

3 検索 (続)

均衡の取れた木構造

今回は木構造が検索に向いてことを見たが、そのために高さを抑えた整列された木構造を使わなければならない。

まずは整列されたリストを元に木構造を作ろう。

```
let rec build_tree n l =
  if n = 0 then Empty, l else
  let t1, l = build_tree ((n-1)/2) l in
  match l with
  [] -> t1, []
| a :: l ->
  let t2, l = build_tree (n/2) l in
  Node (t1, a, t2), l

let build_tree l = fst (build_tree (List.length l) l)
```

この関数で作られた木では、全ての Empty の深さが $\log n$ または $\log n + 1$ である。これは最適に抑えられた高さである。

しかし、途中で木に新しい要素を入れようとすると、また最初から木を作り直さないといけない。

最適の高さではなく、「均衡の取れた」(balanced) な木に条件を緩和すると、木を部分的に変えるだけで新しい要素を追加できる。均衡の取れた木とは、全ての節では左の子の高さのと右の子の高さの差が 1 以下に抑えられていることをいう。

実はその条件が満たされると木の高さが $\log n$ に比例することが証明できる。これを確認するために、最悪の場合を考えればいい。常に左の子が右の子より高ければ最悪になる。そのときに高さ h の木に入る要素の数 $N(h)$ を計算しよう。

$$\begin{aligned} N(0) &= 0 \\ N(1) &= 1 \\ N(2) &= 2 \\ N(h+2) &= 1 + N(h+1) + N(h) \end{aligned}$$

$M(h) = N(h) + 1$ と置くと、

$$\begin{aligned} M(0) &= 1 \\ M(1) &= 2 \\ M(2) &= 3 \\ M(h+2) &= M(h+1) + M(h) \end{aligned}$$

これは 1 ずらした Fibonacci 数列なので、等比数列で近似すると、

$$M(h) \sim \left(\frac{1 + \sqrt{5}}{2} \right)^h$$

そうすると、 $N(h) \geq k^h - 2$ なので、高さは $\log_k n = \frac{\log n}{\log k}$ で抑えられ、全体的には $O(\log n)$ になる。

実際に追加の操作を定義するために、節の情報にその節の高さを追加しないといけない。さらに、値の集合だけではなく、検索のために値から異なる値への写像 (辞書など) も定義したいので、

節に入れる値を新しいデータ型として定義する。ただし、今度は再帰を使わずに中身のデータを定義するだけでいいので、レコードを使う。

```
type ('a,'b) node = {key:'a; data:'b; height:int}          (* レコードの定義 *)
```

```
# {key="garrigue"; data="Ri1-415"; height=0};;
- : (string, string) node = {key = "garrigue"; height = 0; data = "Ri1-415"}
```

```
let height t =
  match t with
  | Empty -> 0
  | Node (_, d, _) -> d.height          (* レコードの height を取り出す *)
val height : ('a, 'b) node tree -> int
```

```
let rec search x t =
  match t with
  | Empty -> raise Not_found          (* 見付からない *)
  | Node (t1, d, t2) ->
    if d.key = x then d.data else
    if x < d.key then search x t1 else search x t2
val search : 'a -> ('a, 'b) node tree -> 'b
```

```
let node t1 d t2 =
  Node (t1, {d with height = 1 + max (height t1) (height t2)}, t2)
          (* with は中身の一部を変えた新しいレコードを返す *)
```

```
let rec bal t1 d t2 =
  let h1 = height t1 and h2 = height t2 in
  if h1 > h2 + 1 then
    match t1 with
    | Empty -> assert false          (* ありえない *)
    | Node (l1, d1, r1) ->
      match r1 with
      | Node (lr1, dr1, rr1) when dr1.height > height l1 ->
        node (node l1 d1 lr1) dr1 (node rr1 d t2)
      | _ -> node l1 d1 (node r1 d t2)
    else if h2 > h1 + 1 then
      match t2 with
      | Empty -> assert false
      | Node (l2, d2, r2) ->
        match l2 with
        | Node (ll2, dl2, rl2) when dl2.height > height r2 ->
          node (node t1 d ll2) dl2 (node rl2 d2 r2)
        | _ -> node (node t1 d l2) d2 r2
    else
      node t1 d t2
```

```
let rec insert d t =
  match t with
  | Empty -> node Empty d Empty
  | Node (t1, d1, t2) ->
    if d.key < d1.key then
      bal (insert d t1) d1 t2
    else if d.key > d1.key then
```

```
    bal t1 d1 (insert d t2)
  else node t1 d t2
val insert : ('a, 'b) node -> ('a, 'b) node tree -> ('a, 'b) node tree
```

bal の実行時間は木の高さによらないので, insert は $O(\log n)$ でできるわけだ.

練習問題 3.1 1. 平衡二分木から値を除く関数を書きなさい.

即ち let $t' = \text{remove } k \ t$ をすると, t' の中では k が含まれないような

```
remove : 'a -> ('a, 'b) node tree -> ('a, 'b) node tree
```

2. 対のリストから木を作る関数および木から対のリストを作る関数を定義せよ.

```
tree_of_list : ('a * 'b) list -> ('a, 'b) node tree
```

```
list_of_tree : ('a, 'b) node tree -> ('a * 'b) list
```

list_of_tree を正しく作れば, 返されたリストが整列されている. この性質を使えば, 両方の関数を組み合わせてリストを整列する関数ができる. 計算量はどうなる?