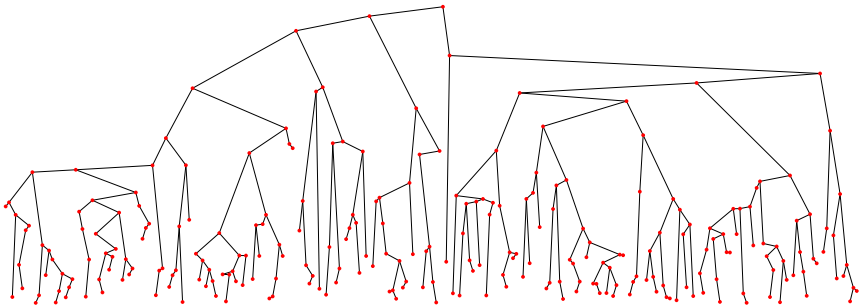


Open Data Structures (in C++) 日本語版

Edition 0.1Gβ

Pat Morin
堀江 慧、陣内 佑、田中 康隆　共訳

目次

訳者まえがき

本書は、“Open Data Structures” という本を日本語に訳したものだ。

訳者らも含め、日本語での生活に慣れていれば、英語よりも日本語で書かれた書籍のほうがだいぶスムーズに読めるだろう。とはいえ、これは訳者の個人的な意見だが、専門書を選ぶときにはついつい翻訳されたものを避けてしまう。品質のばらつきが大きかったり、むしろ読みにくかったりすることが多いと感じるからだ。

しかし、この教科書は入門書である。前提とする知識は、高校で習う数学の一部だけである（もちろん、簡単なプログラミング経験があったほうが内容に実感が持ててありがたみがわかり、楽しく読めるのは間違いない）。本書のような入門書が日本語で読めるようになっているほうが、分野の裾野を広げ、楽しくプログラムを書ける人や効率的なプログラムが書ける人を増やしてくれるだろう。

母国語で大学レベルの教科書が読める国は多くないといわれる。より専門的な内容はもっぱら英語で読むことになるのだから、さっさと崖から突き落としたほうがいいという意見も聞く。大学生になっても日本語で教科書が読めるという恵まれた環境が、日本人の英語アレルギーを支えている可能性もあるだろう。訳者自身も、「英語を読む」ところまでは受験勉強で慣れたものの、大学に入った頃はまだ「英語で読む」ことに抵抗があった。そのような抵抗を可及的速やかに取り除き、「英語で読む」ことに慣れるのは、アクセスできる知識を押し広げるために極めて重要である。

しかし、この教科書は専門分野への橋渡しの、その初っ端に位置するものだ。前提として要求される知識も多くない。それが、英語であるばかりに対象読者が大きく制限されているとしたら残念なことである。母国語でこのような入門書が読める、少なくともその選択肢があるのは望ましいことであろう。

この教科書は、300 ページ程度ながら、丁寧にゆっくりと、それでいて実用的な題材が扱われている。この分野には本書より本格的な教科書も数多く出

版されており、いくつかは翻訳もされている。訳者自身がいまでも折にふれ読み返すような、素晴らしい内容のものもある。例えば、“Algorithm Design”[*1] と “Introduction to Algorithms”[*2] の二冊は翻訳も良い。とはいえ、いずれも大判で 1000 ページ程度、価格は 1 万円程度という本であり、気軽に読めるものではない。こうした、より専門的な書籍への橋渡しであるこの教科書を日本語で、かつ無料でも読めるようにすることが、この翻訳プロジェクトの目的である。

本書の読み方

本書『みんなのデータ構造』の想定読者は、初学者からベテランのエンジニアまで、データ構造にかかわるすべての人である。訳者らが読者に伝えたいことは次の 3 つである。

1. ソフトウェアのほとんどはシンプルなデータ構造の組み合わせでできている。
   本書で紹介するデータ構造はシンプルなものである。よくある誤解は、これらのデータ構造は理論上のものであり、実際のソフトウェアはもっと複雑なデータ構造を使っているというものだ。これはまったくの間違いである。OS やブラウザなどの複雑なソフトウェアも、その実、シンプルなデータ構造の組み合わせでできている。本書で紹介するデータ構造が理解できれば、多くのソフトウェアの骨子が理解できるようになるだろう。言い換えれば、本書が紹介するのはおもちゃのデータ構造ではなく、現実のプログラムの中で実際に使われているデータ構造である。
2. 本書の内容がだいたいわかるようになれば良いエンジニアになれる。
   ソフトウェアのほとんどが基本的なデータ構造の組み合わせでできているということは、基本的なデータ構造を理解すれば、新しいソフトウェアをデザインし、既存のソフトウェアの改良ができるようになるということである。
3. わからない部分は飛ばしてもよい。
   本書には数学の理解を必要とする解析がある。理解できない部分があったら読み飛ばしても差し支えない。あるいは、詳しい知り合いや翻

[*1] Kleinberg, Jon, and Eva Tardos. Algorithm design. Pearson Education, 2006.
[*2] Cormen, Thomas H., et al. Introduction to algorithms. MIT press, 2009.

訳者、著者に質問するべきである。理解できない原因は読み手にはなく、書き手が問題だと考えるべきである。いずれにせよ、わからない箇所があったらそこで立ち止まるのではなく、そのまま先に進めるところまで進んでみることを勧めたい。

上記の 2 つめの点に関し、訳者らは、本書で扱われているデータ構造のうち実用上極めて重要な項目とそうではない項目とを明確に区別しておくことが有益だと考えた。以下に列挙する項目は、本書の中でも特に重要であると、訳者の 3 人全員が判断したものだ。学術研究やプログラマの実務で頻繁に登場する内容なので、すべての学習者が深く理解しておくことが望ましいだろう。

- 第 2 章： ArrayStack、ArrayQueue、ArrayDeque
- 第 3 章： SLList、DLList
- 第 5 章： ChainedHashTable
- 第 6 章： BinaryTree、BinarySearchTree
- 第 9 章： RedBlackTree（9.2.2 節から 9.2.4 節は複雑なので読み飛ばしてよい）
- 第 10 章： BinaryHeap
- 第 11 章： MergeSort、QuickSort
- 第 12 章： 幅優先探索、深さ優先探索

上記に列挙しなかった、ややマイナーなデータ構造にも、別の意味で学ぶ価値はある。マイナーゆえに直接役立つ機会は少ないかもしれないが、その背後にあるアイデアやその解析手法は多くの場面で読者の助けになるはずだ（単純に知的な面白さのある話題も多い）。どの章を重点的に読むか、興味に応じて適宜調整してほしい。

本書の日本語版のプロジェクトページ [*3] には、本書の他言語版やプロジェクトに関する情報がある。本書の日本語版のソースコードは GitHub[*4] にある。

訳者謝辞

まずなにより、本書の原著者である Pat Morin に感謝する。Pat は、Open Data Structures を立ち上げ、再配布、改変、販売を許容するライセンスで公開

[*3] https://sites.google.com/view/open-data-structures-ja
[*4] https://github.com/spinute/ods

してくれた。本書の日本語訳プロジェクトは、Pat が書いた “My hope is that, by doing things this way, this book will continue to be a useful textbook long after my interest in the project, or my pulse, (whichever comes first) has waned.” という一文に惹かれて始めたものである。Pat は、翻訳、クラウドファンディング、出版のいずれの相談にも前向きな返事をくれ、また、そのたびに encourage してくれた。

本プロジェクトでは、成果物の品質を高めるために、書籍をプロの編集者にレビューしてもらうための資金を募るクラウドファンディングを行った。このクラウドファンディングに参加していただいた皆様にも感謝を捧げたい。次の団体、企業、個人をはじめとしてたくさんの方々からご支援いただいた（敬称略）。

- 斉藤淳（J PREP）、東京大学瀧本哲史ゼミ、株式会社バオバブ、有限会社コンセントレーション
- 上田真道、長谷川悠斗、赤野健悟、石田修平、瓜生英尚、Kiyotaka Saito、桑田誠、小林元、石畠正和、t2nis、永浦 尊信、wtokuno、katsyoshi、ysaito、前原貴憲、木正弘、Yoshinari Takaoka（a.k.a mumumu）、T.Miyazawa、佐藤怜、田中哲朗、Masashi Fujiwara、雪村あおい、Tetsuya Yamazaki、永本、武平佑太、Yamachan0928、新海息吹、Daiki Sugiyama、早瀬元、山口駿人、長谷川拓也、okue、飛田晃介、大坂直人、松林祐、若杉武史、Hideki Hamada、落合哲治、鈴木 研吾、redfield920、山崎宏宇、宮田潔志

皆様のおかげで本書のソースコードを原著と同様に Creative Commons Attribution ライセンスで公開できた。また、本書の原稿は出版社でのレビューを経て、かなり読みやすくなった。さらには、余剰金で情報オリンピックの日本代表選抜に参加する学生に、書籍を献本することも計画している。この本が多くの人に読まれ、日本のプログラミングや情報科学を支える人たちの一助となることを強く願っている。

ラムダノート社の鹿野さん、高尾さんにもこの場をお借りしてお礼を申し上げたい。ラムダノートは 2015 年に設立された新進気鋭の技術出版社である。このプロジェクトのクラウドファンディングを見て、本書の編集作業を破格の条件で引き受けてくださった。また、出版の企画を持ちかけ、書籍としての完成度を高めるための惜しみない援助をしてくださった [*5]。

[*5] 本書の原稿はオープンソースであり、GitHub で公開されている。プロの編集者の手にかかると書籍がどう変貌するのかに興味がある読者は、2018 年 1 月時点の原稿と、いまのこ

この翻訳プロジェクトは、2017 年の春に堀江が個人的に始めたものである。その時点では、ただ日本語訳を作成することしか考えていなかった。その後、試しに陣内に原稿を見てもらったときに「この本は価値がある」と言われたことを受け、より多くの読者に読んでもらうためクラウドファンディングを企画した。また、陣内は共訳者として本書のすべての章をレビューし、プロジェクト自体の運営にも携わってくれた。一目置く陣内からの後押しは、GitHub の隅に眠っていたかもしれない本書の運命を変えた。堀江、陣内とは違った角度から問題を解決する能力のある田中も、本書のレビューとプロジェクトの運営との両面で協力してくれた。クラウドファンディングでの成功には人望の厚い田中の協力が欠かせなかった。また、多くの修正や訳注を加えて本書をより理解しやすくしてくれた。大学を卒業した後も、またこの二人と仕事ができて嬉しい。

2018 年 7 月

堀江 慧（@spinute）

の本を比較してみると感動するのではないかと思う。

謝辞

次の方々に感謝を捧げたい。夏に多くの章を勤勉に校正してくれた Nima Hoda、この本の初稿を読んで誤字や誤りをたくさん指摘してくれた 2011 年秋の COMP2402/2002 の受講生たち、完成に近づいた頃の数稿を根気強く校閲してくれた Athabasca University Press の Morgan Tunzelmann である。

なぜこの本を書いたのか

いろいろなデータ構造の入門書がある。出来の良いものもある。ほとんどはタダではないので、コンピュータサイエンスを学ぶ学部生はデータ構造の本にお金を払うだろう。

オンラインで公開されているデータ構造の本もある。名作もあるのだが古くなってきているものが多い。ほとんどは著者や出版社が更新をやめるときに無料になったものである。これらの本は次の理由からふつうは内容を更新できない。(1) 著者または出版社が著作権を持っていて、いずれかの許可を得られないため。(2) 書籍の**ソースコード**が提供されていないため。つまり、本の Word、WordPerfect、FrameMaker、または LaTeX ソースコードが手に入らない、またはそれを扱えるソフトウェアのバージョンが手に入らないため。

このプロジェクトの目標は、コンピュータサイエンスを専攻する学部生が負担するデータ構造の入門書代をゼロにすることだ。そのため、オープンソースのソフトウェアプロジェクトのようにこの本を作ることにした。この本の LaTeX ソース、C++ ソース、およびビルドスクリプトを、著者の Web サイト [*6]、あるいは信頼できるソースコード管理サイト [*7] からダウンロードできる。

ソースコードは Creative Commons Attribution ライセンスで公開されている。つまり、誰でも自由にコピー、配布、送信してよい。内容を取り入れて別の何かを作ってもよい。そしてそれを商業的に利用してもよい。唯一の条件は**帰属の表示**（attribution）である。つまり、派生した作品が `opendatastructures.org` のコードやテキストを含むことを明記しなければならない。

ソースコード管理システム `git` による修正を通して、誰でも本書に貢献で

[*6] `http://opendatastructures.org` ［訳注］日本語版の Web サイトは `https://sites.google.com/view/open-data-structures-ja`

[*7] `https://github.com/patmorin/ods` ［訳注］日本語版のソースコードは `https://github.com/spinute/ods`

きる。本のソースをフォークして、別のバージョンを作ってもよい（例えば、別のプログラミング言語を題材にした版を作れる）。こうしたやり方で、私のやる気や興味が衰えたあとでも、この本が役立つものであり続けることを望んでいる。

Pat Morin

C++ 版のまえがき

この本は基本的なデータ構造の設計と解析の手法、そしてオブジェクト指向言語での実装方法を教えるために書かれた。具体的なオブジェクト指向言語としては C++ を採用している。

この本は C++ を初めて学ぶ人のために書かれた本ではない。一方で、C++ あるいは似た言語の基本的な文法に馴染みがあれば、この本を読むには十分だ。もし付属のソースコードにまで目を通したければ、C++ でのプログラミング経験があったほうがいいだろう。

この本は C++ の標準テンプレートライブラリ（STL）や、その背景にあるジェネリックプログラミングの手ほどきをするものでもない。しかし、この本で実装を紹介するデータ構造の中には、STL でも使われているものも多く含まれている。STL を使うプログラマは、STL ではどうデータ構造を実装していて、それがなぜ効率的なのかを理解できるだろう。

第 1

イントロダクション

データ構造とアルゴリズムに関する授業は全世界でコンピュータサイエンスの課程に含まれている。データ構造はそれほど重要だ。生活の質を上げるだけでなく、毎日のように人の命さえ救っている。データ構造によって数百万ドル、数十億ドルの規模にまでなった企業も多い。

なぜデータ構造はこんなにも重要なのだろう？考えてみれば私たちは普段からさまざまなデータ構造と接している。

- ファイルを開く：ファイルシステムのデータ構造を使って、ファイルをハードディスクなどの上に配置し、検索できる。ハードディスクには数億ものブロックがあり、ファイルの内容はどのブロックに保存されていてもおかしくないので、これは簡単なことではない。
- 電話番号を検索する：入力の途中で連絡先リストから電話番号を検索するためにデータ構造が使われている。連絡先リストには膨大な情報（過去に電話や電子メールをやり取りした全員）が含まれている可能性があること、電話端末には高速なプロセッサや潤沢なメモリは搭載されていないことを考えると、これは簡単なことではない。
- SNS にログインする：ネットワークサーバーではログイン情報からアカウント情報を検索する。利用者が多い SNS には何億人ものアクティブなユーザーがいるので、これは簡単なことではない。
- Web ページを検索する：検索エンジンは検索語から Web ページを見つけるためにデータ構造を使う。インターネットには 85 億以上の Web ページがあり、それぞれのページに検索対象になりうる語句が大量に含まれているので、これは簡単なことではない。
- 緊急通報用番号（9-1-1）に電話する：緊急通報電話では、パトカー、救

急車、消防車を速やかに現場に手配できるよう、電話番号と住所を対応付けるためにデータ構造を使う。電話をかけた人は正確な住所を伝えられないかもしれず、この場面での遅れは生死を分かつこともあるので、これは重要な問題だ。

1.1 効率の必要性

次節では、よく使うデータ構造に対してどんな操作ができるのかを見ていく。ちょっとしたプログラミング経験があれば、正しい結果を返す操作を実装するのは難しくないだろう。データを配列や連結リストに入れ、すべての要素について順番に処理し、必要なら要素を追加したり削除したりするという実装にすればよい。

この実装は簡単だが、効率がよくない。とはいえ、効率について考える価値はあるだろうか？コンピュータはどんどん速くなっている。簡単な実装で十分かもしれない。それを確認するためにざっくりと計算をしてみよう。

操作の数： まあまあの大きさのデータセット、例えば 100 万（10^6）個の要素を持つアプリケーションがあるとする。各要素を少なくとも一回は見たくなるというのは、それなりに妥当な仮定だろう。この場合、少なくとも 100 万（10^6）回、このデータセットから要素を探すことになる。100 万回にわたって 100 万個の要素をすべて確認すると、データを読み出す回数は合計で 1 兆（$10^6 \times 10^6 = 10^{12}$）回になる。

プロセッサの速度： 本書執筆時点では、かなり高速なデスクトップコンピュータでも、毎秒 10 億（10^9）回以上の操作は実行できない[*1]。よって、このアプリケーションの完了には、少なくとも $10^{12}/10^9 = 1000$ 秒、すなわち約 16 分 40 秒かかる。コンピュータにとって 16 分は非常に長い時間だが、人間ならコーヒーブレイクを挟んでそれくらいの時間は待っていられるだろう。

大きなデータセット： Google について考えてみよう。Google では 85 億もの Web ページを対象にした検索を扱っている。先ほどの計算では、このデータに対する問い合わせには少なくとも 8.5 秒かかる。これは私たちが知っている Google とは違う。Google の Web 検索には 8.5 秒もかからないし、

[*1] コンピュータの速度はせいぜい数ギガヘルツ（数十億回/秒）であり、各操作にふつうは数サイクルが必要だ。

Google では特定のページがインデックスに含まれているか以上に複雑な問い合わせを実行している。本書執筆時点で、Google は 1 秒間に約 4,500 クエリを受け付ける。つまり、少なくとも $4,500 \times 8.5 = 38,250$ ものサーバーが必要だ。

解決策： 以上の例からは、安直な実装のデータ構造だと、要素数 n とデータ構造に対する操作数 m が共に大きくなったときに性能が追いつかなくなることがわかる。これらの例の実行にかかる時間は、機械命令の数におよそ $n \times m$ だ。

解決策はもちろん、データ構造内のデータを上手に並べ、各操作のたびに全要素を扱わないようにすることだ。一見すると不可能に思えるかもしれないが、要素がどれだけ多くても平均して 2 つの要素だけを参照すれば探していたデータが見つかるというデータ構造をのちに紹介する。毎秒 10 億回の命令を実行できるとして、10 億個の要素、あるいは兆、京、垓におよぶ数の要素が含まれていても、検索にわずか 0.000000002 秒しかかからないのだ。

要素を整列して保持するデータ構造についても紹介する。このデータ構造では、何らかの操作の実行中に参照される要素の数が、データ構造に格納されている要素数に対する関数として見たときに非常にゆっくりとしか増えない。例えば、どんな操作であれ実行中に最大で 60 個のアイテムしか参照しないで済むように、このデータ構造を整列された状態に維持できる。毎秒 10 億回の命令を実行できるコンピュータであれば、このデータ構造に対する操作がほんの 0.00000006 秒のうちに実行できることになる。

この章の残りの部分では、この本を通して使う主な概念の一部を簡単に解説する。1.2 節については、この本で説明するデータ構造で実装するインターフェースをすべて説明するので、必ず読んでほしい。残りの節では以下の内容を説明する。

- 指数、対数、階乗関数や漸近（ビッグオー）記法、確率、ランダム化などの数学の復習
- 計算のモデル
- 正しさと実行時間、メモリ使用量
- 残りの章の概要
- サンプルコードと組版の規則

これらの内容については、背景知識がある人もない人も、いったん読み飛ばしてから必要に応じて読み直してもらえばよい。

1.2 インターフェース

データ構造について議論するときは、データ構造のインターフェースとその実装との違いを理解することが重要だ。インターフェースはデータ構造が何をするかを、実装はデータ構造がそれをどのようにやるかを表現する。

インターフェース（interface）は、**抽象データ型**（abstract data type）とも呼ばれ、あるデータ構造がサポートしている操作一式と、それらの操作の意味（セマンティクス）を定義するものである。インターフェースを見ても、データ構造がサポートしている操作がどう実装されているかはわからない。インターフェースからわかるのは、そのデータ構造がサポートしている操作の一覧と、それらの操作に対する引数および返り値の特徴だけである。

一方、データ構造の**実装**（implementation）には、データ構造の内部表現と、実際に操作を行うアルゴリズムの定義が含まれる。そのため、1 つのインターフェースに対して複数の実装がありうる。例えば本書では、2 章では配列を使って `List` インターフェースを実装し、3 章ではポインタを使って `List` インターフェースを実装する。どちらも同じ `List` インターフェースだが、実装の方法が異なるというわけだ。

1.2.1 `Queue`、`Stack`、`Deque` インターフェース

`Queue` インターフェースは、要素の集まりを表しており、その集まりに対して要素の追加および特定のルールに従った削除ができる。より正確に言うと、`Queue` インターフェースには次の操作が実行できる。

- `add(x)`：値 `x` を `Queue` に追加する
- `remove()`：（以前に追加された）「次の値」`y` を `Queue` から削除し、`y` を返す

`remove()` は引数を取らない。`Queue` では、さまざまな**取り出し規則**に従って削除する要素が決まる。代表的な取り出し規則としては、FIFO、優先度付き、LIFO、といったものがある。

図 1.1 に FIFO **キュー** を示す。FIFO は first-in-first-out（先入れ先出し）を意味し、追加したのと同じ順番で要素を削除する。これはコンビニのレジに並ぶ列と同じように動作する。最も一般的な `Queue` なので、FIFO を付けずに単に「キュー」といえば、ふつうはこのデータ構造のことを指す。FIFO キューにおける `add(x)`、`remove()` を、それぞれ `enqueue(x)`、`dequeue()` と呼

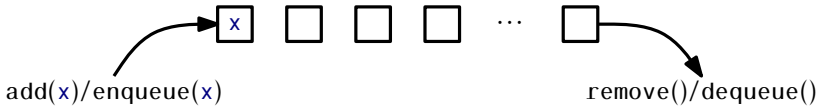


図 1.1: FIFO キュー

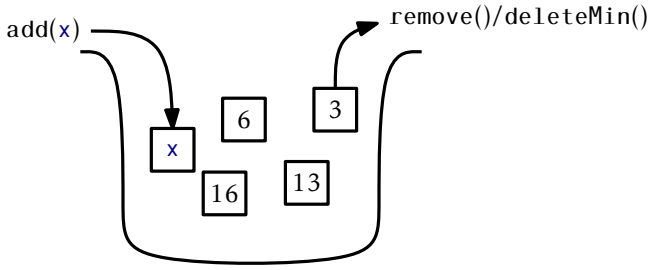


図 1.2: 優先度付きキュー

ぶ流儀の教科書もある。

図 1.2 に**優先度付きキュー**（priority queue）を示す。優先度付きキューでは、Queue から要素を削除するとき、最小のものを削除する。同じ優先度を持つ要素が複数あるときは、そのうちのどれを削除してもよい。優先度付きキューの動作は、病院の救急室で重症患者を優先的に治療する場面に似ている。患者が到着したらまず症状の深刻さを見定め、待合室で待機してもらい、医師の手が空いたら最も重篤な患者から治療するという具合だ。優先度付きキューにおける remove() 操作を deleteMin() と呼ぶ流儀の教科書もある。

キューに対する取り出し規則でもうひとつよく使うのは、図 1.3 に示す LIFO（last-in-first-out、後入れ先出し）だ。この LIFO **キュー**では、最後に追加された要素が次に削除される。LIFO キューの動作は、皿を積んだ状態として視覚化できる。積み上げられた皿を 1 つずつ取るとき、皿は上から順に持っていく。この構造はとてもよく見かけるので、Stack（スタック）という特別な名前が付いている。Stack と呼ぶ場合は、add(x) と remove() のことを、それぞれ push(x) および pop() と呼ぶ。これにより LIFO と FIFO の取り出し規則を区別できる。

FIFO キューと LIFO キュー（スタック）を一般化した Deque というインターフェースもある。Deque は双方向キューと呼ばれ、先頭と末尾を持った要素の列を表しており、先頭または末尾に要素を追加できる。Deque における操作には、addFirst(x)、removeFirst()、addLast(x)、 removeLast()

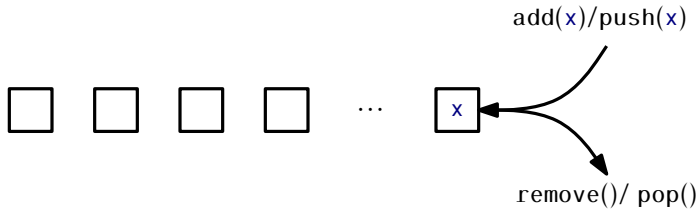


図 1.3: LIFO キュー（スタック）

というわかりやすい名前が付いている。addFirst() および removeFirst() だけを使ってスタックを実装できることは覚えておくとよいだろう。一方、addLast(x) および removeFirst() だけを使えば FIFO キューを実装できる。

1.2.2 List インターフェース：線形シーケンス

この本には Queue（FIFO キュー）や Stack（LIFO キュー）、Deque といったインターフェースの話はあまり出てこない。なぜなら、これらのインターフェースは List インターフェースとしてまとめられるからだ。図 1.4 に List インターフェースを示す。List インターフェースは、値の列 $x_0,\ldots,x_{n-1}$ と、その列に対する以下のような操作からなる。

1. size(): リストの長さ n を返す
2. get(i): x_i の値を返す
3. set(i,x): x_i の値を x にする
4. add(i,x): x を i 番め [*2] として追加し、$x_i,\ldots,x_{n-1}$ を後ろにずらす。
   すなわち、$j \in \{i,\ldots,n-1\}$ について $x_{j+1} = x_j$ とし、n をひとつ増やし、$x_i = x$ とする
5. remove(i): x_i を削除し、$x_{i+1},\ldots,x_{n-1}$ を前にずらす。
   すなわち、$j \in \{i,\ldots,n-2\}$ について $x_j = x_{j+1}$ とし、n をひとつ減らす

これらの操作を使って Deque インターフェースを実装できる。

$$\texttt{addFirst}(x) \Rightarrow \texttt{add}(0,x)$$
$$\texttt{removeFirst}() \Rightarrow \texttt{remove}(0)$$

[*2] コンピュータサイエンスでは序数を 0 から始めることがある。例えば、ここで配列の i 番めの要素とは、先頭から数えて $i+1$ 個めの要素のことである。

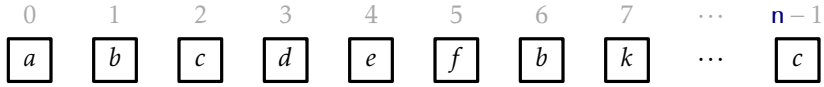


図 1.4: List は 0,1,2,...,n − 1 で添字づけられた列を表現する。この List で get(2) を実行すると値 c が返ってくる

$$\texttt{addLast}(\texttt{x}) \Rightarrow \texttt{add}(\texttt{size}(), \texttt{x})$$
$$\texttt{removeLast}() \Rightarrow \texttt{remove}(\texttt{size}() - 1)$$

以降の章では、Queue（FIFO キュー）、Stack（LIFO キュー）、Deque の各インターフェースについての話はほぼ出てこない。しかし、Stack と Deque という用語を「List インターフェースを実装したデータ構造」の名前として後の章で使うことがある。その場合は、Stack と Deque という名前で呼ぶデータ構造を使うことで、それぞれ Stack と Deque のインターフェースを非常に効率良く実装できるという事実を強調している。例えば、ArrayDeque は List インターフェースの実装であると同時に Deque の実装でもあり、Deque の操作をいずれも定数時間で実行できる [*3]。

1.2.3 USet インターフェース：順序付けられていない要素の集まり

USet インターフェースは、重複がなく順序付けられていない要素の集まりを表現する（USet の U は unordered の意味）。USet インターフェースは数学における**集合（set）**のようなものだ。USet には、n 個の**互いに相異なる**要素が含まれる。つまり、同じ要素が複数入っていることはない。また、USet では要素の並び順は決まっていない。USet には以下の操作を実行できる。

1. size()：集合の要素数 n を返す
2. add(x)：要素 x が集合に入っていなければ集合に追加する。
   x = y を満たす集合の要素 y が存在しないなら、集合に x を加える。x が集合に追加されたら true を返し、そうでなければ false を返す
3. remove(x)：集合から x を削除する。
   x = y を満たす集合の要素 y を探し、集合から取り除く。そのような要素が見つかれば y を、見つからなければ null[*4] を返す

[*3] 実行時間についてはこの章の後半で説明する。「定数時間で実行できる」とは、要素がいくつあっても一定の時間で実行できるということであり、非常に効率が良いことを表す。
[*4] 訳注：null とは何もないことを示す記号である。

4. find(x)：集合に x が入っていればそれを見つける。
   x = y を満たす集合の要素 y を見つける。そのような要素が見つかれば y を、見つからなければ null を返す

上の定義で、探したい x と、見つかる（かもしれない）要素 y とを、わざわざ区別する必要はないように感じるかもしれない。これらを区別する理由は、別のもの（オブジェクト）である x と y とを、何らかの基準で等しいと判定したい場合があるからだ。そのような判定ができると、キーを値に対応付けるインターフェースを実装するのに都合がいい。そうしたインターフェースは**辞書（dictionary）**や**マップ（map）**と呼ばれる。

辞書（マップ）を作るために、まずは Pair という、**キー**と**値**が対になったオブジェクトを作る。2 つの Pair は、キーが等しければ（その値が等しいかどうかにかかわらず）等しいとみなす。Pair である (k,v) を USet に入れてから、x = (k,null) として find(x) を実行すると、y = (k,v) が返ってくる。すなわち、キー k だけから値 v が手に入る。

1.2.4 SSet インターフェース：ソートされた要素の集まり

SSet インターフェースは順序付けされた要素の集まりを表現する（SSet の S は sorted の意味）。SSet には全順序集合の要素が入る。全順序集合とは、任意の 2 つの要素 x と y について大小を比較できるような集合をいう。本書のサンプルコードでは、以下のように定義される compare(x,y) メソッドで比較を行うものとする。

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

SSet は、USet とまったく同じセマンティクスを持つ操作 size()、add(x)、remove(x) をサポートする。USet と SSet の違いは find(x) にある。

4. find(x): 順序付けられた集合から x の位置を特定する。
   すなわち y ≥ x を満たす最小の要素 y を見つける。もしそのような y が存在すればそれを返し、存在しないなら null を返す

SSet の find(x) は**後継探索（successor search）**と呼ばれることがある。x に等しい要素がなくても意味のある結果を返すという点で、USet の find(x)

とは異なる。

USet、SSet における find(x) の区別は、重要だが見落とされることが多い。SSet は、USet より機能が多いが、それだけ実装が複雑で実行時間が長くなりがちだ。例えば、この本で述べる SSet の find(x) の実装は、いずれも集合に含まれる要素数の対数オーダーの時間がかかる。一方、5 章の ChainedHashTable による USet の実装では、find(x) の実行時間の期待値は定数オーダーである。USet にはない SSet の機能が必要でない限り、SSet ではなく USet を使うほうがよいだろう。

1.3 数学的背景

この節では本書で使う数学の記法や基礎知識を復習する。例えば対数やビッグオー記法、確率論などについて説明する。知っておいてほしい項目をまとめるに留め、丁寧な手ほどきはしない。背景知識が足りないと感じた読者はコンピュータサイエンスで使う数学の良い（無料の）教科書を読んでほしい。必要に応じて適切な箇所を読み、練習問題を解いてみるとよいだろう [50]。

1.3.1 指数と対数

b^x と書いて b の x 乗を表す。x が正の整数なら、b にそれ自身を $x-1$ 回掛けた値になる。

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

x が負の整数なら、$b^x = 1/b^{-x}$ である。$x = 0$ なら、$b^x = 1$ である。b が整数でないときも、指数関数 e^x を使って冪乗を定義できる（e については後述する）。この e^x の定義は、指数級数による。こういう話をもっと知りたい人は微分積分学の教科書を読んでほしい。

この本では、$\log_b k$ と書いて **b を底とする対数**を表す。これは次の式を満たす x として一意に決まる。

$$b^x = k$$

底が 2 の対数を**二進対数**（binary logarithm）という。この本に出てくる対数のほとんどは二進対数なので、底に何も書かずに $\log k$ とある場合は、$\log_2 k$ の省略記法とする。

対数の大雑把だがわかりやすいイメージを紹介しよう。$\log_b k$ とは、k を何

回 b で割ると 1 以下になるかを表す数だと考えればよい。例えば、1 回の比較で答えの候補を半分に絞り、最終的に答えの候補が 1 つに絞られるまでこれを繰り返すとして、最終的に何回の比較が必要になるかを見積もりたいとする。1 回の比較で候補の数を 2 で割ることになるので、最初に $n+1$ 個の答えの候補があるなら、比較の回数は $\lceil \log_2(n+1) \rceil$ 以下だ（なお、このような手法を二分探索という）[*5]。

次のように定義される**オイラーの定数**（Euler's constant）e を底とする対数もよく使う [*6]。そこで、$\log_e k$ のことを $\ln k$ と書き、**自然対数**（natural logarithm）と呼ぶ。

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

自然対数は、次の一般的な積分の値が e になることから、よく登場する。

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

対数に関してよく使う操作は 2 つある。1 つめは冪指数にある対数の除去だ。

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

もう 1 つは底の変換操作だ。

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

これら 2 つの操作を使うと、例えば自然対数と二進対数とを比較できる。

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

1.3.2 階乗

この本には**階乗関数**（factorials）を使う場面がいくつかある。n が非負整数のとき、n の階乗 $n!$ は次のように定義される。

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

[*5] 訳注：x を実数とするとき、$\lceil x \rceil$ は x 以上の最小の整数を表す。$\lfloor x \rfloor$ は x 以下の最大の整数である。

[*6] 訳注：e はネイピア数とも呼ぶ。

$n!$ は、相異なる n 要素の置換の総数である。つまり、n 個の要素を並べ変えたときの順列の総数が階乗になる。なお、$n = 0$ のとき、$0!$ は 1 と定義される。

$n!$ の大きさは**スターリングの近似**（Stirling's Approximation）を使って見積もれる [*7]。

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

ここで $\alpha(n)$ は次の条件を満たす。

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

スターリングの近似を使って $\ln(n!)$ の近似値も計算できる。

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

（実際、$\ln(n!) = \ln 1 + \ln 2 + \cdots + \ln n$ を $\int_1^n \ln n \,\mathrm{d}n = n\ln n - n + 1$ で近似するというのが、スターリングの近似の簡単な証明方法でもある。）

階乗関数に関連して、ここで**二項係数**（bonimial coefficients）について説明する。n を非負整数、k を $\{0,\ldots,n\}$ の要素とするとき、二項係数 $\binom{n}{k}$ は次のように定義される。

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

二項係数 $\binom{n}{k}$ は、大きさ n の集合における大きさ k の部分集合の個数である。言い換えると、集合 $\{1,\ldots,n\}$ から相異なる k 個の整数を取り出すときの場合の数を表す値と解釈できる。

1.3.3 漸近記法

データ構造を分析するときは、さまざまな操作の実行時間について考察したい。しかし、正確な実行時間はコンピュータによって異なる。同じコンピュータ上でさえ実行のたびに異なるだろう。この本で操作の実行時間といったら、操作に際してコンピュータが実行する命令の数とする。この数を正確に計算するのは、単純なコードであっても困難な場合がある。そのため、正確な実行時間を求めるのではなく、**漸近記法**（asymptotic notation）あるいは**ビッグオー記法**（big-Oh notation）と呼ばれる方法で実行時間を見積もる。この方

[*7] 訳注：以下、スターリングの近似に関する議論は、初学者は飛ばしてもよいと思われる。

法では、ある関数 $f(n)$ について、次のように定義される関数の集合 $O(f(n))$ を考える。

$$O(f(n)) = \left\{ \begin{array}{l} g(n) : \text{ある } c > 0 \text{ と } n_0 \text{ が存在し、} \\ \quad \text{任意の } n \geq n_0 \text{ について } g(n) \leq c \cdot f(n) \text{ を満たす} \end{array} \right\}$$

イメージとしては、n が十分に大きいとき（つまりグラフの右のほうを見たとき）に $c \cdot f(n)$ のほうが上にくるような関数 $g(n)$ を集めたものが集合 $O(f(n))$ だ。

漸近記法は、関数を単純な形にするのに使う。例えば、$5n\log n + 8n - 200$ の代わりに $O(n\log n)$ と書ける。これは次のように証明できる。

$$\begin{aligned} 5n\log n + 8n - 200 &\leq 5n\log n + 8n \\ &\leq 5n\log n + 8n\log n \quad n \geq 2 \text{ のとき（このとき } \log n \geq 1\text{）} \\ &\leq 13n\log n \end{aligned}$$

$c = 13$ および $n_0 = 2$ とすれば、関数 $f(n) = 5n\log n + 8n - 200$ が集合 $O(n\log n)$ に含まれることがわかる。

漸近記法の便利な性質をいくつか挙げる。まずは、任意の定数 $c_1 < c_2$ について以下が成り立つ。

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

続いて、任意の定数 $a, b, c > 0$ について以下が成り立つ。

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

これらの包含関係は、それぞれに正の値を掛けても保たれる。例えば n を掛けると次のようになる。

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

一般的な慣習に従って、本書でもビッグオー記法を濫用する。すなわち、$f_1(n) = O(f(n))$ と書いて $f_1(n) \in O(f(n))$ であることを表す。そして、「この操作の実行時間は集合 $O(f(n))$ に**含まれる**」ことを、単に「この操作の実行時間は $O(f(n))$ だ」と言う。これらの表現を認めると、冗長な記述が不要になるし、一連の等式で漸近記法を使えるようになる。

ビッグオー記法を濫用することで、例えば次のような不思議な書き方ができる。

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

これは正確に書くとこうなる。

$$T(n) \leq 2\log n + [O(1) \text{ のある要素}]$$

$O(1)$ という記法には別の問題もある。この記法には変数が入ってないので、どの変数が大きくなるのかわからないのだ。これは文脈から読み取る必要がある。上の例では、方程式の中に変数は n しかないので、$T(n) = 2\log n + O(f(n))$ のうちで $f(n) = 1$ の場合であると読み取ることになる。

ビッグオー記法は、新しい記法でもコンピュータサイエンス独自の記法でもない。1894 年には数学者の Paul Bachmann がこの記法を使っていた。その後しばらくして、コンピュータサイエンスにおいてアルゴリズムの実行時間を論ずる際に、この記法が非常に便利なことがわかったのだ。次のコードを考えてみよう。

Simple

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

この関数を 1 回実行すると以下の処理が行われる。

- 代入 1 回（`int i = 0`）
- 比較 `n` + 1 回（`i < n`）
- インクリメント `n` 回（`i++`）
- 配列のオフセット計算 `n` 回（`a[i]`）
- 間接代入 `n` 回（`a[i] = i`）

よって実行時間は以下のようになる。

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

a、b、c、d、e はプログラムを実行するマシンに依存する定数で、それぞれ代入、比較、インクリメント、配列のオフセット計算、間接代入にかかる実行時間を表す。たった 2 行のコードについて実行時間を表すのに、こうも複雑な式がいるようでは、さらに複雑なコードやアルゴリズムは到底扱えないだろう。ビッグオー記法を使えば、実行時間を次のように簡潔に表せる。

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

この式は、簡潔な表現にもかかわらず、最初の式と同じくらいの内容を表している。正確な実行時間は定数 a、b、c、d、e に依存しており、これらの値がすべて判明しないと知りようがないからだ。がんばって値を実測してみても、得られる結論はそのマシンでしか有効でない。

ビッグオー記法を使えば、より抽象的な分析ができ、より複雑な関数も扱える。2 つのアルゴリズムの実行時間がビッグオー記法で同じなら、どちらが速いか優劣はつけられない。一方のアルゴリズムが速いマシンもあれば、もう一方のアルゴリズムが速いマシンもあるだろう。しかし、2 つのアルゴリズムの実行時間がビッグオー記法で異なるなら、**n が十分大きい場合**、実行時間が小さいアルゴリズムのほうがどのようなマシンにおいても速いといえる。

ビッグオー記法を使って 2 つの異なる関数を比べる例を図 1.5 に示す。これは $f_1(n) = 15n$ と $f_2(n) = 2n\log n$ のグラフである。$f_1(n)$ は複雑な線形時間アルゴリズムの実行時間を表し、$f_2(n)$ は分割統治に基づくシンプルなアルゴリズムの実行時間を表している。これを見ると、n が小さいうちは $f_1(n)$ のほうが $f_2(n)$ より大きいが、n が大きくなると大小関係が逆転することがわかる。つまり、n が十分大きいなら、実行時間が $f_1(n)$ であるアルゴリズムのほうが圧倒的に性能がよい。ビッグオー記法の式 $O(n) \subset O(n\log n)$ は、この事実を示している。

多変数関数に対して漸近記法を使うこともある。標準的な定義はないようだが、この本では次の定義を用いる。

$$O(f(n_1,\ldots,n_k)) = \left\{ \begin{array}{l} g(n_1,\ldots,n_k) : \text{ある } c>0 \text{ と } z \text{ が存在し、} \\ \quad g(n_1,\ldots,n_k) \geq z \text{ を満たす任意の } n_1,\ldots,n_k \text{ について、} \\ \quad g(n_1,\ldots,n_k) \leq c\cdot f(n_1,\ldots,n_k) \text{ が成り立つ} \end{array} \right\}$$

興味があるのは引数 $n_1,\ldots,n_k$ によって g が大きくなるときの状況であり、その状況はこの定義で把握できる。$f(n)$ が n に関する増加関数なら、この定義は 1 変数の場合の $O(f(n))$ の定義とも合致する。この本ではこの程度の考察で十分だが、教科書によっては多変数関数と漸近記法に別の定義を与えている可能性もあるので注意してほしい。

1.3.4 ランダム性と確率

この本で扱うデータ構造には**乱択化（randomization）**を利用するものがある。乱択化では、格納されているデータや実行する操作に加えて、サイコロの出目も踏まえて実際の処理を決める。そのため、同じことをしても実行時間が毎回同じとは限らない。このようなデータ構造を分析するときは**期待実行時**

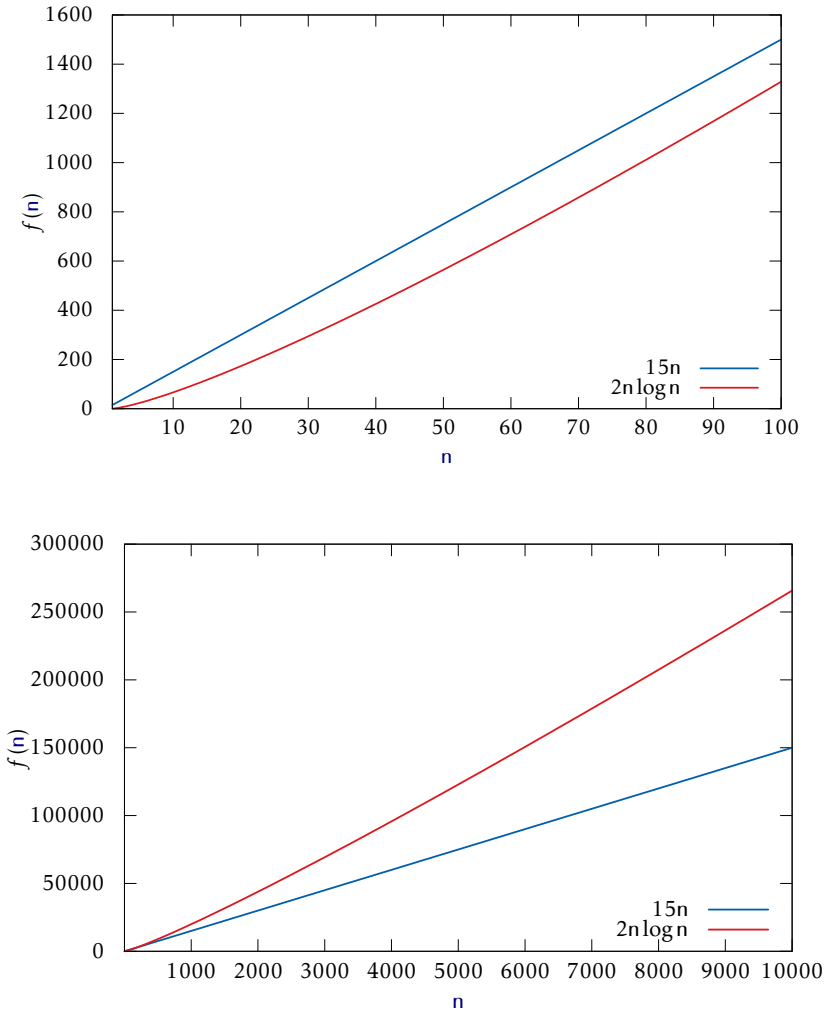


図 1.5: 15n と 2n log n の比較

間（expected running time）を考えるのがよい。

乱択化を利用するデータ構造における操作の実行時間は形式的には確率変数であり、その**期待値**（expected value）を知りたい。可算個の事象全体を U とし、その上で定義された離散確率変数を X とすると、X の期待値 $E[X]$ は次のように定義される。

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

ここで、$\Pr\{\mathcal{E}\}$ は事象 $\mathcal{E}$ の発生確率とする。この本におけるすべての例では、乱択化されたデータ構造におけるランダムな選択についてのみ、これらの確率が関係している。つまり、データ構造に入ってくるデータや実行される操作列がランダムであるような状況は想定していない。

期待値の最も重要な性質のひとつは**期待値の線形性**（linearity of expectation）である。任意の 2 つの確率変数 X と Y について次の式が成り立つ。

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

より一般的には、任意の確率変数 $X_1, \ldots, X_k$ について次の関係が成り立つ。

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

期待値の線形性によって、（上の式の左辺のように）複雑な確率変数の期待値を、（右辺のような）より単純な確率変数の和に分解できる。

便利でよく使う手法に、**インジケータ確率変数**（indicator random variable）と呼ばれる二値の変数を定義するというものがある。この二値変数は、何かを数えるときに役立つ。例を見るとよくわかるだろう。表と裏が等しい確率で出るコインを k 回投げたとき、表が出る回数の期待値を知りたいとする。直観的な答えは $k/2$ だが、これを期待値の定義を使って証明すると次のようになる。

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

$$= k/2$$

この計算をするには、$\Pr\{X = i\} = \binom{k}{i}/2^k$ および二項係数の性質 $i\binom{k}{i} = k\binom{k-1}{i-1}$ や $\sum_{i=0}^{k}\binom{k}{i} = 2^k$ を知っている必要がある。

インジケータ確率変数と期待値の線形性を使えば、この期待値をはるかに簡単に求められる。$\{1,\ldots,k\}$ の各 i に対し、以下のインジケータ確率変数を定義する。

$$I_i = \begin{cases} 1 & i \text{ 番めのコイントスの結果が表のとき} \\ 0 & \text{そうでないとき} \end{cases}$$

そして、I_i の期待値を計算する。

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

ここで、$X = \sum_{i=1}^{k} I_i$ なので次のように所望の値が得られる。

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

少し長い計算ではあるが、不思議な変数はどこにも出てこないし、込み入った確率の計算もない。また、各コイントスでは 1/2 の確率で表が出るので、試行回数の半分くらいは表が出るだろうという直観にも合致する。

1.4 計算モデル

この本では、データ構造における操作の実行時間を理論的に分析する。その正確な分析には、計算についての数学的なモデルが必要だ。そのような数学的モデルとして、**w ビットのワード** RAM（word-RAM）を使うことにする。ここでいう RAM は、Random Access Machine の頭字語である。w ビットのワード RAM モデルでは、それぞれに w ビットのワードを格納できるセルを

集めたランダムアクセスメモリを使える。これはすなわち、メモリの各セルで w 桁の 2 進数を表せる、つまり集合 $\{0,\ldots,2^w-1\}$ のうちのいずれかひとつをメモリの各セルで表せるということである。

ワード RAM モデルでは、ワードに対する基本的な操作に一定の時間を要する。ここでいう基本的な操作とは、算術演算（+、−、*、/、%）や比較（<、>、=、≤、≥）、ビット単位の論理演算（ビット単位の論理積 AND や論理和 OR、排他的論理和 XOR）を指す。

ランダムアクセスメモリでは、どのセルも一定の時間で読み書きできる。コンピュータのメモリはメモリ管理システムによって管理されており、このメモリ管理システムを通じて必要なサイズのメモリブロックの割り当てや解除ができる。サイズ k のメモリブロックの割り当てには $O(k)$ の時間がかかり、新しく割り当てられたメモリブロックへの参照（ポインタ）が返される。この参照は 1 ワードに収まるビットで表現できるものとする。

ワード幅 w は、このモデルにとって重要なパラメータである。この本では、w について、データ構造に格納されうる要素数が n であれば $w > \log n$ ということしか仮定しない。これは控えめな仮定である。なぜなら、せめてこれが成り立たないと、データ構造の要素数を 1 ワードで表すことすらできないからである。

メモリ使用量はワード単位で測るので、データ構造で使うワード数がそのままデータ構造のメモリ使用量になる。この本のデータ構造は、すべて型 `T` の値を格納し、`T` 型の要素は 1 ワードのメモリで表現できると仮定する。

w ビットのワード RAM モデルは、$w = 32$ または $w = 64$ とすると、現代のデスクトップコンピュータ環境によく似ている。すなわち、この本に載っているデータ構造は、いずれも一般的なコンピュータ上で動作するように実装できる。

1.5 正しさ、時間計算量、空間計算量

データ構造の性能を考えるとき重要な項目が 3 つある。

正しさ： データ構造はそのインターフェースを正しく実装しなければならない。

時間計算量（time complexity）： データ構造における操作の実行時間は短いほどよい。

空間計算量（space complexity）： データ構造のメモリ使用量は小さいほど

よい。

この本は入門書なので、上記のうち「正しさ」については大前提とする。つまり、不正確な出力が得られるデータ構造や、適切に更新されないデータ構造については考えない。一方で、メモリ使用量を小さく抑えるための工夫を施したデータ構造については紹介していく。通常、そうした工夫によって操作の（漸近的な）実行時間が変わることはないが、実際のデータ構造の動作が少し遅くなる可能性はある。

データ構造に関して実行時間を議論するときは、次の三種類のいずれかを保証するという話になることが多い。

最悪実行時間（worst-case running time）： 実行時間に対する保証の中で、最も強力なもの。あるデータ構造の操作について最悪実行時間が $f(n)$ であるといったら、そのような操作の実行時間が $f(n)$ より長くなることは**決して**ない。

償却実行時間（amortized running time）： 償却実行時間が $f(n)$ であるとは、典型的な操作にかかるコストが $f(n)$ を超えないことを意味する。より正確には、m 個の操作にかかる実行時間を合計しても、$mf(n)$ を超えないことを意味する。いくつかの操作には $f(n)$ より長い時間がかかるかもしれないが、操作の列全体として考えれば、1 つあたりの実行時間は $f(n)$ という意味だ。

期待実行時間（expected running time）： 期待実行時間が $f(n)$ であるとは、実行時間が確率変数（1.3.4 節を参照）であり、その確率変数の期待値が $f(n)$ であることを意味する。この期待値を計算する際に考えるランダム性は、そのデータ構造内で起こる選択におけるランダム性である。

最悪実行時間、償却実行時間、期待実行時間の違いを理解するには、お金の例で考えてみるとよい。家を購入する費用について考えてみよう。

最悪コストと償却コスト 家の価格が 12 万ドルだとする。毎月 1200 ドルを 120 ヶ月（10 年）にわたって支払うという住宅ローンを組むことで、この家が手に入るとしよう。この場合、月額費用は最悪でも月 1200 ドルだ。

十分な現金を持っていれば、12 万ドルの一括払いでこの家を買うこともできる。その場合、この家の購入代金を 10 年で償却すると考えて月額費用を計算すれば、以下のようになる。

$$120\,000 \text{ ドル} / 120 \text{ ヶ月} = \text{毎月} \$1\,000$$

これはローンの場合に支払う月額 1200 ドルよりだいぶ少ない。

最悪コストと期待コスト　次に、12 万ドルの家に火災保険をかけることを考えてみよう。保険会社が何十万件もの事例を調べた結果、大多数の家では火事を起こさず、いくつかの家では煙による被害程度で済むボヤを起こし、ごく少数の家では全焼被害に至ることがわかった。保険会社は、この情報に基づいて 12 万ドルの家における火災被害額の期待値を月額 10 ドル相当と判断し、自社の儲けを考慮して、火災保険の掛金を月額 15 ドルに設定した。あなたは保険会社に勤めていて、それらの数字を知っていたと仮定しよう。

決断のときだ。最悪コストが月額 15 ドルのこの火災保険に入るべきだろうか？それとも、期待コストである月額 10 ドルを自分で積み立てることにして、月額 5 ドルを節約するという賭けに出るべきだろうか？明らかに、自分で積み立てるほうが安上がりになると期待できるが、いざという場合のコストがはるかに高くなる可能性を考慮しなければならない。すなわち、低い確率ではあるが、家が全焼して実際のコストが 12 万ドルになる可能性がある。

この 2 つの例からわかるように、どちらのコストを優先するかは場合によって変わる [*8] 。償却実行時間と期待実行時間は、最悪実行時間より小さいことが多い。最悪実行時間の長さに目をつむり、償却実行時間や期待実行時間が小さいからと妥協すれば、はるかに単純なデータ構造を採用できる場合がよくあるのだ。

1.6　コードサンプル

この本のサンプルコードは C++ で書いた。ただし、C++ に親しみのない人でも読めるよう、簡潔に書いたつもりだ。例えば、`public` や `private` は出てこない。オブジェクト指向を前面に押し出すこともない。

B、C、C++、C#、Objective-C、D、Java、JavaScript といった ALGOL 系の言語を書いたことのある人なら、本書のコードの意味はわかるだろう。完全な実装に興味がある読者は、この本に付属する C++ のソースコードを見てほしい。

[*8] 訳注：火災保険の例では、いざという場合のコストが大幅に低くなること、月額の差が 5 ドルと比較的少額であることから、火災保険を選ぶ人が多いかもしれない。しかし驚くべきことに**データ構造の世界では、最悪実行時間よりも償却実行時間や期待実行時間が低いことを優先する**ことも多い。いざという場合の損害が家屋の全焼ほど大きくなく、また、その確率も家屋の全焼よりはるかに小さく制御できることが多いからだろう。

この本には、数学的な実行時間の解析と、対象のアルゴリズムを実装したC++ のコードが両方とも含まれている。そのため、ソースコードと数式とで同じ変数が出てくる。このような変数は同じ書式で書く [*9]。特によく出てくるのは、変数 n である。n は常にデータ構造に格納されている要素の個数を表す。

1.7 データ構造の一覧

表 1.1 と表 1.2 に、この本で扱うデータ構造の性能を要約する。これらは、1.2 節で説明した List、USet、SSet を実装する。図 1.6 には、この本の各章の依存関係を示す。破線の矢印は、章のごく一部の内容や結果のみに依存することを示す。

1.8 ディスカッションと練習問題

1.2 節で説明した List、USet、SSet の各インターフェースは、Java Collections Framework [54] の影響を受けている。これらは、Java Collections Framework における List、Set、Map、SortedSet、SortedMap を単純化したものだと考えられる。

この章で扱った漸近記法、対数、階乗、スターリングの近似、確率論の基礎などは、Leyman, Leighton, and Meyer による素晴らしい（そして無料の）書籍 [50] で扱われている。微積分のわかりやすい無料の教科書としては、Thompson による古典的な教科書 [71] がある。この本には指数や対数の形式的な定義が書かれている。

確率論の基礎については、特にコンピュータサイエンスに関連するものとして、Ross の教科書 [63] がおすすめである。漸近記法や確率論については、Graham, Knuth, and Patashnik の教科書 [37] も参考になるだろう。

問 **1.1.** この練習問題は、読者が問題に対する正しいデータ構造を選ぶ練習をするためにある。利用可能な実装やインターフェース（Java ならば Java Collections Framework、C++ ならば Standard Template Library）があれば、それを使って解いてみてほしい。

[*9] 訳注：ソースコード中に現れる変数名は、英語のままにしている。

表 1.1: List、USet の実装の要約

List の実装			
	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

USet の実装			
	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] **償却**実行時間を表す

[E] **期待**実行時間を表す

以下の問題は、テキストの入力を 1 行ずつ読み、各行で適切なデータ構造の操作を実行することで解いてほしい。ファイルが百万行であっても数秒以内に処理できる程度には効率的な実装にすること。

1. 入力を 1 行ずつ読み、その逆順で出力せよ。すなわち、最後の入力行を最初に書き出し、最後から 2 行めを 2 番めに書き出す、というように出力せよ。
2. 先頭から 50 行の入力を読み、それを逆順で出力せよ。その後、続く 50 行を読み、それを逆順で出力せよ。これを読み取る行がなくなるまで繰

表 1.2: SSet、優先度付き Queue の実装の要約

SSet の実装			
	`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`^{Int}	$O(w)$	$O(w)$	§ 13.1
`XFastTrie`^{Int}	$O(\log w)^{A,E}$	$O(w)^{A,E}$	§ 13.2
`YFastTrie`^{Int}	$O(\log w)^{A,E}$	$O(\log w)^{A,E}$	§ 13.3

（優先度付き）Queue の実装			
	`findMin()`	`add(x)`/`remove()`	
`BinaryHeap`	$O(1)$	$O(\log n)^{A}$	§ 10.1
`MeldableHeap`	$O(1)$	$O(\log n)^{E}$	§ 10.2

[A] **償却**実行時間を表す

[E] **期待**実行時間を表す

[Int] このデータ構造は `w` ビットで表現できる整数のみを格納できる

り返し、最後に残った行（50 行未満かもしれない）もやはり逆順で出力せよ。
つまり、出力は 50 行めから始まり、49 行め、48 行め、…、1 行めが続く。その次は、100 行め、99 行め、…、51 行めが続く。
なお、プログラムの実行中に 50 行より多くの行を保持してはならない。

3. 入力を 1 行ずつ読み取り、42 行め以降で空行を見つけたら、その 42 行前の行を出力せよ。例えば、242 行めが空行であれば 200 行めを出

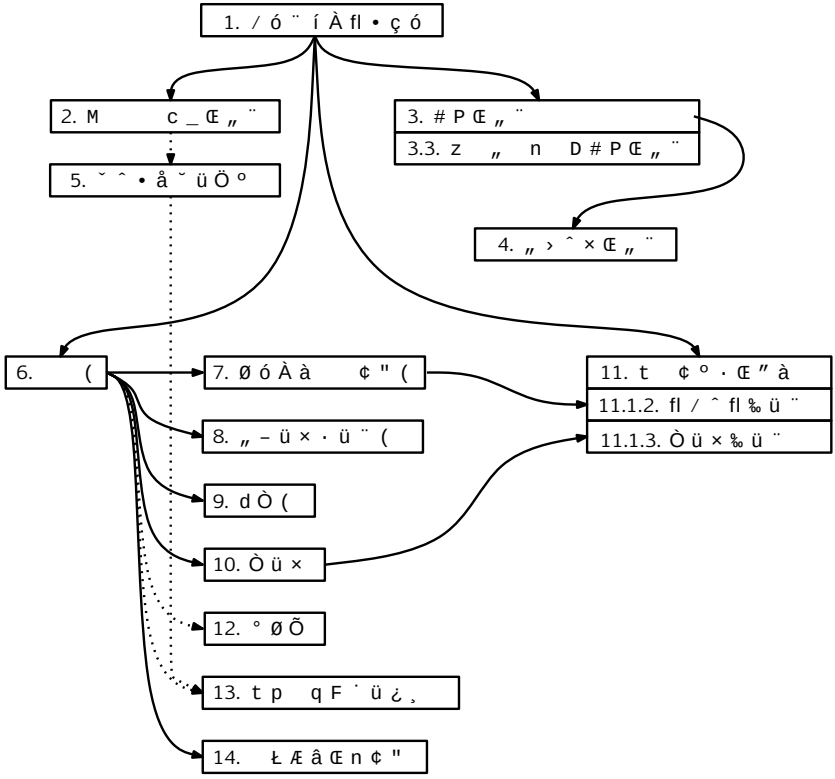


図 1.6: この本の内容の依存関係

力せよ。なお、プログラムの実行中に 43 行以上の行を保持してはならない。

4. 入力を 1 行ずつ読み取り、それまでに読み込んだことがある行と重複しない行を見つけたら出力せよ。重複が多いファイルを読む場合でも、重複なく行を保持するのに必要なメモリより多くのメモリを使わないように注意せよ。
5. 入力を 1 行ずつ読み取り、それまでに読み込んだことがある行と同じなら出力せよ（最終的には、ある行が入力ファイルに初めて現れた箇所をそれぞれ除いたものが出力になる）。重複が多いファイルを読む場合でも、重複なく行を保持するのに必要なメモリより多くのメモリを使わないように注意せよ。

6. 入力をすべて読み取り、短い順に並べ替えて出力せよ。同じ長さの行があるときは、それらの行は辞書順に並べるものとする。また、重複する行は一度だけ出力するものとする。
7. 直前の問題で、重複する行については現れた回数だけ出力するように変更せよ。
8. 入力をすべて読み取り、すべての偶数番めの行を出力したあとに、すべての奇数番めの行を出力せよ（最初の行を 0 行めと数える）。
9. 入力をすべて読み取り、ランダムに並べ替えて出力せよ。どの行の内容も書き換えてはならない。また、入力より行が増えたり減ったりしてもいけない。

問 **1.2.** Dyck word とは、+1 と −1 からなる列で、先頭から任意の `k` 番めの値までの部分列（プレフィックス）の和がいずれも非負なものとする。例えば、+1,−1,+1,−1 は Dyck word だが、+1,−1,−1,+1 は +1 − 1 − 1 < 0 なので Dyck word ではない。Dyck word と、スタックの push(x) 操作および pop() 操作との関係を説明せよ。

問 **1.3.** **マッチした文字列**とは {, }, (,), [,] からなる列で、すべての括弧が適切に対応しているものとする。例えば、「{{()[]}}」はマッチした文字列だが、「{{()]}」は 2 つめの { に対応する括弧が] であるためマッチした文字列ではない。長さ `n` の文字列が与えられたとき、この文字列がマッチしているかどうかを $O(n)$ で判定するのにスタックをどう使えばよいかを説明せよ。

問 **1.4.** push(x) 操作と pop() 操作のみが可能なスタック `s` が与えられたとする。FIFO キュー `q` だけを使って `s` の要素を逆順にする方法を説明せよ。

問 **1.5.** USet を使って Bag を実装せよ。Bag は、USet によく似たインターフェースで、add(x) 操作、remove(x) 操作、find(x) 操作をサポートする。USet との違いは、Bag では重複する要素も格納する点である。Bag の find(x) 操作では、x に等しい要素が 1 つ以上含まれているとき、そのうちの 1 つを返す。さらに、Bag は findAll(x) 操作もサポートする。これは、Bag に含まれる x に等しいすべての要素のリストを返す操作である。

問 **1.6.** List インターフェース、USet インターフェース、SSet インターフェースを実装せよ。効率的な実装でなくてもよい。ここで実装するものは、後の章のより効率的な実装の正しさや性能をテストするのに役立つ（最も簡単なのは要素を配列に入れておく方法だ）。

問 **1.7.** 直前の問題の実装について、性能をアップする工夫として思いつくものをいくつか試みよ。実験してみて、List インターフェースの add(i,x) 操作と remove(i) 操作の性能がどう向上したかを考察せよ。どうすれば、USet インターフェースと SSet インターフェースの find(x) 操作の性能を向上できそうか考えてみよ。この問題は、インターフェースの効率的な実装がどれくらい難しいかを実感するためのものである。

第 2

配列を使ったリスト

この章では backing array と呼ばれる配列にデータを入れて、List インターフェースと Queue インターフェースを実装する方法を解説する [*1]。次の表に、この章で説明するデータ構造の操作にかかる実行時間を要約する。

	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	$O(1)$	$O(n-i)$

データを 1 つの配列に入れて動作するデータ構造には、一般に以下のような利点と欠点がある。

- 配列では任意の要素に一定の時間でアクセスできる。そのため、get(i) 操作と set(i,x) 操作を定数時間で実行できる
- 配列はそれほど動的ではない。リストの中ほどに要素を追加、削除するには、隙間を作ったり埋めたりするため、配列に含まれる多くの要素を移動させる必要がある。add(i,x) 操作と remove(i) 操作の実行時間が n と i に依存するのは、これが原因である
- 配列は伸び縮みしない。backing array のサイズより多くの要素をデータ構造に入れるには、新しい配列を割り当てて古い配列の要素をそちら

[*1] 訳注：訳者が知る限り、backing array には広く通用する訳語がない。意訳するならば、「裏でも要素が一並びになっている配列」となるだろう。頻出用語ではなく、単に配列（array）と読み替えても問題はないだろう。重要なのは、裏でも要素が一並びになっているので、この章で述べる利点と欠点が生じることである。

にコピーしなければならず、この操作のコストは大きい

3 つめは特に重要だ。上記の表の実行時間には、backing array の拡大と縮小にかかるコストが含まれていない。後述するように、注意深く設計すれば、backing array の拡大と縮小にかかるコストを加味しても**平均的な**実行時間にはほぼ影響しない。より正確に言うと、空のデータ構造から始めて、add(i,x) と remove(i) を m 回実行するとき、backing array を拡大、縮小するのにかかる時間の合計は $O(m)$ である。コストが大きい操作もあるが、m 個の操作にわたって均せば、1 つの操作あたりの償却コストは $O(1)$ なのだ。

この章、そしてこの本では、要素数を保持する配列が使えると便利なことが多い。C++ のふつうの配列は要素数を保持していないので、要素数を保持する配列のクラス array を定義する [*2]。このクラスは標準的な C++ の配列 a と整数 length により簡単に実装できる。

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

array の大きさは作成時に指定する。

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

配列の要素は添字により指定できる。

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

また、ある配列を他の配列に割り当てる操作は、ポインタの操作により定数時間で実行できる。

[*2] 訳注：この array クラスにおける = の実装では、右辺の配列を NULL にしている。つまり、他の array に代入した array は、その後使えなくなってしまうことに注意する。

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

2.1 ArrayStack：配列を使った高速なスタック操作

ArrayStack は、backing array を使った List インターフェースの実装だ。以降では、ArrayStack の実装に利用する backing array を配列 a と呼ぶ。リストの i 番めの要素は、a[i] に格納する。配列 a の大きさは、通常は厳密に必要な要素数より大きいので、実際に a に入っているリストの要素数は整数 n で表す。つまり、リストの要素は a[0],...,a[n−1] に格納されている。このとき常に a.length ≥ n である。

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

2.1.1 基本

get(i) や set(i,x) を使って ArrayStack の要素を読み書きする方法は簡単だ。必要に応じて境界チェック [*3] をしたあと、単に a[i] を返すか、a[i] を書

[*3] 訳注：get(i) や set(i,x) における境界チェック（bounds-checking）とは、添字 i が、最初の要素の添字である 0 以上であり、かつ、最後の要素の添字以下である、つまり a.length − 1 以下であると確認することである。

き換えるかすればよい。

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

ArrayStack に要素を追加、削除するための実装を図 2.1 に示す。add(i,x) では、まず a がすでに一杯かどうかを調べる。もしそうなら resize() を呼び出して a を大きくする。resize() の実装方法は後述する。いまのところ、resize() の直後には a.length > n となっている点だけ了解しておけばよい。あとは、x を入れるために a[i],...,a[n−1] を 1 つずつ右に移動させ、a[i] を x にして、n を 1 増やす。

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++;
}
```

resize() のコストを無視すれば、add(i,x) のコストは x を入れる場所を作るためにシフトする要素数に比例する。つまり、この操作の（resize() のコストを無視した）実行時間は、$O(n-i)$ である。

remove(i) も同様に実装できる。a[i+1],...,a[n−1] を左に 1 つシフトし、n の値を 1 つ小さくする（a[i] は書き換えられる前に控えておく）。そして、配列の長さに対して要素数が少なすぎないか、具体的には a.length ≥ 3n か

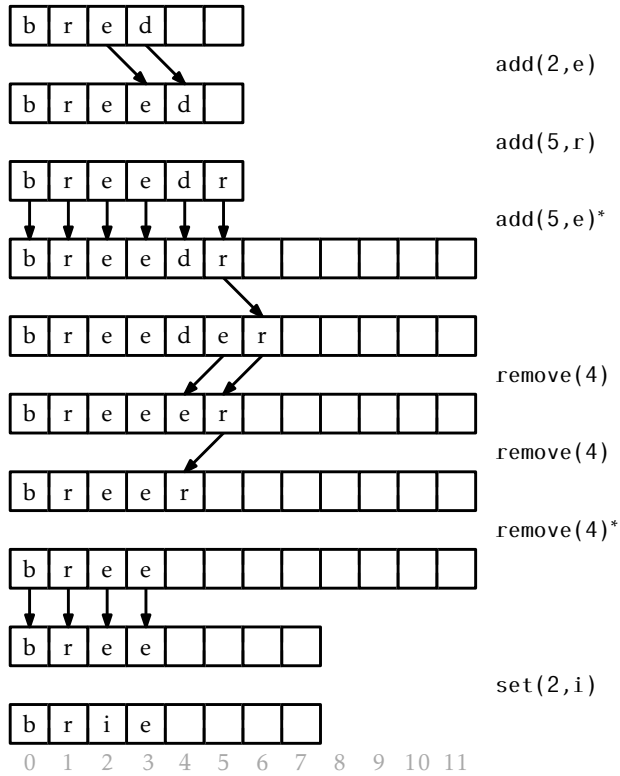


図 2.1: ArrayStack に対する add(i,x) と remove(i) の実行例。矢印は要素のコピーを表す。resize() を呼ぶ操作にはアスタリスクを付した

どうかを確認する。もしそうなら resize() を呼んで a を小さくする。

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;
```

```
}
```

resize() が呼ばれるかもしれないが、そのコストを無視すれば、remove(i) のコストはシフトする要素数に比例し、$O(n-i)$ である。

2.1.2 拡張と収縮

resize() の実装は単純だ。大きさ 2n の新しい配列 b を割り当て、n 個の a の要素を b の先頭の n 個としてコピーする。そして a を b に置き換える。よって、resize() の呼び出し後は a.length = 2n が成り立つ [*4]。

ArrayStack

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

resize() の実際のコストの計算も簡単だ。大きさ 2n の配列 b を割り当て、n 個の要素をコピーする。これには $O(n)$ の時間がかかる。

前節の実行時間分析では resize() のコストを無視していた。この節では**償却解析**（amortized analysis）と呼ばれる手法でこれを解決する。この手法は、個々の add(i,x) および remove(i) における resize() のコストを求めるわけではない。代わりに、add(i,x) と remove(i) からなる m 個の一連の操作の間に呼ばれる resize() の実行時間の合計を考える。特に、次の補題を示す。

補題 **2.1.** 空の *ArrayStack* が作られたあと、$m \geq 1$ 回の add(i,x) および remove(i) からなる操作の列が順に実行されるとき、resize() の実行時間は合計 $O(m)$ である。

証明. resize() が呼ばれるとき、その前の resize() の呼び出しから add および remove が実行された回数が $n/2-1$ 回以上であることを後半で示す。

[*4] 訳注：補題 2.1 の証明でも言及されているが、n = 0 かつ a.length = 1 のときに限ってこの式は成り立たないことがある。

このとき、resize() の i 回めの呼び出しの際の n を n_i、resize() の呼び出し回数を r とすれば、add(i,x) および remove(i) の呼び出し回数の合計は次の関係を満たす。

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

これを変形すると次の式が得られる。

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

$r \le m$ なので、resize() の呼び出しにかかる実行時間の合計は次のようになる。

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

あとは $(i-1)$ 回めの resize() から i 回めの resize() の間に、add(i,x) か remove(i) が呼ばれる回数の合計が $n_i/2-1$ 以上であることを示せばよい。

これは 2 つの場合に分けて考えられる。1 つめは、resize() が add(i,x) の中で呼ばれる場合で、これは backing array が一杯になるとき、つまり a.length = n = n_i が成り立つ場合だ。この 1 つ前に行った resize() 操作について考えよう。その resize() の直後、a の大きさは a.length だが、a の要素数は a.length/2 = $n_i/2$ 以下であった。しかし、a の要素数はいまでは n_i = a.length なのだから、前の resize() から $n_i/2$ 回以上は add(i,x) が呼ばれたことがわかる。

もう 1 つ考えられるのは、resize() が remove(i) の中で呼ばれる場合で、このとき a.length $\ge 3n = 3n_i$ である。この 1 つ前、つまり i − 1 回めの resize() の直後では、a の要素数は a.length/2 − 1 以上であった [*5]。いま、a には $n_i \le$ a.length/3 個の要素が入っている。よって、直前の resize() 以降に実行された remove(i) の回数の下界は次のように計算できる。

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

[*5] この数式における −1 は、特別なケースである n = 0 かつ a.length = 1 を考慮したものだ。

いずれの場合も、$(i-1)$ から i 回めの resize() の間に add(i,x) か remove(i) が呼ばれる回数の合計は $n_i/2-1$ 以上である。 □

2.1.3 要約

次の定理は ArrayStack の性能について整理したものだ。

定理 2.1. *ArrayStack* は *List* インターフェースを実装する。resize() にかかる時間を無視した場合の *ArrayStack* における各操作の実行時間を以下にまとめる。

- get(i) および set(i,x) の実行時間は $O(1)$ である
- add(i,x) および remove(i) の実行時間は $O(1+n-i)$ である

空の *ArrayStack* に対して任意の m 個の add(i,x) および remove(i) からなる操作の列を実行する。このとき resize() にかかる時間の合計は $O(m)$ である。

ArrayStack というデータ構造は、Stack インターフェースを実装する効率的な方法である。特に、push(x) は add(n,x) に相当し、pop() は remove(n−1) に相当する。これらいずれの操作の償却実行時間も $O(1)$ である。

2.2 FastArrayStack：最適化された ArrayStack

ArrayStack で主にやっていることは、(add(i,x) と remove(i) のために) データをシフトすることと、(resize() のために) データをコピーすることである。上記の実装では、これに for ループを使った。しかし実際には、データのシフトやコピーに特化したもっと効率的な機能が使えることが多い。C 言語には、memmove(d,s,n) と memcpy(d,s,n) 関数がある [*6]。C++ には、std::copy(a0,a1,b) アルゴリズムがある。Java には、System.arraycopy(s,i,d,j,n) メソッドがある。

[*6] 訳注：memmove(d,s,n) は、移動先（destination）に移動元（source）から n バイトをコピーする関数である。memcpy(d,s,n) との違いは、移動元と移動先の領域が重なっていてもよいことである。

FastArrayStack

```
void resize() {
  array<T> b(max(1, 2*n));
  std::copy(a+0, a+n, b+0);
  a = b;
}
void add(int i, T x) {
  if (n + 1 >= a.length) resize();
  std::copy_backward(a+i, a+n, a+n+1);
  a[i] = x;
  n++;
}
```

これらの関数は最適化されており、for ループを使う場合と比べてかなり高速にデータのコピーが可能な機械語の命令を使っている可能性がある。これらの関数を使っても漸近的な実行時間は小さくならないが、最適化として試してみる価値はある。

ここで示した C++ の実装では、組み込みの std::copy(a0,a1,b) 関数の利用により、操作の種類によっては 2〜3 倍の高速化に繋がる。自分の手元の環境でどれくらい速くなるか、ぜひ試してみてほしい。

2.3 ArrayQueue：配列を使ったキュー

この節では FIFO（先入れ先出し）キューを実装するデータ構造 ArrayQueue を紹介する。このデータ構造では、（add(x) によって）追加された要素が、同じ順番で（remove() によって）削除される。

FIFO キューの実装に ArrayStack を使うのは好ましくない。これが賢明な選択でないのは、ArrayStack では先頭か末尾のいずれかを要素を追加する側に、他方を削除する側に選ばなければならず、2 つの操作のいずれかがリストの先頭を変更することになるからだ。そうすると、i = 0 で add(i,x) か remove(i) を呼び出すことになり、n に比例する実行時間がかかってしまうのである。

もし無限長の配列 a があれば、配列を使った効率的なキューを簡単に実装で

きるだろう。次に削除する要素を追跡するインデックス j と、キューの要素数 n を記録しておけばよい。そうすれば、キューの要素は以下の場所に入っていることになる。

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

まず j, n を 0 に初期化する。要素を追加するときは、a[j+n] に要素を入れて、n を 1 つ増やす。要素を削除するときは、a[j] から要素を取り出し、j を 1 つ増やして、n を 1 つ減らす。

この方法の明らかな問題点は、無限長の配列が必要なことだ。ArrayQueue を使うことで、無限長の配列を、有限長の配列 a と**剰余算術**で模倣できる。剰余算術というのは、時刻に対して使うような計算だ。例えば、10:00 に 5 時間を足すと 3:00 になる。これを形式的に書けば次のようになる。

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

上の数式の後半は、「12 を法として 15 と 3 は合同である」と読む。mod は次のような二項演算と考えてもよい。

$$15 \bmod 12 = 3$$

整数 a と正整数 m について、ある整数 k が存在し $a = km + r$ を満たす整数 $r \in \{0, \ldots, m-1\}$ を $a \bmod m$ と書く。簡単に言うと、r は a を m で割った余りである。 C++ を含む多くのプログラミング言語では、mod 演算子を % で表す[*7]。

剰余算術は無限長の配列を模倣するのに便利である。i mod a.length が常に 0,...,a.length − 1 の値を取ることを利用して、配列の中にキューの要素をうまく入れられるのだ。

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

ここでは a を**循環配列**として使っている。配列の添字が a.length − 1 を超えると、配列の先頭に戻ってくるわけである。

残りの問題は ArrayQueue の要素数が a の大きさを超えてはならないことだ。

[*7] これは第一引数が負の場合について数学における mod 演算子を正確に実装したものではないので、時として**脳死**した mod 演算子と呼ばれることがある。

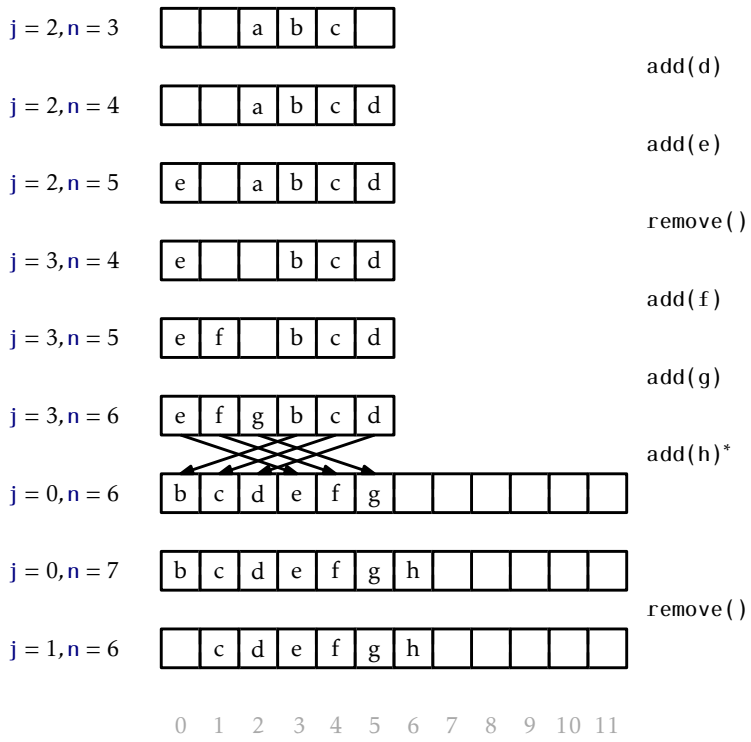


図 2.2: ArrayQueue に対する add(x)、remove() の実行例。矢印は要素のコピーを表す。resize() が発生する呼び出しにはアスタリスクを付した

ArrayQueue

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

ArrayQueue に対して add(x) および remove() からなる操作の列を実行する様子を図 2.2 に示す。add(x) の実装では、まず a が一杯かどうかを確認し、必要に応じて resize() を呼んで a の容量を増やす。続いて、x を a[(j+n)%a.length] に入れて、n を 1 つ増やせばよい。

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

remove() の実装では、まず a[j] をあとで返せるように保存しておく。続いて n を 1 減らし、$j = (j+1) \bmod$ a.length とすることで j を 1 増やす (a.length を法として計算している)。最後に保存しておいた a[j] を返す。もし必要なら resize() を読んで a を小さくする。

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

resize() 操作は ArrayStack の resize() とよく似ている。大きさ 2n の新しい配列 b を割り当て、

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

を

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

にコピーし、$j = 0$ とする。

ArrayQueue
```
void resize() {
  array<T> b(max(2*n, 1));
  for (int k = 0; k < n; k++)
```

```
    b[k] = a[(j+k)%a.length];
  a = b;
  j = 0;
}
```

2.3.1 要約

次の定理は ArrayQueue の性能について整理したものだ。

定理 2.2. *ArrayQueue* は、（*FIFO* の）*Queue* インターフェースの実装である。resize() のコストを無視すると、*ArrayQueue* は add(x)、remove() の実行時間は $O(1)$ である。さらに、空の *ArrayQueue* に対して長さ m の任意の add(x) および remove() からなる操作の列を実行するとき、resize() にかかる時間の合計は $O(m)$ である。

2.4 ArrayDeque：配列を使った高速な双方向キュー

前節の ArrayQueue は、一方の端からは追加だけを、他方の端からは削除だけを効率的に実行できるような列を表すデータ構造であった。この節で紹介する ArrayDeque は、両端に対して追加と削除が効率的に実行できるデータ構造である。このデータ構造により、ArrayQueue を実装するのに使ったのと同じく、循環配列を使って List インターフェースを実装する [*8]。

ArrayDeque

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

ArrayDeque に対する get(i) と set(i,x) は簡単だ。配列の要素 a[(j + i) mod a.length] を読み書きすればよい。

[*8] 訳注：1.2.2 節で言及したように、ArrayDeque は List インターフェースを実装するデータ構造である。ArrayDeque という名称は、Deque インターフェースのすべての操作の実行時間が $O(1)$ であることを強調するために付けられている。

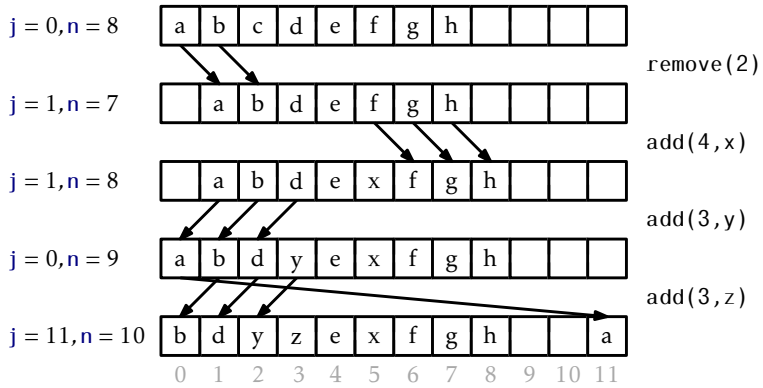


図 2.3: ArrayDeque に対する add(i,x)、remove(i) の実行例。矢印は要素のコピーを表す

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;
  return y;
}
```

add(i,x) の実装には工夫が必要になる。まず a が一杯かどうかを確認し、必要に応じて resize() を呼ぶ。いま、add(i,x) 操作については、i が小さいとき（0 に近いとき）と i が大きいとき（n に近いとき）に特に効率が良くなるようにしたい。そこで、$i < n/2$ かどうかを確認する。もしそうなら、左から i 個の要素をいずれも 1 つずつ左にずらす。そうでないなら、右から $n - i$ 個の要素をいずれも 1 つずつ右にずらす。ArrayDeque に対する add(i,x) と remove(x) を図 2.3 に示す。

ArrayDeque

```
void add(int i, T x) {
  if (n + 1 >= a.length)  resize();
  if (i < n/2) { // a[0],..,a[i-1] を左に 1 つずらす
    j = (j == 0) ? a.length - 1 : j - 1;
    for (int k = 0; k <= i-1; k++)
      a[(j+k)%a.length] = a[(j+k+1)%a.length];
  } else { // a[i],..,a[n-1] を右に 1 つずらす
    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++;
}
```

このように要素をずらせば、add(i,x) によって移動する要素の数が高々 $\min\{i, n-i\}$ 個に保証される。そのため、add(i,x) の実行時間は、resize() を無視すれば $O(1+\min\{i, n-i\})$ である。

remove(i) も同様に実装できる。$i < n/2$ かどうかに応じて、左から i 個の要素をいずれも 1 つずつ右にシフトするか、右から $n-i-1$ 個の要素をいずれも 1 つずつ左にシフトする。remove(i) の実行時間も、やはり $O(1+\min\{i, n-i\})$ である。

ArrayDeque

```
T remove(int i) {
  T x = a[(j+i)%a.length];
  if (i < n/2) { // a[0],..,[i-1] を右に 1 つずらす
    for (int k = i; k > 0; k--)
      a[(j+k)%a.length] = a[(j+k-1)%a.length];
    j = (j + 1) % a.length;
  } else { // a[i+1],..,a[n-1] を左に 1 つずらす
    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 要約

次の定理は ArrayDeque の性能について整理したものだ。

定理 **2.3.** *ArrayDeque* は *List* インターフェースを実装する。resize() のコストを無視すると、*ArrayDeque* における各操作の実行時間は下記のようになる。

- get(i) および set(i,x) の実行時間は $O(1)$ である
- add(i,x) および remove(i) の実行時間は $O(1+\min\{i, n-i\})$ である [*9]

さらに、任意の add(i,x) および remove(i) からなる長さ m の操作の列を空の *ArrayDeque* に対して実行するとき、resize() にかかる時間の合計は $O(m)$ である。

2.5 DualArrayDeque：2 つのスタックから作った双方向キュー

次は、2 つの ArrayStack を使うことで ArrayDeque に近い性能を実現する、DualArrayDeque というデータ構造を紹介する。漸近的な性能が ArrayDeque より向上するわけではないが、2 つのシンプルなデータ構造を組み合わせてより高度なデータ構造を作る例として取り上げる。

DualArrayDeque では、リストを表現するのに 2 つの ArrayStack を使う。ArrayStack に対する操作は、終端付近の要素に対して高速だったことを思い出してほしい。両端の要素に対する操作を高速にするため、DualArrayDeque

[*9] 訳注：これらの結果から、ArrayDeque が確かに Deque インターフェースのすべての操作を $O(1)$ で実現していることを確認できる。つまり、両端に対する add(i,x) および remove(i) の実行時間は、$O(1)$ で済む。

では front と back という名前の 2 つの ArrayStack を背中合わせに配置する。

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

DualArrayDeque では、要素数 n を明示的に保持しない。要素数は n = front.size() + back.size() により求められるからだ。ただし、DualArrayDeque の解析に際しては、いままで通り要素数を n で表すことにする。

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

1 つめの ArrayStack である front には、0,...,front.size() − 1 番めの要素を逆順に入れる。もう 1 つの ArrayStack である back には、front.size(),...,size() − 1 番めの要素をそのままの順番で入れる。あとは、front または back に対して get(i) や set(i,x) を適切に呼べば、get(i) および set(i,x) を $O(1)$ の時間で実行できる。

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 {
```

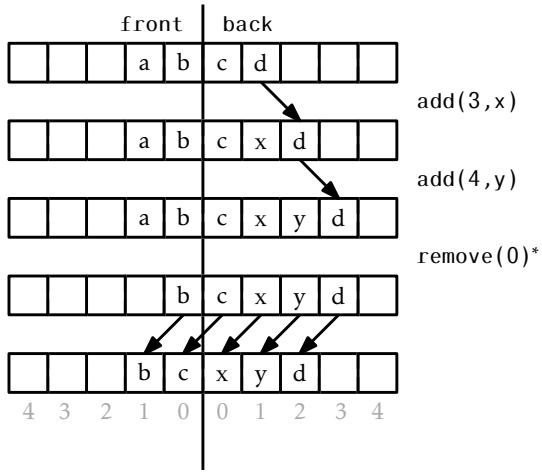


図 2.4: DualArrayDeque に対する add(i,x) および remove(i) の実行例。矢印は要素のコピーを表す。balance() の発生する呼び出しにはアスタリスクを付した

```
    return back.set(i - front.size(), x);
  }
}
```

front には逆順に要素が入っているので、DualArrayDeque の i 番め（i < front.size()）は、front の front.size() − i − 1 番めの要素である。

DualArrayDeque に対する要素の追加と削除については、図 2.4 を見てほしい。add(i,x) により、front または back のいずれか適切なほうが操作される。

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();
}
```

add(i,x) では、front と back の要素数を均すために balance() を呼び出す。balance() の実装は後述する。いまのところは、balance() のおかげで front.size() と back.size() の差が三倍より大きくなることはないと理解しておけばよい（size() < 2 の場合を除く）。具体的には、balance() により、3·front.size() ≥ back.size() かつ 3·back.size() ≥ front.size() であることが保証される。

では、balance() のコストを無視した add(i,x) の実行時間を求めよう。i < front.size() のときは、add(i,x) により front.add(front.size() − i,x) が実行されるだけである。front は ArrayStack なので、この実行時間は次のようになる。

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

一方、i ≥ front.size() のときには、add(i,x) により back.add(i − front.size(),x) が実行されるだけである。このときの実行時間は次のようになる。

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

i < n/4 のときは、1 つめのケース (2.1) に該当する。i ≥ 3n/4 のときは、2 つめのケース (2.2) に該当する [*10]。n/4 ≤ i < 3n/4 のときは、front と back のどちらが操作されるかわからない。しかし、いずれの場合も高々 n 個の要素をずらして新たな要素を配列に入れるので、実行時間は $O(\texttt{n})$ である。以上をまとめると次のようになる。

$$\text{add(i,x) の実行時間} \le \begin{cases} O(1+\texttt{i}) & \text{if } \texttt{i} < \texttt{n}/4 \\ O(\texttt{n}) & \text{if } \texttt{n}/4 \le \texttt{i} < 3\texttt{n}/4 \\ O(1+\texttt{n}-\texttt{i}) & \text{if } \texttt{i} \ge 3\texttt{n}/4 \end{cases}$$

ゆえに、add(i,x) の実行時間は、balance() のコストを無視すれば $O(1 + \min\{\texttt{i}, \texttt{n} - \texttt{i}\})$ である。

remove(i) についても、add(i,x) と同様に実行時間を解析できる。

[*10] 訳注：例えば、$i = 0$ かつ $n = 0$ の場合は (2.2) に該当する。

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 バランスの調整

最後に、add(i,x) と remove(i) の操作において実行される balance() について説明する。この balance() 操作により、front と back の要素数が極端に偏らないことが保証される。具体的には、要素数が 2 以上のとき、front も back も n/4 以上の要素を含むようにする。front か back に n/4 以上の要素が含まれそうな場合は、要素を動かして、front と back にそれぞれちょうど $\lfloor n/2 \rfloor$ 個および $\lceil n/2 \rceil$ 個の要素が含まれるようにする。

DualArrayDeque
```
void balance() {
  if (3*front.size() < back.size()
      || 3*back.size() < front.size()) {
    int n = front.size() + back.size();
    int nf = n/2;
    array<T> af(max(2*nf, 1));
    for (int i = 0; i < nf; i++) {
      af[nf-i-1] = get(i);
    }
    int nb = n - nf;
```

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

balance() の実行時間は簡単に解析できる。balance() によってバランスが調整されるときは、$O(n)$ 個の要素が動かされるので、実行時間は $O(n)$ である。これは一見すると都合が悪い。balance() は add(i,x) および remove(i) のたびに実行されるからだ。しかし次の補題により、balance() の実行時間は平均的には定数であることがわかる。

補題 2.2. 空の *DualArrayDeque* に対して、add(i,x) および remove(i) からなる長さ m の任意の操作の列を実行する。このとき resize() にかかる時間の合計は $O(m)$ である。

証明. balance() によって要素が動かされてから、次に balance() によって要素が動かされるまでに、add(i,x) および remove(i) が実行される回数が $n/2-1$ 以上であることを示す。補題 2.1 の証明と同様に、これを示せば balance() の合計実行時間が $O(m)$ であることを示したことになる。

ここでは**ポテンシャル法（potential method）**という技法を使う。DualArrayDeque の**ポテンシャル** Φ を、front と back の要素数の差と定義する。

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

add(i,x) および remove(i) の処理でバランスを調整しない場合、ポテンシャル Φ の増加が高々 1 であることに注目しよう。

次の式が成り立つので、要素を動かす balance() を呼び出した直後のポテンシャル Φ_0 は 1 以下である点に注目しよう。

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

balance() が呼び出されて要素が動く直前の状況について考えよう。このとき、一般性を失うことなく、3front.size() < back.size() であったと仮定できる。この場合、次の式が成り立つ。

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

このときのポテンシャル Φ_1 は次のように評価できる。

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

以上より、add(i,x) および remove(i) が呼ばれる回数は、それ以前に balance() によって要素が動かされてから $\Phi_1 - \Phi_0 > n/2 - 1$ 以上である。 □

2.5.2 要約

次の定理は DualArrayDeque の性質を整理したものだ。

定理 2.4. *DualArrayDeque* は *List* インターフェースを実装する。resize() と balance() のコストを無視すると、*DualArrayDeque* における各操作の実行時間は次のようになる。

- get(i) および set(i,x) の実行時間は $O(1)$ である
- add(i,x) および remove(i) の実行時間は $O(1 + \min\{i, n - i\})$ である

また、空の *DualArrayDeque* に対して長さ m の任意の add(i,x) および remove(i) からなる操作の列を実行するとき、resize() にかかる時間の合計は $O(m)$ である。

2.6 RootishArrayStack：メモリ効率に優れた配列スタック

これまでに紹介したデータ構造には共通の欠点がある。データの格納に配列を 1 つ、もしくは 2 つしか使っておらず、しかも頻繁なサイズ変更を避けていることから配列に空きがたくさんある状況が多いという点である。例えば、resize() 直後の ArrayStack では配列が半分しか埋まっていない。3 分の 1 しか埋まっていない状況さえある。

この節では、無駄なスペースが少ない RootishArrayStack というデータ構造を紹介する [*11]。RootishArrayStack では、n 個の要素を $O(\sqrt{n})$ 個の配列に入れる。各配列は、常に $O(\sqrt{n})$ 箇所以下しか空いていない。残りのすべての場所にはデータが入っているのだ。つまり、n 個の要素を入れるときに無駄になるスペースは $O(\sqrt{n})$ 以下である。

RootishArrayStack では**ブロック**と呼ぶ r 個の配列に要素を入れる。これらの配列には 0,1,...,r − 1 と番号を付ける。図 2.5 を見てほしい。b 番めのブロックには b + 1 個の要素を入れる。すなわち、r 個のブロックに含まれる要素数の合計は次のように計算できる。

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

この等式が成り立つことは図 2.6 を見ればわかるだろう。

RootishArrayStack

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

リストの要素はブロックに順番に入れる。リストの 0 番めの要素はブロック 0 に、1 番めと 2 番めの要素はブロック 1 に、3 番めと 4 番めと 5 番めの要素はブロック 2 に格納する。リスト全体で見たときに i 番めの要素がどのブロックのどの位置に入っているか、どうすればわかるだろうか？

i 番めの要素がどのブロックに入っているかさえわかれば、ブロック内での位置は簡単に計算できる。i 番めの要素が b 番めのブロックに入っているなら、0,...,b − 1 番めの各ブロックにおける要素数の合計は b(b + 1)/2 である。そのため、i 番めの要素は、b 番めのブロックにおいて下記のように計算でき

[*11] 訳注：RootishArrayStack は、前節までのデータ構造と比べると、実際に目にする機会は少ないように思える。そのため、初学者は読み飛ばしてもよい。ただし課題設定と設計アイデアは興味深い。

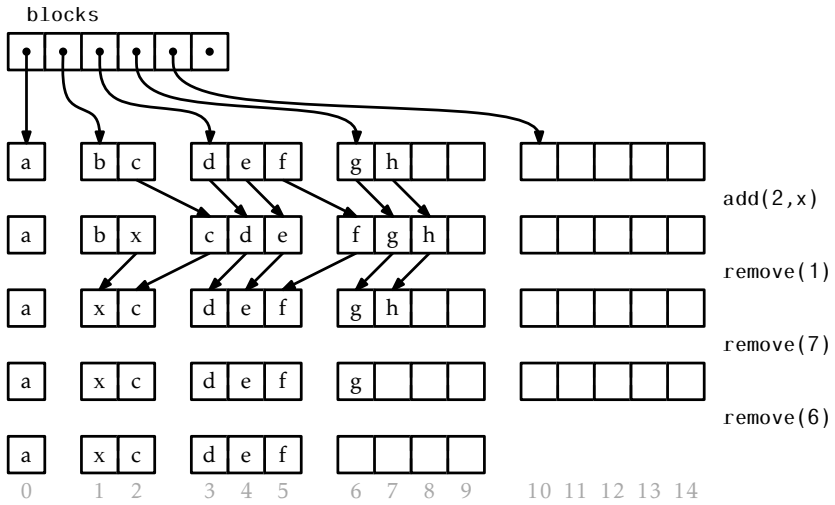


図 2.5: RootishArrayStack に対する add(i,x) および remove(i) の実行例。矢印は要素のコピーを表す

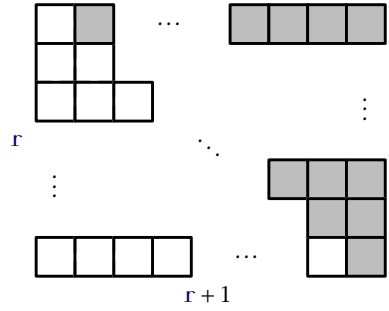


図 2.6: 白い正方形の数は合わせて $1+2+3+\cdots+r$ である。斜線を引いた正方形の数も同じである。白い正方形と斜線を引いた正方形を合わせてできる正方形全体は、$r(r+1)$ 個の正方形からなる

る j 番めの位置に入っている [*12]。

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

i から b を求める、つまり i 番めの要素がどのブロックに入っているのかを計算する方法はもう少しややこしい。i 以下のインデックスを持つ要素は $i+1$ 個ある。一方、0,...,b 番めのブロックに入っている要素数の合計は $(b+1)(b+2)/2$ である。よって、b は次の式を満たす最小の整数である。

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

この式は次のように変形できる。

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

二次方程式 $b^2 + 3b - 2i = 0$ は、2 つの解 $b = (-3 + \sqrt{9+8i})/2$ と $b = (-3 - \sqrt{9+8i})/2$ を持つ。2 つめの解は常に負の値なので捨ててよい。よって、解は $b = (-3 + \sqrt{9+8i})/2$ である。この解は一般に整数とは限らない。とはいえ、元の不等式に立ち戻って考えれば、欲しかったのは $b \geq (-3 + \sqrt{9+8i})/2$ を満たす最小の b である。これは次のように書ける。

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

RootishArrayStack

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

インデックス i からブロック番号 b への変換関数 i2b を用いれば、get(i) と set(i,x) を実装するのは簡単だ。まず b を計算し、そのブロック内のインデックス j を求め、適切な操作を実行すればよい。

[*12] 訳注：例えば、$b = 2$ かつ $i = 3$ のとき、$j = 3 - 3 = 0$ となり、i に対応するのは 2 番めのブロックの 0 番めの要素である。

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;
}
```

この章に出てくるデータ構造のどれかを使って blocks のリストを表現すれば、get(i) も set(i,x) も実行時間は定数である。

add(i,x) については、これまで紹介した他のデータ構造と同じように考えればよい。まずデータ構造が一杯かどうか、つまり $r(r+1)/2 = n$ かどうかを確認する。もしそうなら、新たなブロックを追加するために、grow() というメソッドを呼び出す。その後、i,...,n−1 番めの要素をそれぞれ右に 1 つずらし、新たな i 番めの要素を入れるための隙間を作る。

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);
}
```

grow() メソッドにより新しいブロックが追加される。

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

grow() のコストを無視すれば、シフト操作の回数を数えることで、add(i,x) の実行時間が $O(1+n-i)$ であるとわかる。したがって、ArrayStack と同じである。

remove(i) も add(i,x) と同様だ。$i+1,\ldots,n$ 番めの要素をそれぞれ左に 1 つずつシフトし、2 つ以上の空のブロックがあれば shrink() を呼び出して、使われていないブロックを 1 つだけ残して削除する。

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

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

shrink() のコストを無視すれば、シフト操作の回数を数えることで、remove(i) の実行時間が $O(n-i)$ であるとわかる [*13]。

2.6.1 拡張、収縮の分析

add(i,x) と remove(i) の解析では grow() と shrink() を考慮していなかった。ArrayStack.resize() の場合とは違い、grow() と shrink() では要素がコピーされないことに注意しよう。つまり、grow() と shrink() は、それぞれ大きさ r の配列の割り当ておよび解放をするだけである。環境によって、これは定数時間で実行できたり、r に比例する時間がかかったりする。

grow() と shrink() を呼んだ直後の状況はわかりやすい。最後のブロックは空で、それ以外のブロックが一杯になっている。そのため、次の grow() と shrink() が呼ばれるのは、少なくとも $r-1$ 回だけ要素が追加および削除されたあとである。よって、grow() および shrink() に $O(r)$ だけ時間がかかっても、そのコストは $r-1$ 回の add(i,x) および remove(i) で償却され、grow() と shrink() の償却コストは $O(1)$ である。

2.6.2 空間使用量

RootishArrayStack が使う無駄な領域の量を解析する。RootishArrayStack が確保している配列の中で、データが入っていない箇所を数えたい。そのような箇所を**無駄な領域**ということにする。

remove(i) があるので、RootishArrayStack で空きのあるブロックは高々 2 つしかないことが保証される。よって、n 個の要素を含む RootishArrayStack のブロック数を r とすれば、次の関係が成り立つ。

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

やはり二次方程式の解を考えることで次の式が成り立つ。

$$r \le (3+\sqrt{1+8n})/2 = O(\sqrt{n})$$

末尾の 2 つのブロックの大きさは r と $r-1$ なので、これらのブロックによる無駄な領域の量は高々 $2r-1 = O(\sqrt{n})$ である。もし、これらのブロックを

[*13] 訳注：remove(i) の場合は常に $i < n$、すなわち $n-i > 0$ が成り立つので、add(i,x) の計算量のように 1 を足す必要がない。

（例えば）ArrayStack に入れれば、r 個のブロックを格納する List による無駄な領域の量も $O(r) = O(\sqrt{n})$ である。n の値など、その他の情報を保持するのに使う領域は、$O(1)$ である。以上より、RootishArrayStack の無駄な領域の量は合計 $O(\sqrt{n})$ である。

この領域の量が、空から始めて要素を 1 つずつ追加できるデータ構造のうちで最適であることを示そう。厳密には、n 個の要素を追加する際にはどこかのタイミングで（ほんの一瞬かもしれないが）$\sqrt{n}$ 以上の無駄な領域が生じることを示す。

空のデータ構造に n 個の要素を 1 つずつ追加していくとする。追加が完了した時点では n 個のアイテムがすべてデータ構造に格納されており、それが r 個のブロックに分散している。$r \geq \sqrt{n}$ なら、r 個のブロックを追跡するために r 個のポインタ（参照）を使うしかない。これらのポインタは無駄な領域である [*14]。一方で、$r < \sqrt{n}$ なら、鳩の巣原理により大きさ $n/r > \sqrt{n}$ 以上のブロックが存在する。このブロックが初めて割り当てられた瞬間を考える。このブロックは、割り当てられたときは空なので、$\sqrt{n}$ の無駄な領域が生じている。以上より、n 個の要素を挿入するまでのあるタイミングで、データ構造には $\sqrt{n}$ の無駄な領域が生じる。

2.6.3 要約

次の定理は RootishArrayStack について整理したものだ。

定理 **2.5.** *RootishArrayStack* は *List* インターフェースを実装する。grow() および shrink() のコストを無視すると、*RootishArrayStack* における各操作の実行時間は下記のようになる。

- get(i) および set(i,x) の実行時間は $O(1)$ である
- add(i,x) および remove(i) の実行時間は $O(1 + n - i)$ である

空の *RootishArrayStack* に対して、add(i,x) および remove(i) からなる長さ m の任意の操作の列を実行するとき、grow() および shrink() にかかる時間の合計は $O(m)$ である。

要素数 n の *RootishArrayStack* が使う（ワード単位で測った）空間使用

[*14] 訳注：例えば、第 3 章ではこの考えをさらに進めて、r=n 個のブロックを追跡するために n 個のポインタを用いる連結リストと呼ばれるデータ構造を紹介する。

量[*15] は $n + O(\sqrt{n})$ である。

2.6.4 平方根の計算方法

計算モデルは、計算を理論的に調べるための道具である。ここまでは、w ビットのワード RAM モデルという計算モデルに基づいて操作の実行時間やデータ構造のメモリ使用量を調べてきた。1.4 節によれば、ワード RAM モデルの基本的な操作は算術演算、比較、ビット単位の論理演算であり、平方根の算出は含まない。平方根を計算している `RootishArrayStack` は、ワード RAM モデルに適合しないことに気づいた読者がいるかもしれない。

この節では、平方根の算出が効率的に実装できることを示す。具体的には、長さが $O(\sqrt{n})$ の 2 つの配列（`sqrttab` と `logtab`）を実行時間 $O(\sqrt{n})$ の前処理で作っておけば、どんな自然数 $x \in \{0,\ldots,n\}$ についても定数時間で $\lfloor\sqrt{x}\rfloor$ が計算できることを示す。

次の補題は、x の平方根の計算を、x に関係する値 x' の平方根の計算に帰着できることを示すものだ。

補題 **2.3.** *2 つの数 $x \geq 1$ と $x' = x - a$ について、$0 \leq a \leq \sqrt{x}$ だと仮定する。このとき、$\sqrt{x'} \geq \sqrt{x} - 1$ である。*

証明. 以下を示せばよい。

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

両辺の二乗を取ると、下記のようになる。

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

整理すると下記のようになる。

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

これはどんな $x \geq 1$ についても成り立つ。 □

あらゆる自然数 $x \in \{0,\ldots,n\}$ の平方根について考える前に、少し問題を制約しよう。自然数 x が $2^r \leq x < 2^{r+1}$ を満たす場合、すなわち $\lfloor \log x \rfloor = r$ である場合を考える。このとき自然数 x は、$r+1$ ビットの二進表記で表せる[*16]。

[*15] 1.4 節で説明した、どのようにメモリ量を測るかという話を思い出してほしい。

[*16] 訳注：例えば r が 2 なら、$x = 5, 6, 7$ をそれぞれ 3 ビットで 101, 110, 111 と表せる。

x と x' の関係は補題 2.3 の仮定を満たすので、$\sqrt{x} - \sqrt{x'} \le 1$ が成り立つ。ここで、$x' = x - (x \bmod 2^{\lfloor r/2 \rfloor})$ とおくと、x' の下位 $\lfloor r/2 \rfloor$ ビットはすべて 0 である。したがって、x' が取りうる値としては、$2^{r+1-\lfloor r/2 \rfloor}$ 通りの可能性が考えられる。その可能性は、下記により、$O(\sqrt{n})$ で抑えられることがわかる [*17]。

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

x' として可能性があるのは高々 $O(\sqrt{n})$ 通りなので、$\lfloor \sqrt{x'} \rfloor$ として可能性がある値をすべて格納する配列 `sqrttab` を用意することにしよう。この配列 `sqrttab` の各要素の値は、具体的には下記のようにする。

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

こうすれば、$x \in \{i2^{\lfloor r/2 \rfloor}, \ldots, (i+1)2^{\lfloor r/2 \rfloor} - 1\}$ のそれぞれについて、$\sqrt{x}$ の値と $\texttt{sqrttab}[i]$ の値の差がおよそ 2 より大きくなることはない。言い換えれば、配列の各要素 $s = \texttt{sqrttab}[x \gg \lfloor r/2 \rfloor]$ [*18] は、$\lfloor \sqrt{x} \rfloor$ か、$\lfloor \sqrt{x} \rfloor - 1$ か、$\lfloor \sqrt{x} \rfloor - 2$ のいずれかになる。$(s+1)^2 > x$ となるまで s をインクリメントすることで、x の平方根を下に丸めた自然数 $\lfloor \sqrt{x} \rfloor$ の値を特定できる [*19]。

FastSqrt

```
int sqrt(int x, int r) {
  int s = sqrtab[x>>r/2];
  while ((s+1)*(s+1) <= x) s++; // 高々二回だけ実行する
  return s;
}
```

ここまでは $x \in \{2^r, \ldots, 2^{r+1} - 1\}$ の場合についてのみ考えてきた。また、`sqrttab` は $r = \lfloor \log x \rfloor$ についてのみ使えるものであった。これを一般化する

[*17] 訳注：左辺に 2 を掛けると、左辺と中辺に関する不等号が導出できる。中辺と右辺の不等号については、$2^r \le x < 2^{r+1}$ から導出される $2^{r/2} \le \sqrt{x}$ を用いて得られる。

[*18] `x>>n` は右シフト演算と呼ばれ、x を表すビットのそれぞれを n ビットずつ右にずらす。算術上は、x から $x \bmod 2^n$ を引いた（すなわち、下位 n ビットをすべて 0 にした）あとに 2^n で割った場合と同じ効果を持つ。

[*19] 訳注：平方根はプログラム中で何千回も使いまわされる可能性がある処理だから、空間計算量を少し犠牲にして中間結果を配列 `sqrttab` に入れることで、時間計算量を改善しようという算段である。x' の数が高々 $2^{r+1-\lfloor r/2 \rfloor}$ 通り（例えば一般的なコンピュータにおける $r = 32$ の場合は 10 万通り程度）しかなく、$2^{r+1-\lfloor r/2 \rfloor} \le 4\sqrt{x}$ という結果からビッグオー記法でいえば $O(\sqrt{n})$ 通りしかないのだから、この工夫は実を結ぶ。

には、$\lfloor \log n \rfloor$ 個の sqrttab を $\lfloor \log x \rfloor$ の各値に対して準備すればよさそうだ。各 sqrttab の大きさは等比数列で、最大のものの大きさは高々 $4\sqrt{n}$ である。そのため、すべての sqrttab の大きさを合計すると $O(\sqrt{n})$ である。

しかし、実は sqrttab は 1 つで十分である。$r = \lfloor \log n \rfloor$ の場合の sqrttab だけがあればよい。

$\log x = r' < r$ である x については、$2^{r-r'}$ を掛けて**アップグレード**した次の等式を使えばよい。

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

$2^{r-r'}x$ は $\{2^r, \ldots, 2^{r+1}-1\}$ に含まれるので、上の値は sqrttab に入っている。次のコードは、$\{0, \ldots, 2^{30}-1\}$ に含まれる任意の x について、大きさ 2^{16} の配列 sqrttab を使って $\lfloor \sqrt{x} \rfloor$ を計算するものである。

FastSqrt

```
int sqrt(int x) {
  int rp = log(x);
  int upgrade = ((r-rp)/2) * 2;
  int xp = x << upgrade;  // xp は r ビットまたは r-1 ビット
  int s = sqrtab[xp>>(r/2)] >> (upgrade/2);
  while ((s+1)*(s+1) <= x) s++;  // 高々二回だけ実行する
  return s;
}
```

$r' = \lfloor \log x \rfloor$ の計算方法も説明しておく。平方根の場合と同様に、大きさ $2^{r/2}$ の配列 logtab を使う。$\lfloor \log x \rfloor$ が x を二進表記したときに 1 となる最大の桁の添字であることに気づけば、実装は難しくない。すなわち、$x > 2^{r/2}$ のとき x を r/2 ビットだけ右にシフトし、logtab の添字とする。次のコードは、$\{0, \ldots, 2^{32}-1\}$ に含まれる任意の x について、大きさ 2^{16} の配列 logtab を使って $\lfloor \log x \rfloor$ を計算するものである。

FastSqrt

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

最後に、logtab および sqrttab を初期化するコードを掲載しておく。

FastSqrt

```
void inittabs() {
  sqrtab = new int[1<<(r/2)];
  logtab = new int[1<<(r/2)];
  for (int d = 0; d < r/2; d++)
    for (int k = 0; k < 1<<d; k++)
      logtab[1<<d+k] = d;
  int s = 1<<(r/4);                    // sqrt(2^(r/2))
  for (int i = 0; i < 1<<(r/2); i++) {
    if ((s+1)*(s+1) <= i << (r/2)) s++; // 平方根の値を増やす
    sqrtab[i] = s;
  }
}
```

まとめると、ワード RAM では、$O(\sqrt{n})$ の余分なメモリを使って sqrttab および logtab という配列を作ることで、i2b(i) を用いる各操作を定数時間で実行できる。この配列は、n が 2 倍または 2 分の 1 の大きさになるたびに拡大または縮小してもよい。その場合の実行時間は、ArrayStack のときと同様に、add(i,x) および remove(i) を実行する回数にわたって償却できる。

2.7 ディスカッションと練習問題

この章で説明したデータ構造の大半は古くから知られているもので、多くの議論がなされてきた。30 年以上前の実装さえ見つかる。例えば、Knuth による [46, Section 2.2.2] で述べられているスタック、キュー、双方向キューの実装は、一般化すればここで説明した ArrayStack、ArrayQueue、ArrayDeque になる。

RootishArrayStack について記述し、2.6.2 節で述べた下界 $\sqrt{n}$ を示した最初の文献は、おそらく Brodnik らによる [13] である。彼らは、この章で説明した方法とは異なる巧妙なブロックサイズの選び方も示しており、このやり方では i2b(i) の中で平方根の計算をせずに済む。彼らのやり方では、i を含むブロックのインデックスが $\lfloor \log(i+1) \rfloor$ 番めとなり、このインデックスは

i + 1 を二進表記したときの最高位の桁である。これはコンピュータアーキテクチャによっては専用の命令があり、効率的に計算できる。

RootishArrayStack に関連するデータ構造として、Goodrich と Kloss による [35] で示された二段階の**階層ベクトル（tiered-vector）**というものがある。このデータ構造では、get(i,x) および set(i,x) の実行時間は定数である。add(i,x) および remove(i) の実行時間は $O(\sqrt{n})$ である。問 2.10 では、RootishArrayStack をさらに改良し、これに近い実行時間を達成する。

問 **2.1.** List の addAll(i,c) 操作は、Collectionc の要素をすべてリストの i 番めの位置に順に挿入する（add(i,x) は c = {x} とした特殊な場合である）。この章で説明したデータ構造において、add(i,x) を繰り返し実行して addAll(i,c) を実装すると効率がよくない理由を説明せよ。また、より効率的な実装を考えて実装せよ。

問 **2.2.** RandomQueue を設計、実装せよ。RandomQueue は Queue インターフェースの実装であり、その remove() では、そのときにキューに入っている要素から一様な確率で 1 つを選んで取り出す（カバンに要素を入れておき、中を見ずに適当に要素を取り出すようなものだと考えればよい）。

ただし、RandomQueue における add(x) および remove() の償却実行時間は定数でなければならないとする。

問 **2.3.** Treque（triple-ended queue）を設計、実装せよ。Treque は List の実装であり、get(i) と set(i,x) は定数時間で実行できる。Treque の add(i,x) および remove(i) の実行時間は次のように表せる。

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

つまり、Treque は、両端あるいは中央に近い位置の修正が高速なデータ構造である。

問 **2.4.** 配列 a を「回転」する rotate(a,r) を実装せよ。すなわち、すべての $i \in \{0,\ldots,a.length\}$ について、a[i] を a[(i + r) mod a.length] に動かす操作を実装せよ。

問 **2.5.** List を回転する rotate(r) を実装せよ。すなわち、リストの i 番めの要素を (i + r) mod n 番めに移す操作を実装せよ。ただし、ArrayDeque や DualArrayDeque に対する rotate(r) の実行時間は $O(1 + \min\{r, n - r\})$ でなければならないとする。

問 **2.6.** ArrayDeque を実装せよ。ただし、add(i,x)、remove(i)、resize() におけるシフト処理では高速な System.arraycopy(s,i,d,j,n) を利用すること。

問 **2.7.** % 演算を用いずに ArrayDeque を実装せよ（この演算に時間がかかる環境もある）。a.length が 2 の冪なら次の式が成り立つことを利用してよい。

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

なお、& はビット単位の and 演算オペレータである。

問 **2.8.** 剰余演算を一切使わない ArrayDeque の実装を考えよ。すべてのデータは配列内の連続した領域に順番に並んでいることを利用してよい。データがこの配列の先頭もしくは末尾の外にはみ出したときは、rebuild() 操作を実行する。すべての操作の償却実行時間は ArrayDeque と同じになるように注意すること。
ヒント：rebuild() の実装方法がポイントだ。n/2 回以下の操作で、データがどちらの端からもはみ出さない状態に辿り着かなければならない。
　実装したプログラムの性能を、元の ArrayDeque と比較せよ。実装を（System.arraycopy(a,i,b,i,n) を使って）最適化し、ArrayDeque の性能を上回るかどうか確認せよ。

問 **2.9.** RootishArrayStack を修正し、無駄な領域の量は $O(\sqrt{n})$ だが add(i,x) および remove(i,x) の実行時間が $O(1+\min\{i, n-i\})$ であるデータ構造を設計、実装せよ。

問 **2.10.** RootishArrayStack を修正し、無駄な領域の量は $O(\sqrt{n})$ だが add(i,x) および remove(i,x) の実行時間が $O(1+\min\{\sqrt{n}, n-i\})$ であるデータ構造を設計、実装せよ（3.3 節が参考になるだろう）。

問 **2.11.** RootishArrayStack を修正し、無駄な領域の量は $O(\sqrt{n})$ だが add(i,x) および remove(i,x) の実行時間が $O(1+\min\{i, \sqrt{n}, n-i\})$ であるデータ構造を設計、実装せよ（3.3 節が参考になるだろう）。

問 **2.12.** CubishArrayStack を設計、実装せよ。CubishArrayStack は List インターフェースを実装する三段階のデータ構造であり、無駄な領域の量が $O(n^{2/3})$ である。CubishArrayStack では、get(i) および set(i,x) が定数時間で実行できる。add(i,x) および remove(i) の償却実行時間は $O(n^{1/3})$ である。

第 3

連結リスト

この章でも List インターフェースの実装を扱う。ただし今度は配列ではなくポインタを使う方法である。この章のデータ構造は、リストの要素を収めたノードの集まりである。参照（ポインタ）を使ってノードを繋げ、列を作る。まずは単方向連結リストを紹介する。これを使うと Stack と（FIFO）Queue の操作を定数時間で実行できる。次に双方向連結リストを紹介する。これを使うと Deque の操作を定数時間で実行できる。

List インターフェースの実装に連結リストを使うことには、配列を使う場合と比較して、次のような短所と長所がある。連結リストを使う主な短所は、get(i) や set(i,x) がすべての要素に対して定数時間ではなくなることだ。配列とは違って、i 番めの要素を読み書きする際に、そこまでリストをひとつずつ辿らなければならないのである。連結リストを使う主な長所は、動的な操作がしやすいことだ。リストのノードへの参照 u があれば、u の削除や u の隣へのノードの挿入が定数時間でできる。このとき u はリストの中のどのノードであってもよい [*1]。

3.1 SLList：単方向連結リスト

SLList（singly-linked list、単方向連結リスト）は、Node（ノード）からなる列である。各ノード u は、データ u.x と参照 u.next を保持している。参照は、列における次のノードを指している。列の末尾のノード w においては

[*1] 訳注：これも第 2 章における backing array を用いた List インターフェースの実装とは対照的である。第 2 章では、削除と挿入をどれだけ高速に実行できるかは、どのデータ構造も添字 i に依存していた。

w.next = null である。

SLList
```
class Node {
public:
  T x;
  Node *next;
  Node(T x0) {
    x = 0;
    next = NULL;
  }
};
```

効率のため、SLList では、列の先頭と末尾のノードへの参照を保持する変数 head および tail を利用する。また、列の長さを表す変数 n も利用する。

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

SLList における Stack 操作と Queue 操作を図 3.1 に示す。

SLList を使うと Stack の push(x) と pop() を効率的に実装できる。列の先頭に対して追加もしくは削除をすればよい。push(x) は新しいノード u を作り、そのデータには値 x を、参照 u.next にはそれまでの先頭を設定する。そして u を新しい先頭にする。最後に、SLList の要素が 1 つ増えたので、n を 1 だけ大きくする。

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

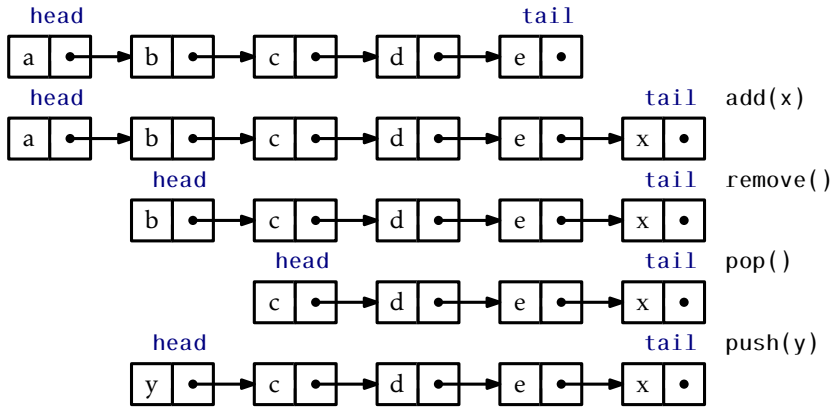


図 3.1: SLList における、Queue 操作（add(x)、remove()）と、Stack 操作（push(x)、pop()）

```
  n++;
  return x;
}
```

pop() では、SLList が空でないことを確認してから head = head.next として先頭を削除し、n を 1 だけ小さくする。最後の要素を削除する場合は特別で、tail を null に設定する。

SLList

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

push(x) と pop() の実行時間は、いずれも明らかに $O(1)$ である。

3.1.1 キュー操作

SLList を使って、定数時間で add(x) と remove() を実行できる FIFO キュー操作も実装できる。削除についてはリストの先頭から行うので pop() と同じである。

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

一方、要素の追加はリストの末尾に対して行う。新たに加えるノードを u とすると、ほとんどの場合は tail.next = u とすればよい。しかし n = 0 の場合は特別で、tail = head = null とする [*2]。この場合、tail も head も u になる。

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

[*2] 訳注：tail が null の場合に tail.next にアクセスするとエラーとなるので別の対応が必要となる。

```
  }
  tail = u;
  n++;
  return true;
}
```

add(x) と remove() はいずれも明らかに定数時間で実行できる。

3.1.2 要約

次の定理は SLList の性能を整理したものである。

定理 **3.1.** *SLList* は、*Stack* と（*FIFO*）*Queue* インターフェースを実装する。push(x)、pop()、add(x)、remove() の実行時間はいずれも $O(1)$ である。

SLList で Deque の操作もほぼすべて実装できる。足りないのは SLList の末尾を削除する操作だ。SLList の末尾を削除するのは難しい。これは、新しい末尾を現在の末尾の 1 つ前のノードに設定しなければならないためである。末尾の 1 つ前のノード w では、w.next = tail となる。困ったことに、このような w を見つけるには、SLList の各ノードを head から順に n − 2 回辿っていかなければならない。

3.2 DLList: 双方向連結リスト

DLList（doubly-linked list、双方向連結リスト）は、SLList に似たノードの列である。違いは、ノード u が直後のノード u.next への参照だけでなく、直前のノード u.prev への参照も持っている点だ。

DLList

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

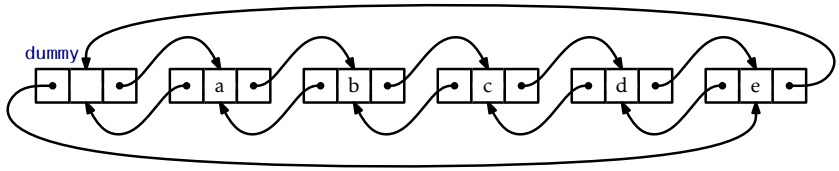


図 3.2: a,b,c,d,e からなる DLList

SLList の実装では特別扱いが必要なケースがいくつかあった。例えば、SLList の最後のノードを削除するときや、空の SLList にノードを追加するときは、head と tail をふつうと違うやり方で更新する必要があった。DLList では、このような特別なケースがさらに増える。そこで、DLList における特別なケースをシンプルに書くために、ダミーノードを使う。ダミーノードとは、データを含まず、ただ場所を占めるだけの空のノードであり、DLList で特別扱いが必要なノードをなくすための仕掛けである。具体的には、すべてのノードには next と prev に加えて dummy を持たせる。この dummy は、リストの最後のノードの直後にあり、かつ最初のノードの直前にあるとみなす。これにより、双方向連結リストでは、図 3.2 に示すようにノードが循環する。

DLList

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

DLList では、添字を指定して簡単にノードを見つけられる。先頭（dummy.next）から順方向に列を辿るか、末尾（dummy.prev）から逆方向に列を辿ればよい。こうすれば、i 番めのノードを見つけるのにかかる時間は $O(1+\min\{i, n-i\})$ である。

DLList

```
Node* getNode(int i) {
```

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

get(i) と set(i,x) も簡単である。i 番めのノードを見つけ、その値を読み書きすればよい。

DLList

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

これらの操作の実行時間は、i 番めのノードを見つける時間に大きく左右されるので、$O(1+\min\{i,n-i\})$ である。

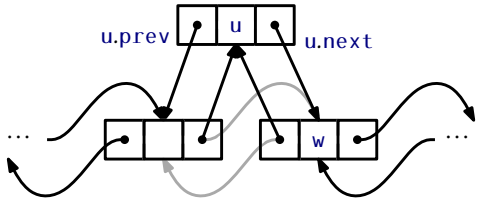


図 3.3: DLList において、u をノード w の直前に挿入する

3.2.1 追加と削除

DLList においてノード w の直前にノード u を追加したいとき、ノード w の参照がわかっているなら、u.next = w および u.prev = w.prev とし、u.prev.next と u.next.prev を適切に調整すればよい（図 3.3 参照）。ダミーノードがあるので、w.prev や w.next がない場合を特別扱いせずに済む。

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;
}
```

add(i,x) 操作の実装は自明だ。DLList の i 番めのノードを見つけ、データ x を持つ新しいノード u をその直前に挿入すればよい。

DLList

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

```
}
```

add(i,x) の処理のうち、実行時間が定数でないのは、(getNode(i) を使って) i 番めのノードを見つける処理だけだ。よって、add(i,x) の実行時間は $O(1+\min\{i,n-i\})$ である。

DLList からノード w を削除するのは簡単である。w.next と w.prev のポインタを w をスキップするように調整すればよい。ここでもダミーノードのおかげで複雑な場合分けの必要がなくなっている。

DLList

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

ここまでくると remove(i) も自明だ。i 番めのノードを見つけ、これを削除すればよい。

DLList

```
T remove(int i) {
  Node *w = getNode(i);
  T x = w->x;
  remove(w);
  return x;
}
```

remove(i) の実行時間は、getNode(i) によって i 番めのノードを見つける処理に左右されるので、$O(1+\min\{i,n-i\})$ である。

3.2.2 要約

次の定理は DLList の性能をまとめたものである。

定理 **3.2.** *DLList* は、*List* インターフェースを実装する。get(i)、set(i,x)、add(i,x)、remove(i) の実行時間はいずれも $O(1+\min\{i, n-i\})$ である。

DLList の操作の実行時間は、getNode(i) のコストを無視すると、いずれも定数時間である。つまり、DLList の操作において時間のかかる部分は、目的のノードを見つける処理だけだ。目的のノードさえ見つかれば、追加、削除、データの読み書きはいずれも定数時間で実行できる。

これは、2 章で説明した配列を使った List の実装とは対照的である。2 章では、目的のノードは定数時間で見つかるが、要素を追加したり削除したりするために配列内の要素をシフトする必要があり、結果として各処理は非定数時間であった。

このことからわかるように、連結リストは何か別の方法でノードの参照が得られるようなアプリケーションに適している。 例えば、連結リストのノードへの参照を USet に格納しておくという手法がある。そうすれば、連結リストから x を削除したい場合には x を含むノードを USet から素早く探し出し、当該のノードを連結リストから定数時間で削除できる。

3.3 SEList：空間効率の良い連結リスト

連結リストの欠点のひとつは、リストの中央付近の要素へのアクセスに時間がかかる点を除けば、メモリ使用量が多いことである。DLList のノードは、いずれも前後 2 つのノードへの参照を持つ。Node のフィールドのうち 2 つがリストを維持するために占められ、データを入れるのに使われるのが残りの 1 つだけなのである。

SEList（space-efficient list）は、この無駄な領域をシンプルなアイデアで削減するデータ構造だ。DLList のように要素を 1 個ずつノードに入れるのではなく、複数の要素を含むブロック（配列）をデータとしてノードに入れる。もう少し正確に説明しよう。SEList には**ブロックサイズ**を表す変数 b がある。SEList の個々のノードは、b + 1 個の要素を収容できる配列をデータとして持つのである。

あとで詳しく説明するが、個々のブロックに対して Deque の操作を実行できると都合がよい。そこで、BDeque（bounded deque）というデータ構造を使うことにする。BDeque は、2.4 節で説明した ArrayDeque に似たデータ構造だが、少しだけ異なる点がある。具体的には、新しい BDeque を作るときに用意する配列 a の大きさは b + 1 であり、その後は拡大も縮小もされない。

BDeque の重要な特徴は、先頭や末尾に対する要素の追加や削除を定数時間で実行できることだ。これは要素を他のブロックから移動するうえで都合がよい。

SEList
```
class BDeque : public ArrayDeque<T> {
public:
  BDeque(int b) {
    n = 0;
    j = 0;
    array<int> z(b+1);
    a = z;
  }
  ~BDeque() { }
  void add(int i, T x) {
    ArrayDeque<T>::add(i, x);
  }
  bool add(T x) {
    ArrayDeque<T>::add(size(), x);
    return true;
  }
  void resize() {}
};
```

SEList はブロックの双方向連結リストである。

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

SEList

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

3.3.1 必要なメモリ量

SEList では、ブロックに含められる要素数に次のような強い制限がある。すなわち、末尾以外のブロックはすべて b − 1 個以上 b + 1 個以下の要素を含む。つまり、SEList が n 要素を含むなら、ブロック数は次の値以下である [*3]。

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

末尾以外の各ブロックの BDeque は b − 1 個以上の要素を含むので、各ブロック内の無駄な領域は高々定数である。ブロックが使う余分なメモリも定数である。よって、SEList の無駄な領域は $O(b+n/b)$ である [*4]。b を $\sqrt{n}$ の定数倍にすれば、SEList の無駄な領域を 2.6.2 節で導出した下界に等しくすることができる。

3.3.2 要素を検索

SEList の最初の課題は、リストの i 番めの要素を見つけることである。要素の位置は次の 2 つから決まる。

1. i 番めの要素を含むブロックをデータとして持つノード u
2. そのブロックの中の要素の添字 j

SEList

```
class Location {
public:
  Node *u;
  int j;
  Location() { }
```

[*3] 訳注：1 が足されているのは $n = 0$ のような特殊な場合への対応と考えられる。
[*4] 訳注：最初の項 b は、末尾のブロック内の無駄な領域を表す。

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

ある要素を含むブロックを見つけるには、DLList のときと同じ方法を使う。つまり、目的のノードを先頭から順方向に、もしくは末尾から逆方向に探す。唯一の違いは、ノードからノードに移るたびにブロックをまるごとスキップできる点だ。

SEList

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

どのブロックにも少なくとも $b-1$ 個の要素が入っている（そうなっていないブロックは高々 1 つである）ことを考えれば、毎回のステップで、探している要素に最低でも $b-1$ 個ずつは近づいていく。よって、順方向に探索するときは、目的のノードに $O(1+i/b)$ ステップで到達する。一方、逆方向では $O(1+(n-i)/b)$ ステップである。この 2 つの値の小さいほうが、このアルゴリズムの実行時間を決める。つまり、i 番めの要素を特定するのに要する時間は $O(1+\min\{i,n-i\}/b)$ である。

i 番めの要素を含むブロックさえ特定できれば、あとは目的のブロックの中でノードを取得するなり設定するなりして `get(i)` もしくは `set(i,x)` を実行できる。

SEList

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

`get(i)` と `set(i,x)` の実行時間は、i 番めの要素を含むブロックを探す時間に左右されるので、$O(1+\min\{i,n-i\}/b)$ である。

3.3.3 要素の追加

SEList への要素の追加はもう少し複雑だ。一般的な場合を考える前に、より簡単な操作である末尾への要素の追加 `add(x)` を考えよう。末尾のブロックが一杯（あるいはそもそもブロックが 1 つもない）ときは、新しいブロックを割り当ててリストの末尾に追加する。すると、末尾のブロックが一杯でないこと

が保証されるので、x をその末尾のブロックに追加できる。

SEList

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

add(i,x) でリストの中に要素を追加しようとすると話が複雑になる。まず、リストにおける i 番めの要素が入るはずのノード u を特定する。ここで問題になるのは、ノード u のブロックがすでに b+1 個の要素を含んでおり、x を入れる隙間がない場合である。

$u_0, u_1, u_2, \ldots$ がそれぞれ u,u.next,u.next.next,... を表すとする。x を入れる余地があるノードがないか、$u_0, u_1, u_2, \ldots$ を探索する。この探索の過程で 3 つの場合が考えられる（図 3.4 参照）。

1. すぐ（$r+1 \le b$ ステップ以内）に、一杯でないブロックを持つノード u_r が見つかる。この場合は、r 回のシフトによって要素を次のブロックに移し、u_r の空いたスペースを u_0 に持ってくる。すると、x を u_0 のブロックに挿入できるようになる
2. すぐ（$r+1 \le b$ ステップ以内）に、ブロックのリストの末尾に到達する。この場合には、新しい空のブロックをリストの末尾に追加し、最初のケースと同様の処理を行う
3. b ステップ探しても空きがあるブロックが見つからない。この場合、$u_0, \ldots, u_{b-1}$ はいずれも b+1 個の要素を含むブロックの列である。新しいブロック u_b をこの列の直後に追加し、もともとあった $b(b+1)$ 個の要素を、$u_0, \ldots, u_b$ がいずれも b 個の要素を含むように分配する。すると、u_0 のブロックは b 個の要素しか含まないので、ここに x を挿入できる

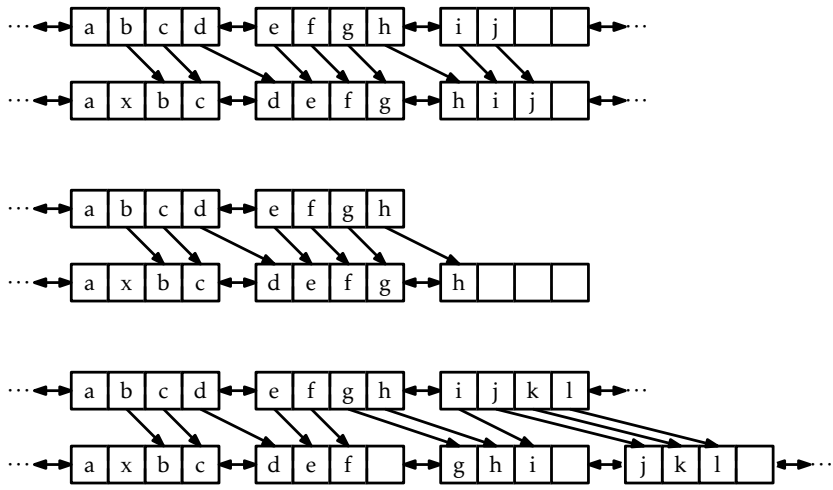


図 3.4: SEList に対する要素 x の追加で起こりえる 3 つの状況（この SEList ではブロックの大きさ b は 3 である）

SEList

```
void add(int i, T x) {
  if (i == n) {
    add(x);
    return;
  }
  Location l; getLocation(i, l);
  Node *u = l.u;
  int r = 0;
  while (r < b && u != &dummy && u->d.size() == b+1) {
    u = u->next;
    r++;
  }
  if (r == b) { // b+1 要素を含むブロックが b 個あった
    spread(l.u);
```

```
    u = l.u;
  }
  if (u == &dummy) { // 末尾まで到達したので新たなノードを加える
    u = addBefore(u);
  }
  while (u != l.u) { // 逆方向に要素をシフトする
    u->d.add(0, u->prev->d.remove(u->prev->d.size()-1));
    u = u->prev;
  }
  u->d.add(l.j, x);
  n++;
}
```

add(i,x) の実行時間は、上の 3 つの場合のどれが起きるかによって決まる。最初の 2 つの場合では、最大 b ブロックにわたって要素を探してシフトするので、実行時間は $O(b)$ である。3 つめの場合では、spread(u) を呼び出して $b(b+1)$ 個の要素を動かすので、実行時間は $O(b^2)$ である。3 つめの場合のコストを無視すれば、i 番めの位置に要素 x を挿入するときの実行時間は $O(b+\min\{i,n-i\}/b)$ である（3 つめの場合のコストはあとで償却法で説明する）。

3.3.4 要素の削除

SEList から要素を削除する操作は、要素を追加する操作に似ている。すなわち、まず i 番めの要素を含むノード u を特定し、そのうえで u から要素を削除した結果として u のブロックの要素数が $b-1$ より小さくなってしまう場合への対策が必要だ。

やはり u,u.next,u.next.next,... を $u_0,u_1,u_2,\ldots$ で表そう。u_0 のブロックの要素数を $b-1$ 以上にするため、$u_0,u_1,u_2,\ldots$ を順番に調べていって、要素を持ってくるノードを見つける。ここでも考えられる可能性は 3 つある（図 3.5 参照）。

1. すぐ（$r+1 \le b$ ステップ以内）に、$b-1$ より多くの要素を含むノードが見つかる。この場合は、r 回のシフトで要素をあるブロックから後方

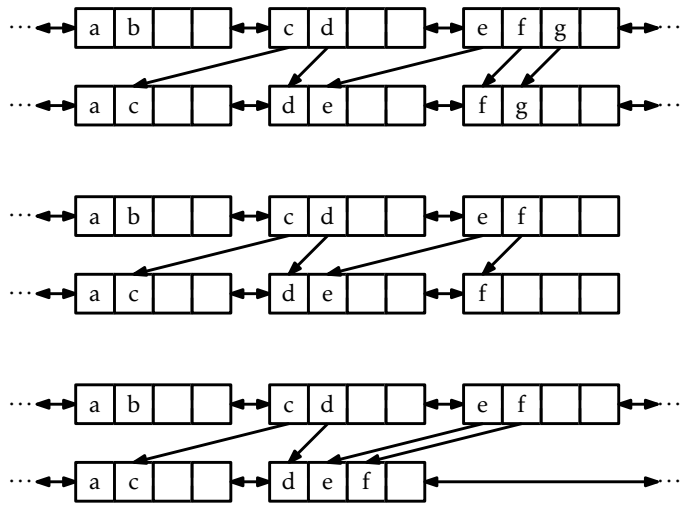


図 3.5: SEList に対する要素 x の削除で起こりえる 3 つの状況（この SEList ではブロックの大きさ b は 3 である）

のブロックに送り、u_r の余った要素を u_0 に持ってくる。すると、u_0 のブロックから目的の要素を削除できるようになる

2. すぐ（$r+1 \le b$ ステップ以内）に、リストの末尾に到達する。この場合、u_r は末尾のノードであり、そのブロックには $b-1$ 個以上の要素を含むという制約がない。そのため、1 つめの場合と同様に u_r から要素を借りてきて u_0 に足してよい。その結果として u_r のブロックが空になったら削除する
3. b ステップの間に $b-1$ 個より多くの要素を含むブロックが見つからない。この場合、$u_0,\ldots,u_{b-1}$ はいずれも要素数 $b-1$ のブロックの列である。そこで gather() を呼び、$b(b-1)$ 要素を $u_0,\ldots,u_{b-2}$ に集める。これらの $b-1$ 個のブロックは、いずれもちょうど b 個の要素を含むようになる。空になった u_{b-1} を削除すれば、u_0 のブロックが b 要素を含むようになるので、ここから適当な要素を削除すればよい

SEList

```
T remove(int i) {
```

```
  Location l; getLocation(i, l);
  T y = l.u->d.get(l.j);
  Node *u = l.u;
  int r = 0;
  while (r < b && u != &dummy && u->d.size() == b - 1) {
    u = u->next;
    r++;
  }
  if (r == b) { // b-1 要素を含むブロックが b 個あった
    gather(l.u);
  }
  u = l.u;
  u->d.remove(l.j);
  while (u->d.size() < b - 1 && u->next != &dummy) {
    u->d.add(u->next->d.remove(0));
    u = u->next;
  }
  if (u->d.size() == 0)
    remove(u);
  n--;
  return y;
}
```

3 つめの場合における gather(u) を無視すれば、add(i,x) と同様に、remove(i) の実行時間は $O(b+\min\{i,n-i\}/b)$ である。

3.3.5 spread と gather の償却解析

続いて、add(i,x) と remove(i) で実行される可能性がある gather(u) と spread(u) のコストを考える。はじめにコードを示す。

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);
}
```

いずれの実行時間でも、支配的なのは二段階ネストしたループである。内側と外側いずれのループも最大 $b+1$ 回実行されるので、これらの操作の実行時間はどちらも $O((b+1)^2) = O(b^2)$ である。しかし次の補題により、これらのメソッドは add(i,x) および remove(i) の呼び出し b 回につき多くとも 1 回しか呼ばれないことがわかる。

補題 **3.1.** 空の *SEList* が作られ、$m \geq 1$ 回の add(i,x) および remove(i) が実行されるとする。このとき、spread() および gather() に要する時間の合計は $O(bm)$ である。

証明. ここでは償却解析のためのポテンシャル法を使う。ノード u のブロックの要素数が b でないとき、u は**不安定（unstable）**であるという（すなわち、u は末尾のノードか、要素数が $b-1$ または $b+1$ である）。ブロックの要素数がちょうど b であるノードは**安定（stable）**であるという。SEList の**ポテンシャル**を不安定なノードの数で定義する。ここでは add(i,x) と spread(u) の呼び出し回数の関係だけを議論する。remove(i) と gather(u) の解析も同様である。

add(i,x) の 1 つめの場合分けでは、ブロックの大きさが変化するノードは u_r だけである。よって高々 1 つのノードだけが安定から不安定になる。2 つめの場合分けでは新しいノードが作られ、そのノードは不安定である。一方、他のノードの大きさは変わらず、不安定なノードの数は 1 つだけ増える。以上より、1 つめ、2 つめいずれの場合でも、SEList のポテンシャルの増加は高々 1 である。

最後に 3 つめの場合分けでは、$u_0,\ldots,u_{b-1}$ はいずれも不安定である。spread(u_0) が呼ばれると、これらの b 個の不安定なノードは $b+1$ 個の安定なノードに置き換えられる。そして x が u_0 のブロックに追加され、u_0 は不安定になる。ポテンシャルの減少は、合計で $b-1$ である。

まとめると、ポテンシャルは 0 から始まる（リストに 1 つもノードがない状態）。1 つめと 2 つめの場合分けでは、ポテンシャルは高々 1 増える。3 つめの場合分けでは、ポテンシャルは $b-1$ 減る。不安定なノードの数を表すポテンシャルが 0 より小さくなることはない。つまり、3 つめの場合分けのたびに、少なくとも $b-1$ 回、1 つめの場合分けと 2 つめの場合分けが起きる。以上より、spread(u) が呼ばれるたびに、少なくとも b 回は add(i,x) が呼ばれることが示された。 □

3.3.6 要約

次の定理は SEList の性能をまとめたものだ。

定理 **3.3.** *SEList* は、*List* インターフェースを実装する。spread(u) と gather(u) のコストを無視すると、b 個のブロックを持つ *SEList* の操作について次が成り立つ。

- get(i) と set(i,x) の実行時間は $O(1+\min\{i,n-i\}/b)$ である
- add(i,x) と remove(i) の実行時間は $O(b+\min\{i,n-i\}/b)$ である

さらに、空の *SEList* から始めて、add(i,x) および remove(i) からなる m 個

の操作の列における spread(u) と gather(u) の実行時間は、合計で $O(bm)$ である。

要素数 n の *SEList* の空間使用量は、ワード単位で測ると $n + O(b + n/b)$ である [*5]。

SEList により、ArrayList か DLList かのトレードオフを調整できる。つまり、2 つのデータ構造のどちらに寄せるかを、ブロックの大きさ b によって調整できるのである。一方の極端な状況は $b = 2$ の場合であり、このとき SEList のノードは最大で 3 つの値を持ち、これは DLList と同じである。もう一方の極端な状況は $b > n$ の場合であり、このときすべての要素が 1 つの配列に格納され、これは ArrayList のようなものである。これらの両極端な状況の間で、リストに対する要素の追加と削除にかかる時間と、特定の要素を見つけるのにかかる時間とのトレードオフを考えることになる。

3.4 ディスカッションと練習問題

単方向連結リストも双方向連結リストも 40 年以上前からプログラムで使われており、研究され尽くされた技法である。例えば、Knuth の [46, Sections 2.2.3–2.2.5] で議論されている。SEList もデータ構造における有名な練習問題である。SEList は unrolled linked list[67] と呼ばれることもある。

双方向連結リストの空間使用量を減らすための別な手法として、XOR-list と呼ばれるものもある。XOR-list では、各ノード u はポインタの代わりに u.nextprev だけを持つ。 このポインタの代わりとなる値は、u.prev と u.next の XOR（排他的論理和）を取ったものである。リストでは、dummy を指すポインタと dummy.next を指すポインタの 2 つが必要だ（dummy.next はリストが空なら dummy を、そうでないなら先頭のノードを指す）。この技法では、u と u.prev があれば、u.next を次の関係式から計算できることを利用している（^は 2 つの引数の排他的論理和とする）。

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

この技法の欠点は、コードが少し複雑になること、Java や Python などのガーベッジコレクションのある言語では使えないことである。 XOR-list についてのさらに踏み込んだ議論は、Sinha の雑誌記事 [68] を参照してほしい。

[*5] メモリの計測方法については 1.4 節の説明を参照。

問 **3.1.** SLList において、push(x)、pop()、add(x)、remove() の特殊なケースすべてをダミーノードを使って避けられないのはなぜか。

問 **3.2.** SLList のメソッド secondLast() を設計、実装せよ。これは SLList の末尾の 1 つ前の要素を返すメソッドである。リストの要素数 n を使わずに実装してみよ。

問 **3.3.** SLList の get(i)、set(i,x)、add(i,x)、remove(i) を実装せよ。いずれの操作の実行時間も $O(1+i)$ であること。

問 **3.4.** SLList の reverse() 操作を設計、実装せよ。これは SLList の要素の順番を逆にする操作である。この操作の実行時間は $O(n)$ でなければならず、再帰を使ってはならない。また、他のデータ構造を補助的に使ったり、新しいノードを作ったりしてもいけない。

問 **3.5.** SLList および DLList のメソッド checkSize() を設計、実装せよ。これはリストを辿り、n の値がリストに入っている要素の数と一致するかを確認するメソッドである。このメソッドは何も返さないが、もし要素数が n と一致しなければ例外を投げる。

問 **3.6.** addBefore(w) を再実装せよ。これはノード u を作り、それをノード w の直前に追加する操作だ。本章のコードは参照しないこと。本章のコードと完全に一致しなくても、正しいコードになっている可能性はある。自分が書いたコードをテストし、正しく動くかどうかを確認せよ。

以降の問題は、DLList の操作に関連するものだ。これらの問題では、新しいノードや一時的な配列を割り当ててはいけない。これらの問題は、いずれもノードの prev と next を書き換えるだけで解ける。

問 **3.7.** DLList のメソッド isPalindrome() を実装せよ。これはリストが**回文**であるとき true を返す。すなわち、任意の $i \in \{0,\ldots,n-1\}$ について、i 番めの要素と $n-i-1$ 番めの要素が等しいかどうかを確認する。実行時間は $O(n)$ であること。

問 **3.8.** メソッド rotate(r) を実装せよ。これは DLList の要素を回転し、i 番めの要素を $(i+r) \bmod n$ 番めの位置に移動する。実行時間は $O(1+\min\{r,n-r\})$ であること。リスト内のノードを修正してはならない。

問 **3.9.** メソッド truncate(i) を実装せよ。これは DLList を i 番めで切り

詰める。このメソッドを実行すると、リストの要素数は i になり、$0,\ldots,i-1$ 番めの要素だけが残る。返り値は、$i,\ldots,n-1$ 番めの要素を含む別の DLList である。実行時間は $O(\min\{i,n-i\})$ であること。

問 **3.10.** DLList のメソッド absorb(12) を実装せよ。引数として別の DLList（12）を取り、12 を空にしてその中身を自分の要素として追加する。例えば、11 が a,b,c を含み、12 が d,e,f を含むとき、11.absorb(12) を実行すると、11 は a,b,c,d,e,f を含み、12 は空になる。

問 **3.11.** deal() を実装せよ。これは DLList から偶数番めの要素を削除し、それらの要素を含む DLList を返す操作だ。例えば 11 が a,b,c,d,e,f を含むとき、11.deal() を呼ぶと、11 の要素は a,c,e になり、b,d,f を含むリストが返される。

問 **3.12.** メソッド reverse() を実装せよ。これは DLList の要素の順序を逆転する。

問 **3.13.** この問題では DLList を整列するマージソートというアルゴリズムを実装することになる。マージソートは 11.1.1 節で扱う。

1. DLList のメソッド takeFirst(12) を実装せよ。これは 12 の先頭ノードを取り出してレシーバーに追加する。新しいノードを作らないことを除けば、add(size(),12.remove(0)) と等価である。
2. DLList の静的メソッド merge(11,12) を実装せよ。これは 2 つの整列済みのリスト 11 と 12 を統合し、その結果を含む新たな整列済みリストを返す。この操作により 11 と 12 は空になる。例えば、11 の要素が a,c,d、12 の要素が b,e,f であるとき、このメソッドは a,b,c,d,e,f を含むリストを返す。
3. DLList のメソッド sort() を実装せよ。これはマージソートを使ってリストのすべての要素を整列する。この再帰的なアルゴリズムは次のように動作する。
   （a）リストの要素数が 0 または 1 なら何もしない
   （b）そうでないなら、truncate(size()/2) によって、リストをほぼ等しい大きさの 2 つのリスト 11 と 12 に分割する
   （c）再帰的に 11 を整列する
   （d）再帰的に 12 を整列する
   （e）最後に 11 と 12 を統合して 1 つの整列済みリストとする

以降の問題は発展的なもので、要素が追加および削除される際に Stack と Queue の最小値がどうなるかについての理解が必要になる。

問 **3.14.** データ構造 MinStack を設計、実装せよ。これは比較可能な要素を持ち、スタックの操作 push(x)、pop()、size() をサポートし、min() 操作も可能なデータ構造である。min() はデータ構造に入っている要素のうち最小の値を返す。すべての操作の実行時間は定数であること。

問 **3.15.** データ構造 MinQueue を設計、実装せよ。これは比較可能な要素を持ち、キューの操作 add(x)、remove()、size() をサポートし、min() 操作も可能なデータ構造である。すべての操作の償却実行時間は定数であること。

問 **3.16.** データ構造 MinDeque を設計、実装せよ。これは比較可能な要素を持ち、双方向キューの操作 addFirst(x)、addLast(x)、removeFirst()、removeLast()、size() をサポートし、min() 操作も可能なものである。すべての操作の償却実行時間は定数であること。

以降の問題は、空間効率の良い SLList の解析と実装について理解度を測るためのものである。

問 **3.17.** SEList が Stack のように使われるとき、つまり SEList が $\texttt{push}(x) \equiv \texttt{add}(\texttt{size}(), x)$ と $\texttt{pop}() \equiv \texttt{remove}(\texttt{size}() - 1)$ によってのみ更新されるとき、これらの操作の償却実行時間がいずれも b の値に依存しない定数であることを証明せよ。

問 **3.18.** Deque の操作をすべてサポートし、いずれの償却実行時間も b に依存しない定数であるような SEList を設計、実装せよ。

問 **3.19.** ビット単位の排他的論理和^によって 2 つの int 型の値を入れ替える方法を説明せよ。ただし、その際に 3 つめの変数を使ってはならないものとする。

第 4

スキップリスト

この章ではスキップリストという素晴らしいデータ構造を紹介する。スキップリストはさまざまな形で活用できる。例えば、get(i)、set(i,x)、add(i,x)、remove(i) の実行時間がいずれも $O(\log n)$ であるような List を実装できる。また、すべての操作の期待実行時間が $O(\log n)$ であるような SSet もスキップリストで実装できる。

スキップリストが効率的なのはランダム性を利用していることによる。新しい要素を追加するとき、スキップリストでは、ランダムなコイントスの結果に応じて要素の高さを決める。スキップリストの性能評価には、実行時間の期待値と、要素を見つけるための経路の長さの期待値を使う。期待値で評価するのは、スキップリストではコイントスにより決まる確率を利用するからだ。スキップリストで使うコイントスは、実際には擬似乱数生成器を使ってシミュレーションする。

4.1 基本的な構造

スキップリストは、単方向連結リスト $L_0,\ldots,L_h$ を並べたものだと考えられる。n 個の要素を含むスキップリストを例に考えてみよう。1 つめの単方向連結リストである L_0 には、それら n 個の要素をすべて含める。その L_0 から一部の要素を取り出して L_1 を作る。さらに L_1 から L_2 を作る。これを繰り返す。L_{r-1} の要素を L_r に含めるかどうかは、各要素についてコインを投げて決める。投げたコインが表なら、その要素を L_r に含める。最終的に L_r が空になったら繰り返しを終える。スキップリストの例を図 4.1 に示す。

スキップリストの各要素 x について、x を含む単方向連結リスト L_r の添字

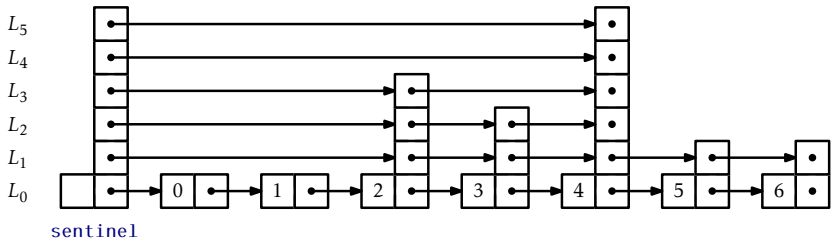


図 4.1: 7 つの要素を含むスキップリストの例

r のうち最大のものを、x の**高さ（height）**と定義する。例えば、x が L_0 だけに含まれているなら、x の高さは 0 である。少し考えてみればわかるように、x は、裏が出るまでにコインを繰り返し投げた回数である。それまでに表は何回出ただろうか。この問いの答え、そして x の高さの期待値は、当然 1 である（コイントスの回数の期待値としては 2 だが、最後のトスは表ではないので、表が出た回数の期待値は 1 になる）。スキップリストの各要素の高さのうち、最も高いものを、そのスキップリストの**高さ**と定義する。

すべてのリストの先頭には、**番兵（sentinel）**と呼ばれる特別なノードを置く。これは、当該のリストに対するダミーノードとして機能する。スキップリストの重要な性質は、L_h の番兵から L_0 の各ノードまでの短いパスが存在することだ。このパスを**探索経路（search path）**と呼ぶ。あるノード u への探索経路は簡単に構築できる。左上端のノード（つまり L_h の番兵）からスタートし、u に到達したり u を通り越したりしない限り、右へと進み続ける。u に到達、もしくは u を通り越してしまうときは、右ではなく下に進む（図 4.2 参照）。

もう少し正確に説明する。L_h の番兵 w から L_0 のノード u への探索経路を見つけるには、まず w.next を見て、これが L_0 の中で u より前にあれば w = w.next とする。そうでなければ 1 つ下のリストに降りて、L_{h-1} における w から処理を続ける。これを L_0 における u の直前の要素に辿り着くまで繰り返す。

このような探索経路は、次の補題が示すように、かなり短い（この補題は 4.4 節で証明する）。

補題 **4.1.** L_0 のノード u について、u の探索経路の長さの期待値は $2\log n + O(1) = O(\log n)$ である。

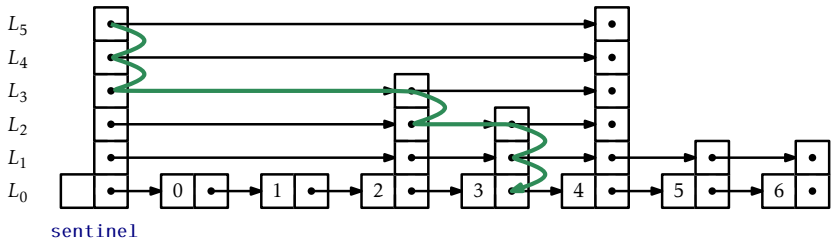


図 4.2: あるスキップリストにおける、4 を含むノードの探索経路

空間効率の良い方法でスキップリストを実装するには、ノード u がデータ x とポインタの配列 next を含むようにし、u.next[i] が L_i における u の次のノードを指すようにすればよい。x は複数のリストに現れることがあるが、こうすればノードとしての実体は 1 つだけあれば済む。

SkiplistSSet
```
struct Node {
  T x;
  int height;
  Node *next[];
};
```

以降の 2 つの節ではスキップリストの応用を紹介する。いずれの応用でも、主な構造（リストや整列された集合）は L_0 に入れる。2 つの応用の違いは、探索経路の辿り方である。特に、L_r から下（L_{r-1}）に降りるか L_r の中でそのまま右に進むかを決める方法が異なる。

4.2 SkiplistSSet：効率的な SSet

SkiplistSSet は、スキップリストを使った SSet インターフェースの実装である。SSet インターフェースの実装にスキップリストを使う場合、L_0 には SSet の要素を整列して格納する。探索経路に沿って y ≥ x を満たすような最小の y を探すメソッド find(x) を下記に示す。

SkiplistSSet
```
Node* findPredNode(T x) {
  Node *u = sentinel;
  int r = h;
  while (r >= 0) {
    while (u->next[r] != NULL
                && compare(u->next[r]->x, x) < 0)
      u = u->next[r]; // リスト r の中で右に進む
    r--; // リスト r-1 に下がる
  }
  return u;
}
T find(T x) {
  Node *u = findPredNode(x);
  return u->next[0] == NULL ? null : u->next[0]->x;
}
```

y までの探索経路を辿るのは簡単だ。L_r の中のノード u にいるとしたら、まず右隣 u.next[r].x を見る。x > u.next[r].x なら、L_r の中で右に進む。そうでないなら、L_{r-1} に下がる。各ステップ（右または下に進む）は一定の時間で実行できる。よって、補題 4.1 より、find(x) の期待実行時間は $O(\log n)$ である。

SkipListSSet に要素を追加する方法を考えるには、新しいノードの高さ k を決めるコイントスのシミュレート方法を考える必要がある。これには、ランダムな整数 z を生成し、z の二進表記において連続する 1 の数を数える [*1]。

SkiplistSSet
```
int pickHeight() {
  int z = rand();
  int k = 0;
```

[*1] k は int のビット数より常に小さくなるから、この方法でコイントスを完全には再現できない。しかし、要素数が $2^{32} = 4294967296$ を超える場合でもない限り、この影響は無視できるほど小さい。

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

SkiplistSSet の add(x) を実装するには、x を入れる場所を見つけ、高さ k を pickHeight() で決め、x を $L_0,\ldots,L_k$ に継ぎ足す。これを最も簡単に実現するため、配列 stack を用意し、探索経路においてリスト L_r からリスト L_{r-1} へと下がる箇所に該当するノードを記録しておく。より正確に言うと、探索経路において L_r から L_{r-1} へと下がるときの L_r のノードを stack[r] とする。こうすると、x の挿入時に修正が必要なノードが、まさに stack[0],...,stack[k] になる（図 4.3）。このアルゴリズムを利用した add(x) の実装を下記に示す。

SkiplistSSet

```
bool add(T x) {
  Node *u = sentinel;
  int r = h;
  int comp = 0;
  while (r >= 0) {
    while (u->next[r] != NULL
               && (comp = compare(u->next[r]->x, x)) < 0)
      u = u->next[r];
    if (u->next[r] != NULL && comp == 0)
      return false;
    stack[r--] = u;         // u を入れて下に進む
  }
  Node *w = newNode(x, pickHeight());
  while (h < w->height)
    stack[++h] = sentinel; // 高さが増える
```

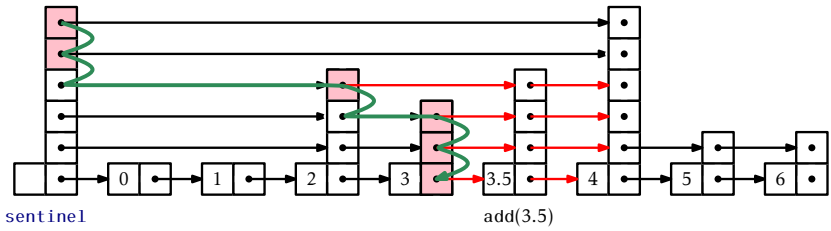


図 4.3: 値 3.5 を含むノードを skiplist に追加する。stack に格納されるノードを強調している

```
  for (int i = 0; i <= w->height; i++) {
    w->next[i] = stack[i]->next[i];
    stack[i]->next[i] = w;
  }
  n++;
  return true;
}
```

要素 x の削除も同様だ。ただし、削除では stack を使って探索経路を記録する必要はない。削除は探索経路を辿っていく途中でできる。x を探す途中にノード u から下に向かうとき、u.next.x = x であるなら u を繋ぎ替えればよい（図 4.4）。

SkiplistSSet

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

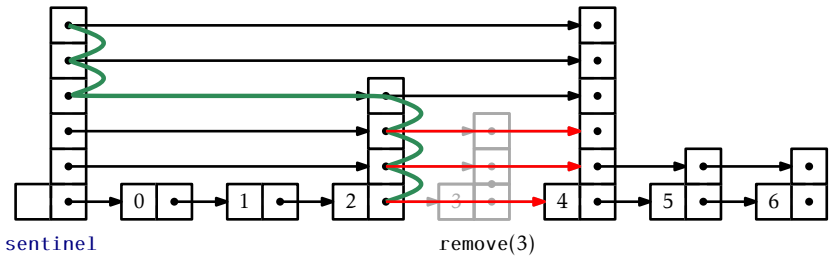


図 4.4: 値 3 を含むノードを skiplist から削除する

```
    if (u->next[r] != NULL && comp == 0) {
      removed = true;
      del = u->next[r];
      u->next[r] = u->next[r]->next[r];
      if (u == sentinel && u->next[r] == NULL)
        h--; // スキップリストの高さを小さくする
    }
    r--;
  }
  if (removed) {
    delete del;
    n--;
  }
  return removed;
}
```

4.2.1 要約

定理 4.1 は、全順序集合（SSet）の実装にスキップリストを使った場合の性能について要約したものだ。

定理 4.1. *SkiplistSSet* は *SSet* インターフェースの実装である。

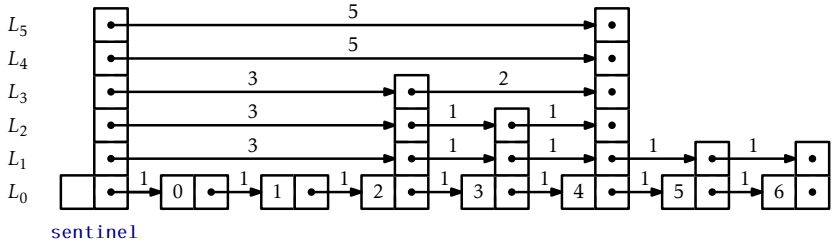


図 4.5: skiplist における辺の長さ

SkiplistSSet における add(x)、remove(x)、find(x) の実行時間の期待値は、いずれも $O(\log n)$ である。

4.3 SkiplistList：効率的なランダムアクセス List

SkiplistList は、スキップリストを使った List インターフェースの実装である。SkiplistList では、L_0 に、リスト List の要素がリストにおける順番通りに含まれる。SkiplistSSet と同様に、要素の追加、削除、読み書きのいずれの実行時間も $O(\log n)$ である。

これを可能にするためには、まず L_0 における i 番めの要素を見つける方法が必要だ。これを最も簡単に実現するため、リスト L_r の各辺について**長さ**を定義する。L_0 については、どの辺の長さも 1 とする。L_r（$r > 0$）については、辺 e の長さを、L_{r-1} において e の下にある辺の長さの和とする。これは、e の下にある L_0 の辺の数と等しい。あるスキップリストに辺を併記した例を図 4.5 に示す。スキップリストの辺は配列に格納されているので、その長さも同様にして格納すればよい。

SkiplistList

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

この定義には、L_0 において j 番めのノードから長さ ℓ の辺を辿ると、L_0 において $j+\ell$ のノードに移るという良い性質がある。これを利用すれば、探索経路を辿りながら L_0 におけるインデックス j を算出できる。L_r のノード u にいるとき、辺 u.next[r] の長さと j の和が i より小さいなら右に進む。そうでないなら下（L_{r-1}）に進む。

SkiplistList

```
Node* findPred(int i) {
  Node *u = sentinel;
  int r = h;
  int j = -1;   // リスト 0 における現在のノードのインデックス
  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;
}
```

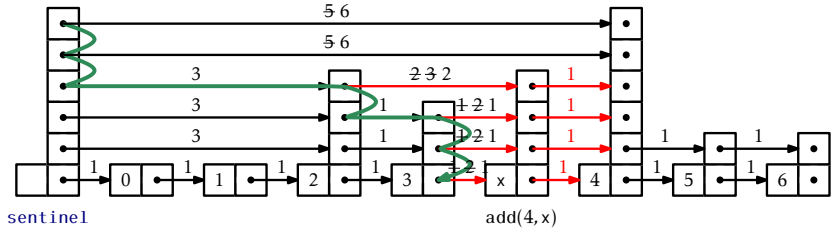


図 4.6: SkiplistList への要素の追加

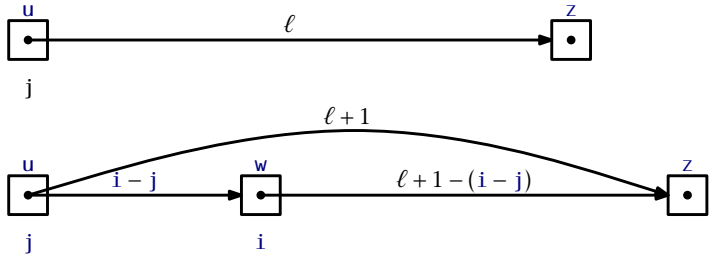


図 4.7: ノード w を skiplist に追加するときの、辺の長さの更新

get(i) および set(i,x) において最も計算時間がかかるのは L_0 の i 番めのノードを見つける処理なので、get(i) および set(i,x) 操作の実行時間は $O(\log n)$ である。

SkiplistList の i 番めの位置に要素を追加するのは簡単だ。SkiplistSSet とは違い、新しいノードが必ず追加されるので、ノードの位置を見つける処理とノードを追加する処理とを同時に実行できる。まずは新たに挿入するノード w の高さ k を決め、i の探索経路を辿る。L_r から下に進むのは $r \le k$ のときで、このとき w を L_r に継ぎ足す。その際には辺の長さを適切に更新する必要があることにも注意する（図 4.6 参照）。

探索経路上でリスト L_r のノード u に降りたときは、i 番めの位置に要素を追加するため、辺 u.next[r] の長さを 1 つ大きくする。ノード w を 2 つのノード u と z の間に追加する様子を図 4.7 に示す。L_0 において u が何番めなのかは、探索経路を辿りながら数えられる。そのため、u から w までの辺の長さは $i - j$ とわかる。さらに、u から z への辺の長さ ℓ から、w から z への辺の長さを計算できる。こうして w を挿入し、関連する辺の長さの更新を定数時間で終えることができる。

複雑そうに聞こえるかもしれないが、実際のコードはとても単純である。

SkiplistList

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

SkiplistList

```
Node* add(int i, Node *w) {
  Node *u = sentinel;
  int k = w->height;
  int r = h;
  int j = -1; // u のインデックス
  while (r >= 0) {
    while (u->next[r] != NULL && j + u->length[r] < i) {
      j += u->length[r];
      u = u->next[r];
    }
    u->length[r]++; // 新たなノードがリスト 0 において何番めなのかを数える
    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++;
```

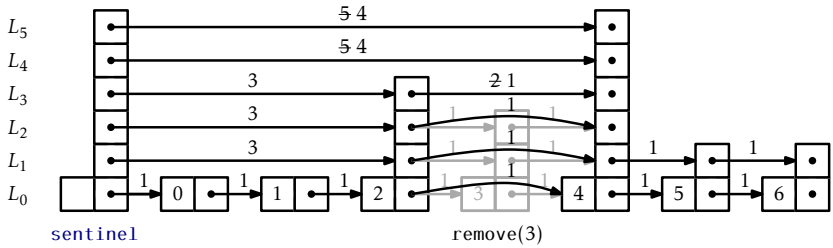


図 4.8: SkiplistList から要素を削除する

```
  return u;
}
```

ここまでの話から、SkiplistList における remove(i) の実装は明らかである。i 番めの位置への探索経路を辿る。高さ r のノード u から経路が下に向かうとき、その高さにおける u から出る辺の長さを 1 つ小さくする。また、u.next[r] が高さ i の要素であるかどうかを確認し、もしそうならリストから取り除く（図 4.8）。

SkiplistList

```
T remove(int i) {
  T x = null;
  Node *u = sentinel, *del;
  int r = h;
  int j = -1; // u のインデックス
  while (r >= 0) {
    while (u->next[r] != NULL && j + u->length[r] < i) {
      j += u->length[r];
      u = u->next[r];
    }
    u->length[r]--; // ノードを削除するので、辺の長さを減らす
    if (j + u->length[r] + 1 == i && u->next[r] != NULL) {
      x = u->next[r]->x;
```

```
      u->length[r] += u->next[r]->length[r];
      del = u->next[r];
      u->next[r] = u->next[r]->next[r];
      if (u == sentinel && u->next[r] == NULL)
        h--;
    }
    r--;
  }
  deleteNode(del);
  n--;
  return x;
}
```

4.3.1 要約

定理 4.2 は、SkiplistList の性能を要約したものだ。

定理 **4.2.** *SkiplistList* は *List* インターフェースを実装する。*Skiplist-List* における get(i)、set(i,x)、add(i,x)、remove(i) の実行時間の期待値は、いずれも $O(\log n)$ である。

4.4 スキップリストの解析

この節では、スキップリストの高さ、大きさ、探索経路の長さの期待値を解析する。基本的な確率論の知識を前提とする。いくつかの証明では、次に述べるコイントスについての考察を利用する。

補題 **4.2.** T を、表裏が等しい確率で出るコインを投げて、表が出るまでに要するコイントスの回数とする（表が出た回数も含まれる点に注意）。このとき、$\mathrm{E}[T] = 2$ である。

証明. 表が出るまでコイントスを繰り返すとき、次のインジケータ確率変数を

定義する。

$$I_i = \begin{cases} 0 & \text{コイントスが } i \text{ 回よりも少ないとき} \\ 1 & \text{コイントスが } i \text{ 回以上のとき} \end{cases}$$

$I_i = 1$ は、最初の $i-1$ 回の結果がいずれも裏であることと同値である。よって、$\mathrm{E}[I_i] = \Pr\{I_i = 1\} = 1/2^{i-1}$ である。コイントスの合計回数 T は、$T = \sum_{i=1}^{\infty} I_i$ と書ける。以上より、次のことがわかる。

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

□

次の 2 つの補題では、スキップリストにおけるノード数が要素数に概ね比例することを示す。

補題 4.3. n 要素からなるスキップリストにおける（番兵を除く）ノード数の期待値は、$2n$ である。

証明. 要素 x がリスト L_r に含まれる確率は $1/2^r$ である。よって、L_r のノード数の期待値は $n/2^r$ である [*2]。以上より、すべてのリストに含まれるノードの総数の期待値が求まる。

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

□

補題 4.4. n 要素を含むスキップリストの高さの期待値は $\log n + 2$ 以下である。

証明. $r \in \{1, 2, 3, \ldots, \infty\}$ について次の確率変数を定義する。

$$I_r = \begin{cases} 0 & L_r \text{ が空のとき} \\ 1 & L_r \text{ が空でないとき} \end{cases}$$

[*2] インジケータ確率変数と期待値の線形性からこの結果を得る方法は 1.3.4 節を参照せよ。

スキップリストの高さ h は次のように計算できる。

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

I_r はリスト L_r の長さ $|L_r|$ を超えないことに注意する。

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

以上より、次のことがわかる。

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

□

補題 **4.5.** n 要素からなるスキップリストのノード数の期待値は、番兵を含めて $2n + O(\log n)$ である。

証明. 補題 4.3 より、番兵を除いたノード数の期待値は $2n$ である。番兵の数の期待値はスキップリストの高さ h に等しく、これは補題 4.4 より $\log n + 2 = O(\log n)$ 以下である。 □

補題 **4.6.** スキップリストにおける探索経路の長さの期待値は $2\log n + O(1)$ 以下である。

証明. 最も簡単な証明方法は、ノード x の**逆探索経路**を考えることだ。この逆探索経路は、L_0 における x の直前のノードから始まる。パスが上に向かえるときは上に向かう。そうでないなら左に進む。少し考えると、x の逆探索経路は、方向が逆であることを除いて探索経路と同じになることがわかるだろう。

ある高さで逆探索経路が通過するノードの数 r は、コインを投げて表が出れば上に向かってから停止し、裏が出れば左に向かい試行を続ける、という試行に関連している。すなわち、表が出るまでにコインを投げる回数は、逆探索経路において、ある高さで左に向かうステップの数に対応している [*3]。補題 4.2 より、初めて表が出るまでのコイントスの回数の期待値は 1 である。

（順方向の）探索経路において、高さ r で右に進む回数を S_r で表す。すると、$\mathrm{E}[S_r] \leq 1$ である。さらに、L_r では L_r の長さよりも多く右に進むことはないので、$S_r \leq |L_r|$ である。よって次の式が成り立つ。

$$\mathrm{E}[S_r] \leq \mathrm{E}[|L_r|] = n/2^r$$

あとは補題 4.4 と同様に証明を完成すればよい。S をスキップリストにおけるノード u の探索経路の長さとする。また、h をそのスキップリストの高さとする。このとき、次の式が成り立つ。

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

□

[*3] 実際には、最初に表が出る、もしくは探索経路で番兵に出くわす場合に試行が終了するはずなので、左に向かう回数を多く数えてしまう可能性がある。しかし、いま考えている補題は上界に関するものなので問題はない。

次の定理にこの節の結果をまとめる。

定理 **4.3.** n 要素を含むスキップリストの大きさの期待値は $O(n)$ である。ある要素の探索経路の長さの期待値は $2\log n + O(1)$ 以下である。

4.5 ディスカッションと練習問題

スキップリストを導入したのは Pugh [60] である。Pugh はスキップリストの拡張や応用も数多く提案した [59]。それ以来、スキップリストについては広く研究されている。スキップリストの i 番めの要素を見つける探索経路の長さの期待値や分散については、複数の研究 [45, 44, 56] でさらに正確に解析されている。決定的な（ランダム性を用いない）変種 [53]、偏りのある変種 [8, 26]、適応的な変種 [12] も開発されている。スキップリストはさまざまな言語やフレームワークで書かれ、またオープンソースのデータベースシステムでも使われている [69, 61]。オペレーティングシステム HP-UX におけるカーネルのプロセス制御の構造として、スキップリストの変種が使われている [42]。

問 **4.1.** 図 4.1 のスキップリストにおける 2.5 と 5.5 の探索経路を説明せよ。

問 **4.2.** 図 4.1 のスキップリストに対して値 0.5 の要素を高さ 1 で追加し、その後、値 3.5 の要素を高さ 2 で追加するときの振る舞いを説明せよ。

問 **4.3.** 図 4.1 のスキップリストから 1 と 3 を削除するときの振る舞いを説明せよ。

問 **4.4.** 図 4.5 の `SkiplistList` に `remove(2)` を実行するときの振る舞いを説明せよ。

問 **4.5.** 図 4.5 の `SkiplistList` に `add(3, x)` を実行するときの振る舞いを説明せよ。なお、`pickHeight()` は新たなノードの高さとして 4 を選択すると仮定せよ。

問 **4.6.** `add(x)` または `remove(x)` を実行するとき、`SkiplistSet` のポインタのうち操作されるものの数の期待値は定数であることを示せ。

問 **4.7.** ある要素を L_{i-1} から L_i へ昇格させるときにコイントスを使わず、確率 $p (0 < p < 1)$ を利用するとする。

1. このとき、探索経路の長さの期待値は $(1/p)\log_{1/p} n + O(1)$ 以下であることを示せ。
2. これを最小にする p を求めよ。
3. スキップリストの高さの期待値を求めよ。
4. スキップリストのノード数の期待値を求めよ。

問 **4.8.** SkiplistSet の find(x) では**冗長な比較**を行うことがある。具体的には、x と同じ値を複数回にわたって比較することがある。このような冗長な比較は、u.next[r] = u.next[r − 1] を満たすノード u が存在すると発生する。どのようにして冗長な比較が発生するかを説明し、find(x) において冗長な比較が発生しないようにする方法を示せ。そして、このように修正した find(x) メソッドによる比較の回数を解析せよ。

問 **4.9.** SSet の要素であって x より小さいものの個数を SSet における要素 x のランクと呼ぶ。SSet インターフェースを実装するスキップリストを、ランクによる要素への高速アクセスが可能になるように設計、実装せよ。ランク i の要素を返す get(i) も実装せよ。get(i) 操作の実行時間は $O(\log n)$ とすること。

問 **4.10.** 探索経路において下に向かうノードを保存した配列のことを、スキップリストの**指**（finger）と呼ぶ（93 ページのコードで、add(x) における変数 stack は**指**である。図 4.3 の網掛のノードが指を表している）。指は、L_0 における経路を示していると解釈することもできる。

指探索（finger search）は、指を利用した find(x) の実装である。u.x < x かつ (u.next = null *or* u.next.x > x) を満たすノード u に到達するまでは指の要素を見ていき、u からふつうの x の探索を実行する。L_0 において b から指が指示する値までの間にある値の数を r とするとき、**指探索**のステップ数の期待値は $O(1 + \log r)$ である。

内部で指を利用して find(x) を実装した Skiplist のサブクラス、SkiplistWithFinger を実装せよ。このサブクラスでは指を保持し、find(x) を指探索として実装するためにその指を使う。find(x) では、x の位置を返すと同時に、指が常に前回の find(x) の結果を指示するように更新する。

問 **4.11.** SkiplistList を i 番めの位置で切り詰める truncate(i) メソッドを実装せよ。このメソッドを実行すると、リストの大きさは i になり、リストは添字 0,...,i − 1 の要素のみを含むようになる。このメソッドの返り値は、

添字 i,...,n−1 の要素を含む SkiplistList である。このメソッドの実行時間は $O(\log n)$ でなければならない。

問 **4.12.** SkiplistList を引数に取り、引数の SkiplistList を空にして、その要素をレシーバーにそのままの順番で追加するメソッド absorb(skiplistlist) を実装せよ。例えば、スキップリスト l1 の要素が a,b,c、スキップリスト l2 の要素が d,e,f であるとき、l1.absorb(l2) を呼ぶと、l1 の要素は a,b,c,d,e,f になり、l2 は空になる。このメソッドの実行時間は $O(\log n)$ でなければならない。

問 **4.13.** SEList のアイデアを転用し、空間効率の良い SSet である SESSet を設計、実装せよ。要素を順に SEList に格納し、この SEList のブロックを SSet に格納すればよい。オリジナルの SSet の実装で n 要素の保持に $O(n)$ のメモリを使うとしたら、SESSet では n 要素を格納するためのメモリに加えて $O(n/b+b)$ だけの余分な空間を使うことになるだろう。

問 **4.14.** SSet を使って（大きな）テキストを読み込み、そのテキストの任意の部分文字列をインタラクティブに検索できるアプリケーションを設計、実装せよ。ユーザーがクエリを入力したら、テキストのマッチしている部分があればそれを結果として返すこと。
ヒント 1：任意の部分文字列は、ある接尾辞の接頭辞である。よって、テキストファイルのすべての接尾辞を保存すれば十分である。
ヒント 2：任意の接尾辞は、テキストの中のどこから接尾辞が始まるかを表す 1 つの整数で簡潔に表現できる。
書いたアプリケーションを長いテキストでテストせよ。プロジェクト Gutenberg [1] から書籍のテキストを入手できる。正しく動いたらレスポンスを速くしよう。すなわち、キー入力から結果が得られるまでに要する時間を認識できないくらい小さくしよう。

問 **4.15.** （この練習問題は、6.2 節で二分探索木について学んでから取り組むこと。）
スキップリストを二分探索木と比較せよ。

1. スキップリストから辺をいくつか削除すると二分木のような構造が得られること、および、二分探索木に似ることを説明せよ。
2. スキップリストと二分探索木とでは、使うポインタの数はだいたい同じである（ノードあたり 2 つ）。しかし、スキップリストのほうが使い方がうまい。その理由を説明せよ。

第 5

ハッシュテーブル

ハッシュテーブルは、広範囲（例えば $U = \{0, \ldots, 2^w - 1\}$）におよぶ整数のうち、少数（例えば n 個）の要素を格納する能率の良い方法だ。**ハッシュテーブル（hash table）**という言葉が指すデータ構造はたくさんある。本章の前半ではハッシュテーブルの一般的な実装を 2 つ紹介する。チェイン法を使う実装と、線形探索を使う実装だ。

ハッシュテーブルに整数ではないデータを格納することも多い。その場合には、各データに対応する**ハッシュ値（hash code）**という整数を使って、データをハッシュテーブルに格納する。この章の後半ではハッシュ値の生成方法について説明する。

本章で扱う手法のうちいくつかでは、ある範囲の整数からランダムにどれかを選択する方法が必要になる。このランダムな整数は、本章のサンプルコード中ではハードコードされた定数になっている。この定数を生成した際には、空気中のノイズを利用したランダムなビット列を利用した [*1]。

5.1 ChainedHashTable: チェイン法を使ったハッシュテーブル

ChainedHashTable というデータ構造では、**チェイン法（chaining）**を使って、データをリストの配列 t に保存する。リスト全体に格納されている要素数の合計を整数 n とする（図 5.1）。

[*1] 訳注：物理現象を利用するとランダムな整数が得られる、という事実だけ把握すれば、この本の理解には差し支えない。

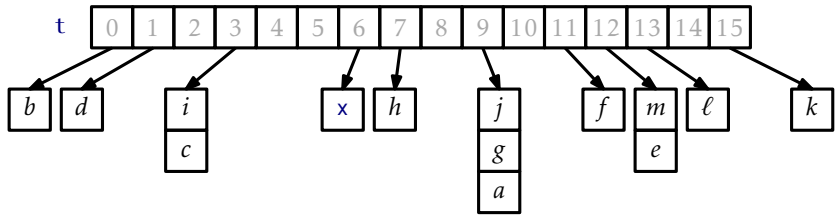


図 5.1: n = 14、t.length = 16 である ChainedHashTable の例。この例では hash(x) = 6 である

ChainedHashTable

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

データ x の**ハッシュ値**を hash(x) と書く。hash(x) は {0,...,t.length − 1} の範囲にある値である。ハッシュ値が i であるようなデータは、すべてリスト t[i] に格納する。リストが長くなりすぎないように、次の不変条件を保持する。

$$n \le t.length$$

こうすると、リストの平均要素数は常に 1 以下になる（$n/t.length \le 1$）。

ハッシュテーブルに要素 x を追加するには、配列 t の大きさを増やす必要があるかどうかを確認し、その必要があれば t を拡張する。あとは、x から {0,...,t.length − 1} 内の整数であるハッシュ値 i を計算し、x をリスト t[i] に追加すればよい。

ChainedHashTable

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

要素の追加時に配列の拡張が必要な場合は、t の大きさを 2 倍にし、元のテーブルに入っていた要素をすべて新しいテーブルに入れ直す。これは ArrayStack のときと同じ戦略であり、同じ結果を適用できる。すなわち、要素を追加する操作の列で均せば、配列の拡張にかかる償却実行時間は定数である（32 ページの補題 2.1 を参照）。

配列の拡張にかかる時間以外に、x を ChainedHashTable に追加する操作にかかる時間は、リスト t[hash(x)] へ x を追記するのにかかる時間である。2 章や 3 章で説明したどのリストの実装を使っても、この操作は定数時間で可能である。

要素 x をハッシュテーブルから削除するには、x が見つかるまでリスト t[hash(x)] を辿ればよい。

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

削除にかかる実行時間は、n_i をリスト t[i] の長さとするとき、$O(n_{\mathtt{hash}(x)})$ である。

ハッシュテーブルから要素 x を見つける操作も同様である。次のようにリスト t[hash(x)] を線形に探索すればよい。

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

```
      return t[j].get(i);
  return null;
}
```

探索の操作にも、リスト t[hash(x)] の長さに比例する時間がかかる。

ハッシュテーブルの性能はハッシュ関数の選択に大きく左右される。良いハッシュ関数は、要素を t.length 個のリストに均等に分散する関数であり、このときの各リストの長さの期待値は $O(n/t.length) = O(1)$ となる。一方、最悪のハッシュ関数は、すべての要素を同じリストに追加してしまう。すなわち、リスト t[hash(x)] の長さを n にしてしまう。次項では良いハッシュ関数の作り方を検討する。

5.1.1 乗算ハッシュ法

乗算ハッシュ法は、剰余算術（2.3 節参照）と整数の割り算からハッシュ値を効率的に計算する方法である。割り算の商となる整数を計算し、余りを捨てるには、div 演算子を使う。この演算子は、形式的には、任意の整数 $a \geq 0$ と $b \geq 1$ について $a \operatorname{div} b = \lfloor a/b \rfloor$ と定義される。

乗算ハッシュ法では、ある整数 d について、大きさ 2^d であるハッシュテーブルを使う。整数 d は**次数（dimension）**と呼ばれる。整数 $x \in \{0,\ldots,2^w-1\}$ のハッシュ値は次のように計算する。

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

ここで、z は奇数の集合 $\{1,3,5,\ldots,2^w-1\}$ からランダムに選択する。整数のビット数を w とするとき、整数の演算の結果は 2^w を法として合同になる [*2] ことを思い出すと、このハッシュ関数の効率がとても良いことがわかる（図 5.2 参照）。しかも、2^{w-d} による整数の割り算は、二進法で右側の $w-d$ ビットを落とせば計算できる（ビットを右に $w-d$ 個だけシフトすればよいので、実装は上の式よりも単純になる）。

ChainedHashTable

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

[*2] C、C#、C++、Java といった多くのプログラミング言語だと整数の演算結果はこうなるのだが、残念なことに Ruby や Python だと整数の演算結果が w ビットの固定桁のビットに収まらなくなると可変桁数の整数表現が使われてしまう。

2^w (4294967296)	10000000000000000000000000000000
z (4102541685)	11110100100001111101000101110101
x (42)	00000000000000000000000000101010
z · x	10100000011110010010000101110100110010
(z · x) mod 2^w	00011110010010000101110100110010
((z · x) mod 2^w) div 2^{w-d}	00011110

図 5.2: w = 32、d = 8 とした乗算ハッシュ法の操作

```
}
```

次の補題により、乗算ハッシュ法ではハッシュ値の衝突がうまく回避されることがわかる（証明は後述）。

補題 **5.1.** x と y を、$\{0,\ldots,2^w-1\}$ 内の任意の整数であって、$x \neq y$ を満たすものとする。このとき、$\Pr\{\mathtt{hash}(x) = \mathtt{hash}(y)\} \leq 2/2^d$ が成り立つ。

補題 5.1 より、remove(x) と find(x) の性能は簡単に解析できる。

補題 **5.2.** 任意のデータ x について、x がハッシュテーブルに現れる回数を n_x とする。このとき、リスト t[hash(x)] の長さの期待値は $n_x + 2$ 以下である。

証明. ハッシュテーブルに含まれる x ではない要素の集合を S とする。要素 $y \in S$ について、次のインジケータ確率変数を定義する。

$$I_y = \begin{cases} 1 & \text{if } \mathtt{hash}(x) = \mathtt{hash}(y) \\ 0 & \text{otherwise} \end{cases}$$

ここで、補題 5.1 より、$\mathrm{E}[I_y] \leq 2/2^d = 2/\mathtt{t.length}$ である。リスト t[hash(x)] の長さの期待値は次のように求まる。

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

$$
\begin{aligned}
&\leq n_x + \sum_{y \in S} 2/n \\
&= n_x + (n - n_x)2/n \\
&\leq n_x + 2
\end{aligned}
$$

□

続いて補題 5.1 を証明する。だがその前に、整数論の定理から導かれる結果が必要だ。次の証明では、$(b_r, \ldots, b_0)_2$ と書いて、$\sum_{i=0}^{r} b_i 2^i$ を表す。ここで、b_i は 0 か 1 である。すなわち、$(b_r, \ldots, b_0)_2$ は、二進表記が $b_r, \ldots, b_0$ となる整数である。値が不明な桁については $\star$ で表すことにする。

補題 **5.3.** $\{1, \ldots, 2^w - 1\}$ 内の奇数の集合を S とする。S の任意の要素を 2 つ選び、それぞれ q および i とする。このとき、$zq \bmod 2^w = i$ を満たす $z \in S$ の要素が一意に存在する。

証明. i は z を選ぶと決まるので、$zq \bmod 2^w = i$ を満たす $z \in S$ が一意に決まることを示せばよい。

背理法で示す。整数 z と z' が存在し $z > z'$ であると仮定する。このとき、以下がいえる。

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

よって、以下のようになる。

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

しかしこれは、ある整数 k について次の式が成り立つことを意味する。

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

これを 2 進数として考えると以下のようになる。

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

したがって、$(z - z')q$ の末尾 w 桁はすべて 0 である。

加えて、$q \neq 0$ かつ $z - z' \neq 0$ より $k \neq 0$ である。q は奇数なので、この二進表記の末尾の桁は 0 ではない。

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

$|z - z'| < 2^w$ より、$z - z'$ の末尾に連続して並ぶ 0 の個数は w 未満である。

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

よって、積 $(z - z')q$ の末尾に連続して並ぶ 0 の個数は w 未満である。

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

以上より、$(z - z')q$ は (5.1) を満たさず、矛盾する。したがって、背理法により補題 5.3 が満たされる。 □

補題 5.3 から、次の便利な事実がわかる。すなわち、z が S から一様な確率でランダムに選ばれるとき、zt は S 上に一様分布する。この便利な事実を使うと、z の二進表記が $w-1$ 桁のランダムなビットに 1 が続くものと考えられる。次の証明ではこれを使う。

補題 *5.1.*「$\mathsf{hash}(x) = \mathsf{hash}(y)$」という条件は、「$zx \bmod 2^w$ の上位 d ビットと $zy \bmod 2^w$ の上位 d ビットが等しい」と同値である。この条件の必要条件は、$z(x-y) \bmod 2^w$ の上位 d ビットがすべて 0 であるか、もしくはすべて 1 であることである。これは、$zx \bmod 2^w > zy \bmod 2^w$ ならば次のように表せる。

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

もしくは、$zx \bmod 2^w < zy \bmod 2^w$ ならば次のように表せる。

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

よって、$z(x-y) \bmod 2^w$ が (5.2) か (5.3) のどちらかであることを示せばよい。

q を、ある整数 $r \geq 0$ について $(x - y) \bmod 2^w = q2^r$ を満たす一意な奇数とする。補題 5.3 より、$zq \bmod 2^w$ の二進表記は、$w-1$ 桁のランダムなビットを持つ（最下位の桁は 1）。

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

よって、$z(x-y) \bmod 2^w = zq2^r \bmod 2^w$ は $w-r-1$ 桁のランダムなビットを持つ（その後に 1 が続き、さらに r 個の 0 が続く）。

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

これで次のようにして証明を完成できる。$r > w-d$ ならば、$z(x-y) \bmod 2^w$ の上位 d ビットは 0 と 1 を共に含む。よって、$z(x-y) \bmod 2^w$ が (5.2) または (5.3) である確率は 0 である。$r = w-d$ ならば、(5.2) の確率は 0 だが、(5.3) である確率は $1/2^{d-1} = 2/2^d$ である（$b_1,\ldots,b_{d-1} = 1,\ldots,1$ が必要だから）。$r < w-d$ ならば、$b_{w-r-1},\ldots,b_{w-r-d} = 0,\ldots,0$ か、もしくは $b_{w-r-1},\ldots,b_{w-r-d} = 1,\ldots,1$ である。いずれの場合の確率も $1/2^d$ であり、また、それぞれの事象は互いに排反である。よって、このどちらかである確率は $2/2^d$ である。 □

5.1.2 要約

次の定理は ChainedHashTable の性能をまとめたものだ。

定理 5.1. *ChainedHashTable* は、*USet* インターフェースを実装する。grow() のコストを無視すると、*ChainedHashTable* における add(x)、remove(x)、find(x) の期待実行時間は $O(1)$ である [*3]。

さらに、空の *ChainedHashTable* に対して m 個の add(x)、remove(x) からなる任意の操作列を順に実行するとき、grow() の呼び出しに要する償却実行時間は $O(m)$ である。

5.2 LinearHashTable：線形探索法

前節のデータ構造 ChainedHashTable ではリストの配列を使い、i 番めのリストに hash(x) = i となる x をすべて格納する。これに対し、**オープンアドレス法（open addressing）**と呼ばれる、配列 t に直接要素を収める方法がある [*4]。本節では、この方法を採用した LinearHashTable というデータ構

[*3] 訳注：この実行時間は非常に優れている。これまで紹介したデータ構造には、追加、削除、検索のすべてを定数時間で行えるものはなかった。

[*4] 訳注：例えば、Python の最も一般的な実装である CPython 3.7 では、オープンアドレス法で dictionary が実装されている。

造について説明する。このデータ構造は、文献によっては**線形探索法（linear probing）によるオープンアドレス**と呼ばれることもある。

LinearHashTable の背景となるのは、i = hash(x) となる要素 x をテーブルに格納するときに理想的な場所は t[i] であろうという考え方だ。すでにそこに他の要素が格納されている場合は、t[(i + 1) mod t.length] を試す。それも無理なら、t[(i + 2) mod t.length] を試す。x を格納できる場所が見つかるまで、これを繰り返す。

t の値としては次の 3 種類を使う。

1. データの値：USet に実際に入っている値
2. null：データが入っていないことを示す値
3. del：データが入っていたがそれが削除されたことを示す値

カウンタとしては、LinearHashTable の要素数 n と、データの個数と del の個数の合計 q を使う。つまり、q の値は、t の中の del の個数を n に加えた値である。効率を考えると、t には null になっている場所がたくさん欲しいので、t の大きさを q に比べて十分に大きくする必要がある。そこで、LinearHashTable の操作では、不変条件 t.length ≥ 2q を常に満たすようにする。

整理すると、LinearHashTable では、要素の配列 t と整数 n および q を利用する。n はデータ値の個数、q は null でない値の個数を保持する。さらに、ハッシュ関数の多くには値域となるテーブルの大きさが 2 の冪という制限があるので、不変条件 t.length = 2^d を満たす整数 d も保持する。

LinearHashTable

```
array<T> t;
int n;   // 値の個数
int q;   // null でない値の個数
int d;   // t.length = 2^d
```

LinearHashTable の find(x) 操作は単純だ。i = hash(x) として、t[i], t[(i + 1) mod t.length], t[(i + 2) mod t.length],... を順番に探し、t[i′] = x または t[i′] = null を満たす添字 i′ を探す。t[i′] = x のときは、x が見つかったとして t[i′] を返す。t[i′] = null のときは、x はハッシュテーブルに含まれないとして null を返す。

LinearHashTable

```
T find(T x) {
```

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

LinearHashTable の add(x) 操作も簡単に実装できる。find(x) を使って x がハッシュテーブルに含まれているかどうかを確認できる。x が含まれていなければ、$t[i], t[(i+1) \bmod t.length], t[(i+2) \bmod t.length], \ldots$ を順番に探し、null か del を見つけたら x に書き換え、n と q を 1 ずつ増やす。

LinearHashTable

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

ここまでくれば、remove(x) の実装も明らかだろう。$t[i], t[(i+1) \bmod t.length], t[(i+2) \bmod t.length], \ldots$ を順番に探し、$t[i'] = x$ または $t[i'] =$ null である添字 i' を見つける。$t[i'] = x$ ならば $t[i'] =$ del とし、true を返す。$t[i'] =$ null ならば、x はハッシュテーブルに含まれていない（そのため削除できない）ので、false を返す。

LinearHashTable

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

```
  while (t[i] != null) {
    T y = t[i];
    if (y != del && x == y) {
      t[i] = del;
      n--;
      if (8*n < t.length) resize(); // 利用率は 12.5% 以上
      return y;
    }
    i = (i == t.length-1) ? 0 : i + 1;  // i を増やす
  }
  return null;
}
```

del を使っているおかげで、find(x)、add(x)、remove(x) の正しさは簡単に検証できる。いずれの操作でも、null でない値が null に書き換えられることはない。そのため、$t[i'] = \texttt{null}$ となる添字 i' に辿り着けば、探している x はハッシュテーブルに含まれていないと証明できる。$t[i']$ はずっと null だったのだから、i' より先の添字の場所に add(x) で要素が追加されていることはないからである。

resize() が呼び出されるのは、add(x) に際して null でないエントリの数が t.length/2 を上回るとき、もしくは、remove(x) に際してデータの入っているエントリ数が t.length/8 を下回るときである。resize() は、配列を使った他のデータ構造の場合と同様に機能する。すなわち、まず $2^d \geq 3n$ を満たす最小の非負整数 d を見つける。続いて、大きさ 2^d の配列 t を割り当て、古い配列の要素をすべて移し替える。この処理の過程で、q を n に等しくリセットする。その理由は、新しい配列 t には del が含まれないからである。

LinearHashTable

```
void resize() {
  d = 1;
  while ((1<<d) < 3*n) d++;
  array<T> tnew(1<<d, null);
  q = n;
  // tnew にすべてを移す
```

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

5.2.1 線形探索法の解析

add(x)、remove(x)、find(x) は、いずれも null であるエントリが t に見つかると（あるいはその前に）終了する。直観的に線形探索法を解析すると、t のエントリの半分以上は null なので、すぐに null のエントリが見つかって操作の完了まで長い時間はかからないように見える。しかし、この直観を当てにしてはいけない。この直観に従うと、ある操作の完了までに探索する t のエントリは平均すると高々 2 個という結論になるが、これは正しくない。

この節では、ハッシュ値が $\{0,\ldots,\texttt{t.length}-1\}$ の範囲の一様な確率分布に従う独立な値であると仮定する。これは現実的な仮定ではないが、これを仮定すれば線形探索法の解析が可能になる。この節の後半では、Tabulation Hashing という、線形探索法で使うぶんには十分優秀なハッシュ法を説明する。さらに、t の添字はすべて t.length で割った剰余と等しいことも仮定する。つまり、単に t[i] と書いたら、t[i mod t.length] のことである。

ハッシュテーブルのエントリ $\texttt{t}[\texttt{i}],\texttt{t}[\texttt{i}+1],\ldots,\texttt{t}[\texttt{i}+k-1]$ がいずれも null でなく、$\texttt{t}[\texttt{i}-1]=\texttt{t}[\texttt{i}+k]=\texttt{null}$ であるとき、**i から始まる長さ k の連続 (run)** という。t の null でない要素の数はちょうど q であり、add(x) では常に $\texttt{q} \le \texttt{t.length}/2$ となることが保証されている。最後に resize() されてから t に挿入された q 個の要素を $\texttt{x}_1,\ldots,\texttt{x}_\texttt{q}$ とする。仮定より、ハッシュ値 $\texttt{hash}(\texttt{x}_j)$ はいずれも一様分布に従う互いに独立な確率変数である。以上の準備で、線形探索法の解析に必要な次の補題を示せる。

補題 **5.4.** $\{0,\ldots,\texttt{t.length}-1\}$ から i を *1* つ取って固定する。このとき、i か

ら始まる長さ k の連続が発生する確率は、定数 c（$0 < c < 1$）を使って $O(c^k)$ と表せる。

証明. i から始まる長さ k の連続があるということは、$\texttt{hash}(\texttt{x}_j) \in \{\texttt{i}, \ldots, \texttt{i}+k-1\}$ を満たすような相異なる k 個の要素 $\texttt{x}_j$ が存在する。この事象の発生確率は次のように計算できる。

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

なぜなら、そのような k 個の要素からなる連続のそれぞれについて、k 個の要素がハッシュテーブルの k 箇所に、残りの $\texttt{q}-k$ 個の要素がハッシュテーブルの残りの $\texttt{t.length}-k$ 箇所に格納されるはずだからだ [*5]。

次の導出では少しズルをして、$r!$ を $(r/e)^r$ で置き換える。この置き換えを許すと導出が簡単になるのである。$r!$ と $(r/e)^r$ の差は、スターリングの近似（1.3.2 節）より、$O(\sqrt{r})$ 程度である。問 5.4 では、スターリングの近似を使ってより厳密な計算を読者にやってもらう予定だ。

p_k は、t.length が最小値を取るときに最大値を取る。また、このデータ構造は不変条件 $\texttt{t.length} \geq 2\texttt{q}$ を保つ。よって、次の式が成り立つ。

$$\begin{aligned}
p_k &\leq \binom{\texttt{q}}{k}\left(\frac{k}{2\texttt{q}}\right)^k\left(\frac{2\texttt{q}-k}{2\texttt{q}}\right)^{\texttt{q}-k}\\
&= \left(\frac{\texttt{q}!}{(\texttt{q}-k)!k!}\right)\left(\frac{k}{2\texttt{q}}\right)^k\left(\frac{2\texttt{q}-k}{2\texttt{q}}\right)^{\texttt{q}-k}\\
&\approx \left(\frac{\texttt{q}^{\texttt{q}}}{(\texttt{q}-k)^{\texttt{q}-k}k^k}\right)\left(\frac{k}{2\texttt{q}}\right)^k\left(\frac{2\texttt{q}-k}{2\texttt{q}}\right)^{\texttt{q}-k} \qquad \text{[スターリングの近似]}\\
&= \left(\frac{\texttt{q}^k\texttt{q}^{\texttt{q}-k}}{(\texttt{q}-k)^{\texttt{q}-k}k^k}\right)\left(\frac{k}{2\texttt{q}}\right)^k\left(\frac{2\texttt{q}-k}{2\texttt{q}}\right)^{\texttt{q}-k}\\
&= \left(\frac{\texttt{q}k}{2\texttt{q}k}\right)^k\left(\frac{\texttt{q}(2\texttt{q}-k)}{2\texttt{q}(\texttt{q}-k)}\right)^{\texttt{q}-k}\\
&= \left(\frac{1}{2}\right)^k\left(\frac{(2\texttt{q}-k)}{2(\texttt{q}-k)}\right)^{\texttt{q}-k}\\
&= \left(\frac{1}{2}\right)^k\left(1+\frac{k}{2(\texttt{q}-k)}\right)^{\texttt{q}-k}
\end{aligned}$$

[*5] p_k は、その定義に $\texttt{t}[\texttt{i}-1] = \texttt{t}[\texttt{i}+k] = \texttt{null}$ という必要条件が含まれていないので、i から始まる長さ k の連続が発生する確率よりも大きいことに注意。

$$\leq \left(\frac{\sqrt{e}}{2}\right)^k$$

最後の変形では、$x > 0$ ならば $(1+1/x)^x \leq e$ であることを利用した。ここで、$\sqrt{e}/2 < 0.824360636 < 1$ なので、補題が示された。 □

補題 5.4 を使うことで、find(x)、add(x)、remove(x) の期待実行時間の上界を直接的に計算できる。まずは、find(x) を呼ぶが x が LinearHashTable に入っていないという、最もシンプルな場合を考える。この場合、i = hash(x) は $\{0,\ldots,$ t.length $-1\}$ の値を取り、t の中身とは独立な確率変数である。i が長さ k の連続の一部なら、find(x) の実行時間は $O(1+k)$ 以下である。よって、実行時間の期待値の上界を次のように計算できる。

$$O\left(1+\left(\frac{1}{\texttt{t.length}}\right)\sum_{i=1}^{\texttt{t.length}}\sum_{k=0}^{\infty} k \Pr\{\texttt{i} \text{ が長さ } k \text{ の連続の一部}\}\right)$$

内側の和を取っている長さ k の連続は k 回カウントされているので、これをまとめて k^2 とすれば、上の和は次のように変形できる。

$$\begin{aligned}
&O\left(1+\left(\frac{1}{\texttt{t.length}}\right)\sum_{i=1}^{\texttt{t.length}}\sum_{k=0}^{\infty} k^2 \Pr\{\texttt{i} \text{ から長さ } k \text{ の連続が始まる}\}\right)\\
&\leq O\left(1+\left(\frac{1}{\texttt{t.length}}\right)\sum_{i=1}^{\texttt{t.length}}\sum_{k=0}^{\infty} k^2 p_k\right)\\
&= O\left(1+\sum_{k=0}^{\infty} k^2 p_k\right)\\
&= O\left(1+\sum_{k=0}^{\infty} k^2 \cdot O(c^k)\right)\\
&= O(1)
\end{aligned}$$

最後の変形 $\sum_{k=0}^{\infty} k^2 \cdot O(c^k)$ では、指数級数の性質を使っている [*6]。以上より、LinearHashTable に入っていない x について、find(x) の期待実行時間は $O(1)$ である。

[*6] 解析学の教科書では、この和は比を計算して求める。すなわち、ある正の数 k_0 が存在し、任意の $k \geq k_0$ について、$\frac{(k+1)^2 c^{k+1}}{k^2 c^k} < 1$ を満たす。

resize() のコストを無視していいなら、これで LinearHashTable における操作はすべて解析できたことになる。

まず、上で示した find(x) の解析は、add(x) において x がハッシュテーブルに含まれないときにもそのまま適用できる。x がハッシュテーブルに含まれるときの find(x) の解析は、add(x) によって x を追加するときのコストと同じである。そして、remove(x) のコストも find(x) のコストと同じになる。

まとめると、resize() のコストを無視すれば、LinearHashTable における操作の期待実行時間はいずれも $O(1)$ である。resize() のコストを考える場合は、2.1 節で行った ArrayStack の償却解析と同様に考える。

5.2.2 要約

次の定理は LinearHashTable の性能をまとめたものだ。

定理 **5.2.** *LinearHashTable* は、*USet* インターフェースを実装する。resize() のコストを無視すると、*LinearHashTable* における add(x)、remove(x)、find(x) の期待実行時間は $O(1)$ である。

さらに、空の *LinearHashTable* に対して add(x)、remove(x) からなる m 個の操作の列を順に実行するとき、resize() にかかる時間の合計は $O(m)$ である。

5.2.3 Tabulation Hashing

LinearHashTable の解析では強い仮定を置いていた。すなわち、任意の相異なる要素 $\{x_1,\ldots,x_n\}$ について、そのハッシュ値 $\texttt{hash}(x_1),\ldots,\texttt{hash}(x_n)$ が一様な確率で $\{0,\ldots,\texttt{t.length}-1\}$ 内に独立に分布するという仮定である。この仮定を実現する方法として、大きさ 2^w の巨大な配列 tab を準備し、すべてのエントリを互いに独立な w ビットのランダムな整数で初期化するというものがある。こうしておけば、tab[x.hashCode()] から d ビットの整数を取り出すことで hash(x) を実装できる。

LinearHashTable

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

あいにく、大きさ 2^w の配列を 1 つ保持するのはメモリ利用の観点からは禁じ手である。そこで Tabulation Hashing では、w ビットの整数を、それぞれが r ビットの w/r 個の整数から構成されたものとして扱う。 この方法であれば、それぞれ大きさ 2^r の配列が w/r 個あればよい。これらの配列の全エントリを、互いに独立な w ビットのランダムな整数とする。hash(x) を計算するには、x.hashCode() を w/r 個の r ビット整数に分割し、それぞれを配列の添字として使う。その後、各配列の値からビット単位の排他的論理和を計算し、その結果を hash(x) とする。次のコードは $w = 32$、$r = 8$ の場合のものである。

LinearHashTable

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

この場合、tab は 4 つの列と $2^8 = 256$ の行からなる二次元配列となる。

任意の x について hash(x) が $\{0,\ldots,2^d-1\}$ の値を一様な確率で取ることは簡単に検証できる。少し計算すれば、ハッシュ値のペアが互いに独立であることも検証できる。これは、ChainedHashTable における乗算ハッシュ法の代わりに Tabulation Hashing を使えることを意味する。

残念ながら、相異なる任意の n 個の値の組について、そのハッシュ値が互いに独立というわけではない。だとしても、Tabulation Hashing は定理 5.2 で示した性質を保証するぶんには十分なハッシュ法である。この話題については、本章の終わりで紹介する参考文献を参照してほしい。

5.3 ハッシュ値

前節のハッシュテーブルでは、データに対して w ビットの整数のキーを対応させていた。しかし、キーは整数でないことも多い。例えば、文字列、オブジェクト、配列、あるいは他の複合データ型の場合もあるだろう。そのよう

なデータにもハッシュテーブルを使うには、これらのデータ型を w ビットのハッシュ値に対応させなければならない。このハッシュ値への対応は、以下の性質を満たす必要がある。

1. x と y が等しいとき、x.hashCode() と y.hashCode() は等しい
2. x と y が等しくないとき、x.hashCode() = y.hashCode() である確率は小さい（すなわち $1/2^w$ に近い）

1 つめの性質により、x をハッシュテーブルに入れたあとで x と等しい y を検索したときに、当然 x が見つかることが保証される。2 つめの性質は、整数への変換によるオブジェクトの損失を最小限にするためのものだ。2 つめの性質により、相異なる 2 つの要素が、通常はハッシュテーブルの違う場所に入ることが保証される。

5.3.1 プリミティブな型のハッシュ値

char、byte、int、float などの小さいプリミティブな型については、簡単にハッシュ値を計算できる。これらの型には常に 2 進数による表現があり、その表現は通常は w ビットよりも短い（例えば、C++ では char はふつうは 8 ビット型であり、float は 32 ビット型である）。このビット列を、単純に $\{0,\ldots,2^w-1\}$ の範囲の整数として解釈すればよい。そうすれば、2 つの異なる値は異なるハッシュ値を持つ。また、2 つの同じ値は同じハッシュ値を持つ。

w ビットより長いプリミティブ型もいくつかある。その長さは、通常は整数 c を使って cw ビットと表せる（Java の long 型と double 型は $c=2$ の例である）。このようなデータ型は、次項で扱うように、c 個のオブジェクトを組み合わせたものとして考えればよい。

5.3.2 複合オブジェクトのハッシュ値

複合オブジェクトのハッシュ値は、構成要素のハッシュ値を組み合わせて計算したい。これは思うほど簡単でない。ビット単位の排他的論理和を計算するといった小手先の手法はいくらでもあるが、そのうちの多くはうまくいかない（問 5.7 から問 5.9 を参照）。ただし、2w ビットの算術精度でよければ、単純かつ確実な方法がある。複合オブジェクトの構成要素を $P_0,\ldots,P_{r-1}$ としよう。$P_0,\ldots,P_{r-1}$ のそれぞれのハッシュ値は $x_0,\ldots,x_{r-1}$ であるとする。このとき、互いに独立な w ビットの乱数 $z_0,\ldots,z_{r-1}$ と、2w ビットのランダムな奇

数 z から、複合オブジェクトのハッシュ値を次のように計算できる。

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

このハッシュ値の計算過程では、最後に z を掛けて 2^w で割っていることに注目してほしい。これは 2w ビットの中間結果に 5.1.1 節で紹介した乗算ハッシュ法を使って w ビットの最終結果を得ている。3 つの構成要素 x0、x1、x2 からなる複合オブジェクトの場合について例を示す。

Point3D

```
unsigned hashCode() {
  // random.org から取得したランダムな値
  long long z[] = {0x2058cc50L, 0xcb19137eL, 0x2cb6b6fdL};
  long zz = 0xbea0107e5067d19dL;
  long h0 = ods::hashCode(x0);
  long h1 = ods::hashCode(x1);
  long h2 = ods::hashCode(x2);
  return (int)(((z[0]*h0 + z[1]*h1 + z[2]*h2)*zz) >> 32);
}
```

次の定理は、この方法が実装の単純さだけでなく良い性質を持つことを示す。

定理 **5.3.** $x_0,\ldots,x_{r-1}$ と $y_0,\ldots,y_{r-1}$ を、いずれも $\{0,\ldots,2^w-1\}$ の範囲にある w ビットの整数からなる列とする。さらに、少なくとも *1* 箇所の添字 $i \in \{0,\ldots,r-1\}$ について、$x_i \neq y_i$ が成り立つと仮定する。このとき、次が成り立つ。

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

証明. 最後の乗算ハッシュ法については後半に考える。次の関数を定義する。

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

$h'(x_0,\ldots,x_{r-1}) = h'(y_0,\ldots,y_{r-1})$ であるとする。これは次のように書き直せる。

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

ここで t は次のものである。

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

$x_i > y_i$ と仮定しても一般性を失わない。すると (5.4) は次のようになる。

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

上記のようになるのは、z_i と $(x_i - y_i)$ はいずれも $2^w - 1$ 以下なので、これらの積は $2^{2w} - 2^{w+1} + 1 < 2^{2w} - 1$ 以下だからである。仮定より $x_i - y_i \neq 0$ なので、(5.5) は z_i について高々 1 つの解を持つ。z_i と t は互いに独立（$z_0, \ldots, z_{r-1}$ は互いに独立）なので、$h'(x_0, \ldots, x_{r-1}) = h'(y_0, \ldots, y_{r-1})$ を満たす z_i を選ぶ確率は $1/2^w$ 以下である。

最後に、乗算ハッシュ法を適用することで、$2w$ ビットの中間結果 $h'(x_0, \ldots, x_{r-1})$ を w ビットの最終結果 $h(x_0, \ldots, x_{r-1})$ に縮める。補題 5.1 より、$h'(x_0, \ldots, x_{r-1}) \neq h'(y_0, \ldots, y_{r-1})$ ならば $\Pr\{h(x_0, \ldots, x_{r-1}) = h(y_0, \ldots, y_{r-1})\} \leq 2/2^w$ である。以上より、次の式が成り立つ。

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

□

5.3.3 配列と文字列のハッシュ値

前項の手法は、オブジェクトが固定数の構成要素からなる場合にはうまくいく。しかし、w ビットの乱数 z_i を構成要素の数だけ使う必要があるので、可変長のオブジェクトはうまく扱えない。

擬似乱数列を使って必要な個数の z_i を生成できるかもしれないが、そうすると z_i が互いに独立でなくなってしまう。それらの擬似乱数列がハッシュ関数に悪影響をおよぼさないことは証明が難しい。特に、定理 5.3 の証明において t と z_i の独立性が使えなくなってしまう。

ここでは、より確実な方法として、ハッシュ値を得るのに素体上の多項式を利用する。ここで素体上の多項式とは、p を素数として、$x_i \in \{0, \ldots, p-1\}$ が

係数であるようなふつうの多項式のことを指す。次の定理は、素体上の多項式は通常の多項式とだいたい同じように扱えるという主張だ。本項で説明する方法で良い性質のハッシュ値が得られるのは、この定理のおかげである。

定理 **5.4.** p を素数、$f(z) = x_0 z^0 + x_1 z^1 + \cdots + x_{r-1} z^{r-1}$ を $x_i \in \{0,\ldots,p-1\}$ を係数とする非自明な多項式（つまり $x_0 = 0$ でない）とする。このとき、方程式 $f(z) \bmod p = 0$ は、$z \in \{0,\ldots,p-1\}$ の範囲に高々 $r-1$ 個の解を持つ。

定理 5.4 を使うために、それぞれが $x_i \in \{0,\ldots,p-2\}$ である整数の列 $x_0,\ldots,x_{r-1}$ のハッシュ値を、ランダムに選んだ整数 $z \in \{0,\ldots,p-1\}$ を使って次のように求める。

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

最後に追加した項 $(p-1)z^r$ に注目してほしい。この $(p-1)$ は、$x_0,\ldots,x_r$ という整数列の末尾の要素 x_r だと考えてもよい。この末尾の要素は、整数列 $\{0,\ldots,p-2\}$ の要素のいずれとも異なる。$p-1$ は、列の終わりを示すマーカーとみなせる。

z の選択におけるランダム性は大きくないが、それでも上記は良いハッシュ値になる。これは、同じ長さの 2 つの列に関する次の定理よりいえる。

定理 **5.5.** p を $p > 2^w + 1$ を満たす素数とする。$x_0,\ldots,x_{r-1}$ と $y_0,\ldots,y_{r-1}$ を $\{0,\ldots,2^w-1\}$ の要素である w ビットの整数からなる列であるとする。$i \in \{0,\ldots,r-1\}$ のうち少なくとも *1* つ $x_i \neq y_i$ が成り立つと仮定する。このとき、次の式が成り立つ。

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

証明. 等式 $h(x_0,\ldots,x_{r-1}) = h(y_0,\ldots,y_{r-1})$ は次のように変形できる。

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

$x_i \neq y_i$ なので、左辺の多項式は次数 i について非自明である。$i < r$ なので、定理 5.4 より、z に関する方程式の解は高々 $r-1$ 個である。それらの解になるように z を選択してしまう確率は、$(r-1)/p$ 以下である。 □

2 つの列の長さが異なる場合、さらには、一方の列が他方の列の一部に含まれている場合も、このハッシュ関数で問題なく扱える。これは、このハッシュ関数が次のような無限列を扱えることによる。

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

$r > r'$ として、長さがそれぞれ r および r' の 2 つの列があるとき、2 つの列は添字 $i = r$ で異なる。このとき、(5.6) は次のようになる。

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

これは、定理 5.4 より、z について高々 r 個の解を持つ。定理 5.5 と合わせると、次のより一般的な定理が示せる。

定理 5.6. p を、$p > 2^w + 1$ を満たす素数とする。$x_0, \ldots, x_{r-1}$ と $y_0, \ldots, y_{r'-1}$ を、$\{0, \ldots, 2^w - 1\}$ の範囲にある w ビットの整数から構成される相異なる列とする。このとき次の式が成り立つ。

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

次のサンプルコードを見れば、配列 x を含むオブジェクトが、このハッシュ関数によりどう扱われるかがわかるだろう。

GeomVector

```
unsigned hashCode() {
  long p = (1L<<32)-5;    // 2^32 - 5 は素数
  long z = 0x64b6055aL;   // random.org から取得した 32 ビットの乱数
  int z2 = 0x5067d19d;    // 32 ビットのランダムな奇数
  long s = 0;
  long zi = 1;
  for (int i = 0; i < x.length; i++) {
    // 31 ビットに縮める
    long long xi = (ods::hashCode(x[i]) * z2) >> 1;
    s = (s + zi * xi) % p;
    zi = (zi * z) % p;
  }
  s = (s + zi * (p-1)) % p;
  return (int)s;
}
```

このコードは、実装上の都合で、衝突確率がやや大きい。特に、x[i].hashCode() を 31 ビットに縮めるため、5.1.1 節で $d = 31$ とした乗算

ハッシュ法を使っている。これは、素数 $p = 2^{32} - 5$ を法とする足し算や掛け算を、符号なし 63 ビット整数で実行するためである。そのため、長いほうが長さ r である 2 つの相異なる列のハッシュ値が一致する確率は次の値以下になる。

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

これは、定理 5.6 で求めた $r/(2^{32} - 5)$ よりも大きい。

5.4 ディスカッションと練習問題

ハッシュテーブルとハッシュ値は広大で活発な研究分野であり、この章ではほんのさわりを説明しただけである。ハッシュ法のオンライン参考文献一覧 [10] には 2000 近いエントリが含まれている。

ハッシュテーブルにはさまざまな実装がある。5.1 節で紹介したものは、**チェイン法によるハッシュ法**（hashing with chaining）と呼ばれる（各配列のエントリに要素のチェイン（`List`）が含まれる）。チェイン法によるハッシュ法の起源は、1953 年 1 月の H. P. Luhn による IBM の内部報告書にまでさかのぼる。この報告書は、連結リストに関する最古の文献の 1 つであるとも思われる。

チェイン法によるハッシュ法の代替手段となるのが**オープンアドレス法**だ。オープンアドレス法では、データを配列に直接格納する。5.2 節で説明した `LinearHashTable` は、このオープンアドレス法の一種である。この方法もやはり IBM の別のグループによって 1950 年代に提案された。オープンアドレス法では**衝突の解決**（collision resolution）について考えなければならない。ここでいう衝突は、2 つの値が配列の同じ位置に割り当てられることを指す。衝突の解決方法にはいくつか種類があり、それぞれ性能保証も異なる。この章で示したものよりも精巧なハッシュ関数が必要になることが多い。

さらに別のハッシュテーブルの実装として、**完全ハッシュ法**（perfect hashing）と呼ばれる種類のものがある。完全ハッシュ法では、`find(x)` の実行時間が、最悪の場合でも $O(1)$ となる。データセットが静的であれば、データセットに対する**完全ハッシュ関数**（perfect hash function）を見つけることで、完全ハッシュ法を実現できる。完全ハッシュ関数とは、各データを配列内で別々の位置に対応付けるような関数である。データが動的な場合には、FKS **二段階ハッシュテーブル**（two-level hash table）[31, 24] や cuckoo hashing [55] などが完全ハッシュ法として知られている。

この章で紹介したハッシュ関数は、任意のデータセットに対してうまく動作することが証明できる既知の手法としては、おそらく最も実用的なものである。他の方法としては、Carter と Wegman による先駆け的な研究成果である**ユニバーサルハッシュ法**（universal hashing）を使ったものがあり、用途に応じてさまざまなハッシュ関数が提案されている [14]。5.2.3 節で説明した Tabulation hashing は Carter と Wegman の研究 [14] によるものだが、この手法を線形探索法（と他のいくつかのハッシュテーブルの実装）に適用した場合の解析は、Pătraşcu と Thorup の研究成果である [58]。

乗算ハッシュ法（multiplicative hashing）のアイデアは非常に古く、もはや伝承しか残されていない [48, Section 6.4]。しかし、乗数 z をランダムな奇数から選ぶという 5.1.1 節で説明したアイデアと解析は、Dietzfelbinger らの研究成果である [23]。この乗算ハッシュ法は、最もシンプルなハッシュ法の 1 つだが、衝突確率が $2/2^d$ である。これは、2^w から 2^d へのランダムな関数に期待される衝突確率と比べると 2 倍も大きい。multiply-add **ハッシュ法**は次の関数を使う方法だ。

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

ここで、z と b はいずれも $\{0,\ldots,2^{2w}-1\}$ からランダムに選出される。multiply-add ハッシュ法の衝突確率は $1/2^d$ である [21]。ただし、$2w$ ビット精度の四則演算が必要になる。

固定長の w ビットの整数列からハッシュ値を得る方法はたくさんある。特に高速な方法は次のものだ [11]。

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

ここで r は偶数であり、$a_0,\ldots,a_{r-1}$ はいずれも $\{0,\ldots,2^w\}$ からランダムに選出される。この $2w$ ビットのハッシュ値が衝突する確率は $1/2^w$ である。これは、乗算ハッシュ法（か multiply-add ハッシュ法）を使って w ビットに縮めることができる。この計算は $r/2$ 回の $2w$ ビット乗算だけで実現でき、高速である。5.3.2 節の方法だと、r 回の乗算が必要であった（mod の計算は、w または $2w$ ビットの足し算や掛け算では暗に実行される）。

5.3.3 節で説明した素体を使った可変長配列のハッシュ法は、Dietzfelbinger らによるものだ [22]。この方法では mod を使うが、これは時間のかかる機械語の命令であり、結果的に高速に計算できない。剰余の法として 2^w-1 を使う工夫をすれば、mod を加算とビット単位の and 演算に置き換えられる [47, Section 3.6]。他の方法としては、固定長の高速なハッシュ法を使って長

さ $c > 1$ のブロックに対するハッシュ値を計算し、その結果の $\lceil r/c \rceil$ 個のハッシュ値の配列に素体を使った方法でハッシュ値を求めるという手法がある。

問 **5.1.** ある大学では生徒が初めて講義を履修するときに学生番号を発行する。この番号は 1 ずつ増える整数で、何年も前に 0 から始まり、いまでは数百万になっている。百人の一年生が受講する講義にて、各生徒に学生番号から計算したハッシュ値を割り当てる。このとき、下の 2 桁、あるいは上の 2 桁のどちらを使うのが優れているだろうか。説明せよ。

問 **5.2.** 5.1.1 節の方法において、$n = 2^d$ かつ $d \leq w/2$ である場合を考える。

1. z によらず、相異なる n 個の入力であって、同じハッシュ値を持つものが存在することを示せ（ヒント：これは簡単な問題であり、数論の知識などは必要ない）。
2. z が与えられたとき、n 個の同じハッシュ値を持つ値を求めよ（これは少し難しい問題で、基本的な数論の知識が必要だ）。

問 **5.3.** 補題 5.1 で得た上界 $2/2^d$ は、ある意味で最適であることを示せ。$x = 2^{w-d-2}$ かつ $y = 3x$ のとき、$\Pr\{\mathrm{hash}(x) = \mathrm{hash}(y)\} = 2/2^d$ であることを示せ（ヒント：zx と $z3x$ の二進表記を考え、$z3x = zx + 2zx$ であることを利用せよ）。

問 **5.4.** 1.3.2 節で与えたスターリングの公式を使って、補題 5.4 を、今度は誤魔化しなしで証明せよ。

問 **5.5.** 次に示したのは、要素 x を LinearHashTable に追加するコードを簡略化したものである。このコードでは、単純に、最初に見つけた null のエントリへ x を入れる。このコードは非常に遅い場合があることを示せ。具体的には、$O(n)$ 個の add(x)、remove(x)、find(x) からなる操作の列で、実行時間が n^2 になる例を挙げよ。

LinearHashTable

```
bool addSlow(T x) {
  if (2*(q+1) > t.length) resize();   // 利用率 50% 以下
  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; // i を増やす
  }
```

```
  t[i] = x;
  n++; q++;
  return true;
}
```

問 **5.6.** 昔の Java では、String クラスの hashCode() メソッドでは、長い文字列のすべての文字を使っていなかった。例えば、16 文字の文字列の場合、偶数番めの 8 文字だけを使っていた。これがよくないアイデアであること、すなわち、同じハッシュ値を持つ文字列がたくさん現れるような例を挙げよ。

問 **5.7.** 2 つの w ビットの整数 x と y からなるオブジェクトがあるとき、x + y をハッシュ値とするのはよくないことを示せ。すなわち、ハッシュ値が 0 となるようなオブジェクトの例をたくさん挙げよ。

問 **5.8.** 2 つの w ビットの整数 x と y からなるオブジェクトがあるとき、x + y をハッシュ値とするのはよくないことを示せ。すなわち、同じハッシュ値を持つオブジェクトの集まりの例を挙げよ。

問 **5.9.** 2 つの w ビットの整数 x と y からなるオブジェクトがあるとする。決定的な関数 $h(x,y)$ により、w ビットの整数となるハッシュ値を計算するとする。このとき、ハッシュ値が一致するオブジェクトの集合であって、要素数の大きいものが存在することを示せ。

問 **5.10.** ある正の数 w について、$p = 2^w - 1$ であるとする。正の数 x について次の式が成り立つ理由を説明せよ。

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

（この式より、$x \bmod (2^w - 1)$ を計算する方法として、$x \leq 2^w - 1$ を満たすまで次のコードを繰り返すというアルゴリズムが得られる。）

```
x = x&((1 << w) - 1) + x >> w
```

問 **5.11.** 標準ライブラリや、本書の HashTable および LinearHashTable を参考に、よく使われるハッシュテーブルの実装を見つけ、整数 x に対して find(x) が線形時間で実行できるプログラムを実装せよ。つまり、テーブルの中の同じ位置に対応付けられる n 個の整数の集まりを見つけよ。

実装の出来不出来によっては、コードを見るだけで実行時間がわかる場合もあれば、挿入や検索をしてみて実行時間を測るコードを書く必要があるかも

しれない（これは Web サーバーへの DoS 攻撃に使われることがある [17]）。

第 6

二分木

この章では、コンピュータサイエンスで現れる最も基本的な構造のうちのひとつ、二分木を紹介する。この構造を**木（tree）**と呼ぶのは、図示したときに（森に生えている）木に似ているためである。二分木には複数の定義がある。数学的な**二分木（binary tree）**の定義は、連結（connected）[*1] な有限無向グラフであって、閉路（cycle）[*2] を持たず、すべての頂点の次数（degree）が 3 以下のものである [*3]。

コンピュータサイエンスにおける応用では、二分木はふつう**根を持つ（rooted）**。木の**根（root）**と呼ばれる特殊なノード r があり、このノードの次数は 2 以下である。ノード u($\neq$ r) から r に向かう経路における 2 番めのノードを u の**親（parent）**という。u に隣接する親以外のノードを u の**子（child）**という。特に**順序付けられている（ordered）**二分木に興味があることが多いので、子を**左の子**、**右の子**と呼び分けることにする。

二分木を図示するとき、ふつうは根を一番上に書く。また、ノード u の左右の子は、u の左下と右下にそれぞれ描く（図 6.1）。図 6.2 の (a) に、9 個のノードを持つ二分木の例を示す。

木（および二分木）は重要なので、その性質を記述するための専用の語彙が使われている。木におけるノード u の**深さ（depth）**とは、u から根までの

[*1] 訳注：グラフが連結であるとは、グラフ上の辺を辿ることで任意の 2 頂点間を行き来できることである。そうして辿った辺の列のことを経路（path）と呼ぶ。

[*2] （単純）閉路とは、ある頂点から同じ頂点および同じ辺を通らずにその頂点に戻る経路である。

[*3] 訳注：無向グラフの頂点における次数とは、その頂点が持つ辺の数である。例えば図 6.1 において、u の次数は 3 である。

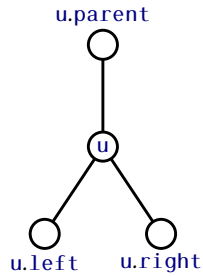


図 6.1: BinaryTree における、ノード u の親、左の子、右の子

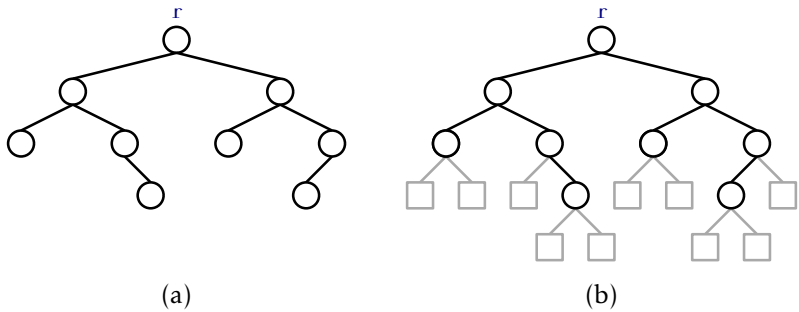


図 6.2: (a) 9 個の本物のノードを持つ二分木と、(b) 10 個の外部ノードを持つ二分木

経路の長さである [*4]。ノード w が u から r への経路に含まれるとき、w を u の**祖先**（ancestor）という。一方、このとき u を w の**子孫**（descendant）という。木におけるノード u の**部分木**（subtree）とは、u を根とし、u のすべての子孫を含む木である。ノード u の**高さ**（height）とは、u から u の子孫への経路の長さの最大値である。木の**高さ**とは、その根の高さである。ノード u が子を持たない場合、u は**葉**（leaf）という。

木を考えるとき、**外部ノード**（external node）で拡張すると便利なことがある。左の子を持たないノードであれば、左の子として外部ノードを持つ。同様に、右の子を持たないノードであれば、右の子として外部ノードを持つ（図 6.2(b) を参照）。帰納法により、$n \geq 1$ 個の（本物の）ノードを持つ二分木は、$n+1$ 個の外部ノードを持つことが示せる。

[*4] 訳注：例えば u=r であるとき、u の深さは 0 である。

6.1 BinaryTree：基本的な二分木

二分木におけるノード u を簡単に表現するには、u に隣接するノードを明示的に保持すればよい。

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

隣接する頂点は最大で 3 つあるが、そのうち存在しないものを指す変数は nil とする。すると、外部ノードも根の親も nil に対応する。

このように二分木を表現すると、二分木自体は根 r への参照として表現できる [*5]。

BinaryTree
```
Node *r;    // 根（root）ノード
```

ノード u の深さは、u から根への経路を辿るときのステップ数として計算できる。

BinaryTree
```
int depth(Node *u) {
  int d = 0;
  while (u != r) {
    u = u->parent;
    d++;
```

[*5] 訳注：本章の冒頭で、二分木を連結なグラフとして定義したことを思い出そう。連結なグラフなので、木に含まれるいずれかのノードへの参照が 1 つあれば、そこから他のノードへ辿り着ける。そのような参照として根 r を選べば、その二分木を示せるというわけである。なお、もし根以外の参照を選ぶと、後の章で出てくる B 木（14.2.2 節参照）のような木では親への参照がないので、複数のノードが必要になる。

```
  }
  return d;
}
```

6.1.1 再帰的なアルゴリズム

再帰的なアルゴリズムを使うと二分木に関する計算が簡単になる。例えば、u を根とする二分木のサイズ（ノードの数）は、u の子を根とする部分木のサイズを再帰的に計算し、足し合わせ、その結果に 1 加えると求まる。

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

ノード u の高さは、u の 2 つの部分木の高さの最大値を計算し、その結果に 1 加えると求まる。

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

6.1.2 二分木の走査

前項で説明した 2 つのアルゴリズムでは、二分木のすべてのノードを訪問するために再帰を使った。いずれのアルゴリズムも、二分木のノードを次のコードと同じ順番で訪問していた。

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

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

再帰を使うとこのように簡潔なコードを書けるが、時に困ることもある。再帰の深さの最大値は、二分木におけるノードの深さの最大値、すなわち木の高さである。これが非常に大きいと、再帰のためのスタックとして利用できる量以上の領域を要求し、プログラムがクラッシュしてしまうことがある [*6]。

再帰なしで二分木を走査するためには、どこから来たかによって次の行き先を決めるアルゴリズムを使えばよい（図 6.3）。ノード u に u.parent から来たときは、次は u.left に向かう。u.left から来たときは、次は u.right に向かう。u.right から来たときは、u の部分木を辿り終えたので u.parent に戻る。次のコードはこれを実装したものである。u.left、u.right、u.parent が nil であるケースも適切に処理している。

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;
  }
```

[*6] 訳注：この問題はスタックオーバーフローと呼ばれる。

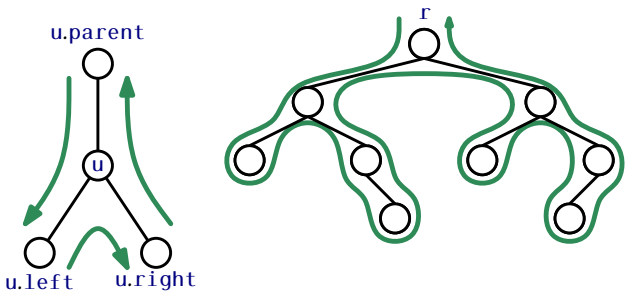


図 6.3: 再帰を使わずに二分木を走査してノード u を訪れる 3 通りの方法と、そのときの木の走査

```
}
```

再帰アルゴリズムで計算できることは、こうして再帰なしでも計算できる。例えば木のサイズを計算するためには、カウンタ n を保持し、新しいノードを訪問するたびにその値を 1 ずつ増やせばよい。

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;
  }
```

二分木の実装では parent を使わないこともある。この場合にも再帰を使わない実装は可能だが、いま訪問しているノードから根までの経路を List か Stack を使って記録しておく必要がある。

ここまでの方法とは別の走査方法として、**幅優先**（breadth-first）な走査がある [*7]。幅優先に走査する場合、根から下に向かって深さごとにすべてのノードを訪問する。同じ深さのノードは左から右の順に訪問する（図 6.4 を参照せよ）。これは英語の文章の読み方と似ている。幅優先の走査はキュー q を使って実装できる。初期状態では q は根だけを含む。各ステップでは、q から次のノード u を取り出し、u を処理し、u.left と u.right を（nil でなければ）q に追加する。

BinaryTree

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

[*7] 訳注：12.3.1 節では、木の一般化であるグラフにおける幅優先探索アルゴリズムを紹介する。

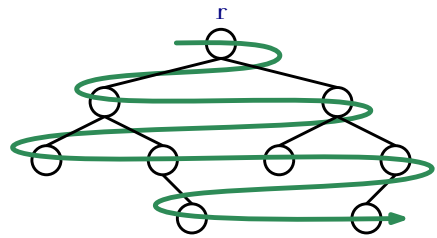


図 6.4: 幅優先な走査では、二分木の各ノードを深さごとに訪問する。各深さでは左から右の順で訪問する

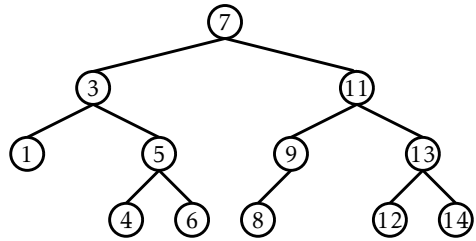


図 6.5: 二分探索木の例

6.2 BinarySearchTree：バランスされていない二分探索木

ノード u が、ある全順序な集合の要素 x をデータ u.x として持つような特別な二分木、BinarySearchTree を考える。各ノードとそのデータは、次の**二分探索木の性質**を満たすとする。すなわち、ノード u について、u.left を根とする部分木に含まれるデータはすべて u.x より小さく、u.right を根とする部分木に含まれるデータはすべて u.x より大きい。このような BinarySearchTree の例を図 6.5 に示す。

6.2.1 探索

二分探索木の性質はとても有用だ。この性質を利用して、二分探索木から値 x を高速に見つけられる。具体的には、まず根 r から x を探し始める。ノード u を訪問しているとき、次の 3 つの場合がありうる。

1. x < u.x なら u.left に進む
2. x > u.x なら u.right に進む
3. x = u.x なら値が x であるノード u を見つけた

この探索は 3 つめのケース、または u = nil になると終了する。前者なら x が見つかったことになる。後者なら x がこの木に含まれていないとわかる。

BinarySearchTree

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

二分探索木における探索の例を図 6.6 に 2 つ示す。2 つめの例から、x が見つからない場合でも、役に立つ情報が得られることがわかる。探索における最後のノード u にて、先の場合分けの 1 つめのケースであったなら、u.x は木に含まれるデータであって、x よりも大きい値のうちで最小のものである。同様に、場合分けの 2 つめのケースであったなら、u.x は x より小さい値のうちで最大のものである。よって、場合分けの 1 つめのケースが最後に発生したノード z を記録しておけば、x 以上の値のうちで最小のものを返すように

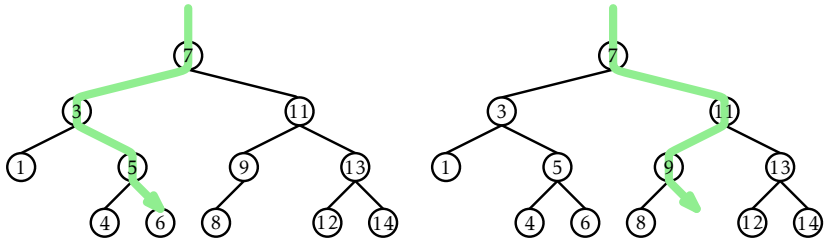


図 6.6: 二分探索木において、(a) 探索が成功する例（6 が見つかる）と、(b) 探索が失敗する例（10 が見つからない）

BinarySearchTree の find(x) を実装できる。

BinarySearchTree

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

6.2.2 追加

BinarySearchTree に値 x を追加するには、まず x を検索する。もし見つかれば挿入の必要がない。見つからなければ、検索において最後に出会ったノード p の子である葉として、x を保存する。このとき、新しいノードが p の右の子か左の子かを、x と p.x の比較結果によって決める。

BinarySearchTree

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

BinarySearchTree

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

BinarySearchTree
```
bool addChild(Node *p, Node *u) {
    if (p == nil) {
      r = u;                // 空っぽの木に挿入する
    } 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 はすでに木に含まれている
      }
      u->parent = p;
    }
    n++;
    return true;
  }
```

図 6.7 に例を示す。最も時間がかかるのは x を検索する処理で、この時間は新たに追加するノード u の深さに比例する。これは、最悪の場合、BinarySearchTree の高さである。

6.2.3 削除

ある値を格納するノード u を BinarySearchTree から削除する処理はもう少し複雑だ。u が葉なら、u を単に親から切り離せばよい。u が子を 1 つだけ持っているなら、u で両端を継ぎ合わせる、すなわち、u.parent と u の子とを親子関係にすればよい（図 6.8 を参照）。

BinarySearchTree
```
void splice(Node *u) {
  Node *s, *p;
  if (u->left != nil) {
```

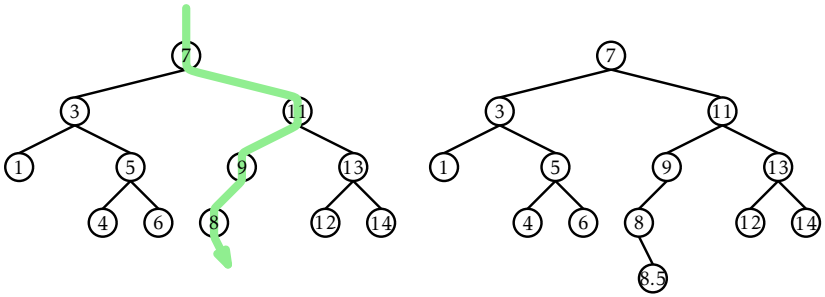


図 6.7: 二分探索木に 8.5 を追加

```
    s = u->left;
  } else {
    s = u->right;
  }
  if (u == r) {
    r = s;
    p = nil;
  } else {
    p = u->parent;
    if (p->left == u) {
      p->left = s;
    } else {
      p->right = s;
    }
  }
  if (s != nil) {
    s->parent = p;
  }
  n--;
}
```

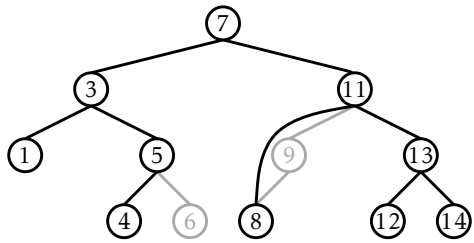


図 6.8: 葉（6）、または子を 1 つだけ持つノード（9）の削除は簡単

u が子を 2 つ持っている場合は、もっと手の込んだ操作が必要になる。この場合、子の数が 1 以下のノード w で、w.x と u.x とを入れ替えられるようなものを見つけるのが最も単純だ。二分探索木の性質を保つためには、w.x の値と u.x の値が近ければよい。例えば、w.x が、u.x より大きいものの中で最小の値であればよい。このような w は簡単に見つけられる。これは u.right を根とする部分木の中で最小の値である。このノードは左の子を持たないので、取り除くのも簡単である（図 6.9 を参照）。

BinarySearchTree

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

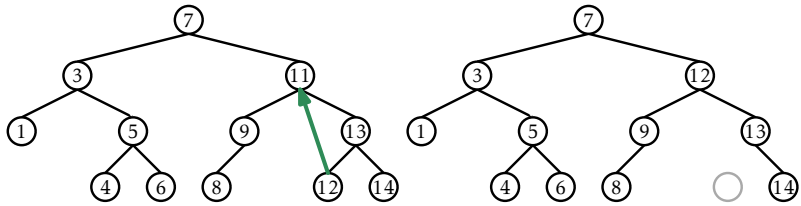


図 6.9: 2 つの子を持つノード u から値 11 を削除するために、u の値と、u の右の部分木における最小の値とを入れ替える

6.2.4 要約

BinarySearchTree における find(x)、add(x)、remove(x) の処理は、いずれも根から特定のノードへの経路を辿る処理を伴う。木の形状について何らかの仮定をしない限り、この経路の長さについて、「木の中のノード数を超えない」というより強い主張をするのは難しい。次の定理は、あまり面白いものではないが、BinarySearchTree の性能をまとめたものだ。

定理 **6.1.** *BinarySearchTree は SSet インターフェースの実装であって、add(x)、remove(x)、find(x) の実行時間は $O(n)$ である。*

定理 4.1 と比べると、定理 6.1 の性能は良くない。SkiplistSSet では、各操作の期待実行時間が $O(\log n)$ であるように SSet インターフェースを実装できた。BinarySearchTree には、木の形状が**アンバランス（unbalanced）**かもしれないという問題がある。図 6.5 のような木の形ではなく、ほとんどのノードが子を 1 つだけ持ち、n 個のノードからなる長い鎖のような見た目かもしれないのである [*8]。

二分探索木がアンバランスになることを回避する方法はたくさんある。そのような方法を採用すれば、$O(\log n)$ の時間で各操作を行えるようになる。7 章では、ランダム性を利用することで、$O(\log n)$ の**期待実行時間**を達成する方法を説明する。8 章では、部分的な再構築を利用することで、$O(\log n)$ の**償却実行時間**を達成する方法を説明する。9 章では、子を 4 つまで持ちうる木をシミュレートすることで、$O(\log n)$ の**最悪実行時間**を達成する方法を説明

[*8] 訳注：このような長い鎖のような見た目をしたデータ構造には見覚えがあるかもしれない。3 章で扱った連結リストを思い出そう（細かな相違はある）。

する。

6.3 ディスカッションと練習問題

二分木は血縁関係のモデルとして数千年にわたって使われてきた。二分木を使うことで、家系図を自然にモデル化できる。ある家系図の書き方では、ある人物を根に配し、その人物の両親を左右の子ノードとする。生物学における系統樹でも、数世紀にわたって二分木が使われてきた。生物学では、現存の種を二分木における葉で表し、**分化**の発生を内部ノードで表す。分化とは、1 つの種から 2 つの別々の種が派生することである。

二分探索木は、1950 年代に複数のグループが独立に発見したようである [48, Section 6.2.2]。個々の二分探索木に関する詳細な文献は後の各章で紹介する。

二分木をゼロから実装するときには、設計上の考慮が必要になる点がいくつかある。その 1 つは、各ノードが親へのポインタを持つかどうかである。根から葉への経路を辿るだけの操作が多いなら、親へのポインタは不要である。親へのポインタを組み込めばメモリが無駄になり、バグの原因ともなりうる。一方、親へのポインタがないと、走査のために再帰を使うことになる（もしくは明示的にスタックを利用することになる）。また、ある種の二分探索木における挿入や削除など、実装が複雑になってしまう操作もある。

もう 1 つの設計上のポイントは、親と左右の子へのポインタをどう持つかである。本章の実装では、それぞれを別々の変数に保持していた。そうではなく、長さ 3 の配列 p を使って、u.p[0]、u.p[1]、u.p[2] がそれぞれ u の左右の子と親へのポインタを保持するようにしてもよい。配列を使うことで、プログラム内の `if` 文の連続を代数的な表現でより単純に書けるようになる。

例えば、木を辿る処理をより単純に書ける。u.p[i] から u に来たとき、次に向かうのは u.p[(i + 1) mod 3] である。左右の対称性があるときにも似たようなことができる。すなわち、u.p[i] の兄弟が u.p[(i + 1) mod 2] になる。これは、u.p[i] が左の子（i = 0）であっても右の子（i = 1）であっても有効だ。この表現を使うことで、左右に分けてそれぞれ書いていた複雑なコードを 1 つにまとめられる場合がある。164 ページの `rotateLeft(u)` と `rotateRight(u)` がその好例である。

問 **6.1.** n ≥ 1 個のノードからなる二分木は n − 1 本の辺を持つことを示せ。

問 **6.2.** $n \geq 1$ 個の（本物の）ノードからなる二分木は $n+1$ 個の外部ノードを持つことを示せ。

問 **6.3.** 二分木 T が葉を 1 つ以上持つとき、T における根の子の数が 1 以下であるか、T が葉を 2 つ以上持つかのいずれかであることを示せ。

問 **6.4.** ノード u を根とする部分木の大きさを計算する再帰的でないメソッド `size2(u)` を実装せよ。

問 **6.5.** ノード u の高さを計算する再帰的でないメソッド `height2(u)` を実装せよ。

問 **6.6.** 二分木が**サイズでバランスされている**（size-balanced）とは、任意のノード u について、`u.left` を根とする部分木のサイズと、`u.right` を根とする部分木のサイズとの差が 1 以下であることをいう。二分木がこの意味でバランスされているかを判定する再帰的なメソッド `isBalanced()` を書け。なお、このメソッドの実行時間は $O(n)$ でなければならない（さまざまな形状の大きい木でテストしてみること。$O(n)$ よりも時間がかかる実装は簡単である）。

行きがけ順（pre-order）とは、二分木の訪問順であって、ノード u をそのいずれの子よりも先に訪問するものである。**通りがけ順**（in-order）とは、二分木の訪問順であって、ノード u を左の部分木に含まれる子よりも後かつ右の部分木に含まれる子よりも先に訪問するものである。**帰りがけ順**（post-order）とは、二分木の訪問順であって、ノード u を u を根とする部分木に含まれるいずれの子よりも後に訪問するものである。行きがけ番号、通りがけ番号、帰りがけ番号とは、それぞれの対応する順序に従って頂点を訪問したときにノードに付される訪問順の番号である。図 6.10 に例を示す。

問 **6.7.** `BinarySearchTree` のサブクラスとして、ノードのフィールドに行きがけ番号、通りがけ番号、帰りがけ番号を持つものを作れ。これらの値を適切に割り当てる再帰的な関数 `preOrderNumber()`、`inOrderNumber()`、`postOrderNumber()` を書け。なお、いずれの実行時間も $O(n)$ でなければならない。

問 **6.8.** 再帰的でない関数 `nextPreOrder(u)`、`nextInOrder(u)`、`nextPostOrder(u)` を実装せよ。これらは各順序におけるノード u の次のノードを返す関数である。いずれの償却実行時間も高々定数でなければならない。なお、ノード u から始めて、この関数を繰り返し呼んでノードを辿り、`u = null` になるまでこれを続けるとき、すべての呼び出しの合計コストは $O(n)$ でなければなら

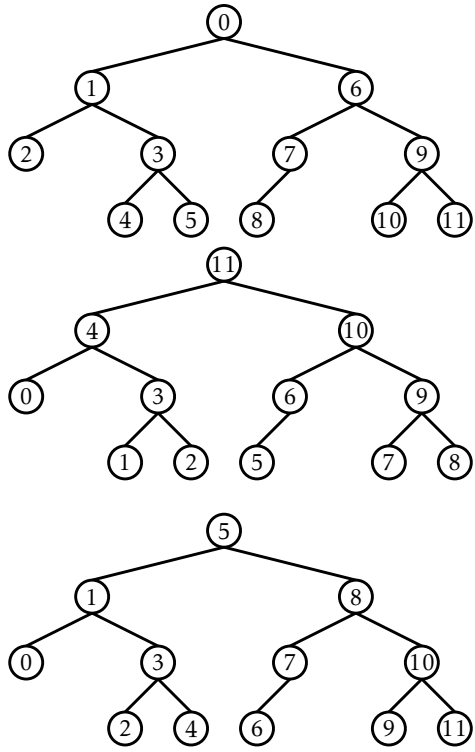


図 6.10: 二分木における行きがけ順、通りがけ順、帰りがけ順

ない。

問 **6.9.** ノードに行きがけ番号、通りがけ番号、帰りがけ番号が付された二分木があるとする。この番号を使って次の質問に定数時間で答える方法を考えよ。

1. ノード u が与えられたとき、u を根とする部分木の大きさを求めよ。
2. ノード u が与えられたとき、u の深さを求めよ。
3. ノード u と w が与えられたとき、u が w の祖先であるかを判定せよ。

問 **6.10.** ノードに対する行きがけ番号、通りがけ番号の組からなるリストが与えられたとする。このような行きがけ番号、通りがけ番号が付される木は一意に定まることを示せ。また、具体的にこの木を構成する方法を与えよ。

問 **6.11.** n 個のノードからなる二分木は 2(n−1) ビット以下で表現できることを示せ。
ヒント：木を走査する際に起きることを記録し、これを再生して木を再構築することを考えるとよい。

問 **6.12.** 図 6.5 の二分木に 3.5 を追加し、続けて 4.5 を追加するときの様子を図示せよ。

問 **6.13.** 図 6.5 の二分木に 3 を削除し、続けて 5 を削除するときの様子を図示せよ。

問 **6.14.** BinarySearchTree のメソッド getLE(x) を実装せよ。これは、木に含まれる要素のうち、x 以下のものを集めたリストを返すものである。このメソッドの実行時間は $O(n' + h)$ でなければならない。ここで、n' は木に含まれる x 以下の要素の数、h は木の高さである。

問 **6.15.** 空の BinarySearchTree に {1,...,n} をすべて追加し、結果として得られる木の高さが n−1 になるためにはどうすればよいか。また、そのやり方は何通りあるか。

問 **6.16.** ある BinarySearchTree に add(x) を実行し、同じ x について remove(x) を実行すると、木は必ず元の状態に戻るか。

問 **6.17.** BinarySearchTree において remove(x) を実行するとき、あるノードの高さが大きくなることはあるか。もしそうなら、どのくらい大きくなりうるか。

問 **6.18.** BinarySearchTree において add(x) を実行するとき、あるノードの高さが大きくなることはあるか。また、そのとき木の高さが大きくなることはあるか。もしそうなら、どれくらい大きくなりうるか。

問 **6.19.** 各ノード u が u.size（u を根とする部分木の大きさ）、u.depth（u の深さ）、u.height（u を根とする部分木の高さ）を保持するような BinarySearchTree の変種を設計、実装せよ。
なお、add(x) と remove(x) を呼んでもこれらの値は適切に保たれる必要がある。add(x) や remove(x) のコストが定数時間より大きくはならないように注意すること。

第 7

ランダム二分探索木

この章では、乱択化を使って各操作について $O(\log n)$ の実行時間を達成する二分探索木を紹介する。

7.1 ランダム二分探索木

図 7.1 に示した 2 つの二分探索木を見てほしい。これらはいずれも $n = 15$ 個のノードからなる。左の木はリストであり、右の木は完全にバランスされた二分探索木である。左の木の高さは $n - 1 = 14$ で、右の木の高さは 3 である。

この 2 つの木の構築方法を考えてみよう。空の `BinarySearchTree` に対し

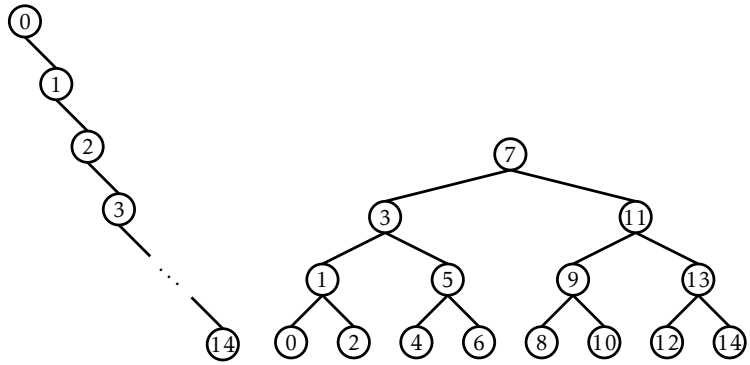


図 7.1: 0,...,14 からなる 2 つの二分探索木

て、次の順で要素を追加すると、左の木になる。

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

逆に、左の木になるような要素の追加順は、この順に限る（n についての帰納法で証明できる）。一方、次の順で要素を追加すると、右の木になる。

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

右の木は、ほかにも次のような追加順で得られる。

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

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

実際、右の木が得られるような要素の追加順は、全部で 21,964,800 種類ある。その一方で、左の木が得られるような要素の追加順はたった 1 つしかない。

この例からは次のようなことがいえるだろう。すなわち、0,...,14 をランダムな順番で選んで二分探索木に追加すれば、多くの場合は図 7.1 の右に示すようなバランスされた木になり、左に示すような極めて偏った木になることはあまりなさそうである。

この主張は、ランダム二分探索木を考えることで形式的に説明できる。まずは大きさ n の**ランダム二分探索木**（random binary search tree）の作り方を述べる。

0,...,n−1 のランダムな置換を 1 つ選ぶ。これを $x_0,\ldots,x_{n-1}$ とし、この順番に要素を `BinarySearchTree` に追加する。ここで、**ランダムな置換**（random permutation）とは、全部で n! 通りある 0,...,n − 1 の置換はそれぞれ等確率 1/n! で選び出せるので、そのうちの 1 つを選んだものである。

0,...,n − 1 は、n 個の値であれば何でもよいことに注意してほしい。別の値であってもランダム二分探索木の性質には影響しない。$x \in \{0,\ldots,n-1\}$ は、順序が定義された n 個の要素からなる集合における x 番めの要素を表しているにすぎない。

ランダム二分探索木の性質を説明する前に、少し脱線して、ランダムな構造を解析する際にしばしば登場する**調和数**（harmonic number）について説明しよう。k を非負整数とすると、k 番めの調和数 H_k は次のように定義される。

$$H_k = 1 + 1/2 + 1/3 + \cdots + 1/k$$

調和数 H_k は単純な閉じた式では書けないが、自然対数と密接な関係があり、次の式が成り立つ。

$$\ln k < H_k \leq \ln k + 1$$

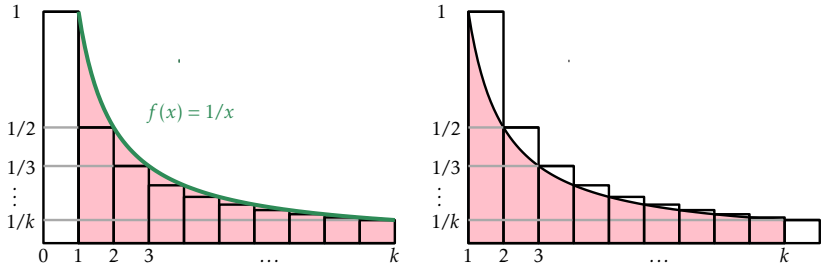


図 7.2: k 番めの調和数 $H_k = \sum_{i=1}^{k} 1/i$ の上界、下界を積分で計算できる。各積分値は図の斜線部の面積であり、H_k は長方形の部分の面積である

解析学を学んでいれば、$\int_1^k (1/x)\,\mathrm{d}x = \ln k$ からこれを示せるだろう。積分は、曲線と x 軸とが囲む領域の面積と解釈できるので、H_k の下界は $\int_1^k (1/x)\,\mathrm{d}x$、上界は $1 + \int_1^k (1/x)\,\mathrm{d}x$ である（図 7.2 を見れば視覚的に理解できるだろう）。

補題 **7.1.** 大きさ n のランダム二分探索木について以下が成り立つ。

1. 任意の $\mathsf{x} \in \{0,\ldots,\mathsf{n}-1\}$ について、x を探すときの探索経路の長さの期待値は $H_{\mathsf{x}+1} + H_{\mathsf{n}-\mathsf{x}} - 2$ である [*1]
2. 任意の $\mathsf{x} \in (-1,n) \setminus \{0,\ldots,\mathsf{n}-1\}$ について、x を探すときの探索経路の長さの期待値は $H_{\lceil \mathsf{x} \rceil} + H_{\mathsf{n}-\lceil \mathsf{x} \rceil}$ である

補題 7.1 の証明は次項で行う。ここでは補題 7.1 の意味を考える。1 つめの主張は、木に含まれる要素を探す場合、木の要素数を n とすると、探索経路の長さの期待値は $2\ln \mathsf{n} + O(1)$ 以下であるというものだ。2 つめの主張は、木に含まれない要素を探す場合について、やはり探索経路の長さの期待値に関する評価を与えている。 これらを比べると、木に含まれない要素を探すより、含まれる要素を探すほうが少しだけ速いことがわかる。

7.1.1 補題 7.1 の証明

補題 7.1 の証明では次の考察が鍵となる。ランダム二分探索木 T における値 $\mathsf{x} \in (-1,\mathsf{n})$ の探索経路に、$i < \mathsf{x}$ を満たす i をキーとするノードが含まれる必要十分条件は、T を作るランダムな置換において i が $\{i+1, i+2, \ldots, \lfloor \mathsf{x} \rfloor\}$ のい

[*1] $\mathsf{x}+1$ と $\mathsf{n}-\mathsf{x}$ は x 以上の要素の個数、x 以下の要素の個数と解釈できる。

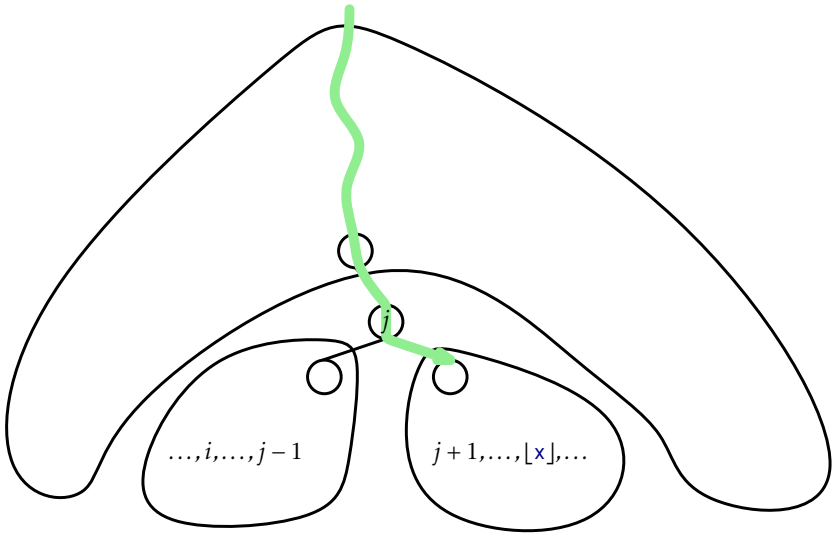


図 7.3: 値 $i < \mathtt{x}$ が $\mathtt{x}$ の探索経路中にあることの必要十分条件は i が $\{i, i+1, \ldots, \lfloor \mathtt{x} \rfloor\}$ のうち最初に木に加えられた要素であることである

ずれよりも前に現れることである。

そのことを確認するには、図 7.3 を見てほしい。$\{i, i+1, \ldots, \lfloor \mathtt{x} \rfloor\}$ のいずれかが追加されるまで、探索経路 $(i-1, \lfloor \mathtt{x} \rfloor + 1)$ に含まれる要素の探索経路は等しい（2 つの要素の探索経路が別々になるためには、一方以上かつ他方以下の要素がなければならないことを思い出そう）。ランダムな置換において最初に現れる $\{i, i+1, \ldots, \lfloor \mathtt{x} \rfloor\}$ の要素を j とする。j は、$\mathtt{x}$ の探索経路上にずっと存在していることに注意しよう。$j \neq i$ ならば、j を含むノード $\mathtt{u}_j$ は、i を含むノード $\mathtt{u}_i$ より先に作られる。そのあとで i を追加するときは、$i < j$ なので、$\mathtt{u}_j.\mathtt{left}$ を根とする部分木に $\mathtt{u}_i$ が追加される。一方、$\mathtt{x}$ の探索経路は、この部分木を通らない。なぜなら、この経路は $\mathtt{u}_j$ を訪問したあと、$\mathtt{u}_j.\mathtt{right}$ に向かうからである。

$i > \mathtt{x}$ の場合も、キー i が $\mathtt{x}$ の探索経路に含まれる必要十分条件は、T を作るランダムな置換において、i が $\{\lceil \mathtt{x} \rceil, \lceil \mathtt{x} \rceil + 1, \ldots, i-1\}$ のいずれよりも前に現れることである。

$\{0, \ldots, \mathtt{n}\}$ のランダムな置換では、そのうちの一部だけを取り出した部分列 $\{i, i+1, \ldots, \lfloor \mathtt{x} \rfloor\}$ および $\{\lceil \mathtt{x} \rceil, \lceil \mathtt{x} \rceil + 1, \ldots, i-1\}$ も、やはりそれぞれ対応する要素からなる列のランダムな置換になっている。このとき、T を作るランダム

な置換において、どちらの部分列でも先頭には各要素が等しい確率で現れる。そのため次の式が得られる。

$$\Pr\{i \text{ が } \mathsf{x} \text{ の探索経路に含まれる}\} = \begin{cases} 1/(\lfloor \mathsf{x} \rfloor - i + 1) & \text{if } i < \mathsf{x} \\ 1/(i - \lceil \mathsf{x} \rceil + 1) & \text{if } i > \mathsf{x} \end{cases}$$

以上の考察により、調和数を使った簡単な計算で補題 7.1 を証明できる。

補題 *7.1* の証明. I_i をインジケータ確率変数とする。I_i の値は、i が探索経路に現れるときは 1、そうでないときは 0 である。このとき探索経路の長さを次のように計算できる。

$$\sum_{i \in \{0,\ldots,\mathsf{n}-1\} \setminus \{\mathsf{x}\}} I_i$$

よって、$\mathsf{x} \in \{0,\ldots,\mathsf{n}-1\}$ なら探索経路の長さの期待値は次のように計算できる（図 7.4(a) 参照）。

$$\begin{aligned} \mathrm{E}\left[\sum_{i=0}^{\mathsf{x}-1} I_i + \sum_{i=\mathsf{x}+1}^{\mathsf{n}-1} I_i\right] &= \sum_{i=0}^{\mathsf{x}-1} \mathrm{E}[I_i] + \sum_{i=\mathsf{x}+1}^{\mathsf{n}-1} \mathrm{E}[I_i] \\ &= \sum_{i=0}^{\mathsf{x}-1} 1/(\lfloor \mathsf{x} \rfloor - i + 1) + \sum_{i=\mathsf{x}+1}^{\mathsf{n}-1} 1/(i - \lceil \mathsf{x} \rceil + 1) \\ &= \sum_{i=0}^{\mathsf{x}-1} 1/(\mathsf{x} - i + 1) + \sum_{i=\mathsf{x}+1}^{\mathsf{n}-1} 1/(i - \mathsf{x} + 1) \\ &= \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\mathsf{x}+1} \\ &\quad + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\mathsf{n}-\mathsf{x}} \\ &= H_{\mathsf{x}+1} + H_{\mathsf{n}-\mathsf{x}} - 2 \ . \end{aligned}$$

値 $\mathsf{x} \in (-1, n) \setminus \{0,\ldots,\mathsf{n}-1\}$ の場合も同様である（図 7.4(b) 参照）。 □

7.1.2 要約

次の定理はランダム二分探索木の性能をまとめたものだ。

定理 **7.1.** ランダム二分探索木の構築にかかる時間は $O(\mathsf{n} \log \mathsf{n})$ である。ランダム二分探索木における `find`(x) の実行時間の期待値は $O(\log \mathsf{n})$ である。

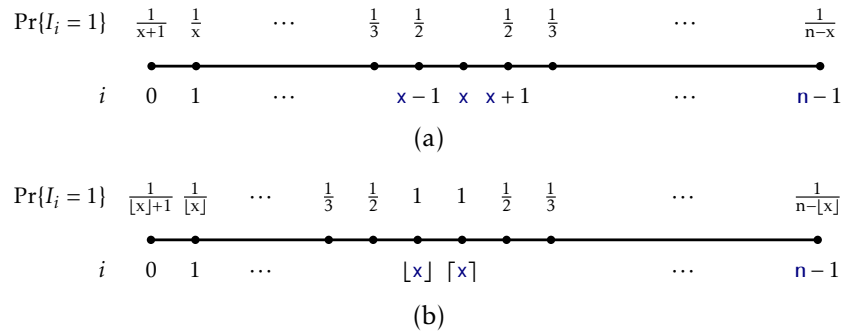


図 7.4: x の探索経路に各要素が現れる確率 (a)、x が整数のとき (b) x が整数でないとき

定理 7.1 における期待値は、ランダム二分探索木を作るための置換のランダム性に基づく。つまり、x をランダムに選ぶことには依存しておらず、任意の x について定理 7.1 は成り立つ。

7.2 Treap: 動的ランダム二分探索木の一種

前節で説明したランダム二分探索木の問題は、動的でないことである。すなわち、SSet インターフェースの add(x) および remove(x) をサポートしていない。この節では Treap というデータ構造を説明する。これは、補題 7.1 を使って SSet インターフェースを実装するデータ構造である [*2]。

Treap のノードは値 x を持つ。その点で Treap は BinarySearchTree に似ているが、それに加えて Treap のノードは一意な**優先度** p を持つ。この p はランダムに割り当てられる。

Treap
```
class TreapNode : public BSTNode<Node, T> {
  friend class Treap<Node,T>;
  int p;
```

[*2] Treap の名称は、このデータ構造が二分木 **tree**（6.2 節）であり、ヒープ **heap**（10 章）でもあることによる。

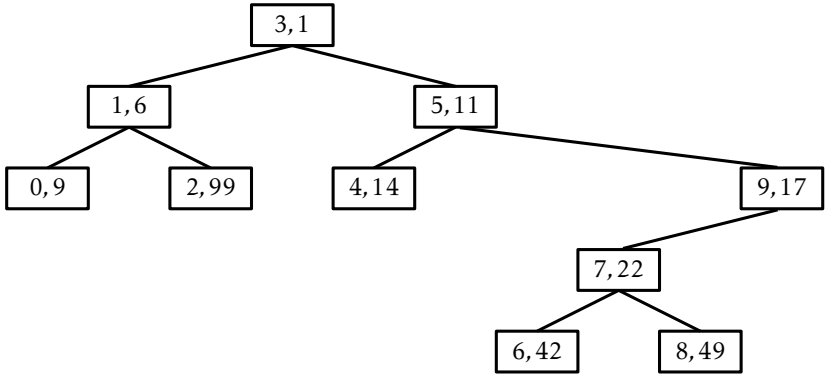


図 7.5: 整数 0,...,9 を含む Treap の例。ノード u を表す四角形の内部に u.x,u.p を表示してある

```
};
```

Treap のノードは、二分探索木の性質に加えて、次の**ヒープ性**(heap property)も満たす。

- 根でない任意のノード u について u.parent.p < u.p が成り立つ

言い換えると、どのノードの優先度も、そのいずれの子ノードの優先度よりも小さい。図 7.5 に Treap の例を示す。

Treap の形状は、ヒープ性と二分探索木の性質を共に満たすことから、キー x と優先度 p が決まれば一意に定まる。具体的には、ヒープ性から、最小の優先度を持つノードが Treap の根 r になる。さらに、二分探索木の性質から、r.x より小さなキーを持つノードは r.left を根とする部分木に含まれ、r.x より大きなキーを持つノードは r.right を根とする部分木に含まれる。

Treap の優先度の重要な特徴は、値 x に対して一意であり、かつランダムに割り当てられることだ。このことから、Treap の解釈には 2 つの等価なものが考えられる。まず、先ほど定義したように、Treap はヒープ性と二分探索木の性質を共に満たす木として解釈できる。もう 1 つの解釈は、Treap は優先度の昇順にノードが追加される BinarySearchTree であるというものである。例えば、空の BinarySearchTree に対して値 (x,p) を次の順で追加す

ると、図 7.5 の Treap が得られる。

$$\langle (3,1),(1,6),(0,9),(5,11),(4,14),(9,17),(7,22),(6,42),(8,49),(2,99)\rangle$$

Treap のノードにおける優先度はランダムに決まる。ということは、キーをランダムに置換して空の BinarySearchTree へと順番に追加しても同じことである。例えば、上の例は、次のようなキーの置換を BinarySearchTree へと順番に追加することに対応する。

$$\langle 3,1,0,5,9,4,7,6,8,2\rangle$$

ということは、Treap の形状がランダム二分探索木と同様にして決まるということである。特に、キー x をそのランク [*3] に置き換えれば、補題 7.1 を Treap にも適用できる。補題 7.1 を Treap に合わせて言い換えると次のようになる。

補題 **7.2.** n 個のキーからなる集合 S を保持する *Treap* について以下が成り立つ。

1. 任意の $x \in S$ について、x の探索経路の長さの期待値は $H_{r(x)+1} + H_{n-r(x)} - 2$ である
2. 任意の $x \notin S$ について、x の探索経路の長さの期待値は $H_{r(x)} + H_{n-r(x)}$ である

ここで、$r(x)$ は集合 $S \cup \{x\}$ における x のランクである。

補題 7.2 についても、期待値は優先度のランダム性に基づくものである。キーのランダム性について何らかの仮定は必要ない。

補題 7.2 より、Treap の `find`(x) は効率良く実装できる。しかし、本当に有用なのは、`add`(x) と `delete`(x) を実装できることだ。そのために、木を回転する操作を使ってヒープ性を保つ（図 7.6）。二分探索木の**回転（rotation）**とはノード w とその親 u について、二分探索木の性質を保ちながら w と u の親子関係を逆転する操作である。回転には、**右回転（right rotation）**と**左回転（left rotation）**の二種類があり、w が u の右の子なら右回転を、w が u の左の子なら左回転を使う。

実装にあたっては、左回転と右回転の 2 つの場合を処理し、コーナーケース（u が根である場合）にも注意しなければならない。そのため、コードは図 7.6

[*3] x のランクとは、x を集合 S の要素とするとき、S の要素のうち x より小さいものの個数である。

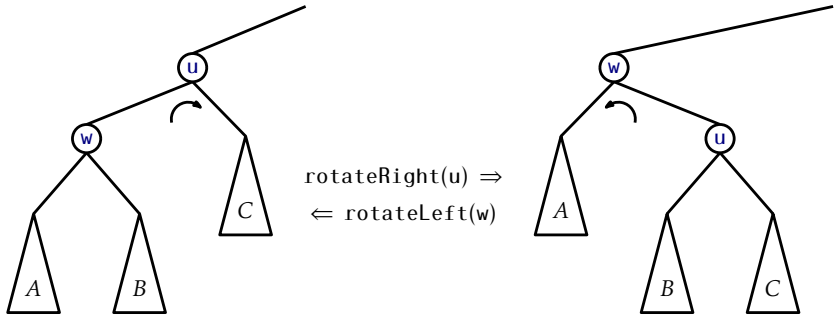


図 7.6: 二分探索木の左回転と右回転

から想像されるよりも少し長くなる。

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; }
}
```

なお、Treap の回転については、w の深さが 1 減って u の深さが 1 増えるという、重要な性質がある。

回転を使って add(x) を次のように実装できる。新しいノード u を作り、u.x = x とし、u.p を乱数で初期化する。u を BinarySearchTree の add(x) アルゴリズムを使って追加する。このとき u は Treap の葉になる。ここで、Treap は二分探索木の性質を満たすが、ヒープ性を満たすとは限らない。具体的には、u.parent.p > u.p の場合、Treap はヒープ性を満たさない。この場合は w=u.parent として回転を実行し、u を w の親にする。このとき、まだ u がヒープ性を満たさないなら、もう一度回転を実行する。そのたびに u の深さが 1 減るので、u が根になるか、u.parent.p < u.p を満たしたら、処理を終了する。

Treap

```
bool add(T x) {
  Node *u = new Node;
  u->x = x;
```

```
  u->p = rand();
  if (BinarySearchTree<Node,T>::add(u)) {
    bubbleUp(u);
    return true;
  }
  delete u;
  return false;
}
void bubbleUp(Node *u) {
  while (u->parent != nil && u->parent->p > u->p) {
    if (u->parent->right == u) {
      rotateLeft(u->parent);
    } else {
      rotateRight(u->parent);
    }
  }
  if (u->parent == nil) {
    r = u;
  }
}
```

図 7.7 に add(x) 操作の例を示す。

add(x) 操作の実行時間は、x の探索経路の長さと、新たに追加されたノード u を Treap におけるあるべき位置まで移動するための回転の回数から求まる。補題 7.2 より、探索経路の長さの期待値は $2\ln n + O(1)$ 以下である。さらに回転のたびに u の深さが減る。u が根になると処理が終了するので、回転の回数の期待値は探索経路の長さの期待値以下である。よって、Treap における add(x) の実行時間の期待値は $O(\log n)$ である（この操作における回転の回数の期待値が実は $O(1)$ であることを問 7.5 で見る）。

Treap における remove(x) は、add(x) の逆の操作である。x を含むノード u を探し、u が葉にくるまで下方向に回転を繰り返し、最後に u を取り外す。u を下方向に動かすとき、右に回転するか左に回転するかの選択肢があることに注意する。この選択は次の規則に従う。

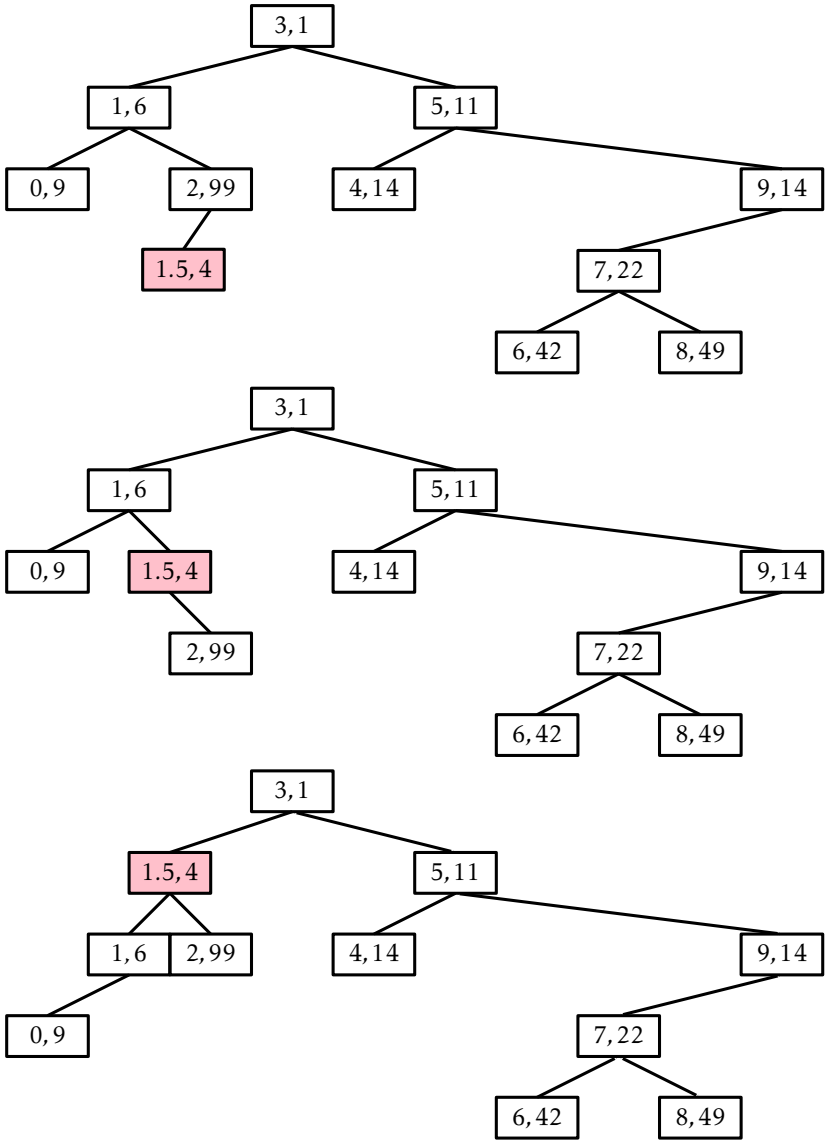


図 7.7: 図 7.5 の Treap に値 1.5 を追加する

1. u.left と u.right がいずれも null なら、u は葉なので回転の必要はない
2. u.left または u.right が null なら、null でないほうと回転して u を入れ替える
3. u.left.p < u.right.p ならば右に回転し、そうでないなら左に回転する

この規則のおかげで、Treap の連結性とヒープ性が保たれる。

Treap

```
bool remove(T x) {
  Node *u = findLast(x);
  if (u != nil && compare(u->x, x) == 0) {
    trickleDown(u);
    splice(u);
    delete u;
    return true;
  }
  return false;
}
void trickleDown(Node *u) {
  while (u->left != nil || u->right != nil) {
    if (u->left == nil) {
      rotateLeft(u);
    } else if (u->right == nil) {
      rotateRight(u);
    } else if (u->left->p < u->right->p) {
      rotateRight(u);
    } else {
      rotateLeft(u);
    }
    if (r == u) {
      r = u->parent;
    }
  }
```

```
}
```

図 7.8 に remove(x) の例を示す。

remove(x) の実行時間を解析するときは、add(x) の逆の操作になっている点に注目する。特に、x を同じ優先度 u.p で再挿入することを考えると、add(x) 操作によりちょうど同じ回数だけ回転が実行されることで Treap が remove(x) の直前の状態に戻る（図 7.8 を下から上に見ると、値 9 を Treap に追加している状態）。これは、大きさ n の Treap における remove(x) 操作の実行時間の期待値が、大きさ $n-1$ の Treap における add(x) 操作の実行時間の期待値に比例するということである。すなわち、remove(x) の実行時間の期待値は $O(\log n)$ である。

7.2.1 要約

次の定理に Treap の性能をまとめる。

定理 **7.2.** *Treap* は *SSet* インターフェースを実装する。*Treap* は add(x)、remove(x)、find(x) をサポートし、いずれの実行時間の期待値も $O(\log n)$ である。

Treap と SkiplistSSet を比べてみると面白いだろう。いずれも SSet の実装で、各操作の実行時間の期待値は $O(\log n)$ である。どちらのデータ構造でも、add(x)、remove(x) は、検索に続けて定数回のポインタの更新からなる（問 7.5 参照）。よって、どちらのデータ構造でも、探索経路の長さの期待値が性能を決める重要な値である。SkiplistSSet では、探索経路の長さの期待値は以下のようになる。

$$2\log n + O(1)$$

Treap では以下のようになる。

$$2\ln n + O(1) \approx 1.386\log n + O(1)$$

よって、Treap における探索経路のほうが短く、各操作についても Skiplist より Treap のほうがかなり速いと解釈できるだろう。4 章の問 4.7 で示したように、偏ったコイントスを使うことで、Skiplist における探索経路の長さの期待値を次のように減らせる。

$$e\ln n + O(1) \approx 1.884\log n + O(1)$$

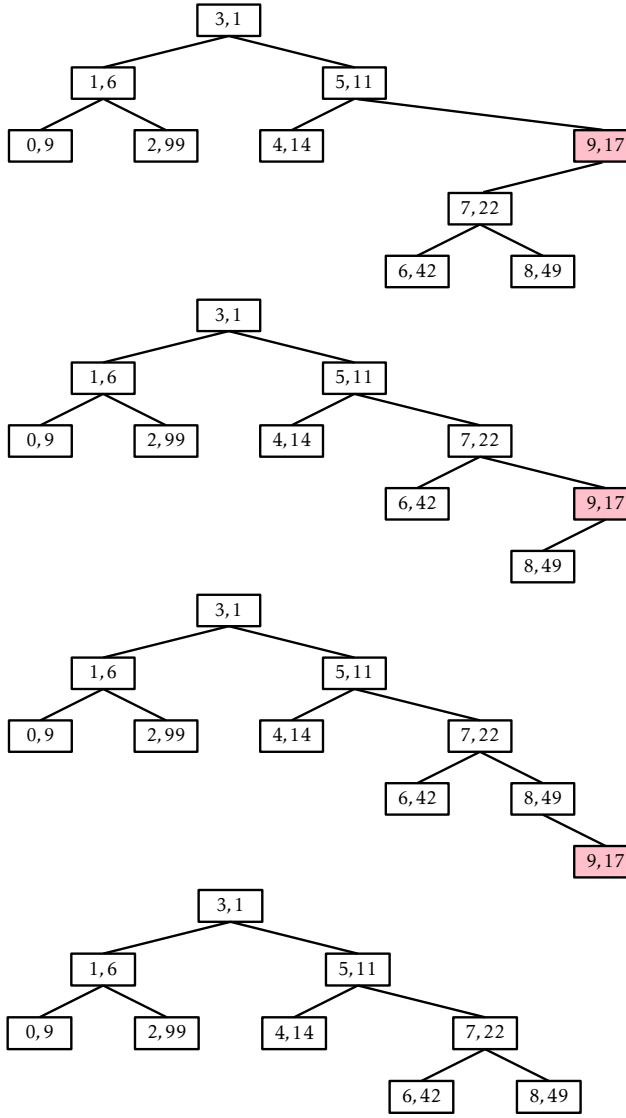


図 7.8: 図 7.5 の Treap から値 9 を削除する

この最適化を採用しても、SkiplistSSet における探索経路の期待値は、やはり Treap のそれよりだいぶ長い。

7.3 ディスカッションと練習問題

ランダム二分探索木についての研究は多岐にわたる。Devroye[19] では、補題 7.1 とそれに関連する結果とが証明されている。より強い事実もいくつか示されているが、その中で最も印象的なのは Reed[62] の成果である。この文献では、ランダム二分探索木の高さの期待値が次の式で表せることが示されている。

$$\alpha \ln n - \beta \ln \ln n + O(1)$$

ここで $\alpha \approx 4.31107$ は、$[2,\infty)$ 範囲での $\alpha \ln((2e/\alpha)) = 1$ の解であり、$\beta = \frac{3}{2\ln(\alpha/2)}$ である。さらに、高さの分散が一定であることも示されている。

Treap という名前は Seidel と Aragon[65] で提案された。この文献では、Treap といくつかの変種について述べられている。ただし、基本的な Treap のアイデアはそれ以前にも Vuillemin[74] で研究されている。こちらの文献では、このデータ構造のことを Cartesian tree と呼んでいる。

Treap のメモリ使用量を最適化する技法として、優先度 p を各ノードで明示的には保存しないというものがある。その代わりに、u のメモリアドレスのハッシュ値を u の優先度として用いる。その際、実用上は多くのハッシュ関数を問題なく利用できるものの、補題 7.1 の証明における重要なポイントを成り立たせるためには、ランダム化され、かつ min-wise independent **性**を満たすハッシュ関数を用いる必要がある。ここで、min-wise independent 性とは次の性質である。すなわち、相異なる任意の値 $x_1,\ldots,x_k$ について各ハッシュ値 $h(x_1),\ldots,h(x_k)$ が高い確率で相異なる値を取ること、具体的には、ある定数 c が存在し、任意の $i \in \{1,\ldots,k\}$ について次の式が成り立つことをいう。

$$\Pr\{h(x_i) = \min\{h(x_1),\ldots,h(x_k)\}\} \le c/k$$

この性質を持つハッシュ関数で実装が簡単、かつ高速なものとしては、tabulation hashing がある（5.2.3 節を参照せよ）。

優先度を各ノードで保持しない Treap の変種としては、Martínez と Roura [51] による動的ランダム二分探索木もある。この変種では、すべてのノード u に、u を根とする部分木の大きさ u.size を持たせる。add(x) および remove(x) のアルゴリズムはいずれも乱択化される。u を根とする部分木に、x を追加するアルゴリズムは、次のようになる。

1. 確率 1/(size(u)+1) で、x を通常通りに葉として追加する。その後、この部分木の根のほうへ x を持ち上げるために回転を実行する
2. そうでなければ（すなわち確率 1 − 1/(size(u)+1) で、x を u.left または u.right の適切なほうを根とする部分木に再帰的に追加する

上記の 1 つめの場合は、Treap における add(x) 操作で、x のノードが受け取るランダムな優先度のほうが u の部分木に含まれる size(u) 個の優先度のどれよりも小さい場合に相当する。1 つめの場合が起こる確率も、この場合とまったく同じである。

動的ランダム二分探索木からの値の削除は、Treap における削除とよく似ている。値 x を含むノード u を見つけ、回転を繰り返して深さを増やし、葉に到達したら木から切り離す。各ステップにおける回転が右か左かはランダムに決める。

1. 確率 u.left.size/(u.size − 1) で u において右回転を行う。すなわち u.left を部分木の根に持ってくる
2. 確率 u.right.size/(u.size − 1) で u において左回転を行う。すなわち u.right を部分木の根に持ってくる

こちらも、Treap における削除のアルゴリズムで u の左回転または右回転を実行した確率とまったく等しいことが簡単に確かめられる。

動的ランダム二分探索木には、Treap と比べると短所がある。要素の追加および削除の際にランダムな選択を多く実行すること、そして、部分木の大きさを保持しなければならないことだ。一方、動的ランダム二分探索木には、部分木の大きさを他の目的にも使えるという長所がある。例えば、ランクを $O(\log n)$ の期待実行時間で計算する際に流用できる（問 7.10 参照）。Treap における優先度には、木のバランスを保つ以外の用途はない。

問 **7.1.** 図 7.5 の Treap に 4.5 を優先度 7 で追加し、続いて値 7.5 を優先度 20 で追加する様子を図示せよ。

問 **7.2.** 図 7.5 の Treap から 5 と 7 を削除する様子を図示せよ。

問 **7.3.** 図 7.1 の右の木を生成する操作の列が 21,964,800 通りあることを示せ（ヒント：高さ h の完全二分木の個数に関する漸化式を作り、$h = 3$ の場合を評価せよ）。

問 **7.4.** permute(a) メソッドを設計、実装せよ。これは n 個の相異なる値を含む配列 a を入力とし、a のランダムな置換を返すメソッドである。実行時間

は $O(n)$ であり、n! 通りの置換がいずれも等確率で現れる必要がある。

問 **7.5.** 補題 7.2 を利用して、add(x) における回転の回数の期待値が $O(1)$ であることを示せ（remove(x) の場合も同様にわかる）。

問 **7.6.** Treap の実装を変更し、明示的に優先度を保持しないようにせよ。その際、優先度としては、各ノードのハッシュ値を利用せよ。

問 **7.7.** 二分探索木の各ノード u が、高さ u.height と、u を根とする部分木の大きさ u.size とを保持していると仮定する。

1. 左または右の回転を u で実行すると、回転によって影響を受けるすべてのノードにおいて、2 つの値をそれぞれ定数時間で更新できることを示せ。
2. 各ノードの深さも保持することにすると、上と同様の結果が成り立たなくなることを説明せよ。

問 **7.8.** n 要素からなる整列済み配列 a から Treap を構築するアルゴリズムを設計、実装せよ。この操作の実行時間は最悪の場合でも $O(n)$ である必要がある。また、このアルゴリズムで得られる Treap は、a の要素を順に add(x) メソッドで追加して得られる Treap と同一でなければならない。

問 **7.9.** この問題では、Treap において与えられたポインタの近くにあるノードを効率的に見つける方法を明らかにする。

1. 各ノードで、自身を根とする部分木における最大値と最小値が保持されているような Treap を設計、実装せよ。
2. 上記で追加した情報を使うことにより、x を含むノードからそれほど離れていない u を活用して find(x) を実行する、fingerFind(x,u) という操作を実装せよ。この操作は、u から上に向かって進み、$w.min \le x \le w.max$ を満たすノード w を見つけて、w から通常の方法で x を検索するというものである（fingerFind(x,u) の実行時間は $O(1+\log r)$ であることが示せる。ここで r は、x と u.x の間にある Treap の要素の数である）。
3. Treap の実装を拡張し、直近に実行した find(x) で見つかったノードから find(x) の探索を開始するようにせよ。

問 **7.10.** Treap におけるランクが i であるようなキーを返す操作 get(i) を設計、実装せよ（ヒント：各ノード u で、u を根とする部分木の大きさを保持す

るようにするとよい)。

問 **7.11.** TreapList を実装せよ。これは List インターフェースを Treap として実装したものだ。各ノードはリストのアイテムを保持し、行きがけ順で辿るとリストに入っている順でアイテムが見つかる。List の操作である get(i)、set(i,x)、add(i,x)、remove(i) の期待実行時間はいずれも $O(\log n)$ である必要がある。

問 **7.12.** split(x) 操作をサポートする Treap を設計、実装せよ。この操作は、Treap に含まれる x より大きいすべての値を削除し、削除された値をすべて含む新たな Treap を返すものである。
例：t2 = t.split(x) は、t から x より大きい値をすべて削除し、削除した値をすべて含む新たな Treap として t2 を返す。split(x) の実行時間の期待値は $O(\log n)$ である必要がある。
注意：この修正後も size() が定数時間で正しく動作するためには問 7.10 の実装が必要である。

問 **7.13.** absorb(t2) 操作をサポートする Treap を設計、実装せよ。この操作は、split(x) の逆の操作とみなせるもので、t2 という Treap からすべての値を削除し、それらをレシーバーに追加する。この操作では、t の最小値がレシーバーの最大値よりも大きいことを前提とする。なお、absorb(t2) の期待実行時間は $O(\log n)$ である必要がある。

問 **7.14.** この節で紹介した Martínez の動的ランダム二分探索木を実装せよ。自分の実装の性能を、Treap の実装と比較せよ。

第 8

スケープゴート木

この章では二分探索木の一種である ScapegoatTree を紹介する。スケープゴート（scapegoat）とは、罪を負わされたヤギ、転じて身代わりや生贄のことである。現実では、何かがうまくいかないとき、まず責任者を探そうとすることが多い。それと同じ発想に基づくデータ構造が ScapegoatTree である。責任者がはっきり決まったら、それをスケープゴートにして問題を解決させればよい。

ScapegoatTree では、**部分的な再構築**（partial rebuilding）によってバランスを保つ。部分的な再構築とは、部分木を一度分解し、非常にバランスのよい二分木として再構築するプロセスである。ここで、非常にバランスのよい二分木とは、サイズでバランスされている完全二分木のことである（問 6.6 参照）。また、完全二分木とは、任意の葉の高さの差が高々 1 である木のことである（第 10 章参照）。ノード u を根とする部分木を再構築して非常にバランスのよい二分木にするやり方はたくさんある。中でも単純なのは、u の部分木を辿ってすべてのノードを配列 a に集め、a から再帰的に非常にバランスのよい二分木を構築するやり方だ。m = a.length/2 とし、a[m] を新たな部分木の根とする。そして、a[0],...,a[m−1] は左の部分木に、a[m+1],...,a[a.length−1] は右の部分木にそれぞれ再帰的に配置する。

ScapegoatTree

```
void rebuild(Node *u) {
  int ns = BinaryTree<Node>::size(u);
  Node *p = u->parent;
  Node **a = new Node*[ns];
  packIntoArray(u, a, 0);
```

```
  if (p == nil) {
    r = buildBalanced(a, 0, ns);
    r->parent = nil;
  } else if (p->right == u) {
    p->right = buildBalanced(a, 0, ns);
    p->right->parent = p;
  } else {
    p->left = buildBalanced(a, 0, ns);
    p->left->parent = p;
  }
  delete[] a;
}
int packIntoArray(Node *u, Node **a, int i) {
  if (u == nil) {
    return i;
  }
  i = packIntoArray(u->left, a, i);
  a[i++] = u;
  return packIntoArray(u->right, a, i);
}
```

rebuild(u) の実行時間は O(size(u)) である。結果として得られる部分木は高さが最小である。すなわち、size(u) 個のノードを持つこの木より低い木は存在しない。

8.1 ScapegoatTree：部分的に再構築する二分探索木

ScapegoatTree は二分探索木である。木に含まれるノードの数 n に加えて、ノードの数に関する上界を表すカウンタ q を保持する。

ScapegoatTree

```
int q;
```

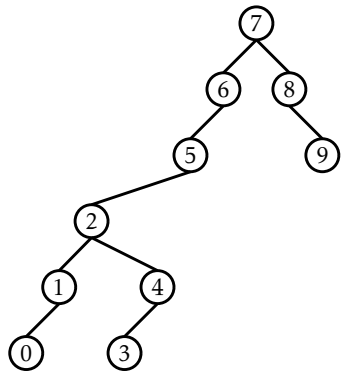


図 8.1: 10 個のノードを持ち、高さが 5 である ScapegoatTree の例

n と q は常に次の関係を満たす。

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

ScapegoatTree には、木の高さがノード数の対数で抑えられるという性質がある。具体的には、ScapegoatTree の高さは常に $\log_{3/2} q$ 以下であり、この値は以下の性質を満たす。

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

このような制約があるものの、ScapegoatTree の見た目は意外なほど偏ることもある。例えば、$q = n = 10$ で高さが $5 < \log_{3/2} 10 \approx 5.679$ の ScapegoatTree を図 8.1 に示す。

ScapegoatTree における `find(x)` の実装では、BinarySearchTree のアルゴリズムをそのまま使う（6.2 節参照）。実行時間は木の高さに比例し、これは (8.1) により $O(\log n)$ である。

`add(x)` の実装では、まず n と q を 1 ずつ増やし、それから x を BinarySearchTree に追加するアルゴリズムをそのまま使う。すなわち、x を探し、新たな葉 u を追加し、$u.x = x$ とする。このとき、たまたま u の深さが $\log_{3/2} q$ 以下になっているなら、それ以上何もしなくてよい。

しかし、$\mathtt{depth}(u) > \log_{3/2} q$ になっていることもある。この場合には高さを減らさなければならないが、深さが $\log_{3/2} q$ を超えてしまうノードはいま加え

た u だけなので、さほど難しくはない。木を上に向かって辿りながら、**スケープゴート**となるノード w を探す。スケープゴート w は、特にバランスされていないノードである。根から u へと至る経路上の子（w.child）との間で、次のような性質を持つノードが、w である。

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

このようなスケープゴート w が存在するかどうかについては、すぐあとで証明する。いまは w が存在するものとしよう。スケープゴート w が見つかったら、w を根とする部分木を再構築することで、全体を非常にバランスのよい二分木にする。w を根とする部分木は、(8.2) のような性質なので、u を加える前から完全二分木ではなかった。そこで w を再構築するときは、ScapegoatTree の高さが再び $\log_{3/2} q$ 以上になるように、w の高さを 1 以上減らす。

ScapegoatTree

```
bool add(T x) {
  // まずは深さを調べながら素朴に挿入する
  Node *u = new Node;
  u->x = x;
  u->left = u->right = u->parent = nil;
  int d = addWithDepth(u);
  if (d > log32(q)) {
    // 深すぎるなら、スケープゴートを見つける
    Node *w = u->parent;
    int a = BinaryTree<Node>::size(w);
    int b = BinaryTree<Node>::size(w->parent);
    while (3*a <= 2*b) {
      w = w->parent;
      a = BinaryTree<Node>::size(w);
      b = BinaryTree<Node>::size(w->parent);
    }
    rebuild(w->parent);
  } else if (d < 0) {
    delete u;
```

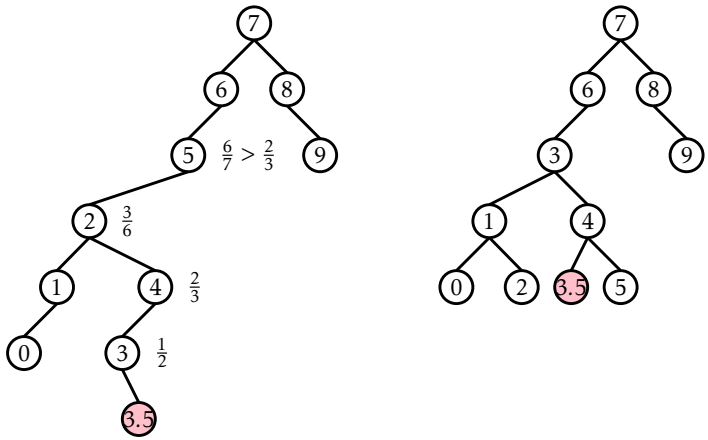


図 8.2: ScapegoatTree に 3.5 を追加する。このとき木の高さは 6 に増え、$6 > \log_{3/2} 11 \approx 5.914$ より (8.1) が成り立たない。スケープゴートは値 5 を含むノードで見つかる

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

スケープゴート w を見つけるコストと、w を根とする部分木を再構築するコストを無視すれば、add(x) の実行時間のうち支配的なのは最初の検索にかかるコストであり、これは $O(\log q) = O(\log n)$ である。スケープゴートを見つけて部分木を再構築するコストは、次項で償却解析を使って説明する。

ScapegoatTree における remove(x) の実装はとても単純である。x を探し、BinarySearchTree におけるアルゴリズムを使って削除する（これによって木の高さが増えることはない）。そのうえで、n を 1 つ小さくし、q はそのままにしておく。最後に $q > 2n$ かどうかを確認し、もしそうなら**木全体を再構築**して非常にバランスのよい二分木とし、$q = n$ とする。

ScapegoatTree

```
bool remove(T x) {
  if (BinarySearchTree<Node,T>::remove(x)) {
```

```
    if (2*n < q) {
      if (r != NULL)
        rebuild(r);
      q = n;
    }
    return true;
  }
  return false;
}
```

ここでも、再構築のコストを無視すれば、remove(x) の実行時間は木の高さに比例し、$O(\log n)$ である。

8.1.1 正しさの証明と実行時間の解析

ここでは ScapegoatTree の各操作の正しさと償却実行時間とを解析する。まず、add(x) 操作によって (8.1) を満たさない状態になった場合には必ずスケープゴートが見つかることを示すことにより、ScapegoatTree の各操作の正しさを証明する。

補題 **8.1.** *ScapegoatTree* において深さ $d > \log_{3/2} q$ となるノードを u とする。このとき、u から根への経路上に次の条件を満たすノード w が存在する。

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

証明. 背理法で示す。u から根への経路上の任意のノード w について次の式が成り立つと仮定する。

$$\frac{\mathtt{size}(w)}{\mathtt{size}(\mathtt{parent}(w))} \le 2/3$$

また、根から u への経路を $r = u_0, \ldots, u_d = u$ とする。このとき $\mathtt{size}(u_0) = n$、$\mathtt{size}(u_1) \le \frac{2}{3}n$、$\mathtt{size}(u_2) \le \frac{4}{9}n$ であり、より一般には次の式が成り立つ。

$$\mathtt{size}(u_i) \le \left(\frac{2}{3}\right)^i n$$

ここで、$\mathtt{size}(\mathtt{u}) \geq 1$ より、以下のようになる。

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

こうして矛盾が導かれた。

続いて、まだ説明していない実行時間について解析する。解析は、スケープゴートとなるノードを探すときに $\mathtt{size}(\mathtt{u})$ を呼び出すコストと、スケープゴート $\mathtt{w}$ を見つけたときに $\mathtt{rebuild}(\mathtt{w})$ を呼び出すコストに分けて考える。これら 2 つの操作の間には次のような関係がある。

補題 **8.2.** *ScapegoatTree* の $\mathtt{add}(\mathtt{x})$ において、スケープゴート $\mathtt{w}$ を見つけて $\mathtt{w}$ を根とする部分木を再構築するコストは $O(\mathtt{size}(\mathtt{w}))$ である。

証明. スケープゴートとなるノード $\mathtt{w}$ を見つけたあと、$\mathtt{w}$ を根とする部分木を再構築する際の実行時間は、$O(\mathtt{size}(\mathtt{w}))$ である。スケープゴートを見つけるには、$\mathtt{u}_k = \mathtt{w}$ を見つけるまで、$\mathtt{u}_0, \ldots, \mathtt{u}_k$ に対して順番に $\mathtt{size}(\mathtt{u})$ を実行する。しかし、$\mathtt{u}_k$ はこの列における最初のスケープゴートとなるノードなので、任意の $i \in \{0, \ldots, k-2\}$ について次の式が成り立つ。

$$\mathtt{size}(\mathtt{u}_i) \leq \frac{2}{3}\mathtt{size}(\mathtt{u}_{i+1})$$

よって、$\mathtt{size}(\mathtt{u})$ を呼び出すコストをすべて合計すると次のようになる。

$$\begin{aligned}
O\left(\sum_{i=0}^{k} \mathtt{size}(\mathtt{u}_{k-i})\right) &= O\left(\mathtt{size}(\mathtt{u}_k) + \sum_{i=0}^{k-1} \mathtt{size}(\mathtt{u}_{k-i-1})\right) \\
&= O\left(\mathtt{size}(\mathtt{u}_k) + \sum_{i=0}^{k-1} \left(\frac{2}{3}\right)^{i} \mathtt{size}(\mathtt{u}_k)\right) \\
&= O\left(\mathtt{size}(\mathtt{u}_k)\left(1 + \sum_{i=0}^{k-1} \left(\frac{2}{3}\right)^{i}\right)\right) \\
&= O(\mathtt{size}(\mathtt{u}_k)) = O(\mathtt{size}(\mathtt{w}))
\end{aligned}$$

最後の行では等比数列の和を計算している。 $\square$

最後に、m 個の操作を順に実行するときの $\mathtt{rebuild}(\mathtt{u})$ の合計コストの上界を示す。

補題 **8.3.** 空の *ScapegoatTree* に対して、m 個の add(x) および remove(x) からなる操作の列を順に実行するとき、rebuild(u) に要する時間の合計は $O(m \log m)$ である。

証明. **出納法**（credit scheme）を使って示す。各ノードは預金を持っていると考える。預金が c だけあれば再構築のための支払いができる。預金の合計は $O(m \log m)$ で、rebuild(u) の呼び出しにかかるコストは、u が蓄えている預金を使って支払われる。

ノード u を挿入、削除する際に、根から u への経路上にある各ノードの預金を 1 だけ増やす。こうして 1 回の操作で増える預金の合計は最大 $\log_{3/2} q \le \log_{3/2} m$ である。削除の際には多めに預金を蓄えることになる。こうして最大で $O(m \log m)$ を預金する。あとは、これだけの預金ですべての rebuild(u) の支払いに十分であることを示せばよい。

挿入の際に rebuild(u) を実行するなら、u がスケープゴートである。次のように仮定しても一般性を失わない。

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

ここで、次の事実が成り立っている。

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

これを使うと、次の式が成り立つ。

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

このとき、さらに次の式が成り立つ。

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

u を含む部分木が直前に再構築されたとき（もし u を含む部分木が一度も再構築されていなければ、u が挿入されたとき）、次の式が成り立つ。

$$\mathtt{size(u.left)} - \mathtt{size(u.right)} \le 1$$

よって、u.left と u.right に影響を与えた add(x) および remove(x) の数の合計は次の値以上である。

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

u には少なくともこれだけの預金が蓄えられており、rebuild(u) に必要な $O(\texttt{size}(u))$ の支払いには十分である。

削除において rebuild(u) が呼ばれるときは、$q > 2n$ である。この場合、$q - n > n$ だけ余分に預金が蓄えられており、根の再構築に必要な $O(n)$ の支払いには十分である。 □

8.1.2 要約

次の定理に ScapegoatTree の性能をまとめる。

定理 8.1. *ScapegoatTree は SSet インターフェースを実装する。rebuild(u) のコストを無視すると、ScapegoatTree は add(x)、remove(x)、find(x) をいずれも $O(\log n)$ の時間で実行できる。さらに、m 個の add(x) および remove(x) からなる操作の列を空の ScapegoatTree に対して順に実行するとき、rebuild(u) に要する時間の合計は $O(m \log m)$ である。*

8.2 ディスカッションと練習問題

scapegoat tree という名称は、このデータ構造を定義して解析した Galperin と Rivest により提案された [33]。ただし、同じデータ構造は Andersson によって先に発見されており、そこでは general balanced trees と呼ばれていた [5, 7]。これは、このデータ構造が、高さが小さいならどのような形状も取れることによる。

ScapegoatTree を実装し、実験してみると、この本で紹介した他の SSet の実装と比べてかなり遅いことがわかる。高さの上界は以下である。

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

これは Skiplist の探索経路の長さの期待値よりも小さく、Treap ともほぼ同じなので、ScapegoatTree が遅いのは意外かもしれない。ScapegoatTree の最適化として、各ノードに部分木の大きさを保持したり、すでに計算した部分木のサイズを再利用したりできる（問 8.5 と問 8.6 を参照）。これらの最適化をしても、依然として add(x) や delete(x) を繰り返し実行すると、ScapegoatTree は他の SSet の実装より遅いことがあるだろう。

ScapegoatTree が、本章で紹介した他の SSet の実装と比べて性能が芳しくないのは、再構築にかかる時間が長いからである。本章で紹介した他の

SSet の実装では、n 個の操作の間にデータ構造の再構築に費やす時間は $O(n)$ であった。一方、問 8.3 より、ScapegoatTree では n 個の操作を実行する間に $n \log n$ オーダーの時間を rebuild(u) に費やす。これは、データ構造の再構築をすべて rebuild(u) で行っていることに起因する [20]。

性能が劣るとはいえ、ScapegoatTree が正しい選択になる場合もある。それは、各ノードに追加のデータを入れる場合である。特に、回転操作では定数時間で更新できないが、rebuild(u) では定数時間で更新できる場合がある。そのような場合には ScapegoatTree（もしくは部分的な再構築を行う他のデータ構造）を選ぶのがよい。応用例を問 8.11 で取り上げる。

問 **8.1.** 図 8.1 の ScapegoatTree に 1.5、1.6 を順に追加する様子を描け。

問 **8.2.** 空の ScapegoatTree に 1, 5, 2, 4, 3 を順に追加する様子を描け。加えて、補題 8.3 の証明で使った預金がどう移動し、どのように使われるかも説明せよ。

問 **8.3.** 空の ScapegoatTree に対し、$x = 1, 2, 3, \ldots, n$ について順に add(x) を呼び出す。このとき、ある定数 $c > 0$ が存在し、rebuild(u) に要する時間の合計が $cn \log n$ 以上であることを示せ。

問 **8.4.** ScapegoatTree における探索経路の長さは $\log_{3/2} q$ を超えない。

1. ScapegoatTree を修正し、$1 < b < 2$ を満たすパラメータ b について探索経路の長さが $\log_b q$ を超えないデータ構造を設計、解析、実装せよ。
2. find(x)、add(x)、remove(x) の償却コストが n と b の関数としてどう表せるか、解析および実験によって考えよ。

問 **8.5.** ScapegoatTree の add(x) メソッドを修正し、すでに計算した部分木の大きさは再計算しないように無駄を省け。size(w) を計算するとき、size(w.left) か size(w.right) はすでに計算しているので、このような最適化が可能である。修正前後での性能を比較せよ。

問 **8.6.** ScapegoatTree の変種として、明示的に各ノードを根とする部分木の大きさを蓄えるものを実装せよ。もともとの ScapegoatTree や問 8.5 での実装と、ここでの実装とを性能比較せよ。

問 **8.7.** この章の最初に説明した rebuild(u) を、再構築する部分木に含まれるノードを蓄える配列を使わずに再実装せよ。代わりに、まずは再帰を使ってこれらのノードを連結リストにし、この連結リストを非常にバランスのよい

二分木に変換せよ（いずれのステップにも華麗な再帰を使った実装がある）。

問 **8.8.** WeightBalancedTree を設計、実装せよ。このデータ構造では、根以外の各ノード u が**バランス条件** size(u) ≤ (2/3)size(u.parent) を満たす。WeightBalancedTree における add(x) および remove(x) 操作は、通常の BinarySearchTree における各操作とほぼ同じだが、ノード u でバランス条件が成り立たないときには u.parent を根とする部分木が再構築される。

さらに、WeightBalancedTree の償却実行時間が $O(\log n)$ であることを示せ。

問 **8.9.** CountdownTree を設計、実装せよ。このデータ構造では、各ノード u は**タイマー** u.t を持っている。CountdownTree における add(x) および remove(x) 操作は、通常の BinarySearchTree における各操作とほぼ同じだが、いずれかの操作が u の部分木に影響を与えるときには u.t を 1 つ小さくする。u.t = 0 のときは、u を根とする部分木を非常にバランスのよい二分木へと再構築する。ノード u が再構築に関与する（u が再構築されるか、u の祖先のうちの 1 つが再構築される）とき、u.t は size(u)/3 にリセットされる。

さらに、CountdownTree の償却実行時間が $O(\log n)$ であることを示せ（ヒント：バランスに関するある種の不変条件を任意のノード u が満たすことを最初に示すとよい）。

問 **8.10.** DynamiteTree を設計、実装せよ。このデータ構造では、すべてのノード u は、u を根とする部分木の大きさを u.size として保持する。add(x) および remove(x) 操作は、通常の BinarySearchTree とほぼ同じだが、いずれかの操作が u の部分木に影響を与えるときには u が確率 1/u.size で**爆発**する。u が爆発したときは、u を根とする部分木を非常にバランスのよい二分木に再構築する。

さらに、DynamiteTree の実行時間の期待値が $O(\log n)$ であることを示せ。

問 **8.11.** 要素の列を保持するデータ構造 Sequence を設計、実装せよ。これは次のような操作を提供する。

- addAfter(e)：要素 e の次に新たな要素を追加して、新たに追加した要素を返す（e が null なら新たな要素を列の先頭に追加する）
- remove(e)：e を列から削除する
- testBefore(e1,e2)：e1 が e2 の前にある場合、またその場合に限り、true を返す

はじめの 2 つの操作の償却実行時間は $O(\log n)$ でなければならない。3 つめの操作は定数時間でなければならない。

Sequence は、列の中の順序を使って ScapegoatTree のようにデータを蓄えることで実装できる。testBefore(e1,e2) を定数時間で実装するには、要素 e を根から e への経路を符号化した整数でラベル付けする。そうして testBefore(e1,e2) で e1 と e2 のラベルを比較すればよい。

第 9

赤黒木

この章では、赤黒木（red-black tree）という、木の高さが要素数の対数で抑えられる二分木を紹介する。赤黒木は最も広く使われるデータ構造のひとつである。Java のコレクションフレームワークや C++ の標準テンプレートライブラリのいくつかの実装など、多くのライブラリで探索のための基本的なデータ構造として採用されている。また、Linux の OS カーネル内部でも使われている。赤黒木が広く利用されている理由はいくつかある。

1. n 個の要素を含む赤黒木の高さが $2\log n$ 以下になる
2. add(x) および remove(x) を**最悪の場合でも** $O(\log n)$ の時間で実行できる
3. add(x) および remove(x) における回転の回数は、償却すると定数である

上記のうち、はじめの 2 つの性質が、Skiplist、Treap、Scapegoat に対する赤黒木の優位性を示している。Skiplist と Treap ではランダム化を利用するので、実行時間 $O(\log n)$ は期待値にすぎない。Scapegoat には高さに関する保証があるものの、add(x) と remove(x) の実行時間 $O(\log n)$ は償却実行時間にすぎない。上記のうち 3 つめの性質は、おまけである。この性質から、要素 x を追加および削除するときにかかる主な時間は、x を見つける処理によることがわかる [*1]。

しかし、赤黒木の優れた性質には代償がある。実装が複雑なのである。高さの上界を $2\log n$ に保つのは容易ではない。さまざまな可能性を慎重に解析する必要がある。そして、そのすべてにおいて、実装が正しくなければならない

[*1] スキップリストや Treap も平均的にはこの性質を持つ。問 4.6 および問 7.5 参照。

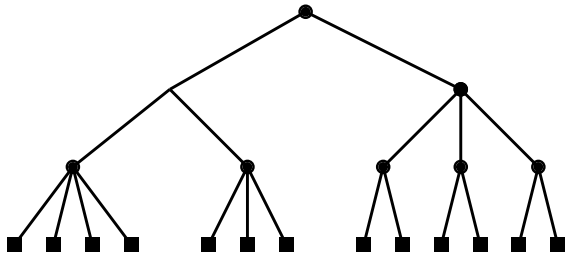


図 9.1: 高さ 3 である 2-4 木

のである。回転を 1 つ間違えたり、色を間違えたりすると、わかりにくいバグが発生することになる。

ここでは、いきなり赤黒木の実装に取り掛からず、まずは関連する背景知識として 2-4 木について説明する。これにより、赤黒木が発見された経緯と、なぜ赤黒木を効率的に管理できるのかを理解する助けとなるだろう。

9.1 2-4 木

2-4 木は次の性質を持つ根付き木である。

性質 **9.1** (高さ)**.** すべての葉の深さは等しい

性質 **9.2** (次数)**.** すべての内部ノードは 2 個以上 4 個以下の子ノードを持つ

2-4 木の例を図 9.1 に示す。上記の性質より、2-4 木の高さは葉の数の対数で抑えられる。

補題 **9.1.** n 個の葉を持つ *2-4* 木の高さは $\log n$ 以下である。

証明**.** 内部ノードの子の数は 2 以上なので、2-4 木の高さを h とすると葉の数は 2^h 以上である。

$$n \geq 2^h$$

両辺の対数を取ると $h \leq \log n$ である。 □

9.1.1 葉の追加

2-4 木に葉を追加するのは簡単である（図 9.2）。別の葉の親ノードである w の子として葉 u を追加したいときは、単に u を w の子とする。このとき、高さの制約は保たれるが、次数の制約が成り立たなくなるかもしれない。つまり、u を追加する前に w が子を 4 つ持っていたなら、w の子の数が 5 となってしまう。この場合、w を**分割**し、子を 2 つ持つノード w と、子を 3 つ持つノード w' とする。このとき w' には親がいないので、w の親を w' の親とする。この処理を再帰的に繰り返す。つまり、先の処理の結果として w の親が持つ子の数が多くなりすぎるかもしれないので、その場合には w の親を分割する。この処理を、子の数が 4 未満のノードができるか、根 r を r と r' に分割する状況に至るまで繰り返す。後者の場合には新しい根を作り、r と r' をその新しい根の子とする。そのときにはすべての葉の深さが同時に増えることになるので、高さの性質はやはり保たれる [*2]。

2-4 木の高さは常に $\log n$ 以下なので、葉の追加は $\log n$ ステップ以下で完了する。

9.1.2 葉の削除

2-4 木から葉を削除するには少し工夫が必要である（図 9.3）。葉 u をその親 w から削除するときは、単に u を削除する。その直前に w の子が 2 つだけだったなら、w の子が 1 つだけになるので、次数についての制約を満たさなくなる。

これを修正するため、w の兄弟 w' を調べる。w の親が持つ子の数は 2 以上なので、兄弟 w' は必ず存在する。w' の子が 3 つまたは 4 つなら、そのうち 1 つを w' から w に移す。すると、w の子は 2 つ、w' の子は 2 つもしくは 3 つになるので、処理を終える。

一方、w' の子が 2 つなら、w と w' を**併合**して子を 3 つ持つノード w とする。そのうえで、w' を w' の親から取り除く。この処理は、自分自身もしくは兄弟が子を 3 つ以上持つようなノード u が見つかるか、根に到達したら終了する。後者の場合、根の子は 1 つだけなので、その子を新たな根として以前の根を削除する。この場合もすべての葉の高さが同時に減るので、高さの性質はやはり保たれる。

2-4 木の高さは常に $\log n$ 以下なので、葉の削除もやはり $\log n$ ステップ以下で完了する。

[*2] 訳注：これに似た議論は 14.2.2 節において現れる。

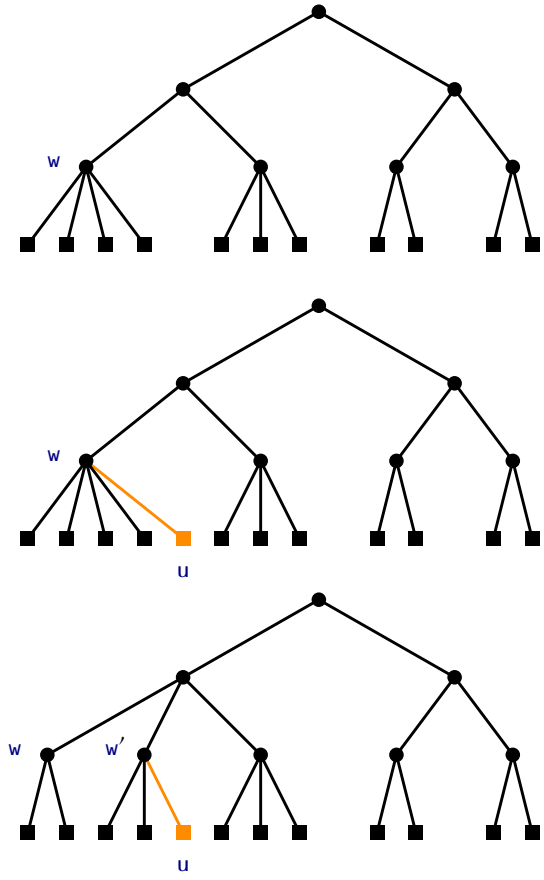


図 9.2: 2-4 木に葉を追加する。w.parent の次数が 4 未満なので、この処理は 1 回の分割の後終了する

9.2 RedBlackTree：2-4 木をシミュレートする二分木

赤黒木は、各ノード u が**赤**か**黒**の**色**を持つ二分探索木である。赤ノードは 0 で、黒ノードは 1 で表現される。

RedBlackTree

```
class RedBlackNode : public BSTNode<Node, T> {
  friend class RedBlackTree<Node, T>;
```

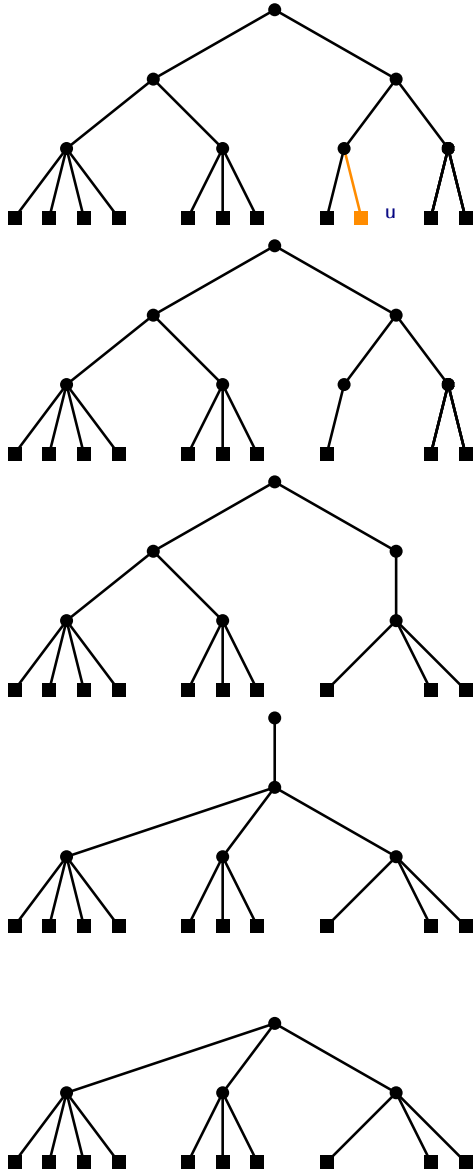


図 9.3: 2-4 木から葉を削除する。u の祖先とその兄弟は、いずれも子が 2 つなので、この処理は根まで繰り返す

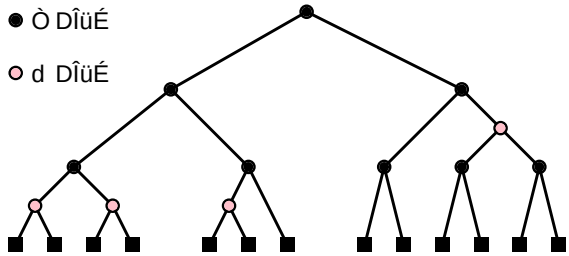


図 9.4: 黒の高さが 3 である赤黒木の例。正方形は外部ノード（nil）を表す

```
  char colour;
};
int red = 0;
int black = 1;
```

赤黒木においては、すべての操作の前後で、次の 2 つの性質が満たされる。どちらの性質も、赤および黒の色と、0 および 1 の数値の、両方を使って定義される。

性質 **9.3** (黒の高さの性質)**.** 葉から根への経路には、いずれも黒ノードが同じ数だけ含まれる。（葉から根への経路について、色の総和はすべて等しい）

性質 **9.4** (赤の辺の性質)**.** 赤の辺はない。すなわち、赤ノード同士は隣接しない。（根でない任意のノード u について、u.colour + u.parent.colour ≥ 1 が成り立つ。）

これらの性質は、根 r がどちらの色であっても満たされる。そこで、ここでは根が黒であると仮定する。また、赤黒木を更新するアルゴリズムでは、根は黒のまま保たれるものとする。さらに、赤黒木を単純化するためのもう 1 つの工夫として、外部ノード（nil で表す）を黒ノードとして扱う。図 9.4 に赤黒木の例を示す。

9.2.1 赤黒木と 2-4 木

赤黒木は、前項で定義した「黒の高さの性質」と「赤の辺の性質」を保ちながら効率的に更新できる。この事実を初めて聞くと驚くことだろう。だが、そ

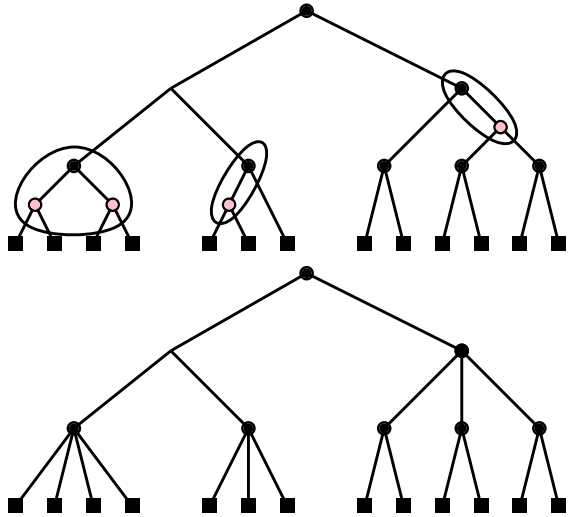


図 9.5: 任意の赤黒木には、対応する 2-4 木が存在する

もそもこれらの性質がなんの役に立つのかわからない人もいるだろう。実は、赤黒木は、2-4 木を二分木として効率的にシミュレートするように設計されているのである。

図 9.5 を見てほしい。n 個のノードを持つ赤黒木 T に対し、「すべての赤ノード u を取り除き、u の 2 つの子を両方とも u の親（黒ノード）に直接接続する」という操作を実行する。こうして得られる木 T' は、黒ノードだけを含む。

T' の内部ノードは、すべて 2〜4 個の子を持つ。黒い子を 2 つ持っていた黒ノードは、変換後も黒い子を 2 つ持つ。赤い子と黒い子を 1 つずつ持っていた黒ノードは、変換後は黒い子を 3 つ持つ。赤い子を 2 つ持っていた黒ノードは、変換後は黒い子を 4 つ持つ。加えて、「黒の高さの性質」より、T' の任意の葉から根への経路の長さは同じである。つまり、T' は 2-4 木なのである！

2-4 木 T' は $n+1$ 個の葉を持ち、各葉は赤黒木の $n+1$ 個の外部ノードと対応する。よって、この木の高さは $\log(n+1)$ 以下である。2-4 木のすべての葉から根への経路は、赤黒木 T' における根から外部ノードへの経路に対応する。この経路の最初と最後のノードは黒で、2 つの内部ノードのうち赤は 1 つ以下なので、この経路にあるノードのうち黒は $\log(n+1)$ 個以下、赤は

$\log(n+1)-1$ 個以下である。よって、任意の $n \geq 1$ について、根から任意の**内部**ノードへの経路のうちで最長のものの長さは、次の左辺の値以下である。

$$2\log(n+1)-2 \leq 2\log n$$

このことから、赤黒木の最も重要な次の性質を示せる。

補題 **9.2.** n 個のノードからなる赤黒木の高さは $2\log n$ 以下である。

2-4 木と赤黒木の関係がわかれば、赤黒木を維持したまま効率的に要素の追加や削除が可能であると思えてくるだろう。

これまでの章で、BinarySearchTree に対して要素を追加するには、新たな葉を追加すればいいことを見てきた。よって、赤黒木における add(x) を実装するには、2-4 木において子を 5 つ持つノードの分割をシミュレートする方法があればよい。子を 5 つ持つ 2-4 木のノードは、赤い子を 2 つ持つ黒ノード w であって、2 つの赤い子のうちの一方がさらに赤い子を持つようなものである。w を「分割」するには、w を赤ノードにし、w の子をいずれも黒ノードにすればよい。例を図 9.6 に示す。

remove(x) を実装するには、2 つのノードを併合する方法と、兄弟から子を借りる方法があればよい。2 つのノードの併合は、図 9.6 で示した分割とは逆の処理であり、黒い兄弟をいずれも赤ノードにし、その共通の赤い親を黒ノードにすればよい。兄弟から子を借りる操作が最も複雑で、回転と色の変更が両方とも必要になる。

もちろん「赤の辺の性質」と「黒の高さの性質」をいずれも満たす必要がある。それが可能なことは驚くに値しないだろうが、2-4 木をシミュレートして赤黒木とするには、考慮すべき場合分けがかなり多くなる。ある程度までいったら、背景となる 2-4 木は無視してしまい、赤黒木の性質を保つことだけを考えてシンプルな実装にする。

9.2.2 左傾赤黒木

赤黒木の定義は 1 つではない。むしろ、add(x) と remove(x) の前後で「赤の辺の性質」と「黒の高さの性質」を維持できるようなデータ構造は複数ある。データ構造が違えば、実装方法も異なる。この節で実装するデータ構造を、RedBlackTree と呼ぶことにしよう。RedBlackTree は赤黒木の一種であり、左傾性（left-leaning）と呼ぶ次の性質を満たす。

性質 **9.5** (左傾性)**.** 任意のノード u について、u.left が黒ならば u.right も

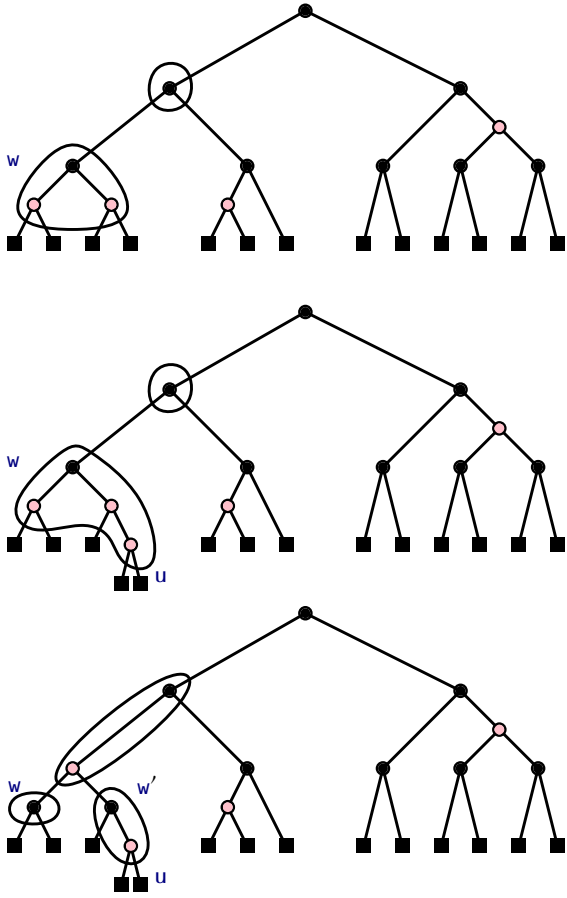


図 9.6: 2-4 木における追加の際の分割を赤黒木で模倣する（これは図 9.2 に示した 2-4 木への要素の追加を模倣している）

黒である。

例えば、図 9.4 は左傾性を満たしていない。右に向かう経路の赤ノードの親がこの性質を満たしていないからだ。

左傾性の維持に意味があるのは、これにより `add(x)` と `remove(x)` において木を更新するときの場合分けが単純になるからである。なぜ単純になるかというと、対応する 2-4 木の表現が、次のように一意に定まるからである。すな

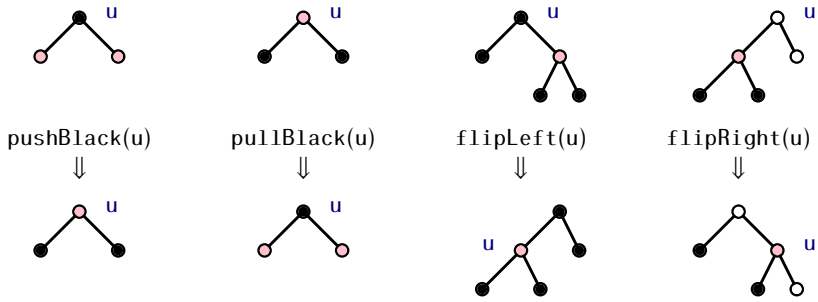


図 9.7: push、pull、flip

わち、2-4 木における次数が 2 のノードは赤黒木における黒ノードであり、黒い子を 2 つ持つ。 次数が 3 のノードは黒ノードであり、左の子が赤ノード、右の子が黒ノードである。次数が 4 のノードは黒ノードであり、赤い子を 2 つ持つ。

add(x) と remove(x) の実装の詳細に入る前に、図 9.7 に示す単純なサブルーチンを導入する。最初の 2 つのサブルーチンは、「黒の高さの性質」を保ったまま色を操作するものである。pushBlack(u) の入力 u は、赤い子を 2 つ持つ黒ノードで、u を赤ノードに、その 2 つの子をいずれも黒ノードに塗り替える。pullBlack(u) は、その逆の操作である。

RedBlackTree
```
void pushBlack(Node *u) {
  u->colour--;
  u->left->colour++;
  u->right->colour++;
}
void pullBlack(Node *u) {
  u->colour++;
  u->left->colour--;
  u->right->colour--;
}
```

flipLeft(u) は、u と u.right の色を入れ替え、そのあとで u を左回転する。この操作は、これら 2 つのノードの色と親子関係をいずれも入れ替える

のである。

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

flipLeft(u) は、u が左傾性を満たしていないとき、すなわち u.left が黒ノード、u.right が赤ノードである場合に、左傾性を取り戻すのに役立つ。この場合は特に、この操作によって、「黒の高さの性質」と「赤の辺の性質」がいずれも満たされることが保証される。flipRight(u) は、flipLeft(u) を左右対称に入れ替えた操作である。

RedBlackTree
```
void flipRight(Node *u) {
  swapcolours(u, u->left);
  rotateRight(u);
}
```

9.2.3 要素の追加

RedBlackTree において add(x) を実装するには、BinarySearchTree における通常の挿入操作によって、u.x = x かつ u.colour = red を満たす新たな葉 u を追加すればよい。この処理では、どのノードでも黒の高さは変わらないので、「黒の高さの性質」を満たさなくなることはない。しかし、左傾性が満たされなくなる可能性がある（u が右の子である場合）。また、「赤の辺の性質」が満たされなくなることもある（u の親が赤ノードの場合）。そこで、これらの性質を満たすようにするため、addFixup(u) を呼び出す。

RedBlackTree
```
bool add(T x) {
  Node *u = new Node();
  u->left = u->right = u->parent = nil;
  u->x = x;
  u->colour = red;
```

```
  bool added = BinarySearchTree<Node,T>::add(u);
  if (added)
    addFixup(u);
  else
    delete u;
  return added;
}
```

図 9.8 に図示したように、addFixup(u) は赤ノード u を入力に取るが、これにより「赤の辺の性質」や左傾性を満たさなくなるかもしれない。この先の議論を理解するには、図 9.8 をよく観察したり、自分で紙に描いてみたりしないと難しいだろう。続きを読む前に、図 9.8 に目を通して意味を考えてみてほしい。

u が木の根なら、u を黒ノードにすれば 2 つの性質が成り立つ。u の兄弟も赤ノードなら、u の親は黒ノードなので、2 つの性質はすでに成り立っている。

このいずれでもないとき、まずは u の親 w が左傾性を満たしているか確認し、もし満たしていないなら flipLeft(w) を実行して u = w とする。すると、u は親である w の左の子になるので、w が左傾性を満たすようになる。あとは、u について「赤の辺の性質」が満たされることを示せばよい。w が黒ノードなら、すでに「赤の辺の性質」は満たされているので、w が赤ノードである場合だけを心配すれば十分である。

まだ処理は終わりではない。いま、u と w はいずれも赤ノードである。「赤い辺の性質」(u では満たされていないが、w では満たされている) より、u には親の親 g が存在し、それは黒ノードである。g の右の子が赤ノードなら、左傾性より、g の子は共に赤ノードである。pushBlack(g) を呼ぶと、g は赤ノードに、w は黒ノードになる。これにより、u について「赤の辺の性質」が満たされるが、g については「赤の辺の性質」が満たされなくなっているかもしれないので、u = g として同じ処理を再度繰り返す。

もし g の右の子が黒ノードなら、flipRight(g) を呼べば w は g の (黒い) 親になり、w は u と g という 2 つの赤い子を持つ。これは、u が「赤い辺の性質」を満たすこと、g が左傾性を満たすことを保証する。この場合、処理はここで終了してよい。

RedBlackTree

```
void addFixup(Node *u) {
  while (u->colour == red) {
```

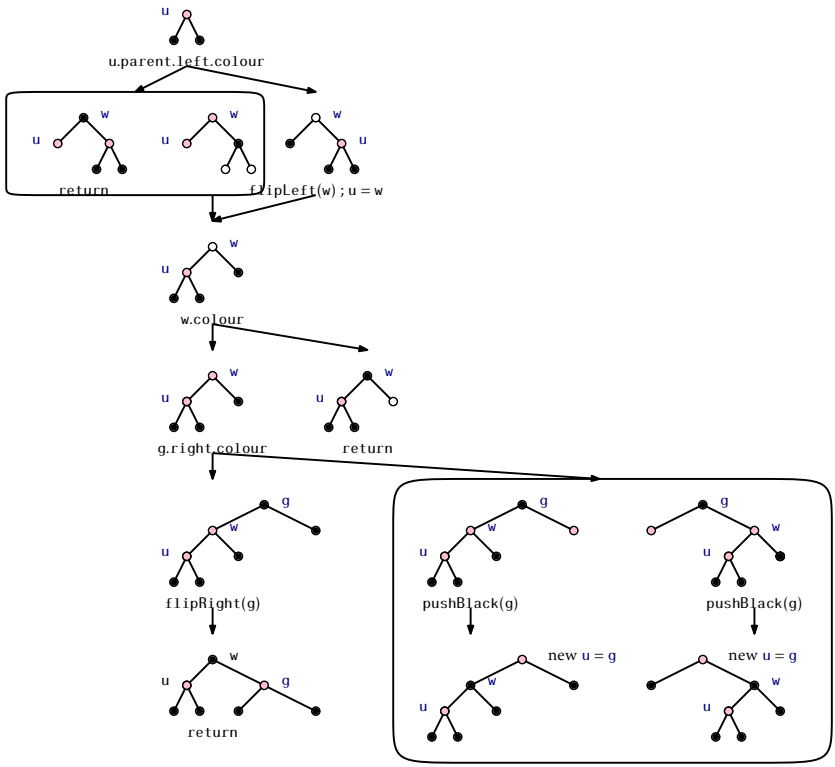


図 9.8: 要素を挿入したあと、2 つの性質を満たすようにするための処理

```
if (u == r) { // u は根なので完了
  u->colour = black;
  return;
}
Node *w = u->parent;
if (w->left->colour == black) { // 左傾性を保つ
  flipLeft(w);
  u = w;
  w = u->parent;
}
```

```
    if (w->colour == black)
      return; // 赤い辺がないので完了
    Node *g = w->parent; // u の祖父母
    if (g->right->colour == black) {
      flipRight(g);
      return;
    } else {
      pushBlack(g);
      u = g;
    }
  }
}
```

insertFixup(u) は、繰り返しごとにかかる実行時間は定数で、繰り返しのたびに u を根のほうへ移動するか、もしくは処理が終了する。よって、insertFixup(u) は $O(\log n)$ 回の繰り返しのあとに終了し、このときの実行時間は $O(\log n)$ である。

9.2.4 要素の削除

RedBlackTree の実装で最も複雑なのは remove(x) である。これは既知の赤黒木のすべてについていえる。突き詰めて考えれば、**二分探索木**における remove(x) のように、u だけを子に持つノード w を特定し、u を u.parent と接続して、w を木から取り除くことになる。

このとき、w が黒ノードだと、「黒の高さの性質」が w.parent で成り立たなくなってしまう。w.colour を u.colour に継ぎ足せば、この問題は一時的に解決する。もちろん、そうすると、今度は次の 2 つの問題が発生する可能性がある。(1) u と w が共に黒ノードのときは、u.colour + w.colour = 2 であり、不正な色（ダブルブラックと呼ぶことにする）になってしまう。(2) w が赤ノードのときは、u と入れ替えることで、u.parent の左傾性が崩れるかもしれない。これらの問題は、いずれも removeFixup(u) を呼ぶことで解決できる。

RedBlackTree

```
bool remove(T x) {
```

```
  Node *u = findLast(x);
  if (u == nil || compare(u->x, x) != 0)
    return false;
  Node *w = u->right;
  if (w == nil) {
    w = u;
    u = w->left;
  } else {
    while (w->left != nil)
      w = w->left;
    u->x = w->x;
    u = w->right;
  }
  splice(w);
  u->colour += w->colour;
  u->parent = w->parent;
  delete w;
  removeFixup(u);
  return true;
}
```

removeFixup(u) の入力であるノード u の色は、1 もしくは 2（ダブルブラック）である。u の色がダブルブラックなら、removeFixup(u) によって回転と色の塗り替え操作を繰り返すことで、ダブルブラックのノードを木から追い出す。この処理では、更新している部分木の根をノード u が参照するようになるまで、更新を繰り返す。この部分木の根の色は変わっているかもしれない。具体的には、赤から黒に変わっているかもしれない。そのため removeFixup(u) では、最後に u の親が左傾性を満たしているかどうかを確認し、必要があれば修正する。

RedBlackTree

```
void removeFixup(Node *u) {
  while (u->colour > black) {
    if (u == r) {
```

```
      u->colour = black;
    } else if (u->parent->left->colour == red) {
      u = removeFixupCase1(u);
    } else if (u == u->parent->left) {
      u = removeFixupCase2(u);
    } else {
      u = removeFixupCase3(u);
    }
  }
  if (u != r) { // 必要であれば左傾性を満たすようにする
    Node *w = u->parent;
    if (w->right->colour == red && w->left->colour == black) {
      flipLeft(w);
    }
  }
}
```

図 9.9 に removeFixup(u) を図示する。以降の説明も、図 9.9 をよく観察しないと理解が難しいだろう。removeFixup(u) を繰り返すたびに、ダブルブラックのノード u は、次の 4 つの場合分けに従って処理される。

Case 0：u が根である場合。このときは最も簡単で、単に u を黒に塗り直せばよい（これは赤黒木のいずれの性質も崩さない）。

Case 1：u の兄弟 v が赤ノードの場合。このときは、左傾性より、u の兄弟 v はその親 w の左の子である。w で右回転を実行し、次の繰り返しに進む。この操作では、w の親が左傾性を満たさなくなり、u の深さが 1 大きくなることに注意する。しかし、次の繰り返しは w が赤ノードの場合（Case 3）である。あとで説明する Case 3 を実行すればうまく処理できる。

RedBlackTree

```
Node* removeFixupCase1(Node *u) {
  flipRight(u->parent);
  return u;
}
```

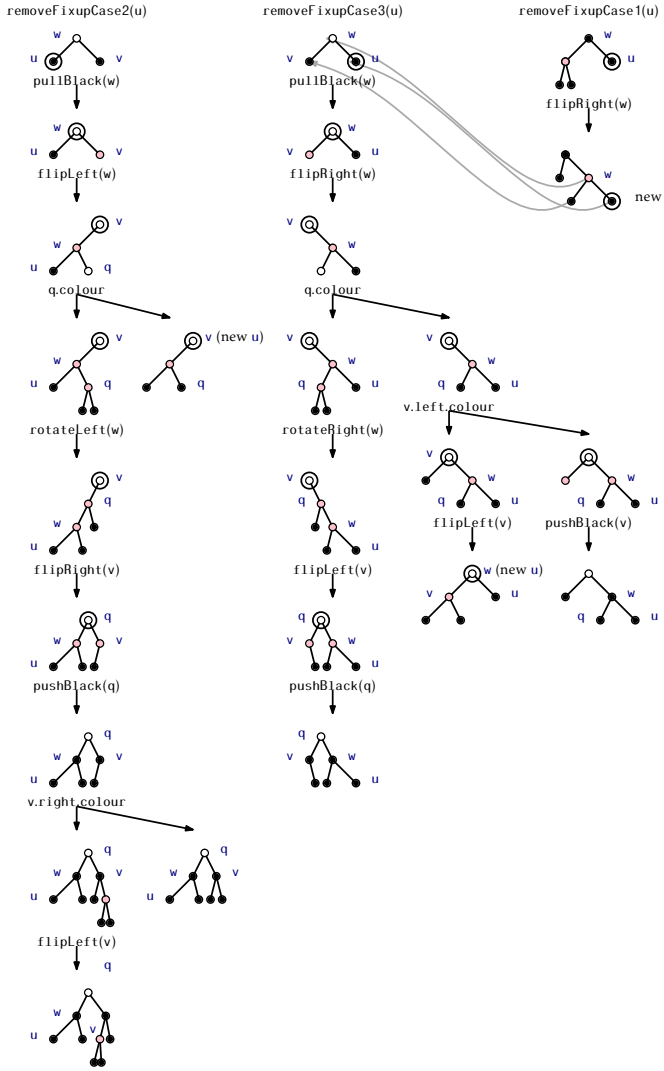


図 9.9: 削除のあと、double-black であるノードを追い出す様子

Case 2：u の兄弟 v が黒ノードの場合。このとき、u は親 w の左の子である。pullBlack(w) を呼ぶと、u は黒ノードに、v は赤ノードになり、w は黒もしくはダブルブラックになる。このとき w は左傾性を満たしておらず、flipLeft(w) を呼んでこれを解決する。

この時点で w は赤ノードであり、v は処理を開始した部分木の根となっている。w が「赤の辺の性質」を満たしているか確認する必要がある。そこで、w の右の子 q を見る。q が黒ノードなら、w は「赤の辺の性質」を満たしているので、u = v として次の繰り返しに進める。

そうでない（q が赤ノード）なら、q と w によって「赤の辺の性質」と左傾性が満たされていない。左傾性を成り立たせるには、rotateLeft(w) を呼べばよい。この時点で、「赤の辺の性質」は依然として崩れている。q は v の左の子で、w は q の左の子である。また、q と w はいずれも赤ノードであり、v は黒またはダブルブラックである。flipRight(v) を実行すれば、q が v と w の両方の親になる。続いて pushBlack(q) を呼ぶと、v と w はいずれも黒ノードになり、q の色は w のもともとの色になる。

これでダブルブラックのノードはなくなり、「赤の辺の性質」と「黒の高さの性質」がいずれも満たされる。あと気にする必要があるのは、v の右の子は赤ノードかもしれないという点であり、その場合は左傾性が満たされなくなっている。そのためこれを確認し、もし必要なら flipLeft(v) を実行する。

RedBlackTree

```
Node* removeFixupCase2(Node *u) {
  Node *w = u->parent;
  Node *v = w->right;
  pullBlack(w); // w->left
  flipLeft(w); // 今は w は赤
  Node *q = w->right;
  if (q->colour == red) { // q と w はいずれも赤
    rotateLeft(w);
    flipRight(v);
    pushBlack(q);
    if (v->right->colour == red)
      flipLeft(v);
    return q;
  } else {
```

```
    return v;
  }
}
```

Case 3：u の兄弟が黒ノードで、u が右の子である場合。この場合は Case 2 と対称であり、ほぼ同様に処理できる。ただし、左傾性が非対称であることに起因する相違がある。そのため少し違った処理をする必要もある。

前と同様に、まずは pullBlack(w) によって v を赤ノードに、u を黒ノードにする。flipRight(w) を呼ぶと、v が部分木の根になる。この時点で w は赤ノードである。そこで、w の左の子の色と q の色に応じて処理が分岐する。

q が赤ノードのときは、Case 2 の場合とまったく同様に処理を終えられる。ただし、v が左傾性を満たさなくなる心配がないので、さらに単純な処理で済ませられる。

複雑なのは q が黒ノードである場合の処理である。この場合、v の左の子の色を確認する。もし赤ノードなら、v の 2 つの子は赤ノードであり、pushBlack(v) を実行して余分な黒を下に送ることができる。この時点で v は w のもともとの色になっているので処理を終えられる。

v の左の子が黒いなら、v は左傾性を満たさなくなっているので、flipLeft(v) を呼んでこれを解消する。そして、次の removeFixup(u) の繰り返しを u = v として続けるために、ノード v を返す。

RedBlackTree

```
Node* removeFixupCase3(Node *u) {
  Node *w = u->parent;
  Node *v = w->left;
  pullBlack(w);
  flipRight(w);              // w は赤
  Node *q = w->left;
  if (q->colour == red) { // q と w はいずれも赤
    rotateRight(w);
    flipLeft(v);
    pushBlack(q);
    return q;
  } else {
    if (v->left->colour == red) {
```

```
      pushBlack(v); // v の子はみな赤
      return v;
    } else { // 左傾性を保つ
      flipLeft(v);
      return w;
    }
  }
}
```

removeFixup(u) の各繰り返しは定数時間で実行できる。Case 2 と Case 3 では、処理が終了するか、もしくは u を根に近づける。Case 0 では、u が根であり、処理は常に終了する。Case 1 は、すぐに Case 3 を引き起こし、この場合も処理は終了する。木の高さは高々 $2\log n$ なので、高々 $O(\log n)$ 回の removeFixup(u) を繰り返せばよいことがわかる。したがって removeFixup(u) の実行時間は $O(\log n)$ である。

9.3 要約

次の定理に RedBlackTree の性能をまとめる。

定理 **9.1.** *RedBlackTree* は *SSet* インターフェースの実装である。*RedBlackTree* には操作 add(x)、remove(x)、find(x) があり、いずれの最悪実行時間も $O(\log n)$ である。

加えて次の定理も成り立つ。

定理 **9.2.** 空の *RedBlackTree* に対して、m 個の add(x) および remove(x) からなる操作の列を実行するとき、addFixup(u) および removeFixup(u) に使われる時間の合計は $O(m)$ である。

定理 9.2 の証明は概要を示すだけとする。addFixup(u) および removeFixup(u) と、2-4 木における葉の追加および削除とを比べると、この性質は 2-4 木に由来するものであることが推察できるだろう。特に、2-4 木における分割、併合、子を借りる処理に必要な時間が $O(m)$ であることを示せれば、これから定理 9.2 を導けるはずだ。

2-4 木に関するこの定理の証明は、ポテンシャル法を使った償却解析による[*3]。2-4 木の内部ノード u のポテンシャルを次のように定義する。

$$\Phi(u) = \begin{cases} 1 & u \text{ の子の数が } 2 \text{ のとき} \\ 0 & u \text{ の子の数が } 3 \text{ のとき} \\ 3 & u \text{ の子の数が } 4 \text{ のとき} \end{cases}$$

また、2-4 木のポテンシャルを、そのすべてのノードのポテンシャルの和と定義する。分割の際には、4 つの子を持つノードが、それぞれ 2 つの子と 3 つの子を持つ 2 つのノードになる。すなわち、全体のポテンシャルは $3 - 1 - 0 = 2$ だけ小さくなる。併合の際には、それぞれ 2 つの子を持っていた 2 つのノードが、3 つの子を持つ 1 つのノードになる。このとき、全体のポテンシャルは $2 - 0 = 2$ だけ小さくなる。よって、分割や併合の際には、ポテンシャルは 2 だけ小さくなる。

分割と併合以外の処理については、葉を加えたり削除したりしたときに子の数が変わるノードの個数は定数である。ノードを追加するときは、1 つのノードの子が 1 だけ増え、ポテンシャルは高々 3 増える。ノードを削除するときは、1 つのノードの子が 1 だけ減り、ポテンシャルは高々 1 増える。また、子を借りる処理には 2 つのノードが関連し、それらのポテンシャルは高々 1 だけ増える。

まとめると、分割と併合はポテンシャルを 2 以上減らす。分割と併合以外では、追加と削除はポテンシャルを高々 3 増やし、ポテンシャルは常に非負である。よって、空の木に対して m 回の追加と削除を実行するとき、分割と併合は合わせて高々 $3m/2$ 回だけ実行される。定理 9.2 はこの解析の帰結であり、2-4 木と赤黒木の間の対応を示している。

9.4 ディスカッションと練習問題

赤黒木は Guibas と Sedgewick によって最初に提案された [38]。赤黒木は実装が非常に複雑であるにもかかわらず、ライブラリやアプリで最も頻繁に使われるデータ構造のうちの 1 つである。アルゴリズムやデータ構造の本の多くでは、何種類かの赤黒木が説明されている。

Andersson [6] では、赤黒木に似た、左傾なバランスされた木が提案されている。この木では、任意のノードが最大で 1 つの赤い子を持つ、という制約が

[*3] ポテンシャル法の他の例としては、補題 2.2 や補題 3.1 の証明を参照せよ。

加えられている。そのため、このデータ構造がシミュレートするのは、2-4 木ではなく 2-3 木である。このデータ構造は、本章で説明した RedBlackTree よりもかなり単純である。

Sedgewick [64] では、二種類の左傾な赤黒木が提案されている。それらのデータ構造では、2-4 木における上から下方向への分割および併合のシミュレーションに加えて、再帰が利用されている。これによりプログラムが短くエレガントになっている。

赤黒木に関連したより古いデータ構造として、AVL **木** [3] がある。AVL 木は、height-balanced **性**と呼ばれる性質を満たす木である。height-balanced 性とは、「任意のノード u について、`u.left` を根とする部分木の高さと、`u.right` を根とする部分木の高さとの差は、高々 1 である」という性質だ。この性質より、$F(h)$ を高さ h である木のうちで葉が最も少ないものの葉の数とするとき、$F(h)$ が次のフィボナッチの漸化式を満たすことが従う。

$$F(h) = F(h-1) + F(h-2)$$

ただし $F(0) = 1$, $F(1) = 1$ である。ここで**黄金比**を $\varphi = (1+\sqrt{5})/2 \approx 1.61803399$ とするとき、$F(h)$ は近似的に $\varphi^h/\sqrt{5}$ である（より正確には $|\varphi^h/\sqrt{5} - F(h)| \le 1/2$ である）。補題 9.1 の証明で述べたように、これは次の式を含意する。

$$h \le \log_\varphi n \approx 1.440420088 \log n$$

よって、AVL 木の高さは赤黒木の高さよりも低い。add(x) および remove(x) の際は、根に向かって戻る際に通過する各ノード u について、u の左右の部分木の高さが 2 以上異なる場合にはバランスを再調整して高さのバランスを保つ（図 9.10）。

Andersson のデータ構造、Sedgewick のデータ構造、AVL 木は、いずれも RedBlackTree よりも実装が単純である。しかし、いずれも、バランスを再調整するための償却実行時間が $O(1)$ であることを保証できない。特に、定理 9.2 のような保証はない。

問 **9.1.** 図 9.11 の RedBlackTree に対応する 2-4 木を図示せよ。

問 **9.2.** 図 9.11 の RedBlackTree に 13、3.5、3.3 を順に追加する様子を図示せよ。

問 **9.3.** 図 9.11 の RedBlackTree から 11、9、5 を順に削除する様子を図示せよ。

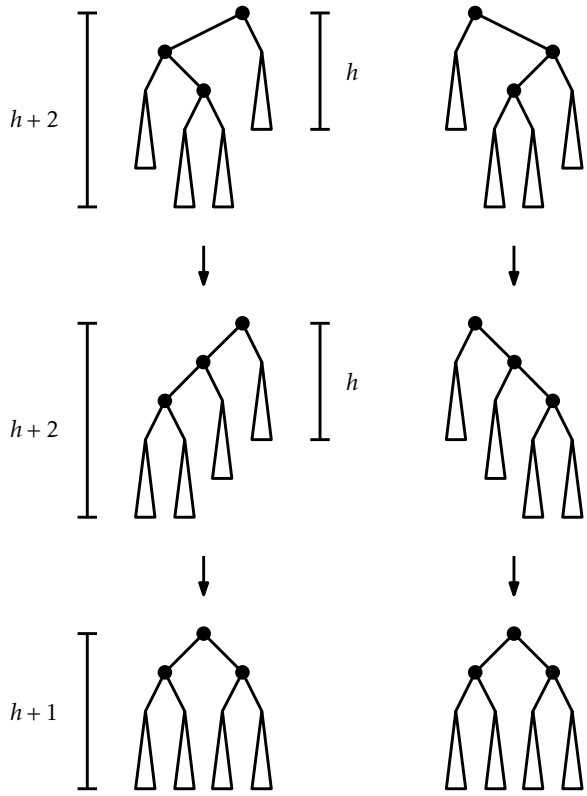


図 9.10: AVL 木におけるバランスの再調整の様子。その子の部分木の大きさが h と $h+2$ であるノードについて、いずれの子の部分木の高さも $h+1$ とする際に必要な回転は高々 2 回である

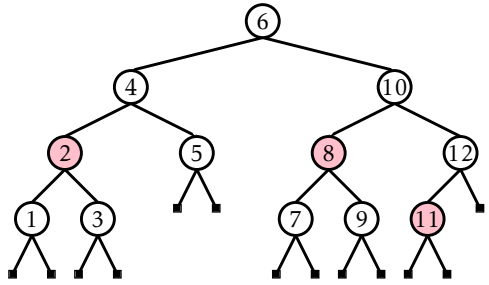


図 9.11: 練習問題のための赤黒木

問 **9.4.** どれだけ大きな n についても、n 個のノードを持ち、高さが $2\log n - O(1)$ である赤黒木が存在することを示せ。

問 **9.5.** 操作 pushBlack(u) および pullBlack(u) を考える。これらは、赤黒木によって表現されている 2-4 木に対し、どんなことをする操作か。

問 **9.6.** どれだけ大きな n についても、n 個のノードを持ち、高さが $2\log n - O(1)$ である赤黒木を構成する add(x) および remove(x) からなる操作の列が存在することを示せ。

問 **9.7.** RedBlackTree の実装における remove(x) で、u.parent = w.parent とするのはなぜか説明せよ。この処理は splice(w) においてすでに実行済みではないだろうか。

問 **9.8.** 2-4 木 T は、n_ℓ 個の葉と n_i 個の内部ノードを持つとする。

1. n_ℓ が与えられたとき、n_i の最小値を求めよ。
2. n_ℓ が与えられたとき、n_i の最大値を求めよ。
3. T を表現する赤黒木を T' とすると、T' が持つ赤ノードの個数を求めよ。

問 **9.9.** n 個のノードを持ち、高さが $2\log n - 2$ 以下の二分探索木があるとする。このとき、この木のすべてのノードを、「黒の高さの性質」と「赤の辺の性質」をいずれも満たすように、赤または黒に塗ることはできるか。もし可能なら、それに加えて左傾性を満たすことはできるか。

問 **9.10.** 赤黒木 T_1 および T_2 は、黒の高さがいずれも h であり、T_1 の最大のキーは T_2 の最小のキーよりも小さいとする。このとき、$O(h)$ の時間で T_1 と T_2 を 1 つの赤黒木に併合する方法を示せ。

問 **9.11.** 問 9.10 の解法を拡張し、T_1 および T_2 の黒の高さ h_1 および h_2 が異なるとき、すなわち $h_1 \neq h_2$ であるときにも適用可能にせよ。ただし、実行時間は $O(\max\{h_1, h_2\})$ とする。

問 **9.12.** AVL 木における add(x) の間に、最大で一度だけバランスの再調整操作を実行しなければならないことを証明せよ（このとき、回転を最大で 2 回実行することになる。図 9.10 参照）。また、remove(x) 操作の際に必要となるバランス再調整のオーダーが $\log n$ であるような AVL 木の例を挙げよ。

問 **9.13.** 上で説明した AVL 木の実装として、AVLTree クラスを作成せよ。作

成した実装の性能と RedBlackTree の性能とを比較せよ。find(x) が高速なのはどちらの実装か。

問 **9.14.** SSet の実装である SkiplistSSet、ScapegoatTree、Treap、RedBlackTree における find(x)、add(x)、remove(x) の相対的な性能を評価する実験を設計、実装せよ。ランダムなデータや整列済みのデータ、ランダムな順序や規則正しい順序での削除などを含む、さまざまなテストシナリオを作ること。

第 10

ヒープ

優先度付きキューは利用価値がとても高い。この章では、優先度付きキューの実装を 2 つ説明する。いずれの実装も特殊な二分木であり、**ヒープ**（heap）と呼ばれている。ヒープには「雑多に積まれたもの」という意味がある。これは「高度に構造化されて積み上げられたもの」であった二分探索木とは対照的である。

1 つめのヒープの実装では、完全二分木をシミュレートするのに配列を使う。この実装は極めて高速であり、ヒープソート（11.1.3 節参照）という整列アルゴリズムを実装できる。ヒープソートは既知の整列アルゴリズムの中で最速なもののひとつである。2 つめに紹介する実装では、完全二分木に限らず、より柔軟な二分木を扱う。この実装では、優先度付きキューの要素すべてを別の優先度付きキューに取り込む操作を利用する。

10.1 BinaryHeap：二分木を間接的に表現する

1 つめに紹介する優先度付きキューの実装は、400 年以上前に発見された Eytzinger **法**という手法に基づく。この手法により、木のノードを幅優先順（6.1.2 節）に配列に入れていくことで、完全二分木を表現できる。具体的には、配列の添字 0 の位置に木の根を、添字 1 の位置に根の左の子を、添字 2 の位置に根の右の子を、添字 3 の位置に根の左の子の左の子を格納していく（図 10.1）。

大きな木に対して Eytzinger 法を適用すると規則性が見えてくる。添字 i のノードの左の子は添字 $\mathtt{left}(i) = 2i + 1$ の位置に入り、右の子は添字 $\mathtt{right}(i) = 2i + 2$ の位置に入る。添字 i のノードの親は、添字 $\mathtt{parent}(i) =$

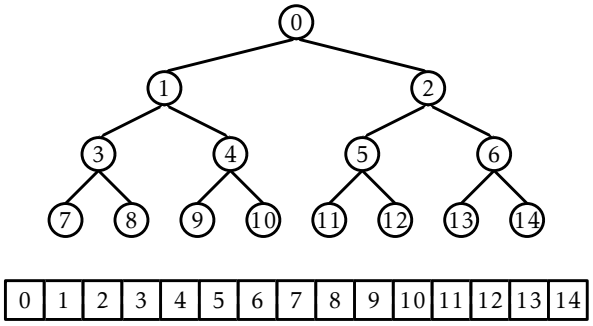


図 10.1: Eytzinger 法により、配列を使って完全二分木を表現する

$(i-1)/2$ の位置にある。

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;
}
```

BinaryHeap では、この手法を使うことで、要素が**ヒープ順**（heap order）に並んだ間接的な形で完全二分木を表す。ヒープ順とは、添字 i（$i = 0$）の位置に格納されている値が、添字 parent(i) に格納されている値以上であるような順番である。したがって BinaryHeap では、優先度付きキューにおける最小値が添字 0 の位置、つまり根に格納されていることになる。

BinaryHeap では n 個の要素を配列 a に格納する。

BinaryHeap

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

add(x) の実装は簡単である。他の配列ベースのデータ構造と同じく、まずは a が一杯かどうかを確認する（a.length = n かどうかを確認する）。もしそうなら a を拡張する。続いて x を a[n] に入れ、n を 1 増やす。あとはヒープ性（要素がヒープ順に並んでいること）を保てばよい。これは、x とその親とを交換する操作を、x が親以上になるまで繰り返せばよい（図 10.2）。

BinaryHeap

```
bool add(T x) {
  if (n + 1 > a.length) resize();
  a[n++] = x;
  bubbleUp(n-1);
  return true;
}
void bubbleUp(int i) {
  int p = parent(i);
  while (i > 0 && compare(a[i], a[p]) < 0) {
    a.swap(i,p);
    i = p;
    p = parent(i);
  }
}
```

remove() は、ヒープから最小の値を削除する。この操作の実装には少し工夫が必要になる。根が最小値であることはわかっているのだが、これを削除したあともヒープ性が成り立つことを保証しなければならない。

最も簡単な方法は、根と a[n − 1] を交換し、交換後に a[n − 1] にある値を削除して、n を 1 小さくすることだ。しかし、結果として新しく根となった値は、おそらく最小値ではない。そこで、これを下方向に動かしていく必要がある。新しく根となった要素を 2 つの子と比較し、新しく根となった要素の値が 3 つのうちで最小ならば処理を終了する。そうでないなら、2 つの子のうち小さいものと入れ替え、同様の処理を繰り返す。

BinaryHeap

```
T remove() {
  T x = a[0];
  a[0] = a[--n];
```

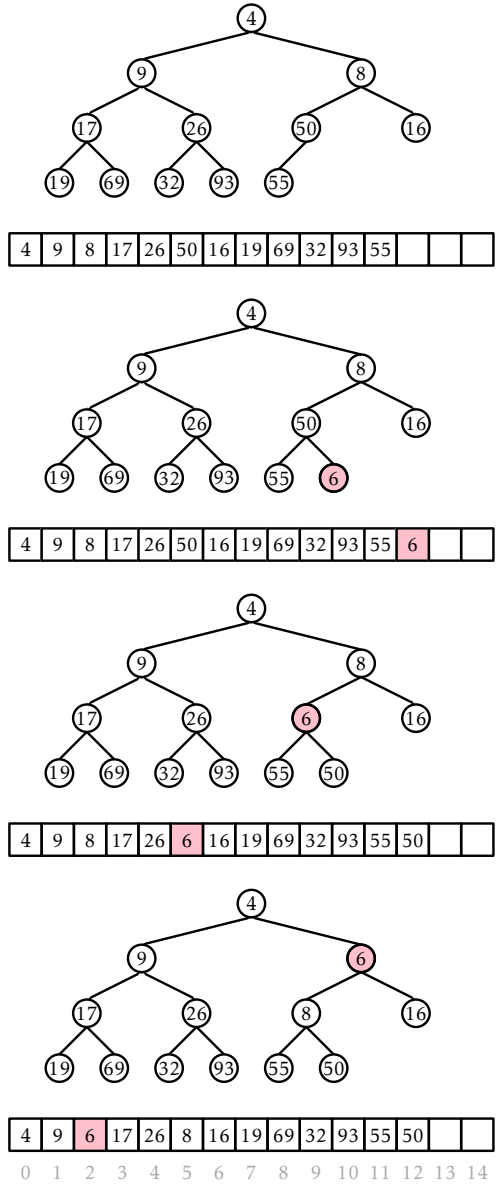


図 10.2: BinaryHeap に値 6 を追加する

```
  trickleDown(0);
  if (3*n < a.length) resize();
  return x;
}
void trickleDown(int i) {
  do {
    int j = -1;
    int r = right(i);
    if (r < n && compare(a[r], a[i]) < 0) {
      int l = left(i);
      if (compare(a[l], a[r]) < 0) {
        j = l;
      } else {
        j = r;
      }
    } else {
      int l = left(i);
      if (l < n && compare(a[l], a[i]) < 0) {
        j = l;
      }
    }
    if (j >= 0)  a.swap(i, j);
    i = j;
  } while (i >= 0);
}
```

他の配列ベースのデータ構造と同様に、resize() のための時間は無視することにする。resize() の時間は補題 2.1 の償却解析において考慮されるからだ。すると、add(x) と remove() の実行時間は、配列によって間接的に表されている二分木の高さに依存する。この二分木は、都合がいいことに、**完全**二分木である。どの深さにも、最下層を除いて、その層で取りうる個数のノードが含まれている。よって、h を木の高さとすると、この木には少なくとも 2^h 個

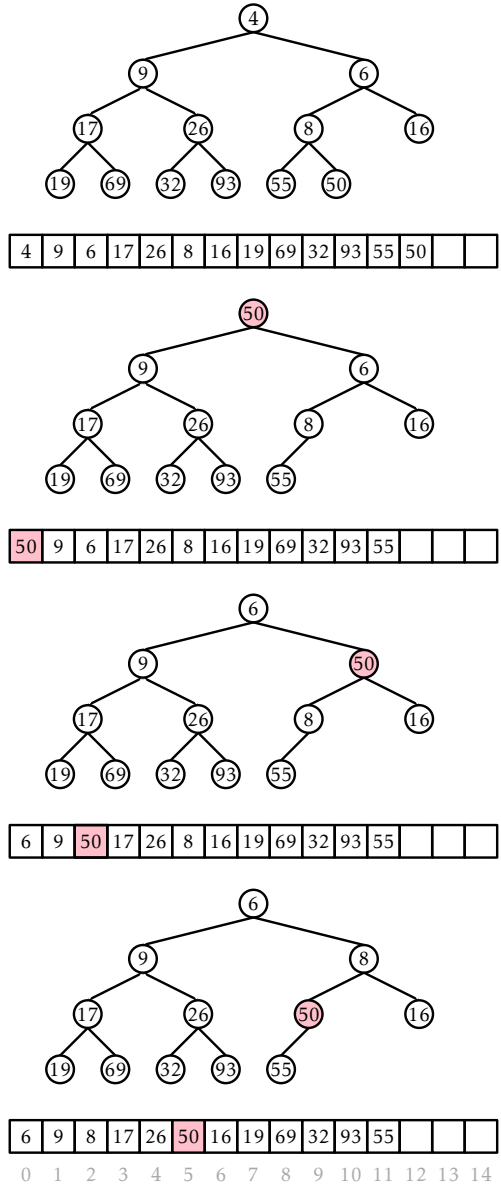


図 10.3: BinaryHeap から最小の値 4 を削除する

のノードがある。言い換えると、次の式が成り立つ。

$$n \geq 2^h$$

両辺の対数を取ると、次の式が成り立つ。

$$h \leq \log n$$

以上より、add(x) と remove() の実行時間はいずれも $O(\log n)$ である。

10.1.1 要約

次の定理に BinaryHeap の性能をまとめる。

定理 **10.1.** *BinaryHeap* は、優先度付きキューのインターフェースを実装する。*BinaryHeap* は add(x) と remove() をサポートする。resize() のコストを無視すると、いずれの操作の実行時間も $O(\log n)$ である。

空の *BinaryHeap* から、m 個の add(x) 操作および remove() 操作からなる任意の列を実行する。このとき、すべての resize() にかかる時間の合計は $O(m)$ である。

10.2 MeldableHeap：つなぎ合わせられるランダムなヒープ

この節では MeldableHeap を紹介する。MeldableHeap も BinaryHeap と同じく優先度付きキューの実装であり、背後にある構造もヒープである。しかし、BinaryHeap と違って、要素数から二分木の形が一意には決まらない。MeldableHeap では、背後にある二分木の形状に制約がない。

MeldableHeap における add(x) および remove() は、merge(h1,h2) という操作を使って実装される。この操作は、ヒープのノード h1 と h2 を引数とし、h1 を根とする部分木と h2 を根とする部分木のすべてのノードを含むヒープの根を返す。

merge(h1,h2) は再帰的に定義できる（図 10.4）。h1 または h2 が nil なら、単に h2 と h1 をそれぞれ返せばよい。そうでない場合について考える。一般性を失うことなく、h1.x ≤ h2.x の場合について考えればよい。この場合、新たなヒープの根の値は h1.x である。続いて、h2 を h1.left か h1.right のどちらかと再帰的に併合する。ここでランダム性の出番だ。コインを投げ、h2 を h1.left と h1.right のどちらと併合するかを決める。

MeldableHeap
```
Node* merge(Node *h1, Node *h2) {
  if (h1 == nil) return h2;
  if (h2 == nil) return h1;
  if (compare(h1->x, h2->x) > 0) return merge(h2, h1);
    // この時点で h1->x <= h2->x だとわかる
  if (rand() % 2) {
    h1->left = merge(h1->left, h2);
    if (h1->left != nil) h1->left->parent = h1;
  } else {
    h1->right = merge(h1->right, h2);
    if (h1->right != nil) h1->right->parent = h1;
  }
  return h1;
}
```

merge(h1,h2) の実行時間については、期待値が $O(\log n)$ であることを次節で示す。ここで、n は h1 と h2 の要素数の合計である。

merge(h1,h2) を使えば、add(x) は簡単に実装できる。x を含む新たなノード u を作り、u をヒープの根と併合すればよい。

MeldableHeap
```
bool add(T x) {
  Node *u = new Node();
  u->left = u->right = u->parent = nil;
  u->x = x;
  r = merge(u, r);
  r->parent = nil;
  n++;
  return true;
}
```

このとき、実行時間の期待値は $O(\log(n+1)) = O(\log n)$ である。

remove() も同様に簡単に実装できる。削除したいのは根なので、その 2 つ

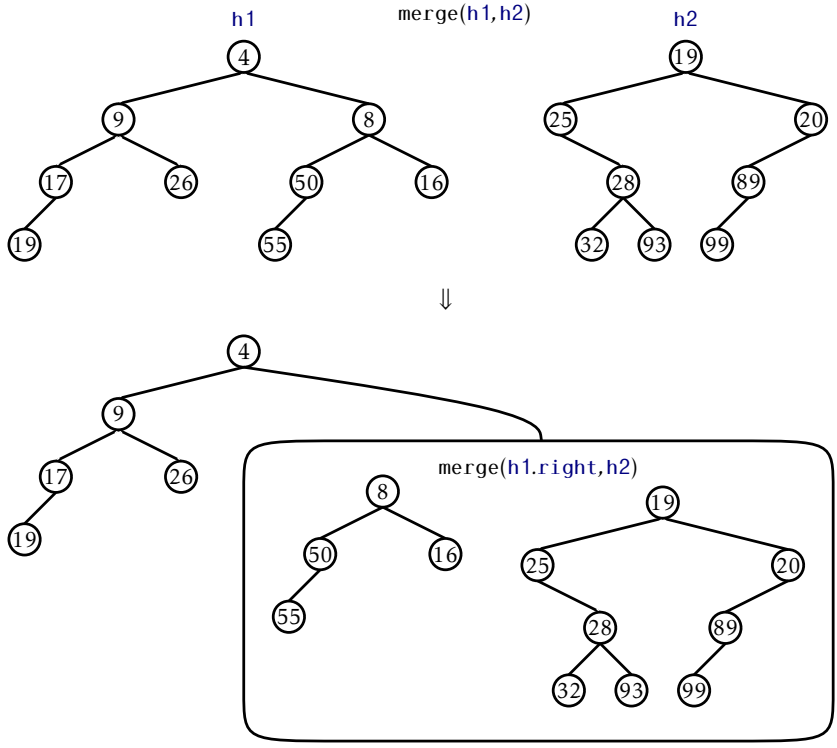


図 10.4: h1 と h2 を併合するために、h1.left または h1.right のいずれかに h2 を併合する

の子を併合し、その結果を新たな根とすればよい。

MeldableHeap

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

```
}
```

このときも実行時間の期待値は $O(\log n)$ である。

実行時間の期待値が $O(\log n)$ である `MeldableHeap` の操作は、このほかにもいくつかある。例えば次のような操作がある。

- `remove(u)`：ノード `u` をヒープから削除する
- `absorb(h)`：ヒープに `h` の要素をすべて追加し、`h` を空にする

いずれの操作も、定数回の `merge(h1,h2)` を使って実装できる。

10.2.1 `merge(h1,h2)` の解析

`merge(h1,h2)` は、二分木におけるランダムウォークを考えることで解析できる。二分木における**ランダムウォーク**は、根から出発する。各ステップでコインを投げ、その結果に応じて、いまいるノードから右の子もしくは左の子に進む。木からはみ出したら、つまり、進んだ先が `nil` になったら、処理を終了する。

次の補題は、二分木の形状にかかわらず成り立つという点で注目に値する。

補題 10.1. n 個のノードからなる二分木におけるランダムウォークの長さの期待値は $\log(n+1)$ 以下である。

証明. n に関する帰納法により証明する。$n = 0$ のとき、ステップ数は $0 = \log(n+1)$ である。以下では、任意の非負整数 $n' < n$ について、示したい補題が成り立つと仮定する。

n_1 を左の部分木の大きさとする。このとき、$n_2 = n - n_1 - 1$ が右の部分木の大きさである。根から出発して一歩進み、大きさ n_1 の部分木または n_2 の部分木の処理へと進む。n_1 と n_2 はいずれも n より小さいので、帰納法の仮定より、ステップ数の期待値は次のようになる。

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

log は上に凸な関数なので、$\mathrm{E}[W]$ は $n_1 = n_2 = (n-1)/2$ で最大値を取る。よって、ランダムウォークのステップ数の期待値は次のようになる。

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

$$= 1 + \log((n+1)/2)$$
$$= \log(n+1) \qquad \square$$

なお、情報理論では、補題 10.1 をエントロピーの用語を使って証明できる。

補題 *10.1* の情報理論的な証明. d_i を、添字 i に対応する外部ノード（nil）の深さとする。n 個のノードを持つ二分木が n + 1 個の外部ノードを持つことを思い出そう。ランダムウォークが i 番めの外部ノードに辿り着く確率は $p_i = 1/2^{d_i}$ である。よって、ランダムウォークのステップ数の期待値は次のように計算できる。

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

右辺は、n + 1 個の要素にわたって確率分布のエントロピーを求めたものだとわかるだろう。n + 1 個の要素にわたる確率分布のエントロピーに関する基本的な事実として、この値は $\log(n+1)$ を超えない。よって、補題が示された。 □

ランダムウォークに関するこの結果より、merge(h1,h2) の実行時間の期待値は $O(\log n)$ であることを示せる。

補題 **10.2.** h1 および h2 は *2* つのヒープの根であり、それぞれのヒープは n_1 個および n_2 個のノードを含むとする。このとき、merge(h1,h2) の実行時間は $O(\log n)$ 以下である。ただし、$n = n_1 + n_2$ である。

証明. merge の各ステップでは、ランダムウォークを 1 ステップ実行し、h1 か h2 のいずれかを根とする部分木に進む。このアルゴリズムは、2 つのランダムウォークのどちらか一方が木からはみ出して h1 = null または h2 = null になったら終了する。よって、併合アルゴリズムのステップ数の期待値は次の値以下である。

$$\log(n_1 + 1) + \log(n_2 + 1) \le 2 \log n \qquad \square$$

10.2.2 要約

次の定理に MeldableHeap の性能をまとめる。

定理 **10.2.** *MeldableHeap* は、優先度付きキューのインターフェースを実装する。*MeldableHeap* は、add(x) と remove() をサポートする。いずれの操作も、実行時間の期待値は $O(\log n)$ である。

10.3 ディスカッションと練習問題

完全二分木を配列またはリストとして間接的に表現する方法を最初に提案したのは Eytzinger[27] であると考えられてきた。彼は、貴族の家系図が記された書物でこの手法を使った。この章で説明した BinaryHeap は、Williams が最初に提案したものである [76]。

この章で説明した、ランダム性を利用した MeldableHeap は、Gambin と Malinowski が最初に提案したものである [34]。同様なデータ構造はほかにもいくつか知られている。leftist heaps [16, 48, Section 5.3.2]、binomial heaps [73]、Fibonacci heaps [30]、pairing heaps [29]、skew heaps [70] などである。しかし、いずれも MeldableHeap ほどシンプルではない。

これらのデータ構造の中には、decreaseKey(u, y) という操作をサポートするものもある。これはノード u に格納される値を小さくして y とする操作である（$y \le u.x$ であることを前提とする）。これらのデータ構造の大部分では、このような操作のためにノード u を削除して y を追加するので、$O(\log n)$ の時間がかかる。しかし、もっと効率的にこの操作を実装できるデータ構造もある。中でも、Fibonacci heaps は、$O(1)$ の償却実行時間で decreaseKey(u, y) を実行できる。pairing heaps の特殊なものでは、$O(\log \log n)$ の償却実行時間で decreaseKey(u, y) を実行できる [25]。効率的な decreaseKey(u, y) は、ダイクストラ法など、グラフアルゴリズムの高速化に役立つ [30]。

問 **10.1.** 図 10.2 に示した BinaryHeap に 7 と 3 を順に追加する様子を図示せよ。

問 **10.2.** 図 10.3 に示した BinaryHeap から次の 2 つの値（6 と 8）を順に削除する様子を図示せよ。

問 **10.3.** remove(i) を実装せよ。これは BinaryHeap における a[i] の値を削除するメソッドである。このメソッドの実行時間は $O(\log n)$ でなければならない。また、なぜこのメソッドが役に立ちそうにないかを説明せよ。

問 **10.4.** d 分木は二分木の一般化である。これは各内部ノードが d 個の子を持つ木である。Eytzinger の方法を使えば、完全 d 分木も配列を使って表現できる。添字 i が与えられたとき、i の親と、i の d 個の子、それぞれの添字を計算する方法を与えよ。

問 **10.5.** 問 10.4 で学んだことを使って DaryHeap を設計、実装せよ。これは d 分木版の BinaryHeap である。DaryHeap の操作の実行時間を解析し、

DaryHeap の性能を BinaryHeap と比較せよ。

問 **10.6.** 図 10.4 に示した MeldableHeap に 17、82 を順に追加する様子を図示せよ。ランダムなビットが必要なときにはコイン投げを使うこと。

問 **10.7.** 図 10.4 に示した MeldableHeap から、次の 2 つの値（4 と 8）を削除せよ。ランダムなビットが必要なときにはコイン投げを使うこと。

問 **10.8.** MeldableHeap からノード u を削除する remove(u) を実装せよ。ただし、このメソッドの実行時間の期待値は $O(\log n)$ でなければならない。

問 **10.9.** BinaryHeap および MeldableHeap において 2 番めに小さい値を定数時間で見つける方法を示せ。

問 **10.10.** BinaryHeap および MeldableHeap において k 番めに小さい値を $O(k \log k)$ の時間で見つける方法を示せ。（ヒント：別のヒープを使ってみよう。）

問 **10.11.** k 個の整列済みリストであって、合計の長さが n であるものがあるとき、ヒープを使ってこれらのリストを 1 つの整列済みリストにする方法を示せ。このとき、実行時間は $O(n \log k)$ でなければならない。（ヒント：$k = 2$ の場合から考えてみよう。）

第 **11**

整列アルゴリズム

この章では、大きさ n の集合を整列するアルゴリズムを紹介する。データ構造の教科書なのに、整列アルゴリズムの説明が始まるなんて、奇妙に思うかもしれない。これにはいくつか理由がある。例えば、この章で紹介するクイックソートはランダム二分探索木と深い関係があるし、ヒープソートはデータ構造のヒープと深い関係がある。

11.1 節では、比較に基づく整列アルゴリズムを 3 つ紹介する。3 つとも、整列にかかる実行時間が $O(n \log n)$ であるような整列アルゴリズムである。さらに、この実行時間が漸近的に最適であることを示す。つまり、比較に基づく整列アルゴリズムでは、最悪実行時間にせよ平均実行時間にせよ、最低でも約 $n \log n$ 回の比較が必要なのだ。

なお、これまでの章で説明してきた整列済み集合 `SSet` や優先度付き `Queue` であれば、どれを使っても $O(n \log n)$ の時間で整列可能なアルゴリズムを実装できる。例えば、n 個の要素を `BinaryHeap` または `MeldableHeap` に `add(x)` し、続いて `remove()` を n 回繰り返せば、順番に要素を取り出せる。あるいは、何らかの二分探索木に対して `add(x)` を n 回実行し、そのあと行きがけ順（問 6.8）で木を巡回すれば、整列された順で要素が見つかる。ただし、どちらの例でも、わざわざ作ったデータ構造の限られた機能しか使っておらず、かなり無駄なことをしている。整列は、可能な限り速く、単純で、省メモリな手法をあえて開発するだけの価値がある、極めて重要な問題なのである。

章の後半では、比較のほかに使える操作があれば、さらに優れた実行時間で整列が可能なことを見る。実際、配列のランダムアクセスを使うことで、n 個の要素 $\{0,\ldots,n^c-1\}$ を $O(cn)$ の時間で整列できることを説明する。

11.1 比較に基づく整列

この節では、マージソート、クイックソート、ヒープソートという 3 つの整列アルゴリズムを紹介する。3 つとも、配列 a を入力すると、$O(n \log n)$ の（期待）実行時間で a の要素を昇順に整列する。いずれも**比較に基づく**アルゴリズムである。 整列するデータの型は何でもよい。ただし、データの比較を実行するメソッドがなければならない。1.2.4 節と同様に、以降では compare(a,b) というメソッドがあるとする。compare(a,b) は、a < b なら負の値を、a > b なら正の値を、a = b ならゼロを返すものとする。

11.1.1 マージソート

マージソート（merge sort）は、再帰的な分割統治法の例として古典的なアルゴリズムである。配列 a の長さが 1 以下なら、a はすでに整列されており、何もする必要はない。そうでなければ、配列 a を半分ずつ、すなわち、配列 a0 = a[0],...,a[n/2 − 1] および配列 a1 = a[n/2],...,a[n − 1] に分ける。a0 と a1 をそれぞれ再帰的に整列し、整列済みの a0 と a1 とを併合することで、整列済みの配列 a を得る。

Algorithms

```
void mergeSort(array<T> &a) {
  if (a.length <= 1) return;
  array<T> a0(0);
  array<T>::copyOfRange(a0, a, 0, a.length/2);
  array<T> a1(0);
  array<T>::copyOfRange(a1, a, a.length/2, a.length);
  mergeSort(a0);
  mergeSort(a1);
  merge(a0, a1, a);
}
```

図 11.1 にマージソートの例を示す。

a0 と a1 の併合は、整列と比べればかなり簡単である。a に要素を 1 つずつ加えていけばよい。a0 か a1 がどちらかが空になったら、空でないほうの

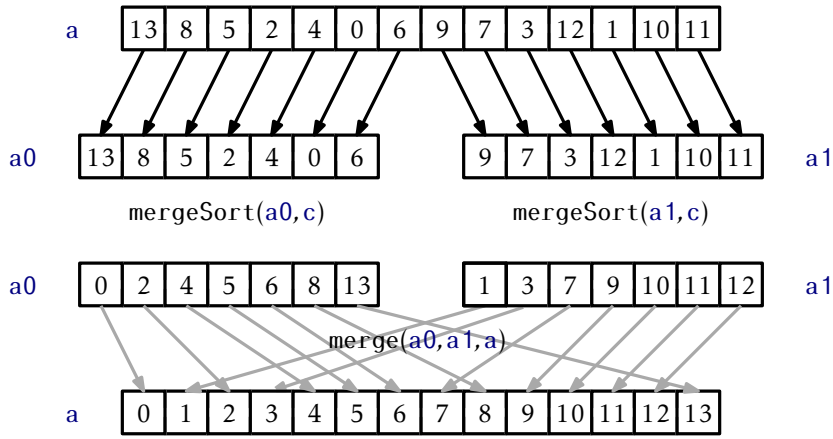


図 11.1: mergeSort(a,c) を実行する様子

配列の残りの要素をすべて a に加える。それまでは、a0 の次の要素と a1 の次の要素のうち、小さいほうを a に加えていく。

Algorithms

```
void merge(array<T> &a0, array<T> &a1, array<T> &a) {
  int i0 = 0, i1 = 0;
  for (int i = 0; i < a.length; i++) {
    if (i0 == a0.length)
      a[i] = a1[i1++];
    else if (i1 == a1.length)
      a[i] = a0[i0++];
    else if (compare(a0[i0], a1[i1]) < 0)
      a[i] = a0[i0++];
    else
      a[i] = a1[i1++];
  }
}
```

a0 と a1 の要素数の合計を n とすると、merge(a0,a1,a,c) では、a0 または

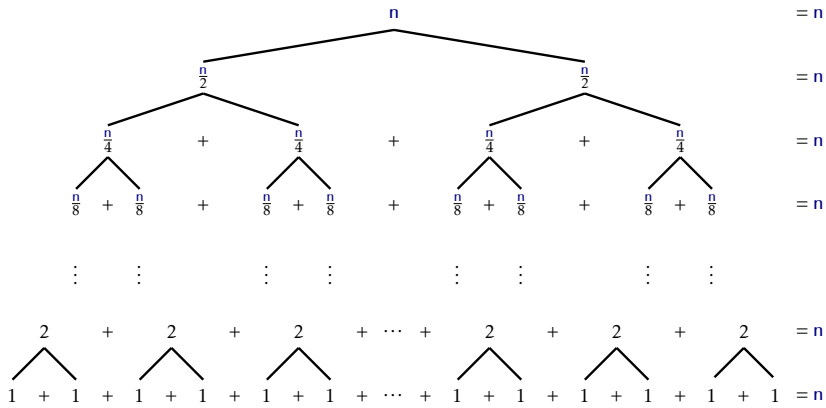


図 11.2: マージソートの再帰木

a1 が空になる前に最大で n − 1 回の比較を行う。

マージソートの実行時間を求めるには、図 11.2 のような再帰的な木を考えるとよい。いま、n が 2 の冪乗であると仮定する。このとき、$n = 2^{\log n}$ であり、$\log n$ は整数となる。マージソートでは、n 個の要素の整列問題を、「n/2 個の要素の並べ替え」という 2 つの問題に分割して考える。分割した 2 つの問題をそれぞれさらに 2 つの問題に分割することで、「n/4 個の要素の並べ替え」が 4 つになる。この 4 つの問題をそれぞれさらに分割することで、「n/8 個の要素の並べ替え」が 8 つになる。これを繰り返して、「2 個の要素の並べ替え」が n/2 個になり、最終的には「1 個の要素の並べ替え」が n 個になる。大きさ $n/2^i$ の問題を解くには、すでに解かれた 2 つの部分問題の解答をマージしながらコピーするのに、$O(n/2^i)$ の時間がかかる。大きさ $n/2^i$ の問題が 2^i 個あるので、大きさ 2^i の問題のために必要な時間の合計は次のようになる（まだ再帰的に数え上げていないことに注意）。

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

よって、マージソートに必要な時間の合計は次のようになる。

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

次の定理は、以上の解析に基づいて証明できる。ただし、n が 2 の冪乗でない場合も扱うので、少しだけ注意が必要になる。

定理 **11.1.** *mergeSort(a)* の実行時間は $O(n \log n)$ であり、最大で $n \log n$ 回の比較を行う。

証明. n についての帰納法により証明する。$n = 1$ の場合は自明である。配列の長さが 0 または 1 のときは、単に配列を返すだけで、比較を行わない。

長さの合計が n である 2 つのリストを併合するときは、最大で $n-1$ 回の比較が必要である。長さ n の配列 a に対して `mergeSort(a,c)` を実行するときに必要な比較回数の最大値を $C(n)$ とする。n が偶数なら、それぞれの部分問題に対して帰納法の仮定を適用し、次のように計算できる。

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

n が奇数の場合は、やや複雑になる。この場合は、次に示す 2 つの不等式を使う。まず、任意の $x \ge 1$ について次の不等式が成り立つ。

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

また、任意の $x \ge 1/2$ について次の不等式が成り立つ。

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

いずれの不等式も簡単に検証できる。まず、不等式 (11.1) は、$\log(x) + 1 = \log(2x)$ が成り立つことからいえる。不等式 (11.2) は、log が上に凸な関数であるであることによりいえる。これらの不等式を利用することで、奇数 n について次の式が成り立つ。

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

□

11.1.2 クイックソート

クイックソート（quicksort）も古典的な分割統治アルゴリズムである。クイックソートでは、2 つの部分問題を解いたあとで結果を併合するマージソートとは違い、事前にすべての処理を済ませてしまう。

クイックソートの説明は単純だ。まず、a からランダムに**軸**（pivot）となる要素 x を選ぶ。そして、x より小さい要素、x と同じ要素、x より大きい要素の 3 つに a を分割する。そして、分割の 1 つめと 3 つめを再帰的に整列する。図 11.3 に例を示す。

Algorithms

```
void quickSort(array<T> &a) {
  quickSort(a, 0, a.length);
}
void quickSort(array<T> &a, int i, int n) {
  if (n <= 1) return;
  T x = a[i + rand()%n];
  int p = i-1, j = i, q = i+n;
  // a[i..p]<x,  a[p+1..q-1]??x, a[q..i+n-1]>x
  while (j < q) {
    int comp = compare(a[j], x);
    if (comp < 0) {
      a.swap(j++, ++p); // 配列の前方に移す
    } else if (comp > 0) {
      a.swap(j, --q);   // 配列の後方に移す
    } else {
      j++;              // 中ほどに残す
    }
  }
  // a[i..p]<x,  a[p+1..q-1]=x, a[q..i+n-1]>x
  quickSort(a, i, p-i+1);
  quickSort(a, q, n-(q-i));
}
```

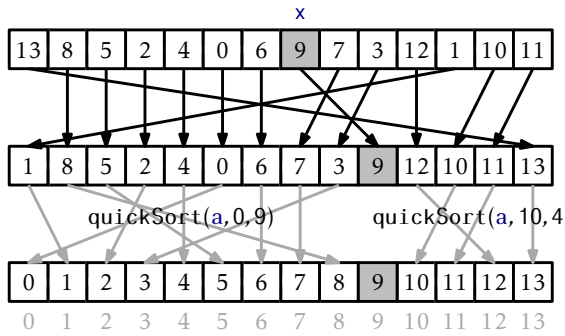


図 11.3: quickSort(a,0,14) の実行例

すべての処理は in-place[*1] で実行される。したがって、整列のために配列の各部をコピーすることはない。その代わり、quickSort(a,i,n) では、配列の一部 a[i],...,a[i+n−1] だけを整列する。このメソッドを、最初に quickSort(a,0,n) として呼び出すわけである。

クイックソートのアルゴリズムで最も肝心なポイントは、in-place な分割を行うアルゴリズムである。このアルゴリズムでは、無駄な領域を使わずに a の中の要素を入れ替え、次の制約を満たすような添字 p と q を計算する。

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

コード中の while ループでは、繰り返しのたびに、これらの制約の 1 つめと 3 つめの条件を保ちながら p が増加し、q は減少していく。繰り返しの各ステップで、j 番めの位置にある要素は、前に動くか、その場に留まるか、後ろに動く。前に動くか、その場に留まる場合は、j を 1 増やす。後ろに動く場合は、j 番めの要素は未処理ということなので、j を増やさない。

クイックソートは、7.1 節で学んだランダム二分探索木と深い関係がある。実は、相異なる要素をクイックソートに入力した場合の再帰木は、ランダム二分探索木とみなせるのである。ランダム二分探索木を作るときは、まずランダ

[*1] 訳注：in-place とは、入力配列 a 以外にはどの時点においても定数個の変数しか使わない、ということを表す。前節のマージソートの実装は、a の一部をコピーするために別の配列を必要としたので、in-place なアルゴリズムではない。ただし、マージソートを in-place に行う方法も存在する。

ムに要素 x を選び、これを根にしたうえで各要素を x と比較して、最終的に小さい要素を左の部分木とし、大きい要素を右の部分木とした。そのことを思い出せば、両者の関係が見えてくるだろう。

クイックソートでは、ランダムに要素 x を選び、x と全要素とを比較したうえで、x より小さい要素を配列の前方に、大きい要素を配列の後方に集める。それから、前方の配列と後方の配列をそれぞれ再帰的に整列する。同じように、ランダム二分木では、小さい要素を左の部分木、大きい要素を右の部分木とする操作を再帰的に繰り返した。

ランダム二分探索木とクイックソートにこのような対応があるということは、補題 7.1 をクイックソートの言葉で言い換えられるということである。

補題 **11.1.** クイックソートで整数 $0,\ldots,n-1$ を含む配列を整列するとき、要素 i と軸とが比較される回数の期待値は、$H_{i+1}+H_{n-i}$ 以下である。

調和数の計算により、クイックソートの実行時間に関する次の定理が得られる。

定理 **11.2.** n 個の相異なる要素をクイックソートで整列するとき、実行される比較の回数の期待値は $2n\ln n+O(n)$ 以下である。

証明**.** n 個の相異なる要素をクイックソートで整列するときに実行される比較の回数を T とする。補題 11.1 と期待値の線形性より次が成り立つ。

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

□

定理 11.3 は、整列する要素が互いに相異なる場合についての定理である。入力の配列 a が重複する要素を含むとき、クイックソートの実行時間の期待値は悪くはならず、むしろ向上する場合さえある。これは、重複する要素 x が軸に選ばれた場合に x がまとめられ、2 つの部分問題のいずれにも含まれないからである。

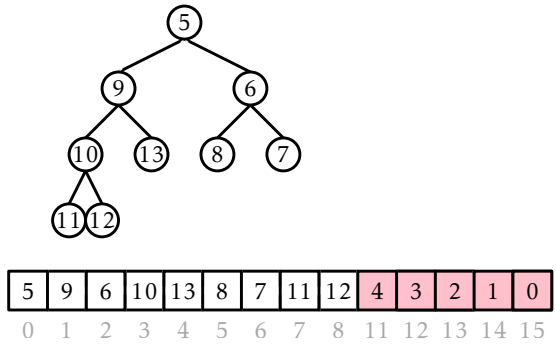


図 11.4: heapSort(a,c) の実行中のある瞬間の様子。色がついている部分はすでに整列済みの配列である。色がついていない部分は BinaryHeap になっている。次の繰り返しでは、要素 5 が配列の位置 8 に移る

定理 **11.3.** quickSort(a) の実行時間の期待値は $O(n \log n)$ である。また、実行される比較の回数の期待値は $2n \ln n + O(n)$ 以下である。

11.1.3 ヒープソート

ヒープソート（heap sort）も in-place で処理を行う整列アルゴリズムである。ヒープソートでは、10.1 節で説明した二分木の BinaryHeap を使う。BinaryHeap では、ヒープを表現するのに配列を 1 つ使っていたことを思い出そう。入力の配列をヒープに変換したあとで最小値を取り出すという操作を繰り返すのがヒープソートである。

具体的には、n 個の要素を配列 a に格納する。各要素は a[0],...,a[n − 1] に入っており、最小値は根、すなわち a[0] である。a を BinaryHeap に変換したあとで、次の操作を繰り返すのがヒープソートだ。すなわち、a[0] と a[n − 1] を入れ替え、n を 1 減らし、a[0],...,a[n − 2] を再びヒープにするために trickleDown(0) を呼ぶ。繰り返しが終了した段階、すなわち n = 0 になった段階で、a の要素は降順に並んでいる。よって、a を逆順にすれば、最終的な整列された状態になる [*2]。図 11.4 に heapSort(a,c) の実行の様子を示す。

[*2] compare(x,y) を定義し直せば、繰り返しの結果がそのまま昇順に並ぶようにもできる。

BinaryHeap
```
void sort(array<T> &b) {
  BinaryHeap<T> h(b);
  while (h.n > 1) {
    h.a.swap(--h.n, 0);
    h.trickleDown(0);
  }
  b = h.a;
  b.reverse();
}
```

ヒープソートにおいて最も重要な処理は、未整列の配列 a からヒープを構築する部分である。BinaryHeap の add(x) を繰り返し実行することで、これを $O(n \log n)$ の時間で簡単に済ませてもいいだろう。しかし、ボトムアップなアルゴリズムを採用することで、さらに効率良くヒープを構築できる。BinaryHeap では、a[i] の子が a[2i+1] と a[2i+2] に入っていた。ということは、$a[\lfloor n/2 \rfloor], \ldots, a[n-1]$ は子を持っていない。別の言い方をすれば、$a[\lfloor n/2 \rfloor], \ldots, a[n-1]$ は、それぞれ大きさ 1 の部分ヒープである。そこで、逆順に、$i \in \{\lfloor n/2 \rfloor - 1, \ldots, 0\}$ に対して順番に trickleDown(i) を呼べばよい。これでうまくいくのは、trickleDown(i) を呼ぶ時点までに、a[i] の子がいずれも部分ヒープの根になっているからだ。したがって、trickleDown(i) を呼ぶと、a[i] が次の部分ヒープの根になる。

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

このボトムアップな方法には、add(x) を n 回実行するよりも効率的という特徴がある。それを見るには、葉の位置に存在する要素は n/2 個あり、こ

れらの要素には何もする必要がないこと、葉の親であって、a[i] を根とする高さが 1 の部分ヒープは n/4 個あり、これらに対しては trickleDown(i) を一度だけ呼べばよいこと、高さが 2 の部分ヒープは n/8 個あり、これらに対しては trickleDown(i) を二度呼べばよいこと……、に注目すればよい。trickleDown(i) の実行時間は、a[i] を根とする部分ヒープの高さに比例するので、実行時間の合計は高々次の値である。

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

最後から 2 つめの等号は、$\sum_{i=1}^{\infty} i/2^i$ がコインを投げて表が出るまでに要する回数（表が出た回を含む）の期待値に等しいこと（期待値の定義）、および、補題 4.2 から成り立つ。

heapSort(a,c) の性能は、次の定理によって説明できる。

定理 **11.4.** heapSort(a,c) の実行時間は $O(n \log n)$ であり、その際に実行する比較の回数は、最大でも $2n \log n + O(n)$ である。

証明. このアルゴリズムには 3 つのステップがある。すなわち、(1) a をヒープに変形し、(2) a の最小値を繰り返し取り出し、(3) a を逆順にする。ステップ 1 の実行時間は $O(n)$ で、$O(n)$ 回の比較を行う。ステップ 3 の実行時間は $O(n)$ で、比較は行わない。ステップ 2 では trickleDown(0) を n 回呼ぶ。i 番めの呼び出しは、大きさ $n-i$ のヒープに対するもので、最大で $2\log(n-i)$ 回の比較を行う。i についての和を取ると次の値が得られる。

$$\sum_{i=0}^{n-1} 2\log(n-i) \le \sum_{i=0}^{n-1} 2\log n = 2n \log n$$

3 つのステップにおける比較の実行回数を足し合わせれば証明が完成する。 □

11.1.4 比較ベースの整列における下界

比較に基づく 3 つの整列アルゴリズムの実行時間は、いずれも $O(n \log n)$ であった。ここで気になるのは、もっと速いアルゴリズムがあるかどうかである。端的に答えると、これは存在しない。a の要素に対して実行できる操作が比較だけなら、$n \log n$ 回程度の比較が必要なのである。これを示すのは難し

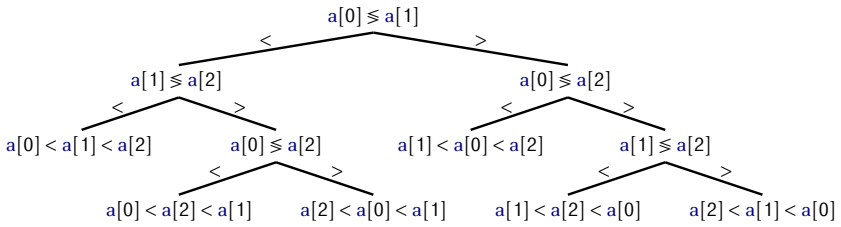


図 11.5: 長さ n = 3 の配列 a[0], a[1], a[2] を整列するときの比較木

くないが、そのためには想像力が必要だ。最終的は次の事実から導かれる。

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

（この事実の証明を問 11.10 とした。）

まずは、マージソートやヒープソートのような決定的なアルゴリズムを、ある特定の値 n について実行した場合について考える。このようなアルゴリズムで n 個の相異なる要素を整列する場面を想像してみよう。下界を示すには、n を固定したとき、決定的なアルゴリズムで最初に比較する要素が常に決まったペアであることに注目する。例えば heapSort(a, c) では、n が偶数の場合、最初に trickleDown(i) を呼ぶときは i = n/2 − 1 であり、a[n/2 − 1] と a[n − 1] とを最初に比較する。

入力の要素はすべて相異なっているので、最初の比較の結果は 2 通りしかない。2 回めの比較の結果は、最初の比較の結果に依存するだろう。3 回めの比較は、その前の 2 回の比較の結果に依存するだろう。以降も同様に考えると、比較に基づく決定的な整列アルゴリズムは、根を持った二分木になる。この二分木を**比較木**（comparison tree）と呼ぶことにする。比較木のノード u には、u.i と u.j という添字の組が対応していて、a[u.i] < a[u.j] ならば整列アルゴリズムが左の部分木に進むものと考える。そうでなければ、整列アルゴリズムが右の部分木に進むものと考える。比較木の葉 w には、0, . . . , n − 1 の置換 w.p[0], . . . , w.p[n − 1] のいずれかが対応している。整列アルゴリズムが、比較木の特定の葉に到達するということは、その葉に対応する置換で a の整列を表現できることを意味する。つまり、a は次のように整列される。

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

大きさ n = 3 の配列に対する比較木の例を図 11.5 に示す。

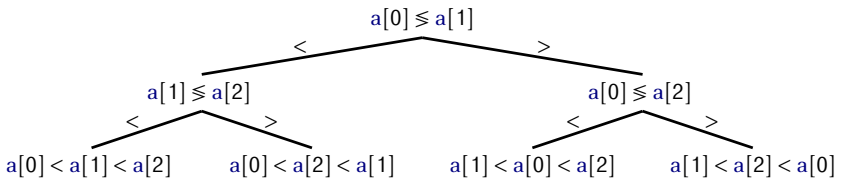


図 11.6: 正しく整列できない入力が存在する比較木

整列アルゴリズムの比較木を見ることで、そのアルゴリズムのすべてがわかる。n 個の相異なる要素からなる配列 a を入力とした場合に実行される一連の比較が正確にわかり、a を整列するためにアルゴリズムがどうやって a を並べ替えているかが見えるのである。必然的に、比較木には少なくとも n! 個の葉がある。もしそうでなければ、同じ葉に到達する別々の入力が 2 つ存在してしまう。その場合、そのアルゴリズムでは、同じ葉に到達する入力のうち少なくとも 1 つを正しく整列できないことになる。

例えば図 11.6 に示した比較木には 4 つ（$< 3! = 6$）の葉がある。この木を見ると、2 つの入力 $3,1,2$ と $3,2,1$ がいずれも右端の葉に到達することがわかる。入力 $3,1,2$ に対する出力は $a[1] = 1, a[2] = 2, a[0] = 3$ であり、これは正しく整列されている。しかし、入力 $3,2,1$ に対する出力は $a[1] = 2, a[2] = 1, a[0] = 3$ となり、正しく整列されていない。以上の議論より、比較に基づくアルゴリズムの本質的な下界が得られる。

定理 **11.5.** 比較に基づく任意の決定的な整列アルゴリズム $\mathcal{A}$ と、任意の整数 $n \geq 1$ について、ある長さ n の入力配列 a が存在して $\mathcal{A}$ が a を整列するとき、$\log(n!) = n \log n - O(n)$ 回の比較が実行される。

証明. これまでの議論から、$\mathcal{A}$ の比較木には少なくとも n! 個の葉が必要である。帰納法により、k 個の葉を持つ二分木の高さが $\log k$ 以上であることを簡単に示せる。よって、$\mathcal{A}$ の比較木には深さ $\log(n!)$ 以上の葉 w が存在し、ある入力配列 a がこの葉に到達する。この a に対して、$\mathcal{A}$ では $\log(n!)$ 回以上の比較が実行される。 □

定理 11.5 は、マージソートやヒープソートなどの決定的なアルゴリズムに関するものである。クイックソートのようなランダム性を利用するアルゴリズムに関しては、定理 11.5 では何もわからない。ランダム性を利用するアルゴリズムであれば、比較回数の下界 $\log(n!)$ を打ち破れるだろうか。実は、や

はりできないのである。これを示すには、ランダム性を利用するアルゴリズムを別の観点から考えてみればよい。

以下の議論では、決定木は次の意味で「整理されている」と仮定する。具体的には、どのような入力配列 a によっても到達できないノードは取り除かれているとする。このとき、木にはちょうど n! 個だけの葉が含まれる。葉の数が n! 以上であることは、そうでないと正しく整列できないことからわかる。葉の数が n! 以下であることは、n 個の相異なる要素の置換は n! 通りであり、それぞれが決定木における根から葉への経路をちょうど 1 つ辿ることからわかる。

ランダムなソートアルゴリズム $\mathcal{R}$ は、2 つの入力を取る決定的なアルゴリズムだと考えられる。その 2 つの入力とは、整列すべき入力配列 a と、$[0,1]$ 内のランダムな実数の長い列 $b = b_1, b_2, b_3, \ldots, b_m$ である。b は、アルゴリズムが利用するランダム性のために必要になる。すなわち、アルゴリズムでコイン投げによるランダムな選択が必要になったときに b の要素を使う。例えば、クイックソートにおいて最初の軸を選ぶとき、アルゴリズムでは式 $\lfloor nb_1 \rfloor$ を使う。

b として、ある固定的な列 $\hat{b}$ を使うと、$\mathcal{R}$ は決定的なアルゴリズムになる。このアルゴリズムを $\mathcal{R}(\hat{b})$ とし、これによる比較木を $\mathcal{T}(\hat{b})$ とする。a を $\{1,\ldots,n\}$ の置換からランダムに選ぶことは、$\mathcal{T}(\hat{b})$ の n! 個の葉のうちの 1 つをランダムに選ぶことと同じである点に注意してほしい。

問 11.12 の証明では、k 個の葉を持つ二分木の葉をランダムに選ぶときに、葉の深さの期待値が $\log k$ 以上であることが必要だった。したがって、$\{1,\ldots,n\}$ の置換からランダムに選んだものを入力とするとき、決定的なアルゴリズム $\mathcal{R}(\hat{b})$ で実行される比較の回数の期待値は $\log(n!)$ 以上である。結局、任意の $\hat{b}$ についてこれが成り立つので、$\mathcal{R}$ についても同じことが成り立つ。こうしてランダム性を利用するアルゴリズムについても下界が示された。

定理 11.6. 任意の整数 $n \geq 1$ と、任意の比較（決定的でもランダムでもよい）に基づく整列アルゴリズム $\mathcal{A}$ について、$\{1,\ldots,n\}$ の置換からランダムに選んだ入力を整列するときに実行する比較の回数の期待値は $\log(n!) = n\log n - O(n)$ 以上である。

11.2 計数ソートと基数ソート

この節では、比較に基づくものではない整列アルゴリズムを 2 つ紹介する。この節で紹介するアルゴリズムは、小さい整数を整列することに特化し、要素（の一部）を配列の添字として使うことで、定理 11.5 の下界に制約されない。次のような文を考えてみよう。

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

この文は定数時間で実行できるが、a[i] の値が何であるかに応じて、c.length 種類の値を取りうる。つまり、このような文を使うアルゴリズムは、比較木のような二分木ではモデル化できない。突き詰めると、これこそが、この節で紹介するアルゴリズムが比較に基づくアルゴリズムよりも速く整列をこなせる理由なのである。

11.2.1 計数ソート

0,...,k − 1 の範囲の要素 n 個からなる入力の配列 a があるとする。**計数ソート**（counting sort）では、a を整列するのに、カウンタを保持する補助配列 c を使う。そして、整列された a を、補助配列 b として返す。

計数ソートの背後にあるアイデアは単純だ。各 i ∈ {0,...,k − 1} について i が a に登場する回数を数え、それを c[i] に入れていく。整列が終わったときには、c[0] 個の 0 が続き、c[1] 個の 1 が続き、c[2] 個の 2 が続き、…、c[k − 1] 個の k − 1 が続くような出力になっているだろう。この操作を巧妙なコードで実行する。実行の様子を図 11.7 に示す。

Algorithms

```
void countingSort(array<int> &a, int k) {
  array<int> c(k, 0);
  for (int i = 0; i < a.length; i++)
    c[a[i]]++;
  for (int i = 1; i < k; i++)
    c[i] += c[i-1];
  array<int> b(a.length);
  for (int i = a.length-1; i >= 0; i--)
    b[--c[a[i]]] = a[i];
```

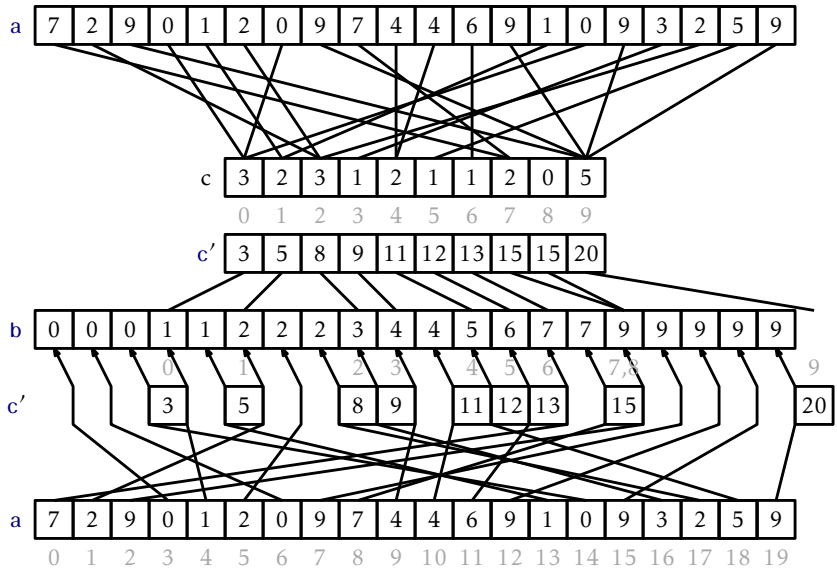


図 11.7: 0,…,k − 1 = 9 のいずれかを格納する、長さ n = 20 の配列に対する計数ソートの操作

```
  a = b;
}
```

このコードの最初の for ループでは、それぞれカウンタ c[i] を設置し、a において i が何回現れるかを数えている。a の値を添字として使うことにより、すべての処理を 1 つの for ループにより $O(n)$ で終えている。この段階で、c から直接、出力配列 b を埋めていくこともできるだろう。しかし、a の要素に何かしらのデータが関連づけられている場合、それだとうまくいかない。そこで、a から b へと要素をコピーする作業が追加で必要になる。

次の for ループでは、c[i] が a における i 以下の要素の個数になるように、カウンタを順に足しこんでいる。これには $O(k)$ の時間がかかる。任意の $i \in \{0,\dots,k-1\}$ について、出力配列 b は特に次の式を満たす。

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

最後に、要素を b へと順番に入れていくため、a を逆順に辿る。その際は、要素 $a[i] = j$ を $b[c[j]-1]$ に入れ、$c[j]$ を 1 減らす。

定理 **11.7.** countingSort(a,k) は、集合 {0,...,k − 1} に含まれる n 個の要素からなる配列 a を $O(n+k)$ の時間で整列する。

計数ソートには**安定性**（stable）という良い性質がある。これは、元の配列において等しい要素同士の相対的な位置が、整列後の配列でも保たれるという性質である。すなわち、要素 a[i] と a[j] が等しい値で、i < j であるとき、b においても a[i] が a[j] の前にくる。この性質は次の項で役に立つ。

11.2.2 基数ソート

計数ソートは、配列内の要素の最大値 k − 1 と比べて配列の長さ n がそれほど小さくない場合には非常に効率的である。これから説明する**基数ソート**（radix sort）では、最大値が比較的大きな場合でも効率的になるように、計数ソートを複数回利用する。

基数ソートでは、w ビットの整数を整列させるのに、一度に d ビットずつ、w/d 回の計数ソートを実行する [*3]。より正確に言うと、基数ソートでは、まず整数の最下位 d ビットだけを見て整列する。続いて、次の d ビットだけを見て整列する。これを繰り返し、最後には整数の最高位 d ビットだけを見て整列する。

Algorithms

```
void radixSort(array<int> &a) {
  int d = 8, w = 32;
  for (int p = 0; p < w/d; p++) {
    array<int> c(1<<d, 0);
        // 次の 3 つの for ループで計数ソートを行う
        array<int> b(a.length);
    for (int i = 0; i < a.length; i++)
      c[(a[i] >> d*p)&((1<<d)-1)]++;
    for (int i = 1; i < 1<<d; i++)
      c[i] += c[i-1];
    for (int i = a.length-1; i >= 0; i--)
      b[--c[(a[i] >> d*p)&((1<<d)-1)]] = a[i];
    a = b;
```

[*3] d は w を割り切れると仮定する。もしそうでないときは、w を d⌈w/d⌉ に増やす必要がある。

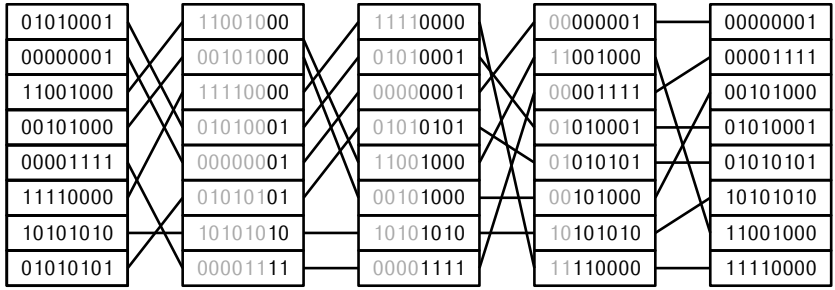


図 11.8: 基数ソートにより $w = 8$ ビットの整数を整列する。$d = 2$ ビットの整数を計数ソートによって整列する処理を 4 回行う

```
  }
}
```

（このコードでは、(a[i] >> d*p)&((1 << d) − 1) と書くことで、a[i] の二進表記において $(p+1)d-1,\ldots,pd$ ビットめを抜き出して並べた整数を取り出している。）

このアルゴリズムの例を図 11.8 に示す。

このアルゴリズムで正しく整列できるのは、計数ソートが安定なソートアルゴリズムだからである。a の 2 つの要素 x と y が $x < y$ を満たし、x と y が添字 r の位置で異なるなら、$\lfloor r/d \rfloor$ 回めの整列で x は y より前に置かれる。そして、以降は x と y の相対的な位置は変わらない。

基数ソートでは w/d 回の計数ソートを実行する。各計数ソートの実行時間は $O(n+2^d)$ である。よって、基数ソートの性能は次の定理のようになる。

定理 11.8. 任意の整数 $d > 0$ について、radixSort(a,k) は、n 個の w ビット整数を含む配列 a を、$O((w/d)(n+2^d))$ の時間で整列する。

配列の要素が $\{0,\ldots,n^c-1\}$ の範囲の整数であり、$d = \lceil \log n \rceil$ だとすれば、定理 11.8 は次のように整理できる。

系 11.1. radixSort(a,k) は、$\{0,\ldots,n^c-1\}$ の範囲の n 個の整数からなる配列 a を、$O(cn)$ の時間で整列する。

11.3 ディスカッションと練習問題

整列は計算機科学における基本的なアルゴリズムであり、長い歴史がある。Knuth によれば、マージソートは von Neumann が 1945 年に考案したものだという [48]。クイックソートは Hoare が考案した [39]。最初にヒープソートを考案したのは Williams である [76]。ただし、本節で説明した $O(n)$ の時間でボトムアップにヒープを構築する方法は、Floyd によるものである [28]。比較に基づくソートの下界は古くから知られていた。次の表に、比較に基づくアルゴリズムの性能をまとめる。

	時間計算量	in-place
マージソート	$n \log n$ 最悪実行時間	No
クイックソート	$1.38 n \log n + O(n)$ 期待実行時間	Yes
ヒープソート	$2n \log n + O(n)$ 最悪実行時間	Yes

これらの比較に基づくアルゴリズムには、それぞれ長所と短所がある。マージソートは比較の回数が最も少なく、ランダムでもない。しかし、併合に補助配列が必要だ。この配列を確保するコストが高い。また、メモリの制限によりソートに失敗する可能性もある。クイックソートは、入力配列だけで処理が可能であり、比較の回数も 2 番めに少ないが、ランダム性を利用するアルゴリズムなので実行時間の保証が常には成り立たない。ヒープソートは、比較の回数は最も多いが、入力配列だけを使った決定的なアルゴリズムである。

マージソートが明らかに一番優れている場面がある。それは、連結リストを整列する場合である。この場合は補助的な配列が必要ない。2 つの整列済みの連結リストは、ポインタ操作によって簡単に併合でき、整列済みの配列が 1 つ得られる（問 11.2 を参照）。

この章で説明した計数ソートと基数ソートは Seward によるものである [66, Section 2.4.6]。ただし、1920 年代から、パンチカードを整列するために基数ソートの一種が使われていた機械がある。この機械では、カードの山を、ある場所に穴が空いているかどうかを判定して 2 つの山に分けた。さまざまな穴の位置についてこの処理を繰り返すことで、基数ソートになる。

最後に、計数ソートと基数ソートがいずれも非負整数以外の数も整列できることを確認しておく。計数ソートを単純に修正して、$\{a, \ldots, b\}$ の範囲の整数を整列できるようにすれば、実行時間は $O(n + b - a)$ になる。同様に、基数ソートは同じ範囲の整数を $O(n(\log_n(b-a))$ の時間で整列するように修正できる。なお、いずれのアルゴリズムも、IEEE754 形式の浮動小数点数の整列にも使

える。これは、IEEE 754 形式は、数の大きさに符号が付いた二進整数表現とみなして比較できるように設計されているからである [2]。

問 **11.1.** 1,7,4,6,2,8,3,5 からなる配列を入力とする、マージソートとヒープソートを実行する様子を描け。また、同じ配列について、クイックソートを実行する様子としてありえるものを 1 つ描け。

問 **11.2.** マージソートの一種で、補助配列を使わずに `DLList` を整列するものを実装せよ（問 3.13 を参照）。

問 **11.3.** `quickSort(a,i,n)` の実装には、常に $a[i]$ を軸として選ぶものがある。この実装で $\binom{n}{2}$ 回の比較が実行されるような、長さ n の入力の例を示せ。

問 **11.4.** `quickSort(a,i,n)` の実装には、常に $a[i+n/2]$ を軸として選ぶものがある。この実装で $\binom{n}{2}$ 回の比較が実行されるような、長さ n の入力の例を示せ。

問 **11.5.** 軸を決定的に選び、最初に $a[i],\ldots,a[i+n-1]$ を見ることがない `quickSort(a,i,n)` の実装には、どんな実装であっても、$\binom{n}{2}$ 回の比較が実行されるような長さ n のある入力が存在することを示せ。

問 **11.6.** `quickSort(a,i,n)` に渡す `Comparator c` で、クイックソートにおいて $\binom{n}{2}$ 回の比較が実行されるようなものを設計せよ（ヒント：`Comparator` は比較する値を実際に見なくてもよい）。

問 **11.7.** クイックソートが実行する比較の回数の期待値を、定理 11.3 の証明より細かく解析せよ。具体的には、比較の回数の期待値が $2nH_n - n + H_n$ であることを示せ。

問 **11.8.** ヒープソートを実行する際の比較の回数が $2n\log n - O(n)$ になる入力配列を与え、そのことを説明せよ。

問 **11.9.** 図 11.6 の比較木によって正しく整列できない 1,2,3 の置換として、別の例を見つけよ。

問 **11.10.** $\log n! = n\log n - O(n)$ を示せ。

問 **11.11.** k 個の葉を持つ二分木の高さは $\log k$ 以上であることを示せ。

問 **11.12.** k 個の葉を持つ二分木の葉をランダムに選ぶとき、その葉の高さの期待値は $\log k$ 以上であることを示せ。

問 **11.13.** この章で説明した `radixSort(a,k)` は、入力配列 `a` が非負整数だけからなるときに動作する。入力配列が負の整数を含むときにも正しく動作するように実装を修正せよ。

第 12

グラフ

この章ではグラフの 2 つの表現方法を説明し、グラフを扱う基本的なアルゴリズムを紹介する。

数学における**有向グラフ**（directed graph）とは、**頂点**（vertex）の集合 V と、**辺**（edge）の集合 E からなる、組 $G = (V, E)$ である。なお、辺は頂点の組 (i, j) であり、i から j に向かっているものとする。i は辺の**始点**（source）と呼ばれ、j は**終点**（target）と呼ばれる。頂点の列 $v_0, \ldots, v_k$ は、任意の $i \in \{1, \ldots, k\}$ について辺 (v_{i-1}, v_i) が E に含まれる場合、G における**経路**（path）と呼ばれる。経路 $v_0, \ldots, v_k$ は、(v_k, v_0) も E の要素であるとき、**循環**（cycle）と呼ばれる。経路（または循環）に含まれる頂点が互いに異なる場合、その経路（または循環）は**単純**（simple）であるという。頂点 v_i から頂点 v_j への経路があるとき、v_j は v_i から**到達可能**（reachable）であるという。図 12.1 にグラフの例を示す。

グラフを使ってモデル化できる現象は多く、そのためグラフには多くの応用がある。自明な例をいくつか挙げよう。コンピュータのネットワークは、コンピュータを頂点、それらを繋ぐ（直接の）通信路を辺とみなせば、グラフとしてモデル化できる。街道は、交差点を頂点、それらを繋ぐ通りを辺とみなせば、グラフとしてモデル化できる。

グラフが集合における二項関係のモデルであることに着目すると、もっと意外な例が見つかる。例えば、大学の時間割における**衝突グラフ**（conflict graph）というものが考えられる。このグラフでは、頂点を大学の講義とする。辺 (i, j) は、講義 i と講義 j の両方を受講する生徒がいることを表している。よって、そのような辺があれば、講義 i と講義 j のテストを同じ時間に実施できないことがわかる。

この節を通じて、n は頂点の数、m は辺の数を表すことにする。すなわち、

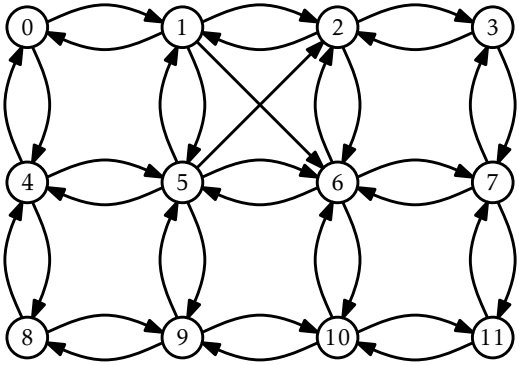


図 12.1: 12 個の頂点からなるグラフ。頂点は番号付きの円で、辺は source から target に向かう矢印で描く

n = |V| かつ m = |E| である。さらに、V = {0,...,n − 1} と仮定する。V の各頂点とひも付けられたデータを保存するには、大きさ n の配列にデータを入れておけばよい。

グラフに対する典型的な操作には次のようなものがある。

- addEdge(i, j)：辺 (i, j) を E に加える
- removeEdge(i, j)：辺 (i, j) を E から除く
- hasEdge(i, j)：(i, j) ∈ E かどうかを調べる
- outEdges(i)：(i, j) ∈ E を満たす整数整数 j のリストを返す
- inEdges(i)：(j, i) ∈ E を満たす整数整数 j のリストを返す

これらの操作を効率的に実装するのはさほど難しくない。例えば、はじめの 3 つの操作は USet を使って実装できる。5 章で説明したハッシュテーブルを使えば期待実行時間は定数である。最後の 2 つの操作は、各頂点ごとに隣接する頂点のリストを保持すれば、定数時間で実行できる。

しかし、グラフを何に応用するかによって、各操作への要求は異なる。理想的には、そうした要求をすべて満たす中で最も単純な実装を使いたい。そのため、この章ではグラフを表現する方法を大きく 2 つに分けて議論する。

12.1 AdjacencyMatrix：行列によるグラフの表現

n 個の頂点を持つグラフ $G=(V,E)$ を、真偽値を並べた n×n 行列 a を使って表現したものを、**隣接行列**（adjacency matrix）と呼ぶ。

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

隣接行列の要素 a[i][j] は次のように定義される。

$$\texttt{a[i][j]} = \begin{cases} \texttt{true} & (\texttt{i},\texttt{j}) \in E \text{ のとき} \\ \texttt{false} & \text{そうでないとき} \end{cases}$$

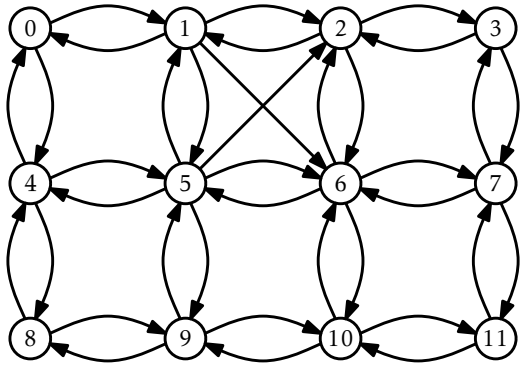
図 12.1 のグラフの隣接行列を図 12.2 に示す。

隣接行列による表現において、addEdge(i,j)、removeEdge(i,j)、hasEdge(i,j) の各操作は、いずれも行列の要素 a[i][j] を読み書きするだけで実装できる。

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

いずれの操作も明らかに定数時間で実行できる。

隣接行列による表現で効率がよくない操作は、outEdges(i) と inEdges(i) である。これらを実装するには、a の対応する行または列にある n 個の要素を順にすべて見て、各添字 j についてそれぞれ a[i][j] と a[j][i] が真かどうかを確認しなければならない。



	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

図 12.2: グラフとその隣接行列

AdjacencyMatrix

```
void outEdges(int i, List &edges) {
  for (int j = 0; j < n; j++)
    if (a[i][j]) edges.add(j);
}
void inEdges(int i, List &edges) {
```

```
  for (int j = 0; j < n; j++)
    if (a[j][i]) edges.add(j);
}
```

これらの操作には明らかに $O(n)$ の時間がかかる。

隣接行列による表現のもう 1 つの短所は、行列がかさばることである。真偽値からなる $n \times n$ 行列を格納するのに、n^2 ビット以上のメモリが必要になる。この実装では `bool` という値を使っているので、実際に必要なメモリは n^2 バイトのオーダーになる。実装を工夫して `w` 個の真偽値を各ワードに詰め込めば、空間使用量を $O(n^2/w)$ ワードに減らせるだろう。

定理 **12.1.** *AdjacencyMatrix* は *Graph* インターフェースを実装する。*AdjacencyMatrix* は以下の各操作をサポートする。

- `addEdge(i,j)`、`removeEdge(i,j)`、`hasEdge(i,j)` を定数時間で実行できる
- `inEdges(i)`、`outEdges(i)` を $O(n)$ の時間で実行できる

AdjacencyMatrix の空間使用量は $O(n^2)$ である。

メモリ使用量の多さと、`inEdges(i)` および `outEdges(i)` の性能の低さにもかかわらず、`AdjacencyMatrix` が有用な場合はある。具体的には、グラフ G が密なとき、つまり辺の数が n^2 に近い場合には、n^2 というメモリ使用量を許容できるだろう。

`AdjacencyMatrix` が広く使われる別の理由として、グラフ `G` の性質を効率的に計算するために行列 `a` の代数的な操作を使えるという点が挙げられる。これはアルゴリズムに関する話題だが、そのような性質を 1 つ紹介しよう。`a` の要素を整数（`true` が 1、`false` が 0）であるとみなし、`a` 同士の積を行列の掛け算を使って計算すると、行列 a^2 が求まる。積の定義から成り立つ次の関係を思い出してほしい。

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

グラフ G の文脈で解釈すると、この和は G が辺 `(i,k)` と辺 `(k,j)` を共に持つ頂点 `k` の個数になる。つまり、この和は `i` から `j` への（中間頂点 `k` を通る）経路のうち、長さがちょうど 2 であるものの個数である。この観察に基づいて、G におけるすべての頂点の対についての最短経路を $O(\log n)$ 回だけの行

列の積で計算するアルゴリズムが考案されている。

12.2 AdjacencyLists：リストの集まりとしてのグラフ

隣接リスト（adjacency list）は、グラフの表現において辺を重視するアプローチである。隣接リストの実装にはさまざまな方法がある。この節では、その中でも単純な実装について説明し、それ以外の方法については末尾で紹介する。隣接リストによる表現では、グラフ $G=(V,E)$ を、リストの配列 adj で表現する。リスト adj[i] には、頂点 i と、隣接するすべての頂点が含まれる。つまり、adj[i] は、$(i,j)\in E$ を満たすすべての添字 j からなるリストである。

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

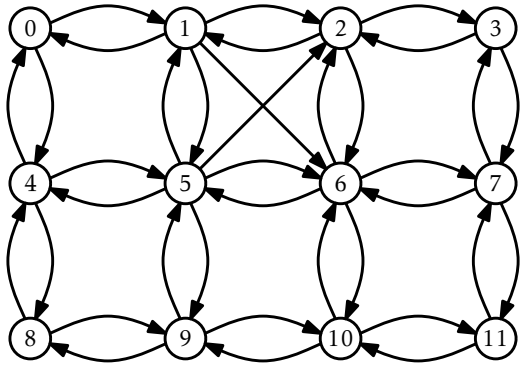
図 12.3 に例を示す。この実装では、リスト adj は ArrayStack のサブクラスとする。なぜなら、添字を使って定数時間で要素にアクセスしたいからである。他の選択肢もありうる。特に、adj を DLList として実装してもよいだろう。

addEdge(i,j) は、リスト adj[i] に j を加えるだけだ。これは定数時間で実行できる。

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

removeEdge(i,j) では、リスト adj[i] から j を見つけ、それを削除する。これには $O(\deg(i))$ の時間がかかる。ここで、deg(i) は、E の要素のうち i から出ている辺の個数であり、i の**次数**（degree）と呼ばれる。

AdjacencyLists
```
void removeEdge(int i, int j) {
  for (int k = 0; k < adj[i].size(); k++) {
    if (adj[i].get(k) == j) {
      adj[i].remove(k);
      return;
```



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						

図 12.3: グラフとその隣接リスト

```
    }
  }
}
```

hasEdge(i, j) も同様だ。リスト adj[i] から j を探し、見つかれば真を、見つからなければ偽を返す。これにかかる時間は $O(\deg(i))$ である。

AdjacencyLists

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

outEdges(i) は、単純にリスト adj[i] の中身を出力リストにコピーする。これにかかる時間は $O(\deg(i))$ である。

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

inEdges(i) ではもう少し手順が増える。すべての頂点 j について (i,j) が存在するかどうかを確認し、もし存在したら j を出力リストに追加する。この操作にはかなり時間が必要で、すべての頂点の隣接リストを見て回る必要があるので、$O(n+m)$ の時間がかかる。

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

このデータ構造の性能を次の定理にまとめる。

定理 **12.2.** *AdjacencyLists* は *Graph* インターフェースを実装する。*AdjacencyLists* は以下のように各操作をサポートする。

- addEdge(i,j) は定数時間で実行できる
- removeEdge(i,j) および hasEdge(i,j) にかかる時間は $O(\deg(i))$ である
- outEdges(i) にかかる時間は $O(\deg(i))$ である
- inEdges(i) にかかる時間は $O(n+m)$ である

AdjacencyLists の空間使用量は $O(n+m)$ である。

すでに触れたように、グラフを隣接リストとして実装する具体的な方法には多くの選択肢がある。それらの中から実装方法を選ぶ際に考慮する点としては次のようなものがある。

- adj の要素を格納するのに、どんなデータ構造を使うか。配列ベースか、ポインタベースか、あるいはハッシュテーブルか
- 各 i について $(j,i) \in E$ を満たす頂点 j のリストを inadj とした場合、これを二次的な隣接リストとして利用するかどうか。二次的な隣接リ

ストを利用することで、inEdges(i) の実行時間が劇的に改善するが、辺を追加、削除する際の仕事が少し増える
- adj[i] における辺 (i, j) に、対応する inadj[j] のエントリへの参照を持たせるべきか
- 辺をオブジェクトとして明示的に実装し、関連データを持たせるべきか。その場合は、adj に、頂点のリストではなく辺のリストを持たせることになる

これらの選択肢の多くは、実装の時間的および空間的な複雑さと実装の性能とのトレードオフに帰結する。

12.3 グラフの走査

この節では、グラフの頂点 i から開始して、i から到達可能なすべての頂点を探索するアルゴリズムを 2 つ紹介する。2 つとも、隣接リストで表現されたグラフに最適なアルゴリズムである。そのため、この節の解析では、グラフの表現が AdjacencyLists であることを仮定する。

12.3.1 幅優先探索

幅優先探索（breadth-first search）では、頂点 i から開始し、i に隣接する頂点、i の隣の隣、i の隣の隣の隣、という順番で訪問していく。

このアルゴリズムは、二分木における幅優先の走査アルゴリズム（6.1.2 節）の一般化であり、とてもよく似ている。木の幅優先走査では、初期状態で i だけを含めたキュー q を使う。それから、q の要素を取り出してその要素に隣接する要素を q に追加していく。その際には、要素に隣接する要素について、いずれも過去に q に登場していないことを前提としていた。グラフにおける幅優先探索アルゴリズムでは、木の走査の場合と違い、同じ頂点を q に 2 回以上追加しないように気をつける必要がある。そのためには、真偽値の補助配列 seen を使い、すでに見つかっている頂点を記録しておけばよい。

Algorithms
```
void bfs(Graph &g, int r) {
  bool *seen = new bool[g.nVertices()];
  SLList<int> q;
  q.add(r);
```

```
  seen[r] = true;
  while (q.size() > 0) {
    int i = q.remove();
    ArrayStack<int> edges;
    g.outEdges(i, edges);
    for (int k = 0; k < edges.size(); k++) {
      int j = edges.get(k);
      if (!seen[j]) {
        q.add(j);
        seen[j] = true;
      }
    }
  }
  delete[] seen;
}
```

図 12.1 に対して bfs(g,0) を実行する様子の一例を図 12.4 に示す。この処理の順番は隣接リストの並び順によって異なる。図 12.4 の処理では図 12.3 の隣接リストを使った。

bfs(g,i) の実行時間は簡単に解析できる。seen のおかげで、同じ頂点が q に 2 回以上追加されることはない。q に頂点を追加する（そしてあとで削除する）処理は定数時間で実行でき、合計 $O(n)$ だけの時間がかかる。すべての頂点が内側のループで高々 1 回処理されるので、すべての隣接リストが高々 1 回処理される。よって、G の辺は高々 1 回だけ処理される。内部ループが一周するたびに辺が 1 つ処理され、一周は定数時間で実行できるので、合計 $O(m)$ だけの時間がかかる。以上より、アルゴリズム全体の実行時間は $O(n+m)$ である。

次の定理に bfs(g,r) の性能をまとめる。

定理 **12.3.** bfs(g,r) の実行時間は、*AdjacencyLists* で実装された *Graph* g を入力とすると、$O(n+m)$ である。

幅優先の走査には特別な性質がある。bfs(g,r) を呼ぶと、r からの有向経路が存在するすべての頂点が、最終的には q に追加（およびあとで削除）され

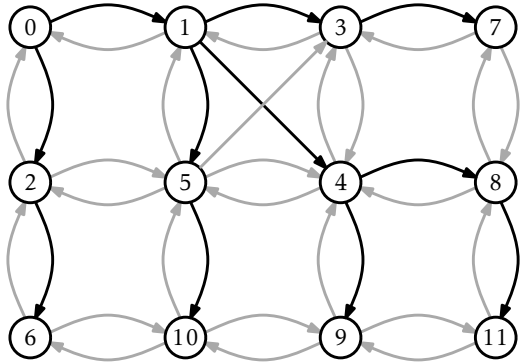


図 12.4: ノード 0 から始まる幅優先探索の例。ノードの数字はこの探索において q に追加される順番を表している。ノードが q に追加されるときに辿られる辺を黒、それ以外の辺を灰色で描いている

る。また、r から距離 0 の頂点（r 自身）は、r から距離 1 の頂点より先に q に追加され、距離 1 の頂点は距離 2 の頂点よりも先に q に追加される。以降も同様である。そのため、bfs(g,r) では、r からの距離の昇順で頂点を訪問していく。そして、r から到達不可能な頂点を訪問することはない。

このような方法で走査を行う幅優先探索の便利な応用として、最短経路の計算がある。r からすべての頂点への最短経路を求めるために、長さ n の補助配列 p を利用する bfs(g,r) の変種が利用できるのである。この変種では、頂点 j を q に追加するとき、p[j] = i とする。こうすると、p[j] が、r から j への最短経路における最後から 2 つめの頂点になる。p[p[j]]、p[p[p[j]]]、と繰り返し求めていくことで、r から j への最短経路を（逆順に）再構築できる。

12.3.2 深さ優先探索

グラフに対する**深さ優先探索**（depth-first search）は、二分木を走査するときの標準的なアルゴリズムに似ている。すなわち、ある部分木を完全に探索し終えてから根の方向に戻り、それから別の部分木の探索に進む。深さ優先探索は、スタックの代わりにキューを使う幅優先探索であると考えてもよい。

各頂点 i には、深さ優先探索の実行中、色 c[i] を割り当てる。未訪問の頂点の色を white、現在訪問中の頂点の色を grey、すでに訪問した頂点の色を black にする。深さ優先探索の最も簡単な方法は、再帰的なアルゴリズムに

よるものである。まず、r を訪問するところから処理を開始する。頂点 i を訪問するときには i の色を grey にする。それから i の隣接リストを見て、その中の white な頂点を再帰的に訪問する。i に対する処理が終わったら、最後に i の色を black にする。

Algorithms
```
void dfs(Graph &g, int i, char *c) {
  c[i] = grey;  // i を訪問し始める
  ArrayStack<int> edges;
  g.outEdges(i, edges);
  for (int k = 0; k < edges.size(); k++) {
    int j = edges.get(k);
    if (c[j] == white) {
      c[j] = grey;
      dfs(g, j, c);
    }
  }
  c[i] = black; // i を訪問し終えた
}
void dfs(Graph &g, int r) {
  char *c = new char[g.nVertices()];
  dfs(g, r, c);
  delete[] c;
}
```

図 12.5 にこのアルゴリズムの処理の例を示す。

再帰は、深さ優先探索について考えるには最適だが、実装方法としては最善でない。上のコードは、スタックのオーバーフローが原因で、大きなグラフの探索に失敗してしまうことがある。そこで、再帰のスタックを明示的なスタック s で置き換える実装が考えられる。次の実装はこの方法を採用したものである。

Algorithms
```
void dfs2(Graph &g, int r) {
  char *c = new char[g.nVertices()];
```

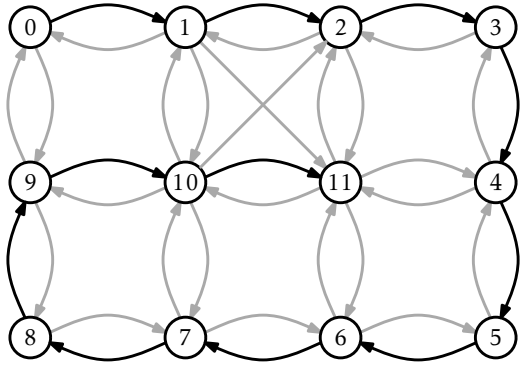


図 12.5: ノード 0 から深さ優先探索を開始した例。各ノードに併記してある数字は、この探索において q に追加される順番。再帰的呼び出しになる辺を黒、それ以外の辺を灰色で描いている

```
  SLList<int> s;
  s.push(r);
  while (s.size() > 0) {
    int i = s.pop();
    if (c[i] == white) {
      c[i] = grey;
      ArrayStack<int> edges;
      g.outEdges(i, edges);
      for (int k = 0; k < edges.size(); k++)
        s.push(edges.get(k));
    }
  }
  delete[] c;
}
```

上のコードでは、次の頂点 i が処理されるときに i の色を grey にし、i の隣接リストに入っていた頂点をスタックに積んでから、そのうちの 1 つを i にしている。

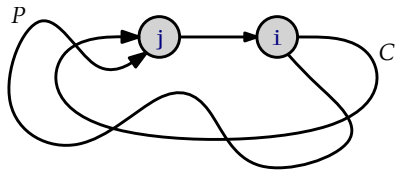


図 12.6: 深さ優先探索アルゴリズムにより、グラフ G の循環を検出できる。ノード i が灰色であるとき、ノード j も灰色である。そのため、深さ優先探索木において i から j への経路 P が存在する。辺 (j,i) が存在することから、P が循環であることもわかる

当然のことだが、dfs(g,r) および dfs2(g,r) の実行時間は bfs(g,r) と同じである。

定理 **12.4.** *AdjacencyLists* で実装された *Graph* g を入力すると、dfs(g,r) および dfs2(g,r) の実行時間はいずれも $O(n+m)$ である。

幅優先探索と同様に、深さ優先探索にも、実行に対応する木が考えられる。頂点 $i \neq r$ の色が white から grey になるのは、ある頂点 i′ を再帰的に処理する中で dfs(g,i,c) を呼び出したからである（dfs2(g,r) の場合、i は i′ をスタックで置き換えた頂点のうちの 1 つである）。i′ を i の親だと考えると、r を根とする木が得られる。この木は、図 12.5 だと、頂点 0 から頂点 11 への経路に相当する。

深さ優先探索には、次のような重要な性質がある。すなわち、i の色が grey のときに i から他の頂点 j への経路で白い頂点だけを辿るものがある場合、i の色が black になるよりも前に、j の色が grey になってから black になる（これは背理法で証明できる。i から j への任意の経路 P を考えればよい）。

この性質は、例えば循環の検出に役立つ。図 12.6 で、r から到達可能なある循環 C があるとする。C の中で色が grey である最初の頂点を i とし、C において i の前にある頂点を j とする。このとき、上の性質から j の色は grey になり、辺 (j,i) を辿るときにも i の色はまだ grey である。したがって、深さ優先探索木において i から j への経路 P が存在すること、および辺 (j,i) が存在することが、アルゴリズムによりわかる。つまり、P が循環であるとわかる。

12.4 ディスカッションと練習問題

定理 12.3 と定理 12.4 から導かれる幅優先探索と深さ優先探索のアルゴリズムの実行時間は、ある意味で厳しすぎるものだといえる。G における頂点 `i` のうち、`i` から `r` への経路が存在するものの個数を、$\mathtt{n}_\mathtt{r}$ と定義する。また、そのような頂点から出る辺の本数を $\mathtt{m}_\mathtt{r}$ とする。このとき、幅優先探索と深さ優先探索のより正確な実行時間に関して、次の定理が成り立つ（練習問題で扱うアルゴリズムのうちの一部でこの定理が役に立つ）。

定理 **12.5.** `bfs(g,r)`、`dfs(g,r)`、`dfs2(g,r)` の実行時間は、*AdjacencyLists* で実装された *Graph* `g` を入力とすると、いずれも $O(\mathtt{n}_\mathtt{r} + \mathtt{m}_\mathtt{r})$ である。

幅優先探索は、Moore と Lee によって独立に考案されたようである [52, 49]。前者では迷路の探索において、後者では回路における経路に関する文脈において発見された。

グラフの隣接リストによる表現は、それまで一般的であった隣接行列による表現の代替として、Hopcroft と Tarjan により提案された [40]。隣接リストによる表現は、深さ優先探索のほか、Hopcroft-Tarjan の平面性テストのアルゴリズムにおいても重要な役割を果たす。これは、辺が交差しないようにグラフを平面に描けるかどうかを $O(\mathtt{n})$ の時間で調べるアルゴリズムである [41]。

以下の練習問題において、無向グラフとは、辺 $(\mathtt{i},\mathtt{j})$ が存在するとき、またそのときに限り、辺 $(\mathtt{j},\mathtt{i})$ が存在するようなグラフであるとする。

問 **12.1.** 図 12.7 のグラフの隣接リストによる表現および隣接行列による表現を書け。

問 **12.2.** グラフ G の**接続行列**（incidence matrix）とは、$\mathtt{n}\times\mathtt{m}$ 行列 A であって、次のように定義されるものである。

$$A_{i,j} = \begin{cases} -1 & \text{頂点 } i \text{ が辺 } j \text{ の始点であるとき} \\ +1 & \text{頂点 } i \text{ が辺 } j \text{ の終点であるとき} \\ 0 & \text{それ以外のとき} \end{cases}$$

1. 図 12.7 のグラフの接続行列を書け。
2. グラフの隣接行列による表現を設計、実装せよ。また、空間使用量を解析し、`addEdge(i,j)`、`removeEdge(i,j)`、`hasEdge(i,j)`、`inEdges(i)`、`outEdges(i)` の実行時間を求めよ。

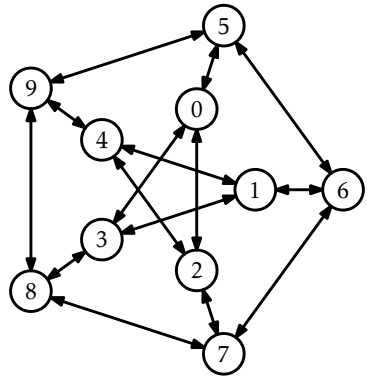


図 12.7: 例題用のグラフ

問 **12.3.** 図 12.7 のグラフ G について、`bfs(G,0)` および `dfs(G,0)` を実行する様子を図示せよ。

問 **12.4.** G を無向グラフとする。G が**連結**（connected）であるとは、任意の相異なる頂点の組 $(\mathtt{i},\mathtt{j})$ について、$\mathtt{i}$ から $\mathtt{j}$ への辺があることをいう（なお、このとき G は無向グラフなので、$\mathtt{j}$ から $\mathtt{i}$ にも辺がある）。G が連結かどうかを $O(\mathtt{n}+\mathtt{m})$ の時間で確認する方法を示せ。

問 **12.5.** G を無向グラフとする。G の**連結成分ラベル付け**（connected components labelling）とは、それぞれが連結な部分グラフになるような G の分割方法のうち、分割後の集合が極大となるように G を分割することである。これを $O(\mathtt{n}+\mathtt{m})$ の時間で計算する方法を示せ。

問 **12.6.** G を無向グラフとする。G の**全域森**（spanning forest）は木の集まりであって、各木の辺が G の辺であり、G のすべての頂点をある木に含むようなものである。これを $O(\mathtt{n}+\mathtt{m})$ の時間で計算する方法を示せ。

問 **12.7.** グラフ G が**強連結**（strongly-connected）であるとは、G の任意の頂点の組 $(\mathtt{i},\mathtt{j})$ について、$\mathtt{i}$ から $\mathtt{j}$ への経路が存在することをいう。これを $O(\mathtt{n}+\mathtt{m})$ の時間で確認する方法を示せ。

問 **12.8.** グラフ $G=(V,E)$ と、特別な頂点 $\mathtt{r}\in V$ があるとき、$\mathtt{r}$ から全頂点 $\mathtt{i}\in V$ への最短経路の長さを計算する方法を示せ。

問 **12.9.** `dfs(g,r)` が `dfs2(g,r)` と異なる順番で頂点を訪問する単純な例を与

えよ。また、`dfs(g, r)` と常に同じ順番で頂点を訪問する `dfs2(g, r)` を実装せよ（ヒント：`r` から 2 つ以上の辺が出ているグラフをいくつか作り、それぞれのアルゴリズムがどう動くか考えてみるといいだろう）。

問 **12.10.** グラフ G の universal sink とは、$n-1$ 個の辺の行き先になっており、かつそこから辺が出ていない頂点である [*1]。`AdjacencyMatrix` で表現されるグラフ G が universal sink を持つかどうかを判定するアルゴリズムを設計、実装せよ。ただし、実行時間は $O(n)$ でなければならない。

[*1] universal sink `v` を**セレブリティ**（celebrity）と呼ぶこともある。部屋の中の全員が `v` のことを知っているが、`v` は部屋の中の人が誰だかまったく知らないため、有名人（セレブリティ）のような状態になっているからである。

第 13

整数を扱うデータ構造

この章では再び SSet の実装を扱う。ただし、w ビットの整数の集まりを表すことに特化した SSet を紹介する。すなわち、$x \in \{0,\ldots,2^w-1\}$ と仮定して、add(x)、remove(x)、find(x) を実装する。整数のデータや整数のキーを扱う状況が多いことは想像に難くないだろう。

この章で説明するデータ構造は 3 つある。1 つめは BinaryTrie というデータ構造で、SSet の 3 つの操作をいずれも $O(w)$ の時間で実行できる。この実行時間は、それほど驚くようなものでもないだろう。$\{0,\ldots,2^w-1\}$ の部分集合の大きさ n は $n \le 2^w$ であり、$\log n \le w$ が成り立つからだ。この本でこれまでに解説した SSet の実装では、各操作の実行時間は $O(\log n)$ であった。すなわち、いずれも BinaryTrie と同じくらいは高速であったということである。

2 つめは XFastTrie というデータ構造で、BinaryTrie の検索をハッシュ法を使って高速化したものである。この高速化により、find(x) の実行時間が $O(\log w)$ になる。しかし、XFastTrie における add(x) および remove(x) の実行時間は依然として $O(w)$ であり、空間使用量は $O(n \cdot w)$ である。

3 つめは YFastTrie というデータ構造だ。このデータ構造では、およそ w 個ごとに、1 つの要素だけを XFastTrie に格納し、それ以外の要素を通常の SSet に格納する。この工夫により、add(x) および remove(x) の実行時間は $O(\log w)$ になり、空間使用量は $O(n)$ に抑えられる。

この章の実装には、整数と対応付けが可能な任意の型のデータを格納できる。サンプルコードでは、x に対応する整数を ix で表し、x を ix に変換する関数を intValue(x) とする。また、簡単のため、文中では x を整数として扱う。

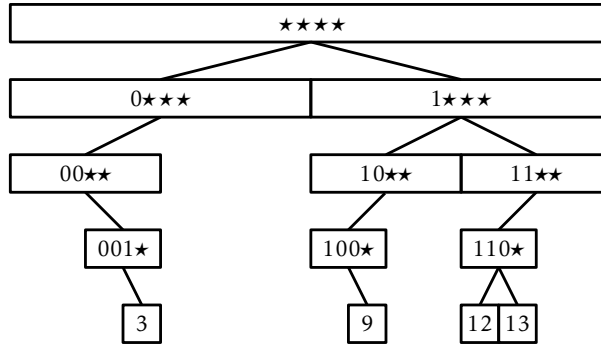


図 13.1: 二分トライ木では、整数を根から葉への経路として符号化する

13.1 BinaryTrie：二分トライ木

BinaryTrie[*1] は、w ビット整数の集合を二分木で符号化したものである。BinaryTrie の葉の深さはいずれも w であり、根から葉への経路が、ある整数を表す。整数 x を表す経路では、深さ i において、整数 x の上から i 番めのビットが 0 なら左に向かい、1 なら右に向かう。図 13.1 に、w = 4 の場合の例を示す。この例では、整数 3（0011）、9（1001）、12（1100）、13（1101）が二分トライ木に格納されている。

二分トライ木に格納されている x を探し出すときの探索経路は、x の二進表記で決まる。そこで、ノード u の子を指すポインタ u.child[0] および u.child[1] に、それぞれ left および right という名前を付けておくと便利である。葉には子がないので、葉ではこのポインタを別の用途に使う。具体的には、このポインタを使って、葉の双方向連結リストを作る。すなわち、二分トライ木の葉では u.child[0] がリストにおける u の直前のノード（prev）を指し、u.child[1] はリストにおける u の直後のノード（next）を指す。さらに、特別なノード dummy により、先頭のノードの前のノード、および、末尾のノードの後のノードを指すことにする（3.2 節を参照）。サンプルコードでは、u.child[0]、u.left、u.prev がいずれもノード u の同じフィールドを参照している。u.child[1]、u.right、u.next も同様である。

[*1] 訳注：Trie はトライと読む。本来は Re”trie”val に由来する造語で、ツリーに近い発音だったようだが、tree と区別するためにトライと発音するようである。

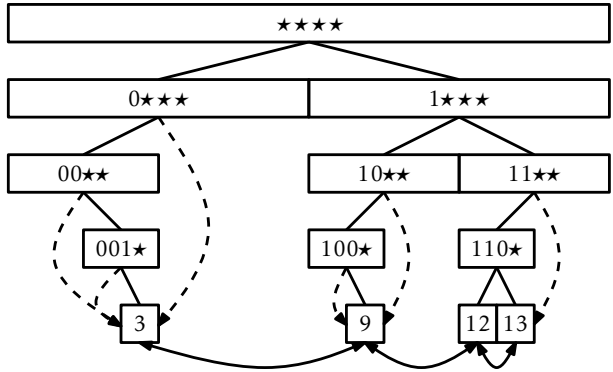


図 13.2: 二分トライ木における jump ポインタを破線の矢印で、リストのリンクを実線の矢印で表す

各ノード u には、さらに u.jump というポインタも持たせる。u.jump は、u が左の子を持たないときは、u を根とする部分木における最小の葉を指す。u が右の子を持たないときは、u を根とする部分木における最大の葉を指す。BinaryTrie の jump ポインタと、葉の双方向連結リストの例を、図 13.2 に示す。

BinaryTrie における find(x) では、単純に x の探索経路を辿ればよい。葉に辿り着ければ、x が存在するとわかる。進みたい方向に子がなく、それ以上進めないノード u に辿り着いたときは、u.jump を辿る。そうすると、x より大きい最小の葉、または、x より小さい最大の葉が見つかる。どちらになるかは、u の左右どちらに子がいないかに応じて決まる。u が左の子を持たないなら、欲しいノードを見つけたことになる [*2]。u が右の子を持たないなら、連結リストを辿ることで欲しいノードが見つかる。図 13.3 に、この 2 つの場合を示す。

BinaryTrie

```
T find(T x) {
  int i, c = 0;
  unsigned ix = intValue(x);
  Node *u = &r;
```

[*2] 訳注：第 1 章で定義したように、SSet の find(x) は、x 以上の最小の要素を見つける操作である。

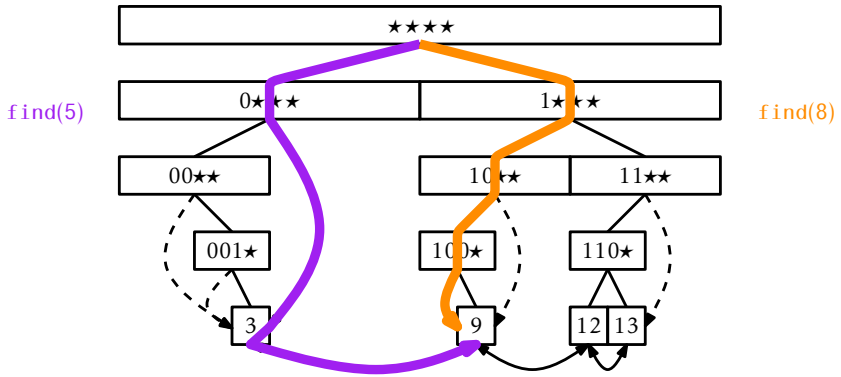


図 13.3: find(5) および find(8) が辿る経路

```
  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;  // 見つけた
  u = (c == 0) ? u->jump : u->jump->next;
  return u == &dummy ? null : u->x;
}
```

find(x) の実行時間において支配的なのは、根から葉への経路を辿る処理であり、これにかかる時間は $O(w)$ である。

BinaryTrie における add(x) も単純だが、手順はかなり多い。

1. x の探索経路を辿り、それ以上進めないノード u を見つける
2. u から x を含む葉への、探索経路に足りない部分を作る
3. x を含むノード u′ を葉の連結リストに追加する（u の jump ポインタを使うことで、連結リストにおける u′ の直前のノード pred が得られる）
4. これまで辿ってきた経路を戻りながら、x を指すべき jump ポインタを更新する

図 13.4 に要素を追加する様子を示す。

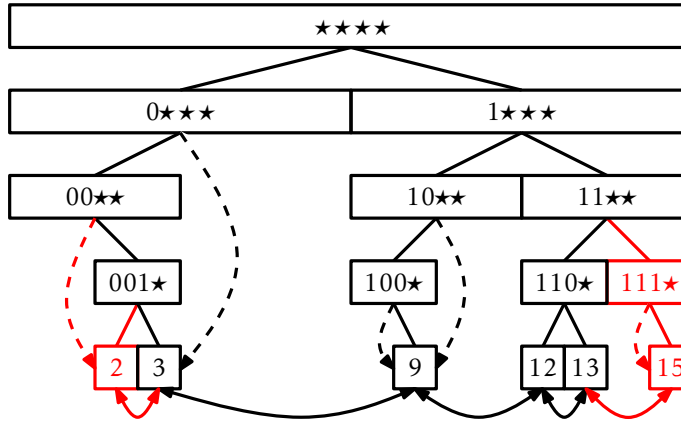


図 13.4: 図 13.2 の BinaryTrie に、値 2、値 15 を追加する

BinaryTrie

```
bool add(T x) {
  int i, c = 0;
  unsigned ix = intValue(x);
  Node *u = &r;
  // 1. 木の端に着くまで ix を探す
  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; // x はすでに入っているので中止する
  Node *pred = (c == right) ? u->jump : u->jump->left;
  u->jump = NULL;  // u は 2 つの子を持つようになる
  // 2. ix への経路を追加する
  for (; i < w; i++) {
    c = (ix >> (w-i-1)) & 1;
    u->child[c] = new Node();
```

```
    u->child[c]->parent = u;
    u = u->child[c];
  }
  u->x = x;
  // 3. u を連結リストに追加する
  u->prev = pred;
  u->next = pred->next;;
  u->prev->next = u;
  u->next->prev = u;
  // 4. 上に戻りながら jump ポインタを更新する
  Node *v = u->parent;
  while (v != NULL) {
    if ((v->left == NULL
        && (v->jump == NULL || intValue(v->jump->x) > ix))
    || (v->right == NULL
        && (v->jump == NULL || intValue(v->jump->x) < ix)))
      v->jump = u;
    v = v->parent;
  }
  n++;
  return true;
}
```

このメソッドは、まず x の探索経路を辿り、それから根の方向に向かって戻る。各ステップは定数時間で実行できるので、add(x) の実行時間は $O(w)$ である。

remove(x) は、add(x) の処理を取り消す。add(x) と同様に手順は多い。

1. x の探索経路を辿り、x を含む葉 u を見つける
2. u を双方向連結リストから削除する
3. u を削除し、x の探索経路に含まれない子を持つノード v が見つかるまで x の探索経路を逆に辿りながら、その過程で訪問したノードを削除する

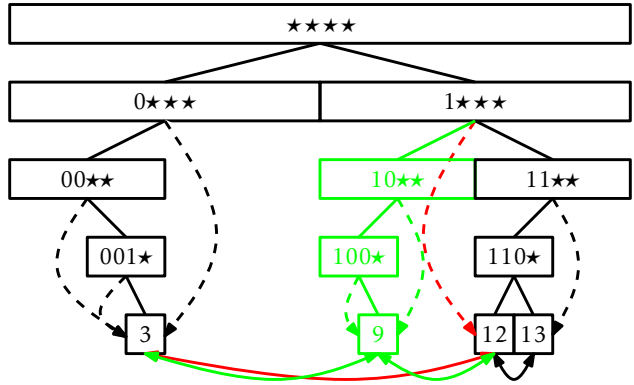


図 13.5: 図 13.2 の BinaryTrie から値 9 を削除する

4. v から根まで辿りながら、u を指していた jump があれば更新する

図 13.5 に削除の様子を示す。

BinaryTrie

```
bool remove(T x) {
  // 1. x を含む葉 u を見つける
  int i = 0, c;
  unsigned ix = intValue(x);
  Node *u = &r;
  for (i = 0; i < w; i++) {
    c = (ix >> (w-i-1)) & 1;
    if (u->child[c] == NULL) return false;
    u = u->child[c];
  }
  // 2. u を連結リストから削除する
  u->prev->next = u->next;
  u->next->prev = u->prev;
  Node *v = u;
  // 3. u を根から u への経路上のノードから削除する
  for (i = w-1; i >= 0; i--) {
```

```
    c = (ix >> (w-i-1)) & 1;
    v = v->parent;
    delete v->child[c];
    v->child[c] = NULL;
    if (v->child[1-c] != NULL) break;
  }
  // 4. jump ポインタを更新する
  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;
}
```

定理 **13.1.** *BinaryTrie* は、w ビット整数を格納するための *SSet* インターフェースの実装である。*BinaryTrie* における add(x)、remove(x)、find(x) の実行時間はいずれも $O(w)$ である。n 個の要素を格納する *BinaryTrie* の空間使用量は $O(n \cdot w)$ である。

13.2 XFastTrie：$O(\log(\log n))$ 時間での検索

BinaryTrie の性能は、あまりパッとしない。データ構造に格納できる要素数 n は最大でも 2^w なので、$\log n \le w$ である。少なくとも、この本で説明してきた比較に基づく SSet の実装は、いずれも BinaryTrie と同じくらい効率的であった。それに、整数しか格納できないという BinaryTrie のような制限

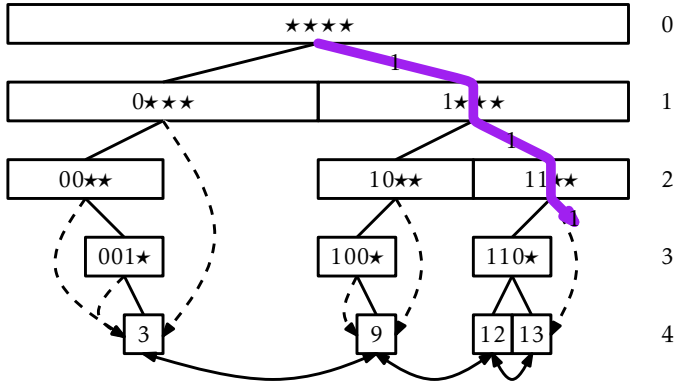


図 13.6: ラベル 111★ を持つノードは存在しないので、14（1110）の探索経路はラベル 11★★ を持つノードで終了する

もなかった。

これから説明する XFastTrie は、各深さに 1 つずつ、合計で w+1 個のハッシュテーブルを BinaryTrie に付け足しただけのデータ構造である。XFastTrie では、これらのハッシュテーブルを使うことで、find(x) の実行時間を $O(\log w)$ に改善できる。BinaryTrie における find(x) は、x への探索経路を辿り、左右の進みたい方向に子を持たないノード u を見つけた時点でほぼ完了であった。その時点で、u.jump を利用して葉 v にジャンプし、v もしくは葉のリストにおける v の直前のノードのどちらかを返すだけである。XFastTrie では、深さに関する二分探索によりノード u を見つけることで、この探索処理を高速に行う。

二分探索では、探しているノード u について、ある深さ i より上にあるのか、i またはその下にあるのかを判定する必要がある。これは、x の二進表記における上位 i ビットを見ればわかる。このビット列によって、根から深さ i までの x の探索経路が決まる。例えば、図 13.6 を見てほしい。14（二進表記では 1110）の探索経路における最後のノード u は、深さ 2 のところにある、11★★ というラベルが付いたノードである。これは、深さ 3 の位置に 111★ というラベルが付いたノードがないためである。このようにして、深さ i のノードすべてに、i ビットの整数でラベルを付けられる。すると、探している u が深さ i、またはそれより下にあるのは、深さ i に x の上位 i ビットと一致するラベルを持つノードがあるとき、かつそのときに限られる。

XFastTrie では、$i \in \{0,\ldots,w\}$ について、深さ i のすべてのノードを USet

t[i] に格納する。USet はハッシュテーブル（5 章参照）で実装する。USet を使うことで、深さ i に x の上位 i ビットと一致するラベルを持つノードがあるかどうか、定数オーダーの期待実行時間で判定できる。具体的には、そのようなノードを t[i].find(x >> (w − i)) のようにして探し出せる。

u を見つけ出すのに二分探索が使えるのは、ハッシュテーブル t[0],...,t[w] のおかげである。最初の段階でわかっているのは、$0 \le i < w+1$ を満たす深さ i に u があることである。そこで、まず l = 0 および h = w + 1 とし、$i = \lfloor (l+h)/2 \rfloor$ についてハッシュテーブル t[i] を繰り返し検索する。そして、t[i] が x の上位 i ビットと一致するラベルを持つノードを含むとき（したがって u が深さ i、またはそれよりも下にあるとき）、l = i とする。そうでない、つまり u が深さ i よりも上にあるときは、h = i とする。$h - l \le 1$ になった段階で、この処理を終える。このとき、u は深さ l にある。あとは、u.jump および葉の双方向連結リストを使って、find(x) の処理を完了する。

XFastTrie

```
T find(T x) {
  int l = 0, h = w+1;
  unsigned ix = intValue(x);
  Node *v, *u = &r;
  while (h-l > 1) {
    int i = (l+h)/2;
    XPair<Node> p(ix >> (w-i));
    if ((v = t[i].find(p).u) == NULL) {
      h = i;
    } else {
      u = v;
      l = i;
    }
  }
  if (l == w) return u->x;
  Node *pred = (((ix >> (w-l-1)) & 1) == 1)
               ? u->jump : u->jump->prev;
  return (pred->next == &dummy) ? nullt : pred->next->x;
```

```
}
```

上に示したメソッドでは、`while` ループにおける各繰り返しにおいて、`h-l` が約半分になる。よって、このループを $O(\log w)$ 回繰り返したところで `u` が見つかる。繰り返しのたびに、毎回一定量の作業を実行し、USet の `find(x)` を 1 回だけ呼ぶので、USet における検索の実行時間の期待値は定数オーダーである。残りの処理の実行時間も定数オーダーなので、XFastTrie における `find(x)` の実行時間の期待値は $O(\log w)$ である。

XFastTrie における `add(x)` および `remove(x)` は、BinaryTrie における各操作とほとんど同じである。ハッシュテーブル `t[0],...,t[w]` の更新処理を追加すればよい。`add(x)` の実行中に、深さ `i` でノードが作られたら、このノードを `t[i]` に加えるようにする。`remove(x)` の実行中に、深さ `i` でノードが削除されるなら、このノードを `t[i]` から削除するようにする。ハッシュテーブルにおける追加と削除の期待実行時間は定数オーダーなので、この修正によって `add(x)` と `remove(x)` の期待実行時間は定数オーダーしか増えない。`add(x)` および `remove(x)` のコードは、BinaryTrie のときに示した長いコードとほぼ同じなので、ここには掲載しない。

次の定理に XFastTrie の性能をまとめる。

定理 **13.2.** *XFastTrie* は、`w` ビット整数を格納するための *SSet* インターフェースの実装である。*XFastTrie* がサポートするのは次の操作である。

- `add(x)` および `remove(x)` の実行時間の期待値は $O(w)$ である
- `find(x)` の実行時間の期待値は $O(\log w)$ である

`n` 個の要素を格納する *XFastTrie* の空間使用量は $O(n \cdot w)$ である。

13.3 YFastTrie：$O(\log(\log n))$ 時間の SSet

XFastTrie における検索の実行時間は、BinaryTrie に比べて指数的に速くなった。しかし、`add(x)` と `remove(x)` の実行時間は依然としてそれほど速くない。そのうえ、空間使用量は $O(n \cdot w)$ であり、この本で紹介した他の SSet の実装の $O(n)$ と比べて大きい。この 2 つの問題は互いに関連している。具体的には、`n` 回の `add(x)` によって大きさ $n \cdot w$ の構造を作れば、`add(x)` の 1 回あたりの実行時間と空間使用量は `w` 程度のオーダーになる。

この節で紹介する YFastTrie は、XFastTrie の実行時間と空間使用量を

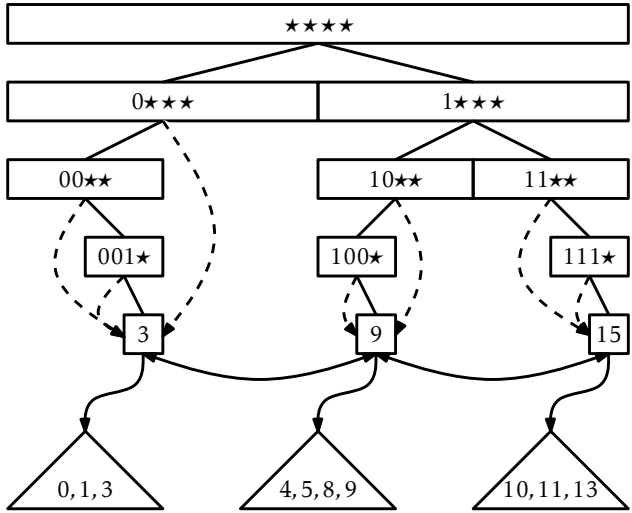


図 13.7: 0, 1, 3, 4, 6, 8, 9, 10, 11, 13 を含む YFastTrie

改善するものだ。YFastTrie では XFastTrie（以降では xft とする）を使うが、この xft には $O(n/w)$ 個の値しか入れない。こうすると、xft の空間使用量は $O(n)$ になる。さらに、add(x) と remove(x) を w 回実行するときは、そのうち 1 回のみを xft に対して実行する。このようにすることで、xft における add(x) と remove(x) の平均実行時間は定数になる。

xft には n/w 個の要素しか格納しないのに、残りの $n(1-1/w)$ 個の要素はどこへ行ってしまうのか。これらの要素は、**二次構造**へと移動する。ここでは、二次構造として、Treap（7.2 節参照）を拡張したデータ構造を使う。二次構造は全部でおよそ n/w 個あり、1 つの二次構造には平均して $O(w)$ 個の要素を格納することにする。Treap は SSet の操作を対数時間でサポートするので、各操作の実行時間は、当初の目論見通り $O(\log w)$ である。

より具体的に言うと、YFastTrie では、確率 $1/w$ で選り抜いたデータを格納するための XFastTrie を保持しておく。この XFastTrie（xft）には、都合により、常に 2^w-1 という値を含めておく。また、xft に含める要素は、$x_0 < x_1 < \cdots < x_{k-1}$ とする。各要素 x_i には Treap が対応しており（以降では x_i に対応する Treap を t_i とする）、これに $x_{i-1}+1,\ldots,x_i$ の範囲の値をすべて格納する。図 13.7 にこの様子を示す。

YFastTrie における find(x) は実に簡単である。x を xft から検索すれ

ば、t_i に対応する x_i が見つかる。そこで t_i の find(x) メソッドを使えば、求めていることが実現できる。このメソッド全体は 1 行で書ける。

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

はじめの xft に対する find(x) にかかる時間は $O(\log w)$ である。その次の Treap に対する find(x) にかかる時間は $O(\log r)$ である。ここで、r は Treap の大きさである。Treap の大きさの期待値は $O(w)$ なので（この節の後半で示す）、結局、この操作の実行時間は $O(\log w)$ である [*3]。

YFastTrie への要素の追加も、ほとんどの場合は実に単純だ。add(x) メソッドでは、x を挿入すべき Treap を特定するために、xft.find(x) を呼ぶ。適切な Treap が見つかったら、それを t とし、t.add(x) を呼ぶことで x を t に追加する。ここで、確率 1/w で表が、確率 1 − 1/w で裏が出るような偏りのあるコインを投げる。もし表が出れば、x を xft に追加する。

ここから少し複雑になる。x を xft に追加するとき、t を 2 つの Treap に分割しなければならない。それらを t1 および t′ としよう。t1 には、x 以下の値をすべて含める。t′ は、それ以外の値を含むように t を更新したものである。最後に、(x,t1) という組を xft に追加する。図 13.8 に例を示す。

YFastTrie
```
bool add(T x) {
  unsigned ix = intValue(x);
  Treap1<T> *t = xft.find(YPair<T>(ix)).t;
  if (t->add(x)) {
    n++;
    if (rand() % w == 0) {
      Treap1<T> *t1 = (Treap1<T>*)t->split(x);
      xft.add(YPair<T>(ix, t1));
    }
    return true;
  }
```

[*3] $E[r] = w$ ならば $E[\log r] \le \log w$ という Jensen の不等式の応用である。

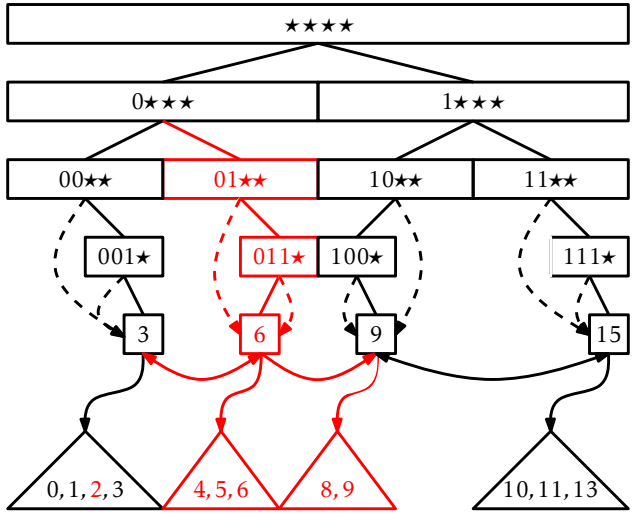


図 13.8: YFastTrie に値 2、値 6 を追加する。6 を追加するときにコイン投げで表が出たので、6 は xft に追加され、4,5,6,8,9 を含む Treap は分割される

```
  return false;
}
```

x を t に追加するのにかかる時間は $O(\log w)$ である。問 7.12 では、t を分割して t1 と t' を得る処理の実行時間の期待値が $O(\log w)$ であることを示した。(x,t1) を xft に追加する処理の実行時間は $O(w)$ だが、これが起こる確率は $1/w$ である。以上より、add(x) の実行時間の期待値は次のようになる。

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

remove(x) は、add(x) による操作を取り消す。まず、xft.find(x) の結果を含んでいる xft から、葉 u を見つける。u から x を含んでいる Treap（t）を得て、その Treap から x を削除する。もし x が xft にも含まれていれば、そして x が $2^w - 1$ でなければ、x を xft から削除し、x を含む Treap の要素をすべて別の Treap（t2）に追加する。ここで、t2 は、連結リストにおける u の直後のノードに対応する Treap である。図 13.9 にこの様子を示す。

YFastTrie

```
bool remove(T x) {
```

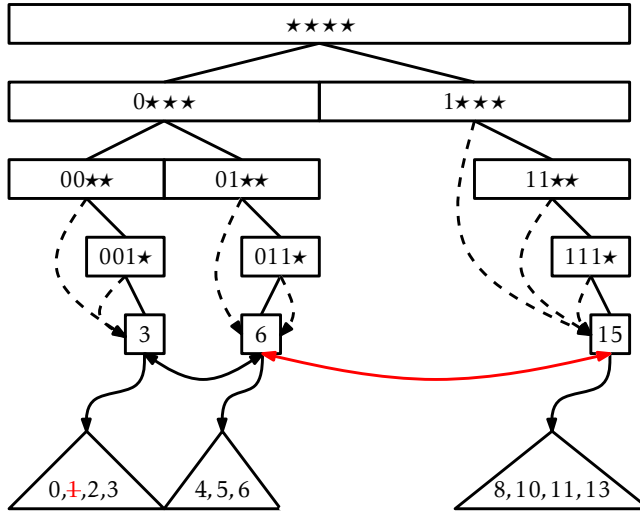


図 13.9: 図 13.8 の YFastTrie から値 1、値 9 を削除する

```
  unsigned ix = intValue(x);
  XFastTrieNode1<YPair<T> > *u = xft.findNode(ix);
  bool ret = u->x.t->remove(x);
  if (ret) n--;
  if (u->x.ix == ix && ix != UINT_MAX) {
    Treap1<T> *t2 = u->child[1]->x.t;
    t2->absorb(*u->x.t);
    xft.remove(u->x);
  }
  return ret;
}
```

xft からノード u を見つけるのに必要な時間の期待値は $O(\log w)$ である。t から x を削除するのにかかる時間の期待値も $O(\log w)$ である。繰り返しになるが、問 7.12 では、t を分割して t1 と t′ を得る処理の実行時間の期待値も $O(\log w)$ であることを示した。xft から x を削除する必要があるときは、この処理に $O(w)$ の時間がかかるが、xft に x が含まれる確率は $1/w$ である。よっ

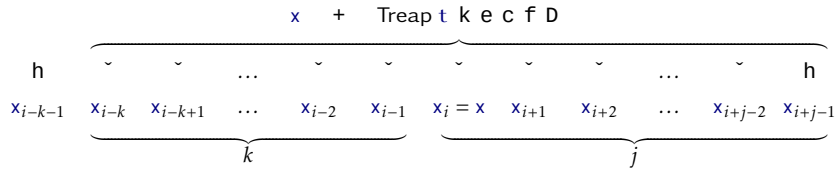


図 13.10: x を含む Treap t の要素数は連続した 2 回のコイン投げ試行により決まる

て、YFastTrie における削除処理の実行時間の期待値は $O(\log w)$ である。

ここまでの説明では、このデータ構造における各 Treap の大きさの説明を後回しにしていた。この章を終える前に、必要な定理を証明しておく。まずは次の補題を示す。

補題 **13.1.** *YFastTrie* に格納する整数を x とし、x を含む *Treap* t の要素数を n_x とする。このとき、$E[n_x] \le 2w - 1$ が成り立つ。

証明. 図 13.10 を見てほしい。いま、YFastTrie の各要素を、$x_1 < x_2 < \cdots < x_i = x < x_{i+1} < \cdots < x_n$ とする。Treap t に含まれる x 以上の要素を $x_i, x_{i+1}, \ldots, x_{i+j-1}$ とする。このうち、add(x) の際の偏りのあるコイン投げで表が出たのは、x_{i+j-1} だけである。つまり、$E[j]$ は、偏りのあるコイン投げを表が出るまで繰り返すときのコイン投げ回数の期待値と等しい [*4]。コイン投げは独立な試行であり、表が出る確率は $1/w$ である。そのため $E[j] \le w$ である（$w = 2$ の場合の解析については補題 4.2 を参照してほしい）。

同様に、t の要素で x より小さいもの $x_{i-1}, \ldots, x_{i-k}$ について、これらに対応する k 回のコイン投げはいずれも裏であり、x_{i-k-1} のコイン投げは表である。先の段落と同じく、偏りのあるコイン投げを表が出るまで繰り返すときに裏が出る回数を考えると、$E[k] \le w - 1$ だとわかる。

まとめると、$n_x = j + k$ より、以下のようになる。

$$E[n_x] = E[j + k] = E[j] + E[k] \le 2w - 1 \qquad \square$$

補題 13.1 は、YFastTrie の性能をまとめた次の定理を示す最後のピースである。

定理 **13.3.** *YFastTrie* は、w ビット整数を格納するための *SSet* インター

[*4] この解析では、j が $n - i + 1$ を超えることがないことを無視している。しかし、その場合は $E[j]$ が小さくなるので、上界に関する性質は依然として成り立つ。

フェースの実装である。*YFastTrie* は、add(x)、remove(x)、find(x) をサポートし、いずれの実行時間の期待値も $O(\log w)$ である。n 個の要素を格納する *YFastTrie* の空間使用量は $O(n+w)$ である。

空間使用量に w が影響するのは、xft が常に値 2^w-1 を格納していることによる。実装を修正して、この値を格納せずに済ませることも可能だ（ただし、いくつか場合分けをコードに追加する必要がある）。この場合、上の定理における空間使用量は $O(n)$ になる。

13.4 ディスカッションと練習問題

add(x)、remove(x)、find(x) の実行時間がいずれも $O(\log w)$ であるデータ構造として初めて提案されたのは、van Emde Boas によるもので、van Emde Boas 木（または stratified 木）という名で知られている [72]。オリジナルの van Emde Boas 木の大きさは 2^w だったので、大きな整数を扱うには非実用的であった。

XFastTrie と YFastTrie は、Willard によって提案された [75]。XFastTrie と van Emde Boas 木には密接な関係がある。例えば、XFastTrie におけるハッシュテーブルは van Emde Boas 木の配列を置き換えたものである。つまり、ハッシュテーブル t[i] に要素を格納する代わりに、van Emde Boas 木では長さ 2^i の配列に要素を格納する。

整数を格納するためのデータ構造としては、これらのほかに、Fredman と Willard による fusion 木がある [32]。このデータ構造は、n 個の w ビット整数を $O(n)$ の領域に格納でき、find(x) を $O((\log n)/(\log w))$ の時間で実行できる。$\log w > \sqrt{\log n}$ ならば fusion 木を、$\log w \le \sqrt{\log n}$ なら YFastTrie を使えば、空間使用量が $O(n)$ で find(x) にかかる時間が $O(\sqrt{\log n})$ であるようなデータ構造が得られる。近年、Pătraşcu and Thorup が示した下界によると、$O(n)$ だけの領域を使うデータ構造としては最適なものになる [57]。

問 **13.1.** 単純化された BinaryTrie を設計、実装せよ。これは連結リストや jump ポインタを持たないものとする。ただし、find(x) の実行時間は依然として $O(w)$ である必要がある。

問 **13.2.** 単純化された XFastTrie を設計、実装せよ。これは二分トライ木を使わないものとする。代わりに、この実装では、すべてを双方向連結リストと $w+1$ 個のハッシュテーブルを使って格納する。

問 **13.3.** `BinaryTrie` は、長さ `w` のビット列を根から葉への経路として表現するデータ構造であると考えられる。この発想を、可変長の文字列を格納する `SSet` の実装に拡張し、`add(s)`、`remove(s)`、`find(s)` をいずれも `s` の長さに比例する時間で実行できるデータ構造を実装せよ。
ヒント：データ構造の各ノードは、文字の値によってインデックスを計算するハッシュテーブルを格納する。

問 **13.4.** 整数 $x \in \{0,\ldots 2^w-1\}$ について、`x` と `find(x)` の返り値との差を $d(x)$ と定義する（`find(x)` が `null` を返すときは、$d(x)$ は 2^w であるとする）。例えば、`find(23)` が 43 を返すとき、$d(23) = 20$ である。

1. `XFastTrie` における `find(x)` を修正し、実行時間の期待値が $O(1+\log d(x))$ であるものを設計、実装せよ。ヒント：ハッシュテーブル $t[w]$ には、$d(x) = 0$ であるような値 `x` がすべて格納されているので、処理を開始するのに適切な位置になるだろう。
2. `XFastTrie` における `find(x)` を修正し、実行時間の期待値が $O(1+\log\log d(x))$ であるものを設計、実装せよ。

第 **14**

外部メモリの探索

この本を通じて、計算のモデルとしては、1.4 節で定義した `w` ビットのワード RAM モデルを使ってきた。このモデルでは、データ構造内のすべてのデータを格納できるくらいコンピュータの RAM が大きいことを暗に仮定している。この仮定は、場合によっては成り立たない。大きすぎてコンピュータのメモリには収まりきらないデータもある。そのような場合は、ハードディスクドライブ（HDD）、ソリッドステートドライブ（SSD）、あるいはネットワーク越しのサーバーなど、外部ストレージにデータを保持するしかない。

外部ストレージへのデータアクセスは非常に遅い。この本を書くのに使っているコンピュータでは、ハードディスクへの平均アクセス時間は 19ms である。SSD でも 0.3ms かかる。これに対し、RAM への平均アクセス時間は 0.000113ms 未満である。RAM へのアクセスは、SSD と比べて 2500 倍、HDD と比べると 160000 倍以上も高速なのである。

RAM のほうが、HDD や SSD よりも、数千倍も高速にランダムアクセスが可能である。この速度そのものには何も特別なことはない。問題は、アクセスにかかる時間だけですべてを説明できるわけではないことにある。HDD や SSD 上のバイトにアクセスするとき、実際には、**ブロック**（block）単位で読み出しを行う。コンピュータに接続されている各ドライブのブロックの大きさは 4,096 である [*1]。つまり、1 バイトの読み出しをするたびに、ドライブは 4,096 バイトを返してくる。このことを踏まえてデータ構造を設計すれば、どのような操作を完了する場合であれ、HDD や SDD から 4,096 バイトのデータを引き出してこられるだろう。

これが**外部メモリモデル**（external memory model）の背景となる発想だ。

[*1] 訳注：異なる大きさが使われることもあるが、ここでは簡単のため 4,096 で統一している。

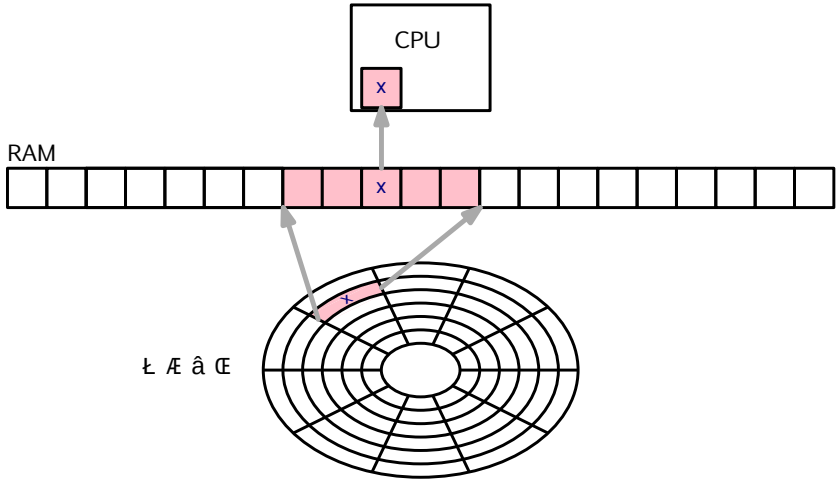


図 14.1: 外部メモリモデルでは、外部メモリに含まれる要素 x にアクセスするために、x を含むブロックをまるごと RAM に読み込む必要がある

図 14.1 に模式図を示す。このモデルでは、コンピュータはすべてのデータが保存されている大きな外部メモリにアクセスできる。このメモリは**ブロック**に分割されている。各ブロックは B ワードのデータを含む。コンピュータには、計算を実行できる有限の内部メモリもある。内部メモリと外部メモリの間でブロックを転送するには一定の時間がかかる。内部メモリでの計算は**フリー**である。つまり、一切の時間がかからない。奇妙な仮定に感じるかもしれないが、外部メモリへのアクセスが非常に遅いことを強調しているだけである。

本格的な外部メモリモデルでは、内部メモリの大きさも変数として考慮する必要がある。しかし、この章で扱うデータ構造では、大きさが $O(B + \log_B n)$ の内部メモリがあると考えれば十分である。つまり、定数個のブロックと高さ $O(\log_B n)$ のスタックを保持できるだけの内部メモリが必要ということである。必要な内部メモリの大きさを左右するのは、多くの場合、$O(B)$ の項である。例えば、B が比較的小さな値 32 であるとしても、すべての $n \leq 2^{160}$ について $B \geq \log_B n$ が成り立つ。十進表記で書くと、以下を満たす任意の n について $B \geq \log_B n$ が成り立つ。

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

14.1 BlockStore

外部メモリには HDD や SSD などさまざまなデバイスが含まれる。ブロックの大きさはデバイスごとに定義されており、それぞれ独自のシステムコールによってアクセスされる。汎用性がある考え方を伝えるために、この章では解説を単純にしたいので、BlockStore というオブジェクトで外部メモリのデバイスを隠蔽することにする。BlockStore には、ブロックの集まりが格納されている。各ブロックの大きさは B である。各ブロックは整数のインデックスで一意に識別できる。BlockStore がサポートする操作は次のとおり。

1. readBlock(i)：インデックス i で示されるブロックの内容を返す
2. writeBlock(i,b)：インデックス i で示されるブロックに b の内容を書く
3. placeBlock(b)：新規のインデックスを返し、そのインデックスが示すブロックに b の内容を書く
4. freeBlock(i)：インデックス i が示すブロックを開放する。これは、指定したブロックの内容をもう使わず、このブロックに割り当てられていた外部メモリを別の用途に使ってよいことを意味する

B バイトごとのブロックに分割されたディスク上のファイルが BlockStore であると考えるとわかりやすいだろう。readBlock(i) および writeBlock(i,b) は、このファイルに対するバイト列 $iB,\ldots,(i+1)B-1$ の読み書きに相当する。さらに、BlockStore では、利用可能なブロックからなる**フリーリスト**を保持してもよい。フリーリストには、freeBlock(i) により解放されたブロックを追加する。そのうえで placeBlock(b) ではフリーリストのブロックを使い、もし利用可能なフリーリストがなければファイルの末尾に新しいブロックを追加すればよい。

14.2 B 木

この節では、B 木と呼ばれる、二分木を一般化したデータ構造について説明する。B 木は外部メモリモデルにおける効率的なデータ構造である。なお、9.1 節で説明した 2-4 木の自然な一般化として B 木を考えることもできる（B 木において $B=2$ とおくと 2-4 木になる）。

$B \geq 2$ を任意の整数とする。*B* **木**とは、すべての葉が同じ深さにある木であ

り、すべての根でない内部ノード u について、その子の数が B 以上 $2B$ 以下であるようなものである。ノード u の子は、配列 u.children に格納される。根については、子の数に対する条件を緩くして、2 以上 $2B$ 以下とする。

B 木の高さが h のとき、葉の数 ℓ は次の式を満たす。

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

この式の左辺は、根の子が 2 個のみですべての内部ノードが B 個の子を持つときの葉の数に対応する。右辺は、葉以外のノードの子がすべて $2B$ 個であるときの葉の数に対応する。最初の不等式の両辺から対数を取り、項を並べ替えると、次の式が得られる。

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

つまり、B 木の高さは B を底とする葉の数の対数に比例する。

B 木における各ノード u には、キーの配列 u.keys[0],...,u.keys[$2B-1$] を格納する。u が k 個の子を持つ内部ノードのとき、u に格納されるキーの数はちょうど $k-1$ 個であり、それぞれ u.keys[0],...,u.keys[$k-2$] に格納される。u.keys における残りの $2B-k+1$ 個の配列のエントリは null にしておく。u が根でない葉ノードのとき、u は $B-1$ 個以上 $2B-1$ 個以下のキーを持つ。B 木におけるキーは、二分探索木と同様の順序に従う。$k-1$ 個のキーを格納する任意のノード u は次の式を満たす[*2]。

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

u が内部ノードなら、任意の i $\in \{0,\ldots,k-2\}$ について、u.keys[i] は u.children[i] を根とする部分木に格納されるどのキーよりも大きく、u.children[i + 1] を根とする部分木に格納されるどのキーよりも小さい。つまり、厳密な書き方ではないが、次が成り立つ。

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

$B = 2$ である B の例を図 14.2 に示す。

[*2] 訳注：この本では、B 木をキーの重複がない SSet インターフェースを実装するために使うので、等号なしの不等号になる。キーの重複がある multiset の実装時は、u.keys[0] ≤ u.keys[1] ≤ ··· ≤ u.keys[$k-2$]、u.children[i] ≤ u.keys[i] ≤ u.children[i + 1] を満たす。

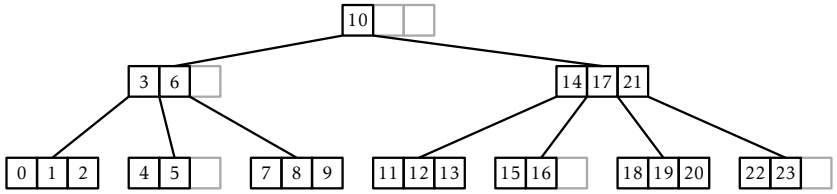


図 14.2: $B = 2$ である B 木

B 木のノードに格納されるデータの大きさは $O(B)$ である。そのため、外部メモリとして使うことを考えると、B 木の B の値は外部メモリのブロックの大きさに合わせて選ぶことになる。そうすれば、外部メモリモデルにおいて B 木の操作にかかる時間は、操作時にアクセス（読み書き）するノードの数に比例する。

例えば、キーが 4 バイト整数であり、ノードのインデックスも 4 バイトであるとする。このとき、$B = 256$ とすれば、各ノードは次式により 4,096 バイトのデータを格納することになる。

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

この章の冒頭で説明したように、ハードディスクや SSD のブロックサイズは 4096 バイトなので、この B はこれらのデバイスに適した値である。

BTree クラスは、B 木の実装である。BTree クラスには、BlockStore オブジェクト `bs` を格納する。この `bs` に、根のインデックス `ri` と、BTree のノードを格納する。他のデータ構造の場合と同様に、整数 `n` はデータ構造の要素数を表す。

BTree

```
int n; // 木に含まれる要素の個数
int ri; // 根のインデックス
BlockStore<Node*> bs;
```

14.2.1 要素の探索

`find(x)` の実装（図 14.3 に示したもの）は、二分探索木における `find(x)` 操作の一般化である。`x` の探索を根から開始し、ノード `u` のキーを利用して、`u` の子のうちのどちらに探索を進めるべきかを決める。

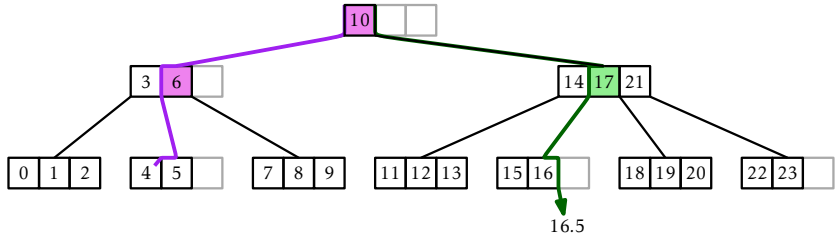


図 14.3: *B* 木における成功する探索（4 を探す）と、失敗する探索（16.5 を探す）の様子。色を付けたノードは探索の途中に値が更新されるものである

具体的には、ノード `u` において、探索している `x` が `u.keys` に格納されているかどうかを確認する。格納されていれば、`x` が見つかったので処理を終了する。格納されていなければ、`u.keys[i]` > `x` を満たす最小の整数 `i` を求め、`u.children[i]` を根とする部分木に進んで探索を続ける。`u.keys` に `x` より大きなキーがないときは、`u` の一番右の子に進んで探索を続ける。二分探索木の場合と同様に、このアルゴリズムでは、`x` より大きなキーのうち最後に訪れたもの `z` を記録しておく。`x` が見つからなかったときは、`x` 以上の最小の値である `z` を返す。

BTree

```
T find(T x) {
  T z = null;
  int ui = ri;
  while (ui >= 0) {
    Node *u = bs.readBlock(ui);
    int i = findIt(u->keys, x);
    if (i < 0) return u->keys[-(i+1)]; // 見つけた
    if (u->keys[i] != null)
      z = u->keys[i];
    ui = u->children[i];
  }
  return z;
}
```

`find(x)` の肝は、`null` で埋められた配列 `a` から `x` を探す、`findIt(a,x)` とい

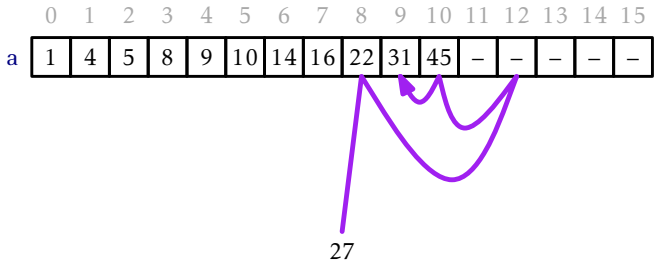


図 14.4: findIt(a,27) を実行する様子

うメソッドである。図 14.4 に示したように、a[0],...,a[$k-1$] はキーが整列された状態であり、a[k],...,a[a.length -1] にはすべて null が入っている。x がこの配列の i 番めの位置に入っているとき、findIt(a,x) は $-i-1$ を返す。そうでないときは、a[i] > x または a[i] = null を満たす最小のインデックス i を返す。

BTree
```
int findIt(array<T> &a, T x) {
  int lo = 0, hi = a.length;
  while (hi != lo) {
    int m = (hi+lo)/2;
    int cmp = a[m] == null ? -1 : compare(x, a[m]);
    if (cmp < 0)
      hi = m;       // 前半を見る
    else if (cmp > 0)
      lo = m+1;     // 後半を見る
    else
      return -m-1; // 見つけた
  }
  return lo;
}
```

findIt(a,x) では二分探索を使う。各ステップで探索空間が半分ずつ減っていくので、$O(\log(\text{a.length}))$ の時間で処理が完了する。この実装では a.length = $2B$ なので、findIt(a,x) の（RAM モデルでの）実行時間は

$O(\log B)$ である。

B 木における `find(x)` の実行時間は、ワード RAM モデル（全命令を数える）でも、外部メモリモデル（アクセスするノードの数だけを数える）でも解析できる。B 木の葉には、少なくとも 1 つのキーが格納されており、ℓ 個の葉を持つ B 木の高さは $O(\log_B \ell)$ なので、`n` 個のキーを格納する B 木の高さは $O(\log_B n)$ である。よって、外部メモリモデルにおける `find(x)` の実行時間は $O(\log_B n)$ である。ワード RAM モデルにおける実行時間を計算するためには、アクセスするすべてのノードについて、`findIt(a,x)` 呼び出しのコストを考えればよい。したがって、この場合の `find(x)` の実行時間は次のようになる。

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

14.2.2 要素の追加

B 木と、6.2 節で説明した `BinarySearchTree` との重要な違いは、B 木のノードには親へのポインタがないことである。なぜ B 木で親へのポインタを保持していないかは、あとで軽く説明する。親へのポインタがないことから、B 木における `add(x)` と `remove(x)` は再帰を使って実装するのが最も簡単である。

他のバランスされた探索木と同様に、`add(x)` では何らかのバランス調整が必要になる。B 木では、ノードの**分割**によってバランスを調整する。以降の説明は図 14.5 を見ながら読んでほしい [*3]。分割は二段階の再帰におよぶのだが、$2B$ 個のキーを含んで $2B+1$ 個の子を持つノード `u` を引数とする操作であると考えると理解しやすいだろう。新たなノード `w` を作り、このノードに `u.children`$[B]$,...,`u.children`$[2B]$ を引き受けさせる。新たなノード `w` には、`u` のキーのうち、大きいほうから B 個（`u.keys`$[B]$,...,`u.keys`$[2B-1]$）も持たせる。この時点で、`u` は、B 個の子と B 個のキーを持っている。余分なキーである `u.keys`$[B-1]$ は、`u` の親に渡す。`u` の親には、`w` も子として引き受けさせる。

分割で操作するノードは 3 つあることに注目してほしい。具体的には、`u`、`u` の親、新たなノード `w` を操作する。B 木で親へのポインタを持たないことが重要なのは、これが理由である。もし親へのポインタがあれば、`w` が引き取る $B+1$ 個の子すべてについて、親へのポインタを `w` へのポインタとして書き換える必要がある。これによって外部メモリへのアクセスが 3 回から $B+4$ 回に

[*3] 訳注： 図 14.5 の ¢ 記号は、後の節で実行時間の解析に使うものなので、ここでは無視してよい。

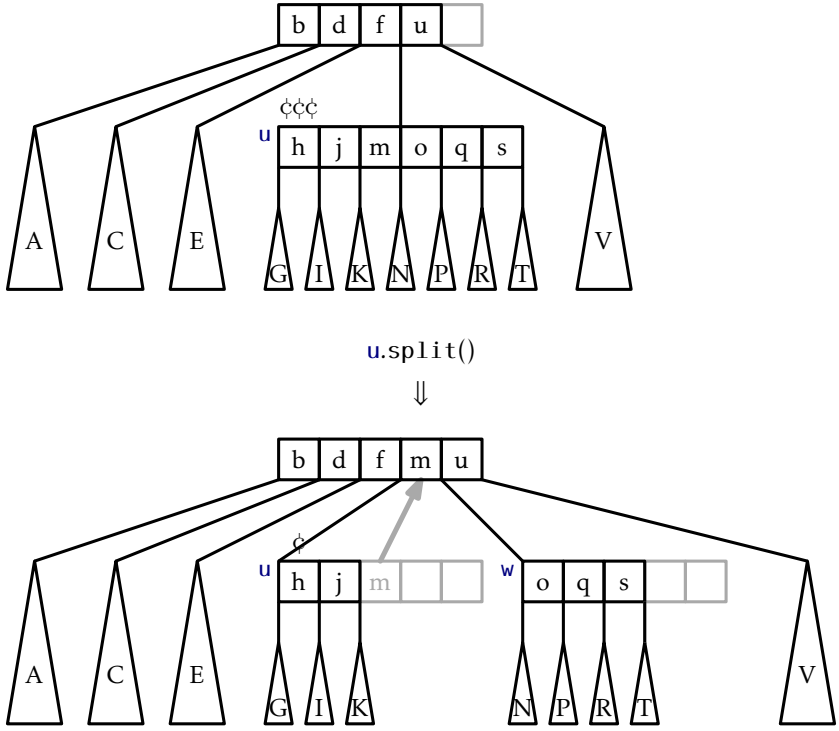


図 14.5: $B = 3$ である B 木における、ノード u の分割。キー u.keys[2] = m は u からその親に移る

増えるので、B が大きいときに B 木が非効率になってしまう。

B 木における add(x) の様子を図 14.6 に示す。俯瞰的に捉えると、add(x) メソッドによって、値 x を追加すべき葉 u が見つかる。追加によって u が一杯になる（つまり、すでに u に $2B-1$ 個のキーがある）場合には、u を分割する。それによって u の親が一杯になる場合には、u の親を分割する。それによって u の親の親が一杯になる場合には、u の親の親を分割する……という操作を繰り返す。木を 1 つずつ上に登りながら、一杯でないノードを見つけるか、根を分割することになるまで、この操作を繰り返す。一杯でないノードが見つかった場合には、単に操作を終了する。根を分割することになる場合には、新たな根を作り、元の根を分割して得られる 2 つのノードを両方とも新し

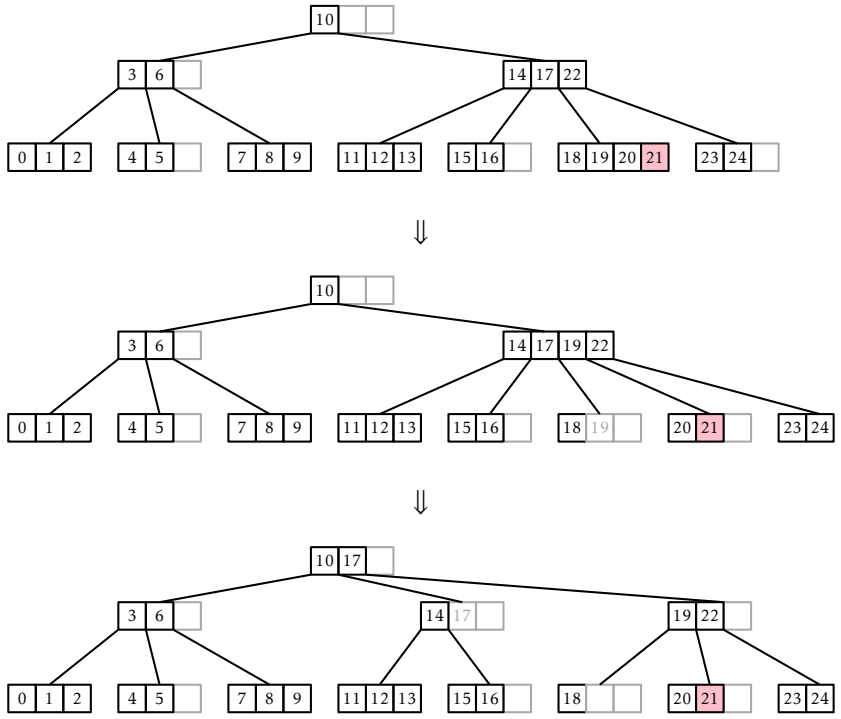


図 14.6: BTree における add(x) の様子。値 21 を追加すると、2 つのノードが分割される

い根の子にする [*4]。

add(x) メソッドが実行することを整理すると、次のようになる。すなわち、x を追加すべき葉を探して根から開始し、見つけた葉に x を追加して、それから根に向かって戻り、その途中で一杯になったノードを見かけたらすべて分割する。おおまかな動作がわかったところで、再帰的な実装方法を見ていこう。

add(x) で処理のほとんどを担当する addRecursive(x,ui) は、識別子 ui を持つノード u を根とする部分木に x を追加するメソッドだ。u が葉なら、単に x を u.keys に挿入する。そうでないときは、u の子のうちで適切なもの u′ に対し、x を再帰的に処理する。この再帰的な呼び出しは、通常は null を返すが、u′ が分割された場合は、新たに作られるノード w の参照を返すことが

[*4] 訳注：これに似た議論は 9.1.1 節に出てくる。

ある。後者の場合には、u が w を子として最初のキーを引き取り、u′ の分割処理を終える。

addRecursive(x,ui) では、u または u の子孫に x を追加したあと、u の持つキーが多すぎないか（$2B-1$ より多くないか）どうかを確認する。もし多すぎるなら、u を**分割**しなければならないので、u.split() を呼ぶ。u.split() の返り値である新しいノードが、addRecursive(x,ui) の返り値として使われる。

BTree

```
Node* addRecursive(T x, int ui) {
  Node *u = bs.readBlock(ui);
  int i = findIt(u->keys, x);
  if (i < 0) throw(-1);
  if (u->children[i] < 0) { // 葉ノードである。単に追加する
    u->add(x, -1);
    bs.writeBlock(u->id, u);
  } else {
    Node* w = addRecursive(x, u->children[i]);
    if (w != NULL) {  // 子は分割された。w は新たな子である
      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;
}
```

addRecursive(x,ui) は add(x) の下請けである。add(x) では、x を B 木の根に挿入するために、addRecursive(x,ri) を呼ぶ[*5]。addRecursive(x,ri) によって根が分割される場合、新しい根は、古い根および古い根の分割におい

[*5] 訳注：整数 ri は、BTree クラスで根のインデックスとして定義したことを思い出そう。一方で、addRecursive(x,ui) の引数として用いられている ui は、最初の呼び出しでは ri そのものであるが、以降はノード u のインデックスであることに注意。

て新たに作られたノードを子として持つ。

BTree

```
bool add(T x) {
       Node *w;
       try {
         w = addRecursive(x, ri);
       } catch (int e) {
         return false; // 重複した値を加えようとしている
       }
       if (w != NULL) {   // 根は分割された。新たな根を作る
    Node *newroot = new Node(this);
    x = w->remove(0);
    bs.writeBlock(w->id, w);
    newroot->children[0] = ri;
    newroot->keys[0] = x;
    newroot->children[1] = w->id;
    ri = newroot->id;
    bs.writeBlock(ri, newroot);
       }
       n++;
       return true;
}
```

add(x) および addRecursive(x,ui) は、二段階に分けて解析できる。

下向きに進む段階 再帰において下向きに進む段階では、x を追加する前に、各ノードにて findIt(a,x) を呼び、BTree のノードを順番にアクセスする。find(x) と同様に、このメソッドの実行時間は、外部メモリモデルでは $O(\log_B n)$、ワード RAM モデルでは $O(\log n)$ である。

上向きに進む段階 再帰において上向きに進む段階では、x を追加したあと、合計で最大 $O(\log_B n)$ 回の分割を行う。各分割は 3 つのノードだけに影響するので、この段階の実行時間は、外部メモリモデルでは $O(\log_B n)$ である。しかし、各分割では B 個のキーと子をノードからノードに移

すので、ワード RAM モデルでは $O(B \log n)$ である。

B の値はかなり大きくなる。$\log n$ と比べてもだいぶ大きいことを思い出そう。そのため、ワード RAM モデルでは、B 木への要素の追加はバランスされた二分探索木への追加よりもかなり遅くなる可能性がある。このあと、14.2.4 節で、状況がそれほど悲惨ではないことを示す。実は、償却すると、add(x) で実行される分割の回数は定数オーダーなのである。したがって、ワード RAM モデルにおける add(x) の償却実行時間は、$O(B + \log n)$ である。

14.2.3 ノードの削除

BTree における remove(x) も、実装には再帰を使うのが最も簡単だ。再帰による remove(x) の実装は、いくつかのメソッドにまたがる複雑なものだが、図 14.7 に示したように全体として見ればとても素直である。削除という課題は、うまくキーを入れ替えることで、ある葉 u から値 x′ を削除するという問題に帰結される。x′ を削除することで、u の持つキーの数が $B-1$ 未満になる場合もあるだろう。この状態を**アンダーフロー**（underflow）と呼ぶことにする。

アンダーフローが発生した場合、u は、自分の兄弟からキーを借用するか、自分の兄弟のいずれかと併合することになる。兄弟と併合する場合は、u の親が持つ子とキーの数が 1 ずつ減り、結果として今度は u の親でアンダーフローが発生するかもしれない。このアンダーフローは、再び兄弟からのキーの借用か、兄弟との併合により解決される。併合すれば、今度は u の親の親でアンダーフローが発生するかもしれない。この処理は根へと上向きに進み、アンダーフローが発生しなくなるか、根の 2 つの子が 1 つに併合されるかすれば終了する。後者の場合は根が削除され、その唯一の子が新たな根になる。

それでは、各ステップの実装方法を詳細に見ていこう。remove(x) で最初にすべきことは、削除する要素 x の探索である。x が葉で見つかったなら、その葉から x を削除する。そうでなく、x がある内部ノード u の u.keys[i] で見つかったなら、u.children[i + 1] を根とする部分木の最小値 x′ を削除する。x′ は、x より大きい値を格納する BTree の最小値である。続いて、x′ の値で u.keys[i] の x を置き換える。図 14.8 にこの処理の様子を示す。

removeRecursive(x,ui) は、上記で説明したアルゴリズムの再帰的な実装である。

BTree

```
T removeSmallest(int ui) {
```

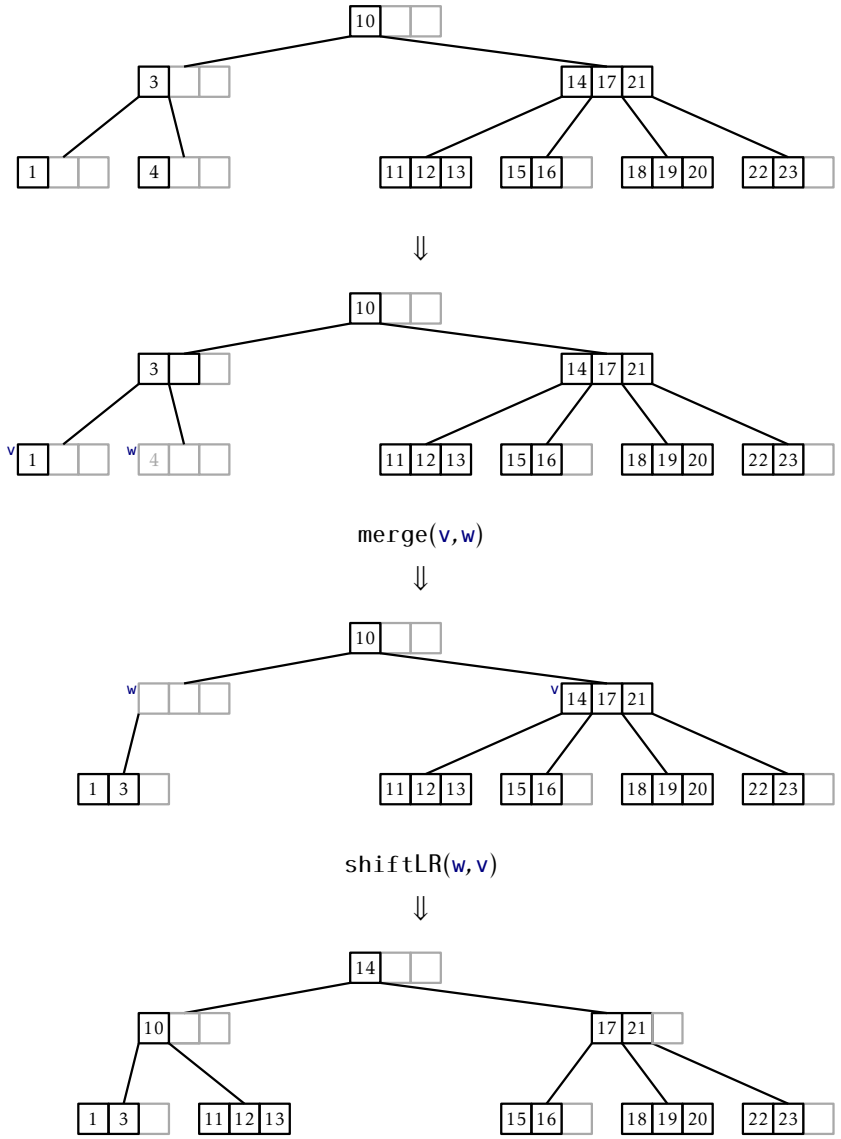


図 14.7: この B 木から値 4 を削除すると、併合と借用が 1 回ずつ発生する

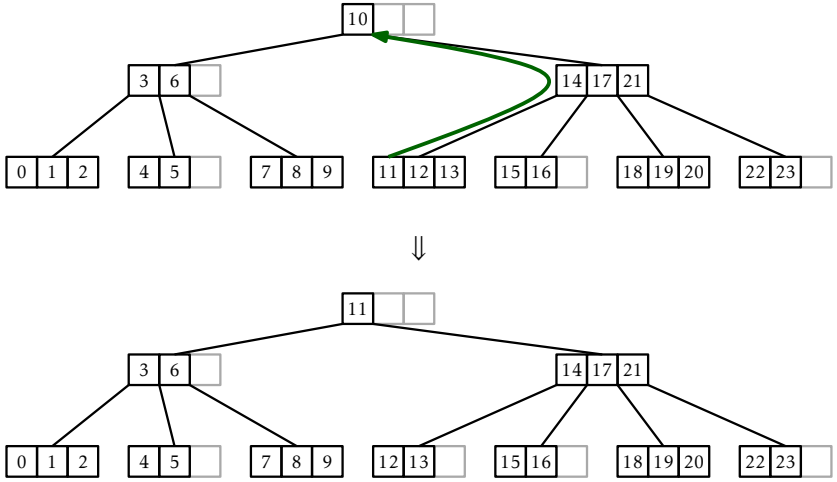


図 14.8: BTree において remove(x) を実行する様子。値 x = 10 を削除するとき、その値を x′ = 11 で上書きし、値 11 を含む葉を削除する

```
  Node* u = bs.readBlock(ui);
  if (u->isLeaf())
    return u->remove(0);
  T y = removeSmallest(u->children[0]);
  checkUnderflow(u, 0);
  return y;
}
bool removeRecursive(T x, int ui) {
  if (ui < 0) return false;  // 見つからなかった
  Node* u = bs.readBlock(ui);
  int i = findIt(u->keys, x);
  if (i < 0) { // 見つけた
    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;
}
```

removeRecursive(x,ui) では、u の i 番めの子から値 x を再帰的に削除したあと、この子が少なくとも $B-1$ 個のキーを持っていることを保証しなければならない。上記のコードでは、checkUnderflow(x,i) がこの処理を行っている。このメソッドは、u の i 番めの子についてアンダーフローの発生を確認し、修正する。w を u の i 番めの子とする。w のキーが $B-2$ 個しかないなら、修正の必要がある。これには w の兄弟を利用する。u の $i+1$ 番め、または $i-1$ 番めの子を使う。通常は u の $i-1$ 番めの子 v、つまり w のすぐ左の兄弟を使う。$i=0$ のときだけはこれがうまくいかないので、w のすぐ右の兄弟を使う。

BTree

```
void checkUnderflow(Node* u, int i) {
  if (u->children[i] < 0) return;
  if (i == 0)
    checkUnderflowZero(u, i); // u の右の兄弟を使う
  else
    checkUnderflowNonZero(u, i);
}
```

ここでは $i \neq 0$ の場合のみを考え、u の i 番めの子で発生したアンダーフローが u の $(i-1)$ 番めの子の助けを借りて修正できることを確認する。$i=0$ の場合も同様に処理できるので、詳細はソースコードを参照してほしい。

w におけるアンダーフローを解決するには、w に追加するキー（場合によっ

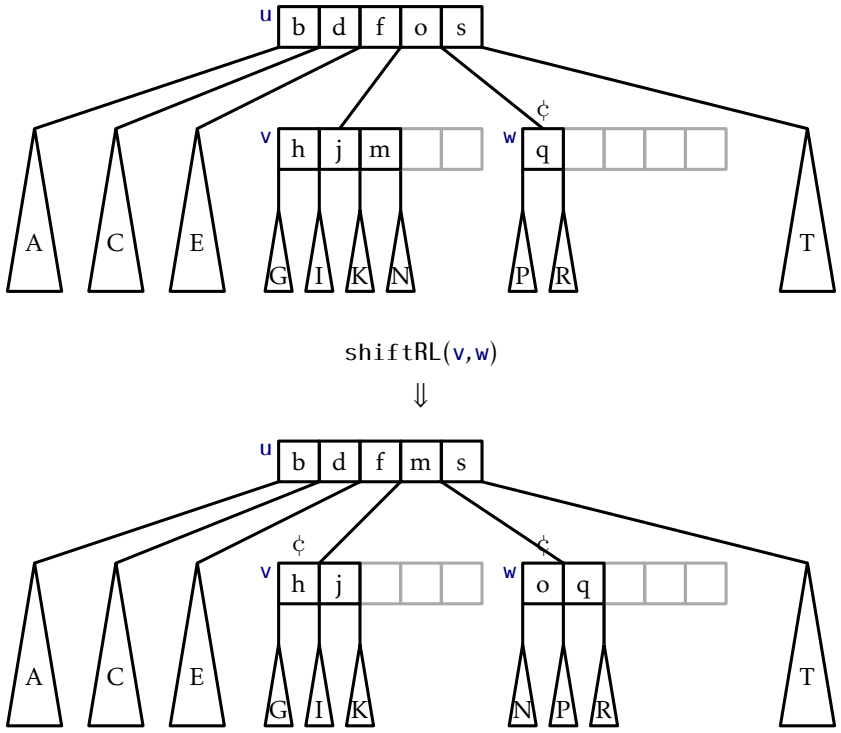


図 14.9: v のキーの個数が $B-1$ 個より多いとき、w は v からキーを借りられる

ては子も）を見つけてくる必要がある。そのための操作は 2 種類ある。

借用 w の兄弟 v が持っているキーの個数が $B-1$ より多ければ、w は v からキー（場合によっては子も）を借りられる。具体的には、v のキーの個数が size(v) なら、v と w とが持っているキーの個数の合計は次のようになる。

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

よって、v から w にキーを移し、v と w のいずれもが $B-1$ 個以上のキーを持つ状態にできる。この操作の様子を図 14.9 に示す。

併合 v のキーの個数が $B-1$ 個だけしかないとき、v にはキーを渡す余裕がないので、もっと思い切った操作が必要になる。それは、図 14.10 に示すように、v と w の**併合**である。併合は分割とは逆の操作である。合計で $2B-3$ 個のキーを持つ 2 つのノードを併合し、$2B-2$ 個のキーを

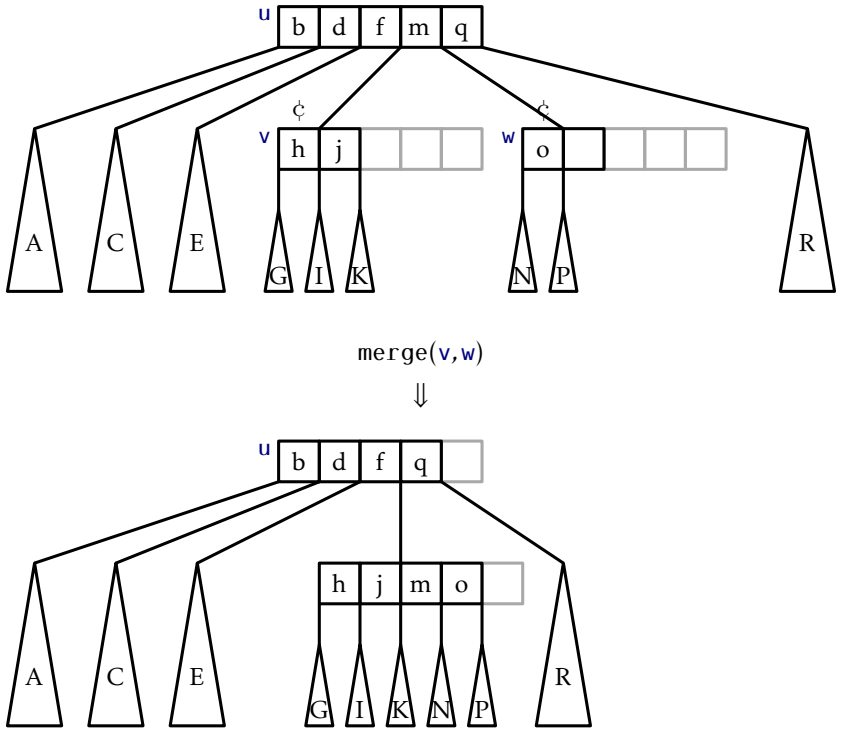


図 14.10: $B = 3$ である B 木の兄弟 v と w を併合する

持つ 1 つのノードにする（併合されたノードでキーの個数が 1 つ増えるのは、v と w を併合すると両者の親 u の子の数が 1 つ減るので、u が保有するキーの 1 つを併合されたノードに受け渡す必要があるからである）。

BTree

```
void checkUnderflowZero(Node *u, int i) {
  Node *w = bs.readBlock(u->children[i]);
  if (w->size() < B-1) {  // w でアンダーフローが発生
    Node *v = bs.readBlock(u->children[i+1]);
    if (v->size() > B) { // w は v から借用できる
      shiftRL(u, i, v, w);
```

```
    } else { // v は w を併合する
      merge(u, i, w, v);
      u->children[i] = w->id;
    }
  }
}
void checkUnderflowNonZero(Node *u, int i) {
  Node *w = bs.readBlock(u->children[i]);
  if (w->size() < B-1) {  // w でアンダーフローが発生
    Node *v = bs.readBlock(u->children[i-1]);
    if (v->size() > B) {  // w は v から借用できる
      shiftLR(u, i-1, v, w);
    } else { // v は w を併合する
      merge(u, i-1, v, w);
    }
  }
}
```

まとめると、B 木における remove(x) では、根から葉まである経路を辿り、x′ を葉 u から削除して、そのあとで 0 回以上の併合を u とその祖先に対して実行し、高々 1 回の借用をする。併合や借用では 3 つのノードしか修正せず、操作の回数は $O(\log_B n)$ なので、外部メモリモデルにおける remove(x) の全体としての実行時間は $O(\log_B n)$ である。ただし、ワード RAM モデルでは併合やノードの借用に $O(B)$ だけの時間がかかるので、remove(x) の実行時間は $O(B\log_B n)$ である（それ以上のことは現時点ではわからない）。

14.2.4 B 木の償却解析

ここまでに、次の事実を見てきた。

1. 外部メモリモデルでは、B 木における find(x)、add(x)、remove(x) の実行時間はそれぞれ $O(\log_B n)$ である
2. ワード RAM モデルでは、find(x) の実行時間は $O(\log n)$ であり、

add(x) および remove(x) の実行時間は $O(B \log n)$ である

次の補題により、B 木における分割および併合操作の回数に関するこれまでの見積もりが過剰であったことがわかる。

補題 **14.1.** 空の B 木から始めて、add(x) および remove(x) からなる m 個の操作の列を順に実行するとき、分割、併合、借用は合わせて高々 $3m/2$ 回しか実行されない。

証明. $B = 2$ という特別な場合については、すでに 9.3 節で証明の概要を示した。この補題を証明するには、次のような性質のもとで出納法を使えばよい。

1. 分割、併合、借用の際に、預金 2 を支払う（失う）
2. add(x) または remove(x) の際には、最大で預金 3 が得られる

最大で得られる預金が $3m$ であり、分割、併合、借用されるたびに支払う預金が 2 なので、最大で $3m/2$ 回の分割、併合、借用が実行されることになる。図 14.5、図 14.9、図 14.10 では、預金を ¢ で表した。

証明では、預金の値を追うために、次の**預金不変条件**（credit invariant）を考える。

- $B-1$ 個のキーを持つ任意の根でないノードは預金を 1 だけ持つ
- $2B-1$ 個のキーを持つノードは預金を 3 だけ持つ
- B 以上 $2B-2$ 以下のキーを持つノードは預金を持たない

そのうえで、毎回の add(x) および remove(x) の操作で預金不変条件を保持できること、証明の冒頭で提示した性質 1 と 2 を満たせることを示せばよい。

追加の場合 add(x) では併合や借用が発生しないので、分割だけを考えれば十分である。

すでに $2B-1$ 個のキーを持つノード u にキーを追加すると、分割が発生する。この場合、u は 2 つのノード u′ と u″ に分割され、それぞれは $B-1$ 個および B 個のキーを持つ。直前には u が $2B-1$ 個のキーを持っていたので、預金は 3 あった。そのうちの 2 は分割のために支払われ、残りの 1 は $B-1$ 個のキーを持つ u′ に渡されるので、預金不変条件は保持される。よって、いかなる分割においても、預金不変条件を保ちながら、その分割のための預金を支払える。

それ以外に、add(x) の実行中にノードに対して発生する操作は、分割が起こるとして、それらの分割がすべて終わったあとで実行される。発生する操作

は、新たなキーをあるノード u′ に追加する処理に関係するものだ。それ以前に u′ の子の数が $2B-2$ 個だったならば、子の数が $2B-1$ になるので、u′ は預金 3 を得ることになる。add(x) メソッドによって放出される預金はこれだけである。

削除の場合 remove(x) の際には、0 回以上の併合と、それに続く借用が 1 回発生する可能性がある。併合が発生するのは、remove(x) を呼ぶ前に 2 つのノード v と w が共に $B-1$ 個のキーを持っていて、この 2 つのノードが $2B-2$ 個のキーを持つ 1 つのノードに併合されるという状況である。そのため、併合のたびに、併合に使われる預金 2 が支払われている。

併合のあとには、借用が高々 1 回発生する。それ以降は、併合も借用も発生しない。借用が起こるのは、$B-1$ 個のキーを持つ葉 v からキーを削除する場合に限られる。このとき v は預金 1 を持っており、この預金が借用のコストとして使われる。しかし、借用のコストは 2 なので、預金が 1 足らない。支払いを完了するには、預金があと 1 必要である。

この時点で作り出した預金が 1 あるので、預金不変条件が保持されていることを示す必要がある。最悪の場合、v の兄弟 w が借用の前にちょうど B 個のキーを持っていて、直後には v と w が両方とも $B-1$ 個のキーを持つことになる。これは、操作が完了するとき、v と w が預金を 1 持っている必要があることを意味する。この場合には、v と w に渡すために、追加で 2 の預金を作る必要がある。借用は remove(x) の処理において高々 1 回発生するので、必要に応じて最大で 3 の預金を作ることになる。

remove(x) において借用が発生しないのは、操作の前に B 個以上のキーを持っていたノードからキーを削除して終了した場合である。最悪の場合、キーを削除されたノードは操作の前にちょうど B 個のキーを持っていて、そのため操作後のキーの個数が $B-1$ になっているので、作った預金から 1 を与えなければならない。

削除が借用で終わるにせよ、そうでないにせよ、預金不変条件を保持して併合と借用のコストを支払うためには、remove(x) の呼び出しに際して高々 3 の預金を作る必要がある。以上より補題が示された。 □

補題 14.1 の目的は、ワード RAM モデルにおいて add(x) および remove(x) からなる m 個の操作の列を順に実行するとき、分割、併合、借用にかかる時間が合わせて $O(Bm)$ であることを示すことにあった。つまり、これらの操作の償却コストは $O(B)$ であり、ワード RAM モデルにおける add(x) および remove(x) の償却コストは $O(B+\log n)$ である。この結果を次の 2 つの定理にまとめる。

定理 **14.1** (外部メモリモデルにおける B 木)**.** *BTree* は *SSet* インターフェースを実装する。*BTree* は add(x)、remove(x)、find(x) をサポートし、外部メモリモデルではいずれの実行時間も $O(\log_B n)$ である。

定理 **14.2** (ワード RAM モデルにおける B 木)**.** *BTree* は *SSet* インターフェースを実装する。*BTree* は add(x)、remove(x)、find(x) をサポートする。ワード *RAM* モデルでは、分割、併合、借用のコストを無視すると、いずれの実行時間も $O(\log_B n)$ である。さらに、空の *BTree* に対して add(x) および remove(x) からなる m 個の操作の列を順に実行するとき、分割、併合、借用のためにかかる時間は合わせて $O(Bm)$ である。

14.3 ディスカッションと練習問題

外部メモリモデルを提案したのは Aggarwal と Vitter である [4]。このモデルは I/O **モデル**や**ディスクアクセスモデル**と呼ばれることもある。

B 木は、内部メモリを使った探索における二分探索木を、外部メモリの場合に拡張したものである。B 木は、McCreight が 1970 年に提案した [9]。それから 10 年を待たずして、Comer によるサーベイ（論文のタイトルは “The Ubiquitous B-Tree”）が出版され、その中で B 木は「このデータ構造はいたるところで使われている」と紹介された [15]。

二分探索木と同様に、B 木には多くの種類がある。例えば、B^+ 木、B^* 木、counted B 木などである。B 木は、多くのファイルシステムにおける基本的なデータ構造として、本当にいたるところで使われている。例えば、Apple の HFS+、Microsoft の NTFS、Linux の Ext4 などがある。すべてのメジャーなデータベースシステムも B 木の例である。クラウドコンピューティングで使われているキーバリューストアにも利用例がある。Graefe による近年のサーベイ [36] では、200 ページ以上にわたって、B 木の現代における応用やデータ構造の変種、最適化などが述べられている。

B 木は SSet インターフェースを実装する。USet インターフェースだけが必要なら、B 木の代わりに外部メモリハッシュ法を使うこともできるだろう。外部メモリハッシュ法も広く研究されている。例えば Jensen と Pagh の論文 [43] を見てほしい。外部メモリハッシュ法では、$O(1)$ の期待実行時間で、外部メモリモデルにおいて USet の操作を実行できる。しかし、いくつかの理由から、多くのアプリケーションでは USet の操作だけが必要だとしても B 木を使っている。

B 木がよく利用される理由のひとつに、$O(\log_B n)$ という実行時間の上界から受ける印象よりも実際の性能がよい場合が少なくないという点が挙げられる。外部メモリモデルでは、B の値はふつうかなり大きく、数百あるいは数千である。そのため、B 木におけるデータのうち 99%、あるいは 99.9% は葉に保存されている。大きなメモリを持つデータベースシステムでは、内部ノードはすべてのデータのうちの 1%、あるいは 0.1% 程度なので、すべて RAM にキャッシュできるかもしれない。この場合、B 木の検索では RAM 上にある内部ノードをすべて非常に高速に処理でき、外部メモリに 1 回だけアクセスして葉が得られる。

問 **14.1.** 図 14.2 の B 木に 1.5、7.5 を順に追加するときの様子を描け。

問 **14.2.** 図 14.2 の B 木から 3、4 を順に削除するときの様子を描け。

問 **14.3.** n 個のキーを格納する B 木の内部ノードの数の最大値を求めよ。（これは n と B の関数である。）

問 **14.4.** この章の冒頭で、B 木の内部メモリとして必要なのは $O(B+\log_B n)$ だけであると述べた。しかし、この章で示した実装では、実はより多くのメモリが必要である。

1. この章で示した `add(x)` および `remove(x)` の実装では、$B\log_B n$ に比例する内部メモリを使うことを示せ。
2. これを $O(B+\log_B n)$ に減らすための修正方法を説明せよ。

問 **14.5.** 補題 14.1 の証明で使った預金の様子を、図 14.6 と図 14.7 の木に描け。また、追加の預金 3 で分割、併合、借用のコストを支払い、預金不変条件を保持できることを確認せよ。

問 **14.6.** B 木を修正し、ノードの子の数が B 以上 $3B$ 以下（したがってキーの数は $B-1$ 以上 $3B-1$ 以下）のデータ構造を設計せよ。また、この新しい B 木では、m 回の操作を順に実行する際に $O(m/B)$ 回だけ分割、併合、借用を実行することを示せ。（ヒント：これを実現するには、場合によっては併合が必要になる前に 2 つのノードを併合して、より積極的に併合処理を実施する必要があるだろう。）

問 **14.7.** この練習問題では、B 木の分割と併合を修正して最大で 3 つのノードを一度に考慮することで、分割、借用、併合処理の漸近的な実行回数を減らす。

1. 一杯になったノードを u とし、u のすぐ右の兄弟を v とする。u のノードが溢れるのを解消する方法は 2 通りある。
   (a) u のキーをいくつか v に渡す
   (b) u を分割し、u と v のキーを平等に u と v、それに新しいノード w で分け合う
   この操作のあと、ある定数 $\alpha > 0$ について、関連する（最大 3 つの）ノードはいずれも $B + \alpha B$ 個以上 $2B - \alpha B$ 個以下のキーを持つようにできることを示せ
2. ノード u はアンダーフローしているものとし、v および w を u の兄弟とする。u のアンダーフローを解消する方法は 2 通りある。
   (a) u、v、w の間でキーを分配し直す
   (b) u、v、w を併合して 2 つのノードにする。それぞれが持っていたキーを 2 つのノードに分配し直す
   この操作のあと、ある定数 $\alpha > 0$ について、関連する（最大 3 つの）ノードはいずれも $B + \alpha B$ 個以上 $2B - \alpha B$ 個以下のキーを持つようにできることを示せ
3. 以上の修正によって、m 回の操作を実行する間に発生する併合、借用、分割の回数が $O(m/B)$ になることを示せ。

問 **14.8.** 図 14.11 に示した B^+ 木では、すべてのキーを葉に格納し、すべての葉を双方向連結リストとして格納する。これまで通り、葉にはそれぞれ $B-1$ 個以上 $2B-1$ 個以下のキーを格納する。葉よりも上側の部分は通常の B 木であり、その内部ノードには、葉のリストのうち末尾を除いた最大の値が格納されている。

1. B^+ 木における `add(x)`、`remove(x)`、`find(x)` の高速な実装を説明せよ。
2. `findRange(x,y)` の効率的な実装方法を説明せよ。これは、B^+ 木に含まれる `x` より大きく `y` より小さい値をすべて報告するメソッドである。
3. `find(x)`、`add(x)`、`remove(x)`、`findRange(x,y)` を持つクラス `BPlusTree` を実装せよ。
4. B^+ 木では、B 木の部分とリストの部分の両方に同じキーを格納するため、キーの重複がある。B の値が大きいとき、この重複が深刻な問題にならない理由を説明せよ。

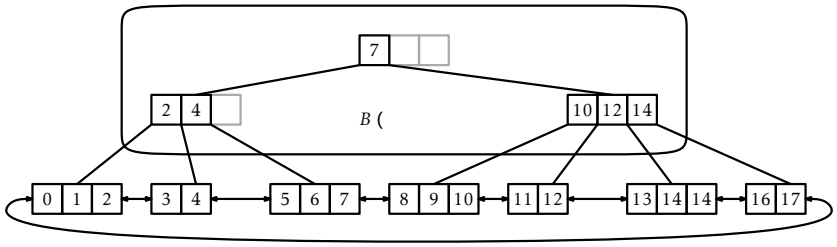


図 14.11: B^+ 木は、双方向連結リストの上に B 木が乗ったデータ構造である

参考文献

[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://courses.csail.mit.edu/6.042/spring12/mcs.pdf [cited 2012-09-06].

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

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

[53] J. I. Munro, T. Papadakis, and R. Sedgewick. Deterministic skip lists. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms (SODA'92)*, pages 367–375, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

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

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

[56] T. Papadakis, J. I. Munro, and P. V. Poblete. Average search and up-

date costs in skip lists. *BIT*, 32:316–332, 1992.

[57] M. Pătraşcu and M. Thorup. Randomization does not help searching predecessors. In N. Bansal, K. Pruhs, and C. Stein, editors, *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7–9, 2007*, pages 555–564. SIAM, 2007.

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

[59] W. Pugh. A skip list cookbook. Technical report, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, 1989. URL: ftp://ftp.cs.umd.edu/pub/skipLists/cookbook.pdf [cited 2011-07-20].

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

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

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

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

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

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

[66] H. H. Seward. Information sorting in the application of electronic digital computers to business operations. Master's thesis, Massachusetts Institute of Technology, Digital Computer Laboratory, 1954.

[67] Z. Shao, J. H. Reppy, and A. W. Appel. Unrolling lists. In *Proceedings of the 1994 ACM conference LISP and Functional Programming (LFP'94)*, pages 185–195, New York, 1994. ACM.

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

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

[70] D. Sleator and R. Tarjan. Self-adjusting binary trees. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25–27 April, 1983, Boston, Massachusetts, USA*, pages 235–245. ACM, ACM, 1983.

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

[72] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977.

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

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

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

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

索引

日本語牽引