# Adressage d'arbres - Correction

Samy Jaziri

## 1 Introduction

### Structure de donnée

Ce sujet s'attaque à l'adressage des arbres. Pour adresser les arbres, il va falloir les étiqueter. En étudiant la définition des arbres $A_n$ on peut conclure que l'on travaille avec des arbres binaires **entiers**, **non vide**. On choisit donc un type récursif classique où les feuilles consistent en deux entiers et les nœuds en deux entiers et deux sous arbres.
L'étiquetage de l'arbre par les adresses de la partie 2 peut être fait a posteriori. Si on choisit de faire cela il faudra étiqueter l'arbre avec des références. Ce qui donnerait

```
type arbre = Feuille of int ref * int ref
           | Noeud of int ref * int ref * arbre * arbre ;;
```

Dans ce corrigé on choisit plutôt d'étiqueter l'arbre à sa construction et de se passer donc de références. On travaillera avec le type :

```
type arbre = Feuille of int * int
           | Noeud of int * int * arbre * arbre ;;
```

### Génération d'arbres aléatoires

#### Suite de nombre aléatoire

Pas de difficulté ici

```
(*
Paramètre : n entier positif
Génère u_n tel que décrit dans l'énoncé
*)
let rec u ( n : int ) : int =
  if n == 0 then u0 else 15091 * ( u ( n - 1) ) mod 64007 ;;
```

#### Arbres binaires aléatoires

On commence par construire les arbres $A_n$ tels que décrits dans l'énoncé. Pour l'instant les étiquettes seront $(0, 0)$.

```
(*
Paramètre = n : entier positif
Génère A_n tel que décrit dans l'énoncé
*)
let rec a ( n : int ) : arbre =
  if n == 0 then Feuille ( 0, 0 )
  else Noeud ( 0, 0,
         ( a ( ( u ( 2 * n ) ) mod n ) ),
         ( a ( ( u ( 2 * n + 1 ) ) mod n ) ) ) ;;
```

Pour calculer le nombre de nœuds, de feuilles et la hauteur de l'arbre, on réalise un parcours de l'arbre. On choisit un parcours en profondeur suffixe. Il faut calculer les grandeurs relatives au deux sous-arbres pour en déduire les grandeurs de l'arbre demandées.

```
(*
   Paramètre = arbre : un arbre binaire
   Renvoie un triplé ( n, f, h ) où
    - n est le nombre de noeud de l'arbre
    - f est le nombre de feuilles de l'arbre
    - h est la hauteur de l'arbre
*)
let rec grandeurs_arbre ( arbre : arbre ) : int * int * int =
  match arbre with
  | Feuille( _, _ ) -> ( 1, 1, 1 )
  | Noeud ( _, _, fils_gauche, fils_droit ) ->
        let nb_noeuds_gauche, nb_feuilles_gauche, hauteur_gauche =
          ( grandeurs_arbre fils_gauche ) in
        let nb_noeuds_droit, nb_feuilles_droit, hauteur_droit =
          ( grandeurs_arbre fils_droit ) in
        ( 1 + nb_noeuds_gauche + nb_noeuds_droit, (* Nombre de noeuds *)
          nb_feuilles_gauche + nb_feuilles_droit, (* Nombre de feuilles *)
          1 + ( max hauteur_gauche hauteur_droit ) ) (* Hauteur de l'arbre *)
;;
```

Soit $A$ un arbre binaire entier non vide. On note $\mathcal{N}(A)$ le nombre de nœuds de $A$, $\mathcal{F}(A)$ le nombre de feuilles de $A$ et $\mathbf{h}(A)$ la hauteur de $A$. On appellera *taille de l'arbre* $A$, son nombre de nœuds.
On remarque d'abord que l'arbre $A_n$ décrit par l'énoncé un bien un arbre de hauteur finie plus petite que $n+1$. La preuve formelle peut être faîte par récurrence et se base sur le fait que si $A_k$ est un fils de $A_n$ alors $0 \leq k \leq n-1$ par définition du reste de la division euclidienne. $A_n$ est donc un arbre fini.
**Terminaison :** L'arbre passé en paramètre est un arbre fini de taille $n$. L'algorithme effectue deux appels récursifs. Le premier sur le sous-arbre gauche et le second sur le sous-arbre droit, chacun de taille inférieure ou égale à $n-1$. L'algorithme termine donc.
**Correction :** On prouve la correction par récurrence forte sur la taille de l'arbre. Soit $A$ l'arbre binaire entier non vide passé en paramètre.

- *Initialisation :* $\mathcal{N}(A) = 1$ implique que $A$ est réduit à une feuille. On a bien $(\mathcal{N}(A), \mathcal{F}(A), \mathbf{h}(A)) =$

$(1, 1, 1)$.

- *Hérédité :* Soit $\mathcal{N}(A) = n > 1$ et supposons que l'algorithme soit correct quelque soit $1 \leq k < n$. $\mathcal{N}(A) > 1$ on peut donc extraire $A_g$ et $A_d$ les sous-arbres gauche et droit respectivement de $A$. On sait que
  - $\mathcal{N}(A) = \mathcal{N}(A_g) + \mathcal{N}(A_d) + 1$
  - $\mathcal{F}(A) = \mathcal{F}(A_g) + \mathcal{F}(A_d)$
  - $\mathbf{h}(A) = \max(\mathbf{h}(A_g), \mathbf{h}(A_d)) + 1$

On sait de plus que $1 \leq \mathcal{N}(A)_g < \mathcal{N}(A)$ et $0 \leq \mathcal{N}(A)_d < \mathcal{N}(A)$, ainsi, par hypothèse de récurrence, les grandeurs calculées par les appels récursifs de l'algorithme correspondent bien à celles des sous-arbres de $A$ et l'algorithme est correct pour un arbre $A$ de taille $n$.
**Complexité :** Chaque nœud de $A$ est visité exactement une fois dans ce parcours préfixe. La complexité est en $\mathcal{O}(\mathcal{N}(A))$.

## 2 Schéma d'adressage des arbres

### Adressage

On choisit pour résoudre cette question d'étiqueter les arbres $A_n$ à leur création. Pour ce faire on réalise un parcours en profondeur préfixe, pour calculer les $l_i$, et suffixe, pour calculer les $m_i$. On remarque en effet que, pour un nœud donné de l'arbre, *la plus grande valeur de $l_j$ pour les nœuds du sous-arbre enraciné en ce nœud* est *la plus grande valeur de $l_j$ des noeuds de son* ***sous-arbre droit***. La preuve de ce fait se basera sur la nature de l'étiquetage, fondé sur la numérotation des nœuds selon l'ordre du parcours en profondeur **préfixe**.

```
(*
Paramètre = arbre : un arbre binaire entier non vide
Renvoie l'adresse de la racine de l'arbre
*)
let adresse_racine ( arbre : arbre ) : int * int = match arbre with
  | Feuille ( l, m ) -> ( l, m )
  | Noeud ( l, m, _, _ ) -> ( l, m ) ;;
```

```
(*
Paramètre = n : entier positif
Génère A_n étiqueté par les ( l_i, m_i ), tels que décrit dans l'énoncé.
*)
let a ( n : int ) : arbre =
  (* Paramètre = l : prochaine étiquette à donner lors du parcour *)
  let rec creation_et_adressage ( l : int ) ( n : int ) : arbre =
    if n == 0 then Feuille ( l, l )
    else
      let fils_g =
        creation_et_adressage ( l + 1 ) ( ( u ( 2 * n ) ) mod n ) in
      let _, m_g = adresse_racine fils_g in
      let fils_d =
        creation_et_adressage ( m_g + 1 ) ( ( u ( 2 * n + 1 ) ) mod n ) in
      let _, m_d = adresse_racine fils_d in
      Noeud ( l, m_d, fils_g, fils_d ) in
  creation_et_adressage 0 n ;;
```

**Correction :** On montre la correction de l'algorithme `creation_et_adressage` par récurrence forte sur la taille de l'arbre binaire entier non vide passé en paramètre de l'algorithme. On note $\mathcal{T}(n)$ la taille de l'arbre $A_n$.
La propriété qui sera montrée par récurrence est la suivante :

$\mathcal{P}(\mathcal{T}(n))$ = *Pour tous entiers $l$ et $n$, passés en paramètre, l'algorithme renvoie l'arbre $A_n$ de l'énoncé tel que chaque nœud $n_i$ est étiqueté par un couple d'entier $(k_i, m_i)$ tels que :*

- *$k_i$ est le numéro de $n_i$ dans un parcours en profondeur préfixe, en supposant que la racine porte le numéro $l$ (premier numéro)*
- *$m_i$ est la plus grande valeur de $k_j$ pour les nœuds $n_j$ du sous-arbre enraciné en $n_i$*

*Initialisation :* $\mathcal{P}(1)$ : Par définition $n = 0$ et $A_n$ est une feuille. Quelque soit $l \in \mathbb{N}$, pour une feuille la propriété est vrai, car la racine est la feuille elle-même et est numéroté $l$. Les sous-arbres sont vides ainsi la plus grande valeur des $k_j$ pour les nœuds du sous-arbre est $l$.
*Hérédité :* Soit $v > 1$ et soit $n \in \mathbb{N}^*$ tel que $\mathcal{T}(n) = v$. Supposons $\mathcal{P}(u)$ vrai pour tout $1 \leq u < v$. Soit $l \in \mathbb{N}$ passé en paramètre.
Soit $A$ l'arbre construit par l'algorithme. Soit $A_g$ et $A_d$ les sous-arbres gauche et droit de $A$ calculés par appels récursifs de l'algorithme. Nécessairement $1 \leq \mathcal{T}(u_{2\,n} \mod n) < \mathcal{T}(n)$ et $1 \leq \mathcal{T}(u_{2\,n+1} \mod n) < \mathcal{T}(n)$ (par définition de $A_n$). Par hypothèse de récurrence $A_g$ est l'arbre $A_{u_{2\,n} \mod n}$ de l'énoncé et $A_d$ est l'arbre $A_{u_{2\,n+1} \mod n}$ de l'énoncé. Donc $A$ est l'arbre $A_n$ de l'énoncé par définition.
Soit $(l_g, m_g)$ et $(l_d, m_d)$ les étiquettes de la racine de $A_g$ et $A_d$ respectivement. Par hypothèse de récurrence, $l_g = l + 1$ et $l_d = m_g + 1$. Aussi $m_g$ est la plus grande valeur de numéros qui apparaît dans le sous-arbre gauche et $m_d$ la plus grande valeur de numéro trouvée dans le sous-arbre droit. Le parcours en profondeur préfixe numérote d'abord la racine, puis son sous-arbre gauche récursivement puis son sous-arbre droit.
On commence par noter que le nœud de $A$ ayant le plus grand numéro est le dernier nœud visité par le parcours en profondeur préfixe, i.e. un nœud de $A_d$. D'où $m = m_d$ avec $m$ la plus grande valeur des numéros apparaissant dans $A$.

Ensuite l'algorithme numérote bien la racine par $l$, puis fait un appel récursif sur le sous-arbre gauche avec $l = l + 1$. Par hypothèse de récurrence, chaque nœud de $A_g$ est bien étiqueté avec son numéro dans un parcours en profondeur préfixe débutant par $l + 1$ sur sa racine, ce qui correspond aux numéros du parcours en profondeur préfixe débutant à la racine de $A$. Par hypothèse de récurrence, chaque nœud de $A_d$ est bien étiqueté avec son numéro dans un parcours en profondeur préfixe débutant par $m_g + 1$ sur sa racine, ce qui correspond au plus grand numéro des nœuds de $A_g$ donc au dernier numéro attribué lors du parcours en profondeur préfixe de $A_g$, après avoir numéroté la racine de $A$. Les étiquettes de $A_d$ correspondent donc bien aux numéros attribués par lors du parcours en profondeur préfixe de $A$.
On peut conclure par récurrence la validité de la propriété.
On note que l'algorithme `construction_et_adressage` est appelé avec comme paramètre 0 $n$ ce qui permet de prouver la correction de l'algorithme `a`.

### Recherche d'adresse

Pour trouver ensuite le $m_i$ associé à un $l_i$ on fera une recherche dans l'arbre. On exploitera les propriétés du parcours en profondeur infixe pour choisir à chaque fois le bon sous-arbre à explorer. En effet, si on cherche une adresse $l$ qui n'est pas celle du nœud courant et que les sous-arbres gauche et droit sont respectivement étiquetés $(l_g, m_g)$ et $(l_d, m_d)$ on peut faire le raisonnement suivant :

- Si $l \leq m_g$ alors $l$ se situe dans le sous-arbre gauche
- Sinon $l$ se situe dans le sous-arbre droit.

```
(*
Paramètre = adresse : l'adresse du noeud à chercher
Paramètre = arbre : l'arbre dans lequel chercher l'adresse
Renvoie l'étiquette du noeud à l'adresse passée en paramètre'
*)
let rec chercher_noeud ( arbre : arbre ) ( adresse : int ) : int * int =
  match arbre with
  | Feuille ( l, m ) when l != adresse ->
      raise ( Invalid_argument "Adresse non présente dans l'arbre" )
  | Feuille ( l, m ) -> ( l, m )
  | Noeud ( l, m, _, _ ) when l == adresse -> ( l, m )
  | Noeud ( _, _, fils_g, fils_d ) ->
      let _, m_g  = adresse_racine fils_g in
      if adresse <= m_g then chercher_noeud fils_g adresse
      else chercher_noeud fils_d adresse ;;
```

**Complexité :** Chaque appel récursif se fait sur un seul des deux sous arbre. On descend ainsi jusqu'à une feuille. On a donc au maximum autant d'appels récursif que la hauteur de l'arbre passé en paramètre. A chaque appel, les opérations effectuées sont en $\mathcal{O}(1)$, la complexité finale est en $\mathcal{O}(\mathbf{h}(A)) = \mathcal{O}(\log(\mathcal{N}(A)))$ pour un arbre $A$ passé en paramètre.

### Trouver son chemin

Nous avons déjà vu comment exploiter l'adressage lorsque l'on part de la racine de l'arbre. Il s'agit maintenant de partir d'un nœud quelconque. Il manque pour cela dans le raisonnement de la partie

précédente, à déterminer comment savoir s'il faut remonter l'arbre. Si l'on cherche à aller à une adresse $(l, m)$ depuis un nœud d'adresse $(l', m') \neq (l, m)$ on peut distinguer deux cas :

- $l < l' < m$, alors le nœud d'adresse recherché est dans le sous-arbre du nœud de départ
- dans l'autre cas le nœud d'adresse recherché n'est pas dans le sous-arbre du nœud de départ, il faut remonter l'arbre.

Pour calculer la distance parcourue en nombre de nœuds traversés d'un nœud $n$ à un nœud $m$, nous allons chercher le plus petit sous-arbre de l'arbre de départ qui contient $n$ et $m$. Il suffira ensuite de faire la somme de la distance de $n$ et $m$ à la racine de cet arbre obtenue en adaptant la fonction `chercher_noeud` pour renvoyer le nombre de nœuds traversés.

```
(*
   Paramètre = arbre : l'arbre dans lequel le calcul de distance a lieu
   Paramètre = adresse : l'adresse du noeud dont on calcul la distance
   Renvoie la distance à la racine de l'arbre du noeud d'adresse <adresse>
*)
let distance_a_la_racine ( arbre : arbre ) ( adresse : int ) : int =
    (*
        Paramètre = arbre : l'arbre dans lequel le calcul de distance a lieu
        Paramètre = distance : la distance accumulée depuis le début du parcours
        Renvoie <distance> + la distance à la racine de l'arbre du noeud
            d'adresse <adresse>
    *)
  let rec recherche ( arbre : arbre ) ( distance : int ) : int =
    match arbre with
    | Feuille ( l, m ) when l != adresse ->
        raise ( Invalid_argument "Adresse non présente dans l'arbre" )
    | Feuille ( l, m ) -> distance
    | Noeud ( l, m, _, _ ) when l == adresse -> distance
    | Noeud ( _, _, fils_g, fils_d ) ->
        let _, m_g = adresse_racine fils_g in
        if adresse <= m_g then recherche fils_g ( distance + 1 )
        else recherche fils_d ( distance + 1 ) in
  recherche arbre 0 ;;
```

```
(*
   Paramètre = arbre : l'arbre dans lequel le calcul de distance a lieu
   Paramètre = adresse_source : l'adresse du noeud de départ
   Paramètre = adresse_destination : l'adresse du noeud de destination
   Renvoie la distance du noeud de départ au noeud de destination dans l'arbre
*)
let rec distance
    ( arbre : arbre  )
    ( adresse_source : int )
    ( adresse_destination : int )
    : int =
  match arbre with
    | Feuille ( l, m ) when l != adresse_source || l != adresse_destination ->
        raise ( Invalid_argument "Incohérence dans l'arbre" )
    | Feuille ( l, m ) -> 0
    | Noeud ( l, m, fils_g, fils_d ) ->
        let l_g, m_g = adresse_racine fils_g in
        if ( l_g <= adresse_source && adresse_source <= m_g ) &&
           ( l_g <= adresse_destination && adresse_destination <= m_g )
        then distance fils_g adresse_source adresse_destination
        else let l_d, m_d = adresse_racine fils_d in
            if ( l_d <= adresse_source && adresse_source <= m_d ) &&
            ( l_d <= adresse_destination && adresse_destination <= m_d )
            then distance fils_d adresse_source adresse_destination
            else ( distance_a_la_racine arbre adresse_source )
                 + ( distance_a_la_racine arbre adresse_destination )
                 - 1 (* la racine est parcouru deux fois sinon *) ;;
```

## 3 Affichage des résultats de la fiche réponse

```
Printf.printf "Question 1 : a) %i  b) %i  c) %i \n"
  ( u 10 ) ( u 100 ) ( u 1000 ) ;;

let a10 = a 10 ;;
let a100 = a 100 ;;
let a1000 = a 1000 ;;

let noeuds_10, feuilles_10, hauteur_10 = grandeurs_arbre a10 in
let noeuds_100, feuilles_100, hauteur_100 = grandeurs_arbre a100 in
let noeuds_1000, feuilles_1000, hauteur_1000 = grandeurs_arbre a1000 in
Printf.printf
  "Question 2 : a) ( %i, %i, %i )  b) ( %i, %i, %i )  c) ( %i, %i, %i )\n"
  noeuds_10 feuilles_10 hauteur_10
  noeuds_100 feuilles_100 hauteur_100
  noeuds_1000 feuilles_1000 hauteur_1000 ;;

let _, m_10 = chercher_noeud a10 3 in
let _, m_100 = chercher_noeud a100 9 in
let _, m_1000 = chercher_noeud a1000 30 in
Printf.printf "Question 3 : a)  %i  b) %i  c) %i\n" m_10 m_100 m_1000 ;;

let d_10 = distance a10 3 9 in
let d_100 = distance a100 5 30 in
let d_1000 = distance a1000 30 90 in
Printf.printf "Question 4 : a)  %i  b) %i  c) %i\n" d_10 d_100 d_1000 ;;
```