情報科学 2006　久野クラス　＃6

久野　靖*

2006.12.1

はじめに

駒場祭はお楽しみになりましたか。11 月はなんだかんだで 1 回しか授業ができませんでしたが、12 月はみっちり 4 回できますから頑張って進みましょう。今回は新しい内容として次のものをやります。

- 有限オートマトン (文字列などの解析に適した状態遷移モデル)
- 動的データ構造/再帰的データ構造

その前に演習問題解説がありますが、ゆっくり説明していると本題の時間がありませんので、おおむね「読んでおいてください」ということでお願いします。

1　演習問題解説

1.1　演習 4a

このあたりは簡単にコードだけ示そう。大文字にするのはメソッド toUpperCase() を呼ぶだけでよい。呼び方は「toUpperCase(str)」ではなく「str.toUpperCase()」ですね。メソッドの呼び方に慣れていただくための演習という趣旨でした。

```
import java.io.*;

public class r5ex4a {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    while(true) {
      System.out.print("String> ");
      String str = in.readLine();
      if(str.equals("")) break;
      String res = str.toUpperCase(); // ☆以下ではこの
      System.out.println(res);        // ☆ 2 行だけ変更していく
    }
  }
}
```

なお

```
  System.out.println(str.toUpperCase());
```

のように 1 行にしてしまって変数 res は使わない、というのでもよい。ときに

```
  str.toUpperCase();
```

とだけ書いてうまく行かないで悩んでいる人がいたようだが、あくまでも toUpperCase() は「小文字を大文字に変えた新しい文字列を返す」のであって、その結果を変数に入れない限り元の str は元のまんまであることに注意。

*筑波大学大学院経営システム科学専攻

1.2 演習 4b

以下では上のリスティングで☆のついた行の差し換えだけ示します。

```
String res = str.replace('a', '*');
System.out.println(res);
```

これは上とほとんど同様だが、`toUpperCase()` ではメソッドに引数がなかったのに対し、`replace()` では「どの文字を」「どの文字に」という 2 つの引数が必要になる。なお、Java では「`"..."`」は String(文字列) を表し、「`'?'`」は 1 文字を表すという区別があるのに注意。なお、先の例題と同様いきなり

```
System.out.println(str.replace('a', '*'));
```

としても構わない (美観や趣味の問題)。

1.3 演習 4c

これはヒントにあるように、`replace()` を 5 回使えばできる。

```
String res = str.replace('a', '*');
res = res.replace('e', '*');
res = res.replace('i', '*');
res = res.replace('o', '*');
res = res.replace('u', '*');
System.out.println(res);
```

まず「a」は先の通り処理したから、次に「`res` に対して」今度は「e」の処理を同様に行い、その結果を…新しい変数に入れてもいいのだが、同じ変数でもいいので `res` に入れ直してしまう。これを続ければいいわけだ。

ところで、毎回 `res` に入れ直さなくても次のようにしてもよい。

```
String res = str.replace('a', '*').replace('e', '*').
  replace('i', '*').replace('o', '*').replace('u', '*');
```

なぜこれでいいかというと、`replace()` が返すのは `String` オブジェクトだから、それに対しまた次の `replace()` を呼んでも構わないから。この方がかっこいいかどうか?

しかし、5 文字くらいだったらこれでも (また先の方法でも) いいけど、20 文字くらいになったらもう書くのがイヤになるでしょう? そこで次のようにしてもよい。

```
String res = str;
String sub = "aeiou";
for(int i = 0; i < sub.length(); ++i) {
  res = res.replace(sub.charAt(i), '*');
}
```

つまり、置き換える文字をまとめて文字列として用意し、その各文字を引数として `replace()` を適用する、というのを for 文で繰り返すわけである。まあ今回はここまでしなくてもよかったけど。なお、正規表現関係のメソッドを使えばもっと簡単にできます (興味あったら調べてみてください)。

1.4 演習 4d

これは `substring()` を使って文字列の先頭文字から N 文字目までを取り出す、というのを N を変えながら繰り返せばよい。

```
for(int i = 0; i < str.length(); ++i) {
  System.out.println(str.substring(0, str.length()-i));
}
```

つまり、`str.length()` が 4 であれば、for 文の中で「`str.substring(0, 4)`」、「`str.substring(0, 3)`」、「`str.substring(0, 2)`」、「`str.substring(0, 1)`」を順に出力するわけである。

1.5 演習 4e

これは後ろの方から取り出せばいいのだが、その前に適当な数の空白を用意しないと右側がそろってくれない。その空白もループ 1 周につき 1 文字ずつ増えるのだから、もう 1 つ変数を用意して、それに空白を 1 文字ずつ連結して伸ばして行く。

```
String space = "";
for(int i = 0; i < str.length(); ++i) {
  System.out.println(space + str.substring(i));
  space = space + " ";
}
```

なお、substring() でパラメタ (引数) の数が 1 つのものは、その文字から末尾までを取り出すことになっている。しかし、せっかくさまざまなメソッドがあるのに「1 文字ずつ空白をくっつける」という細かい操作をやるのは悔しいですね。たとえば、space という変数にうんと長い空白文字列が入れてあったとすると、N 文字の空白は「space.substring(0, N)」と書けば済むでしょう? その方向で「別解」を考えてみよう。

ここで問題は、どんな入力を持って来てもそれより長いような空白文字列を予め用意することができない、ということ。しかし、使う前に「十分長く」してしまうことは簡単ですね。というわけで、まずメソッドの先頭部分で

```
String space = "                         "; // とりあえずの長さ
```

とした上で、☆の個所を次のようにすればよい。

```
while(space.length() < str.length()) {  // 長さが足りなければ
  space = space + space;                 // 長くする!
}
for(int i = 0; i < str.length(); ++i) {
  System.out.println(space.substring(0, i) + str.substring(i));
}
```

1.6 演習 4f

これはじつは簡単で、単に「2 文字目から最後まで」と「先頭」とをくっつければ 1 文字ずれるので、それを何回も単に繰り返す。

```
String res = str;
for(int i = 0; i < str.length(); ++i) {
  res = res.substring(1) + res.charAt(0);
  System.out.println(res);
}
```

1.7 演習 5a

ここからは入力がいらない (引数で渡す) し、ループも不要なのでプログラムは短くなる。

```
public class r5ex5a {
  public static void main(String args[]) throws Exception {
    String str = args[0];
    String res = "", up = str.toUpperCase(), low = str.toLowerCase();
    for(int i = 0; i < str.length(); ++i) {
      if(str.charAt(i) != up.charAt(i)) { res += up.charAt(i); }
      else                              { res += low.charAt(i); }
    }
    System.out.println(res);
  }
}
```

「一気に」全部大文字や小文字にするのなら toUpperCase() や toLowerCase() を使えばいいが、入れ換えるとなると面倒である。考え方としては、toUpperCase() で「変化した」文字があればそれは元が小文字だったのでその変化した値を取ればいい、それ以外の文字は大文字にせよ英字ではない文字にせよ toLowerCase() の結果を取ればいい、ということ。

1.8 演習 5b

長さ L の文字列が回文かどうか調べるには、1 文字目と L 文字目、2 文字目と $L-1$ 文字目、…が等しいかどうか順次調べていって、等しくないところが見つかれば回文ではない、最後まで等しければ回文という方針にする。これを下請け関数 kaibun() で判断する。

```
public class r5ex5b {
  public static void main(String args[]) throws Exception {
    System.out.println(kaibun(args[0]));
  }
  public static boolean kaibun(String s) {
    for(int i = 0, j = s.length()-1; i < j; ++i, --j) {
      if(s.charAt(i) != s.charAt(j)) { return false; }
    }
    return true;
  }
}
```

for 文で 2 つの変数 i、j を扱っているところはちょっと目新しいかも知れない。

1.9 演習 5c

文字列 str の長さが L だとして、str から長さ 3、4、…、L の部分文字列を順次取り出して先の方法で回文かどうか調べ、回文なら打ち出す。

```
public class r5ex5c {
  public static void main(String args[]) throws Exception {
    String str = args[0];
    for(int i = 3; i <= str.length(); ++i) {
      for(int j = 0; j+i <= str.length(); ++j) {
        String t = str.substring(j, j+i);
        if(kaibun(t)) { System.out.println(t); }
      }
    }
  }
  public static boolean kaibun(String s) {
    for(int i = 0, j = s.length()-1; i < j; ++i, --j) {
      if(s.charAt(i) != s.charAt(j)) { return false; }
    }
    return true;
  }
}
```

メソッド substring() の使い方がちょっと分かりにくかったでしょうか。先頭の文字番号と、取り出す**次の**文字番号を指定するので。

1.10 演習 5d

これはむしろ 5c より簡単で、2 番目の文字列と同じ長さの部分文字列だけを取り出してチェックすればよい。

```
public class r5ex5d {
  public static void main(String args[]) throws Exception {
    String s1 = args[0], s2 = args[1];
    boolean found = false;
    for(int i = 0; i+s2.length() < s1.length(); ++i) {
      if(s1.substring(i, i+s2.length()).equals(s2)) { found = true; }
    }
    System.out.println(found);
  }
}
```

ところで 2 番目の方が長い文字列だったらどうなるでしょう? そのときは、その長さの部分文字列が存在しないので for 文の本体は 1 回も実行されず、found は元のままだから false が出力される。正しいでしょう?

1.11 演習 5e

今度は 2 番目の文字列が「間欠的に」存在しているかどうか見るのでちょっと面倒である。このようなものは再帰関数が適している。つまり、次のように考える:

- s2 が空文字列なら true(空文字列はどこにでも含まれているから)。
- そうでなければ s1 の中で s2 の先頭文字が含まれている箇所を探す。なければ false。
- あれば、s1 のその位置以降に、s2 の 2 文字目以降が間欠的に含まれているかどうか調べればよい。

これをコードにするとたとえば次のようになる:

```
public class r5ex5e {
  public static void main(String args[]) throws Exception {
    String s1 = args[0], s2 = args[1];
    System.out.println(intermittentFind(s1, s2));
  }
  public static boolean intermittentFind(String s1, String s2) {
    if(s2.equals("")) { return true; }
    for(int i = 0; i < s1.length(); ++i) {
      if(s1.charAt(i) == s2.charAt(0)) {
        return intermittentFind(s1.substring(i+1), s2.substring(1));
      }
    }
    return false;
  }
}
```

1.12 演習 5f

これも実は前の問題とよく似ているが、「s2 が空文字列なら s1 も空文字列のときだけ true」「s1 の中に s2 の 1 文字目が見つかったときは s1、s2 ともにその文字を取り除いて残りについてアナグラムかどうか調べる」という形になる。

```
public class r5ex5f {
  public static void main(String args[]) throws Exception {
    String s1 = args[0], s2 = args[1];
    System.out.println(anagram(s1, s2));
  }
  public static boolean anagram(String s1, String s2) {
    if(s2.equals("")) { return s1.equals(""); }
    for(int i = 0; i < s1.length(); ++i) {
      if(s1.charAt(i) == s2.charAt(0)) {
        return anagram(s1.substring(0,i)+s1.substring(i+1), s2.substring(1));
      }
    }
    return false;
  }
}
```

1.13 演習 6

この演習は 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。追加する部分近辺だけ示そう。まず main() では演算を使って見せる部分を増やした。

```
System.out.println(r1 + " + " + r2 + " = " + (r1.add(r2)));
System.out.println(r1 + " - " + r2 + " = " + (r1.sub(r2)));
System.out.println(r1 + " * " + r2 + " = " + (r1.mul(r2)));
System.out.println(r1 + " / " + r2 + " = " + (r1.div(r2)));
```

そしてメソッドの追加は、加算以外の 3 つの演算も普通にやればよい。除算はひっくり返して掛ければいいわけだし。

```
public Rational add(Rational r) {
  return new Rational(a*r.b + r.a*b, r.b*b);
}
public Rational sub(Rational r) {
```

```
    return new Rational(a*r.b - r.a*b, r.b*b);
  }
  public Rational mul(Rational r) {
    return new Rational(a*r.a, b*r.b);
  }
  public Rational div(Rational r) {
    return new Rational(a*r.b, b*r.a);
  }
```

1.14 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。

```
import java.io.*;

public class r5ex7 {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("a1 = ");
    double a1 = new Double(in.readLine()).doubleValue();
    System.out.print("b1 = ");
    double b1 = new Double(in.readLine()).doubleValue();
    System.out.print("a2 = ");
    double a2 = new Double(in.readLine()).doubleValue();
    System.out.print("b2 = ");
    double b2 = new Double(in.readLine()).doubleValue();
    Complex r1 = new Complex(a1, b1);
    Complex r2 = new Complex(a2, b2);
    System.out.println(r1 + " + " + r2 + " = " + (r1.add(r2)));
    System.out.println(r1 + " - " + r2 + " = " + (r1.sub(r2)));
    System.out.println(r1 + " * " + r2 + " = " + (r1.mul(r2)));
    System.out.println(r1 + " / " + r2 + " = " + (r1.div(r2)));
    System.out.println("check: " + r1.div(r2).mul(r2));
  }
  static class Complex {
    double a, b;
    public Complex(double x) { a = x; b = 0; }
    public Complex(double x, double y) { a = x; b = y; }
    public Complex add(Complex r) {
      return new Complex(a+r.a, b+r.b);
    }
    public Complex sub(Complex r) {
      return new Complex(a-r.a, b-r.b);
    }
    public Complex mul(Complex r) {
      return new Complex(a*r.a - b*r.b, a*r.b + b*r.a);
    }
    public Complex div(Complex r) {
      double norm = r.a*r.a + r.b*r.b;
      return new Complex((a*r.a + b*r.b)/norm, (b*r.a - a*r.b)/norm);
    }
    public String toString() {
      if(b < 0) { return a + (b + "i"); } else { return a + "+" + b + "i"; }
    }
  }
}
```

`toString()` 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。そしてこのとき、$a+b$ を先に計算してしまうと足し算されてしまうので、`b` と文字列`"i"`の連結を先にするようにかっこをつけている。

1.15 演習 8

これは結構骨が折れますね。できるだけ簡単にするため、正の整数のみ、演算は足し算と掛け算のみで作った。main() は資料に書いた通り。

```
public class r5ex8 {
  public static void main(String[] args) throws Exception {
    int x = (new Integer(args[0])).intValue();
    Bigint n = new Bigint(1);
    for(int i = 2; i <= x; ++i) { n = n.mul(new Bigint(i)); }
    System.out.println("fact(" + x + ") = " + n);
  }
```

問題は Bigint クラスだが、配列の大きさは実行時に決めるようにして、なおかつ下から何桁まで使っているかを変数 count で持つようにした。コンストラクタは配列の大きさを指定するものとしないもの (20 に決め打ち) を用意し、前者はクラス内からだけ呼べるようにした。「this(...)」という書き方は、コンストラクタの中で同じクラスの別のコンストラクタを呼び出す時に使う記法。2 引数のコンストラクタでは渡された整数を 1 桁ずつ分解して配列に入れている。

```
  static class Bigint {
    int[] a; int count = 0;
    public Bigint(int n) { this(n, 20); }
    private Bigint(int n, int size) {
      a = new int[size];
      while(n != 0) { a[count] = n%10; ++count; n = n / 10; }
    }
```

shiftMulAdd() というメソッドは、渡された Bigint を $m \times 10^s$ 倍して自分の値に足し込むもので、後の掛け算のときに使う (ついでに足し算でも)。1 桁足すごとに桁上がりを伝播する必要があることに注意。ついでに count の値も最後の 0 でない桁が入るようにしている。

```
    private void shiftMulAdd(Bigint b, int m, int s) {
      for(int i = 0; i <= b.count; ++i) {
        a[i+s] += b.a[i]*m;
        for(int j = i+s; a[j] >= 10; ++j) { a[j+1] += a[j]/10; a[j] %= 10; }
        if(a[i+s] != 0) { count = i+s; }
      }
    }
```

shiftMulAdd() があれば、足し算も掛け算もこれを呼ぶだけでできる。演算結果が何桁あれば収まるかを計算して結果用の Bigint を割り当てている。

```
    public Bigint add(Bigint b) {
      Bigint c = new Bigint(0, Math.max(count,b.count)+1);
      c.shiftMulAdd(this, 1, 0); c.shiftMulAdd(b, 1, 0); return c;
    }
    public Bigint mul(Bigint b) {
      Bigint c = new Bigint(0, count+b.count+1);
      for(int i = 0; i <= count; ++i) { c.shiftMulAdd(b, a[i], i); }
      return c;
    }
    public String toString() {
      String str = "";
      for(int i = 0; i <= count; ++i) { str = a[i] + str; }
      return str;
    }
  }
}
```

toString() は右から 1 桁ずつくっつけて行けばいいだけ。これで確かに 100 の階乗は計算できたので動作は大丈夫でしょう。

```
% java r5ex8 4
fact(4) = 24
% java r5ex8 5
fact(5) = 120
% java r5ex8 100
fact(100) = 93326215443944152681699238856266700490715968264381621
46859296389521759999322991560894146397615651828625369792082722375
82511852109168640000000000000000000000000
%
```

2 有限オートマトン

2.1 決定性有限オートマトン

「ある文字列がある規則にあてはまっているかどうかを調べる」という問題を考えてみる。規則とは、たとえば「a からはじまり、a と b が交互に現れ、最後は b で終わる※」みたいなものをいう。どうやって調べます?

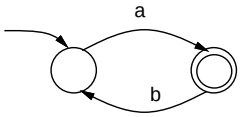

図 1: オートマトンの例

以下ではこのような問題を「複数の状態の間を遷移していく」というモデルで考えていく (図 1)。ここで「○」は**状態**を表す。そのうちの 1 つは**初期状態** (最初にいるところ)、「◎」は**受理状態**/**最終状態** (最後にここにいれば OK)。文字列の文字を 1 つ進むごとに、その文字のラベルがついた矢線を進んで次の状態に**遷移**する。遷移できなければ「NO」。最後の文字が終わった時に「◎」にいない場合も「NO」。図 1 を実際にたどってみると、※にあてはまる文字列だけ「OK」になることが分かる。このようなグラフ (初期状態、受理状態を含む有限個の状態とそれらを結ぶラベルつき矢線から成るグラフ) のことを**有限オートマトン** と呼ぶ。

このような有限オートマトンを Java のクラスで実現してみる (名前を `Automata` とした)。

- `Automata(int 状態数, int 初期状態, int[] 受理状態)` — 状態数、初期状態番号、受理状態番号 (複数あってもよいので配列で指定) を指定しデータ構造を初期化。
- `void trans(int 状態 1, String ラベル, int 状態 2)` — 「状態 1 である文字が来たら状態 2 へ」という遷移を追加する。複数の文字についてまとめて指定できるように文字列で指定。
- `boolean accept(String 文字列)` — 指定した文字列が OK か否かを判定する。

つまり、最初に `new` で用意したあと、矢線の情報を `trans()` で必要なだけ追加するようになっている。これを使って※を受理判定するプログラムを作ってみた (クラス `Automata` も後につけた):

```
public class R6Sample1 {
  public static void main(String[] args) throws Exception {
    Automata atm = new Automata(2, 1, new int[]{1});
    atm.trans(1, "a", 2); atm.trans(2, "b", 1);
    System.out.println(atm.accept(args[0]));
  }
  static class Automata {
    int init; boolean[] fin; byte[][] next;
    public Automata(int s, int i, int[] f) {
      init = i; next = new byte[s][65536]; fin = new boolean[s];
      for(int k = 0; k < f.length; ++k) { fin[f[k]-1] = true; }
    }
    public void trans(int f, String s, int t) {
      for(int i = 0; i < s.length(); ++i) { next[f-1][s.charAt(i)]=(byte)t; }
    }
    public boolean accept(String s) {
      int cur = init;
      for(int i = 0; i < s.length(); ++i) {
        if((cur = next[cur-1][s.charAt(i)]) == 0) { return false; }
      }
      return fin[cur-1];
    }
  }
}
```

`main()` の側は、※のオートマトンを用意して、コマンド行で指定した文字列に対してオートマトンによる判定結果を出力するだけ。

Automata クラスの方を見てみよう。初期状態の番号を保持する変数 init、各状態についてそれが最終状態か否かを保持する boolean の配列 fin、および矢線の情報を保持する 2 次元配列 next がインスタンス変数。next については、要素 next[状態番号-1][文字番号] にその文字で遷移する次の状態 (遷移がない場合は 0) が格納されているものとした。文字番号は char 型の値で、0〜65535 の範囲を取るので、配列の 2 次元目もその範囲に合わせて取る。やや大きいので領域節約のため byte 型を利用することにして、状態の数は最大 128 まで、ということにした。1 次元目は状態数だけ用意すればよいが、0 を「状態なし」の意味で使っているので、状態数が N なら状態番号は 1〜N、これに対して配列の添字は 0 からだから配列を参照するときは 1 を引いている。

コンストラクタでは単にこれらのインスタンス変数を初期化するだけ (配列については、特に何も入れないので byte 値はすべて 0、論理値はすべて false が入っている)。fin については、渡された配列の各要素について、その番号-1(マイナス 1 する理由は上と同じ) の要素を true に変更することで最終状態であることを表す。

trans については、渡された文字列の各文字について、その文字に対する矢線の遷移を配列 next に記録している。

accept については、現在の状態 cur を最初は初期状態にして、そこから 1 文字ずつ配列 next に従って遷移していき (途中で 0 つまり遷移なしになったら false を返す)、最後まで来たら「現在状態が受理状態か否か」の論理値を返せばよい。動かしている様子を示す:

```
% java R6Sample1 abab
true
% java R6Sample1 ababa
false
% java R6Sample1 abababab
true
```

演習 1 上のプログラムをそのまま打ち込んで動かせ。動いたら、オートマトンを図 2 の (a)〜(c) のものに変更してみよ (最初に作るところで指定する状態数を増やすのを忘れないように)。これらがどのような文字列を受理するオートマトンかも考えること。

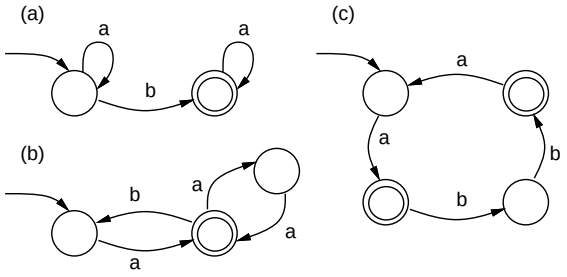

図 2: 演習 1 の問題のオートマトン

演習 2 次のような文字列「のみ」を受理するオートマトンを考え、それを上記のプログラムに組み込んで動かしてみよ。

(a) a と b がさまざまに混ざって現れるが、a の連続は最大 2 個まで。

(b) a と b がさまざまに混ざって現れるが、a は偶数個。

(c) Java の名前 (英字からはじまり、英字と数字が任意個並んだもの)。

(d) Java の整数定数。具体的には、符号があってもなくてもよく、続いて数字が 1 個以上並んだもの。

(e) Java の実数定数。具体的には、符号があってもなくてもよく、続いて数字が 1 個以上並び、小数部 (「.」と数字 1 個以上の並び) があるか、指数部 (「e」または「E」に続いて符号があってもなくてもよく、その後に数字 1 個以上の並び) があるか、または小数部と指数部の両方がこの順であるかのいずれか。

2.2 非決定性オートマトン

演習をやってみて分かったと思うが、有限オートマトンとは要するに「現在 N 個の状態のうちどこにいるか」という情報だけしか記憶できないような装置のモデルになっている。だから、「奇数か偶数か」「既に小数点を通過したか」といったことは状態で表せるが、「何個あるか」は (整数は無限にあるので) 記憶できない。このため、「a が並び、続いて b が同個数並ぶ」のようなものは有限オートマトンでは表現できない。

ところで、前節で説明したオートマトンでは、1 文字ごとに次の状態は一意的に (曖昧さなく) 決っていた。このようなオートマトンを**決定性有限オートマトン** (Deterministic Finite Automata、DFA) と呼ぶ。先のようなプログラムで簡単にオートマトンが実現できたのは、決定性有限オートマトンだったからだといえる。

一般のオートマトンでは、次の状態が一意に決まらないこともある。たとえば図 3 のようなものがそうである。ここで「ε」のラベルがついた遷移は**空遷移**と呼び、空な文字列に対応して遷移できる。空な文字列はどこにでもいくつでもあると考えることができるので、空遷移は「たどってもたどらなくてもよい」ので、一意に決まらない遷移になる (空遷移がなくても、ある状態から同じラベルを持つ 2 つの矢線が出ていればやはり一意に決まらなくなる)。このような、決定性でない有限オートマトンを**非決定性有限オートマトン** (Nondeterministic Finite Automata、NFA) と呼ぶ。NFA での受理は「さまざまな選択肢のうち 1 つ以上において受理状態に到達して終わる」ことと定義される。

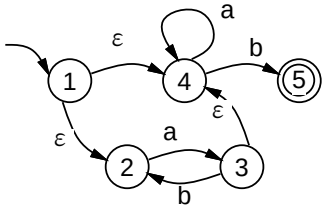

図 3: 非決定性オートマトン

こうして見ると、NFA の方が DFA よりも広い表現力を持っているような気がしますよね? ところが。ここで驚くべき先人の成果をお知らせしよう。実は、**任意の非決定性有限オートマトンに対して、それが受理するものと同じものだけを受理する決定性オートマトンを作ることができる。**

それには「NFA の状態の**集合**が 1 つの状態であるようなオートマトン」を作る。具体的には次のようにすればよい (図 4)。

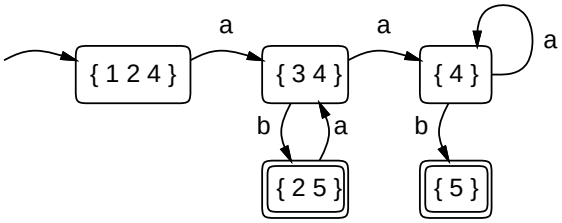

図 4: 非決定性オートマトンの決定化

NFA の初期状態 s に対して、ε 遷移で到達できるようなすべての状態の集合を求め、これを集合 S とする。図 3(状態に番号を割り振って表している) の場合は「{ 1 2 4 }」がこれに当たる。これが DFA(図 4) の初期状態となる。

次に、S に含まれる NFA の状態の中に文字 x で遷移可能なものがあるなら、そのような状態のどれかから x と ε によって遷移可能なすべての状態の集合 T を求め、DFA では S から T へ a で遷移するものとする。

図 4 の場合、「{ 1 2 4 }」から「a」で遷移できるのは状態 1 ではなし、状態 2 のとき 3、状態 4 のとき 4 なので、3 から 4 への空遷移も併せて (この場合は併せても変化しないが)「{ 3 4 }」となる。「{ 1 2 4 }」から「b」での遷移はない。同様にして、「{ 3 4 }」から「a」での遷移は「{ 4 }」、「b」での遷移は「{ 2 5 }」。「{ 4 }」から「a」での遷移は「{ 4 }」、「b」での遷移は「{ 5 }」。「{ 5 }」からはこれ以上遷移はない。「{ 2 5 }」から「a」での遷移は「{ 3 4 }」、「b」での遷移はない。これらをまとめると図 4 が完成する。なお、最終状態は元の NFA での最終状態である 5 を含む状態 2 つということになる。

このようにして次々に何らかの記号で遷移可能な DFA の状態 (NFA の状態の集合に相当) を求めていくが、有限集合のすべての部分集合は有限個しかないので、いつかは状態も遷移も増えなくなって停止する。

そして、できあがった DFA について見れば、ε 遷移はなくなっており、またある状態から 1 つの記号で遷移する先は 1 つしかないので、確かに決定性有限オートマトンになっていることが分かる。

なお、決定性への変換と類似したものとして**最小化**、つまりあるオートマトンに対して、それと同じものを受理する最も状態数が少ないオートマトンを求めるという手順も存在している (詳細略)。

演習 3 図 5 の非決定性オートマトンを決定性オートマトンに変換せよ。また、それを Java プログラムとして動かせ。

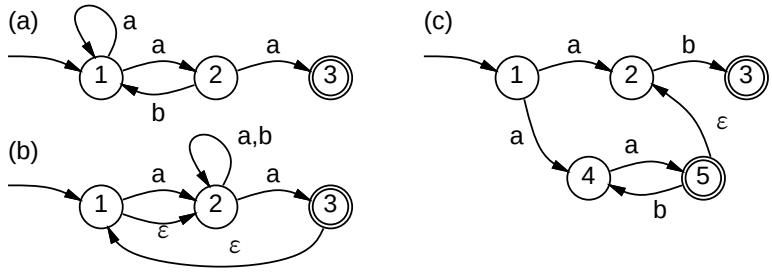

図 5: 演習 3 の非決定性オートマトン

2.3 正則表現 (正規表現) とオートマトン

オートマトンは見た目分かりやすいが、グラフなのでキーボードから打ち込んだりするのには向いていない。そこで実用的には、**正則表現** (**正規表現**) という別の記法で文字列が従う規則を表すことが多い。正則表現の記法を以下に示す:

a — 文字 a そのものを表す。
$\alpha \mid \beta$ — 「α にあてはまるもの、または β にあてはまるもの」を表す。
$\alpha\,\beta$ — 「α にあてはまるものに続いて β にあてはまるもの」を表す。
$\alpha*$ — 「α が 0 個以上繰り返したもの」を表す。
$\alpha+$ — $\alpha\,\alpha*$とおなじ。
(α) — α とおなじ (くくり出しのかっこ)。

たとえば、上で問題に出した「Java の実数定数」は正則表現では次のように書ける:

$$(+|-|)(0|\cdots|9)+(.(0|\cdots|9)+|\ (e|E)(+|-|)(0|\cdots|9)+|\ .(0|\cdots|9)+(e|E)(+|-|)(0|\cdots|9)+)$$

そして、任意の正則表現について、それにあてはまる文字列と同じものだけを受理するオートマトンが構成できる。実はその逆も言えて、従ってオートマトンと正則表現は表現能力が等価だということが示されているのだが、ここでは正則表現からオートマトンへの変換方法だけ示そう。

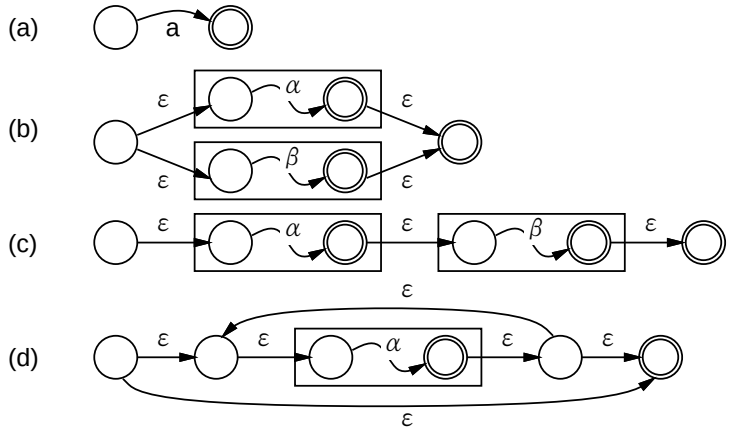

図 6: 正則 (正規) 表現とオートマトンの対応

- 記号 a については、それを受理するオートマトンは図 6(a) のように作ることができる。
- $\alpha \mid \beta$ については、α と β を受理するオートマトンをもとに、図 6(b) のように作ることができる。
- $\alpha\,\beta$ については、α と β を受理するオートマトンをもとに、図 6(c) のように作ることができる。

- α*については、αを受理するオートマトンをもとに、図 6(d) のように作ることができる。
- α+については、αを受理するオートマトンをもとに、図 6(d) から一番下の矢線を取った形で作ることができる。

実際にはこうやって作ったものはε遷移だらけの非決定性オートマトンだが、これを決定性オートマトンに変換できることは上で述べた通り。

演習 4 次の正則表現 (正規表現) を NFA に変換し、続いて DFA に変換して Java プログラムで動かせ。

(a) $ab*a$
(b) $a(ba+)+$
(c) $(ab*c)|(a*cb+)$

3 動的データ構造/再帰的データ構造

3.1 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。Java では、オブジェクト型の値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 (クラス) を見てみる:

```
static class Cell {
  public String str; Cell next;
  public Cell(String s) { str = s; }
}
```

これは、`next` というフィールドに他のセルへの参照を入れることができるから、これを使って「数珠つなぎ」の動的データ構造を作ることができる (図 7(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 `str` も `String` 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。

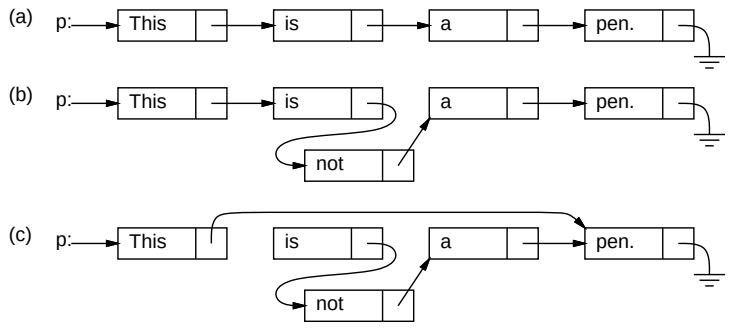

図 7: 単連結リストの動的データ構造

なお、`Cell` の定義を見ると `next` がまた `Cell` 型になっているので、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 7 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 `null` 値が入っている (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 7(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 7(c) で使わなくなった「is」「not」「a」の箱はどうなるのだろう? Java 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 7 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。

3.2 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
>                         ←次の行
  This is not a book.
>                         ←次の行
  How are you?
>                         ←次の行
  EOF                     ←終わりの印
>t                        ←再度先頭へ
  This is a pen.
>iI am a boy.             ←挿入
>                         ←次の行
  This is not a book.
>d                        ←削除
  How are you?
>t                        ←先頭から全部見る
  I am a boy.
>
  This is a pen.
>
  How are you?
>
  EOF
>q
```

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、これを実現するプログラムを見ていただこう。

まず main() から。このプログラムでは、単連結リストのセルを上記の Cell クラスであらわし、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。

main() の冒頭では 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 であるように参照を格納していることにも注意)。

```
import java.io.*;

public class R6Sample2 {
  static Cell head, tail, prev, cur;
  public static void main(String[] args) throws Exception {
    head = new Cell(""); tail = new Cell("EOF"); head.next = tail; top();
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    while(true) {
      System.out.print(">");
      String line = in.readLine();
      if(line.equals("") || line.charAt(0) == 'n') { next(); print(); }
      else if(line.charAt(0) == 't') { top(); print(); }
      else if(line.charAt(0) == 'p') { print(); }
      else if(line.charAt(0) == 'i') { insert(line.substring(1)); }
      else if(line.charAt(0) == 'd') { delete(); print(); }
      else if(line.charAt(0) == 'q') { break; }
      else                           { System.out.println("?"); }
    }
  }
```

あとの main() 本体では、入力から 1 行ずつ読み、先頭の文字 (コマンド) に応じて適切な手続きを呼ぶだけ。コマンドが「q」なら終了。実際の操作は各手続きが行うのでそれらを見てみる。

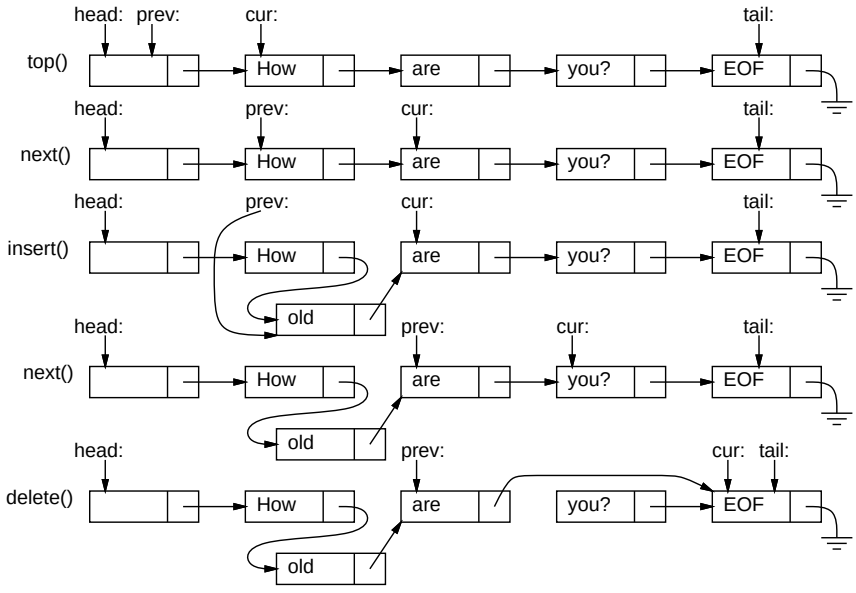

図 8: エディタの各手続きによる操作

```
  static void print() { System.out.println("  " + cur.str); }
  static void top() { prev = head; cur = head.next; }
  static void next() { if(cur != tail) { prev = cur; cur = cur.next; } }
  static void insert(String s) {
    prev.next = new Cell(s); prev = prev.next; prev.next = cur;
  }
  static void delete() {
    if(cur != tail) { prev.next = cur.next; cur = prev.next; }
  }
```

print() は現在行の文字列を表示する。top() は prev と cur を先頭に設定する。next() 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。最後の insert() と delete が構造を変更するものだがこれは図 8 と見比べて参照のつけ替えを理解して欲しい。最後にクラス (レコード)Cell は先に出て来た通り。

```
    static class Cell {
      public String str; Cell next;
      public Cell(String s) { str = s; }
    }
  }
```

演習 5 エディタのプログラムを打ち込んで動かせ。動いたら次のようなコマンドを追加してみよ。

(a) 全部の行を一気に表示するコマンド。または、現在行の前後数行を表示するコマンド (この場合表示行数を指定できるとなおよい)。

(b) 1 つ前の行に戻るコマンド (単リストだとちょっと面倒!)。

(c) 現在の行と次の行を「前後入れ替える」コマンド。

(d) 現在行の文字列 α を文字列 β に置き換えるコマンド。書き方はたとえば「s/α/β/」などのようにする。

(e) 編集内容をファイルから読み込むコマンド、編集内容をファイルに書き出すコマンド。

ファイルの読み書きについては、授業では全く何も説明していないので、自力で調べてやってみようと思う人だけにしてください。どうしてもやりたい場合は次のことをヒントに (API ドキュメントも参照):

- ファイル読み書きする手続きは main() と同様「throws Exception」を指定する (別の方法もあるけど…)。
- 読む時は「new BufferedReader(new FileReader(文字列))」で指定した名前のファイルから読む BufferedReadeer が作れる。書く時は「new PrintWriter(new FileWrite(文字列))」で指定した名前のファイルに書く PrintWriter (println() が使えるオブジェクト) が作れる。どちらも、使い終わったら close() で後始末する。
- 読む時に「ファイルが終わった」ことは、readLine() が null を返すことで分かる。

3.3 表と探索

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。そのコンストラクタ/メソッドとして次のものが使えるようにする。

- IntStrTable(int サイズ) — 最大サイズ個の要素が登録できるような表を初期化するコンストラクタ。
- void put(int 鍵, String 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。
- String get(int 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 null 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。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。main() は引数で調べたサイズのテーブルを作り、乱数でデータを生成して put()/get() して時間を計測する。get() の回数が多いのは、多くの表ではデータを更新するよりも検索する頻度の方が多いためこうしてみた。

```
public class R6Sample3 {
  public static void main(String[] args) throws Exception {
    int size = (new Integer(args[0])).intValue();
    IntStrTable tbl = new IntStrTable(size);
    long t1 = System.currentTimeMillis();
    for(int i = 0; i < size; ++i) {
      int k = (int)(Math.random()*size*10000); tbl.put(k, ""+k);
    }
    for(int i = 0; i < size*10; ++i) {
      int k = (int)(Math.random()*size); String s = tbl.get(k);
```

```
    }
    long t2 = System.currentTimeMillis();
    System.out.println("time = " + (t2-t1));
  }
```

表の本体は次のようになる。関数 `find()` が `put()`、`get()` どちらの場合でも下請けとして呼ばれ、指定した鍵が何番目の項目にあるかを返す (まだその鍵が登録されていなければ-1 を返す)。

```
  static class IntStrTable {
    Pair[] a;
    int size, count;
    public IntStrTable(int s) { a = new Pair[s]; size = s; count = 0; }
    private int find(int k) {
      for(int i = 0; i < count; ++i) { if(a[i].key == k) { return i; } }
      return -1;
    }
    public void put(int k, String s) {
      int i = find(k);
      if(i < 0) { a[count] = new Pair(k, s); ++count; }
      else      { a[i].str = s; }
    }
    public String get(int k) {
      int i = find(k);
      if(i < 0) { return null; } else { return a[i].str; }
    }
  }
  static class Pair {
    public int key;
    public String str;
    public Pair(int k, String s) { key = k; str = s; }
  }
}
```

この `find()` のように「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方法を考えて実装し、どれくらい改善されたか調べよ。

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。しかし鍵の範囲は (上のプログラムでは) サイズの 10,000 倍あるので、そんな巨大な配列を作るのは無理そう。そこで、その鍵の範囲を適当な計算式で「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。

3.4 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 9)。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)。

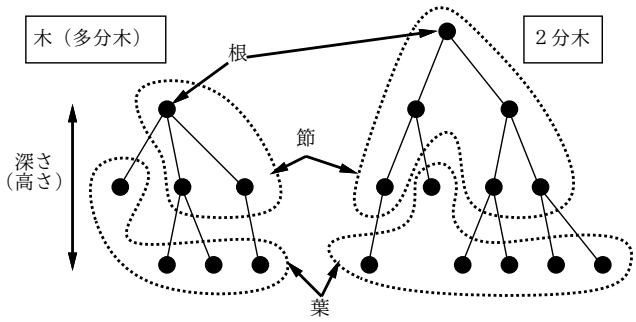

図 9: 木とその用語

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 10 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。
のが吉。

3.5 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 9)。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)。

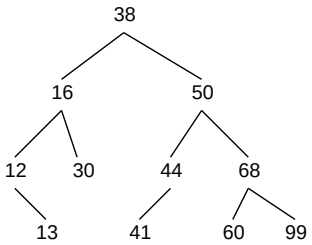

図 10: 2 分探索木の例

すでに紙面一杯なのであとはコードを示すだけにする (`main()` は先の例題と同じなので表とノードのクラスだけ)。

```
static class IntStrTable {
  Node root = null;
  public IntStrTable(int s) { /* do nothing */ }
  public void put(int k, String s) {
    if(root == null) { root = new Node(k, s); } else { put1(root, k, s); }
  }
  private void put1(Node n, int k, String s) {
    if(n.key == k) {
      n.str = s;
```

```
    } else if(n.key > k) {
      if(n.left==null) { n.left = new Node(k,s); } else { put1(n.left,k,s); }
    } else { /* n.key < k */
      if(n.right==null) { n.right = new Node(k,s); } else {put1(n.right,k,s);}
    }
  }
  public String get(int k) { return get1(root, k); }
  private String get1(Node n, int k) {
    if(n.key == k) {
      return n.str;
    } else if(n.key > k) {
      if(n.left == null) { return null; } else { return get1(n.left, k); }
    } else { /* n.key < k */
      if(n.right == null) { return null; } else { return get1(n.right, k); }
    }
  }
}
static class Node {
  public int key;
  public String str;
  public Node left, right;
  public Node(int k, String s) { str = s; key = k; left = right = null; }
}
```

演習 7 この例題を打ち込んで動かし、先の線形探索と所要時間比較を行い、また時間計算量の分析を行いなさい。自分なりの改良版 (演習 6) も含まれているとなおよい。

A　本日の課題 6A

「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 6A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 「演習 2」で動かしたプログラムどれか 1 つのソース。
4. 以下のアンケートの回答。

   Q1. オートマトンについて学びましたが、納得しましたか。

   Q2. 動的データ構造 (再帰的データ構造) については納得しましたか。

   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 6B

次回までの課題は「演習 3」〜「演習 7」の (小) 課題から 2 つ以上選んで報告することです。ただし「演習 3」「演習 4」からは最大 1 個。実力相応に選んでください。あまり無理はしないで結構ですので。

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 6B」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ましたか。

   Q2. 表と検索についてはどうでしたか。

   Q3. 課題に対する感想と今後の要望をお書きください。