Devoir surveillé n°3

Durée : 2 heures, calculatrices et documents interdits

I - Arbres binaires stricts et complets – questions de cours

On rappelle que la *profondeur* du nœud d'un arbre est la distance entre ce nœud et la racine de l'arbre, et que la *hauteur* d'un arbre est la plus grande profondeur possible de ses nœuds. On convient que la hauteur de l'arbre vide est -1.
On rappelle également qu'un *arbre binaire strict* est un arbre dont tous les nœuds internes ont exactement deux fils, et qu'un *arbre binaire complet* est un arbre dont toutes les feuilles sont à la même hauteur.

On utilisera pour les arbres le type suivant :

```
type 'a arbre =
  | Nil
  | Noeud of 'a * 'a arbre * 'a arbre
;;
```

1) Écrire une fonction Caml de type `'a arbre -> int` et renvoyant sa hauteur.

2) Soit A un arbre binaire strict comportant n nœuds internes. Combien a-t-il de feuilles ? Le démontrer.

3) Soit A un arbre binaire de hauteur h. Donner un encadrement du nombre n de ses nœuds en fonction de h, et le démontrer.

4) Tracer deux arbres de hauteur 3 pour lesquels les bornes de l'encadrement précédent sont atteintes.

5) Combien de nœuds le niveau de profondeur p d'un arbre binaire complet a-t-il de noeuds ? Le démontrer.

II - Partition et union d'arbres binaires de recherche

Un arbre binaire étiqueté par des entiers peut être défini par le type Caml :

```
type int_arbre = Nil | Noeud of int * int_arbre * int_arbre ;;
```

Soit A un tel arbre, on notera `Etiq` (A) l'ensemble des étiquettes de l'arbre A défini récursivement par :

```
Etiq (Nil)={}
Etiq (Noeud(i, g, d))={i} ∪ Etiq (g) ∪ Etiq (d)
```

Un arbre binaire de recherche A (en abrégé ABR), étiqueté par des entiers distincts, est un arbre qui vérifie la propriété suivante :

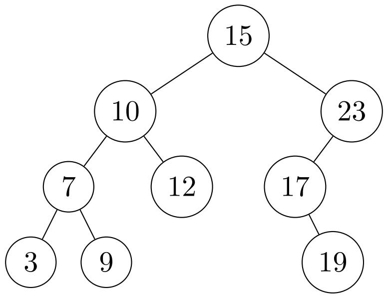
$\forall$ n nœud de A de la forme $n =$`Noeud` (i, g, d) alors :

— $\forall\, j \in$ `Etiq` (g), $j < i$;
— $\forall\, j \in$ `Etiq` (d), $j > i$.

Exemple :

```
A = Noeud(15,
    Noeud(10,
        Noeud(7, Noeud(3,Nil,Nil), Noeud(9,Nil,Nil)),
        Noeud(12,Nil,Nil)
        )
    Noeud(23,
        Noeud(17,Nil,Noeud(19,Nil,Nil)),
        Nil
        )
    )
```

est un ABR qui peut être représenté graphiquement par :



Tous les arbres qui seront considérés dans ce sujet sont des arbres binaires de recherche.

On appelle *coupure* d'un ABR A selon une valeur c une partition de A en un couple de sous-arbres (G_A, D_A) tels que :

- G_A et D_A sont deux ABR ;
- `Etiq` (A) =`Etiq` $(G_A) \cup$ `Etiq` (D_A) ;
- $\forall\, j \in$ `Etiq` (G_A), $j \leqslant c$;
- $\forall\, j \in$ `Etiq` (D_A), $j > c$.

A : G_A : D_A :

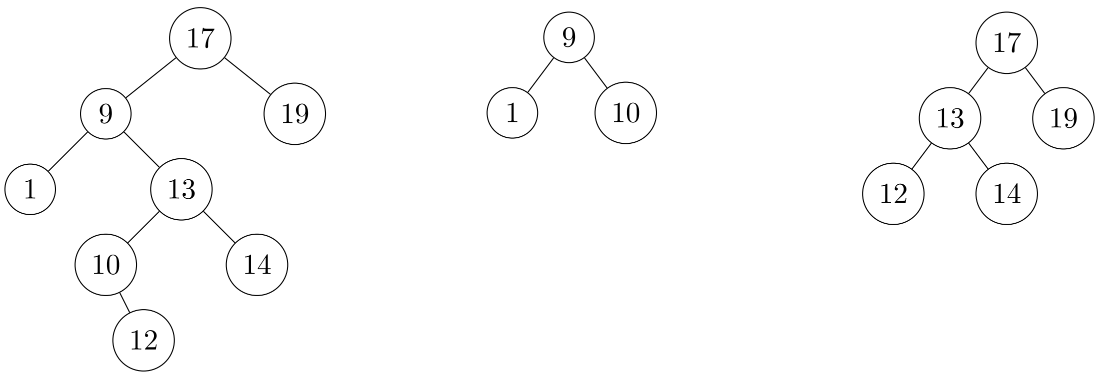


FIGURE 1 – Une coupure de A par rapport à 10

1) On remarque que `(Nil, Nil)` est une coupure de `Nil` selon c (pour tout c).
Soit `Noeud`(c, g, d) un ABR, montrer que (`Noeud` $(c, g,$ `Nil`$), d)$ est une coupure de `Noeud`(c, g, d) selon c.

2) Soient i un entier, `Noeud`(i, A, B) un ABR et c un entier. Montrer que si $c < i$ et si (G_A, D_A) est une coupure de A selon c, alors $(G_A,$`Noeud`$(i, D_A, B))$ est une coupure de `Noeud`(i, A, B) selon c.

3) Par symétrie, construire une coupure de l'ABR `Noeud`(i, A, B) selon c, dans le cas où $c > i$.

4) Écrire une fonction récursive Caml `couper : int-> int_arbre-> (int_arbre * int_arbre)` telle que (`couper` c a) renvoie une coupure de a selon c.

5) Si $|a|$ désigne la hauteur de l'arbre a, montrer que la complexité de la fonction (`couper` c a), en nombre d'appels récursifs, est inférieure ou égale à $|a|$.

6) En utilisant la fonction `couper`, écrire une fonction non récursive Caml `inserer_racine : int -> int_arbre -> int_arbre` telle que si b =(`inserer_racine` i a) alors `Etiq`(b) =`Etiq`$(a) \cup \{i\}$ et b est un ABR dont la racine est étiquetée par i.
On supposera que les arguments i et a de la fonction sont tels que $i \notin$`Etiq`(a).

7) Écrire une fonction récursive Caml `min : int_arbre -> int` qui renvoie l'étiquette minimale d'un ABR différent de `Nil`.

8) En utilisant les fonctions `min` et `couper`, écrire une fonction non récursive Caml `union : int_arbre -> int_arbre -> int_arbre` telle que si a et b sont deux ABR tels que les étiquettes de a sont toutes inférieures aux étiquettes de b, c'est-à-dire [$\forall\ x \in$ `Etiq`(a), $(\min b) > x$], alors si $c =$ (`union` a b) alors c est un ABR et `Etiq`(c) =`Etiq`$(a)\cup$ `Etiq`(b).

9) Donner un majorant de la complexité de (`union` a b) dans le pire cas en fonction de $|a|$ et $|b|$, en comptant uniquement les appels récursifs de fonction.

III - Backtracking et résolution de sudoku

Piles et résolution d'un sudoku

Un sudoku est une matrice de format 9×9. Chaque case peut être remplie par :
- Un entier de l'intervalle $[\![1, 9]\!]$;
- Un zéro, auquel cas nous dirons que la case est vide.

On créée en `OCaml` le type suivant afin d'implémenter des grilles de sudoku :

```
type sudoku = int array array;
```

Un sudoku s est *valide* s'il vérifie les trois conditions suivantes :
- Pour tout $n \in [\![1, 9]\!], n$ ne peut apparaître deux fois dans une même ligne ;
- Pour tout $n \in [\![1, 9]\!], n$ ne peut apparaître deux fois dans une même colonne ;
- On divise la matrice s en 9 blocs de format 3×3 comme sur la figure :

7	6	3	8	9	2	5	4	1
9	2	1	7	5	4	8	6	3
5	4	8	1	3	6	9	7	2
6	5	4	2	1	7	3	8	9
2	8	9	4	6	3	1	5	7
3	1	7	9	8	5	4	2	6
4	9	2	3	7	8	6	1	5
8	3	5	6	2	1	7	9	4
1	7	6	5	4	9	2	3	8

On demande alors que pour tout $n \in [\![1, 9]\!]$, n n'apparaisse pas deux fois dans un même bloc.
Si de plus aucune case de s n'est vide, on dira que s est *résolu* (l'exemple ci-dessus montrait un sudoku résolu).
Un sudoku non résolu comporte des cases vides, que nous remplirons en fait par des 0 dans le tableau correspondant. Un sudoku valide mais non résolu est par exemple le suivant :

			7	6	2		9	
			3	8			2	7
2	8			5	9	1	6	3
						6		1
	1	5			3	2		
6				4	5	7		8
	2		9			4		5
	7		8	2	4			
9			5		7		8	

et il est représenté par le tableau suivant :

```
let s = [|
[|0;0;0;7;6;2;0;9;0|];
[|0;0;0;3;8;0;0;2;7|];
[|2;8;0;0;5;9;1;6;3|];
[|0;0;0;0;0;0;6;0;1|];
[|0;1;5;0;0;3;2;0;0|];
[|6;0;0;0;4;5;7;0;8|];
[|0;2;0;9;0;0;4;0;5|];
[|0;7;0;8;2;4;0;0;0|];
[|9;0;0;5;0;7;0;8;0|]
|];;
```

Les lignes et les colonnes seront numérotées de 0 à 8 ! !
Notre but est d'écrire un programme prenant un sudoku valide et le remplissant si possible afin d'obtenir un sudoku résolu.
Nous allons suivre une méthode naïve : on choisit une case, on y inscrit un chiffre qui ne contredit aucune des trois règles ci-dessus puis on passe à une case suivante. Et si on arrive à une impasse (aucune possibilité pour une certaine case) il faut effacer les chiffres inscrits et recommencer avec d'autres.
Ce type de méthode consistant à essayer toutes les possibilités, en revenant en arrière dès qu'on détecte une impossibilité s'appelle en anglais une méthode par "backtracking (bbacktracking " signifie "retour en arrière »). Une telle méthode peut être implémentée par une pile, ou par une fonction récursive. Nous allons implémenter les deux points de vue.

A) Premières fonctions

La première étape est, étant donnés un sudoku s et une case (i, j) de celui-ci, de calculer tous les chiffres pouvant être inscrits dans cette case tout en satisfaisant les trois règles du jeu. Pour ce, on va créer un tableau `possible` de 10 cases, initialement toutes vraies, puis pour tout $n \in [\![1, 9]\!]$, on mettra faux dans `possible.(n)` si on ne peut par mettre n dans la case `s.(i).(j)`. Attention, il y a bien 10 cases : en effet les cases remplies comportent des numéros de 1 à 9, et les cases non remplies comportent un 0. La valeur de `possible.(0)` n'aura pas d'importance, car on cherchera pas à remplir une case avec un 0.
On pourra supposer que la case (i, j) de s est vide.

1) Écrire une fonction créant un tableau nommé `possible`, contenant 10 cases valant toutes `true`.

2) Écrire une fonction `regleLigne` de type `sudoku -> int * int -> bool array -> unit` prenant en entrée le sudoku s, le couple de coordonnées (i, j) et le tableau `possible`, et passant `possible.(n)` à false pour tout entier n qu'on ne peut inscrire en case (i, j) de s sans contrevenir à la règle sur les lignes, c'est-à-dire pour tout entier $n \in [\![1, 9]\!]$ apparaissant déjà dans une case de la ligne i.

3) Écrire une fonction `regleColonne` de type `sudoku -> int * int -> bool array -> unit` prenant en entrée le sudoku s, le couple de coordonnées (i, j) et le tableau `possible`, et passant `possible.(n)` à false pour tout n qu'on ne peut inscrire en case (i, j) de s sans contrevenir à la règle sur les lignes.

4) Les neuf blocs de format 3×3 utilisés par la troisième règle seront repérés par un couple $(ib, jb) \in [\![0, 3[\![^2$ de la manière naturelle. Par exemple la case $(2, 1)$ de s est dans le bloc $(0, 0)$. La case $(3, 7)$ est dans le bloc $(1, 2)$. Quelles opérations simples permettent de trouver les indices (ib, jb) du bloc contenant la case (i, j) de s ?

5) Écrire une fonction `regleBlocs` analogue aux précédentes pour prendre en compte la troisième règle.

6) Écrire enfin une fonction `valeursPossibles` de type `sudoku -> int * int -> bool array` prenant en entrée un sudoku s et le couple de coordonnées (i, j) d'une de ses cases, et renvoyant le tableau `possible` indiquant les valeurs pouvant être inscrites dans la case `s.(i).(j)` sans contrevenir aux trois règles.

7) Combien de lectures de cases du sudoku sont-elles effectuées lors d'un appel à `valeursPossibles` ?

B) Implémentation impérative

Nous utiliserons trois piles mutables `aRemplir`, `remplies`, et `aEssayer`.

- `aRemplir` contiendra les cases à remplir, et `remplies` les cases déjà remplies. Ces piles contiendront donc des couples d'entiers.
- `aEssayer` contiendra les prochains essais à faire. Ce sera une liste de triplets : un triplet (i, j, n) signifiera que l'essai à effectuer est d'inscrire n dans la case (i, j).

Nous utiliserons les piles du module `Stack` de `OCaml`. En plus des fonctions `Stack.create`, `Stack.pop`, `Stack.push`, et `Stack.is_empty` déjà utilisées en cours, on pourra utiliser `Stack.top` qui renvoie l'élément au sommet d'une pile mais sans le dépiler (c'est donc une fonction pure, sans effet de bord).

8) Écrire une fonction `init_aRemplir` de type `sudoku -> (int * int) Stack.t` qui prend un sudoku s et qui crée et renvoie la pile `aRemplir` en y mettant tous les couples d'entiers (i, j) correspondant aux cases vides de s.

9) Écrire une fonction `empilePossibles` de type `sudoku -> (int * int)-> (int * int * int) Stack.t -> unit` qui prendra en entrée le sudoku s, un couple (i, j) de coordonnées d'une case, et la pile `aEssayer`, et qui empilera dans `aEssayer` tous les essais possibles en la case de coordonnées (i, j).

10) Le point clé est la phase de retour en arrière, lorsqu'on efface les essais infructueux. On veut alors écrire une fonction `retourArriere` de type `sudoku -> (int * int) Stack.t -> (int * int) Stack.t -> int * int -> unit`. Elle prendra en arguments le sudoku s, les piles `remplies` et `aRemplir`, ainsi que le couple (i, j) des coordonnées de la case intervenant dans le prochain essai contenu dans `aEssayer`. Tant que la case (i, j) en entrée est différente de la case du haut de la pile `aRemplir`, cette fonction effectuera les instructions suivantes :

- dépiler la case (k, l) de `remplies` ;
- passer `s.(k).(l)` à zéro ;
- empiler (k, l) dans `aRemplir`

Ainsi, après avoir éxécuté cette fonction, la prochaine case à remplir correspondra au prochain essai contenu dans la pile `aEssayer`, et on pourra continuer.
On remarquera que cette fonction ne fera rien si la case (i, j) en entrée est égale à la case du haut de la pile `aRemplir`.

11) Passons enfin à la fonction finale ! Elle effectuera les instructions suivantes :

- créer les différentes piles utilisées ;
- tant que la pile `aRemplir` n'est pas vide :
  - effectuer `empilePossibles` avec pour arguments la case du haut de la pile `aRemplir` et la pile `aEssayer` ;

— dépiler `aEssayer`, qui renvoie un triplet (i, j, n) ;
— effectuer `retourArriere` (si `empilePossibles` n'a rien empilé c'est que le dernier essai est une impasse et il faut revenir en arrière ; sinon `retourArriere` ne fera rien mais on l'éxécute tout de même systématiquement à cette étape) ;
— remplir la case (i, j) avec n ;
— dépiler `aRemplir` ;
— empiler (i, j) dans `remplies`.

On supposera le sudoku résoluble : on ne se préoccupera donc pas de lever une exception dans le cas où la résolution n'aboutit pas.