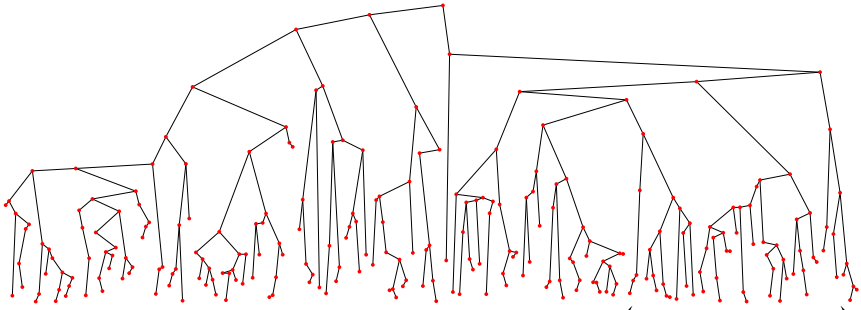


Open Data Structures (in Java)

Edition 0.1G

Pat Morin

目次

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。

なぜこの本を読むのか

データ構造の入門書は多くある。非常に良いものもある。大半は無料ではなく、コンピュータサイエンスの学部生はデータ構造の本にきっとお金を払うだろう。

オンラインで無料で入手できるデータ構造の本もある。非常に良いものもあるが、大部分は古くなっている。著者や出版社が更新をやめるときに無料になったものが大部分である。これらは通常、次の 2 つの理由から更新できない。(1) 著作権は著者または出版社に属し、いずれかの許可が得られないため。(2) 書籍のソースコードが利用できないため。つまり、本の Word、WordPerfect、FrameMaker、または LaTeX ソースコードが利用できなかったり、そのソースを扱うソフトウェアのバージョンが利用できなかったりするため。

このプロジェクトはコンピュータサイエンスの学部生がデータ構造の入門書代を支払わなくてよくすることを目指す。この目標を達成するため、この本をオープンソースソフトウェアプロジェクトのように扱うことにした。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 [*1] あるいは信頼できるソースコード管理サイト [*2] からダウンロードできる。

ソースコードは Creative Commons Attribution ライセンスで公開されている。つまり誰でも自由にコピー、配布、送信してよい。取り込んで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。このとき唯一の条件は *attribution* です。つまり派生した作品が `opendatastructures.org` のコードやテキストが含むことを認める必要がある。

誰でもソースコード管理システム `git` を使って手を加えられる。誰でも本のソースをフォークして別バージョンを作れる（例えば別のプログラミング

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

言語版)。私の望みは、私のやる気や興味が衰えた後も、この本が有用であり続けることだ。

第 1

Introduction

データ構造とアルゴリズムに関するコースは世界の全てのコンピュータサイエンスカリキュラムに含まれている。データ構造はそれほどに重要だ。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？立ち止まって考えてみると、私達は普段から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、ファイルをディスク上に配置し、検索できる。これは簡単ではない。ディスクには数億ものブロックがある。ファイルの内容はそのどこかに保存されるのだ。
- 連絡先を検索する：ダイヤル途中の部分的な情報にもとづき、連絡先リストから電話番号を見つけるためにデータ構造が使われる。これは簡単ではない。連絡先リストに含まれる情報はとても多いかもしれない。これまで電話や電子メールで連絡したことのある全ての人を想像してみてほしい。また、電話のプロセッサはあまり高性能でなく、メモリも潤沢でない。
- SNS にログインする：ネットワークサーバーは、ログイン情報からアカウント情報を検索する。これは簡単ではない。人気のソーシャルネットワークには何億人ものアクティブユーザーがいる。
- Web ページを検索する：検索エンジンは検索語を含む Web ページを見つけるためにデータ構造を使う。これは簡単ではない。インターネットには 85 億以上の Web ページがあり、個々のページには検索されるかもしれない単語が多く含まれている。

- 緊急サービス（9-1-1）に電話する：緊急サービスネットワークはパトカー・救急車・消防車が速やかに現場に到着できるよう、電話番号と住所を対応づけるデータ構造を使う。これは重要だ。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、遅れは生死を別つかもしれない。

効率の必要性

次節では多くのデータ構造がサポートする操作を見ていく。ちょっとしたプログラミング経験があれば、これらを正しく実装することは難しくない。データを配列または線形リストに格納し、配列または連結リストの全要素を見てまわり、要素を追加したり削除したりすればよいのだ。

こういう実装は簡単だが、あまり効率的ではない。さて、このことを考える価値はあるだろうか？コンピュータはどんどん高速化している。自明な実装で十分かもしれない。確認のためにざっくりとした計算をしてみよう。

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）のアイテムを持つアプリケーションがあるとする。各アイテムを最低一回は参照すると仮定してよいことが多いだろう。つまり少なくとも 100 万（10^6）回はこのデータから検索をしたいわけだ。この 10^6 回の検索それぞれが 10^6 個のアイテムをすべて確認すると、合計 $10^6 \times 10^6 = 10^{12}$（1 兆）回の確認処理が必要だ。

プロセッサの速度： 本書の執筆の時点で、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）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。コンピューターの時間では 16 分は非常に長いが、人は気にしないかもしれない。（例えば、プログラマがコーヒーブレイクに向かうならそれで構わないだろう。）

大きなデータセット： Google を考えてみよう。Google では 85 億もの Web ページからの検索を扱う。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。だが実際にはそんなに時間がかからないこと

[*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。

を知っているだろう。Web 検索には 8.5 秒もかからないし、あるページがインデックスに含まれているかよりさらに複雑な問い合わせも実行する。執筆時点で Google は 1 秒間に約 4,500 クエリを受けつける。つまり少なくとも $4,500 \times 8.5 = 38,250$ ものサーバーが必要なのだ。

解決策： 以上の例から、アイテムの数 n と実行される操作数 m が共に大きくなると、データ構造の自明な実装はスケールしないことがわかる。今の例で、（機械命令数で数えた）時間はおよそ $n \times m$ だ。

解決策はもちろん、データ構造内のデータを上手に並べ、各操作がすべてのデータを見て回る必要がないようにすることだ。一見そんなことはムリなように思えるが、データ構造に格納されているデータの数とは関係なく、平均して 2 つのデータだけを参照すればすむデータ構造をのちに紹介する。 1 秒あたり 10 億命令を実行できるとして、10 億個のデータ（兆、京、垓であっても）を含むデータ構造を検索するのにわずか 0.000000002 秒しかかからないのだ。

データ構造内のデータを整列することで、データ数に対して参照されるデータ数が非常にゆっくり増加するデータ構造も後で紹介する。例えば 10 億個のアイテムを整列しておけば、最大 60 個のアイテムを参照することで各操作を実行できる。毎秒 10 億命令実行できるコンピュータでは、これらの操作は 0.00000006 秒しかかからない。

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。Section **??**は本書で説明するデータ構造で実装されるインターフェースを全て説明するので読む必要があると考えて欲しい。残りの節では、以下のものを解説する。

- 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

これらの背景知識があってもなくても、いったん気軽に飛ばし必要に応じて戻り読みしてもよい。

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実装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どうやるかを示す。

インターフェース（抽象データ型と呼ばれることもある）は、データ構造がサポートする操作一式とその意味を定義する。インターフェースを見ても操作がどう実装されているかはわからない。サポートする操作の一覧とその引数、返り値の特徴だけを教えてくれる。

一方でデータ構造の実装には、データ構造の内部表現と、操作を実装する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考えられる。例えば本書では Chapter 2 では配列を、Chapter 3 ではポインタを使った `List` インターフェースの実装を紹介する。それぞれ同じ `List` インターフェースを実装しているが、実装方法は異なるのだ。

`Queue`、`Stack`、`Deque` インターフェース

`Queue` インターフェースは要素を追加したり、次の要素を削除したりできる要素の集まりを表す。より正確には、`Queue` インターフェースがサポートする操作は以下のものだ。

- `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()` 操作の引数はないことに注意する。`Queue` は取り出し規則に従って次に削除する要素を決める。取り出し規則は色々と考えられるが、主なものとしては FIFO や優先度、LIFO などがある。

Figure 1.1 に示す *FIFO*（*first-in-first-out*、先入れ先出し）*Queue* は、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で並ぶ列と同じようなものだ。これは最も一般的な `Queue` なので、修飾子の FIFO は省略されることが多い。他の教科書では FIFO `Queue` の `add(x)`、`remove()` は、それぞれ `enqueue(x)`、`dequeue()` と呼ばれていることも多い。

Figure 1.2 に示す優先度付き `Queue` は、`Queue` から最小の要素を削除する。同点要素が複数あるときは、そのうちのいずれかを任意に選ぶ。これは病院の救急室で重症患者を優先的に治療することに似ている。患者が到着したら、

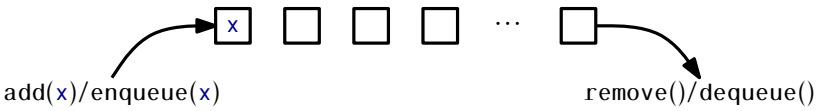


図 1.1: FIFO Queue.

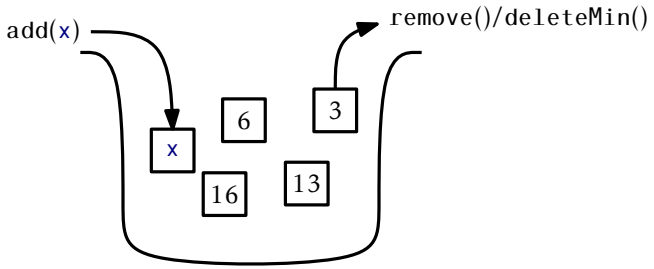


図 1.2: A priority Queue.

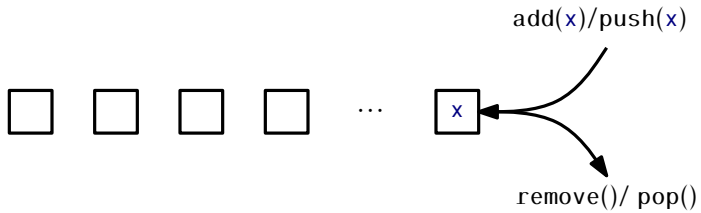


図 1.3: スタック

症状を見積もってから待合室で待機していてもらう。医師の手が空くと、最も重篤な患者から治療する。優先度付き Queue における remove() 操作を、他の教科書ではよく deleteMin() などと呼んでいる。

よく使う取り出し規則は、Figure 1.3 に示す LIFO（last-in-first-out、後入れ先出し）だ。 *LIFO* キューでは、最後に追加された要素が次に削除される。これは皿を積むように視覚化するとよい。皿はスタックの上に積まれ、上から順に持って行かれる。この構造はとてもよく見かけるので Stack という名前が付いている。Stack について話すとき、add(x)、remove() のことを、push(x)、pop() と呼ぶ。こうすれば LIFO と FIFO の取り出し規則を区別できる。

Deque（双方向キュー）は FIFO キューと LIFO キュー（スタック）の一般化だ。 Deque は先頭と末尾のある要素の列を表し、列の先頭または末

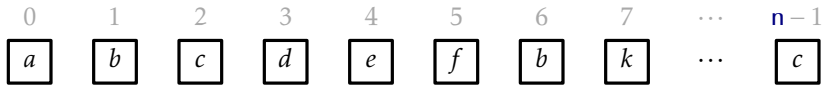


図 1.4: List は 0,1,2,...,n − 1 で添え字づけられた列を表現する。この List で get(2) を実行すると値 c が返ってくる。

尾に要素を追加できる。 Deque 操作の名前はわかりやすく、addFirst(x)、removeFirst()、addLast(x)、 removeLast() だ。スタックは addFirst() と removeFirst() だけを使えば実装できることは知っておくと良いだろう。一方 FIFO キューには addLast(x) と removeFirst() を使えばよい。

List インターフェース：線形シーケンス

この本では、FIFO Queue や Stack、Deque のインターフェースについての話はほとんどしない。なぜなら、これらのインターフェースは List インターフェースにまとめられるためだ。図 1.4 に示す List は、値の列 $x_0,\ldots,x_{n-1}$ を表現する。List インターフェースは以下の操作を含む。

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 番目として追加し、$x_i,\ldots,x_{n-1}$ をずらす。
   すなわち、$j \in \{n-1,\ldots,i\}$ 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。

これらの操作を使うことで Deque インターフェースが実装できる。

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

この後の章では Stack、Deque、FIFO Queue の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 は、List インターフェース

を実装するデータ構造として出てくることがある。その場合、Stack や Deque のインターフェースは非常に効率良く実装できる。たとえば ArrayDeque クラスは List インターフェースの実装だ。これはすべての Deque 操作をひとつだけの定数時間操作で実装できる。

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。これは数学における集合を模したものだ。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、要素の並び順は決まっていない。USet は以下の操作をサポートする。

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

今述べた定義で、探したい x と、見つかるかもしれない集合の要素 y の区別を小難しく感じるかもしれない。これを区別する理由は、x と y は実は異なるオブジェクトだが等しいと判定されるかもしれないためだ。異なるオブジェクトを等しいと判定するインターフェースは、キーを値に対応づける辞書 (マップ) を作るのに便利なのだ。

辞書 (マップ) を作るために、まず Pair という複合オブジェクトを作る。Pair はキーと値からなる。 2 つの Pair は、キーが等しいければ等しいとみなされる。Pair である (k,v) を USet に入れてから、x = (k,null) として find(x) を呼び出すと、y = (k,v) が返ってくる。すなわち、キー k だけから値 v を復元できるのだ。

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 は全順序な要素を格納するので、任意の 2 つの要素 x と y は比較可能である。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。

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

SSET は USet と全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。
   $y \geq x$ を満たす最小の要素 y を探す。このような y が存在すればそれを返し、そうでないなら null を返す。

この find(x) は、後継探索 *(XXX:訳語)* 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点で USet.find(x) とは異なる。

USet、SSet における find(x) の区別は重要なのだが見過ごされがちである。SSet は追加の仕事をしてくれるぶん、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) の実装では、要素数の対数だけの時間がかかる。一方、Chapter 5 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である。SSet の追加機能が本当に必要でないなら、SSet ではなく常に USet を使うべきだ。

数学的背景

この節では本書で使用する数学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、確率論などだ。内容はあっさりしたもので、丁寧な手解きはしない。背景知識が足りないと感じる読者のために、コンピュータサイエンスのための数学の優れた無料の教科書がある。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].

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、やはり（後述する）指数関数 e^x の観点から、べき乗を定義できる。指数関数もまた指数級数を使って定義されているが、こういうな話は微積分の教科書に任せることにする。

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、

$$b^x = k \ .$$

この本に出てくる対数の底はほとんどの場合 2 である。底が 2 の対数を二進対数という。そのため、底になにも書かない $\log k$ は $\log_2 k$ の省略記法とする。

対数の大雑把だが分かりやすいイメージを紹介する。$\log_b k$ とは k を何回 b で割ると 1 より小さくなるかを表す数だと考えればよい。例を挙げよう。二分探索という手法を使うと、一回の比較処理のたびに、答えの候補の個数が半分になる。答えの候補が 1 つに絞られるまで、この処理を繰り返す。$n+1$ 個の答えの候補が最初にあるなら、二分検索に必要な比較の回数は $\lceil \log_2(n+1) \rceil$ 以下だ。

この本で自然対数という別の対数も何度か出てくる。$\ln k$ と書いて $\log_e k$ を表すことにする。ここで、e は次のように定義されるオイラーの定数だ。

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

自然対数は頻繁に現れる。これは e がよく見かける次のような積分値であるためだ。

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

よく使う対数の操作は 2 つある。ひとつは指数部からの取り出し操作だ。

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

もう一つは対数の底を取り替え操作だ。

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

これら 2 つの操作を使えば、例えば自然対数と二進対数を比較できる。

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

階乗

この本で階乗関数を使う箇所がいくつかある。n が非負整数のとき、$n!$（「n の階乗」と読む）は次のように定義される。

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

$n!$ は n 要素の相異なる順列の個数である。つまり n 個の相異なる要素の並べ方の数として階乗は現れる。なお、$n = 0$ のときについて、$0!$ は 1 と定義される。

$n!$ の大きさは、スターリングの近似によって近似的に求められる。

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

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。

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

（スターリングの近似を証明する簡単な方法として、$\ln(n!) = \ln 1 + \ln 2 + \cdots + \ln n$ を $\int_1^n \ln n \, \mathrm{d}n = n\ln n - n + 1$ で近似するというものがある。）

二項係数は階乗関数とつながりがある。n を非負整数、k を $\{0,\ldots,n\}$ の要素とするとき、$\binom{n}{k}$ 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。すなわち、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である。

漸近記法

データ構造を分析する際には操作の実行時間についての議論が有用だ。正確な実行時間はコンピュータごとに異なる。同じコンピュータで実行する場合ですらバラつくだろう。ここで操作の実行時間とは操作に必要なコンピュータ命令数を指す。単純なコードであっても、この量を正確に計算するのは難しいことがある。そのため実行時間を正確に解析するのではなく、ビッグオー記法と呼ばれる記法を使う。$f(n)$ を関数とするとき、$O(f(n))$ は次のような関数の集合を表す。

$$O(f(n)) = \left\{ \begin{array}{c} g(n) : \text{there exists } c > 0, \text{ and } n_0 \text{ such that} \\ g(n) \leq c \cdot f(n) \text{ for all } n \geq n_0 \end{array} \right\} .$$

グラフ上で考えると、n が十分に大きくなると $c \cdot f(n)$ が $g(n)$ よりも大きくなるか同じになる関数 $g(n)$ を集めたものがこの集合だ。

漸近表記を使えば関数を単純化できる。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。

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

$c = 13$ 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。

漸近表記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ります。

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

これは有名な記号の濫用なのだが、$f_1(n) = O(f(n))$ と書いて $f_1(n) \in O(f(n))$ であることを表す。また、「この操作の実行時間は $O(f(n))$ 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。これらの短い言い方は語感を整え、漸近記法を連続する等式の中で使いやすくするのに役立つ。

この書き方の、奇妙な例を挙げる。

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

これは正確に書くとこうなる。

$$T(n) \leq 2\log n + [\text{some member of } O(1)]$$

$O(1)$ 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。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 $f(n) = 1$ であるものと読み取ることになる。

ビッグオー記法は新しいものでも、コンピュータサイエンス独自のものでもない。1894 年には数学者 Paul Bachmann が使っていた。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のに非常に便利なことが判明したのだ。次のコードを考えてみましょう。

Simple

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

このメソッドを 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]`)
- 関節代入 `n` 回 (`a[i] = i`)

よって実行時間は以下のようになる。

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

a、b、c、d、e はプログラムを実行するマシンに依存する定数で、それぞれ代入、比較、インクリメント、配列のオフセット計算、間接代入の実行時間を

表す。しかしたった 2 行のコードの実行時間を表す式がこうだと、より複雑なコードやアルゴリズムをこのやり方では扱えないだろう。ビッグオー記法を使うと、実行時間は次のようにより単純になる。

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

この書き方はよりコンパクトながらさっきの式と同じくらいのことを教えてくれる。実行時間は定数 a、b、c、d、e に依存している。これらの値がわからないと、実行時間はわからず比較できないのだ。これらの定数を明らかにするため努力しても（例えば実際に時間を測ってみる）、得られる結論はそのマシンにおいてのみ有効なだけだ。

ビッグオー記法を使うと、より高次の分析をすることが出来るのでより複雑な関数も扱うことが出来る。 2 つのアルゴリズムのビッグオー記法での実行時間が同じなら、どちらが速いかわからず、はっきりとした勝ち負けがつかないかもしれない。あるマシンでは一方が速く、別のマシンでは他方が速いかもしれない。しかし 2 つのアルゴリズムのビッグオー記法での実行時間が異なることを示せれば、実行時間が小さい方は n が十分大ければ速いとわかる。

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 Figure 1.5 示す。これは $f_1(\mathrm{n}) = 15\mathrm{n}$ と $f_2(n) = 2\mathrm{n}\log\mathrm{n}$ の増加を比べたものだ。$f_1(\mathrm{n})$ は複雑な線形時間アルゴリズムの実行時間、$f_2(\mathrm{n})$ 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だ。これを見ると、n が小さいうちは $f_1(\mathrm{n})$ はより大きいが $f_2(\mathrm{n})$、n が大きくなると逆転することがわかる。そして、最終的には $f_1(\mathrm{n})$ が圧倒的に性能がよくなるのだ。ビッグオー記法を使った解析で $O(\mathrm{n}) \subset O(\mathrm{n}\log\mathrm{n})$ となることから、このことを知ることができる。

複数の変数を持つ関数に対して漸近表記を使用する場合もある。標準的な定義は定まっていないようだが、この本では次の定義を用いる。

$$O(f(n_1,\ldots,n_k)) = \left\{ \begin{array}{c} g(n_1,\ldots,n_k) : \text{there exists } c>0, \text{ and } z \text{ such that} \\ g(n_1,\ldots,n_k) \le c\cdot f(n_1,\ldots,n_k) \\ \text{for all } n_1,\ldots,n_k \text{ such that } g(n_1,\ldots,n_k) \ge z \end{array} \right\} \ .$$

この定義を使えば我々が考えたいことを表現することが出来る。引数 $n_1,\ldots,n_k$ が g を大きくするときのことだ。この定義は $f(n)$ が n の増加関数なら一変数の場合の $O(f(n))$ の定義と同じだ。我々の目的のためにはこの定義でよいのだが、多変数の場合の漸近記法を異なる定義を与えている教科書もあることには注意が必要だ。

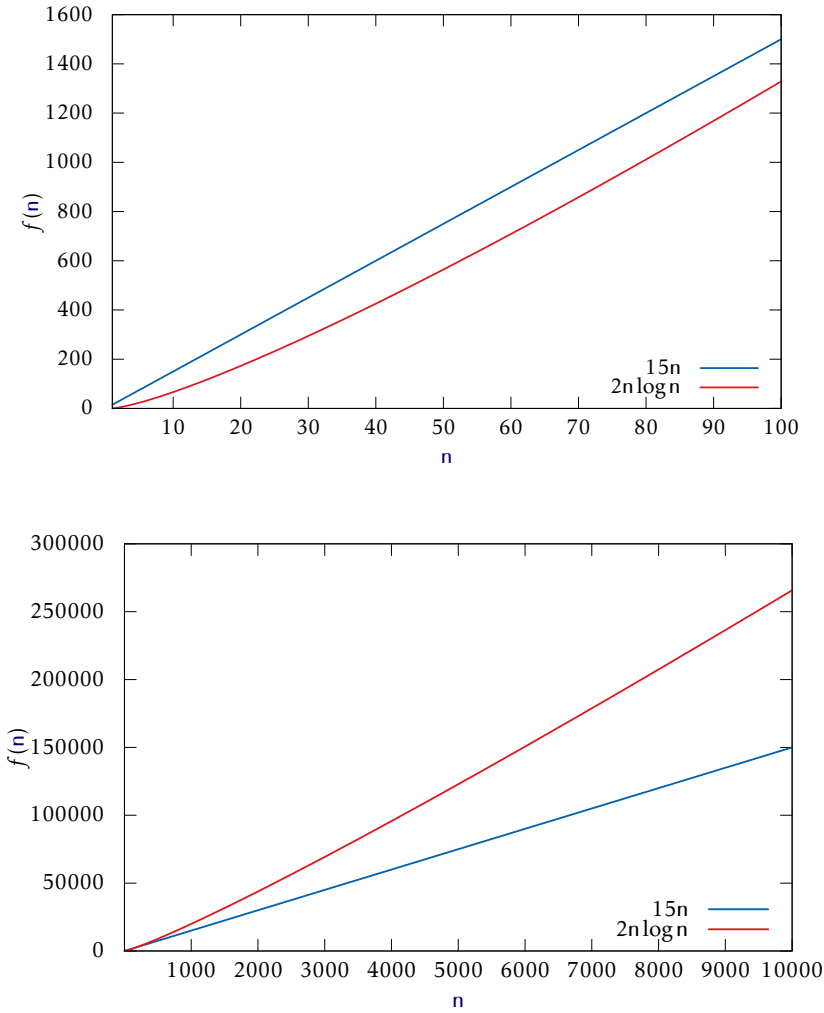


図 1.5: 15n 対 2nlogn のプロット

ランダム性と確率

この本で扱うデータ構造にはランダム性を利用するものがある。格納されているデータや実行された操作だけでなく、サイコロの目もふまえて動作を決めるのだ。そのため同じことをしても実行時間は毎回同じであるとは限らない。こういうデータ構造を分析するときは平均または期待実行時間を考えるのがよい。

形式的には、ランダム性を利用するデータ構造における操作の実行時間は確率変数である。そしてその期待値を知りたい。全事象 U の値をとる離散確率変数を X とするとき、X の期待値 $E[X]$ は以下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、データ構造の内部で発生するランダム性のみを考慮して確率を定める。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だという仮定は置かないことに注意する。

期待値の最も重要な性質の一つとして期待値の線形性がある。任意のふたつの確率変数 X と Y について以下の関係が成り立つ。

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

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

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

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

インジケータ確率変数はよく使う便利なトリックだ。この二値変数はなにかを数えたいときに役立つ。例を見るとよくわかるだろう。表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。直感的な答えは $k/2$ だが、期待値の定義を使って証明しようとすると次のようになる。

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

この計算は $\Pr\{X = i\} = \binom{k}{i}/2^k$ および 2 項係数の性質 $i\binom{k}{i} = k\binom{k-1}{i}$ や $\sum_{i=0}^{k}\binom{k}{i} = 2^k$ を知っていないとできない。

インジケータ変数と期待値の線形性を使えばはるかに簡単になる。$\{1,\ldots,k\}$ の各 i に対し以下のインジケータの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、以下の計算を行う。

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

ここで、$X = \sum_{i=1}^{k} I_i$ なので以下のように所望の値を得られる。

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

この計算は少し長いものの、不思議な等式や非自明な確率計算は必要ない。各コイントスは 1/2 の確率で表が出るので結果はたぶんコイン数の半分だ、という直感の説明でもある。

計算モデル

本書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。これを正確に行うための計算の数学的なモデルが必要だ。そのために w ビットのワード *RAM* モデルを使うことにする。RAM はランダムアクセスマシン (Random Access Machine) の頭字語である。このモデルではランダムアクセスメモリを使える。ランダムアクセスメモリはセルの集まりで、これはそれぞれ w ビットのワードを格納できる。つまり、各セルは $\{0,\ldots,2^w-1\}$ の中のひとつを表せる。

ワード RAM モデルではワードの基本的な操作に一定の時間が必要である。基本的な操作は算術演算（$+, -, *, /, \%$）や比較（$<, >, =, \leq, \geq$）、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 EXOR）である。

どのセルも定数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る。メモリ管理システムは必要に応じてメモリブロックを割り当てたり割り当て解除したりしてくれる。サイズ k のメモリブロックの割当てには $O(k)$ の時間がかかり、新しく割り当てられたメモリブロックへの参照（ポインタ）が返される。この参照はひとつのワードで表せる程度小さい。

ワード幅 w はこのモデルの重要なパラメータである。この本で w に置く仮定は、n をデータ構造に格納されうる要素数とするとき、$w \geq \log n$ であるということだけだ。これは控えめな仮定である。なぜならこれが成り立たないとひとつのワードではデータ構造の要素数を数えることすらできないためである。

領域はワード単位で測るので、データ構造で使う領域の広さとはメモリの使用するワード数のことである。我々のデータ構造はみなジェネリック型 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。

この本に載っているデータ構造は、実装できないような特殊なトリックを使ってはいない。

正しさ、時間複雑性、空間複雑性

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。

時間複雑性： データ構造における操作の実行時間は短いほどよい。

空間複雑性： 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正しさを満たすデータ構造しか扱わない。つまり、不正確な出力が得られることがあったり、更新をちゃんとしなかったりするデータ構造のことは考えない。一方で、メモリ使用量を最小限に抑えるための工夫をしているデータ構造は紹介する。これは操作の（漸近的な）実行時間には影響しないことが多いが、実用上ではデータ構造を少し遅くするかもしれない。

データ構造の実行時間を考えるとき、3 つの異なる実行時間保証を扱うことがよくある。

最悪実行時間： これは最も強力な実行時間の保証である。操作の最悪実行時間が $f(n)$ ならば、操作の実行時間は決して $f(n)$ よりも長いことはない。

償却実行時間： 償却実行時間が $f(n)$ ならば、典型的な操作のコストが $f(n)$ であることを意味する。より正確には、m 個の操作の列が $mf(n)$ であることを意味する。いくつかの操作には、個別では $f(n)$ よりも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ると、ひとつあたりのコストは $f(n)$ 以下なのである。

期待実行時間： 期待実行時間が $f(n)$ ならば、実際の実行時間は確率変数（Section 1.3.4 を参照）であり、この確率変数の期待値が $f(n)$ である。ここでいうランダム性はデータ構造が行う選択のランダム性を指す。

最悪、償却、期待実行時間の違いを理解するのには、お金の例え話が役に立つ。家を買う費用のことを考えてみよう。

最悪コストと償却コスト 家の価格が 120000 ドルだとする。毎月 1200 ドルの 120 ヶ月（10 年）の住宅ローンでこの家が手に入るかもしれない。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 120000 ドルの一括払いで家を買うこともできる。こうするとこの家を購入代金を 10 年で償却した月額費用は以下のようになる。

$$\$120\,000/120 \text{ months} = \$1\,000 \text{ per month} \ .$$

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 120000 ドルの家における火災保険を考えてみよう。保険会社はたくさん事例を調べた結果、この家における火災のリス

クは月額 10 ドル相当だと判断した。ほとんどの家庭では火災が発生せず、ごく一部の家庭がボヤを経験し、全焼してしまう家の数はもっともっと少ない。この情報から保険会社は火災保険の料金を月に 15 ドルにした。

さて、どうしよう。最悪でも月額 15 ドル火災保険費用を支払うべきだろうか、それとも月額 10 ドルの自家保険を積み立てるべきだろうか。明らかに 1 か月あたり 10 ドルの費用が期待値では安いが、最悪の場合ではコストがはるかに高くなる。万一全焼すれば 120000 ドル支払うことになる。

この例から、最悪でなく償却あるいは期待実行時間を選ぶことがある理由もわかるだろう。償却・期待実行時間は最悪実行時間よりも小さいことが多い。償却・期待実行時間を使うことにすれば、はるかに単純なデータ構造採用できる場合がよくあるのだ。

コードサンプル

XXX: Ruby の話を書くことになると思うので、見直す必要があるように思う。

この本のコードサンプルは Ruby で書いた。しかし、Ruby に親しみのない人も読めるようシンプルに書いたつもりだ。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 などを ALGOL 系の言語書いたことのある人は本書のコードを見て意味がわかるだろう。完全な実装に興味のある読者はこの本に付属の Ruby ソースコードを見てほしい。

この本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 Ruby のコードとを共に含む。そのためソースコードと数式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。一番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ものとする。

データ構造の一覧

表 1.1 と表 1.2 は本書で扱うデータ構造における性能の要約である。これらは Section **??**で説明した `List` や `USet`、`SSet` を実装する。Figure 1.6 はこの本の各章の依存関係を示している。破線の矢印は弱い依存関係を示している。これは章のごく小さいが依存や、一部の結果のみに依存することを示す。

TODO: 表の翻訳

List implementations			
	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 implementations			
	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] Denotes an *amortized* running time.

[E] Denotes an *expected* running time.

表 1.1: Summary of List and USet implementations.

ディスカッションと練習問題

Section **??**で説明した List・USet・SSet 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 の List・Set・Map・SortedSet・SortedMap をシンプルにしたのである。

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[50] の素晴らしい（そしてフリーの）本が扱っている。分かりやすい微積分の教科書としては、無料で手に入る Thompson[71] の古典的な教科書がある。この本では指数や対数の形式的な

SSet implementations			
	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 implementations			
	findMin()	add(x)/remove()	
BinaryHeap	$O(1)$	$O(\log n)^{A}$	§ 10.1
MeldableHeap	$O(1)$	$O(\log n)^{E}$	§ 10.2

[I] This structure can only store w-bit integer data.

[X] This denotes the running time in the external-memory model; see Chapter 14.

表 1.2: Summary of SSet and priority Queue implementations.

定義が書かれている。

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

Exercise 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があれば、それを使って解いてみてほしい。

XXX: Ruby の場合の話を書く

以下の問題はテキストの入力を一行ずつ読み、各行で適切なデータ構造の操

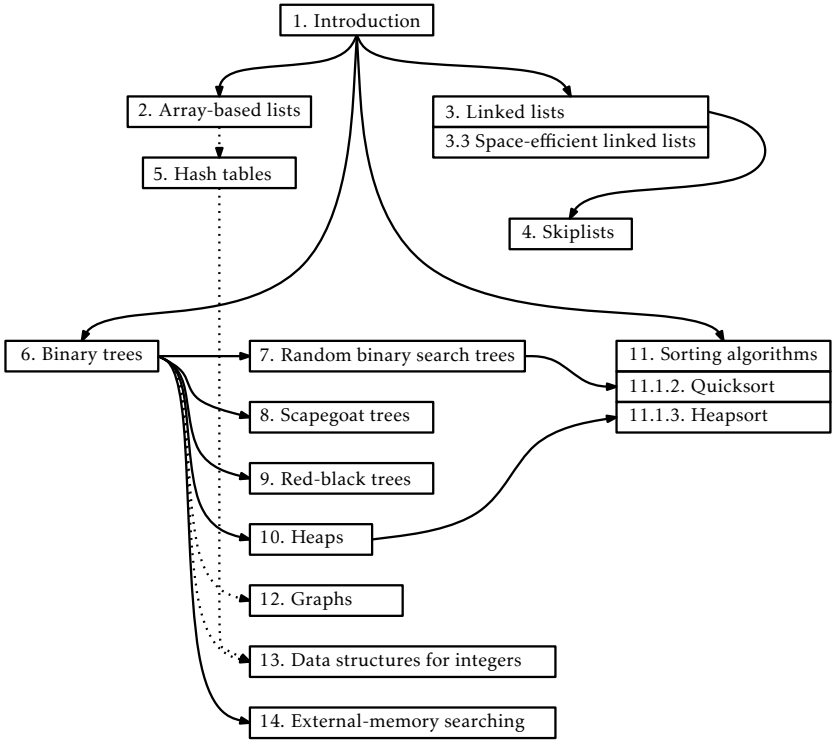


図 1.6: The dependencies between chapters in this book.

作を実行することで解いてほしい。ただし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効率的な実装でなければならないものとする。

1. 入力を一行ずつ読み、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、最後から二番目の入力行を二番目に書き出す、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無くなるまで繰り返し、最後に残っていた行（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 番目の行からはじまり、49、48、...、1 番目の行が続く。この次は 100 番目の行で、99、...、51 番目の行が続く。
   またプログラム実行中に 50 より多くの行を保持してはならない。

3. 入力を一行ずつ読み取り、42 行目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 行目が空行であれば、200 行目を出力せよ。またプログラム実行中に 43 行以上の行を保持してはならない。
4. 入力を一行ずつ読み取り、もしこれまでと重複のない行を見つけたら出力せよ。重複がたくさんあるファイルを読む場合にも、重複なく行を保持するのに必要なメモリより多くメモリを使わないように注意せよ。
5. 入力を一行ずつ読み取り、それがこれまでに読んだことのある行と同じなら出力せよ。（最終的には、入力の中のはじめて現れた行を除いたものが得られる。）重複がたくさんあるファイルを読む場合にも、重複なく行を保持するのに必要なメモリより多くメモリを使わないように注意せよ。
6. 入力を全て読み取り、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、それらの行の順序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、重複する行は現れた回数だけ出力するように変更した問題を解け。
8. 入力をすべて読み、全ての偶数番目の行を出力した後に全ての奇数番目の行を出力せよ。（なお、最初の行を 0 行目と数える。）
9. 入力をすべて読み、ランダムに並び替えて出力せよ。どの行の内容も書き換えてはならない。また、入力とくらべて行を減らしたり増やしたりしてもいけない。

Exercise 1.2. *Dyck word* とは +1, -1 からなる列で、先頭から任意の k 番目の値までの部分列（プレフィックス）の和がいずれも非負であるものである。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)・pop() 操作の関係を説明せよ。

Exercise 1.3. マッチした文字列とは {, }, (,), [,] のからなる列で、すべての括弧が適切に対応しているものである。例えば、「{{()[]}}」はマッチした文字列だが、「{{()]}」はふたつめの { 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を $O(n)$ で判定するにはスタックをどう使えばよいかを説明せよ。

Exercise 1.4. push(x)・pop() 操作のみが可能なスタック s が与えられる。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。

Exercise 1.5. USet を使って Bag を実装せよ。Bag とは USet みたいなものである。Bag は add(x)・remove(x)・find(x) 操作をサポートするが、重複する要素も格納するところが異なる。Bag の find(x) 操作は 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ひとつを返す。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。

Exercise 1.6. List・USet・SSet インターフェースのゼロから実装せよ。必ずしも効率的な実装でなくてもよい。ここで実装するものは、後の章で出てくるより効率的な実装の正しさや性能をテストするために役立つ。（最も簡単な方法は要素を配列に入れておく方法だ。）

Exercise 1.7. 直前の問題の実装の性能をアップするための思いつく工夫をいくつか試みよ。実験してみて、List の add(i,x)・remove(i) の性能がどう向上したか考察せよ。USet・SSet の find(x) の性能はどうすれば向上しそうか考えてみよ。この問題はインターフェースの効率的な実装がどの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、配列にデータを格納することによってList・Queue インターフェースを実装する方法について解説する。*backing array*. 次の表は、この章で説明するデータ構造の操作時間を要約したものだ。

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

データをひとつの配列に入れるデータ構造には以下のような利点・欠点がある。

- 配列の任意の要素には一定の時間でアクセスできる。そのため get(i)・set(i,x) はいずれも定数時間で実行される。
- 配列は動的ではない。リストの真ん中付近に要素を追加・削除するためには、隙間を作ったり埋めたりするために多くの要素が移動することになる。add(i,x)・remove(i) 操作の実行時間が n・i に依存するのはこのためだ。
- 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、新しい配列を割当て、古い配列の要素をそちらにコピーする必要がある。この操作のコストは大きい。

3 つめの点は重要だ。上記の表に記載された実行時間は backing array の拡大・縮小にかかるコストは含まれていない。後述するように、注意深く設

計すれば、backing array の拡大・縮小のコストを加味しても平均的な実行時間はほとんど増えない。より正確に言うと、空のデータ構造からはじめて、add(i,x)・remove(i) を m 回実行するときの、backing array の拡大・縮小のための合計コストは $O(m)$ である。個々の操作のコストは大きいが、m 個の操作にわたる償却コストを考えれば、ひとつの操作あたりのコストは $O(1)$ なのだ。

ArrayStack：配列を使った高速なスタック操作

ArrayStack は *backing array*a を使ったリストインターフェースの実装だ。リストの i 番目の要素を a[i] とする。ほとんどの場合 a は実際に必要な値よりも大きい。そのため整数 n によって実際に a に入っている要素数を表す。つまり、リストの要素は a[0],...,a[n − 1] に格納される。また、関係 a.length ≥ n が常に成り立つ。

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

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である。必要に応じて境界チェックをしたあと単に a[i] を返すか、a[i] を書き換えるかすればよいのだ。

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 Figure 2.1 に示す。add(i,x) では、まず a が既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() の実装方法については後述する。resize() の直後では a.length > n であることだけ知っていれば今は十分である。あとは x が入るように a[i],...,a[n − 1] をひとつずつ右に移動させ、a[i] を x にし、n を 1 増やせばよい。

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() が呼ばれるかもしれないが、こ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、この操作のコストは（resize のことを無視すれば）$O(n-i)$ である。

remove(i) の実装も似ている。a[i + 1],...,a[n − 1] を左にひとつシフトし（a[i] は書き換えられる）、n の値をひとつ小さくする。その後 n が a.length より小さすぎないか、具体的には a.length ≥ 3n を確認する。もしそうなら、resize() を呼んで a を小さくする。

ArrayStack

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

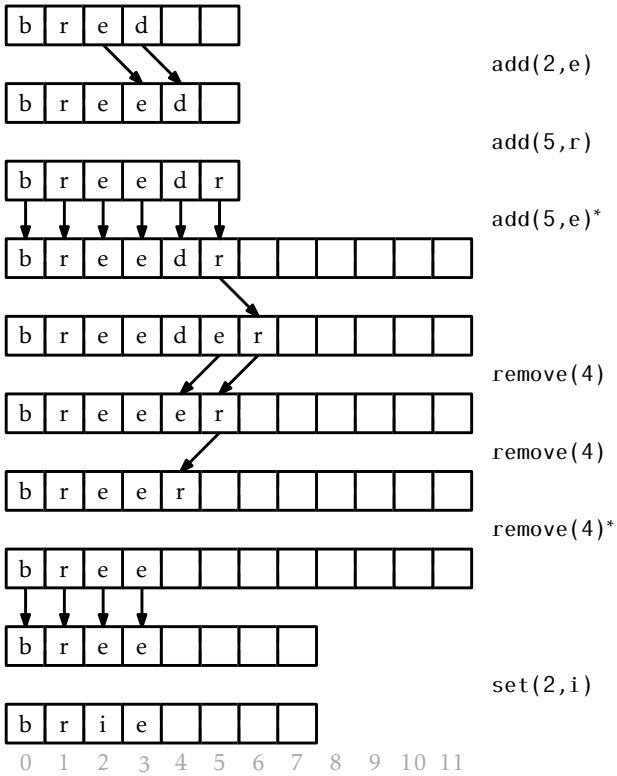


図 2.1: add(i,x)・remove(i) を ArrayStack に実行する。矢印は要素のコピーを表す。resize() を呼ぶ操作にはアスタリスクを付した。

```
  n--;
  if (a.length >= 3*n) resize();
  return x;
}
```

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

拡張と収縮

resize() の実装は単純だ。大きさ 2n の新しい配列 b を割当て、n 個の a の要素を b のはじめの 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。

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 を割当て、n 要素をコピーする。これは $O(n)$ の時間がかかる。

前節からの実行時間分析は resize() のコストを無視していた。この節では償却解析として知られる方法でこれを解決する。 この手法は個々の add(i,x)・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)・remove(i) からなる m 個の操作の列の間に呼ばれる resize() の計算コストの合計を考える。

次の補題を示す。

Lemma 2.1. 空の *ArrayStack* が作られたあと、$m \geq 1$ 個の add(i,x)・remove(i) からなる操作の列が順に実行されるとき、この間に呼ばれる resize() の合計実行時間は O(m) である。

Proof. resize() が呼ばれるとき、その前の resize() 呼び出しから add・remove が実行された回数は $n/2-1$ 以下である。i 回目の resize() 呼び出しの際の n を n_i とする。r を resize() の呼び出し回数とする。このとき、add(i,x) と remove(i) の合計呼び出し回数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。

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

$r \leq m$ より resize() 呼び出しのための実行時間の合計は次のようになる。

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

あとは $(i-1)$ から i 回目の resize() の間に add(i,x) か remove(i) が呼ばれる回数が $\mathtt{n}_i/2$ 以下であることを示す。

2 つの場合が考えられる。ひとつは resize() が add(i,x) に呼ばれる場合で、これは backing array が一杯になるとき、つまり a.length = n = $\mathtt{n}_i$ が成り立つときだ。この一つ前に行った resize() 操作について考えよう。この resize() の直後、a の大きさは a.length だが a の要素数は a.length/2 = $\mathtt{n}_i/2$ 以下であった。しかし a の要素数は今では $\mathtt{n}_i$ = a.length なのだから、前の resize() から $\mathtt{n}_i/2$ 回以上は add(i,x) が呼ばれたことがわかる。

もうひとつ考えられるのは、resize() が remove(i) に呼ばれる場合で、このとき a.length $\geq 3\mathtt{n} = 3\mathtt{n}_i$ である。前の resize() の直後では a の要素数は a.length/2 − 1 以下であった。[*1] 今 a には $\mathtt{n}_i \leq$ 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。

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

いずれの場合でも、$(i-1)$ 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。 □

要約

次の定理は ArrayStack の性能を整理するものだ。

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

Theorem 2.1. *ArrayStack* 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、*ArrayStack* 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 $O(1+n-i)$ である。

空の *ArrayStack* から任意の m 個の add(i,x)・remove(i) 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。

ArrayStack は Stack を実装する効率的な方法である。特に push(x) は add(n,x)、pop() は remove(n−1) のようにそれぞれ実装できる。またいずれの操作も償却実行時間 $O(1)$ である。

FastArrayStack：最適化された ArrayStack

ArrayStack が主にやっていることは、データの（add(i,x) と remove(i) のための）シフトと（resize() のための）コピーである。

素朴な実装では for ループを使うだろう。しかしデータのコピーや移動用の効率的な機能があるプログラミング環境も多いだろう。C 言語には memcpy(d,s,n)・memmove(d,s,n) 関数がある。C++ 言語には 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。

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;
}
```

これらは最適化されており、for ループを使うよりかなり速くコピーができる特殊な機械命令を使うかもしれない。これらを使っても漸近的な実行時間は減らないのだが、やる価値のある最適化ではある。

C++ や Java の実装で高速な配列コピーを使って 2〜3 倍の高速化できたこともある。どのくらい速くなるかは環境によるので是非試してみてほしい。

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。(add(x) によって) 要素が追加されたのと同じ順番で、(remove() によって）キューから要素が削除される。

FIFO キューの実装に ArrayStack はあまり向いていない。一方の端から要素を追加し他方から要素を削除することになるので、賢明な選択ではないのだ。2 つの操作のうち一方はリストの先頭を変更することになる。つまり、$i = 0$ で add(i,x) か remove(i) を呼びだす。このとき、n に比例する実行時間がかかってしまう。

配列を使ったキューの効率的な実装は、もし無限の配列 a があれば簡単だろう。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。

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

まず i, j 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 をひとつ増やす。要素を削除するときは、$a[j]$ から要素を取り出し、j をひとつ増やし、n をひとつ減らす。

もちろんこの方法の問題点は無限の配列が必要になることだ。ArrayQueue

は有限の配列 a と剰余算術で無限配列を模倣する。剰余算術は時間の計算をするときに使っているものだ。例えば 10:00 に 5 時間を足すと 3:00 になる。形式的に書けば次のようになる。

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

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 は次のように二項演算と考えてもよい。

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

より一般的には整数 a と正整数 m について、ある整数 k が存在し $a=r+km$ をみたす $\{0,\ldots,m-1\}$ の一意な要素を $a \bmod m$ と書く。雑に言うと r とは a を m で割った余りである。C や C++、Ruby、Java など多くのプログラミング言語では mod 演算子は % で表される。

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

a[j%a.length],a[(j+1)%a.length],...,a[(j+n−1)%a.length]

これは a を循環配列として使っている。配列の添字が a.length − 1 を超えると、配列の先頭に戻ってくるのである。

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)・remove() からなる操作の列を実行する様子を Figure 2.2 に示す。add(x) はまず a が一杯かどうかを確認し、必要に応じて resize() を呼んで a の容量を増やす。続いて x を a[(j+n)%a.length] に入れて、n をひとつ増やせばよい。

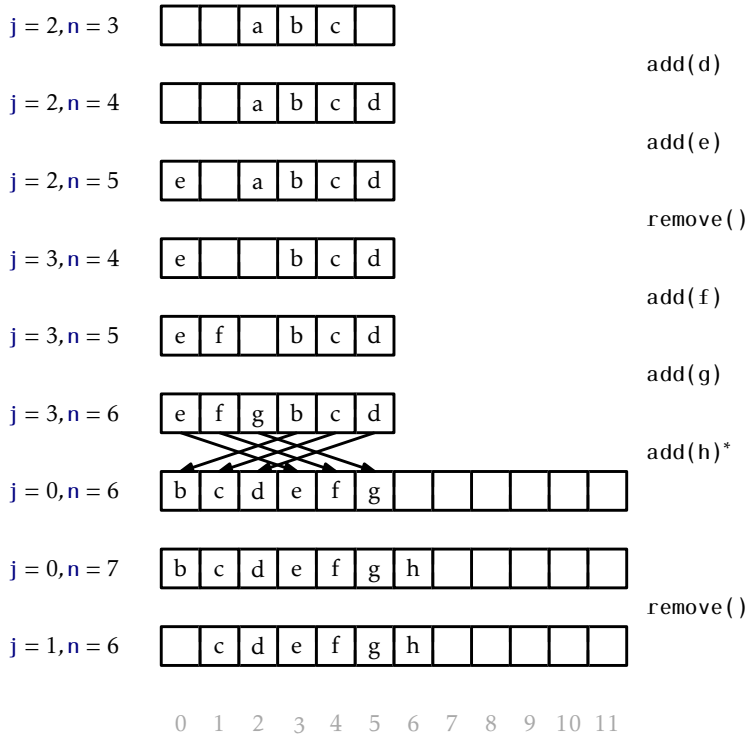


図 2.2: A sequence of `add(x)` and `remove(i)` operations on an `ArrayQueue`. Arrows denote elements being copied. Operations that result in a call to `resize()` are marked with an asterisk.

ArrayQueue

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

最後になるが、`resize()` 操作は `ArrayStack` のものとよく似ている。大き

さ 2n の新しい配列 b を割当て、

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

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;
}
```

要約

次の定理は ArrayQueue の性能を整理するものだ。

Theorem 2.2. *ArrayQueue は (FIFO)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() *を $O(1)$ の時間で実行できる。さらに、空の ArrayStack 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() *に使われる実行時間の合計は $O(m)$ である。*

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が他方の端から削除が効率的にできる列を表現するデータ構造だった。ArrayDeque は両方の端で効率的な追加と削除ができるデータ構造である。ArrayQueue を表現するために使った循環配列をここでもまた使って List インタフェースを実装する。

ArrayDeque

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

ArrayDeque における get(i) と set(i,x) の実装は難しくない。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) の実装はもう少し興味深い。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。つづいて、$i < n/2$ かどうかを確認する。もしそうなら、a[0],...,a[i − 1] をそれぞれひとつずつ左にずらす。そうでないなら、a[i],...,a[n − 1] をそれぞれひとつずつ右にずらす。add(i,x) と remove(x) の説明として Figure 2.3 を見てほしい。

ArrayDeque

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

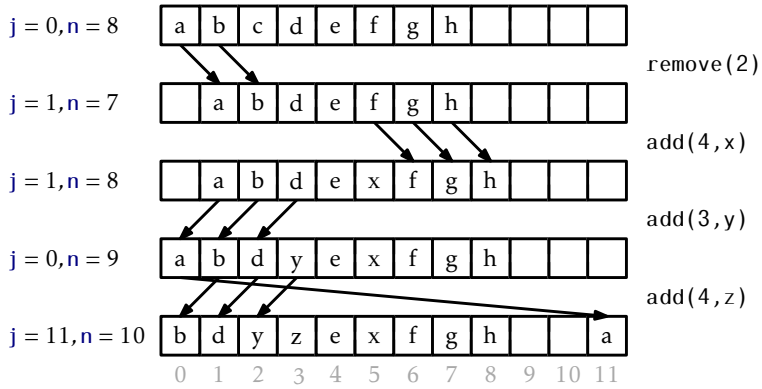


図 2.3: A sequence of add(i,x) and remove(i) operations on an ArrayDeque. Arrows denote elements being copied.

```
    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++;
}
```

このようにシフトを行うことで、add(i,x) は $\min\{i, n-i\}$ より多くの要素をシフトしなくてよい。よって 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) の実装も似たようなものだ。a[0],...,a[i−1] をそれぞれ右にひとつずつシフトするか、a[i+1],...,a[n−1] をそれぞれ左にひとつずつシフトするか、i < n/2 かどうかに応じてどちらかを行う。やはり remove(i) も $O(1+\min\{i, n-i\})$ だけの時間で要素を動かし終えることができる。

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;
}
```

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。

空の ArrayDeque から任意の m 個の 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。

DualArrayDeque：2 つのスタックから作った双方向キュー

次は 2 つの ArrayStack を使って ArrayDeque に近い性能を示すデータ構造 DualArrayDeque を紹介する。DualArrayDeque の漸近的な性能は ArrayDeque より優れているわけではないのだが、2 つのシンプルなデータ構造を組み合わせてより高度なデータ構造を作る良い例なのでここで学ぶ価値がある。

DualArrayDeque は、2 つの ArrayStack を使ってリストを表現する。ArrayStack では終端付近の要素を高速に修正できたことを思い出してほしい。DualArrayDeque は front と back という名のふたつの ArrayStack を

後ろ合わせに配置する。そのため両端での高速な操作が可能だ。

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() によって求めることが出来るからだ。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、逆さまの順番で格納する。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格納する。こうして、front か back に対する get(i) か set(i,x) を適切に呼び出すことで、get(i)・set(i,x) を $O(1)$ の実行時間で実現できる。

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);
```

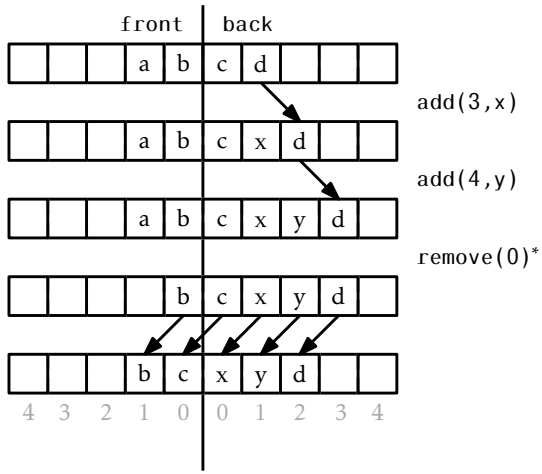


図 2.4: A sequence of add(i,x) and remove(i) operations on a DualArrayDeque. Arrows denote elements being copied. Operations that result in a rebalancing by balance() are marked with an asterisk.

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

front には逆順に要素を蓄えているので、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 Figure 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) はふたつの ArrayStack front・back のバランスを調整するために balance() を呼び出す。balance() の実装は後で説明するが、size() < 2 であるときを除いて front.size() と back.size() は三倍以上離れないことを balance() は保証することを知っておけば十分である。具体的には 3·front.size() ≥ back.size() と 3·back.size() ≥ front.size() であることを保証する。

つづいて add(i,x) のうち balance() のコストを無視したコストを求める。i < front.size() のとき add(i,x) は front.add(front.size() − i − 1,x) で実装できる。front は ArrayStack なのでこのコストは次のようになる。

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

一方 i ≥ front.size() のとき add(i,x) は back.add(i − front.size(),x) で実装できる。このコストは次のようになる。

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

i < n/4 のときはひとつめのケース (2.1) に該当する。i ≥ 3n/4 のときはふたつめのケース (2.2) に該当する。n/4 ≤ i < 3n/4 のときは、front と back どちらを操作するかわからない。しかし i ≥ n/4 かつ n − i > n/4 なのでいずれ場合も実行時間は $O(n) = O(i) = O(n-i)$ である。以上をまとめると次のようになる。

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

ゆえに add(i,x) の実行時間は balance() の呼び出しのことを無視すれば $O(1+\min\{i,n-i\})$ である。

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;
}
```

バランスの調整

最後に add(i,x) と remove(i) によって実行される balance() の説明をする。この操作は front・back のどちらも大きく（または小さく）なりすぎないことを保証するものだ。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 ⌊n/2⌋・⌈n/2⌉ 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 $O(n)$ 個の要素を動かすので $O(n)$ の時間がかかる。balance() は add(i,x)・remove(i) で毎回呼ばれるのでこれは一見好ましくない。しかし次の補題より、balance() のための平均時間は定数であることがわかる。

Lemma 2.2. 空の *DualArrayDeque* に対して $m \geq 1$ 個の add(i,x)・remove(i) からなる操作の列を順に実行する。このとき全ての balance() の呼び出しの実行時間の合計は $O(m)$ である。

Proof. balance() が要素を動かすとき、前に balance() が要素を動かしたときから呼ばれた add(i,x)・remove(i) の合計数は $n/2-1$ 以下であることを示す。Lemma 2.1 の証明と同様に、これを示せば balance() の合計時間が $O(m)$ であることを示すには十分である。

ここではポテンシャル法. として知られる手法を解析に用いる。DualArrayDeque のポテンシャル Φ を front と back の要素数の差と定義する。

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

このポテンシャルの興味深い性質は、バランス調整を行わない add(i,x)・remove(i) の呼び出しはポテンシャルを 1 増やすことだ。

次の式が成り立つことから、balance() が要素を動かした直後にはポテンシャル Φ_0 は 1 以下であることがわかるだろう。

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

要素を動かす balance() の直前には $3\texttt{front.size()} < \texttt{back.size()}$ であっ

たと仮定して一般性を失わない。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() によって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。 □

要約

次の定理は DualArrayDeque の性質をまとめるものだ。

Theorem 2.3. *DualArrayDeque* は *List* インターフェースを実装する。resize()・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 $O(1 + \min\{\mathtt{i}, \mathtt{n} - \mathtt{i}\})$ である。

空の *DualArrayDeque* から任意の m 個の add(i,x)・remove(i) 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。*Furthermore, beginning with an empty DualArrayDeque, any sequence of m* add(i,x) *and* remove(i) *operations results in a total of $O(m)$ time spent during all calls to* resize() *and* balance().

- 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 $O(1 + \min\{\mathtt{i}, \mathtt{n} - \mathtt{i}\})$ である。

空の DualArrayDeque から任意の m 個の add(i,x)・remove(i) からなる操作の列を実行する。このとき resize()・balance() にかかる時間の合計は $O(m)$ である。

2.6 RootishArrayStack：空間効率に優れた配列スタック

ここまで紹介してきたデータ構造には共通の欠点がある。データは 1 つか 2 つの配列に入れ、配列のサイズを変更しないようにしているので、配列に隙間が多い傾向がある点だ。例えば resize() 直後の ArrayStack では配列 a は半分しか埋まっていない。a は 3 分の 1 しか埋まっていないことさえある。

この節ではこの無駄なスペースの問題を解決する RootishArrayStack というデータ構造を紹介する。RootishArrayStack は n 個の要素を $O(\sqrt{n})$ 個の配列に格納する。この配列ではデータが格納されていない場所は常に $O(\sqrt{n})$ 以下である。 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 はブロックと呼ばれる r 個の配列に要素を格納する。この配列は 0,1,...,r − 1 と添字付けられる。Figure 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。

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

この式が成り立つのは Figure 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内に順番に配置される。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 に格納される。ここで問題になるのは、全体で 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をどう知るかである。

i に対応する要素がどのブロックに入っているかさえ分かれば、ブロック内の位置は簡単に計算できる。 インデックス i の要素が b 番目のブロックに入っているなら、0,...,b−1 番目のブロックにおける要素数の合計は b(b+1)/2

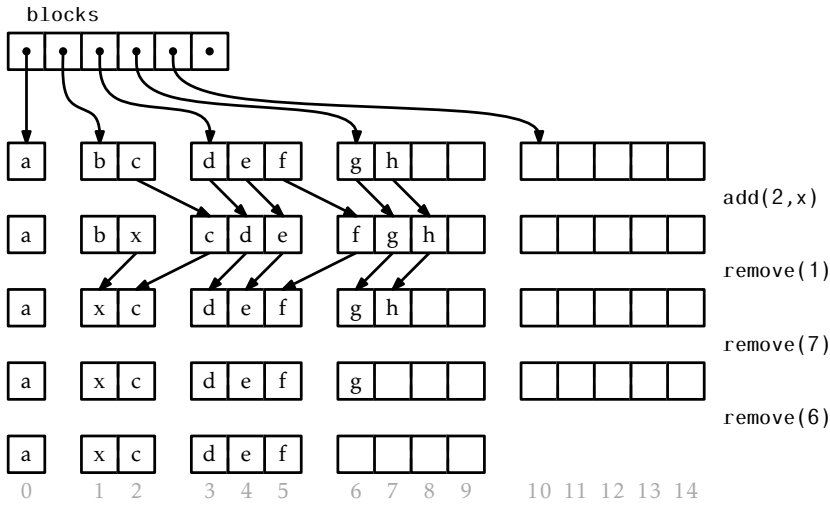


図 2.5: A sequence of `add(i,x)` and `remove(i)` operations on a `RootishArrayStack`. Arrows denote elements being copied.

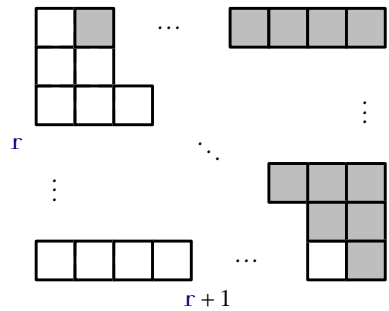


図 2.6: The number of white squares is $1 + 2 + 3 + \cdots + r$. The number of shaded squares is the same. Together the white and shaded squares make a rectangle consisting of $r(r+1)$ squares.

である。そのため、i は

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

として b 番目のブロックの j 番目の位置に入っている。b を求めること、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難しい。i 以下のインデックスを持つ要素は $i+1$ 個ある。一方で、$0,\ldots,b$ 番目のブロックに入っている要素数の合計は $(b+1)(b+2)/2$ である。よって、b は次の式を満たす最小の整数である。

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

この式は次のように変形できる。

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

対応する 2 次方程式 $b^2 + 3b - 2i = 0$ はふたつの解 $b = (-3+\sqrt{9+8i})/2$ と $b = (-3-\sqrt{9+8i})/2$ 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 $b = (-3+\sqrt{9+8i})/2$ 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 $b \geq (-3+\sqrt{9+8i})/2$ を満たす最小の b が欲しかったのであった。これは次のように書ける。

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

RootishArrayStack

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

話を変えるが、get(i) と set(i,x) は簡単に実装できる。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を使って blocks リストを表現すれば、get(i) も set(i,x) も定数時間で実行できる。

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,...,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() のコストを無視すると、add(i,x) の操作はシフト操作のコストを考えれば十分で、これは $O(1+n-i)$ である。これは ArrayStack と同じだ。

remove(i) は add(i,x) に似ている。$i+1,\ldots,n$ 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) のコストはシフトのコストを考えれば十分で、これは $O(n-i)$ である。

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() は shrink() 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。

領域使用量

次に、RootishArrayStack が使用する余分な領域の量を分析する。RootishArrayStack が使用する領域のうち、リストの要素を保持していないものを数えたい。これを無駄な領域ということにする。

XXX: 以下、原著怪しい (?) ので確認

remove(i) は RootishArrayStack のうち一杯でないブロックは 2 つまでである。よって n 個の要素を含む RootishArrayStack が用意するブロック数を 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。

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

末尾のブロックふたつの大きさは r と $r-1$ なので、これらのブロックによって生じる無駄な領域の量は $2r-1 = O(\sqrt{n})$ 以下である。もしこれらのブロックを（例えば）ArrayStack に入れれば、r 個のブロックを入れる List による無駄な領域の量も $O(r) = O(\sqrt{n})$ である。n 個の要素と関連情報を保持するのに必要なその他の領域は $O(1)$ である。以上より、RootishArrayStack の無駄な領域の量は合計 $O(\sqrt{n})$ 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、n 個の要素を追加する際にはど

こかのタイミングで（ほんの一瞬かもしれないが）$\sqrt{n}$ 以上の無駄な領域が生じることを示す。

空のデータ構造に n 個の要素をひとつずつ追加していくとする。完了したときには、r 個のブロックに分散して n 個のアイテムがデータ構造に入っている。$r \geq \sqrt{n}$ なら、r 個のブロックを追跡するために r 個のポインタ（参照）を使い、ポインタは無駄な領域である。一方で $r < \sqrt{n}$ なら鳩の巣原理より大きさ $n/r > \sqrt{n}$ 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、$\sqrt{n}$ の無駄な領域が生じている。以上より、n 個の要素を挿入するまでのあるタイミングでデータ構造は $\sqrt{n}$ の無駄な領域を生じることが示された。

要約

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

Theorem 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(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 $O(1 + n - i)$ である。

空の *RootishArrayStack* から任意の m 個の add(i,x)・remove(i) からなる操作の列を実行する。このときすべての grow()・shrink() にかかる時間の合計は $O(m)$ である。

要素数 n の *RootishArrayStack* が使う（ワード単位で測った）使用領域量 [*2] は $n + O(\sqrt{n})$ である。

Computing Square Roots

XXX: Pseudo-code edition では無い節だが、Ruby edition では書くか?

[*2] Section 1.4 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。

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伝承のようなものだ。30 年以上前の実装さえ見つかる。例えば、ここで扱った ArrayStack・ArrayQueue・ArrayDeque の実装を簡単に一般化できるスタック・キュー・双方向キューが Knuth [46, Section 2.2.2] により議論されている。

恐らく Brodnik *et al.* [13] が RootishArrayStack を記述し、Section 2.6.2 で述べたような下界 $\sqrt{n}$ を示した最初の文献である。彼らは他の洗練されたブロックサイズの選び方も示しており、これは i2b(i) の中で冪根の計算をせずに済むものだ。このやり方では i 番目の要素を含むブロックは $\lfloor \log(i+1) \rfloor$ 番目のもので、これは単に $i+1$ の二進表現における最高位の桁である。この計算をするための命令を提供するコンピュータ・アーキテクチャもある。

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がある。この構造体は get(i,x)・set(i,x) を定数時間で実行できる。add(i,x)・remove(i) の実行時間は $O(\sqrt{n})$ である。これと同じような実行時間は Exercise 2.10 で扱う RootishArrayStack のより練られた実装によっても達成できる。

Exercise 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 c = {x} とした特殊な場合である。) この章で説明したデータ構造において 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。

Exercise 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。

Exercise 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。

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

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。

Exercise 2.4. rotate(a,r) 操作を実装せよ。配列 a を「回転」する、すなわち i ∈ {0,...,a.length} のすべてについて a[i] を a[(i+r) mod a.length] に動かすものだ。

Exercise 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 (i+r) mod 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 $O(1+\min\{r,n-r\})$ でなければならないとする。

Exercise 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。

Exercise 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。

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

なお & はビット単位の and 演算オペレータである。

Exercise 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 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。

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

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

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

Exercise 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 $O(n^{2/3})$ 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 $O(n^{1/3})$ 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が、次は配列ではなくポインタを使ったデータ構造の話をする。この章のデータ構造は、リストの要素を含むノードから構成される。ノード間は参照（ポインタ）によって繋げられ、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。

連結リストを使って List インターフェースの実装するのは、配列を使う場合と比べて長所・短所がある。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のが主な短所だ。配列と違い、i 番目の要素までリストをひとつずつ辿らなければならないのである。連結リストの主な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を定数時間で実行できる。これが u がリストの中のどのノードであっても成り立つのだ。

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。

SLList

```
class Node {
  T x;
```

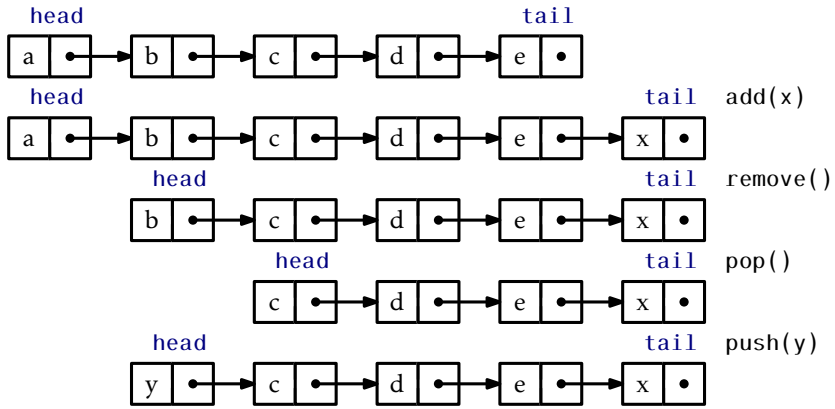


図 3.1: A sequence of Queue (add(x) and remove()) and Stack (push(x) and pop()) operations on an SLList.

```
  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 Figure 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。

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、代わりに tail = head = null とする必要がある。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。

要約

次の定理は SLList の性能を整理したものである。

Theorem 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 個のノードを辿っていかなければならないのである。

DLList: 双方向連結リスト

DLList（doubly-linked list、双方向連結リスト）は SLList によく似ている。違いがあるのは、DLList ではノード u が直後のノード u.next への参照と直前のノード u.prev への参照の両方を持っている点だ。

DLList

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

SLList を実装するときにはいくつか特別な処理があった。 例えば SLList の最後のノードを削除したり、空の SLList にノードを追加するときは head・tail を適切に更新するため特別な処理が必要であった。DLList ではこういう特別な場合というのがかなり増える。DLList におけるこれら全ての特別な場合を綺麗に扱う最善の方法はおそらくダミーノードを使うことだろう。ダミーノードはデータを含まない空のノードで、これを置くことで特別なノードが生じなくなるのだ。 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、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 Figure 3.2 に示すようなサイクルになる。

DLList

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

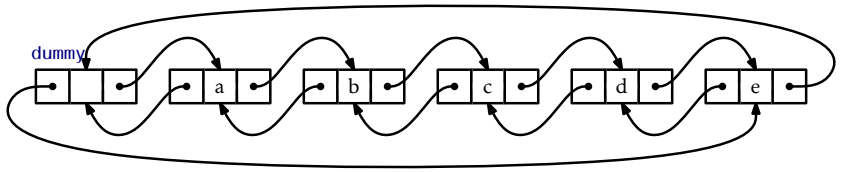


図 3.2: A DLList containing a,b,c,d,e.

```
  dummy.next = dummy;
  dummy.prev = dummy;
  n = 0;
}
```

DLList で番号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 $O(1+\min\{i,n-i\})$ である。

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、u.next = w、u.prev = w.prev とし、u.prev.next・u.next.prev を適切に調整すればよい。（Figure 3.3 を参照せよ。）ダミーノードがあるので w.prev・w.next がない場合を気にする必要はない。

DLList

```
Node addBefore(Node w, T x) {
  Node u = new Node();
  u.x = x;
  u.prev = w.prev;
  u.next = w;
  u.next.prev = u;
  u.prev.next = u;
  n++;
```

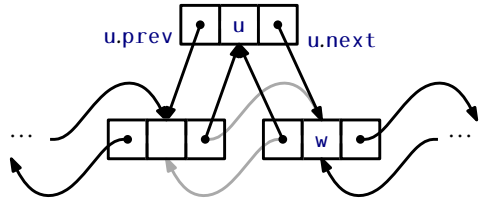


図 3.3: Adding the node u before the node w in a DLList.

```
  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) の実行時間のうち定数でないのは（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。

要約

次の定理は DLList の性能をまとめたものである。

Theorem 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 Chapter 2 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。

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 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 Section 2.4 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;
```

XXX: rubyimport

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 $b-1$ 以上 $b+1$ 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。

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

末尾以外の各ブロックの BDeque は $b+1$ 以下の要素を含むので各配列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。b を $\sqrt{n}$ の定数倍にすれば、SEList の無駄な領域を Section 2.6.2 で導出した下界に等しくすることができる。

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。唯一の違うの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);
  }
}
```

最大でひとつのブロックを除いて、すべてのブロックの要素数は $b-1$ 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 $b-1$ 以上ずつ近づいていく。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 i によって決まる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、実行時間は $O(1+\min\{i,n-i\}/b)$ である。

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存在し、一杯でないので、x をそのブロックの末尾に追加できる。

SEList
```
boolean add(T x) {
  Node last = dummy.prev;
  if (last == dummy || last.d.size() == b+1) {
```

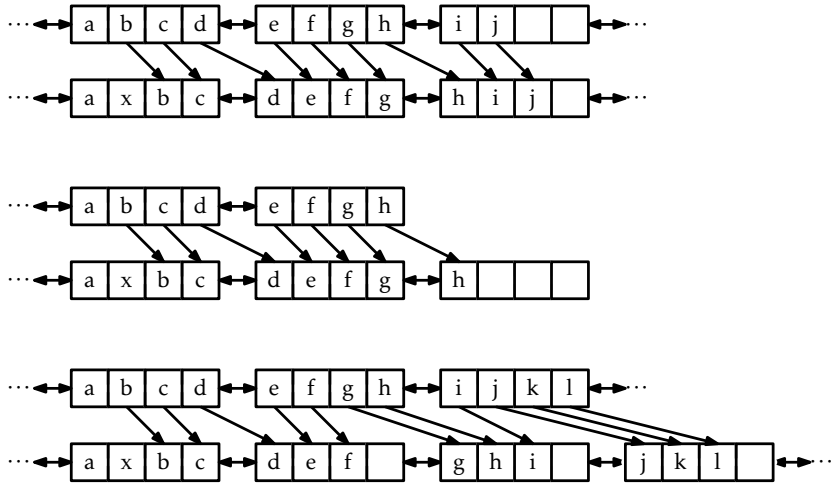


図 3.4: The three cases that occur during the addition of an item x in the interior of an SEList. (This SEList has block size b = 3.)

```
    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 のブロックは既に b+1 個の要素を含んでいるため 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。$u_0, u_1, u_2, \ldots$ を x を入れられるスペースを求めて探索する。この探索の過程で 3 つの可能性が考えられる。(Figure 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 ステップ探してもから出ない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 $b(b+1)$ この要素を広げる spread を呼ぶ。$u_0,\ldots,u_b$ はいずれも b 個の要素を含むようになる。すると u_0 のブロックは b 個の要素を含むため、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依って決まる。上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 $b(b+1)$ 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。)

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。

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。(Figure 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

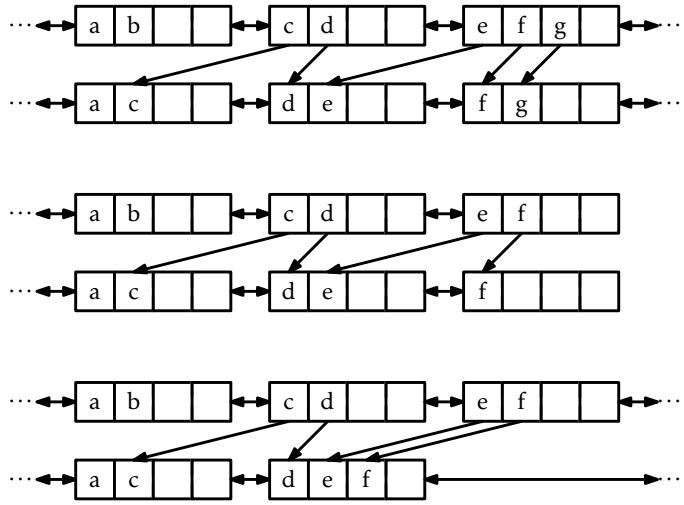


図 3.5: The three cases that occur during the removal of an item x in the interior of an SEList. (This SEList has block size b = 3.)

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。

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 $O((b+1)^2) = O(b^2)$ 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。

Lemma 3.1. 空の *SEList* が作られ、$m \geq 1$ 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 $O(bm)$ である。

Proof. 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 は不安定であるという。（すなわち、u は末尾のノードか、要素数が $b-1$ または $b+1$ である。）ブロックの要素数がちょうど b であるノードは安定であるという。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 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 $u_0,\ldots,u_{b-1}$ はいずれも不安定である。spread(u_0) が呼ばれると、これらの b 個の不安定なノードは b + 1 個の安定なノードに置き換えられる。そして x が u_0 のブロックに追加され、u_0 は不安定になる。合わせてポテンシャルは 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 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 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。 □

要約

次の定理は SEList の性能をまとめたものだ。

Theorem 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。

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

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

要素数 n の *SEList* における（ワード単位で測った）[*1] 領域使用量は $n+O(b+n/b)$ 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 b によって、ふたつのデータ構造の濃さを調整できるである。極端な場合として 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 b > 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

[*1] Section 1.4 で説明したメモリの図り方の議論を思い出すこと。

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。

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*[67] 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 u はひとつだけのポインタ u.nextprev を持つ。 このポインタは u.prev と u.next の XOR を取ったものである。リストは、dummy を指すポインタと dummy.next を指すポインタの二つを持つ必要がある。(dummy.next はリストが空なら dummy を、そうでないなら先頭のノードを指す。) このテクニックは u と u.prev があれば u.next を次の関係式から計算できることを利用している。

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

(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 [68] を参照してほしい。

Exercise 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。

Exercise 3.2. SLList のメソッド secondLast() を設定・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。

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

Exercise 3.4. SLList の reverse() 操作を設定・実装せよ。これは SLList の要素の順番を逆にする操作である。この操作の実行時間は $O(n)$ でなければならず、再帰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。

Exercise 3.5. SLList および DLList の checkSize() 操作を設計・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ものだ。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。

Exercise 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 の直前に追加するものだ。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。

Exercise 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、$i \in \{0,\ldots,n-1\}$ のいずれの場合も i 番目の要素が $n-i-1$ 番目の要素と等しいかどうかを確認するものである。実行時間は $O(n)$ である必要がある。

Exercise 3.8. rotate(r) を実装せよ。これは DLList の要素を回転するもので、i 番目の要素を $(i+r) \bmod n$ 番目の位置に移動するものだ。実行時間は $O(1+\min\{r,n-r\})$ 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。

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

Exercise 3.10. DLList の操作 absort(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。

Exercise 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 を返すものだ。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。

Exercise 3.12. reverse() を実装せよ。これは DLList の要素の順序を逆転するものだ。

Exercise 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 Section 11.1.1 扱う。In your implementation, perform comparisons between elements using the compareTo(x) method so that the resulting implementation can sort any DLList containing elements that implement the Comparable interface.

1. DLList の takeFirst(12) 操作を実装せよ。この操作は 12 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(),12.remove(0)) と等価である。
2. DLList の静的メソッド merge(11,12) を実装せよ。これはふたつの整列済みのリスト 11・12 を統合し、その結果を含む新たな整列済みリストを返す。この操作をすると 11・12 は空になる。例えば 11 の要素は a,c,d、12 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 11 と 12 に分割する。
   (c) 再帰的に 11 を整列する。
   (d) 再帰的に 12 を整列する。
   (e) 最後に 11 と 12 を統合して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。

Exercise 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。

Exercise 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。

Exercise 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。

Exercise 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。

Exercise 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。

Exercise 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面白くて実際の応用も多いデータ構造を紹介する。スキップリストは get(i)・set(i,x)・add(i,x)・remove(i) をいずれも $O(\log n)$ の時間で実行できる List の実装である。SSet の実装でもあり、すべての操作の期待実行時間は $O(\log n)$ 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によって要素の高さを決める。スキップリストの性能は期待実行時間とパス長を使って表現できる。コイントスの結果に応じて決まる確率からこの期待値は計算される。ランダムなコイントスは擬似乱数（あるいはランダムビット）生成器によるシミュレーションで実装される。

基本的な構造

イメージとしてはスキップリストは単方向連結リストが並んだもの $L_0,\ldots,L_h$ である。リスト L_r は L_{r-1} の部分集合を含む。まず n 個の要素を含む入力 L_0 がある。L_0 から L_1 を作り、L_1 から L_2 を作り、というのを繰り返す。L_{r-1} の各要素についてコインを投げ、表が出たら L_r はこれを含む。リスト L_r が空ならこの繰り返しを終える。スキップリストの例を Figure 4.1 に示した。

スキップリストの要素 x について、x の高さを For an element, x, in a skiplist, we call the *height* 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。

コインを裏が出るまで繰り返し投げる。表は何回出るだろうか。この答え、

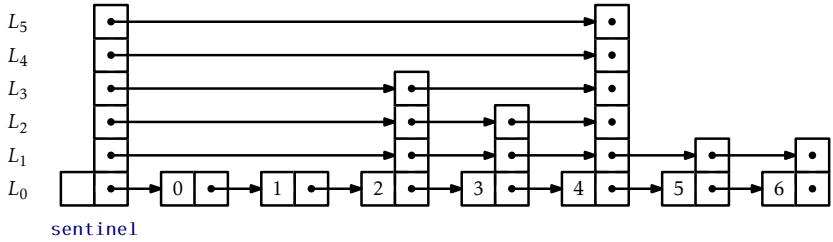


図 4.1: A skiplist containing seven elements.

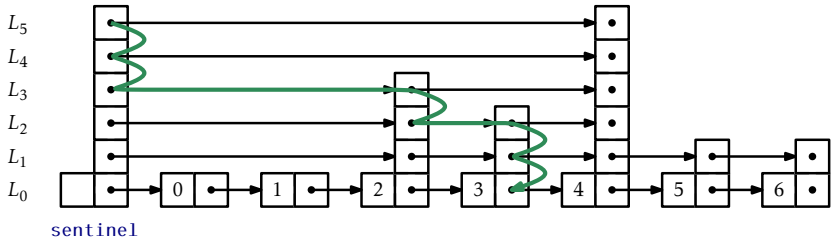


図 4.2: The search path for the node containing 4 in a skiplist.

そして高さの期待値は 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。（Figure 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` を通り過ぎない限り右に進む。`u` を通り過ぎてしまう場合はその代わりに下に進む。

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 secrefskiplist-analysis で証明するが、探索経路が非常に短いことを主張する。

Lemma 4.1. L_0 の任意のノード u への探索経路の長さの期待値は $2\log n + O(1) = O(\log n)$ 以下である。

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_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それぞれ紹介する。そこでは L_0 が主な構造（リストや整列された集合）を保持する。違いはどのように探索経路を辿り方である。下に進んで L_{r-1} に移るか、L_r の中で右に進むかの選び方に違いがあるのである。

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 y ≥ 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 Lemma 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) {
    k++;
```

[*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。

```
    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] にはパスにおいて 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;
  }
  n++;
```

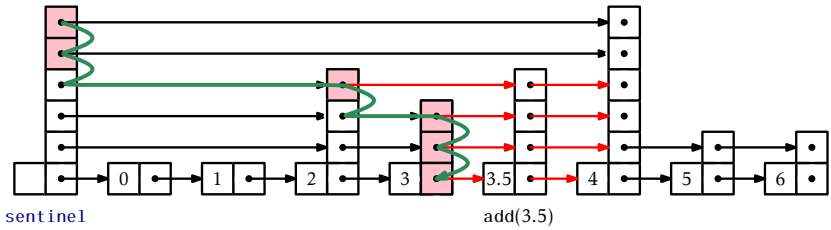


図 4.3: Adding the node containing 3.5 to a skiplist. The nodes stored in `stack` are highlighted.

```
  return true;
}
```

要素 `x` を削除するのも同様に行える。ただし `stack` で探索経路を覚えておく必要はない。削除は探索経路を辿りながら行うことが出来る。`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)
        h--;  // height has gone down
```

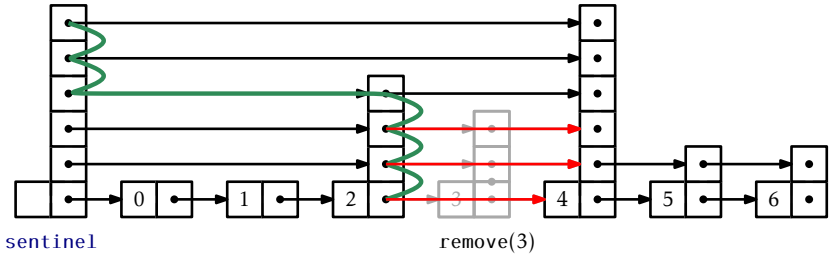


図 4.4: Removing the node containing 3 from a skiplist.

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

Summary

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。

Theorem 4.1. *SkiplistSSet* は *SSet* インターフェースの実装である。*SkiplistSSet* は操作 add(x)・remove(x)・find(x) を持ち、いずれの期待実行時間も $O(\log n)$ である。

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。

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

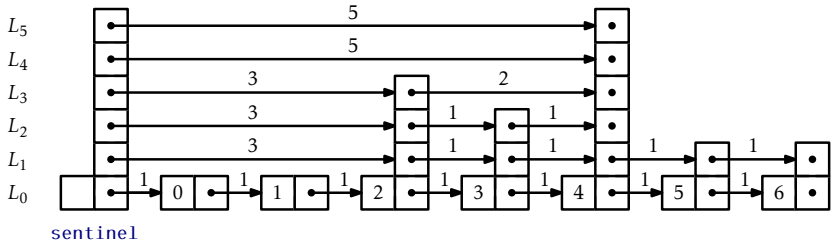


図 4.5: The lengths of the edges in a skiplist.

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。これは辺 e の長さは L_0 において e の下にある辺の数であるとするのと等価な定義である。この定義の例として Figure 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として、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 $j+\ell$ のノードに移るというものがある。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 を算出することができる。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。そうでないな下、すなわち 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操作は L_0 の i 番目のノードを見つける処理なので、これらの処理の実行時間は $O(\log n)$ である。

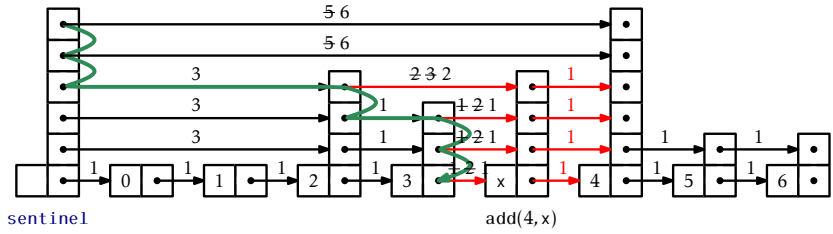


図 4.6: Adding an element to a SkiplistList.

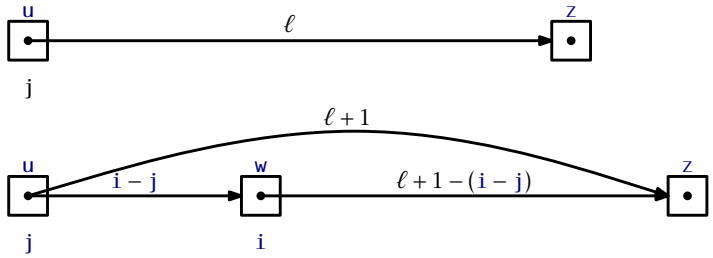


図 4.7: Updating the lengths of edges while splicing a node w into a skiplist.

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 ≤ k のときで、このとき w を L_r と次合わせる。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。Figure 4.6 を見よ。

探索経路上でリスト L_r のノード u に探索経路上で下ったとき、i 番目の位置に要素を追加することがわかるため辺 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が Figure 4.7 に示されている。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ることができる。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;
```

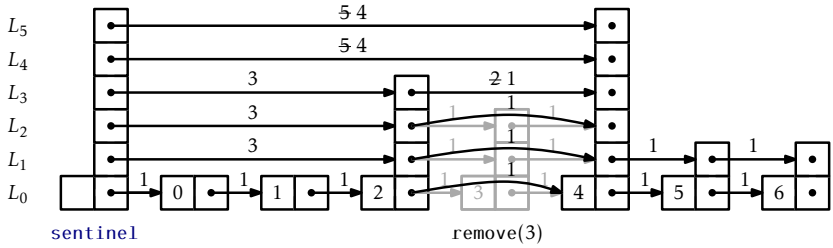


図 4.8: Removing an element from a SkiplistList.

```
}
```

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。Figure 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;
}
```

Summary

次の定理は SkiplistList の性能をまとめたものだ。

Theorem 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) の期待実行時間はいずれも $O(\log n)$ である。

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。いくつかの証明はコイントスについての次に述べる考察を利用する。

Lemma 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。）

Proof. 表が出たらにコイントスをやめるとする。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。

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

次のふたつの補題からスキップリストの大きさは要素数に対して線形だとわかる。

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

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

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

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

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

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

スキップリストの高さ h は次のように計算できる。

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

[*2] Section **??**を参照せよ。

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

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

よって

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

□

Lemma 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。

Proof. Lemma 4.3 より番兵を含まない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 Lemma 4.4 より $\log n + 2 = O(\log n)$ 以下である。 □

Lemma 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。

Proof. 最も簡単な方法はノード 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]Lemma 4.2 よりはじめて表が出るまでのコイントスの回数の期待値は 1 である。

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。$E[S_r] \leq 1$ 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 $S_r \leq |L_r|$ である。よって次の式が成り立つ。

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

あとは Lemma 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。

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

□

次の定理はこの節の結果をまとめるものだ。

[*3] これは大きく数えてしまうかもしれない。なぜなら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わるからである。しかしこれは問題ではない。なぜなら今考えている補題は上界に関するものだからである。

Theorem 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。

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 [60] であり、多くの拡張や応用も提案されている。[59] 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, 56]. 決定的な変種や [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。[69, 61] 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].

Exercise 4.1. Figure 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。

Exercise 4.2. Figure 4.1 のスキップリストに対して、0.5 を高さ 1 に追加し、その後 3.5 を高さ 2 に追加するときの振る舞いを説明せよ。

Exercise 4.3. Figure 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。

Exercise 4.4. Figure 4.5 の `SkiplistList` に `remove(2)` を実行する時の振る舞いを説明せよ。

Exercise 4.5. Figure 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。

Exercise 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。

Exercise 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 $(1/p)\log_{1/p} n + O(1)$ 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。

Exercise 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。

Exercise 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。

Exercise 4.10. XXX: 日本語汚い

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からなる配列である。（83 のコードで add(x) における変数 stack は指である。Figure 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 に到達するまで指を登り、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) の間に指は前回の find(x) の結果を指すように更新される。

Exercise 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。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。

Exercise 4.12. SkiplistList の absorb(l2) メソッドを実装せよ。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、11 の要素が a,b,c、12 の要素が d,e,f であるとき、11.absorb(12) を呼ぶと、11 の要素は a,b,c,d,e,f になり 12 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。

Exercise 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。

Exercise 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。

Exercise 4.15. （この練習問題は Section 6.2 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

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（n は小さい整数）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ふたつを紹介する。これはチェイン、または線形探索を使うものだ。

ハッシュテーブルは整数で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というデータに対応する値を使う。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。

この章で扱う手法にはある範囲からランダムに生成した整数を利用する。サンプルコードではこのランダム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から得られる。

ChainedHashTable: チェイン法を使ったハッシュテーブル

ChainedHashTable とはチェイン法を使ってデータをリストの配列 t に蓄え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。(Figure 5.1 を参照せよ。)

ChainedHashTable

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

データ x のハッシュ値 hash(x) とは {0,...,t.length−1} の中のある値であ

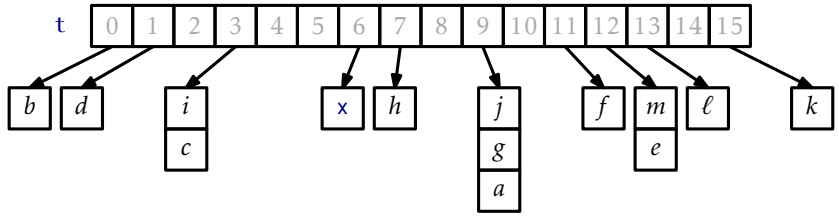


図 5.1: An example of a ChainedHashTable with n = 14 and t.length = 16. In this example hash(x) = 6

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。

$$n \le t.length$$

こうると、リストの平均要素数は常に 1 以下である。$n/t.length \le 1$

ハッシュテーブルに要素 x を追加するには配列 t の大きさを増やす必要があるかどうかを確認し、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、t の大きさを二倍にし、元の配列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。(必要ならページ 29 の Lemma 2.1 を見直すこと。)

拡張のあとはは 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 n_i をリスト t[i] の長さとするとき、$O(n_{\mathtt{hash}(x)})$ である。

ハッシュテーブルから要素 x を見つけるのも同様である。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 t[hash(x)] の長さに比例する時間がかかる。

ハッシュテーブルの性能はハッシュ関数の選択に大きく影響される。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し、各リストの長さの期待値は $O(n/t.length) = O(1)$ である。一方、よくないハッシュ関数はすべての要素を同じリストに追加してしまう。すなわち、リスト t[hash(x)] の長さは n になってしまう。次の小節ではよいハッシュ関数について説明する。

Multiplicative Hashing

乗算ハッシュ法は剰余算術（Section 2.3 で説明した）と整数の割り算からハッシュ値を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（これは次数と呼ばれるについて大きさ 2^d であるハッシュテーブルを使う。整数 $x \in \{0,\ldots,2^w - 1\}$ のハッシュ値は次のように計算される。

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

ここで z は $\{1,\ldots,2^w - 1\}$ のうちの奇数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^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。[*1]integer operation that overflows is upgraded to a variable-length representation.（Figure 5.2 を参照せよ。）さらに、2^{w-d} による整数の割り算は二進法で右側の $w-d$ ビットを落とせば計算できる。（これはビットを右に $w-d$ 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);
}
```

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。）

[*1] 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。w ビットの固定桁の演算の結果がビットに収まらなくなったときには、可変桁数の整数表現が使われるのである。

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

図 5.2: $w = 32$、$d = 8$ とした乗算ハッシュ法の操作

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

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

Lemma 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[hash(x)] の長さの期待値は $n_x + 2$ 以下である。

Proof. 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。

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

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

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

□

Now, we want to prove 続いて Lemma 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 $(b_r,\dots,b_0)_2$ 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、$(b_r,\dots,b_0)_2$ は二進表記で $b_r,\dots,b_0$ 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。

Lemma 5.3. S を $\{1,\dots,2^w-1\}$ 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、$z \in S$ の要素が一意に存在して $zq \bmod 2^w = i$ を満たす。

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

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

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

よって、

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

しかしこれはある整数 k について次の式が成り立つことを意味する。

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

2 進数のことを考えると

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

なので、$(z-z')q$ の末尾 w 桁はすべて 0 である。

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

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

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

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

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

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

以上より、$(z-z')q$ は (5.1) を満たさず、矛盾する。 □

Lemma 5.3 から次の便利な事実がわかる。z が S から一様な確率でランダムに選ばれるとき、zt は S 上に一様分布する。次の証明では z の最下位の 1 である桁を除いた、w−1 桁のランダムなビットを考えるのがポイントだ。

Proof of Lemma 5.1. 条件 hash(x) = hash(y) と「zx mod 2^w の上位 d ビット zy mod 2^w の上位 d ビットが等しい」は同値である。この条件の必要条件は、z(x−y) mod 2^w の上位 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、zx mod 2^w > zy mod 2^w ならば次の条件である。

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

一方、zx mod 2^w < zy mod 2^w ならば次の条件である。

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

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

q を、ある整数 $r \geq 0$ が存在し、(x−y) mod $2^w = q2^r$ を満たす一意な奇数とする。Lemma 5.3 より、zq mod 2^w の二進表現は w−1 桁のランダムなビットを持つ。（最下位桁は 1 である。）

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

よって z(x−y) mod 2^w = z$q2^r$ mod 2^w は桁の w−r−1 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。）

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

これで証明が終わる。r > w−d ならば z(x−y) mod 2^w の上位 d ビットは 0 と 1 を共に含む。よって z(x−y) mod 2^w が (5.2) または (5.3) である確率は 0 である。r = w−d ならば (5.2) の確率は 0 だが、(5.3) である確率は $1/2^{d-1} = 2/2^d$ である。（これは $b_1,\ldots,b_{d-1} = 1,\ldots,1$ である必要があるためだ。）r < w−d ならば $b_{w-r-1},\ldots,b_{w-r-d} = 0,\ldots,0$、すなわち $b_{w-r-1},\ldots,b_{w-r-d} = 1,\ldots,1$ である。いずれの場合の確率も $1/2^d$ であり、またそれぞれの事象は互いに排反である。よって、このどちらかである確率は $2/2^d$ である。 □

要約

次の定理は ChainedHashTable の性能をまとめたものだ。

Theorem 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。

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合計時間は $O(m)$ である。

LinearHashTable：線形探索法

The 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。オープンアドレス法と呼ばれる別の方法があり、これは配列 t に直列要素を収めるものだ。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。

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、上の一つ目と三つ目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。効率的にこれを実装するために、t は 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 $t.length \geq 2q$ 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。

これはそれぞれ要素数と、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 2 の冪に制限されていることが多いので、整数不変条件 t.length = 2^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、t[i], t[(i+1) mod t.length], t[(i+2) mod t.length], ... と順に、t[i′] = x または t[i′] = null を満たす添え字 i′ を探す。t[i′] = x のとき、x が見つかったとして t[i′] を返す。t[i′] = 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 $t[i'] = \texttt{null}$ 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。

null でないエントリの数が t.length/2 より大きいときに add(x) を呼ぶとき、またはデータの入っているエントリ数が t.length/8 よりも小さいときに remove(x) を呼ぶと 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 $2^d \geq 3n$ を満たす最小の非負整数 d を見つける。大きさ 2^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];
    }
  }
}
```

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）終了することに注意する。配列 t の半分以上は null なので、直感的には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そうに思う。しかしあまりこの直感を当てにはできない。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、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 t[i] と書いても t[i mod t.length] のことである。

XXX: run の訳語 i から始まる長さ k の *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 $\mathtt{q} \le \mathtt{t.length}/2$ であることを保証する。直前の rebuild() 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。

Lemma 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 *run* が発生する確率を $O(c^k)$ と表せる。

Proof. i から始まる長さ k の run 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。[*2]

[*2] p_k は i から始まる長さ k の run 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

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（Section 1.3.2）からこれは真実からのずれは $O(\sqrt{r})$ 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。Exercise 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。

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 t.length ≥ 2q を保つ。よって次の式が成り立つ。

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

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

Lemma 5.4 を使えば find(x)・add(x)・remove(x) の期待実行時間の上界は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 に入っていないときを考える。この場合 i = hash(x) は $\{0,\ldots,$ t.length $-1\}$ の値を取り、t の中身と独立な確率変数である。i が長さ k の run 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

る。

行時間の期待値の上界を計算できる。

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

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

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

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。[*3] 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、この解析で LinearHashTable の解析を終わりだ。

まず上の 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。

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、Section 2.1 で ArrayStack の償却解析を行ったのと同様である。

[*3] 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。

Summary

次の定理はLinearHashTableの性能をまとめたものだ。

Theorem 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。

Tabulation Hashing

LinearHashTableの解析では強い仮定を置いていた。すなわち、任意の相異なる要素 $\{x_1,\ldots,x_n\}$ についてそのハッシュ値 $\texttt{hash}(x_1),\ldots,\texttt{hash}(x_n)$ が独立に一様な確率で $\{0,\ldots,\texttt{t.length}-1\}$ 内を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^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^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 ビットの整数の代わりに w/r 個の r ビット整数で妥協する。こうすれば大きさ 2^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。-bit integers. To obtain the value of 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 $w=32, r=4$ 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 $2^{32/4} = 256$ の行からなる二次元配列である。XXX: これ必要?

任意の x について hash(x) は $\{0,\ldots,2^d-1\}$ の値を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 個の値の組みについて、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 Theorem 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。

ハッシュ値

XXX: ハッシュ値と Hash Code は区別したほうがよいか?

前節のハッシュテーブルではデータに対応する w ビットの整数を使っていた。しかしキーが整数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。

Hash Codes for Primitive Data Types

`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。これらの方はバイナリ表現があり、これは `w` ビット以下である。XXX: Ruby の整数の話をするか? or fewer bits. (For example, in Java, `byte` is an 8-bit type and `float` is a 32-bit type.) このビット列を $\{0,\ldots,2^w-1\}$ の範囲の整数であると解釈すればよい。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、cw 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。

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 $x_0,\ldots,x_{r-1}$ であるとする。このとき互いに独立な w ビットの乱数 $z_0,\ldots,z_{r-1}$ と、$2w$ ビットのランダムな奇数 z から、複合オブジェクトのハッシュ値を計算できる。

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

このハッシュ値の計算過程は最後に z をかけ、2^w で割っていることに注目してほしい。これは $2w$ ビットの中間結果に Section 5.1.1 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。

Theorem 5.3. *Let* $x_0,\dots,x_{r-1}$ と $y_0,\dots,y_{r-1}$ はいずれも、$\{0,\dots,2^w-1\}$ の要素である w ビットの整数からなる列とする。さらに、少なくとも一箇所の添え字 $i \in \{0,\dots,r-1\}$ で $x_i \neq y_i$ が成り立つと仮定する。このとき、次が成り立つ。

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

Proof. 最後の乗算ハッシュ法については後半に考える。次の関数を定義する。

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

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

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

ここで t は次のものである。

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

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

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

これは $z_i \cdot (x_i - y_i)$ 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（$z_0, \ldots, z_{r-1}$ は互いに独立である）なので、$h'(x_0, \ldots, x_{r-1}) = h'(y_0, \ldots, y_{r-1})$ を満たす z_i を選ぶ確率は $1/2^w$ 以下である。

最後の処理は乗算ハッシュ法であり、$2w$ ビットの中間結果 $h'(x_0, \ldots, x_{r-1})$ を w ビットの最終結果 $h(x_0, \ldots, x_{r-1})$ に縮める。Theorem 5.3 より、$h'(x_0, \ldots, x_{r-1}) \neq h'(y_0, \ldots, y_{r-1})$ ならば $\Pr\{h(x_0, \ldots, x_{r-1}) = h(y_0, \ldots, y_{r-1})\} \leq 2/2^w$ である。

以上より、次の式が成り立つ。

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

□

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部分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 を部分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、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 Theorem 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。

Theorem 5.4. *XXX: non-trivial polynomial の正確な定義は?*

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

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

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

最後に追加されたの項 $(p-1)z^r$ に注意する。これは $(p-1)$ を整数列の末尾の要素として $x_0,\ldots,x_r$ 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 $\{0,\ldots,p-2\}$ 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、良い出力を返すことを示すものだ。

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

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

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

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

Since $x_i \neq y_i$ なので、この多項式は非自明である。よって Theorem 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、一方が他方の接頭語になっていても構わない。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。

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

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 Theorem 5.4 より z について高々 r 個の解をもつ。Theorem 5.5 と合わせると次のより一般的な定理が示せる。

Theorem 5.6. p を $p > 2^w+1$ を満たす素数とする。$x_0,\ldots,x_{r-1}$ と $y_0,\ldots,y_{r'-1}$ は $\{0,\ldots,2^w-1\}$ の要素である 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。

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

次のサンプルコードを見れば配列 x を含むオブジェクト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をやや損なっている。特に x[i].hashCode() を 31 ビットに縮めるために Section 5.1.1 で d = 31 とした乗算ハッシュ法を使っている。これは素数 $p = 2^{32}-5$ 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 Theorem 5.6 で求めた $r/(2^{32} - 5)$ よりも大きい。

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。

Section 5.1 で説明したものはチェイン法 *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。Section 5.2 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消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 $O(1)$ になるハッシュ法だ。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 [31, 24] や *cuckoo hashing* [55] 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を使ったものがある。いくつかのことなるシナリ

オのためのハッシュ関数が提案されている。[14]. Section 5.2.3 で説明した Tabulation hashing は Carter と Wegman の研究 [14] によるものだが、この手法を線形探索法（と他のいくつかのハッシュテーブルの実装）に適用した場合の解析は Pătraşcu と Thorup の研究成果である。[58].

The idea of 乗算ハッシュ法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民俗学の一部である。[48, Section 6.4] しかし、Section 5.1.1 で説明した乗数 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 $2/2^d$、つまり 2^w から 2^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。XXX: multiply-add の訳語 *multiply-add* ハッシュ法は次の関数を使う方法だ。

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

ここで z と b いずれも $\{0,\ldots,2^{2w}-1\}$ からランダムに選出される。Multiply-add ハッシュ法の衝突確率は $1/2^d$ である。[21] しかし、$2w$ ビット精度の四則演算が必要である。

固定長の w ビットの整数列からハッシュ値を得る方法はたくさんある。特に高速な方法は次のものだ。[11]

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

ここで r は偶数であり、$a_0,\ldots,a_{r-1}$ はいずれも $\{0,\ldots,2^w\}$ からランダムに選出される。

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

The method from Section 5.3.3 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、結果この方法は速くない。剰余の法として 2^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。

Exercise 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。

Exercise 5.2. Section 5.1.1 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。)

Exercise 5.3. Lemma 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、$\Pr\{\mathsf{hash}(x) = \mathsf{hash}(y)\} = 2/2^d$ 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。)

Exercise 5.4. Section 1.3.2 で与えたスターリングの公式を使って、Lemma 5.4 を今度は誤魔化しなしで証明せよ。

Exercise 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;
}
```

Exercise 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。

Exercise 5.7. ふたつの w ビットの整数 x と y からなるオブジェクトがあるとき、$x \oplus y$ 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。

Exercise 5.8. ふたつの w ビットの整数 x と y からなるオブジェクトがあるとき、$x + y$ をハッシュ値とするのはよくないことを示せ。すなわち、同じハッシュ値を持つオブジェクトの集まりの例を挙げよ。

Exercise 5.9. ふたつの w ビットの整数 x と y からなるオブジェクトがあるとする。決定的な関数 $h(x,y)$ により w ビットの整数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。

Exercise 5.10. ある正の数 w について、$p = 2^w - 1$ であるとする。正の数 x について次の式が成り立つ理由を説明せよ。

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

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

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

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

Exercise 5.11. 標準ライブラリやこの本の HashTable・LinearHashTable の実装について、find(x) が定数時間でなくなるデータをテーブルに挿入する

プログラムを書け。つまり、テーブルの中の同じ位置に対応付けられる n 個の整数の集まりを見つけよ。

実装によって、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 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。この構造を木と呼ぶのは、図示したときに（森に生えてる）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、二分木とは連結な有限無向グラフであって、サイクルを持たず、すべての頂点の次数が 3 以下のものである。

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。次数 2 以下のノードのうち、特別なもの r を、木の根と呼ぶ。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 の親という。u に隣接する親以外のノードを u の子という。特に順序付けられている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、ふつう根から下に向かって描く。根が一番上に描かれる。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。（Figure 6.1）Figure 6.2.a は 9 個のノードを持つ二分木の例である。

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。二分木におけるノード u の深さとは、u から根までの経路の長さである。ノード w が u から r への経路に含まれるとき、w を u の祖先という。一方、このとき u を w の子孫という。二分木におけるノード u の部分木とは、u を根とし、u のすべての子孫を含む二分木である。ノード u の高さとは、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 は葉である。

外部ノード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。（Figure 6.2.b を参照せよ。）帰納法により、n $\geq$ 1 個の（本

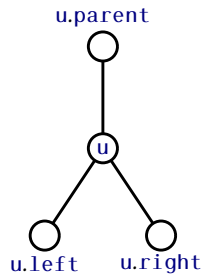


図 6.1: BinaryTree における、ノード u の親・左の子・右の子

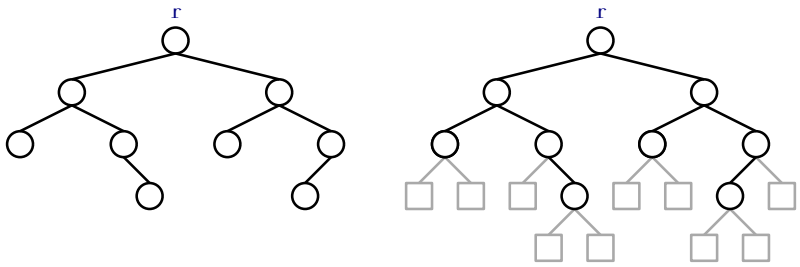


図 6.2: (a) 9 個の本物のノードを持つ二分木と、(b) 10 個の外部ノードを持つ二分木

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。

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、（3 つ以下の）u に隣接する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;
}
```

隣接する頂点が 3 つ揃っていないときには、そこは 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、根 r への参照として二分木自体は表現できる。

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;
}
```

再帰的なアルゴリズム

再帰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。

BinaryTree

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

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。

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。

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。Figure 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。

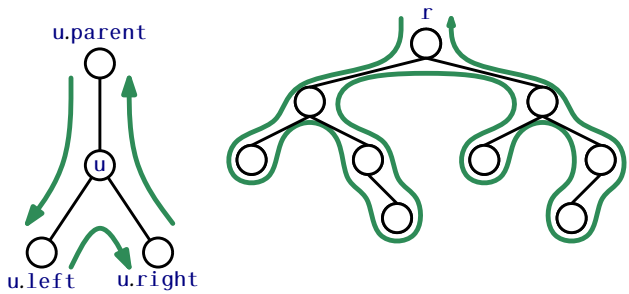


図 6.3: 二分木を再帰を使わずに走査するときの、三通りのノード u の訪れ方と、その結果として生じる木の走査

BinaryTree

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

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、幅優先な走査がある。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。(Figure 6.4 を参照せよ。)これは英語の文章の読み方と似ている。(訳注：横書きなら、日本語でも同様である。)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

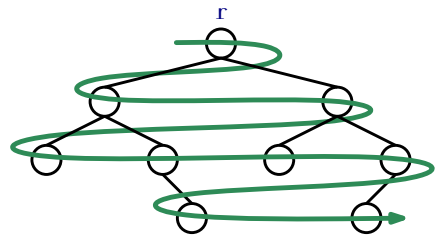


図 6.4: 幅優先な走査では、二分木の各ノードは深さ毎に、また各深さでは左から右の順で訪問される。

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);
  }
}
```

BinarySearchTree：バランスされていない二分探索木

BinarySearchTree はある性質を持つ二分木である。ノード u はデータ u.x を持ち、このデータはある全順序な集合の要素である。二分探索木の各ノードとそのデータは次の二分探索木性を満たす。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

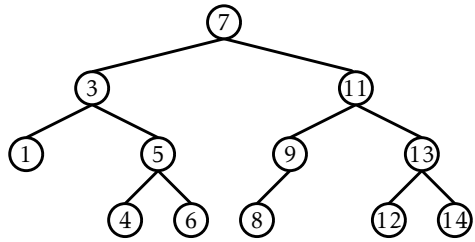


図 6.5: 二分探索木の例

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 Figure 6.5 に示す。

探索

二分探索木性は有用だ。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。

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ふたつ Figure 6.6 に示す。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;
```

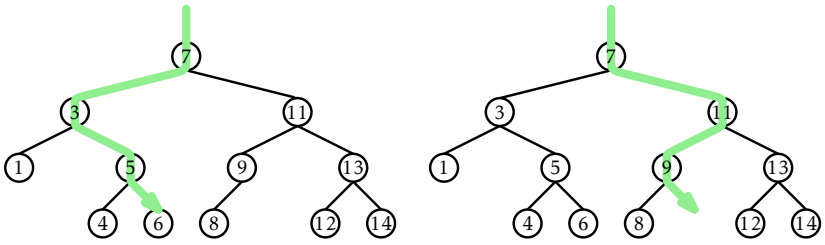


図 6.6: 二分探索木において、(a) 探索が成功する例（6 が見つかる）と、(b) 探索が失敗する（10 が見つからない）例

```
}
```

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);
  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;
```

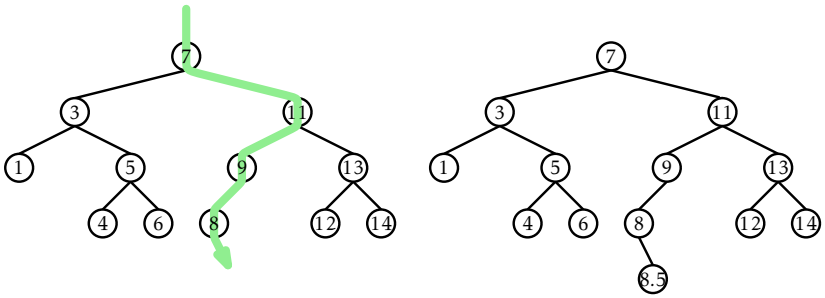


図 6.7: 二分探索木に 8.5 を追加する様子

```
}
```

Figure 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。

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。(Figure 6.8 を参照せよ。)

BinarySearchTree

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

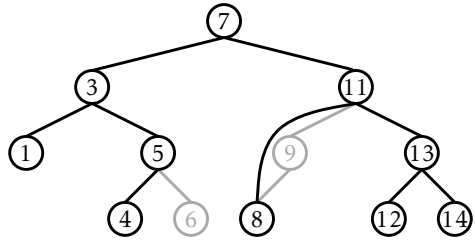


図 6.8: 葉 (6)、または一つだけの子を持つノード (9) を削除するのは簡単である。

```
    r = s;
    p = nil;
  } else {
    p = u.parent;
    if (p.left == u) {
      p.left = s;
    } else {
      p.right = s;
    }
  }
  if (s != nil) {
    s.parent = p;
  }
  n--;
}
```

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。二分探索木性を保つためには、w.x の値は u.x の値に近いのがよい。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

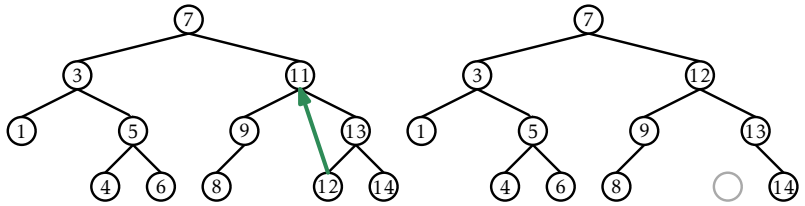


図 6.9: ふたつの子を持つノード u から値 11 を削除するために、u の値と、u の右の部分木における最小の値とを入れかえる。

くのも簡単である。(Figure 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);
  }
}
```

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。

Theorem 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。

Theorem 4.1 と比べると、Theorem 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かもしれないことだ。Figure 6.5 のような形ではなく、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。

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。Chapter 7 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。Chapter 8 では償却実行時間 $O(\log n)$ を、部分的な再構築を利用して達成する方法を説明する。Chapter 9 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。

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。家系図を自然にモデル化すると二分木になる。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。ここ数百年の話としては、生物学における系統樹が二分木である。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 154 の `rotateLeft(u)`・`rotateRight(u)` を参照せよ。

Exercise 6.1. n ≥ 1 個のノードからなる二分木は n − 1 本の辺を持つことを示せ。

Exercise 6.2. n ≥ 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。

Exercise 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。

Exercise 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。

Exercise 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。

Exercise 6.6. 二分木がサイズでバランスされているとは、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 Figure 6.10 を見よ。

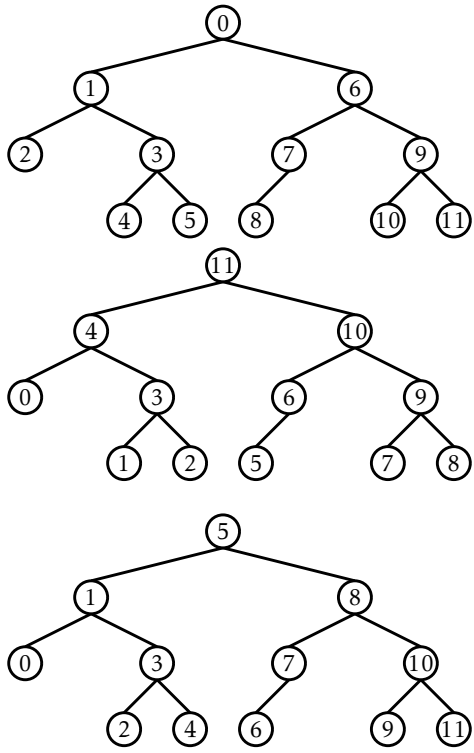


図 6.10: 二分木における行きがけ順・通りがけ順・帰りがけ順

Exercise 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。

Exercise 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 $O(n)$ でなければならない。

Exercise 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。

Exercise 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。

Exercise 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：木を辿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。）

Exercise 6.12. Figure 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。

Exercise 6.13. Figure 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。

Exercise 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。

Exercise 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。

Exercise 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?

Exercise 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?

Exercise 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?

Exercise 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) を読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

ランダム二分探索木

この章ではランダム化を利用することで各操作の期待実行時間が $O(\log n)$ であるような二分探索木を紹介する。

ランダム二分探索木

Figure 7.1 に示したふたつの二分探索木を見てほしい。これらはいずれも $n = 15$ 個のノードを含む。左のものはリストであり、右のものは完全にバランスされた二分探索木である。左のものの高さは $n - 1 = 14$ で、右のものの高さは 3 である。

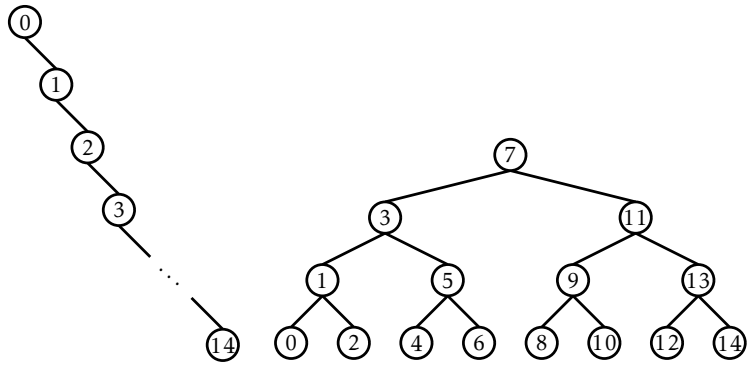


図 7.1: Two binary search trees containing the integers $0, \ldots, 14$.

このふたつの木がどう構築されるかを考えてみよ。左のものは空の BinarySearchTree に次の要素の列を順に追加すると得られる。

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

この木が得られる追加操作の列はこれしかない。（証明は n についての帰納法で行う。）一方右の木は次の列を順に追加すれば得られる。

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

他にも

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

や

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

でもこの木は得られる。右の木を作る操作の列は実は 21,964,800 種類ある。一方で左の木の場合には唯一であった。

上の例は定性的に、$0,\ldots,14$ をランダムに並び替えた列の要素を順に二分探索木に入れると、非常に（Figure 7.1 の右のもののような）バランスの良い木ができることが多く、（Figure 7.1 の左のもののような）非常にバランスの悪い木は滅多にできないことを示している。

これを形式的に表現するための記法としてランダム二分探索木のことを考える。サイズ n のランダム二分探索木は次のように得られる。

$0,\ldots,n-1$ の置換からランダムに選出した $x_0,\ldots,x_{n-1}$ をひとつずつ順に BinarySearchTree に追加する。ここでランダムな置換とは、$0,\ldots,n-1$ の $n!$ 個ある並び替えを、いずれも等しい確率 $1/n!$ でひとつ選出したものことをいう。

値 $0,\ldots,n-1$ は順序を持った集合の要素 n 個組みと入れ替えてよく、そうしてもランダム二分探索木の性質は変わらないことに注意する。$x \in \{0,\ldots,n-1\}$ は単にサイズ n の順序付き集合の x 番目の数を表しているだけなのである。

ランダム二分探索木についての主要な成果を説明する前に、少し脱線してランダムな構造を考える際によく出てくる数についての話をする。非負整数 k について、k 番目の調和数 H_k は次のように定義される。

$$H_k = 1 + 1/2 + 1/3 + \cdots + 1/k\ .$$

調和数 H_k に単純な閉じた書き方はないが、自然対数との間には密接な関係がある。特に次の式が成り立つ。

$$\ln k < H_k \leq \ln k + 1\ .$$

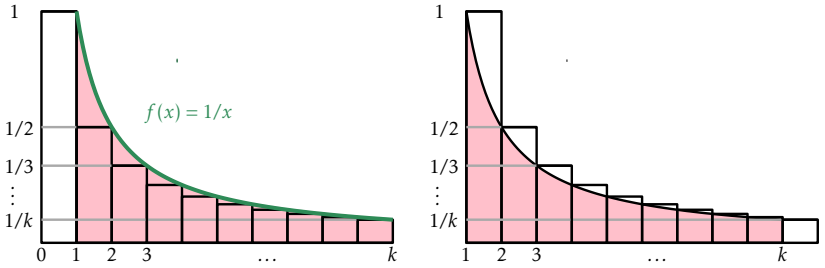


図 7.2: The kth harmonic number $H_k = \sum_{i=1}^{k} 1/i$ is upper- and lower-bounded by two integrals. The value of these integrals is given by the area of the shaded region, while the value of H_k is given by the area of the rectangles.

解析学を学んだ読者は $\int_1^k (1/x)\,\mathrm{d}x = \ln k$ からこれを導けるだろう。積分は曲線と x 軸との囲む領域の面積と解釈でき、H_k の値の下界として $\int_1^k (1/x)\,\mathrm{d}x$、上界として $1 + \int_1^k (1/x)\,\mathrm{d}x$ がある。(Figure 7.2 を参考にせよ。)

Lemma 7.1. サイズ n のランダム二分探索木について次の命題が成り立つ。

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

Lemma 7.1 は次の小節で証明する。ここでは Lemma 7.1 のふたつの部分からなにがわかるかを考える。ひとつめの項目はサイズ n の木から要素を探索するとき、探索経路の長さの期待値が $2\ln n + O(1)$ 以下であることを主張する。ふたつめの項目は木に含まれない要素の探索に関するものだ。ふたつを比べると、木に入っている要素を探すのは入っていない要素を探すのに比べて少しだけ早いことがわかる。

[*1] x + 1 と n − x はそれぞれ木の要素のうち x 以上のものと x 以下のものの数であると解釈できる。

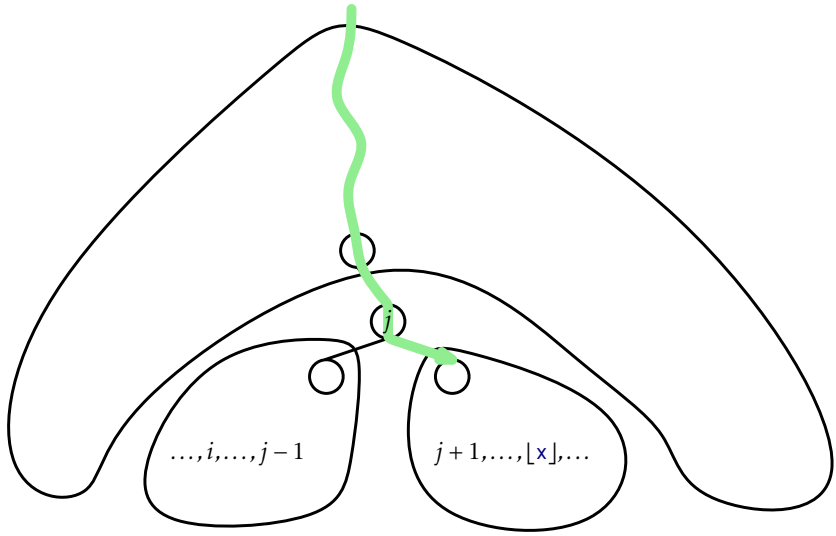


図 7.3: The value $i < \mathtt{x}$ is on the search path for $\mathtt{x}$ if and only if i is the first element among $\{i, i+1, \ldots, \lfloor \mathtt{x} \rfloor\}$ added to the tree.

Lemma 7.1 の証明

Lemma 7.1 の証明に必要な観察は次のものだ。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 $\mathtt{i}$ 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 $\{i+1, i+2, \ldots, \lfloor \mathtt{x} \rfloor\}$ のいずれかが i より前に現れることである。

これは Figure 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かったことから確認できる。XXX: よくわからん??? (Remember that for two values to have different search paths, there must be some element in the tree that compares differently with them.) 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 $\mathtt{x}$ 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 $\mathtt{x}$ の探索経路はこの部分木を通らない。なぜならこの経路は $\mathtt{u}_j$ を訪問したあと $\mathtt{u}_j.\mathtt{right}$ に向かうからである。

同様に $i > \mathtt{x}$ について、i が $\mathtt{x}$ の探索経路に現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 $\{\lceil x\rceil, \lceil x\rceil+1, \ldots, i-1\}$ のいずれよりも前に現れることである。

$\{0,\ldots,n\}$ のランダムな置換を考えるとき、$\{i, i+1, \ldots, \lfloor x\rfloor\}$・$\{\lceil x\rceil, \lceil x\rceil+1, \ldots, i-1\}$ 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。

XXX: この辺からちょっと意味がわからない

permutations of their respective elements. Each element, then, in the subsets $\{i, i+1, \ldots, \lfloor x\rfloor\}$・$\{\lceil x\rceil, \lceil x\rceil+1, \ldots, i-1\}$ もやはり同様に等しい確率で is equally likely to appear before any other in its subset in the random permutation used to create T. So we have

$$\Pr\{i \text{ is on the search path for } x\} = \begin{cases} 1/(\lfloor x\rfloor - i + 1) & \text{if } i < x \\ 1/(i - \lceil x\rceil + 1) & \text{if } i > x \end{cases} .$$

With this observation, the proof of Lemma 7.1 involves some simple calculations with harmonic numbers:

Proof of Lemma 7.1. I_i を指示確率変数とする。これは i が探索経路に現れるなら 1、そうでないなら 0 になる。このとき探索経路の長さを次のように計算できる。

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

よって $x \in \{0,\ldots,n-1\}$ なら探索経路の長さの期待値は次のように計算できる。(Figure 7.4.a を見よ。)

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

値 $x \in (-1, n) \setminus \{0,\ldots,n-1\}$ の対応する計算もほぼ同様である。(Figure 7.4.b を見よ。) □

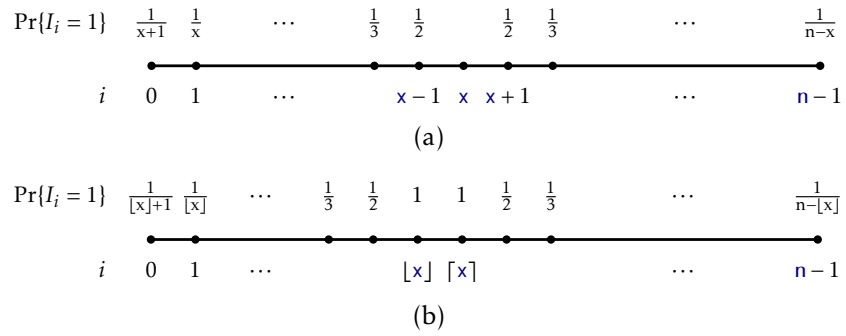


図 7.4: The probabilities of an element being on the search path for x when (a) x is an integer and (b) when x is not an integer.

Summary

次の定義はランダム二分探索木の性能をまとめたものだ。

Theorem 7.1. ランダム二分探索木は $O(n \log n)$ の時間で構築できる。ランダム二分探索木における `find(x)` の期待実行時間は $O(\log n)$ である。

Theorem 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よることを強調しておく。これは x の選び方がランダムであるというわけでなく、任意の x について成り立つものなのである。

Treap

ランダム二分探索木の問題は当然ながら動的でないことだ。SSet インターフェースを実装するために必要な `add(x)`・`remove(x)` をサポートしていないのである。この節では Treap と呼ばれるデータ構造を説明する。これは Lemma 7.1 を使って SSet インターフェースを実装する。[*2]

Treap のノードは値 x を持つ点で BinarySearchTree に似ているが、それに加えて一意の数である優先度 p を持つ。そしてこの p はランダムに割当て

[*2] Treap の名はこのデータ構造は二分木 **tr**ee(Section 6.2) であると同時にヒープ **heap**(Chapter 10) でもあることによる。

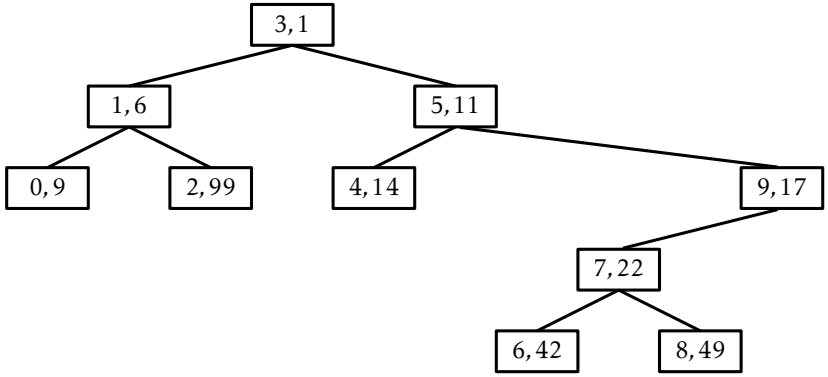


図 7.5: An example of a Treap containing the integers 0,…,9. Each node, u, is illustrated as a box containing u.x, u.p.

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、ヒープ性も満たす。

- （ヒープ性）根でない任意のノード u について u.parent.p < u.p が成り立つ。

言い換えると、どのノードの優先度もそのふたつの子のいずれの優先度よりも小さい。Figure 7.5 にこの例を示した。

ヒープと二分探索木の性質を共に満たすことから、キー x と優先度 p が決まると Treap の形状が完全に定まる。ヒープ性から最小の優先度を持つノードが Treap の根 r であることがわかる。二分探索木性から r.x より小さなキーを持つノードは r.left を根とする部分木に含まれ、r.x より大きなキーを持つノードは r.right を根とする部分木に含まれることがわかる。

Treap の優先度の重要な特徴は一意であり、ランダムに割当てられることである。このことからふたつの Treap についての等価な考え方がある。先に定義したように、Treap はヒープ性と二分探索木性に従う。この代わりに

Treap をノードが優先度の昇順に追加されていく BinarySearchTree であると考えることもできる。例えば Figure 7.5 の Treap は、BinarySearchTree に対して次の値 (x,p) の列を追加することで得られる。

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

優先度はランダムに決まるのでこれはキーのランダムな置換を取るのと同じである。この場合は次の置換に対応する。

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

これらを BinarySearchTree に追加すればよい。これは treap の形状はランダム二分探索木と同じであることを意味する。特にもしキー x をそのランクに置き換えると [*3]rank of an element x in a set S of elements is the number of Lemma 7.1 を適用できる。Lemma 7.1 を Treap の用語で言い換えると次のようになる。

Lemma 7.2. n 個のキーからなる集合 S を保持する *Treap* において次の命題が成り立つ。

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

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

繰り返しになるが、Lemma 7.2 の期待値は頂点への優先度の割当てのランダム性によることを強調しておく。これはキーがランダムであることは全く仮定していない。

Lemma 7.2 から Treap の `find(x)` は効率よく実装できることがわかる。しかし本当に嬉しいのは `add(x)`・`delete(x)` 操作を実装できることである。このためにはヒープ性を保つための回転操作が必要である。Figure 7.6 を参照せよ。二分探索木の回転とはノード w と親 u について、二分探索木性を保ちながら w を u の親にする操作である。回転には右回転と左回転の二種類があって、それぞれ w が u の右の子であるか、左の子であるかに対応する。

[*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。

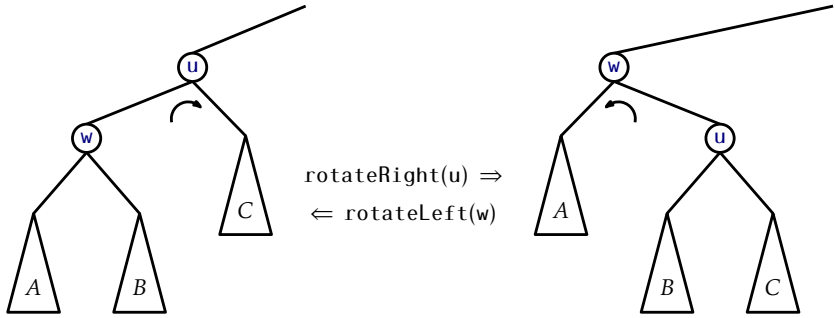


図 7.6: Left and right rotations in a binary search tree.

これを実装するときには、ふたつの場合を処理し、コーナーケース（u が根である場合）に気をつける必要がある。そのため実際のコードは Figure 7.6 を見て想像するものよりも少し長い。

BinarySearchTree

```
void rotateLeft(Node u) {
  Node w = u.right;
  w.parent = u.parent;
  if (w.parent != nil) {
    if (w.parent.left == u) {
      w.parent.left = w;
    } else {
      w.parent.right = w;
    }
  }
  u.right = w.left;
  if (u.right != nil) {
    u.right.parent = u;
  }
  u.parent = w;
  w.left = u;
```

```
  if (u == r) { r = w; r.parent = nil; }
}
void rotateRight(Node u) {
  Node w = u.left;
  w.parent = u.parent;
  if (w.parent != nil) {
    if (w.parent.left == u) {
      w.parent.left = w;
    } else {
      w.parent.right = w;
    }
  }
  u.left = w.right;
  if (u.left != nil) {
    u.left.parent = u;
  }
  u.parent = w;
  w.right = u;
  if (u == r) { r = w; r.parent = nil; }
}
```

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、ヒープ性を満たすとは限らない。特にこれは u.parent.p > u.p の場合である。この場合、w=u.parent で回転操作実行し、u を w の親にする。u が引き続きヒープ性を犯しているなら、これを繰り返す。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;
  }
}
```

Figure 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。Lemma 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。(Exercise 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。)

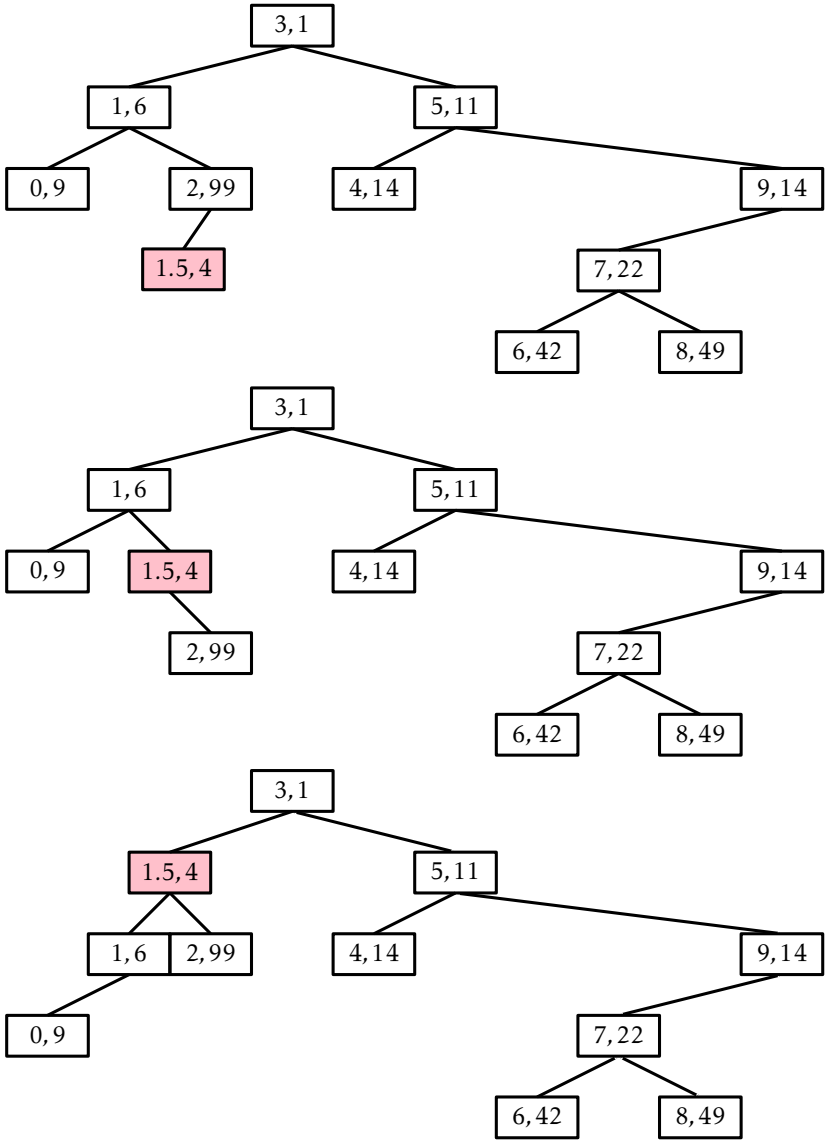


図 7.7: Adding the value 1.5 into the Treap from Figure 7.5.

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

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;
    }
  }
}
```

Figure 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) 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。(Figure 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。

Summary

次の定理は Treap の性能をまとめるものだ。

Theorem 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 Exercise 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、これは各操作も Skiplist より

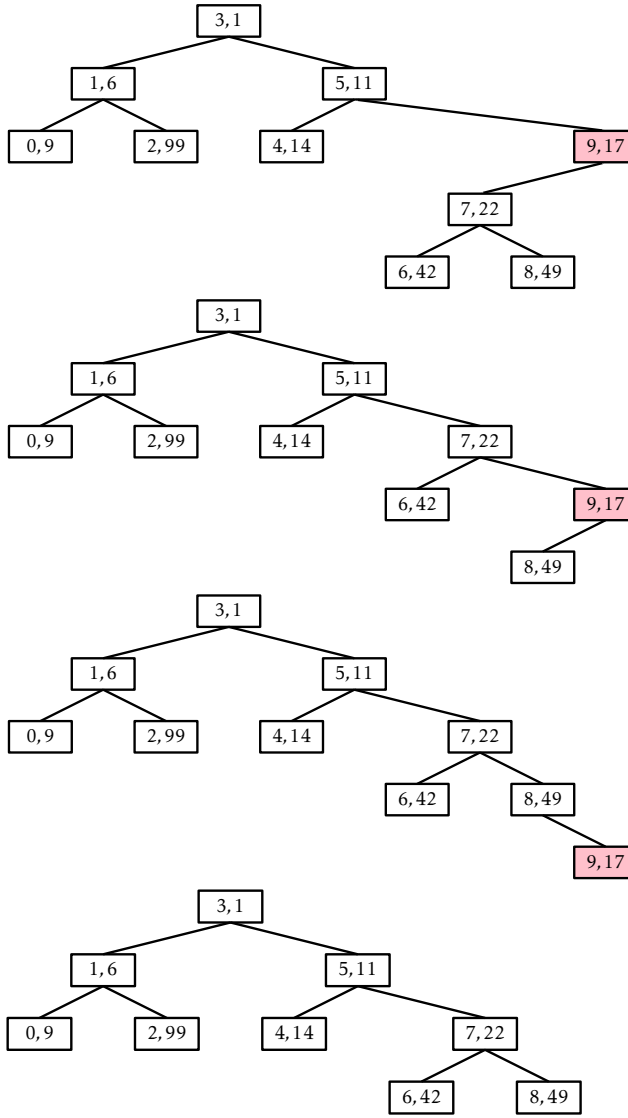


図 7.8: Removing the value 9 from the Treap in Figure 7.5.

も Treap の方がかなり早いと解釈できるだろう。Chapter 4 の Exercise 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。XXX: SkiplistSSet ではない?

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

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。

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。Random binary search trees have been studied extensively. Devroye[19] が Lemma 7.1 とそれに関連した成果を証明した。より強い結果も複数知られているが、もっとも印象的なのは Reed[62] の成果である。この文献ではランダム二分探索木の高さの期待値が次の式になることを示した。

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

ここで $\alpha \approx 4.31107$ は $[2, \infty)$ 範囲における $\alpha \ln((2e/\alpha)) = 1$ の唯一の解であり、$\beta = \frac{3}{2\ln(\alpha/2)}$ である。さらに、高さの分散は定数である。

Treap という名前は Seidel と Aragon[65] が考えた。この文献では Treap といくつかの変種を議論している。しかし、Treap の基本的なデータ構造は Vuillemin[74] が先に研究しており、この文献では Cartesian tree と呼んでいた。

Treap における空間効率の最適化として、各ノードに明示的に優先度 p を蓄えずに済ませる方法がある。代わりに、u の優先度として u のメモリアドレスのハッシュ値を用いる。多くのハッシュ関数が実用的には上手く動作するが、Lemma 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使う必要がある。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。

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

このようなハッシュ関数のクラスであり、実装が簡単で高速なものとして *tabulation hashing* がある。(Section 5.2.3 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、randomized binary search tree がある。これは Martínez と Roura [51] が提案した。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ランダム化されている。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 は部分木の根に移動してくる。
2. そうでなければ（すなわち確率 1−1/(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。

ひとつめの場合は 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 randomized binary search tree 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。

ここでも Treap において u で左右の回転を行う確率と同じであることを簡単に確認できる。

Treap と比べて randomized binary search tree には短所がある。要素の追加・削除の際にたくさんランダムな選択をする必要があり、また部分木の大きさを保持しなければならないのである。randomized binary search tree の長所としては、部分木の大きさは他の便利な目的、例えばランクを $O(\log n)$ の期待実行時間で計算するのに使うことができる点である。（Exercise 7.10 を参照せよ。）一方で Treap の優先度には木のバランスを保つ以外の用途はない。

Exercise 7.1. Figure 7.5 の Treap に 4.5 を優先度 7 で追加し、値 7.5 を優先度 20 で追加する様子を図示せよ。

Exercise 7.2. Figure 7.5 の Treap から 5 と 7 を削除する様子を図示せよ。

Exercise 7.3. Figure 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。）

Exercise 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 $O(n)$ 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。

Exercise 7.5. Lemma 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。）

Exercise 7.6. `Treap` の実装を明示的に優先度を保持しないように修正せよ。その際優先度として、各ノードのハッシュ値を利用せよ。

Exercise 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ふたつの値を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。

Exercise 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。XXX: よくわからん This method should run in $O(n)$ worst-case time and should construct a `Treap` that is indistinguishable from one in which the elements of `a` were added one at a time using the `add(x)` method.

Exercise 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. 各ノードがそれ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` ≤ `x` ≤ `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。

Exercise 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。）

Exercise 7.11. TreapList を実装せよ。これは List インターフェースと 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。

Exercise 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 Exercise 7.10 のために必要である。

Exercise 7.13. split(x) の逆であると考えられる 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。

Exercise 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

Scapegoat Tree

この章では二分探索木の一種である ScapegoatTree を紹介する。このデータ構造は何か誤りがあるとき、それは誰の責任なのかを決めようとする現実でよくある考え方に基づく。(scapegoat とは罪を負わされたヤギ、転じて身代わりのことである。) 責任の所在が決まれば、そいつに問題を解決させることができる。

ScapegoatTree は部分再構築によってバランスを保つ。部分再構築の間に、ある部分木全体が分解され完全にバランスされた部分木として再構築される。ノード u を根とする部分木を完全にバランスされた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 から再帰的にバランスされた木を構築するものだ。m = a.length/2 とするとき、a[m] を新たな部分木の根とし、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格納される。

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 $O(\texttt{size}(u))$ である。結果として得られる部分木は高さ最小のものである。すなわち、`size(u)` 個のノードを持ちこの木より低

い木は存在しない。

ScapegoatTree：部分再構築する二分探索木

ScapegoatTree とは、BinarySearchTree であり、ノード数 n に加えてノード数の上界を保持する q を持つ。

ScapegoatTree

```
int q;
```

n と q は常に次の式を満たす。

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

加えて ScapegoatTree の高さは対数程度である。すなわち scapegoat tree の高さは常に次の値以下である。

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

この制約を満たしても、ScapegoatTree は意外と偏って見た目になりうる。例えば Figure 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 の場合の標準的なもの（Section 6.2 を見よ）を使う。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) の実装では、まず n と q をひとつずつ増やし、x を二分探索木に追加するふつうのアルゴリズムを使う。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、運良く u の深さが $\log_{3/2} q$ 以下なら、これ以上なにもしなくてよい。

depth(u) > $\log_{3/2} q$ であることもある。この場合、高さを減らす必要がある。これはそんなに大変ではない。今、ノード u だけが、木の中で深さが $\log_{3/2} q$ を超えているノードである。u を修正するために、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ドである。ここでバランスは次の式で判断される。

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

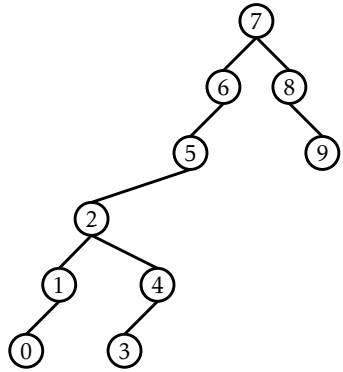


図 8.1: A ScapegoatTree with 10 nodes and height 5.

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 w が見つかれば w を根とする部分木を完全に取り壊し、完全にバランスされた二分探索木として再構築すればよい。binary search tree. We know, from (8.2) より、u を加える前から w の部分木は完全二分木ではない。よって、w を再構築するときにその高さは 1 以上減り、ScapegoatTree の高さは再度 $\log_{3/2} q$ 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))
      w = w.parent;
    rebuild(w.parent);
```

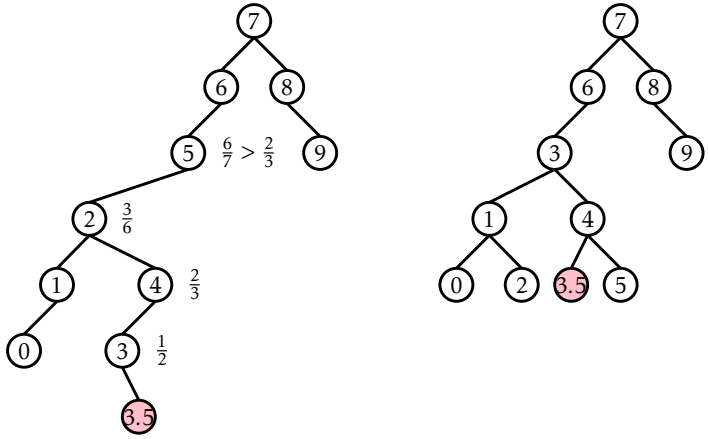


図 8.2: Inserting 3.5 into a ScapegoatTree increases its height to 6, which violates (8.1) since $6 > \log_{3/2} 11 \approx 5.914$. A scapegoat is found at the node containing 5.

```
  }
  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、完全にバランスされた二分探索木にして、$q = n$ 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。

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) が成り立たなくなったなら常に scapegoat を見つけられることを示す。

Lemma 8.1. u を *ScapegoatTree* における深さ $h > \log_{3/2} q$ 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。

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

Proof. 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。

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

また、根から u への経路を $r = u_0, \ldots, u_h = u$ とする。このとき $\texttt{size}(u_0) = n$、$\texttt{size}(u_1) \le \frac{2}{3}n$、$\texttt{size}(u_2) \le \frac{4}{9}n$ であり、より一般に次の式が成り立つ。

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

ここで $\texttt{size}(u) \ge 1$ より次の式が成り立つことを示す。

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

しかしこれは成り立たず、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 size(u)・rebuild(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。

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

Proof. scapegoat ノード w を見つけたあと、そこから再構築を行うコストは $O(\mathtt{size}(\mathtt{w}))$ である。scapegoat を見つけるためには size(u) を $\mathtt{u}_k = \mathtt{w}$ を見つけるまで $\mathtt{u}_0,\ldots,\mathtt{u}_k$ に順に実行する。しかし、$\mathtt{u}_k$ 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。

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

よって、すべての size(u) 呼び出しのコストの合計は次のようになる。

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

最後の行は減少幾何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 rebuild(u) の合計コストの上界を示す。

Lemma 8.3. 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。

Proof. XXX: 訳語は? accounting method のことだろうか? *credit scheme* を使って示す。各ノードは預金を持っていると考える。預金が c だけあれば再

構築のための支払いができる。預金の合計は $O(m \log m)$ で、rebuild(u) は u に蓄えられている預金を使って支払われる。

挿入・削除の際に挿入・削除されるノード 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 $\log_{3/2} q \leq \log_{3/2} m$ 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(u) の支払いに十分であることを示せばよい。

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

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

次の事実を仮定すると、

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

次の式が成り立つ。

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

このとき、さらに次の式が成り立つ。

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

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

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

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

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

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

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

以上で示された。 □

要約

次の定理は ScapegoatTree の性能をまとめるものだ。

Theorem 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。

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 の実装で実験してみると、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

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

であり、これは Skiplist の探索経路の長さの期待値よりも良く、Treap とも遠くないため、この結果は意外かもしれない。最適化として、部分木のサイズを各ノードが保持したり、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)・delete(x) からなる操作の列を ScapegoatTree で実行したとき、他の SSet の実装より遅いことがあるだろう。

この本で扱った他の SSet の実装と異なり ScapegoatTree は再構築自体に多くの時間を消費するため、このような性能差が現れる。この本で紹介した他の SSet の実装では、n 個の操作の間に $O(n)$ 程度のデータ構造の変形をすればよかった。一方、Exercise 8.3 から n 個の操作の列を ScapegoatTree に実行するとき、$n \log n$ のオーダーの時間を rebuild(u) に費やすことがわかる。これは再構築をすべて rebuild(u) で行っていることの帰結である。 [20].

性能は劣るものの、ScapegoatTree を使うのが正しい選択となる場合がある。これは、各ノードに追加のデータがあり、それを回転操作においては定数時間で更新できないが、rebuild(u) 操作の際に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が有効である。このような応用の例を Exercise 8.11 で取り上げている。

Exercise 8.1. Figure 8.1 の ScapegoatTree に 1.5、1.6 を順に追加する様子を描け。

Exercise 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、Lemma 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。

Exercise 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 $cn \log n$ 以上であることを示せ。

Exercise 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。

Exercise 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。

Exercise 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 Exercise 8.5 での実装と、ここでの実装とを性能比較せよ。

Exercise 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、この連結リストを完全にバランスされた二分木に変換せよ。(いずれのステップにも華麗な再帰による実装がある。)

Exercise 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 size(u) ≤ (2/3)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。

Exercise 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完全にバランスされた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。）

Exercise 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完全にバランスされた二分木に再構築される。

そして、DynamiteTree の償却実行時間は $O(\log n)$ であることを示せ。

Exercise 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

この章では赤黒木という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うちのひとつである。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、OS である Linux のカーネルにも使われている。赤黒木が人気である理由を挙げる。

1. n 個の値を持つ赤黒木の高さは $2\log n$ 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。要素 x の追加・削除に必要な主要な時間は x を見つける処理によることを明らかにする。[*1]

しかし、赤黒木のよい性質には代償もある。これは実装の複雑さである。高さの上界を $2\log n$ に保つのは容易ではない。様々な場合についての慎重な解析が必要なのである。すべての場合において、確実に正しい実行をし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のである。

[*1] スキップリストや Treap も平均的にだがこの声質を持っている。4.6 と 7.5 を参照せよ。

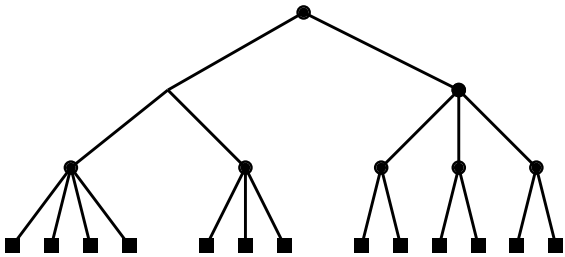


図 9.1: A 2-4 tree of height 3.

赤黒木の実装に直接取り掛かるのでなく、関連するデータ構造である 2-4 木についての背景知識をまずは説明する。こうするとどうやって赤黒木が発見され、なぜこのデータ構造を効率的に管理できるのかを理解する助けになるだろう。

2-4 木

2-4 木は次の性質を持つ根付き木である。

Property 9.1 (height)**.** すべての葉の深さは同じである。

Property 9.2 (degree)**.** すべての内部ノードは 2-4 個の子ノードを持つ。

2-4 の例を Figure 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である。

Lemma 9.1. n 個の葉を持つ *2-4* 木の高さは $\log n$ 以下である。

Proof. 内部ノードの子の数は 2 以上なので、2-4 木の高さを h とすると葉の数は 2^h 以上である。

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

両辺の対数を取ると $h \leq \log n$ である。 □

葉の追加

2-4 木に葉を追加するのは簡単である。(Figure 9.2 を参照せよ。) 下から二番目の深さのノード w の子として葉 u を追加したいとき、単に u を w の子とする。これは高さの制約は保つが、次数の制約を犯すかもしれない。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、w と w′ というそれぞれ 2 つ、3 つずつの子を持つノードとする。このとき w′ には親がいないので、w′ を w の親の子とする。この処理は再帰的に行われる。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、そのときはまた分割を行う。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。

2-4 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。

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。(Figure 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 が子をふたつしか持っていなかったなら、w の子はひとつだけになり、次数の制約を犯すことになる。

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。後者の場合は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。

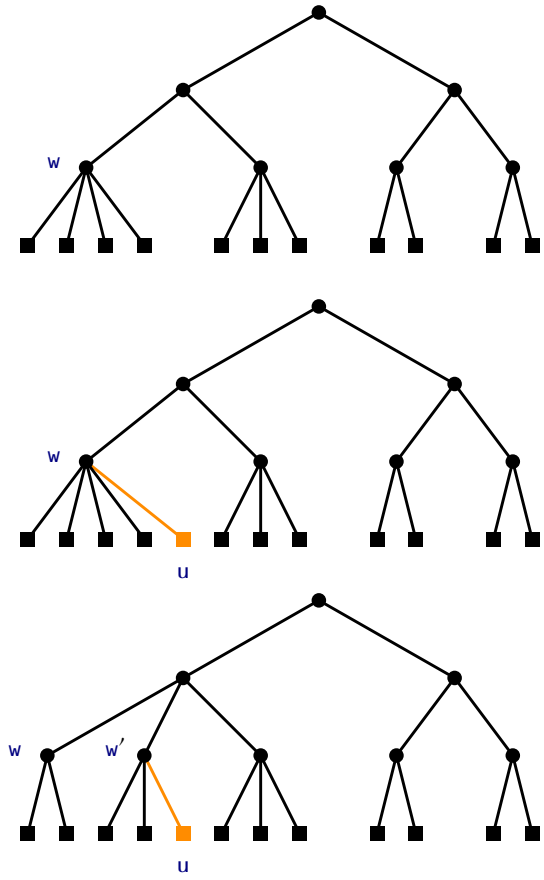


図 9.2: Adding a leaf to a 2-4 Tree. This process stops after one split because w.parent has a degree of less than 4 before the addition.

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。赤は 0 で、黒は 1 で表現される。

RedBlackTree

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

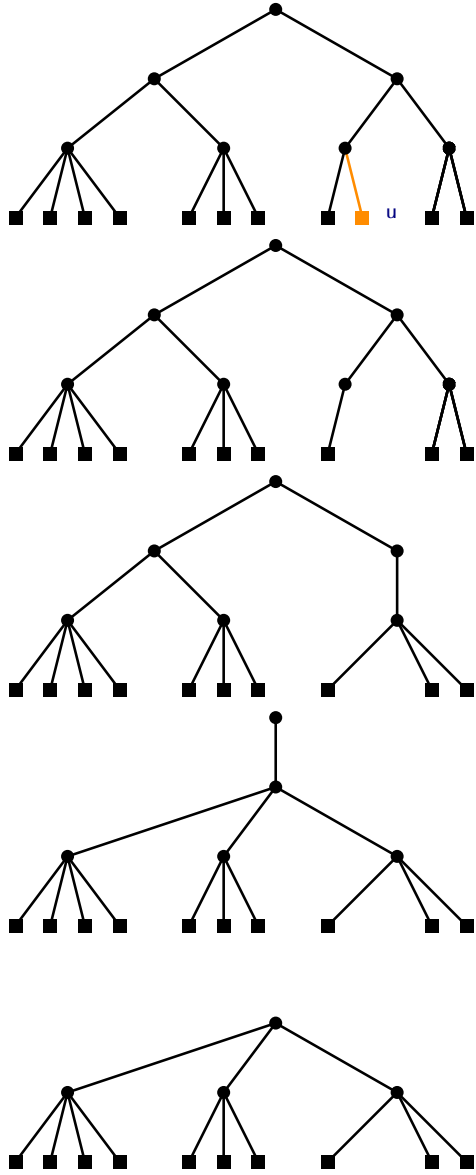


図 9.3: Removing a leaf from a 2-4 Tree. This process goes all the way to the root because each of u's ancestors and their siblings have only two children.

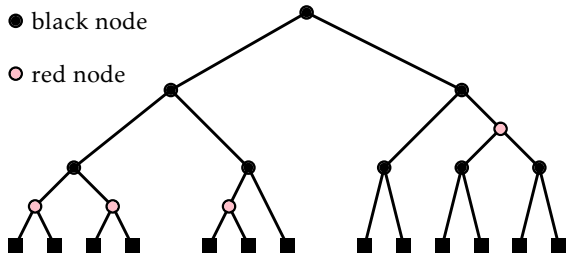


図 9.4: An example of a red-black tree with black-height 3. External (nil) nodes are drawn as squares.

```
  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。

Property 9.3 (black-height)**.** 「黒い高さ」が一様：任意の葉から根への経路上には、同じ数だけ黒いノードがある。

Property 9.4 (no-red-edge)**.** 赤い辺が無い：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、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。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。Figure 9.4 に赤黒木の例を示した。

赤黒木と 2-4 木

前の小節で定義した黒い高さと赤い辺についての性質を、赤黒木が効率的のは一見して驚くかもしれない。一方、これらの性質がなんの役に立つのかよくわからないかもしれない。しかし、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。

Figure 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。

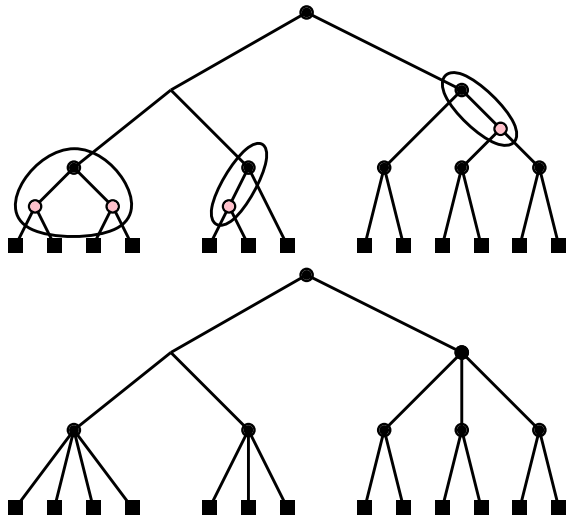


図 9.5: Every red-black tree has a corresponding 2-4 tree.

すべての赤いノード 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、変換後も 4 つの黒い子を持つ。加えて黒い高さの性質より、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 木になのである！

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上は赤いので、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、任意の内部ノードから根への経路のうち最長のものの長さは次の値以下である。

$$2\log(n+1) - 2 \leq 2\log n$$

このことから、赤黒木の最も重要な性質を示せる。

Lemma 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 つの個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。このような例を Figure 9.6 に示した。

同様に remove(x) をするためには、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 Figure 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。

もちろん赤い辺の制約、黒い高さの制約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は考えなければならない場合分けはやはり多いのである。背景にある 2-4 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。

Left-Leaning Red-Black Trees

XXX: left-learning に定訳はあるか?

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、赤い辺の制約・黒い高さの制約を保てる、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達成する。ここでは、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、特にある追加の性質を満たすものの実装である。

Property 9.5 (left-leaning)**.** 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 Figure 9.4 の left-leaning 性を満たしていない。右に進む経路の赤いノードの親がこの性質を犯しているためだ。

left-leaning 性を保持するのは、これにより add(x)・remove(x) において木

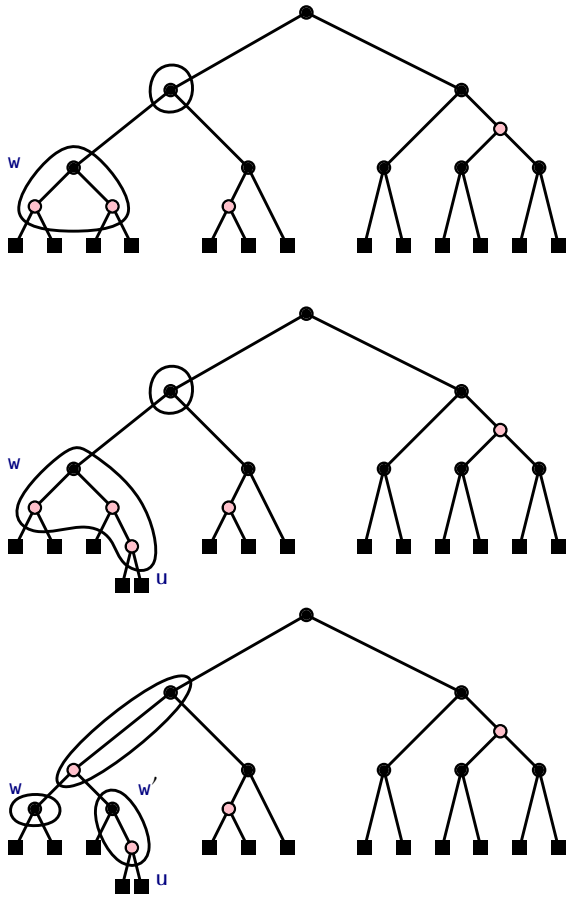


図 9.6: Simulating a 2-4 tree split operation during an addition in a red-black tree. (This simulates the 2-4 tree addition shown in Figure 9.2.)

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 木の表現が一意に定まるためである。次数が 2 のノードはふたつの黒い子を持つ黒いノードになる。次数が 3 のノードは左の子が赤、右の子が黒の黒いノードになる。次数が 4 のノードはふたつの黒い子を持つ赤いノードになる。

add(x)・remove(x) の実装の詳細に入る前に、Figure 9.7 に図示した単純なサブルーチンを説明する。最初のふたつのサブルーチンは黒い高さの制約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

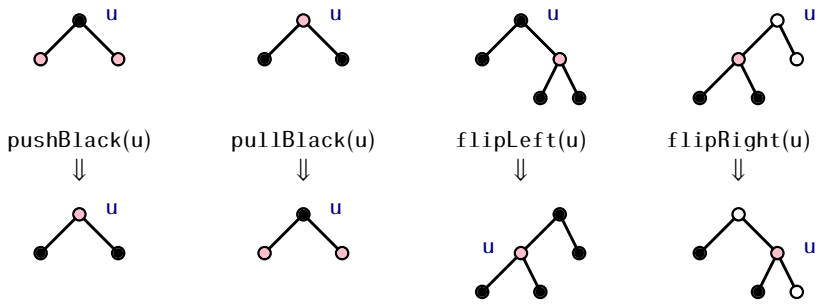


図 9.7: Flips, pulls and pushes

い子を持つ黒いノードで、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 left-leaning 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い高さ・赤い辺の制約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。

XXX: 図のノードが一部白抜きなのはなぜ?

RedBlackTree

```
void flipRight(Node<T> u) {
  swapColors(u, u.left);
  rotateRight(u);
}
```

要素の追加

RedBlackTree における 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。この処理は任意のノードの黒い高さは変わらないので黒い木の制約は破られない。しかし、left-leaning 性を犯すことがあるかもしれない。（これは u が右の子であるときである。）また赤い辺の制約を犯すこともあるかもしれない。（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は addFixup(u) を呼べばよい。

RedBlackTree

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

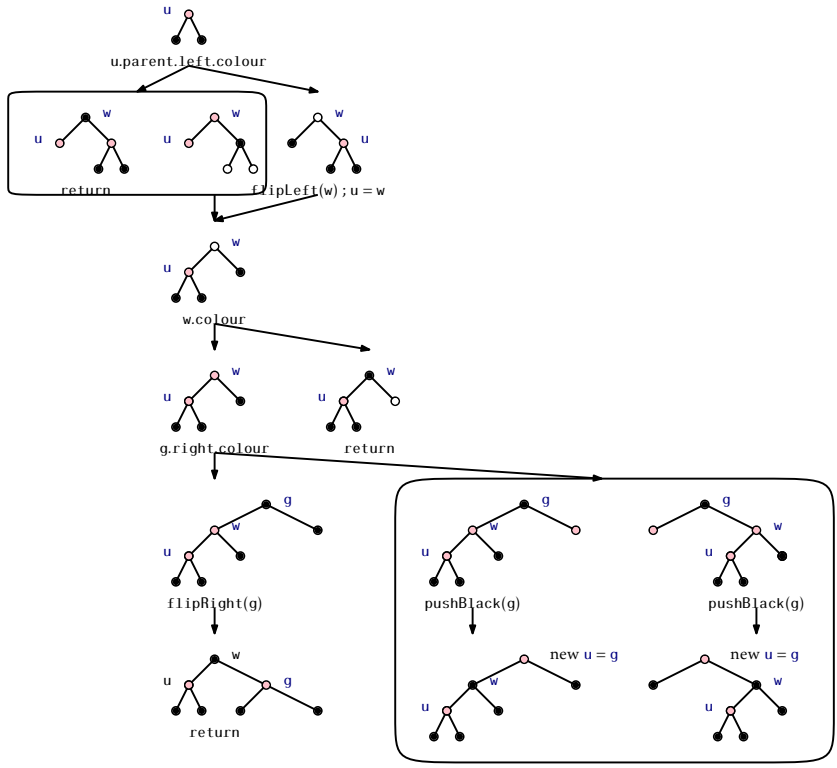


図 9.8: A single round in the process of fixing Property 2 after an insertion.

```
}
```

Illustrated in Figure 9.8 に図示したように、addFixup(u) 赤いノード u を入力に取るが、これは赤い辺の制約や left-leaning 性を犯しているかもしれない。この先の議論は、Figure 9.8 を見るかそれを再度紙に書いてみるかでもしないと理解できないと思う。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を成り立たせることができる。u の兄弟も赤いなら u の親は黒で、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 left-leaning 性を犯しているかを確認し、もしそうなら flipLeft(w) を実装し、u = w とする。この結果、

次の状態になる。u は親である w の左の子になり、そのため w は left-leaning 性を満たすようになる。あとは u の赤い辺の制約が満たされることを示せばよい。w が黒いなら既に赤い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。

まだ制約が破られているので、u と w はいずれも赤い。赤い辺の制約（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 left-leaning 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 について赤い辺の制約が満たされるが、g について赤い辺の制約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 が赤い辺の制約を満たすこと、g が left-leaning 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。

Removal

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。

このとき問題となるのは、w が黒なら黒い高さの制約が w.parent で破られることだ。w.colour を u.colour に足すとこの問題を一時的に解決できる。もちろんこの結果ふたつの別の問題が発生する。(1) もし u と w が共に黒いと u.colour+w.colour = 2 であり、これは不正な色になってしまう。XXX: (2)? w が赤いとき、u と入れ替えると u.parent の left-leaning 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（無向な色）である。u の色が 2 なら、removeFixup(u) は一連の回転と塗り替え操作を実行し、このノードが木から追い出す。ノード u が更新しようとしている部分木の根になるまで処理を行う。XXX: I'm 迷子 During this process, the node u changes until, at the end of this process, u refers to the root of the subtree that has been changed. The root of this subtree may have changed colour. In particular, it may have gone from red to black, so the removeFixup(u) method finishes by checking if u's parent violates the left-leaning property and, if so, fixing it.

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);
    }
  }
}
```

Figure 9.9 は removeFixup(u) を図示したものだ。この説明も Figure 9.9 を見ながら出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 のノード u は次の 4 つの場合分けに基づき処理される。
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。u を黒に塗り直せばよい。（これは赤黒木のいずれの制約を犯すこともない。）
Case 1: u の兄弟 v が赤い場合である。このとき left-leaning 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 left-leaning 性を犯すようになり、u の深さが 1 大きくなることに注する。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 を実行すると、処理がうまく終了することがわかるだろう。

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 left-leaning 性を満たしておらず、flipLeft(w) を読んでこれを解決する。

At this point, w is red and v is the root of the subtree with which we

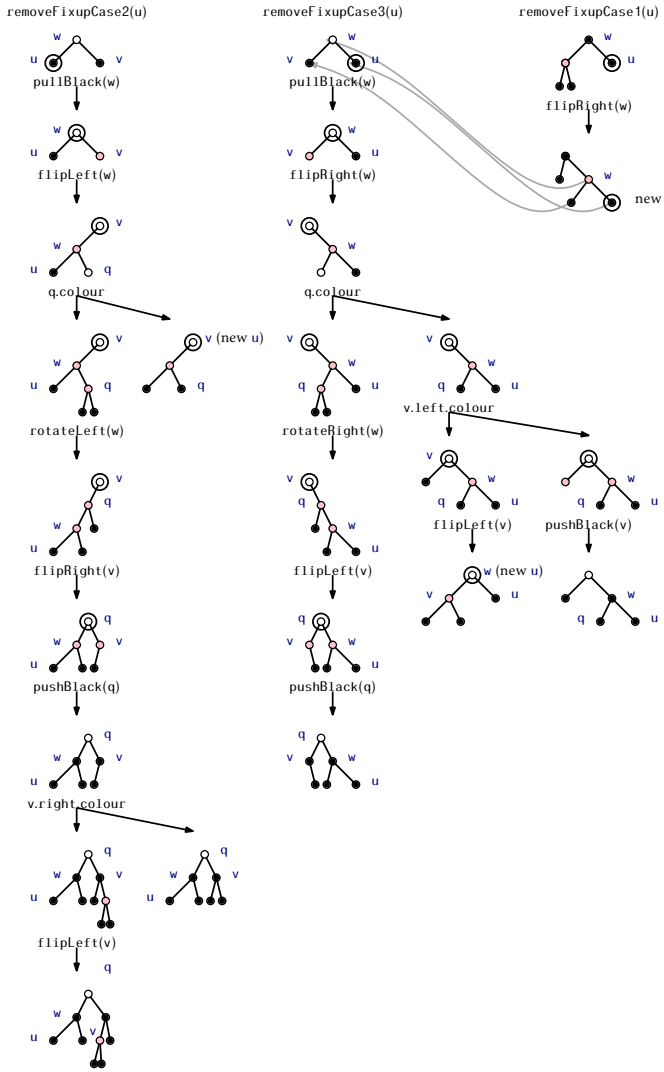


図 9.9: A single round in the process of eliminating a double-black node after a removal.

started. We need to check if w causes the no-red-edge property to be violated. We do this by inspecting w's right child, q. If q is black, then w satisfies the no-red-edge property and we can continue the next iteration with u = v.

Otherwise (q is red), so both the no-red-edge property and the left-leaning properties are violated at q and w, respectively. The left-leaning property is restored with a call to `rotateLeft(w)`, but the no-red-edge property is still violated. At this point, q is the left child of v, w is the left child of q, q and w are both red, and v is black or double-black. A `flipRight(v)` makes q the parent of both v and w. Following this up by a `pushBlack(q)` makes both v and w black and sets the colour of q back to the original colour of w.

At this point, the double-black node is has been eliminated and the no-red-edge and black-height properties are reestablished. Only one possible problem remains: the right child of v may be red, in which case the left-leaning property would be violated. We check this and perform a `flipLeft(v)` to correct it if necessary.

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's sibling is black and u is the right child of its parent, w. This case is symmetric to Case 2 and is handled mostly the same way. The only differences come from the fact that the left-leaning property is asymmetric, so it requires different handling.

As before, we begin with a call to `pullBlack(w)`, which makes v red and u black. A call to `flipRight(w)` promotes v to the root of the subtree. At this point w is red, and the code branches two ways depending on the colour of w's left child, q.

If q is red, then the code finishes up exactly the same way as Case 2 does, but is even simpler since there is no danger of v not satisfying the left-leaning property.

The more complicated case occurs when q is black. In this case, we examine the colour of v's left child. If it is red, then v has two red children and its extra black can be pushed down with a call to `pushBlack(v)`. At this point, v now has w's original colour, and we are done.

If v's left child is black, then v violates the left-leaning property, and we restore this with a call to `flipLeft(v)`. We then return the node v so that the next iteration of `removeFixup(u)` then continues with u = v.

RedBlackTree

```
Node<T> removeFixupCase3(Node<T> u) {
  Node<T> w = u.parent;
  Node<T> v = w.left;
  pullBlack(w);
  flipRight(w); // w is now red
  Node<T> q = w.left;
  if (q.colour == red) { // q-w is red-red
    rotateRight(w);
    flipLeft(v);
    pushBlack(q);
    return q;
```

```
  } else {
    if (v.left.colour == red) {
      pushBlack(v); // both v's children are red
      return v;
    } else { // ensure left-leaning
      flipLeft(v);
      return w;
    }
  }
}
```

Each iteration of removeFixup(u) takes constant time. Cases 2 and 3 either finish or move u closer to the root of the tree. Case 0 (where u is the root) always terminates and Case 1 leads immediately to Case 3, which also terminates. Since the height of the tree is at most $2\log n$, we conclude that there are at most $O(\log n)$ iterations of removeFixup(u), so removeFixup(u) runs in $O(\log n)$ time.

要約

次の定理は RedBlackTree の性能をまとめたものだ。

Theorem 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 $O(\log n)$ である。

加えて次の定理も成り立つ。

Theorem 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。

Theorem 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。とくに 2-4 木に

おける分割・併合・borrowing に必要な時間が $O(m)$ であることを示せれば、これは Theorem 9.2 を含むだろう。

2-4 木に対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。[*2]2-4 木の内部ノード u のポテンシャルを次のように定義する。

$$\Phi(u) = \begin{cases} 1 & \text{if } u \text{ has 2 children} \\ 0 & \text{if } u \text{ has 3 children} \\ 3 & \text{if } u \text{ has 4 children} \end{cases}$$

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。分割が起こるとき、4 つの子を持つノードがそれぞれ 2 つと 3 つの子を持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。併合が起こるとき、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、分割や併合が起きるときは、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、ポテンシャルは最大で 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、ポテンシャルは最大で 1 増える。また borrowing にはふたつのノードが関連し、それらのポテンシャルは最大で 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最大で 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 $3m/2$ 回までしか行われない。Theorem 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でもある。

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化の赤黒木を説明している。

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

Andersson [6] は left-learning 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 [64] は二種類の left-learning 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 $|\varphi^h/\sqrt{5} - F(h)| \leq 1/2$ である。）Lemma 9.1 の証明で述べたように、これは式を含意する。

$$h \leq \log_\varphi n \approx 1.440420088 \log n$$

よって、AVL 木の高さは赤黒木の高さよりも低い。`add(x)`・`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。Figure 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 Theorem 9.2 のような保証はない。

Exercise 9.1. Figure 9.11 の RedBlackTree に対応する 2-4 木を図示せよ。

Exercise 9.2. Figure 9.11 の RedBlackTree に 13, 3.5, 3.3 を順に追加する様子を図示せよ。

Exercise 9.3. Figure 9.11 の RedBlackTree から 11, 9, 5 を順に削除する様子を図示せよ。

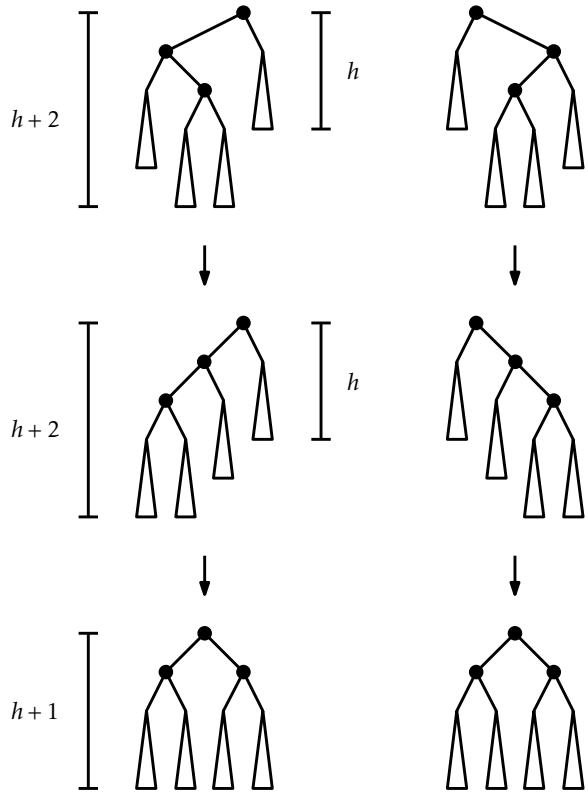


図 9.10: Rebalancing in an AVL tree. At most two rotations are required to convert a node whose subtrees have a height of h and $h+2$ into a node whose subtrees each have a height of at most $h+1$.

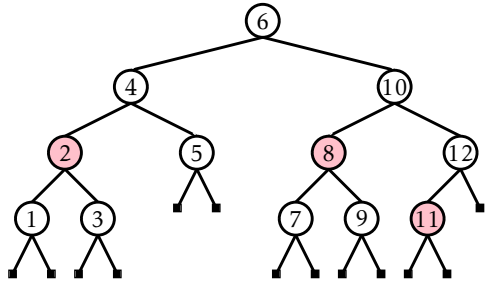


図 9.11: A red-black tree on which to practice.

Exercise 9.4. どれだけ大きな n についても、n 個のノードを持ち、高さが $2\log n - O(1)$ である赤黒木が存在することを示せ。

Exercise 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。

Exercise 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 $2\log n - O(1)$ 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。

Exercise 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。

Exercise 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。

Exercise 9.9. n 個のノードを持ち、高さが $2\log n - 2$ 以下の二分探索木があるとする。このとき、この木のすべてのノードを、黒い高さの制約と赤い辺の制約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 left-learning 性を満たすことはできるか。

Exercise 9.10. 赤黒木 T_1, T_2 は黒い高さがいずれも 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。

Exercise 9.11. Exercise 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。

Exercise 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。Figure 9.10 を参照せよ。）また、`remove(x)` 操作の際にオーダーで $\log n$ 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。

Exercise 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。

Exercise 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

ヒープ

この章では優先度付きキューという役に立つデータ構造のふたつの実装を説明する。いずれも特殊な二分木であり、ヒープ（乱雑に積まれたもの）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。この効率的な実装はヒープソート（Section 11.1.3 参照）という名の、最速の整列アルゴリズムのうちのひとつの基礎になっている。ふたつめの実装はより柔軟である。これはある優先度付きキューの要素を別の優先度付きキューに取り込む `meld(h)` 操作を提供する。

BinaryHeap：暗黙の二分木

（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に配列に入れて完全二分木を表現する。（Section 6.1.2 を参照せよ。）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。Figure 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。

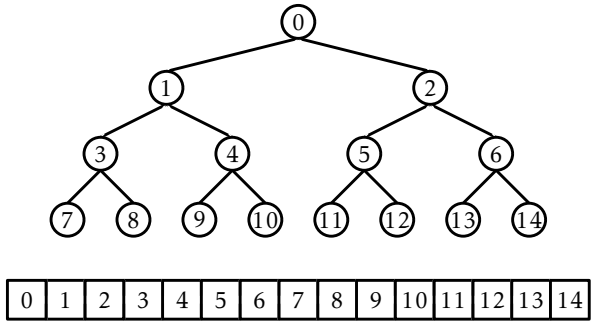


図 10.1: Eytzinger's method represents a complete binary tree as an array.

BinaryHeap

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

BinaryHeap はこの手法を使って完全二分木を暗黙に表現する。特に、この木の中では要素はヒープ順に並んでいる。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。Figure 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() {
```

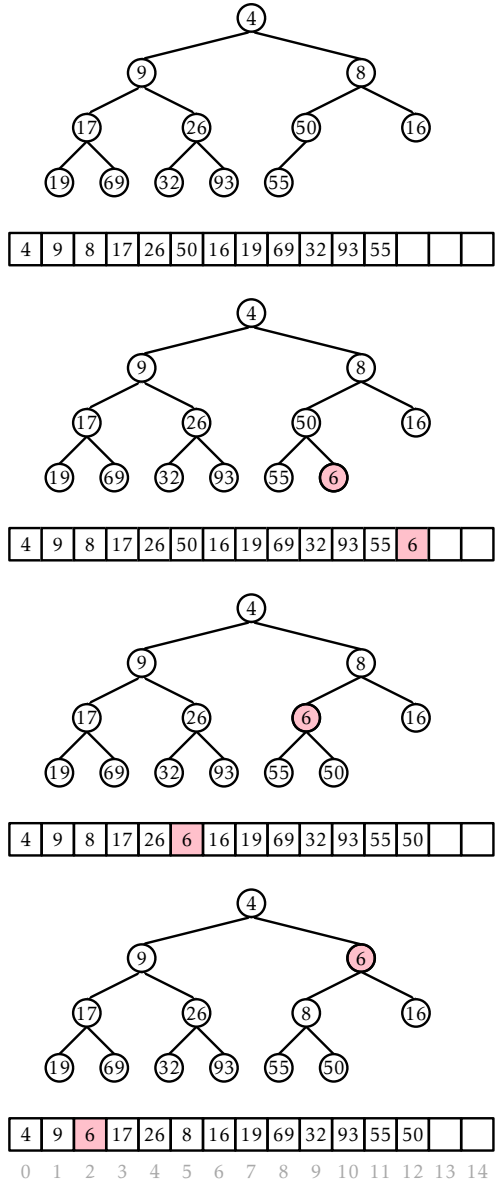


図 10.2: Adding the value 6 to a `BinaryHeap`.

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

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 Lemma 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（暗黙の）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

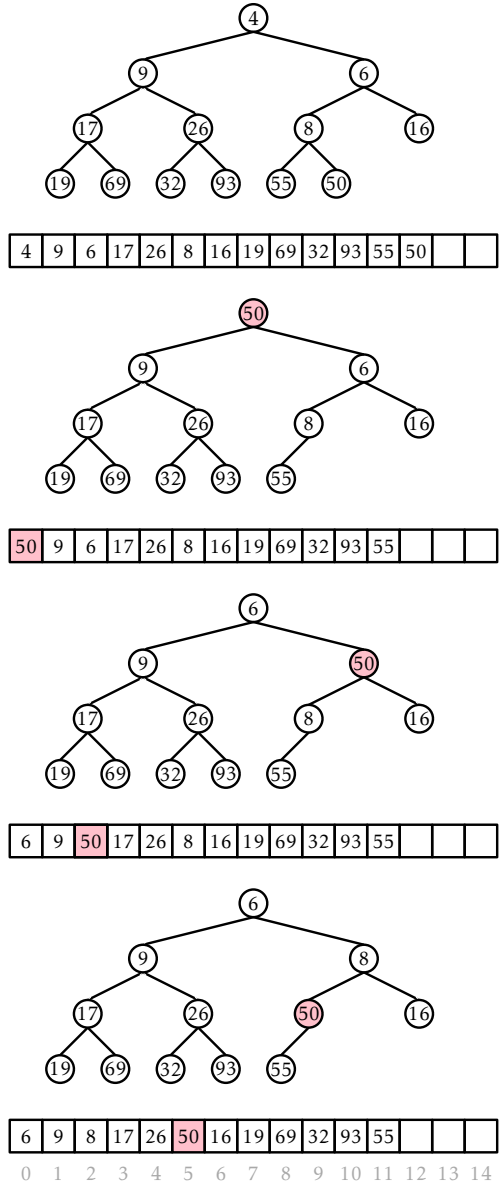


図 10.3: Removing the minimum value, 4, from a `BinaryHeap`.

ドがある。よって、h を木の高さとすると、少なくとも 2^h 個のノードがある。言い換えると、次の式が成り立つ。

$$n \geq 2^h$$

両辺の対数を取ると、次の式が成り立つ。

$$h \leq \log n$$

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

要約

次の定理は BinaryHeap の性能をまとめたものだ。

Theorem 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 $O(\log n)$ 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。

MeldableHeap：ランダムな Meldable ヒープ

XXX: melbadle は定訳がある?
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 を引数にとり、それぞれを根とするふたつの部分木内のすべてのノードを含むヒープの根を返す。
嬉しいことに merge(h1,h2) は再帰的に定義できる。Figure 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、h1.x ≤ 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は h1.x を含む。h2 は再帰的に h1.left か h1.right の望む方と併合してよい。ここでランダム性の出番だ。コインを投げ、h 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。

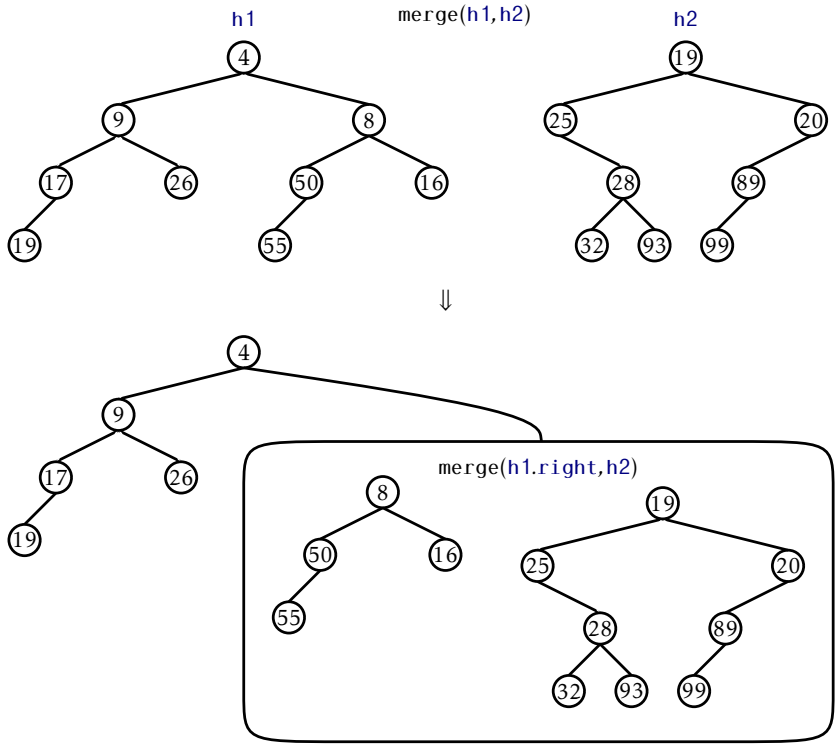


図 10.4: Merging h1 and h2 is done by merging h2 with one of h1.left or h1.right.

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 $O(\log n)$ である。

さらに MeldableHeap は他の色々な操作も $O(\log n)$ の期待実行時間で実装できる。例えば次のものがある。

- 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。

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の右か左の子に進む。木からはみ出ると処理を終了する。（これは今のノードが nil になったときである。）

次の補題は注目に値する。これは二分木の形状に関わらず成り立つためである。

Lemma 10.1. n 個のノードからなる二分木におけるランダムウォークの長さの期待値は $\log(n+1)$ 以下である。

Proof. n に関する帰納法により証明する。$n = 0$ のとき、ステップ数は $0 = \log(n+1)$ である。以下では、任意の非負整数 $n' < n$ について、示したい補題が成り立つと仮定する。

n_1 を根の左の部分木の大きさとし、$n_2 = n - n_1 - 1$ を根の右の部分木の大きさとする。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。

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

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 は凹関数なので、$\mathrm{E}[W]$ は $n_1 = n_2 = (n-1)/2$ で最大値を取る。よって、ランダムウォークのステップ数の期待値は次のようになる。

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

$$= 1 + \log((n+1)/2)$$
$$= \log(n+1)\ .$$ □

余談だが、情報理論について知っている読者は Lemma 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。

Information Theoretic Proof of Lemma 10.1. d_i を i 番目の外部ノードとする。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すこと。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。

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

右辺は n + 1 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(n + 1) を超えない。よって補題が示された。 □

ランダムウォークに関するこの結果より、merge(h1,h2) の実行時間の期待値は $O(\log n)$ であることを示せる。

Lemma 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 n_1, n_2 個のノードを含むとする。このとき、merge(h1,h2) の実行時間は $O(\log n)$ 以下である。ただし、ここで $n = n_1 + n_2$ である。

Proof. 併合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。

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

要約

次の定理は MeldableHeap をまとめるものである。

Theorem 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 $O(\log n)$ である。

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暗黙に表現する方法は Eytzinger [27] が最初に提案したようである。彼は高貴な一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 [76] 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 [73]、Fibonacci heaps [30]、pairing heaps [29]、skew heaps [70] 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 $y \leq u.x$ 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]

Exercise 10.1. Figure 10.2 に示した BinaryHeap に 7, 3 を順に追加する様子を図示せよ。

Exercise 10.2. Figure 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。

Exercise 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。

Exercise 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。

Exercise 10.5. Exercise 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。

Exercise 10.6. Figure 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。

Exercise 10.7. Figure 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。

Exercise 10.8. MeldableHeap からノード u を削除する remove(u) を実装せよ。ただし、このメソッドの実行時間の期待値は $O(\log n)$ でなければならない。

Exercise 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。

Exercise 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。）

Exercise 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

整列アルゴリズム

この章では n 個の要素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にこの題材が入っているのは変に思うかもしれないが、これにはいくつか理由がある。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と深い関係がある。（ランダム二分探索木とヒープ）

最初の小節では比較だけを使った整列アルゴリズムであって、実行時間が $O(n \log n)$ であるものを 3 つ紹介する。またこれらの 3 つはいずれも漸近的に最適であることも示す。つまり比較だけを使うアルゴリズムでは、最低でも $n \log n$ 回程度の比較が最悪の場合でも、また平均的な場合であっても必要なのである。

ここで、これまでの章で説明した SSet や優先度付き Queue の実装はいずれも $O(n \log n)$ の時間で整列アルゴリズムを実装するのに使えることを注意しておく。例えば n 個の要素を、BinaryHeap または MeldableHeap における n 回の add(x) に続く n 回の remove() 操作で整列できる。代わりに n 回の add(x) を二分探索木のどれかに実行し、そのあと行きがけ順（Exercise 6.8）で要素を整列された順に取り出すことができる。しかしいずれの場合も完全に活用するわけでないデータ構造を構築するための無駄がかなり生じる。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話が変わることを見ていく。実際に配列のランダムアクセスを使って、$\{0,\ldots,n^c-1\}$ の要素である n 個の整数の整列を $O(cn)$ の時間で実行できることを説明する。

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。Their second argument, c, is a Comparator that implements the compare(a,b) method. 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。Section 1.2.4 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。

マージソート

マージソートは再帰的な分割統治法の古典的な例である。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、a を半分ずつ a0 = a[0],...,a[n/2 − 1] と a1 = a[n/2],...,a[n − 1] 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、完全に整列された配列 a を得る。

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);
}
```

Figure 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。

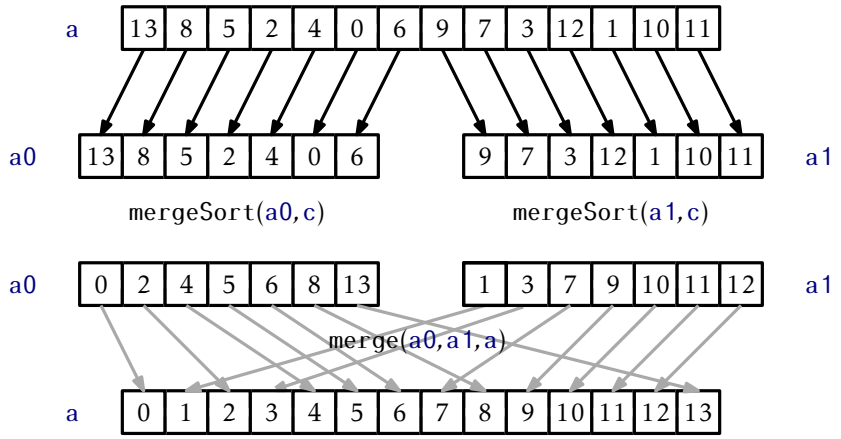


図 11.1: The execution of mergeSort(a,c)

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++];
  }
}
```

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。

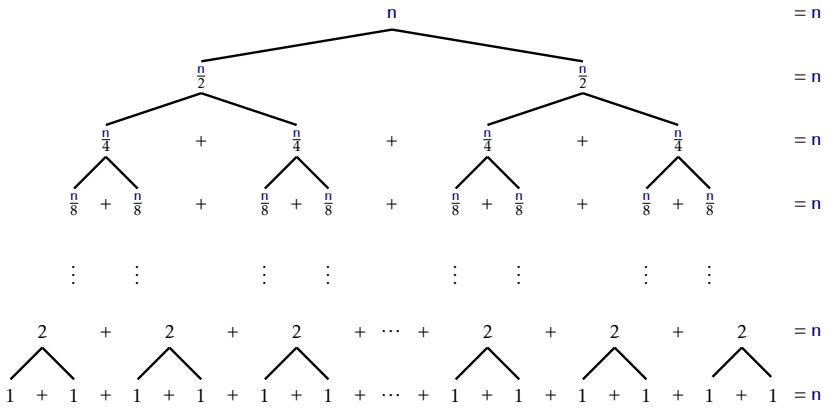


図 11.2: The merge-sort recursion tree.

マージソートの実行時間を求めるためには再帰木のことを考えるのがよい。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。Figure 11.2 を参照せよ。マージソートは n 個の要素の整列問題を、$n/2$ 個の要素の並び替え問題ふたつに変換する。これらの部分問題はそれぞれふたつの問題に変換され、合計 4 つの大きさ $n/4$ の問題になる。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。最終的には $n/2$ 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。each of size one. For each subproblem of size $n/2^i$ の部分問題について、これを併合し、データを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。

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

次の定理の証明は先程の解析に基づくが、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。

Theorem 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。

Proof. 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。

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

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 $x \ge 1$ について次の式が成り立つ。

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

また、任意の $x \ge 1/2$ について次の式が成り立つ。

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

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。

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

□

クイックソート

クイックソートはもうひとつの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仕事を行う。

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、ひとつめとみっつめの分割を再帰的に整列する。Figure 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);
  quickSort(a, q, n-(q-i), c);
```

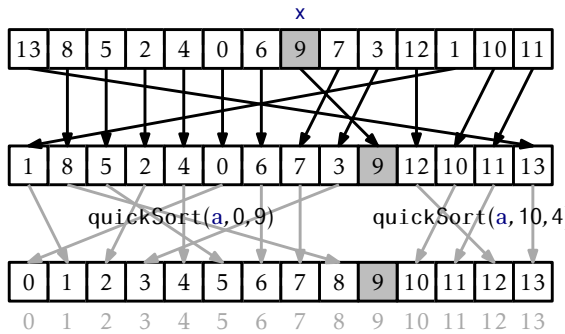


図 11.3: An example execution of quickSort(a, 0, 14, c)

```
}
```

XXX: in-place 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その場で実行され、quickSort(a, i, n, c) は a[i],..., 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、その場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 Section 7.1 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はランダム二分探索木なのである。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、直ちに全要素 x と比較し、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、再帰的

に配列の前方・後方をそれぞれ整列する。一方、ランダム二分木では再帰的に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繰り返す。

ランダム二分木とクイックソートとの上の対応から、Lemma 7.1 をクイックソートの場合に置き換えて考えてみる。

Lemma 11.1. クイックソートは整数 $0,\dots,n-1$ 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 $H_{i+1}+H_{n-i}$ 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。

Theorem 11.2. n 個の相異なる要素をクイックソートで整列するとき、実行される比較の回数の期待値は $2n\ln n+O(n)$ 以下である。

Proof. 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。Lemma 11.1 と期待値の線形性より次が成り立つ。

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

□

Theorem 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。

Theorem 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。

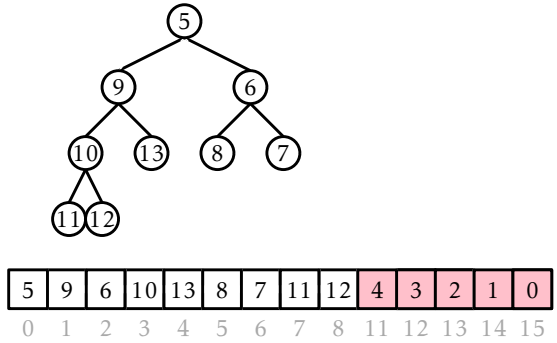


図 11.4: A snapshot of the execution of heapSort(a,c). The shaded part of the array is already sorted. The unshaded part is a BinaryHeap. During the next iteration, element 5 will be placed into array location 8.

ヒープソート

XXX: in-place ヒープソートも処理をその場で行う整列アルゴリズムである。ヒープソートは Section 10.1 で説明した二分ヒープを使う。BinaryHeap 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、その後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。More specifically, a heap stores n elements in an array, a, at array locations 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。[*1]Figure 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) {
  BinaryHeap<T> h = new BinaryHeap<T>(a, c);
```

[*1] 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

```
  while (h.n > 1) {
    h.swap(--h.n, 0);
    h.trickleDown(0);
  }
  Collections.reverse(Arrays.asList(a));
}
```

ヒープソートの中で肝に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 BinaryHeap の add(x) を繰り返し実行すれば $O(n \log n)$ の時間でこれを行うのは容易いだろう。しかし、もっと巧みなボトムアップなやり方がある。二分ヒープにおいて、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 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。n/2 個の要素のためにはなにもする必要がなく、n/4 個の要素のために a[i] を根とする部分ヒープに対して trickleDown(i) を一度呼び、n/8 個の要素のために、この高さの部分ヒープに対して trickleDown(i) を二度呼び... これを繰り返す。trickleDown(i) によって実行される処理は a[i] を根とする部分ヒープの高さに比例するので、全体として必要な処理の量は高々

次の値である。

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

最後から二番目の等号は、期待値の定義より、$\sum_{i=1}^{\infty} i/2^i$ とコインを投げる（表が出る回も含む）回数が等しいことと、Lemma 4.2 から成り立つ。

次の定理は heapSort(a, c) の性能を説明する。

Theorem 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。

Proof. 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。

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

3 つのステップにおいて実行される比較の回数を足し合わせると、証明が完成する。 □

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、$n \log n$ 回程度の比較を必ずすることになるのである。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 Exercise 11.11 とした。）

まずは決定的な n を固定し、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のことを考える。このようなアルゴリズムで n 個の相異なる要

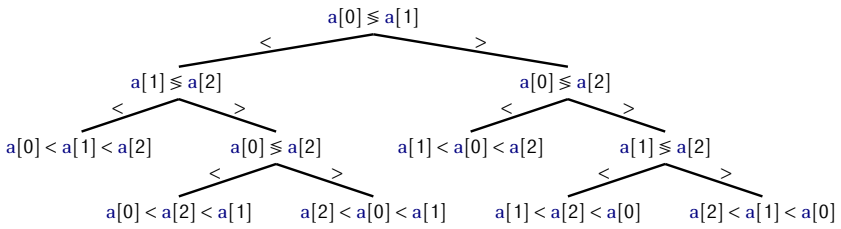


図 11.5: A comparison tree for sorting an array $a[0], a[1], a[2]$ of length $n = 3$.

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 Figure 11.5 に示す。

整列アルゴリズムの比較木から必要なことはすべてわかる。n 個の相異なる要素を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。そうするとアルゴリズムは少なくともそれらの置換を正しくは整列できないことになる。

例えば Figure 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 $a[1] = 1, a[2] = 2, a[0] = 3$ を出力する。しかし入

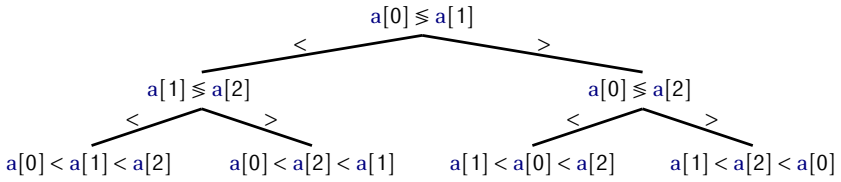


図 11.6: A comparison tree that does not correctly sort every input permutation.

力 3,2,1 の出力は $a[1] = 2, a[2] = 1, a[0] = 3$ 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。

Theorem 11.5. 任意の決定的な比較に基づく整列アルゴリズムと、任意の整数 $n \geq 1$ について、ある長さ n の入力配列 a が存在し、$\mathcal{A}$ が a を整列するとき $\log(n!) = n \log n - O(n)$ 回の比較を実行する。

Proof. これまでの議論から $\mathcal{A}$ の比較木は少なくとも $n!$ 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 $\log(n!)$ 以上の葉 w が存在し、ある入力配列 a が存在し、この葉に到達する。この a に対して $\mathcal{A}$ は $\log(n!)$ 回以上の比較を実行する。 □

Theorem 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 $\log(n!)$ 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。

XXX: decision tree の訳語以下の議論では決定木は次の意味で「散らかっていない」と仮定する。どのような入力配列 a によっても到達できないノードは削除される。このとき木にはちょうど $n!$ 個だけの葉が含まれる。葉の数が $n!$ 以上なのは、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 $n!$ 以下なのは、n 個の相異なる要素の置換は $n!$ 通りであり、それぞれが決定木における根から葉への経路をちょうど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 $\{1,\ldots,n\}$ の置換からランダムに選ぶのは、$\mathcal{T}(\hat{b})$ の 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。

Exercise 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 $\log(n!)$ 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。

Theorem 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 $\log(n!) = n\log n - O(n)$ 以上である。

計数ソートと基数ソート

この節では比較に基づいたものでないアルゴリズムをふたつ紹介する。小さい整数に特化し、要素 (の一部) を配列の添え字として使うことで、これらのアルゴリズムは Theorem 11.5 の下界に縛られない。次の文を考えよう。

$$\texttt{c[a[i]]} = 1$$

この文は定数時間で実行できるが、a[i] の値により c.length の値は異なる。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、これこそがこの節のアルゴリズムが比較に基づくアルゴリズムよりも速く整列をこなせる理由なのである。

計数ソート

0,...,k − 1 の範囲の要素 n 個からなる入力の配列 a があるとする。計数ソートはカウンタの補助配列 c 使って a を整列する。そして、補助配列 b として

整列された a を返す。

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 Figure 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。

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

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。

Theorem 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。

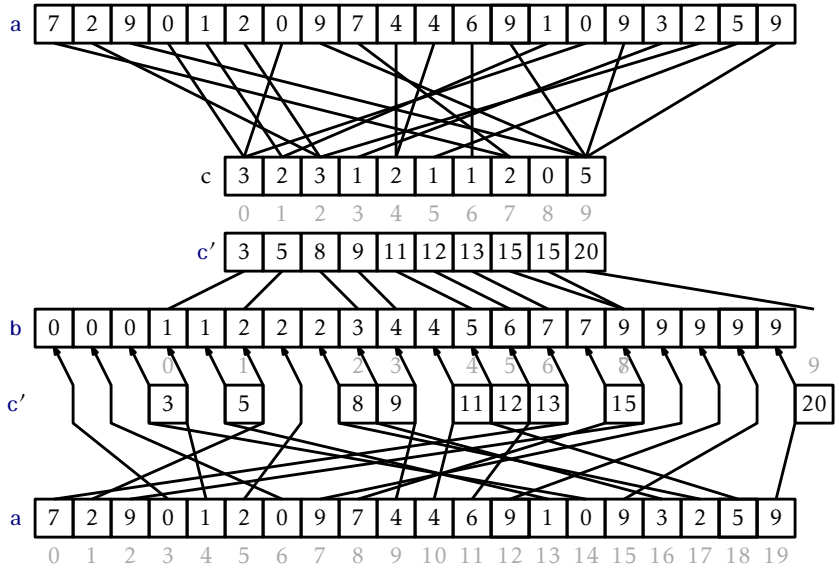


図 11.7: The operation of counting sort on an array of length $n = 20$ that stores integers $0, \ldots, k-1 = 9$.

計数ソートは安定性という良い性質を持つ。これは等しい要素の相対的な位置を保つ性質である。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。

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 $k-1$ と比べてかなり小さくないなら、非常に効率的である。今から説明する基数ソートは複数回の計数ソートにより、もっと大きな範囲の最大値があっても大丈夫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。[*2] 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

[*2] d は w を割り切れると仮定する。もしそうでないときは w を $d\lceil w/d \rceil$ に増やす必要がある。

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 (a[i]>>d*p)&((1<<d)−1) と書いて、a[i] の 2 進数表記において (p+1)d−1,...,pd ビット目である整数を抽出している。）このアルゴリズムの例を Figure 11.8 に示した。

この注目に値するアルゴリズムが正しく整列を行うのは、計数ソートが安定なソートアルゴリズムであるからである。a のふたつの要素 x と y が x < y を満たし、x と y が添え字 r の位置で異なるなら、$\lfloor r/d \rfloor$ 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 $O(n+2^d)$ である。よって、基数ソートの性能は次の定理のようになる。

Theorem 11.8. 任意の整数 d > 0 について、radixSort(a,k) は n 個の w ビット整数を含む配列 a を $O((w/d)(n+2^d))$ の時間で整列する。

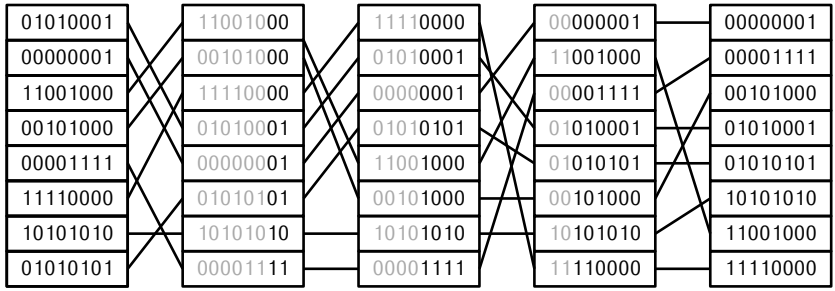


図 11.8: Using radixsort to sort w = 8-bit integers by using 4 passes of counting sort on d = 2-bit integers.

配列の要素は $\{0,\ldots,n^c-1\}$ の範囲の整数であり、$d = \lceil \log n \rceil$ だとすれば、Theorem 11.8 は次のように整理できる。

Corollary 11.1. `radixSort(a,k)` は、$\{0,\ldots,n^c-1\}$ の範囲の n 個の整数からなる配列 a を、$O(cn)$ の時間で整列する。

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 The original heap-sort algorithm is due to Williams [76] だが、本節で説明した $O(n)$ の時間でボトムアップにヒープを構築する方法は Floyd [28] によるものである。比較に基づくソートの下界は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。

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

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。(Exercise 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 [66, 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 の形式で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].

Exercise 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。

Exercise 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。(Exercise 3.13 を参照せよ。)

Exercise 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 `n` の入力の例を示せ。

Exercise 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 `n` の入力の例を示せ。

Exercise 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、`a[i]`,...,`a[i+n-1]` を先に見ないならば、長さ `n` のある入力が存在し、

$\binom{n}{2}$ 回の比較を実行することを示せ。

Exercise 11.6. `quickSort(a,i,n,c)` を実行するとき $\binom{n}{2}$ 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。）

Exercise 11.7. Theorem 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 $2nH_n - n + H_n$ であることを示せ。

Exercise 11.8. ヒープソートを実行するときの、比較の回数が $2n\log n - O(n)$ 回になる入力配列を与え、そのことを説明せよ。

Exercise 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。）

Exercise 11.10. Figure 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。

Exercise 11.11. $\log n! = n\log n - O(n)$ を示せ。

Exercise 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。

Exercise 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。

Exercise 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

グラフ

この章ではグラフのふたつの表現方法を説明し、それらを使う基本的なアルゴリズムを紹介する。

数学的には、（有向）グラフとは組み $G = (V, E)$ である。ここで V は頂点の集合であり、E は辺と呼ばれる頂点の組みの集合である。辺 (i, j) は i から j に向いている。XXX: source と target の訳語 i は辺の *source* と呼ばれ、j は *target* と呼ばれる。G における経路 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 から到達可能であるという。Figure 12.1 にグラフの例を示した。

グラフは多くの現象をモデル化できるので多くの応用を持つ。自明な例がいくつかある。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。

もうすこし巧みな例は、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すると見つかる。例えば大学の時間割りにおける衝突グラフを考えられる。ここで頂点は大学の講義で、辺 (i, j) は i と j の両方を受講する生徒がいることを表している。よってこの辺から講義 i・j のテストは同じ時間に割当てられてはならないことがわかる。

この節を通じて n は頂点の数を、m は辺の数を表すことにする。すなわち $n = |V|$ かつ $m = |E|$ である。さらに $V = \{0, \ldots, n-1\}$ と仮定する。他のデータを扱いたければ、大きさ n の配列にデータを入れて於けば良い。

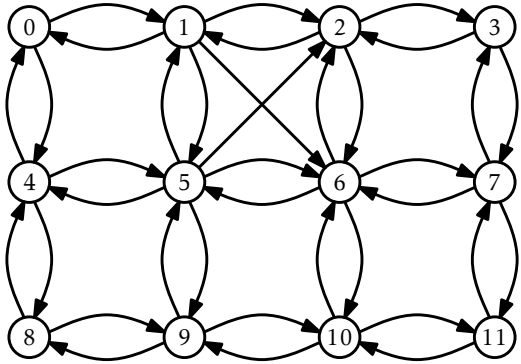


図 12.1: A graph with twelve vertices. Vertices are drawn as numbered circles and edges are drawn as pointed curves pointing from source to target.

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)`：$(i, j) \in E$ かどうかを調べる。
- `outEdges(i)`：$(i, j) \in E$ を満たす整数整数 `j` のリストを返す。
- `outEdges(i)`：$(j, i) \in E$ 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、Chapter 5 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頂点を隣接行列のリストに入れれば定数時間で実行できる。

しかし、グラフにおける応用によって、各操作への要求が異なり、理想的にはこ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、ふたつのグラフの表現方法のカテゴリについての説明をする。

`AdjacencyMatrix`：行列によるグラフの表現

隣接行列は `n` 個の頂点を持つグラフ $G = (V, E)$ を、各エントリが真偽値である `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。

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

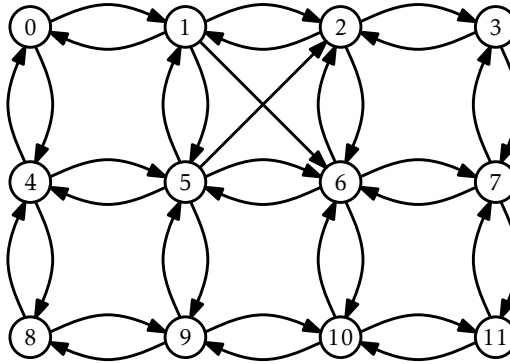
Figure 12.1 のグラフの隣接行列を Figure 12.2 に示した。

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。



	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: A graph and its adjacency matrix.

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、これ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では実際は n^2 バイトのメモリを使う。より手の込んだ実装で、w 個の真偽値をワードに詰め込めば、領域使用量は $O(n^2/w)$ ワードのメモリに減らせる。

Theorem 12.1. *AdjacencyMatrix* は *Graph* インターフェースを実装する。*AdjacencyMatrix* は次の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 が有向な場合もある。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、ここでも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、次の関係を思い出してほしい。

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

この和をグラフ G の文脈で解釈すると、これは g が辺 (i,k) と辺 (k,j) を共に持つ頂点 k の個数を数えている。つまりこれは i から j への（中間頂点 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 $O(\log n)$ 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。

AdjacencyLists：リストの集まりとしてのグラフ

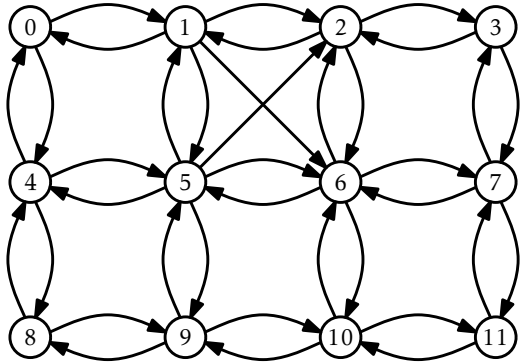
グラフの隣接リスト表現は辺を重視するアプローチである。隣接リストの実装方法は色々ありうる。この節では単純なものを説明する。そしてこの節の最後に別のやり方について述べる。隣接リスト表現ではグラフ $G=(V,E)$ はリストの配列 adj で表現される。リスト adj[i] は頂点 i と隣接するすべての頂点を含む。つまり、$(i,j)\in E$ を満たす添え字 j をすべて含むのである。

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 Figure 12.3 に示す）この実装ではリスト adj は ArrayStack としている。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。

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



0	1	2	3	4	5	6	7	8	9	10	11
1	0	1	2	0	1	5	6	4	8	9	10
4	2	3	7	5	2	2	3	9	5	6	7
	6	6		8	6	7	11		10	11	
	5				9	10					
					4						

図 12.3: A graph and its adjacency lists

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 $O(\deg(\texttt{i}))$ の時間がかかる。ここで deg(i)（i の次数）は E のうち i から出ているものの個数である。

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 $O(\deg(\texttt{i}))$ 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) はもう少しタイヘンだ。すべての頂点 j について (i,j) が存在するかどうか確認し、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;
}
```

この操作は非常に時間がかかる。すべての頂点の隣接リストを見て回る必要があるので、$O(\texttt{n}+\texttt{m})$ 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。

Theorem 12.2. *AdjacencyLists* は *Graph* インターフェースを実装する。

AdjacencyLists は次の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方法には色々ある。いくつか気になることがあるだろう。

- 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 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をを少し増やす。
- 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。
- 辺は一級オブジェクトであるべきだろうか。このとき adj は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。

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。いずれも場合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。よって、この節でアルゴリズムを分析するときにはグラフの表現が AdjacencyLists であることを仮定する。

幅優先探索

幅優先探索を頂点 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（Section 6.1.2）の一般化であって、非常に似ている。i だけをキュー q を使う。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であるとする。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、グラフの場合には同じ頂点を q に二度以上追加しないよう気をつける必要があることである。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覚えておけばよい。

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;
      }
    }
  }
}
```

Figure 12.1 において bfs(g,0) を実行する様子の一例を Figure 12.4 に示した。異なる処理の仕方もありえる。これは隣接リストの並び順によって決まる。Figure 12.4 では Figure 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 によって同じ頂点は q に二度以上追加されることはない。q に頂点の追加する（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。すべての頂点が内部ループにおいて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

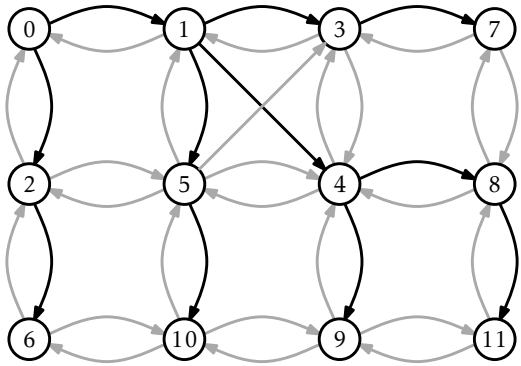


図 12.4: An example of breadth-first-search starting at node 0. Nodes are labelled with the order in which they are added to q. Edges that result in nodes being added to q are drawn in black, other edges are drawn in grey.

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。

Theorem 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。（そしてそれをいつか q から取り出す。）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、幅優先探索の特に便利な応用は最短経路の計算である。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。

深さ優先探索

深さ優先探索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。未訪問の頂点は `white`、現在訪問中の頂点は `grey`、既に訪問した頂点は `black` とする。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、その中の白い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) {
      c[j] = grey;
      dfs(g, j, c);
    }
  }
  c[i] = black; // done visiting i
}
```

Figure 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。XXX: stack overflow 上のコードは、stack の overflow 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。

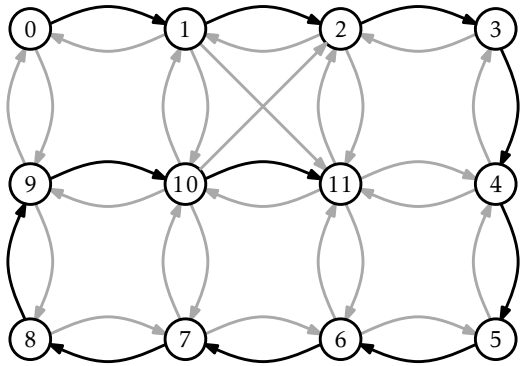


図 12.5: An example of depth-first-search starting at node 0. Nodes are labelled with the order in which they are processed. Edges that result in a recursive call are drawn in black, other edges are drawn in grey.

次の実装はこれを行ったものである。

XXX: black にしなくていいのか?

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

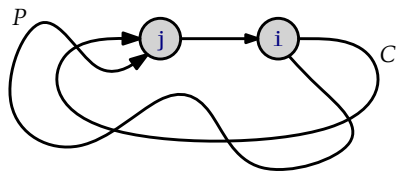


図 12.6: The depth-first-search algorithm can be used to detect cycles in G. The node `j` is coloured `grey` while `i` is still `grey`. This implies that there is a path, P, from `i` to `j` in the depth-first-search tree, and the edge (`j`,`i`) implies that P is also a cycle.

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。

Theorem 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 $O(n+m)$ 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 $i \neq r$ の色が `white` から `grey` になるのは、ある頂点 `i'` を再帰的に処理する中で `dfs(g,i,c)` を呼び出したからである。(`dfs2(g,r)` の場合は `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。Figure 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。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。Figure 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。

ディスカッションと練習問題

12.3 と 12.4 では、幅優先探索・深さ優先探索の実行時間は少し雑に述べられている。$\mathtt{n}_\mathtt{r}$ を G における頂点 `i` であって、`i` から `r` への経路が存在するものの個数と定義する。$\mathtt{m}_\mathtt{r}$ 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・深さ優先探索の実行時間に関してより正確に述べる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。）

Theorem 12.5. *`AdjacencyLists`* で実装された *`Graph`* `g` を入力すると、`bfs(g,r)`・`dfs(g,r)`・`dfs2(g,r)` の実行時間はいずれも $O(\mathtt{n}_\mathtt{r} + \mathtt{m}_\mathtt{r})$ 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 and 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。

Exercise 12.1. Figure 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。

Exercise 12.2. グラフ G の接続行列とは、$\mathtt{n} \times \mathtt{m}$ 行列 A であって、次のように定義されるものである。

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

1. Figure 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。

Exercise 12.3. Figure 12.7 のグラフ G について、`bfs(G,0)`・`dfs(G,0)` を実

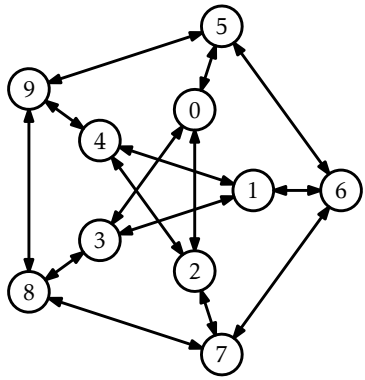


図 12.7: An example graph.

行する様子を図示せよ。

Exercise 12.4. G を無向グラフとする。G が連結であるとは、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 $O(n+m)$ の時間で確認する方法を示せ。

Exercise 12.5. G を無向グラフとする。G の連結成分ラベル付けとは、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 $O(n+m)$ の時間で計算する方法を示せ。

Exercise 12.6. G を無向グラフとする。G の全域森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 $O(n+m)$ の時間で計算する方法を示せ。

Exercise 12.7. グラフ G が強連結であるとは、G の任意の頂点の組み i, j について、i から j への経路が存在することをいう。これを $O(n+m)$ の時間で確認する方法を示せ。

Exercise 12.8. グラフ $G=(V,E)$ と、特別な頂点 $r \in V$ があるとき、r から全頂点 $i \in V$ への最短経路の長さを計算する方法を示せ。

Exercise 12.9. dfs(g, r) が dfs2(g, r) と異なる順番で頂点を訪問する単純な例を与えよ。また、dfs(g, r) と常に同じ順番で頂点を訪問する dfs2(g, r) を実装せよ。（ヒント：r から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。）

Exercise 12.10. XXX: universal sink の訳語グラフ G の *universal sink* とは、$n-1$ 個の辺の行き先になっており、かつそこから辺が出ていない頂点である。[*1] `AdjacencyMatrix` で表現されるグラフ G が universal sink を持つかどうかを判定するアルゴリズムを設計・実装せよ。ただし、実行時間は $O(n)$ でなければならない。

[*1] universal sink v を *celebrity* と言うこともある。部屋の中のみなが v のことを知っているが、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 SSet の実装を再び扱う。ただしここでは SSet の要素は w ビットの整数だと仮定する。すなわち $x \in \{0,\ldots,2^w-1\}$ について add(x)・remove(x)・find(x) を実装する。データやキーが整数である応用は明らかにたくさんあるだろう。

以上のことをふまえた 3 つのデータ構造についてこの章では説明する。一つ目は BinaryTrie であり、これは SSet の 3 つの操作をいずれも $O(w)$ の時間で実行する。これにはさほど驚かないかもしれない。$\{0,\ldots,2^w-1\}$ の部分集合の大きさは $n \le 2^w$ であり、$\log n \le w$ が成り立つためだ。この本で説明した SSet の実装は各操作の実行時間が $O(\log n)$ であった。すなわち、いずれも BinaryTrie と同じくらいは高速であった。

二つ目は XFastTrie であり、これは BinaryTrie の検索をハッシュ法を利用して高速化するものである。この高速化により、find(x) の実行時間は $O(\log w)$ 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 $O(w)$ であり、領域使用量は $O(n \cdot w)$ である。

三つ目は YFastTrie であり、これは XFastTrie に約 w 個にひとつのサンプルを格納し、それ以外をふつうの 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 $O(\log w)$ に、領域使用量は $O(n)$ に抑えられる。

この章における実装例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ける ix は x に対応する整数を表し、intValue(x) は x を ix に変換するメソッドであるとする。しかし、文章中においては単に x が整数であるかのように扱う。

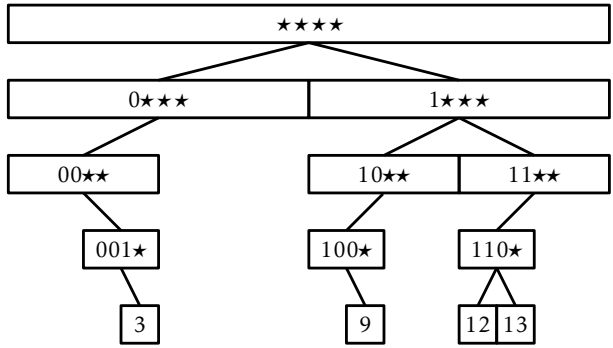


図 13.1: The integers stored in a binary trie are encoded as root-to-leaf paths.

BinaryTrie：デジタル探索木

BinaryTrie は w ビットの整数の集まりを二分木で符号化したものである。この木の任意の葉の深さは w であって、各整数は根から葉への経路として符号化される。整数 x への経路は深さ i において、もし上から i 番目のビットが 0 なら左、1 なら右に向かう。Figure 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) がトライに格納されている。

x の探索経路は x の二進表現によって決まるので、ノード u の子を u.child[0] (left)・u.child[1] (right) と呼ぶことにすると便利である。この子を指すポインタは二重の意味で使われる。二分トライの葉は子を持たないので、ここではポインタを使って葉の双方向連結リストを作る。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、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を表現するために使われる。(Section 3.2 を参照せよ。)

ノード u は u.jump というポインタも持つ。u が左の子を持たないとき、u.jump は u の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 の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を、Figure 13.2 に示す。

BinaryTrie における find(x) 操作は簡単である。x の探索経路を辿ればよい。葉にたどり着けば、x が見つかる。(進みたい方向の子を持っていない

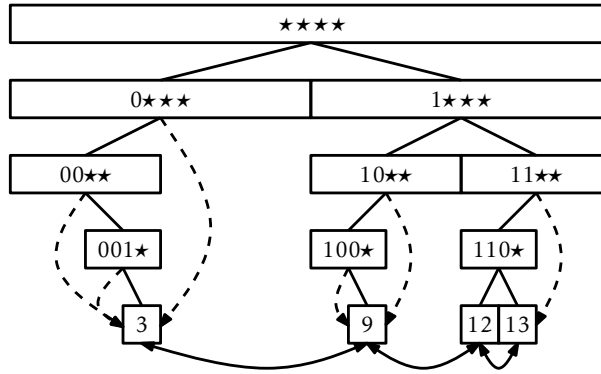


図 13.2: A BinaryTrie with jump pointers shown as curved dashed edges.

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。どちらになるかは u が左右どちらの子を持た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。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。Figure 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;
}
```

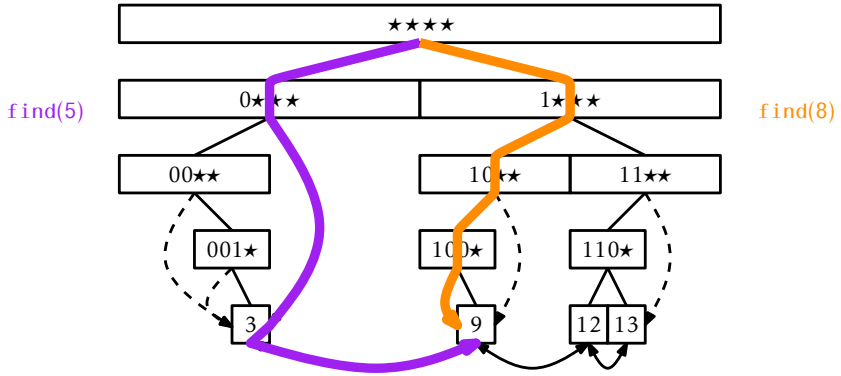


図 13.3: The paths followed by `find(5)` and `find(8)`.

`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、やらなければならない処理はたくさんある。

1. `x` の探索経路を辿り、それ以上進めないノード `u` を得る。
2. `u` から `x` を含む葉への、探索経路の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。（最初のステップで得た `u` の `jump` ポインタを利用して、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逆に辿り、`x` を指す必要のある `jump` ポインタを調整する。

Figure 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;
```

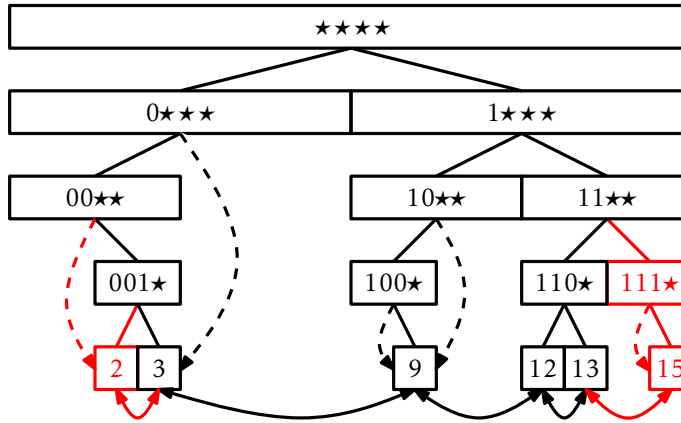


図 13.4: Adding the values 2 and 15 to the BinaryTrie in Figure 13.2.

```
    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 x の探索経路を辿り、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 $O(w)$ である。

remove(x) は add(x) のすることを取り消す。add(x) と同様にやらなければならないことがたくさんある。

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。

Figure 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;
```

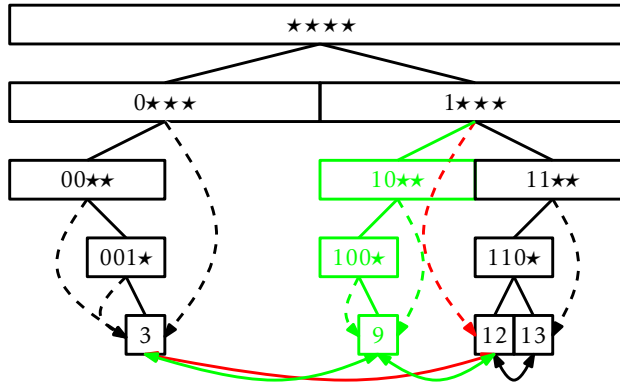


図 13.5: Removing the value 9 from the BinaryTrie in Figure 13.2.

```
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;
  }
```

Theorem 13.1. *BinaryTrie* は w ビット整数のための *SSet* インターフェースの実装である。*BinaryTrie* は add(x)・remove(x)・find(x) をいずれも $O(w)$ の時間で実行できる。n 個の要素を格納する *BinaryTrie* の領域使用量は $O(n \cdot w)$ である。

XXX: doubly-logarithmic は log(log(x)) のことだろうか（なんと訳そう）

XFastTrie：Doubly-Logarithmic 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^w であり、$\log n \le w$ が成り立つ。つまりこの本で説明して比較に基づく SSet の実装はいずれも、少なくとも BinaryTrie と同じ程度効率的であり、またそれらには整数しか格納できないという制限はなかった。

次は XFastTrie を説明する。これは単に BinaryTrie に加えて、トライの各深さにひとつずつ、$w+1$ 個のハッシュテーブルを置いたものである。これを使って、find(x) の性能を $O(\log w)$ に上げられる。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。トライのある深さにおける二分探索でノード u を見つけることで、XFastTrie はこの探索処理を高速に行う。

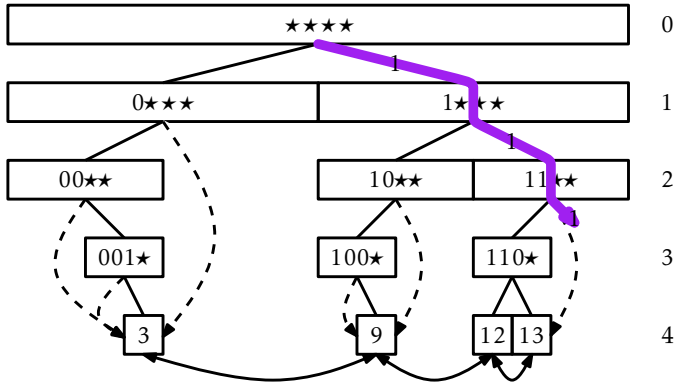


図 13.6: Since there is no node labelled 111★, the search path for 14 (1110) ends at the node labelled 11★★ .

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、Figure 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、$i \in \{0,\ldots,w\}$ について深さ i のすべてのノードを USet t[i] に格納する。USet はハッシュテーブル（Chapter 5）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 ≤ 1 になればこの処理を終了する。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) の処理を完了できる。

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 $O(\log w)$ 回繰り返すと u が見つかる。各繰り返しは決まった量だけの仕事をし、一回だけ USet の find(x) を呼ぶ。USet の検索処理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 $O(\log w)$ 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。修正が必要なのはハッシュテーブル t[0],...,t[w] を管理する必要があることだけである。add(x) の実行中に深さ i でノードが

作られるなら、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）コードとほぼおなじなので、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。

Theorem 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 $O(w)$ である。
- find(x) の実行時間の期待値は $O(\log w)$ である。

n 個の要素を格納する *XFastTrie* の領域使用量は $O(n \cdot w)$ である。

YFastTrie：実行時間が Doubly-Logarithmic な SSet

XFastTrie は BinaryTrie と比べて問い合わせの応答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さほど速くない。さらに、operations are still not terribly fast. Furthermore, the space usage, 領域使用量は $O(n \cdot w)$ であり、この本で紹介した他の SSet の実装の $O(n)$ と比べて大きい。この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 $n \cdot w$ の構造を作るなら、add(x) の一回あたりの実行時間・領域使用量は少なくとも 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共に改善する。YFastTrie は XFastTrie xft を使うが、xft には $O(n/w)$ 個の値しか格納しない。こうすると xft の領域使用量は $O(n)$ になる。さらに、w 回に一回だけの add(x)・remove(x) が xft に add(x)・remove(x) を実行する。こうして、xft における add(x)・remove(x) の平均呼び出しコストは定数になる。

きっと疑問を感じるだろう。xft には n/w 個だけの要素を格納するなら、残りの $n(1-1/w)$ 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(Section 7.2) を拡張したデータ構造に格納する。二次構造は約 n/w 個あり、平均的にはそれぞれ $O(w)$ 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、それぞの操作の実行時間は期待通り

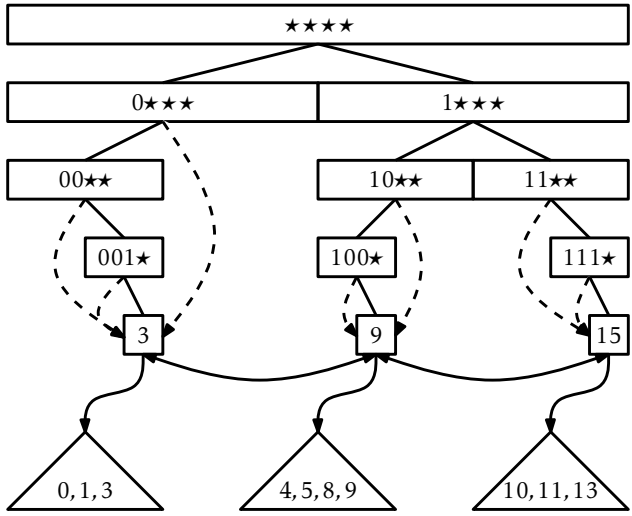


図 13.7: A YFastTrie containing the values 0, 1, 3, 4, 6, 8, 9, 10, 11, and 13.

$O(\log w)$ である。

具体的には、YFastTrie は、独立に確率 1/w でランダムに選り抜いたデータを格納する XFastTrie xft を含む。都合上、xft は常に値 $2^w - 1$ を含むものとする。また、xft が含む要素を $x_0 < x_1 < \cdots < x_{k-1}$ とする。要素 x_i には対応する Treap t_i があり、これは $x_{i-1}+1, \ldots, x_i$ の範囲の値をすべて格納する。Figure 13.7 にこの様子を示す。

YFastTrie における find(x) は簡単である。x を xft から検索し、Treap t_i に対応するある値 x_i を得る。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。[*1]

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。Figure 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。Exercise 7.12 では、t を t1 と t′ とに分割するのにかかる時間の期待値も $O(\log w)$ であることを示す。組み (x,t1) を xft に追加するのは $O(w)$ の時間がかかるが、これは確率

[*1] これは *Jensen* の不等式の応用すればよい。$\mathrm{E}[r] = w$ ならば $\mathrm{E}[\log r] \le \log w$ である。

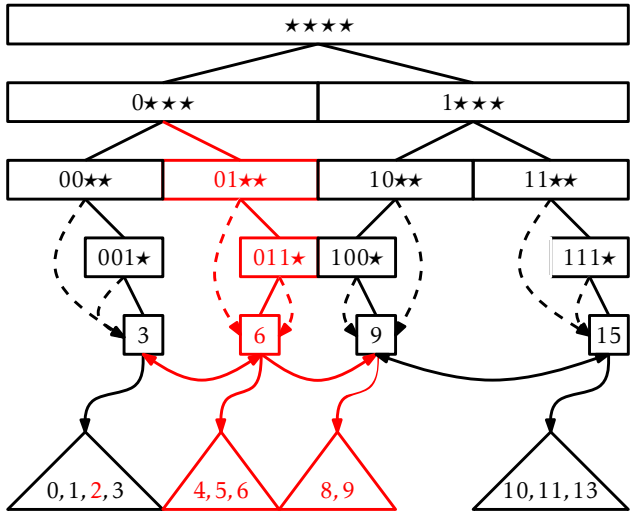


図 13.8: Adding the values 2 and 6 to a YFastTrie. The coin toss for 6 came up heads, so 6 was added to xft and the treap containing 4, 5, 6, 8, 9 was split.

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 を使って、xft.find(x) の結果を教えてくれる葉 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 にも含まれれいれば（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 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。Figure 13.9 にこの様子を示す。

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) {
```

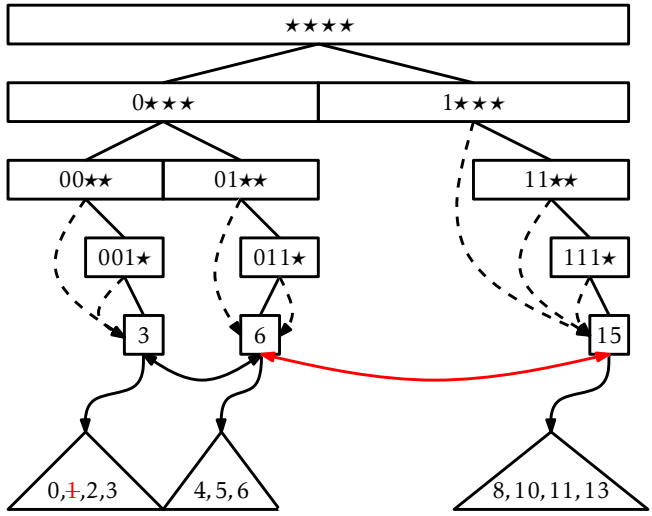


図 13.9: Removing the values 1 and 9 from a YFastTrie in Figure 13.8.

```
    STreap<T> t2 = u.child[1].x.t;
    t2.absorb(u.x.t);
    xft.remove(u.x);
  }
  return ret;
}
```

xft からノード u を見つけるのに必要な時間の期待値は $O(\log w)$ である。t から x を削除するのにかかる時間の期待値も $O(\log w)$ である。繰り返しになるが、Exercise 7.12 では、t を t1 と t′ とに分割するのにかかる時間の期待値も $O(\log w)$ であることを示す。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 から要素を削除するときにかかる時間の期待値は $O(\log w)$ である。

議論の前半で、このデータ構造における各 Treap の大きさについて説明するの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。

Lemma 13.1. x を *YFastTrie* に格納する整数とし、n_x を x を含む *Treap* t

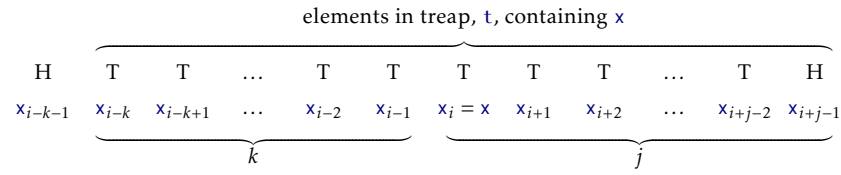


図 13.10: The number of elements in the treap t containing x is determined by two coin tossing experiments.

の要素数とする。このとき $\mathrm{E}[n_x] \le 2w-1$ が成り立つ。

Proof. Figure 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 は x 以上の要素を含む。これらを $x_i, x_{i+1}, \ldots, x_{i+j-1}$ をとすると、x_{i+j-1} はこのうち add(x) のときの偏りのあるコイン投げで表が出た唯一の要素である。つまり、$\mathrm{E}[j]$ は偏りのあるコイン投げで、はじめて表が出るまで繰り返すときの試行回数の期待値に等しい。[*2] コイン投げは独立な試行であり、確率 1/w で表が出る。そのため $\mathrm{E}[j] \le w$ である。($w = 2$ の場合の解析として Lemma 4.2 を参照せよ。)

同様に、t の x よりも小さい要素 $x_{i-1}, \ldots, x_{i-k}$ 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、x_{i-k-1} のコイン投げは表である。これは、先の段落と同じコイン投げ試行において、最後の試行を数えない場合なので、$\mathrm{E}[k] \le w-1$ である。

まとめると、$n_x = j+k$ より、

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

Lemma 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。

Theorem 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 $O(\log w)$ である。n 要素を格納する *YFastTrie* の領域使用量は $O(n+w)$ 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^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

[*2] この解析は j が $n-i+1$ 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。

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 $O(n)$ になる。

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* (or *stratified*) 木という名で知られている。[72] オリジナルの van Emde Boas 木の大きさは 2^w 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 [75] 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 tree 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。

他の整数を格納するため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 [57] 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すくなくともほぼ最適である。

Exercise 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。

Exercise 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とに格納する。

Exercise 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。この発想を可変章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。

Exercise 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 $O(1 + \log d(x))$ であるものを設計・実装せよ。ヒント：ハッシュテーブル $t[w]$ は $d(x) = 0$ 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 $O(1 + \log\log d(x))$ 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 Section 1.4 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の集まりが存在するのである。このような場合、ハードディスク・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。HHD や SSD のバイトにアクセスするときには、実際はブロック単位での読み出しが行われる。XXX: 著者の環境の話? 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 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のがよいだろう。

これが外部メモリモデルの背後にあるアイデアである。Figure 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

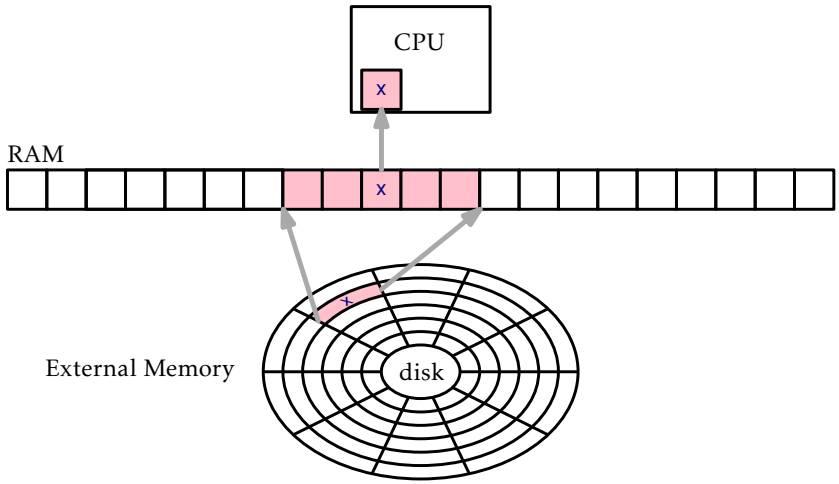


図 14.1: In the external memory model, accessing an individual item, x, in the external memory requires reading the entire block containing x into RAM.

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 $O(B + \log_B n)$ で十分である。これは定数個だけのブロックと、高さ $O(\log_B n)$ 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 $n \leq 2^{160}$ について $B \geq \log_B n$ が成り立つ。十進法では、$B \geq \log_B n$ が以下の範囲で成り立つ。

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

Block Store

様々なデバイスが外部メモリの概念には含まれる。それぞれブロックサイズが定義されており、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 はメモリ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 $iB,\ldots,(i+1)B-1$ 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。

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 Section 9.1 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 $B=2$ とした特殊可すると 2-4 木になる。)

For any integer 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。

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

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 木の高さは葉の数の B を底とする対数に比例する。

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には `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。

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

`u` が内部ノードなら、任意の `i` $\in \{0,\ldots,k-2\}$ について `u.keys`$[$`i`$]$ は `u.children`$[$`i`$]$ を根とする部分木に格納されるどのキーよりも大きく、`u.children`$[$`i`$+1]$ を根とする部分木に格納されるどのキーよりも小さい。つまり、厳密な書き方ではないが、次が成り立つ。

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

$B=2$ である B の例を Figure 14.2 に示す。

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サイズ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、ノードの添字も 4 バイトであるとる。このとき $B=256$ とすれば各ノードは

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

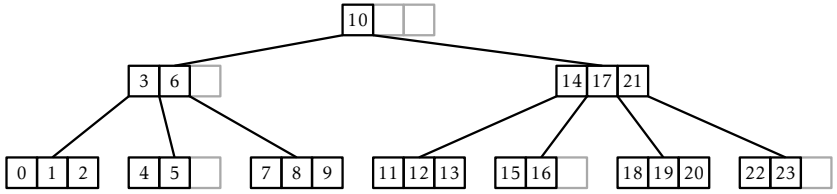


図 14.2: A B-tree with $B = 2$.

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。

XXX: インデックス?添え字? BTree クラスは B 木の実装である。BTree のノードを格納する BlockStore 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;
```

要素の検索

Figure 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 に格納されているかどうかを確認する。もしそうなら、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、最後に見たもの 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。

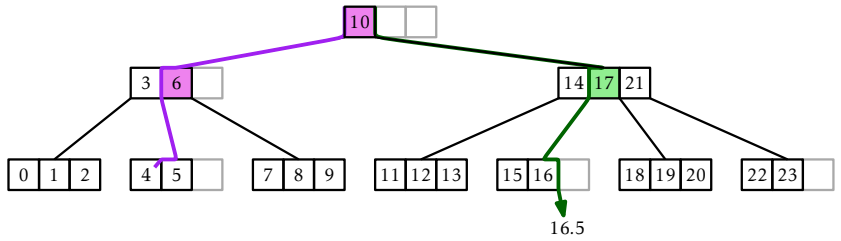


図 14.3: A successful search (for the value 4) and an unsuccessful search (for the value 16.5) in a *B*-tree. Shaded nodes show where the value of z is updated during the searches.

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` を探すメソッドである。Figure 14.4 に示したように、$a[0],\ldots,a[k-1]$ にはキーが整列された状態で、$a[k],\ldots,a[a.length-1]$ 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 $a[i] = null$ を満たす最小の添え字 `i` を返す。

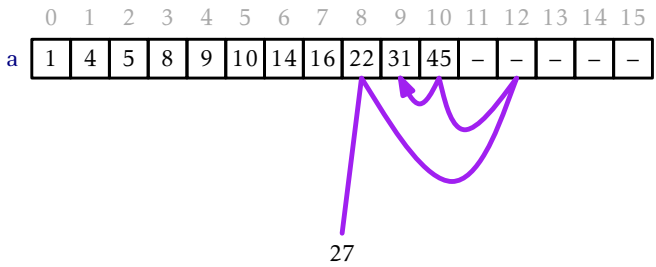


図 14.4: The execution of findIt(a,27).

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、$O(\log(\texttt{a.length}))$ 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、n 個のキーを格納する B 木の高さは $O(\log_B \texttt{n})$ である。よって、外部メモリモデルでは、find(x) の実行時間

は $O(\log_B n)$ 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。

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

要素の追加

B 木と Section 6.2 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)` は再帰を使って簡単に実装できるのだ。

他のバランスされた探索木と同様に、`add(x)` の際にある種のバランスの調整が必要になる。B 木ではこれはノードの分割によって行われる。Figure 14.5 を続く内容の参考にしてほしい。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` の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、すべての親へのポインタを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。

Figure 14.6 に B 木における `add(x)` の様子を示す。XXX: at a high level? なにが? ざっくり言うと、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` が一杯になったら（つまり既に $B-1$ 個のキーを持っていれば)、`u` を分割する。この結果 `u` の親が一杯になるかもしれず、その場合には `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るかもしれず... 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。

`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

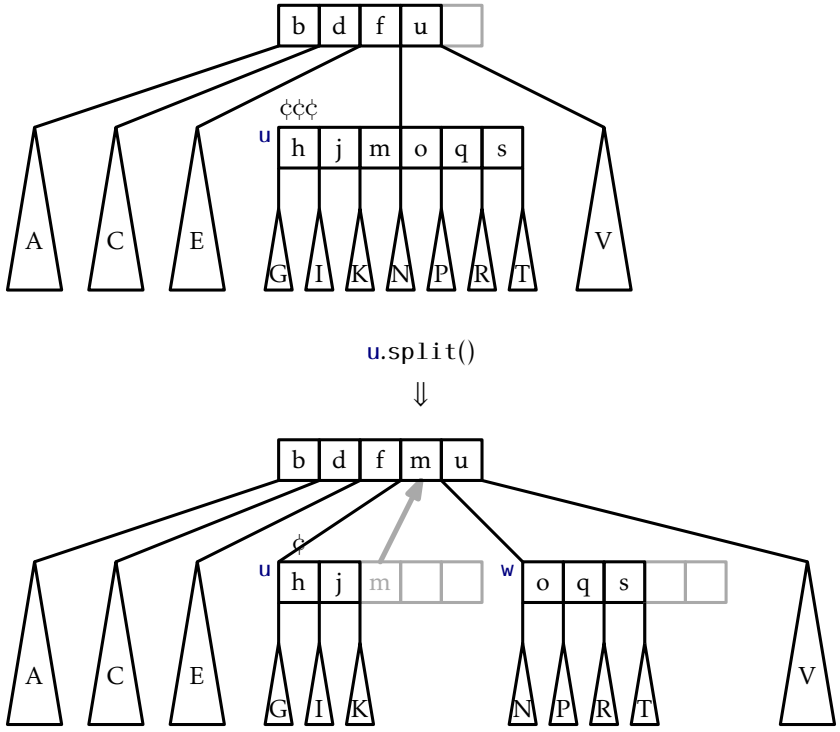


図 14.5: Splitting the node u in a B-tree ($B = 3$). Notice that the key u.keys[2] = m passes from u to its parent.

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。この高レベルな様子を頭に入れて、次はこ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) の処理は実際には addRecursive(x,ui) が行う。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 は 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。この場合は u は w を取り込み、w の最初の子を奪って、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

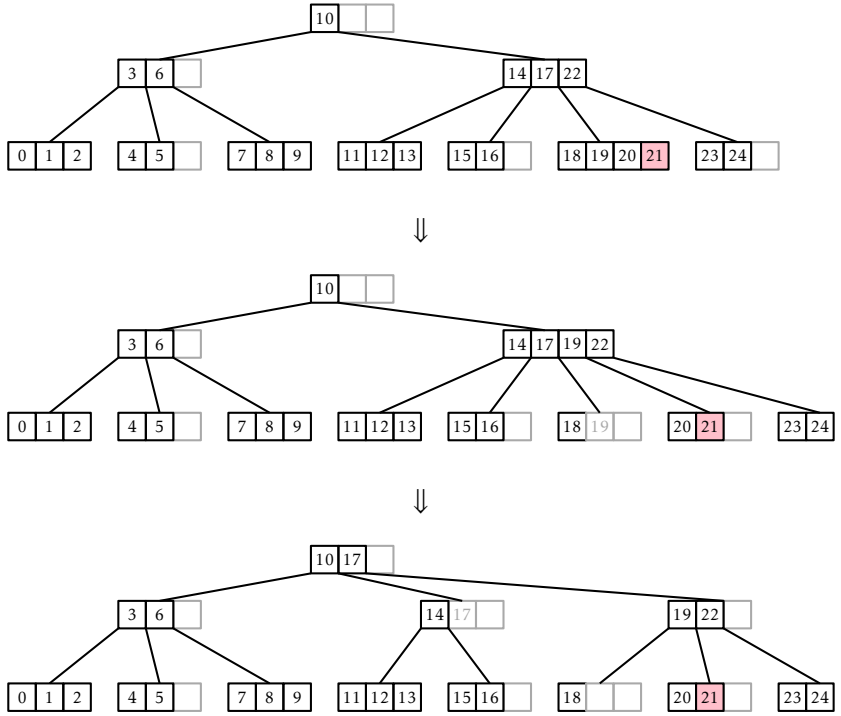


図 14.6: The add(x) operation in a BTree. Adding the value 21 results in two nodes being split.

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) を読んで x を B 木の根に挿入する。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 {
    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 $O(\log_B n)$、ワード RAM モデルでは $O(\log n)$ である。

上向きに進む段階　再帰の上向きに進む段階では、x を追加した後、合わせて最大 $O(\log_B n)$ 回の分割を行う。各分割は 3 つのノードだけに影響するので、この段階の実行時間は外部メモリモデルでは $O(\log_B n)$ である。しかし、各分割は B 個のキーと子をノードからノードに移すので、ワード RAAM モデルでは、この実行時間は $O(B \log n)$ である。

B の値はかなり大きく、$\log n$ 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。Section 14.2.4 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。

ノードの削除

繰り返しになるが、BTree における remove(x) は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、Figure 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と呼ぶことにする。

アンダーフローが発生すると、兄弟から u は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

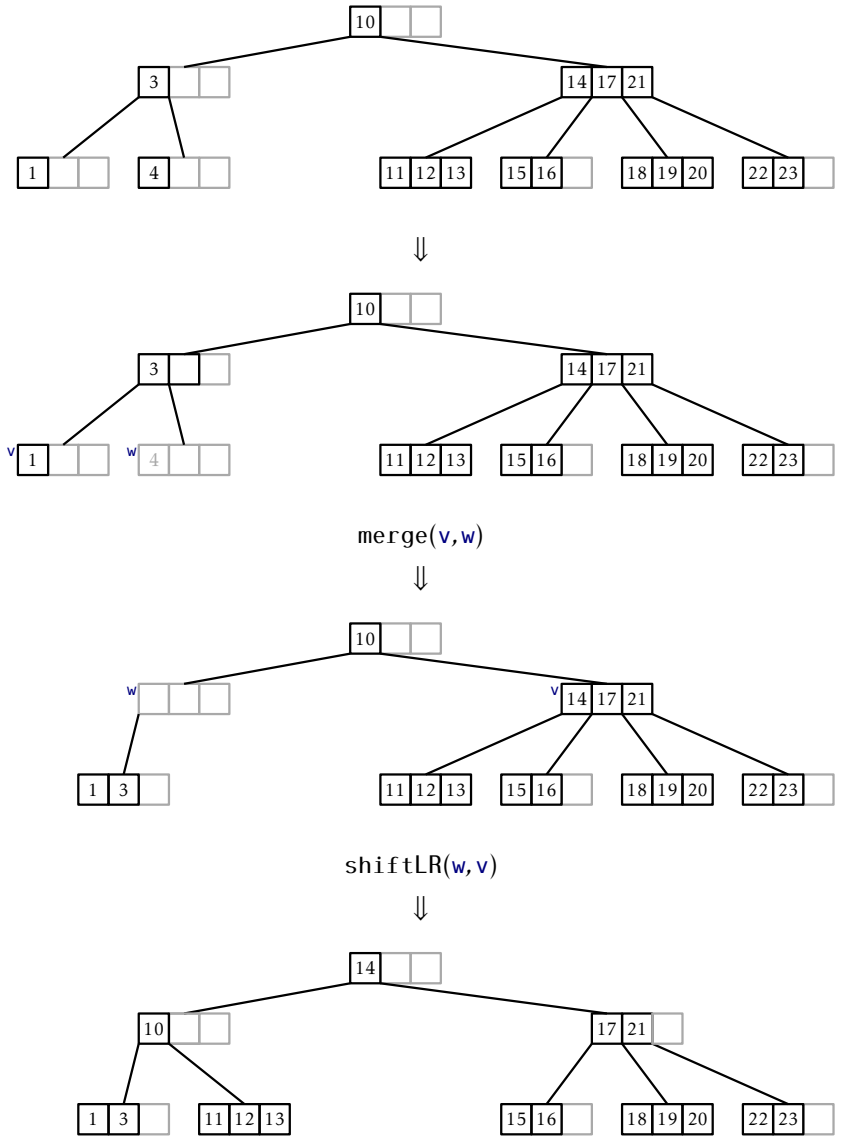


図 14.7: Removing the value 4 from a *B*-tree results in one merge and one borrowing operation.

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。Figure 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()) {
      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;
}
```

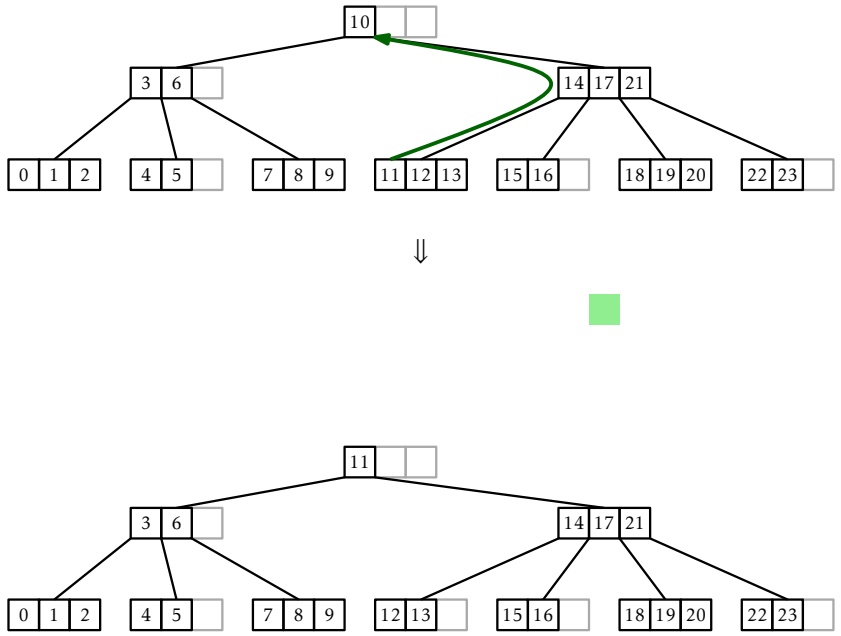


図 14.8: The remove(x) operation in a BTree. To remove the value x = 10 we replace it with the the value x′ = 11 and remove 11 from the leaf that contains it.

```
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 $B-1$ 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 $i-1$ 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。このための方法はふたつある。

借りてくる　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。

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

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。この処理の様子を Figure 14.9 に示す。

併合する　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、Figure 14.10 に示したように v と w とを併合する。併合は分割の

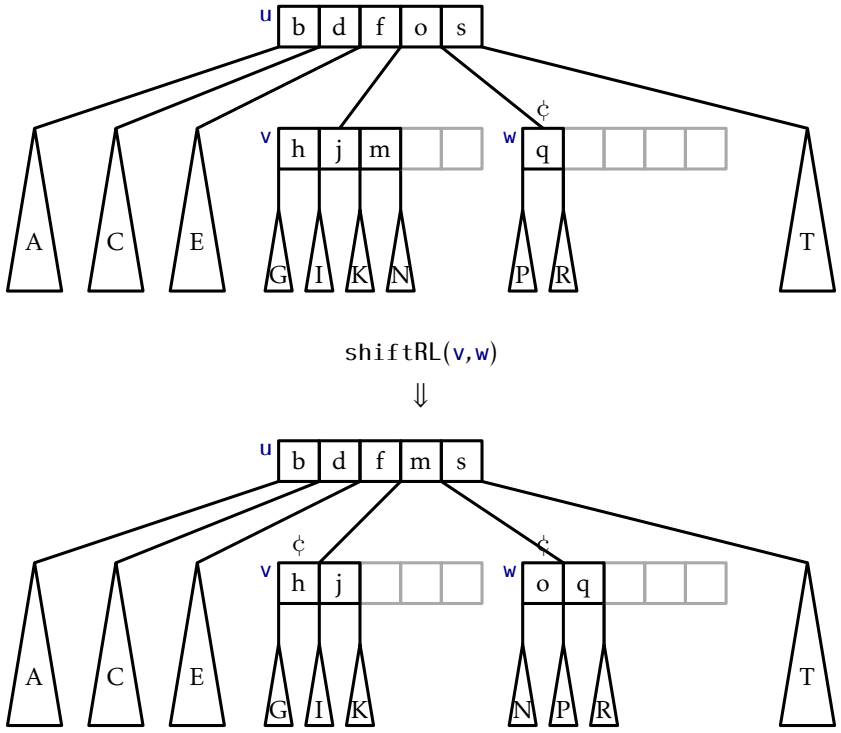


図 14.9: If v has more than $B-1$ keys, then w can borrow keys from v.

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。（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 がキーをひとつ持てなくなり、これがひとつキーが増える原因である。）

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
```

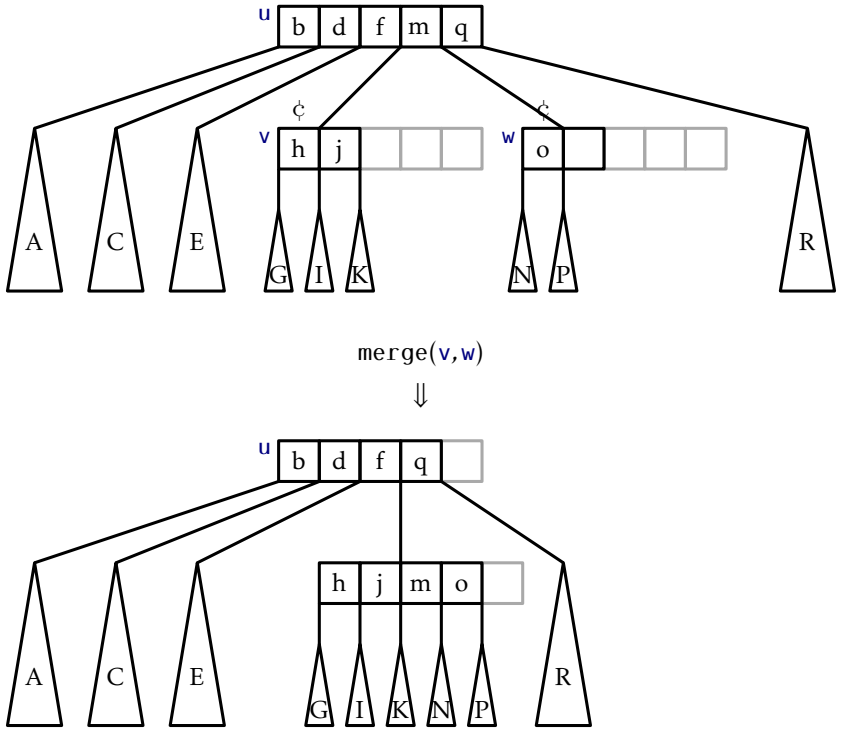


図 14.10: Merging two siblings v and w in a B-tree ($B = 3$).

```
      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、高々一回ノードを借りてくる。併合やノードを借りるときには 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。

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。

Lemma 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。

Proof. $B = 2$ の特別な場合について、Section 9.3 で既に証明の概要を示している。XXX: credit scheme この補題はここではクレジット法で証明する。

1. 分割・併合・借用の際にクレジット 2 を支払う。

2. add(x) または remove(x) の際には、最大 3 のクレジットが得られる。

最大で $3m$ のクレジットが得られており、各分割・併合・借用はクレジットを 2 消費するので、最大 $3m/2$ 回の分割・併合・借用が実行される。 14.5、14.9、14.10 ではクレジットは ¢ で表した。

クレジットの値を追うために、証明では次のクレジット不変条件を保つ。$B-1$ 個のキーを持つ任意の根でないノードはクレジットを 1 だけ持つ。$2B-1$ 個のキーを持つノードはクレジットを 3 だけ持つ。B 以上 $2B-2$ 以下のキーを持つノードはクレジットを持たない。あとは、各 add(x)・remove(x) の間に、クレジット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クレジットは 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、クレジット不変条件が保たれる。よって、分割の際に、そのためのクレジットを支払え、クレジット不変条件を保つことができる。

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、クレジットを 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クレジットの範囲内である。

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、この併合によってクレジットを 2 得られ、これを併合のコストとして使える。

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・借用は発生しない。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 はクレジットを 1 持っており、このクレジット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、クレ

ジットが 1 足らず、支払いを完了するためにクレジットをあと 1 必要である。

ここまでで、クレジットを 1 得ており、クレジット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 がクレジットを 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クレジットを 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、最大で 3 のクレジットを作る必要があることがわかった。

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、クレジットを 1 作って与えなければならない。

XXX: ちょっと迷子いずれの場合にも、つまり削除が借用で終わっても、そうでなくても、クレジット不変条件を保ち、併合と借用のコストを支払うためには、高々 3 のクレジットを remove(x) の間に作る必要がある。以上より定理が示された。 □

Lemma 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 $O(B+\log n)$ である。この結果を次のふたつの定理にまとめる。This is summarized by the following pair of theorems:

Theorem 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。外部メモリモデルでは、*BTree* は add(x)・remove(x)・find(x) をサポートし、いずれの実行時間も $O(\log_B n)$ である。

Theorem 14.2 (ワード RAM モデルにおける B 木)**.** *BTree* は *SSet* インターフェースを実装する。*BTree* は add(x)・remove(x)・find(x) をサポートする。ワード *RAM* モデルでは、分割・併合・借用のコストを無視すると、いずれの実行時間も $O(\log_B n)$ である。さらに、空の *BTree* に対して、add(x)・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。

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。このデータ構造はいたるところで使われているという Comer のサーベイ（論文のタイトルが"The Ubiquitous B-Tree"）を出版されるまで、10 年もかからなかった。[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、$O(\log_B n)$ 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。この理由は、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。

Exercise 14.1. Figure 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。

Exercise 14.2. Figure 14.2 の B 木から 3、4 を順に削除するときの様子を描け。

Exercise 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。）

Exercise 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 $O(B+\log_B n)$ 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 $B\log_B n$ に比例する内部メモリを使うことを示せ。
2. これを $O(B+\log_B n)$ に減らすための修正方法を説明せよ。

Exercise 14.5. Lemma 14.1 の証明で使ったクレジットの様子を、図 14.6 と図 14.7 の木に描け。また、（追加のクレジット 3 で）分割・併合・借用のコストを支払い、クレジット不変条件を保てることを確認せよ。

Exercise 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。）

Exercise 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。

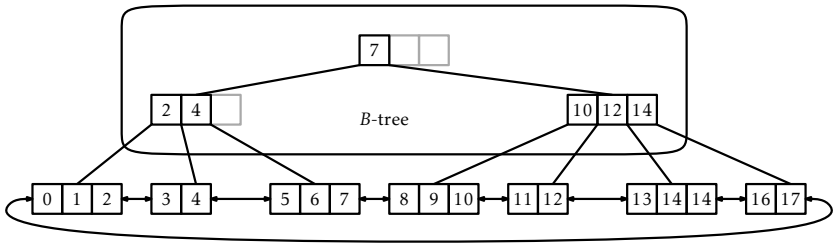


図 14.11: A B^+-tree is a B-tree on top of a doubly-linked list of blocks.

(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。

Exercise 14.8. Figure 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、この重複が対して問題とならない理由を説明せよ。

参考文献

[1] Free eBooks by Project Gutenberg. URL: http://www.gutenberg.org/ [cited 2011-10-12].

[2] IEEE Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, August 2008. doi:10.1109/IEEESTD.2008.4610935.

[3] G. Adelson-Velskii and E. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3(1259-1262):4, 1962.

[4] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[5] A. Andersson. Improving partial rebuilding by using simple balance criteria. In F. K. H. A. Dehne, J.-R. Sack, and N. Santoro, editors, *Algorithms and Data Structures, Workshop WADS '89, Ottawa, Canada, August 17–19, 1989, Proceedings*, volume 382 of *Lecture Notes in Computer Science*, pages 393–402. Springer, 1989.

[6] A. Andersson. Balanced search trees made simple. In F. K. H. A. Dehne, J.-R. Sack, N. Santoro, and S. Whitesides, editors, *Algorithms and Data Structures, Third Workshop, WADS '93, Montréal, Canada, August 11–13, 1993, Proceedings*, volume 709 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 1993.

[7] A. Andersson. General balanced trees. *Journal of Algorithms*, 30(1):1–18, 1999.

[8] A. Bagchi, A. L. Buchsbaum, and M. T. Goodrich. Biased skip lists. In P. Bose and P. Morin, editors, *Algorithms and Computation, 13th International Symposium, ISAAC 2002 Vancouver, BC, Canada, November*

21–23, 2002, Proceedings, volume 2518 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2002.

[9] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. In *SIGFIDET Workshop*, pages 107–141. ACM, 1970.

[10] Bibliography on hashing. URL: http://liinwww.ira.uka.de/bibliography/Theory/hash.html [cited 2011-07-20].

[11] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In M. J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15–19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 79–79. Springer, 1999.

[12] P. Bose, K. Douïeb, and S. Langerman. Dynamic optimality for skip lists and b-trees. In S.-H. Teng, editor, *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20–22, 2008*, pages 1106–1114. SIAM, 2008.

[13] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick. Resizable arrays in optimal time and space. In Dehne et al. [18], pages 37–48.

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

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

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

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

[18] F. K. H. A. Dehne, A. Gupta, J.-R. Sack, and R. Tamassia, editors. *Algorithms and Data Structures, 6th International Workshop, WADS '99, Vancouver, British Columbia, Canada, August 11–14, 1999, Proceedings*, volume 1663 of *Lecture Notes in Computer Science*. Springer, 1999.

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

[20] P. Dietz and J. Zhang. Lower bounds for monotonic list labeling. In J. R. Gilbert and R. G. Karlsson, editors, *SWAT 90, 2nd Scandinavian Workshop on Algorithm Theory, Bergen, Norway, July 11–14, 1990, Proceedings*, volume 447 of *Lecture Notes in Computer Science*, pages 173–180. Springer, 1990.

[21] M. Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In C. Puech and R. Reischuk, editors, *STACS 96, 13th Annual Symposium on Theoretical Aspects of Computer Science, Grenoble, France, February 22–24, 1996, Proceedings*, volume 1046 of *Lecture Notes in Computer Science*, pages 567–580. Springer, 1996.

[22] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In W. Kuich, editor, *Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria, July 13–17, 1992, Proceedings*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246. Springer, 1992.

[23] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997.

[24] M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. M. auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

[25] A. Elmasry. Pairing heaps with $O(\log \log n)$ decrease cost. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 471–476. Society for Industrial and Applied Mathematics, 2009.

[26] F. Ergun, S. C. Sahinalp, J. Sharp, and R. Sinha. Biased dictionaries with fast insert/deletes. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 483–491, New York, NY, USA, 2001. ACM.

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

[28] R. W. Floyd. Algorithm 245: Treesort 3. *Communications of the ACM*,

7(12):701, 1964.

[29] M. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

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

[31] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with 0 (1) worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[32] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of computer and system sciences*, 47(3):424–436, 1993.

[33] I. Galperin and R. Rivest. Scapegoat trees. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 165–174. Society for Industrial and Applied Mathematics, 1993.

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

[35] M. T. Goodrich and J. G. Kloss. Tiered vectors: Efficient dynamic arrays for rank-based sequences. In Dehne et al. [18], pages 205–216.

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

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

[38] L. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, 16–18 October 1978, Proceedings*, pages 8–21. IEEE Computer Society, 1978.

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

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

[41] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *Journal of*

the ACM, 21(4):549–568, 1974.

[42] HP-UX process management white paper, version 1.3, 1997. URL: http://h21007.www2.hp.com/portal/download/files/prot/files/STK/pdfs/proc_mgt.pdf [cited 2011-07-20].

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

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

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

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

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

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

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

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

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

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

[53] J. I. Munro, T. Papadakis, and R. Sedgewick. Deterministic skip lists. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms (SODA'92)*, pages 367–375, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

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

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

[56] T. Papadakis, J. I. Munro, and P. V. Poblete. Average search and up-

date costs in skip lists. *BIT*, 32:316–332, 1992.

[57] M. Pătraşcu and M. Thorup. Randomization does not help searching predecessors. In N. Bansal, K. Pruhs, and C. Stein, editors, *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7–9, 2007*, pages 555–564. SIAM, 2007.

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

[59] W. Pugh. A skip list cookbook. Technical report, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, 1989. URL: ftp://ftp.cs.umd.edu/pub/skipLists/cookbook.pdf [cited 2011-07-20].

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

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

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

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

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

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

[66] H. H. Seward. Information sorting in the application of electronic digital computers to business operations. Master's thesis, Massachusetts Institute of Technology, Digital Computer Laboratory, 1954.

[67] Z. Shao, J. H. Reppy, and A. W. Appel. Unrolling lists. In *Proceedings of the 1994 ACM conference LISP and Functional Programming (LFP'94)*, pages 185–195, New York, 1994. ACM.

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

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

[70] D. Sleator and R. Tarjan. Self-adjusting binary trees. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25–27 April, 1983, Boston, Massachusetts, USA*, pages 235–245. ACM, ACM, 1983.

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

[72] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977.

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

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

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

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

索引