ENSEIRB - Première Année Filière Informatique Année 2006-20007

Structures arborescentes

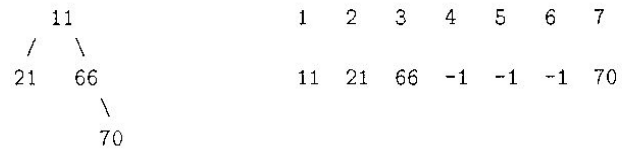
Durée : 2h.
Notes de cours et de TD autorisées.

La complexité en temps (resp. espace) désigne la complexité en temps (resp. espace) dans le pire des cas.

Exercice 1 On souhaite écrire un algorithme qui décide si une expression est bien parenthésée. Les expressions considérées sont des séquences contenant pour seuls caractères (,), {, }, [et]. Par exemple, les expressions `(([])])` et `(([]` sont mal parenthésées alors que les expressions `(([]([])))` et `(([]))[]` sont bien parenthésées.

1. Choisissez le type abstrait le plus approprié pour représenter cette séquence.
2. Ecrire un algorithme de complexité en temps linéaire et en espace constant qui décide si l'expression est bien parenthésée.

Exercice 2 Une façon simple de représenter un arbre binaire d'entiers positifs ou nuls est d'utiliser un tableau d'entiers où l'entier contenu à la racine est placée à l'indice 1 et tel que si un noeud est placé à l'indice i son fils gauche (resp. droit) est placé à l'indice $2i$ (resp. $2i+1$). Ainsi par exemple l'arbre binaire ci-dessous est représenté par le tableau ci-dessous.



1	2	3	4	5	6	7
11	21	66	-1	-1	-1	70

1. Expliquer pourquoi cette représentation n'est pas satisfaisante dans le cas d'arbres binaires quelconques.
2. Qu'en est il dans le cas d'arbres équilibrés ?

Exercice 3 On souhaite implémenter le type abstrait `sequence` suivant :

```
estVide sequence -> booléen
lire    sequence × rang -> entier
ecrire  sequence × element × rang -> sequence
inserer sequence × element × rang -> sequence
```

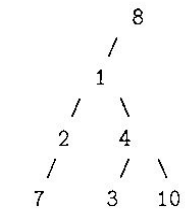
Nous rappelons les égalités `ecrire((10,20,30),21,2)=(10,21,30)` et `inserer(10,20,30),21,2)=(10,21,20,30)`.

1. Rappeler les deux principales façons d'implémenter un tel type.
2. Préciser chaque fois la complexité en temps de chacune des primitives.

3. Décrire rapidement comment on peut implémenter le type sequence en utilisant un abre binaire T et obtenir une complexité en temps pour chaque primitive égale à $O(hauteur(T))$. Indication : on devra pour tout noeud savoir calculer le nombre de noeuds qui lui descendants (fonction `nNoeudsDesc`). Pour cela, vous indiquerez comment représenter la séquence $(11, 2, 56, 78)$.
4. Ecrire l'algorithme `lire`.
5. Ecrire l'algorithme `inserer`.
6. Indiquer en quelques lignes comment affiner cette implémentation de façon à avoir des complexités en temps égales à $\theta(\log(n))$ où n est la longueur de la séquence.

Exercice 4 On considère ici des arbres binaires T composés de noeuds. L'*ancêtre commun* à deux noeuds x et y, noté $a(x, y)$ est l'ancêtre commun à x et y le plus éloigné de la racine. Rappelant que x est ancêtre de lui même, l'ancêtre de x et x est x lui-même.

La *distance* entre x et un de ses ancêtres y est le nombre d'ancêtres compris entre x et y sans compter x. La distance entre deux noeuds x et y est la distance de x à l'ancêtre de x et y plus la distance entre cet ancêtre commun et y. La distance est notée $d_T(x, t)$. De façon plus concise, la distance entre deux noeuds x et y est le "plus petit" nombre de liaisons père-fils entre x et y.



Dans l'exemple précédent la distance entre les noeuds 7 et 10 est 4.

On suppose que l'on puisse à partir d'un noeud x accéder notamment en temps constant à son fils gauche, son fils droit et à son père. On suppose en outre que chaque noeud est muni de champs supplémentaires de types booléens, entiers que l'on peut lire et modifier à souhait.

1. Ecrire un algorithme qui calcule pour tout arbre T et pour tout couple de noeuds x et y la distance entre x et y. La complexité en temps de de votre algorithme sera $O(max(d_T(x, r), d_T(y, r)))$ où r_T est la racine de T. Si lors de votre calcul, vous avez besoin de mémoire auxiliaire et utilisez des champs des noeuds parcourus, vous nommerez judicieusement ces champs et indiquerez leur contenu.

   Le *diamêtre* de l'arbre T est la plus grande distance entre deux noeuds de T.
2. Calculer le diamètre de l'arbre dessiné ci-dessus ainsi que celui son sous-arbre de racine 1 (resp. 2 et 4).
3. Ecrire un premier algorithme naïf calculant le *diamêtre* de T. Evaluer sa complexité en temps.
4. Ecrire un second algorithme récursif calculant le *diamêtre* de T et de complexité en temps optimale ! Fournir la définition récursive de la complexité en temps et évaluer la.