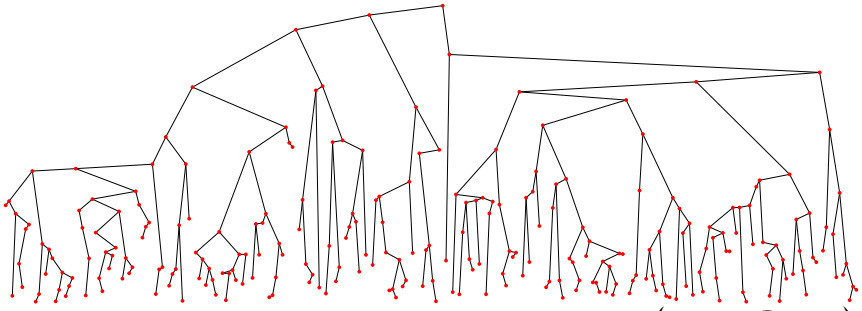


Estruturas de Dados Abertas(em C++)

Edição 0.1G β
30/10/2017

Pat Morin
tradução para o 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] e também, mais importante, em uma fonte confiável de gerenciamento de código.[2]

O código-fonte no site é disponível sob uma licença Creative Commons Attribution, o que significa que qualquer pessoa é livre para *compartilhar*: para copiar, distribuir e transmitir a obra; e para *remixar*: para

[1] http://opendatastructures.org
[2] https://github.com/patmorin/ods

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 podem 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.

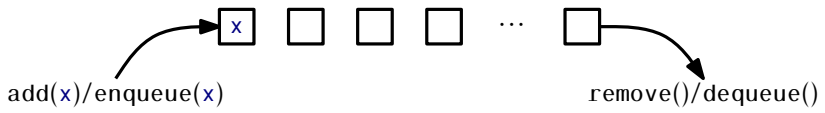


Figura 1.1: Uma Fila FIFO.

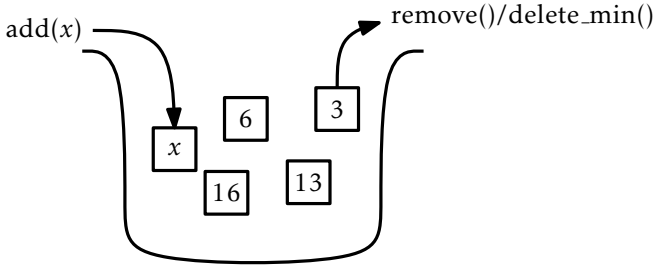


Figura 1.2: Uma Fila de Prioridade.

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 sequência ou na parte de trás da sequência. Os nomes das operações

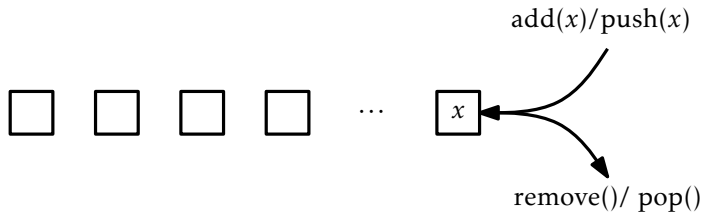


Figura 1.3: Uma Pilha.

de Deque são auto-explicativos: addFirst(x),removeFirst(), addLast(x) e removeLast(). Vale a pena notar que uma Pilha pode ser implementada usando apenas addFirst(x) e removeFirst(), 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 igua 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 $x_{i+1},\ldots,x_{n-1}$;
   Set $x_j = x_{j+1}$, para todo $j \in \{i,\ldots,n-2\}$ e decrementa n

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

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

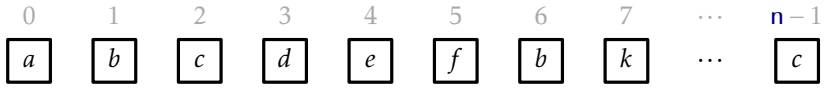


Figura 1.4: Uma Lista representa uma sequência indexada por $0, 1, 2, \ldots, n-1$. Nesta `Lista`, uma chamada para `get(2)` deve retornar o valor c.

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 for-

necida 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 [49].

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

$$T(\mathtt{n}) = a + b(\mathtt{n}+1) + c\mathtt{n} + d\mathtt{n} + e\mathtt{n} \ ,$$

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

$$T(\mathtt{n}) = O(\mathtt{n}) \ .$$

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* $\mathtt{n}$.

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 $f_1(\mathtt{n}) = 15\mathtt{n}$ versus $f_2(n) = 2\mathtt{n}\log\mathtt{n}$. 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 $f_1(\mathtt{n})$ seja maior que $f_2(n)$ para valores pequenos de $\mathtt{n}$, 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 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

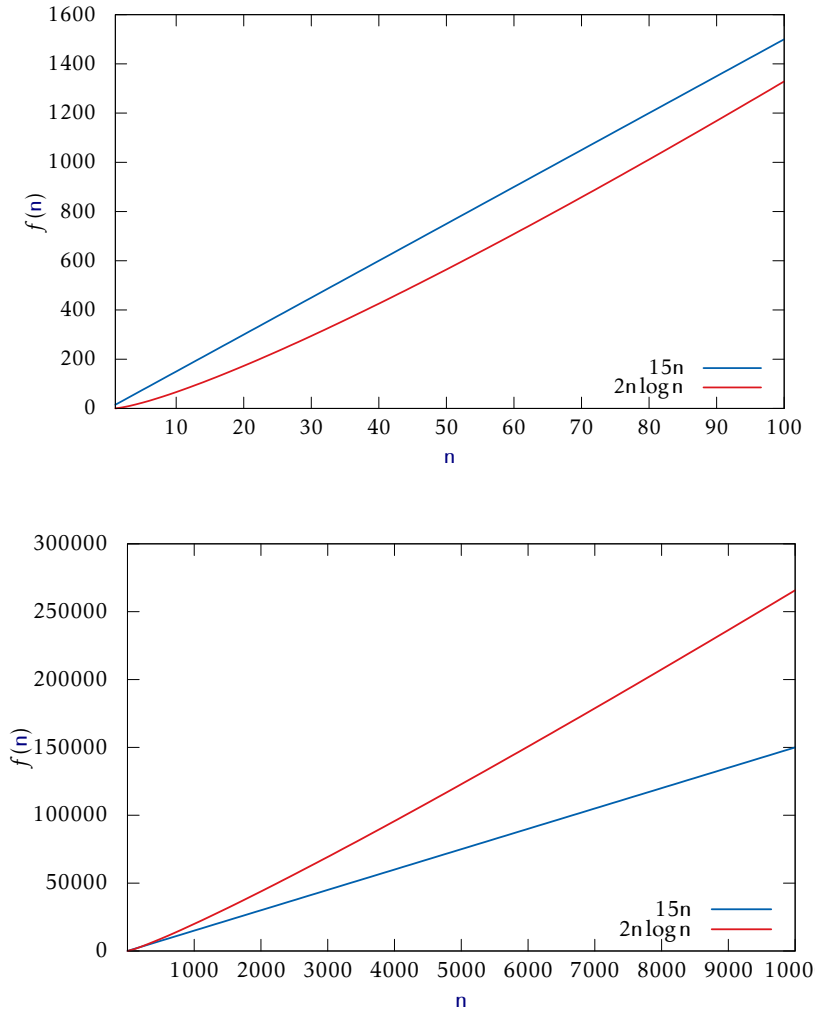


Figura 1.5: Gráfico para $15n$ versus $2n \log n$.

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*. 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

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

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

No modelo de palavra-RAM, operações básicas em palavras levam tempo constante. Isso inclui operações aritméticas (+, −, *, /, %), comparações (<, >, =, ≤, ≥), 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.

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` des-

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`.

critas 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 [53]. Essas são versões essencialmente simplificadas das interfaces `Lista`, `Set`, `Map`, `SortedSet` e `SortedMap` encontradas no Framework Java Collections.

Para um tratamento soberbo (e livre) da matemática discutido neste capítulo, incluindo a notação assintótica, logaritmos, fatoriais, aproximação de Stirling, probabilidade básica e muito mais, veja o livro de texto de

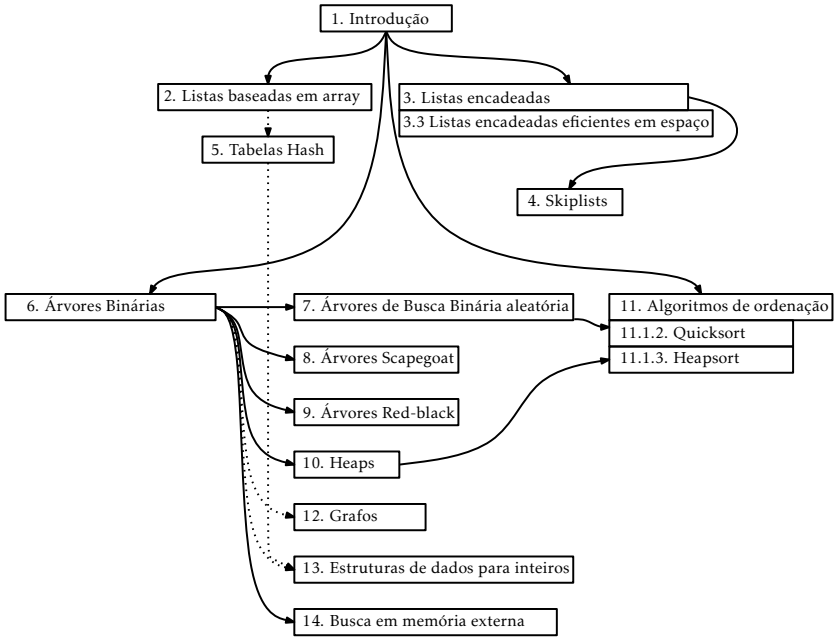


Figura 1.6: As dependências entre capítulos neste livro.

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.

Leyman, Leighton e Meyer [49]. Para um texto de cálculo suave que inclui definições formais de exponenciais e logaritmos, veja o texto clássico (livremente disponível) por Thompson [70].

Para obter mais informações sobre probabilidade básica, especialmente no que se refere à ciência da computação, consulte o livro de texto de Ross [62]. Outra boa referência, que abrange tanto a notação assintótica quanto a probabilidade, é o livro de Graham, Knuth e Patashnik [36].

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++)
```

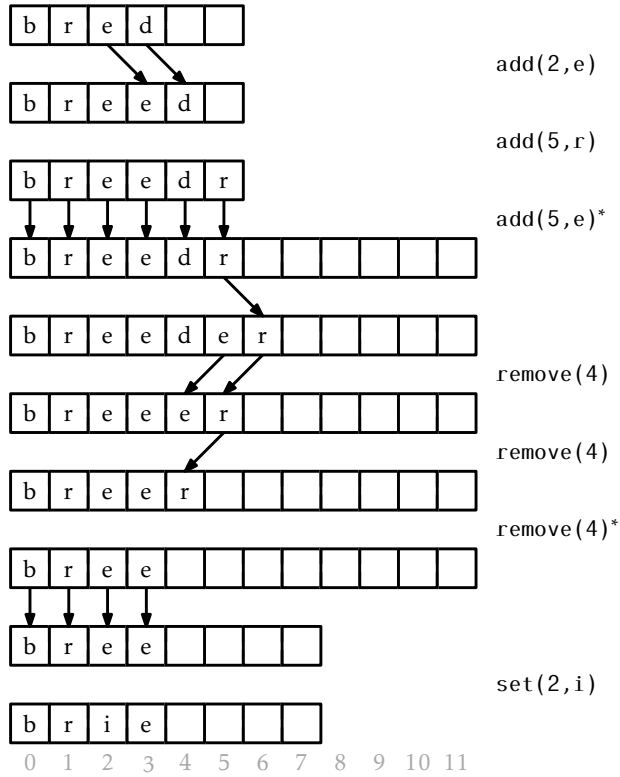


Figura 2.1: Uma sequência de operações `add(i,x)` e `remove(i)` em um `ArrayStack`. As setas indicam elementos que estão sendo copiados. As operações que resultam em uma chamada para `resize()` são marcadas com um asterisco.

```
    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.

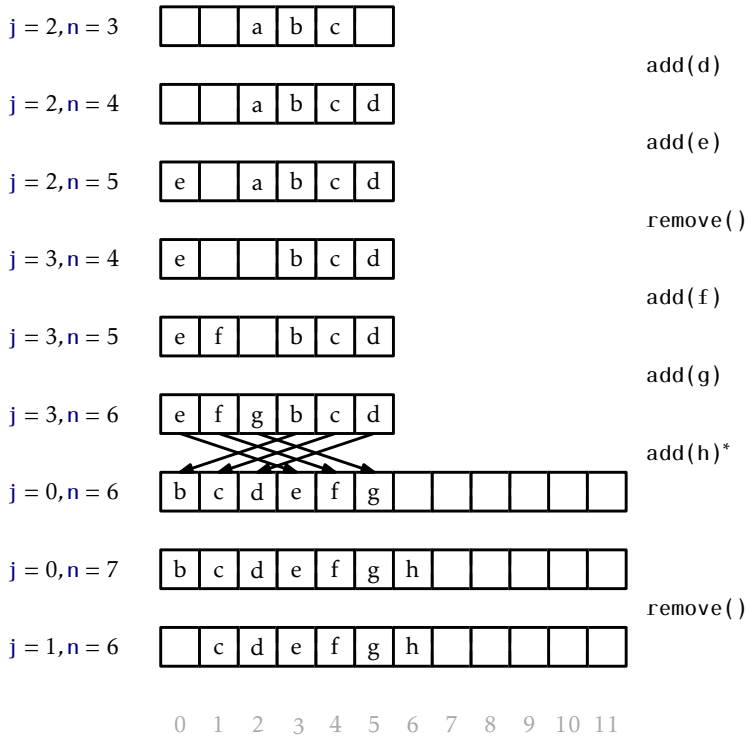


Figura 2.2: Sequência de operações add(x) e remove(i) em uma ArrayQueue. As setas indicam elementos que estão sendo copiados. Operações que resultam em uma chamada de resize() estão marcadas com um asterisco.

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;
```

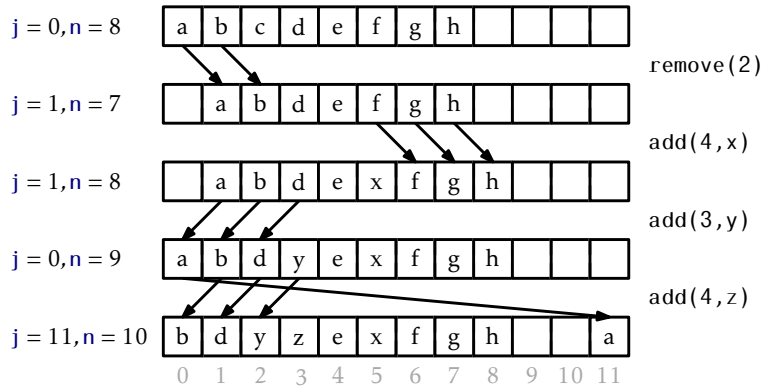


Figura 2.3: Uma sequência de operações `add(i,x)` e `remove(i)` em um `ArrayDeque`. As setas indicam elementos que estão sendo copiados.

```
  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()

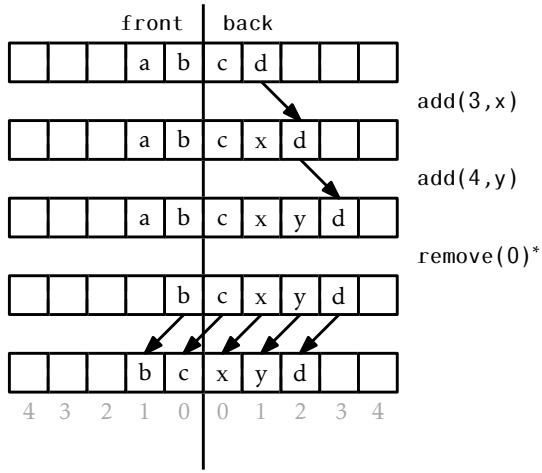


Figura 2.4: Uma sequência de operações add(i,x) e remove(i) em um DualArrayDeque. As setas indicam elementos que estão sendo copiados. Operações que resultam em um rebalanceamento por balance() são marcadas com um asterisco.

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

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

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

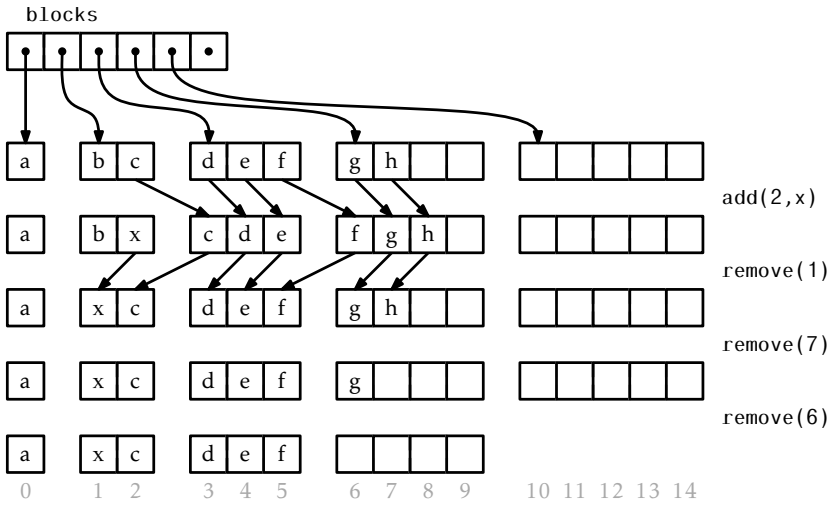


Figura 2.5: Uma sequência de operações `add(i,x)` e `remove(i)` em um `RootishArrayStack`. As flechas indicam elementos sendo copiados.

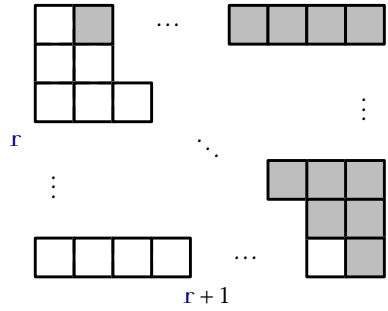


Figura 2.6: O número de quadrados brancos é $1 + 2 + 3 + \cdots + r$. O número de quadrados cinzas é o mesmo. Juntando os quadrados brancos e cinzas cria um retângulo consistindo de $r(r+1)$ quadrados.

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

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

Novamente, usando a equação quadrática nele fornece

$$r \le (3+\sqrt{1+4n})/2 = O(\sqrt{n}) \ .$$

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, mostramos que, em algum ponto durante a adição de n itens, a estrutura de dados está desperdiçando uma quantidade de espaço de no máximo $\sqrt{n}$ (embora possa estar apenas desperdiçado 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 um 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

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

Elevando ao quadrado ambos os lados da desigualdade, obtemos

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

E reunir 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 $2^r \leq x < 2^{r+1}$, so that $\lfloor \log x \rfloor = r$, 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 de 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 dentro de within 2 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 oartir 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 = ⌊log x⌋. Para superar isso, poderíamos calcular ⌊log n⌋ diferentes arrays sqrttab, um para cada possível valor de ⌊log x⌋. Os tamanhos dessas tabelas formam uma seqüência 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 = ⌊log n⌋. Qualquer valor x com log x = r' < r pode ser *pro,ovido* 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 [45, Section 2.2.2].

Brodnik *et al.* [12] 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 [34]. 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 = 0;
    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;
}
```

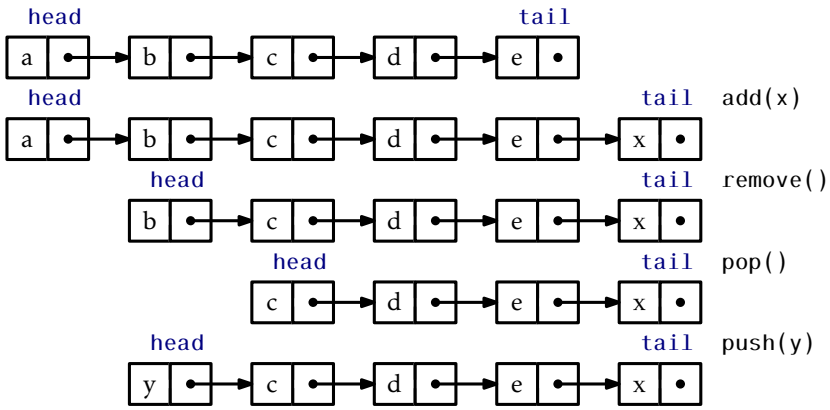


Figura 3.1: Uma sequência de operações de Queue (add(x) e remove()) e de Stack (push(x) e pop()) em uma SLList.

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

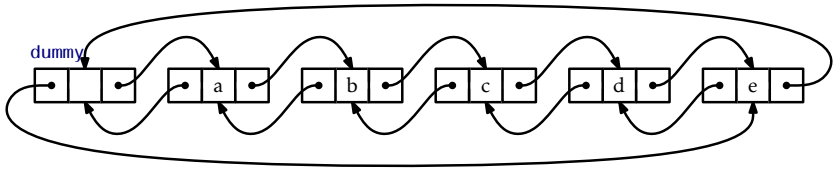


Figura 3.2: Uma DLList contendo a,b,c,d,e.

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;
}
```

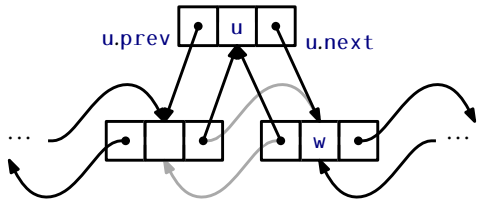


Figura 3.3: Adicionando o nó `u` antes do nó `w` em uma `DLList`.

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 index `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 index `i`; e
2. o index `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.

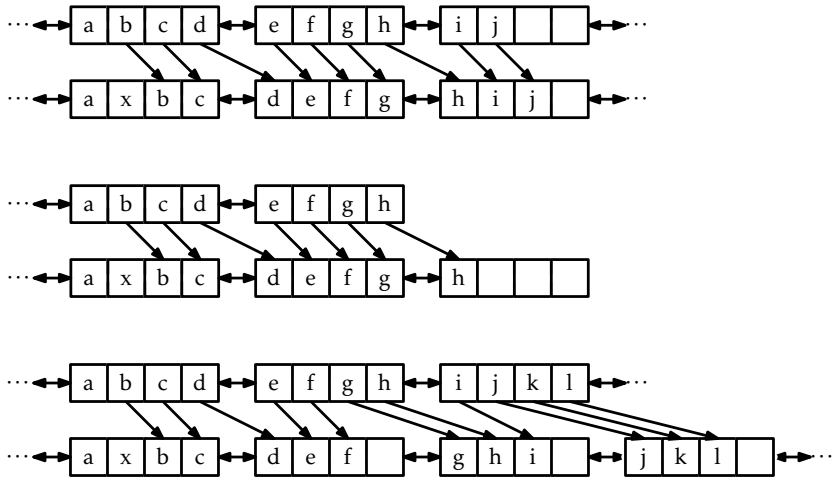


Figura 3.4: Os três casos que ocorrem durante a adição de um item x no interior de uma SEList. (Essa SEList tem bloco de tamanho b = 3.)

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

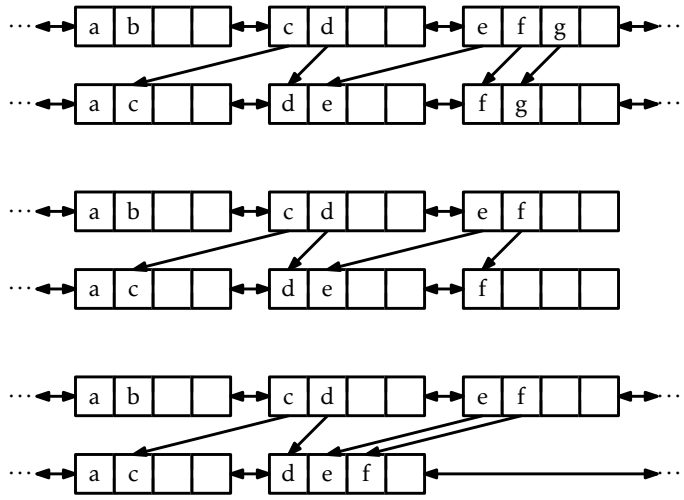


Figura 3.5: Os três casos que ocorrem durante a remoção de um item x no interior de uma SEList. (Esta SEList tem bloco de tamanho b = 3.)

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 [45, 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* [66].

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 [67] 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 operação 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 skilist, 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

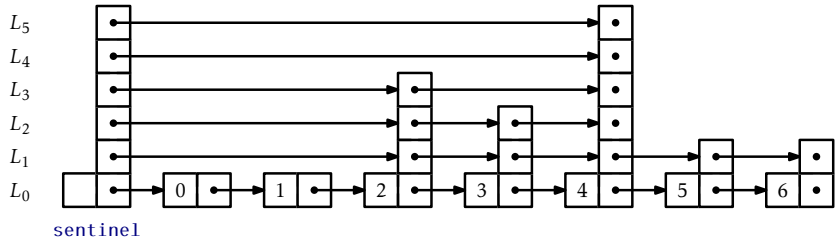


Figura 4.1: Uma skiplist com sete elementos.

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. Lembrando como construir um caminho de busca para o nó, `u`, é fácil (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 uma 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

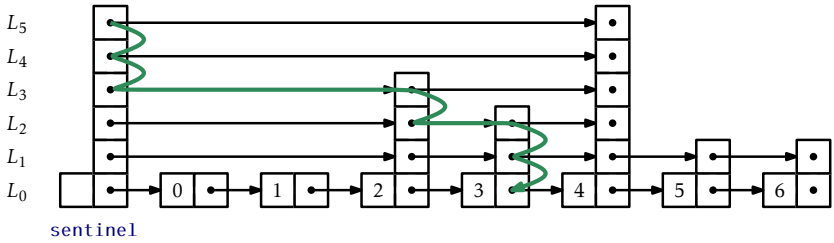


Figura 4.2: Caminho de busca para o nó que contém o valor 4 em uma skiplist.

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;
  }
```

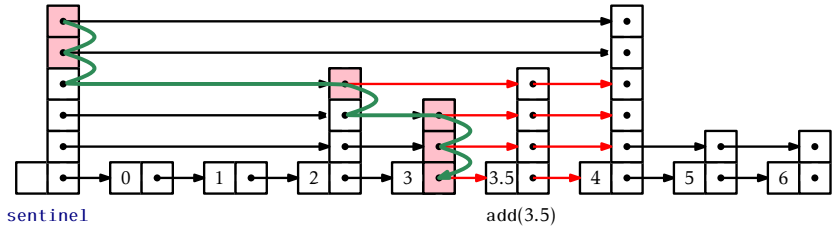


Figura 4.3: Adicionando o nó contendo o valor 3.5 a uma skiplist. Os nós armazenados em `stack` estão marcados.

```
  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--;
```

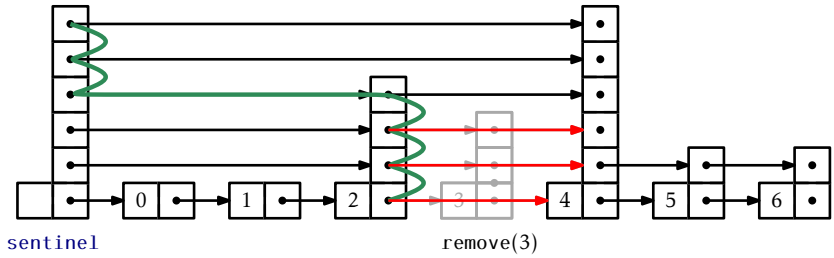


Figura 4.4: Removendo o nó que contém o valor 3 da skiplist.

```
  }
  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 $O(\log n)$.

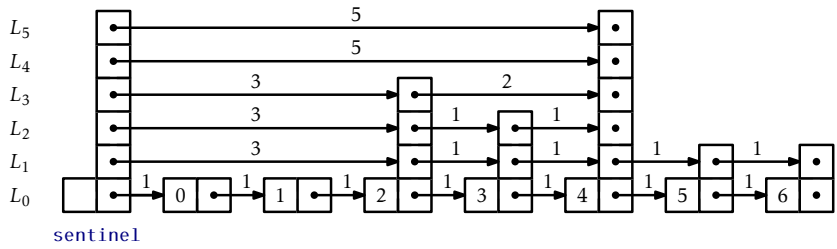


Figura 4.5: Os tamanhos das arestas em uma skiplist.

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, para que possamos 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 esta-

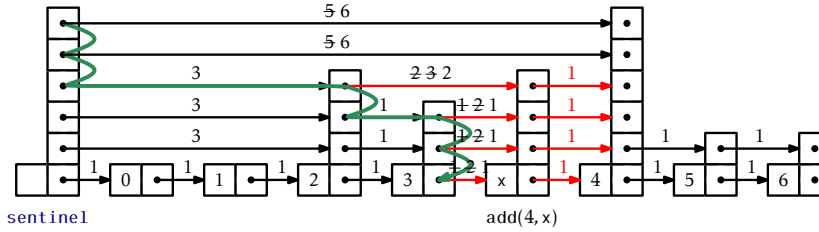


Figura 4.6: Adicionando um elemento em uma `SkiplistList`.

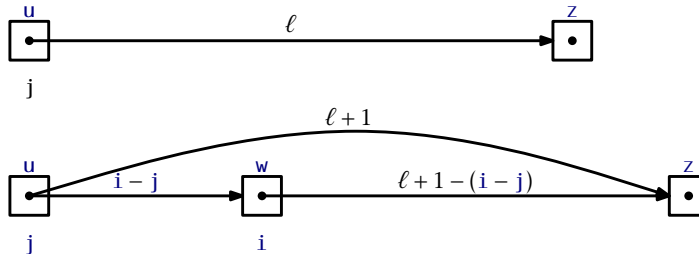


Figura 4.7: Atualizando os comprimentos das arestas enquanto junta-se o nó `w` em uma skiplist.

mos 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;
}
```

Até 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 classificação `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) {
```

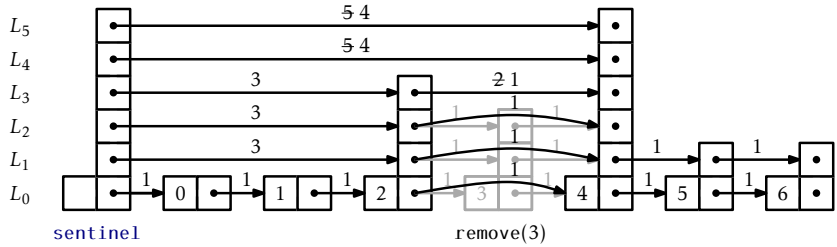


Figura 4.8: Removing an element from a SkiplistList.

```
    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 a performance 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 execuçao* $O(\log n)$ *esperado.*

4.4 Análise de Skiplists

Nesta seção, analizamos 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, o primeiro $i-1$ lançamento da moeda der 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. By Lema 4.3, the expected number of nodes, not including the sentinel, is $2n$. The number of occurrences of the sentinel is equal to the height, h, of the skiplist so, by Lema 4.4 the expected number of occurrences of the sentinel is at most $\log n + 2 = O(\log n)$. □

Lema 4.6. *The expected length of a search path in a skiplist is at most* $2\log n + O(1)$.

Demonstração. The easiest way to see this is to consider the *reverse search path* for a node, x. This path starts at the predecessor of x in L_0. At any point in time, if the path can go up a level, then it does. If it cannot go up a level then it goes left. Thinking about this for a few moments will convince us that the reverse search path for x is identical to the search path for x, except that it is reversed.

The number of nodes that the reverse search path visits at a particular level, r, is related to the following experiment: Toss a coin. If the coin comes up as heads, then move up and stop. Otherwise, move left and repeat the experiment. The number of coin tosses before the heads represents the number of steps to the left that a reverse search path takes at a particular level.[3] Lema 4.2 tells us that the expected number of coin tosses before the first heads is 1.

Let S_r denote the number of steps the forward search path takes at level r that go to the right. We have just argued that $E[S_r] \leq 1$. Furthermore, $S_r \leq |L_r|$, since we can't take more steps in L_r than the length of L_r, so

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

We can now finish as in the proof of Lema 4.4. Let S be the length of the search path for some node, u, in a skiplist, and let h be the height of the

[3] Note that this might overcount the number of steps to the left, since the experiment should end either at the first heads or when the search path reaches the sentinel, whichever comes first. This is not a problem since the lemma is only stating an upper bound.

skiplist. Then

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

□

The following theorem summarizes the results in this section:

Teorema 4.3. *A skiplist containing n elements has expected size $O(n)$ and the expected length of the search path for any particular element is at most $2\log n + O(1)$.*

4.5 Discussion and Exercises

Skiplists were introduced by Pugh [59] who also presented a number of applications and extensions of skiplists [58]. Since then they have been studied extensively. Several researchers have done very precise analyses of the expected length and variance of the length of the search path for the ith element in a skiplist [44, 43, 55]. Deterministic versions [52], biased versions [8, 25], and self-adjusting versions [11] of skiplists have all been developed. Skiplist implementations have been written for various

languages and frameworks and have been used in open-source database systems [68, 60]. A variant of skiplists is used in the HP-UX operating system kernel's process management structures [41].

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 inteiro *código hash* 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):

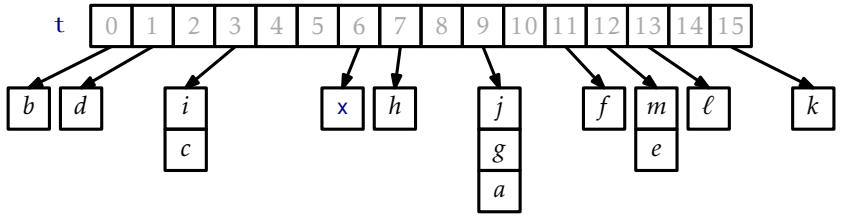


Figura 5.1: Um exemplo de `ChainedHashTable` com `n` = 14 e `t.length` = 16. Nesse exemplo `hash(x)` = 6

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 preservar 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 os 0 ou todos os 1. Isso é,

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

when $zx \bmod 2^w > zy \bmod 2^w$ or

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

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 t para ser 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 mantenha o invariante que $t.length = 2^d$.

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 entradas de pesquisa 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 $2^d \geq 3n$. Reorganizamos 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 k que começa em* `i` ocorre quando todas as entradas da tabela `t[i]`, `t[i + 1]`,..., `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 `rebuild()`. 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 por *tabulação hashing* 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 cordas, 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 `w`-bit. 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 cw 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 acabam por ser fáceis de imprimir (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 cite hash contém cerca de 2000 entradas.

Existe uma variedade de implementações de tabela hash diferentes. O descrito em Seção 5.1 é conhecido 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* [30, 23] e *cuckoo hashing* [54].

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 [13]. Hashing por tabulação, descrito em Seção 5.2.3, é graças a Carter e Wegman [13], mas sua análise, quando aplicada a sondagem linear (e vários outros esquemas de tabela de hash), é graças a Pătraşcu e Thorup [57].

A ideia do *hash multiplicativo* é muito antiga e parece ser parte do folclore hashing [47, 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.* [22]. 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 *multiply-add 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 $1/2^d$[20], 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 [10] 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.* [21]. 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 (&) [46, 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. Consider the hashing scheme in Seção 5.1.1, and suppose $n = 2^d$ and $d \leq w/2$.

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 $\mathtt{x} \oplus \mathtt{y}$ 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 $\mathtt{x} + \mathtt{y}$ 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 $h(\mathtt{x}, \mathtt{y})$ 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 $p = 2^{\mathtt{w}} - 1$ para algum inteiro positivo `w`. Explique porque, para um inteiro positivo x

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

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

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

até $\mathtt{x} \leq 2^{\mathtt{w}} - 1$.)

Exercício 5.11. Encontre algumas implementações de tabelas hash comumente usadas, tais como as implementações (

javaonlyJava Collection Framework `HashMap`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 [16].)

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-

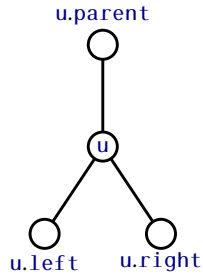


Figura 6.1: O pai, o filho esquerdo e o filho direito do nó u em uma BinaryTree.

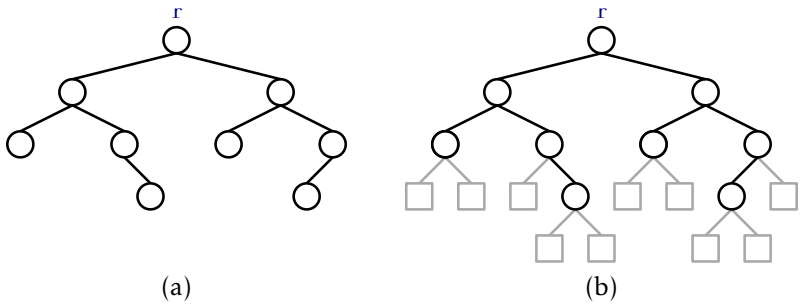


Figura 6.2: Uma árvore binária com (a) nove nós reais e (b) dez nós externos.

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-

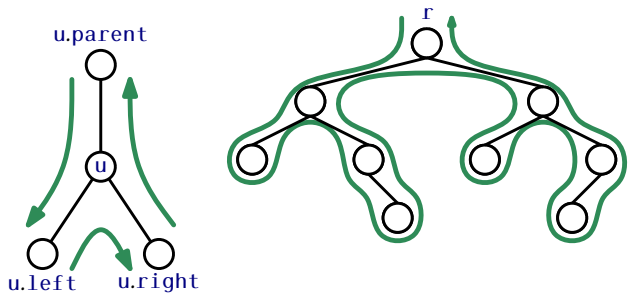


Figura 6.3: Os três casos que ocorrem no nó u quando percorremos uma árvore binária não recursivamente, e o resultado da travessia pela árvore.

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

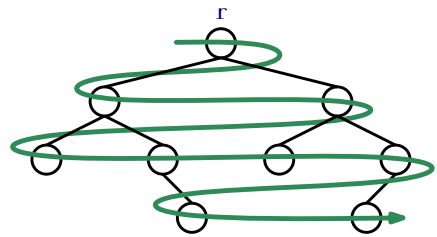


Figura 6.4: Durante um percurso em profundidade, os nós de uma árvore binária são visitados nível por nível e da esquerda para a direita dentro de cada nível.

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);
  }
}
```

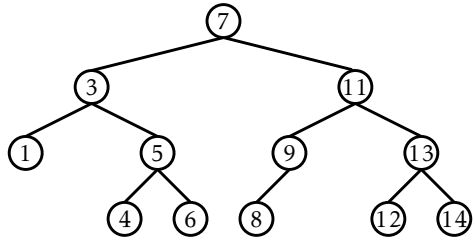


Figura 6.5: Uma árvore binária de busca.

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;
```

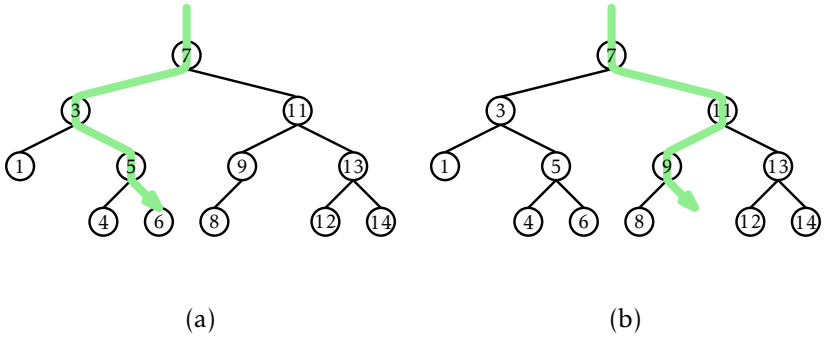


Figura 6.6: Um exemplo de (a) uma busca com sucesso (por 6) e (b) uma busca frustrada (por 10) em uma árvore binária de busca.

```
    }
  }
  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`.

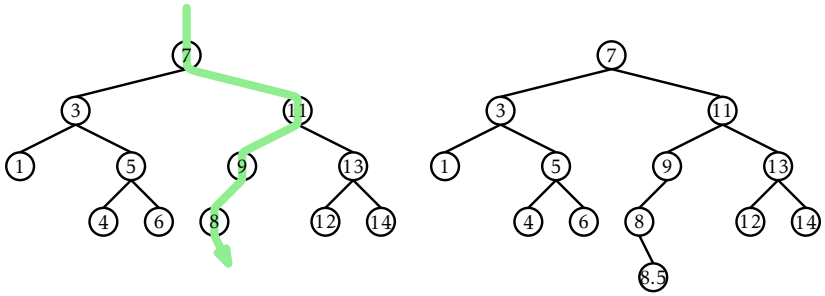


Figura 6.7: Inderindo o valor 8.5 na árvore binária de busca.

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) {
```

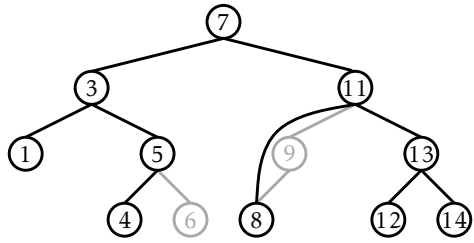


Figura 6.8: Removendo uma folha (6) ou um nó com apenas um filho (9) é fácil.

```
    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;
  }
}
```

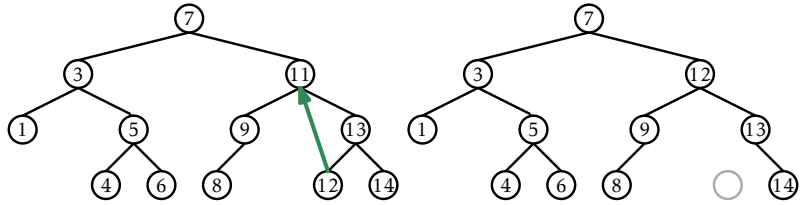


Figura 6.9: Apagar um valor (11) de um nó, u, com dois filhos é realizado substituindo o valor de u pelo menor valor na subárvore direita de u.

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 [47, 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 $u.p[(i+1) \bmod 3]$. Exemplos similares ocorrem quando existe uma simetria esquerda-direita. Por exemplo, o irmão de $u.p[i]$ é $u.p[(i+1) \bmod 2]$. 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 $n \geq 1$ nós possui $n - 1$ arestas.

Exercício 6.2. Prove que uma árvore binária com $n \geq 1$ 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*

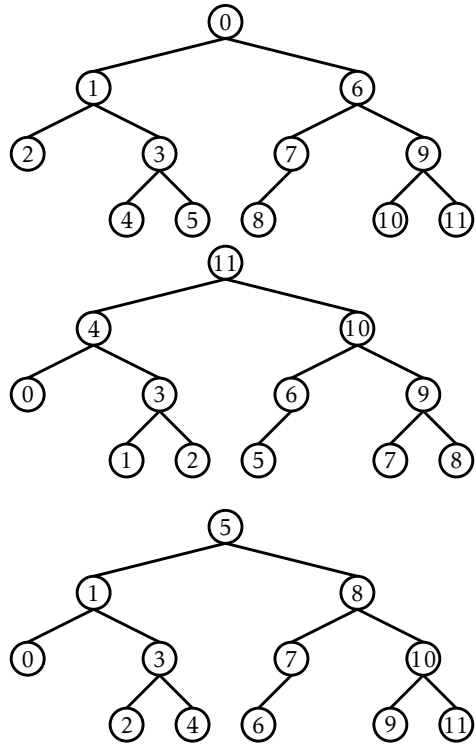


Figura 6.10: numeração em pré-ordem, pós-ordem, e em-ordem de uma árvore binária.

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 para a `BinarySearchTree`, `obtemLE(x)`, 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

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

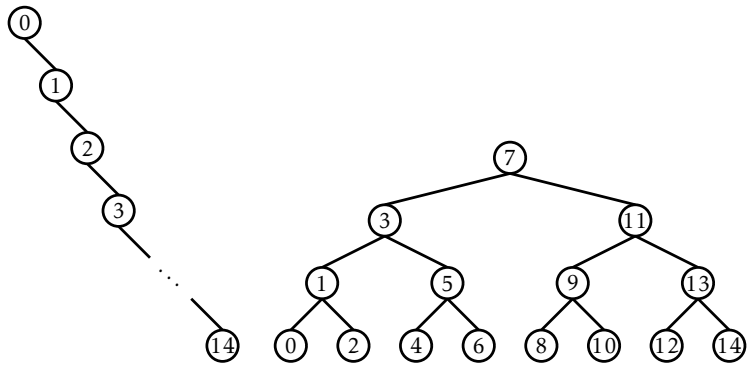


Figura 7.1: Duas árvores binárias de busca com inteiros $0,\ldots,14$.

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-

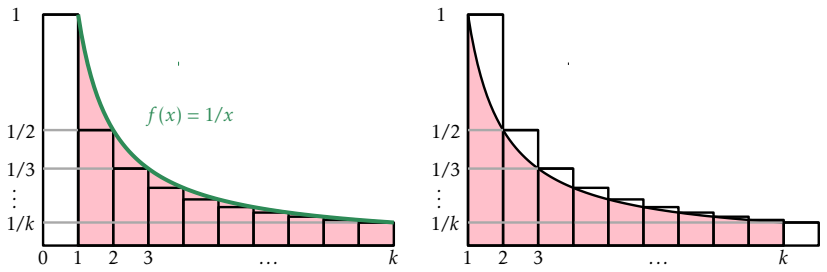


Figura 7.2: O $k$ésimo número harmônico $H_k = \sum_{i=1}^{k} 1/i$ tem limite superior e inferior dados por duas integrais. O valor dessas integrais é dado pela área da região sombreada, enquanto o valor de H_k é dado pela área dos retângulos.

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

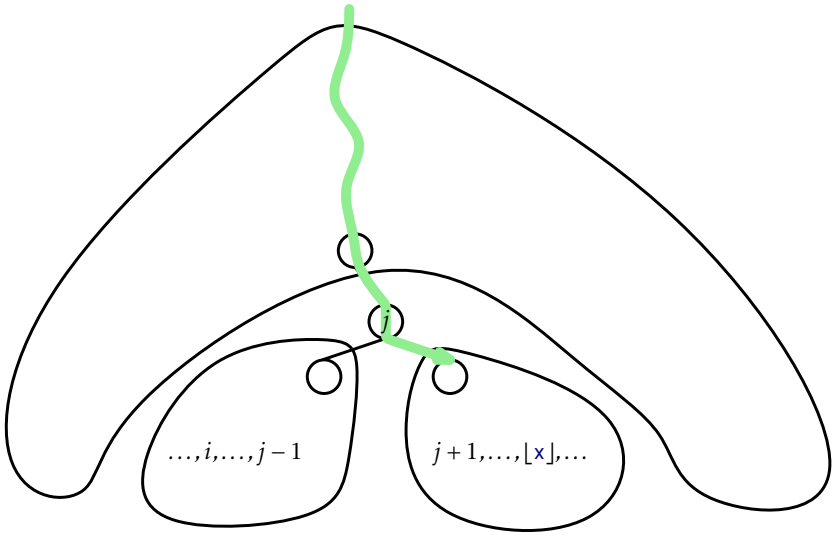


Figura 7.3: O valor $i < x$ está no caminho de busca para x se e somente se i é o primeiro elemento entre $\{i, i+1, \ldots, \lfloor x \rfloor\}$ inserido na árvore.

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

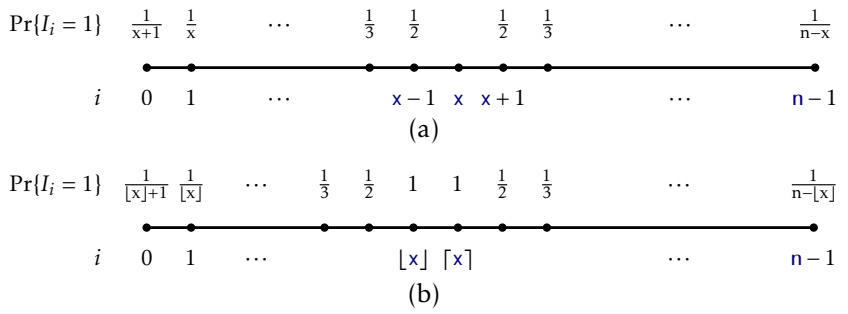


Figura 7.4: As probabilidades de um elemento estar no caminho de busca para x quando (a) x é um inteiro e (b) quando x não é um inteiro.

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).

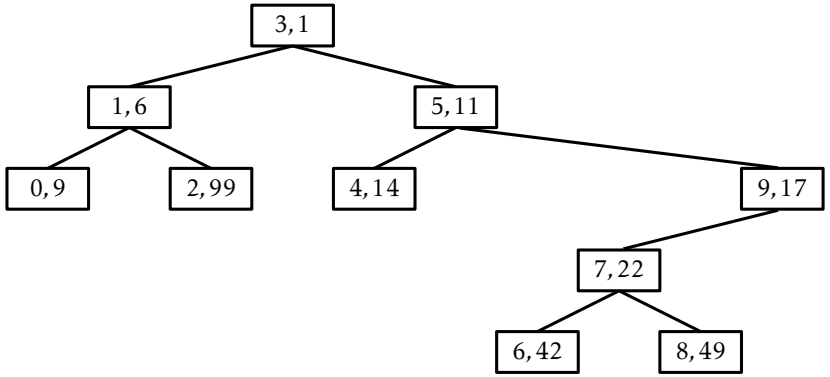


Figura 7.5: Um exemplo de uma Treap contendo os inteiros $0, \ldots, 9$. Cada nó, u, é ilustrado como uma caixa contendo u.x, u.p.

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.

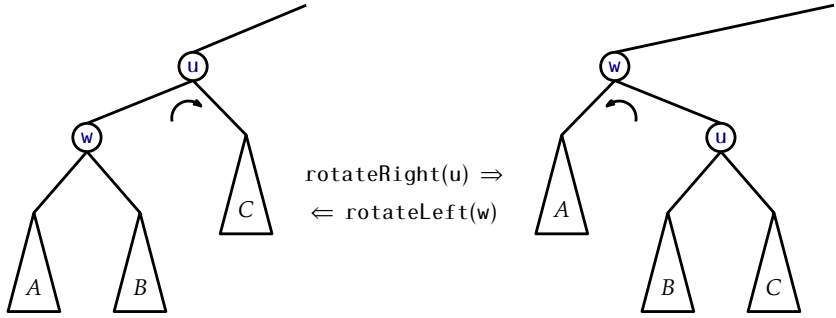


Figura 7.6: Rotações à esquerda e à direita em uma árvore binária de busca.

```
    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.

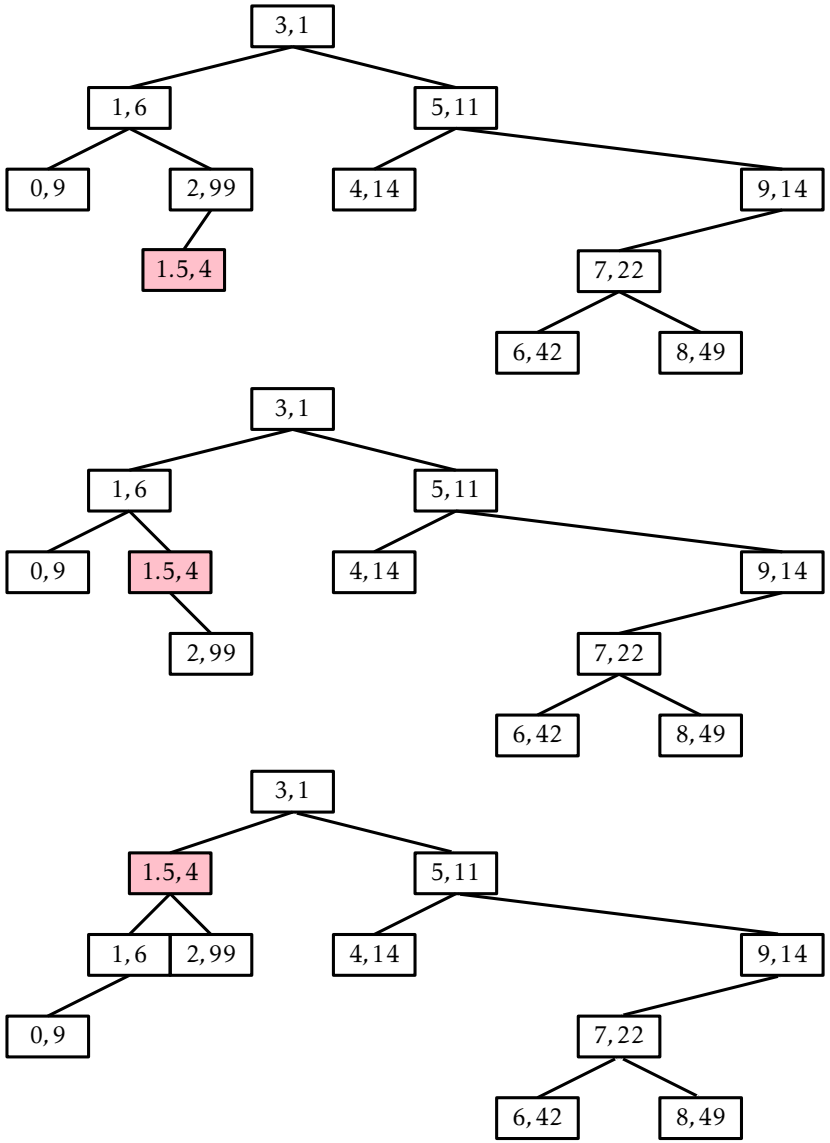


Figura 7.7: Inserindo o valor 1.5 na Treap da Figura 7.5.

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

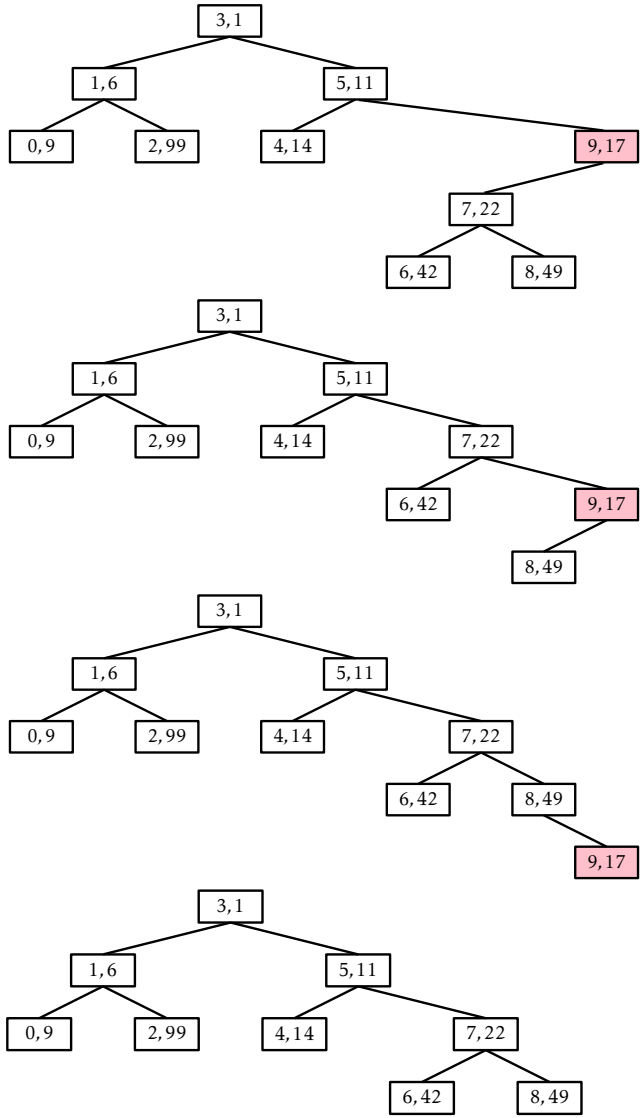


Figura 7.8: Removendo o valor 9 da Treap na Figura 7.5.

$O(\log n)$.

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 [18] 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 [61], 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 [64] que discutiram Treaps e algumas de suas variantes. Contudo, a estrutura básica foi estudada anteriormente por Vuillemin [73] que as chamou de árvores Cartesinas.

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 [50]. 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

Scapegoat Trees

In this chapter, we study a binary search tree data structure, the ScapegoatTree. This structure is based on the common wisdom that, when something goes wrong, the first thing people tend to do is find someone to blame (the *scapegoat*). Once blame is firmly established, we can leave the scapegoat to fix the problem.

A ScapegoatTree keeps itself balanced by *partial rebuilding operations*. During a partial rebuilding operation, an entire subtree is deconstructed and rebuilt into a perfectly balanced subtree. There are many ways of rebuilding a subtree rooted at node u into a perfectly balanced tree. One of the simplest is to traverse u's subtree, gathering all its nodes into an array, a, and then to recursively build a balanced subtree using a. If we let m = a.length/2, then the element a[m] becomes the root of the new subtree, a[0],...,a[m−1] get stored recursively in the left subtree and a[m+1],...,a[a.length−1] get stored recursively in the right subtree.

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);
}
```

A call to `rebuild(u)` takes $O(\texttt{size}(u))$ time. The resulting subtree has minimum height; there is no tree of smaller height that has `size(u)` nodes.

8.1 ScapegoatTree: A Binary Search Tree with Partial Rebuilding

A `ScapegoatTree` is a `BinarySearchTree` that, in addition to keeping track of the number, n, of nodes in the tree also keeps a counter, q, that maintains an upper-bound on the number of nodes.

ScapegoatTree

```
int q;
```

At all times, n and q obey the following inequalities:

$$q/2 \le n \le q \ .$$

In addition, a `ScapegoatTree` has logarithmic height; at all times, the height of the scapegoat tree does not exceed

$$\log_{3/2} q \le \log_{3/2} 2n < \log_{3/2} n + 2 \ . \tag{8.1}$$

Even with this constraint, a `ScapegoatTree` can look surprisingly unbalanced. The tree in Figura 8.1 has $q = n = 10$ and height $5 < \log_{3/2} 10 \approx 5.679$.

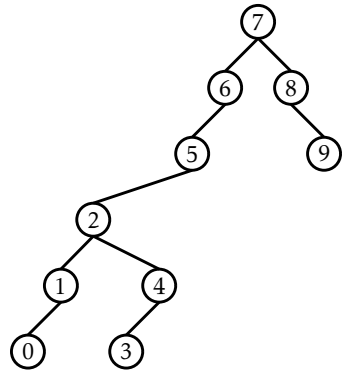


Figura 8.1: A ScapegoatTree with 10 nodes and height 5.

Implementing the `find(x)` operation in a `ScapegoatTree` is done using the standard algorithm for searching in a `BinarySearchTree` (see Seção 6.2). This takes time proportional to the height of the tree which, by (8.1) is $O(\log n)$.

To implement the `add(x)` operation, we first increment `n` and `q` and then use the usual algorithm for adding `x` to a binary search tree; we search for `x` and then add a new leaf `u` with `u.x = x`. At this point, we may get lucky and the depth of `u` might not exceed $\log_{3/2} q$. If so, then we leave well enough alone and don't do anything else.

Unfortunately, it will sometimes happen that `depth(u)` $> \log_{3/2} q$. In this case, we need to reduce the height. This isn't a big job; there is only one node, namely `u`, whose depth exceeds $\log_{3/2} q$. To fix `u`, we walk from `u` back up to the root looking for a *scapegoat*, `w`. The scapegoat, `w`, is a very unbalanced node. It has the property that

$$\frac{\texttt{size(w.child)}}{\texttt{size(w)}} > \frac{2}{3} \ , \tag{8.2}$$

where `w.child` is the child of `w` on the path from the root to `u`. We'll very shortly prove that a scapegoat exists. For now, we can take it for granted. Once we've found the scapegoat `w`, we completely destroy the subtree rooted at `w` and rebuild it into a perfectly balanced binary search tree. We

know, from (8.2), that, even before the addition of u, w's subtree was not a complete binary tree. Therefore, when we rebuild w, the height decreases by at least 1 so that the height of the ScapegoatTree is once again at most $\log_{3/2} q$.

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;
}
```

If we ignore the cost of finding the scapegoat w and rebuilding the subtree rooted at w, then the running time of add(x) is dominated by the initial search, which takes $O(\log q) = O(\log n)$ time. We will account for the cost of finding the scapegoat and rebuilding using amortized analysis in the next section.

The implementation of remove(x) in a ScapegoatTree is very simple. We search for x and remove it using the usual algorithm for removing a node from a BinarySearchTree. (Note that this can never increase the height of the tree.) Next, we decrement n, but leave q unchanged. Finally,

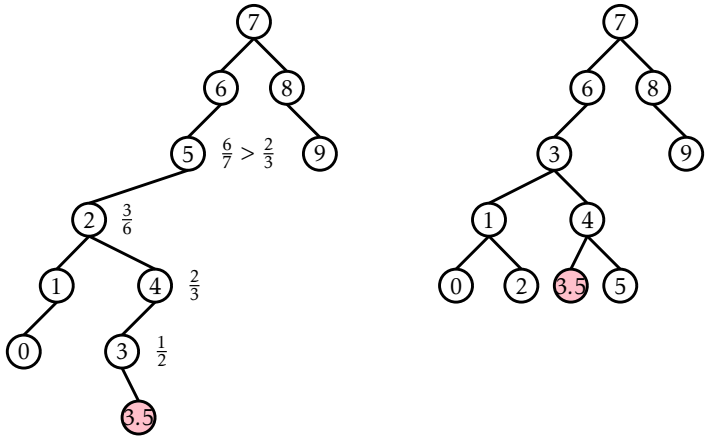


Figura 8.2: Inserting 3.5 into a ScapegoatTree increases its height to 6, which violates (8.1) since $6 > \log_{3/2} 11 \approx 5.914$. A scapegoat is found at the node containing 5.

we check if $q > 2n$ and, if so, then we *rebuild the entire tree* into a perfectly balanced binary search tree and set $q = n$.

ScapegoatTree

```
bool remove(T x) {
  if (BinarySearchTree<Node,T>::remove(x)) {
    if (2*n < q) {
      rebuild(r);
      q = n;
    }
    return true;
  }
  return false;
}
```

Again, if we ignore the cost of rebuilding, the running time of the `remove(x)` operation is proportional to the height of the tree, and is therefore $O(\log n)$.

8.1.1 Analysis of Correctness and Running-Time

In this section, we analyze the correctness and amortized running time of operations on a ScapegoatTree. We first prove the correctness by showing that, when the add(x) operation results in a node that violates Condition (8.1), then we can always find a scapegoat:

Lema 8.1. *Let* u *be a node of depth* $h > \log_{3/2} q$ *in a ScapegoatTree. Then there exists a node* w *on the path from* u *to the root such that*

$$\frac{\texttt{size}(w)}{\texttt{size}(\texttt{parent}(w))} > 2/3 \ .$$

Demonstração. Suppose, for the sake of contradiction, that this is not the case, and

$$\frac{\texttt{size}(w)}{\texttt{size}(\texttt{parent}(w))} \leq 2/3 \ .$$

for all nodes w on the path from u to the root. Denote the path from the root to u as $r = u_0, \ldots, u_h = u$. Then, we have $\texttt{size}(u_0) = n$, $\texttt{size}(u_1) \leq \frac{2}{3}n$, $\texttt{size}(u_2) \leq \frac{4}{9}n$ and, more generally,

$$\texttt{size}(u_i) \leq \left(\frac{2}{3}\right)^i n \ .$$

But this gives a contradiction, since $\texttt{size}(u) \geq 1$, hence

$$1 \leq \texttt{size}(u) \leq \left(\frac{2}{3}\right)^h n < \left(\frac{2}{3}\right)^{\log_{3/2} q} n \leq \left(\frac{2}{3}\right)^{\log_{3/2} n} n = \left(\frac{1}{n}\right) n = 1 \ . \qquad \square$$

Next, we analyze the parts of the running time that are not yet accounted for. There are two parts: The cost of calls to size(u) when searching for scapegoat nodes, and the cost of calls to rebuild(w) when we find a scapegoat w. The cost of calls to size(u) can be related to the cost of calls to rebuild(w), as follows:

Lema 8.2. *During a call to* add(x) *in a ScapegoatTree, the cost of finding the scapegoat* w *and rebuilding the subtree rooted at* w *is* $O(\texttt{size}(w))$.

Demonstração. The cost of rebuilding the scapegoat node w, once we find it, is $O(\texttt{size}(w))$. When searching for the scapegoat node, we call size(u)

on a sequence of nodes $u_0, \ldots, u_k$ until we find the scapegoat $u_k = w$. However, since u_k is the first node in this sequence that is a scapegoat, we know that

$$\texttt{size}(u_i) < \frac{2}{3}\texttt{size}(u_{i+1})$$

for all $i \in \{0, \ldots, k-2\}$. Therefore, the cost of all calls to $\texttt{size}(u)$ is

$$\begin{aligned}
O\left(\sum_{i=0}^{k} \texttt{size}(u_{k-i})\right) &= O\left(\texttt{size}(u_k) + \sum_{i=0}^{k-1} \texttt{size}(u_{k-i-1})\right) \\
&= O\left(\texttt{size}(u_k) + \sum_{i=0}^{k-1} \left(\frac{2}{3}\right)^i \texttt{size}(u_k)\right) \\
&= O\left(\texttt{size}(u_k)\left(1 + \sum_{i=0}^{k-1} \left(\frac{2}{3}\right)^i\right)\right) \\
&= O(\texttt{size}(u_k)) = O(\texttt{size}(w)) \ ,
\end{aligned}$$

where the last line follows from the fact that the sum is a geometrically decreasing series. □

All that remains is to prove an upper-bound on the cost of all calls to $\texttt{rebuild}(u)$ during a sequence of m operations:

Lema 8.3. *Starting with an empty* `ScapegoatTree` *any sequence of* m $\texttt{add}(x)$ *and* $\texttt{remove}(x)$ *operations causes at most* $O(m \log m)$ *time to be used by* $\texttt{rebuild}(u)$ *operations.*

Demonstração. To prove this, we will use a *credit scheme*. We imagine that each node stores a number of credits. Each credit can pay for some constant, c, units of time spent rebuilding. The scheme gives out a total of $O(m \log m)$ credits and every call to $\texttt{rebuild}(u)$ is paid for with credits stored at u.

During an insertion or deletion, we give one credit to each node on the path to the inserted node, or deleted node, u. In this way we hand out at most $\log_{3/2} q \leq \log_{3/2} m$ credits per operation. During a deletion we also store an additional credit "on the side." Thus, in total we give out at most $O(m \log m)$ credits. All that remains is to show that these credits are sufficient to pay for all calls to $\texttt{rebuild}(u)$.

If we call `rebuild(u)` during an insertion, it is because `u` is a scapegoat. Suppose, without loss of generality, that

$$\frac{\texttt{size(u.left)}}{\texttt{size(u)}} > \frac{2}{3} \ .$$

Using the fact that

$$\texttt{size(u)} = 1 + \texttt{size(u.left)} + \texttt{size(u.right)}$$

we deduce that

$$\frac{1}{2}\texttt{size(u.left)} > \texttt{size(u.right)}$$

and therefore

$$\texttt{size(u.left)} - \texttt{size(u.right)} > \frac{1}{2}\texttt{size(u.left)} > \frac{1}{3}\texttt{size(u)} \ .$$

Now, the last time a subtree containing `u` was rebuilt (or when `u` was inserted, if a subtree containing `u` was never rebuilt), we had

$$\texttt{size(u.left)} - \texttt{size(u.right)} \leq 1 \ .$$

Therefore, the number of `add(x)` or `remove(x)` operations that have affected `u.left` or `u.right` since then is at least

$$\frac{1}{3}\texttt{size(u)} - 1 \ .$$

and there are therefore at least this many credits stored at `u` that are available to pay for the $O(\texttt{size(u)})$ time it takes to call `rebuild(u)`.

If we call `rebuild(u)` during a deletion, it is because `q` > 2`n`. In this case, we have `q` – `n` > `n` credits stored "on the side," and we use these to pay for the $O(\texttt{n})$ time it takes to rebuild the root. This completes the proof. □

8.1.2 Summary

The following theorem summarizes the performance of the `ScapegoatTree` data structure:

Teorema 8.1. *A* `ScapegoatTree` *implements the* `SSet` *interface. Ignoring the cost of* `rebuild(u)` *operations, a* `ScapegoatTree` *supports the operations* `add(x)`, `remove(x)`, *and* `find(x)` *in* $O(\log n)$ *time per operation.*

Furthermore, beginning with an empty `ScapegoatTree`, *any sequence of* m `add(x)` *and* `remove(x)` *operations results in a total of* $O(m \log m)$ *time spent during all calls to* `rebuild(u)`.

8.2 Discussion and Exercises

The term *scapegoat tree* is due to Galperin and Rivest [32], who define and analyze these trees. However, the same structure was discovered earlier by Andersson [5, 7], who called them *general balanced trees* since they can have any shape as long as their height is small.

Experimenting with the `ScapegoatTree` implementation will reveal that it is often considerably slower than the other `SSet` implementations in this book. This may be somewhat surprising, since height bound of

$$\log_{3/2} q \approx 1.709 \log n + O(1)$$

is better than the expected length of a search path in a `Skiplist` and not too far from that of a `Treap`. The implementation could be optimized by storing the sizes of subtrees explicitly at each node or by reusing already computed subtree sizes (Exercises 8.5 and 8.6). Even with these optimizations, there will always be sequences of `add(x)` and `delete(x)` operation for which a `ScapegoatTree` takes longer than other `SSet` implementations.

This gap in performance is due to the fact that, unlike the other `SSet` implementations discussed in this book, a `ScapegoatTree` can spend a lot of time restructuring itself. Exercício 8.3 asks you to prove that there are sequences of n operations in which a `ScapegoatTree` will spend on the order of $n \log n$ time in calls to `rebuild(u)`. This is in contrast to other `SSet` implementations discussed in this book, which only make $O(n)$ structural changes during a sequence of n operations. This is, unfortunately, a necessary consequence of the fact that a `ScapegoatTree` does all its restructuring by calls to `rebuild(u)` [19].

Despite their lack of performance, there are applications in which a

ScapegoatTree could be the right choice. This would occur any time there is additional data associated with nodes that cannot be updated in constant time when a rotation is performed, but that can be updated during a rebuild(u) operation. In such cases, the ScapegoatTree and related structures based on partial rebuilding may work. An example of such an application is outlined in Exercício 8.11.

Exercício 8.1. Illustrate the addition of the values 1.5 and then 1.6 on the ScapegoatTree in Figura 8.1.

Exercício 8.2. Illustrate what happens when the sequence $1, 5, 2, 4, 3$ is added to an empty ScapegoatTree, and show where the credits described in the proof of Lema 8.3 go, and how they are used during this sequence of additions.

Exercício 8.3. Show that, if we start with an empty ScapegoatTree and call add(x) for $x = 1, 2, 3, \ldots, n$, then the total time spent during calls to rebuild(u) is at least $cn \log n$ for some constant $c > 0$.

Exercício 8.4. The ScapegoatTree, as described in this chapter, guarantees that the length of the search path does not exceed $\log_{3/2} q$.

1. Design, analyze, and implement a modified version of ScapegoatTree where the length of the search path does not exceed $\log_b q$, where b is a parameter with $1 < b < 2$.

2. What does your analysis and/or your experiments say about the amortized cost of find(x), add(x) and remove(x) as a function of n and b?

Exercício 8.5. Modify the add(x) method of the ScapegoatTree so that it does not waste any time recomputing the sizes of subtrees that have already been computed. This is possible because, by the time the method wants to compute size(w), it has already computed one of size(w.left) or size(w.right). Compare the performance of your modified implementation with the implementation given here.

Exercício 8.6. Implement a second version of the ScapegoatTree data structure that explicitly stores and maintains the sizes of the subtree rooted at each node. Compare the performance of the resulting implemen-

tation with that of the original ScapegoatTree implementation as well as the implementation from Exercício 8.5.

Exercício 8.7. Reimplement the rebuild(u) method discussed at the beginning of this chapter so that it does not require the use of an array to store the nodes of the subtree being rebuilt. Instead, it should use recursion to first connect the nodes into a linked list and then convert this linked list into a perfectly balanced binary tree. (There are very elegant recursive implementations of both steps.)

Exercício 8.8. Analyze and implement a WeightBalancedTree. This is a tree in which each node u, except the root, maintains the *balance invariant* that size(u) ≤ (2/3)size(u.parent). The add(x) and remove(x) operations are identical to the standard BinarySearchTree operations, except that any time the balance invariant is violated at a node u, the subtree rooted at u.parent is rebuilt. Your analysis should show that operations on a WeightBalancedTree run in $O(\log n)$ amortized time.

Exercício 8.9. Analyze and implement a CountdownTree. In a CountdownTree each node u keeps a *timer* u.t. The add(x) and remove(x) operations are exactly the same as in a standard BinarySearchTree except that, whenever one of these operations affects u's subtree, u.t is decremented. When u.t = 0 the entire subtree rooted at u is rebuilt into a perfectly balanced binary search tree. When a node u is involved in a rebuilding operation (either because u is rebuilt or one of u's ancestors is rebuilt) u.t is reset to size(u)/3.

Your analysis should show that operations on a CountdownTree run in $O(\log n)$ amortized time. (Hint: First show that each node u satisfies some version of a balance invariant.)

Exercício 8.10. Analyze and implement a DynamiteTree. In a DynamiteTree each node u keeps tracks of the size of the subtree rooted at u in a variable u.size. The add(x) and remove(x) operations are exactly the same as in a standard BinarySearchTree except that, whenever one of these operations affects a node u's subtree, u *explodes* with probability 1/u.size. When u explodes, its entire subtree is rebuilt into a perfectly balanced binary search tree.

Your analysis should show that operations on a `DynamiteTree` run in $O(\log n)$ expected time.

Exercício 8.11. Design and implement a Sequence data structure that maintains a sequence (list) of elements. It supports these operations:

- `addAfter(e)`: Add a new element after the element `e` in the sequence. Return the newly added element. (If `e` is null, the new element is added at the beginning of the sequence.)
- `remove(e)`: Remove `e` from the sequence.
- `testBefore(e1,e2)`: return `true` if and only if `e1` comes before `e2` in the sequence.

The first two operations should run in $O(\log n)$ amortized time. The third operation should run in constant time.

The Sequence data structure can be implemented by storing the elements in something like a `ScapegoatTree`, in the same order that they occur in the sequence. To implement `testBefore(e1,e2)` in constant time, each element `e` is labelled with an integer that encodes the path from the root to `e`. In this way, `testBefore(e1,e2)` can be implemented by comparing the labels of `e1` and `e2`.

Capítulo 9

Red-Black Trees (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 temp $O(\log n)$ no *pior caso*.

3. O número de rotações amortizadas realizadas durante uma operação `add(x)` or `remove(x)` é constante.

As duas primeiras dessas propriedades já colocam as árvores rubro-negras à frente de skiplists, treaps, e scapegoat trees. 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

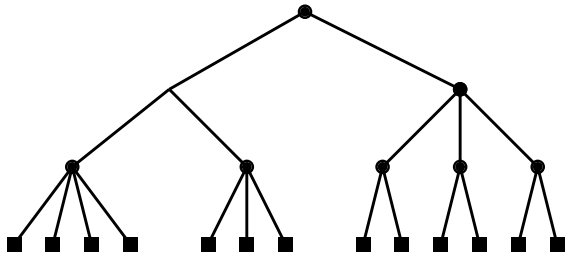


Figura 9.1: Uma árvore 2-4 de altura 3.

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 2-4 Trees (Árvores 2-4)

Uma árvore 2-4 é uma árvore enraizada com as seguintes propriedades:

Property 9.1 (Altura)**.** Todas as folhas têm a mesma profundidade.

Property 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 and 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

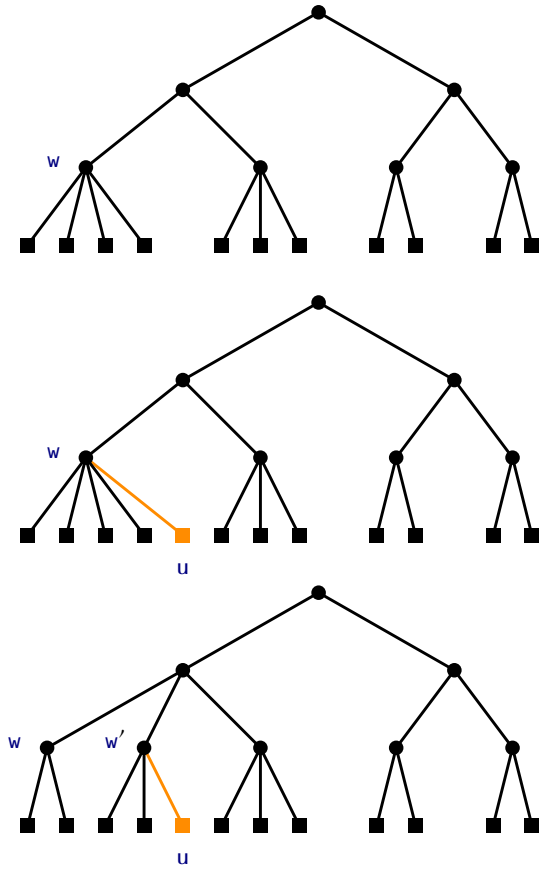


Figura 9.2: Adicionando uma folha na árvore 2-4. Este processo termina depois de uma divisão porque `w.parent` tem um grau inferior a 4 antes da adição.

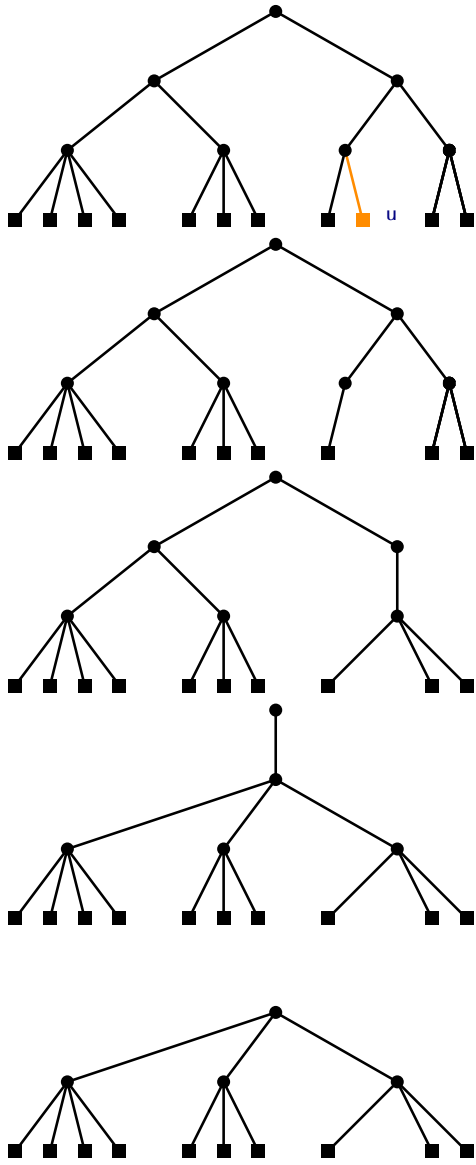


Figura 9.3: Removendo uma folha de uma árvore 2-4. Este processo vai até a raiz porque cada um dos antecessores u e seus irmãos têm somente dois filhos.

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 RedBlackTree: 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* or *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.

Property 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.)

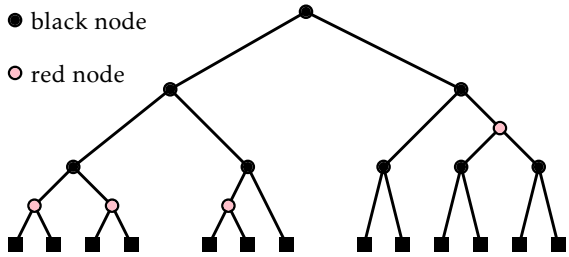


Figura 9.4: Um exemplo de uma árvore rubro-negra com altura preta 3. Os nós externos (`nil`) são desenhados como quadrados.

Property 9.4 (sem-borda-vermelha)**.** Dois nós vermelhos nunca são adjacentes. (Para qualquer nó `u`, exceto a raiz, $\texttt{u.colour} + \texttt{u.parent.colour} \geq 1$.)

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 Red-Black Trees and 2-4 Trees

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 Figura 9.5. Considere qualquer árvore rubro-negra, T, tendo `n` nós e executando a seguinte transformação: Remove cada nó vermelho `u` e conecta os dois filhos de `u` diretamente no pai (preto) de `u`. Após essa transformação nós ficamos com uma árvore T' tendo apenas nós pretos.

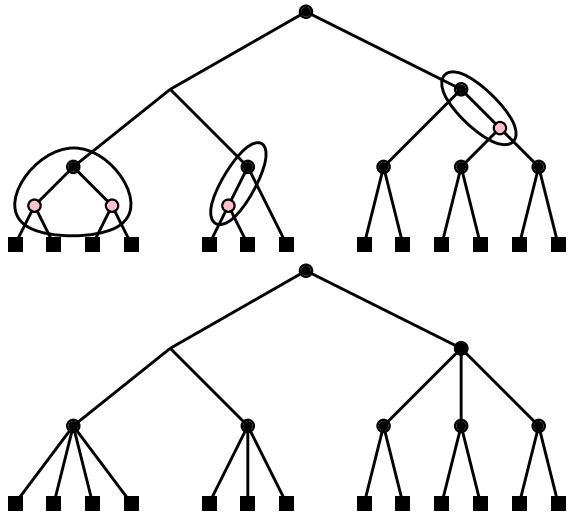


Figura 9.5: Toda árvore rubro-negra possui uma árvore 2-4 correspondente.

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 `BinarySearchTree` 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 Árvores Rubro-Negras caindo pra 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

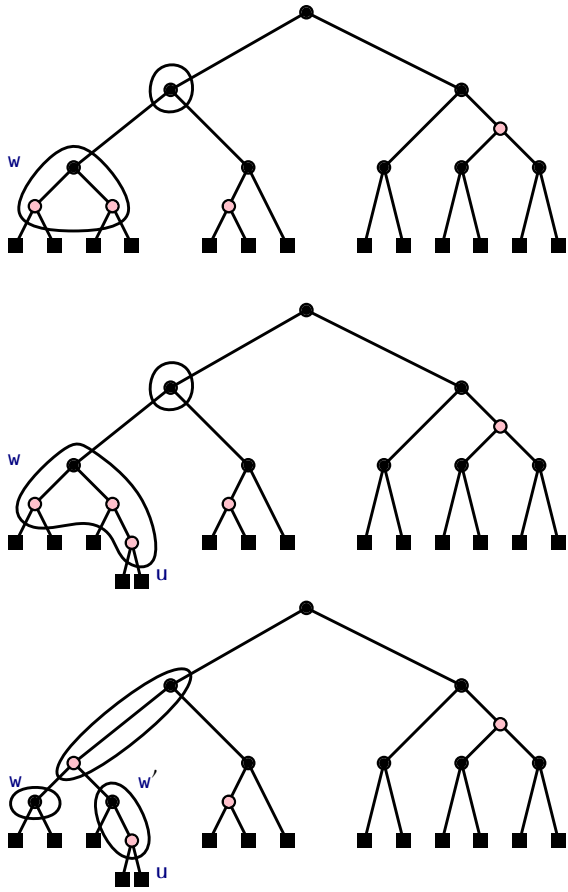


Figura 9.6: Simulando uma operação de divisão em árvore 2-4 durante uma adição em uma árvore rubro-negra. (Isto simula a adição em árvore 2-4 mostrada em Figura 9.2.)

estrutura implementa uma variante particular das árvores rubro-negras que satisfaz uma propriedade adicional:

Property 9.5 (left-leaning)**.** 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 caindo-pra-esquerda; ela é violada pelo pai do nó vermelho no caminho mais à direita.

O motivo para manter a propriedade caindo-pra-equerda é que ele reduz o número de casos encontrados ao atualizar a árvore durante `add(x)` e `remove(x)` operações. Em termos de 2-4 árvores, isso implica que cada 2-4 árvore 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 é vermelho e cujo filho 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 em Figura 9.7. As duas primeiras sub-rotinas são para manipular cores, preservando a prorpriedade 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:

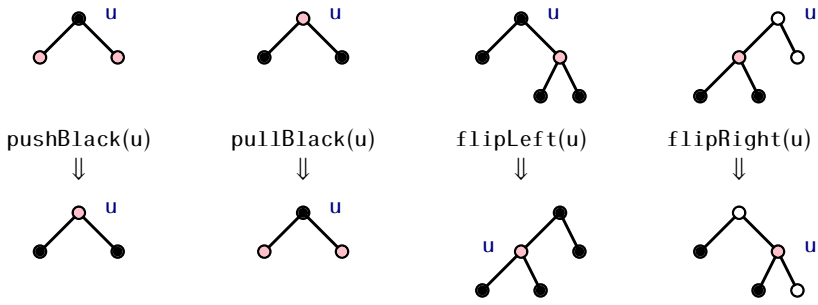


Figura 9.7: Flips, pulls and pushes

RedBlackTree
```
void flipLeft(Node *u) {
  swapcolours(u, u->right);
  rotateLeft(u);
}
```

A operação `flipLeft(u)` é especialmente útil para restaurar a propriedade caindo-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 caindo-pra-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 caindo-pra-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 caindo-pra-esquerda e sem-borda-vermelha se mantenham.

Caso contrário, primeiro determinamos se o pai de u, w, viola a propriedade caindo-pra-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 caindo-pra-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 ver-

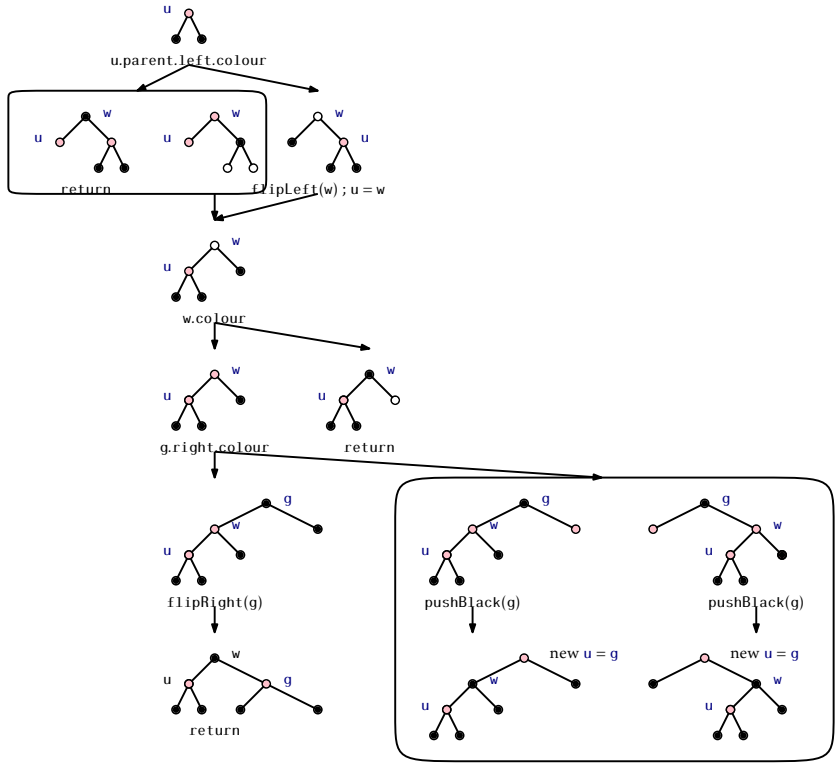


Figura 9.8: Uma única rodada no processo de fixação da propriedade 2 após uma inserção.

melho, então a propriedade caindo-pra-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 caindo-pra-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 Removal

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 por fazer `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 caindo-pra-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 caindo-pra 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 caindo-pra-esquerda). Nós realizamos um right-flip direito em w e depois prosseguimos para a próxima

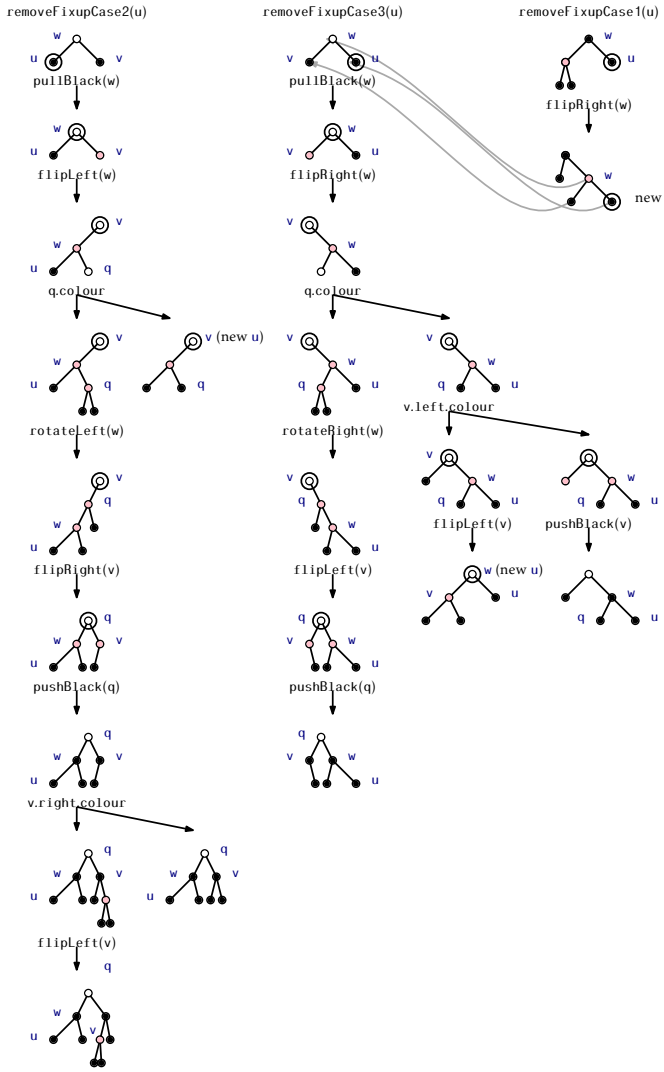


Figura 9.9: Uma única rodada no processo de eliminar um nó duplo-preto após uma remoção.

iteração. Observe que esta ação faz com que o pai de w viole a propriedade caindo-pra-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 deu, 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 caindo-pra-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 caindo-pra-esquerda são violadas em q e w, respectivamente. A propriedade caindo-pra-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 caindo-pra-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 caindo-pra-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` black. 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 caindo-pra-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 caindo-pra-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 Summary

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 de 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 ávore 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 [37]. 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 [63] descreve duas versões de árvores rubro-negras caindo pra 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 *height-balanced*: 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 $F(0) = 1$ and $F(1) = 1$. 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

$$h \leq \log_\varphi \mathtt{n} \approx 1.440420088 \log \mathtt{n} \ ,$$

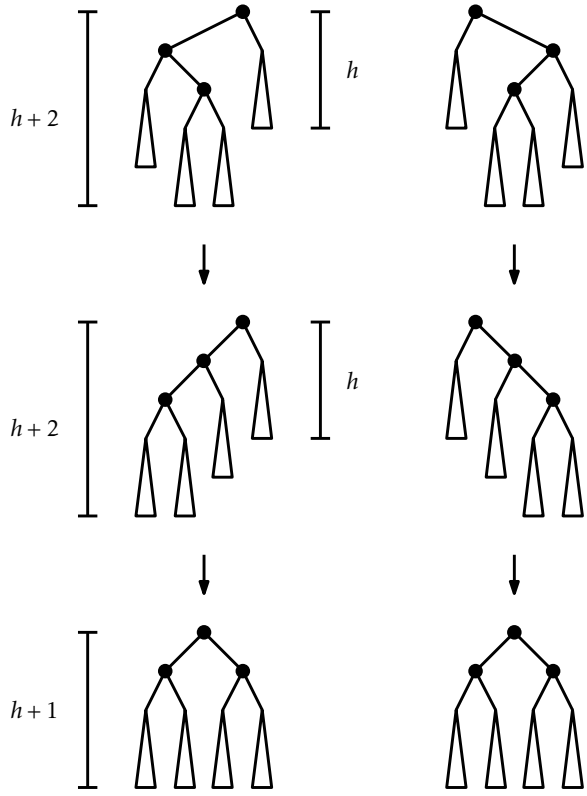


Figura 9.10: Reequilibrando em uma árvore AVL. No máximo, duas rotações são necessárias para converter um nó cujas subárvores tenham uma altura de h and $h+2$ em um nó cujas subárvores têm uma altura de no máximo $h+1$.

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 `c` definida aqui. Infelizmente, nenhuma delas pode garantir que o tempo amortizado gasto rebalanceamento é $O(1)$ por atualização. Em particular,

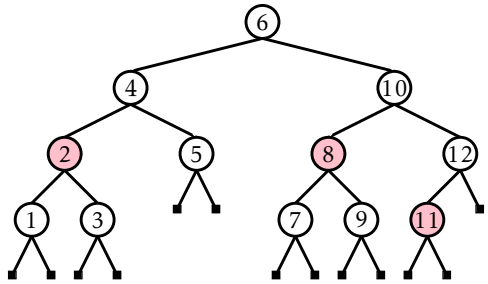


Figura 9.11: Uma árvore rubro-negra para praticar.

não há analogias de Teorema 9.2 para essas estruturas.

Exercício 9.1. Ilustre a árvore 2-4 que corresponde à `RedBlackTree` em Figura 9.11.

Exercício 9.2. Ilustre a adição de 13, depois de 3.5, e depois de 3.3 na `RedBlackTree` em Figura 9.11.

Exercício 9.3. Ilustre a remoção de 11, depois de 9, e depois de 5 na `RedBlackTree` em Figura 9.11.

Exercício 9.4. Mostre que, para valores arbitrariamente grandes de n, há vermelho-preto árvores 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 vermelho-pretas 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 a propriedade caindo-pra-esquerda?

Exercício 9.10. Suponha que você tenha duas árvores vermelho-pretas T_1 and 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 and 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

In this chapter, we discuss two implementations of the extremely useful priority `Queue` data structure. Both of these structures are a special kind of binary tree called a *heap*, which means "a disorganized pile." This is in contrast to binary search trees that can be thought of as a highly organized pile.

The first heap implementation uses an array to simulate a complete binary tree. This very fast implementation is the basis of one of the fastest known sorting algorithms, namely heapsort (see Seção 11.1.3). The second implementation is based on more flexible binary trees. It supports a `meld(h)` operation that allows the priority queue to absorb the elements of a second priority queue `h`.

10.1 `BinaryHeap`: An Implicit Binary Tree

Our first implementation of a (priority) `Queue` is based on a technique that is over four hundred years old. *Eytzinger's method* allows us to represent a complete binary tree as an array by laying out the nodes of the tree in breadth-first order (see Seção 6.1.2). In this way, the root is stored at position 0, the root's left child is stored at position 1, the root's right child at position 2, the left child of the left child of the root is stored at position 3, and so on. See Figura 10.1.

If we apply Eytzinger's method to a sufficiently large tree, some patterns emerge. The left child of the node at index `i` is at index `left(i)` =

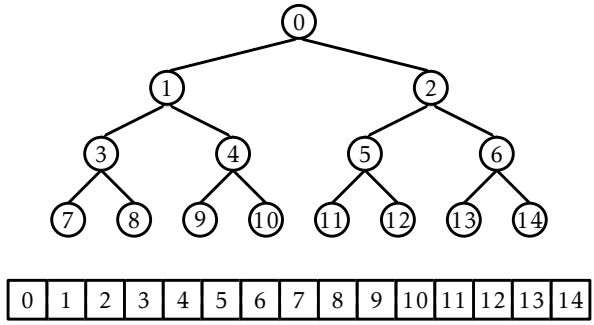


Figura 10.1: Eytzinger's method represents a complete binary tree as an array.

$2i+1$ and the right child of the node at index `i` is at index $\texttt{right}(i) = 2i+2$. The parent of the node at index `i` is at index $\texttt{parent}(i) = (i-1)/2$.

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;
}
```

A `BinaryHeap` uses this technique to implicitly represent a complete binary tree in which the elements are *heap-ordered*: The value stored at any index `i` is not smaller than the value stored at index `parent(i)`, with the exception of the root value, $i = 0$. It follows that the smallest value in the priority `Queue` is therefore stored at position 0 (the root).

In a `BinaryHeap`, the `n` elements are stored in an array `a`:

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

Implementing the `add(x)` operation is fairly straightforward. As with all array-based structures, we first check to see if `a` is full (by checking if

a.length = n) and, if so, we grow a. Next, we place x at location a[n] and increment n. At this point, all that remains is to ensure that we maintain the heap property. We do this by repeatedly swapping x with its parent until x is no longer smaller than its parent. See 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);
  }
}
```

Implementing the remove() operation, which removes the smallest value from the heap, is a little trickier. We know where the smallest value is (at the root), but we need to replace it after we remove it and ensure that we maintain the heap property.

The easiest way to do this is to replace the root with the value a[n − 1], delete that value, and decrement n. Unfortunately, the new root element is now probably not the smallest element, so it needs to be moved downwards. We do this by repeatedly comparing this element to its two children. If it is the smallest of the three then we are done. Otherwise, we swap this element with the smallest of its two children and continue.

BinaryHeap

```
T remove() {
  T x = a[0];
  a[0] = a[--n];
  trickleDown(0);
  if (3*n < a.length) resize();
  return x;
}
void trickleDown(int i) {
```

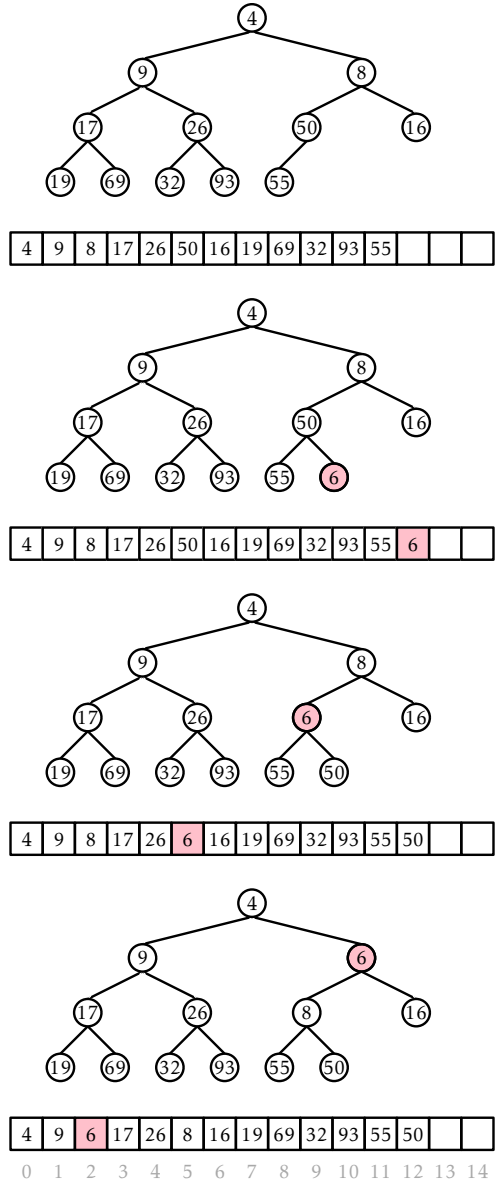


Figura 10.2: Adding the value 6 to a `BinaryHeap`.

```
  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);
}
```

As with other array-based structures, we will ignore the time spent in calls to `resize()`, since these can be accounted for using the amortization argument from Lema 2.1. The running times of both `add(x)` and `remove()` then depend on the height of the (implicit) binary tree. Luckily, this is a *complete* binary tree; every level except the last has the maximum possible number of nodes. Therefore, if the height of this tree is h, then it has at least 2^h nodes. Stated another way

$$n \geq 2^h \ .$$

Taking logarithms on both sides of this equation gives

$$h \leq \log n \ .$$

Therefore, both the `add(x)` and `remove()` operation run in $O(\log n)$ time.

10.1.1 Summary

The following theorem summarizes the performance of a `BinaryHeap`:

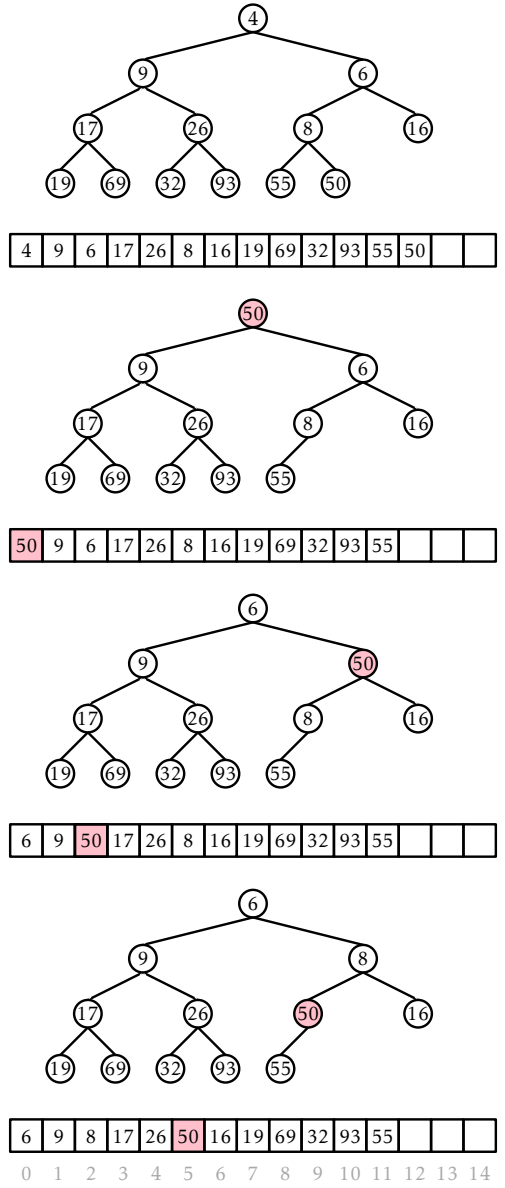


Figura 10.3: Removing the minimum value, 4, from a `BinaryHeap`.

Teorema 10.1. *A* `BinaryHeap` *implements the (priority)* `Queue` *interface. Ignoring the cost of calls to* `resize()`*, a* `BinaryHeap` *supports the operations* `add(x)` *and* `remove()` *in* $O(\log n)$ *time per operation.*

Furthermore, beginning with an empty `BinaryHeap`*, any sequence of* m `add(x)` *and* `remove()` *operations results in a total of* $O(m)$ *time spent during all calls to* `resize()`*.*

10.2 MeldableHeap: A Randomized Meldable Heap

In this section, we describe the `MeldableHeap`, a priority `Queue` implementation in which the underlying structure is also a heap-ordered binary tree. However, unlike a `BinaryHeap` in which the underlying binary tree is completely defined by the number of elements, there are no restrictions on the shape of the binary tree that underlies a `MeldableHeap`; anything goes.

The `add(x)` and `remove()` operations in a `MeldableHeap` are implemented in terms of the `merge(h1,h2)` operation. This operation takes two heap nodes `h1` and `h2` and merges them, returning a heap node that is the root of a heap that contains all elements in the subtree rooted at `h1` and all elements in the subtree rooted at `h2`.

The nice thing about a `merge(h1,h2)` operation is that it can be defined recursively. See Figura 10.4. If either `h1` or `h2` is `nil`, then we are merging with an empty set, so we return `h2` or `h1`, respectively. Otherwise, assume $\texttt{h1.x} \leq \texttt{h2.x}$ since, if $\texttt{h1.x} > \texttt{h2.x}$, then we can reverse the roles of `h1` and `h2`. Then we know that the root of the merged heap will contain `h1.x`, and we can recursively merge `h2` with `h1.left` or `h1.right`, as we wish. This is where randomization comes in, and we toss a coin to decide whether to merge `h2` with `h1.left` or `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);
```

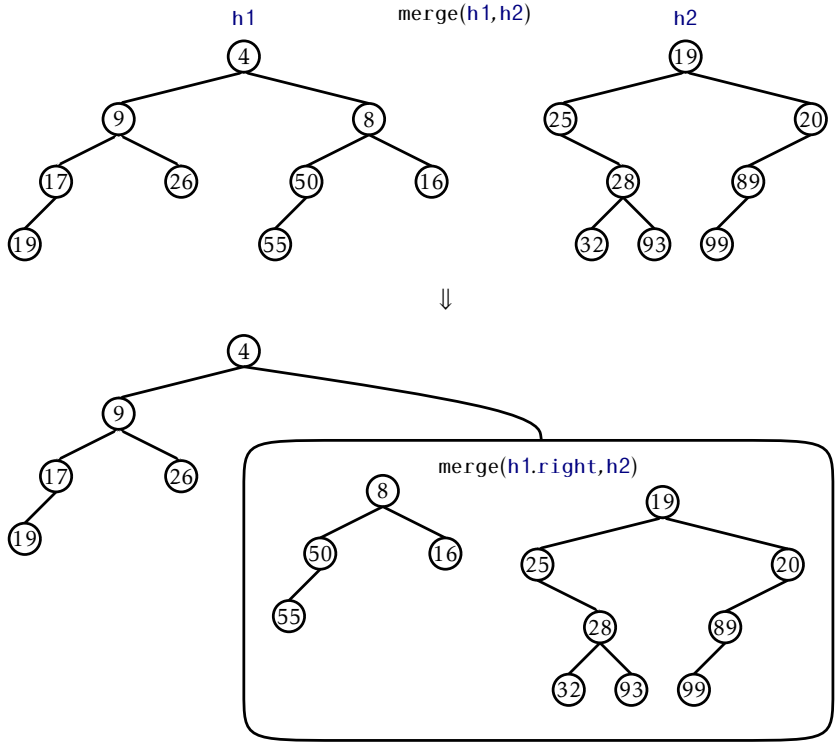


Figura 10.4: Merging h1 and h2 is done by merging h2 with one of h1.left or h1.right.

```
    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;
}
```

In the next section, we show that merge(h1,h2) runs in $O(\log n)$ expected time, where n is the total number of elements in h1 and h2.

With access to a merge(h1,h2) operation, the add(x) operation is easy. We create a new node u containing x and then merge u with the root of our 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;
}
```

This takes $O(\log(n+1)) = O(\log n)$ expected time.

The `remove()` operation is similarly easy. The node we want to remove is the root, so we just merge its two children and make the result the root:

MeldableHeap

```
T remove() {
  T x = r->x;
  Node *tmp = r;
  r = merge(r->left, r->right);
  delete tmp;
  if (r != nil) r->parent = nil;
  n--;
  return x;
}
```

Again, this takes $O(\log n)$ expected time.

Additionally, a `MeldableHeap` can implement many other operations in $O(\log n)$ expected time, including:

- `remove(u)`: remove the node `u` (and its key `u.x`) from the heap.
- `absorb(h)`: add all the elements of the `MeldableHeap` `h` to this heap, emptying `h` in the process.

Each of these operations can be implemented using a constant number of `merge(h1,h2)` operations that each take $O(\log n)$ expected time.

10.2.1 Analysis of `merge(h1,h2)`

The analysis of `merge(h1,h2)` is based on the analysis of a random walk in a binary tree. A *random walk* in a binary tree starts at the root of the tree. At each step in the random walk, a coin is tossed and, depending on the result of this coin toss, the walk proceeds to the left or to the right child of the current node. The walk ends when it falls off the tree (the current node becomes `nil`).

The following lemma is somewhat remarkable because it does not depend at all on the shape of the binary tree:

Lema 10.1. *The expected length of a random walk in a binary tree with* n *nodes is at most* $\log(n+1)$.

Demonstração. The proof is by induction on n. In the base case, $n = 0$ and the walk has length $0 = \log(n+1)$. Suppose now that the result is true for all non-negative integers $n' < n$.

Let n_1 denote the size of the root's left subtree, so that $n_2 = n - n_1 - 1$ is the size of the root's right subtree. Starting at the root, the walk takes one step and then continues in a subtree of size n_1 or n_2. By our inductive hypothesis, the expected length of the walk is then

$$\mathrm{E}[W] = 1 + \frac{1}{2}\log(n_1+1) + \frac{1}{2}\log(n_2+1) \ ,$$

since each of n_1 and n_2 are less than n. Since log is a concave function, $\mathrm{E}[W]$ is maximized when $n_1 = n_2 = (n-1)/2$. Therefore, the expected number of steps taken by the random walk is

$$\begin{aligned}\mathrm{E}[W] &= 1 + \frac{1}{2}\log(n_1+1) + \frac{1}{2}\log(n_2+1) \\ &\le 1 + \log((n-1)/2+1) \\ &= 1 + \log((n+1)/2) \\ &= \log(n+1) \ .\end{aligned}$$

□

We make a quick digression to note that, for readers who know a little about information theory, the proof of Lema 10.1 can be stated in terms of entropy.

Information Theoretic Proof of Lema 10.1. Let d_i denote the depth of the ith external node and recall that a binary tree with n nodes has $n+1$ external nodes. The probability of the random walk reaching the ith external node is exactly $p_i = 1/2^{d_i}$, so the expected length of the random walk is given by

$$H = \sum_{i=0}^{n} p_i d_i = \sum_{i=0}^{n} p_i \log\left(2^{d_i}\right) = \sum_{i=0}^{n} p_i \log(1/p_i)$$

The right hand side of this equation is easily recognizable as the entropy of a probability distribution over $n+1$ elements. A basic fact about the entropy of a distribution over $n+1$ elements is that it does not exceed $\log(n+1)$, which proves the lemma. □

With this result on random walks, we can now easily prove that the running time of the `merge(h1,h2)` operation is $O(\log n)$.

Lema 10.2. *If* `h1` *and* `h2` *are the roots of two heaps containing* n_1 *and* n_2 *nodes, respectively, then the expected running time of* `merge(h1,h2)` *is at most* $O(\log n)$, *where* $n = n_1 + n_2$.

Demonstração. Each step of the merge algorithm takes one step of a random walk, either in the heap rooted at `h1` or the heap rooted at `h2`. The algorithm terminates when either of these two random walks fall out of its corresponding tree (when `h1 = null` or `h2 = null`). Therefore, the expected number of steps performed by the merge algorithm is at most

$$\log(n_1+1) + \log(n_2+1) \leq 2\log n \ .$$ □

10.2.2 Summary

The following theorem summarizes the performance of a `MeldableHeap`:

Teorema 10.2. *A* `MeldableHeap` *implements the (priority)* `Queue` *interface. A* `MeldableHeap` *supports the operations* `add(x)` *and* `remove()` *in* $O(\log n)$ *expected time per operation.*

10.3 Discussion and Exercises

The implicit representation of a complete binary tree as an array, or list, seems to have been first proposed by Eytzinger [26]. He used this representation in books containing pedigree family trees of noble families. The `BinaryHeap` data structure described here was first introduced by Williams [75].

The randomized `MeldableHeap` data structure described here appears to have first been proposed by Gambin and Malinowski [33]. Other meldable heap implementations exist, including leftist heaps [15, 47, Section 5.3.2], binomial heaps [72], Fibonacci heaps [29], pairing heaps [28], and skew heaps [69], although none of these are as simple as the `MeldableHeap` structure.

Some of the above structures also support a `decreaseKey(u, y)` operation in which the value stored at node `u` is decreased to `y`. (It is a precondition that $y \leq u.x$.) In most of the preceding structures, this operation can be supported in $O(\log n)$ time by removing node `u` and adding `y`. However, some of these structures can implement `decreaseKey(u, y)` more efficiently. In particular, `decreaseKey(u, y)` takes $O(1)$ amortized time in Fibonacci heaps and $O(\log \log n)$ amortized time in a special version of pairing heaps [24]. This more efficient `decreaseKey(u, y)` operation has applications in speeding up several graph algorithms, including Dijkstra's shortest path algorithm [29].

Exercício 10.1. Illustrate the addition of the values 7 and then 3 to the `BinaryHeap` shown at the end of Figura 10.2.

Exercício 10.2. Illustrate the removal of the next two values (6 and 8) on the `BinaryHeap` shown at the end of Figura 10.3.

Exercício 10.3. Implement the `remove(i)` method, that removes the value stored in `a[i]` in a `BinaryHeap`. This method should run in $O(\log n)$ time. Next, explain why this method is not likely to be useful.

Exercício 10.4. A d-ary tree is a generalization of a binary tree in which each internal node has d children. Using Eytzinger's method it is also possible to represent complete d-ary trees using arrays. Work out the

equations that, given an index `i`, determine the index of `i`'s parent and each of `i`'s d children in this representation.

Exercício 10.5. Using what you learned in Exercício 10.4, design and implement a *DaryHeap*, the d-ary generalization of a `BinaryHeap`. Analyze the running times of operations on a `DaryHeap` and test the performance of your `DaryHeap` implementation against that of the `BinaryHeap` implementation given here.

Exercício 10.6. Illustrate the addition of the values 17 and then 82 in the `MeldableHeap` `h1` shown in Figura 10.4. Use a coin to simulate a random bit when needed.

Exercício 10.7. Illustrate the removal of the next two values (4 and 8) in the `MeldableHeap` `h1` shown in Figura 10.4. Use a coin to simulate a random bit when needed.

Exercício 10.8. Implement the `remove(u)` method, that removes the node `u` from a `MeldableHeap`. This method should run in $O(\log n)$ expected time.

Exercício 10.9. Show how to find the second smallest value in a `BinaryHeap` or `MeldableHeap` in constant time.

Exercício 10.10. Show how to find the kth smallest value in a `BinaryHeap` or `MeldableHeap` in $O(k \log k)$ time. (Hint: Using another heap might help.)

Exercício 10.11. Suppose you are given `k` sorted lists, of total length `n`. Using a heap, show how to merge these into a single sorted list in $O(n \log k)$ time. (Hint: Starting with the case $k = 2$ can be instructive.)

Capítulo 11

Sorting Algorithms

This chapter discusses algorithms for sorting a set of n items. This might seem like a strange topic for a book on data structures, but there are several good reasons for including it here. The most obvious reason is that two of these sorting algorithms (quicksort and heap-sort) are intimately related to two of the data structures we have already studied (random binary search trees and heaps, respectively).

The first part of this chapter discusses algorithms that sort using only comparisons and presents three algorithms that run in $O(n \log n)$ time. As it turns out, all three algorithms are asymptotically optimal; no algorithm that uses only comparisons can avoid doing roughly $n \log n$ comparisons in the worst case and even the average case.

Before continuing, we should note that any of the `SSet` or priority `Queue` implementations presented in previous chapters can also be used to obtain an $O(n \log n)$ time sorting algorithm. For example, we can sort n items by performing n `add(x)` operations followed by n `remove()` operations on a `BinaryHeap` or `MeldableHeap`. Alternatively, we can use n `add(x)` operations on any of the binary search tree data structures and then perform an in-order traversal (Exercício 6.8) to extract the elements in sorted order. However, in both cases we go through a lot of overhead to build a structure that is never fully used. Sorting is such an important problem that it is worthwhile developing direct methods that are as fast, simple, and space-efficient as possible.

The second part of this chapter shows that, if we allow other operations besides comparisons, then all bets are off. Indeed, by using array-

indexing, it is possible to sort a set of n integers in the range $\{0,\ldots,n^c-1\}$ in $O(cn)$ time.

11.1 Comparison-Based Sorting

In this section, we present three sorting algorithms: merge-sort, quick-sort, and heap-sort. Each of these algorithms takes an input array a and sorts the elements of a into non-decreasing order in $O(n \log n)$ (expected) time. These algorithms are all *comparison-based*. These algorithms don't care what type of data is being sorted; the only operation they do on the data is comparisons using the compare(a,b) method. Recall, from Seção 1.2.4, that compare(a,b) returns a negative value if $a < b$, a positive value if $a > b$, and zero if $a = b$.

11.1.1 Merge-Sort

The *merge-sort* algorithm is a classic example of recursive divide and conquer: If the length of a is at most 1, then a is already sorted, so we do nothing. Otherwise, we split a into two halves, $a0 = a[0],\ldots,a[n/2-1]$ and $a1 = a[n/2],\ldots,a[n-1]$. We recursively sort a0 and a1, and then we merge (the now sorted) a0 and a1 to get our fully sorted array 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);
}
```

An example is shown in Figura 11.1.

Compared to sorting, merging the two sorted arrays a0 and a1 is fairly easy. We add elements to a one at a time. If a0 or a1 is empty, then we add the next elements from the other (non-empty) array. Otherwise, we

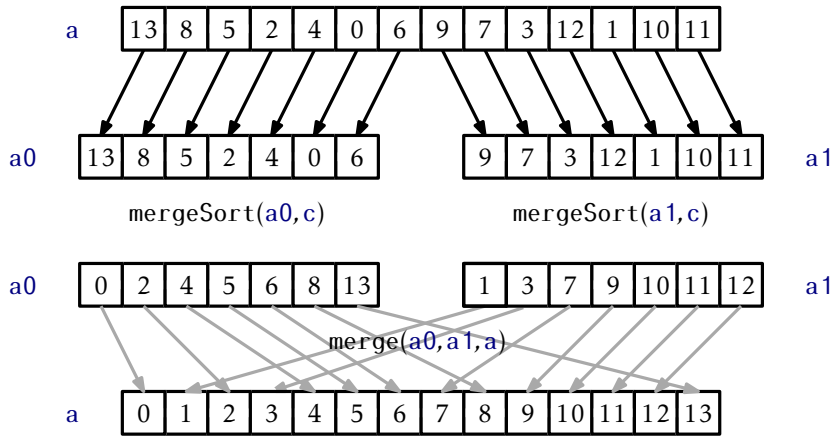


Figura 11.1: The execution of `mergeSort(a,c)`

take the minimum of the next element in `a0` and the next element in `a1` and add it to `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++];
  }
}
```

Notice that the `merge(a0,a1,a,c)` algorithm performs at most $n-1$ comparisons before running out of elements in one of `a0` or `a1`.

To understand the running-time of merge-sort, it is easiest to think of it in terms of its recursion tree. Suppose for now that n is a power of two, so that $n = 2^{\log n}$, and $\log n$ is an integer. Refer to Figura 11.2. Merge-sort turns the problem of sorting n elements into two problems, each of

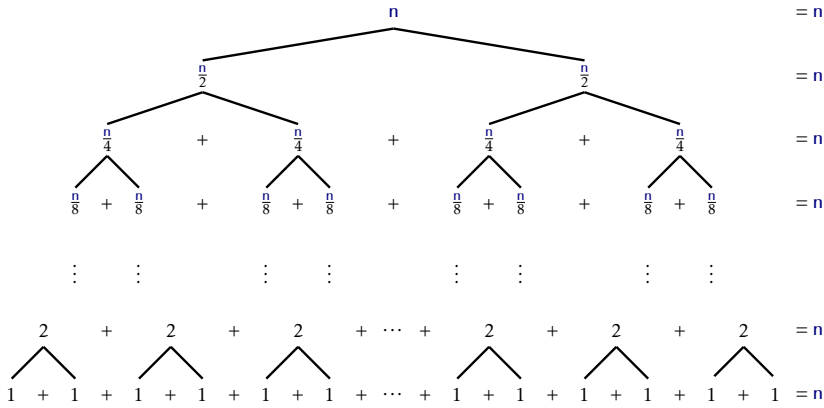


Figura 11.2: The merge-sort recursion tree.

sorting $n/2$ elements. These two subproblem are then turned into two problems each, for a total of four subproblems, each of size $n/4$. These four subproblems become eight subproblems, each of size $n/8$, and so on. At the bottom of this process, $n/2$ subproblems, each of size two, are converted into n problems, each of size one. For each subproblem of size $n/2^i$, the time spent merging and copying data is $O(n/2^i)$. Since there are 2^i subproblems of size $n/2^i$, the total time spent working on problems of size 2^i, not counting recursive calls, is

$$2^i \times O(n/2^i) = O(n) \ .$$

Therefore, the total amount of time taken by merge-sort is

$$\sum_{i=0}^{\log n} O(n) = O(n \log n) \ .$$

The proof of the following theorem is based on preceding analysis, but has to be a little more careful to deal with the cases where n is not a power of 2.

Teorema 11.1. *The mergeSort(a) algorithm runs in $O(n \log n)$ time and performs at most $n \log n$ comparisons.*

Demonstração. The proof is by induction on n. The base case, in which $n = 1$, is trivial; when presented with an array of length 0 or 1 the algorithm simply returns without performing any comparisons.

Merging two sorted lists of total length n requires at most $n-1$ comparisons. Let $C(n)$ denote the maximum number of comparisons performed by `mergeSort(a,c)` on an array `a` of length n. If n is even, then we apply the inductive hypothesis to the two subproblems and obtain

$$\begin{aligned} C(n) &\leq n - 1 + 2C(n/2) \\ &\leq n - 1 + 2((n/2)\log(n/2)) \\ &= n - 1 + n\log(n/2) \\ &= n - 1 + n\log n - n \\ &< n\log n \ . \end{aligned}$$

The case where n is odd is slightly more complicated. For this case, we use two inequalities that are easy to verify:

$$\log(x+1) \leq \log(x) + 1 \ , \tag{11.1}$$

for all $x \geq 1$ and

$$\log(x + 1/2) + \log(x - 1/2) \leq 2\log(x) \ , \tag{11.2}$$

for all $x \geq 1/2$. Inequality (11.1) comes from the fact that $\log(x) + 1 = \log(2x)$ while (11.2) follows from the fact that log is a concave function. With these tools in hand we have, for odd n,

$$\begin{aligned} C(n) &\leq n - 1 + C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) \\ &\leq n - 1 + \lceil n/2 \rceil \log \lceil n/2 \rceil + \lfloor n/2 \rfloor \log \lfloor n/2 \rfloor \\ &= n - 1 + (n/2 + 1/2)\log(n/2 + 1/2) + (n/2 - 1/2)\log(n/2 - 1/2) \\ &\leq n - 1 + n\log(n/2) + (1/2)(\log(n/2 + 1/2) - \log(n/2 - 1/2)) \\ &\leq n - 1 + n\log(n/2) + 1/2 \\ &< n + n\log(n/2) \\ &= n + n(\log n - 1) \\ &= n\log n \ . \end{aligned}$$

□

11.1.2 Quicksort

The *quicksort* algorithm is another classic divide and conquer algorithm. Unlike merge-sort, which does merging after solving the two subproblems, quicksort does all of its work upfront.

Quicksort is simple to describe: Pick a random *pivot* element, `x`, from `a`; partition `a` into the set of elements less than `x`, the set of elements equal to `x`, and the set of elements greater than `x`; and, finally, recursively sort the first and third sets in this partition. An example is shown in 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 {
      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));
}
```

All of this is done in place, so that instead of making copies of subarrays being sorted, the `quickSort(a,i,n,c)` method only sorts the subarray $a[i],\ldots,a[i+n-1]$. Initially, this method is invoked with the arguments `quickSort(a,0,a.length,c)`.

At the heart of the quicksort algorithm is the in-place partitioning al-

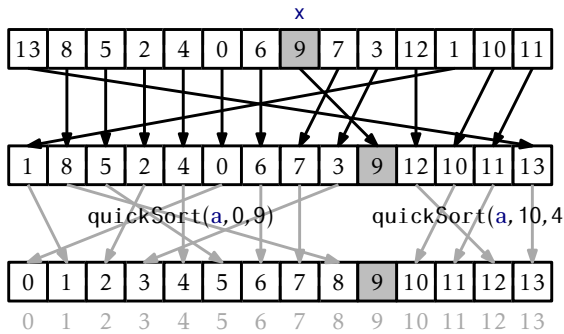


Figura 11.3: An example execution of `quickSort(a,0,14)`

gorithm. This algorithm, without using any extra space, swaps elements in `a` and computes indices `p` and `q` so that

$$a[i]\begin{cases} < x & \text{if } 0 \le i \le p \\ = x & \text{if } p < i < q \\ > x & \text{if } q \le i \le n-1 \end{cases}$$

This partitioning, which is done by the `while` loop in the code, works by iteratively increasing `p` and decreasing `q` while maintaining the first and last of these conditions. At each step, the element at position `j` is either moved to the front, left where it is, or moved to the back. In the first two cases, `j` is incremented, while in the last case, `j` is not incremented since the new element at position `j` has not yet been processed.

Quicksort is very closely related to the random binary search trees studied in Seção 7.1. In fact, if the input to quicksort consists of `n` distinct elements, then the quicksort recursion tree is a random binary search tree. To see this, recall that when constructing a random binary search tree the first thing we do is pick a random element `x` and make it the root of the tree. After this, every element will eventually be compared to `x`, with smaller elements going into the left subtree and larger elements into the right.

In quicksort, we select a random element `x` and immediately compare everything to `x`, putting the smaller elements at the beginning of the array and larger elements at the end of the array. Quicksort then recursively sorts the beginning of the array and the end of the array, while the random

binary search tree recursively inserts smaller elements in the left subtree of the root and larger elements in the right subtree of the root.

The above correspondence between random binary search trees and quicksort means that we can translate Lema 7.1 to a statement about quicksort:

Lema 11.1. *When quicksort is called to sort an array containing the integers* $0, \ldots, n-1$*, the expected number of times element* i *is compared to a pivot element is at most* $H_{i+1} + H_{n-i}$.

A little summing up of harmonic numbers gives us the following theorem about the running time of quicksort:

Teorema 11.2. *When quicksort is called to sort an array containing* n *distinct elements, the expected number of comparisons performed is at most* $2n \ln n + O(n)$.

Demonstração. Let T be the number of comparisons performed by quicksort when sorting n distinct elements. Using Lema 11.1 and linearity of expectation, we have:

$$\begin{aligned} \mathrm{E}[T] &= \sum_{i=0}^{n-1} (H_{i+1} + H_{n-i}) \\ &= 2 \sum_{i=1}^{n} H_i \\ &\leq 2 \sum_{i=1}^{n} H_n \\ &\leq 2n \ln n + 2n = 2n \ln n + O(n) \end{aligned}$$

□

Teorema 11.3 describes the case where the elements being sorted are all distinct. When the input array, `a`, contains duplicate elements, the expected running time of quicksort is no worse, and can be even better; any time a duplicate element `x` is chosen as a pivot, all occurrences of `x` get grouped together and do not take part in either of the two subproblems.

Teorema 11.3. *The* `quickSort(a)` *method runs in* $O(n \log n)$ *expected time and the expected number of comparisons it performs is at most* $2n \ln n + O(n)$.

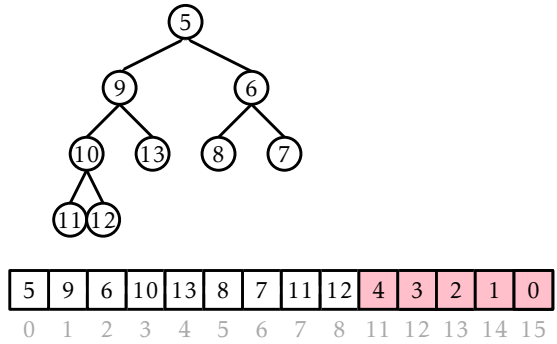


Figura 11.4: A snapshot of the execution of heapSort(a,c). The shaded part of the array is already sorted. The unshaded part is a BinaryHeap. During the next iteration, element 5 will be placed into array location 8.

11.1.3 Heap-sort

The *heap-sort* algorithm is another in-place sorting algorithm. Heap-sort uses the binary heaps discussed in Seção 10.1. Recall that the BinaryHeap data structure represents a heap using a single array. The heap-sort algorithm converts the input array a into a heap and then repeatedly extracts the minimum value.

More specifically, a heap stores n elements in an array, a, at array locations a[0],...,a[n−1] with the smallest value stored at the root, a[0]. After transforming a into a BinaryHeap, the heap-sort algorithm repeatedly swaps a[0] and a[n−1], decrements n, and calls trickleDown(0) so that a[0],...,a[n−2] once again are a valid heap representation. When this process ends (because n = 0) the elements of a are stored in decreasing order, so a is reversed to obtain the final sorted order.[1] Figura 11.4 shows an example of the execution of heapSort(a,c).

BinaryHeap

```
void sort(array<T> &b) {
  BinaryHeap<T> h(b);
  while (h.n > 1) {
```

[1] The algorithm could alternatively redefine the compare(x,y) function so that the heap sort algorithm stores the elements directly in ascending order.

```
    h.a.swap(--h.n, 0);
    h.trickleDown(0);
  }
  b = h.a;
  b.reverse();
}
```

A key subroutine in heap sort is the constructor for turning an unsorted array a into a heap. It would be easy to do this in $O(n \log n)$ time by repeatedly calling the BinaryHeap add(x) method, but we can do better by using a bottom-up algorithm. Recall that, in a binary heap, the children of a[i] are stored at positions a[2i + 1] and a[2i + 2]. This implies that the elements $a[\lfloor n/2 \rfloor], \ldots, a[n-1]$ have no children. In other words, each of $a[\lfloor n/2 \rfloor], \ldots, a[n-1]$ is a sub-heap of size 1. Now, working backwards, we can call trickleDown(i) for each $i \in \{\lfloor n/2 \rfloor - 1, \ldots, 0\}$. This works, because by the time we call trickleDown(i), each of the two children of a[i] are the root of a sub-heap, so calling trickleDown(i) makes a[i] into the root of its own subheap.

BinaryHeap

```
BinaryHeap(array<T> &b) : a(0) {
  a = b;
  n = a.length;
  for (int i = n/2-1; i >= 0; i--) {
    trickleDown(i);
  }
}
```

The interesting thing about this bottom-up strategy is that it is more efficient than calling add(x) n times. To see this, notice that, for n/2 elements, we do no work at all, for n/4 elements, we call trickleDown(i) on a subheap rooted at a[i] and whose height is one, for n/8 elements, we call trickleDown(i) on a subheap whose height is two, and so on. Since the work done by trickleDown(i) is proportional to the height of the sub-heap rooted at a[i], this means that the total work done is at most

$$\sum_{i=1}^{\log n} O((i-1)n/2^i) \leq \sum_{i=1}^{\infty} O(in/2^i) = O(n) \sum_{i=1}^{\infty} i/2^i = O(2n) = O(n) \ .$$

The second-last equality follows by recognizing that the sum $\sum_{i=1}^{\infty} i/2^i$ is equal, by definition of expected value, to the expected number of times we toss a coin up to and including the first time the coin comes up as heads and applying Lema 4.2.

The following theorem describes the performance of `heapSort(a,c)`.

Teorema 11.4. *The* `heapSort(a,c)` *method runs in* $O(n \log n)$ *time and performs at most* $2n \log n + O(n)$ *comparisons.*

Demonstração. The algorithm runs in three steps: (1) transforming a into a heap, (2) repeatedly extracting the minimum element from a, and (3) reversing the elements in a. We have just argued that step 1 takes $O(n)$ time and performs $O(n)$ comparisons. Step 3 takes $O(n)$ time and performs no comparisons. Step 2 performs n calls to `trickleDown(0)`. The ith such call operates on a heap of size $n-i$ and performs at most $2\log(n-i)$ comparisons. Summing this over i gives

$$\sum_{i=0}^{n-i} 2\log(n-i) \leq \sum_{i=0}^{n-i} 2\log n = 2n\log n$$

Adding the number of comparisons performed in each of the three steps completes the proof. □

11.1.4 A Lower-Bound for Comparison-Based Sorting

We have now seen three comparison-based sorting algorithms that each run in $O(n \log n)$ time. By now, we should be wondering if faster algorithms exist. The short answer to this question is no. If the only operations allowed on the elements of a are comparisons, then no algorithm can avoid doing roughly $n \log n$ comparisons. This is not difficult to prove, but requires a little imagination. Ultimately, it follows from the fact that

$$\log(n!) = \log n + \log(n-1) + \cdots + \log(1) = n\log n - O(n) \ .$$

(Proving this fact is left as Exercício 11.10.)

We will start by focusing our attention on deterministic algorithms like merge-sort and heap-sort and on a particular fixed value of n. Imagine such an algorithm is being used to sort n distinct elements. The key

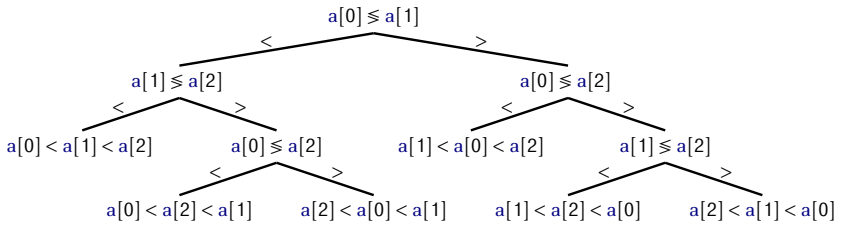


Figura 11.5: A comparison tree for sorting an array $a[0], a[1], a[2]$ of length $n = 3$.

to proving the lower-bound is to observe that, for a deterministic algorithm with a fixed value of n, the first pair of elements that are compared is always the same. For example, in `heapSort(a,c)`, when n is even, the first call to `trickleDown(i)` is with $i = n/2 - 1$ and the first comparison is between elements $a[n/2 - 1]$ and $a[n - 1]$.

Since all input elements are distinct, this first comparison has only two possible outcomes. The second comparison done by the algorithm may depend on the outcome of the first comparison. The third comparison may depend on the results of the first two, and so on. In this way, any deterministic comparison-based sorting algorithm can be viewed as a rooted binary *comparison tree*. Each internal node, u, of this tree is labelled with a pair of indices $u.i$ and $u.j$. If $a[u.i] < a[u.j]$ the algorithm proceeds to the left subtree, otherwise it proceeds to the right subtree. Each leaf w of this tree is labelled with a permutation $w.p[0], \ldots, w.p[n-1]$ of $0, \ldots, n-1$. This permutation represents the one that is required to sort a if the comparison tree reaches this leaf. That is,

$$a[w.p[0]] < a[w.p[1]] < \cdots < a[w.p[n-1]] \ .$$

An example of a comparison tree for an array of size $n = 3$ is shown in Figura 11.5.

The comparison tree for a sorting algorithm tells us everything about the algorithm. It tells us exactly the sequence of comparisons that will be performed for any input array, a, having n distinct elements and it tells us how the algorithm will reorder a in order to sort it. Consequently, the comparison tree must have at least $n!$ leaves; if not, then there are two distinct permutations that lead to the same leaf; therefore, the algorithm

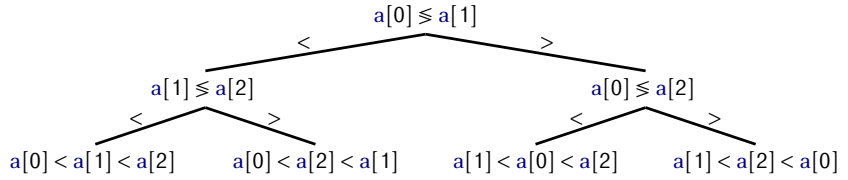


Figura 11.6: A comparison tree that does not correctly sort every input permutation.

does not correctly sort at least one of these permutations.

For example, the comparison tree in Figura 11.6 has only $4 < 3! = 6$ leaves. Inspecting this tree, we see that the two input arrays $3, 1, 2$ and $3, 2, 1$ both lead to the rightmost leaf. On the input $3, 1, 2$ this leaf correctly outputs $a[1] = 1, a[2] = 2, a[0] = 3$. However, on the input $3, 2, 1$, this node incorrectly outputs $a[1] = 2, a[2] = 1, a[0] = 3$. This discussion leads to the primary lower-bound for comparison-based algorithms.

Teorema 11.5. *For any deterministic comparison-based sorting algorithm $\mathcal{A}$ and any integer $n \geq 1$, there exists an input array a of length n such that $\mathcal{A}$ performs at least $\log(n!) = n \log n - O(n)$ comparisons when sorting a.*

Demonstração. By the preceding discussion, the comparison tree defined by $\mathcal{A}$ must have at least $n!$ leaves. An easy inductive proof shows that any binary tree with k leaves has a height of at least $\log k$. Therefore, the comparison tree for $\mathcal{A}$ has a leaf, w, with a depth of at least $\log(n!)$ and there is an input array a that leads to this leaf. The input array a is an input for which $\mathcal{A}$ does at least $\log(n!)$ comparisons. □

Teorema 11.5 deals with deterministic algorithms like merge-sort and heap-sort, but doesn't tell us anything about randomized algorithms like quicksort. Could a randomized algorithm beat the $\log(n!)$ lower bound on the number of comparisons? The answer, again, is no. Again, the way to prove it is to think differently about what a randomized algorithm is.

In the following discussion, we will assume that our decision trees have been "cleaned up" in the following way: Any node that can not be reached by some input array a is removed. This cleaning up implies that the tree has exactly $n!$ leaves. It has at least $n!$ leaves because, otherwise, it

could not sort correctly. It has at most n! leaves since each of the possible n! permutation of n distinct elements follows exactly one root to leaf path in the decision tree.

We can think of a randomized sorting algorithm, $\mathcal{R}$, as a deterministic algorithm that takes two inputs: The input array a that should be sorted and a long sequence $b = b_1, b_2, b_3, \ldots, b_m$ of random real numbers in the range $[0, 1]$. The random numbers provide the randomization for the algorithm. When the algorithm wants to toss a coin or make a random choice, it does so by using some element from b. For example, to compute the index of the first pivot in quicksort, the algorithm could use the formula $\lfloor nb_1 \rfloor$.

Now, notice that if we fix b to some particular sequence $\hat{b}$ then $\mathcal{R}$ becomes a deterministic sorting algorithm, $\mathcal{R}(\hat{b})$, that has an associated comparison tree, $\mathcal{T}(\hat{b})$. Next, notice that if we select a to be a random permutation of $\{1, \ldots, n\}$, then this is equivalent to selecting a random leaf, w, from the n! leaves of $\mathcal{T}(\hat{b})$.

Exercício 11.12 asks you to prove that, if we select a random leaf from any binary tree with k leaves, then the expected depth of that leaf is at least $\log k$. Therefore, the expected number of comparisons performed by the (deterministic) algorithm $\mathcal{R}(\hat{b})$ when given an input array containing a random permutation of $\{1, \ldots, n\}$ is at least $\log(n!)$. Finally, notice that this is true for every choice of $\hat{b}$, therefore it holds even for $\mathcal{R}$. This completes the proof of the lower-bound for randomized algorithms.

Teorema 11.6. *For any integer $n \geq 1$ and any (deterministic or randomized) comparison-based sorting algorithm $\mathcal{A}$, the expected number of comparisons done by $\mathcal{A}$ when sorting a random permutation of $\{1, \ldots, n\}$ is at least* $\log(n!) = n \log n - O(n)$.

11.2 Counting Sort and Radix Sort

In this section we study two sorting algorithms that are not comparison-based. Specialized for sorting small integers, these algorithms elude the lower-bounds of Teorema 11.5 by using (parts of) the elements in a as

indices into an array. Consider a statement of the form

$$c[a[i]] = 1 \ .$$

This statement executes in constant time, but has `c.length` possible different outcomes, depending on the value of `a[i]`. This means that the execution of an algorithm that makes such a statement cannot be modelled as a binary tree. Ultimately, this is the reason that the algorithms in this section are able to sort faster than comparison-based algorithms.

11.2.1 Counting Sort

Suppose we have an input array `a` consisting of `n` integers, each in the range $0,\ldots,k-1$. The *counting-sort* algorithm sorts `a` using an auxiliary array `c` of counters. It outputs a sorted version of `a` as an auxiliary array `b`.

The idea behind counting-sort is simple: For each $i \in \{0,\ldots,k-1\}$, count the number of occurrences of `i` in `a` and store this in `c[i]`. Now, after sorting, the output will look like `c[0]` occurrences of 0, followed by `c[1]` occurrences of 1, followed by `c[2]` occurrences of 2,..., followed by `c[k-1]` occurrences of `k-1`. The code that does this is very slick, and its execution is illustrated in 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;
}
```

The first `for` loop in this code sets each counter `c[i]` so that it counts the number of occurrences of `i` in `a`. By using the values of `a` as indices, these counters can all be computed in $O(n)$ time with a single for loop. At

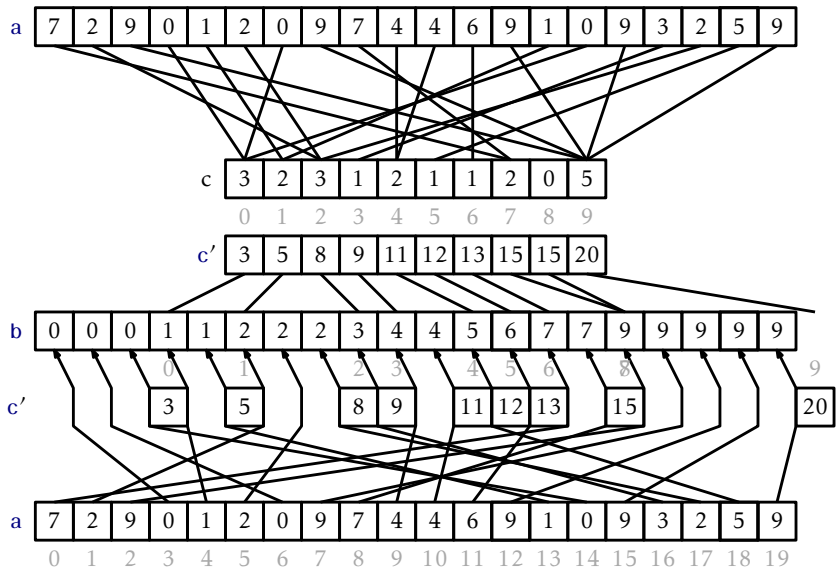


Figura 11.7: The operation of counting sort on an array of length $n = 20$ that stores integers $0, \ldots, k - 1 = 9$.

this point, we could use c to fill in the output array b directly. However, this would not work if the elements of a have associated data. Therefore we spend a little extra effort to copy the elements of a into b.

The next `for` loop, which takes $O(k)$ time, computes a running-sum of the counters so that $c[i]$ becomes the number of elements in a that are less than or equal to i. In particular, for every $i \in \{0,\ldots,k-1\}$, the output array, b, will have

$$b[c[i-1]] = b[c[i-1]+1] = \cdots = b[c[i]-1] = i \ .$$

Finally, the algorithm scans a backwards to place its elements, in order, into an output array b. When scanning, the element $a[i] = j$ is placed at location $b[c[j]-1]$ and the value $c[j]$ is decremented.

Teorema 11.7. *The* `countingSort(a,k)` *method can sort an array* a *containing* n *integers in the set* $\{0,\ldots,k-1\}$ *in* $O(n+k)$ *time.*

The counting-sort algorithm has the nice property of being *stable*; it preserves the relative order of equal elements. If two elements $a[i]$ and $a[j]$ have the same value, and $i < j$ then $a[i]$ will appear before $a[j]$ in b. This will be useful in the next section.

11.2.2 Radix-Sort

Counting-sort is very efficient for sorting an array of integers when the length, n, of the array is not much smaller than the maximum value, $k-1$, that appears in the array. The *radix-sort* algorithm, which we now describe, uses several passes of counting-sort to allow for a much greater range of maximum values.

Radix-sort sorts w-bit integers by using w/d passes of counting-sort to sort these integers d bits at a time.[2] More precisely, radix sort first sorts the integers by their least significant d bits, then their next significant d bits, and so on until, in the last pass, the integers are sorted by their most significant d bits.

Algorithms

```
void radixSort(array<int> &a) {
  int d = 8, w = 32;
```

[2] We assume that d divides w, otherwise we can always increase w to $d\lceil w/d \rceil$.

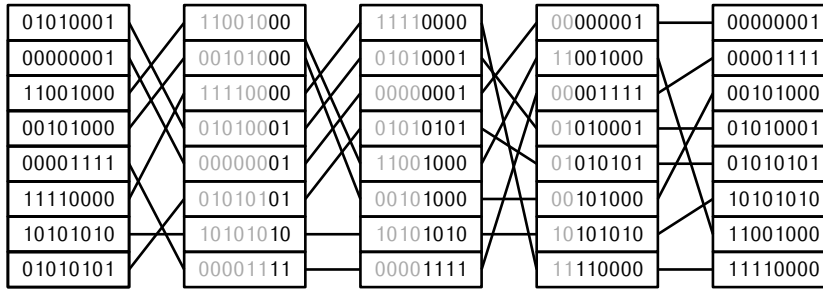


Figura 11.8: Using radixsort to sort `w` = 8-bit integers by using 4 passes of counting sort on `d` = 2-bit integers.

```
  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;
  }
}
```

(In this code, the expression $(\texttt{a[i]} >> \texttt{d}*\texttt{p})\&((1 << \texttt{d}) - 1)$ extracts the integer whose binary representation is given by bits $(\texttt{p}+1)\texttt{d}-1,\ldots,\texttt{pd}$ of `a[i]`.) An example of the steps of this algorithm is shown in Figura 11.8.

This remarkable algorithm sorts correctly because counting-sort is a stable sorting algorithm. If `x` < `y` are two elements of `a`, and the most significant bit at which `x` differs from `y` has index r, then `x` will be placed before `y` during pass $\lfloor r/\texttt{d} \rfloor$ and subsequent passes will not change the relative order of `x` and `y`.

Radix-sort performs `w/d` passes of counting-sort. Each pass requires $O(\texttt{n} + 2^{\texttt{d}})$ time. Therefore, the performance of radix-sort is given by the following theorem.

Teorema 11.8. *For any integer* $d > 0$, *the* `radixSort(a,k)` *method can sort an array* `a` *containing* `n` `w`*-bit integers in* $O((w/d)(n + 2^d))$ *time.*

If we think, instead, of the elements of the array being in the range $\{0,\ldots,n^c - 1\}$, and take $d = \lceil \log n \rceil$ we obtain the following version of Teorema 11.8.

Corollary 11.1. *The* `radixSort(a,k)` *method can sort an array* `a` *containing* `n` *integer values in the range* $\{0,\ldots,n^c - 1\}$ *in* $O(cn)$ *time.*

11.3 Discussion and Exercises

Sorting is *the* fundamental algorithmic problem in computer science, and it has a long history. Knuth [47] attributes the merge-sort algorithm to von Neumann (1945). Quicksort is due to Hoare [38]. The original heap-sort algorithm is due to Williams [75], but the version presented here (in which the heap is constructed bottom-up in $O(n)$ time) is due to Floyd [27]. Lower-bounds for comparison-based sorting appear to be folklore. The following table summarizes the performance of these comparison-based algorithms:

	comparisons		in-place
Merge-sort	$n \log n$	worst-case	No
Quicksort	$1.38n \log n + O(n)$	expected	Yes
Heap-sort	$2n \log n + O(n)$	worst-case	Yes

Each of these comparison-based algorithms has its advantages and disadvantages. Merge-sort does the fewest comparisons and does not rely on randomization. Unfortunately, it uses an auxilliary array during its merge phase. Allocating this array can be expensive and is a potential point of failure if memory is limited. Quicksort is an *in-place* algorithm and is a close second in terms of the number of comparisons, but is randomized, so this running time is not always guaranteed. Heap-sort does the most comparisons, but it is in-place and deterministic.

There is one setting in which merge-sort is a clear-winner; this occurs when sorting a linked-list. In this case, the auxiliary array is not needed; two sorted linked lists are very easily merged into a single sorted linked-list by pointer manipulations (see Exercício 11.2).

The counting-sort and radix-sort algorithms described here are due to Seward [65, Section 2.4.6]. However, variants of radix-sort have been used since the 1920s to sort punch cards using punched card sorting machines. These machines can sort a stack of cards into two piles based on the existence (or not) of a hole in a specific location on the card. Repeating this process for different hole locations gives an implementation of radix-sort.

Finally, we note that counting sort and radix-sort can be used to sort other types of numbers besides non-negative integers. Straightforward modifications of counting sort can sort integers, in any interval $\{a,\ldots,b\}$, in $O(n+b-a)$ time. Similarly, radix sort can sort integers in the same interval in $O(n(\log_n(b-a))$ time. Finally, both of these algorithms can also be used to sort floating point numbers in the IEEE 754 floating point format. This is because the IEEE format is designed to allow the comparison of two floating point numbers by comparing their values as if they were integers in a signed-magnitude binary representation [2].

Exercício 11.1. Illustrate the execution of merge-sort and heap-sort on an input array containing $1,7,4,6,2,8,3,5$. Give a sample illustration of one possible execution of quicksort on the same array.

Exercício 11.2. Implement a version of the merge-sort algorithm that sorts a `DLList` without using an auxiliary array. (See Exercício 3.13.)

Exercício 11.3. Some implementations of `quickSort(a,i,n,c)` always use `a[i]` as a pivot. Give an example of an input array of length `n` in which such an implementation would perform $\binom{n}{2}$ comparisons.

Exercício 11.4. Some implementations of `quickSort(a,i,n,c)` always use `a[i+n/2]` as a pivot. Given an example of an input array of length `n` in which such an implementation would perform $\binom{n}{2}$ comparisons.

Exercício 11.5. Show that, for any implementation of `quickSort(a,i,n,c)` that chooses a pivot deterministically, without first looking at any values in $a[i],\ldots,a[i+n-1]$, there exists an input array of length `n` that causes this implementation to perform $\binom{n}{2}$ comparisons.

Exercício 11.6. Design a `Comparator`, `c`, that you could pass as an argument to `quickSort(a,i,n,c)` and that would cause quicksort to perform

$\binom{n}{2}$ comparisons. (Hint: Your comparator does not actually need to look at the values being compared.)

Exercício 11.7. Analyze the expected number of comparisons done by Quicksort a little more carefully than the proof of Teorema 11.3. In particular, show that the expected number of comparisons is $2nH_n - n + H_n$.

Exercício 11.8. Describe an input array that causes heap sort to perform at least $2n \log n - O(n)$ comparisons. Justify your answer.

Exercício 11.9. Find another pair of permutations of $1, 2, 3$ that are not correctly sorted by the comparison tree in Figura 11.6.

Exercício 11.10. Prove that $\log n! = n \log n - O(n)$.

Exercício 11.11. Prove that a binary tree with k leaves has height at least $\log k$.

Exercício 11.12. Prove that, if we pick a random leaf from a binary tree with k leaves, then the expected height of this leaf is at least $\log k$.

Exercício 11.13. The implementation of `radixSort(a,k)` given here works when the input array, `a` contains only unsigned integers. Write a version that works correctly for signed integers.

Capítulo 12

Graphs

In this chapter, we study two representations of graphs and basic algorithms that use these representations.

Mathematically, a *(directed) graph* is a pair $G = (V, E)$ where V is a set of *vertices* and E is a set of ordered pairs of vertices called *edges*. An edge $(\mathtt{i}, \mathtt{j})$ is *directed* from $\mathtt{i}$ to $\mathtt{j}$; $\mathtt{i}$ is called the *source* of the edge and $\mathtt{j}$ is called the *target*. A *path* in G is a sequence of vertices $v_0, \ldots, v_k$ such that, for every $i \in \{1, \ldots, k\}$, the edge (v_{i-1}, v_i) is in E. A path $v_0, \ldots, v_k$ is a *cycle* if, additionally, the edge (v_k, v_0) is in E. A path (or cycle) is *simple* if all of its vertices are unique. If there is a path from some vertex v_i to some vertex v_j then we say that v_j is *reachable* from v_i. An example of a graph is shown in Figura 12.1.

Due to their ability to model so many phenomena, graphs have an enormous number of applications. There are many obvious examples. Computer networks can be modelled as graphs, with vertices corresponding to computers and edges corresponding to (directed) communication links between those computers. City streets can be modelled as graphs, with vertices representing intersections and edges representing streets joining consecutive intersections.

Less obvious examples occur as soon as we realize that graphs can model any pairwise relationships within a set. For example, in a university setting we might have a timetable *conflict graph* whose vertices represent courses offered in the university and in which the edge $(\mathtt{i}, \mathtt{j})$ is present if and only if there is at least one student that is taking both class $\mathtt{i}$ and class $\mathtt{j}$. Thus, an edge indicates that the exam for class $\mathtt{i}$ should not be

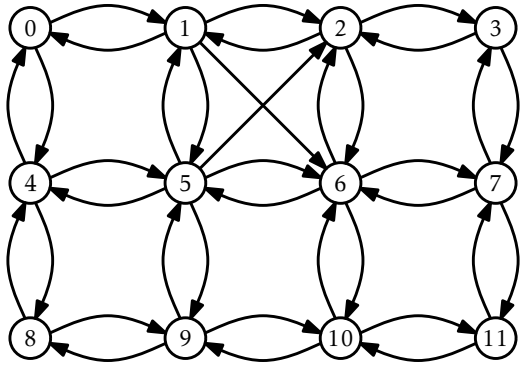


Figura 12.1: A graph with twelve vertices. Vertices are drawn as numbered circles and edges are drawn as pointed curves pointing from source to target.

scheduled at the same time as the exam for class `j`.

Throughout this section, we will use `n` to denote the number of vertices of G and `m` to denote the number of edges of G. That is, `n` $= |V|$ and `m` $= |E|$. Furthermore, we will assume that $V = \{0, \ldots, \texttt{n}-1\}$. Any other data that we would like to associate with the elements of V can be stored in an array of length `n`.

Some typical operations performed on graphs are:

- `addEdge(i, j)`: Add the edge `(i, j)` to E.
- `removeEdge(i, j)`: Remove the edge `(i, j)` from E.
- `hasEdge(i, j)`: Check if the edge `(i, j)` $\in E$
- `outEdges(i)`: Return a `List` of all integers `j` such that `(i, j)` $\in E$
- `inEdges(i)`: Return a `List` of all integers `j` such that `(j, i)` $\in E$

Note that these operations are not terribly difficult to implement efficiently. For example, the first three operations can be implemented directly by using a `USet`, so they can be implemented in constant expected time using the hash tables discussed in Capítulo 5. The last two operations can be implemented in constant time by storing, for each vertex, a list of its adjacent vertices.

However, different applications of graphs have different performance requirements for these operations and, ideally, we can use the simplest implementation that satisfies all the application's requirements. For this reason, we discuss two broad categories of graph representations.

12.1 AdjacencyMatrix: Representing a Graph by a Matrix

An *adjacency matrix* is a way of representing an n vertex graph $G = (V, E)$ by an $n \times n$ matrix, a, whose entries are boolean values.

AdjacencyMatrix

```
int n;
bool **a;
```

The matrix entry a[i][j] is defined as

$$\texttt{a[i][j]} = \begin{cases} \texttt{true} & \text{if } (\texttt{i}, \texttt{j}) \in E \\ \texttt{false} & \text{otherwise} \end{cases}$$

The adjacency matrix for the graph in Figura 12.1 is shown in Figura 12.2.

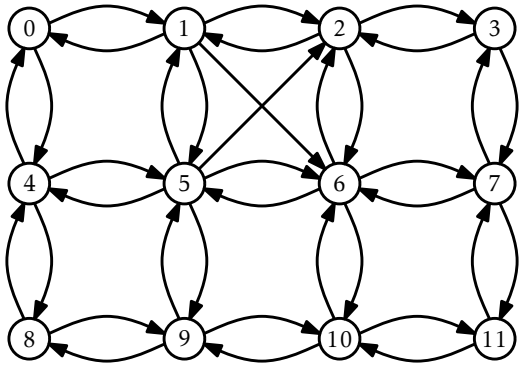
In this representation, the operations addEdge(i, j), removeEdge(i, j), and hasEdge(i, j) just involve setting or reading the matrix entry a[i][j]:

AdjacencyMatrix

```
void addEdge(int i, int j) {
  a[i][j] = true;
}
void removeEdge(int i, int j) {
  a[i][j] = false;
}
bool hasEdge(int i, int j) {
  return a[i][j];
}
```

These operations clearly take constant time per operation.

Where the adjacency matrix performs poorly is with the outEdges(i) and inEdges(i) operations. To implement these, we must scan all n entries in the corresponding row or column of a and gather up all the indices, j, where a[i][j], respectively a[j][i], is true.



	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: A graph and its adjacency matrix.

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);
}
```

These operations clearly take $O(n)$ time per operation.

Another drawback of the adjacency matrix representation is that it is large. It stores an $n \times n$ boolean matrix, so it requires at least n^2 bits of memory. The implementation here uses a matrix of `bool` values so it actually uses on the order of n^2 bytes of memory. A more careful implementation, which packs `w` boolean values into each word of memory, could reduce this space usage to $O(n^2/w)$ words of memory.

Teorema 12.1. *The `AdjacencyMatrix` data structure implements the `Graph` interface. An `AdjacencyMatrix` supports the operations*

- `addEdge(i,j)`, `removeEdge(i,j)`, *and* `hasEdge(i,j)` *in constant time per operation; and*
- `inEdges(i)`, *and* `outEdges(i)` *in* $O(n)$ *time per operation.*

The space used by an `AdjacencyMatrix` is $O(n^2)$.

Despite its high memory requirements and poor performance of the `inEdges(i)` and `outEdges(i)` operations, an `AdjacencyMatrix` can still be useful for some applications. In particular, when the graph G is *dense*, i.e., it has close to n^2 edges, then a memory usage of n^2 may be acceptable.

The `AdjacencyMatrix` data structure is also commonly used because algebraic operations on the matrix `a` can be used to efficiently compute properties of the graph G. This is a topic for a course on algorithms, but we point out one such property here: If we treat the entries of `a` as integers (1 for `true` and 0 for `false`) and multiply `a` by itself using matrix multiplication then we get the matrix a^2. Recall, from the definition of

matrix multiplication, that

$$a^2[i][j] = \sum_{k=0}^{n-1} a[i][k] \cdot a[k][j] \ .$$

Interpreting this sum in terms of the graph G, this formula counts the number of vertices, `k`, such that G contains both edges (`i`,`k`) and (`k`, `j`). That is, it counts the number of paths from `i` to `j` (through intermediate vertices, `k`) whose length is exactly two. This observation is the foundation of an algorithm that computes the shortest paths between all pairs of vertices in G using only $O(\log n)$ matrix multiplications.

12.2 AdjacencyLists: A Graph as a Collection of Lists

Adjacency list representations of graphs take a more vertex-centric approach. There are many possible implementations of adjacency lists. In this section, we present a simple one. At the end of the section, we discuss different possibilities. In an adjacency list representation, the graph $G = (V, E)$ is represented as an array, `adj`, of lists. The list `adj[i]` contains a list of all the vertices adjacent to vertex `i`. That is, it contains every index `j` such that (`i`, `j`) $\in E$.

AdjacencyLists

```
int n;
List *adj;
```

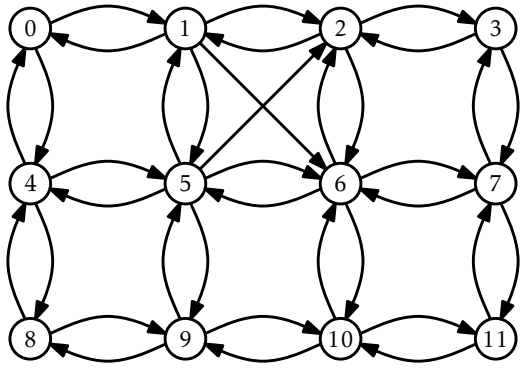
(An example is shown in Figura 12.3.) In this particular implementation, we represent each list in `adj` as a subclass of `ArrayStack`, because we would like constant time access by position. Other options are also possible. Specifically, we could have implemented `adj` as a `DLList`.

The `addEdge(i, j)` operation just appends the value `j` to the list `adj[i]`:

AdjacencyLists

```
void addEdge(int i, int j) {
  adj[i].add(j);
}
```

This takes constant time.



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: A graph and its adjacency lists

The `removeEdge(i, j)` operation searches through the list `adj[i]` until it finds `j` and then removes it:

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;
    }
  }
}
```

This takes $O(\deg(i))$ time, where $\deg(i)$ (the *degree* of `i`) counts the number of edges in E that have `i` as their source.

The `hasEdge(i, j)` operation is similar; it searches through the list `adj[i]` until it finds `j` (and returns true), or reaches the end of the list (and returns false):

AdjacencyLists
```
bool hasEdge(int i, int j) {
  return adj[i].contains(j);
}
```

This also takes $O(\deg(i))$ time.

The `outEdges(i)` operation is very simple; it copies the values in `adj[i]` into the output list:

AdjacencyLists
```
void outEdges(int i, ListT &edges) {
  for (int k = 0; k < adj[i].size(); k++)
    edges.add(adj[i].get(k));
}
```

This clearly takes $O(\deg(i))$ time.

The `inEdges(i)` operation is much more work. It scans over every vertex j checking if the edge `(i, j)` exists and, if so, adding `j` to the output list:

AdjacencyLists
```
void inEdges(int i, ListT &edges) {
  for (int j = 0; j < n; j++)
    if (adj[j].contains(i))  edges.add(j);
}
```

This operation is very slow. It scans the adjacency list of every vertex, so it takes $O(n+m)$ time.

The following theorem summarizes the performance of the above data structure:

Teorema 12.2. *The* `AdjacencyLists` *data structure implements the* `Graph` *interface. An* `AdjacencyLists` *supports the operations*

- `addEdge(i, j)` *in constant time per operation;*
- `removeEdge(i, j)` *and* `hasEdge(i, j)` *in* $O(\deg(i))$ *time per operation;*
- `outEdges(i)` *in* $O(\deg(i))$ *time per operation; and*
- `inEdges(i)` *in* $O(n+m)$ *time per operation.*

The space used by a `AdjacencyLists` *is* $O(n+m)$.

As alluded to earlier, there are many different choices to be made when implementing a graph as an adjacency list. Some questions that come up include:

- What type of collection should be used to store each element of `adj`? One could use an array-based list, a linked-list, or even a hashtable.
- Should there be a second adjacency list, `inadj`, that stores, for each `i`, the list of vertices, `j`, such that $(j,i) \in E$? This can greatly reduce the running-time of the `inEdges(i)` operation, but requires slightly more work when adding or removing edges.
- Should the entry for the edge `(i, j)` in `adj[i]` be linked by a reference to the corresponding entry in `inadj[j]`?
- Should edges be first-class objects with their own associated data? In this way, `adj` would contain lists of edges rather than lists of vertices (integers).

Most of these questions come down to a tradeoff between complexity (and space) of implementation and performance features of the implementation.

12.3 Graph Traversal

In this section we present two algorithms for exploring a graph, starting at one of its vertices, `i`, and finding all vertices that are reachable from `i`. Both of these algorithms are best suited to graphs represented using an adjacency list representation. Therefore, when analyzing these algorithms we will assume that the underlying representation is an `AdjacencyLists`.

12.3.1 Breadth-First Search

The *bread-first-search* algorithm starts at a vertex `i` and visits, first the neighbours of `i`, then the neighbours of the neighbours of `i`, then the neighbours of the neighbours of the neighbours of `i`, and so on.

This algorithm is a generalization of the breadth-first traversal algorithm for binary trees (Seção 6.1.2), and is very similar; it uses a queue, `q`, that initially contains only `i`. It then repeatedly extracts an element from `q` and adds its neighbours to `q`, provided that these neighbours have never been in `q` before. The only major difference between the breadth-first-search algorithm for graphs and the one for trees is that the algorithm for graphs has to ensure that it does not add the same vertex to `q` more than once. It does this by using an auxiliary boolean array, `seen`, that tracks which vertices have already been discovered.

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++) {
      int j = edges.get(k);
      if (!seen[j]) {
        q.add(j);
        seen[j] = true;
```

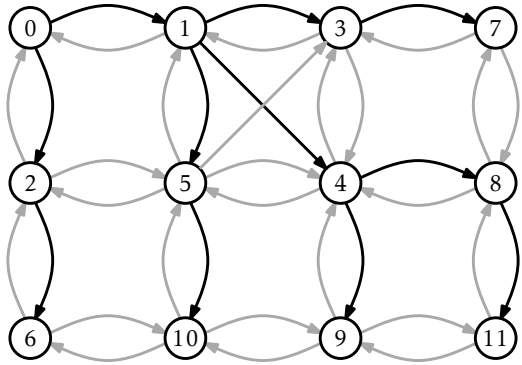


Figura 12.4: An example of bread-first-search starting at node 0. Nodes are labelled with the order in which they are added to q. Edges that result in nodes being added to q are drawn in black, other edges are drawn in grey.

```
      }
    }
  }
  delete[] seen;
}
```

An example of running bfs(g,0) on the graph from Figura 12.1 is shown in Figura 12.4. Different executions are possible, depending on the ordering of the adjacency lists; Figura 12.4 uses the adjacency lists in Figura 12.3.

Analyzing the running-time of the bfs(g, i) routine is fairly straightforward. The use of the seen array ensures that no vertex is added to q more than once. Adding (and later removing) each vertex from q takes constant time per vertex for a total of $O(n)$ time. Since each vertex is processed by the inner loop at most once, each adjacency list is processed at most once, so each edge of G is processed at most once. This processing, which is done in the inner loop takes constant time per iteration, for a total of $O(m)$ time. Therefore, the entire algorithm runs in $O(n+m)$ time.

The following theorem summarizes the performance of the bfs(g, r) algorithm.

Teorema 12.3. *When given as input a Graph, g, that is implemented using*

the `AdjacencyLists` *data structure, the* `bfs(g,r)` *algorithm runs in* $O(n+m)$ *time.*

A breadth-first traversal has some very special properties. Calling `bfs(g,r)` will eventually enqueue (and eventually dequeue) every vertex `j` such that there is a directed path from `r` to `j`. Moreover, the vertices at distance 0 from `r` (`r` itself) will enter `q` before the vertices at distance 1, which will enter `q` before the vertices at distance 2, and so on. Thus, the `bfs(g,r)` method visits vertices in increasing order of distance from `r` and vertices that cannot be reached from `r` are never visited at all.

A particularly useful application of the breadth-first-search algorithm is, therefore, in computing shortest paths. To compute the shortest path from `r` to every other vertex, we use a variant of `bfs(g,r)` that uses an auxilliary array, `p`, of length `n`. When a new vertex `j` is added to `q`, we set `p[j] = i`. In this way, `p[j]` becomes the second last node on a shortest path from `r` to `j`. Repeating this, by taking `p[p[j]]`, `p[p[p[j]]]`, and so on we can reconstruct the (reversal of) a shortest path from `r` to `j`.

12.3.2 Depth-First Search

The *depth-first-search* algorithm is similar to the standard algorithm for traversing binary trees; it first fully explores one subtree before returning to the current node and then exploring the other subtree. Another way to think of depth-first-search is by saying that it is similar to breadth-first search except that it uses a stack instead of a queue.

During the execution of the depth-first-search algorithm, each vertex, `i`, is assigned a colour, `c[i]`: `white` if we have never seen the vertex before, `grey` if we are currently visiting that vertex, and `black` if we are done visiting that vertex. The easiest way to think of depth-first-search is as a recursive algorithm. It starts by visiting `r`. When visiting a vertex `i`, we first mark `i` as `grey`. Next, we scan `i`'s adjacency list and recursively visit any white vertex we find in this list. Finally, we are done processing `i`, so we colour `i` black and return.

Algorithms

```
void dfs(Graph &g, int i, char *c) {
  c[i] = grey;  // currently visiting i
  ArrayStack<int> edges;
```

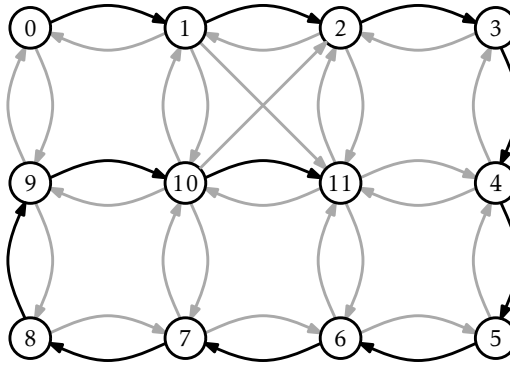


Figura 12.5: An example of depth-first-search starting at node 0. Nodes are labelled with the order in which they are processed. Edges that result in a recursive call are drawn in black, other edges are drawn in grey.

```
  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;
}
```

An example of the execution of this algorithm is shown in Figura 12.5.

Although depth-first-search may best be thought of as a recursive algorithm, recursion is not the best way to implement it. Indeed, the code given above will fail for many large graphs by causing a stack overflow. An alternative implementation is to replace the recursion stack with an explicit stack, s. The following implementation does just that:

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();
    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;
}
```

In the preceding code, when the next vertex, `i`, is processed, `i` is coloured `grey` and then replaced, on the stack, with its adjacent vertices. During the next iteration, one of these vertices will be visited.

Not surprisingly, the running times of `dfs(g,r)` and `dfs2(g,r)` are the same as that of `bfs(g,r)`:

Teorema 12.4. *When given as input a* `Graph`, `g`, *that is implemented using the* `AdjacencyLists` *data structure, the* `dfs(g,r)` *and* `dfs2(g,r)` *algorithms each run in* $O(n+m)$ *time.*

As with the breadth-first-search algorithm, there is an underlying tree associated with each execution of depth-first-search. When a node $i \neq r$ goes from `white` to `grey`, this is because `dfs(g,i,c)` was called recursively while processing some node i'. (In the case of `dfs2(g,r)` algorithm, `i` is one of the nodes that replaced i' on the stack.) If we think of i' as the parent of `i`, then we obtain a tree rooted at `r`. In Figura 12.5, this tree is a path from vertex 0 to vertex 11.

An important property of the depth-first-search algorithm is the following: Suppose that when node `i` is coloured `grey`, there exists a path from `i` to some other node `j` that uses only white vertices. Then `j` will be coloured first `grey` then `black` before `i` is coloured `black`. (This can be

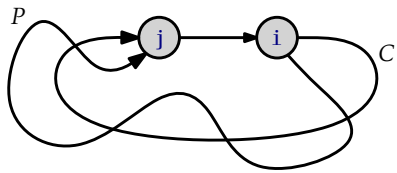


Figura 12.6: The depth-first-search algorithm can be used to detect cycles in G. The node `j` is coloured `grey` while `i` is still `grey`. This implies that there is a path, P, from `i` to `j` in the depth-first-search tree, and the edge `(j,i)` implies that P is also a cycle.

proven by contradiction, by considering any path P from `i` to `j`.)

One application of this property is the detection of cycles. Refer to Figura 12.6. Consider some cycle, C, that can be reached from `r`. Let `i` be the first node of C that is coloured `grey`, and let `j` be the node that precedes `i` on the cycle C. Then, by the above property, `j` will be coloured `grey` and the edge `(j,i)` will be considered by the algorithm while `i` is still `grey`. Thus, the algorithm can conclude that there is a path, P, from `i` to `j` in the depth-first-search tree and the edge `(j,i)` exists. Therefore, P is also a cycle.

12.4 Discussion and Exercises

The running times of the depth-first-search and breadth-first-search algorithms are somewhat overstated by the Theorems 12.3 and 12.4. Define $\mathtt{n}_\mathtt{r}$ as the number of vertices, `i`, of G, for which there exists a path from `r` to `i`. Define $\mathtt{m}_\mathtt{r}$ as the number of edges that have these vertices as their sources. Then the following theorem is a more precise statement of the running times of the breadth-first-search and depth-first-search algorithms. (This more refined statement of the running time is useful in some of the applications of these algorithms outlined in the exercises.)

Teorema 12.5. *When given as input a* `Graph`, `g`, *that is implemented using the* `AdjacencyLists` *data structure, the* `bfs(g,r)`, `dfs(g,r)` *and* `dfs2(g,r)` *algorithms each run in* $O(\mathtt{n}_\mathtt{r} + \mathtt{m}_\mathtt{r})$ *time.*

Breadth-first search seems to have been discovered independently by

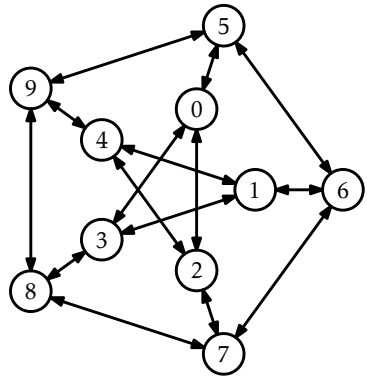


Figura 12.7: An example graph.

Moore [51] and Lee [48] in the contexts of maze exploration and circuit routing, respectively.

Adjacency-list representations of graphs were presented by Hopcroft and Tarjan [39] as an alternative to the (then more common) adjacency-matrix representation. This representation, as well as depth-first-search, played a major part in the celebrated Hopcroft-Tarjan planarity testing algorithm that can determine, in $O(\mathtt{n})$ time, if a graph can be drawn, in the plane, and in such a way that no pair of edges cross each other [40].

In the following exercises, an undirected graph is one in which, for every $\mathtt{i}$ and $\mathtt{j}$, the edge $(\mathtt{i},\mathtt{j})$ is present if and only if the edge $(\mathtt{j},\mathtt{i})$ is present.

Exercício 12.1. Draw an adjacency list representation and an adjacency matrix representation of the graph in Figura 12.7.

Exercício 12.2. The *incidence matrix* representation of a graph, G, is an $\mathtt{n}\times\mathtt{m}$ matrix, A, where

$$A_{i,j} = \begin{cases} -1 & \text{if vertex } i \text{ the source of edge } j \\ +1 & \text{if vertex } i \text{ the target of edge } j \\ 0 & \text{otherwise.} \end{cases}$$

1. Draw the incident matrix representation of the graph in Figura 12.7.

2. Design, analyze and implement an incidence matrix representation of a graph. Be sure to analyze the space, the cost of `addEdge(i, j)`, `removeEdge(i, j)`, `hasEdge(i, j)`, `inEdges(i)`, and `outEdges(i)`.

Exercício 12.3. Illustrate an execution of the `bfs(G, 0)` and `dfs(G, 0)` on the graph, `G`, in Figura 12.7.

Exercício 12.4. Let G be an undirected graph. We say G is *connected* if, for every pair of vertices `i` and `j` in G, there is a path from `i` to `j` (since G is undirected, there is also a path from `j` to `i`). Show how to test if G is connected in $O(n+m)$ time.

Exercício 12.5. Let G be an undirected graph. A *connected-component labelling* of G partitions the vertices of G into maximal sets, each of which forms a connected subgraph. Show how to compute a connected component labelling of G in $O(n+m)$ time.

Exercício 12.6. Let G be an undirected graph. A *spanning forest* of G is a collection of trees, one per component, whose edges are edges of G and whose vertices contain all vertices of G. Show how to compute a spanning forest of of G in $O(n+m)$ time.

Exercício 12.7. We say that a graph G is *strongly-connected* if, for every pair of vertices `i` and `j` in G, there is a path from `i` to `j`. Show how to test if G is strongly-connected in $O(n+m)$ time.

Exercício 12.8. Given a graph $G = (V, E)$ and some special vertex $r \in V$, show how to compute the length of the shortest path from `r` to `i` for every vertex $i \in V$.

Exercício 12.9. Give a (simple) example where the `dfs(g, r)` code visits the nodes of a graph in an order that is different from that of the `dfs2(g, r)` code. Write a version of `dfs2(g, r)` that always visits nodes in exactly the same order as `dfs(g, r)`. (Hint: Just start tracing the execution of each algorithm on some graph where `r` is the source of more than 1 edge.)

Exercício 12.10. A *universal sink* in a graph G is a vertex that is the target of $n-1$ edges and the source of no edges.[1] Design and implement an

[1] A universal sink, `v`, is also sometimes called a *celebrity*: Everyone in the room recognizes `v`, but `v` doesn't recognize anyone else in the room.

algorithm that tests if a graph G, represented as an `AdjacencyMatrix`, has a universal sink. Your algorithm should run in $O(n)$ time.

Capítulo 13

Data Structures for Integers

In this chapter, we return to the problem of implementing an SSet. The difference now is that we assume the elements stored in the SSet are w-bit integers. That is, we want to implement add(x), remove(x), and find(x) where $x \in \{0,\ldots,2^w-1\}$. It is not too hard to think of plenty of applications where the data—or at least the key that we use for sorting the data—is an integer.

We will discuss three data structures, each building on the ideas of the previous. The first structure, the BinaryTrie performs all three SSet operations in $O(w)$ time. This is not very impressive, since any subset of $\{0,\ldots,2^w-1\}$ has size $n \le 2^w$, so that $\log n \le w$. All the other SSet implementations discussed in this book perform all operations in $O(\log n)$ time so they are all at least as fast as a BinaryTrie.

The second structure, the XFastTrie, speeds up the search in a BinaryTrie by using hashing. With this speedup, the find(x) operation runs in $O(\log w)$ time. However, add(x) and remove(x) operations in an XFastTrie still take $O(w)$ time and the space used by an XFastTrie is $O(n \cdot w)$.

The third data structure, the YFastTrie, uses an XFastTrie to store only a sample of roughly one out of every w elements and stores the remaining elements in a standard SSet structure. This trick reduces the running time of add(x) and remove(x) to $O(\log w)$ and decreases the space to $O(n)$.

The implementations used as examples in this chapter can store any type of data, as long as an integer can be associated with it. In the code

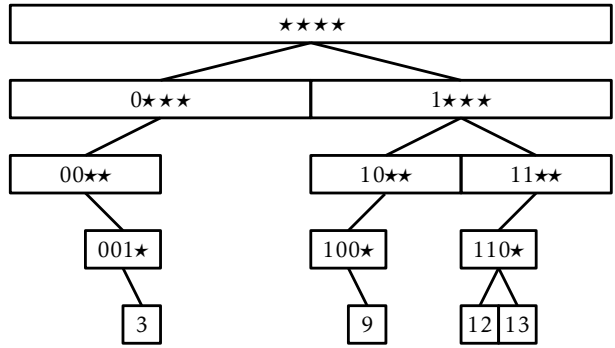


Figura 13.1: The integers stored in a binary trie are encoded as root-to-leaf paths.

samples, the variable `ix` is always the integer value associated with `x`, and the method `intValue(x)` converts `x` to its associated integer. In the text, however, we will simply treat `x` as if it is an integer.

13.1 BinaryTrie: A digital search tree

A `BinaryTrie` encodes a set of `w` bit integers in a binary tree. All leaves in the tree have depth `w` and each integer is encoded as a root-to-leaf path. The path for the integer `x` turns left at level `i` if the `i`th most significant bit of `x` is a 0 and turns right if it is a 1. Figura 13.1 shows an example for the case `w` = 4, in which the trie stores the integers 3(0011), 9(1001), 12(1100), and 13(1101).

Because the search path for a value `x` depends on the bits of `x`, it will be helpful to name the children of a node, `u`, `u.child`[0] (`left`) and `u.child`[1] (`right`). These child pointers will actually serve double-duty. Since the leaves in a binary trie have no children, the pointers are used to string the leaves together into a doubly-linked list. For a leaf in the binary trie `u.child`[0] (`prev`) is the node that comes before `u` in the list and `u.child`[1] (`next`) is the node that follows `u` in the list. A special node, `dummy`, is used both before the first node and after the last node in the list (see Seção 3.2). In the code samples, `u.child`[0], `u.left`, and `u.prev` refer

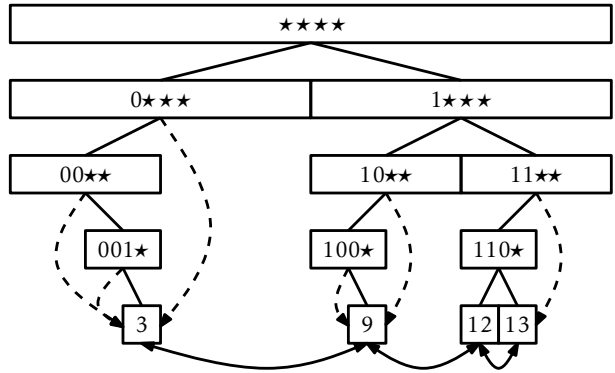


Figura 13.2: A BinaryTrie with jump pointers shown as curved dashed edges.

to the same field in the node u, as do u.child[1], u.right, and u.next.

Each node, u, also contains an additional pointer u.jump. If u's left child is missing, then u.jump points to the smallest leaf in u's subtree. If u's right child is missing, then u.jump points to the largest leaf in u's subtree. An example of a BinaryTrie, showing jump pointers and the doubly-linked list at the leaves, is shown in Figura 13.2.

The find(x) operation in a BinaryTrie is fairly straightforward. We try to follow the search path for x in the trie. If we reach a leaf, then we have found x. If we reach a node u where we cannot proceed (because u is missing a child), then we follow u.jump, which takes us either to the smallest leaf larger than x or the largest leaf smaller than x. Which of these two cases occurs depends on whether u is missing its left or right child, respectively. In the former case (u is missing its left child), we have found the node we want. In the latter case (u is missing its right child), we can use the linked list to reach the node we want. Each of these cases is illustrated in Figura 13.3.

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;
```

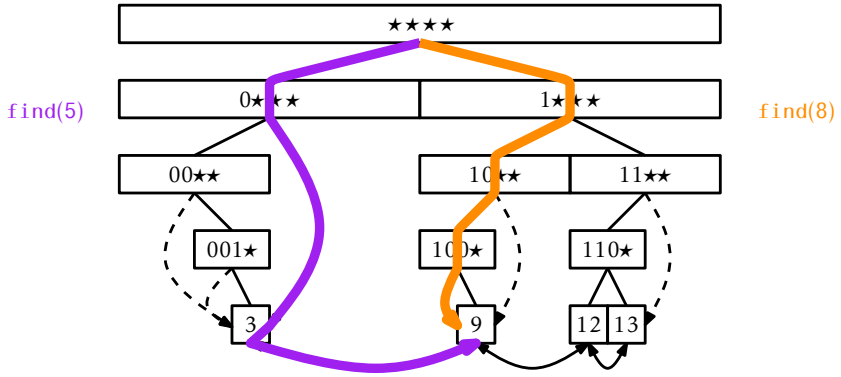


Figura 13.3: The paths followed by `find(5)` and `find(8)`.

```
    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;
}
```

The running-time of the `find(x)` method is dominated by the time it takes to follow a root-to-leaf path, so it runs in $O(w)$ time.

The `add(x)` operation in a `BinaryTrie` is also fairly straightforward, but has a lot of work to do:

1. It follows the search path for `x` until reaching a node `u` where it can no longer proceed.

2. It creates the remainder of the search path from `u` to a leaf that contains `x`.

3. It adds the node, `u'`, containing `x` to the linked list of leaves (it has access to the predecessor, `pred`, of `u'` in the linked list from the `jump` pointer of the last node, `u`, encountered during step 1.)

4. It walks back up the search path for `x` adjusting `jump` pointers at the nodes whose `jump` pointer should now point to `x`.

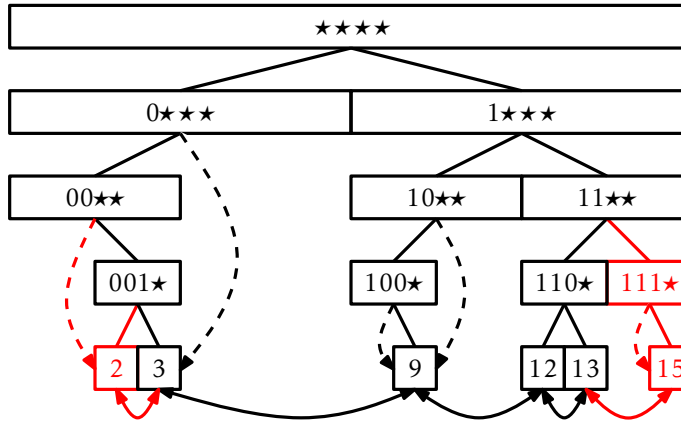


Figura 13.4: Adding the values 2 and 15 to the `BinaryTrie` in Figura 13.2.

An addition is illustrated in 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;
}
```

This method performs one walk down the search path for `x` and one walk back up. Each step of these walks takes constant time, so the `add(x)` method runs in $O(w)$ time.

The `remove(x)` operation undoes the work of `add(x)`. Like `add(x)`, it has a lot of work to do:

1. It follows the search path for `x` until reaching the leaf, `u`, containing `x`.
2. It removes `u` from the doubly-linked list.
3. It deletes `u` and then walks back up the search path for `x` deleting nodes until reaching a node `v` that has a child that is not on the search path for `x`.
4. It walks upwards from `v` to the root updating any `jump` pointers that point to `u`.

A removal is illustrated in Figura 13.5.

BinaryTrie

```
bool remove(T x) {
  // 1 - find leaf, u, containing x
  int i = 0, c;
```

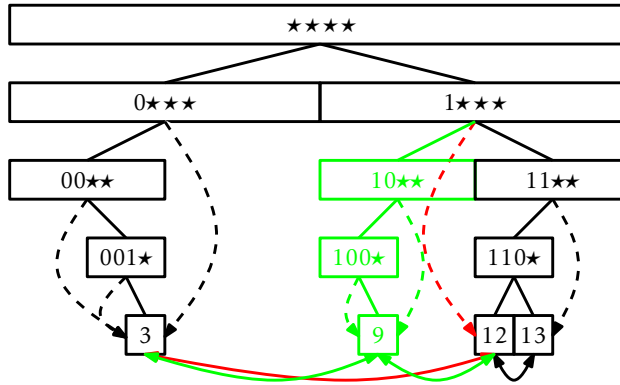


Figura 13.5: Removing the value 9 from the `BinaryTrie` in Figura 13.2.

```
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. *A* `BinaryTrie` *implements the* `SSet` *interface for* `w`*-bit integers. A* `BinaryTrie` *supports the operations* `add(x)`, `remove(x)`, *and* `find(x)` *in* $O(w)$ *time per operation. The space used by a* `BinaryTrie` *that stores* `n` *values is* $O(n \cdot w)$.

13.2 XFastTrie: Searching in Doubly-Logarithmic Time

The performance of the `BinaryTrie` structure is not very impressive. The number of elements, `n`, stored in the structure is at most 2^w, so $\log n \leq w$. In other words, any of the comparison-based `SSet` structures described in other parts of this book are at least as efficient as a `BinaryTrie`, and are not restricted to only storing integers.

Next we describe the `XFastTrie`, which is just a `BinaryTrie` with `w`$+1$ hash tables—one for each level of the trie. These hash tables are used to speed up the `find(x)` operation to $O(\log w)$ time. Recall that the `find(x)` operation in a `BinaryTrie` is almost complete once we reach a node, `u`, where the search path for `x` would like to proceed to `u.right` (or `u.left`) but `u` has no right (respectively, left) child. At this point, the search uses `u.jump` to jump to a leaf, `v`, of the `BinaryTrie` and either return `v` or its successor in the linked list of leaves. An `XFastTrie` speeds up the search process by using binary search on the levels of the trie to locate the node `u`.

To use binary search, we need a way to determine if the node `u` we are looking for is above a particular level, `i`, of if `u` is at or below level `i`. This information is given by the highest-order `i` bits in the binary representation of `x`; these bits determine the search path that `x` takes from the root to level `i`. For an example, refer to Figura 13.6; in this figure the

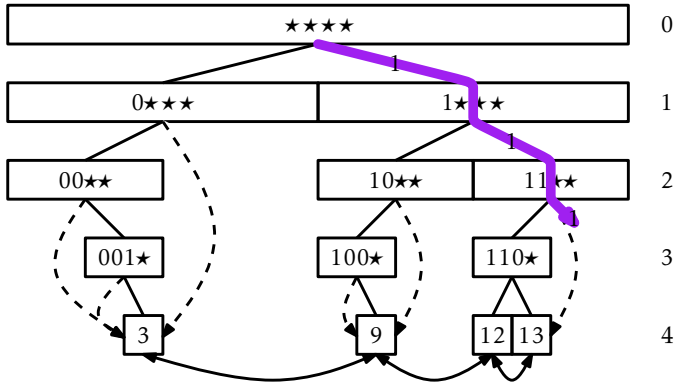


Figura 13.6: Since there is no node labelled 111★, the search path for 14 (1110) ends at the node labelled 11★★ .

last node, `u`, on search path for 14 (whose binary representation is 1110) is the node labelled 11★★ at level 2 because there is no node labelled 111★ at level 3. Thus, we can label each node at level `i` with an `i`-bit integer. Then, the node `u` we are searching for would be at or below level `i` if and only if there is a node at level `i` whose label matches the highest-order `i` bits of `x`.

In an `XFastTrie`, we store, for each $i \in \{0,\ldots,w\}$, all the nodes at level `i` in a `USet`, `t[i]`, that is implemented as a hash table (Capítulo 5). Using this `USet` allows us to check in constant expected time if there is a node at level `i` whose label matches the highest-order `i` bits of `x`. In fact, we can even find this node using `t[i].find(x >> (w - i))`

The hash tables `t[0]`,...,`t[w]` allow us to use binary search to find `u`. Initially, we know that `u` is at some level `i` with $0 \le i < w+1$. We therefore initialize $l = 0$ and $h = w+1$ and repeatedly look at the hash table `t[i]`, where $i = \lfloor (l+h)/2 \rfloor$. If `t[i]` contains a node whose label matches `x`'s highest-order `i` bits then we set $l = i$ (`u` is at or below level `i`); otherwise we set $h = i$ (`u` is above level `i`). This process terminates when $h - l \le 1$, in which case we determine that `u` is at level `l`. We then complete the `find(x)` operation using `u.jump` and the doubly-linked list of leaves.

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;
}
```

Each iteration of the `while` loop in the above method decreases $h - l$ by roughly a factor of two, so this loop finds `u` after $O(\log w)$ iterations. Each iteration performs a constant amount of work and one `find(x)` operation in a `USet`, which takes a constant expected amount of time. The remaining work takes only constant time, so the `find(x)` method in an `XFastTrie` takes only $O(\log w)$ expected time.

The `add(x)` and `remove(x)` methods for an `XFastTrie` are almost identical to the same methods in a `BinaryTrie`. The only modifications are for managing the hash tables `t[0]`,...,`t[w]`. During the `add(x)` operation, when a new node is created at level `i`, this node is added to `t[i]`. During a `remove(x)` operation, when a node is removed form level `i`, this node is removed from `t[i]`. Since adding and removing from a hash table take constant expected time, this does not increase the running times of `add(x)` and `remove(x)` by more than a constant factor. We omit a code listing for `add(x)` and `remove(x)` since the code is almost identical to the (long) code listing already provided for the same methods in a `BinaryTrie`.

The following theorem summarizes the performance of an `XFastTrie`:

Teorema 13.2. *An `XFastTrie` implements the `SSet` interface for `w`-bit integers. An `XFastTrie` supports the operations*

- `add(x)` *and* `remove(x)` *in* $O(w)$ *expected time per operation and*
- `find(x)` *in* $O(\log w)$ *expected time per operation.*

The space used by an `XFastTrie` *that stores* n *values is* $O(n \cdot w)$.

13.3 YFastTrie: A Doubly-Logarithmic Time SSet

The `XFastTrie` is a vast—even exponential—improvement over the `BinaryTrie` in terms of query time, but the `add(x)` and `remove(x)` operations are still not terribly fast. Furthermore, the space usage, $O(n \cdot w)$, is higher than the other SSet implementations described in this book, which all use $O(n)$ space. These two problems are related; if n `add(x)` operations build a structure of size $n \cdot w$, then the `add(x)` operation requires at least on the order of w time (and space) per operation.

The `YFastTrie`, discussed next, simultaneously improves the space and speed of `XFastTries`. A `YFastTrie` uses an `XFastTrie`, `xft`, but only stores $O(n/w)$ values in `xft`. In this way, the total space used by `xft` is only $O(n)$. Furthermore, only one out of every w `add(x)` or `remove(x)` operations in the `YFastTrie` results in an `add(x)` or `remove(x)` operation in `xft`. By doing this, the average cost incurred by calls to `xft`'s `add(x)` and `remove(x)` operations is only constant.

The obvious question becomes: If `xft` only stores n/w elements, where do the remaining $n(1 - 1/w)$ elements go? These elements move into *secondary structures*, in this case an extended version of treaps (Seção 7.2). There are roughly n/w of these secondary structures so, on average, each of them stores $O(w)$ items. Treaps support logarithmic time SSet operations, so the operations on these treaps will run in $O(\log w)$ time, as required.

More concretely, a `YFastTrie` contains an `XFastTrie`, `xft`, that contains a random sample of the data, where each element appears in the sample independently with probability $1/w$. For convenience, the value $2^w - 1$, is always contained in `xft`. Let $x_0 < x_1 < \cdots < x_{k-1}$ denote the elements stored in `xft`. Associated with each element, x_i, is a treap, t_i, that stores all values in the range $x_{i-1} + 1, \ldots, x_i$. This is illustrated in Figura 13.7.

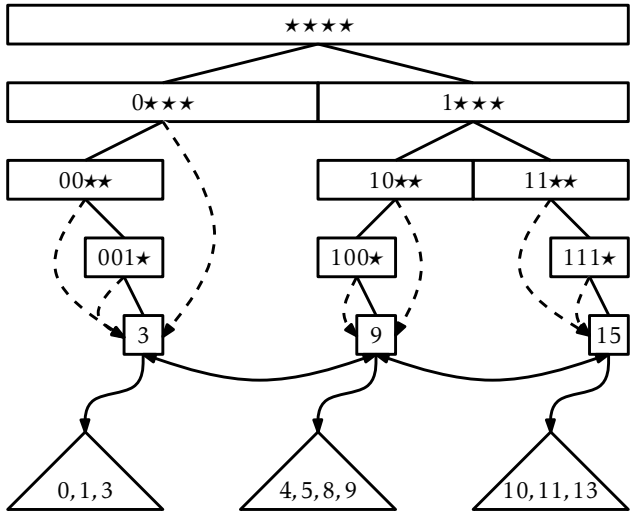


Figura 13.7: A YFastTrie containing the values 0, 1, 3, 4, 6, 8, 9, 10, 11, and 13.

The `find(x)` operation in a `YFastTrie` is fairly easy. We search for `x` in `xft` and find some value x_i associated with the treap t_i. We then use the treap `find(x)` method on t_i to answer the query. The entire method is a one-liner:

YFastTrie
```
T find(T x) {
  return xft.find(YPair<T>(intValue(x))).t->find(x);
}
```

The first `find(x)` operation (on `xft`) takes $O(\log w)$ time. The second `find(x)` operation (on a treap) takes $O(\log r)$ time, where r is the size of the treap. Later in this section, we will show that the expected size of the treap is $O(w)$ so that this operation takes $O(\log w)$ time.[1]

Adding an element to a `YFastTrie` is also fairly simple—most of the time. The `add(x)` method calls `xft.find(x)` to locate the treap, `t`, into which `x` should be inserted. It then calls `t.add(x)` to add `x` to `t`. At this point, it tosses a biased coin that comes up as heads with probability $1/w$ and as tails with probability $1 - 1/w$. If this coin comes up heads, then `x`

[1] This is an application of *Jensen's Inequality*: If $\mathrm{E}[r] = w$, then $\mathrm{E}[\log r] \leq \log w$.

will be added to xft.

This is where things get a little more complicated. When x is added to xft, the treap t needs to be split into two treaps, t1 and t′. The treap t1 contains all the values less than or equal to x; t′ is the original treap, t, with the elements of t1 removed. Once this is done, we add the pair (x, t1) to xft. Figura 13.8 shows an example.

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;
  }
  return false;
  return true;
}
```

Adding x to t takes $O(\log w)$ time. Exercício 7.12 shows that splitting t into t1 and t′ can also be done in $O(\log w)$ expected time. Adding the pair (x,t1) to xft takes $O(w)$ time, but only happens with probability $1/w$. Therefore, the expected running time of the add(x) operation is

$$O(\log w) + \frac{1}{w}O(w) = O(\log w) \ .$$

The remove(x) method undoes the work performed by add(x). We use xft to find the leaf, u, in xft that contains the answer to xft.find(x). From u, we get the treap, t, containing x and remove x from t. If x was also stored in xft (and x is not equal to $2^w - 1$) then we remove x from xft and add the elements from x's treap to the treap, t2, that is stored by u's successor in the linked list. This is illustrated in Figura 13.9.

YFastTrie

```
bool remove(T x) {
  unsigned ix = intValue(x);
  XFastTrieNode1<YPair<T> > *u = xft.findNode(ix);
```

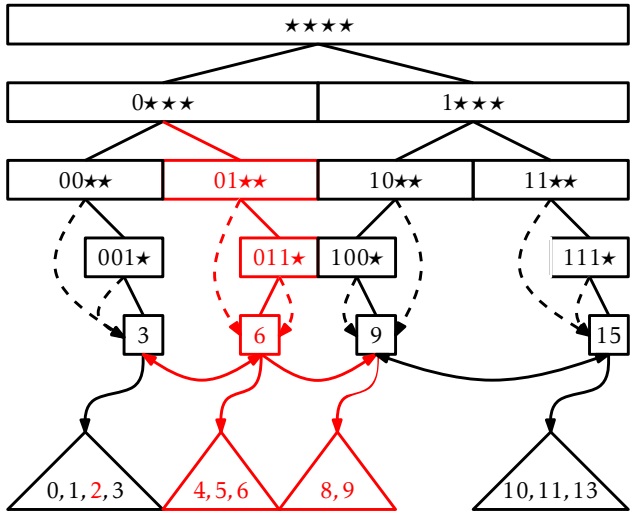


Figura 13.8: Adding the values 2 and 6 to a `YFastTrie`. The coin toss for 6 came up heads, so 6 was added to `xft` and the treap containing 4, 5, 6, 8, 9 was split.

```
  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;
}
```

Finding the node `u` in `xft` takes $O(\log w)$ expected time. Removing `x` from `t` takes $O(\log w)$ expected time. Again, Exercício 7.12 shows that merging all the elements of `t` into `t2` can be done in $O(\log w)$ time. If necessary, removing `x` from `xft` takes $O(w)$ time, but `x` is only contained in `xft` with probability $1/w$. Therefore, the expected time to remove an element from a `YFastTrie` is $O(\log w)$.

Earlier in the discussion, we delayed arguing about the sizes of treaps in this structure until later. Before finishing this chapter, we prove the result we need.

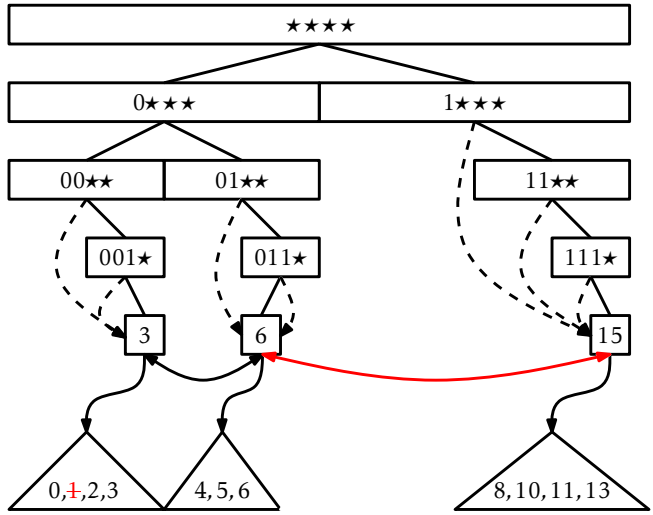


Figura 13.9: Removing the values 1 and 9 from a `YFastTrie` in Figura 13.8.

Lema 13.1. *Let* x *be an integer stored in a* `YFastTrie` *and let* n_x *denote the number of elements in the treap,* t*, that contains* x*. Then* $E[n_x] \leq 2w - 1$.

Demonstração. Refer to Figura 13.10. Let $x_1 < x_2 < \cdots < x_i = x < x_{i+1} < \cdots < x_n$ denote the elements stored in the `YFastTrie`. The treap t contains some elements greater than or equal to x. These are $x_i, x_{i+1}, \ldots, x_{i+j-1}$, where x_{i+j-1} is the only one of these elements in which the biased coin toss performed in the `add(x)` method turned up as heads. In other words, $E[j]$ is equal to the expected number of biased coin tosses required to obtain the first heads.[2] Each coin toss is independent and turns up as heads with probability $1/w$, so $E[j] \leq w$. (See Lema 4.2 for an analysis of this for the case $w = 2$.)

Similarly, the elements of t smaller than x are $x_{i-1}, \ldots, x_{i-k}$ where all these k coin tosses turn up as tails and the coin toss for x_{i-k-1} turns up as heads. Therefore, $E[k] \leq w - 1$, since this is the same coin tossing experiment considered in the preceding paragraph, but one in which the last

[2] This analysis ignores the fact that j never exceeds $n - i + 1$. However, this only decreases $E[j]$, so the upper bound still holds.

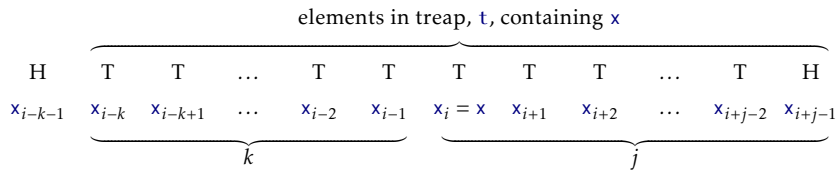


Figura 13.10: The number of elements in the treap `t` containing `x` is determined by two coin tossing experiments.

toss is not counted. In summary, $n_x = j + k$, so

$$\mathrm{E}[n_x] = \mathrm{E}[j + k] = \mathrm{E}[j] + \mathrm{E}[k] \leq 2w - 1 \ . \qquad \square$$

Lema 13.1 was the last piece in the proof of the following theorem, which summarizes the performance of the `YFastTrie`:

Teorema 13.3. *A `YFastTrie` implements the `SSet` interface for `w`-bit integers. A `YFastTrie` supports the operations* `add(x)`, `remove(x)`, *and* `find(x)` *in $O(\log w)$ expected time per operation. The space used by a `YFastTrie` that stores `n` values is $O(n + w)$.*

The `w` term in the space requirement comes from the fact that `xft` always stores the value $2^w - 1$. The implementation could be modified (at the expense of adding some extra cases to the code) so that it is unnecessary to store this value. In this case, the space requirement in the theorem becomes $O(n)$.

13.4 Discussion and Exercises

The first data structure to provide $O(\log w)$ time `add(x)`, `remove(x)`, and `find(x)` operations was proposed by van Emde Boas and has since become known as the *van Emde Boas* (or *stratified*) *tree* [71]. The original van Emde Boas structure had size 2^w, making it impractical for large integers.

The `XFastTrie` and `YFastTrie` data structures were discovered by Willard [74]. The `XFastTrie` structure is closely related to van Emde Boas trees; for instance, the hash tables in an `XFastTrie` replace arrays in a

van Emde Boas tree. That is, instead of storing the hash table $t[i]$, a van Emde Boas tree stores an array of length 2^i.

Another structure for storing integers is Fredman and Willard's fusion trees [31]. This structure can store n w-bit integers in $O(n)$ space so that the $\texttt{find}(x)$ operation runs in $O((\log n)/(\log w))$ time. By using a fusion tree when $\log w > \sqrt{\log n}$ and a YFastTrie when $\log w \le \sqrt{\log n}$, one obtains an $O(n)$ space data structure that can implement the $\texttt{find}(x)$ operation in $O(\sqrt{\log n})$ time. Recent lower-bound results of Pătraşcu and Thorup [56] show that these results are more or less optimal, at least for structures that use only $O(n)$ space.

Exercício 13.1. Design and implement a simplified version of a BinaryTrie that does not have a linked list or jump pointers, but for which $\texttt{find}(x)$

still runs in $O(w)$ time.

Exercício 13.2. Design and implement a simplified implementation of an XFastTrie that doesn't use a binary trie at all. Instead, your implementation should store everything in a doubly-linked list and $w+1$ hash tables.

Exercício 13.3. We can think of a BinaryTrie as a structure that stores bit strings of length w in such a way that each bitstring is represented as a root to leaf path. Extend this idea into an SSet implementation that stores variable-length strings and implements $\texttt{add}(s)$, $\texttt{remove}(s)$, and $\texttt{find}(s)$ in time proporitional to the length of s.
Hint: Each node in your data structure should store a hash table that is indexed by character values.

Exercício 13.4. For an integer $x \in \{0,\ldots 2^w - 1\}$, let $d(x)$ denote the difference between x and the value returned by $\texttt{find}(x)$ [if $\texttt{find}(x)$ returns null, then define $d(x)$ as 2^w]. For example, if $\texttt{find}(23)$ returns 43, then $d(23) = 20$.

1. Design and implement a modified version of the $\texttt{find}(x)$ operation in an XFastTrie that runs in $O(1 + \log d(x))$ expected time. Hint: The hash table $t[w]$ contains all the values, x, such that $d(x) = 0$, so that would be a good place to start.

2. Design and implement a modified version of the `find(x)` operation in an `XFastTrie` that runs in $O(1 + \log\log d(x))$ expected time.

Capítulo 14

External Memory Searching

Throughout this book, we have been using the w-bit word-RAM model of computation defined in Seção 1.4. An implicit assumption of this model is that our computer has a large enough random access memory to store all of the data in the data structure. In some situations, this assumption is not valid. There exist collections of data so large that no computer has enough memory to store them. In such cases, the application must resort to storing the data on some external storage medium such as a hard disk, a solid state disk, or even a network file server (which has its own external storage).

Accessing an item from external storage is extremely slow. The hard disk attached to the computer on which this book was written has an average access time of 19ms and the solid state drive attached to the computer has an average access time of 0.3ms. In contrast, the random access memory in the computer has an average access time of less than 0.000113ms. Accessing RAM is more than 2 500 times faster than accessing the solid state drive and more than 160 000 times faster than accessing the hard drive.

These speeds are fairly typical; accessing a random byte from RAM is thousands of times faster than accessing a random byte from a hard disk or solid-state drive. Access time, however, does not tell the whole story. When we access a byte from a hard disk or solid state disk, an entire *block* of the disk is read. Each of the drives attached to the computer has a block size of 4 096; each time we read one byte, the drive gives us a block containing 4 096 bytes. If we organize our data structure carefully, this

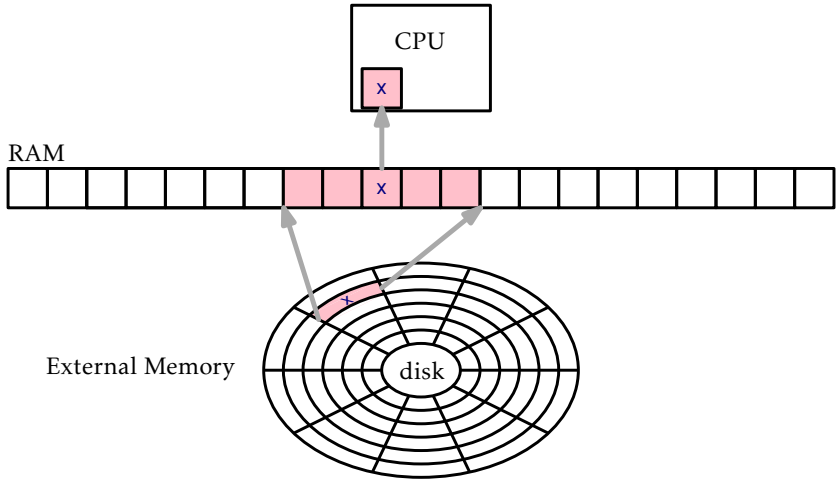


Figura 14.1: In the external memory model, accessing an individual item, x, in the external memory requires reading the entire block containing x into RAM.

means that each disk access could yield 4 096 bytes that are helpful in completing whatever operation we are doing.

This is the idea behind the *external memory model* of computation, illustrated schematically in Figura 14.1. In this model, the computer has access to a large external memory in which all of the data resides. This memory is divided into memory *blocks* each containing B words. The computer also has limited internal memory on which it can perform computations. Transferring a block between internal memory and external memory takes constant time. Computations performed within the internal memory are *free*; they take no time at all. The fact that internal memory computations are free may seem a bit strange, but it simply emphasizes the fact that external memory is so much slower than RAM.

In the full-blown external memory model, the size of the internal memory is also a parameter. However, for the data structures described in this chapter, it is sufficient to have an internal memory of size $O(B + \log_B n)$. That is, the memory needs to be capable of storing a constant number of blocks and a recursion stack of height $O(\log_B n)$. In most cases, the $O(B)$ term dominates the memory requirement. For example, even with the relatively small value $B = 32$, $B \geq \log_B n$ for all $n \leq 2^{160}$. In

decimal, $B \geq \log_B n$ for any

$$n \leq 1\,461\,501\,637\,330\,902\,918\,203\,684\,832\,716\,283\,019\,655\,932\,542\,976 \ .$$

14.1 The Block Store

The notion of external memory includes a large number of possible different devices, each of which has its own block size and is accessed with its own collection of system calls. To simplify the exposition of this chapter so that we can focus on the common ideas, we encapsulate external memory devices with an object called a `BlockStore`. A `BlockStore` stores a collection of memory blocks, each of size B. Each block is uniquely identified by its integer index. A `BlockStore` supports these operations:

1. `readBlock(i)`: Return the contents of the block whose index is `i`.
2. `writeBlock(i,b)`: Write contents of `b` to the block whose index is `i`.
3. `placeBlock(b)`: Return a new index and store the contents of `b` at this index.
4. `freeBlock(i)`: Free the block whose index is `i`. This indicates that the contents of this block are no longer used so the external memory allocated by this block may be reused.

The easiest way to imagine a `BlockStore` is to imagine it as storing a file on disk that is partitioned into blocks, each containing B bytes. In this way, `readBlock(i)` and `writeBlock(i,b)` simply read and write bytes $iB, \ldots, (i+1)B - 1$ of this file. In addition, a simple `BlockStore` could keep a *free list* of blocks that are available for use. Blocks freed with `freeBlock(i)` are added to the free list. In this way, `placeBlock(b)` can use a block from the free list or, if none is available, append a new block to the end of the file.

14.2 B-Trees

In this section, we discuss a generalization of binary trees, called B-trees, which is efficient in the external memory model. Alternatively, B-trees

can be viewed as the natural generalization of 2-4 trees described in Seção 9.1. (A 2-4 tree is a special case of a B-tree that we get by setting $B = 2$.)

For any integer $B \geq 2$, a *B-tree* is a tree in which all of the leaves have the same depth and every non-root internal node, u, has at least B children and at most $2B$ children. The children of u are stored in an array, u.children. The required number of children is relaxed at the root, which can have anywhere between 2 and $2B$ children.

If the height of a B-tree is h, then it follows that the number, ℓ, of leaves in the B-tree satisfies

$$2B^{h-1} \leq \ell \leq 2(2B)^{h-1} \ .$$

Taking the logarithm of the first inequality and rearranging terms yields:

$$\begin{aligned} h &\leq \frac{\log \ell - 1}{\log B} + 1 \\ &\leq \frac{\log \ell}{\log B} + 1 \\ &= \log_B \ell + 1 \ . \end{aligned}$$

That is, the height of a B-tree is proportional to the base-B logarithm of the number of leaves.

Each node, u, in B-tree stores an array of keys u.keys[0],...,u.keys[$2B-1$]. If u is an internal node with k children, then the number of keys stored at u is exactly $k-1$ and these are stored in u.keys[0],...,u.keys[$k-2$]. The remaining $2B-k+1$ array entries in u.keys are set to null. If u is a non-root leaf node, then u contains between $B-1$ and $2B-1$ keys. The keys in a B-tree respect an order similar to the keys in a binary search tree. For any node, u, that stores $k-1$ keys,

$$\texttt{u.keys}[0] < \texttt{u.keys}[1] < \cdots < \texttt{u.keys}[k-2] \ .$$

If u is an internal node, then for every i $\in \{0,\ldots,k-2\}$, u.keys[i] is larger than every key stored in the subtree rooted at u.children[i] but smaller than every key stored in the subtree rooted at u.children[i + 1]. Informally,

$$\texttt{u.children}[\texttt{i}] \prec \texttt{u.keys}[\texttt{i}] \prec \texttt{u.children}[\texttt{i}+1] \ .$$

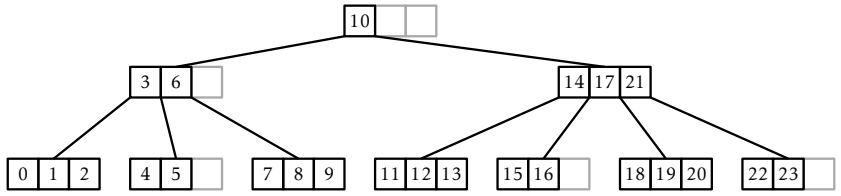


Figura 14.2: A B-tree with $B = 2$.

An example of a B-tree with $B = 2$ is shown in Figura 14.2.

Note that the data stored in a B-tree node has size $O(B)$. Therefore, in an external memory setting, the value of B in a B-tree is chosen so that a node fits into a single external memory block. In this way, the time it takes to perform a B-tree operation in the external memory model is proportional to the number of nodes that are accessed (read or written) by the operation.

For example, if the keys are 4 byte integers and the node indices are also 4 bytes, then setting $B = 256$ means that each node stores

$$(4 + 4) \times 2B = 8 \times 512 = 4096$$

bytes of data. This would be a perfect value of B for the hard disk or solid state drive discussed in the introduction to this chaper, which have a block size of 4096 bytes.

The `BTree` class, which implements a B-tree, stores a `BlockStore`, `bs`, that stores `BTree` nodes as well as the index, `ri`, of the root node. As usual, an integer, `n`, is used to keep track of the number of items in the data structure:

BTree
```
int n; // number of elements stored in the tree
int ri; // index of the root
BlockStore<Node*> bs;
```

14.2.1 Searching

The implementation of the `find(x)` operation, which is illustrated in Figura 14.3, generalizes the `find(x)` operation in a binary search tree. The

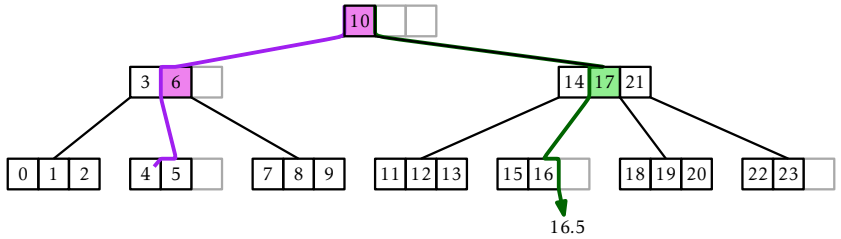


Figura 14.3: A successful search (for the value 4) and an unsuccessful search (for the value 16.5) in a *B*-tree. Shaded nodes show where the value of `z` is updated during the searches.

search for `x` starts at the root and uses the keys stored at a node, `u`, to determine in which of `u`'s children the search should continue.

More specifically, at a node `u`, the search checks if `x` is stored in `u.keys`. If so, `x` has been found and the search is complete. Otherwise, the search finds the smallest integer, `i`, such that `u.keys[i]` > `x` and continues the search in the subtree rooted at `u.children[i]`. If no key in `u.keys` is greater than `x`, then the search continues in `u`'s rightmost child. Just like binary search trees, the algorithm keeps track of the most recently seen key, `z`, that is larger than `x`. In case `x` is not found, `z` is returned as the smallest value that is greater or equal to `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)
      z = u->keys[i];
    ui = u->children[i];
  }
  return z;
}
```

Central to the `find(x)` method is the `findIt(a,x)` method that searches in a `null`-padded sorted array, `a`, for the value `x`. This method, illustrated in

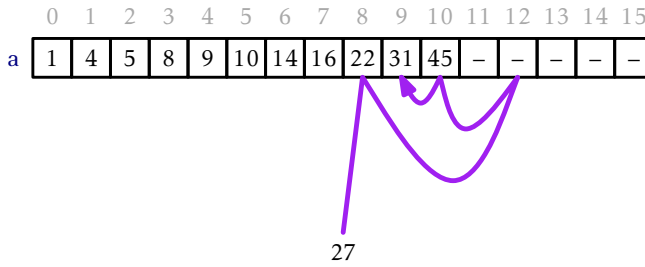


Figura 14.4: The execution of `findIt(a,27)`.

Figura 14.4, works for any array, `a`, where $a[0],\ldots,a[k-1]$ is a sequence of keys in sorted order and $a[k],\ldots,a[a.length-1]$ are all set to `null`. If `x` is in the array at position `i`, then `findIt(a,x)` returns $-i-1$. Otherwise, it returns the smallest index, `i`, such that $a[i] > x$ or $a[i] = null$.

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;
}
```

The `findIt(a,x)` method uses a binary search that halves the search space at each step, so it runs in $O(\log(a.length))$ time. In our setting, $a.length = 2B$, so `findIt(a,x)` runs in $O(\log B)$ time.

We can analyze the running time of a B-tree `find(x)` operation both in the usual word-RAM model (where every instruction counts) and in the external memory model (where we only count the number of nodes accessed). Since each leaf in a B-tree stores at least one key and the height of a B-Tree with ℓ leaves is $O(\log_B \ell)$, the height of a B-tree that stores

n keys is $O(\log_B n)$. Therefore, in the external memory model, the time taken by the find(x) operation is $O(\log_B n)$. To determine the running time in the word-RAM model, we have to account for the cost of calling findIt(a,x) for each node we access, so the running time of find(x) in the word-RAM model is

$$O(\log_B n) \times O(\log B) = O(\log n) \ .$$

14.2.2 Addition

One important difference between B-trees and the BinarySearchTree data structure from Seção 6.2 is that the nodes of a B-tree do not store pointers to their parents. The reason for this will be explained shortly. The lack of parent pointers means that the add(x) and remove(x) operations on B-trees are most easily implemented using recursion.

Like all balanced search trees, some form of rebalancing is required during an add(x) operation. In a B-tree, this is done by *splitting* nodes. Refer to Figura 14.5 for what follows. Although splitting takes place across two levels of recursion, it is best understood as an operation that takes a node u containing $2B$ keys and having $2B+1$ children. It creates a new node, w, that adopts u.children[B],...,u.children[$2B$]. The new node w also takes u's B largest keys, u.keys[B],...,u.keys[$2B-1$]. At this point, u has B children and B keys. The extra key, u.keys[$B-1$], is passed up to the parent of u, which also adopts w.

Notice that the splitting operation modifies three nodes: u, u's parent, and the new node, w. This is why it is important that the nodes of a B-tree do not maintain parent pointers. If they did, then the $B+1$ children adopted by w would all need to have their parent pointers modified. This would increase the number of external memory accesses from 3 to $B+4$ and would make B-trees much less efficient for large values of B.

The add(x) method in a B-tree is illustrated in Figura 14.6. At a high level, this method finds a leaf, u, at which to add the value x. If this causes u to become overfull (because it already contained $B-1$ keys), then u is split. If this causes u's parent to become overfull, then u's parent is also split, which may cause u's grandparent to become overfull, and so on. This process continues, moving up the tree one level at a time until

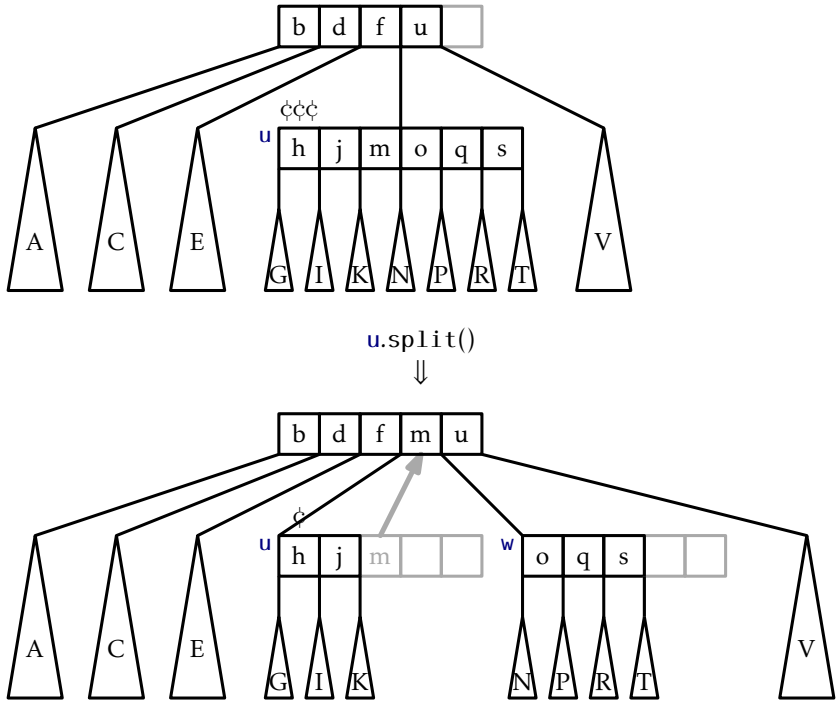


Figura 14.5: Splitting the node u in a B-tree ($B = 3$). Notice that the key u.keys[2] = m passes from u to its parent.

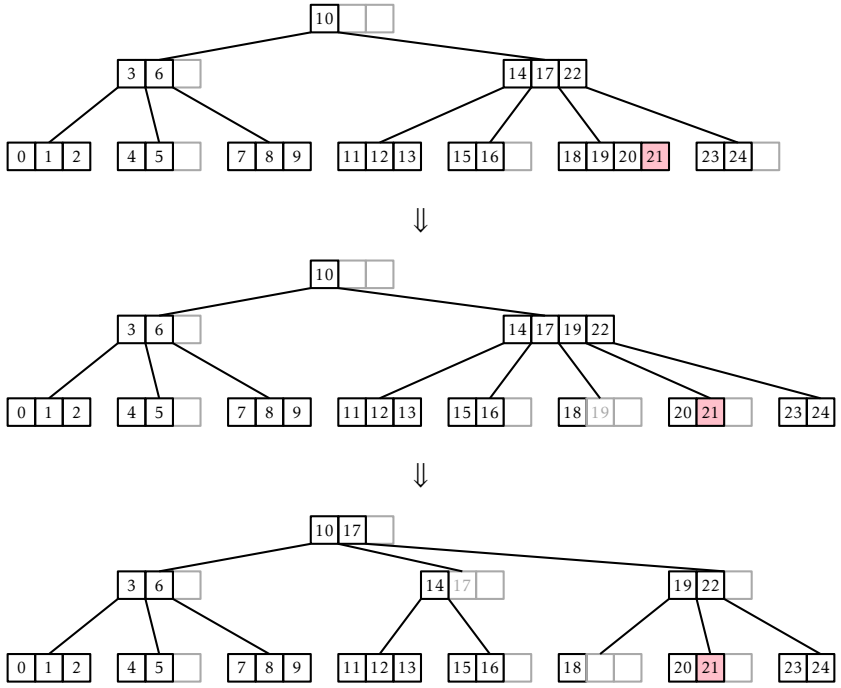


Figura 14.6: The `add(x)` operation in a `BTree`. Adding the value 21 results in two nodes being split.

reaching a node that is not overfull or until the root is split. In the former case, the process stops. In the latter case, a new root is created whose two children become the nodes obtained when the original root was split.

The executive summary of the `add(x)` method is that it walks from the root to a leaf searching for `x`, adds `x` to this leaf, and then walks back up to the root, splitting any overfull nodes it encounters along the way. With this high level view in mind, we can now delve into the details of how this method can be implemented recursively.

The real work of `add(x)` is done by the `addRecursive(x,ui)` method, which adds the value `x` to the subtree whose root, `u`, has the identifier `ui`. If `u` is a leaf, then `x` is simply inserted into `u.keys`. Otherwise, `x` is added recursively into the appropriate child, `u'`, of `u`. The result of this recursive call is normally `null` but may also be a reference to a newly-created node,

w, that was created because u' was split. In this case, u adopts w and takes its first key, completing the splitting operation on u'.

After the value x has been added (either to u or to a descendant of u), the addRecursive(x,ui) method checks to see if u is storing too many (more than $2B-1$) keys. If so, then u needs to be *split* with a call to the u.split() method. The result of calling u.split() is a new node that is used as the return value for 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;
}
```

The addRecursive(x,ui) method is a helper for the add(x) method, which calls addRecursive(x,ri) to insert x into the root of the B-tree. If addRecursive(x,ri) causes the root to split, then a new root is created that takes as its children both the old root and the new node created by the splitting of the old root.

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;
}
```

The `add(x)` method and its helper, `addRecursive(x,ui)`, can be analyzed in two phases:

Downward phase: During the downward phase of the recursion, before `x` has been added, they access a sequence of `BTree` nodes and call `findIt(a,x)` on each node. As with the `find(x)` method, this takes $O(\log_B n)$ time in the external memory model and $O(\log n)$ time in the word-RAM model.

Upward phase: During the upward phase of the recursion, after `x` has been added, these methods perform a sequence of at most $O(\log_B n)$ splits. Each split involves only three nodes, so this phase takes $O(\log_B n)$ time in the external memory model. However, each split involves moving B keys and children from one node to another, so in the word-RAM model, this takes $O(B \log n)$ time.

Recall that the value of B can be quite large, much larger than even $\log n$. Therefore, in the word-RAM model, adding a value to a B-tree can be much slower than adding into a balanced binary search tree. Later, in Seção 14.2.4, we will show that the situation is not quite so bad; the amortized number of split operations done during an `add(x)` operation is constant. This shows that the (amortized) running time of the `add(x)` operation in the word-RAM model is $O(B + \log n)$.

14.2.3 Removal

The remove(x) operation in a BTree is, again, most easily implemented as a recursive method. Although the recursive implementation of remove(x) spreads the complexity across several methods, the overall process, which is illustrated in Figura 14.7, is fairly straightforward. By shuffling keys around, removal is reduced to the problem of removing a value, x′, from some leaf, u. Removing x′ may leave u with less than $B-1$ keys; this situation is called an *underflow*.

When an underflow occurs, u either borrows keys from, or is merged with, one of its siblings. If u is merged with a sibling, then u's parent will now have one less child and one less key, which can cause u's parent to underflow; this is again corrected by borrowing or merging, but merging may cause u's grandparent to underflow. This process works its way back up to the root until there is no more underflow or until the root has its last two children merged into a single child. When the latter case occurs, the root is removed and its lone child becomes the new root.

Next we delve into the details of how each of these steps is implemented. The first job of the remove(x) method is to find the element x that should be removed. If x is found in a leaf, then x is removed from this leaf. Otherwise, if x is found at u.keys[i] for some internal node, u, then the algorithm removes the smallest value, x′, in the subtree rooted at u.children[i + 1]. The value x′ is the smallest value stored in the BTree that is greater than x. The value of x′ is then used to replace x in u.keys[i]. This process is illustrated in Figura 14.8.

The removeRecursive(x,ui) method is a recursive implementation of the preceding algorithm:

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;
}
bool removeRecursive(T x, int ui) {
```

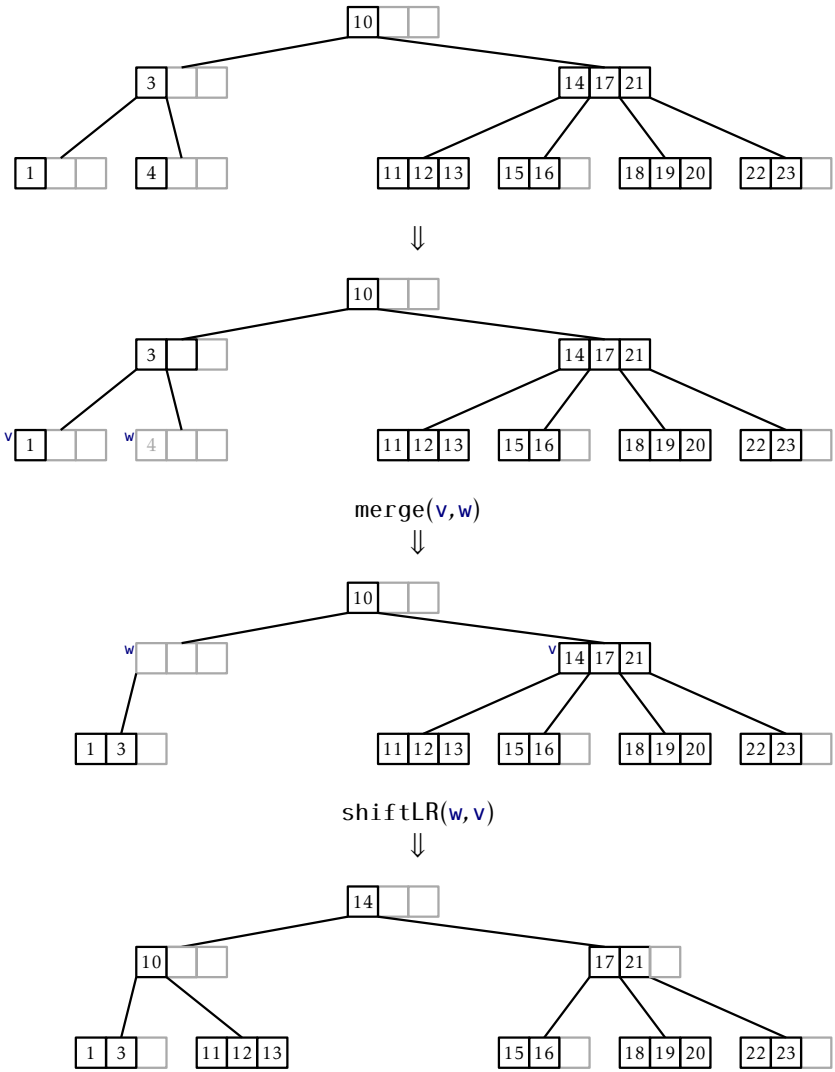


Figura 14.7: Removing the value 4 from a *B*-tree results in one merge and one borrowing operation.

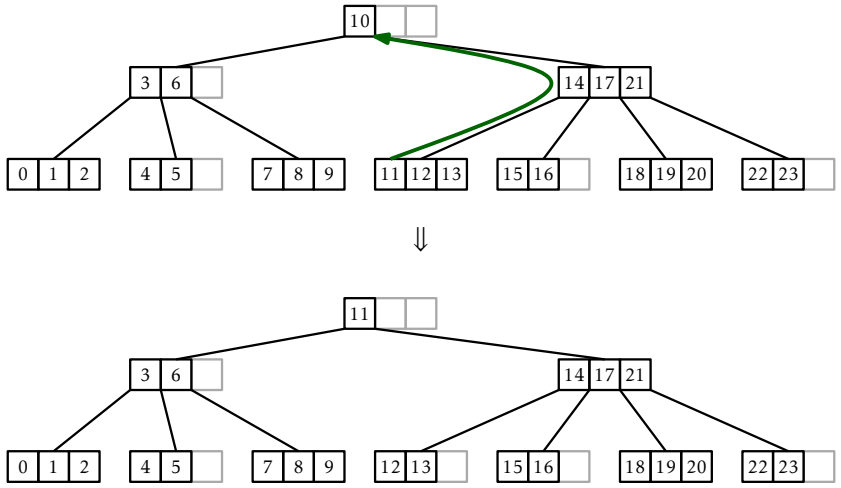


Figura 14.8: The `remove(x)` operation in a `BTree`. To remove the value `x` = 10 we replace it with the the value `x'` = 11 and remove 11 from the leaf that contains it.

```
  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;
}
```

Note that, after recursively removing the value `x` from the `i`th child of `u`, `removeRecursive(x,ui)` needs to ensure that this child still has at

least $B-1$ keys. In the preceding code, this is done using a method called `checkUnderflow(x,i)`, which checks for and corrects an underflow in the `i`th child of `u`. Let `w` be the `i`th child of `u`. If `w` has only $B-2$ keys, then this needs to be fixed. The fix requires using a sibling of `w`. This can be either child `i+1` of `u` or child `i-1` of `u`. We will usually use child `i-1` of `u`, which is the sibling, `v`, of `w` directly to its left. The only time this doesn't work is when `i=0`, in which case we use the sibling directly to `w`'s right.

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);
}
```

In the following, we focus on the case when `i` $\neq 0$ so that any underflow at the `i`th child of `u` will be corrected with the help of the `(i-1)`st child of `u`. The case `i` $= 0$ is similar and the details can be found in the accompanying source code.

To fix an underflow at node `w`, we need to find more keys (and possibly also children), for `w`. There are two ways to do this:

Borrowing: If `w` has a sibling, `v`, with more than $B-1$ keys, then `w` can borrow some keys (and possibly also children) from `v`. More specifically, if `v` stores `size(v)` keys, then between them, `v` and `w` have a total of

$$B-2+\texttt{size}(\texttt{w}) \geq 2B-2$$

keys. We can therefore shift keys from `v` to `w` so that each of `v` and `w` has at least $B-1$ keys. This process is illustrated in Figura 14.9.

Merging: If `v` has only $B-1$ keys, we must do something more drastic, since `v` cannot afford to give any keys to `w`. Therefore, we *merge* `v` and `w` as shown in Figura 14.10. The merge operation is the opposite of the split operation. It takes two nodes that contain a total of $2B-3$ keys and merges them into a single node that contains $2B-2$ keys. (The additional key comes from the fact that, when we merge `v` and

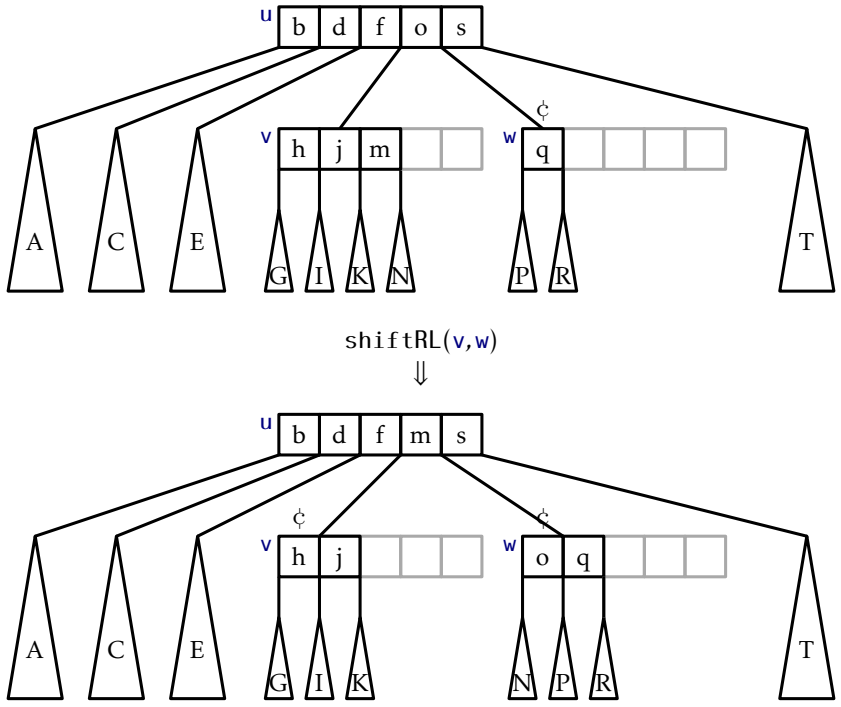


Figura 14.9: If v has more than $B-1$ keys, then w can borrow keys from v.

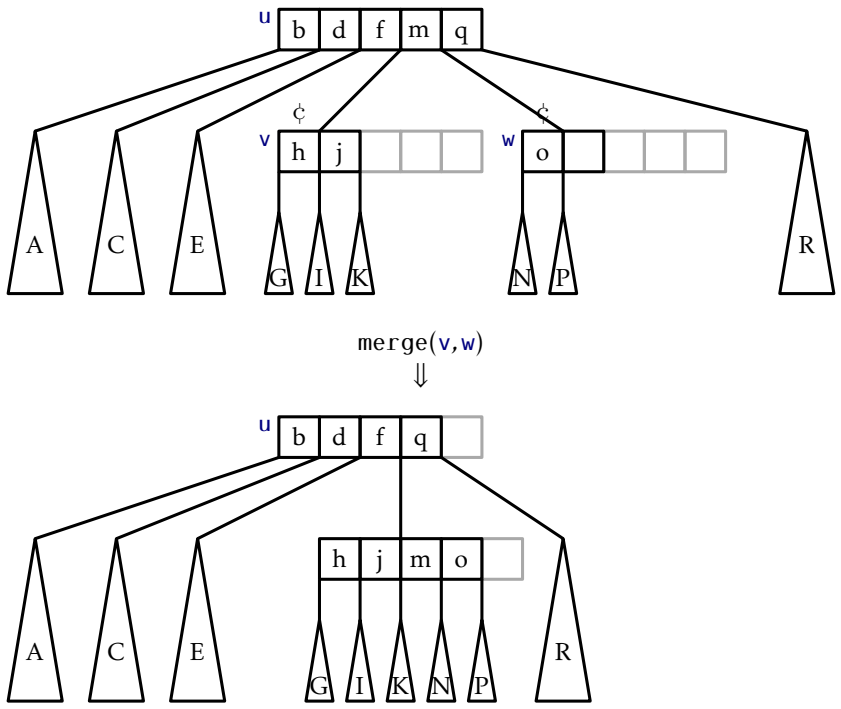


Figura 14.10: Merging two siblings `v` and `w` in a B-tree ($B = 3$).

`w`, their common parent, `u`, now has one less child and therefore needs to give up one of its keys.)

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);
    }
  }
}
```

To summarize, the `remove(x)` method in a B-tree follows a root to leaf path, removes a key `x` from a leaf, `u`, and then performs zero or more merge operations involving `u` and its ancestors, and performs at most one borrowing operation. Since each merge and borrow operation involves modifying only three nodes, and only $O(\log_B n)$ of these operations occur, the entire process takes $O(\log_B n)$ time in the external memory model. Again, however, each merge and borrow operation takes $O(B)$ time in the word-RAM model, so (for now) the most we can say about the running time required by `remove(x)` in the word-RAM model is that it is $O(B \log_B n)$.

14.2.4 Amortized Analysis of B-Trees

Thus far, we have shown that

1. In the external memory model, the running time of `find(x)`, `add(x)`, and `remove(x)` in a B-tree is $O(\log_B n)$.
2. In the word-RAM model, the running time of `find(x)` is $O(\log n)$ and the running time of `add(x)` and `remove(x)` is $O(B \log n)$.

The following lemma shows that, so far, we have overestimated the number of merge and split operations performed by B-trees.

Lema 14.1. *Starting with an empty B-tree and performing any sequence of m* `add(x)` *and* `remove(x)` *operations results in at most $3m/2$ splits, merges, and borrows being performed.*

Demonstração. The proof of this has already been sketched in Seção 9.3 for the special case in which $B = 2$. The lemma can be proven using a credit scheme, in which

1. each split, merge, or borrow operation is paid for with two credits, i.e., a credit is removed each time one of these operations occurs; and

2. at most three credits are created during any `add(x)` or `remove(x)` operation.

Since at most $3m$ credits are ever created and each split, merge, and borrow is paid for with with two credits, it follows that at most $3m/2$ splits, merges, and borrows are performed. These credits are illustrated using the ¢ symbol in Figures 14.5, 14.9, and 14.10.

To keep track of these credits the proof maintains the following *credit invariant*: Any non-root node with $B-1$ keys stores one credit and any node with $2B-1$ keys stores three credits. A node that stores at least B keys and most $2B-2$ keys need not store any credits. What remains is to show that we can maintain the credit invariant and satisfy properties 1 and 2, above, during each `add(x)` and `remove(x)` operation.

Adding: The `add(x)` method does not perform any merges or borrows, so we need only consider split operations that occur as a result of calls to `add(x)`.

Each split operation occurs because a key is added to a node, `u`, that already contains $2B-1$ keys. When this happens, `u` is split into two nodes, `u'` and `u''` having $B-1$ and B keys, respectively. Prior to this operation, `u` was storing $2B-1$ keys, and hence three credits. Two of these credits can be used to pay for the split and the other credit can be given to `u'` (which has $B-1$ keys) to maintain the credit invariant. Therefore, we can pay for the split and maintain the credit invariant during any split.

The only other modification to nodes that occur during an `add(x)` operation happens after all splits, if any, are complete. This modification involves adding a new key to some node `u'`. If, prior to this, `u'` had $2B-2$ children, then it now has $2B-1$ children and must therefore receive three credits. These are the only credits given out by the `add(x)` method.

Removing: During a call to remove(x), zero or more merges occur and are possibly followed by a single borrow. Each merge occurs because two nodes, v and w, each of which had exactly $B-1$ keys prior to calling remove(x) were merged into a single node with exactly $2B-2$ keys. Each such merge therefore frees up two credits that can be used to pay for the merge.

After any merges are performed, at most one borrow operation occurs, after which no further merges or borrows occur. This borrow operation only occurs if we remove a key from a leaf, v, that has $B-1$ keys. The node v therefore has one credit, and this credit goes towards the cost of the borrow. This single credit is not enough to pay for the borrow, so we create one credit to complete the payment.

At this point, we have created one credit and we still need to show that the credit invariant can be maintained. In the worst case, v's sibling, w, has exactly B keys before the borrow so that, afterwards, both v and w have $B-1$ keys. This means that v and w each should be storing a credit when the operation is complete. Therefore, in this case, we create an additional two credits to give to v and w. Since a borrow happens at most once during a remove(x) operation, this means that we create at most three credits, as required.

If the remove(x) operation does not include a borrow operation, this is because it finishes by removing a key from some node that, prior to the operation, had B or more keys. In the worst case, this node had exactly B keys, so that it now has $B-1$ keys and must be given one credit, which we create.

In either case—whether the removal finishes with a borrow operation or not—at most three credits need to be created during a call to remove(x) to maintain the credit invariant and pay for all borrows and merges that occur. This completes the proof of the lemma. □

The purpose of Lema 14.1 is to show that, in the word-RAM model the cost of splits, merges and joins during a sequence of m add(x) and remove(x) operations is only $O(Bm)$. That is, the amortized cost per operation is only $O(B)$, so the amortized cost of add(x) and remove(x) in the word-RAM model is $O(B+\log n)$. This is summarized by the following pair of theorems:

Teorema 14.1 (External Memory *B*-Trees)**.** *A* BTree *implements the* SSet *interface. In the external memory model, a* BTree *supports the operations* add(x), remove(x), *and* find(x) *in* $O(\log_B n)$ *time per operation.*

Teorema 14.2 (Word-RAM *B*-Trees)**.** *A* BTree *implements the* SSet *interface. In the word-RAM model, and ignoring the cost of splits, merges, and borrows, a* BTree *supports the operations* add(x), remove(x), *and* find(x) *in* $O(\log n)$ *time per operation. Furthermore, beginning with an empty* BTree*, any sequence of* m add(x) *and* remove(x) *operations results in a total of* $O(Bm)$ *time spent performing splits, merges, and borrows.*

14.3 Discussion and Exercises

The external memory model of computation was introduced by Aggarwal and Vitter [4]. It is sometimes also called the *I/O model* or the *disk access model*.

B-Trees are to external memory searching what binary search trees are to internal memory searching. *B*-trees were introduced by Bayer and McCreight [9] in 1970 and, less than ten years later, the title of Comer's ACM Computing Surveys article referred to them as ubiquitous [14].

Like binary search trees, there are many variants of *B*-Trees, including B^+-trees, B^*-trees, and counted *B*-trees. *B*-trees are indeed ubiquitous and are the primary data structure in many file systems, including Apple's HFS+, Microsoft's NTFS, and Linux's Ext4; every major database system; and key-value stores used in cloud computing. Graefe's recent survey [35] provides a 200+ page overview of the many modern applications, variants, and optimizations of *B*-trees.

B-trees implement the SSet interface. If only the USet interface is needed, then external memory hashing could be used as an alternative to *B*-trees. External memory hashing schemes do exist; see, for example, Jensen and Pagh [42]. These schemes implement the USet operations in $O(1)$ expected time in the external memory model. However, for a variety of reasons, many applications still use *B*-trees even though they only require USet operations.

One reason *B*-trees are such a popular choice is that they often perform better than their $O(\log_B n)$ running time bounds suggest. The rea-

son for this is that, in external memory settings, the value of B is typically quite large—in the hundreds or even thousands. This means that 99% or even 99.9% of the data in a B-tree is stored in the leaves. In a database system with a large memory, it may be possible to cache all the internal nodes of a B-tree in RAM, since they only represent 1% or 0.1% of the total data set. When this happens, this means that a search in a B-tree involves a very fast search in RAM, through the internal nodes, followed by a single external memory access to retrieve a leaf.

Exercício 14.1. Show what happens when the keys 1.5 and then 7.5 are added to the B-tree in Figura 14.2.

Exercício 14.2. Show what happens when the keys 3 and then 4 are removed from the B-tree in Figura 14.2.

Exercício 14.3. What is the maximum number of internal nodes in a B-tree that stores n keys (as a function of n and B)?

Exercício 14.4. The introduction to this chapter claims that B-trees only need an internal memory of size $O(B + \log_B n)$. However, the implementation given here actually requires more memory.

1. Show that the implementation of the `add(x)` and `remove(x)` methods given in this chapter use an internal memory proportional to $B \log_B n$.

2. Describe how these methods could be modified in order to reduce their memory consumption to $O(B + \log_B n)$.

Exercício 14.5. Draw the credits used in the proof of Lema 14.1 on the trees in Figures 14.6 and 14.7. Verify that (with three additional credits) it is possible to pay for the splits, merges, and borrows and maintain the credit invariant.

Exercício 14.6. Design a modified version of a B-tree in which nodes can have anywhere from B up to $3B$ children (and hence $B-1$ up to $3B-1$ keys). Show that this new version of B-trees performs only $O(m/B)$ splits, merges, and borrows during a sequence of m operations. (Hint: For this to work, you will have to be more agressive with merging, sometimes merging two nodes before it is strictly necessary.)

Exercício 14.7. In this exercise, you will design a modified method of splitting and merging in B-trees that asymptotically reduces the number of splits, borrows and merges by considering up to three nodes at a time.

1. Let u be an overfull node and let v be a sibling immediately to the right of u. There are two ways to fix the overflow at u:

   (a) u can give some of its keys to v; or

   (b) u can be split and the keys of u and v can be evenly distributed among u, v, and the newly created node, w.

   Show that this can always be done in such a way that, after the operation, each of the (at most 3) affected nodes has at least $B+\alpha B$ keys and at most $2B-\alpha B$ keys, for some constant $\alpha > 0$.

2. Let u be an underfull node and let v and w be siblings of u There are two ways to fix the underflow at u:

   (a) keys can be redistributed among u, v, and w; or

   (b) u, v, and w can be merged into two nodes and the keys of u, v, and w can be redistributed amongst these nodes.

   Show that this can always be done in such a way that, after the operation, each of the (at most 3) affected nodes has at least $B+\alpha B$ keys and at most $2B-\alpha B$ keys, for some constant $\alpha > 0$.

3. Show that, with these modifications, the number of merges, borrows, and splits that occur during m operations is $O(m/B)$.

Exercício 14.8. A B^+-tree, illustrated in Figura 14.11 stores every key in a leaf and keeps its leaves stored as a doubly-linked list. As usual, each leaf stores between $B-1$ and $2B-1$ keys. Above this list is a standard B-tree that stores the largest value from each leaf but the last.

1. Describe fast implementations of `add(x)`, `remove(x)`, and `find(x)` in a B^+-tree.

2. Explain how to efficiently implement the `findRange(x,y)` method, that reports all values greater than x and less than or equal to y, in a B^+-tree.

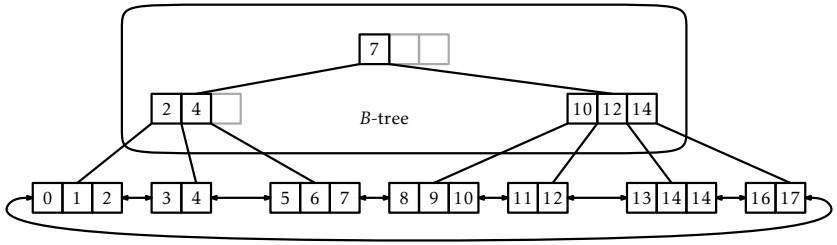


Figura 14.11: A B^+-tree is a B-tree on top of a doubly-linked list of blocks.

3. Implement a class, `BPlusTree`, that implements `find(x)`, `add(x)`, `remove(x)`, and `findRange(x,y)`.

4. B^+-trees duplicate some of the keys because they are stored both in the B-tree and in the list. Explain why this duplication does not add up to much for large values of B.

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] 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.

[11] 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.

[12] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick. Resizable arrays in optimal time and space. In Dehne et al. [17], pages 37–48.

[13] J. Carter and M. Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.

[14] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[15] C. Crane. Linear lists and priority queues as balanced binary trees. Technical Report STAN-CS-72-259, Computer Science Department, Stanford University, 1972.

[16] S. Crosby and D. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*, pages 29–44, 2003.

[17] 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.

[18] L. Devroye. Applications of the theory of records in the study of random trees. *Acta Informatica*, 26(1):123–130, 1988.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] M. Eytzinger. *Thesaurus principum hac aetate in Europa viventium (Cologne)*. 1590. In commentaries, 'Eytzinger' may appear in variant forms, including: Aitsingeri, Aitsingero, Aitsingerum, Eyzingern.

[27] R. W. Floyd. Algorithm 245: Treesort 3. *Communications of the ACM*, 7(12):701, 1964.

[28] 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.

[29] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[30] 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.

[31] 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.

[32] 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.

[33] A. Gambin and A. Malinowski. Randomized meldable priority queues. In *SOFSEM'98: Theory and Practice of Informatics*, pages 344–349. Springer, 1998.

[34] M. T. Goodrich and J. G. Kloss. Tiered vectors: Efficient dynamic arrays for rank-based sequences. In Dehne et al. [17], pages 205–216.

[35] G. Graefe. Modern b-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2010.

[36] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 2nd edition, 1994.

[37] 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.

[38] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.

[39] J. E. Hopcroft and R. E. Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[40] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, 1974.

[41] 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].

[42] M. S. Jensen and R. Pagh. Optimality in external memory hashing. *Algorithmica*, 52(3):403–411, 2008.

[43] P. Kirschenhofer, C. Martinez, and H. Prodinger. Analysis of an optimized search algorithm for skip lists. *Theoretical Computer Science*, 144:199–220, 1995.

[44] P. Kirschenhofer and H. Prodinger. The path length of random skip lists. *Acta Informatica*, 31:775–792, 1994.

[45] D. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.

[46] D. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.

[47] D. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1997.

[48] C. Y. Lee. An algorithm for path connection and its applications. *IRE Transaction on Electronic Computers*, EC-10(3):346–365, 1961.

[49] E. Lehman, F. T. Leighton, and A. R. Meyer. *Mathematics for Computer Science*. 2011. URL: http://courses.csail.mit.edu/6.042/spring12/mcs.pdf [cited 2012-09-06].

[50] C. Martínez and S. Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, 1998.

[51] E. F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292, 1959.

[52] 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.

[53] Oracle. *The Collections Framework*. URL: http://download.oracle.com/javase/1.5.0/docs/guide/collections/ [cited 2011-07-19].

[54] R. Pagh and F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[55] T. Papadakis, J. I. Munro, and P. V. Poblete. Average search and update costs in skip lists. *BIT*, 32:316–332, 1992.

[56] 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.

[57] M. Pătraşcu and M. Thorup. The power of simple tabulation hashing. *Journal of the ACM*, 59(3):14, 2012.

[58] 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].

[59] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[60] Redis. URL: http://redis.io/ [cited 2011-07-20].

[61] B. Reed. The height of a random binary search tree. *Journal of the ACM*, 50(3):306–332, 2003.

[62] S. M. Ross. *Probability Models for Computer Science*. Academic Press, Inc., Orlando, FL, USA, 2001.

[63] R. Sedgewick. Left-leaning red-black trees, September 2008. URL: http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf [cited 2011-07-21].

[64] R. Seidel and C. Aragon. Randomized search trees. *Algorithmica*, 16(4):464–497, 1996.

[65] 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.

[66] 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.

[67] P. Sinha. A memory-efficient doubly linked list. *Linux Journal*, 129, 2005. URL: http://www.linuxjournal.com/article/6828 [cited 2013-06-05].

[68] SkipDB. URL: http://dekorte.com/projects/opensource/SkipDB/ [cited 2011-07-20].

[69] 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.

[70] S. P. Thompson. *Calculus Made Easy*. MacMillan, Toronto, 1914. Project Gutenberg EBook 33283. URL: http://www.gutenberg.org/ebooks/33283 [cited 2012-06-14].

[71] 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.

[72] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

[73] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.

[74] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inf. Process. Lett.*, 17(2):81–84, 1983.

[75] J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

Índice Remissivo