

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.**

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 *fil gauche* (resp. *fil droit*) du nœud R la racine du sous-arbre gauche (resp. droit).
Voir Figure 1

Remarque 7.2 - à propos 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 : *Voir Figure 2*

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 *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 nœuds, 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 une valeur.

Exemple 7.7 - *concernant les précédentes définitions*

Considérons l'arbre binaire suivant. Voir *Figure 3*

- il s'agit d'un arbre binaire de taille 7, (il comporte 7 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, 5, 7
- Ses nœuds internes sont les nœuds 2, 3 et 5.
- 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,
    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-arbres 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 , $|\mathcal{A}| = 2^{h+1} - 1$.

- 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 $2^h - 1$.
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,

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

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 :

```
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.

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

On crée alors un arbre binaire quelconque en assemblant ses sous-arbres gauche et droit.

```
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 mémoire allouée sur le tas.

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

Pour créer une feuille,

```
arbrebin* feuille = assemble_arbrebin(NULL,NULL);
```

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

```
bool est_complet(arbrebin* a){
    if (a == NULL){return true;} // arbre vide
}
return taille(a->gauche) == taille(a->droit) && est_complet(a->gauche) && est_complet(a->droit)
```

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

Voir Figure 4

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

```
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 !**

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).

2.1 Parcours en profondeur

Définition 7.15 - *parcours infixe, postfixe, préfixe*

On effectue 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-arbre 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 -

Voir Figure 5

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;
```