Jacques Garrigue, 2012 年 11 月 7 日

10 探索アルゴリズムと木構造

線形探索

```
let rec assoc (x : 'a) (l : ('a * 'b) list) =
  match l with
    [] -> raise Not_found
  | (a,b)::l ->
      if x = a then b else assoc x l
val assoc : 'a -> ('a * 'b) list -> 'b = <fun>
# assoc 2 [3,"c"; 1,"a"; 2,"b"];;
- : string = "b"
```

長所： 要素が簡単に追加できる $O(1)$
短所： 探索が遅い $O(n)$

例外　上の **raise** は例外を起こす関数である．例外は exn 型の値であり，様々な種類がある．Not_found はその一例である．

二分探索

```
let search (x : 'a) (arr : ('a * 'b) array) =
  let rec search first last =
    if first = last then
      let (a,b) = arr.(first) in
      if x = a then b else raise Not_found
    else
      let middle = (first + last) / 2 in
      let (a,_) = arr.(middle) in
      if x <= a then search first middle
      else search (middle+1) last
  in search 0 (Array.length arr - 1)
val search : 'a -> ('a * 'b) array -> 'b = <fun>
# search 3 [|1,"a"; 2,"b"; 3,"c"; 4,"d"|] ;;
- : string = "c"
```

長所： 探索が早い $O(\log n)$
短所： 追加の度に整列しなければならない $O(n)$

木構造

リストと同様に，木構造が定義できる．

```
# type 'a tree = Empty | Node of 'a tree * 'a * 'a tree ;;
# let rec depth t =
    match t with
      Empty -> 0
    | Node (t1, x, t2) -> 1 + max (depth t1) (depth t2)
  ;;
val depth : 'a tree -> int
# depth (Node (Node (Empty, 1, Empty), 2, Empty)) ;;
- : int = 2
```

二分木探索

上に定義した木構造は効率のよい検索に向いている．そのために各節を見るだけで探しているものがどちらの子にあるかを知らなければなない．順序付けされた木では，左の子の要素が全て節の値より小さく，右の子の要素が全て節の値より大きい．

```
let rec search x t =
  match t with
    Empty -> raise Not_found
  | Node (left, (key, data), right) ->
      if x = key then data else
      if x < key then search x left else search x right
val search : 'a -> ('a*'b) tree -> 'b = <fun>

let rec add x d t =
  match t with
    Empty -> Node (Empty, (x, d), Empty)
  | Node (left, (key, data as kd), right) ->
      if x < key then Node (add x d left, kd, right) else
      if x > key then Node (left, kd, add x d right) else
      Node (left, (key, d), right)
val add : 'a -> 'b -> ('a*'b) tree -> ('a*'b) tree = <fun>
```

長所： バランスが取れていれば，追加も検索も早い $O(\log n)$
短所： バランスがくずれると遅くなる 最悪 $O(n)$

検索の効率をよくするために，木の深さを最小に抑えないといけない．深さ n の木には 2^n 要素が入る．

練習問題 9.1 整列されたリストを元に，深さ $\log_2 n$ の整列された木を作る関数を書きなさい．ヒント：`Array.of_list` で配列に変換してから、二分探索のアルゴリズムを使って作ればいい．

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．

均衡の取れた木構造

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でいいので，レコードろ使う．

```
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                                    (* 条件付きパターン *)
          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 9.2

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？