L3-INFO Algorithmique

Quick 2 – 18 novembre 2022 – durée 1h

Les documents et les téléphones (incluant smartphone, tablettes,... tout ce qui contient une interface réseau) sont interdits. Les calculettes sont autorisées.
Seuls les dictionnaires pour les personnes de langue étrangère sont autorisés.
Toutes les réponses doivent être justifiées. Le barème est indicatif.

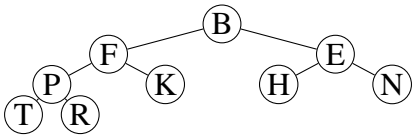
Dans tout ce devoir, on manipulera les arbres binaires avec les fonctions usuelles :
$ArbreVide()$, $Noeud(g, x, d)$, $EstVide(a)$, $FilsGauche(a)$, $FilsDroit(a)$, $Cle(a)$.
On notera systématiquement :
— n le nombre de nœuds dans l'arbre ;
— et h sa hauteur.

---

1 Nœuds plus petits que x dans un tas (*9 points*)

Dans cet exercice, on considère des tas-min (où les nœuds sont plus petits que leurs fils), représentés sous forme d'arbres binaires.

Pour tous les exemples donnés dans l'énoncé, on utilisera le tas :



Question 1 : *(2 points)*

Écrire un algorithme qui, étant donné un tas binaire T et une valeur x, affiche les étiquettes de **tous les nœuds** dont la valeur est **inférieure à** x.

Votre algorithme devra visiter le plus petit nombre de nœuds possible.

Avec le tas donné en exemple :
— si x vaut F on affichera BE ;
— si x vaut Q on affichera $BFPKEHN$
(pas forcément dans cet ordre).

Réponse 1 :

On profite de l'aspect ordonné des tas : dès qu'on rencontre une valeur $\geqslant x$, on n'a plus besoin d'aller tester les valeurs du sous-arbre.

Question 2 : *(3 points)*

Déterminer la complexité au pire de votre algorithme :
— d'abord en fonction du nombre n de nœuds dans le tas ;
— puis également en fonction de la hauteur h du tas.
On prendra soin de justifier les réponses proposées.

Réponse 2 :

```
Inferieurs(T, x)
Données : T : un tas binaire sous forme d'arbre, x : une valeur
Résultat : les éléments inférieurs à x dans T
si non EstVide(T) et Clé(T) < x alors
    Affiche Clé(T)
    Inferieurs(FilsGauche(T), x)
    Inferieurs(FilsDroit(T), x)
// Pas de "sinon", on ne fait rien dans l'autre cas
```

— Dans le pire des cas, toutes les valeurs dans T sont inférieures à x, ce qui nous amène à parcourir l'arbre intégralement. La complexité de l'algorithme est donc en $\mathcal{O}(n)$.
— Dans un tas on sait que tous les niveaux sont complets (sauf le dernier), ce qui nous assure que $2^h \leqslant n < 2^{h+1}$. On peut donc également affirmer que la complexité est en $\mathcal{O}(2^h)$.

Question 3 : *(2 points)*

Proposer un algorithme qui affiche les étiquettes de valeurs **supérieures** à x.

Dans notre exemple, pour $x = M$, on affichera $NPRT$ (pas forcément dans cet ordre).

Réponse 3 :

Cette fois l'aspect ordonné des tas ne nous aidera pas, on est obligé de tout parcourir.

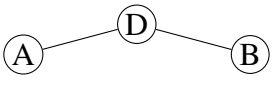
```
Superieurs(T, x)
Données : T : un tas binaire sous forme d'arbre, x : une valeur
Résultat : les éléments supérieurs à x dans T
si non EstVide(T) alors
    si Clé(T) > x alors
        Affiche Clé(T)
    Superieurs(FilsGauche(T), x)
    Superieurs(FilsDroit(T), x)
```

La seule façon de bénéficier de la structure de tas serait d'arrêter de tester Clé(T) > x pour les descendants des nœuds qui sont dans ce cas (on sait que leurs clés seront toutes supérieures à x). La complexité resterait linéaire en n, il s'agit d'un parcours d'arbre.

Question 4 : *(2 points)*

Pour chacun des deux algorithmes proposés, dites s'il fonctionnerait aussi sur un arbre binaire quelconque et expliquez pourquoi.

Réponse 4 :

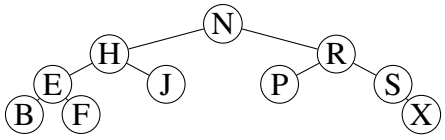
— Le premier algorithme est spécifique aux tas, par exemple sur l'arbre 

et avec $x = C$ il échouerait à afficher A et B.

— Comme dit, le second algorithme se contente d'effectuer un parcours et ne profite pas de la structure de tas, il fonctionnerait donc aussi sur un arbre quelconque.

2 Premiers éléments dans un arbre binaire de recherche (*11 points*)

Pour tous les exemples donnés dans l'énoncé, on utilisera l'arbre binaire de recherche :



Il n'est pas demandé de justifier la complexité des algorithmes écrits pour cet exercice.

Question 5 : *(2 points)*

Écrire un algorithme qui renvoie le plus petit élément dans un arbre binaire de recherche (contenant au moins un nœud).

Dans notre exemple, il faut renvoyer B.

Votre algorithme devra avoir une complexité au pire en $\mathcal{O}(h)$ et visiter le plus petit nombre de nœuds possible.

Réponse 5 :

Comme vu en cours :

```
Min(A)
Données : A : un arbre binaire de recherche
Résultat : le minimum de A
tant que non EstVide(FilsGauche(A)) faire
    A := FilsGauche(A)
Renvoyer Clé(A)
```

ou bien la version récursive :

```
Min(A, x)
Données : A : un arbre binaire de recherche
Résultat : le minimum de A
si EstVide(FilsGauche(A)) alors
    Renvoyer Clé(A)
sinon
    Renvoyer Min(FilsGauche(A))
```

Notons qu'on n'a pas besoin de traiter le cas où A est vide car l'arbre est supposé non vide, et ensuite lors des appels récursifs on s'assure que l'argument n'est pas vide.

Question 6 : *(5 points)*

Écrire un algorithme qui renvoie le 2e plus petit élément dans un arbre binaire de recherche (contenant au moins 2 nœuds).

Dans notre exemple, il faut renvoyer E.

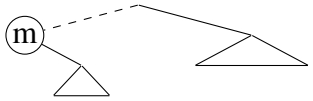
Représentez tous les cas possibles par des schémas et assurez-vous que votre algorithme les traite correctement.

Votre algorithme devra toujours avoir une complexité au pire en $\mathcal{O}(h)$ et visiter le plus petit nombre de nœuds possible.

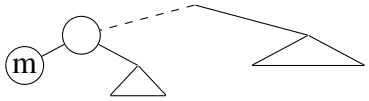
Réponse 6 :

Il y a principalement deux cas à envisager :

Le minimum de l'arbre a un fils droit, et alors le 2è élément est dans ce fils.



Le minimum de l'arbre n'a pas de un fils droit, et alors le 2è élément est le nœud parent de ce minimum.



Le plus simple est probablement d'écrire un algorithme itératif :

```
Min2(A)
Données : A : un arbre binaire de recherche
Résultat : le second minimum de A
tant que non EstVide(FilsGauche(A)) faire
    P := A // P sert à mémoriser le père du noeud courant A
    A := FilsGauche(A)
si EstVide(FilsDroit(A)) alors
    Renvoyer Clé(P)
sinon
    Renvoyer Min(FilsDroit(A))
```

(Dans le cas où on renvoie Clé(P), on a la garantie que P a été initialisé, sinon l'arbre initial n'avait qu'un seul nœud.)

Une version récursive est également possible :

```
Min2(A)
Données : A : un arbre binaire de recherche
Résultat : le second minimum de A
si EstVide(FilsGauche(A)) alors
    Renvoyer Min(FilsDroit(A))
sinon
    si EstVide(FilsGauche(FilsGauche(A))) et EstVide(FilsDroit(FilsGauche(A))) alors
        Renvoyer Clé(A)
    sinon
        Renvoyer Min2(FilsGauche(A))
```

Question 7 : *(4 points)*

Écrire un algorithme qui, étant donné un entier k (inférieur ou égal à n), renvoie le k-ième plus petit élément dans un arbre binaire de recherche.

Dans notre exemple, pour $k = 5$, il faut renvoyer J.

Vous pouvez pour cet algorithme utiliser une structure auxiliaire comme un tableau, une file, une pile... L'usage de variables globales est autorisé, par contre évitez à tout prix les `break` et autres instructions similaires.

Un algorithme qui n'utilise pas de variable globale rapportera plus de points.

Votre algorithme devra avoir une complexité au pire en $\mathcal{O}(n)$.

Réponse 7 :

Ici on ne cherche pas la complication, la complexité demandée nous autorise à faire un parcours exhaustif.

On se contentera donc de parcourir l'arbre en ordre infixe (de sorte à récupérer ses éléments en ordre croissant), puis de sélectionner le k-ième élément rencontré.

```
i : un entier, variable globale
T : un tableau de taille n, variable globale
i := 0
def ParcoursInfixe(A) :
    si non EstVide(A) alors
        ParcoursInfixe(FilsGauche(A))
        T[i] := Clé(A)
        i := i + 1
        ParcoursInfixe(FilsDroit(A))
def Kieme(A, k) :
    ParcoursInfixe(A)
    Renvoyer T[k-1]
```

Question 8 :

Cette question est difficile, et sera valorisée en points de bonus uniquement sous condition d'avoir déjà traité sérieusement le reste de l'énoncé.

Écrire un **autre algorithme** qui résout le **même problème** (renvoyer le k-ième plus petit élément dans un arbre binaire de recherche), mais cette fois sans utiliser de structure auxiliaire, ni modifier l'arbre.

Votre algorithme devra avoir une complexité au pire en $\mathcal{O}(h + k)$ et visiter le plus petit nombre de nœuds possible.

Indication : il est recommandé d'ajouter des paramètres supplémentaires à votre procédure et/ou de renvoyer un couple de valeurs bien choisies.

Réponse 8 :

- on passe en argument le rang k de l'élément recherché
- on renvoie un couple (v, i) :
  - si $i = k$ alors v est la valeur du k-ième élément recherché ;

— sinon la valeur de v n'a pas d'importance (on écrira '–' par convention), et i est le nombre de nœuds visités dans l'arbre.

On sait qu'on a atteint le k-ième élément de A quand la valeur i renvoyée vaut $k-1$; attention, dans les appels récursifs à droite, il faut modifier la valeur de l'argument pour tenir compte des nœuds vus à gauche.

On obtient donc :

```
Kieme(A, k) :
Données : A : un arbre binaire de recherche, k : un entier
Résultat : un couple : soit (v, k) où v est le k-ième élément de A ;
                       soit (–, i) où i < k est le nombre de nœuds dans A
1  si EstVide(A) alors
2      Renvoyer (–, 0)
3  sinon
4      (v, i) := Kieme(FilsGauche(A), k)
5      si i=k alors
6          Renvoyer (v, i)
7      sinon si i=k − 1 alors
8          Renvoyer (Clé(A), k)
9      sinon
10         Renvoyer Kieme(FilsDroit(A), k-i-1)
```

Schématiquement, notre algorithme :

— descend jusqu'au nœud minimal (pour un coût $\mathcal{O}(h)$) ;
— puis commence un parcours infixe de l'arbre en comptant le nombre de nœuds visités ;
— interrompt ce parcours dès qu'il a visité k nœuds, d'où un coût $\mathcal{O}(k)$ pour ce début de parcours ;
— il ne reste alors plus qu'à remonter jusqu'à la racine (c'est le rôle des **return** lignes 6 et 10), à nouveau pour un coût $\mathcal{O}(h)$.

Au total le coût est bien en $\mathcal{O}(h+k)$.