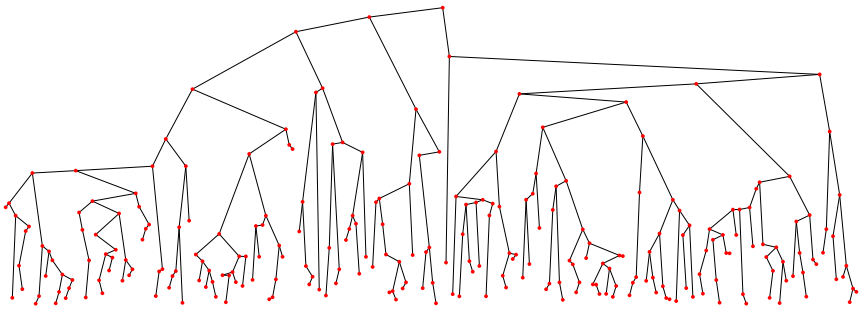


Estruturas de Dados Abertas: Uma introdução em Java

1ª edição

Autor: Pat Morin, Tradutor: Marcelo Keese Albertini

Conteúdo

Agradecimentos **ix**

Comentários do Tradutor **xi**

Por que este livro? **xiii**

1 Introdução **1**

1.1 A Necessidade de Eficiência 2

1.2 Interfaces 4

1.2.1 As Interfaces `Queue`, `Stack` e `Deque` 5

1.2.2 A Interface `List`: Sequências Lineares 7

1.2.3 A Interface `USet`: Unordered Sets (Conjuntos Desordenados) 8

1.2.4 A Interface `SSet`: Sorted Sets—Conjuntos Ordenados 9

1.3 Conceitos Matemáticos 10

1.3.1 Exponenciais e Logaritmos 10

1.3.2 Fatoriais 12

1.3.3 Notação assintótica 13

1.3.4 Randomização e Probabilidades 16

1.4 O Modelo de Computação 19

1.5 Corretude, Complexidade de Tempo e Complexidade de Espaço 21

1.6 Trechos de Código 23

1.7 Lista de Estruturas de Dados 24

1.8 Discussão e Exercícios 24

2 Listas Baseadas em Arrays **31**
2.1 ArrayStack: Operações de Stack Rápida Usando um Array 32
2.1.1 O Básico 33
2.1.2 Expansão e Redução 35
2.1.3 Resumo 37
2.2 FastArrayStack: Uma ArrayStack otimizada 38
2.3 ArrayQueue: Uma Queue Baseada Em Array 39
2.3.1 Resumo 43
2.4 ArrayDeque: Operações Rápidas para Deque Usando um Array 43
2.4.1 Resumo 45
2.5 DualArrayDeque: Construção de um Deque a Partir de Duas Stacks 46
2.5.1 Balanceamento 49
2.5.2 Resumo 51
2.6 RootishArrayStack: Uma Stack Array Eficiente No Uso de Espaço 52
2.6.1 Análise de expandir e encolher 57
2.6.2 Uso de Espaço 57
2.6.3 Resumo 58
2.6.4 Computando Raízes Quadradas 59
2.7 Discussão e Exercícios 62

3 Listas Ligadas **67**
3.1 SLList: Uma Lista Simplesmente Ligada 67
3.1.1 Operações de Queue 69
3.1.2 Resumo 70
3.2 DLList: Uma Lista Duplamente Ligada 71
3.2.1 Adicionando e Removendo 73
3.2.2 Resumo 74
3.3 SEList: Uma Lista Ligada Eficiente no Espaço 75
3.3.1 Requisitos de Espaço 77
3.3.2 Procurando Elementos 77
3.3.3 Adicionando um Elemento 79
3.3.4 Remoção de um Elemento 82
3.3.5 Análise amortizada do Espalhamento e Acumulação 84

3.3.6 Resumo 86
3.4 Discussão e Exercícios 86

4 Skiplists **91**
4.1 A Estrutura Básica 91
4.2 `SkiplistSSet`: Um `SSet` Eficiente 94
4.2.1 Resumo 97
4.3 `SkiplistList`: Uma `List` com Acesso Aleatório Eficiente . 97
4.3.1 Resumo 103
4.4 Análise de Skiplists 103
4.5 Discussão e Exercícios 106

5 Tabelas Hash **111**
5.1 `ChainedHashTable`: Hashing com Encadeamento 111
5.1.1 Hashing Multiplicativo 114
5.1.2 Resumo 118
5.2 `LinearHashTable`: Sondagem Linear 118
5.2.1 Análise da Sondagem Linear 122
5.2.2 Resumo 125
5.2.3 Hashing por tabulação 126
5.3 Códigos Hash 127
5.3.1 Códigos hash para Tipos de Dados Primitivos 127
5.3.2 Códigos Hash para Objetos Compostos 128
5.3.3 Códigos Hash para Arrays e Strings 130
5.4 Discussão e Exercícios 133

6 Árvores Binárias **139**
6.1 `BinaryTree`: Uma Árvore Binária Básica 141
6.1.1 Algoritmos recursivos 142
6.1.2 Percorrendo Árvores Binárias 142
6.2 `BinarySearchTree`: Uma Árvore Binária de Busca Desbalanceada 145
6.2.1 Busca 146
6.2.2 Adição 148
6.2.3 Remoção 150
6.2.4 Resumo 152

6.3 Discussão e Exercícios 153

7 Árvores Binárias de Busca Aleatórias 159

7.1 Árvores Binárias de Busca Aleatórias 159

7.1.1 Prova do Lema 7.1 162

7.1.2 Resumo 164

7.2 `Treap`: Uma Árvore Binária de Busca Aleatória 165

7.2.1 Resumo 174

7.3 Discussão e Exercícios 175

8 Árvores Scapegoat 181

8.1 `ScapegoatTree`: Uma Árvore Binária de Busca com Reconstrução Parcial 183

8.1.1 Análise de Corretude e Tempo de Execução 186

8.1.2 Resumo 189

8.2 Discussão e Exercícios 189

9 Árvores Rubro-Negras 193

9.1 Árvore 2-4 194

9.1.1 Adição de uma Folha 195

9.1.2 Remoção de uma Folha 195

9.2 `RedBlackTree`: Uma Árvore 2-4 Simulada 198

9.2.1 Árvores Rubro-Negras e Árvores 2-4 199

9.2.2 Árvores Rubro-Negras Pendentes à Esquerda 201

9.2.3 Adição 204

9.2.4 Remoção 207

9.3 Resumo 213

9.4 Discussão e Exercícios 215

10 Heaps 219

10.1 `BinaryHeap`: Uma Árvore Binária Implícita 219

10.1.1 Resumo 225

10.2 `MeldableHeap`: Uma Heap Randomizada Combinável . . . 225

10.2.1 Análise da Operação `merge(h1,h2)` 228

10.2.2 Resumo 230

10.3 Discussão e Exercícios 230

11 Algoritmos de Ordenação **233**
11.1 Ordenação Baseada em Comparação 234
11.1.1 Mergesort 234
11.1.2 Quicksort 238
11.1.3 Heapsort 241
11.1.4 Um Limitante Inferior para Ordenação 244
11.2 Counting Sort e Radix Sort 247
11.2.1 Counting Sort 248
11.2.2 Radix sort 250
11.3 Discussão e Exercícios 252

12 Grafos **255**
12.1 `AdjacencyMatrix`: Um Grafo com uma Matriz 257
12.2 `AdjacencyLists`: Um Grafo como uma Coleção de Listas . 260
12.3 Travessia em Grafos 264
12.3.1 Busca em Largura 265
12.3.2 Busca em Profundidade 267
12.4 Discussão e Exercícios 270

13 Estruturas de Dados para Inteiros **275**
13.1 `BinaryTrie`: Uma Árvore de Busca Digital 276
13.2 `XFastTrie`: Buscando em Tempo Duplamente Logarítmico 282
13.3 `YFastTrie`: Um `SSet` com Tempo Duplamente Logarítmico 285
13.4 Discussão e Exercícios 291

14 Busca em Memória Externa **293**
14.1 A *Block Store* 295
14.2 B-Trees 296
14.2.1 Busca 298
14.2.2 Adição 300
14.2.3 Remoção 305
14.2.4 Análise Amortizada da *B*-Tree 311
14.3 Discussão e Exercícios 314

Referências **319**

Índice **327**

Agradecimentos

Do Autor, Pat Morin

I am grateful to Nima Hoda, who spent a summer tirelessly proofreading many of the chapters in this book; to the students in the Fall 2011 offering of COMP2402/2002, who put up with the first draft of this book and spotted many typographic, grammatical, and factual errors; and to Morgan Tunzelmann at Athabasca University Press, for patiently editing several near-final drafts.

Do Tradutor, Marcelo Keese Albertini

Agradeço à todos que me motivaram a fazer esta tradução. Agradeço Hélio Albertini e Luiz Brito por ajudarem na revisão do texto. Em especial, agradeço à minha esposa Madeleine Albertini por me apoiar neste trabalho, que exigiu mais horas de trabalho do que o normal.

Comentários do Tradutor

Na tradução deste livro, várias decisões de traduções foram tomadas ponderando, simultâneamente, a simplicidade de compreensão do texto e a fidelidade à palavra originalmente usada na literatura especializada.

Os nomes de estruturas de dados foram usados em inglês somente quando não há tradução consolidada para o português.

Uma decisão importante foi optar por não traduzir os códigos fontes (e algoritmos), incluindo nomes de variáveis, campos, métodos e comentários pois assim eles serão encontrados nos códigos disponíveis nos repositórios de softwares abertos e nas *Application Programming Interface* (API) de bibliotecas e linguagens de programação.

Em especial, acredito que os comentários nos códigos sejam curtos e simples o suficientes para serem compreendidos usando o contexto em questão.

Por que este livro?

Existem muitos livros introdutórios à disciplina de estruturas de dados. Alguns deles são muito bons. A maior parte deles são pagos e a grande maioridade de estudantes de Ciência da Computação e Sistemas de Informação irá gastar algum dinheiro em um livro de estruturas de dados.

Vários livros gratuitos sobre estruturas de dados estão disponíveis online. Alguns muito bons, mas a maior parte está ficando desatualizada. Grande parte deles torna-se gratuita quando seus autores e/ou editores decidem parar de atualizá-los. Atualizar esses livros frequentemente não é possível, por duas razões: (1) Os direitos autorais pertencem ao autor e/ou editor, os quais podem não autorizar tais atualizações. (2) O *código-fonte* desses livros muitas vezes não está disponível. Isto é, os arquivos Word, WordPerfect, FrameMaker ou LaTeX para o livro não estão acessíveis e, ainda, a versão do software que processa esses arquivos pode não estar mais disponível.

A meta deste projeto é libertar estudantes de graduação de Ciência da Computação de ter que pagar por um livro introdutório à disciplina de estruturas de dados.

Eu [1] decidi implementar essa meta ao tratar esse livro como um projeto de Software Aberto.

Os arquivos-fonte originais em LaTeX, Java e scripts para montar este livro estão disponíveis para download a partir do website do autor[2] e também, de modo mais importante, em um site confiável de gerenciamento

[1] The translator to Portuguese language is deeply grateful to the original author of this book Pat Morin for his decision, which allows the availability of a good quality and free book in the native language of my Brazilian students.

[2] http://opendatastructures.org

de códigos-fonte.[3][4]

O código-fonte disponível é publicado sob uma licença Creative Commons Attribution, o que quer dizer que qualquer um é livre para *compartilhar*: copiar, distribuir e transmitir essa obra; e *modificar*: adaptar a obra, incluindo o direito de fazer uso comercial da obra. A única condição a esse direitos é *atribuição*: você deve reconhecer que a obra derivada contém código e/ou texto de `opendatastructures.org`.

Qualquer pessoa pode contribuir com correções usando o sistema de gerenciamento de códigos-fonte `git`. Qualquer pessoa também pode fazer fork (criar versão alternativa) dos arquivos-fonte deste livro e desenvolver uma versão separada (por exemplo, em outra linguagem de programação). Minha esperança é que, desse modo, este livro continuará a ser útil mesmo depois que o meu interesse no projeto diminua.

[3] `https://github.com/patmorin/ods`

[4] Tradução em português em `https://github.com/albertiniufu/ods`

Capítulo 1

Introdução

Todo currículo de Ciência da Computação no mundo inclui ao menos uma disciplina sobre estruturas de dados e algoritmos. Estruturas de dados são *realmente* importantes; elas melhoram nossa qualidade de vida e até salvam vidas. Muitas companhias que valem múltiplos milhões e bilhões de dólares foram construídas em torno de estruturas de dados.

Como isso é possível? Se pararmos para pensar nisso, percebemos que interagimos com estruturas de dados constantemente.

- Abrir um arquivo: Sistema de arquivos estruturas de dados são usadas para encontrar as partes daquele arquivo em disco para serem recuperadas. Isso não é fácil; discos contém centenas de milhões de blocos. O conteúdo do seu arquivo pode estar armazenado em qualquer um deles.

- Procurar contato no telefone: Uma estrutura de dados é usada para procurar por um número de telefone na sua lista de contatos baseada em informação parcial mesmo antes de você terminar de discar/digitar Isso não é fácil; o seu telefone pode ter informações sobre muitas pessoas—todos que você já entrou em contato via telefone ou email—e seu telefone não tem um processador muito rápido ou muita memória.

- Entrar na sua rede social preferida: Os servidores da rede usam a sua informação de login para procurar as informações da sua conta. Isso não é fácil; as redes sociais mais populares tem centenas de milhões de usuários ativos.

- Fazer uma busca na web: Um motor de busca usa estruturas de dados para buscar por páginas na web que contêm os seus termos de busca. Isso não é fácil; existem mais de 8.5 bilhões de páginas web na Internet e cada página tem muitos potenciais termos de busca.

- Serviços de telefone de emergência (9-1-1): Os serviços de emergência procuram pelo seu número de telefone em uma estrutura de dados que mapeia números de telefone em endereços para que carros de polícia, ambulâncias ou os bombeiros possam ser enviados imediatamente. Isso é importante; a pessoa fazendo a chamada pode não conseguir fornecer o endereço exato de onde está e um atraso pode significar a diferença entre a vida e a morte.

1.1 A Necessidade de Eficiência

Na próxima seção, estudaremos as operações aceitas pelas estruturas de dados mais comuns. Qualquer pessoa com um pouco de experiência em programação verá que essas operações não são difíceis de implementar corretamente. Podemos implementar ao percorrer/iterar todos os elementos de um array ou lista e possivelmente adicionar ou remover um elemento. Esse tipo de implementação é fácil, mas não muito eficiente. Isso realmente importa? Computadores estão ficando cada vez mais rápidos. Talvez a implementação mais óbvia seja boa o suficiente. Vamos fazer alguns cálculos aproximados para verificar isso.

Número de operações: Imagine uma aplicação com um conjunto de dados de tamanho moderado, digamos de um milhão (10^6) de itens. É razoável, na maior parte das aplicações, presumir que a aplicação vai precisar verificar cada item pelo menos uma vez. Isso significa que podemos esperar fazermos pelo menos um milhão (10^6) de buscas nesses dados. Se cada uma dessas 10^6 buscas inspecionar cada um dos 10^6 itens, isso resulta em um total de $10^6 \times 10^6 = 10^{12}$ (mil bilhões) inspeções.

Velocidades do Processador: No momento de escrita deste livro, mesmo um computador de mesa bem rápido não pode fazer mais de um bilhão

(10^9) de operações por segundo.[1] Isso significa que essa aplicação vai levar pelo menos $10^{12}/10^9 = 1000$ segundos, ou aproximadamente 16 minutos e 40 segundos. Dezesseis minutos é uma eternidade em tempos computacionais, mas uma pessoa pode estar disposta a aceitá-lo (por exemplo, se ela estiver indo para um café).

Conjuntos de dados maiores: Agora considere que uma empresa como a Google, que indexa mais de 8.5 bilhões de páginas web. De acordo com os nossos cálculos, fazer qualquer tipo de consulta nesses dados levaria pelo menos 8.5 segundos. Nós já sabemos que esse não é o caso; buscas web são respondidas em bem menos de 8.5 segundos e elas são bem mais complicadas do que somente verificar se uma dada página está na lista de páginas indexadas. No momento de escrita deste livro, a Google recebe aproximadamente 4500 consultas por segundo, o que significa que eles precisariam de pelo menos $4500 \times 8.5 = 38,250$ excelentes servidores somente para manter esse tempo de resposta.

A solução: Esses exemplos nos dizem que as implementações mais óbvias de estruturas de dados não funcionam bem quando o número de itens, n, na estrutura de dados e o número de operações m, feitas na estrutura de dados são altos. Nesses casos, o tempo (medido em termos de, digamos, duração da execução de instruções de máquina) é proporcional a $n \times m$.

A solução, é claro, cuidadosamente organizar os dados na estrutura de dados de forma que nem toda operação exija que todos os itens de dados sejam inspecionados. Embora pareça inicialmente impossível, descreveremos estruturas de dados em que a busca requer olhar em apenas dois itens em média, independentemente do número de itens guardados na estrutura de dados. No nosso computador de um bilhão de instruções por segundo, levaria somente 0.000000002 segundos para buscar em uma estrutura de dados contendo um bilhão de itens (ou um trilhão ou um quatrilhão ou mesmo um quintilhão de itens).

Nós veremos também implementações de estruturas de dados que mantêm os itens em ordem, onde o número de itens inspecionados du-

[1] Velocidades de computadores vão até poucos gigahertz (bilhões de ciclos por segundo) e cada operação tipicamente exige alguns ciclos.

rante uma operação cresce muito lentamente em função do número de itens na estrutura de dados. Por exemplo, podemos manter um conjunto ordenado com um bilhão de itens e inspecionar no máximo 60 itens para qualquer operação. No nosso computador de um bilhão de instruções por segundo, essas operações levam 0.00000006 segundos cada.

O resto deste capítulo revisa brevemente alguns dos principais conceitos usados ao longo do resto do livro. A Seção 1.2 descreve as interfaces implementadas por todas as estruturas de dados descritas neste livro e devem ser consideradas como leitura obrigatória.

O restante das seções discute:

- revisão de conceitos matemáticos incluindo exponenciais, logaritmos, fatoriais, notação assintótica (big-Oh), probabilidades e aleatorização;
- modelos de computação;
- corretude, tempo de execução e uso de espaço;
- uma visão geral do resto dos capítulos; e
- uma amostra de código juntamente com convenções de escrita de código usadas neste livro.

Um leitor com (ou mesmo sem) conhecimento desses assuntos pode simplesmente pulá-las agora e voltar se necessário.

1.2 Interfaces

Ao discutir sobre 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 o faz.

Uma *interface*, às vezes também chamada de *tipo abstrato de dados* (*abstract data type*, em inglês), define o conjunto de operações aceitas por uma estrutura de dados e a semântica, ou significado, dessas operações.

Uma interface nos diz nada sobre como a estrutura de dados implementa essas operações; ela somente provê uma lista de operações aceitas

juntamente com as especificações sobre quais tipos de argumentos cada operação aceita e o valor retornado por cada operação.

A *implementação* de uma estrutura de dados, por outro lado, inclui a representação interna da estrutura de dados assim como as definições dos algoritmos que implementam as operações aceitas pela estrutura de dados. Então, pode haver muitas implementações de uma dada interface. Por exemplo, no Capítulo 2, veremos implementações da interface `List` usando arrays e no Capítulo 3 veremos implementações da interface `List` usando estruturas de dados baseadas em ponteiros. Ambas implementam a mesma interface, `List`, mas de modos distintos.

1.2.1 As Interfaces `Queue`, `Stack` e `Deque`

A interface `Queue` (Fila, em português) representa uma coleção de elementos à qual podemos adicionar elementos e remover o próximo elemento. Mais precisamente, as operações realizadas pela interface `Queue` são

- `add(x)`: adiciona o valor `x` à `Queue`
- `remove()`: remove o próximo (previamente adicionado) valor, `y`, da `Queue` e retorna `y`

Note que a operação `remove()` não recebe argumentos. A *política de enfileiramento* da `Queue` decide qual é o próximo elemento a ser removido.

Existem muitas políticas de enfileiramento possíveis, sendo que entre as mais comuns estão FIFO, por prioridades e LIFO.

Uma *`Queue` FIFO (first-in-first-out – primeiro a entrar, primeiro a sair)*, que é ilustrada em Figura 1.1, remove itens na mesma ordem em que são adicionados, do meio jeito que uma fila de compras em um supermercado funciona. Esse é o tipo mais comum de `Queue` de forma que o qualificador FIFO é frequentemente omitido. Em outros textos, as operações `add(x)` e `remove()` em uma `Queue` FIFO são frequentemente chamadas de `enqueue(x)` (enfileirar) e `dequeue()` (desenfileirar), respectivamente.

Uma *`Queue` com prioridades*, ilustrada em Figura 1.2, sempre remove o menor elemento da `Queue`, decidindo empates arbitrariamente. Isso é similar ao modo no qual pacientes passam por triagem em uma sala

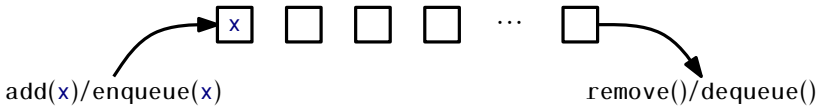


Figura 1.1: Uma Queue FIFO — primeiro a entrar, primeiro a sair.

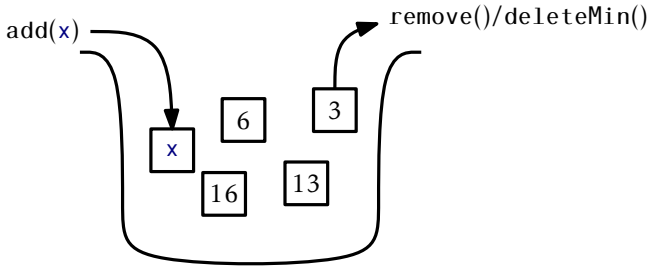


Figura 1.2: Uma Queue com prioridades — itens são removidos de acordo com suas prioridades (em inglês, *priority queue*).

de emergência de um hospital. Conforme os pacientes chegam, eles são avaliados e, então, vão à sala de espera. Quando um médico torna-se disponível, ele primeiro trata o paciente na situação mais grave. A operação remove() em um Queue com prioridades é geralmente chamada de deleteMin() em outros textos.

Uma política de enfileiramento muito comum é a política LIFO (last-in-first-out, último a entrar, primeiro a sair, em português) , ilustrada na Figura 1.3. Em uma *Queue LIFO*, o elemento mais recentemente adicionado é o próximo a ser removido. Isso é melhor visualizado em termos de uma pilha de pratos; pratos são posicionados no topo da pilha a também removidos do topo. Essa estrutura é tão comum que recebe seu próprio nome: Stack (pilha, em português). Frequentemente, ao referenciar uma Stack, as operações add(x) e remove() recebem os nomes de push(x) e pop(); isso é para evitar confusões entre as políticas LIFO e FIFO.

Uma Deque é uma generalização de ambos Queue FIFO e Queue LIFO (Stack).

Uma Deque representa uma sequência de elementos, com uma frente e um verso (um início e um fim). Elementos podem ser adicionados no início da sequência ou no final da sequência. Os nomes das operações da Deque são auto-explicativos: addFirst(x), removeFirst(), addLast(x) e

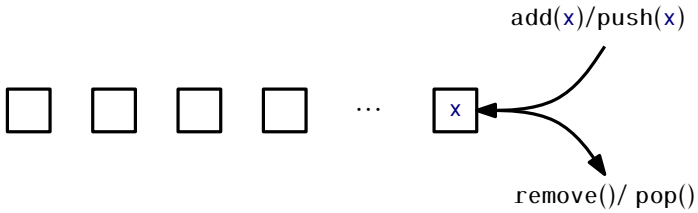


Figura 1.3: Uma stack (pilha, em português) — LIFO (último-a-entrar-primeiro-a-sair).

removeLast(). Vale notar que uma Stack pode ser implementada usando somente addFirst(x) e removeFirst() enquanto uma Queue FIFO pode ser implementada usando addLast(x) e removeFirst().

1.2.2 A Interface List: Sequências Lineares

Este livro vai falar muito pouco sobre as interfaces Queue FIFO, Stack, ou Deque. Isso porque essas interfaces são englobadas pela interface List. Uma List, ilustrada na Figura 1.4, representa uma sequência, $x_0, \ldots, x_{n-1}$, de valores. A interface List inclui as operações a seguir:

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

2. get(i): retorna o valor x_i

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

4. add(i, x): adicionar x à posição i, deslocando $x_i, \ldots, x_{n-1}$;
   Atribua $x_{j+1} = x_j$, para todo $j \in \{n-1, \ldots, i\}$, incremente n, e faça $x_i = x$

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

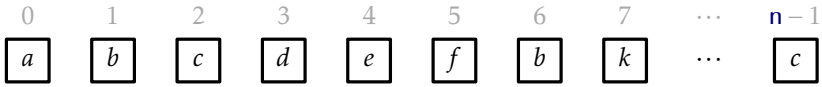


Figura 1.4: Uma List representa uma sequência indexada por $0, 1, 2, \ldots, n-1$. Nessa List uma chamada para get(2) retornaria o valor c.

Note que essas operações são suficientes para implementar a interface Deque:

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

Embora normalmente não discutiremos as interfaces Stack, Deque e Queue FIFO nos seguintes capítulos, os termos Stack e Deque são às vezes usados nos nomes das estruturas de dados que implementam a interface List. Quando isso acontece, queremos destacar que essas estruturas de dados podem ser usadas para implementar a interface Stack ou Deque eficientemente. Por exemplo, a classe ArrayDeque é uma implementação da interface List que implementa todas as operações Deque em tempo constante por operação.

1.2.3 A Interface USet: Unordered Sets (Conjuntos Desordenados)

A interface USet representa um conjunto desordenado de elementos únicos (sem repetições), que simulam um *conjunto* (em inglês, *set*) matemático. Um USet contém n elementos *distintos*; nenhum elemento aparece mais de uma vez; os elementos não estão em nenhuma ordem específica. Um USet possui as seguintes operações:

1. size(): retorna o número, n, de elementos no conjunto
2. add(x): adiciona o elemento x ao conjunto, se não já presente; Adiciona x ao conjunto considerando que não há elemento y no conjunto tal que x é considerado igual a y. Retorna true se x foi adicionado ao conjunto e false caso contrário.

3. `remove(x)`: remove `x` do conjunto;
   Achar um elemento `y` no conjunto tal que `x` iguala a `y` e remove `y`. Retorna `y`, ou `null` se tal elemento não existe.

4. `find(x)`: achar `x` no conjunto se existir;
   Achar um elemento `y` no conjunto tal que `y` seja igual a `x`. Retornar `y` ou `null` se tal elemento não exista no conjunto.

Essas definições são um pouco cuidadosas ao distinguir `x`, o elemento que estamos a remover ou buscar, de `y`, o elemento que poderemos remover ou achar. Isso é porque `x` e `y` podem ser na realidade objetos distintos que são, para todos os efeitos nessa situação, tratados como iguais.

[2] Tal distinção é útil porque permite a criação de *dicionários* ou *mapas* que mapeiam chaves em valores.

Para criar um dicionário/mapa, forma-se objetos chamados `Pair` (par, em português) cada qual contém uma *chave* (*key*) e um *valor* (*valor*). Dois pares `Pair` são tratados como iguais se suas chaves forem iguais. Se armazenamos algum par (`k`, `v`) em um `USet` e então 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`, usando somente a chave `k`.

1.2.4 A Interface `SSet`: Sorted Sets—Conjuntos Ordenados

A interface `SSet` representa um conjunto ordenado de elementos. Um `SSet` guarda elementos provenientes de alguma ordem total, tal que quaisquer dois elementos `x` e `y` podem ser comparados. Em código, isso será feito com um método chamado `compare(x,y)` no qual

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

Um `SSet` possui os métodos `size()`, `add(x)` e `remove(x)` com exatamente a mesma semântica que a interface `USet`. A diferença entre um `USet` e um `SSet` está no método `find(x)`:

[2] Em Java, isso é realizado ao sobrescrever, `@Override`, os métodos da classe dos elementos `equals(y)` e `hashCode()`.

4. `find(x)`: localizar `x` no conjunto ordenado;
   Achar o menor elemento `y` no conjunto tal que $y \geq x$. Retornar `y` ou `null` se o elemento não existir.

Essa versão da operação `find(x)` é às vezes chamada de *busca de sucessor* (em inglês, *successor search*). Ela difere de um modo fundamental de `USet.find(x)` pois retorna um resultado mesmo quando não há elemento igual a `x` no conjunto.

A distinção entre a operação `find(x)` de `USet` e de `SSet` é muito importante e frequentemente esquecida ou não percebida. A funcionalidade extra provida por um `SSet` frequentemente vem com um preço que inclui tanto tempo de execução mais alto quanto uma complexidade de implementação maior. Por exemplo, a maior parte das implementações `SSet` discutidas neste livro tem operações `find(x)` com tempo de execução que são logarítmicas no tamanho do conjunto. Por outro lado, a implementação de um `USet` com uma `ChainedHashTable` no Capítulo 5 tem uma operação `find(x)` que roda em tempo esperado constante. Ao escolher quais dessas estruturas usar, deve-se sempre usar um `USet` a menos que a funcionalidade extra oferecida por um `SSet` seja verdadeiramente necessária.

1.3 Conceitos Matemáticos

Nesta seção, revisaremos algumas noções matemáticas e ferramentas usadas ao longo deste livro, incluindo logaritmos, notação big-Oh e teoria das probabilidades. Esta revisão será breve e não tem a intenção de ser uma introdução. Leitores que acham que precisam saber mais desses conceitos são encorajados a ler, e fazer os exercícios relacionados, as seções apropriadas do excelente (e livre) texto didático sobre Matemática para Ciência da Computação [50].

1.3.1 Exponenciais e Logaritmos

A expressão b^x denota o número b elevado à potência x. Se x é um inteiro positivo, então o resultado é somente o valor de b multiplicado por ele

mesmo $x-1$ 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, podemos ainda definir a exponenciação em termos da função exponencial e^x (ver a seguir), que é definida em termos de uma série exponencial, mas isso é melhor deixar para um texto sobre cálculo.

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

$$b^x = k \ .$$

A maior parte dos logaritmos neste livro são base 2 (*logaritmos binários*). Nesse caso, omitiremos a base, de forma que $\log k$ é uma abreviação para $\log_2 k$.

Um jeito informal, mas útil, de pensar sobre logaritmos é pensar de $\log_b k$ como o número de vezes que temos que dividir k por b antes do resultado ser menor ou igual a 1. Por exemplo, quando alguém usa uma busca binária, cada comparação reduz o número de possíveis respostas por um fator de 2. Isso é repetido até que exista no máximo uma única resposta possível. Portanto, o número de comparações feitas pela busca binária quando existem inicialmente até $n+1$ possíveis respostas 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 — a *constante de Euler* — é dada por

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

O logaritmo natural aparece 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 trocar a base de um logaritmo:

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

Por exemplo, podemos usar essas duas manipulações para comparar os logaritmos natural e binário:

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

1.3.2 Fatoriais

Em uma ou duas partes deste livro, a função *fatorial* é usada. Para um inteiro não-negativo n, a notação $n!$ (pronunciada "n fatorial") é definida para representar

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

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

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

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

onde

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

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 provada aproximando $\ln(n!) = \ln 1 + \ln 2 + \cdots + \ln n$ com a integral $\int_1^n \ln n \, \mathrm{d}n = n \ln n - n + 1$.)

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

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

O coeficiente binomial $\binom{n}{k}$ (pronunciado "n, escolhidos k a k") conta o número de subconjuntos com k elementos de um conjunto de tamanho n i.e., o número de formas de escolher k inteiros distintos do conjunto $\{1,\ldots,n\}$.

1.3.3 Notação assintótica

Ao analisar estruturas de dados neste livro, queremos discutir os tempos de execução de várias operações. O tempo exato de execução irá, é claro, variar de computador para computador e até entre diferentes execuções no mesmo computador. Ao discutirmos sobre o tempo de execução de uma operação estamos nos referindo ao número de instruções de computador executadas durante a operação. Mesmo para códigos simples, essa quantidade pode ser difícil de computar com exatidão. Portanto, em vez de analisar tempos de execução exatos, iremos usar a famosa *notação big-Oh*: Para uma função $f(n)$, $O(f(n))$ denota um conjunto de funções,

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

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

Geralmente usamos notação assintótica para simplificar funções. Por exemplo, em vez de usarmos $5n\log n+8n-200$ podemos escrever $O(n\log n)$. Isso é provado da seguinte forma:

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

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 $n_0 = 2$.

Vários atalhos 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 eles ainda valerão. Por exemplo, multiplicar por n chegamos a:

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

Continuando uma longa e notável tradição, iremos abusar dessa notação ao escrever coisas do tipo $f_1(n) = O(f(n))$ quando o que realmente queremos expressar é $f_1(n) \in O(f(n))$. Também iremos fazer afirmações do tipo "o tempo de execução dessa operação é $O(f(n))$" quando essa afirmação deveria ser na verdade "o tempo de execução dessa operação é *um membro de* $O(f(n))$." Esses atalhos servem principalmente para evitar frases estranhas e tornar mais fácil o uso de notação assintótica em manipulações sequenciais de equações.

A exemplo particularmente estranho disso ocorre quando escrevemos afirmações tipo

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

De novo, isso seria mais corretamente escrito da forma

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

A expressão $O(1)$ também traz à tona outra questão. Como não tem nenhuma variável nessa expressão, pode não ser claro qual variável está aumentando. Sem contexto, não tem como dizer.

No exemplo anterior, como a única variável no resto da equação é n, podemos assumir que isso deve ser lido como $T(n) = 2 \log n + O(f(n))$, onde $f(n) = 1$.

A notação Big-Oh não é nova nem exclusiva à Ciência da Computação. Ela foi usada pelo matemático especialista em Teoria dos Números Paul Bachmann desde pelo menos 1894 e é imensamente útil para descrever o tempo de execução de algoritmos de computadores. Considere o seguinte trecho de código:

Simple

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

Uma execução desse método envolve:

- 1 atribuição (`int i = 0`),
- $\mathsf{n}+1$ comparações ($\mathsf{i} < \mathsf{n}$),
- n incrementos ($\mathsf{i}++$),
- n cálculos de deslocamentos em array ($\mathsf{a}[\mathsf{i}]$), e
- n atribuições indiretas ($\mathsf{a}[\mathsf{i}] = \mathsf{i}$).

Então podemos escrever esse tempo de execução como

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

onde a, b, c, d e e são constantes que dependem da máquina rodando o código e que representam o tempo para realizar atribuições, comparações, incrementos, cálculos de deslocamento em array e atribuições indiretas, respectivamente. Entretanto, se essa expressão representa o tempo de execução de duas linhas de código, então claramente esse tipo de análise não será viável para códigos ou algoritmos complicados. Ao usar a notação big-Oh, o tempo de execução pode ser simplificado a

$$T(\mathsf{n}) = O(\mathsf{n}) \quad .$$

Isso não somente é mais compacto, mas também provê praticamente a mesma informação. O fato de que o tempo de execução depende das constantes a, b, c, d e e no exemplo anterior significa que, em geral, não será possível comparar dois tempos de execução para saber qual é mais rápido sem saber os valores dessas constantes. Mesmo se fizermos esforços para determinar essas constantes (digamos, usando testes de medição de tempo), então nossa conclusão será válida somente para a máquina na qual rodamos nossos testes.

Notação Big-Oh nos permite avaliar a situação a um nível mais alto, possibilitando a análise de funções mais complicadas. Se dois algoritmos têm o mesmo tempo Big-Oh, então não saberemos qual é mais rápido e que não pode não haver um ganhador em todas as situações. Um algoritmo pode ser mais rápido em uma máquina enquanto o outro em uma máquina diferente. Porém, se os dois algoritmos têm tempos de execução

big-Oh distintos, então teremos certeza que aquele com menor função Big-Oh será mais rápido *para valores* n *grandes o suficiente*.

Um exemplo de como a notação big-Oh nos permite comparar duas diferentes funções é mostrado na Figura 1.5, que compara a taxa de crescimento de $f_1(\mathsf{n}) = 15\mathsf{n}$ versus $f_2(n) = 2\mathsf{n}\log\mathsf{n}$. Hipoteticamente, $f_1(n)$ seria o tempo de execução de um complicado algoritmo de tempo linear enquanto $f_2(n)$ é o tempo de execução de um algoritmo bem mais simples baseado no paradigma de divisão e conquista. Isso exemplifica essa situação, embora $f_1(\mathsf{n})$ seja maior que $f_2(n)$ para valores baixos de n, o oposto é verdade para valores mais altos de n. Eventualmente $f_1(\mathsf{n})$ ganha e por uma margem crescente. Análise usando notação big-Oh nos indica que isso aconteceria, pois $O(\mathsf{n}) \subset O(\mathsf{n}\log\mathsf{n})$.

Em alguns casos, usaremos notação assintótica em funções com mais de uma variável. Pode não ser comum mas, para nossos objetivos, a seguinte definição é suficiente:

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

Essa definição captura a situação que mais nos importa: quando os argumentos $n_1,\ldots,n_k$ faz g assumir valores altos. Essa definição também está de acordo com a definição univariada de $O(f(n))$ quando $f(n)$ é uma função crescente de n. O leitor deve ficar atento ao fato de que, embora isso funcione para o objetivo deste livro, outros textos podem tratar funções multivariadas e notação assintótica diferentemente.

1.3.4 Randomização e Probabilidades

Algumas das estruturas de dados apresentadas neste livro são *randomizadas*[3]; eles fazem escolhas aleatórias que são independentes dos dados nelas armazenadas ou das operações realizadas neles. Por essa razão, realizar o mesmo conjunto de operações mais de uma vez usando essas estruturas pode resultar em tempos de execução variáveis. Ao analisar essas estruturas de dados estamos interessados no seu tempo de execução médio ou *esperado*.

[3] Randomização é um neologismo para indicar o uso de números aleatórios que influenciam na execução de um algoritmo

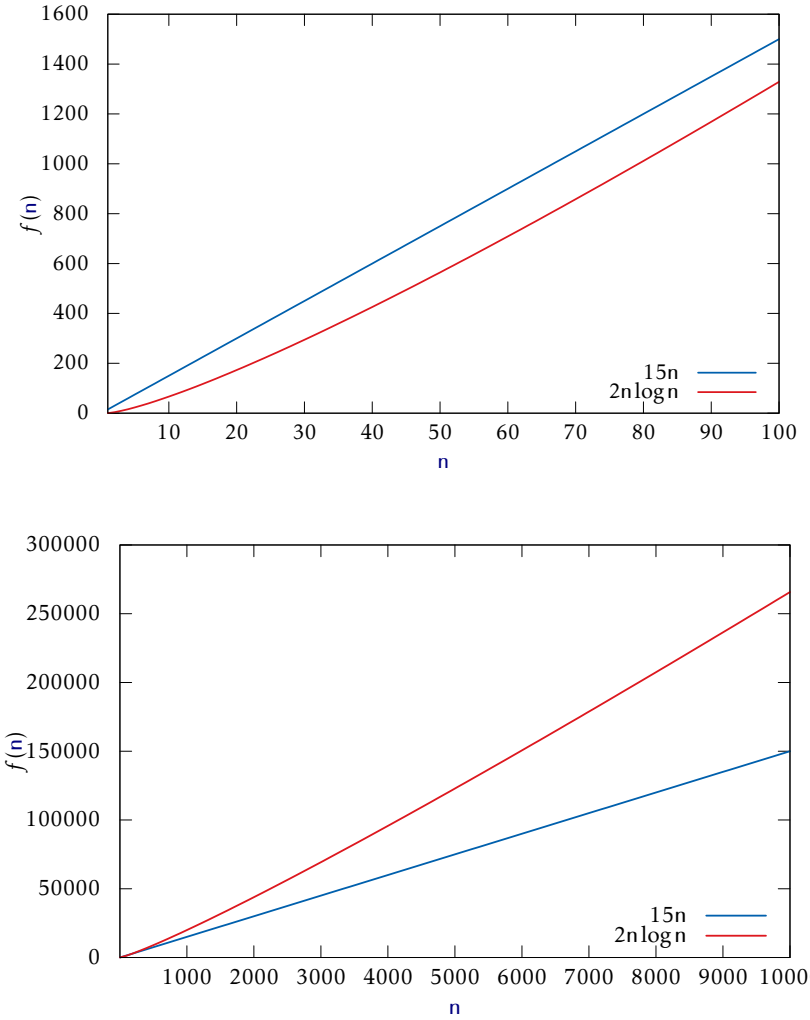


Figura 1.5: Comparação de $15n$ versus $2n \log n$.

Formalmente, o tempo de execução de uma operação em uma estrutura de dados randomizada é aleatório e queremos estudar o seu *valor esperado*; Para uma variável aleatória discreta X assumindo valores em um universo contável U, o valor esperado de X, denotado $\mathrm{E}[X]$, é dado pela fórmula

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

Aqui $\Pr\{\mathcal{E}\}$ denota a probabilidade que o evento $\mathcal{E}$ ocorre. Em todos os exemplos deste livro, essas probabilidades são somente em relação às escolhas aleatórias feitas pela estrutura de dados randomizada; não supomos que os dados armazenados na estrutura, nem a sequência de operações realizadas na estrutura de dados, seja aleatória.

Uma das propriedades mais importantes dos valores esperados é *linearidade da esperança*. Para duas variáveis aleatórias X e Y,

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

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

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

Linearidade da esperança nos permite quebrar variáveis aleatórias complicadas (como os lados esquerdos das equações acima) em somas de variáveis aleatória mais simples (lados direitos).

Um truque útil, que usaremos repetidamente, é definir *variáveis aleatórias indicadoras*. Essas variáveis binárias são úteis quando queremos contar algo e são melhor ilustradas por um exemplo. Suponha que lancemos uma moeda honesta k vezes e que queremos saber o número esperado de vezes que o lado cara aparece. Intuitivamente, sabemos que a resposta é $k/2$, mas se tentarmos provar usando a definição de valor esperado, temos

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

$$
\begin{aligned}
&= k \cdot \sum_{i=0}^{k-1} \binom{k-1}{i} / 2^k \\
&= k/2 \ .
\end{aligned}
$$

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

Usando variáveis indicadoras e linearidade da esperança torna esse trabalho bem mais fácil. Para cada $i \in \{1, \ldots, k\}$, defina a variável aleatória indicadora

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

Então

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

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

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

Esse caminho é mais longo, mas não exige que saibamos identidades mágicas ou que obtenhamos expressões não triviais de probabilidades. Melhor ainda, ele vai de encontro à intuição que temos sobre metade das moedas saírem cara precisamente porque cada moeda individual sai cara com probabilidade 1/2.

1.4 O Modelo de Computação

Neste livro, iremos analisar o tempo teórico de execução das operações das estruturas de dados que estudamos. Para fazer precisamente isso,

precisamos um modelo matemático de computação. Para isso, usamos o modelo w-*bit word-RAM*. RAM é uma sigla para Random Access Machine — Máquina de Acesso Aleatório. Nesse modelo, temos acesso a uma memória de acesso aleatório consistindo de *células*, cada qual armazena uma w-bit *word*, ou seja, uma palavra com w bits de memória. Isso implica que uma célula de memória pode representar, por exemplo, qualquer inteiro no conjunto $\{0, \ldots, 2^w - 1\}$.

No modelo word-RAM, operações básicas em words levam tempo constante. Isso inclui operações aritméticas ($+, -, *, /, \%$), comparações ($<, >, =, \leq, \geq$) e operações booleanas bit-a-bit (AND, OR e OR exclusivo bit-a-bit).

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

O tamanho da word w é um parâmetro muito importante desse modelo. A única premissa que faremos sobre w é um limitante inferior $w \geq \log n$, onde n é o número de elementos guardados em qualquer estrutura de dados. Essa premissa é bem razoável, pois caso contrário uma word não seria grande o suficiente para contar o número de elementos guardados na estrutura de dados.

Espaço é medido em words, de forma que quando falarmos sobre a quantidade de espaço usado por uma estrutura de dados, estamos nos referindo ao número de words de memória usada pela estrutura. Todas as nossas estruturas de dados guardam valores de um tipo genérico T, e nós presumimos que um elemento de tipo T ocupa uma word de memória. (Na realidade, estamos guardando referências para objetos do tipo T, e essas referências ocupam somente uma word de memória.)

O modelo word-RAM com w bits é bem próximo à Java Virtual Machine (JVM) de 32 bits quando $w = 32$. As estruturas de dados apresentadas neste livro não usam truques especiais que não são implementáveis na JVM ou na maior parte das arquiteturas.

1.5 Corretude, Complexidade de Tempo e Complexidade de Espaço

Ao estudar o desempenho de um estrutura de dados, existem três coisas mais importantes:

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

Complexidade de tempo: Os tempos de execução de operações na estrutura de dados deve ser o menor possível.

Complexidade de espaço: A estrutura de dados deve usar o mínimo de memória possível.

Neste texto introdutório, consideraremos a corretude como presumida; não consideramos estruturas de dados que dão respostas incorretas a consultas ou que não realizam atualizações corretamente. Iremos, entretanto, ver estruturas de dados que fazem um esforço extra para manter uso de espaço a um mínimo. Isso não irá em geral afetar o tempo (assintótico) de execução de operações, mas pode fazer as estruturas de dados um pouco mais lentas na prática.

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

Tempos de execução no pior caso: Esse é o tipo mais forte de garantia de tempo de execução. Se uma operação de uma estrutura de dados tem tempo de execução no pior caso de $f(n)$, então uma dessas operações *nunca* leva mais tempo que $f(n)$.

Tempo de execução amortizado: Se dizemos 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 tempo de execução amortizado de $f(n)$, então uma sequência de m operações leva de tempo, no máximo, $mf(n)$. Algumas operações individuais podem levar mais tempo que $f(n)$ mas o tempo médio, sobre a sequência inteira de operações, é até $f(n)$.

Tempo de Execução Esperado: Se dizemos que o tempo de esperado de execução de uma operação em uma estrutura de dados é $f(n)$, isso significa que o tempo real de execução é uma variável aleatória (veja a Seção 1.3.4) e o valor esperado dessa variável aleatória é no máximo $f(n)$. A randomização aqui é em respeito às escolhas aleatórias feitas pela estrutura de dados.

Para entender a diferença entre o pior caso, o tempo amortizado e o tempo esperado, ajuda se considerarmos um exemplo financeiro. Considere o custo de comprar uma casa:

Pior caso versus custo amortizado: Considere uma casa que o dono está vendendo por R$120 000. A fim de comprar essa casa, podemos pegar um empréstimo de 120 meses (10 anos) com pagamentos mensais de R$1 200. Nesse caso, o custo mensal no pior caso de pagar o empréstimo é R$1 200.

Se temos suficiente dinheiro em mãos, podemos escolher comprar a casa sem empréstimo, com um pagamento de R$120 000. Nesse caso, para o período de 10 anos, o custo amortizado mensal de comprar essa casa é

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

Esse valor é bem menor que os R$1 200 mensais que teríamos que pagar se pegássemos o empréstimo.

Pior caso versus custo esperado: A seguir, considere o caso de um seguro de incêndio na nossa casa de R$120 000. Por analisar centenas de milhares de casos, seguradoras têm determinado que a quantidade esperada de danos por incêndios causado a uma casa como a nossa é de R$10 por mês. Esse é uma valor bem baixo, pois a maior parte das casa nunca pegam fogo, algumas poucas tem incêndios pequenos que causam algum dano e um número minúsculo de casas queimam até as cinzas. Baseada nessa informação, a seguradora cobra R$15 mensais para a contratação de um seguro.

Agora é o momento da decisão. Devemos pagar os R$15 referentes ao custo de pior caso mensalmente para o seguro ou devemos apostar fazer uma poupança anti-incêndio ao custo esperado de R$10 mensal? Claramente, R$10 por mês custa menos *em expectativa*, mas temos que

aceitar que a possibilidade de *custo real* possa ser bem maior. No evento improvável em que a casa inteira queime, o custo real será de R$120 000.

Esses exemplos financeiros também oferecem a oportunidade de vermos porque às vezes aceitamos um tempo de execução amortizado ou esperado em vez de considerarmos o tempo de execução no pior caso. Frequentemente é possível obter um tempo de execução esperado ou amortizado menor que o obtido no pior caso. No mínimo, frequentemente é possível obter uma estrutura de dados bem mais simples se estivermos dispostos a aceitar tempo de execução amortizado ou esperado.

1.6 Trechos de Código

Os trechos de código neste livro são escritos na linguagem Java. Entretanto, para fazer o livro acessível para leitores não familiares com todas as construções e *keywords* definidas pela linguagem Java, os trechos de código foram simplificados. Por exemplo, um leitor não irá encontrar keywords como `public`, `protected`, `private` ou `static`. O leitor também não encontrará discussão aprofundada sobre hierarquia de classes. Quais interfaces uma determinada classes implementa ou qual classes ela estende, se relevante para a discussão, deve estar claro do texto do contexto em questão.

Essas convenções deve fazer os trechos de código mais fáceis de entender por qualquer um com background linguagens que variantes da linha ALGOL, incluindo B, C, C++, C#, Objective-C, D, Java, JavaScript e assim por diante. Leitores em busca de detalhes das implementações são encorajados a olhar no código fonte da linguagem Java que acompanha este livro.

Este livro mistura análise matemática de tempos de execução com o código-fonte da linguagem Java para os algoritmos em análise. Isso significa que algumas equações contêm variáveis também encontradas no código-fonte.

Essas variáveis são formatadas consistentemente, tanto dentro do código fonte quanto nas equações. A variável mais comum desse tipo é a variável `n` que, sem exceção, sempre refere-se ao número de itens atualmente armazenados na estrutura de dados.

Implementações de List			
	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} Denota um tempo de execução *amortizado*.
^{E} Denota um tempo de execução *esperado*.

Tabela 1.1: Resumo das implementações de List e USet.

1.7 Lista de Estruturas de Dados

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

1.8 Discussão e Exercícios

As interfaces List, USet e SSet descritas na Seção 1.2 são influenciadas pela Java Collections Framework [54]. Elas são essencialmente ver-

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
BTree	$O(\log n)$	$O(B+\log n)^{A}$	§ 14.2
BTree[X]	$O(\log_B n)$	$O(\log_B n)$	§ 14.2

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

[I] Essa estrutura pode guardar somente dados inteiros de w-bit.

[X] Isso denota o tempo de execução no modelo de memória externa; veja o Capítulo 14.

Tabela 1.2: Resumo de implementações de SSet e Queue de prioridades.

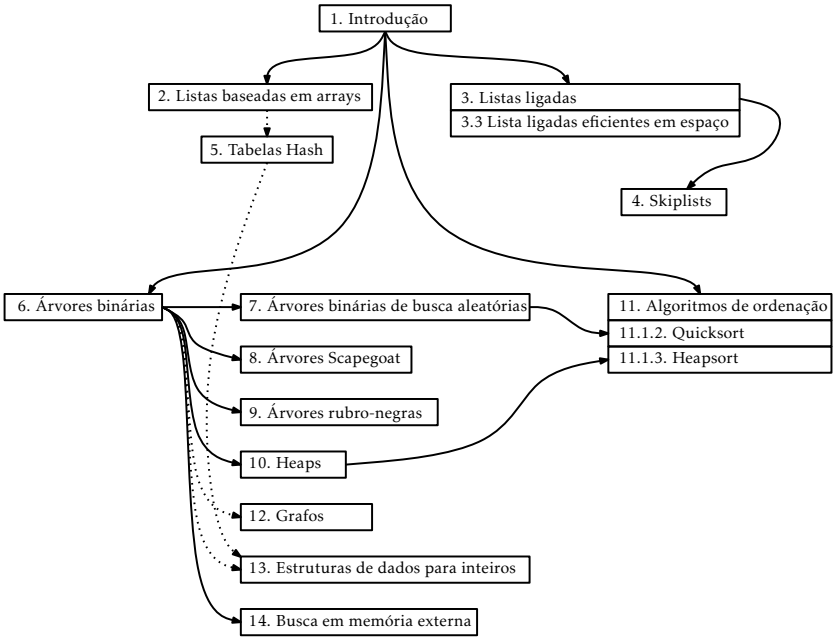


Figura 1.6: Dependências entre capítulos deste livro.

sões simplificadas das interfaces `List`, `Set`, `Map`, `SortedSet` e `SortedMap` encontradas no Java Collections Framework. O código-fonte que acompanha este livro inclui classes auxiliares para fazer das implementações `USet` e `SSet` também implementações de `Set`, `Map`, `SortedSet` e `SortedMap`.

Para um soberbo (e livre) tratamento da matemática discutida neste capítulo, incluindo notação assintótica, logaritmos, fatoriais, aproximação de Sterling, probabilidade básica e muito mais, veja o livro-texto por Leyman, Leighton e Meyer [50]. Para um texto de cálculo suave que inclui definições formais de exponenciais e logaritmos, veja o (disponível livremente) texto clássico de Thompson [73].

Para maiores informações sobre probabilidade básica, especialmente como ela se relaciona com Ciência da Computação, veja o livro didático de Ross [65]. Outra boa referência, que cobre notação assintótica e probabilidades, é o livro-texto de Graham, Knuth e Patashnik [37].

Leitores que desejam afinar suas habilidades de programação Java podem achar muitos tutoriais online em [56].

Exercício 1.1. Este exercício é planejado para auxiliar o leitor a se familiarizar na escolha da estrutura de dados certa para o problema correto. Se implementado, partes deste exercício devem ser realizadas usando uma implementação da interface relevante (`Stack`, `Queue`, `Deque`, `USet` ou `SSet`) providas pela Java Collections Framework.

Resolva os seguintes problemas fazendo a leitura de um arquivo de texto uma linha por vez e executando operações em cada linha nas estrutura(s) de dados adequada(s). Suas implementações devem ser rápidas o suficiente tal que até arquivos com um milhão de linhas podem ser processadas em alguns segundos.

1. Leia a entrada uma linha por vez e então escreva na tela as linha em ordem invertida, tal que a última linha lida é a primeira mostrada, e então a penúltima lida é a segunda a ser mostrada e assim por diante.

2. Leia as primeiras 50 linhas da entrada e então as escreva na saída em ordem reversa. Leia as seguintes 50 linhas e então escreva na saída em ordem reversa. Faça isso até que não haja mais linhas a

serem linhas. Nesse ponto as linhas restantes lidas também devem ser mostradas em ordem invertida. Em outras palavras, sua saída iniciará com a 50ª linha, então 49ª linha, então a 48ª e assim até a primeira linha. Isso deverá ser seguido pela centésima linha, seguida pela 99ª até a 51ª linha. Assim por diante. O seu código nunca deverá manter mais de 50 linhas a qualquer momento.

3. Leia a entrada uma linha por vez. A qualquer momento após ler as primeiras 42 linhas, se alguma linha está em branco (i.e., uma string de comprimento 0), então produza a linha que ocorreu 42 linhas antes dela. Por exemplo, se Linha 242 está em branco, então o seu programa deve mostrar a linha 200. Esse programa deve ser implementado tal que ele nunca guarda mais que 43 linhas da entrada a qualquer momento.

4. Leia a entrada uma linha por vez e mostre cada linha se não for uma duplicata de alguma linha anterior. Tenha cuidado especial para que um arquivo com muitas linhas duplicadas não use mais memória do que é necessário para o número de linhas únicas.

5. Leia a entrada uma linha por vez e mostre na tela cada linha somente se você já encontrou uma linha igual antes. (O resultado final é que você remove a primeira ocorrência de cada linha.). Tenha cuidado para que um arquivo com muitas linhas duplicadas não use mais memória do que seja necessário para o número de linhas únicas.

6. Leia a entrada uma linha por vez. Então, mostre na tela todas as linhas ordenadas por comprimento, com as linhas mais curtas primeiro. No caso em que duas linhas tem o mesmo tamanho, decida sua ordem usando a ordem usual de texto. Linhas duplicadas devem ser mostradas somente uma vez.

7. Faça o mesmo que a questão anterior exceto que linhas duplicadas devem ser mostradas o mesmo número de vezes que elas aparecem na entrada.

8. Leia a entrada uma linha por vez e então mostre na tela as linhas

pares (começando com a primeira linha, linha 0) seguidas das linhas ímpares.

9. Leia a entrada inteira uma linha por vez e aleatoriamente troque as linhas antes de mostrá-las. Para ser claro: você não deve mudar o conteúdo de qualquer linha. Em vez disso, a mesma coleção de linhas deve ser mostrada, mas em ordem aleatória.

Exercício 1.2. Uma *palavra Dyck* é uma sequência de +1s e -1s com propriedade de que soma de qualquer prefixo da sequência nunca é negativa. Por exemplo, $+1, -1, +1, -1$ é uma palavra Dyck, mas $+1, -1, -1, +1$ não é uma palavra Dyck pois o prefixo $+1-1-1<0$. Descreva qualquer relação entre palavras Dyck e as operações da interface `Stack`, `push(x)` e `pop()`.

Exercício 1.3. Uma *string pareada* é uma sequência de caracteres {, }, (,), [, e] que são apropriadamente pareados. Por exemplo, "{{()[]}}" é uma string pareada, mas esta "{{()]}" não é, uma vez que a segunda { é pareada]. Mostre como usar uma stack para que, dada uma string de comprimento `n`, você possa determinar se é uma string pareada em tempo $O(n)$.

Exercício 1.4. Suponha que você tem uma `Stack`, `s`, que aceita somente as operações `push(x)` e `pop()`. Mostre como, usando somente uma `Queue` FIFO, `q`, você pode inverter a ordem de todos os elementos em `s`.

Exercício 1.5. Usando `USet`, implemente uma Bag. Uma Bag é como um `USet`—ele aceita os métodos `add(x)`, `remove(x)` e `find(x)` — mas ele aceita armazenar elementos duplicados. A operação `find(x)` em uma Bag retorna algum elemento (se tiver) que é igual a `x`. Em adição, uma Bag aceita a operação `findAll(x)` que retorna uma lista de todos os elementos na Bag que são iguais a `x`.

Exercício 1.6. A partir do zero, escreva e teste implementações das interfaces `List`, `USet` e `SSet`. Essas implementações não precisam ser eficientes. Elas podem ser usadas depois para testar a corretude e desempenho de implementações mais eficientes. (O jeito mais fácil de fazer isso é guardar os elementos em um array.)

Exercício 1.7. Trabalhe para melhorar o desempenho das suas implementações da questão anterior usando quaisquer truques que puder imaginar. Experimente e pense como você pode melhorar o desempenho de

`add(i,x)` e `remove(i)` na sua implementação de `List`. Pense como você poderia melhorar o desempenho da operação `find(x)` na suas implementações dos `USet` e `SSet`. Este exercício é tem o intuito de te dar uma ideia da dificuldade de conseguir implementações eficientes dessas interfaces.

Capítulo 2

Listas Baseadas em Arrays

Neste capítulo, iremos estudar implementações das interfaces `List` e `Queue`, onde os dados são armazenados em um array chamado de *backing array* ou *array de apoio*. A tabela a seguir resume o tempo de execução de operações para estruturas de dados apresentadas nestes 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 em um único array têm muitas vantagens e limitações em comum:

- Arrays permitem acesso em tempo constante a qualquer valor no array. Isso é o que permite `get(i)` e `set(i,x)` rodarem 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 elemento no array precisam ser deslocados para abrir espaço para o elemento recentemente adicionado ou preencher a lacuna criada pelo elemento removido. Essa é a razão pela qual as operações `add(i,x)` e `remove(i)` têm tempos de execução que dependem de `n` e `i`.

- Arrays não podem expandir ou encolher por si só. Quando o número de elementos na estrutura de dados excede o tamanho do array de apoio, um novo array precisa ser alocado e os dados do array antigo precisa ser copiado no novo array. Essa é uma operação cara.

Um terceiro ponto é importante. Os tempos de execução citados na tabela acima não incluem o custo associado com expandir ou encolher o array de apoio. Veremos que, se não gerenciado com cuidado, o custo de expandir ou encolher o array de apoio não aumenta muito o custo *médio* de uma operação. Mais precisamente, se iniciarmos com um estrutura de dados vazia e realizarmos qualquer sequência de m operações `add(i,x)` ou `remove(i)` , então o custo total de expandir e encolher o array de apoio, sobre a sequência inteira de m operações é $O(m)$. Embora algumas operações individuais sejam mais caras, o custo amortizado, quando dividido por todas as m operações, é somente $O(1)$ por operação.

2.1 ArrayStack: Operações de Stack Rápida Usando um Array

Um `ArrayStack` implementa a interface lista usando um array `a`, chamado de *array de apoio*. O elemento da lista com índice `i` é armazenado em `a[i]`. Na maior parte do tempo, `a` é maior que o estritamente necessário, então um inteiro `n` é usado para registrar o número de elementos realmente armazenados em `a`. Dessa maneira, os elementos da lista são guardados em `a[0]`,...,`a[n-1]` e, sempre, `a.length` $\geq$ `n`.

ArrayStack

```
T[] a;
int n;
int size() {
  return n;
}
```

2.1.1 O Básico

Acessar e modificar os elementos de uma ArrayStack usando get(i) e set(i,x) é trivial. Após realizar as verificações de limites necessárias, 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 estão ilustradas na Figura 2.1. Para implementar a operação add(i,x), primeiro verificamos se a está cheio. Caso positivo, chamamos o método resize() para aumentar o tamanho de a. Como resize() é implementado será discutido depois. Por ora, é suficiente saber que, após uma chamada para resize(), temos certeza que a.length > n. Com isso resolvido, deslocamos os elementos a[i],...,a[n − 1] para uma posição à direita para abrir espaço para x, atribuir a[i] igual a x e incrementar 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 de uma potencial chamada a resize(), então o custo da operação add(i,x) é proporcional ao número de elementos que temos que deslocar para abrir espaço para x. Portanto o custo dessa operação (ignorando o custo de redimensionar a) é $O(n - i)$.

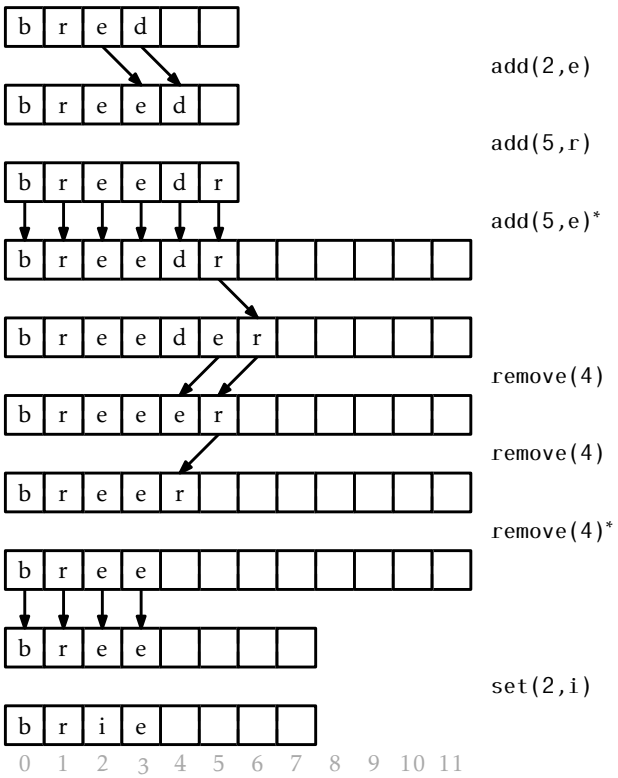


Figura 2.1: Uma sequência de operações `add(i,x)` e `remove(i)` em um `ArrayStack`. Flechas denotam elementos sendo copiados. Operações que resultam em uma chamada para `resize()` são marcados com um asterisco.

Implementar a operação remove(i) é similar. Desloca-se os elementos $a[i+1], \ldots, a[n-1]$ à esquerda por uma posição (sobrescrevendo $a[i]$) e decrementa-se o valor de n. Após fazer isso, verificamos se n é muito menor que a.length ao verificar se a.length $\geq 3n$. Caso positivo, então 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++)
    a[j] = a[j+1];
  n--;
  if (a.length >= 3*n) resize();
  return x;
}
```

Ignorando 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 Expansão e Redução

O método resize() é razoavelmente simples; ele aloca um novo array b de tamanho $2n$ e copia os n elementos de a nas primeiras n posições em b e então atribui a em b. Após isso, faz uma chamada a resize(), a.length $= 2n$.

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

Analisar o custo real da operação resize() é fácil. Ela aloca um array b de tamanho $2n$ e copia os n elementos de a em b. Isso leva tempo $O(n)$.

A análise de tempo de execução da seção anterior ignorou o custo de chamadas a resize(). Nesta seção analisaremos esse custo usando uma

técnica chamada de *análise amortizada*. Essa técnica não tenta determinar o custo de redimensionar o array durante cada operação add(i,x) e remove(i). Em vez disso, ela considera o custo de todas as chamadas a resize() durante a sequência de m chamadas a add(i,x) ou remove(i). Em particular, mostraremos que:

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

Demonstração. Mostraremos que em qualquer momento que resize() é chamada, o número de chamadas a add() ou remove() desde a última chamada a resize() é pelo menos $\mathtt{n}/2 - 1$. Portanto, se $\mathtt{n}_i$ 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

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

o que é equivalente a

$$\sum_{i=1}^{r} \mathtt{n}_i \leq 2m + 2r \ .$$

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

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

uma vez que r não é maior que m. O que resta é mostrar que 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 $\mathtt{n}_i/2$.

Há dois casos a considerar. No primeiro caso, resize() está sendo chamado por add(i,x) pois o array de apoio a está cheio, i.e., a.length = n = $\mathtt{n}_i$. Considere a chamada anterior a resize(): após essa chamada prévia, o tamanho de a era a.length, mas o número de elementos guardados em a era no máximo a.length/2 = $\mathtt{n}_i/2$. Mas agora o número de elementos guardados em a é $\mathtt{n}_i$ = a.length, então deve ter havido pelo menos $\mathtt{n}_i/2$ chamadas a add(i,x) desde a chamada anterior a resize().

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

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

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

2.1.3 Resumo

O teorema a seguir resume o desempenho de uma ArrayStack:

Teorema 2.1. *Uma* ArrayStack *implementa a interface* List*. Ignorando o custo de chamadas a* resize()*, uma* ArrayStack *possui as operações*

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

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

O ArrayStack é um jeito eficiente de implementar a Stack. Em especial, podemos implementar push(x) como add(n, x) e pop() como remove(n − 1) e nesse caso essas operações rodarão em tempo $O(1)$ amortizado.

[1] O -1 nessa fórmula inclui o caso especial que ocorre quando n = 0 e a.length = 1.

2.2 FastArrayStack: Uma ArrayStack otimizada

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

FastArrayStack

```
void resize() {
  T[] b = newArray(max(2*n,1));
  System.arraycopy(a, 0, b, 0, n);
  a = b;
}
void add(int i, T x) {
  if (n + 1 > a.length) resize();
  System.arraycopy(a, i, a, i+1, n-i);
  a[i] = x;
  n++;
}
T remove(int i) {
  T x = a[i];
  System.arraycopy(a, i+1, a, i, n-i-1);
  n--;
  if (a.length >= 3*n) resize();
  return x;
}
```

Essas funções são, em geral, altamente otimizadas e podem usar até mesmo instruções de máquina especiais que podem fazer esse cópia muito mais rápida do que poderíamos usando um laço for. Embora o uso dessas funções não diminuam o tempo de execução assintoticamente falando, pode ser uma otimização que vale a pena.

Nas implementações em Java deste livro, o uso do método disponível nativamente System.arraycopy(s,i,d,j,n) resultaram em um fator de

aceleração, *speedup*, entre 2 e 3, dependendo dos tipos de operações realizadas. O resultados podem variar de acordo com o caso.

2.3 ArrayQueue: Uma Queue Baseada Em Array

Nesta seção apresentamos a estrutura de dados ArrayQueue, que implementa uma queue do tipo FIFO (first-in-first-out, primeiro-que-chega-primeiro-que-sai); elementos são removidos (usando a operação remove()) da queue na mesma ordem em que são adicionados (usando a operação add(x)).

Note que uma ArrayStack é uma escolha ruim para uma implementação de uma queue do tipo FIFO. Não é uma boa escolha porque precisamos escolher um lado da lista ao qual adicionaremos elementos e então remover elementos do outro lado. Uma das duas operações precisa trabalhar na cabeça da lista, o que envolve chamar add(i, x) ou remove(i) com um valor de $i = 0$. Isso resulta em um tempo de execução proporcional a n.

Para obter uma implementação de queue eficiente baseada em array, primeiro observamos que o problema seria fácil se tivéssemos um array a infinito. Poderíamos manter um índice j que guarda qual é o próximo elemento a remover e um inteiro n que conta o número de elementos na queue. Os elementos da queue sempre seriam guardados em

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

Inicialmente, ambos j e n receberiam o valor 0. Para adicionar um elemento, teríamos que colocá-lo em $a[j+n]$ e incrementar n. Para remover um elemento, o removeríamos de $a[j]$, incrementando j, e decrementando n.

É claro, o problema com essa solução é que ela requer um array infinito. Um ArrayQueue simula isso ao usar um array finito a e *aritmética modular*. Esse é o tipo de aritmética usada quando estamos falando sobre a hora do dia. Por exemplo, 10:00 mais cinco horas resulta em 3:00. Formalmente, dizemos que

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

Lemos a última parte dessa equação da forma "15 é congruente a 3 módulo 12." Podemos também tratar mod como um operador binário, tal que

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

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

Aritmética modular é útil para simular um array infinito uma vez que `i` mod `a.length` sempre resulta em um valor no intervalo $0, \ldots,$ `a.length` $- 1$. Usando aritmética modular podemos guardar os elementos da queue em posições do array

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

Desse jeito trata-se o array `a` como um *array circular* no qual os índices do array maiores que `a.length` $- 1$ "dão a volta" ao começo do array.

A única coisa que falta é cuidar para que o número de elementos no `ArrayQueue` não ultrapasse o tamanho de `a`.

ArrayQueue
```
T[] a;
int j;
int n;
```

A sequência de operações `add(x)` e `remove()` em um `ArrayQueue` é ilustrado na Figura 2.2. Para implementar `add(x)`, primeiro verificamos se `a` está cheio e, se necessário, chamamos `resize()` para aumentar o tamanho de `a`. Em seguida, guardamos `x` em `a[(j+n)%a.length]` e incrementamos `n`.

ArrayQueue
```
boolean add(T x) {
  if (n + 1 > a.length) resize();
```

[2] Isso às vezes é chamado de operador mod com *morte cerebral*, pois não implementa corretamente o operador matemático mod quando o primeiro argumento é negativo.

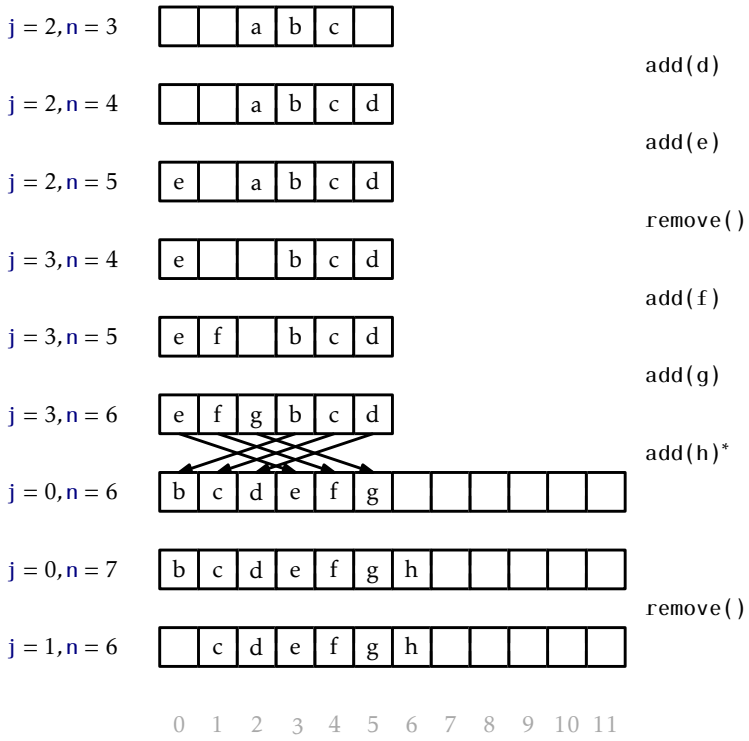


Figura 2.2: Uma sequência de operações `add(x)` e `remove(i)` em um `ArrayQueue`. Flechas denotam elementos sendo copiados. Operações que resultam em uma chamada a `resize()` são marcadas com um asterisco.

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

Para implementar remove(), primeiro guardamos a[j] para que possamos reutilizá-lo depois. A seguir, decrementamos n e incrementamos j (módulo a.length) ao atribuir $j = (j+1) \bmod a.length$. Finalmente, retornamos o valor guardado de a[j]. Se necessário, podemos chamar resize() para diminuir o tamanho de a.

ArrayQueue

```
T remove() {
  if (n == 0) throw new NoSuchElementException();
  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() do ArrayStack. Ela aloca um novo array, b, de tamanho 2n e copia

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

em

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

e atribui $j = 0$.

ArrayQueue

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

2.3.1 Resumo

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

Teorema 2.2. *Um* ArrayQueue *implementa a interface (FIFO)* Queue*. Ignorando o custo das chamadas a* resize()*, uma* ArrayQueue *aceita as operações* add(x) *e* remove() *em tempo* $O(1)$ *por operação. Além disso, ao começar com um* ArrayQueue *vazio, qualquer sequência de* m *operações* add(i,x) *e* remove(i) *resulta em um total de tempo* $O(m)$ *gasto durante todas as chamadas a* resize()*.*

2.4 ArrayDeque: Operações Rápidas para Deque Usando um Array

O ArrayQueue da seção anterior é uma estrutura de dados para representação de sequências que nos permite eficientemente adicionar a um lado da sequência e remover do outro.

A estrutura de dados ArrayDeque permite a edição e remoção eficiente em ambos lados. Essa estrutura implementa a interface List ao usar a mesma técnica de array circular usada para representar um ArrayQueue.

ArrayDeque

```
T[] a;
int j;
int n;
```

As operações get(i) e set(i,x) em um ArrayDeque são simples . Elas obtém ou atribui a um 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;
```

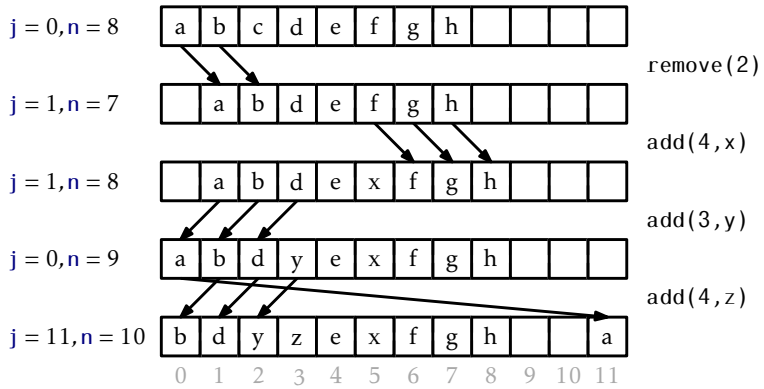


Figura 2.3: Uma sequência de operações add(i,x) e remove(i) em um ArrayDeque. Flechas denotam elementos sendo copiados.

```
  return y;
}
```

A implementação de add(i,x) é um pouco mais interessante. Como sempre, primeiro verificamos se a está cheio e, se necessário, chamados resize() para redimensionar a. Lembre-se que queremos que essa operação seja rápida quando i for pequeno (perto de 0) ou quando i é grande (perto de n). Portanto, verificamos se $i < n/2$. Caso positivo, deslocamos os elementos $a[0],\ldots,a[i-1]$ à esquerda por uma posição. Caso contrário, ($i \geq n/2$), deslocamos os elementos $a[i],\ldots,a[n-1]$ à direita por uma posição. Veja a Figura 2.3 para uma representação 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; //(j-1)mod a.length
    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 deslocar dessa maneira, nós garantimos que add(i,x) nunca tem que deslocar mais de $\min\{i, n-i\}$ elementos. Então, 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 similar. Ela desloca elementos $a[0],\ldots,a[i-1]$ à direita em uma posição ou desloca elementos $a[i+1],\ldots,a[n-1]$ à esquerda em uma posição dependendo se $i < n/2$. Novamente, isso significa que remove(i) nunca gasta mais de $O(1+\min\{i, n-i\})$ tempo 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 teorema a seguir resume o desempenho da estrutura de dados ArrayDeque:

Teorema 2.3. *Uma* ArrayDeque *implementa a interface* List*. Ignorando o custo de chamadas a* resize(), *um* ArrayDeque *aceita as operações*

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

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

2.5 DualArrayDeque: Construção de um Deque a Partir de Duas Stacks

A seguir, apresentamos uma estrutura de dados, o `DualArrayDeque`, que tem o mesmo desempenho que um `ArrayDeque` ao usar duas `ArrayStacks`. Embora o desempenho assintótico do `DualArrayDeque` não seja melhor que do `ArrayDeque`, ainda vale estudá-lo , pois oferece um bom exemplo de como obter uma estrutura de dados mais sofisticada pela combinação de duas estruturas de dados mais simples.

Um `DualArrayDeque` representa uma lista usando duas `ArrayStacks`. Relembre-se que um `ArrayStack` é rápido quando as operações dele modificam elementos perto do final. Uma `DualArrayDeque` posiciona duas `ArrayStacks`, chamadas de `frontal` e `traseira`, de modo complementar para que as operações sejam rápidas em ambas as direções.

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

Um `DualArrayDeque` não guarda explicitamente o número, `n`, de elementos contidos. Ele não precisa, pois ele contém `n = front.size() + back.size()` elementos. De qualquer forma, ao analisar o `DualArrayDeque` iremos usar ainda o `n` para denotar o número de elementos nele.

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

A ArrayStack front guarda os elementos da lista cujos índices são $0,\ldots,$ front.size() -1, mas guarda-os em ordem reversa. O ArrayStack back contém elementos da lista com índices em front.size(),...,size()−1 na ordem normal. Desse jeito, get(i) e set(i,x) traduzem-se em chamadas apropriadas para get(i) ou set(i,x) e ambos front ou back, que levam tempo $O(1)$ time 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, pois os elementos de front são guardados em ordem inversa.

A adição e a remoção de elementos de uma DualArrayDeque são ilustradas na Figura 2.4. A operação add(i,x) manipula 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();
```

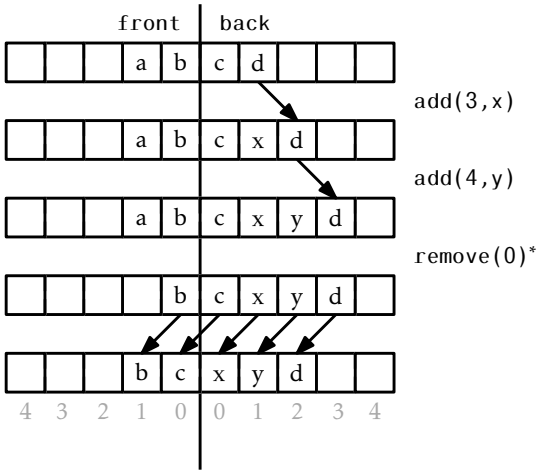


Figura 2.4: Uma sequência de operações add(i,x) e remove(i) em um DualArrayDeque. Flechas denotam elementos sendo copiados. Operações que resultam em um rebalanceamento por balance() são marcados com um asterisco.

```
}
```

O método add(i,x) realiza o balanceamento das duas ArrayStacks front e back, ao chamar o método balance(). A seguir descrevemos a implementação de balance(), mas no momento é suficiente saber que balance() garante que, a não ser que size() < 2, front.size() e back.size() não diferem por mais de um fator de 3. Em especial, $3 \cdot$ front.size() $\geq$ back.size() e $3 \cdot$ back.size() $\geq$ front.size().

A seguir, nós analisamos o custo de add(i,x), ignorando o custo de chamadas balance(). Se i < front.size(), então add(i,x) é implementada pela chamada a front.add(front.size() − i − 1, x). Como front é uma ArrayStack, o custo disso é

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

Por outro lado, se i $\geq$ front.size(), então add(i,x) é implementada como back.add(i − front.size(), x). O custo disso é

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

Note que o primeiro caso (2.1) ocorre quando $i < n/4$. O segundo caso (2.2) ocorre quando $i \geq 3n/4$. Quando $n/4 \leq i < 3n/4$, não temos certeza se a operação afeta `front` ou `back`, mas nos dois casos, a operação leva tempo $O(n) = O(i) = O(n-i)$, pois $i \geq n/4$ e $n - i > n/4$. Resumindo a situação, temos

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

Então, o tempo de operação de `add(i,x)`, se ignorarmos o custo à chamada `balance()`, é $O(1+\min\{i, n-i\})$.

A operação `remove(i)` e sua análise lembra a 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, passamos à operação `balance()` usada por `add(i,x)` e `remove(i)`. Essa operação garante que nem `front` nem `back` se tornem grandes demais (ou pequenos demais). Ela garante que, ao menos que haja menos de dois elementos, o `front` e o `back` possuem $n/4$ elementos cada. Se esse não for o caso, então ele move elementos entre elas de tal forma que `front` e `back` contenham exatamente $\lfloor n/2 \rfloor$ elementos e $\lceil n/2 \rceil$ elementos, respectivamente.

DualArrayDeque
```
void balance() {
  int n = size();
```

```
  if (3*front.size() < back.size()) {
    int s = n/2 - front.size();
    List<T> l1 = newStack();
    List<T> l2 = newStack();
    l1.addAll(back.subList(0,s));
    Collections.reverse(l1);
    l1.addAll(front);
    l2.addAll(back.subList(s, back.size()));
    front = l1;
    back = l2;
  } else if (3*back.size() < front.size()) {
    int s = front.size() - n/2;
    List<T> l1 = newStack();
    List<T> l2 = newStack();
    l1.addAll(front.subList(s, front.size()));
    l2.addAll(front.subList(0, s));
    Collections.reverse(l2);
    l2.addAll(back);
    front = l1;
    back = l2;
  }
}
```

Aqui há pouco para analisar. Se a operação `balance()` faz rebalanceamento , então ela move $O(n)$ elementos e isso leva $O(n)$ de tempo. Isso é ruim, pois `balance()` é chamada a cada operação `add(i,x)` e `remove(i)`. Porém, o lema a seguir mostra que na média, `balance()` gasta somente um tempo constante por operação.

Lema 2.2. *Se uma* `DualArrayDeque` *vazia é criada e qualquer sequência de chamadas* $m \geq 1$ *a* `add(i,x)` *e* `remove(i)` *são realizadas, então o tempo total gasto durante todas as chamadas a* `balance()` *é* $O(m)$.

Demonstração. Iremos 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 elementos foram deslocador por `balance()` é pelo menos $n/2 - 1$. Assim como na prova do Lema 2.1, é suficiente provar que o tempo total gasto por `balance()` é $O(m)$.

Iremos realizar a nossa análise usando uma técnica conhecida como o *método do potencial*. Definimos o *potencial*, Φ, do `DualArrayDeque` como a

diferença dos tamanhos entre `front` e `back`:

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

Uma propriedade interessante desse potencial é que uma chamada a `add(i,x)` ou `remove(i)` que não faz nenhum balanceamento pode aumentar o potencial em até 1.

Observe que, imediatamente após uma chamada a `balance()` que desloca elementos a ,o potencial, Φ_0, é no máximo 1, pois

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

Considere a situação no momento exatamente anterior a uma chamada `balance()` que desloca elementos e suponha, sem perda de generalidade, que `balance()` está deslocando elementos porque $3\texttt{front}.\texttt{size}() < \texttt{back}.\texttt{size}()$. Note que, nesse caso

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

Além disso, o potencial nesse momento é

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

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

2.5.2 Resumo

O teorema a seguir resume as propriedades de uma `DualArrayDeque`:

Teorema 2.4. *Uma* `DualArrayDeque` *implementa a interface* `List`*. Ignorando o custo de chamadas a* `resize()` *e* `balance()`*, uma* `DualArrayDeque` *possui as operações*

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

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

2.6 RootishArrayStack: Uma Stack Array Eficiente No Uso de Espaço

Uma das desvantagens de todas as estruturas de dados anteriores neste capítulo é que, porque elas guardam os dados em um array ou dois e evitam redimensionar esses arrays com frequência, os arrays frequentemente não estão muito cheios. Por exemplo, imediatamente após uma operação `resize()` em uma `ArrayStack`, o array de apoio `a` tem somente metade do espaço em uso. E pior, às vezes somente um terço de `a` contém dados.

Nesta seção, discutimos a estrutura de dados `RootishArrayStack`, que resolve o problema de espaço desperdiçado. A `RootishArrayStack` guarda `n` elementos usando arrays de tamanho $O(\sqrt{n})$. Nesses arrays, no máximo $O(\sqrt{n})$ posições do array estão vazias a qualquer momento. Todo o restante do array está sendo usado para guardar dados. Portanto, essas estruturas de dados gastam até $O(\sqrt{n})$ de espaço ao guardar `n` elementos.

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

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

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

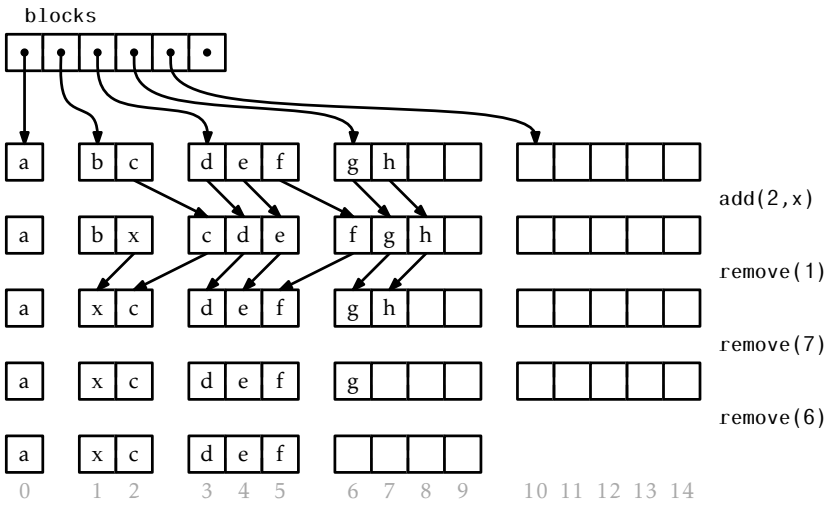


Figura 2.5: Uma sequência de operações add(i,x) e remove(i) em uma RootishArrayStack. Flechas denotam elementos sendo copiados.

```
RootishArrayStack
List<T[]> blocks;
int n;
```

Como podemos esperar, os elementos da lista estão dispostos em ordem dentro dos blocos. O elemento da lista com índice 0 é guardado no bloco 0, elementos com índices 1 e 2 são guardados no bloco 1, elementos com índices 3, 4 e 5 são guardados no bloco 2 e assim por diante. O principal problema que temos para resolver é o de determinar, dado um índice i, qual bloco contém i e também o índice correspondente a i naquele bloco.

Determinar o índice de i no bloco acaba sendo fácil. Se o índice i está no bloco b, então o número de elementos nos blocos $0,\ldots,b-1$ é $b(b+1)/2$. Portanto, i é guardado na posição

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

dentro do bloco b. Um pouco mais desafiador é o problema de determinar o valor de b. O número de elementos que têm índices menores que ou

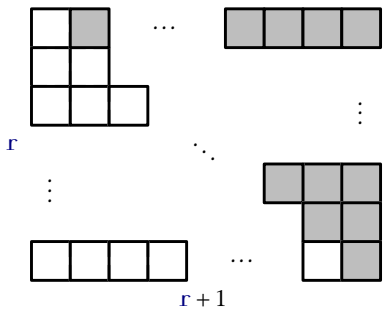


Figura 2.6: O número de quadrados brancos é $1 + 2 + 3 + \cdots + r$. O número de quadrados sombreados é o mesmo. Juntos, os quadrados brancos e sombreados compõem um retângulo consistindo de $r(r+1)$ quadrados.

iguais a i é $i + 1$. Por outro lado, o número de elementos nos blocos $0, \ldots, b$ é $(b+1)(b+2)/2$. Portanto, b é o menor inteiro tal que

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

Podemos reescrever essa equação da forma

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

A equação quadrática correspondente $b^2 + 3b - 2i = 0$ tem duas soluções : $b = (-3 + \sqrt{9 + 8i})/2$ e $b = (-3 - \sqrt{9 + 8i})/2$. A segunda solução não faz sentido na nossa aplicação pois é um valor negativo. Portanto, obtemos a solução $b = (-3 + \sqrt{9 + 8i})/2$. Em geral, essa solução não é um inteiro, mas retornando à nossa desigualdade, queremos o menor inteiro b tal que $b \geq (-3 + \sqrt{9 + 8i})/2$. Isso é simplesmente

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

RootishArrayStack

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

Com isso fora do caminho, os métodos get(i) e set(i,x) são diretos. Primeiramente computamos o bloco b apropriado e o índice j dentro do bloco e realizar 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 quaisquer estruturas de dados deste capítulo para representar a lista de blocks, então get(i) e set(i,x) irão rodar em tempo constante.

O método add(i,x) irá, a esta altura, parecer familiar. Primeiro verificamos se nossa estrutura de dados está cheia ao verificar se o número de blocos , r, é tal que $r(r+1)/2 = n$. Caso positivo, chamamos grow() para adicionar outro bloco. Com isso feito, deslocamos elementos com índices $i,\ldots,n-1$ à direita em uma posição para abrir 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 dele. Ele adiciona um novo bloco à estrutura de dados:

RootishArrayStack
```
void grow() {
  blocks.add(newArray(blocks.size()+1));
}
```

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

A operação remove(i) é similar a add(i,x). Ela desloca os elementos com índices $i+1,\ldots,n$ à esquerda em uma posição e então, se há mais de um bloco vazio, ela chama 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) {
    blocks.remove(blocks.size()-1);
    r--;
  }
}
```

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

2.6.1 Análise de expandir e encolher

A análise acima de add(i,x) e remove(i) não leva em conta o custo de grow() e shrink(). Note que, diferentemente da operação ArrayStack.resize(), grow() e shrink() não copiam nenhum dado. Elas simplesmente alocam ou liberam um array de tamanho r. Em alguns ambientes, isso leva apenas tempo constante, enquanto em outros, pode ser necessário tempo proporcional a r.

Note que, imediatamente após uma chamada a grow() ou shrink(), a situação é clara. O bloco final está completamente vazio e todos os outros blocos estão completamente cheios. Outra chamada a grow() ou shrink() não acontecerá até que pelo menos $r-1$ elementos tenham sido adicionados ou removidos. Portanto, mesmo se grow() e shrink() leve tempo $O(r)$, esse custo pode ser amortizado em pelo menos $r-1$ operações add(i,x) ou remove(i) fazendo com que o custo amortizado de grow() e shrink() sejam $O(1)$ por operação.

2.6.2 Uso de Espaço

A seguir, analisaremos a quantidade de espaço extra usada por uma RootishArrayStack.

Em especial, queremos contar qualquer espaço usado por uma RootishArrayStack que não seja de um elemento de array usado no momento para guardar um elemento da lista. Chamamos todo esse espaço de *espaço desperdiçado*.

A operação remove(i) assegura que uma RootishArrayStack nunca tenha mais do que dois blocos que não estejam completamente cheios. O número de blocos r, usados por uma RootishArrayStack que guarda n elementos portanto satisfaz

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

De novo, usando a equação quadrática resulta em

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

Os últimos dois blocos têm tamanhos r e $r-1$, então o espaço desperdiçado por esses dois blocos é até $2r-1 = O(\sqrt{n})$. Se guardarmos os blo-

cos em (por exemplo) uma `ArrayStack`, então a quantidade de espaço gasto pela `List` que guarda esses r blocks também é $O(r) = O(\sqrt{n})$. O espaço adicional necessário para guardar n e outras informações auxiliares é $O(1)$. Portanto, a quantidade total de espaço desperdiçado em uma `RootishArrayStack` é $O(\sqrt{n})$.

A seguir, demonstramos que esse uso de espaço é ótimo para qualquer estrutura de dados que inicia-se vazia e permite a adição de um item por vez. Mais precisamente, mostraremos que, em algum momento durante a adição de n itens, a estrutura de dados está desperdiçando pelo menos um espaço de $\sqrt{n}$ (embora possa ser por apenas um curto momento).

Suponha que iniciamos com uma estrutura de dados vazia e adicionamos n itens, um por vez. No fim desse processo, todos os n itens são guardados na estrutura e distribuídos entre uma coleção de r blocos de memória. Se $r \geq \sqrt{n}$, então a estrutura de dados precisa usar r ponteiros (ou referências) para gerenciar esses r blocos e esses ponteiros são espaço desperdiçado. Por outro lado, se $r < \sqrt{n}$ então, pelo princípio das casas de pombos, algum bloco deve ter tamanho de pelo menos $n/r > \sqrt{n}$. Considere o momento no qual esse bloco foi inicialmente alocado. Imediatamente após alocá-lo, esse bloco estava vazio e portanto desperdiçando um espaço de $\sqrt{n}$. Portanto, em algum momento durante a inserção de n elementos, a estrutura de dados estava gastando $\sqrt{n}$ de espaço.

2.6.3 Resumo

O seguinte teorema resume a discussão sobre a estrutura de dados `RootishArrayStack`:

Teorema 2.5. *Uma `RootishArrayStack` implementa a interface `List`. Ignorando o custo das chamadas a `grow()` e `shrink()`, uma `RootishArrayStack` possui as operações*

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

Além disso, ao começar com uma `RootishArrayStack` vazia, qualquer sequência de m *operações* `add(i,x)` *e* `remove(i)` *resulta em um total de* $O(m)$ *tempo gasto durante todas as chamadas de* `grow()` *e* `shrink()`. *O espaço (medido*

em palavras)[3] *usada por uma* `RootishArrayStack` *que guarda* n *elementos é* $n + O(\sqrt{n})$.

2.6.4 Computando Raízes Quadradas

Um leitor que tenha tido alguma exposição a modelos de computação pode notar que a `RootishArrayStack`, conforme descrita anteriormente, não se encaixa no modelo de computação normal word-RAM (Seção 1.4) porque requer a computação de raízes quadradas. A operação raiz quadrada não é geralmente considerada uma operação básica e portanto não é normalmente parte do modelo word-RAM.

Nesta seção, mostraremos que a operação raiz quadrada pode ser implementada eficientemente. Em particular, mostramos que para qualquer inteiro $x \in \{0, \ldots, n\}$, $\lfloor \sqrt{x} \rfloor$ pode ser computado em tempo constante, após um pré-processamento com tempo de execução de $O(\sqrt{n})$ que cria dois arrays de tamanho $O(\sqrt{n})$. O lema a seguir mostra que podemos reduzir o problema de computar a raiz quadrada de x à 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

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

Eleva-se ao quadrado ambos lados dessa desigualdade para obter

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

e junta-se os termos para obter

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

o que é claramente verdadeiro para qualquer $x \geq 1$. □

Inicia-se restringindo o problema um pouco e assume-se que $2^r \leq x < 2^{r+1}$, tal que $\lfloor \log x \rfloor = r$, isto é, x é um inteiro com $r + 1$ bits na sua representação binária. Podemos fazer que $x' = x - (x \bmod 2^{\lfloor r/2 \rfloor})$. Agora, x'

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

satisfaz as condições do Lema 2.3, então $\sqrt{x} - \sqrt{x'} \leq 1$. Além disso, x' tem todos os seu bits de baixa ordem $\lfloor r/2 \rfloor$ iguais a 0, então só há

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

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

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

Desse modo, $\texttt{sqrttab}[i]$ tem uma diferença de até 2 em relação a $\sqrt{x}$ para todo $x \in \{i2^{\lfloor r/2 \rfloor}, \ldots, (i+1)2^{\lfloor r/2 \rfloor} - 1\}$. De outra forma, a entrada do array $s = \texttt{sqrttab}[x \gg \lfloor r/2 \rfloor]$ é igual a $\lfloor \sqrt{x} \rfloor$, a $\lfloor \sqrt{x} \rfloor - 1$ ou a $\lfloor \sqrt{x} \rfloor - 2$. A partir de s podemos determinar o valor $\lfloor \sqrt{x} \rfloor$ ao incrementar s até u $(s+1)^2 > x$.

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;
}
```

Isso somente funciona para $x \in \{2^r, \ldots, 2^{r+1} - 1\}$ e `sqrttab` é uma tabela especial que funciona somente para um valor especial de $r = \lfloor \log x \rfloor$. Para superar isso, poderíamos computar $\lfloor \log n \rfloor$ arrays `sqrttab` diferentes, um para cada valor possível de $\lfloor \log x \rfloor$. Os tamanhos dessas tabelas formam uma sequência exponencial cujo maior valor é até $4\sqrt{n}$, então o tamanho total de todas as tabelas é $O(\sqrt{n})$.

Porém, acontece que mais um array `sqrttab` é desnecessário; somente precisamos de um array `sqrttab` para o valor $r = \lfloor \log n \rfloor$. Qualquer valor x com $\log x = r' < r$ pode ser *atualizado* ao multiplicar x por $2^{r-r'}$ e usar a equação

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

A quantidade $2^{r-r'} x$ está no intervalo $\{2^r, \ldots, 2^{r+1} - 1\}$ então podemos procurar sua raiz quadrada em `sqrttab`. O código a seguir implementa essa ideia para computar $\lfloor \sqrt{x} \rfloor$ para todo inteiro não negativo 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 tínhamos como certo até agora depende de como computamos $r' = \lfloor \log x \rfloor$. De novo, esse problema pode ser resolvido com um array, `logtab`, de tamanho $2^{r/2}$. Nesse caso, o código é particularmente simples, pois $\lfloor \log x \rfloor$ é somente o índice do bit 1 mais significativo na representação binária de `x`. Isso significa que, para $x > 2^{r/2}$, podemos deslocar à direita os bits de `x` por $r/2$ posições antes de usá-lo como um índice em `logtab`. O código a seguir faz isso usando um array `logtab` de tamanho 2^{16} para computar $\lfloor \log x \rfloor$ para todo `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 completude, incluímos o código a seguir 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++)
    Arrays.fill(logtab, 1<<d, 2<<d, 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;
```

```
  }
}
```

Para resumir, as computações feitas pelo método i2b(i) podem ser implementadas em tempo constante no modelo word-RAM usando $O(\sqrt{n})$ de memória extra para guardar os arrays sqrttab e logtab. Esses arrays podem ser reconstruídos quando n aumenta ou diminui por um fator de dois e o custo dessa reconstrução pode ser amortizado sobre o número de operações add(i,x) e remove(i) que causaram a mudança do mesmo jeito que o custo de resize() é analisado na implementação da ArrayStack.

2.7 Discussão e Exercícios

A maior parte das estruturas de dados descritas neste capítulo são folclore. Elas podem ser encontradas em implementações de pelo menos 30 anos atrás. Por exemplo, implementações de stacks, queues e deques que generalizam facilmente para as estruturas ArrayStack, ArrayQueue e ArrayDeque descritas aqui, são discutidas por Knuth [46, Section 2.2.2].

Brodnik *et al.* [13] parecem terem sido os primeiros a descrever a RootishArrayStack e provar um limitante inferior de $\sqrt{n}$ como descrito na Seção 2.6.2. Eles também apresentam uma estrutura diferente que usa uma escolha de tamanhos de blocos mais sofisticada a fim de evitar a computação de raízes quadradas usando o método i2b(i). Com o esquema deles, o bloco contendo i é o bloco $\lfloor \log(i+1) \rfloor$, que é simplesmente o índice do primeiro bit 1 na representação binária de $i+1$. Algumas arquiteturas de computadores provêm uma instrução para computar o índice do primeiro bit 1 em um inteiro. Em Java, a classe Integer provê um método chamado numberOfLeadingZeros(i) a partir do qual pode-se facilmente computar $\lfloor \log(i+1) \rfloor$.

Uma estrutura relacionada à RootishArrayStack é o *vetor em camadas* de Goodrich e Kloss [35]. Essa estrutura suporta as operações get(i,x) e set(i,x) em tempo constante e add(i,x) e remove(i) em $O(\sqrt{n})$ de tempo. Esses tempos de execução são similares ao que pode ser conseguido com uma implementação mais cuidadosa de uma RootishArrayStack discutida no Exercício 2.11.

Exercício 2.1. Na implementação da ArrayStack, após a primeira chamada a remove(i), o array de apoio, a, contém $n+1$ valores não null apesar do fato de que a ArrayStack somente contém n elementos. Onde está o valor não null extra? Discuta as consequências que esse valor não null possa ter no gerenciador de memória da Java Runtime Environment.

Exercício 2.2. O método da List addAll(i, c) insere todos os elementos da Collection c na posição i da lista. (O método add(i, x) é um caso especial onde $c = \{x\}$.) Explique porque, para as estruturas de dados deste capítulo, não é eficiente implementar addAll(i, c) fazendo chamadas repetidas a add(i, x). Projete e codifique uma implementação mais eficiente.

Exercício 2.3. Projete e implemente uma *RandomQueue*. Essa é uma implementação da interface Queue na qual a operação remove() remove um elemento que é escolhido de uma distribuição aleatória uniforme entre todos os elementos atualmente na queue. (Considere uma RandomQueue como sendo uma sacola na qual podemos adicionar elementos ou por a mão e, sem ver, remover algum elemento aleatório.) As operações add(x) e remove() em uma RandomQueue deve rodar em um tempo constante amortizado por operação.

Exercício 2.4. Projete e implemente uma Treque (uma queue com três pontas). Essa é uma implementação de uma List na qual get(i) e set(i, x) rodam em tempo constante e add(i, x) e remove(i) rodam em tempo

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

Em outras palavras, modificações são rápidas se elas estão perto ou do fim, ou do meio da lista.

Exercício 2.5. Implemente um método rotate(a, r) que "rotaciona" o array a de tal forma que a[i] moves para a[(i + r) mod a.length], para todo $i \in \{0, \ldots, a.length\}$.

Exercício 2.6. Implemente um método rotate(r) que "rotaciona" uma List tal que o item da lista i se torna o item (i + r) mod n. Ao rodar em uma ArrayDeque ou uma DualArrayDeque, rotate(r) deve rodar em tempo $O(1 + \min\{r, n-r\})$.

Exercício 2.7. Modifique a implementação da ArrayDeque tal que o deslocamento feito por add(i, x), remove(i) e resize() é realizado usando o método rápido System.arraycopy(s, i, d, j, n).

Exercício 2.8. Modifique a implementação da ArrayDeque tal que não use o operador % (que é caro em alguns sistemas). Em vez disso, deve fazer uso do fato que se a.length é uma potência de 2, então

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

(Aqui, & é o operador and bit-a-bit.)

Exercício 2.9. Projete e implemente uma variante da ArrayDeque que não faz uso de aritmética modular. Em vez disso, todos os dados ficam em um bloco consecutivo, em ordem, dentro de um array. Quando os dados sobrescrevem o começo ou o fim desse array, uma operação rebuild() modificada é realizada. O custo amortizado de todas as operações deve ser o mesmo que em uma ArrayDeque.
Dica: fazer isso funcionar tem a ver com a forma que você implementa a operação rebuild(). Deve-se fazer rebuild() para colocar dados na estrutura de dados em um estado onde os dados não podem sair pelas duas pontas até que pelo menos n/2 operações tenham sido realizadas.

Teste o desempenho da sua implementação em comparação à ArrayDeque. Otimize sua implementação (usando System.arraycopy(a, i, b, i, n)) e veja se consegue fazê-la mais rápida que a implementação da ArrayDeque.

Exercício 2.10. Projete e implemente uma versão de uma RootishArrayStack que desperdiça somente $O(\sqrt{n})$ de espaço, mas que pode realizar as operações add(i, x) e remove(i, x) de $O(1 + \min\{i, n - i\})$ de tempo.

Exercício 2.11. Projete e implemente uma versão de RootishArrayStack que desperdiça somente $O(\sqrt{n})$ de espaço, mas que pode rodar as operações add(i, x) e remove(i, x) em $O(1 + \min\{\sqrt{n}, n - i\})$ de tempo. (Para uma ideia de como fazer isso, veja a Seção 3.3.)

Exercício 2.12. Projete e implemente uma versão de uma RootishArrayStack que desperdiça somente $O(\sqrt{n})$ de espaço, mas que pode rodar as operações add(i, x) e remove(i, x) em $O(1 + \min\{i, \sqrt{n}, n - i\})$ de tempo. (Veja a Seção 3.3 para uma ideia de como conseguir isso.)

Exercício 2.13. Projete e implemente uma `CubishArrayStack`. Essa estrutura de três níveis implementa a interface `List` desperdiçando $O(n^{2/3})$ de espaço. Nessa estrutura, `get(i)` e `set(i,x)` funcionam em tempo constante; enquanto `add(i,x)` e `remove(i)` levam $O(n^{1/3})$ de tempo amortizado.

Capítulo 3

Listas Ligadas

Neste capítulo, continuamos a estudar implementações da interface `List` e agora usando estruturas de dados baseadas em ponteiros em vez de arrays. As estruturas neste capítulo são compostas de nodos (ou nós) que contêm os itens da lista. Usando referências (ponteiros), os nodos são ligados entre si em uma sequência. Primeiro estudamos listas simplesmente ligadas (listas com somente uma ligação), que podem implementar operação da `Stack` e `Queue` (FIFO) em tempo constante por operação e então seguimos para listas duplamente ligadas, que podem implementar operações `Deque` em tempo constante.

Listas ligadas têm vantagens e desvantagens quando comparadas a implementações baseadas em arrays da interface `List`. A principal desvantagem é que perdemos a habilidade de acessar elementos usando `get(i)` ou `set(i,x)` em tempo constante. Em vez disso, temos que percorrer a lista, um elemento por vez, até chegarmos no `i`-ésimo elemento. A principal vantagem é que são mais dinâmicos: dada uma referência a qualquer nodo da lista `u`, podemos deletar `u` ou inserir um nodo adjacente a `u` em tempo constante. Isso vale não importa onde `u` esteja na lista.

3.1 SL`List`: Uma Lista Simplesmente Ligada

Uma `SLList` (em inglês, singly-linked list) é uma sequência de `Node`s (em português, nodo). Cada nodo `u` guarda um valor de dado `u.x` e uma referência `u.next` ao próximo nodo na sequência. Para o último nodo `w` na

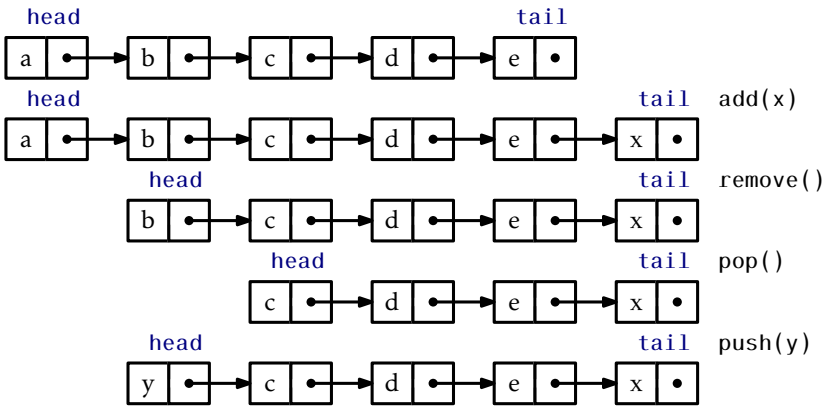


Figura 3.1: Uma sequência de operações da `Queue` (`add(x)` e `remove()`) e da `Stack` (`push(x)` e `pop()`) em uma `SLList`.

sequência vale `w.next` = `null`

SLList
```
class Node {
  T x;
  Node next;
}
```

Para eficiência, uma `SLList` usa variáveis `head` (cabeça) e `tail` (cauda) para guardar os acessos à primeiro e último nodos na sequência, assim como um inteiro `n` para guardar o comprimento da sequência:

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

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

Uma `SLList` pode implementar eficientemente as operações da `Stack` `push()` e `pop()` ao adicionar e remover elementos na cabeça da sequência. A operação `push()` simplesmente cria um novo nodo `u` com valor de dado

x, atribuído a u.next à antiga cabeça da lista e transforma u na nova cabeça da lista. Finalmente, ela incrementa n pois o tamanho de SLList aumentou em uma unidade:

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

A operação pop(), após verificar que a SLList não está vazia, remove a cabeça fazendo head = head.next e também decrementando n.

SLList
```
T pop() {
  if (n == 0)  return null;
  T x = head.x;
  head = head.next;
  if (--n == 0) tail = null;
  return x;
}
```

Claramente, ambas operações push() e pop() rodam em tempo 0(1).

3.1.1 Operações de Queue

Uma SLList também pode implementar as operações de queue FIFO add(x) e remove() em tempo constante. Remoções são feitas da cabeça da lista e são idênticas à operação pop():

SLList
```
T remove() {
  if (n == 0)  return null;
```

```
  T x = head.x;
  head = head.next;
  if (--n == 0) tail = null;
  return x;
}
```

Adições, por outro lado, são feitas na cauda da lista. Na maior parte dos casos, isso é feito ao atribuir `tail.next = u`, onde `u` é mais novo nodo que contém `x`. Entretanto, um caso especial ocorre quando `n` = 0, no qual `tail = head = null`. Nesse caso, tanto `tail` quanto `head` são atribuídos a `u`.

SLList

```
boolean add(T x) {
  Node u = new Node();
  u.x = 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 teorema a seguir resume o desempenho de uma `SLList`:

Teorema 3.1. *Uma `SLList` implementa as interfaces `Stack` e `Queue` (FIFO). As operações* `push(x)`, `pop()`, `add(x)` *e* `remove()` *rodam em tempo* $O(1)$ *por operação.*

Uma `SLList` quase implementa o conjunto completo de operações de uma `Deque`. A única operação que falta é remoção da cauda de uma `SLList`. Remover da cauda de um `SLList` é difícil porque requer a atualização do valor da `tail` de forma que ela aponte ao nodo `w` que precede `tail`

na SLList; esse é o nodo w tal que w.next = tail. Infelizmente, o único jeito de chegar a w é percorrendo SLList a começar da head fazendo n − 2 passos.

3.2 DLList: Uma Lista Duplamente Ligada

Uma DLList (do inglês, doubly-linked list) é muito similar a uma SLList exceto que cada nodo u em uma DLList tem referências ao nodo que ele precede, u.next, e ao nodo que sucede, u.prev.

DLList
```
class Node {
  T x;
  Node prev, next;
}
```

Ao implementar uma SLList, vimos que haviam vários casos especiais para tomar cuidado. Por exemplo, ao remover o último elemento de uma SLList ou adicionar um elemento a uma SLList vazia precisamos tomar cuidado especial para garantir que head e tail sejam corretamente atualizados. Em uma DLList, o número desses casos especiais aumentam consideravelmente. Talvez o modo mais simples de cuidar de todos esses casos especiais em uma DLList é introduzir um nodo dummy, também conhecido como *sentinela*. Esse nodo não contém nenhum dado, mas atua como um marcador para que não tenhamos nenhum nodo especial; todo nodo tem um next e um prev, com um dummy agindo como o nodo que sucede o último nodo na lista e que precede o primeiro nodo da lista. Desse jeito, os nodos da lista são (duplamente) ligados em um ciclo, como ilustrado na Figura 3.2.

DLList
```
int n;
Node dummy;
DLList() {
  dummy = new Node();
  dummy.next = dummy;
  dummy.prev = dummy;
```

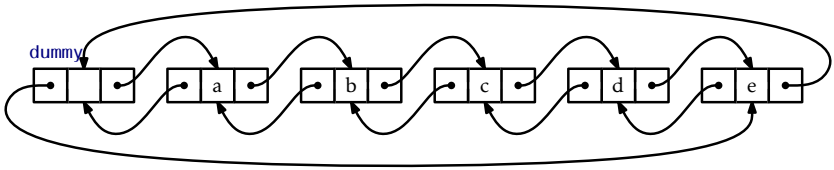


Figura 3.2: Uma DLList contendo a,b,c,d,e.

```
  n = 0;
}
```

Encontrar o nodo com um dado índice em uma DLList é fácil; podemos começar na cabeça da lista (dummy.next) e ir para frente, ou começar na cauda da lista (dummy.next) e ir para trás. Isso permite alcançarmos o i-ésimo nodo em $O(1+\min\{i, n-i\})$ de tempo:

DLList

```
Node getNode(int i) {
  Node p = null;
  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) também são fáceis agora. Primeiro achamos o i-ésimo nodo e então pegamos ou atribuímos seu valor x:

DLList

```
T get(int i) {
  return getNode(i).x;
}
T set(int i, T x) {
```

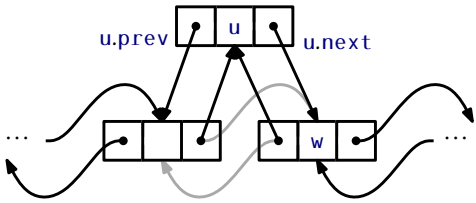


Figura 3.3: Adicionando o nodo u antes do nodo w em uma DLList.

```
  Node u = getNode(i);
  T y = u.x;
  u.x = x;
  return y;
}
```

O tempo de execução dessas operações é dominado pelo tempo para achar o i-ésimo nodo, e portanto é $O(1+\min\{i, n-i\})$.

3.2.1 Adicionando e Removendo

Se temos uma referência a um nodo w em uma DLList e queremos inserir um nodo u antes de w, então isso é só uma questão de atribuir u.next = w, u.prev = w.prev e então ajustar u.prev.next e u.next.prev. (Veja a Figura 3.3.) Graças ao nodo dummy, não há necessidade de se preocupar sobre w.prev existir ou não.

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;
}
```

Agora, a operação da lista add(i,x) é trivial para implementar. Achamos o i-ésimo nodo na DLList e inserimos um novo nodo u que contém x logo 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 que leva para achar o i-ésimo nodo (usando getNode(i)). Então, add(i,x) roda em tempo $O(1+\min\{i,n-i\})$.

Remover um nodo w de uma DLList é fácil. Precisamos somente ajustar ponteiros em w.next e w.prev tal que eles *saltem* w. De novo, o uso do nodo dummy elimina a necessidade de considerar casos especiais:

DLList
```
void remove(Node w) {
  w.prev.next = w.next;
  w.next.prev = w.prev;
  n--;
}
```

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

DLList
```
T remove(int i) {
  Node w = getNode(i);
  remove(w);
  return w.x;
}
```

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

3.2.2 Resumo

O teorema a seguir resume o desempenho de uma DLList:

Teorema 3.2. *Uma* `DLList` *implementa a interface* `List`*. Nessa implementação, as operações* `get(i)`, `set(i,x)`, `add(i,x)` *e* `remove(i)` *rodam em tempo* $O(1 + \min\{i, n - i\})$ *por operação.*

Vale notar que, se ignorarmos o custo de `getNode(i)`, então todas as operações em uma `DLList` leva tempo constante. Então, a única parte cara das operações em uma `DLList` é achar o nodo relevante. Uma vez que temos o nodo de interesse, adicionar, remover ou acessar os dados naquele nodo leva somente um tempo constante.

Há um nítido contraste em relação às implementações da `List` baseadas em arrays do Capítulo 2; naquelas implementações, o item buscado no array pode ser achado em tempo constante. Entretanto, adicionar ou remover requer o deslocamento de elementos no array e, em geral, não leva tempo constante.

Por essa razão, as estruturas de listas ligadas são adequadas para aplicações em que referências a nodos da lista podem ser obtidos por meios externos.

Um exemplo disso é a estrutura de dados `LinkedHashSet` existente na Java Collections Framework, na qual um conjunto de itens é guardado em uma lista duplamente encadeada e os nodos dessa lista são guardados em uma tabela hash (discutido no Capítulo 5). Quando um elemento é removido de uma `LinkedHashSet`, a tabela hash é usada para achar o nodo em questão com tempo constante e então o nodo da lista é removido (também em tempo constante).

3.3 SE`List`: Uma Lista Ligada Eficiente no Espaço

Uma das limitações das listas encadeadas (além do tempo que ela leva para acessar elementos que estão no meio da lista) é o seu uso de espaço. Cada nodo em uma `DLList` requer duas referências adicionais para o próximo nodo e para o nodo anterior na lista. Dois dos campos em um `Node` são dedicados a manter a lista, e somente um dos campos é para guardar dados!

Uma `SEList` (do inglês, space-efficient list) reduz esse espaço desperdiçado usando uma ideia simples: em vez de guardar elementos individuais em uma `DLList`, guardamos um bloco (array) contendo vários itens.

Mais precisamente, uma SEList é parametrizada pelo tamanho do bloco *block size* b. Cada nodo individual em uma SEList guarda um bloco que pode guardar até $b+1$ elementos.

Para razões que vão se tornar claras depois, será útil se podermos fazer operações de Deque em cada bloco.

A estrutura de dados que escolhemos para isso é uma BDeque (do inglês, bounded deque – deque delimitado), derivado da estrutura ArrayDeque descrita na Seção 2.4. A BDeque difere do ArrayDeque em um detalhe: quando uma nova BDeque é criada, o tamanho do array de apoio a é fixo em $b+1$ e nunca expande ou escolhe. A propriedade mais importante de um BDeque é que permite a adição ou remoção de elementos tanto na frente quanto atrás em tempo constante. Isso será útil quando elementos forem deslocados de um bloco a outro.

SEList
```
class BDeque extends ArrayDeque<T> {
  BDeque() {
    super(SEList.this.type());
    a = newArray(b+1);
  }
  void resize() { }
}
```

Uma SEList é então uma lista duplamente encadeada de blocos:

SEList
```
class Node {
  BDeque d;
  Node prev, next;
}
```

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

3.3.1 Requisitos de Espaço

Uma SEList impõe restrições bem firmes ao número de elementos em um bloco: a não ser que o bloco seja o último, então esse bloco contém pelo menos $b-1$ e até $b+1$ elementos. Isso significa que, se uma SEList contém n elementos, então ela tem até

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

blocos. A BDeque para cada bloco contém um array de tamanho $b+1$ mas para todo bloco exceto o último, no máxima um quantidade constante de espaço é desperdiçada nesse array. O resto da memória usada por um bloco também é constante. Isso significa que o espaço desperdiçado em uma SEList é somente $O(b+n/b)$. Ao escolher um valor de b dentro de um fator constante de $\sqrt{n}$, fazemos o custo de espaço extra de uma SEList se aproximar ao limitante inferior $\sqrt{n}$ descrito na Seção 2.6.2.

3.3.2 Procurando Elementos

O primeiro desafio que enfrentamos com uma SEList é procurar o item da lista com um dado índice i. Note que a localização de um elemento consiste de duas partes:

1. o nodo u que contém o bloco que contém o elemento com índice i; e

2. o índice j do elemento dentro desse bloco.

SEList

```
class Location {
  Node u;
  int j;
  Location(Node u, int j) {
    this.u = u;
    this.j = j;
  }
}
```

Para achar o bloco que contém um dado elemento, fazemos do mesmo jeito que é feito em uma DLList Iniciamos do início da lista e percorremos até o final, ou a partir do final da lista percorremos em direção ao início até encontrarmos o nodo que queremos. A única diferença é que cada vez que movemos de um nodo ao seguinte, pulamos um bloco inteiro de elementos.

SEList

```
Location getLocation(int i) {
  if (i < n/2) {
    Node u = dummy.next;
    while (i >= u.d.size()) {
      i -= u.d.size();
      u = u.next;
    }
    return new Location(u, i);
  } else {
    Node u = dummy;
    int idx = n;
    while (i < idx) {
      u = u.prev;
      idx -= u.d.size();
    }
    return new Location(u, i-idx);
  }
}
```

Lembre-se que, com exceção de no máximo um bloco, cada bloco contém pelo menos $b-1$ elementos, então cada passo na nossa busca nos leva a $b-1$ elementos mais próximos ao elemento que estamos procurando. Se estamos procurando do começo ao fim, isso significa que alcançamos o nodo que queremos após $O(1+i/b)$ passos. Se procuramos de trás para frente, então alcançamos o nodo que queremos após $O(1+(n-i)/b)$ passos. O algoritmo usa a menor dessas duas quantidades dependendo do valor de i, então o tempo de localizar o item com índice i é $O(1+\min\{i,n-i\}/b)$.

Uma vez que sabemos como localizar o item com índice i, as operações $\texttt{get}(i)$ e $\texttt{set}(i,x)$ são traduzidas em obter e atribuir em um índice particular no bloco correto:

SEList

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

Os tempos de execução dessas operações são dominados pelo tempo que leva para achar o item, então eles também rodam em tempo $O(1 + \min\{i, n-i\}/b)$.

3.3.3 Adicionando um Elemento

Adicionar elementos a uma SEList é um pouco mais complicado. Antes de considerar o caso geral, consideraremos a operação mais fácil, add(x), na qual x é adicionada ao final da lista. Se o último bloco está cheio (ou não existe porque não há blocos ainda), então primeiro alocamos um novo bloco e o acrescentamos na lista de blocos. Agora que temos certeza de que o último bloco existe e não está cheio, acrescentamos x ao último bloco.

SEList

```
boolean add(T x) {
  Node last = dummy.prev;
  if (last == dummy || last.d.size() == b+1) {
    last = addBefore(dummy);
  }
  last.d.add(x);
  n++;
  return true;
}
```

A situação complica quando adicionamos ao interior da lista usando add(i,x) Primeiro alocamos i para obter o nodo u cujo bloco contém o

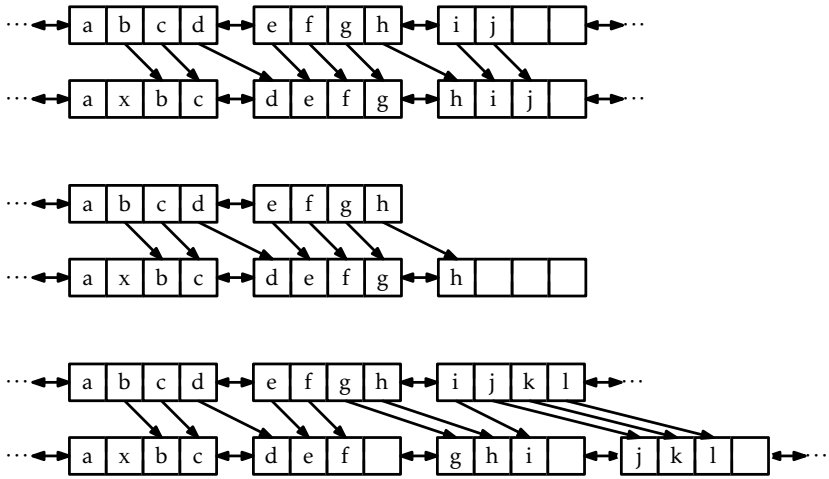


Figura 3.4: Os três casos que ocorrem durante a adição de um item x no interior de uma SEList. (Essa SEList tem tamanho de bloco b = 3.)

i-ésimo item da lista. O problema é que queremos inserir x no bloco de u, mas temos que estar preparados para o caso em que o bloco de u já contenha b + 1 elementos, ou seja, cheio e sem espaço para x.

Considere que $u_0, u_1, u_2, \ldots$ denotam u, u.next, u.next.next e assim por diante. Exploramos $u_0, u_1, u_2, \ldots$ procurando por um nodo que provê espaço para x. Três casos podem ocorrer durante nossa exploração (ver a Figura 3.4):

1. Rapidamente (em $r + 1 \leq b$ passos) achamos um nodo u_r cujo bloco não está cheio. Nesse caso, realizamos r deslocamentos de um elemento de um bloco ao próximo, de forma que o espaço livre em u_r torne-se um espaço livre em u_0. Podemos então inserir x no bloco do u_0.

2. Podemos rapidamente (em $r + 1 \leq b$ passos) ir ao fim da lista de blocos. Nesse caso, adicionamos um bloco vazio ao fim da lista de bloco e preceder como no primeiro caso.

3. Após b passos não achamos nenhum bloco que não esteja cheio.

Nesse caso, $u_0, \ldots, u_{b-1}$ é uma sequência de b blocos onde cada contém $b+1$ elementos. Nós inserimos um novo bloco u_b no fim dessa sequência e *espalhamos* 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 somente b elementos, então ele tem espaço para inserir x.

SEList

```
void add(int i, T x) {
  if (i == n) {
    add(x);
    return;
  }
  Location l = getLocation(i);
  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 de quais dos três casos anteriores ocorrem. Casos 1 e 2 envolvem examinar e deslocar elementos ao longo de até b blocos e levam $O(b)$ de tempo. Caso 3 envolve chamar o método spread(u), que move $b(b+1)$ elementos e leva $O(b^2)$ de

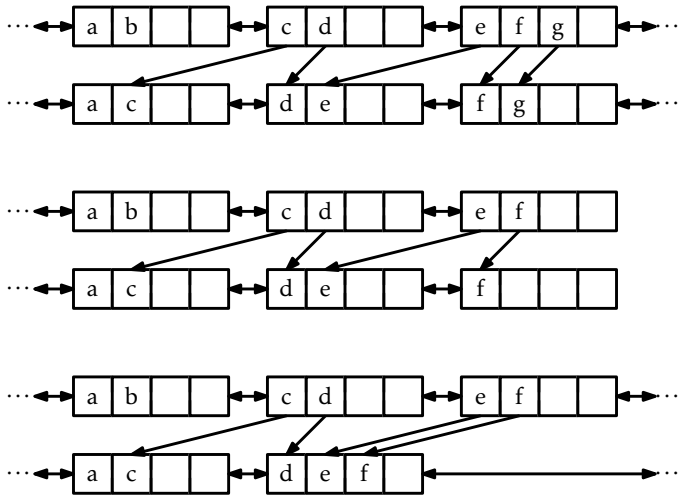


Figura 3.5: Os três casos que ocorrem durante a remoção de um item x no interior de uma SEList. (Essa SEList tem bloco de tamanho b = 3.)

tempo. Se ignorarmos o custo do Caso 3 (que iremos levar em consideração depois, com amortização) isso significa que o tempo total de execução para localizar i e realizar a inserção de x é $O(b + \min\{i, n - i\}/b)$.

3.3.4 Remoção de um Elemento

Remover um elemento de uma SEList é similar a adicionar um elemento. Nós primeiro localizamos o nodo u que contém o elemento com índice i. Agora, temos que estar preparados para o caso no qual não podemos remover um elemento de u sem fazer que o bloco de u se torne menor que $b - 1$.

Novamente, sejam $u_0, u_1, u_2, \ldots$ os ponteiros u, u.next, u.next.next e assim por diante. Nós examinamos $u_0, u_1, u_2, \ldots$ para procurar por um nodo do qual podemos emprestar um elemento para fazer o tamanho do bloco de u_0 pelo menos $b - 1$. Há outros casos a considerar (veja a Figura 3.5):

1. Rapidamente (em $r + 1 \leq b$ passos) achamos um nodo cujo bloco

contém mais de $b-1$ elementos. Nesse caso, realizamos r deslocamentos de um bloco no anterior, de forma tal que o elemento extra em u_r se torna um elemento extra em u_0. Podemos então remover o elemento apropriado do bloco de u_0.

2. Podemos rapidamente (em $r+1 \leq b$ passos) pular ao fim da lista de blocos. Nesse caso, u_r é o último bloco e não há necessidade para o bloco de u_r conter pelo menos $b-1$ elementos. Portanto, da mesma maneira que acima, emprestamos um elemento de u_r para fazer um elemento extra em u_0. Se isso causar o bloco de u_r se tornar vazio, então o removemos.

3. Após b passos, não achamos nenhum bloco contendo mais de $b-1$ elementos. Nesse caso, $u_0, \ldots, u_{b-1}$ é uma sequência de b blocos que contém $b-1$ elementos. Nós *acumulamos* esses $b(b-1)$ elementos em $u_0, \ldots, u_{b-2}$ de forma tal que cada um desses $b-1$ blocos contém 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);
  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) {  // 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.isEmpty()) remove(u);
  n--;
  return y;
}
```

Assim 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 do Espalhamento e Acumulação

A seguir, avaliaremos o custo dos métodos gather(u) e spread(u) que podem ser executados pelos métodos add(i,x) e remove(i). Por questão de completude, eles são mostrados a seguir:

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 desses métodos é dominado pelos dois laços aninhados. Ambos laços interno e externo executam no máximo $\mathtt{b}+1$ vezes, então o tempo total de execução de cada um desses métodos é $O((\mathtt{b}+1)^2) = O(\mathtt{b}^2)$. Porém, o lema a seguir mostra que esses métodos executam em, no máximo, uma a cada $\mathtt{b}$ chamadas de $\mathtt{add}(\mathtt{i},\mathtt{x})$ ou $\mathtt{remove}(\mathtt{i})$.

Lema 3.1. *Se uma SEList é criada e qualquer sequência de $m \geq 1$ chamadas a* $\mathtt{add}(\mathtt{i},\mathtt{x})$ *e* $\mathtt{remove}(\mathtt{i})$ *é realizada então o tempo total gasto durante todas as chamadas a* $\mathtt{spread}()$ *e* $\mathtt{gather}()$ *é* $O(\mathtt{b}m)$.

Demonstração. Usaremos o método do potencial da análise amortizada. Dizemos que um nodo $\mathtt{u}$ é *frágil* se o bloco de $\mathtt{u}$ não contém $\mathtt{b}$ elementos (então $\mathtt{u}$ pode ser o último nodo ou contém $\mathtt{b}-1$ ou $\mathtt{b}+1$ elementos). Qualquer nodo cujo bloco contém $\mathtt{b}$ elementos é *durável*. Defina o *potencial* de uma SEList como o número de nodos frágeis que possui. Iremos considerar somente a operação $\mathtt{add}(\mathtt{i},\mathtt{x})$ e sua relação com o número de chamadas a $\mathtt{spread}(\mathtt{u})$. As análises de $\mathtt{remove}(\mathtt{i})$ e $\mathtt{gather}(\mathtt{u})$ são idênticas.

Note que, se o Caso 1 ocorrer durante o método $\mathtt{add}(\mathtt{i},\mathtt{x})$, então somente um nodo $\mathtt{u}_r$ tem o tamanho de seu bloco alterado. Portanto, no máximo um nodo, $\mathtt{u}_r$, deixa de ser durável e torna-se frágil. Se o Caso 2 ocorrer, então um novo nodo é criado, e esse nó é frágil, mas nenhum outro nodo muda de tamanho, então o número de nodos frágeis aumenta em um. Então, nos Casos 1 ou 2 o potencial da SEList aumenta em até uma unidade.

Finalmente, se Caso 3 ocorrer, é porque $\mathtt{u}_0,\ldots,\mathtt{u}_{\mathtt{b}-1}$ são todos nodos frágeis. Então $\mathtt{spread}(u_0)$ é chamado e esses $\mathtt{b}$ nodos frágeis são substituídos por $\mathtt{b}+1$ nodos duráveis. Finalmente, $\mathtt{x}$ é adicionado ao bloco de $\mathtt{u}_0$ tornando $\mathtt{u}_0$ frágil. No total, o potencial reduz em $\mathtt{b}-1$.

Em resumo, o potencial inicia em 0 (não há nodos na lista). Cada vez que um Caso 1 ou um Caso 2 ocorre o potencial aumenta em até 1 unidade. Cada vez que um Caso 3 ocorre, o potencial decresce em $\mathtt{b}-1$. O potencial (que conta o número de nodos frágeis) nunca é menor que 0. Concluímos que, para toda ocorrência do Caso 3, houveram pelo menos $\mathtt{b}-1$ ocorrências de um Caso 1 ou um Caso 2. Então, para toda chamada a $\mathtt{spread}(\mathtt{u})$, existem pelo menos $\mathtt{b}$ chamadas a $\mathtt{add}(\mathtt{i},\mathtt{x})$. Isso conclui a prova. □

3.3.6 Resumo

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

Teorema 3.3. *Uma SEList implementa a interface List. Ignorando o custo de chamadas a* spread(u) *e* gather(u)*, uma SEList tamanho de bloco* b *possui as operações*

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

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

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

A SEList foi projetada visando um meio termo de custos entre ArrayList e uma DLList, onde a mistura de custos entre essas duas estruturas depende do tamanho do bloco b. Em um extremo $b = 2$, cada nodo da SEList guarda no máximo três valores, o que não é muito diferente em uma DLList. No outro extremo, $b > n$, todos os elementos são guardados em um único array, assim como em uma ArrayList. Entre esses dois extremos está um balanceamento entre o tempo que leva para adicionar ou remover um item da lista e o tempo que leva para localizar um dado item na lista.

3.4 Discussão e Exercícios

Ambas as listas simplesmente e duplamente encadeadas são técnicas bem estabelecidas e têm sido usadas em programas por mais de 40 anos. Elas são discutidas, por exemplo, por Knuth [46, Sections 2.2.3–2.2.5]. Mesmo a estrutura de dados SEList parece ser uma variante de estrutura de dados bem conhecida. A SEList é às vezes conhecida pelo nome de *unrolled linked list* [69].

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

Outra forma de economizar espaço em uma lista duplamente encadeada é usar uma XOR-list. Em uma XOR-list, cada nodo, u contém somente um ponteiro, chamado u.nextprev, que guarda o resultado do ou-exclusivo bit-a-bit entre u.prev e u.next. A lista em si precisa guardar dois ponteiros, um para o nodo dummy e um para dummy.next (o primeiro nodo, ou dummy se a lista está vazia). Essa técnica usa o fato que, se temos os ponteiros para u e u.prev, então podemos extrair u.next usando a fórmula

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

(Aqui ^ computa o ou-exclusivo bit-a-bit de seus dois argumentos.) Essa técnica complica o código um pouco e não é possível em algumas linguagem como Java e Python, que têm coletores de lixo, porém fornece uma implementação de uma lista duplamente encadeada que requer somente um ponteiro por nodo. Veja o artigo de Sinha [70] para uma discussão detalhada dessa XOR-list.

Exercício 3.1. Porque não é possível usar um nodo dummy em uma 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 novo método para a SLList chamado secondLast() que retorna o penúltimo elemento de uma SLList. Faça isso sem usar a variável membro, n, que registra o tamanho da lista.

Exercício 3.3. Implemente as operações de uma List get(i), set(i,x), add(i,x) e remove(i) em uma SLList. Cada uma dessas operações devem rodar tempo $O(1 + \texttt{i})$.

Exercício 3.4. Projete e implemente um método para a SLList chamado reverse() que inverte a ordem de elementos em uma SLList. Esse método deve rodar em tempo $O(\texttt{n})$, não deve usar recursão, não deve usar nenhuma estrutura de dados secundária e não deve criar nenhum nodo novo.

Exercício 3.5. Projete e implemente o método checkSize() para as estruturas SLList e DLList. Esse método percorre a lista e conta o número de nodos para verificar se ele é igual ao valor n guardado na lista. Esse método não retorna nada, mas lança uma exceção se o tamanho que ele computa não for idêntico ao valor de n.

Exercício 3.6. Tente recriar o código para a operação addBefore(w) que cria um nodo , u, o adiciona em uma DLList logo antes do nodo w. Não use o código deste capítulo. Mesmo se o seu código não seja exatamente igual ao código deste livro ele ainda pode estar correto. Teste e veja se funciona.

Os próximos exercícios envolvem a realização de manipulações em DLLists. Você deve realizá-los sem alocar novos nodos ou arrays temporários. Todos eles podem ser feitos apenas com trocas de valores de prev e next dos nodos existentes.

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

Exercício 3.8. Implemente um método rotate(r) que "rotaciona" uma DLList tal que o item i da lista se torna o item $(i + r) \bmod n$. Esse método deve rodar em $O(1 + \min\{r, n - r\})$ de tempo não deve modificar quaisquer nodos na lista.

Exercício 3.9. Escreva um método, truncate(i), que trunca uma DLList na posição i. Após executar esse método, o tamanho da lista será i e deve conter somente os elementos nos índices $0, \ldots, i - 1$. Esse método retorna outra DLList que contém os elementos nos índices $i, \ldots, n-1$. Esse método deve rodar em tempo $O(\min\{i, n - i\})$.

Exercício 3.10. Escreva um método para a DLList chamado absorb(l2), que recebe como argumento uma DLList, l2, a esvazia e coloca seu conteúdo, em ordem, na lista receptora. Por exemplo, se l1 contém a, b, c e l2 contém d, e, f, então após chamar l1.absorb(l2), l1 terá a, b, c, d, e, f e l2 estará vazia.

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

Exercício 3.12. Escreva um método `reverse()` que inverte a ordem de elementos em uma `DLList`.

Exercício 3.13. Este exercício te guia para realizar uma implementação do algoritmo merge-sort para a ordenação de uma `DLList`, conforme discutido na Seção 11.1.1. Na sua implementação, realize comparações entre elementos usando o método `compareTo(x)` de forma que a implementação resultante possa ordenar qualquer `DLList` contendo elementos que implementam a interface `Comparable`.

1. Escreva um método para a `DLList` chamado `takeFirst(12)`. Esse método recebe o primeiro nodo de 12 e o combina à lista que efetua o método. Isso é equivalente a `add(size(), 12.remove(0))`, exceto que não deve criar um nodo novo.

2. Escreva um método estático para a `DLList` chamado `merge(11,12)`, que recebe sua listas ordenadas 11 e 12, faz a junção (em inglês, merge) delas, e retorna uma nova lista ordenada contendo o resultado. Essa junção faz com que 11 e 12 sejam esvaziadas no processo. Por exemplo, se 11 contém a, c, d e 12 contém b, e, f, então esse método retorna uma nova lista contendo a, b, c, d, e, f.

3. Escreva um método para a `DLList` que ordena os elementos contidos na lista usando o algoritmo merge sort. Esse algoritmo recursivo funciona da seguinte maneira:

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

   (b) Usando o método `truncate(size()/2)`, divida a lista em duas listas de tamanhos aproximadamente iguais 11 e 12;

   (c) Recursivamente ordene 11;

   (d) Recursivamente ordene 12; e finalmente

   (e) Faça a junção (em inglês, merge) da 11 com 12 em uma única lista ordenada.

Os próximos exercícios são mais avançados e requerem a compreensão aprofundada do que acontece ao valor mínimo armazenado em uma `Stack` ou `Queue` quando itens são adicionados ou removidos.

Exercício 3.14. Projete e implemente uma estrutura de dados `MinStack` que pode guardar elementos comparáveis e possui as operações de stack `push(x)`, `pop()` e `size()`, assim como a operação `min()`, que retorna o valor mínimo atualmente guardado na estrutura de dados. Todas as operações devem rodar em tempo constante.

Exercício 3.15. Projete e implemente uma estrutura de dados `MinQueue` que pode guardar elementos comparáveis e possui as operações de queue `add(x)`, `remove()` e `size()`, assim como a operação `min()`, que retorna o valor mínimo atualmente armazenado na estrutura de dados. Todas as operações devem rodar em tempo constante amortizado.

Exercício 3.16. Projete e implemente uma estrutura de dados `MinDeque` que pode guardar elementos comparáveis e possui todas as operações da deque `addFirst(x)`, `addLast(x)`, `removeFirst()`, `removeLast()` e `size()`, e a operação `min()`, que retorna o valor mínimo atualmente guardado na estrutura de dados. Todas as operações devem rodas em tempo amortizado constante.

Os exercícios a seguir são planejados para testar a compreensão do leitor a respeito da implementação e análise da lista eficiente em uso do espaço `SEList`:

Exercício 3.17. Prove que, se uma `SEList` é usada como uma `Stack` (adicionado à `SEList` as operações de inserção `push(x)` $\equiv$ `add(size(),x)` e remoção `pop()` $\equiv$ `remove(size()` $-$ `1))`, então elas operações rodam em tempo constante amortizado, independentemente do valor de `b`.

Exercício 3.18. Projete e implemente uma versão de uma `SEList` que aceita todas as operações de uma `Deque` em tempo amortizado constante por operação, independentemente do valor de `b`.

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

Capítulo 4

Skiplists

Neste capítulo, discutimos uma elegante estrutura de dados: a skiplist, que tem uma variedade de aplicações. Usando uma skiplist podemos implementar uma `List` que tem implementações $O(\log n)$ de tempo para `get(i)`, `set(i,x)`, `add(i,x)` e `remove(i)`. Podemos também implementar um `SSet` no qual todas as operações rodam em tempo $O(\log n)$.

A eficiência de skiplist está no seu uso inteligente de decisões aleatórias. Quando um novo elemento é adicionado a uma skiplist, ela usa lançamentos de uma moeda para determinar a altura do novo elemento. O desempenho das skiplists é expresso em termos da esperança do tempo de execução e do comprimento de caminhos percorridos. Essa esperança, relativa ao valor esperado ou médio, é feita sobre os lançamentos da moeda usados pela skiplist. Na implementação, os lançamentos 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 ligadas $L_0, \ldots, L_h$. Cada lista L_r contém um subconjunto de itens em L_{r-1}. Iniciamos com a lista de entrada L_0 que contém `n` itens e construímos L_1 usando L_0, L_2 usando L_1 e assim por diante. Os itens em L_r são obtidos por lançamento de uma moeda para cada elemento `x` na L_{r-1} e incluindo `x` na L_r se a moeda sai do lado cara. Esse processo termina quando criamos uma lista L_r que está vazia. Um exemplo de uma skiplist é mostrado na

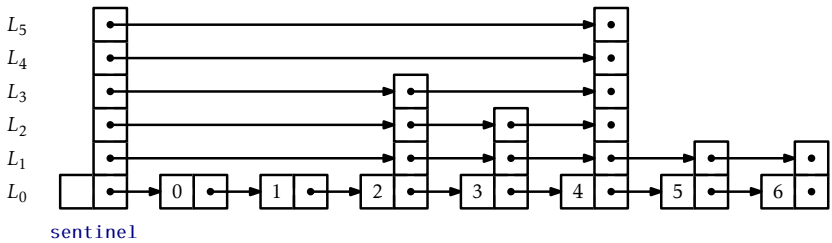


Figura 4.1: Uma skiplist com sete elementos.

Figura 4.1.

Para um elemento `x` em uma skiplist, chamamos de *altura* de `x` o maior valor r tal que `x` aparece na lista L_r. Então, por exemplo, elementos que somente aparecem em L_0 tem altura 0. Se pararmos alguns momentos para pensar sobre isso, notamos que a altura de `x` corresponde ao seguinte experimento: jogar uma moeda repetidamente até que saia coroa. Quantas vezes sai cara? A resposta, pouco surpreendentemente, é que a altura esperada de um nodo é 1. (Esperamos jogar a moeda duas vezes antes de sair coroa, mas não contamos o último lançamento.) A *altura* de uma skiplist é a altura do seu nodo mais alto.

A cabeça da lista toda é um nodo especial, chamado de *sentinela*, que atua como um nodo dummy para a lista. A propriedade chave das skiplists é que existe um caminho curto, chamado de *caminho de busca*, do sentinela em L_h para todo nodo em L_0. Como construir um caminho de busca para um nodo `u` é fácil (veja a Figura 4.2) : inicie no canto superior esquerdo da sua skiplist (o sentinela em L_h) e sempre vá à direita a menos que ultrapasse `u`, nesse caso você deve descer à lista logo abaixo.

Mais precisamente, para construir o caminho de busca para o nodo `u` em L_0, começamos no sentinela `w` em L_h. Depois examinamos `w.next`. Se `w.next` contém um item que aparece antes de `u` em L_0, então fazemos `w = w.next`. Caso contrário, seguimos a busca na lista abaixo e buscamos a ocorrência de `w` na lista

O resultado a seguir, que provaremos na Seção 4.4, mostra que o caminho de busca é bem curto:

Lema 4.1. *O comprimento esperado do caminho de busca para qualquer nodo*

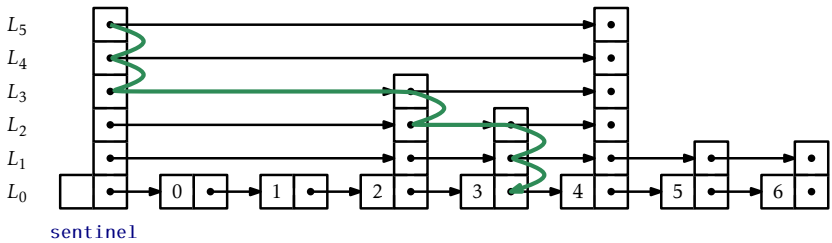


Figura 4.2: O caminho de busca para o nodo contendo 4 em uma skiplist.

`u` *em* L_0 *é no máximo* $2\log \mathtt{n} + O(1) = O(\log \mathtt{n})$.

Um modo de implementar uma skiplist que economiza memória é definir um `Node` `u` consistindo de um valor `x` e um array `next` de ponteiros onde `u.next[i]` aponta para o sucessor de `u` na lista $L_{\mathtt{i}}$. Desse jeito, o valor `x` em um nodo é referenciado somente uma vez `x` pode aparecer em várias listas.

SkiplistSSet

```
class Node<T> {
  T x;
  Node<T>[] next;
  Node(T ix, int h) {
    x = ix;
    next = Array.newInstance(Node.class, h+1);
  }
  int height() {
    return next.length - 1;
  }
}
```

As próximas duas seções deste capítulo discutem duas diferentes aplicações de skiplists. Em cada uma dessas aplicações, L_0 guarda a estrutura principal (uma lista de elementos ou um conjunto ordenado de elementos). A principal diferença entre essas estruturas está em como um caminho de busca é navegado; em particular, eles se diferem na forma de decidir se um caminho de busca deve para a lista abaixo L_{r-1} ou à direita em L_r.

4.2 SkiplistSSet: Um SSet Eficiente

Um SkiplistSSet usa uma estrutura de skiplist para implementar a interface SSet. Quando usada dessa maneira, a lista L_0 guarda os elementos de SSet de modo ordenado. O método find(x) funciona seguindo o caminho de busca para o menor valor y tal que $y \geq x$:

SkiplistSSet
```
Node<T> findPredNode(T x) {
  Node<T> 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<T> 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 nodo 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 a lista L_{r-1}. Cada passo (à direita ou abaixo) nessa busca leva apenas um tempo constante, então, pelo Lema 4.1, o tempo esperado de execução de find(x) é $O(\log n)$.

Antes de podermos adicionar um elemento a uma SkipListSSet, precisamos de um método para simular o lançamento de moedas para determinar a altura, k, de um novo nodo. Fazemos isso ao escolher um inteiro aleatório, z, e contar o número de 1s na representação binária de z:
[1]

[1] Esse método não replica exatamente o experimento de lançamento de moedas pois o valor de k será sempre menor que o número de bits em um int. Porém, esse fato terá um impacto muito pequeno a menos que o número de elementos em uma estrutura seja muito maior que $2^{32} = 4294967296$.

SkiplistSSet

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

Para implementar o método add(x) em uma SkiplistSSet procuramos por x e então dividimos x em algumas listas $L_0,\ldots,L_k$, onde k é selecionado usando o método pickHeight(). O jeito mais fácil de fazer isso é usar um array stack, que registra os nodos em que o caminho de busca desce de alguma lista L_r para L_{r-1}. Mais precisamente, stack[r] é o nodo em L_r onde o caminho de busca desceu em L_{r-1}. Os nodos que modificamos para inserir x são precisamente os nodos stack[0],...,stack[k]. O código a seguir implementa esse algoritmo para add(x):

SkiplistSSet

```
boolean add(T x) {
  Node<T> 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<T> w = new Node<T>(x, pickHeight());
  while (h < w.height())
    stack[++h] = sentinel;   // height increased
  for (int i = 0; i < w.next.length; i++) {
    w.next[i] = stack[i].next[i];
    stack[i].next[i] = w;
  }
```

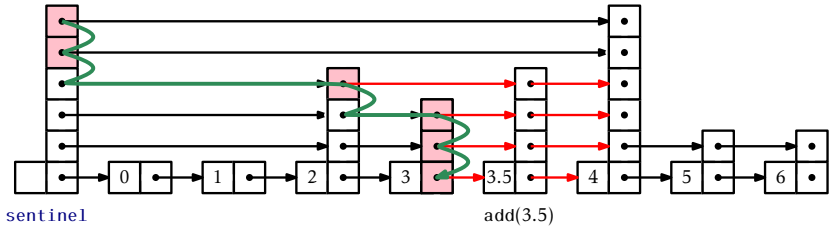


Figura 4.3: Adicionando o nodo contendo 3.5 a uma skiplist. Os nodos guardados em `stack` estão em destaque.

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

A remoção de um elemento `x` é feita de modo similar, exceto que não há necessidade para a `stack` registrar o caminho de busca. A remoção pode ser feita conforme vamos seguindo o caminho de busca. Buscamos por `x` e cada vez que a busca segue abaixo de um novo `u`, verificamos se `u.next.x` é igual a `x` e, caso positivo, tiramos `u` da lista:

SkiplistSSet

```
boolean remove(T x) {
  boolean removed = false;
  Node<T> 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) {
      removed = true;
      u.next[r] = u.next[r].next[r];
      if (u == sentinel && u.next[r] == null)
        h--;  // height has gone down
    }
    r--;
```

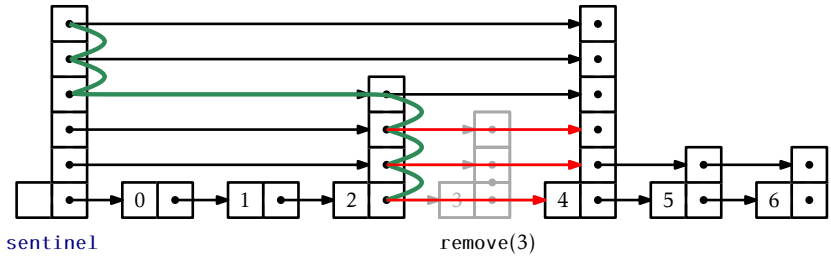


Figura 4.4: Remoção do nodo contendo 3 de uma skiplist.

```
  }
  if (removed) n--;
  return removed;
}
```

4.2.1 Resumo

O teorema a seguir resume o desempenho de skiplists quando usadas para implementar conjuntos ordenados:

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

4.3 `SkiplistList`: Uma `List` com 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 que eles aparecem na lista. Como em uma `SkiplistSSet`, elementos podem ser adicionados, removidos e acessados em $O(\log n)$ de tempo.

Para isso ser possível, precisamos de um jeito de seguir o caminho de busca para o `i`-ésimo em L_0. O jeito mais fácil de fazer isso é definir a noção de *comprimento* de uma aresta em alguma lista, $L_{\{r\}}$.

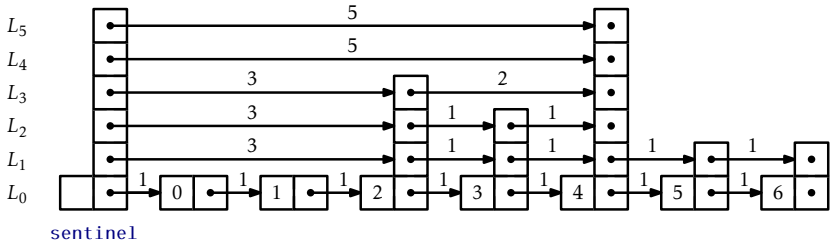


Figura 4.5: Os comprimentos das arestas em uma skiplist.

Definimos o comprimento de toda aresta em L_0 como 1. O comprimento de uma aresta, e, em L_r, $r > 0$, é definido como a soma dos comprimentos das arestas abaixo de e em L_{r-1}. De maneira equivalente, o comprimento de e é o número de arestas em L_0 abaixo de e. Veja a Figura 4.5 para um exemplo de uma skiplist com os comprimentos de suas arestas mostradas. Como as arestas de skiplists são guardadas em arrays, os comprimentos podem ser guardados do mesmo jeito:

SkiplistList

```
class Node {
  T x;
  Node[] next;
  int[] length;
  Node(T ix, int h) {
    x = ix;
    next = Array.newInstance(Node.class, h+1);
    length = new int[h+1];
  }
  int height() {
    return next.length - 1;
  }
}
```

Essa definição de comprimento tem a útil propriedade de que se estivermos atualmente em um nodo que está na posição j em L_0 e seguimos uma aresta de comprimento ℓ, então vamos para um nodo cuja posição, em L_0, é $j+\ell$. Desse jeito, enquanto seguirmos um caminho de busca, podemos rastrear a posição, j, do nodo atual em L_0. Quando em um nodo,

u, em L_r, iremos à direita se j mais o comprimento da aresta u.next[r] for menor que i. Caso contrário, descemos para L_{r-1}.

SkiplistList
```
Node findPred(int i) {
  Node u = sentinel;
  int r = h;
  int j = -1;   // 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;
}
```

Uma vez que a parte mais difícil das operações get(i) e set(i,x) é achar o i-ésimo nodo em L_0, essas operações rodam em tempo $O(\log n)$.

Adicionar um elemento na SkiplistList em uma posição i é razoavelmente simples. Diferentemente de uma SkiplistSSet, temos certeza que um novo nodo será realmente adicionado, então podemos fazer a adição ao mesmo tempo que procuramos pelo lugar de um novo nodo. Primeiro pegamos a altura k do nodo recentemente inserido w e então seguimos o caminho de busca para i. Toda vez que o caminho de busca descer a partir de L_r com $r \leq k$, inserimos w em L_r. O último cuidado extra ne-

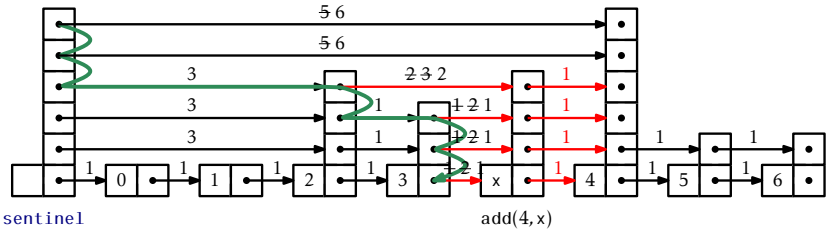


Figura 4.6: Adicionando um elemento a uma SkiplistList.

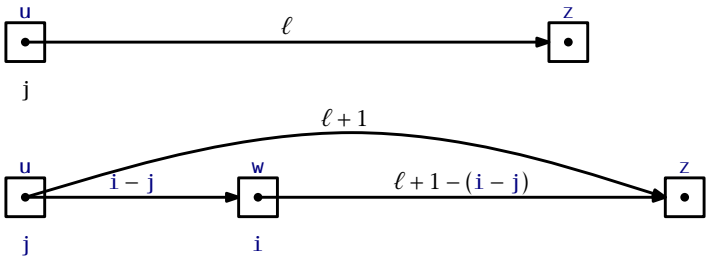


Figura 4.7: Atualizando os comprimentos das arestas ao dividir um nodo w de uma skiplist.

cessário é assegurar que os comprimentos das arestas sejam atualizados corretamente. Veja a Figura 4.6.

Note que, cada vez que o caminho de busca desce em um nodo u em L_r, o comprimento da aresta u.next[r] aumenta em um, pois estamos adicionando um elemento abaixo daquela aresta na posição i.

A divisão, em inglês *splice*, do nodo w em dois nodos, u e z funciona como mostrado na Figura 4.7. Ao seguir o caminho de busca já estamos guardando a posição 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 a partir do comprimento ℓ de uma aresta de u a z. Portanto, podemos dividir em w e atualizar os comprimentos das arestas em tempo constante.

Isso parece ser mais complicado do que realmente é, porque o código é bem simples:

```
SkiplistList
void add(int i, T x) {
  Node w = new Node(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]++; // accounts for new node in list 0
    if (r <= k) {
      w.next[r] = u.next[r];
      u.next[r] = w;
      w.length[r] = u.length[r] - (i - j);
      u.length[r] = i - j;
    }
    r--;
  }
  n++;
  return u;
}
```

Agora a implementação da operação `remove(i)` em uma `SkiplistList` deve ser óbvia. Seguimos o caminho de pesquisa para o nodo na posição `i`. Cada vez que o caminho de busca desce a partir de um nodo `u` no nível `r` decrementamos o comprimento da aresta que deixa `u` naquele nível. Também verificamos se `u.next[r]` é o elemento do `i` e, se o for, o removemos da lista naquele nível. Um exemplo é mostrado na Figura 4.8.

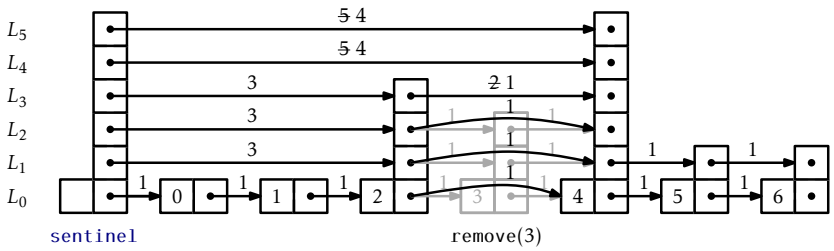


Figura 4.8: Remoção de um elemento de uma SkiplistList.

SkiplistList

```
T remove(int i) {
  T x = null;
  Node u = sentinel;
  int r = h;
  int j = -1; // index of node 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]--;  // 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];
      u.next[r] = u.next[r].next[r];
      if (u == sentinel && u.next[r] == null)
        h--;
    }
    r--;
  }
  n--;
  return x;
}
```

4.3.1 Resumo

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

Teorema 4.2. *Uma* `SkiplistList` *implementa a interface* `List`*. Uma* `SkiplistList` *possui a operações* `get(i)`, `set(i,x)`, `add(i,x)` *e* `remove(i)` *em tempo esperado* $O(\log n)$ *por operação.*

4.4 Análise de Skiplists

Nesta seção, analisaremos a altura esperada, o tamanho e o comprimento do caminho de busca em uma skiplist. Esta seção requer conhecimentos básicos de probabilidades. Várias provas são baseadas na seguinte obervação básica sobre lançamentos de moedas.

Lema 4.2. *Seja T o número de vezes que uma moeda honesta é lançada, incluindo a primeira vez que a moeda saiu com a face cara. Então* $\mathrm{E}[T] = 2$.

Demonstração. Suponha que paramos de lançar a moeda na primeira vez que sai cara. Defina a variável indicadora

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

Note que $I_i = 1$ se, e somente se, os primeiros $i-1$ lançamentos saem coroa, então $\mathrm{E}[I_i] = \Pr\{I_i = 1\} = 1/2^{i-1}$. Observe que T, o número total de lançamentos, 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 próximos dois lemas afirmam que skiplists têm tamanho linear:

Lema 4.3. *O número esperado de nodos em uma skiplist com* n *elementos, sem incluir as ocorrências da sentinela, é* $2n$.

Demonstração. A probabilidade que qualquer elemento em particular, x, seja incluído na lista L_r é $1/2^r$, então o número esperado de nodos em L_r é $n/2^r$.[2] Portanto, o número total esperado de nodos 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 *elementos é de até* $\log n + 2$.

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

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

A altura h da skiplist é então dada por

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

Note que I_r nunca é maior que o comprimento, $|L_r|$, de L_r, então

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

Portanto, temos

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

[2] Veja a Seção 1.3.4 para ver como isso é obtido usando variáveis indicadores e linearidade de esperança.

$$
\begin{aligned}
&\leq \sum_{r=1}^{\lfloor \log n \rfloor} 1 + \sum_{r=\lfloor \log n \rfloor + 1}^{\infty} n/2^r \\
&\leq \log n + \sum_{r=0}^{\infty} 1/2^r \\
&= \log n + 2 \ .
\end{aligned}
$$

□

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

Demonstração. Segundo o Lema 4.3, o número esperado de nodos, sem incluir o sentinela, é $2n$. O número de ocorrências do sentinela é igual à altura h da skiplist então, segundo o Lema 4.4, o número esperado de ocorrências da sentinela é de até $\log n + 2 = O(\log n)$. □

Lema 4.6. *O comprimento esperado de um caminho de busca em uma skiplist é de até* $2\log n + O(1)$.

Demonstração. O jeito mais fácil de ver isso é considerar o *caminho de busca reverso*. Esse caminho inicia-se no predecessor de x em L_0. Em qualquer momento, se o caminho puder subir um nível, então ele o faz. Se ele não puder subir um nível, então vai à esquerda. Considerando isso por alguns momentos vemos que o caminho de busca reverso para x é idêntico ao caminho de busca para x, exceto que é invertido.

O número de nodos que o caminho de busca reverso visita em um dado nível r é relacionado ao seguinte experimento: lance uma moeda. Se a moeda sair cara, então suba um nível e pare. Caso contrário, vá à esquerda e repita o experimento. O número de lançamentos obtido dessa forma representa o número de passos à esquerda que um caminho de busca invertido faz em um dado nível. [3] O Lema 4.2 afirma que o número esperado de lançamentos de moedas antes da primeira cara é 1. Seja S_r o número de passos que um caminho de busca (direta) usa no nível r para ir à direita. Acabamos de mostrar que $E[S_r] \leq 1$. Além disso, $S_r \leq |L_r|$,

[3] Note que isso pode superestimar o número de passos à esquerda pois o experimento deve terminar em um lançamento cara ou quando o caminho de busca chega no sentinela, aquele que vier primeiro. Isso não é um problema porque o lema está somente apresentando um limitante superior.

pois não podemos fazer mais passos em L_r que o próprio comprimento de L_r, então

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

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

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

O teorema a seguir resume os resultados nessa seção:

Teorema 4.3. *Uma skiplist contendo n elementos tem tamanho esperado $O(n)$ e o comprimento esperado do caminho de busca para qualquer elemento é de até $2 \log n + O(1)$.*

4.5 Discussão e Exercícios

Skiplists foram inicialmente propostas por [62] que também apresentou várias aplicações e extensões de skiplists [61]. Desde então, ela tem

sido extensivamente estudada. Vários pesquisadores têm realizado análises bem precisas do comprimento esperado e da variância do comprimento do caminho de busca para o `i`-ésimo elemento em uma skiplist [45, 44, 58]. Versões determinísticas [53], versões tendenciosas [8, 26], e versões auto ajustáveis [12] de skiplists têm surgido. Implementações de skiplists têm sido escritas para várias linguagens e bibliotecas e usadas em sistemas de bancos de dados open-source [71, 63]. Uma variante de skiplist é usada nas estruturas do gerenciador de processos do kernel do sistema operacional HP-UX. [42]. Skiplists são até parte da Java 1.6 API [55].

Exercício 4.1. Desenhe os caminhos de busca para 2.5 e 5.5 na skiplist da Figura 4.1.

Exercício 4.2. Desenhe a adição de valores 0.5 (com altura de 1) e então 3.5 (com uma altura de 2) para a skiplist da Figura 4.1.

Exercício 4.3. Desenhe a remoção de valores 1 e então 3 da skiplist na Figura 4.1.

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

Exercício 4.5. Desenhe a execução de `add(3, x)` na `SkiplistList` da Figura 4.5. Assuma que `pickHeight()` seleciona uma altura de 4 para o nodo recentemente criado.

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

Exercício 4.7. Suponha que, em vez de promover um elemento de L_{i-1} em L_i usando um lançamento de moeda , fazemos a promoção com uma probabilidade p, $0 < p < 1$.

1. Mostre que, com essa modificação, o comprimento esperado de um caminho de busca é até $(1/p)\log_{1/p} \mathsf{n} + O(1)$.

2. Qual é o valor esperado de p que minimiza a expressão precedente?

3. Qual é altura esperada da skiplist?

4. Qual é o número esperado de nodos na skiplist?

Exercício 4.8. O método `find(x)` em uma `SkiplistSet` às vezes faz *comparações redundantes*; essas ocorrem quando `x` é comparado ao mesmo valor mais de uma vez. Elas podem ocorrer quando, para algum nodo `u` `u.next[r] = u.next[r − 1]`. Mostre quando essas comparações redundantes acontecem e modifique `find(x)` para que sejam evitadas. Analise o número esperado de comparações feitas pelo seu método modificado `find(x)`.

Exercício 4.9. Projete e implemente uma versão de uma skiplist que implemente a interface `SSet`, mas também permite acesso rápido a eles pelo rank. Isto é, também aceita a função `get(i)`, que retorna o elemento cujo rank é `i` `i` em $O(\log n)$ de tempo esperado. (O rank de um elemento `x` em uma `SSet` é o número de elementos em `SSet` que são menores que `x`.)

Exercício 4.10. Um *finger* em uma skiplist é um array que guarda a sequência de nodos em um caminho de busca no qual o caminho de busca desce. (A variável `stack` no código `add(x)` na página 95 é um finger; os nodos destacados na Figura 4.3 mostram o conteúdo do finger.) Pode-se pensar de um finger como uma forma de apontar o caminho para um nodo na lista mais baixa, L_0.

Uma *busca finger* implementa a operação `find(x)` usando um finger, ao percorrer a lista usando o finger até chegar em um nodo `u` tal que `u.x < x` e `u.next = null` ou `u.next.x > x` e então fazer uma busca normal por `x` começando de `u`. É possível provar que o número esperado de passos necessários para uma busca finger é $O(1 + \log r)$, onde r é o número de valores em L_0 entre `x` e o valor apontado pelo finger.

Implemente uma subclasse de `Skiplist` chamada `SkiplistWithFinger` que implementa a operação `find(x)` usando um finger interno. Essa subclasse guarda um finger, que é então usado para que toda operação `find(x)` seja implementada como uma busca finger. Durante cada operação `find(x)` o finger é atualizado tal que cada operação `find(x)` usa como ponto de partida um finger que aponta para o resultado da operação `find(x)` prévia.

Exercício 4.11. Escreva um método `truncate(i)` que trunca uma `SkiplistList` na posição `i`. Após a execução desse método, o tamanho da

lista é i e contém somente os elementos nos índices $0,\ldots,i-1$. O valor retornado é outra SkiplistList que contém os elementos nos índices $i,\ldots,n-1$. Esse método deve rodar em tempo $O(\log n)$.

Exercício 4.12. Escreva um método para a SkiplistList chamado absorb(12), que recebe como argumento uma SkiplistList, 12, a esvazia e coloca seu conteúdo na estrutura receptora. Por exemplo, se 11 contém a,b,c e 12 contém d,e,f, então após chamar 11.absorb(12), 11 conterá a,b,c,d,e,f e 12 estará vazia. Esse método deve rodar em $O(\log n)$ de tempo.

Exercício 4.13. Usando as ideias da lista eficiente em espaço, SEList, projete e implemente uma SSet eficiente em espaço SSets, SESSet. Para isso, guarde os dados, em ordem, em uma SEList e armazene os blocos dessa SEList em uma SSet. Se a implementação original SSet usa $O(n)$ de espaço para guardar n elementos, então a SESSet irá usar espaço suficiente para n elementos mais $O(n/b+b)$ de espaço desperdiçado.

Exercício 4.14. Usando uma SSet como estrutura básica, projete e implemente uma aplicação que lê um arquivo de texto (potencialmente enorme) e permite você buscar, iterativamente, por qualquer substring contida no texto. Conforme o usuário digita a consulta, uma parte do texto que corresponde à busca (se houver) deve aparecer como resultado.
Dica 1: toda substring é um prefixo de algum sufixo, então basta guardar todos os sufixos do arquivo de texto.
Dica 2: qualquer sufixo pode ser representado como um único inteiro indicando onde o sufixo começa no texto.
Teste sua aplicação em textos maiores, tais como alguns livros disponibilizados pelo Projeto Gutenberg [1]. Se feito corretamente, a sua aplicação será bem rápida; não deve haver atraso considerável entre digitar letras e ver os resultados.

Exercício 4.15. (Este exercício deve ser feito após a leitura sobre árvores binárias de busca na Seção 6.2.) Compare as skiplists com árvores binárias de busca das seguintes maneiras:

1. Explique como algumas arestas de uma skiplist levam a uma estrutura que parece uma árvore binária e é similar a uma árvore de busca binária.

2. Skiplists e árvores de busca binária usam aproximadamente o mesmo número de ponteiros (2 por nodo). Contudo, skiplists fazem melhor uso desses ponteiros. Explique porque.

Capítulo 5

Tabelas Hash

Tabelas hash são um eficiente método de guardar um pequeno número n de inteiros provenientes de um grande intervalo $U = \{0, \ldots, 2^w - 1\}$. O termo tabela hash, ou *hash table*, inclui um grande número de tipos de estruturas de dados. A primeira parte deste capítulo foca em duas das implementações mais comuns de tabelas hash: hashing com encadeamento e sondagem linear.

Muito frequentemente, tabelas hash guardam tipos de dados que não são inteiros. Nesse caso, um inteiro chamado de código *hash* é associado com cada item de dados e é usado na tabela hash. A segunda parte deste capítulo discute como tais códigos hash são gerados.

Alguns desses métodos que usados neste capítulo requerem obter inteiros aleatórios em algum intervalo. Nos trechos de código, alguns desses inteiros "aleatórios" são constantes fixas no código. Essas constantes foram obtidas usando bits aleatórios gerados a partir de ruído atmosférico.

5.1 ChainedHashTable: Hashing com Encadeamento

Uma estrutura de dados ChainedHashTable usa *hashing com encadeamento* para guardar dados como um array t de listas. Um inteiro n registra o número total de itens em todas as listas. (ver a Figura 5.1):

ChainedHashTable

```
List<T>[] t;
int n;
```

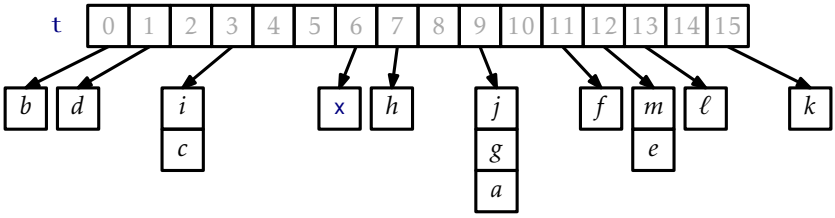


Figura 5.1: Uma exemplo de `ChainedHashTable` com $n = 14$ e $t.length = 16$. Nesse exemplo $hash(x) = 6$

O valor *hash* de um item de dados `x`, denotado `hash(x)` é um valor no intervalo $\{0,\ldots,t.length-1\}$. Todos os itens com um valor hash `i` são guardados na lista em `t[i]`. Para garantir que as listas não ficam muito longas, mantemos a invariante

$$n \le t.length$$

tal que o número médio de elementos guardados em uma dessas listas é $n/t.length \le 1$.

Para adicionar um elemento `x` à tabela primeiro verificamos se o tamanho de `t` precisar ser aumentado e, se precisar, expandimos `t`. Com isso resolvido, calculamos a hash de `x` para obter um inteiro `i` no intervalo $\{0,\ldots,t.length-1\}$ e adicionamos `x` à lista `t[i]`:

ChainedHashTable

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

Expandir a tabela, caso necessário, envolve dobrar o comprimento de `t` e reinserir todos os elementos na nova tabela. Essa estratégia é exatamente a mesma que aquela usada na implementação do `ArrayStack` e o mesmo resultado se aplica. O custo de expandir somente é visto

como constante quando amortizado sobre uma sequência de inserções (ver o Lema 2.1 na página 36).

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

Para remover um elemento x da tabela hash, iteramos sobre a lista t[hash(x)] até acharmos x tal que podemos removê-lo:

ChainedHashTable

```
T remove(T x) {
  Iterator<T> it = t[hash(x)].iterator();
  while (it.hasNext()) {
    T y = it.next();
    if (y.equals(x)) {
      it.remove();
      n--;
      return y;
    }
  }
  return null;
}
```

Isso leva $O(n_{\mathtt{hash}(x)})$ de tempo, onde n_i denota o comprimento da lista guardada em t[i].

Buscar pelo elemento x x em uma tabela hash é similar. Fazemos uma busca linear na lista t[hash(x)]:

ChainedHashTable

```
T find(Object x) {
  for (T y : t[hash(x)])
    if (y.equals(x))
      return y;
  return null;
}
```

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

O desempenho de uma tabela hash depende criticamente na escolha da função hash. Uma boa função hash irá espalhar os elementos de modo uniforme entre as `t.length` listas, tal que o tamanho esperado da lista $t[\mathsf{hash}(x)]$ é $O(n/t.\mathsf{length}) = O(1)$. Por outro lado, uma função hash ruim irá distribuir todos os valores (incluindo `x`) à mesma posição da tabela. Nesse caso, o tamanho da lista $t[\mathsf{hash}(x)]$ será `n`. Na seção a seguir descrevemos uma boa função hash.

5.1.1 Hashing Multiplicativo

Hashing multiplicativo é um método eficiente de gerar valores hash baseado em aritmética modular (discutido na Seção 2.3) e divisão inteira. Ele usa o operador div, que calcula a parte inteira de um quociente e descarta o resto. Formalmente, para quaisquer inteiros $a \geq 0$ e $b \geq 1$, $a \operatorname{div} b = \lfloor a/b \rfloor$.

Em hashing multiplicativo, usamos uma tabela hash de tamanho 2^d para um inteiro `d` (chamado de *dimensão*). A fórmula para aplicar hashing em um inteiro $x \in \{0, \ldots, 2^w - 1\}$ é

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

Aqui, `z` é um inteiro ímpar escolhido aleatoriamente em $\{1, \ldots, 2^w - 1\}$. Essa função hash pode ser computada muito eficientemente ao observar que, por padrão, operações em inteiros são módulo 2^w onde `w` é o número de bits em um inteiro.[1] (Ver a Figura 5.2.) Além disso, divisão inteira por 2^{w-d} é equivalente a descartar os $w - d$ bits mais à direita em uma representação binária (o que é implementado usando o deslocamento de bits $w - d$ à direita usando o operador »>). Dessa forma, o código que implementa a fórmula acima é mais simples que a própria fórmula:

ChainedHashTable

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

[1] Isso é verdade para a maior parte de linguagens de programação incluindo C, C#, C++, e Java. Exceções notáveis são Python e Ruby, nas quais o resultado de uma operação inteira de tamanho fixo `w`-bit que passa do intervalo permitido é convertido a uma variável com tamanho de representação variável.

2^w (4294967296)	100000000000000000000000000000000
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 da função de hashing multiplicativo com $w = 32$ e $d = 8$.

O lema a seguir, cuja prova será feita posteriormente nesta seção, mostra que o hashing multiplicativo faz um bom trabalho em evitar colisões:

Lema 5.1. *Sejam* 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* 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. Seja S o conjunto de elementos guardado na tabela hash que não são iguais a x. Para um elemento $y \in S$, defina a variável indicadora

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

e note que, segundo o 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 necessário. □

Agora, queremos provar o Lema 5.1, mas primeiro precisamos de um resultado da Teoria de Números. Na prova a seguir, 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 bit, 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 valor de bit desconhecido.

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

Demonstração. Como o número de escolhas para $\mathtt{z}$ e i é o mesmo, é suficiente provar que há *no máximo* um valor $\mathtt{z} \in S$ que satisfaz $\mathtt{z}q \bmod 2^{\mathtt{w}} = i$.

Suponha, para efeitos de uma contradição, que há dois tais valores $\mathtt{z}$ e $\mathtt{z}'$, com $\mathtt{z} > \mathtt{z}'$. Portanto

$$\mathtt{z}q \bmod 2^{\mathtt{w}} = \mathtt{z}'q \bmod 2^{\mathtt{w}} = i$$

Então

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

Mas isso significa que

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

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

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

tal que os últimos $\mathtt{w}$ bits na representação binária de $(\mathtt{z}-\mathtt{z}')q$ são todos 0.

Além disso, $k \neq 0$, pois $q \neq 0$ e $\mathtt{z}-\mathtt{z}' \neq 0$. Uma vez que q é ímpar, não há um 0 terminando sua representação binária:

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

Como $|\mathtt{z}-\mathtt{z}'| < 2^{\mathtt{w}}$, $\mathtt{z}-\mathtt{z}'$ tem menos que $\mathtt{w}$ bits no final de sua representação binária:

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

Portanto, o produto $(z - z')q$ tem menos que w bits 0 no término da sua representação binária:

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

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

A utilidade do Lema 5.3 vem da seguinte observação: Se z for escolhido de forma uniformemente aleatória de S, então zt é uniformemente distribuído sobre S. Na prova a seguir, ajuda a lembrar da representação binária de z, que consiste de $w - 1$ bits aleatórios seguidos por um 1.

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

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

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

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

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

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

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

Podemos agora terminar a prova: se $r > \mathtt{w}-\mathtt{d}$, então os $\mathtt{d}$ bits de alta ordem de $\mathtt{z}(\mathtt{x}-\mathtt{y}) \bmod 2^{\mathtt{w}}$ contêm tanto 0 quanto 1, tal que a probabilidade de que $\mathtt{z}(\mathtt{x}-\mathtt{y}) \bmod 2^{\mathtt{w}}$ sejam (5.2) ou (5.3) é 0.

Se $\mathtt{r} = \mathtt{w}-\mathtt{d}$, então a probabilidade de ocorrer (5.2) é 0, mas a probabilidade de ocorrer (5.3) é $1/2^{\mathtt{d}-1} = 2/2^{\mathtt{d}}$ (como precisamos ter $b_1,\ldots,b_{d-1} = 1,\ldots,1$). Se $r < \mathtt{w}-\mathtt{d}$, então precisamos ter $b_{\mathtt{w}-r-1},\ldots,b_{\mathtt{w}-r-\mathtt{d}} = 0,\ldots,0$ ou $b_{\mathtt{w}-r-1},\ldots,b_{\mathtt{w}-r-\mathtt{d}} = 1,\ldots,1$. A probabilidade de cada um desses casos é $1/2^{\mathtt{d}}$ e eles são mutualmente exclusivos, então a probabilidade de cada desses casos é $2/2^{\mathtt{d}}$. Isso completa a prova. □

5.1.2 Resumo

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

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

Além disso, começando com uma `ChainedHashTable` vazia, uma sequência de m operações `add(x)` e `remove(x)` resulta em um total de $O(m)$ de tempo gasto durante todas a chamadas a `grow()`.

5.2 `LinearHashTable`: Sondagem Linear

A estrutura de dados `ChainedHashTable` usa um array de listas onde a `i`-ésima lista guarda todos os elementos `x` tais que $\mathtt{hash}(\mathtt{x}) = \mathtt{i}$. Uma outra opção de projeto, chamado de *endereçamento aberto*, é guardar elementos diretamente em um array `t` sendo cada posição `t` para armazenar no máximo um valor. Essa abordagem é implementada pela `LinearHashTable` descrita nesta seção. Em alguns lugares, essa estrutura de dados é descrita como *endereçamento aberto com sondagem linear*.

A principal ideia por trás da `LinearHashTable` é que deveríamos, idealmente, guardar o elemento `x` com valor hash $\mathtt{i} = \mathtt{hash}(\mathtt{x})$ na posição da tabela $\mathtt{t}[\mathtt{i}]$. Se não podemos fazer isso (porque algum elemento já está armazenado lá) então tentamos armazená-lo na posição $\mathtt{t}[(\mathtt{i}+1) \bmod$

t.length]; se isso não for possível, então tentamos t[(i+2) mod t.length], e assim por diante, até encontrarmos um lugar para x.

Há três tipos de entradas guardadas em t:

1. valores de dados: valores reais no USet que estamos representando;

2. valores null: em posições do array onde nenhum valor foi guardado antes; e

3. valores del: em posições do array onde dados foram previamente guardados mas que desde então foram deletados.

Em adição ao contador n que registra o número de elementos na LinearHashTable, um contador, q, registra o número de elementos dos Tipos 1 e 3. Isso é, q é igual a n mais o número de valores del em t. Para fazer isso eficientemente, precisamos que t seja consideravelmente maior que q, tal que haja muitos null em t. As operações em LinearHashTable portanto mantém a invariante $t.length \geq 2q$.

Para resumir, uma LinearHashTable contém um array t que guarda elementos de dados e inteiros n e q que registram o número de elementos de dados e de valores não-null de t respectivamente. Como muitas funções hash somente funcionam para tamanhos de tabela que são potência de 2, também mantemos um inteiro d e a invariante $t.length = 2^d$.

LinearHashTable

```
T[] t;    // the table
int n;    // the size
int d;    // t.length = 2^d
int q;    // number of non-null entries in t
```

A operação find(x) na LinearHashTable é simples. Iniciamos na posição do array t[i] onde i = hash(x) e buscamos as posições t[i], t[(i + 1) mod t.length], t[(i + 2) mod t.length], e assim segue, até encontrarmos um índice i′ tal que t[i′] = x ou t[i′] = null. No primeiro caso retornamos t[i′]. No outro, quando t[i′] = null, concluímos que x não está contido na tabela hash e retornamos null.

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

A operação add(x) também é razoavelmente simples para implementar. Após verificar que x não está na tabela (usando find(x)), procuramos em t[i], t[(i + 1) mod t.length], t[(i + 2) mod t.length], e assim por diante, até acharmos um null ou del e guardamos x naquela posição, incrementando n e q, se apropriado.

LinearHashTable
```
boolean 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;
}
```

Até agora, a implementação da operação remove(x) deve ser óbvia. Buscamos t[i], t[(i + 1) mod t.length], t[(i + 2) mod t.length], e assim por diante até acharmos um índice i′ tal que t[i′] = x ou t[i′] = null. No primeiro caso, atribuímos t[i′] = del e retornamos true. No segundo caso, concluímos que x não foi armazenado na tabela (e portanto não pode ser deletado) e retornamos false.

LinearHashTable
```
T remove(T x) {
  int i = hash(x);
```

```
  while (t[i] != null) {
    T y = t[i];
    if (y != del && x.equals(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 corretude dos métodos `find(x)`, `add(x)` e `remove(x)` é fácil de verificar embora dependa do uso dos valores `del`. Note que nenhuma dessas operações nunca atribui `null` a uma posição diferente de `null`. Portanto, se alcançarmos um índice i' tal que $t[i'] = null$ isso comprova que o elemento x pelo qual buscamos não está guardado na tabela ; $t[i']$ sempre foi `null`, então não há porque uma operação prévia `add(x)` continuar além do índice i'.

O método `resize()` é chamado por `add(x)` quando o número de posições não-`null` exceder `t.length`/2 ou por `remove(x)` quando o número de entrada de dados for bem menor que `t.length`/8. O método `resize()` funciona como os métodos `resize()` em outras estruturas de dados baseadas em array. Achamos o menor inteiro não negativo d tal que $2^d \geq 3n$. Realocamos o array t tal que tenha tamanho 2^d, e então inserimos todos os elementos na versão antiga de t na nova cópia redimensionada de t. Ao fazer isso, reiniciamos q igual a n pois o novo t contém nenhum valor `del`.

LinearHashTable

```
void resize() {
  d = 1;
  while ((1<<d) < 3*n) d++;
  T[] told = t;
  t = newArray(1<<d);
  q = n;
  // insert everything from told
  for (int k = 0; k < told.length; k++) {
```

```
    if (told[k] != null && told[k] != del) {
      int i = hash(told[k]);
      while (t[i] != null)
        i = (i == t.length-1) ? 0 : i + 1;
      t[i] = told[k];
    }
  }
}
```

5.2.1 Análise da Sondagem Linear

Note que cada operação add(x), remove(x) ou find(x), termina assim que (ou mesmo antes) encontra a primeira posição null em t. A intuição por trás da análise da sondagem linear é que, como pelo menos metade dos elementos em t são iguais a null, uma operação não deve demorar muito para completar pois irá rapidamente achar uma posição com valor null. Não devemos usar essa intuição ao pé da letra pois levaria à conclusão (incorreta) de que o número esperado de posições em t que foram examinadas por uma operação é de até 2.

Para o resto desta seção, iremos assumir que todos os valores hash são independentemente e uniformemente distribuídos em $\{0,\ldots,$ t.length $-1\}$. Essa não é uma premissa realista, mas permitirá uma análise da sondagem linear. Mais a seguir, nesta seção, iremos descrever um método, chamado de hashing de tabulação, que produz uma função hash que é "boa o suficiente" para sondagem linear. Também isso assumir que todos os índices das posições de t são obtidos módulo t.length, tal que t[i] é na verdade um atalho para t[i mod t.length].

Dizemos que um *agrupamento de comprimento k iniciando em* i ocorre quando as posições da tabela $t[i], t[i+1], \ldots, t[i+k-1]$ não são null e $t[i-1] = t[i+k] =$ null. O número de elementos não-null de t é exatamente q e o método add(x) garante que, a qualquer momento, $q \leq$ t.length/2. Há q elementos $x_1, \ldots, x_q$ que foram inseridos em t desde a última operação resize(). De acordo com nossa premissa, cada um desses elementos tem valor hash hash(x_j) que é de uma distribuição uniforme e cada qual é independente do resto. Nesse cenário podemos provar o principal lema necessário para analisar a sondagem linear.

Lema 5.4. *Fixe um valor* $\mathtt{i} \in \{0, \ldots, \mathtt{t.length} - 1\}$. *Então a probabilidade de que um agrupamento de comprimento* k *inicie-se em* $\mathtt{i}$ *é* $O(c^k)$ *para uma constante* $0 < c < 1$.

Demonstração. Se um agrupamento de comprimento k se iniciar em $\mathtt{i}$, então haverá exatamente $\mathtt{k}$ elementos $\mathtt{x}_j$ tais que $\mathtt{hash}(\mathtt{x}_j) \in \{\mathtt{i}, \ldots, \mathtt{i}+k-1\}$. A probabilidade disso ocorrer é exatamente

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

pois para alguma escolha de k elementos, eles precisam mapear com a hash para uma dessas k posições e o restante dos $\mathtt{q}-k$ elementos precisa mapear para as outras $\mathtt{t.length}-k$ posições na tabela.
[2]

Na próxima derivação iremos trapacear um pouco e substituir $r!$ com $(r/e)^r$. A aproximação de Stirling (Seção 1.3.2) mostra que isso difere apenas $O(\sqrt{r})$ da verdade. Fazemos isso para simplificar a derivação; o Exercício 5.4 pede ao leitor refazer os cálculos de maneira mais rigorosa usando a aproximação de Stirling completa.

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

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

[2] Note que p_k é maior que a probabilidade de que um agrupamento de tamanho k inicie em $\mathtt{i}$, pois a definição de $\mathtt{p_k}$ não inclui o requisito $\mathtt{t}[\mathtt{i}-1] = \mathtt{t}[\mathtt{i}+k] = \mathtt{null}$.

$$
\begin{aligned}
&= \left(\frac{1}{2}\right)^k \left(1 + \frac{k}{2(\mathtt{q}-k)}\right)^{\mathtt{q}-k} \\
&\le \left(\frac{\sqrt{e}}{2}\right)^k \quad .
\end{aligned}
$$

(No último passo, usamos a desigualdade $(1+1/x)^x \le e$, que vale para todo $x > 0$.) Como $\sqrt{e}/2 < 0.824360636 < 1$, isso completa a prova. □

Usar o Lema 5.4 para provar limitantes superiores no tempo de execução esperado de `find(x)`, `add(x)` e `remove(x)` agora é mais simples. Considere o caso mais simples, onde executamos `find(x)` para algum valor `x` nunca foi guardado em `LinearHashTable`. Nesse caso, `i` = `hash(x)` é uma variável aleatória em $\{0,\ldots,\mathtt{t.length}-1\}$ independente do conteúdo de `t`. Se `i` é parte de um agrupamento de tamanho k, então o tempo que leva para executar a operação `find(x)` é até $O(1+k)$. Então, o tempo esperado de execução pode ser limitado superiormente por

$$
O\left(1 + \left(\frac{1}{\mathtt{t.length}}\right) \sum_{i=1}^{\mathtt{t.length}} \sum_{k=0}^{\infty} k \Pr\{\mathtt{i} \text{ está em um agrupamento de tamanho } k\}\right).
$$

Note que cada agrupamento de tamanho k contribui à soma interna k vezes para uma contribuição total de k^2, então a soma anterior pode ser reescrita como

$$
\begin{aligned}
&O\left(1 + \left(\frac{1}{\mathtt{t.length}}\right) \sum_{i=1}^{\mathtt{t.length}} \sum_{k=0}^{\infty} k^2 \Pr\{\mathtt{i} \text{ inicia um agrupamento de tamanho } k\}\right) \\
&\le O\left(1 + \left(\frac{1}{\mathtt{t.length}}\right) \sum_{i=1}^{\mathtt{t.length}} \sum_{k=0}^{\infty} k^2 p_k\right) \\
&= O\left(1 + \sum_{k=0}^{\infty} k^2 p_k\right) \\
&= O\left(1 + \sum_{k=0}^{\infty} k^2 \cdot O(c^k)\right) \\
&= O(1) \quad .
\end{aligned}
$$

O último passo dessa derivação vem do fato que $\sum_{k=0}^{\infty} k^2 \cdot O(c^k)$ é uma série

exponencial decrescente.[3]

Portanto, concluímos que o tempo de execução esperado da operação `find(x)` para um valor `x` que não é contido em uma `LinearHashTable` é $O(1)$.

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

Primeiramente, a análise de `find(x)` dada acima aplica-se à operação `add(x)` quando`x` não está na tabela. Para analisar a operação `find(x)` quando `x` está na tabela, precisamos somente notar que isso tem o mesmo custo da operação `add(x)` que anteriormente adicionou `x` à tabela. Finalmente, o custo de uma operação `remove(x)` é o mesmo que o custo de uma operação `find(x)`.

Em resumo, se ignorarmos o custo de chamadas a `resize()`, todas as operações em uma `LinearHashTable` rodam em $O(1)$ de tempo esperado. Considerar o custo de redimensionamento pode ser feito usando o mesmo tipo de análise amortizada realizada para a estrutura de dados `ArrayStack` feita na Seção 2.1.

5.2.2 Resumo

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

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

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

[3] Em muitos livros de cálculo, essa soma passa no teste de convergência: existe um inteiro positivo k_0 tal que, para todo $k \geq k_0$, $\frac{(k+1)^2 c^{k+1}}{k^2 c^k} < 1$.

5.2.3 Hashing por tabulação

Ao analisarmos a estrutura `LinearHashTable`, fazemos uma suposição muito forte: para qualquer conjunto de elementos $\{x_1,\ldots,x_n\}$ os valores de hash $hash(x_1),\ldots,hash(x_n)$ são independentes e uniformemente distribuídos sobre o conjunto $\{0,\ldots,t.length-1\}$. Um jeito de obter isso é guardar em um array gigante `tab` de tamanho 2^w, onde cada entrada é um inteiro aleatório `w`-bit, independente de todas as outras posições. Dessa maneira, poderíamos implementar `hash(x)` pela extração de um inteiro com `d` bits da `tab[x.hashCode()]`:

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

Infelizmente, guardar um array de tamanho 2^w é proibitivo em termos de uso de memória. A abordagem usada pela *hashing por tabulação* é, em vez disso, tratar inteiros de `w` bits como sendo compostos de `w/r` inteiros, cada qual tem somente `r` bits. Dessa maneira, hashing por tabulação precisa somente de `w/r` arrays cada qual com tamanho 2^r. Todas as entradas nesses arrays são inteiros aleatórios independentes com `w` bits.

Para obter o valor de `hash(x)` dividimos `x.hashCode()` em `w/r` inteiros com `r` bits e os usamos como índices desses arrays. Então combinamos todos esses valores com o operador bit-a-bit ou-exclusivo para obter `hash(x)`. O código a seguir mostra como isso funciona quando `w` = 32 e `r` = 4:

LinearHashTable
```
int hash(T x) {
  int h = x.hashCode();
  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.

É fácil verificar que, para qualquer x, hash(x) é uniformemente distribuído sobre $\{0, \ldots, 2^d - 1\}$. Com um pouco de trabalho, é possível verificar que qualquer par de valores tem suas hash independentes. Isso implica que hashing por tabulação poderia ser usado no lugar do hashing multiplicativo para a implementação da ChainedHashTable.

Entretanto, não é verdade que qualquer conjunto de n valores distintos resulte em n valores hash independentes. De qualquer forma, quando hashing por tabulação é usado, o limitante Teorema 5.2 ainda vale. Referências para isso são providas no final deste capítulo.

5.3 Códigos Hash

Tabelas 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. Elas podem ser strings, objetos, arrays, ou outras estruturas compostas. Para usar tabelas hash para esses tipos de dados, precisamos mapear esses tipos de dados para códigos hash de w bits. Mapeamentos de códigos hash devem ter as seguintes propriedades:

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

A primeira propriedade assegura que se guardarmos x em um tabela hash e depois olharmos um valor y igual a x, então iremos achar x — conforme deveríamos. A segunda propriedade minimiza a perda ao converter nossos objetos a inteiros. Ela garante que objetos distintos usualmente tenham códigos hash diferentes e então provavelmente são guardados em posições distintas na nossa tabela hash.

5.3.1 Códigos hash para Tipos de Dados Primitivos

Tipos de dados primitivos pequenos como char, byte, int e float são normal fáceis de obter códigos hash. Esses tipos de dados sempre têm

uma representação binária que usualmente consistem de w ou menos bits. (Por exemplo, em Java, `byte` é um tipo de 8 bits e `float` é um tipo de 32 bits.) Nesses casos, nós somente tratamos esses bits como a representação de um inteiro no intervalo $\{0,\ldots,2^w-1\}$. Se dois valores são diferentes, eles recebem códigos hash distintos. Se são iguais, recebem o mesmo código hash.

Alguns dados de tipos primitivos são compostos de mais de w bits, normalmente cw bits para alguma constante c. (No Java, os tipos `long` e `double` são exemplos disso com $c = 2$.) Esses tipos de dados podem ser tratados como objetos compostos de c partes, conforme descrito na seção a seguir.

5.3.2 Códigos Hash para Objetos Compostos

Para um objeto composto, queremos criar um código hash combinando os códigos hash individuais das partes que constituem o objeto. Isso não é tão fácil quanto parece.

Embora seja possível achar muitos hacks para isso (por exemplo, combinar os códigos hash com operações bitwise ou-exclusivo), muitos desses hacks acabam sendo fáceis de enganar (veja os Exercícios 5.7–5.9). Porém, se pudermos fazer aritmética com $2w$ bits de precisão, então existem métodos simples e robustos a nossa disposição. Suponha que temos um objeto composto de várias partes $P_0,\ldots,P_{r-1}$ cujos códigos hash são $x_0,\ldots,x_{r-1}$. Então, podemos escolher inteiros de w bits mutuamente independentes $z_0,\ldots,z_{r-1}$ e um inteiro ímpar aleatório com $2w$ bits z e computar um código hash para nosso objeto com

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

Note que esse código hash tem um passo final (multiplicar por z e dividir por 2^w) que usa uma função hash multiplicativo da Seção 5.1.1 para obter um resultado intermediário $2w$ bits e reduzir a um resultado final de w bits. Aqui segue um exemplo desse método aplicado a um único objeto composto de três partes `x0`, `x1` e `x2`:

Point3D

```
int hashCode() {
  // random numbers from rand.org
  long[] z = {0x2058cc50L, 0xcb19137eL, 0x2cb6b6fdL};
  long zz = 0xbea0107e5067d19dL;

  // convert (unsigned) hashcodes to long
  long h0 = x0.hashCode() & ((1L<<32)-1);
  long h1 = x1.hashCode() & ((1L<<32)-1);
  long h2 = x2.hashCode() & ((1L<<32)-1);

  return (int)(((z[0]*h0 + z[1]*h1 + z[2]*h2)*zz)
              >>> 32);
}
```

O teorema a seguir mostra que, em adição a ser de simples implementação, esse método é comprovadamente bom:

Teorema 5.3. *Sejam* $\mathtt{x}_0,\ldots,\mathtt{x}_{r-1}$ *e* $\mathtt{y}_0,\ldots,\mathtt{y}_{r-1}$ *sequências de inteiros de* $\mathtt{w}$ *bits em* $\{0,\ldots,2^{\mathtt{w}}-1\}$ *e assuma* $\mathtt{x}_i \neq \mathtt{y}_i$ *para pelo menos um índice* $i \in \{0,\ldots,r-1\}$. *Então*

$$\Pr\{h(\mathtt{x}_0,\ldots,\mathtt{x}_{r-1}) = h(\mathtt{y}_0,\ldots,\mathtt{y}_{r-1})\} \leq 3/2^{\mathtt{w}} \ .$$

Demonstração. Iremos, a princípio, ignorar o passo final de hashing multiplicativo e ver como aquele passo contribui depois. Defina:

$$h'(\mathtt{x}_0,\ldots,\mathtt{x}_{r-1}) = \left(\sum_{j=0}^{r-1} \mathtt{z}_j \mathtt{x}_j\right) \bmod 2^{2\mathtt{w}} \ .$$

Assuma que $h'(\mathtt{x}_0,\ldots,\mathtt{x}_{r-1}) = h'(\mathtt{y}_0,\ldots,\mathtt{y}_{r-1})$. Podemos reescrever isso como:

$$\mathtt{z}_i(\mathtt{x}_i - \mathtt{y}_i) \bmod 2^{2\mathtt{w}} = t \tag{5.4}$$

onde

$$t = \left(\sum_{j=0}^{i-1} \mathtt{z}_j(\mathtt{y}_j - \mathtt{x}_j) + \sum_{j=i+1}^{r-1} \mathtt{z}_j(\mathtt{y}_j - \mathtt{x}_j)\right) \bmod 2^{2\mathtt{w}}$$

Se assumirmos, sem perda de generalidade, que $\mathtt{x}_i > \mathtt{y}_i$, então a (5.4) torna-se

$$\mathtt{z}_i(\mathtt{x}_i - \mathtt{y}_i) = t \ , \tag{5.5}$$

pois cada z_i e $(x_i - y_i)$ são até $2^w - 1$, então os seus produtos são até $2^{2w} - 2^{w+1} + 1 < 2^{2w} - 1$.

Por suposição, $x_i - y_i \neq 0$, então a (5.5) tem no máximo uma solução em z_i. Portanto, como z_i e t são independentes ($z_0, \ldots, z_{r-1}$ são mutuamente independentes), a probabilidade de selecionarmos z_i tal que $h'(x_0, \ldots, x_{r-1}) = h'(y_0, \ldots, y_{r-1})$ é no máximo $1/2^w$.

O passo final da função hash é aplicar o hashing multiplicativo para reduzir nosso resultado intermediário de $2w$ bits $h'(x_0, \ldots, x_{r-1})$ a um resultado final com w bits $h(x_0, \ldots, x_{r-1})$.

Usando o Teorema 5.3, se $h'(x_0, \ldots, x_{r-1}) \neq h'(y_0, \ldots, y_{r-1})$, então podemos concluir que $\Pr\{h(x_0, \ldots, x_{r-1}) = h(y_0, \ldots, y_{r-1})\} \leq 2/2^w$.

Para resumir,

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

5.3.3 Códigos Hash para Arrays e Strings

O método da seção anterior funciona bem para objetos que tem um número constante, fixo, de componentes. Entretanto, ele não funciona bem quando queremos usá-lo com objetos que tem um número variável de componentes, pois querer um inteiro aleatório de w bits z_i para cada componente. Poderíamos usar uma sequência pseudo aleatória para gerar quantos z_i fossem necessários, mas então os z_i não seriam mutuamente independentes e se torna difícil provar que os números pseudo aleatórios não interagem mal com a função hash que estamos usando. Em particular, os valores de t e z_i na prova do Teorema 5.3 não seriam mais independentes.

Uma abordagem mais rigorosa é obter nossos códigos hash com base em polinômios sobre corpos primais; esse são basicamente polinômios comuns que são avaliados em módulo algum número primo, p. Esse método é baseado no seguinte teorema, que diz que polinômios sobre corpos

primais se comportam como polinômios comuns:

Teorema 5.4. *Seja* p *um número primo e seja* $f(\mathsf{z}) = \mathsf{x}_0\mathsf{z}^0 + \mathsf{x}_1\mathsf{z}^1 + \cdots + \mathsf{x}_{r-1}\mathsf{z}^{r-1}$ *um polinômio não trivial com coeficientes* $\mathsf{x}_i \in \{0,\ldots,\mathsf{p}-1\}$. *Então a equação* $f(\mathsf{z}) \bmod \mathsf{p} = 0$ *tem até* $r-1$ *soluções para* $\mathsf{z} \in \{0,\ldots,p-1\}$.

Para usar o Teorema 5.4, aplicamos uma função hash em uma sequência de inteiros $\mathsf{x}_0,\ldots,\mathsf{x}_{r-1}$ cada qual com $\mathsf{x}_i \in \{0,\ldots,\mathsf{p}-2\}$ usando um inteiro aleatório $\mathsf{z} \in \{0,\ldots,\mathsf{p}-1\}$ via a fórmula

$$h(\mathsf{x}_0,\ldots,\mathsf{x}_{r-1}) = \left(\mathsf{x}_0\mathsf{z}^0 + \cdots + \mathsf{x}_{r-1}\mathsf{z}^{r-1} + (\mathsf{p}-1)\mathsf{z}^r\right) \bmod \mathsf{p} \ .$$

Note o termo extra $(\mathsf{p}-1)\mathsf{z}^r$ no fim da fórmula. Ajuda a pensar de $(\mathsf{p}-1)$ como o último elemento, x_r, na sequência $\mathsf{x}_0,\ldots,\mathsf{x}_r$. Note que esse elemento difere de todos os outros elementos na sequência (cada qual está no conjunto $\{0,\ldots,\mathsf{p}-2\}$). Podemos pensar que $\mathsf{p}-1$ atua como um marcador de fim de sequência.

O teorema a seguir, que considera o caso de duas sequências de mesmo tamanho, mostra que essa função hash produz bons resultados para uma pequena quantidade de randomização necessária para escolher z:

Teorema 5.5. *Seja* $\mathsf{p} > 2^{\mathsf{w}} + 1$ *um primo, com* $\mathsf{x}_0,\ldots,\mathsf{x}_{r-1}$ *e* $\mathsf{y}_0,\ldots,\mathsf{y}_{r-1}$ *sendo sequências de inteiros com* w *bits em* $\{0,\ldots,2^{\mathsf{w}}-1\}$, *e assuma* $\mathsf{x}_i \neq \mathsf{y}_i$ *para pelo menos um índice* $i \in \{0,\ldots,r-1\}$. *Então*

$$\Pr\{h(\mathsf{x}_0,\ldots,\mathsf{x}_{r-1}) = h(\mathsf{y}_0,\ldots,\mathsf{y}_{r-1})\} \leq (r-1)/\mathsf{p} \ .$$

Demonstração. A equação $h(\mathsf{x}_0,\ldots,\mathsf{x}_{r-1}) = h(\mathsf{y}_0,\ldots,\mathsf{y}_{r-1})$ pode ser reescrita como

$$\left((\mathsf{x}_0-\mathsf{y}_0)\mathsf{z}^0 + \cdots + (\mathsf{x}_{r-1}-\mathsf{y}_{r-1})\mathsf{z}^{r-1}\right) \bmod \mathsf{p} = 0. \tag{5.6}$$

Como $\mathsf{x}_i \neq \mathsf{y}_i$, esse polinômio é não-trivial. Portanto, usando o Teorema 5.4, ele tem no máximo $r-1$ soluções em z. A probabilidade que escolhemos z para seja uma dessas soluções é portanto no máximo $(r-1)/\mathsf{p}$. □

Note que essa função hash também considera o caso em que duas sequências tem comprimentos distintos, mesmo quando uma das sequências é um prefixo de outra. Isso porque essa função efetivamente obtém o hash da sequência infinita

$$\mathsf{x}_0,\ldots,\mathsf{x}_{r-1},\mathsf{p}-1,0,0,\ldots \ .$$

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

$$\left(\sum_{i=0}^{i=r'-1}(\mathtt{x}_i - \mathtt{y}_i)\mathtt{z}^i + (\mathtt{x}_{r'} - \mathtt{p} + 1)\mathtt{z}^{r'} + \sum_{i=r'+1}^{i=r-1}\mathtt{x}_i\mathtt{z}^i + (\mathtt{p}-1)\mathtt{z}^r\right) \bmod \mathtt{p} = 0 \ ,$$

que, segundo o Teorema 5.4, tem até r soluções em $\mathtt{z}$. Isso combinado com o Teorema 5.5 é suficiente para provar o seguinte teorema mais geral:

Teorema 5.6. *Seja* $\mathtt{p} > 2^{\mathtt{w}} + 1$ *um número primo, usando duas sequências distintas de* $\mathtt{w}$ *bits* $\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}$ *e* $\mathtt{y}_0, \ldots, \mathtt{y}_{r'-1}$ *em* $\{0, \ldots, 2^{\mathtt{w}} - 1\}$. *Então*

$$\Pr\{h(\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}) = h(\mathtt{y}_0, \ldots, \mathtt{y}_{r-1})\} \leq \max\{r, r'\}/\mathtt{p} \ .$$

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

GeomVector

```
int 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 xi = (x[i].hashCode() * z2) >>> 1;
    s = (s + zi * xi) % p;
    zi = (zi * z) % p;
  }
  s = (s + zi * (p-1)) % p;
  return (int)s;
}
```

O código precedente sacrifica alguma probabilidade de colisão por conveniência na implementação. Em particular, ele aplica a função hash multiplicativa de Seção 5.1.1, com $\mathtt{d} = 31$ para reduzir `x[i].hashCode()` a um valor de 31 bits. Isso é feito dessa para que adições e multiplicações feitas módulo o primo $\mathtt{p} = 2^{32} - 5$ possam ser realizadas usando aritmética

63 bits sem sinal. Então a probabilidade de duas diferentes sequências, a mais longa com comprimento r, terem o mesmo código hash é até

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

em vez da $r/(2^{32} - 5)$ especificada no Teorema 5.6.

5.4 Discussão e Exercícios

Tabelas hash e códigos hash representam um campo de pesquisa muito ativo que é brevemente pincelado neste capítulo. A online *Bibliography on Hashing* [10] contém aproximadamente 2000 referências.

Existe uma grande variedade de implementações de diferentes tabelas. Aquela descrita na Seção 5.1 é conhecida como *hashing with chaining* ou *hashing com encadeamento* (cada posição do array contém uma lista, por vezes encadeada, (`List`) de elementos). Hashing com encadeamento tem suas origens em um memorando interno da IBM escrito por H. P. Luhn que data de Janeiro de 1953. Esse memorando também parece ser uma das primeiras referências a listas ligadas.

Uma alternativa a hashing com encadeamento é aquela que usa esquemas de *open addressing*, ou , *endereçamento aberto*, onde todos os dados são armazenados diretamente em um array. Esses esquemas incluem a estrutura `LinearHashTable` da Seção 5.2. Essa ideia também foi proposta independentemente por um grupo na IBM na década de 1950. Esquemas de endereçamento aberto devem tratar do problema de resolução de colisões: *resolução de colisões*: o caso em que dois valores hash mapeiam para a mesma posição do array. Distintas estratégias existem para resolução de colisões; essas provêem diferentes garantias de desempenho e frequentemente requerem funções hash mais sofisticadas do que aquelas descritas aqui.

Outra categoria de implementações de tabelas hash são os famosos métodos de *hashing perfeito*. Nesses métodos a operação `find(x)` leva $O(1)$ de tempo no pior caso. Para conjuntos de dados estáticos isso pode ser realizado com *funções hash perfeitas* para os dados; existem funções que mapeiam cada dado de uma única posição do array. Para dados que mudam com o tempo, métodos de hashing perfeito incluem *tabelas hash de*

dois níveis FKS [31, 24] e *hashing cuckoo* [57].

As funções hash apresentadas neste capítulo estão provavelmente entre os métodos práticos atualmente conhecidos que funcionam bem para qualquer conjunto de dados. Outros métodos bons datam do trabalho pioneiro de Carter e Wegman que criaram o conceito de *hashing universal* e descreveram várias funções hash para diferentes cenários [14]. Hashing por tabulação, descrita na Seção 5.2.3, foi proposta por Carter e Wegman [14], mas sua análise, quando aplicada a sondagem linear (e vários outros esquemas de tabela hash) é creditada a Pătraşcu e Thorup [60].

A ideia de *hashing multiplicativo* é muito antiga e parece ser parte do folclore de hashing [48, Section 6.4]. Porém, a ideia de escolher um $\mathtt{z}$ sendo um número aleatório *ímpar*, e a análise na Seção 5.1.1 é de Dietzfelbinger *et al.* [23]. Essa versão de hashing multiplicativo é uma das mais simples, mas sua probabilidade de colisão de $2/2^{\mathtt{d}}$ é um fator de 2 maior do que se poderia esperar com uma função aleatória de $2^{\mathtt{w}} \to 2^{\mathtt{d}}$. O método de *hashing multiplica e soma*

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

onde $\mathtt{z}$ e $\mathtt{b}$ são escolhidos aleatoriamente de $\{0,\ldots,2^{2\mathtt{w}}-1\}$. Hashing multiplica e soma tem uma probabilidade de colisão de somente $1/2^{\mathtt{d}}$ [21], mas precisa de aritmética de $2\mathtt{w}$ bits.

Há vários métodos de obter códigos hash a partir de sequências de inteiros com $\mathtt{w}$ bits. Um método rápido [11] é a função

$$\begin{aligned} &h(\mathtt{x}_0,\ldots,\mathtt{x}_{r-1}) \\ &\quad = \Big(\textstyle\sum_{i=0}^{r/2-1}((\mathtt{x}_{2i}+\mathtt{a}_{2i}) \bmod 2^{\mathtt{w}})((\mathtt{x}_{2i+1}+\mathtt{a}_{2i+1}) \bmod 2^{\mathtt{w}})\Big) \bmod 2^{2\mathtt{w}} \end{aligned}$$

onde r é um número par e $\mathtt{a}_0,\ldots,\mathtt{a}_{r-1}$ são escolhidos aleatoriamente de $\{0,\ldots,2^{\mathtt{w}}\}$. Isso resulta em um código hash de $2\mathtt{w}$ bits com probabilidade de colisão $1/2^{\mathtt{w}}$. Isso pode ser reduzido a um código hash de $\mathtt{w}$ bits usando hashing multiplicativo (ou multiplica e soma). Esse método é rápido porque requer somente $r/2$ multiplicações de $2\mathtt{w}$ bits enquanto o método descrito na Seção 5.3.2 usa r multiplicações. (As operações mod ocorrem implicitamente usando aritmética de $\mathtt{w}$ e $2\mathtt{w}$ bits para as adições e multiplicações, respectivamente.)

O método da Seção 5.3.3 de usar polinômios em corpos primais para fazer hash de arrays de tamanho variável e strings é atribuído a Dietzfel-

binger *et al.* [22]. Devido ao uso do operador mod que usa uma instrução de máquina de alto custo, não é, infelizmente, muito rápida. Algumas variantes desse método escolhem o primo p para ser da forma $2^w - 1$, que nesse caso o operador mod pode ser substituído pelas operações de adição (+) e AND bit a bit (&) [47, Section 3.6]. Outra opção é aplicar um dos métodos rápidos para strings de tamanho fixo para blocos de tamanho c para alguma constante $c > 1$ e então aplicar o método de corpo primal à sequência resultante de $\lceil r/c \rceil$ códigos hash.

Exercício 5.1. Uma certa universidade atribui a cada um de seus estudantes números na primeira vez que eles registram-se para qualquer disciplina. Esses números são sequências de inteiros que iniciaram em 0 muitos anos atrás e agora estão na casa dos milhões. Suponha que temos uma turma de cem alunos do primeiro ano que queremos atribuí-los código hash baseados em seus números de estudantes. Faz mais sentido usar os dois primeiros dois dígitos ou os últimos dois dígitos do número de estudante deles? Justifique sua resposta.

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

1. Mostre que, para qualquer escolha de multiplicador z, existe n valores que têm o mesmo código hash. (Dica: isso é fácil e não precisa usar teoria dos números.)

2. Dado um multiplicador z, descreva n valores que têm o mesmo código hash. (Dica: isso é mais difícil e requer um pouco de teoria dos números.)

Exercício 5.3. Prove que o limitante $2/2^d$ no Lema 5.1 é o melhor limitante possível ao mostrar que, se $x = 2^{w-d-2}$ e $y = 3x$, então $\Pr\{\mathtt{hash}(x) = \mathtt{hash}(y)\} = 2/2^d$. (Dica: verifique as representações binárias de zx e $z3x$ e use o fato de que $z3x = zx + 2zx$.)

Exercício 5.4. Prove o Lema 5.4 usando a versão completa da aproximação de Stirling dada na Seção 1.3.2.

Exercício 5.5. Considere a seguinte versão simplificada do código para adicionar um elemento x a uma `LinearHashTable`, que simplesmente

guarda x na primeira posição do array que esteja null. Explique porque isso poderia ficar muito lento fornecendo um exemplo de uma sequência de operações $O(n)$ add(x), remove(x), e find(x) que levaria $O(n^2)$ de tempo para executar.

LinearHashTable

```
boolean 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 hashCode() do Java para a classe String não usava todos os caracteres disponíveis em strings longas. Por exemplo, para uma string com dezesseis caracteres, o código hash era computado usando somente oito caracteres com índices pares. Explique porque isso é uma ideia muito ruim fornecendo um exemplo de um grande conjunto de strings que possuem o mesmo código hash.

Exercício 5.7. Suponha que você tem um objeto composto de dois inteiros de w bits, x e y. Mostre porque $x \oplus y$ não é um bom código hash para o seu objeto. Dê um exemplo de um grande conjunto de objetos que teriam código hash 0.

Exercício 5.8. Suponha que você tem um objeto feito de dois inteiros de w bits, x e y. Mostre porque $x + y$ não seria um bom código hash para o seu objeto. Dê um exemplo de grande conjunto de objetos que teriam o mesmo código hash.

Exercício 5.9. Suponha que você tem um objeto composto de dois inteiros de w bits, x e y. Suponha que o código hash para o seu objeto é definido por alguma função hash determinística $h(x, y)$ que produz um único inteiro de w bits. Prove que existe um grande conjunto de objetos que têm o mesmo código hash.

Exercício 5.10. Seja $p = 2^{\mathtt{w}} - 1$ para algum inteiro positivo $\mathtt{w}$. 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 resulta em um algoritmo para computar $x \bmod (2^{\mathtt{w}} - 1)$ atribuindo repetidamente

$$\mathtt{x} = \mathtt{x} \& ((1 \ll \mathtt{w}) - 1) + \mathtt{x} \gg \mathtt{w}$$

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

Exercício 5.11. Ache alguma implementação comumente usada de tabela hash como, por exemplo, Java Collection Framework `HashMap`, que são implementações da `HashTable` ou `LinearHashTable` deste livro e projete um programa que guarda inteiros nessa estrutura de dados de forma que haja inteiros $\mathtt{x}$ tais que $\mathtt{find}(\mathtt{x})$ gaste tempo linear. Isso é, encontre um conjunto de $\mathtt{n}$ inteiros para os quais existam $c\mathtt{n}$ elementos cuja hash mapeia para a mesma posição da tabela.

Dependendo da qualidade da implementação, você pode ser capaz de fazê-los ao inspecionar o código da implementação, ou você pode ter que escrever algum código que fazer tentativas de inserções e buscas, medindo o tempo que leva para adicionar e achar valores. (Isso pode ser, tem sido, usado para efetuar ataques de negação de serviço em servidores web [17].)

Capítulo 6

Árvores Binárias

Este capítulo apresenta uma das estruturas mais fundamentais na Ciência da Computação: as árvores binárias. O uso da palavra *árvore* vem do fato que, quando as desenhamos, elas frequentemente lembram árvores encontradas em uma floresta. Existem muitos jeitos de definir árvores binárias. Matematicamente, uma *árvore binária* é um grafo finito, conectado, não direcionado, sem ciclos e sem nenhum vértice de grau maior que três.

Para a maioria das aplicações em Ciência da Computação, árvores binárias são *enraizadas*: Um nodo especial r de grau até dois é chamado de *raiz* da árvore. Para todo nodo $u \neq r$, o segundo nodo no caminho de u a r é chamado de *pai* de u. Cada um dos outros nodos adjacentes a u é chamado de *filho* de u. A maior parte das árvores binárias que estamos interessados são *ordenadas*, então diferenciamos entre o *filho à esquerda* e *filho à direita* de u.

Em desenhos, árvores binárias são tipicamente desenhadas de ponta-cabeça, com a raiz no topo do desenho e os filhos à esquerda e direita respectivamente dados pelas posições à esquerda e direita no desenho (Figura 6.1). Por exemplo, a Figura 6.2.a mostra uma árvore binária com nove nodos.

Devido à importância de árvores binárias, uma certa terminologia se desenvolveu para elas: a *profundidade* de um nodo u, em uma árvore binária é o comprimento do caminho de u à raiz da árvore. Se um nodo w está no caminho de u para r, então w é chamado de *ancestral* de u e u um *descendente* de w. A *subárvore* de um nodo , u, é a árvore binária que é en-

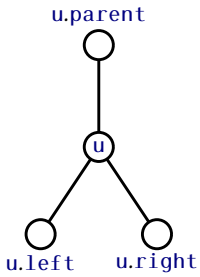


Figura 6.1: O pai, filho à esquerda e filho à direita do nodo u em uma BinaryTree.

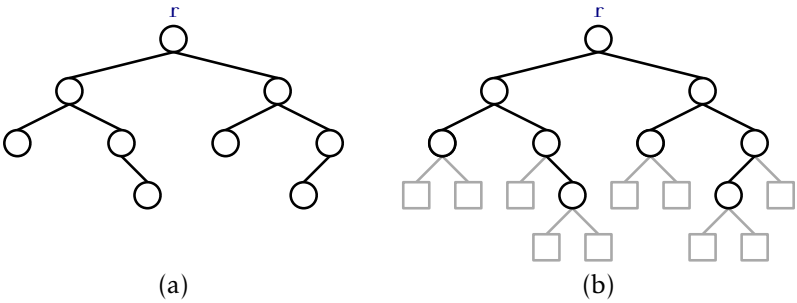


Figura 6.2: Uma árvore binária com (a) nove nodos reais (ou internos) e (b) dez nodos externos.

raizada em u e contém todos os descendentes de u. A *altura* de um nodo, u, é o comprimento do caminho mais longo de u e um de seus descendentes. A *altura* de uma árvore é a altura de sua raiz. Um nodo u é uma *folha* se não tem filhos.

Ás vezes consideramos que árvores possuem *nodos externos*. Qualquer nodo que não tem filho à esquerda tem um nodo externo como filho à esquerda e, de modo similar, um nodo que não tem filho à direita tem um nodo externo como filho à direita (ver a Figura 6.2.b). É fácil verificar, por indução, que uma árvore binária com $n \geq 1$ nodos reais tem $n + 1$ nodos externos.

6.1 BinaryTree: Uma Árvore Binária Básica

O jeito mais simples de representar um nodo u em uma árvore binária é explicitamente guardar os (no máximo três) vizinhos de u:

BinaryTree
```
class BTNode<Node extends BTNode<Node>> {
  Node left;
  Node right;
  Node parent;
}
```

Se um desses três vizinhos não estiver presente, o atribuímos a nil. Desse jeito, ambos nodos externos da árvore e o pai da raiz correspondem ao valor nil.

A própria árvore binária pode então ser representada por uma referência ao seu nodo raiz, r:

BinaryTree
```
Node r;
```

Podemos computar a profundidade de um nodo u em uma árvore binária contando o número de passos do caminho de u à 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 facilita a computação de fatos sobre árvores binárias. Por exemplo, para computar o tamanho de (número de nodos) uma árvore binária enraizada no nodo u, recursivamente computamos os tamanhos de duas subárvores enraizadas aos filhos de u, somar esses tamanhos, e somar um:

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

Para computar a altura de um nodo u, podemos computar a altura das duas subárvores de u, obter o máximo e somar um:

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

6.1.2 Percorrendo Árvores Binárias

Os dois algoritmos da seção anterior usam recursão para visitar todos os nodos em uma árvore binária. Cada um deles visita os nodos da árvore binária na mesma ordem que o código a seguir:

```
BinaryTree
void traverse(Node u) {
  if (u == nil) return;
  traverse(u.left);
  traverse(u.right);
}
```

Usar recursão desse jeito resulta em um código bem curto e simples mas pode também ser problemático. A profundidade máxima da recursão é dada pela profundidade máxima de um nodo na árvore binária, i.e., a altura da árvore.

Se a altura da árvore é grande, então essa recursão pode muito bem usar mais espaço de pilha de memória do que está disponível, causando um erro.

Para percorrer uma árvore binária sem recursão, você pode usar um algoritmo que usa a informação de onde veio para determinar para onde irá a seguir. Veja a Figura 6.3. Ao chegarmos no nodo `u` a partir de `u.parent` então o próximo passo é visitar `u.left`. Se chegarmos em `u` vindo de `u.right`, então terminamos a visita da subárvore de `u` e retornamos a `u.parent`. O código a seguir implementa essa ideia, incluindo o tratamento dos casos onde `u.left`, `u.right` ou `u.parent` sejam `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;
```

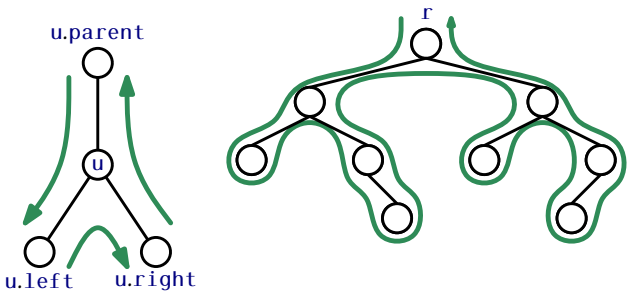


Figura 6.3: Os três casos que ocorrem no nodo u ao percorrer uma árvore binária não recursiva e o caminho resultante na árvore.

```
  }
}
```

Os mesmos fatos que podem ser computados com algoritmos recursivos também podem ser obtidos dessa maneira, sem recursão. Por exemplo, para computar o tamanho de uma árvore mantemos um contador n e o incrementamos toda vez que visitarmos um nodo 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 árvores binárias, o campo `parent` é não usado. Quando esse é o caso, uma implementação não recursiva ainda é possível, mas a implementação tem que usar uma `List` (ou `Stack`) para registrar o caminho do nodo atual à raiz.

Um tipo especial de percurso que não se adequa ao padrão das funções acima é a *travessia em largura* ou, em inglês, *breadth-first traversal*. Em uma travessia em largura, os nodos são visitados nível-a-nível iniciando na raiz e descendo, visitando os nodos a cada nível da esquerda à direita (ver Figura 6.4). Isso é similar à forma que leríamos uma página de texto em inglês. Travessia em largura é implementada usando uma queue , `q`, que inicialmente contém somente a raiz `r`. A cada passo, extraímos o próximo nodo `u` de `q`, processamos `u` e adicionamos `u.left` e `u.right` (se eles forem não-`nil`) a `q`:

BinaryTree

```
void bfTraverse() {
  Queue<Node> q = new LinkedList<Node>();
  if (r != nil) q.add(r);
  while (!q.isEmpty()) {
    Node u = q.remove();
    if (u.left != nil) q.add(u.left);
    if (u.right != nil) q.add(u.right);
  }
}
```

6.2 BinarySearchTree: Uma Árvore Binária de Busca Desbalanceada

Uma `BinarySearchTree` é um tipo especial de árvore binária em que cada nodo `u` também guarda um valor de dado `u.x` a partir de alguma ordem total. Os valores de dados em uma árvore binária de busca obedecem a

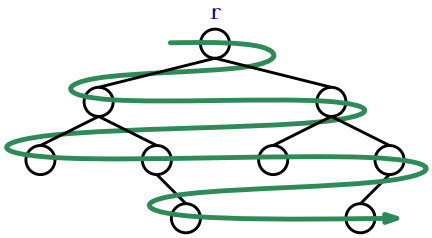


Figura 6.4: Durante uma travessia em largura, os nodos de uma árvore binária são visitados um nível por vez e da esquerda para a direita a cada nível.

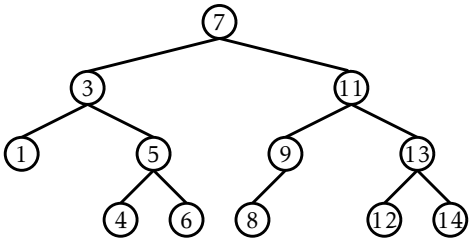


Figura 6.5: Uma árvore binária de busca.

propriedade da árvore binária de busca: Para um nodo `u`, todo valor guardado na subárvore enraizada em `u.left` é menor que `u.x` e todo valor de dado guardado em uma subárvore enraizada em `u.right` é maior que `u.x`. Um exemplo de uma `BinarySearchTree` é mostrado em Figura 6.5.

6.2.1 Busca

A propriedade da árvore binária de busca é extremamente útil porque nos permite rapidamente localizar um valor `x` em uma árvore binária de busca. Para fazer isso, iniciamos procurando por `x` na raiz `r`. Ao examinar um nodo `u`, há três casos:

1. Se $x < u.x$, então a busca segue para `u.left`;

2. Se $x > u.x$, então a busca segue para `u.right`;

3. Se $x = u.x$, então achamos o nodo `u` contendo `x`.

A busca termina quando o Caso 3 ocorre ou quando $u = nil$. No Caso 3, achamos `x`. Quando $u = nil$, concluímos que `x` não está na árvore binária de busca.

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

Dois exemplos de buscas em uma árvore binária de busca são mostrados em Figura 6.6. Conforme o segundo exemplo mostra, mesmo se não acharmos x no último nodo `u`, no qual ocorre o Caso 1, vemos que `u.x` é o menor valor na árvore que é maior que `x`. De modo similar, o último nodo no qual Caso 2 ocorreu contém o maior valor na árvore que é menor que `x`. Portanto, ao registrar o último nodo `z` no qual Caso 1 ocorre, uma `BinarySearchTree` pode implementar a operação `find(x)` que retorna o menor valor guardado na árvore 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) {
```

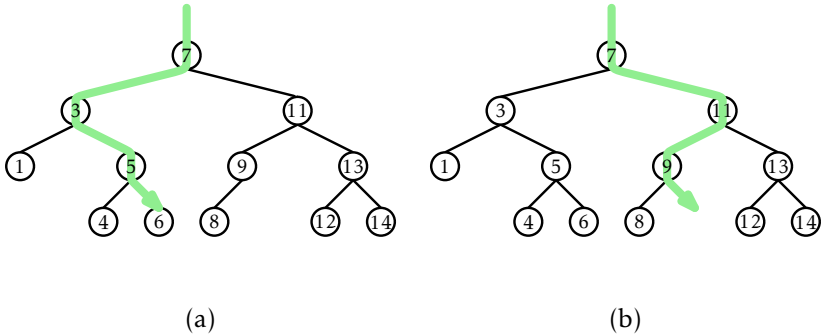


Figura 6.6: Um exemplo de (a) uma busca bem sucedida (por 6) e (b) uma busca mal sucedida (por 10) em uma árvore binária de busca.

```
      z = w;
      w = w.left;
    } else if (comp > 0) {
      w = w.right;
    } else {
      return w.x;
    }
  }
  return z == nil ? null : z.x;
}
```

6.2.2 Adição

Para adicionar um novo valor `x` a uma `BinarySearchTree`, primeiro buscamos por `x`. Se acharmos, então não há necessidade de o inserirmos. Por outro lado, guardamos `x` na folha do último nodo `p` encontrado durante a busca por `x`. Se um novo nodo está no filho à esquerda ou à direita de `p` depende no resultado da comparação entre `x` e `p.x`.

BinarySearchTree

```
boolean add(T x) {
  Node p = findLast(x);
  return addChild(p, newNode(x));
```

```
}
```

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

```
boolean 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 em Figura 6.7. A parte que mais gasta tempo desse processo é a busca inicial por x, que leva um tempo proporcional

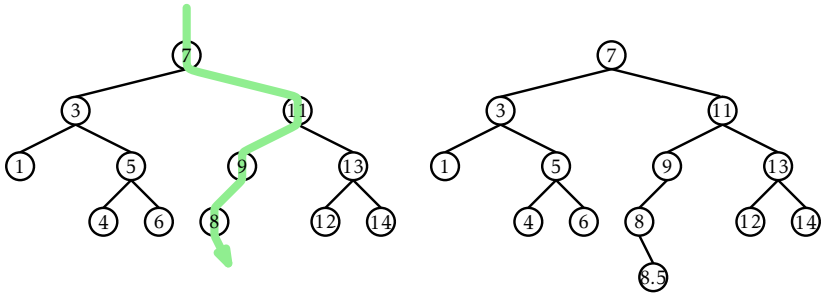


Figura 6.7: Inserindo o valor 8.5 em uma árvore binária de busca.

à altura do novo nodo u. No pior caso, isso é igual à altura da BinarySearchTree.

6.2.3 Remoção

Remover um valor guardado em um nodo u de uma BinarySearchTree é um pouco mais difícil. Se u for uma folha, então podemos simplesmente desconectar u de seu pai.

Melhor ainda, se u tiver um único filho, então podemos destacar u da árvore fazendo u.parent adotar o filho de u (veja a 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 {
```

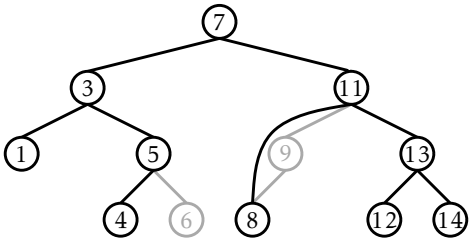


Figura 6.8: Remover uma folha (6) ou um nodo com somente um filho (9) é fácil.

```
      p.right = s;
    }
  }
  if (s != nil) {
    s.parent = p;
  }
  n--;
}
```

A situação complica quando `u` tem dois filhos. Nesse caso, o mais simples a fazer é achar um nodo `w`, que seja menor que os dois filhos e tal que `w.x` possa substituir `u.x`. Para manter a propriedade da árvore de busca binária, o valor `w.x` deveria ser próximo ao valor de `u.x`. Por exemplo, escolher um `w` tal que `w.x` é o menor valor maior que `u.x` funcionaria. Achar o nodo `w` é fácil; é o menor valor na subárvore enraizada em `u.right`. Esse nodo pode ser facilmente removido porque não tem filho à esquerda (veja a Figura 6.9).

BinarySearchTree

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

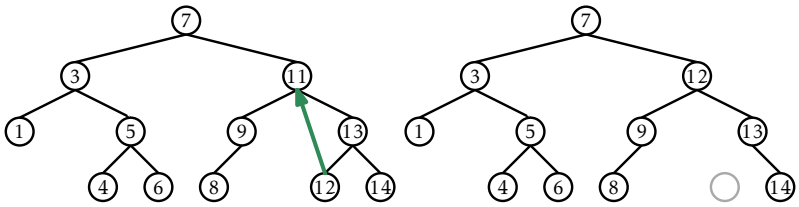


Figura 6.9: A remoção de um valor (11) de um nodo, u, com dois filhos é feita pela substituição do valor de u com o menor valor na subárvore à direita de u.

```
    }
}
```

6.2.4 Resumo

As operações `find(x)`, `add(x)` e `remove(x)` em uma `BinarySearchTree` envolvem seguir um caminho da raiz da árvore a algum nodo. Sem saber mais sobre o formato da árvore é difícil dizer algo sobre o comprimento desse caminho, exceto que é menor que n, o número de nodos na árvore. A seguinte teorema (pouco impressionante) resume o desempenho da estrutura de dados `BinarySearchTree`:

Teorema 6.1. *`BinarySearchTree` implementa a interface `SSet` interface e aceita as operações* `add(x)`, `remove(x)` *e* `find(x)` *em* $O(n)$ *de tempo por operação.*

O Teorema 6.1 demonstra um desempenho ruim em comparação ao Teorema 4.1, que mostra que uma estrutura `SkiplistSSet` implementa a interface `SSet` com $O(\log n)$ de tempo esperado por operação. O problema com a estrutura `BinarySearchTree` é que pode se tornar *desbalanceada*. Em vez de parecer como uma árvore na Figura 6.5, ela pode parecer como uma longa cadeia de n nodos, todos exceto o último com exatamente um filho.

Existem várias maneiras de evitar árvores binárias desbalanceadas, todas elas levam a estruturas de dados que tem operações que executam com $O(\log n)$ de tempo. No Capítulo 7 mostramos como operações

com $O(\log n)$ de tempo *esperado* podem ser obtidas com randomização. No Capítulo 8 mostramos como operações que executam em $O(\log n)$ de tempo *amortizado* podem ser obtidas com operações de reconstrução parcial. No Capítulo 9 mostramos como operações que executam em $O(\log n)$ de tempo no *pior caso* podem ser obtidas ao simular uma árvore que não é binária: uma árvore cujos nodos podem ter até quatro filhos.

6.3 Discussão e Exercícios

Árvores binárias têm sido usadas para modelar relações por milhares de anos. Uma razão para isso é que árvores binárias naturalmente modelam árvores genealógicas (e de pedigree). Essas árvores genealógicas em que a raiz é uma pessoa, os filhos à esquerda e à direita são os pais de uma pessoa e assim por diante, recursivamente. Em séculos mais recentes, árvores binárias também tem sido usadas para modelar espécies de árvores em biologia, onde as folhas da árvore representam uma espécie existente e nodos internos de uma árvore representam *eventos de especiação* no qual duas populações de uma única espécie evoluem em duas espécies distintas.

Árvores binárias de busca parecem terem sido descobertas independentemente por vários grupos na década de 1950 [48, Section 6.2.2]. Referências adicionais a tipos específicos de árvores binárias de busca são fornecidas nos capítulos a seguir.

Ao implementar uma árvore binária desde o início, várias decisões de projeto devem ser feitas. Uma delas é se nodos devem guardar um ponteiro para o seu pai.

Se a maior parte das operações simplesmente seguirem um caminho da raiz para a folha, então ponteiros para nodos-pai são desnecessários, desperdício de memória e uma potencial fonte de erros de codificação. Por outro lado, a falta de ponteiros para nodos-pai significa que travessias em árvores devem serem feitas recursivamente ou com o uso de uma stack explícita.

Alguns outros métodos (como inserir ou remover em alguns tipos de árvores binárias de busca balanceadas) também podem ficar mais complicados sem o uso de ponteiros para nodo-pai.

Outra decisão de projeto refere-se a como guardar os ponteiros pai, filho à esquerda e filho à direita em um nodo. Na implementação dada aqui, esses ponteiros são guardados como variáveis separadas. Outra opção é guardá-los em um array p de tamanho 3, tal que u.p[0] é o filho à esquerda de u, u.p[1] é o filho à direita de u e u.p[2] é o pai de u. Usando um array dessa forma implica que algumas sequências de comandos `if` podem ser simplificadas em expressões algébricas.

Um exemplo dessa simplificação ocorre durante uma travessia de árvore. Se uma travessia chega em um nodo u a partir de u.p[i], então o próximo nodo na travessia é u.p[(i + 1) mod 3]. Exemplos similares ocorrem quando há simetria esquerda-direita. Por exemplo, o irmão de u.p[i] é u.p[(i + 1) mod 2]. Esse truque funciona se u.p[i] é um filho à esquerda (i = 0) ou um filho à direita (i = 1) de u. Em alguns casos, isso significa que algum código complicado que de outra maneira precisaria ter uma versão para atuar na esquerda e uma versão para atuar na direita poderia ser escritos apenas uma vez. Como exemplo, veja os métodos `rotateLeft(u)` e `rotateRight(u)` na página 169.

Exercício 6.1. Prove que uma árvore binária com $n \geq 1$ nodos tem $n - 1$ arestas.

Exercício 6.2. Prove que uma árvore binária com $n \geq 1$ nodos reais tem $n + 1$ nodos externos.

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

Exercício 6.4. Implemente um método não recursivo, `size2(u)`, que computa o tamanho da subárvore enraizada no nodo u.

Exercício 6.5. Escreva um método não recursivo `height2(u)` que computa a altura do nodo u em uma `BinaryTree`.

Exercício 6.6. Uma árvore binária é *balanceada no tamanho* se, para todo nodo u, o tamanho das subárvores enraizadas em u.`left` e u.`right` diferem em no máximo uma unidade. Escreva `isBalanced()`, um método recursivo que testa se uma árvore binária é balanceada. O seu método deve rodar em $O(n)$ de tempo. (Teste o seu código em algumas árvores

maiores com diferentes formas; é fácil escrever um método que usa bem mais de $O(n)$ de tempo.)

Uma travessia em *pré-ordem* (em inglês, *pre-order* traversal) de uma árvore binária é uma travessia que visita cada nodo u antes de qualquer de seus filhos. Uma travessia em ordem (em inglês, *in-order* traversal) visita u depois de visitar todos os nodos na subárvore à esquerda de u mas antes de visitar qualquer nodo na subárvore à direita de u.

Um travessia em *pós-ordem* (em inglês, *post-order*) visita u somente após visitar todos os outros nodos na subárvore de u. A numeração pré/em/pós-ordem de uma árvore marca os nodos de uma árvore com os inteiros $0, \ldots, n-1$ na ordem em que são visitados por uma travessia em pré/em/pós-ordem. Para um exemplo, veja Figura 6.10.

Exercício 6.7. Crie uma subclasse de `BinaryTree` cujos nodos tem campos para guardar numerações pré/in/pós-ordem. Escreva métodos recursivos `preOrderNumber()`, `inOrderNumber()` e `postOrderNumbers()` que atribui esses números corretamente. Cada um desses métodos deve rodar em $O(n)$ de tempo.

Exercício 6.8. Implemente as funções não recursivas `nextPreOrder(u)`, `nextInOrder(u)` e `nextPostOrder(u)` que retornam o nodo que segue u em uma travessia pré/em/pós-ordem, respectivamente. Essas funções devem levar tempo constante amortizado; se iniciarmos em qualquer nodo u e chamarmos uma dessas funções e atribuirmos o valor a u até u = `null`, então o custo de todas essas chamadas deve ser $O(n)$.

Exercício 6.9. Suponha que recebemos uma árvore binária com numerações pré/em/pós ordem atribuídas aos nodos. Mostre como esses números podem ser usados para responder cada uma das perguntas a seguir em tempo constante:

1. Dado um nodo u, determine o tamanho da subárvore enraizada em u.

2. Dado um nodo u, determine a profundidade de u.

3. Dados dois nodos u e w, determine se u é um ancestral de w.

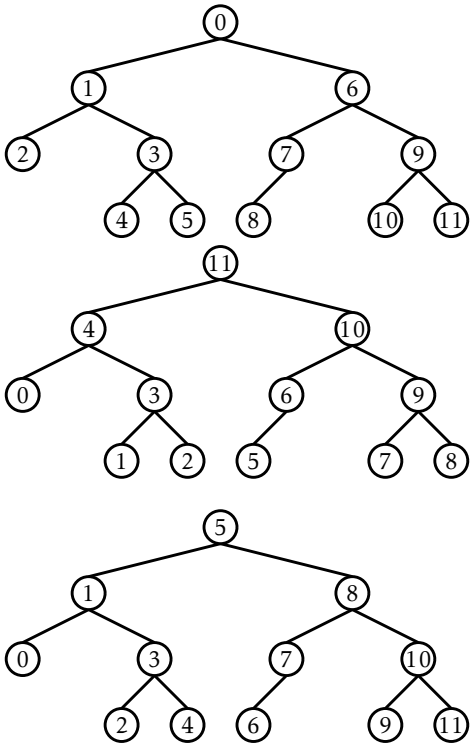


Figura 6.10: Numerações pré-ordem, pós-ordem e em ordem de uma árvore binária.

Exercício 6.10. Suponha que você recebeu uma lista de nodos com numerações pré/em-ordem atribuídas. Prove que existe no máximo uma árvore possível com essas numerações e mostre como construí-la.

Exercício 6.11. Mostre que o tamanho de qualquer árvore binária em n nodos pode ser representado usando no máximo $2(n-1)$ bits. (Dicas: pense em gravar o que acontece durante uma travessia em então repetir essa gravação para reconstruir a árvore.)

Exercício 6.12. Ilustre o que acontece quando adicionamos os valores 3.5 e então 4.5 à árvore binária de busca em Figura 6.5.

Exercício 6.13. Ilustre o que acontece ao removermos os valores 3 e então 5 da árvore binária de busca em Figura 6.5.

Exercício 6.14. Implemente um método na `BinarySearchTree` chamado de `getLE(x)` que retorna uma lista de todos os itens na árvore que são menores que ou iguais a x. O tempo de execução do seu método deve ser $O(n' + h)$ onde n' é o número de itens menores que ou iguais a x e h é a altura da árvore.

Exercício 6.15. Descreva como adicionar elementos $\{1,\ldots,n\}$ a uma `BinarySearchTree` inicialmente vazia de tal forma que a árvore resultante tem altura $n-1$. Quantas formas existem para fazer isso?

Exercício 6.16. Se tivermos uma `BinarySearchTree` e realizarmos as operações `add(x)` seguidas de `remove(x)` (com o mesmo valor de x) retornamos à árvore original?

Exercício 6.17. É possível uma operação `remove(x)` aumentar a altura de um nodo em uma `BinarySearchTree`? Se sim, em quanto?

Exercício 6.18. Uma operação `add(x)` consegue aumentar a altura de um nodo em uma `BinarySearchTree`? Ela pode aumentar a altura da árvore? Se sim, em quanto?

Exercício 6.19. Projete e implemente uma versão da `BinarySearchTree` em que cada nodo u, mantém valores `u.size` (o tamanho da subárvore enraizada em u), `u.depth` (a profundidade de u) e `u.height` (a altura da subárvore enraizada de u).

Esses valores devem ser mantidos, mesmo durante chamadas às operações `add(x)` e `remove(x)`, mas isso não deve aumentar o custo dessas operações em mais do que um fator constante.

Capítulo 7

Árvores Binárias de Busca Aleatórias

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

7.1 Árvores Binárias de Busca Aleatórias

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

Imagine como essas duas árvores podem ter sido construídas. Aquela na esquerda ocorre se iniciamos com uma `BinarySearchTree` vazia e adicionando a sequência

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

Nenhuma outra sequência de adições irá criar essa árvore (como você pode provar por indução em n). Por outro lado, a árvore na 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 também funcionariam, incluindo

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

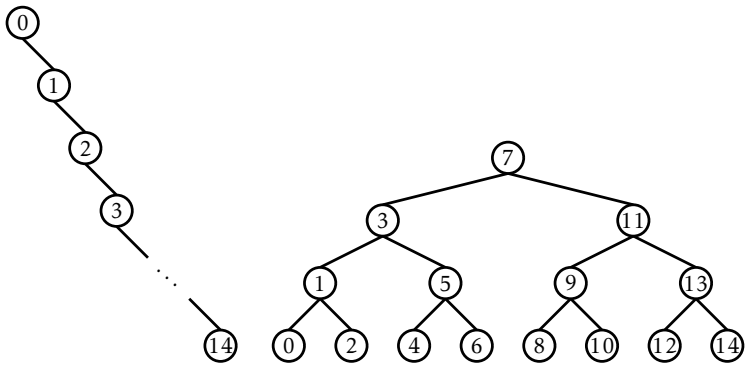


Figura 7.1: Duas árvores binárias de busca contendo inteiros $0,\ldots,14$.

e

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

De fato, há $21,964,800$ sequências que gerariam a árvore na direita e somente uma gera a árvore na esquerda.

O exemplo acima passa uma ideia de que, se escolhermos uma permutação aleatória de $0,\ldots,14$, e a adicionarmos em uma árvore binária de busca, então teremos uma chance maior de obter uma árvore muito balanceada (o lado direito da Figura 7.1) que temos de obter uma árvore muito desbalanceada (o lado esquerdo da Figura 7.1).

Podemos formalizar essa noção ao estudar árvores binárias de busca aleatórias. Uma *árvore binária de busca aleatória* de tamanho n é obtida da seguinte forma: pegue uma permutação aleatória, $x_0,\ldots,x_{n-1}$, de inteiros $0,\ldots,n-1$ e adicione seus elementos, um por um em uma `BinarySearchTree`. Queremos dizer com *permutação aleatória* que cada uma das possíveis $n!$ permutações (reordenações) de $0,\ldots,n-1$ é igualmente provável, tal que a probabilidade de obter uma permutação em particular é $1/n!$.

Note que os valores $0,\ldots,n-1$ poderiam ser substituídos por qualquer conjunto ordenado de n elementos sem alterar a propriedade das árvores binárias de busca aleatória. O elemento $x \in \{0,\ldots,n-1\}$ está simplesmente representando o elemento de ranking (posição) x em um conjunto ordenado de tamanho n.

Antes de apresentarmos o nosso principal resultado sobre árvores bi-

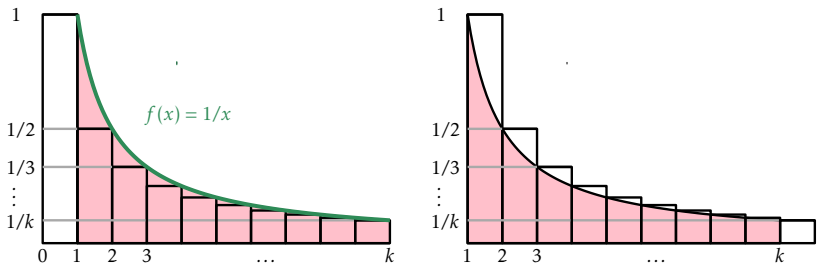


Figura 7.2: O k-ésimo número harmônico $H_k = \sum_{i=1}^{k} 1/i$ é limitado superiormente e inferiormente por duas integrais. O valor dessas integrais é dado pela área da região sombreada, enquanto o valor de H_k é dado pela área dos retângulos.

nárias de busca aleatórias, precisamos fazer uma curta discussão sobre um tipo de número que aparece frequentemente ao estudar estruturas randomizadas. Para um inteiro não-negativo k, o k-ésimo *número harmônico*, denotado por H_k, é definido como

$$H_k = 1 + 1/2 + 1/3 + \cdots + 1/k \ .$$

O número harmônico H_k na tem forma fechada simples, mas é muito relacionado ao logaritmo natural de k. Em particular,

$$\ln k < H_k \leq \ln k + 1 \ .$$

Leitores que estudaram cálculo podem perceber que isso é verdadeiro pois a integral $\int_1^k (1/x)\,\mathrm{d}x = \ln k$. Tendo em mente que uma integral pode ser interpretada como a área entre uma curva e o eixo x, o valor de H_k pode ser limitado inferiormente pela integral $\int_1^k (1/x)\,\mathrm{d}x$ e limitado superiormente por $1 + \int_1^k (1/x)\,\mathrm{d}x$. (Veja a Figura 7.2 para uma explicação gráfica.)

Lema 7.1. *Em uma árvore binária de busca aleatória de tamanho* n*, as seguintes afirmações valem:*

1. *Para qualquer* $\mathsf{x} \in \{0, \ldots, \mathsf{n}-1\}$*, o comprimento esperado do caminho de busca por* 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 menor que ou igual a x e o número de elementos na árvore maiores que ou iguais a x.

2. *Para qualquer* $\mathtt{x} \in (-1, n) \setminus \{0, \ldots, \mathtt{n}-1\}$, *o comprimento esperado do caminho de busca para* $\mathtt{x}$ *é* $H_{\lceil \mathtt{x} \rceil} + H_{\mathtt{n}-\lceil \mathtt{x} \rceil}$.

Provaremos o Lema 7.1 na seção a seguir. Por enquanto, considere o que as duas partes do Lema 7.1 nos diz. A primeira parte nos diz que se buscarmos por um elemento em uma árvore de tamanho $\mathtt{n}$, então o comprimento esperado do caminho de busca é no máximo $2 \ln n + O(1)$.

A segunda parte no diz sobre a busca de um valor não guardado na árvore. Ao compararmos duas partes do lema, vemos que buscar algo que está na árvore é apenas um pouco mais rápido do que algo que não está.

7.1.1 Prova do Lema 7.1

A observação chave necessária para provar o Lema 7.1 é a seguinte: o caminho de busca por um valor $\mathtt{x}$ no intervalo aberto $(-1, \mathtt{n})$ em uma árvore binária de busca aleatória T contém o nodo com chave $i < \mathtt{x}$ se, e somente se, na permutação aleatória usada para criar T, i aparece antes de qualquer $\{i+1, i+2, \ldots, \lfloor \mathtt{x} \rfloor\}$.

Para ver isso, observe a Figura 7.3 e note que até que algum valor em $\{i, i+1, \ldots, \lfloor \mathtt{x} \rfloor\}$ seja adicionado, os caminhos de busca para cada valor no intervalo aberto $(i-1, \lfloor \mathtt{x} \rfloor + 1)$ são idênticos. (Lembre-se que para dois valores terem caminhos distintos, precisa haver algum elemento na árvore que compara de modo diferente entre eles.) Seja j o primeiro elemento em $\{i, i+1, \ldots, \lfloor \mathtt{x} \rfloor\}$ para aparecer na permutação aleatória. Note que j está e sempre estará no caminho de busca por $\mathtt{x}$. Se $j \neq i$ então o nodo $\mathtt{u}_j$ contendo j é criado antes do nodo $\mathtt{u}_i$ que contém i. Depois, quando i for adicionado, ele será adicionado na subárvore enraizada em $\mathtt{u}_j.\mathtt{left}$, pois $i < j$. Por outro lado, o caminho de busca para $\mathtt{x}$ nunca visitará essa subárvore porque irá proceder para $\mathtt{u}_j.\mathtt{right}$ após visitar $\mathtt{u}_j$.

De modo similar, para $i > \mathtt{x}$, i aparece no caminho de busca para $\mathtt{x}$ se e somente se i aparecer antes de $\{\lceil \mathtt{x} \rceil, \lceil \mathtt{x} \rceil + 1, \ldots, i-1\}$ na permutação aleatória usada para criar T.

Note que, se iniciarmos com uma permutação aleatória de $\{0, \ldots, \mathtt{n}\}$, então as subsequências contendo somente $\{i, i+1, \ldots, \lfloor \mathtt{x} \rfloor\}$ e $\{\lceil \mathtt{x} \rceil, \lceil \mathtt{x} \rceil + 1, \ldots, i-1\}$ também são permutações aleatórias de seus respectivos elementos. Cada

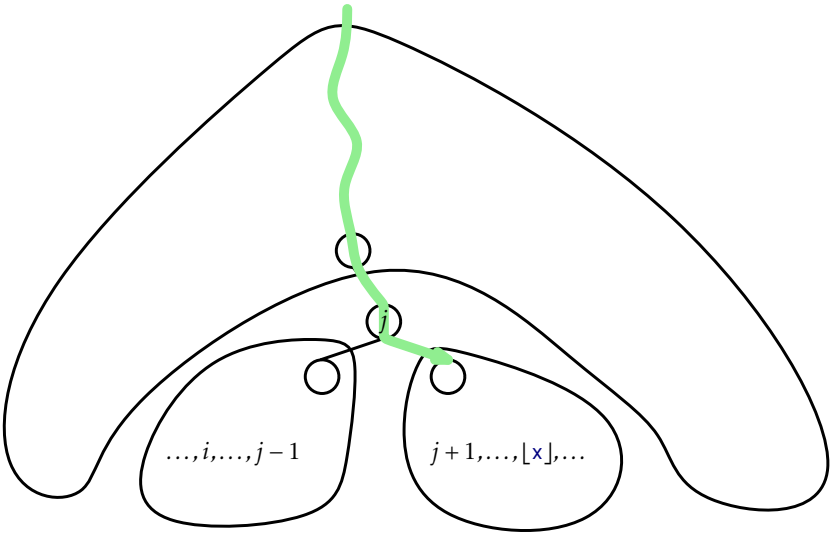


Figura 7.3: O valor $i < x$ está no caminho de árvore para x se e somente se i for o primeiro elemento entre $\{i, i+1, \ldots, \lfloor x \rfloor\}$ adicionado à árvore.

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 que qualquer outro no seu subconjunto na permutação aleatória usada para criar T. Então temos

$$\Pr\{i \text{ está no caminho de busca por } x\} = \begin{cases} 1/(\lfloor x \rfloor - i + 1) & \text{se } i < x \\ 1/(i - \lceil x \rceil + 1) & \text{se } i > x \end{cases} .$$

A partir dessa observação, a prova do Lema 7.1 envolve alguns cálculos simples com números harmônicos:

Prova do Lema 7.1. Seja I_i seja uma variável indicadora aleatória que é igual a um quando i aparecer no caminho de busca por x e zero caso contrário. Então, o comprimento do caminho de busca é dado por

$$\sum_{i \in \{0, \ldots, n-1\} \setminus \{x\}} I_i$$

então, se $x \in \{0, \ldots, n-1\}$, o comprimento esperado do caminho de busca

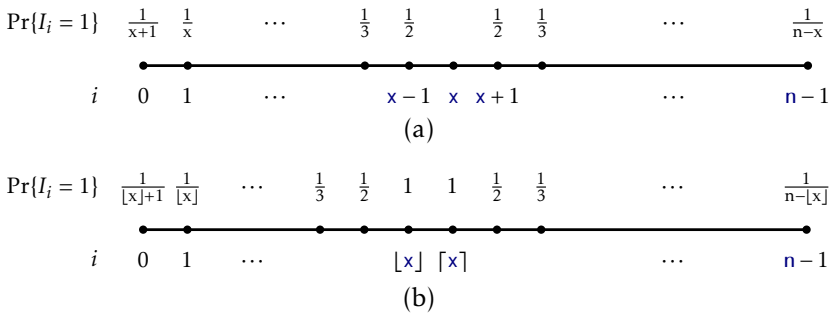


Figura 7.4: As probabilidades de um elemento estar no caminho de busca por x quando (a) x é um inteiro e (b) quando x não é um inteiro.

é dado por (veja a 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 um valor buscado $x \in (-1, n) \setminus \{0, \ldots, n-1\}$ são quase idênticos (veja a Figura 7.4.b). □

7.1.2 Resumo

O teorema a seguir resume o desempenho de uma árvore binária de busca aleatória:

Teorema 7.1. *Uma árvore binária de busca aleatória pode ser construída em* $O(n \log n)$ *de tempo. Em uma árvore binária de busca aleatória, a operação* `find(x)` *leva* $O(\log n)$ *de tempo esperado.*

Devemos enfatizar novamente que o valor esperado no Teorema 7.1 é em respeito à permutação aleatória usada para criar a árvore binária de busca aleatória. Em especial, não depende de uma escolha aleatória de `x`; isso é verdade para todo valor de `x`.

7.2 Treap: Uma Árvore Binária de Busca Aleatória

O problema com uma árvore binária de busca aleatória é, claramente, que ela não é dinâmica. Ela não aceita as operações `add(x)` ou `remove(x)` necessárias para implementar a interface `SSet`. Nesta seção descrevemos uma estrutura de dados chamada de `Treap` que usa o Lema 7.1 para implementar a interface `SSet`.[2]

Um nodo em uma `Treap` é como um nodo em uma `BinarySearchTree` em que ele tem um valor de dado `x` mas também contém um valor único chamado *priority* `p` `p`, que é atribuído aleatoriamente:

Treap
```
class Node<T> extends BSTNode<Node<T>,T> {
  int p;
}
```

Além de ser uma árvore binária de busca, os nodos em uma `Treap` também obedecem a *propriedade das heaps*:

- (Propriedade das Heaps) Em todo nodo `u`, exceto na raiz, `u.parent.p` < `u.p`.

Em outras palavras, cada nodo tem uma prioridade menor que aquelas de seus filhos. Um exemplo é mostrado na Figura 7.5.

As condições impostas pela heap e por uma árvore binária de busca juntas asseguram que, uma vez que a chave (`x`) e a prioridade (`p`) para

[2] O nome `Treap` vem do fato que essa estrutura de dados é simultâneamente uma árvore binária de busca (do inglês, binary search **tr**ee) (Seção 6.2) e uma h**eap** (Capítulo 10).

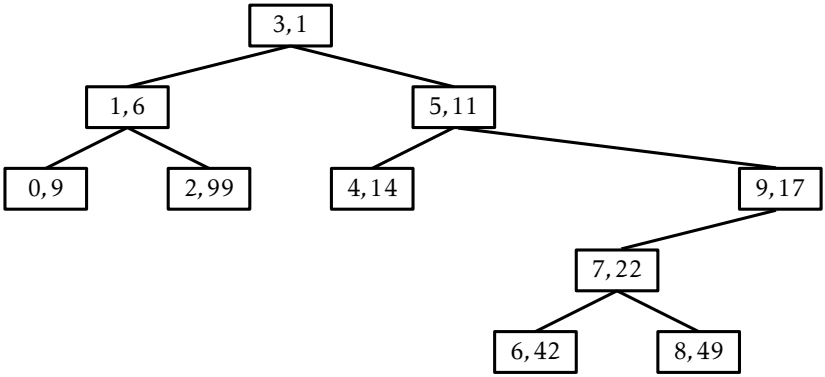


Figura 7.5: Um exemplo de uma Treap contendo os inteiros $0,\ldots,9$. Cada nodo, u, é ilustrado como uma caixa contendo u.x, u.p.

cada nodo estejam definidos, a forma da Treap está completamente determinada.

A propriedade das heaps nos diz que o nodo com prioridade mínima tem que ser a raiz r da Treap. A propriedade das árvores binárias de busca nos diz que todos os nodos com chaves menores que r.x são guardados na subárvores enraizada na r.left e todos os nodos com chaves maiores que r.x são guardados na subárvore enraizada em r.right.

Um fato importante sobre os valores de prioridades em uma Treap é que ele são únicos e atribuídos aleatoriamente. Por causa disso, existem duas formas equivalentes que podem pensar sobre uma Treap. Conforme definido acima, uma Treap obedece as propriedades das árvores binárias de busca e das heaps. Alternativamente, podemos usar uma Treap como uma BinarySearchTree cujos nodos são adicionados em ordem crescente de prioridade. Por exemplo, a Treap na Figura 7.5 pode ser obtida ao adicionar a sequência (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 BinarySearchTree.

Como as prioridades são escolhidas aleatoriamente, isso é equivalente a obter uma permutação aleatória das chaves — nesse caso a permutação

é

$$\langle 3, 1, 0, 5, 9, 4, 7, 6, 8, 2 \rangle$$

—e inseri-las à uma BinarySearchTree.

Mas isso significa que a forma de uma treap é idêntica àquela de uma árvore binária de busca binária. Em particular, se substituímos cada chave x por seu rank, [3] então o Lema 7.1 aplica-se.

Reafirmando o Lema 7.1 em termos de Treaps, temos:

Lema 7.2. *Em uma Treap que guarda um conjunto S de n chaves, as seguintes afirmações valem:*

1. *Para qualquer $x \in S$, o comprimento esperado do caminho de busca por x é $H_{r(x)+1} + H_{n-r(x)} - O(1)$.*

2. *Para qualquer $x \notin S$, comprimento esperado do caminho de busca por x é $H_{r(x)} + H_{n-r(x)}$.*

Aqui, $r(x)$ denota o rank x no conjunto $S \cup \{x\}$.

Novamente, enfatizamos que o valor esperado no Lema 7.2 é obtido sobre escolhas aleatórias das prioridades para cada nodo. Isso não requer quaisquer premissas sobre a aleatoriedade nas chaves.

O Lema 7.2 nos diz que Treaps podem implementar a operação find(x) eficientemente. Contudo, o benefício real de uma Treap é que ela pode implementar as operações add(x) e delete(x). Para fazer isso, ela precisa realizar rotações para manter a propriedade das heaps. Veja a Figura 7.6. Uma *rotação* em uma árvore binária de busca é uma modificação local que pega um pai u de um nodo w e torna w o pai de u, preservando a propriedade das árvores binárias de busca. Rotações vêm em dois sabores: à *esquerda* e à *direita* dependendo em se w é um filho à direita ou esquerda de u, respectivamente.

O código que implementa isso tem que lidar com essas duas possibilidades e ser cuidadoso com um caso especial (quando u for a raiz), então o código real é um pouco mais longo que a Figura 7.6 daria a entender:

[3] O rank de um elemento x em um conjunto S de elementos é o número de elementos em S que são menores que x.

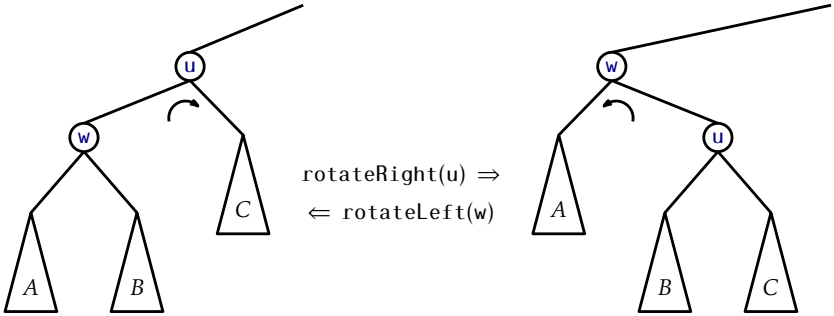


Figura 7.6: Rotações à esquerda e direita em uma árvore binária de busca.

BinarySearchTree

```
void rotateLeft(Node u) {
  Node w = u.right;
  w.parent = u.parent;
  if (w.parent != nil) {
    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 é reduzida em 1 enquanto a profundidade de w aumenta em 1.

Usando rotações, podemos implementar a operação add(x) da forma a seguir: criamos um nodo u, atribuímos u.x = x e pegamos um valor aleatório para u.p. Depois adicionamos u usando o algoritmo add(x) usual para uma BinarySearchTree, então u é agora uma folha da Treap. Nesse ponto, nossa Treap satisfaz a propriedade das árvores binárias de busca, mas não necessariamente a propriedade das heaps.

Em particular, pode ser o caso que u.parent.p > u.p. Se esse é o caso, então realizamos uma rotação no nodo w=u.parent tal que u se torna o pai de w.

Se u continua a violar a propriedade das heaps, teremos que repetir isso, decrescendo a profundidade de u em um a cada vez, até que u se torne raiz ou u.parent.p < u.p.

Treap

```
boolean add(T x) {
  Node<T> u = newNode();
  u.x = x;
  u.p = rand.nextInt();
  if (super.add(u)) {
    bubbleUp(u);
    return true;
  }
  return false;
}
void bubbleUp(Node<T> 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 mostrado na Figura 7.7.

O tempo de execução da operação add(x) é dado pelo tempo que leva para seguir o caminho de busca para x mais o número de rotações realizadas para mover o novo nodo u subindo a árvore até sua posição correta na Treap. De acordo com o Lema 7.2, o comprimento esperado do caminho de busca é no máximo $2\ln n + O(1)$. Além disso, cada rotação reduz a profundidade de u. Isso para se u se torna raiz, então o número esperado de rotações não pode exceder o comprimento esperado do caminho de busca. Portanto o tempo esperado de execução da operação add(x) em uma Treap é $O(\log n)$. (o Exercício 7.5 pede que se mostre que o número esperado de rotações realizadas durante uma adição é na verdade somente $O(1)$.)

A operação remove(x) em uma Treap é o oposto da operação add(x). Buscamos pelo nodo u, contendo x, então realizamos rotações para mover u abaixo na árvore até que se torne uma folha e então removemos u da Treap. Note que, para mover u árvore abaixo, podemos fazer uma rotação à esquerda ou direita em u, que substitui u com u.right ou u.left, respectivamente. A escolha é feita de acordo com a primeira das seguintes regras que for adequada:

1. Se u.left e u.right forem ambos null, então u é uma folha e nenhuma rotação é realizada.

2. Se u.left (ou u.right) for null, então realizamos uma rotação à direita (ou à esquerda, respectivamente) em u.

3. Se u.left.p < u.right.p (ou u.left.p > u.right.p), então realizamos uma rotação à direita (ou rotação à esquerda, respectivamente) em

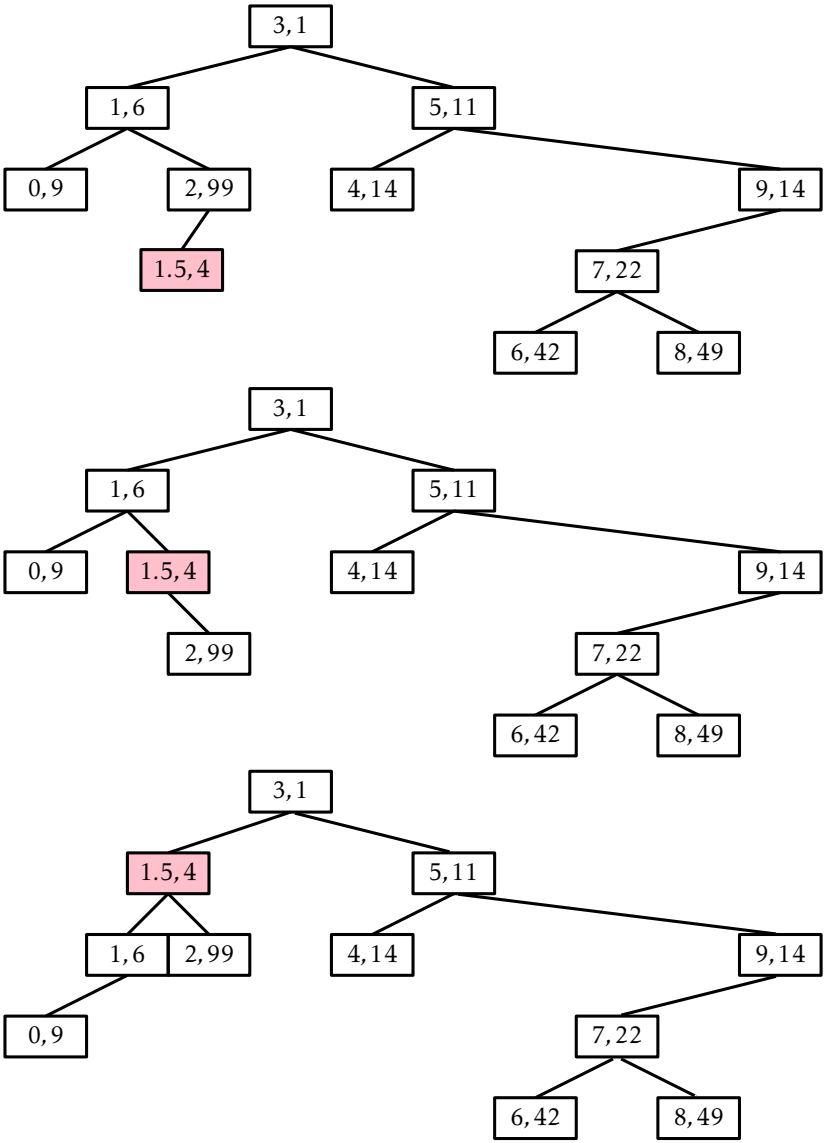


Figura 7.7: Adicionando o valor 1.5 na Treap na Figura 7.5.

u.

Essas três regras asseguram que Treap não se torne desconectada e que a propriedade das heaps é restabelecida uma vez que u seja removida.

Treap

```
boolean remove(T x) {
  Node<T> u = findLast(x);
  if (u != nil && compare(u.x, x) == 0) {
    trickleDown(u);
    splice(u);
    return true;
  }
  return false;
}
void trickleDown(Node<T> 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 essa operação inverte a operação add(x). Em particular, se fossemos reinserir x usando a mesmo prioridade u.p, então a operação add(x) faria exatamente o mesmo número de rotações e reestabeleceria a Treap para exatamente o mesmo estado que estava antes que a operação remove(x) ocorreu.

(Lendo de cima para baixo, a Figura 7.8 ilustra a adição do valor 9

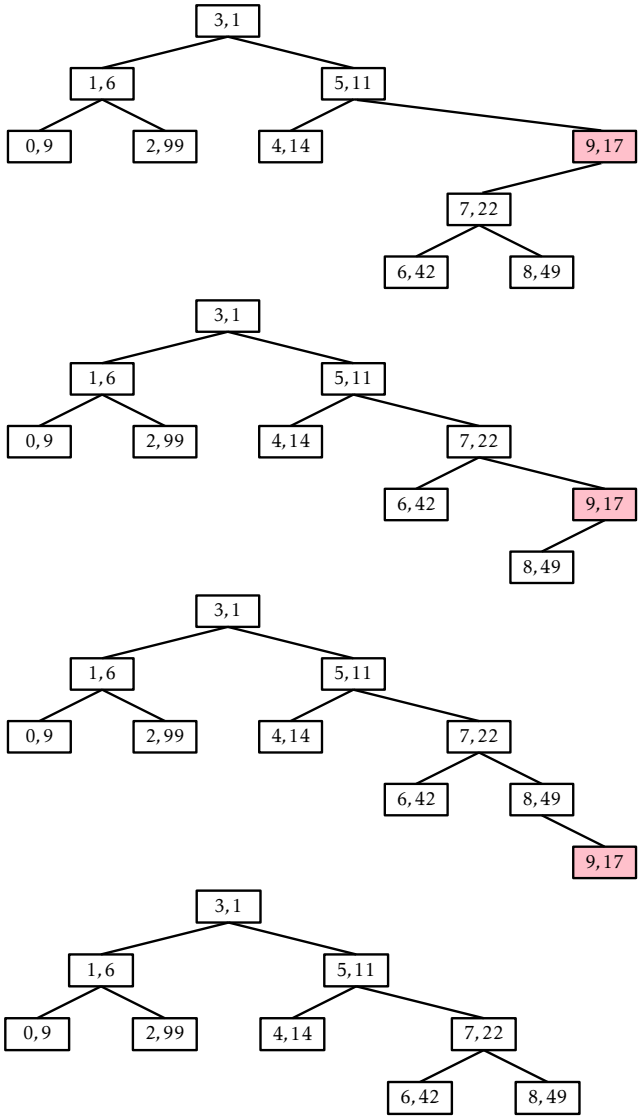


Figura 7.8: Removendo o valor 9 da Treap na Figura 7.5.

em uma Treap.) Isso 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 $O(\log n)$.

7.2.1 Resumo

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

Teorema 7.2. *Uma Treap implementa a interface SSet. Uma Treap executa as operações* add(x), remove(x) *e* find(x) *em tempo esperado* $O(\log n)$ *por operação.*

Vale a pena comparar a estrutura de dados Treap à estrutura de dados SkiplistSSet. Ambas implementam operações SSet em $O(\log n)$ de tempo esperado por operação. Na duas estruturas de dados, add(x) e remove(x) envolvem uma busca e então um número constante de mudanças de ponteiros (veja a Exercício 7.5 a seguir). Então, para essas estruturas, o comprimento esperado do caminho de busca é um valor crítico na avaliação de 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

$$2\ln n + O(1) \approx 1.386\log n + O(1) \ .$$

Então, os caminhos de busca em um Treap são consideravelmente mais curtos e isso traduz em operações mais rápidas em Treaps que Skiplists. O Exercício 4.7 no Capítulo 4 mostra como o comprimento esperado do caminho de busca em uma Skiplist pode ser reduzido a

$$e\ln n + O(1) \approx 1.884\log n + O(1)$$

pelo uso de lançamentos de moedas tendenciosas. Mesmo com essa otimização, o comprimento esperado de caminhos de busca em uma SkiplistSSet é notavelmente maior que em uma Treap.

7.3 Discussão e Exercícios

Árvores binárias de busca aleatórias têm sido estudadas extensivamente. Devroye [19] provou o Lema 7.1 e outros resultados relacionados. Há resultados bem mais precisos na literatura também, o mais impressionante é de Reed [64], que mostra que a altura esperada de uma árvore binária de busca aleatória é

$$\alpha \ln n - \beta \ln \ln n + O(1)$$

onde $\alpha \approx 4.31107$ é a única solução 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 cunhado por Seidel e Aragon [67] que discutiram as propriedades das Treaps e algumas de suas variantes. Entretanto, a estrutura básica da Treap foi estudada muito antes por Vuillemin [76] que chamou elas de árvores cartesianas.

Uma possibilidade de otimização relacionada ao uso de memória da estrutura de dados Treap é a eliminação do armazenamento explícito da prioridade p em cada nodo. Em vez disso, a prioridade de um nodo u é computado usando o hashing do endereço de u em memória (no Java 32-bits, isso é equivalente a fazer o hashing de u.hashCode()). Embora muitas funções hash provavelmente funcionam bem para isso na prática, para que as partes importantes da prova do Lema 7.1 continuarem a serem válidas, a função hash deve ser randomizada e ter a *propriedade de independência em relação ao mínimo*: Para quaisquer valores distintos $x_1, \ldots, x_k$, cada um dos valores hash $h(x_1), \ldots, h(x_k)$ devem ser distintos 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. Uma classe de funções hash desse tipo que é fácil de implementar e razoavelmente rápida é o *hashing por tabulação* (Seção 5.2.3).

Outra variante Treap que não guarda a prioridade em cada nodo é a árvore binária de busca randomizada. de Martínez e Roura [51]. Nessa variante, todo nodo u guarda o tamanho u.size da subárvore enraizada em u. Ambos algoritmos add(x) e remove(x) são randomizados. O algoritmo para adicionar x à subárvore enraizada em u faz o seguinte:

1. Com probabilidade $1/(\texttt{size}(\texttt{u})+1)$, o valor x é adicionado normalmente, como uma folha, e rotações são então feitas para trazer x à raiz dessa subárvore.

2. Caso contrário, (com probabilidade $1-1/(\texttt{size}(\texttt{u})+1)$), o valor x é recursivamente adicionado em uma das duas subárvores enraizadas em u.left ou u.right, conforme apropriado.

O primeiro caso corresponde a uma operação add(x) em uma Treap onde o nodo de x recebe uma prioridade aleatória que é menor que qualquer uma das prioridades na subárvore de u e esse caso ocorre com exatamente a mesma probabilidade.

A remoção de um valor x de uma árvore binária de busca randomizada é similar ao processo de remoção de uma Treap. Achamos o nodo u que contém x e então realizamos rotações que repetidamente aumentar a profundidade de u até tornar-se uma folha, onde podemos remover da árvore. A decisão de fazer rotações à esquerda ou direita é randomizada.

1. Com probabilidade $\texttt{u.left.size}/(\texttt{u.size}-1)$, fazemos uma rotação à direita em u, fazendo u.left a raiz da subárvore que anteriormente estava enraizada em u.

2. Com probabilidade $\texttt{u.right.size}/(\texttt{u.size}-1)$, realizamos uma rotação à esquerda em u, fazendo u.right a raiz da subárvore que estava anteriormente enraizada em u.

Novamente, podemos facilmente verificar que essas são exatamente as mesmas probabilidades que o algoritmo de remoção em uma Treap terá para fazer rotação à esquerda ou direita de u.

Árvores binárias de busca randomizada têm a desvantagem, em relação às treaps, de que ao adicionar ou remover elementos elas farão muitas escolhas aleatórias e precisam manter os tamanhos das subárvores. Uma vantagem de árvores binárias de busca randomizada em relação às treaps é que os tamanhos de subárvores pode servir para outro uso: prover acesso por rank em tempo esperado $O(\log n)$ (veja o Exercício 7.10). Em comparação, as prioridades aleatórias guardadas em nodos de uma treap somente são úteis para manter a treap balanceada.

Exercício 7.1. Simule a adição de 4.5 (com prioridade 7) e então 7.5 (com prioridade 20) na `Treap` da Figura 7.5.

Exercício 7.2. Simule a remoção do elemento 5 e então do elemento 7 na `Treap` da Figura 7.5.

Exercício 7.3. Prove que existem $21,964,800$ sequências que geram a árvore do lado direito da Figura 7.1. (Dica: obtenha uma fórmula recursiva para o número de sequências que geram uma árvore binária de altura h e use essa fórmula para $h = 3$.)

Exercício 7.4. Projete e implemente o método `permute(a)` que contém `n` valores distintos e aleatoriamente permuta `a`. O método deve rodar em $O(n)$ de tempo e você deve provar que cada uma das `n`! possíveis permutações de `a` é igualmente provável.

Exercício 7.5. Use ambas partes do Lema 7.2 para provar que o número esperado de rotações realizadas por uma operação `add(x)` (e portanto uma operação `remove(x)`) é $O(1)$.

Exercício 7.6. Modifique a implementação `Treap` dada aqui para que não guarde explicitamente os valores de prioridades. Em vez disso, você deve simular esses valores com o hashing de cada nodo usando `hashCode()`.

Exercício 7.7. Suponha que uma árvore binária de busca guarda em cada nodo `u` a altura `u.height` da subárvore enraizada em `u` e o tamanho `u.size` da subárvore enraizada em `u`.

1. Mostre que, se fizermos uma rotação à esquerda ou direita em `u`, então essas duas quantidades podem ser atualizadas em tempo constante para todos os nodos afetados pela rotação.

2. Explique porque o mesmo resultado não é possível se tentarmos também guardar a profundidade `u.depth` de cada nodo `u`.

Exercício 7.8. Projete e implemente um algoritmo que constrói uma `Treap` a partir de um array ordenado `a` de `n` elementos. Esse método deve rodar em $O(n)$ de tempo no pior caso e deve construir uma `Treap` que é indistinguível de uma em que os elementos de `a` foram adicionados um por vez usando o método `add(x)`.

Exercício 7.9. Este exercício trabalha os detalhes de como é possível eficientemente fazer buscas em uma `Treap` dado um ponteiro que está próximo ao nodo que estamos procurando.

1. Projete e implemente uma `Treap` em que cada nodo registra os valores mínimos e máximos em sua subárvore.

2. Usando essa informação extra, adicione um método `fingerFind(x,u)` que executa a operação `find(x)` com a ajuda de um ponteiro para o nodo `u` (que espera-se que não esteja distante do nodo que contém `x`). Essa operação deve iniciar em `u` subir na árvore até que alcance um nodo `w` tal que `w.min` $\leq$ `x` $\leq$ `w.max`. A partir desse ponto, deve-se realizar uma busca padrão por `x` partindo de `w`. (É possível mostrar que `fingerFind(x,u)` leva $O(1 + \log r)$ de tempo, onde r é o número de elementos em uma treap cujo valor está entre `x` e `u.x`.)

3. Estenda sua implementação em uma versão de treap que inicia todas as operações `find(x)` a partir do nodo mais recentemente encontrado por `find(x)`.

Exercício 7.10. Projete e implemente uma versão de `Treap` que inclui uma operação `get(i)` que retorna a chave com rank `i` na `Treap`. (Dica: faça que cada nodo `u` registre o tamanho da subárvore enraizada em `u`.)

Exercício 7.11. Codifique uma `TreapList`, uma implementação de uma interface `List` na forma de uma treap. Cada nodo na treap deve guardar um item da lista e uma travessia em-ordem na treap encontra os itens na mesma ordem que ocorrem na lista. Todas as operações da `List`, `get(i)`, `set(i,x)`, `add(i,x)` e `remove(i)` devem rodar em tempo esperado $O(\log n)$.

Exercício 7.12. Projete e implemente uma versão de uma `Treap` que aceita a operação `split(x)`. Essa operação remove todos os valores da `Treap` que são 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` contendo todos esses valores. A operação `split(x)` deve rodar em tempo esperado $O(\log n)$.
Aviso: Para essa modificação funcionar adequadamente e ainda possibilitar o método `size()` rodar em tempo constante, é necessário implementar as modificações no Exercício 7.10.

Exercício 7.13. Projete e implemente uma versão de uma `Treap` que aceita a operação `absorb(t2)`, que pode ser pensada como o inverso da operação `split(x)`. Essa operação remove todos os valores da `Treap` `t2` e os adiciona ao receptor. Essa operação pressupõe que o menor valor em `t2` é maior que o maior valor no receptor. A operação `absorb(t2)` deve rodar em tempo esperado $O(\log n)$.

Exercício 7.14. Implemente a árvore binária de busca randomizada de Martinez conforme discutido nesta seção. Compare o desempenho da sua implementação com a implementação da `Treap`.

Capítulo 8

Árvores Scapegoat

Neste capítulo, estudamos uma estrutura de dados de árvore binária de busca, a ScapegoatTree. Essa estrutura é baseada na ideia popular de que quando algo sai errado, a primeira coisa que pessoas tendem a fazer é achar alguém para culpar (o bode expiatório, em inglês *scapegoat*). Uma vez que culpa está estabelecida, podemos deixar que o bode expiatório resolva o problema.

Uma ScapegoatTree se mantém balanceada usando *operações de reconstrução parcial* Durante uma operação de reconstrução parcial, uma subárvore inteira é desconstruída e reconstruída em uma subárvore perfeitamente balanceada. Existem muitas forma de reconstrução de uma subárvore enraizada no nodo u em uma árvore perfeitamente balanceada. Uma das mais simples é percorrer a subárvore de u, coletando todos seu nodos em um array, a e então recursivamente construir uma subárvore balanceada usando a.

Se fazemos $m = a.length/2$, então o elemento $a[m]$ torna-se a raiz da nova subárvore, $a[0],\ldots,a[m-1]$ é armazenada recursivamente na subárvore à esquerda e $a[m+1],\ldots,a[a.length-1]$ é armazenada na subárvore à direita.

ScapegoatTree

```
void rebuild(Node<T> u) {
  int ns = size(u);
  Node<T> p = u.parent;
  Node<T>[] a =  Array.newInstance(Node.class, 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;
  }
}
int packIntoArray(Node<T> u, Node<T>[] a, int i) {
  if (u == nil) {
    return i;
  }
  i = packIntoArray(u.left, a, i);
  a[i++] = u;
  return packIntoArray(u.right, a, i);
}
Node<T> buildBalanced(Node<T>[] a, int i, int ns) {
  if (ns == 0)
    return nil;
  int m = ns / 2;
  a[i + m].left = buildBalanced(a, i, m);
  if (a[i + m].left != nil)
    a[i + m].left.parent = a[i + m];
  a[i + m].right = buildBalanced(a, i + m + 1, ns - m - 1);
  if (a[i + m].right != nil)
    a[i + m].right.parent = a[i + m];
  return a[i + m];
}
```

Uma chamada a `rebuild(u)` leva tempo $O(\texttt{size}(\texttt{u}))$. A subárvore resultante tem altura mínima; não há árvore de menor altura com `size(u)` nodos.

8.1 ScapegoatTree: Uma Árvore Binária de Busca com Reconstrução Parcial

Uma ScapegoatTree é uma BinarySearchTree que além de registrar o número n de nodos na árvore também mantém um contador q que mantém um limitante superior no número de nodos.

ScapegoatTree

```
int q;
```

Durante todo seu uso, n e q seguem as seguintes desigualdades:

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

Além disso, uma ScapegoatTree tem altura logarítmica; e a altura da árvore scapegoat não excede:

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

Mesmo com essa restrição, uma ScapegoatTree pode parecer surpreendentemente desbalanceada. A árvore na Figura 8.1 tem $q = n = 10$ e altura $5 < \log_{3/2} 10 \approx 5.679$.

A implementação da operação find(x) em uma ScapegoatTree é feita usando o algoritmo padrão para buscas em uma BinarySearchTree (veja a Seção 6.2). Isso leva tempo proporcional à altura da árvore que, de acordo com a (8.1), é $O(\log n)$.

Para implementar a operação add(x), primeiramente incrementamos n e q e então usamos o algoritmo usual para adicionar x a uma árvore binária de busca; buscamos por x e então adicionamos uma nova filha u com u.x = x. A esse ponto, podemos ter sorte e a profundidade de u pode não exceder $\log_{3/2} q$. Se assim for, então deixamos a árvore como está e não fazemos mais nada.

Infelizmente, em algumas vezes pode acontecer que $\mathrm{depth}(u) > \log_{3/2} q$. Nesse caso, precisamos reduzir essa altura. Isso não é nada demais; há somente um nodo, u, cuja profundidade excede $\log_{3/2} q$. Para arrumar u, saímos de u para subir na árvore em busca de um *bode expiatório* w. Esse bode expiatório é um nodo muito desbalanceado. Ele tem a seguinte

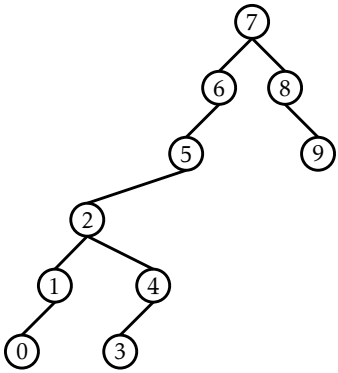


Figura 8.1: Uma ScapegoatTree com 10 nodos e altura 5.

propriedade

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

onde `w.child` é o filho de `w` no caminho da raiz a `u`. Iremos rapidamente provar que um bode expiatório existe. Por agora, podemos assumir que ele existe. Uma vez que encontramos o bode expiatório `w`, podemos completamente destruir a subárvore enraizada em `w` e reconstruí-la em uma árvore binária de busca perfeitamente balanceada. Sabemos da (8.2) que mesmo antes da adição de `u`, a subárvore de `w` não era uma árvore binária completa. Portanto, ao reconstruirmos `w`, a altura decresce por pelo menos 1 para que a altura de ScapegoatTree seja novamente até $\log_{3/2} q$.

ScapegoatTree

```
boolean add(T x) {
  // first do basic insertion keeping track of depth
  Node<T> u = newNode(x);
  int d = addWithDepth(u);
  if (d > log32(q)) {
    // depth exceeded, find scapegoat
    Node<T> w = u.parent;
    while (3*size(w) <= 2*size(w.parent))
      w = w.parent;
```

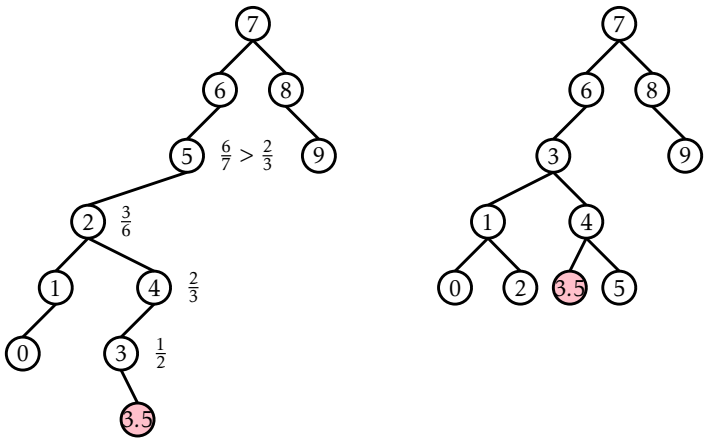


Figura 8.2: Inserindo 3.5 em uma ScapegoatTree aumenta sua altura para 6, o que viola a (8.1) pois $6 > \log_{3/2} 11 \approx 5.914$. Um bode expiatório é achado no nodo contendo 5.

```
    rebuild(w.parent);
  }
  return d >= 0;
}
```

Se ignorarmos o custo de achar o bode expiatório w e reconstruirmos a subárvore enraizada em w, então o tempo de execução de add(x) é dominado pela busca inicial, o que leva $O(\log q) = O(\log n)$ de tempo. Iremos considerar o custo de encontrar o bode expiatório e de reconstrução usando análise amortizada na seção a seguir.

A implementação de remove(x) em uma ScapegoatTree é muito simples. Buscamos por x e o removemos usando o algoritmo usual para remover um nodo de uma BinarySearchTree. (Note que isso nunca aumenta a altura da árvore.) A seguir, decrementamos n, mas não alteramos q. Finalmente, verificamos se $q > 2n$ e, caso positivo, então *reconstruímos a árvore inteira* em uma árvore binária perfeitamente balanceada e atribuímos $q = n$.

ScapegoatTree

```
boolean remove(T x) {
```

```
  if (super.remove(x)) {
    if (2*n < q) {
      rebuild(r);
      q = n;
    }
    return true;
  }
  return false;
}
```

Novamente, se ignorarmos o custo de recontrução, o tempo de execução da operação `remove(x)` é proporcional à altura da árvore, que é $O(\log n)$.

8.1.1 Análise de Corretude e Tempo de Execução

Nesta seção, analisaremos a corretude e o tempo amortizado de operações em uma `ScapegoatTree`. Primeiro provamos a corretude ao mostrar que, quando a operação `add(x)` resulta em um nodo que viola a condição da (8.1), então sempre podemos achar um bode expiatório:

Lema 8.1. *Seja* u *um nodo de profundidade* $h > \log_{3/2} q$ *em uma* `ScapegoatTree`*. Então existe um nodo* w *no caminho de* u *à raiz tal que*

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

Demonstração. Suponha, para efeitos de contradição, que esse não é o caso, e

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

para todos os nodos w no caminho de u à raiz. Denote o caminho da raiz a u como $r = u_0, \ldots, u_h = u$. Então, temos $\texttt{size}(u_0) = n$, $\texttt{size}(u_1) \le \frac{2}{3}n$, $\texttt{size}(u_2) \le \frac{4}{9}n$ e, de modo mais geral,

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

Mas isso resulta em uma contradição, pois $\texttt{size}(u) \ge 1$, portanto

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

A seguir analisamos as partes do tempo de execução que ainda não foram levadas em conta. Existem duas partes: o custo das chamadas a `size(u)` ao buscar por nodos bodes expiatórios e o custo de chamadas a `rebuild(w)` quando encontramos um bode expiatório `w`. O custo das chamadas a `size(u)` pode ser relacionado ao custo de chamadas a `rebuild(w)` da seguinte forma:

Lema 8.2. *Durante uma chamada a* `add(x)` *em uma ScapegoatTree, o custo de encontrar o bode expiatório* `w` *e reconstruir a subárvore enraizada em* `w` *é* $O(\texttt{size}(\texttt{w}))$.

Demonstração. O custo de reconstruir o nodo bode expiatório `w` uma vez que o achamos é $O(\texttt{size}(\texttt{w}))$. Ao buscar o nodo bode expiatório, chamamos `size(u)` em uma sequência de nodos $\texttt{u}_0,\ldots,\texttt{u}_k$ até que encontramos um bode expiatório $\texttt{u}_k = \texttt{w}$. Porém, como $\texttt{u}_k$ é o primeiro nodo nessa sequência que é um bode expiatório, sabemos que

$$\texttt{size}(\texttt{u}_i) < \frac{2}{3}\texttt{size}(\texttt{u}_{i+1})$$

para todo $i \in \{0,\ldots,k-2\}$. Portanto, o custo de todas as chamadas a `size(u)` é

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

onde a última linha segue do fato que a soma é uma série geométrica decrescente. □

Somente falta provar um limitante superior no custo de todas as chamadas a `rebuild(u)` durante uma sequência de m operações:

Lema 8.3. *Iniciando com uma ScapegoatTree vazia, uma sequência de m operações* `add(x)` *e* `remove(x)` *faz com que as operações* `rebuild(u)` *usem tempo* $O(m \log m)$.

Demonstração. Para provar isso, iremos usar um *esquema de créditos*. Imaginamos que cada nodo guarda um número de créditos. Cada crédito pode pagar por alguma constante, c, unidades de tempo gastos na reconstrução.

O esquema resulta provê um total de $O(m \log m)$ créditos e toda chamada de `rebuild(u)` é paga com esses créditos guardados em `u`. Durante uma inserção ou deleção, cedemos um crédito a cada nodo no caminho ao nodo inserido, ou removido, `u`. Dessa maneira gastamos até $\log_{3/2} q \leq \log_{3/2} m$ créditos por operação. Durante a remoção também guardamos um crédito adicional de reserva. Então, no total cedemos até $O(m \log m)$ créditos. Tudo o que resta é mostrar que esses créditos são suficientes para pagar por todas as chamadas a `rebuild(u)`. Se chamarmos `rebuild(u)` durante uma inserção, é porque `u` é um bode expiatório. Suponha, sem perda de generalidade, que

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

Usando o fato que

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

deduzimos que

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

e portanto

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

Agora, a última vez que uma subárvore contendo `u` foi reconstruída (ou quando `u` foi inserido, se uma subárvore contendo `u` nunca foi reconstruída), temos

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

Portanto, o número de operações `add(x)` ou `remove(x)` que afetaram `u.left` ou `u.right` desde então é pelo menos

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

e há portanto pelo menos essa quantidade de créditos guardados em u que estão disponíveis para pagar pelo $O(\texttt{size}(u))$ de tempo que leva para chamar rebuild(u).

Se chamamos rebuild(u) durante uma remoção, é porque $q > 2n$. Nesse caso, temos $q - n > n$ créditos guardados em uma reserva e os usamos para pagar pelo $O(n)$ de tempo que leva para reconstruir a raiz. Isso completa a prova. □

8.1.2 Resumo

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

Teorema 8.1. *Uma ScapegoatTree implementa a interface SSet. Ignorando o custo de operações* rebuild(u)*, uma ScapegoatTree possui as operações* add(x)*,* remove(x) *e* find(x) *em tempo* $O(\log n)$ *por operação.*

Além disso, iniciando com uma ScapegoatTree vazia, uma sequência de m *operações* add(x) *e* remove(x) *resulta em um tempo total* $O(m \log m)$ *gasto nas chamadas a* rebuild(u)*.*

8.2 Discussão e Exercícios

O termo *scapegoat tree* é atribuído a Galperin e Rivest [33], que definiram e analizaram essas árvores. Porém, a mesma estrutura foi descoberta anteriormente por [5, 7], que as chamou de *árvores balanceadas gerais (originalmente, em inglês, general balanced trees)* pois elas podem ter qualquer forma desde que a altura seja pequena.

Experimentos com a implementação ScapegoatTree mostram que ela costuma ser consideravelmente mais lenta que as outras implementações de SSet neste livro. Isso pode ser surpreendente pois a altura é limitada a

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

que é melhor que o comprimento esperado de um caminho de busca em uma Skiplist e não muito longe de um em uma Treap.

A implementação pode ser otimizada ao armazenar os tamanhos das

subárvores explicitamente em cada nodo ou reutilizando tamanhos de subárvores previamente computados (Exercícios 8.5 e 8.6). Mesmo com essas otimizações, sempre haverão sequências de operações add(x) e delete(x) para as quais uma ScapegoatTree leva mais tempo que outras implementações da SSet.

Essa diferença em desempenho é devido ao fato que, diferentemente de outras implementações SSet discutidas neste livro, uma ScapegoatTree pode gastar muito tempo se reestruturando. O Exercício 8.3 pede que você prove que existem sequências de n operações em que uma ScapegoatTree irá gastar $n \log n$ de tempo em chamadas a rebuild(u). Isso contrasta a outras implementações SSet discutidas neste livro, que fazem somente $O(n)$ mudanças estruturais durante uma sequência de n operações. Isto é, infelizmente, uma consequência necessária do fato que uma ScapegoatTree faz sua restruturação usando chamadas a rebuild(u) [20].

Apesar de seu desempenho relativamente ruim, há aplicações em que uma ScapegoatTree pode ser a escolha certa. Isso ocorre quando há dados adicionais associados a nodos que não podem ser atualizados em tempo constante quando uma rotação é realizada mas que podem ser atualizados em uma operação rebuild(u). Nesses casos, a ScapegoatTree e outras estruturas similares baseadas em reconstrução parcial podem funcionar bem. Um exemplo de tal aplicação é esboçado no Exercício 8.11.

Exercício 8.1. Simule a adição dos valores 1.5 e depois 1.6 na ScapegoatTree na Figura 8.1.

Exercício 8.2. Ilustre o que acontece quando a sequência $1, 5, 2, 4, 3$ é adicionada a uma ScapegoatTree vazia e mostre onde os créditos descritos na prova do Lema 8.3 vão e como eles são usados durante essa sequência de adições.

Exercício 8.3. Mostre que, se começarmos com uma ScapegoatTree vazia e chamarmos add(x) para $x = 1, 2, 3, \ldots, n$, então o tempo total gasto em chamadas a rebuild(u) é pelo menos $cn \log n$ para alguma constante $c > 0$.

Exercício 8.4. A ScapegoatTree, conforme descrita neste capítulo, garante que o comprimento do caminho de busca não excede $\log_{3/2} q$.

1. Projete, analise e implemente uma versão modificada de ScapegoatTree onde o comprimento do caminho de busca que não excede

$\log_b q$, onde b é um parâmetro com $1 < b < 2$.

2. O que sua análise e experimentos dizem sobre o custo amortizado de find(x), add(x) e remove(x) como uma função de n e b?

Exercício 8.5. Modifique o método add(x) da ScapegoatTree para que não gaste tempo recomputando os tamanhos das subárvores que foram anteriormente computados. Isso é possível porque quando o método quer computar size(w), ele já computou size(w.left) ou size(w.right). Compare o desempenho de sua implementação modificada com a implementação fornecida neste livro.

Exercício 8.6. Implemente uma segunda versão da estrutura de dados ScapegoatTree que explicitamente guarda e mantém os tamanhos da subárvore enraizada em cada nodo. Compare o desempenho da implementação resultante com a ScapegoatTree original assim como a implementação feita para o Exercício 8.5.

Exercício 8.7. Reimplemente o método rebuild(u) discutido no começo desse capítulo para que não necessite do uso de um array para guardar os nodos da subárvore sendo reconstruída. Em vez disso, deve-se usar recursão primeiro para conectar os nodos em uma lista ligada e então converter essa lista ligada em uma árvore binária perfeitamente balanceada. (Existem implementações recursivas muito elegantes de ambos os passos.)

Exercício 8.8. Analise e implemente uma WeightBalancedTree. Essa é uma árvore em que cada nodo u, exceto a raiz, mantém a *invariante de balanceamento* $\text{size}(u) \leq (2/3)\text{size}(u.\text{parent})$. As operações add(x) e remove(x) são idênticas às operações padrões da BinarySearchTree, exceto que toda vez que a invariante de balanceamento é violada em um nodo u, a subárvore enraizada em u.parent é reconstruída. A sua análise deve mostrar que operações em uma WeightBalancedTree rodam em tempo amortizado $O(\log n)$.

Exercício 8.9. Analise e implemente uma CountdownTree. Em uma CountdownTree cada nodo u mantém um *timer* u.t. As operações add(x) e remove(x) são exatamente as mesmas que em uma BinarySearchTree padrão exceto que, sempre que uma dessas operações afetam a subárvore u a variável

u.t é decrementada. Quando u.t = 0 a subárvore inteira enraizada em u é reconstruída em uma árvore binária de busca perfeitamente balanceada. Quando um nodo u estiver envolvido em uma operação de reconstrução (porque u foi reconstruída ou um dos ancestrais de u foi reconstruído) u.t é reiniciada a size(u)/3.

Sua análise deve mostrar que as operações em uma CountdownTree roda em tempo amortizado $O(\log n)$. (Dica: primeiro mostre que cada nodo u satisfaz alguma versão de uma invariante de balanceamento.)

Exercício 8.10. Analise e implemente uma DynamiteTree. Em uma DynamiteTree cada nodo u registra o tamanho da subárvore enraizada em u em uma variável u.size. As operações add(x) e remove(x) são exatamente as mesmas que em uma BinarySearchTree padrão exceto que, sempre que uma dessas operações afeta uma subárvore de um nodo u, esse nodo *explode* com probabilidade 1/u.size. Quando u explode, sua subárvore inteira é reconstruída em uma árvore binária de busca perfeitamente balanceada. A sua análise deve mostrar que operações em uma DynamiteTree rodam em tempo esperado $O(\log n)$.

Exercício 8.11. Projete e implemente uma estrutura de dados Sequence que mantém uma sequência (lista) de elementos. Ela suporta as seguintes operações:

- addAfter(e): adiciona um novo elemento após o elemento e na sequência. Retorna o elemento adicionado. (Se e for null, o novo elemento é adicionado no início da sequência.)
- remove(e): Remove e da sequência.
- testBefore(e1, e2): retorna true se e somente se e1 vem antes de e2 na sequência.

As primeiras duas operações devem rodar em tempo amortizado $O(\log n)$. A terceira operação deve rodar em tempo constante.

A estrutura de dados Sequence pode ser implementada na mesma ordem que ocorrem na sequência. Para implementar testBefore(e1, e2) em tempo constante, cada elemento e é marcado com um inteiro que codifica o caminho da raiz até e. Dessa forma, testBefore(e1, e2) pode ser implementada pela comparação das marcações de e1 e e2.

Capítulo 9

Árvores Rubro-Negras

Neste capítulo, apresentamos as árvores rubro-negras (em inglês, red-black trees), uma versão de árvores binárias de busca com altura logarítmica. Árvores rubro-negras são as estruturas de dados mais amplamente usadas. Elas aparecerem como estrutura de busca primária em muitas implementações, incluindo a Java Collections Framework e várias implementações da C++ Standard Template Library. Elas também são usadas dentro do kernel Linux. Existem várias razões para a popularidade das árvores rubro-negras:

1. Uma árvore rubro-negra com n valores tem altura no máximo $2 \log n$.
2. As operações `add(x)` e `remove(x)` em uma árvore rubro-negra rodam em tempo $O(\log n)$ no *pior caso*.
3. O número amortizado de rotações realizado durante uma operação `add(x)` ou `remove(x)` é constante.

As duas primeiras propriedades põem árvores rubro-negras a frente de skiplists, treaps e árvores scapegoat. Skiplists e treaps dependem de randomização e seus tempos de execução $O(\log n)$ somente são esperados. Árvores scapegoat têm um limitante garantido na altura, mas `add(x)` e `remove(x)` rodam em tempo amortizado $O(\log n)$. A terceira propriedade é apenas a cereja do bolo. Ela nos diz que o tempo necessário para adicionar ou remover um elemento x é minúsculo em relação ao tempo que leva para achar x. [1]

[1] Note que skiplists e treaps também têm essas propriedades de forma esperada. Veja os exercícios 4.6 e 7.5.

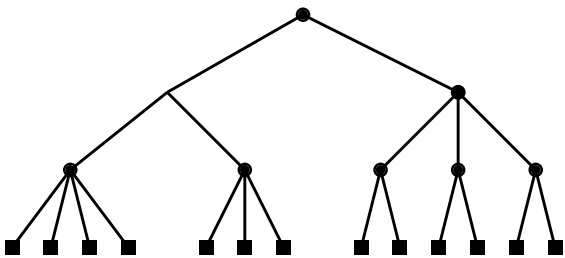


Figura 9.1: Uma árvore 2-4 de altura 3.

Entretanto, as boas propriedades de árvores rubro-negras têm um preço: complexidade de implementação. Manter um limitante de $2 \log n$ na altura não é fácil. Isso exige uma análise cuidadosa de vários casos. Precisamos assegurar que a implementação siga exatamente os passos corretos em cada caso. Uma rotação mal posicionada ou um alteração de cor errada causa um bug que pode ser muito difícil de entender e resolver.

Em vez de pular diretamente à implementação da árvore rubro-negra, primeiro construímos uma base por meio de uma estrutura de dados diretamente relacionada: árvore 2-4. Isso fornecerá entendimento de como árvores rubro-negras foram descobertas e porque é possível manter essa estrutura eficientemente.

9.1 Árvore 2-4

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

Propriedade 9.1 (height)**.** Todas as folhas têm a mesma profundidade.

Propriedade 9.2 (degree)**.** Todo nodo interno tem 2,3 ou 4 filhos.

Um exemplo de uma árvore 2-4 é mostrado na Figura 9.1. As propriedades de uma árvore 2-4 implicam que sua altura é logarítmica no número de folhas:

Lema 9.1. *A árvore 2-4 com* n *folhas tem altura até* $\log n$.

Demonstração. O limitante inferior de 2 no número de filhos de um nodo interno implica que, se altura de uma árvore 2-4 for h, então ela tem pelo

menos 2^h folhas. Em outras palavras,

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

Aplicando logaritmos em ambos lados dessa desigualdade resulta em $h \leq \log n$. □

9.1.1 Adição de uma Folha

Adicionar uma folha a uma árvore 2-4 é fácil (veja a Figura 9.2). Se queremos adicionar uma folha u coma filha de um nodo w no penúltimo nível, então simplesmente fazemos u um filho de w. Isso certamente mantém a propriedade da altura, mas pode violar a propriedade do grau; se w tinha quatro filhos antes de adicionar u, então w agora tem cinco filhos. Neste caso, *repartimos* w em dois nodos, w e w', com dois e três filhos, respectivamente. Mas agora w' não tem pai, então recursivamente tornamos w' um filho do pai de w. Novamente, isso pode fazer com que o pai de w tenha muitos filhos e, dessa forma, teremos que reparti-los. Esse processo segue repetidamente até alcançar um nodo que tem menos que quatro filhos, ou até repartimos a raiz w em dois nodos r e r'. No último caso, fazemos uma nova raiz que tem r e r' como filhos. Isso simultaneamente aumenta a profundidade de todas as folhas e assim mantém propriedade da altura.

Como a altura da árvore 2-4 nunca é maior que $\log n$, o processo de adicionar uma folha termina após no máximo $\log n$ passos.

9.1.2 Remoção de uma Folha

Remoção de uma folha de uma árvore 2-4 é um pouco mais complicada (veja a Figura 9.3). Para remover uma folha u de seu pai w, então simplesmente a removemos. Se w tivesse somente dois filhos antes à remoção de u, então w é deixado com somente um filho, o que viola a propriedade de grau.

Para corrigir isso, olhamos em um irmão de w, o w'. O nodo w' com certeza existe pois o pai de w tem pelo menos dois filhos. Se w' tem três ou quatro filhos, então pegamos um desses filhos de w' e o transferimos para w. Agora w tem dois filhos e w' tem dois ou três filhos e terminamos o processo.

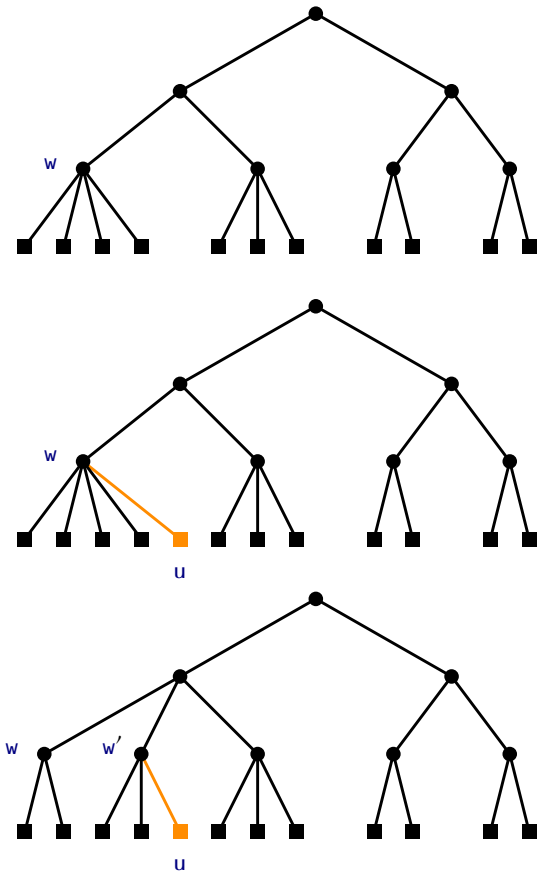


Figura 9.2: Adição de uma folha a uma árvore 2-4. Esse processo para após um repartição porque `w.parent` tem um grau menor que 4 antes da adição.

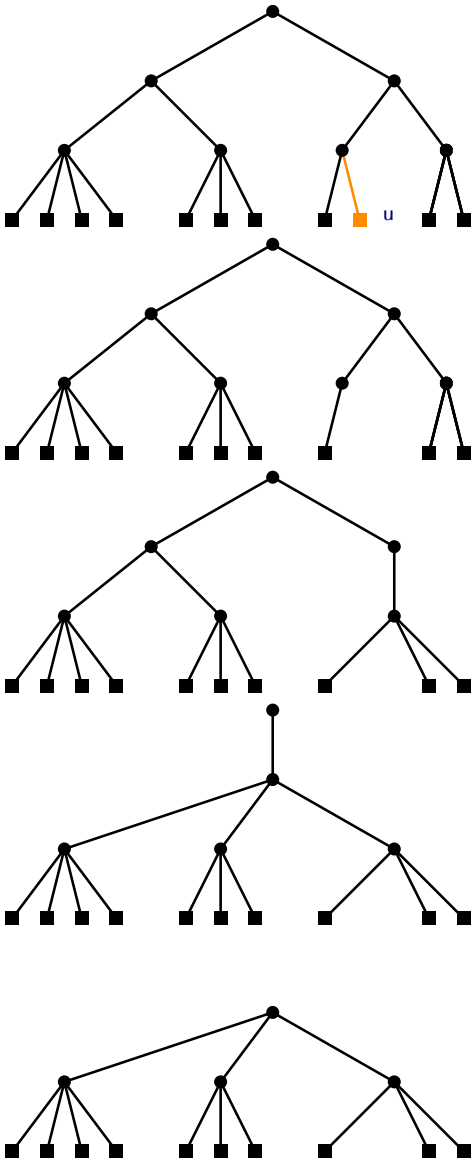


Figura 9.3: Remoção de uma folha de uma árvore 2-4. Esse processo segue até a raiz pois cada ancestral de u e seus irmãos têm somente dois filhos.

Por outro lado, se w′ tem somente dois filhos, então fazemos uma fusão (em inglês, *merge*) de w e w′ em um único nodo, w, que tem três filhos. A seguir, recursivamente removemos w′ do pai de w′. Esse processo termina guando alcançamos um nodo u, onde u ou um irmão seu tem mais de dois filhos, ou quando chegamos à raiz. No último caso, se a raiz é deixada com somente um filho, então removemos a raiz e fazemos seu filho a nova raiz.

Novamente, isso simultaneamente reduz a altura de todas as folhas e portanto mantém a propriedade de altura.

Como a altura da árvore não passa de $\log n$, o processo de remover uma folha acaba após no máximo $\log n$ passos.

9.2 RedBlackTree: Uma Árvore 2-4 Simulada

Uma árvore rubro-negra é uma árvore binária de busca em que cada nodo u tem uma *cor* que é *vermelha* ou *preta*. Vermelho é representado pelo valor 0 e preto pelo valor 1.

RedBlackTree
```
class Node<T> extends BSTNode<Node<T>,T> {
  byte colour;
}
```

Antes e depois de qualquer operação em uma árvore rubro-negra, as seguintes duas propriedades são satisfeitas. Cada propriedade é definida em termos das cores vermelha e preta e em termos dos valores numéricos 0 e 1.

Propriedade 9.3 (black-height)**.** Há o mesmo número de nodos pretos em todo caminho da raiz para a folha. (A soma das cores em qualquer caminho da raiz a uma folha é a mesma.)

Propriedade 9.4 (no-red-edge)**.** Não há nodos vermelhos adjacentes. (Para qualquer nodo u, exceto a raiz, $u.colour + u.parent.colour \geq 1$.)

Note que sempre podemos colorir a raiz r de uma árvore rubro-negra de preto sem violar nenhuma dessas duas propriedades, então iremos assumir que a raiz é preta e os algoritmos para atualizar uma árvore rubro-negra irão manter isso. Outro truque que simplifica na implementação

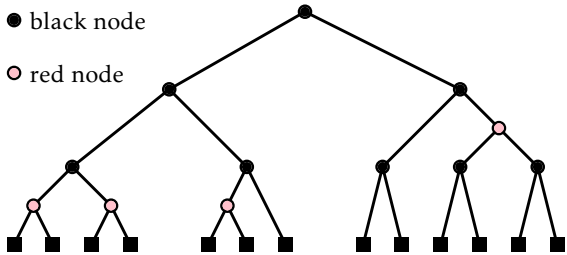


Figura 9.4: Um exemplo de uma árvore rubro-negra com uma altura preta de 3. Nodos externos (nil) são desenhados como quadrados.

é tratar os nodos externos (representados por nil) como nodos pretos. Dessa forma, todo nodo real, u, de uma árvore rubro-negra tem exatamente dois filhos, cada qual com uma cor bem definida. Um exemplo de uma árvore rubro-negra é mostrado na Figura 9.4.

9.2.1 Árvores Rubro-Negras e Árvores 2-4

À primeira vista, pode parecer surpreendente que uma árvore rubro-negra possa ser atualizada eficientemente para manter as propriedades de altura preta e de nenhuma aresta vermelha e parece estranho considerar essas propriedades como úteis. Entretanto, árvores rubro-negras foram projetadas para serem uma simulação eficiente das árvores 2-4 na forma de árvores binárias.

Veja a Figura 9.5. Considere qualquer árvore rubro-negra, T, com n nodos e realize a seguinte transformação: remova cada nodo vermelho u e conecte os dois filhos de u diretamente ao pai (preto) de u. Após essa transformação, a árvore resultante T' possui somente nodos pretos.

Todo nodo interno em T' tem dois, três ou quatro filhos: um nodo preto que tinha dois filhos pretos continuará com dois filhos pretos após essa transformação. Um nodo preto que tinha com um nodo preto e um vermelho terá três filhos após essa transformação. Um nodo preto que começou com dois filhos vermelhos terá quatro filhos após essa transformação. Além disso, a propriedade de altura preta garante que todo caminho da raiz até a folha em T' tem o mesmo comprimento. Em outras palavras, T' é uma árvore 2-4!

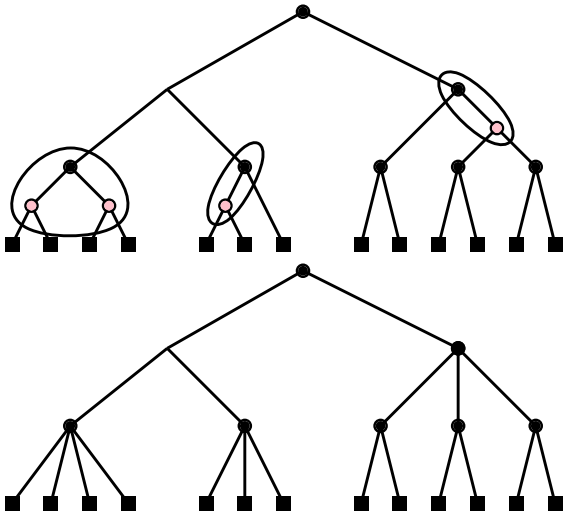


Figura 9.5: Toda árvore rubro-negra tem uma árvore 2-4 correspondente.

A árvore 2-4 T' tem $n+1$ folhas que correspondem a $n+1$ nodos externos da árvore rubro-negra. Portanto, essa árvore tem altura de até $\log(n+1)$. Agora, todo caminho da raiz para uma folha na árvore 2-4 corresponde a um caminho da raiz da árvore rubro-negra T a um nodo externo.

O primeiro e último nodo nesse caminho são pretos e no máximo um de cada dois nodos internos é vermelho, então esse caminho tem no máximo $\log(n+1)$ nodos pretos e no máximo $\log(n+1)-1$ nodos vermelhos. Portanto, o caminho mais longo da raiz para qualquer nodo *interno* em T é no máximo

$$2\log(n+1)-2 \leq 2\log n \ ,$$

para todo $n \geq 1$. Isso é a propriedade mais importante das árvores rubro-negras:

Lema 9.2. *A altura de uma árvore rubro-negra com* n *nodos é no máximo* $2\log n$.

Agora que estudamos a relação entre as árvores 2-4 e as árvores rubro-negras, não é difícil de acreditar que podemos manter eficientemente

uma árvore rubro-negra ao adicionar e remover elementos.

Vimos que a adição de elementos em uma `BinarySearchTree` pode ser feita pela adição de uma nova folha. Portanto, para implementar `add(x)` em uma árvore rubro-negra precisamos de um método de simular a repartição de um nodo com cinco filhos em uma árvore 2-4. O nodo da árvore 2-4 com cinco filhos é representado por um nodo preto que tem dois filhos vermelhos, um dos quais também tem um filho vermelho. Podemos "reparticionar" esse nodo pintando ele de vermelho e pintando seus dois filhos de preto. Um exemplo disso é mostrado na Figura 9.6.

De modo similar, implementar `remove(x)` exige um método de unir dois nodos e emprestar um filho de um irmão. A união de dois nodos é o inverso da repartição (mostrado na Figura 9.6), e envolve pintar dois irmãos vermelhos de preto e pintar o pai (que é vermelho) de preto. Emprestar um irmão é o procedimento mais complicado e envolve ambas as rotações e pintar os nodos.

Obviamente, durante tudo isso devemos ainda manter a propriedade de nenhuma aresta vermelha e a propriedade da altura preta. Enquanto não é mais surpreendente que isso pode ser feito, existe um grande número de casos que devem ser considerados se tentarmos fazer uma simulação direta de uma árvore 2-4 com uma árvore rubro-negra. Eventualmente, torna-se mais simples desconsiderar a árvore 2-4 e trabalhar diretamente com a manutenção das propriedades da árvore rubro-negra.

9.2.2 Árvores Rubro-Negras Pendentes à Esquerda

Não existe uma única definição de árvores rubro-negras. Em vez disso, existe uma família de estruturas que conseguem manter as propriedades da altura preta e nenhuma aresta vermelha durante as operações `add(x)` e `remove(x)`. Estruturas diferentes fazem isso diferentemente. Aqui, implementamos uma estrutura de dados que chamamos de `RedBlackTree`. Essa estrutura implementa uma variante de árvore rubro-negra que satisfaz uma propriedade adicional:

Propriedade 9.5 (left-leaning)**.** Em qualquer nodo `u`, se `u.left` for preto, então `u.right` é preto.

Note que a árvore rubro-negra mostrada na Figura 9.4 não satisfaz a

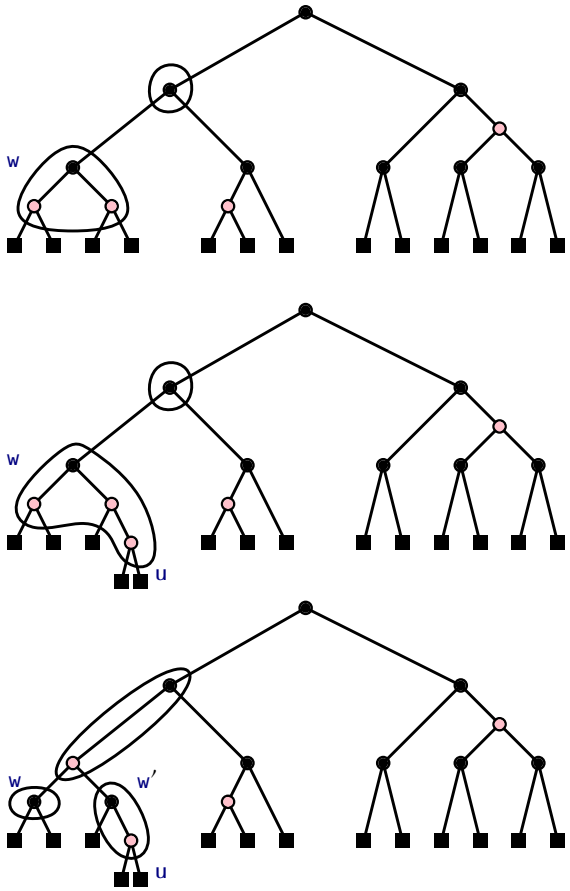


Figura 9.6: Simulando uma operação de reparticionamento de uma árvore 2-4 durante a adição em uma árvore rubro-negra. (Isso simula a adição da árvore 2-4 mostrada na Figura 9.2.)

propriedade de pender à esquerda; ela é violada pelo pai do nodo vermelho no caminho mais à esquerda.

A razão para manter a propriedade de pender à esquerda é que ela reduz o número de casos encontrados durante a atualização da árvore nas operações add(x) e remove(x). Em termos de uma árvore 2-4, isso implica que toda árvore 2-4 tem uma representação única: um nodo de grau dois se torna um nodo preto com dois filhos pretos. Um nodo de grau três se torna um nodo preto cujo filho à esquerda é vermelho e cujo filho à direita é preto. Um nodo de grau quatro se torna um nodo preto com dois filhos vermelhos.

Antes de descrevermos a implementação de add(x) e remove(x) em detalhes, primeiro apresentamos alguma sub-rotinas simples usadas por esses métodos que estão ilustradas na Figura 9.7. As duas primeiras sub-rotinas são para manipular as cores e presentar a propriedade da altura preta. O método pushBlack(u) recebe como entrada um nodo preto u que tem dois filhos vermelhos e pinta u de vermelho e seus dois filhos de preto. O método pullBlack(u) inverte essa operação

RedBlackTree
```
void pushBlack(Node<T> u) {
  u.colour--;
  u.left.colour++;
  u.right.colour++;
}
void pullBlack(Node<T> u) {
  u.colour++;
  u.left.colour--;
  u.right.colour--;
}
```

O método flipLeft(u) troca as cores de u e u.right e então realiza uma rotação à esquerda em u. Esse método inverte as cores desses dois nodos assim como seu relacionamento pai-filho:

RedBlackTree
```
void flipLeft(Node<T> u) {
  swapColors(u, u.right);
  rotateLeft(u);
```

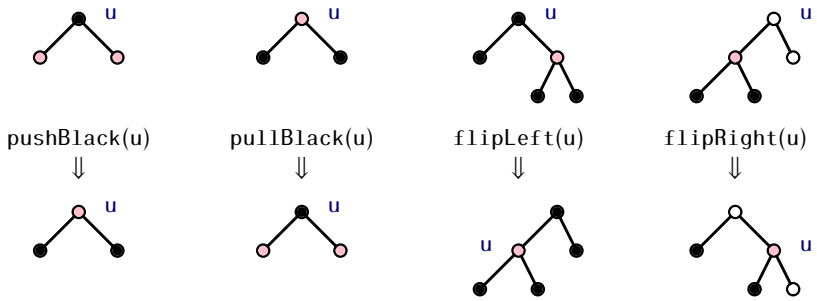


Figura 9.7: Operações de *flips*, *pulls* e *pushes*

```
}
```

A operação `flipLeft(u)` é especialmente útil ao restaurar a propriedade de pender à esquerda em um nodo `u` que a viola (porque `u.left` é preto e `u.right` é vermelho). Nesse caso especial, temos a certeza de que essa operação preserva as propriedades de altura preta e de nenhuma aresta vermelha. A operação `flipRight(u)` é simétrica à `flipLeft(u)`, quando os papéis de esquerda e direita estão invertidos.

RedBlackTree
```
void flipRight(Node<T> u) {
  swapColors(u, u.left);
  rotateRight(u);
}
```

9.2.3 Adição

Para implementar `add(x)` em uma `RedBlackTree`, fazemos uma inserção de um `BinarySearchTree` padrão para adicionar uma nova folha `u`, com `u.x = x` e atribuímos `u.colour = red`. Note que isso não muda a altura preta de nenhum nodo e, portanto, não viola a propriedade de altura preta. Ela pode, entretanto, violar a propriedade de pender à esquerda (se `u` é o filho à direita de seu pai) e pode violar a propriedade de nenhuma aresta vermelha (se o pai de `u` for `red`) Para restaurar essas propriedades, chamamos o método `addFixup(u)`.

```
RedBlackTree
boolean add(T x) {
  Node<T> u = newNode(x);
  u.colour = red;
  boolean added = add(u);
  if (added)
    addFixup(u);
  return added;
}
```

Ilustrado na Figura 9.8, o método `addFixup(u)` recebe como entrada um nodo `u` cuja cor é vermelha e que pode violar a propriedade de nenhuma aresta vermelha e/ou a propriedade pendente à esquerda. A discussão a seguir é provavelmente impossível de acompanhar sem observar a Figura 9.8 ou recriá-la em um papel. Certamente, o leitor deve estudar essa figura antes de continuar a ler este capítulo.

Se `u` for a raiz da árvore, então podemos pintar `u` de preto para reestabelecer as duas propriedades. Se o irmão de `u` também for vermelho, então o pai de `u` precisa ser preto, de forma que as propriedades de pender à esquerda e nenhuma aresta vermelha continuem valendo.

Caso contrário, primeiro determinamos se o pai de `u`, que é `w`, viola a propriedade de pender à esquerda e, se esse for o caso, realizamos uma operação `flipLeft(w)` e atribuímos `u = w`. Isso nos deixa em um estado bem definido: `u` é o filho de seu pai `w` então `w` agora satisfaz a propriedade de pender à esquerda.

Resta apenas assegurarmos que não há nenhuma aresta vermelha em `u`. Nós somente devemos nos preocupar com o caso em que `w` é vermelho, pois caso contrário `u` satisfaz a propriedade de nenhuma aresta vermelha.

Como ainda não terminamos, `u` é vermelho e `w` é vermelho. A propriedade de nenhuma aresta vermelha (que é violada por `u` e não por `w`) implica que o avô de `u`, o nodo `g`, existe e é preto. Se o filho à direita de `g` for vermelho, então a propriedade de pender à esquerda garante que os dois filhos de `g` sejam vermelhos e uma chamada a `pushBlack(g)` torna `g` vermelho e `w` preto. Isso restaura a propriedade nenhuma aresta vermelha em `u`, mas pode fazer que seja violada em `g` e, por isso, o processo reinicia com `u = g`.

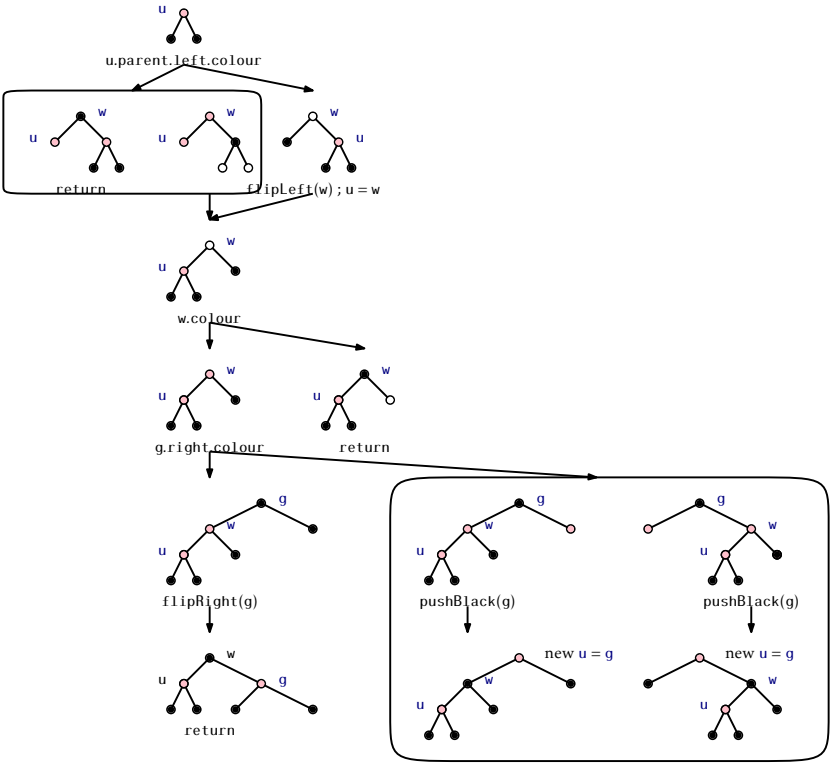


Figura 9.8: Uma única rodada no processo de restaurar a Propriedade 2 após uma inserção.

Se o filho à direita de g for preto, então uma chamada a flipRight(g) faz w ser o pai (preto) de g e dá a w dois filhos vermelhos, u e g. Isso assegura que u satisfaz a propriedade de nenhuma aresta vermelha e g satisfaz a propriedade de pender à esquerda. Nesse caso podemos parar.

RedBlackTree

```
void addFixup(Node<T> u) {
  while (u.colour == red) {
    if (u == r) { // u is the root - done
      u.colour = black;
      return;
    }
    Node<T> 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<T> 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 ou termina ou move u mais próximo à raiz. Portanto, o método insertFixup(u) termina após $O(\log n)$ iterações em tempo $O(\log n)$.

9.2.4 Remoção

A operação remove(x) em uma RedBlackTree é a mais complicada para implementar e isso é verdade para todas as variantes conhecidas de árvores rubro-negras. Assim como a operação remove(x) em uma Binary-

SearchTree, essa operação se resume a achar um nodo w com somente um filho u e remover w da árvore fazendo o w.parent adotar u.

O problema com isso é que, se w for preto, então a propriedade da altura preta será violada em w.parent. Podemos evitar esse problema, temporariamente, ao adicionar w.colour a u.colour. Isso causa dois outros problemas: (1) se u e w iniciaram-se pretos, então u.colour + w.colour = 2 (preto duplo), que é uma cor inválida. Se w era vermelho, então é substituído por um nodo preto u, o que pode violar a propriedade de pender à esquerda em u.parent. Esse dois problemas podem ser resolvidos com uma chamada ao método removeFixup(u).

RedBlackTree

```
boolean remove(T x) {
  Node<T> u = findLast(x);
  if (u == nil || compare(u.x, x) != 0)
    return false;
  Node<T> 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;
  removeFixup(u);
  return true;
}
```

O método removeFixup(u) recebe como entrada um nodo u cuja cor é preto (1) ou preto duplo(2). Se u for preto duplo, então removeFixup(u) realiza uma série de operações de rotações e alterações de cor que movem o nodo preto duplo acima na árvore até que possa ser eliminado. Durante esse processo, o nodo u muda até que, no final desse processo, u refere-se à raiz da subárvore que foi alterada. A raiz dessa subárvore pode ter mu-

dado 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 de pender à esquerda e, caso positivo, corrige esse problema.

RedBlackTree

```
void removeFixup(Node<T> 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<T> w = u.parent;
    if (w.right.colour == red && w.left.colour == black) {
      flipLeft(w);
    }
  }
}
```

O método removeFixup(u) é ilustrado na Figura 9.9. Novamente, o texto a seguir será difícil, senão impossível, de acompanhar sem observar a Figura 9.9. Cada iteração do laço em removeFixup(u) processa o nodo preto duplo u, baseado em um de quatro casos:
Caso 0: u é a raiz. Esse é o caso mais fácil de tratar. Repintamos u em preto (isso não viola nenhuma das propriedades da árvore rubro-negra).
Caso 1: o irmão de u, v, é vermelho. Nesse caso, o irmão de u é filho à esquerda de seu pai, w (de acordo com a propriedade de pender à esquerda). Nós realizamos um right-flip em w e então procedemos à próxima iteração. Note que essa ação faz com que o pai de w viole a propriedade de pender à esquerda e faz a profundidade de u aumentar. Entretanto, isso também implica que a próxima iteração será no Caso 3 com w pintado de vermelho. Ao examinar o Caso 3 a seguir, veremos que o processo irá parar na próxima iteração.

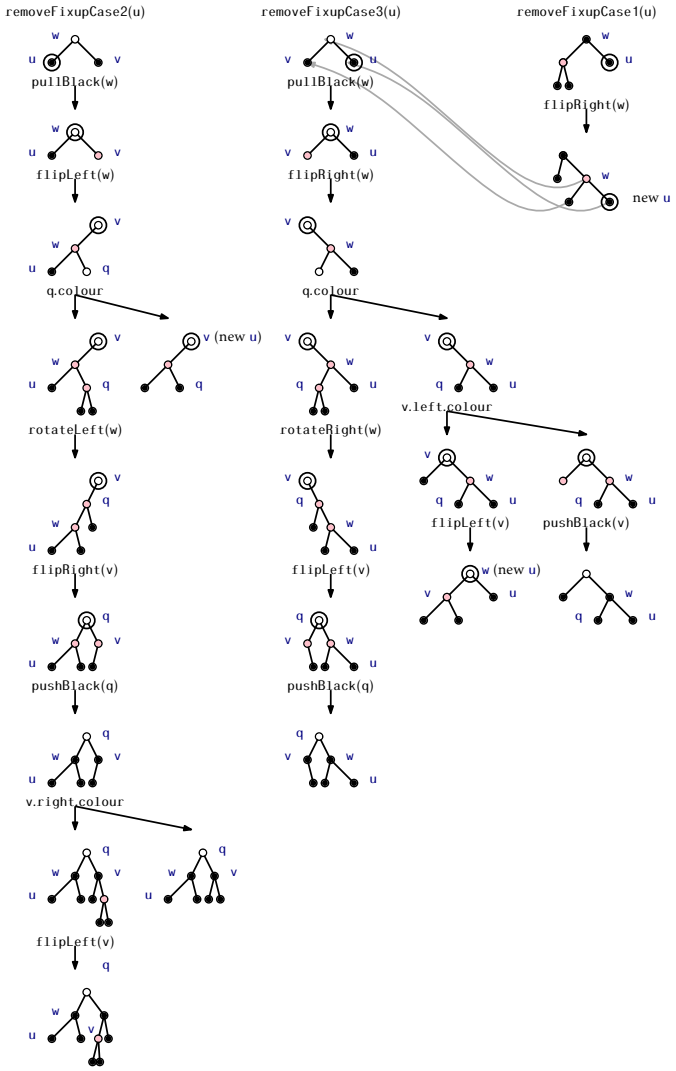


Figura 9.9: Uma rodada no processo de eliminar um nodo preto duplo após uma remoção.

```
RedBlackTree
Node<T> removeFixupCase1(Node<T> u) {
  flipRight(u.parent);
  return u;
}
```

Caso 2: o irmão de u, v, é preto, e u é o filho à esquerda de seu pai, w. Nesse caso, chamamos pullBlack(w), fazendo u preto, v vermelho, e tornando w preto ou duplo preto. Nesse ponto, w não satisfaz a propriedade de pender à esquerda, então chamamos flipLeft(w) para arrumar isso. Além disso, w é vermelho e v é a raiz da subárvore com que começamos. Precisamos verificar se w faz a propriedade de não haver nenhuma aresta vermelha ser violada. Fazemos isso inspecionando o filho à direita de w, q. Se q é preto, então w satisfaz a propriedade de nenhuma aresta vermelha e podemos continuar a próxima iteração com u = v.

Caso contrário (q é vermelho) então as duas propriedades de não haver nenhuma aresta vermelha e de pender à esquerda são violadas em q e w, respectivamente.

A propriedade de pender à esquerda é restaurada com uma chamada a rotateLeft(w), mas a propriedade de não haver nenhuma aresta vermelha continua sendo violada. Nesse ponto, q é o filho à esquerda de v, w é o filho à esquerda de q, q e w são vermelhos e v é preto ou duplo preto. Uma flipRight(v) torna q o pai de v e de w. Em seguida, um pushBlack(q) faz v e w pretos e devolve a cor de q de volta à cor original de w.

Nesse ponto, o nodo duplo preto foi eliminado e as propriedades nenhuma aresta vermelha e altura preta são restabelecidas. Somente resta resolver um problema: o filho à direita de v pode ser vermelho e, nesse caso, a propriedade de pender à esquerda seria violada. Verificamos isso e realizamos, se necessária uma correção, uma chamada a flipLeft(v).

```
RedBlackTree
Node<T> removeFixupCase2(Node<T> u) {
  Node<T> w = u.parent;
  Node<T> v = w.right;
  pullBlack(w); // w.left
  flipLeft(w); // w is now red
  Node<T> 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 à direita de seu pai, w. Esse caso é simétrico ao Caso 2 e é tratado quase da mesma forma. As únicas diferenças vêm do fato que a propriedade de pender à esquerda é assimétrica e, portanto, requer tratamento diferenciado.

Como antes, começamos com uma chamada a pullBlack(w), que torna v vermelho e u preto. Uma chamada a flipRight(w) promove v à raiz da subárvore. Nesse momento, w é vermelho e existem duas formas de tratamento dependendo da cor do filho à esquerda de w, q.

Se q for vermelho, então o código termina exatamente da mesma maneira que o Caso 2, mas é ainda mais simples pois não há perigo de v não satisfazer a propriedade de pender à esquerda.

O caso mais complicado ocorre quando q é preto. Nesse caso, examinamos a cor do filho à esquerda de v. Se for vermelho, então v tem dois filhos vermelhos e seu preto extra pode ser empurrado abaixo com uma chamada a pushBlack(v). Nesse ponto, v agora tem a cor original de w e terminamos o processo.

Se o filho à esquerda de v's for preto, então v viola a propriedade de pender à esquerda e a restauramos com uma chamada a flipLeft(v). Então retornamos o nodo v para que a próxima iteração de removeFixup(u) então continue com u = v.

RedBlackTree

```
Node<T> removeFixupCase3(Node<T> u) {
  Node<T> w = u.parent;
  Node<T> v = w.left;
  pullBlack(w);
```

```
  flipRight(w); // w is now red
  Node<T> 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. Casos 2 e 3 ou terminam ou movem `u` mais próximo à raiz da árvore. Caso 0 (o `u` é a raiz) sempre termina e Caso 1 levam imediatamente ao Caso 3, que também termina. Como a altura da árvore é no máximo $2\log n$, concluímos o processo em até $O(\log n)$ iterações de `removeFixup(u)`, então `removeFixup(u)` roda em tempo $O(\log n)$.

9.3 Resumo

O teorema a seguir resume o desempenho de uma estrutura de dados `RedBlackTree`:

Teorema 9.1. *Uma `RedBlackTree` implementa a interface `SSet` e possui as operações `add(x)`, `remove(x)` e `find(x)` em tempo $O(\log n)$ no pior caso, por operação.*

O bônus extra a seguir não está incluído no teorema anterior:

Teorema 9.2. *Começando com uma `RedBlackTree` vazia, qualquer sequência de m operações `add(x)` e `remove(x)` resulta em um total de tempo $O(m)$ durante todas as chamadas `addFixup(u)` e `removeFixup(u)`.*

Apenas rascunhamos uma prova do Teorema 9.2. Ao comparar as operações `addFixup(u)` e `removeFixup(u)` com os algoritmos para adicionar ou remover uma folha em uma árvore 2-4, podemos nos convencer que essa propriedade é herdada de uma árvore 2-4. Em particular, se podemos mostrar que o tempo total gasto com repartições de nodos, uniões e empréstimos em uma árvore 2-4 é $O(m)$ então isso implica no Teorema 9.2.

A prova desse teorema para a árvore 2-4 usa o método potencial de análise amortizada.[2] Defina o potencial de um nodo 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 de seus nodos. Quando uma repartição ocorre, é porque um nodo com quatro filhos se torna dois nodos, com dois e três filhos. Isso significa que o potencial geral cai em $3 - 1 - 0 = 2$. Quando uma união ocorre, dois modos que tinham dois filhos são substituídos por um nodo com três filhos. O resultado é uma queda em potencial de $2 - 0 = 2$. Portanto, para todas repartição ou união, o potencial diminui em dois.

A seguir note que se ignorarmos a repartição e união de nodos, há somente um número constante de nodo cujo número de filhos é alterado pela adição ou remoção de uma folha. Ao adicionar um nodo, algum nodo tem seu número número de filhos aumentado em um, aumentando o potencial por até três. Durante a remoção de uma folha, um nodo tem seu número de filhos reduzido em um, aumentando o potencial em até um, e dois nodos podem estar envolvidos em uma operação de empréstimo aumentando seu potencial total em até um.

Para resumir, cada união e repartição faz que o potencial caia por ao menos dois. Ignorando uniões e repartições, cada adição ou remoção faz com que o potencial aumente em até três e o potencial sempre é não negativo. Portanto, o número de repartições e uniões causados por m adições ou remoções em uma árvore inicialmente vazia é até $3m/2$. O Teorema 9.2 é uma consequência dessa análise e da correspondência entre árvores 2-4

[2] Veja as prova do Lema 2.2 e do Lema 3.1 para outras aplicações do método potencial.

e árvores rubro-negras.

9.4 Discussão e Exercícios

A árvore rubro-negra foi inicialmente desenvolvida por Guibas e Sedgewick [38]. Embora sua implementação seja de alta complexidade, ela é encontrada em algumas das bibliotecas e aplicações mais utilizadas. A maior parte dos livros didáticos de algoritmos e estruturas de dados discutem alguma variante de árvores rubro-negras.

Andersson [6] descreve uma versão pendente à esquerda de árvores balanceadas que é similar às árvores rubro-negras mas tem a restrição adicional de que todo nodo tem no máximo um filho vermelho. Isso implica que essas árvores simulam árvores 2-3 em vez de árvores 2-4. Ela são significantemente mais simples que a estrutura `RedBlackTree` apresentada neste capítulo.

Sedgewick [66] descreve duas versões de árvores rubro-negras pendentes à esquerda. Elas usam recursão juntamente com uma simulação *top-down* de repartição e união em árvore 2-4. A combinação dessas duas técnicas tornam seus códigos especialmente curtos e elegantes.

Uma estrutura de dados relacionada e mais antiga é a *árvore AVL* [3]. Árvores AVL são *balanceadas na altura*: A cada nodo u, a altura da subárvore enraizada em `u.left` e a subárvore enraizada em `u.right` difere em até um. Segue imediatamente que se $F(h)$ é o número de folhas em uma árvore de altura h, então $F(h)$ segue a recorrência de Fibonacci

$$F(h) = F(h-1) + F(h-2)$$

com casos base $F(0) = 1$ e $F(1) = 1$. Isso significa que $F(h)$ é aproximadamente $\varphi^h/\sqrt{5}$, onde $\varphi = (1+\sqrt{5})/2 \approx 1.61803399$ é a *razão dourada*. (Mais precisamente, $|\varphi^h/\sqrt{5} - F(h)| \leq 1/2$.) Seguindo de modo similar à prova do Lema 9.1, isso implica em

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

então as árvores AVL tem altura menor que árvore rubro-negras. O balanceamento da altura pode ser mantido durante as operações `add(x)` e

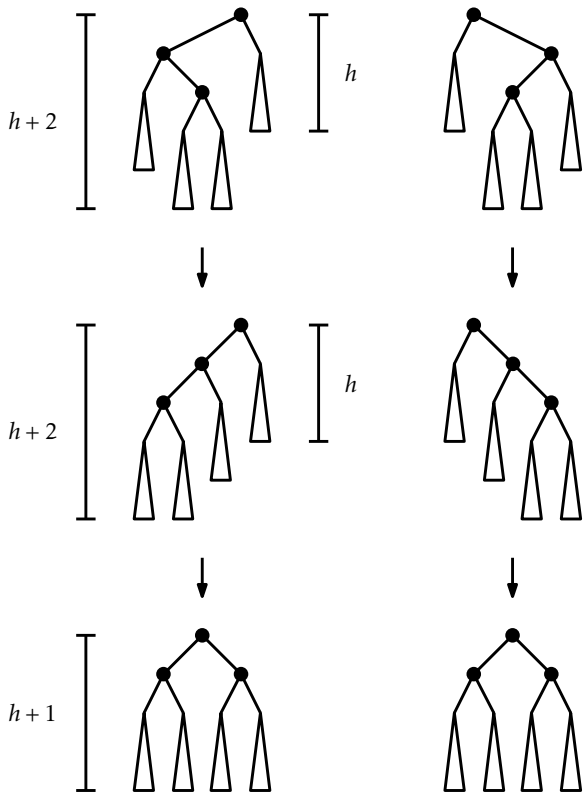


Figura 9.10: Rebalanceamento em uma árvore AVL. No máximo duas rotações são necessárias para converter um nodo cujas subárvores têm alturas de h e $h+2$ em um nodo cujas subárvores têm alturas de até $h+1$

`remove(x)` ao percorrer o caminho em direção à raiz realizando uma operação de rebalanceamento em cada nodo `u` onde a altura das subárvores à esquerda e à direita de `u` difere em dois. Veja a Figura 9.10.

As variantes de árvores rubro-negras de Andersson e de Sedgewick e as árvores AVL são mais simples de implementar que a estrutura `RedBlackTree` aqui definida. Infelizmente nenhuma delas pode garantir que o tempo amortizado gasto rebalanceando é $O(1)$ por atualização. Em especial não há teorema análogo ao Teorema 9.2 para essas estruturas.

Exercício 9.1. Desenhe a árvore 2-4 que corresponde à `RedBlackTree` na

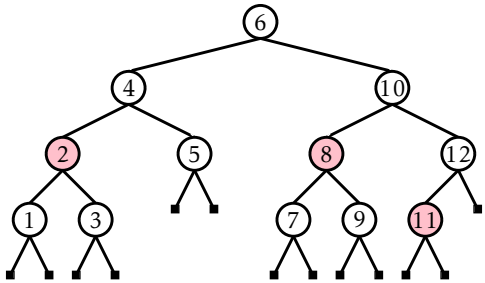


Figura 9.11: Uma árvore rubro-negra para praticar.

Figura 9.11.

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

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

Exercício 9.4. Mostre que, para valores arbitrariamente grandes de n, há árvores rubro-negras com n nodos com altura $2\log n - O(1)$.

Exercício 9.5. Considere as operações `pushBlack(u)` e `pullBlack(u)`. O que essas operações fazem com a árvore 2-4 implícita 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 resultam em árvores rubro-negras com n nodos que têm altura $2\log n - O(1)$.

Exercício 9.7. Porque o método `remove(x)` na implementação da `RedBlackTree` realiza a atribuição `u.parent = w.parent`? Isso não seria feito pela chamada `splice(w)`?

Exercício 9.8. Suponha que a árvore 2-4, T, tem n_ℓ folhas e n_i nodos internos.

1. Qual é o valor mínimo de n_i, como uma função de n_ℓ?

2. Qual é o valor máximo de n_i, como uma função de n_ℓ?

3. Se T' é uma árvore rubro-negra que representa T, então quantos nodos vermelhos T' tem?

Exercício 9.9. Suponha que você tem uma árvore binária de busca com n nodos e uma altura de até $2\log n - 2$. É sempre possível colorir os nodos vermelhos e pretos tal que a árvore satisfaz as propriedades da altura preta e de que não há nenhuma aresta vermelha? Se sim, também é possível fazê-la satisfazer a propriedade de pender à esquerda?

Exercício 9.10. Suponha que você tem duas árvores rubro-negras T_1 e T_2 que têm a mesma altura preta , h, e tal que a maior chave em T_1 é menor que a menor chave em T_2. Mostre como juntar 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 onde as duas árvores T_1 e T_2 têm diferentes alturas pretas, $h_1 \neq h_2$. 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 realizar até uma operação de rebalanceamento (que envolve até duas rotações; veja a Figura 9.10). Dê um exemplo de uma árvore AVL e uma operação remove(x) nessa árvore que requer na ordem de $\log n$ operações de rebalanceamento.

Exercício 9.13. Implemente uma classe AVLTree que implementa árvores AVL conforme descrito acima. Compare seu desempenho à implementação da RedBlackTree. Qual implementação tem a operação find(x) mais rápida?

Exercício 9.14. Projete e implemente uma série de experimentos que comparam o desempenho de find(x), add(x) e remove(x) para as implementações SSet: SkiplistSSet, ScapegoatTree, Treap e RedBlackTree. Inclua diferentes cenários de teste, incluindo casos onde os dados são aleatórios, previamente ordenados, são removidos em ordem aleatória, são removidos em ordem crescente e assim por diante.

Capítulo 10

Heaps

Neste capítulo, nós discutiremos duas implementações da estrutura de dados extremamente útil `Queue` de prioridades. Essas duas estruturas são um tipo especial de árvore binária chamada de *heap*, que significa "uma pilha desorganizada". Isso contrasta com árvores binárias de busca que podem ser pensadas como pilhas altamente organizadas.

A primeira implementação da heap usa um array que simula uma árvore binária completa. Essa implementação muito rápida é a base de um dos algoritmos mais rápido de ordenação, chamado de heapsort (veja a Seção 11.1.3). A segunda implementação é baseada em árvores binárias mais flexíveis. Ela possui a operação `meld`(`h`) que permite que a fila de prioridades absorva os elementos de outra fila de prioridades *h*.

10.1 `BinaryHeap`: Uma Árvore Binária Implícita

Nossa primeira implementação de uma `Queue` (de prioridades) é baseada em uma técnica criada há quatrocentos anos atrás. O *método de Eytzinger* nos permite representar uma árvore binária completa como um array listando os nodos da árvore em ordem de largura (veja a Seção 6.1.2). Dessa forma, a raiz é guardada na posição 0, o filho à esquerda da raiz é guardada na posição 1, o filho à direita da raiz na posição 2, o filho à esquerda do filho à esquerda da raiz é guardado na posição 3 e assim por diante. Veja a Figura 10.1.

Se aplicarmos o método de Eytzinger a alguma árvore grande o su-

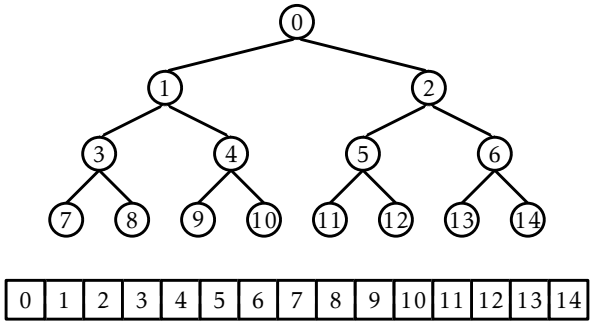


Figura 10.1: O método de Eytzinger representa uma árvore binária completa como um array.

ficiente, alguns padrão emergem. O filho à esquerda do nodo no índice i está no índice $\texttt{left}(i) = 2i + 1$ e o filho à direita do nodo no índice i está no índice $\texttt{right}(i) = 2i + 2$. O pai do nodo no índice i está no índice $\texttt{parent}(i) = (i - 1)/2$.

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;
}
```

Uma BinaryHeap usa essa técnica para implicitamente representar uma árvore binária na qual os elementos estão *ordenados em uma pilha*: O valor guardado em qualquer índice i não é menor que o valor guardado no índice $\texttt{parent}(i)$, com exceção do valor na raiz, $i = 0$. Portanto, o menor valor na Queue com prioridades é guardado na posição 0 (a raiz).

Em uma BinaryHeap, os n elementos são guardados em um array a:

BinaryHeap
```
T[] a;
```

```
int n;
```

Implementar a operação add(x) é razoavelmente simples. Assim como todas as estruturas baseadas em array, primeiro verificamos se a está cheio (verificando se a.length = n) e, caso positivo, expandimos a. A seguir, posicionamos x na posição a[n] e incrementamos n. Nesse ponto, resta garantirmos que mantemos a propriedade heap. Fazemos isso repetidamente trocando x com seu pai, x, até não seja mais menor que seu pai. Veja a Figura 10.2.

BinaryHeap

```
boolean 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) {
    swap(i,p);
    i = p;
    p = parent(i);
  }
}
```

A implementação da operação remove(), que remove o menor valor da heap é um pouco mais complicada. Sabemos onde o menor valor está (na raiz) mas precisamos substituí-lo após sua remoção e garantir a manutenção da propriedade heap.

O modo mais fácil de fazer isso é substituir a raiz com o valor a[n − 1], remover esse valor e decrementar n. Infelizmente, o novo elemento na raiz agora provavelmente não é o menor elemento, então ele precisa ser movido para baixo na árvore. Fazemos isso repetidamente, comparando esse elemento aos seus dois filhos. Se ele for o menor dos três, então paramos. Caso contrário, trocamos esse elemento com o menor de seus dois filhos e continuamos.

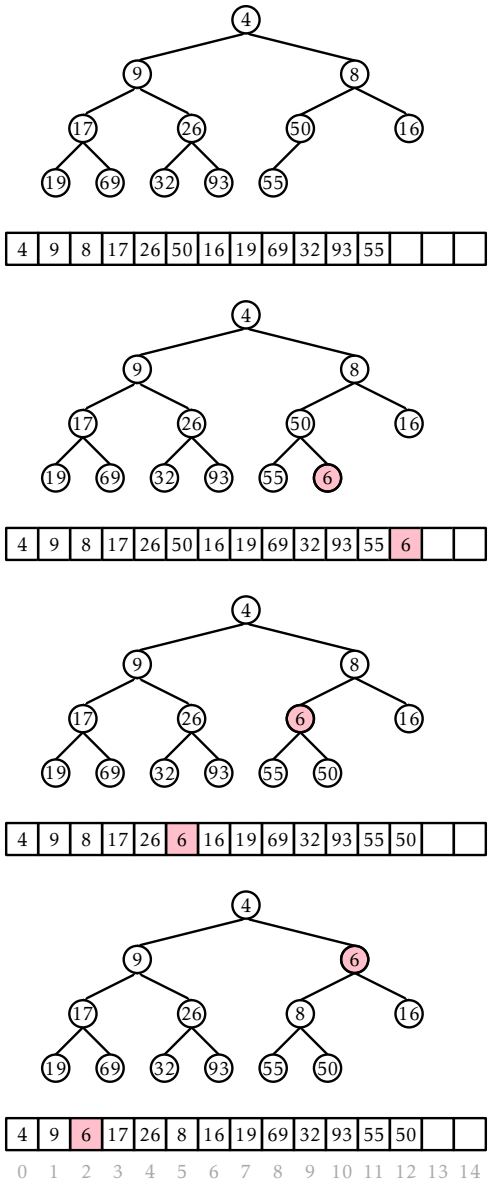


Figura 10.2: Adição do valor 6 a uma BinaryHeap.

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) {
  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)  swap(i, j);
    i = j;
  } while (i >= 0);
}
```

Assim como em outras estruturas baseadas em array, iremos ignorar o tempo gasto em chamadas `resize()`, pois essas podem ser consideradas usando o argumento de amortização do Lema 2.1. Os tempos de execução de `add(x)` e de `remove()` então dependem da altura da árvore binária (implícita). Felizmente, essa árvore binária está *completa* ; todo nível exceto o último tem o maior número possível de nodos. Portanto, se a altura dessa árvore é h, então ele tem pelo menos 2^h nodos. Se outra forma podemos afirmar

$$\mathtt{n} \geq 2^h \ .$$

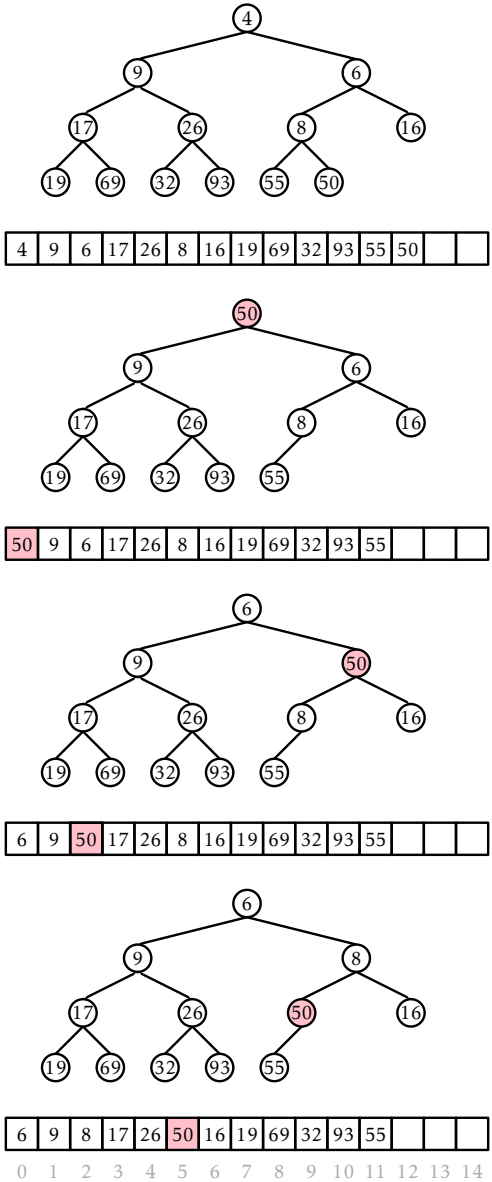


Figura 10.3: Removendo o valor mínimo, 4, de uma BinaryHeap.

Fazendo logaritmos nos dois lados dessa equação resulta em

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

Portanto, as operações add(x) e remove() rodam em $O(\log n)$ de tempo.

10.1.1 Resumo

O teorema a seguir resume o desempenho de uma BinaryHeap:

Teorema 10.1. *Uma BinaryHeap implementa a interface Queue de prioridades. Ignorando o custo de chamadas ao método* resize(), *uma BinaryHeap possui as operações* add(x) *e* remove() *em* $O(\log n)$ *de tempo por operação. Além disso, começando com uma BinaryHeap vazia, uma sequência de* m *operações* add(x) *e* remove() *resulta em um total de* $O(m)$ *de tempo gasto durante todas as chamadas a* resize().

10.2 MeldableHeap: Uma Heap Randomizada Combinável

Nesta seção, descrevemos a MeldableHeap, uma implementação da Queue com prioridades na qual a estrutura básica também é uma árvore binária do tipo heap ordenada. Entretanto, diferentemente de uma BinaryHeap na qual a estrutura básica é completamente definida pelo número de elementos, não há restrições na forma da árvore binária que implementa uma MeldableHeap, uma heap combinável; qualquer forma vale.

As operações add(x) e remove() em uma MeldableHeap são implementadas em termos da operação merge(h1,h2). Essa operação recebe dois nodos de heap h1 e h2 os junta, retornando um nodo de heap que é a raiz de uma heap que contém todos os elementos em uma subárvore enraizada em h1 e todos elementos em uma subárvore enraizada em h2.

O lado bom de uma operação merge(h1,h2) é que ela pode ser definida recursivamente. Veja a Figura 10.4. Se h1 ou h2 forem nil, então estaremos combinando com um conjunto vazio, então retornamos h1 ou h2, respectivamente. Por outro lado, assumimos h1.x $\leq$ h2.x pois, se h1.x $>$ h2.x, então invertemos os papéis de h1 e h2. Então sabemos que a raiz da heap combinada conterá h1.x e podemos recursivamente combinar h2 com h1.left ou h1.right, conforme desejarmos. É nisso que a

randomização entra e lançamos uma moeda para decidir se combinamos h2 com h1.left ou h1.right:

MeldableHeap
```
Node<T> merge(Node<T> h1, Node<T> h2) {
  if (h1 == nil) return h2;
  if (h2 == nil) return h1;
  if (compare(h2.x, h1.x) < 0) return merge(h2, h1);
  // now we know h1.x <= h2.x
  if (rand.nextBoolean()) {
    h1.left = merge(h1.left, h2);
    h1.left.parent = h1;
  } else {
    h1.right = merge(h1.right, h2);
    h1.right.parent = h1;
  }
  return h1;
}
```

Na próxima seção, veremos que merge(h1,h2) roda em tempo esperado $O(\log n)$, onde n é o número total de elementos em h1 e h2.

Com acesso à operação merge(h1,h2), a operação add(x) é fácil. Criamos um novo nodo u contendo x e então combinamos u com a raiz da nossa heap:

MeldableHeap
```
boolean add(T x) {
  Node<T> u = newNode();
  u.x = x;
  r = merge(u, r);
  r.parent = nil;
  n++;
  return true;
}
```

Isso leva $O(\log(n+1)) = O(\log n)$ de tempo esperado.

A operação remove() também é fácil. O nodo que queremos remover é a raiz, então simplesmente combinamos seus seu filhos e fazemos o resultado ser a raiz:

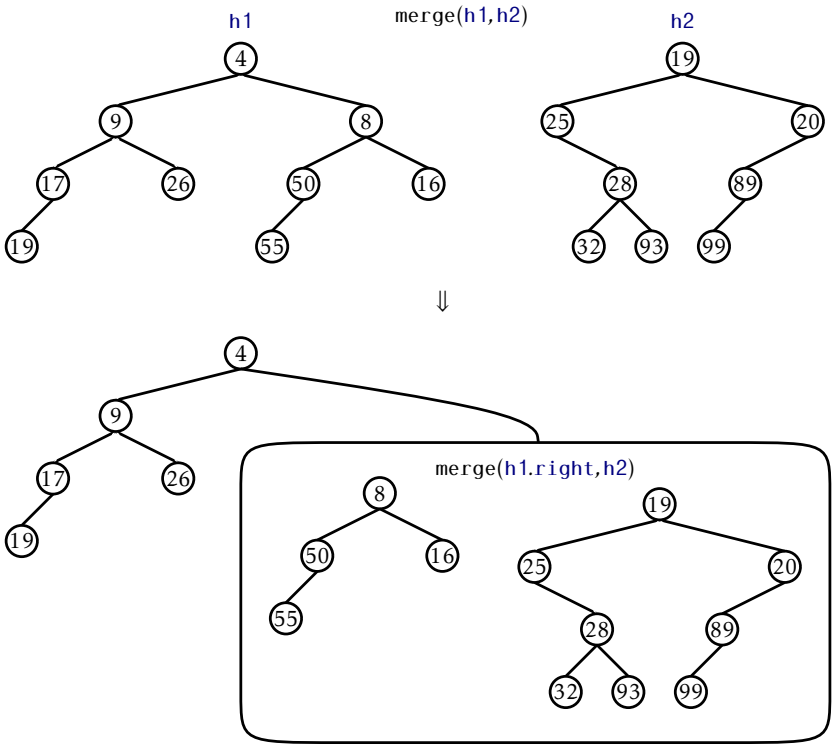


Figura 10.4: A combinação, operação *merge*, de h1 e h2 é feita pela combinação de h2 com a subárvore h1.left ou a subárvore h1.right.

MeldableHeap

```
T remove() {
  T x = r.x;
  r = merge(r.left, r.right);
  if (r != nil) r.parent = nil;
  n--;
  return x;
}
```

Novamente, isso leva $O(\log n)$ de tempo esperado.

Adicionalmente, uma MeldableHeap pode implementar muitas outras operações em $O(\log n)$ de tempo esperado, incluindo:

- remove(u): remove o nodo u (e sua chave u.x) da heap.
- absorb(h): adiciona todos os elementos de uma MeldableHeap h para essa heap, esvaziando h no processo.

Cada uma dessas operações pode ser implementada usando um número constante de operações merge(h1,h2) que leva $O(\log n)$ de tempo esperado.

10.2.1 Análise da Operação merge(h1,h2)

A análise de merge(h1,h2) é baseada na análise da uma caminhada aleatória em uma árvore binária. Uma *caminhada aleatória* em uma árvore binária inicia-se na raiz da árvore. A cada passo na caminhada aleatória, uma moeda é lançada e, dependendo do resultado desse lançamento, a caminhada prossegue ao filho à esquerda ou à direita do nodo atual. Essa caminhada termina quando sai da árvore (o nodo atual torna-se nil).

O lema a seguir é de certa forma impressionante porque não depende da forma da árvore binária:

Lema 10.1. *O comprimento esperado de uma caminhada aleatória em uma árvore binária com* n *nodos é no máximo* $\log(n+1)$.

Demonstração. A prova é por indução em n. No caso base, $n = 0$ e a caminhada tem comprimento $0 = \log(n+1)$. Suponha que o resultado é verdadeiro para todos os inteiros não negativos $n' < n$.

Considere que n_1 denota o tamanho da subárvore à esquerda da raiz tal que $\mathsf{n}_2 = \mathsf{n} - \mathsf{n}_1 - 1$ seja o tamanho da subárvore à direita da raiz. Iniciando na raiz, a caminhada faz um passo e então continua em uma subárvore de tamanho n_1 ou n_2. Pela nossa hipótese indutiva, o comprimento esperado de uma caminhada é então

$$\mathrm{E}[W] = 1 + \frac{1}{2}\log(\mathsf{n}_1 + 1) + \frac{1}{2}\log(\mathsf{n}_2 + 1) \ ,$$

pois n_1 e n_2 são menores que n. Como log é uma função côncava, $\mathrm{E}[W]$ é maximizada quando $\mathsf{n}_1 = \mathsf{n}_2 = (\mathsf{n} - 1)/2$. Portanto, o número esperado de passos feitos pela caminhada aleatória é

$$\begin{aligned}
\mathrm{E}[W] &= 1 + \frac{1}{2}\log(\mathsf{n}_1 + 1) + \frac{1}{2}\log(\mathsf{n}_2 + 1) \\
&\leq 1 + \log((\mathsf{n} - 1)/2 + 1) \\
&= 1 + \log((\mathsf{n} + 1)/2) \\
&= \log(\mathsf{n} + 1) \ .
\end{aligned}$$

□

Fazemos uma rápida digressão para notar que, para leitores que sabem um pouco sobre teoria da informação, a prova do Lema 10.1 pode ser escrita em termos da entropia.

Prova baseada em Teoria da Informação do Lema 10.1. Seja d_i a profundidade do i-ésimo nodo externo e lembre-se que uma árvore binária com n nodos tem $\mathsf{n} + 1$ nodos externos. A probabilidade da caminhada aleatória alcançar o i-ésimo nodo externo é exatamente $p_i = 1/2^{d_i}$, então o comprimento esperado da caminhada aleatória é dado por

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

O lado direito dessa equação é facilmente reconhecível como a entropia da uma distribuição de probabilidade sobre $\mathsf{n} + 1$ elementos. Um fato básico sobre a entropia de uma distribuição sobre $\mathsf{n} + 1$ elementos é que ela não passa de $\log(\mathsf{n} + 1)$, o que prova o lema. □

Com esse resultado sobre caminhadas aleatórias, podemos facilmente provar que o tempo de execução da operação `merge`($\mathsf{h1}, \mathsf{h2}$) é $O(\log \mathsf{n})$.

Lema 10.2. *Sendo* h1 *e* h2 *as raízes de duas heaps contendo* n_1 *e* n_2 *nodos, respectivamente, então o tempo esperado da execução de* merge(h1, h2) *é* $O(\log n)$, *onde* $n = n_1 + n_2$.

Demonstração. Cada passo do algoritmo de combinação merge faz um passo da caminhada aleatória, seja na heap enraizada em h1 ou na heap enraizada em h2. O algoritmo termina quando uma dessas duas caminhadas aleatórias saem da sua própria árvore (quando h1 = null ou h2 = null). Portanto, o número esperado de passos feitos pelo algoritmo merge é no máximo

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

10.2.2 Resumo

O teorema a seguir resume o desempenho de uma MeldableHeap:

Teorema 10.2. *Uma* MeldableHeap *implementa a interface de uma* Queue *com prioridades. Uma* MeldableHeap *possui as operações* add(x) *e* remove() *em* $O(\log n)$ *de tempo esperado por operação.*

10.3 Discussão e Exercícios

A representação implícita de uma árvore binária completa como um array, ou lista, parece ter sido proposta pela primeira vez por Eytzinger [27]. Ele usou essa representação contendo árvores genealógicas de famílias nobres. A estrutura de dados BinaryHeap descrita aqui foi descoberta por Williams [78].

A estrutura de dados randomizada MeldableHeap descrita aqui parece ter sido originalmente proposta por Gambin e Malinowski [34]. Outras implementações de heaps combináveis existem, incluindo heaps esquerdistas (em inglês, *leftist heaps*) [16, 48, Section 5.3.2], heaps binomiais[75], heaps de Fibonacci [30], heaps de pareamento (em inglês, *pairing heaps*) [29], e *skew heaps* [72], embora nenhuma dessas seja tão simples quanto a estrutura MeldableHeap.

Algumas dessas estruturas também aceitam uma operação chamada de decreaseKey(u, y), na qual o valor no nodo u é reduziado a y. (É uma

pré-condição que $y \leq u.x$.) Na maior parte desses estruturas, essa operação pode rodar em $O(\log n)$ de tempo ao remover o nodo u e inserir y. Porém, algumas dessas estruturas podem implementar decreaseKey(u, y) mais eficientemente. Em especial, decreaseKey(u, y) leva $O(1)$ de tempo amortizado em uma heap de Fibonacci e $O(\log \log n)$ de tempo amortizado em uma versão especial das *pairing heaps*[25]. Essa operação decreaseKey(u, y) mais eficiente pode ser aplicada para acelerar vários algoritmos de grafos, incluindo o importante algoritmo de encontrar menores caminhos de Dijkstra [30].

Exercício 10.1. Ilustre a adição dos valores 7 e então 3 à BinaryHeap mostrada no final da Figura 10.2.

Exercício 10.2. Simule a remoção dos próximos dois valores (6 e 8) na BinaryHeap mostrada no final da Figura 10.3.

Exercício 10.3. Implemente o método remove(i) que remove o valor guardado em a[i] em uma BinaryHeap. Esse método devem rodar em tempo $O(\log n)$. A seguir explique porque esse método provavelmente não é útil.

Exercício 10.4. Uma árvore d-ária é uma generalização de uma árvore binária em que cada nodo interno tem d filhos. Usando o método de Eytzinger também é possível representar árvores d-árias usando arrays. Trabalhe as equações de forma que, dado um índice i, determine o pai de i e os seus d filhos nessa representação.

Exercício 10.5. Usando o que você aprendeu no Exercício 10.4, projete e implemente uma *DaryHeap*, a generalização d-ária de uma BinaryHeap. Analise os tempos de execução das operações em uma DaryHeap e teste o desempenho da sua implementação comparando-a com a implementação da BinaryHeap dada neste capítulo.

Exercício 10.6. Ilustre a adição dos valores 17 e então 82 na MeldableHeap h1 mostrada em Figura 10.4. Use uma moeda para simular um bit aleatório quando necessário.

Exercício 10.7. Ilustre a remoção dos valores 4 e 8 na MeldableHeap h1 mostrada na Figura 10.4. Use uma moeda para simular um bit aleatório quando necessário.

Exercício 10.8. Implemente o método `remove(u)`, que remove o nodo `u` de uma `MeldableHeap`. Esse método deve rodar em $O(\log n)$ de tempo esperado.

Exercício 10.9. Mostre como achar o segundo menor valor em uma `BinaryHeap` ou `MeldableHeap` em tempo constante.

Exercício 10.10. Mostre como achar o k-ésimo menor valor em uma `BinaryHeap` ou `MeldableHeap` em tempo $O(k \log k)$. (Dica: usar uma outra heap pode ajudar.)

Exercício 10.11. Suponha que você recebe `k` listas ordenadas, de um comprimento total de `n`. Usando uma heap, mostre como combinar essas listas em uma única lista ordenada em tempo $O(n \log k)$. (Dica: iniciar com o caso $k = 2$ pode ser instrutivo.)

Capítulo 11

Algoritmos de Ordenação

Este capítulo apresenta algoritmos para ordenação de um conjunto de n itens. Esse pode ser um tópico estranho para um livro sobre estruturas de dados, mas existem várias boas razões para incluí-lo aqui. A razão mais óbvia é que dois desses algoritmos de ordenação (quicksort e heapsort) são intimamente relacionados com duas das estruturas de dados que são estudadas neste livro (árvores binárias de busca aleatórias e heaps).

A primeira parte deste capítulo discute algoritmos que ordenam usando somente comparações e apresenta três algoritmos que rodam em $O(n \log n)$ de tempo. Acontece que esses três algoritmos são assintoticamente ótimos; nenhum algoritmo que usa somente comparações pode evitar fazer aproximadamente $n \log n$ comparações no pior caso e mesmo no caso médio.

Antes de continuar, devemos notar que quaisquer implementações do `SSet` ou da `Queue` de prioridades apresentadas nos capítulos anteriores também podem ser usadas para obter um algoritmo de ordenação $O(n \log n)$.

Por exemplo, podemos ordenar n itens fazendo n operações `add(x)` seguidas de n operações `remove()` em uma `BinaryHeap` ou em uma `MeldableHeap`. Alternativamente, podemos usar n operações `add(x)` em qualquer estrutura de dados de árvore de busca binária e então realizar uma travessia em ordem (Exercício 6.8) para extrair os elementos ordenados Porém, nos dois casos teremos muitos gastos para construir uma estrutura que nunca é completamente usada. Ordenação é um problema extremamente importante e vale a pena desenvolver métodos diretos que

são o mais rápidos, simples e econômicos em uso da memória possíveis.

A segunda parte deste capítulo mostra que, se for possível usar outras operações além de comparações, então tudo é possível. Realmente, ao usar indexação por arrays, é possível ordenar um conjunto de n inteiros no intervalo $\{0,\ldots,n^c-1\}$ usando somente $O(cn)$ de tempo.

11.1 Ordenação Baseada em Comparação

Nesta seção, apresentamos três algoritmos: mergesort, quicksort e heapsort. Cada um desses algoritmos recebe um array de entrada a e ordena os elementos de a em ordem não decrescente em $O(n \log n)$ de tempo (esperado). Esses algoritmos são todos *baseados em comparação*. O segundo argumento, c, é um `Comparator` que implementa o método `compare`(a,b) . Esses algoritmos dependem de qual tipo de dados está sendo ordenado; a única operação que eles fazem nos dados é comparações usando o método `compare`(a,b). Lembre-se, da Seção 1.2.4, que `compare`(a,b) retorna um valor negativo se $a < b$, um valor positivo se $a > b$ e zero se $a = b$.

11.1.1 Mergesort

O algoritmo *mergesort* é um clássico exemplo do paradigma de divisão e conquista (recursiva): Se o tamanho de a for até 1, então a está ordenado e não fazemos nada. Caso contrário, dividimos a em duas metades, $a0 = a[0],\ldots,a[n/2-1]$ e $a1 = a[n/2],\ldots,a[n-1]$. Nós recursivamente ordenamos a0 e a1 e então fazermos o merge (em português, a combinação) dos, agora ordenados, a0 e a1 para ordenar o array a completo:

Algorithms

```
<T> void mergeSort(T[] a, Comparator<T> c) {
  if (a.length <= 1) return;
  T[] a0 = Arrays.copyOfRange(a, 0, a.length/2);
  T[] a1 = Arrays.copyOfRange(a, a.length/2, a.length);
  mergeSort(a0, c);
  mergeSort(a1, c);
  merge(a0, a1, a, c);
}
```

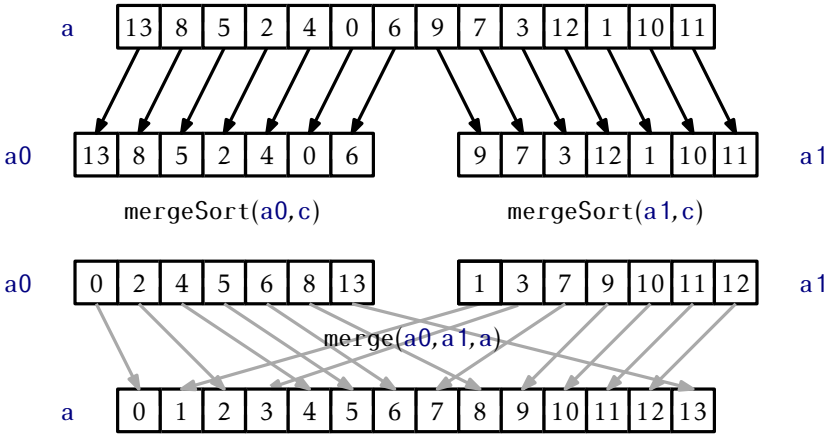


Figura 11.1: A execução de mergeSort(a, c)

Um exemplo é mostrado na Figura 11.1.

Comparada à ordenação, fazer a combinação (*merge*) de dois arrays ordenados a0 e a1 é razoavelmente simples. Adicionamos elementos a a um por vez. Se a0 ou a1 estão vazios, então adicionamos os próximos elementos do outro array com elementos. Caso contrário, pegamos o mínimo entre o próximo elemento em a0 e o próximo elemento em a1 e o adicionamos a a:

Algorithms

```
<T> void merge(T[] a0, T[] a1, T[] a, Comparator<T> c) {
  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++];
  }
}
```

Note que o algoritmo merge(a0, a1, a, c) realiza até $n-1$ comparações antes de acabarem os elementos em a0 ou a1.

Para entender o tempo de execução do mergesort, é mais fácil pensar em termos de sua árvore de recursão. Considere por agora que n seja uma potência de dois, para que $n = 2^{\log n}$ e $\log n$ seja um inteiro. Veja a Figura 11.2. Mergesort torna-se o problema de ordenar n elementos em dois subproblemas, cada qual ordenando $n/2$ elementos. Esse dois subproblemas se tornam em dois subproblemas cada, totalizando quatro subproblemas, cada um de tamanho $n/4$. Esses quatro subproblemas se tornam oito subproblemas, cada um de tamanho $n/8$, assim por diante. Na base desse processo, $n/2$ subproblemas, cada um de tamanho dois, são convertidos em n problemas, cada de tamanho um. Para cada subproblema de tamanho $n/2^i$, o tempo gasto fazendo merge e copiando dados é $O(n/2^i)$. Como há 2^i subproblemas de tamanho $n/2^i$, o tempo total gasto trabalhando em problemas de tamanho 2^i, sem contar as chamadas recursivas é

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

Portanto a tempo total usado pelo mergesort é

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

A prova do teorema a seguir é baseada na análise discutida anteriormente, mas tem que ser um pouco cuidadosa para considerar os casos em que n não sejam uma potência de 2.

Teorema 11.1. *O algoritmo* mergeSort(a, c) *roda em* $O(n \log n)$ *de tempo e faz até* $n \log n$ *comparações.*

Demonstração. Essa prova é por indução em n. O caso base, em que $n = 1$, é trivial; ao ser apresentado com um array de tamanho 0 ou 1, o algoritmo retorna sem fazer nenhuma comparação. Fazer o merge de duas listas ordenadas de tamanho total n requer no máximo $n-1$ comparações. Seja $C(n)$ o número máximo de comparações realizadas por mergeSort(a, c) em um array a de tamanho n. Se n for par, então aplicamos a hipótese

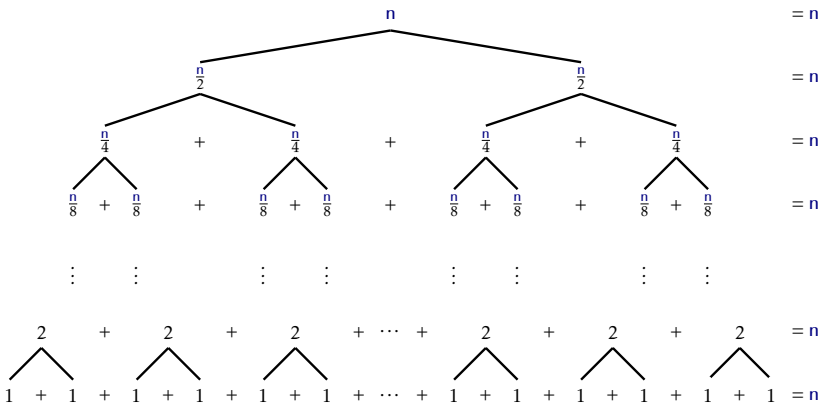


Figura 11.2: A árvore de recursão do mergesort.

indutiva para os dois subproblemas e obtemos

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

O caso em que n é ímpar é ligeiramente mais complicado. Para esse caso, usamos duas desigualdades que são de fácil verificação:

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

para todo $x \ge 1$ e

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

para todo $x \ge 1/2$. A desigualdade (11.1) vem do fato que $\log(x) + 1 = \log(2x)$ enquanto a (11.2) segue do fato que log é uma função côncava.

Com essas ferramentas em mãos, para n ímpar,

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

11.1.2 Quicksort

O algoritmo *quicksort* é outro exemplo clássico do paradigma de divisão e conquista. Diferentemente do mergesort, que faz uma combinação após resolver os dois subproblemas, o quicksort faz esse trabalho logo de uma vez.

Quicksort é simples de descrever: escolher um elemento aleatório chamado de *pivot*, `x`, de `a`; particionar `a` em um conjunto de elementos menores que `x`, um conjunto de elementos igual a `x` e um conjunto de elementos maiores que `x`; e, finalmente, ordenar recursivamente o primeiro e o terceiro conjunto dessa partição. Um exemplo é mostrado na Figura 11.3.

Algorithms

```
<T> void quickSort(T[] a, Comparator<T> c) {
  quickSort(a, 0, a.length, c);
}
<T> void quickSort(T[] a, int i, int n, Comparator<T> c) {
  if (n <= 1) return;
  T x = a[i + rand.nextInt(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
      swap(a, j++, ++p);
    } else if (comp > 0) {
```

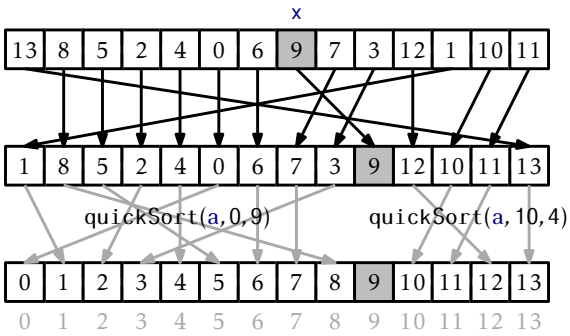


Figura 11.3: Um exemplo de execução de `quickSort(a,0,14,c)`

```
      swap(a, 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, c);
  quickSort(a, q, n-(q-i), c);
}
```

Isso tudo é feito *in place* (ou seja, no próprio espaço de memória do array), então em vez de fazer cópias de subarrays sendo ordenados, o método `quickSort(a,i,n,c)` somente ordena o subarray $a[i],\ldots,a[i+n-1]$. Inicialmente, esse método é chamado com os argumentos que indicam o uso do array completo da seguinte forma `quickSort(a,0,a.length,c)`.

No coração do algoritmo quicksort está o particionamento *in place*. Esse algoritmo, sem usar espaço extra, troca elementos em `a` e computa índices `p` e `q` tal que

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

Esse particionamento, que é feito pelo laço `while` no código, funciona iterativamente aumentando `p` e decrementando `q` enquanto mantém a primeira e última dessas condições. A cada passo, o elemento na posição `j` é

movido para a frente, à esquerda do onde está ou movido para trás. Nos dois primeiros casos, j é incrementado, enquanto no último caso j não é incrementado pois o novo elemento na posição j ainda não foi processado.

O quicksort é profundamente ligado às árvores binárias aleatórias de busca estudadas na Seção 7.1. De fato, se a entrada ao quicksort consistir de n elementos distintos, então a árvore de recursão do quicksort será uma árvore binária de busca aleatória.

Para ver isso, lembre-se que ao construir uma árvore binária de busca aleatória a primeira coisa a fazer é escolher um elemento aleatório x e torná-lo a raiz da árvore. Após isso, todo elemento será eventualmente comparado a x, com elementos menores indo para a subárvore à esquerda e elementos maiores à direita.

No quicksort, escolhemos um elemento aleatório x e imediatamente comparamos todo o array a x, colocando os elementos menores que ele no começo do array e elementos maiores no final do array. O quicksort então recursivamente ordena o começo do array e o final do array, enquanto a árvore binária de busca aleatória recursivamente insere os elementos menores na subárvores à esquerda da raiz e os maiores elementos na subárvore à direita da raiz.

Essa correspondência entre as árvores binárias de busca aleatórias e o quicksort significa que podemos traduzir o Lema 7.1 em uma afirmação sobre o quicksort:

Lema 11.1. *Quando o quicksort é chamado para ordenar um array contendo os inteiros* $0, \ldots, n-1$, *o número esperado de comparações entre o elemento* i *e o pivot é até* $H_{i+1} + H_{n-i}$.

Uma soma aproximada dos números harmônicos nos dá o seguinte teorema sobre o tempo de execução do quicksort:

Teorema 11.2. *Quando o quicksort é chamado para ordenar um array contendo* n *elementos distintos, o número esperado de comparações realizadas é até* $2n \ln n + O(n)$.

Demonstração. Seja T o número de comparações realizadas pelo quicksort para ordenar n elementos distintos. Usando o Lema 11.1 e a linearidade

da esperança, temos:

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

O Teorema 11.3 descreve o caso em que os elementos sendo ordenados são todos distintos. Quando o array de entrada, a, contém elementos duplicados o tempo esperado do quicksort não é pior e pode ser ainda melhor; toda vez que um elemento duplicado x é escolhido como um pivot, todas as ocorrências de x estão agrupadas e não participam de nenhum dos dois subproblemas que se seguem.

Teorema 11.3. *O método* `quickSort(a,c)` *roda em tempo esperado de* $O(n \log n)$ *e o número esperado de comparações que ele realiza é* $2n \ln n + O(n)$.

11.1.3 Heapsort

O algoritmo *heapsort* é outro algoritmo de ordenação que funciona *in place*.

O heapsort usa as heaps binárias discutidas na Seção 10.1. Lembre-se que a estrutura de dados `BinaryHeap` representa uma heap usando um único array. O algoritmo heapsort converte o array de entrada a em uma heap e então repetidamente extrai o valor mínimo.

Mais especificamente, uma heap guarda n elementos em um array, a, em em posições do array $a[0], \ldots, a[n-1]$ com o menor valor guardado na raiz, $a[0]$. Após transformar a em uma `BinaryHeap`, o algoritmo heapsort repetidamente troca $a[0]$ e $a[n-1]$, decrementa n e chama `trickleDown(0)` para que $a[0], \ldots, a[n-2]$ volte a ser uma representação válida de uma heap. Quando esse processo termina (porque $n = 0$) os elementos de a são guardados em ordem decrescente, então a é invertido para obter a

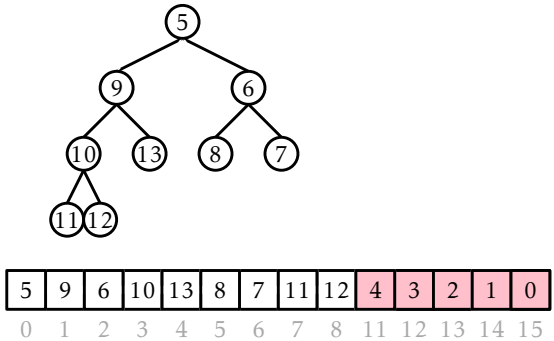


Figura 11.4: Uma etapa da execução do heapSort(a, c). A parte destacada do array já está ordenada. A parte não destacada é uma BinaryHeap. Durante a próxima iteração, o elemento 5 será colocado na posição 8 do array.

forma ordenada final. [1] A Figura 11.4 mostra um exemplo da execução do heapSort(a, c).

BinaryHeap

```
<T> void sort(T[] a, Comparator<T> c) {
  BinaryHeap<T> h = new BinaryHeap<T>(a, c);
  while (h.n > 1) {
    h.swap(--h.n, 0);
    h.trickleDown(0);
  }
  Collections.reverse(Arrays.asList(a));
}
```

Uma sub-rotina chave do heapsort é construtor que transforma um array desordenado a em uma heap. Seria fácil fazer isso usando $O(n \log n)$ de tempo ao chamar repetidamente o método da BinaryHeap add(x), mas podemos fazer melhor ao usar um algoritmo bottom-up, que atua de baixo para cima no processo, ou seja, dos subarrays menores até os maiores.

Lembre-se que, em uma heap binária, os filhos de $a[i]$ são guardados nas posições $a[2i+1]$ e $a[2i+2]$. Isso implica que os elementos nas posições $a[\lfloor n/2 \rfloor], \ldots, a[n-1]$ não têm filhos. Em outras palavras, cada um dos

[1] O algoritmo poderia redefinir a função compare(x, y) para que o heapsort guarde os elementos diretamente na ordem crescente.

elementos em $a[\lfloor n/2 \rfloor], \ldots, a[n-1]$ é uma subheap de tamanho 1. Agora, trabalhando de trás para frente, podemos chamar `trickleDown(i)` para cada $i \in \{\lfloor n/2 \rfloor - 1, \ldots, 0\}$. Isso funciona porque ao chamarmos a operação `trickleDown(i)`, cada um dos dois filhos de $a[i]$ são raízes de uma subheap, então chamar `trickleDown(i)` torna $a[i]$ a raiz de sua própria subheap.

BinaryHeap

```
BinaryHeap(T[] a, Comparator<T> c) {
  this.c = c;
  this.a = a;
  n = a.length;
  for (int i = n/2-1; i >= 0; i--) {
    trickleDown(i);
  }
}
```

Um fato interessante sobre essa estratégia bottom-up é que ela é mais eficiente que chamar `add(x)` n vezes. Para ver isso, note que para $n/2$ elementos, não é necessária nenhuma operação, para $n/4$ elementos, chamamos `trickleDown(i)` em uma subheap com raiz em $a[i]$ e cuja altura é 1, para $n/8$ elementos, chamamos `trickleDown(i)` em uma subheap cuja altura é dois e assim segue. Como o trabalho feito por `trickleDown(i)` é proporcional à altura da subheap enraizada em $a[i]$, isso significa que o trabalho total realizado é até

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

A penúltima igualdade pode ser obtida ao reconhecer que a soma $\sum_{i=1}^{\infty} i/2^i$ é igual, pela definição de valor esperado, ao número esperado de lançamentos de uma moeda, incluindo no primeiro lançamento, cair do lado cara e aplicar o Lema 4.2.

O teorema a seguir descreve o desempenho do `heapSort(a,c)`.

Teorema 11.4. *O método* `heapSort(a,c)` *roda em* $O(n \log n)$ *de tempo e realiza no máximo* $2n \log n + O(n)$ *comparações.*

Demonstração. O algoritmo roda em três passos: (1) transformar a em uma heap, (2) repetidamente extrair o elemento mínimo de a e (3) inverter os elementos em a.

Previamente argumentamos que o passo 1 leva $O(n)$ de tempo e realiza $O(n)$ comparações. Passo 3 leva $O(n)$ de tempo e não faz comparações. Passo 2 faz n chamadas a `trickleDown`(0). A i-ésima chamada opera em uma heap de tamanho $n-i$ e faz até $2\log(n-i)$ comparações. Somando os custos do passo 2 sobre os possíveis valores de i resulta em

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

A soma do número de comparações realizadas em cada um dos três passos completa a prova. □

11.1.4 Um Limitante Inferior para Ordenação

Estudamos até agora três algoritmos de ordenação baseados em comparação em funcionam em $O(n\log n)$ de tempo. Nesse momento, podemos nos perguntar se existem algoritmos mais rápidos. A resposta a essa questão é não. Se as únicas operações permitidas nos elementos de a são comparações, então nenhum algoritmo pode evitar fazer aproximadamente $n\log n$ comparações. Isso não é difícil provar, mas requer um pouco de imaginação. No final das contas, isso vêm do fato que

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

(Fazer a prova desse fato é sugerido como o Exercício 11.11.)

Iniciaremos ao focar nossa atenção em algoritmos determinísticos como mergesort e heapsort e em um valor fixo de n. Imagine que tal algoritmo está sendo usado para ordenar n elementos distintos. A chave para provar o limitante inferior é observar que, para um algoritmo determinístico com valor fixo de n, o primeiro par de elementos que são comparados é sempre o mesmo. Por exemplo, no `heapSort`(a, c), quando n é par, a primeira chamada a `trickleDown`(i) é com $i = n/2 - 1$ e a primeira comparação está entre os elementos $a[n/2-1]$ e $a[n-1]$.

Uma vez que todos os elementos de entrada são distintos, essa primeira comparação tem somente duas possíveis saídas. A segunda com-

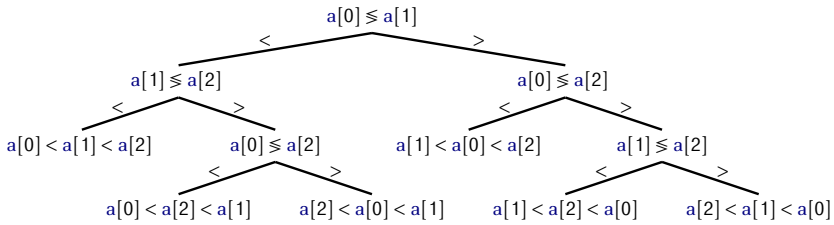


Figura 11.5: Uma árvore de comparações para ordenar um array $a[0], a[1], a[2]$ de tamanho $n = 3$.

paração feita pelo algoritmo depende da saída da primeira comparação. A terceira comparação depende dos resultados das primeiras duas e assim segue. Desse jeito, qualquer algoritmo determinístico de ordenação baseado em comparações pode ser visto como uma *árvore de comparações*. Cada nodo interno, u, dessa árvore é marcado com um par de índices $u.i$ e $u.j$. Se $a[u.i] < a[u.j]$, o algoritmo segue para a subárvore à esquerda. caso contrário ela vai para a subárvore à direita. Cada folha w dessa árvore é marcada com uma permutação $w.p[0], \ldots, w.p[n-1]$ de $0, \ldots, n-1$. Essa permutação representa aquilo que é necessário para ordenar a se a árvore de comparação atingir essa folha. Isso é,

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

Um exemplo de uma árvore de comparações para um array de tamanho $n = 3$ é mostrado na Figura 11.5.

A árvore de comparações para um algoritmo de ordenação nos diz tudo sobre o algoritmo. Ela nos diz exatamente a sequência de comparações que será realizada para qualquer array de entrada, a, tendo n elementos distintos e nos diz como o algoritmo irá permutar a para organizá-lo.

Consequentemente, a árvore de comparações deve ter pelo menos $n!$ folhas; senão, então há duas permutações distintas que levam à mesma folha; portanto, o algoritmo não ordena corretamente pelo uma dessas permutações.

Por exemplo, a árvore de comparações na Figura 11.6 tem somente $4 < 3! = 6$ folhas. Ao inspecionar essa árvore, vemos que os dois arrays de entrada $3, 1, 2$ e $3, 2, 1$ levam à folha mais a direita. Na entrada $3, 1, 2$ essa

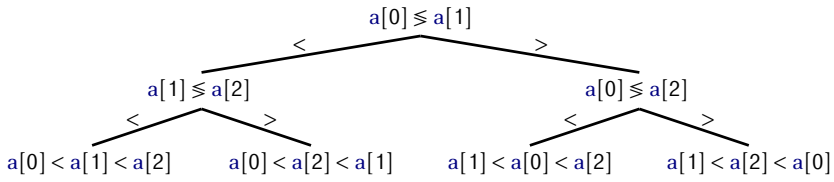


Figura 11.6: Uma árvore de comparações que não ordena corretamente todas as permutações de entrada.

folha corretamente produz $a[1] = 1, a[2] = 2, a[0] = 3$. Entretanto, na entrada $3, 2, 1$, esse nodo incorretamente produz $a[1] = 2, a[2] = 1, a[0] = 3$. Essa discussão nos conduz ao limitante inferior para algoritmos baseados em comparação.

Teorema 11.5. *Para qualquer algoritmo de ordenação determinístico baseado em comparação $\mathcal{A}$ e qualquer inteiro $n \geq 1$, existe um array de entrada a de tamanho n tal que $\mathcal{A}$ usa pelo menos $\log(n!) = n \log n - O(n)$ comparações ao ordenar a.*

Demonstração. De acordo com a discussão anterior, a árvore de comparações definida por $\mathcal{A}$ deve ter pelo menos $n!$ folhas. Uma prova indutiva fácil mostra que qualquer árvore binária com k folhas tem altura de pelo menos $\log k$. Portanto, a árvore de comparações para $\mathcal{A}$ tem uma folha, w, com profundidade de pelo menos $\log(n!)$ e existe um array de entradas a que leva a essa folha. O array de entrada a é uma entrada para qual $\mathcal{A}$ faz pelo menos $\log(n!)$ comparações. □

O Teorema 11.5 trata de algoritmos determinísticos como mergesort e heapsort, mas não nos diz nada sobre algoritmos randomizados como o quicksort. Poderia um algoritmo randomizado usar um número de comparações menor que o limitante inferior $\log(n!)$? A resposta é, novamente, não. De novo, a forma de prová-lo é pensar diferentemente sobre o que um algoritmo randomizado é.

Na discussão a seguir, iremos assumir que nossas árvores de decisões foram "limpadas" da seguinte forma: qualquer nodo que não pode ser alcançado por algum array de entrada a é removido. Essa limpeza implica que a árvore tem exatamente $n!$ folhas. Ela tem pelo menos $n!$ folhas

porque, caso contrário, não ordenaria corretamente. Ela tem até $\mathsf{p}!$ folhas pois cada uma das possíveis permutações $\mathsf{n}!$ de n elementos distintos segue exatamente um caminho da raiz à folha na árvore de decisões.

Podemos imaginar que um algoritmo de ordenação randomizado $\mathcal{R}$ age como um algoritmo determinístico que recebe duas entradas: o array de entradas a que devem ser ordenado e uma longa sequência números reais aleatórios $b = b_1, b_2, b_3, \ldots, b_m$ no intervalo $[0,1]$. Os números aleatórios provêem a randomização ao algoritmo. Quando o algoritmo quer lançar uma moeda ou fazer uma escolha aleatória, ele o faz usando algum elemento de b. Por exemplo, para computar o índice do primeiro pivot no quicksort, o algoritmo poderia usar a fórmula $\lfloor n b_1 \rfloor$.

Agora, note que se fixarmos B a uma sequência particular $\hat{b}$ então $\mathcal{R}$ torna-se um algoritmo de ordenação determinístico, $\mathcal{R}(\hat{b})$, que tem uma árvore de comparações associada, $\mathcal{T}(\hat{b})$. A seguir, note que se selecionarmos a para ser uma permutação aleatória de $\{1, \ldots, \mathsf{n}\}$, então isso é equivalente a selecionar uma folha aleatória w dentre as $\mathsf{n}!$ folhas de $\mathcal{T}(\hat{b})$.

O Exercício 11.13 pede que você prove que, se selecionarmos uma folha aleatória de qualquer árvore binária com k folhas, então a profundidade esperada daquela folha é pelo menos $\log k$. Portanto, o número esperado de comparações realizado pelo algoritmo (determinístico) $\mathcal{R}(\hat{b})$ quando passado um array de entrada contendo uma permutação aleatória de $\{1, \ldots, n\}$ é pelo menos $\log(\mathsf{n}!)$. Finalmente, note que isso é verdadeiro para toda escolha de $\hat{b}$, portanto isso vale mesmo para $\mathcal{R}$. Isso completa a prova do limitante inferior para algoritmos randomizados.

Teorema 11.6. *Para qualquer inteiro $n \geq 1$ e qualquer algoritmo de ordenação baseado em comparações (determinístico ou randomizado) $\mathcal{A}$, o número esperado de comparações feito por $\mathcal{A}$ ao ordenar uma permutação aleatória de $\{1, \ldots, n\}$ é pelo menos* $\log(\mathsf{n}!) = \mathsf{n} \log \mathsf{n} - O(\mathsf{n})$.

11.2 Counting Sort e Radix Sort

Nesta seção estudamos dois algoritmos de ordenação que não são baseados em comparação. Especializados para ordenar valores inteiros baixos, esses algoritmos se esquivam dos limites do Teorema 11.5 ao usar (partes de) elementos em a como índices para um array. Considere um comando

da forma

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

Esse comando executa em tempo constante, mas tem `c.length` diferentes saídas, dependendo do valor de `a[i]`. Isso significa que a execução de um algoritmo que usa tal comando não pode ser modelado como uma árvore binária. É por essa razão que algoritmos desta seção são capazes de ordenar mais rapidamente que algoritmos baseados em comparação.

11.2.1 Counting Sort

Suponha que temos um array de entrada `a` consistindo de `n` inteiros, cada qual no intervalo $0, \ldots, k-1$. O algoritmo *counting sort* ordena `a` usando um array auxiliar `c` de contadores. Ele produz uma versão ordenada de `a` como um array auxiliar `b`.

A ideia por trás do counting sort é simples: para cada $i \in \{0, \ldots, k-1\}$, conte o número de ocorrências de `i` em `a` e guarde essa contagem em `c[i]`. Agora, após a ordenação, a saída vai ser do tipo `c[0]` ocorrências de 0, seguida de `c[1]` ocorrências de 1, seguida de `c[2]` ocorrências de 2,..., seguida de `c[k-1]` ocorrências de `k-1`. O código que faz isso é bem elegante e sua execução é ilustrada na Figura 11.7:

Algorithms

```
int[] countingSort(int[] a, int k) {
  int c[] = new int[k];
  for (int i = 0; i < a.length; i++)
    c[a[i]]++;
  for (int i = 1; i < k; i++)
    c[i] += c[i-1];
  int b[] = new int[a.length];
  for (int i = a.length-1; i >= 0; i--)
    b[--c[a[i]]] = a[i];
  return b;
}
```

O primeiro laço `for` nesse código usa cada contador `c[i]` tal que ele conta o número de ocorrências de `i` em `a`. Ao usar os valores de `a` como índices, esses contadores podem ser computados em $O(n)$ de tempo com

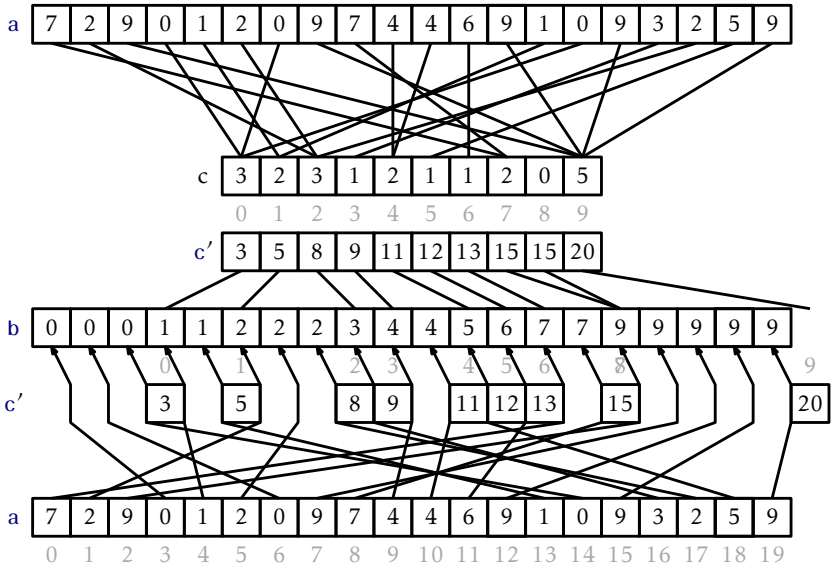


Figura 11.7: A execução do counting sort em um array de tamanho $n = 20$ que guarda inteiros $0, \ldots, k-1 = 9$.

um único `for`. Nesse ponto, poderíamos usar c para preencher o array de saída diretamente. Porém isso não funcionaria se os elementos de a tiverem dados associados. Portanto, gastamos um pouco de esforço extra para copiar os elementos de a em b.

O próximo laço `for`, que leva $O(k)$ de tempo, computa uma soma acumulativa dos contadores tal que $c[i]$ se torna o número de elementos em a que são menores que ou iguais a i. Em particular, para todo $i \in \{0,\ldots,k-1\}$, o array de saída, b, terá

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

Finalmente, o algoritmo percorre a de trás para frente para posicionar seus elementos em ordem no array de saída b. Ao percorrer, o elemento $a[i] = j$ é colocado na posição $b[c[j]-1]$ e o valor $c[j]$ é decrementado.

Teorema 11.7. *O método* `countingSort`(a,k) *ordena um array* a *contendo* n *inteiros no conjunto* $\{0,\ldots,k-1\}$ *em* $O(n+k)$ *de tempo.*

O algoritmo counting sort tem a interessante propriedade de ser *estável*; ele preserva a ordem relativa de elementos iguais. Se dois elementos $a[i]$ e $a[j]$ tem o mesmo valor e $i < j$ então $a[i]$ irá aparecer antes de $a[j]$ em b. Isso será útil na seção a seguir.

11.2.2 Radix sort

Counting sort é muito eficiente para ordenar um array de inteiros quando o tamanho, n, do array não é muito menor que o maior valor $k-1$ que aparece no array. O algoritmo *radix sort* , que agora descreveremos, usa várias passadas do counting sort para permitir o uso seu uso em arrays cujo maior valor é bem maior que o tamanho do array.

O radix sort ordena inteiros de w-bits usando w/d iterações do counting sort para ordenar esses inteiros considerando somente d bits por iteração. [2] Mais precisamente, o radix sort primeiro ordena os inteiros usando seus d bits menos significativos e então seus próximos d bits significativos e assim por diante até, na última iteração, os inteiros estão ordenados pelos seus d bits mais significativos.

[2] Assumimos que d divide w, caso contrário podemos sempre aumentar w para $d\lceil w/d \rceil$.

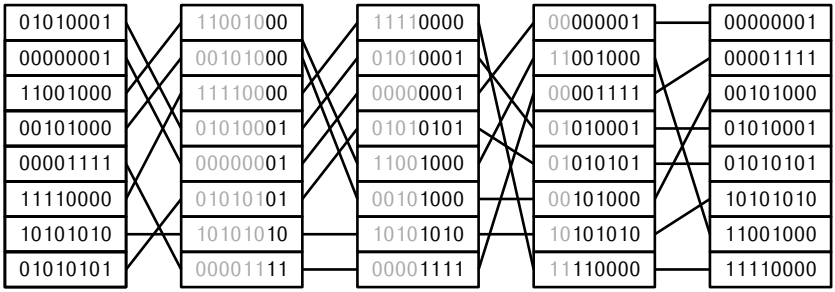


Figura 11.8: Usando radix sort para ordenar inteiros de $w = 8$-bits usando 4 iterações do counting sort em inteiros de $d = 2$-bits.

Algorithms

```
int[] radixSort(int[] a) {
  int[] b = null;
  for (int p = 0; p < w/d; p++) {
    int c[] = new int[1<<d];
    // the next three for loops implement counting-sort
    b = new int[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;
  }
  return b;
}
```

(Nesse código, a expressão $(a[i]\text{»}d*p)\&((1\text{«}d)-1)$ extrai o inteiro cuja representação binária é dada pelos bits $(p+1)d-1,\ldots,pd$ de $a[i]$.) Um exemplo dos passos desse algoritmo é mostrado na Figura 11.8.

Esse algoritmo memorável ordena corretamente porque o counting sort é um algoritmo de ordenação estável. Se $x < y$ são dois elementos de a, e o bit mais significativo em que x difere de y tem índice r, então x será posicionado antes de y durante a iteração $\lfloor r/d \rfloor$ e as iterações seguintes não alterarão a ordem relativa entre x e y.

O radix sort usa $\mathtt{w}/\mathtt{d}$ iterações do algoritmo counting sort. Cada iteração requer $O(\mathtt{n} + 2^{\mathtt{d}})$ de tempo. Portanto, o desempenho do radix sort é dado pelo seguinte teorema.

Teorema 11.8. *Para qualquer inteiro* $\mathtt{d} > 0$, *o método* `radixSort(a,k)` *pode ordenar um array* a *contendo* n *inteiros de* w*-bits em* $O((\mathtt{w}/\mathtt{d})(\mathtt{n}+2^{\mathtt{d}}))$ *de tempo.*

Alternativamente, se os elementos do array estiverem no intervalo $\{0,\ldots,\mathtt{n}^c - 1\}$ e usarmos $\mathtt{d} = \lceil \log \mathtt{n} \rceil$, então obtemos a seguinte versão do Teorema 11.8.

Corolário 11.1. *O método* `radixSort(a,k)` *pode ordenar um array* a *contendo* n *valores inteiros no intervalo* $\{0,\ldots,\mathtt{n}^c - 1\}$ *em* $O(c\mathtt{n})$ *de tempo.*

11.3 Discussão e Exercícios

Ordenação é *o* principal problema fundamental em Ciência da Computação e tem uma longa história. Knuth [48] atribui o algoritmo merge sort a von Neumann (1945). O quicksort foi proposto por Hoare [39]. O algoritmo heapsort original é de Williams [78], mas a versão apresentada aqui (na qual a heap é construída bottom-up em $O(\mathtt{n})$ de tempo) foi proposta por Floyd [28]. Limitantes inferiores para ordenação baseada em comparações parecem ser folclore. A tabela a seguir resume o desempenho desses algoritmos baseados em comparações:

	comparações		in place
mergesort	$\mathtt{n}\log\mathtt{n}$	pior caso	Não
quicksort	$1.38\mathtt{n}\log\mathtt{n} + O(\mathtt{n})$	esperado	Sim
heapsort	$2\mathtt{n}\log\mathtt{n} + O(\mathtt{n})$	pior caso	Sim

Cada um desses algoritmos baseados em comparação tem suas vantagens e desvantagens. O mergesort faz o menor número de comparações e não precisa de randomização. Infelizmente, ele usa um array auxiliar durante sua fase de merge. Alocar esse array pode ser caro e é um ponto de falha se a memória estiver escassa. O quicksort funciona *in place* e é o segundo melhor em termos do número de comparações, mas é randomizado então esse tempo de execução nem sempre é garantido. Heapsort faz mais comparações mas não usa memória extra e é determinístico. Existe

um cenário em que o mergesort é o vencedor: ao ordenar uma lista ligada. Nesse caso, o array auxiliar extra não é necessário; duas listas ligadas ordenadas são facilmente juntadas em uma única lista ligada ordenada usando manipulação de ponteiros (veja o Exercício 11.2).

Os algoritmos counting sort e radix sort aqui descritos foram propostos por Seward [68, Section 2.4.6]. Porém, variantes do radix sort têm sido usadas desde a década de 1920 para ordenar cartões perfurados usando máquinas de ordenação. Essas máquinas podem ordenar uma pilha de cartões em duas pilhas usando a existência (ou não) de um furo em uma posição específica no cartão. Ao repetir esse processo para diferentes posições de furos resulta em uma implementação do radix sort.

Finalmente, notamos que counting sort e radix sort podem ser usados para ordenar outros tipos de números além de inteiros não negativos. Modificações do counting sort podem ordenar inteiros em qualquer intervalo $\{a, \ldots, b\}$, em $O(\mathsf{n} + b - a)$ de tempo. De modo similar, radix sort pode ordenar inteiros no mesmo intervalo em $O(\mathsf{n}(\log_{\mathsf{n}}(b - a))$ de tempo. Finalmente, esses dois algoritmos também podem ser usados para ordenar números em ponto flutuante no formato IEEE 754. Isso porque o formato IEEE 754 é projetado para permitir a comparação de dois números ponto flutuantes ao comparar seus valores como se estivessem em uma representação binária de inteiros com bit de sinal. [2].

Exercício 11.1. Ilustre a execução do mergesort e do heapsort em um array de entrada contendo $1, 7, 4, 6, 2, 8, 3, 5$. Dê um exemplo simples de uma possível execução do quicksort no mesmo array.

Exercício 11.2. Implemente uma versão do algoritmo mergesort que ordena uma `DLList` sem usar um array auxiliar (Veja o Exercício 3.13).

Exercício 11.3. Algumas implementações do `quickSort(a, i, n, c)` usam `a[i]` como um pivot. Dê um exemplo de um array de entrada de tamanho `n` no qual tal implementação faria $\binom{\mathsf{n}}{2}$ comparações.

Exercício 11.4. Algumas implementações de `quickSort(a, i, n, c)` usam `a[i + n/2]` como um pivot. Dê um exemplo de um array de entrada de tamanho `n` no qual tal implementação faria $\binom{\mathsf{n}}{2}$ comparações.

Exercício 11.5. Mostre que, para qualquer implementação de `quickSort(a, i, n, c)` que escolha um pivot deterministicamente, sem primeiro olhar os valores

em $a[i], \ldots, a[i+n-1]$, existe um array de entrada de tamanho n que faz que essa implementação realize $\binom{n}{2}$ comparações.

Exercício 11.6. Projete um `Comparator`, c, que você poderia passar como um argumento ao `quickSort`(a, i, n, c) e que faria o quicksort usar $\binom{n}{2}$ comparações. (Dica: O seu comparador não precisa olhar nos valores sendo comparados.)

Exercício 11.7. Analise o número esperado de comparações feitos pelo quicksort um pouco mais cuidadosamente que a prova do Teorema 11.3. Em particular, mostre que o número esperado de comparações é $2nH_n - n + H_n$.

Exercício 11.8. Descreva um array de entrada que faz o heapsort realizar pelo menos $2n \log n - O(n)$ comparações. Justifique sua resposta.

Exercício 11.9. A implementação do heapsort aqui descrita ordena os elementos em ordem reversa e então inverte o array. Esse último passo poderia ser evitar ao definir um novo `Comparator` que nega os resultados da entrada `Comparator`, c. Explique porque isso não seria seria uma boa otimização. (Dica: considere quantas negações seriam necessárias para fazer em relação a quanto tempo levar a inverter o array.)

Exercício 11.10. Ache outro par de permutações de $1, 2, 3$ que não são corretamente ordenados pela árvore de comparações na Figura 11.6.

Exercício 11.11. Prove que $\log n! = n \log n - O(n)$.

Exercício 11.12. Prove que uma árvore binária com k folhas tem altura de pelo menos $\log k$.

Exercício 11.13. Prove que, se escolhermos uma folha aleatória de uma árvore binária com k folhas, então a altura esperada dessa folha é, pelo menos, $\log k$.

Exercício 11.14. A implementação do `radixSort`(a, k) fornecida aqui funciona quando o array de entrada a contém somente inteiros não negativos . Adapte essa implementação para que também funcione corretamente quando a contém inteiros negativos e não negativos.

Capítulo 12

Grafos

Neste capítulo, estudamos duas representações de grafos e algoritmos básicos que usam essas representações.

Matematicamente, um *grafo (direcionado)* é um par $G = (V, E)$ onde V é um conjunto de *vértices* e E é um conjunto de pares ordenados de vértices chamados de *arestas*. Uma aresta $(\mathtt{i}, \mathtt{j})$ é *direcionada* de $\mathtt{i}$ para $\mathtt{j}$; $\mathtt{i}$ é chamado de *origem* , ou *source* em inglês, da aresta e $\mathtt{j}$ é chamado de destino, ou *target* em inglês. Um caminho, *path*, em G é uma sequência de vértices $v_0, \ldots, v_k$ tal que, para todo $i \in \{1, \ldots, k\}$, a aresta (v_{i-1}, v_i) está em E. Um caminho $v_0, \ldots, v_k$ é um *ciclo* se, adicionalmente, a aresta (v_k, v_0) está em E. Um caminho (ou ciclo) é *simples* se todos o seus vértices são únicos. Se existir um caminho de algum vértice v_i para algum vértice v_j então dizemos que v_j é *alcançável* de v_i. Um exemplo de um grafo é mostrado na Figura 12.1.

Devido à sua habilidade de modelar tantos fenômenos, grafos têm um enorme número de aplicações. Existem muitos exemplos óbvios. Redes de computadores podem ser modeladas como grafos, com vértices correspondentes a computadores e arestas correspondentes a links (direcionados) de comunicação entre computadores. Ruas de cidades podem ser modeladas como grafos, com vértices representando cruzamentos e arestas representando ruas.

Exemplos menos óbvios ocorrem assim que percebemos que grafos podem modelar qualquer relações que ocorrem em pares de elementos de um conjunto. Por exemplo, em uma universidade podemos ter um *grafo de conflitos* de horários cujos vértices representam cursos oferecidos

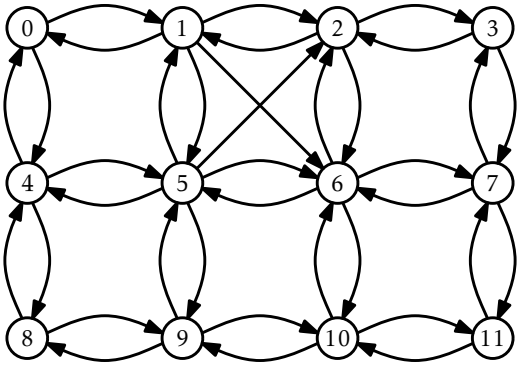


Figura 12.1: Um grafo com doze vértices. Vértices são desenhados como círculos numerados e arestas são desenhadas como curvas com setas apontando da origem ao destino.

em uma universidade no qual a aresta (`i`, `j`) está presente se e somente se existe pelo menos um estudante que está frequentando a disciplina `i` e a disciplina `j`. Então, uma aresta indica que a prova para a disciplina `i` não pode ser agendada no mesmo horário que o exame para a disciplina `j`.

Ao longo dessa seção, iremos usar `n` para denotar o número de vértices de G e `m` para denotar o número de arestas de G. Isto é, $\texttt{n} = |V|$ e $\texttt{m} = |E|$. Além disso, assumiremos que $V = \{0, \ldots, \texttt{n} - 1\}$. Outros dados que gostaríamos de associar com os elementos de V podem ser guardados em um array de tamanho `n`.

Algumas operações típicas realizadas em grafos são:

- `addEdge`(`i`, `j`): Adicionar a aresta (`i`, `j`) a E.
- `removeEdge`(`i`, `j`): Remover a aresta (`i`, `j`) de E.
- `hasEdge`(`i`, `j`): Verificar se a aresta (`i`, `j`) $\in E$
- `outEdges`(`i`): Retornar uma `List` de todos os inteiros `j` tais que (`i`, `j`) $\in E$
- `inEdges`(`i`): Retornar uma `List` de todos os inteiros `j` tais que (`j`, `i`) $\in E$

Note que essas operações não são terrivelmente difíceis de implementar eficientemente. Por exemplo, as três primeiras operações podem ser implementadas diretamente usando um `USet`, de forma que podem ser implementadas em tempo esperado constante usando tabelas hash discutidas no Capítulo 5. As duas últimas operações podem ser implementadas em tempo constante guardando, para cada vértice, uma lista de seu vértices adjacentes.

Entretanto, diferentes aplicações de grafos têm diferentes exigências de desempenho para essas operações e, idealmente, podemos usar a implementação mais simples que satisfaz todos os requisitos da aplicação. Devido a essa razão, nós discutimos duas grandes categorias de representações de grafos.

12.1 `AdjacencyMatrix`: Um Grafo com uma Matriz

Uma *matriz de adjacências* é uma forma de representar um grafo com n vértices $G = (V, E)$ com uma matriz $n \times n$, a, cujas entradas são valores booleanos.

AdjacencyMatrix

```
int n;
boolean[][] a;
AdjacencyMatrix(int n0) {
  n = n0;
  a = new boolean[n][n];
}
```

A valor da matriz $a[i][j]$ é definido como

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

A matriz de adjacências para o grafo na Figura 12.1 é mostrado na Figura 12.2.

Nessa representação, as operações `addEdge(i, j)`, `removeEdge(i, j)` e `hasEdge(i, j)` somente atribuem e lêem a entrada da matriz $a[i][j]$:

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

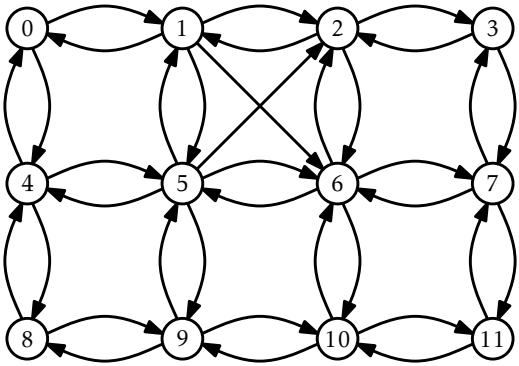
Essas operações claramente levam tempo constante por operação.

A matriz de adjacências tem desempenho insatisfatório com as operações `outEdges(i)` e `inEdges(i)`. Para implementá-las, precisamos varrer todas as `n` entradas na correspondente linha ou coluna de `a` e reunir todos os índices `j` onde `a[i][j]`, respectivamente `a[j][i]`, seja verdadeiro.

AdjacencyMatrix
```
List<Integer> outEdges(int i) {
  List<Integer> edges = new ArrayList<Integer>();
  for (int j = 0; j < n; j++)
    if (a[i][j]) edges.add(j);
  return edges;
}
List<Integer> inEdges(int i) {
  List<Integer> edges = new ArrayList<Integer>();
  for (int j = 0; j < n; j++)
    if (a[j][i]) edges.add(j);
  return edges;
}
```

Essas operações claramente levam $O(n)$ de tempo por operação.

Outra desvantagem da matriz de adjacências é o seu tamanho. Ela guarda uma matriz booleana de tamanho $n \times n$, o que exige o uso de pelo menos n^2 bits de memória. A implementação aqui usa uma matriz de valores `boolean` tal que ela na verdade usa na ordem de n^2 bytes de memória. Uma implementação mais cuidadosa, que empacota w valores booleanos em cada palavra de memória, poderia reduzir esse uso de espaço a $O(n^2/w)$ palavras de memória.



	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	0	0	1	0	0	0	0	0	0	0
1	1	0	1	0	0	1	1	0	0	0	0	0
2	1	0	0	1	0	0	1	0	0	0	0	0
3	0	0	1	0	0	0	0	1	0	0	0	0
4	1	0	0	0	0	1	0	0	1	0	0	0
5	0	1	1	0	1	0	1	0	0	1	0	0
6	0	0	1	0	0	1	0	1	0	0	1	0
7	0	0	0	1	0	0	1	0	0	0	0	1
8	0	0	0	0	1	0	0	0	0	1	0	0
9	0	0	0	0	0	1	0	0	1	0	1	0
10	0	0	0	0	0	0	1	0	0	1	0	1
11	0	0	0	0	0	0	0	1	0	0	1	0

Figura 12.2: Um grafo e sua matriz de adjacências.

Teorema 12.1. *A estrutura de dados* `AdjacencyMatrix` *implementa a interface* `Graph`*. Uma* `AdjacencyMatrix` *possui as operações*

- `addEdge(i, j)`, `removeEdge(i, j)` *e* `hasEdge(i, j)` *em tempo constante por operação; e*
- `inEdges(i)` *e* `outEdges(i)` *em* $O(n)$ *tempo por operações.*

O espaço usado por uma `AdjacencyMatrix` *é* $O(n^2)$.

Apesar de alto uso de memória e desempenho ruim das operações `inEdges(i)` e `outEdges(i)`, uma `AdjacencyMatrix` pode ainda ser útil para algumas aplicações. Em especial, quando o grafo G é *denso*, ele tem quase n^2 arestas, então um uso de memória de n^2 pode ser aceitável.

A estrutura de dados `AdjacencyMatrix` também é frequentemente usada porque operações algébricas na matriz `a` podem ser usadas para computar eficientemente as propriedades do grafo G. Isso é um tópico para uma disciplina sobre algoritmos, mas uma dessas propriedades é: se tratarmos as entradas de `a` como inteiros (1 para `true` e 0 para `false`) e multiplicarmos `a` por si mesma usando multiplicação de matrizes então obtemos a matriz a^2. Lembre-se, da definição de multiplicação de matrizes, que

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

Interpretando essa soma em termos do grafo G, essa fórmula conta o número de vértices, `k`, tal que G contém as arestas (i,k) e (k,j). Isto é, isso conta o número de caminhos de `i` a `j` (por meio de vértices intermediários, `k`) cujo comprimento é exatamente dois. Essa observação é a fundação de um algoritmo que computa os caminhos mais curtos entre todos os pares de vértices em G usando somente $O(\log n)$ multiplicações de matrizes.

12.2 `AdjacencyLists`: Um Grafo como uma Coleção de Listas

Representações de grafos usando *listas de adjacências* é uma abordagem centrada nos vértices. Existem muitas implementações possíveis de listas de adjacências. Nesta seção, apresentamos uma simples. No final da

seção, discutimos diferentes possibilidades. Em uma representação de listas de adjacências, o grafo $G = (V, E)$ é representado como um array adj de listas. A lista adj[i] contém uma lista de vértices adjacentes ao vértice i. Isto é, ela contém todo índice j tal que $(i, j) \in E$.

AdjacencyLists
```
int n;
List<Integer>[] adj;
AdjacencyLists(int n0) {
  n = n0;
  adj = (List<Integer>[])new List[n];
  for (int i = 0; i < n; i++)
    adj[i] = new ArrayStack<Integer>();
}
```

(Uma exemplo é mostrado na Figura 12.3.) Nessa implementação em especial, representamos cada lista em adj como uma ArrayStack, porque gostaríamos acesso pela posição em tempo constante. Especificamente, poderíamos implementar adj como uma DLList.

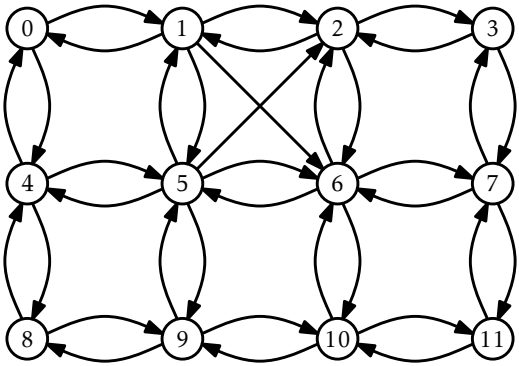
A operação addEdge(i, j) simplesmente acrescenta o valor j no final da lista adj[i]:

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

Isso leva tempo constante.

A operação removeEdge(i, j) busca ao longo da lista adj[i] até que ache j e então o remove:

AdjacencyLists
```
void removeEdge(int i, int j) {
  Iterator<Integer> it = adj[i].iterator();
  while (it.hasNext()) {
    if (it.next() == j) {
      it.remove();
      return;
    }
```



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: Uma grafo e suas listas de adjacência.

```
  }
}
```

Isso leva $O(\deg(\texttt{i}))$ de tempo, onde $\deg(\texttt{i})$ (o *grau* de `i`) conta o número de arestas em E que têm `i` como origem.

A operação `hasEdge(i, j)` é similar: ela busca ao longo da lista `adj[i]` até achar `j` (e retorna true), ou chega no final da lista (e retorna false):

AdjacencyLists

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

Isso também leva $O(\deg(\texttt{i}))$ de tempo.

A operação `outEdges(i)` é bem simples: retorna a lista `adj[i]`:

AdjacencyLists

```
List<Integer> outEdges(int i) {
  return adj[i];
}
```

Isso claramente leva tempo constante.

A operação `inEdges(i)` é muito mais trabalhosa. Ela varre todo vértice j verificando se a aresta $(\texttt{i}, \texttt{j})$ existe e, caso positivo, adiciona `j` a uma lista de saída:

AdjacencyLists

```
List<Integer> inEdges(int i) {
  List<Integer> edges = new ArrayStack<Integer>();
  for (int j = 0; j < n; j++)
    if (adj[j].contains(i))  edges.add(j);
  return edges;
}
```

Essa operação é muito lenta. Ela verifica toda lista de adjacência de todos os vértices, então leva $O(\texttt{n}+\texttt{m})$ de tempo.

O teorema a seguir resume o desempenho das estruturas de dados anteriormente descrita:

Teorema 12.2. *A estrutura de dados* `AdjacencyLists` *implementa a interface* `Graph`*. Uma* `AdjacencyLists` *possui as operações*

- `addEdge(i, j)` *em tempo constante por operação;*
- `removeEdge(i, j)` *e* `hasEdge(i, j)` *em tempo* $O(\deg(\mathtt{i}))$ *por operação;*
- `outEdges(i)` *em tempo constante por operação; e*
- `inEdges(i)` *em* $O(\mathtt{n}+\mathtt{m})$ *de tempo por operação.*

O espaço usado por uma `AdjacencyLists` *é* $O(\mathtt{n}+\mathtt{m})$.

Conforme mencionado anteriormente, existem muitas escolhas que podem ser feitas ao implementar um grafo como listas de adjacências. Algumas escolhas incluem:

- Que tipo de coleção deve ser usada para guardar cada elemento de `adj`? Podem ser usadas uma listas baseadas em arrays, listas ligadas ou mesmo tabelas hash.
- Deve haver uma segunda lista de adjacência, `inadj`, que guarda para cada `i`, a lista de vértices, `j`, tal que $(\mathtt{j},\mathtt{i}) \in E$? Isso pode reduzir enormemente o tempo de execução da operação `inEdges(i)`, mas requer ligeiramente mais trabalho ao adicionar ou remover arestas.
- A entrada para a aresta `(i, j)` na `adj[i]` deve ser ligada por uma referência à entrada correspondente entrada em `inadj[j]`?
- Arestas devem ser objetos de primeira classe com seus próprios dados associados? Dessa maneira, `adj` conteria listas de arestas em vez de listas de vértices (inteiros).

A maior parte dessas questões resultam em um balanço entre complexidade (e espaço) de implementação e características de desempenho da implementação.

12.3 Travessia em Grafos

Nesta seção, apresentamos dois algoritmos para explorar um grafo, iniciando em um de seus vértices `i` e encontrando todos os vértices que são

alcançáveis a partir de i. Esses dois algoritmos são mais adequados para grafos representação usando listas de adjacências. Portanto, ao analisar esses algoritmos iremos supor que a representação é uma AdjacencyLists.

12.3.1 Busca em Largura

O algoritmo de *busca em largura* inicia-se em um vértice i e visita primeiramente os vizinhos de i e, então, os vizinhos dos vizinhos de i e depois os vizinhos dos vizinhos dos vizinhos de i e assim por diante.

Esse algoritmo é uma generalização do algoritmo de travessia em largura para árvores binárias (Seção 6.1.2), e é muito similar a ele; utiliza-se uma queue, q, que inicialmente contém somente i. Então repetidamente extrai um elemento de q e adiciona seus vizinhos a q, dado que esses vizinhos nunca estiveram em q antes. A maior diferença entre o algoritmo de busca em largura para grafos e o de árvores é que o algoritmo para grafos tem que garantir que não vai adicionar o mesmo vértice a q mais de uma vez.

Isso é feito usando um array booleano auxiliar, seen (vistos, em português), que guarda quais vértices foram descobertos até o momento.

Algorithms

```
void bfs(Graph g, int r) {
  boolean[] seen = new boolean[g.nVertices()];
  Queue<Integer> q = new SLList<Integer>();
  q.add(r);
  seen[r] = true;
  while (!q.isEmpty()) {
    int i = q.remove();
    for (Integer j : g.outEdges(i)) {
      if (!seen[j]) {
        q.add(j);
        seen[j] = true;
      }
    }
  }
}
```

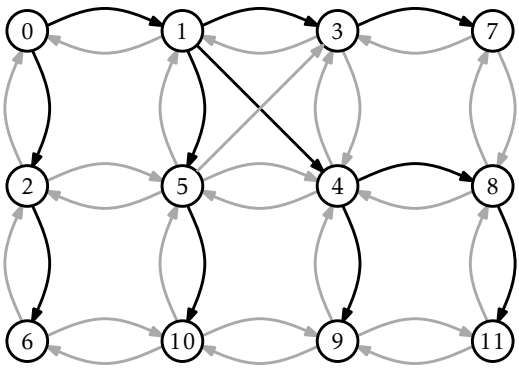


Figura 12.4: Um exemplo de busca em largura iniciando no nodo 0. Nodos são marcados com a ordem em que eles são adicionados a q. Arestas que resultam em nodos sendo adicionados a q são desenhados em preto, outras arestas são desenhadas em cinza.

Um exemplo de execução bfs(g, 0) no grafo da Figura 12.1 é mostrado na Figura 12.4. Execuções diferentes são possíveis, dependendo da ordem das listas de adjacências; a Figura 12.4 usa as listas de adjacências da Figura 12.3.

Analisar o tempo de execução da rotina bfs(g, i) é razoavelmente fácil. O uso do array seen assegura que nenhum vértice é adicionado a q mais de uma vez. Adicionar (e depois remover) cada vértice de q leva tempo constante por vértice, levando a um total de tempo $O(n)$. Como cada vértice é processado pelo laço interno no máximo uma vez, cada lista de adjacência é processada no máximo uma vez, então cada aresta de G é processada somente uma vez. Esse processamento, que é feito pelo laço interno, leva tempo constante por iteração, totalizando em $O(m)$ de tempo. Portanto, o algoritmo por inteiro roda em $O(n+m)$ de tempo.

O teorema a seguir resume o desempenho do algoritmo bfs(g, r).

Teorema 12.3. *O algoritmo* bfs(g, r) *roda, em um* Graph g *implementado usando a estrutura de dados* AdjacencyLists*, em tempo* $O(n+m)$.

Uma travessia em largura tem algumas propriedades especiais. Chamar bfs(g, r) irá eventualmente enfileirar (eventualmente desenfileirar) todo vértice j, tal que existe um caminho direto de r para j. Além disso,

os vértices com distância 0 a partir de r (o próprio r) irá entrar em q antes dos vértices com distância 1, que entrarão em q antes dos vértices com distância 2 e assim por diante. Portanto, o método bfs(g, r) visita vértices em ordem crescente de distância de r e os vértices que não podem ser alcançados a partir de r nunca serão visitados.

Uma aplicação especialmente útil do algoritmo de busca em largura é, portanto, na computação dos menores caminhos. Para computar o menor caminho de r a todo outro vértice, usamos uma variante de bfs(g, r) que usa um array auxiliar p de comprimento n. Quando um novo vértice j é adicionado a q, fazemos p[j] = i. Dessa forma, p[j] se torna o penúltimo nodo no caminho mais curto de r a j. Ao repetir isso fazendo p[p[j]], p[p[p[j]]] e assim por diante podemos reconstruir o caminho mais curto (invertido) de r a j.

12.3.2 Busca em Profundidade

O algoritmo de *busca em profundidade* é similar ao algoritmo padrão para percorrer árvores binárias; ele primeiro explora completamente uma subárvore antes de retornar ao nodo atual e então explora outra subárvore. Outra forma de pensar na busca em profundidade é dizer que é similar à busca em largura exceto que ele usa uma pilha em vez de uma fila.

Durante a execução do algoritmo de busca em profundidade, cada vértice i é atribuído a uma cor c[i]: white se nunca encontramos o vértice antes grey se estamos visitando o vértice e black se terminamos de visitar o vértice. O jeito mais fácil de pensar da busca em profundidade é na forma de um algoritmo recursivo. Ele inicia com uma visita a r. Ao visitar um vértice i, primeiro marcamos i como cinza. Depois, varremos a lista de adjacência de i e recursivamente visitamos qualquer vértice branco que não achamos nessa lista. Finalmente, terminamos o processamento de i e então pintamos i de preto e retornamos.

Algorithms

```
void dfs(Graph g, int r) {
  byte[] c = new byte[g.nVertices()];
  dfs(g, r, c);
}
void dfs(Graph g, int i, byte[] c) {
```

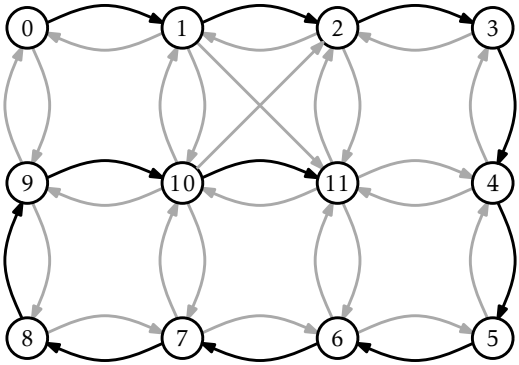


Figura 12.5: Um exemplo da busca em profundidade iniciando no nodo 0. Nodos são marcados com a ordem em que são processados. Arestas que resultam em uma chamada recursiva são desenhados em preto, outras arestas são desenhadas em cinza.

```
  c[i] = grey;  // currently visiting i
  for (Integer j : g.outEdges(i)) {
    if (c[j] == white) {
      c[j] = grey;
      dfs(g, j, c);
    }
  }
  c[i] = black; // done visiting i
}
```

Um exemplo da execução desse algoritmo é mostrado na Figura 12.5.

Embora a busca em profundidade possa ser melhor pensada como um algoritmo recursivo, usar recursão não é a melhor forma de implementá-la. Realmente, o código dado acima irá falhar para grafos grandes causando uma estouro da pilha de memória, um *stack overflow*. Uma implementação alternativa troca a pilha de recursão por uma pilha explícita s. A implementação a seguir faz exatamente isso:

Algorithms

```
void dfs2(Graph g, int r) {
  byte[] c = new byte[g.nVertices()];
  Stack<Integer> s = new Stack<Integer>();
```

```
  s.push(r);
  while (!s.isEmpty()) {
    int i = s.pop();
    if (c[i] == white) {
      c[i] = grey;
      for (int j : g.outEdges(i))
        s.push(j);
    }
  }
}
```

No código anterior, quando o vértice a seguir i é processado, i é pintado de grey e então substituído na pilha com seus vértices adjacentes. Durante a próxima iteração, um desses vértices será visitado.

Pouco surpreendentemente, os tempos de execução de dfs(g, r) e dfs2(g, r) são os mesmos que os de bfs(g, r):

Teorema 12.4. *Ao processar um* Graph *g que é implementado usando a estrutura de dados* AdjacencyLists*, os algoritmos* dfs(g, r) *e* dfs2(g, r) *rodam em tempo* $O(n+m)$.

De modo similar ao algoritmo de busca em largura, existe uma árvore associada com cada execução da busca em profundidade. Quando um nodo $i \neq r$ vai de white a grey, é porque dfs(g, i, c) foi chamado recursivamente ao processar algum nodo i'. (No caso do algoritmo dfs2(g, r), i é um dos nodos que substituem i' na pilha.) Se pensarmos de i' como o pai de i, então obtemos uma árvore enraizada em r. Na Figura 12.5, essa árvore é um caminho do vértice 0 ao vértice 11.

Uma propriedade importante do algoritmo de busca em profundidade é a seguinte: suponha que quando um nodo i é pintado de grey, existe um caminho de i a algum outro nodo j que usa somente vértices brancos. Então j será pintado primeiro de grey e depois de preto antes que i seja pintado de black. (Isso pode ser provado por contradição, ao considerar qualquer caminho P de i para j.)

Uma aplicação dessa propriedade é a detecção de ciclos. Veja a Figura 12.6. Considere algum ciclo C, que pode ser alcançado a partir de r. Seja i o primeiro nodo de C que é pintado de grey e seja j o nodo que precede i no ciclo C. Então, pela propriedade acima, j será pintado de

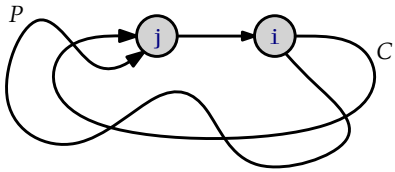


Figura 12.6: O algoritmo de busca em profundidade pode ser usado para detectar ciclos em G. O nodo j está grey enquanto i ainda for grey. Isso implica que existe um caminho P de i a j na árvore de busca em profundidade e a aresta (j, i) implica que P também é um ciclo.

grey e a aresta (j, i) será considerada pelo algoritmo enquanto i ainda for grey. Portanto, o algoritmo pode concluir que existe um caminho P partindo de i a j na árvore de busca em profundidade e que a aresta (j, i) existe. Portanto, P também é um ciclo.

12.4 Discussão e Exercícios

Os tempos de execução dos algoritmos da busca em profundidade e da busca em largura são de certa forma exagerados pelos Teoremas 12.3 e 12.4. Defina n_r como sendo o número de vértices i de G para os quais existem um caminho de r a i. Defina m_r como o número de arestas que têm esses vértices como suas origens. Então o teorema a seguir é mais preciso na descrição dos tempos de execução dos algoritmos de busca em largura e em profundidade. (Esse teorema mais preciso do tempo de execução é útil em algumas das aplicações descritas nos exercícios deste capítulo.)

Teorema 12.5. *Ao processar um* Graph, g, *que é implementado usando a estrutura de dados* AdjacencyLists, *os algoritmos* bfs(g, r), dfs(g, r) *e* dfs2(g, r) *rodam em tempo* $O(n_r + m_r)$.

Busca em largura parece ter sido descoberta independentemente por Moore [52] e Lee [49] nos contextos de exploração de labirintos e roteamento de circuitos, respectivamente.

Representações de grafos baseadas em listas de adjacências foram propostas por Hopcroft e Tarjan [40] como uma alternativa à (então mais

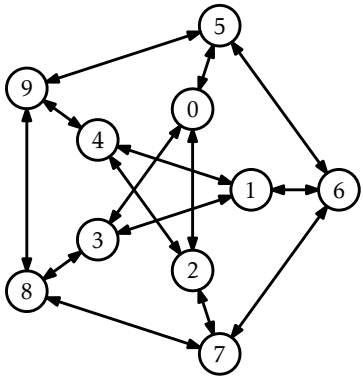


Figura 12.7: Um exemplo de grafo.

comum) representação baseada em matriz de adjacências. Essa representação, assim como busca em profundidade, desempenhou um papel importante no famoso algoritmo de teste de planaridade de Hopcroft-Tarjan que pode determinar, em tempo $O(\mathtt{n})$ se um grafo pode ser desenhado no plano de forma que nenhum par de arestas se cruzem [41].

Nos exercícios a seguir, um grafo não direcionado é um em que, para todo `i` e `j`, a aresta (`i`, `j`) existe se e somente se a aresta (`j`, `i`) existe.

Exercício 12.1. Desenhe uma representação baseada em uma lista de adjacências e uma baseada em matriz de adjacências do grafo na Figura 12.7.

Exercício 12.2. A representação baseada em *matriz de incidência* de um grafo, G, é uma $\mathtt{n} \times \mathtt{m}$ matriz, A, onde

$$A_{i,j} = \begin{cases} -1 & \text{se o vértice } i \text{ for a origem da aresta para } j \\ +1 & \text{se o vértice } i \text{ for o destino da aresta iniciada em } j \\ 0 & \text{caso contrário.} \end{cases}$$

1. Desenhe a representação baseada em matriz de incidência do grafo na Figura 12.7.

2. Projete, analise e implemente uma representação baseada em matriz de incidência de um grafo. Analise o custo de espaço e o custo de tempo das operações `addEdge(i, j)`, `removeEdge(i, j)`, `hasEdge(i, j)`, `inEdges(i)` e `outEdges(i)`.

Exercício 12.3. Desenhe uma execução do algoritmos `bfs(G,0)` e `dfs(G,0)` no grafo `G` da Figura 12.7.

Exercício 12.4. Seja um grafo não direcionado G. Dizemos que G é *conectado* se, para todo par de vértices `i` e `j` em G, existe um caminho de `i` a `j` (como G não é direcionado, também existe um caminho de `j` a `i`). Mostre como testar se G é conectado em tempo $O(\mathtt{n}+\mathtt{m})$.

Exercício 12.5. Seja G um grafo não direcionado. *Marcações de componentes conectados* (em inglês, *connected-component labelling*) de G particiona os vértices de G em conjuntos maximais, cada qual forma um subgrafo conectado. Mostre como computar as marcações de componentes conectados de G em $O(\mathtt{n}+\mathtt{m})$ de tempo.

Exercício 12.6. Seja G um grafo não direcionado. Uma *floresta geradora* de G é uma coleção de árvores, uma por componente, cujas arestas são arestas de G e cujos vértices contêm todos os vértices de G. Mostre como computar uma floresta geradora de G em $O(\mathtt{n}+\mathtt{m})$ de tempo.

Exercício 12.7. Dizemos que um grafo G é *fortemente conectado* se, para todo par de vértices `i` e `j` em G, existir um caminho de `i` para `j`. Mostre como testar se G é fortemente conectado em $O(\mathtt{n}+\mathtt{m})$ de tempo.

Exercício 12.8. Dado um grafo $G=(V,E)$ e algum vértice especial $\mathtt{r}\in V$, mostre como computar o comprimento do caminho mais curto a partir de `r` para `i` para todo vértice $\mathtt{i}\in V$.

Exercício 12.9. Ache um exemplo (simples) em que o código `dfs(g,r)` visita os nodos de um grafo em uma ordem que é diferente daquela do código do algoritmo `dfs2(g,r)`. Escreva uma versão de `dfs2(g,r)` que sempre visita nodos exatamente na mesma ordem que `dfs(g,r)`. (Dica: simplesmente inicie verificando a execução de cada algoritmo em algum grafo onde `r` é a fonte de mais de 1 aresta.)

Exercício 12.10. Uma *universal sink* em um grafo G é um vértice que é o alvo de $\mathtt{n}-1$ arestas e a origem de nenhuma aresta. [1] Projete e implemente um algoritmo que testa se um grafo G, representado por uma

[1] Uma universal sink, `v`, é também às vezes chamada de *celebridade*: Todos na sala reconhecem `v`, mas `v` não reconhece ninguém na sala.

`AdjacencyMatrix`, tem uma *universal sink*. O seu algoritmo deve rodar em $O(n)$ de tempo.

Capítulo 13

Estruturas de Dados para Inteiros

Neste capítulo, retornaremos ao problema de implementar uma SSet. A diferença é que agora iremos assumir que os elementos guardados no SSet são inteiros de w bits. Isto é, queremos implementar add(x), remove(x), e find(x) onde $x \in \{0, \ldots, 2^w - 1\}$. Não é muito difícil de pensar em muitas aplicações em que os dados — ou pelo menos a chave que usamos para organizar os dados — é um inteiro.

Iremos discutir três estruturas de dados, cada uma estendendo as ideias da anterior. A primeira estrutura, a BinaryTrie cobre as três operações da SSet em tempo $O(w)$. Isso não é muito impressionante, pois qualquer subconjunto de $\{0, \ldots, 2^w - 1\}$ tem tamanho $n \le 2^w$, tal que $\log n \le w$. Outras implementações de SSet discutidas neste livro têm operações que rodam em $O(\log n)$ de tempo sendo tão rápidas quanto uma BinaryTrie.

A segunda estrutura, a XFastTrie, acelera a busca em uma BinaryTrie usando hashing. Com essa aceleração, a operação find(x) roda em tempo $O(\log w)$. Entretanto, as operações add(x) e remove(x) em uma XFastTrie ainda leva $O(w)$ de tempo e o espaço usado por uma XFastTrie é $O(n \cdot w)$.

A terceira estrutura de dados, a YFastTrie, usa uma XFastTrie para guardar somente uma amostra de aproximadamente um a cada w elementos e guarda os elementos restantes em uma estrutura padrão SSet. Esse truque reduz o tempo de execução de add(x) e remove(x) a $O(\log w)$ e reduz o espaço a $O(n)$.

As implementações usadas como exemplos neste capítulo podem guardar qualquer tipo de dado, desde que um inteiro possa ser associado a

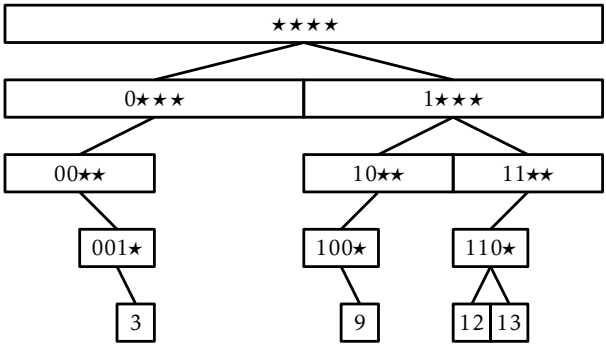


Figura 13.1: Os inteiros guardados em uma trie binária são codificados como caminhos da raiz a uma folha.

ele. Nos trechos de código, a variável `ix` é sempre o valor inteiro associado com `x` e o método `in.intValue(x)` converte `x` ao seu inteiro associado. Porém, neste texto trataremos `x` como se fosse um inteiro.

13.1 `BinaryTrie`: Uma Árvore de Busca Digital

Uma `BinaryTrie` codifica um conjunto de inteiros com `w` bits em uma árvore binária. Todas as folhas em uma árvore têm profundidade `w` e cada inteiro é codificado como um caminho da raiz para uma folha. O caminho para o inteiro `x` vai para a esquerda no nível `i` se o `i`-ésimo bit mais significativo de `x` for um 0 e vai para a direita se for um 1. A Figura 13.1 mostra um exemplo para o caso $w = 4$, no qual a trie guarda inteiros 3(0011), 9(1001), 12(1100) e 13(1101).

Como o caminho de busca por um valor `x` depende nos bits de `x`, é útil nomear os filhos de um nodo `u`, `u.child[0]` (`left`) e `u.child[1]` (`right`). Esses ponteiros dos filhos vão ter dupla utilidade. Como as folhas em uma trie binária não tem filhos, os ponteiros são usados para conectar as folhas em uma lista duplamente ligada. Para uma folha em uma trie binária, `u.child[0]` (`prev`) é o nodo que antecede `u` na lista e `u.child[1]` (`next`) é o nodo que sucede `u` na lista. Um nodo especial, `dummy`, é usado

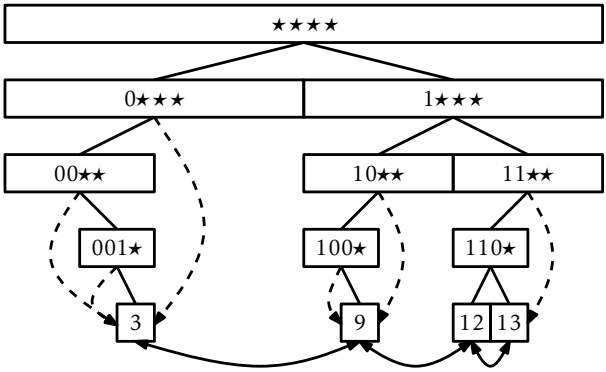


Figura 13.2: Uma `BinaryTrie` com ponteiros `jump` mostrados como arestas curvas tracejadas.

tanto antes do primeiro nodo quanto depois do último nodo na lista (veja a Seção 3.2).

Cada nodo, `u`, também possui um ponteiro adicional `u.jump`. Se o filho à esquerda da `u` não existe, então `u.jump` aponta para a menor folha na subárvore de `u`. Se o filho à direita de `u` não existe então `u.jump` aponta para a maior folha na subárvore de `u`. Um exemplo de uma `BinaryTrie`, mostrando ponteiros `jump` e a lista duplamente ligada na folhas é mostrado na Figura 13.2.

A operação `find(x)` em uma `BinaryTrie` é razoavelmente direta. Tentamos seguir o caminho de busca para `x` na trie. Se alcançarmos uma folha, então achamos `x`. Se alcançarmos um nodo `u` de onde não podemos prosseguir (porque `u` está sem um filho), então seguimos `u.jump`, que nos leva à menor folha maior que `x` ou à maior folha menor que `x`. Qual desses dois casos ocorrerá depende em se `u` não tem seu filho à esquerda ou direita, respectivamente. No caso em que `u` não tem seu filho à esquerda, achamos o nodo que queríamos. No caso em que `u` não tem seu filho à direita, podemos usar a lista ligada para alcançar o nodo que queremos. Cada um desses casos é ilustrado na Figura 13.3.

BinaryTrie

```
T find(T x) {
  int i, c = 0, ix = it.intValue(x);
```

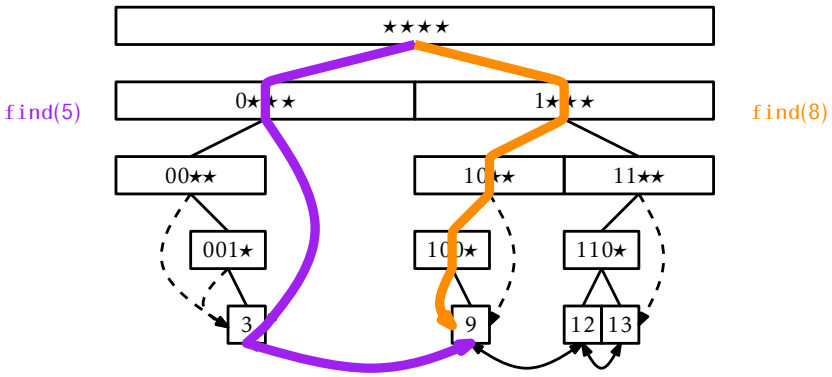


Figura 13.3: Os caminhos usados por find(5) e find(8).

```
  Node u = r;
  for (i = 0; i < w; i++) {
    c = (ix >>> w-i-1) & 1;
    if (u.child[c] == null) break;
    u = u.child[c];
  }
  if (i == w) return u.x;  // found it
  u = (c == 0) ? u.jump : u.jump.child[next];
  return u == dummy ? null : u.x;
}
```

O tempo de execução do método find(x) é dominado pelo tempo que ele leva para seguir um caminho da raiz para a folha, portanto ele roda em tempo $O(w)$.

A operação add(x) em uma BinaryTrie também é relativamente simples mas tem muita coisa a fazer:

1. Ela segue o caminho de busca por x até alcançar um nodo u onde não poderá mais prosseguir.

2. Ela cria o restante do caminho de busca de u a uma folha que contém x.

3. Ela adiciona o nodo u′ contendo x para a lista ligada de folhas (ela tem acesso ao predecessor pred, de u′ na lista ligada do ponteiro

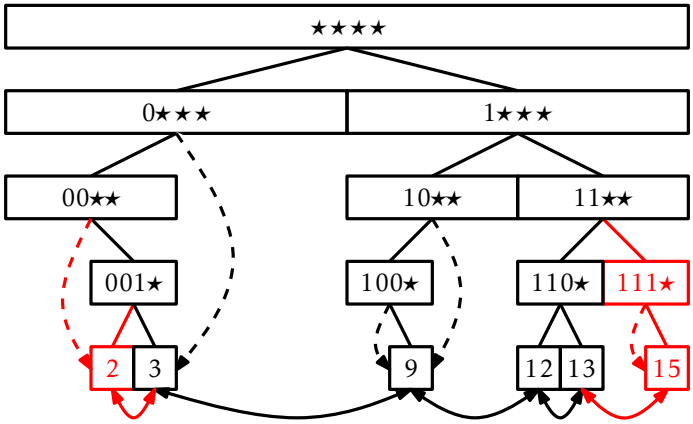


Figura 13.4: Adicionando os valores 2 e 15 na BinaryTrie da Figura 13.2.

jump do último nodo u encontrado durante o passo 1.)

4. Ela retrocede no caminho de busca por x ajustando os ponteiros jump nos nodos cujos ponteiros jump devem agora apontar para x.

Uma adição é ilustrada na Figura 13.4.

BinaryTrie

```
boolean add(T x) {
  int i, c = 0, ix = it.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.child[0];
  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] = newNode();
    u.child[c].parent = u;
```

```
    u = u.child[c];
  }
  u.x = x;
  // 3 - add u to linked list
  u.child[prev] = pred;
  u.child[next] = pred.child[next];
  u.child[prev].child[next] = u;
  u.child[next].child[prev] = u;
  // 4 - walk back up, updating jump pointers
  Node v = u.parent;
  while (v != null) {
    if ((v.child[left] == null
        && (v.jump == null || it.intValue(v.jump.x) > ix))
    || (v.child[right] == null
        && (v.jump == null || it.intValue(v.jump.x) < ix)))
      v.jump = u;
    v = v.parent;
  }
  n++;
  return true;
}
```

Esse método realiza uma descida pelo caminho de busca por `x` e uma subida em seguida. Cada passo dessas caminhadas leva tempo constante, então o método roda em tempo $O(w)$.

A operação `remove(x)` desfaz o trabalho de `add(x)`. Assim como `add(x)`, ela tem muito a fazer:

1. Ela segue o caminho de busca por `x` até alcançar a folha `u` contendo `x`.

2. Ela remove `u` da lista duplamente ligada.

3. Ela remove `u` e então retrocede no caminho de busca de `x` removendo nodos até alcançar um nodo `v` que tem um filho que não está no caminho de busca de `x`.

4. Ela sobe de `v` à raiz atualizando os ponteiros `jump` que apontam para `u`.

Uma remoção é ilustrada na Figura 13.5.

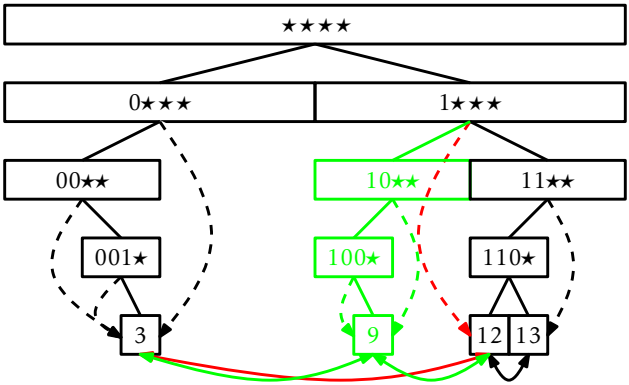


Figura 13.5: Remoção do valor 9 da BinaryTrie na Figura 13.2.

BinaryTrie

```
boolean remove(T x) {
  // 1 - find leaf, u, containing x
  int i, c, ix = it.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.child[prev].child[next] = u.child[next];
  u.child[next].child[prev] = u.child[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;
    v.child[c] = null;
    if (v.child[1-c] != null) break;
  }
  // 4 - update jump pointers
  c = (ix >>> w-i-1) & 1;
  v.jump = u.child[1-c];
  v = v.parent;
```

```
  i--;
  for (; i >= 0; i--) {
    c = (ix >>> w-i-1) & 1;
    if (v.jump == u)
      v.jump = u.child[1-c];
    v = v.parent;
  }
  n--;
  return true;
}
```

Teorema 13.1. *Uma* `BinaryTrie` *implementa a interface* `SSet` *para inteiros de* `w` *bits. Uma* `BinaryTrie` *possui as operações* `add(x)`, `remove(x)` *e* `find(x)` *em tempo de* $O(w)$ *por operação. O espaço usado por uma* `BinaryTrie` *que guarda* `n` *valores em* $O(n \cdot w)$.

13.2 XFastTrie: Buscando em Tempo Duplamente Logarítmico

O desempenho da estrutura `BinaryTrie` não é muito impressionante. O número de elementos `n` guardados na estrutura é no máximo 2^w, então $\log n \le w$. Em outras palavras, qualquer estrutura `SSet` baseada em comparações descrita em outras partes deste livro são pelo menos tão eficientes quanto uma `BinaryTrie` e não se restringem a somente guardar inteiros.

A seguir descrevemos a `XFastTrie`, que é somente uma `BinaryTrie` com $w + 1$ tabelas hash — uma para cada nível da trie. Essas tabelas hash são usadas para acelerar a operação `find(x)` para $O(\log w)$. Lembre-se que a operação `find(x)` em uma `BinaryTrie` está quase terminada ao alcançarmos um nodo `u` onde o caminho de busca para `x` tenta prosseguir para `u.right` (ou `u.left`) mas `u` não tem filho à direita (respectivamente, esquerda). Neste ponto, a busca usa `u.jump` para pular para uma folha `v` da `BinaryTrie` e retorna `v` ou seu sucessor na lista ligada de folhas. Uma `XFastTrie` acelera o processo de busca usando a busca binária nos níveis da trie para achar o nodo `u`.

Para usar busca binária, precisamos de um modo para determinar se o

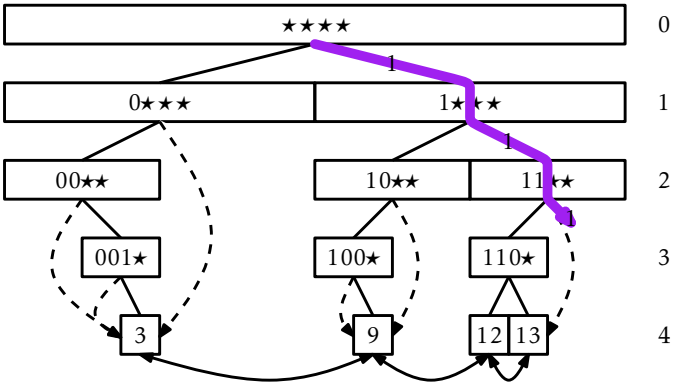


Figura 13.6: Como não há nenhum nodo marcado com 111★, o caminho de busca por 14 (1110) termina no nodo marcado com 11★★ .

nodo u que estamos buscando está acima de dado nível i ou se u está nesse nível ou abaixo. Essa informação é dada pelos i bits de maior ordem na representação binária de x; esses bits determinam o caminho de busca que x percorre da raiz ao nível i. Para um exemplo, veja a Figura 13.6; nessa figura o último nodo u, no caminho de busca por 14 (cuja representação binária é 1110) é o nodo marcado com 11★★ no nível 2 porque não há nodos marcados com 111★ no nível 3. Portanto, podemos marcar cada nodo no nível i com um inteiro com i bits. Então, o nodo u que estamos buscando seria no nível i ou abaixo se, e somente se, existir um nodo no nível i cuja marcação corresponde aos i bits de mais alta ordem de x.

Em uma XFastTrie, guardamos para cada $i \in \{0,\ldots,w\}$, todos os nodos no nível i em um USet t[i] que é implementado como uma tabela hash (Capítulo 5). Usar esse USet nos permite verificar em tempo constante se existe um nodo no nível i cuja marcação corresponde aos i bits de ordem mais alta de x. Podemos encontrar esse nodo usando t[i].find(x»>(w − i))

As tabelas hash $t[0],\ldots,t[w]$ nos permitem usar busca binária para achar u. Inicialmente, sabemos que u está em algum nível i com $0 \le i < w+1$. Portanto inicializamos $l = 0$ e $h = w+1$ e repetidamente olhamos na tabela hash t[i], onde $i = \lfloor (l+h)/2 \rfloor$. Se t[i] contém um nodo cuja marcação corresponde aos i bits de maior ordem de x então fazemos a atribuição $l = i$ (u está no mesmo nível ou abaixo de i); caso contrário

fazemos $h = i$ (u está acima do nível i). Esse processo termina quando $h - l \leq 1$ e neste caso determinamos que u está no nível l. Então completamos a operação find(x) usando u.jump e a lista duplamente ligada de folhas.

XFastTrie

```
T find(T x) {
  int l = 0, h = w+1, ix = it.intValue(x);
  Node v, u = r, q = newNode();
  while (h-l > 1) {
    int i = (l+h)/2;
    q.prefix = ix >>> w-i;
    if ((v = t[i].find(q)) == 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.child[0];
  return (pred.child[next] == dummy)
               ? null : pred.child[next].x;
}
```

Cada iteração do laço while no método anteriormente descrito reduz $h - l$ por aproximadamente um fator de dois, então esse laço encontra u após $O(\log w)$ iterações. Cada iteração realiza uma quantidade constante de trabalho e uma operação find(x) em uma USet, que leva uma quantidade de tempo esperado constante. O restante da operação leva somente tempo constante, então o método find(x) em uma XFastTrie usa somente $O(\log w)$ de tempo esperado.

Os métodos add(x) e remove(x) para uma XFastTrie são quase idênticos aos mesmos mesmos em uma BinaryTrie. As únicas modificações são para gerenciar as tabelas hash $t[0],\ldots,t[w]$. Durante a operação add(x), quando um novo nodo é criado no nível i, esse nodo é adicionado a $t[i]$. Durante a operação remove(x), quando um nodo é removido do nível i, esse nodo é removido de $t[i]$. Como a adição e a remoção de uma ta-

bela hash leva tempo esperado constante, isso não aumenta os tempos de execução de add(x) e remove(x) em mais do que um fator constante. Omitimos os códigos para add(x) e remove(x) pois são quase idênticos ao código longo previamente fornecido para os mesmos métodos em uma BinaryTrie.

O teorema a seguir resume o desempenho de uma XFastTrie:

Teorema 13.2. *Uma XFastTrie implementa a interface SSet para inteiros com w bits. Uma XFastTrie possui as operações*

- add(x) *e* remove(x) *em tempo esperado* $O(w)$ *por operation e*
- find(x) *em tempo esperado* $O(\log w)$ *por operação.*

O espaço usado por uma XFastTrie que guarda n valores é $O(n \cdot w)$.

13.3 YFastTrie: Um SSet com Tempo Duplamente Logarítmico

A XFastTrie é uma melhoria exponencial sobre a BinaryTrie em termos de tempo de consulta, mas as operações add(x) e remove(x) ainda não são muito rápidas. Além disso, o uso de espaço $O(n \cdot w)$ é maior que em outras implementações de SSet descritas neste livro, que usam $O(n)$ de espaço. Esses dois problemas são relacionados; se n operações add(x) construírem uma estrutura de tamanho $n \cdot w$, então a operação add(x) exige pelo menos na ordem de w de tempo (e espaço) por operação.

A YFastTrie, discutida a seguir, simultaneamente melhora os custos de espaço e tempo da XFastTrie. Uma YFastTrie usa uma XFastTrie, xft, mas guarda somente $O(n/w)$ valores em xft. Dessa maneira, o espaço total usado por xft é somente $O(n)$. Adicionalmente, somente uma a cada w operações add(x) ou remove(x) da YFastTrie resulta em uma operação add(x) ou remove(x) em xft. Fazendo isso, o custo médio das chamadas às operações add(x) e remove(x) da xft é constante.

A pergunta óbvia é: se xft somente guarda n/w elementos, para onde o resto dos $n(1 - 1/w)$ elementos vão? Esses elementos são movidos para *estruturas secundárias*, nesse caso uma versão estendida de uma treap (Seção 7.2). Existem aproximadamente n/w dessas estruturas secundárias

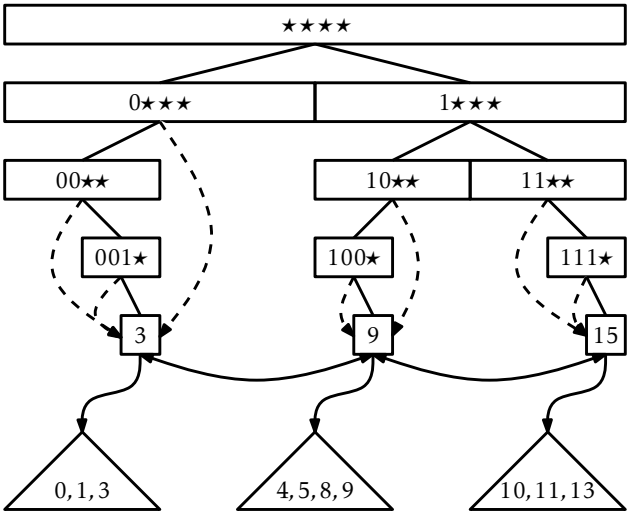


Figura 13.7: Uma YFastTrie contendo os valores 0, 1, 3, 4, 6, 8, 9, 10, 11 e 13.

então, em média, cada uma delas guarda $O(w)$ itens. Treaps aceitam operações SSet em tempo logarítmico, então as operações nessas treaps irão rodar em tempo $O(\log w)$, conforme exigido.

Mais concretamente, uma YFastTrie contém uma XFastTrie, xft, que contém uma amostra aleatória dos dados, onde cada elemento aparece na amostra independentemente com probabilidade $1/w$. Por conveniência, o valor $2^w - 1$ estará sempre contido em xft. Considere que $x_0 < x_1 < \cdots < x_{k-1}$ representam os elementos guardados em xft. Uma treap t_i é associada a cada elemento x_i que guarda todos os valores no intervalo $x_{i-1} + 1, \ldots, x_i$. Isso é ilustrado na Figura 13.7.

A operação find(x) em uma YFastTrie é razoavelmente simples. Buscamos por x em xft e achamos algum valor x_i associado com a treap t_i. Então usamos o método find(x) da treap t_i para responder à consulta. O método completo se resume a uma linha:

YFastTrie
```
T find(T x) {
  return xft.find(new Pair<T>(it.intValue(x))).t.find(x);
}
```

A primeira operação find(x) (em xft) leva $O(\log w)$ de tempo. A segunda operação find(x) (em uma treap) leva $O(\log r)$ de tempo, onde r é o tamanho da treap. Mais adiante nesta seção, mostraremos que o tamanho esperado da treap é $O(w)$ tal que essa operação usa $O(\log w)$ de tempo.[1]

Adicionar um elemento a uma YFastTrie é razoavelmente simples – na maior parte do tempo. O método add(x) chama xft.find(x) para achar a treap t, onde x deve ser inserido. Então utiliza-se t.add(x) para adicionar x a t. Nesse ponto, lançamos uma moeda tendenciosa que sai cara com probabilidade $1/w$ e coroa com probabilidade $1 - 1/w$. Se nesse lançamento sair cara, então x será adicionado a xft.

É aqui que as coisas ficam um pouco mais complicadas. Quando x é adicionado a xft, a treap t precisa ser dividida em duas treaps, t1 e t′. A treap t1 contém todos os valores menores ou iguais a x; t′ é a treap original, t, com os elementos de t1 removidos. Assim que isso é feito, adicionamos o par (x, t1) em xft. A Figura 13.8 mostra um exemplo.

YFastTrie

```
boolean add(T x) {
  int ix = it.intValue(x);
  STreap<T> t = xft.find(new Pair<T>(ix)).t;
  if (t.add(x)) {
    n++;
    if (rand.nextInt(w) == 0) {
      STreap<T> t1 = t.split(x);
      xft.add(new Pair<T>(ix, t1));
    }
    return true;
  }
  return false;
}
```

Adicionar x a t leva $O(\log w)$ de tempo. O Exercício 7.12 mostra que dividir t em t1 e t′ também pode ser feito em $O(\log w)$ de tempo esperado. Adicionar o par (x,t1) em xft custa $O(w)$ de tempo, mas acontece com probabilidade $1/w$. Portanto, o tempo esperado de execução da operação

[1] Essa é uma aplicação da *Desigualdade de Jensen*: Se $\mathrm{E}[r] = w$, então $\mathrm{E}[\log r] \le \log w$.

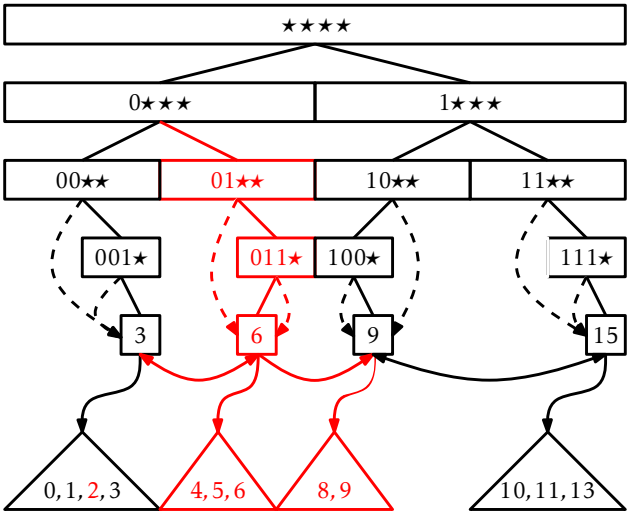


Figura 13.8: Adicionando os valores 2 e 6 em uma `YFastTrie`. O lançamento de moeda para 6 sai cara, então 6 foi adicionada a `xft` e a treap contendo 4, 5, 6, 8, 9 foi dividida.

`add(x)` é

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

O método `remove(x)` desfaz o trabalho de `add(x)`. Usamos `xft` para achar a folha, `u`, em `xft` que contém a resposta a `xft.find(x)`. A partir de `u`, pegamos a treap, `t`, contendo `x` e removemos `x` de `t`. Se `x` também foi guardado em `xft` (e `x` não é igual a $2^w - 1$) então removemos `x` de `xft` e adicionamos os elementos da treap de `x` à treap `t2`, que é guardada pelo sucessor de `u` na lista ligada. Isso é ilustrado na Figura 13.9.

YFastTrie

```
boolean remove(T x) {
  int ix = it.intValue(x);
  Node<T> u = xft.findNode(ix);
  boolean ret = u.x.t.remove(x);
  if (ret) n--;
  if (u.x.x == ix && ix != 0xffffffff) {
    STreap<T> t2 = u.child[1].x.t;
    t2.absorb(u.x.t);
```

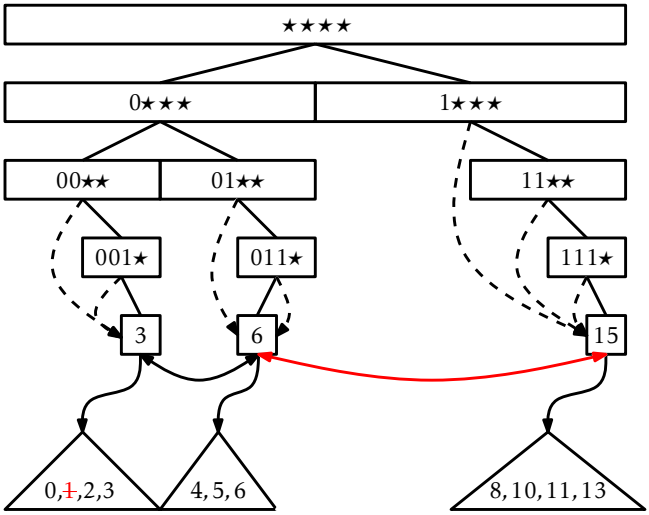


Figura 13.9: Remoção dos valores 1 e 9 de uma YFastTrie na Figura 13.8.

```
    xft.remove(u.x);
  }
  return ret;
}
```

Achar o nodo u em xft leva $O(\log w)$ de tempo esperado. Remover x de t leva $O(\log w)$ de tempo esperado. Novamente, o Exercício 7.12 mostra que a junção de todos os elementos de t em t2 t2 pode ser feito em tempo $O(\log w)$. Se necessária, a remoção de x de xft leva tempo $O(w)$, mas x é somente contido em xft com probabilidade $1/w$. Portanto, o tempo esperado para remover um elemento de uma YFastTrie é $O(\log w)$.

Anteriormente, postergamos a discussão sobre os tamanhos das treaps nessa estrutura. Antes de terminar este capítulo, provamos o resultado que necessitamos.

Lema 13.1. *Seja* x *um inteiro guardado em uma* YFastTrie *e seja* n_x *o número de elementos na* t *que contém* x*. Então* $E[n_x] \leq 2w - 1$.

Demonstração. Veja a Figura 13.10. Sejam $x_1 < x_2 < \cdots < x_i = x < x_{i+1} < \cdots < x_n$ os elementos guardados na YFastTrie. A treap t contém alguns

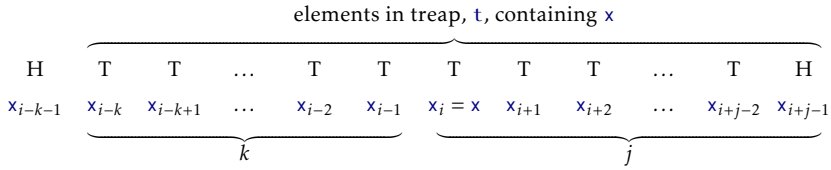


Figura 13.10: O número de elementos na treap t contendo x é determinado por dois lançamentos de moedas.

elementos maiores que ou iguais a x. Esses são $x_i, x_{i+1}, \ldots, x_{i+j-1}$, onde x_{i+j-1} é o único desses elementos para o qual o lançamento de moeda tendenciosa no método $\mathtt{add}(x)$ saiu no lado cara. Em outras palavras, $E[j]$ é igual ao número esperado de lançamentos tendenciosos necessários para obter a primeira cara. [2] Cada lançamento é independente e sai cara com probabilidade $1/w$, então $E[j] \le w$. (Veja o Lema 4.2 para uma análise para o caso $w = 2$.)

De modo similar, os elementos de t menores que x são $x_{i-1}, \ldots, x_{i-k}$ onde esses k lançamentos resultam em coroa e o lançamento para x_{i-k-1} sai cara. Portanto, $E[k] \le w - 1$, como esse é o mesmo experimento de lançamento de moedas considerados no parágrafo anterior, mas um em que o último lançamento não é contado. Em resumo, $n_x = j + k$, então

$$E[n_x] = E[j+k] = E[j] + E[k] \le 2w - 1 \quad . \qquad \square$$

O Lema 13.1 foi a última peça na prova do teorema a seguir, que resume o desempenho da `YFastTrie`:

Teorema 13.3. *Uma* `YFastTrie` *implementa a interface* `SSet` *para inteiros de* w *bits. Uma* `YFastTrie` *aceita as operações* $\mathtt{add}(x)$, $\mathtt{remove}(x)$ *e* $\mathtt{find}(x)$ *em tempo esperado* $O(\log w)$ *por operação. O espaço usado por uma* `YFastTrie` *que guarda* n *valores é* $O(n + w)$.

O termo w nos custos de espaço vem do fato que $\mathtt{xft}$ sempre guarda o valor $2^w - 1$. A implementação pode ser modificada (em troca de alguns casos extras para serem considerados no código) tal que é desnecessário

[2] Essa análise ignora o fato que j nunca passa de $n - i + 1$. Entretanto, isso somente reduz $E[j]$, então o limitante superior ainda vale.

guardar esse valor. Nesse caso, o custo de espaço no teorema torna-se $O(n)$.

13.4 Discussão e Exercícios

A primeira estrutura de dados a prover as operações add(x), remove(x) e find(x) com tempo $O(\log w)$ foi proposta por van Emde Boas e é conhecida como a *árvore de van Emde Boas* (ou *árvore estratificada*) [74]. A estrutura original de van Emde Boas tinha tamanho 2^w, tornando-a inadequada para inteiros grandes.

As estruturas de dados XFastTrie e YFastTrie foram descobertas por Willard [77]. A estrutura XFastTrie é relacionada às árvores de van Emde Boas; por exemplo, as tabelas hash em uma XFastTrie substituem arrays em uma árvore de van Emde Boas. Isto é, em vez de guardar a tabela hash $t[i]$, uma árvore de van Emde Boas guarda um array de comprimento 2^i.

Outra estrutura para guardar inteiros é a árvore de fusão Fredman e Willard [32]. Essa estrutura pode guardar n inteiros de w bits em espaço $O(n)$ tal que a operação find(x) roda em tempo $O((\log n)/(\log w))$.

Ao usar uma árvore de fusão quando $\log w > \sqrt{\log n}$ e uma YFastTrie quando $\log w \leq \sqrt{\log n}$, é possível obter uma estrutura de dados que usa $O(n)$ de espaço que pode implementar a operação find(x) em tempo $O(\sqrt{\log n})$. Resultados recentes de limitantes inferiores de Pătraşcu e Thorup [59] mostram que esses resultados são mais ou menos ótimos, pelo menos para estruturas que usam somente $O(n)$ de espaço.

Exercício 13.1. Projete e implemente uma versão simplificada de uma BinaryTrie que não tem uma lista ligada nem ponteiros de salto mas cujo find(x) que roda em tempo $O(w)$.

Exercício 13.2. Projete e implemente uma implementação simplificada de uma XFastTrie que não usa uma trie binária. Em vez disso, a sua implementação deve guardar tudo em uma lista duplamente ligada em $w+1$ tabelas hash.

Exercício 13.3. Podemos pensar da BinaryTrie como uma estrutura que guarda strings de bits de comprimento w de forma que cada bitstring é re-

presentada como um caminho da raiz para uma folha. Amplie essa ideia em uma implementação SSet que guarda strings de comprimento variável e implementa add(s), remove(s) e find(s) em tempo proporcional ao comprimento de s.
Dica: Cada nodo na sua estrutura de dados deve guardar uma tabela hash que é indexada por valores de caracteres.

Exercício 13.4. Para um inteiro $x \in \{0, \ldots 2^w - 1\}$, seja $d(x)$ a diferença entre x e o valor retornado por find(x) [se find(x) retorna null, então defina $d(x)$ como 2^w]. Por exemplo, se find(23) retorna 43, então $d(23) = 20$.

1. Projete e implemente uma versão modificada da operação find(x) em uma XFastTrie que roda em tempo esperado $O(1 + \log d(x))$. Dica: a tabela hash $t[w]$ contém todos os valores x tal que $d(x) = 0$, então esse seria um bom ponto para começar.

2. Projete e implemente uma versão modificada da operação find(x) em uma XFastTrie que roda em tempo esperado $O(1 + \log\log d(x))$.

Capítulo 14

Busca em Memória Externa

Ao longo deste livro, consideramos o modelo de computação word-RAM definido na Seção 1.4. Uma premissa implícita desse modelo é que nosso computador tem memória de acesso aleatório grande o suficiente para guardar todos os dados na estrutura de dados. Em algumas situação essa premissa não é válida. Existem coleções de dados tão grandes que nenhum computador tem memória suficiente para guardá-las. Nesses casos, a aplicação deve guardar os dados em alguma mídia de armazenamento externo tal como um disco rígido, um disco em estado sólido ou mesmo um servidor de arquivos em rede (que possui seu próprio armazenamento externo).

Acessar um item em um armazenamento externo é extremamente lento. O disco rígido conectado ao computador no qual este livro foi escrito tem um um tempo de acesso médio de 19ms e o disco em estado sólido conectado ao computador tem tempo médio de acesso de 0.3ms. Em comparação, a memória de acesso aleatório no computador tem tempo médio de acesso de menor que 0.000113ms. Acessar a RAM é mais do que 2 500 vezes mais rápido que acessar um disco em estado sólido e mais de 160 000 vezes mais rápido do que acessar o disco rígido.

Essas velocidades são típicas; acessar um byte aleatório na RAM é milhares de vezes mais rápido que acessar um byte aleatório em um disco rígido ou um disco de estado sólido. Tempo de acesso, entretanto, não conta a história por completo. Ao acessar um byte de um disco rígido ou disco de estado sólido, um *bloco* inteiro do disco é lido. Cada um dos discos conectados ao computador tem um tamanho de bloco de 4 096; cada

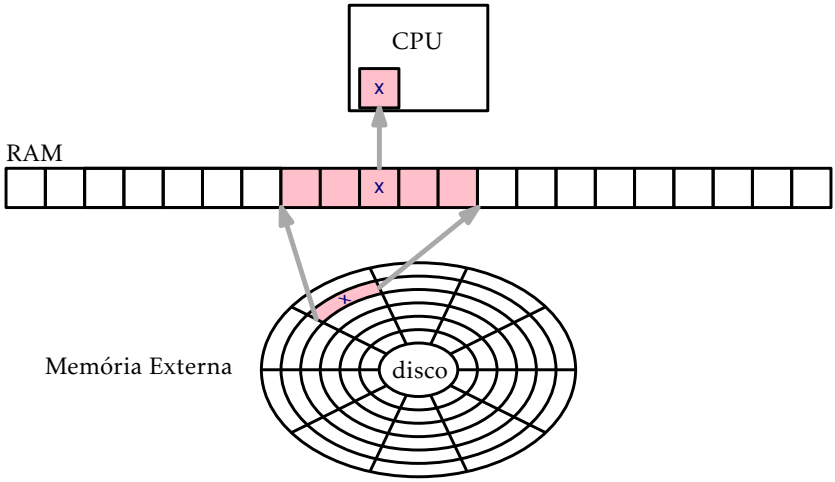


Figura 14.1: No modelo de memória externa, acessar um único item x na memória externa exige a cópia bloco inteiro contendo x para a RAM.

vez que lemos um byte, o disco nos fornece um bloco contendo 4 096 bytes. Se organizarmos nossa estrutura de dados cuidadosamente, isso significa que cada acesso a disco pode nos enviar 4 096 bytes que podem ser úteis em completar quaisquer operação que estamos executando.

Essa é a ideia por trás do *modelo de memória externa* de computação, ilustrado esquematicamente na Figura 14.1. Neste modelo, o computador tem acesso a uma grande memória externa no qual todos os dados são guardados. Essa memória é dividida em *blocos* de memória , cada um contendo B palavras. O computador também tem memória interna limitada na qual é possível realizar computações. Transferir um bloco entre a memória interna e a memória externa leva tempo constante. Computações realizadas na memória interna são *grátis*; elas não gastam *nenhum* instante de tempo. O fato que computações em memória externa são grátis são parecer estranho, mas isso enfatiza o fato que memória externa é muito mais lenta que a RAM.

Em um modelo de memória externa mais detalhado, o tamanho da memória interna também é um parâmetro. Entretanto, para as estruturas de dados descritas neste capítulo, é suficiente ter uma memória interna de tamanho $O(B + \log_B n)$. Por isso, a memória precisa ser capaz de guardar

um um número constante de blocos e uma pilha de recursão de altura $O(\log_B \mathtt{n})$. Na maior parte dos casos, o termo $O(B)$ domina os custos de memória. Por exemplo, mesmo com um relativamente pequeno $B = 32$, $B \geq \log_B \mathtt{n}$ para todo $\mathtt{n} \leq 2^{160}$. Em decimal, $B \geq \log_B \mathtt{n}$ para qualquer

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

14.1 A *Block Store*

A noção de memória externa inclui um grande número de diferentes dispositivos, cada qual com seu próprio tamanho de bloco e é acessado com sua própria coleção de chamadas de sistema. Para simplificar a exposição deste capítulo para que possamos nos concentrar nas ideias em comum entre eles, encapsulamos os dispositivos de memória externa com um objeto chamado de `BlockStore` (em português, armazém de blocos). Uma `BlockStore` guarda uma coleção de blocos de memória, cada uma com tamanho B. Cada bloco é unicamente identificado por seu índice inteiro. Uma `BlockStore` possui as operações:

1. `readBlock(i)`: retorna o conteúdo do bloco cujo índice é `i`.
2. `writeBlock(i,b)`: escreve o conteúdo de `b` ao bloco cujo índice é `i`,
3. `placeBlock(b)`: retorna um novo índice e guarda o conteúdo de `b` nesse índice.
4. `freeBlock(i)`: Libera o bloco cujo índice é `i`. Isso indica que o conteúdo desse bloco não será mais usado, então a memória externa alocada por esse bloco pode ser reusada.

O jeito mais fácil de usar a `BlockStore` é imaginá-la guardando um arquivo em disco que é particionado em blocos, cada contendo B bytes. Dessa maneira, `readBlock(i)` e `writeBlock(i,b)` simplesmente lêem e escrevem os bytes $\mathtt{i}B, \ldots, (\mathtt{i}+1)B - 1$ desse arquivo. Adicionalmente, uma simples `BlockStore` poderia manter uma *lista de blocos livres* que lista aqueles que estão disponíveis para uso. Blocos liberados com `freeBlock(i)` são adicionados à lista de blocos livres. Dessa forma, `placeBlock(b)` pode

usar um bloco da lista de blocos livres, ou caso nenhum esteja livre, adiciona um novo bloco no final do arquivo.

14.2 B-Trees

Nesta seção, discutimos uma generalização de árvores binárias chamadas B-trees, que é eficiente no modelo de memória externa. Alternativamente, B-trees podem ser vistas como generalização natural da árvore 2-4 descrita na Seção 9.1. (Uma árvore 2-4 é um caso especial de uma B-tree quando $B = 2$.)

Para qualquer inteiro $B \geq 2$, uma *B-tree* é uma árvore na qual todas as folhas têm a mesma profundidade e todo nodo interno (exceto a raiz) `u` tem pelo menos B filhos e no máximo $2B$ filhos. Os filhos de `u` são guardados em um array `u.children`. O número obrigatório de filhos é diferente na raiz, que pode ter entre 2 e $2B$ filhos.

Se a altura de uma B-tree é h, então o número ℓ, de folha em uma B-tree satisfaz

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

Aplicando o logaritmo na primeira desigualdade e rearranjando termos chegamos em:

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

Isto é, a altura de uma B-tree é proporcional ao logaritmo base-B do número de folhas.

Cada nodo `u` na B-tree guarda um array de até $2B-1$ posições contendo as chaves `u.keys`$[0], \ldots,$ `u.keys`$[2B-1]$. Se `u` é um nodo interno com k filhos, então o número de chaves guardado em `u` é exatamente $k-1$ e esses são guardados em `u.keys`$[0], \ldots,$ `u.keys`$[k-2]$. O restante das $2B-k+1$ entradas do array em `u.keys` é atribuído a `null`.

Se `u` for uma folha não raiz , então `u` contém entre $B-1$ e $2B-1$ chaves. As chaves em uma B-tree respeitam uma ordem similar às chaves em uma

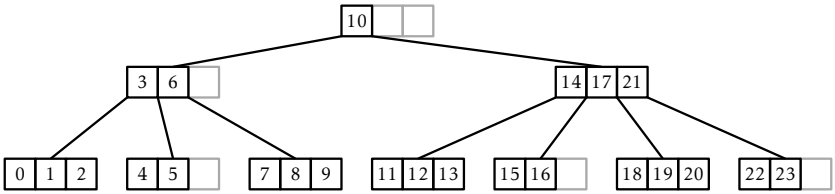


Figura 14.2: Uma B-tree com $B = 2$.

árvore binária de busca. Para qualquer nodo u que guarda $k-1$ chaves,

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

Se u for um nodo interno, então para todo i $\in \{0,\ldots,k-2\}$, u.keys[i] é maior que toda chave na subárvore enraizada em u.children[i] e menor que toda chave guardada nas subárvores enraizadas em u.children[i + 1]. Informalmente,

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

Um exemplo de uma B-tree com $B = 2$ é mostrado na Figura 14.2.

Note que os dados guardados em um nodo de uma B-tree tem tamanho $O(B)$. Portanto, em um contexto de memória externa, o valor de B em uma B-tree é escolhido para que um nodo caiba em um único bloco de memória externo. Dessa forma, o tempo que leva para realizar uma operação da B-tree no modelo de memória externa é proporcional ao número de nodos que são acessados (lidos ou escritos) pela operação.

Por exemplo, se as chaves são inteiros de 4 bytes e os índices dos nodos também ocupam 4 bytes, então fazer $B = 256$ significa que cada nodo guarda

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

bytes de dados.

Esse seria um valor perfeito de B para o disco rígido ou o disco de estado sólido discutido na introdução deste capítulo, que tem um tamanho de bloco de 4096 bytes.

A classe BTree, que implementa uma B-tree, mantém uma BlockStore, bs, que guarda nodos BTree assim como o índice ri do nodo raiz.

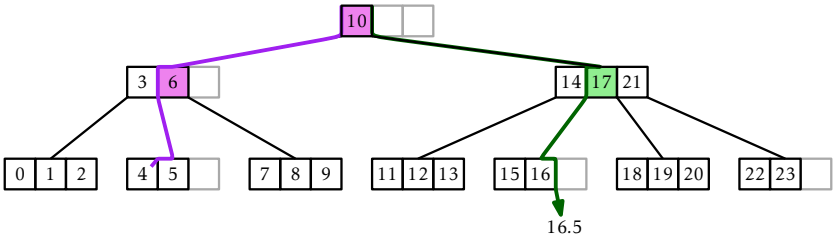


Figura 14.3: Uma busca bem sucedida (pelo valor 4) e uma busca mal sucedida (pelo valor 16.5) em uma *B*-tree. Nodos destacados mostram onde o valor de `z` é atualizado durante as buscas.

Como de costume, um inteiro `n`, é usado para registrar o número de itens na estrutura de dados:

BTree
```
int n;
BlockStore<Node> bs;
int ri;
```

14.2.1 Busca

A implementação da operação `find(x)`, que é ilustrada na Figura 14.3, generaliza a operação `find(x)` em uma árvore binária de busca. A busca por `x` começa na raiz e usa as chaves guardadas em um nodo `u` para determinar em qual dos filhos de `u` a busca deve continuar.

Mais especificamente, em um nodo `u`, a busca verifica se `x` está guardada em `u.keys`. Caso positivo, `x` foi encontrado e a busca está completa.

Por outro lado, a busca acha o menor inteiro `i` tal que `u.keys[i]` > `x` e continua a busca na subárvore enraizada em `u.children[i]`. Se nenhuma chave em `u.keys` for maior que `x`, então a busca continua no filho mais à direita de `u`. Assim como em árvores binárias de busca, o algoritmo registra a chave mais recentemente vista `z` que é maior que `x`. No caso de `x` não ser encontrado, `z` é retornado como o menor valor que é maior ou igual a `x`.

BTree
```
T find(T x) {
```

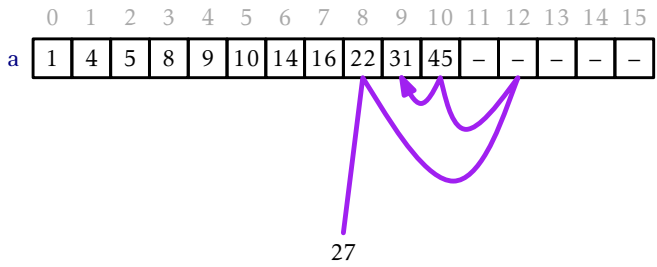


Figura 14.4: A execução de `findIt(a,27)`.

```
  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;
}
```

O método `findIt(a,x)` é central para a operação `find(x)` que busca em `a`, um array ordenado completado com valores `null`, pelo valor `x`. Esse método, ilustrado na Figura 14.4, funciona para qualquer array `a`, onde `a`$[0],\ldots,$`a`$[k-1]$ é uma sequência de chaves ordenadas `a`$[k],\ldots,$`a`$[$`a.length`$-1]$ são todas atribuídas a `null`. Se `x` está na posição `i` array, então `findIt(a,x)` retorna $-$`i`-1. Caso contrário, ele retorna o menor índice `i` tal que `a`$[$`i`$] >$ `x` ou `a`$[$`i`$] =$ `null`.

BTree

```
int findIt(T[] a, T x) {
  int lo = 0, hi = a.length;
  while (hi != lo) {
    int m = (hi+lo)/2;
    int cmp = a[m] == null ? -1 : compare(x, a[m]);
    if (cmp < 0)
      hi = m;      // look in first half
```

```
    else if (cmp > 0)
      lo = m+1;    // look in second half
    else
      return -m-1; // found it
  }
  return lo;
}
```

O método `findIt(a,x)` usa uma busca binária que divide o espaço de busca a cada passo, tal que ele roda em tempo $O(\log(\texttt{a.length}))$. No nosso cenário, $\texttt{a.length} = 2B$, então `findIt(a,x)` roda em tempo $O(\log B)$.

Podemos analisar o tempo de execução de uma operação `find(x)` da B-tree tanto no modelo word-RAM usual (onde cada instrução conta) quanto no modelo de memória externa (onde somente contamos o número de nodos acessados).

Como cada folha em uma B-tree guarda pelo menos uma chave e a altura de uma B-Tree com ℓ folhas é $O(\log_B \ell)$, a altura de uma B-tree que guarda `n` chaves é $O(\log_B \texttt{n})$. Portanto, no modelo de memória externa, o tempo gasto pela operação `find(x)` é $O(\log_B \texttt{n})$. Para determinar o tempo de execução no modelo word-RAM, temos que considerar o custo de chamar `findIt(a,x)` para cada nodo que acessamos, então o tempo de execução de `find(x)` no modelo word-RAM é

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

14.2.2 Adição

Uma diferença importante entre as estruturas de dados B-tree e `BinarySearchTree` da Seção 6.2 é que os nodos de uma B-tree não guardam ponteiros para nodos pai. A razão disso vai ser explicada logo a seguir. A falta de ponteiros dos nodos pai significa que as operações `add(x)` e `remove(x)` em B-trees são mais facilmente implementadas usando recursão.

Assim como todas as árvores binárias balanceadas, alguma forma de balanceamento é necessária durante uma operação `add(x)`. Em uma B-tree, isso é feito pela *divisão* (em inglês, *split*) de nodos. Observe na Figura 14.5 como isso ocorre. Embora a divisão aconteça entre dois níveis de recursão, ela é melhor entendida como uma operação que recebe um

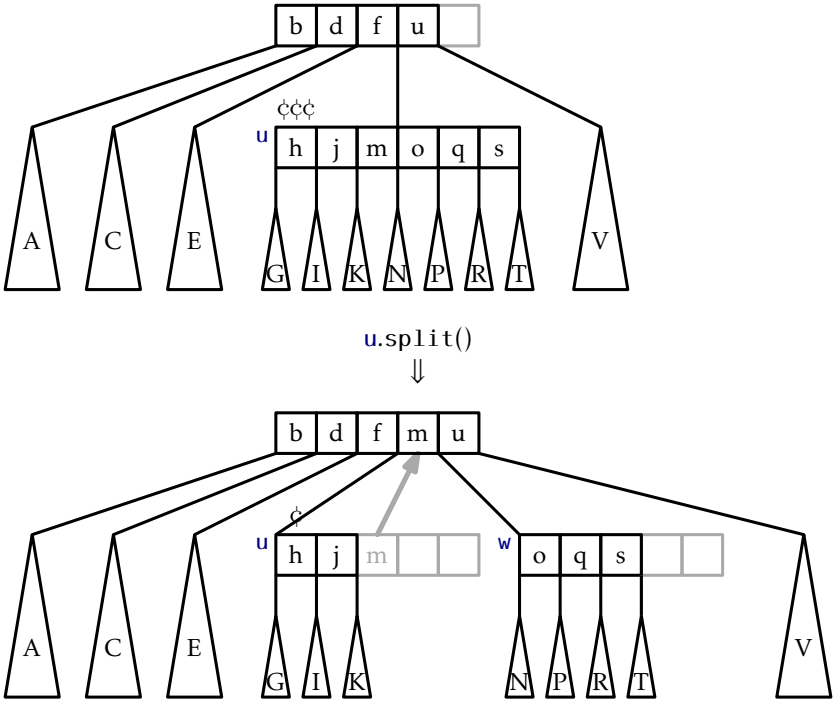


Figura 14.5: Divisão do nodo u em uma B-tree ($B = 3$). Note que a chave u.keys[2] = m passa de u ao seu pai.

nodo u contendo $2B$ chaves e com $2B+1$ filhos. A operação cria um novo nodo w que adota u.children[B],...,u.children[$2B$]. O novo nodo w também recebe as B maiores chaves de u, u.keys[B],...,u.keys[$2B-1$]. Neste ponto, u tem B filhos e B chaves. A chave extra, u.keys[$B-1$], é promovida para o pai de u, que também adota w.

Note que a operação de divisão modifica três nodos: u, o pai de u e o novo nodo w. Por isso é importante que os nodos de uma B-tree não mantenham ponteiros para os nodos pai. Se assim fosse, então os $B+1$ filhos adotados por w todos necessitariam ter os seus ponteiros para o nodo pai modificados. Isso aumentaria o número de acessos à memória externa de 3 para $B+4$ e faria a B-tree muito menos eficiente para valores altos de B.

O método add(x) em uma B-tree está ilustrado na Figura 14.6. Em

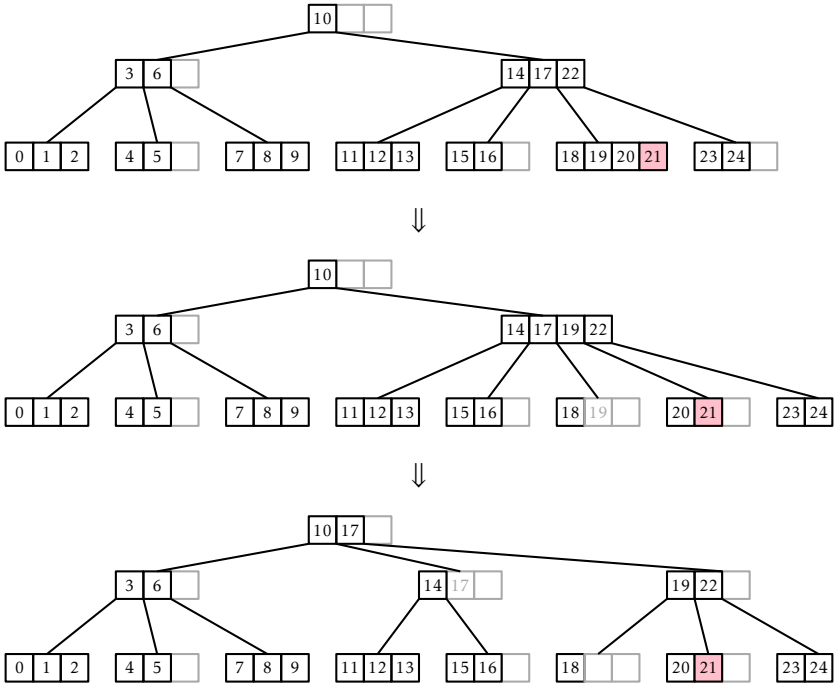


Figura 14.6: A operação add(x) em uma BTree. A adição do valor 21 resulta em dois nodos divididos.

uma descrição sem entrar em detalhes, esse método acha uma folha u para achar o valor x. Se isso faz com que u fique cheia demais, então o pai de u também é dividido, o que pode fazer que o avô de u fique cheio demais e assim por diante.

Esse processo continua, subindo na árvore um nível por vez até alcançar um nodo que não está cheio e até que a raiz seja dividida. Ao encontrar um nodo com espaço, o processo é interrompido. Caso contrário, uma nova raiz é criada com dois filhos obtidos a partir da divisão da raiz original.

O resumo do método add(x) é que ele caminha da raiz para a folha em busca de x, adiciona x a essa folha e retorna de volta para a raiz, dividindo qualquer nodo cheio demais que encontre no caminho. Com essa visão em mente, podemos entrar nos detalhes de como esse método pode ser

implementado recursivamente.

O principal trabalho de add(x) é feito pelo método addRecursive(x,ui), que adiciona o valor x à subárvore cuja raiz u tem o identificador ui. Se u for uma folha, então x simplesmente é adicionado em u.keys. Caso contrário, x é adicionado recursivamente no filho u′ de u. O resultado dessa chamada recursiva é normalmente null mas também pode ser uma referência a um novo nodo w que foi criado porque u′ foi dividido. Nesse caso, u adota w e recebe sua primeira chave, completando a operação de divisão em u′.

Após o valor x ser adicionado (em u ou em um descendente de u), o método addRecursive(x,ui) verifica se u está cheio demais (mais de $2B-1$) chaves. Desse modo, então u precisa ser dividido com uma chamada ao método u.split(). O resultado de chamar u.split() é um novo nodo que é usado como um valor de retorno para addRecursive(x,ui).

BTree

```
Node addRecursive(T x, int ui) {
  Node u = bs.readBlock(ui);
  int i = findIt(u.keys, x);
  if (i < 0) throw new DuplicateValueException();
  if (u.children[i] < 0) { // leaf node, just add it
    u.add(x, -1);
    bs.writeBlock(u.id, u);
  } else {
    Node w = addRecursive(x, u.children[i]);
    if (w != null) {  // child was split, w is new child
      x = w.remove(0);
      bs.writeBlock(w.id, w);
      u.add(x, w.id);
      bs.writeBlock(u.id, u);
    }
  }
  return u.isFull() ? u.split() : null;
}
```

O método addRecursive(x,ui) é auxiliar ao método add(x), que chama addRecursive(x,ri) para inserir x na raiz da B-tree. Se addRecursive(x,ri) fizer que a raiz seja dividida, então uma nova raiz é criada e ela recebe

como filhos tanto a antiga raiz quanto o novo nodo criado pela divisão da antiga raiz.

BTree
```
boolean add(T x) {
  Node w;
  try {
    w = addRecursive(x, ri);
  } catch (DuplicateValueException e) {
    return false;
  }
  if (w != null) {   // root was split, make new root
    Node newroot = new Node();
    x = w.remove(0);
    bs.writeBlock(w.id, w);
    newroot.children[0] = ri;
    newroot.keys[0] = x;
    newroot.children[1] = w.id;
    ri = newroot.id;
    bs.writeBlock(ri, newroot);
  }
  n++;
  return true;
}
```

O método add(x) e seu auxiliar, addRecursive(x,ui), podem ser analisados em duas fases:

Fase da descida: Durante a fase da descida da recursão, antes da adição de x, é acessada uma sequência de nodos da BTree e é chamado findIt(a,x) em cada nodo. Assim como o método find(x), isso leva $O(\log_B n)$ de tempo no modelo de memória externa e $O(\log n)$ de tempo no modelo word-RAM.

Fase de subida: Durante a fase de subida da recursão, após a adição de x, esses métodos realizam uma sequência de até $O(\log_B n)$ divisões. Cada divisão envolve somente de três nodos e, então, essa fase leva $O(\log_B n)$ de tempo no modelo de memória externa. Entretanto, cada divisão envolve mover B chaves e filhos de um nodo a outro, então no modelo word-RAM, isso leva $O(B \log n)$ de tempo.

Lembre-se que o valor de B pode ser bem alto, muito maior que $\log n$. Portanto no modelo word-RAM com palavras de `w` bits, adicionar um valor a uma B-tree pode ser bem mais lento do que adicionar em uma árvore binária de busca balanceada. Posteriormente, na Seção 14.2.4, mostraremos que a situação não é tão ruim; o número amortizado de operações de divisão durante uma operação `add(x)` é constante. Isso mostra que o tempo de execução amortizado da operação `add(x)` no modelo word-RAM é $O(B + \log n)$.

14.2.3 Remoção

A operação `remove(x)` em uma `BTree` é, novamente, mais facilmente implementada como um método recursivo. Embora a implementação recursiva de `remove(x)` distribui sua complexidade entre vários métodos, o processo como um todo, que é ilustrado na Figura 14.7, é razoavelmente direto. Ao trabalhar com as chaves, remoção é reduzida ao problema de remoção de um valor `x'`, de alguma folha `u`. Remover `x'` pode deixar `u` com menos que $B - 1$ chaves; essa situação é chamada de *underflow*.

Quando um *underflow* acontece, `u` pode emprestar chaves ou ser juntado com seus irmão. Se `u` for juntado com um irmão, então o pai de `u` terá agora um filho a menos e uma chave a menos, o que pode fazer o pai de `u` sofrer um *underflow*; isso é novamente corrigido por meio de um empréstimo ou de uma junção, embora a junção pode fazer que o avô de `u` sofra *underflow*. Esse processo se repete até a raiz até que não ocorra mais *underflows* ou até que a raiz tenha seus dois filhos juntados em um filho único. Quando os filhos são juntados, a raiz é removida e seu único filho se torna a nova raiz.

A seguir, entramos nos detalhes de como esses passos são implementados. O primeiro trabalho do método `remove(x)` é achar o elemento `x` que deve ser removido. Se `x` é achado em uma folha, então `x` é removido dessa folha. Caso contrário, se `x` é achado em `u.keys[i]` para algum nodo interno `u` então o algoritmo remove o menor valor `x'` na subárvore enraizada em `u.children[i + 1]`. O valor `x'` é o menor valor guardado na `BTree` que é maior que `x`. O valor de `x'` é então usado para substituir `x` em `u.keys[i]`. Esse processo é ilustrado na Figura 14.8.

O método `removeRecursive(x,ui)` é uma implementação recursiva do

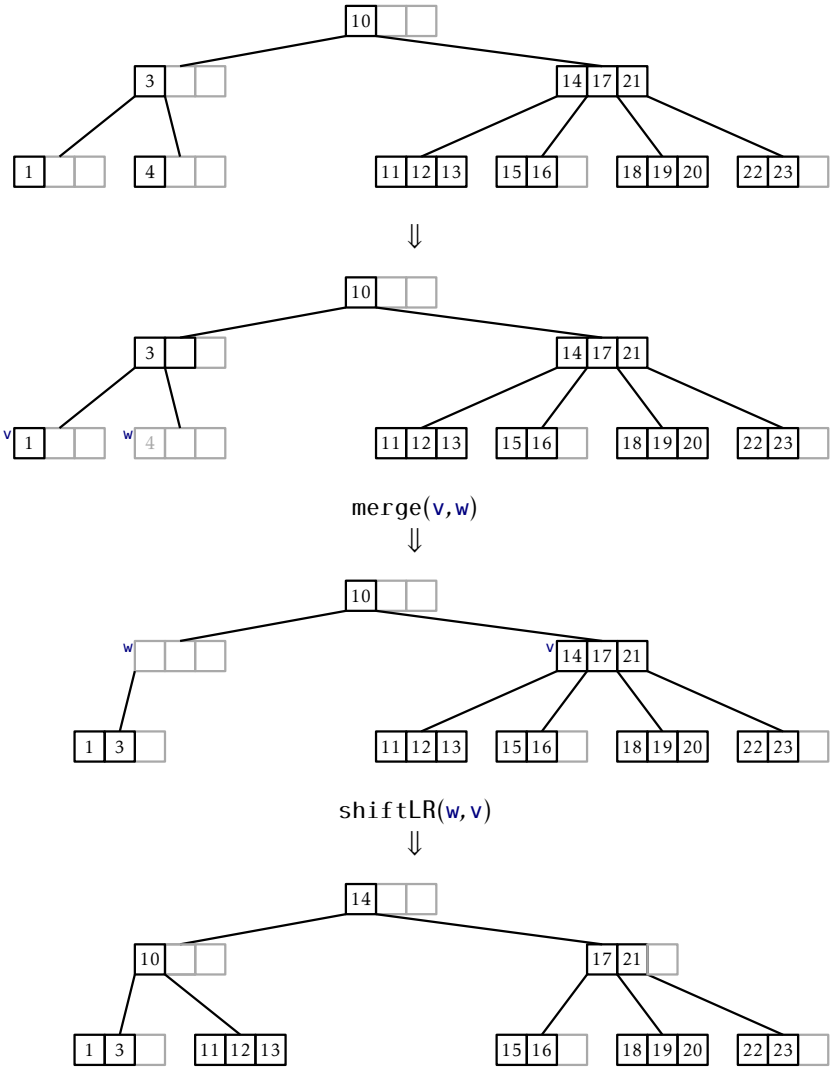


Figura 14.7: Remoção do valor 4 de uma *B*-tree resulta em uma operação de junção e um empréstimo.

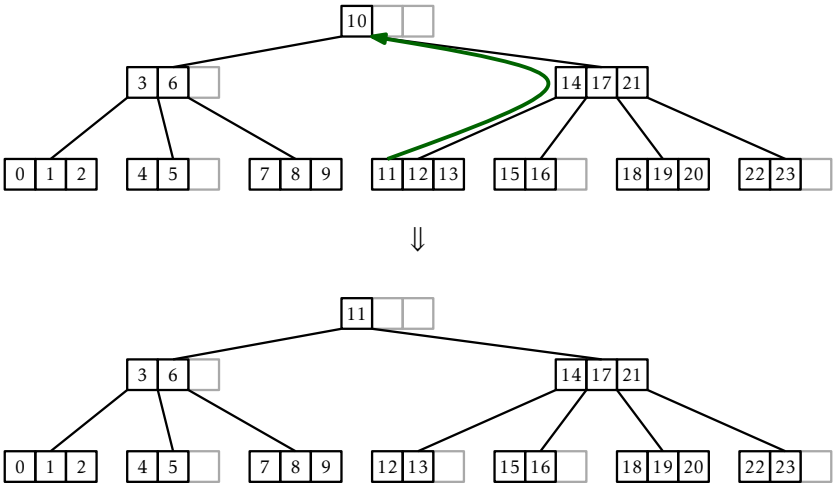


Figura 14.8: A operação remove(x) em uma BTree. Para remover o valor x = 10, o substituímos com o valor x′ = 11 e removemos 11 da folha que o contém.

algoritmo anteriormente discutido:

BTree

```
boolean removeRecursive(T x, int ui) {
  if (ui < 0) return false;  // didn't find it
  Node u = bs.readBlock(ui);
  int i = findIt(u.keys, x);
  if (i < 0) { // found it
    i = -(i+1);
    if (u.isLeaf()) {
      u.remove(i);
    } else {
      u.keys[i] = removeSmallest(u.children[i+1]);
      checkUnderflow(u, i+1);
    }
    return true;
  } else if (removeRecursive(x, u.children[i])) {
    checkUnderflow(u, i);
    return true;
  }
  return false;
```

```
}
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;
}
```

Note que, após remover recursivamente o valor `x` o `i`-ésimo filho de `u`, `removeRecursive(x,ui)` precisa garantir que esse filho ainda tem pelo menos $B-1$ chaves. No código precedente, isso é feito usando um método chamado `checkUnderflow(x,i)`, que verifica se ocorreu *underflow* e o corrige no `i`-ésimo filho de `u`. Seja `w` o `i`-ésimo filho de `u`. Se `w` tem somente $B-2$ chaves, então isso precisa ser corrigido. A correção requer usar um irmão de `w`. Ele pode ser o filho `i` + 1 ou `i` − 1 de `u`. Normalmente, iremos usar o filho `i` − 1 de `u`, que é o irmão `v` de `w` diretamente à sua esquerda. A única situação em que isso não funciona é quando `i` = 0 e nesse caso em que usamos o irmão de `w` diretamente à sua direita.

BTree
```
void checkUnderflow(Node u, int i) {
  if (u.children[i] < 0) return;
  if (i == 0)
    checkUnderflowZero(u, i); // use u's right sibling
  else
    checkUnderflowNonZero(u,i);
}
```

A seguir, nos concentramos no caso em que `i` ≠ 0 tal que qualquer *underflow* no `i`-ésimo filho de `u` será corrigido com a ajuda do filho (`i` − 1) de `u`. O caso `i` = 0 é similar e os detalhes podem ser encontrados no código que acompanha este livro.

Para corrigir um *underflow* no nodo `w`, precisamos achar mais chaves (e possivelmente também filhos) para `w`. Existem duas formas de fazer isso:

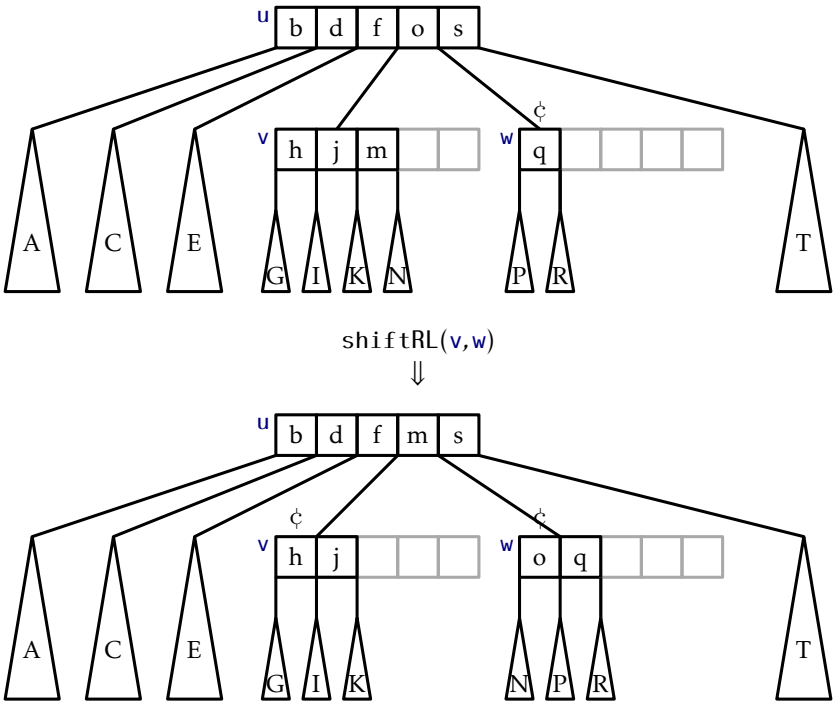


Figura 14.9: Se v tem mais que $B-1$ chaves, então w pode emprestar chaves de v.

Empréstimo: Se w tem um irmão v com mais de $B-1$ chaves, então w pode emprestar algumas chaves (e possivelmente também filhos) de v. Mais especificamente, se v guarda size(v) chaves, então entre eles, v e w tem um total de

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

chaves. Podemos portanto deslocar chaves de v a w para que v e w tenham pelo menos $B-1$ chaves. Esse processo está ilustrado na Figura 14.9.

Junção: Se v tem somente $B-1$ chaves, precisamos fazer algo mais drástico , como v não pode emprestar chaves a w. Portanto, *juntamos* v e w conforme mostrado na Figura 14.10. A operação de junção é o oposto da operação de divisão.

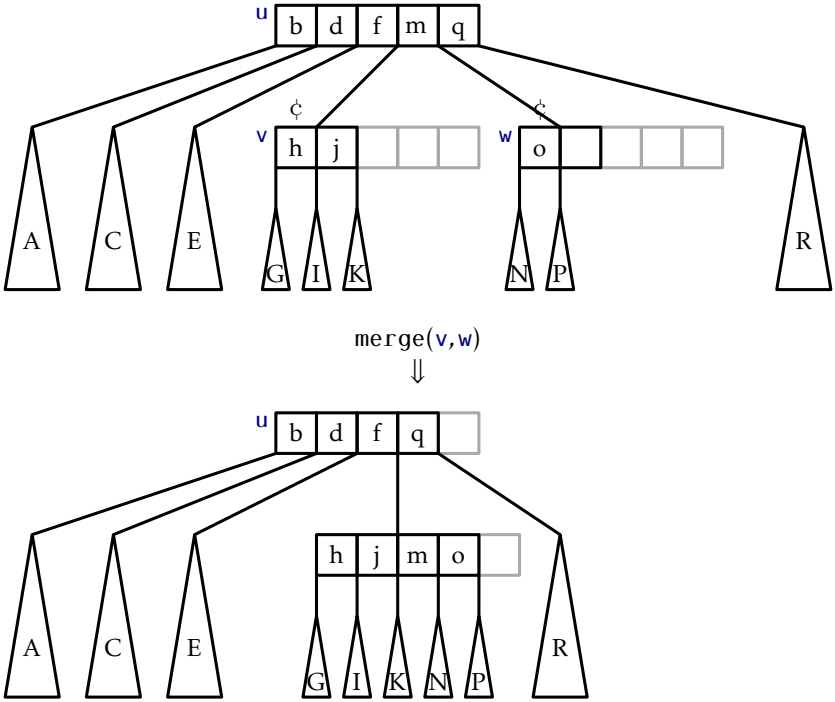


Figura 14.10: Junção de dois irmãos v e w em uma B-tree ($B = 3$).

Ela usa dois nodos que contêm um total de $2B - 3$ chaves e os junta em um único nodo que contém $2B - 2$ chaves. (A chave adicional vem do fato que, quando juntamos v e w, o pai deles u agora tem um filho a menos e portanto precisa ceder uma de suas chaves.)

BTree

```
void checkUnderflowNonZero(Node u, int i) {
  Node w = bs.readBlock(u.children[i]);  // w is child of u
  if (w.size() < B-1) {  // underflow at w
    Node v = bs.readBlock(u.children[i-1]); // v left of w
    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);
    }
```

```
  }
}
void checkUnderflowZero(Node u, int i) {
  Node w = bs.readBlock(u.children[i]); // w is child of u
  if (w.size() < B-1) {  // underflow at w
    Node v = bs.readBlock(u.children[i+1]); // v right of w
    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;
    }
  }
}
```

Para resumir, o método `remove(x)` em uma B-tree segue um caminho da raiz para a folha, remove uma chave `x'` de uma folha `u` e então realiza zero ou mais operações de junção envolvendo `u` e seus ancestrais e também realiza no máximo uma operação de empréstimo.

Como cada operação de junção e empréstimo, envolve modificar somente três nodos e somente $O(\log_B n)$ dessas operações acontecem, o processo completo leva $O(\log_B n)$ de tempo no modelo de memória externa. Novamente, entretanto, cada operação de junção e empréstimo leva cerca de $O(B)$ de tempo no modelo word-RAM, então (por ora) o máximo que podemos dizer sobre o tempo de execução necessário para `remove(x)` no modelo RAM é que é $O(B \log_B n)$.

14.2.4 Análise Amortizada da B-Tree

Até agora, mostramos que

1. no modelo de memória externa, o tempo de execução de `find(x)`, `add(x)` e `remove(x)` em uma B-tree é $O(\log_B n)$.

2. no modelo RAM com palavras de `w` bits, o tempo de execução de `find(x)` é $O(\log n)$ e o tempo de execução de `add(x)` e `remove(x)` é $O(B \log n)$.

O lema a seguir mostra que, até o momento, superestimamos o número de operações de junção e divisão realizadas por uma B-tree.

Lema 14.1. *Começando com uma B-tree vazia e realizando uma sequência de m operações* add(x) *e* remove(x) *resulta em até* $3m/2$ *divisões, junções e empréstimos realizados.*

Demonstração. A prova desse fato foi anteriormente rascunhada na Seção 9.3 para o caso especial em que $B = 2$. O lema pode ser provado usando um esquema de créditos, em que

1. cada operação de divisão, junção e empréstimo é pago com dois créditos, isto é, um crédito é removido cada vez que uma dessas operações ocorre; e

2. no máximo três créditos são criados durante qualquer operação add(x) ou remove(x).

Como no máximo $3m$ créditos são criados e cada divisão, junção e empréstimo é pago com dois créditos, segue que no máximo $3m/2$ divisões, junções e empréstimos são realizados. Esses créditos são ilustrados usando o símbolo ¢ nas Figuras 14.5, 14.9 e 14.10.

Para manter o controle desses créditos, a prova mantém o seguinte *invariante de crédito*: Qualquer nodo que não seja a raiz com $B-1$ chaves guarda um crédito e qualquer nodo com $2B-1$ chaves guarda três créditos. Um nodo que guarda pelo menos B três créditos. Um nodo que guarda pelo menos B chaves e até $2B-2$ chaves não precisa guardar nenhum crédito. O que resta mostrar é que podemos manter a invariante de crédito e satisfazer as propriedades 1 e 2 acima durante cada operação add(x) e remove(x).

Adição: O método add(x) não faz nenhuma junção ou empréstimo, então precisamos somente considerar as operações de divisão resultantes de chamadas a add(x).

Cada operação de divisão ocorre porque uma chave é adicionada a um nodo u que possui $2B-1$ chaves. Quando isso acontece, u é dividido em dois nodos, u′ e u″ com $B-1$ e B chaves, respectivamente. Antes de operação, u guardava $2B-1$ chaves e portanto três créditos. Dois desses créditos podem ser usados para pagar pela divisão e o outro crédito pode ser dado para u′ (que tem B–1 chaves) para manter a invariante de crédito.

Portanto, podemos pagar pela divisão e manter a invariante de crédito durante qualquer divisão.

O única outra modificação em nodos que ocorrem durante uma operação add(x) acontece após todas divisões, se houverem, forem completadas. Essa modificação envolve adicionar uma nova chave a algum nodo u'. Se, antes disso, u' tinha $2B-2$ filhos, então ele tem agora $2B-1$ filhos e deve portanto receber três créditos. Esses são os únicos créditos cedidos pelo método add(x).

Remoção: Durante uma chamada a remove(x), zero ou mais junções ocorrem e são possivelmente seguidas por um único empréstimo. Cada junção ocorre porque dois nodos, v e w, possuem cada um $B-1$ chaves antes da chamada ao método remove(x) são juntados em um único nodo com exatamente $2B-2$ chaves. Cada uma dessas junções portanto liberam dois créditos que podem ser usados para pagar pela junção.

Após as junções serem realizadas, ocorre no máximo uma operação e depois dela nenhuma junção ou empréstimo adicional é realizado. Essa operação de empréstimo somente ocorre se removermos uma chave de uma folha v que tem $B-1$ chaves. O nodo v portanto tem um crédito e esse crédito vai pagar o custo do empréstimo. Esse crédito não é suficiente para pagar o empréstimo, então criamos um crédito para completar o pagamento.

Nesse ponto, criamos um crédito e ainda necessitamos mostrar que a invariante de crédito pode ser mantida. No pior caso, o irmão de v, w, tem exatamente B chaves antes do empréstimo tal que, após isso, tanto v quanto w têm $B-1$ chaves. Isso significa que v e w devem estar guardando 1 crédito cada quando a operação terminar. Portanto, nesse caso, criamos dois créditos adicionais para dar para v e w. Como um empréstimo acontece no máximo uma vez durante uma operação remove(x), isso significa que criamos no máximo três créditos, conforme necessário.

Se a operação remove(x) não inclui uma operação de empréstimo, isso é porque ela termina removendo uma chave de algum nodo que, antes da operação, tinha B ou mais chaves. No pior caso, esse nodo tinha B chaves, então que ele tem agora $B-1$ chaves e precisa receber um crédito, que então criamos.

Nos dois casos—se a remoção termina com uma operação de emprés-

timo ou não— no máximo três créditos precisam ser criados durante uma chamada a remove(x) para manter a invariante de crédito e pagar por todos os empréstimos e junções que ocorrerem. Isso completa a prova do lema. □

O propósito do Lema 14.1 é mostrar que, no modelo word-RAM, o custo de divisões e junções durante uma sequência de m operações add(x) e remove(x) é somente $O(Bm)$. Isto é, o custo amortizado por operação é somente $O(B)$, então o custo amortizado de add(x) e remove(x) no modelo word-RAM é $O(B + \log n)$. Isso é resumido no seguinte par de teoremas:

Teorema 14.1 (B-Tree em Memória Externa)**.** *Uma BTree implementa a interface SSet. No modelo de memória externa, uma BTree suporta as operações* add(x), remove(x) *e* find(x) *em tempo* $O(\log_B n)$ *por operação.*

Teorema 14.2 (B-Tree em word-RAM)**.** *Uma BTree implementa a interface SSet. No modelo word-RAM, e ignorando o custo das divisões, junções e empréstimos, uma BTree provê as operações* add(x), remove(x) *e* find(x) *em tempo* $O(\log n)$ *por operação. Adicionalmente, iniciando com uma BTree vazia, uma sequência de m operações* add(x) *e* remove(x) *resulta em um total de* $O(Bm)$ *tempo gasto realizando divisões, junções e empréstimos.*

14.3 Discussão e Exercícios

O modelo de memória externa de computação foi introduzido por Aggarwal e Vitter [4]. Ele também é conhecido como o *modelo de I/O* ou como o *modelo de acesso a disco*.

B-Tree é para buscas em memória externa o que árvores binárias de busca são para busca em memória interna. B-trees foram inventadas por Bayer e McCreight [9] em 1970 e, em menos de 10 anos, o título do artigo de Comer na revista *ACM Computing Surveys* as referia como onipresentes [15].

Assim como para árvores binárias de busca, existem muitas variantes de B-Trees, incluindo B^+-trees, B^*-trees, e *counted B-trees*. B-trees são realmente onipresentes e são a estrutura de dados primária em muitos sistemas de arquivos, incluindo o HFS+ da Apple, o NTFS da Microsoft,

e o Ext4 usado no Linux; todos os principais sistemas de banco de dados; e armazéns de chave-valor usados em computação em nuvem. O levantamento recente de Graefe [36] provê uma visão geral de mais de 200 páginas de muitas aplicações modernas, variantes e otimizações de B-trees.

B-trees implementa a interface `SSet`. Se a interface `USet` for necessária, então hashing em memória externa pode ser usado como alternativa à B-tree. Esquemas de hashing em memória externa existem; veja, por exemplo, o trabalho de Jensen e Pagh [43]. Esse esquemas implementam as operações `USet` em tempo $O(1)$ esperado no modelo de memória externa. Entretanto, devido a vários motivos, muitas aplicações ainda usam B-tree embora somente sejam necessárias operações `USet`.

Uma razão de B-trees serem tão populares é elas funcionam melhor que o limitante $O(\log_B n)$ de tempo de execução sugere. A razão para isso é, em cenários de memória externa, o valor de B é tipicamente bem alto – na casa das centenas ou milhares. Isso significa que 99% ou mesmo 99.9% dos dados em uma B-tree está guardado nas folhas.

Em um sistema de banco de dados com muita memória, pode ser possível manter todos os nodos internos de uma B-tree em RAM, pois eles representam somente 1% ou 0.1% do conjunto total de dados. Quando isso acontece, isso significa que uma busca em uma B-tree envolve uma busca bem rápida na RAM, nos nodos internos, seguida por um único acesso à memória externa para obter uma folha.

Exercício 14.1. Mostre o que acontece quando as chaves 1.5 e então 7.5 são adicionadas à B-tree na Figura 14.2.

Exercício 14.2. Mostre o que acontece quando as chaves 3 e então 4 são removidas da B-tree na Figura 14.2.

Exercício 14.3. Qual é o número máximo de nodos internos em uma B-tree que guarda `n` chaves (como uma função de `n` e B)?

Exercício 14.4. A introdução deste capítulo afirma que B-trees somente precisam de uma memória interna de tamanho $O(B + \log_B n)$. Porém, a implementação dada neste livro na verdade precisa de mais memória.

1. Mostre que a implementação dos métodos `add(x)` e `remove(x)` dados neste capítulo usam memória interna proporcional a $B \log_B n$.

2. Descreva como esses métodos poderiam ser modificados a fim de reduzir o consumo de memória a $O(B + \log_B n)$.

Exercício 14.5. Desenhe os créditos usados na prova do Lema 14.1 nas árvores das Figuras 14.6 e 14.7. Verifique que (com três créditos adicionais) é possível pagar pelas divisões, junções e empréstimos e manter a invariante de crédito.

Exercício 14.6. Projete uma versão modificada de uma B-tree em que nodos podem ter de B a $3B$ filhos (e portanto de $B-1$ até $3B-1$ chaves). Mostre que essa nova versão de B-tree realiza somente $O(m/B)$ divisões, junções e empréstimos durante uma sequência de m operações. (Dica: para isso funcionar, você deve ser mais agressivo com a junção e às vezes juntar dois novos antes que seja estritamente necessário.)

Exercício 14.7. Neste exercício, você irá projetar um método modificado de dividir e juntar em B-trees que assintoticamente reduz o número de divisões, empréstimos e junções ao considerar três nodos por vez.

1. Seja u um nodo sobrecarregado e seja v um irmão imediatamente à direita de u. Existem duas formas de consertar o excesso de chaves em u:

   (a) u pode dar algumas de suas chaves a v; ou

   (b) u pode ser dividido e as chaves de u e v podem ser igualmente distribuídas entre u, v e o novo nodo w.

   Mostre que isso sempre pode ser feito de maneira que, após a operação, cada um dos nodos afetados (no máximo 3) tem pelo menos $B + \alpha B$ chaves e no máximo $2B - \alpha B$ chaves, para alguma constante $\alpha > 0$.

2. Seja u um nodo com falta de chaves e sejam v e w irmão de u, existem duas maneiras de corrigir a falta de chaves em u:

   (a) chaves podem ser redistribuídas entre u, v e w; ou

   (b) u, v e w podem ser juntados em dois nodos e as chaves de u, v e w podem ser redistribuídos entre esses nodos.

Mostre que isso pode ser sempre feito de forma que, após a operações, cada um dos nodos afetados (no máximo 3) tem pelo menos $B + \alpha B$ chaves e no máximo $2B - \alpha B$ chaves, para alguma constante $\alpha > 0$.

3. Mostre que, com essas modificações, o número de junções, empréstimos e divisões que ocorrem durante m operações é $O(m/B)$.

Exercício 14.8. Uma B^+-tree, ilustrada na Figura 14.11, guarda todas as chaves em folhas e mantém suas folhas guardadas como uma lista duplamente encadeada. Como de costume, cada folha guarda entre $B - 1$ e $2B - 1$ chaves. Acima dessa lista está uma B-tree padrão que guarda o maior valor de cada folha exceto o último.

1. Descreva implementações rápidas de `add(x)`, `remove(x)`x, e `find(x)` em uma B^+-tree.

2. Explique como implementar eficientemente o método `findRange(x, y)`, que encontra todos os valores maiores que `x` e menores que ou iguais a `y`, em uma B^+-tree.

3. Implemente uma classe, `BPlusTree`, que implementa `find(x)`, `add(x)`, `remove(x)` e `findRange(x, y)`.

4. B^+-trees duplicam algumas das chaves porque elas são guardados na B-tree e na lista. Explique porque essa duplicação não é excessiva para valores altos de B.

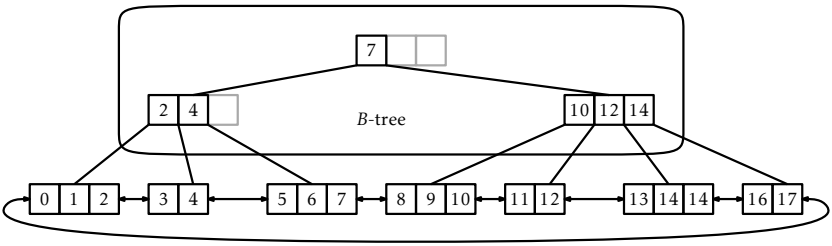


Figura 14.11: Uma B^+-tree é uma B-tree em cima de uma lista duplamente encadeada de blocos.

Bibliografia

[1] Free eBooks by Project Gutenberg. URL: http://www.gutenberg.org/ [cited 2011-10-12].

[2] IEEE Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, August 2008. doi:10.1109/IEEESTD.2008.4610935.

[3] G. Adelson-Velskii and E. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3(1259-1262):4, 1962.

[4] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[5] A. Andersson. Improving partial rebuilding by using simple balance criteria. In F. K. H. A. Dehne, J.-R. Sack, and N. Santoro, editors, *Algorithms and Data Structures, Workshop WADS '89, Ottawa, Canada, August 17–19, 1989, Proceedings*, volume 382 of *Lecture Notes in Computer Science*, pages 393–402. Springer, 1989.

[6] A. Andersson. Balanced search trees made simple. In F. K. H. A. Dehne, J.-R. Sack, N. Santoro, and S. Whitesides, editors, *Algorithms and Data Structures, Third Workshop, WADS '93, Montréal, Canada, August 11–13, 1993, Proceedings*, volume 709 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 1993.

[7] A. Andersson. General balanced trees. *Journal of Algorithms*, 30(1):1–18, 1999.

[8] A. Bagchi, A. L. Buchsbaum, and M. T. Goodrich. Biased skip lists. In P. Bose and P. Morin, editors, *Algorithms and Computation, 13th International Symposium, ISAAC 2002 Vancouver, BC, Canada, November 21–23, 2002, Proceedings*, volume 2518 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2002.

[9] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. In *SIGFIDET Workshop*, pages 107–141. ACM, 1970.

[10] Bibliography on hashing. URL: http://liinwww.ira.uka.de/bibliography/Theory/hash.html [cited 2011-07-20].

[11] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In M. J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15–19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 79–79. Springer, 1999.

[12] P. Bose, K. Douïeb, and S. Langerman. Dynamic optimality for skip lists and b-trees. In S.-H. Teng, editor, *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20–22, 2008*, pages 1106–1114. SIAM, 2008.

[13] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick. Resizable arrays in optimal time and space. In Dehne et al. [18], pages 37–48.

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

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

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

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

[18] F. K. H. A. Dehne, A. Gupta, J.-R. Sack, and R. Tamassia, editors. *Algorithms and Data Structures, 6th International Workshop, WADS '99, Vancouver, British Columbia, Canada, August 11–14, 1999, Proceedings*, volume 1663 of *Lecture Notes in Computer Science*. Springer, 1999.

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

[20] P. Dietz and J. Zhang. Lower bounds for monotonic list labeling. In J. R. Gilbert and R. G. Karlsson, editors, *SWAT 90, 2nd Scandinavian Workshop on Algorithm Theory, Bergen, Norway, July 11–14, 1990, Proceedings*, volume 447 of *Lecture Notes in Computer Science*, pages 173–180. Springer, 1990.

[21] M. Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In C. Puech and R. Reischuk, editors, *STACS 96, 13th Annual Symposium on Theoretical Aspects of Computer Science, Grenoble, France, February 22–24, 1996, Proceedings*, volume 1046 of *Lecture Notes in Computer Science*, pages 567–580. Springer, 1996.

[22] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In W. Kuich, editor, *Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria, July 13–17, 1992, Proceedings*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246. Springer, 1992.

[23] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997.

[24] M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. M. auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

[25] A. Elmasry. Pairing heaps with $O(\log\log n)$ decrease cost. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 471–476. Society for Industrial and Applied Mathematics, 2009.

[26] F. Ergun, S. C. Sahinalp, J. Sharp, and R. Sinha. Biased dictionaries with fast insert/deletes. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 483–491, New York, NY, USA, 2001. ACM.

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

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

[29] M. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

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

[31] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with 0 (1) worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[32] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of computer and system sciences*, 47(3):424–436, 1993.

[33] I. Galperin and R. Rivest. Scapegoat trees. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 165–174. Society for Industrial and Applied Mathematics, 1993.

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

[35] M. T. Goodrich and J. G. Kloss. Tiered vectors: Efficient dynamic arrays for rank-based sequences. In Dehne et al. [18], pages 205–216.

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

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

[38] L. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, 16–18 October 1978, Proceedings*, pages 8–21. IEEE Computer Society, 1978.

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

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

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

[42] HP-UX process management white paper, version 1.3, 1997. URL: `http://h21007.www2.hp.com/portal/download/files/prot/files/STK/pdfs/proc_mgt.pdf` [cited 2011-07-20].

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

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

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

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

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

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

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

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

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

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

[53] J. I. Munro, T. Papadakis, and R. Sedgewick. Deterministic skip lists. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms (SODA'92)*, pages 367–375, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

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

[55] Oracle. *Java Platform Standard Ed. 6*. URL: http://download.oracle.com/javase/6/docs/api/ [cited 2011-07-19].

[56] Oracle. *The Java Tutorials*. URL: http://download.oracle.com/javase/tutorial/ [cited 2011-07-19].

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

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

[59] 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.

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

[61] 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].

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

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

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

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

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

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

[68] 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.

[69] 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.

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

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

[72] 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.

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

[74] 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.

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

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

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

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

Índice

9-1-1, 2

aleatorização, 16
algoritmo de ordenação
    baseado em comparação, 234
algoritmo de ordenação estável, 250
algoritmo in place, 252
algoritmo recursivo, 142
altura
    de uma skiplist, 92
    de uma árvore, 141
    em uma árvore, 141
ancestral, 139
Aproximação de Stirling, 12
aresta, 255
aresta direcionada, 255
aritmética modular, 39
armazenamento externo, 293
array circular, 40
array de apoio, 31
`ArrayDeque`, 43
`ArrayQueue`, 39
arrays, 31
`ArrayStack`, 32
ataque de complexidade algoritmico, 137

B^*-tree, 314
B^+-tree, 314
B-tree, 296
backing array, 31
`Bag`, 29
balanceada no tamanho, 154
balanceamento na altura, 215
`BDeque`, 76
Bibliography on Hashing, 133
binary search tree, 145
    red-black, 193
binary tree
    search, 145
`BinaryHeap`, 219
`BinarySearchTree`, 145
`BinaryTree`, 141
`BinaryTrie`, 276
block store, 295
`BlockStore`, 295
bloco, 293, 294
bode expiatório, 181
bounded deque, 76
`BPlusTree`, 317
breadth-first traversal, 145
breadth-first-search, 265
busca binária, 282, 300
busca de sucessor, 10
busca em largura, 265
busca em profundidade, 267

busca finger
    em uma treap, 178
busca na web, 2

caminho de busca
    em uma BinaryTrie, 276
    em uma skiplist, 92
    em uma árvore binária de busca, 146
caminho/ciclo simples, 255
celebridade, *ver* universal sink
ChainedHashTable, 111
ciclo, 255
circular
    array, 40
coeficientes binomiais, 12
Comparator, 234
compare(a,b), 234
compare(x,y), 9
compartilhar, xiv
complexidade
    espaço, 21
    tempo, 21
complexidade de espaço, 21
complexidade de tempo, 21
componentes conectados, 272
constante de Euler, 11
conted *B*-tree, 314
cor, 198
corpo primal, 130
corretude, 21
CountdownTree, 191
counting-sort, 248
CubishArrayStack, 65
custo amortizado, 22
custo esperado, 22
código hash, 111, 127
    para arrays, 130
    para dados primitivos, 127
    para strings, 130
códigos hash
    para objetos compostos, 128

DaryHeap, 231
decreaseKey(u,y), 230
dependências, 24
deque, 6
    bounded, 76
    delimitado, 76
descendente, 139
detecção de ciclos, 269
dicionário, 9
dictionary, 9
disco rígido, 293
divisão e conquista, 234
DLList, 71
doubly-linked list, 71
DualArrayDeque, 46
dummy node, 71
DynamiteTree, 192

e (constante de Euler), 11
eficiente no espaço
    lista ligada, 75
elemento pivot, 238
empréstimo, 308
encadeamento, 111
endereçamento aberto, 118, 133
espaço desperdiçado, 57
espécies de árvores, 153
esquema de créditos, 188, 312
estrutura de dados aleatorizada, 16

estrutura secundária, 285
evento de especiação, 153
exponencial, 10
Ext4, 315

`FastArrayStack`, 38
fatorial, 12
FIFO queue, 5
fila, 5
fila com prioridades, 5
fila LIFO, 6
filho, 139
    direita, 139
    esquerda, 139
filho à direita, 139
filho à esquerda, 139
finger, 108, 178
finger search
    in a skiplist, 108
floresta geradora, 272
função hash
    perfeita, 133
funções hash perfeitas, 133
fusão, 198

general balanced tree, 189
gerenciador de memória, 63
`git`, xiv
Google, 3
grafo, 255
    fortemente conectado, 272
    não direcionado, 271
grafo conectado, 272
grafo de conflitos, 255
grafo direcionado, 255
grafo fortemente contectado, 272
grafo não direcionado, 271
graph
    connected, 272
grau, 263

H_k (número harmônico), 161
hash table, 111
`hash(x)`, 112
hashing
    multiplica e soma, 134
    multiplicativo, 134
    tabulação, 175
hashing com encadeamento, 111, 133
hashing cuckoo , 134
hashing em memória externa, 315
hashing multiplica e soma, 134
hashing multiplicativo, 114, 134
hashing perfeito, 133
hashing por tabulação, 126, 175
hashing universal , 134
hashing with chaining, 133
heap, 219
    binomial, 230
    binária, 219
    esquerdista, 230
    Fibonacci, 230
    pairing, 230
heap binomial, 230
heap binária, 219
heap de Fibonacci, 230
heap esquerdista, 230
heap-sort, 241
HFS+, 314

independência em relação ao mínimo, 175

interface, 4
invariante de crédito, 312

Java Collections Framework, 24
Java Runtime Environment, 63
junção, 309

lançamento de moeda, 18, 103
LIFO queue, *ver também* stack, 6
limitante inferior, 244
limitante inferior para ordenação, 244
`LinearHashTable`, 118
linearidade da esperança, 18
linked list, 67
    doubly-, 71
    singly-, 67
    space-efficient, 75
    unrolled, *ver também* `SEList`
`List`, 7
lista de contatos, 1
lista duplamente ligada, 71
lista ligada, 67
    dupla, 71
lista simplesmente ligada, 67
listas de adjacências, 260
logaritmo, 11
    binário, 11
    natural, 11
logaritmo binário, 11
logaritmo natural, 11

map, 9
mapa, 9
matriz de adjacências, 257
matriz de incidência, 271
`MeldableHeap`, 225
`memcpy(d,s,n)`, 38
memória externa, 293
merge-sort, 89
mergesort, 234
`MinDeque`, 90
`MinQueue`, 90
`MinStack`, 90
modelo de acesso a disco, 314
modelo de I/O, 314
modelo de memória externa, 294
modificar, xiv
multiplicativo
    hashing, 114
método de Eytzinger, 219
método do potencial, 50, 85
método potencial, 214

`n`, 23
nodo dummy, 71
nodo preto, 198
nodo sentinela, 92
nodo vermelho, 198
notação assintótica, 13
notação big-Oh, 13
NTFS, 315
numeração
    em-ordem, 155
    pré-ordem, 155
    pós-ordem, 155
numeração em-ordem, 155
numeração pré-ordem, 155
numeração pós-ordem, 155
número harmônico, 161

O notation, 13
open addressing, 133
Open Source, xiii

ordem em uma pilha, 220
ordenação baseada em comparação, 234
origem, 255

pai, 139
pair, 9
pairing heap, 230
Palavra Dyck, 29
palíndromo, 88
path, 255
permutação, 12
    aleatória, 160
permutação aleatória, 160
pilha, 6
potencial, 50
potential method, 85
prime field, 130
priority queue, *ver também* heap, 5
probabilidade, 16
profundidade, 139
propriedade da altura preta, 198
propriedade da árvore binária de busca, 146
propriedade de nenhuma aresta vermelha, 198
propriedade de pender à esquerda, 201
propriedade raiz, 165

queue, 5
    FIFO, 5
    LIFO, 6
    priority, 5
quicksort, 238

radix sort, 250
RAM, 20
randomized algorithm, 16
randomized data structure, 16
`RandomQueue`, 63
raízes quadradas, 59
reconstrução parcial, 181
red-black tree, 193
`RedBlackTree`, 201
rede social, 1
repartição, 195
resolução de colisões, 133
`RootishArrayStack`, 52
rotação, 167
rotação à direita, 167
rotação à esquerda, 167
run, 122

scapegoat, 181
`ScapegoatTree`, 183
`SEList`, 75
sentinela, 71
`Sequence`, 192
serviços de emergência, 2
singly-linked list, 67
sistema de arquivos, 1
skew heap, 230
skiplist, 91
    versus árvore binária de busca, 109
`SkiplistList`, 97
`SkiplistSSet`, 94
`SLList`, 67
Software Aberta, xiii
solid-state drive, 293
sondagem linear, 118
sorting algorithm

comparison-based, 234
split, 300
SSet, 9
stack, 6
std :: copy(a0,a1,b), 38
string
pareada, 29
string pareada, 29
successor search, 10
System.arraycopy(s, i, d, j, n), 38

tabela hash, 111
cuckoo, 134
dois níveis, 134
tabela hash de dois níveis, 134
target, 255
tempo de execução, 21
amortizado, 21
esperado, 16, 22
pior caso, 21
tempo de execução amortizado, 21
tempo de execução esperado, 16, 22
tempo de execução no pior caso, 21
teste de planaridade, 271
tipo abstrato de dados, *ver* interface
traversal
breadth-first, 145
travessia
de uma árvore binária, 142
em ordem, 155
pré-ordem, 155
pós-ordem, 155
travessia de árvore, 142
travessia de árvore binária, 142
travessia em largura, 145
travessia em ordem, 155
travessia pos ordem, 155
travessia pré-ordem, 155
Treap, 165
TreapList, 178
Treque, 63

underflow, 305
universal
hashing, 134
universal sink, 272
unrolled linked list, *ver também* SEList
USet, 8

valor esperado, 18
valor hash, 112
variável aleatória indicadora, 18
vetor em camadas, 62
vértice, 255
vértice alcançável, 255

WeightBalancedTree, 191
word, 20
word-RAM, 20

XFastTrie, 282
XOR-list, 87

YFastTrie, 285

árvore, 139
d-ária, 231
binária, 139
enraizada, 139
ordenada, 139
árvore AVL, 215
árvore binária, 139
completa, 223

ordenada em pilha, 220
árvore binária completa, 223
árvore binária de busca, 145
aleatória, 160
balanceada no tamanho, 154
balanceamento na altura, 215
randomizada, 175
rubro-negra, 193
versus skiplist, 109
árvore binária de busca aleatória, 160
árvore binária de busca randomizada, 175
árvore binária ordenada em pilha, 220
árvore de busca binária
reconstrução parcial, 181
árvore de comparações, 245
árvore de fusão, 291
árvore de van Emde Boas, 291
árvore enraizada, 139
árvore estratificada, 291
árvore ordenada, 139
árvore rubro-negra, 193, 201
árvore rubro-negra pendente à esquerda, 201
árvores balanceadas gerais, 189
árvores de pedigree, 153
árvores genealógicas, 153, 230