Question 1 –

```
let get_taille: abe -> int = function
  | Vide -> 0
  | N((_,n), _, _) -> n;;
```

Question 2.a –

```
let rec arbre_of_abe: abe -> 'a arbre = function
  | Vide -> Vide
  | N((c,_), g, d) -> N(c, arbre_of_abe g, arbre_of_abe  d);;
```

Question 2.b –

```
let rec abe_of_arbre: 'a arbre -> abe = function
  | Vide -> Vide
  | N(c1, g1, d1) ->
     let g2 = abe_of_arbre g1 in
     let d2 = abe_of_arbre d1 in
     let n = 1 + get_taille g2 + get_taille d2 in
     N((c1,n), g2, d2);;
```

Question 3 – Pour rechercher si une clé c est présente dans un ABR, on procède récursivement :

- Si l'arbre est vide alors la clé n'appartient pas à l'arbre.
- Si c est l'étiquette de la racine alors elle appartient à l'arbre.
- Sinon on a deux cas :
  - Si l'étiquette de la racine est strictement supérieure à c alors on recherche c dans le sous-arbre gauche.
  - Si l'étiquette de la racine est strictement inférieure à c alors on recherche c dans le sous-arbre droit.

Question 4.a –

```
let rec get_profondeur c0 (a: abe) = match a with
  | Vide -> failwith "get_profondeur: étiquette absente de l'arbre"
  | N((c,_), _, _) when c = c0 -> 0
  | N((c,_), g, _) when c0 < c -> 1 + get_profondeur c0 g
  | N(   _ , _, d) -> 1 + get_profondeur c0 d;;
```

Question 4.b – Lors d'un appel à la fonction sur un arbre de hauteur $h \geqslant 0$, il y a au plus un appel récursif sur un arbre de hauteur $h-1$. Toutes les autres opérations s'effectuent en temps constant, donc le temps d'exécution est en $\mathcal{O}\Big(h\Big)$.

Question 5.a – Soient n_G et n_D les tailles du sous-arbre gauche et du sous-arbre droit de x. Par définition de la taille, on a :

$$n = 1 + n_G + n_D$$

De plus, comme l'arbre est $\frac{1}{2}$-équilibré en poids, on a :

$$n_G \leqslant \frac{n}{2} \qquad \text{et} \qquad n_D \leqslant \frac{n}{2}$$

On en déduit que :

$$n_G = n - 1 - n_D \geqslant n - 1 - \frac{n}{2} = \frac{n}{2} - 1 \qquad \text{et de même} \qquad n_D \geqslant \frac{n}{2} - 1$$

Finalement $\frac{n}{2} - 1 \leqslant n_G, n_D \leqslant \frac{n}{2}$, c'est à dire :

- Si n est impair, on a $n_G = n_D = \dfrac{n-1}{2}$.
- Si n est pair, on a $n_G = \dfrac{n}{2}$ et $n_D = \dfrac{n}{2} - 1$ ou inversement.

Question 5.b –

```
(* La fonction aux construit un arbre 1/2-équilibré en poids en utilisant les
   clés tab.(k) avec i <= k <= j *)
let abe_of_array tab: abe =
  let rec aux i j =
    if i > j then Vide else begin
        let m = (i+j+1)/2 in (* ou m = (i+j)/2 *)
        let g = aux i (m-1) in
        let d = aux (m+1) j in
        let n = j-i+1 in
        N((tab.(m),n), g, d)
      end
  in
  aux 0 (Array.length tab - 1);;
```

Question 5.c –

```
(* On utilise un parcours infixe de l'arbre afin d'obtenir un tableau trié
   contenant les clés.
 * La fonction parc_inf stocke le parcours infixe de a dans tab à partir de
   l'indice i. Elle renvoie le premier indice inutilisé après le parcours *)
let rec parc_inf tab i a = match a with
  | Vide -> i
  | N((c,_), g, d) ->
     let i1 = parc_inf tab i g in
     tab.(i1) <- c;
     parc_inf tab (i1+1) d;;
```

```
let array_of_abe (a: abe) = match a with
  | Vide -> [||]
  | N((c0,n0), _, _) ->
     let tab = Array.make n0 c0 in
     let i = parc_inf tab 0 a in
     if i <> n0 then failwith "array_of_abe: Problème";
     tab;;
```

```
let equilibrer a =
  let tab = array_of_abe a in
  abe_of_array tab;;
```

Question 6 – On sait que la hauteur minimale pour un arbre binaire de taille n est $\lfloor \log_2(n) \rfloor$. On va montrer que cette borne est atteinte pour un arbre $\frac{1}{2}$-équilibré en poids.

Soit A un arbre non vide $\frac{1}{2}$-équilibré en poids de taille $n \geqslant 1$ et de hauteur $h \geqslant 0$. Montrons par récurrence sur n que :

$$n \geqslant 2^h.$$

- Si $n = 1$ alors $h = 0$, donc l'inégalité est vérifiée.
- Soit $n > 1$, on suppose la propriété vraie jusqu'au rang $n-1$. Soient n_G et h_G (resp. n_D et h_D) la taille et la hauteur du sous-arbre gauche (resp. droit). On a alors deux cas :
  - Si $h_G \geqslant h_D$ alors $h = h_G + 1$. Par hypothèse de récurrence et en utilisant le fait que l'arbre est $\frac{1}{2}$-équilibré, on a :

$$n \geqslant 2n_G \geqslant 2^{h_G+1} = 2^h$$

  - Si $h_D \geqslant h_G$ on procède de la même façon en utilisant le sous-arbre droit à la place du sous-arbre gauche.

Finalement, pour un arbre non vide $\frac{1}{2}$-équilibré en poids, on a $2^h \leqslant n$ et donc $h \leqslant \lfloor \log_2(n) \rfloor$.

On sait (d'après le cours) que pour un arbre binaire quelconque : $n < 2^{h+1}$. Cette inégalité est vraie même si l'arbre n'est pas $\frac{1}{2}$-équilibré en poids et se montre par récurrence sur la taille de l'arbre de manière similaire à la preuve ci-dessus. Ainsi, tout arbre binaire vérifie $h \geqslant \lfloor \log_2(n) \rfloor$.

En conclusion, un arbre non vide $\frac{1}{2}$-équilibré en poids vérifie $h = \lfloor \log_2(n) \rfloor$ et tout arbre binaire de la même taille est de hauteur $\geqslant h$. Le cas de l'arbre vide est évident.

Question 7.a – On montre la contraposée : si tous les ancêtres de x sont α-équilibrés en poids alors la profondeur de x est inférieure ou égale à $h_\alpha(n)$.
Soit $x_0, x_1, \ldots, x_p$ les ancêtres de x du plus profond au moins profond. Ainsi :

- $x_0 = x$, p est la profondeur de x et x_p est la racine de l'arbre.
- Pour tout i : x_{i+1} est le père de x_i.

Pour tout i, on note n_i la taille du sous-arbre enraciné en x_i. Comme les x_i sont α-équilibrés en poids :

$$n_p \geqslant \frac{1}{\alpha} n_{p-1} \geqslant \left(\frac{1}{\alpha}\right)^2 n_{p-2} \geqslant \ldots \geqslant \left(\frac{1}{\alpha}\right)^p n_0$$

Comme $n_0 \geqslant 1$ et $n_p = n$, on obtient $p \leqslant h_\alpha(n)$.

Question 7.b – On utilise la contraposée de la proposition de la question 7.a. Si tous les nœuds d'un arbre sont α-équilibrés en poids, c'est que tous les nœuds sont à profondeur inférieure ou égale à $h_\alpha(n)$. Ainsi, l'arbre est α-équilibré en hauteur et donc sa hauteur est inférieure ou égale à $h_\alpha(n) = \Theta\Big(\log_2(n)\Big)$.

Question 8.a –

```
(* Il s'agit d'une question de cours *)
let rec ajout_abr c0 (a: abe): abe = match a with
  | Vide -> N((c0,1), Vide, Vide)
  | N((c,n), g, d) when c = c0 -> failwith "ajout_abr: cle déjà présente"
  | N((c,n), g, d) when c0 < c -> N((c,n+1), ajout_abr c0 g, d)
  | N((c,n), g, d) -> N((c,n+1), g, ajout_abr c0 d);;
```

Question 8.b –

```
(* Indique si la racine de l'arbre est équilibrée en poids
 * Hypothèse: l'arbre est non vide *)
let est_equilibre (a: abe) alpha = match a with
  | Vide -> failwith "est_equilibre: arbre vide"
  | N((_,n), g, d) ->
     let n = float_of_int n in
     let ng = float_of_int (get_taille g) in
     let nd = float_of_int (get_taille d) in
     ng <= alpha *. n && nd <= alpha *. n;;
```

```
(* c0 est l'étiquette qui vient d'être insérée dans l'arbre
 * Hypothèse: il existe un noeud bouc-emissaire sur le chemin qui relie la
   racine à c *)
let rec equilibrer_be c0 (a: abe) alpha = match a with
  | Vide -> failwith "equilibrer_be: étiquette absente"
  | a when not (est_equilibre a alpha) -> equilibrer a
  | N((c,_), g, _) when c = c0 ->
     failwith "equilibrer_be: pas de bouc emissaire"
  | N((c,n), g, d) when c0 < c -> N((c,n), equilibrer_be c0 g alpha, d)
  | N((c,n), g, d)             -> N((c,n), g, equilibrer_be c0 d alpha);;
```

Question 8.c –

```
let fct_h alpha n =
  int_of_float (log(float_of_int n)/.log(1./.alpha));;
```

```
let ajout_abe c0 a0 alpha =
  let a1 = ajout_abr c0 a0 in
  let n = get_taille a1 in
  let prof_c0 = get_profondeur c0 a1 in
  if prof_c0 > fct_h alpha n then
    equilibrer_be c0 a1 alpha
  else
    a1;;
```

Question 9.a – On note n la taille de l'arbre, n_G la taille du sous-arbre gauche et n_D la taille du sous-arbre droit. Comme la racine est α-déséquilibrée en poids, on a :

$$n_G > \alpha\, n \qquad \text{ou} \qquad n_D > \alpha\, n.$$

Dans la suite, on traite le cas où $n_G > \alpha\, n$, l'autre cas étant symétrique. On a :

$$\frac{1}{\alpha}\, n_G > n = 1 + n_G + n_D$$

Donc :

$$n_G > \frac{\alpha}{1-\alpha}(1 + n_D)$$

Comme $\alpha \in \left]\frac{1}{2}\,;1\right[$, on obtient :

$$n_G > 1 + n_D$$

Puisque les quantités mises en jeu sont des entiers :

$$n_G \geqslant 2 + n_D$$

Question 9.b – Soit n la taille de A, soit x le nœud à profondeur h dans A et A'' l'arbre obtenu lorsqu'on supprime x de A. L'arbre A'' est de hauteur $h-1$ et de taille $n-1$. D'après la question précédente, le sous-arbre gauche de A'' contient au moins un nœud de plus que le sous-arbre droit (ou inversement). On peut donc ajouter le nœud x dans l'arbre A'' sans en modifier la hauteur. L'arbre obtenu noté A''' est de hauteur $h-1$ et de taille n. L'arbre A' est $\frac{1}{2}$-équilibré en poids et de taille n. Ainsi, d'après la question 6, la hauteur de A' est d'au plus $h-1$.

Question 10 – Il suffit de montrer que si A_1 n'est pas α-équilibré en hauteur alors A_2 est α-équilibré en hauteur. Soient n_0, n_1, n_2 et h_0, h_1, h_2 les tailles et les hauteurs de A_0, A_1, A_2. Les nœuds x et y sont ceux définis par l'énoncé.

Si A_1 n'est pas équilibré en hauteur, alors x est le seul nœud à profondeur h_1 dans A_1. Soit A l'arbre enraciné en y et h sa hauteur. Par hypothèse, la racine de A n'est pas α-équilibrée en poids. De plus, x est le seul nœud à profondeur h dans A. D'après la question 9.b, appliquer la fonction `equilibrer` à A diminue sa hauteur. Par conséquent, $h_2 \leqslant h_1 - 1 = h_0$. Comme A_0 est α-équilibré en hauteur, on obtient :

$$h_2 \leqslant h_0 \leqslant h_\alpha(n_0) = h_\alpha(n_2 - 1) \leqslant h_\alpha(n_2)$$

Donc l'arbre A_2 est α-équilibré en hauteur.

Question 11 – Le nœud x n'étant pas α-équilibré en poids, on peut supposer sans perte de généralité que :

$$n_G > \alpha n.$$

On a donc :

$$n_G - n_D = n_G - (n - n_G - 1) = 2n_G - n + 1 > 2\alpha n - n + 1 = (2\alpha - 1)n + 1.$$

Comme $(2\alpha - 1)n + 1 > 0$, on a :

$$\Delta_A(x) = |n_G - n_D| = n_G - n_D > (2\alpha - 1)n + 1.$$

Question 12.a – Le temps d'exécution des fonctions `ajout_abr` et `get_profondeur` est linéaire en la profondeur de l'arbre. Puisque B_i est α-équilibré, la question 7.b assure que les appels à ces fonctions s'exécutent en temps $\mathcal{O}\Big(\log_2(i)\Big)$. Les autres opérations s'exécutent en temps constant. Donc $T_i = \mathcal{O}\Big(\log_2(i)\Big)$

Étudions la valeur de $\Delta(B_i) - \Delta(B_{i-1})$. Soit x_0 le nœud qui vient d'être ajouté. Ce nœud appartient à B_i, mais pas à B_{i-1}. De plus :

- x_0 est une feuille donc son potentiel dans B_i est nul : $\Delta_{B_i}(x_0) = 0$.
- Si un nœud x ne se trouve pas sur le chemin qui relie la racine de B_i et x_0 alors les sous-arbres de x n'ont pas été modifiés et donc $\Delta_{B_i}(x) = \Delta_{B_{i-1}}(x)$.
- Si un nœud x se trouve sur le chemin entre la racine de B_i et x_0 alors la taille de l'un des sous-arbres de x est modifié d'une unité. Donc $\Delta_{B_i}(x) = \Delta_{B_{i-1}}(x) + \epsilon$ avec $\epsilon \in \{-2; -1; 0; 1; 2\}$.

Étant donné que le chemin qui relie la racine de B_i à x_0 est de taille $\mathcal{O}\Big(\log_2(i)\Big)$, on en déduit que $\Delta(B_i) - \Delta(B_{i-1}) = \mathcal{O}\Big(\log_2(i)\Big)$. En conclusion :

$$C_{i,k_0} = T_i + k_0\Big(\Delta(B_i) - \Delta(B_{i-1})\Big) = \mathcal{O}\Big(\log_2(i)\Big) + k_0 \times \mathcal{O}\Big(\log_2(i)\Big) = \mathcal{O}\Big(\log_2(i)\Big).$$

Question 12.b – Lors de l'insertion, on commence par insérer le nœud dans l'arbre B_{i-1} pour obtenir un arbre B'_{i-1} qu'il faut rééquilibrer. Soit y le nœud de B'_{i-1} désigné comme le bouc-émissaire (on sait que y n'est pas le nœud qui vient d'être ajouté).
Soit D le sous-arbre de B_{i-1} enraciné en y.
Soit D' le sous-arbre de B'_{i-1} enraciné en y et $n' \geqslant 1$ sa taille.
Soit D'' le sous-arbre de B_i obtenu après application de la fonction `equilibrer` sur D'.

Il existe une constante k_2 telle que le temps d'exécution de `equilibrer` est inférieur à $k_2 \times n'$. On a donc :

$$T_i \leqslant R_i + k_2 \times n'$$

où $R_i = \mathcal{O}\Big(\log_2(i)\Big)$ est le temps d'exécution de l'insertion sans compter l'appel à `equilibrer`.

Intéressons nous maintenant à $\Delta(B_i) - \Delta(B_{i-1}) = \Big(\Delta(B_i) - \Delta(B'_{i-1})\Big) + \Big(\Delta(B'_{i-1}) - \Delta(B_{i-1})\Big)$:

- Avec le même raisonnement que dans la question 12.a, on a $\Delta(B'_{i-1}) - \Delta(B_{i-1}) = \mathcal{O}\Big(\log_2(i)\Big)$.
- Puisque D'' est $\frac{1}{2}$-équilibré en poids, la question 5.a assure que $\Delta(D'') = 0$.
- On a $\Delta(D') \geqslant \Delta_{D'}(y) > (2\alpha - 1)n'$ d'après la question 11.
- On a :

$$\Delta(B_i) - \Delta(B'_{i-1}) = \Delta(D'') - \Delta(D') < -(2\alpha - 1)n'.$$

Si on prend $k_0 = \dfrac{k_2}{2\alpha - 1} > 0$, on obtient :

$$\begin{aligned} C_{i,k_0} &= T_i + k_0\Big(\Delta(B_i) - \Delta(B_{i-1})\Big) \\ &\leqslant R_i + k_2 \times n' + k_0\Big(\Delta(B_i) - \Delta(B'_{i-1})\Big) + k_0\Big(\Delta(B'_{i-1}) - \Delta(B_{i-1})\Big) \\ &\leqslant R_i + k_2 \times n' - k_2 \times n' + k_0\Big(\Delta(B'_{i-1}) - \Delta(B_{i-1})\Big) \end{aligned}$$

On conclut en utilisant le fait que $R_i = \mathcal{O}\Big(\log_2(i)\Big)$ et $\Delta(B'_{i-1}) - \Delta(B_{i-1}) = \mathcal{O}\Big(\log_2(i)\Big)$.

Question 12.c – D'après les questions précédentes, il existe deux constantes $k, k_0 > 0$ telle que pour tout $i \in [\![1; n]\!]$:

$$C_{i,k_0} \leqslant k \log_2(i) \leqslant k \log_2(n).$$

En sommant ces inégalités :

$$\sum_{i=1}^{n} C_{i,k_0} \leqslant k \times n \log_2(n)$$

De plus :

$$\begin{aligned} \sum_{i=1}^{n} T_i &= \sum_{i=1}^{n} C_{i,k_0} - k_0(\Delta(B_n) - \Delta(B_0)) \\ &\leqslant k \times n \log_2(n) - k_0(\Delta(B_n) - 0) \\ &\leqslant k \times n \log_2(n) \end{aligned}$$

Comme $\displaystyle\sum_{i=1}^{n} T_i \geqslant 0$, on conclut que $\displaystyle\sum_{i=1}^{n} T_i = \mathcal{O}\Big(n \log_2(n)\Big)$.

Question 13 –

```
let ajout_abe2 c a alpha =
  let a2 = ajout_abe c a.arbre alpha in {
    arbre = a2;
    taille_max = max a.taille_max (get_taille a2)};;
```

Question 14 –

```
(* Fonction faite en cours *)
let rec get_max: abe -> 'a * abe = function
  | Vide -> failwith "get_max: étiquette absente"
  | N((c,n), g, Vide) -> c, g
  | N((c,n), g, d) ->
     let m,d2 = get_max d in
     m, N((c,n-1), g, d2);;
```

```
(* Fonction faite en cours *)
let rec suppression_abr c0 (a: abe): abe = match a with
  | Vide -> failwith "suppression_abr: étiquette absente"
  | N((c,n), g, d) when c0 < c -> N((c,n-1), suppression_abr c0 g, d)
  | N((c,n), g, d) when c0 > c -> N((c,n-1), g, suppression_abr c0 d)
  | N((c,n), Vide, d) -> d
  | N((c,n), g, d) ->
     let m,g2 = get_max g in
     N((m,n-1), g2, d);;
```

```
let suppression_abe2 c a alpha =
  let a2 = suppression_abr c a.arbre in
  let a3 = {arbre = a2;
            taille_max = max a.taille_max (get_taille a2);
           } in
  if float_of_int (get_taille a3.arbre) <
       alpha *. (float_of_int a3.taille_max) then begin
      let a4 = equilibrer a2 in {
          arbre = a4;
          taille_max = get_taille a4;
        }
    end
  else a3;;
```