

# I Rappels sur les arbres binaires de recherche

On utilisera les types suivants d'arbres binaires enracinés :

```
type 'a arb = V | N of 'a arb * 'a * 'a arb
```

```
typedef struct arb {
    struct arb* g;
    int r;
    struct arb* d;
} arb;
```

En C, un arbre vide est représenté par un pointeur `NULL`.

## Théorème : Induction sur les arbres binaires

Soit  $P(a)$  une propriété sur un arbre binaire  $a$ . Supposons :

- $P(V)$  vraie.
- Pour tout arbre binaire  $g, d$  et tout élément  $r$ , si  $P(g)$  et  $P(d)$  sont vraies, alors  $P(N(g, r, d))$  est vraie.

Alors  $P(a)$  est vraie pour tout arbre binaire  $a$ .

Remarque : On peut aussi raisonner par récurrence sur la hauteur de l'arbre ou le nombre de sommets.

## Définition : Hauteur

La hauteur  $h(a)$  d'un arbre binaire  $a$  est la longueur maximum d'un chemin (en nombre d'arêtes) de la racine à une feuille.

```
let rec h a = match a with
| V -> 0
| N(g, _, d) -> 1 + max (h g) (h d)
```

```
int h(arb* a) {
    if(!a) return 0;
    return 1 + max(h(a->g), h(a->d));
}
```

Si  $a$  est un arbre binaire, on note  $f(a)$  son nombre de feuilles,  $n(a)$  son nombre de nœuds,  $n_i(a)$  son nombre de nœuds internes (non feuilles) et  $h(a)$  sa hauteur.

### Exercice 1.

Montrer que si  $a$  est un arbre binaire strict (chaque nœud a 0 ou 2 fils) alors  $f(a) = n(a) + 1$ .

---



---



---



---



---

## Théorème : Formules sur les arbres binaires

Si  $a$  est un arbre binaire alors :

$$f(a) \leq 2^{h(a)}$$

$$h(a) + 1 \leq n(a) \leq 2^{h(a)+1} - 1$$

$$\log_2(n(a) + 1) - 1 \leq h(a) \leq n(a) - 1$$

On dit que  $a$  est équilibré si  $h(a) = O(\log(n(a)))$ .

### Exercice 2.

On note  $d(a)$  le diamètre d'un arbre binaire  $a$ , c'est-à-dire la longueur maximum d'un chemin entre deux sommets de  $a$ .

1. Écrire une fonction `d : 'a arb -> int` qui calcule le diamètre d'un arbre binaire en temps linéaire.
2. Généraliser avec un arbre enraciné quelconque défini par `type 'a arb = N of 'a arb list`.
3. Avec quelle complexité peut-on trouver le diamètre d'un graphe quelconque ?

[illegible]

Si  $a$  est un arbre binaire, on note  $r(a), g(a), d(a)$  respectivement la racine, le sous-arbre gauche et le sous-arbre droit de  $a$ .

### Définition : Arbre binaire de recherche

Un arbre binaire de recherche (ABR) est un arbre binaire  $a$  tel que pour chaque nœud  $b$  de  $a$ , toutes les étiquettes de  $g(b)$  sont inférieures ou égales à  $r(b)$  et toutes les étiquettes de  $d(b)$  sont supérieures ou égales à  $r(b)$ .

Remarque : La propriété d'ABR est héréditaire : un sous-arbre d'un ABR est un ABR. C'est utile pour écrire une fonction récursive et raisonner par récurrence sur les ABR.

### Exercice 3.

Prouver ou réfuter l'affirmation suivante : un arbre binaire  $a$  est un ABR si et seulement si  $f(a)$ ,  $g(a)$  sont des ABR et  $r(g(a)) \leq r(a) \leq r(d(a))$ .

---

---

---

---

### Théorème

Soit  $a$  un arbre binaire. Alors  $a$  est un ABR si et seulement si son parcours infixe est croissant.

Remarque : Cela donne un algorithme simple pour tester si un arbre binaire est un ABR.

---

Preuve :

---

---

---

---

---

---

---

```

let rec add e a = match a with
| V -> N(V, e, V)
| N(g, r, d) ->
    if e < r then N(add e g, r, d)
    else N(g, r, add e d)

let rec has e a = match a with
| V -> false
| N(g, r, d) ->
    if e = r then true
    else if e < r then has e g
    else has e d

let rec del_min a = match a with
| V -> failwith "Empty"
| N(V, r, d) -> r, d
| N(g, r, d) ->
    let m, g' = del_min g in
    m, N(g', r, d)

let rec del e a = match a with
| V -> V
| N(g, r, d) ->
    if e = r then
        match g, d with
        | V, _ -> d
        | _, V -> g
        | _, _ ->
            let m, d' = del_min d in
            N(g, m, d')
    else if e < r then N(del e g, r, d)
    else N(g, r, del e d)

```

---



---

```

arb* add(int e, arb* a) {
    if(!a) return N(NULL, e, NULL);
    if(e < a->r)
        a->g = add(e, a->g);
    else
        a->d = add(e, a->d);
    return a;
}

bool has(int e, arb* a) {
    if(!a) return false;
    if(e == a->r) return true;
    if(e < a->r) return has(e, a->g);
    return has(e, a->d);
}

int del_min(arb** a) {
    if(!(*a)->g) {
        int r = (*a)->r;
        *a = (*a)->d;
        return r;
    }
    return del_min(&(*a)->g);
}

arb* del(int e, arb* a) {
    if(!a) return NULL;
    if(e == a->r) {
        if(!a->d) return a->g;
        if(!a->g) return a->d;
        int m = del_min(&a->d);
        a->r = m;
        return a;
    }
    if(e < a->r)
        a->g = del(e, a->g);
    else
        a->d = del(e, a->d);
    return a;
}

```

---

## I.1 Arbre rouge-noir

### Définition : Arbre rouge-noir

Un arbre rouge-noir (ARN)  $a$  est un ABR dont les noeuds sont coloriés en rouge ou noir et vérifiant :

- La racine est noire (non obligatoire, mais rend le code plus simple).
- Si un sommet est rouge, ses éventuels fils sont noirs.
- Le nombre de noeuds noirs le long de n'importe quel chemin de la racine à un emplacement vide (**V**) est le même, que l'on appelle hauteur noire et note  $h_b(a)$ .

### Théorème

Soit  $a$  un ARN avec  $n(a)$  noeuds et de hauteur  $h(a)$ . Alors :

- $h(a) \leq 2h_b(a)$ .
- $n(a) \geq 2^{h_b(a)} - 1$ .
- $h \leq 2\log_2(n + 1)$  (un arbre rouge-noir est donc équilibré).

Preuve :

---



---

---

---

---

---

---

---

---

---

## II *Treap*

On peut montrer qu'un arbre binaire de recherche (ABR) construit en ajoutant un à un  $n$  entiers choisis « uniformément au hasard » a une hauteur moyenne  $O(\log(n))$ .

Si les éléments à rajouter ne sont pas générés aléatoirement, mais sont tous connus à l'avance, on peut commencer par les mélanger aléatoirement puis les rajouter dans cet ordre aléatoire pour obtenir à nouveau une hauteur moyenne  $O(\log(n))$ . Pour cela, on considère l'algorithme suivant (mélange de Knuth), où `Random.int (i+1)` renvoie un entier uniformément au hasard entre 0 et  $i$  :

```
let shuffle a =  
  for i = 1 to Array.length a - 1 do  
    swap a i (Random.int (i + 1))  
  done
```

1. Écrire la fonction `swap` : `'a array -> int -> int` utilisée par `shuffle`, telle que `swap t i j` échange `t.(i)` et `t.(j)`.

2. Montrer que `shuffle t` applique une permutation choisie uniformément au hasard sur le tableau `t`.

Lorsque la totalité des éléments à rajouter n'est pas connue à l'avance, on peut utiliser une structure appelée **arbretas** qui est un arbre binaire (défini par `type 'a arb = V | N of 'a * 'a arb * 'a arb`) dont les noeuds sont étiquetés par des couples (élément, priorité), où la priorité est un nombre entier choisi uniformément au hasard au moment de l'ajout de l'élément. De plus :

- les éléments doivent vérifier la propriété d'ABR.
  - la priorité d'un sommet doit être inférieure à la priorité de ses éventuels fils (propriété de tas min sur les priorités).
3. Dessiner un arbretas dont les couples (élément, priorité) sont : (1, 4), (5, 6), (3, 8), (2, 2), (0, 7).

4. Étant donnés des éléments et priorités tous distincts, montrer qu'il existe un unique arbretas les contenant.

---

---

---

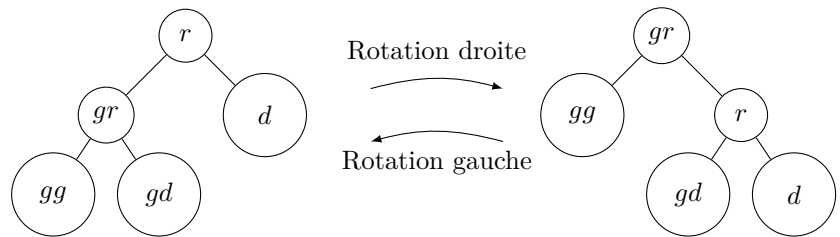
---

---

---

---

Nous allons utiliser des opérations de rotation sur un arbretas  $\mathbf{N}(\mathbf{r}, \mathbf{N}(\mathbf{gr}, \mathbf{gg}, \mathbf{gd}), \mathbf{d})$  :



5. Écrire une fonction `rotd` effectuant une rotation droite sur un arbre  $\mathbf{N}(\mathbf{r}, \mathbf{N}(\mathbf{gr}, \mathbf{gg}, \mathbf{gd}), \mathbf{d})$ .

---

---

---

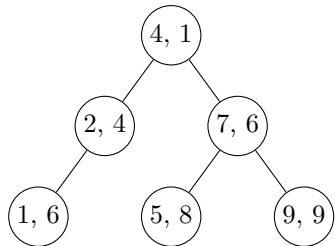
---

---

On supposera définie dans la suite une fonction `rotg` pour effectuer « l'inverse » d'une rotation droite. On remarquera que, si  $\mathbf{a}$  est un ABR, `rotg a` et `rotd a` sont aussi des ABR.

Pour ajouter un sommet  $s$  dans un arbretas (en conservant la structure d'arbretas), on l'ajoute comme dans un ABR classique (en ignorant les priorités) puis, si sa priorité est inférieure à celle de son père, on applique une rotation sur son père pour faire remonter  $s$  et on continue jusqu'à rétablir la structure d'arbretas.

6. Dessiner l'arbretas obtenu en rajoutant  $(6, 0)$  à l'arbretas suivant :



7. Écrire une fonction utilitaire `prio` renvoyant la priorité de la racine d'un arbre (on renverra `max_int`, c'est à dire le plus grand entier représentable en base 2 sur le processeur, si cet arbre est vide).

---

---

---

8. Écrire une fonction `add a e` ajoutant `e` (qui est un couple (élément, priorité)) à un arbretas `a`, en conservant la structure d'arbretas.

---

---

---

---

---

---

---

---

Pour supprimer un élément d'un arbretas, on commence par le chercher comme dans un ABR classique (en ignorant les priorités) puis on le fait descendre avec des rotations jusqu'à ce qu'il devienne une feuille qu'on peut alors supprimer librement.

9. Écrire une fonction `del` supprimant un élément dans un arbretas, en conservant la structure d'arbretas.

---

---

---

---

---