

## Quick 2 – 8 novembre 2024 – Durée 1 h

Une feuille A4 manuscrite recto-verso autorisée. Les calculatrices et les téléphones (incluant smartphone, tablettes,... tout ce qui contient une interface réseau) sont interdits.

Les dictionnaires pour les personnes de langue étrangère sont autorisés.

Le barème est indicatif.

**Vous êtes vivement encouragés à illustrer vos recherches et vos réponses par des schémas.**

### 1 Arbre Binaire de Recherche (6 points)

Pour cet exercice, on utilisera le type abstrait Arbre binaire vu en cours :

- `Arbre_vider()` : renvoie un arbre vide.
- `Noeud(g, e, d)` : renvoie un arbre dont la racine est étiquetée par  $e$ , et dont les sous-arbres gauche et droit sont  $g$  et  $d$ .
- `Est_vider(a)` : renvoie vrai si et seulement si  $a$  est l'arbre vide.

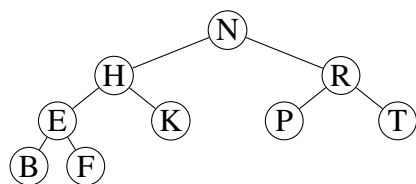
Et, pour les arbres non vides :

- `FG(a)` et `FD(a)` : renvoient respectivement le fils gauche et le fils droit de  $a$ .
- `Clé(a)` : renvoie la clé du nœud à la racine de  $a$ .

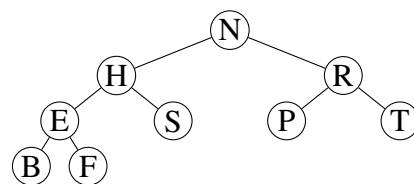
#### Question unique

Écrire une fonction qui vérifie si l'arbre binaire donné en argument est un arbre binaire de recherche correct.

*Exemples* : Voici deux arbres avec les réponses attendues.



La fonction renvoie *vrai*.



La fonction renvoie *faux*.

Vous pouvez écrire des fonctions auxiliaires si besoin.

*Conseil* : Inspirez-vous des exercices vus en TD !

Les critères qui seront pris en compte par les correcteurs sont :

- (3 points) L'algorithme résout-il correctement le problème posé ?
- (2 points) L'algorithme est-il efficace ?
- (1 point) L'algorithme est-il lisible, compréhensible ?

**Solution:** Comme pour les ABRI, il ne suffit pas de vérifier que chaque nœud est compris entre son fils gauche et son fils droit, il faut également propager cette information dans tout le sous-arbre (par exemple dans l'énoncé l'arbre de droite est incorrect car  $S > N$ ).

Une version simple (à condition d'avoir codé  $\max$  et  $\min$  comme vu en cours) mais inefficace est donc :

`VerifABR_naif(a) :`

```

┌   si a est vide alors
├     renvoyer Vrai
├   si  $FG(a)$  non vide et  $Clé(a) < \max(FG(a))$  alors
├     renvoyer Faux
├   si  $FD(a)$  non vide et  $Clé(a) > \min(FD(a))$  alors
├     renvoyer Faux
└   renvoyer VerifABR_naif(FG(a)) et VerifABR_naif(FD(a))

```

Il est plus efficace de calculer les maximum et minimum en même temps que la vérification. On écrit donc une fonction auxiliaire :

`VerifABR(a) :`

```

┌   si a est vide alors
├     renvoyer Vrai
├   sinon
├      $min, \text{verif}, max := \text{VerifABR\_rec}(a)$ 
├     renvoyer verif
└

```

`VerifABR_rec(a) :`

```

┌   si  $FG(a)$  non vide alors
├      $ming, \text{verif}g, maxg := \text{VerifABR\_rec}(FG(a))$ 
├   sinon
├      $ming, \text{verif}g, maxg := Clé(a), \text{Vrai}, Clé(a)$ 
├   si  $FD(a)$  non vide alors
├      $mind, \text{verif}d, maxd := \text{VerifABR\_rec}(FD(a))$ 
├   sinon
├      $mind, \text{verif}d, maxd := Clé(a), \text{Vrai}, Clé(a)$ 
├   si  $maxg \leq Clé(a)$  et  $Clé(a) \leq mind$  alors
├     renvoyer  $ming, \text{verif}g \& \& \text{verif}d, maxd$ 
├   sinon
├     renvoyer 0, Faux, 0
└

```

Alternativement, on peut ajouter à notre fonction de vérification deux paramètres  $min$  et  $max$  qui délimitent l'intervalle de valeurs entre lesquelles les clés de l'arbre doivent être comprises. Il faut dans ce cas gérer l'initialisation de ces valeurs.

### Détail du barème

- L'algorithme résout-il correctement le problème posé ?
  - construction récursive correcte (la récursion se termine, pas d'appel de clé ou FG sur des arbres vides)
  - l'algorithme compare (correctement) au moins  $clé(A)$  avec les clés de ses fils
  - l'algorithme compare (correctement)  $clé(A)$  avec le max de son fils gauche et le min de son fils droit
- L'algorithme est-il efficace ?
  - on coupe les appels récursifs dès qu'il est clair que l'arbre est incorrect
  - dans le cas de l'algo correct, on évite les calculs de min/max à part
- L'algorithme est-il lisible, compréhensible ?
  - les différents cas sont faciles à distinguer
  - les variables sont bien nommées, initialisées, etc.
  - les booléens sont utilisés élégamment

## 2 Preuve d'algorithme (15 points)

Soit un tableau  $T$  (indexé de 0 à  $n-1$ ), supposé trié en ordre croissant, mais pouvant contenir des doublons.

L'objectif de ce problème est de ne conserver qu'un seul exemplaire de chaque valeur présente dans  $T$ . On demande à ce que l'algorithme place ces valeurs uniques à gauche du tableau, et qu'il renvoie le nombre d'éléments distincts rencontrés. Une fois l'algorithme exécuté, les valeurs contenues dans  $T$  à droite des valeurs uniques n'a pas d'importance.

Par exemple, pour  $T = [E, E, G, K, K, K, S]$ , l'objectif est de modifier ce tableau pour que  $T = [E, G, K, S, ?, ?, ?]$  (où les valeurs ? sont quelconques) et de renvoyer 4.

On propose l'algorithme suivant :

```

Uniques ( $T$ ) :
  Données : Un tableau  $T$  de  $n$  éléments en ordre croissant
  Résultat : Le nombre d'éléments distincts dans  $T$ 
  Effet de bord :  $T$  est modifié pour placer ses éléments uniques à gauche
1   $i := 0$ 
2   $j := 1$ 
3  tant que  $j < n$  faire
4    si  $T[i] \neq T[j]$  alors
5       $i := i + 1$ 
6      Échanger  $T[i]$  et  $T[j]$ 
7     $j := j + 1$ 
8  renvoyer  $i + 1$ 

```

1. (2 points) Justifiez que cet algorithme se termine.

**Solution:** Le variant  $n - j$  convient : il est bien entier, positif, et strictement décroissant (il diminue de 1 à chaque itération).

2. On appelle  $r$  la valeur renvoyée par l'algorithme.

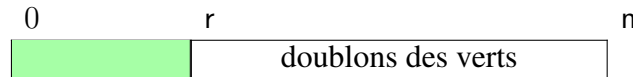
Une des postconditions consiste à exprimer que les valeurs laissées en fin de tableau sont superflues :

*Après exécution de l'algorithme, chaque élément de  $T$  d'indice compris entre  $r$  et  $n - 1$  est un doublon d'un élément présent entre 0 et  $r - 1$ .*

- (1 point) Représentez sous forme de schéma la postcondition énoncée ci-dessus.
- (2 points) Exprimez un invariant vérifié par la boucle **tant que** de notre algorithme (juste avant la ligne 4), sous la forme d'une phrase en français, en faisant référence explicitement à des indices du tableau.  
Votre invariant devra bien entendu permettre de démontrer la postcondition donnée ci-dessus.
- (1 point) Présentez également cet invariant sous forme de schéma.
- (1 point) Justifiez que cet invariant est vérifié après la ligne 2 de l'algorithme.
- (3 points) Démontrez que la boucle **tant que** maintient effectivement cet invariant.
- (2 points) Démontrez enfin la postcondition à l'aide de l'invariant en sortie de boucle.

**Solution:**

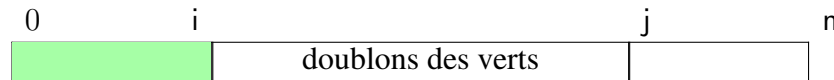
(a)



(b) Dans  $T$ , tous les éléments d'indices compris entre  $i + 1$  et  $j - 1$  sont des doublons d'un des éléments entre 0 et  $i$ .

$$\forall i < l < j, \exists 0 \leq k \leq i, T[l] = T[k]$$

(c)



(d) Comme  $i = 0$  et  $j = 1$ , il n'y a aucun élément dans  $T[i + 1..j - 1]$  donc l'initialisation de l'invariant est triviale.

(e) On suppose l'invariant vérifié en début d'itération.

Il y a deux cas à considérer :

— Si  $T[i] = T[j]$ , alors on peut incrémenter  $j$  et les éléments de  $T[i + 1..j' - 1]$  sont toujours bien des doublons d'éléments de  $T[0..i]$  ( $j' = j + 1$  donc  $j' - 1 = j$ , il n'y a que  $T[j]$  comme « nouvel » élément dans ce segment).

— Si  $T[i] \neq T[j]$  alors  $i' = i + 1$  et  $j' = j + 1$ . Comme on échange  $T[i']$  avec  $T[j]$ , le segment  $T[i' + 1..j' - 1]$  conserve globalement les mêmes éléments (son élément le plus à gauche a été déplacé à sa droite). Comme ses éléments étaient déjà des doublons d'éléments de  $T[0..i]$  par hypothèse (premier invariant en début d'itération), c'est encore le cas en fin d'itération.

(f) En sortie de boucle  $j = n$  et on renvoie  $r = i + 1$ .

On a donc bien les éléments de  $T[r..n - 1]$  qui sont tous des doublons d'éléments de  $T[0..r - 1]$ .

3. La postcondition étudiée plus haut ne suffit pas à assurer que notre algorithme réalise la tâche souhaitée, il nous en faut une autre.

(a) (2 points) Exprimez cette autre postcondition par une phrase en français, en faisant référence explicitement à des indices du tableau  $T$ .

(b) (1 point) Représentez également sous forme de schéma cette seconde postcondition.

Il n'est **pas demandé** de démontrer cette seconde postcondition.

### Solution:

(a) Les éléments d'indices 0 à  $r - 1$  dans  $T$  sont tous distincts (et en ordre croissant).

Si on préfère, on peut l'exprimer par la formule  $\forall 0 \leq i < r - 1, T[i] < T[i + 1]$ .

(La croissance n'est pas indispensable à la correction de l'algorithme, mais elle sera plus facile à prouver que la simple absence de doublons.)

*Pourquoi ces deux postconditions prouvent-elles que l'algorithme est correct ?*

Au final, les éléments de  $T$  auront donc été réordonnés (on n'en a perdu aucun puisqu'on fait des échanges), de sorte que tous les éléments d'indices compris entre 0 et  $r - 1$  soient uniques (car en ordre strictement croissant), et que les éléments entre  $r$  et  $n - 1$  soient des doublons.

(b)

