

TD n°2 - Correction

Arbres binaires de recherche

Exercice 1 L'algorithme n'est pas correct. Par exemple l'arbre $(2 < 3 > 5) < 4 > 5$ n'est pas un ABR. Un bon algorithme est, par exemple :

```
1 Fonction estABR(a : arbre) : booléen
2 début
3   si (estVide(a)) alors
4     retourner vrai;
5   si (estVide(G(a))) alors
6     gauche := clé(a);
7   sinon
8     gauche := max(G(a));
9     // calcule la clé maximum de l'arbre
10  si (estVide(D(a))) alors
11    droite := clé(a);
12  sinon
13    droite := min(D(a));
14    // calcule la clé minimum de l'arbre
15  retourner (gauche ≤ clé(a) ≤ droite ∧ estABR(G(a)) ∧ estABR(D(a)));
16 fin
```

Exercice 2

Question 1.

```
1 Fonction main(a : arbre, k : entier) : entier
2 début
3   i := 1;
4   VisiteGRD(a);
5   retourner T[k];
6 fin
```

```
1 Procédure VisiteGRD(a : arbre)
2 début
3   si (¬estVide(a)) alors
4     VisiteGRD(G(a));
5     T[i] := clé(a);
6     i := i + 1;
7     VisiteGRD(D(a));
8 fin
```

La complexité de cette méthode est $\Theta(n)$.

Question 2.

```
1 Fonction RecherchePos(a : arbre, k : entier) : booléen × entier
2 début
3   si (estVide(a)) alors
4     └ retourner (faux, 0);
5   (b, n) := RecherchePos(G(a), k);
6   si (b = vrai) alors
7     └ retourner (vrai, n);
8   sinon
9     si (k = n + 1) alors
10      └ retourner (vrai, clé(a));
11     sinon
12       (b', n') := RecherchePos(D(a), k - n - 1);
13       si (b' = vrai) alors
14         └ retourner (vrai, n');
15       sinon
16         └ retourner (faux, n + n' + 1);
17 fin
```

La complexité de la fonction `RecherchePos` est $\Theta(k)$.

Question 3. On remarque qu'il est nécessaire de parcourir la branche gauche d'un arbre uniquement si k est supérieur au nombre de nœuds du fils gauche. Une possibilité est donc de calculer préalablement la taille de fils gauche avant de décider si on effectue la recherche à gauche ou à droite.

```
1 Fonction RecherchePos(a : arbre, k : entier) : arbre
2 début
3   tailleFG := taille(G(a));
4   si (tailleFG ≥ k) alors
5     └ retourner RecherchePos(G(a), k);
6   sinon
7     si (tailleFG = k - 1) alors
8       └ retourner a;
9     sinon
10      └ retourner RecherchePos(D(a), k - tailleFG - 1);
11 fin
```

Le nombre maximum d'appel récursif de la fonction `RecherchePos` étant h , la hauteur de l'arbre, on pourrait dire naïvement que la complexité dans le pire cas de cet algorithme est $O(h)$. Ce n'est le cas que si la fonction `taille` est en $\Theta(1)$.

Or, normalement une fonction `taille` pour les ABR est en $\Theta(n)$ (où n est la taille de l'arbre). Dans ce cas, si l'on recherche le premier élément d'un arbre de taille n complètement déséquilibré à gauche, la complexité de la fonction `RecherchePos` est en réalité $O(n^2)$ (cela se vérifie par récursion sur la taille de l'arbre).

Question 4. La seule manière d'obtenir une fonction `taille` en $\Theta(1)$ est de modifier la structure de l'arbre pour que chaque nœud stocke sa taille.

Pour que cette information sur la taille reste cohérente, il faut modifier les fonction d'insertion et de suppression. Ces modifications peuvent être faites sans changer leur complexités respectives.

- Lors de l’insertion il suffit de rajouter sur le chemin du nouveau nœud jusqu’à la racine 1 à la taille de chaque arbre. On peut aisément voir que lorsqu’on ajoute un nœud à un ABR, on augmente la taille de 1 récursivement sur les sous-arbres concernés. Il s’agit d’incrémenter $O(h)$ compteurs (opération en temps constant sur chaque nœud) donc la complexité reste $O(h)$.
- Concernant la suppression, en utilisant la méthode de suppression qui utilise le successeur, il faut décrémenter de 1 la taille de tout les nœuds allant de la racine à l’ancienne position du successeur. La nouvelle taille de la racine est l’ancienne taille -1 . Il s’agit de décrémenter $O(h)$ compteurs (opération en temps constant sur chaque nœud) donc la complexité reste $O(h)$.

Exercice 3

Question 1.

1. Supposons que l’élément minimum m ait un fils gauche g . Par définition d’un ABR, $g \leq m$. Puisque toutes les clés sont distinctes, on a même $g < m$, ce qui contredit la minimalité de m .
2. Appelons a le nœud en question.
 - Puisque a a un fils droit, il existe des éléments de l’arbre supérieurs à a . Donc a a un successeur dans l’arbre.
 - Montrons que ce successeur s est un descendant de a . Par l’absurde, on distingue deux cas :
 - Si a était un descendant de s , alors il serait forcément dans le sous-arbre gauche de s puisqu’il est plus petit. Mais les nœuds du sous-arbre droit (non vide) de a sont alors plus grands que a et plus petits que s , une contradiction.
 - Sinon, et si s n’est pas non plus un descendant de a , soit b le nœud le plus profond tel que a soit dans le sous-arbre gauche de b et s dans son sous-arbre droit (b est le premier ancêtre commun de a et s). On a alors $a < b < s$, une contradiction.

Ainsi, s est un descendant de a .

- Le nœud s est forcément dans le sous-arbre droit de a puisqu’il est plus grand.
 - Enfin, maintenant que l’on a montré que s est dans le sous-arbre droit de a , montrons qu’il en est le plus petit élément. Dans le cas contraire, l’élément minimal m du sous-arbre droit satisferait $a < m < s$, une contradiction puisque s est le successeur de a .
3. **Première méthode** (résumé) : appelons a le nœud en question et b le premier de ses ancêtres dont le fils gauche est aussi un ancêtre de a . D’abord, on montre de la même manière qu’au 2 que le prédécesseur d’un nœud ayant un fils gauche est le maximum de son sous-arbre gauche. Puis on remarque que a est le maximum du sous-arbre gauche de b , ce qui conclut.

Seconde méthode (plus détaillée) : appelons a le nœud en question et s son successeur, s’il existe. Montrons d’abord que s est un ancêtre de a . Comme au 2, par l’absurde, on distingue deux cas :

- Si s était un descendant de a , puisqu’il est plus grand il serait dans le sous-arbre droit de a , qui est vide : contradiction.
- Sinon, et si a n’est pas non plus un descendant de s , soit b le nœud le plus profond tel que a soit dans le sous-arbre gauche de b et s dans son sous-arbre droit (b est le premier ancêtre commun de a et s). On a alors $a < b < s$, une contradiction.

Ainsi, s est un ancêtre de a . Par ailleurs, a est forcément dans le sous-arbre gauche de s puisqu’il est plus petit, donc s est un ancêtre de a dont le fils gauche est aussi un ancêtre de a . Il nous reste à montrer que s est le premier ancêtre à avoir cette propriété. Si ce n’était pas le cas, il y aurait un nœud b , descendant de s (dans le sous-arbre gauche) et ancêtre de a , tel que a soit dans son sous-arbre gauche. On aurait donc $a < b < s$, une contradiction.

Question 2.

```

1 Fonction min(a : arbre) : arbre
2 début
3   si (estVide(G(a))) alors
4     └ retourner a;
5   retourner min(G(a));
6 fin

```

```

1  Fonction succ( $a$  : arbre) : arbre
2  début
3  | si ( $\neg$ estVide(D( $a$ ))) alors
4  | | retourner min(D( $a$ ));
5  |  $y :=$  père( $a$ );
6  | tant que ( $\neg$ estVide( $y$ )  $\wedge$   $x =$  D( $y$ )) faire
7  | |  $x := y$ ;
8  | |  $y :=$  père( $y$ );
9  | retourner  $y$ ;
10 fin

```

On effectue un simple parcours de l'arbre à partir du nœud x soit vers le bas, soit vers le haut, donc la complexité en temps est $O(h)$. Si on veut calculer la complexité en fonction du nombre n de clés, dans le meilleur des cas la complexité est $O(\lg n)$ et dans le pire des cas $O(n)$.