Table des matières

Premier devoir surveillé d'informatique

Durée quatre heures. Le sujet se compose de deux exercices et d'un problème. Il est possible de rendre un complément d'ici la fin de la semaine dans mon casier.

Première partie
Exercices

Pour ces exercices, on utilise un type d'arbre binaire défini par :

```
1    type 'a arbre = Vide | Noeud of ('a arbre * 'a * 'a arbre);;
```

Exercice 1. ** Diviser pour régner

On rappelle que la fonction `fusion_avec_colle` vue en TD a pour type `'a arbre -> 'a -> 'a arbre -> 'a arbre` et que si on lui fournit deux tas $t1$ et $t2$ et un élément x, elle renvoie un tas contenant les éléments de $t1$, de $t2$, ainsi que x. Ce nouveau tas a pour hauteur $1 + \max(h_{t1}, h_{t2})$. Et la complexité de ce calcul est en $O\left(\max(h_{t1}, h_{t2})\right)$.

On propose la méthode suivante, de type « diviser pour régner », pour créer un tas à partir des éléments contenus dans une liste.

- *Diviser :* La liste est découpée en un élément et deux sous listes de longueurs égales ou presque ;
- *Appels récursifs :* Les deux sous listes sont converties en tas ;
- *Régner :* Les deux tas et l'élément restant sont rassemblés au moyen de la fonction `fusion_avec_colle`.

On ne demande pas de programmer cette méthode mais d'en donner la complexité, en justifiant.

solution :

- Pour toute liste l, en notant $n = |l|$, découper la liste en un éléments et deux sous-listes se fait en $O(n)$.
- La complexité de `fusion_avec_colle` est dominée par la hauteur du plus grand des tas à fusionner.
- La fonction `fusion_avec_colle` entre des tas équilibrés nous fournira toujours des tas équilibrés, donc de hauteur logarithmique en le nombre d'éléments.
- Notons pour tout $n \in \mathbb{N}$, C_n le nombre maximal de comparaisons pour convertir une liste de longueur au plus n en un tas. On aura alors :
$$\begin{cases} C_0 = 0 \\ C_1 = 0 \\ \forall n \in \mathbb{N},\ C_n \leqslant C_{\lfloor n/2 \rfloor} + C_{\lfloor (n-1)/2 \rfloor} + O(n) + O(\log n) \leqslant 2C_{\lfloor n/2 \rfloor} + O(n) \end{cases}$$
(Le $O(n)$ correspond au découpage de la liste en deux, et le $O(\log n)$ à la fusion.)
On est dans le cas critique du master theorem (avec les notations du cours, $a + b = 2$, donc $\log_2(a + b) = 1$ et $\beta = 1$). D'où $C_n = O(n \log n)$.
Ainsi l'ordre de grandeur est le même que pour la méthode naïve, et mon bon que pour la méthode de Floyd.

Exercice 2. * k-fusion**

(En anglais « K-way merge »)

Dans l'algorithme du tri fusion, on fusionne deux listes triées. Dans certaines circonstances, on peut avoir besoin d'en fusionner un plus grand nombre. On fixe $k \in \mathbb{N}^*$, et on va étudier une méthode pour fusionner k listes triées.

1. Donner, en fonction de k et de la somme n des longueurs des k listes , la complexité de l'algorithme naïf consistant à rechercher à chaque étape le plus petit élément parmi les k têtes de listes pour le placer dans le résultat.
   solution : Le calcul du maximum des k têtes de listes se fait en $O(k)$. Il faut refaire ce calcul n fois pour obtenir la liste finale, d'où une complexité en $O(nk)$.
2. On propose de maintenir un tas-min pour contenir les k têtes de liste. Quelle sera alors la complexité ?
   solution : Avec cette méthode, le calcul du minimum des k têtes de listes est en $O(1)$. Cependant, il faudra prévoir un $O(\log k)$ pour extraire ce minimum du tas, et un autre $O(\log k)$ pour insérer la nouvelle tête de liste à la place. Ce qui nous donnera une complexité totale en $O\left(n \log(k)\right)$.
3. *Programmation :*

   (a) Programmer une fonction `insertion` qui prend un arbre a, supposé avoir une structure de tas-min, et un élément x et qui renvoie un tas-min contenant les éléments de a ainsi que x. L'usage répété de la fonction `insertion` doit produire un arbre aussi équilibré que possible.
   solution : Pour une structure de tas-min, il suffit d'inverser toutes les comparaisons dans les fonctions faites en cours pour des tas-max.

```
let rec insertion x = function
  | Vide -> Noeud( Vide, x, Vide)
  | Noeud(fg, e, fd) when x <= e ->
     Noeud(fd, x, insertion e fg)   (* Attention : on veut un tas-min. Et ne pas
    ↪ oublier l'auto-équilibrage.*)
  | Noeud(fg, e, fd) ->
     Noeud(fd, e, insertion x fg)
;;
```

   (b) Expliquer le rôle de la fonction suivante, et préciser son type :

```
let prochain t i=
      match t.(i) with
        |[] -> failwith "liste vide"
        |x::suite -> begin
                       t.(i) <- suite;
                       x
                     end
;;
```

   solution : Cette fonction permet de supprimer en effet de bord la tête de la $i$ème liste, et de renvoyer celle-ci. Il s'agit d'une fonction-procédure semblable au « pop » des piles.

   (c) Écrire une fonction `tas_initial` prenant en entrée le tableau de listes t et renvoyant un tas min contenant la tête de chaque liste de t. Plus précisément, le tas contiendra des couples de la forme *(tête d'une liste, indice de cette liste dans t)*. Le tableau t sera supposé contenir uniquement des listes non vides. En outre, votre fonction devra avoir un effet de bord : supprimer la tête de chaque liste.
   solution :

```
let tas_initial t =

  let k=Array.length t in

  let rec aux i =
    if i=k then Vide
    else
      insertion (prochain t i, i)  (aux (i+1))

  in
  aux 0
;;
```

(d) Écrire la fonction finale de k-fusion.
solution :

```
let k_fusion t =
  (* Entrée : t, tableau de listes triées
     Sortie : la liste fusion de toutes les listes de t.*)


  let rec aux tas =
    (* tas : tas-min contenant les têtes de listes *)
    if tas=Vide then []
    else
      let (m,i), suite_tas = extraitMin tas in
      let nv_tas = if t.(i) = [] then suite_tas
                   else insertion (prochain t i, i) suite_tas
      in
      m :: aux nv_tas

  in
  aux (tas_initial t)
;;
```

Deuxième partie

Problème

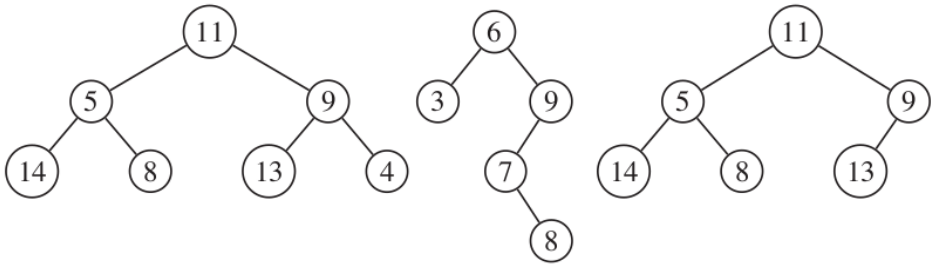
Les fonctions écrites devront être récursives ou faire appel à des fonctions auxiliaires récursives. Elles ne devront pas utiliser d'instructions itératives (c'est-à-dire for, while, …) ni de références.

On rappelle que le symbole $\vee$ signifie « ou », et $\wedge$ signifie « et ».

1 Arbres binaires d'entiers

Définition 1.1. *Un arbre binaire d'entiers a est une structure qui peut, soit être vide (notée $\emptyset$), soit être un nœud qui contient une étiquette entière (notée $\mathcal{E}(a)$), un sous-arbre gauche (noté $\mathcal{G}(a)$) et un sous-arbre droit (noté $\mathcal{D}(a)$) qui sont tous deux des arbres binaires d'entiers. L'ensemble des étiquettes de tous les nœuds de l'arbre a est noté $\mathcal{C}(a)$.*

Exemple 1. Voici trois exemples d'arbres binaires étiquetés par des entiers (les sous-arbres vides qui sont fils gauche ou droit des nœuds ne sont pas représentés) :



Définition 1.2. *Les branches d'un arbre relient la racine aux sous-arbres vides. La profondeur d'un arbre a est égale au nombre de liaisons entre nœuds de la branche la plus longue. Nous la noterons $|a|$. Nous associerons la profondeur -1 à l'arbre vide $\emptyset$.*

Exemple 2. Les profondeurs des trois arbres binaires de l'exemple 1 sont respectivement 2, 3 et 2.

Définition 1.3. *Un arbre binaire complet est un arbre binaire dont tous les niveaux sont complets, c'est-à-dire que tous les nœuds d'un même niveau ont deux fils non vides sauf les nœuds du niveau le plus profond qui n'ont aucun fils (c'est-à-dire deux fils vides).*

Exemple 3. Le premier arbre binaire de l'exemple 1 est complet.

1. Montrer que, dans un arbre binaire complet non vide, le niveau de profondeur p contient 2^p nœuds.
   solution :

   ⚷ Pour cette question presque évidente, je ne pense pas qu'une démonstration par récurrence était indispensable.

   Soit a un arbre binaire complet non vide et h sa profondeur. Pour tout $p \in [\![0, h-1]\!]$, le niveau $p+1$ a deux fois plus d'éléments que celui de profondeur p puisque chaque nœud de l'étage p a exactement deux fils. Comme le niveau de profondeur 0 contient un nœud (la racine), il vient que pour tout $p \in [\![0, h]\!]$, le niveau de profondeur p a 2^p nœuds (suite géométrique de raison 2).
2. Calculer le nombre n de nœuds d'un arbre binaire complet non vide de profondeur p.
   solution : Pour tout $k \in [\![0, p]\!]$, il y a 2^k nœuds à la profondeur k, cela nous fait un total de $\sum_{k=0}^{p} 2^k$ nœuds, soit $2^{p+1} - 1$ nœuds.
3. En déduire la profondeur p d'un arbre binaire complet non vide contenant n éléments.
   solution : Soit a un arbre binaire complet non vide contenant n nœuds, et soit p sa profondeur. On a par la question précédente $n = 2^{p+1} - 1$. D'où $p = \log_2(n+1) - 1$.

2 Exercice : Tri par arbre binaire de recherche

Définition 2.1. *Un arbre binaire de recherche est un arbre binaire d'entiers dont :*

- *les fils de la racine sont des arbres binaires de recherche ;*

- *les étiquettes de tous les nœuds composant le fils gauche de la racine sont inférieures ou égales à l'étiquette de la racine ;*
- *et les étiquettes de tous les nœuds composant le fils droit de la racine sont strictement supérieures à l'étiquette de la racine.*

Ces contraintes s'expriment sous la forme :

$$ABR(a) \Leftrightarrow a = \emptyset \vee \begin{cases} ABR\left(\mathcal{G}(b)\right) \wedge ABR\left(\mathcal{D}(a)\right) \\ \forall e \in \mathcal{C}\left(\mathcal{G}(a)\right),\ e \leqslant \mathcal{E}(a) \\ \forall e \in \mathcal{C}\left(\mathcal{D}(a)\right),\ e > \mathcal{E}(a) \end{cases}$$

Exemple 4. Le deuxième arbre de l'exemple 1 est un arbre binaire de recherche.

Un arbre binaire d'entiers est représenté par le type arbre dont la définition est :

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

Dans l'appel `Noeud( fg, v, fd)`, les paramètres `fg`, `v` et `fd` sont respectivement le fils gauche, l'étiquette et le fils droit de la racine de l'arbre créé.

Exemple 5. L'expression suivante :

```
Noeud(
  Noeud( Vide, 3, Vide)
  6,
  Noeud(
    Noeud(
      Vide,
      7,
      Noeud( Vide, 8, Vide)),
    9,
    Vide))
```

est alors associée au deuxième arbre binaire représenté graphiquement dans l'exemple 1.

Définition 2.2. *Une séquence s de taille n de valeurs entières v_i avec $1 \leq i \leq n$ est notée $\langle v_1, ..., v_n \rangle$. Une même valeur v peut figurer plusieurs fois dans s, nous noterons $card(v, s)$ le cardinal de v dans s, c'est-à-dire le nombre de fois que v figure dans s avec :*

$$card(v, \langle v1, ..., vn \rangle) = Card\left\{ i \in \mathbb{N} \mid \ 1 \leq i \leq n, v = v_i \right\}.$$

Une séquence d'entiers est représentée par le type sequence dont la définition est :
`type sequence = int list;;`
Soit le programme en langage CaML :

```
let rec ajouter v a =
  match a with
    | Vide -> Noeud(Vide,v,Vide)
    | Noeud (g,e,d) ->
        if (v = e)
          then Noeud(Noeud(g,e,Vide),v,d)
          else
            (if (v < e)
               then Noeud((ajouter v g),e,d)
               else Noeud(g,e,(ajouter v d)));;

let trier s =
  let rec aux1 l r =
  match l with
    | [] -> r
    | t::q -> (aux1 q (ajouter t r)) in
  let rec aux2 a =
    match a with
```

```
19      | Vide -> []
20      | Noeud(g,v,d) -> (aux2 g) @ (v :: (aux2 d)) in
21  (aux2 (aux1 s Vide));;
```

Soit la constante `exemple` définie et initialisée par : `let exemple = [ 2; 1; 3];;`

4. Détailler les étapes du calcul de `(trier exemple)` en précisant pour chaque appel aux fonctions `ajouter`, `aux1`, `aux2` et `trier`, la valeur du paramètre et du résultat.
5. Montrer que le calcul des fonctions `ajouter`, `aux1`, `aux2` et `trier` se termine quelles que soient les valeurs de leurs paramètres respectant le type des fonctions.
   solution :
   - `ajouter` : la hauteur de l'arbre passé en argumente est un entier positif qui décroît strictement à chaque appel récursif.
   - `aux1` : La longueur de la liste passée en argument est un entier positif qui décroît strictement à chaque appel récursif.
   - `aux2` : idem `aux1`
   - `trier` : un seul appel à `aux1` et à `aux2`.
6. Soit l'arbre binaire d'entiers a, soit l'entier v et soit l'arbre binaire d'entiers b tel que `b = (ajouter v a)`, montrer que :
   $$ABR(a) \Rightarrow ABR(b)$$
   Indication : Récurrence sur la hauteur de a.

   solution : On fixe un entier v. Pour tout $p \in \mathbb{N}$, soit $P(p)$: « Pour tout arbre binaire a de profondeur p, on a $ABR(a) \Rightarrow ABR($`ajouter v a`$)$.».
   - `ajouter v Vide` est une feuille, c'est donc un ABR, donc $P(0)$ est vrai.
   - Soit $p \in \mathbb{N}$ tel que $\forall k \in [\![0, n]\!],\ P(k)$. Soit a un arbre binaire de profondeur $p+1$, soient g et d les fils de sa racine, et e son étiquette. On sépare les trois cas qui apparaissent dans le code :
     - ◇ Si $e = v$, alors `ajouter v e = Noeud(Noeud(g,e,Vide),v,d)`. À gauche de v on trouve e et g, qui ne contiennent que des étiquettes $\leqslant e$. À droite de v on trouve d qui ne contient que des étiquettes $> e$. Enfin, les deux fils `Noeud(g,e,Vide)` et `d` sont eux aussi des ABR.
     - ◇ Si $v < e$, l'arbre renvoyé est `Noeud (( ajouter v g),e,d)`. Or, `( ajouter v g)` est un ABR par l'hypothèse de récurrence, il ne contient de plus que des étiquettes $\leqslant e$. Et d est un ABR qui ne contient que des étiquettes $> e$. Donc l'arbre total est bien un ABR.
     - ◇ Si $v > e$, raisonnement analogue.

     Dans tous les cas, `ajouter v a` est un ABR. D'où $P(p+1)$.

   En conclusion, $\forall p \in \mathbb{N}$, $P(p)$.
7. Soit la séquence $p = \langle p_1, ..., p_m \rangle$ de taille m, soit la séquence $r = \langle r_1, ..., r_n \rangle$ de taille n telle que `r = (trier p)`, montrer que :
   (a) $m = n$
   (b) $\forall i, 1 \leq i \leq m,\ card(p_i, r) = \text{Card}(p_i, p)$
   (c) $\forall i, 1 \leq i < n,\ r_i \leq r_{i+1}$

   Indication : Ici il n'est pas raisonnable de rédiger les récurrences...

   solution :
   - Une récurrence simple prouve que pour tout arbre binaire a et tout entier v, `ajouter v a` renvoie un arbre dont les étiquettes sont celles de a ainsi que v. Dès lors on prouve encore par une simple récurrence que pour toute liste l, `aux1 l` renvoie un arbre dont les étiquettes sont les éléments de l. Et enfin que pour tout arbre a, `aux2 a` renvoie une liste dont les éléments sont les étiquettes de a.

     Au final, pour toute liste l, `trier l` est donc une liste ayant les mêmes éléments que l. D'où les deux premiers points.
   - Par la question précédente, et au moyen d'une troisième récurrence, on voit que pour toute séquence s, `aux1 s Vide` est un ABR.

     Une dernière récurrence montre que pour tout ABR a, `aux2 a` renvoie une liste triée. Je donne le principe de l'hérédité : si `a = Noeud(g,v,d)`, par hypothèse de récurrence et par le point précédent, `aux2 d` est une liste triée contenant les étiquettes de d. Comme de plus a est un ABR, les étiquettes de d sont $> v$. Donc `v::aux2 d` est triée. Enfin, `aux2 g` est un ABR contenant les étiquettes de g, qui sont donc toutes $\leqslant v$. Dès lors `aux2 g (v::aux2 d)` est triée.

8. Donner des exemples de valeurs du paramètre s de la fonction `trier` qui correspondent aux meilleur et pire cas en nombre d'appels récursifs effectués. On ne demande pas la démonstration du fait que ce sont effectivement le meilleur et le pire des cas.

   Calculer une estimation de la complexité dans les meilleur et pire cas de la fonction `trier` en fonction du nombre de valeurs dans les séquences données en paramètre. Cette estimation ne prendra en compte que le nombre d'appels récursifs à `ajouter`, `aux1`, `aux2` et `trier` effectués.

   Indication : Quelle sera la forme des arbres utilisés dans le pire et le meilleur des cas ?

   solution : Pour tout arbre a, je noterai h_a sa profondeur.

   Commençons par la complexité des fonctions intermédiaires :

   - Pour tout arbre a et entier v, `ajouter v a` a une complexité en $O(h_a)$, car la fonction parcourt au plus une branche de a.
   - Pour tout arbre a, `aux2 a` a une complexité égale au nombre de nœuds de a car elle parcourt une fois a en entier.
   - Soit l une liste et n sa longueur, `aux1 l` appel `ajouter` pour chaque élément de l. Le pire cas est atteint si l'arbre intermédiaire a toujours une hauteur maximale, c'est-à-dire si c'est un peigne. Ce cas est par exemple atteint si l était triée. Dans ce cas pour tout $i \in [\![0, n]\!]$, après insertion des i premiers éléments de l, l'arbre intermédiaire a pour hauteur $i-1$, donc la complexité de l'insertion suivante sera en $O(i)$. Cela nous fait une complexité totale en $\sum_{i=0}^{n} O(i)$, donc en $O(n^2)$.

     À l'opposé la situation la plus favorable est celle où les arbres intermédiaire sont les plus équilibrés, c'est-à-dire si tous les nœuds sauf ceux du dernier étage ont deux fils. Cette situation est atteinte par exemple si l contient les éléments d'un ABR dans l'ordre d'un parcours en largeur. Par exemple `[10; 5; 15; 2;8;12;20;1]`.

     Dans ce cas, après i insertions, la hauteur est en $O(\log i)$. Donc la complexité finale est $\sum_{i=1}^{n} O(\log i)$, ce qui donne $O(n \log n)$.

   En bilan, l'appel à `aux2` a un coût négligeable devant celai à `aux1`. Donc le coût du tri est celui de `aux1`, soit $O(n^2)$ au pire et $O(n \log n)$ au mieux.

3 Problème : Représentation de systèmes creux

La résolution numérique de problèmes en physique repose souvent sur la recherche de solutions pour un système linéaire qui résulte de la discrétisation du modèle mathématique continu utilisé pour représenter le problème. Lorsque les différentes parties du problème sont faiblement couplées, le système linéaire contient un grand nombre de valeurs nulles. Il est alors qualifié de système creux. Cette notion peut être généralisée à un système contenant un grand nombre de valeurs identiques. Nous appelons cette valeur identique, valeur par défaut des éléments. Pour les systèmes faiblement couplés évoqués précédemment, cette valeur par défaut est 0.0 [1].

L'objectif de ce problème est l'étude d'une représentation d'un tableau creux par un arbre binaire en numérotant les nœuds par les indices des éléments du tableau et en étiquetant les nœuds par les valeurs des éléments du tableau différentes de la valeur par défaut. L'utilisation de cette structure permet de réduire le temps d'accès aux éléments du tableau creux par rapport à l'utilisation naturelle d'une structure de liste. Par contre, cette structure utilise plus de mémoire que la liste mais moins que la représentation habituelle des tableaux.

Un tableau creux est représenté par un arbre binaire dont les noeuds sont soit vides, soit étiquetés par les valeurs des éléments contenus dans le tableau creux qui sont différentes de la valeur par défaut, et les chemins de la racine de l'arbre aux nœuds sont extraits du codage binaire de l'indice de l'élément dont la valeur est contenue dans le nœud.

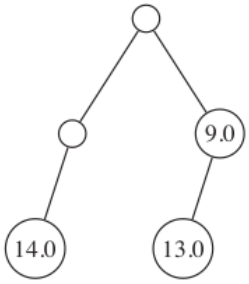
3.1 Arbre binaire partiel de réels

Un arbre binaire partiel de réels est un arbre binaire de réels dont certains nœuds ne contiennent pas d'étiquettes.

Définition 3.1. *Un arbre binaire partiel de réels a est une structure qui peut soit être vide (notée $\emptyset$), soit être un nœud qui contient une étiquette réelle (notée $\mathcal{E}(a)$), un sous- arbre gauche (noté $\mathcal{G}(a)$) et un sous-arbre droit (noté $\mathcal{D}(a)$) qui sont tous deux des arbres binaires partiels de réels, soit être une fourche qui est un nœud qui ne contient pas d'étiquette. L'ensemble des fourches de l'arbre a est noté $\mathcal{F}(a)$. L'ensemble des nœuds de l'arbre a qui ne sont pas des fourches est noté $\mathcal{N}(a)$.*

Exemple 6. Voici un exemple d'arbre binaire partiel étiqueté par des réels (les sous-arbres vides des nœuds ne sont pas représentés) :

1. NDT : Le choix d'utiliser des flottants vient du contexte choisi, l'analyse numérique.



Pour exploiter un arbre binaire partiel pour représenter un vecteur creux, il faut associer un indice à chaque nœud de l'arbre contenant une valeur.

solution : Remarques sur cette partie :

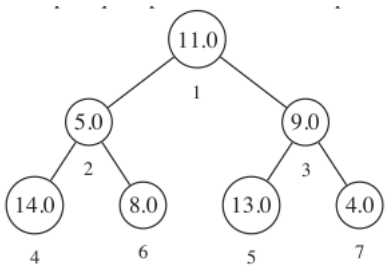
- p et a en général par définis par l'énoncé...
- Les mots « numéro » et « indice » semblent permutables.
- Une erreur d'énoncé.

Définition 3.2. *La numérotation hiérarchique des nœuds d'un arbre binaire consiste à associer le numéro 1 à la racine de l'arbre puis à calculer les numéros des fils dans un nœud à partir du numéro de leur père. Soit un nœud de numéro n à la profondeur p, le numéro de son fils gauche est $n + 2^p$, le numéro de son fils droit est $n + 2^{p+1}$.*

Dans l'exemple suivant, le numéro de chaque nœud sera noté en-dessous de son étiquette.

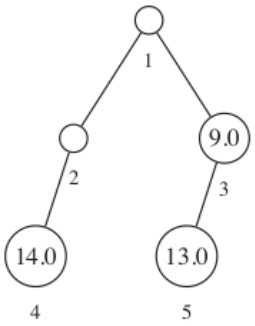
Exemple 7.

- Voici un arbre complet qui représente un vecteur plein de 7 éléments.



Le vecteur plein correspondant est : $\{1 \mapsto 11.0, 2 \mapsto 5.0, 3 \mapsto 9.0, 4 \mapsto 14.0, 5 \mapsto 13.0, 6 \mapsto 8.0, 7 \mapsto 4.0\}$.

- Voici un arbre partiel qui représente un vecteur creux de 3 éléments avec une valeur par défaut 0.0.



Le vecteur creux correspondant dont la valeur par défaut 0.0 des éléments est explicitée : $\{1 \mapsto 0.0, 2 \mapsto 0.0, 3 \mapsto 9.0, 4 \mapsto 14.0, 5 \mapsto 13.0, 6 \mapsto 0.0, 7 \mapsto 0.0\}$.

9. Montrer que l'ensemble des nœuds de profondeur p est égal à l'intervalle $[2^p, 2^{p+1} - 1]$. *Indication :* Ici on peut faire une vraie récurrence. Ne pas oublier une des deux inclusions.

   solution :

L'énoncé semble confondre un nœud avec son numéro. Pour ma part je noterai pour tout nœud n, i_n l'indice de n, et ce dans toute la suite de ce corrigé.

Soit a un arbre binaire partiel de profondeur h. Pour tout $p \in [\![0, h]\!]$, soit $P(p)$: « l'ensemble des nœuds de profondeur p est $[2^p, 2^{p+1} - 1]$ ».

- L'ensemble des nœuds de profondeur 0 est la racine, de numéro 1. Donc l'ensemble des numéros de nœuds de profondeur 0 est bien $[2^0, 2^1-]$.
- Soit $p \in [\![0, h[\![$, supposons $P(p)$.
  - ◇ Soit n un nœud de profondeur $p+1$, et soit m son père. Par hypothèse de récurrence, $2^p \leqslant i_m < 2^{p+1}$. De plus, i_n vaut $i_m + 2^n$ ou $i_m + 2^{p+1}$. Ainsi :

    ⧙ Notez la petite astuce qui m'évite de traiter à part les deux cas.

    $$\left.\begin{array}{l} i_m + 2^p \leqslant i_n \leqslant i_m + 2^{p+1} \\ \text{donc } 2^p + 2^p \leqslant i_n \leqslant 2^{p+1} - 1 + 2^{p+1} \\ \text{donc } 2^{p+1} \leqslant i_n < 2^{p+2} \end{array}\right\} \textit{par } P(p)$$

    Nous avons prouvé que l'ensemble des numéros des nœuds de profondeur $p+1$ est inclus dans $[\![2^{p+1}, 2^{p+2} - 1]\!]$.
  - ◇ Passons à l'autre inclusion.

    L'autre inclusion n'est manifestement vraie que si a est complet : erreur d'énoncé. Ci-dessous, je suppose a complet.

    Soit $i \in [\![2^{p+1}, 2^{p+2} - 1]\!]$.
    - ◦ Si $i < 2^{p+1} + 2^p$, posons $j = i - 2^p$. On a alors $j \in [\![2^{p+1} - 2^p, 2^{p+1} - 1]\!] = [\![2^p, 2^{p+1} - 1]\!]$. Dès lors par hypothèse de récurrence, il existe un nœud m dont le numéro est j. Comme l'arbre est complet de profondeur $> p$, m admet un fils gauche. Et celui-ci a pour indice i.
    - ◦ Si $i \in [\![2^{p+1} + 2^p, 2^{p+2} - 1]\!]$, procéder de même en posant $j = i - 2^{p+1}$.

  D'où $P(p+1)$.

Par le théorème de la récurrence, pour tout $p \in \mathbb{N}$, $P(p)$.

10. Montrer que la numérotation de chaque nœud est unique.

solution : Je fais la démonstration dans le cas où l'arbre est complet au moyen de la question précédente.

Soient n et m deux nœuds de même numéro. Soient p et q leurs profondeurs respectives. On a $i_n \in [\![2^p, 2^{p+1}[\![$ et $i_m \in [\![2^q, 2^{q+1}[\![$. Si $p < q$ ou si $p > q$, alors cet deux intervalles sont disjoints. Donc $p = q$.

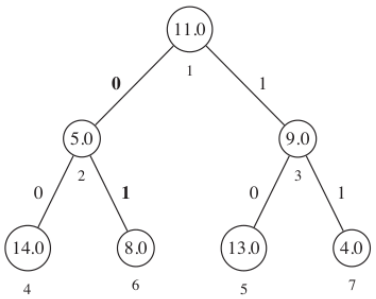
Ensuite la restriction de la fonction i de l'ensemble des nœuds de profondeur p vers $[\![2^p, 2^{p+1}[\![$ est bien définie et surjective par la question précédente. En outre ses ensembles de départ et d'arrivée ont même cardinal, elle est donc injective. Dès lors, notre hypothèse $i_n = i_m$ entraîne que $n = m$.

Si a n'est pas complet, il suffit de considérer a' obtenu en complétant a en rajoutant des nœuds étiquetés par 0 (par exemple) pour remplir chaque étage. On vérifie que cela ne change pas les numéro des nœuds déjà présents dans a.

3.2 Occurrence d'un nœud dans un arbre

Pour accéder à un nœud, nous devons représenter le chemin dans l'arbre allant de la racine au nœud. Pour cela, nous étiquetons la liaison entre un nœud et son fils gauche par l'entier 0 et la liaison entre un nœud et son fils droit par l'entier 1. Nous appelons alors occurrence, ou chemin, du nœud de numéro n la liste des étiquettes suivies pour aller de la racine à ce nœud. Dans l'exemple suivant, l'occurrence du nœud de numéro 6 étiqueté par la valeur 8.0 est $\langle 0, 1 \rangle$.

Exemple 8.



Si l'occurrence du nœud de numéro n situé à la profondeur p est notée $\langle c_1, ..., c_p \rangle$ (c_1 étant le lien avec la racine), on remarque que l'occurrence du père de n est $\langle c_1, ..., c_{p-1} \rangle$.

11. Soit un nœud de numéro n situé à la profondeur p et d'occurence $\langle c_1, ..., c_p \rangle$, montrer que :

$$n = 2^p + \sum_{i=0}^{p-1} c_{i+1} \times 2^i.$$

solution :

Petite astuce pour me simplifier les notations dans la récurrence. Je fixe p et je fait une récurrence le long du chemin.

Pour tout $k \in [\![0, p]\!]$, je note n_k le nœud dont le chemin est $\langle c_1, ..., c_k \rangle$. Ainsi pour tout $k \in [\![0, p[\![$, n_{k+} est un fils de n_k. En outre n_k est à la profondeur k.
Pour tout $k \in [\![0, p]\!]$, soit $P(k)$: « $i_{n_k} = 2^k + \sum_{i=0}^{k-1} c_{i+1} \times 2^i$ ».

- n_0 est la racine, qui a pour numéro 1. Or $2^0 + \underbrace{\sum_{i=0}^{-1} c_{i+1} \times 2^i}_{\text{somme vide, donc } 0} = 1$.

  D'où $P(0)$.
- Soit $k \in [\![0, p[\![$, supposons $P(k)$.

  Pour éviter de distinguer les deux cas, j'ai piqué cette astuce à Léo Samuel

  On remarque que $i_{n_{k+1}} = i_{n_k} + 2^k \times (c_{k+1} + 1)$. Ne reste plus qu'à faire un simple calcul :

$$\begin{aligned} i_{n_{k+1}} &= i_{n_k} + 2^k \times (c_{k+1} + 1) \\ &= 2^k + \sum_{i=0}^{k-1} c_{i+1} \times 2^i + 2^k c_{k+1} + 2^k \\ &= 2^{k+1} + \sum_{i=0}^{k} c_{i+1} \times 2^i \end{aligned} \quad \Big) \text{ par } P(k)$$

  d'où $P(k+1)$.

Ainsi, pour tout $k \in [\![0, p]\!]$, $P(k)$. En particulier $P(p)$ est vrai, et c'est ce qu'il fallait montrer.

12. Exprimer la valeur du coefficient c_i en fonction de n et i.
solution : On peut donc dire que pour tout $i \in [\![1, p]\!]$, c_i est le i-ème chiffre de l'écriture en base 2 de n. En formule, ce serait $\left\lfloor \frac{n}{2^{i-1}} \right\rfloor$ `mod 2` (`mod` étant de reste de la division euclidienne comme en Caml).

3.3 Représentation des arbres binaires partiels en CaML

Un arbre binaire partiel d'entiers est représenté par le type `arbre` dont la définition est :

```
type arbre =
| Vide
| Fourche of arbre * arbre
| Noeud of arbre * float * arbre;;
```

Dans l'appel `Noeud( fg, v, fd)`, les paramètres `fg`, `v` et `fd` dont respectivement le fils gauche, l'étiquette et le fils droit de la racine de l'arbre créé. Dans l'appel `Fourche( fg, fd)`, les paramètres `fg` et `fd` sont respectivement le fils gauche et le fils droit de la racine de l'arbre créé.

Exemple 9. L'expression suivante :

```
Fourche(
    Fourche( Noeud( Vide, 14.0, Vide), Vide),
    Noeud(Noeud( Vide, 13.0, Vide),
        9.0,
        Vide))
```

est alors associée à l'arbre binaire représenté graphiquement dans l'exemple 6.

13. Écrire en CaML une fonction `taille` de type `arbre -> int` telle que l'appel `(taille a)` renvoie le nombre de nœuds étiquetés contenus dans l'arbre binaire partiel `a`, c'est- à-dire le cardinal de $N(a)$. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre a. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.
solution :

```
let rec taille = function
  |Vide -> 0
  |Noeud (fg, _, fd) -> taille fg + taille fd +1
  |Fourche (fg, fd) -> taille fg + taille fd
;;
```

14. Une paire d'entiers est représentée par le type paire défini par : `type paire = int*int;;`.
Écrire en CaML une fonction bornes de type `arbre -> paire` telle que l'appel `(bornes a)` sur un arbre binaire partiel `a` non vide renvoie une paire d'entiers qui correspondent au plus petit, respectivement au plus grand, indice des valeurs contenues dans l'arbre `a`. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre `a`. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.
Par exemple, sur l'exemple 6, la fonction`bornes` devra renvoyer $(3, 5)$.
solution :

```
let bornes a =

  let rec aux i deux_puissance_p = function
    (* i : indice du œnud actuel
         deux_puissance_p : 2^p, où p est la profondeur du œnud actuel.
     *)
    | Vide -> max_int, min_int
    | Fourche(fg, fd) ->
       let (ming, maxg) = aux (i+deux_puissance_p) (2*deux_puissance_p) fg
       and (mind,maxd) = aux (i+2*deux_puissance_p) (2*deux_puissance_p) fd
       in (min ming mind, max maxg maxd)
    | Noeud(fg,_,fd) ->
       let (ming, maxg) = aux (i+deux_puissance_p) (2*deux_puissance_p) fg
       and (mind,maxd) = aux (i+2*deux_puissance_p) (2*deux_puissance_p) fd
       in
       (min_triple ming i mind), (max_triple maxg maxd i)

  in aux 1 1 a
;;
```

15. Écrire en CaML une fonction `remplaced` de type `int -> float -> arbre-> arbre` telle que l'appel `(remplaced i e a)` sur un arbre binaire partiel `a` non vide contenant une valeur pour l'indice i renvoie un arbre binaire partiel contenant la valeur e à l'indice i et les mêmes valeurs que l'arbre `a` pour les autres indices que i.
L'algorithme utilisé ne devra parcourir qu'une seule fois la branche de la racine jusqu'au nœud d'indice i. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.
solution :

```
œœ
let rec remplaced i e a =

  (* Cas d'arrêt *)
  if i=1 then match a with
              | Fourche(fg, fd) -> Noeud(fg, e, fd)
              | Noeud(fg, _, fd) -> Noeud(fg, e, fd)
              | Vide -> Noeud(Vide, e, Vide)

                            (* Cas récursifs *)
  else let c1 = i mod 2 and j = i/2 in
       (* Du coup, j est l'indice du œnud i dans le fils de la racine indiqué par c1 (par
    ↪ la question 11) *)
       if c1=0 then match a with
                    | Vide -> Fourche(remplaced j e Vide, Vide)
                    | Fourche(fg, fd) -> Fourche(remplaced j e fg, fd)
                    | Noeud(fg,x,fd) -> Noeud(remplaced j e fg, x, fd)
       else match a with
            | Vide -> Fourche(Vide, remplaced j e Vide)
            | Fourche(fg, fd) -> Fourche(fg, remplaced j e fd)
            | Noeud(fg,x,fd) -> Noeud(fg, x, remplaced j e fd)
;;
```

3.4 Codage d'un vecteur creux par un arbre partiel

L'arbre partiel utilisé dans les questions précédentes correspond à des vecteurs creux dont les valeurs par défaut sont 0.0. Ils ne contiennent donc que les valeurs différentes de 0.0. Cette représentation peut être étendue pour représenter des vecteurs creux avec d'autres valeurs par défaut. Les étiquettes contenues dans l'arbre partiel correspondent alors aux indices et valeurs des éléments du vecteur creux qui sont différentes de la valeur par défaut du vecteur. La valeur par défaut doit faire partie explicitement de la structure du vecteur creux.

Pour améliorer les performances d'accès, la structure de vecteur creux contient en plus de l'arbre partiel et de la valeur par défaut, les minimum et maximum des indices des valeurs contenues dans le vecteur qui sont différentes de la valeur par défaut ainsi que le nombre de valeurs différentes de la valeur par défaut.

Soit v un vecteur creux, nous noterons respectivement $\mathcal{V}_d(v)$, $\mathcal{V}_{min}(v)$, $\mathcal{V}_{max}(v)$, $\mathcal{V}_t(v)$ et $\mathcal{V}_a(v)$, la valeur par défaut, l'indice minimum, l'indice maximum, le nombre de valeurs différentes de la valeur par défaut et l'arbre partiel contenant les valeurs de v.

Définition 3.3. *(Vecteur creux bien formé)*
Un vecteur creux implanté avec un arbre binaire partiel est bien formé si et seulement si :

- *les indices minimum et maximum du vecteur creux correspondent aux bornes de l'arbre binaire partiel ;*
- *l'arbre binaire partiel ne contient pas la valeur par défaut du vecteur creux ;*
- *le nombre de valeurs contenues dans l'arbre binaire partiel est égal au nombre d'éléments différents de la valeur par défaut du vecteur creux ;*
- *l'arbre binaire partiel qui contient les valeurs ne doit pas contenir de fourches dont les sous-arbres gauche et droit sont tous deux vides.*

16. Exprimer la définition 3.3 sous la forme d'une formule logique $VBF(v)$ qui indique que v est un vecteur creux bien formé.
solution :

Un vecteur creux est représenté par le type vecteur dont la définition est :

```
type vecteur = int * int * int * float * arbre;;
```

Dans l'expression `( imin, imax, t, v, a)`, les paramètres `imin`, `imax`, `t`, `v` et `a` sont respectivement l'indice minimum, l'indice maximum et le nombre des éléments dont la valeur est différente de la valeur par défaut, ainsi que la valeur par défaut et l'arbre binaire partiel contenant les valeurs du vecteur.

17. Écrire en CaML une fonction `estValide` de type `vecteur -> bool` telle que l'appel `(estValide v)` renvoie la valeur `true` si le vecteur creux v est bien formé (c'est-à-dire si $VBF(v)$) et la valeur `false` sinon. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre associé à v. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.
solution :

```
(*(*(*
let estValide (imina, imaxa, n_different, defaut, a)=

  let rec aux i deux_puissance_p = function
    (* Renvoie le quadruplet ( est valide, nb de valeurs  defaut, indice min, indice max ) *)
    (* Mêmes arg supplémentaire que pour la fonction bornes.*)

    | Vide -> true, 0, max_int, min_int

    | Fourche(Vide, Vide) -> false,0, max_int, min_int

    | Fourche(fg, fd) ->
       let vg, ng, iming, imaxg = aux (i+deux_puissance_p) (2*deux_puissance_p) fg
       and vd, nd, imind, imaxd = aux (i+2*deux_puissance_p) (2*deux_puissance_p) fd in
       ( vg && vd,
         nd+ng,
         min iming imind,
         max imaxg imaxd
       )

    | Noeud(fg, e, fd) ->
```

```
105         let vg, ng, iming, imaxg  = aux (i+deux_puissance_p) (2*deux_puissance_p) fg
106         and vd, nd, imind, imaxd = aux (i+2*deux_puissance_p) (2*deux_puissance_p) fd in
107         ( vg && vd && e!=defaut,
108           nd+ng+1,
109           min_triple i imind imaxd,
110           max_triple i imaxg imaxd
111         )
112
113   in
114   let va, na, imin, imax = aux 1 1 a in
115   va && (imina, imaxa, n_different) = (imin, imax, na)
116 ;;
```

18. Écrire en CaML une fonction `lecture` de type `int -> vecteur -> float` telle que l'appel `(lecture i v)` renvoie la valeur qui se trouve à l'indice i dans le vecteur creux v. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre associé à v. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.
solution :

```
1 œœœ
123 let lecture i (imin, imax, na, defaut, a) =
124
125   (* On reprend la structure de la fonction remplaced *)
126   let rec aux i  a =
127     if i=1 then match a with
128                 | Fourche(fg, fd) -> defaut
129                 | Noeud(fg, e, fd) -> e
130                 | Vide -> defaut
131
132
133     else let c1 = i mod 2 and j = i/2 in
134          (* Du coup, j est l'indice du œnud i dans le fils de la racine indiqué par c1 (
       ↪ par la question 11) *)
135          if c1=0 then match a with
136                       | Vide -> defaut
137                       | Fourche(fg, fd) -> aux j  fg
138                       | Noeud(fg,_,fd) -> aux j  fg
139          else match a with
140               | Vide -> defaut
141               | Fourche(fg, fd) -> aux j  fd
142               | Noeud(fg,x,fd) -> aux j  fd
143
144   in aux i a
145 (* Souvent j'essaie de mettre un nom différent aux arguments de la fonction aux et de la
       ↪ fonction principale, mais pour une fois non *)
146 ;;
```

19. Écrire en CaML une fonction `ecriture` de type `int -> float -> vecteur -> vecteur` telle que l'appel `(ecriture i e v)`, avec la valeur de e différente de la valeur par défaut de v, renvoie un vecteur creux contenant la valeur e à l'indice i et les mêmes valeurs que le vecteur creux v pour les autres indices que i. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre associé à v. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.
solution :

```
1 œœœ œ
151 (*  Le sujet ne précise pas mais je soupçonne qu'il fallait conserver le caractère bien
       ↪ formé des vecteurs. Ça nous oblige à reprendre en l'adaptant le code de remplaced.
       ↪ *)
152
153 let ecriture i valeur (imin, imax, n_diff, defaut, a) =
154   (* Précondition : valeur   defaut *)
155
156   let rec aux i e a =
157     (* Renvoie l'arbre où la valeur d'indice i a été remplacée par e, ainsi que le nombre
       ↪ de nouvelles valeurs différentes de la valeur par défaut.*)
```

```
158
159   if i=1 then match a with
160                | Fourche(fg, fd) ->  (Noeud(fg, e, fd), 1)
161                | Noeud(fg, _, fd) -> (Noeud(fg, e, fd), 0)
162                | Vide ->  (Noeud(Vide, e, Vide), 1)
163
164   else let c1 = i mod 2 and j = i/2 in
165        (* j est l'indice du œnud i dans le fils de la racine indiqué par c1 (par la
       ↪ question 11) *)
166        if c1=0 then match a with
167                     (* descente à gauche *)
168                     | Vide -> let nfg, delta_diff = aux j e Vide in Fourche(nfg, Vide),
       ↪ delta_diff
169                     | Fourche(fg, fd) -> let nfg, delta_diff = aux j e fg in Fourche(nfg,
       ↪ fd), delta_diff
170                     | Noeud(fg,x,fd) ->let nfg, delta_diff = aux j e fg in Noeud(nfg, x,
       ↪ fd), delta_diff
171        else match a with
172             (* descente à droite *)
173             | Vide -> let nfd, delta_diff = aux j e Vide in Fourche(Vide, nfd), delta_diff
174             | Fourche(fg, fd) -> let nfd, delta_diff = aux j e fd in Fourche(fg, nfd),
       ↪ delta_diff
175             | Noeud(fg,x,fd) -> let nfd, delta_diff = aux j e fd in  Noeud(fg, x, nfd),
       ↪ delta_diff
176
177   in
178   let (na, delta_diff) = aux i valeur a in
179   (min i imin, max i imax, n_diff+delta_diff, defaut, na)
180 ;;
```

20. Écrire en CaML une fonction `somme` de type `vecteur -> vecteur -> vecteur` telle que l'appel `(somme v1 v2)` renvoie un vecteur creux dont les éléments ont pour valeur la somme des valeurs des éléments de `v1` et de `v2` de même indice. L'algorithme utilisé ne devra parcourir qu'une seule fois les arbres associés à $v1$ et à $v2$. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.
solution : Je décide que la valeur par défaut du résultat sera celle du premier vecteur.
Ma fonction aux renverra également le nombre de valeurs différentes du défaut ainsi que les bornes.
Enfin pour alléger le code je crée une fonction intermédiaire chargée de sommer deux nœuds une fois qu'on lui fournit les deux étiquettes et les quatre fils à assembler.

```
1 œœœ œ œ
187 let somme (imina, imaxa, _, defauta, a) (iminb, imaxb, _, defautb, b) =
188   (* Je prends defauta comme valeur par défaut du résultat *)
189
190   let rec somme_noeud i x y ag ad bg bd dpp =
191     (* Fonction pour faire la somme à un œnud *)
192     (* x et y : valeur à sommer à la racine
193        ad, ag, bd, bg : sous arbres à sommer puis mettre en dessous
194        Sortie : (imin, imax, n_diff, arbre somme)
195      *)
196
197     let  (iming, imaxg, n_diffg, fg) = aux (i + dpp ) ag bg (2*dpp)
198     and (imind, imaxd, n_diffd, fd) = aux (i + dpp*2 ) ad bd (2*dpp)
199     in
200
201     if x +. y = defauta then
202        (* Prenons garde à éviter Fourche(Vide,Vide) *)
203       if (fg, fd) <> (Vide, Vide) then
204         ( min iming imind, max imaxg imaxd, n_diffg + n_diffd, Fourche(fg, fd))
205       else
206         (max_int, min_int, 0, Vide)
207     else
208       if (fg, fd) <> (Vide, Vide) then
209         (i, max imaxd imaxg, n_diffg+n_diffd+1, Noeud(fg, x+.y, fd))
210       else
```

```
211         (i,i,0, Noeud(Vide, x+. x, Vide))
212
213
214   and aux i _a _b dpp =
215     (* i : indice du œnud actuel
216        dpp : 2^p, où p est la profondeur actuelle
217      Sortie : (imin, imax, n_diff, arbre somme) *)
218     match _a, _b with
219     |Vide, Vide -> (max_int, min_int, 0, Vide)
220     |Vide, Fourche(bg, bd) -> somme_noeud i defauta defautb Vide Vide bg bd dpp
221     |Vide, Noeud(bg, be, bd) -> somme_noeud i defauta be Vide Vide bg bd dpp
222     |Fourche(ag, ad) , Fourche(bg, bd) -> somme_noeud i defauta defautb ag ad bg bd dpp
223     |Fourche(ag, ad), Noeud(bg, e, bd) -> somme_noeud i defauta e ag ad bg bd dpp
224     |Noeud(ag, ae, ad), Noeud(bg, be, bd) -> somme_noeud i ae be ag ad bg bd dpp
225     |_ -> aux i _b _a dpp
226
227   in
228   let (imin, imax, n_diff, a_somme) = aux 1 a b 1 in
229   (imin, imax, n_diff, defauta, a_somme)
230 ;;
```