



Master BioInformatique

ANNÉE : 2010/2011

SESSION DE DÉCEMBRE 2010

PARCOURS : Master 1

UE : Algorithmes et structures de données

Épreuve : Examen

Date : Vendredi 17 décembre 2010

Heure : 14 heures

Durée : 2 heures

Documents : autorisés

Épreuve de M. Alain GRIFFAULT

Code d'anonymat :

Avertissement

- La plupart des questions sont indépendantes.
- Le barème total est de 24 points car le sujet est assez long.
- (*) indique une question plus difficile.
- L'espace laissé pour les réponses est suffisant (sauf si vous utilisez ces feuilles comme brouillon, ce qui est fortement déconseillé).

Rappels

Définition 1 *Un arbre binaire est une structure dynamique A récursive qui :*

- soit ne contient aucun nœud.
- soit contient trois ensembles disjoints de nœuds :
 - un nœud racine,
 - un arbre binaire dénommé sous arbre gauche.
 - un arbre binaire dénommé sous arbre droit.

Définition 2 *Un arbre binaire A de hauteur h est presque parfait lorsque :*

- Toute les feuilles sont à distances h ou h – 1 de la racine.
- La feuille la plus à gauche est à distance h de la racine.
- Si une feuille est à distance h – 1 de la racine, alors toutes les feuilles à sa droite sont à distance h – 1 de la racine.

En cours, vous avez vu la correspondance entre les arbres binaires presque parfait de taille N, et les tableaux indicés de 0 à N-1 munis des fonctions :

- $fg(i) = 2*i + 1$ qui simule le pointeur fg.
- $fd(i) = 2*i + 2$ qui simule le pointeur fd.
- $pere(i) = (i - 1)/2$ pour $i > 0$ et $pere(0)=0$ qui simule le pointeur pere.

Dans la suite vous supposerez disposer des opérateurs suivants sur les tableaux :

- **Etendre(T)** qui ajoute une “case” à la fin du tableau T, et qui met à jour la valeur de T.longueur en conséquence, de complexité $O(1)$.
- **Reduire(T)** qui retire la “dernière case” du tableau T, et qui met à jour la valeur de T.longueur en conséquence, de complexité $O(1)$.

Vous supposerez également que chaque “case” d’un tableau contient :

- un champ **data** pour stocker les données.
- un champ **cle** pour ordonner les données.

Définition 3 *Un tas T de taille N est un tableau de longueur N tel que :*

$$\forall i \in [0, N - 1], T[pere(i)].cle \geq T[i].cle$$

La structure de tas est munie des primitives suivantes vues en cours :

- **EntasserRécursif(T, i, M)** qui construit un tas de taille M dans un tableau de longueur $T.longueur \geq M$ à partir d’une structure qui est un tas sauf peut-être pour la “case” d’indice i.

- ExtraireMax(T) qui supprime d'un tas l'élément ayant la plus grande clé.
 - InsérerEchange(T, X) qui ajoute au tas T une nouvelle case contenant X.data, X.cle et positionnée en fonction de X.cle.
- dont voici les pseudo codes.

```

EntasserRecuratif(T, i, M) {
// M la taille du tas est un nombre inferieur ou egal a la longueur du tableau
M <- Minimum(M, T.longueur); // pour eviter des debordements
indiceMax <- i;
si ((fg(i) < M) && (T[fg(i)].cle < T[indiceMax].cle)) alors {
    indiceMax <- fg(i);
}
si ((fd(i) < M) && (T[fd(i)].cle < T[indiceMax].cle)) alors {
    indiceMax <- fd(i);
}
si (indiceMax != i) alors {
    Echanger(T[i], T[indiceMax]);
    EntasserRecuratif(T, indiceMax, M);
}
}

ExtraireMax(T) {
    si (T.longueur > 0) alors {
        max <- T[0];
        T[0] <- T[T.longueur - 1];
        Reduire(T);
        Entasser(T, 0);
    }
    sinon {
        Ecrire "Erreur, le tas est vide";
    }
}

InsérerEchange(T, X) {
    Etendre(T);
    i <- T.longueur - 1;
    T[i] <- X;
    tant que (i != pere(i) && T[pere(i)].cle < T[i].cle) faire {
        Echanger(T[i], T[pere(i)]);
        i <- pere(i);
    }
}

```

Exercice 1 (Parcours avec file des arbres binaires (8 points))

Il est possible de parcourir un arbre binaire en utilisant une file comme structure de stockage.

```

ParcoursFile(A){
    F <- CreerFileVide();
    si (A != nil) alors {
        Enfiler(F, A);
    }
    tant que (non FileVide(F)) faire {
        Aux <- TeteFile(F);
        Afficher(Aux.cle);
        Defiler(F);
        si (Aux.fg != nil) alors {
            Enfiler(F, Aux.fg);
        }
        si (Aux.fd != nil) alors {
            Enfiler(F, Aux.fd);
        }
    }
}

```

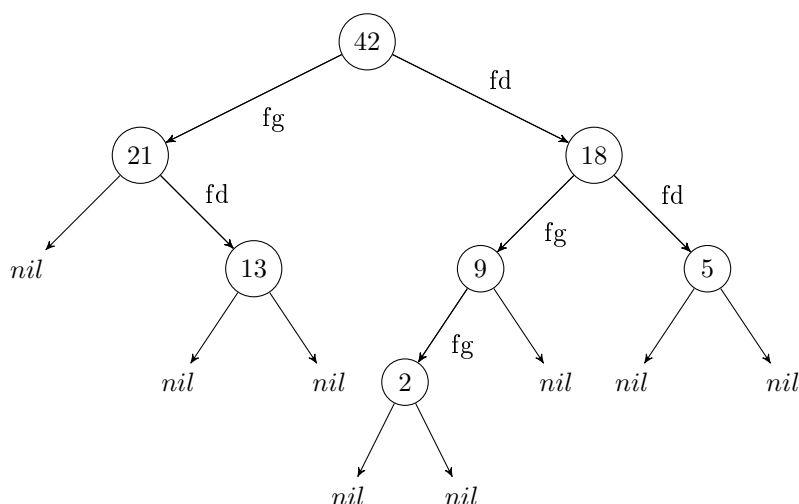


FIGURE 1 – Un arbre binaire et les clés des nœuds

Question 1.1 (4 points) Complétez l'exécution de l'algorithme `ParcoursFile(A)` sur l'arbre binaire de la figure 1.

<i>F</i>		()	(42)
<i>non FileVide(F)</i>			
<i>Aux</i>			
<i>Affichage</i>			
<i>F</i>			
<i>non FileVide(F)</i>			
<i>Aux</i>			
<i>Affichage</i>			
<i>F</i>			
<i>non FileVide(F)</i>			
<i>Aux</i>			
<i>Affichage</i>			

Question 1.2 (2 points) Dans le cas où votre structure de tas serait implémentée avec des pointeurs et non avec un tableau, quel serait le résultat de l'algorithme `ParcoursFile(T)` si *T* était un tas trié.

Question 1.3 (2 points) Si le résultat de l'algorithme `ParcoursFile(A)`, où *A* est un arbre binaire, est la liste ordonnée décroissante des clés, peut-on conclure que *A* est un tas. Vous justifierez votre réponse.

Exercice 2 (Une structure de Tas plus complète (16 points))

L'objectif de l'exercice est d'optimiser légèrement deux de ces primitives et d'ajouter la suivante :

- `ModifierCle(T, i, k)` qui remplace la clé de `T[i]` par une valeur `k` et maintient une structure de tas.

Question 2.1 (3 points) Donnez et justifiez les complexités dans le pire des cas pour les trois primitives `EntasserRecuratif(T, i, M)`, `ExtraireMax(T)` et `InsererEchange(T, X)`.

Question 2.2 (3 points (*)) L'algorithme `InsererEchange(T, X)` utilise l'algorithme `Echanger(A, B)` qui coûte 3 affectations à chaque appel. En vous inspirant d'une technique utilisée en cours pour remplacer l'usage d'échanges par une technique de "décalage", écrivez une primitive `InsererDecalage(T, X)` produisant le même effet. Vous donnerez la complexité dans le pire des cas de votre solution.

Question 2.3 (4 points (*)) L'algorithme `EntasserRecuratif(T, i, M)` utilise la récursivité. Écrivez une version itérative `EntasserIteratif(T, i, M)` produisant le même effet. Vous donnerez la complexité dans le pire des cas de votre solution.

Question 2.4 (2 points) Écrivez un algorithme `DiminuerCle(T, i, k)` qui remplace la clé de l'élément `T[i]` par la plus petite des deux valeurs entre `T[i].cle` et `k`, tout en maintenant une structure de tas. Vous donnerez la complexité dans le pire des cas de votre solution.

Question 2.5 (2 points) Écrivez un algorithme `AugmenterCle(T, i, k)` qui remplace la clé de l'élément `T[i]` par la plus grande des deux valeurs entre `T[i].cle` et `k`, tout en maintenant une structure de tas. Vous donnerez la complexité dans le pire des cas de votre solution.

Question 2.6 (2 points) Écrivez un algorithme `ModiferCle(T, i, k)` qui remplace la clé de l'élément `T[i]` par `k`, tout en maintenant une structure de tas. Vous donnerez la complexité dans le pire des cas de votre solution.