UNIVERSITÉ BORDEAUX 1 Sciences Technologies

Master BioInformatique

Année : 2012/2013 — Semestre de décembre 2012

PARCOURS : Master 1
UE J1BS7202 : Algorithmique et Programmation
Épreuve : Examen
Date : Lundi 17 décembre 2012
Heure : 10 heures
Durée : 2 heures
Documents : autorisés
Épreuve de M. Alain Griffault

SUJET + CORRIGE

Avertissement

- La plupart des questions sont indépendantes.
- Vous pouvez au choix utiliser un *langage algorithmique* ou bien le *langage python* pour répondre aux questions.
- L'espace laissé pour les réponses est suffisant (sauf si vous utilisez ces feuilles comme brouillon, ce qui est fortement déconseillé).

Question	Points	Score
ABR : algorithmes et complexités	20	
Total:	20	

Exercice 1 : ABR : algorithmes et complexités (20 points)

Rappels : Les Arbres Binaires de Recherche (ABR) sont des arbres binaires qui satisfont la propriété suivante : $\forall N$ un nœud de l'arbre, $\forall G$ un nœud du sous arbre gauche de N, $\forall D$ un nœud du sous arbre droit de N, $G.info \leq N.info \leq D.info$, où $info$ est une valeur entière servant à comparer les nœuds.

Voici pour mémoire une version python de deux algorithmes vus en cours sur les ABR.

```python
def minimumABR(A):
    if A==None:
        return None
    else:
        Aux = A
        while Aux.fg!=None:
            Aux = Aux.fg
        return Aux
```

```python
def successeurABR(A,P):
    # P doit etre un noeud non nil de A
    if P.fd!=None:
        return minimumABR(P.fd)
    else:
        Aux = P
        AuxPere = P.pere
        while AuxPere!=None and AuxPere.fd==Aux:
            Aux = AuxPere
            AuxPere = Aux.pere
        return AuxPere
```

(a) (2 points) En vous inspirant de la fonction `minimumABR(A)` vue en cours, écrire une fonction itérative `maximumABR(A)` qui retourne un pointeur sur le nœud de l'arbre `A` possédant la plus grande valeur entière s'il existe, la valeur `nil` sinon.

Solution:

```python
def maximumABR(A):
    if A==None:
        return None
    else:
        Aux = A
        while Aux.fd!=None:
            Aux = Aux.fd
        return Aux
```

(b) (2 points) Donner et justifier les complexités (pire des cas et meilleur des cas) de votre fonction `maximumABR(A)` en fonction du nombre de nœuds n ou de la hauteur h de l'arbre `A`.

Solution:
- Meilleur des cas : $\Omega(1)$ lorsque `A` n'a pas de fils droit.
- Pire des cas : $(\mathcal{O})(\backslash)$ lorsque `A` n'a qu'une descendance par ses fils droits. La hauteur h de l'arbre est alors égale à n.

(c) (3 points) En vous inspirant de la fonction `successeurABR(A,P)` vue en cours, écrire une fonction `predecesseur(A,P)` qui retourne un pointeur sur le nœud possédant la valeur qui précède la valeur contenue dans `P`.

Solution:

```
def predecesseurABR(A,P):
    # P doit etre un noeud non nil de A
    if P.fg!=None:
        return maximumABR(P.fg)
    else:
        Aux = P
        AuxPere = P.pere
        while AuxPere!=None and AuxPere.fg==Aux:
            Aux = AuxPere
            AuxPere = Aux.pere
        return AuxPere
```

(d) (2 points) Donner et justifier les complexités (pire des cas et meilleur des cas) de votre fonction `predecesseurABR(A,P)` en fonction du nombre de nœuds n ou de la hauteur h de l'arbre `A`.

Solution:
- Meilleur des cas : $\Omega(1)$ lorsque `P` a un fils gauche qui n'a pas de fils droit.
- Pire des cas : $(\mathcal{O})(\backslash)$ lorsque `P` est la racine, qu'il n'a pas de fils droit, qu'il a un fils gauche qui n'a qu'une descendance par ses fils droits. La hauteur h de l'arbre est alors égale à n.

(e) (3 points) En utilisant les fonctions `minimumABR(A)` et `successeurABR(A,P)` vues en cours, écrire une fonction itérative `afficherCroissantABR(A)` qui affiche les valeurs de l'arbre binaire de recherche `A` en ordre croissant.

Solution:

```
def afficherCroissantABR(A):
    if A!=None:
        Aux = minimumABR(A)
        while Aux!=None:
            print(Aux.info)
            Aux = successeurABR(A, Aux)
```

(f) (2 points) Donner et justifier les complexités (pire des cas et meilleur des cas) de votre fonction `afficherCroissantABR(A)` en fonction du nombre de nœuds n ou de la hauteur h de l'arbre `A`.

Solution:
- Meilleur des cas : $\Omega(n)$.
- Pire des cas : $(\mathcal{O})(\backslash)$.
- Soit : $Theta(n)$

(g) (2 points) En utilisant les fonctions `maximumABR(A)` et `predecesseurABR(A,P)` vues en cours, écrire une fonction itérative `afficherDecroissantABR(A)` qui affiche les valeurs de l'arbre binaire de recherche `A` en ordre décroissant.

Solution:

```
def afficherDecroissantABR(A):
    if A!=None:
        Aux = maximumABR(A)
        while Aux!=None:
```

```
            print(Aux.info)
            Aux = predecesseurABR(A,Aux)
```

(h) (2 points) Écrire une fonction récursive `afficherDecroissantRecursifABR(A)` qui affiche les valeurs de l'arbre binaire de recherche `A` en ordre décroissant.

Solution:

```
def afficherDecroissantRecursifABR(A):
    if A!=None:
        afficherDecroissantRecursifABR(A.fd):
        print(A.info)
        afficherDecroissantRecursifABR(A.fg):
```

(i) (2 points) Soit `A` un ABR contenant au moins deux nœuds `P1` et `P2`, et les deux séquences de calculs :

```
B1 = supprimerABR(A,P1)          B2 = supprimerABR(A,P2)
B1 = supprimerABR(B1,P2)         B2 = supprimerABR(B2,P1)
```

A-t-on toujours `B1 = B2` après ces calculs ? Si oui, justifier. Si non, donner un exemple.

Solution: Non.