# 情報科学 2011　久野クラス　＃8

久野　靖*

2011.12.2

駒場祭はいかがでしたか。1 週間あいている間にクラスとかすっかり忘れたとかでないとよいのですが… 予告通り、今回で B 課題 (次回までの課題) は最後です。次回からは A 課題 (当日課題) のみとなります。

今回の内容ですが、プログラム中で多様なデータを表現する手段である動的データ構造と、データ構造をパッケージして外部から安全に利用可能にする方法である抽象データ型の考え方について取り上げます。

## 1　前回の演習問題の解説

### 1.1　演習 1 — クラス定義の練習

この演習はメソッドとインスタンス変数が追加できればよいということで、コードだけ掲載しておきましょう:

```
class Dog
  def initialize(name)
    @name = name; @speed = 0.0; @count = 3
  end
  def talk
    puts('my name is ' + @name)
  end
  def addspeed(d)
    @speed = @speed + d
    puts('speed = ' + @speed.to_s)
  end
  def setcount(c)
    @count = c
  end
  def bark
    @count.times do puts('Vow! ') end
  end
end
```

### 1.2　演習 2 — 簡単なクラスを書いてみる

この演習はクラスの書き方の練習みたいなものなので、見ていただけば十分でしょう。Memory2 はちょっと頭を使う必要がありますかね。

```
class Memory
  def initialize()
    @mem = nil
  end
  def put(x)
```

*筑波大学大学院経営システム科学専攻

```
    @mem = x
  end
  def get()
    return @mem
  end
end

class Memory2
  def initialize()
    @mem2 = @mem1 = nil
  end
  def put(x)
    @mem2 = @mem1; @mem1 = x
  end
  def get()
    x = @mem1; @mem1 = @mem2; @mem2 = nil; return x
  end
end

class Concat
  def initialize
    @str = ""
  end
  def add(s)
    @str = @str + s
  end
  def get()
    return @str
  end
  def reset()
    @str = ""
  end
end
```

### 1.3 演習 3 — 有理数クラス

この演習も Relation クラスのメソッドを「同様に」増やせばよいだけなので、難しくはありません。追加するメソッドだけ掲載します:

```
  def -(r)
    return Rational.new(@a*r.get_divisor-r.get_dividend*@b,
                        @b*r.get_divisor)
  end
  def *(r)
    return Rational.new(@a*r.get_dividend, @b*r.get_divisor)
  end
  def /(r)
    return Rational.new(@a*r.get_divisor, @b*r.get_dividend)
  end
```

要は、引き算は足し算と同様、乗算は分母どうし掛け、除算はひっくり返して掛けるということですね。

## 1.4 演習 4 — 複素数クラス

複素数も 2 つの値 (実部、虚部) の組なので、有理数によく似ています。ただし個々の値として整数でなく実数を使います。演算はちょっと面倒 (とくに除算) ですが、作る時に約分とか分母が 0 とか考えなくてよい部分は簡単になります:

```
class Complex
  def initialize(r = 1.0, i = 0.0)
    @re = r; @im = i
  end
  def get_re
    return @re
   end
  def get_im
     return @im
  end
  def to_s
    if @im < 0 then
      return "#{@re}#{@im}i"
    else
      return "#{@re}+#{@im}i"
    end
  end
  def +(r)
    return Complex.new(@re + r.get_re, @im + r.get_im)
  end
  def -(r)
    return Complex.new(@re - r.get_re, @im - r.get_im)
  end
  def *(r)
    return Complex.new(@re*r.get_re - @im*r.get_im,
                       @im*r.get_re + @re*r.get_im)
  end
   def /(r)
    norm = (r.get_re**2 + r.get_im**2).to_f
    return Complex.new((@re*r.get_re + @im*r.get_im) / norm,
                       (@im*r.get_re - @re*r.get_im) / norm)
  end
end
```

to_s でヘンなことをやっているのは、「$a + bi$」の形で表示させようとした時、虚数部が負の場合には「$a - bi$」にしたいためです。

## 1.5 演習 6 — モンテカルロ法の誤差

答えがすぐ分かる例として、関数 $y = x$ の区間 $[0, 1)$ における積分を求めてみましょう。答えは両辺が 1 の直角 2 等辺三角形の面積ですから、0.5 であることはすぐ分かります。プログラムは次のとおり:

```
def integrandom(n)
  count = 0
  n.times do
    x = rand(); y = rand()
```

```
    if y < x then count = count + 1 end
  end
  return count / n.to_f
end
```

では実行させてみます:

```
irb> integrandom 100
=> 0.55              ←誤差 0.05
irb> integrandom 1000
=> 0.475             ←誤差 0.025
irb> integrandom 10000
=> 0.4933            ←誤差 0.0067
irb> integrandom 100000
=> 0.50127           ←誤差 0.00127
irb> integrandom 1000000
=> 0.500674          ←誤差 0.000674
```

試行数が 100 倍になると誤差が $\frac{1}{10}$ になるように見えます。これはなぜでしょうか。

そもそも、なぜこの方法で面積が求まるのかに立ち帰って考えて見ましょう。このプログラムでは 1 回の**試行** (trial — サイコロを振ること) において得られるのは「打った点が関数 $f$ の上か下か」つまり「0 か 1 か」の情報だけです。そして上であれば count は増やさず、つまり 0 を足し、下であれば count を 1 増やし、最後に $N$ で割りますから、この「0 か 1 か」の確率変数の平均を求めているわけです。この確率変数が 1 である確率は関数の面積と等しいので、$N$ を増やしていけば**大数の法則** (law of large number) により、観測される平均値は理論的平均値 (この場合は関数の面積) に近づいていきます。

では、どれくらい近づくのでしょう。それは**中心極限定理** (central limit theorem) が教えてくれます。観測される平均値を $\overline{X}$、真の平均を $\mu$ とすると、

$$\sqrt{N}(\overline{X} - \mu)$$

は $N(0,1)$ つまり 平均 0、分散 1 の正規分布に収束します。言い換えれば、誤差を $\sqrt{N}$ 倍したものが同じ分布になるのですから、試行数を $N$ 倍にすると誤差は $\frac{1}{\sqrt{N}}$ 倍になるわけです。これは確かに上の結果と合致しています。

## 1.6 演習 7 — 格子上の点で調べる

上でやったのと同じ問題を、格子上の点でやってみましょう。簡単のため、縦横の分割数を同じ値とし、この値 $N$ を与えるようにしました:

```
def integgrid(n)
  count = 0; d = 1.0/n
  n.times do |i|
    y = i*d
    n.times do |j|
      if y <= j*d then count = count + 1 end
    end
  end
  return count / (n**2).to_f
end
```

動かしてみた結果は次のとおり:

```
irb> integgrid 10
=> 0.55
irb> integgrid 100
```

```
=> 0.505
irb> integgrid 1000
=> 0.5005
```

じっさいの格子点の数は 2 乗なので 100、10,000、1,000,000 となっていることに注意。しかしずいぶん「規則的」な数字に見えます。

そこで、区間 $(\frac{0}{100}, \frac{1}{100})$、$(\frac{2}{100}, \frac{3}{100})$、…、$(\frac{98}{100}, \frac{99}{100})$ で 1、それ以外で 0 を値とするというちょっといじわるな関数を考えてみましょう (図 1)。[1]

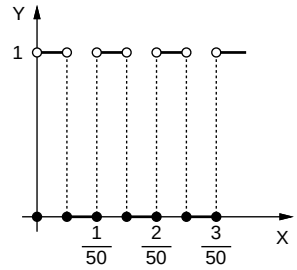


図 1: いじわるな関数

この関数を計算するメソッドと、それを格子点で積分するメソッドを用意しました:

```
def oddfunc(x)
  t = x*100 % 2
  if 0 < t && t < 1 then return 1 end
  return 0
end

def integoddgrid(n)
  count = 0; d = 1.0/n
  n.times do |i|
    n.times do |j|
      if j*d <= oddfunc(i*d) then count = count + 1 end
    end
  end
  return count / (n**2).to_f
end
```

動かしてみると次のとおり:

```
irb> integoddgrid 10
=> 0.28
irb> integoddgrid 100
=> 0.0496
irb> integoddgrid 1000
=> 0.454546
```

ほとんど「めちゃくちゃ」ですね…(正しくは 0.5 になるはずです)。では、モンテカルロ法ではどうでしょうか:

```
def integoddrandom(n)
  count = 0
  n.times do
    x = rand(); y = rand()
```

[1]区間の両端は含まれていないことに注意。

```
      if y <= oddfunc(x) then count = count + 1 end
    end
    return count / n.to_f
  end
```

実行結果は次の通り:

```
irb> integoddrandom 100
=> 0.46
irb> integoddrandom 10000
=> 0.4982
irb> integoddrandom 1000000
=> 0.499454
```

こちらは前と変わらず、試行数を $N$ 倍にすると誤差は $\frac{1}{\sqrt{N}}$ になっています。つまり、格子点だと先のいじわるな関数のように格子のところに段差が当たるなどすると偏った結果が出ておかしくなるわけです。それにそもそも、格子で済むのなら関数の値をもとに普通に台形公式やシンプソンで積分する方がずっと高速なのでした。というわけで、「おかしな」関数でもそれなりに計算できるというモンテカルロ法の利点がお分かりいただけたかと思います。

# 2 動的データ構造/再帰的データ構造

## 2.1 動的データ構造とその特徴

**データ構造** (data structure) とは「プログラムが扱うデータのかたち」を言います。ここでは、プログラムの実行につれて構造を自在に変化させられる、**動的データ構造** (dynamic data structure) について学びます。これに対し、ここまでに扱ってきたプログラムのように、それぞれの変数に決まった形のデータが入っていて、全体の形が変わらないものを**静的データ構造** (static data structure) と呼びます。一般に、**静的** (static) とは「プログラム記述時に決まる」、**動的** (dynamic) とは「プログラム実行時に決まる」という意味があります。

動的データ構造は、プログラム言語が持つ「データのありかを指す」機能を用いて作られます。Ruby では、複合型 (配列、レコード等) や一般のオブジェクトの値は実際は、それらのデータやオブジェクトのありかを指す**参照**になっているので、これを利用します。既に忘れている人がいるかもしれませんが、**レコード**とは複数のフィールドが集まったデータであり、Ruby では `Record.new` においてフィールド名を表す記号を必要なだけ指定して定義するのでしたね。

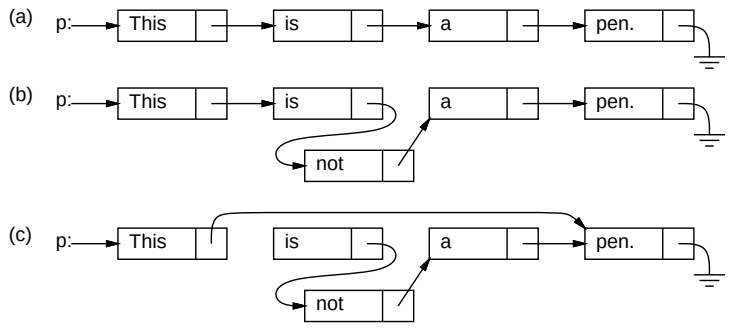


図 2: 単連結リストの動的データ構造

たとえば、次のレコードを見てみましょう:

```
Cell = Record.new(:data, :next)
```

これは 2 つのフィールド `data` と `next` を持つ `Cell` という名前のレコードを定義していますが、ここで各セルのフィールド `next` に「次」のセルへの参照を入れることで、「数珠つなぎ」の動的データ構造を作ることができます (図 2(a))。[2]このような「数珠つなぎ」の構造のことを**単連結リスト** (single linked list) ないし**単リスト**と呼びます。

[2]本当はフィールド `obj` も文字列オブジェクトを参照しているので、文字列を箱の外に描いて矢線で指させるべきなのですが、ごちゃごちゃして見づらくなるのでここでは箱の中に直接描いていています。

この Cell の使い方は、各 Cell の next がまた Cell になっていて、自分の中に自分が入っているように思えます。これは再帰関数と同様で、このようにデータ型 (構造) の中に自分自身と同じデータ型 (構造) への参照を含むものを**再帰的データ構造** (recursive data structure) と呼びます。実際には自分自身が入っているわけではなく図 2 のように「同種のデータへの参照」が入っているだけですから、何ら問題はありません。

一番最後のところ (アース記号で表している) は「何も入っていない」という印である **nil** が入っています。このあたりも、「簡単な場合は自分自身を呼ばずにすぐ値が決まる」再帰関数とちょっと似ていますね。

ところで、動的データ構造だと何がよいのでしょうか? たとえば、図 2(a) で途中に単語「not」を入れたくなったとします。文字列の配列であれば、途中に挿入するためには後ろの要素を 1 個ずつずらして空いた場所に入れる必要があります。しかし、単連結リストでは、矢線 (参照) を (b) のようにつけ替えるだけで挿入ができてしまうのです。逆に、数単語削除したいような場合も、(c) のように参照のつけ換えで行えます。このように、動的データ構造は柔軟な構造の変更が行えるという特徴を持っています。

参照のつけ替えは、具体的にはどうすればいいのでしょう? 参照というのは要するに「場所を示す値」なので、その値をコピーすることは「矢印の根本を別の場所にコピーする (矢印自体も 2 本になる) と考えればいいのです。たとえば、図 3 では、p.next.next というのは B の箱の next フィールド、p.next は A の箱の next フィールドなので、「p.next = p.next.next」で B の箱を迂回して A の箱の next フィールドに C の箱を指させることになります。参照を入れてある単独変数 (この例では p) などでも同様です。

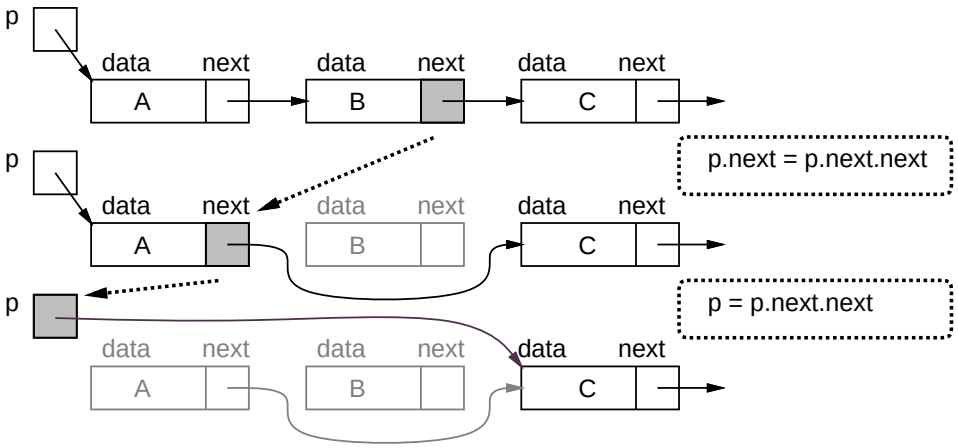


図 3: 参照のつけ替え

ときに、図 2 や 3 で使わなくなった箱はどうなるのでしょう? Ruby では、使われなくなったオブジェクトの領域は**ごみ集め** (garbage collection) と呼ばれる機構によって自動的に回収され、再利用されます。「使っている」かどうかは、どれかの変数からたどれるかどうかで決まります。たとえば図 2 の場合は、先頭のセルを変数 p が指していて、ここからたどれるセルは「使っている」と見なされるのです。

## 2.2 例題: 単連結リストを使ったエディタ

ではここで、単連結リストを使った例題として、簡単な**テキストエディタ** (text editor) を作ってみましょう。「簡単」なので、編集に使うコマンドは次のものしかありません:

- 「i 文字列」 — 文字列を新しい行として現在位置の直前に挿入する。
- 「d」 — 現在位置の行を削除する。
- 「t」 — 先頭行を表示し、そこを現在位置とする。
- 「p」 — 現在位置の内容を表示する。
- 「n」または改行 — 現在位置を次の行へ移しその行を表示する。
- 「q」 — 終了する。

実際にこれを使っている様子を示します (すごく面倒そうですが、実際にこういうプログラムを使ってファイルの編集をしていた時代は実在しました):

```
>iThis is a pen.       ←挿入
>iThis is not a book. ←挿入
>iHow are you?        ←挿入
>t                    ←先頭へ
  This is a pen.
>                     ←次の行
  This is not a book.
>                     ←次の行
  How are you?
>                     ←次の行
  EOF                 ←おしまい
>t                    ←再度先頭へ
  This is a pen.
>iI am a boy.         ←挿入
>                     ←次の行
  This is not a book.
>iWho are you?        ←挿入
>t                    ←再度先頭へ行き全部見る
  I am a boy.
>
  This is a pen.
>
  Who are you?
>
  This is not a book.
>
  How are you?
>
 EOF
>q                    ←おしまい
```

これをこれから実現してみましょう。

## 2.3 エディタバッファ

以下では、単リストのデータ構造を先頭や現在位置などの各変数も含めてクラスとしてパッケージします:

```
class Buffer
  Cell = Struct.new(:data, :next)
  def initialize
    @tail = @cur = Cell.new("EOF", nil)
    @head = @prev = Cell.new("", @cur)
  end
  def atend
    return @cur == @tail
  end
  def top
    @prev = @head; @cur = @head.next
  end
  def forward
    if atend then return end
```

```
      @prev = @cur; @cur = @cur.next
    end
    def insert(s)
      @prev.next = Cell.new(s, @cur); @prev = @prev.next
    end
    def print
      puts("  " + @cur.data)
    end
  end
```

レコード定義もクラスに入れることにしました。また、「1 つの値を複数箇所に代入する」のに=を連続して書いてみました。もちろん、2 つの代入に分けても一向に構いません。

このクラスでは、単リストのセルを上記の Cell レコードであらわし、これを指すための変数として次の 4 つを使っています:

- @head — 一番先頭に「ダミーの」セルを置き、そのセルを常にこの変数で指しておく (ダミーがあると、先頭行を削除するのを特別扱いしないで済ませられるため、プログラムの作成が楽になります)。
- @cur — 「現在行」のセルを指しておく。
- @prev — 「現在行の 1 つ前」のセルを指しておく (挿入や削除の時にこの変数があるとコードを書くのが楽です)。
- @tail — 一番最後にも「ダミーの」セルを置き、そのセルをこの変数で指しておく (表示することがあるので内容は「EOF」(end of file) としてあります)。

initialize では 2 つのダミーセルと上記 4 変数を用意します。head の次が tail であるように Cell.new にパラメタを渡していることにも注意。

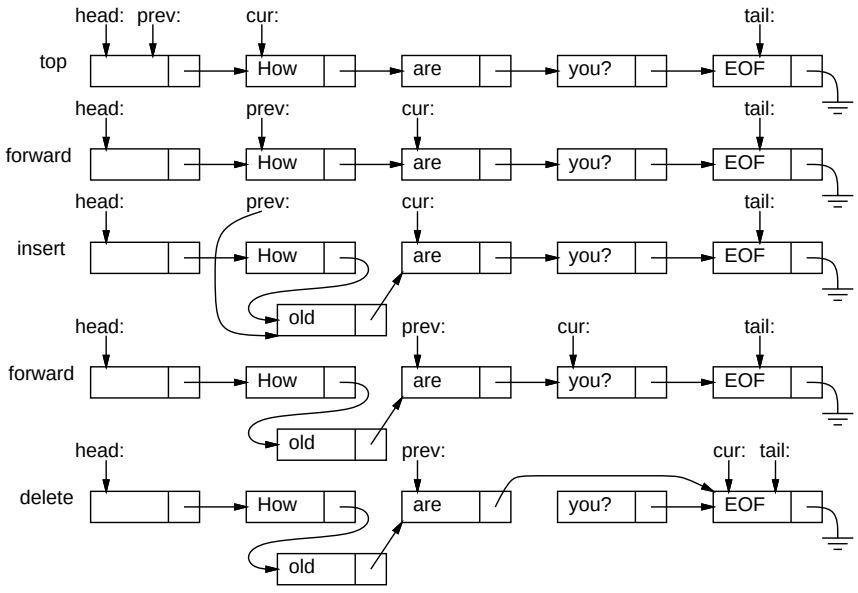


図 4: エディタバッファに対する操作

では次に、メソッドを見てみましょう。図 4 に、適当なバッファの状態で操作を行った例を示します (最後の delete は課題の参考用です)。atend は現在行が末尾にあるか (@tail と等しいか) を調べます。top は@prev と@cur を先頭に設定します。forward は@prev と@cur を 1 つ先に進めますが、現在行が@tail の時は「果て」なので何もしません。print は現在行の文字列を表示します。insert は新しいセルが@prev となり、元の@prev のセルの次が新しいセル、新しい@prev の次が@cur のセルとなります。これを動かした様子を見てみましょう: [3]

```
irb> e = Buffer.new
```

[3] irb の結果表示はうるさいので省略しています。

```
=> ...
irb> e.insert('abc')
=> ...
irb> e.insert('def')
=> ...
irb> e.insert('ghi')
=> ...
irb> e.top
=> nil
irb> e.print
   abc
=> ...
irb> e.forward
=> ...
irb> e.print
   def
=> nil
```

確かに文字列が順序どおり挿入でき、それをたどることができています。

このクラスは「行の挿入や削除が自在にできる機能を持ったオブジェクト」を作り出しています。内部では込み入ったデータ構造を管理していますが、その様子はクラスの外側からは見えません。このように内部構造は**カプセル化**によって隠して、操作を通じて整合性のある汎用的な機能を提供するものを、**抽象データ型** (abstract data type — ADT) と呼びます。

クラス方式のオブジェクト指向言語では、抽象データ型はクラスによって定義するのが自然です。前章に出てきた有理数クラスや複素数クラスも抽象データ型の例だといえます。

**演習 1** 図 5 は「How」という行と「are」という行の間に「old」という行を挿入する様子 (A → B)、および、「old」「are」「you?」という 3 行のうちから「are」を削除する様子 (B → C) を示しています。資料 (ないし同じ図を描き写したもの) の上に赤ペンで次のものを記入しなさい。

a. (A) の図の上に、(A) から (B) につながりが変化するための矢線のつけ替えを、(1)、(2) のようにつけ替えを行う順番つきで記入しなさい。ただし、矢印のつけ替えを行う時には、その出発点がどれかの変数そのものであるか、またはどれかの変数から矢線でたどれる箱であることが必要である。

b. (B) の図の上に、(B) から (C) につながりが変化するための矢線のつけ替えを、(1)、(2) のようにつけ替えを行う順番つきで記入しなさい。ただし、矢印のつけ替えを行う時には、その出発点がどれかの変数そのものであるか、またはどれかの変数から矢線でたどれる箱であることが必要である。

**演習 2** クラス `Buffer` を打ち込み、動作を確認せよ。動いたら、以下の操作 (メソッド) を追加してみよ。

a. 現在行を削除する (EOF 行は削除しないように注意…)

b. 現在行と次の行の順序を交換する (EOF は交換しないように…)

c. 1 つ前の行に戻る (実は大変かも)

d. すべての行の順番を逆順にする (かなり過激)

**演習 3** 単方向リストでは各セルが「次」の要素への参照だけを保持していたが、各セルが「次」と「前」2 つの参照を持つようなリストもある。これを**双連結リスト** (double linked list) ないし**双リスト**と呼ぶ。編集バッファの双リスト版を作り、その得失を検討せよ。[4]

[4] ちなみに、双リストなら単リストでの「頭」と「最後」を 1 つで兼ねることもできます。無理に兼ねなくてもよいですが。

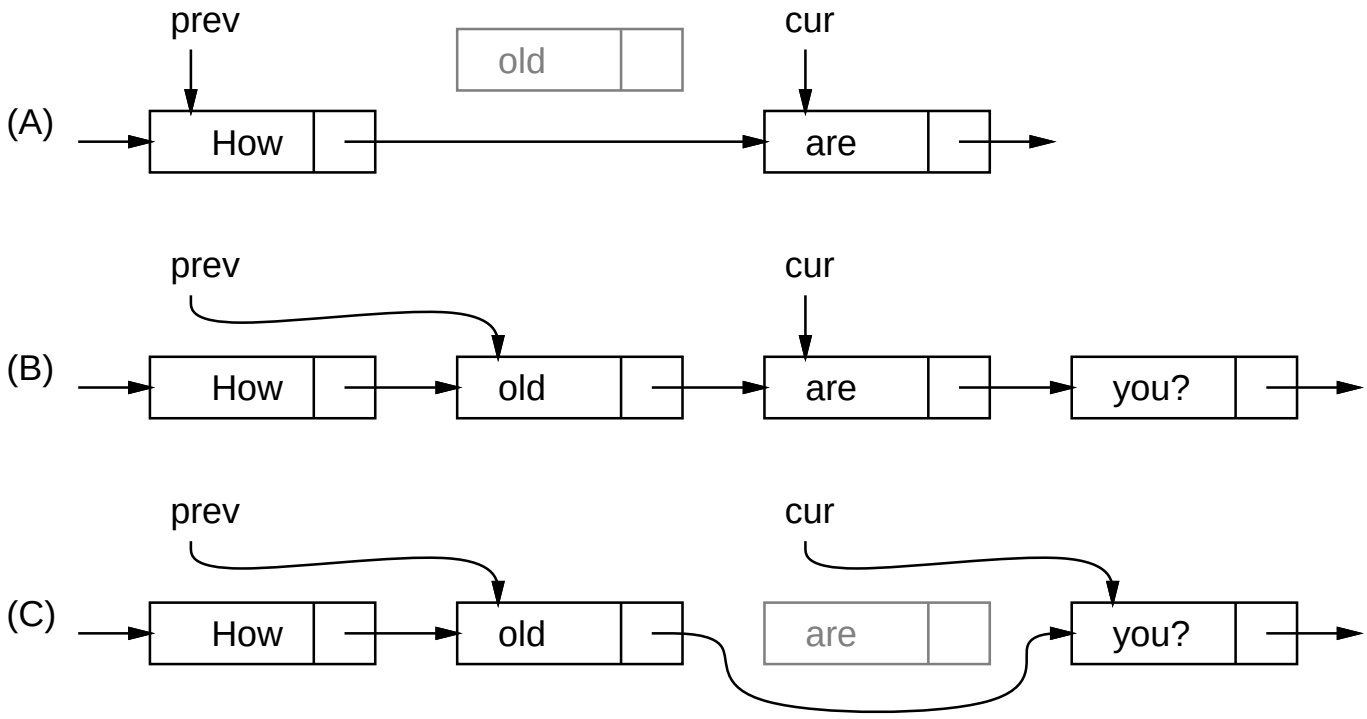


図 5: 挿入と削除のようす

### 2.4 エディタドライバ

バッファのメソッドを呼ぶだけでも編集はできますが、面倒です。先にお見せしたように「コマンド (+パラメタ)」ですらすら編集ができるように、エディタとして動作するコードも作ってみました。内容はとても簡単で、バッファを生成し、その後無限ループでプロンプトを出し、1 行読んでは先頭の 1 文字でどのコマンドを実行するか枝分かれします (コメントにしてあるのはあなたが作るか、後で機能を追加するためのものです):

```
def edit
  e = Buffer.new
  while true do
    printf(">")
    line = gets; c = line[0..0]; s = line[1..-2]
    if c == "q"    then return
    elsif c == "t" then e.top; e.print
    elsif c == "p" then e.print
    elsif c == "i" then e.insert(s)
#   elsif c == "r" then e.read(s)
#   elsif c == "w" then e.save(s)
#   elsif c == "s" then e.subst(s); e.print
#   elsif c == "d" then e.delete
    else                e.forward; e.print
    end
  end
end
```

文字列の一部を取り出すには「[位置.. 位置]」という添字指定を使います。1 文字だけの場合でも文字列として取り出したい場合は位置を 2 つ指定するので、先頭の文字は `line[0..0]` で取り出しているわけです。なお、Ruby 1.9 からは位置 1 つだけでも文字列として取り出せるようになりましたが、それ以前は位置 1 つだけだと「**文字コード** (character code) を表す数値」が返されます。

i(insert) コマンド等では、2 文字目から最後の文字の手前まで (最後の文字は改行文字) も必要なので、これも取り出しています。Ruby では文字列や配列中の位置として負の整数を指定すると末尾からの位置指定になります。

どのコマンドでもない場合 (や改行だけの場合) はいちばんよく使う「1 行進んで表示」にしました。

**演習 4** エディタドライバを打ち込んで先のクラスと組み合わせて動作を確認せよ。動いたら以下のような改良を試みよ (クラス側を併せて改良しても、このメソッドだけを改良しても、どちらでも構いません。文字列を数値にする必要が生じたら、メソッド to_i を使ってください)。

a. 演習 1 で追加した機能が使えるようにコマンドを増やす。
b. 現在行の「次に」新しい行を追加するコマンド「a」を作る (追加した行が新たな現在行になるようにしてください)。
c. 現在行の内容をそっくり打ち直すコマンド「r」を作る。
d. 「g 行数」で指定した行へ行くコマンド「g」を作る。
e. コマンド「p」を「p 行数」でその行数ぶん打ち出すように改良 (その際、できれば現在位置は変更しないほうが望ましいです)。
f. その他、自分が使うのに便利だと思うコマンドを作る。

## 2.5 文字列置換とファイル入出力

せっかくエディタができたのに、行内の置き換えとかファイルの読み書きができないと実用になりませんから、これらを一応解説しておきます。

まず、行内の置き換は「s/$\alpha$/$\beta$/」により現在行中の部分文字列 $\alpha$ を $\beta$ に置き換えるというコマンドにしました。エディタドライバからはバッファのメソッド subst を呼ぶだけとしたので、こちらの中身を示します:

```
def subst(str)
  if atend then return end
  a = str.split('/')
  @cur.data[Regexp.new(a[1])] = a[2]
end
```

文字列のメソッド split は、渡されたパラメタ「/」のところで文字列を分割した配列を返します。その 1 番目を Regexp(パターン) オブジェクトに変換して文字列に添字アクセスすると、そのパターンの箇所があれば、代入によりそこを別の文字列に置き換えられます。

ファイルの読み書きみは、4 章で学んだ open でファイルを開き、読む場合は付属ブロック内でそのファイルの各行を insert、書く場合は逆にバッファの各行をファイルに puts で書き出します:

```
def read(file)
  open(file, "r") do |f|
    f.each do |s| insert(s) end
  end
end
def save(file)
  top
  open(file, "w") do |f|
    while not atend do f.puts(@cur.data); forward end
  end
end
```

**演習 5** 自分が改良したエディタでどれか 1 課題ぶん全部の編集を行い、体験を述べよ。エディタの機能として何があれば必要十分なのか、エディタの使いやすさは何によって決まるかについて考察すること。[5]

[5] なお、エディタのバグによりせっかく作ったプログラムがぐちゃぐちゃになるなどの被害に逢ったとしても、当局は一切関知しませんので、そのつもりでお願いします。

# 3 表と探索

## 3.1 線形探索

**表** (table) とは、**鍵** (key) となる値を指定してデータを登録でき、後で同じ鍵を指定して登録したデータを取り出せるようなデータ構造ないしデータ型を言います。たとえば「学籍番号」を鍵として、氏名や電話番号などをデータとして格納する表などはよく使いそうです。

ここでは鍵が整数、データが文字列であるような表のクラスを `IntStrTable` という名前で作ってみます。そのメソッドとして、次のものが使えるようにしましょう:

- `put(鍵, データ)` — 指定した鍵で指定したデータを表に登録。既にその鍵でデータが登録されていれば前のデータを新しいデータに取り換える。
- `get(鍵)` — 指定した鍵に対応するデータ (文字列) を返す。そのようなデータが登録されていなければ `nil` を返す。

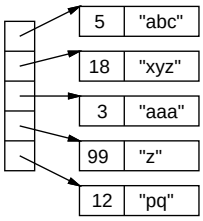


図 6: 素直な表のデータ構造

これを素直に実現するとしたら、鍵とデータの対をレコードとして、そのレコードを並べた配列で表を表す方法がまず思い浮かぶはずです (図 6)。指定した鍵が何番目にあるかはループで調べればよいわけです:

```
class IntStrTable
  Entry = Struct.new(:key, :val)
  def initialize
    @arr = []
  end
  def put(key, val)
    @arr.length.times do |i|
      if @arr[i].key == key then @arr[i].val = val; return end
    end
    @arr.push(Entry.new(key, val))
  end
  def get(key)
    @arr.length.times do |i|
      if @arr[i].key == key then return @arr[i].val end
    end
    return nil
  end
end
```

`put` では順に配列中のレコードを調べていき、鍵が一致するものがあったらそこに新しい値を格納します。最後まで行っても一致するのがなければ、新しいレコードを配列に追加します。`get` でも順に配列中のレコードを調べていき、鍵が一致するものがあれば対応する値を返します。最後まで一致するものがなければ、`nil` を返します。

では、時間計測をしてみましょう。いつもの `bench` を呼び出してて `put` と `get` に分けて計測をするためのメソッドを用意しました:

```
def benchtable(count1, count2, range, tbl)
```

```
    puts(bench(count1) do tbl.put(rand(range), "") end)
    puts(bench(count2) do tbl.get(rand(range)) end)
  end
```

これを使って計測してみましょう:

```
irb> benchtable(1000, 1000, 100000, IntStrTable.new)
0.3984375
0.78125
=> nil
irb> benchtable(2000, 2000, 100000, IntStrTable.new)
1.578125
3.0546875
=> nil
irb> benchtable(3000, 3000, 100000, IntStrTable.new)
3.5625
7.0390625
=> nil
```

表に入るデータの量が2倍、3倍になると、1回当たりの登録/検索時間が2倍、3倍になり、その検索を2倍、3倍の回数繰り返すから全体の時間としては4倍、9倍になります。つまり、1回の登録や検索について見れば、データ量 $N$ について、計算量が $O(N)$ になるわけです。なぜそうかというと当り前で、$N$ 個のデータがあったら、見つかる場合は平均して $\frac{N}{2}$ 回、見つからない場合は常に $N$ 回の比較が必要だからです。

このような、表を「順番に調べていく」方法を**線形探索** (linear search) と呼びます。整列の時の単純選択法などと同様、線形探索はあまり速い方法ではありません。

**演習 6** 線形探索の表のプログラムを打ち込んで動かせ。動いたら、検索速度を改善する何らかの方法を考えて実装し、どれくらい改善されたか調べよ。挿入の速度は改善されなくても、または遅くなってもよいことにする (挿入に比べて検索の回数が多いことを想定)。

改良の方法の1つとして、現在はデータが追加された順に並んでいますが、代わりに鍵の値の順に並ぶようにすることが考えられます。こうすると、2章の区間2分法と同じ原理で、「指定された鍵がある範囲」を半分半分にして探すことができます。これを **2 分探索** (binary search) と言います。

別の方法として、ビンソートのように「直接鍵の添字の場所にデータを入れる」ことも考えられます。これは最強の速さですが、領域が沢山必要です。そこで、鍵の範囲が大きい場合でも使える方法として、その鍵の範囲の値を適当な計算式で「畳み込んで」配列の範囲に収める方法が考えられます。

ただし、畳み込むと複数の鍵が同じ位置に当たることがあるので、その時は後から入れるものは「その次の場所」そこも空いていなければ「そのまた次の場所」のように、よけて入れる必要があります。この場合、「次の場所」を本当の隣にするとデータがふさがった塊ができてしまうので、実際には「いくつ飛び」で次を決める方が優れています。また、この方法では、あまり表が満杯になると効率が悪くなるので、登録できる数に上限を設けるのがよいでしょう。

## 3.2 2 分探索木

**木** (tree) というのは再帰的なデータ構造の一種で、先の単連結リストでは「次」の要素1個を指すことで直線的な数珠つなぎを作っていたのに対し、「子」の要素 $N$ 個を指すことで枝分かれした形を作るものを言います。その枝分かれの点を**節** (node)、一番先端の部分を**葉** (leaf)、一番最初の部分を**根** (root) と言います。さらに、枝分かれの数 $N$ の最大が2のものを **2 分木** (binary tree)、$N$ の場合には $N$ **分木** ($N$-ary tree)、$N > 2$ のものを**多分木**と言います。そして根から一番遠い葉までの枝の数をその木の**段数** (level)、**深さ** (depth)、**高さ** (height) 等と呼びます (図 7)。木が再帰的なデータ構造だということは、ある木の子供もまた木になっていることから分かります。なお、子供の木のことを**部分木** (subtree) と呼びます。葉だけでも木の一種と考えることに注意。

次に、2分木を使ってその各節に鍵とデータを格納することで表を実現する方法の1つである、**2 分探索木** binary search tree について説明しましょう。2分木では、その「子供」は最大2つあります。これらを**左の子** (left subtree)、**右の子** (right subtree) と呼びます。そして、次の性質を持たせるように2分木を構成したものが2分探索木です:

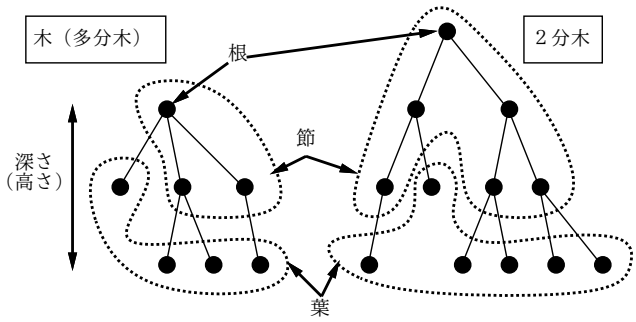


図 7: 木とその用語

- どの節を取っても、その節が持つ鍵より小さい鍵は左の子以下に、またその節が持つ鍵より大きい鍵は右の子以下に、格納されている。

図 8 は 2 分探索木の例です (整数の鍵のみ図示しました)。2 分探索木であれば、ある鍵を探す時、それぞれの節についてそこにある鍵が一致しなかった時「どちら側の子」へ行けばその鍵が見つかるか (または登録されていないと分かるか) 大小比較で決められるので、最大で木の深さだけ比較を繰り返せば済みます。

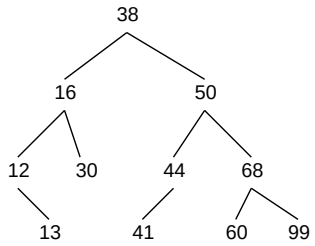


図 8: 2 分探索木の例

では、2 分探索木を使って表を実現してみましょう:

```
class IntStrTable2
  Node = Struct.new(:key, :data, :left, :right)
  def initialize() @root = nil end
  def put(key, val)
    if @root == nil then @root = Node.new(key, val); return end
    node = @root
    while true do
      if key == node.key
        node.data = val; return
      elsif key < node.key
        if node.left == nil then node.left = Node.new(key) end
        node = node.left
      else
        if node.right == nil then node.right = Node.new(key) end
        node = node.right
      end
    end
  end
  def get(key)
    node = @root
    while node != nil do
```

```
      if key == node.key   then return node.data
      elsif key < node.key then node = node.left
      else                      node = node.right
      end
    end
    return nil
  end
end
```

Node が木のノードを表すレコードであり、鍵とデータに加えて、右と左の枝を保持するフィールドを持ちます。@root が木の根元です。

put ではまず、@root が nil なら最初のノードを割り当ててそこに鍵と値を格納して終わりです。それ以外であれば、「現在見ているところ」を変数 node に入れて順次処理しますが、その初期値は@root から始めます。各段階において、鍵が node.key と一致したら、そのノードに値を格納して終わります。そうでない場合、鍵と node.key の大小に応じて右か左の枝に進みますが、進もうとして nil だった場合はそこに指定の鍵を持つ新しいノードを割り当てて入れてから進むので、次の周回でそのノードに値を格納することになります。

get でも node に「現在見ているところ」が入るのは同じですが、いきなり@root を入れて始まります。そして node が nil ならループを終わってメソッド末尾で nil を返します。それ以外なら指定の鍵が node.key に等しいならそのノードの値を返し、そうでなければ鍵と node.key の大小に応じて右か左の枝に進みます。nil だった時に新しいノードを割り当てる処理が要らないので、get の方が短く簡潔になるわけです。

では、こちらも時間を計測してみましょう:

```
irb> benchtable(1000, 1000, 100000, IntStrTable2.new)
0.03125
0.015625
=> nil
irb> benchtable(2000, 2000, 100000, IntStrTable2.new)
0.0703125
0.0546875
=> nil
irb> benchtable(3000, 3000, 100000, IntStrTable2.new)
0.09375
0.078125
=> nil
```

線形探索よりも圧倒的に速いことが分かります。

**演習 7** この例題を打ち込んで動かし、時間計算量の分析/検討を行いなさい。

**演習 8** 実は、2 分探索木は、格納する際に鍵の値が昇順または降順に並んでいると不利である (クイックソートみたいですね)。この現象を確認し、理由を検討しなさい (これを改良できるとさらにすばらしい)。

## 3.3 連想配列 (Hash)

さてここまで引っ張って今更ですが、実は Ruby には言語組み込みの表機能のクラス Hash が備わっています。そしてそれは、配列と似たような見た目を持っていることから、**連想配列** (associative array) とも呼ばれます。

Perl、JavaScript、Python など多くの言語に連想配列の機能が備わっています。ハッシュというのは Ruby や Perl での呼び名ですが、それはこの機能の実現に**ハッシュ表** (hash table) のアルゴリズムが使われていることによります。ハッシュ表のアルゴリズムについては、次回に解説します。

Ruby の場合については、ハッシュの使い方は次のとおりです:

- h = Hash.new(値) — 連想配列を作る。「値」は表を検索して見つからなかった場合に返される値を指定 (省略した場合は nil になる)。変数名 h は説明用で、実際には何でもいい。

- `h[鍵] = 値` — 連想配列に鍵を指定して値を格納。
- `h[鍵]` — 連想配列から鍵を指定して値を取り出す。

まさに機能としては表と同じですね。さらに、配列式 (いわゆる配列リテラル)`[1, 2, 3]` などと同様、次のように直接ハッシュの値を指定することもできます (これも通称は**ハッシュリテラル** (hash literal) ですが、正式には**ハッシュ式** (hash expression) と呼ばれています)。

```
h = {1 => "abc", 8 => "xyz", 3 => "a"}
h = {"abc" => 3, "def" => "u"}
```

このように、`{...}`の内側に「鍵 `=>` 値」という形のものを 0 個以上、カンマで区切って並べて指定します。なお、この例でも分かるように、鍵の値は整数でなくても実数、文字列など任意の値が指定できます

**演習 9** クラス Hash に対して登録/検索処理の時間計算量を計測し分析せよ。

**演習 10** 動的データ構造を活用した、何か面白いプログラムを作れ。面白さの定義は各自に任されます。

## A　本日の課題 8A

「演習 2」〜「演習 4」で動かしたプログラムを含む小レポートを今日中に久野までメールで送ってください。

1. Subject: は「Report 8A」とする。
2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、投稿日時を書く。
3. プログラムどれか 1 つのソース。
4. 以下のアンケートの回答。

   Q1. 動的データ構造とはどのようなものか理解しましたか。

   Q2. 「行の並びを扱う部分 (バッファ)」とそれを使う部分を分離するという考え方に納得しましたか。

   Q3. 本日の全体的な感想と今後の要望をお書きください。

## B　次回までの課題 8B

次回 (といいつつ次々回) までの課題は、「演習 2」〜「演習 10」の (小) 課題から 2 つ以上選んで報告することです (8A で提出したものは除く)。

各課題のために作成したプログラムは複数ある場合もすべてレポートに掲載してください。レポートは「12/9(金) の」授業開始時刻の **10 分前**までに久野までメールで送付してください。

1. Subject: は「Report 8B」とする。
2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、投稿日時を書く。
3. 1 つ目のプログラムのソース。
4. その説明と分析/考察。
5. 2 つ目のプログラムのソース。
6. その説明と分析/考察。
7. 下記のアンケートの回答。

   Q1. 何らかの動的データ構造が扱えるようになりましたか。

   Q2. 表の機能と「代表的な」実現方法を理解しましたか。

   Q3. 課題に対する感想と今後の要望をお書きください。