情報科学 2007　久野クラス　＃8

久野　靖*

2006.12.7

はじめに

前回はオブジェクト指向入門でクラスとかインスタンスとかやりましたが、あまり抵抗はないようでよろしかったです。これからはクラスを作ることが多くなりますので…今回は新しい内容として次のものをやります。

- 動的データ構造/再帰的データ構造
- 表と探索

その前に演習問題解説がありますが、常微分方程式の分は皆様問題なさそうですし省略します。

1　演習問題解説

1.1　演習 3

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こう。

```
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
```

*筑波大学大学院経営システム科学専攻

1.2 演習 4

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ない。追加するメソッドだけ掲載する。

```
  def -(r)
    return Rational.new(@a*r.getDivisor-r.getDividend*@b, @b*r.getDivisor)
  end
  def *(r)
    return Rational.new(@a*r.getDividend, @b*r.getDivisor)
  end
  def /(r)
    return Rational.new(@a*r.getDivisor, @b*r.getDividend)
  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。

1.3 演習 7

復素数の演算も 2 つの値 (実部、虚部) を保持するので有理数によく似ている (ただし整数でなく実数を使う)。演算はちょっと面倒 (とくに除算) だが、作るときに約分とか分母が 0 とか考えなくていい部分は簡単になる。

```
class Complex
  def initialize(r = 1.0, i = 0.0)
    @re = r; @im = i
  end
  def getRe
    return @re
   end
  def getIm
     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.getRe, @im + r.getIm)
  end
  def -(r)
    return Complex.new(@re - r.getRe, @im - r.getIm)
  end
  def *(r)
    return Complex.new(@re*r.getRe - @im*r.getIm, @im*r.getRe + @re*r.getIm)
  end
  def /(r)
    norm = (r.getRe**2 + r.getIm**2).to_f
    return Complex.new((@re*r.getRe + @im*r.getIm) / norm,
                       (@im*r.getRe - @re*r.getIm) / 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とき、虚数部が負の場合には「$a-bi$」にしたいため。

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配置のしかた」を言う。これまでに学んで来たプログラムでは、いくつかの変数 (その中には配列やレコードやレコードの配列を保持する場合もある) の集まりがデータ構造となってきた。この場合、配列の大きさなどは毎回変わることもあるが、データが持つ構造は基本的にいつも「同じ形」をしている→**静的** (static)。

これに対して、プログラムの実行につれてデータが持つ構造が柔軟に変化していくようなものを、**動的データ構造** (dynamic data structure) と呼ぶ。これはどのようにして実現できるかというと、プログラム言語が持つ「あるデータのありかを指す」機能を活用する。Ruby では、オブジェクトの値 (配列、レコードを含む) はそのオブジェクトのありかを指す**参照** になっている。これを利用して、動的データ構造を作ることができる。

たとえば、次のようなレコードを見てみる:

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

既に忘れている人がいるかと思うが、レコードとは複数のフィールドが集まったデータで、Ruby では Record.new においてフィールド名を表すシンボルを必要なだけ指定することで定義できるのでしたね。

さて上の例は 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るわけだが、各セルにおいて next というフィールドに他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る (図 1(a))。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ぶ。なお、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書いて矢線で指させるべきなのだが、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書いている。

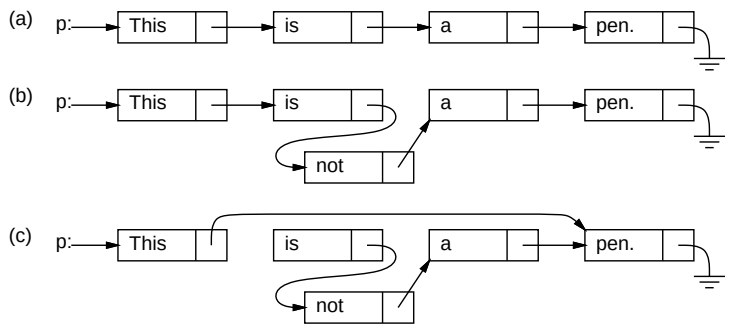

図 1: 単連結リストの動的データ構造

なお、この Cell の使い方を見ると、各 Cell の next がまた Cell になっているので、自分の中に自分が入っているような気がする。これは再帰関数と同様であり、このようにあるデータ型 (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) と呼ぶ。

実際には自分自身が入っているわけではなく図 1 のように「同種のデータへの参照」が入っているだけだから問題はない。また、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 が入っている (このあたりも「単純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る)。

ところで、動的データ構造だと何がいいのだろうか? たとえば、図 1(a) で途中に単語「not」を入れたくなったとする。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る。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。逆に、数単語削除したいような場合でも (c) のように参照のつけ替えで行える。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つ。

ところで、図 1(c) で使わなくなった「is」「not」「a」の箱はどうなるのだろう? Ruby では、使われなくなったオブジェクト (やレコード) の領域は自動的に回収して再利用される。この回収機構のことを**ごみ集め** (garbage collection) と呼ぶ。「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める。図 1 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。

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エディタ (Emacs のようなもの) を作ってみよう。ただし「おもちゃ」なので、ファイルからの読み込みやファイルへの保存はできず、編集に使うコマンドも次のものしかない。

- 「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は次のようになる。

```
>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
```

あほみたいだが、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というのは実在した。それはさておき、これをこれから実現してみよう。

2.3 エディタバッファ

まず、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する。レコード定義もクラスに入れることにしよう。

```
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
```

このクラスでは、単連結リストのセルを上記の Cell レコードあらわし、これを指すための変数として次の 4 つを使っている。

- @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。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、ダミーがある方が楽。
- @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。挿入や削除のときにこの変数があると楽。
- @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) としておいた。

このため、initialize のでは 2 つのダミーセルを用意し、上記 4 つの変数を初期化する (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)。

では次に、メソッドを見てみよう。図 2 に、適当なバッファの状態で操作を行った例を示す (最後の delete は課題の参考用)。atend は現在行が末尾にあるか (@tail と等しいか) を調べる。top は@prev と@cur を先頭に設定する。forward は@prev と@cur を 1 つ先に進めるが、現在行が@tail のときは何もしない。print は現在行の文字列を表示する。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 のセルとなる。

では、これを動かした様子を見ていただこう (irb の結果表示はうるさいので省略している)。

```
irb(main):011:0> e = Buffer.new
=> ...
```

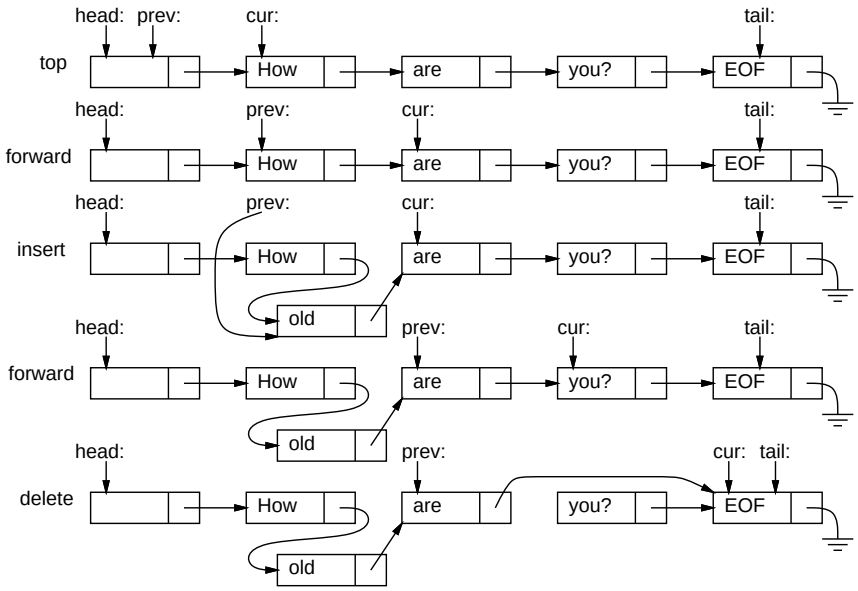

図 2: エディタバッファに対する操作

```
irb(main):012:0> e.insert('abc')
=> ...
irb(main):013:0> e.insert('def')
=> ...
irb(main):014:0> e.insert('ghi')
=> ...
irb(main):015:0> e.top
=> nil
irb(main):016:0> e.print
   abc
=> ...
irb(main):017:0> e.forward
=> ...
irb(main):018:0> e.print
   def
=> nil
```

確かに、バッファに文字列が順序通り挿入されていて、それをたどることができるようだ。

演習 1 クラス Buffer に以下の機能 (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 2 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」への参照を持つようなリスト (**双方向リスト**) もある。編集バッファの双方向リスト版を作り、その得失を検討せよ。(ちなみに、双方向リストなら単方向リストでの「頭」と「最後」を 1 つで兼ねることもできる。兼ねなくてもいいけど。)

2.4 エディタドライバ

確かにバッファのメソッドを呼ぶだけでも編集はできるが、ちょっと大変すぎる。先に説明したように「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メソッドを作ってみた (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)。

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

内容は非常に簡単で、エディタバッファを生成し、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する。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えばよい。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だ。あと、`i`(insert) コマンドなどでは、2 文字目から最後の文字の手間 (最後の文字は改行文字なので) までを取り出している (位置としてマイナスを指定すると末尾からの位置指定になる)。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」にした。

演習 2 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よい。

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」新しい行を追加し、その行が新たな現在行になるコマンド「a」を作る。
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」を改良し、「p 行数」でその行数ぶん打ち出すようにする (現在位置はできれば変更しない方が望ましい)。
f. その他、自分が使うのに便利だと思うコマンドを作る。

なお、`line` の 2 文字以降に数値が入ってるのを取り出すのは「`line[1..-2].to_i`」ですね。

2.5 おまけ: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使えないので、これらを一応解説しておく。本来学ぶ内容でない「おまけ」なので簡単に。もっときちんと学びたい場合は Rubu 本などをあさってください。

まず、行内の置き換は「`s/`α`/`β`/`」により現在行中の部分文字列 α を β に置き換えるというコマンド。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 で渡されたパラメタを「/」のところで分割して配列に分ける。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る。

ファイルからの読み込みは、「open」でファイルを開き、付属ブロック内でそのファイルの各行について insert するだけ。

```
def read(file)
  open(file, "r") do |f|
    f.each do |s| insert(s) end
  end
end
```

ファイルへの書き出しは画像のときにやったが、「open」でファイルを開き、付属ブロック内で順次ファイルに puts する。

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

演習 3 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。エディタの機能と使いやすさについて考察すること。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、当局は一切関知しない。

3 表と探索

3.1 線形探索

表 (table) とは、ここでは**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いう。たとえば「学籍番号」が鍵で「氏名」がデータであるような表を使ったりしそうですよね?

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る。そのメソッドとして次のものが使えるようにする。

- 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替える。
- 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。

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付く (図 3)。指定した鍵が何番目にあるかはループで調べればよい。この方針で作ったサンプルを示そう。

```
class IntStrTable
  Entry = Struct.new(:key, :val)
  def initialize
```

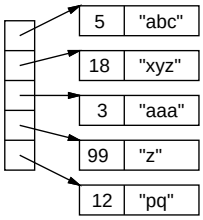

図 3: 素直な表のデータ構造

```
    @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行き、鍵が一致するものがあったらそこに新しい値を格納する。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する。get でも順に配列中のレコードを調べて行き、鍵が一致するものがあれば対応する値を返す。最後まで一致するものがなければ nil を返す。

では、時間計測をしてみよう。いつもの bench と、それを使って put と get に分けて計測をするためのメソッドを用意した。

```
def bench(count, &block)
  t1 = Process.times.utime
  count.times do yield end
  t2 = Process.times.utime
  puts t2-t1
end
def benchtable(count1, count2, range, tbl)
  bench(count1) do tbl.put(rand(range), "") end
  bench(count2) do tbl.get(rand(range)) end
end
```

これを使って計測してみた。

```
irb(main):017:0> benchtable(1000, 1000, 100000, IntStrTable.new)
0.3984375
0.78125
=> nil
irb(main):018:0> benchtable(2000, 2000, 100000, IntStrTable.new)
1.578125
3.0546875
```

```
=> nil
irb(main):019:0> 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倍になる。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 $O(N)$ になるわけだ。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。

このような、表を「順番に調べて行く」方法を**線形探索** (linear search) と呼ぶ。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ない。

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)。

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るが、代わりに鍵の値の順に並ぶようにすることが考えられる。こうすると、求根の区間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る (これを **2 分探索**という)。

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てしまう」ことも考えられる。これは最強の速さだが、メモリが沢山要るのでしたね。鍵の範囲がずっと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」配列の範囲に入れることが考えられる。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、そのとき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る (さらに、「その次」を本当に隣にすると塊でふさがってしまうので、「いくつ飛び」で次を決める方がよりよい)。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吉。

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う。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) という。枝分かれの数 N の最大が2のものを **2 分木** (binary tree)、それ以上のものを多分木という。そして根から一番遠い葉までの枝の数をその木の**段数**、**深さ**、**高さ**などと呼ぶ (図 4)。木が再帰的なデータ構造だというのは、ある木の子供もまた木 (**部分木**、subtree) になっていることから分かる (葉だけでも木の一種と考える)。

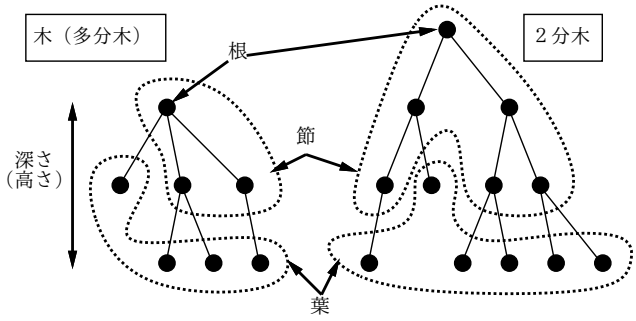

図 4: 木とその用語

さて、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について考えてみる。このとき、2 分木なのでその「子供」は最大2つある。これを「左の子」「右の子」と呼ぶ。そして、次の性質を持たせるように2分木を構成する (このような 2 分木を **2 分探索木**と呼ぶ):

- 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、格納されているようになっている。

図 5 は 2 分探索木の例である (整数の鍵のみ図示した)。2 分探索木であれば、ある鍵を探すとき、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探せば済む。

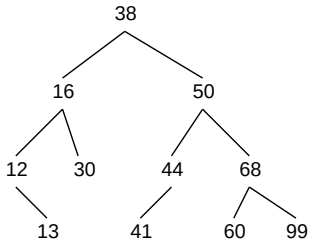

図 5: 2 分探索木の例

では、先の「配列のレコード」の代わりに 2 分探索木を使って表を実現してみる。

```
class IntStrTable2
  Node = Struct.new(:key, :data, :left, :right)
  def initialize
    @root = nil
  end
  def put(key, val)
    @root = put1(@root, key, val)
  end
  def put1(node, key, val)
    if node == nil
      return Node.new(key, val, nil, nil)
    elsif key == node.key
      node.data = val; return node
    elsif key < node.key
      node.left = put1(node.left, key, val); return node
    else
      node.right = put1(node.right, key, val); return node
    end
  end
  def get(key)
    return get1(@root, key)
  end
  def get1(node, key)
    if node == nil
      return nil
    elsif key == node.key
      return node.data
    elsif key < node.key
      return get1(node.left, key)
    else
      return get1(node.right, key)
    end
```

```
    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 (サブツリー) を保持するフィールドを持つ。@root が木の根元で、最初は nil。put は単に@root と鍵と値をパラメタとして下請けメソッド put1 を呼び、それが返した値を@root に入れ直しているだけ。

put1 はサブツリー (の根元ノード) と鍵、値を引数として受け取り、そのサブツリーに値を格納する。もしサブツリーが空 (nil) なら、新しいノードを割り当てて鍵と値を格納し、それを新しいサブツリーとして返す。これ以外の場合はすべて、引数として渡されて来たノードをそのまま返す。つまり木構造が変化するのは末端の nil になっているところに新しいノードが追加される形だけということ。さて残りの場合だが、まず現在のノードの鍵が一致すれば現在のノードに格納し、そうでない場合は鍵の大小関係に応じて右または左のサブツリーに格納する。これは自分自身への再帰呼び出しで行い、返されてきた値を右または左の子として格納し直す。

get はやはりサブツリーを受け取って動作する下請けメソッド get1 を呼ぶだけ。get1 はそのツリーが空なら「見つからなかった」ので nil を返し、鍵が一致すれば現在のノードの値を返し、それ以外は鍵の大小関係に応じて左右いずれかのサブツリーに対して自分を再帰的に呼び出して検索する。

では、こちらも時間を計測してみよう。

```
irb(main):020:0> benchtable(1000, 1000, 100000, IntStrTable2.new)
0.03125
0.015625
=> nil
irb(main):021:0> benchtable(2000, 2000, 100000, IntStrTable2.new)
0.0703125
0.0546875
=> nil
irb(main):022:0> benchtable(3000, 3000, 100000, IntStrTable2.new)
0.09375
0.078125
=> nil
```

圧倒的に速いことがわかる。

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、格納する鍵の値が昇順または降順に並んでいると計算量が悪くなる (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 (というかそれを実現するクラス Hash) が備わっている。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、一般には**連想配列**と呼ばれる (Perl、JavaScript、Python など多くの言語にこの機能が備わっている)。Ruby の場合についてその使い方は次のとおり。

- 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。

こういうわけで、まさに機能としては表と同じですね。さらに、配列の「[1, 2, 3]」などと同様、次のように「直接」値を指定することもできる。

```
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すればよい。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る。

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 1」〜「演習 3」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、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

次回までの課題は「演習 1」〜「演習 9」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、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。