Option informatique : Corrigé Succinct du Devoir Surveillé n° 2

Problème 1

Le problème de CCP 2008 traite des arbres bicolores et de leur utilisation pour représenter des ensembles. Il serait très facile de les étendre pour représenter des tableaux associatifs. De manière générale, les fonctions de validation des arbres peuvent s'écrire de plusieurs manières relativement proches, tant qu'elles sont correctes et respectent les contraintes (« parcourir l'arbre une seule fois »).

1. Simple insertion dans une liste.

```
let rec insertion_ens i l = match l with
    | [] -> [i]
    | j::l' -> if i=j then l else j::insertion i l' ;;
```

2. Complexité en $O(E)$ car on doit parcourir la liste en entier exactement une fois.

3. Déletion d'un élément dans une liste.

```
let rec elimination_ens i l = match l with
    | [] -> []
    | j::l' -> if i=j then l' else j::elimination_ens i l' ;;
```

4. Le pire cas se produit quand l'ensemble ne contient pas l'élément, auquel cas il faut effectivement le parcourir en entier ; par exemple `elimination_ens 1 [2;4;10;0]`. Le meilleur cas est lorsque l'élément à éliminer se trouve en tête de la liste, par exemple `elimination_ens 2 [2;4;10;0]`.

5. $|A| = \begin{cases} 0 & \text{si } A = \emptyset \\ 1 + \max |G(a)||D(a)| & \text{sinon} \end{cases}$

6. Un arbre totalement déséquilibré, par exemple dont chaque nœud a l'arbre vide pour fils gauche, a une profondeur maximale pour son nombre de nœuds (correspond à une structure de liste). L'arbre de profondeur minimale est un arbre binaire complet dont toutes les feuilles sont à la même profondeur et tous les nœuds ont 0 ou 2 fils non vides.

Dans le premier cas, avec n le nombre de nœuds de l'arbre, $|A| = n$. Dans le second, il existe k tel quel $n = 2^k - 1$ et $|A| = k$, par récurrence sur la profondeur de l'arbre. En effet, si on a deux arbre A_1 et A_2 de profondeur k, Noeud(A_1, i, A_2) est un arbre binaire complet de profondeur $k+1$ et avec $1 + 2 \cdot (2^k - 1)$ nœuds (par hypothèse), c'est-à-dire $2^{k+1} - 1$ nœuds.

7.

— $R(A) \leq |A|$ est trivial car la branche de profondeur maximale contient exactement $R(A)$ nœuds gris, et possiblement des nœuds blancs, donc est de hauteur au moins $R(A)$.

— La branche la plus longue possible alterne nœuds blancs et $R(A)$ nœuds gris (deux blancs successif impossible par P2 ; maximalité car on ne peut y insérer aucun nœud sans casser un invariant). La racine est blanche et la feuille aussi, elle contient donc $R(A)+1$ nœuds blancs et donc est de hauteur $2R(A)+1$, ce qui borne $|A| \leq 2R(A)+1$.

— le plus petit arbre de rang fixe $R(A) = k$ ne contient que des nœuds gris. Il forme un arbre binaire complet, dont le nombre de nœuds est $2^k - 1$ par une induction facile (cours de spé, question précédente). Tout arbre de rang k est obtenu en intercalant des nœuds blancs dans celui-ci, donc $2^{R(A)} - 1 \leq n$ pour tout arbre.

8. On a $n \leq 2^{|A|} - 1$ (cours), donc $\log_2(n) \leq \log_2(2^{|A|} - 1) \leq |A|$.
Pour l'autre inégalité, raisonnons par induction sur la structure de l'arbre. Pour un arbre à deux élément, $|A| = 2 \leq 2E(\log_2(n)) = 2$. Pour l'hérédité, considérons que la propriété est vraie pour les sous-arbres g et d de $N(,g,x,d)$. L'autre inégalité est fausse : si on prend un arbre équilibré à trois éléments, sa hauteur est 2 et $E(\log_2(n)) = E(\log_2(3)$
9.

```
let rec rang a = match a with
    | Vide -> 0
    | Noeud (Gris, g, _, d) -> 1 + rang g
    | Noeud (Blanc, g, _, d) -> rang g ;;
```

10. La fonction utilise une fonction auxiliaire `aux`, prenant en paramètre un sous-arbre et un booléen qui vaut `true` ssi le parent est blanc, et qui renvoie `Some d` si l'arbre est valide et de profondeur d.

```
let pas_blanc = function
    | Noeud (Blanc, _, _, _) -> false
    | _ -> true ;;

let validation_bicolore a =
    let rec aux a = match a with
    | Vide -> true, 0
    | Noeud (c, g, _, d) ->
        let ok_g, ok_d = aux g, aux d in
        match c with
        | Gris ->
            ok_g && ok_d && rg_g = rg_d, rg_g + 1
        | Blanc ->
            ok_g && ok_d && rg_g = rg_d && pas_blanc g && pas_blanc d, rg_g
    in
    fst (aux a) ;;
```

11. Cette fonction est linéaire, de complexité $O(n)$ avec n le nombre de nœuds. En effet, elle examine chaque nœud de l'arbre au plus une fois.

12. On utilise une fonction auxiliaire qui prend en paramètre supplémentaire un maximum et un minimum pour les éléments de l'arbre [1].

```
let validation_abr a =
    let rec aux min max a = match a with
    | Vide -> true
    | Noeud (_, g, x, d) ->
        min < x
        && x < max
        && aux x max d
        && aux min x g
    in
    aux min_int max_int a ;;
```

On peut aussi écrire une version plus générale, qui fonctionnera aussi sur les types qui n'ont pas de borne supérieure (par exemple `string`). Pour cela on peut envelopper les bornes dans des options, ou passer des fonctions. C'est cette dernière possibilité que l'on va utiliser : `l` et `h` (« low » et « high ») valident respectivement une propriété de borne inférieure et supérieure sur le sous-arbre.

```
let validation_abr a =
    let rec aux l h a = match a with
    | Vide -> true
    | Noeud (_, g, x, d) ->
        l x
        && h x
        && aux l (fun y -> y<x) g
        && aux (fun y -> x<y) h d
    in
    aux (fun _ -> true) (fun _ -> true) a ;;
```

13. On insère dans une feuille bien choisie.

```
let rec insertion_abr v a = match a with
    | Vide -> Noeud (Blanc, Vide, v, Vide)
    | Noeud (c, g, y, d) ->
        if x<y
            then Noeud (c, insertion_abr v g, y, d)
        else if x=y
            then a
            else Noeud (c, g, y, insertion_abr v d) ;;
```

14. La complexité est en $O(|A|) = O(\log_2(n))$ (question 8) car il faut au pire parcourir la branche la plus longue pour y ajouter une feuille. Dans le cas général, la profondeur peut être égale au nombre de nœuds (liste) et on retombe dans le cas $O(n)$.

1. dans un cadre plus général, on envelopperait ces deux bornes dans des options car à la racine il n'y a aucune contrainte.

15. Seule P2 peut être invalidée, si la feuille blanche est insérée immédiatement sous un nœud blanc.
16. Il suffit de constater que le résultat de la correction ajoute un nœud gris à chaque chemin menant à F_1, F_2, F_3 et F_4, respectant P3, et donc donne un arbre de rang $n+1$. Par ailleurs P2 est valide sur le résultat.
17. -
18. Si on part de la feuille nouvellement créée, il n'existe à tout instant qu'au plus une correction blanche possible. La paire de nœuds blancs concernés, si elle existe, se rapproche de la racine à chaque correction et ne peut donc que terminer. Le sujet, toutefois, oublie une opération cruciale qui consiste, si on remonte jusqu'à la racine et qu'elle est blanche (avec un fils blanc aussi), à la colorer en gris (augmentant ainsi le rang et re-validant P2). Avec cette réserve, l'arbre sous la correction blanche est toujours valide et donc le résultat est un arbre bicolore valide.
19. En reprenant la question précédente, on voit que la correction ne peut s'appliquer qu'au plus une fois par nœud blanc sur la branche (sans compter la racine). Le nombre de nœuds blancs qui ne sont pas la racine est $|A| - R(A)$, d'où le nombre de corrections maximal à appliquer.
20. filtrage !

```
let correction_blanche a = match a with
    | Noeud (Gris, Noeud (Blanc, Noeud (Blanc, f1, v1, f2), v2, f3), v3, f4)
    | Noeud (Gris, Noeud (Blanc, f1, v1, Noeud (Blanc, f2, v2, f3)), v3, f4)
    | Noeud (Gris, f1,v1, Noeud(Blanc,f2,v2, Noeud(Blanc,f3,v3,f4)))
    | Noeud (Gris, f1,v2, Noeud(Blanc,Noeud(Blanc,f2,v2,f3),v3,f4)) ->
        Noeud (Blanc, Noeud(Gris,f1,v1,f2), v2, Noeud(Gris,f3,v3,f4))
    | _ -> a ;;
```

21. et 22. On va insérer de la même manière, mais en reconstruisant l'arbre, chaque construction de nœud sera suivie d'une étape de correction blanche. Ainsi, on garantit que l'arbre obtenu est valide. On remarque qu'il ne suffit pas de composer insertion et correction à la racine.

```
let rec insertion_abr v a = match a with
    | Vide -> Noeud (Blanc, Vide, v, Vide)
    | Noeud (c, g, y, d) ->
        let a' = if x<y
            then Noeud (c, insertion_abr v g, y, d)
            else Noeud (c, g, y, insertion_abr v d)
        in
        correction_blanche a' ;;
```

Problème 2

Dans ce problème (X 2014) on va étudier une structure de file de priorité nommée « Skew Heap ». Attention, les notations changent, et $|t|$ est maintenant la *taille* de l'arbre t.

1. Fonction facile de par la structure des arbres.

```
let minimum t = match t with
    | E -> failwith "impossible"
    | N(g, x, d) -> x ;;
```

2. Encore de la validation. Attention à ne pas parcourir tout l'arbre à chaque étape si on veut une complexité linéaire.

```
let rec est_un_arbre_croissant t = match t with
    | E -> true
    | N (g, x, d) ->
        est_un_arbre_croissant g &&
        est_un_arbre_croissant d &&
        (match g with
            | E -> true
            | N (_, y, _) -> x <= y
        ) &&
        (match d with
            | E -> true
            | N (_, y, _) -> x <= y
        );;
```

3. Par récurrence sur n. Un arbre binaire avec n nœuds possède $n+1$ "emplacements" vides sous les feuilles (preuve : chaque nœud fournit 2 emplacements et en consomme un (qui le relie à son parent), sauf la racine. On a donc $2n-(n-1) = n+1$ emplacements). Donc si, par récurrence, on a $n!$ arbres croissants à étiquettes dans $1, \ldots, n$, on ne peut ajouter un nœud étiqueté $n+1$ que sous les feuilles, donc de $n+1$ manières différentes pour chacun des $n!$ arbres possibles. Cela fait donc $(n+1) \cdot n! = (n+1)!$ arbres.

4. -

5. D'abord, l'arbre obtenu par fusion est croissant, par induction sur la somme des tailles des arbres. Dans le cas $x_1 < x_2$, x_1 est inférieur aux éléments de g_1 et d_1 par hypothèse, mais aussi aux éléments de d_2 et g_2 car les arbres sont croissants. On fusionne d_1 avec $N(g_2, x_2, d_2)$ (par induction le résultat est croissant et tous ses éléments sont $\geq x_1$) pour obtenir d_1' et on construit $N(d_1', x_1, g_1)$ qui est croissant. Un raisonnement symétrique s'applique dans le cas $x_1 \geq x_2$.

6. On fusionne l'arbre avec pour seul élément x avec t :

```
let ajoute x t = fusion (N (E, x, E)) t;;
```

7. On supprime l'élément à la racine ainsi :

```
let supprime_minimum t = match t with
    | E -> failwith "impossible"
    | N(g,x,d) -> fusion g d ;;
```

8.

```
let ajouts_successifs v =
    let t = ref E in
    for i = 0 to vect_length v - 1 do
        t := fusion !t (N (E, v.(i), E));
    done;
    !t ;;
```

9. Si la suite est décroissante, le dernier cas de fusion s'applique toujours, et ajoute l'élément x_i à la racine de t_i (la fusion avec d_2 est l'identité car $d_2 = E$) en inversant les fils gauche et droit. Par conséquent la hauteur de l'arbre augmente à chaque ajout (on obtient un « zig-zag »). La suite de longueur n avec $x_i = n - i$ convient donc.

10. Ici la séquence est croissante, le premier cas de fusion non trivial s'applique donc toujours. On va prouver que l'arbre est de profondeur $\lceil \log_2(n) \rceil + 1$ pour $n > 0$, en remarquant que la structure de l'arbre t_i correspond à la décomposition des valeurs $x_1, \ldots, x_i$ en binaire (un sous-arbre, gauche ou droit selon la parité de i, contient les valeurs paires, et l'autres les valeurs impaires ; dans chacun de ces sous-arbres un fils contient les valeurs divisibles par 4 et l'autre celles congrues à 1 modulo 4, etc.).