7 Arbres

Nous avons jusqu'à présent étudié des structures de données séquentielles (piles, files, tables de hachage, etc.). Dans ce chapitre, on étudie des structures de données hiérarchiques.
On peut définir la structure d'*arbre* de différentes façons, avec quelques subtilités, ce qui offre une certaine souplesse et permet de s'adapter au problème étudié. **Il convient d'être particulièrement vigilant à la lecture des sujets proposés pour identifier correctement la structure attendue**.

7.1 Arbres binaires

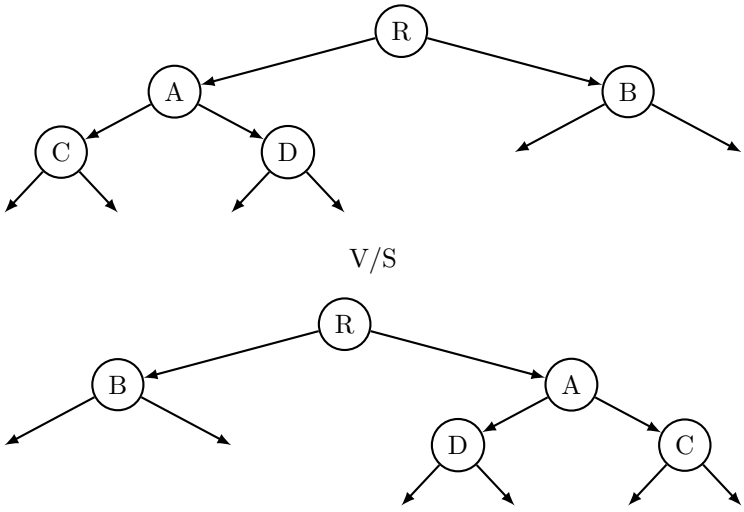
Définition 7.1 - *arbre binaire*

Un *arbre binaire* est un ensemble (éventuellement vide) de *nœuds* et est défini de manière inductive. Un arbre binaire est :

- Ou bien *l'arbre vide*
- Ou bien constitué d'un nœud R, appelé *racine*, et de deux *sous-arbres* binaires. Si existence, on appelle *fils gauche* (resp. *fils droit*) du nœud R la racine du sous-arbre gauche (resp. droit).
  Voir Figure 1

Remarque 7.2 - *à propos de la précédente définition*

Dans cette définition, si on permute les sous-arbres gauche et droit, on obtient un arbre différent. Ainsi, les deux arbres représentés ci-dessous sont deux arbres distincts :



Définition 7.3 - *feuille, père, arête, taille*

- Une *feuille* est un nœud dont les sous-arbres gauche et droit sont l'arbre vide.
- Un nœud est le *père* de ses éventuels fils. La liaison d'un père vers un fils est appelée une *une arête*.
- Tous les nœuds d'un arbre, hormis la racine et les feuilles, sont appelés des *nœuds internes*.
- La *taille* d'un arbre est son nombre de nœuds. On note $|\mathcal{A}|$ la taille de l'arbre $\mathcal{A}$

Remarque 7.4 - *définition inductive de la taille d'un arbre binaire*

- La taille de l'arbre vide est nulle
- Un arbre binaire comportant un sous-arbre gauche de taille n_g et un sous-arbre droit de taille n_d est de taille $1 + n_g + n_d$

Définition 7.5 - *hauteur, profondeur*

La *hauteur* d'un arbre binaire est définie inductivement :

- L'arbre vide est de hauteur : -1
- Un arbre binaire comportant un sous-arbre gauche de hauteur h_g et un sous-arbre droit de hauteur h_d est de hauteur $1 + \max(h_g, h_d)$.

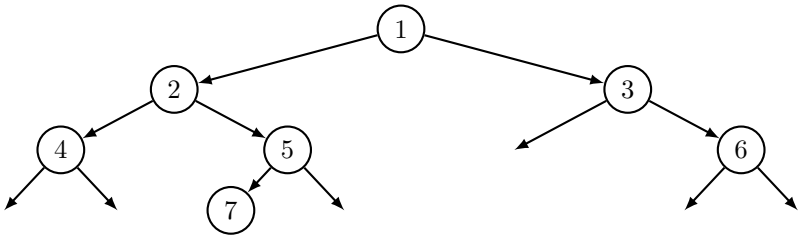
La *profondeur* d'un nœud est la distance de celui-ci à la racine (si existence). La hauteur d'un arbre est alors la profondeur maximale de ses neuds, et donc la profondeur maximale de ses feuilles.

Remarque 7.6 - *notion d'étiquettes*

Les nœuds et les arêtes d'un arbre peuvent porter des *étiquettes* : on leur associe alors une valeur.

Exemple 7.7 - *concernant les précédentes définitions*

Considérons l'arbre binaire suivant:



- Il s'agit d'un arbre binaire de taille 7, (il comporte sept nœuds étiquetés de 1 à 7) et de hauteur 3.
- Sa racine est le nœud 1, de profondeur 0.
- Les nœuds 2 et 3 sont de profondeur 1.
- Les nœuds 4, 5 et 6 sont de profondeur 2.
- Le nœud 7 est de profondeur 3.
- Ses feuilles sont les nœuds 4, 6 et 7.
- Ses nœuds internes sont les nœuds 2, 3 et 5.
- Le nœud 2 comporte 2 fils : le nœud 4 (son fils gauche) et le nœud 5 (son fils droit).
- Le nœud 3 est le père du nœud 6.

Exemple 7.8 - *implémentation en Ocaml*

Les arbres binaires étiquetés peuvent être implémentés en Ocaml en définissant le type suivant :

```
type 'a bintree = Vide | Noeud of 'a * 'a bintree 'a * bintree
```

On déclare alors l'arbre de l'exemple précédent comme ceci :

```
let a = Noeud(1,
    Noeud(2,
        Noeud(4,Vide,Vide),
        Noeud(5,
            Noeud(7,Vide,Vide),
            Vide)),
    Noeud(3,
        Vide,
        Noeud(6, Vide, Vide)))
```

La fonction suivante permet de déterminer la taille d'un arbre binaire.

```
let taille arbre = match arbre with
    |Vide -> 0
    |Noeud (e, g, d) -> 1 + taille g + taille d

let () = print_int (taille a ; print_string "\n")
```

Définition 7.9 - *arbres complets*

Un arbre binaire est dit *complet* si l'une des propriétés est vraie :

- il est vide
- ses sous-abres gauche et droit sont complets, et de même hauteur.

Proposition 7.10 - *lien entre hauteur et taille pour un arbre complet*

Tout arbre binaire complet de hauteur h possède exactement $2^{h+1} - 1$ noeuds.

Démonstration

On rappelle que pour tout arbre $\mathcal{A}$ on note $|\mathcal{A}|$ sa taille, i.e. son nombre de nœuds.
Montrons par récurrence sur h que pour tout arbre complet $\mathcal{A}$ de hauteur h, $|\mathcal{A}| = 2^{k+1} - 1$.

- L'unique arbre de hauteur -1 est l'arbre vide, qui est bien de taille $0 = 2^{-1+1} - 1$.
- Soit $h \in \mathbb{N}$ fixé. On suppose que tout arbre binaire complet de hauteur $h-1$ est de taille $2^k - 1$.
  On considère un arbre binaire complet $\mathcal{A}$ de hauteur h. Par définition, $\mathcal{A}$ possède un sous-arbre gauche $\mathcal{A}_g$ et un sous-arbre droit $\mathcal{A}_d$, tous deux complets et de hauteur $h-1$. Alors,

$$\begin{aligned} |\mathcal{A}| &= 1 + |\mathcal{A}_g| + |\mathcal{A}_d| \\ &= 1 + 2^k - 1 + 2^k - 1 && \text{Hypothèse de récurrence} \\ &= 2^{k+1} - 1 \end{aligned}$$

Par principe de récurrence, le prédicat est vrai pour tout arbre $\mathcal{A}$ de hauteur h $\in \mathbb{N} \cup -1$

Exemple 7.11 - *implémentation en C*

En C, on peut définir une structure d'arbre binaire dans laquelle on stocke la taille de l'arbre :

```
{C}
        struct Arbrebin{
            int taille;
            struct Arbrebin* gauche;
            struct Arbrebin* droit;
        };
        type def struct Arbrebin arbrebin;
```

L'arbre vide est représenté par le pointeur `NULL`. Pour pouvoir accéder à la taille, y compris de l'arbre vide, on implémente une fonction taille :

```
{C}
        int taille(arbrebin* a){
            if (a == NULL){
                return 0;
            }
            return a->taille;
        }
```

On crée alors un abre binaire quelconque en assemblant ses sous-arbres gauche et droit :

```
{C}
        arbrebin* assemble_arbrebin(arbrebin* ssArbre_g, arbre* ssArbre_d){
            arbrebin* a = malloc(sizeof (arbrebin));
            assert(a != NULL);
            a->gauche = ssArbre_g;
            a->droite = ssArbre_d;
            a->taille = ssArbre_g->taille + ssArbre_d->taille + 1;
        }
```

On pense également à créer un destructeur pour pouvoir libérer la mémoi allouée sur le tas :

```
{C}
        void libere_arbre(arbrebin* a){
            if (a != NULL){
                libere_arbrebin(a->gauche);
                libere_arbrebin(a->droit);
                free(a);
            }
        }
```

Pour créer une feuille,


```
{C}
        arbrebin* feuille = assemble_arbrebin(NULL,NULL);
```

La fonction suivante permet de tester si un arbre binaire est complet ou non :

```
{C}
        bool est_complet(arbrebin* a){
```

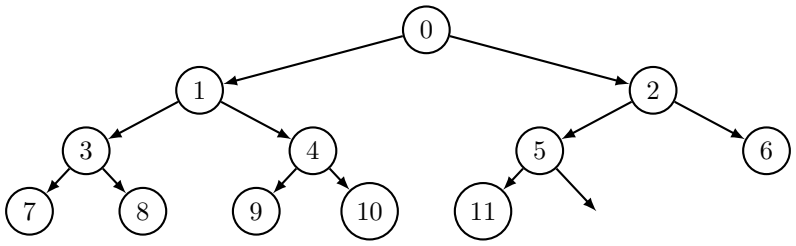
Remarque 7.12 - *comparaison entre C et Ocaml*

Dans ce précédent exemple, on représente des arbres binaires non étiquetés, mais on pouvait tout à fait ajouter un champ étiquette. Par ailleurs, en C, on a naturellement une structure mutable, ce qui permet de modifier la structure d'un arbre ou ses étiquettes. Ce n'est pas le cas en Ocaml.

Exemple 7.13 - *autre structure de définition*

Dans le cas d'un arbre binaire complet ou *presque complet* (tous les niveaux de profondeur sont remplis sauf éventuellement le dernier qui est rempli de gauche à droite), on peut également utiliser une structure de tableau. Pour cela, il suffit de numéroter les nœuds de haut en bas et de gauche à droite, à partir de 0, en remarquant que:

- S'il existe, le fils gauche du nœud d'étiquette i porte le numéro $2i+1$
- S'il existe, le fils gauche du nœud d'étiquette i porte le numéro $2i+2$
- Le père du noeud d'étiquette i (sauf pour la racine qui n'a pas de père) porte le numéro $\lfloor \frac{i-1}{2} \rfloor$



En C, pour manipuler de tels arbres, on peut utiliser la structure suivante :

{C}
```
typedef struct ArbreBC{
    int taille;
    int* etiquettes; //ou un autre type, en fonction des étiquettes
} arbreBC;

arbreBC* cree_arbreBC(int nb_noeuds){
    arbreBC* a = malloc(sizeof (arbreBC));
    assert(a != NULL); //allocation réussie
    a->taille = nb_noeuds;
    a->etiquettes = malloc(nb_noeuds * sizeof (int)); // attention, étiquettes non renseignées.
    assert(a->etiquettes != NULL);
    return a;
}

void libere_arbreBC(arbreBC* a){
    if (a != NULL){
        free(a->etiquettes);
        free(a);
    }
}
```

Remarque 7.14

Dans la littérature on peut trouver plusieurs définitions, parfois contradictoires, d'arbres binaires complets ou parfaits. **Là encore, il convient de bien lire les énoncés !**

7.2 Parcours d'arbres binaires

Parcourir un arbre signifie parcourir l'ensemble de ses noeuds (par exemple dans le but de rechercher un élément particulier, éventuellement pour le modifier), une et une seule fois. Il existe plusieurs façons de parcourir un arbre binaire, mais nous allons décrire les parcours en profondeur et en largeur (dans les deux cas, le parcours s'effectue de gauche à droite).

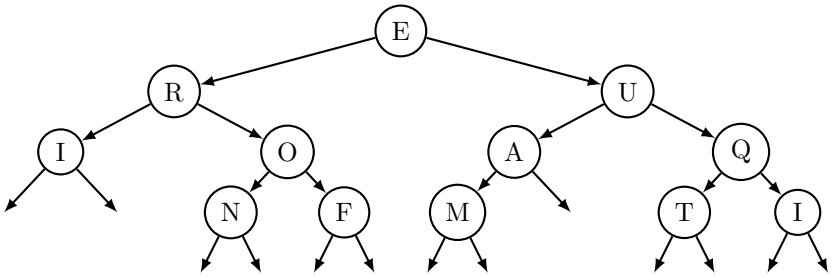
7.2.1 Parcours en profondeur

Définition 7.15 - *parcours infixe, postfixe, préfixe*

On effecue un *parcours en profondeur* lorsqu'on parcourt les noeuds en descendant le plus profondément possible dans l'arbre avant de remonter pour parcourir le reste de l'arbre.
Il existe 3 parcours classiques en profondeur :

- le *parcours préfixe* traite chaque noeud avant de parcourir ses deux sous-arbres (gauche puis droit).
- le *parcours infixe* traite chaque noeud entre le parcours du sous-arbres gauche et celui du sous-arbre droit.
- le *parcours postfixe* traite chaque noeud après avoir parcouru ses deux sous-arbres (le gauche puis le droit).

Exemple 7.16 - *application des parcours en profondeur*



Si on parcourt l'arbre binaire ci-dessus pour afficher l'étiquette de chaque noeud, on obtient :

- parcours préfixe : ERIOFUAMQTI
- parcours infixe : IRNOFEMAUTQI
- parcours postfixe : INFORMATIQUE

En reprenant le type défini dans l'exemple (7.8), cela donne :

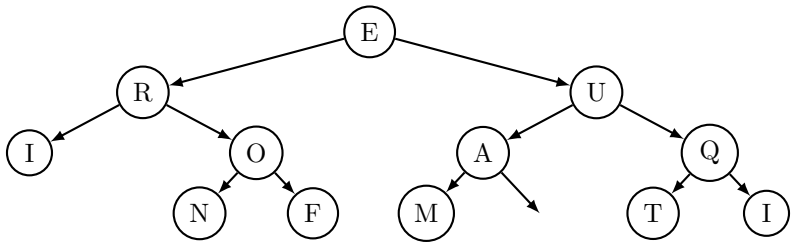
```
let rec affiche_infixe(arbrebin:char bintree) =
match arbrebin with
    | Vide -> ()
    | Noeud (e,g,d) -> affiche_infixe g;
    print_char e;
    affiche_infixe d;
```

Remarque 7.17 - *compatibilité de ces parcours*

Les parcours en profondeur sont particulièrement adaptés à la définition inductive des arbres binaires et s'implémentent avec des fonctions récursives.

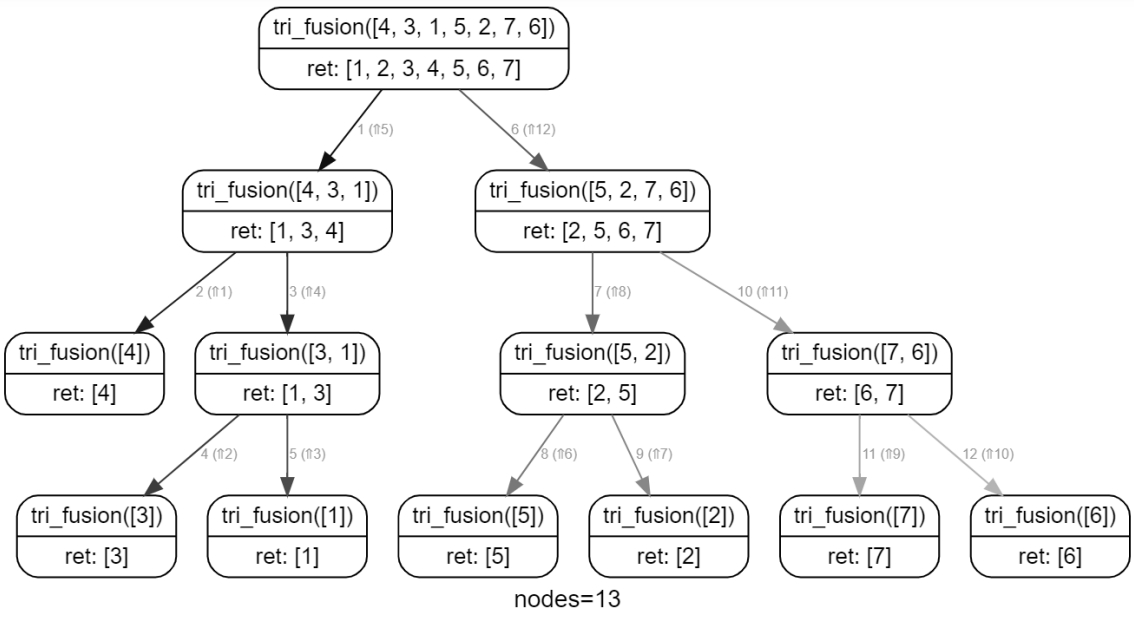
Remarque 7.18 - *changement de pratique*

Maintenant que nous avons manipulé des arbres binaires en gardant en tête que chaque noeud doit avoir exactement deux sous-arbres (éventuellement vides), on peut sans ambiguïté ne plus représenter les sous-arbres vides. Cela donne pour l'exemple précédent :



Exemple 7.19 - *cas du tri fusion*

Lorsqu'on fait appel à une fonction récursive, avec appels multiples, on peut représenter ces appels à l'aide d'un arbre (exemple du tri fusion ci-dessous) et l'empilement des blocs d'activation dans la pile d'appels correspond alors à un parcours **postfixe** . Ainsi les sous-arbres de chaque noeud sont bient traités **avant** le noeud considéré.



7.2.2 Parcours en largeur

Définition 7.20 - *parcours en largeur*

On effectue un *parcours en largeur* lorsqu'on parcourt les noeuds par niveau de profondeur : on explore la racine, puis les noeuds de profondeur 1 **de gauche à droite**, puis ceux de profondeur 2, etc.

Exemple 7.21 - *application du parcours en largeur*

Un parcours en largeur de l'arbre donné dans l'exemple 7.16 affiche :

ERUIOAQNFMTI

.

Exemple 7.22 - *implémentation en OCaml*

On suppose avoir défini une structure de file ainsi que les opérations élémentaires associées (voir section 5.2.3). La fonction suivante permet alors d'effectuer un parcours en largeur d'un arbre binaire :

```
type a' queue = {entrants : 'a list ; sortants : 'a list}
(* empty_queue : unit -> 'a queue
is_empty : 'a queue -> bool
enfile : 'a -> 'a queue -> 'a queue
defile : 'a queue -> 'a * 'a queue *)

let affiche_largeur arbrebin =
    let rec affiche niveau suivant = match niveau with
        (* manipule deux files d'arbres binaires :
            - niveau en cours d'affichage
            - suivant : niveau suivant *)
        | f when is_empty f -> if not is_empty suivant then affiche suivant empty_queue
        | _ -> let a, file = defile niveau in
            begin match a with
                | Vide -> affiche file suivant
                | Noeud (e, g, d) ->
                    print_char e;
                    let next = enfile d (enfile g suivant) in affiche file next
                    (* les sous-arbres vont au niveau suivant *)
            end
    in affiche {entrants = [arbrebin]; sortants = []}
```

Remarque 7.23 - *sur l'implémentation d'un arbre quasi-complet ou complet*

On note que dans un parcours en largeur, la définition inductive des arbres n'est pas naturellement exploitable. Si, en revanche, on utilise une structure de tableau comme mentionné dans (7.13), cela revient à un parcours séquentiel du tableau.

7.3 Arbres binaires de recherche

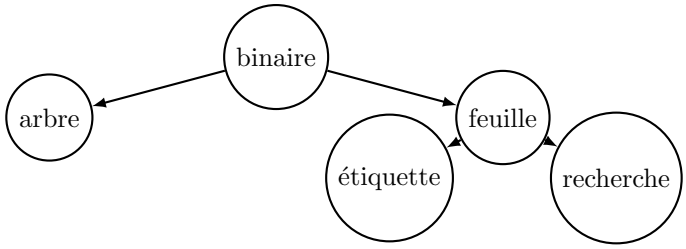
Définition 7.24 - *arbres binaires de recherche (ABR)*

Un *arbre binaire de recherche* (abrégé en ABR, BST (Binary Search Tree) en anglais) est un arbre binaire étiqueté tel que :

- l'ensemble des étiquettes est muni d'un **ordre total**
- pour tout noeud d'étiquette e,
  - l'éventuel sous-arbre gauche comporte des noeuds étiquetés par des valeurs inférieures à e
  - l'éventuel sous-arbre droit comporte des noeuds étiquetés par des valeurs supérieures à e

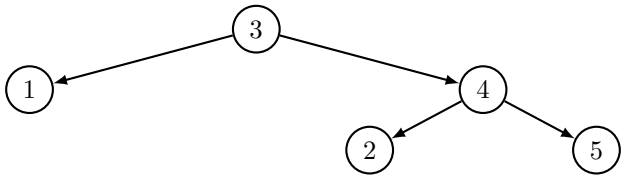
Exemple 7.25 - *cas de l'ordre lexicographique*

En munissant l'ensemble des mots de l'ordre lexicographique (i.e. alphabétique), l'arbre binaire suivant est un arbre binaire de recherche :



Exemple 7.26 - *contre-exemple d'ABR*

L'arbre suivant n'est pas un ABR :



Remarque 7.27 - *Définition inductive d'un ABR*

- L'arbre vide est un ABR.
- Un arbre binaire comportant une racine et deux sous-arbres qui sont des ABR et tel que toutes les étiquettes de son sous-arbre gauche (resp. droit) sont inférieures (resp. supérieures) à celle de sa racine est un ABR.

Proposition 7.28 - *caractérisation d'un ABR*

Un arbre binaire étiqueté $\mathcal{A}$ est un ABR si et seulement si la liste des étiquettes de $\mathcal{A}$ obtenue par un parcours infixe de $\mathcal{A}$ est triée.

Démonstration

$\Longrightarrow$ On raisonne par induction structurelle sur $\mathcal{A}$.

- Si $\mathcal{A}$ est l'arbre vide, c'est immédiat.
- Si $\mathcal{A}$ est construit à partir d'une racine étiquetée par la valeur r, d'un sous-arbre gauche $\mathcal{A}_g$ et d'un sous-arbre droit $\mathcal{A}_d$, on suppose la propriété vraie pour les sous-arbres $\mathcal{A}_g$ et $\mathcal{A}_d$.
  On note respectivement `lg` et `ld` les listes d'étiquettes obtenues par un parcours infixe de $\mathcal{A}_g$ et $\mathcal{A}_d$ respectivement. Par définition d'un parcours infixe, la liste `l` des étiquettes de $\mathcal{A}$ das l'ordre infixe est telle que `l` = `lg @ [r] @ ld`.
- Si $\mathcal{A}$ est un ABR, tous les éléments de `lg` sont inférieurs à l'étiquette de la racine tandis que ceux de `ld` sont supérieurs à `r`. D'autre part, $\mathcal{A}_g$ et $\mathcal{A}_d$ sont des ABR, par hypothèse d'induction, les listes `lg` et `lg` sont alors triées et on en déduit que `l` = `lg @ [r] @ ld` est triée.