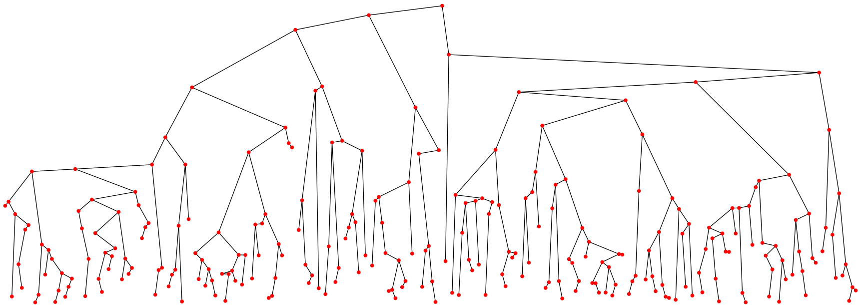


Open Data Structures (in Java) 日本語版

Edition 0.1G

Pat Morin

堀江 慧、陣内 デイビッド 佑、田中 康隆　共訳

目次

この本の読み方（翻訳者まえがき）

実用上極めて重要で息をするように取り出せるようにしておくべきものと、そうではないものとを明確に区別しておくことは有益だと我々は考えた。以下に列挙するデータ構造（とそれらの上に成り立つアルゴリズム）は学術研究やプログラマーの実務で頻繁に登場し、全ての学習者が深く理解しておくことが望ましいと訳者３人の意見が一致したものだ。

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

ここに並べなかったややマイナーなデータ構造にも別の点で学ぶ価値がある。それは、新しいデータ構造に用いられるアイデアを知ること、そしてそのデータ構造を解析する道具立てを学ぶことである。マイナーかは必ずしも重要ではない。どの章を重点的に読むか、興味に応じて適宜調整して頂ければ幸いである。

翻訳者謝辞

クラウドファンディングに参加して頂いた皆様に感謝を捧げたい。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 ライセンスで公開することが出来た。本書のプロジェクトページ [*1] には他言語版やプロジェクトに関する情報がある。本書のソースコードは Github [*2] にある。

*1 https://sites.google.com/view/open-data-structures-ja
*2 https://github.com/spinute/ods

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 の受講生達、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。できの良いものもある。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、Java ソース、およびビルドスクリプトを、著者の Web サイト [*3] あるいは信頼できるソースコード管理サイト [*4] からダウンロードできる。[*5] である。また、日本語版のソースコードは `https://github.com/spinute/ods` にある。

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 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。

[*3] `http://opendatastructures.org`

[*4] `https://github.com/patmorin/ods`

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

ソースコード管理システム git を使って、誰でも本書の修正案を送れる。本のソースをフォークして別バージョンを作ることもできる。（例えば別のプログラミング言語を使う版を作れる。）こうしたやり方で、私のやる気や興味が衰えた後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。

第 1

Introduction

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づ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、各操作に普通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 $\mathtt{n}$ 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、機械命令の数にしておよそ $\mathtt{n} \times m$ 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すむように、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、ひと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。より正確にいうと、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() は引数をとらない。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

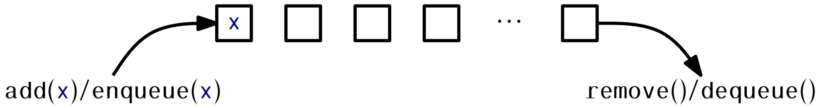


図 1.1: FIFO キュー

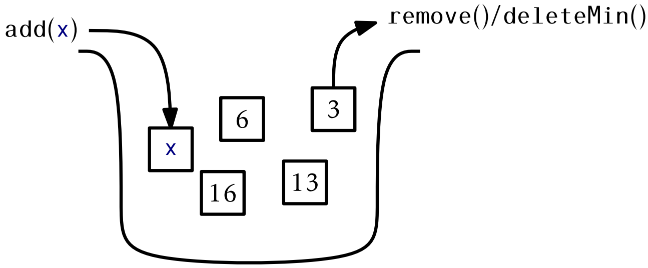


図 1.2: 優先度付きキュー

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() 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。LILO 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ひとつずつ取るとき、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、先頭と末尾をも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()

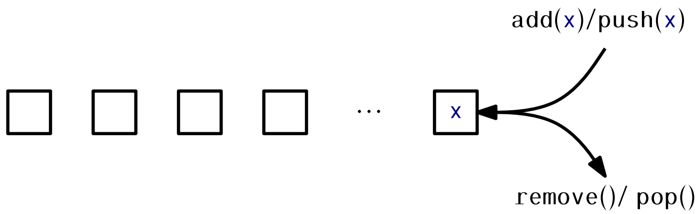


図 1.3: LIFO キュー（スタック）

というわかりやすい名前がついている。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 インタフェースを示す。List インタフェースは、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。

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

[*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 番目の要素とは、先頭から数えて $i+1$ 個目の要素のことである。

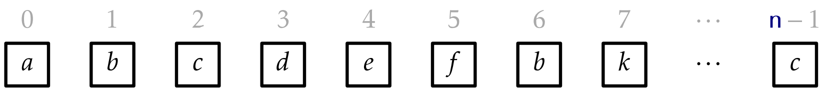


図 1.4: List は $0,1,2,\ldots,\mathtt{n}-1$ で添字づけられた列を表現する。この List で get(2) を実行すると値 c が返ってくる。

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

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。
   $\mathtt{x} = \mathtt{y}$ 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。
   $\mathtt{x} = \mathtt{y}$ 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くなる場合があるからだ [*5]。そのような判定ができると、キーを値に対応づけるインターフェースである**辞書** (dictionary)（**マップ** (map) とも呼ばれる）を実装するのに都合がいい。

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関わらず）等しいとみなす。Pair である (k,v) を USet に入れてから、$x = (k, \mathrm{null})$ 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づ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。

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

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): 順序づけられた集合から x の位置を特定する。
   すなわち $y \geq x$ 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

[*5] Java では、クラスの equals(y)・hashCode() メソッドをオーバーライドするとこれを行える。

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、底になにも書かずに $\log k$ 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）[*6]。

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 [*7]。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。

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

[*6] 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。

[*7] 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。[*8]

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

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

[*8] 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良いと思われる。

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。たとえば、$5n\log n + 8n - 200$ 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。

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

つづいて、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。たとえば 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、たと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。

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

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。

$$T(n) = O(n)$$

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 $f_1(\mathtt{n}) = 15\mathtt{n}$ と $f_2(n) = 2\mathtt{n}\log\mathtt{n}$ のグラフである。$f_1(\mathtt{n})$ は複雑な線形時間アルゴリズムの実行時間を表し、$f_2(\mathtt{n})$ 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 $f_1(\mathtt{n})$ のほうが $f_2(\mathtt{n})$ 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 $f_1(\mathtt{n})$ 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 $O(\mathtt{n}) \subset O(\mathtt{n}\log\mathtt{n})$ は、この事実を示している。

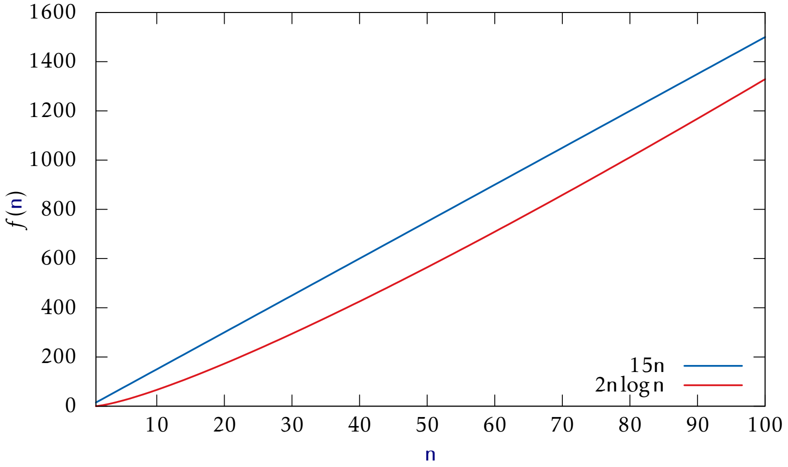
多変数関数に対して漸近記法を使うこともある。標準的な定義はないようだが、この本では次の定義を用いる。

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

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、サイコロの出目もふ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**期待実行時間**



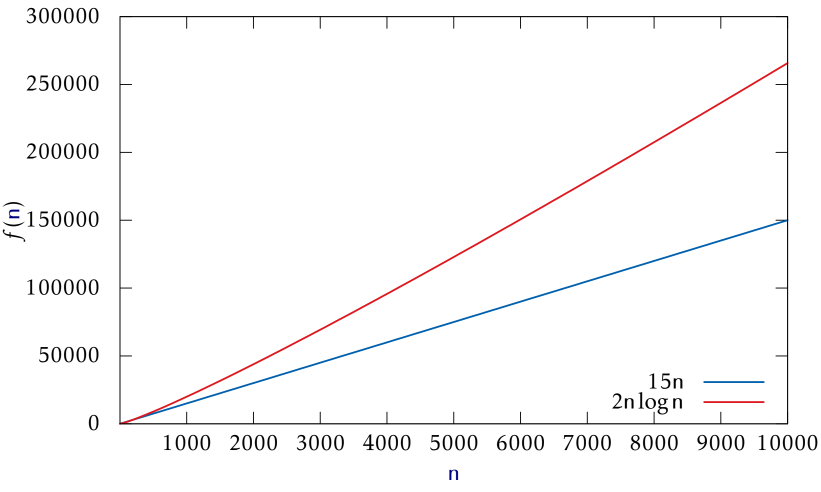


図 1.5: 15n と 2n log n の比較

(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のみを考慮して確率を定める。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ことは仮定しない。

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。

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

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 のメモリブロックを割り当てると、$O(k)$ の時間をかけて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 $\mathtt{w} > \log \mathtt{n}$ 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（実際、Java では `T` 型のオブジェクトの参照を格納しており、この参照は 1 ワードのメモリを占める。）。

`w` ビットのワード RAM モデルは、$\mathtt{w} = 32$ とすると、（32 ビット）Java 仮想マシン（JVM）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なかで、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 $f(\mathtt{n})$ であるといったら、そのような操作の実行時間が $f(\mathtt{n})$ 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 $f(\mathtt{n})$ であるとは、典型的な操作にかかるコストが $f(\mathtt{n})$ 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、$mf(\mathtt{n})$ を超えないことを意味する。いくつかの操作には $f(\mathtt{n})$ 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 $f(\mathtt{n})$ という意味だ。

期待実行時間（expected running time）：　期待実行時間が $f(\mathtt{n})$ 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 $f(\mathtt{n})$ 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。

$$\$120\,000/120\text{ ヶ月} = \$1\,000\text{ 月あたり}$$

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。

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 ドルの節約に賭けるべきだろうか？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。

この例からわかるように、どちらを選ぶかは場合によって変わる [*9]。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、より短い償却実行時間や期待実行時間で妥協すれば、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 Java で書いた。ただし、Java 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 Java ソースコードを見てほ

[*9] 訳注：この例では、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**ことのほうがはるかに多い。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。

しい。

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Java 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。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。付属のソースコードには、USet、SSet の実装を Set、Map、SortedSet、SortedMap の実装にするためのラッパークラスが含まれている。

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 [73] 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 [65] 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。

Java のプログラミング力を磨きたい読者のためには、オンラインの Java のチュートリアル [56] がある。

問 **1.1.** 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のものだ。利用可能な実装やインターフェース（Java ならば Java

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

表 1.1: List、USet の実装の要約

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。

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. 最初の 50 行入力を読み、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り

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[I]	$O(w)$	$O(w)$	§ 13.1
XFastTrie[I]	$O(\log w)^{A,E}$	$O(w)^{A,E}$	§ 13.2
YFastTrie[I]	$O(\log w)^{A,E}$	$O(\log w)^{A,E}$	§ 13.3
BTree	$O(\log n)$	$O(B+\log n)^{A}$	§ 14.2
BTree[X]	$O(\log_B n)$	$O(\log_B n)$	§ 14.2

(Priority) 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

[I] 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

[X] これは外部メモリモデル（14 章を参照）での実行時間である

表 1.2: SSet、優先度付き Queue の実装の要約

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 行めからはじまり、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。

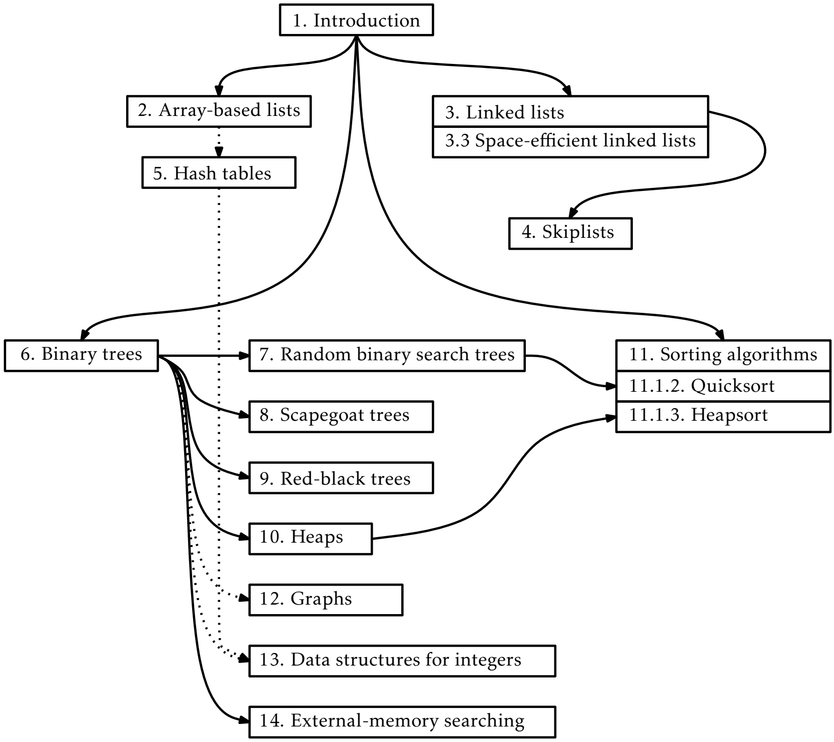


図 1.6: この本の内容の依存関係

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はじ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、短い順に並び替えて出力せよ。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後に、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、Stack の push(x) 操作および pop() 操作の関係を説明せよ。

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	$O(1)$	$O(\mathtt{n}-\mathtt{i})$
ArrayDeque	$O(1)$	$O(\min\{\mathtt{i},\mathtt{n}-\mathtt{i}\})$
DualArrayDeque	$O(1)$	$O(\min\{\mathtt{i},\mathtt{n}-\mathtt{i}\})$
RootishArrayStack	$O(1)$	$O(\mathtt{n}-\mathtt{i})$

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はじめて、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。

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 番目の要素は、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 $a[0],\ldots,a[n-1]$ に格納されている。このとき常に $a.length \geq n$ である。

ArrayStack
```
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 [*2] をしたあと、単に a[i] を返すか、a[i] を書き換えるかすればよい。

[*2] 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、最後の要素の添字である、つまり $a.length-1$ 以下であると確認することである。

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 $\mathtt{a.length} > \mathtt{n}$ 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 $\mathtt{a}[\mathtt{i}],\dots,\mathtt{a}[\mathtt{n}-1]$ 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、$O(\mathtt{n}-\mathtt{i})$ である。

remove(i) も同様に実装できる。$\mathtt{a}[\mathtt{i}+1],\dots,\mathtt{a}[\mathtt{n}-1]$ 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 $\mathtt{a.length} \geq 3\mathtt{n}$ かどうかを確認する。もしそうなら resize() を呼んで a を小さくする。

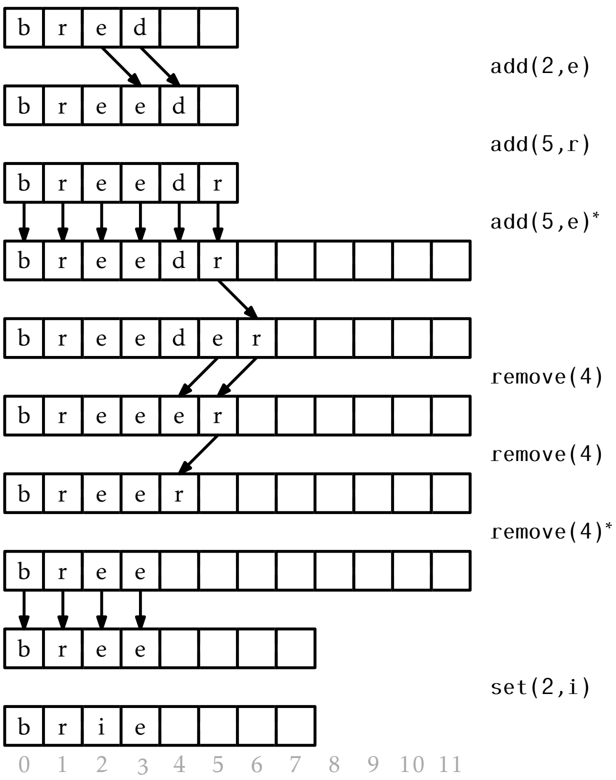


図 2.1: ArrayStack に対する add(i,x) と remove(i) の実行例。矢印は要素のコピーを表す。resize() を呼ぶ操作にはアスタリスクを付した

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、$O(\mathtt{n}-\mathtt{i})$ 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 [*3]。

ArrayStack
```
void resize() {
  T[] b = newArray(max(n*2,1));
  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 $O(\mathtt{n})$ 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 $O(\mathtt{m})$ 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

[*3] 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。

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。

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。

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

これを変形すると次の式が得られる。

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

$r \leq m$ 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。

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

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 $\texttt{a.length} = n = n_i$ 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 $\texttt{a.length}/2 = n_i/2$ 以下であった。しかし、a の要素数は今では $n_i = \texttt{a.length}$ 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 $\texttt{a.length} \geq 3n = 3n_i$ である。この 1 つ前、つまり $i-1$ 回目の resize() の直後では、a の要素数は $\texttt{a.length}/2-1$ 以上であった [*4]。今、a には $n_i \leq \texttt{a.length}/3$ 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。

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

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

いずれの場合も、$(i-1)$ から i 回目の resize() の間に add(i,x) か remove(i) が呼ばれる回数の合計は $\mathtt{n}_i/2-1$ 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 $O(1+\mathtt{n}-\mathtt{i})$ である [*5]

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。

[*5] 訳注：n = i = 0 のときであってもビッグオー記法で書けるように、1 が足されていることに注意。

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

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 Java の実装では、組み込みの System.arraycopy(s,i,d,j,n) 関数の利用により、操作の種類によっては 2〜3 倍の高速化につながる。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) を呼びだすことになり、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、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、時間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 $a = km + r$ をみたす整数 $r \in \{0, \ldots, m-1\}$ 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。 Java 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]

[*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。

剰余算術は無限長の配列を模倣するのに便利である。$\mathtt{i} \bmod \mathtt{a.length}$ が常に $0,\dots,\mathtt{a.length}-1$ 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。

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

ここでは a を**循環配列**として使っている。配列の添字が $\mathtt{a.length}-1$ 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。

ArrayQueue
```
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 $\mathtt{a}[(\mathtt{j}+\mathtt{n})\%\mathtt{a.length}]$ に入れて、n を 1 つ増やせばよい。

ArrayQueue
```
boolean 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、$\mathtt{j} = (\mathtt{j}+1) \bmod \mathtt{a.length}$ 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() {
```

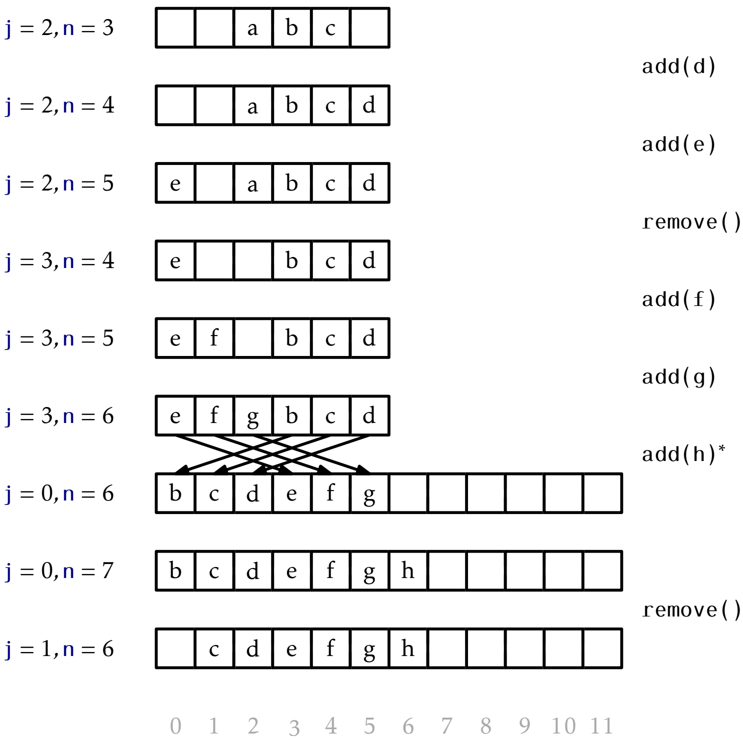


図 2.2: ArrayQueue に対する add(x)・remove() の実行例。矢印は要素のコピーを表す。resize() が発生する呼び出しにはアスタリスクを付した

```
  if (n == 0) throw new NoSuchElementException();
  T x = a[j];
  j = (j + 1) % a.length;
  n--;
  if (a.length >= 3*n) resize();
  return x;
}
```

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

2.3.1 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、*ArrayStack* は add(x)・remove() の実行時間は $O(1)$ である。さらに、空の *ArrayStack* 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。

2.4 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、一列に並んだデータを表すデータ構造であった。つづいて紹介する ArrayDeque は両端に追加・削除を効率的に実行できるデータ構造である。ArrayQueue の実装に使った循環配列をここでも使って、

ArrayDeque の List インタフェースを実装する。[*8]

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

ArrayDeque における get(i) と set(i,x) の実装は簡単だ。配列の要素 a[(j + i) mod a.length] を読み書きすればよいのだ。

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) の実装はひと工夫必要だ。まず、a が一杯かどうかを確認し、必要に応じて resize() を呼ぶ。ここで、i が小さいとき（0 に近いとき）と大きいとき（n に近いとき）に、特に効率的に操作したいのだということを覚えておいてほしい。続いて 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ひとつずつ左にずらす。そうでないなら右から $n-i$ 個の要素をいずれもひとつずつ右にずらす。add(i,x) と remove(x) の説明として図 2.3 を見てほしい。

ArrayDeque
```
void add(int i, T x) {
  if (n+1 > a.length) resize();
```

[*8] 訳注：第一章で言及したように、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 インターフェイスの全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。

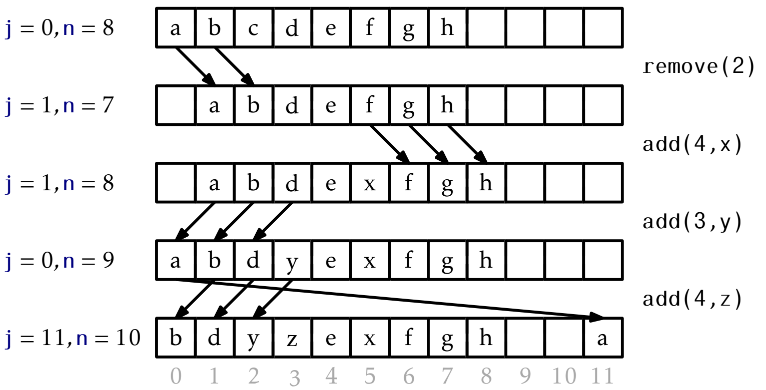


図 2.3: ArrayDeque に対する add(i,x)・remove(i) の実行例。矢印は要素のコピーを表す。

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

このように要素をずらすと add(i,x) は高々 $\min\{i, n-i\}$ 個の要素を移動する。以上より add(i,x) の（resize() のことを無視した）実行時間は $O(1+\min\{i, n-i\})$ である。

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

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

2.4.1 要約

次の定理は ArrayDeque の性能を整理するものだ。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、

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

さらに、空の ArrayDeque に対して長さ m の任意の add(i,x)・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。

[*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全ての操作を $O(1)$ 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)・remove(i) の実行時間は、$O(1)$ で済む。

2.5 DualArrayDeque：2つのスタックから作った双方向キュー

次はふたつの ArrayStack を使って ArrayDeque に近い性能を示すデータ構造 DualArrayDeque を紹介する。DualArrayDeque は漸近的な性能が ArrayDeque より良いわけではないのだが、ふた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ここで紹介する。

DualArrayDeque は、ふたつの ArrayStack を使ってリストを表現する。ArrayStack の終端付近の要素を高速に操作できることを思い出してほしい。DualArrayDeque は front と back という名前のふたつの ArrayStack を背中合わせに配置する。そのため両端が終端になり、両端を高速に操作できる。

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

DualArrayDeque は要素数 n を明示的に保持しない。要素数は n = front.size() + back.size() と求められるからだ。ただし DualArrayDeque の解析では相変わらず n で要素数を表すことにする。

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

ひとつめの ArrayStack である front には 0,...,front.size() − 1 番目の要素を、逆順に入れる。もうひとつの ArrayStack である back には front.size(),...,size()−1 番目の要素を普通の順番で入れる。あとは front か back に対して get(i) か set(i,x) を適切に呼べば get(i)・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 {
    return back.set(i-front.size(), x);
  }
}
```

front には逆順に要素が入っているので、インデックス $i <$ front.size() は front の front.size() $- i - 1$ 番目の要素である。

DualArrayDeque における要素の追加・削除は図 2.4 を見てほしい。add(i,x) は front か back を必要に応じて操作する。

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) は front と back の要素数を均すために balance() を呼び出す。balance() の実装は後述するが、size() < 2 でなければ front.size() と back.size() は三倍よりも離れないようにするとだけ今は知っていればよい。具体的には balance() は常に $3 \cdot$ front.size() $\geq$ back.size() かつ $3 \cdot$ back.size() $\geq$ front.size() であることを保証する。

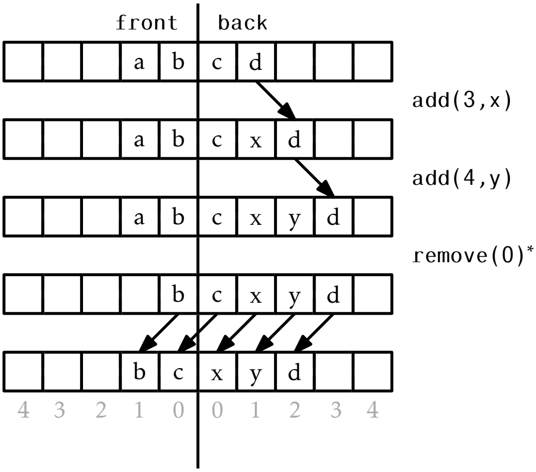


図 2.4: DualArrayDeque に対する add(i,x)・remove(i) の実行例。矢印は要素のコピーを表す。balance() の発生する呼び出しにはアスタリスクを付した。

つづいて add(i,x) のうち balance() のコストを無視した実行時間を求める。$i <$ front.size() のとき add(i,x) は front.add(front.size() − i,x) を実行するだけである。front は ArrayStack なのでこの実行時間は次のようである。

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

一方 $i \geq$ front.size() のとき add(i,x) は back.add(i − front.size(),x) を実行するだけである。このときの実行時間は次のようである。

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

$i < n/4$ のときはひとつめのケース (2.1) に該当する。$i \geq 3n/4$ のときはふたつめのケース (2.2) に該当する [*10]。$n/4 \leq i < 3n/4$ のときは、front と back のどちらを操作するかわからない。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 $O(n)$ である。以上

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

をまとめると次のようになる。

$$\text{Running time of add}(\mathtt{i},\mathtt{x}) \le \begin{cases} O(1+\mathtt{i}) & \text{if } \mathtt{i} < \mathtt{n}/4 \\ O(\mathtt{n}) & \text{if } \mathtt{n}/4 \le \mathtt{i} < 3\mathtt{n}/4 \\ O(1+\mathtt{n}-\mathtt{i}) & \text{if } \mathtt{i} \ge 3\mathtt{n}/4 \end{cases}$$

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

remove(i) の実行時間の解析は add(i,x) のものと同様なので省略する。

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) が実行する balance() の説明をする。この操作は front・back の要素数が極端には偏らないよう保つ。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のだ。そうでないときは要素を動かして、front・back がそれぞれちょうど $\lfloor \mathtt{n}/2 \rfloor$・$\lceil \mathtt{n}/2 \rceil$ 個の要素を持つようにする。

DualArrayDeque
```
void balance() {
  int n = size();
  if (3*front.size() < back.size()) {
    int s = n/2 - front.size();
```

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

balance() の解析は簡単である。balance() がバランス調整をするとき $O(\mathtt{n})$ 個の要素を動かすので実行時間は $O(\mathtt{n})$ である。balance() は add(i,x)・remove(i) で毎回呼ばれるのでこれは一見すると好ましくない。しかし次の補題より 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* に対して長さ m の任意の add(i,x)・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。

証明. 前に balance() が要素を動かしたときから、次に balance() が要素を動かすときまでに add(i,x)・remove(i) が実行された回数は $\mathtt{n}/2-1$ 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。

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

バランス調整を行わない add(i,x)・remove(i) の処理では、ポテンシャルは高々 1 しか増えないことに注目しよう。

次の式が成り立つので、balance() が要素を動かした直後のポテンシャル Φ_0 は 1 以下である。

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

balance() が要素を動かす直前には $3\mathtt{front.size()} < \mathtt{back.size()}$ であったと仮定しても一般性を失わない。次の式が成り立つ。

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

このときのポテンシャル Φ_1 は次のように評価できる。

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

以上より、前に balance() が要素を動かしてから、add(i,x)・remove(i) が呼ばれた回数は $\Phi_1 - \Phi_0 > \mathtt{n}/2 - 1$ 以上である。 □

2.5.2 要約

次の定理は DualArrayDeque の性質をまとめたものだ。

定理 **2.3.** *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、

- get(i)・set(i,x) の実行時間は $O(1)$ である。
- add(i,x)・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)・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。

2.6 RootishArrayStack：メモリ効率に優れた配列スタック

ここまで紹介したデータ構造には共通の欠点がある。1 つか 2 つの配列だけを使い、配列のサイズを頻繁には変更するのを避けているので、配列の中に空きがしばしばある点だ。例えば resize() 直後の ArrayStack では配列は半分しか埋まっていない。3 分の 1 しか埋まっていないことさえある。

この節では無駄なスペースの少ない RootishArrayStack というデータ構造を紹介する [*11]。RootishArrayStack は n 個の要素を $O(\sqrt{n})$ 個の配列に入れる。常に空きは合わせて $O(\sqrt{n})$ 箇所である。 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無駄になるスペースは $O(\sqrt{n})$ 以下である。

RootishArrayStack は**ブロック**と呼ぶ r 個の配列に要素を入れる。この配列は $0,1,\ldots,r-1$ と添字付けられる。参考のために図 2.5 を見てほしい。ブロック b は $b+1$ 個の要素を含む。すなわち r 個のブロックが含む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のは図 2.6 を見ればわかるだろう。

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

リストの要素はブロックに順番に入れる。0 番目の要素はブロック 0 に、1・2 番目の要素はブロック 1 に、3・4・5 番目の要素はブロック 2 に格納さ

[*11] 訳注：RootishArrayStack は前節までのデータ構造と比べると日常での遭遇頻度が極端に落ちる印象を受けるので、初学者は飛ばしてもよい。ただし課題設定と設計アイデアは興味深い。

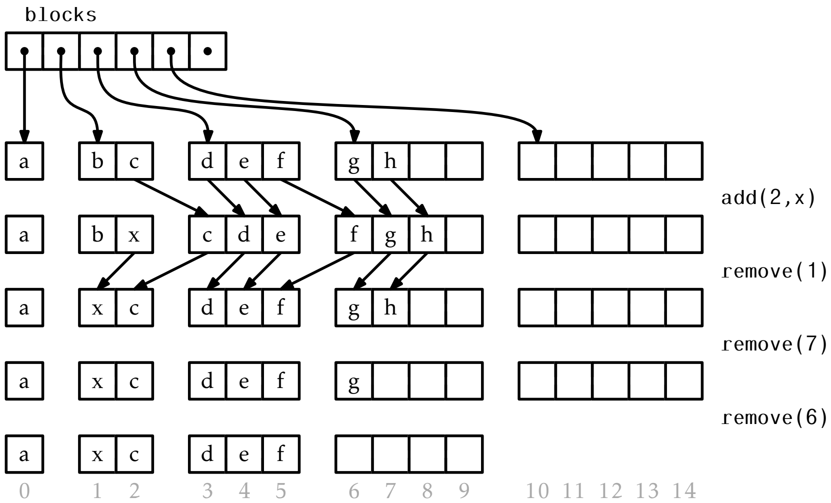


図 2.5: `RootishArrayStack` に対する `add`(i,x)・`remove`(i) の実行例。矢印は要素のコピーを表す。

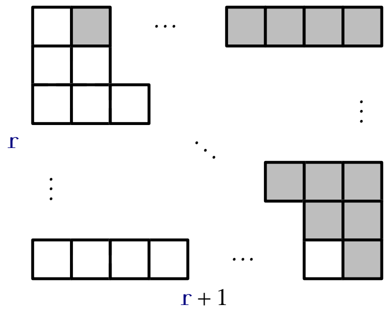


図 2.6: 白い正方形の数は合わせて $1+2+3+\cdots+r$ である。斜線を引いた正方形の数も同じである。白い正方形と斜線を引いた正方形を合わせてできる、正方形全体は、$r(r+1)$ 個の正方形からなる。

れる。全体で $\mathtt{i}$ 番目の要素がどのブロックのどの位置に入っているかわかるだろうか？

$\mathtt{i}$ 番目の要素がどのブロックに入っているかさえ分かれば、ブロック内での位置は簡単に計算できる。インデックス $\mathtt{i}$ の要素が $\mathtt{b}$ 番目のブロックに入っているなら、$0,\ldots,\mathtt{b}-1$ 番目のブロックにおける要素数の合計は $\mathtt{b}(\mathtt{b}+1)/2$ である。そのため $\mathtt{i}$ は

$$\mathtt{j} = \mathtt{i} - \mathtt{b}(\mathtt{b}+1)/2$$

として $\mathtt{b}$ 番目のブロックの $\mathtt{j}$ 番目の位置に入っている[*12]。$\mathtt{i}$ から $\mathtt{b}$、つまり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。$\mathtt{i}$ 以下のインデックスを持つ要素は $\mathtt{i}+1$ 個ある。一方 $0,\ldots,b$ 番目のブロックに入っている要素数の合計は $(\mathtt{b}+1)(\mathtt{b}+2)/2$ である。よって $\mathtt{b}$ は次の式を満たす最小の整数である。

$$(\mathtt{b}+1)(\mathtt{b}+2)/2 \geq \mathtt{i}+1$$

この式は次のように変形できる。

$$\mathtt{b}^2 + 3\mathtt{b} - 2\mathtt{i} \geq 0$$

2 次方程式 $\mathtt{b}^2 + 3\mathtt{b} - 2\mathtt{i} = 0$ はふたつの解 $\mathtt{b} = (-3+\sqrt{9+8\mathtt{i}})/2$ と $\mathtt{b} = (-3-\sqrt{9+8\mathtt{i}})/2$ を持つ。ふたつめの解は常に負の値なので捨ててよい。よって、解は $\mathtt{b} = (-3+\sqrt{9+8\mathtt{i}})/2$ である。この解は一般に整数とは限らない。しかし元の不等式に戻ると $\mathtt{b} \geq (-3+\sqrt{9+8i})/2$ を満たす最小の $\mathtt{b}$ が欲しかったのであった。これは次のように書ける。

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

RootishArrayStack

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

[*12] 訳注：たとえば 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 番目のブロックの 0 番目の要素である。

このインデックス 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。

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;
}
```

この章のデータ構造のどれかを使って blocks 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) はもう手慣れたものだろう。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 grow() を呼び出し新たなブロックを追加する。その後 $i,\ldots,n-1$ 番目の要素をそれぞれ右にひとつずらし、新たな i 番目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() メソッドはやってほしいことをしてくれる、つまり新しいブロックを追加してくれる。

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

grow() のコストを無視すれば、add(i,x) の実行時間はシフト操作の回数を数えれば十分で、これは $O(1+\mathtt{n}-\mathtt{i})$、すなわち ArrayStack と同じである。

remove(i) は add(i,x) に似ている。$\mathtt{i}+1,\ldots,\mathtt{n}$ 番目の要素をそれぞれ左にひとつずつシフトし、ふたつ以上の空のブロックがあれば shrink() を呼び出し、使われていないブロックをひと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) {
    blocks.remove(blocks.size()-1);
    r--;
```

```
    }
}
```

ここでもまた shrink() のコストを無視すれば、remove(i) の実行時間はシフト操作の回数を数えれば十分で、これは $O(n-i)$ である [*13]。

2.6.1 拡張・収縮の分析

上の add(i,x)・remove(i) の解析では grow()・shrink() のことを考慮していなかった。まず、ArrayStack.resize() とは違い、grow() と shrink() は要素をコピーしないことに注意する。つまり大きさ r の配列を割り当て・解放するだけである。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()・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、その他のブロックは一杯である。そのため、次の grow()・shrink() が呼ばれるのは、少なくとも $r-1$ 回要素が追加・削除された後である。よって、grow()・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)・remove(i) で償却され、grow()・shrink() の償却コストは $O(1)$ である。

2.6.2 領域使用量

つづいて 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の数を数えたい。これを**無駄な領域**ということにする。

remove(i) は RootishArrayStack が空きのあるブロックを高々 2 つしか持たないことを保証する。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。

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

ここでまた二次式の解を考えれば次の式が成り立つ。

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

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

末尾のブロックふたつの大きさは $\mathtt{r}$ と $\mathtt{r}-1$ なので、これらのブロックによる無駄な領域の量は $2\mathtt{r}-1=O(\sqrt{\mathtt{n}})$ である。もしこれらのブロックを（例えば）ArrayStack に入れれば、$\mathtt{r}$ 個のブロックを入れる List による無駄な領域の量も $O(\mathtt{r})=O(\sqrt{\mathtt{n}})$ である。$\mathtt{n}$ の値やその他の情報を保持するのに使うその他の領域は $O(1)$ である。以上より、RootishArrayStack の無駄な領域の量は合計 $O(\sqrt{\mathtt{n}})$ である。

この空間領域量は、空からはじまり要素をひとつずつ追加できるデータ構造の中で最適であることを示す。正確にいうと、$\mathtt{n}$ 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）$\sqrt{\mathtt{n}}$ 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 $\mathtt{n}$ 個の要素をひとつずつ追加していくとする。完了したときには、$\mathtt{r}$ 個のブロックに分散して $\mathtt{n}$ 個のアイテムがデータ構造に入っている。$\mathtt{r}\geq\sqrt{\mathtt{n}}$ なら、$\mathtt{r}$ 個のブロックを追跡するために $\mathtt{r}$ 個のポインタ（参照）を使い、ポインタは無駄な領域である [*14]。一方で $\mathtt{r}<\sqrt{\mathtt{n}}$ なら鳩の巣原理より大きさ $\mathtt{n}/\mathtt{r}>\sqrt{\mathtt{n}}$ 以上のブロックが存在する。このブロックがはじめて割当てられた瞬間を考える。このブロックは割当てられたとき空なので、$\sqrt{\mathtt{n}}$ の無駄な領域が生じている。以上より、$\mathtt{n}$ 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は $\sqrt{\mathtt{n}}$ の無駄な領域を生じる。

2.6.3 要約

次の定理は RootishArrayStack についての議論をまとめたものだ。

定理 **2.4.** *RootishArrayStack* は *List* インターフェースを実装する。grow()・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、

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

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

要素数 $\mathtt{n}$ の *RootishArrayStack* が使う（ワード単位で測った）使用領域

[*14] 訳注：たとえば続く第 3 章では、この考えを更に進めて $\mathtt{r}$=$\mathtt{n}$ 個のブロックを追跡するために $\mathtt{n}$ 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

第一章の計算モデルの説明を覚えていれば、RootishArrayStack は平方根を計算しているため、これまで使ってきたワード RAM モデル（1.4 節）に基づけばまだ平方根の計算方法について検討する必要があるとに気づいたかもしれない。[*16]。平方根の算出は基本的な操作とは一般的にみなされておらず、ワード RAM モデルに含まれていない。

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。特に「長さが $O(\sqrt{n})$ の 2 つの配列 sqrttab と logtab を作り、実行時間が $O(\sqrt{n})$ である前処理」のあとで、どんな自然数 $x\in\{0,\ldots,n\}$ についても定数時間で $\lfloor\sqrt{x}\rfloor$ を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′ 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. 二つの数 $x\ge 1$ と $x'=x-a$ について、$0\le a\le\sqrt{x}$ だと仮定する。このとき、$\sqrt{x'}\ge\sqrt{x}-1$ である。

証明. 以下を示せばよい。

$$\sqrt{x-\sqrt{x}}\ge\sqrt{x}-1\enspace .$$

両辺の二乗を取ると、

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

となり、整理すると

$$\sqrt{x}\ge 1$$

となる。これはどんな $x\ge 1$ についても成り立つ。 □

あらゆる自然数 $x\in\{0,\ldots,n\}$ の前に、まずは一部の x の平方根を計算する方法から始めよう。自然数 x が $2^r\le x<2^{r+1}$ を満たす、すなわち $\lfloor\log x\rfloor=r$

[*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：計算モデルとは計算を理論的に調べるための道具である。我々はここまで w ビットのワード RAM 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 節で言及したように、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。すなわち我々は平方根の算出の実行時間をまだ知らない。

である場合を考える。このとき自然数 x は 2 進数 $\mathtt{r}+1$ ビットで表せる [*17]。ここで、$\mathtt{x}' = \mathtt{x} - (\mathtt{x} \bmod 2^{\lfloor r/2 \rfloor})$ について考えると、この $\mathtt{x}'$ は補題 2.3 を満たすため $\sqrt{\mathtt{x}} - \sqrt{\mathtt{x}'} \le 1$ が成り立つ。更に、$\mathtt{x}'$ の下位 $\lfloor \mathtt{r}/2 \rfloor$ ビットは 0 である。すると残りのビット数から考えて、$\mathtt{x}'$ は

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

通りの値しか取れない [*18]。これは $\lfloor \sqrt{\mathtt{x}'} \rfloor$ の値を入れておく配列 sqrttab を [*19]。もう少し正確に言えば、配列 sqrttab の各要素の値は

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

である。こうすることで、各要素 sqrttab[i] はあらゆる $\mathtt{x} \in \{i2^{\lfloor r/2 \rfloor}, \ldots, (i+1)2^{\lfloor r/2 \rfloor} - 1\}$ について、$\sqrt{\mathtt{x}}$ の値からおよそ 2 しか離れていない。言い換えれば、配列要素 $\mathtt{s} = \mathtt{sqrttab}[\mathtt{x}{>>}\lfloor \mathtt{r}/2 \rfloor]$ について、$\mathtt{s}$ は $\lfloor \sqrt{\mathtt{x}} \rfloor$ か、$\lfloor \sqrt{\mathtt{x}} \rfloor - 1$ か、$\lfloor \sqrt{\mathtt{x}} \rfloor - 2$ のいずれかの値を取る [*20]。$(\mathtt{s}+1)^2 > \mathtt{x}$ となるまで $\mathtt{s}$ 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 $\lfloor \sqrt{\mathtt{x}} \rfloor$ の値を特定できる。

FastSqrt

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

[*17] 訳注：2 進数表記とは 0 と 1 というふたつの数字を用いて整数を表す表記法である。10 進法表記の 0, 1, 2, 3, 4 は、2 進法表記で 3 ビットを用いると 000, 001, 010, 011, 100 と表記される。コンピュータは 0 と 1 のふたつの状態を取れる部品を用いて数を表現するため、ある数を 2 進数表記する際に何ビット必要なのかは重要である。

[*18] 訳注：中辺は単に左辺に 2 をかけて導出し、中辺から右辺にかけては $2^{\mathtt{r}} \le \mathtt{x} < 2^{\mathtt{r}+1}$ から導出される $2^{\mathtt{r}/2} \le \sqrt{\mathtt{x}}$ を用いた。右辺から、$\mathtt{x}'$ の可能な値の数が $O(\sqrt{\mathtt{n}})$ でバウンドされていると分かる。

[*19] 訳注：つまり、平方根はプログラム中で何千回も使いまわされる処理でありうるのだから、その処理を高速化できるように、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。$\mathtt{x}'$ の数が高々 $2^{\mathtt{r}+1-\lfloor \mathtt{r}/2 \rfloor}$ 通り（たと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）しかない、ビッグオー記法で表記すれば $2^{\mathtt{r}+1-\lfloor \mathtt{r}/2 \rfloor} \le 4\sqrt{\mathtt{x}}$ という結果から $O(\sqrt{\mathtt{n}})$ 通りしかないのだから、この工夫は実を結ぶ。

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

```
  return s;
}
```

ここまでは $\mathtt{x} \in \{2^{\mathtt{r}},\dots,2^{\mathtt{r}+1}-1\}$ の場合についてのみ考えてきた。また、sqrttab は $\mathtt{r} = \lfloor \log \mathtt{x} \rfloor$ についてのみ使えるものであった。これを一般化するには $\lfloor \log \mathtt{n} \rfloor$ 個の sqrttab を $\lfloor \log \mathtt{x} \rfloor$ 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 $4\sqrt{\mathtt{n}}$ である。そのため、すべての sqrttab の大きさを合計すると $O(\sqrt{\mathtt{n}})$ である。

しかし実は sqrttab はひとつだけで十分なのである。$\mathtt{r} = \lfloor \log \mathtt{n} \rfloor$ の場合の sqrttab だけがあればよいのだ。

$\log \mathtt{x} = \mathtt{r}' < \mathtt{r}$ である x については $2^{\mathtt{r}-\mathtt{r}'}$ をかけてアップグレードし、次の等式を使えばよい。

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

$2^{\mathtt{r}-\mathtt{r}'}x$ は $\{2^{\mathtt{r}},\dots,2^{\mathtt{r}+1}-1\}$ に含まれるので、上の値は sqrttab に入っている。次のコードは $\{0,\dots,2^{30}-1\}$ の任意の要素 x について、大きさ 2^{16} の配列 sqrttab を使って $\lfloor\sqrt{\mathtt{x}}\rfloor$ 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 has r or r-1 bits
  int s = sqrtab[xp>>(r/2)] >> (upgrade/2);
  while ((s+1)*(s+1) <= x) s++;  // executes at most twice
  return s;
}
```

$\mathtt{r}' = \lfloor \log \mathtt{x} \rfloor$ 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 $2^{\mathtt{r}/2}$ の配列 logtab を使う。$\lfloor \log \mathtt{x} \rfloor$ は x を二進法で表記したとき 1 である最大の桁の添え字であることさえわかれば、実装は難しくない。すなわち $\mathtt{x} > 2^{\mathtt{r}/2}$ のとき x を $\mathtt{r}/2$ ビットだけ右にシフトし、logtab の添え字とする。次のコードは $\{0,\dots,2^{32}-1\}$ の任意の要素 x について、大きさ 2^{16} の配列 logtab を使って $\lfloor \log \mathtt{x} \rfloor$ 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];
}
```

最後になるが、logtab および sqrttab を初期化するコードも掲載しておく。

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

まとめると、ワード RAM では $O(\sqrt{n})$ だけのメモリを使って配列（sqrttab や logtab）を作れば、i2b(i) で使う各操作は実行時間で実行できる。この配列は n が二倍または二分の一の大きさになるたび拡大・縮小してもよい。この場合実行時間は ArrayStack のときと同様に add(i,x)・remove(i) の実行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 [46, Section 2.2.2] では、一般化すると ArrayStack・ArrayQueue・ArrayDeque になるスタック・キュー・双方向キューの実装を述べている。

RootishArrayStack を記述し 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 et al. [13] である。彼らはこの章で説明したものとは別の巧み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) の中で平衡根の計算をせずに済む。このやり方では i 番目の要素を含むブロックは $\lfloor\log(\mathtt{i}+1)\rfloor$ 番目のもので、これは $\mathtt{i}+1$ の二進表現における最高位の桁である。コンピュータ・アーキテクチャによってはこの計算を行うための命令があり、これは効率的に計算できる。Java では、Integer クラスの numberOfLeadingZeros(i) メソッドにより、簡単に $\lfloor\log(\mathtt{i}+1)\rfloor$ を計算できる。

RootishArrayStack に関連するデータ構造として、Goodrich and Kloss [35] の二段階の**階層ベクトル** (tiered-vector) がある。このデータ構造では get(i,x)・set(i,x) の実行時間は定数である。add(i,x)・remove(i) の実行時間は $O(\sqrt{\mathtt{n}})$ である。問 2.10 で扱う RootishArrayStack をさらに改良した実装では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 Collection c の要素をすべてリストの i 番目の位置に順に挿入する。(add(i,x) は $\mathtt{c}=\{\mathtt{x}\}$ 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 addAll(i,c) を add(i,x) 繰り返し実行して実装するのはなぜ効率がよくないのかを説明せよ。またより効率的な実装を考え、実装せよ。

問 **2.2.** RandomQueue を設計・実装せよ。これは Queue インターフェースの実装で、remove() 操作はそのときキューに入っている要素から一様な確率でひとつ選んで取り出すものである。(RandomQueue は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)・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 の実装であって、get(i)・set(i,x) は定数時間で実行でき、add(i,x)・remove(i) の実行時間は次のように表せるものだ。

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

つまり、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.** rotate(a,r) 操作を実装せよ。配列 a を「回転」する、すなわち

$\mathtt{i} \in \{0,\ldots,\mathtt{a.length}\}$ のすべてについて $\mathtt{a}[\mathtt{i}]$ を $\mathtt{a}[(\mathtt{i}+\mathtt{r}) \bmod \mathtt{a.length}]$ に動かすものだ。

問 **2.5.** List の回転操作 rotate(r) を実装せよ。これはリストの i 番目の要素を $(\mathtt{i}+\mathtt{r}) \bmod \mathtt{n}$ 番目に移す。ただし ArrayDeque や DualArrayDeque に対しての rotate(r) の実行時間は $O(1+\min\{\mathtt{r},\mathtt{n}-\mathtt{r}\})$ 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は高速な System.arraycopy(s,i,d,j,n) を利用して実現すること。

問 **2.7.** % 演算を用いずに ArrayDeque を実装せよ。（この演算に多くの時間がかかる環境があるのだ。）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。

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

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・末尾の外にはみ出たときは、rebuild() 操作を実行する。全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。データがどちらの端からもハミ出ない状態に $\mathtt{n}/2$ 回以下の操作で辿りつ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、無駄な領域量は $O(\sqrt{\mathtt{n}})$ だが、add(i,x)・remove(i,x) の実行時間が $O(1+\min\{\mathtt{i},\mathtt{n}-\mathtt{i}\})$ であるデータ構造を設計・実装せよ。

問 **2.10.** RootishArrayStack を修正し、無駄な領域量は $O(\sqrt{\mathtt{n}})$ だが、add(i,x)・remove(i,x) の実行時間が $O(1+\min\{\sqrt{\mathtt{n}},\mathtt{n}-\mathtt{i}\})$ 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、無駄な領域量は $O(\sqrt{\mathtt{n}})$ だが、add(i,x)・remove(i,x) の実行時間が $O(1+\min\{\mathtt{i},\sqrt{\mathtt{n}},\mathtt{n}-\mathtt{i}\})$ 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って、無駄な領域量が $O(\mathtt{n}^{2/3})$ であるものだ。`get(i)`・`set(i,x)` は定数時間で実行できる。`add(i,x)`・`remove(i)` の償却実行時間は $O(\mathtt{n}^{1/3})$ 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・(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 インターフェースを実装するために、連結リスト・配列のいずれを使うのがよいかは場合による。連結リストの短所はどんな要素の get(i)・set(i,x) も定数時間で行えるわけではないことだ。配列とは違って i 番目の要素を読み書きする際にリストをひとつずつ辿らなければならないのである。連結リストの長所は動的な操作がしやすいことだ。ノードの参照 u があれば、u を削除したり、u の隣にノードを挿入したりするのにかかる時間は定数である。このとき u がリストの中のどのノードであってもよいのだ [*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 の列である。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 w.next = null である。

[*1] 訳注：第 2 章で紹介された backing array を用いたデータ構造とは対照的である。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。

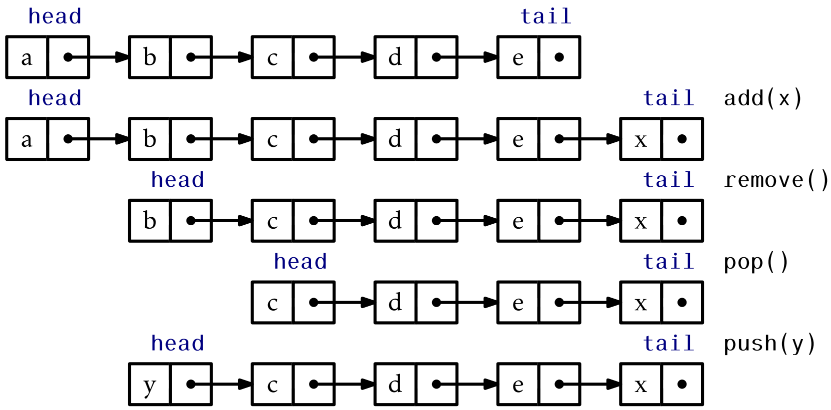


図 3.1: SLList における、Queue 操作（add(x)・remove()）と、Stack 操作（push(x)・pop()）

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

効率のため SLList は変数 head・tail で列の先頭・末尾のノードへの参照を保持している。また n は列の長さを表している。

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

SLList における Stack・Queue 操作を図 3.1 に示した。

SLList を使って Stack の push(x)・pop() を効率的に実装できる。列の先頭に追加・削除すればよいのである。push(x) は新しいノード u を作り、データ値に x を設定し、u.next を古い先頭とし、u を新しい先頭にする。最後に SLList の要素がひとつ増えたので、n を 1 だけ大きくする。

SLList

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

pop() では、SLList が空でないことを確認し、head = head.next として先頭を削除し、n を 1 だけ小さくする。最後の要素が削除される場合は特別で、tail を null に設定する必要がある。

SLList

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

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

3.1.1 キュー操作

SLList を使って FIFO キューの操作 add(x)・remove() を定数時間で実行することもできる。削除はリストの先頭から行われるので、pop() と同じである。

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

一方で要素の追加はリストの末尾に対して行う。u を新たに加えるノードとすると、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 となっている [*2]。この場合、tail も head も u になる。

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

明らかに add(x)・remove() はいずれも定数時間で実行できる。

[*2] 訳注：つまり tail が null で tail.next にアクセスするとエラーとなるため、別の対応が必要となる。

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*・*(FIFO) Queue* インターフェースの実装である。push(x)・pop()・add(x)・remove() の実行時間はいずれも $O(1)$ である。

SLList は Deque の操作をほぼすべて実装している。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が、これは新しい末尾を現在の末尾のひとつ前のノードに設定しなければならないためである。末尾のひとつ前のノード w とは w.next = tail であるもののことだ。困ったことに w を見つけるには SLList を head から順に $n-2$ 個のノードだけ辿っていかなければならないのである。

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 に似ている。違いはノード u が直後のノード u.next への参照と直前のノード u.prev への参照との両方を持っている点だ。

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

SLList を実装には特別扱いしなければならない場合がいくつかあった。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には head・tail をふつうと違うやり方で更新する必要があった。DLList ではこういう特別な場合がより多い。ダミーノードを使うと DLList のこれら特別な場合をシンプルに書ける。ダミーノードはデータを含まずただ場所だけを占める空のノードで、これを置くと特別扱いする必要のあるノードがなくなるのだ。すべてのノードには next と prev がある。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見なす。こうすると双方向連結リストのノードは図 3.2 に示すように循環する。

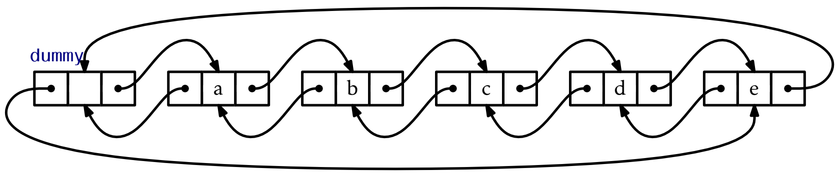


図 3.2: a,b,c,d,e からなる DLList

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

DLList から添え字を指定してノードを見つけるのは簡単だ。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。こうして i 番目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 = null;
  if (i < n / 2) {
    p = dummy.next;
    for (int j = 0; j < i; j++)
      p = p.next;
  } else {
    p = dummy;
    for (int j = n; j > i; j--)
```

```
      p = p.prev;
  }
  return p;
}
```

get(i)・set(i,x) もまた簡単である。i 番目の頂点を見つけ、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のうち支配的なのは i 番目のノードを見つける時間なので、実行時間は $O(1+\min\{\mathtt{i},\mathtt{n}-\mathtt{i}\})$ である。

3.2.1 追加と削除

DLList におけるノード w の参照を持っていて、ノード u を w の直前に追加したいときは、$\mathtt{u.next} = \mathtt{w}$、$\mathtt{u.prev} = \mathtt{w.prev}$ とし、u.prev.next・u.next.prev を適切に調整すればよい。（図 3.3 を参照せよ。）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すむ。

DLList
```
Node addBefore(Node w, T x) {
  Node u = new Node();
  u.x = x;
```

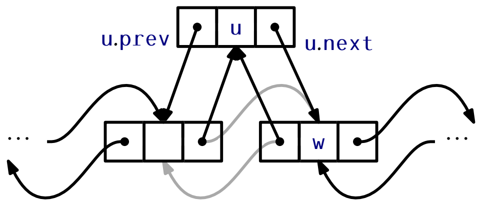


図 3.3: DLList において、u をノード w の直前に挿入する。

```
  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 番目のノードを見つけ、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 番目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・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のだ。ここでもまた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;
    n--;
}
```

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

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

getNode(i) によって i 番目のノードを見つける処理が支配的なので、remove(i) の実行時間は $O(1+\min\{i,n-i\})$ 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。

もし getNode(i) のコストを無視すると、DLList の操作の実行時間はいずれも定数であることは注目に値する。つまり DLList の操作における時間のかかる部分は、興味のあるノードを見つける処理だけなのである。興味のある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。そのときは興味のあるノードは定数時間で見つかるのだが、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、その結果として各処理は非定数時間であった。

このことから連結リストは何か別の方法でノードの参照が得られるアプリ

ケーションに適している。 Java Collections Framework における LinkedHashSet はこの例である。このデータ構造では、集合の要素は双方向連結リストに格納され、双方向連結リストのノードはハッシュテーブルに格納されている。(ハッシュテーブルについては、5 章で説明する。) LinkedHashSet から要素を削除するときは、ハッシュテーブルから対象となるノードを定数時間で見つけ、リストからもそのノードは削除される。(これも定数時間で実行できる。)

3.3 SEList:空間効率のよい連結リスト

連結リストの欠点はそのメモリ使用量である。(リストの真ん中に近い要素へのアクセスに時間がかかるのも欠点だが。) DLList のノードはみな前後合わせてふたつの参照を持つ。Node のフィールドのうちふたつはリストを維持するために占められ、残りのひとつだけがデータを入れるのに使われるのである。

SEList(space-efficient list) はシンプルなアイデアでこの無駄な領域を削減する。DLList のように一個ずつ要素を入れるのではなく、複数の要素を含むブロック(配列)をデータとして入れるのである。もう少し正確に説明する。SEList のパラメータとして**ブロックサイズ** 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。

後で詳しく説明するが、個々のブロックには Deque の操作を実行できると便利だ。このために BDeque (bounded deque) というデータ構造を使うことにする。これは 2.4 節で説明した ArrayDeque みたいなものだ。BDeque は ArrayDeque と少しだけ違う。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、その後拡大も縮小もされない。BDeque の重要な特徴は先頭・末尾の要素を追加・削除する操作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のに役立つ。

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

```
  void resize() { }
}
```

SEList はブロックの双方向連結リストである。

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

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

3.3.1 必要なメモリ量

SEList はブロックに含む要素数に次のような強い制限がある。末尾でないブロックはみな $b-1$ 個以上 $b+1$ 個以下の要素を含む。これはつまり 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] 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かと思われる。
[*4] 訳注：最初の項 b は末尾のブロック内の無駄な領域から来ている。

3.3.2 要素を検索

SEList の最初の課題はリストの i 番目の要素を見つけることである。要素の位置は次のふたつから決まる。

1. i 番目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 {
  Node u;
  int j;
  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ために DLList のときと同じ方法を使う。目的のノードを、先頭から順方向にあるいは末尾から逆方向に探すのだ。唯一の違いはノードからノードに移る度にブロックをまるごとスキップすることになる点である。

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

```
    Node u = dummy;
    int idx = n;
    while (i < idx) {
      u = u.prev;
      idx -= u.d.size();
    }
    return new Location(u, i-idx);
  }
}
```

ひとつ以下のブロックを除いて、すべてのブロックの要素数は $b-1$ 個以上であることを思い出してほしい。そのため全てのステップで探している要素に最低 $b-1$ 個の要素ずつ近づいていく。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。このふたつの値の小さい方がこのアルゴリズムの実行時間を決める。つまり、i 番目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 番目の要素を含むブロックを特定できたので、get(i)・set(i,x) はあとは目的のブロックの中の添え字を計算すればよい。

SEList

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

これらの操作の実行時間のうち i 番目の要素を含むブロックを探す時間が支配的なので、get(i) と set(i,x) の実行時間は $O(1+\min\{i,n-i\}/b)$ 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、末尾に要素を追加する操作 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ひとつも無い）ときは、新しいブロックを割当ててリストの末尾に追加する。すると末尾のブロックは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
```
boolean add(T x) {
  Node last = dummy.prev;
  if (last == dummy || last.d.size() == b+1) {
    last = addBefore(dummy);
  }
  last.d.add(x);
  n++;
  return true;
}
```

add(i,x) でリストの中に要素を追加するのはより複雑だ。まず i 番目の要素を入れるべきノード u を特定する。ここで問題になるのは、u のブロックが既に $b+1$ 個の要素を含んでおり、x を入れる隙間が無い場合である。

$u_0,u_1,u_2,\ldots$ がそれぞれ u, u.next, u.next.next... を表すとする。u_0 に x を入れるスペースを提供してくれるブロックを求めて、$u_0,u_1,u_2,\ldots$ 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 を参照せよ。）

1. すぐに（$r+1 \le b$ 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、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. すぐに（$r+1 \le b$ ステップ以内に）ブロックのリストの末尾に到達す

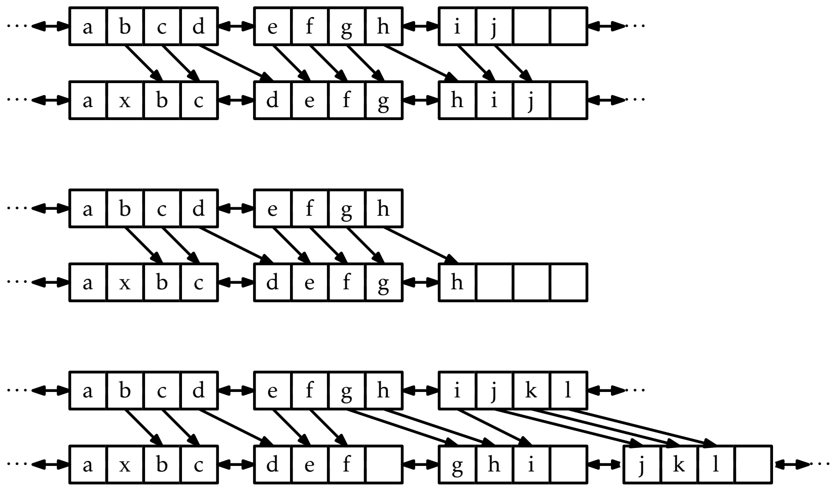


図 3.4: SEList において、要素 x を追加する際に起きる 3 つの場合。（この SEList ではブロックの大きさ b は 3 である。）

る。この場合、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、元々あった $b(b+1)$ 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ため、ここに x を挿入できる。

SEList

```
void add(int i, T x) {
  if (i == n) {
    add(x);
    return;
  }
  Location l = getLocation(i);
  Node u = l.u;
```

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

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。最初のふたつの場合は最大 b ブロックにわたって要素を探しシフトするので、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 $b(b+1)$ 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 番目の位置に要素 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。まずは i 番目の要素を含むノード u を特定する。そして u から要素を削除すると 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の対策が必要だ。

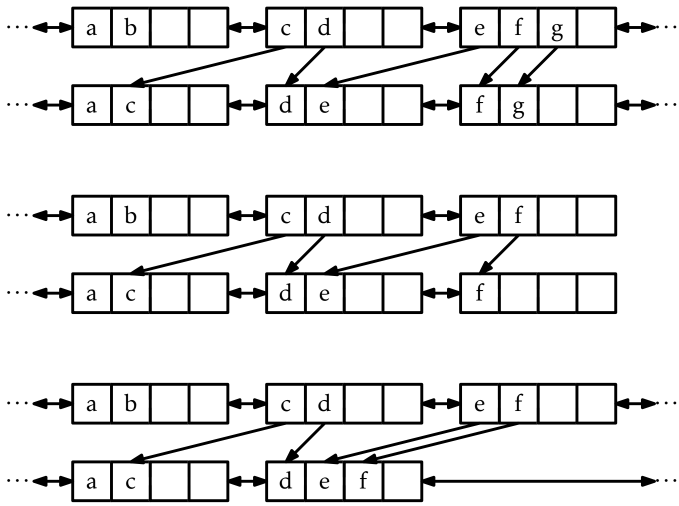


図 3.5: SEList において、要素 x を削除する際に起きる 3 つの場合。（この SEList ではブロックの大きさ b は 3 である。）

ここでもまた $u_0, u_1, u_2, \ldots$ は u, u.next, u.next.next, ... を表すとする $u_0, u_1, u_2, \ldots$ を順に u_0 のブロックの要素数を $b-1$ 以上にするために要素をもらえるノードを探す。ここでも考えられる 3 つの可能性がある。（図 3.5 を参照せよ。）

1. すぐに（$r+1 \le b$ 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、r 回のシフトで要素をあるブロックから後方のブロックに送り、u_r の余剰の要素を 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. すぐに（$r+1 \le b$ ステップ以内に）リストの末尾に到達する。この場合、u_r は末尾のノードなので u_r 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。そのためひと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。この結果 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。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 要素を含むようになる。そして空になった u_{b-1} を削除する。すると、u_0 のブロックは b 要素を含むようになったので、ここから適当な要素を削除できる。

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

add(i,x) と同様に、3 つめの場合での gather(u) を無視すれば、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)・remove(i) で実行されるかもしれない gather(u)・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);
}
```

いずれの実行時間においても支配的なのは二段階ネストしたループである。内側・外側いずれのループも最大 $b+1$ 回実行されるのでいずれの操作の実

行時間も $O((\mathtt{b}+1)^2) = O(\mathtt{b}^2)$ である。しかし、次の補題によってこれらのメソッドは、add(i,x)・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、$m \ge 1$ 回 add(i,x)・remove(i) が実行されるこのとき spread()・gather() に要する時間の合計は $O(\mathtt{b}m)$ 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 $\mathtt{b}-1$ または $\mathtt{b}+1$ 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。しかし remove(i)・gather(u) の解析も同様である。

add(i,x) のひと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 $\mathtt{u}_r$ ひとつだけである。よって高々一つのノードだけが安定から不安定になる。ふたつめの場合わ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ひとつだけ増える。以上よりひとつめふた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 つめの場合わけでは $\mathtt{u}_0,\ldots,\mathtt{u}_{\mathtt{b}-1}$ 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 $\mathtt{b}+1$ 個の安定なノードに置き換えられる。そして x が $\mathtt{u}_0$ のブロックに追加され、$\mathtt{u}_0$ は不安定になる。合わせてポテンシャルは $\mathtt{b}-1$ 減少する。

まとめると、ポテンシャルは 0 からはじまる。（リストに一つもノードがない状態である。）ケース１・２では、ポテンシャルは高々 1 増える。ケース３ではポテンシャルは $\mathtt{b}-1$ 減る。不安定なノードの数であるポテンシャルは、0 より小さくなることはない。つまり、ケース３が起きるたびに、少なくとも $\mathtt{b}-1$ 回のケース１・２が起きる。以上より spread(u) が呼ばれる毎に、少なくとも b 回 add(i,x) が呼ばれていることが示された。 □

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)・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。

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

さらに、空の *SEList* からはじめて、$\mathtt{add(i,x)}$・$\mathtt{remove(i)}$ からなる m 個の操作の列における、$\mathtt{spread(u)}$・$\mathtt{gather(u)}$ の実行時間は合わせて $O(\mathtt{b}m)$ である。

要素数 $\mathtt{n}$ の *SEList* における（ワード単位で測った）[*5] 領域使用量は $\mathtt{n}+O(\mathtt{b}+\mathtt{n}/\mathtt{b})$ である。

SEList により ArrayList と DLList の間のトレードオフを調整できる。ブロックの大きさ $\mathtt{b}$ によって、ふたつのデータ構造のどちらに寄せるかを調整できるのである。極端な場合として $\mathtt{b}=2$ のとき、SEList のノードは最大 3 つの値を持ち、これは DLList と同じである。もう一方の極端な場合として $\mathtt{b}>\mathtt{n}$ のとき、すべての要素は一つの配列に格納され、これは ArrayList みたいなものだ。これらの間の調整は、リストへの要素の追加・削除の時間と、特定の要素を見つける時間のトレードオフでもある。

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、研究され尽くしているテクニックである。例えば Knuth の [46, Sections 2.2.3–2.2.5] で議論されている。SEList もデータ構造の有名な練習問題である。SEList は unrolled linked list[69] と呼ばれることもある。

双方向連結リストの領域使用量を減らすための別の手法として XOR-lists と呼ばれるものもある。XOR-list では各ノード $\mathtt{u}$ はひとつだけのポインタ $\mathtt{u.nextprev}$ を持つ。 このポインタは $\mathtt{u.prev}$ と $\mathtt{u.next}$ の XOR を取ったものである。リストは、$\mathtt{dummy}$ を指すポインタと $\mathtt{dummy.next}$ を指すポインタの二つを持つ必要がある。（$\mathtt{dummy.next}$ はリストが空なら $\mathtt{dummy}$ を、そうでないなら先頭のノードを指す。）このテクニックは $\mathtt{u}$ と $\mathtt{u.prev}$ があれば $\mathtt{u.next}$ を次の関係式から計算できることを利用している。

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

（ここで^はふたつの引数の排他的論理和を計算する。）このテクニックはコー

[*5] 1.4 節で説明したメモリの図り方の議論を思い出すこと。

ドを少し複雑すること、Java や Python などガーベッジコレクションのある言語では使えないことは欠点である。 XOR-list のもっと踏み込んだ議論は Sinha の雑誌記事 [70] を参照してほしい。

問 **3.1.** SLList においてダミーノードを使って push(x)・pop()・add(x)・remove() の全ての特殊なケースを避けることができないのは何故か説明せよ。

問 **3.2.** SLList のメソッド secondLast() を設計・実装せよ。これは SLList の末尾の一つ前の要素を返すものだ。この実装の際にリストの要素数 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 $O(1+\mathtt{i})$ 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 $O(\mathtt{n})$ でなければならず、再帰は使ってはならない。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てもいけない。

問 **3.5.** SLList および DLList の 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操作だ。このメソッドはなにも返さないが、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、これをノード w の直前に追加する操作だ。この章を確認しながら実装してはならない。もしこの本のコードと完全に一致しなくともあなたの書くコードは正しいかもしれない。そのコードをテストし、正しく動くかどうかを確認せよ。

続くいくつかの問題は 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当ててはいけない。これらの問題はいずれもノードの prev・next を書き換えるだけで解くことができる。

問 **3.7.** DLList の操作 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、$i \in \{0,\ldots,\mathtt{n}-1\}$ のいずれの場合も i 番目の要素が $\mathtt{n}-i-1$ 番目の要素と等しいかどうかを確認する操作である。実行時間は $O(\mathtt{n})$ である必要がある。

問 **3.8.** rotate(r) を実装せよ。これは DLList の要素を回転する操作で、

i 番目の要素を $(\mathtt{i}+\mathtt{r}) \bmod \mathtt{n}$ 番目の位置に移動するものだ。実行時間は $O(1+\min\{\mathtt{r},\mathtt{n}-\mathtt{r}\})$ である必要があり、リスト内のノードを修正してはならない。

問 **3.9.** `truncate(i)` を実装せよ。これは `DLList` を i 番目で切り詰める操作のだ。この操作を実行すると、リストの要素数は i になり、$0,\ldots,\mathtt{i}-1$ 番目の要素だけが残る。返り値も別の `DLList` で、これは $\mathtt{i},\ldots,\mathtt{n}-1$ 番目の要素を含むものである。この操作の実行時間は $O(\min\{\mathtt{i},\mathtt{n}-\mathtt{i}\})$ である。

問 **3.10.** `DLList` の操作 `absorb(l2)` を実装せよ。これは別の `DLList` `l2` を引数に取り、`l2` を空にし、その中身を自分の要素として追加する。例えば `l1` が a,b,c を含み、`l2` が d,e,f を含むとき、`l1.absorb(l2)` を実行すると `l1` は a,b,c,d,e,f を含み、`l2` は空になる。

問 **3.11.** `deal()` を実装せよ。これは `DLList` から偶数番目の要素を削除し、それらの要素を含む `DLList` を返す操作だ。例えば `l1` が a,b,c,d,e,f を含むとき、`l1.deal()` を呼ぶと、`l1` 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.** `reverse()` を実装せよ。これは `DLList` の要素の順序を逆転する操作だ。

問 **3.13.** この問題は `DLList` を整列するマージソートというアルゴリズムを実装してみるものだ。マージソートは 11.1.1 節扱う。 要素の比較を `compareTo(x)` メソッドで行えば、`Comparable` インターフェースの実装である要素からなる任意の `DLList` を整列する際に実装を使いまわせる。

1. `DLList` の `takeFirst(l2)` を実装せよ。この操作は `l2` の先頭ノードを取り出しレシーバに追加するものだ。これは新しいノードを作らないことを除けば、`add(size(),l2.remove(0))` と等価である。
2. `DLList` の静的メソッド `merge(l1,l2)` を実装せよ。これはふたつの整列済みのリスト `l1`・`l2` を統合し、その結果を含む新たな整列済みリストを返す。この操作をすると `l1`・`l2` は空になる。例えば `l1` の要素は a,c,d、`l2` の要素は 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 `sort()` メソッドを実装せよ。これはマージソートを使ってリストの全ての要素を整列するものである。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 ならなにもしない。
（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ふたつのリスト l1 と l2 に分割する。
（c）再帰的に l1 を整列する。
（d）再帰的に l2 を整列する。
（e）最後に l1 と l2 を統合して一つの整列済みリストとする。

つづく数問は発展的なもので、要素が追加・削除される際に Stack・Queue の最小値がどうなるかについての理解を要求するものである。

問 **3.14.** 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() 操作も可能なものである。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。全ての操作の実行時間は定数である。

問 **3.15.** 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() 操作も可能なものである。全ての操作の償却実行時間は定数である。

問 **3.16.** 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。全ての操作の償却実行時間は定数である。

次の問題は領域効率のよい SLList の解析・実装の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 は push(x) ≡ add(size(), x) と pop() ≡ remove(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はいずれも b の値に依ら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 に依らない定数である SEList を設計・実装せよ。

問 **3.19.** ビット単位の排他的論理和^によってふたつの int 型の値を入れ替える方法を説明せよ。ただし、このときにみっ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オシャレで実用的なデータ構造を紹介する。スキップリストは List の実装であり、get(i)・set(i,x)・add(i,x)・remove(i) の実行時間はいずれも $O(\log n)$ である。SSet の実装でもあり、いずれの操作の期待実行時間も $O(\log \mathtt{n})$ である。

スキップリストの効率性のキモはランダム性である。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を要素を見つけるための経路の長さの期待値で表現する。コイントスにより決まる確率からこの期待値を計算する。ランダムなコイントス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では、L_0 は n 個の要素すべてを含む。L_0 から L_1 を作り、L_1 から L_2 を作り、という作業を次のように繰り返す。リスト L_r の要素は L_{r-1} の要素の部分集合である。L_{r-1} のどの要素を L_r に含むかを決めるために、各要素についてコインを投げる。表が出た要素を L_r に含める。リスト L_r が空なら繰り返しを終える。スキップリストの例を図 4.1 に示した。

スキップリストのノード x について、x の**高さ** (height) を x を含むリスト L_r の添え字 r のうち最大のものと定義する。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。少し考えると x は次の試行と関連していることがわかるだろう。

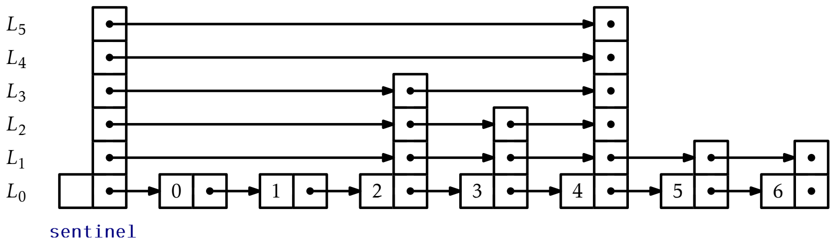


図 4.1: 7 つの要素を含む skiplist の例

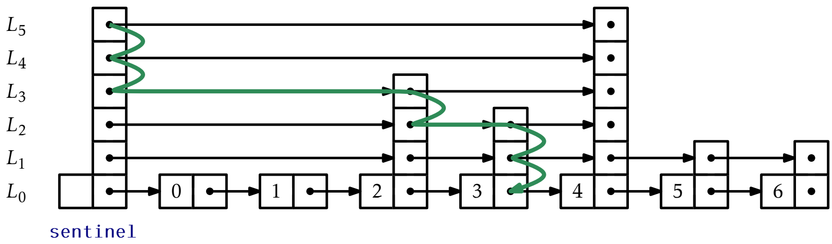


図 4.2: ある skiplist における、4 を含むノードの探索経路

裏が出るまでコインを繰り返し投げる。表は何回出るだろうか。この問いの答え、そして x の高さの期待値は 1 である。（コイントスの回数の期待値は 2 回だが、最後のトスは表でないため表が出る回数の期待値は 1 だ。）スキップリストの**高さ**とは、最も高いノードの高さである。

すべてのリストの先頭は特別で、**番兵**と呼ばれる。これはリストのダミーノードのようなものだ。スキップリストの重要な性質は、**探索経路** (search path) と呼ばれる 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。ノード u へのパスの作り方は簡単だ。（図 4.2 を参照のこと。）左上の端（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に進み続ける。u に到達する、または u を通し越してしまうときは右ではなく下に進む。

もうすこし正確に説明する。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。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、ひとつ下のリストに下がり、L_{h-1} の w から処理を続ける。これを L_0 における u の直前の要素にたどり着くまで繰り返す。

次の補題は探索経路が非常に短いことを主張する。（4.4 節で証明する。）

補題 4.1. L_0 のノード u について、u の探索経路長の期待値は $2\log \mathtt{n} + O(1) = O(\log \mathtt{n})$ である。

空間効率のよいスキップリストの実装方法を説明する。ノード u はデータ x・ポインタの配列 next を含む。u.next[i] で $L_{\mathtt{i}}$ における u の次のノードを指せばよい。こうすると x は複数のリストに現れるかもしれないが、ノードとしての実体はひとつだけあれば済む。

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

この章の続くふたつの節ではスキップリストの応用を紹介する。いずれの場合でも L_0 に主な構造（リストや整列された集合）を入れる。違いは探索経路の辿り方である。L_r にいるとき下（L_{r-1}）に向かうか（L_r のまま）右に進むかが異なることがある。

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。ここでは、L_0 は SSet の要素を整列して格納する。find(x) は探索経路に沿って $\mathtt{y} \geq \mathtt{x}$ を満たす最小の y を探す。

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

y の探索経路を辿るのは簡単だ。L_r の中のノード u にいるとすると、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の前に、新しいノードの高さ k を決めるためのコイントスをシミュレートする方法を考える。ランダムな整数 z を生成し、z の 2 進数表現において連続する 1 の数を数える。[*1]

SkiplistSSet
```
int pickHeight() {
  int z = rand.nextInt();
  int k = 0;
  int m = 1;
  while ((z & m) != 0) {
```

[*1] この方法はコイントスを完全に再現しているわけではない。なぜなら k は int のビット数より常に小さいからである。しかし要素数が $2^{32} = 4294967296$ を越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。

```
    k++;
    m <<= 1;
  }
  return k;
}
```

SkiplistSSet の add(x) の実装は、x を入れる場所を見つけ、高さ k を pickHeight() で決め、x を $L_0,\ldots,L_k$ に継ぎ合せる。これを実現する最も簡単な方法は、リスト L_r からリスト L_{r-1} に下がるノードを記録する配列 stack を使うことだ。より正確にいうと、stack[r] には探索経路において L_r から L_{r-1} に下がるノードが記録されている。x を挿入する時に修正する必要のあるノードはちょうど stack[0],...,stack[k] である。次のコードはこの add(x) アルゴリズムの実装である。

SkiplistSSet

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

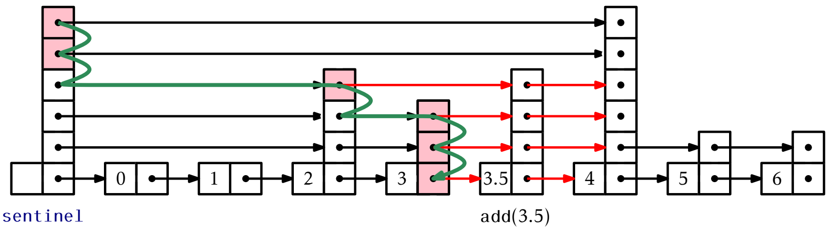


図 4.3: 値 3.5 を含むノードを skiplist に追加する。stack に格納されるノードを強調している。

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

要素 x の削除も同様に行える。ただし 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りながら行える。x を探す途中にノード u から下に向かうとき、u.next.x = x なら u を繋ぎ替える。

SkiplistSSet

```
boolean remove(T x) {
  boolean removed = false;
  Node<T> u = sentinel;
  int r = h;
  int comp = 0;
  while (r >= 0) {
    while (u.next[r] != null
           && (comp = compare(u.next[r].x, x)) < 0) {
      u = u.next[r];
    }
    if (u.next[r] != null && comp == 0) {
      removed = true;
      u.next[r] = u.next[r].next[r];
      if (u == sentinel && u.next[r] == null)
```

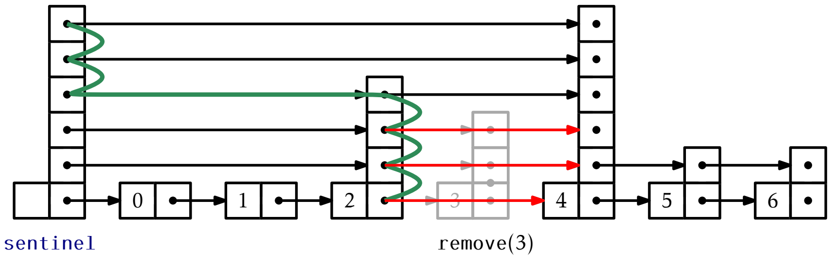


図 4.4: 値 3 を含むノードを skiplist から削除する

```
            h--;  // height has gone down
        }
        r--;
    }
    if (removed) n--;
    return removed;
}
```

4.2.1 要約

次の定理はスキップリストを使った整列集合の性能をまとめたものだ。

定理 4.1. *SkiplistSSet* は *SSet* インターフェースの実装である。*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だ。SkiplistList では、L_0 はリストの要素をリストにおける順序通りに含む。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。

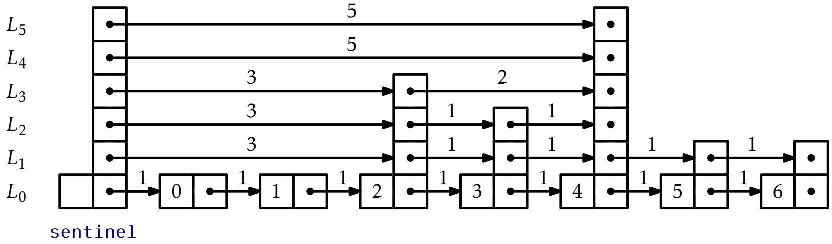


図 4.5: skiplist における辺の長さ

これを可能にするためにはまず L_0 における i 番目の要素を見つける方法が必要だ。このための最も簡単な方法はリスト L_r における**辺の長さ**を定義することだ。L_0 における辺の長さをいずれも 1 とする。$L_r(r > 0)$ の辺 e の辺の長さを、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 L_0 において e の下にある辺の数を e の長さとするのと等価な定義である。この定義の例として図 4.5 を参照せよ。スキップリストの辺は配列に格納されており、その長さも同様に格納すればよい。

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

この定義の良い性質は、L_0 において j 番目のノードから長さ ℓ の辺を辿る

と、L_0 において j + ℓ のノードに移ることだ。こうして、探索パスを辿りながら 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;   // index of the current node in list 0
  while (r >= 0) {
    while (u.next[r] != null && j + u.length[r] < i) {
      j += u.length[r];
      u = u.next[r];
    }
    r--;
  }
  return u;
}
```

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

get(i)・set(i,x) において最も計算時間がかかるのは L_0 の i 番目のノード

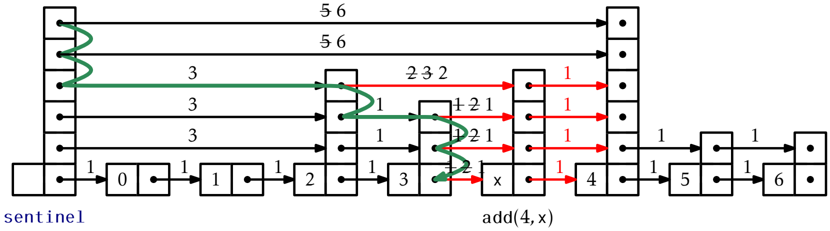


図 4.6: SkiplistList への要素の追加

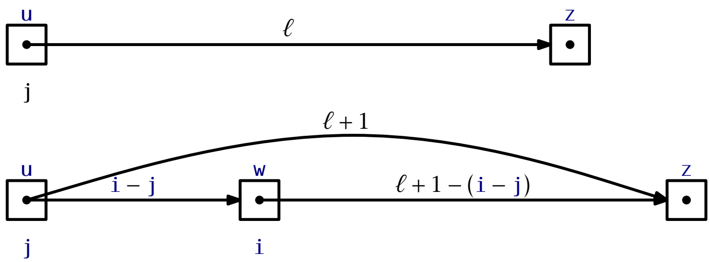


図 4.7: ノード w を skiplist に追加するときの、辺の長さの更新

を見つける処理なので、これらの操作の実行時間は $O(\log n)$ である。

SkiplistList の i 番目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 $r \leq k$ のときで、このとき w を L_r とを継ぎ合わせる。このとき辺の長さも適切に更新する必要があることに注意する。図 4.6 を見よ。

探索経路上でリスト L_r のノード u に降りたとき、i 番目の位置に要素を追加するため辺 u.next[r] の長さをひとつ大きくする。ノード w をふたつのノード u と z の間に追加する様子を図 4.7 に示す。探索経路を辿りながら L_0 において u が何番目なのかを数えられる。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 = new Node(x, pickHeight());
  if (w.height() > h)
    h = w.height();
  add(i, w);
}
```

SkiplistList

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

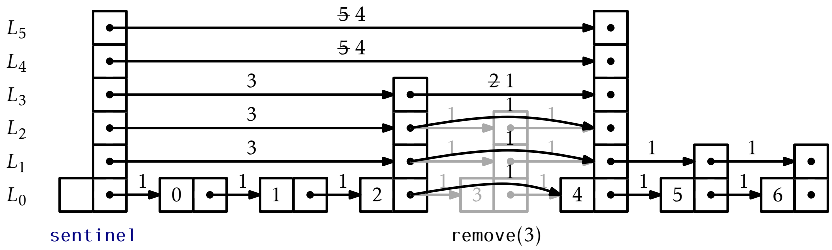


図 4.8: SkiplistList から要素を削除する。

```
}
```

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 番目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ひとつ小さくする。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 それを除く。図 4.8 に例が描かれている。

SkiplistList
```
T remove(int i) {
  T x = null;
  Node u = sentinel;
  int r = h;
  int j = -1; // index of node u
  while (r >= 0) {
    while (u.next[r] != null && j+u.length[r] < i) {
      j += u.length[r];
      u = u.next[r];
    }
    u.length[r]--;  // for the node we are removing
    if (j + u.length[r] + 1 == i && u.next[r] != null) {
      x = u.next[r].x;
      u.length[r] += u.next[r].length[r];
```

```
      u.next[r] = u.next[r].next[r];
      if (u == sentinel && u.next[r] == null)
        h--;
    }
    r--;
  }
  n--;
  return x;
}
```

4.3.1 要約

次の定理は SkiplistList の性能をまとめたものだ。

定理 **4.2.** *SkiplistList* は *List* インターフェースを実装する。*SkiplistList* 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 $O(\log \mathtt{n})$ である。

4.4 スキップリストの解析

この節では高さ・大きさ・探索経路の長さの期待値を解析する。ここでは基本的な確率論の知識を前提とする。いくつかの証明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。(表が出た回も含めて数えることに注意する。)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、次の指示変数を定義する。

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

$I_i=1$ なのは最初の $i-1$ 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。

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

次のふたつの補題ではスキップリストにおけるノード数は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 $2\mathsf{n}$ である。

証明. 要素 x がリスト L_{r} に含まれる確率は $1/2^{\mathsf{r}}$ である。よって L_{r} のノード数の期待値は $\mathsf{n}/2^{\mathsf{r}}$ 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。

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

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 $\log \mathsf{n} + 2$ 以下である。

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

$$I_{\mathsf{r}} = \begin{cases} 0 & L_{\mathsf{r}} \text{ が空のとき} \\ 1 & L_{\mathsf{r}} \text{ が空でないとき} \end{cases}$$

スキップリストの高さ h は次のように計算できる。

$$\mathsf{h} = \sum_{r=1}^{\infty} I_{\mathsf{r}}$$

I_{r} はリスト L_{r} の長さ $|L_{\mathsf{r}}|$ を越えないことに注意する。

$$\mathrm{E}[I_{\mathsf{r}}] \le \mathrm{E}[|L_{\mathsf{r}}|] = \mathsf{n}/2^{\mathsf{r}}$$

[*2] 指示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。

よって

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

□

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そうする。そうでないなら左に進む。少し考えると、x の逆探索経路は探索経路と方向が逆であることを除いて同じであ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 は次の試行と関連している。コインを投げる。表が出れば上に向かい停止する。裏が出れば左に向かい試行を続ける。このとき、表が出るまでにコインを投げる回数は逆探索経路の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 よりはじめて表

[*3] これは左に向かう回数を大きく数えてしまうかもしれない。なぜなら実際には試行は表が出るか番兵に出くわすかのどちらかが起きたときに終えるべきだからである。しかし今考えている補題は上界に関するものであるので、ここでは問題にならない。

が出るまでのコイントスの回数の期待値は 1 である。

S_r を（順方向の）探索経路における高さ r で右に進む回数を表す。$E[S_r] \le 1$ である。さらに L_r では L_r の長さより多く右に進むことはないので $S_r \le |L_r|$ である。よって次の式が成り立つ。

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

あとは補題 4.4 と同様に証明を完成できる。S をスキップリストにおけるノード u の探索経路の長差とする。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。

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

□

次の定理はこの節の結果をまとめるものだ。

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

スキップリストを提案したのは Pugh [62] であり、多くの拡張や応用も提案されている [61]。その後さらに多くの研究が行われている。スキップリストの i 番目の要素を見つける探索経路の長さの期待値や分散はより正確に求められている。 [45, 44, 58] 決定的な (ランダム性を用いない) 変種や [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。スキップリストは様々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使われている。[71, 63] スキップリストの変種がオペレーティング・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使われている。[42] Skiplists は Java 1.6 API の一部として含まれている。 [55]

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) を実行する時の振る舞いを説明せよ。

問 **4.5.** 図 4.5 の `SkiplistList` に `add`(3, `x`) を実行する時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.** L_{i-1} から L_i に要素を上げるかどうかを決めるとき、コイントスではなく確率 $p (0 < p < 1)$ を用いる。

1. このとき探索経路長の期待値は $(1/p)\log_{1/p} \mathtt{n} + O(1)$ 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) は**冗長な比較**を行うことがある。これは x と同じ値の比較を複数回行うことである。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。どのように冗長な比較が発生するかを説明し、find(x) においてこれ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) での比較操作の回数を解析せよ。

問 **4.9.** SSet における要素 x のランクとは、SSet の要素であって、x より小さいものの個数である。SSet インターフェースの実装であり、ランクによる要素への高速アクセスが可能なスキップリストを設計・実装せよ。これはランク i の要素を返す get(i) を持つ。この操作の実行時間は $O(\log n)$ である。

問 **4.10.** スキップリストの**指**とは探索経路において下に向かうノードを保存した配列である。（91 のコードで add(x) における変数 stack は**指**である。図 4.3 において影になっているノードは指を表している。）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 に到達するまで指の要素を見ていき、そして u からふつうの x の探索を実行する。L_0 において b と指が指す値と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。

Skiplist のサブクラス SkiplistWithFinger を実装せよ。これは find(x) を指を利用して実装する。このサブクラスでは指を保持し、指探索によって find(x) を実装する。find(x) は x の位置を返すと同時に、指が常に前回の find(x) の結果を指すように更新する。

問 **4.11.** truncate(i) を実装せよ。これは SkiplistList を i 番目の位置で切り詰める操作である。このメソッドを実行するとリストの大きさは i になり、リストは添え字 0,...,i − 1 の要素のみを含むようになる。返り値は SkiplistList であって、添え字 i,...,n − 1 の要素を含むものである。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 の操作 absorb(l2) を実装せよ。これは SkiplistList l2 を引数に取り、これを空にし、元々入っていた要素をそのままの順番でレシーバーに追加するものだ。例えば、l1 の要素が a,b,c、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。もし使った SSet の実装が n 要素を $O(n)$ のメモリだけを使って保持できるなら、SESSet は n 要素を格納ためのメモリに加えて、消費する無駄なメモリは $O(n/b+b)$ である。

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。このアプリはユーザーがクエリを入力するときテキストのマッチしている部分があればこれを結果として返す。
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のかを表す一つの整数でコンパクトに表現できる。
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] から本を入手できる。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べきだ。）スキップリストを二分探索木と比較せよ。

1. スキップリストの辺を削除することが、二分木のようにみえること、また二分探索木ににていることを説明せよ。
2. スキップリストと二分探索木は使うポインタの数は同じである。(ノードあたり 2 つ）しかしスキップリストの方がこれを上手く使っている。これは何故か、説明せよ。

第 5 章

ハッシュテーブル

ハッシュテーブルは大きな集合 $U = \{0,\ldots,2^{w}-1\}$ の要素 n 個（U の要素数と比べると、n は相対的に小さい整数）を格納するための効率的な方法だ。**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。この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。一方はチェインを使うもので、もう一方は線形探索を使うものだ。

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よくある。この場合**ハッシュ値** (hash code) というデータに対応する整数を使う。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。

この章で扱う手法は、ある範囲からランダムに生成した整数を必要とする。サンプルコードではこのランダムな整数はハードコードされた定数になっている。この定数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て作ったものだ[*1]。

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

ChainedHashTable とは**チェイン法** (chaining) を使ってデータをリストの配列 t に保存するデータ構造である。整数 n はすべてのリストにおける要素数の合計である。（図 5.1 を参照せよ。）

[*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、ということだけ把握すればこの本の理解には差し支えない。

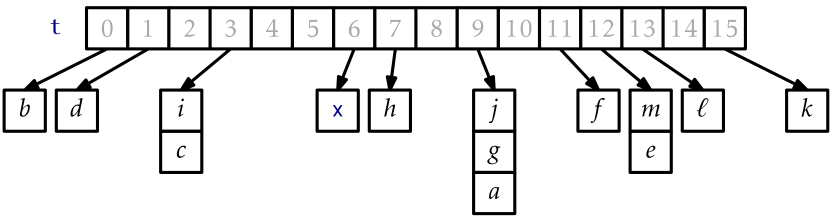


図 5.1: $n = 14$, $t.length = 16$ である ChainedHashTable の例。この例では $hash(x) = 6$ である。

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

データ x の**ハッシュ値** hash(x) とは $\{0,\ldots,t.length-1\}$ の中のある値である。ハッシュ値が i であるデータはリスト t[i] に入れられる。リストが長くなり過ぎないように、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（$n/t.length \le 1$）である。

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、必要があれば t を拡張する。あとは x から $\{0,\ldots,t.length-1\}$ 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
```
boolean add(T x) {
  if (find(x) != null) return false;
  if (n+1 > t.length) resize();
  t[hash(x)].add(x);
  n++;
  return true;
}
```

配列を拡張するときは、t の大きさを二倍にし、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、あのときと同じ結果が適用できる。すなわち、操作列についての拡張操作の償却実行時間は定数である。(必要なら 31 ページの補題 2.1 を見直すこと。)

拡張のあとは x を ChainedHashTable リスト t[hash(x)] に追加すればよい。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ためには、リスト t[hash(x)] を x が見つかるまで辿ればよい。

ChainedHashTable

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

この実行時間は $\mathtt{n}_\mathtt{i}$ をリスト t[i] の長さとするとき、$O(\mathtt{n}_{\mathtt{hash(x)}})$ である。

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

ChainedHashTable

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

```
}
```

これもリスト $\mathtt{t}[\mathtt{hash}(\mathtt{x})]$ 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影響される。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 $O(\mathtt{n}/\mathtt{t.length}) = O(1)$ 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 $\mathtt{t}[\mathtt{hash}(\mathtt{x})]$ の長さは n になってしまう。次の小節ではよ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 節で説明した）と整数の割り算からハッシュ値を計算する効率的な方法である。div は割り算の商を求める演算子である。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（これは**次数** (dimension) と呼ばれる）について大きさ $2^{\mathtt{d}}$ であるハッシュテーブルを使う。整数 $\mathtt{x} \in \{0,\ldots,2^{\mathtt{w}}-1\}$ のハッシュ値を次のように計算する。

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

ここで z は奇数の集合 $\{1,3,5,\ldots,2^{\mathtt{w}}-1\}$ からランダムに選択される。整数の演算は整数のビット数を w とするとき、$2^{\mathtt{w}}$ で勝手に剰余を取られることを利用すると、このハッシュ関数は非常に効率よく実現できる [*2]。（図 5.2 を参照せよ。）さらに、$2^{\mathtt{w}-\mathtt{d}}$ による整数の割り算は二進法で右側の $\mathtt{w}-\mathtt{d}$ ビットを落とせば計算できる。（これはビットを右に $\mathtt{w}-\mathtt{d}$ 個だけシフトすればよく、実装は上の式よりも単純になる。）

ChainedHashTable

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

[*2] ほとんどのプログラミング言語ではこうなのだが、残念なことに Ruby や Python などではそうではない。これらの言語では、w ビットの固定桁の演算結果が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るのである。

2^{w} (4294967296)	100000000000000000000000000000000
z (4102541685)	11110100100001111101000101110101
x (42)	00000000000000000000000000101010
$z \cdot x$	10100000011110010010000101110100110010
$(z \cdot x) \bmod 2^{w}$	00011110010010000101110100110010
$((z \cdot x) \bmod 2^{w}) \operatorname{div} 2^{w-d}$	00011110

図 5.2: $w = 32$、$d = 8$ とした乗算ハッシュ法の操作

次の補題は乗算ハッシュ法がうまくハッシュ値の衝突を避けることを示す。（証明はこの節の後半に回す。）

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 $\Pr\{\mathrm{hash}(x) = \mathrm{hash}(y)\} \leq 2/2^{d}$ 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、n_x を x がハッシュテーブルに現れる回数とするとき、リスト $t[\mathrm{hash}(x)]$ の長さの期待値は $n_x + 2$ 以下である。

証明. S をハッシュテーブルに含まれる x ではない要素の集合とする。要素 $y \in S$ について、次の指示変数を定義する。

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

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

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

$$
\begin{aligned}
&= \mathsf{n_x} + (\mathsf{n} - \mathsf{n_x})2/\mathsf{n} \\
&\le \mathsf{n_x} + 2
\end{aligned}
$$

□

続いて補題 5.1 を証明する。まずは整数論の定理からはじめる。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、$(b_r,\ldots,b_0)_2$ は二進表記で $b_r,\ldots,b_0$ である整数のことである。また、$\star$ は値の不明な桁を表すとする。

補題 5.3. S を $\{1,\ldots,2^{\mathtt{w}}-1\}$ 内の奇数の集合とする。q,i は S の任意の要素とする。このとき、$\mathtt{z}q \bmod 2^{\mathtt{w}} = i$ を満たす $\mathtt{z} \in S$ の要素が一意に存在する。

証明. $\mathtt{z}$ を選ぶと i は決まるので、$\mathtt{z}q \bmod 2^{\mathtt{w}} = i$ を満たす $\mathtt{z} \in S$ が一意に決まることを示せば良い。

背理法で示す。整数 $\mathtt{z}$ と $\mathtt{z}'$ が存在し $\mathtt{z} > \mathtt{z}'$ であると仮定する。このとき、

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

よって、

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

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。

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

これを 2 進数として考えると

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

なので、$(\mathtt{z}-\mathtt{z}')q$ の末尾 $\mathtt{w}$ 桁はすべて 0 である。

加えて、$q \ne 0$ かつ $\mathtt{z}-\mathtt{z}' \ne 0$ より $k \ne 0$ である。q は奇数なのでこの二進表記の末尾桁は 0 ではない。

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

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

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

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

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

以上より、$(\mathtt{z}-\mathtt{z}')q$ は (5.1) を満たさず、矛盾する。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。$\mathtt{z}$ が S から一様な確率でランダムに選ばれるとき、$\mathtt{zt}$ は S 上に一様分布する。次の証明では $\mathtt{z}$ の最下位の 1 である桁を除いた、$\mathtt{w}-1$ 桁のランダムなビットを考えるのがポイントだ。

補題 *5.1*. 条件 $\mathtt{hash(x)} = \mathtt{hash(y)}$ と「$\mathtt{zx} \bmod 2^{\mathtt{w}}$ の上位 $\mathtt{d}$ ビットと $\mathtt{zy} \bmod 2^{\mathtt{w}}$ の上位 $\mathtt{d}$ ビットが等しい」は同値である。この条件の必要条件は、$\mathtt{z}(\mathtt{x}-\mathtt{y}) \bmod 2^{\mathtt{w}}$ の上位 $\mathtt{d}$ ビットがすべて 0 である、またはすべて 1 であることである。これは、$\mathtt{zx} \bmod 2^{\mathtt{w}} > \mathtt{zy} \bmod 2^{\mathtt{w}}$ ならば次の条件である。

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

一方、$\mathtt{zx} \bmod 2^{\mathtt{w}} < \mathtt{zy} \bmod 2^{\mathtt{w}}$ ならば次の条件である。

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

よって、$\mathtt{z}(\mathtt{x}-\mathtt{y}) \bmod 2^{\mathtt{w}}$ 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 $r \geq 0$ が存在し、$(\mathtt{x}-\mathtt{y}) \bmod 2^{\mathtt{w}} = q2^r$ を満たす一意な奇数とする。補題 5.3 より、$\mathtt{z}q \bmod 2^{\mathtt{w}}$ の二進表現は $\mathtt{w}-1$ 桁のランダムなビットを持つ。（最下位桁は 1 である。）

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

よって $\mathtt{z}(\mathtt{x}-\mathtt{y}) \bmod 2^{\mathtt{w}} = \mathtt{z}q2^r \bmod 2^{\mathtt{w}}$ は $\mathtt{w}-r-1$ 桁のランダムなビットを持つ。（その後 1 が続き、さらに r 個の 0 が続く。）

$$\mathtt{z}(\mathtt{x}-\mathtt{y}) \bmod 2^{\mathtt{w}} = \mathtt{z}q2^r \bmod 2^{\mathtt{w}} = (\underbrace{b_{\mathtt{w}-r-1},\ldots,b_1}_{\mathtt{w}-r-1},1,\underbrace{0,0,\ldots,0}_{r})_2$$

これで証明が終わる。$r > \mathtt{w}-\mathtt{d}$ ならば $\mathtt{z}(\mathtt{x}-\mathtt{y}) \bmod 2^{\mathtt{w}}$ の上位 $\mathtt{d}$ ビットは 0 と 1 を共に含む。よって $\mathtt{z}(\mathtt{x}-\mathtt{y}) \bmod 2^{\mathtt{w}}$ が (5.2) または (5.3) である確

率は 0 である。$\mathtt{r} = \mathtt{w} - \mathtt{d}$ ならば (5.2) の確率は 0 だが、(5.3) である確率は $1/2^{\mathtt{d}-1} = 2/2^{\mathtt{d}}$ である。(これは $b_1,\ldots,b_{d-1} = 1,\ldots,1$ である必要があるためだ。) $r < \mathtt{w}-\mathtt{d}$ ならば $b_{\mathtt{w}-r-1},\ldots,b_{\mathtt{w}-r-\mathtt{d}} = 0,\ldots,0$ か、$b_{\mathtt{w}-r-1},\ldots,b_{\mathtt{w}-r-\mathtt{d}} = 1,\ldots,1$ である。いずれの場合の確率も $1/2^{\mathtt{d}}$ 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 $2/2^{\mathtt{d}}$ 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

ChainedHashTable はリストの配列を使うデータ構造であった。i 番目のリストは hash(x) = i である x を全て格納していた。これに対して**オープンアドレス法** (open addressing) と呼ばれる方法は配列 t に直接要素を収めるものである。[*4] このやり方はこの節で説明する LinearHashTable が採用しているものだ。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 の背後にあるアイデアは i = hash(x) である要素 x を理想的には t[i] に入れたい、というものだ。もしこれが（他の要素が既にそこに入っていて）ムリなら、t[(i + 1) mod t.length] に要素を入れてみる。これもムリなら t[(i + 2) mod t.length] に入れてみる。これを x が入れる場所が見つかるまで繰り返す。

t の値は次の三種類のいずれかだ。

[*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：たとえば、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。

1. データの値：USet に入っている実際の値である。
2. null：データが入っていないことを示す。
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。

LinearHashTable の要素数を数えるカウンタ n に加えて、値の個数と del の個数の合計を数えるカウンタ q を用意する。q の値は n に del の個数を加えた値である。これが効率的であるためには t が q より十分大きい必要がある。このとき、t には null である場所がたくさんある。よって LinearHashTable の操作は不変条件 $\mathtt{t.length} \geq 2\mathtt{q}$ を常に満たすようにする。

整理すると、LinearHashTable は要素の配列 t に加え、整数 n、q を持つ。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、ハッシュ関数の値域の大きさは 2 の冪に制限されていることが多いので、整数不変条件 $\mathtt{t.length} = 2^{\mathtt{d}}$ を満たす整数 d も持ち回る。

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

LinearHashTable の find(x) 操作は単純である。i = hash(x) として、$\mathtt{t}[\mathtt{i}], \mathtt{t}[(\mathtt{i}+1) \bmod \mathtt{t.length}], \mathtt{t}[(\mathtt{i}+2) \bmod \mathtt{t.length}], \ldots$ と順に、$\mathtt{t}[\mathtt{i}'] = \mathtt{x}$ または $\mathtt{t}[\mathtt{i}'] = \mathtt{null}$ を満たす添え字 $\mathtt{i}'$ を探す。$\mathtt{t}[\mathtt{i}'] = \mathtt{x}$ のとき、x が見つかったとして $\mathtt{t}[\mathtt{i}']$ を返す。$\mathtt{t}[\mathtt{i}'] = \mathtt{null}$ のとき、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 && x.equals(t[i])) return t[i];
    i = (i == t.length-1) ? 0 : i + 1; // increment i
  }
  return null;
```

```
}
```

LinearHashTable の add(x) 操作も簡単に実装できる。find(x) を使えば x が入っているかどうか確認できる。x が入っていなければ t[i], t[(i + 1) mod t.length], t[(i + 2) mod t.length], ... と順に探し、null か del を見つけたらそこを x に書き換え、n と q をひとつずつ増やす。

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

ここまでで remove(x) の実装も明らかだろう。t[i], t[(i + 1) mod t.length], t[(i + 2) mod t.length], ... と t[i′] = x または t[i′] = null である添え字 i′ を見つけるまで探す。t[i′] = x ならば t[i′] = del とし true を返す。t[i′] = null ならば x はテーブルに入っていなかった（そのため削除できない）として 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.equals(y)) {
      t[i] = del;
```

```
      n--;
      if (8*n < t.length) resize(); // min 12.5% occupancy
      return y;
    }
    i = (i == t.length-1) ? 0 : i + 1;  // increment i
  }
  return null;
}
```

find(x)・add(x)・remove(x) の正しさは簡単に確認できる。ただし、これは del を使うことに依存している。これらの操作は null でない値を null に書き換えないことに注意する。そのため t[i′] = null である添え字 i′ を見つけると、x は配列に入っていないことがわかる。t[i′] はずっと null であったといえるので、先立って、すなわち i′ よりも先の添字に add(x) が要素を追加していることはないのである。

add(x) により null でないエントリの数が t.length/2 を上回るとき、または remove(x) によりデータの入っているエントリ数が t.length/8 を下回るとき、resize() が呼ばれる。resize() は他の配列を使ったデータ構造の場合と同様に働く。まず $2^{\mathtt{d}} \geq 3\mathtt{n}$ を満たす最小の非負整数 d を見つける。続いて、大きさ $2^{\mathtt{d}}$ の配列 t を割当て、古い配列の要素を全て移し替える。この処理の過程で、q を n に等しくリセットする。これは新しい配列 t は del を含まないためである。

LinearHashTable

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

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

5.2.1 線形探索法の解析

add(x)・remove(x)・find(x) のいずれも null であるエントリを見つけると（あるいはその前に）終了する。解析の直感的な説明としては、配列 t の半分以上は null なので、線形探索法はすぐに null のエントリを見つけて処理を終えるはずである。しかしこの直感を当てにしすぎてはいけない。例えば t のエントリを平均的には 2 つだけ見れば良さそうだが、実はこれは正しくない。この節ではハッシュ値は $\{0,\ldots,\mathtt{t.length}-1\}$ から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 Tabulation Hashing という、線形探索法の用途には「十分よい」ハッシュ法を説明する。もうひとつの仮定として、t の添字はすべて t.length で剰余を取っているとする。つまり単に $\mathtt{t}[\mathtt{i}]$ と書いても $\mathtt{t}[\mathtt{i} \bmod \mathtt{t.length}]$ のことである。

i から始まる長さ k の連続 (run) が発生するとはテーブルのエントリ $\mathtt{t}[\mathtt{i}],\mathtt{t}[\mathtt{i}+1],\ldots,\mathtt{t}[\mathtt{i}+k-1]$ がいずれも null でなく、$\mathtt{t}[\mathtt{i}-1]=\mathtt{t}[\mathtt{i}+k]=\mathtt{null}$ であることをいう。t の null でない要素の数は q で、add(x) は常に $\mathtt{q}\leq\mathtt{t.length}/2$ であることを保証する。直前の resize() 以後、t に挿入された q 個の要素を $\mathtt{x}_1,\ldots,\mathtt{x}_\mathtt{q}$ とする。仮定より、ハッシュ値 $\mathtt{hash}(\mathtt{x}_j)$ 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。ここまでの準備で線形探索法の解析における主要な補題を示せる。

補題 5.4. i を $\{0,\ldots,\mathtt{t.length}-1\}$ のある要素に固定する。このときある定数 $c(0<c<1)$ が存在して、i から始まる長さ k の連続が発生する確率を $O(c^k)$ と表せる。

証明. i から始まる長さ k の連続が発生するとき、相異なる k 個の要素 $\mathtt{x}_j$ が存在し、$\mathtt{hash}(\mathtt{x}_j)\in\{\mathtt{i},\ldots,\mathtt{i}+k-1\}$ を満たす。この事象の発生確率は次のよ

うに計算できる。

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

これは k 個の要素の選び方によらず、これら k 個の要素はいずれも k 箇所のうちのいずれかに、そして残りの $\mathtt{q}-k$ 個の要素は残りの $\mathtt{t.length}-k$ 箇所に割り振られなければならないためだ。[*5]

次の導出ではすこしズルをしている。$r!$ を $(r/e)^r$ に置き換える部分である。スターリング近似（1.3.2 節）からこれは真の値との差は $O(\sqrt{r})$ 程度だとわかる。これを許すと導出が簡単になるのである。問 5.4 ではスターリング近似を使ったより厳密な計算を読者にやってもらう予定だ。

$\mathtt{t.length}$ が最小値を取るとき p_k は最大値を取る。またデータ構造は不変条件 $\mathtt{t.length} \ge 2\mathtt{q}$ を保つ。よって次の式が成り立つ。

$$\begin{aligned}
p_k &\le \binom{\mathtt{q}}{k}\left(\frac{k}{2\mathtt{q}}\right)^k\left(\frac{2\mathtt{q}-k}{2\mathtt{q}}\right)^{\mathtt{q}-k} \\
&= \left(\frac{\mathtt{q}!}{(\mathtt{q}-k)!k!}\right)\left(\frac{k}{2\mathtt{q}}\right)^k\left(\frac{2\mathtt{q}-k}{2\mathtt{q}}\right)^{\mathtt{q}-k} \\
&\approx \left(\frac{\mathtt{q}^\mathtt{q}}{(\mathtt{q}-k)^{\mathtt{q}-k}k^k}\right)\left(\frac{k}{2\mathtt{q}}\right)^k\left(\frac{2\mathtt{q}-k}{2\mathtt{q}}\right)^{\mathtt{q}-k} && \text{[Stirling's approximation]} \\
&= \left(\frac{\mathtt{q}^k\mathtt{q}^{\mathtt{q}-k}}{(\mathtt{q}-k)^{\mathtt{q}-k}k^k}\right)\left(\frac{k}{2\mathtt{q}}\right)^k\left(\frac{2\mathtt{q}-k}{2\mathtt{q}}\right)^{\mathtt{q}-k} \\
&= \left(\frac{\mathtt{q}k}{2\mathtt{q}k}\right)^k\left(\frac{\mathtt{q}(2\mathtt{q}-k)}{2\mathtt{q}(\mathtt{q}-k)}\right)^{\mathtt{q}-k} \\
&= \left(\frac{1}{2}\right)^k\left(\frac{(2\mathtt{q}-k)}{2(\mathtt{q}-k)}\right)^{\mathtt{q}-k} \\
&= \left(\frac{1}{2}\right)^k\left(1+\frac{k}{2(\mathtt{q}-k)}\right)^{\mathtt{q}-k} \\
&\le \left(\frac{\sqrt{e}}{2}\right)^k
\end{aligned}$$

最後の変形では $x>0$ ならば $(1+1/x)^x \le e$ 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。 □

[*5] p_k は $\mathtt{i}$ 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する。これは p_k は必要条件 $\mathtt{t}[\mathtt{i}-1] = \mathtt{t}[\mathtt{i}+k] = \mathtt{null}$ を要求しないためである。

補題 5.4 を使って 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。まずは最もシンプルな、find(x) を呼ぶが x が LinearHashTable に入っていない場合を考える。この場合 i = hash(x) は $\{0,\dots,\mathtt{t.length}-1\}$ 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計算できる。

$$O\left(1+\left(\frac{1}{\mathtt{t.length}}\right)\sum_{i=1}^{\mathtt{t.length}}\sum_{k=0}^{\infty}k\Pr\{\mathtt{i}\text{ is part of a run of length }k\}\right)$$

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。

$$\begin{aligned}
&O\left(1+\left(\frac{1}{\mathtt{t.length}}\right)\sum_{i=1}^{\mathtt{t.length}}\sum_{k=0}^{\infty}k^2\Pr\{\mathtt{i}\text{ starts a run of length }k\}\right)\\
&\le O\left(1+\left(\frac{1}{\mathtt{t.length}}\right)\sum_{i=1}^{\mathtt{t.length}}\sum_{k=0}^{\infty}k^2 p_k\right)\\
&= O\left(1+\sum_{k=0}^{\infty}k^2 p_k\right)\\
&= O\left(1+\sum_{k=0}^{\infty}k^2\cdot O(c^k)\right)\\
&= O(1)
\end{aligned}$$

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。

resize() のコストを無視していいなら、これで LinearHashTable の解析は終わりだ。

上の find(x) の解析は、add(x) において x が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 を加えたときのコストと同じである。最後に、remove(x) のコストも find(x) のコストと同じだ。

---

*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 $k\ge k_0$ について、$\frac{(k+1)^2c^{k+1}}{k^2c^k}<1$ を満たす。

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

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、m 個の add(x)・remove(x) からなる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 $\mathrm{hash}(x_1),\ldots,\mathrm{hash}(x_n)$ が独立に一様な確率で $\{0,\ldots,\mathtt{t.length}-1\}$ 内に分布するという仮定である。これを実現するひとつのやり方は大きさ $2^{\mathtt{w}}$ 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-bit の整数乱数で初期化することだ。このとき、hash(x) は tab[x.hashCode()] から d ビットを整数として取り出せばよい。

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

あいにく大きさ $2^{\mathtt{w}}$ の配列はメモリ使用量の観点から現実的でない。Tabulation Hashing では、w ビットの整数に対する乱数表の代わりに、w/r 個の r ビット整数に対する乱数表で妥協する。 こうすれば大きさ $2^{\mathtt{r}}$ の配列 w/r 個で済むのである。これらの配列に入っている整数はいずれも互いに独立な w ビットの乱数である。hash(x) を計算するために、x.hashCode() を w/r 個の r ビット整数に分け、それぞれを配列の添え字として使う。その後各配列の

値をビット単位の排他的論理和を計算し、この結果を 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) {
  int h = x.hashCode();
  return (tab[0][h&0xff]
           ^ tab[1][(h>>>8)&0xff]
           ^ tab[2][(h>>>16)&0xff]
           ^ tab[3][(h>>>24)&0xff])
          >>> (w-d);
}
```

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である。

任意の x について hash(x) は $\{0,\ldots,2^d - 1\}$ の値を一様な確率で取ることを簡単に確認できる。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確認できる。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。しかしそうであっても、Tabulation Hashing は定理 5.2 で示した性質を保証するのに十分よいハッシュ法である。この話題についてはこの章の終わりで参考文献を紹介する。

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 ビットの整数を対応させていた。しかしキーが整数でないことはよくある。例えば文字列・オブジェクト・配列や他の複合データ型である。こういうデータにもハッシュテーブルを使うにはこれらの型から w ビットのハッシュ値を計算すればよい。このハッシュ値が満たすべき性質は次のものだ。

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。（$1/2^w$ に近いということだ。）

一つ目の性質は、x をハッシュテーブルに入れたあと、x と等しい y を検索すると x がちゃんと見つかることを保証する。二つ目の性質は、オブジェクトを整数に変換する際のロスを小さくするものだ。これは相異なるふたつの要素はハッシュテーブルの違う場所に入ることが多いことを保証する。

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のハッシュ値は簡単に計算できる。これらの型にはバイナリ表現があり、これは w ビット以下である。（たとえば Java では byte 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 $\{0,\ldots,2^{\mathtt{w}}-1\}$ の範囲の整数として解釈すればよい。そうすれば、ふたつの異なる値は異なるハッシュ値を持つ。また、ふたつの同じ値は同じハッシュ値を持つ。

w ビットよりも多くのビットを持つプリミティブ型は少ない。ふつうある整数 c が存在し、$c\mathtt{w}$ ビットである。（Java の long・double 型は $c=2$ である例である。）これらのデータ型は c 個のオブジェクトの複合型と考えられる。この扱いは次の小節に譲る。

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、その構成要素のハッシュ値を組み合わせて計算する。これは思うほど簡単でない。いい感じのやり方はたくさん思いつく（例えばビット単位の排他的論理和を計算する）が、そのうちの多くはうまくいかない。（5.7 から 5.9 を参照せよ。）しかし 2w ビットの算術精度があれば単純でロバストな方法がある。$P_0,\ldots,P_{r-1}$ からなる複合オブジェクトがあり、それぞれのハッシュ値は $\mathtt{x}_0,\ldots,\mathtt{x}_{r-1}$ であるとする。このとき互いに独立な w ビットの乱数 $\mathtt{z}_0,\ldots,\mathtt{z}_{r-1}$ 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計算できる。

$$h(\mathtt{x}_0,\ldots,\mathtt{x}_{r-1}) = \left(\left(\mathtt{z}\sum_{i=0}^{r-1}\mathtt{z}_i\mathtt{x}_i\right) \bmod 2^{2\mathtt{w}}\right) \operatorname{div} 2^{\mathtt{w}}$$

このハッシュ値の計算過程は最後に z をかけ、$2^{\mathtt{w}}$ 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。x0・x1・x2 の 3 つの構成要素からなる複合オブジェクトの場合の例を示す。

Point3D
```
int hashCode() {
  // random numbers from rand.org
  long[] z = {0x2058cc50L, 0xcb19137eL, 0x2cb6b6fdL};
  long zz = 0xbea0107e5067d19dL;

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

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

実装が単純なだけでなく、次の定理はこの方法が良い性質を持つことを示す。

定理 5.3. $\mathtt{x}_0,\ldots,\mathtt{x}_{r-1}$ と $\mathtt{y}_0,\ldots,\mathtt{y}_{r-1}$ はいずれも、$\{0,\ldots,2^{\mathtt{w}}-1\}$ の要素である $\mathtt{w}$ ビットの整数からなる列とする。さらに、少なくとも一箇所の添え字 $i\in\{0,\ldots,r-1\}$ で $\mathtt{x}_i\neq\mathtt{y}_i$ が成り立つと仮定する。このとき、次が成り立つ。

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

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。

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

$h'(\mathtt{x}_0,\ldots,\mathtt{x}_{r-1})=h'(\mathtt{y}_0,\ldots,\mathtt{y}_{r-1})$ であるとする。これは次のように書き直せる。

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

ここで t は次のものである。

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

$x_i > y_i$ 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。

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

これは 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 $2^{2w}-2^{w+1}+1 < 2^{2w}-1$ 以下であるためである。仮定より $x_i - y_i \neq 0$ なので、(5.5) は z_i について高々ひとつの解を持つ。z_i と t は互いに独立（$z_0, \dots, z_{r-1}$ は互いに独立である）なので、$h'(x_0, \dots, x_{r-1}) = h'(y_0, \dots, y_{r-1})$ を満たす z_i を選ぶ確率は $1/2^w$ 以下である。

最後の処理は乗算ハッシュ法であり、$2w$ ビットの中間結果 $h'(x_0, \dots, x_{r-1})$ を w ビットの最終結果 $h(x_0, \dots, x_{r-1})$ に縮める。定理 **??**より、$h'(x_0, \dots, x_{r-1}) \neq h'(y_0, \dots, y_{r-1})$ ならば $\Pr\{h(x_0, \dots, x_{r-1}) = h(y_0, \dots, y_{r-1})\} \leq 2/2^w$ である。以上より、次の式が成り立つ。

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

□

5.3.3 配列と文字列のハッシュ値

前小節の手法はオブジェクトが固定数の構成要素からなるときにはうまくいく。しかし、可変長のオブジェクトをうまく扱えない。なぜなら 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ためである。

必要なだけ z_i を生成するためには擬似乱数列を使えるが、その場合 z_i は互いに独立ではなく、擬似乱数がハッシュ関数に対して悪影響を及ぼさないことを証明するのは難しい。例えば定理 5.3 の証明における t と z_i の独立性は成り立たなくなる。

ここでは素数体上の多項式を使ったハッシュ法を使う。これは正規多項式の値を計算し、素数 p による剰余を取るものだ。次の定理は素数体上の多項式が普通の多項式と似た振る舞いをすることを主張する。

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 $x_i \in \{0, \dots, p-1\}$ を係数とする非自明 ($x_0 = x_1 = \dots = x_{r-1} = 0$ でない) な多項式とする。このとき、等式 $f(z) \bmod p = 0$ は $z \in \{0, \dots, p-1\}$ の範囲に高々 $r-1$ 個の解を持つ。

定理 5.4 より、$z \in \{0,\dots,p-1\}$ を使えば、$x_i \in \{0,\dots,p-2\}$ である整数の列 $x_0,\dots,x_{r-1}$ のハッシュ値を計算できる。

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

最後に追加された項 $(p-1)z^r$ に注意する。$(p-1)$ を整数列の末尾の要素として $x_0,\dots,x_r$ と考えると便利かもしれない。この要素は整数列の要素のいずれとも異なる。整数列の要素は $\{0,\dots,p-2\}$ の要素である。$p-1$ を列の終わりを示すマーカーだと考える。

次の定理はふたつの同じ長さの列について、z だけの小さなランダム性にも関わらず、良いハッシュ値が得られることを示すものだ。

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

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

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

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

$x_i \neq y_i$ なので、この多項式は非自明である。よって定理 5.4 より z についての解は高々 $r-1$ 個である。以上より、z を選んでこの解のうちの一つを引く確率は $(r-1)/p$ 以下である。 □

このハッシュ関数はふたつの入力列の長さが異なる場合にも対応できる。この場合、一方の入力列が他方の入力列を含んでいても構わない。この関数は入力が無限列であってもそのまま問題なく処理できるのだ。

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

長さ $r,r'(r>r')$ のふたつの入力があるとき、ふたつの列は添え字 $i=r$ 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 個の解をもつ。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 $\mathtt{p} > 2^{\mathtt{w}}+1$ を満たす素数とする。$\mathtt{x}_0,\dots,\mathtt{x}_{r-1}$ と $\mathtt{y}_0,\dots,\mathtt{y}_{r'-1}$ は $\{0,\dots,2^{\mathtt{w}}-1\}$ の要素である $\mathtt{w}$ ビット整数からなる相異なる列であるとする。このとき次の式が成り立つ。

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

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をこのハッシュ関数がどう扱うかがわかるだろう。

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

このコードは実装上の都合で衝突確率がやや大きくなっている。特に x[i].hashCode() を 31 ビットに縮めるために 5.1.1 節で $\mathtt{d}=31$ とした乗算ハッシュ法を使っている。これは素数 $\mathtt{p}=2^{32}-5$ で剰余を取った足し算や掛け算を、符号なし 63 ビット整数で実行するためである。

よって、長い方の長さが r であるふた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である。

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

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。ハッシュ方のオンライン参考文献一覧は [10]2000 近いエントリを含む。

ハッシュテーブルには他にも様々な実装がある。

5.1 節で説明したものは**チェイン法によるハッシュ法**（hashing with chaining）である。（各配列のエントリは要素のチェイン（`List`）である。）チェイン法によるハッシュテーブルは IBM にて H. P. Luhn が 1953 年 1 月に出した内部報告書で提案された。この報告書は連結リストの最古の文献のうちの一つでもあると思われる。

別の方法として、**オープンアドレス法**がある。これは全てのデータを配列に直接格納するものだ。5.2 節で説明した `LinearHashTable` はこのやり方のうちのひとつである。このアイデアもまた別の IBM のグループによって独立に 1950 年代に提案された。オープンアドレス法は**衝突の解決** (collision resolution) のことを考えなければならない。衝突とはふたつの値が配列の同じ位置に割当てられることだ。このための方法にはいくつか種類がある。それぞれ異なる性能保証があり、またこの章で説明したものよりも精巧なハッシュ関数を用いるものもある。

また別のハッシュテーブルの実装に関する話題としては、**完全ハッシュ法** (perfect hashing) と呼ばれるものがある。これは `find`(x) の実行時間が最悪の場合でも $O(1)$ であるハッシュ法だ。データセットが静的な場合にはこれはデータセットに対する**完全ハッシュ関数** (perfect hash function) を見つけることで実現できる。これはすべてのデータを別々の配列内の位置に対応させるハッシュ関数である。データが動的な場合には完全ハッシュ法として FKS **二段階ハッシュテーブル** (two-level hash table) [31, 24] や cuckoo hashing [57] などが知られている。

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の中でおそらく最も実用的なものである。別のよい方法として、Carter と Wegman による先駆け的な研究成果であった**ユニバーサルハッシュ法** (universal hashing) を使ったものがある。用途に応じた別々の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 Pǎtraşcu と Thorup の研究成果である。[60]

乗算ハッシュ法 (multiplicative hashing) のアイデアは非常に古くからあり、おそらくこれはハッシュ法の伝承といってもよいだろう。[48, Section 6.4] しかし、5.1.1 節で説明した乗数 $\mathtt{z}$ をランダムな奇数から選ぶアイデアとその解析は 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もののうちの一つだが、衝突確率が $2/2^{\mathtt{d}}$、つまり $2^{\mathtt{w}}$ から $2^{\mathtt{d}}$ への全ての関数からランダムに選出した場合（理想的な場合）とくらべて衝突確率が二倍になってしまう。multiply-add **ハッシュ法**は次の関数を使う方法だ。

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

ここで $\mathtt{z}$ と $\mathtt{b}$ いずれも $\{0,\ldots,2^{2\mathtt{w}}-1\}$ 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 $1/2^{\mathtt{d}}$ である。[21] しかし、$2\mathtt{w}$ ビット精度の四則演算が必要である。

固定長の $\mathtt{w}$ 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]

$$\begin{aligned} &h(\mathtt{x}_0,\ldots,\mathtt{x}_{r-1}) \\ &\quad = \left(\sum_{i=0}^{r/2-1} ((\mathtt{x}_{2i} + \mathtt{a}_{2i}) \bmod 2^{\mathtt{w}})((\mathtt{x}_{2i+1} + \mathtt{a}_{2i+1}) \bmod 2^{\mathtt{w}})\right) \bmod 2^{2\mathtt{w}} \end{aligned}$$

ここで r は偶数であり、$\mathtt{a}_0,\ldots,\mathtt{a}_{r-1}$ はいずれも $\{0,\ldots,2^{\mathtt{w}}\}$ からランダムに選出される。

この $2\mathtt{w}$ ビットのハッシュ値が衝突する確率は $1/2^{\mathtt{w}}$ である。これを乗算ハッシュ法（か Multiply-add）を使って $\mathtt{w}$ ビットに縮めることができる。これは $r/2$ 回の $2\mathtt{w}$ ビット乗算だけで実現でき、これは高速である。5.3.2 節の方法は r 回の乗算が必要であった。（mod の計算は $\mathtt{w}$ または $2\mathtt{w}$ ビットの足し算、掛け算では暗に実行される。）

5.3.3 節で説明した素数体を使った可変長配列のハッシュ法は Dietzfelbinger et al.[22] による。この方法は mod を使うが、これは時間のかかる機械語命令であり、その結果この方法は速くはない。剰余の法として $2^{\mathtt{w}}-1$ を使う工夫がある。こうすると 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 $c > 1$ のブロックに対して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数体を使った方法でハッシュ値を求めるものがある。

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。この番号はひとつずつ増える整数で、何年も前に 0 から始まり、今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。このとき下の二桁、あるいは上の二桁のどちらを使うのはいいアイデアだろうか?説明せよ。

問 **5.2.** 5.1.1 節の方法において、$n = 2^d$ かつ $d \le w/2$ である場合を考える。

1. z によらず、持つ相異なる 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:これは簡単である。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。(これはより難しく、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。(ヒントとしては、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.** 要素 x を LinearHashTable に要素を追加するための簡略化された次のコードを見よ。これは単純に x をはじめに見つけた null であるエントリに入れる。このコードは非常に遅いことがあることを示せ。すなわち、$O(n)$ 個の add(x)・remove(x)・find(x) からなる操作の列で n^2 の実行時間がかかる例を挙げよ。

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

```
}
```

問 **5.6.** 昔の Java では `String` クラスの `hashCode()` メソッドは長い文字列の全ての文字を使ってはいなかった。例えば 16 文字の文字列の場合は偶数番目の 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.** ふたつの $\mathtt{w}$ ビットの整数 $\mathtt{x}$ と $\mathtt{y}$ からなるオブジェクトがあるとき、$\mathtt{x} \oplus \mathtt{y}$ 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.** ふたつの $\mathtt{w}$ ビットの整数 $\mathtt{x}$ と $\mathtt{y}$ からなるオブジェクトがあるとき、$\mathtt{x} + \mathtt{y}$ 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.** ふたつの $\mathtt{w}$ ビットの整数 $\mathtt{x}$ と $\mathtt{y}$ からなるオブジェクトがあるとする。決定的な関数 $h(\mathtt{x},\mathtt{y})$ により $\mathtt{w}$ ビットの整数であ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 $\mathtt{w}$ について、$p = 2^{\mathtt{w}} - 1$ 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。

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

（これは $x \bmod (2^{\mathtt{w}} - 1)$ を計算するための方法として、

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

を $\mathtt{x} \le 2^{\mathtt{w}} - 1$ を満たすまで繰り返すアルゴリズムを与えている。）

問 **5.11.** 標準ライブラリやこの本の `HashTable`・`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 $\mathtt{x}$ 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 $\mathtt{n}$ 個の整数の集まりを見つけよ。

実装によって、単にコードを見れば十分だったり、あるいは試しに挿入・検索をしてみてその時間を測ってみたりする必要があるだろう。これはウェブサーバーへの DoS (denial of service) 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。数学的には、**二分木** (binary tree) とは連結 (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二番目のノードを 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、ふつう根を一番上に書く。また、ノード u の左右の子は、u の左下・右下にそれぞれ描かれる。（図 6.1）図 6.2.a は 9 個のノードを持つ二分木の例である。

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

[*1] 訳注：グラフが連結 (connected) 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・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。たとえば図 6.1 において、u の次数は 3 である。

[*4] 訳注：たとえば u=r であるとき、u の深さは 0 である。

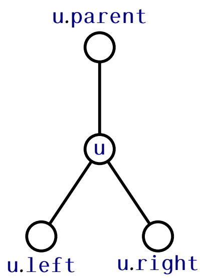


図 6.1: BinaryTree における、ノード u の親・左の子・右の子

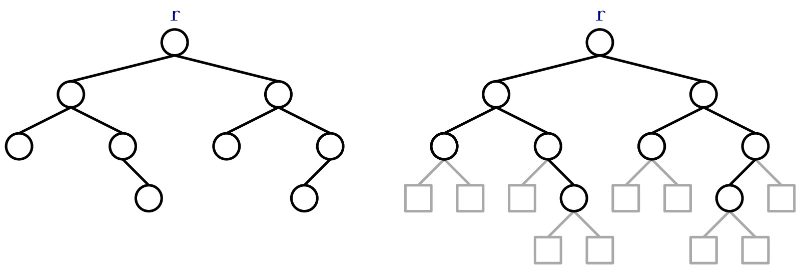


図 6.2: (a) 9 個の本物のノードを持つ二分木と、(b) 10 個の外部ノードを持つ二分木

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 が子を持たないなら、u は**葉** (leaf) である。

外部ノード (external node) という概念を考えると便利なことがある。左の子を持たないノードは左の子として外部ノードを持ち、右の子を持たないノードは右の子として外部ノードを持つとする。（図 6.2.b を参照せよ。）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を示せる。

6.1 BinaryTree：基本的な二分木

二分木におけるノード u を表現するには、u に隣接する（3 つ以下の）ノードを明示的に保持するのが簡単だ。

BinaryTree
```
class BTNode<Node extends BTNode<Node>> {
  Node left;
  Node right;
  Node parent;
}
```

隣接する頂点のうち、存在しないものを指す変数は nil とする。このとき、外部ノードと根の親とが nil に対応する。

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;
```

ノード u の深さを、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++;
  }
  return d;
```

[*5] 訳注：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。木に含まれる何らかのノードへの参照が 1 つあれば、他のノードへたどり着ける。そのような参照として根を選ぶ理由は、後の章で出てくるような木（たとえば B 木。14.2.2 節を参照せよ。）では、親への参照が存在せず、根以外の参照を選んだ場合に複数持つ必要が生じるためだ。

```
}
```

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 のふた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 [*6]。

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

先の小節で説明したふたつのアルゴリズム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。

[*6] 訳注：完全な余談だが、このように二分木の高さを求めるのは、米国におけるソフトウェアエンジニア面接の頻出問題の一つである。https://leetcode.com/problems/maximum-depth-of-binary-tree/

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 [*7]。

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 を参照せよ。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。ただし、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;
    }
```

[*7] 訳注：この問題はスタックオーバーフローと呼ばれる。

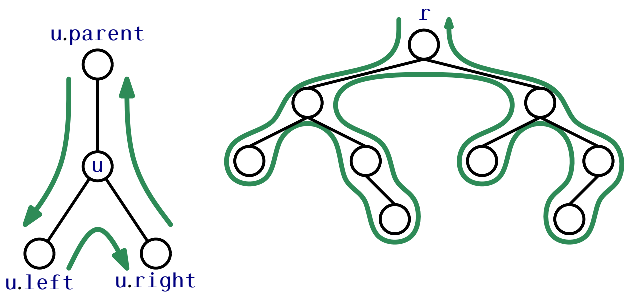


図 6.3: 二分木を再帰を使わずに走査するときの、三通りのノード u の訪れ方と、その結果として生じる木の走査

```
    prev = u;
    u = next;
  }
}
```

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ひとつずつ増やせばよい。

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;
}
```

二分木の実装には、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、List か Stack を使って、今訪問しているノードから根までの経路を記録しておく必要がある。

ここまで説明したものとは別の走査方法として、**幅優先** (breadth-first) な走査がある [*8]。幅優先に走査する場合、根から下に向かって深さごとに、同じ深さのノードは左から右の順に、すべてのノードを訪問する。（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() {
  Queue<Node> q = new LinkedList<Node>();
  if (r != nil) q.add(r);
  while (!q.isEmpty()) {
    Node u = q.remove();
    if (u.left != nil) q.add(u.left);
    if (u.right != nil) q.add(u.right);
  }
```

[*8] 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。

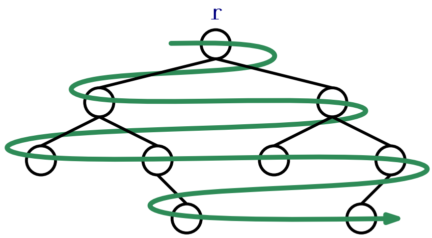


図 6.4: 幅優先な走査では、二分木の各ノードは深さ毎に、また各深さでは左から右の順で訪問される。

```
}
```

6.2 BinarySearchTree：バランスされていない二分探索木

BinarySearchTree（二分探索木）はノード u がある全順序な集合の要素であるデータ u.x を持つ二分木である。二分探索木の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。 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。BinarySearchTree の例を図 6.5 に示す。

6.2.1 探索

二分探索木の性質は有用だ。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 に訪問しているとき、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。

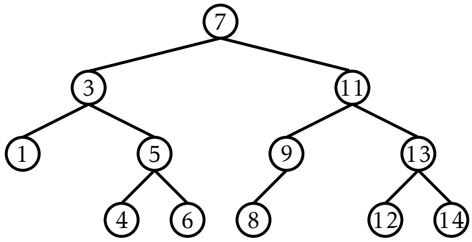


図 6.5: 二分探索木の例

この探索は三つ目のケース、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 u = r;
  while (u != nil) {
    int comp = compare(x, u.x);
    if (comp < 0)
      u = u.left;
    else if (comp > 0)
      u = u.right;
    else
      return u.x;
  }
  return null;
}
```

二分探索木における探索の例を図 6.6 にふたつ示す。二つ目の例から、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、先の場合分けの一つ目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、最小のものである。同様に場合分けの二つ目のケースであったなら、u.x は x より小さい値のうち、最大のものである。よって、場合分けの一つ目のケースが最後に発生したノード z を記録しておけば、BinarySearchTree の find(x) を、x 以上の値のうち最小のものを返すよう実装することもできる。

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 を追加するために、まずは 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
```
boolean add(T x) {
  Node p = findLast(x);
```

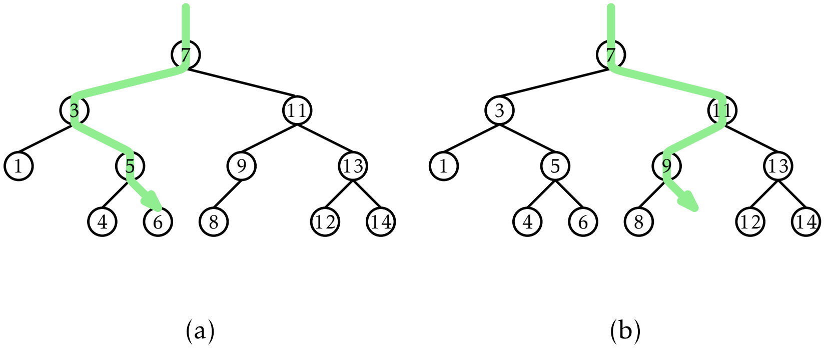


図 6.6: 二分探索木において、(a) 探索が成功する例（6 が見つかる）と、(b) 探索が失敗する（10 が見つからない）例

```
  return addChild(p, newNode(x));
}
```

BinarySearchTree

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

```
}
```

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

図 6.7 に例を示した。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。 最悪の場合にはこれは BinarySearchTree の高さである。

6.2.3 削除

BinarySearchTree から、ある値を格納するノード u を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 がひとつだけの子を持つなら、u の点を継ぎ合わせる、すなわち u.parent と u の子とを新たに親子関係とすればよい。（図 6.8 を参照せよ。）

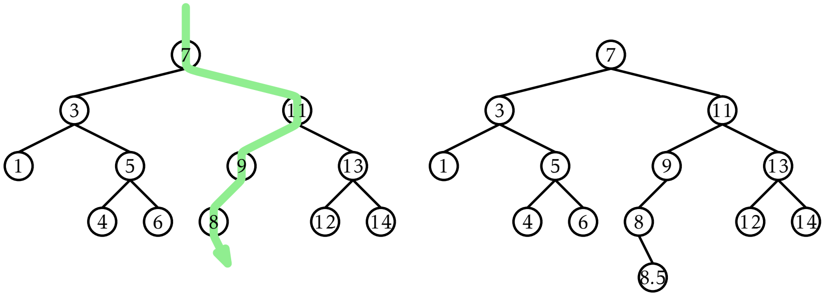


図 6.7: 二分探索木に 8.5 を追加する様子

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

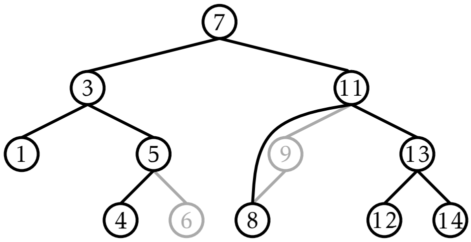


図 6.8: 葉 (6)、または一つだけの子を持つノード (9) を削除するのは簡単である。

```
  }
  n--;
}
```

u がふたつの子を持つ場合はもっと手の込んだことをする必要がある。この場合、子の数が 1 以下のノード w であって、w.x と u.x とを入れ替えられるものを見つけるのが最も単純なやり方だ。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ため、取り除くのも簡単である。（図 6.9 を参照せよ。）

BinarySearchTree

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

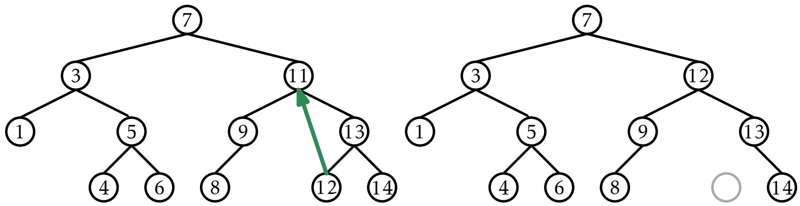


図 6.9: ふたつの子を持つノード u から値 11 を削除するために、u の値と、u の右の部分木における最小の値とを入れかえる。

```
}
```

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。木の形状についてなにか仮定しない限り、この経路の長さについて「木の中のノード数は超えない」より強い主張をするのは難しい。次の（あまりパッとしない）定理は 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 は SSet インターフェースの操作を期待実行時間 $O(\log n)$ で実装していた。BinarySearchTree の問題は、木の形状が**アンバランス** (unbalanced) かもしれないことだ。図 6.5 のような木の形ではなく、ほとんどのノードがひとつだけの子を持ち、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[*9]。

アンバランスな二分探索木を避ける方法はたくさんあり、そうすると $O(\log n)$ の時間で各操作を行えるようになる。7 章で**期待実行時間** $O(\log n)$ を、ランダム性を利用して達成する方法を説明する。8 章では**償却実行時間**

[*9] 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（細かな違いはあるが）。

$O(\log n)$ を、部分的な再構築を利用して達成する方法を説明する。9 章では**最悪実行時間** $O(\log n)$ を、4 つまで子を持ちうる木をシミュレートすることで達成する方法を説明する。

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渡って使われている。二分木を使って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、根をある人、左右の子ノードをその人の両親とする。数世紀前からは生物学における系統樹にも二分木は使われている。ここでは葉は現存の種を表し、内部ノードは**分化**が起きたことを表す。分化とは一つの種から、二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] 個々の二分探索木のより詳細な文献は後の章で紹介する。

ゼロから二分木を実装するとき、いくつか設計上考えることがある。ひとつは各ノードが親へのポインタを持つかどうかである。多くの操作が根から葉への経路を辿るだけのものなら、親へのポインタは不要で、メモリを無駄にし、バグを入れ込む原因となりうる。一方親へのポインタがないと、走査のために再帰を（または明示的にスタックを）使うことになる。また、実装が複雑になってしまう操作もある。（ある種の二分探索木における挿入や削除など）

もうひと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。この章の実装では、それぞれを別々の変数に保持していた。一方、長さ 3 の配列 p を使うこともできる。この場合、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。配列を使うと、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。

例えば、この単純化は木を辿るときに役立つ。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] の兄弟は u.p[(i + 1) mod 2] である。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。この表現を使うと、左右の場合ためにそれぞれ書いていた複雑なコードを、ひとつにまとめられることがある。例として 162 ページの `rotateLeft(u)`・`rotateRight(u)` を参照せよ。

問 **6.1.** $n \geq 1$ 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 が一つ以上葉を持つとき、T における根の子の数が 1 以下であるか、T は二つ以上の葉を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。（色々な形状の大きい木でテストしてみること。$O(n)$ より多く時間を使う実装は簡単である。）

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、各対応する順序に従って頂点を訪問した時のノードに付された訪問順の番号である。例として図 6.10 を見よ。

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。また、ノード u からはじめて、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

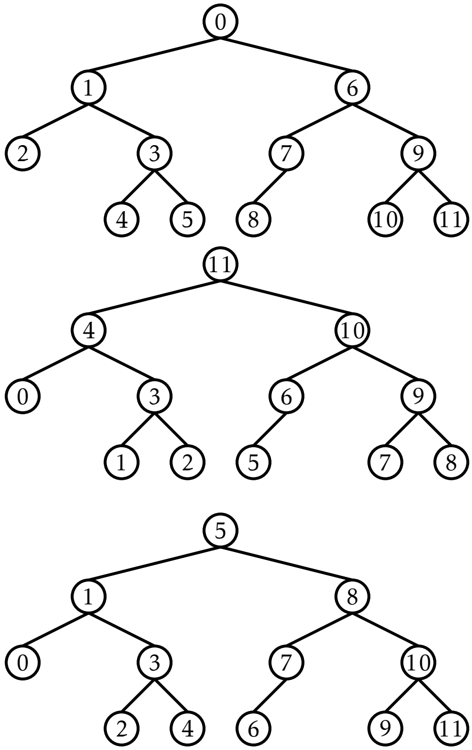


図 6.10: 二分木における行きがけ順・通りがけ順・帰りがけ順

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・通りがけ番号の組みの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 $\{1,\ldots,n\}$ 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、こ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、必ず木は元の状態に戻るか?

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があるか? 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があるか? また、そのとき木の高さが大きくなることがあるか? もしそうなら、どのくらい大きくなりうるか?

問 **6.19.** BinarySearchTree の一種であり、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）を保持するものを設計・実装せよ。

なお、add(x)・remove(x) を呼んでもこれらの値は適切に保たれる必要があり、一方で、これらの操作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 章

ランダム二分探索木

この章では乱択化を使った各操作の実行時間が $O(\log n)$ であ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。左のものはリストであり、右のものは完全にバランスの取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。

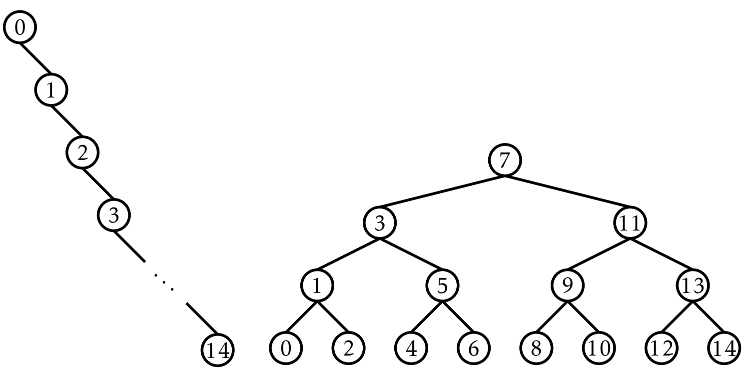


図 7.1: 0,...,14 からなる 2 つの二分探索木

この 2 つの木の構築方法を考えてみよう。空の `BinarySearchTree` に要素を次の順で追加すると左の木になる。

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

また逆に、左の木になる要素の追加順はこれに限る。（証明は $\mathtt{n}$ についての帰納法で行う。）一方要素を次の順で追加すると右の木になる。

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

他にも

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

や

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

からも、この木が得られる。

右の木が得られる要素の追加順は 21,964,800 種類ある一方で、左の木が得られる要素の追加 s 順はたった 1 つしかない。

上の例から、0,...,14 をランダムな順番で二分探索木に追加すると、多くの場合は（図 7.1 の右のもののような）バランスのよい木になり、（図 7.1 の左のもののような）極めて偏った木になるのは稀だと言えそうだ。

ランダム二分探索木を考えるとこの直感を形式的に説明できる。まずは大きさ $\mathtt{n}$ の**ランダム二分探索木** (random binary search tree) の作り方を述べる。

$0,\ldots,\mathtt{n}-1$ のランダムな置換を 1 つ選ぶ。これを $\mathtt{x}_0,\ldots,\mathtt{x}_{\mathtt{n}-1}$ とし、順に `BinarySearchTree` に追加する。ここで、**ランダムな置換** (random permutation) とは、$0,\ldots,\mathtt{n}-1$ の並べ方 $\mathtt{n}!$ 通りから、いずれも等しい確率 $1/\mathtt{n}!$ で 1 つを選出したものである。

$0,\ldots,\mathtt{n}-1$ を別の $\mathtt{n}$ 個の値に変えてもランダム二分探索木の性質には影響しないことに注意する。$\mathtt{x}\in\{0,\ldots,\mathtt{n}-1\}$ は $\mathtt{n}$ 個の順序が定義された集合における $\mathtt{x}$ 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現れる数の話をしよう。k を非負整数とすると、k 番めの**調和数** (harmonic number)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。

$$\ln k < H_k \le \ln k + 1$$

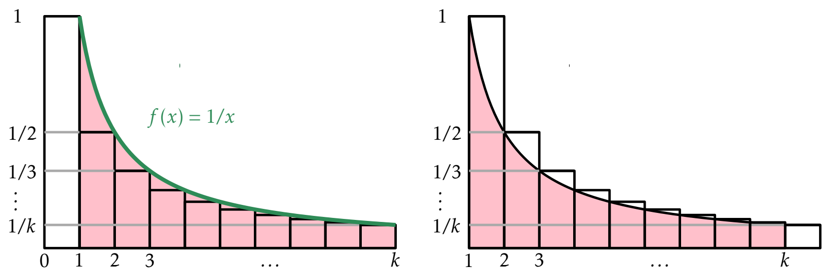


図 7.2: k 番めの調和数 $H_k = \sum_{i=1}^{k} 1/i$ の上界・下界を積分で計算できる。各積分値は図の斜線部の面積であり、H_k は長方形の部分の面積である。

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 軸との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次のことが成り立つ。

1. 任意の $\mathtt{x} \in \{0,\ldots,\mathtt{n}-1\}$ について、x の探索経路の長さの期待値は $H_{\mathtt{x}+1} + H_{\mathtt{n}-\mathtt{x}} - 2$ である。[*1]
2. 任意の $\mathtt{x} \in (-1,n) \setminus \{0,\ldots,\mathtt{n}-1\}$ について、x の探索経路の長さの期待値は $H_{\lceil \mathtt{x} \rceil} + H_{\mathtt{n}-\lceil \mathtt{x} \rceil}$ である。

補題 7.1 は次の小節で証明する。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とき、探索経路の長さの期待値は $2\ln n + O(1)$ 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の探索経路の長さの期待値に関するものだ。 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方が、少しだけ早いことがわかる。

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 $\mathtt{x} \in (-1,\mathtt{n})$ の探索経路に $i < \mathtt{x}$ 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 $\{i+1, i+2, \ldots, \lfloor \mathtt{x} \rfloor\}$ のいず

[*1] $\mathtt{x}+1$ と $\mathtt{n}-\mathtt{x}$ 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。

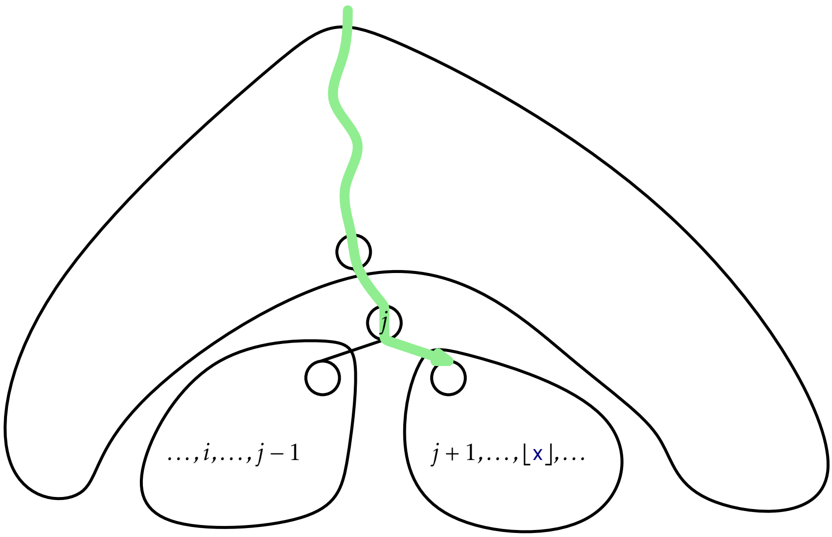


図 7.3: 値 $i < \mathtt{x}$ が x の探索経路中にあることの必要十分条件は i が $\{i, i+1, \ldots, \lfloor \mathtt{x} \rfloor\}$ のうち最初に木に加えられた要素であることである。

れよりも前に現れることである。

図 7.3 でいうと、$\{i, i+1, \ldots, \lfloor \mathtt{x} \rfloor\}$ 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ことから確認できる。（2 つの要素の探索経路が異なるなら、その一方以上かつ他方以下の要素が存在する。）j をランダムな置換において最初に現れる $\{i, i+1, \ldots, \lfloor \mathtt{x} \rfloor\}$ の要素とする。j はずっと x の探索経路上にあることに注意する。$j \neq i$ 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。そしてその後、i が追加されるとき、$i < j$ なので $\mathtt{u}_j.\mathtt{left}$ を根とする部分木に $\mathtt{u}_i$ は追加される。一方 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 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。

$\{0, \ldots, \mathtt{n}\}$ のランダムな置換を考えるとき、そのうち $\{i, i+1, \ldots, \lfloor \mathtt{x} \rfloor\}$ または $\{\lceil \mathtt{x} \rceil, \lceil \mathtt{x} \rceil + 1, \ldots, i-1\}$ だけを取り出した部分列も、やはりそれぞれのランダムな置換になっている。ランダムな置換を作ると、どの要素もみな等しい確率で

先頭に現れるので次の式が得られる。

$$\Pr\{i \text{ が } \mathtt{x} \text{ の探索経路に含まれる }\} = \begin{cases} 1/(\lfloor \mathtt{x} \rfloor - i + 1) & \text{if } i < \mathtt{x} \\ 1/(i - \lceil \mathtt{x} \rceil + 1) & \text{if } i > \mathtt{x} \end{cases} .$$

この考察により、調和数の簡単な計算で補題 7.1 を証明できる。

補題 7.1 の証明. I_i を指示確率変数とする。この値は、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。

$$\sum_{i \in \{0,\ldots,\mathtt{n}-1\} \setminus \{\mathtt{x}\}} I_i$$

よって $\mathtt{x} \in \{0,\ldots,\mathtt{n}-1\}$ 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 を見よ。）

$$\begin{aligned}
\mathrm{E}\left[\sum_{i=0}^{\mathtt{x}-1} I_i + \sum_{i=\mathtt{x}+1}^{\mathtt{n}-1} I_i\right] &= \sum_{i=0}^{\mathtt{x}-1} \mathrm{E}[I_i] + \sum_{i=\mathtt{x}+1}^{\mathtt{n}-1} \mathrm{E}[I_i] \\
&= \sum_{i=0}^{\mathtt{x}-1} 1/(\lfloor \mathtt{x} \rfloor - i + 1) + \sum_{i=\mathtt{x}+1}^{\mathtt{n}-1} 1/(i - \lceil \mathtt{x} \rceil + 1) \\
&= \sum_{i=0}^{\mathtt{x}-1} 1/(\mathtt{x} - i + 1) + \sum_{i=\mathtt{x}+1}^{\mathtt{n}-1} 1/(i - \mathtt{x} + 1) \\
&= \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\mathtt{x}+1} \\
&\quad + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\mathtt{n}-\mathtt{x}} \\
&= H_{\mathtt{x}+1} + H_{\mathtt{n}-\mathtt{x}} - 2 .
\end{aligned}$$

値 $\mathtt{x} \in (-1, n) \setminus \{0,\ldots,\mathtt{n}-1\}$ の場合も同様である。（図 7.4.b を見よ。） □

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 $O(\mathtt{n} \log \mathtt{n})$ 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 $O(\log \mathtt{n})$ である。

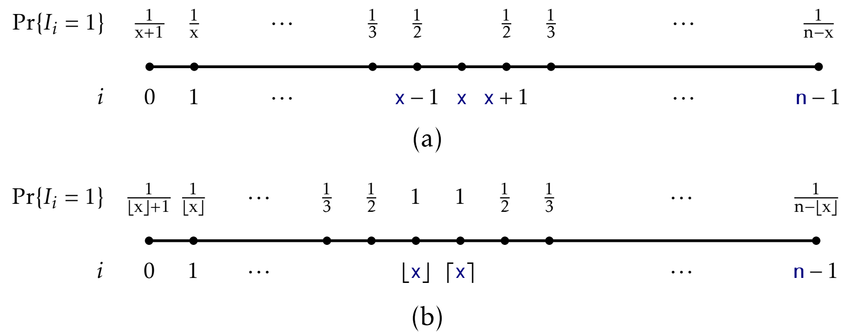


図 7.4: x の探索経路に各要素が現れる確率 (a) x が整数のとき (b) x が整数でないとき

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 についてこれは成り立つ。

7.2 Treap: 動的ランダム二分探索木

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だ。すなわち SSet インターフェースの add(x)・remove(x) をサポートしていないのである。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 を持つ点で BinarySearchTree に似ているが、それに加えて一意な**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 Node<T> extends BSTNode<Node<T>,T> {
  int p;
}
```

Treap のノードは、二分探索木の性質に加えて、**ヒープ性** (heap property)

[*2] Treap の名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。

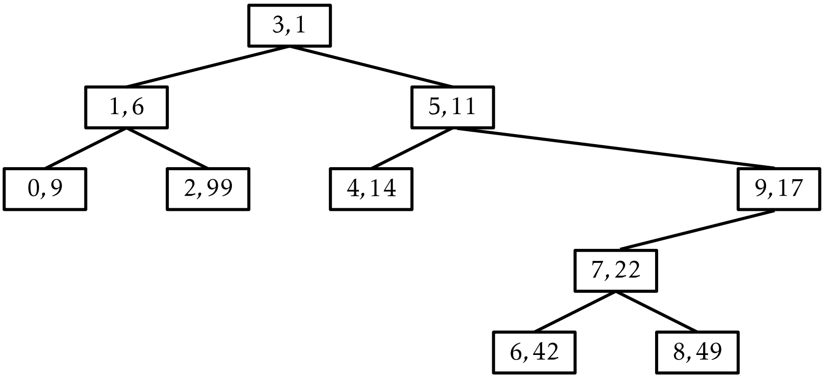


図 7.5: 整数 0,...,9 を含む Treap の例。ノード u は u.x,u.p を含む四角形として描かれている。

も満たす。

- （ヒープ性）根でない任意のノード u について $\texttt{u.parent.p} < \texttt{u.p}$ 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 の例を示した。

ヒープ性と二分探索木の性質とを共に満たすので、キー x と優先度 p が決まれば Treap の形状は一意に定まる。ヒープ性から、最小の優先度を持つノードが Treap の根 r である。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 より大さ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、かつランダムであることだ。このことから Treap の 2 つ等価な解釈の仕方がある。先ほど定義したように、Treap はヒープ性と二分探索木性とを共に満たす。別の角度から見ると、Treap は優先度の昇順にノードが追加される BinarySearchTree だと解釈してもよい。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 を得られる。

$$\langle(3,1),(1,6),(0,9),(5,11),(4,14),(9,17),(7,22),(6,42),(8,49),(2,99)\rangle$$

Treap におけるノードの優先度はランダムに決まるので、これはキーをラ

ンダムに置換するのと同じである。例えば上の例は次の置換に対応する。

$$\langle 3,1,0,5,9,4,7,6,8,2\rangle$$

この順番で要素を空の BinarySearchTree に追加すると、先ほどの Treap に対応する BinarySearchTree が得られる。以上より Treap の形状はランダム二分探索木と同様に決まることがわかっただろう。特に、キー x をそのランクに置き換えれば [*3] 補題 7.1 を Treap にも適用できる。補題 7.1 を Treap のために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* について次のことが成り立つ。

1. 任意の $\mathtt{x}\in S$ について x の探索経路の長さの期待値は $H_{r(\mathtt{x})+1}+H_{\mathtt{n}-r(\mathtt{x})}-2$ である。
2. 任意の $\mathtt{x}\notin S$ について x の探索経路の長さの期待値は $H_{r(\mathtt{x})}+H_{\mathtt{n}-r(\mathtt{x})}$ である。

ここで $r(\mathtt{x})$ は集合 $S\cup\{\mathtt{x}\}$ における x のランクである。

繰り返しになるが、補題 7.2 の期待値は優先度のランダム性に基づくもので、キーがランダムだとは仮定していない。

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。こ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 を参照せよ。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、左の w が u の左の子なら左回転を使う。

これを実装するためには、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 から想像するものより少し長くなる。

BinarySearchTree
```
void rotateLeft(Node u) {
  Node w = u.right;
```

[*3] x を集合 S の要素とするとき、x のランクとは S の要素のうち x より小さいものの個数である。

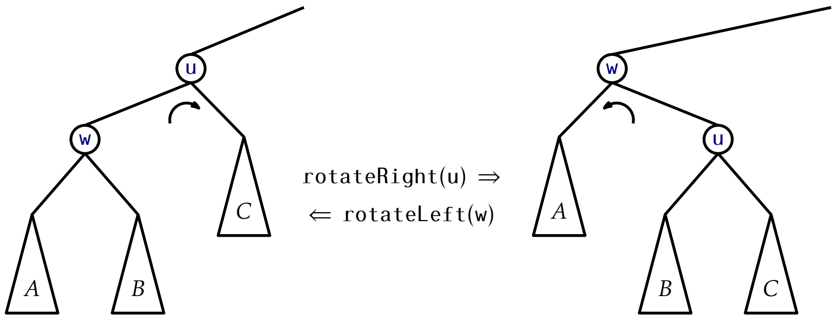


図 7.6: 二分探索木の左回転・右回転

```
  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; }
}
```

Treap における回転の重要な性質は、w の深さが 1 減り u の深さが 1 増えることだ。

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 は二分探索木性を満たすが、一方ヒープ性を満たすとは限らない。具体的には u.parent.p > u.p だとこの Treap はヒープ性を犯している。この場合、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 がヒープ性を犯しているなら、もう一度回転を実行する。この度に u の深さは 1 減り、u が根になるか u.parent.p < u.p を満たすと処理は終了する。

Treap

```
boolean add(T x) {
  Node<T> u = newNode();
  u.x = x;
  u.p = rand.nextInt();
  if (super.add(u)) {
    bubbleUp(u);
    return true;
```

```
  }
  return false;
}
void bubbleUp(Node<T> u) {
  while (u.parent != nil && u.parent.p > u.p) {
    if (u.parent.right == u) {
      rotateLeft(u.parent);
    } else {
      rotateRight(u.parent);
    }
  }
  if (u.parent == nil) {
    r = u;
  }
}
```

図 7.7 に add(x) 操作の例を示した。

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する回数から求まる。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 の深さは減る。u が根になると処理が終了するので、回転回数の期待値は探索経路長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。(問 7.5 はこの操作における回転の回数の期待値は実は $O(1)$ であることを示す問題である。)

Treap における remove(x) は add(x) の逆である。x を含むノード u を探し、u が葉に来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。

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 でない方と回転で 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

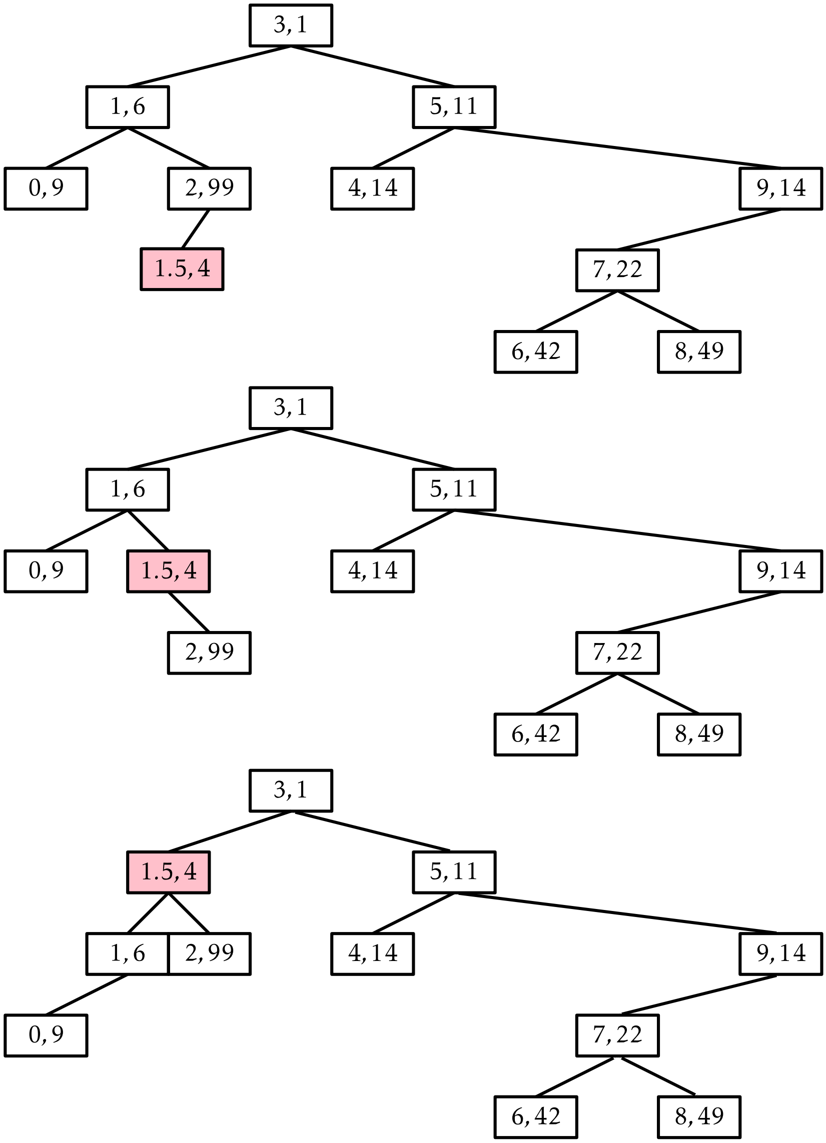


図 7.7: 図 7.5 の Treap に値 1.5 を追加する。

この規則により Treap は連結であり、またヒープ性も保たれることがわかる。

Treap
```
boolean remove(T x) {
  Node<T> u = findLast(x);
  if (u != nil && compare(u.x, x) == 0) {
    trickleDown(u);
    splice(u);
    return true;
  }
  return false;
}
void trickleDown(Node<T> u) {
  while (u.left != nil || u.right != nil) {
    if (u.left == nil) {
      rotateLeft(u);
    } else if (u.right == nil) {
      rotateRight(u);
    } else if (u.left.p < u.right.p) {
      rotateRight(u);
    } else {
      rotateLeft(u);
    }
    if (r == u) {
      r = u.parent;
    }
  }
}
```

図 7.8 に remove(x) の例を示した。

remove(x) の実行時間の解析におけるポイントは、add(x) の逆の操作になっていることだ。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)

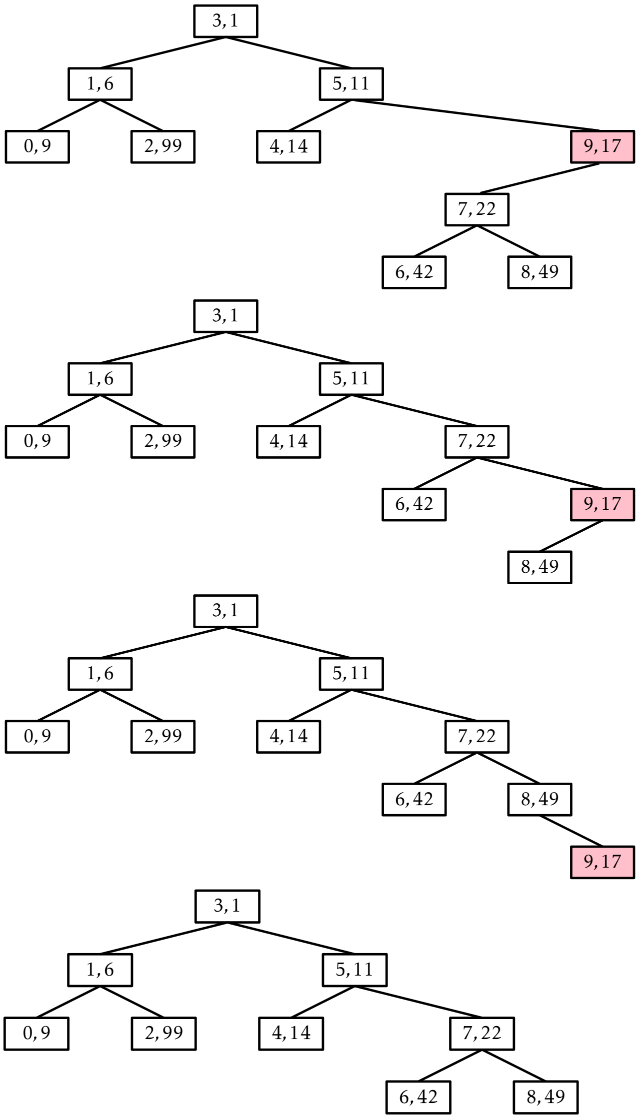


図 7.8: 図 7.5 の Treap から値 9 を削除する。

操作はちょうど同じ数の回転を実行し、Treap は 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 に追加している様子になっている。）これは大きさ n の Treap の remove(x) 操作の実行時間の期待値は、大きさ $n-1$ の Treap の 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

次の定理は Treap の性能をまとめるものだ。

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 を比べてみるのは面白いだろう。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) は検索に続く定数回のポインタの更新からなる。（下の問 7.5 を見よ）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次のようである。

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

Treap では次のようである。

$$2\ln n + O(1) \approx 1.386\log n + O(1)$$

よって Treap における探索経路の方が短く、各操作も Skiplist よりも Treap の方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って、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。

$$e\ln n + O(1) \approx 1.884\log n + O(1)$$

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のである。

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渡る。Devroye[19] が補題 7.1 とそれに関連する結果とを証明した。いくつかのより強い事実も示されているが、その中でもっとも印象的なのは Reed[64] 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は次の式で表せることを示した。

$$\alpha \ln n - \beta \ln\ln n + O(1)$$

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。加えて、高さの分散は定数であることも示している。

Treap という名前は Seidel と Aragon[67] が提案した。この文献は Treap とその変種いくつかについて述べている。しかし、基本的な Treap のアイデアは Vuillemin[76] が先んじて研究しており、この文献ではこのデータ構造を Cartesian tree と呼んでいた。

Treap のメモリ使用量を最適化する方法に、各ノードにおいて優先度 p を明示的には保存せずに済ます技法がある。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。このやり方で実用的には上手く動作するハッシュ関数が多いが、補題 7.1 の証明の正しさを保つためには、min-wise independent 性を満たす関数族からランダムに選出した関数をハッシュ関数として使わなければならない。min-wise independent 性とは次の性質である。相異なる任意の値 $x_1,\ldots,x_k$ について、それぞれのハッシュ値 $h(x_1),\ldots,h(x_k)$ は高い確率で相異なる値を取る。すなわち、ある定数 c が存在して、任意の $i\in\{1,\ldots,k\}$ について次の式が成り立つ。

$$\Pr\{h(x_i) = \min\{h(x_1),\ldots,h(x_k)\}\} \le c/k$$

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 tabulation hashing がある。(5.2.3 節を参照せよ。)

Treap の他の変種であって、優先度を各ノードに蓄えないものとして、Martínez と Roura [51] による動的ランダム二分探索木がある。任意のノード u は u を根とする部分木の大きさ u.size を保持する。add(x)・remove(x) いずれのアルゴリズムも乱択化されている。x を u を根とする部分木に追加するアルゴリズムは次のものである。

1. 確率 1/(size(u) + 1) で x はふつうに葉として追加され、回転によって x は部分木の根に向けて動いてくる。

2. そうでなければ（すなわち確率 $1-1/(\texttt{size}(u)+1)$ で、x は u.left または u.right の適切な方を根とする部分木に再帰的に追加される。

1 つめの場合は Treap における add(x) において x のノードがランダムな優先度として size(u) 個のいずれの値よりも小さい値を取る場合に対応しており、この事象の発生確率をそのままアルゴリズムに使っている。

x を動的ランダム二分探索木から削除するやり方は Treap における削除と似ている。x を含むノード u を見つけ、これが葉に到達するまで繰り返し深さを増やしながら回転し、そこで木から切り離す。各ステップにおける回転が右か左かをランダムに決める。

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。

こちらの確率も Treap において u で左右の回転を行う確率に等しいことを簡単に確認できる。

Treap と比べてこの動的ランダム二分探索木には短所がある。要素の追加・削除の際にランダムな選択を多く行うことと、部分木の大きさを保持しなければならないことだ。この動的ランダム二分探索木の長所は、部分木の大きさを他の目的にも使えることだ。例えばランクを $O(\log \texttt{n})$ の期待実行時間で計算する際にも流用できる。（問 7.10 を参照せよ。）一方、Treap の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ものである。実行時間は $O(\texttt{n})$ であり、n! 通りの置換がいずれも当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) における回転回数の期待値が $O(1)$ であ

ることを示せ。（このことから remove(x) の場合も同様のことがわかる。）

問 **7.6.** Treap の実装を明示的に優先度を保持しないように変更せよ。その際優先度として、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 は高さ u.height、u を根とする部分木の大きさ u.size 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ける 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 a の要素を順に add(x) メソッドで追加して得られる Treap と同一の Treap が得られなければならない。

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. 各ノードが自身を根とする部分木における最大値・最小値を保持する Treap を設計・実装せよ。
2. この情報を使って、fingerFind(x,u) を実装せよ。これは u の助けを借りて find(x) を実行する操作である。（u は x から遠くないノードであれば望ましい。）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 を見つける。その後は w からふつうのやり方で x を検索する。（fingerFind(x,u) の実行時間は $O(1+\log r)$ であることを示せる。ここで、r は Treap の要素であって、その値が x と u.x の間にあるものの数である。）
3. Treap の実装を拡張し、find(x) の探索を開始するノードを、直近の find(x) で見つかったノードとするようにせよ。

問 **7.10.** Treap におけるランクが i であるキーを返す操作 get(i) を設計・実装せよ。（ヒント：各ノード u が 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 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) 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 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() は定数時間で正しく動く必要がある。これは問 7.10 のために必要である。

問 **7.13.** split(x) の逆の操作と見なせる absorb(t2) をサポートする Treap を設計・実装せよ。この操作は Treap t2 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。また、この操作は t の最小値は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.** この節で説明した Martinez の randomized binary search trees を実装せよ。また、Treap の実装と性能を比較せよ。

第 8 章

Scapegoat Tree

この章では二分探索木の一種である ScapegoatTree を紹介する。これは、なにか誤りがあったときそれが誰の責任なのかを決めようとする、現実でよくある発想に基づくデータ構造である。(scapegoat とは罪を負わされたヤギ、転じて身代わりや生け贄のことである。) 責任の所在が決まれば、その scapegoat に問題を解決させられる。

ScapegoatTree 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（問 6.6）完全二分木のことである。また、完全二分木（10 章）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。 ノード 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やり方はたくさんある。もっとも単純なやり方のひとつは u の部分木を辿り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。$\mathtt{m} = \mathtt{a.length}/2$ 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<T> u) {
  int ns = size(u);
  Node<T> p = u.parent;
  Node<T>[] a =  Array.newInstance(Node.class, ns);
  packIntoArray(u, a, 0);
  if (p == nil) {
```

```
    r = buildBalanced(a, 0, ns);
    r.parent = nil;
  } else if (p.right == u) {
    p.right = buildBalanced(a, 0, ns);
    p.right.parent = p;
  } else {
    p.left = buildBalanced(a, 0, ns);
    p.left.parent = p;
  }
}
int packIntoArray(Node<T> u, Node<T>[] a, int i) {
  if (u == nil) {
    return i;
  }
  i = packIntoArray(u.left, a, i);
  a[i++] = u;
  return packIntoArray(u.right, a, i);
}
Node<T> buildBalanced(Node<T>[] a, int i, int ns) {
  if (ns == 0)
    return nil;
  int m = ns / 2;
  a[i + m].left = buildBalanced(a, i, m);
  if (a[i + m].left != nil)
    a[i + m].left.parent = a[i + m];
  a[i + m].right = buildBalanced(a, i + m + 1, ns - m - 1);
  if (a[i + m].right != nil)
    a[i + m].right.parent = a[i + m];
  return a[i + m];
}
```

rebuild(u) の実行時間は $O(\mathtt{size}(u))$ 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：部分再構築する二分探索木

ScapegoatTree は二分探索木である。ノード数 n に加えてノード数の上界 q とを保持する。

```
ScapegoatTree
int q;
```

n と q は常に次の関係を満たす。

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

加えて ScapegoatTree の高さはノードの数に対して対数程度であるという性質を持つ。すなわち scapegoat tree 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。

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

この制約を満たしていても、ScapegoatTree は意外と偏った見た目になりうる。例えば図 8.1 は $q = n = 10$ であり、高さ $5 < \log_{3/2} 10 \approx 5.679$ の木である。

ScapegoatTree における find(x) の実装は BinarySearchTree のアルゴリズムをそのまま使う。（6.2 節を参照せよ。）実行時間は木の高さに比例し、(8.1) よりこれは $O(\log n)$ である。

add(x) の実装でも、まず n と q をひとつずつ増やし、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、u の深さが運良く既に $\log_{3/2} q$ 以下なら、これ以上なにもしなくてよい。

depth(u) $> \log_{3/2} q$ になっていることもあり、この場合には高さを減らさなければならない。これはさほど難しくない。このとき、深さが $\log_{3/2} q$ を超えているノードは今加えた u だけである。修正するために、木を上に向かって辿りながら scapegoat であるノード w を探す。w は非常にバランスの悪いノー

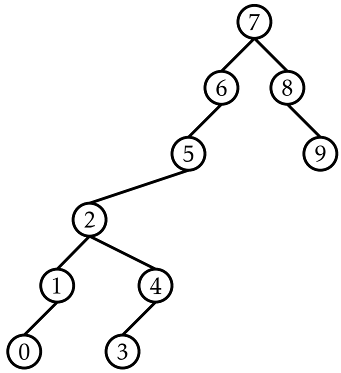


図 8.1: 10 個のノードを持ち、高さが 5 である ScapegoatTree の例

ドである。ここではバランスの悪さを次の式で判定する。

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

w.child は w の子であって、根から u に至る経路上にあるものである。scapegoat が存在することを示すのは難しくない。今はこれを事実として認めることにする。scapegoat である w が見つかれば、w 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ればよい。(8.2) より、u を加える前には w の部分木は非常にバランスのよい二分木ではない。よって、再構築後の木の高さは 1 以上減り、ScapegoatTree の高さは再び $\log_{3/2} \texttt{q}$ 以上になる。

ScapegoatTree
```
boolean add(T x) {
  // first do basic insertion keeping track of depth
  Node<T> u = newNode(x);
  int d = addWithDepth(u);
  if (d > log32(q)) {
    // depth exceeded, find scapegoat
    Node<T> w = u.parent;
    while (3*size(w) <= 2*size(w.parent))
```

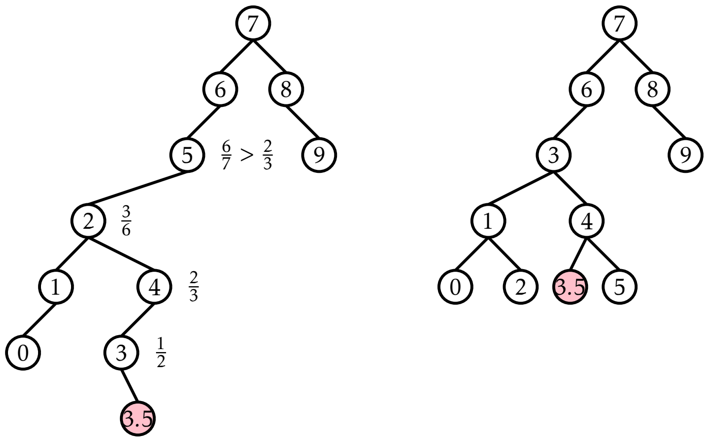


図 8.2: ScapegoatTree に 3.5 を追加する。このとき木の高さは 6 に増え、$6 > \log_{3/2} 11 \approx 5.914$ より (8.1) が成り立たない。scapegoat は値 5 を含むノードで見つかる。

```
      w = w.parent;
    rebuild(w.parent);
  }
  return d >= 0;
}
```

scapegoat w を見つけるコスト、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のものであり、これは $O(\log q) = O(\log n)$ である。scapegoat を見つけ、部分木を再構築するコストは、次小節で償却解析を使って説明する。

ScapegoatTree における remove(x) の実装は非常に単純である。x を探し、BinarySearchTree におけるアルゴリズムを使ってそれを削除する。（これは木の高さを増やすことはない。）そして n をひとつ小さくし、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、非常にバランスのよい二分木にして、$q = n$ とする。

ScapegoatTree
```
boolean remove(T x) {
  if (super.remove(x)) {
    if (2*n < q) {
      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の解析とを行う。まずは正しさを示すために、add(x) 操作において (8.1) が成り立たないなら scapegoat が見つかることを示す。

補題 8.1. u を *ScapegoatTree* における深さ $d > \log_{3/2} q$ のあるノードとする。 このとき u から *root* への経路上に次の条件を満たすノード w が存在する。

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

証明. 背理法で示す。u から root 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。

$$\frac{\mathtt{size}(\mathtt{w})}{\mathtt{size}(\mathtt{parent}(\mathtt{w}))} \le 2/3$$

また、根から u への経路を $\mathtt{r} = \mathtt{u}_0, \ldots, \mathtt{u}_d = \mathtt{u}$ とする。このとき $\mathtt{size}(\mathtt{u}_0) = \mathtt{n}$、$\mathtt{size}(\mathtt{u}_1) \le \frac{2}{3}\mathtt{n}$、$\mathtt{size}(\mathtt{u}_2) \le \frac{4}{9}\mathtt{n}$ であり、より一般に次の式が成り立つ。

$$\mathtt{size}(\mathtt{u}_i) \le \left(\frac{2}{3}\right)^i \mathtt{n}$$

ここで $\texttt{size}(\texttt{u}) \geq 1$ より、

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

こうして矛盾が導かれた。

続いてまだ説明していない部分の実行時間の解析を行う。scapegoat ノードを探す際の $\texttt{size}(\texttt{u})$・$\texttt{rebuild}(\texttt{w})$ のコストを改正する。これらふたつの操作の間には次のような関係がある。

補題 8.2. *ScapegoatTree* の $\texttt{add}(\texttt{x})$ において、*scapegoat* $\texttt{w}$ を見つけて $\texttt{w}$ を根とする部分木を再構築するコストは $O(\texttt{size}(\texttt{w}))$ である。

証明. scapegoat ノード $\texttt{w}$ を見つけた後、$\texttt{w}$ を根とする部分木の再構築を行う際の実行時間は $O(\texttt{size}(\texttt{w}))$ である。scapegoat を見つけるためには $\texttt{size}(\texttt{u})$ を $\texttt{u}_k = \texttt{w}$ を見つけるまで $\texttt{u}_0,\ldots,\texttt{u}_k$ に順に実行する。しかし、 $\texttt{u}_k$ はこの列における最初の scapegoat 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。

$$\texttt{size}(\texttt{u}_i) \leq \frac{2}{3}\texttt{size}(\texttt{u}_{i+1})$$

よって、すべての $\texttt{size}(\texttt{u})$ 呼び出しのコストの合計は次のようになる。

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

最後の行は等比数列の和を計算している。 □

最後に、m 個の操作を順に実行する時の $\texttt{rebuild}(\texttt{u})$ 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 $\texttt{add}(\texttt{x})$・$\texttt{remove}(\texttt{x})$ からなる操作の列を順に実行するとき、$\texttt{rebuild}(\texttt{u})$ 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($\mathtt{u}$) は $\mathtt{u}$ に蓄えられている預金を使って支払われる。

ノード $\mathtt{u}$ を挿入・削除の際に、根から $\mathtt{u}$ への経路上にある各ノードの預金を 1 だけ増やす。こうして一回の操作で増える預金の合計は最大 $\log_{3/2}\mathtt{q} \le \log_{3/2} m$ 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 $O(m\log m)$ だけの預金を行う。あとは、これだけの預金ですべての rebuild($\mathtt{u}$) の支払いに十分であることを示せばよい。

挿入の際に rebuild($\mathtt{u}$) を実行するなら、$\mathtt{u}$ は scapegoat である。次のことを仮定しても一般性を失わない。

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

次の事実を仮定使うと、

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

次の式が成り立つ。

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

このとき、さらに次の式が成り立つ。

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

$\mathtt{u}$ を含む部分木が直前に再構築されたとき（もし、$\mathtt{u}$ を含む部分木が一度も再構築されていなければ、$\mathtt{u}$ が挿入されたとき）、次の式が成り立つ。

$$\mathtt{size}(\mathtt{u.left}) - \mathtt{size}(\mathtt{u.right}) \le 1$$

よって、$\mathtt{u.left}$・$\mathtt{u.right}$ に影響を与えた add($\mathtt{x}$)・remove($\mathtt{x}$) の数の合計は次の値以上である。

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

$\mathtt{u}$ には少なくともこれだけの預金が蓄えられており、rebuild($\mathtt{u}$) に必要な $O(\mathtt{size}(\mathtt{u}))$ だけの支払いには十分である。

削除において rebuild($\mathtt{u}$) が呼ばれるとき、$\mathtt{q} > 2\mathtt{n}$ である。この場合、$\mathtt{q} - \mathtt{n} > \mathtt{n}$ だけ余分に預金が蓄えられており、根の再構築に必要な $O(\mathtt{n})$ だけの支払いには十分である。 □

8.1.2 要約

次の定理は ScapegoatTree の性能をまとめるものだ。

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、空の *ScapegoatTree* に対して、m 個の add(x)・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。

8.2 ディスカッションと練習問題

Galperin と Rivest[33] が scapegoat tree という名前を提案し、このデータ構造を定義し、解析した。しかし同じデータ構造が Andersson [5, 7] 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。これはこのデータ構造は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

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

である。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、既に計算した部分木のサイズを再利用したりできる。（8.5 と 8.6 を参照せよ。）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。

この本で紹介した他の SSet の実装と比べて、ScapegoatTree は再構築にかかる時間が長いため、性能が芳しくないのである。この本で紹介した他の 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 $n \log n$ のオーダーの時間を 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].

性能は劣るものの、ScapegoatTree が正しい選択となることもある。これは、各ノードに追加のデータを入れる場合で、特に、回転操作ではそれを定数時間では更新できないが、rebuild(u) では定数時間で更新できる場合であ

る。この場合 ScapegoatTree（や部分的な再構築を行う他のデータ構造）を選ぶのがよい。このような応用の例を問 8.11 では取り上げている。

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 に対して、$x = 1,2,3,\ldots,n$ 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は $c n \log n$ 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. 解析・実験によると、find(x)・add(x)・remove(x) の償却コストは n と b の関数としてどう表せるか。

問 **8.5.** ScapegoatTree の add(x) メソッドを修正し、既に計算した部分木の大きさは再計算せず、無駄を省くように修正せよ。size(w) を計算するとき、size(w.left) か size(w.right) は既に計算しているため、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 は**バランス条件** $\mathtt{size}(u) \le (2/3)\mathtt{size}(u.parent)$ を満たす。add(x)・remove(x) 操作はふつうの BinarySearchTree 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

構築される点でのみ異なっている。そして、WeightBalancedTree の償却実行時間は $O(\log n)$ 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。add(x)・remove(x) 操作はふつうの 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、u.t をひとつ小さくする点で異なる。u.t = 0 のとき、u を根とする部分木は非常にバランスのよい二分木に再構築される。ノード u が再構築に関わるとき（u が再構築されるか、u の祖先のうちのひとつが再構築されるとき）u.t は size(u)/3 にリセットされる。

そして、CountdownTree の償却実行時間は $O(\log n)$ であることを示せ。（ヒント：まずは任意のノードがあるバランスに関する不変条件を満たすことを示せ。）

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 を根とする部分着の大きさを u.size として保持する。add(x)・remove(x) 操作はふつうの 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、u は確率 1/u.size で**爆発**する。u が爆発すると、u を根とする部分木は非常にバランスのよい二分木に再構築される。

そして、DynamiteTree の実行時間の期待値は $O(\log n)$ 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する。また、新たに追加した要素を返す。（e が null なら新たな要素は列の先頭に追加される。）
- remove(e)：e を列から削除する。
- testBefore(e1,e2)：e1 が e2 の前にあるならば、またそのときに限って true を返す。

はじめのふたつの操作の償却実行時間は $O(\log n)$ でなければならない。みっつめの操作は定数時間でなければならない。

Sequence は、列の中の順序を使い、ScapegoatTree のようにデータを蓄えれば実装できる。testBefore(e1,e2) を定数時間で実装するために、要素 e は根から e への経路を符号化した整数でラベル付けされる。こうすると testBefore(e1,e2) は 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、要素数に対して木の高さを対数程度に抑え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。例えば、多くのライブラリの実装における基本的なデータ構造であり、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）に使われている。また、Linux という OS のカーネルでも使われている。赤黒木が人気な理由を挙げよう。

1. n 個の要素を含む赤黒木の高さは $2\log n$ 以下である
2. add(x)・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)・remove(x) における、回転の回数は償却すると定数である

はじめのふたつの性質が Skiplist・Treap・Scapegoat に対する赤黒木の優位性を示している。Skiplist・Treap はランダム化を使うため実行時間 $O(\log n)$ は期待値にすぎない。Scapegoat tree には高さの保証があるものの、add(x)・remove(x) の実行時間 $O(\log n)$ は償却実行時間にすぎない。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・削除するときの主な時間は、x を見つける処理によることがわかる。[*1]

しかし、赤黒木におけるよい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。様々な可能性を、慎重に解析しなければならない。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のである。ひとつ回転を間違えたり、色を間違えると、わかりにくいバグが発生することになる。

[*1] スキップリストや Treap も平均的にはこの性質を持つ。4.6 と 7.5 を参照せよ。

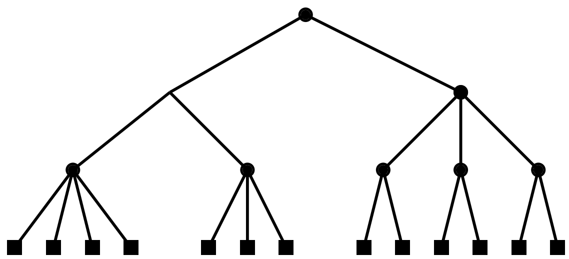


図 9.1: 高さ 3 である 2-4 木

赤黒木の実装に直接取り掛かるのでなく、まずは関連する背景知識として 2-4 木を説明する。これは赤黒木が発見された経緯と、なぜ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。2-4 木の性質より、この木の高さは葉の数の対数程度である。

補題 **9.1.** $\mathtt{n}$ 個の葉を持つ *2-4* 木の高さは $\log \mathtt{n}$ 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。

$$\mathtt{n} \geq 2^h \enspace .$$

両辺の対数を取ると $h \leq \log \mathtt{n}$ 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 を参照せよ。）他の葉の親ノード $\mathtt{w}$ の子として葉 $\mathtt{u}$ を追加したいときは、単に $\mathtt{u}$ を $\mathtt{w}$ 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 を追加する直前に w の 4 つの子を持っていたなら、w の子の数は今では 5 となる。この場合、 w を**分割**し、2 つの子を持つノード w と、3 つの子を持つノード 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。この処理は再帰的に行われる。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ず、その場合にはまた分割を行う。この処理を、子の数が 4 未満のノードが見つかるか、根 r を r と r′ に分割するまで繰り返す。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 を参照せよ。）葉 u をその親 w から削除するとき、単に u を削除する。その直前に、w がふたつしか子を持っていなければ、w の子はひとつだけになり、次数の制約を犯す。

これを修正するため、w の兄弟 w′ を見る。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′ が 3 つまたは 4 つの子を持つなら、そのうちひとつを w′ から w に移す。すると w の子の数は 2、w′ の子の数は 2 か 3 になり、処理を終えられる。

一方、w′ の子の数が 2 なら、w と w′ を**併合**し、子を 3 つ持つ一つのノード w とする。続いて w′ を w′ の親から取り除く。この処理はノード u かその兄弟 u′ が 3 つ以上子を持つような u を見つけるか、根に到達すると終了する。後者の場合、根はひとつの子だけを持つので、根は削除して、その子を新たな根とする。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。

ここでも、2-4 木の高さは常に $\log n$ 以下なので、葉の削除は $\log n$ 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。

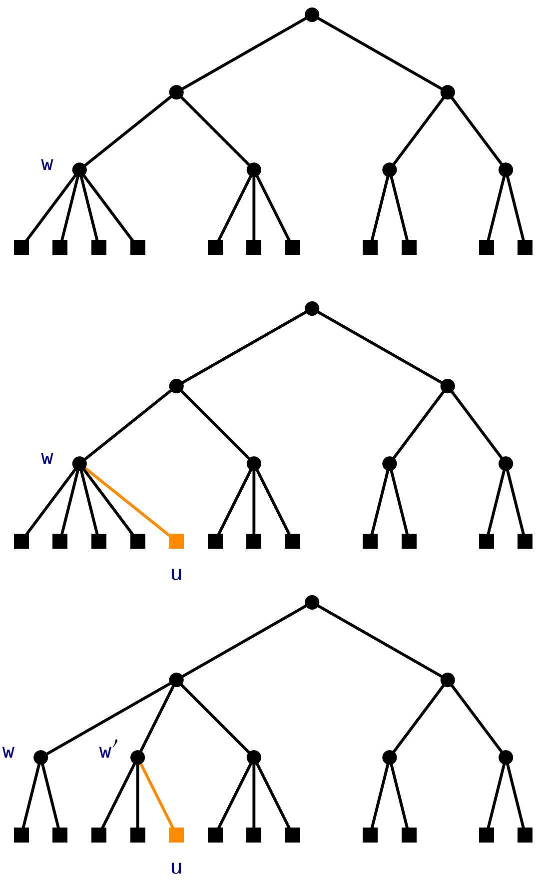


図 9.2: 2-4 木に葉を追加する。`w.parent` の次数が 4 未満なので、この処理は 1 回の分割の後終了する。

9.2 RedBlackTree：2-4 木のシミュレーション

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 Node<T> extends BSTNode<Node<T>,T> {
```

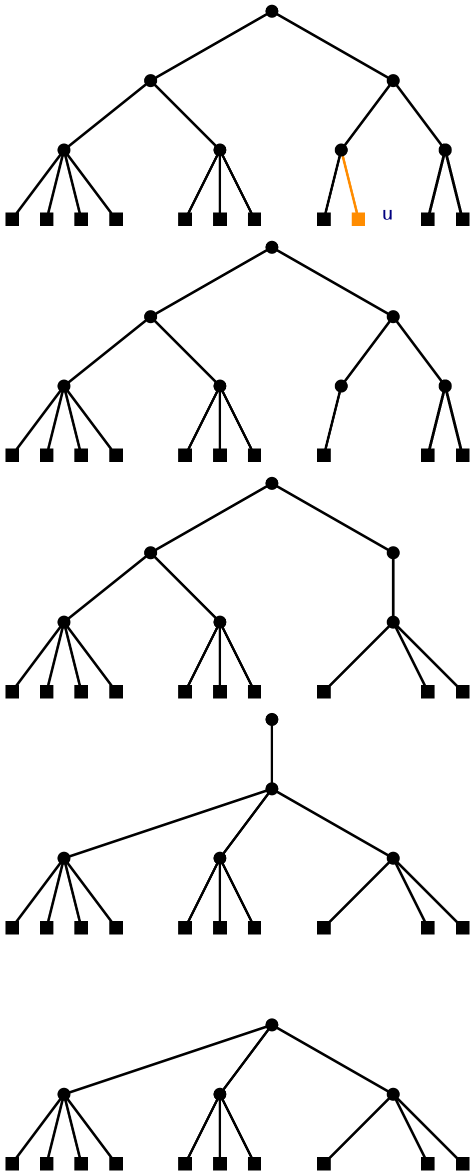


図 9.3: 2-4 木から葉を削除する。u の祖先とその兄弟はみな 2 つしか子を持っていないため、この処理は根まで繰り返される。

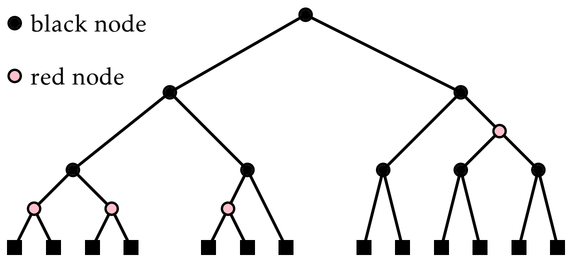


図 9.4: 黒の高さが 3 である赤黒木の例。外部ノード (`nil`) を正方形で描いている。

```
  byte colour;
}
```

赤黒木を操作する前後では次のふたつの性質が満たされる。いずれも赤・黒の色と、0・1 の数値を使って定義される。

性質 **9.3** (黒の高さの性質)**.** 「黒の高さ」が一様：葉から根への経路はいずれも、同じ数だけ黒いノードを含む。

性質 **9.4** (赤の辺の性質)**.** 赤の辺が無い：赤い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。)

根 `r` については、どちらの色であってもこれらの性質が満たされる。そのためここでは、根は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はこれを保つようにする。赤黒木を単純化するための別の工夫として、外部ノード（`nil` で表現される）を黒いノードと扱うのがよい。図 9.4 に赤黒木の例を示した。

9.2.1 赤黒木と 2-4 木

前の小節で定義した黒の高さと赤の辺についての性質を保ちながら、赤黒木を効率的に更新できる事実には、はじめて聞くと驚くかもしれない。一方で、これらの性質がなんの役に立つのかわからないかもしれない。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 を参照せよ。`n` 個のノードを持つ赤黒木 T に次の変換を施す。すべ

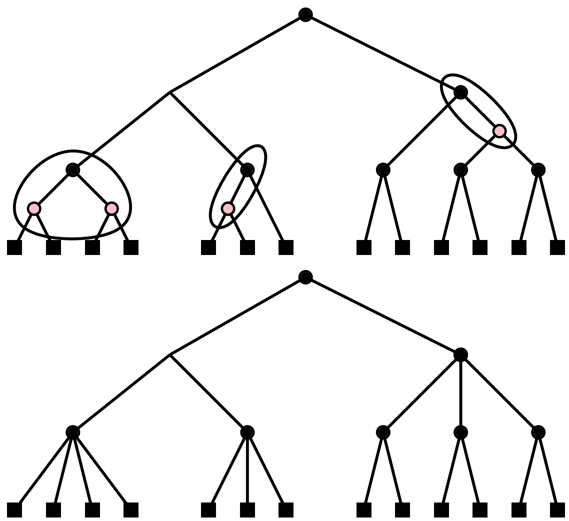


図 9.5: 任意の赤黒木には、対応する 2-4 木が存在する。

ての赤いノード u を取り除き、u のふたつの子をいずれも（黒い）u の親に直接接続する。こうして得られる木 T は黒いノードだけを含む。

T' の内部ノードはみな 2-4 個の子を持つ。ふたつの黒い子を持っていた黒いノードは、変換後も 2 つの黒い子を持つ。赤い子と黒い子をひとつずつ持っていた黒いノードは、変換後は 3 つの黒い子を持つ。ふたつの赤い子を持っていた黒いノードは、変換後は 4 つの黒い子を持つ。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・最後のノードは黒色で、内部ノードのふたつにひとつ以下が赤いので、この経路にあるノードのうち黒いものは $\log(n+1)$ 個以下、赤いものは $\log(n+1)-1$ 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、赤黒木を保ちながら効率的に要素の追加・削除ができる気がしてきたことだろう。

BinarySearchTree における要素の追加は新たな葉を追加することで行えることはこれまでの章で説明した。よって、赤黒木における add(x) を実装するためには、2-4 木における 5 つの子を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。5 つの子を持つ 2-4 木のノードは、ふたつの赤い子を持つひとつの黒いノード w であって、子のうちの一方が更に赤い子を持つものである。w を「分割」するには、w を赤く、w の子をいずれも黒くすればよい。例を図 9.6 に示す。

一方、remove(x) のためには、ふた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。ふたつのノードの併合は図 9.6 で示した分割の逆の処理であり、いずれも黒い兄弟を赤に、その共通の赤い親を黒にすればよい。兄弟から子を借りる操作が最も複雑で、回転と色の変更を共に行う必要がある。

もちろん赤の辺の性質、黒の高さの性質をいずれも満たす必要がある。これが可能なことくらいではもう驚かないかもしれないが、2-4 木を赤黒木でシミュレートするために考慮すべき場合分けはやはり多い。背景にある 2-4 木を無視して赤黒木の性質を保つことを直接的に考えることで、よりシンプルになることもある。

9.2.2 左傾赤黒木

赤黒木の定義の仕方は複数ある。add(x)・remove(x) を実行しながら、赤の辺の性質・黒の高さの性質を保てる、いくつかのデータ構造があるのだ。異なる構造では、異なるやり方でこれを実現する。この節では RedBlackTree と呼ぶデータ構造を実装する。これは赤黒木の一種であって、次の左傾性 (left-leaning) を満たすものである。

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 も黒である。

例えば図 9.4 の左傾性を満たしていない。右に向かう経路の赤いノードの親がこの性質を犯しているためだ。

左傾性を保持するのは、これにより add(x)・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ためである。これは対応する 2-4 木の表現

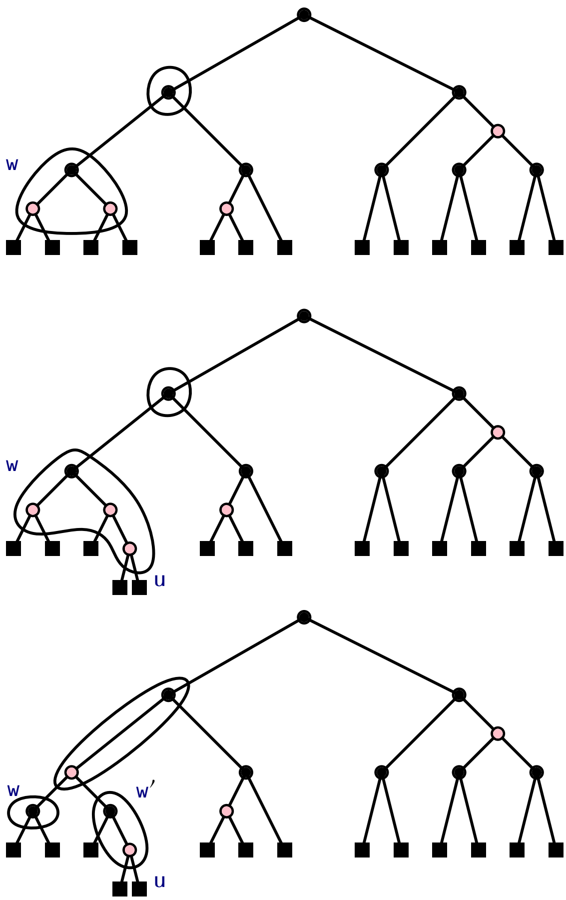


図 9.6: 2-4 木における追加の際の分割を赤黒木で模倣する。（これは図 9.2 に示した 2-4 木への要素の追加を模倣している。）

が一意に定まるためである。次数が 2 のノードは黒いノードであって、ふたつの黒い子を持つものである。次数が 3 のノードは黒いノードであって、左の子が赤く、右の子が黒の黒いものである。次数が 4 のノードは黒いノードであって、ふたつの赤い子を持つものである。

add(x)・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。最初のふたつのサブルーチンは黒い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 はふたつの赤い子を

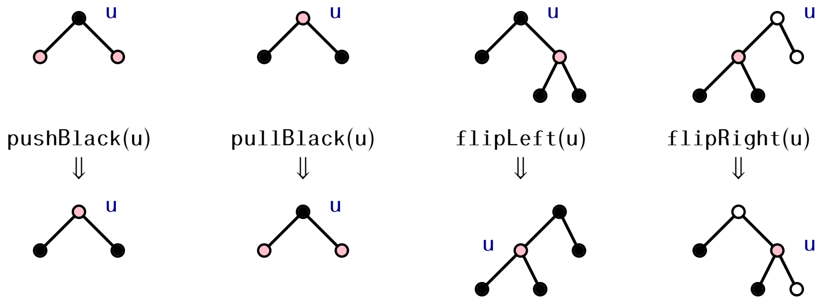


図 9.7: push・pull・flip

持つ黒いノードで、u を赤に、そのふたつの子をいずれも黒に塗り替える。pullBlack(u) はこの逆の操作である。

RedBlackTree
```
void pushBlack(Node<T> u) {
  u.colour--;
  u.left.colour++;
  u.right.colour++;
}
void pullBlack(Node<T> u) {
  u.colour++;
  u.left.colour--;
  u.right.colour--;
}
```

flipLeft(u) は u と u.right の色を入れ替え、その後 u を左回転する。この操作はこれらふたつのノードの色と親子関係をいずれも入れ替える。

RedBlackTree
```
void flipLeft(Node<T> u) {
  swapColors(u, u.right);
  rotateLeft(u);
```

```
}
```

flipLeft(u) は u が左傾性を犯しているとき、この性質を取り戻すのに役立つ。これは u.left が黒で u.right が赤であるためだ。この場合は特に、この操作によって黒の高さ・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<T> u) {
  swapColors(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 におけるふつう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の黒の高さも変わらないので、黒の高さの性質が犯されることはない。しかし、左傾性が犯される可能性がある。（これは u が右の子であるときである。）また赤の辺の性質を犯すかもしれない。（これは u の親が赤いときである。）これらの性質を取り戻すために、addFixup(u) を呼び出す。

```
RedBlackTree
boolean add(T x) {
  Node<T> u = newNode(x);
  u.colour = red;
  boolean added = add(u);
  if (added)
    addFixup(u);
  return added;
}
```

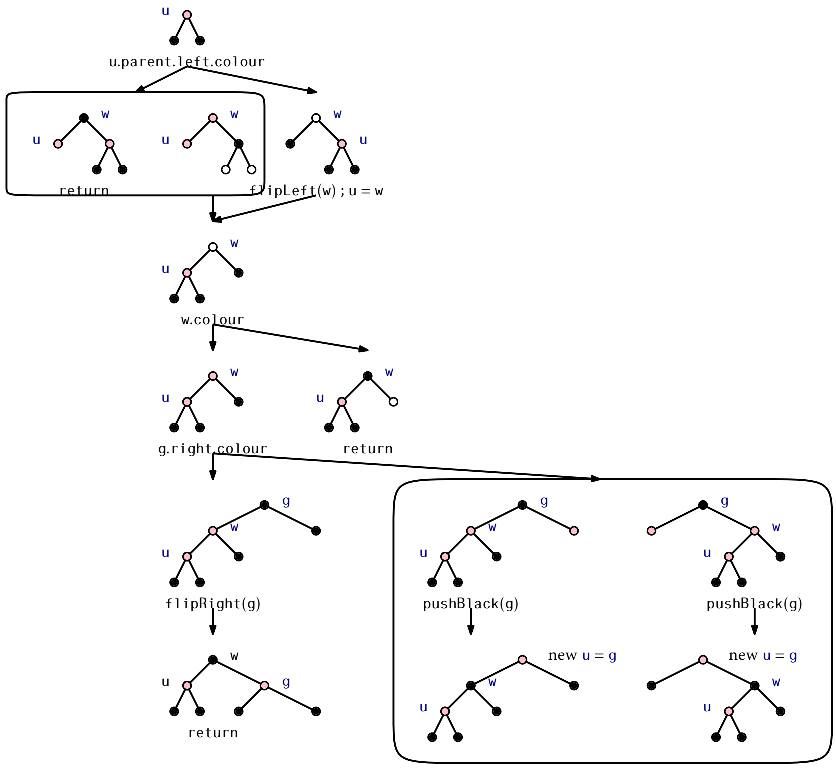


図 9.8: 要素を挿入したあと、ふたつの性質を満たすようにするための処理

図 9.8 に図示したように、addFixup(u) は赤いノード u を入力に取るが、これは赤の辺の性質や左傾性を犯しているかもしれない。この先の議論は、図 9.8 を見たり、それを一度紙に描いてみたりしなければ、理解するのが難しいかもしれない。続きを読む前に、この図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 を黒くすればふたつの性質が成り立つ。u の兄弟も赤いなら、u の親は黒いのでふたつの性質は既に成り立っている。

このいずれでもないとき、まずは u の親 w が左傾性を犯しているかを確認し、もしそうなら flipLeft(w) を実行し、u = w とする。すると、u は親である w の左の子になり、そのため w は左傾性を満たすようになる。あとは u の赤の辺の性質が満たされることを示せばよい。w が黒いなら既に赤の辺の性質は満たされているので、赤い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く、u と w はいずれも赤い。赤い辺の性質（u によって侵されているが、w によっては侵されていない）より、u には親の親 g が存在し、それは黒い。g の右の子が赤いなら左傾性より g の子は共に赤く、pushBlack(g) を呼ぶと g は赤く、w は黒くなる。すると、u について赤の辺の性質が満たされるが、g について赤の辺の性質が侵されるかもしれず、u = g として同じ処理を再度繰り返す。

もし g の右の子が黒いなら、flipRight(g) を呼べば w は g の（黒い）親になり、w は u と g のふたつの赤い子を持つ。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<T> u) {
  while (u.colour == red) {
    if (u == r) { // u is the root - done
      u.colour = black;
      return;
    }
    Node<T> w = u.parent;
    if (w.left.colour == black) { // ensure left-leaning
      flipLeft(w);
      u = w;
      w = u.parent;
    }
    if (w.colour == black)
      return; // no red-red edge = done
    Node<T> g = w.parent; // grandparent of u
    if (g.right.colour == black) {
      flipRight(g);
      return;
    } else {
      pushBlack(g);
      u = g;
```

```
    }
  }
}
```

insertFixup(u) の繰り返し一度あたりの実行時間は定数で、繰り返しの度に u を根に向けて動かすが処理が終了する。よって、insertFixup(u) は $O(\log n)$ 回の繰り返しの後に終了し、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 では remove(x) の実装が最も複雑であり、これは知られているどの赤黒木の場合でも同様である。**二分探索木**における remove(x) のように、この操作は唯一の子 u を持つノード w を特定し、u を u.parent と接続し、w を木から取り除く。

このとき w が黒いと、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、ふたつの別の問題が発生しうる。(1) u と w が共に黒いとき、u.colour + w.colour = 2 であり、不正な色 double-black になってしまう。(2) w が赤いとき、u と入れ替えると u.parent の左傾性が犯されるかもしれない。これらの問題はいずれも removeFixup(u) を呼ぶと解決できる。

RedBlackTree
```
boolean remove(T x) {
  Node<T> u = findLast(x);
  if (u == nil || compare(u.x, x) != 0)
    return false;
  Node<T> w = u.right;
  if (w == nil) {
    w = u;
    u = w.left;
  } else {
    while (w.left != nil)
      w = w.left;
    u.x = w.x;
```

```
    u = w.right;
  }
  splice(w);
  u.colour += w.colour;
  u.parent = w.parent;
  removeFixup(u);
  return true;
}
```

removeFixup(u) の入力であるノード u の色は 1 か 2（無効な色）である。u の色が 2 なら、removeFixup(u) は回転と色の塗り替え操作とを繰り返し、double-black 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しれない。具体的には赤から黒に変わっているかもしれず、そのため removeFixup(u) は最後に u の親が left-learning 性を満たしているか確認し、もし必要があればそれを修正する。

RedBlackTree

```
void removeFixup(Node<T> u) {
  while (u.colour > black) {
    if (u == r) {
      u.colour = black;
    } else if (u.parent.left.colour == red) {
      u = removeFixupCase1(u);
    } else if (u == u.parent.left) {
      u = removeFixupCase2(u);
    } else {
      u = removeFixupCase3(u);
    }
  }
  if (u != r) { // restore left-leaning property if needed
    Node<T> w = u.parent;
    if (w.right.colour == red && w.left.colour == black) {
```

```
      flipLeft(w);
    }
  }
}
```

図 9.9 は removeFixup(u) を図示したものだ。以降の説明も図 9.9 をよく見てからでないと理解するのは難しいだろう。removeFixup(u) の繰り返し毎に、double-black であるノード u は、次の 4 つの場合分けに従って処理される。

Case 0: u が根である場合である。このときは最も簡単である。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を犯すこともない。)

Case 1: u の兄弟 v が赤い場合である。このとき左傾性より、u の兄弟 v はその親 w 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 の親は左傾性を犯すようになり、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 が赤い場合の Case 3 である。このとき、後で説明する Case 3 を実行すれば、うまく処理できる。

```
RedBlackTree
Node<T> removeFixupCase1(Node<T> u) {
  flipRight(u.parent);
  return u;
}
```

Case 2: u の兄弟 v が黒い場合である。このとき、u は親 w の左の子である。続いて pullBlack(w) を呼ぶ。すると、u は黒く、v は赤くなり、w はより黒く、すなわち黒または double-black 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 は赤い。v を処理を開始した部分木の根とする。w が赤の辺の性質を犯していないか確認する必要がある。このために、w の右の子 q を見る。q が黒いなら、w は赤の辺の性質を満たしており、u = v として次の繰り返しに進める。

そうでない(q が赤い)なら、q と w によって、赤の辺の性質と左傾性とが満たされていない。左傾性を成り立たせるために、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犯されている。q は v の左の子で、w は q の左の子である。また、q と w はいずれも赤く、v は黒または

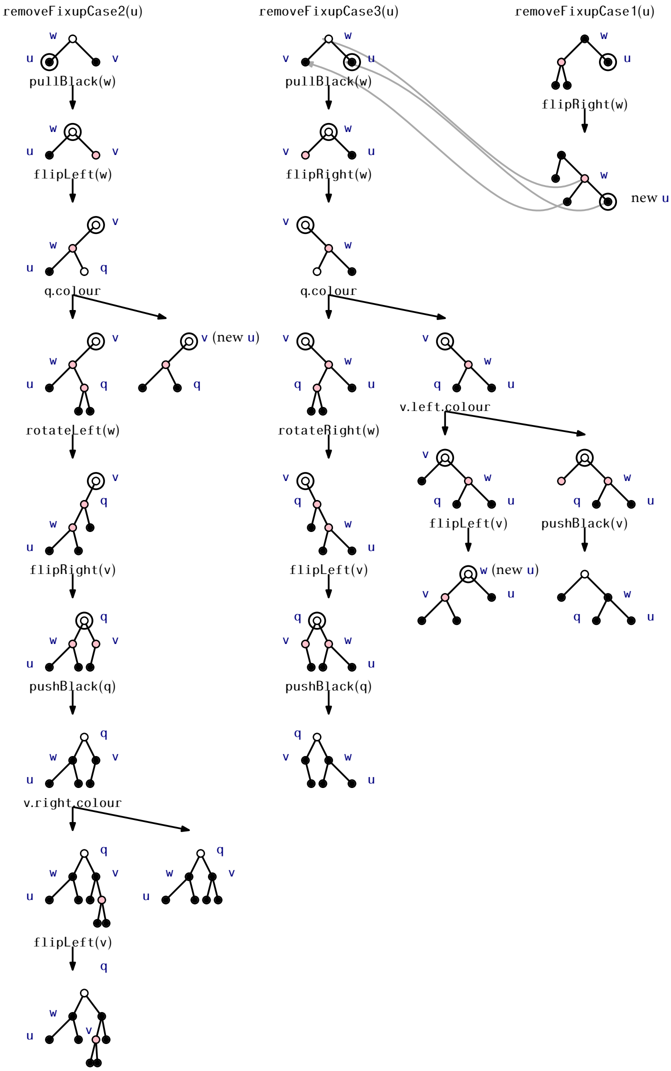


図 9.9: 削除のあと、double-black であるノードを追い出す様子

double-black である。flipRight(v) を実行すれば、q を v と w 両方の親になる。続いて、pushBlack(q) を呼ぶと v と w はいずれも黒くなり、q の色は w の元々の色になる。

すると、double-black のノードは無くなり、赤の辺の性質・黒の高さの性質はいずれも満たされる。あと気にする必要があるのは、v の右の子は赤いかもしれず、すると左傾性が犯されている。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<T> removeFixupCase2(Node<T> u) {
  Node<T> w = u.parent;
  Node<T> v = w.right;
  pullBlack(w); // w.left
  flipLeft(w); // w is now red
  Node<T> q = w.right;
  if (q.colour == red) { // q-w is red-red
    rotateLeft(w);
    flipRight(v);
    pushBlack(q);
    if (v.right.colour == red)
      flipLeft(v);
    return q;
  } else {
    return v;
  }
}
```

Case 3: u の兄弟が黒く、u が右の子である場合である。この場合は Case 2 と対称性があり、ほぼ同様に処理できる。唯一の違いは、左傾性が非対称であることから生じる。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 を赤く、u を黒くする。flipRight(w) を呼ぶと v が部分木の根になる。この時点で w は赤く、コードは w の左の子の色と q の色とに応じて分岐する。

q が赤いとき、Case 2 の場合と全く同様に処理を終えることが出来る。ただし、v が左傾性を満たさない心配はないので、より単純な処理で済ませられる。

より複雑なのは q が黒い場合の処理である。この場合、v の左の子の色を確認する。もし赤いなら v はふたつの赤い子を持っており、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 の元々の色になっており、処理を終えられる。

v の左の子が黒いなら、v は左傾性を犯しており、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<T> removeFixupCase3(Node<T> u) {
  Node<T> w = u.parent;
  Node<T> v = w.left;
  pullBlack(w);
  flipRight(w); // w is now red
  Node<T> q = w.left;
  if (q.colour == red) { // q-w is red-red
    rotateRight(w);
    flipLeft(v);
    pushBlack(q);
    return q;
  } else {
    if (v.left.colour == red) {
      pushBlack(v); // both v's children are red
      return v;
    } else { // ensure left-leaning
      flipLeft(v);
      return w;
    }
  }
```

```
}
```

`removeFixup(u)` の各繰り返しは定数時間で実行できる。Case 2・Case 3 は処理を終了するか、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 Case 3 に続き、その場合も処理は終了する。木の高さは高々 $2\log \mathtt{n}$ なので、高々 $O(\log \mathtt{n})$ 回の `removeFixup(u)` を繰り返せばよいことがわかるので、`removeFixup(u)` の実行時間は $O(\log \mathtt{n})$ である。

9.3 要約

次の定理は `RedBlackTree` の性能をまとめたものだ。

定理 9.1. *RedBlackTree* は *SSet* インターフェースの実装である。*RedBlackTree* は操作 `add(x)`・`remove(x)`・`find(x)` を持ち、いずれの最悪実行時間も $O(\log \mathtt{n})$ 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)`・`remove(x)` からなる操作の列を実行するときの、`addFixup(u)`・`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にする。`addFixup(u)`・`removeFixup(u)` と、2-4 木における葉の追加・削除とを比べると、この性質は 2-4 木に由来するものという気がしてくるだろう。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 を導けるだろう。

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。

$$\Phi(\mathtt{u}) = \begin{cases} 1 & \mathtt{u} \text{ の子の数が } 2 \text{ のとき} \\ 0 & \mathtt{u} \text{ の子の数が } 3 \text{ のとき} \\ 3 & \mathtt{u} \text{ のこの数が } 4 \text{ のとき} \end{cases}$$

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 つと 3 つの子を持

[*3] ポテンシャル法の他の例としては、補題 2.2・補題 3.1 の証明を参照せよ。

つふたつのノードになる。すなわち、全体のポテンシャルは $3-1-0=2$ 小さくなる。併合の際には、それぞれふたつの子を持っていたふたつのノードが、3 つの子を持つ一つのノードになる。このとき、全体のポテンシャルは $2-0=2$ 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。

分割と併合以外では、葉を加えたり削除したりして、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、ひとつのノードの子が 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、ひとつのノードの子が 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ふた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・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・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[38] によってはじめに提案された。赤黒木は実装が非常に複雑であるにも関わらず、ライブラリやアプリで最も頻繁に使われるもののうちの一つである。ほとんどのアルゴリズムやデータ構造の本では何種類かの赤黒木を説明している。

Andersson [6] は左傾なバランスされた木であって、赤黒木に似ているが任意のノードは最大で一つの赤い子を持つという制約を加えたものを提案している。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。しかし、このデータ構造はこの章で説明した `RedBlackTree` よりもかなり単純である。

Sedgewick [66] は二種類の左傾赤黒木を提案している。こ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・併合のシミュレーションに加えて、再帰を使っている。こうすると、プログラムが短くエレガントになる。

関連するより古いデータ構造としては AVL **木** [3] がある。AVL 木は次の height-balanced 性を満たす木である。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。$F(h)$ を高さ h である木のうち、葉が最も少ないものの葉の数とする

とき、$F(h)$ は次のフィボナッチの漸化式を満たす。

$$F(h) = F(h-1) + F(h-2)$$

ただし $F(0) = 1, F(1) = 1$ である。ここで**黄金比** $\varphi = (1+\sqrt{5})/2 \approx 1.61803399$ 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、これは式を含意する。

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

よって、AVL 木の高さは赤黒木の高さよりも低い。`add(x)`・`remove(x)` の際、根に向かって戻り、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 以上異なるときバランスの再調整を行うことで高さのバランスを保つ。図 9.10 を参照せよ。

Andersson のものや Sedgewick のもの、AVL 木のいずれも `RedBlackTree` よりも実装は単純である。しかし、いずれもバランスの再調整の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。

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 $2\log \mathtt{n} - O(1)$ である赤黒木が存在することを示せ。

問 **9.5.** 操作 `pushBlack(u)`・`pullBlack(u)` を考える。これらの操作は赤黒木によって表現されている 2-4 木に対して何を行うか。

問 **9.6.** どれだけ大きな `n` についても、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 $2\log \mathtt{n} - O(1)$ である赤黒木を校正する `add(x)`・`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。これは `splice(w)` において既に行われているはずではないか。

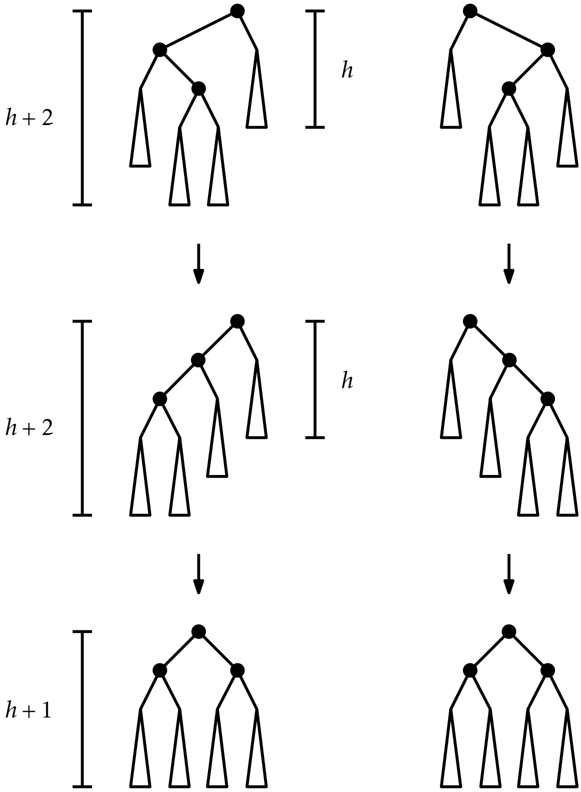


図 9.10: AVL 木におけるバランスの再調整の様子。その子の部分木の大きさが h と $h+2$ であるノードについて、いずれの子の部分木の高さも $h+1$ とする際に必要な回転は高々 2 回である。

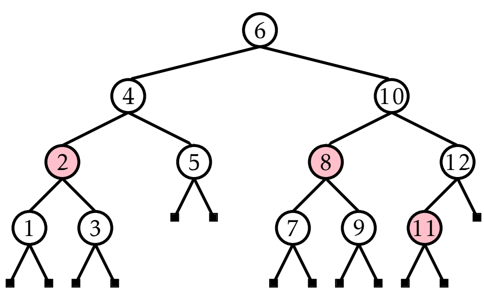


図 9.11: 練習問題のための赤黒木

問 **9.8.** 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' の持つ赤い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 $2\log \mathsf{n} - 2$ 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。もしそう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, 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ものであるとする。このとき、$O(h)$ の時間で T_1 と T_2 を一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, T_2 の黒い高さ h_1, 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。(このとき最大二度の回転を実行することになる。図 9.10 を参照せよ。) また、`remove(x)` 操作の際にオーダーで $\log \mathsf{n}$ だけのバランスの再調整操作が必要な AVL 木の例を挙げよ。

問 **9.13.** 上で説明した AVL 木の実装である `AVLTree` クラスを作成せよ。この性能と `RedBlackTree` の性能とを比較せよ。`find(x)` が高速に行える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一連の実験を設計・実装せよ。なお、ランダムなデータや整列済みのデータ、ランダム・規則正しい順序での削除などを含む、多くのテストのシナリオを作ること。

第 10 章

ヒープ

この章では優先度付きキューの実装を 2 つ説明する。いずれも特殊な二分木であり、**ヒープ** (heap)（雑多に積まれた山）と呼ばれている。これは二分探索木が高度に構造化されながら積み上げられていたのとは対照的である。

ひとつめのヒープの実装は配列を使って完全二分木をシミュレートする。これを使ってヒープソート（11.1.3 節参照）というの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のもののひとつである。ふたつめに紹介する実装は、より柔軟な二分木である。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サポートする。

10.1 BinaryHeap：非明示的な二分木

優先度付きキューの最初の実装は 400 年以上前に発見された手法に基づく。Eytzinger **法**では木のノードを幅優先順（6.1.2 節）に配列に入れて完全二分木を表現する。こうすると、根は 0 番目、根の左の子は 1 番目、右の子は 2 番目、根の左の子の左の子は 3 番目の位置に格納される。図 10.1 を参照せよ。

Eytzinger 法を大きな木に適用してみると、規則性が見えてくる。添え字 i のノードの左の子の添え字は $\texttt{left}(i) = 2i + 1$ であり、右の子の添え字は $\texttt{right}(i) = 2i + 2$ である。また、添え字 i のノードの親の添え字は $\texttt{parent}(i) = (i - 1)/2$ である。

BinaryHeap
```
int left(int i) {
```

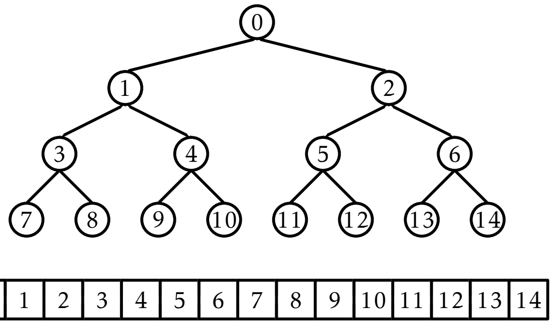


図 10.1: Eytzinger 法により、配列を使って完全二分木を表現する。

```
  return 2*i + 1;
}
int right(int i) {
  return 2*i + 2;
}
int parent(int i) {
  return (i-1)/2;
}
```

BinaryHeap はこの手法を使って完全二分木を非明示的に表現する。特に、この木の中では要素は**ヒープ順** (heap order) に並んでいる。すなわち、添え字 i の位置に格納されている値は、parent(i) に格納されている値以上である。（ただし、i = 0 である根は除く。）このとき、優先度付き Queue における最小値が 0 番目の位置（根）に格納されていることがわかる。

BinaryHeap では n 個の要素が配列 a に格納されている。

```
BinaryHeap
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 を参照せよ。

BinaryHeap
```
boolean add(T x) {
  if (n + 1 > a.length) resize();
  a[n++] = x;
  bubbleUp(n-1);
  return true;
}
void bubbleUp(int i) {
  int p = parent(i);
  while (i > 0 && compare(a[i], a[p]) < 0) {
    swap(i,p);
    i = p;
    p = parent(i);
  }
}
```

remove() はヒープから最小の値を削除するが、この実装にはすこし工夫が必要だ。根が最小値なのはわかっているが、これを削除したあとにもヒープ性が成り立つことを保証しなければならない。

もっと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、n を 1 小さくすることだ。しかし、その結果新たな根はおそらく最小値ではなくなっている。そのためこれを下方向に動かす必要がある。このため、この要素を子供と比較することを繰り返す。もしこの要素が三つ（自分と子供達）のうち最小ならば処理を終了する。そうでないなら、子供達のうち小さいものと、この要素とを入れ替え、処理を繰り返す。

BinaryHeap
```
T remove() {
  T x = a[0];
```

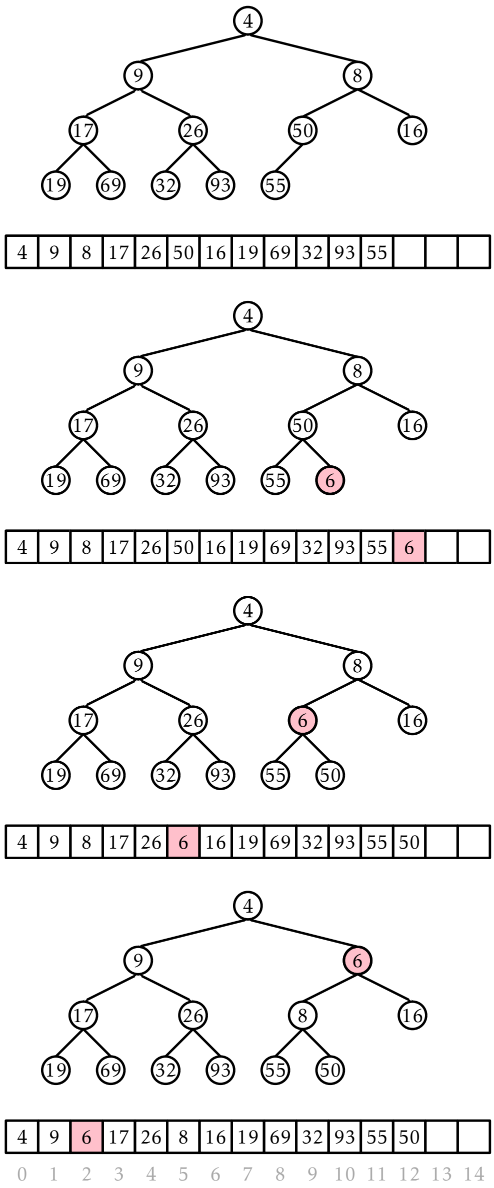


図 10.2: BinaryHeap に値 6 を追加する。

```
    a[0] = a[--n];
    trickleDown(0);
    if (3*n < a.length) resize();
    return x;
  }
  void trickleDown(int i) {
    do {
      int j = -1;
      int r = right(i);
      if (r < n && compare(a[r], a[i]) < 0) {
        int l = left(i);
        if (compare(a[l], a[r]) < 0) {
          j = l;
        } else {
          j = r;
        }
      } else {
        int l = left(i);
        if (l < n && compare(a[l], a[i]) < 0) {
          j = l;
        }
      }
      if (j >= 0)  swap(i, j);
      i = j;
    } while (i >= 0);
  }
```

他の配列ベースのデータ構造と同様に、resize() のための時間を無視する。この実行時間は補題 2.1 の償却解析によってわかる。すると、add(x)・remove() の実行時間は（非明示の）二分木の高さに依存する。嬉しいことに、これは**完全**二分木 (complete binary tree) である。最大の深さを除く各深さには、可能な最大数のノードがある。 よって、h を木の高さとすると、少なく

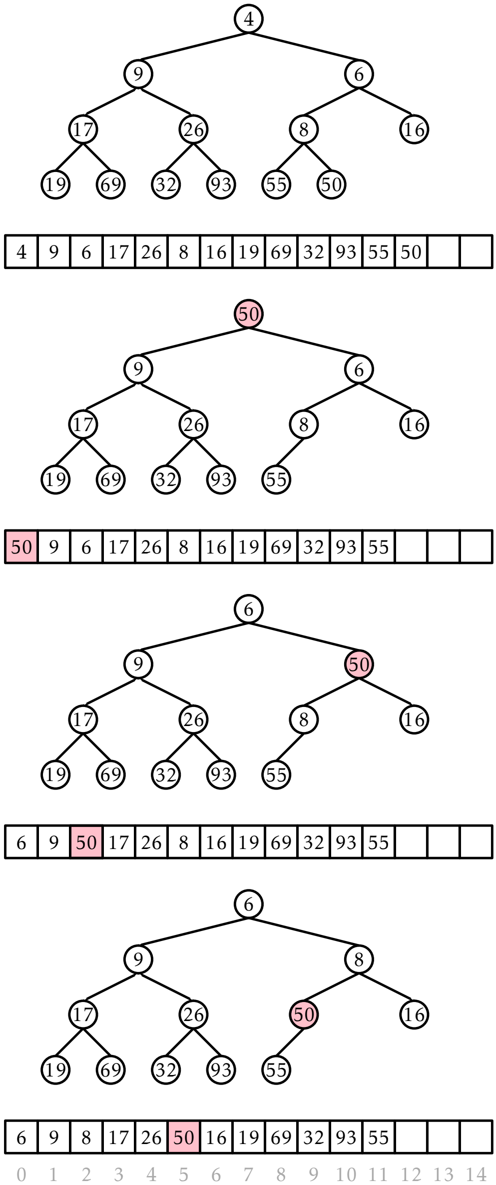


図 10.3: BinaryHeap から最小の値 4 を削除する。

とも 2^h 個のノードがある。言い換えると、次の式が成り立つ。

$$\mathtt{n} \geq 2^h$$

両辺の対数を取ると、次の式が成り立つ。

$$h \leq \log \mathtt{n}$$

以上より、add(x)・remove() のいずれの実行時間も $O(\log \mathtt{n})$ である。

10.1.1 要約

次の定理は BinaryHeap の性能をまとめたものだ。

定理 10.1. *BinaryHeap* は（優先度付き）*Queue* インターフェースの実装である。*BinaryHeap* は add(x)・remove() をサポートし、resize() のコストを無視すると、いずれの実行時間も $O(\log \mathtt{n})$ である。

空の *BinaryHeap* から任意の m 個の add(x)・remove() からなる操作の列を実行する。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：ランダムな Meldable ヒープ

この節では、MeldableHeap を紹介する。これは優先度付き Queue の実装で、背後にある構造もまたヒープであるものである。しかし、BinaryHeap のようには要素数から二分木の形が一意には決まらず、MeldableHeap における背後にある二分木の形状には制約がない。

MeldableHeap における add(x)・remove() は merge(h1,h2) 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、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 を参照せよ。h1 または h2 が nil なら、単に h2, h1 をそれぞれ返せばよい。そうでないとき、$\mathtt{h1.x} \leq \mathtt{h2.x}$ として一般性を失わない。このとき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<T> merge(Node<T> h1, Node<T> h2) {
  if (h1 == nil) return h2;
  if (h2 == nil) return h1;
  if (compare(h2.x, h1.x) < 0) return merge(h2, h1);
  // now we know h1.x <= h2.x
  if (rand.nextBoolean()) {
    h1.left = merge(h1.left, h2);
    h1.left.parent = h1;
  } else {
    h1.right = merge(h1.right, h2);
    h1.right.parent = h1;
  }
  return h1;
}
```

次の節では merge(h1,h2) の実行時間の期待値が $O(\log n)$ であることを示す。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) は簡単である。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
```
boolean add(T x) {
  Node<T> u = newNode();
  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() も同様に簡単である。削除したいのは根なので、そのふたつの子

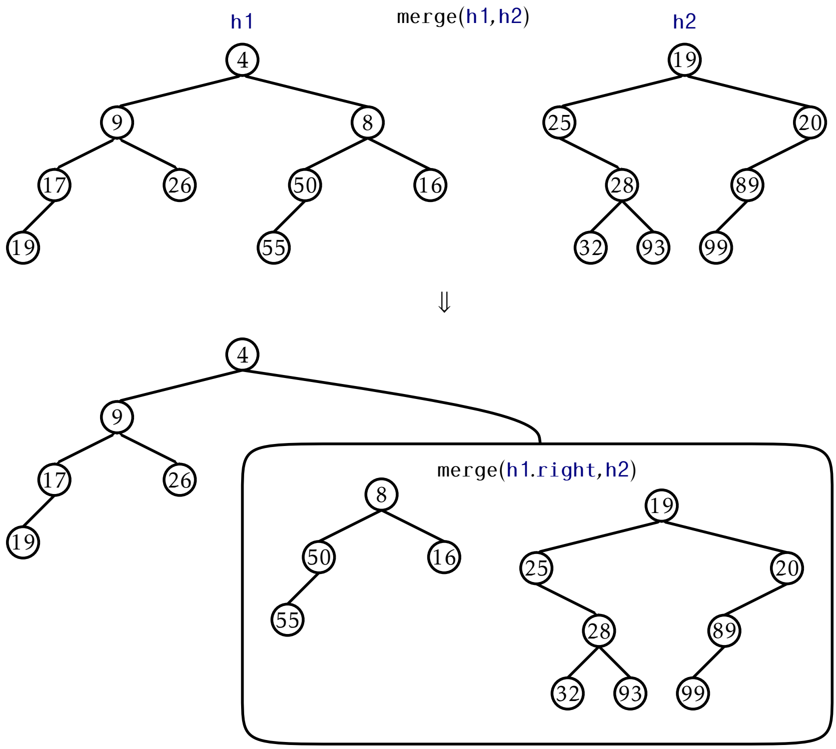


図 10.4: h1 と h2 を併合するために、h1.left または h1.right のいずれかに h2 を併合する。

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;
  r = merge(r.left, r.right);
  if (r != nil) r.parent = nil;
  n--;
  return x;
}
```

このときも実行時間の期待値は $O(\log \mathsf{n})$ である。

実行時間の期待値が $O(\log \mathsf{n})$ である `MeldableHeap` の操作は他にもいくつかある。例えば次のものである。

- `remove(u)`：ヒープからノード `u` を削除する
- `absorb(h)`：このヒープに `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だけの `merge(h1,h2)` を使って実装できる。

10.2.1 `merge(h1,h2)` の解析

`merge(h1,h2)` の解析は二分木のランダムウォークに基づく。二分木の**ランダムウォーク**は根から始まる。各ステップではコインを投げ、その結果に応じて今のノードから右の子か左の子に進む。木からはみ出ると処理を終了する。（これは進んだ先が `nil` であったときである。）

次の補題は注目に値する。この補題は二分木の形状に関わらず成り立つ。

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 $\log(\mathsf{n}+1)$ 以下である。

証明. n に関する帰納法により証明する。$\mathsf{n}=0$ のとき、ステップ数は $0=\log(\mathsf{n}+1)$ である。以下では、任意の非負整数 $\mathsf{n}'<\mathsf{n}$ 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、$\mathsf{n}_2=\mathsf{n}-\mathsf{n}_1-1$ が右の部分木の大きさである。根からはじめて、一歩進み、大きさ n_1 または n_2 の部分木について処理を続ける。仮定より、ステップ数の期待値は次のようになる。

$$\mathrm{E}[W] \le 1+\frac{1}{2}\log(\mathsf{n}_1+1)+\frac{1}{2}\log(\mathsf{n}_2+1)$$

これは n_1 と n_2 はいずれも n より小さいためである。log は凹関数（上に凸な関数）なので、$\mathrm{E}[W]$ は $\mathsf{n}_1=\mathsf{n}_2=(\mathsf{n}-1)/2$ 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。

$$\begin{aligned}\mathrm{E}[W] &\le 1+\frac{1}{2}\log(\mathsf{n}_1+1)+\frac{1}{2}\log(\mathsf{n}_2+1)\\ &\le 1+\log((\mathsf{n}-1)/2+1)\\ &= 1+\log((\mathsf{n}+1)/2)\\ &= \log(\mathsf{n}+1)\end{aligned}$$

□

余談だが、情報理論について知っている読者は補題 10.1 の証明をエントロピーの用語で言い換えることができるだろう。

補題 *10.1* の情報理論的な証明. d_i を i 番目の外部ノード (nil) の深さとする。n 個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を思い出そう。ランダムウォークが i 番目の外部ノードにたどり着く確率は $p_i = 1/2^{d_i}$ 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。

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

右辺は n + 1 個の要素に渡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 個の要素にわたる分布のエントロピーに関する基本的な事実として、これはこの値は $\log(\mathtt{n}+1)$ 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 $O(\log \mathtt{n})$ であることを示せる。

補題 10.2. h1・h2 はふたつのヒープの根であり、それぞれのヒープは $\mathtt{n}_1$, $\mathtt{n}_2$ 個のノードを含むとする。このとき、merge(h1,h2) の実行時間は $O(\log \mathtt{n})$ 以下である。ただし、ここで $\mathtt{n} = \mathtt{n}_1 + \mathtt{n}_2$ である。

証明. merge の各ステップはランダムウォークを 1 ステップで、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ふたつのランダムウォークのどちらかが木からはみ出すと終了する。（これは h1 = null または h2 = null のときである。）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。

$$\log(\mathtt{n}_1 + 1) + \log(\mathtt{n}_2 + 1) \le 2 \log \mathtt{n}$$ □

10.2.2　要約

次の定理は MeldableHeap をまとめるものである。

定理 10.2. *MeldableHeap* は優先度付き *Queue* インターフェースを実装する。add(x)・remove() をサポートし、いずれの実行時間の期待値も $O(\log \mathtt{n})$ 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を使って非明示的に表現する方法は Eytzinger [27] が最初に提案したようである。彼は貴族の家系図が書いてある本でこれを使った。この章で説明した BinaryHeap は Williams [78] が最初に提案したものである。

この章で説明したランダム操作を利用した MeldableHeap は Gambin と Malinowski [34] が最初に提案した。他の Meldable Heap も存在し、leftist heaps [16, 48, Section 5.3.2]、binomial heaps [75]、Fibonacci heaps [30]、pairing heaps [29]、skew heaps [72] などがある。しかし、いずれも MeldableHeap ほどシンプルではない。

上のデータ構造の中には decreaseKey(u, y) 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 とするものである。（これを実行する前には $y \le u.x$ である必要がある。）上に挙げたデータ構造の大部分では、ノード u を削除し、y を追加するので $O(\log n)$ の時間がかかる。しかし、一部のデータ構造ではこれをもっと効率的に実装できる。とくに、Fibonacci heaps では $O(1)$ の償却実行時間で、pairing heaps [25] の特殊なものでは $O(\log\log n)$ の償却実行時間でそれぞれ decreaseKey(u, y) を実行できる。効率的な decreaseKey(u, y) はグラフアルゴリズムの高速化に役立つ。（例えばダイクストラ法）[30]

問 **10.1.** 図 10.2 に示した BinaryHeap に 7, 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 から次のふたつの値（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。すなわち、添え字 i が与えられたとき、i の親と、i の d 個の子、それぞれの添え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を使うこと。

問 **10.7.** 図 10.4 に示した MeldableHeap から、次のふたつの値（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 $O(\log \mathtt{n})$ でなければならない。

問 **10.9.** BinaryHeap・MeldableHeap における二番目に小さい値を定数時間で見つける方法を示せ。

問 **10.10.** BinaryHeap・MeldableHeap における k 番目に小さい値を $O(k\log k)$ の時間で見つける方法を示せ。（ヒント：別のヒープを使うといいかもしれない。）

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一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るのがよいかもしれない。）

第 11 章

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のを奇妙に思うかもしれないが、これにはいくつかの理由がある。例えばここで紹介する整列アルゴリズム（クイックソートとヒープソート）はこれまでに学んだデータ構造（ランダム二分探索木とヒープ）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であり、その実行時間が $O(n\log n)$ であるものを 3 つ紹介する。またこの実行時間は漸近的に最適であることも示す。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 $n\log n$ 回の比較を行う必要があるのだ。

ここで、これまでの章で説明した、整列済みの集合 SSet や優先度付き Queue を使って、$O(n\log n)$ の時間で整列アルゴリズムを実装できることに触れておく。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、いずれかの二分探索木に n 回 add(x) を実行し、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。しかし、いずれの場合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。 整列は重要な問題なので、可能な限り速く、単純で、省メモリな手法を開発する価値がある。

この章の後半では、比較以外の操作も使える場合には、さらによい実行時間を実現できることを見ていく。配列のランダムアクセスを使って、$\{0,\ldots,n^c-1\}$ の要素 n 個の整列を $O(cn)$ の時間で実行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 3 つの整列アルゴリズム、マージソート・クイックソート・ヒープソートを紹介する。いずれも配列 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。どれも**比較に基づく**アルゴリズムである。第二引数 c は Comparator である。これは compare(a,b) メソッドを実装するオブジェクトである。整列するデータの型はなんでもよい。ただ、データの比較のための compare(a,b) メソッドを持っていなければならない。1.2.4 節でも説明したが、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。

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古典的な例である。配列 a の長さが 1 以下なら、a は既に整列されており、なにもする必要はない。そうでなければ、配列 a を半分ずつに、すなわち配列 $a0 = a[0], \ldots, a[n/2-1]$ と配列 $a1 = a[n/2], \ldots, a[n-1]$ とに分ける。再帰的に a0 と a1 を整列し、そして（整列済みの）a0 と a1 とを併合し、整列済みの配列 a を得る [*1]。

Algorithms
```
<T> void mergeSort(T[] a, Comparator<T> c) {
  if (a.length <= 1) return;
  T[] a0 = Arrays.copyOfRange(a, 0, a.length/2);
  T[] a1 = Arrays.copyOfRange(a, a.length/2, a.length);
  mergeSort(a0, c);
  mergeSort(a1, c);
  merge(a0, a1, a, c);
}
```

図 11.1 に例を示した。

整列と比べて、a0 と a1 を併合するのは簡単である。a に要素を一つずつ加

[*1] 訳注：Java の Arrays.copyOfRange(T[] original, int from, int to) メソッドは、to をコピー対象として含まないことに注意。そのため擬似コードでは a.length/2 が from と to で一度ずつ用いられている。

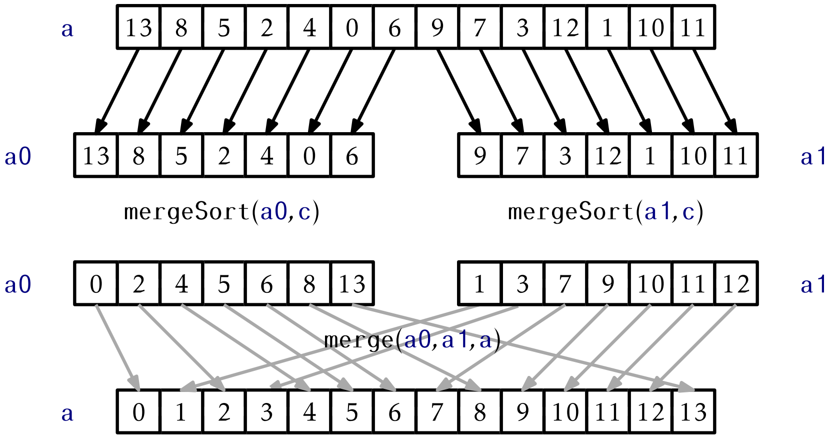


図 11.1: mergeSort(a,c) を実行する様子

えていく。もし a0 か a1 が空になれば、空でない方の配列の残りの要素をすべて a に加える。そうでなければ、a0 の次の要素と a1 の次の要素のうち、小さい方を a に加える。

Algorithms

```
<T> void merge(T[] a0, T[] a1, T[] a, Comparator<T> c) {
  int i0 = 0, i1 = 0;
  for (int i = 0; i < a.length; i++) {
    if (i0 == a0.length)
      a[i] = a1[i1++];
    else if (i1 == a1.length)
      a[i] = a0[i0++];
    else if (compare(a0[i0], a1[i1]) < 0)
      a[i] = a0[i0++];
    else
      a[i] = a1[i1++];
  }
}
```

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 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ためには図 11.2 に示したような再帰木を考えるのがよい。n が 2 の冪乗であると仮定する。すなわち、$n = 2^{\log n}$ であり、$\log n$ は整数である。マージソートは n 個の要素の整列問題を、2 つの「$n/2$ 個の要素の並び替え問題」に変換する。次に、これらの部分問題はそれぞれふたつの問題に変換され、4 つの「$n/4$ 個の要素の並び替え問題」になる。そして、この 4 つの部分問題は 8 つの「大きさ $n/8$ の問題」になる。このようなことを繰り返す。最終ステップでは、$n/2$ 個の「大きさ 2 の部分問題」が、n 個の「大きさ 1 の問題」に変換される。大きさ $n/2^i$ の問題を解く際に、既に解かれた 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かかる時間は $O(n/2^i)$ である。2^i 個の大きさ $n/2^i$ の問題があるので、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。（これはまだ再帰的には数えていないことに注意する。）

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

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。

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

次の定理の証明は以上の解析に基づくが、n が 2 の冪乗でない場合を扱うためもう少し注意することがある。

定理 **11.1.** mergeSort(a,c) 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 のときに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 であるふたつのリストを併合するとき、最大で $n-1$ 回の比較が必要である。$C(n)$ を長さ 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の最大値とする。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。

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

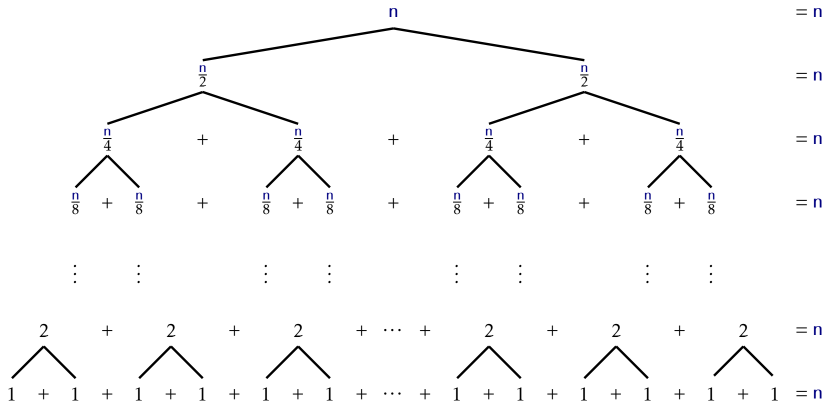


図 11.2: マージソートの再帰木

$$
\begin{aligned}
&= n - 1 + n\log n - n \\
&< n\log n
\end{aligned}
$$

n が奇数のときはもう少し複雑である。この場合ふたつの簡単に確認できる不等式を使う。任意の $x \geq 1$ について次の式が成り立つ。

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

また、任意の $x \geq 1/2$ について次の式が成り立つ。

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

不等式 (11.1) は $\log(x) + 1 = \log(2x)$ が成り立つことによる。不等式 (11.2) は log が凹関数（上に凸な関数）であるであることによる。これを利用して、奇数 n について次の式が成り立つ。

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

$$= n \log n \qquad \square$$

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。ふたつの部分問題を解いてから結果を併合するマージソートとは違い、クイックソートはもっと直接的に処理を行う。

クイックソートの説明は単純である。a からランダムに**軸** (pivot) となる要素 x を選ぶ。a を x より小さい要素、x と同じ要素、x より大きい要素に分割する。そして、一つめと三つめの分割を再帰的に整列する。図 11.3 に例を示した。

Algorithms

```
<T> void quickSort(T[] a, Comparator<T> c) {
  quickSort(a, 0, a.length, c);
}
<T> void quickSort(T[] a, int i, int n, Comparator<T> c) {
  if (n <= 1) return;
  T x = a[i + rand.nextInt(n)];
  int p = i-1, j = i, q = i+n;
  // a[i..p]<x,  a[p+1..q-1]??x, a[q..i+n-1]>x
  while (j < q) {
    int comp = compare(a[j], x);
    if (comp < 0) {        // move to beginning of array
      swap(a, j++, ++p);
    } else if (comp > 0) {
      swap(a, j, --q);  // move to end of array
    } else {
      j++;              // keep in the middle
    }
  }
  // a[i..p]<x,  a[p+1..q-1]=x, a[q..i+n-1]>x
  quickSort(a, i, p-i+1, c);
```

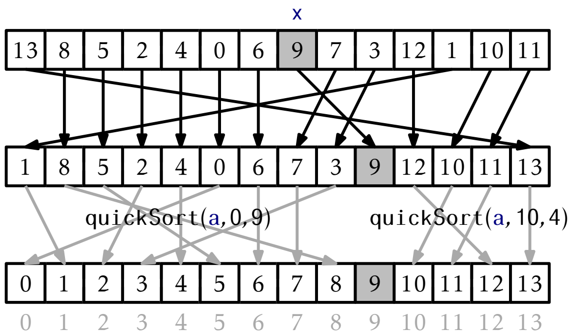


図 11.3: quickSort(a,0,14,c) の実行例

```
  quickSort(a, q, n-(q-i), c);
}
```

すべての処理は配列の部分的なコピーを取るのではなく in-place [*2] で実行される。quickSort(a,i,n,c) は $a[i],\ldots,a[i+n-1]$ を整列する。最初にこのメソッドを quickSort(a,0,a.length,c) と呼び出す。

クイックソートアルゴリズムの肝は、in-place で分割を行うことである。このアルゴリズムは余分な領域を使わず a の要素を入れ替え、次のような添え字 p と q を計算する。

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

この分割処理は while ループの中で、p を増やし、q を減らし、上の制約を保ちながら繰り返し実行される。各ステップで j 番目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。初めのふたつの場合は j を 1 増やし、最後の場合は j 番目の要素は未処理なので j は増やさない。

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 n 個の相異なる要素を入力すると、クイックソートの再帰木をランダム

[*2] 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。「前掲のマージソートは」a 全体をコピーするような別の配列を必要としたため、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。

二分探索木と見なせるのである。これを確認するためには、ランダム二分探索木を作るとき、まずはランダムに要素 x を選び、これを根にしたことを思い出そう。続いて、各要素を x と比較し、最終的に小さい要素を左の部分木に、大きい要素を右の部分木としたのだった。

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、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。続いて、前方の配列・後方の配列をそれぞれ再帰的に整列するこれに対応して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ことを再帰的に繰り返していた。

上のようなランダム二分探索木とクイックソートとの対応から、補題 7.1 をクイックソートの場合に置き換えて考えてみる。

補題 **11.1.** クイックソートは整数 $0,\ldots,\mathtt{n}-1$ を含む配列を整列するために呼ばれるとき、要素 i が軸と比較される回数の期待値は $H_{\mathtt{i}+1}+H_{\mathtt{n}-\mathtt{i}}$ 以下である。

調和数を計算すると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 $2\mathtt{n}\ln\mathtt{n}+O(\mathtt{n})$ 以下である。

証明*.* T を 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。

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

定理 11.3 は整列する要素が互いに異なる場合についてのものである。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、むしろよくなることさえある。重複する要素 x が軸に選ばれると、x はまとめられ、ふたつの部分問題のいずれにも含まれないためである。

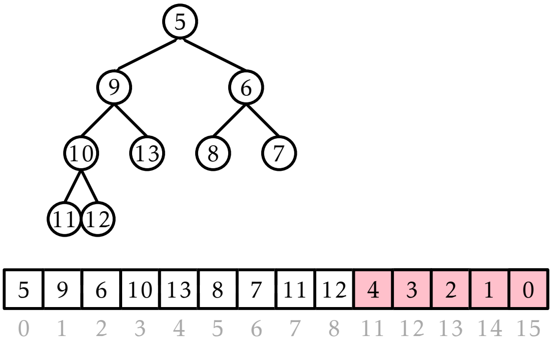


図 11.4: heapSort(a,c) の実行中のある瞬間の様子。色の付いている部分は既に整列済みの配列である。色の付いていない部分は BinaryHeap になっている。次の繰り返しでは、要素 5 が配列の位置 8 に移される。

定理 11.3. quickSort(a,c) 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 節で説明した二分ヒープを使う。BinaryHeap ではひとつの配列を使ってヒープを表現していたことを思い出そう。ヒープソートは入力の配列をヒープに変換した後で、最小値を取り出すことを繰り返す。

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。ヒープソートは次の操作を繰り返す。a を BinaryHeap に変換した後、a[0] と a[n−1] を入れ替え、n を 1 減らし、trickleDown(0) を呼んで a[0],...,a[n−2] を再度ヒープにする。n = 0 となってこの処理が終了すると、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。[*3] 図 11.4 は heapSort(a,c) の実行の様子を表している。

BinaryHeap
```
<T> void sort(T[] a, Comparator<T> c) {
```

[*3] compare(x,y) を修正すれば、結果が直接昇順に並ぶようにすることもできる

```
  BinaryHeap<T> h = new BinaryHeap<T>(a, c);
  while (h.n > 1) {
    h.swap(--h.n, 0);
    h.trickleDown(0);
  }
  Collections.reverse(Arrays.asList(a));
}
```

ヒープソートで肝となるのは、未整列の配列 a からヒープを構築する処理である。これを $O(n\log n)$ の時間で行うのは容易い。BinaryHeap の add(x) を繰り返し実行すればよい。しかし、ボトムアップなやり方でこのヒープの構築はもっと効率よく行える。BinaryHeap において、a[i] の子は a[2i+1] と a[2i+2] に入っていることを思い出そう。 そのため、$a[\lfloor n/2\rfloor],\ldots,a[n-1]$ は子を持っていない。別の言い方をすれば、$a[\lfloor n/2\rfloor],\ldots,a[n-1]$ は大きさ 1 の部分ヒープである。逆に考えると、$i\in\{\lfloor n/2\rfloor-1,\ldots,0\}$ に対して順番に trickleDown(i) を呼べる。trickleDown(i) を呼ぶとき a[i] の子はいずれも部分ヒープの根なのでこれは可能で、trickleDown(i) を呼ぶと a[i] は次の部分ヒープの根になる。

BinaryHeap
```
BinaryHeap(T[] a, Comparator<T> c) {
  this.c = c;
  this.a = a;
  n = a.length;
  for (int i = n/2-1; i >= 0; i--) {
    trickleDown(i);
  }
}
```

興味深いことに、ボトムアップなやり方は add(x) を n 回実行するよりも効率的である。それを理解するためには、葉の位置に存在する n/2 個の要素のためにはなにもする必要がなく、葉の親である n/4 個の要素のためには a[i] を根とする部分ヒープに対して trickleDown(i) を一度呼べばよく、n/8 個の要素のためにはこの高さの部分ヒープに対して trickleDown(i) を二度呼べ

ばよい… という処理の流れに注目すればよい。`trickleDown(i)` の実行時間は高々 `a[i]` を根とする部分ヒープの高さのオーダーなので、全実行時間の合計は高々次の値である。

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

最後から二番目の等号は、期待値の定義より、$\sum_{i=1}^{\infty} i/2^i$ とコインを投げる（表が出る回も含めた）回数とが等しいことと、補題 4.2 から成り立つ。

次の定理は `heapSort(a,c)` の性能を説明する。

定理 11.4. `heapSort(a,c)` の実行時間は $O(n\log n)$ であり、このメソッドは最大 $2n\log n + O(n)$ の比較を実行する。

証明. このアルゴリズムには 3 つのステップがある。(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 番目の呼び出しは大きさ $n-i$ のヒープに対するもので、最大 $2\log(n-i)$ 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。

$$\sum_{i=0}^{n-1} 2\log(n-i) \leq \sum_{i=0}^{n-1} 2\log n = 2n\log n$$

3 つのステップにおいて比較の実行回数を足し合わせると、証明が完成する。 □

11.1.4　比較ベースの整列における下界

3 つの比較に基づく整列アルゴリズムの実行時間がいずれも $O(n\log n)$ であることを見てきた。ここで気になるのはもっと速いアルゴリズムがあるのかどうかである。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くないが、そのためには想像力が必要だ。最終的にこれ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 11.11 とした。）

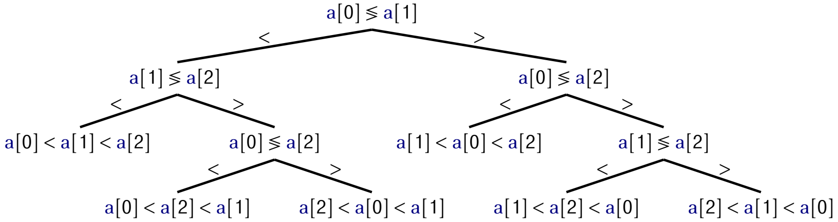


図 11.5: 長さ $n = 3$ の配列 $a[0], a[1], a[2]$ を整列するときの比較木

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せよ。下界を示すために注目すべきは、n を固定すると決定的なアルゴリズムが最初に比較する要素は常に決まったペアであることである。例えば heapSort(a, c) では n が偶数なら最初に trickleDown(i) を呼ぶとき $i = n/2 - 1$ であり、最初に行われるのは $a[n/2-1]$ と $a[n-1]$ との比較である。

入力は相異なるので最初の比較の結果は二通りである。二番目の比較は最初の比較の結果に依存して行われるかもしれない。三度目の比較は最初のふたつの比較の結果に依存するかもしれず、以降も同様である。こうして、任意の決定的な比較に基づく整列アルゴリズムを根付き二分**比較木** (comparison tree) とみなせる。この木の各内部ノード u は添え字のペア $u.i$ と $u.j$ でラベル付けられている。$a[u.i] < a[u.j]$ ならアルゴリズムは左の部分木に進み、そうでないなら右の部分木に進む。この木の各葉 w は $0, \ldots, n-1$ のある置換 $w.p[0], \ldots, w.p[n-1]$ でラベル付けられている。この置換は比較木がこの葉に到達するとき a を整列するのに必要な比較を表現している。これは次のものである。

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

大きさ $n = 3$ である配列の比較木の例を図 11.5 に示す。

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任意の入力配列 a について必要な比較の列や、アルゴリズムが a を整列するためにどういう順で並べ替えを行うかがわかるのである。そのため比較木は少なくとも $n!$ 個の葉を持つ。もしそうでなければ、相異なる置換であって同じ葉に到達するものが存在してしまう。するとそのアルゴリズムは、同じ葉に到達する置換のうち少なくともどれか 1 つを正しく整列できないことになる。

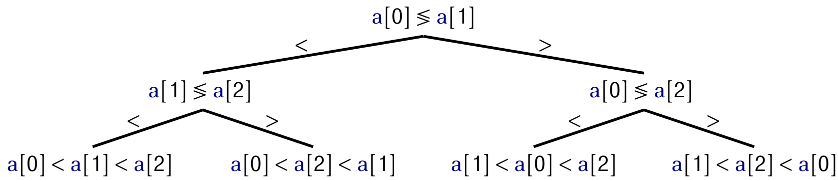


図 11.6: 正しく整列できない入力が存在する比較木

例えば図 11.6 に示した比較木は $4(< 3! = 6)$ 個の葉を持つ。この木を見ると、ふたつの入力 $3,1,2$ と $3,2,1$ はいずれも右端の葉に到達することがわかる。入力 $3,1,2$ は正しく $\mathtt{a}[1] = 1, \mathtt{a}[2] = 2, \mathtt{a}[0] = 3$ を出力する。しかし入力 $3,2,1$ の出力は $\mathtt{a}[1] = 2, \mathtt{a}[2] = 1, \mathtt{a}[0] = 3$ となり正しくない。この議論から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 $\mathtt{n} \geq 1$ について、ある長さ $\mathtt{n}$ の入力配列 $\mathtt{a}$ が存在し、$\mathcal{A}$ が $\mathtt{a}$ を整列するとき $\log(\mathtt{n}!) = \mathtt{n}\log\mathtt{n} - O(\mathtt{n})$ 回の比較を実行する。

証明**.** これまでの議論から $\mathcal{A}$ の比較木は少なくとも $\mathtt{n}!$ 個の葉を持つ必要がある。帰納法により簡単に k 個の葉を持つ二分木の高さが $\log k$ 以上であることを示せる。よって $\mathcal{A}$ の比較木には深さ $\log(\mathtt{n}!)$ 以上の葉 $\mathtt{w}$ が存在し、ある入力配列 $\mathtt{a}$ が存在し、この葉に到達する。この $\mathtt{a}$ に対して $\mathcal{A}$ は $\log(\mathtt{n}!)$ 回以上の比較を実行する。 □

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り、クイックソートのようなランダムなアルゴリズムに関してはなにもわからない。ランダム性を利用するアルゴリズムは比較回数の下界 $\log(\mathtt{n}!)$ を打ち破れるのだろうか。これもやはりできないのである。これを示すには、ランダムなアルゴリズムについて違った視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「整理されてある」と仮定する。 どのような入力配列 $\mathtt{a}$ によっても到達できないノードは削除される。このとき木にはちょうど $\mathtt{n}!$ 個だけの葉が含まれる。葉の数が $\mathtt{n}!$ 以上なのは、そうでないと整列を正しく行えないからである。一方、葉の数が $\mathtt{n}!$ 以下なのは、$\mathtt{n}$ 個の相異なる要素の置換は $\mathtt{n}!$ 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一つ辿るためである。

ランダムなソートアルゴリズム $\mathcal{R}$ は、ふたつの入力を取る決定的なアルゴ

リズムだと考えられる。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。このランダムな数によってアルゴリズムはランダム化される。アルゴリズムがコインを投げてランダムな選択をしたくなったとき、b の要素を使ってこれを行う。例えばクイックソートにおける最初の軸を選ぶとき、アルゴリズムは式 $\lfloor nb_1 \rfloor$ を使う。

b を特定の列 $\hat{b}$ に替えると、$\mathcal{R}$ 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。また、a を $\{1, \ldots, \mathrm{n}\}$ の置換からランダムに選ぶのは、$\mathcal{T}(\hat{b})$ の $\mathrm{n}!$ 個の葉のうちのひとつをランダムに選ぶのは同じである。

問 11.13 で示す必要があったのは、k 個の葉を持つ二分木の葉をランダムに選ぶとき、葉の深さの期待値が $\log k$ 以上であることであった。以上より、$\{1, \ldots, n\}$ の置換からランダムに選んだものを入力するとき、(決定的な) アルゴリズム $\mathcal{R}(\hat{b})$ が実行する比較の回数の期待値は $\log(\mathrm{n}!)$ 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。ランダムなアルゴリズムについての下界がこうして示された。

定理 **11.6.** 任意の整数 $n \geq 1$ と、任意の (決定的でもランダムでもよい) 比較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 $\log(\mathrm{n}!) = \mathrm{n} \log \mathrm{n} - O(\mathrm{n})$ 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。小さい整数を整列することに特化し、要素 (の一部) を配列の添え字として使うことで、この節で紹介するアルゴリズムは定理 11.5 の下界に縛られない。次の文を考えよう。

$$\mathtt{c}[\mathtt{a}[\mathtt{i}]] = 1$$

この文は定数時間で実行できるが、`a[i]` は `c.length` 種類の値を取りうる。これはこのような文を使うアルゴリズムは二分木でモデル化できないということである。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す理由なのである。

11.2.1 計数ソート

$0,\dots,k-1$ 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) はカウンタの補助配列 c を使って a を整列する。そして、整列された a を補助配列 b として返す。

計数ソートのアイデアは単純だ。任意の $i \in \{0,\dots,k-1\}$ について、i の出て来る回数を数え、これを c[i] に入れておく。整列の後では、出力は c[0] 個の 0 からはじまり、c[1] 個の 1 が続き、c[2] 個の 2 が続き、…、c[k−1] 個の k−1 で終わる。このコードは巧妙である。実行の様子を図 11.7 に示す。

```
Algorithms
int[] countingSort(int[] a, int k) {
  int c[] = new int[k];
  for (int i = 0; i < a.length; i++)
    c[a[i]]++;
  for (int i = 1; i < k; i++)
    c[i] += c[i-1];
  int b[] = new int[a.length];
  for (int i = a.length-1; i >= 0; i--)
    b[--c[a[i]]] = a[i];
  return b;
}
```

このコードと最初の for ループでは各カウンタ c[i] が a において i が何回現れるかを数えている。a の値を添え字として使うことで、この処理を合計 $O(n)$ のひとつの for ループで終えている。続いて、c を使って直接出力配列 b を埋めていける。しかし、a にデータが関連づけられているときはこれはできない。よって、a から b に要素をコピーするための追加の仕事が必要になる。

次の for ループでは、$O(k)$ の時間をかけてカウンタを順に足しこんでいる。こうすると c[i] は a における i 以下の要素の個数になる。特に、任意の $i \in \{0,\dots,k-1\}$ について、出力配列 b は次の式を満たす。

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

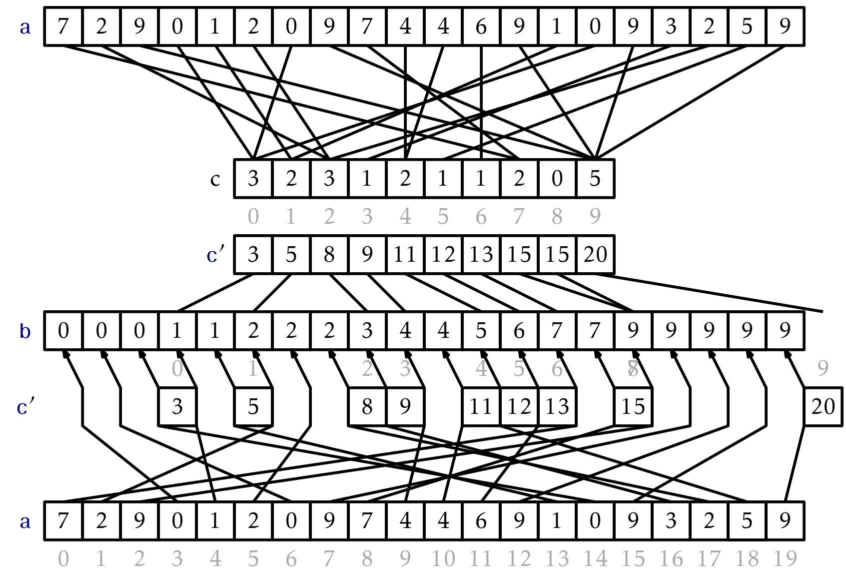


図 11.7: $0,\dots,k-1=9$ のいずれかを格納する、長さ $n=20$ である配列に対する、計数ソートの操作

最後に a を逆に辿って、要素を b に入れる。a を辿りながら、要素 $a[i]=j$ は $b[c[j]-1]$ に入れられ、$c[j]$ をひとつ減らす。

定理 11.7. countingSort(a,k) は $\{0,\dots,k-1\}$ の要素 n 個からなる配列 a を $O(n+k)$ の時間で整列する。

計数ソートは**安定性** (stable) という良い性質を持つ。これは元の配列における等しい要素同士の相対的な位置を、整列後の配列でも保つ性質である。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 $a[j]$ の前に来るのである。この性質は次の節で役立つ。

11.2.2 基数ソート

計数ソートは配列の長さ n が、配列内の最大値 $k-1$ と比べてそれほど小さくないなら、非常に効率的である。今から説明する**基数ソート** (radix sort) は複数回の計数ソートにより、最大値が比較的大きな場合でも効率的なアルゴリズムである。

基数ソートは w ビットの整数を w/d 回の計数ソートにより、一度に d ビットずつ整列する。[*4] より正確に言うと、基数ソート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。このような処理を繰り返し、最後には整数の最高位 d ビットだけを見て整列する。

Algorithms
```
int[] radixSort(int[] a) {
  int[] b = null;
  for (int p = 0; p < w/d; p++) {
    int c[] = new int[1<<d];
    // the next three for loops implement counting-sort
    b = new int[a.length];
    for (int i = 0; i < a.length; i++)
      c[(a[i] >> d*p)&((1<<d)-1)]++;
    for (int i = 1; i < 1<<d; i++)
      c[i] += c[i-1];
    for (int i = a.length-1; i >= 0; i--)
      b[--c[(a[i] >> d*p)&((1<<d)-1)]] = a[i];
    a = b;
  }
  return b;
}
```

（このコードでは $(\mathtt{a}[\mathtt{i}]{>>}\mathtt{d}*\mathtt{p})\&((1{<<}\mathtt{d})-1)$ と書いて、$\mathtt{a}[\mathtt{i}]$ の 2 進数表記において $(\mathtt{p}+1)\mathtt{d}-1,\ldots,\mathtt{pd}$ ビット目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 に示した。

このアルゴリズムが正しく整列を行えるのは、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 のふたつの要素 x と y が $\mathtt{x}<\mathtt{y}$ を満たし、x と y が添え字 r の位置で異なるなら、$\lfloor r/\mathtt{d}\rfloor$ 回目の整列で 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。

基数ソートは w/d 回の計数ソートを行う。各計数ソートの実行時間は $O(\mathtt{n}+2^{\mathtt{d}})$ 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。

[*4] 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 $\mathtt{d}\lceil \mathtt{w}/\mathtt{d}\rceil$ に増やす必要がある。

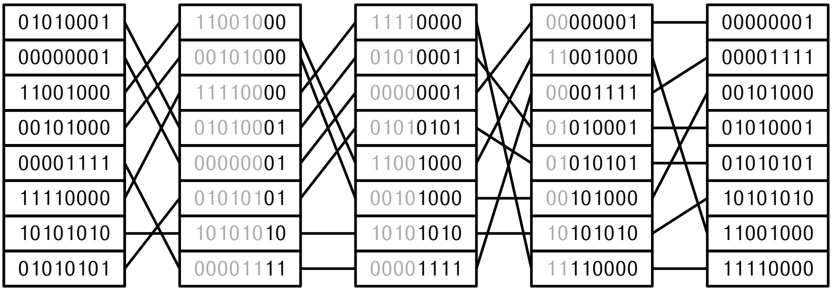


図 11.8: 基数ソートにより $\mathtt{w} = 8$ ビットの整数を整列する。$\mathtt{d} = 2$ ビットの整数を計数ソートによって整列する処理を 4 回行う。

定理 11.8. 任意の整数 $\mathtt{d} > 0$ について、radixSort(a,k) は $\mathtt{n}$ 個の $\mathtt{w}$ ビット整数を含む配列 a を $O((\mathtt{w}/\mathtt{d})(\mathtt{n} + 2^{\mathtt{d}}))$ の時間で整列する。

配列の要素は $\{0,\ldots,\mathtt{n}^c - 1\}$ の範囲の整数であり、$\mathtt{d} = \lceil \log \mathtt{n} \rceil$ 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、$\{0,\ldots,\mathtt{n}^c - 1\}$ の範囲の $\mathtt{n}$ 個の整数からなる配列 a を、$O(c\mathtt{n})$ 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 [48] によればマージソートは von Neumann (1945) が考案したものだという。クイックソートは Hoare [39] が考案した。はじめにヒープソートを考案したのは Williams [78] だが、本節で説明した $O(\mathtt{n})$ の時間でボトムアップにヒープを構築する方法は Floyd [28] によるものである。比較に基づくソートの下界は古くから言い継がれてきたものである。次の表は比較に基づくアルゴリズムの性能をまとめたものだ。

	時間計算量		in-place
マージソート	$\mathtt{n}\log\mathtt{n}$	最悪実行時間	No
クイックソート	$1.38\mathtt{n}\log\mathtt{n} + O(\mathtt{n})$	期待実行時間	Yes
ヒープソート	$2\mathtt{n}\log\mathtt{n} + O(\mathtt{n})$	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の長所・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。しかしマージのときに補助配列が必要だ。この配列を確保するのはコストが高く、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の中だけで処理を済ませ、比較の回数も二番目に少ないか、ランダムな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。これは連結リストを整列するときである。この場合は補助的な配列が必要ないのである。ふた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、整列済みの一つの配列が得られる。(問 11.2 を参照せよ。)

この章で説明した計数ソートと基数ソートは Seward [68, Section 2.4.6] によるものである。しかし、基数ソートの一種が 1920 年代からパンチカードを整列するために機械によって使われていた。この機械はカードの山を、ある場所に穴が空いているかどうかを判定してふたつの山に分けた。色々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・基数ソートは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 $\{a,\ldots,b\}$ の範囲の整数を整列できるよう素直に修正すれば、実行時間は $O(\mathtt{n}+b-a)$ になる。同様に基数ソートは同じ範囲の整数を $O(\mathtt{n}(\log_{\mathtt{n}}(b-a))$ の時間で整列できる。また、いずれのアルゴリズムも、IEEE754 形式の浮動小数点数を整列するのにも使える。これは IEEE 754 の形式は数の大きさに符号が付いた二進整数表現だと見なして比較が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の実行の様子を描け。また同じ配列について、クイックソートを実行の様子として、ありえるもののうちのひとつを描け。

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 を参照せよ。)

問 11.3. `quickSort(a,i,n,c)` の実装には、軸として常に `a[i]` を選ぶものがある。この実装が、$\binom{\mathtt{n}}{2}$ 回の比較を実行する長さ $\mathtt{n}$ の入力の例を示せ。

問 11.4. `quickSort(a,i,n,c)` の実装には、軸として常に `a[i+n/2]` を選ぶものがある。この実装が、$\binom{\mathtt{n}}{2}$ 回の比較を実行する長さ $\mathtt{n}$ の入力の例を示せ。

問 11.5. どのような `quickSort(a,i,n,c)` の実装でも、軸を決定的に選び、

$\mathtt{a}[\mathtt{i}],\ldots,\mathtt{a}[\mathtt{i}+\mathtt{n}-1]$ を先に見ないならば、長さ $\mathtt{n}$ のある入力が存在し、$\binom{\mathtt{n}}{2}$ 回の比較を実行することを示せ。

問 **11.6.** quickSort(a,i,n,c) を実行するとき $\binom{\mathtt{n}}{2}$ 回の比較を実行する Comparator c 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.** 定理 11.3 の証明よりも細かくクイックソートが実行する比較の回数の期待値を解析せよ。具体的には、比較の回数の期待値が $2\mathtt{n}H_{\mathtt{n}}-\mathtt{n}+H_{\mathtt{n}}$ であることを示せ。

問 **11.8.** ヒープソートを実行するときの、比較の回数が $2\mathtt{n}\log\mathtt{n}-O(\mathtt{n})$ 回になる入力配列を与え、そのことを説明せよ。

問 **11.9.** この章で説明したヒープソートの実装では、要素を逆順に並び替え、その後で順番を逆にしていた。入力の Comparator c の結果を否定して新しい Comparator を定義すれば、最後のステップを省ける。これがよい最適化でない理由を説明せよ。（ヒント：配列を逆さまにするのと比べて、否定のためにどのくらい時間が必要になるかを考えよ。）

問 **11.10.** 図 11.6 の比較木によって正しく整列できない 1,2,3 の別の置換を見つけよ。

問 **11.11.** $\log\mathtt{n}!=\mathtt{n}\log\mathtt{n}-O(\mathtt{n})$ を示せ。

問 **11.12.** 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 **11.13.** 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 **11.14.** 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 章

グラフ

この章ではグラフのふた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。

数学的には、**（有向）グラフ** (directed graph) とは組み $G = (V, E)$ である。ここで V は**頂点** (vertex) の集合であり、E は**辺** (edge) と呼ばれる頂点の組みの集合である。辺 $(\mathtt{i}, \mathtt{j})$ は $\mathtt{i}$ から $\mathtt{j}$ に向いている。$\mathtt{i}$ は辺の**始点** (source) と呼ばれ、$\mathtt{j}$ は**終点**と呼ばれる。 G における**経路** (path) とは頂点の列 $v_0, \ldots, v_k$ であって 任意の $i \in \{1, \ldots, k\}$ について辺 (v_{i-1}, v_i) が E に含まれるものである。経路 $v_0, \ldots, v_k$ が**循環** (cycle) である（循環している）とは、(v_k, v_0) も E の要素であることをいう。経路（または循環）が**単純** (simple) であるとは、経路に含まれる頂点が互いに異なることをいう。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 にグラフの例を示した。

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）通信路を辺と見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見なせばグラフとしてモデル化できる。

より意外な例は、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したものだ。例えば大学の時間割における**衝突グラフ** (conflict graph) を考えられる。ここでは頂点は大学の講義で、辺 $(\mathtt{i}, \mathtt{j})$ は $\mathtt{i}$ と $\mathtt{j}$ の両方を受講する生徒がいることを表している。よってこの辺があるとき、講義 $\mathtt{i}$ と講義 $\mathtt{j}$ のテストを同じ時間に行うことはできないとわかる。

この節を通じて $\mathtt{n}$ は頂点の数を、$\mathtt{m}$ は辺の数を表すことにする。すなわち $\mathtt{n} = |V|$ かつ $\mathtt{m} = |E|$ である。さらに $V = \{0, \ldots, \mathtt{n}-1\}$ と仮定する。V の各頂

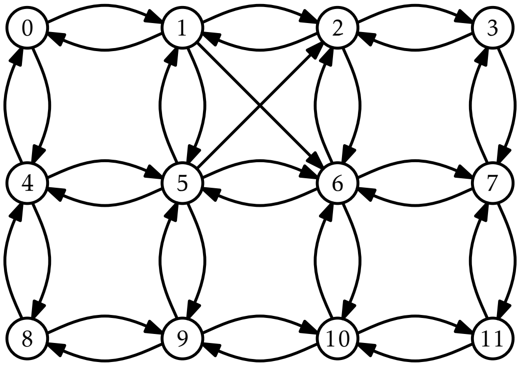


図 12.1: 12 個の頂点からなるグラフ。頂点は番号付きの円で、辺は source から target に向かう矢印で描く。

点とひも付けられたデータを保存するためには、大きさ n の配列にデータを入れておけば良い。

グラフに対する典型的な操作は次のものだ。

- 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)：$(\mathtt{i}, \mathtt{j}) \in E$ かどうかを調べる。
- outEdges(i)：$(\mathtt{i}, \mathtt{j}) \in E$ を満たす整数整数 j のリストを返す。
- inEdges(i)：$(\mathtt{j}, \mathtt{i}) \in E$ 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、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。最後のふたつの操作は各頂点毎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、グラフの応用によって各操作への要求が異なり、そして理想的にはこれらの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

隣接行列 (adjacency matrix) は n 個の頂点を持つグラフ $G=(V,E)$ を、真偽値が並んだ n×n 行列 a を使って表現したものである。

AdjacencyMatrix
```
int n;
boolean[][] a;
AdjacencyMatrix(int n0) {
  n = n0;
  a = new boolean[n][n];
}
```

行列のエントリ a[i][j] は次のように定義される。

$$\mathtt{a[i][j]} = \begin{cases} \mathtt{true} & (\mathtt{i},\mathtt{j}) \in E \text{ のとき} \\ \mathtt{false} & \text{そうでないとき} \end{cases}$$

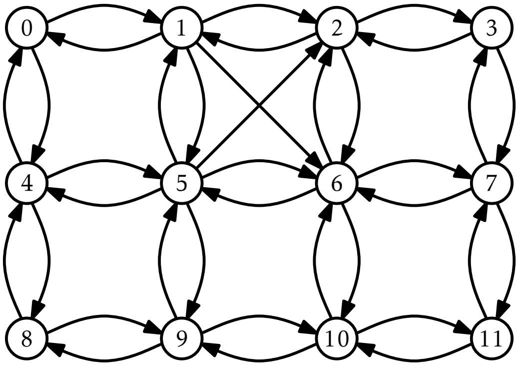
図 12.1 のグラフの隣接行列を図 12.2 に示した。

この表現における addEdge(i, j)・removeEdge(i, j)・hasEdge(i, j) はいずれもエントリ a[i][j] を読み書きすればよい。

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

これらの操作は明らかに定数時間で実行できる。

隣接行列で効率がよくないのは outEdges(i) と inEdges(i) である。これ



	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

を実装するためには、a における対応する行または列の n 個のエントリを順に見て、各添え字 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。

AdjacencyMatrix
```
List<Integer> outEdges(int i) {
```

```
  List<Integer> edges = new ArrayList<Integer>();
  for (int j = 0; j < n; j++)
    if (a[i][j]) edges.add(j);
  return edges;
}
List<Integer> inEdges(int i) {
  List<Integer> edges = new ArrayList<Integer>();
  for (int j = 0; j < n; j++)
    if (a[j][i]) edges.add(j);
  return edges;
}
```

これらの操作は明らかに $O(n)$ の時間がかかる。

隣接行列による表現のもうひとつの短所は行列が大きいことである。$n \times n$ の真偽値の行列を格納するには n^2 ビット以上のメモリが必要である。真偽値を単純にならべた行列は n^2 バイトのメモリを使う。実装を工夫して、w 個の真偽値をワードに詰め込むと、領域使用量を $O(n^2/w)$ ワードに減らせる。

定理 12.1. *AdjacencyMatrix* は *Graph* インターフェースを実装する。*AdjacencyMatrix* は以下のように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) を時間 $O(n)$ で実行できる。

AdjacencyMatrix の領域使用量は $O(n^2)$ である。

メモリ使用量の多さと inEdges(i)・outEdges(i) の性能の低さにもかかわらず、AdjacencyMatrix が有効な場合もある。具体的にはグラフ G が**密**なとき、つまり辺の数が n^2 に近く、メモリ使用量 n^2 が許容できる場合である。

AdjacencyMatrix が広く使われるのは、グラフ G の性質を計算するための行列 a の代数的な操作を効率的に実行できるからでもある。これはアルゴリズムの授業のトピックだが、ひとつだけそのような性質を挙げる。a のエントリを整数（true が 1、false が 0）であると見なして、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。

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

グラフ G の文脈で解釈すると、この和は G が辺 $(\mathtt{i},\mathtt{k})$ と辺 $(\mathtt{k},\mathtt{j})$ を共に持つ頂点 $\mathtt{k}$ の個数を数えている。つまりこの和は $\mathtt{i}$ から $\mathtt{j}$ への（中間頂点 $\mathtt{k}$ を通る）経路であって、長さがちょうど 2 であるものの個数である。この観察は、G におけるすべての頂点の対についての最短経路を $O(\log \mathtt{n})$ 回だけの行列の積で計算するアルゴリズムの基礎になっている。

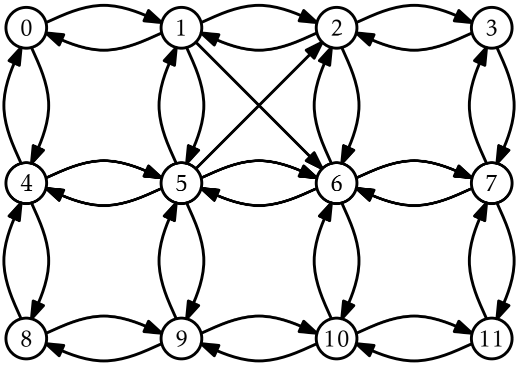
12.2 AdjacencyLists：リストの集まりとしてのグラフ

グラフの**隣接リスト** (adjacency list) 表現は辺を重視するアプローチである。隣接リストの実装は色々な方法がある。この節ではその中でも単純なものを説明する。また、この節の最後に別のやり方も紹介する。隣接リスト表現ではグラフ $G=(V,E)$ はリストの配列 adj で表現される。リスト adj[i] は頂点 i と隣接するすべての頂点を含む。つまり、adj[i] は $(\mathtt{i},\mathtt{j}) \in E$ を満たすすべての添え字 j からなるリストである。

AdjacencyLists
```
int n;
List<Integer>[] adj;
AdjacencyLists(int n0) {
  n = n0;
  adj = (List<Integer>[])new List[n];
  for (int i = 0; i < n; i++)
    adj[i] = new ArrayStack<Integer>();
}
```

（例を図 12.3 に示す）ここでの実装ではリスト adj は ArrayStack とする。なぜなら添え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。



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

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

これは定数時間で実行できる。

`removeEdge(i,j)` はリスト `adj[i]` から `j` を見つけ、それを削除する。

AdjacencyLists
```
void removeEdge(int i, int j) {
  Iterator<Integer> it = adj[i].iterator();
  while (it.hasNext()) {
    if (it.next() == j) {
```

```
      it.remove();
      return;
    }
  }
}
```

これは $O(\deg(\mathtt{i}))$ の時間がかかる。ここで $\deg(\mathtt{i})$（i の**次数** (degree)）は E の要素のうち i から出ている辺の個数である。

hasEdge(i, j) も同様だ。リスト adj[i] から j を探して、見つかれば真を、そうでないなら偽を返す。

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

これにかかる時間は $O(\deg(\mathtt{i}))$ である。

outEdges(i) は単純である。これはリスト adj[i] を返す。

AdjacencyLists
```
List<Integer> outEdges(int i) {
  return adj[i];
}
```

これは定数時間で実行できる。

inEdges(i) 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、もしそうなら j を出力リストに追加する。

AdjacencyLists
```
List<Integer> inEdges(int i) {
  List<Integer> edges = new ArrayStack<Integer>();
  for (int j = 0; j < n; j++)
    if (adj[j].contains(i))  edges.add(j);
  return edges;
```

```
}
```

この操作にはかなり時間がかかる。すべての頂点の隣接リストを見て回る必要があるので、$O(\mathtt{n}+\mathtt{m})$ の時間がかかる。

次の定理は上で説明したデータ構造の性能をまとめたものである。

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)・hasEdge(i, j) にかかる時間は $O(\deg(\mathtt{i}))$ である。
- outEdges(i) は定数時間で実行できる。
- inEdges(i) にかかる時間は $O(\mathtt{n}+\mathtt{m})$ である。

AdjacencyLists の領域使用量は $O(\mathtt{n}+\mathtt{m})$ である。

先程少し言ったように、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。実装方法の選び方の例を挙げる。

- adj の要素を格納するにはどんなデータ構造を使うのがいいだろう。配列ベースのもの、ポインタベースのもの、あるいはハッシュテーブルだろうか。
- 任意の i について $(\mathtt{j},\mathtt{i})\in E$ を満たす j のリストである二次隣接リスト inadj があるべきだろうか。これは inEdges(i) の実行時間を劇的に改善するが、辺を追加・削除する際の仕事を少し増やす。
- adj[i] における辺 (i, j) は対応する inadj[j] のエントリへの参照を持つべきだろうか。
- 辺は明示的にオブジェクトとして実装し、関連データを持たせるべきだろうか。このとき adj は頂点のリストではなく、辺のリストを持つことになる。

これらの選択肢の大部分は、実装の複雑さと性能とのトレードオフをふまえて考えることになる。

12.3 グラフの走査

この節ではグラフの頂点 i からはじめて、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ふたつ紹介する。いずれのアルゴリズムでも、隣接リストで表現されたグラフを使うのが適切である。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) を頂点 i からはじめると、まずは i に隣接する頂点を訪問し、続いて i の隣の隣、続いて i の隣の隣の隣、というように進んでいく。

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、非常に似ている。キュー q を使うが、これは初期状態では i だけを含む。q から要素を取り出し、取り出した要素に隣接する要素を q に追加する。ここで、追加する要素はまだこれまで q に追加していないものだけである。木とグラフにおける、幅優先探索アルゴリズムの大きな違いは、グラフの場合は同じ頂点を q に二度以上追加しないよう気をつけなければならないことである。このためには真偽値の補助配列 seen を使って、どの頂点が既に見つかっているかを記録しておけばよい。

Algorithms
```
void bfs(Graph g, int r) {
  boolean[] seen = new boolean[g.nVertices()];
  Queue<Integer> q = new SLList<Integer>();
  q.add(r);
  seen[r] = true;
  while (!q.isEmpty()) {
    int i = q.remove();
    for (Integer j : g.outEdges(i)) {
      if (!seen[j]) {
        q.add(j);
        seen[j] = true;
      }
```

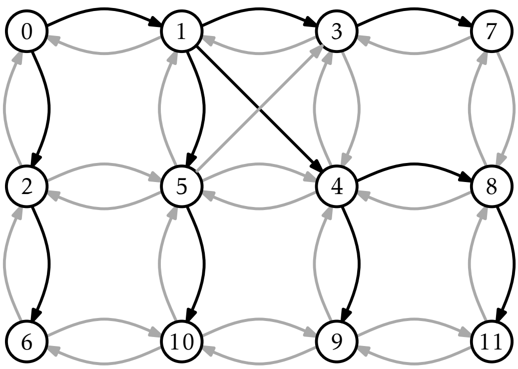


図 12.4: ノード 0 から始まる幅優先探索の例。ノードの数字はこの探索において q に追加される順番を表している。ノードが q に追加されるときに辿られる辺を黒、それ以外の辺を灰色に描いている。

```
        }
    }
}
```

図 12.1 において bfs(g,0) を実行する様子の一例を図 12.4 に示した。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 の処理は図 12.3 の隣接リストを使った。

bfs(g,i) の実行時間の解析は簡単である。seen のおかげで、同じ頂点が q に二度以上追加されることはない。q に頂点を追加する（そして後で削除する）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一度処理されるので、すべての隣接リストが高々一度処理される。よって G の辺は高々一度だけ処理される。内部ループが一周すると辺がひとつ処理され、この各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。

次の定理は bfs(g,r) の性能をまとめたものである。

定理 12.3. *AdjacencyLists* で実装された *Graph* g を入力すると、bfs(g,r) の実行時間は $O(n+m)$ 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を q に追加する（頂点はいずれ取り出される）。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、これが繰り返される。そのため、bfs(g, r) は r からの距離の昇順で頂点を訪問し、r から到達不可能な頂点を訪問することはない。

そのため、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) の変種を使える。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] は r から j への最短経路における、最後から二番目の頂点になる。p[p[j]], p[p[p[j]]]... とこれを繰り返すと、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

深さ優先探索 (depth-first search) は二分木における標準的な走査アルゴリズムに似ている。このアルゴリズムではある部分木を完全に探索し終えてから根の方向に戻り、そして別の部分木の探索に進む。別の考え方をすると、深さ優先探索は幅優先探索に似ていて、その違いはスタックの代わりにキューを使うことである。

深さ優先探索において各頂点 i には色 c[i] を割り当てる。未訪問の頂点を white、現在訪問中の頂点を grey、既に訪問した頂点を black にする。深さ優先探索は再帰的なアルゴリズムとして考えるのが簡単である。r を訪問するところから処理がはじまる。頂点 i を訪問するとき、i の色を grey にする。続いて i の隣接リストを見て、その中の white である頂点を再帰的に訪問する。最後に i の色を black にして、i の処理を終える。

Algorithms
```
void dfs(Graph g, int r) {
  byte[] c = new byte[g.nVertices()];
  dfs(g, r, c);
}
void dfs(Graph g, int i, byte[] c) {
  c[i] = grey;  // currently visiting i
  for (Integer j : g.outEdges(i)) {
    if (c[j] == white) {
```

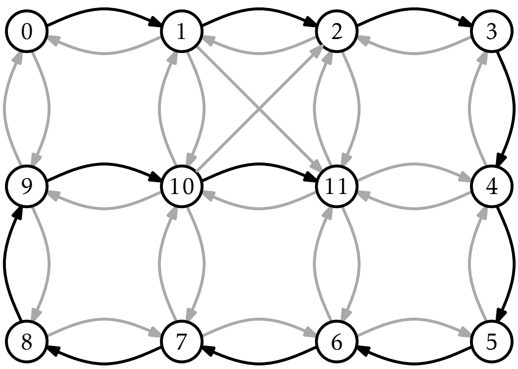


図 12.5: ノード 0 から始まる深さ優先探索の例。ノードにはこの探索において q に追加される順番が付されている。再帰的呼び出しになる辺を黒、それ以外の辺を灰色に描いている。

```
      c[j] = grey;
      dfs(g, j, c);
    }
  }
  c[i] = black; // done visiting i
}
```

図 12.5 にこのアルゴリズムの処理の例を示す。

深さ優先探索のことを考えるのには再帰は便利なのだが、実装する際にはこれは最善の方法ではない。上のコードは、スタックのオーバーフローによって大きなグラフの探索に失敗してしまうことがある。 別の実装方法として、再帰を明示的なスタック s に置き換えることが考えられる。次の実装はこれを行ったものである。

```
Algorithms
void dfs2(Graph g, int r) {
  byte[] c = new byte[g.nVertices()];
  Stack<Integer> s = new Stack<Integer>();
  s.push(r);
```

```
    while (!s.isEmpty()) {
      int i = s.pop();
      if (c[i] == white) {
        c[i] = grey;
        for (int j : g.outEdges(i))
          s.push(j);
      }
    }
}
```

上のコードでは、次の頂点 i が処理されるとき、i の色を grey にし、i の隣接行列に入っていた頂点をスタックに積み、次はそのうちの一つを i にする。

当然だが dfs(g,r)・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)・dfs2(g,r) の実行時間はいずれも $O(\mathtt{n}+\mathtt{m})$ である。

幅優先探索と同様に、深さ優先探索の各実行にもある木を対応づけられる。頂点 $\mathtt{i} \neq \mathtt{r}$ 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) の場合は i は i′ をスタックで置き換えた頂点のうちの一つである。）i′ を i の親だと考えると、r を根とする木が得られる。図 12.5 では、この木は頂点 0 から頂点 11 への経路である。

深さ優先探索の重要な性質を述べる。i の色が grey であるとき、i から他の頂点 j への白い頂点だけを辿る経路が存在するとする。そして、i の色が black になるよりも前に、j の色は grey、そして black になる。（これは背理法で証明できる。i から j へのある経路 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 を参照せよ。r から到達可能なある循環 C があるとする。i を C の中で色が grey である最初の頂点とし、j を C において i の前にある頂点とする。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。深さ優先探索において i から j への経路 P が存在し、一方辺 (j,i) も存在するので、P も循環であることがわかる。

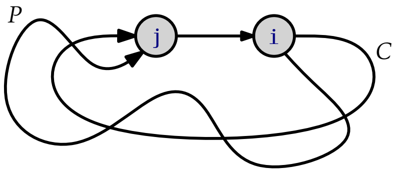


図 12.6: 深さ優先探索アルゴリズムにより、グラフ G の循環を検出できる。ノード $\mathtt{i}$ が灰色であるとき、ノード $\mathtt{j}$ も灰色である。そのため、深さ優先探索木において $\mathtt{i}$ から $\mathtt{j}$ への経路 P が存在する。また、辺 $(\mathtt{j},\mathtt{i})$ が存在するとき、P は循環である。

12.4 ディスカッションと練習問題

12.3 と 12.4 で示されている幅優先探索・深さ優先探索の実行時間はあまりタイトではない。$\mathtt{n_r}$ を G における頂点 $\mathtt{i}$ であって、$\mathtt{i}$ から $\mathtt{r}$ への経路が存在するものの個数と定義する。$\mathtt{m_r}$ をこのような頂点から出る辺の個数とする。このとき、幅優先探索・深さ優先探索のより正確な実行時間を示す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、この定理は役に立つ。）

定理 12.5. *AdjacencyLists* で実装された *Graph* $\mathtt{g}$ を入力すると、$\mathtt{bfs}(\mathtt{g},\mathtt{r})$・$\mathtt{dfs}(\mathtt{g},\mathtt{r})$・$\mathtt{dfs2}(\mathtt{g},\mathtt{r})$ の実行時間はいずれも $O(\mathtt{n_r}+\mathtt{m_r})$ である。

幅優先探索は Moore [52] と Lee [49] によって独立に考案されたようである。それぞれは迷路の探索と回路における経路に関する文脈で発見された。

グラフの隣接リスト表現は（より一般的であった）隣接行列表現の代替として Hopcroft と Tarjan [40] が提案した。隣接リスト表現は、深さ優先探索の他にも、Hopcroft-Tarjan の平面性テストのアルゴリズムにおいて重要な役割を果たす。これはグラフを辺が交差しないように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 $(\mathtt{i},\mathtt{j})$ が存在するならば、またそのときに限って辺 $(\mathtt{j},\mathtt{i})$ 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表現および隣接行列表現を書け。

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

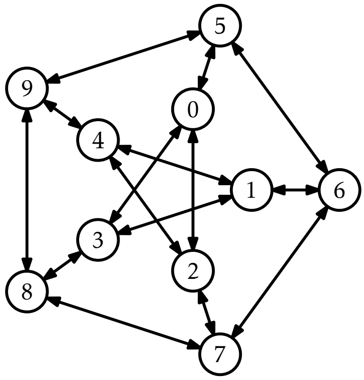


図 12.7: 例題用のグラフ

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. グラフの隣接行列表現を設計・実装せよ。また、領域使用量を解析し、addEdge(i, j)・removeEdge(i, j)・hasEdge(i, j)・inEdges(i)・outEdges(i) の実行時間を求めよ。

問 12.3. 図 12.7 のグラフ G について、bfs(G,0)・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み i, j について、i から 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、j から 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** (connected components) ラベル付けとは、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であるものである。これを $O(\mathtt{n}+\mathtt{m})$ の時間で計算する方法を示せ。

問 12.6. 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、各木の辺は G の辺であり、G のすべての頂点をある木に含むものである。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 の任意の頂点の組み $\mathtt{i}$, $\mathtt{j}$ 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：$\mathtt{r}$ からふた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、$\mathtt{n}-1$ 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 $O(\mathtt{n})$ でなければならない。

[*1] universal sink $\mathtt{v}$ を celebrity と言うこともある。部屋の中のみなが $\mathtt{v}$ のことを知っているが、$\mathtt{v}$ は部屋の中の他の人が誰だか全く知らないのである。

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。ただしここでは $\mathtt{w}$ 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 $\mathtt{x} \in \{0,\ldots,2^{\mathtt{w}}-1\}$ 仮定し、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場面が多岐に渡るのは想像に難くないだろう。

この章では以上のことをふまえた 3 つのデータ構造を説明する。一つ目は BinaryTrie であり、このデータ構造は SSet の 3 つの操作をいずれも $O(\mathtt{w})$ の時間で実行できる。この実行時間はさして驚くようなものではないだろう。$\{0,\ldots,2^{\mathtt{w}}-1\}$ の部分集合の大きさ $\mathtt{n}$ は $\mathtt{n} \le 2^{\mathtt{w}}$ であり、$\log \mathtt{n} \le \mathtt{w}$ が成り立つからだ。この本でこれまで解説した SSet の実装では、各操作の実行時間は $O(\log \mathtt{n})$ 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。

二つ目は XFastTrie であり、このデータ構造は BinaryTrie の検索をハッシュ法を使って高速化するものである。この高速化により、find(x) の実行時間が $O(\log \mathtt{w})$ になる。しかし XFastTrie における add(x)・remove(x) の実行時間は依然として $O(\mathtt{w})$ であり、領域使用量は $O(\mathtt{n}\cdot\mathtt{w})$ である。

三つ目は YFastTrie であり、このデータ構造はおよそ $\mathtt{w}$ 個ごとにひとつの要素だけは XFastTrie に、それ以外の要素はふつうの SSet に格納するデータ構造である。この工夫により add(x)・remove(x) の実行時間は $O(\log \mathtt{w})$ になり、領域使用量は $O(\mathtt{n})$ に抑えられる。

この章の実装は整数と対応付けられる任意の型のデータを格納できる。サンプルコードにおいて、ix は x に対応する整数を表し、intValue(x) は x を ix に変換する関数とする。また簡単のため、文中では x を整数として扱う。

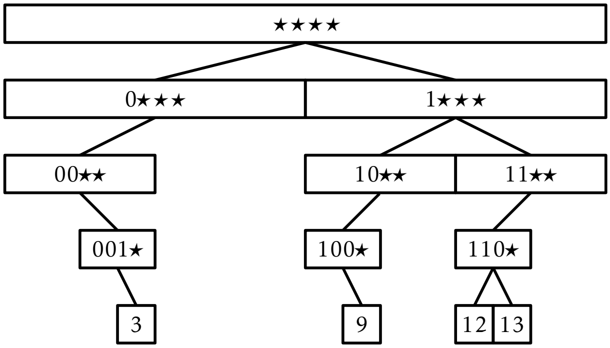


図 13.1: 二分トライでは、整数を根から葉への経路として符号化する。

13.1 BinaryTrie：デジタル探索木

BinaryTrie 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。 この木では、葉の深さはいずれも w で、根から葉への経路がある整数を表している。整数 x への経路は、深さ i において、上から i 番目のビットが 0 なら左、1 なら右に向かう。図 13.1 は $w = 4$ の場合の例を示しており、ここでは整数 3(0011), 9(1001), 12(1100), 13(1101) が二分トライに格納されている。

x の探索経路は x の二進数表記から決まるので、ノード u の子を指すポインタを u.child[0] (left)・u.child[1] (right) と書くと便利である。葉ではこのポインタを別の用途に使う。葉には子がいないので、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、二分トライの葉では、u.child[0] (prev) はリストにおける u の直前のノードを指し、u.child[1] (next) はリストにおける u の直後のノードを指す。特別なノード dummy は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である。（3.2 節を参照せよ。）

各ノード u はポインタ u.jump も保持している。u が左の子を持たないとき、u.jump 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、u.jump 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とを描いた例が図 13.2 である。

BinaryTrie における find(x) は簡単だ。x の探索経路を辿ればよい。葉に辿り着いたら、x が存在することがわかる。（進みたい方向の子がいないため）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。この

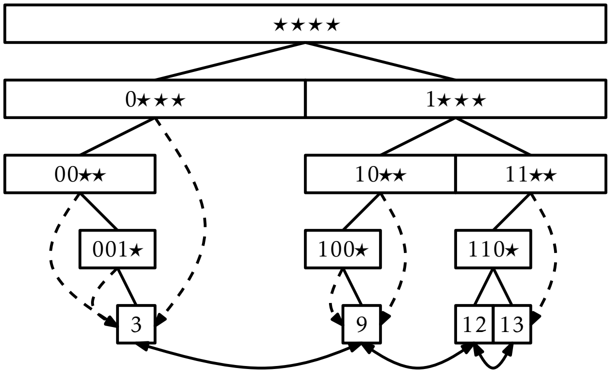


図 13.2: 二分トライにおける jump ポインタを破線の矢印で、リストのリンクを実線の矢印で表す。

どちらになるかは、u が左右どちらの子がいないのかに応じて決まる。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。[*1] u が右の子を持たないなら、連結リストを辿れば欲しいノードが見つかる。図 13.3 にはこのふたつの場合を描いた。

BinaryTrie

```
T find(T x) {
  int i, c = 0, ix = it.intValue(x);
  Node u = r;
  for (i = 0; i < w; i++) {
    c = (ix >>> w-i-1) & 1;
    if (u.child[c] == null) break;
    u = u.child[c];
  }
  if (i == w) return u.x;  // found it
  u = (c == 0) ? u.jump : u.jump.child[next];
  return u == dummy ? null : u.x;
```

[*1] 訳注：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。

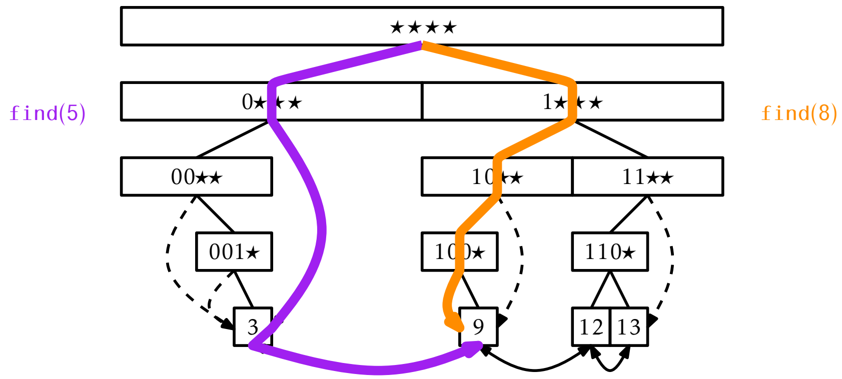


図 13.3: find(5) および find(8) が辿る経路

```
}
```

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、この時間は $O(w)$ である。

BinaryTrie における add(x) も単純だが、その手順は長い。

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 ポインタを使って、連結リストにおける u′ の直前のノード pred を得られる。)
4. これまで来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した。

BinaryTrie
```
boolean add(T x) {
  int i, c = 0, ix = it.intValue(x);
  Node u = r;
  // 1 - search for ix until falling out of the trie
  for (i = 0; i < w; i++) {
    c = (ix >>> w-i-1) & 1;
```

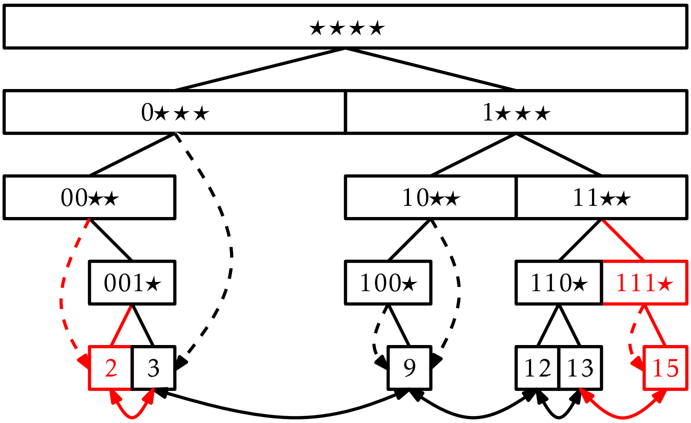


図 13.4: 図 13.2 の BinaryTrie に、値 2、値 15 を追加する。

```
    if (u.child[c] == null) break;
    u = u.child[c];
  }
  if (i == w) return false; // already contains x - abort
  Node pred = (c == right) ? u.jump : u.jump.child[0];
  u.jump = null;  // u will have two children shortly
  // 2 - add path to ix
  for (; i < w; i++) {
    c = (ix >>> w-i-1) & 1;
    u.child[c] = newNode();
    u.child[c].parent = u;
    u = u.child[c];
  }
  u.x = x;
  // 3 - add u to linked list
  u.child[prev] = pred;
  u.child[next] = pred.child[next];
  u.child[prev].child[next] = u;
```

```
  u.child[next].child[prev] = u;
  // 4 - walk back up, updating jump pointers
  Node v = u.parent;
  while (v != null) {
    if ((v.child[left] == null
        && (v.jump == null || it.intValue(v.jump.x) > ix))
    || (v.child[right] == null
        && (v.jump == null || it.intValue(v.jump.x) < ix)))
      v.jump = u;
    v = v.parent;
  }
  n++;
  return true;
}
```

このメソッドはまず x の探索経路を辿り、その後に根方向に向かって戻る。この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 $O(\mathtt{w})$ である。

remove(x) は add(x) のすることを取り消す。add(x) と同様に手順は長い。

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 を見つけるまで 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。
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 に削除の様子を描いた。

BinaryTrie

```
boolean remove(T x) {
  // 1 - find leaf, u, containing x
  int i, c, ix = it.intValue(x);
  Node u = r;
```

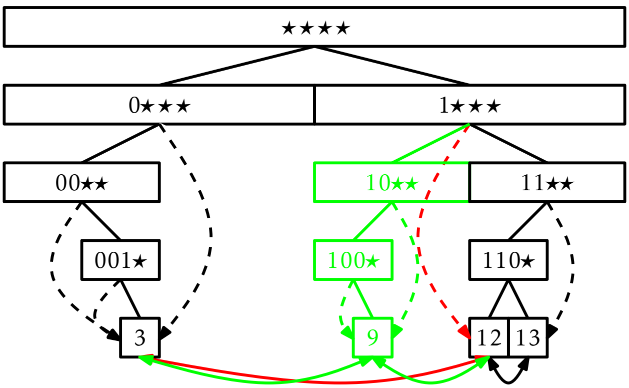


図 13.5: 図 13.2 の BinaryTrie から値 9 を削除する。

```
for (i = 0; i < w; i++) {
  c = (ix >>> w-i-1) & 1;
  if (u.child[c] == null) return false;
  u = u.child[c];
}
// 2 - remove u from linked list
u.child[prev].child[next] = u.child[next];
u.child[next].child[prev] = u.child[prev];
Node v = u;
// 3 - delete nodes on path to u
for (i = w-1; i >= 0; i--) {
  c = (ix >>> w-i-1) & 1;
  v = v.parent;
  v.child[c] = null;
  if (v.child[1-c] != null) break;
}
// 4 - update jump pointers
c = (ix >>> w-i-1) & 1;
v.jump = u.child[1-c];
```

```
    v = v.parent;
    i--;
    for (; i >= 0; i--) {
      c = (ix >>> w-i-1) & 1;
      if (v.jump == u)
        v.jump = u.child[1-c];
      v = v.parent;
    }
    n--;
    return true;
}
```

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 $O(\mathtt{w})$ である。n 個の要素を格納する *BinaryTrie* の領域使用量は $O(\mathtt{n}\cdot\mathtt{w})$ である。

13.2 XFastTrie：$log(log n)$ 時間で検索を行う

BinaryTrie の性能はパッとしないものであった。要素数 n は最大で $2^{\mathtt{w}}$ であり、$\log \mathtt{n} \le \mathtt{w}$ が成り立つ。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、少なくとも BinaryTrie と同じくらいは効率的であり、一方で BinaryTrie のように整数しか格納できない制限はなかった。

次は XFastTrie を説明する。これは単なる BinaryTrie に加えて、各深さにひとつずつ、合計 $\mathtt{w}+1$ 個のハッシュテーブルを置いたものである。これらのハッシュテーブルを使って、find(x) の実行時間を $O(\log \mathtt{w})$ に改善できる。BinaryTrie における find(x) は、x の探索経路を辿り、進みたい方向の子を持たないノード u を見つければ、ほぼ完了であった。あとは、u.jump を利用して葉 v にジャンプし、v か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った。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、XFastTrie はこの探索処理を高速に行う。

二分探索を行うためには探しているノード u が、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 の二進表現

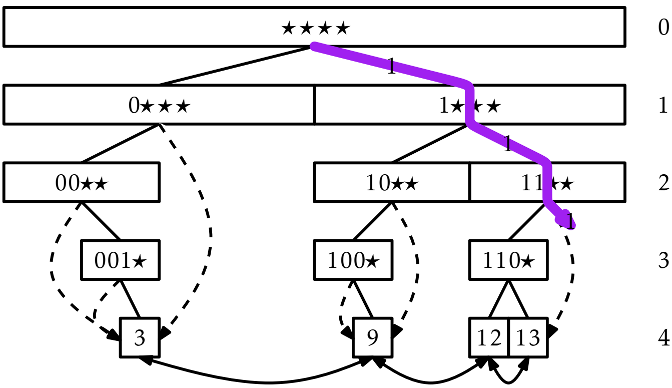


図 13.6: ラベル 111★ を持つノードは存在しないので、14 (1110) の探索経路はラベル 11★★ を持つノードで終了する。

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（二進表現では 1110）の探索経路における最後のノード u は、深さ 2 にある 11★★ とラベル付けられたノードである。これは深さ 3 に 111★ とラベル付けられたノードが無いためである。このように、深さ i のノードをみな i ビットの整数でラベル付けられる。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、かつそのときに限る。

XFastTrie では、$i \in \{0,\dots,w\}$ 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）で実装する。USet を使うと、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を期待定数時間で判定できる。具体的には、このノードを次のように見つけられる。t[i].find(x>>>(w − i))

ハッシュテーブル t[0],...,t[w] によって、二分探索で u を見つけられる。最初は、$0 \le i < w+1$ を満たすある深さ i に u があることを知っている。まずは $l = 0, h = w+1$ とする。$i = \lfloor (l+h)/2 \rfloor$ として、ハッシュテーブル t[i] を繰り返し検索する。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、$l = i$ とする。（このとき、u は深さ i、またはそれよりも下にある。）そうでなければ、$h = i$ とする。（このとき、u は深さ i よりも上にある。）$h - l \le 1$ になればこの処理を終える。このとき、u は深さ l にある。あとは u.jump と葉の双方向連結リストとを使って、find(x) の処理を完了する。

XFastTrie

```
T find(T x) {
  int l = 0, h = w+1, ix = it.intValue(x);
  Node v, u = r, q = newNode();
  while (h-l > 1) {
    int i = (l+h)/2;
    q.prefix = ix >>> w-i;
    if ((v = t[i].find(q)) == null) {
      h = i;
    } else {
      u = v;
      l = i;
    }
  }
  if (l == w) return u.x;
  Node pred = (((ix >>> w-l-1) & 1) == 1)
         ? u.jump : u.jump.child[0];
  return (pred.child[next] == dummy)
               ? null : pred.child[next].x;
}
```

上のメソッドの while ループにおける各繰り返しにおいて、h − l は約半分になる。よって、このループを $O(\log \mathtt{w})$ 回繰り返すと u が見つかる。各繰り返しでは一定量の仕事をし、一回だけ USet の find(x) を呼ぶ。USet における検索の実行時間の期待値は定数である。残りの処理の実行時間も定数なので、XFastTrie における find(x) の実行時間の期待値は $O(\log \mathtt{w})$ である。

XFastTrie における add(x)・remove(x) は BinaryTrie におけるそれらの操作とほとんど同じである。修正点はハッシュテーブル t[0],...,t[w] を更新することだけである。add(x) の実行中に深さ i でノードが作られるとき、このノードを t[i] に加える。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。ハッシュテーブルにおける追加・削除の実行時間の期待値は定数なので、この修正によって add(x)・remove(x)

の実行時間の期待値は定数程度しか増えない。add(x)・remove(x) のコードは、BinaryTrie のときに提示した（長い）コードとほぼ同じなので、ここには掲載しない。

次の定理は XFastTrie の性能をまとめたものだ。

定理 **13.2.** *XFastTrie* は w ビット整数の *SSet* インターフェースを実装する。*XFastTrie* がサポートするのは次の操作である。

- add(x)・remove(x) の実行時間の期待値は $O(\mathtt{w})$ である。
- find(x) の実行時間の期待値は $O(\log \mathtt{w})$ である。

n 個の要素を格納する *XFastTrie* の領域使用量は $O(\mathtt{n}\cdot\mathtt{w})$ である。

13.3 YFastTrie：実行時間が Doubly-Logarithmic な SSet

BinaryTrie と比べると XFastTrie における検索の実行時間は指数的に速くなった。しかし、add(x)・remove(x) の実行時間は依然として速くない。さらに、領域使用量は $O(\mathtt{n}\cdot\mathtt{w})$ であり、この本で紹介した他の SSet の実装の $O(\mathtt{n})$ と比べて大きい。このふたつの問題は関連している。n 回の add(x) によって大きさ $\mathtt{n}\cdot\mathtt{w}$ の構造を作るなら、add(x) の一回あたりの実行時間・領域使用量は w 程度のオーダーになる。

次に紹介する YFastTrie は、XFastTrie の実行時間と領域使用量を改善する。YFastTrie は XFastTrie xft を使うが、xft には $O(\mathtt{n}/\mathtt{w})$ 個の値しか入れない。こうすると xft の領域使用量は $O(\mathtt{n})$ になる。さらに、add(x)・remove(x) を w 回実行するとき、そのうち 1 回のみ xft に add(x)・remove(x) を実行する。こうして、xft における add(x)・remove(x) の平均実行時間は定数になる。

きっと疑問を感じるだろう。xft には $\mathtt{n}/\mathtt{w}$ 個の要素しか格納しないなら、残りの $\mathtt{n}(1-1/\mathtt{w})$ 個の要素はどこに行くのだろう。これらの要素は**二次構造**である Treap(7.2 節) を拡張したデータ構造に格納する。二次構造は約 $\mathtt{n}/\mathtt{w}$ 個あり、平均的にはそれぞれ $O(\mathtt{w})$ 個の要素を格納する。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期待通り $O(\log \mathtt{w})$ である。

具体的には、YFastTrie は確率 $1/\mathtt{w}$ で選り抜いたデータを格納する XFastTrie xft を保持する。都合上、xft は常に値 $2^{\mathtt{w}}-1$ を含むものとする。また、xft が含む要素を $\mathtt{x}_0 < \mathtt{x}_1 < \cdots < \mathtt{x}_{k-1}$ とする。要素 $\mathtt{x}_i$ には対応する Treap $\mathtt{t}_i$ があり、これは $\mathtt{x}_{i-1}+1,\ldots,\mathtt{x}_i$ の範囲の値をすべて格納する。図 13.7 にこの

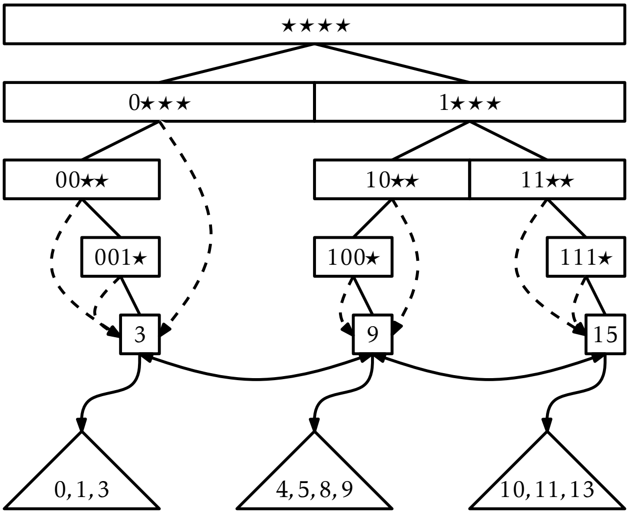


図 13.7: 0, 1, 3, 4, 6, 8, 9, 10, 11, 13 を含む YFastTrie

様子を示す。

YFastTrie における find(x) は簡単である。x を xft から検索し、Treap t_i を見つける。続いて、t_i の find(x) メソッドを使って、問い合わせに答える。このメソッド全体を一行で書ける。

YFastTrie
```
T find(T x) {
  return xft.find(new Pair<T>(it.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。二回目の 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。この節の後半で Treap の大きさの期待値は $O(w)$ であることを示すので、結局この操作の実行時間は $O(\log w)$ である。[*2]

YFastTrie に要素を追加するのも、ほとんどの場合は単純である。add(x) メソッドは xft.find(x) を呼んで、x を挿入すべき Treap t を特定する。続い

[*2] これは Jensen の不等式の応用すればよい。$E[r] = w$ ならば $E[\log r] \le \log w$ である。

て t.add(x) を呼んで 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、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。

これが少し複雑なところである。x を xft に追加するとき、Treap t をふたつの Treap t1, t′ に分割しなければならない。t1 は x 以下の値をすべて含む。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、組み (x, t1) を xft に追加する。図 13.8 に例を示す。

YFastTrie
```
boolean add(T x) {
  int ix = it.intValue(x);
  STreap<T> t = xft.find(new Pair<T>(ix)).t;
  if (t.add(x)) {
    n++;
    if (rand.nextInt(w) == 0) {
      STreap<T> t1 = t.split(x);
      xft.add(new Pair<T>(ix, t1));
    }
    return true;
  }
  return false;
}
```

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、t を分割し t1 と t′ と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、これは確率 $1/w$ でのみ起きる。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) のしたことを取り消す。xft.find(x) により葉 u を見つける。u から x を含む Treap t を得て、t 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 の Treap の要素をすべて 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

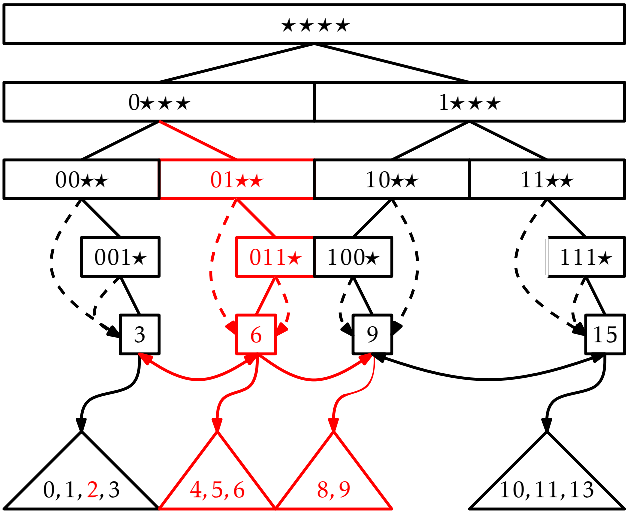


図 13.8: YFastTrie に値 2、値 6 を追加する。6 を追加する時のコイン投げで表が出たので、6 は xft に追加され、4,5,6,8,9 を含む Treap は分割される。

示す。

YFastTrie
```
boolean remove(T x) {
  int ix = it.intValue(x);
  Node<T> u = xft.findNode(ix);
  boolean ret = u.x.t.remove(x);
  if (ret) n--;
  if (u.x.x == ix && ix != 0xffffffff) {
    STreap<T> t2 = u.child[1].x.t;
    t2.absorb(u.x.t);
    xft.remove(u.x);
  }
  return ret;
}
```

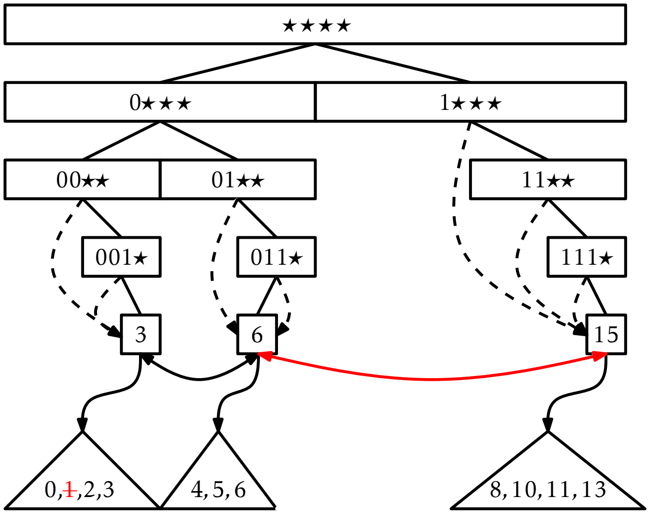


図 13.9: 図 13.8 の YFastTrie から値 1、値 9 を削除する。

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′ と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。

この節の前半で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、必要な結果を示しておく。

補題 13.1. x を *YFastTrie* に格納する整数とし、n_x を x を含む *Treap* t の要素数とする。このとき $E[n_x] \le 2w-1$ が成り立つ。

証明. 図 13.10 を参照せよ。$x_1 < x_2 < \cdots < x_i = x < x_{i+1} < \cdots < x_n$ を YFastTrie の各要素とする。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

$$\begin{matrix}\mathrm{H}\\ \mathtt{x}_{i-k-1}\end{matrix}\ \overbrace{\underbrace{\begin{matrix}\mathrm{T}&\mathrm{T}&\cdots&\mathrm{T}&\mathrm{T}\\ \mathtt{x}_{i-k}&\mathtt{x}_{i-k+1}&\cdots&\mathtt{x}_{i-2}&\mathtt{x}_{i-1}\end{matrix}}_{k}\ \underbrace{\begin{matrix}\mathrm{T}&\mathrm{T}&\mathrm{T}&\cdots&\mathrm{T}&\mathrm{H}\\ \mathtt{x}_i=\mathtt{x}&\mathtt{x}_{i+1}&\mathtt{x}_{i+2}&\cdots&\mathtt{x}_{i+j-2}&\mathtt{x}_{i+j-1}\end{matrix}}_{j}}^{\text{elements in treap, } \mathtt{t}\text{, containing } \mathtt{x}}$$

図 13.10: x を含む treap t の要素数は二回の連続コイン投げ試行により決まる。

ときの、コイン投げ回数の期待値と等しい。[*3] コイン投げは独立な試行であり、表が出る確率は確率 $1/\mathtt{w}$ である。そのため $\mathrm{E}[j] \le \mathtt{w}$ である。（$\mathtt{w} = 2$ の場合の解析として補題 4.2 を参照せよ。）

同様に、t の x よりも小さい要素 $\mathtt{x}_{i-1},\ldots,\mathtt{x}_{i-k}$ について、これらの k 回のコイン投げはいずれも裏であり、$\mathtt{x}_{i-k-1}$ 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の、裏が出る回数を考えると $\mathrm{E}[k] \le \mathtt{w}-1$ だとわかる。

まとめると、$\mathtt{n}_\mathtt{x} = j + k$ より、

$$\mathrm{E}[\mathtt{n}_\mathtt{x}] = \mathrm{E}[j+k] = \mathrm{E}[j] + \mathrm{E}[k] \le 2\mathtt{w} - 1 \qquad \square$$

補題 13.1 が次の定理を示す最後のピースであった。次の定理は YFastTrie の性能をまとめるものである。

定理 13.3. *YFastTrie* は w ビット整数の *SSet* を実装する。*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 $O(\log \mathtt{w})$ である。n 要素を格納する *YFastTrie* の領域使用量は $O(\mathtt{n}+\mathtt{w})$ である。

領域使用量における項 w があるのは xft が常に値 $2^\mathtt{w}-1$ を格納しているためである。実装を修正し、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、上の定理における領域使用量は $O(\mathtt{n})$ になる。

[*3] この解析は j が $\mathtt{n}-i+1$ を越えることがないことを無視している。しかし、これは $\mathrm{E}[j]$ を減らすため、上界に関する性質はやはり成り立つ。

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はじ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。[74] オリジナルの van Emde Boas 木の大きさは 2^w で、このために大きな整数についてこのデータ構造は非実用的であった。

XFastTrie・YFastTrie は Willard [77] によって提案された。XFastTrie と van Emde Boas 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 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他のデータ構造としては、Fredman と Willard の fusion tree 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 tree を、$\log w \le \sqrt{\log n}$ なら YFastTrie を使えば、領域使用量 $O(n)$ であり、find(x) にかかる時間は $O(\sqrt{\log n})$ であるデータ構造が得られる。近年の Pătraşcu and Thorup [59] 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である。

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ジャンプポインタを持たないが、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。これは二分トライを使わない。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、$d(x)$ を x と find(x) の返り値との差と定義する。（なお、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 $O(1 + \log d(\mathtt{x}))$ であるものを設計・実装せよ。ヒント：ハッシュテーブル $t[\mathtt{w}]$ は $d(\mathtt{x}) = 0$ であるすべての値 `x` を格納することで、処理を開始する良い位置を見つけられる。
2. `XFastTrie` における `find(x)` を修正し、実行時間の期待値が $O(1 + \log\log d(\mathtt{x}))$ 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。これは、コンピュータのランダムアクセスメモリはデータ構造内のすべてのデータを格納できるくらい大きいと暗に仮定していたということである。しかし時にはこの仮定が成り立たないこともある。大きすぎてどんなコンピュータのメモリにも収まりきらないデータはたくさん存在するのである。このような場合、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）・ネットワーク越しのサーバーなどの外部ストレージにデータを蓄えざるを得ない。

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、SSD の場合は 0.3ms である。これに対してにランダムアクセスメモリの場合は 0.000113ms 未満である。RAM へのアクセスは SSD と比べて 2500 倍、HDD と比べて 160000 倍以上高速である。

これらの速度は典型的な値である。RAM へのランダムアクセスは HDD や SSD へのランダムアクセスと比べて数千倍高速である。しかしアクセスにかかる時間だけを考えれば十分というわけではない。HDD や SSD のバイトにアクセスするときには、実際は**ブロック** (block) 単位での読み出しが行われる。コンピュータに接続されるドライブは大きさ 4096 のブロックを持つ [*1]。1 バイトの読み出しをする度にドライブは 4096 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、この 4096 バイトを操作にうまく利用することができるだろう。

これが**外部メモリモデル** (external memory model) の背後にあるアイデア

[*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 4096 で統一している。

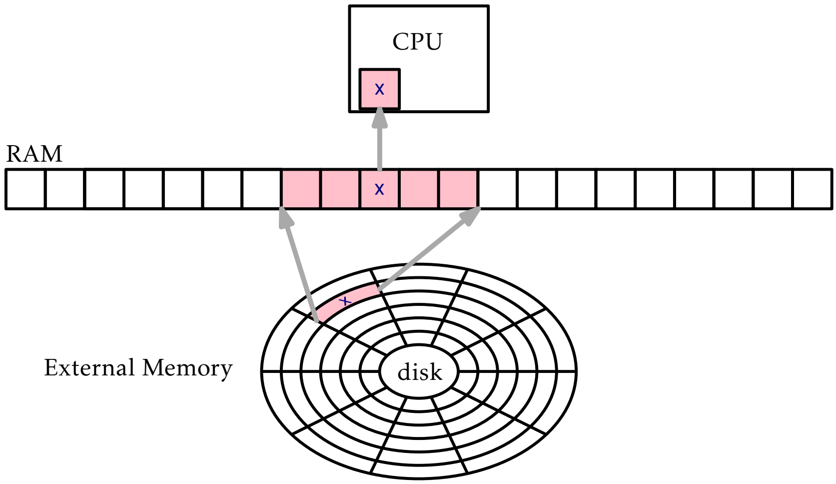


図 14.1: 外部メモリモデルでは、外部メモリに含まれる要素 x にアクセスするためには x を含むブロックをまるごと RAM に読み込む必要がある。

である。図 14.1 にはこれを図示した。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**に分割されており、それぞれのブロックは B ワードからなる。このコンピュータは有限の内部メモリも利用でき、ここで計算を実行できる。内部メモリと外部メモリの間でブロックを転送するのには一定の時間がかかる。内部メモリでの計算は**タダ**である。つまり一切の時間がかからない。この仮定は奇妙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た結果である。

本格的な外部メモリモデルでは内部メモリの大きさもパラメータである。しかし、この章で扱うデータ構造では内部メモリのサイズは $O(B+\log_B \mathtt{n})$ で十分である。これは定数個だけのブロックと、高さ $O(\log_B \mathtt{n})$ だけのスタックに必要なメモリである。多くの場合 $O(B)$ が支配的な項である。例えば比較的小さい値 $B=32$ であっても、すべての $\mathtt{n} \le 2^{160}$ について $B \ge \log_B \mathtt{n}$ が成り立つ。十進法では、$B \ge \log_B \mathtt{n}$ が以下の範囲で成り立つ。

$$\mathtt{n} \le 1\,461\,501\,637\,330\,902\,918\,203\,684\,832\,716\,283\,019\,655\,932\,542\,976$$

14.1 BlockStore

外部メモリには HDD や SSD など様々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。説明を単純にし、一般的なアイデアに焦点をあてるため、外部メモリのデバイスを `BlockStore` と呼ばれるオブジェクトに隠蔽する。`BlockStore` はブロックの集まりを格納する。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によって一意に特定される。`BlockStore` は次の操作をサポートする。

1. `readBlock(i)`：インデックス `i` のブロックの内容を返す。
2. `writeBlock(i,b)`：インデックス `i` のブロックに `b` の内容を書く。
3. `placeBlock(b)`：新たなインデックスを返し、そこに `b` の内容を書く。
4. `freeBlock(i)`：インデックス `i` の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は別の用途に使ってよいことを示す。

`BlockStore` は B バイトのブロックに分割されたディスク上のファイルだと考えるのが最も想像しやすいだろう。このとき、`readBlock(i)`・`writeBlock(i,b)` は、このファイルのバイト列 $\mathtt{i}B,\ldots,(\mathtt{i}+1)B-1$ の読み・書きである。さらに、`BlockStore` 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。`freeBlock(i)` により解放されたブロックはフリーリストに追加される。こうすれば、`placeBlock(b)` はフリーリストのブロックを使い、利用可能なブロックがないときだけファイルの末尾に新しいブロックを追加するようにできる。

14.2 B 木

この節では二分木の一般化である、B 木と呼ばれる、外部メモリモデルにおいて効率的なデータ構造を紹介する。それ以外にも B 木は 9.1 節で説明した 2-4 木の自然な一般化だと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。）

任意の整数 $B\geq 2$ について ***B* 木**とは木であって、すべての葉は同じ深さにあり、すべての根でない内部ノードの子の数が B 以上 $2B$ 以下なものである。ノード `u` の子は配列 `u.children` に格納される。ただし、子の数の条件は根

では緩和され、2 以上 $2B$ 以下である。

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 個のみで全ての内部ノードが B 個の子を持つときの葉の数に対応し、右辺は葉以外のノードの子が全て $2B$ 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`$[0],\ldots,$`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 $k-1$ であり、それぞれ `u.keys`$[0],\ldots,$`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 木のキーは二分探索木と同様の順序を守る。$k-1$ 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。

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

$B = 2$ である B の例を図 14.2 に示す。

[*2] 訳注：この本では B 木はキーの重複がない SSet インターフェースを実装するために使うため、等号なし不等号による。キーの重複がある multiset の実装時は `u.keys`$[0] \leq$ `u.keys`$[1] \leq \cdots \leq$ `u.keys`$[k-2]$、`u.children[i]` $\leq$ `u.keys[i]` $\leq$ `u.children[i+1]` を満たす。

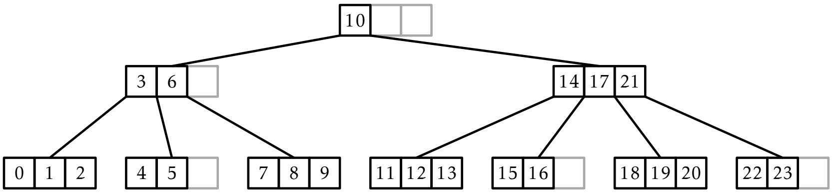


図 14.2: $B = 2$ である B 木

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のことを考えると、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ばれる。こ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、操作の際にアクセス（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

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

バイトのデータを格納することになる。この章の最初に説明したように、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 のノードを格納する BlockStore オブジェクト bs と、根のインデックス ri を格納する。また、他のデータ構造と同様に、整数 n はデータ構造の要素数を表す。

BTree
```
int n;
BlockStore<Node> bs;
int ri;
```

14.2.1 要素の検索

図 14.3 に示した find(x) の実装は、二分探索木における find(x) の実装の一般化である。x を検索するために、根から処理を開始し、ノード u のキーを利用して次に u の子のうちのどれに進むべきかを決める。

より具体的には、ノード u にいるとき、まずは x が u.keys に格納されて

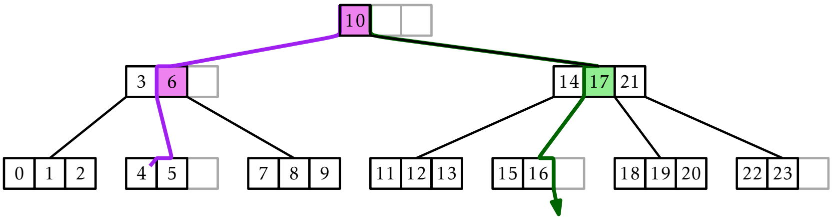


図 14.3: B 木における成功する探索（4 を探す）と、失敗する探索（16.5 を探す）の様子。色を付けたノードは探索の途中に値を更新されるものである。

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。そうでなければ、最小の 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。もし 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。二分探索木と同じように、このアルゴリズムは 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)]; // found it
    if (u.keys[i] != null)
      z = u.keys[i];
    ui = u.children[i];
  }
  return z;
}
```

find(x) の肝は findIt(a,x) であり、これは null 埋めされた配列 a から x

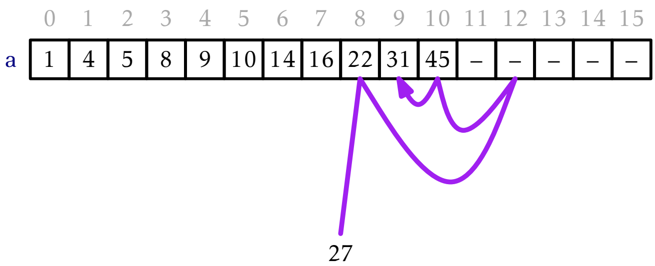


図 14.4: findIt(a,27) を実行する様子

を探すメソッドである。図 14.4 に示したように、a[0],...,a[$k-1$] にはキーが整列された状態で、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 番目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(T[] a, T x) {
  int lo = 0, hi = a.length;
  while (hi != lo) {
    int m = (hi+lo)/2;
    int cmp = a[m] == null ? -1 : compare(x, a[m]);
    if (cmp < 0)
      hi = m;       // look in first half
    else if (cmp > 0)
      lo = m+1;     // look in second half
    else
      return -m-1; // found it
  }
  return lo;
}
```

findIt(a,x) は二分探索を使う。これは各ステップで探索空間を半分に減らすことで、$O(\log($a.length$))$ の時間で処理を終える。ここでは 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)` の実行時間をいつものワード RAM モデル（全命令を数える）でも、または外部メモリモデル（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 木の葉は少なくともひとつのキーを格納し、ℓ 個の葉を持つ B 木の高さは $O(\log_B \ell)$ なので、$\mathtt{n}$ 個のキーを格納する B 木の高さは $O(\log_B \mathtt{n})$ である。よって、外部メモリモデルでは、`find(x)` の実行時間は $O(\log_B \mathtt{n})$ 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。よって、この場合の `find(x)` の実行時間は次のようになる。

$$O(\log_B \mathtt{n}) \times O(\log B) = O(\log \mathtt{n})$$

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 木のノードは親へのポインタを持っていないことである。この理由を軽く説明する。親へのポインタがないため、B 木における `add(x)`・`remove(x)` は再帰を使うのがもっとも簡単な実装方法となる。

他のバランスされた探索木と同様に、`add(x)` の際に何らかのバランス調整が必要になる。B 木ではこれはノードの**分割**によって行われる。図 14.5 を見ながら続く説明を読んで欲しい（訳注：¢ 記号は後の節で実行時間の解析に使うため、ここでは無視してよい。）。分割はふたつの再帰レベルに渡って起きるが、$2B$ 個のキーを含み $2B+1$ 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、このノードは `u.children`$[B],\ldots,$`u.children`$[2B]$ を引き受ける。`u` のキーのうち大きい方から B 個 `u.keys`$[B],\ldots,$`u.keys`$[2B-1]$ も `w` に持たせる。この段階で、`u` は B 個の子と、B 個のキーを持っている。さらに追加で `u.keys`$[B-1]$ は `u` の親に渡され、`u` の親は `w` も子として引き受ける。

分割操作は 3 つのノードを修正する。これは `u`・`u` の親・新たなノード `w` である。これが B 木において親へのポインタを持たない理由なのである。もし親へのポインタがあると、`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。これは外部メモリのアクセスを 3 回から $B+4$ 回に増やすことになり、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。概要としては、まずこのメソッドは値 `x` を追加する葉 `u` を見つける。このときに `u` が一杯になった場合

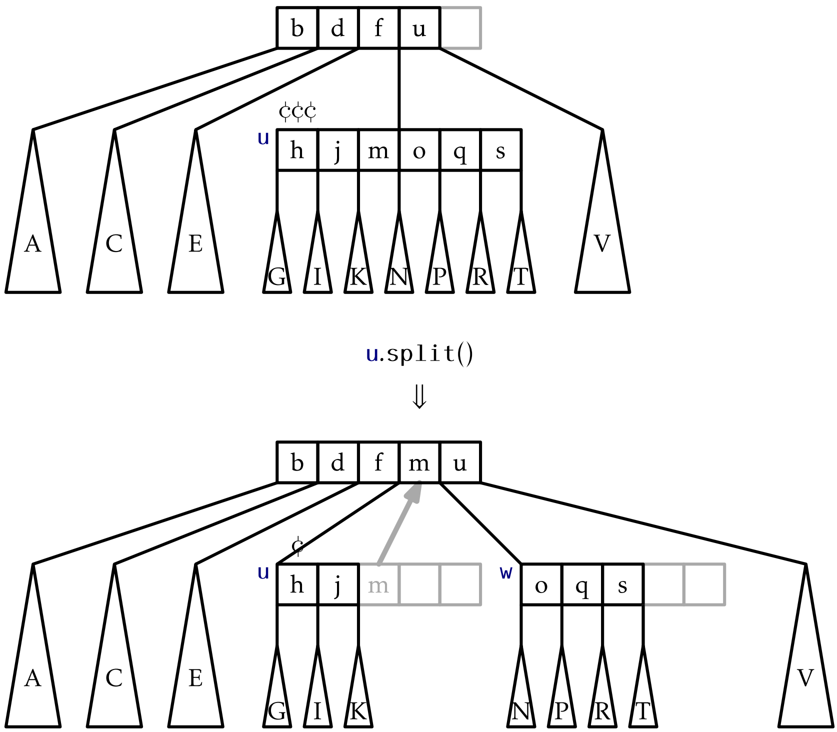


図 14.5: $B = 3$ である B 木における、ノード u の分割。キー u.keys[2] = m は u からその親に移る。

には（つまり既に $2B-1$ 個のキーを持っていれば）、u を分割する。この結果 u の親が一杯になった場合には、u の親を分割する。さらにその結果 u の親の親が一杯になった場合には... ということを繰り返す。ひとつずつ木を上に登りながら、一杯でないノードを見つけるか、根を分割するまでこの処理を繰り返す。一杯でないノードが見つかった場合には単に処理を終了する。根を分割した場合には、新たな根を作り、元の根を分割して得られたふたつのノードを共に新たな根の子にする [*3]。

add(x) の実行について整理すると、まずは根からスタートし、x を追加すべき葉を見つけ、その葉に x を追加し、根に向かって戻りながら、その途中で見かけた一杯になったノードを分割する。このような概要を頭に入れて、次はこ

[*3] 訳注：似た議論を 9.1.1 節で行なったことを思い出そう。

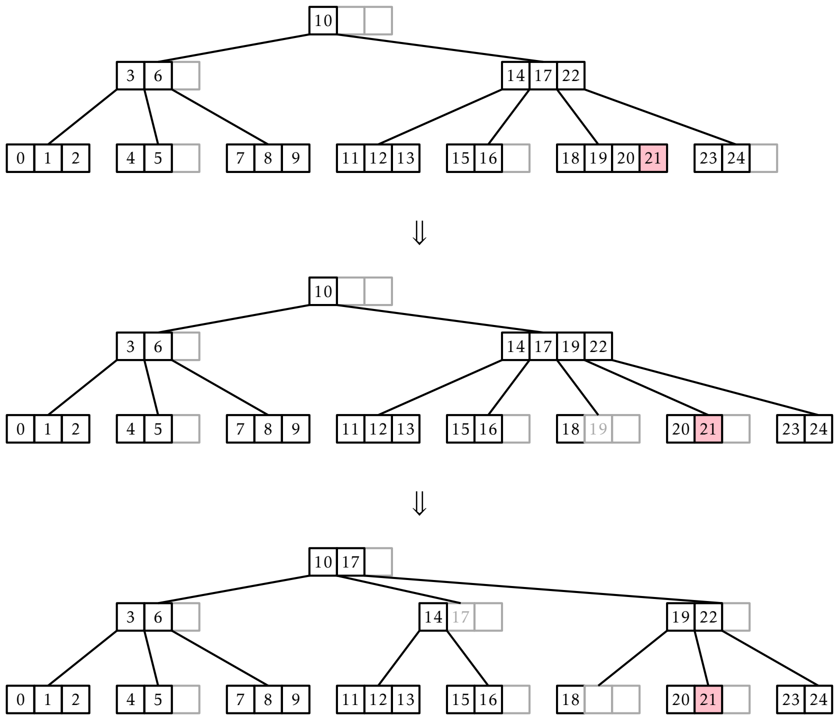


図 14.6: BTree における add(x) の様子。値 21 を追加すると、ふたつのノードが分割される。

のメソッドをどう再帰的に実装するかの詳細を見ていく。

add(x) が行う処理のほとんどは addRecursive(x,ui) が担当する。これは識別子 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、x を u の子のうち適切なもの u′ に再帰的に追加する。この再帰的な呼び出しはふつうは null を返すが、u′ が分割されるときは新たに作られたノード w の参照を返す。この場合、u は w を子にした上で、その w の最初のキーを奪って、u′ の分割処理を終える。

x が（u または u の子孫に）追加された後、addRecursive(x,ui) は 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。もしそうなら u を**分割**しなければならず、u.split() を呼ぶ。u.split() の返り値である新しいノードは 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 new DuplicateValueException();
  if (u.children[i] < 0) { // leaf node, just add it
    u.add(x, -1);
    bs.writeBlock(u.id, u);
  } else {
    Node w = addRecursive(x, u.children[i]);
    if (w != null) {  // child was split, w is new child
      x = w.remove(0);
      bs.writeBlock(w.id, w);
      u.add(x, w.id);
      bs.writeBlock(u.id, u);
    }
  }
  return u.isFull() ? u.split() : null;
}
```

addRecursive(x,ui) は add(x) のヘルパーであり、add(x) は addRecursive(x,ri) を呼んで x を *B* 木の根に挿入する [*4]。addRecursive(x,ri) によって根が分割されるときは、古い根と古い根の分割において新たに作られたノードとを、新たな根は子として持つ。

BTree
```
boolean add(T x) {
  Node w;
  try {
```

[*4] 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。

```
    w = addRecursive(x, ri);
  } catch (DuplicateValueException e) {
    return false;
  }
  if (w != null) {   // root was split, make new root
    Node newroot = new Node();
    x = w.remove(0);
    bs.writeBlock(w.id, w);
    newroot.children[0] = ri;
    newroot.keys[0] = x;
    newroot.children[1] = w.id;
    ri = newroot.id;
    bs.writeBlock(ri, newroot);
  }
  n++;
  return true;
}
```

add(x) とそのヘルパー addRecursive(x,ui) は二段階に分けて分析できる。

下向きに進む段階 再帰の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 $O(\log_B \mathtt{n})$、ワード RAM モデルでは $O(\log \mathtt{n})$ である。

上向きに進む段階 再帰の上向きに進む段階では、x を追加した後、合わせて最大 $O(\log_B \mathtt{n})$ 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 $O(\log_B \mathtt{n})$ である。しかし、各分割は 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、この実行時間は $O(B \log \mathtt{n})$ である。

B の値はかなり大きく、$\log \mathtt{n}$ より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よりもかなり遅いかもしれない。14.2.4 節では、それほど悪くはないことを示す。実は償却すると、add(x) で実行される分割の回数は定数なのである。

そのため、ワード RAM モデルにおける add(x) の償却実行時間は $O(B+\log n)$ なのである。

14.2.3 ノードの削除

BTree における remove(x) も再帰で実装するのが簡単だ。remove(x) を再帰で実装するといくつかのメソッドは複雑になるものの、図 14.7 に示したように全体としては非常に素直になる。ここでは結局、うまくキーを入れ替えて、ある葉 u から値 x′ を削除したい。x′ を削除すると、u の持つキーの数は $B-1$ 未満になるかもしれない。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すると、u は兄弟からキーを借りてくるか、兄弟のいずれかと併合される。u が兄弟と併合される場合には、u の親が持つ子とキーの数はそれぞれ 1 減り、その結果今度は u の親でアンダーフローが発生するかもしれない。これは再度、兄弟からキーを借りるか、兄弟と併合されるかで解決されるが、併合する場合には、今度は u の親の親がアンダーフローするかもしれない。この処理は、根に向かいながら行われ、アンダーフローが発生しなくなるか、根のふたつの子が一つに併合されるかすると終了する。後者の場合には、根は削除され、その唯一の子が新たな根になる。

続いて、各ステップの実装方法を詳細に見ていく。remove(x) はまず、削除したい要素 x を見つける。x が葉で見つかれば、x をこの葉から削除する。そうでなく、x がある内部ノード u の u.keys[i] で見つかれば、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) は上のアルゴリズムの再帰的な実装である。

BTree
```
boolean removeRecursive(T x, int ui) {
  if (ui < 0) return false;  // didn't find it
  Node u = bs.readBlock(ui);
  int i = findIt(u.keys, x);
  if (i < 0) { // found it
    i = -(i+1);
    if (u.isLeaf()) {
```

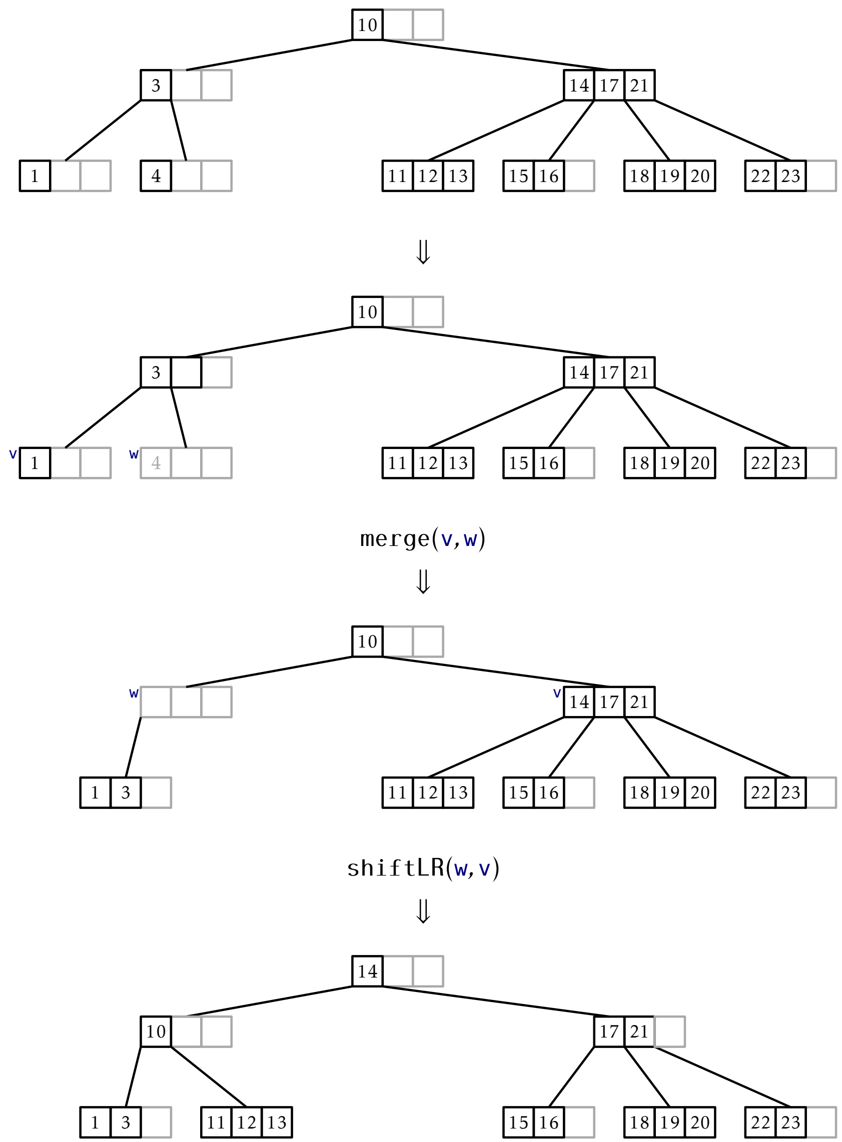


図 14.7: この B 木から値 4 を削除すると、併合・借用が一回ずつ発生する。

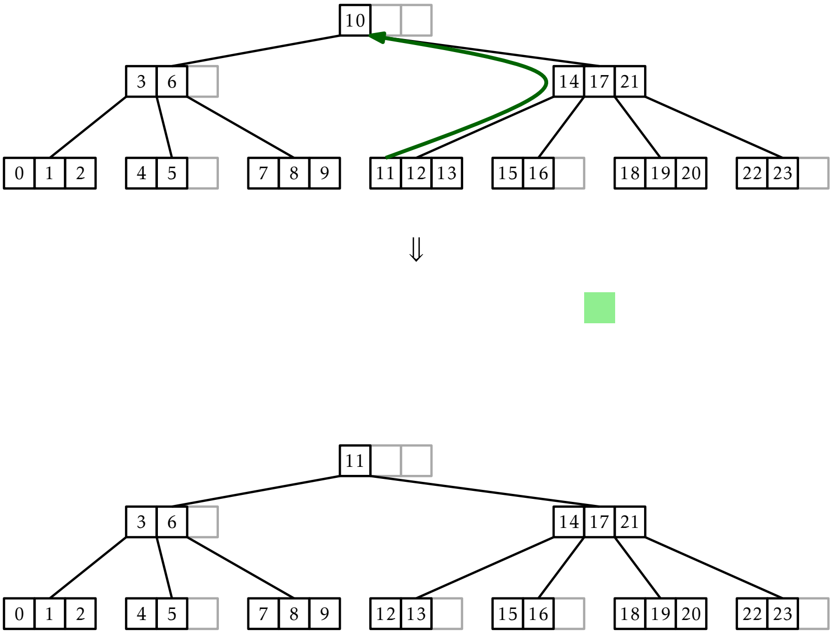


図 14.8: BTree において remove(x) を実行する様子。値 x = 10 を削除するとき、その値を x′ = 11 で上書きし、値 11 を含む葉を削除する。

```
        u.remove(i);
      } else {
        u.keys[i] = removeSmallest(u.children[i+1]);
        checkUnderflow(u, i+1);
      }
      return true;
    } else if (removeRecursive(x, u.children[i])) {
      checkUnderflow(u, i);
      return true;
```

```
    }
    return false;
}
T removeSmallest(int ui) {
    Node u = bs.readBlock(ui);
    if (u.isLeaf())
        return u.remove(0);
    T y = removeSmallest(u.children[0]);
    checkUnderflow(u, 0);
    return y;
}
```

u の i 番目の子から値 x を再帰的に削除したあと、removeRecursive(x,ui) は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。先のコードでは checkUnderflow(x,i) がこの処理を行っている。これは u の i 番目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 番目の子とする。w が $B-2$ 個のキーしか持たないなら、修正の必要がある。これには w の兄弟を利用する。u の $i+1$ 番目または $i-1$ 番目の子を使う。ふつうは u の $i-1$ 番目の子 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); // use u's right sibling
    else
        checkUnderflowNonZero(u,i);
}
```

ここでは $i \neq 0$ の場合のみを考え、u の i 番目の子で発生したアンダーフローは u の $(i-1)$ 番目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、詳細は付属のソースコードを参照してほ

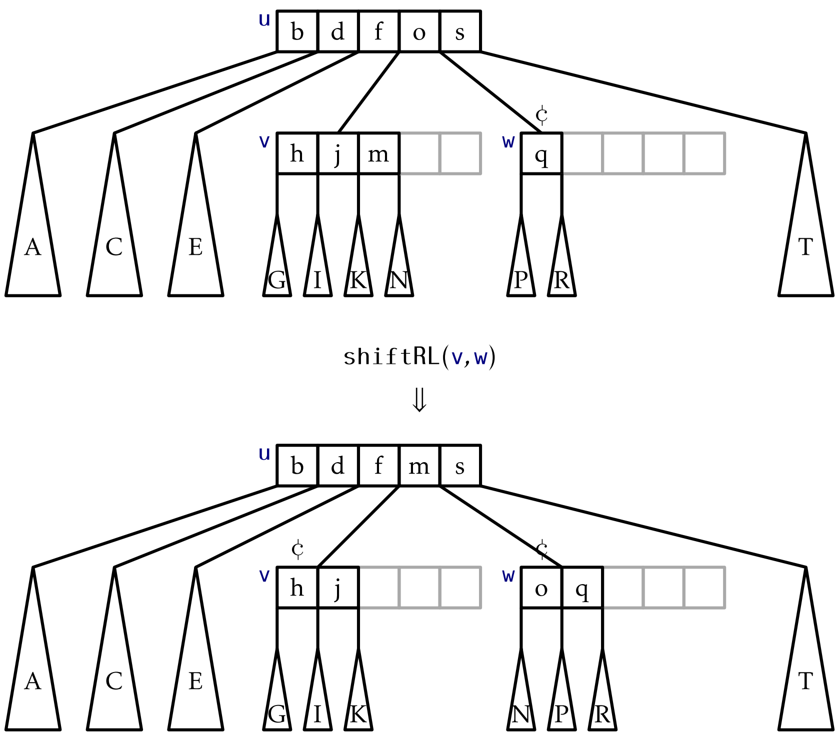


図 14.9: v が $B-1$ 個より多くのキーを持つ時、w は v からキーを借りることができる。

しい。

w におけるアンダーフローを解決するために、w のためのキー（や子）を見つけてくる必要がある。これを行うための操作はふたつある。

借用 w の兄弟 v が $B-1$ 個よりも多くのキーを持っているなら、w はキーを（あとは可能なら子も）v から借りる。より具体的には v が size(v) 個のキーを持つなら、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。

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

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 が $B-1$ 個だけしかキーを持っていないとき、v にはキーを渡す余裕が無いので、もっと思い切ったことをする必要がある。そのために、

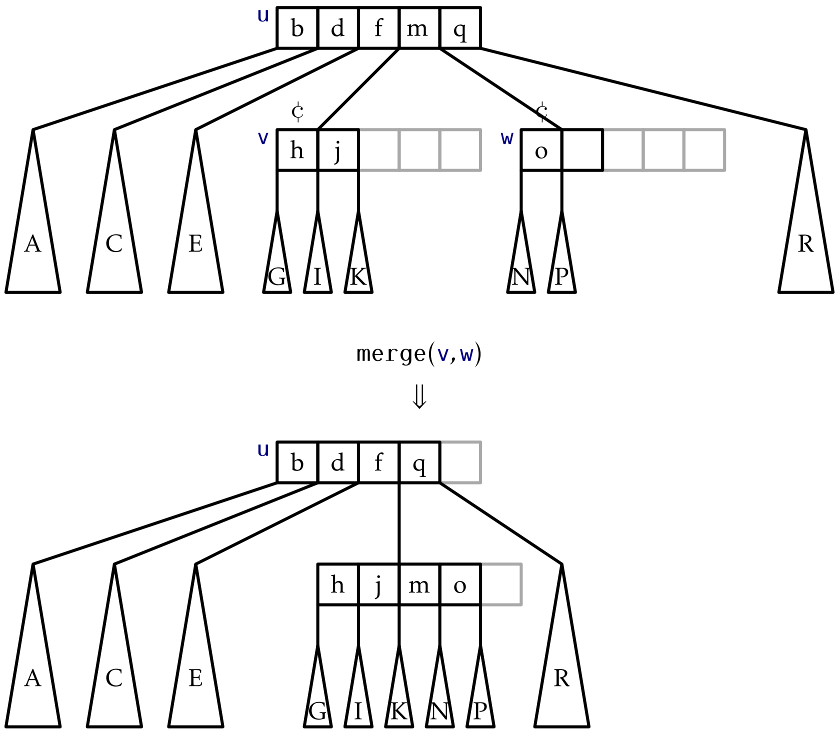


図 14.10: $B = 3$ である B 木の兄弟 v・w を併合する。

図 14.10 に示したように v と w とを**併合**する。併合は分割の逆の操作である。これは合わせて $2B-3$ 個のキーを持つふたつのノードを併合し、$2B-2$ 個のキーを持つひとつのノードとする操作である。(併合されたノードのキー数がひとつ増えているのは、v と w を併合すると、それらの共通の親 u の子の数がひとつ減ることから、u は保有するキーのひとつを、併合されたノードに受け渡す必要があるためだ。)

BTree
```
void checkUnderflowNonZero(Node u, int i) {
  Node w = bs.readBlock(u.children[i]);  // w is child of u
  if (w.size() < B-1) {  // underflow at w
    Node v = bs.readBlock(u.children[i-1]); // v left of w
```

```
    if (v.size() > B) {  // w can borrow from v
      shiftLR(u, i-1, v, w);
    } else { // v will absorb w
      merge(u, i-1, v, w);
    }
  }
}
void checkUnderflowZero(Node u, int i) {
  Node w = bs.readBlock(u.children[i]); // w is child of u
  if (w.size() < B-1) {  // underflow at w
    Node v = bs.readBlock(u.children[i+1]); // v right of w
    if (v.size() > B) { // w can borrow from v
      shiftRL(u, i, v, w);
    } else { // w will absorb w
      merge(u, i, w, v);
      u.children[i] = w.id;
    }
  }
}
```

まとめると、B 木における remove(x) は根から葉まである経路を辿り、x′ を葉 u から削除し、その後 0 回以上の併合を u とその祖先に対して実行し、高々一回借用操作を行う。併合や借用には 3 つのノードしか修正せず、$O(\log_B n)$ 回だけの操作を行うので、外部メモリモデルにおける全体としての実行時間は $O(\log_B n)$ である。しかしここでも、ワード RAM モデルでは併合やノードを借りる操作には $O(B)$ だけの時間がかかるので、ワード RAM モデルにおける remove(x) の実行時間は $O(B\log_B n)$ であるとしか（現時点では）言えない。

14.2.4 *B* 木の償却解析

ここまでに、次のことを示してきた。

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)・remove(x) の実行時間は $O(B\log n)$ である。

次の補題は、これまで B 木における分割・併合操作の数を過大評価していたことを示す。

補題 14.1. 空の B 木からはじめて、add(x)・remove(x) からなる m 個の操作の列を順に実行するとき、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**.** $B = 2$ の特別な場合について、9.3 節で既に証明の概要を示している。この補題はここでは出納法で証明する。

1. 分割・併合・借用の際に 2 の預金を支払う（失う）。
2. add(x) または remove(x) の際には、最大 3 の預金が得られる。

最大で $3m$ の預金が得られており、各分割・併合・借用は預金を 2 支払うので、最大 $3m/2$ 回の分割・併合・借用が実行される。 図 14.5、図 14.9、図 14.10 では預金は ¢ で表した。

預金の値を追うために、証明では次の**預金不変条件** (credit invariant) を保つ。$B-1$ 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。あとは、各 add(x)・remove(x) の間に、預金不変条件を保つことと、上で説明した性質 1・2 を満たすこととを示す。

追加の場合　add(x) は併合や借用を行わないため、分割だけを考えれば十分である。

既に $2B-1$ 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 はふたつのノード u′ と u″ に分割され、それぞれは $B-1$ 個、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、あと 1 は u′（$B-1$ 個のキーを持つ）に渡されるので、預金不変条件が保たれる。よって、いかなる分割においても、その分割のための預金を支払える上に、預金不変条件を保てる。

add(x) を実行するとき、ノードに対する他の修正は、すべての分割処理を終えたあと実行される。これは新たなキーをあるノード u′ に追加する処理である。

直前に u′ が $2B-2$ 個の子を持っていれば、子の数は $2B-1$ になるので、預

金を 3 得ることになる。これは高々 add(x) によって得られることになっている預金の範囲内である。

削除の場合 remove(x) の際には、0 回以上の併合と、それに続く借用が一度発生するかもしれない。併合のシナリオとしては、ふたつの v と w がともに $B-1$ 個のキーを remove(x) を呼ぶ直前に持っており、これらが $2B-2$ 個のキーをもつひとつのノードに併合されるというものである。そのため、併合のために 2 の預金が支払われている。

併合のあとには、高々一度の借用処理があり、その後にはもう併合も借用も発生しない。この借用が起きるのは、$B-1$ 個のキーを持つ葉 v からキーを削除する場合に限る。このとき v は預金を 1 持っており、この預金は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 足らず、支払いを完了するために預金をあと 1 必要である。

ここまでで、預金を 1 得ており、預金不変条件が保たれていることを示す必要がある。最悪の場合には 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 も w 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。よってこの場合には v と w とに渡すための追加の預金を 2 作る必要がある。借用は remove(x) の処理の間に高々一回発生するので、場合に応じて最大で 3 の預金が作られることがわかった。

もし remove(x) において借用がないなら、これはあるノードでキーを削除して終了したためであり、このノードは操作の前には B 個以上のキーを持っていたことになる。最悪の場合には、このノードがちょうど B 個のキーを持っており、そのため操作の後では $B-1$ 個のキーを持ち、預金を 1 作って与えなければならない。

削除が借用で終わる場合、そうでない場合のいずれにおいても、預金不変条件を保ち、併合と借用のコストを支払うためには、高々 3 の預金を remove(x) の間に作る必要がある。以上より定理が示された。 □

補題 14.1 の目的は、ワード RAM モデルにおいて、add(x)・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・借用にかかる時間は合わせて $O(Bm)$ であることを示すことであった。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)・remove(x) の償却コストは $O(B+\log \mathtt{n})$ である。この結果を次のふた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 $O(\log_B \mathtt{n})$ 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 $O(\log_B \mathtt{n})$ である。さらに、空の *BTree* 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、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[4] である。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。

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が B 木である。B 木は McCreight [9] が 1970 年に提案した。10 年を待たずして Comer のサーベイ（論文のタイトルが"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 木は本当にいたるところで使われていて、多くのファイルシステムにおける基本的なデータ構造である。例えば、Apple の HFS+、Microsoft の NTFS、Linux の Ext4 などの例がある。また、すべてのメジャーなデータベースシステムもそうである。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いくつも例がある。Graefe の近年のサーベイ [36] では 200 ページ以上にわたって現代の応用やデータ構造の変種、B の最適化などが述べられている。

B は SSet インターフェースを実装する。もし USet インターフェースだけが必要なときには、外部メモリハッシュ法を B 木の代わりに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。この手法では、外部メモリモデルにおいて $O(1)$ の期待実行時間で USet の操作を実行できる。しかし、いくつかの理由で、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 木を使う。

B 木が人気がある理由のひとつに、$O(\log_B \mathtt{n})$ という実行時間の上界から受ける印象より実際には性能がよいことがしばしばあることを挙げられる。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の検索はすべて非常に高速に処理でき、一回だけの外部メモリアクセスで葉が得られる。

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.** この章のはじめに、B 木の内部メモリとして必要なのは $O(B+\log_B \mathsf{n})$ だけであると言った。しかし、ここで示した実装では実はより多くのメモリが必要であった。

1. この章で示した add(x)・remove(x) の実装は $B\log_B \mathsf{n}$ に比例する内部メモリを使うことを示せ。
2. これを $O(B+\log_B \mathsf{n})$ 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、預金不変条件を保てることを確認せよ。

問 **14.6.** B 木を修正し、ノードの持つ子の数が B 以上 $3B$ 以下（そのため、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 B 木では、m 回の操作を順に実行する間に、$O(m/B)$ 回だけの分割・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、併合処理をもっと頑張らなければならない。必ずしも必要でないときにも、併合を行わなければならないことがある。）

問 **14.7.** この練習問題では、B 木の分割・併合を修正し、最大 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. u を一杯になったノード、v を u のすぐ右の兄弟とする。u のノード溢れを解消する方法は二通りある。
   (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 と新しいノード w とで分け合う。

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おり、v と w はの兄弟であるとする。u のアンダーフローを解消する方法は二通りある。
   (a) キーを u・v・w の間で分配しなおす。
   (b) u・v・w を併合し、ふたつのノードにする。それぞれの持っていたキーはふたつのノードに分配しなおす。

   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・分割の回数は $O(m/B)$ になることを示せ。

問 **14.8.** 図 14.11 に示した B^+ 木は、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。葉はそれぞれ、ふつう $B-1$ 個以上 $2B-1$ 個以下のキーを格納する。葉から上側はふつうの B 木で、最後のもの以外の各葉の最大値を内部ノードは蓄えている。

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 の値が大きい時、この重複が大して問題とならない理由を説明せよ。

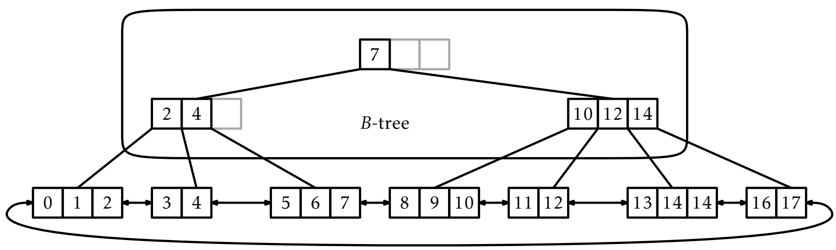


図 14.11: B^+ 木は、ブロックの双方向連結リストの上に B 木が乗ったデータ構造である。

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] Oracle. *Java Platform Standard Ed. 6*. URL: http://download.oracle.com/javase/6/docs/api/ [cited 2011-07-19].

[56] Oracle. *The Java Tutorials*. URL: http://download.oracle.com/

javase/tutorial/ [cited 2011-07-19].

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

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

[59] M. Pătraşcu and M. Thorup. Randomization does not help searching predecessors. In N. Bansal, K. Pruhs, and C. Stein, editors, *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7–9, 2007*, pages 555–564. SIAM, 2007.

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

[61] W. Pugh. A skip list cookbook. Technical report, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, 1989. URL: ftp://ftp.cs.umd.edu/pub/skipLists/cookbook.pdf [cited 2011-07-20].

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

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

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

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

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

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

[68] H. H. Seward. Information sorting in the application of electronic digital computers to business operations. Master's thesis, Massachusetts Institute of Technology, Digital Computer Laboratory, 1954.

[69] Z. Shao, J. H. Reppy, and A. W. Appel. Unrolling lists. In *Proceedings of the 1994 ACM conference LISP and Functional Programming (LFP'94)*, pages 185–195, New York, 1994. ACM.

[70] P. Sinha. A memory-efficient doubly linked list. *Linux Journal*, 129,

2005. URL: http://www.linuxjournal.com/article/6828 [cited 2013-06-05].

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

[72] D. Sleator and R. Tarjan. Self-adjusting binary trees. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25–27 April, 1983, Boston, Massachusetts, USA*, pages 235–245. ACM, ACM, 1983.

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

[74] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977.

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

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

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

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

索引

日本語索引