Correction : dictionnaires, arbres binaires de recherche et équilibrage

1 Introduction

2 Implémentation élémentaire

1. On extrait simplement le bon élément de la structure s'il s'agit d'un `Node` :

```
let taille = function
  | Empty -> 0
  | Node (_, _, _, t, _) -> t
```

2. Pour la plus petite clé, on descend le plus à gauche possible :

```
let rec mini_cle = function
  | Empty -> failwith "Dictionnaire vide"
  | Node (Empty, c, _, _, _) -> c
  | Node (fg, _, _, _, _) -> mini_cle fg
```

La complexité correspond à la profondeur de la clé obtenue, elle est majorée par la taille de l'arbre.

3. Pour la plus petite valeur, il n'y a pas d'autre solution que d'explorer tout l'arbre :

```
let rec mini_val = function
  | Empty -> failwith "Dictionnaire vide"
  | Node (Empty, _, v, _, Empty) -> v
  | Node (fg, _, v, _, Empty) -> min (mini_val fg) v
  | Node (Empty, _, v, _, fd) -> min (mini_val fd) v
  | Node (fg, _, v, _, fd) -> min (min (mini_val fg) (mini_val fd)) v
```

La complexité est proportionnelle à la taille de l'arbre (nombre de couples mémorisés).

4. On applique la recherche classique d'un arbre binaire de recherche :

```
let rec membre dict cle = match dict with
  | Empty -> false
  | Node (_, c, _, _, _) when c = cle -> true
  | Node (_, c, _, _, fd) when c < cle -> membre fd cle
  | Node (fg, _, _, _, _) -> membre fg cle
```

5. Même chose, mais on retourne la valeur associée à la clé si on la trouve :

```
let rec valeur dict cle = match dict with
  | Empty -> failwith "Clé introuvable"
  | Node (_, c, v, _, _) when c = cle -> v
  | Node (_, c, _, _, fd) when c < cle -> valeur fd cle
  | Node (fg, _, _, _, _) -> valeur fg cle
```

6. Plusieurs approches possibles, car il faut savoir, lorsque l'on reconstruit les arbres après l'ajout, si la taille a augmenté ou non. On peut faire en sorte que la fonction récursive effectuant l'ajout retourne, en plus de l'arbre modifié, la différence entre la nouvelle taille et l'ancienne (soit 1 s'il y a eu ajout, et 0 sinon) :

```
let ajoute dict cle valeur =
  let rec ajoute_aux = function (* ('a,'b) dict -> ('a,'b) dict * int *)
    | Empty -> Node (Empty, cle, valeur, 1, Empty), 1
    | Node (fg, c, v, t, fd) when c = cle
        -> Node (fg, c, valeur, t, fd), 0
    | Node (fg, c, v, t, fd) when c < cle
        -> let nouv_fd, diff = ajoute_aux fd
           in Node (fg, c, v, t+diff, nouv_fd), diff
    | Node (fg, c, v, t, fd)
        -> let nouv_fg, diff = ajoute_aux fg
           in Node (nouv_fg, c, v, t+diff, fd), diff
  in fst (ajoute_aux dict)
```

Ou bien on peut déterminer (avec `membre`) s'il s'agira d'un ajout ou d'une substitution, et avoir deux fonctions récursives distinctes pour les deux cas :

```
let ajoute dict cle valeur =

  let rec remplace = function (* La clé EST présente *)
    | Empty -> failwith "Impossible"
    | Node (fg, c, v, t, fd) when c = cle
        -> Node (fg, c, valeur, t, fd)
    | Node (fg, c, v, t, fd) when c < cle
        -> Node (fg, c, v, t, remplace fd)
    | Node (fg, c, v, t, fd)
        -> Node (remplace fg, c, v, t, fd)

  and ajoute_strict = function (* La clé N'EST PAS présente *)
    | Empty -> Node (Empty, cle, valeur, 1, Empty)
    | Node (fg, c, v, t, fd) when c < cle
        -> Node (fg, c, v, t+1, ajoute_strict fd)
    | Node (fg, c, v, t, fd)
        -> Node (ajoute_strict fg, c, v, t+1, fd)

  in (if membre dict cle then remplace else ajoute_strict) dict
```

7. Il s'agit simplement de partir d'un arbre vide (`Empty`) et d'ajouter tous les éléments de la liste un à un, par exemple récursivement :

```
let rec construit = function
  | [] -> Empty
  | (c, v)::q -> ajoute (construit q) c v
```

Ou bien encore avec `List.fold_left` :

```
let construit lst =
  List.fold_left
    (fun dict (cle, valeur) -> ajoute dict cle valeur) Empty lst
```

Dans le pire des cas, on produit un arbre-branche, et la complexité est quadratique ($O(n^2)$) en la longueur n de la liste. Dans le meilleur des cas, on produit un arbre équilibré et la complexité est quasi-linéaire ($O(n\log n)$). En moyenne (sur tout les ordres possibles pour les couples), on peut montrer que la complexité sera également quadratique.

8. Il existe plusieurs approches possibles, mais comme on souhaite que l'élément au milieu de la liste se retrouve en racine et que les deux moitiés soient utilisées pour construire récursivement les deux sous-arbres, on sera souvent amené à diviser une liste en deux éléments. Pour ce faire, on peut écrire une fonction prenant un entier k et une liste et retournant la liste des k premiers éléments et la liste des éléments restants (on supposera k plus petit que la longueur de la liste, dans le cas contraire tous les éléments se retrouveront dans la première liste). Par exemple :

```
let rec split n lst = match lst with
  | [] -> [], []
  | lst when n=0 -> [], lst
  | t::q -> let l1, l2 = split (n-1) q in t::l1, l2
```

Cette fonction écrite, la construction de l'arbre est élémentaire :

```
let rec construit_eq = function
  | [] -> Empty
  | lst -> let n = List.length lst in
           let l1, l2 = split (n / 2) lst in
           let cle, valeur = List.hd l2 in
             Node (construit l1, cle, valeur, n, construit (List.tl l2))
```

La complexité de cette construction est quasi-linéaire ($O(n\log n)$ en la taille n de la liste (la première étape coûte $\Theta(n)$ opérations à cause de l'appel à `split`, puis on recommence récursivement avec deux listes de tailles $\lfloor (n-1)/2 \rfloor$ et $\lceil (n-1)/2 \rceil$).

On peut faire mieux, et comme dans le cas de la dichotomie, c'est facilité en utilisant un tableau plutôt qu'une liste, car l'accès à un élément via son indice est en temps constant. On peut ainsi utiliser une fonction récursive prenant en argument deux entiers i et j et retournant un arbre équilibré contenant les couples aux positions dans le tableau de i à j inclus. Cela donne par exemple :

```
let construit_eq lst =
  let tab = Array.of_list lst in
  let rec construit_aux i j = (* Construit et retourne un arbre
                                 avec les éléments de tab[i..j] *)
    if i>j then Empty
           else let k = (i+j)/2 in
                Node (construit_aux i (k-1), fst tab.(k), snd tab.(k),
                      j-i+1, construit_aux (k+1) j)
  in construit_aux 0 (Array.length tab - 1)
```

On obtient ainsi une complexité linéaire (il y a n appels récursifs, puisque chaque appel crée un nœud de l'arbre, et tous sont de complexité constante, plus la construction initiale du tableau qui est elle aussi linéaire). Dans de telles situations, transformer la liste en tableau est parfaitement acceptable.

9. Un classique également, mais attention à éviter dans la mesure du possible les

concaténations qui ne permettront pas ici d'obtenir une complexité linéaire en la taille de l'arbre. On peut travailler avec une référence et un parcours infixe (privilégiant le fils droit sur le fils gauche car on remplit la liste de droite à gauche) :

```
let vers_liste dict =
  let res = ref [] in
  let rec explore = function
    | Empty -> ()
    | Node (fg, c, v, _, fd) -> explore fd;
                                res := (c, v)::!res;
                                explore fg
  in explore dict;
     !res
```

Ou bien une solution purement récursive, avec un argument supplémentaire contenant les éléments déjà parcourus :

```
let vers_liste dict =
  let rec accumule trouves = function
    | Empty -> trouves
    | Node (fg, c, v, _, fd)
        -> accumule ((c,v)::(accumule trouves fd)) fg
  in accumule [] dict
```

3 Équilibrage

10. Par exemple :

```
let rec est_equil_t = function
  | Empty -> true
  | Node (fg, _, _, t, fd) -> 3 * taille fg < 2 * t
                              && 3 * taille fd < 2 * t
```

11. Question difficile... par rapport à l'ajout précédent, on peut faire en sorte que la fonction récursive retourne, en plus de l'arbre et de la variation de taille, un booléen indiquant si l'ajout a été effectué à une profondeur trop grande. Pour ce faire, il faut ajouter un paramètre supplémentaire à la fonction indiquant la profondeur courante. Lors du retour de l'appel récursif, lorsque le booléen indique qu'un équilibrage sera nécessaire et que la racine du sous-arbre courant est déséquilibrée, on utilise le passage par une liste et la reconstruction précédente pour rééquilibrer l'arbre :

```
let ajoute_be dict cle valeur =
  let n = taille dict + 1 in
  let equilibre dict = construit_eq (vers_liste dict) in
  let rec ajoute_aux h = function
    | Empty -> let a_equilibrer = float_of_int h >=
                                  1.0 +. log (float_of_int n) /. log 1.5
               Node (Empty, cle, valeur, 1, Empty), 1, a_equilibrer
    | Node (fg, c, v, t, fd) when c = cle
        -> Node (fg, c, valeur, t, fd), 0, false
    | Node (fg, c, v, t, fd) when c < cle
        -> let nouv_fd, diff, a_equ = ajoute_aux (h+1) fd in
           let nouv_d = Node (fg, c, v, t+diff, nouv_fd) in
             if a_equ && not (est_equil_t nouv_d)
               then equilibre nouv_d, diff, false
               else nouv_d, diff, true
    | Node (fg, c, v, t, fd)
        -> let nouv_fg, diff, a_equ = ajoute_aux (h+1) fg in
           let nouv_d = Node (nouv_fg, c, v, t+diff, fd) in
             if a_equ && not (est_equil_t nouv_d)
               then equilibre nouv_d, diff, false
               else nouv_d, diff, true
  in let nouv_d, _, _ = ajoute_aux 0 dict in nouv_d
```