ 1 & \text{se } L_r \text{ não é vazia} \end{cases}$$

A altura, h, da skiplist é dada por

$$h = \sum_{r=1}^{\infty} I_r \ .$$

Note que I_r nunca é maior que o tamanho, $|L_r|$, de L_r, assim

$$E[I_r] \leq E[|L_r|] = n/2^r \ .$$

Portanto, teremos

$$\begin{aligned} E[h] &= E\left[\sum_{r=1}^{\infty} I_r\right] \\ &= \sum_{r=1}^{\infty} E[I_r] \\ &= \sum_{r=1}^{\lfloor \log n \rfloor} E[I_r] + \sum_{r=\lfloor \log n \rfloor + 1}^{\infty} E[I_r] \\ &\leq \sum_{r=1}^{\lfloor \log n \rfloor} 1 + \sum_{r=\lfloor \log n \rfloor + 1}^{\infty} n/2^r \\ &\leq \log n + \sum_{r=0}^{\infty} 1/2^r \\ &= \log n + 2 \ . \end{aligned}$$

$\square$

[2] Veja Seção 1.3.4 para ver como isso é derivado usando variáveis indicadoras e linearidade de expectativa.

Lema 4.5. *O número esperado de nós em uma skiplist contendo* n *elementos, incluindo todas as ocorrências do sentinela, é* $2n + O(\log n)$.

Demonstração. Por Lema 4.3, o número esperado de nós, sem incluir o sentinela, é $2n$. O número de ocorrências do sentinela é igual à altura, h, da skiplist assim, por Lema 4.4 o número esperado de ocorrências do sentinela é no máximo $\log n + 2 = O(\log n)$. □

Lema 4.6. *O comprimento esperado do caminho de busca em uma skiplist é no máximo* $2\log n + O(1)$.

Demonstração. A maneira mais fácil de ver isso é considerar o *caminho de pesquisa reversa* para um nó, x. Este caminho começa no predecessor de x em L_0. A qualquer momento, se o caminho puder subir um nível, isso acontecerá. Se não conseguir subir um nível, ele irá para a esquerda. Pensar nisso por alguns instantes nos convencerá de que o caminho de pesquisa reversa para x é idêntico ao caminho de pesquisa para x, exceto que é invertido.

O número de nós que o caminho de pesquisa reverso visita em um nível específico, r, está relacionado à seguinte experiência: lançar uma moeda. Se a moeda surgir como cara, vá para cima e pare. Caso contrário, vá para a esquerda e repita a experiência. O número de lançamentos antes da cara representa o número de passos à esquerda que um caminho de busca reversa toma em um determinado nível. [3] Lema 4.2 nos diz que o número esperado de lançamentos de moeda antes das primeiras caras é 1.

Seja S_r o número de passos que o caminho de busca direta no nível r que vão para a direita. Acabamos de argumentar que $E[S_r] \leq 1$. Além disso, $S_r \leq |L_r|$, já que não podemos dar mais passos em L_r do que o tamanho de L_r, assim

$$E[S_r] \leq E[|L_r|] = n/2^r \ .$$

Podemos agora terminar como na prova de Lema 4.4. Seja S o comprimento do caminho de busca para algum nó, u, em uma skiplist, e seja h a

[3] Observe que isso pode sobrecarregar o número de etapas à esquerda, já que a experiência deve terminar em as primeiras caras ou quando o caminho de busca alcança o sentinela, o que ocorrer primeiro. Isto não é um problema já que o lema está apenas indicando um limite superior.

altura da skiplist. Então

$$\begin{aligned}
E[S] &= E\left[h + \sum_{r=0}^{\infty} S_r\right] \\
&= E[h] + \sum_{r=0}^{\infty} E[S_r] \\
&= E[h] + \sum_{r=0}^{\lfloor \log n \rfloor} E[S_r] + \sum_{r=\lfloor \log n \rfloor + 1}^{\infty} E[S_r] \\
&\leq E[h] + \sum_{r=0}^{\lfloor \log n \rfloor} 1 + \sum_{r=\lfloor \log n \rfloor + 1}^{\infty} n/2^r \\
&\leq E[h] + \sum_{r=0}^{\lfloor \log n \rfloor} 1 + \sum_{r=0}^{\infty} 1/2^r \\
&\leq E[h] + \sum_{r=0}^{\lfloor \log n \rfloor} 1 + \sum_{r=0}^{\infty} 1/2^r \\
&\leq E[h] + \log n + 3 \\
&\leq 2\log n + 5 \ . \qquad \square
\end{aligned}$$

O seguinte teorema resume os resultados desta seção:

Teorema 4.3. *Uma skiplist contendo n elementos tem tamanho esperado $O(n)$ e o comprimento esperado do caminho de busca de qualquer elemnento particular é no máximo $2\log n + O(1)$.*

4.5 Discussão e Exercícios

Skiplists foram introduzidas por Pugh [60] que também apresentou numerosas aplicações e extensões da skiplists [59]. Desde então elas têm sido estudadas extensivamente. Diversos pesquisadores têm feito precisas análises do comprimento esperado e da variância do caminho de busca para o i-ésimo elemento em uma skiplist [45, 44, 56]. Versões determinísticas [53], tendenciosas [8, 26], e de ajuste próprio [12] das skiplists têm sido desenvolvidas. Implementações da Skiplist têm sido escritas

em diversas linguagens e frameworks e têm sido usadas em sistemas de banco de dados open-source [69, 61]. Uma variante da skiplist é usada na estrutura de gerência de processos do núcleo do sistema operacional HP-UX [42].

Exercício 4.1. Explique os caminhos de pesquisa para 2.5 e 5.5 na skiplist da Figura 4.1.

Exercício 4.2. Explique a adição dos valores 0,5 (com uma altura de 1) e 3,5 (com uma altura de 2) na skiplist da Figura 4.1.

Exercício 4.3. Explique a remoção dos valores 1 e 3 na skiplist da Figura 4.1.

Exercício 4.4. Explique a execução de `remove(2)` na `SkiplistList` da Figura 4.5.

Exercício 4.5. Ilustre a execução de `add(3,x)` na `SkiplistList` da Figura 4.5, assumindo que o método `pickHeight()` seleciona uma altura de 4 para o nó recém-criado.

Exercício 4.6. Mostre que, durante uma operação `add(x)` ou `remove(x)`, o número esperado de ponteiros em `SkiplistSet` que são alterados é constante.

Exercício 4.7. Suponha que, em vez de promover um elemento de L_{i-1} para L_i com base num lançamento de moeda, promovamos com alguma probabilidade p, $0 < p < 1$.

1. Mostre que, com esta modificação, o comprimento esperado de um caminho de pesquisa é no máximo $(1/p)\log_{1/p} n + O(1)$.
2. Qual é o valor de p que minimiza a expressão anterior?
3. Qual é a altura esperada da skiplist?
4. Qual é o número esperado de nós na skiplist?

Exercício 4.8. O método `find(x)` em uma `SkiplistSet` às vezes executa *comparações redundantes*; estas ocorrem quando `x` é comparado com o mesmo valor mais de uma vez. Elas podem ocorrer quando, para algum nó, `u`, `u.next[r] = u.next[r-1]`. Mostre como essas comparações

redundantes acontecem e modifique `find(x)` para que elas sejam evitadas. Analise o número esperado de comparações feitas pelo seu método `find(x)` modificado.

Exercício 4.9. Projete e implemente uma versão de uma skiplist que implemente a interface SSet, mas também permite acesso rápido a elementos por classificação. Ou seja, ele também suporta a função `get(i)`, que retorna o elemento cuja classificação é `i` no tempo esperado $O(\log n)$. (A classificação de um elemento `x` em um SSet é o número de elementos no SSet que são menores que `x`.)

Exercício 4.10. Um *indicador* em uma skiplist é um array que armazena a sequência de nós em um caminho de busca no qual o caminho de busca evolui. (A variável `pilha` no código de `add(x)` na página 95 é um indicador, os nós sombreados na Figura 4.3 mostram o conteúdo do indicador). Pode-se pensar em um dedo apontando o caminho para um nó na lista mais baixa, L_0.

Uma *busca por indicador* implementa a operação `find(x)` usando um indicador que percorre a lista até alcançar um nó `u`, tal que `u.x` < `x` e `u.next` = `null` ou `u.next.x` > `x`, e em seguida realiza uma pesquisa normal para `x` a partir de `u`. É possível provar que o número esperado de passos necessários para uma pesquisa por indicador é $O(1 + \log r)$, onde r é o número de valores em L_0 entre `x` e o valor apontado pelo indicador.

Implementar uma subclasse de `Skiplist` chamada `SkiplistWithFinger` que implementa operações `find(x)` usando um indicador interno. Esta subclasse armazena um indicador, que é então usado para que cada operação `find(x)` seja implementada como uma pesquisa de indicador. Durante cada operação `find(x)`, o indicador é atualizado para que cada operação `find(x)` use, como ponto de partida, um indicador que aponte para o resultado da operação `find(x)` anterior.

Exercício 4.11. Escreva um método, `truncate(i)`, que trunca uma `SkiplistList` na posição `i`. Após a execução deste método, o tamanho da lista é `i` e contém apenas os elementos nos índices $0,\ldots,$`i`-1. O valor de retorno é outra `SkiplistList` que contém os elementos nos índices `i`$,\ldots,$`n`-1. Esse método deve ser executado em um tempo $O(\log n)$.

Exercício 4.12. Escreva um método `SkiplistList`, `absorb(l2)`, que toma

como argumento uma SkiplistList, l2, esvazia-a e anexa seu conteúdo, em ordem, ao receptor. Por exemplo, se l1 contiver a, b, c e l2 contém d, e, f, depois de chamar l1.absorb(l2), l1 conterá a, b, c, d, e, f e l2 estará vazia. Esse método deve ser executado em um tempo $O(\log n)$.

Exercício 4.13. Usando as idéias da lista eficiente em termos de espaço, SEList, projete e implemente um SSet eficiente em espaço, SESSet. Para fazer isso, armazene os dados, em ordem, em uma SEList, e armazene os blocos desta SEList em um SSet. Se a implementação SSet original usa $O(n)$ espaço para armazenar n elementos, então SESSet usará espaço suficiente para n elementos mais $O(n/b + b)$ espaço perdido.

Exercício 4.14. Usando SSet como sua estrutura subjacente, projete e implemente um aplicativo que leia um arquivo de texto (grande) e permita pesquisar, de forma interativa, qualquer subcadeia contida no texto. À medida que o usuário digita sua consulta, uma parte correspondente do texto (se houver) deve aparecer como resultado.
Dica 1: Cada substring é um prefixo de algum sufixo, então basta armazenar todos os sufixos do arquivo texto.
Dica 2: Qualquer sufixo pode ser representado de forma compacta como um inteiro simples indicando onde o sufixo começa no texto.
Teste sua aplicação em alguns textos grandes, como alguns dos livros disponíveis no Project Gutenberg [1]. Se for feito corretamente, suas aplicações serão bem responsivas; não deve haver atraso notável entre as teclas de digitação e os resultados.

Exercício 4.15. (Este exercício deve ser feito depois de ler sobre árvores de busca binária, em Seção 6.2.) Compare as skiplists com árvores de pesquisa binária das seguintes maneiras:

1. Explicar como a remoção de algumas arestas de uma skiplists leva a uma estrutura que se parece a uma árvore binária e é semelhante a uma árvore de pesquisa binária.

2. Skiplists e árvores de pesquisa binária usam cada uma o mesmo número de ponteiros (2 por nó). As skiplists fazem um melhor uso desses ponteiros. Explique o porquê.

Capítulo 5

Tabelas Hash

As tabelas Hash são um método eficiente de armazenar um número pequeno, n, de números inteiros de uma grande faixa $U = \{0, \ldots, 2^w - 1\}$. O termo *tabela hash* inclui uma ampla gama de estruturas de dados. A primeira parte deste capítulo concentra-se em duas das implementações mais comuns de tabelas de hash: hashing com encadeamento e sondagem linear.

Muitas vezes, as tabelas hash armazenam tipos de dados que não são inteiros. Nesse caso, um *código hash* inteiro está associado a cada item de dados e é usado na tabela hash. A segunda parte deste capítulo discute como esses códigos de hash são gerados.

Alguns dos métodos utilizados neste capítulo exigem escolhas aleatórias de números inteiros em algum intervalo específico. Nos exemplos de código, alguns desses números inteiros "aleatórios" são constantes codificadas. Essas constantes foram obtidas usando bits aleatórios gerados pelo ruído atmosférico.

5.1 `ChainedHashTable`: Uma Tabela de Dispersão por Encadeamento

Uma estrutura de dados `ChainedHashTable` usa *dispersão por encadeamento* para armazenar dados como um array, `t`, de listas. Um número inteiro, `n`, acompanha o número total de itens em todas as listas (veja Figura 5.1):

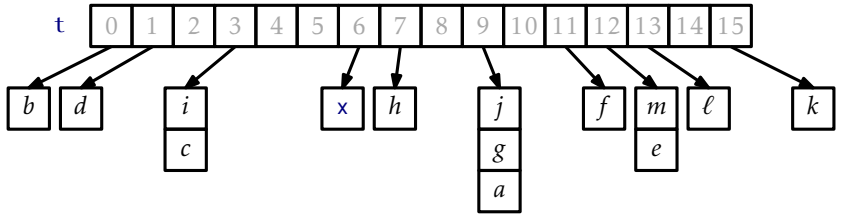


Figura 5.1: Um exemplo de `ChainedHashTable` com `n` = 14 e `t.length` = 16. Nesse exemplo `hash(x)` = 6

ChainedHashTable
```
array<List> t;
int n;
```

O *valor hash* de um item de dados `x`, indicado por `hash(x)`, é um valor na faixa {0,...,`t.length` − 1}. Todos os itens com valor de hash `i` são armazenados na lista em `t[i]`. Para garantir que as listas não sejam grandes, mantemos o invariante

$$n \le t.length$$

assim, a média de números armazenados na lista será $n/t.length \le 1$.

Para adicionar um elemento, `x`, à tabela de hash, primeiro verificamos se o comprimento de `t` precisa ser aumentado e, se assim for, crescemos `t`. Com isso resolvido, vamos decodificar `x` por meio de uma função hash para obtermos um número inteiro, `i`, no intervalo {0,...,`t.length` − 1} e anexamos `x` à lista `t[i]`:

ChainedHashTable
```
bool add(T x) {
  if (find(x) != null) return false;
  if (n+1 > t.length) resize();
  t[hash(x)].add(x);
  n++;
  return true;
}
```

Crescer a tabela, se necessário, envolve duplicar o comprimento de `t` e reinserir todos os elementos na nova tabela. Esta estratégia é exatamente a mesma que a utilizada na implementação da `ArrayStack` e o mesmo

resultado se aplica: O custo de crescimento é apenas constante quando amortizado em uma sequência de inserções (veja Lema 2.1 na página 36).

Além de crescer, o único outro trabalho feito ao adicionar um novo valor x a uma ChainedHashTable envolve anexar x à lista t[hash(x)]. Para qualquer uma das implementações de lista descritas nos Capítulos 2 ou 3, isso leva um tempo constante.

Para remover um elemento, x, da tabela hash, iteramos sobre a lista t[hash(x)] até encontrar x para, então, removê-lo:

ChainedHashTable
```
T remove(T x) {
  int j = hash(x);
  for (int i = 0; i < t[j].size(); i++) {
    T y = t[j].get(i);
    if (x == y) {
      t[j].remove(i);
      n--;
      return y;
    }
  }
  return null;
}
```

Isso leva $O(n_{\texttt{hash(x)}})$ tempo, onde n_i indica o comprimento da lista armazenada em t[i].

Procurar o elemento x em uma tabela de hash é semelhante. Realizamos uma pesquisa linear na lista t[hash(x)]:

ChainedHashTable
```
T find(T x) {
  int j = hash(x);
  for (int i = 0; i < t[j].size(); i++)
    if (x == t[j].get(i))
      return t[j].get(i);
  return null;
}
```

Novamente, isso leva tempo proporcional ao comprimento da lista t[hash(x)].

O desempenho de uma tabela hash depende criticamente da escolha da função hash. Uma boa função de hash irá espalhar os elementos uni-

formemente entre as listas t.length, de modo que o tamanho esperado da lista t[hash(x)] será $O(n/t.length) = O(1)$. Por outro lado, uma má função de hash espalhará todos os valores (incluindo x) para a mesma localização da tabela, caso em que o tamanho da lista t[hash(x)] será n. Na próxima seção, descrevemos uma boa função de hash.

5.1.1 Hash Multiplicativo

O hashing multiplicativo é um método eficiente de gerar valores de hash com base na aritmética modular (discutido em Seção 2.3) e na divisão de números inteiros. Ele usa o operador div, que calcula a parte inteira de um quociente, ao descartar o restante. Formalmente, para qualquer número inteiro $a \geq 0$ e $b \geq 1$, $a \operatorname{div} b = \lfloor a/b \rfloor$.

No hash multiplicativo, usamos uma tabela hash de tamanho 2^d para um número inteiro d qualquer (chamado *dimensão*). A função hash de um inteiro $x \in \{0, \ldots, 2^w - 1\}$ é

$$\text{hash}(x) = ((z \cdot x) \bmod 2^w) \operatorname{div} 2^{w-d} \ .$$

Aqui, z é um inteiro *ímpar* escolhido aleatoriamente em $\{1, \ldots, 2^w - 1\}$. Esta função de hash pode ser realizada de forma muito eficiente observando que, por padrão, as operações em números inteiros já são feitas em modulo 2^w onde w é o número de bits em um número inteiro. [1] (Ver Figura 5.2.) Além disso, a divisão inteira por 2^{w-d} é equivalente a descartar os $w - d$ bits mais à direita em uma representação binária (que é implementado deslocando os bits da direita por $w - d$ usando o operador >>). Desta forma, o código que implementa a fórmula acima é mais simples que a própria fórmula:

ChainedHashTable

```
int hash(T x) {
  return ((unsigned)(z * hashCode(x))) >> (w-d);
}
```

O lema a seguir, cuja prova é adiada até mais tarde nesta seção, mostra que o hash multiplicativo faz um bom trabalho para evitar colisões:

[1] Isso é verdadeiro para a maioria das linguagens de programação, incluindo C, C#, C++ e Java. Exceções notáveis são Python e Ruby, no qual o resultado de uma operação de números inteiros de comprimento w-bit fixo, é atualizado para uma representação de comprimento variável.

2^w (4294967296)	10000000000000000000000000000000
z (4102541685)	11110100100001111101000101110101
x (42)	00000000000000000000000000101010
$z \cdot x$	10100000011110010010000101110100110010
$(z \cdot x) \bmod 2^w$	00011110010010000101110100110010
$((z \cdot x) \bmod 2^w) \operatorname{div} 2^{w-d}$	00011110

Figura 5.2: A operação de um hash multiplicativo com $w = 32$ e $d = 8$.

Lema 5.1. *Seja x e y dois valores em $\{0,\ldots,2^w-1\}$ com $x \neq y$. Então* $\Pr\{\mathtt{hash}(x) = \mathtt{hash}(y)\} \leq 2/2^d$.

Com o Lema 5.1, o desempenho de `remove(x)` e `find(x)` são fáceis de analisar:

Lema 5.2. *Para qualquer valor de dados x, o comprimento esperado da lista $t[\mathtt{hash}(x)]$ é no máximo $n_x + 2$, onde n_x é o número de ocorrências de x na tabela hash.*

Demonstração. Tomando S pelo (multi-)conjunto de elementos armazenados na tabela hash que não são iguais a x. Para um elemento $y \in S$, define-se a variável indicadora

$$I_y = \begin{cases} 1 & \text{se } \mathtt{hash}(x) = \mathtt{hash}(y) \\ 0 & \text{caso contrário} \end{cases}$$

E observe que, pelo Lema 5.1, $\mathrm{E}[I_y] \leq 2/2^d = 2/\mathtt{t.length}$. O comprimento esperado da lista $t[\mathtt{hash}(x)]$ é dado por:

$$\begin{aligned} \mathrm{E}[t[\mathtt{hash}(x)].\mathtt{size}()] &= \mathrm{E}\left[n_x + \sum_{y \in S} I_y\right] \\ &= n_x + \sum_{y \in S} \mathrm{E}[I_y] \\ &\leq n_x + \sum_{y \in S} 2/\mathtt{t.length} \\ &\leq n_x + \sum_{y \in S} 2/n \\ &\leq n_x + (n - n_x)2/n \\ &\leq n_x + 2 \ , \end{aligned}$$

conforme exigido. □

Agora, queremos provar Lema 5.1, mas primeiro precisamos de um resultado da teoria dos números. Na seguinte prova, usamos a notação $(b_r,\ldots,b_0)_2$ para denotar $\sum_{i=0}^{r} b_i 2^i$, onde cada b_i é um pouco, seja 0 ou 1. Em outras palavras, $(b_r,\ldots,b_0)_2$ é o inteiro cuja representação binária é dada por $b_r,\ldots,b_0$. Usamos $\star$ para denotar um bit de valor desconhecido.

Lema 5.3. *Seja S o conjunto de inteiros ímpares em $\{1,\ldots,2^w-1\}$; Seja q e i dois elementos em S. Então há exatamente um valor $z \in S$, de modo que $zq \bmod 2^w = i$.*

Demonstração. Uma vez que o número de escolhas para z e i é o mesmo, basta provar que existe *no máximo* um valor $z \in S$ que satisfaça $zq \bmod 2^w = i$.

Suponhamos, por uma questão de contradição, que existem dois desses valores z e z', com $z > z'$. Então

$$zq \bmod 2^w = z'q \bmod 2^w = i$$

Portanto

$$(z-z')q \bmod 2^w = 0$$

Mas isso significa que

$$(z-z')q = k2^w \tag{5.1}$$

para qualquer inteiro k. Pensando em termos de números binários, temos que

$$(z-z')q = k \cdot (1, \underbrace{0,\ldots,0}_{w})_2 \ ,$$

de modo que os w bits de fuga na representação binária de $(z-z')q$ são todos os 0's.

Além disso, $k \neq 0$, desde $q \neq 0$ e $z - z' \neq 0$. Uma vez que q é impar, não possui 0's na sua representação binária.

$$q = (\star,\ldots,\star,1)_2 \ .$$

Desde $|z-z'| < 2^w$, $z-z'$ tem menos do que w 0's seguidos na sua representação binária:

$$z - z' = (\star,\ldots,\star,1,\underbrace{0,\ldots,0}_{<w})_2 \ .$$

Portanto, o produto $(z - z')q$ tem menos do que w 0's na sua representação binária:

$$(z - z')q = (\star, \cdots, \star, 1, \underbrace{0, \ldots, 0}_{<w})_2 \ .$$

Dessa forma, $(z - z')q$ não pode satisfazer (5.1), produzindo uma contradição e completando a prova. □

A utilidade do Lema 5.3 vem da seguinte observação: Se z for escolhido uniformemente aleatoriamente de S, então zt é distribuído uniformemente em S. Na seguinte prova, ajuda a pensar na representação binária de z, que consiste em $w - 1$ bits aleatórios seguido por um 1.

Prova do Lema 5.1. Primeiro, observamos que a condição $\mathsf{hash}(x) = \mathsf{hash}(y)$ é equivalente à declaração "a ordem mais alta d bits de $zx \bmod 2^w$ e os d bits de ordem superior $zy \bmod 2^w$ são os mesmos." Uma condição necessária dessa afirmação é que os bits d de ordem superior na representação binária de $z(x - y)\ bmod 2^w$ são todos 0 ou todos 1. Isso é,

$$z(x - y) \bmod 2^w = (\underbrace{0, \ldots, 0}_{d}, \underbrace{\star, \ldots, \star}_{w-d})_2 \tag{5.2}$$

quando $zx \bmod 2^w > zy \bmod 2^w$ ou

$$z(x - y) \bmod 2^w = (\underbrace{1, \ldots, 1}_{d}, \underbrace{\star, \ldots, \star}_{w-d})_2 \ . \tag{5.3}$$

quando $zx \bmod 2^w < zy \bmod 2^w$. Portanto, apenas temos que limitar a probabilidade de que $z(x - y) \bmod 2^w$ pareça (5.2) ou (5.3).

Seja q um inteiro ímpar exclusivo tal que $(x - y) \bmod 2^w = q2^r$ para algum inteiro $r \geq 0$. Pela Lema 5.3, a representação binária de $zq \bmod 2^w$ tem $w - 1$ bits aleatórios, seguido por um 1:

$$zq \bmod 2^w = (\underbrace{b_{w-1}, \ldots, b_1}_{w-1}, 1)_2$$

Portanto, a representação binária de $z(x - y) \bmod 2^w = zq2^r \bmod 2^w$ tem $w - r - 1$ bits aleatórios, seguido de um 1, seguido de r 0's:

$$z(x - y) \bmod 2^w = zq2^r \bmod 2^w = (\underbrace{b_{w-r-1}, \ldots, b_1}_{w-r-1}, 1, \underbrace{0, 0, \ldots, 0}_{r})_2$$

Agora podemos finalizar a prova: se $r > w - d$, então os d bits de ordem superior de $z(x - y) \bmod 2^w$ contêm tanto 0's quanto 1's, então a probabilidade de que $z(x-y) \bmod 2^w$ pareça (5.2) ou (5.3) é 0 . Se $r = w-d$, então a probabilidade de se parecer com (5.2) é 0, mas a probabilidade de se parecer com (5.3) é $1/2^{d-1} = 2/2^d$ (uma vez que devemos ter $b_1,\ldots,b_{d-1} = 1,\ldots,1$). Se $r < w - d$, então devemos ter $b_{w-r-1},\ldots,b_{w-r-d} = 0,\ldots,0$ ou $b_{w-r-1},\ldots,b_{w-r-d} = 1,\ldots,1$. A probabilidade de cada um desses casos é $1/2^d$ e eles são mutuamente exclusivos, então a probabilidade de um desses casos é $2/2^d$. Isso completa a prova. □

5.1.2 Resumo

O seguinte teorema resume o desempenho de uma estrutura de dados `ChainedHashTable`:

Teorema 5.1. *Uma `ChainedHashTable` implementa a interface `USet`. Ignorando o custo das chamadas para* `grow()`, *a `ChainedHashTable` suporta as operações* `add(x)`, `remove(x)` *e* `find(x)` *em $O(1)$ tempo esperado por operação.*

Além disso, começando com uma `ChainedHashTable` vazia, qualquer sequência de operações de m `add(x)` *e* `remove(x)` *resultará num total de $O(m)$ tempo gasto durante todas as chamadas para* `grow()`.

5.2 `LinearHashTable`: Sondagem Linear

A estrutura de dados `ChainedHashTable` usa uma matriz de listas, onde a `i`-ésima lista armazena todos os elementos `x`, tal que `hash(x) = i`. Uma alternativa, chamada *endereçamento aberto* é armazenar os elementos diretamente em um array, `t`, com cada local do array em `t` armazenando no máximo um valor. Essa abordagem é tomada pela `LinearHashTable` descrita nesta seção. Em alguns lugares, esta estrutura de dados é descrita como *endereçamento aberto com sondagem linear*.

A principal ideia por trás de uma `LinearHashTable` é que gostaríamos, de preferência, de armazenar o elemento `x` com o valor hash `i = hash(x)` na localização da tabela `t[i]`. Se não pudermos fazer isso (porque algum elemento já está armazenado), então tentamos armazená-lo na localização `t[(i + 1) mod t.length]`; Se isso não for possível, tentemos `t[(i +`

2) mod t.length], e assim por diante, até encontrar um lugar para x.

Há três tipos de registros armazenados em t:

1. dados: valores reais no USet que estamos representando;
2. valores null: em locais de matriz onde nenhum dado foi armazenado; e
3. valores del: em locais de matriz onde os dados foram armazenados uma vez, mas que já foram excluídos.

Além do contador, n, que acompanha o número de elementos na LinearHashTable, um contador, q, acompanha o número de elementos dos Tipos 1 e 3. Isso é, q é igual a n mais o número de del valores em t. Para que isso funcione de forma eficiente, precisamos que t seja consideravelmente maior do que q, de modo que existam muitos valores null em t. As operações na LinearHashTable mantêm, portanto, o invariante que $t.length \geq 2q$.

Para resumir, um LinearHashTable contém uma matriz, t, que armazena elementos de dados e números inteiros n e q que acompanham o número de elementos de dados e valores não null de t, respectivamente. Como muitas funções de hash funcionam apenas para tamanhos de tabela que são uma potência de 2, também mantemos um inteiro d e asseguramos o invariante $t.length = 2^d$.

LinearHashTable

```
array<T> t;
int n;   // number of values in T
int q;   // number of non-null entries in T
int d;   // t.length = 2^d
```

A operação find(x) em LinearHashTable é simples. Nós começamos na entrada do array t[i] onde i = hash(x) e pesquisamos as entradas t[i], t[(i + 1) mod t.length], t[(i + 2) mod t.length], e assim por diante, até encontrarmos um índice i' tal que, também, t[i'] = x, ou t[i'] = null. No primeiro caso, nós retornamos t[i']. No último caso, concluímos que x não está contido na tabela hash e retorna null.

LinearHashTable

```
T find(T x) {
  int i = hash(x);
```

```
  while (t[i] != null) {
    if (t[i] != del && t[i] == x) return t[i];
    i = (i == t.length-1) ? 0 : i + 1; // increment i
  }
  return null;
}
```

A operação add(x) também é bastante fácil de implementar. Depois de verificar que x ainda não está armazenado na tabela (usando find(x)), buscamos t[i], t[(i + 1) mod t.length], t[(i + 2) mod t.length], e assim por diante, até encontrar null ou del e armazenar x nessa localização, incrementar n e q, se apropriado.

LinearHashTable

```
bool add(T x) {
  if (find(x) != null) return false;
  if (2*(q+1) > t.length) resize();   // max 50% occupancy
  int i = hash(x);
  while (t[i] != null && t[i] != del)
    i = (i == t.length-1) ? 0 : i + 1; // increment i
  if (t[i] == null) q++;
  n++;
  t[i] = x;
  return true;
}
```

Agora, a implementação da operação remove(x) deve ser óbvia. Nós buscamos t[i], t[(i + 1) mod t.length], t[(i + 2) mod t.length], e assim por diante até encontrar um índice i′ tal que t[i′] = x ou t[i′] = null. No primeiro caso, definimos t[i′] = del e retornamos true. No último caso, concluímos que x não foi armazenado na tabela (e, portanto, não pode ser excluído) e retornar false.

LinearHashTable

```
T remove(T x) {
  int i = hash(x);
  while (t[i] != null) {
    T y = t[i];
    if (y != del && x == y) {
      t[i] = del;
      n--;
      if (8*n < t.length) resize(); // min 12.5% occupancy
```

```
      return y;
    }
    i = (i == t.length-1) ? 0 : i + 1;  // increment i
  }
  return null;
}
```

A correção dos métodos `find(x)`, `add(x)` e `remove(x)` é fácil de verificar, contudo ela se apoia no uso de valores `del`. Observe que nenhuma dessas operações nunca estabeleceu uma entrada não-`null` para `null`. Portanto, quando alcançamos um índice `i'` tal que `t[i'] = null`, esta é uma prova de que o elemento `x`, que estamos procurando, não está armazenado na tabela; `t[i']` sempre foi `null`, então não há nenhuma razão para que uma operação anterior `add(x)` tenha prosseguido além do índice `i'`.

O método `resize()` é chamado por `add(x)` quando o número de entradas não-`null` excede `t.length`/2 ou `remove(x)` quando o número de entradas de dados é inferior a `t.length`/8. O método `resize()` funciona como os métodos `resize()` de outras estruturas de dados baseadas em array. Encontramos o menor inteiro não negativo `d` tal que $2^d \geq 3n$. Realocamos o array `t` para que ela tenha tamanho 2^d e, em seguida, inserimos todos os elementos da versão anterior de `t` na nova cópia redimensionada de `t`. Ao fazer isso, restabelecemos `q` igual a `n`, pois o recém-alocado `t` não contém valores `del`.

LinearHashTable

```
void resize() {
  d = 1;
  while ((1<<d) < 3*n) d++;
  array<T> tnew(1<<d, null);
  q = n;
  // insert everything into tnew
  for (int k = 0; k < t.length; k++) {
    if (t[k] != null && t[k] != del) {
      int i = hash(t[k]);
      while (tnew[i] != null)
        i = (i == tnew.length-1) ? 0 : i + 1;
      tnew[i] = t[k];
    }
  }
  t = tnew;
```

```
}
```

5.2.1 Análise da Sondagem Linear

Observe que cada operação, `add(x)`, `remove(x)` ou `find(x)`, termina assim que (ou antes que) descubra a primeira entrada `null` em `t`. A intuição por trás da análise de sondagem linear é que, uma vez que pelo menos metade dos elementos em `t` são iguais a `null`, uma operação não demorará muito para ser concluída, pois será muito rápido encontrar uma entrada `null`. No entanto, não devemos confiar muito nessa intuição, porque isso nos levaria à conclusão (incorreta) de que o número esperado de locais em `t` examinado por uma operação é no máximo de 2.

Para o restante desta seção, assumiremos que todos os valores de hash são distribuídos de forma independente e uniforme em $\{0,\ldots,$ `t.length` $-1\}$. Esta não é uma suposição realista, mas permitirá que analisemos a sondagem linear. Mais tarde, nesta seção, descreveremos um método, chamado de hash de tabulação, que produz uma função de hash que é "suficientemente boa"para sondagem linear. Também assumiremos que todos os índices nas posições de `t` são tomados no módulo `t.length`, de modo que `t[i]` é realmente uma abreviatura para `t[i` mod `t.length]`.

Dizemos que um *percurso de tamanho k que começa em* `i` ocorre quando todas as entradas da tabela `t[i]`, `t[i + 1]`, $\ldots$, `t[i` $+k-1$`]` não são `null` e `t[i` -1`] = t[i` $+k$`] = null`. O número de elementos não-`null` de `t` é exatamente `q` e o método `add(x)` garante que, em todos os momentos, `q` $\leq$ `t.length`$/2$. Existem `q` elementos $x_1,\ldots,x_q$ que foram inseridos em `t` desde a última operação `resize()`. Por nossa suposição, cada um deles tem um valor de hash, `hash(`x_j`)`, Isso é uniforme e independente do resto. Com esta configuração, podemos provar o lema principal necessário para analisar a sondagem linear.

Lema 5.4. *Corrija um valor* `i` $\in \{0,\ldots,$ `t.length` $-1\}$. *Então, a probabilidade de que um período de k começado em* `i` *seja* $O(c^k)$ *para alguma constante* $0 < c < 1$.

Demonstração. Se um período de k começar em `i`, então existem exatamente k elementos x_j such that `hash(`x_j`)` $\in \{$`i`$,\ldots,$ `i`$+k-1\}$. A probabilidade

de isso ocorrer é exatamente

$$p_k = \binom{\mathtt{q}}{k}\left(\frac{k}{\mathtt{t.length}}\right)^k\left(\frac{\mathtt{t.length}-k}{\mathtt{t.length}}\right)^{\mathtt{q}-k},$$

uma vez que, para cada escolha de k elementos, esses k elementos devem se dispersar para um dos k locais e o restante $\mathtt{q}-k$ elementos devem se dispersar para os outros $\mathtt{t.length}-k$ locais da tabela. [2]

Na derivação a seguir, vamos trapacear um pouco e substituir $r!$ por $(r/e)^r$. A Aproximação de Stirling (Seção 1.3.2) mostra que isso é apenas um fator de $O(\sqrt{r})$ da verdade. Isso é feito apenas para tornar a derivação mais simples; Exercício 5.4 pede ao leitor para refazer o cálculo de forma mais rigorosa usando a Aproximação de Stirling na sua totalidade.

O valor de p_k é máximo quando $\mathtt{t.length}$ é mínimo, e a estrutura de dados mantém a invariante de $\mathtt{t.length} \geq 2\mathtt{q}$, então

$$\begin{aligned}
p_k &\leq \binom{\mathtt{q}}{k}\left(\frac{k}{2\mathtt{q}}\right)^k\left(\frac{2\mathtt{q}-k}{2\mathtt{q}}\right)^{\mathtt{q}-k} \\
&= \left(\frac{\mathtt{q}!}{(\mathtt{q}-k)!k!}\right)\left(\frac{k}{2\mathtt{q}}\right)^k\left(\frac{2\mathtt{q}-k}{2\mathtt{q}}\right)^{\mathtt{q}-k} \\
&\approx \left(\frac{\mathtt{q}^{\mathtt{q}}}{(\mathtt{q}-k)^{\mathtt{q}-k}k^k}\right)\left(\frac{k}{2\mathtt{q}}\right)^k\left(\frac{2\mathtt{q}-k}{2\mathtt{q}}\right)^{\mathtt{q}-k} \qquad \text{[Aproximação de Stirling]} \\
&= \left(\frac{\mathtt{q}^k\mathtt{q}^{\mathtt{q}-k}}{(\mathtt{q}-k)^{\mathtt{q}-k}k^k}\right)\left(\frac{k}{2\mathtt{q}}\right)^k\left(\frac{2\mathtt{q}-k}{2\mathtt{q}}\right)^{\mathtt{q}-k} \\
&= \left(\frac{\mathtt{q}k}{2\mathtt{q}k}\right)^k\left(\frac{\mathtt{q}(2\mathtt{q}-k)}{2\mathtt{q}(\mathtt{q}-k)}\right)^{\mathtt{q}-k} \\
&= \left(\frac{1}{2}\right)^k\left(\frac{(2\mathtt{q}-k)}{2(\mathtt{q}-k)}\right)^{\mathtt{q}-k} \\
&= \left(\frac{1}{2}\right)^k\left(1+\frac{k}{2(\mathtt{q}-k)}\right)^{\mathtt{q}-k} \\
&\leq \left(\frac{\sqrt{e}}{2}\right)^k .
\end{aligned}$$

(Na última etapa, usamos a desigualdade $(1+1/x)^x \leq e$, que é válida para todos os $x > 0$.) Como $\sqrt{e}/2 < 0.824360636 < 1$, isso completa a prova. □

[2] Note que p_k é maior do que a probabilidade de que um percurso k comece em $\mathtt{i}$, uma vez que a definição de p_k não inclui o requisito $\mathtt{t}[\mathtt{i}-1] = \mathtt{t}[\mathtt{i}+k] = \mathtt{null}$.

Usando Lema 5.4 para provar limites superiores no tempo de execução esperado de `find(x)`, `add(x)` e `remove(x)` agora é bastante sucinto. Considere o caso mais simples, onde executamos `find(x)` para algum valor `x` que nunca tenha sido armazenado no `LinearHashTable`. Nesse caso, `i` = `hash(x)` é um valor aleatório em $\{0,\ldots,\texttt{t.length}-1\}$ independentemente do conteúdo de `t`. Se `i` for parte de uma extensão de k, então o tempo necessário para executar a operação `find(x)` é no máximo $O(1+k)$. Assim, o tempo de execução esperado pode ser delimitado por

$$O\left(1+\left(\frac{1}{\texttt{t.length}}\right)\sum_{i=1}^{\texttt{t.length}}\sum_{k=0}^{\infty} k\Pr\{\texttt{i} \text{ é parte de uma série } k\}\right) .$$

Observe que cada rodada de comprimento k contribui para a soma interna k vezes para uma contribuição total de k^2, então a soma acima pode ser reescrita como

$$\begin{aligned}
&O\left(1+\left(\frac{1}{\texttt{t.length}}\right)\sum_{i=1}^{\texttt{t.length}}\sum_{k=0}^{\infty} k^2\Pr\{\texttt{i} \text{ começa uma série } k\}\right)\\
&\le O\left(1+\left(\frac{1}{\texttt{t.length}}\right)\sum_{i=1}^{\texttt{t.length}}\sum_{k=0}^{\infty} k^2 p_k\right)\\
&= O\left(1+\sum_{k=0}^{\infty} k^2 p_k\right)\\
&= O\left(1+\sum_{k=0}^{\infty} k^2\cdot O(c^k)\right)\\
&= O(1) .
\end{aligned}$$

O último passo nesta derivação vem do fato de que $\sum_{k=0}^{\infty} k^2\cdot O(c^k)$ é uma série exponencialmente decrescente. [3] Portanto, concluimos que o tempo de execução esperado da operação `find(x)` para um valor `x` que não está contido em `LinearHashTable` é $O(1)$.

Se ignorarmos o custo da operação `resize()`, a análise acima nos dá tudo o que precisamos para analisar o custo das operações em um `LinearHashTable`.

[3] Na terminologia de muitos textos de cálculo, esta soma passa o teste de razão: existe um inteiro positivo k_0 tal que, para todos $k \ge k_0$, $\frac{(k+1)^2 c^{k+1}}{k^2 c^k} < 1$.

Em primeiro lugar, a análise de `find(x)` fornecida acima aplica-se à operação `add(x)` quando `x` não está contido na tabela. Para analisar a operação `find(x)` quando `x` estiver contida na tabela, precisamos apenas observar que isso é o mesmo que o custo da operação `add(x)` que adicionou `x` na tabela anterior. Finalmente, o custo de uma operação `remove(x)` é igual ao custo de uma operação `find(x)`.

Em resumo, se ignorarmos o custo das chamadas para `resize()`, todas as operações em `LinearHashTable` são executadas em $O(1)$ tempo esperado. A contabilização do custo do redimensionamento pode ser feita usando o mesmo tipo de análise amortizada realizada para a estrutura de dados `ArrayStack` em Seção 2.1.

5.2.2 Resumo

O seguinte teorema resume o desempenho da estrutura de dados `LinearHashTable`:

Teorema 5.2. *A `LinearHashTable` implementa a interface `USet`. Ignorando o custo das chamadas para `resize()`, `LinearHashTable` suporta as operações `add(x)`, `remove(x)` e `find(x)` em $O(1)$ tempo esperado por operação.*

Além disso, começando uma `LinearHashTable` vazia, qualquer sequência de m `add(x)` e `remove(x)` operações resulta em um total de $O(m)$ tempo gasto durante todas as chamadas para `resize()`.

5.2.3 Hashing por Tabulação

Ao analisar a estrutura `LinearHashTable`, fizemos uma suposição muito forte: que para qualquer conjunto de elementos, $\{x_1,\ldots,x_n\}$, os valores de hash $\texttt{hash}(x_1),\ldots,\texttt{hash}(x_n)$ são distribuídos de forma independente e uniforme sobre o conjunto $\{0,\ldots,\texttt{t.length}-1\}$. Uma maneira de conseguir isso é armazenar em um array gigante, `tab`, de comprimento 2^w, onde cada entrada é um inteiro de `w`-bit aleatório, independente de todas as outras entradas. Desta forma, podemos implementar `hash(x)` extraindo um inteiro de `d`-bit de `tab[x.hashCode()]`:

LinearHashTable

```
int idealHash(T x) {
  return tab[hashCode(x) >> w-d];
```

```
}
```

Infelizmente, armazenar uma matriz de tamanho 2^w é proibitivo em termos de uso de memória. A abordagem usada pela *hashing por tabulação* é, em vez disso, tratar números w-bit como sendo compostos de w/r inteiros, cada um com apenas r bits. Desta forma, o hashing de tabulação só precisa de w/r arrays cada um do comprimento 2^r. Todas as entradas nesses arrays são w-bit inteiros aleatoriamente independentes. Para obter o valor de hash(x), divide-se os números inteiros x.hashCode() em w/r r-bit e usamos estes como índices nesses arrays. Em seguida, combinamos todos esses valores com o operador exclusivo de bit a bit para obter hash(x). O código a seguir mostra como isso funciona quando w = 32 e r = 4:

LinearHashTable

```
int hash(T x) {
  unsigned h = hashCode(x);
  return (tab[0][h&0xff]
       ^ tab[1][(h>>8)&0xff]
       ^ tab[2][(h>>16)&0xff]
       ^ tab[3][(h>>24)&0xff])
      >> (w-d);
}
```

Nesse caso, tab é um array bidimensional com quatro colunas e $2^{32/4} = 256$ linhas.

Pode-se verificar facilmente que, para qualquer x, hash(x) é uniformemente distribuído em $\{0,\ldots,2^d-1\}$. Com um pouco de trabalho, pode-se verificar se qualquer par de valores possui valores de hash independentes. Isso implica que o hash de tabulação poderia ser usado em lugar de hashing multiplicativo para a implementação ChainedHashTable.

No entanto, não é verdade que qualquer conjunto de n valores distintos forneça um conjunto de valores de hash independentes n. Contudo, quando o hashing de tabulação é usado, o limite de Teorema 5.2 ainda é válido. Referências para isso são fornecidas no final deste capítulo.

5.3 Hash Codes

As tabelas de hash discutidas na seção anterior são usadas para associar dados com chaves inteiras consistindo de `w` bits. Em muitos casos, temos chaves que não são inteiros. Podem ser strings, objetos, arrays ou outras estruturas compostas. Para usar tabelas de hash para esses tipos de dados, devemos mapear esses tipos de dados para os códigos de hash com `w`-bits. Os mapeamentos de código Hash devem ter as seguintes propriedades:

1. Se `x` e `y` forem iguais, então `x`.hashCode() e `y`.hashCode() são iguais.
2. Se `x` e `y` não forem iguais, então a probabilidade de que `x`.hashCode() = `y`.hashCode() sejam iguais deve ser pequena (perto de $1/2^w$).

A primeira propriedade garante que, se armazenarmos `x` em uma tabela hash e, mais tarde, procuremos um valor `y` igual a `x`, então encontraremos `x` — como deveríamos. A segunda propriedade minimiza a perda de converter nossos objetos em números inteiros. Ele garante que os objetos desiguais geralmente tenham códigos de hash diferentes e, portanto, provavelmente serão armazenados em locais diferentes em nossa tabela de hash.

5.3.1 Códigos Hash para Tipos Primitivos de Dados

Pequenos tipos de dados primitivos, como `char`, `byte`, `int` e `float`, geralmente, são fáceis de encontrar códigos de hash. Esses tipos de dados sempre têm uma representação binária e essa representação binária geralmente consiste em `w` ou menos bits. (Por exemplo, em C ++ `char` geralmente é um tipo de 8 bits e `float` é um tipo de 32 bits). Nesses casos, tratamos esses bits como a representação de um número inteiro na faixa $\{0,\ldots,2^w-1\}$. Se dois valores são diferentes, eles obtêm códigos hash diferentes. Se eles são o mesmo, eles obtêm o mesmo código hash.

Alguns tipos de dados primitivos são compostos por mais de `w` bits, geralmente c`w` bits para algum inteiro constante c. (Os tipos `long` e `double` do Java são exemplos disso com $c = 2$.) Esses tipos de dados podem ser tratados como objetos compostos feitos de c partes, conforme descrito na próxima seção.

5.3.2 Códigos Hash para Objetos Compostos

Para um objeto composto, queremos criar um código hash combinando os códigos hash individuais das partes constituintes do objeto. Isso não é tão fácil quanto parece. Embora se possa encontrar muitos hacks para isso (por exemplo, combinando os códigos de hash com operações bitwise ou-exclusivo), muitos desses hacks terminam por serem frustrantes (ver exercícios 5.7–5.9). No entanto, se alguém estiver disposto a fazer aritmética com $2w$ bits de precisão, existem métodos simples e robustos disponíveis. Suponha que possamos ter um objeto composto por várias partes $P_0,\ldots,P_{r-1}$ cujos códigos hash são $x_0,\ldots,x_{r-1}$. Então, podemos escolher w-bits inteiros aleatórios e mutuamente independentes $z_0,\ldots,z_{r-1}$ e um $2w$-bit inteiro impar z e, então, calcular um código hash para nosso objeto com

$$h(x_0,\ldots,x_{r-1}) = \left(\left(z\sum_{i=0}^{r-1} z_i x_i\right) \bmod 2^{2w}\right) \operatorname{div} 2^w \ .$$

Observe que este código hash tem um passo final (multiplicando por z e dividindo por 2^w) que usa a função de hash multiplicativa de Seção 5.1.1 para tirar o $2w$-bit resultado intermediário e reduzi-lo para um resultado final de w-bit. Aqui está um exemplo desse método aplicado a um objeto composto simples com três partes `x0`, `x1`, and `x2`:

Point3D
```
unsigned hashCode() {
  // random number from random.org
  long long z[] = {0x2058cc50L, 0xcb19137eL, 0x2cb6b6fdL};
  long zz = 0xbea0107e5067d19dL;
  long h0 = ods::hashCode(x0);
  long h1 = ods::hashCode(x1);
  long h2 = ods::hashCode(x2);
  return (int)(((z[0]*h0 + z[1]*h1 + z[2]*h2)*zz) >> 32);
}
```

O seguinte teorema mostra que, além de ser direto para implementar, esse método provavelmente é bom:

Teorema 5.3. *Sejam $x_0,\ldots,x_{r-1}$ e $y_0,\ldots,y_{r-1}$ sequencias de w bit inteiros em $\{0,\ldots,2^w-1\}$ e assumindo $x_i \neq y_i$ para pelo menos um índice $i \in \{0,\ldots,r-1\}$.*

Então

$$\Pr\{h(x_0,\ldots,x_{r-1}) = h(y_0,\ldots,y_{r-1})\} \leq 3/2^w \ .$$

Demonstração. Primeiro, ignoraremos o passo de hashing multiplicativo final e veremos como essa etapa contribui mais tarde. Definir:

$$h'(x_0,\ldots,x_{r-1}) = \left(\sum_{j=0}^{r-1} z_j x_j\right) \bmod 2^{2w} \ .$$

Suponhamos que $h'(x_0,\ldots,x_{r-1}) = h'(y_0,\ldots,y_{r-1})$. Podemos reescrever isso como:

$$z_i(x_i - y_i) \bmod 2^{2w} = t \tag{5.4}$$

onde

$$t = \left(\sum_{j=0}^{i-1} z_j(y_j - x_j) + \sum_{j=i+1}^{r-1} z_j(y_j - x_j)\right) \bmod 2^{2w}$$

Se assumirmos, sem perda de generalidade, que $x_i > y_i$, então (5.4) se torna

$$z_i(x_i - y_i) = t \ , \tag{5.5}$$

já que cada z_i e $(x_i - y_i)$ é ao menos $2^w - 1$, então o produto deles é ao menos $2^{2w} - 2^{w+1} + 1 < 2^{2w} - 1$. Pressuposto, $x_i - y_i \neq 0$, então (5.5) tem ao menos uma solução em z_i. Portanto, uma vez que z_i e t são independentes $z_0,\ldots,z_{r-1}$ são mutuamente independentes), a probabilidade de selecionar $h'(x_0,\ldots,x_{r-1}) = h'(y_0,\ldots,y_{r-1})$ seja no máximo $1/2^w$.

O passo final da função hash é aplicar o hashing multiplicativo para reduzir o resultado intermediário $h'(x_0,\ldots,x_{r-1})$ de $2w$-bit para resultado final $h(x_0,\ldots,x_{r-1})$ de w-bit. Pelo Teorema 5.3, se $h'(x_0,\ldots,x_{r-1}) \neq h'(y_0,\ldots,y_{r-1})$, então $\Pr\{h(x_0,\ldots,x_{r-1}) = h(y_0,\ldots,y_{r-1})\} \leq 2/2^w$.

Resumindo,

$$\begin{aligned}
&\Pr\left\{\begin{array}{l} h(x_0,\ldots,x_{r-1}) \\ \quad = h(y_0,\ldots,y_{r-1}) \end{array}\right\} \\
&= \Pr\left\{\begin{array}{l} h'(x_0,\ldots,x_{r-1}) = h'(y_0,\ldots,y_{r-1}) \text{ or} \\ h'(x_0,\ldots,x_{r-1}) \neq h'(y_0,\ldots,y_{r-1}) \\ \quad \text{and } zh'(x_0,\ldots,x_{r-1}) \operatorname{div} 2^w = zh'(y_0,\ldots,y_{r-1}) \operatorname{div} 2^w \end{array}\right\} \\
&\leq 1/2^w + 2/2^w = 3/2^w \ .
\end{aligned}$$

□

5.3.3 Códigos Hash para Arrays e Strings

O método da seção anterior funciona bem para objetos que possuem um número fixo, constante, de componentes. No entanto, ele se destrói quando queremos usá-lo com objetos que possuem uma quantidade variável de componentes, uma vez que requer um número aleatório z_i de w-bit para cada componente. Poderíamos usar uma sequência pseudo-randômica para gerar tantos z_i's quanto precisássemos, mas então os z_i não seriam mutuamente independentes e, assim, torna-se-ia difícil provar que os números de pseudo-randômicos não interagem bem com a função hash que estamos usando. Em particular, os valores de t e z_i na prova de Teorema 5.3 não são mais independentes.

Uma abordagem mais rigorosa é basear nossos códigos de hash em polinômios sobre os campos principais; Estes são apenas polinômios regulares que são avaliados em um número primo, p. Este método baseia-se no seguinte teorema, que diz que os polinômios sobre os campos principais se comportam quase como os polinômios usuais:

Teorema 5.4. *Seja p um número primo, e seja $f(z) = x_0 z^0 + x_1 z^1 + \cdots + x_{r-1} z^{r-1}$ uma expressão polinomial, não trivial, com coeficientes $x_i \in \{0,\ldots,p-1\}$. Então a equação $f(z) \bmod p = 0$ tem no mínimo $r-1$ soluções para $z \in \{0,\ldots,p-1\}$.*

Para usar o Teorema 5.4, nós "hasheamos"uma sequência de inteiros $x_0,\ldots,x_{r-1}$ com cada $x_i \in \{0,\ldots,p-2\}$ usando um inteiro aleatório $z \in \{0,\ldots,p-1\}$ via fórmula

$$h(x_0,\ldots,x_{r-1}) = \left(x_0 z^0 + \cdots + x_{r-1} z^{r-1} + (p-1)z^r\right) \bmod p \ .$$

Observe o termo extra $(p-1)z^r$ no final da fórmula. Isso ajuda a pensar em $(p-1)$ como o último elemento, x_r, na sequência x_r. Observe, também, que este elemento difere de qualquer outro elemento na sequência (cada um dos quais está no conjunto $\{0,\ldots,p-2\}$). Podemos pensar em $p-1$ como um marcador de fim de sequência.

O seguinte teorema, que considera o caso de duas sequências do mesmo comprimento, mostra que esta função hash dá um bom retorno para a pequena quantidade de aleatorização necessária para escolher z:

Teorema 5.5. *Seja* $p > 2^w + 1$ *um primo, seja* $x_0, \ldots, x_{r-1}$ *e* $y_0, \ldots, y_{r-1}$ *sequências de inteiros de* w*-bit em* $\{0, \ldots, 2^w - 1\}$*, e assumindo-se* $x_i \neq y_i$ *para pelo menos um índice* $i \in \{0, \ldots, r-1\}$*. Então*

$$\Pr\{h(x_0, \ldots, x_{r-1}) = h(y_0, \ldots, y_{r-1})\} \leq (r-1)/p \ .$$

Demonstração. A equação $h(x_0, \ldots, x_{r-1}) = h(y_0, \ldots, y_{r-1})$ pode ser reescrita como

$$\left((x_0 - y_0)z^0 + \cdots + (x_{r-1} - y_{r-1})z^{r-1}\right) \bmod p = 0. \tag{5.6}$$

Sendo $x_i \neq y_i$, um polinômio não trivial. Portanto, pelo Teorema 5.4, tem no máximo $r-1$ soluções em z. A probabilidade de escolher z para ser uma dessas soluções é, portanto, no máximo $(r-1)/p$. □

Note-se que esta função de hash também trata do caso em que duas sequências têm comprimentos diferentes, mesmo quando uma das sequências é um prefixo do outro. Isso ocorre porque esta função efetivamente colmeia a sequência infinita

$$x_0, \ldots, x_{r-1}, p-1, 0, 0, \ldots \ .$$

Isso garante que, se tivermos duas sequências de comprimento r e r' com $r > r'$, essas duas sequências diferem no índice $i = r$. Nesse caso, (5.6) se torna

$$\left(\sum_{i=0}^{i=r'-1} (x_i - y_i)z^i + (x_{r'} - p + 1)z^{r'} + \sum_{i=r'+1}^{i=r-1} x_i z^i + (p-1)z^r\right) \bmod p = 0 \ ,$$

que, pelo Teorema 5.4, tem no máximo r soluções em z. Isto combinado com Teorema 5.5 é suficiente para comprovar o seguinte teorema mais geral:

Teorema 5.6. *Seja* $p > 2^w + 1$ *um primo, seja* $x_0, \ldots, x_{r-1}$ *e* $y_0, \ldots, y_{r'-1}$ *sequências distintas de inteiros de* w*-bit em* $\{0, \ldots, 2^w - 1\}$*. Então*

$$\Pr\{h(x_0, \ldots, x_{r-1}) = h(y_0, \ldots, y_{r-1})\} \leq \max\{r, r'\}/p \ .$$

O código de exemplo a seguir mostra como esta função hash é aplicada a um objeto que contém um array, `x`, de valores:

GeomVector

```
unsigned hashCode() {
  long p = (1L<<32)-5;   // prime: 2^32 - 5
  long z = 0x64b6055aL;  // 32 bits from random.org
  int z2 = 0x5067d19d;   // random odd 32 bit number
  long s = 0;
  long zi = 1;
  for (int i = 0; i < x.length; i++) {
    // reduce to 31 bits
    long long xi = (ods::hashCode(x[i]) * z2) >> 1;
    s = (s + zi * xi) % p;
    zi = (zi * z) % p;
  }
  s = (s + zi * (p-1)) % p;
  return (int)s;
}
```

O código anterior sacrifica alguma probabilidade de colisão para conveniência de implementação. Em particular, aplica a função hash multiplicativa de Seção 5.1.1, com `d` = 31 para reduzir `x[i].hashCode()` para um valor de 31 bits. Isto é para que as adições e multiplicações que são feitas modulo o principal $p = 2^{32} - 5$ podem ser realizadas usando aritmética não assinada de 63 bits. Assim, a probabilidade de duas sequências diferentes, cujo maior comprimento é r, tendo o mesmo código de hash no máximo

$$2/2^{31} + r/(2^{32} - 5)$$

em vez de $r/(2^{32} - 5)$ especificado em Teorema 5.6.

5.4 Discussões e Exercícios

As tabelas Hash e os códigos hash representam um campo de pesquisa enorme e ativo que é apenas abordado neste capítulo. A Bibliografia online sobre Hashing [10] contém cerca de 2000 entradas.

Existe uma variedade de implementações de tabela hash diferentes. A descrita na Seção 5.1 é conhecida como *hashing com encadeamento* (cada entrada do array contém uma cadeia (`List`) de elementos). Hashing com encadeamento remonta a um memorando interno da IBM criado por H.

P. Luhn e datado de janeiro de 1953. Este memorando também parece ser uma das primeiras referências às linked lists.

Uma alternativa ao hashing com encadeamento é a usada pelos esquemas *open address*, onde todos os dados são armazenados diretamente em um array. Esses esquemas incluem a estrutura `LinearHashTable` de Seção 5.2. Essa idéia também foi proposta, independentemente, por um grupo da IBM na década de 1950. Os esquemas de endereçamento aberto devem lidar com o problema de *resolução de colisão*: index collision resolutiono caso em que dois valores hash para o mesmo local da matriz. Existem estratégias diferentes para resolução de colisão; Estes fornecem garantias de desempenho diferentes e muitas vezes exigem funções de hash mais sofisticadas do que as descritas aqui.

Outra categoria de implementações de tabela de hash são os chamados métodos de *hashing perfeito*. Estes são métodos em que as operações `find(x)` levam $O(1)$ tempo no pior caso. Para conjuntos de dados estáticos, isso pode ser conseguido encontrando *funções de hash perfeitas* para os dados; Essas são funções que mapeiam cada peça de dados para um local de matriz exclusivo. Para os dados que mudam ao longo do tempo, os métodos de hash perfeitos incluem *tabelas de hash de dois níveis FKS* [31, 24] e *cuckoo hashing* [55].

As funções de hash apresentadas neste capítulo estão provavelmente entre os métodos mais práticos atualmente conhecidos que podem comprovadamente funcionar bem para qualquer conjunto de dados. Outros métodos provavelmente bons datam do trabalho pioneiro de Carter e Wegman, que introduziram a noção de *hash universal* e descreveram várias funções de hash para diferentes cenários [14]. Hashing por tabulação, descrito em Seção 5.2.3, é graças a Carter e Wegman [14], mas sua análise, quando aplicada a sondagem linear (e vários outros esquemas de tabela de hash), é graças a Pătraşcu e Thorup [58].

A ideia do *hash multiplicativo* é muito antiga e parece ser parte do folclore hashing [48, Section 6.4]. No entanto, a idéia de escolher o multiplicador `z` para ser um número aleatório *impar* e a análise em Seção 5.1.1 é devida a Dietzfelbinger *et al.* [23]. Esta versão do hashing multiplicativo é uma das mais simples, mas a probabilidade de colisão de $2/2^d$ é um fator de dois maiores do que o que se poderia esperar com uma função

aleatória de $2^w \to 2^d$. O método *multiplica-soma hashing* usa a função

$$h(x) = ((zx + b) \bmod 2^{2w}) \operatorname{div} 2^{2w-d}$$

onde z e b são escolhidos aleatoriamente de $\{0,\ldots,2^{2w}-1\}$. O hashing Multiply-add tem uma probabilidade de colisão de apenas $1/2^d$[21], mas requer aritmética de precisão $2w$-bit.

Há uma série de métodos para obter códigos hash de sequências de comprimento fixo de w-bit inteiros. Um método particularmente rápido [11] é a função

$$h(x_0,\ldots,x_{r-1}) = \left(\sum_{i=0}^{r/2-1}((x_{2i}+a_{2i}) \bmod 2^w)((x_{2i+1}+a_{2i+1}) \bmod 2^w)\right) \bmod 2^{2w}$$

onde r é igual e $a_0,\ldots,a_{r-1}$ são escolhidos aleatoriamente de $\{0,\ldots,2^w\}$. Isso produz um código hash de $2w$-bit com probabilidade de colisão $1/2^w$. Isso pode ser reduzido para um código hash de w-bit usando o hashing multiplicativo (ou multiplicativo). Esse método é rápido porque requer apenas $r/2$ $2w$-bit multiplicações , enquanto o método descrito em Seção 5.3.2 requer r multiplicações. (As operações mod ocorrem implicitamente usando a aritmética w e $2w$-bit para as adições e multiplicações, respectivamente.)

O método de Seção 5.3.3 de usar polinômios sobre campos primos para matrizes de comprimento variável de hash e strings é devido a Dietzfelbinger *et al.* [22]. Devido ao uso do operador mod que depende de uma instrução de máquina dispendiosa, infelizmente não é muito rápido. Algumas variantes deste método escolhem o primo p para ser um da forma $2^w - 1$, caso em que o operador mod pode ser substituído por adição (+) e operações bitwise-E (&) [47, Seção 3.6]. Outra opção é aplicar um dos métodos rápidos para sequências de caracteres de comprimento fixo para blocos de comprimento c para alguma constante $c > 1$ e, em seguida, aplicar o método de campo primário à sequência resultante de $\lceil r/c \rceil$ códigos hash.

Exercício 5.1. Uma determinada universidade atribui cada um dos números de seus alunos a primeira vez que se inscreveram para qualquer curso. Esses números são números inteiros sequenciais que começaram

em 0 há muitos anos e agora estão em milhões. Suponhamos que tenhamos uma classe de alunos do primeiro ano e queremos atribuir-lhes códigos hash com base nos números de seus alunos. Faz mais sentido usar os dois primeiros dígitos ou os dois últimos dígitos do número do estudante? Justifique sua resposta.

Exercício 5.2. Considere o esquema de hashing na Seção 5.1.1, e suponha $n = 2^d$ e $d \leq w/2$.

1. Mostre que, para qualquer escolha do multiplicador, z, existe n valores que possuem o mesmo código de hash. (Sugestão: isso é fácil e não exige nenhuma teoria de números).

2. Dado o multiplicador, z, descreva n valores de modo que todos tenham o mesmo código de hash. (Sugestão: isto é mais difícil e requer uma teoria básica de números).

Exercício 5.3. Prove que o limite $2/2^d$ no Lema 5.1 seja o melhor limite possível mostrando que, se $x = 2^{w-d-2}$ e $y = 3x$, então$\Pr\{\text{hash}(x) = \text{hash}(y)\} = 2/2^d$. (Observe as representações binárias de zx e $z3x$, e use o fato de que $z3x = zx+2zx$.)

Exercício 5.4. Prove novamente o Lema 5.4 usando a versão completa da Aproximação de Stirling dada em Seção 1.3.2.

Exercício 5.5. Considere a seguinte versão simplificada do código para adicionar um elemento x a um `LinearHashTable`, que simplesmente armazena x na primeira entrada `null` do array encontrada. Explique por que isso pode ser muito lento, dando um exemplo de uma sequência de $O(n)$ `add(x)`, `remove(x)` e `find(x)` operações que levariam na ordem de n^2 tempo de executar.

LinearHashTable

```
bool addSlow(T x) {
  if (2*(q+1) > t.length) resize();   // max 50% occupancy
  int i = hash(x);
  while (t[i] != null) {
      if (t[i] != del && x.equals(t[i])) return false;
      i = (i == t.length-1) ? 0 : i + 1; // increment i
  }
  t[i] = x;
```

```
  n++; q++;
  return true;
}
```

Exercício 5.6. Versões iniciais do método Java `hashCode()` para a classe `String` trabalhada ao não usar todos os caracteres encontrados em strings longos. Por exemplo, para uma sequência de dezesseis caracteres, o código hash foi computado usando apenas os oito caracteres de índices pares. Explique por que esta foi uma idéia muito ruim dando um exemplo de grande conjunto de strings que todos têm o mesmo código hash.

Exercício 5.7. Suponha que você tenha um objeto composto por dois números `w`-bit, `x` e `y`. Mostre porque $x \oplus y$ não faz um bom código hash para o seu objeto. Dê um exemplo de um grande conjunto de objetos que todos teriam código hash 0.

Exercício 5.8. Suponha que você tenha um objeto composto por dois números `w`-bit, `x` e `y`. Mostre porque $x + y$ não faz um bom código hash para o seu objeto. Dê um exemplo de um grande conjunto de objetos que todos teriam o mesmo código hash.

Exercício 5.9. Suponha que você tenha um objeto composto por dois números `w`-bit, `x` e `y`. Suponha que o código hash para seu objeto seja definido por alguma função determinística $h(x, y)$ que produz um inteiro `w`-bit inteiro. Prove que exista um grande conjunto de objetos que tenham o mesmo código hash.

Exercício 5.10. Seja $p = 2^w - 1$ para algum inteiro positivo `w`. Explique porque, para um inteiro positivo x

$$(x \bmod 2^w) + (x \operatorname{div} 2^w) \equiv x \bmod (2^w - 1) \ .$$

(Isso nos dá um algoritmo $x \bmod (2^w - 1)$ para repetidamente "settar"

$$x = x\&((1 << w) - 1) + x >> w$$

até $x \leq 2^w - 1$.)

Exercício 5.11. Encontre algumas implementações de tabelas hash comumente usadas, tais como as implementações The C++ STL `unordered_map`

ou `HashTable` ou `LinearHashTable` neste livro e crie uma programa que armazena inteiros nesta estrutura de dados de modo que haja números inteiros, `x`, de modo que `find(x)` demore tempo linear. Ou seja, encontre um conjunto de `n` inteiros para os quais existem cn elementos que dispersem para o mesmo local da tabela.

Dependendo de quão boa seja a implementação, você poderá fazer isso apenas inspecionando o código para a implementação, ou talvez seja necessário que escreva algum código que faça inserções e pesquisas de teste, medindo quanto tempo demora para adicionar e encontrar valores específicos. (Isso pode ser, e tem sido usado, para lançar ataques de negação de serviço em servidores web [17].)

Capítulo 6

Árvores Binárias

Este capítulo introduz uma das estruturas mais fundamentais na Ciência da Computação: árvores binárias. O uso da palavra *árvore* vem do fato que, quando as desenhamos, o resultado frequentemente se assemelha às árvores de uma floresta. Existem muitas maneiras de definir uma árvore binária. Matematicamente, uma *árvore binária* é um grafo conectado, não direcionado, finito, sem ciclos e sem nenhum vértice com grau maior que três.

Para muitas aplicações na ciência da computação, árvores binárias possuem *raízes:* um nó especial, r, com grau de no máximo dois é chamado de *raiz* da árvore. Para cada nó $u \neq r$, o segundo nó no caminho de u para r é chamado de *pai* de u. Cada um dos outros nós adjacentes a u é chamado de *filho* de u. Muitas da árvores binárias em que estamos interessados são *ordenadas*, assim distinguimos entre o *filho esquerdo* e o *filho direito* de u.

Nas ilustrações, árvores binárias são comumente desenhadas da raiz para baixo, com a raiz no topo do desenho e os filhos esquerdo e direito dados respectivamente pelas posições esquerda e direita no desenho. (Figura 6.1). Por exemplo, a Figura 6.2.a mostra uma árvore binária com nove nós.

As árvores binárias são tão importantes que foi criada uma terminologia para elas: a *profundidade* de um nó, u, em uma árvore binária é o comprimento do caminho de u até a raiz da árvore. Se um nó, w, está no caminho de u até r, então w é chamado de *ancestral* de u e u um *descendente* de w. A *subárvore* de um nó, u, é uma árvore binária com raiz em u e con-

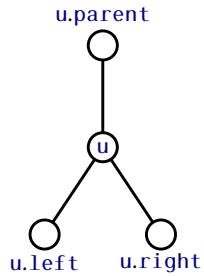


Figura 6.1: O pai, o filho esquerdo e o filho direito do nó u em uma BinaryTree.

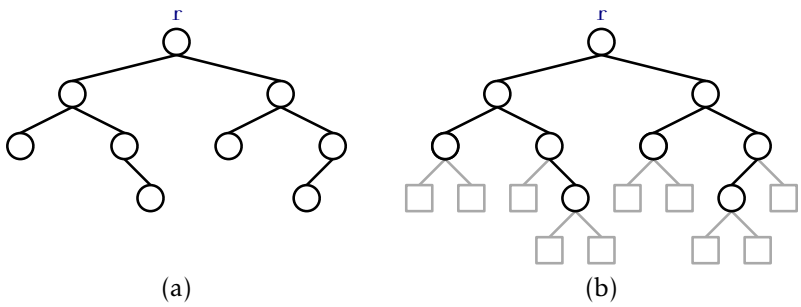


Figura 6.2: Uma árvore binária com (a) nove nós reais e (b) dez nós externos.

tém todos os descendentes de u. A *altura* de um nó, u, é o comprimento do percurso mais longo entre u e um dos seus descendentes. A *altura* de uma árvore é a altura de sua raiz. Um nó, u, é uma *folha* se ele não possui filhos.

Algumas vezes pensamos a árvore como se ela fosse expandida com *nós externos*. Qualquer nó que não possua um filho esquerdo possui um nó externo como seu filho esquerdo e, da mesma maneira, qualquer nó que não tenha um filho direito possui um nó externo como seu filho direito (veja Figura 6.2.b). É fácil verificar, por indução, que uma árvore binária com $n \geq 1$ nós reais possui $n+1$ nós externos.

6.1 BinaryTree: Uma Árvore Binária Básica

Um modo simples de representar um nó, u, em uma árvore binária é armazenar explicitamente, e no máximo, três vizinhos de u:

BinaryTree
```
class BTNode {
  N *left;
  N *right;
  N *parent;
  BTNode() {
    left = right = parent = NULL;
  }
};
```

Quando um dos três vizinhos não está presente, atribuímos a ele o valor `nil`. Deste modo, ambos os nós externos da árvore e o pai da raiz correspondem ao valor `nil`.

A própria árvore binária pode ser representada por uma ponteiro ao seu nó raiz, r:

BinaryTree
```
Node *r;    // root node
```

Podemos calcular a profundidade de um nó, u, em uma árvore binária contando o número de passos no caminho de u até a raiz:

BinaryTree
```
int depth(Node *u) {
  int d = 0;
```

```
  while (u != r) {
    u = u->parent;
    d++;
  }
  return d;
}
```

6.1.1 Algoritmos Recursivos

Usar algoritmos recursivos torna muito fácil o cálculo envolvendo árvores binárias. Por exemplo, para calcular o tamanho (número de nós) de uma árvore binária com raízes no nó u, recursivamente calculamos o tamanho das duas subárvores com raiz nos filhos de u, somamos os tamanhos e adicionamos um:

BinaryTree
```
int size(Node *u) {
  if (u == nil) return 0;
  return 1 + size(u->left) + size(u->right);
}
```

Para calcular a altura de um nó u, podemos calcular a altura das duas subárvores de u, pegar o valor máximo e adicionar 1:

BinaryTree
```
int height(Node *u) {
  if (u == nil) return -1;
  return 1 + max(height(u->left), height(u->right));
}
```

6.1.2 Percurso em Árvores Binárias

Ambos os algoritmos da seção anterior usam a recursão para visitar todos os nós de uma árvore binária. Cada um deles visita os nós da árvore binária na mesma ordem que o seguinte código:

BinaryTree
```
void traverse(Node *u) {
    if (u == nil) return;
```

```
    traverse(u->left);
    traverse(u->right);
}
```

O uso da recursão neste caso produz um código bem sucinto e simples, porém ele pode ser também problemático. A profundidade máxima da recursão é dada pela profundidade máxima do nó na árvore binária, i.e., a altura da árvore. Se a altura da árvore é muito grande, então essa recursão pode muito bem utilizar mais espaço da pilha do que esteja disponível, causando um fechamento do programa.

Para percorrer uma árvore binária sem utilizar a recursão, você pode usar um algoritmo que se baseie de onde ele vem para saber para onde vai. Veja a Figura 6.3. Se chegamos ao nó `u` a partir de `u.pai`, então a próxima coisa a ser feita é visitar `u.esquerdo`. Se chegamos a `u` por `u.esquerdo`, então a próxima coisa a ser feita é visitar `u.direito`. Se chegamos em `u` por `u.direito`, então terminamos de visitar a subárvore de `u`, e retornamos para `u.pai`. O código seguinte implementa esta ideia, com o código incluído para lidar com os casos em que qualquer um de `u.esquerdo`, `u.direito`, ou `u.pai` seja `nil`:

BinaryTree

```
void traverse2() {
  Node *u = r, *prev = nil, *next;
  while (u != nil) {
    if (prev == u->parent) {
      if (u->left != nil) next = u->left;
      else if (u->right != nil) next = u->right;
      else next = u->parent;
    } else if (prev == u->left) {
      if (u->right != nil) next = u->right;
      else next = u->parent;
    } else {
      next = u->parent;
    }
    prev = u;
    u = next;
  }
}
```

Os mesmos resultados que podem ser obtidos usando a recursão tam-

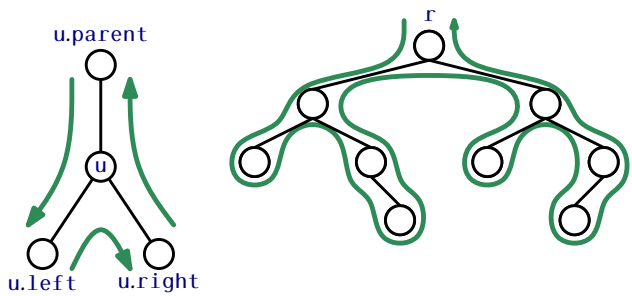


Figura 6.3: Os três casos que ocorrem no nó u quando percorremos uma árvore binária não recursivamente, e o resultado da travessia pela árvore.

bém podem ser obtidos desta maneira, sem recursão. Por exemplo, para calcular o tamanho da árvore mantemos um contador, n, e incrementamos n sempre que visitamos um nó pela primeira vez:

BinaryTree

```
int size2() {
    Node *u = r, *prev = nil, *next;
    int n = 0;
    while (u != nil) {
      if (prev == u->parent) {
        n++;
        if (u->left != nil) next = u->left;
        else if (u->right != nil) next = u->right;
        else next = u->parent;
      } else if (prev == u->left) {
        if (u->right != nil) next = u->right;
        else next = u->parent;
      } else {
        next = u->parent;
      }
      prev = u;
      u = next;
    }
    return n;
  }
```

Em algumas implementações de árvore binárias, o campo pai não é usado. Quando é este o caso, um implementação não recursiva ainda é

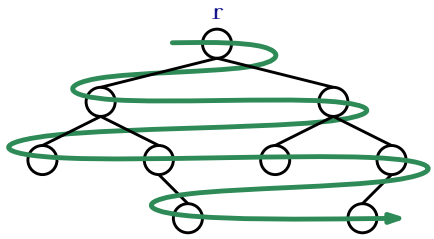


Figura 6.4: Durante um percurso em profundidade, os nós de uma árvore binária são visitados nível por nível e da esquerda para a direita dentro de cada nível.

possível, porém a implementação deve usar uma `Lista` (ou `Pilha`) para acompanhar o caminho do nó atual até a raiz.

Um tipo especial de percurso que não cabe no padrão das funções acima é o *percurso em profundidade*. Em um percurso em profundidade, os nós são visitados nível por nível, começando pela raiz e indo para baixo, visitando os nós de cada nível, da esquerda para a direita (veja Figura 6.4). Isto é similar ao modo pelo qual lemos um texto em português. O percurso em profundidade é implementado usando uma fila, `q`, que inicialmente contém apenas a raiz, `r`. Em cada passo, extraímos o próximo nó, `u`, de `q`, processamos `u` e adicionamos `u.esquerdo` e `u.direito` (se eles não são `nil`) a `q`:

BinaryTree

```
void bfTraverse() {
  ArrayDeque<Node*> q;
  if (r != nil) q.add(q.size(),r);
  while (q.size() > 0) {
    Node *u = q.remove(q.size()-1);
    if (u->left != nil) q.add(q.size(),u->left);
    if (u->right != nil) q.add(q.size(),u->right);
  }
}
```

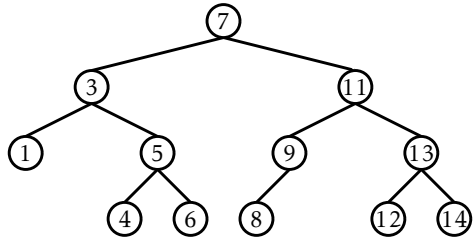


Figura 6.5: Uma árvore binária de busca.

6.2 BinarySearchTree: Uma Árvore Binária de Busca não Balanceada

Uma BinarySearchTree é um tipo especial de árvore binária na qual cada nó, u, também armazena um valor, u.x, de uma ordem total. Os valores em uma árvore binária de busca obedece à *propriedade da árvore binária de busca*: para um nó, u, cada valor armazenado na subárvore com raiz em u.esquerdo é menor que u.x e cada valor armazenado na subárvore com raiz em u.direito é maior que u.x. Um exemplo de uma BinarySearchTree é mostrado na Figura 6.5.

6.2.1 Busca

A propriedade da árvore binária de busca é extremamente útil porque ela permite localizar rapidamente um valor, x, dentro da árvore binária de busca. Para isto, começamos procurando por x na raiz, r. Quando examinamos um nó, u, podem ocorrer três casos:

1. Se x < u.x, então a procura continua em u.esquerdo;
2. Se x > u.x, então a procura continua em u.direito;
3. Se x = u.x, então encontramos o nó u que contém x.

A busca termina quando o Caso 3 ocorre ou quando u = nil. No primeiro

caso, encontramos x. No último caso, concluímos que x não está na árvore binária de busca.

BinarySearchTree

```
T findEQ(T x) {
  Node *w = r;
  while (w != nil) {
    int comp = compare(x, w->x);
    if (comp < 0) {
      w = w->left;
    } else if (comp > 0) {
      w = w->right;
    } else {
      return w->x;
    }
  }
  return null;
}
```

Dois exemplos de busca em uma árvore binária de busca são mostrados na Figura 6.6. Como é mostrado no segundo exemplo, mesmo se não encontramos x na árvore, ainda obtemos alguma informação valiosa. Se olhamos para o último nó, u, no qual o Caso 1 ocorre, percebemos que u.x é o menor valor na árvore que é maior que x. De modo análogo, o último nó no qual o Caso 2 ocorre contém o maior valor na árvore que é menor x. Deste modo, guardando a informação do último nó, z, no qual o Caso 1 ocorre, uma BinarySearchTree pode implementar a operação encontra(x) que retorna o menor valor armazenado que é maior que ou igual a x:

BinarySearchTree

```
T find(T x) {
  Node *w = r, *z = nil;
  while (w != nil) {
    int comp = compare(x, w->x);
    if (comp < 0) {
      z = w;
      w = w->left;
    } else if (comp > 0) {
      w = w->right;
    } else {
      return w->x;
```

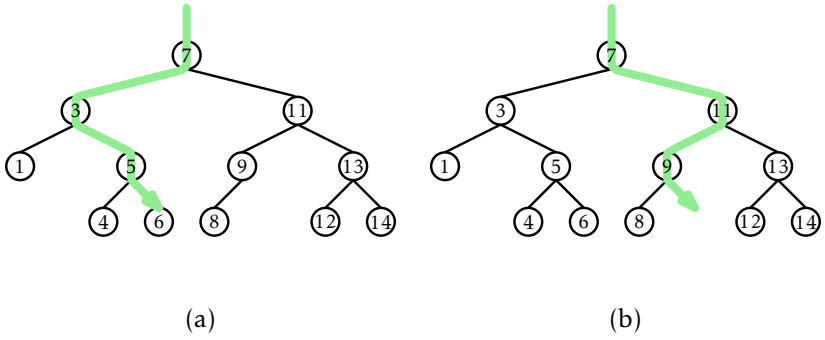


Figura 6.6: Um exemplo de (a) uma busca com sucesso (por 6) e (b) uma busca frustrada (por 10) em uma árvore binária de busca.

```
    }
  }
  return z == nil ? null : z->x;
}
```

6.2.2 Inserção

Para inserir um novo valor, `x`, a uma `BinarySearchTree`, procuramos primeiro por `x`. Se o encontramos, então não precisamos inseri-lo. Caso contrário, armazenamos `x` em um filho do último nó, `p`, encontrado durante a busca por `x`. Se o novo nó é o filho esquerdo ou direito de `p` depende do resultado da comparação de `x` e `p.x`.

BinarySearchTree

```
bool add(T x) {
  Node *p = findLast(x);
  Node *u = new Node;
  u->x = x;
  return addChild(p, u);
}
```

BinarySearchTree

```
Node* findLast(T x) {
  Node *w = r, *prev = nil;
```

```
  while (w != nil) {
    prev = w;
    int comp = compare(x, w->x);
    if (comp < 0) {
      w = w->left;
    } else if (comp > 0) {
      w = w->right;
    } else {
      return w;
    }
  }
  return prev;
}
```

BinarySearchTree

```
bool addChild(Node *p, Node *u) {
    if (p == nil) {
      r = u;              // inserting into empty tree
    } else {
      int comp = compare(u->x, p->x);
      if (comp < 0) {
        p->left = u;
      } else if (comp > 0) {
        p->right = u;
      } else {
        return false;   // u.x is already in the tree
      }
      u->parent = p;
    }
    n++;
    return true;
  }
```

Um exemplo é mostrado na Figura 6.7. A parte que consome mais tempo neste processo é a busca inicial por `x`, que demanda um tempo proporcional à altura do nó recém adicionado `u`. No pior caso, isto é igual à altura da `BinarySearchTree`.

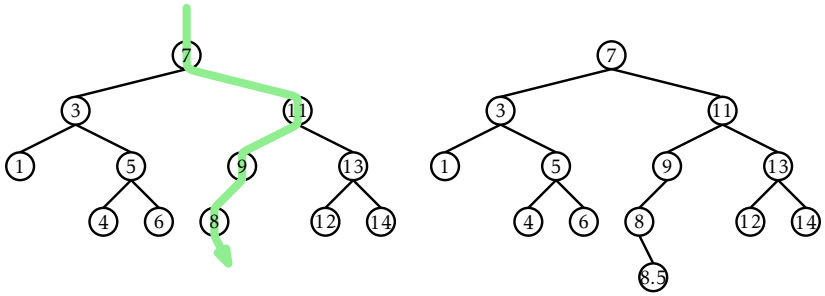


Figura 6.7: Inserindo o valor 8.5 na árvore binária de busca.

6.2.3 Remoção

Apagar um valor armazenado em um nó, `u`, de uma `BinarySearchTree` é um pouco mais difícil. Se `u` é uma folha, então podemos simplesmente desligar `u` do seu pai. Melhor ainda: se `u` possui somente um filho, nós podemos separar `u` da árvore fazendo `u.pai` adotar o filho de `u` (veja Figura 6.8):

BinarySearchTree

```
void splice(Node *u) {
  Node *s, *p;
  if (u->left != nil) {
    s = u->left;
  } else {
    s = u->right;
  }
  if (u == r) {
    r = s;
    p = nil;
  } else {
    p = u->parent;
    if (p->left == u) {
      p->left = s;
    } else {
      p->right = s;
    }
  }
  if (s != nil) {
```

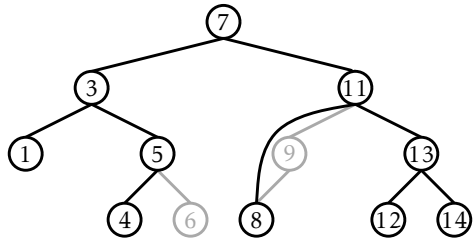


Figura 6.8: Removendo uma folha (6) ou um nó com apenas um filho (9) é fácil.

```
    s->parent = p;
  }
  n--;
}
```

As coisas ficam complicadas, contudo, quando `u` possui dois filhos. Neste caso, a coisa mais simples a fazer é encontrar um nó, `w`, que possua menos que dois filhos de modo que `w.x` possa substituir `u.x`. Para manter a propriedade da árvore binária de busca, o valor `w.x` deveria ser próximo ao valor de `u.x`. Por exemplo, escolher `w` tal que `w.x` seja o menor valor maior que `u.x` irá funcionar perfeitamente. Encontrar o nó `w` é fácil; ele é o menor valor na subárvore com raiz em `u.direito`. Este nó pode ser removido facilmente porque ele não possui filho esquerdo (veja Figura 6.9).

BinarySearchTree

```
void remove(Node *u) {
  if (u->left == nil || u->right == nil) {
    splice(u);
    delete u;
  } else {
    Node *w = u->right;
    while (w->left != nil)
      w = w->left;
    u->x = w->x;
    splice(w);
    delete w;
  }
}
```

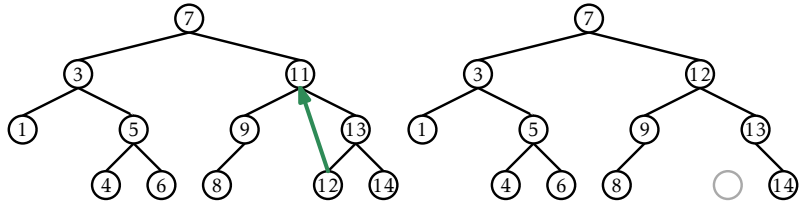


Figura 6.9: Apagar um valor (11) de um nó, u, com dois filhos é realizado substituindo o valor de u pelo menor valor na subárvore direita de u.

6.2.4 Resumo

Cada uma das operações encontrar(x), inserir(x), e remover(x) em uma BinarySearchTree envolve seguir um caminho da raiz da árvore até algum nó árvore. Sem conhecer mais sobre o formato da árvore é difícil dizer muito mais sobre o comprimento deste caminho, exceto que ele é menor que n, o número de nós na árvore. O teorema seguinte (não impressionante) resume o desempenho de uma estrutura de dados BinarySearchTree:

Teorema 6.1. *BinarySearchTree implementa a interface ConjuntoOrdenado e suporta as operações* inserir(x), remover(x), *e* encontrar(x) *em* $O(n)$ *tempos por operação.*

Teorema 6.1 se compara de modo inferior com o Teorema 4.1, que mostra que a estrutura SkiplistConjuntoOrdenado pode implementar a interface ConjuntoOrdenado com tempo esperado de $O(\log n)$ por operação. O problema com a estrutura da BinarySearchTree é que ela pode ficar *desbalanceada*. Em vez de parecer como a árvore na Figura 6.5 ela pode parecer uma longa sequência de n nós, com todos, exceto o último nó possuindo exatamente um único filho.

Existem numerosas formas de evitar uma árvore binária de busca desbalanceada, todos os quais levam a estruturas de dados que têm $O(\log n)$ tempos das operações. No Capítulo 7 mostraremos como tempos de operações *esperados* de $O(\log n)$ podem ser atingidos com a aleatoriedade. No Capítulo 8 mostramos como tempos de operações *amortizados* de $O(\log n)$ podem ser alcançados com operações de reconstruções parciais. No Ca-

pítulo 9 mostramos como tempos de operações de *pior caso* de $O(\log n)$ podem ser alcançados com uma árvore que não seja binária: uma nos quais os nós podem ter até quatro filhos.

6.3 Discussão e Exercícios

Árvores Binárias vêm sendo usadas para modelar relacionamentos por milhares de anos. Uma razão para isso é que árvores binárias modelam naturalmente árvores de famílias (pedigree). Essas são as árvores nas quais a raiz é uma pessoa, os filhos esquerdo e direito são os pais da pessoa, e assim sucessivamente, recursivamente. Nos séculos mais recentes árvore binárias têm também sido usadas para modelar árvores de espécie na biologia, nas quais as folhas da árvore representam espécies existentes e os nós internos representam *eventos de especiação* nos quais duas populações de uma única espécie evoluem em duas espécies separadas.

Árvores Binárias de Busca parecem ter sido descobertas independentemente por vários grupos nos anos 1950 [48, Section 6.2.2]. Referências adicionais para tipos específicos de árvores binárias de busca são fornecidas nos capítulos subsequentes.

Quando implementamos uma árvore binária a partir do zero, várias decisões de projeto devem ser tomadas. Uma delas é a questão se cada nó guarda um ponteiro para seu pai ou não. Se a maioria das operações envolvem simplesmente um caminho seguindo da raiz para uma folha, então o ponteiro para o pai é desnecessário, uma perda de espaço e uma fonte potencial de erros de codificação. Por outro lado, a falta do ponteiro para o pai significa que o percurso da árvore deve ser feito recursivamente ou com o uso de uma pilha explícita. Alguns outros métodos (como inserir ou apagar em alguns tipos de árvores binárias de busca desbalanceadas) são também mais complicados pela ausência do ponteiro para o pai.

Outra decisão de projeto está relacionada em como armazenar o pai, os filhos esquerdo e direito em um nó. Na implementação fornecida aqui, esses ponteiros são armazenados como variáveis separadas. Outra opção é armazená-los em um vetor, `p`, de tamanho 3, de modo que `u.p[0]` é o filho esquerdo de `u`, `u.p[1]` é o filho direito de `u`, e `u.p[2]` é o pai de `u`. O uso do

vetor assim implica que algumas sequências do comando `if` podem ser simplificadas em expressões algébricas.

Um exemplo de tal simplificação ocorre durante o percurso na árvore. Se um percurso chega a um nó u por u.p[i], então o próximo nó neste percurso é u.p[(i + 1) mod 3]. Exemplos similares ocorrem quando existe uma simetria esquerda-direita. Por exemplo, o irmão de u.p[i] é u.p[(i + 1) mod 2]. Este truque funciona melhor se u.p[i] é um filho esquerdo (i = 0) ou um filho direito (i = 1) de u. Em diversos casos isso significa que algum código complicado que de outra maneira precisaria ter ambas versões para os lados esquerdo e direito podem ser escritos apenas uma vez. Veja os métodos `giraEsquerda`(u) and `giraDireita`(u) na página 169 para um exemplo.

Exercício 6.1. Prove que uma árvore binária com n ≥ 1 nós possui n − 1 arestas.

Exercício 6.2. Prove que uma árvore binária com n ≥ 1 nós reais (internos) possui n + 1 nós externos.

Exercício 6.3. Prove que, se uma árvore binária, T, possui ao menos uma folha, então ou (a) a raiz de T possui no máximo um filho ou (b) T possui mais de uma folha.

Exercício 6.4. Implemente um método não recursivo, `tamanho2`(u), que calcule o tamanho da subárvore com raiz no nó u.

Exercício 6.5. Escreva um método não recursivo, `altura2`(u), que calcule a altura do nó u em uma `BinaryTree`.

Exercício 6.6. Uma árvore binária é *balanceada em tamanho* se, para cada nó u, os tamanhos das subárvores com raiz em u.`esquerdo` e u.`direito` diferem de no máximo um. Escreva um método recursivo, `ehBalanceada`(), que testa se uma árvore binária é balanceada. Seu método deve executar num tempo $O(n)$. (Certifique-se de testar seu código em algumas árvores grandes com diferentes formas; é fácil escrever um método que leve muito mais que $O(n)$ em tempo de execução.)

Um percurso em *pré-ordem* de uma árvore binária é um percurso que visita cada nó, u, antes de qualquer de seus filhos. Um percurso *em-ordem*

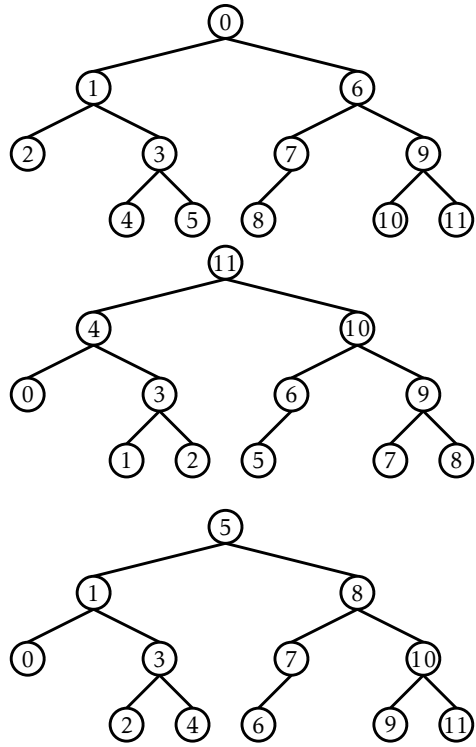


Figura 6.10: numeração em pré-ordem, pós-ordem, e em-ordem de uma árvore binária.

visita u após ter visitado todos os nós na subárvore esquerda de u porém antes de visitar qualquer um dos nós da subárvore direita de u. Um percurso em *pós-ordem* visita u somente após ter visitado todos os outros nós nas subárvores de u. A numeração em pré/em/pós-ordem rotula os nós da árvore com inteiros $0, \ldots, n-1$ na ordem na qual eles são encontrados por um percurso pré/em/pós-ordem. Veja Figura 6.10 para um exemplo.

Exercício 6.7. Crie uma subclasse de `BinaryTree` cujos nós tenham campos para armazenar números de pré-ordem, pós-ordem, e em-ordem. Escreva os métodos recursivos `numeroPreOrdem()`, `numeroEmOrdem()`, e `numeroPosOrdem()` que atribua esses números corretamente. Esses métodos devem executar num tempo $O(n)$.

Exercício 6.8. Implemente as funções não recursivas `proxPreOrdem(u)`, `proxEmOrdem(u)`, e `proxPosOrdem(u)` que retornam o nó seguinte a `u` em um percurso em pré-ordem, em-ordem, ou pós-ordem, respectivamente. Essas funções devem ter um tempo de execução amortizado constante; se começamos em qualquer nó `u` e repetidamente chamarmos uma dessas funções e atribuirmos o valor retornado a `u` até que `u = null`, então o custo de todas essas chamadas deveria ser de $O(n)$.

Exercício 6.9. Suponha que temos uma árvore binária com números de pré-, pós-, e em-ordem atribuídos aos nós. Mostre como esses números podem ser usados para responder cada uma das seguintes questões em tempo constante:

1. Dado um nó `u`, determine o tamanho da subárvore com raiz em `u`.
2. Dado um nó `u`, determine a profundidade de `u`.
3. Dados dois nós `u` e `w`, determine se `u` é um ancestral de `w`

Exercício 6.10. Suponha que você receba uma lista de nós com números de pré-ordem e em-ordem atribuídos a eles. Prove que existe no máximo uma possível árvore com esses números de pré-ordem/em-ordem e mostre como construi-la.

Exercício 6.11. Mostre que o formato de qualquer árvore binária com n nós pode ser representada usando no máximo $2(n-1)$ bits. (Dica: pense sobre gravar o que acontece durante o percurso e então recuperar este registro para reconstruir a árvore.)

Exercício 6.12. Ilustre o que acontece quando adicionamos o valor 3.5 e depois 4.5 na árvore binária de busca na Figura 6.5.

Exercício 6.13. Ilustre o que acontece quando removemos o valor 3 e depois 5 da árvore binária de busca na Figura 6.5.

Exercício 6.14. Implemente um método **obtemLE(x)**, para a `BinarySearchTree`, que retorne uma lista de todos os itens em uma árvore que sejam menores que ou iguais a `x`. O tempo de execução do seu método deve ser $O(n' + h)$ no qual n' é o número de itens menores que ou iguais a `x` e `h` é a altura da árvore.

Exercício 6.15. Descreva como inserir os elementos $\{1,\ldots,n\}$ para uma `BinarySearchTree` inicialmente vazia de modo que a árvore resultante tenha altura $n-1$. De quantos modos podemos fazer isso?

Exercício 6.16. Se temos uma `BinarySearchTree` e executamos a operação `inserir(x)` seguida por `remover(x)` (com o mesmo valor de `x`) necessariamente retornamos à árvore original?

Exercício 6.17. Uma operação `remover(x)` pode aumentar a altura de algum nó em uma `BinarySearchTree`? Se sim, de quanto?

Exercício 6.18. Uma operação `inserir(x)` pode aumentar a altura de algum nó em uma `BinarySearchTree`? Se sim, de quanto?

Exercício 6.19. Projete e implemente uma versão da `BinarySearchTree` na qual cada nó, `u`, mantém os valores `u.tamanho` (o tamanho da subárvore com raiz em `u`), `u.profundidade` (a profundidade de `u`), e `u.altura` (a altura da subárvore com raiz em `u`).

Estes valores devem ser mantidos mesmo durante chamadas a operações de `inserir(x)` e `remover(x)`, porém isto não deve aumentar o custo dessas operações de mais de um valor constante.

Capítulo 7

Árvores Binárias Aleatórias de Busca

Neste capítulo, apresentamos uma estrutura de árvore binária de busca que usa a aleatoriedade para conseguir $O(\log n)$ de tempo esperado para todas as operações.

7.1 Árvores Binárias Aleatórias de Busca

Considere as duas árvores binárias de busca mostradas na Figura 7.1, cada uma das quais possui $n = 15$ nós. A árvore da esquerda é uma lista e a outra é uma árvore binária de busca perfeitamente balanceada. A árvore da esquerda possui uma altura de $n - 1 = 14$ e aquela à direita possui uma altura de três.

Imagine como essas duas árvores poderiam ser construídas. A árvore da esquerda ocorre se começamos com uma `ArvoreBinariaDeBusca` vazia e acrescentamos a sequência

$$\langle 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 \rangle \ .$$

Nenhuma outra sequência de adições vai criar esta árvore (como você pode provar por indução em n). Por outro lado, a árvore da direita pode ser criada pela sequência

$$\langle 7, 3, 11, 1, 5, 9, 13, 0, 2, 4, 6, 8, 10, 12, 14 \rangle \ .$$

Outras sequências funcionam bem, incluindo

$$\langle 7, 3, 1, 5, 0, 2, 4, 6, 11, 9, 13, 8, 10, 12, 14 \rangle \ ,$$

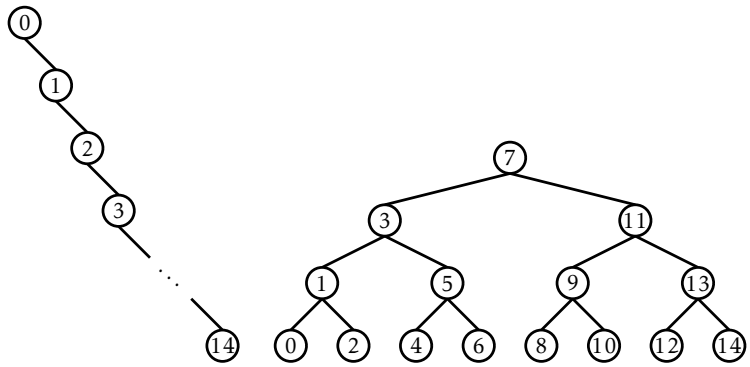


Figura 7.1: Duas árvores binárias de busca com inteiros $0,\ldots,14$.

e

$$\langle 7,3,1,11,5,0,2,4,6,9,13,8,10,12,14\rangle \ .$$

De fato, existem $21,964,800$ sequências que geram a árvore da direita e somente uma que gera a árvore da esquerda.

O exemplo acima nos dá uma evidência factual que, se escolhemos uma permutação aleatória de $0,\ldots,14$, e inserimos em uma árvore binária, é mais provável obtermos uma árvore balanceada (a árvore da direita de Figura 7.1) que obtermos uma árvore totalmente desbalanceada (aquela do lado esquerdo de Figura 7.1).

Podemos formalizar esta noção estudando as árvores binárias aleatórias de busca. Uma *árvore binária aleatória de busca* de tamanho n é obtida do seguinte modo: Considere uma permutação aleatória, $x_0,\ldots,x_{n-1}$, de inteiros $0,\ldots,n-1$ e insira seus elementos, um a um, em uma `ArvoreBinariaDeBusca`. Por *permutação aleatória* queremos dizer que cada possível $n!$ permutação (ordenação) de $0,\ldots,n-1$ é igualmente provável, assim como que a probabilidade de obter qualquer permutação particular é $1/n!$.

Note que os valores $0,\ldots,n-1$ poderiam ser substituídos por qualquer conjunto ordenado de n elementos sem mudar qualquer uma das propriedades da árvore binária aleatória de busca. O elemento $x \in \{0,\ldots,n-1\}$ significa apenas o elemento de ordem x de um conjunto ordenado de tamanho n.

Antes de apresentarmos nosso resultado principal sobre árvores biná-

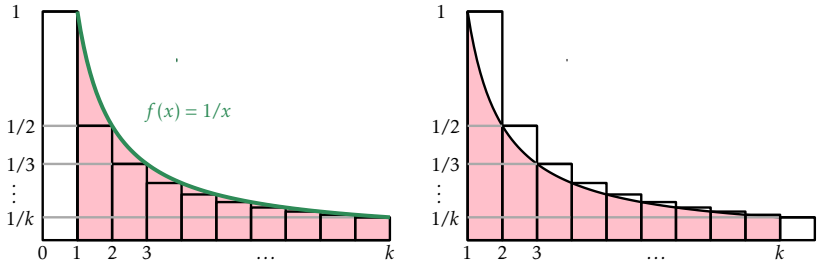


Figura 7.2: O $k$ésimo número harmônico $H_k = \sum_{i=1}^{k} 1/i$ tem limite superior e inferior dados por duas integrais. O valor dessas integrais é dado pela área da região sombreada, enquanto o valor de H_k é dado pela área dos retângulos.

rias aleatórias de busca, devemos tomar um tempo para uma curta digressão para discutir um tipo de número que aparece frequentemente quando estudamos estruturas aleatórias. Para um inteiro não negativo, k, o k-ésimo *número harmônico*, identificado por H_k, é definido como

$$H_k = 1 + 1/2 + 1/3 + \cdots + 1/k \ .$$

O número harmônico H_k não tem uma forma analítica simples, porém ele é ligado de forma muito estreita ao logaritmo natural de k. Particularmente,

$$\ln k < H_k \leq \ln k + 1 \ .$$

Leitores que estudaram cálculo podem notar que isto ocorre porque a integral $\int_1^k (1/x)\,\mathrm{d}x = \ln k$. Percebendo que uma integral pode ser interpretada como a área entre uma curva e o eixo x, o valor de H_k pode ter como limite inferior a integral $\int_1^k (1/x)\,\mathrm{d}x$ e como limite superior $1 + \int_1^k (1/x)\,\mathrm{d}x$. (Veja Figura 7.2 para uma explicação gráfica.)

Lema 7.1. *Em uma árvore binária aleatória de busca de tamanho* n*, as seguintes declarações são verdadeiras:*

1. *Para qualquer* $\mathsf{x} \in \{0, \ldots, \mathsf{n}-1\}$*, o comprimento esperado do caminho de busca para* x *é* $H_{\mathsf{x}+1} + H_{\mathsf{n}-\mathsf{x}} - O(1)$.[1]

[1] As expressões $\mathsf{x}+1$ e $\mathsf{n}-\mathsf{x}$ podem ser interpretadas respectivamente como o número de elementos na árvore menores que ou iguais a x e o número de elementos na árvore maiores que ou iguais a x.

2. *Para qualquer* $x \in (-1, n) \setminus \{0, \ldots, n-1\}$, *o comprimento esperado do caminho de busca para* x *é* $H_{\lceil x \rceil} + H_{n-\lceil x \rceil}$.

Vamos provar Lema 7.1 na próxima seção. Por agora, considere o que as duas partes de Lema 7.1 nos dizem. A primeira parte nos diz que se procuramos por um elemento em uma árvore de tamanho n, então o comprimento esperado do caminho de busca é no máximo $2\ln n + O(1)$. A segunda parte nos diz a mesma coisa sobre a busca de um valor que não esteja armazenado na árvore. Quando comparamos a duas partes do lema, vemos que é somente ligeiramente mais rápido procurar por algo que esteja na árvore do que por algo que não esteja.

7.1.1 Prova de Lema 7.1

A observação chave necessária para provar o Lema 7.1 é a seguinte: O caminho de busca para um valor x no intervalo aberto $(-1, n)$ em uma árvore binária aleatória de busca, T, contém o nó com a chave $i < x$ se, e somente se, na permutação aleatória usada para criar T, i apareça antes de qualquer um de $\{i+1, i+2, \ldots, \lfloor x \rfloor\}$.

Para ver isso, olhe a Figura 7.3 e note que até que algum valor de $\{i, i+1, \ldots, \lfloor x \rfloor\}$ seja inserido, os caminhos de busca para cada valor no intervalo aberto $(i-1, \lfloor x \rfloor + 1)$ são idênticos. (Lembre-se que para dois valores terem caminhos de busca diferentes, deve existir algum elemento na árvore que seja comparado diferentemente entre eles.) Seja j o primeiro elemento em $\{i, i+1, \ldots, \lfloor x \rfloor\}$ a aparecer na permutação aleatória. Note que j está neste momento e sempre estará no caminho de busca de x. Se $j \neq i$ então o nó u_j contendo j é criado antes do nó u_i que contém i. Mais tarde, quando i for inserido, ele será inserido na subárvore com raiz em u_j.`esquerdo`, posto que $i < j$. Por outro lado, o caminho de busca para x nunca passará por esta subárvore porque ele seguirá para u_j.`direito` após visitar u_j.

De maneira análoga, para $i > x$, i aparece no caminho de busca para x se e somente se i aparece antes de qualquer um de $\{\lceil x \rceil, \lceil x \rceil + 1, \ldots, i-1\}$ em uma permutação aleatória usada para criar T.

Note que, se começamos com uma permutação aleatória de $\{0, \ldots, n\}$, então as subsequências contendo somente $\{i, i+1, \ldots, \lfloor x \rfloor\}$ e $\{\lceil x \rceil, \lceil x \rceil + 1, \ldots, i-1\}$ são também permutações aleatórias de seus respectivos elementos. Cada

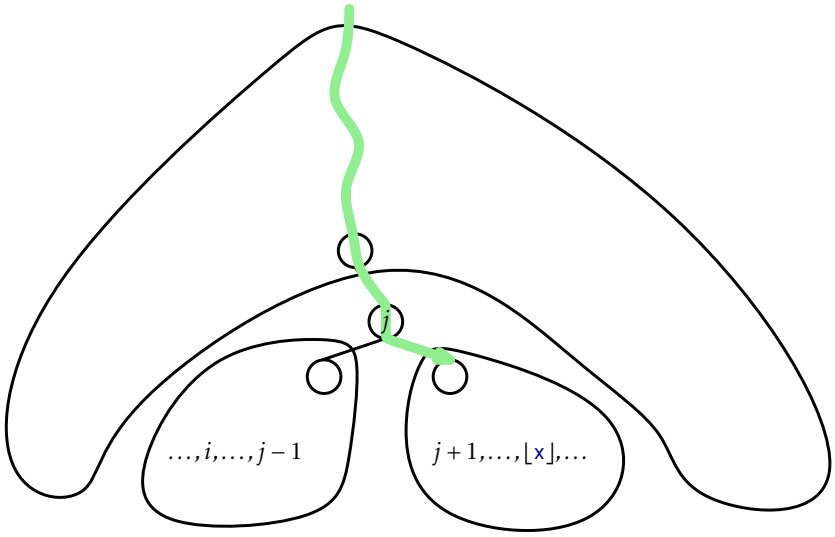


Figura 7.3: O valor $i < x$ está no caminho de busca para x se e somente se i é o primeiro elemento entre $\{i, i+1, \ldots, \lfloor x \rfloor\}$ inserido na árvore.

elemento, então, nos subconjuntos $\{i, i+1, \ldots, \lfloor x \rfloor\}$ e $\{\lceil x \rceil, \lceil x \rceil + 1, \ldots, i-1\}$ é igualmente provável de aparecer antes de qualquer outro no seu subconjunto na permutação aleatória usada para criar T. Assim, temos

$$\Pr\{i \text{ está no caminho de busca para } x\} = \begin{cases} 1/(\lfloor x \rfloor - i + 1) & \text{if } i < x \\ 1/(i - \lceil x \rceil + 1) & \text{if } i > x \end{cases} .$$

Com esta observação, a prova de Lema 7.1 envolve alguns cálculos simples com números harmônicos:

Prova de Lema 7.1. Seja I_i a variável aleatória indicadora que é igual a um quando i aparece no caminho de busca para x e zero caso contrário. Então o comprimento do caminho de busca é dado por

$$\sum_{i \in \{0,\ldots,n-1\} \setminus \{x\}} I_i$$

deste modo, se $x \in \{0, \ldots, n-1\}$, o comprimento esperado do caminho de

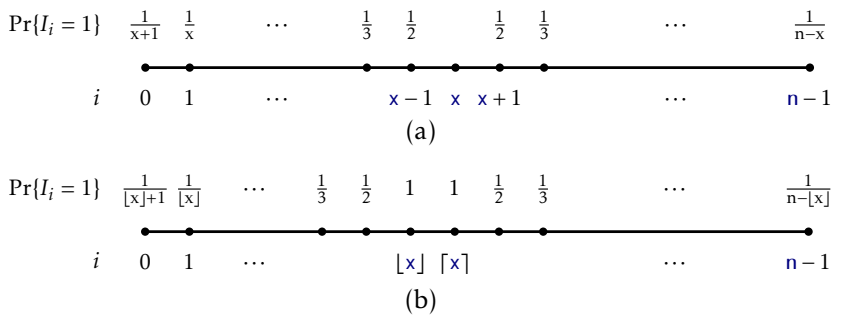


Figura 7.4: As probabilidades de um elemento estar no caminho de busca para x quando (a) x é um inteiro e (b) quando x não é um inteiro.

busca é dado por (veja Figura 7.4.a)

$$\begin{aligned}
\mathrm{E}\left[\sum_{i=0}^{x-1} I_i + \sum_{i=x+1}^{n-1} I_i\right] &= \sum_{i=0}^{x-1} \mathrm{E}[I_i] + \sum_{i=x+1}^{n-1} \mathrm{E}[I_i] \\
&= \sum_{i=0}^{x-1} 1/(\lfloor x \rfloor - i + 1) + \sum_{i=x+1}^{n-1} 1/(i - \lceil x \rceil + 1) \\
&= \sum_{i=0}^{x-1} 1/(x - i + 1) + \sum_{i=x+1}^{n-1} 1/(i - x + 1) \\
&= \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{x+1} \\
&\quad + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-x} \\
&= H_{x+1} + H_{n-x} - 2 \ .
\end{aligned}$$

Os cálculos correspondentes para o valor de busca $x \in (-1, n) \setminus \{0, \ldots, n-1\}$ são praticamente idênticos (veja Figura 7.4.b). □

7.1.2 Resumo

O teorema seguinte resume o desempenho de uma árvore binária aleatória de busca:

Teorema 7.1. *Uma árvore binária aleatória de busca pode ser construída em um tempo* $O(n \log n)$. *Em uma árvore binária aleatória de busca, a operação* `find(x)` *tem um tempo esperado de* $O(\log n)$.

Devemos enfatizar novamente que a expectativa no Teorema 7.1 é relativa a uma permutação aleatória usada para criar a árvore binária aleatória de busca. Em particular, ela não depende de uma escolha aleatória de `x`; ela é verdadeira para cada valor de `x`.

7.2 Treap: Uma Árvore Binária de Busca Aleatorizada

O problema com as árvores binárias aleatórias de buscas é, é claro, que elas não são dinâmicas. Elas não suportam as operações `add(x)` ou `remove(x)` necessárias para implementar a interface `ConjuntoOrdenado`. Nesta seção, descrevemos uma estrutura de dados chamada `Treap` que usa o Lema 7.1 para implementar a interface `SSet`.[2]

Um nó em uma `Treap` é como um nó em uma `ArvoreBinariaDeBusca` no sentido que ele tem um valor para o dado, `x`, porém ele também tem uma *prioridade*, numérica única, `p`, que é associada aleatoriamente:

Treap
```
class TreapNode : public BSTNode<Node, T> {
  friend class Treap<Node,T>;
  int p;
};
```

Adicionalmente, para ser um árvore binária de busca, os nós de uma `Treap` também obedecem à *propriedade de heap*:

- (Propriedade de Heap) Para cada nó `u`, exceto a raiz, `u.pai.p` < `u.p`.

em outras palavras, cada nó possui uma prioridade menor que aquelas de seus dois filhos. Um exemplo é mostrado na Figura 7.5.

As condições de heap e de árvore binária de busca juntas garantem que, uma vez que a chave (`x`) e a prioridade (`p`) de cada nó seja definido, o formato da `Treap` está completamente determinado. A propriedade de

[2] O nome `Treap` vem do fato que esta estrutura é, simultaneamente, uma árvore binária de busca (**tr**ee, em inglês) (Seção 6.2) e um h**eap** (Capítulo 10).

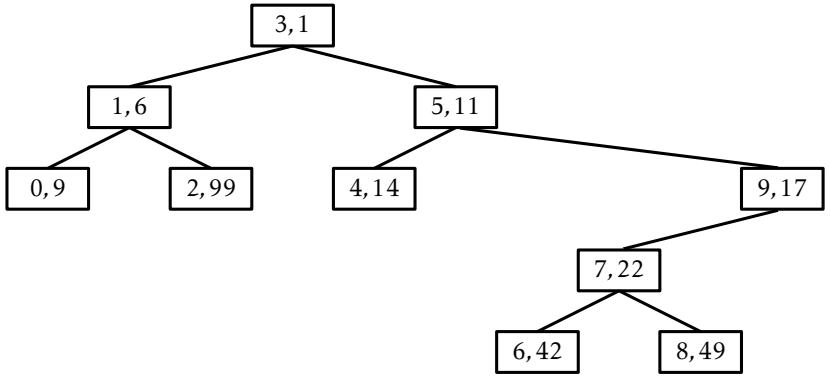


Figura 7.5: Um exemplo de uma Treap contendo os inteiros $0, \ldots, 9$. Cada nó, u, é ilustrado como uma caixa contendo u.x, u.p.

heap nos diz que o nó com prioridade mínima tem que ser a raiz, r, da Treap. A propriedade da árvore binária de busca nos diz que todos os nós com chaves menores que r.x são armazenados na subárvore com raiz em r.esquerdo e todos os nós com chaves maiores que r.x são armazenados na subárvore com raiz em r.direito.

Um ponto importante sobre os valores de prioridade em uma Treap é que eles são únicos e atribuídos aleatoriamente. Por conta disso, existem dois modos equivalentes de pensar sobre uma Treap. Como definido acima, uma Treap obedece às propriedades do heap e da árvore binária de busca. Alternativamente, podemos pensar uma Treap como uma ArvoreBinariaDeBusca cujos nós sejam inseridos em uma ordem crescente de prioridade. Por exemplo, A Treap na Figura 7.5 pode ser obtida com a inserção da sequência de valores (x,p)

$$\langle (3,1),(1,6),(0,9),(5,11),(4,14),(9,17),(7,22),(6,42),(8,49),(2,99)\rangle$$

em uma ArvoreBinariaDeBusca.

Como as prioridades são escolhidas aleatoriamente, isto é equivalente a pegar uma permutação aleatória das chaves—neste caso a permutação é

$$\langle 3,1,0,5,9,4,7,6,8,2\rangle$$

—e inserindo essas em uma ArvoreBinariaDeBusca. Porém isso significa

que a forma de uma treap é idêntica àquela de uma árvore binária aleatória de busca. Particularmente, se substituirmos cada chave x por sua posição,[3] então o Lema 7.1 é válido. Redeclarando o Lema 7.1 em termos de uma Treaps, temos:

Lema 7.2. *Em uma Treap que armazena um conjunto S de n chaves, as seguintes declarações são verdadeiras:*

1. *Para qualquer $x \in S$, o tamanho esperado do caminho de busca para x é $H_{r(x)+1} + H_{n-r(x)} - O(1)$.*
2. *Para qualquer $x \notin S$, o tamanho esperado do caminho de busca para x é $H_{r(x)} + H_{n-r(x)}$.*

Aqui, $r(x)$ indica a posição x no conjunto $S \cup \{x\}$.

Novamente, enfatizamos que a expectativa no Lema 7.2 é tirada sobre as escolhas aleatórias das prioridades de cada nó. Ela não requer qualquer pressuposto sobre a aleatoriedade nas chaves.

O Lema 7.2 nos diz que Treaps pode implementar a operação `find(x)` eficientemente. Contudo, o benefício real de uma Treap é que ela pode suportar as operações `add(x)` e `delete(x)`. Para fazer isto, ela precisa executar rotações de modo a manter a propriedade de heap. Veja a Figura 7.6. Uma *rotação* em uma árvore binária de busca é uma modificação local que pega um pai u de um nó w e torna w o pai de u, enquanto preserva a propriedade da árvore binária de busca. Rotações vêm com dois sabores: *à esquerda* or *à direita* dependendo se w é um filho direito ou esquerdo de u, respectivamente.

O código que implementa isto deve prever essas duas possibilidades e tomar cuidado com um caso limite (quando u é a raiz), deste modo, o código real é um pouco mais longo que a Figura 7.6 poderia levar um leitor a crer:

BinarySearchTree

```
void rotateLeft(Node *u) {
  Node *w = u->right;
  w->parent = u->parent;
  if (w->parent != nil) {
```

[3] A posição de um elemento x em um conjunto de elementos S é o número de elementos em S que são menores que x.

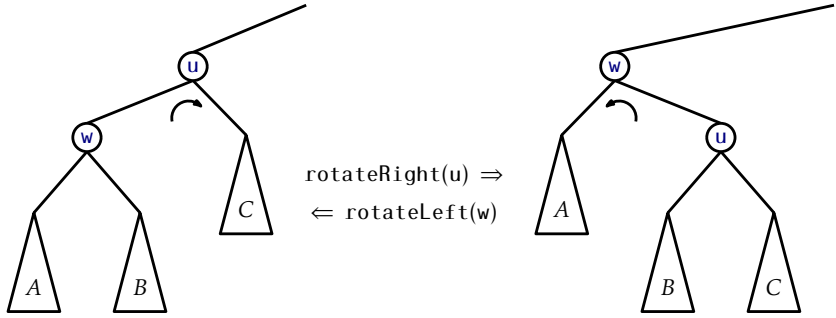


Figura 7.6: Rotações à esquerda e à direita em uma árvore binária de busca.

```
    if (w->parent->left == u) {
      w->parent->left = w;
    } else {
      w->parent->right = w;
    }
  }
  u->right = w->left;
  if (u->right != nil) {
    u->right->parent = u;
  }
  u->parent = w;
  w->left = u;
  if (u == r) { r = w; r->parent = nil; }
}
void rotateRight(Node *u) {
  Node *w = u->left;
  w->parent = u->parent;
  if (w->parent != nil) {
    if (w->parent->left == u) {
      w->parent->left = w;
    } else {
      w->parent->right = w;
    }
  }
  u->left = w->right;
  if (u->left != nil) {
    u->left->parent = u;
```

```
  }
  u->parent = w;
  w->right = u;
  if (u == r) { r = w; r->parent = nil; }
}
```

Em termos da estrutura de dados Treap, a propriedade mais importante de uma rotação é que a profundidade de w diminui de um enquanto a profundidade de u aumenta de um.

Usando rotações, podemos implementar a operação add(x) como se segue: Criamos um novo nó, u, atribuímos $u.x = x$, e escolhemos um valor aleatório para $u.p$. Em seguida, inserimos u usando o algoritmo usual add(x) para uma ArvoreBinariaDeBusca, assim, u agora é uma nova folha de Treap. Neste ponto, nossa Treap satisfaz a propriedade da árvore binária de busca, mas não necessariamente a propriedade de heap. Particularmente, pode ser o caso em que $u.pai.p > u.p$. Se este é o caso, então executamos uma rotação no nó w=$u.pai$ de modo que u se torne o pai de w. Se u continua a violar a propriedade de heap, teremos que repetir isso, diminuindo a profundidade de u por um a cada vez, até que u se torne a raiz ou $u.pai.p < u.p$.

Treap

```
bool add(T x) {
  Node *u = new Node;
  u->x = x;
  u->p = rand();
  if (BinarySearchTree<Node,T>::add(u)) {
    bubbleUp(u);
    return true;
  }
  delete u;
  return false;
}
void bubbleUp(Node *u) {
  while (u->parent != nil && u->parent->p > u->p) {
    if (u->parent->right == u) {
      rotateLeft(u->parent);
    } else {
      rotateRight(u->parent);
    }
```

```
  }
  if (u->parent == nil) {
    r = u;
  }
}
```

Um exemplo de uma operação add(x) é mostrada na Figura 7.7.

O tempo de execução de uma operação add(x) é dado pelo tempo que leva para seguir o caminho de busca para x mais o número de rotações executadas para mover o nó recém adicionado, u, na sua localização correta na Treap. Pelo Lema 7.2, o comprimento esperado do caminho de busca é no máximo $2 \ln n + O(1)$. Além disso, cada rotação diminui a profundidade de u. Esse processo cessa se u se torna a raiz, assim o número esperado de rotações não pode ultrapassar o comprimento esperado do caminho de busca. Então, o tempo esperado de execução da operação add(x) em uma Treap é $O(\log n)$. (O Exercício 7.5 pede para demonstrar que o número esperado de rotações executadas durante uma inserção é, de fato, somente $O(1)$.)

A operação remove(x) em uma Treap é o oposto da operação add(x). Procuramos pelo nó, u, contendo x, então executamos rotações para mover u para baixo até que ele se torne uma folha, e então separamos u da Treap. Note que, para mover u para baixo, podemos executar ou uma rotação para esquerda ou uma pra direita em u, que vai substituir u por u.direito ou u.esquerdo, respectivamente. A escolha é feita de acordo com a primeira situação seguinte que aparece:

1. Se u.esquerdo e u.direito são ambos null, então u é uma folha e nenhuma rotação é feita.

2. Se u.esquerdo (ou u.direito) é null, então executamos uma rotação à direita (ou à esquerda, respectivamente) em u.

3. Se u.esquerdo.p < u.direito.p (ou u.esquerdo.p > u.direito.p), então executamos uma rotação à direita (ou rotação à esquerda, respectivamente) em u.

Essas três regras asseguram que a Treap não se torne desconectada e que a propriedade de heap seja restabelecida uma vez que u seja removido.

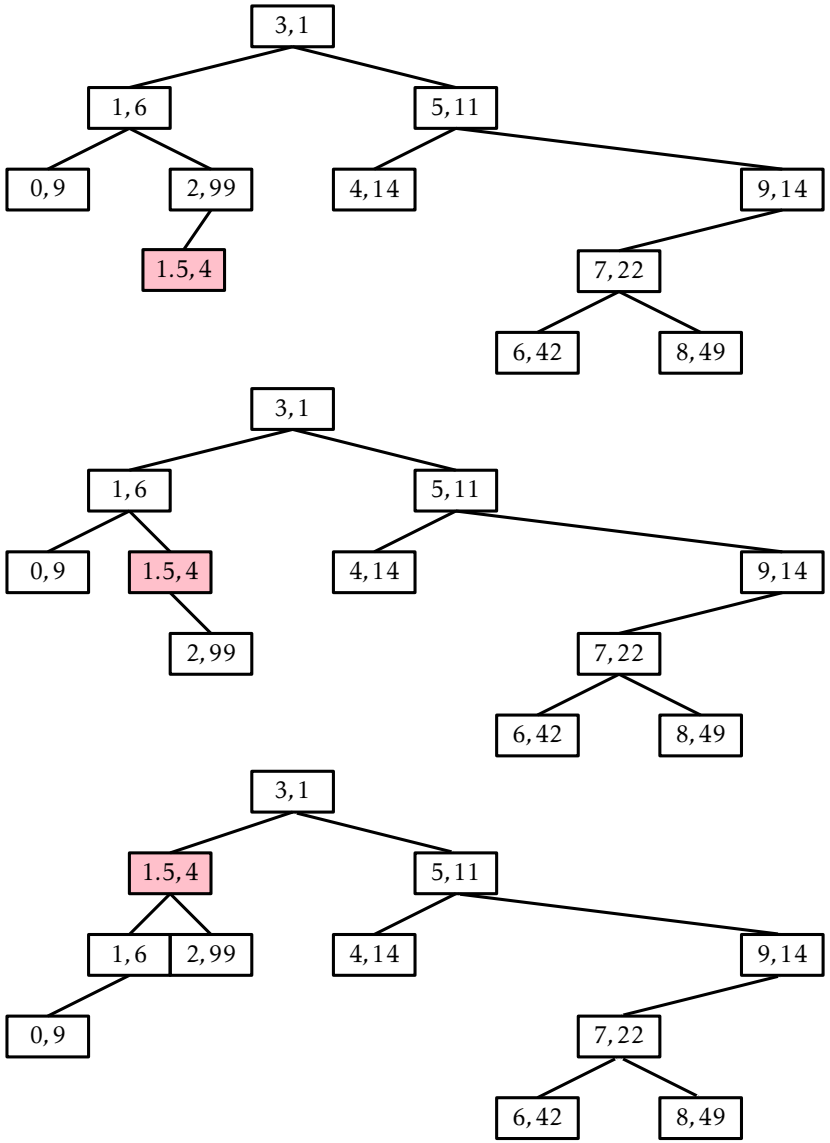


Figura 7.7: Inserindo o valor 1.5 na Treap da Figura 7.5.

Treap

```
bool remove(T x) {
  Node *u = findLast(x);
  if (u != nil && compare(u->x, x) == 0) {
    trickleDown(u);
    splice(u);
    delete u;
    return true;
  }
  return false;
}
void trickleDown(Node *u) {
  while (u->left != nil || u->right != nil) {
    if (u->left == nil) {
      rotateLeft(u);
    } else if (u->right == nil) {
      rotateRight(u);
    } else if (u->left->p < u->right->p) {
      rotateRight(u);
    } else {
      rotateLeft(u);
    }
    if (r == u) {
      r = u->parent;
    }
  }
}
```

Um exemplo da operação remove(x) é mostrado na Figura 7.8.

O truque para analisar o tempo de execução da operação remove(x) é notar que a operação inverte a operação add(x). Particularmente, se reinserimos x, usando a mesma prioridade u.p, então a operação add(x) faria exatamente o mesmo número de rotações e iria restabelecer a Treap para exatamente o mesmo estado em que estava antes da operação remove(x) ter sido executada. (Lendo de baixo para cima, a Figura 7.8 ilustra a inserção do valor 9 em uma Treap.) Isto significa que o tempo de execução esperado de remove(x) em uma Treap de tamanho n é proporcional ao tempo esperado de execução da operação add(x) em uma Treap de tamanho $n-1$. Concluímos que o tempo esperado de execução de remove(x) é

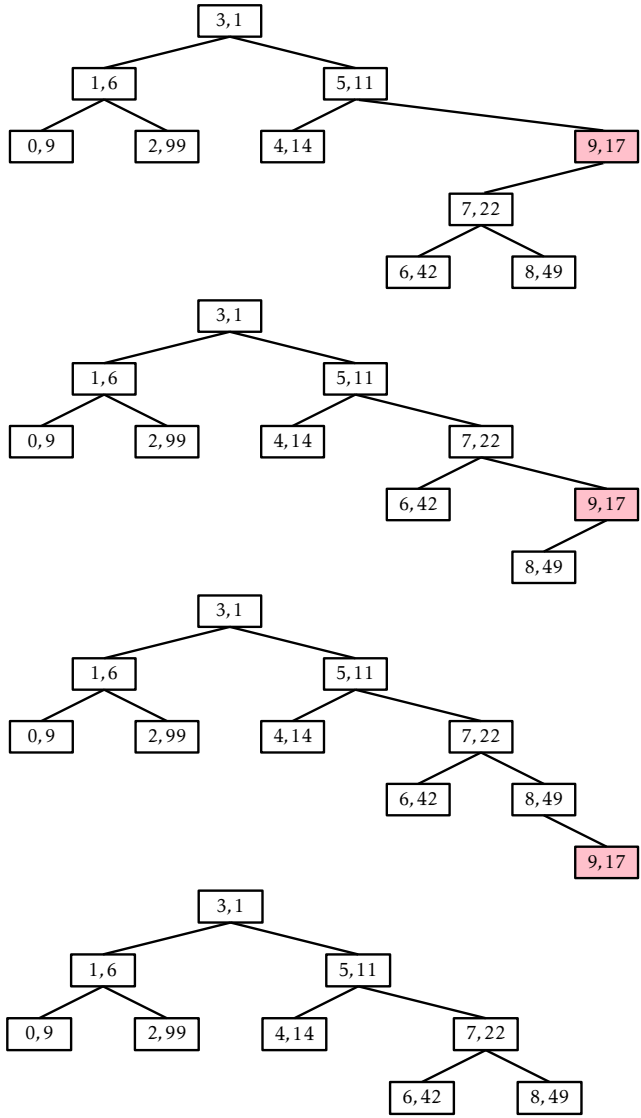


Figura 7.8: Removendo o valor 9 da Treap na Figura 7.5.

$O(\log n)$.

7.2.1 Resumo

O teorema seguinte resume o desempenho de uma estrutura de dados `Treap`:

Teorema 7.2. *Uma `Treap` implementa a interface `SSet`. Uma `Treap` suporta as operações* `add(x)`, `remove(x)`, *e* `find(x)` *em um tempo esperado de* $O(\log n)$ *por operação.*

Vale a pena comparar a estrutura de dados `Treap` a uma estrutura de dados `SkiplistSSet`. Ambas implementam as operações `SSet` em um tempo esperado de $O(\log n)$ por operação. Em ambas estruturas de dados, `add(x)` e `remove(x)` envolvem uma busca e então um número constante de mudanças em ponteiros (veja Exercício 7.5 abaixo). Assim, para ambas as estruturas, o comprimento esperado do caminho de busca é o valor crítico para valiar seus desempenhos. Em uma `SkiplistSSet`, o comprimento esperado de um caminho de busca é

$$2\log n + O(1) \ ,$$

Em uma `Treap`, o comprimento esperado de um caminho de busca é

$$2\ln n + O(1) \approx 1.386\log n + O(1) \ .$$

Assim, os caminhos de busca em uma `Treap` são consideravelmente menores e isto se traduz em operações notadamente mais rápidas em uma `Treaps` que em uma `Skiplists`. O Exercício 4.7 no Capítulo 4 mostra como o comprimento esperado de um caminho de busca em uma `Skiplist` pode ser reduzido para

$$e\ln n + O(1) \approx 1.884\log n + O(1)$$

usando o lançamento de uma moeda viciada. Mesmo com esta otimização, o comprimento esperado dos caminhos de busca em uma `SkiplistSSet` é notadamente mais longo que em uma `Treap`.

7.3 Discussão e Exercícios

Árvores binárias de buscas aleatórias foram estudadas extensivamente. Devroye [19] fornece uma prova do Lema 7.1 e resultados relacionados. Existem resultados muito mais fortes na literatura, o mais impressionante dos quais é devido a Reed [62], que mostra que a altura esperada de uma árvore binária aleatória de busca é

$$\alpha \ln n - \beta \ln \ln n + O(1)$$

onde $\alpha \approx 4.31107$ é o valor da solução única no intervalo $[2, \infty)$ da equação $\alpha \ln((2e/\alpha)) = 1$ e $\beta = \frac{3}{2\ln(\alpha/2)}$. Além disso, a variância da altura é constante.

O nome Treap foi criado por Seidel e Aragon [65] que discutiram Treaps e algumas de suas variantes. Contudo, a estrutura básica foi estudada anteriormente por Vuillemin [74] que as chamou de árvores Cartesianas.

Uma possível otimização de espaço de uma estrutura de dados Treap é a eliminação do armazenamento explícito da prioridade p em cada nó. Em vez disso, a prioridade do nó, u, é calculada pelo endereço de hash de u na memória . Embora um bom número de funções de hash provavelmente funcionem bem para esta prática, para que as partes importantes da prova do Lema 7.1 permaneçam válidas, a função de hash deve ser aleatória e ter a *propriedade independente min-wise*: Para qualquer valor distinto $x_1, \ldots, x_k$, cada um dos valores de hash $h(x_1), \ldots, h(x_k)$ deve ser distinto com alta probabilidade e, para cada $i \in \{1, \ldots, k\}$,

$$\Pr\{h(x_i) = \min\{h(x_1), \ldots, h(x_k)\}\} \leq c/k$$

para alguma constante c. Um desses tipos de funções de hash que é fácil de implementar e razoavelmente rápida é a *hashing por tabulação* (Seção 5.2.3).

Outra variante de Treap que não armazena prioridades em cada nó é a árvore binária de busca aleatorizada de Martínez and Roura [51]. Nesta variante, cada nó, u, armazena o tamanho, u.size, da subárvore com raiz em u. Ambos os algoritmos add(x) e remove(x) são aleatorizados. O algoritmo para inserir x à subárvore com raiz em u faz o seguinte:

1. Com probabilidade $1/(\texttt{size}(\texttt{u})+1)$, o valor `x` é inserido da maneira usual, como uma folha, e são feitas rotações para levar `x` até a raiz de sua subárvore.

2. Caso contrário (com probabilidade $1-1/(\texttt{size}(\texttt{u})+1)$), o valor `x` é inserido recursivamente em uma das duas subárvores com raiz em `u.esquerdo` ou `u.direito`, da maneira apropriada.

O primeiro caso corresponde a uma operação add(`x`) em uma `Treap` onde o nó `x` recebe um prioridade aleatória que é menor que qualquer das `size(u)` prioridades na subárvore de `u`, e este caso ocorre com exatamente a mesma probabilidade.

Remover um valor `x` de uma árvore binária de busca aleatorizada é similar ao processo de remover de uma `Treap`. Encontramos o nó, `u`, que contém `x` e então executamos rotações que repetidademente aumentam a profundidade de `u` até que ele se torne uma folha, neste ponto podemos removê-lo da árvore. A escolha de executar uma rotação à esquerda ou à direita em cada passo é aleatória.

1. Com probalidade $\texttt{u.esquerdo.size}/(\texttt{u.size}-1)$, executamos uma rotação à direita em `u`, fazendo `u.esquerdo` a raiz da subárvore que anteriormente tinha a raiz em `u`.

2. Com probalidade $\texttt{u.direito.size}/(\texttt{u.size}-1)$, executamos uma rotação à esquerda em `u`, fazendo `u.direito` a raiz da subárvore que anteriormente tinha a raiz em `u`.

Novamente, podemos facilmente verificar que estas são exatamente as mesmas probabilidades que o algoritmo de remoção em uma `Treap` irá executar uma rotação à esquerda ou à direita de `u`.

As árvores binárias de buscas aleatorizadas têm a desvantagem, comparadas às treaps, de que, quando inserindo ou removendo elementos, elas fazem muitas escolhas aleatórias, e elas devem manter o tamanho das subárvores. Uma vantagem da árvore binária de buscas aleatorizada em relação às treaps é que o tamanho das subárvores podem ter outro propósito, a saber, ter acesso por posição em um tempo esperado de $O(\log \texttt{n})$ (veja Exercício 7.10). Em comparação, as prioridades aleatórias armazenadas nos nós da treap não têm outro uso que manter a árvore balanceada.

Exercício 7.1. Ilustre a inserção de 4.5 (com prioridade 7) e a seguir 7.5 (com prioridade 20) na `Treap` da Figura 7.5.

Exercício 7.2. Ilustre a remoção de 5 e a seguir de 7 na `Treap` da Figura 7.5.

Exercício 7.3. Prove a asserção de que existem 21,964,800 sequências que geram a árvore do lado direito da Figura 7.1. (Dica: Forneça uma fórmula recursiva para o número de sequências que geram uma árvore binária completa de altura h e avalie esta fórmula para $h = 3$.)

Exercício 7.4. Projete e implemente o método `permute(a)` que tem como entrada um vetor, `v`, que contém `n` valores distintos e que permute `v`. O método deve executar no tempo $O(n)$ e você deve provar que cada uma das `n`! possíveis permutações de `v` são igualmente prováveis.

Exercício 7.5. Use ambas as partes do Lema 7.2 para provar que o número esperado de rotações executadas por uma operaçãoadd(`x`) (e consequentemente também pela operação `remove(x)`) é $O(1)$.

Exercício 7.6. Modifique a implementação de `Treap` dada aqui para que ela não armazene explicitamente as prioridades. Em vez disso, ela deve simulá-las fazendo hash com `hashCode()` de cada nó.

Exercício 7.7. Suponha que uma árvore binária de busca armazene, em cada nó, `u`, a altura, `u.height`, da subárvore com raiz em `u`, e o tamanho, `u.size` da subárvore com raiz em `u`.

1. Mostre como, se executamos uma rotação à esquerda ou à direita em `u`, então essas duas quantidades devem ser atualizadas, em um tempo constante, para todos os nós afetados pela rotação.

2. Explique porque o mesmo resultado não é possível se tentamos armazenar a profundidade, `u.depth`, de cada nó `u`.

Exercício 7.8. Projete e implemente um algoritmo que construa uma `Treap` a partir de um vetor ordenado, `v`, de `n` elementos. Este método deve executar em um tempo $O(n)$ no pior caso de deve construir uma `Treap` que seja indistinguível de uma cujos elementos de `v` foram inseridos um por um usando o método `add(x)`.

Exercício 7.9. Este exercício trabalha os detalhes de como podemos buscar de maneira eficiente em uma `Treap` dado um ponteiro que esteja próximo ao nó que estamos procurando.

1. Projete e implemente uma `Treap`em que cada nó mantenha registro dos valores mínimo e máximo de sua subárvore.

2. Usando esta informação extra, crie um método `fingerFind(x,u)` que execute a operação `find(x)` com a ajuda deste ponteiro para o nó `u` (que esperamos esteja próximo do nó que contém `x`). Esta operação deve iniciar em `u` e percorrer em direção ao topo até encontrar um nó `w` tal que $\texttt{w.min} \leq \texttt{x} \leq \texttt{w.max}$. Deste ponto em diante, ela deve executar uma busca padrão para `x` começando de `w`. (Pode-se mostrar que `fingerFind(x,u)` consome um tempo $O(1 + \log r)$, onde r é o número de elementos na treap cujos valores está entre `x` e `u.x`.)

3. Estenda sua implementação para uma versão que inicie as operações `find(x)` a partir do nó mais recentemente encontrado por `find(x)`.

Exercício 7.10. Projete e implemente uma versão de uma `Treap` que inclua uma operação `get(i)` que retorna a chave de posição `i` na `Treap`. (Dica: Faça com que cada nó, `u`, mantenha o registro do tamanho da subárvore com raiz em `u`.)

Exercício 7.11. Implemente uma `TreapList`, uma implementação da interface da `Lista` como uma treap. Cada nó na treap deve armazenar um item da lista, e um percurso em-ordem da the treap encontra os itens na mesma ordem que eles ocorrem na lista. Todas as operações da `Lista`, `get(i)`, `set(i,x)`, `add(i,x)` e `remove(i)` devem executar em um tempo esperado $O(\log n)$.

Exercício 7.12. Projete e implemente uma versão de uma `Treap` que suporte a operação `split(x)`. Esta operação remove todos os valores de uma `Treap` que sejam maiores que `x` e retorna uma segunda `Treap` que contém todos os valores removidos.
Exemplo: o código `t2 = t.split(x)` remove de `t` todos os valores maiores que `x` e retorna uma nova `Treap` `t2` que contém todos esses valores. A operação `split(x)` deve executar em um tempo esperado de $O(\log n)$.

Aviso: Para esta mdficação funcionar adequadamente e ainda permitir que o método `size()` execute em um tempo constante, [e necessário implementar as modificações em Exercício 7.10.

Exercício 7.13. Projete e implemente uma versão de uma `Treap` que suporte a operação `absorb(t2)`, que pode ser concebida como o inverso da operação `split(x)`. Esta operação remove todos os valores de `Treap t2` e os insere no receptor. Esta operação pressupõe que o menor valor em `t2` é maior que o maior valor no recpetor. A operação `absorb(t2)` deve exexutar em um tempo esperado de $O(\log n)$.

Exercício 7.14. Implemente a árvore binária de buscas aleatorizada de Martinez, como discutido nesta seção. Compare o desempenho de sua implementação com a implementação da `Treap`.

Capítulo 8

Árvores de Bode Expiatório

Neste capítulo, estudamos uma estrutura de dados de árvore de busca binária, a ScapegoatTree. Essa estrutura é baseada na sabedoria comum de que, quando algo dá errado, a primeira coisa que as pessoas tendem a fazer é encontrar alguém para culpar (o *bode expiatório*). Uma vez que a culpa esteja firmemente estabelecida, podemos deixar o bode expiatório resolver o problema.

Uma ScapegoatTree se mantém equilibrada por meio de operações de *reconstrução parcial*. Durante uma operação de reconstrução parcial, uma subárvore inteira é desconstruída e reconstruída em uma subárvore perfeitamente equilibrada. Há muitas maneiras de reconstruir uma subárvore com raiz no nó u em uma árvore perfeitamente equilibrada. Um dos mais simples é percorrer a subárvore de u, reunindo todos os nós em um array, a e, em seguida, criar recursivamente uma subárvore equilibrada usando a. Se fizermos m = a.length/2, então o elemento a[m] se tornará a raiz da nova subárvore, a[0],...,a[m−1] são armazenados recursivamente na subárvore esquerda e a[m+1],...,a[a.length−1] são armazenados recursivamente na subárvore direita.

ScapegoatTree

```
void rebuild(Node *u) {
  int ns = BinaryTree<Node>::size(u);
  Node *p = u->parent;
  Node **a = new Node*[ns];
  packIntoArray(u, a, 0);
  if (p == nil) {
    r = buildBalanced(a, 0, ns);
    r->parent = nil;
```

```
  } else if (p->right == u) {
    p->right = buildBalanced(a, 0, ns);
    p->right->parent = p;
  } else {
    p->left = buildBalanced(a, 0, ns);
    p->left->parent = p;
  }
  delete[] a;
}
int packIntoArray(Node *u, Node **a, int i) {
  if (u == nil) {
    return i;
  }
  i = packIntoArray(u->left, a, i);
  a[i++] = u;
  return packIntoArray(u->right, a, i);
}
```

Uma chamada para `rebuild(u)` leva um tempo $O(\texttt{size}(u))$. A subárvore resultante tem altura mínima; não há árvore de altura menor que tenha `size(u)` nós.

8.1 ScapegoatTree: Uma Árvore de Busca Binária com Reconstrução Parcial

Uma `ScapegoatTree` é uma `BinarySearchTree` que, além de rastrear o número `n` dos nós na árvore também mantém um contador, `q`, que mantém um limite superior no número de nós.

ScapegoatTree

```
int q;
```

Em todos os momentos, `n` e `q` obedecem às seguintes desigualdades:

$$q/2 \le n \le q \ .$$

Além disso, uma `ScapegoatTree` tem altura logarítmica; nunca a altura da árvore do bode expiatório excede

$$\log_{3/2} q \le \log_{3/2} 2n < \log_{3/2} n + 2 \ . \tag{8.1}$$

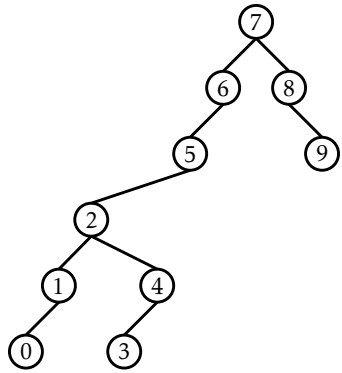


Figura 8.1: Uma ScapegoatTree com 10 nós e altura 5.

Mesmo com essa restrição, uma ScapegoatTree pode parecer surpreendentemente desequilibrada. A árvore em Figura 8.1 tem $q = n = 10$ e altura $5 < \log_{3/2} 10 \approx 5.679$.

A implementação da operação `find(x)` em uma ScapegoatTree é feita usando o algoritmo padrão para pesquisar em uma BinarySearchTree (veja Seção 6.2). Isso leva um tempo proporcional à altura da árvore que, por (8.1) é $O(\log n)$.

Para implementar a operação `add(x)`, primeiro incrementamos `n` e `q` e usamos o algoritmo usual para adicionar `x` a uma árvore de busca binária; nós procuramos `x` e então adicionamos uma nova folha `u` com `u.x = x`. Neste ponto, podemos ter sorte e a profundidade de `u` pode não exceder $\log_{3/2} q$. Se assim for, então deixamos como está e não fazemos mais nada.

Infelizmente, às vezes acontece que $\texttt{depth}(u) > \log_{3/2} q$. Neste caso, precisamos reduzir a altura. Este não é um grande trabalho; existe apenas um nó, a saber `u`, cuja profundidade excede $\log_{3/2} q$. Para corrigir `u`, percorremos de `u` de volta para a raiz procurando um *bode expiatório*, `w`. O bode expiatório, `w`, é um nó muito desequilibrado. Ele tem a propriedade

$$\frac{\texttt{size}(w.\texttt{child})}{\texttt{size}(w)} > \frac{2}{3}\ , \tag{8.2}$$

onde `w.child` é o filho de `w` no caminho da raiz para `u`. Em breve prova-

remos que existe um bode expiatório. Por enquanto, podemos dar como certo. Uma vez que encontramos o bode expiatório `w`, destruímos completamente a subárvore com raiz em `w` e a reconstruímos em uma árvore de busca binária perfeitamente balanceada. Sabemos, de (8.2), que, mesmo antes da adição da subárvore `u`, `w` não era uma árvore binária completa. Portanto, quando reconstruirmos `w`, a altura diminuirá em pelo menos 1, de modo que a altura da `ScapegoatTree` seja novamente no máximo $\log_{3/2} q$.

ScapegoatTree

```
bool add(T x) {
  // first do basic insertion keeping track of depth
  Node *u = new Node;
  u->x = x;
  u->left = u->right = u->parent = nil;
  int d = addWithDepth(u);
  if (d > log32(q)) {
    // depth exceeded, find scapegoat
    Node *w = u->parent;
    int a = BinaryTree<Node>::size(w);
    int b = BinaryTree<Node>::size(w->parent);
    while (3*a <= 2*b) {
      w = w->parent;
      a = BinaryTree<Node>::size(w);
      b = BinaryTree<Node>::size(w->parent);
    }
    rebuild(w->parent);
  } else if (d < 0) {
    delete u;
    return false;
  }
  return true;
}
```

Se ignorarmos o custo de encontrar o bode expiatório `w` e reconstruir a subárvore enraizada em `w`, então o tempo de execução de `add(x)` é dominado pela pesquisa inicial, que toma um tempo $O(\log q) = O(\log n)$. Iremos contabilizar o custo de encontrar o bode expiatório e a reconstrução usando a análise amortizada na próxima seção.

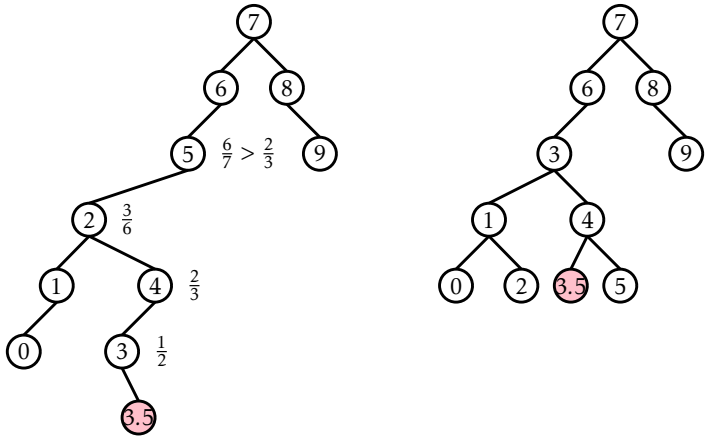


Figura 8.2: Inserindo 3.5 em uma ScapegoatTree aumenta sua altura para 6, o que viola (8.1) pois $6 > \log_{3/2} 11 \approx 5.914$. Um bode expiatório é encontrado no nó contendo 5.

A implementação de remove(x) em uma ScapegoatTree é muito simples. Nós procuramos x e o removemos usando o algoritmo usual para remover um nó de uma BinarySearchTree. (Note que isso nunca pode aumentar a altura da árvore). Em seguida, nós decrementamos n, mas deixamos q inalterado. Finalmente, nós verificamos se $q > 2n$ e, em caso afirmativo, então nós *reconstruímos a árvore inteira* em uma árvore de busca binária perfeitamente balanceada e configuramos $q = n$.

ScapegoatTree

```
bool remove(T x) {
  if (BinarySearchTree<Node,T>::remove(x)) {
    if (2*n < q) {
      rebuild(r);
      q = n;
    }
    return true;
  }
  return false;
}
```

Novamente, se ignorarmos o custo da reconstrução, o tempo de execução

da operação remove(x) será proporcional à altura da árvore e, portanto, será $O(\log n)$.

8.1.1 Análise de Corretude e Tempo de Execução

Nesta seção, analisamos a corretude e o tempo de execução amortizado das operações em uma ScapegoatTree. Primeiro provamos a corretude mostrando que, quando a operação add(x) resulta em um nó que viola a Condição (8.1), então sempre podemos encontrar um bode expiatório:

Lema 8.1. *Seja u um nó de profundidade $h > \log_{3/2} q$ em uma ScapegoatTree. Então existe um nó w no caminho do u para a raiz tal que*

$$\frac{\texttt{size}(w)}{\texttt{size}(\texttt{parent}(w))} > 2/3 \ .$$

Demonstração. Suponha, por uma questão de contradição, que este não é o caso, e

$$\frac{\texttt{size}(w)}{\texttt{size}(\texttt{parent}(w))} \leq 2/3 \ .$$

para todos os nós w no caminho de u para a raiz. Indique o caminho da raiz para u como $r = u_0, \ldots, u_h = u$. Então nós temos $\texttt{size}(u_0) = n$, $\texttt{size}(u_1) \leq \frac{2}{3}n$, $\texttt{size}(u_2) \leq \frac{4}{9}n$ e, de maneira mais geral,

$$\texttt{size}(u_i) \leq \left(\frac{2}{3}\right)^i n \ .$$

Mas isso gera uma contradição, já que $\texttt{size}(u) \geq 1$, daí

$$1 \leq \texttt{size}(u) \leq \left(\frac{2}{3}\right)^h n < \left(\frac{2}{3}\right)^{\log_{3/2} q} n \leq \left(\frac{2}{3}\right)^{\log_{3/2} n} n = \left(\frac{1}{n}\right) n = 1 \ . \qquad \square$$

Em seguida, analisamos as partes do tempo de execução que ainda não foram contabilizadas. Existem duas partes: o custo das chamadas para size(u) ao procurar por nós de bodes expiatórios e o custo das chamadas para rebuild(w) quando encontramos um bode expiatório w. O custo das chamadas para size(u) pode estar relacionado ao custo das chamadas para rebuild(w), da seguinte forma:

Lema 8.2. *Durante uma chamada para* add(x) *em uma ScapegoatTree, o custo de encontrar o bode expiatório w e reconstruir a subárvore com raiz em w é $O(\texttt{size}(w))$.*

Demonstração. O custo de reconstruir o nó do bode expiatório `w`, uma vez encontrado, é $O(\texttt{size}(\texttt{w}))$. Ao procurar pelo nó do bode expiatório, chamamos $\texttt{size}(\texttt{u})$ em uma sequência de nós $\texttt{u}_0,\ldots,\texttt{u}_k$ até encontrarmos o bode expiatório $\texttt{u}_k = \texttt{w}$. No entanto, como $\texttt{u}_k$ é o primeiro nó dessa sequência que é um bode expiatório, sabemos que

$$\texttt{size}(\texttt{u}_i) < \frac{2}{3}\texttt{size}(\texttt{u}_{i+1})$$

para todo $i \in \{0,\ldots,k-2\}$. Portanto, o custo de todas as chamadas para $\texttt{size}(\texttt{u})$ é

$$\begin{aligned}
O\left(\sum_{i=0}^{k} \texttt{size}(\texttt{u}_{k-i})\right) &= O\left(\texttt{size}(\texttt{u}_k) + \sum_{i=0}^{k-1} \texttt{size}(\texttt{u}_{k-i-1})\right) \\
&= O\left(\texttt{size}(\texttt{u}_k) + \sum_{i=0}^{k-1} \left(\frac{2}{3}\right)^i \texttt{size}(\texttt{u}_k)\right) \\
&= O\left(\texttt{size}(\texttt{u}_k)\left(1 + \sum_{i=0}^{k-1} \left(\frac{2}{3}\right)^i\right)\right) \\
&= O(\texttt{size}(\texttt{u}_k)) = O(\texttt{size}(\texttt{w}))\ ,
\end{aligned}$$

onde a última linha segue do fato de que a soma é uma série geometricamente decrescente. □

Tudo o que resta é provar um limite superior no custo de todas as chamadas para $\texttt{rebuild}(\texttt{u})$ durante uma sequência de m operações:

Lema 8.3. *Começando com uma* `ScapegoatTree` *vazia, qualquer sequência de m operações* $\texttt{add}(\texttt{x})$ *e* $\texttt{remove}(\texttt{x})$ *causam um tempo de no máximo* $O(m \log m)$ *a ser usado pelas operações de* $\texttt{rebuild}(\texttt{u})$.

Demonstração. Para provar isso, usaremos um *esquema de crédito*. Nós imaginamos que cada nó armazena um número de créditos. Cada crédito pode pagar por algumas unidades de tempo gasto c constante na reconstrução. O esquema dá um total de $O(m \log m)$ créditos e cada chamada para $\texttt{rebuild}(\texttt{u})$ é paga com créditos armazenados em `u`.

Durante uma inserção ou exclusão, damos um crédito a cada nó no caminho para o nó inserido ou nó excluído, `u`. Desta forma distribuímos

no máximo $\log_{3/2} \mathtt{q} \leq \log_{3/2} m$ créditos por operação. Durante uma exclusão, também armazenamos um crédito adicional "de lado". Assim, no total, distribuímos no máximo $O(m \log m)$ créditos. Tudo o que resta é mostrar que esses créditos são suficientes para pagar todas as chamadas para `rebuild(u)`.

Se chamarmos `rebuild(u)` durante uma inserção, é porque `u` é um bode expiatório. Suponha, sem perda de generalidade, que

$$\frac{\mathtt{size(u.left)}}{\mathtt{size(u)}} > \frac{2}{3} \ .$$

Usando o fato que

$$\mathtt{size(u)} = 1 + \mathtt{size(u.left)} + \mathtt{size(u.right)}$$

deduzimos que

$$\frac{1}{2}\mathtt{size(u.left)} > \mathtt{size(u.right)}$$

e, portanto,

$$\mathtt{size(u.left)} - \mathtt{size(u.right)} > \frac{1}{2}\mathtt{size(u.left)} > \frac{1}{3}\mathtt{size(u)} \ .$$

Agora, a última vez que uma subárvore contendo `u` foi reconstruída (ou quando `u` foi inserido, se uma subárvore contendo `u` nunca foi reconstruída), temos

$$\mathtt{size(u.left)} - \mathtt{size(u.right)} \leq 1 \ .$$

Portanto, o número de operações `add(x)` ou `remove(x)` que afetaram `u.left` ou `u.right` desde então é pelo menos

$$\frac{1}{3}\mathtt{size(u)} - 1 \ .$$

e há, portanto, pelo menos, tantos créditos armazenados em `u` que estão disponíveis para pagar pelo tempo $O(\mathtt{size(u)})$ que leva para chamar `rebuild(u)`.

Se chamarmos `rebuild(u)` durante uma exclusão, é porque $\mathtt{q} > 2\mathtt{n}$. Neste caso, temos $\mathtt{q} - \mathtt{n} > \mathtt{n}$ créditos armazenados "de lado", e nós os usamos para pagar o tempo $O(\mathtt{n})$ que leva para reconstruir a raiz. Isso conclui a prova. □

8.1.2 Resumo

O seguinte teorema resume o desempenho da estrutura de dados ScapegoatTree:

Teorema 8.1. *Uma ScapegoatTree implementa a interface SSet. Ignorando o custo das operações de* `rebuild(u)`, *uma ScapegoatTree suporta as operações* `add(x)`, `remove(x)` *e* `find(x)` *em tempo* $O(\log n)$ *por operação.*

Além disso, começando com uma ScapegoatTree vazia, qualquer sequência de m operações `add(x)` *e* `remove(x)` *resulta em um total de* $O(m \log m)$ *de tempo gasto durante todas as chamadas para* `rebuild(u)`.

8.2 Discussão e Exercícios

O termo *árvore scapegoat* é creditado a Galperin e Rivest [33], que definem e analisam essas árvores. No entanto, a mesma estrutura foi descoberta anteriormente por Andersson [5, 7], que as chamou de *árvores balanceadas geral*, já que elas podem ter qualquer forma desde que sua altura seja pequena.

Teste experimentais com a implementação da ScapegoatTree revelará que ela é consideravelmente mais lenta que as outras implementações de SSet neste livro. Isso pode ser um pouco surpreendente, já que a altura limitada a

$$\log_{3/2} q \approx 1.709 \log n + O(1)$$

é melhor que o tamanho esperado de um caminho de busca em uma Skiplist e não muito distante de uma Treap. A implementação pode ser otimizada armazenando os tamanhos de subárvores explicitamente em cada nó ou reutilizando os tamanhos de subárvores já calculados (Exercícios 8.5 e 8.6). Mesmo com essas otimizações, sempre haverá sequências de operações `add(x)` e `delete(x)` para as quais a ScapegoatTree demora mais que outras implementações de SSet.

Essa lacuna no desempenho se deve ao fato de que, diferentemente das outras implementações da SSet discutidas neste livro, uma ScapegoatTree pode passar muito tempo se reestruturando. Exercício 8.3 pede para você provar que existem sequências de n operações nas quais a Scape-

goatTree irá gastar um tempo da ordem de $n \log n$ em chamadas para rebuild(u). Isso está em contraste com outras implementações do SSet discutidas neste livro, que fazem apenas $O(n)$ alterações estruturais durante uma sequência de n operações. Esta é, infelizmente, uma consequência necessária do fato de que uma ScapegoatTree faz toda a sua reestruturação por meio de chamadas para rebuild(u) [20].

Apesar de sua falta de desempenho, existem aplicações em que uma ScapegoatTree poderia ser a escolha certa. Isso ocorreria sempre que houvesse dados adicionais associados a nós que não pudessem ser atualizados em tempo constante quando uma rotação fosse executada, mas que pudesse ser atualizada durante uma operação de rebuild(u). Em tais casos, a ScapegoatTree e estruturas relacionadas baseadas em reconstrução parcial podem funcionar. Um exemplo de tal aplicação é descrito em Exercício 8.11.

Exercício 8.1. Ilustre a adição dos valores 1.5 e 1.6 na ScapegoatTree em Figura 8.1.

Exercício 8.2. Ilustre o que acontece quando a sequência $1, 5, 2, 4, 3$ é adicionada a uma ScapegoatTree vazia e mostre onde os créditos descritos na prova do Lema 8.3 vão e como eles são usados durante essa sequência de adições.

Exercício 8.3. Mostre que, se começarmos com uma ScapegoatTree vazia e chamarmos add(x) para $x = 1, 2, 3, \ldots, n$, então o tempo total gasto durante as chamadas para rebuild(u) é de pelo menos $cn \log n$ para alguma constante $c > 0$.

Exercício 8.4. A ScapegoatTree, conforme descrita neste capítulo, garante que o comprimento do caminho de pesquisa não exceda $\log_{3/2} q$.

1. Projete, analise e implemente uma versão modificada de ScapegoatTree onde o comprimento do caminho de pesquisa não exceda $\log_b q$, onde b é um parâmetro com $1 < b < 2$.

2. O que sua análise e/ou seus experimentos dizem sobre o custo amortizado de find(x), add(x) e remove(x) como uma função de n e b?

Exercício 8.5. Modifique o método add(x) da ScapegoatTree para que ele não perca tempo recalculando os tamanhos de subárvores que já foram

computados. Isso é possível porque, no momento em que o método deseja calcular `size(w)`, ele já calculou um dos `size(w.left)` ou `size(w.right)`. Compare o desempenho de sua implementação modificada com a implementação fornecida aqui.

Exercício 8.6. Implemente uma segunda versão da estrutura de dados `ScapegoatTree` que armazena e mantém explicitamente os tamanhos da subárvore com raiz em cada nó. Compare o desempenho da implementação resultante com a implementação original de `ScapegoatTree`, bem como a implementação de Exercício 8.5.

Exercício 8.7. Reimplemente o método `rebuild(u)` discutido no início deste capítulo para que ele não exija o uso de um array para armazenar os nós da subárvore que está sendo reconstruída. Em vez disso, ele deve usar recursão para primeiro conectar os nós a uma lista encadeada e depois converter essa lista encadeada em uma árvore binária perfeitamente balanceada. (Existem implementações recursivas muito elegantes de ambas as etapas.)

Exercício 8.8. Analise e implemente uma `WeightBalancedTree`. Esta é uma árvore na qual cada nó `u`, exceto a raiz, mantém o *invariante de equilíbrio* no qual $\texttt{size}(u) \leq (2/3)\texttt{size}(u.\texttt{parent})$. As operações `add(x)` e `remove(x)` são idênticas às operações da `BinarySearchTree` padrão, exceto que sempre que a invariante de balanceamento for violada em um nó `u`, a subárvore com raiz em `u.parent` é reconstruída. Sua análise deve mostrar que as operações na `WeightBalancedTree` são executadas em um tempo amortizado de $O(\log n)$.

Exercício 8.9. Analise e implemente uma `CountdownTree`. Em uma `CountdownTree`, cada nó `u` mantém um *temporizador* `u.t`. As operações `add(x)` e `remove(x)` são exatamente as mesmas que em uma `BinarySearchTree` padrão, exceto que, sempre que uma dessas operações afeta a subárvore `u`, `u.t` é decrementado. Quando $u.t = 0$, toda a subárvore com raiz em `u` é reconstruída em uma árvore de busca binária perfeitamente balanceada. Quando um nó `u` está envolvido em uma operação de reconstrução (seja porque `u` foi recriado ou um dos ancestrais de `u` foi reconstruído) `u.t` é reiniciado para `size(u)/3`.

Sua análise deve mostrar que as operações em uma `CountdownTree` são executadas em um tempo amortizado de $O(\log n)$. (Dica: primeiro mostre que cada nó `u` satisfaz a alguma versão de uma invariante de balanceamento.)

Exercício 8.10. Analise e implemente uma `DynamiteTree`. Em uma `DynamiteTree`, cada nó `u` mantém as informações sobre o tamanho da subárvore com raiz em `u` em uma variável `u.size`. As operações `add(x)` e `remove(x)` são exatamente as mesmas que em uma `BinarySearchTree` padrão, exceto que, sempre que uma dessas operações afetar uma subárvore do nó `u`, `u` *explode* com probabilidade $1/$`u.size`. Quando `u` explode, toda a subárvore é reconstruída em uma árvore de busca binária perfeitamente balanceada.

Sua análise deve mostrar que as operações em uma `DynamiteTree` são executadas em um tempo esperado igual a $O(\log n)$.

Exercício 8.11. Projete e implemente uma estrutura de dados `Sequencia` que mantenha uma sequência (lista) de elementos. Ela suporta estas operações:

- `addAfter(e)`: Adiciona um novo elemento após o elemento `e` na sequência. Retorna o elemento recém-adicionado. (Se `e` for nulo, o novo elemento será adicionado no início da sequência.)
- `remove(e)`: Remove `e` da sequencia.
- `testBefore(e1,e2)`: retorna `true` se e somente se `e1` venha antes de `e2` na sequência.

As duas primeiras operações devem ser executadas em um tempo amortizado igual a $O(\log n)$. A terceira operação deve ser executada em tempo constante.

A estrutura de dados Sequence pode ser implementada ao armazenar os elementos em algo como uma ScapegoatTree, na mesma ordem em que ocorrem na sequência. Para implementar `testBefore(e1,e2)` em tempo constante, cada elemento `e` é rotulado com um inteiro que codifica o caminho da raiz para `e`. Dessa forma, `testBefore(e1,e2)` pode ser implementado comparando os rótulos de `e1` e `e2`.

Capítulo 9

Árvores Rubro-Negras

Neste capítulo apresentamos as árvores rubro-negras, uma versão das árvores binárias de busca com altura logarítmica. As árvores rubro-negras são uma das estruturas de dados mais utilizadas. Elas aparecem como a principal estrutura de busca em muitas implementações de bibliotecas, incluindo a Java Collections Framework e várias implementações da C++ Standard Template Library. Elas também são utilizadas no kernel do sistema operacional Linux. Existem várias razões para a popularidade das árvores rubro-negras:

1. Uma árvore rubro-negra com n elementos tem altura de no máximo $2\log n$.

2. As operações `add(x)` e `remove(x)` em uma árvore rubro-negra são executadas em tempo $O(\log n)$ no *pior caso*.

3. O número de rotações amortizadas realizadas durante uma operação `add(x)` ou `remove(x)` é constante.

As duas primeiras dessas propriedades já colocam as árvores rubro-negras à frente de skiplists, treaps, e árvores scapegoat. Skiplists e treaps dependem de randomização, e seus tempos de execução $O(\log n)$ são apenas esperados. As árvores scapegoat têm limite de altura garantido, mas as operações `add(x)` e `remove(x)` executam apenas em tempo amortizado $O(\log n)$. A terceira propriedade é a cereja do bolo. Ela nos diz que o tempo necessário para adicionar ou remover um elemento `x` é limitado

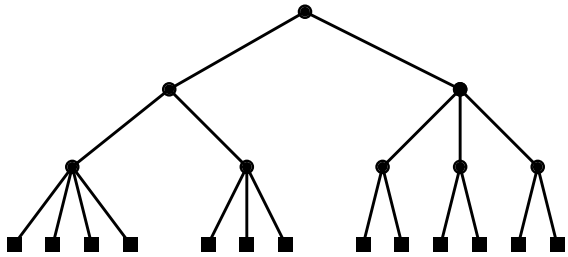


Figura 9.1: Uma árvore 2-4 de altura 3.

pelo tempo necessário para encontrar x.[1]

No entanto, as propriedades legais das árvores rubro-negras vêm com um preço: complexidade de implementação. Manter um limite de $2 \log n$ na altura não é fácil. Isso exige uma análise cuidadosa de vários casos. Devemos garantir que a implementação faça exatamente a coisa certa em cada caso. Uma mudança de cor ou rotação mal feita produz um bug que pode ser muito difícil de entender e rastrear.

Em vez de pular diretamente para a implementação das árvores rubro-negras, primeiro passaremos alguma base sobre uma estrutura de dados relacionada: a árvore 2-4. Isso dará uma visão de como as árvores rubro-negras foram descobertas e porque a manutenção eficiente delas pode ser possível.

9.1 Árvores 2-4

Uma árvore 2-4 é uma árvore enraizada com as seguintes propriedades:

Propriedade 9.1 (Altura)**.** Todas as folhas têm a mesma profundidade.

Propriedade 9.2 (grau)**.** Cada nó interno tem 2, 3 ou 4 filhos.

Um exemplo de uma árvore 2-4 é mostrado na Figura 9.1. As propriedades das árvores 2-4 implicam que sua altura é logarítmica no número de folhas:

[1] Note que, nesse sentido, as skiplists e treaps também têm essa propriedade. Veja os exercicios 4.6 e 7.5.

Lema 9.1. *Uma árvore 2-4 com* n *folhas tem altura máxima de* $\log n$.

Demonstração. O limite inferior de 2 no número de filhos de um nó interno implica que, se a altura de uma árvore 2-4 for h, então ela tem no mínimo 2^h folhas. Em outras palavras,

$$n \geq 2^h \ .$$

Tirando o log de ambos os lados desta desigualdade resulta em $h \leq \log n$. □

9.1.1 Adicionando uma folha

Adicionar uma folha a uma árvore 2-4 é fácil (veja Figura 9.2). Se nós queremos adicionar uma folha u como filho de algum nó w no penúltimo nível, então simplesmente fazemos u um filho de w. Isso certamente mantém a propriedade da altura, mas poderia violar a propriedade do grau; se w tinha quatro filhos antes de adicionar u, então w agora tem cinco filhos. Neste caso, nós *dividimos* w em dois nós, w e w', tendo dois e três filhos, respectivamente. Mas agora w' não tem pai, então, recursivamente, fazemos w' um filho do pai de w. Novamente, isso pode fazer com que o pai de w tenha muitos filhos, caso em que o dividimos. Este processo continua até alcançar um nó com menos de quatro filhos, ou até dividir a raiz, r, em dois nós r e r'. Nesse último caso, criamos uma nova raiz que tem r e r' como filhos. Isso aumenta a profundidade de todas as folhas simultaneamente e, portanto, mantém a propriedade da altura.

Uma vez que a altura da árvore 2-4 nunca é maior que $\log n$, o processo de adicionar uma folha termina após $\log n$ passos, no máximo.

9.1.2 Removendo uma folha

Remover uma folha da árvore 2-4 é um pouco mais complicado (veja Figura 9.3). Para remover uma folha u do seu pai w, apenas o removemos. Se w tivesse apenas dois filhos antes da remoção de u, então w seria deixado com apenas um filho e violaria a propriedade do grau.

Para corrigir isso, olhamos para o irmão de w, w'. O nó w' certamente existe, dado que o pai de w tenha pelo menos dois filhos. Se w' tem três

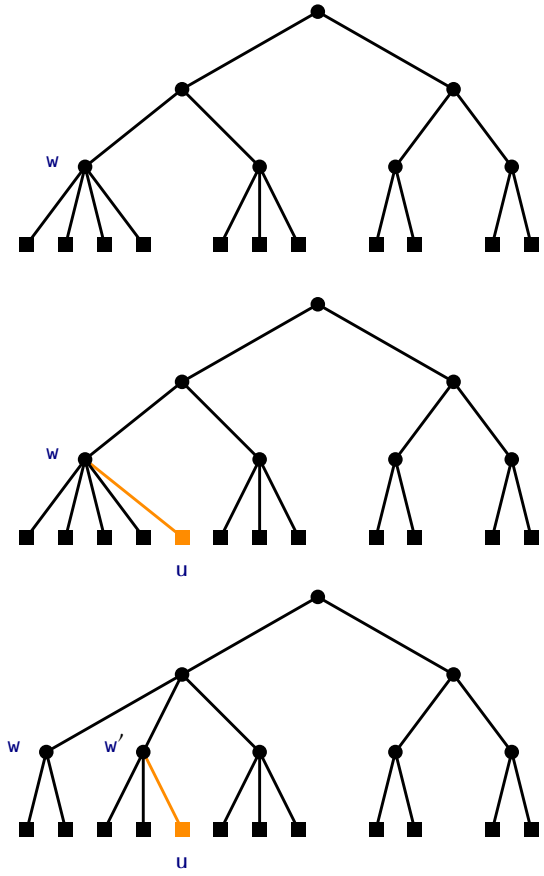


Figura 9.2: Adicionando uma folha na árvore 2-4. Este processo termina depois de uma divisão porque `w.parent` tem um grau inferior a 4 antes da adição.

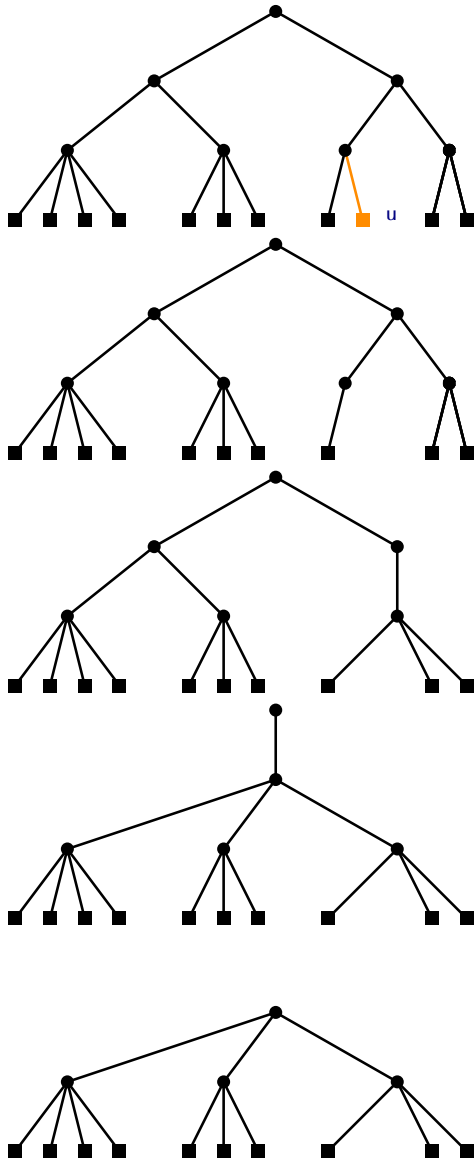


Figura 9.3: Removendo uma folha de uma árvore 2-4. Este processo vai até a raiz porque cada um dos antecessores u e seus irmãos têm somente dois filhos.

ou quatro filhos, então nós levamos um desses filhos de w' para w. Agora w tem dois filhos, w' tem dois ou três filhos e assim terminamos.

Por outro lado, se w' tiver apenas dois filhos, então nós *mesclamos* w e w' em um único nó, w, que tem três filhos. Em seguida, removemos w' do pai de w' recursivamente. Esse processo acaba quando alcançamos um nó, u, onde u ou seu irmão tem mais de dois filhos, ou quando chegamos à raiz. No último caso, se a raiz é deixada com apenas um filho, então nós excluímos a raiz e fazemos do seu filho a nova raiz. Mais uma vez, isso diminui simultaneamente a altura de cada folha e, portanto, mantém a propriedade de altura.

Novamente, uma vez que a altura da árvore nunca é mais de $\log n$, o processo de remoção de uma folha termina após um máximo de $\log n$ passos.

9.2 Árvore Rubro-Negra: Uma Árvore 2-4 Simulada

Uma árvore rubro-negra é uma árvore binária de busca na qual cada nó, u, tem uma *cor* que é *vermelha* ou *preta*. O vermelho é representado pelo valor 0 e preto pelo valor 1.

RedBlackTree

```
class RedBlackNode : public BSTNode<Node, T> {
  friend class RedBlackTree<Node, T>;
  char colour;
};
int red = 0;
int black = 1;
```

Antes e depois de qualquer operação em uma árvore rubro-negra, as duas seguintes propriedades são satisfeitas. Cada propriedade é definida tanto em termos de cores vermelhas e pretas, como em termos dos valores numéricos 0 e 1.

Propriedade 9.3 (altura preta)**.** Há o mesmo número de nós pretos em cada caminho da raiz para a folha. (A soma das cores em qualquer caminho da raiz para a folha é a mesma.)

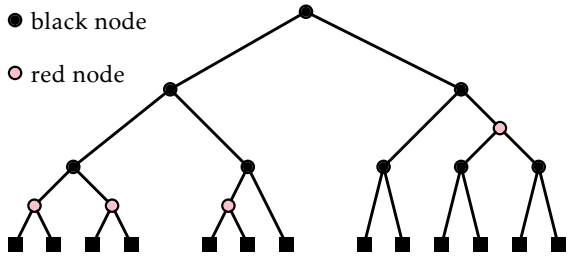


Figura 9.4: Um exemplo de uma árvore rubro-negra com altura preta 3. Os nós externos (`nil`) são desenhados como quadrados.

Propriedade 9.4 (sem-borda-vermelha)**.** Dois nós vermelhos nunca são adjacentes. (Para qualquer nó `u`, exceto a raiz, `u.colour+u.parent.colour` $\geq$ 1.)

Observe que sempre podemos colorir a raiz, `r`, de uma árvore rubro-negra sem violar nenhuma dessas duas propriedades, então assumiremos que a raiz é preta, e os algoritmos para atualizar uma árvore rubro-negra irão manter isso. Outro truque que simplifica a árvore rubro-negra é tratar os nós externos (representados por `nil`) como nós pretos. Desta forma, todo nó real, `u`, de uma árvore rubro-negra tem exatamente dois filhos, cada uma com uma cor bem definida. Um exemplo de árvore rubro-negra é mostrado em Figura 9.4.

9.2.1 Árvores Rubro-Negras e Árvores 2-4

No começo, pode parecer surpreendente que uma árvore rubro-negra possa ser eficientemente atualizada para manter as propriedades altura-preta e sem-borda-vermelha, e parece incomum mesmo considerar estas como propriedades úteis. Contudo, árvores rubro-negras foram projetadas para ser uma simulação eficiente de árvores 2-4 como árvores binárias.

Consulte a Figura 9.5. Considere qualquer árvore rubro-negra, T, tendo `n` nós e executando a seguinte transformação: remova cada nó vermelho `u` e conecte os dois filhos de `u` diretamente ao pai (preto) de `u`. Após essa transformação nós ficamos com uma árvore T' tendo apenas

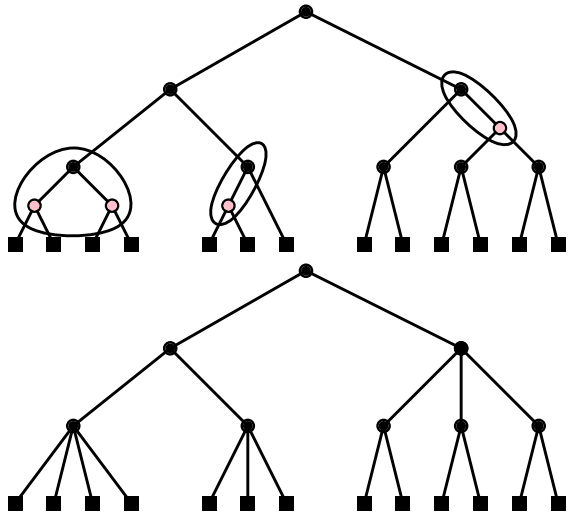


Figura 9.5: Toda árvore rubro-negra possui uma árvore 2-4 correspondente.

nós pretos.

Cada nó interno em T' tem dois, três ou quatro filhos: Um nó preto que começou com dois filhos pretos ainda terá dois filhos pretos após essa transformação. Um nó preto que começou com um filho vermelho e um preto terá três filhos depois dessa transformação. Um nó preto que começou com dois filhos vermelhos terá quatro filhos após essa transformação. Além disso, a propriedade altura-preta agora garante que cada caminho da raiz até a folha em T' tem o mesmo comprimento. Em outras palavras, T' é uma árvore 2-4!

A árvore 2-4 T' tem $n+1$ folhas que correspondem aos $n+1$ nós externos da árvore rubro-negra. Portanto, esta árvore tem, no máximo, altura $\log(n+1)$. Agora, cada caminho da raiz até a folha na árvore 2-4 corresponde a uma rota da raiz da árvore rubro-negra T até um nó externo. O primeiro e último nó neste caminho são pretos, e no máximo um, a cada dois nós internos, é vermelho. Então esta rota tem no máximo $\log(n+1)$ nós pretos e no máximo $\log(n+1)-1$ nós vermelhos. Portanto, a rota mais longa da raiz para qualquer nó *interno* em T é no máximo

$$2\log(n+1) - 2 \leq 2\log n \ ,$$

para qualquer $n \geq 1$. Isso prova a propriedade mais importante da árvore rubro-negra:

Lema 9.2. *A altura da árvore rubro-negra com* n *nós é no máximo* $2 \log n$.

Agora que vimos a relação entre árvores 2-4 e árvores rubro-negras, não é difícil acreditar que podemos manter, eficientemente, uma árvore rubro-negra ao adicionar e remover elementos.

Já vimos que adicionar um elemento em uma *Árvore Binária de Busca* pode ser feito adicionando uma nova folha. Portanto, para implementar `add(x)` em uma árvore rubro-negra, precisamos de um método para simular a divisão de um nó com cinco filhos em uma árvore 2-4. Um nó de árvore 2-4 com cinco filhos é representado por um nó preto que tem dois filhos vermelhos, um dos quais também tem um filho vermelho. Nós podemos dividir esse nó colorindo-o de vermelho e colorindo dois filhos pretos. Um exemplo disso é mostrado em Figura 9.6.

Da mesma forma, implementar o método `remove(x)` requer um método de mesclagem de dois nós, e pegar emprestado um filho de um irmão. Mesclar dois nós é o inverso de uma divisão (mostrado em Figura 9.6), e envolve colorir, de vermelho, dois irmãos pretos e colorir, de preto, o pai vermelho. Pegar emprestado de um irmão é o procedimento mais complicado e envolve rotações e recolorações de nó.

Claro, durante todo isso, ainda devemos manter as propriedades sem-borda-vermelha e altura-preta. Embora não seja mais surpreendente que isso possa ser feito, há uma grande quantidade de casos que precisam ser considerados se tentarmos fazer uma simulação direta de uma árvore 2-4 através de uma árvore rubro-negra. Em algum momento, torna-se mais simples ignorar a árvore 2-4 subjacente e trabalhar diretamente para manter as propriedades da árvore rubro-negra.

9.2.2 Árvore Rubro-Negra inclinada para a esquerda

Não existe uma definição única de árvores rubro-negras. Em vez disso, existe uma família de estruturas que conseguem manter as propriedades altura-preta e sem-borda-vermelha durante as operações `add(x)` e `remove(x)`. Diferentes estruturas fazem isso de diferentes maneiras. Aqui, implementamos a estrutura de dados que chamamos de `RedBlackTree`. Esta

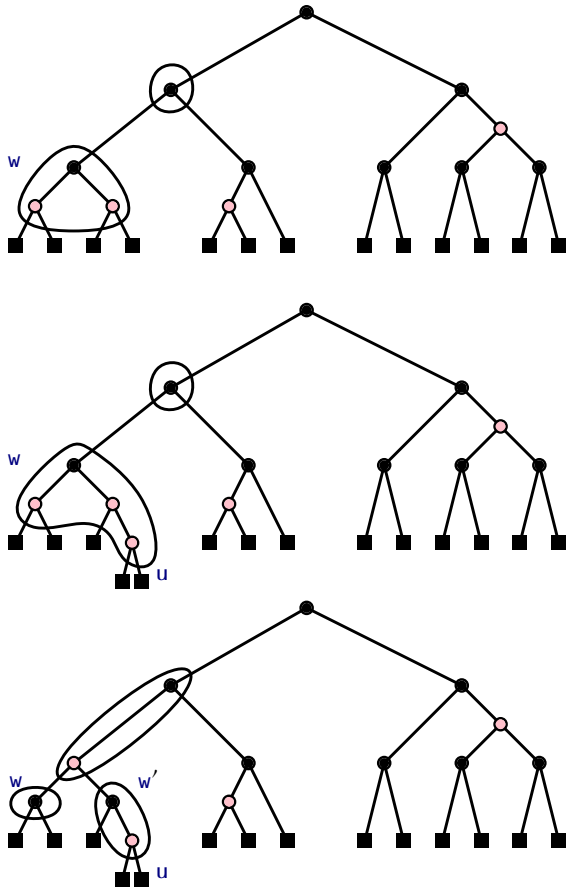


Figura 9.6: Simulando uma operação de divisão na árvore 2-4 durante uma adição em uma árvore rubro-negra. (Isto simula a adição em árvore 2-4 mostrada na Figura 9.2.)

estrutura implementa uma variante particular das árvores rubro-negras que satisfaz uma propriedade adicional:

Propriedade 9.5 (inclinada para a esquerda)**.** Em qualquer nó `u`, se `u.left` for preto, então `u.right` é preto.

Observe que a árvore rubro-negra mostrada em Figura 9.4 não satisfaz a propriedade inclinada-para-esquerda; ela é violada pelo pai do nó vermelho no caminho mais à direita.

O motivo para manter a propriedade inclinada-para-esquerda é que ela reduz o número de casos encontrados ao atualizar a árvore durante `add(x)` e `remove(x)` operações. Em termos de árvores 2-4, isso implica que cada árvore 2-4 tem uma representação única: um nó de grau dois torna-se um nó preto com dois filhos pretos. Um nó de grau três torna-se um nó preto cujo filho esquerdo é vermelho e cujo filho direito é preto. Um nó de grau quatro torna-se um nó preto com dois filhos vermelhos.

Antes de descrever a implementação de `add(x)` e `remove(x)` em detalhe, apresentamos algumas sub-rotinas simples usadas por esses métodos que estão ilustradas na Figura 9.7. As duas primeiras sub-rotinas são para manipular cores, preservando a propriedade altura-negra. O método `pushBlack(u)` recebe como entrada um nó preto `u` que tem dois filhos vermelhos e então colore `u` de vermelho e seus dois filhos de preto. O método `pullBlack(u)` inverte esta operação:

RedBlackTree

```
void pushBlack(Node *u) {
  u->colour--;
  u->left->colour++;
  u->right->colour++;
}
void pullBlack(Node *u) {
  u->colour++;
  u->left->colour--;
  u->right->colour--;
}
```

O método `flipLeft(u)` troca as cores de `u` e `u.right` e então executa uma rotação à esquerda em `u`. Este método inverte as cores desses dois nós, bem como sua relação pai-filho:

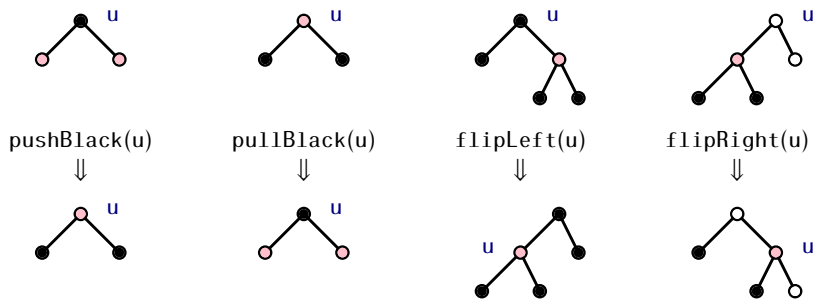


Figura 9.7: Flips, pulls e pushes

RedBlackTree
```
void flipLeft(Node *u) {
  swapcolours(u, u->right);
  rotateLeft(u);
}
```

A operação `flipLeft(u)` é especialmente útil para restaurar a propriedade inclinada-pra-esquerda em um nó `u` que a viola (porque `u.left` é preto e `u.right` é vermelho). Neste caso especial, podemos ter certeza de que esta operação preserva ambas as propriedades altura-preta e sem-borda-vermelha. A operação `flipRight(u)` é simétrica com `flipLeft(u)`, quando os papéis da esquerda e direita são invertidos.

RedBlackTree
```
void flipRight(Node *u) {
  swapcolours(u, u->left);
  rotateRight(u);
}
```

9.2.3 Adição

Para implementar `add(x)` em uma `RedBlackTree`, nós executamos um inserção padrão em `BinarySearchTree` para adicionar uma nova folha, `u`, com `u.x = x` e definir `u.colour = red`. Observe que isso não altera a altura preta de qualquer nó, por isso não viola a propriedade de altura-preta. No entanto, pode violar a propriedade inclinada-para-esquerda (se `u` é

o filho direito de seu pai), e isso pode violar a propriedade sem-borda-vermelha (se o pai de u é vermelho). Para restaurar essas propriedades, chamamos o método addFixup(u).

RedBlackTree

```
bool add(T x) {
  Node *u = new Node();
  u->left = u->right = u->parent = nil;
  u->x = x;
  u->colour = red;
  bool added = BinarySearchTree<Node,T>::add(u);
  if (added)
    addFixup(u);
  else
    delete u;
  return added;
}
```

Ilustrado em Figura 9.8, o método addFixup(u) recebe como entrada um nó u cuja cor é o vermelho e que pode violar as propriedades sem-borda-vermelha e/ou inclinada-para-esquerda. A seguinte discussão é provavelmente impossível de se explicar sem se referir a Figura 9.8 ou recriá-la em um pedaço de papel. De fato, o leitor talvez deseje estudar essa figura antes de continuar.

Se u é a raiz da árvore, podemos colorir u de preto para restaurar as propriedades. Se o irmão de u também for vermelho, então o pai de u deve ser preto, de modo que as propriedades inclinada-para-esquerda e sem-borda-vermelha se mantenham.

Caso contrário, primeiro determinamos se o pai de u, w, viola a propriedade inclinada-para-esquerda, e se for o caso, executamos uma operação flipLeft(w) e definimos u = w. Isso nos deixa em um estado bem definido: u é o filho esquerdo de seu pai, w, então w agora satisfaz a propriedade inclinada-para-esquerda. Tudo o que resta é garantir a propriedade sem-borda-vermelha em u. Nós apenas temos que nos preocupar em caso de w ser vermelho, pois em caso contrário, u já satisfaz a propriedade sem-borda-vermelha.

Uma vez que ainda não terminamos, u é vermelho e w é vermelho. A propriedade sem-borda-vermelha (que só é violada por u e não por w) implica que o avô de u, g, existe e é preto. Se o filho direito de g for verme-

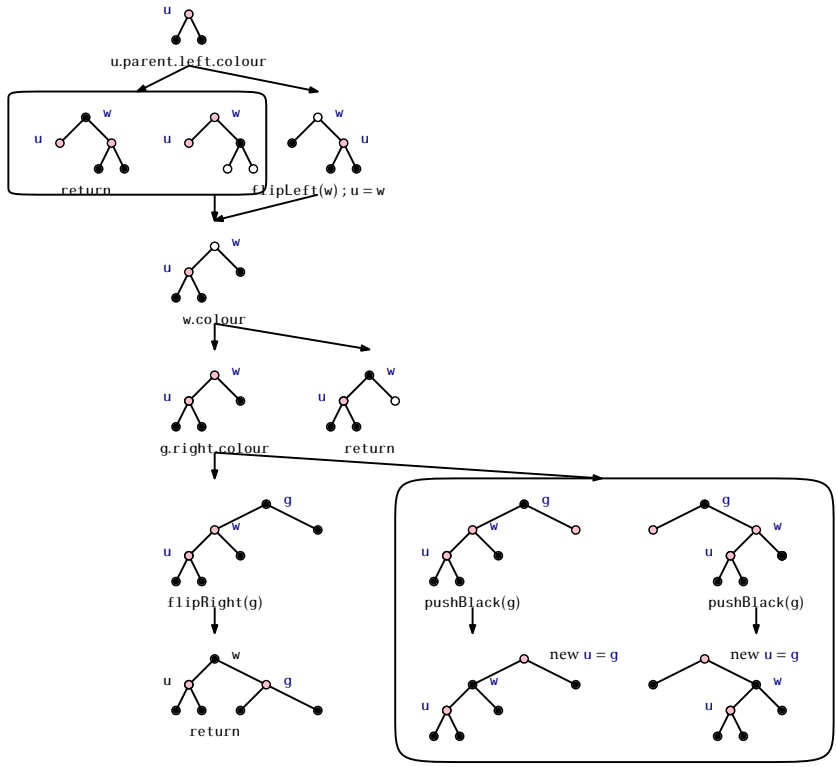


Figura 9.8: Uma única rodada no processo de fixação da propriedade 2 após uma inserção.

lho, então a propriedade inclinada-para-esquerda garante que ambos os filhos de g são vermelhos, e uma chamada de pushBlack(g) faz g vermelho e w preto. Isso restaura a propriedade sem-borda-vermelha em u, mas pode fazer com que seja violada em g, então todo o processo recomeça com u = g.

Se o filho direito de g for preto, então uma chamada para flipRight(g) faz w o pai (preto) de g e dá a w dois filhos vermelhos, u e g. Isso garante que u satisfaça a propriedade sem-borda-vermelha e g satisfaça a propriedade inclinada-para-esquerda. Neste caso, podemos parar.

RedBlackTree

```
void addFixup(Node *u) {
  while (u->colour == red) {
    if (u == r) { // u is the root - done
      u->colour = black;
      return;
    }
    Node *w = u->parent;
    if (w->left->colour == black) { // ensure left-leaning
      flipLeft(w);
      u = w;
      w = u->parent;
    }
    if (w->colour == black)
      return; // no red-red edge = done
    Node *g = w->parent; // grandparent of u
    if (g->right->colour == black) {
      flipRight(g);
      return;
    } else {
      pushBlack(g);
      u = g;
    }
  }
}
```

O método insertFixup(u) leva tempo constante por iteração e cada iteração termina ou move u para mais perto da raiz. Assim sendo, o método insertFixup(u) termina após $O(\log n)$ iterações em tempo $O(\log n)$.

9.2.4 Remoção

A operação `remove(x)` na `RedBlackTree` é a mais complicada para implementar, e isso é verdade para todas as variantes conhecidas de árvore rubro-negra. Assim como a operação `remove(x)` em uma `ÁrvoreBináriaDeBusca`, esta operação resume-se a encontrar um nó `w` com apenas um filho, `u`, e retirar `w` da árvore fazendo `w.parent` adotar `u`.

O problema com isso é que, se `w` for preto, então a propriedade altura-preta agora será violada em `w.parent`. Podemos evitar esse problema temporariamente adicionando `w.colour` em `u.colour`. Claro, isso introduz dois outros problemas: (1) se ambos `u` e `w` começarem preto, então `u.colour + w.colour = 2` (duplo preto), que é uma cor inválida. Se `w` for vermelho, então ele é substituído por um nó preto `u`, que pode violar a propriedade inclinada-para-esquerda em `u.parent`. Ambos os problemas podem ser resolvidos com uma chamada do método `removeFixup(u)`.

RedBlackTree

```
bool remove(T x) {
  Node *u = findLast(x);
  if (u == nil || compare(u->x, x) != 0)
    return false;
  Node *w = u->right;
  if (w == nil) {
    w = u;
    u = w->left;
  } else {
    while (w->left != nil)
      w = w->left;
    u->x = w->x;
    u = w->right;
  }
  splice(w);
  u->colour += w->colour;
  u->parent = w->parent;
  delete w;
  removeFixup(u);
  return true;
}
```

O método `removeFixup(u)` recebe como entrada um nó `u` cuja cor é

preta (1) ou duplo-preto (2). Se u for duplo-preto, então removeFixup(u) executa uma série de rotações e operações de recoloração que movem o nó duplo preto pra cima da árvore até que possa ser eliminado. Durante este processo, o nó u muda até que, no final deste processo, u aponta para a raiz da subárvore que foi alterada. A raiz desta subárvore pode ter mudado de cor. Em particular, pode ter ido de vermelho para preto, então o método removeFixup(u) termina ao verificar se o pai de u viola a propriedade inclinada-para-esquerda, e se for o caso, a corrige.

RedBlackTree

```
void removeFixup(Node *u) {
  while (u->colour > black) {
    if (u == r) {
      u->colour = black;
    } else if (u->parent->left->colour == red) {
      u = removeFixupCase1(u);
    } else if (u == u->parent->left) {
      u = removeFixupCase2(u);
    } else {
      u = removeFixupCase3(u);
    }
  }
  if (u != r) { // restore left-leaning property, if needed
    Node *w = u->parent;
    if (w->right->colour == red && w->left->colour == black) {
      flipLeft(w);
    }
  }
}
```

O método removeFixup(u) é ilustrado em Figura 9.9. Mais uma vez, o seguinte texto será difícil, se não impossível, de se explicar sem se referir a Figura 9.9. Cada iteração do loop em removeFixup(u) processa o nó duplo-preto u com base em um de quatro casos:

Caso 0: u é a raiz. Este é o caso mais fácil de tratar. Nós recolorimos u para ser preto (isso não viola nenhuma das propriedades das árvores rubro-negras).

Caso 1: O irmão de u, v, é vermelho. Nesse caso, o irmão de u é o filho esquerdo de seu pai, w (pela propriedade inclinada-para-esquerda). Nós realizamos um right-flip direito em w e depois prosseguimos para a

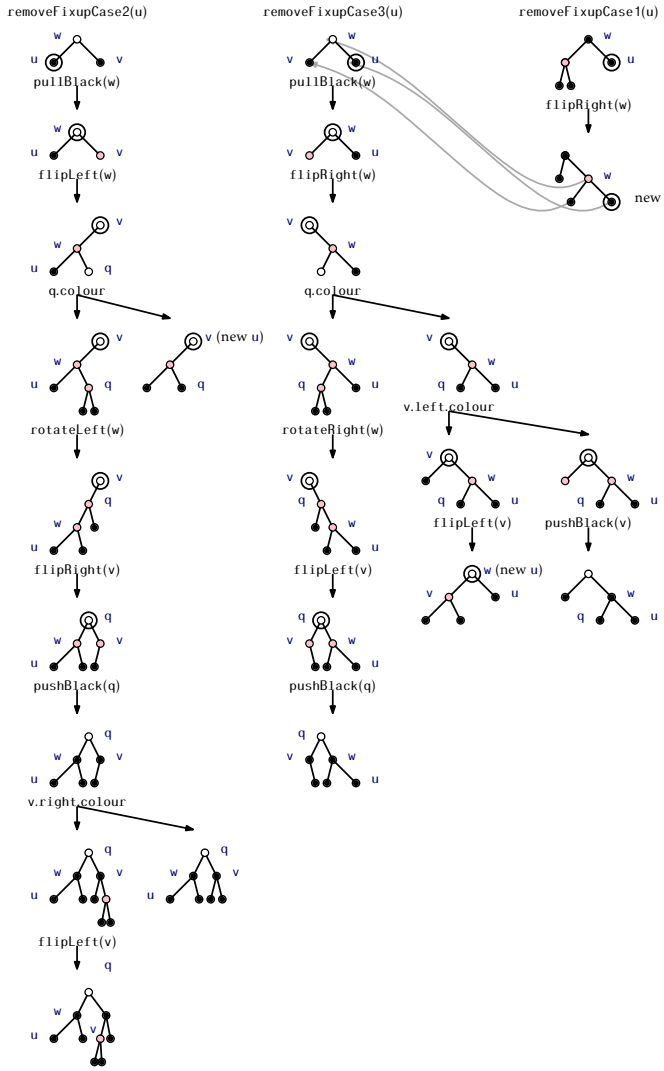


Figura 9.9: Uma única rodada no processo de eliminar um nó duplo-preto após uma remoção.

próxima iteração. Observe que esta ação faz com que o pai de w viole a propriedade inclinada-para-esquerda e a profundidade de u aumente. No entanto, também implica que a próxima iteração estará no Caso 3 com w colorido de vermelho. Ao examinar o Caso 3 abaixo, veremos que o processo irá parar durante a próxima iteração.

RedBlackTree
```
Node* removeFixupCase1(Node *u) {
  flipRight(u->parent);
  return u;
}
```

Caso 2: O irmão de u, v, é preto, e u é o filho esquerdo de seu pai, w. Nesse caso, chamamos pullBlack(w), fazendo u preto, v vermelho e escurecendo a cor de w para preto ou duplo-preto. Neste ponto, w não satisfaz a propriedade inclinada-para-esquerda, então nós chamamos flipLeft(w) para corrigir isso.

Neste ponto, w é vermelho e v é a raiz da subárvore com a qual nós começamos. Precisamos verificar se w faz com que a propriedade sem-borda-vermelha seja violada. Fazemos isso inspecionando o filho direito de w, q. Se q é preto, então w satisfaz a propriedade sem-borda-vermelha e podemos continuar a próxima iteração com u = v.

Caso contrário (q é vermelho), tanto a propriedade sem-borda-vermelha quanto a inclinada-para-esquerda são violadas em q e w, respectivamente. A propriedade inclinada-para-esquerda é restaurada com uma chamada de rotateLeft(w), mas a propriedade sem-borda-vermelha ainda é violada. Neste ponto, q é o filho esquerdo de v, w é o filho esquerdo de q, q e w são vermelhos e v é preto ou duplo-preto. A flipRight(v) faz q o pai de ambos v e w. Seguindo com um pushBlack(q), colore v e w de preto e define a cor de q de volta para a cor original de w.

Neste ponto, o nó duplo-preto foi eliminado e as propriedade sem-borda-vermelha e altura-preta estão restabelecidas. Apenas um problema possível: o filho direito de v pode ser vermelho, caso no qual a propriedade inclinada-para-esquerda seria violada. Verificamos isso e executamos um flipLeft(v) para corrigi-la, se necessário.

RedBlackTree
```
Node* removeFixupCase2(Node *u) {
  Node *w = u->parent;
```

```
  Node *v = w->right;
  pullBlack(w); // w->left
  flipLeft(w); // w is now red
  Node *q = w->right;
  if (q->colour == red) { // q-w is red-red
    rotateLeft(w);
    flipRight(v);
    pushBlack(q);
    if (v->right->colour == red)
      flipLeft(v);
    return q;
  } else {
    return v;
  }
}
```

Caso 3: o irmão de `u` é preto e `u` é o filho certo de seu pai, `w`. Este caso é simétrico ao Caso 2 e é tratado principalmente da mesma maneira. As únicas diferenças vêm do fato de que a propriedade inclinada-para-esquerda é assimétrica, por isso requer uma manipulação diferente.

Como antes, começamos com uma chamada de `pullBlack(w)`, o que torna `v` vermelho e `u` preto. Uma chamada de `flipRight(w)` promove `v` para a raiz da subárvore. Neste ponto `w` é vermelho, e o código se ramifica de duas maneiras dependendo da cor do filho esquerdo de `w`, `q`.

Se `q` estiver vermelho, o código termina exatamente da mesma maneira que o Caso 2 termina, mas é ainda mais simples já que não há perigo de `v` não satifazer a propriedade inclinada-para-esquerda.

O caso mais complicado ocorre quando `q` é preto. Nesse caso, nós examinamos a cor do filho esquerdo de `v`. Se for vermelho, então `v` tem dois filhos vermelhos e seu preto extra podem ser postos pra baixo com uma chamada de `pushBlack(v)`. Neste ponto, `v` agora tem a cor original de `w`, e assim terminamos.

Se o filho esquerdo de `v` for preto, então `v` viola a propriedade inclinada-para-esquerda, e restauramos isso com uma chamada de `flipLeft(v)`. Depois, devolvemos o nó `v` para que a próxima iteração de `removeFixup(u)` continue com `u = v`.

RedBlackTree

```
Node* removeFixupCase3(Node *u) {
  Node *w = u->parent;
```

```
  Node *v = w->left;
  pullBlack(w);
  flipRight(w);             // w is now red
  Node *q = w->left;
  if (q->colour == red) { // q-w is red-red
    rotateRight(w);
    flipLeft(v);
    pushBlack(q);
    return q;
  } else {
    if (v->left->colour == red) {
      pushBlack(v); // both v's children are red
      return v;
    } else { // ensure left-leaning
      flipLeft(v);
      return w;
    }
  }
}
```

Cada iteração de `removeFixup(u)` leva tempo constante. Os casos 2 e 3 terminam ou movem `u` para mais perto da raiz da árvore. O Caso 0 (onde `u` é a raiz) sempre termina, o e Caso 1 leva imediatamente para Caso 3, que também termina. Como a altura da árvore é de no máximo $2\log n$, concluímos que existem no máximo $O(\log n)$ iterações de `removeFixup(u)`, então `removeFixup(u)` é executado em tempo $O(\log n)$.

9.3 Resumo

O seguinte teorema resume o desempenho da estrutura de dados `RedBlackTree`:

Teorema 9.1. *Uma `RedBlackTree` implementa a interface `SSet` e suporta as operações* `add(x)`, `remove(x)` *e* `find(x)` *em tempo de pior caso* $O(\log n)$ *por operação.*

Não incluído no teorema acima é o seguinte bônus extra:

Teorema 9.2. *Começando com uma* `RedBlackTree` *vazia, qualquer seqüência de m operações* `add(x)` *e* `remove(x)` *resultam em um tempo total gasto de* $O(m)$ *durante todas as chamadas de* `addFixup(u)` *e* `removeFixup(u)`.

Nós apenas esboçamos uma prova do Teorema 9.2. Comparando `addFixup(u)` e `removeFixup(u)` com os algoritmos para adicionar ou remover uma folha em uma árvore 2-4, podemos nos convencer de que essa propriedade é herdada de uma árvore 2-4. Em particular, se pudermos mostrar que o tempo total gasto dividindo, mesclando e pegando emprestado em uma árvore 2-4 é $O(m)$, então isso implica Teorema 9.2.

A prova deste teorema para árvores 2-4 usa o método potencial de análise amortizada.[2] Defina o potencial de um nó interno `u` em uma árvore 2-4 como

$$\Phi(u) = \begin{cases} 1 & \text{se } u \text{ tem 2 filhos} \\ 0 & \text{se } u \text{ tem 3 filhos} \\ 3 & \text{se } u \text{ tem 4 filhos} \end{cases}$$

e o potencial de uma árvore 2-4 como a soma dos potenciais dos seus nós. Quando ocorre uma divisão, é porque um nó com quatro filhos se torna dois nós, com dois e três filhos. Isso significa que o potencial total cai em $3 - 1 - 0 = 2$. Quando ocorre uma mesclagem, dois nós que tinham dois filhos são substituídos por um nó com três filhos. O resultado é uma queda de $2 - 0 = 2$ no potencial. Portanto, para cada divisão ou mesclagem, o potencial diminui em dois.

Agora perceba que, se ignorarmos a divisão e a mesclagem de nós, existe apenas um número constante de nós cujo número de filhos é alterado pela adição ou remoção de uma folha. Ao adicionar um nó, um nó tem o número de filhos aumentado em um, aumentando o potencial por no máximo três. Durante a remoção de uma folha, um nó tem seu número de filhos diminuido em um, aumentando o potencial por até um, e dois nós podem estar envolvidos em uma operação de pegar emprestado, aumentando seu potencial total por até um.

Para resumir, cada mesclagem e divisão causam o potencial de cair por ao menos dois. Ignorando a mesclagem e a divisão, cada adição ou remoção faz com que o potencial aumente por até três, e o potencial é sempre

[2] Veja a prova de Lema 2.2 e Lema 3.1 para outras aplicações do método potencial.

não negativo. Portanto, o número de divisões e fusões causadas por m adições ou remoções em uma árvore inicialmente vazia é no máximo $3m/2$. Teorema 9.2 é uma conseqüência dessa análise e a correspondência entre árvores 2-4 e árvores rubro-negras.

9.4 Discussão e Exercícios

Árvores rubro-negras foram introduzidas pela primeira vez por Guibas e Sedgewick [38]. Apesar da sua alta complexidade de implementação, elas são encontradas em algumas das bibliotecas e aplicações mais utilizadas. A maioria das apostilas de algoritmos e e estruturas de dados discutem algumas variantes de árvores rubro-negras.

Andersson [6] descreve uma versão de árvores balanceadas que são semelhantes às árvores rubro-negras, mas tem a restrição adicional de qualquer nó ter no máximo um filho vermelho. Isso implica que essas árvores simulam árvores 2-3 ao invés de árvores 2-4. Elas são significativamente mais simples, no entanto, que a estrutura `RedBlackTree` apresentada neste capítulo.

Sedgewick [64] descreve duas versões de árvores rubro-negras inclinada para esquerda. Estas usam a recursão junto com uma simulação de divisão de cima para baixo e mesclando em árvores 2-4. A combinação dessas duas técnicas fazem um código curto e elegante.

Uma estrutura de dados relacionada e mais antiga é a *ávore AVL* [3]. As árvores de AVL são *balanceada em altura*: Em cada nó u, as alturas da subárvore enraizada em `u.left` e da subárvore enraizada em `u.right` diferem por mais de um. Segue-se imediatamente que, se $F(h)$ for o número mínimo de folhas em uma árvore de altura h, então $F(h)$ obedece a recorrência de Fibonacci

$$F(h) = F(h-1) + F(h-2)$$

com casos base $F(0) = 1$ e $F(1) = 1$. Isso significa que $F(h)$ é aproximadamente $\varphi^h/\sqrt{5}$, onde $\varphi = (1+\sqrt{5})/2 \approx 1.61803399$ é o *coeficiente de ouro*. (Mais precisamente, $|\varphi^h/\sqrt{5} - F(h)| \leq 1/2$.) Argumentando como na prova de Lema 9.1, isso implica

$$h \leq \log_\varphi \mathtt{n} \approx 1.440420088 \log \mathtt{n} \ ,$$

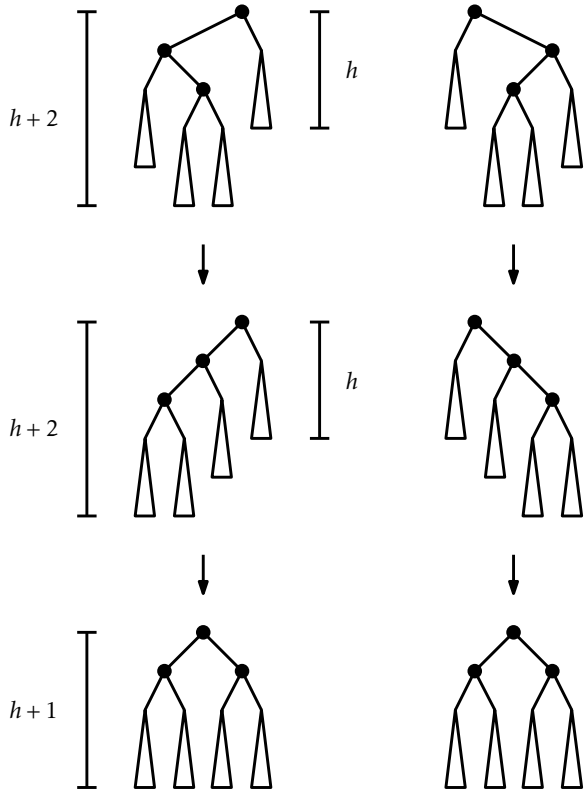


Figura 9.10: Reequilibrando em uma árvore AVL. No máximo, duas rotações são necessárias para converter um nó cujas subárvores tenham uma altura de h e $h+2$ em um nó cujas subárvores têm uma altura de no máximo $h+1$.

então as árvores AVL têm altura menor do que árvores rubro-negras. O balanceamento de altura pode ser mantido durante as operações `add(x)` e `remove(x)` ao caminhar de volta ao caminho da raiz e realizar uma operação de rebalanceamento em cada nó `u` onde a altura das subarvores esquerda e direita de `u` diferem em dois. Veja Figura 9.10.

As variantes de Andersson e de Sedgewick da árvore rubro-negra e as árvores AVL são mais simples de implementar do que a estrutura `RedBlackTree` definida aqui. Infelizmente, nenhuma delas pode garantir que o tempo amortizado gasto rebalanceamento é $O(1)$ por atualização. Em

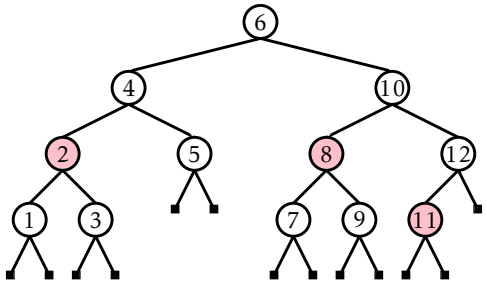


Figura 9.11: Uma árvore rubro-negra para praticar.

particular, não há análogo ao Teorema 9.2 para essas estruturas.

Exercício 9.1. Ilustre a árvore 2-4 que corresponde à `RedBlackTree` da Figura 9.11.

Exercício 9.2. Ilustre a adição de 13, depois de 3.5, e depois de 3.3 na `RedBlackTree` da Figura 9.11.

Exercício 9.3. Ilustre a remoção de 11, depois de 9, e depois de 5 na `RedBlackTree` da Figura 9.11.

Exercício 9.4. Mostre que, para valores arbitrariamente grandes de n, há árvores rubro-negras com n nós com altura de $2\log n - O(1)$.

Exercício 9.5. Considere as operações `pushBlack(u)` e `pullBlack(u)`. O que fazem essas operações para a árvore 2-4 subjacente que está sendo simulada pela árvore rubro-negra?

Exercício 9.6. Mostre que, para valores arbitrariamente grandes de n, existem sequências de operações `add(x)` e `remove(x)` que ocasionam árvores rubro-negras com n nós de altura $2\log n - O(1)$.

Exercício 9.7. Por que o método `remove(x)` na implementação `RedBlackTree` executa a atribuição `u.parent = w.parent`? Isso já não deveria estar feito pela chamada de `splice(w)`?

Exercício 9.8. Suponha que uma árvore 2-4, T, tenha n_ℓ folhas e n_i nós internos.

1. Qual é o valor mínimo de n_i em função de n_ℓ?

2. Qual é o valor máximo de n_i em função de n_ℓ?

3. Se T' é uma árvore rubro-negra que representa T, então, quantos nós vermelhos tem T'?

Exercício 9.9. Suponha que você tenha uma árvore binária de busca com n nós e altura de no máximo $2\log n - 2$. É sempre possível colorir os nós vermelhos e pretos de modo que a árvore satisfaça as propriedades altura-preta e sem-borda-vermelha? Se positivo, também pode ser feito para satisfazer a propriedade inclinada-para-esquerda?

Exercício 9.10. Suponha que você tenha duas árvores rubro-negras T_1 e T_2 de mesma altura preta, h, e tal que a maior chave em T_1 é menor do que a menor chave em T_2. Mostre como combinar T_1 e T_2 em uma única árvore rubro-negra em tempo $O(h)$.

Exercício 9.11. Estenda sua solução para Exercício 9.10 para o caso em que as duas árvores T_1 e T_2 têm alturas pretas diferentes, $h_1 \neq h_2$. O tempo de execução deve ser $O(\max\{h_1, h_2\})$.

Exercício 9.12. Prove que, durante uma operação `add(x)`, uma árvore AVL deve executar no máximo uma operação de rebalanceamento (que envolve no máximo duas rotações; veja Figura 9.10). Dê um exemplo de uma árvore AVL e uma operação `remove(x)` naquela árvore que requer operações de rebalanceamento da ordem de $\log n$

Exercício 9.13. Implemente uma classe `AVLTree` que implementa árvores AVL conforme descrito acima. Compare seu desempenho com o da implementação `RedBlackTree`. Qual implementação tem uma operação `find(x)` mais rápida?

Exercício 9.14. Projete e implemente uma série de experimentos que comparam a performance relativa de `find(x)`, `add(x)` e `remove(x)` para as implementações `SSet` de `SkiplistSSet`, `ScapegoatTree`, `Treap` e `RedBlackTree`. Certifique-se de incluir vários cenários de teste, incluindo casos em que os dados são aleatórios, já ordenado, são removidos em ordem aleatória, são removidos em ordem ordenada, e assim por diante.

Capítulo 10

Heaps

Neste capítulo, discutiremos duas implementações da estrutura de dados de prioridade `Fila` extremamente útil. Ambas as estruturas são um tipo especial de árvore binária chamada *heap*, o que significa "uma pilha desorganizada." Isso contrasta com as árvores de busca binária que podem ser consideradas como uma pilha altamente organizada.

A primeira implementação de *heap* usa um array para simular uma árvore binária completa. Esta implementação muito rápida é a base de um dos mais rápidos algoritmos de ordenação conhecidos, o chamado *heapsort* (ver Seção 11.1.3). A segunda implementação é baseada em árvores binárias mais flexíveis. Ela suporta uma operação `meld(h)` que permite que a fila de prioridades absorva os elementos de uma segunda fila de prioridade `h`.

10.1 `BinaryHeap`: Uma árvore binária implícita

Nossa primeira implementação de uma `Fila` (de prioridade) é baseada em uma técnica que tem mais de quatrocentos anos de idade. O *método Eytzinger* nos permite representar uma árvore binária completa como um array, colocando os nós da árvore em ordem de largura (ver Seção 6.1.2). Desta forma, a raiz é armazenada na posição 0, o filho esquerdo da raiz é armazenado na posição 1, o filho direito da raiz na posição 2, o filho esquerdo do filho esquerdo da raiz é armazenado na posição 3 e assim por diante. Ver Figura 10.1.

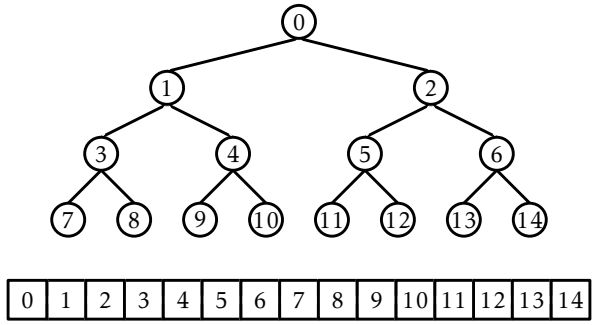


Figura 10.1: O método de Eytzinger representa uma árvore binária completa como um array.

Se aplicarmos o método de *Eytzinger* a uma árvore suficientemente grande, alguns padrões surgem. O filho esquerdo do nó no índice `i` está no índice `left(i)` = 2`i` + 1 e o filho direito do nó no índice `i` está no índice `right(i)` = 2`i` + 2. O pai do nó no índice `i` está no índice `parent(i)` = (`i` − 1)/2.

BinaryHeap
```
int left(int i) {
  return 2*i + 1;
}
int right(int i) {
  return 2*i + 2;
}
int parent(int i) {
  return (i-1)/2;
}
```

Uma `BinaryHeap` usa essa técnica para representar implicitamente uma árvore binária em que os elementos são *ordenados pelo heap*: o valor armazenado em qualquer índice `i` não é menor que o valor armazenado no índice `parent(i)`, com a exceção do valor da raiz, `i` = 0. Segue-se que o menor valor na `Fila` de prioridade é, portanto, armazenado na posição 0 (a raiz).

Na `BinaryHeap`, os `n` elementos são armazenados em um array `a`:

BinaryHeap
```
array<T> a;
```

```
int n;
```

Implementar a operação add(x) é bastante simples. Como acontece com todas as estruturas baseadas em array, primeiro verificamos se a está cheio (verificando se a.length = n) e, em caso afirmativo, aumentamos a. Em seguida, colocamos x no local a[n] e incrementamos n. Neste ponto, tudo o que resta é garantir que mantemos a propriedade do *heap*. Fazemos isso trocando repetidamente x por seu pai até que x não seja menor que seu pai. Ver Figura 10.2.

BinaryHeap

```
bool add(T x) {
  if (n + 1 > a.length) resize();
  a[n++] = x;
  bubbleUp(n-1);
  return true;
}
void bubbleUp(int i) {
  int p = parent(i);
  while (i > 0 && compare(a[i], a[p]) < 0) {
    a.swap(i,p);
    i = p;
    p = parent(i);
  }
}
```

Implementar a operação remove(), que remove o menor valor do *heap*, é um pouco mais complicado. Nós sabemos onde o menor valor está (na raiz), mas precisamos substituí-lo depois de removê-lo e garantir que mantemos a propriedade de *heap*.

A maneira mais fácil de fazer isso é substituir a raiz pelo valor a[n − 1], excluir esse valor e decrementar n. Infelizmente, o novo elemento raiz agora provavelmente não é o menor elemento, por isso ele precisa ser movido para baixo. Fazemos isso comparando repetidamente esse elemento com seus dois filhos. Se é o menor dos três, então estamos prontos. Caso contrário, nós trocamos este elemento pelo menor de seus dois filhos e continuamos.

BinaryHeap

```
T remove() {
  T x = a[0];
```

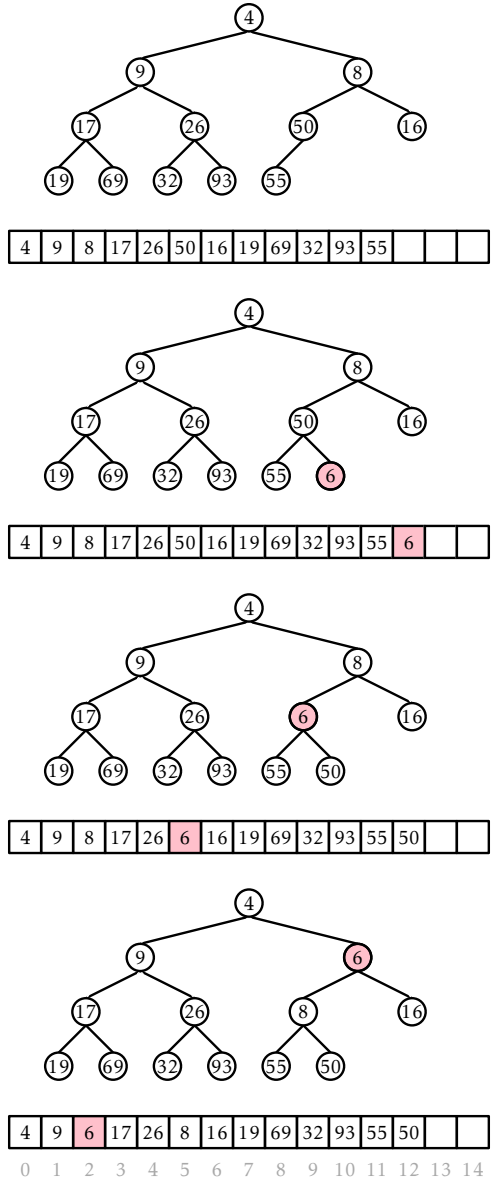


Figura 10.2: Adicionando o valor 6 à `BinaryHeap`.

```
  a[0] = a[--n];
  trickleDown(0);
  if (3*n < a.length) resize();
  return x;
}
void trickleDown(int i) {
  do {
    int j = -1;
    int r = right(i);
    if (r < n && compare(a[r], a[i]) < 0) {
      int l = left(i);
      if (compare(a[l], a[r]) < 0) {
        j = l;
      } else {
        j = r;
      }
    } else {
      int l = left(i);
      if (l < n && compare(a[l], a[i]) < 0) {
        j = l;
      }
    }
    if (j >= 0)  a.swap(i, j);
    i = j;
  } while (i >= 0);
}
```

Como com outras estruturas baseadas em array, iremos ignorar o tempo gasto em chamadas para `resize()`, uma vez que elas podem ser contabilizadas usando o argumento de amortização do Lema 2.1. Os tempos de execução de `add(x)` e `remove()` dependem da altura da árvore binária (implícita). Felizmente, esta é uma árvore binária *completa*; cada nível, exceto o último, tem o número máximo possível de nós. Portanto, se a altura dessa árvore for h, ela terá pelo menos 2^h nós.

$$n \geq 2^h \ .$$

Tomando logaritmos em ambos os lados desta equação dá

$$h \leq \log n \ .$$

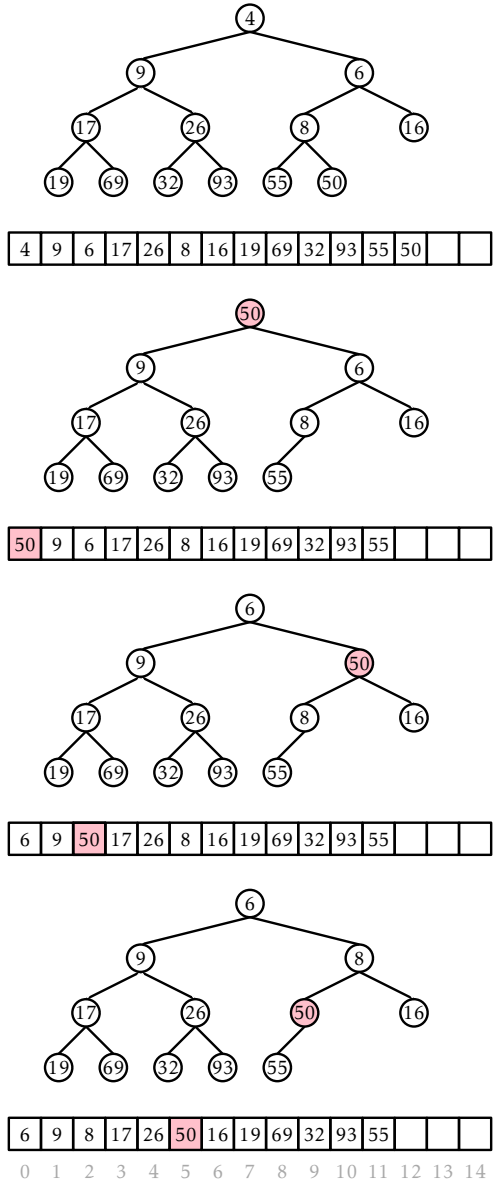


Figura 10.3: Removendo o valor mínimo, 4, de uma `BinaryHeap`.

Portanto, ambas as operações `add(x)` e `remove()` são executadas em um tempo $O(\log n)$.

10.1.1 Resumo

O seguinte teorema resume o desempenho de uma `BinaryHeap`:

Teorema 10.1. *Uma `BinaryHeap` implementa a interface `Fila` (de prioridade). Ignorando o custo das chamadas para `resize()`, a `BinaryHeap` suporta as operações `add(x)` e `remove()` em um tempo por operação de $O(\log n)$.*

Além disso, começando com uma `BinaryHeap` vazia, qualquer sequência de m operações `add(x)` e `remove()` resulta em um total de tempo $O(m)$ gasto durante todas as chamadas para `resize()`.

10.2 `MeldableHeap`: Uma Heap fusionável aleatória

Nesta seção, descrevemos o `MeldableHeap`, uma implementação da `Fila` de prioridade, na qual a estrutura subjacente também é uma árvore binária ordenada por heap. No entanto, ao contrário de uma `BinaryHeap` no qual a árvore binária subjacente é completamente definida pelo número de elementos, não há restrições quanto à forma da árvore binária subjacente na `MeldableHeap`; qualquer coisa serve.

As operações `add(x)` e `remove()` em uma `MeldableHeap` são implementadas em termos da operação `merge(h1,h2)`. Essa operação usa dois nós de heap `h1` e `h2` e mescla-os, retornando um nó de heap que é a raiz de uma heap que contém todos os elementos na subárvore com raiz em `h1` e todos os elementos na subárvore com raiz em `h2`.

O bom de uma operação `merge(h1,h2)` é que ela pode ser definida recursivamente. Veja Figura 10.4. Se `h1` ou `h2` for `nil`, então estamos mesclando com um conjunto vazio, então retornamos `h2` ou `h1`, respectivamente. Caso contrário, assuma $\texttt{h1.x} \leq \texttt{h2.x}$ pois, se $\texttt{h1.x} > \texttt{h2.x}$, poderemos inverter os papéis de `h1` e `h2`. Então, sabemos que a raiz da heap mesclada conterá `h1.x` e podemos mesclar recursivamente `h2` com `h1.left` ou `h1.right`, como desejamos. É aqui que entra a randomização e lançamos uma moeda para decidir se devemos mesclar `h2` com `h1.left` ou `h1.right`:

MeldableHeap
```
Node* merge(Node *h1, Node *h2) {
  if (h1 == nil) return h2;
  if (h2 == nil) return h1;
  if (compare(h1->x, h2->x) > 0) return merge(h2, h1);
        // now we know h1->x <= h2->x
  if (rand() % 2) {
    h1->left = merge(h1->left, h2);
    if (h1->left != nil) h1->left->parent = h1;
  } else {
    h1->right = merge(h1->right, h2);
    if (h1->right != nil) h1->right->parent = h1;
  }
  return h1;
}
```

Na próxima seção, mostramos que `merge(h1,h2)` é executado em um tempo esperado de $O(\log n)$, onde `n` é o número total de elementos em `h1` e `h2`.

Com o acesso a uma operação `merge(h1,h2)`, a operação `add(x)` é fácil. Criamos um novo nó `u` contendo `x` e depois mesclamos `u` com a raiz do heap:

MeldableHeap
```
bool add(T x) {
  Node *u = new Node();
  u->left = u->right = u->parent = nil;
  u->x = x;
  r = merge(u, r);
  r->parent = nil;
  n++;
  return true;
}
```

Isto leva um tempo esperado de $O(\log(n+1)) = O(\log n)$.

A operação `remove()` é igualmente fácil. O nó que queremos remover é a raiz, então apenas mesclamos seus dois filhos e fazemos a raiz ser o resultado:

MeldableHeap
```
T remove() {
  T x = r->x;
```

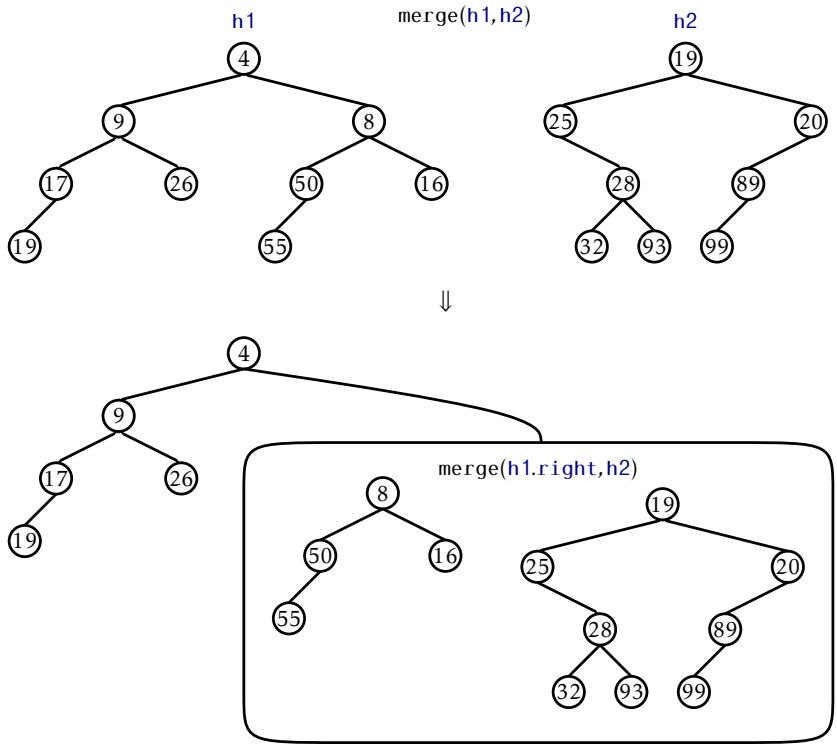


Figura 10.4: A mesclagem de `h1` e `h2` é feita mesclando `h2` com um dos `h1.left` ou `h1.right`.

```
  Node *tmp = r;
  r = merge(r->left, r->right);
  delete tmp;
  if (r != nil) r->parent = nil;
  n--;
  return x;
}
```

Novamente, isto leva um tempo esperado de $O(\log n)$.

Além disso, uma `MeldableHeap` pode implementar muitas outras operações em um tempo esperado de $O(\log n)$, incluindo:

- `remove(u)`: remove o nó `u` (e sua chave `u.x`) do heap.
- `absorb(h)`: adicione todos os elementos da `MeldableHeap` `h` a este heap, esvaziando `h` no processo.

Cada uma dessas operações pode ser implementada usando um número constante de operações de `merge(h1,h2)`, cada uma levando um tempo esperado de $O(\log n)$.

10.2.1 Análise de `merge(h1,h2)`

A análise de `merge(h1,h2)` é baseada na análise de um passeio aleatório em uma árvore binária. Um *passeio aleatório* em uma árvore binária começa na raiz da árvore. Em cada passo da caminhada aleatória, uma moeda é lançada e, dependendo do resultado desse sorteio, a caminhada prossegue para a esquerda ou para o filho direito do nó atual. A caminhada termina quando cai da árvore (o nó atual se torna `nil`).

O seguinte lema é um tanto notável porque não depende de forma alguma da forma da árvore binária:

Lema 10.1. *O comprimento esperado de um passeio aleatório em uma árvore binária com* n *nós é no máximo* $\log(n+1)$.

Demonstração. A prova é por indução em n. No caso base, $n = 0$ e a caminhada tem comprimento $0 = \log(n+1)$. Suponha agora que o resultado seja verdadeiro para todos os inteiros não negativos $n' < n$.

Faça n_1 indicar o tamanho da subárvore esquerda da raiz, de modo que $n_2 = n - n_1 - 1$ seja o tamanho da subárvore direita da raiz.

Começando na raiz, a caminhada dá um passo e depois continua em uma subárvore de tamanho n_1 ou n_2. Pela nossa hipótese indutiva, a duração esperada da caminhada é então

$$\mathrm{E}[W] = 1 + \frac{1}{2}\log(n_1 + 1) + \frac{1}{2}\log(n_2 + 1) \ ,$$

já que cada um de n_1 e n_2 é menor que n. Como log é uma função côncava, $\mathrm{E}[W]$ é maximizado quando $n_1 = n_2 = (n-1)/2$. Portanto, o número esperado de passos feitos pelo passeio aleatório é

$$\begin{aligned}\mathrm{E}[W] &= 1 + \frac{1}{2}\log(n_1 + 1) + \frac{1}{2}\log(n_2 + 1) \\ &\leq 1 + \log((n-1)/2 + 1) \\ &= 1 + \log((n+1)/2) \\ &= \log(n+1) \ .\end{aligned}$$

□

Fazemos uma rápida digressão para observar que, para os leitores que conhecem um pouco da teoria da informação, a prova de Lema 10.1 pode ser expressa em termos de entropia.

Prova Teórica de Informação de Lema 10.1. Faça d_i indicar a profundidade do i-ésimo nó externo e lembre-se de que uma árvore binária com n nós possui $n+1$ nós externos. A probabilidade de o passeio aleatório atingir o i-ésimo nó externo é exatamente $p_i = 1/2^{d_i}$, então o comprimento esperado do passeio aleatório é dado por

$$H = \sum_{i=0}^{n} p_i d_i = \sum_{i=0}^{n} p_i \log\left(2^{d_i}\right) = \sum_{i=0}^{n} p_i \log(1/p_i)$$

O lado direito desta equação é facilmente reconhecível como a entropia de uma distribuição de probabilidade sobre $n+1$ elementos. Um fato básico sobre a entropia de uma distribuição sobre $n+1$ elementos é que ela não excede $\log(n+1)$, o que comprova o lema. □

Com este resultado em caminhadas aleatórias, agora podemos provar facilmente que o tempo de execução da operação `merge(h1,h2)` é $O(\log n)$.

Lema 10.2. *Se* $h1$ *e* $h2$ *forem as raízes de dois heaps contendo* n_1 *e* n_2 *nós, respectivamente, então o tempo de execução esperado de* `merge(h1,h2)` *é no máximo* $O(\log n)$*, onde* $n = n_1 + n_2$*,*

Demonstração. Cada etapa do algoritmo de mesclagem leva um passo de uma caminhada aleatória, na heap com raiz em `h1` ou na heap com raiz em `h2`. O algoritmo termina quando qualquer um desses dois caminhos aleatórios caírem de sua árvore correspondente (quando `h1` = `null` ou `h2` = `null`). Portanto, o número esperado de passos executados pelo algoritmo de mesclagem é no máximo

$$\log(n_1 + 1) + \log(n_2 + 1) \leq 2 \log n \ . \qquad \square$$

10.2.2 Resumo

O seguinte teorema resume o desempenho de uma `MeldableHeap`:

Teorema 10.2. *Uma `MeldableHeap` implementa a interface `Fila` (de prioridade). Uma `MeldableHeap` suporta as operações* `add(x)` *e* `remove()` *em um tempo esperado de* $O(\log n)$ *por operação.*

10.3 Discussão e Exercícios

A representação implícita de uma árvore binária completa como um array, ou lista, parece ter sido proposta pela primeira vez por Eytzinger [27]. Ele usou essa representação em livros contendo árvores genealógicas de pedigree de famílias nobres. A estrutura de dados `BinaryHeap` descrita aqui foi introduzida pela primeira vez por Williams [76].

A estrutura de dados `MeldableHeap` aleatória descrita aqui aparece primeiramente proposta por Gambin e Malinowski [34]. Outras implementações de meldable heap existem, incluindo heaps de esquerda [16, 48, Seção 5.3.2], heaps binomiais [73], heaps de Fibonacci [30], heaps emparelhadas [29], e heaps inclinadas [70], embora nenhum desses seja tão simples quanto a estrutura `MeldableHeap`.

Algumas das estruturas acima também suportam uma operação `decreaseKey(u,y)` em que o valor armazenado no nó `u` é reduzido para `y`. (É uma pré-condição que $y \leq u.x$.) Na maioria das estruturas anteriores, esta operação pode ser suportada em um tempo $O(\log n)$ removendo o nó `u` e adicionando `y`. No entanto, algumas dessas estruturas podem implementar `decreaseKey(u,y)` com mais eficiência. Em parti-

cular, decreaseKey(u, y) leva um tempo amortizado de $O(1)$ nas heaps de Fibonacci e um tempo amortizado de $O(\log\log n)$ numa versão especial de heaps de emparelhamento [25]. Essa operação decreaseKey(u, y) mais eficiente tem aplicações para acelerar vários algoritmos gráficos, incluindo o algoritmo de caminho mais curto de Dijkstra [30].

Exercício 10.1. Ilustre a adição dos valores 7 e depois 3 ao BinaryHeap mostrado no final de Figura 10.2.

Exercício 10.2. Ilustre a remoção dos próximos dois valores (6 e 8) na BinaryHeap mostrada no final de Figura 10.3.

Exercício 10.3. Implemente o método remove(i), que remove o valor armazenado em a[i] em BinaryHeap. Este método deve ser executado em um tempo $O(\log n)$. Em seguida, explique por que esse método provavelmente não será útil.

Exercício 10.4. Uma árvore d-aria é uma generalização de uma árvore binária na qual cada nó interno possui d filhos. Usando o método de Eytzinger também é possível representar árvores completas de d-aria usando arrays. Trabalhe as equações que, dado um índice i, determinam o índice do pai de i e cada um dos d filhos de i nesta representação.

Exercício 10.5. Usando o que você aprendeu em Exercício 10.4, projete e implemente um *DaryHeap*, a generalização d-aria de um BinaryHeap. Analise os tempos de execução de operações em DaryHeap e teste o desempenho de sua implementação DaryHeap em relação à implementação BinaryHeap fornecida aqui.

Exercício 10.6. Ilustre a adição dos valores 17 e 82 na MeldableHeap h1 mostrada em Figura 10.4. Use uma moeda para simular um bit aleatório quando necessário.

Exercício 10.7. Ilustre a remoção dos próximos dois valores (4 e 8) no MeldableHeap h1 mostrado em Figura 10.4. Use uma moeda para simular um bit aleatório quando necessário.

Exercício 10.8. Implemente o método remove(u) que remove o nó u de uma MeldableHeap. Estemétodo deve executar em um tempo esperado de $O(\log n)$.

Exercício 10.9. Mostre como encontrar o segundo menor valor em uma `BinaryHeap` ou `MeldableHeap` em um tempo constante

Exercício 10.10. Mostre como encontrar o k-ésimo menor valor em uma `BinaryHeap` ou `MeldableHeap` em um tempo $O(k \log k)$. (Dica: usar uma outra heap pode ajudar.)

Exercício 10.11. Suponha que você tenha `k` listas ordenadas com um tamanho total de `n`. Utilizando uma heap, mostre como fundi-las em uma única lista ordenada em um tempo $O(n \log k)$. (Dica: Começar com o caso $k = 2$ pode ser instrutivo.)

Capítulo 11

Algoritmos de Ordenação

Este capítulo discute algoritmos para classificar um conjunto de n itens. Isso pode parecer um tópico estranho para um livro sobre estruturas de dados, mas existem várias boas razões para incluí-lo aqui. O motivo mais óbvio é que dois desses algoritmos de classificação (quicksort e heap-sort) estão intimamente relacionados com duas das estruturas de dados que já estudamos (árvores aleatórias de busca binárias e pilhas, respectivamente).

A primeira parte deste capítulo discute algoritmos que classificam usando apenas comparações e apresenta três algoritmos que são executados em um tempo de $O(n \log n)$. Como se verifica, os três algoritmos são assintoticamente ótimos; nenhum algoritmo que use apenas comparações pode evitar de fazer aproximadamente $n \log n$ comparações no pior caso e até mesmo no caso médio.

Antes de continuar, devemos notar que qualquer das implementações de `SSet` ou de `Fila` de prioridades apresentadas em capítulos anteriores também podem ser usadas para obter um algoritmo de classificação com tempo $O(n \log n)$. Por exemplo, podemos classificar n itens executando n operações `add(x)` seguidas de n operações `remove()` em `BinaryHeap` ou `MeldableHeap`. Alternativamente, podemos usar as n operações `add(x)` em qualquer uma das estruturas de dados da árvore de pesquisa binária e, em seguida, executar uma travessia em ordem (Exercício 6.8) para extrair os elementos ordenados. No entanto, em ambos os casos, temos um alto custo para construir uma estrutura que nunca é totalmente utilizada. A classificação é um problema tão importante que vale a pena desenvolver

métodos diretos que sejam tão rápidos, simples e eficientes em termos de espaço quanto possível.

A segunda parte deste capítulo mostra que, se permitimos outras operações além de comparações, todas as possibilidades são possíveis. De fato, usando a indexação de array, é possível classificar um conjunto de n inteiros na faixa $\{0,\ldots,n^c-1\}$ em um tempo $O(cn)$.

11.1 Ordenação Baseada em Comparações

Nesta seção, apresentamos três algoritmos de classificação: merge-sort, quicksort e heap-sort. Cada um desses algoritmos recebe um array de entrada `a` e classifica os elementos de `a` em ordem não decrescente em um tempo (esperado) de $O(n \log n)$. Esses algoritmos são todos *baseados em comparação*. Esses algoritmos não se importam com o tipo de dados que estão sendo classificados; a única operação que eles fazem nos dados é comparações usando o método `compare(a,b)`. Lembre-se de Seção 1.2.4, que `compare(a,b)` retorna um valor negativo se `a < b`, um valor positivo se `a > b` e zero se `a = b`.

11.1.1 Merge-Sort

O algoritmo *merge-sort* é um exemplo clássico de divisão e conquista recursiva: Se o comprimento de `a` for no máximo 1, então `a` já está classificado, então não fazemos nada. Caso contrário, dividimos `a` em duas metades, $a0 = a[0],\ldots,a[n/2-1]$ e $a1 = a[n/2],\ldots,a[n-1]$. Nós classificamos recursivamente `a0` e `a1`, e então mesclamos (o agora ordenado) `a0` e `a1` para obter nossa matriz totalmente ordenada `a`:

Algorithms

```
void mergeSort(array<T> &a) {
  if (a.length <= 1) return;
  array<T> a0(0);
  array<T>::copyOfRange(a0, a, 0, a.length/2);
  array<T> a1(0);
  array<T>::copyOfRange(a1, a, a.length/2, a.length);
  mergeSort(a0);
  mergeSort(a1);
  merge(a0, a1, a);
```

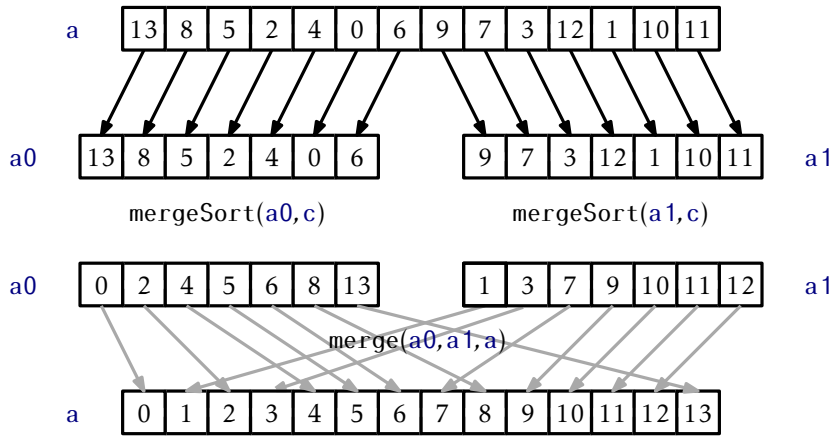


Figura 11.1: A execução de `mergeSort(a,c)`

```
}
```

Um exemplo é mostrado na Figura 11.1.

Em comparação com a classificação, a mesclagem dos dois arrays classificados a0 e a1 é bastante fácil. Nós adicionamos elementos em a um por vez. Se a0 ou a1 estiver vazio, então adicionamos os próximos elementos do outro array (não vazio). Caso contrário, nós colocamos o mínimo do próximo elemento em a0 e o próximo elemento em a1 e o adicionamos a a:

Algorithms

```
void merge(array<T> &a0, array<T> &a1, array<T> &a) {
  int i0 = 0, i1 = 0;
  for (int i = 0; i < a.length; i++) {
    if (i0 == a0.length)
      a[i] = a1[i1++];
    else if (i1 == a1.length)
      a[i] = a0[i0++];
    else if (compare(a0[i0], a1[i1]) < 0)
      a[i] = a0[i0++];
    else
      a[i] = a1[i1++];
  }
```

```
}
```

Observe que o algoritmo `merge`(a0, a1, a, c) executa no máximo n − 1 comparações antes de ficar sem elementos em um a0 ou a1.

Para entender o tempo de execução do merge-sort, é mais fácil pensar nele em termos de sua árvore de recursão. Suponha agora que n é uma potência de dois, de modo que $n = 2^{\log n}$ e log n é um número inteiro. Consulte Figura 11.2. Merge-sort transforma o problema de classificar n elementos em dois problemas, cada um dos elementos de classificação n/2. Estes dois subproblemas são então transformados em dois problemas cada, para um total de quatro subproblemas, cada um de tamanho n/4. Esses quatro subproblemas tornam-se oito subproblemas, cada um de tamanho n/8, e assim por diante. No final deste processo, os subproblemas n/2, cada um de tamanho dois, são convertidos em n problemas, cada um de tamanho um. Para cada subproblema de tamanho $n/2^i$, o tempo gasto para mesclar e copiar dados é $O(n/2^i)$. Uma vez que existem 2^i subproblemas de tamanho $n/2^i$, o tempo total gasto trabalhando em problemas de tamanho 2^i, sem contar chamadas recursivas, é

$$2^i \times O(n/2^i) = O(n) \ .$$

Portanto, a quantidade total de tempo tomado por merge-sort é

$$\sum_{i=0}^{\log n} O(n) = O(n \log n) \ .$$

A prova do seguinte teorema baseia-se na análise anterior, mas tem que ser um pouco mais cuidadosa para lidar com os casos em que n não é uma potência de 2.

Teorema 11.1. *O algoritmo mergeSort(a) executa em um tempo $O(n \log n)$ e faz no máximo $n \log n$ comparações.*

Demonstração. A prova é por indução em n. O caso base, em que n = 1, é trivial; quando apresentado com uma matriz de comprimento 0 ou 1, o algoritmo simplesmente retorna sem realizar comparações.

A combinação de duas listas ordenadas do comprimento total n requer no máximo n − 1 comparações. Faça $C(n)$ indicar o número máximo

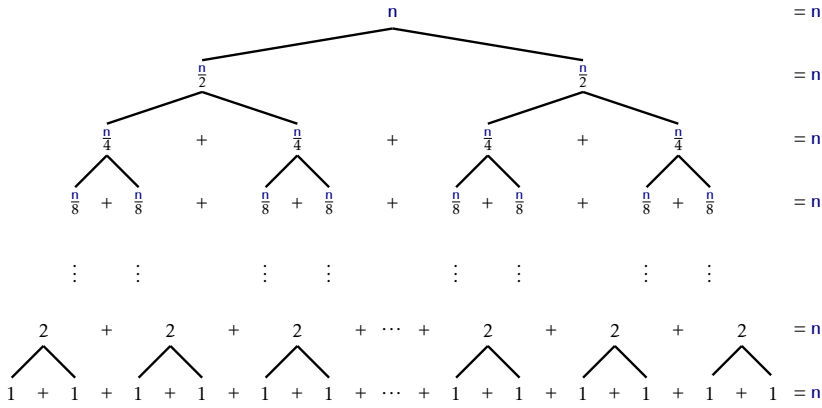


Figura 11.2: A árvore de recursão de merge-sort.

de comparações realizadas por `mergeSort(a,c)` em um array a de comprimento n. Se n for uniforme, então aplicamos a hipótese indutiva aos dois subproblemas e obtemos

$$\begin{aligned} C(n) &\leq n - 1 + 2C(n/2) \\ &\leq n - 1 + 2((n/2)\log(n/2)) \\ &= n - 1 + n\log(n/2) \\ &= n - 1 + n\log n - n \\ &< n\log n \ . \end{aligned}$$

O caso em que n é ímpar é um pouco mais complicado. Para este caso, usamos duas desigualdades que são fáceis de verificar:

$$\log(x+1) \leq \log(x) + 1 \ , \tag{11.1}$$

for all $x \geq 1$ e

$$\log(x + 1/2) + \log(x - 1/2) \leq 2\log(x) \ , \tag{11.2}$$

para todo $x \geq 1/2$. A desigualdade (11.1) vem do fato de que $\log(x) + 1 = \log(2x)$ enquanto (11.2) decorre do fato de que log é uma função côncava.

Com essas ferramentas na mão, temos, para o n ímpar,

$$\begin{aligned}
C(n) &\leq n - 1 + C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) \\
&\leq n - 1 + \lceil n/2 \rceil \log \lceil n/2 \rceil + \lfloor n/2 \rfloor \log \lfloor n/2 \rfloor \\
&= n - 1 + (n/2 + 1/2) \log(n/2 + 1/2) + (n/2 - 1/2) \log(n/2 - 1/2) \\
&\leq n - 1 + n \log(n/2) + (1/2)(\log(n/2 + 1/2) - \log(n/2 - 1/2)) \\
&\leq n - 1 + n \log(n/2) + 1/2 \\
&< n + n \log(n/2) \\
&= n + n(\log n - 1) \\
&= n \log n \ .
\end{aligned}$$

□

11.1.2 Quicksort

O algoritmo *quicksort* é outro algoritmo clássico de divisão e conquista. Ao contrário do merge-sort, que se funde depois de resolver os dois subproblemas, o quicksort faz todo o seu trabalho antecipadamente.

Quicksort é simples de descrever: escolha um elemento aleatório *pivô*, x, de a; divida a no conjunto de elementos menores que x, o conjunto de elementos iguais a x e o conjunto de elementos maior que x; e, finalmente, ordene recursivamente o primeiro e o terceiro conjunto nesta partição. Um exemplo é mostrado na Figura 11.3.

Algorithms

```
void quickSort(array<T> &a) {
  quickSort(a, 0, a.length);
}
void quickSort(array<T> &a, int i, int n) {
  if (n <= 1) return;
  T x = a[i + rand()%n];
  int p = i-1, j = i, q = i+n;
  // a[i..p]<x,  a[p+1..q-1]??x, a[q..i+n-1]>x
  while (j < q) {
    int comp = compare(a[j], x);
    if (comp < 0) {       // move to beginning of array
      a.swap(j++, ++p);
    } else if (comp > 0) {
      a.swap(j, --q);  // move to end of array
    } else {
```

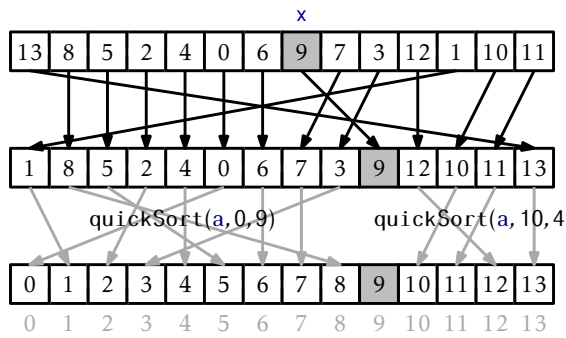


Figura 11.3: Um exemplo da execução de `quickSort(a,0,14)`

```
      j++;                // keep in the middle
    }
  }
  // a[i..p]<x,  a[p+1..q-1]=x, a[q..i+n-1]>x
  quickSort(a, i, p-i+1);
  quickSort(a, q, n-(q-i));
}
```

Tudo isso é feito no próprio array, em vez de fazer as cópias de subarrays serem ordenadas, o método `quickSort(a,i,n,c)` apenas ordena o subarray `a[i]`,...,`a[i+n-1]`. Inicialmente, esse método é invocado com os argumentos `quickSort(a,0,a.length,c)`.

No coração do algoritmo quicksort está o algoritmo de particionamento no local. Este algoritmo, sem usar espaço extra, troca elementos em `a` e calcula índices `p` e `q` de modo que

$$a[i]\begin{cases} < x & \text{if } 0 \le i \le p \\ = x & \text{if } p < i < q \\ > x & \text{if } q \le i \le n-1 \end{cases}$$

Este particionamento, que é feito pelo loop `while` no código, funciona aumentando iterativamente `p` e diminuindo `q` enquanto mantém a primeira e a última dessas condições. Em cada etapa, o elemento na posição `j` é movido ou para a frente, ou à esquerda de onde está ou movido para trás. Nos dois primeiros casos, `j` é incrementado, enquanto no último caso,

j não é incrementado, pois o novo elemento na posição j ainda não foi processado.

O algoritmo quicksort está fortemente relacionado às árvores de pesquisa binária aleatórias estudadas em Seção 7.1. De fato, se a entrada para quicksort consiste em n elementos distintos, a árvore de recursão de quicksort é uma árvore de pesquisa binária aleatória. Para ver isso, lembre-se de que ao construir uma árvore de pesquisa binária aleatória, a primeira coisa que fazemos é escolher um elemento aleatório x e torná-lo a raiz da árvore. Depois disso, cada elemento será eventualmente comparado com x, com elementos menores indo para a subárvore esquerda e elementos maiores para a direita.

Em quicksort, selecionamos um elemento aleatório x e comparamos imediatamente tudo com x, colocando os elementos menores no início do array e elementos maiores no final do array. Quicksort, em seguida, classifica recursivamente o início do array e o fim do array, enquanto a árvore de pesquisa binária aleatória insere recursivamente elementos menores na subárvore da esquerda dos elementos raiz e maiores na subárvore direita da raiz.

A correspondência acima entre árvores de pesquisa binárias aleatórias e quicksort significa que podemos traduzir Lema 7.1 para uma declaração sobre quicksort:

Lema 11.1. *Quando o quicksort é chamado para ordenar um array contendo os inteiros* $0, \ldots, n-1$, *o número esperado de vezes que o elemento* i *é comparado a um elemento de pivô é no máximo* $H_{i+1} + H_{n-i}$.

Uma pequena soma de números harmônicos nos dá o seguinte teorema sobre o tempo de execução do quicksort:

Teorema 11.2. *Quando o quicksort é chamado para ordenar uma matriz contendo* n *elementos distintos, o número esperado de comparações realizadas é no máximo* $2n \ln n + O(n)$.

Demonstração. Seja T o número de comparações realizadas pelo quicksort ao classificar n elementos distintos. Usando Lema 11.1 e a linearidade de

expectativa, temos:

$$\begin{aligned}
\mathrm{E}[T] &= \sum_{i=0}^{n-1} (H_{i+1} + H_{n-i}) \\
&= 2 \sum_{i=1}^{n} H_i \\
&\leq 2 \sum_{i=1}^{n} H_n \\
&\leq 2n \ln n + 2n = 2n \ln n + O(n)
\end{aligned}$$

□

Teorema 11.3 descreve o caso em que os elementos que estão sendo classificados são todos distintos. Quando o array de entrada, a, contém elementos duplicados, o tempo de execução esperado do quicksort não é pior e pode ser ainda melhor; sempre que um elemento duplicado x é escolhido como um pivô, todas as ocorrências de x são agrupadas e não participam de nenhum dos dois subproblemas.

Teorema 11.3. *O método* `quickSort(a)` *é executado em um tempo esperado de* $O(n \log n)$ *e o número esperado de comparações que executam são no máximo* $2n \ln n + O(n)$.

11.1.3 Heap-sort

O algoritmo *heap-sort* é outro algoritmo de classificação local. O heap-sort usa os heaps binários discutidos em Seção 10.1. Lembre-se de que a estrutura de dados `BinaryHeap` representa um heap usando um único array. O algoritmo de classificação de heap converte o array de entrada a em um heap e, em seguida, extrai repetidamente o valor mínimo.

Mais especificamente, um heap armazena n elementos em um array, a, nas posições do array $a[0], \ldots, a[n-1]$ com o menor valor armazenado na raiz, $a[0]$. Depois de transformar a em um `BinaryHeap`, o algoritmo de classificação de heap troca repetidamente $a[0]$ e $a[n-1]$, diminui n e chama `trickleDown(0)` para que $a[0], \ldots, a[n-2]$ mais uma vez seja uma representação de heap válida. Quando este processo termina (porque $n = 0$), os elementos de a são armazenados em ordem decrescente, então a é

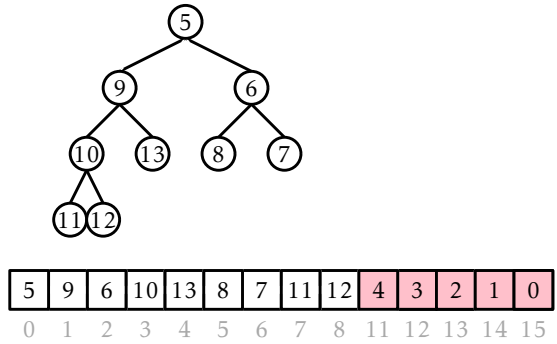


Figura 11.4: Um instantâneo da execução de heapSort(a,c). A parte sombreada da matriz já está classificada. A parte não sombreada é um BinaryHeap. Durante a próxima iteração, o elemento 5 será colocado na posição do array 8.

revertido para obter a ordem final ordenada. [1] A Figura 11.4 mostra um exemplo da execução de heapSort(a,c).

BinaryHeap

```
void sort(array<T> &b) {
  BinaryHeap<T> h(b);
  while (h.n > 1) {
    h.a.swap(--h.n, 0);
    h.trickleDown(0);
  }
  b = h.a;
  b.reverse();
}
```

Uma sub-rotina chave no tipo heap é o construtor para transformar um array não ordenado a em uma pilha. Seria fácil fazer isso em um tempo $O(n \log n)$ chamando repetidamente o método de BinaryHeap add(x), mas podemos fazer melhor usando um algoritmo que execute de baixo para cima. Lembre-se de que, em uma pilha binária, os filhos de a[i] são armazenados nas posições a[2i+1] e a[2i+2]. Isso implica que os elementos $a[\lfloor n/2 \rfloor], \ldots, a[n-1]$ não têm filhos. Em outras palavras, cada um

[1] O algoritmo poderia alternativamente redefinir a função compare(x,y) para que o algoritmo de classificação de heap armazene os elementos diretamente em ordem crescente.

de a[⌊n/2⌋],...,a[n − 1] é um sub-heap de tamanho 1. Agora, trabalhando para trás, podemos chamar `trickleDown(i)` para cada i ∈ {⌊n/2⌋−1,...,0}. Isso funciona, porque no momento em que chamamos `trickleDown(i)`, cada um dos dois filhos de a[i] são a raiz de um sub-heap, então, chamar `trickleDown(i)` faz a[i] a raiz do seu próprio sub-heap.

BinaryHeap

```
BinaryHeap(array<T> &b) : a(0) {
  a = b;
  n = a.length;
  for (int i = n/2-1; i >= 0; i--) {
    trickleDown(i);
  }
}
```

O interessante desta estratégia de baixo para cima é que é mais eficiente do que chamar `add(x)` n vezes. Para ver isso, observe que, para n/2 elementos, não fazemos nenhum trabalho, para n/4 elementos, chamamos `trickleDown(i)` em um sub-heap enraizado em a[i] e cuja altura é um, para n/8 elementos, chamamos `trickleDown(i)` em um subheap cuja altura é dois e assim por diante. Uma vez que o trabalho realizado por `trickleDown(i)` é proporcional à altura do sub-heap enraizado em a[i], isso significa que o trabalho total feito é no máximo

$$\sum_{i=1}^{\log n} O((i-1)n/2^i) \le \sum_{i=1}^{\infty} O(in/2^i) = O(n)\sum_{i=1}^{\infty} i/2^i = O(2n) = O(n) \ .$$

A segunda e última igualdade deriva do reconhecimento que a soma $\sum_{i=1}^{\infty} i/2^i$ é igual, por definição do valor esperado, ao número esperado de vezes que lançamos uma moeda e incluindo a primeira vez que a moeda aparece como cara e aplicando Lema 4.2.

O seguinte teorema descreve o desempenho de `heapSort(a,c)`.

Teorema 11.4. *O método* `heapSort(a,c)` *executa em um tempo* $O(n \log n)$ *e realiza no máximo* $2n\log n + O(n)$ *comparações.*

Demonstração. O algoritmo é executado em três etapas: (1) transformando a em um heap, (2) extraindo repetidamente o elemento mínimo de a e (3) revertendo os elementos em a. Nós apenas argumentamos que o

passo 1 leva um tempo $O(n)$ e executa $O(n)$ comparações. O Passo 3 leva um tempo $O(n)$ e não realiza comparações. O passo 2 executa n chamadas para `trickleDown(0)`. A i-ésima chamada ocorre em um heap de tamanho $n-i$ e executa no máximo $2\log(n-i)$ comparações. Somando isso sobre i dá

$$\sum_{i=0}^{n-i} 2\log(n-i) \leq \sum_{i=0}^{n-i} 2\log n = 2n\log n$$

Adicionando o número de comparações realizadas em cada uma das três etapas completa a prova. □

11.1.4 Um Limite Inferior para classificação baseada em comparação

Nós já vimos três algoritmos de classificação baseados em comparação que executam cada um em um tempo $O(n\log n)$. Agora devemos estar pensando se existem algoritmos mais rápidos. A resposta curta a esta pergunta é não. Se as únicas operações permitidas nos elementos de a são comparações, nenhum algoritmo pode evitar fazer $n\log n$ comparações. Isso não é difícil de provar, mas requer um pouco de imaginação. Em última análise, decorre do fato de que

$$\log(n!) = \log n + \log(n-1) + \cdots + \log(1) = n\log n - O(n) \ .$$

(Provar este fato é deixado para o Exercício 11.10.)

Começaremos concentrando nossa atenção em algoritmos determinísticos como merge-sort e heap-sort e em um valor fixo particular de n. Imagine que esse algoritmo está sendo usado para classificar n elementos distintos. A chave para provar o limite inferior é observar que, para um algoritmo determinístico com um valor fixo de n, o primeiro par de elementos que são comparados é sempre o mesmo. Por exemplo, em `heapSort(a,c)`, quando n é igual, a primeira chamada para `trickleDown(i)` é com $i = n/2-1$ e a primeira comparação é entre os elementos $a[n/2-1]$ e $a[n-1]$.

Uma vez que todos os elementos de entrada são distintos, esta primeira comparação tem apenas dois possíveis resultados. A segunda comparação feita pelo algoritmo pode depender do resultado da primeira comparação. A terceira comparação pode depender dos resultados dos dois primeiros, e assim por diante. Desta forma, qualquer algoritmo de

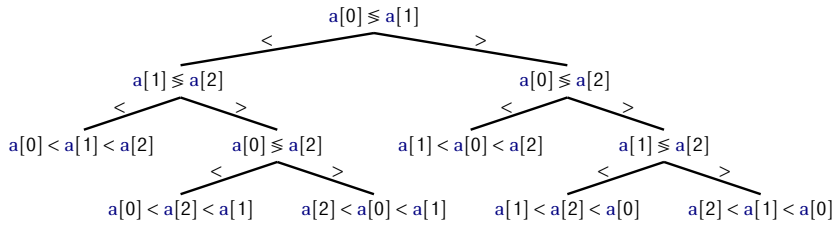


Figura 11.5: Uma árvore de comparação para ordenar um array $a[0], a[1], a[2]$ de tamanho $n = 3$.

classificação determinística baseado em comparação pode ser visto como uma árvore *árvore de comparação* enraizada. Cada nó interno, u, desta árvore é rotulado com um par de índices $u.i$ e $u.j$. Se $a[u.i] < a[u.j]$ o algoritmo prossegue para a subárvore esquerda, caso contrário, ele passa para a subárvore direita. Cada folha w desta árvore é rotulada com uma permutação $w.p[0], \ldots, w.p[n-1]$ de $0, \ldots, n-1$. Essa permutação representa aquele que é necessário para classificar a se a árvore de comparação chegar a esta folha. Isso é,

$$a[w.p[0]] < a[w.p[1]] < \cdots < a[w.p[n-1]] \ .$$

Um exemplo de uma árvore de comparação para um array de tamanho $n = 3$ é mostrado em Figura 11.5.

A árvore de comparação para um algoritmo de classificação nos diz tudo sobre o algoritmo. Ele nos diz exatamente a sequência de comparações que será realizada para qualquer matriz de entrada, a, tendo n elementos distintos e nos diz como o algoritmo irá reordenar a para ordená-lo. Consequentemente, a árvore de comparação deve ter pelo menos $n!$ folhas; se não, então há duas permutações distintas que levam à mesma folha; portanto, o algoritmo não classifica corretamente pelo menos uma dessas permutações.

Por exemplo, a árvore de comparação em Figura 11.6 tem apenas $4 < 3! = 6$ folhas. Inspecionando esta árvore, vemos que os dois arrays de entrada $3,1,2$ e $3,2,1$ ambos levam à folha mais à direita. Na entrada $3,1,2$, esta folha corretamente exibe $a[1] = 1, a[2] = 2, a[0] = 3$. No entanto, na entrada $3,2,1$, este nó incorretamente fornece $a[1] = 2, a[2] = 1, a[0] = 3$.

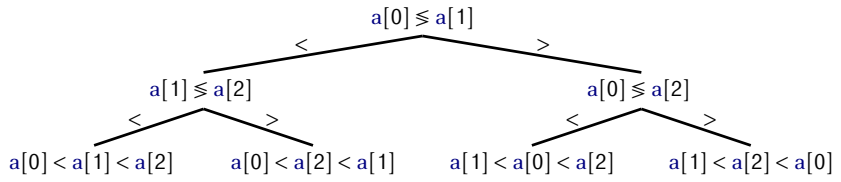


Figura 11.6: Uma árvore de comparação que não classifica corretamente todas as permutações de entrada.

Esta discussão leva ao limite inferior primário para algoritmos baseados em comparação.

Teorema 11.5. *Para qualquer algoritmo de classificação baseado em comparação determinística $\mathcal{A}$ e qualquer inteiro $n \geq 1$, existe um array de entrada a de comprimento n tal que $\mathcal{A}$ executa em menos $\log(n!) = n \log n - O(n)$ comparações ao classificar a.*

Demonstração. Na discussão anterior, a árvore de comparação definida por $\mathcal{A}$ deve ter pelo menos $n!$ folhas. Uma prova indutiva fácil mostra que qualquer árvore binária com k folhas tem uma altura de pelo menos $\log k$. Portanto, a árvore de comparação para $\mathcal{A}$ tem uma folha, w, com uma profundidade de pelo menos $\log(n!)$ e existe um array de entrada a que leva a esta folha. O array de entrada a é uma entrada para a qual $\mathcal{A}$ faz pelo menos $\log(n!)$ comparações. □

Teorema 11.5 trata de algoritmos determinísticos, como o merge-sort e o heap-sort, mas não nos conta nada sobre algoritmos randomizados como quicksort. Poderia um algoritmo randomizado vencer o limite inferior $\log(n!)$ no número de comparações? A resposta, novamente, é não. Novamente, a maneira de provar isso é pensar de forma diferente sobre o que é um algoritmo randomizado.

Na discussão a seguir, assumiremos que nossas árvores de decisão foram "limpas"da seguinte maneira: qualquer nó que não pode ser alcançado por algum array de entrada a é removido. Esta limpeza implica que a árvore tem exatamente $n!$ folhas. Tem pelo menos $n!$ folhas porque, caso contrário, não conseguiria ordenar corretamente. Tem no máximo $n!$ folhas desde que cada uma das possíveis $n!$ permutações de n elementos

distintos seguem exatamente uma raiz para o caminho da folha na árvore de decisão.

Podemos pensar em um algoritmo de classificação aleatorizado, $\mathcal{R}$, como um algoritmo determinista que leva duas entradas: o array de entrada a que deve ser ordenado e uma sequência longa $b = b_1, b_2, b_3, \ldots, b_m$ de números reais aleatórios no intervalo $[0, 1]$. Os números aleatórios fornecem a randomização para o algoritmo. Quando o algoritmo quer jogar uma moeda ou fazer uma escolha aleatória, ele faz isso usando algum elemento de b. Por exemplo, para calcular o índice do primeiro pivô no quicksort, o algoritmo pode usar a fórmula $\lfloor nb_1 \rfloor$.

Agora, note que se nós fixamos b em alguma sequência particular $\hat{b}$, então $mathcal R$ se torna um algoritmo de classificação determinista, $\mathcal{R}(\hat{b})$, que tem uma árvore de comparação associada, $\mathcal{T}(\hat{b})$. Em seguida, note que se selecionarmos a para ser uma permutação aleatória de $\{1, \ldots, \mathtt{n}\}$, então isso é equivalente a selecionar uma folha aleatória, w, a partir de n! folhas de $\mathcal{T}(\hat{b})$.

Exercício 11.12 pede para provar isso, se selecionarmos uma folha aleatória de qualquer árvore binária com k folhas, então a profundidade esperada dessa folha é pelo menos $\log k$. Portanto, o número esperado de comparações realizadas pelo algoritmo (determinístico) $\mathcal{R}(\hat{b})$ quando é dada um array de entrada contendo uma permutação aleatória de $\{1, \ldots, n\}$ é pelo menos $\log(\mathtt{n}!)$. Finalmente, note que isso é verdade para todas as opções de $\hat{b}$, portanto, é válido para $\mathcal{R}$. Isso completa a prova do limite inferior para algoritmos randomizados.

Teorema 11.6. *Para qualquer inteiro $n \geq 1$ e qualquer algoritmo de classificação baseado em comparação (determinístico ou randomizado) $\mathcal{A}$, o número esperado de comparações feito por $\mathcal{A}$ ao classificar uma permutação aleatória de $\{1, \ldots, n\}$ é de pelo menos* $\log(\mathtt{n}!) = \mathtt{n}\log\mathtt{n} - O(\mathtt{n})$.

11.2 Ordenação por Contagem e Ordenação Radix

Nesta seção, estudamos dois algoritmos de classificação que não são baseados em comparação. Especializados para classificar inteiros pequenos, esses algoritmos evitam os limites inferiores de Teorema 11.5 usando (partes de) os elementos em a como índices em um array. Considere uma

declaração da forma

$$c[a[i]] = 1 \ .$$

Esta declaração é executada em tempo constante, mas tem `c.length` possíveis resultados diferentes, dependendo do valor de `a[i]`. Isso significa que a execução de um algoritmo que faz tal afirmação não pode ser modelada como uma árvore binária. Em última análise, essa é a razão pela qual os algoritmos nesta seção podem classificar mais rapidamente do que os algoritmos baseados em comparação.

11.2.1 Ordenação por Contagem

Suponhamos que tenhamos um array de entrada `a` consistindo de `n` inteiros, cada um no intervalo $0,\ldots,k-1$. O *count-sort* algoritmo classifica `a` usando um array auxiliar `c` de contadores. Ele produz uma versão ordenada de `a` como um array auxiliar `b`.

A ideia por trás do tipo de contagem é simples: para cada $i \in \{0,\ldots,k-1\}$, conte o número de ocorrências de `i` em `a` e armazene isso em `c[i]`. Agora, após a classificação, a saída parecerá `c[0]` ocorrências de 0, seguido de `c[1]` ocorrências de 1, seguido de `c[2]` ocorrências de 2, ..., seguido de `c[k-1]` ocorrências de `k-1`. O código que faz isso é bem inteligente, e sua execução está ilustrada em Figura 11.7:

Algorithms

```
void countingSort(array<int> &a, int k) {
  array<int> c(k, 0);
  for (int i = 0; i < a.length; i++)
    c[a[i]]++;
  for (int i = 1; i < k; i++)
    c[i] += c[i-1];
  array<int> b(a.length);
  for (int i = a.length-1; i >= 0; i--)
    b[--c[a[i]]] = a[i];
  a = b;
}
```

O primeiro loop `for` neste código define cada contador `c[i]` para que ele conte o número de ocorrências de `i` em `a`. Ao usar os valores de `a` como índices, esses contadores podem ser computados em um tempo $O(n)$ com

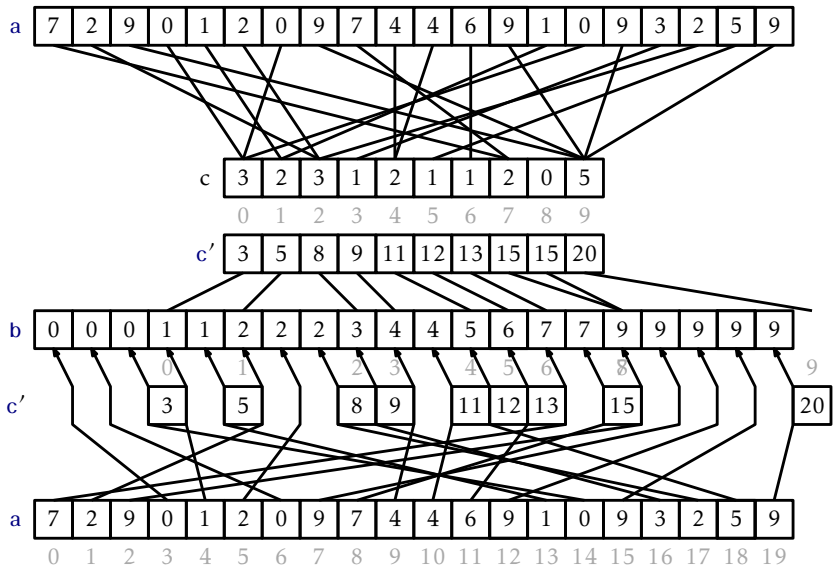


Figura 11.7: A operação do tipo de contagem em um array de comprimento $n = 20$ que armazena números inteiros $0, \ldots, k - 1 = 9$.

um único loop *for*. Neste ponto, poderíamos usar c para preencher o array de saída b diretamente. No entanto, isso não funcionaria se os elementos de a tiverem dados associados. Portanto, gastamos um pouco de esforço extra para copiar os elementos de a em b.

O próximo loop `for`, que leva um tempo $O(k)$, calcula uma soma dos contadores para que $c[i]$ se torne o número de elementos em a que são menores ou iguais a i. Em particular, para cada $i \in \{0,\ldots,k-1\}$, o array de saída, b, terá

$$b[c[i-1]] = b[c[i-1]+1] = \cdots = b[c[i]-1] = i \ .$$

Finalmente, o algoritmo varre a de trás para frente para colocar seus elementos, em ordem, em um array de saída b. Ao varrer, o elemento $a[i] = j$ é colocado na posição $b[c[j]-1]$ e o valor $c[j]$ é decrementado.

Teorema 11.7. *O método* `countingSort(a,k)` *pode classificar um array* a *contendo* n *inteiros no conjunto* $\{0,\ldots,k-1\}$ *em um tempo* $O(n+k)$.

O algoritmo de classificação de contagem tem a boa propriedade de ser *estável*; preserva a ordem relativa de elementos iguais. Se dois elementos $a[i]$ e $a[j]$ tiverem o mesmo valor e $i < j$ então $a[i]$ aparecerá antes $a[j]$ em b. Isso será útil na próxima seção.

11.2.2 Ordenação Radix

A ordenação por contagem é muito eficiente para classificar um array de inteiros quando o comprimento, n, do array não é muito menor do que o valor máximo, $k-1$, que aparece no array. O algoritmo *radix-sort* , que descrevemos agora, usa várias passagens de ordenação por contagem para permitir uma gama muito maior de valores máximos.

A ordenação por radix classifica w-bit inteiros usando w/d passos de ordenação por contagem para classificar esses números inteiros de d bits de cada vez. [2] Mais precisamente, a ordenação por radix primeiro classifica os inteiros por seus d bits menos significativos, então os próximos d bits significativos e assim, até que, na última passagem, os números inteiros sejam classificados por seus d bits mais significativos.

[2] Assumimos que d divide w, caso contrário nós sempre podemos aumentar w para $d\lceil w/d \rceil$.

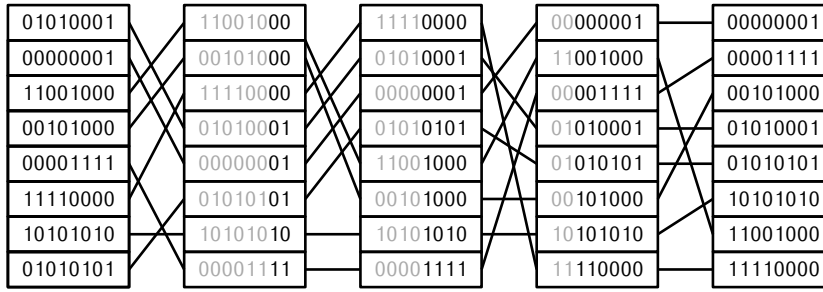


Figura 11.8: Usando radixsort para ordenar inteiros de $w = 8$-bits usando 4 passagens da ordenação por contagem com inteiros de $d = 2$-bits.

Algorithms

```
void radixSort(array<int> &a) {
  int d = 8, w = 32;
  for (int p = 0; p < w/d; p++) {
    array<int> c(1<<d, 0);
    // the next three for loops implement counting-sort
    array<int> b(a.length);
    for (int i = 0; i < a.length; i++)
      c[(a[i] >> d*p)&((1<<d)-1)]++;
    for (int i = 1; i < 1<<d; i++)
      c[i] += c[i-1];
    for (int i = a.length-1; i >= 0; i--)
      b[--c[(a[i] >> d*p)&((1<<d)-1)]] = a[i];
    a = b;
  }
}
```

(Neste código, a expressão $(a[i] >> d*p)\&((1 << d) - 1)$ extrai o inteiro cuja representação binária é dado pelos bits $(p+1)d-1,\ldots,pd$ de $a[i]$.) Um exemplo dos passos deste algoritmo é mostrado na Figura 11.8.

Este algoritmo notável classifica corretamente porque o tipo de contagem é um algoritmo de classificação estável. Se $x < y$ para dois elementos de a, e o bit mais significativo em que x difere de y tenha índice r, então x será colocado antes de y durante o passo $\lfloor r/d \rfloor$ e as passagens subsequentes não alterarão a ordem relativa de x e y.

Radix-sort executa w/d passos de ordenação por contagem. Cada pas-

sagem requer um tempo $O(n + 2^d)$. Portanto, o desempenho da ordenação por radix é dado pelo seguinte teorema.

Teorema 11.8. *Para qualquer inteiro* $d > 0$, *o método* `radixSort(a,k)` *pode classificar um array* `a` *contendo* `n` *inteiros de* `w`*-bits em um tempo* $O((w/d)(n + 2^d))$.

Se pensarmos, em vez disso, que os elementos do array estão no intervalo $\{0,\ldots,n^c - 1\}$ e pegarmos $d = \lceil \log n \rceil$ obtemos a seguinte versão de Teorema 11.8.

Corolário 11.1. *O método* `radixSort(a,k)` *pode classificar um array* `a` *contendo* `n` *valores inteiros na faixa* $\{0,\ldots,n^c - 1\}$ *em um tempo* $O(cn)$.

11.3 Discussão e Exercícios

A classificação é *o* problema algorítmico fundamental na ciência da computação, e tem uma longa história. Knuth [48] atribui o algoritmo de classificação de mesclagem a von Neumann (1945). Quicksort é devido a Hoare [39]. O algoritmo de classificação de heap original é devido a Williams [76], mas a versão apresentada aqui (em que o heap é construído de baixo para cima em um tempo $O(n)$) é devido a Floyd [28]. Os limites inferiores para a classificação baseada em comparação parecem ser folclore. A tabela a seguir resume o desempenho desses algoritmos baseados em comparação:

	comparações		no local
Merge-sort	$n \log n$	pior caso	Não
Quicksort	$1.38n \log n + O(n)$	esperado	Sim
Heap-sort	$2n \log n + O(n)$	pior caso	Sim

Cada um desses algoritmos baseados em comparação tem suas vantagens e desvantagens. Merge-sort tem o menor número de comparações e não depende da randomização. Infelizmente, ele usa um array auxiliar durante a fase de mesclagem. Alocar este array pode ser caro e é um ponto potencial de falha se a memória for limitada. Quicksort é um algoritmo *no local* e é o segundo próximo em termos de número de comparações,

mas é randomizado, portanto, este tempo de execução nem sempre é garantido. Heap-sort faz o maior número de comparações, mas é no local e determinístico.

Existe uma configuração na qual a ordenação por merge é um vencedor claro; isso ocorre ao ordenar uma lista encadeada. Nesse caso, o array auxiliar não é necessário; duas listas encadeadas ordenadas são facilmente incorporadas em uma única lista ordenada por manipulação de ponteiro (veja Exercício 11.2).

Os algoritmos de classificação de contagem e classificação de radix aqui descritos são devidos a Seward [66, Seção 2.4.6]. No entanto, variantes de radix-sort foram utilizadas desde a década de 1920 para classificar os cartões de perfuração usando máquinas de classificação de cartão perfurado. Essas máquinas podem classificar uma pilha de cartões em duas pilhas com base na existência (ou não) de um buraco em um local específico no cartão. Repetir este processo para diferentes locais de furos dá uma implementação de tipo radix.

Finalmente, observamos que a ordenação por contagem e a ordenação por radix podem ser usadas para classificar outros tipos de números além de números inteiros não negativos. Modificações diretas da ordenação por contagem podem classificar inteiros, em qualquer intervalo $\{a, \ldots, b\}$, em $O(n + b - a)$. Da mesma forma, a ordenação por radix pode classificar inteiros no mesmo intervalo em um tempo $O(n(\log_n(ba))$. Finalmente, esses dois algoritmos também podem ser usados para classificar números de ponto flutuante no Formato de ponto flutuante IEEE 754. Isso ocorre porque o formato IEEE foi projetado para permitir a comparação de dois números de ponto flutuante comparando seus valores como se fossem inteiros em uma representação binária de sinal-magnitude [2].

Exercício 11.1. Ilustre a execução de merge-sort e heap-sort em um array de entrada contendo $1, 7, 4, 6, 2, 8, 3, 5$. Dê uma amostra que ilustre uma possível execução de quicksort no mesmo array.

Exercício 11.2. Implementar uma versão do algoritmo de classificação por merge que ordena uma `DLList` sem usar um array auxiliar. (Veja Exercício 3.13.)

Exercício 11.3. Algumas implementações de `quickSort(a, i, n, c)` sempre

usam a[i] como um pivô. Dê um exemplo de um array de entrada de comprimento n no qual tal implementação executaria $\binom{n}{2}$ comparações.

Exercício 11.4. Algumas implementações de `quickSort(a, i, n, c)` sempre usam a[i + n/2] como um pivô. Forneça um exemplo de um array de entrada de comprimento n na qual tal implementação executaria $\binom{n}{2}$ comparações.

Exercício 11.5. Mostre que, para qualquer implementação de `quickSort(a, i, n, c)` que escolha um pivô de forma determinística, sem primeiro olhar para qualquer valor em a[i], ..., a[i + n − 1], existe um array de entrada de comprimento n que faz com que esta implementação realize $\binom{n}{2}$ comparações.

Exercício 11.6. Crie um `Comparador, c`, que você poderia passar como um argumento para `quickSort(a, i, n, c)` e isso faria que o quicksort execute $\binom{n}{2}$ comparações. (Sugestão: o seu comparador na verdade não precisa analisar os valores que estão sendo comparados).

Exercício 11.7. Analise o número esperado de comparações feitas pelo Quicksort um pouco mais cuidadosamente do que a prova de Teorema 11.3. Em particular, mostre que o número esperado de comparações é $2nH_n - n + H_n$.

Exercício 11.8. Descreva um array de entrada que faz com que a ordenação heap realize pelo menos $2n \log n - O(n)$ comparações. Justifique sua resposta.

Exercício 11.9. Encontre outro par de permutações de 1, 2, 3 que não sejam ordenados corretamente pela árvore de comparação em Figura 11.6.

Exercício 11.10. Prove que $\log n! = n \log n - O(n)$.

Exercício 11.11. Prove que uma árvore binária com k folhas tem altura de pelo menos $\log k$.

Exercício 11.12. Prove que, se escolhermos uma folha aleatória de uma árvore binária com k folhas, a altura esperada desta folha é de pelo menos $\log k$.

Exercício 11.13. A implementação de `radixSort(a, k)` fornecida aqui funciona quando o array de entrada, a contém apenas inteiros sem sinal. Escreva uma versão que funcione corretamente para inteiros sinalizados.

Capítulo 12

Grafos

Neste capítulo, estudamos duas representações de grafos e algoritmos básicos que usam essas representações.

Matematicamente, um *grafo (direcionado)* é um par $G = (V, A)$ onde V é um conjunto de *vértices* e A é um conjunto de pares ordenados de vértices chamados *arestas*. Uma aresta (i, j) é *direcionado* de i para j; i é chamado de *fonte* da aresta e j é chamado de *alvo*. Um *caminho* em G é uma sequência de vértices $v_0, \ldots, v_k$ tal que, para cada $i \in \{1, \ldots, k\}$, a aresta (v_{i-1}, v_i) está em A. Um caminho $v_0, \ldots, v_k$ é um *ciclo* se, além disso, a aresta (v_k, v_0) está em A. Um caminho (ou ciclo) é *simples* se todos os seus vértices forem únicos. Se houver um caminho de algum vértice v_i para algum vértice v_j então nós dizemos que v_j está ao *alcance* de v_i. Um exemplo de grafo é mostrado em Figura 12.1.

Devido à sua capacidade de modelar tantos fenômenos, os grafos têm um enorme número de aplicações. Existem muitos exemplos óbvios. Redes de computadores podem ser modeladas como grafos, com vértices correspondendo a computadores e arestas correspondendo a links de comunicação (direcionados) entre esses computadores. As ruas da cidade podem ser modeladas como grafos, com vértices representando interseções e bordas representando ruas que unem interseções consecutivas.

Exemplos menos óbvios ocorrem assim que percebemos que os grafos podem modelar quaisquer relações de pares dentro de um conjunto. Por exemplo, em um ambiente universitário, podemos ter um *grafo de conflito* de horário cujos vértices representam cursos oferecidos na universidade e nos quais a aresta (i, j) está presente se e somente se houver pelo menos

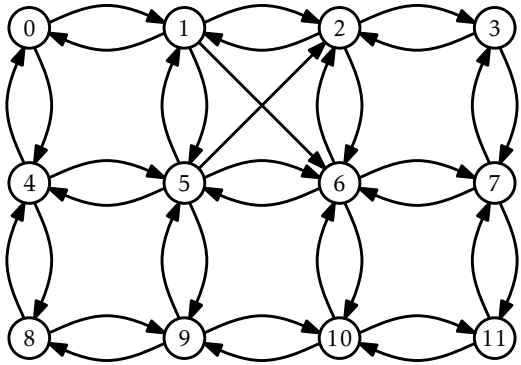


Figura 12.1: Um grafo com doze vértices. Os vértices são desenhados como círculos numerados e as arestas são desenhadas como curvas pontiagudas apontando da origem ao destino.

um aluno que está cursando as aulas `i` e a classe `j`. Assim, uma aresta indica que a prova da classe `i` não deve ser agendado ao mesmo tempo que a prova da classe `j`.

Ao longo desta seção, usaremos `n` para denotar o número de vértices de G e `m` para denotar o número de arestas de G. Ou seja, $\texttt{n} = |V|$ e $\texttt{m} = |A|$. Além disso, assumiremos que $V = \{0, \ldots, \texttt{n} - 1\}$. Quaisquer outros dados que gostaríamos de associar aos elementos de V podem ser armazenados em um array de comprimento `n`.

Algumas operações típicas realizadas em grafos são:

- `addEdge(i, j)`: Adicione a aresta `(i, j)` a A.
- `removeEdge(i, j)`: Remova a aresta `(i, j)` de A.
- `hasEdge(i, j)`: Checa se a aresta $(\texttt{i}, \texttt{j}) \in A$
- `outEdges(i)`: Retorna uma `List` de todos os inteiros `j` tal que $(\texttt{i}, \texttt{j}) \in A$
- `inEdges(i)`: Retorna uma `List` de todos os inteiros `j` tal que $(\texttt{j}, \texttt{i}) \in A$

Observe que essas operações não são terrivelmente difíceis de implementar com eficiência. Por exemplo, as três primeiras operações podem

ser implementadas diretamente usando um `USet`, para que possam ser implementadas em tempo constante esperado usando as tabelas de hash discutidas no Capítulo 5. As duas últimas operações podem ser implementadas em tempo constante armazenando, para cada vértice, uma lista de seus vértices adjacentes.

No entanto, diferentes aplicativos de grafos têm diferentes requisitos de desempenho para essas operações e, idealmente, podemos usar a implementação mais simples que satisfaça todos os requisitos do aplicativo. Por esse motivo, discutimos duas grandes categorias de representações gráficas.

12.1 `AdjacencyMatrix`: Representando um grafo por uma matriz

Uma *matriz adjacência* é uma forma de representar um grafo de `n` vértices $G = (V, A)$ por uma matriz `n` × `n`, `a`, cujas entradas são valores booleanos.

`AdjacencyMatrix`

```
int n;
bool **a;
```

A entrada da matriz `a[i][j]` é definido como

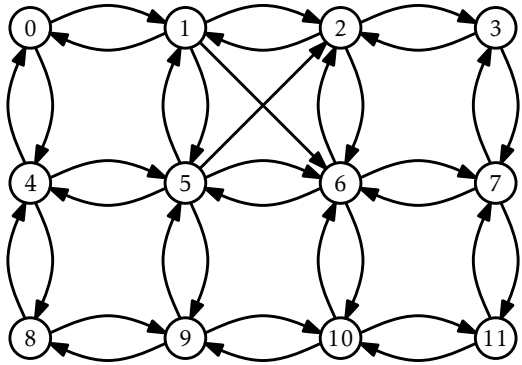
$$\texttt{a[i][j]} = \begin{cases} \texttt{true} & \text{se } (\texttt{i}, \texttt{j}) \in A \\ \texttt{false} & \text{caso contrário} \end{cases}$$

A matriz de adjacência para o grafo da Figura 12.1 é mostrada em Figura 12.2.

Nesta representação, as operações `addEdge(i, j)`, `removeEdge(i, j)` e `hasEdge(i, j)` envolvem apenas definir ou ler a entrada da matriz `a[i][j]`:

`AdjacencyMatrix`

```
void addEdge(int i, int j) {
  a[i][j] = true;
}
void removeEdge(int i, int j) {
  a[i][j] = false;
}
```



	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	0	0	1	0	0	0	0	0	0	0
1	1	0	1	0	0	1	1	0	0	0	0	0
2	1	0	0	1	0	0	1	0	0	0	0	0
3	0	0	1	0	0	0	0	1	0	0	0	0
4	1	0	0	0	0	1	0	0	1	0	0	0
5	0	1	1	0	1	0	1	0	0	1	0	0
6	0	0	1	0	0	1	0	1	0	0	1	0
7	0	0	0	1	0	0	1	0	0	0	0	1
8	0	0	0	0	1	0	0	0	0	1	0	0
9	0	0	0	0	0	1	0	0	1	0	1	0
10	0	0	0	0	0	0	1	0	0	1	0	1
11	0	0	0	0	0	0	0	1	0	0	1	0

Figura 12.2: Um grafo e sua matriz de adjacência.

```
bool hasEdge(int i, int j) {
  return a[i][j];
}
```

Essas operações claramente levam um tempo constante por operação.

O desempenho da matriz de adjacência é insatisfatório nas operações `outEdges(i)` e `inEdges(i)`. Para implementá-los, devemos examinar todas as entradas `n` na linha ou coluna correspondente de `a` e reunir todos os índices, `j`, onde `a[i][j]`, respectivamente `a[j][i]`, é verdade.

AdjacencyMatrix

```
void outEdges(int i, List &edges) {
```

```
  for (int j = 0; j < n; j++)
    if (a[i][j]) edges.add(j);
}
void inEdges(int i, List &edges) {
  for (int j = 0; j < n; j++)
    if (a[j][i]) edges.add(j);
}
```

Essas operações claramente levam tempo por operação $O(n)$.

Outra desvantagem da representação da matriz de adjacência é que ela é grande. Ele armazena uma matriz booleana $n \times n$, então requer pelo menos n^2 bits de memória. A implementação aqui usa uma matriz de valores `bool`, então ela realmente usa na ordem de n^2 bytes de memória. Uma implementação mais cuidadosa, que empacote `w` valores booleanos em cada palavra de memória, poderia reduzir esse uso de espaço para $O(n^2/w)$ palavras de memória.

Teorema 12.1. *A estrutura de dados* `AdjacencyMatrix` *implementa a interface* `Graph`. *Uma* `AdjacencyMatrix` *suporta as operações*

- `addEdge(i, j)`, `removeEdge(i, j)`, *e* `hasEdge(i, j)` *em tempo constante por operação; e*
- `inEdges(i)`, *e* `outEdges(i)` *em um tempo* $O(n)$ *por operação.*

O espaço usado por uma `AdjacencyMatrix` *é* $O(n^2)$.

Apesar de seus altos requisitos de memória e baixo desempenho das operações `inEdges(i)` e `outEdges(i)`, uma `AdjacencyMatrix` ainda pode ser útil para alguns aplicativos. Em particular, quando o grafo G é *denso*, ou seja, tem quase n^2 arestas, então um uso de memória de n^2 pode ser aceitável.

A estrutura de dados `AdjacencyMatrix` também é comumente usada porque as operações algébricas na matriz `a` podem ser usadas para calcular com eficiência as propriedades do grafo G. Este é um tópico para um curso de algoritmos, mas apontamos uma dessas propriedades aqui: se tratarmos as entradas de `a` como inteiros (1 para `verdadeiro` e 0 para `falso`) e multiplicarmos `a` por si mesmo usando a matriz multiplicação

então obtemos a matriz a^2. Lembre-se, a partir da definição de multiplicação de matrizes, que

$$a^2[i][j] = \sum_{k=0}^{n-1} a[i][k] \cdot a[k][j] \ .$$

Interpretando essa soma em termos do grafo G, essa fórmula conta o número de vértices, `k`, de forma que G contenha ambas as arestas (`i`,`k`) e (`k`, `j`). Ou seja, ele conta o número de caminhos de `i` a `j` (por meio dos vértices intermediários, `k`) cujo comprimento é exatamente dois. Essa observação é a base de um algoritmo que calcula os caminhos mais curtos entre todos os pares de vértices em G usando apenas $O(\log n)$ multiplicações de matriz.

12.2 AdjacencyLists: Um grafo como uma coleção de listas

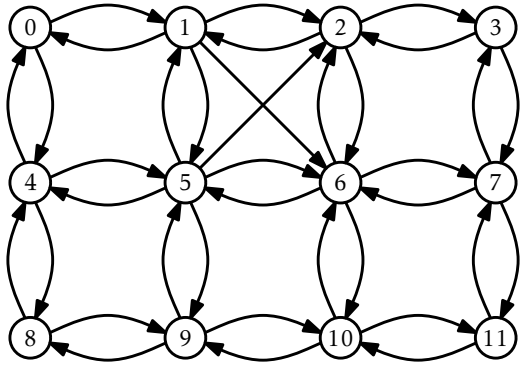
As representações com *listas de adjacências* de grafos têm uma abordagem mais centrada no vértice. Existem muitas implementações possíveis de listas de adjacência. Nesta seção, apresentamos uma simples. No final da seção, discutimos diferentes possibilidades. Em uma representação de lista de adjacência, o grafo $G = (V, A)$ é representado como um array, `adj`, de listas. A lista `adj[i]` contém uma lista de todos os vértices adjacentes ao vértice `i`. Ou seja, ele contém todos os índices `j` tais que (`i`, `j`) $\in A$.

AdjacencyLists
```
int n;
List *adj;
```

(Um exemplo é mostrado em Figura 12.3.) Nesta implementação particular, representamos cada lista em `adj` como uma subclasse de `ArrayStack`, porque gostaríamos de acesso de tempo constante por posição. Outras opções também são possíveis. Especificamente, poderíamos ter implementado `adj` como uma `DLList`.

A operação `addEdge(i, j)` apenas acrescenta o valor `j` à lista `adj[i]`:

AdjacencyLists
```
void addEdge(int i, int j) {
```



0	1	2	3	4	5	6	7	8	9	10	11
1	0	1	2	0	1	5	6	4	8	9	10
4	2	3	7	5	2	2	3	9	5	6	7
	6	6		8	6	7	11		10	11	
	5				9	10					
					4						

Figura 12.3: Um grafo e suas listas adjacentes

```
  adj[i].add(j);
}
```

Isso leva um tempo constante.

A operação `removeEdge(i, j)` pesquisa a lista `adj[i]` até encontrar `j` e depois a remove:

AdjacencyLists

```
void removeEdge(int i, int j) {
  for (int k = 0; k < adj[i].size(); k++) {
    if (adj[i].get(k) == j) {
      adj[i].remove(k);
      return;
    }
  }
}
```

Isso leva um tempo $O(\deg(i))$, onde $\deg(i)$ (o *grau* de `i`) conta o número de arestas em A que têm `i` como fonte.

A operação `hasEdge(i, j)` é semelhante; ela pesquisa na lista `adj[i]` até encontrar `j` (e retorna verdadeiro), ou chega ao final da lista (e retorna falso):

AdjacencyLists

```
bool hasEdge(int i, int j) {
  return adj[i].contains(j);
}
```

Isso também leva tempo $O(\deg(i))$.

A operação `outEdges(i)` é muito simples; copia os valores em `adj[i]` para a lista de saída:

AdjacencyLists

```
void outEdges(int i, ListT &edges) {
  for (int k = 0; k < adj[i].size(); k++)
    edges.add(adj[i].get(k));
}
```

Isso claramente leva um tempo $O(\deg(i))$.

A operação `inEdges(i)` é muito mais trabalhosa. Ele examina cada vértice j verificando se a aresta `(i, j)` existe e, em caso afirmativo, adicionando `j` à lista de saída:

AdjacencyLists

```
void inEdges(int i, ListT &edges) {
  for (int j = 0; j < n; j++)
    if (adj[j].contains(i))  edges.add(j);
}
```

Esta operação é muito lenta. Ele faz a varredura da lista de adjacências de cada vértice, portanto, leva tempo $O(n+m)$.

O teorema a seguir resume o desempenho da estrutura de dados acima:

Teorema 12.2. *A estrutura de dados* `AdjacencyLists` *implementa a interface* `Graph`. *Uma* `AdjacencyLists` *suporta as operações*

- `addEdge(i, j)` *em tempo constante por operação;*
- `removeEdge(i, j)` *e* `hasEdge(i, j)` *em tempo* $O(\deg(i))$ *por operação;*
- `outEdges(i)` *em tempo* $O(\deg(i))$ *por operação; e*
- `inEdges(i)` *em tempo* $O(n+m)$ *por operação.*

O espaço usado por uma `AdjacencyLists` *é* $O(n+m)$.

Conforme mencionado anteriormente, existem muitas opções diferentes a serem feitas ao implementar um grafo como uma lista de adjacências. Algumas perguntas que surgem incluem:

- Que tipo de coleção deve ser usado para armazenar cada elemento de `adj`? Pode-se usar uma lista baseada em array, uma lista encadeada ou até mesmo uma tabela de hash.
- Deve haver uma segunda lista de adjacência, `inadj`, que armazena, para cada `i`, a lista de vértices, `j`, tal que $(j, i) \in A$? Isso pode reduzir bastante o tempo de execução da operação `inEdges(i)`, mas requer um pouco mais de trabalho ao adicionar ou remover arestas.
- A entrada para a aresta `(i, j)` em `adj[i]` deve ser encadeada por uma referência à entrada correspondente em `inadj[j]`?
- As arestas devem ser objetos de primeira classe com seus próprios dados associados? Desta forma, `adj` conteria listas de arestas em vez de listas de vértices (inteiros).

A maioria dessas questões se resume a uma troca entre complexidade (e espaço) de implementação e recursos de desempenho da implementação.

12.3 Percurso em Grafos

Nesta seção, apresentamos dois algoritmos para explorar um grafo, começando em um de seus vértices, `i`, e encontrando todos os vértices que são acessíveis a partir de `i`. Ambos os algoritmos são mais adequados para grafos representados usando uma representação de lista de adjacência. Portanto, ao analisar esses algoritmos, assumiremos que a representação subjacente é uma `AdjacencyLists`.

12.3.1 Busca em Largura

O algoritmo *busca em largura* começa em um vértice `i` e visita, primeiro os vizinhos de `i`, depois os vizinhos dos vizinhos de `i`, então os vizinhos dos vizinhos dos vizinhos de `i` e assim por diante.

Este algoritmo é uma generalização do algoritmo de percurso em largura para árvores binárias (Seção 6.1.2) e é muito semelhante; ele usa uma fila, `q`, que inicialmente contém apenas `i`. Em seguida, extrai repetidamente um elemento de `q` e adiciona seus vizinhos a `q`, desde que esses vizinhos nunca tenham estado em `q` antes. A única grande diferença entre o algoritmo de busca por largura para grafos e o algoritmo para árvores é que o algoritmo para grafos deve garantir que não adiciona o mesmo vértice a `q` mais de uma vez. Ele faz isso usando um array booleano auxiliar, `seen`, que rastreia quais vértices já foram descobertos.

Algorithms

```
void bfs(Graph &g, int r) {
  bool *seen = new bool[g.nVertices()];
  SLList<int> q;
  q.add(r);
  seen[r] = true;
  while (q.size() > 0) {
    int i = q.remove();
    ArrayStack<int> edges;
    g.outEdges(i, edges);
    for (int k = 0; k < edges.size(); k++) {
```

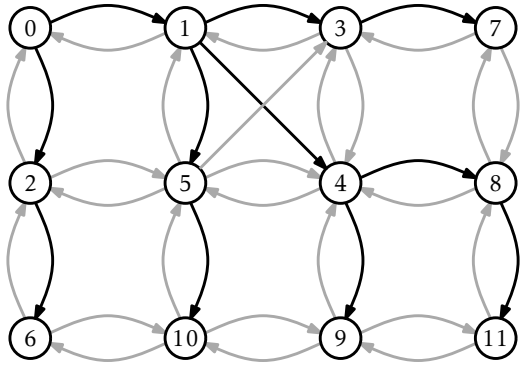


Figura 12.4: Um exemplo de pesquisa em amplitude começando no nó 0. Os nós são rotulados com a ordem em que são adicionados a `q`. As arestas que resultam na adição de nós a `q` são desenhadas em preto, outras arestas são desenhadas em cinza.

```
      int j = edges.get(k);
      if (!seen[j]) {
        q.add(j);
        seen[j] = true;
      }
    }
  }
  delete[] seen;
}
```

Um exemplo de execução de `bfs(g,0)` no grafo de Figura 12.1 é mostrado em Figura 12.4. Diferentes execuções são possíveis, dependendo da ordem das listas de adjacência; Figura 12.4 usa as listas de adjacência em Figura 12.3.

Analisar o tempo de execução da rotina `bfs(g,i)` é bastante simples. O uso do array `seen` garante que nenhum vértice seja adicionado a `q` mais de uma vez. Adicionar (e posteriormente remover) cada vértice de `q` leva um tempo constante por vértice para um total de $O(n)$ tempo. Uma vez que cada vértice é processado pelo laço interno no máximo uma vez, cada lista de adjacência é processada no máximo uma vez, então cada aresta de G é processada no máximo uma vez. Esse processamento, que é feito no

loop interno, leva um tempo constante por iteração, para um tempo total de $O(m)$. Portanto, todo o algoritmo é executado em tempo $O(n+m)$.

O teorema a seguir resume o desempenho do algoritmo `bfs(g,r)`.

Teorema 12.3. *Quando dado como entrada um* `Grafo`, `g`, *que é implementado usando a estrutura de dados* `AdjacencyLists`, *o algoritmo* `bfs(g,r)` *é executado em tempo* $O(n+m)$.

Uma travessia em largura tem algumas propriedades muito especiais. Chamar `bfs(g,r)` eventualmente enfileirará (e eventualmente removerá da fila) cada vértice `j` de forma que haja um caminho direcionado de `r` para `j`. Além disso, os vértices na distância 0 de `r` (o próprio `r`) entrarão `q` antes dos vértices na distância 1, que entrarão `q` antes dos vértices na distância 2, e assim por diante. Assim, o método `bfs(g,r)` visita vértices em ordem crescente de distância de `r` e vértices que não podem ser alcançados a partir de `r` nunca são visitados.

Uma aplicação particularmente útil do algoritmo de busca por largura é, portanto, na computação de caminhos mais curtos. Para calcular o caminho mais curto de `r` para todos os outros vértices, usamos uma variante de `bfs(g,r)` que usa uma matriz auxiliar, `p`, de comprimento `n`. Quando um novo vértice `j` é adicionado a `q`, definimos `p[j] = i`. Desta forma, `p[j]` se torna o penúltimo nó em um caminho mais curto de `r` a `j`. Repetindo isso, tomando `p[p[j]]`, `p[p[p[j]]]`, e assim por diante, podemos reconstruir o (reverso de) um caminho mais curto de `r` para `j`.

12.3.2 Pesquisa em profundidade

O algoritmo *Pesquisa em profundidade* é semelhante ao algoritmo padrão para percorrer árvores binárias; ele primeiro explora completamente uma subárvore antes de retornar ao nó atual e, em seguida, explorar a outra subárvore. Outra maneira de pensar na pesquisa em profundidade é dizer que ela é semelhante à pesquisa em largura, exceto que usa uma pilha em vez de uma fila.

Durante a execução do algoritmo de pesquisa em profundidade, cada vértice, `i`, recebe uma cor, `c[i]`: `branco` se nunca vimos o vértice antes, `cinza` se estivermos atualmente visitando aquele vértice, e `preto` se terminarmos de visitar aquele vértice. A maneira mais fácil de pensar na

pesquisa em profundidade é como um algoritmo recursivo. Ele começa visitando `r`. Ao visitar um vértice `i`, primeiro marcamos `i` como `cinza`. Em seguida, varremos a lista de adjacências de `i` e visitamos recursivamente qualquer vértice branco que encontrarmos nesta lista. Finalmente, terminamos o processamento `i`, então colorimos `i` de preto e retornamos.

Algorithms
```
void dfs(Graph &g, int i, char *c) {
  c[i] = grey;  // currently visiting i
  ArrayStack<int> edges;
  g.outEdges(i, edges);
  for (int k = 0; k < edges.size(); k++) {
    int j = edges.get(k);
    if (c[j] == white) {
      c[j] = grey;
      dfs(g, j, c);
    }
  }
  c[i] = black; // done visiting i
}
void dfs(Graph &g, int r) {
  char *c = new char[g.nVertices()];
  dfs(g, r, c);
  delete[] c;
}
```

Um exemplo da execução deste algoritmo é mostrado em Figura 12.5.

Embora a pesquisa em profundidade possa ser melhor considerada um algoritmo recursivo, a recursão não é a melhor maneira de implementá-la. Na verdade, o código fornecido acima falhará para muitos grafos grandes, causando um estouro de pilha. Uma implementação alternativa é substituir a pilha de recursão por uma pilha explícita, `s`. A implementação a seguir faz exatamente isso:

Algorithms
```
void dfs2(Graph &g, int r) {
  char *c = new char[g.nVertices()];
  SLList<int> s;
  s.push(r);
  while (s.size() > 0) {
    int i = s.pop();
```

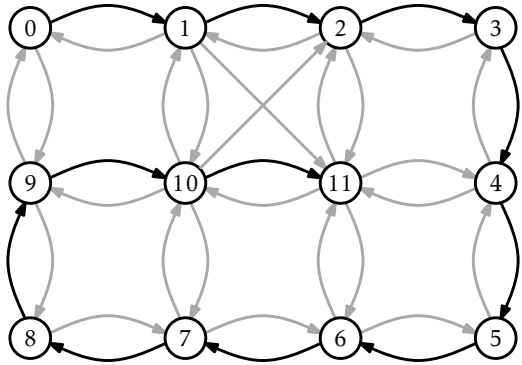


Figura 12.5: Um exemplo de pesquisa em profundidade começando no nó 0. Os nós são rotulados com a ordem em que são processados. As arestas que resultam em uma chamada recursiva são desenhadas em preto, outras arestas são desenhadas em `cinza`.

```
    if (c[i] == white) {
      c[i] = grey;
      ArrayStack<int> edges;
      g.outEdges(i, edges);
      for (int k = 0; k < edges.size(); k++)
        s.push(edges.get(k));
    }
  }
  delete[] c;
}
```

No código anterior, quando o próximo vértice, `i`, é processado, `i` é colorido `cinza` e então substituído, na pilha, por seus vértices adjacentes. Durante a próxima iteração, um desses vértices será visitado.

Não surpreendentemente, os tempos de execução de `dfs(g,r)` e `dfs2(g,r)` são iguais aos de `bfs(g,r)`:

Teorema 12.4. *Quando dado como entrada um* `Grafo`, `g`, *que é implementado usando a estrutura de dados* `AdjacencyLists`, *os algoritmos* `dfs(g,r)` *e* `dfs2(g,r)` *são executados em tempo* $O(n+m)$.

Tal como acontece com o algoritmo de pesquisa em largura, há uma árvore subjacente associada a cada execução da pesquisa em profundi-

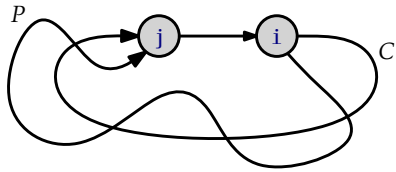


Figura 12.6: O algoritmo de pesquisa em profundidade pode ser usado para detectar ciclos em G. O nó `j` é colorido `cinza` enquanto `i` ainda é `cinza`. Isso implica que há um caminho, P, de `i` para `j` na árvore de pesquisa em profundidade, e a aresta (`j`, `i`) implica que P também é um ciclo.

dade. Quando um nó `i` $\neq$ `r` vai de `branco` para `cinza`, isso ocorre porque `dfs`(`g`, `i`, `c`) foi chamado recursivamente durante o processamento de algum nó `i`'. (No caso do `dfs2`(`g`, `r`) algoritmo, `i` é um dos nós que substituiu `i`' na pilha.) Se pensarmos em `i`' como o pai de `i`, então obtemos uma árvore enraizada em `r`. Em Figura 12.5, esta árvore é um caminho do vértice 0 ao vértice 11.

Uma propriedade importante do algoritmo de pesquisa em profundidade é a seguinte: Suponha que quando o nó `i` é colorido `cinza`, existe um caminho de `i` para algum outro nó `j` que usa apenas vértices brancos. Então `j` será colorido primeiro `cinza` depois `preto` antes de `i` ser colorido `preto`. (Isso pode ser provado por contradição, considerando qualquer caminho P de `i` a `j`.)

Uma das aplicações desta propriedade é a detecção de ciclos. Consulte a Figura 12.6. Considere algum ciclo, C, que pode ser alcançado a partir de `r`. Seja `i` o primeiro nó de C colorido `cinza` e seja `j` o nó que precede `i` no ciclo C. Então, pela propriedade acima, `j` será colorido `cinza` e a aresta (`j`, `i`) será considerada pelo algoritmo enquanto `i` ainda é `cinza`. Assim, o algoritmo pode concluir que existe um caminho, P, de `i` a `j` na árvore de pesquisa em profundidade e a aresta (`j`, `i`) existe. Portanto, P também é um ciclo.

12.4 Discussão e Exercícios

Os tempos de execução dos algoritmos de pesquisa em profundidade e em largura primeiro são um tanto exagerados pelos Teoremas 12.3 e 12.4. Defina $\mathtt{n}_\mathtt{r}$ como o número de vértices, `i`, de G, para os quais existe um caminho de `r` para `i`. Defina $\mathtt{m}_\mathtt{r}$ como o número de arestas que têm esses vértices como suas fontes. Então, o teorema a seguir é uma declaração mais precisa dos tempos de execução dos algoritmos de pesquisa em largura e em profundidade. (Esta declaração mais refinada do tempo de execução é útil em algumas das aplicações desses algoritmos descritos nos exercícios.)

Teorema 12.5. *Quando fornecido como entrada um* `Grafo`, `g`, *que é implementado usando a estrutura de dados* `AdjacencyLists`, *e algoritmos* `bfs(g,r)`, `dfs(g,r)` *e* `dfs2(g,r)`, *cada um executado em tempo* $O(\mathtt{n}_\mathtt{r} + \mathtt{m}_\mathtt{r})$.

A pesquisa em largura parece ter sido descoberta independentemente por Moore [52] e Lee [49] nos contextos de exploração de labirinto e roteamento de circuito, respectivamente.

As representações de listas de adjacências de grafos foram apresentadas por Hopcroft e Tarjan [40] como uma alternativa à (então mais comum) representação de matriz de adjacências. Essa representação, bem como a pesquisa em profundidade, desempenhou um papel importante no famoso algoritmo de teste de planaridade Hopcroft-Tarjan que pode determinar, em um tempo $O(\mathtt{n})$, se um grafo pode ser desenhado, no plano, e de forma que nenhum par de arestas se cruze [41].

Nos exercícios a seguir, um grafo não direcionado é aquele em que, para cada `i` e `j`, a aresta `(i,j)` está presente se e somente se a aresta `(j,i)` está presente.

Exercício 12.1. Desenhe uma representação de lista de adjacência e uma representação de matriz de adjacência do grafo em Figura 12.7.

Exercício 12.2. A representação *matriz de incidência* de um grafo, G, é uma $\mathtt{n} \times \mathtt{m}$, A, onde

$$A_{i,j} = \begin{cases} -1 & \text{se o vértice } i \text{ é a origem da aresta } j \\ +1 & \text{se o vértice } i \text{ é o alvo da aresta } j \\ 0 & \text{caso contrário.} \end{cases}$$

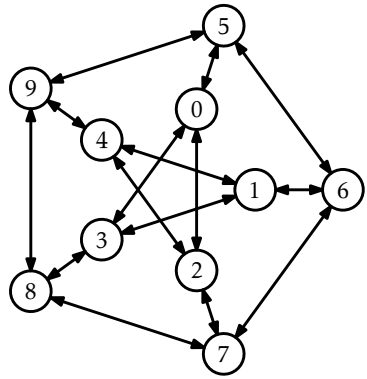


Figura 12.7: Um exemplo de grafo.

1. Desenhe a representação da matriz de incidentes do grafo na Figura 12.7.

2. Projetar, analisar e implementar uma representação de matriz de incidência de um grafo. Certifique-se de analisar o espaço, o custo de `addEdge(i, j)`, `removeEdge(i, j)`, `hasEdge(i, j)`, `inEdges(i)`, e `outEdges(i)`.

Exercício 12.3. Ilustre uma execução de `bfs(G,0)` e `dfs(G,0)` no grafo, `G`, in Figura 12.7.

Exercício 12.4. Seja G um grafo não direcionado. Dizemos que G é *conectado* se, para cada par de vértices `i` e `j` em G, há um caminho de `i` para `j` (uma vez que G é não direcionado, há também um caminho de `j` para `i`). Mostre como testar se G está conectado em um tempo $O(n+m)$.

Exercício 12.5. Seja G um grafo não direcionado. Uma *etiquetagem de componente conectado* de G divide os vértices de G em conjuntos máximos, cada um dos quais forma um subgrafo conectado. Mostre como calcular a etiquetagem de um componente conectado de G em um tempo $O(n+m)$.

Exercício 12.6. Seja G um grafo não direcionado. Uma *floresta extensa* de G é uma coleção de árvores, uma por componente, cujas arestas são arestas de G e cujos vértices contêm todos os vértices de G. Mostre como calcular uma floresta extensa de G em tempo $O(n+m)$.

Exercício 12.7. Dizemos que um grafo G é *fortemente conectado* se, para cada par de vértices `i` e `j` em G, houver um caminho de `i` para `j`. Mostre como testar se G está fortemente conectado em um tempo $O(n+m)$.

Exercício 12.8. Dado um grafo $G = (V, A)$ e algum vértice especial $r \in V$, mostre como calcular o comprimento do caminho mais curto de `r` a `i` para cada vértice $i \in V$.

Exercício 12.9. Dê um exemplo (simples) em que o código `dfs(g, r)` visita os nós de um grafo em uma ordem diferente daquela do código `dfs2(g, r)`. Escreva uma versão de `dfs2(g, r)` que sempre visita os nós exatamente na mesma ordem que `dfs(g, r)`. (Dica: basta começar a rastrear a execução de cada algoritmo em algum grafo onde `r` é a origem de mais de 1 aresta.)

Exercício 12.10. Um *sumidouro universal* em um grafo G é um vértice que é o alvo de $n-1$ arestas e a fonte de nenhuma aresta.[1] Projete e implemente um algoritmo que testa se um grafo G, representado como `AdjacencyMatrix`, tem um sumidouro universal. Seu algoritmo deve ser executado em tempo $O(n)$.

[1] Um sumidouro universal, `v`, também é às vezes chamado de *celebridade*: todos na sala reconhecem `v`, mas `v` não reconhece ninguém na sala.

Capítulo 13

Estruturas de Dados para Inteiros

Neste capítulo, voltamos ao problema de implementar um SSet. A diferença agora é que assumimos que os elementos armazenados no SSet são inteiros com w-bits. Ou seja, queremos implementar add(x), remove(x), e find(x) onde $x \in \{0,\ldots,2^w-1\}$. Não é muito difícil pensar em muitas aplicações em que os dados — ou pelo menos a chave que usamos para classificar os dados — é um número inteiro.

Discutiremos três estruturas de dados, cada uma com base nas idéias da anterior. A primeira estrutura, a BinaryTrie executa todas as três operações de um SSet em um tempo $O(w)$. Isso não é muito impressionante, pois qualquer subconjunto de $\{0,\ldots,2^w-1\}$ tem o tamanho $n \le 2^w$, para que $\log n \le w$. Todas as outras implementações de SSet discutidas neste livro realizam todas as operações em um tempo $O(\log n)$, para que sejam todas pelo menos tão rápidas quanto uma BinaryTrie.

A segunda estrutura, a XFastTrie, acelera a pesquisa em uma BinaryTrie usando hashing. Com esse aumento de velocidade, a operação find(x) é executada em um tempo $O(\log w)$. No entanto, as operações add(x) e remove(x) em uma XFastTrie ainda levam um tempo $O(w)$ e o espaço usado por uma XFastTrie é $O(n \cdot w)$.

A terceira estrutura de dados, YFastTrie, usa uma XFastTrie para armazenar apenas uma amostra de aproximadamente um de cada w elementos e armazena os elementos restantes em uma estrutura SSet padrão. Este truque reduz o tempo de execução de add(x) e remove(x) para $O(\log w)$ e diminui o espaço para $O(n)$.

As implementações usadas como exemplos neste capítulo podem ar-

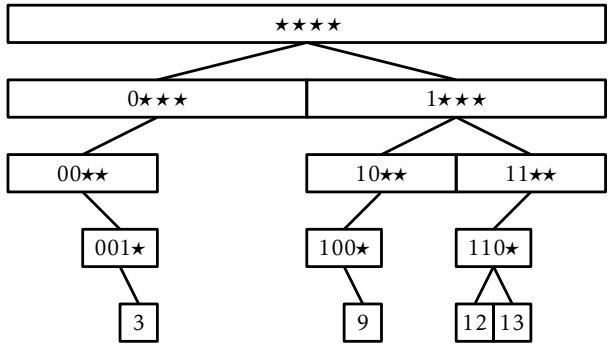


Figura 13.1: Os inteiros armazenados em uma trie binária são codificados como caminhos da raiz para a folha.

mazenar qualquer tipo de dados, desde que um inteiro possa ser associado a eles. Nos exemplos de código, a variável `ix` é sempre o valor inteiro associado a `x`, e o método `IntValue(x)` converte `x` em seu inteiro associado. No texto, entretanto, iremos simplesmente tratar `x` como se fosse um inteiro.

13.1 `BinaryTrie`: Uma árvore de busca digital

Uma `BinaryTrie` codifica um conjunto de inteiros de `w` bits em uma árvore binária. Todas as folhas da árvore têm profundidade `w` e cada número inteiro é codificado como um caminho da raiz para a folha. O caminho para o inteiro `x` vira à esquerda no nível `i` se o `i`ésimo bit mais significativo de `x` é um 0 e vira à direita se for 1. Figura 13.1 mostra um exemplo para o caso `w` = 4, no qual a trie armazena os inteiros 3(0011), 9(1001), 12(1100) e 13(1101).

Como o caminho de pesquisa para um valor `x` depende dos bits de `x`, será útil nomear os filhos de um nó, `u`, `u.child`[0] (`left`) e `u.child`[1] (`right`). Essas referências para os filhos na verdade servirão para dupla função. Como as folhas em uma trie binária não têm filhos, as referências são usadas para amarrar as folhas em uma lista duplamente encadeada.

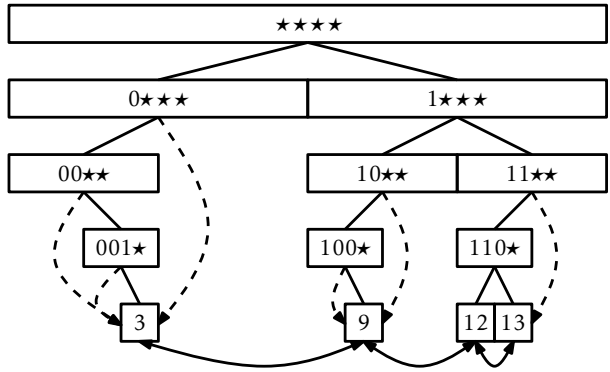


Figura 13.2: Uma `BinaryTrie` com ponteiros `jump` mostrados como arestas tracejadas curvas.

Para uma folha na trie binária `u.child[0]` (`prev`) é o nó que vem antes de `u` na lista e `u.child[1]` (`next`) é o nó que segue `u` na lista. Um nó especial, `dummy`, é usado antes do primeiro nó e depois do último nó da lista (ver Seção 3.2). Nos exemplos de código, `u.child[0]`, `u.left` e `u.prev` referem-se ao mesmo campo no nó `u`, assim como `u.child[1]`, `u.right` e `u.next`.

Cada nó, `u`, também contém um ponteiro adicional `u.jump`. Se o filho esquerdo de `u` estiver faltando, então `u.jump` aponta para a menor folha na subárvore de `u`. Se o filho direito de `u` estiver faltando, `u.jump` aponta para a maior folha na subárvore de `u`. Um exemplo de `BinaryTrie`, mostrando ponteiros `jump` e a lista duplamente encadeada nas folhas, é mostrado na Figura 13.2.

A operação `find(x)` em uma `BinaryTrie` é bastante simples. Tentamos seguir o caminho de pesquisa de `x` na trie. Se chegarmos a uma folha, encontramos `x`. Se chegarmos a um nó `u` onde não podemos prosseguir (porque `u` está faltando um filho), seguimos `u.jump`, que nos leva à menor folha maior que `x` ou à maior folha menor que `x`. Qual desses dois casos ocorre depende se em `u` está faltando seu filho esquerdo ou direito, respectivamente. No primeiro caso (em `u` está faltando seu filho esquerdo), encontramos o nó que desejamos. No último caso (em `u` está faltando seu filho direito), podemos usar a lista encadeada para chegar ao nó que desejamos. Cada um desses casos é ilustrado em Figura 13.3.

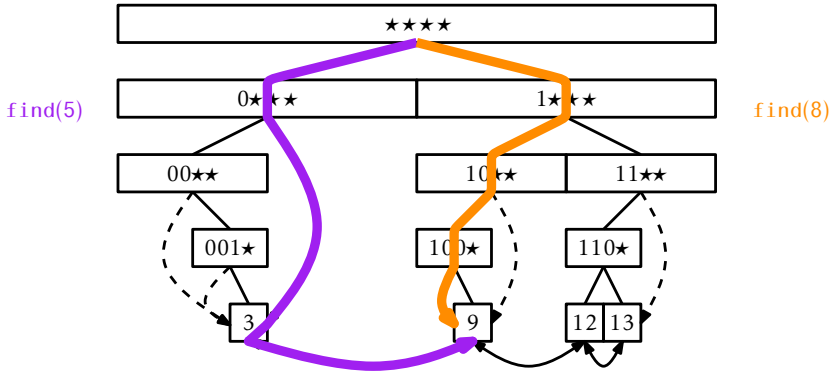


Figura 13.3: Os caminhos seguidos por `find(5)` e `find(8)`.

BinaryTrie

```
T find(T x) {
  int i, c = 0;
  unsigned ix = intValue(x);
  Node *u = &r;
  for (i = 0; i < w; i++) {
    c = (ix >> (w-i-1)) & 1;
    if (u->child[c] == NULL) break;
    u = u->child[c];
  }
  if (i == w) return u->x;  // found it
  u = (c == 0) ? u->jump : u->jump->next;
  return u == &dummy ? null : u->x;
}
```

O tempo de execução do método `find(x)` é dominado pelo tempo que leva para seguir um caminho da raiz para a folha, então ele é executado em tempo $O(w)$.

A operação `add(x)` em uma `BinaryTrie` também é bastante simples, mas tem muito trabalho a fazer:

1. Ele segue o caminho de busca por `x` até chegar a um nó `u` onde não pode mais prosseguir.

2. Ele cria o restante do caminho de pesquisa de `u` para uma folha que

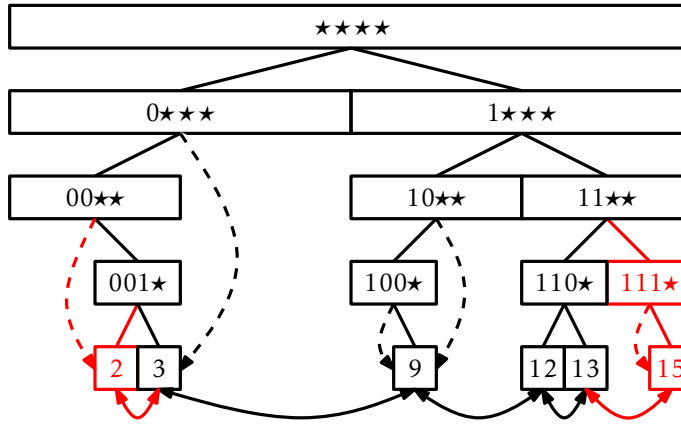


Figura 13.4: Adicionando os valores 2 e 15 à `BinaryTrie` na Figura 13.2.

contém `x`.

3. Ele adiciona o nó, `u'`, contendo `x` à lista encadeada de folhas (tem acesso ao predecessor, `pred`, de `u'` na lista encadeada do ponteiro `jump` do último nó, `u`, encontrado durante a etapa 1.)

4. Ele caminha de volta no caminho de pesquisa para `x`, ajustando os ponteiros `jump` nos nós cujo ponteiro de `jump` deve agora apontar para `x`.

Uma adição é ilustrada na Figura 13.4.

BinaryTrie

```
bool add(T x) {
  int i, c = 0;
  unsigned ix = intValue(x);
  Node *u = &r;
  // 1 - search for ix until falling out of the trie
  for (i = 0; i < w; i++) {
    c = (ix >> (w-i-1)) & 1;
    if (u->child[c] == NULL) break;
    u = u->child[c];
  }
  if (i == w) return false; // already contains x - abort
  Node *pred = (c == right) ? u->jump : u->jump->left;
```

```
  u->jump = NULL;  // u will have two children shortly
  // 2 - add path to ix
  for (; i < w; i++) {
    c = (ix >> (w-i-1)) & 1;
    u->child[c] = new Node();
    u->child[c]->parent = u;
    u = u->child[c];
  }
  u->x = x;
  // 3 - add u to linked list
  u->prev = pred;
  u->next = pred->next;;
  u->prev->next = u;
  u->next->prev = u;
  // 4 - walk back up, updating jump pointers
  Node *v = u->parent;
  while (v != NULL) {
    if ((v->left == NULL
        && (v->jump == NULL || intValue(v->jump->x) > ix))
    || (v->right == NULL
        && (v->jump == NULL || intValue(v->jump->x) < ix)))
      v->jump = u;
    v = v->parent;
  }
  n++;
  return true;
}
```

Este método executa uma caminhada pelo caminho de pesquisa de `x` e uma caminhada de volta para cima. Cada etapa dessas caminhadas leva um tempo constante, portanto o método `add(x)` é executado em um tempo $O(w)$.

A operação `remove(x)` desfaz o trabalho de `add(x)`. Como `add(x)`, ele tem muito trabalho a fazer:

1. Segue o caminho de busca por `x` até chegar à folha, `u`, que contém `x`.
2. Ele remove `u` da lista duplamente encadeada.
3. Ele exclui `u` e, em seguida, retorna ao caminho de pesquisa de `x` excluindo nós até chegar a um nó `v` que tem um filho que não está

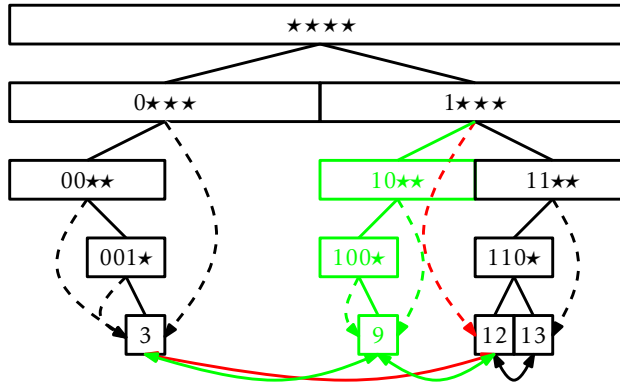


Figura 13.5: Removendo o valor 9 da `BinaryTrie` na Figura 13.2.

no caminho de pesquisa de `x`.

4. Ele sobe de `v` para a raiz, atualizando quaisquer ponteiros `jump` que apontam para `u`.

Uma remoção é ilustrada na Figura 13.5.

BinaryTrie

```
bool remove(T x) {
  // 1 - find leaf, u, containing x
  int i = 0, c;
  unsigned ix = intValue(x);
  Node *u = &r;
  for (i = 0; i < w; i++) {
    c = (ix >> (w-i-1)) & 1;
    if (u->child[c] == NULL) return false;
    u = u->child[c];
  }
  // 2 - remove u from linked list
  u->prev->next = u->next;
  u->next->prev = u->prev;
  Node *v = u;
  // 3 - delete nodes on path to u
  for (i = w-1; i >= 0; i--) {
    c = (ix >> (w-i-1)) & 1;
    v = v->parent;
    delete v->child[c];
```

```
    v->child[c] = NULL;
    if (v->child[1-c] != NULL) break;
  }
  // 4 - update jump pointers
  c = (ix >> (w-i-1)) & 1;
  v->jump = u->child[1-c];
  v = v->parent;
  i--;
  for (; i >= 0; i--) {
    c = (ix >> (w-i-1)) & 1;
    if (v->jump == u)
      v->jump = u->child[1-c];
    v = v->parent;
  }
  n--;
  return true;
}
```

Teorema 13.1. *Uma* `BinaryTrie` *implementa a interface* `SSet` *para inteiros de* `w`*-bits. Uma* `BinaryTrie` *suporta as operações* `add(x)`, `remove(x)` *e* `find(x)` *em um tempo* $O(w)$ *por operação. O espaço usado por uma* `BinaryTrie` *que armazena* `n` *valores é* $O(n \cdot w)$.

13.2 XFastTrie: Pesquisando em tempo duplamente logarítmico

O desempenho da estrutura `BinaryTrie` não é muito impressionante. O número de elementos, `n`, armazenados na estrutura é no máximo 2^w, então $\log n \leq w$. Em outras palavras, qualquer uma das estruturas `SSet` baseadas em comparação descritas em outras partes deste livro são pelo menos tão eficientes quanto `BinaryTrie` e não estão restritas a armazenar apenas inteiros.

Em seguida, descrevemos a `XFastTrie`, que é apenas uma `BinaryTrie` com $w+1$ tabelas hash — uma para cada nível do teste. Essas tabelas hash são usadas para acelerar a operação `find(x)` para um tempo $O(\log w)$. Lembre-se de que a operação `find(x)` em uma `BinaryTrie` está quase completa quando alcançamos um nó, `u`, onde o caminho de pes-

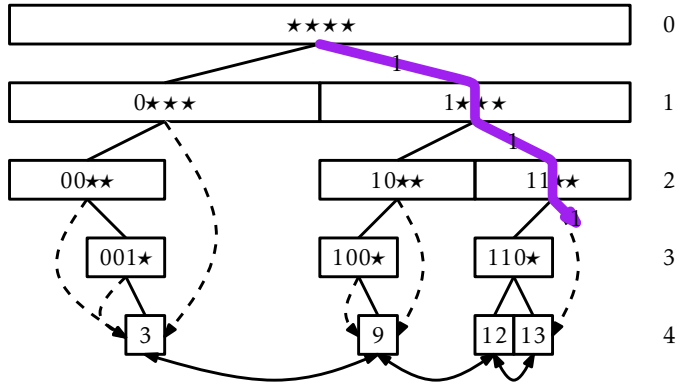


Figura 13.6: Como não há nenhum nó rotulado como 111★, o caminho de pesquisa para 14 (1110) termina no nó rotulado 11★★.

quisa para `x` gostaria de prosseguir para `u.right` (ou `u.left`) mas `u` não tem filho direito (respectivamente, esquerdo). Neste ponto, a pesquisa usa `u.jump` para pular para uma folha, `v`, da `BinaryTrie` e retornar `v` ou seu sucessor na lista encadeada de folhas. Uma `XFastTrie` acelera o processo de pesquisa usando a pesquisa binária nos níveis da trie para localizar o nó `u`.

Para usar a pesquisa binária, precisamos determinar se o nó `u` que estamos procurando está acima de um determinado nível, `i`, ou se `u` está no nível ou abaixo do `i`. Esta informação é fornecida pelos bits `i` de ordem mais alta na representação binária de `x`; esses bits determinam o caminho de pesquisa que `x` leva da raiz ao nível `i`. Para um exemplo, consulte Figura 13.6; nesta figura, o último nó, `u`, no caminho de pesquisa para 14 (cuja representação binária é 1110) é o nó rotulado 11★★ no nível 2 porque não há nenhum nó rotulado 111★ no nível 3. Portanto, podemos rotular cada nó no nível `i` com um número inteiro de `i` bits. Então, o nó `u` que estamos procurando estaria no nível `i` ou abaixo se e somente se houvesse um nó no nível `i` cujo rótulo corresponda aos bits `i` de ordem mais alta de `x`.

Em uma `XFastTrie`, armazenamos, para cada $i \in \{0,\ldots,w\}$, todos os nós no nível `i` em um `USet`, `t[i]`, que é implementado como uma tabela hash (Capítulo 5). Usar este `USet` nos permite verificar em tempo espe-

rado constante se há um nó no nível `i` cujo rótulo corresponde aos bits `i` de ordem mais alta de `x`. Na verdade, podemos até encontrar este nó usando `t[i].find(x >> (w - i))`

As tabelas de hash `t[0],...,t[w]` nos permitem usar a pesquisa binária para encontrar `u`. Inicialmente, sabemos que `u` está em algum nível `i` com $0 \le i < w+1$. Portanto, inicializamos $l = 0$ e $h = w+1$ e repetidamente olhamos para a tabela hash `t[i]`, onde $i = \lfloor (l+h)/2 \rfloor$. Se `t[i]` contém um nó cujo rótulo corresponde aos bits `i` de ordem superior de `x`, então definimos $l = i$ (`u` está no nível ou abaixo de `i`); caso contrário, definimos $h = i$ (`u` está acima do nível `i`). Este processo termina quando $h - l \le 1$, caso em que determinamos que `u` está no nível `l`. Em seguida, completamos a operação `find(x)` usando `u.jump` e a lista duplamente encadeada de folhas.

XFastTrie

```
T find(T x) {
  int l = 0, h = w+1;
  unsigned ix = intValue(x);
  Node *v, *u = &r;
  while (h-l > 1) {
    int i = (l+h)/2;
    XPair<Node> p(ix >> (w-i));
    if ((v = t[i].find(p).u) == NULL) {
      h = i;
    } else {
      u = v;
      l = i;
    }
  }
  if (l == w) return u->x;
  Node *pred = (((ix >> (w-l-1)) & 1) == 1)
               ? u->jump : u->jump->prev;
  return (pred->next == &dummy) ? nullt : pred->next->x;
}
```

Cada iteração do loop `while` no método acima diminui $h - l$ por aproximadamente um fator de dois, então este loop encontra `u` após $O(\log w)$ iterações. Cada iteração executa uma quantidade constante de trabalho e uma operação `find(x)` em um `USet`, que leva um tempo esperado constante. O trabalho restante leva apenas um tempo constante, portanto o

método find(x) em uma XFastTrie leva apenas um tempo esperado de $O(\log w)$.

Os métodos add(x) e remove(x) para uma XFastTrie são quase idênticos aos mesmos métodos em uma BinaryTrie. As únicas modificações são para gerenciar as tabelas hash t[0],...,t[w]. Durante a operação add(x), quando um novo nó é criado no nível i, este nó é adicionado a t[i]. Durante uma operação remove(x), quando um nó é removido do nível i, este nó é removido de t[i]. Como adicionar e remover de uma tabela hash leva um tempo esperado constante, isso não aumenta os tempos de execução de add(x) e remove(x) por mais de um fator constante. Omitimos uma listagem de código para add(x) e remove(x), pois o código é quase idêntico à (longa) listagem de código já fornecida para os mesmos métodos em uma BinaryTrie.

O teorema a seguir resume o desempenho de uma XFastTrie:

Teorema 13.2. *Uma XFastTrie implementa a interface SSet para inteiros com w-bits. Uma XFastTrie suporta as operações*

- add(x) *e* remove(x) *em tempo esperado de* $O(w)$ *por operação e*
- find(x) *em tempo esperado de* $O(\log w)$ *por operação.*

O espaço usado por uma XFastTrie que armazena n *valores é* $O(n \cdot w)$.

13.3 YFastTrie: Um SSet de tempo duplamente logarítmico

A XFastTrie é uma grande – até exponencial – melhoria em relação à BinaryTrie em termos de tempo de consulta, mas as operações add(x) e remove(x) ainda não são terrivelmente rápidas. Além disso, o uso de espaço, $O(n \cdot w)$, é maior do que as outras implementações de SSet descritas neste livro, que usam $O(n)$ espaço. Esses dois problemas estão relacionados; se n add(x) operações constroem uma estrutura de tamanho $n \cdot w$, então a operação add(x) requer pelo menos na ordem de w tempo (e espaço) por operação.

A YFastTrie, discutida a seguir, melhora simultaneamente o espaço e a velocidade das XFastTries. Uma YFastTrie usa uma XFastTrie, xft,

mas armazena apenas $O(n/w)$ valores em `xft`. Desta forma, o espaço total usado por `xft` é apenas $O(n)$. Além disso, apenas uma de cada w `add(x)` ou `remove(x)` operações em `YFastTrie` resulta em uma operação `add(x)` ou `remove(x)` em `xft` . Fazendo isso, o custo médio incorrido por chamadas para `xft` operações `add(x)` e `remove(x)` é apenas constante.

A pergunta óbvia é: Se `xft` armazena apenas n/w elementos, para onde vão os $n(1-1/w)$ elementos restantes? Esses elementos se movem para *estruturas secundárias*, neste caso, uma versão estendida de treaps(Seção 7.2). Existem aproximadamente n/w dessas estruturas secundárias, portanto, em média, cada uma delas armazena $O(w)$ itens. Treaps suportam operações de `SSet` em tempo logarítmico, então as operações nesses treaps serão executadas em tempo $O(\log w)$, conforme necessário.

Mais concretamente, uma `YFastTrie` contém uma `XFastTrie`, `xft`, que contém uma amostra aleatória dos dados, onde cada elemento aparece na amostra independentemente com probabilidade $1/w$. Por conveniência, o valor $2^w - 1$, está sempre contido em `xft`. Faça $x_0 < x_1 < \cdots < x_{k-1}$ denotar os elementos armazenados em `xft`. Associado a cada elemento, x_i, está um treap, t_i, que armazena todos os valores no intervalo $x_{i-1}+1,\ldots,x_i$. Isso é ilustrado em Figura 13.7.

A operação `find(x)` em uma `YFastTrie` é bastante fácil. Procuramos por `x` em `xft` e encontramos algum valor x_i associado ao treap t_i. Em seguida, usamos o método da treap `find(x)` em t_i para responder à consulta. Todo o método é de uma linha:

YFastTrie

```
T find(T x) {
  return xft.find(YPair<T>(intValue(x))).t->find(x);
}
```

A primeira operação `find(x)` (em `xft`) leva um tempo $O(\log w)$. A segunda operação `find(x)` (em uma treap) leva um tempo $O(\log r)$, onde r é o tamanho da treap. Posteriormente nesta seção, mostraremos que o tamanho esperado da treap é $O(w)$, de modo que esta operação leva um tempo $O(\log w)$.[1]

Adicionar um elemento a uma `YFastTrie` também é bastante simples — na maioria das vezes. O método `add(x)` chama `xft.find(x)` para loca-

[1] Esta é uma aplicação da *Desigualdade de Jensen*: Se $E[r] = w$, então $E[\log r] \le \log w$.

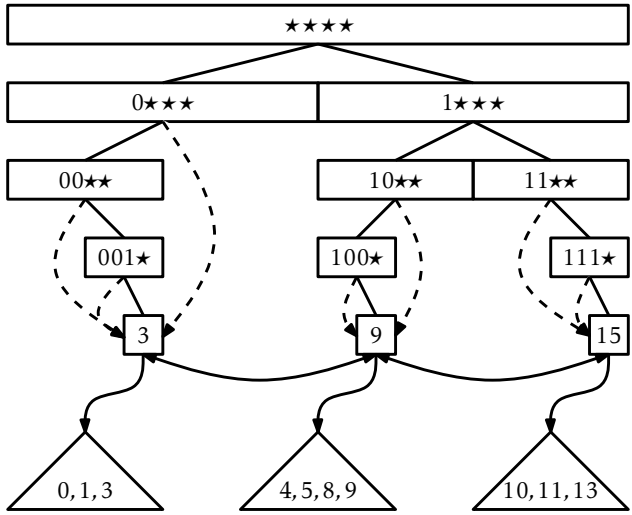


Figura 13.7: Uma YFastTrie contendo os valores 0, 1, 3, 4, 6, 8, 9, 10, 11, e 13.

lizar a treap, t, na qual x deve ser inserido. Em seguida, chama t.add(x) para adicionar x a t. Nesse ponto, ele lança uma moeda tendenciosa que sai cara com probabilidade 1/w e coroa com probabilidade $1 - 1/w$. Se esta moeda der cara, então x será adicionado a xft.

É aqui que as coisas ficam um pouco mais complicadas. Quando x é adicionado à xft, a treap t precisa ser dividida em duas treaps, t1 e t'. A treap t1 contém todos os valores menores ou iguais a x; t' é a treap original, t, com os elementos de t1 removidos. Feito isso, adicionamos o par (x, t1) a xft. A Figura 13.8 mostra um exemplo.

YFastTrie

```
bool add(T x) {
  unsigned ix = intValue(x);
  Treap1<T> *t = xft.find(YPair<T>(ix)).t;
  if (t->add(x)) {
    n++;
    if (rand() % w == 0) {
      Treap1<T> *t1 = (Treap1<T>*)t->split(x);
      xft.add(YPair<T>(ix, t1));
    }
    return true;
```

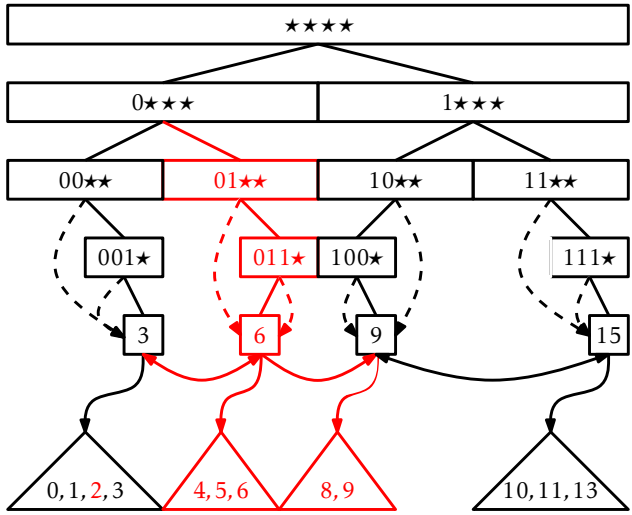


Figura 13.8: Adicionando os valores 2 e 6 a uma `YFastTrie`. O sorteio de 6 deu cara, então 6 foi adicionado à `xft` e a treap contendo 4, 5, 6, 8, 9 foi dividida.

```
  }
  return false;
  return true;
}
```

Adicionar `x` a `t` leva um tempo $O(\log w)$. Exercício 7.12 mostra que dividir `t` em `t1` e `t'` também pode ser feito em um tempo esperado de $O(\log w)$. Adicionar o par (`x`, `t1`) a `xft` leva um tempo $O(w)$, mas só acontece com a probabilidade $1/w$. Portanto, o tempo de execução esperado da operação `add(x)` é

$$O(\log w) + \frac{1}{w} O(w) = O(\log w) \ .$$

O método `remove(x)` desfaz o trabalho executado por `add(x)`. Usamos `xft` para encontrar a folha, `u`, em `xft` que contém a resposta para `xft.find(x)`. De `u`, obtemos a treap, `t`, contendo `x` e removemos `x` de `t`. Se `x` também foi armazenado em `xft` (e `x` não é igual a $2^w - 1$), então removemos `x` de `xft` e adicionamos os elementos da treap de `x` à treap, `t2`, que é armazenada pelo sucessor de `u` na lista encadeada. Isso é ilustrado em Figura 13.9.

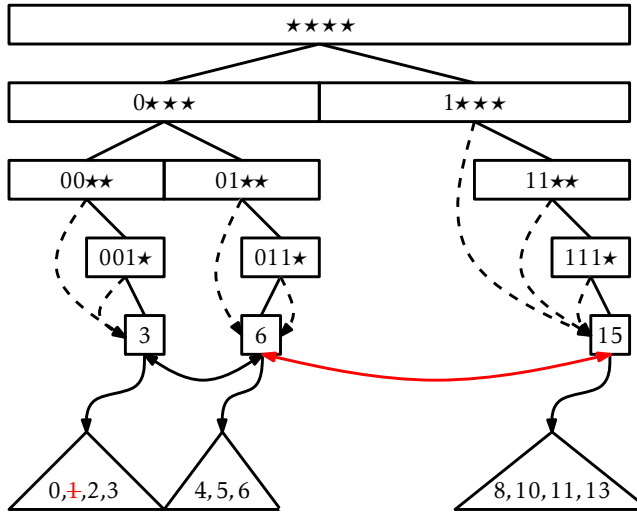


Figura 13.9: Removendo os valores 1 e 9 de uma YFastTrie na Figura 13.8.

YFastTrie

```
bool remove(T x) {
  unsigned ix = intValue(x);
  XFastTrieNode1<YPair<T> > *u = xft.findNode(ix);
  bool ret = u->x.t->remove(x);
  if (ret) n--;
  if (u->x.ix == ix && ix != UINT_MAX) {
    Treap1<T> *t2 = u->child[1]->x.t;
    t2->absorb(*u->x.t);
    xft.remove(u->x);
  }
  return ret;
}
```

Encontrar o nó `u` em `xft` leva um tempo esperado de $O(\log w)$. Remover `x` de `t` leva um tempo esperado $O(\log w)$. Novamente, o Exercício 7.12 mostra que mesclar todos os elementos de `t` em `t2` pode ser feito em tempo $O(\log w)$. Se necessário, remover `x` de `xft` leva um tempo $O(w)$, mas `x` só está contido em `xft` com probabilidade $1/w$. Portanto, o tempo esperado para remover um elemento de uma YFastTrie é $O(\log w)$.

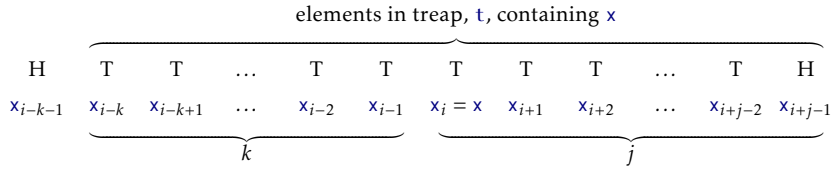


Figura 13.10: O número de elementos na treap $\mathtt{t}$ contendo $\mathtt{x}$ é determinado por dois experimentos de lançamento de moeda.

No início da discussão, atrasamos a discussão sobre os tamanhos das treaps nesta estrutura para mais tarde. Antes de terminar este capítulo, provamos o resultado de que precisamos.

Lema 13.1. *Seja* $\mathtt{x}$ *um inteiro armazenado em uma* `YFastTrie` *e seja* $\mathtt{n_x}$ *o número de elementos na treap,* $\mathtt{t}$*, que contém* $\mathtt{x}$*. Então* $\mathrm{E}[\mathtt{n_x}] \le 2\mathtt{w} - 1$.

Demonstração. Referir-se à Figura 13.10. Seja $\mathtt{x}_1 < \mathtt{x}_2 < \cdots < \mathtt{x}_i = \mathtt{x} < \mathtt{x}_{i+1} < \cdots < \mathtt{x}_n$ os elementos armazenados na `YFastTrie`. A treap $\mathtt{t}$ contém alguns elementos maiores ou iguais a $\mathtt{x}$. Esses são $\mathtt{x}_i, \mathtt{x}_{i+1}, \ldots, \mathtt{x}_{i+j-1}$, onde $\mathtt{x}_{i+j-1}$ é o único desses elementos em que o lance de moeda tendencioso realizado no método `add`$(\mathtt{x})$ deu cara. Em outras palavras, $\mathrm{E}[j]$ é igual ao número esperado de lançamentos tendenciosos de moeda necessários para obter as primeiras caras.[2] Cada sorteio é independente e sai cara com probabilidade $1/\mathtt{w}$, então $\mathrm{E}[j] \le \mathtt{w}$. (Veja Lema 4.2 para uma análise disto para o caso $\mathtt{w} = 2$.)

Da mesma forma, os elementos de $\mathtt{t}$ menores que $\mathtt{x}$ são $\mathtt{x}_{i-1}, \ldots, \mathtt{x}_{i-k}$ onde todos esses k lançamentos de moeda resultam em coroa e o lançamento de $\mathtt{x}_{i-k-1}$ dá cara. Portanto, $\mathrm{E}[k] \le \mathtt{w}-1$, já que este é o mesmo experimento de lançamento de moeda considerado no parágrafo anterior, mas aquele em que o último lançamento não é contado. Em resumo, $\mathtt{n_x} = j + k$, então

$$\mathrm{E}[\mathtt{n_x}] = \mathrm{E}[j + k] = \mathrm{E}[j] + \mathrm{E}[k] \le 2\mathtt{w} - 1 \ . \qquad \square$$

Lema 13.1 foi a última peça na prova do seguinte teorema, que resume o desempenho da `YFastTrie`:

[2] Esta análise ignora o fato de que j nunca excede $\mathtt{n} - i + 1$. No entanto, isso apenas diminui $\mathrm{E}[j]$, então o limite superior ainda se mantém.

Teorema 13.3. *Uma* `YFastTrie` *implementa a interface* `SSet` *para inteiros de* `w`*-bits. Uma* `YFastTrie` *suporta as operações* `add(x)`, `remove(x)` *e* `find(x)` *em tempo esperado de* $O(\log w)$ *por operação. O espaço usado por uma* `YFastTrie` *que armazena* `n` *valores é* $O(n+w)$.

O termo `w` no requisito de espaço vem do fato de que `xft` sempre armazena o valor $2^w - 1$. A implementação pode ser modificada (às custas de adicionar alguns casos extras ao código) para que seja desnecessário armazenar esse valor. Neste caso, o requisito de espaço no teorema torna-se $O(n)$.

13.4 Discussão e Exercícios

A primeira estrutura de dados a fornecer operações `add(x)`, `remove(x)` e `find(x)` em um tempo $O(\log w)$ foi proposta por van Emde Boas e desde então tornou-se conhecida como o *árvore van Emde Boas* (ou *estratificada*) [72]. A estrutura original de van Emde Boas tinha tamanho 2^w, tornando-a impraticável para números inteiros grandes.

As estruturas de dados `XFastTrie` e `YFastTrie` foram descobertas por Willard [75]. A estrutura `XFastTrie` está intimamente relacionada às árvores van Emde Boas; por exemplo, as tabelas de hash em uma `XFastTrie` substituem arrays em uma árvore van Emde Boas. Ou seja, em vez de armazenar a tabela de hash `t[i]`, uma árvore van Emde Boas armazena uma matriz de comprimento 2^i.

Outra estrutura para armazenar inteiros são as árvores de fusão de Fredman e Willard [32]. Esta estrutura pode armazenar `n` inteiros de `w` bits em um espaço $O(n)$ de modo que a operação `find(x)` execute em um tempo $O((\log n)/(\log w))$. Usando uma árvore de fusão quando $\log w > \sqrt{\log n}$ e uma `YFastTrie` quando $\log w \le \sqrt{\log n}$, obtém-se uma estrutura de dados de espaço $O(n)$ que pode implementar a operação `find(x)` em tempo $O(\sqrt{\log n})$. Resultados recentes de limite inferior de Pătraşcu e Thorup [57] mostram que esses resultados são mais ou menos ideais, pelo menos para estruturas que usam apenas espaço $O(n)$.

Exercício 13.1. Projete e implemente uma versão simplificada de uma `BinaryTrie` que não tenha uma lista encadeada ou ponteiros de salto,

mas para o qual `find(x)` ainda é executado em tempo $O(w)$.

Exercício 13.2. Projete e implemente uma implementação simplificada de uma `XFastTrie` que não use um teste binário. Em vez disso, sua implementação deve armazenar tudo em uma lista duplamente encadeada e $w+1$ tabelas de hash.

Exercício 13.3. Podemos pensar na `BinaryTrie` como uma estrutura que armazena sequências de bits de comprimento `w` de forma que cada sequência de bits seja representada como um caminho da raiz para folha. Estenda essa ideia em uma implementação SSet que armazena strings de comprimento variável e implementa `add(s)`, `remove(s)` e `find(s)` no tempo proporcional ao comprimento de `s`.
Dica: Cada nó em sua estrutura de dados deve armazenar uma tabela hash que é indexada por valores de caractere.

Exercício 13.4. Para um inteiro $x \in \{0,\ldots 2^w-1\}$, faça $d(x)$ denotar a diferença entre `x` e o valor retornado por `find(x)` [se `find(x)` retorna `null`, então defina $d(x)$ como 2^w]. Por exemplo, se `find(23)` retorna 43, então $d(23) = 20$.

1. Projete e implemente uma versão modificada da operação `find(x)` em uma `XFastTrie` executada em tempo esperado de $O(1+\log d(x))$. Dica: a tabela hash $t[w]$ contém todos os valores, `x`, de forma que $d(x) = 0$, então esse seria um bom lugar para começar.

2. Projete e implemente uma versão modificada da operação `find(x)` em uma `XFastTrie` executada em um tempo esperado de $O(1 + \log\log d(x))$.

Capítulo 14

Pesquisa em memória externa

Ao longo deste livro, temos usado o modelo de computação com uma palavra de RAM de w-bits definido em Seção 1.4. Uma suposição implícita desse modelo é que nosso computador tem uma memória de acesso aleatório grande o suficiente para armazenar todos os dados na estrutura de dados. Em algumas situações, essa suposição não é válida. Existem coleções de dados tão grandes que nenhum computador tem memória suficiente para armazená-los. Nesses casos, a aplicação deve recorrer ao armazenamento dos dados em algum meio de armazenamento externo, como um disco rígido, um disco de estado sólido ou mesmo um servidor de arquivos de rede (que possui seu próprio armazenamento externo).

O acesso a um item de armazenamento externo é extremamente lento. O disco rígido conectado ao computador no qual este livro foi escrito tem um tempo médio de acesso de 19ms e a unidade de estado sólido conectada ao computador tem um tempo médio de acesso de 0,3ms. Em contraste, a memória de acesso aleatório do computador tem um tempo médio de acesso inferior a 0,000113ms. O acesso à RAM é mais de 2.500 vezes mais rápido do que acessar a unidade de estado sólido e mais de 160.000 vezes mais rápido do que acessar o disco rígido.

Essas velocidades são bastante típicas; acessar um byte aleatório da RAM é milhares de vezes mais rápido do que acessar um byte aleatório de um disco rígido ou unidade de estado sólido. O tempo de acesso, entretanto, não conta toda a história. Quando acessamos um byte de um disco rígido ou disco de estado sólido, um *bloco* inteiro do disco é lido. Cada uma das unidades conectadas ao computador possui um tamanho

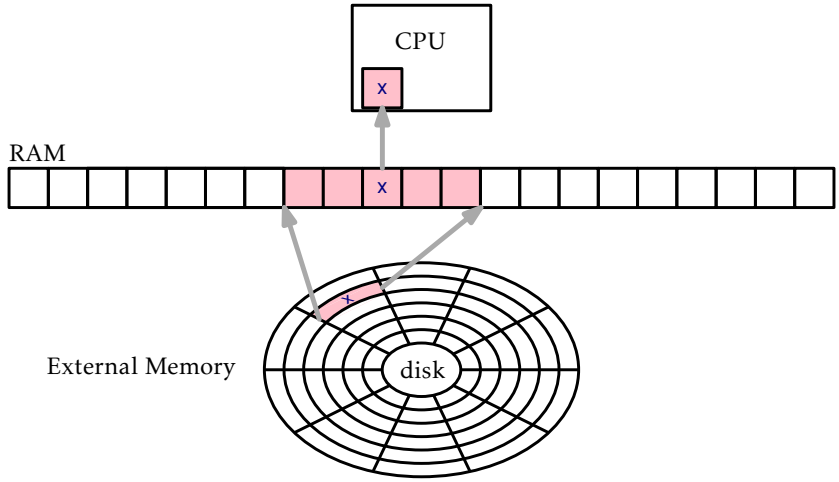


Figura 14.1: No modelo de memória externa, acessar um item individual, x, na memória externa requer a leitura de todo o bloco contendo x na RAM.

de bloco de 4096; cada vez que lemos um byte, o drive nos dá um bloco contendo 4096 bytes. Se organizarmos nossa estrutura de dados com cuidado, isso significa que cada acesso ao disco pode render 4096 bytes que são úteis para completar qualquer operação que estivermos fazendo.

Esta é a ideia por trás do *modelo de memória externa* de computação, ilustrado esquematicamente em Figura 14.1. Nesse modelo, o computador tem acesso a uma grande memória externa na qual residem todos os dados. Esta memória é dividida em *blocos* de memória cada um contendo B palavras. O computador também possui memória interna limitada, na qual pode realizar cálculos. Transferir um bloco entre a memória interna e a memória externa leva um tempo constante. Os cálculos realizados na memória interna são *gratuitos*; eles não levam tempo algum. O fato de os cálculos da memória interna serem gratuitos pode parecer um pouco estranho, mas simplesmente enfatiza o fato de que a memória externa é muito mais lenta do que a RAM.

No modelo de memória externa completo, o tamanho da memória interna também é um parâmetro. Porém, para as estruturas de dados descritas neste capítulo, é suficiente ter uma memória interna de tamanho $O(B + \log_B n)$. Ou seja, a memória precisa ser capaz de armazenar um

número constante de blocos e uma pilha de recursão de altura $O(\log_B n)$. Na maioria dos casos, o termo $O(B)$ domina o requisito de memória. Por exemplo, mesmo com o valor relativamente pequeno $B = 32$, $B \geq \log_B n$ para todos $n \leq 2^{160}$. Em decimal, $B \geq \log_B n$ para qualquer

$$n \leq 1\,461\,501\,637\,330\,902\,918\,203\,684\,832\,716\,283\,019\,655\,932\,542\,976 \ .$$

14.1 O Armazém de Blocos - BlockStore

A noção de memória externa inclui um grande número de dispositivos diferentes possíveis, cada um dos quais tem seu próprio tamanho de bloco e é acessado com sua própria coleção de chamadas de sistema. Para simplificar a exposição deste capítulo para que possamos nos concentrar nas ideias comuns, encapsulamos dispositivos de memória externa com um objeto chamado `BlockStore`. Um `BlockStore` armazena uma coleção de blocos de memória, cada um com o tamanho B. Cada bloco é identificado exclusivamente por seu índice inteiro. Um `BlockStore` oferece suporte a estas operações:

1. `readBlock(i)`: Retorna o conteúdo do bloco cujo índice é `i`.
2. `writeBlock(i,b)`: Grave o conteúdo de `b` no bloco cujo índice é `i`.
3. `placeBlock(b)`: Retorne um novo índice e armazene o conteúdo de `b` neste índice.
4. `freeBlock(i)`: Libere o bloco cujo índice é `i`. Isso indica que o conteúdo deste bloco não é mais usado, então a memória externa alocada por este bloco pode ser reutilizada.

A maneira mais fácil de imaginar um `BlockStore` é imaginá-lo armazenando um arquivo em disco que é particionado em blocos, cada um contendo B bytes. Desta forma, `readBlock(i)` e `writeBlock(i,b)` simplesmente lêem e gravam os bytes $iB,\ldots,(i+1)B-1$ deste arquivo. Além disso, um `BlockStore` simples pode manter uma *lista livre* de blocos que estão disponíveis para uso. Os blocos liberados com `freeBlock(i)` são adicionados à lista livre. Desta forma, `placeBlock(b)` pode usar um bloco da lista livre ou, se nenhum estiver disponível, acrescentar um novo bloco ao final do arquivo.

14.2 Árvores B (B-Trees)

Nesta seção, discutimos uma generalização de árvores binárias, chamadas de árvores B, que são eficientes no modelo de memória externa. Alternativamente, árvores B podem ser vistos como a generalização natural de árvores 2-4 descritas em Seção 9.1. (Uma árvore 2-4 é um caso especial de árvore B que obtemos definindo $B = 2$.)

Para qualquer inteiro $B \geq 2$, uma *árvore B* é uma árvore em que todas as folhas têm a mesma profundidade e cada nó interno não raiz, u, tem pelo menos B filhos e no máximo $2B$ filhos. Os filhos de u são armazenados em uma matriz, u.children. O número necessário de filhos é relaxado na raiz, podendo ter entre 2 e $2B$ filhos.

Se a altura de uma árvore B é h, segue-se que o número, ℓ, de folhas na árvore B satisfaz

$$2B^{h-1} \leq \ell \leq (2B)^h \ .$$

Tomando o logaritmo da primeira desigualdade e reorganizando os termos, obtém-se:

$$\begin{aligned} h &\leq \frac{\log \ell - 1}{\log B} + 1 \\ &\leq \frac{\log \ell}{\log B} + 1 \\ &= \log_B \ell + 1 \ . \end{aligned}$$

Ou seja, a altura de uma árvore B é proporcional ao logaritmo de base B do número de folhas.

Cada nó, u, na árvore B armazena uma matriz de chaves u.keys[0],...,u.keys[$2B-1$]. Se u for um nó interno com k filhos, então o número de chaves armazenadas em u é exatamente $k-1$ e estas são armazenadas em u.keys[0],...,u.keys[$k-2$]. As entradas restantes da matriz $2B-k+1$ em u.keys são definidas como null. Se u for um nó folha não raiz, então u contém entre as chaves $B-1$ e $2B-1$. As chaves em uma árvore B respeitam uma ordem semelhante às chaves em uma árvore de pesquisa binária. Para qualquer nó, u, que armazena $k-1$ chaves,

$$\texttt{u.keys}[0] < \texttt{u.keys}[1] < \cdots < \texttt{u.keys}[k-2] \ .$$

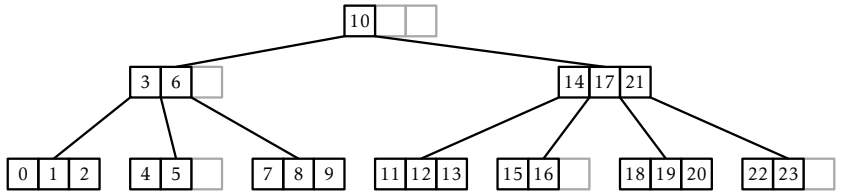


Figura 14.2: Uma árvore B com $B = 2$.

Se `u` for um nó interno, então para cada `i` $\in \{0,\ldots,k-2\}$, `u.keys[i]` é maior do que todas as chaves armazenadas na subárvore com raiz em `u.children[i]`, mas menor do que todas as chaves armazenadas na subárvore com raiz em `u.children[i + 1]`. Informalmente,

$$\texttt{u.children[i]} \prec \texttt{u.keys[i]} \prec \texttt{u.children[i + 1]} \ .$$

Um exemplo de uma árvore B com $B = 2$ é mostrado na Figura 14.2.

Observe que os dados armazenados em um nó da árvore B têm o tamanho $O(B)$. Portanto, em uma configuração de memória externa, o valor de B em uma árvore B é escolhido de forma que um nó caiba em um único bloco de memória externa. Desta forma, o tempo que leva para realizar uma operação na árvore B no modelo de memória externa é proporcional ao número de nós que são acessados (lidos ou gravados) pela operação.

Por exemplo, se as chaves são inteiros de 4 bytes e os índices dos nós também são 4 bytes, então definir $B = 256$ significa que cada nó armazena

$$(4+4) \times 2B = 8 \times 512 = 4096$$

bytes de dados. Este seria um valor perfeito de B para o disco rígido ou unidade de estado sólido discutida na introdução deste capítulo, que tem um tamanho de bloco de 4096 bytes.

A classe `BTree`, que implementa uma árvore B, armazena um `BlockStore`, `bs`, que armazena `BTree` nós, bem como o índice, `ri`, do nó raiz. Como de costume, um número inteiro, `n`, é usado para controlar o número de itens na estrutura de dados:

BTree
```
int n; // number of elements stored in the tree
int ri; // index of the root
```

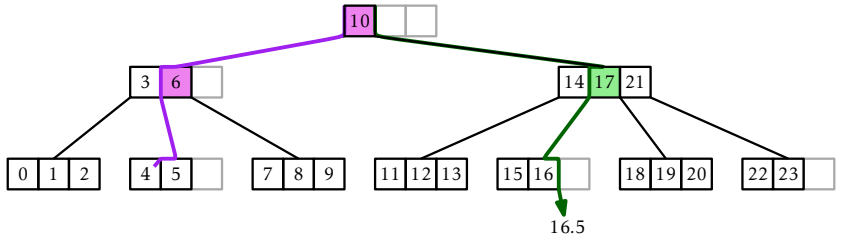


Figura 14.3: Uma pesquisa bem-sucedida (pelo valor 4) e uma pesquisa malsucedida (pelo valor 16.5) em uma árvore *B*. Os nós sombreados mostram onde o valor de `z` é atualizado durante as pesquisas.

```
BlockStore<Node*> bs;
```

14.2.1 Busca

A implementação da operação `find(x)`, ilustrada em Figura 14.3, generaliza a operação `find(x)` em uma árvore de pesquisa binária. A pesquisa por `x` começa na raiz e usa as chaves armazenadas em um nó, `u`, para determinar em qual dos filhos de `u` a pesquisa deve continuar.

Mais especificamente, em um nó `u`, a pesquisa verifica se `x` está armazenado em `u.keys`. Nesse caso, `x` foi encontrado e a pesquisa foi concluída. Caso contrário, a pesquisa encontra o menor inteiro, `i`, de modo que `u.keys[i]` > `x` e continua a pesquisa na subárvore com raiz em `u.children[i]`. Se nenhuma chave em `u.keys` for maior que `x`, a busca continua no filho mais à direita de `u`. Assim como as árvores de busca binária, o algoritmo rastreia a chave vista mais recentemente, `z`, que é maior do que `x`. Caso `x` não seja encontrado, `z` é retornado como o menor valor maior ou igual a `x`.

BTree
```
T find(T x) {
  T z = null;
  int ui = ri;
  while (ui >= 0) {
    Node *u = bs.readBlock(ui);
    int i = findIt(u->keys, x);
    if (i < 0) return u->keys[-(i+1)]; // found it
    if (u->keys[i] != null)
```

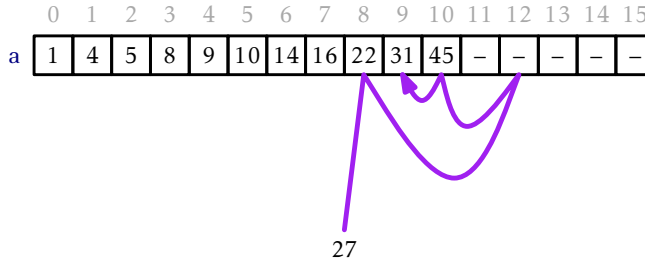


Figura 14.4: A execução de `findIt(a,27)`.

```
      z = u->keys[i];
    ui = u->children[i];
  }
  return z;
}
```

Central para o método `find(x)` é o método `findIt(a,x)` que pesquisa em um array ordenado completado com `null`, `a`, pelo valor `x`. Este método, ilustrado em Figura 14.4, funciona para qualquer array, `a`, onde $a[0],\ldots,a[k-1]$ é uma sequência de chaves em ordem classificada e $a[k],\ldots,a[a.length-1]$ estão todos configurados para `null`. Se `x` estiver na matriz na posição `i`, então `findIt(a,x)` retorna $-i-1$. Caso contrário, ele retorna o menor índice, `i`, de modo que $a[i] > x$ ou $a[i] = null$.

BTree
```
int findIt(array<T> &a, T x) {
  int lo = 0, hi = a.length;
  while (hi != lo) {
    int m = (hi+lo)/2;
    int cmp = a[m] == null ? -1 : compare(x, a[m]);
    if (cmp < 0)
      hi = m;       // look in first half
    else if (cmp > 0)
      lo = m+1;     // look in second half
    else
      return -m-1; // found it
  }
  return lo;
}
```

O método `findIt(a,x)` usa uma pesquisa binária que divide pela metade o espaço de pesquisa em cada etapa, para que seja executado em um tempo $O(\log(\texttt{a.length}))$. Em nosso ambiente, $\texttt{a.length} = 2B$, assim `findIt(a,x)` executa em tempo $O(\log B)$.

Podemos analisar o tempo de execução de uma operação `find(x)` em uma árvore B, tanto no modelo de palavra-RAM usual (onde cada instrução conta) quanto no modelo de memória externa (onde contamos apenas o número de nós acessados). Uma vez que cada folha em uma árvore B armazena pelo menos uma chave e a altura de uma árvore B com ℓ folhas é $O(\log_B \ell)$, a altura de uma árvore B que armazena `n` chaves é $O(\log_B \texttt{n})$. Portanto, no modelo de memória externa, o tempo gasto pela operação `find(x)` é $O(\log_B \texttt{n})$. Para determinar o tempo de execução no modelo palavra-RAM, temos que contabilizar o custo de chamar `findIt(a,x)` para cada nó que acessamos, portanto, o tempo de execução de `find(x)` no modelo de palavra-RAM é

$$O(\log_B \texttt{n}) \times O(\log B) = O(\log \texttt{n}) \ .$$

14.2.2 Adição

Uma diferença importante entre as árvores B e a estrutura de dados `BinarySearchTree` de Seção 6.2 é que os nós de uma árvore B não armazenam ponteiros para seus pais. A razão para isso será explicada em breve. A falta de ponteiros pai significa que as operações `add(x)` e `remove(x)` nas árvores B são mais facilmente implementadas usando recursão.

Como todas as árvores de pesquisa balanceadas, alguma forma de rebalanceamento é necessária durante uma operação `add(x)`. Em uma árvore B, isso é feito por *divisão* de nós. Referir-se à Figura 14.5 para o que segue. Embora a divisão ocorra em dois níveis de recursão, ela é melhor entendida como uma operação que pega um nó `u` contendo $2B$ chaves e tendo $2B+1$ filhos. Ele cria um novo nó, `w`, que adota $\texttt{u.children}[B],\ldots,\texttt{u.children}[2B]$ O novo nó `w` também pega as chaves maiores B de `u`, $\texttt{u.keys}B],\ldots,\texttt{u.keys}[2B-1]$. Neste ponto, `u` tem B filhos e B chaves. A chave extra, $\texttt{u.keys}[B-1]$, é passada para o pai de `u`, que também adota `w`.

Observe que a operação de divisão modifica três nós: `u`, `u` pai e o novo nó, `w`. É por isso que é importante que os nós de uma árvore B não mante-

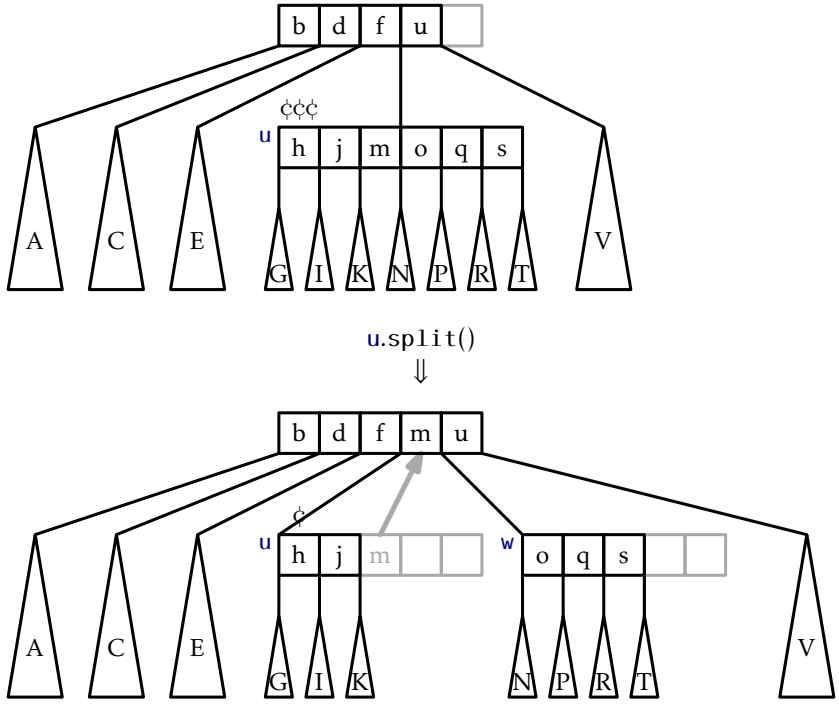


Figura 14.5: Dividindo o nó u em uma árvore B ($B = 3$). Observe que a chave u.keys[2] = m passa de u para seu pai.

nham ponteiros para os pais. Se o fizessem, então os $B+1$ filhos adotados por w precisariam ter seus ponteiros para os pais modificados. Isso aumentaria o número de acessos à memória externa de 3 para $B+4$ e tornaria as árvores B muito menos eficientes para grandes valores de B.

O método add(x) em uma árvore B é ilustrado em Figura 14.6. Em um nível superior, este método encontra uma folha, u, na qual adicionar o valor x. Se isso fizer com que u fique cheio demais (porque já continha $B-1$ chaves), então u será dividido. Se isso fizer com que o pai de u fique cheio demais, então o pai de u também é dividido, o que pode fazer com que o avô de u fique cheio demais, e assim por diante. Este processo continua, subindo na árvore um nível de cada vez até chegar a um nó que não está lotado ou até que a raiz seja dividida. No primeiro caso, o processo para. No último caso, uma nova raiz é criada cujos dois filhos se

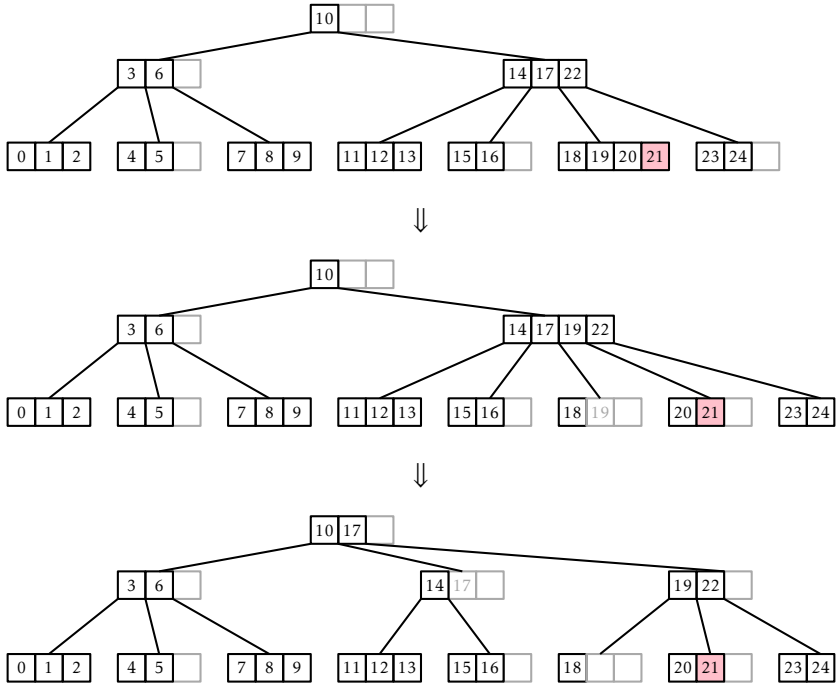


Figura 14.6: A operação `add(x)` em uma `BTree`. Adicionar o valor 21 resulta na divisão de dois nós.

tornam os nós obtidos quando a raiz original foi dividida.

O resumo da execução do método `add(x)` é que ele caminha da raiz para uma folha em busca de `x`, adiciona `x` a esta folha e, em seguida, sobe de volta para a raiz, dividindo quaisquer nós excessivamente cheios que encontrar ao longo do caminho. Com essa visão de alto nível em mente, agora podemos nos aprofundar nos detalhes de como esse método pode ser implementado recursivamente.

O verdadeiro trabalho de `add(x)` é feito pelo método `addRecursive(x,ui)`, que adiciona o valor `x` à subárvore cuja raiz, `u`, tem o identificador `ui`. Se `u` for uma folha, então `x` é simplesmente inserido em `u.keys`. Caso contrário, `x` é adicionado recursivamente no filho apropriado, `u'`, de `u`. O resultado dessa chamada recursiva é normalmente `null`, mas também pode ser uma referência a um nó recém-criado, `w`, que foi criado porque `u'` foi

dividido. Neste caso, u adota w e pega sua primeira chave, completando a operação de divisão em u'.

Após o valor x ter sido adicionado (para u ou para um descendente de u), o método addRecursive(x,ui) verifica se u está armazenando muitas (mais de $2B-1$) chaves. Se sim, então u precisa ser *dividido* com uma chamada para o método u.split(). O resultado de chamar u.split() é um novo nó que é usado como o valor de retorno para addRecursive(x,ui).

BTree
```
Node* addRecursive(T x, int ui) {
  Node *u = bs.readBlock(ui);
  int i = findIt(u->keys, x);
  if (i < 0) throw(-1);
  if (u->children[i] < 0) { // leaf node, just add it
    u->add(x, -1);
    bs.writeBlock(u->id, u);
  } else {
    Node* w = addRecursive(x, u->children[i]);
    if (w != NULL) {  // child was split, w is new child
      x = w->remove(0);
      bs.writeBlock(w->id, w);
      u->add(x, w->id);
      bs.writeBlock(u->id, u);
    }
  }
  return u->isFull() ? u->split() : NULL;
}
```

O método addRecursive(x,ui) é um auxiliar para o método add(x), que chama addRecursive(x,ri) para inserir x na raiz da árvore B. Se addRecursive(x,ri) faz com que a raiz se divida, então uma nova raiz é criada e recebe como seus filhos a raiz antiga e o novo nó criado pela divisão da raiz antiga.

BTree
```
bool add(T x) {
       Node *w;
       try {
         w = addRecursive(x, ri);
       } catch (int e) {
         return false; // adding duplicate value
       }
```

```
        if (w != NULL) {   // root was split, make new root
    Node *newroot = new Node(this);
    x = w->remove(0);
    bs.writeBlock(w->id, w);
    newroot->children[0] = ri;
    newroot->keys[0] = x;
    newroot->children[1] = w->id;
    ri = newroot->id;
    bs.writeBlock(ri, newroot);
        }
        n++;
        return true;
}
```

O método add(x) e seu auxiliar, addRecursive(x,ui), podem ser analisados em duas fases:

Fase descendente: Durante a fase descendente da recursão, antes de x ser adicionado, eles acessam uma sequência de BTree nós e chamam findIt(a,x) em cada nó. Tal como acontece com o método find(x), isso leva um tempo $O(\log_B n)$ no modelo de memória externa e um tempo $O(\log n)$ no modelo palavra-RAM.

Fase ascendente: Durante a fase ascendente da recursão, após a adição de x, esses métodos executam uma sequência de no máximo $O(\log_B n)$ divisões. Cada divisão envolve apenas três nós, portanto, esta fase leva um tempo$O(\log_B n)$ no modelo de memória externa. No entanto, cada divisão envolve mover B chaves e filhos de um nó para outro, portanto, no modelo de palavra-RAM, isso leva um tempo $O(B \log n)$.

Lembre-se de que o valor de B pode ser muito grande, muito maior do que $\log n$. Portanto, no modelo de palavra-RAM, adicionar um valor a uma árvore B pode ser muito mais lento do que adicionar em uma árvore de pesquisa binária balanceada. Posteriormente, em Seção 14.2.4, mostraremos que a situação não é tão ruim; o número amortizado de operações de divisão feitas durante uma operação add(x) é constante. Isso mostra que o tempo de execução (amortizado) da operação add(x) no modelo palavra-RAM é $O(B + \log n)$.

14.2.3 Remoção

A operação `remove(x)` em uma `BTree` é, novamente, mais facilmente implementada como um método recursivo. Embora a implementação recursiva de `remove(x)` espalhe a complexidade por vários métodos, o processo geral, que é ilustrado em Figura 14.7, é bastante direto. Ao embaralhar as chaves, a remoção é reduzida ao problema de remover um valor, `x'`, de alguma folha, `u`. Remover `x'` pode deixar `u` com menos de $B-1$ chaves; esta situação é chamada de *underflow*.

Quando ocorre um estouro negativo (underflow), `u` pega as chaves emprestadas ou é mesclado com um de seus irmãos. Se `u` for mesclado com um irmão, o pai de `u` agora terá um filho a menos e uma chave a menos, o que pode fazer com que o pai de `u` entre em *underflow*; isso é corrigido novamente pedindo emprestado ou mesclando, mas a mesclagem pode fazer com que o avô de `u` fique em *underflow*. Esse processo retorna à raiz até que não haja mais *underflow* ou até que a raiz tenha seus dois últimos filhos mesclados em um único filho. Quando ocorre o último caso, a raiz é removida e seu filho único se torna a nova raiz.

A seguir, nos aprofundamos nos detalhes de como cada uma dessas etapas é implementada. A primeira tarefa do método `remove(x)` é encontrar o elemento `x` que deve ser removido. Se `x` for encontrado em uma folha, então `x` será removido dessa folha. Caso contrário, se `x` for encontrado em `u.keys[i]` para algum nó interno, `u`, então o algoritmo remove o menor valor, `x'`, na subárvore enraizada em `u.children[i+1]`. O valor `x'` é o menor valor armazenado na `BTree` que é maior que `x`. O valor de `x'` é então usado para substituir `x` em `u.keys[i]`. Este processo é ilustrado na Figura 14.8.

O método `removeRecursive(x,ui)` é uma implementação recursiva do algoritmo anterior:

BTree

```
T removeSmallest(int ui) {
  Node* u = bs.readBlock(ui);
  if (u->isLeaf())
    return u->remove(0);
  T y = removeSmallest(u->children[0]);
  checkUnderflow(u, 0);
  return y;
```

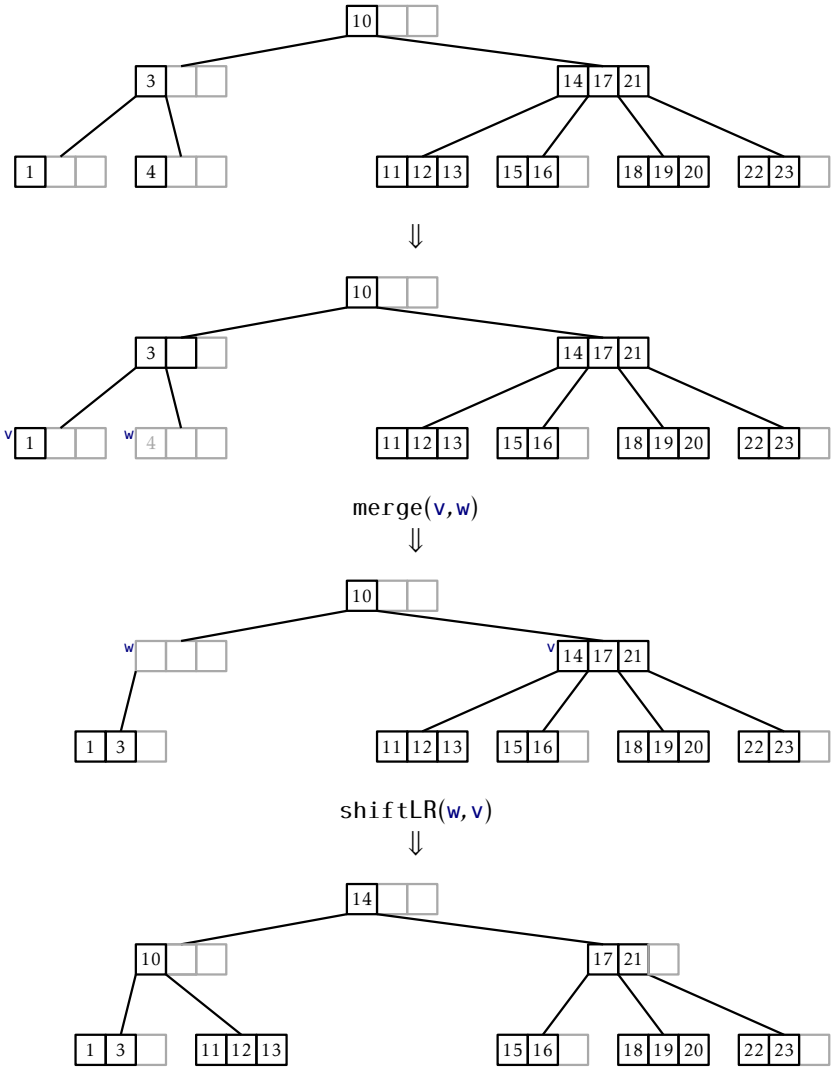


Figura 14.7: Remover o valor 4 de uma árvore *B* resulta em uma fusão e uma operação de empréstimo.

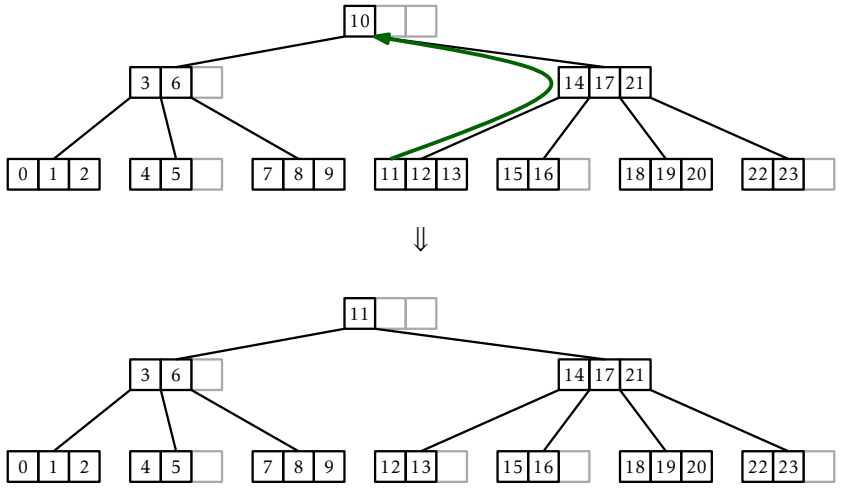


Figura 14.8: A operação remove(x) em uma BTree. Para remover o valor x = 10, nós o substituímos pelo valor x′ = 11 e removemos 11 da folha que o contém.

```
}
bool removeRecursive(T x, int ui) {
  if (ui < 0) return false;  // didn't find it
  Node* u = bs.readBlock(ui);
  int i = findIt(u->keys, x);
  if (i < 0) { // found it
    i = -(i+1);
    if (u->isLeaf()) {
      u->remove(i);
    } else {
      u->keys[i] = removeSmallest(u->children[i+1]);
      checkUnderflow(u, i+1);
    }
    return true;
  } else if (removeRecursive(x, u->children[i])) {
    checkUnderflow(u, i);
    return true;
  }
  return false;
}
```

Observe que, após remover recursivamente o valor x do i-ésimo filho de u, removeRecursive(x,ui) precisa garantir que esse filho ainda tenha pelo menos $B-1$ chaves. No código anterior, isso é feito usando um método chamado checkUnderflow(x,i), que verifica e corrige um estouro negativo no i-ésimo filho de u. Seja w o i-ésimo filho de u. Se w tiver apenas chaves $B-2$, isso precisa ser corrigido. A correção requer o uso de um irmão de w. Pode ser o filho $i+1$ de u ou o filho $i-1$ de u. Normalmente usaremos o filho $i-1$ de u, que é o irmão, v, de w diretamente à sua esquerda. A única vez que isso não funciona é quando $i=0$, caso em que usamos o irmão diretamente à direita de w.

BTree
```
void checkUnderflow(Node* u, int i) {
  if (u->children[i] < 0) return;
  if (i == 0)
    checkUnderflowZero(u, i); // use u's right sibling
  else
    checkUnderflowNonZero(u, i);
}
```

A seguir, nos concentramos no caso em que $i \neq 0$ de modo que qualquer underflow no i-ésimo filho de u seja corrigido com a ajuda do filho $(i-1)$ de u. O caso $i=0$ é semelhante e os detalhes podem ser encontrados no código-fonte que o acompanha.

Para corrigir um estouro negativo no nó w, precisamos encontrar mais chaves (e possivelmente também filhos), para w. Existem duas maneiras de fazer isso:

Pedindo emprestado: Se w tiver um irmão, v, com mais de $B-1$ chaves, então w pode emprestar algumas chaves (e possivelmente também filhos) de v. Mais especificamente, se v armazena size(v) chaves, então, entre elas, v e w têm um total de

$$B-2+\texttt{size}(\texttt{w}) \geq 2B-2$$

chaves. Podemos, portanto, mudar as chaves de v para w de modo que cada um de v e w tenha pelo menos $B-1$ chaves. Este processo é ilustrado na Figura 14.9.

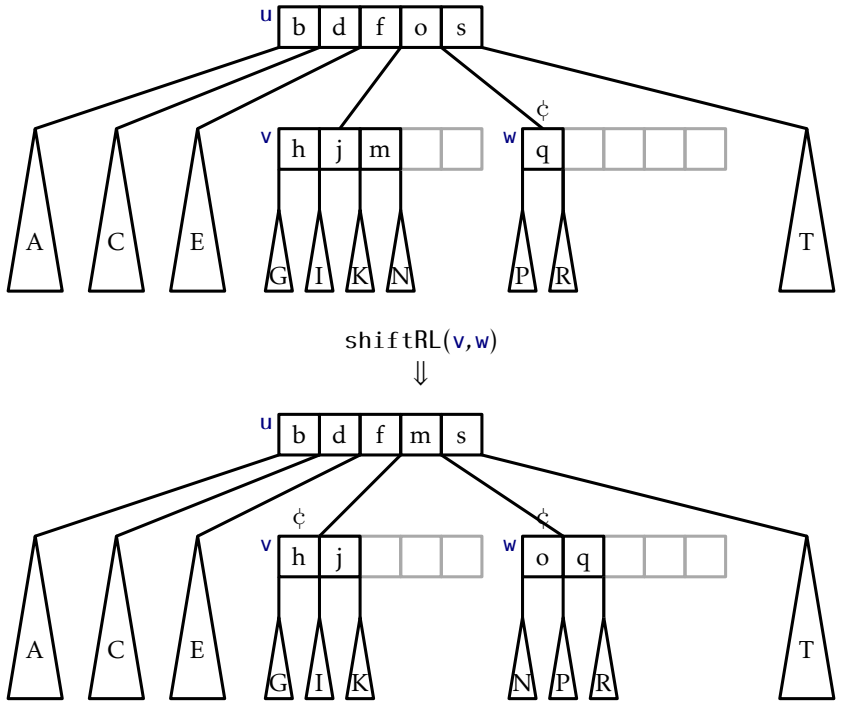


Figura 14.9: Se v tem mais que $B-1$ chaves, então w pedir chaves emprestado a v.

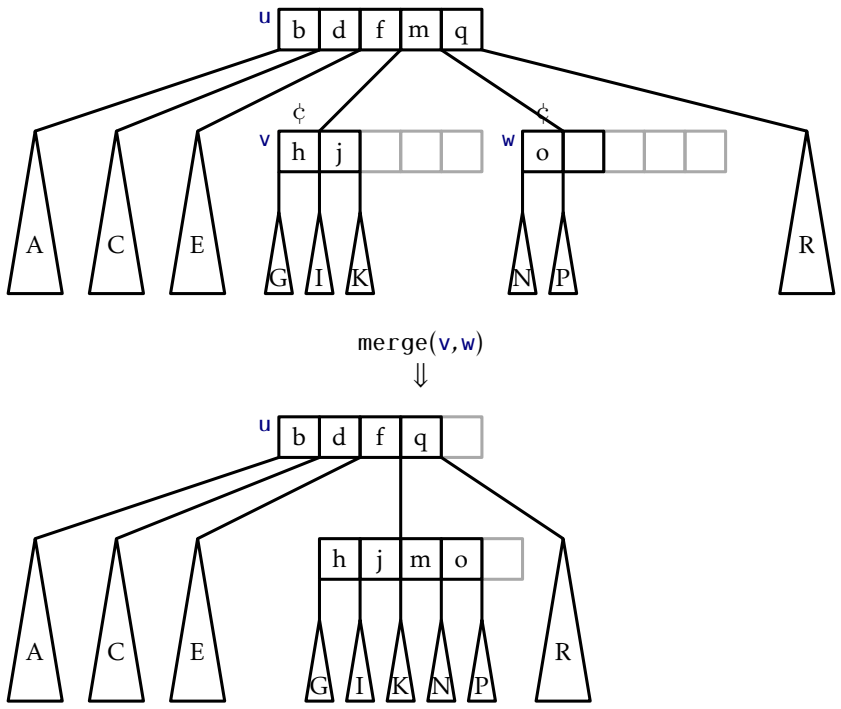


Figura 14.10: Mesclando dois irmãos v e w em uma árvore B ($B = 3$).

Mesclando: Se v tiver apenas chaves $B-1$, devemos fazer algo mais drástico, já que v não pode se dar ao luxo de fornecer chaves para w. Portanto, *mesclamos* v e w como mostrado na Figura 14.10. A operação de mesclagem é o oposto da operação de divisão. Ele pega dois nós que contêm um total de $2B-3$ chaves e os mescla em um único nó que contém $2B-2$ chaves. (A chave adicional vem do fato de que, quando mesclamos v e w, seu pai comum, u, agora tem um filho a menos e, portanto, precisa desistir de uma de suas chaves.)

BTree

```
void checkUnderflowZero(Node *u, int i) {
  Node *w = bs.readBlock(u->children[i]);
  if (w->size() < B-1) {  // underflow at w
    Node *v = bs.readBlock(u->children[i+1]);
```

```
    if (v->size() > B) { // w can borrow from v
      shiftRL(u, i, v, w);
    } else { // w will absorb w
      merge(u, i, w, v);
      u->children[i] = w->id;
    }
  }
}
void checkUnderflowNonZero(Node *u, int i) {
  Node *w = bs.readBlock(u->children[i]);
  if (w->size() < B-1) {  // underflow at w
    Node *v = bs.readBlock(u->children[i-1]);
    if (v->size() > B) {  // w can borrow from v
      shiftLR(u, i-1, v, w);
    } else { // v will absorb w
      merge(u, i-1, v, w);
    }
  }
}
```

Para resumir, o método `remove(x)` em uma árvore B segue um caminho da raiz para a folha, remove uma chave `x`' de uma folha, `u` e, em seguida, executa zero ou mais operações de mesclagem envolvendo `u` e seus ancestrais e executa no máximo uma operação de empréstimo. Como cada operação de mesclagem e empréstimo envolve a modificação de apenas três nós, e apenas $O(\log_B n)$ dessas operações ocorrem, todo o processo leva um tempo $O(\log_B n)$ no modelo de memória externa. Novamente, no entanto, cada operação de mesclagem e empréstimo leva um tempo $O(B)$ no modelo de palavra-RAM, então (por enquanto) o máximo que podemos dizer sobre o tempo de execução exigido por `remove(x)` no modelo de palavra-RAM é $O(B \log_B n)$.

14.2.4 Análise Amortizada de Árvores B

Até agora, mostramos que

1. No modelo de memória externa, o tempo de execução de `find(x)`, `add(x)` e `remove(x)` em uma árvore B é $O(\log_B n)$.
2. No modelo palavra-RAM, o tempo de execução de `find(x)` é $O(\log n)$

e o tempo de execução de add(x) e remove(x) é $O(B \log n)$.

O seguinte lema mostra que, até agora, superestimamos o número de operações de mesclagem e divisão realizadas por árvores *B*.

Lema 14.1. *Começar com uma árvore B vazia e executar qualquer sequência de m operações* add(x) *e* remove(x) *resulta em no máximo* $3m/2$ *divisões, mesclagens e empréstimos sendo executados.*

Demonstração. A prova disso já foi esboçada em Seção 9.3 para o caso especial em que $B = 2$. O lema pode ser comprovado usando um esquema de crédito, no qual

1. cada operação de divisão, fusão ou empréstimo é paga com dois créditos, ou seja, um crédito é retirado cada vez que uma dessas operações ocorre; e

2. no máximo três créditos são criados durante qualquer operação add(x) ou remove(x).

Como no máximo $3m$ créditos são criados e cada divisão, fusão e empréstimo é pago com dois créditos, segue-se que no máximo $3m/2$ de divisões, fusões e empréstimos são realizados. Esses créditos são ilustrados usando o símbolo ¢ nas Figuras 14.5, 14.9, e 14.10.

Para acompanhar esses créditos, a prova mantém o seguinte *invariante de crédito*: Qualquer nó não raiz com $B-1$ chaves armazena um crédito e qualquer nó com $2B-1$ chaves armazena três créditos. Um nó que armazena pelo menos *B* chaves e a maioria das $2B-2$ chaves não precisa armazenar nenhum crédito. O que falta é mostrar que podemos manter a invariante de crédito e satisfazer as propriedades 1 e 2, acima, durante cada operação add(x) e remove(x).

Adicionando: O método add(x) não realiza mesclagens ou empréstimos, portanto, precisamos apenas considerar as operações de divisão que ocorrem como resultado de chamadas para add(x).

Cada operação de divisão ocorre porque uma chave é adicionada a um nó, u, que já contém $2B-1$ chaves. Quando isso acontece, u é dividido em dois nós, u′ e u″ tendo $B-1$ e *B* chaves, respectivamente. Antes desta operação, u estava armazenando $2B-1$ chaves e, portanto, três créditos.

Dois desses créditos podem ser usados para pagar a divisão e o outro crédito pode ser dado a u′ (que tem $B-1$ chaves) para manter a invariante de crédito. Portanto, podemos pagar pela divisão e manter a invariante de crédito durante qualquer divisão.

A única outra modificação nos nós que ocorre durante uma operação add(x) ocorre depois que todas as divisões, se houver, forem concluídas. Esta modificação envolve a adição de uma nova chave a algum nó u′. Se, antes disso, u′ tinha $2B-2$ filhos, agora tem $2B-1$ filhos e deve, portanto, receber três créditos. Estes são os únicos créditos dados pelo método add(x).

Removendo: Durante uma chamada para remove(x), zero ou mais mesclagens ocorrem e são possivelmente seguidas por um único empréstimo. Cada mesclagem ocorre porque dois nós, v e w, cada um dos quais tinha exatamente $B-1$ chaves antes de chamar remove(x), foram mesclados em um único nó com exatamente $2B-2$ chaves. Cada mesclagem, portanto, libera dois créditos que podem ser usados para pagar pela fusão.

Depois que quaisquer fusões são realizadas, no máximo uma operação de empréstimo ocorre, após a qual não ocorrem mais fusões ou empréstimos. Esta operação de empréstimo ocorre apenas se removermos uma chave de uma folha, v, que possui $B-1$ chaves. O nó v, portanto, tem um crédito, e esse crédito vai para o custo do empréstimo. Esse único crédito não é suficiente para pagar o empréstimo, então criamos um crédito para completar o pagamento.

Neste ponto, criamos um crédito e ainda precisamos mostrar que a invariante de crédito pode ser mantida. No pior caso, o irmão de v, w, tem exatamente B chaves antes do empréstimo, de forma que, depois, tanto v quanto w têm $B-1$ chaves. Isso significa que v e w cada um deve armazenar um crédito quando a operação for concluída. Portanto, neste caso, criamos dois créditos adicionais para dar a v e w. Como um empréstimo acontece no máximo uma vez durante uma operação remove(x), isso significa que criamos no máximo três créditos, conforme necessário.

Se a operação remove(x) não inclui uma operação de empréstimo, é porque termina removendo uma chave de algum nó que, antes da operação, tinha B ou mais chaves. No pior caso, este nó tinha exatamente B chaves, de modo que agora tem $B-1$ chaves e deve receber um crédito,

que criamos.

Em ambos os casos — quer a remoção termine com uma operação de empréstimo ou não — no máximo três créditos precisam ser criados durante uma chamada para `remove(x)` para manter a invariante de crédito e pagar por todos os empréstimos e mesclagens que ocorrer. Isso completa a prova do lema. □

O objetivo de Lema 14.1 é mostrar que, no modelo de palavra-RAM, o custo de divisões, fusões e junções durante uma sequência de m operações `add(x)` e `remove(x)` é apenas $O(Bm)$. Ou seja, o custo amortizado por operação é apenas $O(B)$, então o custo amortizado de `add(x)` e `remove(x)` no modelo de palavra-RAM é $O(B + \log n)$. Isso é resumido pelo seguinte par de teoremas:

Teorema 14.1 (Árvore B em Memória Externa)**.** *Uma BTree implementa a interface SSet. No modelo de memória externa, uma BTree suporta as operações* `add(x)`, `remove(x)` *e* `find(x)` *com tempo* $O(\log_B n)$ *por operação.*

Teorema 14.2 (Árvores B em Palavra-RAM)**.** *Uma BTree implementa a interface SSet. No modelo de palavra-RAM, e ignorando o custo de divisões, fusões e empréstimos, uma BTree oferece suporte às operações* `add(x)`, `remove(x)` *e* `find(x)` *em tempo* $O(\log n)$ *por operação. Além disso, começando com uma BTree vazia, qualquer sequência de m operações* `add(x)` *e* `remove(x)` *resulta em um total de tempo* $O(Bm)$ *gasto para realizar divisões, fusões e empréstimos.*

14.3 Discussão e Exercícios

O modelo de computação de memória externa foi introduzido por Aggarwal e Vitter [4]. Às vezes também é chamado de *modelo de E/S* ou de *modelo de acesso ao disco*.

As árvores B estão para a pesquisa de memória externa o que as árvores de pesquisa binárias estão para a pesquisa de memória interna. As árvores B foram introduzidos por Bayer e McCreight [9] em 1970 e, menos de dez anos depois, o título do artigo da ACM Computing Surveys de Comer referia-se a elas como ubíquas [15].

Como árvores de busca binárias, existem muitas variantes de árvores B, incluindo árvores B^+, árvores B^*, e árvores contadas B. Árvores B são realmente onipresentes e são a estrutura de dados primária em muitos sistemas de arquivos, incluindo o HFS+ da Apple, o NTFS da Microsoft, e o Ext4 do Linux; todos os principais sistemas de banco de dados; e armazenamentos de valores-chave usados na computação em nuvem. A pesquisa recente de Graefe [36] fornece uma visão geral de mais de 200 páginas de muitos aplicativos modernos, variantes e otimizações de árvores B.

Árvores B implementam a interface SSet. Se apenas a interface USet for necessária, o hashing da memória externa poderia ser usado como uma alternativa às árvores B. Existem esquemas de hashing de memória externa; veja, por exemplo, Jensen e Pagh [43]. Esses esquemas implementam as operações USet em tempo esperado $O(1)$ no modelo de memória externa. No entanto, por vários motivos, muitos aplicativos ainda usam árvores B, embora requeiram apenas operações USet.

Uma das razões pelas quais as árvores B são uma escolha tão popular é que frequentemente têm um desempenho melhor do que seus limites de tempo de execução $O(\log_B \mathtt{n})$ sugerem. A razão para isso é que, em configurações de memória externa, o valor de B é normalmente muito grande – na casa das centenas ou mesmo milhares. Isso significa que 99% ou mesmo 99,9% dos dados em uma árvore B são armazenados nas folhas. Em um sistema de banco de dados com grande memória, pode ser possível armazenar em cache todos os nós internos de uma árvore B na RAM, pois eles representam apenas 1% ou 0,1% do conjunto de dados total. Quando isso acontece, significa que uma busca em uma árvore B envolve uma busca muito rápida na RAM, através dos nós internos, seguida por um único acesso à memória externa para recuperar uma folha.

Exercício 14.1. Mostre o que acontece quando as chaves 1.5 e 7.5 são adicionadas à árvore B na Figura 14.2.

Exercício 14.2. Mostre o que acontece quando as chaves 3 e 4 são removidas da árvore B na Figura 14.2.

Exercício 14.3. Qual é o número máximo de nós internos em uma árvore B que armazena n chaves (como uma função de n e B)?

Exercício 14.4. A introdução a este capítulo afirma que árvores B preci-

sam apenas de uma memória interna de tamanho $O(B + \log_B n)$. No entanto, a implementação fornecida aqui realmente requer mais memória.

1. Mostre que a implementação dos métodos `add(x)` e `remove(x)` dados neste capítulo usam uma memória interna proporcional a $B \log_B n$.

2. Descreva como esses métodos podem ser modificados a fim de reduzir o consumo de memória para $O(B + \log_B n)$.

Exercício 14.5. Desenhe os créditos usados na prova de Lema 14.1 nas árvores nas Figuras 14.6 e 14.7. Verifique se (com três créditos adicionais) é possível pagar pelas divisões, fusões e empréstimos e manter a invariante de crédito.

Exercício 14.6. Projete uma versão modificada de uma árvore B na qual os nós podem ter de B até $3B$ filhos (e, portanto, $B-1$ até $3B-1$ chaves). Mostre que esta nova versão de árvores B realiza apenas $O(m/B)$ divisões, mesclas e empréstimos durante uma sequência de m operações. (Dica: para que isso funcione, você terá que ser mais agressivo com a mesclagem, às vezes mesclando dois nós antes que seja estritamente necessário.)

Exercício 14.7. Neste exercício, você projetará um método modificado de divisão e fusão em árvores B que reduz assintoticamente o número de divisões, empréstimos e fusões, considerando até três nós por vez.

1. Seja `u` um nó cheio e seja `v` um irmão imediatamente à direita de `u`. Existem duas maneiras de corrigir o estouro em `u`:

   (a) `u` pode fornecer algumas de suas chaves para `v`; ou

   (b) `u` podem ser divididas e as chaves de `u` e `v` podem ser distribuídas uniformemente entre `u`, `v` e o nó recém-criado, `w`.

   Mostre que isso sempre pode ser feito de forma que, após a operação, cada um dos (no máximo 3) nós afetados tenha pelo menos $B + \alpha B$ chaves e no máximo $2B - \alpha B$ chaves, para alguma constante $\alpha > 0$.

2. Faça `u` ser um nó em estouro negativo e faça `v` e `w` serem irmãos de `u`. Existem duas maneiras de corrigir o estouro negativo em `u`:

(a) as chaves podem ser redistribuídas entre `u`, `v`, e `w`; ou

(b) `u`, `v`, e `w` podem ser mesclados em dois nós e as chaves de `u`, `v` e `w` podem ser redistribuídas entre esses nós.

Mostre que isso sempre pode ser feito de forma que, após a operação, cada um dos (no máximo 3) nós afetados tenha pelo menos $B + \alpha B$ chaves e no máximo $2B - \alpha B$ chaves, para alguma constante $\alpha > 0$.

3. Mostre que, com essas modificações, o número de fusões, empréstimos e divisões que ocorrem durante as m operações é $O(m/B)$.

Exercício 14.8. Uma árvore B^+ ilustrada na Figura 14.11 armazena cada chave em uma folha e mantém suas folhas armazenadas como uma lista duplamente encadeada. Como de costume, cada folha armazena entre $B - 1$ e $2B - 1$ chaves. Acima desta lista está uma árvore B padrão que armazena o maior valor de cada folha, exceto o último.

1. Descreva implementações rápidas de `add(x)`, `remove(x)` e `find(x)` em uma árvore B^+.

2. Explique como implementar eficientemente o método `findRange(x,y)`, que relata todos os valores maiores que `x` e menores ou iguais a `y`, em uma árvore B^+.

3. Implemente uma classe, `BPlusTree`, que implementa `find(x)`, `add(x)`, `remove(x)`, e `findRange(x,y)`.

4. Árvores B^+ duplicam algumas das chaves porque elas são armazenadas tanto na árvore B quanto na lista. Explique por que essa duplicação não soma muito para grandes valores de B.

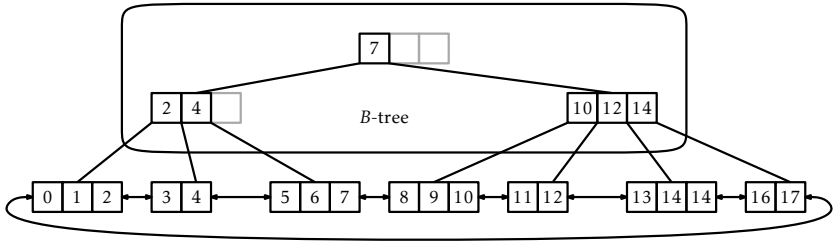


Figura 14.11: Uma árvore B^+ é uma árvore B no topo de uma lista duplamente encadeada de blocos.

Referências Bibliográficas

[1] Free eBooks by Project Gutenberg. URL: http://www.gutenberg.org/ [cited 2011-10-12].

[2] IEEE Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, August 2008. doi:10.1109/IEEESTD.2008.4610935.

[3] G. Adelson-Velskii and E. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3(1259-1262):4, 1962.

[4] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[5] A. Andersson. Improving partial rebuilding by using simple balance criteria. In F. K. H. A. Dehne, J.-R. Sack, and N. Santoro, editors, *Algorithms and Data Structures, Workshop WADS '89, Ottawa, Canada, August 17–19, 1989, Proceedings*, volume 382 of *Lecture Notes in Computer Science*, pages 393–402. Springer, 1989.

[6] A. Andersson. Balanced search trees made simple. In F. K. H. A. Dehne, J.-R. Sack, N. Santoro, and S. Whitesides, editors, *Algorithms and Data Structures, Third Workshop, WADS '93, Montréal, Canada, August 11–13, 1993, Proceedings*, volume 709 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 1993.

[7] A. Andersson. General balanced trees. *Journal of Algorithms*, 30(1):1–18, 1999.

[8] A. Bagchi, A. L. Buchsbaum, and M. T. Goodrich. Biased skip lists. In P. Bose and P. Morin, editors, *Algorithms and Computation, 13th International Symposium, ISAAC 2002 Vancouver, BC, Canada, November 21–23, 2002, Proceedings*, volume 2518 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2002.

[9] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. In *SIGFIDET Workshop*, pages 107–141. ACM, 1970.

[10] Bibliography on hashing. URL: http://liinwww.ira.uka.de/bibliography/Theory/hash.html [cited 2011-07-20].

[11] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In M. J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15–19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 79–79. Springer, 1999.

[12] P. Bose, K. Douïeb, and S. Langerman. Dynamic optimality for skip lists and b-trees. In S.-H. Teng, editor, *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20–22, 2008*, pages 1106–1114. SIAM, 2008.

[13] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick. Resizable arrays in optimal time and space. In Dehne et al. [18], pages 37–48.

[14] J. Carter and M. Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.

[15] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[16] C. Crane. Linear lists and priority queues as balanced binary trees. Technical Report STAN-CS-72-259, Computer Science Department, Stanford University, 1972.

[17] S. Crosby and D. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*, pages 29–44, 2003.

[18] F. K. H. A. Dehne, A. Gupta, J.-R. Sack, and R. Tamassia, editors. *Algorithms and Data Structures, 6th International Workshop, WADS '99, Vancouver, British Columbia, Canada, August 11–14, 1999, Proceedings*, volume 1663 of *Lecture Notes in Computer Science*. Springer, 1999.

[19] L. Devroye. Applications of the theory of records in the study of random trees. *Acta Informatica*, 26(1):123–130, 1988.

[20] P. Dietz and J. Zhang. Lower bounds for monotonic list labeling. In J. R. Gilbert and R. G. Karlsson, editors, *SWAT 90, 2nd Scandinavian Workshop on Algorithm Theory, Bergen, Norway, July 11–14, 1990, Proceedings*, volume 447 of *Lecture Notes in Computer Science*, pages 173–180. Springer, 1990.

[21] M. Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In C. Puech and R. Reischuk, editors, *STACS 96, 13th Annual Symposium on Theoretical Aspects of Computer Science, Grenoble, France, February 22–24, 1996, Proceedings*, volume 1046 of *Lecture Notes in Computer Science*, pages 567–580. Springer, 1996.

[22] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In W. Kuich, editor, *Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria, July 13–17, 1992, Proceedings*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246. Springer, 1992.

[23] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997.

[24] M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. M. auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

[25] A. Elmasry. Pairing heaps with $O(\log \log n)$ decrease cost. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 471–476. Society for Industrial and Applied Mathematics, 2009.

[26] F. Ergun, S. C. Sahinalp, J. Sharp, and R. Sinha. Biased dictionaries with fast insert/deletes. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 483–491, New York, NY, USA, 2001. ACM.

[27] M. Eytzinger. *Thesaurus principum hac aetate in Europa viventium (Cologne)*. 1590. In commentaries, 'Eytzinger' may appear in variant forms, including: Aitsingeri, Aitsingero, Aitsingerum, Eyzingern.

[28] R. W. Floyd. Algorithm 245: Treesort 3. *Communications of the ACM*, 7(12):701, 1964.

[29] M. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[30] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[31] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with 0 (1) worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[32] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of computer and system sciences*, 47(3):424–436, 1993.

[33] I. Galperin and R. Rivest. Scapegoat trees. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 165–174. Society for Industrial and Applied Mathematics, 1993.

[34] A. Gambin and A. Malinowski. Randomized meldable priority queues. In *SOFSEM'98: Theory and Practice of Informatics*, pages 344–349. Springer, 1998.

[35] M. T. Goodrich and J. G. Kloss. Tiered vectors: Efficient dynamic arrays for rank-based sequences. In Dehne et al. [18], pages 205–216.

[36] G. Graefe. Modern b-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2010.

[37] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 2nd edition, 1994.

[38] L. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, 16–18 October 1978, Proceedings*, pages 8–21. IEEE Computer Society, 1978.

[39] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.

[40] J. E. Hopcroft and R. E. Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[41] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, 1974.

[42] HP-UX process management white paper, version 1.3, 1997. URL: `http://h21007.www2.hp.com/portal/download/files/prot/files/STK/pdfs/proc_mgt.pdf` [cited 2011-07-20].

[43] M. S. Jensen and R. Pagh. Optimality in external memory hashing. *Algorithmica*, 52(3):403–411, 2008.

[44] P. Kirschenhofer, C. Martinez, and H. Prodinger. Analysis of an optimized search algorithm for skip lists. *Theoretical Computer Science*, 144:199–220, 1995.

[45] P. Kirschenhofer and H. Prodinger. The path length of random skip lists. *Acta Informatica*, 31:775–792, 1994.

[46] D. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.

[47] D. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.

[48] D. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1997.

[49] C. Y. Lee. An algorithm for path connection and its applications. *IRE Transaction on Electronic Computers*, EC-10(3):346–365, 1961.

[50] E. Lehman, F. T. Leighton, and A. R. Meyer. *Mathematics for Computer Science*. 2011. URL: http://people.csail.mit.edu/meyer/mcs.pdf [cited 2012-09-06].

[51] C. Martínez and S. Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, 1998.

[52] E. F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292, 1959.

[53] J. I. Munro, T. Papadakis, and R. Sedgewick. Deterministic skip lists. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms (SODA'92)*, pages 367–375, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[54] Oracle. *The Collections Framework*. URL: http://download.oracle.com/javase/1.5.0/docs/guide/collections/ [cited 2011-07-19].

[55] R. Pagh and F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[56] T. Papadakis, J. I. Munro, and P. V. Poblete. Average search and update costs in skip lists. *BIT*, 32:316–332, 1992.

[57] M. Pătraşcu and M. Thorup. Randomization does not help searching predecessors. In N. Bansal, K. Pruhs, and C. Stein, editors, *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7–9, 2007*, pages 555–564. SIAM, 2007.

[58] M. Pătraşcu and M. Thorup. The power of simple tabulation hashing. *Journal of the ACM*, 59(3):14, 2012.

[59] W. Pugh. A skip list cookbook. Technical report, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, 1989. URL: ftp://ftp.cs.umd.edu/pub/skipLists/cookbook.pdf [cited 2011-07-20].

[60] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[61] Redis. URL: http://redis.io/ [cited 2011-07-20].

[62] B. Reed. The height of a random binary search tree. *Journal of the ACM*, 50(3):306–332, 2003.

[63] S. M. Ross. *Probability Models for Computer Science*. Academic Press, Inc., Orlando, FL, USA, 2001.

[64] R. Sedgewick. Left-leaning red-black trees, September 2008. URL: http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf [cited 2011-07-21].

[65] R. Seidel and C. Aragon. Randomized search trees. *Algorithmica*, 16(4):464–497, 1996.

[66] H. H. Seward. Information sorting in the application of electronic digital computers to business operations. Master's thesis, Massachusetts Institute of Technology, Digital Computer Laboratory, 1954.

[67] Z. Shao, J. H. Reppy, and A. W. Appel. Unrolling lists. In *Proceedings of the 1994 ACM conference LISP and Functional Programming (LFP'94)*, pages 185–195, New York, 1994. ACM.

[68] P. Sinha. A memory-efficient doubly linked list. *Linux Journal*, 129, 2005. URL: http://www.linuxjournal.com/article/6828 [cited 2013-06-05].

[69] SkipDB. URL: http://dekorte.com/projects/opensource/SkipDB/ [cited 2011-07-20].

[70] D. Sleator and R. Tarjan. Self-adjusting binary trees. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25–27 April, 1983, Boston, Massachusetts, USA*, pages 235–245. ACM, ACM, 1983.

[71] S. P. Thompson. *Calculus Made Easy*. MacMillan, Toronto, 1914. Project Gutenberg EBook 33283. URL: http://www.gutenberg.org/ebooks/33283 [cited 2012-06-14].

[72] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977.

[73] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

[74] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.

[75] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inf. Process. Lett.*, 17(2):81–84, 1983.

[76] J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

Índice Remissivo