I Structure mathématique

Exercice 1

On montre ce résultat par induction structurelle (récurrence sur la forme de l'arbre).

- Si l'arbre est vide, alors il est de taille 0 et de hauteur -1. Les deux inégalités sont vérifiées.
- Supposons qu'un arbre soit de la forme $a = N(x, g, d)$, et que la propriété est vraie pour g et d. On a, par définition, $|a| = 1 + |g| + |d|$ et $h(a) = 1 + \max(h(g), h(d))$. On en déduit donc :
  - $|a| \geq 1 + h(g) + 1 + h(d) + 1 \geq 1 + \max(h(g) + 1, h(d) + 1) \geq 1 + \max(h(g), h(d)) = h(a)$
  - $|a| \leq 1 + 2^{h(g)+1} - 1 + 2^{h(d)+1} - 1 \leq 2 \times 2^{\max(h(g),h(d))+1} - 1 = 2^{h(a)+1} - 1$

  On conclut par induction.

Exercice 2

On montre ce résultat par induction structurelle. Remarquons que comme $n_a + f_a = |a|$, la deuxième égalité est prouvée.

- Si l'arbre est une feuille, alors $n_a = 0 = f_a - 1$.
- Si $a = N(x, g, d)$ et que le résultat est vrai pour g et d, alors $f_a = f_g + f_d$ et $n_a = 1 + n_g + n_d$. On a donc $n_a = 1 + f_g - 1 + f_d - 1 = f_a - 1$.

Le résultat est donc toujours vrai.

II Implémentation

Exercice 3

1.
```
let rec taille = function
    | Feuille x -> 1
    | Noeud(x, l) -> 1 + List.fold_left (fun x y -> x + taille y) 0 l;;
```

2.
```
let rec affichage = function
    | Feuille x -> print_int x
    | Noeud(x, l) -> print_int x; List.iter affichage l;;
```

Exercice 4

```
let rec hauteur = function
    | Vide -> -1
    | N(x, g, d) -> 1 + max (hauteur g) (hauteur d);;

let rec feuilles = function
    | Vide -> 0
    | N(x, Vide, Vide) -> 1
    | N(x, g, d) -> feuilles g + feuilles d;;

let rec internes = function
    | Vide -> 0
    | N(x, Vide, Vide) -> 0
    | N(x, g, d) -> 1 + internes g + internes d;;
```

Exercice 5

Le premier algorithme fait de la simple concaténation. Le second utilise une fonction auxiliaire qui prend en argument la liste en cours de remplissage. Comme il est nécessaire de remplir par la fin, on commence à traiter le côté droit d'un nœud interne. La concaténation étant coûteuse, on préfèrera le second algorithme.

```
let rec liste_feuilles = function
    | Vide -> []
    | N(x, Vide, Vide) -> [x]
    | N(x, g, d) -> (liste_feuilles g) @ (liste_feuilles d);;

let liste_feuilles =
    let rec aux liste = function
        | Vide -> liste
        | N(x, Vide, Vide) -> x :: liste
        | N(x, g, d) -> let l = aux liste d in aux l g in
    aux [];;
```

Exercice 6

```
let rec est_entier = function
    | Vide -> true
    | N(x, Vide, Vide) -> true
    | N(x, _, Vide) -> false
    | N(x, Vide, _) -> false
    | N(x, g, d) -> (est_entier g) && (est_entier d);;
```

Exercice 7

On utilise une fonction auxiliaire qui permet de garder en mémoire le maximum courant.

```
let maximum a =
    let rec aux maxi = function
        | Vide -> maxi
        | N(x, g, d) when x > maxi -> let y = aux x g in aux y d
        | N(x, g, d) -> let y = aux maxi g in aux y d in
    match a with
        | Vide -> failwith "Arbre vide"
        | N(x, _, _) -> aux x a;;
```

Exercice 8

Il s'agit ici, à chaque appel récursif, de déterminer (aléatoirement) la taille du sous-arbre gauche et celle du sous arbre droit. La fonction auxiliaire prend deux arguments qui seront le minimum (inclus) et le maximum (non inclus) des valeurs prises par les étiquettes d'un arbre. La valeur `maxi - mini` donne donc la taille de l'arbre considéré. Après avoir déterminé la taille, on en déduit l'étiquette de la racine, et les appels récursifs à relancer.

```
let arbre_alea n =
   let rec aux mini maxi = match maxi - mini with
      | 0 -> Vide
      | k -> let r = Random.int k in
         let g = aux mini (mini + r) and d = aux (mini + r + 1) maxi in
         N(mini + r, g, d) in
   aux 0 n;;
```

III Parcours d'arbres

Exercice 9

```
let rec prefixe = function
   | Vide -> []
   | N(x, g, d) -> x :: (prefixe g) @ (prefixe d);;

let rec infixe = function
   | Vide -> []
   | N(x, g, d) -> (infixe g) @ (x :: (infixe d));;

let rec postfixe = function
   | Vide -> []
   | N(x, g, d) -> (postfixe g) @ (postfixe d) @ [x];;
```

Pour éviter d'utiliser l'opérateur `@`, on utilise un algorithme similaire à celui de la liste des feuilles, en faisant bien attention au sens de parcours.

```
let prefixe =
   let rec aux liste = function
      | Vide -> liste
      | N(x, g, d) -> let l = aux liste d in x :: (aux l g) in
   aux [];;

let infixe =
   let rec aux liste = function
      | Vide -> liste
      | N(x, g, d) -> let l = aux liste d in (aux (x :: l) g) in
   aux [];;

let postfixe =
   let rec aux liste = function
      | Vide -> liste
      | N(x, g, d) -> let l = aux (x :: liste) d in aux l g in
   aux [];;
```

Exercice 10

1. On montre ce résultat par récurrence forte sur la taille des listes, en notant P et I les listes en parcours préfixe et infixe respectivement. On suppose que a est un arbre associé à ces listes.
   - Si les listes sont de taille 1, alors $P = I$ et l'arbre est unique.
   - Supposons le résultat vrai pour des listes de taille $\leq n$ où $n \in \mathbb{N}^*$ fixé. Posons x le premier

élément de P. x est nécessairement la racine de l'arbre. De plus, la position de x dans I permet de déterminer les sommets du sous-arbre gauche et ceux du sous-arbre droit (respectivement les valeurs à gauche de x dans I et les valeurs à droite de x dans I). On sait alors que les parcours infixes des sous-arbres gauches et droits seront I_g et I_d tels que $I = I_g + [x] + I_d$. De plus, on peut décomposer P en $P = [x] + P_g + P_d$ par cardinalité (c'est-à-dire tel que $\text{Card}(P_g) = \text{Card}(I_g)$ et de même pour P_d et I_d). Ces sous-listes correspondent aux parcours préfixes des sous-arbres gauches et droits. Par hypothèse de récurrence, on a donc bien l'unicité.

2. La preuve ci-dessous donne une idée de l'algorithme. On utilise deux fonctions auxiliaires :
   - `indice` qui permet de déterminer l'indice d'un élément dans une liste.
   - `partition` qui permet de séparer une liste ℓ en deux sous-listes ℓ_1 et ℓ_2 telle que la taille de ℓ_1 est donnée par un paramètre. Ici, le dernier cas de filtrage est inutile et sert uniquement à éviter l'avertissement de non exhaustivité.

```
let rec indice x = function
   | [] -> -1
   | y :: q when y = x -> 0
   | y :: q -> 1 + indice x q;;

let rec partition n l = match n, l with
   | 0, _ -> [], l
   | _, x :: q -> let l1, l2 = partition (n - 1) q in x :: l1, l2
   | _, [] -> [], [];;

let rec const_arbre pre inf = match pre with
   | [] -> Vide
   | x :: q -> let n = indice x inf in
      let gpre, dpre = partition n q in
      let ginf, dinf = partition n inf in
      N(x, const_arbre gpre ginf, const_arbre dpre (List.tl dinf));;
```

Exercice 11

La fonction de traitement affiche une racine et rajoute les fils à la file d'attente. Tant que cette dernière n'est pas vide, on extrait un élément qu'on traite.

```
let largeur a =
   let file = Queue.create () in
   Queue.push a file;
   let traitement = function
      | Vide -> ()
      | N(x, g, d) -> print_int x; Queue.push g file; Queue.push d file in
   while not (Queue.is_empty file) do
      let b = Queue.pop file in
      traitement b
   done;;
```