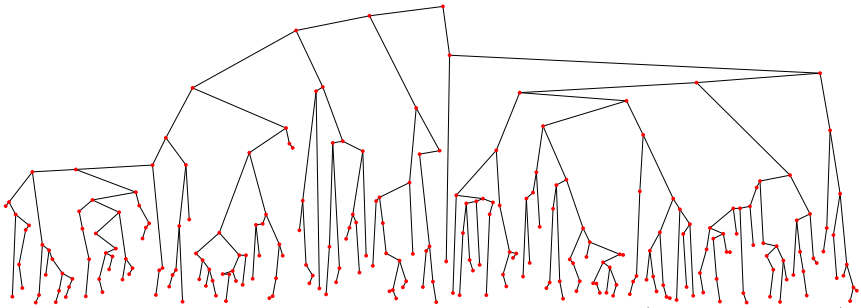


Open Data Structures (in C++)

Edition 0.1Gβ

Pat Morin

目次

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、C++ 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。

なぜこんなことが起こりうるのだろうか？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、遅れが生死を別つ可能性があるためである。

1.1 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、配列または線形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。少なくとも一度は各アイテムを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）回はこのデータで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 $10^6 \times 10^6 = 10^{12}$（1000 億）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）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クエリは少なくとも 8.5 秒かかる。だがこれが事実ではないことはわかるだろう。Web

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

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クエリを実行する。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
- サンプルコードと記号の表記法

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。

1.2 インターフェイス

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、あるインターフェースに対する複数の実装が考えられる。例えば、Chapter 2 では配列を、Chapter 3 ではポインタを使って `List` インターフェースを実装する。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。

1.2.1 Queue、Stack、Deque インターフェース

`Queue` インターフェースは要素を追加したり、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。これは食料品店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します。同点要素が複数あるときは、そのうちのいずれかを適当に選ぶ。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

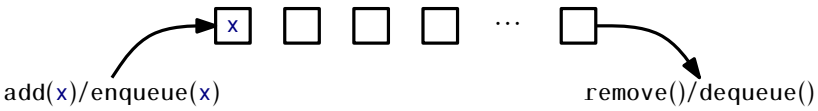


図 1.1: FIFO Queue.

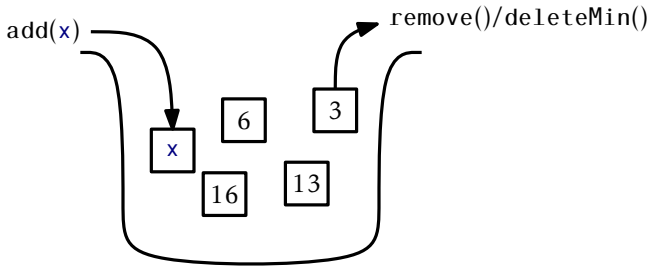


図 1.2: A priority Queue.

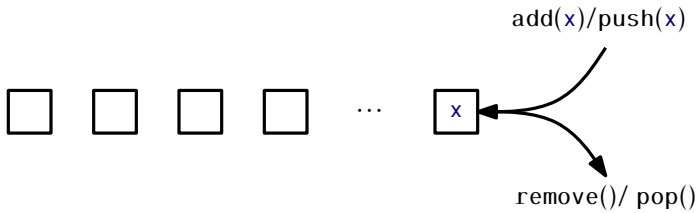


図 1.3: スタック

ら、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、図 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。これはプレートを積むように視覚化するとよい。プレート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 は前と後ろのある要素の列を表す。列の前または後ろ

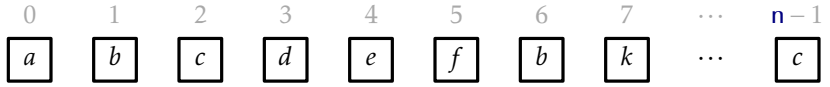


図 1.4: List は 0,1,2,...,n − 1 で添え字づけられた列を表現する。この List で get(2) を実行すると値 c が返ってくる。

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() だけを使って実装できる。一方 FIFO キューには addLast(x) と removeFirst() を使えばよい。

1.2.2 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 インターフェイ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。

これらの操作を使って、Deque インターフェースは簡単に実装できる。

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

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。

1.2.3 USet インターフェース：順序付けられていない要素の集まり

USet インターフェースは重複がなく、順序付けられていない要素の集まりを表現する。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 は等しいと判定される別のオブジェクトかもしれないためだ。こうすると、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 に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。

1.2.4 SSet インターフェース：ソートされた要素の集まり

SSet インターフェースは順序づけされた要素の集まりを表現する。SSet 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) だ。

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気づいていない人もいる。 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。

1.3 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。[25].

1.3.1 指数と対数

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

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

x が負の整数なら、$b^x = 1/b^{-x}$ である。$x = 0$ なら、$b^x = 1$ である。b が整数でないときも、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 2 であることが多い。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。

対数の大雑把なイメージを持つ方法を紹介する。$\log_b k$ 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。

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

1.3.2 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。

1.3.3 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。実行時間の話をしているときは、実際は操作に必要なコンピュータ命令数に注目する。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、いわゆるビッグオー記法を使う。$f(n)$ 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。

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

図形的に考える。n が十分に大きくなると $c \cdot f(n)$ に上から抑えられる関数 $g(n)$ 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。

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

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、高い視点からより複雑な関数を分析することも可能だ。 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 $f_1(\mathsf{n}) = 15\mathsf{n}$ と $f_2(n) = 2\mathsf{n}\log\mathsf{n}$ の増加を比べたものだ。$f_1(\mathsf{n})$ は複雑な線形時間アルゴリズムの実行時間、$f_2(\mathsf{n})$ 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 $f_1(\mathsf{n})$ はより大きいが $f_2(\mathsf{n})$、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 $f_1(\mathsf{n})$ 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 $O(\mathsf{n}) \subset O(\mathsf{n}\log\mathsf{n})$ 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、この本ためには次の定義で十分だ。

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

この定義で我々が気にしていることがわかるのだ。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。我々の目的はこれでよいのだが、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。

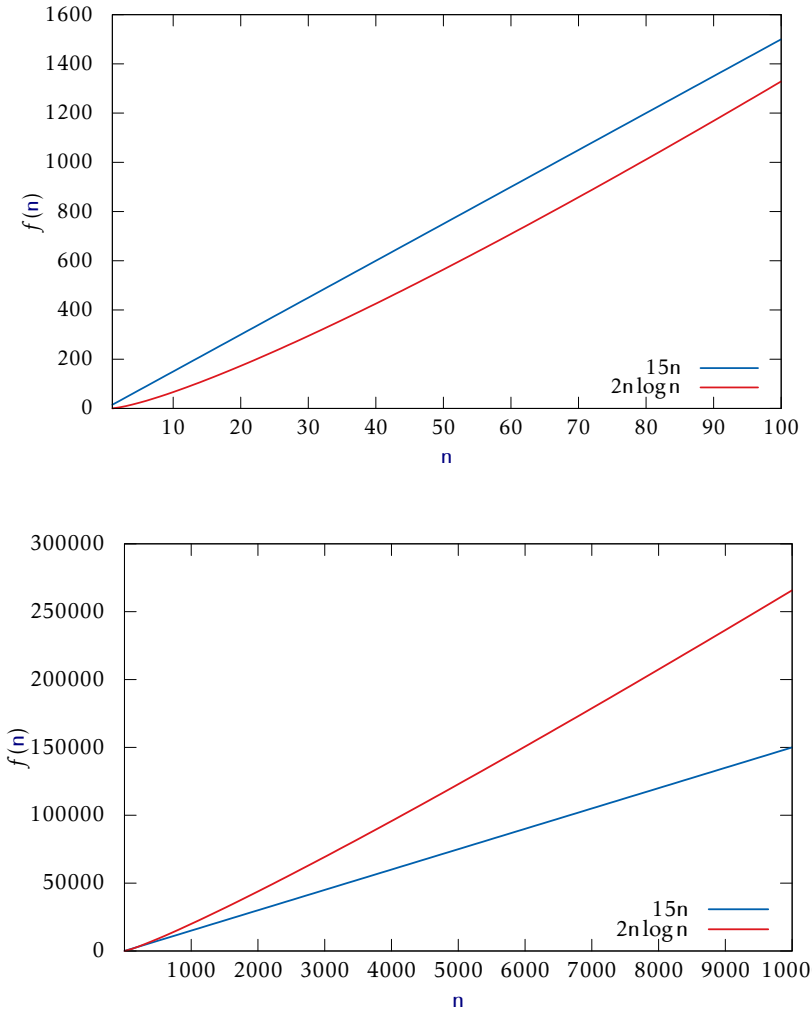


図 1.5: 15n 対 2nlogn のプロット

1.3.4 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。そしせ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。

1.4 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 AND や OR、排他的論理和）である。

どのセルも一定の時間でに読み書きできる。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。

1.5 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は常に満たすことにする。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の中でのものである。

最悪、償却、期待実行時間の違いを理解するのに、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。

1.6 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。

1.7 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.

1.8 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[28] の影響を受けている。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[25] の素晴らしい（そしてフリーの）本が扱っている。丁寧な微積分の教科書として、無料で手に入る Thompson[41] 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定義が書かれている。

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`	$O(\log n)$	$O(\log n)$	§ ??
`BinaryTrie`^{I}	$O(w)$	$O(w)$	§ ??
`XFastTrie`^{I}	$O(\log w)^{A,E}$	$O(w)^{A,E}$	§ ??
`YFastTrie`^{I}	$O(\log w)^{A,E}$	$O(\log w)^{A,E}$	§ ??

(Priority) Queue implementations			
	`findMin()`	`add(x)/remove()`	
`BinaryHeap`	$O(1)$	$O(\log n)^{A}$	§ ??
`MeldableHeap`	$O(1)$	$O(\log n)^{E}$	§ ??

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

表 1.2: Summary of `SSet` and priority `Queue` implementations.

基礎的な確率論についての、特にコンピュータ・サイエンスに関連するものとして Ross[36] の教科書はおすすめである。漸近記法や確率論などを含む Graham, Knuth, and Patashnik[18] 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。

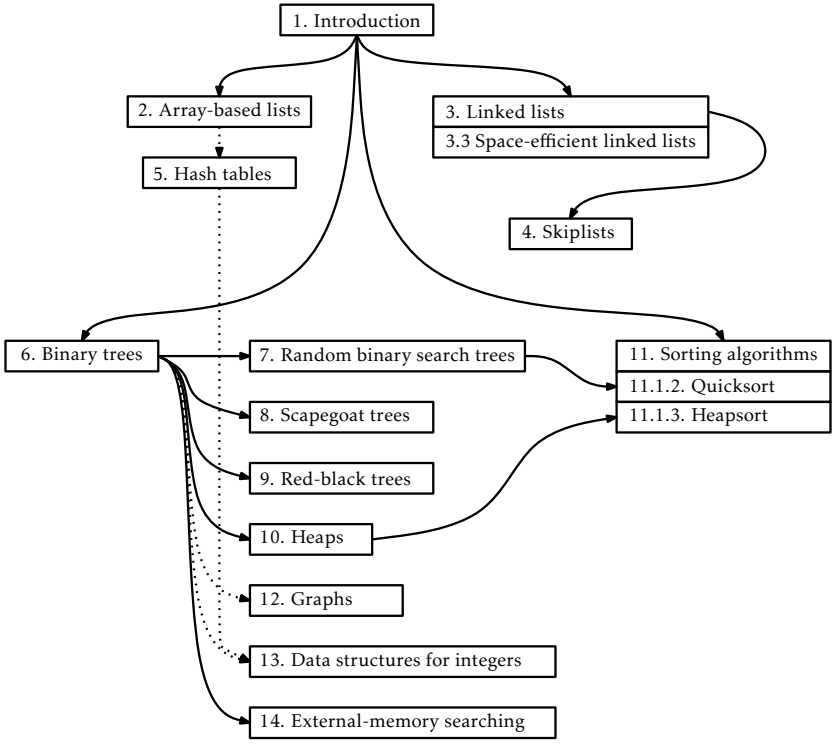


図 1.6: The dependencies between chapters in this book.

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、偶数番目の行の後に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、先頭を含む部分列（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、List・Queue インターフェースの実装について検討する。*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 つめの点が重要だ。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。

2.1 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
```
array<T> a;
int n;
int size() {
  return n;
}
```

2.1.1 基本

get(i) や set(i,x) を使って ArrayStack の要素を読み書きする方法は簡単である。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（リサイズ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];
```

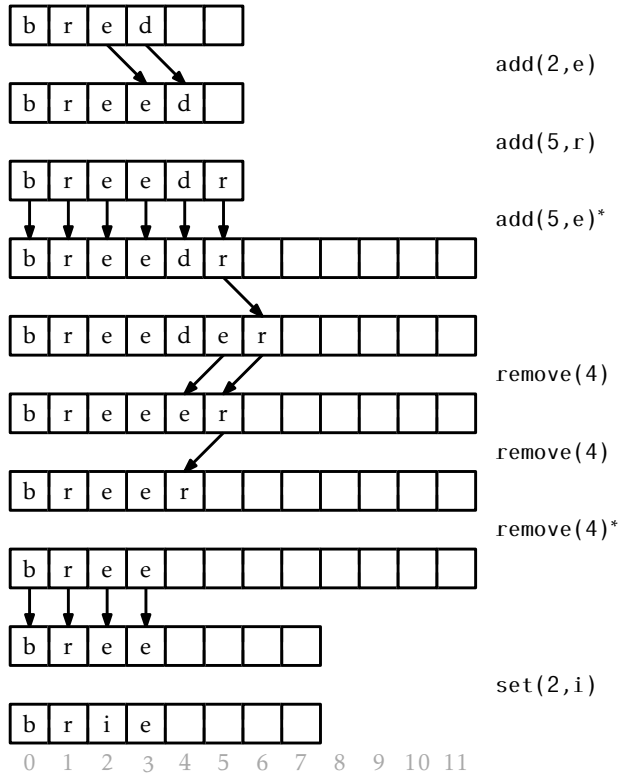


図 2.1: add(i,x)・remove(i) を ArrayStack に実行する。矢印は要素のコピーを表す。resize() を呼ぶ操作にはアスタリスクを付した。

```
  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。

2.1.2 拡張と収縮

resize() の実装は単純だ。大きさ 2n の新しい配列 b を割当て、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() {
  array<T> b(max(2 * n, 1));
  for (int i = 0; i < n; i++)
    b[i] = a[i];
  a = b;
}
```

resize() の実際のコストの計算も簡単だ。大きさ 2n の配列 b を割当て、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、m 個の add(i,x)・remove(i) からなる操作列の間に呼ばれる resize() すべてのことを考える。

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 $\mathtt{a.length} = \mathtt{n} = \mathtt{n}_i$ が成り立つときだ。直前の resize() を考えよう。この resize() の直後、a の大きさは a.length だが a の要素数は $\mathtt{a.length}/2 = \mathtt{n}_i/2$ 以下であった。しかし a の要素数は今では $\mathtt{n}_i = \mathtt{a.length}$ 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 $\mathtt{a.length} \geq 3\mathtt{n} = 3\mathtt{n}_i$ である。前の resize() の直後では a の要素数は $\mathtt{a.length}/2 - 1$ 以下であった。[*1] 今 a には $\mathtt{n}_i \leq \mathtt{a.length}/3$ 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。 □

2.1.3 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 $\mathtt{n} = 0$ かつ $\mathtt{a.length} = 1$ 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。

2.2 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() {
  array<T> b(max(1, 2*n));
  std::copy(a+0, a+n, b+0);
  a = b;
}
void add(int i, T x) {
  if (n + 1 > a.length) resize();
  std::copy_backward(a+i, a+n, a+n+1);
  a[i] = x;
  n++;
```

```
}
```

これら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。

2.3 ArrayQueue : 配列を使ったキュー

この節では FIFO（先入れ先出し）キューを実装するデータ構造 ArrayQueue を紹介する。(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

```
array<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。

ArrayQueue

```
bool 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

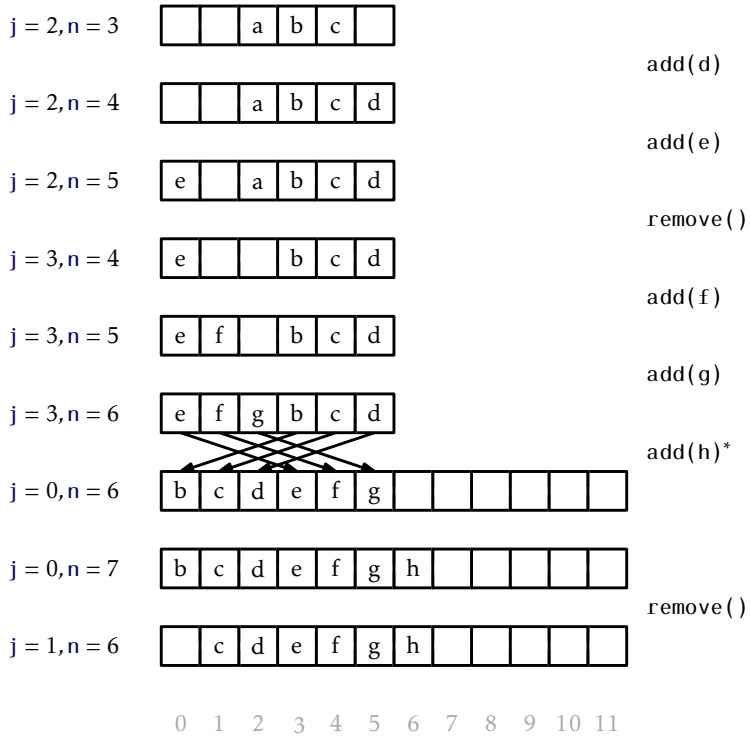


図 2.2: A sequence of add(x) and remove(i) operations on an ArrayQueue. Arrows denote elements being copied. Operations that result in a call to resize() are marked with an asterisk.

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() {
  array<T> b(max(1, 2*n));
```

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

2.3.1 要約

次の定理は ArrayQueue の性能を整理するものだ。

Theorem 2.2. *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。*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。

2.4 ArrayDeque：配列を使った高速な双方向キュー

前節の ArrayQueue は、一方の端からは追加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

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

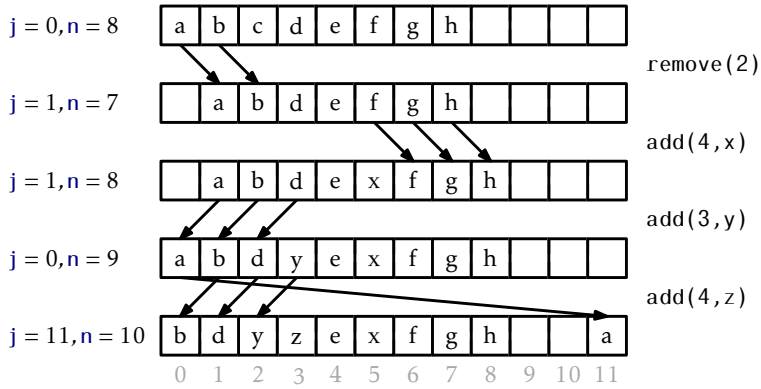


図 2.3: A sequence of add(i,x) and remove(i) operations on an ArrayDeque. Arrows denote elements being copied.

```
  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;
}
```

2.4.1 要約

次の定理は ArrayDeque の性能を整理するものだ。ArrayDeque は List インターフェースを実装する。resize() のコストを無視すると、ArrayDeque における各操作の実行時間は、

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

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。

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

次は 2 つの ArrayStack を使って 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扱う価値がある。

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

```
ArrayStack<T> front;
```

```
ArrayStack<T> back;
```

DualArrayDeque は要素数 n を明示的に保持しない。n = front.size() + back.size() なのでその必要がないのだ。しかし 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

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

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);
  } 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。

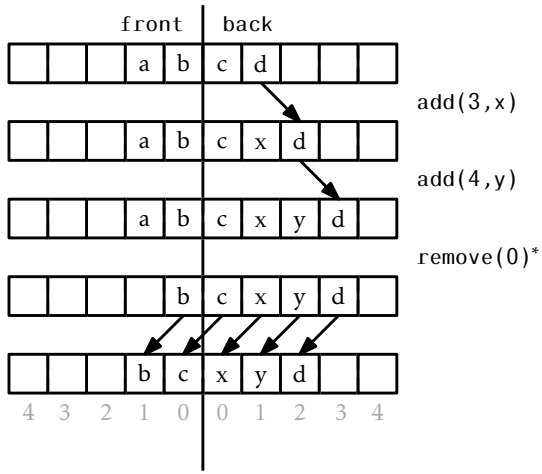


図 2.4: A sequence of add(i,x) and remove(i) operations on a DualArrayDeque. Arrows denote elements being copied. Operations that result in a rebalancing by balance() are marked with an asterisk.

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() のこと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 $O(\texttt{n}) = O(\texttt{i}) = O(\texttt{n} - \texttt{i})$ である。以上をまとめると次のようになる。

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

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

2.5.1 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() {
  if (3*front.size() < back.size()
      || 3*back.size() < front.size()) {
    int n = front.size() + back.size();
    int nf = n/2;
    array<T> af(max(2*nf, 1));
    for (int i = 0; i < nf; i++) {
      af[nf-i-1] = get(i);
    }
    int nb = n - nf;
    array<T> ab(max(2*nb, 1));
    for (int i = 0; i < nb; i++) {
      ab[i] = get(nf+i);
    }
    front.a = af;
    front.n = nf;
    back.a = ab;
    back.n = nb;
  }
}
```

ここまでで分析するようなことは特にないだろう。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 balance() 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 $\mathtt{n}/2-1$ 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。

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

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。

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

要素を動かす balance() の直前には $3\mathtt{front.size}() < \mathtt{back.size}()$ 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。 □

2.5.2 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 $O(1+\min\{i,n-i\})$ 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 $O(1+\min\{i,n-i\})$ 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 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。この配列では使わ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 $0,1,\ldots,r-1$ と添字付けられる。Figure 2.5 を見てほしい。ブ

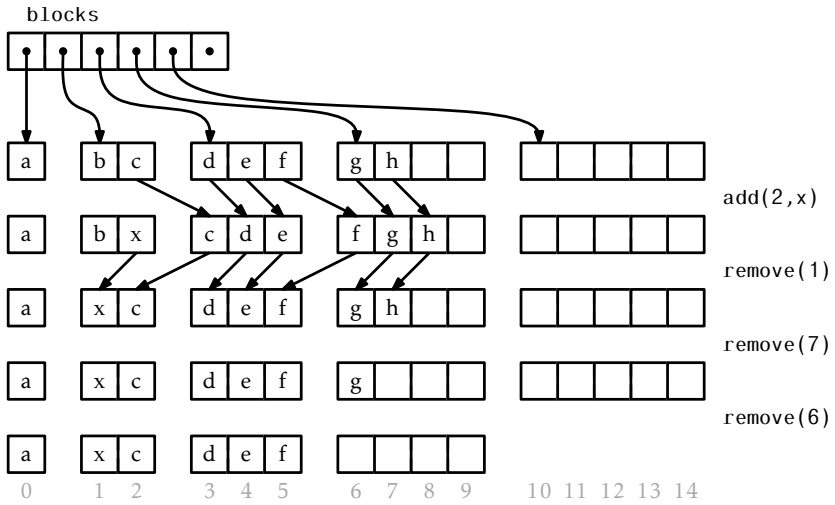


図 2.5: A sequence of add(i,x) and remove(i) operations on a RootishArrayStack. Arrows denote elements being copied.

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

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

リストの要素はブロック内で順番に配置される。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、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、$0,\ldots,b-1$ 番目のブロックにおける要素数の合計は $b(b+1)/2$ である。そのため、i は

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

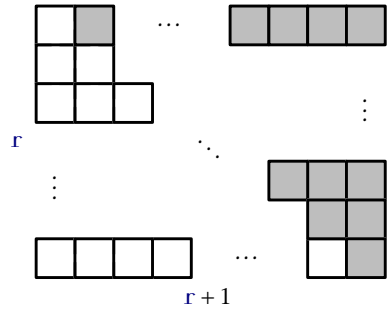


図 2.6: The number of white squares is $1+2+3+\cdots+r$. The number of shaded squares is the same. Together the white and shaded squares make a rectangle consisting of $r(r+1)$ squares.

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 を求めるのはもう少し難しい。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 + sqrt(9 + 8*i)) / 2.0;
  int b = (int)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 blacks 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 $i,\ldots,n-1$ 番目の要素をそれぞれ右にひとつずらし、新たな i 番目の要素を入れるための隙間を作る。

RootishArrayStack

```
void add(int i, T x) {
  int r = blocks.size();
  if (r*(r+1)/2 < n + 1) grow();
  n++;
  for (int j = n-1; j > i; j--)
```

```
    set(j, get(j-1));
  set(i, x);
}
```

grow() メソッドはやってほしいことをしてくれる、つまり新しいブロックを追加してくれる。

RootishArrayStack

```
void grow() {
  blocks.add(blocks.size(), new T[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) {
```

```
            delete [] 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。

2.6.1 拡張・収縮の分析

上の add(i,x)・remove(i) の解析では grow()・shrink() のことを考慮していなかった。まず、ArrayStack.resize() とは違い、grow() 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。

2.6.2 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) の後には 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 要素を含む RootishArrayStack のブロック数を r とすれば次の関係が成り立つ。

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

ここでまた二次式の解を考えれば次の式が成り立つ。

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

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。

2.6.3 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。

[*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

2.6.4 Computing Square Roots

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

2.7 ディスカッションと練習問題

この章で説明したデータ構造は民俗学のようなものだ。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 [22, Section 2.2.2] により議論されている。

Brodnik *et al.* [6] が RootishArrayStack についての記述であり、Section 2.6.2 で述べたような下界 $\sqrt{n}$ を示したようである。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 [17] 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。)

RandomQueue における add(x)・remove() の償却実行時間は定数でなけれ

てほしい。

ばならない。

Exercise 2.3. Treque (triple-ended queue) を設計・実装せよ。これは 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 $i \in \{0,\ldots,a.length\}$ 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。なお ArrayDeque や DualArrayDeque に対しての rotate(r) の実行時間は $O(1+\min\{r, n-r\})$ である必要がある。

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。これは 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、要素を含むノードから構成される。参照（ポインタ）を使ってノードを繋げて列を作る。まずは単方向連結リストを紹介する。これを使って 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。その代わりに 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。一方でより動的であることが主な長所だ。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。

3.1 SLList：単方向連結リスト

SLList（singly-linked list、単方向連結リスト）は Node の列である。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 {
public:
```

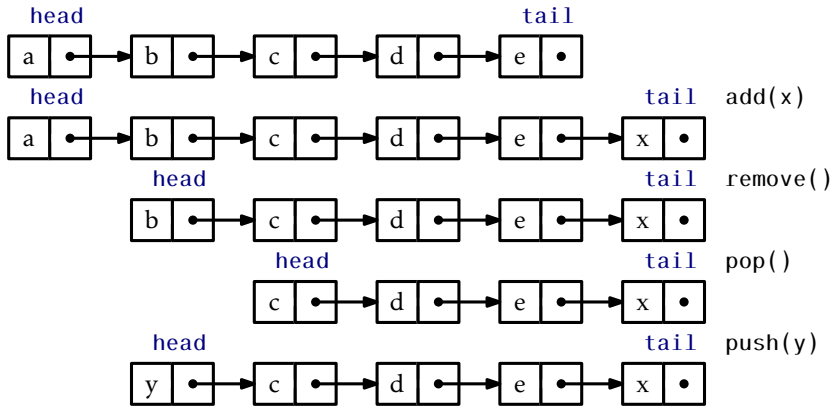


図 3.1: A sequence of Queue (add(x) and remove()) and Stack (push(x) and pop()) operations on an SLList.

```
  T x;
  Node *next;
  Node(T x0) {
    x = 0;
    next = NULL;
  }
};
```

効率のため SLList は変数 head・tail で列の先頭・末尾のノードを保持している。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(x);
  u->next = head;
  head = u;
  if (n == 0)
    tail = u;
  n++;
  return x;
}
```

pop() では、SLList がから出ないことを確認し、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;
  Node *u = head;
  head = head->next;
  delete u;
  if (--n == 0) tail = NULL;
  return x;
}
```

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

3.1.1 キュー操作

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

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

一方で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 になるのだ。

SLList
```
bool add(T x) {
  Node *u = new Node(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。

3.1.2 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。

3.2 DLList: 双方向連結リスト

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

DLList

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

SLList を実装するとき注意しなければならないことがいくつかあった。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注意が必要であった。DLList ではこういう特殊な場合というのがかなり増える。DLList におけるこういう問題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。これはデータを含まないノードで、ただ場所だけを占める。こうするとノードを特別扱いする必要がなくなるのだ。すべてのノードには next と prev がある。dummy は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

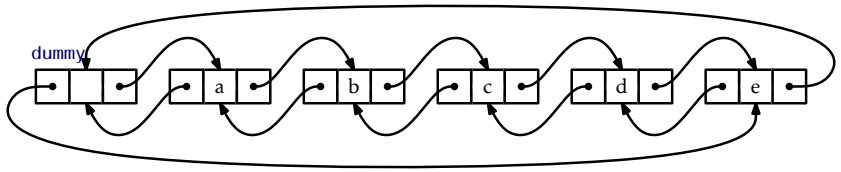


図 3.2: A DLList containing a,b,c,d,e.

なサイクルになる。

DLList

```
Node dummy;
int n;
DLList() {
  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;
  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;
}
```

この際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。

3.2.1 追加と削除

DLList におけるノード w の参照を持っていて、ノード u を w の直前に追加したいときは、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;
```

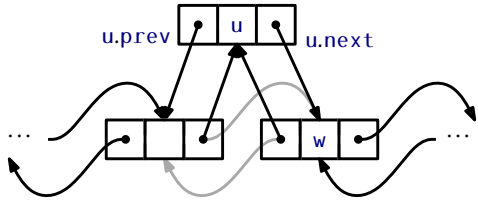


図 3.3: Adding the node u before the node w in a DLList.

```
  u->next = w;
  u->next->prev = u;
  u->prev->next = u;
  n++;
  return u;
}
```

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

DLList

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

add(i,x) の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;
```

```
  delete w;
  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);
  T x = w->x;
  remove(w);
  return x;
}
```

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

3.2.2 要約

次の定理は DLList の性能をまとめたものである。

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。

3.3 SEList：空間効率のよい連結リスト

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

SEList(space-efficient list) はシンプルなアイデアでこの無駄な領域を削減する。DLList のように一個ずつ要素を入れるのではなく、複数の要素を含むブロック（配列）をデータとして入れるのである。もう少し正確に説明する。SEList のパラメータとしてブロックサイズ b がある。SEList の個々のノードは $b+1$ 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 backing array 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 : public ArrayDeque<T> {
public:
  BDeque(int b) {
    n = 0;
    j = 0;
    array<int> z(b+1);
    a = z;
  }
  ~BDeque() { }
  // C++ Question: Why is this necessary?
```

```
  void add(int i, T x) {
    ArrayDeque<T>::add(i, x);
  }
  bool add(T x) {
    ArrayDeque<T>::add(size(), x);
    return true;
  }
  void resize() {}
};
```

SEList はブロックの双方向連結リストである。

SEList

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

SEList

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

XXX: rubyimport

3.3.1 必要なメモリ量

SEList はブロックに含む要素数に次のような強い制限がある。末尾でないブロックはみな $b-1$ 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。

3.3.2 要素を検索

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

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

SEList
```
class Location {
public:
  Node *u;
  int j;
  Location() { }
  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
```
void getLocation(int i, Location &ell) {
  if (i < n / 2) {
```

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

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, l);
```

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

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。

3.3.3 要素の追加

SEList への要素の追加はもう少し複雑だ。一般的な場合を考える前に、より簡単な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

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

add(i,x) でリストの中に要素を追加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

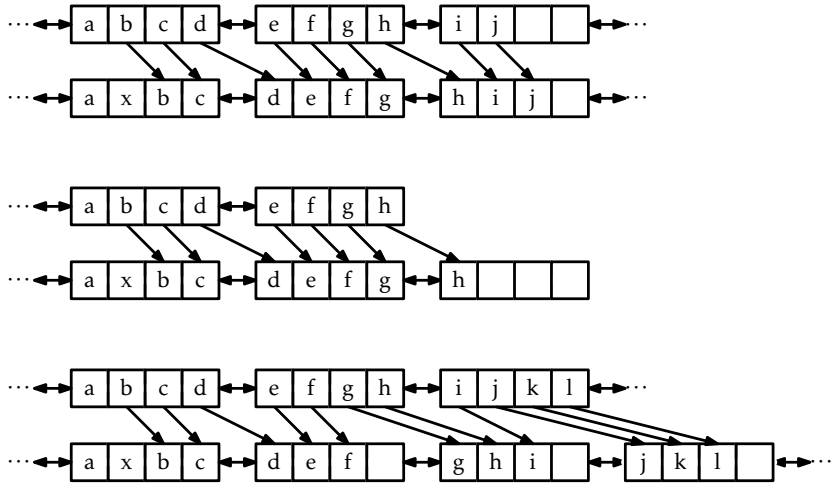


図 3.4: The three cases that occur during the addition of an item x in the interior of an SEList. (This SEList has block size b = 3.)

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

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

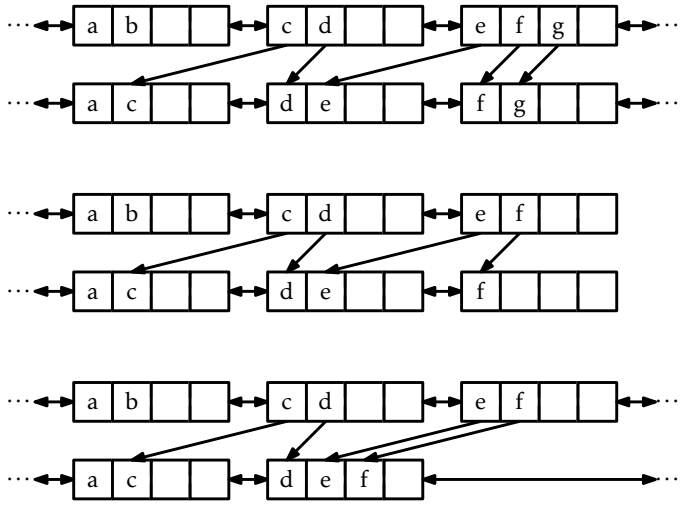


図 3.5: The three cases that occur during the removal of an item x in the interior of an SEList. (This SEList has block size b = 3.)

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。（この解析の妥当性はあとで償却法で説明する。）

3.3.4 要素の削除

SEList から要素を削除するのは要素を追加するのに似ている。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貸してくれるノードを探す。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（$r+1 \leq b$ ステップ以内に）リストの末尾に到達する。この場合、u_r は末尾のノードなので u_r のブロックには $b-1$ 個以上の要素を含むという制約がない。そのためひとつめの場合と同様に u_r から要素を借りてきて u_0 に足してよい。この結果 u_r のブロックが空になったら削除する。
3. b ステップの間に $b-1$ 個より多くの要素を含むブロックが見つからない。この場合 $u_0,\ldots,u_{b-1}$ はいずれも要素数 $b-1$ のブロックの列である。gather を呼び、$b(b-1)$ 要素を $u_0,\ldots,u_{b-2}$ に集める。これらの $b-1$ 個のブロックはいずれもちょうど b 要素を含むようになる。そして空になった u_{b-1} を削除する。すると、u_0 のブロックは b 要素を含むようになったので、ここから適当な要素を削除できる。

SEList

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

```
  if (u->d.size() == 0)
    remove(u);
  n--;
  return y;
}
```

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

3.3.5 spread と gather の償却解析

続いて、add(i,x)・remove(i) で実行されるかもしれない gather(u)・spread(u) のコストを考える。はじめにコードを示す。

SEList

```
void spread(Node *u) {
  Node *w = u;
  for (int j = 0; j < b; j++) {
    w = w->next;
  }
  w = addBefore(w);
  while (w != u) {
    while (w->d.size() < b)
      w->d.add(0, w->prev->d.remove(w->prev->d.size()-1));
    w = w->prev;
  }
}
```

SEList

```
void gather(Node *u) {
  Node *w = u;
  for (int j = 0; j < b-1; j++) {
```

```
    while (w->d.size() < b)
      w->d.add(w->next->d.remove(0));
    w = w->next;
  }
  remove(w);
}
```

いずれの実行時間においても支配的なのは二段階ネストしたループである。内側・外側いずれのループも最大 $b+1$ 回実行されるのでいずれの操作の実行時間も $O((b+1)^2) = O(b^2)$ 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。 □

3.3.6 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 みたいなものだ。これらの間の調整は、リストへの要素の追加・削除の時間と、特定の要素を見つける時間のトレードオフでもある。

3.4 ディスカッションと練習問題

単方向連結リストも双方向連結リストも 40 年以上前からプログラムで使われており、研究され尽くしているテクニックである。例えば Knuth の [22, Sections 2.2.3–2.2.5] で議論されている。SEList でさえもデータ構造の有名な練習問題である。SEList は *unrolled linked list*[38] と呼ばれることも

[*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。

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。リスト自体は、dummy を指すものと dummy.next のふたつのポインタを持つ。(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 [39] 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。この章を戻ってはいけない。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。

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. absort(l2) を実装せよ。これは DLList l2 を引数に取り、l2 を空にし、その中身を元の順でレシーバーに追加する。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 **??**扱う。

1. DLList の takeFirst(l2) 操作を実装せよ。この操作は l2 の先頭ノードを取り出しレシーバに追加するものだ。これは新しいノードを作ら

ないことを除けば、add(size(),l2.remove(0)) と等価である。

2. DLList の静的メソッド merge(l1,l2) を実装せよ。これはふたつの整列済みのリスト l1・l2 を統合し、その結果を含む新たな整列済みリストを返す。この後では l1・l2 は空になっている。例えば l1 の要素は a,c,d、l2 の要素は b,e,f であるとき、このメソッドは a,b,c,d,e,f を含むリストを返す。
3. DLList の sort() メソッドを実装せよ。これはマージソートを使ってリストの全ての要素を整列するものである。この再帰的なアルゴリズムは次のように動作する。
   (a) リストの要素数が 0 または 1 ならなにもしない。
   (b) そうでないなら truncate(size()/2) によってリストをほぼ等しい大きさのふたつのリスト l1 と l2 に分割する。
   (c) 再帰的に l1 を整列する。
   (d) 再帰的に l2 を整列する。
   (e) 最後に l1 と l2 を統合して一つの整列済みリストとする。

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

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)・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にコインを投げ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。

4.1 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、

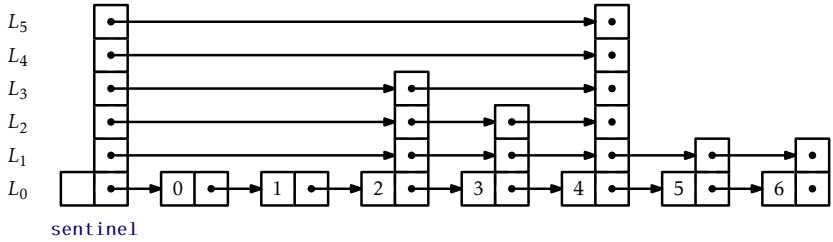


図 4.1: A skiplist containing seven elements.

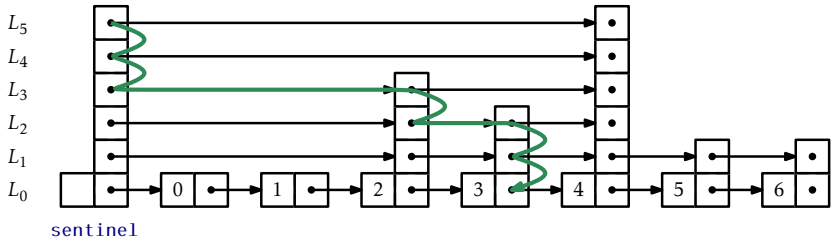


図 4.2: The search path for the node containing 4 in a skiplist.

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
```
struct Node {
  T x;
  int height;     // length of next
  Node *next[];
};
```

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。

4.2 SkiplistSSet：効率的な SSet

SkiplistSSet はスキップリストを使った SSet インターフェースの実装である。ここでは、L_0 は SSet の要素を整列して格納する。find(x) は探索経路に沿って y ≥ x を満たす最小の y を探す。

SkiplistSSet
```
Node* findPredNode(T x) {
  Node *u = sentinel;
  int r = h;
  while (r >= 0) {
    while (u->next[r] != NULL
              && compare(u->next[r]->x, x) < 0)
      u = u->next[r]; // go right in list r
```

```
    r--; // go down into list r-1
  }
  return u;
}
T find(T x) {
  Node *u = findPredNode(x);
  return u->next[0] == NULL ? null : u->next[0]->x;
}
```

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

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] は L_r 内の L_{r-1} に下がるパスのあるノードである。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

```
bool add(T x) {
  Node *u = sentinel;
  int r = h;
  int comp = 0;
  while (r >= 0) {
    while (u->next[r] != NULL
               && (comp = compare(u->next[r]->x, x)) < 0)
      u = u->next[r];
    if (u->next[r] != NULL && comp == 0)
      return false;
    stack[r--] = u;        // going down, store u
  }
  Node *w = newNode(x, pickHeight());
  while (h < w->height)
    stack[++h] = sentinel; // height increased
  for (int i = 0; i <= w->height; i++) {
    w->next[i] = stack[i]->next[i];
    stack[i]->next[i] = w;
  }
  n++;
  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ることで行える。x を探す途中にノード u

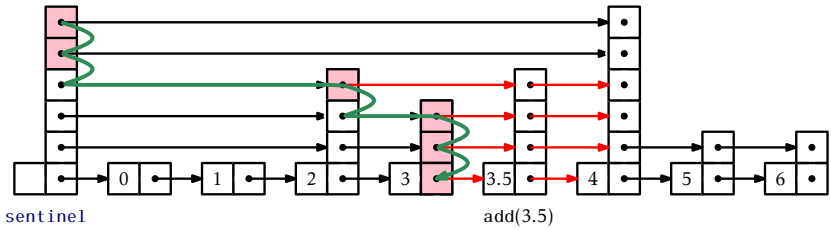


図 4.3: Adding the node containing 3.5 to a skiplist. The nodes stored in stack are highlighted.

から下に向かうとき、u.next.x = x なら u を繋ぎ替える。

SkiplistSSet

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

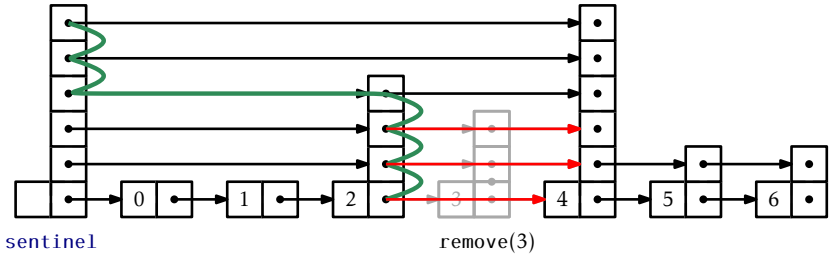


図 4.4: Removing the node containing 3 from a skiplist.

```
    delete del;
    n--;
  }
  return removed;
}
```

4.2.1 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。

4.3 SkiplistList：効率的なランダムアクセス List

SkiplistList はスキップリストを使った List インターフェースの実装だ。SkiplistList では、L_0 はリストの要素をリストにおける順序通りに含む。SkiplistSSet と同様に、要素の追加・削除・読み書きのいずれの実行時間も $O(\log n)$ である。

これを可能にするためにはまず L_0 における i 番目の要素を見つける方法が必要だ。このための最も簡単な方法はリスト L_r における辺の長さを定義することだ。L_0 における辺の長さをいずれも 1 とする。$L_r (r > 0)$ の辺 e の辺の

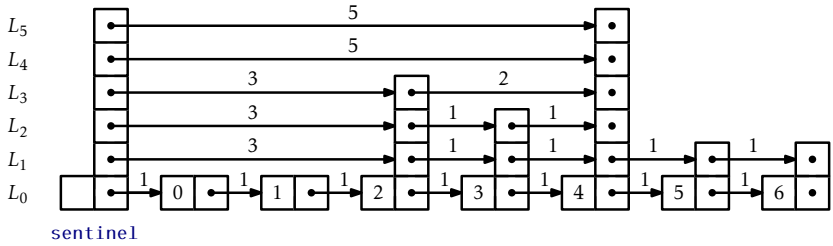


図 4.5: The lengths of the edges in a skiplist.

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

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;   // the index of the current node in list 0
  while (r >= 0) {
```

```
    while (u->next[r] != NULL && j + u->length[r] < i) {
      j += u->length[r];
      u = u->next[r];
    }
    r--;
  }
  return u;
}
```

SkiplistList

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

get(i)・set(i,x) において大変なのは 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。

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 $r \le k$ 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。

探索経路上の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

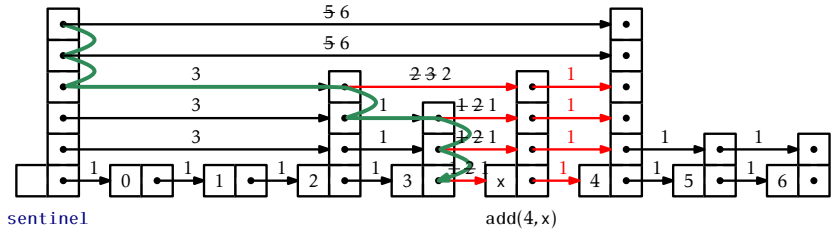


図 4.6: Adding an element to a SkiplistList.

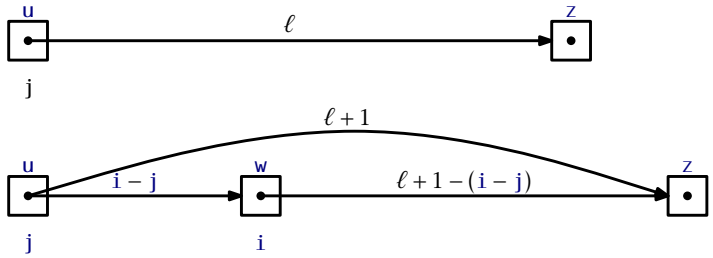


図 4.7: Updating the lengths of edges while splicing a node w into a skiplist.

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 = newNode(x, pickHeight());
  if (w->height > h)
    h = w->height;
  add(i, w);
}
```

SkiplistList

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

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;
```

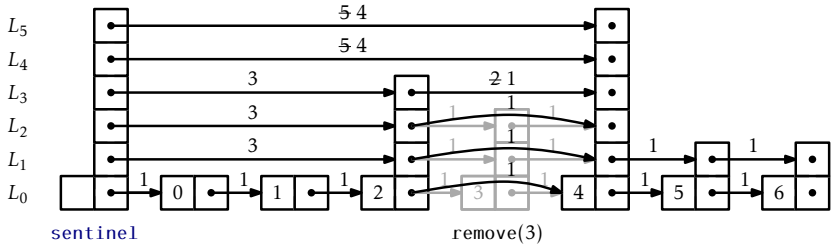


図 4.8: Removing an element from a SkiplistList.

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

```
}
```

4.3.1 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。

4.4 スキップリストの解析

この節では高さ・大きさ・探索経路の長さの期待値を解析する。ここでは基本的な確率論の知識を前提とする。いくつかの証明は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。

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

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。

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

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

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

[*2] Section **??**を参照せよ。

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

□

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

[*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。

る。このとき、次の式が成り立つ。

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

□

次の定理はこの節の結果をまとめるものだ。

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。

4.5 ディスカッションと練習問題

スキップリストを提案したのは Pugh [33] であり、多くの拡張や応用も提案されている。[32] 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。 [21, 20, 30]. 決定的な変種や [27] 偏りのある変種 [2, 15]、適応的な変種 [5] 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。[40, 34] 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。[19].

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。$u.\texttt{next}[r] = u.\texttt{next}[r-1]$ 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。（85 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、l1 の要素が a,b,c、l2 の要素が d,e,f であるとき、l1.absorb(l2) を呼ぶと、l1 の要素は a,b,c,d,e,f になり l2 は空になる。このメソッドの実行時間は $O(\log n)$ でなければならない。

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。

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

ChainedHashTable とはチェイン法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

```
array<List> 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

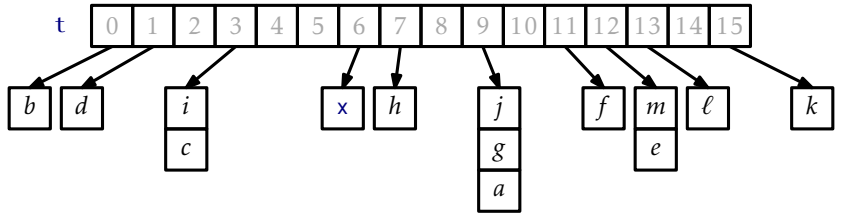


図 5.1: An example of a ChainedHashTable with n = 14 and t.length = 16. In this example hash(x) = 6

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

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

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

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 $O(n/\mathtt{t.length}) = O(1)$ 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。

5.1.1 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(T x) {
  return ((unsigned)(z * hashCode(x))) >> (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。）

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 $\Pr\{\mathsf{hash}(x) = \mathsf{hash}(y)\} \leq 2/2^d$ 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。

[*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 5.2. 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。

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

ここで、Lemma 5.1 より、$E[I_y] \le 2/2^d = 2/\text{t.length}$ 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。

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

□

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 $(b_r, \ldots, b_0)_2$ と書いて、$\sum_{i=0}^{r} b_i 2^i$ を表す。

ここで、b_i は 0 か 1 である。すなわち、$(b_r,\dots,b_0)_2$ 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。 □

5.1.2 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。

5.2 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
```
array<T> t;
int n;   // number of values in T
int q;   // number of non-null entries in T
int d;   // t.length = 2^d
```

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 && t[i] == x) 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
```
bool add(T x) {
  if (find(x) != null) return false;
  if (2*(q+1) > t.length) resize();   // max 50% occupancy
  int i = hash(x);
  while (t[i] != null && t[i] != del)
```

```
    i = (i == t.length-1) ? 0 : i + 1; // increment i
  if (t[i] == null) q++;
  n++;
  t[i] = x;
  return true;
}
```

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 $t[i'] = \texttt{null}$ である添え字 i' を見つけるまで探す。$t[i'] = x$ ならば $t[i'] = \texttt{del}$ とし true を返す。$t[i'] = \texttt{null}$ 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 == 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++;
  array<T> tnew(1<<d, null);
  q = n;
  // insert everything into tnew
  for (int k = 0; k < t.length; k++) {
    if (t[k] != null && t[k] != del) {
      int i = hash(t[k]);
      while (tnew[i] != null)
        i = (i == tnew.length-1) ? 0 : i + 1;
      tnew[i] = t[k];
    }
  }
  t = tnew;
}
```

5.2.1 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、$\mathtt{t}$ の添字はすべて $\mathtt{t.length}$ で剰余を取っているとする。つまり単に $\mathtt{t}[\mathtt{i}]$ と書いても $\mathtt{t}[\mathtt{i} \bmod \mathtt{t.length}]$ のことである。

XXX: run の訳語 $\mathtt{i}$ 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 $\mathtt{null}$ でなく、$\mathtt{t}[\mathtt{i}-1] = \mathtt{t}[\mathtt{i}+k] = \mathtt{null}$ であることをいう。$\mathtt{t}$ の $\mathtt{null}$ でない要素の数は $\mathtt{q}$ で、$\mathtt{add}(\mathtt{x})$ は常に $\mathtt{q} \le \mathtt{t.length}/2$ であることを保証する。直前の $\mathtt{rebuild}()$ 以後、$\mathtt{t}$ に挿入された $\mathtt{q}$ 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. $\mathtt{i}$ 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、$\mathtt{i}$ 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. $\mathtt{i}$ 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]

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。

$\mathtt{t.length}$ が最小値を取るとき p_k は最大値を取る。またデータ構造は不変

[*2] p_k は $\mathtt{i}$ 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る。

条件 t.length ≥ 2q を保つ。よって次の式が成り立つ。

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

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

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,...,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。

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

最後の変形 $\sum_{k=0}^{\infty}k^2\cdot O(c^k)$ では指数級数の性質を使っている。[*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。

5.2.2 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。

[*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、$\frac{(k+1)^2 c^{k+1}}{k^2 c^k} < 1$ を満たす。

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

5.2.3 Tabulation Hashing

LinearHashTable の解析では強い仮定を置いていた。すなわち、任意の相異なる要素 $\{x_1,\ldots,x_n\}$ についてそのハッシュ値 $\mathtt{hash}(x_1),\ldots,\mathtt{hash}(x_n)$ が独立に一様な確率で $\{0,\ldots,\mathtt{t.length}-1\}$ 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[hashCode(x) >> 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) {
  unsigned h = hashCode(x);
  return (tab[0][h&0xff]
        ^ tab[1][(h>>8)&0xff]
        ^ tab[2][(h>>16)&0xff]
        ^ tab[3][(h>>24)&0xff])
      >> (w-d);
}
```

この場合、tab は 4 つの列と $2^{32/4} = 256$ 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。

5.3 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。

5.3.1 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 C++ char is typically 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。

5.3.2 複合オブジェクトのハッシュ値

複合オブジェクトのハッシュ値は、その構成要素のハッシュ値を組み合わせて計算する。これは思うほど簡単でない。いい感じのやり方はたくさん思いつく（例えばビット単位の排他的論理和を計算する）が、そのうちの多くはうまくいかない。（5.7 から 5.9 を参照せよ。）

しかし $2w$ ビットの算術精度があれば単純でロバストな方法がある。$P_0,\ldots,P_{r-1}$ からなる複合オブジェクトがあり、それぞれのハッシュ値は $x_0,\ldots,x_{r-1}$ 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

```
unsigned hashCode() {
  // random number from random.org
  long long z[] = {0x2058cc50L, 0xcb19137eL, 0x2cb6b6fdL};
  long zz = 0xbea0107e5067d19dL;
  long h0 = ods::hashCode(x0);
  long h1 = ods::hashCode(x1);
  long h2 = ods::hashCode(x2);
  return (int)(((z[0]*h0 + z[1]*h1 + z[2]*h2)*zz) >> 32);
```

```
}
```

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。

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

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

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

以上より、次の式が成り立つ。

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

5.3.3 配列と文字列のハッシュ値

前小節の手法はオブジェクトが固定数の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。$\{0,\dots,2^w-1\}$ の要素である w ビットの整数からなる列であるとする。$i \in \{0,\dots,r-1\}$ のうち少なくともひとつ $x_i \neq y_i$ が成り立つと仮定する。このとき、次の式が成り立つ。

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

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

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

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。

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

長さ $r, r'(r > r')$ のふたつの入力があるとき、ふたつの列は添え字 $i = r$ で異なる。このとき (5.6) は次のようになる。

$$\left(\sum_{i=0}^{i=r'-1}(x_i-y_i)z^i + (x_{r'}-p+1)z^{r'} + \sum_{i=r'+1}^{i=r-1}x_i z^i + (p-1)z^r\right) \bmod p = 0$$

これは 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。$x_0,\dots,x_{r-1}$ と $y_0,\dots,y_{r'-1}$ は $\{0,\dots,2^w-1\}$ 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。

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

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

```
unsigned hashCode() {
  long p = (1L<<32)-5;   // prime: 2^32 - 5
```

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

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。

5.4 ディスカッションと練習問題

ハッシュテーブルとハッシュ値は広大で活発な研究分野であり、この章ではほんのさわりを説明しただけである。ハッシュ方のオンライン参考文献一覧は [3]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 [16, 14] や *cuckoo hashing* [29] 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。[7]. Section 5.2.3 で説明した Tabulation hashing は Carter と Wegman の研究 [7] 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。[31].

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。[24, 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。[13] 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。[11] 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。[4]

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

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.*[12] 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。[23, 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

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

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。）[8]

第 6

二分木

この章ではコンピュータサイエンスで最も基本的な構造のうちのひとつである二分木を紹介する。木と呼ばれるのは図示した場合の構造が（森に生えてる）木に似ているためである。二分木の定義は複数ある。数学的には二分木とは連結な有限無向グラフであって、サイクルがなく、すべての頂点の次数が 2 以下であるものである。

コンピュータサイエンスでの二分木には根付きである。次数 2 以下の特別なノード r を、木の根と呼ぶ。すべてのノード u($\neq$ r) について u から r に向かう経路上の二番目のノードを u の親という。それ以外の u に隣接するノードを u の子と呼ぶ。順序付けられた二分木に興味がある事が多い。これは左の子と右の子を区別するということだ。

図示するとき、二分木はふつう根から下に向かって書かれる。根が一番上にあり、左右の子はそれぞれ左下・右下に書かれる。（Figure 6.1）例えば Figure 6.2.a は 9 個のノードを持つ二分木である。

二分木は重要なので、そのための専用の語彙がいくつかある。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 へのパスに含まれるとき、w は u の祖先と呼ばれる。一方 u は w の子孫と呼ばれる。二分木によけるノード 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 の子孫へのパス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 が子を持たないとき、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外部ノードを左の子として持ち、同様に右の子を持たないノードは外部ノードを右の子として持つとする。（Figure 6.2.b を参照）帰納法により、n $\geq$ 1 個の

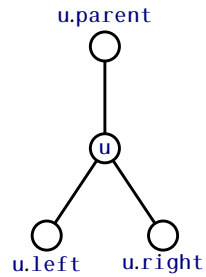


図 6.1: The parent, left child, and right child of the node u in a BinaryTree.

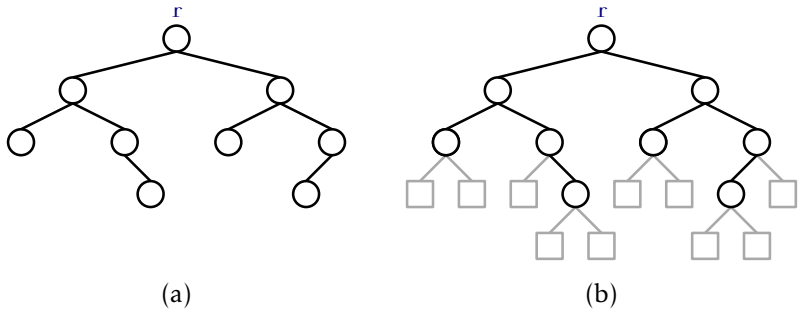


図 6.2: A binary tree with (a) nine real nodes and (b) ten external nodes.

（本物の）ノードを持つ二分木は $n+1$ 個の外部ノードを持つことを示せる。

6.1 BinaryTree：基本的な二分木

二分木のノード u を表す最も単純なやり方は、明示的に（3 つ以下の）隣接するノードを保持することだ。

BinaryTree

```
class BTNode {
  N *left;
  N *right;
  N *parent;
  BTNode() {
```

```
    left = right = parent = NULL;
  }
};
```

隣接する頂点が 3 つもないときはそこには nil を入れる。こうすれば外部ノードと根の親が nil に対応することになる。

すると、二分木自体は根 r への参照として表現できる。

BinaryTree
```
Node *r;    // root node
```

ノード u の深さは u から根への経路をたどるときのステップ数である。

BinaryTree
```
int depth(Node *u) {
  int d = 0;
  while (u != r) {
    u = u->parent;
    d++;
  }
  return d;
}
```

6.1.1 Recursive Algorithms

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最適に計算し、足し合わせると求まる。

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、それに 1 加えると求まる。

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

6.1.2 Traversing Binary Trees

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);
}
```

このように再帰を使うと、短くて単純なコードを書けるが、これには問題もある。再帰の深さの最大値は二分木の深さの最大値、すなわち木の高さである。木の高さが非常に大きいと、この再帰は利用できるスタックの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。

再帰なしで二分木を辿るためには、どこから来たかに基づき次の行き先を決めるアルゴリズムを使える。Figure 6.3 を見よ。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 の部分木は辿り終えたので u.parent に戻る。次のコードはこれを実装したものである。ただし、u.left・u.right・u.parent が nil であるケースも適切に処理している。

BinaryTree
```
void traverse2() {
  Node *u = r, *prev = nil, *next;
```

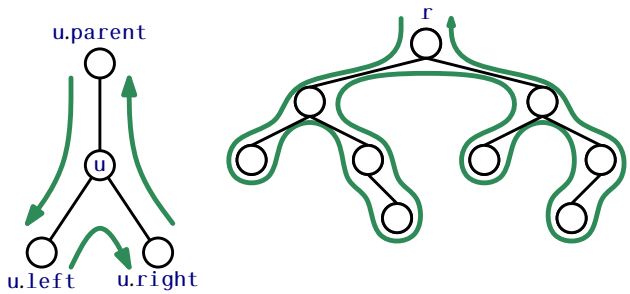


図 6.3: The three cases that occur at node u when traversing a binary tree non-recursively, and the resultant traversal of the tree.

```
  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 を使わないものもある。この場合、再帰でない実装はやはり可能だが、List か Stack を使って今見ているノードから根までの経路を記録する必要がある。

これまで説明したものとは別の辿り方は幅優先なものである。幅優先でノードを辿る場合、根から深さごとに下に向かって、同じ深さのものは左から順に、すべてのノードは訪問される。（Figure 6.4 を参照せよ。）XXX: どういう意味だろうこれは英語の文章読み方と似ている。幅優先の巡回はキュー q を使って実装できる。はじめは q は根だけを含む。ステップごとに次のノード u を q から取り出し、u を処理し、u.left と u.right を（nil じゃなければ）q に加える。

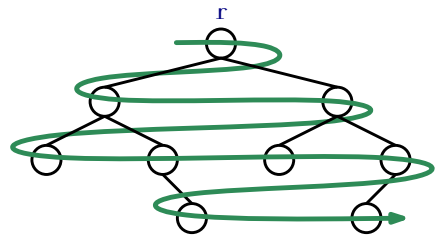


図 6.4: During a breadth-first traversal, the nodes of a binary tree are visited level-by-level, and left-to-right within each level.

BinaryTree

```
void bfTraverse() {
  ArrayDeque<Node*> q;
  if (r != nil) q.add(q.size(),r);
  while (q.size() > 0) {
    Node *u = q.remove(q.size()-1);
    if (u->left != nil) q.add(q.size(),u->left);
    if (u->right != nil) q.add(q.size(),u->right);
  }
}
```

6.2 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。

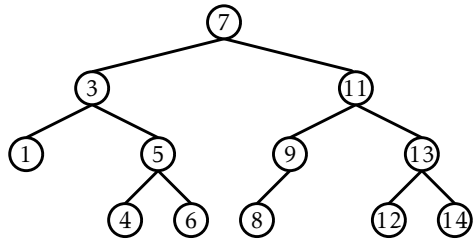


図 6.5: A binary search tree.

6.2.1 探索

二分探索木性は非常に有用である。この性質を利用して二分探索木から値 x の位置を高速に特定できる。具体的には、根 r から x を探し始める。ノード u にいるとき、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 u の値は x である。

この探索は三つ目のケースか、u = nil になると終了する。前者なら x を見つけたことになる。後者なら x が二分探索木には入っていなかったとわかる。

BinarySearchTree

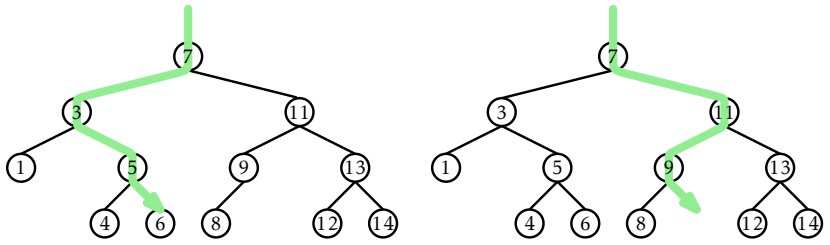
```
T findEQ(T x) {
  Node *w = r;
  while (w != nil) {
    int comp = compare(x, w->x);
    if (comp < 0) {
      w = w->left;
    } else if (comp > 0) {
      w = w->right;
    } else {
```

```
      return w->x;
    }
  }
  return null;
}
```

Two examples of searches in a binary search tree are shown in 二分探索木における探索の例をふたつ Figure 6.6 に示す。二つ目の例として x が見つからない場合にも、役に立つ情報が得られることを示している。探索における最後のノード u で先の場合分けの一つ目のケースだったなら、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だったなら、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発生した最後のノード z を保持しておけば、BinarySearchTree の find(x) 操作は、x 以上の値のうち最小の値を返すように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;
}
```



(a) (b)

図 6.6: An example of (a) a successful search (for 6) and (b) an unsuccessful search (for 10) in a binary search tree.

6.2.2 Addition

BinarySearchTree に値 x を追加するためには最初に x を検索する。もし見つかれば挿入する必要がない。そうでなければ、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。新しいノードが p の右の子か左の子かは x と p.x の比較結果によって決める。

BinarySearchTree

```
bool add(T x) {
  Node *p = findLast(x);
  Node *u = new Node;
  u->x = x;
  return addChild(p, u);
}
```

BinarySearchTree

```
Node* findLast(T x) {
  Node *w = r, *prev = nil;
  while (w != nil) {
    prev = w;
    int comp = compare(x, w->x);
```

```
    if (comp < 0) {
      w = w->left;
    } else if (comp > 0) {
      w = w->right;
    } else {
      return w;
    }
  }
  return prev;
}
```

BinarySearchTree

```
bool addChild(Node *p, Node *u) {
    if (p == nil) {
      r = u;               // 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;
  }
```

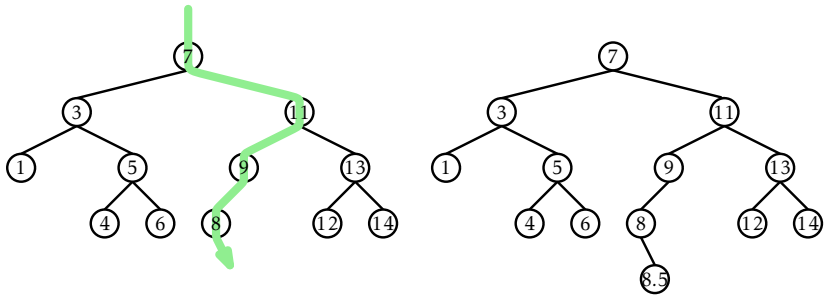


図 6.7: Inserting the value 8.5 into a binary search tree.

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 の高さに比例する。最悪の場合にはこれは BinarySearchTree の高さである。

6.2.3 Removal

BinarySearchTree から値 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すだけでよい。u がひとつだけの子を持つなら u の点を継ぎ合せる、すなわち u.parent と u の子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) {
    r = s;
    p = nil;
  } else {
```

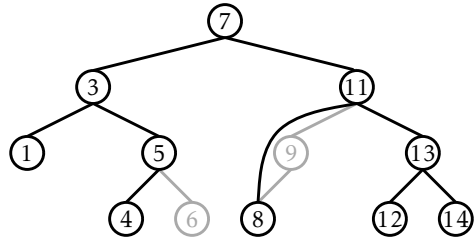


図 6.8: Removing a leaf (6) or a node with only one child (9) is easy.

```
    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 以下で、w.x と u.x を入れ替えても大丈夫なノード w 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 が u.x より大きい中で最小の値であれば十分である。このような w を見つけるのは簡単である。これは u.right を根とする部分木の中で最小の値である。このノードは左の子を持たないため、取り除くのは簡単である。（Figure 6.9 を参照せよ。）

BinarySearchTree

```
void remove(Node *u) {
  if (u->left == nil || u->right == nil) {
    splice(u);
```

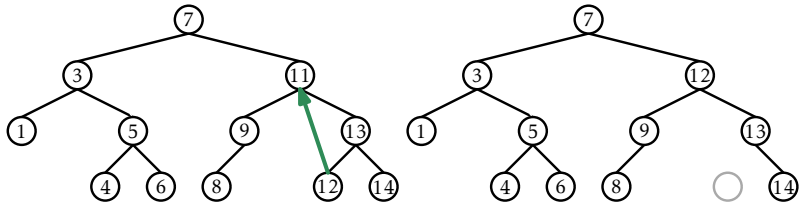


図 6.9: Deleting a value (11) from a node, u, with two children is done by replacing u's value with the smallest value in the right subtree of u.

```
    delete u;
  } else {
    Node *w = u->right;
    while (w->left != nil)
      w = w->left;
    u->x = w->x;
    splice(w);
    delete w;
  }
}
```

6.2.4 要約

BinarySearchTree における find(x)・add(x)・remove(x) 操作はいずれも根からあるノードを見つけることに関連している。木の形状についてなにか仮定しない限り、この探索経路の長さについて「木の中のノード数より短い」ことよりも有用なことを言うのは難しい。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 6.1 は Theorem 4.1 と比べるとショボい。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 のようではなく、ほとんどのノードがひとつの子だけを持ち、n 個のノードの長いチェインみたいな見た目をしている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、いずれも実行時間 $O(\log n)$ での操作を可能にする。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 **??** では最悪実行時間 $O(\log n)$ を、4 つまで子を持ちうる木をシミュレートすることで達成する方法を説明する。

6.3 ディスカッションと練習問題

二分木は血縁関係のモデルとして数千年に渡って使われている。これは家系図は自然と二分木でモデル化されるからだ。家系図では根はある人で、左右の子ノードはその人の両親である。最近の数百年では二分木は生物学における系統樹にも使われている。ここでは葉は現存の種を表し、内部ノードは分化の発生を示す。*speciation events* これは一つの種のふたつの集団からふたつの別々の種が派生することだ。

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。[24, Section 6.2.2] 個々の二分探索木について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。多くの操作が単に根から葉への経路を辿るものならば親へのポインタは不要で、メモリを無駄にしたり、バグを入れ込む原因となったりする。一方親へのポインタがないと、木探索は再帰を（または明示的にスタックを）使うことになる。またいくつか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保持するかである。この章の実装ではそれぞれ別々の変数にこれを保持していた。長さ 3 の配列 p を使う選択肢もある。ここで 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くことができる。

この単純化を木を辿るときに使える。例えば、u.p[i] から u に来たとき、次のノードは 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) であっても使える。これにより、左右の場合それぞれのために複雑なコードを書いていたのをまとめられることがある。例として 156 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 の根の高々ひとつの子を持つか、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 を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・u.right を根とする部分木のサイズの差が高々 1 であることをいう。二分木がこの意味でバランスされているかを判定する再帰的メソッド isBalanced() を書け。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。$O(n)$ より多く時間がかかる実装は簡単である。）

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。

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

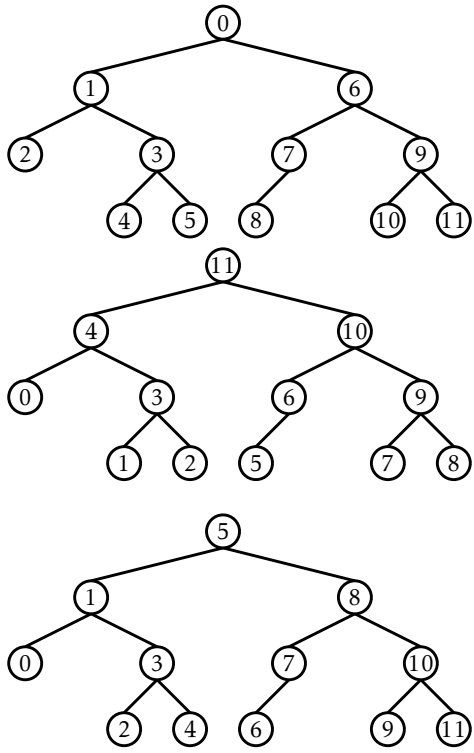


図 6.10: Pre-order, post-order, and in-order numberings of a binary tree.

適切に割り当てる再帰的なメソッド `preOrderNumber()`・`inOrderNumber()`・`postOrderNumbers()` を書け。なお、いずれのメソッドの実行時間も $O(n)$ であること。

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いてノード `u` の次であるノードを返す関数である。いずれも償却実行時間は定数である必要がある。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だ。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、木は常に元の状態に戻るか?

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を定数時間より大きくはし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。

7.1 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。

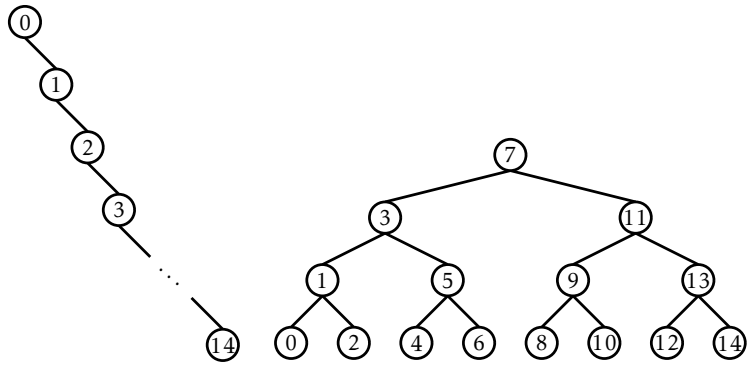


図 7.1: Two binary search trees containing the integers $0, \ldots, 14$.

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,...,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,...,n − 1 の置換からランダムに選出した $x_0,\dots,x_{n-1}$ 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,...,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,...,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。$x \in \{0,\dots,n-1\}$ 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。

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

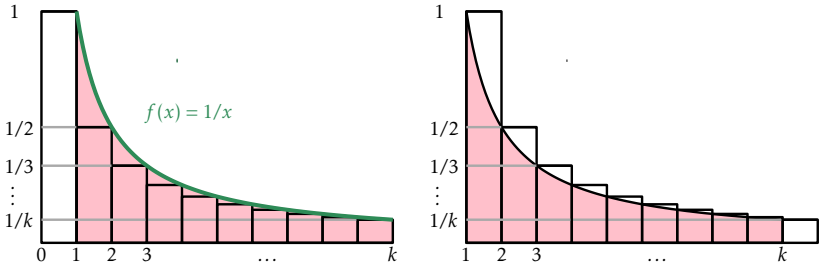


図 7.2: The kth harmonic number $H_k = \sum_{i=1}^{k} 1/i$ is upper- and lower-bounded by two integrals. The value of these integrals is given by the area of the shaded region, while the value of H_k is given by the area of the rectangles.

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。

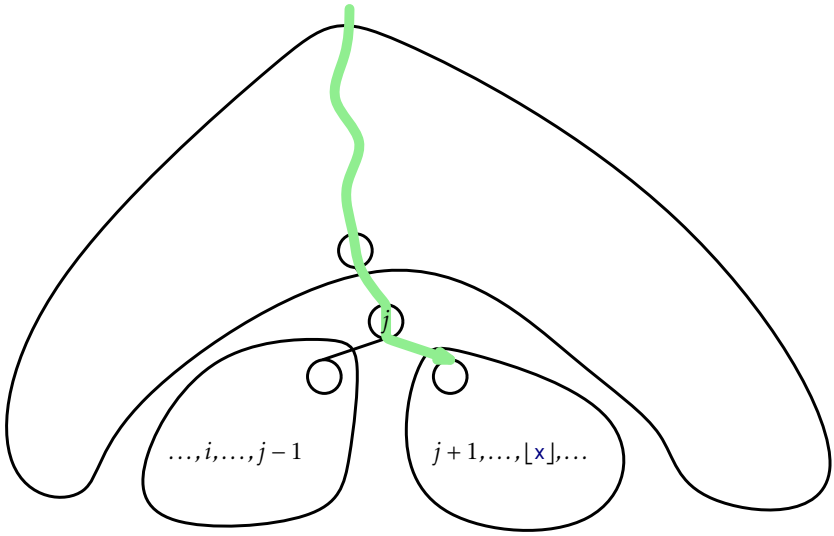


図 7.3: The value $i < x$ is on the search path for x if and only if i is the first element among $\{i, i+1, \ldots, \lfloor x \rfloor\}$ added to the tree.

7.1.1 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 $x \in (-1, n)$ の探索経路に $i < x$ を満たす i をキーとするノードが含まれる必要十分条件は、T を作るランダムな置換において i が $\{i+1, i+2, \ldots, \lfloor x \rfloor\}$ 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 $\{i, i+1, \ldots, \lfloor x \rfloor\}$ のいずれかが追加されるまで探索経路 $(i-1, \lfloor x \rfloor + 1)$ 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 $\{i, i+1, \ldots, \lfloor x \rfloor\}$ の要素とする。j はずっと x の探索経路上にあることに注意する。$j \neq i$ ならば j を含むノード u_j は i を含むノード 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 u_j.`left` を根とする部分木に u_i は追加される。一方 x の探索経路はこの部分木を通らない。なぜならこの経路は u_j を訪問したあと u_j.`right` に向かうからである。

同様に $i > x$ について、i が 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、$\{i, i+1, \ldots, \lfloor x\rfloor\} \cdot \{\lceil x\rceil, \lceil x\rceil+1, \ldots, i-1\}$ 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 $\{i, i+1, \ldots, \lfloor x\rfloor\} \cdot \{\lceil x\rceil, \lceil x\rceil+1, \ldots, i-1\}$ 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。） □

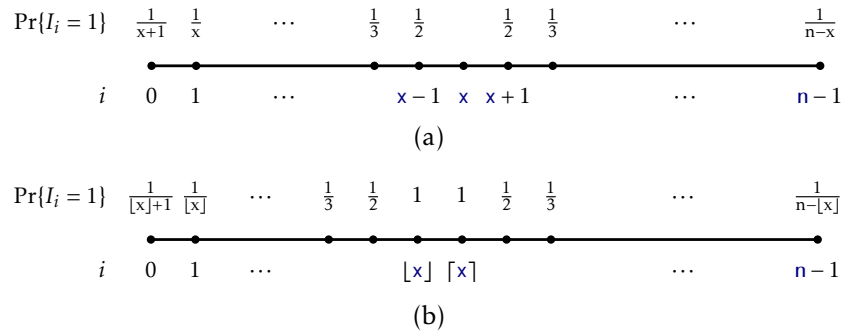


図 7.4: The probabilities of an element being on the search path for x when (a) x is an integer and (b) when x is not an integer.

7.1.2 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。

7.2 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 **??**) でもあることによる。

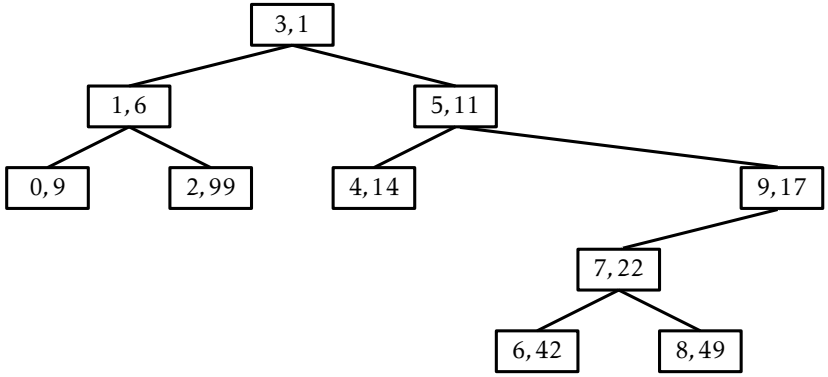


図 7.5: An example of a Treap containing the integers 0,...,9. Each node, u, is illustrated as a box containing u.x, u.p.

られる。

Treap

```
class TreapNode : public BSTNode<Node, T> {
  friend class Treap<Node,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。

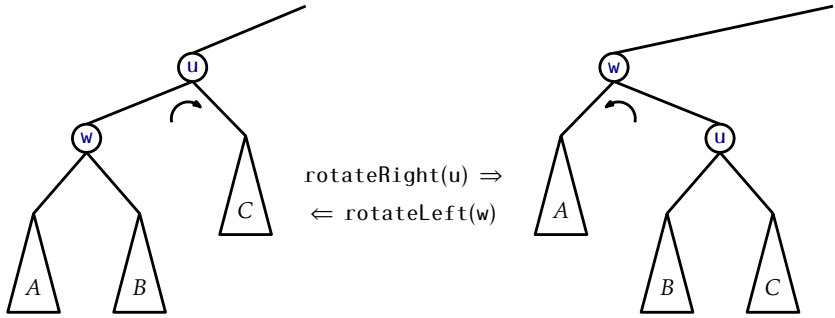


図 7.6: Left and right rotations in a binary search tree.

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

```
bool add(T x) {
  Node *u = new Node;
```

```
  u->x = x;
  u->p = rand();
  if (BinarySearchTree<Node,T>::add(u)) {
    bubbleUp(u);
    return true;
  }
  delete u;
  return false;
}
void bubbleUp(Node *u) {
  while (u->parent != nil && u->parent->p > u->p) {
    if (u->parent->right == u) {
      rotateLeft(u->parent);
    } else {
      rotateRight(u->parent);
    }
  }
  if (u->parent == nil) {
    r = u;
  }
}
```

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。)

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

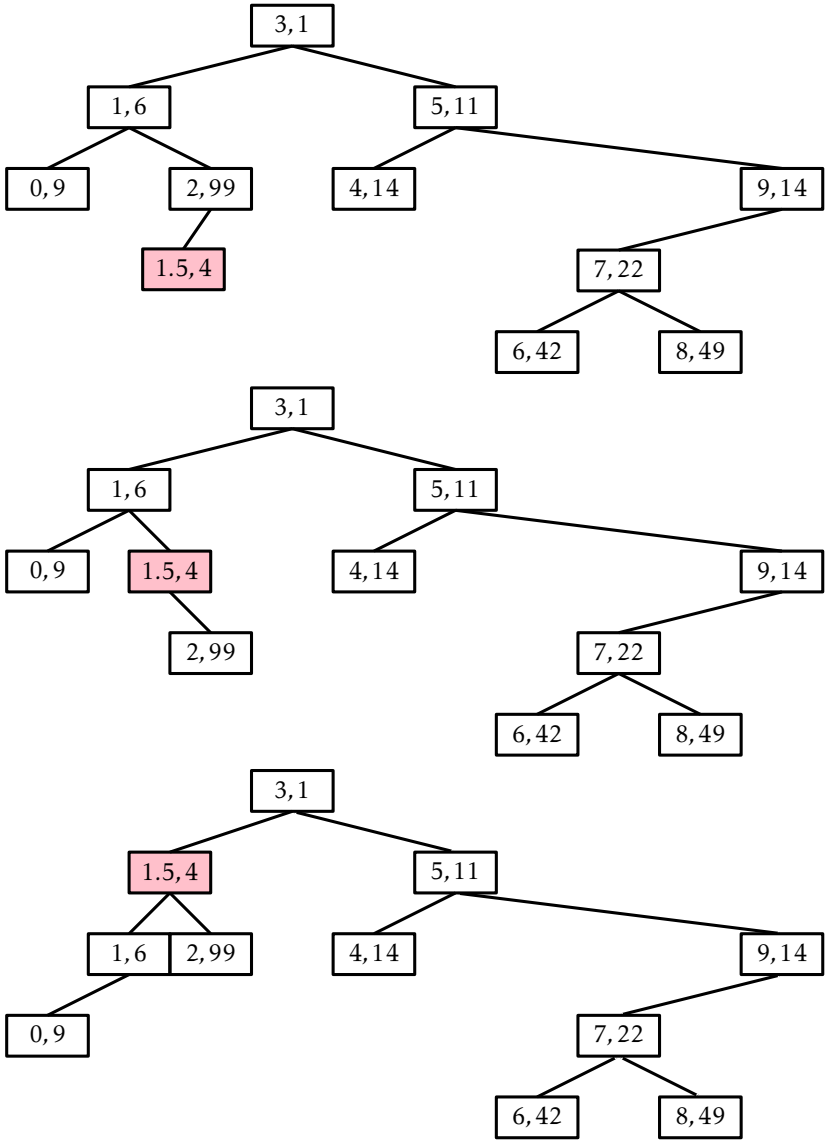


図 7.7: Adding the value 1.5 into the Treap from Figure 7.5.

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

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

```
      r = u->parent;
    }
  }
}
```

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。

7.2.1 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

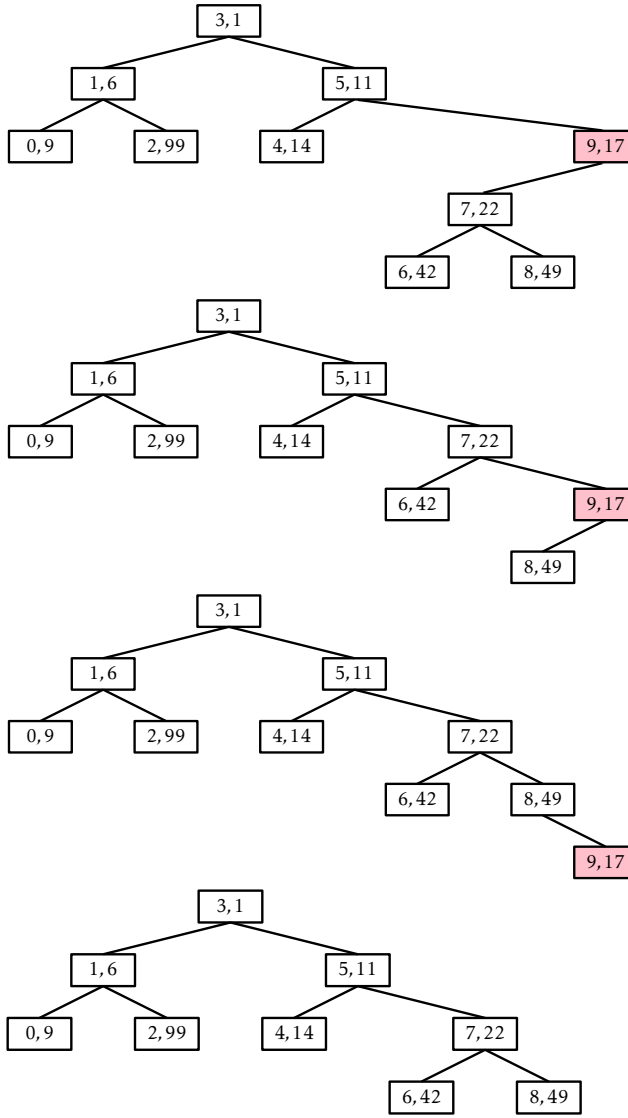


図 7.8: Removing the value 9 from the Treap in Figure 7.5.

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。

7.3 ディスカッションと練習問題

ランダム二分探索木についての研究は多岐に渡る。Random binary search trees have been studied extensively. Devroye[10] 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[35] 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[37] 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[42] 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 [26] 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、$a[0],\ldots,a[m-1]$ は左の部分木に、$a[m+1],\ldots,a[a.length-1]$ は右の部分木にそれぞれ再帰的に格納される。

ScapegoatTree

```
void rebuild(Node *u) {
  int ns = BinaryTree<Node>::size(u);
  Node *p = u->parent;
  Node **a = new Node*[ns];
  packIntoArray(u, a, 0);
  if (p == nil) {
    r = buildBalanced(a, 0, ns);
    r->parent = nil;
  } else if (p->right == u) {
```

```
    p->right = buildBalanced(a, 0, ns);
    p->right->parent = p;
  } else {
    p->left = buildBalanced(a, 0, ns);
    p->left->parent = p;
  }
  delete[] a;
}
int packIntoArray(Node *u, Node **a, int i) {
  if (u == nil) {
    return i;
  }
  i = packIntoArray(u->left, a, i);
  a[i++] = u;
  return packIntoArray(u->right, a, i);
}
```

rebuild(u) の実行時間は $O(\texttt{size}(\texttt{u}))$ 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。

8.1 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。

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

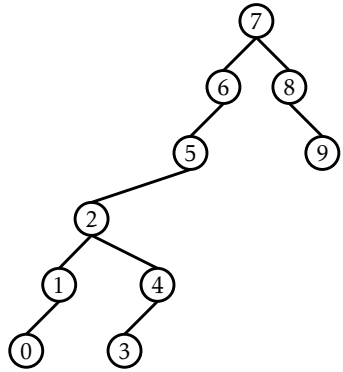


図 8.1: A ScapegoatTree with 10 nodes and height 5.

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。

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

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。

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

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

```
bool add(T x) {
  // first do basic insertion keeping track of depth
  Node *u = new Node;
  u->x = x;
  u->left = u->right = u->parent = nil;
  int d = addWithDepth(u);
  if (d > log32(q)) {
    // depth exceeded, find scapegoat
    Node *w = u->parent;
    int a = BinaryTree<Node>::size(w);
    int b = BinaryTree<Node>::size(w->parent);
    while (3*a <= 2*b) {
      w = w->parent;
      a = BinaryTree<Node>::size(w);
      b = BinaryTree<Node>::size(w->parent);
    }
    rebuild(w->parent);
  } else if (d < 0) {
    delete u;
    return false;
  }
```

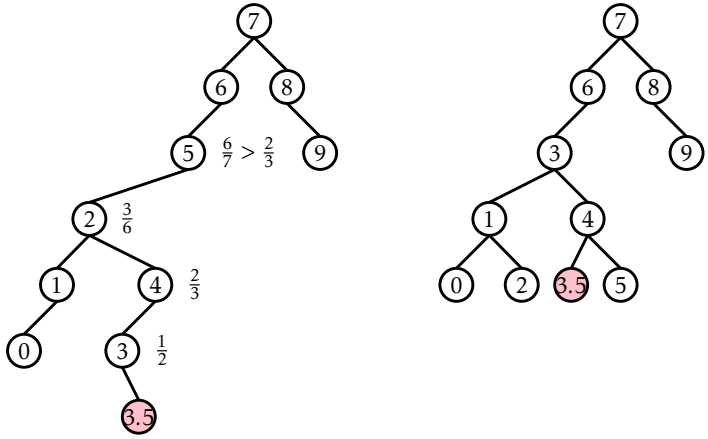


図 8.2: Inserting 3.5 into a ScapegoatTree increases its height to 6, which violates (8.1) since $6 > \log_{3/2} 11 \approx 5.914$. A scapegoat is found at the node containing 5.

```
  return true;
}
```

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

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

ここでも、再構築のコストを無視すれば、remove(x) の実行時間は木の高さに比例し、$O(\log n)$ である。

8.1.1 正しさの証明と実行時間の解析

ここでは ScapegoatTree の各操作の正しさと償却実行時間の解析とを行う。まずは正しさを示すために、add(x) 操作において (8.1) が成り立たなくなったなら常に 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。

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

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。

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

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

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

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

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

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 $O(\texttt{size}(\texttt{w}))$ 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 $O(\texttt{size}(\texttt{w}))$ である。scapegoat を見つけるためには size(u) を $\texttt{u}_k = \texttt{w}$ を見つけるまで $\texttt{u}_0,\ldots,\texttt{u}_k$ に順に実行する。しかし、 $\texttt{u}_k$ はこの列における最初の scapegoat ノードなので、任意の $i \in \{0,\ldots,k-2\}$ について次の式が成り立つ。

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

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。

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

最後の行は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。 □

8.1.2 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。

参考文献

[1] Free eBooks by Project Gutenberg. URL: http://www.gutenberg.org/ [cited 2011-10-12].

[2] 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.

[3] Bibliography on hashing. URL: http://liinwww.ira.uka.de/bibliography/Theory/hash.html [cited 2011-07-20].

[4] 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.

[5] 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.

[6] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick. Resizable arrays in optimal time and space. In Dehne et al. [9], pages 37–48.

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

[8] S. Crosby and D. Wallach. Denial of service via algorithmic complex-

ity attacks. In *Proceedings of the 12th USENIX Security Symposium*, pages 29–44, 2003.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] M. T. Goodrich and J. G. Kloss. Tiered vectors: Efficient dynamic arrays for rank-based sequences. In Dehne et al. [9], pages 205–216.

[18] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*.

Addison-Wesley, 2nd edition, 1994.

[19] 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].

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

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

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

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

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

[25] 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].

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

[27] 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.

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

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

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

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

[32] 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].

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

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

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

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

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

[38] 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.

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

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

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

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