Corrigé DM1

MPSI Lycée Pierre de Fermat

Q1. **dict** est un constructeur, **ajout** et **supprime** sont des transformateurs, **contient** et **recherche** sont des accesseurs.

Q2. A_1 n'est pas un ABR valide car il y a un R dans le sous-arbre gauche du noeud I. A_2 est bien un ABR.

Q3. On descend récursivement dans l'arbre tant qu'il y a un sous-arbre gauche à explorer.

```
let rec min_abr (a: 'a abr) : 'a =
  match a with
  | V -> failwith "Arbre vide"
  | N (x, g, d) -> if g = V then x else min_abr g
```

Q4. La fonction se rappelle récursivement sur un arbre de hauteur strictement inférieure, et les opérations faites lors d'un appel donné sont en temps constant. La complexité $C(h)$ en fonction de la hauteur h vérifie ainsi $C(h) = C(h-1) + \mathcal{O}(1)$, soit $C(h) = \mathcal{O}(h)$.

Q5. Pour $n \in \mathbb{N}$, notons D_n le temps d'exécution de `est_abr` sur l'arbre P_n. $D_n = D_{n-1} + \mathcal{O}(n)$ car pour exécuter `est_abr` sur P_n, il faut:

- se rappeler à gauche, sur P_{n-1}, et à droite sur $\mathbf{V}$, ce qui prend un temps $D_{n-1} + \mathcal{O}(1)$;
- appeler les fonctions `min_abr` et `max_abr` sur les sous-arbres gauche et droit, pour un coût total de $\mathcal{O}(n) + \mathcal{O}(1) = \mathcal{O}(n)$.

En déroulant, on a $D_n = \mathcal{O}(n^2)$.

Q6. Montrons par induction structurelle que pour tout arbre binaire A, on a $P(A)$: "A est un ABR si et seulement si le parcours infixe de A est strictement croissant."

- Montrons $P(\mathbf{V})$. $\mathbf{V}$ est un ABR, et son parcours infixe est [], qui est croissant, donc l'équivalence est vérifiée.
- Soient G, D deux arbres binaires, x une étiquette. On suppose $P(G)$ et $P(D)$, et on veut montrer que $P(\mathbf{N}(x, G, D)$ est vraie. Notons L le parcours infixe de A, L_G, L_D les parcours infixes de G et D. Alors $L = L_G + [x] + L_D$, donc L est croissant si, et seulement si, L_G et L_D sont croissants strict et $\forall x_G \in L_G, \forall x_D \in L_D, x_G < x < x_D$. Donc, par HI, c'est le cas si et seulement si G et D sont des ABR et $\forall x_G \in L_G, \forall x_D \in L_D, x_G < x < x_D$, ce qui est vrai si et seulement si A est un ABR, par définition.

On en déduit une manière plus efficace de tester si un arbre est un ABR: on calcule son parcours infixe, puis on regarde s'il est trié dans l'ordre croissant. Nous avons vu en TP comment calculer des parcours préfixes / infixes / suffixes récursivement en $\mathcal{O}(n)$, et dans le TP sur OCaml impératif une autre version utilisant une pile. Voici une troisième alternative, où l'on fait un parcours récursif en ajoutant les éléments croisés dans une référence, dans l'ordre infixe:

```
let infixe (a: 'a abr) : 'a list =
  let res = ref [] in
  (* ajoute les étiquettes de a dans res, dans l'ordre infixe *)
  let rec infixe_add (a: 'a abr) : unit =
    match a with
    | V -> ()
    | N(x, g, d) -> begin
      infixe_add g;
      res := x :: !res;
      infixe_add d
    end
  in
  infixe a;
  !res

let rec est_triee l =
  match l with
  | [] -> true
  | [x] -> true
  | x :: y :: q -> x < y && est_triee (y :: q)

let est_abr a =
  est_triee (infixe a)
```

Q7. `dict_contient` descend dans l'arbre soit à gauche soit à droite à chaque étape, donc explorer au total des noeuds formant un chemin de la racine vers une feuille dans le pire des cas, d'où une complexité en $\mathcal{O}(h)$.

Q8. Même idée que `dict_contient`, mais on renvoie la valeur quand on la trouve:

```
let rec dict_recherche (a: ('a, 'b) dict) (k: 'a) : 'b =
  match a with
  | V -> failwith "clé non présente"
  | N((k', v'), g, d) ->
    if k = k' then v
    else if k < k' then dict_recherche g k
    else                dict_recherche d k
```

Q9. Erreur courante dans cette question: une fois que l'on ajoute la clé à gauche ou à droite, il ne faut pas oublier de reformer l'arbre total en remettant la racine et l'autre sous-arbre !

```
let rec dict_ajout (a: ('a, 'b) dict) (k: 'a) (v: 'b) : ('a, 'b) dict =
  match a with
  | V -> N((k, v), V, V)
  | N((k', v'), g, d) ->
    if k = k' then N((k, v), g, d)
    else if k < k' then N((k', v'), dict_recherche g k, d)
    else                N((k', v'), g, dict_recherche d k)
```

Q10. On remplace la racine H par F, et on met le E à la place du trou laissé par le F.

Q11.

```
let rec extraire_max (a: 'a abr) : ('a abr * 'a) =
  match a with
  | V -> failwith "Arbre vide"
  | N (x, g, V) -> (g, x) (* g contient exactement tout a sauf x *)
  | N (x, g, d) ->
    let (d', m) = extraire_max d in
    (N (x, g, d'), m)
```

Q12. On cherche la clé à supprimer comme dans les fonctions précédentes, et une fois qu'on l'a trouvée on utilise `extraire_max` pour la remplacer. Erreur courante: interdit d'appeler `extraire_max` sur un arbre vide, donc si le sous-arbre gauche du noeud à supprimer est vide il faut faire autrement, mais dans ce cas on peut simplement remplacer le noeud par son enfant droit puisqu'il n'y a rien à garder à gauche.

```
let rec supprimer (a: ('a, 'b) dict) (k: 'a) : ('a,'b) dict =
  match a with
  | V -> V
  | N((k', v'), g, d) ->
    if      k < k' then N((k', v'), supprimer g k, d)
    else if k > k' then N((k', v'), g, supprimer d k)
    else if g = V then d
    else let (g', m) = extraire_max g in
         N(m, g', d)
```

Q13. `extraire_max` est en $\mathcal{O}(h)$, pour les même raisons que les fonctions précédentes. Pour `supprimer`, on a une première phase qui localise la clé à supprimer en $\mathcal{O}(h)$, puis on appelle `extraire_max` une seule fois, ce qui est encore en $\mathcal{O}(h)$, le coût total est donc $\mathcal{O}(h)$.