Chapitre 1

Arbres binaires de recherche

Introduction

Considérons les trois opérations de base que sont la recherche d'un élément, l'insertion d'un élément et la suppression d'un élément.

Dans le cas d'une structure de tableau :

- la recherche d'un élément est de complexité linéaire, voire de complexité logarithmique dans le cas d'un tableau trié,
- l'insertion ou la suppression d'un élément est de complexité linéaire.

Dans le cas d'une structure de liste :

- la recherche d'un élément est de complexité linéaire,
- l'insertion d'un élément (ajout-en-tête) est de complexité constante,
- la suppression d'un élément est de complexité linéaire.

Nous sommes globalement en complexité linéaire.

Dans un arbre « bien construit » et « bien rangé » :

- « bien construit » : en le remplissant au maximum à chaque niveau, nous allons minimiser la hauteur de l'arbre par rapport au nombre de sommets,
- « bien rangé » : nous allons ranger les nombres par rapport à la racine,

nous espérons améliorer cette complexité linéaire.

1 Rappels et compléments sur les arbres binaires

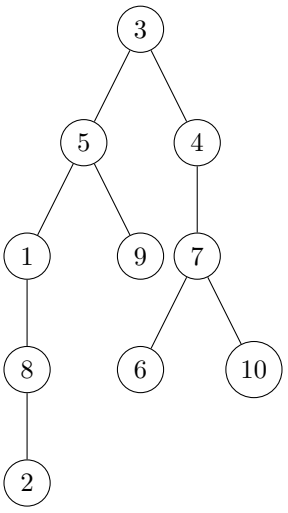
1.1 Vocabulaire

Définition.

Soit A un arbre, c'est-à-dire un graphe connexe, acyclique et enraciné.

- Les **sommets** de A sont les sommets du graphe associé.
- La **taille** de A, notée $|A|$, est le nombre de sommets de A.
- L'unique **sommet** n'ayant pas de père est appelé la **racine** de l'arbre A.
- Un sommet de A n'ayant pas de fils est appelé une **feuille**.
- Un sommet de A ayant au moins un fils est appelé un **nœud**.
- L'**arité** d'un sommet est le nombre de ses fils.
  Les feuilles sont d'arité nulle et les nœuds sont d'arité non nulle.
- La **profondeur** d'un sommet est le nombre d'ascendants stricts de ce sommet.
- La **hauteur** de l'arbre A, notée $h(A)$, est la profondeur maximale de ses sommets.
  Par convention, la hauteur de l'arbre vide vaut -1.
- L'ensemble constitué d'un sommet x et de ses descendants est appelé **sous-arbre de A issus de x**.
- Un arbre est **étiqueté** lorsqu'à chaque sommet est associée une information appelée **étiquette** du sommet.
  Des sommets distincts peuvent porter la même étiquette.

Exercice. Considérons l'arbre A suivant :



1. Donner la taille de A en listant ses sommets, ses nœuds et ses feuilles. Préciser sa racine.

2. Donner l'arité et la profondeur de chaque sommet de A. En déduire la hauteur de A.

3. Donner un exemple de sous-arbre de A.

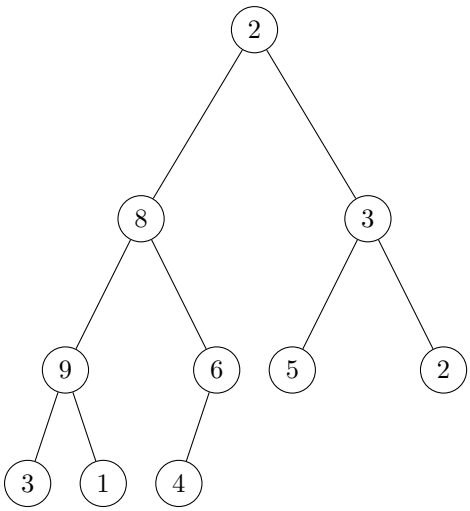
Définition.

- Un arbre est dit **binaire** si tous ses sommets sont d'arité au plus 2.
- Un arbre est dit **binaire entier** (ou **strict**) si tous ses nœuds sont d'arité exactement 2.
- Un arbre binaire A est dit **parfait** si
  - toutes les feuilles de A sont de profondeur $h(A)$ ou $h(A)-1$;
  - pour tout $p < h(A)$, il y a exactement 2^p sommets de profondeur p ;
  - toutes les feuilles de profondeur $h(A)$ sont « le plus à gauche » de A.
- Un arbre binaire A est dit **complet** si c'est un arbre parfait et que toutes ses feuilles sont de la même profondeur égale à $h(A)$.

Remarques. Autrement dit :

- Un arbre binaire est parfait si c'est un arbre binaire dont tous les « niveaux » (sommets de même profondeur) sont totalement remplis sauf éventuellement le dernier « niveau » qui doit être rempli de gauche à droite.
- Un arbre binaire est complet si c'est un arbre binaire dont tous les « niveaux » sont totalement remplis.

Exemple. L'arbre binaire entier suivant est parfait mais pas complet :

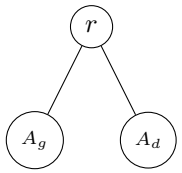


1.2 Définition inductive des arbres binaires

Définition.

On peut définir de manière récursive les **arbres binaires** sur un ensemble Σ :

- Cas de base : L'arbre vide est un arbre binaire.
- Étape d'induction : Si r est un élément de Σ et si A_g et A_d sont deux arbres binaires sur Σ, alors

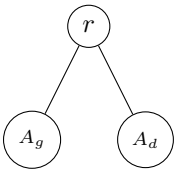
$A =$ 

est un arbre binaire sur Σ.

On notera $\mathcal{A}_\Sigma$ l'ensemble des arbres binaires sur un ensemble Σ.

Remarques.

1. Dans ce dernier cas, l'élément r est la racine de A, A_g est le « fils gauche de A », A_d le « fils droit de A ». Une feuille est alors un sommet dont les deux fils sont des arbres vides.
2. A partir de cette définition, une fonction ϕ peut alors être définie sur $\mathcal{A}_\Sigma$ de manière inductive par :
   - la donnée de la valeur de ϕ évalué en l'arbre vide ;
   - en supposant connaître $r \in \Sigma$, $\phi(A_g)$ et $\phi(A_d)$ pour A_g et $A_d \in \mathcal{A}_\Sigma$, la définition de ϕ évalué en l'arbre



Exercice.

1. Donner une définition inductive de la hauteur d'un arbre binaire homogène.

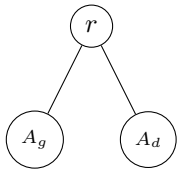
2. Donner une définition inductive de la taille d'un arbre binaire homogène.

Remarque. Du fait des définitions inductives des arbres, les preuves sur les arbres se feront la plupart du temps par induction structurelle et les fonctions OCaml que nous définirons sur les arbres seront récursives.

Définition.

Soit $\mathcal{P}(A)$ un prédicat qu'on cherche à démontrer exprimant une propriété sur tout arbre A de $\mathcal{A}_\Sigma$.
La **démonstration par induction structurelle** se présente comme suit :

- Cas de base : Montrer que $\mathcal{P}(A)$ est vraie si A est l'arbre vide.
- Étape d'induction : Soit $r \in \Sigma$, $A_g \in \mathcal{A}_\Sigma$ et $A_d \in \mathcal{A}_\Sigma$ deux arbres binaires pour lesquels $\mathcal{P}(A_g)$ et $\mathcal{P}(A_d)$ sont vraies. Soit A l'arbre suivant :

  $A =$ 

  Montrer que $\mathcal{P}(A)$ est vraie.
- Conclusion : Quelque soit $A \in \mathcal{A}_\Sigma$, $\mathcal{P}(A)$ est vraie.

Exercice. Prouver par induction structurelle que tout arbre binaire A à n sommets et de hauteur h vérifie

$$h+1 \leq n \leq 2^{h+1}-1 \quad \text{et donc} \quad \log_2(n+1)-1 \leq h \leq n-1.$$

Remarque. La complexité de nombreux algorithmes sur les arbres se calcule en fonction de la hauteur de ces arbres, d'où l'idée d'avoir la hauteur la plus petite possible.
Dans le cas des arbres binaires :

- configuration la pire : $n = h+1$, nous sommes en fait ramenés à une liste,
- configuration la meilleure : $n = 2^{h+1}-1$, configuration d'arbre binaire complet, configuration idéale mais réalisable que pour certaines valeurs de n.

1.3 Arbres binaires équilibrés

Définition.

Un arbre binaire A est **équilibré** si sa hauteur est "minimale", c'est-à-dire si

$$h(A) = O(\log_2(|A|)).$$

Propriété 1 (Parfait implique équilibré)

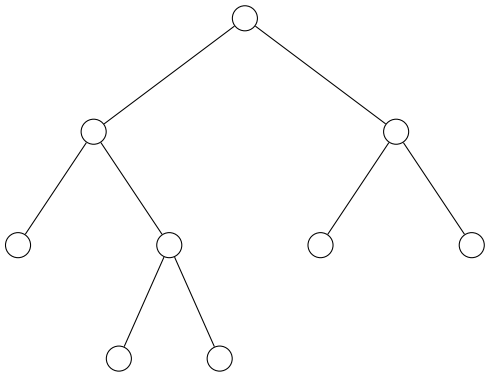
Un arbre binaire parfait A non vide est équilibré et sa hauteur vaut $\lfloor \log_2(|A|) \rfloor$.

Preuve.

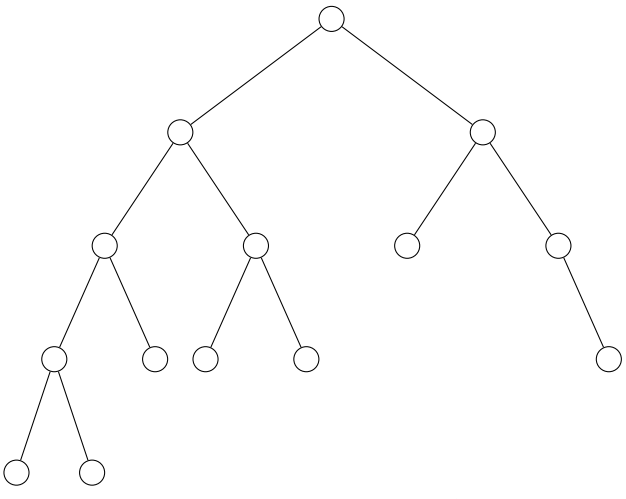
□

Remarque. Il existe d'autres structures d'arbres qui permettent d'obtenir à coup sûr un arbre équilibré : on prouve par induction structurelle que les arbres bien équilibrés (arbres binaires entiers dont toutes les feuilles sont de profondeur $h(A)$ ou $h(A) - 1$) sont équilibrés.

Il est également prouvé que les arbres binaires « de type AVL » tels que pour tout noeud interne de l'arbre, la hauteur du sous-arbre gauche de ce noeud et la hauteur du sous-arbre droit de ce noeud différent au plus de 1 sont aussi équilibrés. Mais aussi les arbres rouge-noir... etc...



Exemple de squelette d'arbre bien équilibré



Exemple de squelette d'arbre AVL

2 Arbres binaires de recherche

2.1 Définition itérative des ABR

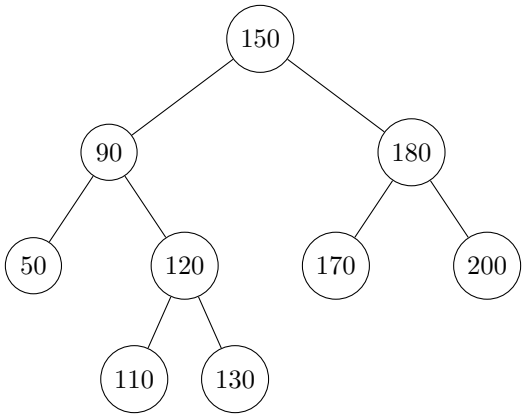
Définition.

Soit $(\Sigma, \leq)$ un ensemble totalement ordonné et A un arbre binaire à étiquettes dans Σ.

On dit que l'arbre A est un **arbre binaire de recherche** (on utilisera l'abréviation ABR) si et seulement si pour tout sommet de l'arbre A d'étiquette e :

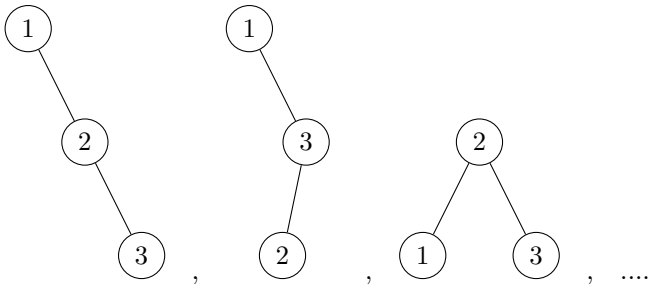
- toutes les étiquettes des sommets issus du sous-arbre gauche de e sont inférieures ou égales à e,
- toutes les étiquettes des sommets issus du sous-arbre droit de e sont strictement supérieures à e.

Exemple. Voici un arbre binaire de recherche :



Remarques.

1. L'arbre vide peut être considéré comme un ABR.
2. Nous n'avons pas unicité de l'ABR représentant une série de données. Par exemple :



3. Dans la définition d'un ABR,
   - nous n'allons pas regarder toutes les étiquettes du sous-arbre gauche mais la valeur maximale des étiquettes des sommets issus du sous-arbre gauche,
   - nous n'allons pas regarder toutes les étiquettes du sous-arbre droit mais la valeur minimale des étiquettes des sommets issus du sous-arbre droit.
4. On avait vu en première année une implémentation des dictionnaires à l'aide de table de hachage.

   Les dictionnaires peuvent aussi être implémentés sous forme d'arbres binaires de recherche : chaque étiquette est alors constituée d'une clé et d'une valeur et l'arbre forme un arbre binaire de recherche selon les clés.

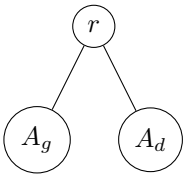
2.2 Définition récursive des ABR

Commençons par donner une définition par induction structurelle du min et du max d'un arbre :

Définition.

Soit $(\Sigma, \leq)$ un ensemble totalement ordonné.

- Cas de base : Pour l'arbre vide A, on pose $\max(A) = -\infty$ et $\min(A) = +\infty$.
- Étape d'induction : Dans le cas d'un arbre binaire A à étiquettes dans Σ de la forme



on pose :

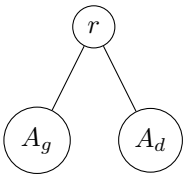
$$\begin{cases} \max(A) = \max\left(\max(A_g), r, \max(A_d)\right), \\ \min(A) = \min\left(\min(A_g), r, \min(A_d)\right). \end{cases}$$

On peut alors donner une définition récursive d'un ABR (pour pouvoir faire ensuite des preuves par induction structurelle...) :

Définition.

Soit $(\Sigma, \leq)$ un ensemble totalement ordonné.

- Cas de base : L'arbre vide est un ABR.
- Étape d'induction : Étant donnés deux ABR A_g et A_d à étiquettes dans Σ et r appartenant à Σ tel que $\max(A_g) \leq r < \min(A_d)$, l'arbre A définit par



est un ABR.

2.3 Parcours en profondeur en mode infixe d'un ABR

Propriété 2 (Parcours en profondeur en mode infixe d'un ABR)

Soit $(\Sigma, \leq)$ un ensemble totalement ordonné et A un ABR à étiquettes dans Σ.
Lors du parcours en profondeur en mode infixe de l'arbre A, les étiquettes sont parcourues par ordre croissant.

Preuve.

□

3 Implémentation de la structure d'ABR

Dans cette partie, nous allons considérer des arbres binaires dont les étiquettes sont de type `int` :

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

Définir en OCaml un arbre binaire de recherche.

3.1 Recherche d'une étiquette dans un ABR

Une des opérations les plus courantes dans un ABR A est la recherche d'une étiquette de valeur k dans l'arbre. La démarche est évidente :

- soit k est égal à l'étiquette de la racine de A ;
- soit k est strictement inférieur à l'étiquette de la racine de A, la recherche se poursuit alors dans le sous-arbre gauche de A ;
- soit k est strictement supérieur à l'étiquette de la racine de A, la recherche se poursuit alors dans le sous-arbre droit de A.

Écrire en OCaml une fonction permettant la recherche d'une étiquette dans un ABR.

3.2 Recherche de l'étiquette minimale/maximale

Au vu de la définition d'un ABR, la démarche est simple pour la recherche de l'étiquette minimale/maximale :

- la recherche de la valeur minimale des étiquettes se poursuit dans le sous-arbre gauche tant que ce dernier n'est pas vide ;
- la recherche de la valeur maximale des étiquettes se poursuit dans le sous-arbre droit tant que ce dernier n'est pas vide.

Écrire en OCaml des fonctions permettant d'obtenir la plus grande et la plus petite étiquette d'un ABR.

3.3 Recherche du prédécesseur/successeur d'une étiquette

Soit A un ABR et k un entier. Nous définissons :

- le **prédécesseur** de k dans A comme la plus grande des étiquettes e de A telle que $e < k$;
- le **successeur** de k dans A comme la plus petite des étiquettes e de A telle que $e > k$.

Écrire en OCaml deux fonctions permettant d'obtenir le prédécesseur et le successeur de k dans un ABR.

3.4 Insertion d'une étiquette

Nous avons deux méthodes : insertion au niveau des feuilles ou insertion au niveau de la racine. Ces deux méthodes doivent conserver la structure d'arbre binaire de recherche.

Insertion au niveau des feuilles

Écrire en OCaml une fonction permettant d'insérer une étiquette au niveau des feuilles d'un ABR.

Insertion au niveau de la racine

Afin d'insérer la nouvelle étiquette à la racine, nous commencerons par partitionner l'arbre A de manière à placer tous les éléments inférieurs à la nouvelle étiquette dans le fils gauche et les autres dans le fils droit.

Écrire en OCaml une fonction permettant de partitionner un ABR en fonction d'une étiquette.

En déduire une fonction en OCaml permettant d'insérer une étiquette au niveau de la racine d'un ABR.

3.5 Suppression d'une étiquette

Afin de supprimer le sommet étiqueté par une valeur k donnée dans un ABR :

- nous commençons par rechercher le sommet étiqueté par la valeur k (notons alors A' le sous-arbre de racine le sommet ainsi trouvé) ;
- deux cas se présentent :
  - soit l'arbre A' n'a pas de fils droit, nous remplaçons alors A' par son fils gauche ;
  - soit l'arbre A' a un fils droit A'_d non vide, nous recherchons alors le sommet s d'étiquette de valeur minimale de l'arbre A'_d que nous supprimons de l'arbre A'_d et nous le plaçons à la racine du nouvel arbre A'.

Écrire en OCaml une fonction qui prend en argument un ABR A et retourne le couple (m, A_1) formé d'un élément m de valeur d'étiquette minimale dans A et de l'arbre $A_1 = A \setminus \{m\}$.

En déduire une fonction en OCaml permettant de supprimer le sommet étiqueté par une valeur k dans un ABR.

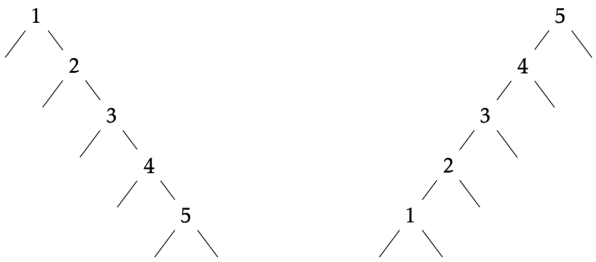
4 Le problème du déséquilibre

On peut remarquer/vérifier que toutes les fonctions de requêtes, d'insertion et de suppression dans un arbre binaire de recherche ont un coût temporel en $O(h(A))$, autrement dit, en posant $n = |A|$:

- un coût linéaire $O(n)$ dans le cas d'un arbre binaire de recherche quelconque,
- un coût logarithmique $O(\log_2(n))$ dans le cas d'un arbre binaire de recherche maintenu équilibré.

Il est donc très important de maintenir l'arbre équilibré pour obtenir une complexité logarithmique.

Malheureusement, les fonctions d'insertion et de suppression que nous avons écrites peuvent conduire à des arbres très déséquilibrés, par exemple des arbres peignes dans le cas où les étiquettes sont insérées par ordre croissant ou décroissant.



Insertion des étiquettes 1, 2, 3, 4, 5 au niveau des feuilles ou au niveau de la racine.

Cependant, on peut démontrer que le comportement du cas moyen est plus proche du cas optimal que du cas le plus défavorable. Plus précisément, si un arbre binaire de recherche est créé en insérant n étiquettes distinctes au niveau des feuilles dans un ordre aléatoire, il existe une constante c telle que la hauteur moyenne de ces $n!$ arbres soit équivalente à $c\log_2(n)$.

Il est néanmoins possible de garantir une complexité dans le pire des cas en $O(\log_2(n))$ à condition de maintenir en place une structure d'arbre qui garantisse l'équilibrage : arbres AVL, arbres rouges et noirs, etc...

Dans le cas d'un dictionnaire, qui peut être implémenté sous la forme d'une table de hachage ou d'un arbre binaire de recherche, on peut alors comparer les complexités :

- complexité avec une implémentation à l'aide d'une table de hachage :

pire des cas		**en moyenne**	
lecture	ajout	lecture	ajout
$O(n)$	$O(n)$	$O(1)$	$O(1)$

- complexité avec une implémentation à l'aide d'un ABR (à condition de le maintenir équilibré) :

pire des cas		**en moyenne**	
lecture	ajout	lecture	ajout
$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$

5 Exercices

Sur les arbres binaires

Pour représenter un arbre binaire homogène étiqueté par le type `'a`, on définit le type :

```
type 'a arbre = Vide | N of 'a * 'a arbre * 'a arbre ;;
```

Exercice 1 (★)

Écrire en OCaml les fonctions suivantes :

1. `taille`, de type `'a arbre -> int`, qui retourne le nombre de sommet d'un arbre.
2. `feuilles`, de type `'a arbre -> int`, qui retourne le nombre de feuilles d'un arbre, c'est-à-dire le nombre de nœuds dont les deux fils sont égaux à `Vide`.
3. `appartient`, de type `'a -> 'a arbre -> bool`, qui détermine si l'un des sommets de l'arbre possède une étiquette donnée.
4. `hauteur`, de type `'a arbre -> int`, qui calcule la hauteur d'un arbre (on convient que la hauteur de `Vide` est -1).

Exercice 2 (★)

Écrire en OCaml une fonction

```
map_arbre : ('a -> 'b) -> 'a arbre  ->  'b arbre
```

équivalente à la fonction `List.map`, mais s'appliquant à des arbres binaires homogènes.

Exercice 3 (★★)

1. Écrire en OCaml une fonction `miroir : 'a arbre -> 'a arbre` qui prend en argument un arbre et détermine l'arbre miroir associé.
2. Écrire en OCaml une fonction `est_miroir : 'a arbre -> 'a arbre -> bool` qui prend en argument deux arbres et détermine si l'un est l'image miroir de l'autre.
3. En déduire une fonction `symetrique : 'a arbre -> bool` en OCaml qui détermine si un arbre binaire est symétrique (c'est-à-dire s'il est égal à `Vide` ou si son fils gauche est l'image miroir de son fils droit).

Exercice 4 (★★)

1. (a) Rappeler la définition du parcours en profondeur en mode préfixe d'un arbre binaire entier.
   (b) Écrire en OCaml une fonction `parcours_prefixe : 'a arbre -> 'a list` qui retourne la liste des étiquettes portées par les sommets d'un arbre binaire entier parcouru en profondeur en mode préfixe.
2. Mêmes questions pour le parcours en profondeur en mode postfixe.
3. Mêmes questions pour le parcours en profondeur en mode infixe.

Exercice 5 (★★)

Un arbre binaire A est complet lorsque ses fils gauche et droit sont complets et de même hauteur.

1. Pour déterminer si un arbre binaire est complet, on propose la fonction suivante :

```
let rec est_complet a = match a with
  | Vide -> true
  | N(_, fg, fd) -> est_complet fg && est_complet fd && hauteur fg = hauteur fd
;;
```

   Évaluer en fonction de $n = |A|$ le coût temporel de cette fonction.

2. Afin d'avoir une meilleure complexité, on propose d'utiliser une fonction auxiliaire qui détermine en même temps si un arbre est complet et la hauteur de cet arbre.

   Écrire en OCaml une fonction qui détermine si un arbre binaire est complet avec cette méthode et déterminer sa complexité.

Sur les arbres binaires de recherche

Pour représenter les arbres binaires de recherche, nous allons considérer des arbres binaires dont les étiquettes sont de type `int` :

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

Exercice 6 (★★)

On souhaite construire une fonction permettant de tester si un arbre binaire est un ABR.

1. Écrire en OCaml des fonctions permettant d'obtenir la plus grande et la plus petite étiquette d'un arbre binaire.
2. En déduire une fonction en OCaml permettant de tester si un arbre binaire est un ABR.

Exercice 7 (★★)

On souhaite trier un tableau de n nombres en commençant par les insérer un par un dans un ABR puis en les ordonnant suivant un parcours infixe.

1. Écrire en OCaml une fonction d'insertion au niveau des feuilles dans un ABR.
2. Écrire en OCaml une fonction qui crée un ABR à partir des éléments d'un tableau.
3. En déduire une fonction en OCaml qui répond au problème posé.
