第9章　再帰データ構造

前章で見たレコードやオブジェクトを使うと、小さなデータを集めて大きなデータを作り、それを集めてさらに複雑なデータを作るということができる。このようなデータの組み立て方は、小さな部品を組み合わせてエンジンや翼を作り、それらを組み合わせて自動車や飛行機のような複雑な機械を作る過程に似ている。

この章では、**再帰データ構造**と呼ばれる種類のデータの作り方とその扱い方を紹介する。再帰データ構造は、自分自身と同種のデータを要素として組み立てられる。自動車や飛行機の部品に例えて考えると奇異にも思えるが、再帰をデータの構造に用いることによって、後から何個でも値を追加できるような柔軟なデータの扱いができるようになる。

以下では、代表的な再帰データ構造として**リスト構造**と木構造の 2 つを紹介する。

9.1　リスト構造

9.1.1　例題: カラオケ演奏機の予約リスト

カラオケ演奏機の予約機能を作ることを考えてみよう。1 つ 1 つの予約は演奏したい曲を表わすデータであり、それを順に並べた「予約リスト」を管理することになるだろう。カラオケ演奏機の使い方を考えてみると、予約リストには次のようなことが言えるだろう。

- 予約は気まぐれに追加されるので、全く予約がない場合もあれば沢山予約が並ぶこともある。つまり、予約リストの長さは予想できない。
- 予約は先着順に演奏される。そのため、次の曲を演奏する際にはリストの先頭から予約を取り出す。新しい予約はリストの最後に追加する。
- 予約をキャンセルすることがある。つまり、リストの途中にある予約を削除することがある。
- いまある予約リストを無視して次に唄う曲を指定したいことがある。つまり、予約をリストの先頭に追加することもある。

この予約リストは予約が 1 列に並んだデータなので配列によって表わせばよいではないかと思うかも知れない。しかし配列は後から大きさを変えることができないため、最大の長さが予想できないようなリストの管理には適していない。配列として用意した大きさを超える数の予約がなされようとした場合は、予約を一時的に禁止するか、より大きな配列を作りそこに全ての予約を書き写すことをしなければいけない。前者は使う人にとって不便であるし、後者はプログラムを作る上での手間が多い。また、リストの途中にある予約を削除するような場合は、その後ろに続く予約を 1 つずつ前にずらすか、予約にキャンセル済という印を付けておくことになる。どちらの場合でも、プログラムを作る上での手間が増えてしまう。

実は Ruby の配列は後から大きさを変えることができる。これは Ruby 言語が、最初に用意したよりも大きな配列が必要になったときに、内部で自動的により大きな配列を用意し直しているためである。

9.1.2　予約をつないだリスト

それでは、個数に制限がなく、追加・挿入・削除の手間も小さい予約リストの作り方を見てゆこう。考え方は、

1 つの予約情報は「曲に関する情報」と「次の予約情報」を持つ

というものである。このような予約情報をレコード (クラス) として定義したものがファイル 9.1 である。なお、説明を容易にするために、予約情報は曲名 (`title`) を持つこととした。もう 1 つのインスタンス変数 `next` が、次の予約情報を表わすが、最初は空 (`nil`) にしている。

`nil` という値は、`Array.new(n)` という式によって配列を作ったときに登場している (p. 38)。

```
1 class Request
2   attr_accessor("title", "next")
3
4   def initialize(t)
5     self.title = t
6     self.next = nil
7   end
8 end
```

ファイル 9.1: `request.rb` (予約情報の定義)

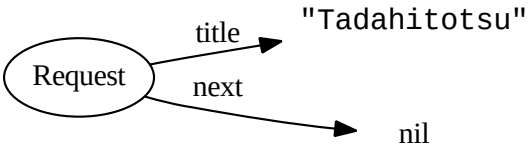
この `Request` クラスを使って予約リストを作ってみよう。以下では、オブジェクトの関係を図示する `view` という関数を使いながら進めてゆく。

まず「ただ一つ」という曲の予約を表わすオブジェクトを作り、変数 `r0` にしまおう。

> 注意:　以下は、配布プログラム `view.rb` をダウンロードして、`irb` 中に読み込んだ (`load`) 後に実行すること。

```
1 irb(main):007:0> r0 = Request.new("Tadahitotsu")
2 => #<Request:0x404545e4 @title="Tadahitotsu", @next=
     nil>
3 irb(main):008:0> view(r0)
4 (結果省略)
```

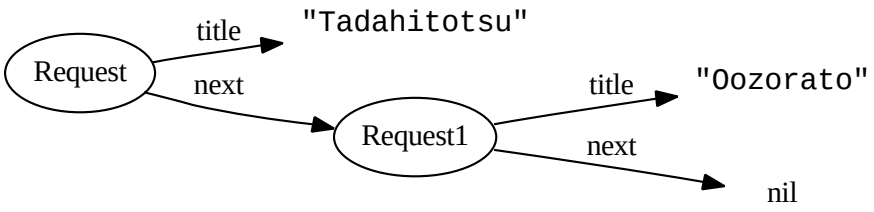
3 行目を実行すると下のような図が表示される。この図は r0 に入っているオブジェクトの様子を表わしているもので、1 つの丸が 1 つのオブジェクトに対応する。丸の中の Request はそのオブジェクト のクラスを表わす。矢印はインスタンス変数を表わしており、例えば title には"Tadahitotsu"という値がしまわれていることを示している。



次に「大空と」という曲の予約を作り、それを「ただ一つ」の「次の予約情報」としてつなげてみよう。

```
1 irb(main):010:0> r1 = Request.new("Oozorato")
2 => #<Request:0x4044b3e0 @title="Oozorato", @next=nil
     >
3 irb(main):011:0> r0.next = r1
4 => #<Request:0x4044b3e0 @title="Oozorato", @next=nil
     >
5 irb(main):012:0> view(r0)
6 (結果省略)
```

再び view を実行すると今度は 2 つのオブジェクトの関係が表示される。



このようにして最後の予約情報の「次」に新たな予約情報を加えてゆくことを続ければ、何個でも予約情報を並べてゆくことができる。配列が、最初に

作ったときの大きさ以上のデータを持つことができないのに対して、**データの個数に制限がない**のが特徴である。

練習 9.1 (予約リストの作成) 配布プログラム ut_songs.rb にある関数 ut_songs() は図 9.1 のような予約リストを作り、先頭の予約情報を返す。irb 中で list = ut_songs() ↵ view(list) と入力し、同じ図が表示されることを確めよ。(注意: 図では “Request2” のようにクラス名の後ろに番号が付く。これはオブジェクトを区別するために関数 view が適当につけたものなので、毎回同じ番号が付くとは限らない。)

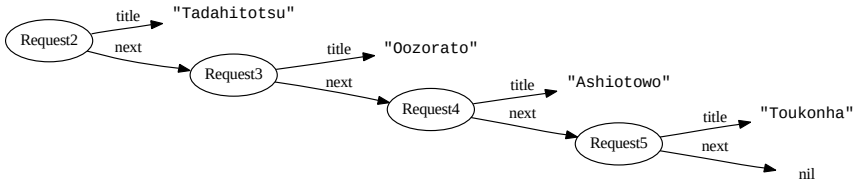


図 9.1: 予約リストの例

9.1.3　再帰データ構造とリスト構造

上で定義した予約情報は**再帰データ構造**の 1 つである。どこに再帰が用いられているのかを、前章で定義した図形を表わすレコードやオブジェクトとの対比しながら考えてみよう。

まず、あるクラスのオブジェクトのインスタンス変数入っている値 (オブジェクト) の種類 (クラス) が何であるかを整理してみよう。前章の図形データを表わすクラスでは次のようになっていた。

> Point の x, y には数値が入る
> Line の p0, p1 には Point が入る
> Bezier の p0, c, p1 には Point が入る
> Circle の p には Point が、r には数値が入る

同様に考えると予約情報は次のようになる。

> Request の title には文字列が、next には Request が入る

このように、データの種類 (クラス) を、内部に入るデータの種類 (クラス) によって考えたものを**データ構造**と言う。図形データのデータ構造は、どんどん分解してゆくとやがて数値や文字列などの基本的なデータの種類に行きつく。

一方、予約情報 Request は、インスタンス変数に Request が入っている。つまり、予約情報のデータ構造は予約情報自身によって表わされており、このこと指して予約情報は**再帰データ構造**だと言う。

予約情報のように、「次の情報」を 1 つだけ持つような再帰データ構造を一般に**リスト構造**という。リスト構造は個数に制限がなく、伸び縮みが簡単にできるので、順に処理すべき仕事や、一般の文章のように長さが予め分からないデータを管理するような場面で多く用いられている。

9.1.4　予約の取り出しと追加

練習 9.1 で行ったように、list に図 9.1 のような予約リストが用意されているとする。この list から予約を取り出したり、予約を追加する方法を見ておこう。

先頭の予約を取り出す

カラオケ演奏機が新しい曲に移るときは、予約リストから先頭の予約情報を 1 つ取り出して曲を演奏する。変数 list には先頭の予約情報そのものが入っているので、これは list に入っている予約情報を使うことにほかならない。仮に play(name) という関数で曲が演奏されるとした場合、以下のような命令を実行すればよい。(なおここでは実際に演奏は行われず、画面にメッセージが表示されるだけである。)

```
1 irb(main):017:0> play(list.title)
2 It starts playing a song titled ‘Tadahitotsu.’
3 => nil
```

先頭の曲を演奏したら、予約リストからその曲を取り除かなければいけない。そのためには変数 list を、これまで先頭だった予約の「次の予約情報」に変更すればよいので、以下のような命令を実行する。

```
1 irb(main):019:0> list = list.next
2 (結果省略)
```

この命令の実行前の変数 list が図 9.1 の「ただ一つ」だったとき、実行後の list は「大空と」になる。

先頭への割り込み

予約を追加する方法の 1 つとして、リストの先頭に割り込む方法を見てみよう。カラオケで予約を割り込ませることは滅多にないかも知れないが、リ

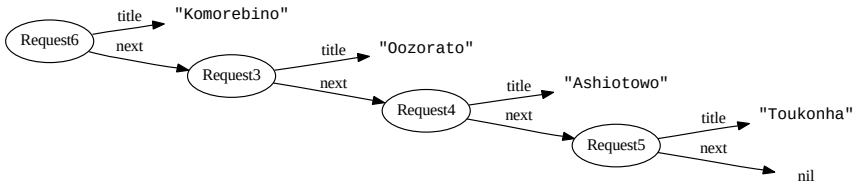
ストの最後に予約を追加するよりもずっと簡単なので、あえて割り込み方を先に紹介する。

例えば「こもれびの (なかに)」という曲を予約リストの先頭に割り込ませたい場合は、次のような手順になる。

1. 新しく予約情報 r2 を作り、
2. r2 の「次の予約情報」を、いままでの予約リストの先頭にし、
3. 予約リストの先頭を r2 で置き換える

```
1 irb(main):021:0> r2 = Request.new("Komorebino")
2 (結果省略)
3 irb(main):022:0> r2.next = list
4 (結果省略)
5 irb(main):023:0> list = r2
6 (結果省略)
7 irb(main):024:0> view(list)
8 (結果省略)
```

予約リストの様子は下図のようになる。



末尾への追加

予約リストの最後に曲を追加するためには、(1) まず最後の予約を見つけ、(2) その予約の「次の予約情報」として新しい予約情報を追加することになる。

そこでまず最後の予約を見つける関数を定義しよう。最後の予約とは「次の予約情報」が無い予約にほかならない。逆に、ある予約 r に「次の予約情報」がある場合には、r の「次の予約情報」から最後の予約を見つければよい。これを再帰関数によって定義すると、ファイル 9.2 の last_request(r) のようになる。

予約リスト r の末尾に予約情報 s を追加するには、last_request(r) によって見つけた最後の予約の「次の予約情報」に新しい予約情報 s を代入すればよいので、ファイル 9.2 の add_request(r,s) のようになる。

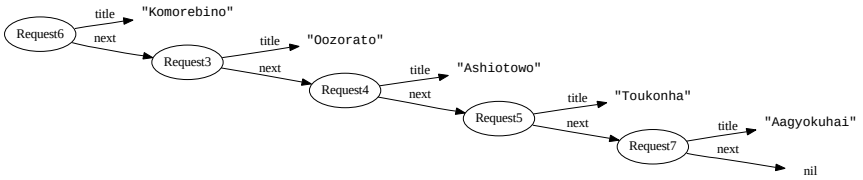
例として「嗚呼玉杯」という唄を add_request を用いて予約リストの末尾に追加してみよう。

```
9  def last_request(r)
10   if r.next == nil
11     r
12   else
13     last_request(r.next)
14   end
15 end
16
17 def add_request(r,s)
18   last = last_request(r)
19   last.next = s
20 end
```

ファイル 9.2: request.rb (末尾への追加)

```
1 irb(main):026:0> add_request(list,
2 irb(main):027:1*  Request.new("Aagyokuhai"))
3 (結果省略)
4 irb(main):028:0> view(list)
5 (結果省略)
```

図を見ると確かに最後に追加されていることが分かるだろう。



(章末の練習問題: 9.5)

9.1.5　予約情報の削除

一度入れた予約をキャンセルすることを考えてみよう。その予約がリストの先頭にあるときは、予約の取り出しと同じことである。リストの最後にあるときは、その 1 つ手前の「次の予約情報」を空にすればよい。リストの途中にある場合は、1 つ手前の予約情報と 1 つ後の予約情報をつなぎ直す必要がある。いずれにせよ、削除する場合には 1 つ手前の予約情報を変更する必要があることに注意しよう。

このような削除は、再帰関数によって簡潔に定義できる。いま、r を先頭とする予約リストから曲名が t であるような予約情報を 1 つ削除された予約リストを答えるような関数を delete_request(r,t) とする。この関数は r の曲名が t と一致するかどうかによって場合分けされ、次のような操作を行えばよい。

- 一致する場合、r 自身を削除することになるので、r.next が削除済みのリストになる
- 一致しない場合、r の次から delete_request を用いて削除する。このとき、r の直後の予約情報が削除される場合もあるので、再帰的に呼び出した delete_request の結果を r の「次の予約情報」に置き換える。削除済みのリストは r 自身となる。

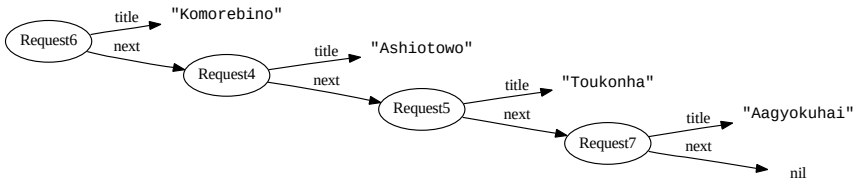
少し複雑に思えるかも知れないが、関数の定義はファイル 9.3 に示すようにそれほどでもない。しかも、この定義は予約リストの途中だけでなく先頭や末尾の予約情報を削除する場合にも対応している。 例えば予約リストから

```
21 def delete_request(r,t)
22   if r.title == t
23     r.next
24   else
25     r.next = delete_request(r.next, t)
26     r
27   end
28 end
```

ファイル 9.3: request.rb (予約情報の削除)

「大空と」を削除するには、次のようにする。この例では delete_request の結果を変数 list に代入し直している。これは、先頭の予約情報が削除される場合のためである。

```
1 irb(main):030:0> list=delete_request(list,
2 irb(main):031:1*  "Oozorato")
3 (結果省略)
4 irb(main):032:0> view(list)
5 (結果省略)
```



(章末の練習問題: 9.6 9.7 9.8 9.9 9.10 9.14)

9.2　木構造と再帰

再帰データ構造のもう 1 つの例として、今度は**木構造**を紹介する。ここでは電話帳の管理を例にとって紹介してゆくが、まずはデータ構造の形を先に見ておこう。

図 9.2 は木構造として表わした電話帳の例である。1 つ 1 つの連絡先 (Contact) が名前 (name) と電話番号 (number) の対応を表わしている。この点はカラオケ予約リストと同じようなものである。

連絡先どうしのつながり方に注目してほしい。1 つの連絡先に 2 つの連絡先がつながっている。ここがリスト構造と違う重要な点である。つまりリスト構造は、1 列にデータが並んでいたのに対し、この木構造は 1 つの連絡先から左右 (left, right) に連絡先が広がる形になっている。

このように、ある種類 (クラス) データが、同じ種類 (クラス) のデータ 2 つあるいはそれ以上に再帰的につながる構造のことを**木構造**という。特にこの電話帳のように 2 つにつながる場合は**二分木**とも言う。これがなぜ木構造と呼ばれるかと言うと、図 9.2 を反時計回りに 90 度回転させてみるとデータの広がりが樹木のように見えるためである。そのため木構造の説明をする場合には、次のような言葉を使うことがある。

根: 出発点となるデータを「根 (root)」と言う。

葉: 後ろ (電話帳では left, right) に他のデータがつながって**いない**データを「葉 (leaf)」と言う。

節: 逆に、後ろにデータがつながっているデータを「節 (node)」と言う。

枝: データどうしをつなぐ線を「枝」と言う。

親子: データ A の後ろ (電話帳では left や right) にデータ B がつながっているとき、A を「親」、B を「子」と言う。(ここは木と関係ない言葉が使われる。) また「親」と「親の親」と「親の親の親」……をまとめて「先祖」、「子」と「子の子」と「子の子の子」……をまとめて「子孫」と言うこともある。

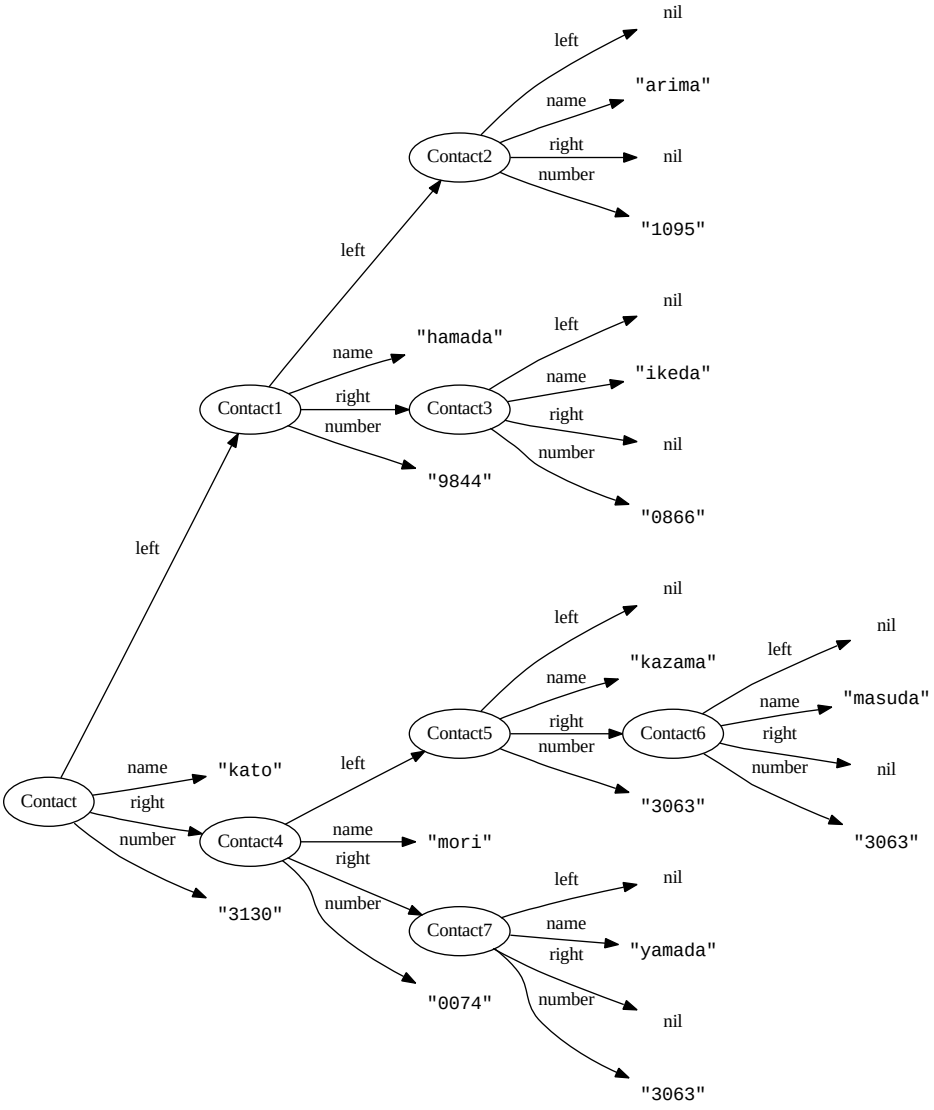


図 9.2: 木構造として表わされた電話帳

練習 9.2 (木構造の用語) 図 9.2 に関して答えよ。

a) 根である連絡先の名前

b) 葉である全ての連絡先の名前

c) 節である全ての連絡先の名前

d) `hamada` の親にあたる連絡先の名前

e) `hamada` の子にあたる全ての連絡先の名前

9.2.1　例題: 電話帳

先に木構造の形を見てしまったが、これに対してどのような計算や操作を行うかを考えてみよう。電話帳を例に考えると、次の 3 つが代表的なものだろう。

検索: 与えられた名前を持つ連絡先情報を探す。

追加: 与えられた連絡先を追加する。

削除: 与えられた名前を持つ連絡先を削除する。

特に検索が多く用いられるだろうから、連絡先は名前順に整列しておき、高速に探せるようにしておくのがよい。

木構造を上手く用いると、大量の連絡先があるような場合でも上の 3 つの計算や操作を高速に行うことができる。以降ではその方法を見てゆく。

他のデータ構造を使う場合と比べると、検索・追加・削除の全てが高速に行えるのが木構造の特徴である。配列を整列して使う場合には、名前順に整列検索を高速に行うことはできても、追加や削除に時間がかかってしまう。リスト構造の場合には検索・追加・削除すべてに時間がかかってしまう。これが、一見複雑そうに見える木構造を用いる理由である。

9.2.2　整列された木構造

検索・追加・削除が高速にできる秘密は、木構造という形だけでなく、そこに配置した連絡先が名前順に並んでいるためである。図 9.2 の根の連絡先 (`kato`) は、左 (left) の子 (`hamada`) よりアルファベット順で後に来る名前を持ち、右 (right) の子 (`mori`) より前に来る。同様に `hamada` とその左右の子の間にも、左の子は前に来る名前、右の子は後に来る名前になっている。

つまり、この木構造の各連絡先は、

> 左の子孫は全て、自身より前に来る名前を持ち、
> 右の子孫は全て、自身より後に来る名前を持つ

ように配置されている。

この性質を使った連絡先の検索は次のようになる。例えば `ikeda` という名前で探す場合は、

1. 根の名前 `kato` よりも前なので左 (left) へ進み、
2. 次の `hamada` よりも後なので右 (right) へ進み、
3. 名前が一致する連絡先を見つける

正確には、1 文字目でアルファベット順に並べ、1 文字目が同じものは 2 文字目でアルファベット順に並べ、2 文字目も同じものは 3 文字目で……という方法で並べた順序のことである。辞書の見出しを並べる際の順序であることから**辞書式順序**という。Ruby では文字列どうしを `<` や `>` で比較すると、辞書式順序に従った結果になる。

という手順になる。見つけるべき連絡先がどこにあっても、後戻りをすることなく連絡先までたどりつけるだろう。

それでは、このような木構造と、それに対する計算や操作がどのように定義になるかを見てゆこう。

9.2.3　連絡先データの定義

1 つの連絡先を表わすデータは、ファイル 9.4 のように定義できる。図 9.2 で見ているように、名前 (name)、電話番号 (number) と左右につながった連絡先 (left, right) の 4 つのインスタンス変数がある。

```
1 class Contact
2   attr_accessor("name","number","left","right")
3   def initialize(name,number)
4     self.name = name
5     self.number = number
6     self.left = nil
7     self.right = nil
8   end
9 end
```

ファイル 9.4: contact.rb

練習 9.3 (電話帳の作成) 配布プログラム phonebook.rb には図 9.2 と同じデータを作るような関数phonebook() が定義されている。view(phonebook()) を実行して図 9.2 と同じ図が表示されることを確めよ。

9.2.4　電話帳に対する再帰的な操作

名前による検索

連絡先 p を根とする電話帳から n という名前を持つ連絡先を探すことを考えよう。これを関数 $f(p,n)$ と書くことにする。p の名前が n と一致する場合は p が解になる。そうでない場合、p の名前と n の前後関係によって左右どちらかの子孫に名前があることが分かるので、$f(p,n)$ は次のような再帰関数として定義できる。

$$f(p,n) = \begin{cases} p & (p \text{ の名前と } n \text{ が一致}) \\ f(p \text{ の左の子}, n) & (p \text{ の名前は } n \text{ より後}) \\ f(p \text{ の右の子}, n) & (p \text{ の名前は } n \text{ より前}) \end{cases}$$

この関係は、そのまま Ruby の再帰関数として定義でき、ファイル 9.5 のようになる。

```
10 def find_contact(p,n)
11   if p.name == n
12     p
13   else
14     if n < p.name
15       find_contact(p.left, n)
16     else
17       find_contact(p.right, n)
18     end
19   end
20 end
```

ファイル 9.5: contact.rb (続き: 名前による検索)

```
1 irb(main):007:0> root = phonebook()
2 (結果省略)
3 irb(main):008:0> find_contact(root, "masuda")
4 => #<Contact:0x4045190c @number="3063", @name="
     masuda", @right=nil, @left=nil>
```

練習 9.4 (最初の連絡先) 連絡先 p を根とする電話帳で、名前順で一番最初の連絡先を見つける関数 first_contact(p) をファイル first_contact.rb に定義せよ。(ヒント: 一番最初とは p から左 (left) へ進み続けた端にある連絡先のことである。)

連絡先の追加

電話帳に新しい連絡先を追加するときは、正しい場所を見つけることが必要となる。つまり、追加後も全ての連絡先が、左の子はそれより前の、右の子はそれより後の名前を持つようになっていなければいけない。例えば名前が hasumi である連絡先を図 9.2 に追加する場合、ikeda の左の子の位置でなければいけない。

そのような正しい場所は、「名前による検索」の場合と同様にして見つけられる。追加される連絡先の名前と現在注目している連絡先の名前の前後関係から、左右どちらかの子をたどってゆき、空 (nil) になっている場所があれば、そこが正しい場所となる。

カラオケ予約リストの末尾への追加 (第 9.1.4 節, p.155) と同じように、連絡先の追加を行う関数を再帰的に定義してみよう。ファイル 9.6 にある関数 add_contact(p,q) は、連絡先 p を根とする電話帳に連絡先 q を追加し、**追加された連絡先を返す**。

```
21 def add_contact(p,q)
22   if p == nil
23     q
24   else
25     if q.name < p.name
26       p.left = add_contact(p.left,q)
27     else
28       p.right = add_contact(p.right, q)
29     end
30     p
31   end
32 end
```

ファイル 9.6: contact.rb (続き: 連絡先の追加)

この関数は 3 通りに場合分けされる。まず、p が空だった場合は、そこに q を追加すべきなので、q を返せばよい (23 行目)。そうでない場合は、p と q の名前の前後関係に従って (25 行目) 左右どちらかの子に追加するかを決め、add_contact を再帰的に呼び出す (26,28 行目)。このとき、p.left や p.right には連絡先がある場合と空の場合の両方がありえるが、add_contact は空の場合には q が、そうでない場合には元の連絡先 (p.left か p.right) を返す (30 行目) ことによって、結果として空だった場合にインスタンス変数 left または right に q を代入している。

この関数を使って、例えば図 9.2 の根に対して hirano と uchida という名前の連絡先を追加すると、木構造全体は図 9.3 のようになる。それぞれの連絡先が適切な場所に追加されている。

```
1 irb(main):010:0> root = phonebook()
2 (結果省略)
3 irb(main):011:0> root = add_contact(root,
4 irb(main):012:1*  Contact.new("hirano", "8602"))
5 (結果省略)
6 irb(main):013:0> root = add_contact(root,
7 irb(main):014:1*  Contact.new("uchida", "4027"))
8 (結果省略)
9 irb(main):015:0> view(root)
```

```
10 (結果省略)
```

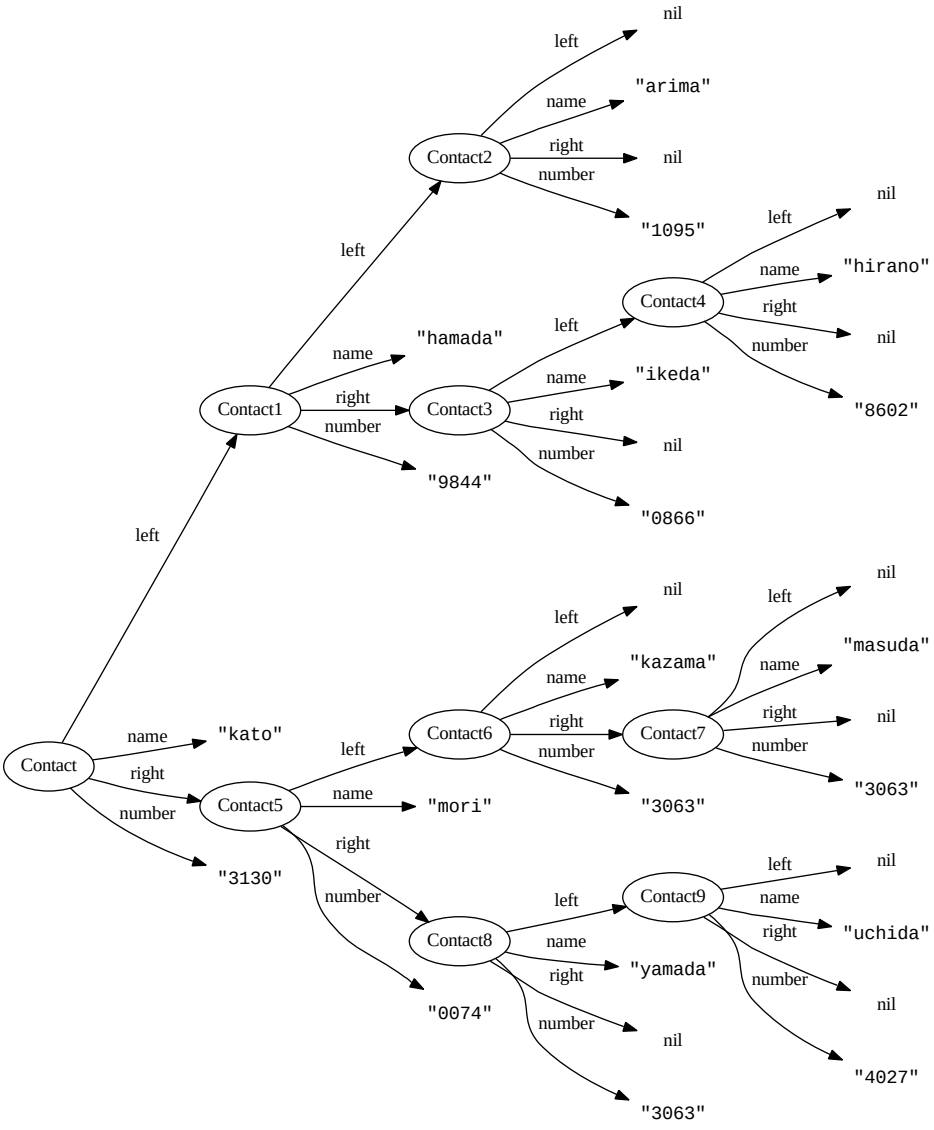


図 9.3: 電話帳に連絡先を追加した様子

人数を求める

与えられた電話帳にある連絡先の個数を数えることを考えよう。p を根とする電話帳にある連絡先の個数 size(p) は次のような関係で求められる。

$$\text{size}(p) = \begin{cases} 0 & (p \text{ が空の場合}) \\ 1 + \text{size}(p \text{ の左}) + \text{size}(p \text{ の右}) & (\text{それ以外}) \end{cases}$$

これはほぼそのまま Ruby の関数として定義でき、ファイル 9.7 のようになる。このように再帰データ構造に対する操作は、再帰的な関数によって定義することが多い。特にこの例のように、左右両方に対して操作を行う必要がある場合には、繰り返しによって定義するよりもはるかに簡単にできる。

```
33 def size(p)
34   if p == nil
35     0
36   else
37     1 + size(p.left) + size(p.right)
38   end
39 end
```

ファイル 9.7: contact.rb (続き: 人数を求める)

(章末の練習問題: 9.11 9.12 9.13)

9.2.5　連絡先の削除 *

連絡先 p を根とする電話帳から名前が n である連絡先 (q とする) を削除することを考えよう。

根である連絡先 p が削除される場合もあるので、関数 delete_contact(p,n) を「p を根とする電話帳から名前が n である連絡先を削除し、電話帳の**新しい根を返す**」ように定義する。使う場合には以下のように delete_contact からの答えを、根を記録している変数に代入し直す形になる。

```
p = delete_contact(p,"kato")
```

関数 delete_contact(p,n) は、名前の前後関係を利用して n という名前の連絡先を見つける。従って delete_contact(p,n) は、ファイル 9.8 に示すように、名前を追加する関数 add_contact(ファイル 9.6, p.163) と同じような形になる。

- 名前が一致した場合は、p を根とする電話帳の根を削除することになる。この部分は後で考えることにしよう。(42 行目。関数 delete_root は後で説明する)

```
40 def delete_contact(p, n)
41   if p.name == n
42     delete_root(p)
43   else
44     if n < p.name
45       p.left = delete_contact(p.left,n)
46     else
47       p.right = delete_contact(p.right,n)
48     end
49     p
50   end
51 end
```

ファイル 9.8: contact.rb (続き: 連絡先の削除)

- 一致しない場合は、delete_contact を再帰的に呼び出して左右どちらかの連絡先から削除をする。このとき p の直後の連絡先が削除される場合があるため、delete_contact の答でインスタンス変数 left または right を置き換える。(45, 47 行目)

  さらにこの場合は、p 自身は削除されなかったので関数の答として p を返す。(49 行目)

次に p を根とする電話帳の根を削除し、新しい根を答える delete_root(p) である。この関数は p の左右に直接つながっている子の数によって場合分けされる。

- 1 つも子がないない場合: p 自身が消えるので新しい電話帳は空になるだけである。
- 左右どちらか 1 方にだけ子がある場合: その子が新しい根になる。
- 左右両方に連絡先がつながっている場合: つながっている連絡先の中の 1 つを選び、それを新しい根とする。

最後の場合はややこみ入っているが、そこを除くとファイル 9.9 のように定義できる。

この関数ではまず left が空の場合に right を新しい根としている (54 行目)。これは「1 つも子がない場合」と「右 (right) にだけ子がある場合」を兼ねている。(前者の場合は nil が答だが、p.right は nil になっている。)

次は左 (left) にだけ子がある場合は、それを新しい根として答えている (57 行目)。最後は両方に連絡先がある場合には次に示す choose_root によって新しい根を選んでいる (59 行目)。

```
52 def delete_root(p)
53   if p.left == nil
54     p.right
55   else
56     if p.right == nil
57       p.left
58     else
59       choose_root(p)
60     end
61   end
62 end
```

ファイル 9.9: contact.rb (続き: 電話帳の根の削除)

さて、p の左右に子がいる場合にはどの連絡先を新しい根とすべきだろうか。電話帳全体が名前順に並んでいるためには、新しい根となるべき連絡先は、名前順で p の直前か直後の連絡先でなければいけない。ここでは p の直後の連絡先を選ぶことにすると、それは p の右の子を根とする電話帳の中で、名前順で最初の連絡先にほかならない。それを q と呼び、これを p の右の子孫からとり除いて p のあった場所に置くことにしよう。

ここで注意しなければいけないのは、q には左の子はいないが、右の子は空とは限らない点である。その場合は、q をとり除いた後に、q のあった場所に q の右の子を置く。

結局、新しい根を選ぶ手順をまとめると図 9.4 のようになり、関数 choose_root(p) はファイル 9.10 のように定義できる。

この定義ではまず、関数 first_contact(練習 9.4) を使って p の (名前順で) 直後に来る連絡先 q を見つける。これを新しい根として p のあった位置に置くために、先に q を元の場所から取り除く (65 行目)。ここでは delete_contact を再帰的に使って取り除いているので、もし q に右の子がいた場合には q の位置に q の右の子が入る。最後に q を p のあった位置におく。そのためには q の左右の子を p の左右の子に置き換えればよいので、q のインスタンス変数 left, right にそれぞれ p の左右の子を代入する (66, 67 行目)。このとき p の右には直接 q が現われていた場合があるので、67 行目は p.right ではなく、p.right から q を取り除いた結果 (r) を代入していることに注意せよ。

図 9.3 の状態の電話帳からいくつか連絡先を取り除いてみよう。

```
1 irb(main):017:0> root=delete_contact(root, "kato")
2 (結果省略)
3 irb(main):018:0> root=delete_contact(root, "ikeda")
4 (結果省略)
```

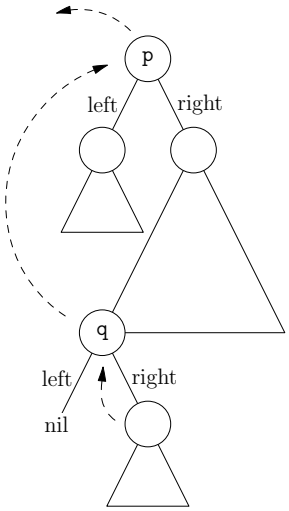


図 9.4: 新しい根を選ぶ手順 (三角形は省略された木構造を表わしている)

```
63 def choose_root(p)
64   q = first_contact(p.right)
65   r = delete_contact(p.right,q.name)
66   q.left = p.left
67   q.right = r
68   q
69 end
```

ファイル 9.10: contact.rb (続き: 新しい根を選ぶ)

```
5 irb(main):019:0> view(root)
6 (結果省略)
```

削除後の電話帳は図 9.5 のようになる。全体の根にあたる連絡先 (kato) が削除されたので、kazama の連絡先が新しい根として選ばれ、名前順が保たれていることが分かる。

(章末の練習問題: 9.15)

9.3　章末問題

練習 9.5 (予約リストの長さ) 予約情報 r から始まる予約リストの長さを求める関数 request_length(r) をファイル request_length.rb に定義せよ。(ヒント: 予約情報 r から始まるリストの長さと r.next から始まるリストの長さの再帰的な関係を考えよ。)

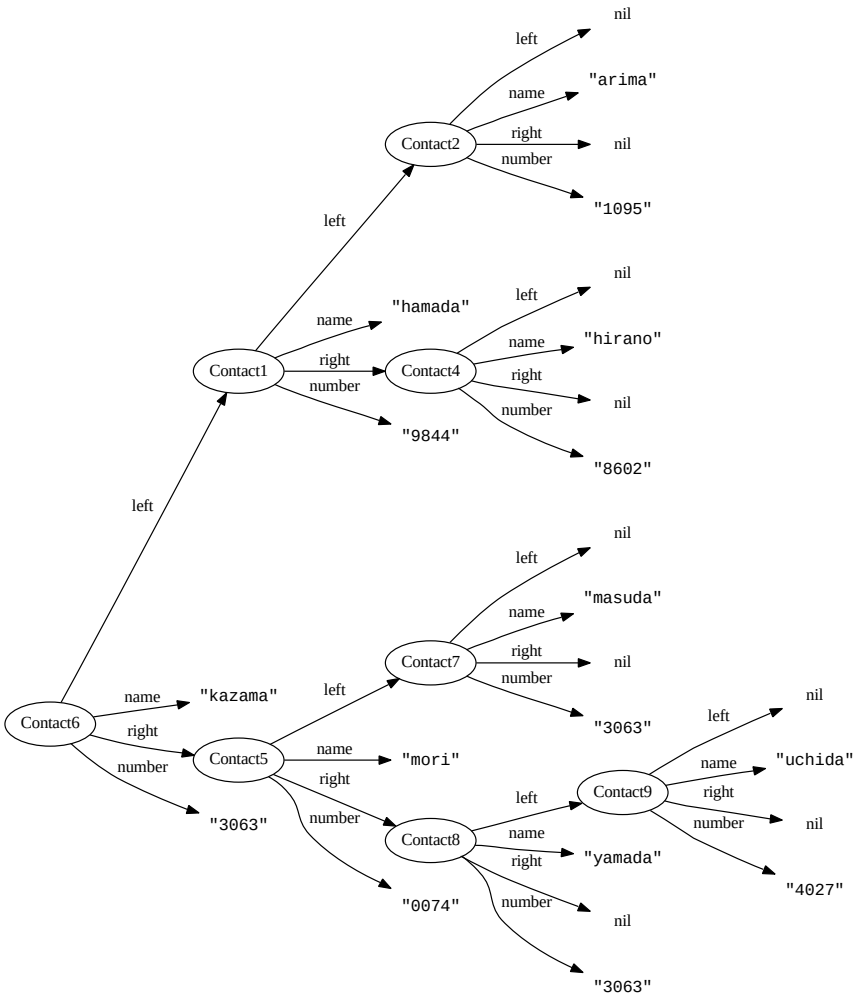


図 9.5: 電話帳から連絡先を削除した結果

練習 9.6 (複数の予約の削除) ファイル 9.3 はリストの先頭から見て最初に見つけた予約を 1 つだけ削除する。また、リスト中に必ず削除されるべき予約があることも仮定している。これを修正して、同じ曲名の予約が複数あった場合にその全てを削除し、また、削除されるべき曲がない場合でもエラーとならないようにせよ。修正した関数はファイル `delete_request.rb` に定義せよ。

練習 9.7 (名前順の追加) 予約情報 `r` から始まる予約リストが曲名順に並んでいたときに、新しい予約情報 `s` を適切な場所に追加し、追加後の予約リストを答える `add_alphabetically(r,s)` を定義せよ。ただし曲名順とは、予約リストにある予約情報 `t` が以下の関係を満たしていることだとする。

$$\texttt{t.title} < \texttt{u.title} \quad (\text{ただし } \texttt{u} = \texttt{t.next})$$

なお、第 9.2.2 節でも説明するように、Ruby では演算子<, >を使って文字列を比較できる。(ヒント: 予約の削除と同様に考えよ。`add_alphabetically(r,s)` が `s` を `r` の手前に追加するのはどのような場合か?)

練習 9.8 (循環リスト) ファイル 9.11 のような関数が作るリストを**循環リスト**という。

```
29 def make_cycle(list)
30   last = last_request(list)
31   last.next = list
32   list
33 end
```

ファイル 9.11: `request.rb` (循環リストの作成)

a) `view(make_cycle(ut_songs()))` を実行して、どのような構造になっているかを確認せよ。

b) カラオケ演奏機の予約リストにこのようなデータが用いられた場合、どのようなことが起きるか?

c) これまでに定義した、予約リストを扱う関数のうち、予約リストが循環していると計算が止まらない (あるいはエラーになる) ものはどれか?

練習 9.9 (カラオケ演奏機クラス) リスト構造 `Request` の操作を組み合わせて、カラオケ演奏機を模したクラス `Karaoke` をファイル `karaoke.rb` に定義せよ。

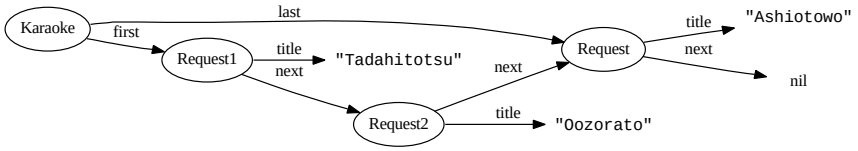
いま `k = Karaoke.new()` として変数 `k` に `Karaoke` オブジェクトをしまっているとする。クラス `Karaoke` は、以下のようなメソッドを持つものとする。

- `k.add("Tadahitotsu")` によって、リストの最後に予約を追加する
- `k.add_top("Tadahitotsu")` によって、リストの先頭に予約を追加する
- `k.play_next()` によって、次の曲を演奏する
- `k.cancel("Tadahitotsu")` によって、指定された曲名の予約を取り消す

練習 9.10 (末尾への追加の効率化) 予約リストの末尾に予約を追加する `add_request` は、リストの先頭から順にたどって最後の予約を見つけていた。この方法では n 個の予約があるときには n に比例した時間がかかるため、(カラオケでは考えにくいが) 非常に長いリストに何回も予約を追加する場合には適していない。

そこで「予約リスト中の最後の予約」を常に変数 `last` に覚えておき、予約の追加は `last` のインスタンス変数 `next` に新しい予約を代入するようにすれば、予約リストの長さによらずに数回の代入だけで予約の追加ができるようになる。(もちろん追加した後は変数 `last` は、新たに追加された予約に変更しておかなければいけない。)

練習 9.9 で定義した `Request` をこのような方法で改良してみよ。具体的にはクラス `Request` にインスタンス変数 `last` を追加して、メソッド `add` の定義を変更すればよい。改良された `Karaoke` オブジェクトを図示すると下のようになるだろう。ただし、予約が 1 件もない場合には「最後の予約」自体が存在しないことに注意せよ。



練習 9.11 (n 人目の連絡先) `p` を根とする電話帳の連絡先をアルファベット順に見たときに、先頭から n 目の連絡先をとり出す関数 `nth(p,n)` をファイル `nth.rb` に定義せよ。ただし、一番最初の連絡先は 0 番目だとする。また `size(p)` $\leq$ `n` の場合は考えなくてよい。(ヒント: `n` $<$ `size(p.left)` のとき、`n` 番目のは `p` の左右どちらにあるか?)

練習 9.12 (深さ) 電話帳の中である連絡先に到達するまでには、インスタンス変数 `left` や `right` を何回かたどる必要がある。その回数を根から連絡先までの深さと言う。電話帳の深さとは、その電話帳に含まれてる連絡先の深さの最大値である。`p` を根とする電話帳の深さを求める関数 `depth(p)` をファイル `depth.rb` に定義せよ。

練習 9.13 (二分木を用いた整列) 次のようにして配列を整列する関数`binarysort(a)` をファイル `binarysort.rb` に定義せよ。

a) 配列 a から電話帳を作る関数 array_to_tree(a) を定義せよ。具体的には、名前 (name) が a[0], 番号 (number) が nil であるような連絡先を作りそれを根とする。さらに 1 番目以降の a[i] を名前として番号が nil であるような連絡先を順に作って根に加える (add_contact)。最後に根を答えとする。

配列 a に入っている値は数値であっても構わない。ここまで連絡先の名前 (name) は文字列だとして説明していたが、実際には ==, <, > によって比較できる値であれば何でもよい。

b) 連絡先 p からたどれる全ての連絡先を名前順にたどり、その名前 (name) を配列 a の i 番目以降に代入する tree_to_array(p,a,i) を定義せよ。またこの関数は、最後に代入した番号を答とする。

c) 関数 array_to_tree, tree_to_array を使って binarysort(a) を定義せよ。

練習 9.14 (二分木を用いた整列の計算量) 練習 9.13 で定義した整列アルゴリズムの計算量を考えてみよう。

a) p を根とする木の深さが n のとき、add_contact(p,q) の計算量を O 記法で表わせ。

b) m 個の連絡先を持つ電話帳の深さが最も小さい場合、その深さを m を使った式で表わせ。以降では、このようになっているときに「理想的な場合」と呼ぶことにする。

c) array_to_tree(a) の結果としてできる電話帳が理想的だったときの array_to_tree の計算量を O 記法で表わせ。ただし a の大きさを m とする。

d) 関数 tree_to_array(p,a,i) の計算量を O 記法で表わせ。ただし p からたどれる連絡先の個数は m だとする。

e) 関数 binarysort(a) の計算量を O 記法で表わせ。ただし a の大きさは m, 途中でできる電話帳は理想的だったとする。

f) array_to_tree(a) が作る電話帳が最も深くなるのは a に並んでいる値がどのような場合か。またそのときの深さはいくらになるか。

練習 9.15 (深さを考えた削除) 関数 choose_root(p) は、p の左右に連絡先があった場合、新しい根として名前順で p の 直後 に来る連絡先を選んでいる。しかしこれは、p の直前の連絡先としても構わない。そこで、p の左右に連絡先があった場合に、そこからたどれる連絡先の個数が多い側から次の根を選ぶように choose_root(p) をファイル contact_balance.rb に定義せよ。

練習 9.16 (図形グループ) 第 8.2.4 節では、複数の図形から成る図案を表示・変形するために、配列に図形要素をしまって drawall, turnall などの関数を定義していた。

この方法のかわりに「図形をグループ化した図形」のクラス Group をファイル oo-group.rb に定義してみよ。クラス Group のオブジェクトは、下のように図形要素配列を引数として作成され、draw や turn などのメソッドを持つものとする。

```
load("./oo-group.rb")
load("./oo-face.rb")
g=Group.new(face())#face()は練習 8.11 を参照
a=make2d(400,400)
g.draw(a)
show(a)
```

このクラス Group は、draw や turn ができるので、Line や Circle と同じく図形要素の 1 種である。しかも Group の内部に入るデータはまた図形要素であるので、再帰データ構造になっている。上の例では、Line や Circle などの図形要素から成るグループを作っているが、それをさらにグループ化して「複数の図形グループから成る図形グループ」といったような構造を作ることもできる。