Premier devoir surveillé d'option informatique : tri par tas (CCP 2015 revu et corrigé)

16/10/19

Si vous repérez ce qui vous paraît une erreur d'énoncé, indiquez-le sur votre copie et précisez les initiatives que vous avez été amenés à prendre. Vous pouvez coder toute fonction complémentaire qui vous semble utile. Dans ce cas indiquez précisément le rôle de cette fonction, la signification de ses paramètres et la nature de la valeur renvoyée.Durée trois heures. Il est possible de rendre un complément jeudi. Le symbole « $\wedge$ » signifie « et », et le symbole « $\vee$ » signifie « ou ».

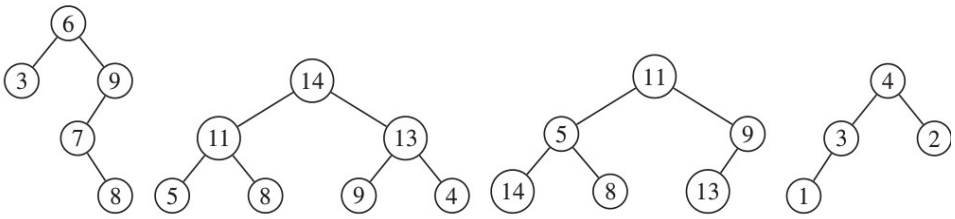
1 Arbres binaires d'entiers

Définition 1.1. *(Arbre binaire d'entiers)*
Un arbre binaire d'entiers a est une structure qui peut, soit être vide (notée $\varnothing$), 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. La taille de l'arbre a, notée $|a|$ est le nombre de nœuds de l'arbre a.

Cette définition s'exprime sous la forme de la propriété $\mathcal{B}(a)$ qui caractérise un arbre binaire d'entiers a :

$$\mathcal{B}(a) \Leftrightarrow (a = \varnothing) \vee a \left(\neq \varnothing \wedge \mathcal{B}(\mathcal{G}(a)) \wedge \mathcal{B}(\mathcal{D}(a)) \wedge \mathcal{E}(a) \in \mathbb{N}\right).$$

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



1.1 Implémentation en langage CaML

Un tel arbre est représenté en langage CaML 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 binaire d'entiers créé.

Exemple 2. L'expression suivante est associée au premier arbre binaire représenté graphiquement dans l'exemple 1.

```
Noeud(
    Noeud( Vide, 3, Vide)
    6,
    Noeud(
```

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

1. Donner l'expression en langage CaML qui correspond au quatrième arbre binaire de l'exemple 1.
solution :

```
    Noeud(
         Noeud( Noeud(Vide, 1, Vide)
         ,3,
         Vide)
    ,4,
    , Noeud(Vide, 2, Vide)
    )

```

1.2 Hauteur dans un arbre binaire

Définition 1.2. *Soit a un arbre binaire et n un nœud de a. La hauteur de n dans a est égale au nombre de nœuds du chemin sans cycle le plus long reliant n à un sous-arbre vide. Nous la noterons $\eta(n)$. Si n est la racine de l'arbre ($n = a$), il s'agit alors de la hauteur de l'arbre. Nous associerons la hauteur 0 à l'arbre binaire vide $\varnothing$.*

solution : On notera que la définition de la hauteur dans ce sujet n'est pas la même que dans le cours : ici les arbres auront une hauteur de hauteur 1 de plus que dans le cours.

Exemple 3. Les hauteurs des quatre arbres binaires de l'exemple 1 sont respectivement 4, 3, 3 et 3.

1. Donner une définition mathématique de la hauteur $\eta(a)$ d'un arbre binaire a en fonction de $\varnothing$, $\mathcal{G}(a)$ et $\mathcal{D}(a)$.
solution : On peut définir la fonction η ainsi :
   - $\eta(\varnothing) = 0$
   - Pour tout arbre binaire a, $\eta(a) = 1 + \max\big(\eta(\mathcal{G}(a)), \eta(\mathcal{D}(a))\big)$.
2. Soit a un arbre binaire de hauteur h. Quel est son nombre minimum et son nombre maximum de nœuds ? On justifiera le résultat par récurrence sur h.
solution : Notons pour tout $h \in \mathbb{N}$, $P(h)$: « un arbre binaire de hauteur h a au moins h, et au plus $2^h - 1$ nœuds.
   - *Initialisation :* Soit $h = 0$. Soit a un arbre de hauteur h, c'est donc $\varnothing$, et il a 0 nœuds. Or, $0 = h = 2^h - 1$, donc $P(0)$.
   - *Hérédité :* Soit $h \in \mathbb{N}$. Supposons que pour tout $k \in [\![0, h]\!]$, $P(k)$, et prouvons $P(h+1)$. Soit donc a un arbre de hauteur $h+1$.
     Un de ses deux fils est de hauteur h, il a donc par $P(h)$ au moins h nœuds. Compte tenu de la racine, cela fait au moins $h+1$ nœuds dans a.
     Ses deux fils sont de hauteur au plus h. Donc par l'hypothèse de récurrence forte, ils ont chacun au plus $2h-1$ nœuds. Compte tenu de la racine, cela fait au plus $1 + 2 \times (2^h - 1)$ nœuds, ce qui fait $2^{h+1} - 1$ nœuds.
     D'où $P(h+1)$.

   Par récurrence, pour tout $h \in \mathbb{N}$, $P(h)$.

1.3 Profondeur d'un nœud et niveau dans un arbre binaire

Définition 1.3. *(Profondeur d'un nœud, niveau dans un arbre binaire)*

- *Soit un arbre binaire a contenant un nœud n, la profondeur du nœud n dans a, notée $\pi(n)$ est définie par :*

$$\pi(n) = \eta(a) - \eta(n).$$

- *Un niveau dans un arbre binaire a est la séquence de gauche à droite des nœuds de même profondeur dans a.*

Exemple 4. Les étiquettes des nœuds de même profondeur dans les quatre arbres de l'exemple 1 sont :

Profondeur	Arbre 1	Arbre 2	Arbre 3	Arbre 4
0	6	14	11	4
1	3, 9	11, 13	5, 9	3, 2
2	7	5, 8, 9, 4	14, 8, 13	1
3	8			

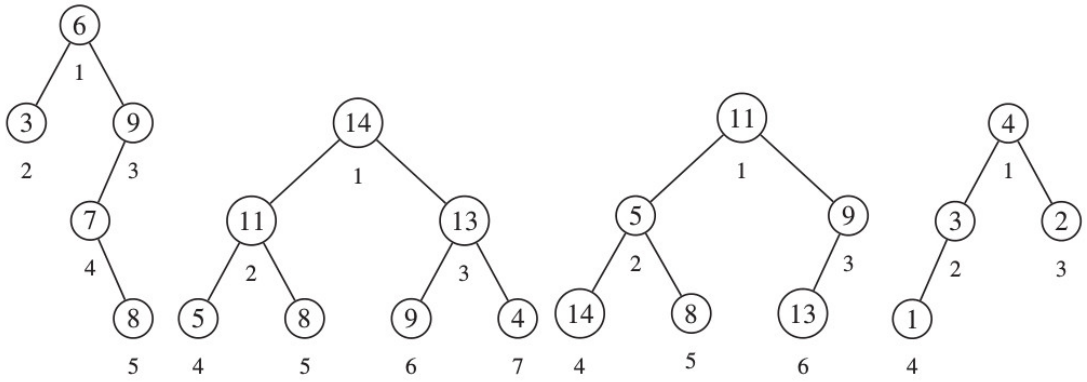
2 Numérotation hiérarchique des nœuds d'un arbre binaire

La numérotation hiérarchique des nœuds d'un arbre binaire a consiste à associer à chaque nœud un numéro compris entre 1 et $|a|$ par un parcours en largeur partant de la racine (numéro 1) et en parcourant chaque niveau de gauche à droite jusqu'au dernier nœud : le plus profond et le plus à droite (numéro $|a|$). Nous noterons $\mathcal{N}_i(a)$ le nœud de l'arbre binaire a de numéro i avec $i \in [\![1, |a|]\!]$.

Dans les exemples suivants, le numéro de chaque nœud sera noté en-dessous de son étiquette.

Exemple 5. (Numérotation hiérarchique)

La numérotation hiérarchique des nœuds des quatre arbres de l'exemple 1 produit les arbres numérotés suivants (les sous-arbres vides qui sont fils gauche ou droit des nœuds ne sont pas représentés) :



1. Écrire en CaML une fonction `lire` dont le type est `int -> arbre -> int` telle que l'appel `(lire i a)` sur l'arbre binaire d'entiers a avec $i \in [\![1, |a|]\!]$ doit renvoyer l'entier $\mathcal{E}(\mathcal{N}_i(a))$. Cette fonction devra au plus parcourir une seule fois chaque élément de l'arbre a. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.

   Indication : Puisque l'énoncé ne propose pas d'implémenter des files d'attentes, et qu'il n'y a pas de contrainte de complexité, vous pouvez utiliser une simple liste en guise de file. Ou si vous préférez le module `Queue` de Caml.

   solution :

```
let rec lireAux i f=
    (* f est une liste utilisée comme file d'attente*)
    match f with
        | []    -> failwith "index out of range :("
        | Vide :: q -> lireAux i q
        | Noeud(_,e,_)::q when i=1 -> e
        | Noeud(fg,_,fd)::q -> lireAux (i-1) (q@[fg;fd])
;;

let lire i a=
    lireAux i [a]
;;
```

3 Arbre binaire partiellement ordonné d'entiers

Définition 3.1. *(Arbre binaire partiellement ordonné d'entiers)*

Un arbre binaire d'entiers a est partiellement ordonné si il est vide ou :

- *les fils gauche et droit de a sont des arbres binaires partiellement ordonnés d'entiers ;*
- *les étiquettes de tous les nœuds composant les fils gauche et droit de a sont inférieures ou égales à l'étiquette de a.*

Cette définition s'exprime sous la forme de la propriété $\mathcal{O}(a)$ qui caractérise les arbres binaires partiellement ordonnés d'entiers a :

$$\mathcal{O}(a) \equiv (a = \varnothing) \vee \Big(a \neq \varnothing \ \wedge \ \mathcal{O}(\mathcal{G}(a)) \ \wedge \ \mathcal{O}(\mathcal{D}(a)) \ \wedge \ \mathcal{E}(\mathcal{G}(a)) \leqslant \mathcal{E}(a) \ \wedge \ \mathcal{E}(\mathcal{D}(a)) \leqslant \mathcal{E}(a) \Big).$$

Exemple 6. (Arbres binaires partiellement ordonnés d'entiers)
Le deuxième et le quatrième arbres de l'exemple 1 sont des arbres binaires partiellement ordonnés d'entiers.

1. Écrire en CaML une fonction vérifier dont le type est `arbre -> bool` telle que l'appel `(vérifier a)` sur l'arbre binaire d'entiers a renvoie la valeur `true` si $\mathcal{O}(a)$ et la valeur `false` sinon. Cette fonction devra au plus parcourir une seule fois chaque nœud de l'arbre a. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.
   solution :

```
let rec verifierAux eMax a=
    (* eMax indique l'étiquette max autorisée *)
    match a with
        |Vide   -> true
        |Noeud(fg,e,fd) -> e <= eMax && verifierAux e fg && verifierAux e fd
;;

let verifier a =
    match a with
        | Vide  -> true
        | Noeud(fg,e,fd) -> verifierAux e fg && verifierAux e fd
;;
```

4 Arbres binaires complets

Définition 4.1. *(Arbre binaire complet)*
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).

Cette définition s'exprime sous la forme de la propriété $\mathcal{C}_n(a)$ qui caractérise les arbres binaires complets a de hauteur n ($\eta(a) = n$) :

$$\mathcal{C}_n(a) \equiv (a = \varnothing \wedge n = 0) \vee (a \neq \varnothing \wedge n \neq 0 \wedge \mathcal{C}_{n-1}(\mathcal{G}(a)) \wedge \mathcal{C}_{n-1}(\mathcal{D}(a))).$$

Exemple 7. Le deuxième arbre de l'exemple 1 est complet.

1. Montrer que, dans un arbre binaire complet non vide a, le niveau de profondeur p contient 2^p nœuds. On pourra raisonner par récurrence sur p.
   solution : Soit a un arbre binaire complet de hauteur h. Notons pour tout $k \in [\![0, h-1]\!]$, $P(k)$: le niveau de profondeur k a au plus 2^k nœuds.
   - *Initialisation :* le niveau de profondeur 0 contient uniquement la racine, ce qui fait un nœuds. Or $2^0 = 1$, d'où $P(0)$.
   - *Hérédité :* Soit $k \in [\![0, h-2]\!]$, supposons $P(k)$. Il y a donc 2^k nœuds au niveau k. Or chacun de ces nœuds a deux fils non vides, car $k < h-1$, donc k n'est pas le dernier niveau de a.. Cela fait 2×2^k,c'est-à-dire 2^{k+1} nœuds au niveau $k+1$. Donc $P(k+1)$.

   Ainsi, pour tout $k \in [\![0, h-1]\!]$, $P(k)$.
2. Calculer le nombre n de nœuds d'un arbre binaire complet non vide a de hauteur $p = \eta(a)$.
   solution : Vu la question précédente, un arbre complet de hauteur p a $\displaystyle\sum_{k=0}^{p-1} 2^k$ nœuds, ce qui fait $2^p - 1$ nœuds.
3. En déduire la hauteur d'un arbre binaire complet non vide a contenant n éléments ($n = |a|$).
   solution : Soit a un arbre binaire complet non vide ayant n éléments. Notons h sa hauteur. Par la question précédente, $n = 2^h - 1$. Donc $h = \log_2(n+1)$.

5 Arbres binaires parfaits

Définition 5.1. *(Arbre binaire parfait)*
Un arbre binaire parfait est un arbre binaire dont tous les niveaux sont complets sauf le niveau le plus profond qui peut être incomplet auquel cas ses nœuds sont alignés à gauche de l'arbre.

Pour tout $n \in \mathbb{N}$ et tout arbre binaire a, nous noterons $\mathcal{P}_n(a)$ le booléen « a est parfait de hauteur n ».

Exemple 8. Le troisième et le quatrième arbres de l'exemple 1 sont parfaits. Le deuxième est également parfait car il est complet.

Soit le type énuméré `categorieArbre` en langage CaML distinguant les arbres binaires complets, parfaits et quelconques :

```
type categorieArbre = Complet| Parfait| Quelconque;;
```

1. Soit $n \in \mathbb{N}$ et a un arbre binaire. Expliquer par des dessins pourquoi $P_n(a)$ peut être décomposé ainsi :

$$\mathcal{P}_n(a) = \mathcal{C}_n(a) \vee \Bigg(a \neq \varnothing \wedge n \neq 0 \wedge \Bigg(\begin{array}{ll} & n \neq 1 \wedge \mathcal{P}_{n-1}(\mathcal{G}(a)) \wedge \mathcal{C}_{n-2}(\mathcal{D}(a)) \\ \vee & n \neq 1 \wedge \mathcal{C}_{n-1}(\mathcal{G}(a)) \wedge \mathcal{C}_{n-2}(\mathcal{D}(a)) \\ \vee & \mathcal{C}_{n-1}(\mathcal{G}(a)) \wedge \mathcal{P}_{n-1}(\mathcal{D}(a)) \Bigg) \Bigg)$$

2. Écrire en CaML une fonction analyser dont le type est `arbre -> int * categorieArbre` telle que l'appel `(analyser a)` sur l'arbre binaire a renvoie un couple contenant la hauteur $\eta(a)$ de l'arbre a ainsi que la valeur `Complet` si a est complet, sinon la valeur `Parfait` si a est parfait, sinon la valeur `Quelconque`.
Cette fonction devra au plus parcourir une seule fois chaque nœud de l'arbre a. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives. *Indication :* Utiliser la formule logique précédente.

solution :

```
(* L'énoncé donnait en fait une grosse indication : faire une fonction qui renvoie é
    ↪ galement la hauteur de l'arbre.
En effet, c'est le moyen le plus simple pour écrire une fonction qui détermine si un
    ↪ arbre est complet ou parfait*)
let rec analyser a=
    match a with
        | Vide  -> 0, Complet (* Dans la définition de ce sujet, Vide est de hauteur
    ↪ 0 *)
        | Noeud(fg,_,fd)    -> let hg, cg = analyser fg and hd, cd = analyser fd in
                begin
                match cg,cd with
                    | Complet, Complet when hg=hd   -> hg+1, Complet
                    | Complet, Complet when hg=hd+1 -> hg+1, Parfait
                    | Parfait, Complet when hg=hd+1 -> hg+1, Parfait
                    | Complet, Parfait when hg=hd   -> hg+1, Parfait
                    | _      -> 1 + max hg hd, Quelconque
                end
;;
```

3. Soit a un arbre parfait et n le numéro d'un de ses nœuds. On suppose que ce nœud admet au moins un fils gauche. Soit également p la profondeur de ce nœud.
   (a) Quel est le numéro du premier nœud du niveau p ?
   solution : Avant le niveau p, il y a un arbre complet de hauteur p, qui contient donc $2^p - 1$ nœuds, numérotés de 1 à $2^p - 1$. Donc le premier nœud de profondeur p a pour numéro 2^p.
   (b) Calculer le nombre de nœuds qui se trouvent à gauche du nœud de numéro n dans le niveau de profondeur p.
   solution : Il y a $n-1$ nœud avant le nœud numéro n. Parmi eux, $2^p - 1$ sont dans des niveaux inférieurs. Il y en a donc $n - 2^p$ dans le niveau p.
   (c) Dans le niveau de profondeur $p+1$ de a, quel est le nombre de nœuds qui se trouvent à la gauche des fils du nœud de numéro n ?
   solution : Il s'agit de compter les fils des nœuds à gauche de celui de numéro n. Comme a est parfait, chaque nœud à gauche du nœud numéro n a deux fils. Ceci nous fait donc $2 \times (n - 2^p)$ nœuds à gauche des fils du numéro n. Ce qui vaut $2n - 2^{p+1}$.

(d) Soit un nœud de numéro n d'un arbre binaire parfait, calculer les numéros de ses fils gauche et droit.
solution : Soit g le fils gauche du nœud numéro n. Avant g, il y a :

- $2^{p+1} - 1$ dans les niveaux précédents ;
- $2n - 2^{p+1}$ dans le niveau $p+1$ d'après la question précédente.

Ceci fait un total de $2n-1$ nœud avant g, qui est donc de numéro $2n$. Et le fils droit du nœud numéro n vient juste après g, il a donc pour numéro $2n+1$.

4. Soit $n \in [\![2, \infty[\![$. Déduire de la question précédente, le numéro du père du nœud de numéro n dans un arbre binaire parfait.
solution : Soit k le numéro cherché. D'après la question précédente, $n = 2k$ ou $n = 2k+1$. Dans les deux cas, $k = \left\lfloor \frac{n}{2} \right\rfloor$. Qui sera noté en Caml `n/2` et en Python `n//2`.
5. Dessiner un arbre parfait à 10 nœuds, et pour chaque nœud, écrire son numéro selon la numérotation hiérarchique, mais écrire ce numéro en base 2.
6. Soit a un arbre binaire et n le numéro d'un de ses nœuds. Soit $\overline{a_0 \dots_a k}^2$ l'écriture de n en base 2. Quelle est l'écriture en base deux du numéro de son fils gauche (s'il existe) ? Et pour son fils droit ?
solution : Le fils gauche a pour numéro $2n$ qui s'écrit $\overline{a_0 \dots_a k0}^2$ en base deux. Pour le fils droit, c'est $\overline{a_0 \dots_a k1}^2$.
7. Écrire une fonction `decompBase2` de type `int -> int list` prenant en entrée un entier n et renvoyant sa décomposition en base 2, le bit de poid fort en tête de liste. Par exemple `decompBase2 10` renverra `[1;0;1;0]`.
solution :

```
let rec decompBase2Aux accu n=
    (* Renvoie "écriture de n en base 2 @ accu"   *)
    if n=0 then accu
    else let r= n mod 2 and q= n/2 in
        decompBase2Aux (r::accu ) q;;

let decompBase2 = decompBase2Aux [];;
```

8. Écrire en CaML une fonction `lire` dont le type est `int -> arbre -> int` telle que pour tout arbre binaire parfait a et tout $i \in [\![1, |a|]\!]$ l'appel `(lire i a)` renvoie l'entier $\mathcal{E}(\mathcal{N}_i(a))$.
Cette fonction devra au plus parcourir une seule fois chaque élément de la branche qui conduit de la racine de a au nœud de numéro i. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.
solution :

```
let rec lireAux l a=
    match l, a with
        | [], Noeud(_,e,_)  -> e
        | 1::q, Noeud(_,_,fd)   -> lireAux q fd
        | 0::q, Noeud(fg, _, _) -> lireAux q fg
        | _, Vide    -> failwith "indice trop grand"
        | x, _  -> failwith "écriture en base 2 douteuse"
;;
let lire n a=
  (* Il faut enlever le premier 1 de la liste. *)
  lireAux (List.tl(decompBase2 n)) a;;
```

6 Arbres en tas

Définition 6.1. *Un arbre en tas est un arbre binaire parfait partiellement ordonné.*

solution : Par rapport au cours, la définition ici impose que l'arbre soit parfait (c'est-à-dire complet gauche).

Cette définition s'exprime sous la forme de la propriété $\mathcal{T}_n(a)$ qui caractérise les arbres en tas a de hauteur n :

$$\mathcal{T}_n(a) \equiv \mathcal{P}_n(a) \wedge \mathcal{O}(a).$$

Exemple 9. Le deuxième et le quatrième arbres binaires de l'exemple 1 sont en tas.

Lorsqu'un arbre binaire parfait n'est pas partiellement ordonné, les étiquettes des nœuds peuvent être permutées pour obtenir un arbre en tas sans changer la structure d'arbre binaire parfait.

6.1 Implantation en langage CaML

1. Écrire une fonction `insereALaPlace` de type `int -> int -> arbre -> arbre` telle que l'appel `insereALaPlace i x a`, où i est le numéro d'un nœud vide de a, renvoie un arbre obtenu en rajoutant x dans a, et en faisant en sorte que le nœud supplémentaire rempli soit le numéro i.
   Cette fonction devra parcourir uniquement la branche allant de la racine vers l'emplacement numéro i de a.
   Indication : S'inspirer de la fonction `lire` de la question 8 partie 5.

   solution :

```
let rec insereAux l a x=
    (* Insère x à la place indiquée par l. Comme avant, l est l'écriture en base 2 du
    ↪  numéro du nœud où aller, privée de son premier 1.*)
    match l, a with
        | [], Vide  -> Noeud(Vide, x, Vide)
        | [], _     -> failwith "le numéro indiqué n'était pas celui d'une place vide
    ↪ "

        (* Descendre à gauche *)
        | 1::q, Noeud(fg,e,fd) when x>e  -> Noeud(fg, x,  insereAux q fd e)
        | 1::q, Noeud(fg,e,fd)           -> Noeud(fg, e,  insereAux q fd x)

        (* Descendre à droite *)
        | 0::q, Noeud(fg,e,fd) when x>e  -> Noeud(insereAux q fg e, x, fd )
        | 0::q, Noeud(fg,e,fd)           -> Noeud(insereAux q fg x, e, fd )

        | _, Vide    -> failwith "indice trop grand"
        | x, _  -> failwith "écriture en base 2 douteuse"
;;

let insereALaPlace i x t=
    insereAux (List.tl(decompBase2 i)) t x;;
```

2. En déduire une fonction `construire` de type `int list -> arbre` qui prend une liste `l` en entrée et renvoie un tas contenant les éléments de `l`.
   solution :

```
let rec construireAux  i t l=
    (* t est un tas de (i-1) éléments. Ceci renvoie le tas obtenu en rajoutant les él
    ↪ éments de l dans t.*)
    match l with
        |[]     -> t
        |x::q   -> construireAux  (i+1) (insereALaPlace i x t) q
;;

let construire  = construireAux 1 Vide;;
```

3. Quelle est l'amélioration apportée par cette fonction par rapport à la fonction `tas_of_list` vue en cours ?
   solution : Cette fonction crée un tas parfait (au sens de ce sujet de concours, c'est-à-dire "complet gauche" dans le cours), alors que dans la fonction créé en cours, les feuilles du dernier étage ne sont pas forcément toutes alignés à gauche.
   Le fait de savoir exactement la forme de l'arbre permettra en outre de déconstruire l'arbre en faisant en sorte qu'il reste toujours parfait, ce qui optimisera sa hauteur pendant cette phase.

6.2 Implantation par un tableau en langage Python

La structure d'arbre parfait peut être implantée très efficacement par un tableau contenant les étiquettes de l'arbre selon une numérotation hiérarchique des nœuds de l'arbre. Un arbre a de n nœuds sera représenté par un tableau `t` tel que pour tout $i \in [\![1, n]\!]$, `t[i]` contient l'étiquette du nœud numéro i de a.
N.B. Bien que cela ne soit pas optimal, pour coller au mieux à l'implantation persistante, et en particulier conserver la même relation entre le numéro d'un nœud et ceuc de ses fils, nous continuons d'utiliser une numérotation des nœuds selon un parcours en largeur en commençant à 1. La case d'indice 0 des tableaux ne sera alors pas utilisée dans tout ce problème. Ainsi, si `t` représente un arbre de n nœuds, on aura `n=len(t)-1`.

Exemple 10. (Arbres binaires parfaits représentés par un tableau)

Les deuxième, troisième et quatrième arbres de l'exemple 1 sont parfaits. Ils sont représentés par les tableaux suivants (les entiers des lignes supérieures sont les numéros des nœuds et les entiers des lignes inférieures correspondent aux valeurs de leurs étiquettes) :

1	2	3	4	5	6	7
14	11	13	5	8	9	4

1	2	3	4	5	6
11	5	9	14	8	13

1	2	3	4
4	3	2	1

Dans le premier tableau, le sous-arbre dont la racine a pour numéro 3 est le sous-arbre formé des nœuds numéros 3, 6, et 7, qui serait implémenté en Caml par `Noeud( Noeud(Vide, 9, Vide), 13, Noeud(Vide, 4, ))`.

Dans certaines situation, seule une partie du tableau `t` représentera un arbre. Dans ce cas, un entier `l` précisera le nombre d'éléments de cet arbre. Alors les cases `t[l+1]` et suivantes ne correspondront pas à des éléments de l'arbre.

1. Dessiner un tas contenant les entiers 5,4, 9, et 6 et donner une représentation Python de ce tas.
2. Écrire une fonction Python `placer` prenant en entrée trois arguments `a`, `l`, et `r` vérifiant :
   - `a` est un tableau représentant un arbre parfait ;
   - `l` est le nombre d'éléments de cet arbre ;
   - `r` est le numéro d'un nœud n tel que $\mathcal{D}(n)$ et $\mathcal{G}(n)$ sont en tas ;

   et permutant certaines cases de `a` pour que n devienne lui aussi un tas.
   Cette fonction devra au plus parcourir une branche de l'arbre binaire parfait représenté par a. Cette fonction ne devra pas être récursive ni faire appel à des fonctions auxiliaires récursives.
   solution :

```
def placer(a,r,l):
    """ Entrée : a un arbre dans un tableau
                 r indice d'un nœud dont les deux fils sont des tas
                 l nombre d'éléments de l'arbre. Donc la dernière case utile du tableau est a[l]"""

    i=r #indice du noeud courant
    # On continue tant que le nœud courant a des fils, et que l'étiquette actuellement en i est mal placée.
    while 2*i+1 <= l and a[i]< max(a[2*i], a[2*i+1]):  #on nous interdit la récursivité :(
        if a[2*i]> a[2*i+1]:
            #descendre le noeud i en 2i
            permute(a,i,2*i)
            i=2*i
        else:
            permute(a,i,2*i+1)
            i=2*i+1

    #peut-être faut-il encore descendre i en 2*i
    if 2*i<=l and a[2*i]> a[i]:
            permute(a,i,2*i)
            i=2*i
```

3. En déduire une fonction `entasser` qui prend en entrée un tableau `a` représentant un arbre parfait quelconque et qui en permute certaines cases pour transformer l'arbre contenu dans a en un arbre en tas.
   solution :

```
def entasser(a):
    n=len(a)
    for i in range(n-1,0,-1): # on veut aller de n-1 à 1 (inclus) à reculons
        placer(a,i,n-1)

```

4. Préciser la complexité de `entasser`.
   solution : Soit a un arbre de n nœuds. Comme a est parfait, sa hauteur est en $O(\log n)$. Ainsi, chaque appel à `placer` s'exécute en $O(\log n)$ puisqu'une seule branche est parcourue.

Dès lors, la complexité de `entasser` est en $O(n \log n)$.

Remarque : La bibliothèque `heapq` de Python implémente les tas. (Un tas s'appelle « a heap » en anglais. Le « q » est pour « queue », c'est-à-dire « file », puisqu'un tas implémente une file de priorité.) La fonction ci-dessus s'appelle `heapify`. Elle est cependant plus efficace que la version naïve suggérée ci-dessus puisqu'elle est de complexité linéaire. Voir l'exercice « construire des tas » de la feuille de TD. https://docs.python.org/3.0/library/heapq.html

7 Tri par tas

La structure d'arbre en tas permet d'implanter un algorithme de tri efficace appelé tri par tas. La structure d'arbre binaire partiellement ordonné assure que l'étiquette de la racine de l'arbre en tas contient la plus grande étiquette du tas. L'algorithme répète l'étape suivante jusqu'à ce que le tas soit vide :

- l'étiquette de la racine est placée à la fin de la séquence triée ;
- le dernier nœud du tas selon la numérotation hiérarchique est enlevé du tas. Son étiquette remplace celle de la racine. L'arbre binaire ainsi obtenu est parfait. Les sous-arbres gauche et droit de sa racine sont des arbres en tas ;
- l'arbre binaire parfait ainsi obtenu est ensuite converti en arbre en tas en utilisant la fonction `placer`.

7.1 Implantation en langage Python

On propose de réaliser un tri par tas en place. Notons `t` le tableau à trier. Pour simplifier, nous supposerons que la case d'indice 0 n'est pas à trier : les éléments utiles sont indicés à partir de 1.

Pendant le vidage du tas, nous utiliserons une variable `l` et l'invariant de boucle suivant sera maintenu :

- le tableau `t` (ou plutôt `t[1:]`) contient les mêmes éléments qu'avant le lancement de la procédure.
- les éléments de `t[1:l+1]` forment un tas ;
- Les éléments de `t[l+1:]` sont triés dans l'ordre croissant et sont les plus grands éléments de `t` ;

1. Détailler les étapes de l'algorithme du tri par tas pour le quatrième arbre de l'exemple 1. Pour chaque étape, donner les représentations du tas sous les formes d'arbre et de tableau.
2. Écrire en Python une procédure `trier` qui permute le contenu de certaines cases d'un tableau `s` pour trier ce tableau selon l'algorithme du tri par tas.
   Cette fonction ne devra pas être récursive ni faire appel à des fonctions auxiliaires récursives.
   solution :

```
def trier(s):
    """ Trie s en place, sauf la case 0 qu'on ignore"""
    entasser(s)
    n=len(s)
    for f in range(n-1,0,-1):
        # invariant de boucle : s[f+1:] contient les plus gros éléments de s et dans
    ↪ le bon ordre
        #                       s[1:f+1] est un tas
        permute(s,1,f) # on mets le plus gros élément restant en case f
        placer(s,1,f-1) # on remet tout en tas
```

3. Quelle est la complexité de ce tri ? A-t-il des atouts face au tri fusion ?
   solution : Soit `t` un tableau de longueur n. On a déjà dit que `entasser(t)` a une conplexité en $O(n \log n)$.
   Ensuite, on effectue encore n appels à `placer` qui ont chacun un coût en $\log(n)$.
   D'ou un coût final en $O(n \log)$.
   Un avantage de ce tri face au tri fusion est qu'il est en place : il ne nécessite donc aucune mémoire supplémentaire !

7.2 Implantation en langage CaML

1. Écrire en CaML une fonction `placer` dont le type est `arbre -> int -> arbre -> arbre` telle que l'appel `(placer g v d)` sur l'entier v et les arbres en tas g et d tels que $\big(\eta(g) = \eta(d)\big) \wedge \mathcal{C}(g)$ ou $\big(\eta(g) = \eta(d) + 1\big) \wedge \mathcal{C}(d)$ (η est définie à la définition 1.2), renvoie un arbre en tas contenant les même étiquettes que g et d ainsi que l'étiquette v, et dont le squelette est le même que `Noeud(g, v, d)`

Cette fonction devra au plus parcourir une branche de g ou de d. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives. *Indication :* Il s'agit de la fonction `fusion_avec_colle` du TD.

solution :

```
(* On n'a pas de fonction insérer. Par contre, le fait que les arbres soient parfaits
    ↪  permet d'éliminer quelques cas.*)
let rec placer g v d=
    match g, d with
        |Vide, Vide   -> Noeud(Vide, v, Vide)
        |Vide, _      -> failwith "arbre pas parfait"
        |Noeud(Vide, x, Vide), Vide -> Noeud( Noeud(Vide, min v x, Vide), max v x,
    ↪ Vide)
        |_, Vide      -> failwith "arbre pas parfait"

        (* cas général *)
        |Noeud(gg, eg, gd), Noeud(dg, ed, dd) when v > max eg ed    -> Noeud(g,v,d)
        |Noeud(gg, eg, gd), Noeud(_, ed,_) when eg>ed    ->  Noeud(  placer gg v gd,
    ↪ eg, d)
        |_, Noeud(dg, ed,dd)      -> Noeud(g, ed, placer dg v dd)
;;
```

2. Écrire en CaML une fonction `extraitMax` dont le type est `int -> arbre -> arbre*int` telle que l'appel `(extraitMax i a)` un arbre en tas `a` et l'entier i tel que $i = |a|$, renvoie le couple (a', m) où a' est un arbre en tas de taille $|a| - 1$ qui contient les mêmes étiquettes que a sauf celle de la racine $\mathcal{E}(a)$ et m est l'étiquette de la racine de a.
Cette fonction devra au plus parcourir une fois la branche allant de la racine au dernier nœud de a. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives. *Indication :* On pourra écrire comme fonction auxiliaire une variante judicieuse de la fonction `lire` de la question 8 de la partie 5.

solution :

```
let rec enleveNoeudAux l a=
    (* l est l'écriture en base 2 du numéro de nœud qu'on veut enlever sans son
    ↪ premier 1. *)
    (* Renvoie le couple (a privé de ce noeud, étiquette de ce noeud) *)
    match l, a with
        | [], Noeud(Vide, e, Vide)  -> Vide, e
        | [], _      -> failwith "l'indice n'était pas celui d'une feuille"
        | 1::q, Noeud(fg,e,fd)   -> let fd', x = enleveNoeudAux q fd in Noeud(fg,e,fd
    ↪ '),x
        | 0::q, Noeud(fg, e, fd) -> let fg', x = enleveNoeudAux q fg in Noeud(fg',e,
    ↪ fd),x
        | _, Vide    -> failwith "indice trop grand"
        | x, _  -> failwith "écriture en base 2 douteuse"
;;

let enleveNoeud i a=
    enleveNoeudAux (List.tl (decompBase2 i)) a;;

let extraitMax i a=
    let a', x = enleveNoeud i a in
         match a' with
              | Vide    -> Vide, x
              | Noeud(fg, m, fd)    -> placer fg x fd, m
;;
```

3. Écrire en CaML une fonction `trier` dont le type est `int list -> int list` telle que l'appel `(trier l)` sur une liste d'entiers `l` renvoie une liste triée d'entiers contenant les mêmes valeurs que `l`. Cette fonction exploitera la technique du tri par tas. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.
solution :

```
let rec detasser i t accu=
```

```
2     (* i est le nombre d'éléments de t. Renvoie (les éléments de t dans l'ordre
      ↪ croissant) @ accu  *)
3     if i=0 then accu
4     else
5         let t', m = extraitMax i t in
6             detasser (i-1) t' (m::accu)
7 ;;
8
9 let trier l =
10    let n = List.length l in
11    let t = construire l in
12       detasser n t []
13 ;;
```